ECNG2006 (EE25M) Introduction to microprocessors

Course Notes

Contributors:
K. Hall
F. Mohammed
C. Radix
A. Sinanan
A. Williams
K. Edwards
A. Joseph
O. Regalado
K. Narine
I. Mohammed
Y. Panchu
S. Patrick
A. Abdool
C. Arneaud
N. Harrichand
H. Lawrence
D. Caberrea
P. Pollucksingh

March 11, 2008

Contents

References

Dictionary.com. Internet. Http://dictionary.reference.com/.

Mathworld. Internet. Http://mathworld.wolfram.com/.

1991. Can specification version 2.0. Http://www.interfacebus.com/CAN_2_Spec.pdf.

1993. The free on-line dictionary of computing. Internet. Http://www.foldoc.org/.

1994-2000. PCW Compiler Help files.

1995. 80C51 family architecture. `http://www.semiconductors.philips.com/acrobat/various/80C51_FAM_ARCH` `http://www.grifo.com/press/DOC/Philips/FAMARCH.PDF`.

1997. PICmicro Mid-Range MCU Family Reference Manual. Technical Reference Document 33023a, MicroChip Technology Inc.

1999. HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver). Datasheet, Hitachi.

1999. MPASM USERS GUIDE with MPLINK and MPLIB. Manual DS33014G, MicroChip Technology Inc.

2000. MPLAB ICD USERS GUIDE. Manual DS51184D, Microchip Technology Inc.

2000. MPLAB IDE, SIMULATOR, EDITOR USERS GUIDE. Manual DS51025D, Microchip Technology Inc.

2000. The I2C-bus specification. http://www.semiconductors.philips.com/acrobat/various/I2C_BUS_SPECIFICATION_3.pdf.

2001. At90s8535 summary. Available at www.atmel.com. Rev. 1041HS - 11/01.

2001. Atis telecom glossary 2000. Internet. Http://www.atis.org/tg2k/t1g2k.html.

2001. PIC16F87X Data Sheet: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. Technical Reference Document 30292c, MicroChip Technology Inc.

Arnold, Ken. 2001. *Embedded controller hardware design.* LLH Technology Publishing.

Awtrey, Dan. 1997. Transmitting data and power over a one-wire bus. *Sensors* Reprint downloaded from www.dalsemi.com.

Ball, Stuart R. 2000. *Embedded microprocessor systems.* Butterworth-Heinemann.

Barr, Michael. 1999. Choosing a compiler: The little things. WebPage: http://www.netrino.com/Articles/CrossCompilers/.

———. 2000. Language lessons. WebPage: http://www.netrino.com/Connecting/2000-03/index.html.

Buchanan, William, and Austin Wilson. 2001. *Advanced pc architecture.* Addison-Wesley.

Cady, Fredrick M. 1997. *Microcontrollers and microcomputers: Principles of software and hardware engineering.* Oxford University Press, Inc.

Fisher, Matt. 2000. Protecting binary executables. Webpage: http://embedded.com/2000/0002/0002feat1.htm.

Hartman, Hope J. 2001. *Developing students' meta-cognitive knowledge and skills*, chap. 3, 33–68. Kluwer Academic Publishers.

Horowitz, Paul, and Winfield Hill. 1989. *The art of electronics.* 2nd ed. Cambridge University Press.

Katzen, Sid. 2003. *The quintessential pic microcontroller.* Springer. 2nd printing.

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language Second Edition.* Prentice-Hall, Inc.

Mazidi, Muhammed Ali, and Janice Gillespie Mazidi. 1995. *The 80x86 ibm pc & compatible computers volumes i & ii: Assembly language design and interfacing.*

Mitra, Sumit. 1997. Power-up considerations. Application Note AN522, DS00522E, Microchip Technology Inc.

Noergaard, Tammy. 2005. *Embedded systems architecture.* Newnes.

Palmer, Mark. 1997a. An520: A comparison of 8-bit microcontrollers. Application Note DS00520D, Microchip Technology Inc.

———. 1997b. Power-up trouble shooting. Application Note AN607, DS00607B, Microchip Technology Inc. Contributions: Richard Hull & Randy Yach.

Peri, Vamsi, and Dan Simon. 2005. Fuzzy logic control for an autonomous robot. In *Annual meeting of the north american fuzzy information processing society (nafips 2005) 26-28 june 2005*, 337–342. Downloaded from http://academic.csuohio.edu/simond/pubs/Vamsi.pdf on March 7th 2007.

Predko, Myke. 2000a. *Debugging your applications*, chap. 12, 561–569. In Predko (2000d).

———. 2000b. *Designing your own picmicro®mcu application*, chap. 11, 547–560. In Predko (2000d).

———. 2000c. *Picmicro®mcu application design and hardware interfacing*, chap. 6, 253–316. In Predko (2000d).

———. 2000d. *Programming and customizing picmicro®microcontrollers.* McGraw-Hill.

———. 2001. *Picmicro®microcontroller pocket reference.* McGraw-Hill.

Stallings, William. 1996. *Computer architecture and organization.* 4th ed. Prentice-Hall, Inc.

———. 2000. *Computer architecture and organization.* 5th ed. Prentice-Hall, Inc.

Vahid, Frank, and Tony Givargis. 2002. *Embedded system design.* John Wiley & Sons, Inc.

Wakerly, J. F. *Digital design principles and practices.* 3rd ed. Prentice-Hall.

Wilmhurst, Tim. 2001. *An introduction to the design of small-scale embedded systems*. Palgrave.

Wolf, Wayne. 2000. *Computers as components:principles of embedded computing system design*. Morgan Kaufmann.

ECNG2006(EE25M) –
Introduction to Microprocessors
The University of the West Indies
Department of Electrical and Computer Engineering
Last Review: December 2, 2005

Course Instructor:
Name: Ajay Joshi
Email: ajoshi@eng.uwi.tt
Phone: ext. 3144
Office: Room 325
Office Hours: as posted on door

Course Support
Name: Aaron Joseph
Email: Given at Later Date
Phone: ext. 3156
Office: Room 329
Office Hours: Tuesday and Thursday 1pm-3pm

Aims
To cultivate the student's ability to utilize, comprehend the operation, and criticize the efficacy of, a microprocessor in an applied context.

Objectives
At the end of this course the student will be able to:

- in general, or (given relevant diagrams and specifications) for a particular system, identify, and describe the role of, the components of
    (a) a microprocessor,
    (b) a microprocessor-based system,
    (c) a development suite (hardware, software) for a microprocessor-based system.
- illustrate how data can be represented in (and accessed from) memory, implement arithmetic and data manipulation operations in assembly language, and assess how well code will perform in a given context, given the architecture and instruction set of a microprocessor
- design and implement an appropriate interface between a microprocessor-based system and a peripheral device (or another microprocessor-based system), given relevant datasheets and suitable parts
- select (and critique the selection of) a microprocessor-based system for an application, given relevant datasheets and application requirements

Course Overview

Microprocessors have been one of the most widely used methods of incorporating flexibility and intelligence into automated devices. Their general-purpose nature, speed and size have made them one of the most common components in Electrical Engineering. It is therefore necessary to develop a good understanding of their operation and how they can be used as building blocks for automated systems and control applications.

This course explores the inner workings of a microprocessor from the programmer's perspective, as well as treating with external hardware issues such as interfacing, and selection criteria for microprocessors. Exercises and examples are based on the PIC 16F877 microcontroller. The syllabus follows:

> Microprocessor architecture (PIC16F877); Microprocessor development and support systems(MPLAB); Binary, integer and floating point arithmetic operations; (PIC16Cxxx) assembly language programming; Interfacing (PIC16F877): I/O ports, Timers, Interrupts, A/D conversion, PWM; System Issues; Serial/Parallel Communication.

Prerequisites
There are no departmental pre-requisites, however, it is presumed that the student has acquired the necessary pre-requisite skills in the preceding electronics and computer systems courses. It is presumed that the student is able to:
- correctly interpret syntax of simple C programs, representation of different numeric bases in C, and utilize an ANSI C compiler/IDE.
- treat with the concept of numeric bases, and perform translation between binary, decimal and hexadecimal bases.
- predict the operation of simple digital/analog circuit components, in particular resistors, capacitors, latches and operational amplifiers.
- interpret pin-out diagrams, functional block diagrams, and timing diagrams as found on datasheets.
- effectively use basic lab equipment: meters, oscilloscopes, power supplies, breadboards.
- troubleshoot basic signal problems in a circuit (i.e. no/low voltage, intermittent signals, different grounds, noise).

Weighting
3 credits
requires 12 teaching weeks; week 13 activities/mock exam not compulsory
each teaching week: 3 hrs lecture; 3 - 6 hrs self-study; 1 hr tutorial
additional time for group project 6-9 hours.
associated labs in ECNG2005 (EE24D):
6 labs @ 3hrs/lab; individual project @ 6 hours

Evaluation

Type of Evaluation	**Description**	**% of Course Grade**
End of Semester Exam:	3 hours – compulsory pass Short answer/design (cover major learning objectives)	60%
In-semester Exam:	Mid-term exams in Wk 6 and 12 @ 6% Mock Exam in Wk 13; not assessed	12%
Laboratory:	Lab exercises & individual project contribute to grade in ECNG2005 (EE24D)	0%
Mini-Project:	Group lab exercise (5-6 people per group)	6%
Research Paper:	Reading Assignment in Wk 12	2%
Other:	Online Tutorials (10 %) 5 Assignments (2% each) Optional participatory activities (1% each contribution; total not exceeding 4%; category total not exceeding 20%)	20%
TOTAL:		100%

2007/08 Semester II Calendar

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Wk 1	14 Jan	15	16	17	18	19	20
Wk 2	21	22	23	24	25	26	27
Wk 3	28	29	30	31	1 Feb	2	3
Wk 4	4-Carnival	5- Carnival	6	7	8	9	10
Wk 5	11	12	13	14	15	16	17
Wk 6	18	19	20	21	22	23	24
	25	26	27	28	29	1 Mar	2
Wk 7	3	4	5	6	7	8	9
Wk 8	10	11	12	13	14	15	16
Wk 9	17	18	19	20	21-Good Friday	22	23
Wk 10	24-Easter Monday	25	26	27	28	29	30-Baptist Liberation
Wk 11	31 Bap Lib Holiday	1 Apr	2	3	4	5	6
Wk 12	7	8	9	10	11	12	13
Wk 13	14	15	16	17	18	19	20
Exams	21	22	23	24	25	26	27
Exams	28	29	30	1 May	2	3	4
Exams	5	6	7	8	9	10	11
Exams	12	13	14				

Coursework

Laboratory Exercises:	Marks towards ECNG2005 (EE24D)
Mini Project:	Group must choose one of the assigned projects 4-6 people per group w/up to 5 groups per topic Marks: Design 2%; Implementation 2%; Report 1%; Demo/Presentation 1% Weighted by individual contribution to group

Assignments

Research Paper/s:	Submit an informative abstract about an article chosen/assigned from IEEE Magazines 2002-7. Item should be a 5-7 page article on a topic of relevance to microprocessor-based systems.
Presentation/s:	Associated with the Group Project – 1 minute
Problem Set/s:	5 Assignments; written, 2 weeks to submit
Other:	tutorial questions online; 30 minute completion time

Resources

Textbooks:

Category A: Essential Texts

1) An introduction to the design of small-scale embedded systems, T. Wilmhurst.

Category B: Highly Recommended Texts

1) The Quintessential PIC Microcontroller, S. Katzen.
2) Embedded Systems Architecture, T. Noergard

Category C: Recommended Texts

1) Programming and customising PICmicro microcontrollers, M. Predko (2nd Edition).
2) Microcontrollers and microcomputers, F. Cady.

General Reference:

1) Computer Organisation & Architecture, W. Stallings, (5th Edition or later)
2) Embedded System Design: A Unified Hardware/Software Introduction, F. Vahid, & T. Givargis
3) Logic and Computer Design Fundamentals, M. Mano & C. Kime
4) Digital Design: Principles & Practice by J. Wakerly, (3rd Edition or later)

Course Notes: will be distributed in class and/or made available online

On-Line Resources: Topical Videos: http://murl.microsoft.com/default.asp

Course site: http://www.eng.uwi.tt/depts/elec/staff/cradix/ee25m/

Study Tips: http://www.iss.stthomas.edu/studyguides

PIC Micro WebRing:

http://www.webring.org/cgi-bin/webring?ring=picmicro;list

Simulation Tools: (in microprocessor and shared labs) MPLAB/CCS compiler

Laboratory Tools: 20 seat lab.

Per seat: LED's, resistors, potentiometers, ICD header + RJ12 cable + ICD debug module + serial cable, Breadboard, wire and 5V power supply, 7 segment LCD display, transistors, LCD text display, DS1821 temperature sensor, additional serial cable, DC motor, stepper motor, encoder disc/IR led/detector, keypad.

Additional components for group projects.

Other: --none--

Lecture/Tutorial Schedule

Course has: 33 learning units; each unit 1 hr lecture + coursework

In addition to learning units there is:

1 hr overview + 1 hr mid-term + 1 hr mid-term

Tutorial activities include

4 hours topic review

2 hours guest lecture – informative abstract;

2 hours midterm review

4 hours group project work

No.	**Unit #: Topic**	**Notes/Delivery/Comments**[1]		**Text**[2]
		Associated C'work	Deadline	
1.	Course Overview	Survey		
2.	1: µP basics	Tutorial 1:a		Ch. 1
3.	2: µP system basics	Tutorial 1:b		Ch. 1/2/4
4.	3: µP support circuitry	Tutorial 2:a		Ch. 2/4
5.	4: µP architecture	Tutorial 2:b		Ch 1
6.	5: Typical instructions	Assignment 1		Ch. 3
7.	6: Software tool chain	Tutorial 3:a		Ch 3/7
8.	7: Development support	Tutorial 3:b		Ch 3/7
9.	8: Families & forms	Tutorial 4:a		Ch. 1
10.	9: PIC16F877 overview	Tutorial 4:b		Ch. 1/2/3
11.	10: MPLAB overview	Lab 1		Ch. 3
12.	11: Real numbers	Assignment 2		Ch. 11
13.	12: Integer arithmetic	Tutorial 5:a		Ch. 11
14.	13: Real number arithmetic	Tutorial 5:b		Ch. 11
15.	14: Data structures	Tutorial 6:a		Ch 3
16.	15: Basic operations	Lab 2		Ch 3
17.	16: Code comparison	Assignment 3		
18.	17: Compiler limitations	Tutorial 6:b		Ch 7
19.	18: Interfacing Idioms	Tutorial 7:a		Ch. 2/9
20.	19: Interrupt Basics	Tutorial 7:b		Ch. 3

[1] Whichever applicable

[2] Specify

No.	**Unit #: Topic**	**Notes/Delivery/Comments**[1]		**Text**[2]
21.	Mid-term			
22.	Midterm review+ Stud't Feedback	Grp. Project		
23.	Guest Lecture – H. Lawrence			
24.	Guest Lecture – H. Lawrence	Reading Assignment		
25.	Topic Review			
26.	Topic Review			
27.	20: Typical peripherals	Tutorial 8:a		Ch. 5/9
28.	21: Digital troubleshooting	Tutorial 8:b		Ch. 3
29.	22: PIC16 peripherals	Lab 3		Ch. 2/4/5
30.	Group Project			
31.	23: Interrupt issues	Assignment 4		Ch. 8
32.	24: µP interface/timing issues	Tutorial 9:a		Ch. 9
33.	25: Power/Signal issues	Tutorial 9:b		Ch. 10
34.	Group Project			
35.	26: µP choice/comparison	Tutorial 10:a		Ch. 12
36.	27: Case study	Tutorial 10:b		Ch. 12
37.	28: Design guidelines	Assignment 5		Ch. 12
38.	Group Project			
39.	29: Communication protocols	Tutorial 11:a		Ch. 6
40.	30: Bit-banging vs. hardware	Tutorial 11:b		Ch. 6
41.	31: I/O on the PC	Lab 4		Ch. 2
42.	Group Project			
43.	32: PC to µP serial link	Lab 5		Ch. 6
44.	33: µP Parallel Interface	Lab 6		Ch. 2
45.	Topic Review			
46.	Topic Review			
47.	Mid-Term			
48.	Stud't F'back;+ Midterm Review			
49.	Group Demos/Presentations			
50.	Mock Exam (3 hours; proctored)			

Unit objectives
At the end of this course the student will be able to:

Section I: µP Overview

1. identify the components of a typical microprocessor (ALU, registers, stack), describe the operation of the components of a typical microprocessor, and describe the representation of instructions in the operation of the instruction cycle.
2. identify the components of a microprocessor-based system (CPU, bus, memory/layout, peripherals), describe the operation of typical components of a microprocessor-based system, and discuss issues involved in the design/choice of the various components of the microprocessor(-based system)
3. describe (and/or illustrate) how bus addressing, conflict resolution and memory maps may be implemented, and explain the function of basic support circuitry for microprocessors such as clocks, reset circuitry, watch dog timer, capacitors to ground etc. .
4. differentiate between organization and architecture, and relate instruction set design to the programmer's model of the microprocessor,.
5. explain the operation of typical machine instructions, addressing modes, status bits, and infer the consequences of sequences of generic instructions.

Section II: µP development

6. explain the roles of the compiler, linker, assembler, simulator, emulator, debugger in the software development tool chain, and the role of scripting languages, make/configure-like utilities and IDE's in the software development process.
7. explain the role of firmware, operating systems, oscilloscopes, logic probes/analyzers, terminals, disk drives, external memory, debug protocols e.g. JTAG and host computers in the development and support of microprocessor based systems.
8. recognize and differentiate between the different commercially available families/forms of microprocessor based systems (Motorola, AVR, PIC – chip/component packages -- SOC, microcontroller, PC, back-plane bus, PC104, standalone, embedded) based on their pictures/properties/descriptions and supported development tools.

Section III: PIC16 Introduction

9. explain the memory layout, operation of the instruction cycle, and the basic instructions (move, add, subtract, shifts) for the PIC16F877 microcontroller.
10. utilize the MPLAB IDE , CCS compiler, and other software tools, to develop software, for the MicroChip PIC16xxx series of microcontroller, in C, C++ and assembly language.

Section IV: Numbers & Data

11. represent real numbers using fixed and floating point binary representations.
12. perform integer arithmetic operations using binary representations of the operands.
13. perform fixed/floating point arithmetic operations using binary representations of the operands.
14. describe how data (and data structures) can be represented and manipulated, with reference to alignment and byte/bit ordering issues.

Section V: Algorithms on PIC16

15. implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.
16. contrast alternate programming techniques in terms of size/speed of the code produced for the PIC16Cxxx series of microcontrollers.
17. give examples of the limitations placed on compiler-generated code, for the PIC16Cxxx series of microcontrollers, by the need for generality.

Section VI: Interfacing Peripherals

18. identify alternate ways in which components may be interfaced with a microprocessor (i.e. common idioms for interacting with hardware): single bit (with debounce), matrix, map into memory space, etc. .
19. explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.
20. explain the operation of commonly used peripherals: PPI/PIA, A/D, Timers/Counters, PWM, D/A.
21. select and apply techniques for troubleshooting digital circuitry using standard laboratory equipment.
22. utilize built-in peripherals of the PIC16F877 microcontroller in simple applications.

Section VII: System Issues

23. understand the implications of (nested/multiple) interrupts for execution times, and the need for context saving.
24. interpret and recognize the implications of, microprocessor timing and interfacing requirements.
25. determine the noise, voltage, current and power considerations due to microprocessors and peripheral fan-in/out and how different logic families (as well as inverted voltage logic) may be used in conjunction in a design.
26. compare alternate microprocessors in terms of the instruction set, and features(such as cache pipelining etc.) and the best performance, which may be facilitated by the instruction set and features.
27. recognize the issues and tradeoffs involved in the design of a microprocessor based (embedded) system, and justify the role of a microprocessor(-based system) in an applied context.
28. remember and follow a checklist of design guidelines, in order to perform a simple design for a microprocessor based system.

Section VIII: Communication

29. understand commonly used parallel and serial communication protocols (I2C, CAN, RS232, USB, Firewire) and interpret descriptions of proprietary protocols (Dallas 1-wire).
30. differentiate between bit-banging and dedicated hardware interfaces for serial/parallel communication (PPI/PIA vs ACIA/UART, PPC)
31. perform I/O programming on the PC in C/Assembly using the Visual C++ compiler
32. produce assembly language programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and PPC datasheets/details, and respective instruction sets.
33. interface external peripherals/devices to the PIC16F877 microcontroller using a parallel bus, and interface the PIC16F877 microcontroller as an external device on a parallel bus (memory mapped).

Course Rules: your rights and responsibilities
You have the right to receive a fair mark for any assessed coursework which you produce.

- Weighting of individual assignments will be issued at the beginning of the semester.
- Marking schemes will be issued with each piece of assessed coursework.
- You may request clarification of marking schemes at any time prior to submission.
- You should receive feedback (individual, general or both) on coursework within 1 week.
- You may query the mark assigned to your coursework, if it has been totaled incorrectly, or is not consistent with the published marking scheme.

You have the right to know the examination format, as well as the requirements for passing the final examination, and the course.

You are entitled to lectures, which start and finish on time. You are entitled to appropriate notice of lecture/lab cancellation.

You are entitled to have any concerns about the course heard. Individual queries should be addressed to the lecturer (or TA) in the first instance and the HoD in the next instance. Anonymous or group queries should be directed to the student representative or your tutor in the first instance, who is obliged to bring it to the attention of the lecturer and/or HoD.

You are responsible for checking posted examination schedules and arriving on time for exams with your examination and/or ID cards, and writing implements.

You are responsible for arriving to lectures/labs on time, and participating in all course activities.

You are responsible for submitting coursework by the posted deadline, and for securing a receipt (or signing a submission sheet) for all submitted coursework.

You are responsible for seeking assistance from or notifying staff if you any difficulties (academic, personal or health-related).

You will be held at fault if you do not comply with all university, departmental and faculty regulations, as well as with the course policy on collaboration. Ignorance is not a valid excuse.

Policies on plagiarism, collaboration and late submission

Late submissions may fall into one of two categories, late submissions for online coursework and late submission for written coursework. All online coursework will be due at 11:59PM on the submission date and all written coursework will be due at 4PM on the submission date.
For online coursework, once the submission time has passed, the coursework will no longer be accepted; and you will receive a grade of zero for the piece of coursework.
For written coursework, will be assessed in accordance with the published marking scheme, but 0.5% of your course mark will be deducted for each 24 hour period (or part thereof) a piece of coursework is late. The minimum mark awarded will be 0%. e.g. an assignment graded at 8 marks out of 10, which translates to 1.6% towards the course mark. If the tutorial was handed in at 8am (16 hours late), 0.5% would be deducted, leaving 1.1% towards the course mark.
Written coursework must be in the possession of the Course Lecturer or Support Team Leader to have been submitted.
The ONLY exception to this rule is if you have a valid medical/sick leave for the days in question, which you have submitted to the Electrical office/Registry; or if an unforeseeable national/personal emergency prevented you from getting here. In either case, I need to get formal notice from the Head.

"**Plagiarism** is using others' ideas and words without clearly acknowledging the source of that information" -- from *Plagiarism: What It is and How to Recognize and Avoid It*, http://www.indiana.edu/~wts/wts/plagiarism.html
Plagiarism can occur in many different ways:

- using someone else's words/work without quotes or acknowledgement
- using someone else's ideas without acknowledgement
- using information gathered or collected by someone else without acknowledgement

-- from *Plagiarism: What It is and How to Recognize and Avoid It*, http://www.indiana.edu/~wts/wts/plagiarism.html
``Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. [solutions,] published books or periodicals), without due acknowledgement in the text." -- EE26A policy on plagiarism

Collaboration on assignments is often helpful, and has been shown to increase understanding of all parties involved in the collaboration. It is OK to ask someone else if they think your work is correct, and if not, they can point out where you made a mistake, and give an explanation of the error. If you give (or are given) a worked example to improve understanding, the example should be as unrelated to the assignment as possible.
Inappropriate collaboration (for the purposes of this course) includes:

- for *individual or group* assignments allowing your assignment to be copied, or copying someone else's assignment, either electronically, manually or by photocopy. Ignorance of the copying is no excuse -- you are advised to secure all coursework material.

Coursework Portfolio

AIM: **to promote reflective learning of both students and lecturer**

Offline Version[1]

Students are asked to keep a manila folder with all coursework and feedback forms. Items should be fastened into the folder using a single hole and a treasury tag. Your ID# should be placed on the folder tab. Your name is not required.

You should use this folder to review your progress throughout the semester. The folder should be submitted to the lecturer after the final exam, who will use the information to determine appropriate course adjustments.

Please feel free to make frank observations, constructive criticism, and realistic suggestions in the feedback sections.

Feedback will not adversely affect your course grade. Any students who have reservations about how such feedback is being used may withhold feedback until after the coursework is marked, or omit feedback entirely.

Please start your folder with the Course Perceptions and Skills Questionnaires within the first 2 weeks of the semester.

Online Version

Students are asked to keep a Wiki containing scans of/reflections on all coursework and feedback forms. You should use this Wiki to review your progress throughout the semester. The Wiki will be viewed by the lecturer, who will use the information to determine appropriate course adjustments, and offer correction or advice.

Please feel free to make frank observations, constructive criticism, and realistic suggestions. These will not adversely affect your course grade. Any students who have reservations about how such information is being used may withhold comments until after the coursework is marked, or until the end of term.

Please start by executing the online Pre-Test and making a Wiki entry reviewing your course preparedness within the first 2 weeks of the semester.

[1] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Course Perceptions Questionnaire Offline Version[2]

Please read the course outline and/or the course web page before attempting this questionnaire. The questionnaire should require about 30 minutes to complete. It is not a test, and will not contribute to your course grade.

- What do you think this course is about?
- What do you want to get from this course?
- What do you think you will have to do in this course?
- What skills do you think you need in order to do well in this course?
- Fill in the blanks in the following statements:
  1. Each week, I expect to do ________ hours of self-study for this course.
  2. The coursework for this course is worth ________ % of my final grade.
  3. There are ________ learning units in this course.
  4. Open Hours will be held on ____________________ from ______ to ______.
- Label each of the following statements as either True(checked)/False:
  - Week 13 classroom activities are all voluntary. ☐
  - I need to pass the exam in order to pass this course. ☐
  - If I cannot make the tutorial, then the deadline doesn't apply to me. ☐
  - Ignorance of departmental, faculty or university regulations is a valid excuse for violating them. ☐

[2]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

- Label each of the following statements, concerning student behaviour in the course, as either Acceptable (checked) or Unacceptable:
  - Student A copies Student B's program onto a diskette with Student B's permission. ☐
  - Student A copies Student B's program onto a diskette without Student B's permission. ☐
  - Student A does Student B's coursework. ☐
  - Student A does a tutorial set individually and compares answers with Student B. ☐
  - Student A has a problem with an assignment, and reads Student B's answer. ☐
  - Student A has a problem with an assignment, and uses "draft" for assistance. ☐
  - Student A has a question about a lab, which Student B answers. ☐
  - Student B reads, and points out an error in, Student A's code. ☐
  - Student B steals Student A's printout off the printer. ☐
  - Student B submits a program that contains code taken from the Internet, and does not acknowledge his source. ☐
  - Students A and B submit identical assignments. ☐
  - Students A and B work together to understand an assignment, and then complete it individually. ☐
  - Students A and B work together, and submit identical assignments. ☐
- For each of the following questions, select the most appropriate answer.
  1. The penalty for late submission of coursework is:
     (a) 1.25 marks per hour. ☐
     (b) 0.125% per 3 hours. ☐
     (c) 0.125% per day. ☐
     (d) there is no penalty for late submission. ☐
  2. The purpose of all these feedback forms, and tutorials is to:
     (a) stress me out. ☐
     (b) catch cheaters. ☐
     (c) make sure that I am doing the work. ☐
     (d) provide continuous feedback about my performance in the course. ☐
  3. In order to learn how to program microprocessors in assembly, which of the following is MOST effective:
     (a) read books about assembly. ☐
     (b) practice writing assembly programs on paper. ☐
     (c) test assembly programs in simulation. ☐
     (d) test assembly programs on microprocessors. ☐
  4. Which of the following is NOT a prerequisite skill for this course:
     (a) I must be able to program in C++. ☐
     (b) I must understand hexadecimal and binary number representations. ☐
     (c) I must be able to predict how simple digital/analog circuits will work. ☐
     (d) I must be able to read/understand diagrams typically found on datasheets. ☐

Prerequisite Skills Questionnaire Offline Version[3]

The questionnaire should require no more than 60 minutes to complete. If you are unable to answer a question please write 'Unable to answer' in the space provided. It is not a test, and will not contribute to your final grade.

1. Indicate which of the following courses you have already registered for, and the last grade you obtained in each course
   - analog electronics ☐ ______
   - digital electronics ☐ ______
   - introduction to computers ☐ ______
   - computer systems / C++ ☐ ______
2. Name any C/C++ compilers/IDE's which you are familiar with/have used.
3. Write down the screen output of the following program:

```
int main()
{
    int i;
    for (i=0; i<5;i++)
    {
        cout<<"The number is "<<i<<"\n";
    }
    return 0;
}
```

4. A group of 12 marbles may be grouped as:
   - 1 group of 10, and 2 groups of 1, OR
   - 1 group of 7, and 5 groups of 1

   Use this example to explain the concept of numeric bases.

[3]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

5. Convert the number 227_{10} to binary (base 2) and hexadecimal (base 16).

6. How would you write the number A_{16} in a C program so that the compiler understood that it was a hexadecimal number?

7. What is the relation between current, voltage, and power dissipated for a resistor of value R?

8. What is the relation between current, charge held, and voltage for a capacitor of value C?

9. Draw the symbol for and describe the operation of a D-type latch.

10. Draw the electrical symbol for and describe the operation of a simple NPN transistor.

11. Draw the electrical symbol for and describe the operation of a basic operational amplifiers.

12. Describe in your own words, the function of each of the following pieces of lab equipment:
    - meter
    - oscilloscope
    - power supply
    - breadboard
13. Which piece of lab equipment would you use to determine if there is no/low voltage on the power line.
14. Which piece of lab equipment would you use to determine if there are intermittent signals on a signal line.
15. Identify one possible source of EM noise in the lab.

16. A pin-out diagram, functional diagram and timing diagram for an DIP packaged IC are shown.

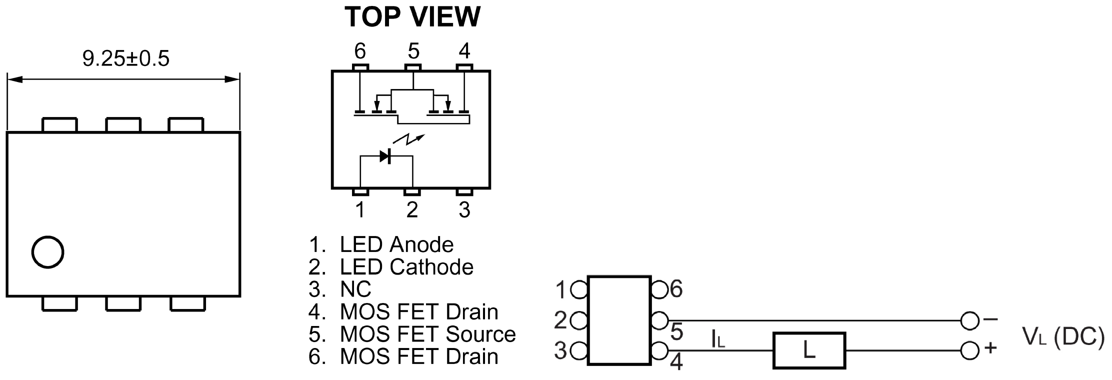
(a) Which pin is the V_o signal connected to? ______

(b) What is the relation of the OUT signal to the signals at pins 1 and 2?

(c) What time will elapse between a change at pin 1 and a corresponding change in V_o?

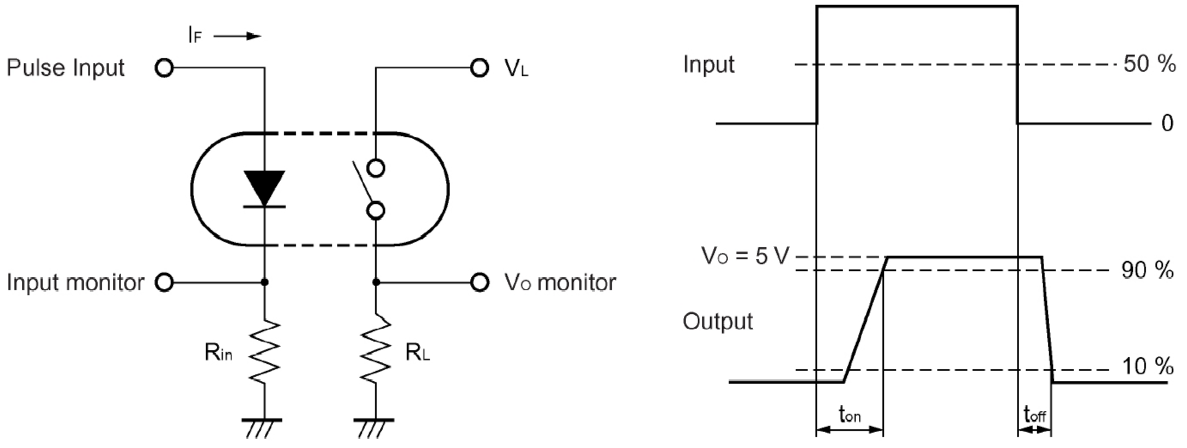
Diagrams taken from
`http://www.datasheetarchive.com/semiconductors/download.php?Datasheet=1696386`

PS7113-1A



1. LED Anode
2. LED Cathode
3. NC
4. MOS FET Drain
5. MOS FET Source
6. MOS FET Drain

***1** Test Circuit for Switching Time



Part I

Microprocessor Overview

Central processing units (*CPU*s) for early computers, were originally constructed from discrete electronic components. The *microprocessor* as we know it today has all the CPU functions contained in a single silicon chip. Because of the component densities involved, such chips are designed using hardware definition languages to facilitate component layout; however they are still made up from the same basic digital components: logic gates, adders, data latches, counters etc.

The intent of this course is to bridge the gap between high-level programming (which requires no knowledge of microprocessor internals) and digital circuit theory (which do not explain how a microprocessor can be constructed therefrom) as covered by other courses; and further to examine the ways in which microprocessors can be used to control external devices.

In this first part, a quick review of microprocessor operations is made to familiarise the reader with terminology and issues.

1 Microprocessor Basics

At the end of this unit the student will be able to:

- *identify the components of a typical microprocessor (ALU, registers, CU, bus),*
- *describe the operation of the components of a typical microprocessor,*
- *describe the representation of instructions in the operation of the instruction cycle.*

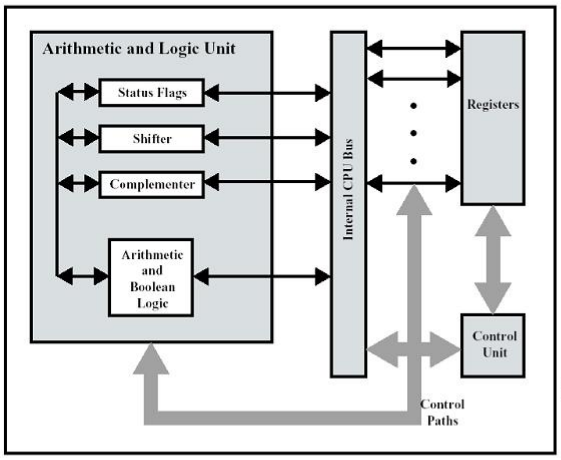
The central processing unit consists of four major parts:

- the *register*s, are a form of internal memory, which may be implemented using edge triggered flip-flops (latches). Registers within the CPU are used to specify the address of the next program instruction (*program counter* PC), to hold the retrieved instruction (*instruction register* IR), the instruction arguments (general purpose registers), the instruction result (*accumulator* or working register), and the validity (flags or status register) of the instruction operation.
- the Arithmetic and Logic Unit (*ALU*) consists of various combinatorial and/or sequential logic circuits which perform addition, shift, complement, bit-wise AND, bit-wise OR etc. on the *data inputs*, producing an *output*. The function to be performed is specified by the *control inputs*.
- the Control Unit (*CU*) exerts the *control signals* for all the other CPU components; it co-ordinates the activities of the CPU components in order to retrieve and perform the desired *machine instruction* through a sequence of *micro-operation*s.

CPU Internals

ALU: manipulates data in response to signals from the control unit

Bus: Transports data between ALU and registers



Registers: Storage for data

Control Unit: issues control path signals (micro-operations) to other parts of CPU

Instruction vs. Micro-operation

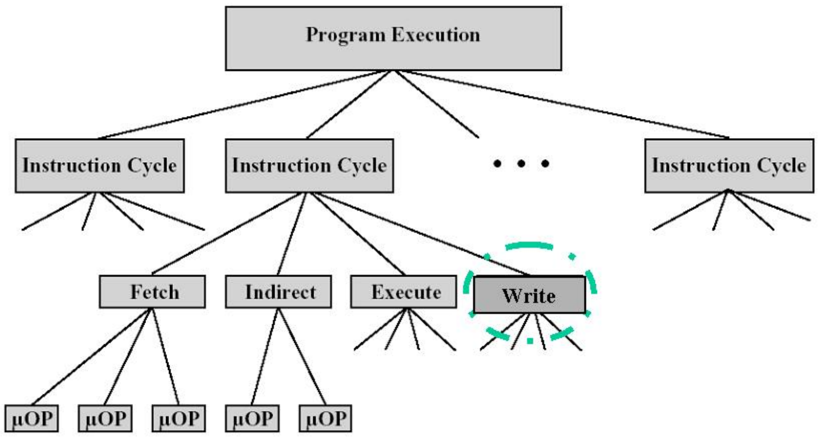


Figure 1: Figures from (Stallings 2000)

- the internal *bus*, which transports information between the different components of the CPU. The bus contains several electrical "lines", whose values are changed/read simultaneously. All other CPU components are connected to the bus, and place data on/read data off it, when instructed by the CU.

As in *sequential logic* circuits, activities within the CPU are coordinated by an externally provided *clock* signal. The time between successive clock edges is often referred to as the *clock cycle (period).* The *machine cycle* is the smallest quantum of time in which the CU carries out batches of micro-operations; the machine cycle must be a multiple of the clock cycle.

The machine instruction is represented as a sequence of 0's and 1's known as *bits.* It is often written as as binary or hexadecimal number. Typical machine instructions include data transfer and arithmetic operations. A *program* which executes on the CPU is a sequence of machine instructions, which are carried out one at a time. The *instruction cycle* as it is called can be broken down into actions which must be repeated for each instruction, such as *fetch*, *indirect (decode)*, *execute*, and *write.* The CU is responsible for co-ordinating the fetch of an instruction, decoding the instruction, co-ordinating the retrieval of any additional operands, instructing the other parts of the CPU to carry out the instruction, and co-ordinating the storage (write) of the final result. It does so by using a sequence of *micro-operations* in consecutive machine cycles. Each micro-operation in effect translates to the assertion of particular *control signals* by the CU.

For example, to execute an addition, the CU must:

- signal for values to be placed on the bus,
- signal the ALU adder inputs to latch it's inputs from the bus,
- assert the ALU control signal for addition
- trigger the ALU
- signal the ALU adder output to be placed on the bus,
- signal for values to be latched into storage from the bus

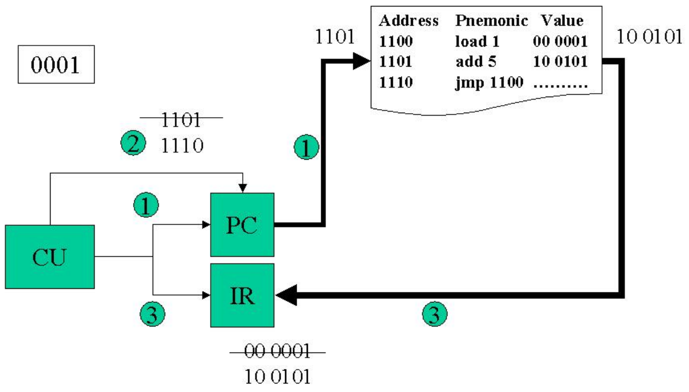
Each step is a micro-operation. Micro-operations can be carried out in parallel as long as they are not dependent on the result of a previous micro-operation. The number of sequential micro-operations required for the various stages of each instruction in the instruction set, determines the duration of the *instruction cycle.*

The indirect and execute parts of the instruction cycle will use different micro-operations depending on the machine instruction. The amount of logic circuits (and hence time for the indirect part of the instruction cycle) for an instruction will depend on the instruction *bit-width* and the complexity of the instruction (i.e. number and type of operands etc). The execution time will also depend on the sequence of control signals the CU must perform, and the time taken for the various parts of the CPU to respond.

Note: Where there is great disparity, certain instructions in the instruction set may be specified as taking more than one instruction cycle.

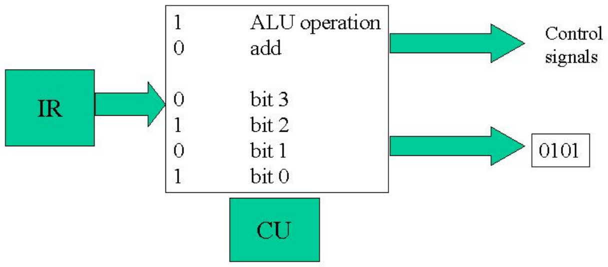
The sequence of micro-operations, required to implement a particular instruction, will also depend on the organisation/layout of the CPU. The sequence/time can be reduced by using additional special-purpose hardware (for example the inclusion of a multiplier unit in the ALU), or duplication of certain heavily used functions (for example multiple addition units).

Add Instruction - Fetch

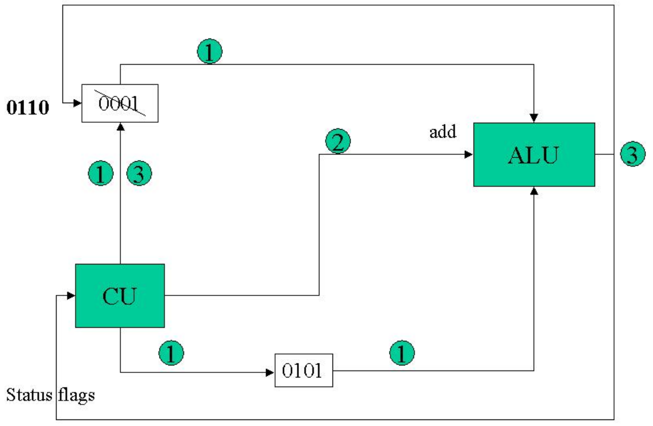


Address	Pnemonic	Value
1100	load 1	00 0001
1101	add 5	10 0101
1110	jmp 1100	....

Add Instruction - Indirect (Decode)



Add Instruction - Execute



The micro-operations for a particular instruction may be hard-wired (i.e. the control signals are fixed w.r.t the instruction code), or can in turn be implemented as *micro-code* (i.e. the micro-operations for a particular instruction are in turn looked up in some internal memory of the CU; effectively another (albeit primitive) programming level). Examples of CU implementation may be found in (Noergaard 2005; Section 4.2.1 Central Processing Unit).

Micro-code is most useful in Complex Instruction Set Computer (*CISC*) architectures, where large multi-operand instruction sets require make it impractical to hard-wire control signal logic. CISC architectures arose when the time for external memory fetch vastly exceeded the processing time within the CPU. It was thus presumed to be advantageous to reduce the number of instruction fetches by making more complex, wider, instructions. For example an addition may be performed in two ways: `load 5, add 1` or `add 5,1`. The former requires two memory fetches, the latter a single memory fetch.

In contrast, once external memory access speeds improved, a different philosophy was proposed: Reduced Instruction Set Computer (*RISC*) Architecture. The main principle behind Reduced Instruction Set Computing (RISC) is that it is better to get a processor to do a few simple instructions very quickly, than to get it to do many different instructions at the expense of speed and efficiency. RISC architectures have so few instructions that there is no advantage (and a time disadvantage) to adding another level of decoding, therefore micro-code is rarely used for such architectures.

Pipelining is a technique developed to improve performance by overlapping operations which can be performed concurrently. Pipelining can be applied to the four parts of the instruction cycle listed previously, i.e. Fetch, Decode, Execute, and Write. Non-pipelined processors do all four actions for a single instruction before starting the process on a new instruction. However, if each stage could be simplified, and the hardware designed such that each operation can occur independently, then a pipeline can be implemented such that the outputs of each stage provide the input to the next. Although each instruction will take the same time for complete execution, consecutive instructions will be executed more quickly. Pipelining works best if all instructions execute within the same time. RISC architectures more readily facilitate pipelining, because of the small fixed instruction size, and the emphasis on the speed of instruction execution.

RISC vs. CISC

- Single-cycle instruction execution
  - Simple, fixed instruction format
  - few instructions and addressing modes
  - Hardwired micro-operations
- Memory Access
  - Load/Store design
  - High-performance memory (registers/cache).
- Predictable Speed/Performance

- Multi-cycle instruction execution
  - Variable size instruction format
  - Micro-programmed instruction set
- Smaller program size (# of instructions)
- Memory Access
  - Multiple addressing modes
- Optimization Complexity

Pipelining

- Assumptions:
  - stages all have same duration
  - no branching
  - consecutive instructions are independent

Fetch1	Indirect1	Execute1	Write1	Fetch2	Indirect2	Execute2	Write2

Fetch1	Indirect1	Execute1	Write1				
	Fetch2	Indirect2	Execute2	Write2			
		Fetch3	Indirect3	Execute3	Write3		
			Fetch4	Indirect4	Execute4	Write4	
				Fetch5	Indirect5	Execute5	Write5

Review Exercises

1. Write a definition for all the italicised terms in this unit.
2. Label the following statements as True/False:
   (a) The CU co-ordinates the operation of the various parts of the CPU.
   (b) Registers are located in a special section of RAM external to the CPU.
   (c) The PC contains the address of the last instruction fetched.
   (d) The CPU co-ordinates all operations within the CU.
   (e) The Program Counter (PC) contains the address of the instruction being executed.
   (f) The ALU is responsible for decoding instructions.
   (g) The ALU is the part which executes the program.
   (h) The internal bus conveys information between the CPU and the microprocessor.
   (i) The ALU is separate from the CPU.
   (j) Registers are storage locations external to the CPU.
   (k) A CPU must have a stack, as well as special function registers.
   (l) The internal CPU bus transports information between the CU and the CPU.
   (m) The clock signal is used by the CPU to time the phases of the instruction cycle.
   (n) CISC architecture processors have larger, more flexible instruction sets than RISC architectures.
   (o) RISC instructions require fewer micro-operations per instruction than CISC instructions.
3. For each major part of the CPU
   (a) state the primary function of the part
   (b) describe the role of the part in a typical CPU instruction cycle
   (c) draw a diagram to illustrate how the part could be implemented using digital logic.
4. Which ONE of the following stages is NOT part of the microprocessor instruction cycle:
   (a) instruction fetch stage
   (b) execute stage
   (c) instruction decode stage
   (d) micro-operation stage
5. The overall purpose of the CU is (choose ONE answer):
   (a) to co-ordinate the operation of the various parts of the CPU.
   (b) to place information on the internal CPU bus.
   (c) to communicate with memory and peripherals.
   (d) to pipeline instructions, by performing different micro-operations simultaneously.

6. Choose the word combination which best completes the following sentence:

   Within the microprocessor, the __6I__ perform(s) arithmetic operations on information delivered from the __6II__ via the __6III__.

   (a) 6I: ALU 6II: registers 6III: bus
   (b) 6I: ALU 6II: registers 6III: CU
   (c) 6I: CU 6II: ALU 6III: bus
   (d) 6I: CU 6II: ALU 6III: registers
   (e) 6I: registers 6II: CU 6III: ALU

7. Differentiate between the following items:

   (a) CPU and microprocessor
   (b) machine instruction and a micro-operation
   (c) ALU and CU
   (d) machine cycle and instruction cycle

8. For a simple CPU, **list the micro-operations** the control unit must do to execute a subtraction; presuming that the ALU can perform subtraction, and that the arguments are in the accumulator and instruction respectively.

9. RISC, CISC and pipelining are all approaches to improving CPU performance, yet they are based on very different concepts. What would you consider if you had to decide which was best for a particular application?

10. Role Play: Assign people to be the parts of the CPU: CU, ALU, bus, and the registers (PC, IR, Accumulator, and a general purpose register). Work your way through the instruction cycle for each instruction of a simple program, involving load, store, jump, and arithmetic operations.

Tutorial Exercise 1A Offline Version[4]

ID# ______

1. In your own words, describe what a register is, and give examples of how it interacts with, or is used by, each of the other three parts of the CPU. *4 marks*

2. In your own words, differentiate between the following items:

   (a) machine instruction and machine cycle *3 marks*

   (b) clock signals and micro-operation control signals *3 marks*

PLAGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

[4]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 1 Your lecturer will put up a diagram with several parts labeled with letters.

Write the letter you have been assigned here________.

Answer the following questions for the part whose letter you were assigned.

1. The part is called the ___________________________.
2. Circle the correct answer. The primary function of the part is:
   - (a) to perform arithmetic and logic functions ☐
   - (b) to co-ordinate operations within the CPU ☐
   - (c) to store data within the CPU ☐
   - (d) to transfer data between different parts of the CPU ☐
3. Describe the role of the part, using a typical CPU instruction cycle.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - identify the components of a typical microprocessor (ALU, registers, CU, bus),
  - describe the operation of the components of a typical microprocessor,
  - describe the representation of instructions in the operation of the instruction cycle.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

2 Microprocessor System Basics

At the end of this unit the student will be able to:

- *identify the components of a microprocessor-based system*
- *describe the operation of typical components of a microprocessor-based system*
- *discuss issues involved in the design/choice of various components for a microprocessor-based system*

Ball (2000) defines a microprocessor as

> "An integrated circuit containing, at minimum, a central processing unit and a means to access external memory."

The central processing unit (CPU) accesses external memory to obtain machine instructions (what to do) and stored data (what to do it to).

There are several types of memory; all memory may be read from, but only certain types may be written to. RAM (Random Access Memory) is the most flexible, allowing the user to read and write without any special considerations. However this is *volatile* (i.e. it disappears when power is removed). Further, *dynamic RAM* must be periodically refreshed even while power is supplied, making *access times* relatively slow. The advantage of dynamic RAM is that it is denser (more memory per unit area) and cheaper than *static RAM* due to the simpler circuitry required.

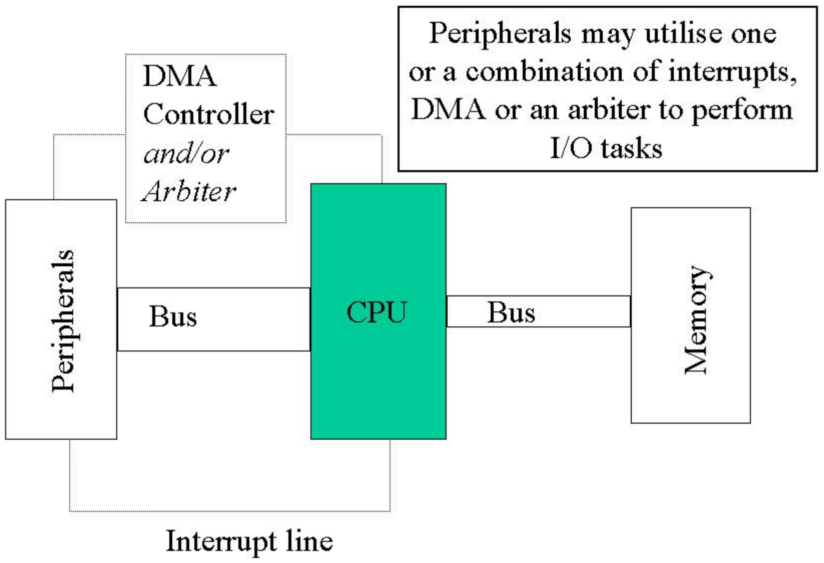
ROM (Read Only Memory) and *PROM* (Programmable Read Only Memory) may only be configured once, and are then permanent. The difference between these two is the means by which they are written. ROM's are configured in the factory before packaging. PROM's offer flexibility, in that they are configured after the chip has been packaged. *EPROM*'s are erased under UV light, and can then be re-programmed in a similar manner to PROM's. *Flash* EEPROM memory has the advantage that it may be re-programmed *in-circuit*. While access times for ROM's are comparable to or better than those for RAM, the write time for Flash EEPROM's are significantly longer.

Each of these types of memory can be visualised as an array of memory *cell*s. The ability to access external (or internal) memory which consists of more than one item, requires some method of *addressing*. i.e. a means of specifying which location within the memory is to be accessed. A single memory cell may be represented stylistically as a black box which stores a binary value (0/1) with 3 connections: a Write Line, a Select Line and a Data Line. When the Select line is not active, no action is taken. When the Select Line is active, and the Write Line is active, information from the Data line is stored. When the Select Line is active and the Write Line is not active, the stored data is output. The actual implementation of the memory cell may be latch based (static RAM), capacitive based (dynamic RAM), or hard wired connections (ROM/PROM).

When choosing memory, there are three major considerations:

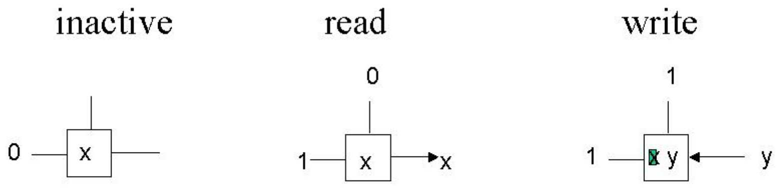
- whether it is Read-Only or Read-Write
- the timing requirements of the CPU/memory for read/write access, and
- the bit-width of the stored data

Microprocessor-based System


Peripherals may utilise one or a combination of interrupts, DMA or an arbiter to perform I/O tasks
DMA Controller *and/or* *Arbiter*
Peripherals
Bus
CPU
Bus
Memory
Interrupt line

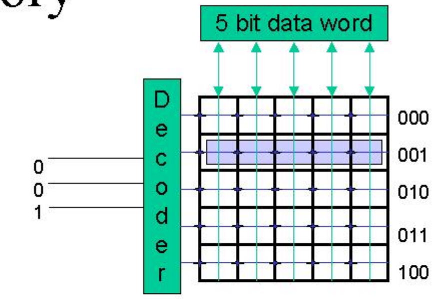
Memory

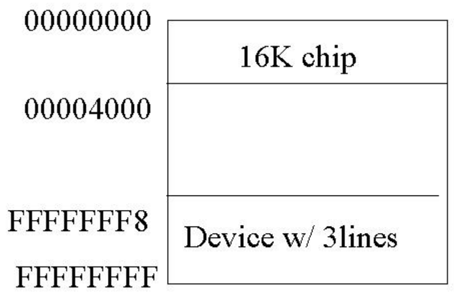
- Memory types
  - RAM
  - ROM, PROM
  - EPROM, EEPROM, Flash
- Access time, size, support circuits (DRAM)
- RAM Memory Cell



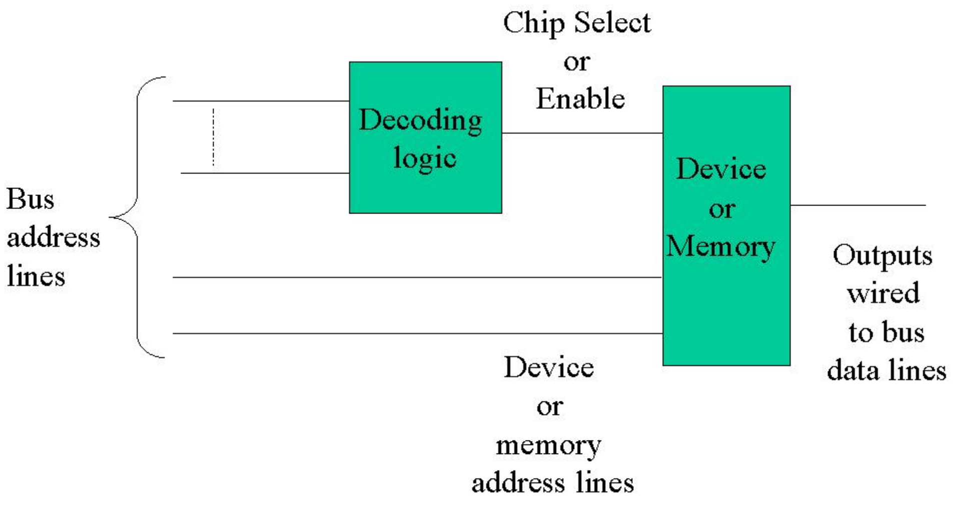
Memory

- Address/Data Width
  - bit, nibble, byte
  - powers of 2
    - kilo(10)
    - mega (20)
- Address Decoding
  - in-array
  - multiple memory chips
  - memory mapped device
  - adjustable:
    - hardware jumpers
    - software selectable





Address Decoding



Address decoding logic is required to connect cells in the array to the addressing mechanism. This can be made up of a sequence of logic gates. Memory is assigned addresses in sequential order according to the line of the decoding logic it's select line is connected to. Multiple memory cells may be connected to the same address line, with their outputs connected in parallel to separate output lines. In this way a *data word* may be retrieved from a single address. The data word size is typically a power of 2. The most frequently used sizes are the *nibble* – 4 bits, *byte* – 8 bits, 16, 32, 64 bits. The number of bits which make up the address are determined by a) the offset address and b) the memory size (power of 2). It is often convenient to make the address and data bit width identical to facilitate the sharing of the *bus* lines.

To retrieve data from (or send data to) external memory, the CPU must use some means of communication with the memory. This interface is known as a *bus*. A bus consists of one or more lines which are connected to multiple components. Data may be sent in one/both (*uni/bi-directional*) directions across the bus. The bus may be *synchronous* (information is read/written at particular instances as determined by the clock line on the bus). If more than one component may place data on the bus, it must be *arbitrated* (components must seek permission to write to the bus) either centrally, or in a distributed manner. Information sent across a bus may use a single (one address - one data word) or block transfer(one address - multiple data words over successive bus cycles).

The bus lines used for providing the memory address may be shared with (*time multiplexed*), or be independent of the bus lines used to return the data to the CPU (i.e. separate data and address bus lines). Bus operation is often described using *timing diagrams*. These diagrams are drawn using horizontal traces each representing the signals on an individual or group of bus lines. A trace moves between two levels representing binary 0 and 1. Where lines in a group may be at either value, a box is used. The relative positions of signal transitions, indicate the sequence in which events occur while cause-and-effect relationships between signal transitions are indicated using arrows. For synchronous buses, all transitions occur on the edge of the clock signal.

The type of bus used for a particular application depends on:

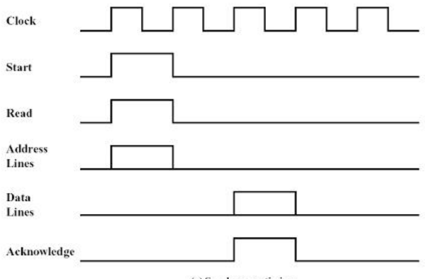
- the size of the memory/device *map* (i.e. the address space accessed by the bus)
- the width of the data transferred across the bus
- the timing requirements of the components connected to the bus

The CPU as reviewed above interacts only with memory, across one or more busses. If the CPU is to interact with *peripheral* devices (e.g. Keyboards, LCD displays, UART, DAC), some means of doing so must be established. Because such devices employ physical phenomena, *transducer*s must be used to convert to/from electrical energy. The transducers cannot be directly connected to the system bus; *I/O modules* are used to interface the transducer to the system bus. An I/O module consists of: circuitry for reading/controlling the transducer(s), a set of registers (to buffer the transducer reading/control commands, control flags etc), and address decoding logic. Each I/O module responds to requests for information in a manner similar to memory. The range of addresses to which the I/O module will respond depends on the number of registers which need to be accessed.

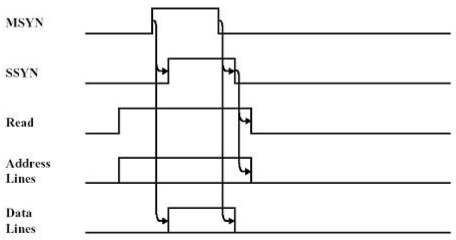
The simplest means of interacting with I/O modules is to use a separate bus (or bus lines), and include separate instructions for retrieving data from the devices. The addresses for I/O are completely independent of the memory addresses, and can even be duplicated (as for code and

Bus

- Collection of signal lines -- uni(bi)directional
- Operation represented using timing diagrams
- Clocked vs Asynchronous
- Shared (Memory) or Separate (I/O)
- Separate data and address buses



(a) Synchronous timing



(b) Asynchronous timing

Input/Output

Isolated vs. Memory mapped
(see von Neumann
vs
Harvard architectures)

DMA

- DMA receives I/O ready
- Cycle stealing
  - Wait for the CPU to request bus
  - Put CPU to sleep
  - Perform 1-word data transfer
  - Wake CPU up
  - CPU proceeds with bus operation
  - if more data loop
- Trigger CPU Interrupt

Arbitration

- When I/O ready
  - request bus from arbiter
  - when bus granted, transfer all data
  - release bus

Interrupt from I/O

- CPU receives I/O ready
- Handler: Dedicated Copy
  - CPU reads I/O
  - CPU writes (completes transfer)

Choose based on peripheral data transfer requirements data in the Harvard architecture). This is referred to as *isolated I/O*. More commonly, the memory and I/O share the bus and address space. This is referred to as *memory mapped I/O*. It's primary advantage is that separate instructions for accessing I/O do not have to be added to the instruction set.

In practice, the information read from certain devices needs to be sequentially stored in memory over time (e.g key-presses). The CPU may spend a significant portion of time, polling/waiting for the (slower) I/O device. Direct memory access (*DMA*) is the practice of delivering data from the I/O module to a specific area of memory without the involvement of the CPU; thereby removing the need to poll, buffer (or lose) data in the I/O module. When the information becomes available, the DMA controller sends it out over the bus one byte at a time. To gain control of the bus, the DMA controller waits until the processor is about to use the bus, and puts the processor to sleep for a bus cycle (*cycle stealing*). Once all data has been transferred, the DMA controller triggers an *interrupt* to let the CPU know.

Interrupt processing (if the microprocessor supports it) is part of the instruction cycle. After the write phase, the control unit checks to see if the interrupt line has been raised. If it has, it executes call-like jump to the interrupt handler. The difference between a call and an interrupt is in the level of protection given to the registers. For a call, only the PC is stored and restored on return. An interrupt should store/restore all registers, so that the program can proceed as though nothing occurred. The starting location of the interrupt handler is generally fixed (in the same way as the reset location is fixed). Interrupts can be used, not only by the DMA controller, but by any I/O module which has access to the interrupt line. In doing so, the CPU can recieve/process information right away. *Interrupt-driven* I/O is a viable alternative to polling, or DMA, when the interrupt routine is small.

Because the I/O module possesses registers, and often has itś own clock signals, and requires timing/complex algorithms to sample/control the transducer at regular intervals, the I/O modules for certain standard peripherals are often implemented using microprocessors (albeit smaller than the main CPU), and packaged separately as Peripheral Interface Controllers (*PIC*s).

The use of DMA, bus arbiters, PIC's and/or interrupts are dependent on the properties of the individual peripheral,

- access time,
- frequency of input/output,
- bit-width, and *density* of the data to transfer

as well as whether the CPU supports the use of these particular "aids". A microprocessor-based system may contain several of each of these "aids".

Review Exercises

1. Write a definition for all the italicised terms in this unit.
2. Non-volatile RAM (NVRAM) is a combination of RAM and EEPROM memory. Data is read from/written to RAM, and backed up to the EEPROM at appropriate times. Can you think of an application where this would be useful?
3. (a) Outline how I/O modules could use the interrupt line to move data to memory as soon as it is available. What are the relative advantages/disadvantages compared to DMA.
   (b) Most microprocessors have a single interrupt line; multiple interrupts are handled by a special I/O module known as an interrupt controller, which has access to the microprocessor interrupt line. Suggest ways in which interrupts from multiple devices could each be handled (with their own handler) in such a scheme. Should an interrupt be allowed to interrupt another interrupt?
   (c) "A DMA module is transferring characters to memory using cycle stealing, from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1 million instructions per second (1 MIPS). By how much will the processor be slowed down by DMA activity?" – question taken from (Stallings 2000; 6.5)
4. List the issues to be considered in the design/choice of the following items for a particular application:
   (a) CPU
   (b) bus
   (c) memory
   (d) peripherals
5. Role Play/Challenge question taken from (Stallings 2000; 6.9):

   "

   Two boys are playing on either side of a high fence. One of the boys named Apple-server, has a beautiful apple tree loaded with delicious apples growing on his side of the fence; he is happy to supply apples to the other boy whenever needed. The other boy named Apple-eater, loves to eat apples but has none. In fact, he must eat his apples at a fixed rate . . . If he eats them faster than that rate he will get sick. If he eats them slower, he will suffer malnutrition. Neither boy can talk, so the problem is to get apples from Apple-server to Apple-eater at the correct rate

   (a) Assume that there is an alarm clock sitting on top of the fence and that the clock can have multiple alarm settings. How can the clock be used to solve the problem? Draw a timing diagram to illustrate the solution.
   (b) Now assume that there is no alarm clock. Instead Apple-eater has a flag that he can wave whenever he needs an apple. Suggest a new solution. Would it be helpful for Apple-server also to have a flag? If so, incorporate this into the solution. Discuss the drawbacks of this approach.
   (c) Now take away the flag and assume the existence of a long piece of string. Suggest a solution that is superior to that of (5b) using the string.

   "

6. One difference between Dynamic RAM (DRAM) and Static RAM (SRAM) is:
   (a) DRAM loses information when power is switched off, but SRAM retains information when power is switched off.
   (b) DRAM tends to be more expensive than SRAM.
   (c) DRAM tends to be less dense than SRAM.
   (d) DRAM is faster than SRAM.
   (e) DRAM loses information unless it is periodically refreshed, SRAM will retain information indefinitely.

7. Which of the following statements about the components of a computer are correct(if any):
   (a) The ALU is the part which executes the program.
   (b) The bus conveys information between the CPU and the microprocessor.
   (c) The clock signal is used by the CPU to time the phases of the instruction cycle.
   (d) A microprocessor is a CPU plus peripherals on a single chip.
   (e) Memory may be read-only (RAM) or read-write (ROM).

8. Which ONE of the following statements about interrupts is correct:
   (a) Interrupts occur when peripherals send bit-parallel signals to the CPU across the bus.
   (b) Interrupts occur when memory devices send bit-parallel signals to the CPU across the bus.
   (c) Interrupts occur when the CPU exerts a special signal line connected to a peripheral.
   (d) Interrupts occur when peripherals exert a special signal line connected to the memory.
   (e) Interrupts occur when peripherals exert a special signal line connected to the CPU.

9. Choose the best answer. The memory map size affects the nature of the microprocessor-based system bus, because:
   (a) the number of data lines must match the memory bit-width.
   (b) the number of address lines must allow the entire address range to be addressed.
   (c) if the number of data and address lines are the same, the lines can be multiplexed.
   (d) 9a and 9b are true but 9c is false.
   (e) 9a, 9b and 9c are all true.

10. Role Play: Investigate bus issues such as, contention arising from two devices with the same address, or no device responding to the address, on both a synchronous and asynchronous bus, and where data and address lines are shared or separate.

    Suggest lining up people (devices) with transparencies representing the data they will place on the bus. Person at back (the bus state) writing on a pad of slightly translucent paper, from back to front, so we can see the transient state of the bus. Other persons to represent the clock or signalling lines, and the address lines(transparency if shared bus lines).

Tutorial Exercise 1B Offline Version[5]

ID#

1. Draw a diagram showing the five phases of the instruction cycle. *3 marks*

2. Identify two advantages/disadvantages of memory-mapped I/O when compared to isolated I/O. *4 marks*

3. Differentiate between bus arbitration and DMA. Your answer should make reference to the CPU's ability to use the bus. *3 marks*

[5]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 2 Your lecturer will put up a diagram with several parts labeled with letters.

Write the letter you have been assigned here________.

Answer the following questions for the part whose letter you were assigned.

1. The part is called the ____________________________.
2. Circle the correct answer. The primary function of the part is:
   (a) to transport data between the CPU and external devices
   (b) to store data outside of the CPU
   (c) to interface the CPU to external devices e.g. actuators, sensors
   (d) to allow devices use of the system bus independently of the CPU
3. List 2 things you need to consider when choosing this component.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - identify the components of a microprocessor-based system
  - describe the operation of typical components of a microprocessor-based system
  - discuss issues involved in the design/choice of various components for a microprocessor-based system
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

3 Microprocessor support circuitry

At the end of this unit the student will be able to:

- *explain the function of basic support circuitry for microprocessors such as clocks, reset circuitry, watch dog timer, capacitors to ground, etc.*
- *describe (and/or illustrate) how bus addressing, conflict resolution, and memory maps may be implemented*

A microprocessor requires power and ground inputs. There may be more than one pin assigned for each of these, and if so, the group of pins for each should be tied together. The power line(s) are typically labelled V_{dd} or V_{cc} for the positive supply input and V_{ee} or V_{ss} for the negative supply input. In most cases, we will be using the positive supply input as +5V, and the negative supply input as ground; however this is not always the case. You are advised to check the datasheet(s) for individual IC's.

Typically, datasheets recommend that a capacitor be placed between the power and ground lines. This is to *decouple* the supply lines. Sourcing additional current, can cause the power supply voltage to fluctuate; similarly, a change in current being sunk can result in ground bounce. The capacitor helps to mitigate these effects. In an ideal world, these effects should be minimised through proper circuit specification/design.

The pins on a microprocessor may be grouped[6] as control and data pins. A signal is said to be *assert*ed when the microprocessor is actively placing a value onto the pin (as opposed to letting the line float). The actual voltage on the pin when it is asserted, depends on two things:

- the voltages placed on the pins for true/false; 0/1; *logic high* and *logic low*
- the polarity of the logic (true – logic high or logic low)
- whether the signal is asserted with logical true, or logical false; active high or active low.

For example the CPU may wish to put the number 5 out on 4 data pins. In binary, 5 is 0101_2. The actual voltages on the 4 data pins will be 5V, 0V, 5V, 0V or 0V, 5V, 0V, 5V depending on the above items. By convention, the names of signals which are active low, are written with a bar e.g. $\overline{D1}$. When you are wiring up a microprocessor (whether physical wires or traces on a circuit board) all connections should be kept short, and isolated from strong sources of electro-magnetic interference. This will reduce signal corruption.

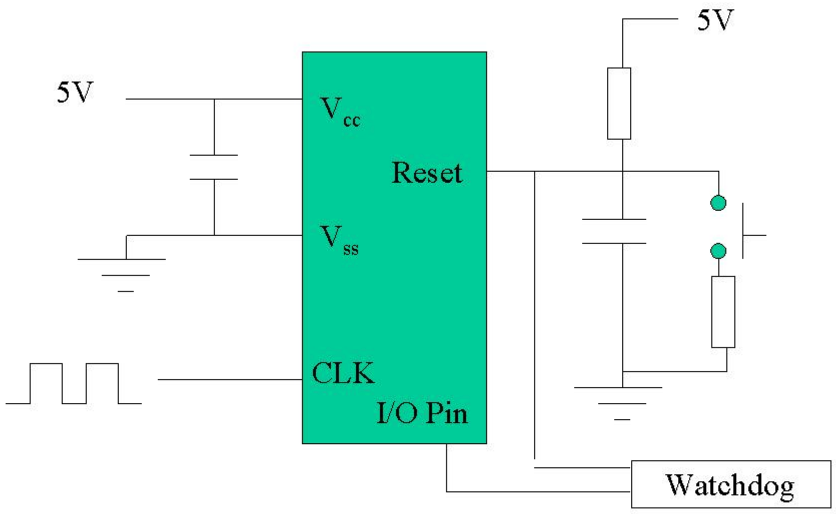
It is presumed that all devices to be connected to the microprocessor, operate at same voltage levels for logic high and logic low. In addition to voltage considerations, there will be a maximum amount of current which can be detected/sunk or sourced by an individual pin. This will either be specified in mA, or for certain logic families, as *"fan- in/out"*. Further there is generally a maximum current that the entire microprocessor can source/sink.

Where voltage and/or current specifications of the microprocessor will be violated, one of the following strategies may be used:

- change signal from active low to active high or vice versa (in order to reduce the total current sunk by microprocessor)

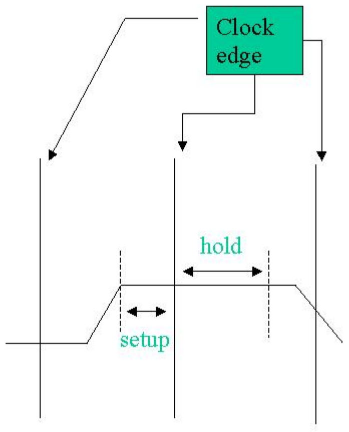
[6]This division is artificial, there is nothing to stop you using a data pin for control, or vice-versa. Typically control pins are associated with particular SFR's or specific hardware within the CPU

Support Circuitry



Timing diagrams

- May be associated with any component of the microprocessor-based system
- Signal lines move between two levels
- Each line represents either a single signal line/pin, or a group of lines/pins which are accessed simultaneously (e.g. data or address)
- For any signal, there is a minimum time which the signal must remain stable, before action takes place (setup time)
- For any signal there is a minimum time which the signal must remain stable while action is taking place (hold time)
- For synchronous systems, these times are referred to the initiating clock edge -- be careful to refer to the correct clock



- pull-up/pull-down resistors (provide alternate current source/sink);
- buffers (reduce loading on source);
- opto–isolators(electrically disconnect source/sink);
- relays and/or transistors (trigger higher powered circuitry)

The preceding discussion presumes that the outputs of the microprocessor are *push-pull* (totem-pole, active pull-up) i.e. logic high and logic low both translate to a virtual connection to either Vcc or GND. Outputs can also be *open-collector* (open-drain), where there is a virtual connection only to GND; in the alternate logic state, the output "floats". A third output structure is the *tri-state*; almost the same as the push-pull output except that it also has a third "floating" state.

The clock signal plays a major role in the operation of the CPU. Clock signals are the basis on which CPU instruction cycles occur, synchronous bus transactions occur, and I/O devices sample transducers. It is important to recognise that:

- all these signals need not necessarily be of the same frequency, or come from the same source.
- instructions are NOT performed at the same rate as the CPU clock frequency.
- the physical circuitry, imposes restrictions on the nature of the clock signal (minimum rise time, minimum fall time, maximum frequency, noise tolerance, load capacitance)
- different types of oscillators have different characteristics : Typical sources of clock signals include relaxation (RC) oscillators, and quartz-crystal oscillators. The choice depends on the frequency, accuracy, and consistency required in the clock signal, and the tolerance of the program/application for changes in temperature/pressure and resulting frequency drift.
- The oscillator may also be (partially) integrated into an IC along with the CPU core.

For example, let us consider a CPU with a 4MHz clock signal, connected to a 2MHz synchronous bus. If we presume 1 machine cycle for each of the phases of the instruction cycle (fetch, indirect, execute, write) the instruction cycle will be 4 times as slow as the machine cycle. i.e. the maximum instruction execution frequency will be 1MHz. If instructions must be fetched across the bus, the minimum time in which an instruction fetch can occur (ignoring memory access timing) is 1MHz. This will not affect the instruction execution frequency if the processor is pipelined, but a non-pipelined processor will be slowed further to approximately 500kHz instruction execution frequency.

When power is first applied to a microprocessor, it is in an indeterminate state. For instance, the clock signal (if source powered at same time) may not yet be stable, or initial power transients may result in spurious values in registers/on buses. Typically, the microprocessor contains an input, known as the "reset" or "master clear". Internally, this line forces all components of the CPU to a pre-defined initial state. In order to make certain of the CPU state, the reset line must be exerted for a certain minimum length of time. An RC series circuit may be used to ensure the pin is active for the required time. In many cases, a switch is connected to the reset line, so that the CPU can be reset without cycling power.

A transient in the power supply may also cause the processor to behave in a strange way, by corrupting register contents/bus states. The transient may not be long enough to properly trigger the reset. To prevent such occurrences, either condition the power supply, or use an external device which will generate a proper reset when a transient is detected. *Watchdog timers* (either internal to the microprocessor or external) are devices which will trigger an output if they are not periodically reset. They are useful devices for determining if the CPU has been subject to a transient, or has "locked-up" or has broken.

The bus is utilised by multiple devices, so voltage/current issues need to be examined with care. If items connected to the bus simultaneously assert signals, the bus will be in an indeterminate state. This is handled by

- using *tri-state* connections between device and bus; the device keeps connections in the high impedance state until bus access is granted/required
- actively detecting/avoiding bus conflicts[7]

Transactions carried out across the bus require that the line voltages are stable before an attempt is made to read the information; and further that the voltages on the bus remain stable for a certain length of time. These times are referred to as the *setup* and *hold* times respectively, and are specified on the device data sheet. These times are generally specified with reference to a timing diagram, which shows the sequence in which the bus lines must be asserted for "correct" device operation.

The bus accommodates Read and Write Requests by the requestor. The device which is the target of the request, must respond to a pre-set address, and either accept the information, or respond with information. The number of address locations is determined by the number of address lines on the bus (e.g. 8 lines; 2^8 addresses from 0 to 255).

The entire address space may be diagrammatically represented as a memory *map*. The memory map may, or may not, be *contiguous*.

The location of the device within the addressing space, depends on

- built-in device support for addressing – lines are connected to a sub-set of the bus address lines
- external address decoding logic, whose inputs are a sub-set of bus address lines, and whose output is connected to the device "enable" line

Finally, it is entirely possible that the address space accessible using the bus address lines is too small. There are two possible ways to extend the address space:

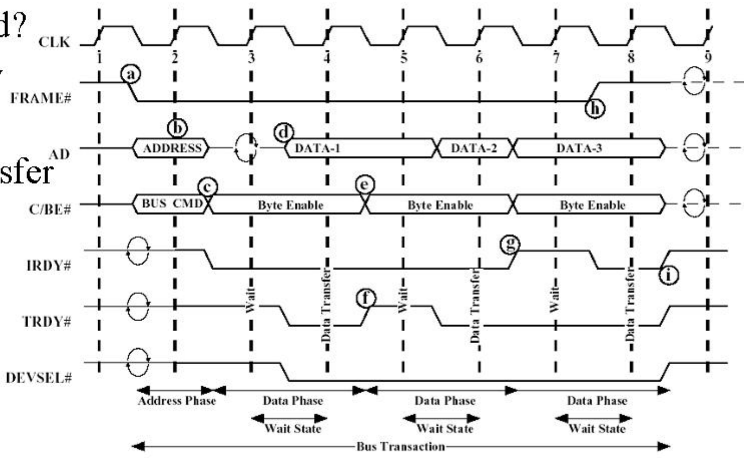
- The memory space can be "re-used" by using additional "non-bus" lines to select a particular device sub-set. This is the principle used to share I/O and memory space with a single bus, and may also be used to "bank" or "page" memory.
- The address width can be increased, by using multi-stage addresses, which are transmitted across the bus in successive words, and stored before device access is triggered.

[7]In the absence of an arbiter, bus conflict may be avoided by monitoring the bus while using it. If what is read is not what was written, there is a conflict, and the device backs off for a random time before retrying the transaction.

System Bus Control Signals

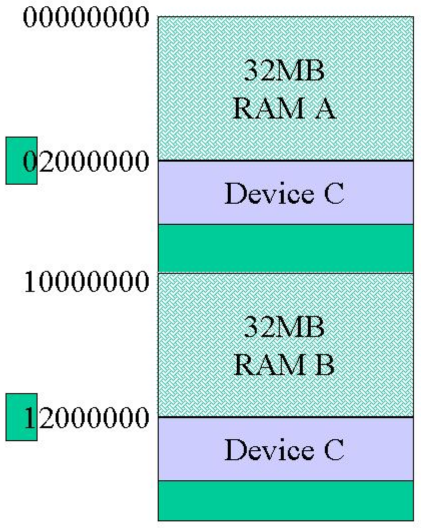
- Multiple users?
  - arbitration
  - sender/reciever id
- Multiplexed?
  - data ready
  - clock
- Block Transfer
  - busy

PCI Bus -- Read



- "Floor plan" for memory
- Mixed memory chip sizes
  - Place largest sizes first
  - Place at "multiple" borders
- Devices
  - Repeated place(s) in map
  - Avoid upper address lines
- Software Settable decoding logic:
  - Use the contents of a "latch" & compare with upper address lines; output of comparator is output of the decoding logic.
  - Give the latch it's own address decoder, and connect the inputs/outputs to the data lines so it can be read from/written to.

Memory Map



Review Exercises

1. Write a definition for all the italicised terms in this unit.

2. The minimum support circuitry required to make a microprocessor operate includes:

   I Address decoders for memory and memory-mapped perpherals.

   II Oscillator (or other external) clock signal

   III Power and Ground connections, with decoupling capacitor

   IV Reset Circuit and/or Button

   V Watchdog Timer

   (a) I, III and V
   (b) I, IV, and V
   (c) II, III, and IV
   (d) II, IV, and V

3. The function of the watchdog timer in a microprocessor-based system is BEST described as:

   (a) to alert/reset the CPU when it has been inactive for a certain period of time
   (b) to hold the reset line low for a minimum time on power-up
   (c) to alert/reset peripherals when there has been no communication for a certain period of time
   (d) to count the number of times the incoming signal goes low

4. You are working on a prototype microprocessor-based system, which contains a CPU and a synchronous bus with independent clocks. The only device on the bus apart from the CPU is the ROM containing the program. Suggest possible reasons for each of the following observations.

   (a) You change the bus frequency, and do not get an improvement in overall performance.
   (b) You change the CPU clock frequency and the system stops working.
   (c) The circuitry works correctly when connected to the 5V desk supply, but fails when connected to a 5V battery.
   (d) The circuitry works for the test cases: all bit inputs off, and each bit input tested singly; but fails when more than 3 bit inputs are tested together.

5. In your own words, comment on the reasons why:

   (a) Unacceptable levels of noise in the clock signal will result in the CPU performing badly (incorrectly).
   (b) "Most processor control lines are active low." (Arnold 2001; pp 56)
   (c) The maximum current sunk/sourced by an IC is generally less than the sum of the maximum currents sunk/sourced by individual pins.

6. A device has support for addressing 8 locations (i.e. it has 3 address lines). It needs to be connected to an 8 bit bus. What addresses does this device occupy in the memory map if:

   (a) the lower 3 address lines are tied to the device address lines, and the upper five lines go through a decoder which asserts the device select if the five lines read 10101_2

   (b) the upper 3 address lines are tied to the device address lines, and the lower five lines go through a decoder which asserts the device select if the five lines read 10101_2

   (c) the upper 4 lines go through a decoder which asserts the device select if the four lines read 1111_2, the next line is unconnected, and the lower 3 lines are tied to the device address lines.

7. A microprocessor comes in a 24 pin package; if the data-bus width is 8 bits, what is the maximum size of the memory map (i.e. 2^n). Hint: remember the standard pins.

8. (a) Question from Wilmhurst (2001; 2.8):

   > "A small embedded system is powered from a 5V supply. The supply itself cannot source more than 75mA. The total system decoupling capacitance is $220\mu F$. Estimate the time required on power-up for the system voltage to reach 5V. Assume the supply is otherwise ideal, and that the current drawn by the system is small."

   (b) Using the preceding calculation as an example, in your own words, explain why the reset line must be asserted for a certain minimum time after power is applied to the circuitry.

9. Question from Wilmhurst (2001; 2.9):

   > " A micro-controller, with a clock frequency of 1MHz, is to be used for a time measurement application, which must be accurate to within $\pm 0.1\%$ over an operating temperature range of $12°C$ to $30°C$. The designer can choose between a crystal clock source of stability 100 ppm (parts-per-million)/$°C$, a ceramic resonator of stability 800 ppm/$°C$, or an R-C network of stability 0.4%/$°C$. Assuming each source is set initially to give the correct frequency at the mid-range temperature, what frequency range will each clock source give for this temperature range? Which sources, if any, are acceptable for this application? "

10. Role Play/Challenge question:

    (a) " Suggest ways in which:

       i. broken bus/signal wires can be detected by a self-test program in a microprocessor based system.

       ii. 16-bit *data* can be passed to a device over a bus with 8 data lines.

       "

    (b) Most of the devices used in this course are from the CMOS digital logic family. Identify one other digital-logic family and determine what must be done to "correctly" interface devices from the two families.

Tutorial Exercise 2A Offline Version[8]

ID# ____

1. Explain why devices should use either tri-state or open-drain outputs (rather than push-pull outputs) to connect to the bus. *2 marks*

2. Is it possible to have "holes" in the memory map (i.e. addresses to which neither memory nor I/O devices respond)? Explain your answer. *2 marks*

3. Differentiate between signals which are logically high, active high and asserted. You may use diagrams/examples to supplement your written answer. *6 marks*

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

[8]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 3 Your lecturer will put up a diagram with several parts labeled with letters.

Write the letter you have been assigned here________.

Answer the following questions for the part whose letter you were assigned.

1. The part is called the ______________________________.
2. Circle the correct answer. The primary function of the part is:
   (a) to provide an independent trigger signal if it is not periodically reset ☐
   (b) to provide a fixed frequency signal ☐
   (c) to counteract transient voltage/current effects in the supply/ground lines ☐
   (d) to ensure that the reset line is asserted for an appropriate time ☐
3. Identify 2 problems which would arise if the part failed to perform as required, and explain how each of these problems could occur.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the function of basic support circuitry for microprocessors such as clocks, reset circuitry, watch dog timer, capacitors to ground, etc.
  - describe (and/or illustrate) how bus addressing, conflict resolution, and memory maps may be implemented
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

4 Microprocessor architecture

At the end of this unit the student will be able to:

- *differentiate between organization and architecture;*
- *relate instruction set design to the programmers model of the microprocessor;*

> "Computer architecture refers to those attributes of the system *visible to the programmer*, or put another way, those attributes that have a direct impact on the logical execution of a program. Computer organization refers to the operational units and their interconnections that realize the architectural specifications." – Stallings (2000)

When designing a CPU the architectural decisions come first. The CPU *architecture* determines whether the machine instructions and data can be stored in the same memory space, and whether they can be accessed using the same mechanism (von Neumann (same) vs. Harvard (different)). The CPU architecture also determines whether the CPU can manipulate or operate on the data directly from memory, or must load it into an internal location (i.e. working register or accumulator) before manipulations can take place (memory-based vs. load-and-store).

The means by which data may be manipulated by the CPU, can be based on one of several different paradigms. Accumulator based architectures use a single register in conjunction with data from memory (or other registers) returning the result to the accumulator. Register based architectures, provide a small set of registers general purpose registers to the programmer, within which any register may be specified as an instruction argument/destination. *Stack* based architectures, perform operations on the items on the stack, returning the result to the stack. The stack size therefore limits the number of instruction arguments.

CPU organization concerns the way in which the machine instructions are carried out within the CPU i.e. the sequence of micro-operations which actually occur. For example, a multiply instruction can be implemented using shifting and addition in a dedicated multiplier unit; 16 bit addition can be carried out in a single 8 bit adder, in two stages, or in a single 16 bit adder. CPU organization could also involve consideration of the logic family used to implement the design, and related layout issues such as power consumption/heat dissipation and track lengths.

The manufacturing/fabrication process involves alternately placing insulating/metal layers on silicon wafers, and exposing them (using photo-resist coatings) to light projected through a "negative". Areas which are not exposed to light are etched away. The process is often identified by the width of the smallest circuit wires; the most recent processes for commercially available micro-processors operate at 65nm. The advantages of using smaller manufacturing processes include:

- more transistors (processors) per wafer
- operation at lower voltages
- increased clock speeds (due to less power consumption/heat dissipation)

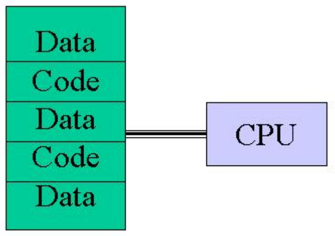
CPU Architecture

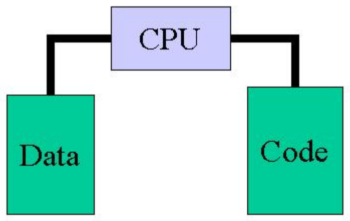
VonNeumann

- code can be modified (accidentally)

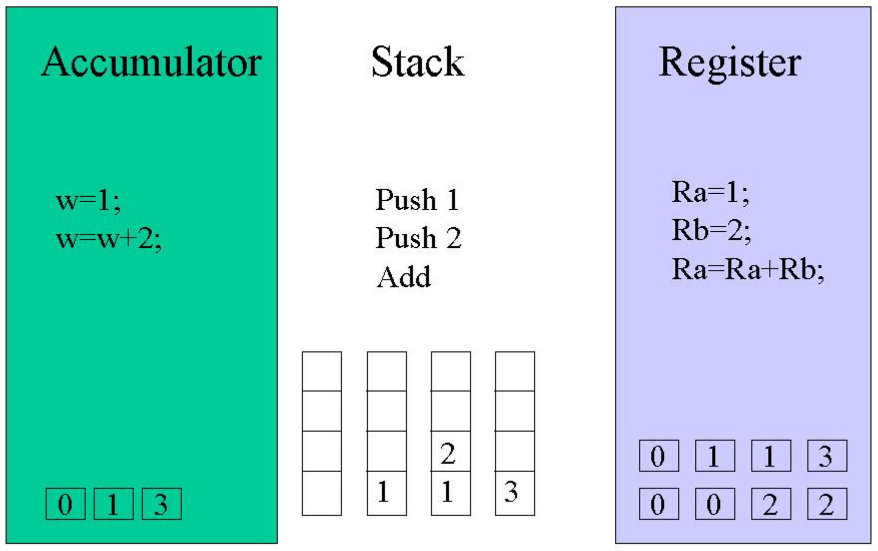
Harvard

- data access does not slow code access
- instruction and data widths unrelated





CPU Architecture



The set of architectural features which the programmer can manipulate via the machine instructions provide the "programmer's view" of the microprocessor; hence in order to effectively program any microprocessor at the level of machine instructions, it is necessary to understand it's particular architecture, but not necessarily the CPU *organization*.

In a similar manner, higher level languages (e.g. C, Basic) tend to mask the programmer from the computer architecture, making it unnecessary to appreciate the characteristics of an individual microprocessor. Whichever language is used by the programmer, it must be converted into machine language, and then somehow placed in memory for the microprocessor to run.

If a machine language program is to be launched each time the microprocessor is reset, then the system must be configured so that the reset memory location (specific to the processor, generally memory location 0000) contains a branch instruction to the start of the program. Alternatively, if the microprocessor is running a program which supports the load/start of a new program (*bootloader* or *operating system*), a reset may not be necessary.

Review Exercises

1. Label the following statements True or False.
   (a) Higher level languages mask the CPU architecture and organisation.
   (b) Harvard architecture CPU's must have a stack.
   (c) Choosing between RISC and CISC is an organisational decision.
   (d) Accumulator-based architectures do not work without a stack.
   (e) The reset vector is not always located at address 0x0000.
   (f) Von Neumann architectures have separate data and program memories
   (g) Stack-based architectures assume the instruction arguments are in the instruction.
   (h) The reset vector is always located at 0x0000
   (i) Assembly/Machine language masks the programmer from the CPU organisation.

2. A programmer knows that there is a multiply instruction in the instruction set of a microprocessor, which takes 4 cycles to multiply two 4 bit numbers, and an add instruction which takes 1 cycle to add two 4 bit numbers. Based on this information he can deduce which of the following (if any):
   (a) the microprocessor ALU contains a 4 bit adder
   (b) the microprocessor ALU contains a 4 bit multiplication unit
   (c) the microprocessor ALU contains a 4 bit multiplication unit that performs partial sums to arrive at it's answer

Architecture vs. Organisation

- Bit Widths
- Instruction set

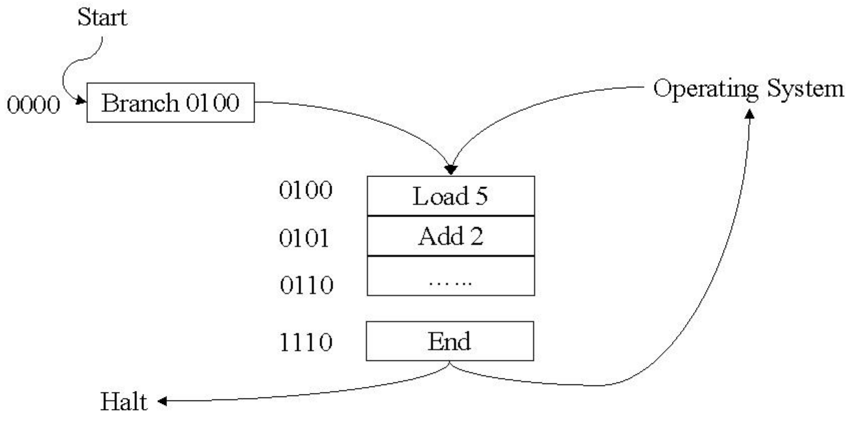
- CPU components
  - bus operations
  - parallel/replicated functions?
  - Pipelining
  - RISC/CISC

- Control signals
- Micro-operation set

- external interface
  - access times
  - RAM: static vs. dynamic
- physical layout
  - manufacturing process
  - logic family
  - track length
  - power/heat considerations

Manufacturing process: http://www.irps.org/irw/98/kn-101/sld027.htm

System Reset / Program Launch



3. A CPU designer wishes to include a multiply instruction in the instruction set of a microprocessor which takes not more then 4 machine cycles to multiply two 4 bit numbers, and an add instruction which takes 1 machine cycle to add two 4 bit numbers. He MUST obey which of the following constraints (if any):

   (a) the ALU needs to have a 4 bit adder

   (b) the ALU needs to have an 8 bit adder

   (c) the ALU needs to have a 4 bit multiplier

4. You are told that a microcontroller has a RISC based instruction set and uses Harvard Architecture. Which of the following can be deduced? *3 marks*

   4-I: the organisation of the processor architecture is not complicated

   4-II: data and instructions are stored in the same address space

   4-III: peripheral registers are memory-mapped

5. The PRIMARY difference between the Harvard and Von Neumann architectures is that Harvard architectures:

   (a) always have CISC-type instruction sets.

   (b) can support pipelining.

   (c) generally support isolated peripheral input-output.

   (d) have separate program and data memories.

   (e) support faster clock speeds.

6. We have three CPU's, which all use the same instruction format. Processor A supports an add instruction. Processors B and C support both add and multiply instructions. Processor B has a 4-bit multiplier in the ALU. Processor C has an 8-bit multiplication unit in the ALU. All processors are accumulator-based and support load and store operations. Processors A and B are microprocessors manufactured using a 180nm process. Processor C was assembled from discrete components. Indicate which of the following statements are TRUE (there may may be more than one):

   (a) Processors A and B support the same instruction set.

   (b) C compiler generated code for Processor A, will also run on Processor C.

   (c) C compiler generated code for Processor B, will also run on Processor C.

   (d) To the assembly language programmer, Processors B and C are the same.

   (e) Processors A and B have the same architecture.

   (f) Processors B and C have the same architecture.

   (g) Processors A and B have the same organisation.

   (h) Processors B and C have the same organisation.

7. Answer the following questions in your own words:
   (a) What is the difference between CPU organisation and CPU architecture?
   (b) Why is the distinction between CPU organisation and CPU architecture necessary?
   (c) When choosing between RISC and CISC are we making a decision about organisation, architecture, or neither? Explain your answer.
   (d) When choosing an IC manufacturing process, are we making a decision about organisation, architecture, or neither? Explain your answer.
   (e) What is the advantage of using the Von Neumann memory structure instead of the Harvard memory structure?
   (f) Is it possible for two devices to have the same address in a microprocessor based system? Explain your answer making reference to the architecture and organisation.
8. A microprocessor comes in a 40 pin package.
   (a) name the standard pins that MUST be on this IC
   (b) if there are equal numbers of address and data lines, what is the maximum size of the memory map
   (c) what does this tell us about the microprocessor architecture and/or organisation.
9. Role Play: You work for a company which is designing a door lock which can be triggered from a cell phone. One of the managers has just heard a sales pitch about RISC and Harvard from a microprocessor company; he wants them used in this product. Form teams which support each of the following combinations for this application:
   - RISC Von Neumann
   - RISC Harvard
   - CISC Von Neumann
   - CISC Harvard

   Each team leader should make a 5-minute presentation stating their case.
10. Challenge:
    (a) Investigate, and then draw/construct a poster to illustrate, the semiconductor manufacturing process.
    (b) View the video "Alpha architecture" by Dick Sites & Dirk Meyer. Identify the architectural, organisational and manufacturing process features described.
    (c) Observe a PC booting, and then starting an application. Explain the sequence of events you observe, and relate it to what you have learnt about the boot and program load processes in a microprocessor.

Tutorial Exercise 2B Offline Version[9]

ID# ____________

1. You are a system designer who has been asked to compare the PIC16F877 and ONE other controller/embedded board. Name the processor you are comparing the PIC16F877 with, and identify at least one difference in the areas shown. *4 marks*

Core/Platform Name	
Architecture	Support Circuitry
Peripherals/Outputs	Memory Map

2. Are the manufacturing process, organisation and architecture of a processor core independent of each other? Explain your answer. You should use an example to make your points. *6 marks*

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

[9] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 4 Write the letter & number you have been assigned here________.
Answer the following questions for the items whose letter & number you were assigned.

1. The item you have been asked to look at for the PIC16F877 is ____________________.
2. This should be determined primarily by the
   (a) (instruction-set) architecture ☐
   (b) (processor-core) organisation ☐
   (c) (integrated-circuit) manufacturing process ☐
3. Where and what did you find about your assigned item?
4. Which additional processor were you asked to look at?____________________.
5. This can be described as
   (a) RISC ☐
   (b) CISC ☐
   (c) Harvard ☐
   (d) Von Neumann ☐
6. Where and what did you find out about your assigned item?
7. Give an example of ONE issue relating to this item which can affect the *programmer* of an embedded system, when using either of these platforms.

Unit 4

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - differentiate between organization and architecture;
  - relate instruction set design to the programmer's model of the microprocessor;
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

5 Typical Microprocessor instructions

At the end of this unit the student will be able to:

- *explain the operation of typical machine instructions, addressing modes, status bits;*
- *infer the consequences of sequences of generic instructions;*

The machine instruction is a unit task which the CPU knows how to perform. The instruction set is the collection of instructions which the CPU understands. Machine instructions may be roughly classified as:

data transfer (move, store, load, exchange, swap, clear, set, push, pop)

arithmetic (add, subtract, multiply, divide, absolute, negate, increment, decrement)

logical (and, or, not, xor, complement, bit-shift, bit-rotate, bit-set, bit-clear, test, compare)

control transfer (skip, branch, conditional branch, subroutine, return, nop, halt, wait)

special purpose (i/o, data conversion, system operation)

There may also be combined instructions which perform one or more operations consecutively(e.g. subtract and branch if not zero). The selection of instructions to be included in the instruction set, and the way in which they may best be encoded will reflect the CPU architecture.

Each instruction retrieved from memory is a binary number of a particular width. This number represents the required instruction, i.e. the instruction has been encoded in some manner. The layout of the encoded instruction is known as the *instruction format.* Within the instruction format are several fields, the first of which (the *op-code*) indicates the operation to be carried out. In order to facilitate easy reading, the opcode is often written in mnemonic form (e.g ADD – addition; JMP – jump (branch)). An *assembly language* program consists of a mnemonic listing of the program's machine instructions. The choice of op-code for a particular instruction is made to facilitate decoding by the control unit. For example, all instructions which use the ALU may be grouped together, so that the ALU can be triggered on the basis of certain instruction bits.

Within the instruction format, the opcode is followed by *operand*s, which the particular instruction uses to perform it's task. Operands may be unused, binary data (e.g. ADD 5), or a reference to a register (ADD R1). The binary data may be used directly, or used to access a memory location. The different ways in which this data may be used are known as the *addressing modes.* The most common addressing modes are:

immediate operand is the data e.g. `load #12`

direct operand is the address of the data e.g. `load [12]`

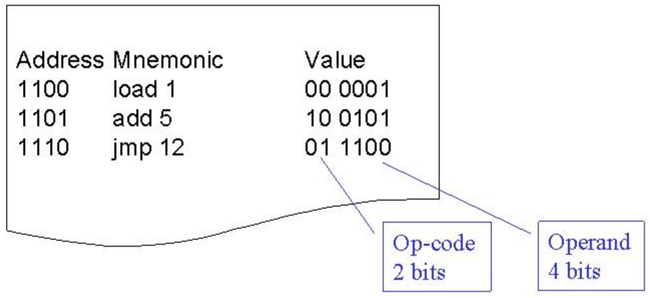
indirect operand is the address of a pointer to the data e.g. `load [[12]]`

register operand is a register; register contains the data e.g. `load r1`

register indirect operand is a register; register contains the address of the data e.g. `load [r1]`

Machine Instruction Examples

Add	• Arithmetic
Load	• Data Transfer
Branch	• Control Transfer



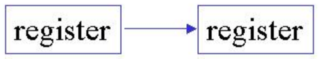
```
Address Mnemonic      Value
1100    load 1        00 0001
1101    add 5         10 0101
1110    jmp 12        01 1100
```

Machine Instructions
Data transfer

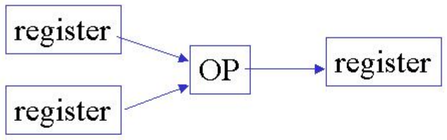
Move	register $\longrightarrow$ register	
Store	register $\longrightarrow$ memory	
Load	memory $\longrightarrow$ register	
Exchange	register $\longleftrightarrow$ register	
Swap	lo-nibble $\longleftrightarrow$ hi-nibble	
Clear	00000000 $\longrightarrow$ register	
Set	11111111 $\longrightarrow$ register	
Push	register $\longrightarrow$ stack	
Pop	stack $\longrightarrow$ register	

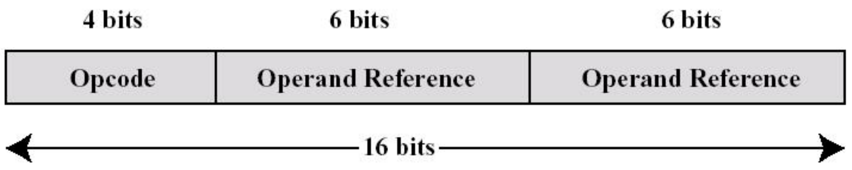
Machine Instructions
Arithmetic/Logical

increment, decrement, not
absolute, negate,
two's-complement



add, subtract, multiply, divide
and, or, exclusive-or





stack no operand is listed in the instruction e.g. `load`

Operands are located on the stack either

1. defined by the stack pointer register
2. or located wholly in register memory

displacement two operands: one of which is a register, and the other a number. e.g. `load r1,#5`

Displacement addressing arguments are combined in one of several ways to form the effective address which is subsequently used to retrieve the data:

1. relative : effective address = current PC + number
2. base-register : effective address = current register value + number
3. indexing : effective address = number (main memory address) + current register value
4. post-indexing : effective address = current register value + current memory location value (whose address is number)
5. pre-indexing : effective address is the contents of the memory location whose address is determined by adding the number to the register contents.

Most microprocessors do not include all of the above addressing modes in the instruction set; however the more complex addressing modes can be implemented using multiple direct and immediate addressing instructions. Where a variety of addressing modes are required, there are two ways in which this may be encoded. The first is to attach the addressing mode to the instruction, i.e. there as many variants of the instruction as there are addressing modes. The second is to attach the addressing mode to the operand. This is more flexible, but requires more room in the operands section of the instruction format.

The number of operands in a particular instruction depends both on the nature of the instruction[10], and the space available within the instruction format[11]. Typically instructions in a set may have different formats, however in RISC architectures (and to facilitate pipelining), it is often advantageous to have a fixed instruction format. A fixed instruction format dictates that all instructions in the set use the same boundaries between op-codes and operands.

The final instruction characteristics to be considered are fixed vs. variable width instructions. The base instruction size is dictated by the bit-width of the CPU/program memory. However, as long as the opcode is contained in the initial instruction word, subsequent words in program memory may be used to provide supplemental data. One example would be for instructions which must refer to memory addresses where the address width is greater than the remaining bits in the instruction format. The advantage of a variable width instruction set, is the space saved for instructions which require no arguments.

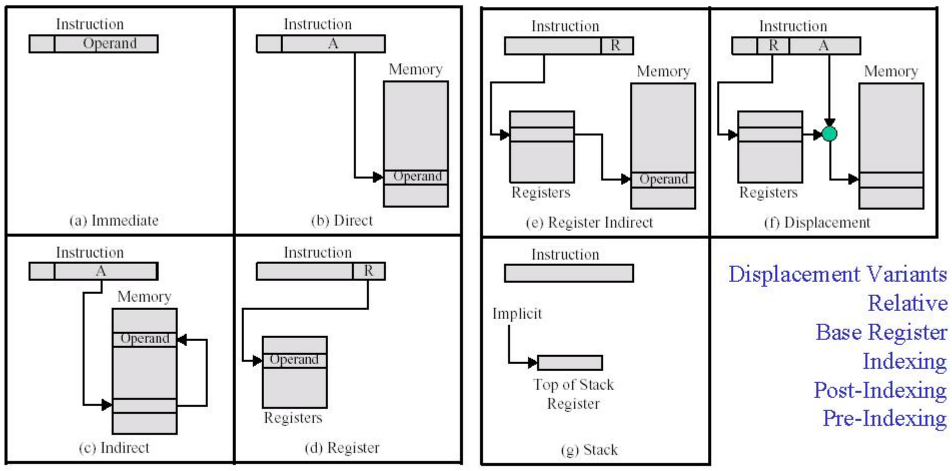
Assembly language is essentially the binary machine instructions (opcode and operands) in which op-codes and the common operands (registers) are replaced with mnemonics/higher base numbers for easy sight reading. As such it is the lowest level at which the microprocessor can be programmed, and requires an understanding of the underlying architecture.

Typical instructions found in assembly language for an accumulator based architecture:

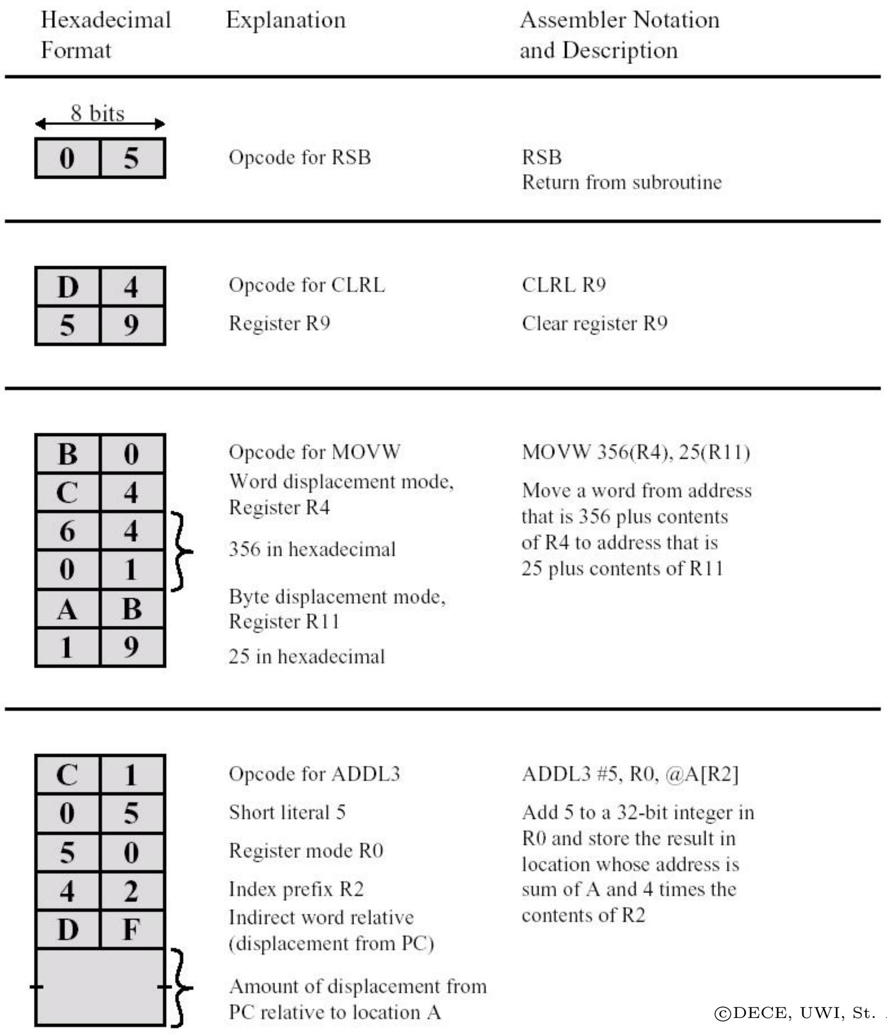
[10] for example the ADD instruction may be specified with no (stack based), one (accumulator based), two or three (memory/register based) operands

[11] for example, in a 16 bit instruction format with a 4 bit op-code, there are 12 bits left to specify operands. These may be used as three 4 bit operands, a 2 bit and a 10 bit operand etc. etc

Addressing Modes



Displacement Variants
Relative
Base Register
Indexing
Post-Indexing
Pre-Indexing



Hexadecimal Format	Explanation	Assembler Notation and Description
8 bits: 0 5	Opcode for RSB	RSB Return from subroutine
D 4	Opcode for CLRL	CLRL R9
5 9	Register R9	Clear register R9
B 0	Opcode for MOVW	MOVW 356(R4), 25(R11)
C 4	Word displacement mode, Register R4	Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
6 4 0 1	356 in hexadecimal	
A B	Byte displacement mode, Register R11	
1 9	25 in hexadecimal	
C 1	Opcode for ADDL3	ADDL3 #5, R0, @A[R2]
0 5	Short literal 5	Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2
5 0	Register mode R0	
4 2	Index prefix R2	
D F	Indirect word relative (displacement from PC)	
	Amount of displacement from PC relative to location A	

load – place the specified value in the accumulator (multiple addressing modes)

store – store the accumulator value at the specified position (multiple addressing modes)

add – add the specified value to the accumulator

jmp – change the program counter to the specified value

One architecture-dependent special function register, which the programmer needs to interact with, is the status (or condition code) register. The bits in this register (status flags) individually represent a particular condition. The conditions represented, and the order in which they appear are architecture dependent. Typically they reflect the results of the ALU operations and/or any internal CPU error conditions (e.g. stack overflow; ALU carry; ALU zero). Note: flags tend to remain on (set; 1) until another operation explicitly turn them off (reset; 0). In particular ALU flags will remain in the same state if the following instruction does not utilise the ALU.

The status flags may be used in several different ways:

- the CU may use them to execute conditional machine instructions; the programmer merely uses the conditional instruction.

  ```
  load  a
  add   5
  addnc 4
  ```

- the programmer may explicitly test their value using a special status testing instruction which returns a result or sets alternate flags.

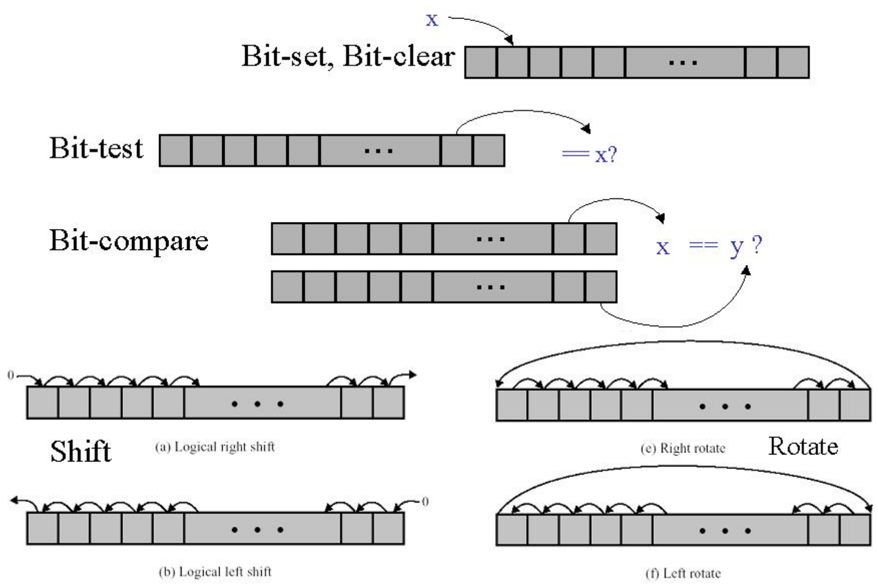
  ```
  load  a
  add   5
  store a
  tstc
  jmpnz   .....
  load a
  add  4
  .....
  ```

- the programmer may explicitly test an individual flag in the status register by applying a bit-mask to, and/or shifting, the register.

A *bit mask* contains a 1 in the position of the bit we wish to test, and 0's elsewhere. To determine if a bit is set, *bitwise-and* the status register with the bit mask. The result will be non-zero only if the flag was originally set.

In some cases, bits in the status register must be explicitly manipulated by the programmer. To set a bit, *bitwise-or* the status register with the bit mask. To toggle a bit (off$-$ > on; on$-$ > off) *bitwise-xor* the status register with the bit mask. To clear a bit, *bitwise-and* the status register with the inverted bit mask (you can invert the mask using bitwise-xor first).

Machine Instructions:Bit--Logical



Applying Bit Masks

Flags Register

`10100111`

76543210

	Bit 5 Mask
	00100000
Extract bit 5: flags & mask	00100000
Set bit 5: flags | mask	10100111
Clear bit 5: flags & NOT(mask)	10000111
Toggle bit 5: flags ^ mask	10000111

Review Exercises

1. Label the following statements True/False:
   (a) Immediate addressing requires no memory referencing, but the range of values is limited by the operand space in the instruction. ☐
   (b) Indirect addressing potentially allows access to a large addressing space, but requires multiple memory references. ☐
   (c) The branch (jump, goto) instruction changes the program counter value, only after storing the previous value. ☐
   (d) The noop instruction has no effect other than incrementing the program counter. ☐
   (e) Register addressing requires two memory references. ☐
   (f) The push and pop instructions are used to manipulate the accumulator. ☐
   (g) The skip instruction is the same as the call (subroutine) instruction. ☐
   (h) The range of addresses which are accessible using Direct Addressing, is limited by the operand space in the instruction. ☐

2. The following values are in the data memory of a Harvard machine with an accumulator.

Address	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2A
Data	0x44	0x10	0x02	0x53	0x43	0x23	0x11	0x34	0x23	0x38	0x22

   What value is in the accumulator after each of the following instructions are executed in sequence:

```
LOAD IMMEDIATE 22
LOAD DIRECT 25
ADD IMMEDIATE 27
ADD INDIRECT 2A
```

   Your answer should show your reasoning.

3. The question refers to the following program which runs on a machine with an accumulator:

```
Line 1             load 5
Line 2   begin:    sub 1
Line 3             jumpnz Begin
```

   Write down the line numbers, and accumulator values as the program executes.

4. (a) What is the difference between a program branch, and a subroutine call?
   (b) Branch instructions move to the specified instruction if the test condition is fulfilled. Skip instructions miss the next instruction if the test condition is fulfilled. Why do we have a skip instruction when we already have a branch?
   (c) Add instructions add a value to the specified register. The increment instruction simply adds one to a register. Why do we need an increment instruction when we have an add instruction?

(d) The nop(no operation) instruction has no effect other than incrementing the program counter. Suggest some uses of the nop instruction. (Stallings 2000; 9.5)

(e) How can CPU use a stack for any purpose, if there are no push and pop operations in the instruction set? (Stallings 2000; 9.7)

(f) "The number of machine instructions in an instruction set is limited by the number of bits used to encode the instruction, and the instruction format." Is this statement true or false? Explain your answer.

(g) How would you perform an indirect addressing operation, using direct and immediate addressing mode instructions?

5. Assuming a single address instruction, and a processor with an accumulator, use the "standard" instructions you have learned here to convert the following piece of C code into machine instructions.

```
a=4;
do
{
   a=a+5;
}while (a<25);
```

You should check your answer by writing down the output from the C code, and your assembly language code.

6. Convert the following machine language instructions into C code.

```
    load    #0
    store   c
    load    a
loop:
    inc     c
    sub     b
    jmpovf  end
    goto    loop
end:
    dec     c
```

Your answer should explain what this piece of code does.

7. Assuming a single address instruction, and a processor with an accumulator, use the "standard" instructions you have learned here to convert the following piece of C code into machine instructions.

```
a=0;
for (i=0;i<5;i++)
{
   a=a+i;
}
```

8. Convert the following machine language instructions into C code:

```
loop:
    load a
    add b
    store a
    jmpc end
    goto loop
end: ....
```

9. Role Play:

   (a) What are the relative advantages and disadvantages of each of the addressing modes? (Hint see: (Stallings 2000; Table 10.1))

   (b) Differentiate between fixed length and fixed format instruction sets. What are their respective implications for CPU organisation, architecture and operation?

   (c) Assume an instruction set that uses a fixed 16 bit instruction length. Operand specifiers are 6 bits in length. There are K two-operand instructions and L zero-operand instructions. What is the maximum number of one-operand instructions that can be supported? (Stallings 2000; 10.9)

   (d) Is there any possible justification for an instruction with two opcodes? (Stallings 2000; 10.12)

10. Role Play/Challenge question:(Stallings 2000; 9.4)

    "Consider a hypothetical computer with an instruction set of only two n-bit instructions. The first bit specifies the op-code, and the remaining bits specify one of the $2^n - 1$ n-bit words of main memory. The two instructions are as follows:

    SUBS X Subtract the contents of location X from the accumulator, and store the result in location X and the accumulator

    JUMP X Place address X in the program counter

    A word in main memory may contain either an instruction or a binary number in twos complement notation. Demonstrate that this instruction repertoire is reasonably complete by specifying how the following operations can be programmed:

    (a) Data transfer: Location X to accumulator, accumulator to location X

    (b) Addition: Add contents of location X to accumulator

    (c) Conditional branch

    (d) Logical OR

    "

Assignment A [12]

ID#

1. An 8-bit processor-status register is configured so that bit-2 is the carry flag, and bit-7 is the zero flag. Write down the bit mask for each flag, and explain how they can be combined, and then used, to simultaneously detect if both bits in the status register are clear. Use examples, and counter-examples, to support your answer. *8 marks*

2. The binary number 11100010_2 is subjected to the following operations on an 8-bit processor core: *4 marks*

```
arithmetic shift right
bit-wise xor $0011 1110_2$
rotate left
logical shift left
```

What is the resulting value? Show all working.

[12]Students are advised that there is no online version of this assignment, and that electronic submissions will not be accepted without the explicit permission of the course lecturer.

3. A CPU designer has decided that his new CPU core will use an 6-bit wide instruction memory, and have a variable-width instruction format, with single operand instructions. The CPU core will have a internal stack, and will be supplied in a 14-pin IC package.

   (a) The support pins required to allow the IC to function include all of the following EXCEPT: *1 mark*

      i. 1 pin for clock signal
      ii. 2 pins for power supply
      iii. 1 pin for reset line
      iv. 1 pin for watchdog timer

   (b) The decision to have an 8 bit wide instruction memory, is a decision about: *1 mark*

      i. CPU architecture
      ii. CPU manufacturing
      iii. CPU organization
      iv. none of the above

   (c) The instruction format MUST: *1 mark*

      i. contain an operand which requires less than 6 bits.
      ii. include `push` and `pop` instructions for the stack.
      iii. specify a maximum of 2^6 unique instructions.
      iv. support at least one form of addressing.

   (d) Assuming that operands are at least 5 bits wide, and that instructions never require more than 2 instruction memory words, calculate the maximum number of op-codes that can be included in the instruction set. Show all working. *4 marks*

4. The following values are in the data memory of a CISC machine with an accumulator.

Address	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2A
Data	0x44	0x10	0x02	0x53	0x43	0x23	0x11	0x34	0x23	0x38	0x22

What value is in the accumulator after each of the following instructions are executed in sequence:

```
load direct     0x22
add  immediate  0x27
store indirect  0x25
load immediate  0x20
add  indirect   0x28
```

Your answer should show your reasoning. *5 marks*

5. The question refers to the following program which runs on a stack-based machine, that has a carry-flag. Labels are used to refer to the *address* of the instruction.:

Address	Label	Instruction
0x00		`push` 1110_2
0x01	`label_begin:`	`push` 0100_2
0x02		`add`
0x03		`push label_end`
0x04		`jump-if-carry-set`
0x05		`push label_begin`
0x06		`jump`
0x07	`label_end:`	`pop`

As the program executes, write down the address of the instruction executed, line numbers, the state of the carry flag (if known) and the values stored on the stack, Your answer should show your reasoning. *6 marks*

6. The instruction set for an accumulator-type Harvard-architecture microprocessor with 8-bit instructions, 6-bit addresses, and 4-bit data is supplied. The C/C++ subroutine, to calculate base-2 logarithms, must be compiled so that the final assembly language routine starts executing from address 0x12.

Instruction set:

	Description	Format
`add`	Add the contents of the accumulator to the operand; Result in accumulator; Sets/Clears zero and carry flags	0010 nnnn
`clf`	Clear all flags	0001 0010
`jnc`	Jump to the specified INSTRUCTION address if the carry flag is not set	01qq qqqq
`lod`	Load the number from the specified DATA address into the accumulator	10qq qqqq
`nop`	No operation	0000 0000
`rtn`	Return to the calling routine (return value should already be in accumulator)	0001 1100
`sbt`	Subtract the operand from the contents of the accumulator; Result in accumulator; Sets/Clears zero and carry flags	0011 nnnn
`shl`	Logical shift left contents of accumulator; result in accumulator	0001 1011
`shr`	Logical shift right contents of accumulator; results in accumulator	0001 1010
`str`	Store the number from the accumulator to the specified DATA address	11qq qqqq
`zer`	Zero the contents of the accumulator; Clears the zero flag	0001 1000

```
NIBBLE base2log_routine(NIBBLE c)
{
    BYTE a,b;

    b=0;
    a=c;
    while (a>=2)
    {
          a=a/2;
          b=b+1;
    }
    return b;
}
```

(a) Manually translate the C/C++ code into assembly language: *5 marks*

(b) Encode your assembly language using the op-codes supplied. Indicate the memory locations used for variables and branch destinations using labels. Place the final values into the data memory locations, presuming that routine was called with the value 0x9 as the function argument. *5 marks*

Data memory				
0x10				
0x11				
0x12				
0x13				
0x14				

Instruction memory								
0x10								
0x11								
0x12								
0x13								
0x14								
0x15								
0x16								
0x17								
0x18								
0x19								
0x1A								
0x1B								
0x1C								
0x1D								
0x1E								
0x1F								
0x20								
0x21								
0x22								
0x23								
0x24								
0x25								
0x26								
0x27								
0x28								
0x29								
0x2A								

This assignment is worth 2% of the 40% assigned for the ECNG2006 coursework mark.
It contains a total of 40 marks.

Unit 5 Write the letter you have been assigned here________.
Answer the following questions for ONE instruction whose letter you were assigned.

1. The instruction is ____________________________.
2. The instruction has ____________________________ operands.
3. The format of this instruction is: ____________________________.
4. Write a sentence which explains what the CPU does in response to this instruction.
5. Write a short program to demonstrate how the instruction can be used by the *programmer* of an embedded system.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the operation of typical machine instructions, addressing modes, status bits;
  - predict the results of sequences of generic instructions;
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

Microprocessor Overview

This section presented a *highly* simplified view of microprocessors and microprocessor-based systems. The more astute student will recognise that the text contains several statements which were never, or are no longer, strictly true.

Among the more obvious:

- The RISC–CISC border has blurred; in most cases features from both are merged. In fact RISC processers are rapidly being outstripped by superscalar and parallel(e.g Dual core) architectures.
- We have not mentioned SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction ...) processors
- Arbiters and DMA controllers have merged. In fact burst mode DMA, utilises the DMA mechanism for sending data over more than one bus cycle.
- Memory is no longer directly accessed over the system bus: it may have a dedicated bus, (e.g. RAM Bus) or Level I/II cache may be used (faster, smaller memory banks "closer" to the microprocessor)
- Peripherals are becoming more intelligent, with a variety of peripheral buses (w/ controllers) being used to transmit data independently of the system bus.
- Due to increasing performance demands, and shrinking transistor size, CPU design is now an extremely complex field. The average chip designer is resorting to the use of predefined CPU parts in a custom SoC (system-on-chip), which includes other circuitry.

The purpose of this section was to address part of the first course objective; i.e. "identify, and describe the role of, the components of a microprocessor, a microprocessor-based system," and as such it is sufficient, if you **the student possesses sufficient appreciation of the parts/mechanisms to facilitate assembly-language programming/debugging.**

The following section will further address the first course objective.

Part II
Microprocessor-based system development

The previous section talked about the hardware components which make up a microprocessor, and a microprocessor based system, and the ways in which they interact. However, there are other issues to choosing/designing a microprocessor; primarily, the issue of developmental support. This support manifests itself in several ways: documentation, software development tools and hardware development tools.

This part gives a brief introduction to these topics, to familiarise the reader with the terminology, as well as common features and techniques.

6 Software tool chain

At the end of this unit the student will be able to:

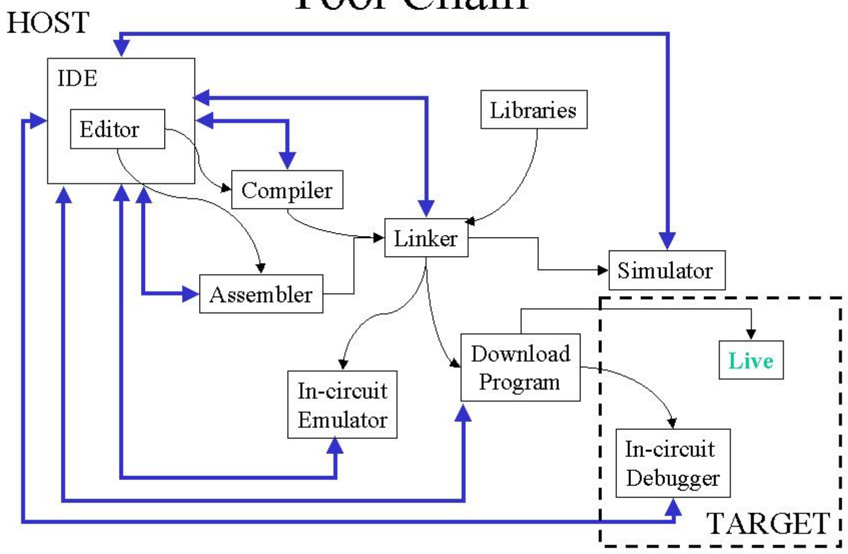
- *explain the roles of the compiler, linker, assembler, simulator, emulator, debugger in the software development tool chain,*
- *explain the role of scripting languages, make/configure-like utilities and IDEs in the software development process.*

When a program is written as text, it must first be *compile*d (high level language) or *assemble*d (assembly language). The compiler(or assembler) is a special program which converts the text into *relocatable* machine instructions. Compilers and assemblers understand two types of commands: programming statements, and pre-processor directives. Programming statements are code i.e. what you want the microprocessor to do. Preprocessor directives are simple instructions which run prior to the actual compiler(assembler); they make the programmer's life easier. For example, the "#define" in a C program is a preprocessor command, which replaces the specified text string throughout the file.

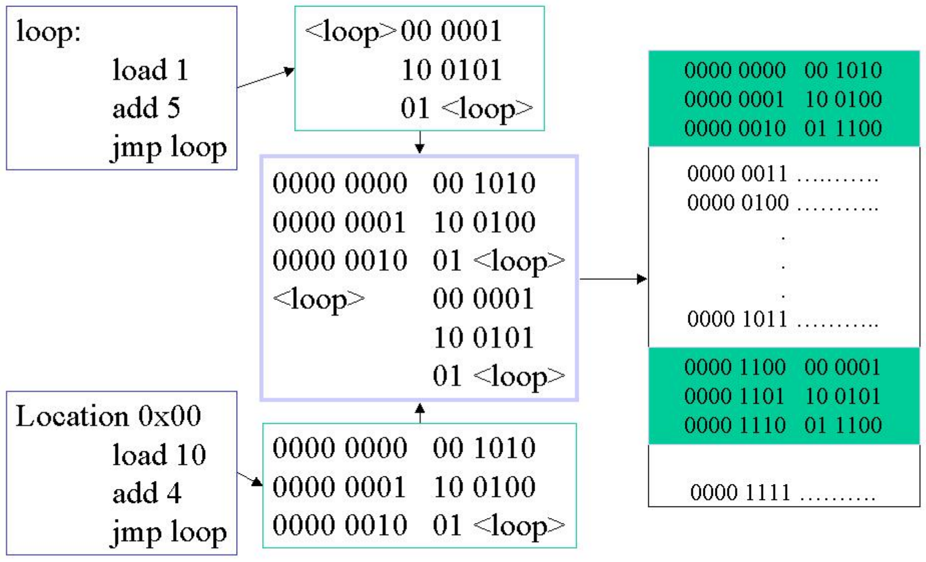
The microprocessor-based system developer must decide which language, assembly or one of the higher-level languages to program in:

> "The factors relevant to a language decision probably include at least:
>
> - Efficiency of compiled code
> - Source code portability
> - Program maintainability
> - Typical bug rates (say, per thousand lines of code)
> - The amount of time it will take to develop the solution
> - The availability and cost of the compilers and other development tools

Software Development Tool Chain



Relocateable code



- Your personal experience (or that of the developers on your team) with specific languages or tools

" (Barr 2000; – from Language Lessons by Michael Barr, Copyright 2000 by CMP Media, Inc.)

If the *host* and *target* are not the same type of microprocessor, a special cross-compiler or cross-assembler must be used (because the host will not understand target instructions and vice versa). The compiled (assembled) file from must be linked before it can be used by the target. Linking is used to provide several facilities, namely multi-file programming and/or use of *libraries*/pre-written routines. The Run Time library is a collection of standard C functions e.g. `printf` compiled for the particular tool-chain/target platform. The linker takes the various sets of relocatable code, and generates the final executable which can be loaded into memory, or run by a simulator/emulator. To do so, the linker decides where code and variables will be placed in memory, and resolves addresses for items referenced in multiple modules. The linker must be aware of the target's memory map, in order to produce a valid *executable* or *binary* file. Information about the memory map is often specified in an additional text-file which is specified when the linker is invoked.

The final binary file will contain machine instructions, and information about the specific locations in memory to which they are to be loaded. Because the file contains code for the target processor, it cannot be run on the host, unless we use another program to interpret the target code. Instruction Set*Simulators*, and *emulators* are similar in that they may both run on the host processor, and accept code produced for the target processor. However, simulators do not interact with hardware, they run within a "virtual target" environment on the host. Whilst emulators, possess a hardware interface, so that they may physically replace the target "in-circuit".

In both cases (simulator and emulator), the objective is to be able to "view" what is going on "within" the target during program execution; this assists the programming in "debugging" i.e. locating portions of code which do not perform as expected/specified. Techniques typically employed in debugging include:

- use of breakpoints to pause the program
- single stepping through program
- "watch"ing or setting specific values in memory

An alternative to simulators, and emulators, is to actually build support for debugging into the target software. When the program code is downloaded to memory, additional code (*debug kernel*) is also downloaded. Using this additional code, the target will communicate with a program on the host (*debugger*) via serial or network connections, and report the state/change of state in the target as code executes. The advantage of using standard debug kernels, is that the same host debugger software may be used for mutliple target microprocessors. Examples of debug kernels include: the Angel debug monitor, and the gdb debug stub.

All three options have their relative advantages and drawbacks. Simulators (because they run much more slowly), cannot cope with or be used to troubleshoot problems due to interaction with external devices/timing. They can however be used to debug/predict execution times of particular pieces of code which have no external interaction. Emulators also tend to run at lower speeds than the physical target; emulators which can match target speeds are expensive compared to other development tools. Both emulators and simulators are designed to match the target specifications and cannot identify problems which arise from targets which somehow violate their specifications (hardware or software). In-circuit debugging, will take CPU time and other resources (e.g. interrupts), and cannot be performed without affecting the performance of the code under test. In practice, each method is applied (as appropriate) to deal with a sub-set of problems, and any remaining problems must be dealt with using hardware-based troubleshooting methods.

Microprocessor software development tools are generally designed to work in conjunction with each other. The term *tool chain* is often used to collectively refer to assemblers, compilers, linkers, libraries, simulators, emulators and debuggers. These are all command-line tools, i.e. in order to run the tool, you must type the name, and provide it with a list of options before it will run. Repeatedly typing in command lines can become tedious, thus many tool chains also include other programs to facilitate the development process.

These “helper” programs can be roughly subdivided into: scripts or batch files, makefiles, window-based IDE’s . Scripts or batch files are simply sequences of commands, which could have been typed in at an interpreter prompt. Often they support replacement of wildcards by parameters specified when the script is invoked. They can be configured to stop if any command gives an unexpected result. In this context, the script file would contain a sequence of tool-chain commands, with wildcards used for the program name(s).

One popular tool coming from the UNIX development environments is the *makefile*. Makefiles operate on a similar principle to scripts, except that they are especially designed for software development, and are executed by the make utility. Makefiles consist of a list of rules and dependencies for individual makefile-targets. If a makefile-target does not exist, then make checks whether all the dependencies exist. If a dependency does not exist, it looks for a rule/depedency list to make the dependency. Once all dependencies exist, the rule is executed. Make also checks the date/time-stamps on files; if the dependencies are time-stamped later than the makefile-target, then the rule is executed. Make is invoked with a single makefile-target By listing all the required makefile-targets as dependencies of a single makefile-target, we can cause all files to be built in the required order.

Integrated Development Environments(*IDE*s) are window-based software packages designed to ease the use of the various software tools required to perform program development. They generally provide text editing (tailored with highlighting/syntax checking), project files to keep track of all the contributing files, and their relevant tool settings, and launch menus for the various items in the tool chain. They may also present the results of these tools for easy correction/review of the text program. For example, after compilation, a window with the errors comes up to prompt the user for changes. Additional features may include a (make-like) build, version control, and code generation from software design tools.

When choosing a compiler or tool-chain (tool-set), the following issues must be considered:

“

Target Platform The first step in selecting a cross compiler is finding one that will produce code for your target processor.

Host Platform The next step is to decide on a development platform. If there are several platforms available, you may want to check some of the other items in this list before making a decision about the host.

Integration with Other Tools Is the compiler compatible with any debugging environments? Is a make utility included? If the compiler is shipped with an IDE, is it extensible so that you can integrate your version control tool?

Standard Libraries Will you need functions from the standard C library, math library, or C++ classes? If so, are they provided with the compiler? Are all of the functions in those libraries reentrant?

Startup Code Is startup code for embedded systems provided? Are the code and its use well documented? If you can't find any mention of startup code in the user's manuals for a potential cross compiler, consider that a bad sign.

Execution Speed Optimizations If your program is too slow, you'll want the compiler to try to speed it up. Will the compiler do this? If so, what specific optimizations are supported? Can they be individually enabled or disabled?

Program Size Optimizations If your program is too big for your target memory, you'll want the compiler to attempt to reduce the amount of code space used. Will the compiler do this? If so, what specific optimizations are supported?

" – (Barr 1999; Extracted from Table 1. Checklist for Cross Compiler Selection Target Processor)

Review Exercises

1. Write a definition for all italicised terms in this unit.
2. Differentiate between:
   (a) an emulator and a simulator.
   (b) an interpreter and a compiler.
   (c) a script and a makefile.
   (d) a library and a linker.
3. Label the following statements about the software tool-chain as True/False:
   (a) The IDE provides provide a common editor/interface for other elements of the software development tool chain. ☐
   (b) The assembler takes assembly language text and optionally produces object files. ☐
   (c) The compiler only runs on the target platform, and generates code for the host platform to run. ☐
   (d) The compiler takes output from the assembler, and produces the executable file. ☐
   (e) The debugger runs on the host, and communicates with the target debug kernel code. ☐
   (f) The difference between the simulator and the emulator is that the simulator runs on the host, and the emulator runs on the target. ☐
   (g) The host and target platforms are always different. ☐
   (h) The linker is never used when the host and target platforms are different. ☐
   (i) The simulator runs on the host platform, and executes code compiled for the target platform. ☐
   (j) The simulator takes assembly language text as input, and runs on the target. ☐
   (k) The tool-chain is a group of programs used to facilitate the development of programs for a particular target(s) on a particular host. ☐
4. From (Wilmhurst 2001; 12.3)

> “ You are employed in a small company whose products incorporate PIC microcontrollers. You are the company’s only development engineer, and you wish to persuade your manager to agree to the purchase of an in-circuit emulator. Your manager, however, knows that you have ...[a simulator] already, and believes (perhaps wrongly) that if you can simulate a program then you don’t need the emulator. Write a justification for your proposed purchase stating clearly the advantages that an emulator would give. ”

5. In addition to pre-processor directives, C compilers often support special keywords, which are not part the standard C. One example is the `asm`/`endasm` keyword pair used to delimit blocks of assembly language code within a C program. What possible advantage could this non-standard keyword offer?

6. “Decompiling is the process of generating source-level code from the executable object code embedded in a product’s memory”. (Fisher 2000) This can be a serious problem for developers of proprietary systems. Investigate this issue, and identify one way in which code can be protected from being decompiled.

7. In your own words, explain why the same C code, when compiled by different compilers may generate different size/speed executables.

8. (a) Explain how scripts may be used to automate testing of the software, as dictated by the client, both in the absence of the hardware, and after the hardware becomes available.
   (b) Investigate and identify 2 scripting languages commonly used in software development.

9. Explain what a debug kernel is. Your answer should mention ways in which a debug kernel can facilitate testing/fixing a micro-processor-based system.

10. Role play: Assembly language programming skills are often valued, because of the perception that the assembly language programmer will always generate more efficient code. Form teams which will debate this issue from the following viewpoints:
    - In (Barr 2000), Michael Barr describes a case where the C program was faster than the assembly language program: “The speedup was actually the result of a better design.”
    - “Assembly will always be at least as fast as C when executing the same algorithm.” (Barr 2000)

Tutorial Exercise 3A Offline Version[13]

ID# ______

Congratulations! You have been assigned to the NOVATE company as a summer trainee in the software development division. NOVATE has recently been granted a contract to develop an embedded system which monitors oil flow in pipelines. The system will provide an estimate of the oil transported via the pipeline, as well as automatically detect changes in flow (e.g. pipeline breaks). The sensor and microprocessor hardware must be specially developed for solar power consumption, and adverse environmental conditions, and will not be available for several months, although a functional/circuit diagram for the system is currently available. You must develop the software and verify that it operates correctly in different (contract-specified) scenarios.

1. At this time, in this project, which of the following items could you use to DEVELOP your software? *5 marks*

   (a) compiler ☐
   (b) IDE ☐
   (c) libraries ☐
   (d) linker ☐
   (e) loader ☐

2. Choose the word combination which best completes the following sentence: *1 mark*

   Within the development tool chain, the simulator runs on the __2I__ and takes __2II__ as input, while the online debugger runs on the host, and communicates with the __2III__.

   (a) 2I: host 2II: assembly language text 2III: target emulator
   (b) 2I: host 2II: binary file (hex file or executable)2III: target debug kernel
   (c) 2I: host 2II: binary file (hex file or executable)2III: target emulator
   (d) 2I: target 2II: assembly language text 2III: target debug kernel
   (e) 2I: target 2II: binary file (hex file or executable)2III: target emulator

3. For this project, how much development and testing can take place before the hardware becomes available? You may use drawings/diagrams to express your answer, if you prefer. *4 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

[13] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 6 Your lecturer will put up a diagram with several parts of the software development tool-chain labeled with letters. Answer the following questions for the part whose letter you were assigned. Write the letter you have been assigned here________.

1. The part is called the ______________________________.
2. Circle the correct answer. The primary function of the part is:
   (a) to convert assembly language text to relocatable machine code .
   (b) to convert high level language text to relocatable machine code .
   (c) to convert relocatable machine from many files into a single binary/executable file .
   (d) to store relocatable machine code for commonly used functions, specific to a tool-chain, high-level language and/or target. .
3. Write a sentence which explains how this part of the software tool-chain operates, with reference to the host and target machines.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the roles of the compiler, linker, assembler, simulator, emulator, debugger in the software development tool chain,
  - explain the role of scripting languages, make/configure-like utilities and IDEs in the software development process.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

7 Development support (hardware)

At the end of this unit the student will be able to:

explain the role of

- *firmware, operating systems, oscilloscopes,logic probes/analyzers,*
- *terminals, disk drives, external memory, host computers debug protocols e.g. JTAG*

in the development and support of microprocessor based systems.

In the previous unit, we looked at the software needed to perform microprocessor-based system development. Some of that software, (for example emulation) works in conjunction with specialist hardware. In this unit we examine some of the more hardware-related items required to support development.

The first two items *firmware* and *operating system kernels* while software, are strongly tied to the hardware upon which they run. Firmware is software stored in a non-volatile medium e.g. EPROM, and is typically where the self-test, boot-loader and/or BIOS code is stored i.e. software that needs to run when the system is first powered up. For EPROM/flash media, firmware may be programmed using a dedicated *universal or chip programmer*, and then the media may be placed in circuit; the newer flash chips support serial programming while the chip is in circuit.

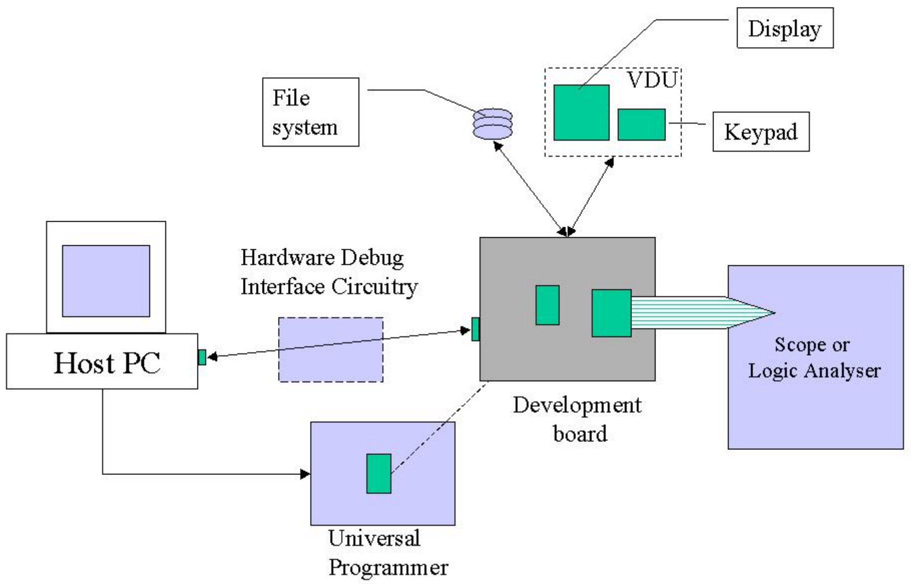
Operating system kernels may be stored as firmware, or may be stored on volatile media, and loaded into memory by the firmware. Both firmware and operating systems are written for specific platforms, with known memory maps. They will not work on different platforms. A target microprocessor-based system, need not necessarily have an operating system installed. In dedicated single function applications, the program is compiled down to a an absolute binary which is placed in memory starting at the reset vector. Where the operating systems on the host and target differ, we must ensure that they both interpret file(s) in the same manner. e.g. UNIX and Windows operating systems interpret ASCII control characters differently.

Development boards, are circuit boards fitted with a microprocessor, support circuitry for the more commonly used peripherals, and expansion space for customisation. They are useful to the microprocessor-based system software developer, as they give the programmer something to test code on/with prior to obtaining the custom board being developed for the application. This means that application hardware and software can be developed in parallel reducing overall development time.

In order to troubleshoot timing and signal problems involving a microprocessor, it is necessary to examine the signals at the pins (Note: we cannot get inside) using an *oscilloscope* or *logic analyser*. Generally, an oscilloscope can show only 2 channels (2 pins) simultaneously. The logic analyser however allows the user to capture either timing (continuous sampling at the logic analyser frequency) or state information (sampling at the system clock frequency) on multiple pins. Because of the quantity of information that the logic analyser captures, it needs to have large data buffers, and provide facilities to capture selective data (trigger conditions), filter the captured data, and/or download to a PC for analysis. Some logic analysers can even list the program begin executed, simply by recording the data passing across the microprocessor bus lines, and *disassembling* the instructions.

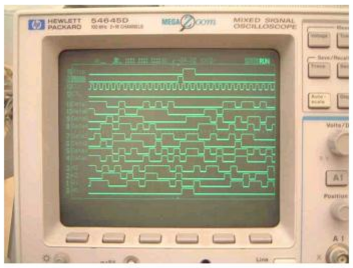
Logic analysers have two main drawbacks. The first is the physical difficulty of making multiple connections on dense pin packages. Special connectors are often delicate, and accidental shorts are

Hardware Development



Development Tools

http://www.pjrc.com/store/dev_pcb_assem.jpg


http://www.cl.cam.ac.uk/~rnc1/descrack/lascreen.jpg

http://www.artbv.nl/ images/jpg/up1.jpg

http://home.att.net/~terminals/images/digital-vt420.jpg

always possible. The second is the fact that signals within the IC cannot be examined. This may not be an issue for microprocessor-only IC's, but where ASIC's or SoC's utilise multiple parts (e.g. memory, and CPU) on a single IC, hardware troubleshooting becomes difficult. Furthermore, the disassembly technique will not work if a microprocessor has on-board cache memory.

To address these problems, the *JTAG* (Joint Test Action Group) protocol was developed. This serial protocol is based on the concept of a simple sampling circuit connected to each "virtual pin" within the IC. All the sampling circuits are connected in a loop, which in turn acts as a large shift register. The state of all the virtual pins at any time, may be obtained by shifting out the value. Similarly the value at any virtual pin may be set by shifting in a particular value.

In the previous unit, we mentioned that where host and target are separate machines, the debug kernel on the target communicates with the debugger on the host. Alternatively, (or in conjunction w/ the debug kernel) there may be *built-in hardware support for debugging.* This is referred to as *in-circuit debug*, or *background debug.* Two examples are Microchip's OCD, and Motorola's BDM. Specialty hardware within the IC can monitor/query the processor state, match a single breakpoint, or stop execution when a signal is detected on a pin. The hardware is programmed/reports serially across dedicated pins.

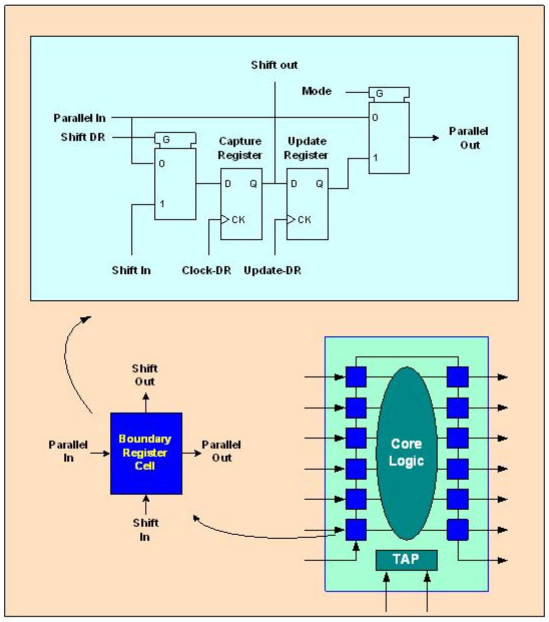
In order to support more sophisticated functions using the basic hardware debug functions, interface circuitry is used between the target microprocessor and host circuitry. The interface circuitry (often containing another microprocessor), generates the appropriate hardware commands, and logs/buffers information reported by the hardware.

Debugging with a debug kernel and debugger, has the disadvantage that the debug kernel code may somehow interact with the application code. This is not a difficulty for development. Both JTAG and hardware debugging support methods have the advantage that each final product (with optimized code) may be tested for hardware and/or software flaws before it is shipped.

The target system often has a rudimentary command line interface, which necessitates the use of a keyboard and display. The *terminal* is a device with a serial connection, which is designed to take keyed input, and relay it out as ASCII data, and display ASCII data transmitted back to it. It is extremely rare to find text terminals e.g. VT100 in use these days, however most operating systems on host computers will allow the PC to emulate a terminal (e.g. hyperterm).

The target system may also require a rudimentary *file storage system.* Random Access filing systems, originated with disk-shaped magnetic media, and the terminology (tracks, sectors, blocks) reflects this. The actual file data is stored in a number of blocks. The FAT (File Allocation Table) (stored at a known location on the medium) is a list of records representing the information stored at each block. Each record specifies the following block. Filing systems for solid media (Flash, RAM, ROM) follow the same format, so that they may be be used interchangeably.

http://www.corelis.com/Introduction%20to%20JTAG.htm



File storage

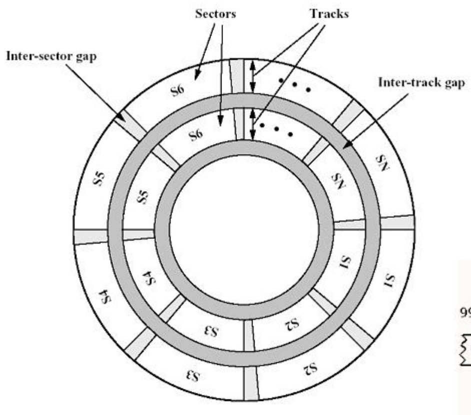
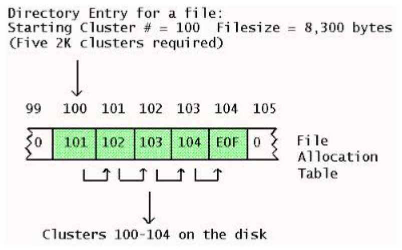
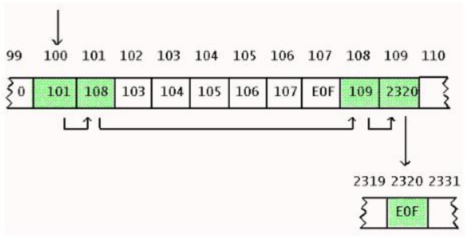


Figure 5.1 Disk Data Layout



```
Directory Entry for a file:
Starting Cluster # = 100  Filesize = 8,300 bytes
(Five 2K clusters required)
```



Stallings COA5e http://www.edm2.com/0410/hpfs1.html

Review Exercises

1. Write a definition for all italicised terms in this unit.
2. Label the following statements as True or False.
   (a) A PC may be used to download software to the target (programming). ☐
   (b) A logic probe provides facilities to capture and display multiple signals simultaneously. ☐
   (c) Debug Kernels provide services which can be accessed by the programmer. ☐
   (d) Firmware is software which is stored in a volatile medium. ☐
   (e) JTAG allows the injection and extraction of values to/from an integrated circuit. ☐
   (f) Terminal emulation software allows a PC to be used as a dumb terminal. ☐
   (g) The file storage system is a device with a serial connection, which is designed to take keyed input, and relay it out as ASCII data. ☐
   (h) The host computer may be used for the display of information reported by the debug kernel. ☐
3. We would like to examine the supply DC voltage for a microprocessor based system, for suspected noise. We should use:
   (a) logic analyser ☐
   (b) logic probe ☐
   (c) memory chip ☐
   (d) oscilloscope ☐
   (e) volt-meter ☐
4. You are developing software for a hardware platform which has been built, but the microcontroller is presently unavailable. Which ONE of the following items will you use to determine whether your hardware platform functions as expected?
   (a) debug kernel ☐
   (b) integrated development environment (IDE) ☐
   (c) emulator ☐
   (d) firmware ☐
   (e) scripts ☐
5. From (Wilmhurst 2001; 12.4)

   > “ You have been asked to set up a .... [microprocessor] development system. List [two] items of hardware .... you would wish to purchase, stating briefly what each would be used for.”

6. Differentiate between the following pairs of items:
   (a) an oscilloscope and a logic analyser
   (b) firmware and software
   (c) JTAG and hardware debug support
   (d) a microprocessor development board and a custom microprocessor-based board.

7. You are developing a microprocessor-based system which requires a display and a keypad. You need to choose between:
   - an LCD display and a keypad.
   - a terminal.

   Identify two issues that you would consider, and explain how you would make your decision based on those issues.

8. A micro-processor based system fails to provide the required output in a limited number of cases. You suspect that there may be a voltage problem which is causing certain bus lines to be incorrectly interpreted as ”low”. What piece of equipment would you use to track down this problem? Explain your answer.

9. Challenge Question: Investigate the following topics
   - JTAG uses the concept of a large shift register. This can become unwieldy for large numbers of virtual pins. How does the JTAG standard address this problem?
   - How do developers of Flash Filing Systems cope with the fact that Flash memory can be written in small blocks but must be erased in larger blocks?

10. Challenge question: ROM emulators, instead of emulating the entire microprocessor, simply emulate the program/data memory. Outline how this could be used to implement debugging.

Tutorial Exercise 3B Offline Version[14]

ID# ____

Congratulations! As a new graduate, NOVATE has hired you to complete an embedded system which monitors oil flow in pipelines. The system will provide an estimate of the oil transported via the pipeline, as well as automatically detect changes in flow (e.g. pipeline breaks). The contract also specifies that the system must be demonstrated to operate correctly in a variety of different scenarios. The sensor and microprocessor hardware have been specially developed for solar power consumption, and adverse environmental conditions, and are now available for testing. The software was previously written and tested on a simulator.

1. Your boss says: "The software already works, so just download it onto the platform and send it out. Don't bother to test it again." What will you say to convince him that testing is required? Your answer should identify potential system flaws, and the tools/methods you will be using to test the software on the platform. *6 marks*

2. The final system has passed all in-lab tests. The system fails in the field. The hardware developers have determined that the hardware meets all electrical specifications. Explain (using diagrams if required) what you will do to find/fix the fault. *4 marks*

[14] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 7 Your lecturer will put up a diagram with several development support items labeled with letters. Answer the following questions for the part whose letter you were assigned. Write the letter you have been assigned here________.

1. The part is called the ____________________________.
2. Circle the correct answer. The primary function of the part is:
   (a) provide random access storage for the target system .
   (b) act as an initial hardware prototype for the designed target system .
   (c) capture and view signals on the pins of target system .
   (d) program firmware for use in the target system .
3. If you did NOT have access to this part, what effect would/could it have on your development of a micro-processor based system? Suggest alternatives to this part.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  Explain the role of
  - firmware, operating systems, oscilloscopes,logic probes/analyzers,
  - terminals, disk drives, external memory, host computers debug protocols e.g. JTAG

  in the development and support of microprocessor based systems.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

8 Microprocessor families and forms

At the end of this unit the student will be able to:

recognize and differentiate between the different commercially available

- *families of microprocessor based systems (e.g. Motorola, AVR, PIC)*
- *forms of microprocessor based systems i.e.chip/component packages – SOC, microcontroller, PC, back-plane bus, PC104, standalone, embedded*

based on their pictures/properties/descriptions and supported development tools.

Microprocessor technology is relatively young and constantly changing. Over the last 30–40 years, most changes in CPU architectural design have been driven by the need for speed. Design tradeoffs made over time include increasing bit widths, RISC/CISC, use of DMA, peripheral support, cache RAM, and dedicated/hierachical busses. Manufacturing improvements also improved clock speeds by reducing heat/distance considerations. What does all this change mean for the developer of a microprocessor based system? In manufacturing a system, we need to be concerned about two things:

- the expected lifetime of an individual system, and whether we will be able to repair/support it for it's lifetime,
- the expected lifetime of the design, and whether we will be able to produce/sell the required number of units to recoup design costs.

Clearly we can do neither if the processor around which we have designed our system becomes obsolete soon after we have gone into production/sale.

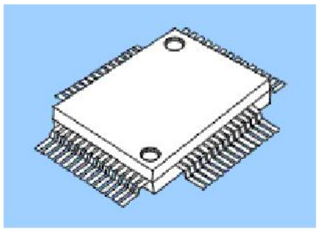
To reconcile the reality of a rapidly changing marketplace, with the customers need for stability, manufacturers of microprocessor have adopted several strategies including:

- clones – a classic case is the 8051; originally produced by Intel, many companies now offer a pin compatible, architecturally equivalent processor. It is used as a replacement for the original 8051, as well as in new designs which can take advantage of instruction set extensions/enhancements.
- families – most people are familiar with the Intel x86 processors. These were designed to be backward compatible i.e. code compiled for a 286 processor will run on a 486 processor. Support for the base instruction sets is built into later members, even though the packaging and overall architecture may be different.
- scaled function versions – where different variants of the CPU are made available, each with a unique set of features, targeting a specific market. For example, device A may require only 10 general I/O lines, but device B only 2 general I/O lines. The manufacturer makes two versions of the CPU, one with more lines and one with less lines, and prices them accordingly. The developer can then choose the lowest cost option that matches his needs.

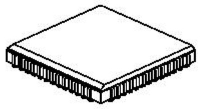
The developer cannot decide on the CPU to be used for a particular application, without investigating the form in which the microprocessor is available. Apart from the myriad packages in

Packaging

www.oki-europe.de/1.Products/ Bilder-Products/Qfp.gif

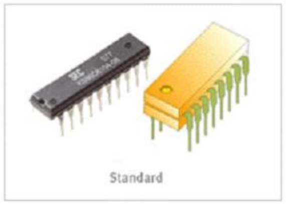


QFP -- gull wing

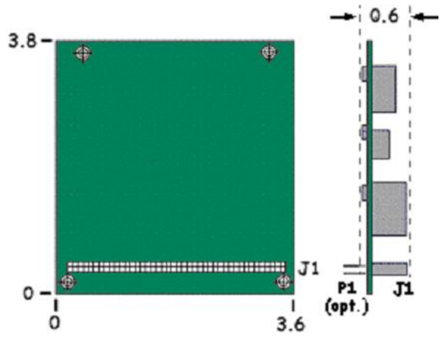


http://batronix.com/gif/package-plcc.gif

DIP

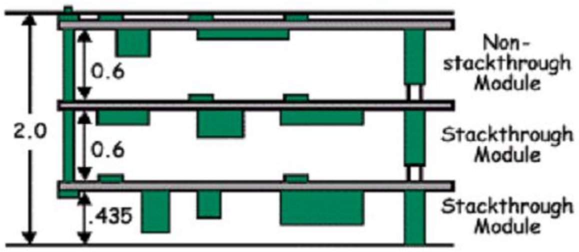


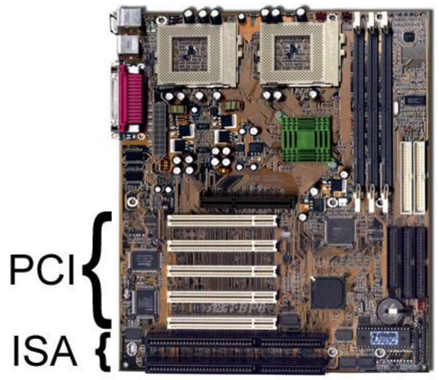
http://www.samsungelectronics.com/semiconductors/
Mask_Rom/technical_data/package_dimensions/dip.gif



PC104

http://www.versalogic.com/support/businfo/PC104Desc.asp



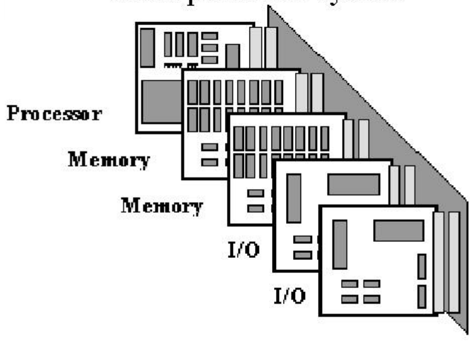


http://www.computer-store.rutgers.edu/
images/motherboard.jpg

PC motherboard

http://www.realtime-info.be/
magazine/96q4/fig/fig02.gif

Back-plane bus system



AN520
A Comparison of 8-Bit Microcontrollers

Author: Mark Palmer
Microchip Technology Inc.

Architecture Based Criterion

- instruction set flexibility e.g. number of addressing modes, memory mapping, RISC/CISC, Von Neumann vs. Harvard
- built-in hardware support features e.g. PWM, A/D
- machine cycle time (maximum clock speed)

Implementation Based Criterion

- power consumption/power saving techniques
- package size and cost
- familiarity with micro-controller family
- availability of multiplexed pin functions
- availability of development tools

BUT the actual performance of the micro-controller in a task depends on

- the space required to store the code
- the execution time of the code

which they are singly supplied (DIP, PLCC etc.), they may also be supplied as either a part of an IC containing other functional parts/peripherals (microcontroller, ASIC, SoC), or as part of a bus-ready (PC104, back-plane) board containing other functional parts and peripherals, or as a stand-alone computer system. In addition, any of these variants can come with different maximum clock speeds, depending on the manufacturing processes used to produce the CPU. The choice of form and clock speed will depend on the particular application.

Some additional items to consider include:

Environmental tolerance Will the processor have to work in dusty, chemical, radiation, hazardous, vibrating, or zero gravity environments? If so special packaging will be required.

Business Is the CPU available? How long will it take to get the product to market? What will the unit cost be? How many units can we sell? What will the net profit be?

History What are we already familiar with? Do we already have the support tools? What legacy code/features do we need to support?

Power What is our entire power budget? What power-saving options are available?

All other things being equal, we can choose a processor on the basis of code efficiency i.e. the time it takes to execute an algorithm and the space required to store the algorithm in memory (Palmer 1997a).

Review exercises

1. In your own words, write out a list of guidelines for choosing a processor for a given application.
2. Using the Intel 8051 as an example, differentiate between clones, families and scaled-function versions of microcontrollers.
3. Manufacturer X supplies an IDE which is appropriate for all the microprocessors that he supplies. We can presume that microprocessors supplied by Manufacturer X: (choose all the appropriate answers)
   (a) all have the same instruction format. ☐
   (b) are all supported by the same assembler/compiler. ☐
   (c) are all pin-compatible with each other. ☐
   (d) all operate at the same clock speed. ☐
   (e) all support on-line debugging. ☐

4. A microcontroller may be supplied in different variants. This means that: (choose all the appropriate answers)
   (a) The number, spacing and type of pins on the chip may be different for each package variant. ☐
   (b) All variants will have the same operating voltages. ☐
   (c) Different variants may have differing environmental tolerances. ☐
   (d) Different variants may have differing maximum clock rates. ☐
5. The microprocessor which was used in a particular design is no longer available. You have been asked to locate a suitable replacement. In order to be able to **use the existing code**, without re-assembly, the replacement MUST: (choose all the appropriate answers)
   (a) come from the same manufacturer ☐
   (b) be pin compatible ☐
   (c) have the same instructions in it's instruction-set ☐
   (d) have the same instructions & instruction-format ☐
6. The microprocessor which was used in a particular design is no longer available. You have been asked to locate a suitable replacement. In order to be able to **use the existing circuitry**, without re-design, the replacement MUST: (choose all the appropriate answers)
   (a) come from the same manufacturer ☐
   (b) be supplied in the same package ☐
   (c) have the same instructions in it's instruction-set ☐
   (d) be a unit with the same power/current characteristics ☐
7. AN520 (Palmer 1997a)compares the performance of several microprocessors using typical algorithms. This is an example of *benchmarking*. **Justify your answers** to each of the following questions. Is it appropriate to use benchmark data to choose a processor without:
   (a) an understanding of the benchmark algorithm?
   (b) an understanding of the application algorithm?
8. Based on (Vahid and Givargis 2002; 3.8): The PIC16F877 is one of a family of microcontrollers.
   (a) Using the datasheet, identify
      i. the basic instruction set features (# of instruction(s) variants, instruction width)
      ii. the I/O facilities (# of ports, tri-state facilities)
   (b) Go to the MicroChip web-site (`http:\\www.microchip.com`) and identify
      i. a scaled back version of the same microcontroller
      ii. another microcontroller belonging to the same family
      iii. another family with a similar (but not the same) instruction set

9. Role Play/Challenge Question: Imagine you are an engineer in a company which used an Alpha processor in a design for a system. Your company would like to continue selling the system for another 10 years. Investigate the current status of the Alpha family, and then write a letter to your line manager making recommendations as to:
   - how your company can continue to support the systems you have already sold,
   - how your company can continue to produce the system.
10. Role Play: A microcontroller based product, for use in a hostile environment, was designed using a microcontroller IC which has become obsolete. The company must decide between a pin-compatible microcontroller which is not binary-compatible, or a binary-compatible microcontroller which is not pin-compatible. Form teams which will debate this issue from the following viewpoints:
    - The software developers want to redesign/modify the board and keep the software because the code has already been checked, and the modified system could be re-tested with existing scripts.
    - The hardware developers want to rewrite/recompile the software and keep the board because it is already designed/checked for operation in a hostile environment.

Tutorial Exercise 4A Offline Version[15]

ID#

Congratulations! Because of your excellent work, NOVATE has promoted you. You head the production team for an embedded system which monitors oil flow in pipelines. The system was successfully developed with a socketed PIC16F877 microcontroller in a 40-pin DIP package, and a 4MHz clock signal. The system utilises the Timer, PWM, and A/D features of the variant. NOVATE is committed to delivering and installing 20,000 modules over 2 years. The procurement department has been ordering parts in batches for 1000 modules on a monthly basis. Ten months into production, procurement advises your team that the PIC16F877 microcontroller is no longer available in the required variant. Furthermore, they cannot locate a pin-compatible variant, with the required features, which will also be binary-compatible.

1. Identify and explain two choices you could have made back in the development stage which would have made this transition easier. *6 marks*

2. How will your team modify the system, so that the remaining modules can be delivered? Your answer should identify changes in hardware, software, testing, tools and/or tool-chains. *4 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

[15] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 8 Write the letter and number you have been assigned here________.
Answer the following questions for the Microprocessor Alternate whose letter you were assigned.

1. The CPU is made by ______________________.
2. Circle the correct answer. What type of IC package is the CPU inside?:
   (a) DIP
   (b) PLCC
   (c) QFP
   (d) Unspecified or Other
3. In what form is the CPU presented for the micro-processor-based system developer?
   (a) Microcontroller
   (b) Microprocessor-only IC
   (c) Motherboard, PC104 board, or Board for back-plane bus system
   (d) Unspecified or other
4. Compare the PIC16F877 & this processor (assign one group member to each item)
   (a) look at the block diagram(s)
      - identify the internal features of the CPU core
      - identify additional features of the microcontroller/board
      - identify other family members/clones
   (b) look at the memory map(s)
      - is this Von Neumann/Harvard architecture?
      - are devices memory-mapped?
      - is there a separate I/O bus?
   (c) locate the branch/jump instruction
      - what is the mnemonic?
      - what is the op-code?
      - what is the branch address size?
   (d) locate the clock signal/data specifications
      - what is the maximum clock frequency?
      - what is the CPI? MIPS ?
      - what is data/instruction width?
   (e) locate the power/current requirements
      - what is an appropriate operating frequency for both platforms?
      - what is the minimum operating voltage for a given frequency?
      - what is the average current (CPU Only) draw at a given voltage/frequency?

Unit 8

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  Recognize and differentiate between the different commercially available
  - families of microprocessor based systems (e.g. Motorola, AVR, PIC)
  - forms of microprocessor based systems i.e. chip/component packages; board forms

  based on their pictures/properties/descriptions and supported development tools.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

Informative Abstract

You have been assigned an article on a topic of relevance to microprocessor-based systems.

Groups A1, B1, C1, D1 : "Fuzzy Logic Control for an Autonomous Robot" by V.M. Peri and D. Simon, Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS), 2005, 26-28 June 2005, page(s): 337- 342.

Groups A2, B2, C2, D2 : "Sniffing Robot" by Silvio Tresoldi, Circuit Cellular, Issue 108, July 1999, page(s): 12-16.

Groups A3, B3, C3, D3 : "Machine Chameleon" by D. Verkest, IEEE Spectrum, Volume 40, Issue 12, Dec. 2003, page(s):41 - 46.

Groups A4, B4, C4, D4 : "The transistor laser" by Nick Holonyak Jr. and Milton Feng, IEEE Spectrum, Volume 43, Issue 2, Feb. 2006, page(s):50 - 55.

1. Read your assigned article (or choose another article from an IEEE magazine, and have it approved by your lecturer), and submit (**at most**) a one (1) page **informative** abstract of the article. Your abstract should include the following areas: *12 marks*
   (a) The objectives of the article or research undertaken;
   (b) The scope of the work;
   (c) The methodology employed;
   (d) The significant findings or results;
   (e) The significant recommendations and;
   (f) The conclusions drawn.
2. Identify, and explain in your own words, either
   (a) a potential application of this research/work to
   (b) a prediction about how this research/work may affect

   the design/production of microprocessor-based systems in the near-future. *2 marks*

Your submission should be headed with the article reference and your name/ID number. You must upload your submission to Moodle using the relevant assignment link. Your submission should be named `Fnnnnnnnn.xxx` where `nnnnnnnn` is your ID number, and `xxx` is the file extension reflecting the file type.

Files may be PDF, Microsoft Word (DOC) or plain ASCII files (TXT).

Please remember to electronically submit AHEAD of the deadline in order to avoid possible problems with system overload.

Microprocessor-based system development

There are many issues to be considered when developing a micro-processor based system. In this section we have examined some of the technology available for software and hardware development support, as well as the different ways in which CPU technology is made available to the designer of a microprocessor-based system.

As such we have addressed part of the first learning objective "**in general, identify, and describe the role of, the components of a development suite (hardware, software) for a microprocessor-based system**" and hinted at the final learning objective "**select (and critique the selection of) a microprocessor-based system for an application, given relevant datasheets and application requirements**".

In the next section we will be looking in some detail at a particular CPU, it's instruction set, and memory map, and at the developmental support tools available for it. In doing so, we will complete the first learning objective.

©DECE, UWI, St. Augustine, Trinidad
original author: Feisal Mohammed; updated: March 11, 2008– CLR

Part III

PIC16 Introduction

All microcomputer systems, irrespective of their complexity, are based on similar building blocks. The CPU or microprocessor is the core component of any microcomputer and it requires the external components such as the ROM, RAM, I/O etc. to accomplish its purpose. A microcontroller is a stripped-down version of the very same architecture, with all the important features placed on one chip. The same system using a microprocessor or a microcontroller looks like Figures 2,3. The microcontroller based system requires no additional circuitry except a clock input and it can, in many cases, directly drive peripheral outputs. The difference between the microprocessor and the microcontroller arises because of their different end-usage.

The microcontroller that will be investigated is the PIC16F877, which is at the upper end of the mid-range series of the microcontrollers developed by MicroChip Inc. It is characterized by a RISC architecture instead of the CISC architecture used, for example, by the Motorola 6809.

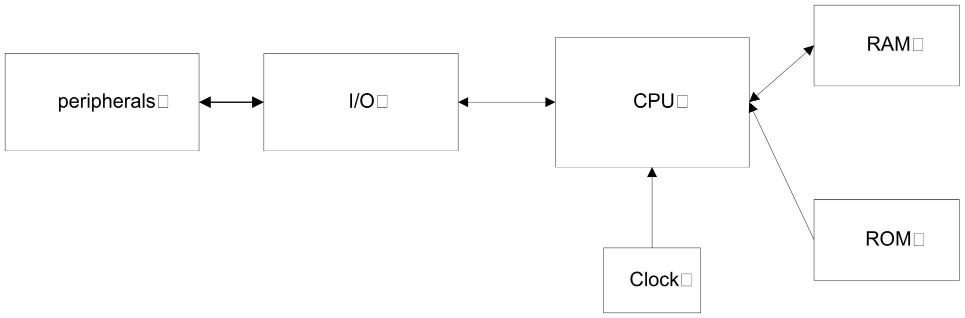


Figure 2: Basic building blocks of a computer

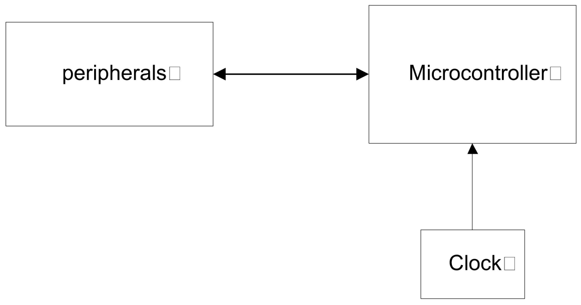


Figure 3: A microcontroller based system

9 PIC16F877 Overview

At the end of this unit the student will be able to:

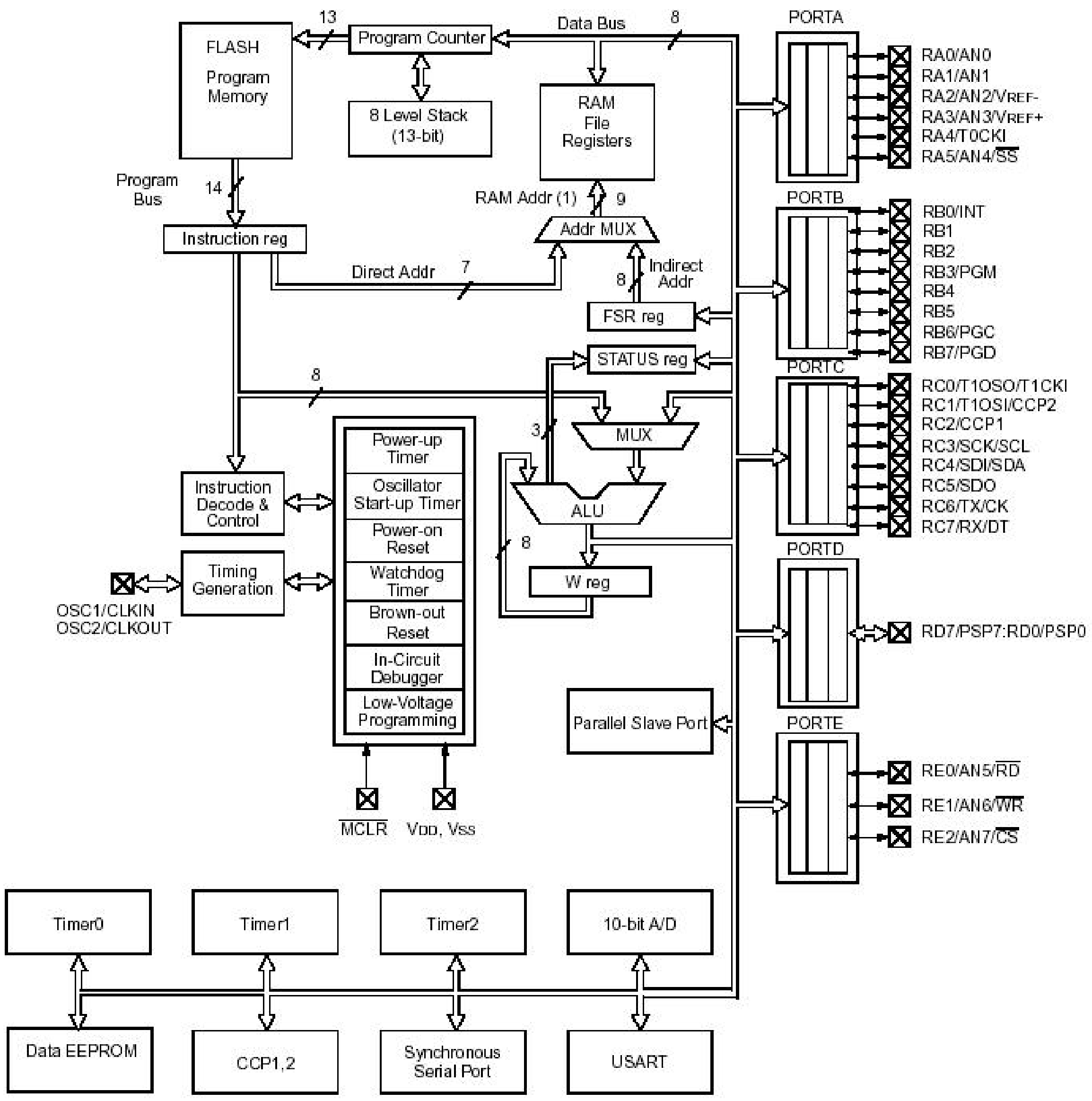
- *explain the memory layout for the PIC16F877*
- *explain the operation of the instruction cycle for the PIC16F877*
- *explain the operation of the basic instructions (move, add, subtract, shifts) for the PIC16F877*

The history of the PIC series of microcontrollers started in in 1965, when General Instruments (GI) formed a Microelectronics Division, and used this division to generate some of the earliest viable EPROM and EEPROM memory architectures. GI also made a 16 bit microprocessor, called the CP1600, in the early 1970s. While this was a reasonable microprocessor, it was not particularly good at handling I/O. Therefore, around 1975, GI designed a Peripheral Interface Controller (or PIC for short) for some very specific applications where good I/O handling was needed. It was designed to be very fast since it was I/O handling for a 16 bit machine, but it did not need a large amount of functionality, so its microcoded instruction set was small. Its architecture was substantially the PIC16C5x architecture of today. The market for the PIC remained small for the next few years. During the early 1980s, GI restructured their business to concentrate more on their core activity which was power semiconductors. As a result of the restructuring, the GI Microelectronics Division became GI Microelectronics Inc (a wholly owned subsidiary) which, in 1985, was finally sold to venture capital investors. The sale included the fabrication plant in Chandler, Arizona. The new owners decided to concentrate on the PICs, the serial and parallel EEPROMs and the parallel EPROMs. A decision was later taken to start a new company, named Arizona MicroChip Technology, with embedded control as its differentiator from the rest of the industry. As part of this strategy, the PIC family was redesigned to use one of the other things that the fledgling company was good at, i.e. EPROM. With the addition of CMOS technology and erasable EPROM program memory the PIC family, as we know it, was born.

Architecture of the PIC microcontroller

The PIC series of microcontrollers are RISC-based processors with an accumulator(also called the working register, W), which use the Harvard[16] architecture; therefore the microcontroller has a program memory data bus and a data memory data bus. Separate buses mean that simultaneous access of program and data can be done, which gives a greater bandwidth over the traditional von Neumann architecture. Separating the program and data memory, allows instructions to be sized differently than the 8-bit wide data word. This separation means that the instruction words can be ideally sized for the specific CPU/application. This is necessary since RISC architectures require that instructions have the source and destination operands be encoded within the instruction. The PIC opcodes for the mid-range processors are 14-bits wide, and the 14-bit wide program bus fetches an instruction in a single cycle.

[16]This architecture had been a scientific curiosity since its invention by Harvard University in a Defense Department funded competition that pitted Princeton against Harvard. Princeton won the competition because the MTBF of the simpler single memory architecture was much better, albeit slower, than the Harvard submission. MicroChip has made a number of enhancements to the original architecture, and updated the functional blocks of the original design with modern advancements made possible by the low cost of semiconductors.



Note 1: Higher order bits are from the STATUS register.

There are 35 single word instructions. A two-stage pipeline overlaps fetch and execution of instructions. As a result, all instructions execute in a single cycle except for program branches. The pipeline uses *static branch prediction* and assumes that the branch is *never taken*, so that conditional branch instructions can take either one or two cycles. One cycle if the branch is not taken, since it would be in the pipeline and two cycles if the branch was taken. Non-conditional branch instructions such as `call` or `goto` always take two cycles. All program memory is internal i.e. the program bus is not accessible outside of the chip. The data path is 8 bits wide. Data memory may be accessible from outside of the chip using the Parallel Slave Port; when enabled, the port register is asynchronously readable and writable by the external world. The mid-range PIC processors can directly or indirectly address their data memory[17]. All special function registers[18], including the program counter, are mapped in the data memory. The design of the instruction set is such that it is possible to carry out any operation on any register using either addressing mode. This design simplifies the programming of the device. The term *file register* is, in PIC terminology, used to denote the locations that an instruction can access via an address. The term *register file* is used to collectively refer to the group of registers.

The PIC ALU can perform arithmetic and Boolean functions between data in the working register and any file register. The unit is 8-bits wide and capable of addition, subtraction, shift and logical operations. In two-operand instructions, typically one operand is the Working (W) register and the other operand is a file register or an immediate constant. In single operand instructions, the operand is either the W register or a file register. Depending on the instruction executed, the ALU may affect the values of the Carry (C), Digit Carry (DC) and Zero (Z) bits in the STATUS register. The C and DC bits operate as $\overline{\text{Borrow}}$ and $\overline{\text{Digit Borrow}}$ bits respectively, in subtraction.

Memory Layout

The PIC16F877 has 8K × 14-bit words of Flash program memory, 368 bytes of data RAM, 256 bytes of data EEPROM and an 8-level x 13 bit wide hardware stack.

The Program Counter (`PC`) is 13 bits wide, thus making it possible to access all 8K x 14 addresses. The low byte is the `PCL` (Program Counter Low byte) register which is a readable and writable register. The high byte of the `PC` (`PC`<12:8>) is not directly readable or writable and comes from the `PCLATH` (Program Counter LATch High) register. The `PCLATH` register is a holding register for the `PC`<12:8>. The contents of the `PCLATH` are transferred to the upper byte of the program counter when the `PC` is loaded with a new value. Although the `PC` is capable of addressing the entire program memory space, conceptually the program memory is represented by four banks of 2K × 14-bit words. Banking is necessary since there are only 11 bits for the address in the instruction word for a `call` or `goto`. The other two bits are obtained from the top two bits of `PCLATH` (i.e. `PCLATH<4:3>`). This means that the user must set those extra bits in `PCLATH` before branching out of the 2K bank that contains the current instruction. Within the program memory space, the reset vector (location to go to on reset) is at 0000h and the interrupt vector (location to go to on interrupt) is at 0004h.

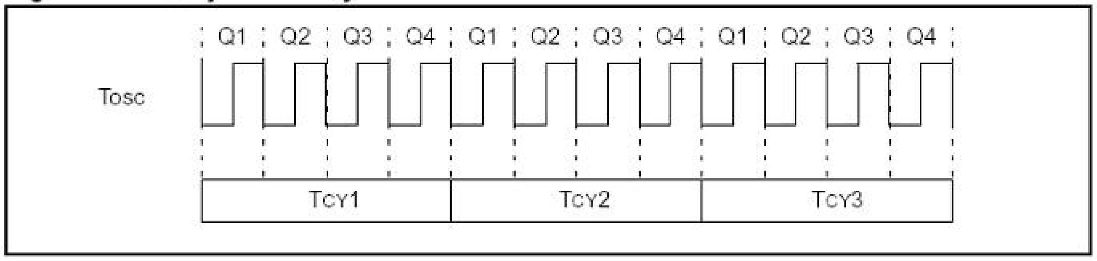
The data memory space effectively has 9 bit addresses, and is also banked. There are 4 banks; each bank holds 128 bytes of addressable memory. The banked arrangement is necessary because there are only 7 bits are available in the instruction word for the addressing of a register, which gives only 128 addresses. The selection of the banks is determined by control bits RP1, RP0 in

[17]Data memory may be EPROM based, or RAM based. General purpose registers are contained in the RAM

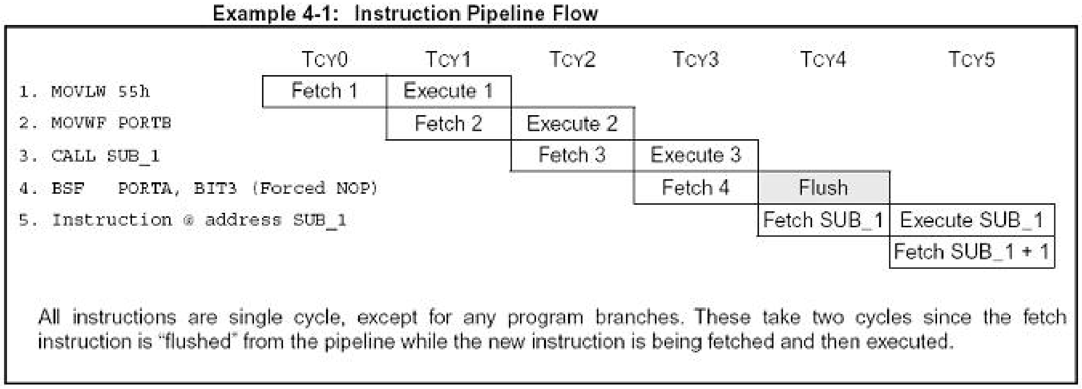
[18]Note: The Working register (W) is an 8-bit register used for all ALU operations; it is not addressable.

Clock/Pipeline

Figure 5-2: Q Cycle Activity



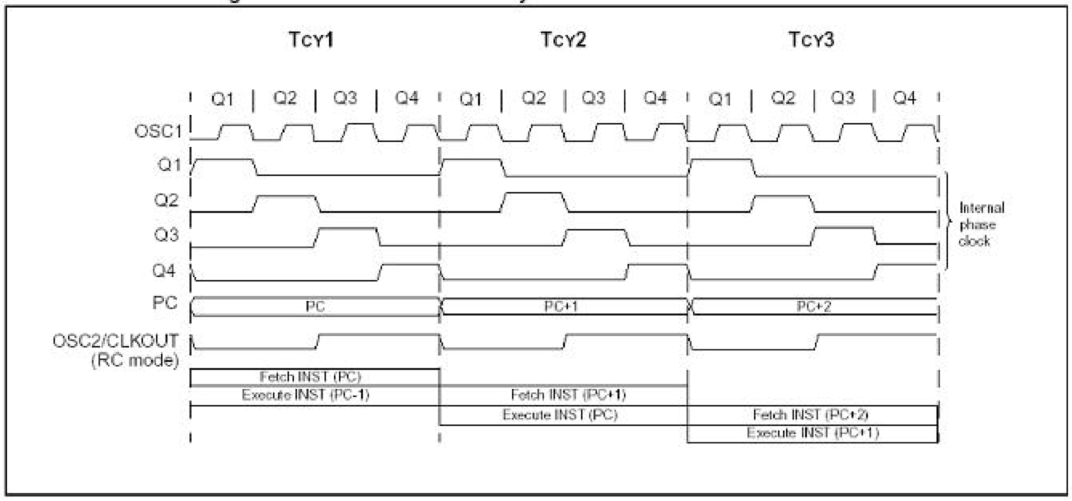
Example 4-1: Instruction Pipeline Flow



	TCY0	TCY1	TCY2	TCY3	TCY4	TCY5
1. MOVLW 55h	Fetch 1	Execute 1				
2. MOVWF PORTB		Fetch 2	Execute 2			
3. CALL SUB_1			Fetch 3	Execute 3		
4. BSF PORTA, BIT3 (Forced NOP)				Fetch 4	Flush	
5. Instruction @ address SUB_1					Fetch SUB_1	Execute SUB_1
						Fetch SUB_1 + 1

All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

Figure 4-3: Clock/Instruction Cycle



the `STATUS` register (`STATUS`<6:5>). Together the RP1, RP0 and the specified 7 bits effectively form a 9 bit address. The first 32 locations of Banks 1 and 2, and the first 16 locations of Banks 2 and 3 are reserved for the mapping of the Special Function Registers (SFR's). SFRs (e.g. `PC`, `STATUS`) are used to control the "core" operation of the microcontroller as well as peripherals such as the I/O ports. The SFRs are mapped into the data memory space to facilitate addressing. The 368 bytes of static RAM which are used as general purpose registers (GPR), are arranged in the data memory space so that each is accessed by a unique address, with the exception of the last 16 bytes of each bank, which are shared. Each mapped SFR and GPR is 8-bits wide. The entire data memory can be accessed either directly using the absolute address of each register file, or indirectly through the INDirect File (INDF) register and the File Select Register (FSR). The INDF register is not a physical register. Any instruction using the INDF register actually accesses the register pointed to by the FSR. Reading the INDF register itself, indirectly (i.e. FSR = 0) will read 00h. Writing to the INDF register indirectly results in a no operation (although status bits may be affected). Because the FSR is 8 bits wide, indirect addressing can access two data memory banks simultaneously. The effective 9-bit address is obtained by concatenating the IRP bit (STATUS<7>) and the 8-bit FSR register.

The 256 bytes (8 bit address) of EEPROM memory is indirectly mapped into the data memory using the **EE**PROM ADRess (EEADR), **EE**PROM CONtrol (EECON1) and **EE**PROM DATA (EEDATA) registers[19]. Reading the data EEPROM memory only requires that the desired address to access be written to the EEADR register and the EEPGD bit of the EECON1 register be cleared (to indicate the data memory is to be read). Then, once the RD bit of the EECON1 register is set[20], data will be available in the EEDATA register on the very next instruction cycle. EEDATA will hold this value until another read operation is initiated or until it is explicitly written.

The stack is not part of the program or data space and the stack pointer is not readable or writable. There are also no operations to place or remove data to or from the stack. Addresses are placed on the stack by the CPU when a `call` instruction is executed or when an interrupt occurs. The various return instructions then remove the previously stored address from the stack. The stack operates as a circular buffer, meaning that after the stack is full, subsequent 'pushes' start overwriting the previously stored data from the top (the very first push). Similarly when the stack is 'popped' nine times, the ninth value is the same as the first pop. The programmer never has to interact directly with the stack, but should always remember the 8-level limit. The stack does not give any indication if overflow or underflow occurs.

The 8-bit working register (W) is also not part of the program or data memory space. It can however be read/written using particular instructions. Register-Register operations are not possible.

Instruction set for the PIC microcontroller

The format of the instruction words are shown in Figure 4(PIC 2001; Figure 13.1) . All instructions can use either direct or indirect addressing to access a register file. Since the addresses range from 00h to 7Fh in each bank, only 7 bits are required to identify the register file address and this is contained within the instruction. The eighth and ninth bits, which identify the bank, then comes from the register bank select bits (RP1,RP0) from the STATUS register. Figure (5) illustrates how the bits are composed to access the appropriate address. In indirect addressing, the 8-bit register

[19]The program memory could be read indirectly in a similar fashion, however this is a multi-cycle operation, and is not normally required.

[20]The RD bit of EECON1 will automatically clear once the read is performed

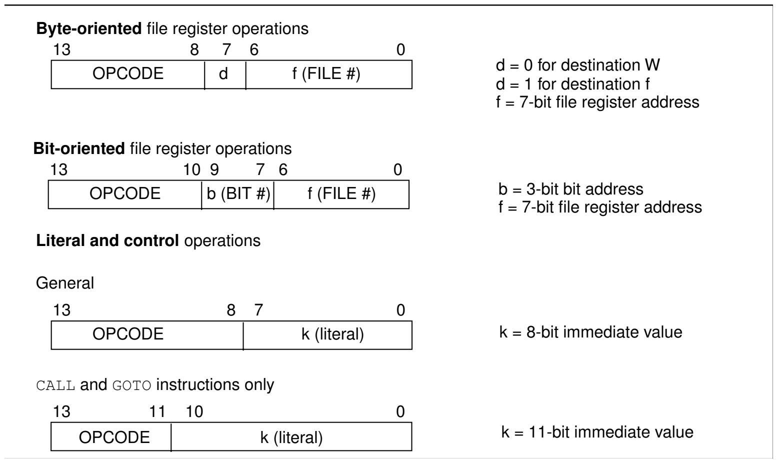

Byte-oriented file register operations
13 8 7 6 0
OPCODE
d
f (FILE #)
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address
Bit-oriented file register operations
13 10 9 7 6 0
OPCODE
b (BIT #)
f (FILE #)
b = 3-bit bit address
f = 7-bit file register address
Literal and control operations
General
13 8 7 0
OPCODE
k (literal)
k = 8-bit immediate value
CALL and GOTO instructions only
13 11 10 0
OPCODE
k (literal)
k = 11-bit immediate value

Figure 4: Format of instructions

Mnemonic, Operands		Description	Status Affected
Byte-oriented file register operations			
ADDWF	f,d	Add W and f	C,DC,Z
ANDWF	f,d	AND W with f	Z
CLRF	f	Clear f	Z
CLRW	-	Clear W	Z
COMF	f,d	Complement f	Z
DECF	f,d	Decrement f	Z
DECFSZ	f,d	Decrement f, Skip if 0	
INCF	f,d	Increment f	Z
INCFSZ	f,d	Increment f, Skip if 0	
IORWF	f,d	Inclusive OR W with f	Z
MOVF	f,d	Move f	Z
MOVWF	d	Move W to f	
NOP	-	No operation	
RLF	f,d	Rotate Left f through Carry	C
RRF	f,d	Rotate Right f through Carry	C
SUBWF	f,d	Subtract W from f	C,DC,Z
SWAPF	f,d	Swap nibbles in f	
XORWF	f,d	Exclusive OR W with f	Z
Bit-oriented file register operations			
BCF	f,b	Bit Clear f	
BSF	f,b	Bit Set f	
BTFSC	f,b	Bit Test f, Skip if Clear	
BTFSS	f,b	Bit Test f, Skip if Set	
Literal and Control operations			
ADDLW	k	Add literal and W	C,DC,Z
ANDLW	k	AND literal with W	Z
CALL	k	Call subroutine	
CLRWDT	-	Clear watchdog timer	$\overline{\text{TO}}$, $\overline{\text{PD}}$
GOTO	k	Goto address	
IORLW	k	Inclusive OR literal with W	Z
MOVLW	k	Move literal to W	
RETFIE	-	Return from interrupt	
RETLW	k	Return with literal in W	
RETURN	-	Return from subroutine	
SLEEP	-	Clear watchdog timer	$\overline{\text{TO}}$, $\overline{\text{PD}}$
SUBLW	k	Subtract W from literal	C,DC,Z
XORLW	k	Exclusive OR literal with W	Z

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
d	Destination select: d = 0, Store result in W d = 1, Store result in file register f default is d = 1

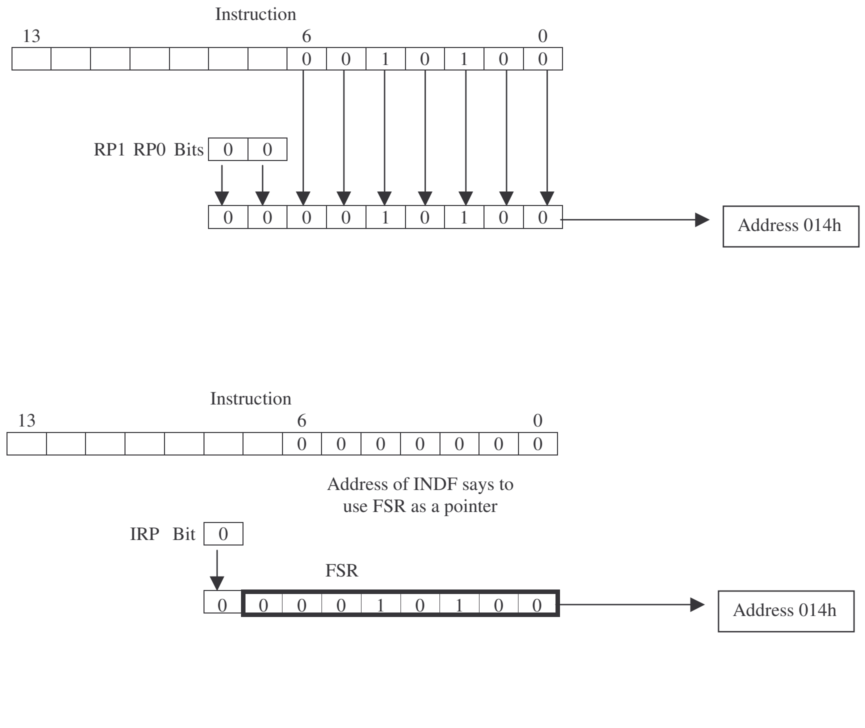


Figure 5: Direct and indirect addressing modes

file address is first written to the the FSR, which is a special purpose register that is used as an address pointer to any address in the entire register file. A subsequent direct access to the INDF register will actually access the register file whose address is contained in the FSR i.e. the FSR acts as a pointer. Since the FSR is an 8-bit register, both banks can be addressed without needing to switch between them. The INDF register is not a physical register like the other special function registers. The FSR register is one of several Special Function registers that have been assigned multiple addresses so that they can be accessed in all banks. Figure (5) shows how the address is formed.

The sparseness of the instruction set means that more programming effort is required in the programming of the PIC. A few of the special considerations are listed below.

Default destination The instructions from the table of the instruction that have `f,d` as the operands use the `d` as an indicator to tell where to place the result. The default is `f` (file register), but this can be confusing so it is better to always specify the destination directly as the following code fragment demonstrates.

```
W       EQU     H'0000' ; standard definitions
F       EQU     H'0001' ; ...to avoid having to write numbers

        incf var,W  ; W = var + 1, var is unchanged
        incf var,F  ; var = var + 1, W is unchanged
```

STATUS register The STATUS register contains the arithmetic status of the ALU, the RESET status and the bank select bit for the data memory. As with any register, the STATUS register can be the destination for any instruction. If the STATUS is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Furthermore the $\overline{TO}$ and $\overline{PD}$ bits are not writable, therefore the result of an instruction with the STATUS register as destination may be different than intended.

For example: `clrf STATUS` will not set the STATUS register to all zeros as expected but will clear the upper three bits and set the Z bit. This leaves the STATUS register as 000u u1uu (where u = unchanged). Only the `bcf`, `bsf` and `movwf` instructions should be used to alter the STATUS register because these instructions do not affect any status bits.

Review Exercises

1. Label the following statements about the PIC16F877 as True or False:
   (a) Both the data and program memory spaces require switching banks to access the full space. ☐
   (b) Data memory is 8 bits wide with a 9 bit address. ☐
   (c) It has 8 bit data registers, and 16 bit instructions. ☐
   (d) PIC16F877 machine cycle consists of 4 clock cycles. ☐
   (e) Register to register operations are supported. ☐
   (f) The INDF register is used to specify the address which is indirectly addressed when FSR is read. ☐
   (g) The PIC16F77 has a RISC-based Von Neumann architecture. ☐
   (h) The banking bits are located in the STATUS register. ☐
   (i) The data memory has 9 bit addresses and the program memory has 13 bit addresses. ☐
   (j) The stack used to store the PC during a subroutine call is only 8 levels deep. ☐
2. Which ONE of the following is NOT a feature of the PIC16F877?
   (a) 2 stage pipeline ☐
   (b) 4 bit multiplier ☐
   (c) Harvard architecture ☐
   (d) built-in timers ☐
   (e) indirect addressing ☐
3. Certain general purpose and special function registers are mapped to multiple addresses in the memory space of the PIC 16F877. What possible advantage could such a scheme offer?
4. Determine which of the following statements concerning the following code snippet is/are NOT correct:

```
movlw   0x21
movwf   0x04
movlw   0x09
movwf   0x00
incf    0x04,1
sleep
```

(a) After this code has been executed, the CARRY flag will be clear. ☐

(b) After this code has been executed, the ZERO flag will be clear. ☐

(c) After this code has been executed, the value stored in location 0x00 is 0x09. ☐

(d) After this code has been executed, the value stored in location 0x04 is 0x21. ☐

(e) The code performs direct addressing of location 0x04, indirect addressing of location 0x21, and immediate addressing of literal 0x09. ☐

5. (a) Explain (and illustrate using pieces of code) the direct and indirect mechanisms of addressing Special Function Registers.

   (b) What happens when the assembly language statement `clrf STATUS` is executed?

   (c) Explain how the INDF register behaves when it is indirectly addressed for read/write.

   (d) Differentiate between the indirect addressing mechanisms for the register file and the data EEPROM.

6. Each instruction requires one cycle for its fetch and one cycle for its execution, yet the PIC16F877 executes a new instruction every cycle.

   (a) Explain how this accomplished.

   (b) Why does a branch instruction, which also requires the same two cycles for fetching and execution, introduce an extra cycle in the CPU's execution of instructions?

7. Explain what the following PIC16F877 assembly language instructions for the PIC16F877 will do (source; operation; destination):

   (a) ADDWF 0x20,0x00

   (b) XORLW 0x02

8. In the PIC16F877, how is the instruction cycle related to machine cycles and clock cycles?

9. Role play: For the PIC 16F877, "The programmer never has to interact directly with the stack, but should always remember the 8-level limit". For a similar processor with a 2-level stack, illustrate (using diagrams) the problems that may occur if the programmer overruns/underruns the stack.

10. Challenge: Find a single instruction that will toggle bit 0 of a file register in the PIC16F877 if:

    (a) you do not care what happens to the other bits in the file register,

    (b) you wish to preserve the values of the other bits in the file register,

    (c) the file register is the PC register

    (d) the file register is the STATUS register

Tutorial Exercise 4B Offline Version[21]

ID# ______

1. Differentiate between the operation of the following instructions: *2 marks*

```
addwf Reg,F
addwf Reg,W
```

2. What types of applications are microcontrollers (as opposed to microprocessors) commonly used for? Explain your answer. *3 marks*

3. The PIC16F877 has both data EEPROM and static data RAM (general purpose registers) included in the microcontroller. Suggest ONE reason why the data EEPROM is also included in the microcontroller. *2 marks*

4. BRIEFLY explain the concept of "banking" and give ONE reason why memory banks might be used in a microcontroller. *3 marks*

[21] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 9 Write the letter you have been assigned here________.

The registers are A: FSR; B: STATUS; C: PCL; D: INDF.

Answer the following questions for the PIC16F877 register whose letter you were assigned.

1. Locate all the mapped addresses in the data memory space

2. What happens when the assembly language instruction `movwf` is executed on this register? Which flags are affected? Are there any strange/unusual effects as compared to general file registers?

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the memory layout for the PIC16F877
  - explain the operation of the instruction cycle for the PIC16F877
  - explain the operation of the basic instructions (move, add, subtract, shifts) for the PIC16F877
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

10 MPLAB Overview

At the end of this unit the student will be able to:

> *utilize the MPLAB IDE and CCS compiler, to develop software for the MicroChip PIC16Cxxx series of microcontroller, in C, C++ and assembly language.*

Assemblers and compilers tend to have specific keywords or pre-processor directives, which are used to: access non-standard hardware features, specify code generation options/placement, supply code "shortcuts" e.g. macros, defines. In this unit, we will examine some of the relevant keywords/directives of the MicroChip MPASM assembler and the CCS compilers.

" Absolute code is the default output from MPASM. ... When a source file is assembled in this manner, all values used in the source file must be defined within that source file, or in files that have been explicitly included. If assembly proceeds without errors, a HEX file will be generated, containing the executable machine code for the target device. This file can then be used in conjunction with a device programmer to program the microcontroller. ...

[extracts from]Table 2.1: MPASM Default Extensions

Extension	Purpose
.ASM	Default source file extension input to MPASM: <source_name>.ASM
.LST	Default output extension for listing files generated by MPASM: <source_name>.LST
.HEX	Output extension from MPASM for hex files (see Appendix A): <source_name>.HEX

... The source code file may be created using any ASCII text file editor. It should conform to the following basic guidelines. Each line of the source file may contain up to four types of information: labels, mnemonics, operands, comments. The order and position of these are important. Labels must start in column one. Mnemonics may start in column two or beyond. Operands follow the mnemonic. Comments may follow the operands, mnemonics or labels, and can start in any column. The maximum column width is 255 characters. Whitespace or a colon must separate the label and the mnemonic, and the mnemonic and the operand(s). Multiple operands must be separated by a comma. A label must start in column 1. It may be followed by a colon (:), space, tab or the end of line. Labels must begin with an alpha character or an under bar (_) and may contain alphanumeric characters, the under bar and the question mark. Labels may be up to 32 characters long. By default they are case sensitive, ... If a colon is used when defining a label, it is treated as a label operator and not part of the label itself. ... If there is a label on the same line, instructions must be separated from that label by a colon, or by one or more spaces or tabs. ... MPASM treats anything after a semicolon as a comment. All characters following the semicolon are ignored through the end of the line. String constants containing a semicolon are allowed and are not confused with comments.

The listing file format produced by MPASM is straight forward: The product name and version, the assembly date and time, and the page number appear at the top of every page. The first column of numbers contains the base address in memory where the code will be placed. The second column displays the 32-bit value of any symbols created with the ... EQU, ... directives. The third column is reserved for the machine instruction. This is the code that will be executed by the PICmicro MCU. The fourth column lists the associated source file line number for this line. The remainder of the line is reserved for the source code line that generated the machine code. Errors, warnings, and messages are embedded between the source lines, and pertain to the following source line. The symbol table lists all symbols defined in the program. The memory usage map gives a graphical representation of memory usage. X marks a used location and - marks memory that is not used by this object. The memory map is not printed if an object file is generated.

Intel Hex Format: This format produces one 8-bit hex file with a low byte, high byte combination. Since each address can only contain 8 bits in this format, all addresses are doubled. Each data record begins with a 9 character prefix and ends with a 2-character checksum. Each record has the following format: `:BBAAAATTHHHH....HHHCC` where:

BB - is a two digit hexadecimal byte count representing the number of data bytes that will appear on the line.

AAAA - is a four digit hexadecimal address representing the starting address of the data record.

TT - is a two digit record type record type that will always be 00 except for the end-of-file record, which will be 01.

HH - is a two digit hexadecimal data byte, presented in low-byte/high-byte combinations.

CC - is a two digit hexadecimal checksum that is the twos complement of the sum of all preceding bytes in the record.

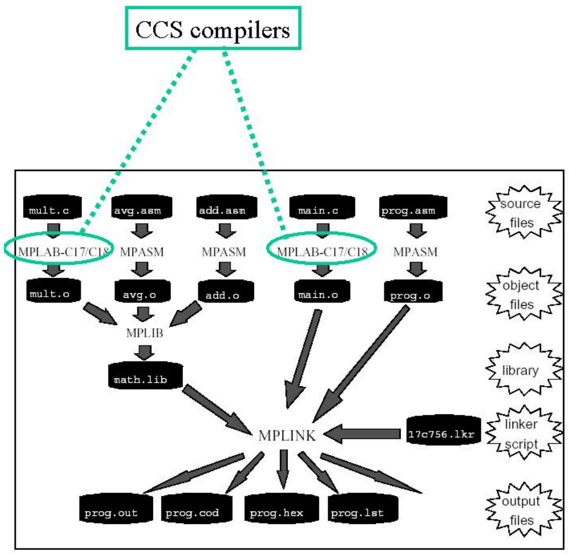


Figure 2.1: Microchip Tool Overview

Hex output

16C64

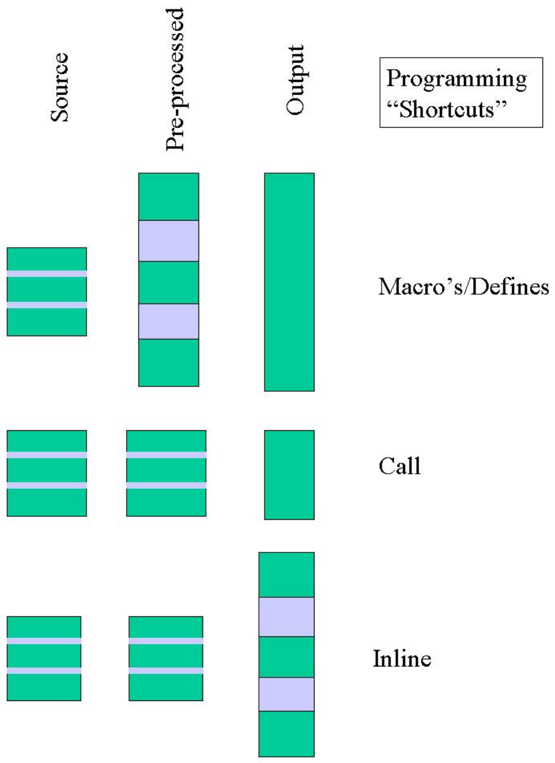
```
:020000040000FA
:080000000A308B008B1100286F
:0203FE000028D5
:00000001FF
```

```
:020000040000FA
:080000000A0C2B006B04000A3E
:0203FE00000AF3
:00000001FF
```

16C54

Insert Assembly in C

```
int find_parity (int data)
{
     int count;
     #asm
               movlw   0x8
               movwf   count
               movlw   0
          loop:
               xorwf   data,w
               rrf     data,f
               decfsz  count,f
               goto    loop
               movwf   _return_
     #endasm
}
```



... Directives are assembler commands that appear in the source code but are not translated directly into opcodes. They are used to control the assembler: its input, output, and data allocation. ... ...

ORG **Set Program Origin.** Syntax: `[<label>] ORG <expr>`. Set the program origin for subsequent code at the address defined in `<expr>`. If `<label>` is specified, it will be given the value of the `<expr>`. If no ORG is specified, code generation will begin at address zero.

LIST **Listing Options.** Syntax: `LIST [<list_option>, ..., <list_option>]`. Occurring on a line by itself, the LIST directive has the effect of turning listing output on, if it had been previously turned off. Otherwise, one of the following list options can be supplied to control the assembly process or format the listing file:

[Extracts from]Table 5.2: List Directive Options

Option	Default	Description
mm=ON—OFF	On	Print memory map in list file.
p=`<type>`	None	Set processor type; for example, PIC16C54.
st=ON—OFF	On	Print symbol table in list file.

INCLUDE **Include Additional Source File.** Syntax: `include <<include_file>>`. The specified file is read in as source code. The effect is the same as if the entire text of the included file were inserted into the file at the location of the include statement. Upon end-of-file, source code assembly will resume from the original source file. Up to six levels of nesting are permitted. `<include_file>` may be enclosed in quotes or angle brackets.

EQU **Define an Assembler Constant.** Syntax: `<label> EQU <expr>`. The value of `<expr>` is assigned to `<label>`.

END **End Program Block.** Indicates the end of the program [file].

"(MPA 1999)

For the CCS C/C++ compilers, " Pre-processor directives all begin with a # and are followed by a specific command. Syntax is dependent on the command. Several of the pre-processor directives are extensions to standard C. .... [e.g.The `#DEFINE`] directive is used to provide a simple replacement of the ID with the given string.

... C provides a pre-processor directive that compilers will accept and ignore or act upon the following data. This implementation will allow any pre-processor directives to begin with `#PRAGMA`. To be compatible with other compilers, this may be used before non-standard features. ...

[The `#ORG`] directive will fix the following function or constant declaration into a specialized ROM area. ...

The lines between the `#ASM` and `#ENDASM` are treated as assembly code to be inserted. These may be used anywhere an expression is allowed. The syntax is described on the following page. The predefined variable `_RETURN_` may be used to assign a return value to a function from the assembly code. Be aware that any C code after the `#ENDASM` and before the end of the function may corrupt the value. ...

[The `#DEVICE`] directive defines to the compiler what the hardware architecture is. This will determine the RAM and ROM map as well as the instruction set used. For chips with more than 256 bytes of RAM you may select either 8 bit or 16 bit pointers. To use 16 bit pointers add a *=16 after the chip name or by itself after the chip is defined. To use the Microchip ICD add an ICD=TRUE after the chip name or by itself after the chip is defined. ... [`__DEVICE__`]This pre-processor identifier is defined by the compiler with the base number of the current device (from a `#DEVICE`). The base number is usually the number after the C in the part number. For example the PIC16C622 has a base number of 622. ...

[The `#INLINE`] directive tells the compiler that the procedure immediately following the directive is to be implemented INLINE. This will cause a duplicate copy of the code to be placed everywhere the procedure is called. This is useful to save stack space and to increase speed. Without this directive the compiler will decide when it is best to make procedures INLINE. "(CCS 1994-2000)

Review Exercises

1. The assembly listing file generated by MPLAB PIC16F877 compilers/assemblers display all of the following types of information except:
   (a) original source code ☐
   (b) generated machine code values ☐
   (c) generated hex file lines formatted for download ☐
   (d) address locations where machine code will be stored ☐
   (e) labels and their equivalent values ☐

2. We wish to specify the target microprocessor to the CCS compiler. Which ONE of the following pre-processor directives is most appropriate:
   (a) #include
   (b) #define
   (c) #device
   (d) #org
   (e) #inline

3. Which ONE of the following statements about the MPASM assembler directives is correct:
   (a) the END directive appears at the end of each subroutine ☐
   (b) the ORG directive indicates the data memory location that the variable is stored at. ☐
   (c) the DT directive is used for lookup tables ☐
   (d) the BANKSEL directive is used to set the tristate registers ☐
   (e) the INCLUDE directive is used to indicate the options to be used by the assembler ☐

4. We wish to specify the data memory locations, for multiple multi-byte variables, to the MPLAB assembler. Which ONE of the following pre-processor directives is most appropriate:
   (a) CBLOCK
   (b) EQU
   (c) END
   (d) LIST
   (e) ORG

5. The instruction `comf 0x1A4,W` would be encoded by the assembler as:
   (a) $A409_{16}$ ☐
   (b) 0924_{16} ☐
   (c) $09A4_{16}$ ☐
   (d) 2409_{16} ☐

6. You wish to assign an assembly language variable to a space in the memory map of a PIC16F877 using the EQU directive. Which of the following is a suitable address:
   (a) 0x003 ☐
   (b) 0x000 ☐
   (c) 0x020 ☐
   (d) 0x300 ☐
   (e) 0x004 ☐

7. In your own words,
   (a) differentiate between an instruction and a directive.
   (b) explain what each of the following assembler directives does
      i. `ORG`
      ii. `EQU`
      iii. `LIST`
      iv. `END`
   (c) explain what each of the following C pre-processor directives does
      i. `#ASM ... #ENDASM`
      ii. `#ORG`
      iii. `#DEVICE`
      iv. `#INLINE`

8. Identify the comments, operands, mnemonics, labels, and directives in the following PIC16F877 assembly code:

```
C       EQU     0x20    ; store the variable C at location 0x020

        movlw   0x04    ; initialise W to 0x04
loop    movwf   C
        addwf   C, W
        goto    loop
        sleep

        END
```

9. Challenge: In your own words, explain the need for
   (a) the `#PRAGMA` C/C++ directive.
   (b) a **standard** hex file format

10. Role Play/Challenge: Write a program (or use role play) to illustrate how an Intel Hex File generated by MPLAB can be "dis-assembled" into assembly language mnemonics.

Lab 1

ID# ______________

You should complete the pre-lab before coming to your lab session. You may not be allowed in the lab if your pre-lab is incomplete. You will have 3 hours in the lab to complete the exercises. All answers should be written on this lab-sheet in PEN. Please do not attach any extraneous pieces of paper.

At the end of this unit the student will be able to:

utilize the MPLAB IDE, CCS compiler and other software tools, to develop software for the MicroChip PIC16xxx series of microcontroller, in C/C++ and assembly language.

Pre-Lab

Please read through the entire lab script FIRST, so you know what is expected of you, and THEN complete the following questions.

1. Please identify
   - your lab group letter (E,F,G,H): ______________
   - your ECNG2006 uP group designator (e.g. A3): ______________
2. In your own words, differentiate between an integrated development environment (IDE) and a simulator. Hint: what do they input/output? what features do they offer? You should use examples to support your answer. *5 marks*
3. Simulators come in three “flavors”:

   Groups A1,B1,C1,D1,C4 instruction-level,

   Groups A2,B2,C2,D2,B4 cycle-level,

   Groups A3,B3,C3,D3,A4,D4 gate-level.

   In your own words, debate why it is advantageous to simulate at the level assigned to your group, rather than at either of the other two levels. *5 marks*

4. For the Microsoft Windows 2000 operating system:

   (a) How do you open a “command window”? *1 mark*

   (b) What is the meaning of the term “command prompt”? *1 mark*

   (c) What is the meaning of the term “command line parameter”? *1 mark*

   (d) How do you specify command parameters when you are using the “command prompt”? *1 mark*

   (e) How do you specify command parameters when you are using the Windows GUI? *1 mark*

   (f) What is the meaning of the term “environment variable”? *1 mark*

   (g) How do you set, clear OR examine the values of environment variables when you are at the “command prompt”? *1 mark*

   (h) How do you set, clear OR examine the values of environment variables when you are using the Windows GUI? *1 mark*

5. In this lab we will be looking at different tool-chains which can be used with the PIC16F877 under Windows 2000. For your assigned tool, identify and explain TWO features (apart from price) which make it superior to the competing tool in it's category. You should support your arguments using user testimonials which are not supplied by the tool creators. Cite all references. *7 marks*

Group(s)	Tool	Category	Tool	Group(s)
A1	MPLAB	IDE	PICC	C2, A2
B4	SDCC	Compiler	CCSC	D1
A4, C4	GPLINK	Linker	MPLINK	B3
B1	MPASM	Assembler	GPASM	C3
B2	MPSIM	Simulator	GPSIM	D4
C1, A3	Command Batch files	Script	MinGW makefiles	D3, D2

6. Algorithms are descriptions of how a function can be achieved. in this lab, we will be using a particular algorithm to calculate the logarithm of an integer. It is based on the principle of comparing the number to an incrementally increasing exponent of the base. This is not the only possible algorithm.

   Groups A1,B1,C1,D1,C4 Borcharts algorithm
   (e.g. http://www.dattalo.com/technical/theory/logs.html)

   Groups A2,B2,C2,D2,B4 Exploits floating point representation
   (e.g http://wehner.org/fpoint/)

   Groups A3,B3,C3,D3,A4,D4 Uses a lookup-table
   (e.g. http://www.dattalo.com/technical/software/pic/piclog.html)

   Read about the alternative algorithm you have been assigned, and contrast it with the one used in this lab. Use the calculation of $\log_5 25$ to support your answer. *5 marks*

Software Installation

Students may wish to install the same software on their own machine. A CD can be obtained from Mr. Hall, or the software can by downloaded from their respective sites. To avoid problems, please install software in the specified order. Please note that the OS in the lab is Windows 2000 and it is installed on the `D:` drive. Administrative privileges are required for a proper installation. We will not be able to give support for problems experienced under other operating systems.

Microchip MPLAB IDE Depending on the version, the default installation directory is either `\ Program Files\ Microchip` OR `\ Program Files\ MPLAB` OR `\ Program Files\ MPLAB IDE`.

Please create a directory called `Third Party` within the installation directory. All other software should be installed here.

CCS PCWH Please install in a subdirectory of your MPLAB installation; for example `\ Program Files\ Microchip\ Third Party\ PICC`. Installation may complain that it cannot find the REG files; if so please use the files in the installation directory of the CD. If the installation either fails to find MPLAB.INI for versions of MPLAB greater than 6.0, or fails to set data in registry, then you should install the plug-in.

CCS MPLAB plug-in Only required for MPLAB versions greater than 6.0. Please Install in the same directory as the CCS PCWH i.e. `Third Party\ PICC`

GPUTILS Please install in `Third Party\GPUtils`

GPUTILS MPLAB plug-in Only required for MPLAB versions greater than 6.0. Please Install in the same directory as the GPUTILS i.e. `Third Party\ GPUtils`

GPSIM Please install in `Third Party\ GPSIM`

SDCC Please install in `Third Party\ SDCC`

MinGW Make Please install in `Third Party\ MinGW`

PIC16F877 Visual Simulator Please install in `Third Party\ Visual Simulator`.

In the Lab

Before you start, please ensure that the computer is set up to show file extensions:

- Double click on My Computer then select the following from the menu(s) and pop-up windows: `Tools -- Folder Options -- View`.
- Ensure that the checkbox labelled `Hide file extensions` is clear.
- Click OK.

Next, create a directory on the `D:` drive . This is the directory you will work with during the lab.

- In the My Computer window, double-click on the icon for the drive.
- Right click and select New Folder.
- Name the directory `uPxxxxxxxx` where `xxxxxxxx` is your ID number.

Finally, unzip the archive `ecng2005-uP-lab1.zip` into the directory you have just created.
Please ENSURE that you delete your directory at the END of the lab (you can copy it to your Z: drive or a key first!)

Command Line Basics

1. Open a command window. Tick each of the following items as you try it:

   - The prompt tells you which directory is currently active. e.g. `D:\myfolder`
   - To change directories we type the command `cd` followed by the name of the directory to which we wish to move e.g. `cd \anotherfolder`. The directory name is the *command line parameter* that we supply for the command `cd`.
   - To change drives we type the new drive letter followed by a colon. e.g `Z:`
   - If we do not specifiy the full *path* of the directory then we will move to a subdirectory of the current directory. We also have directory shortcuts `.` and `..` which stand for the current and parent directories respectively.

   Now lets get to work!

   (a) Use the `cd` command to make the folder you created your active directory. Use the space below to make notes to help you remember how to do this.

   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (b) Within a directory we can list the files and subdirectories using the `dir` command. The `dir` command also works if we give it the path to a directory e.g. `dir C:\otherfolder` Use the `dir` command to list the files in another directory . Use the space below to make notes to help you remember how to do this.

   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (c) At the command line we often cannot remember all the options for a command. Most commands have help. To display help for a command, try typing the command name followed by `-h` , `/h`, or `/?`. Not all of these options will work with all programs, but usually at least one will.

   Try `dir /?`. Which options are available for the `dir` command? Choose one and use it. Write a short description of what you did and what happened.

   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(d) We can print the contents of a file to the screen using the command `type`. Print the contents of any text file (we've supplied `lab1read.asc`) in the command window.

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(e) We can even start applications. To start Windows Notepad type `notepad`. You should close the instance of Windows Notepad which just opened before proceeding.

If you have a sequence of commands that you execute regularly, it can be tedious to type and retype them. You can place a sequence of commands in an ASCII text file called a script or BATCH file. When you type the name of the batch file, the commands will be executed in sequence.

Use the `dir` command to look in the directory for files with the `.bat` extension. Type the name of the file (excluding the `.bat` extension) at the command prompt.

What happened?

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

Edit the batch file (you can use Windows Notepad) and change it so that it uses the commands you have learned to change active directory to your folder, and type out the contents of `lab1read.asc`, regardless of the directory from which it is called. Save and test your batch file. Write down the final batch file commands here.

Show your lecturer or TA and have them sign/stamp your script here. *2 marks*

(f) Type `set` at the command line to see ALL the environment variables which are currently set. You can also use the `set` command to create and alter environment variables.

When you set or clear environment variables at the command line, the changes only affect that command window. If you open another window, the changes will not be there. Close your current command window. Open a new window and have a look.

If you ever need/want to make global changes to environment variables, right-click on "My Computer","Advanced","Environment Variables".

Create an environment variable called ECNG2005 and then change the value from `Test` to `Ok`. What command(s) did you use?

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(g) Environment variables are useful; they often specify default values to applications. Some environment variables actually control how the command window/computer behaves.

For example the PROMPT environment variable controls how the command prompt is displayed. You may use the `prompt` or `set` commands to alter it. Change the prompt so that it displays your name before the drive letter and path. What command did you use?

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(h) The PATH environment variable lists all the places (search paths) that the computer will look for a program. If you type the name of a program which is not in the current directory, and not in one of the search paths, then the computer will complain. Start Windows notepad by typing `notepad` (or `notepad.exe` at the prompt). Try clearing the path variable (`set PATH=`). Does `notepad` still start from the command prompt?

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

Please close all command windows before proceeding.

Build Tools

Compilers take ASCII text files, that contain source code, as their input. You can edit any source code file in any ASCII text editor. This is IMPORTANT! There is no need to cut and paste code between different editor applications - simply close the file, then open the same file in another editor.

Go ahead any create a Visual C/C++ project. You will be able to open the resulting `*.c` or `*.cpp` files in Windows Notepad, change them, and then use the changed file in Visual C/C++.

If you open a file that is not ASCII text in any text editor, it will look like “junk”.

Windows Notepad is an Editor. Visual C/C++ is an Integrated Development Environment that includes an editor. When you click build, it passes a command (just like the ones we have just used) to start the compiler.

We are going to investigate compilers for the PIC16F877.

2. Open an ASCII text editor (e.g. Windows Notepad) and examine the code in `looplogm.c`. This code will calculate the base-`n` logarithm of the integer `x`. The initial value of `n` should be 3, and the initial value of `x` should be 81. If not correct and save the file.

   Make two subdirectories called SDCC and PICC in your working directory. Place a COPY of `looplogm.c` in each of your directories.

   (a) Open a new command window. In this window we will utilise the CCS PIC-C compiler to compile the code.

   - You need to set the PATH environmental variable so that `ccsc.exe` to run correctly. The command will depend on the path to the PICC directory on your computer. A typical command would look like:
     ```
     set PATH=%PATH%;D:\Program Files\MicroChip\Third Party\PICC
     ```
   - You need to change the active directory to PICC subdirectory of your personal folder. A typical command would look like `cd D:\uP12345678\PICC` Edit the copy of `looplogm.c` in this directory. Be sure to include a `#DEVICE` line in your file to specify the target processor (PIC16F877).
   - Run the compiler command: `ccsc.exe looplogm.c +FM +P`. What was displayed? *1 mark*

     If you have any errors, ask your lecturer/TA for assistance.
   - Once your file has compiled correctly, look at your folder. List the names and extensions of the files which have been created. For each file: Indicate whether it is an ASCII text file, and if it is, write out one/two lines from the file. (Try to choose interesting ones!)

Filename	Filesize	ASCII?	Extract (if ASCII)
looplogm.c			
looplogm.sym			
looplogm.hex			

Show your lecturer or TA and have them sign/stamp your script here. *2 marks*

- We could also create a batch file to run CCSC compiler. In your batch file called `myccsc.bat` place the commands you have just learned to set the path, change the directory, and run ccsc on `looplogm.c`.
  If you replace the name of your file within the batch file with `%1` then you can specify the file to be compiled along with the name of the batch file at the command line. For example `myccsc myfile` to compile `myfile.c`.
  Test your final batch file and write it below. Test your batch file in a new command window.

  Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

Close all command windows.

(b) Open a new command window. In this window we will utilise the SDCC compiler to compile the code.

- You need to set the PATH environmental variable for `sdcc.exe` to run correctly. How are you going to do this? *1 mark*

- You need to change the active directory to the SDCC subdirectory of your personal folder. How are you going to do this? *1 mark*

- Make sure there is a copy of `looplogm.c` in the SDCC directory. No preprocessor directives are needed for this compiler. Unlike the #DEVICE needed for PICC, we will specify the processor on the command line for SDCC.
- Run the compiler command: `sdcc -S -V -mpic14 -p16F877 myfile.c`. Please note that the command line options are case sensitive. What was displayed? *1 mark*

  If you have any errors, ask your lecturer/TA for assistance.

- Once your file has compiled correctly, look at your folder. List the names and extensions of the files which have been created. For each file: Indicate whether it is an ASCII text file, and if it is, write out one/two lines from the file. (Try to choose interesting ones!)

Filename	Filesize	ASCII?	Extract (of ASCII files)
looplogm.c			
looplogm.asm			
looplogm.d			

Show your lecturer or TA and have them sign/stamp your script here. *2 marks*

- We could also create a batch file named `mysdcc.bat` to run the SDCC compiler. Try it below!

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

Close all command windows.

The final file which is required for “download” to the PIC16F877 microcontroller is an Intel HEX format file `*.hex"`.

You should have noticed that although both compilers generated files, only CCSC created a `".hex` file. CCSC accomplished the compiling, assembling, and linking in one step.

SDCC generated an assembly language file `*.asm`. To complete the job for SDCC we would need to use a separate assembler and a linker.

Assemblers and linkers can also be used for code which you write directly in assembly.

3. Alter your SDCC batch file so that the search path for `gpasm.exe` and `gplink.exe` is added. They are located in `Third Party\GPUtils\`.

   Add the following lines at the end of your batch file: (These commands are available in the file `batfileadd.txt`; you may need to change `looplogm` to `%1` if you have used that option previously).

   - `set GPUTILS_HEADER_PATH=D:\Program Files\MicroChip\Third Party\GPUtils\header`
   - `set GPUTILS_LKR_PATH=D:\Program Files\MicroChip\Third Party\GPUtils\lkr`
   - `gpasm -c looplogm.asm`
   - `gplink -m -o looplogm.hex looplogm.o` (line wraps)
     "`D:\Program Files\MicroChip\Third Party\SDCC\lib\pic\libsdcc.lib`"

   Run the SDCC batch file in a new command window. If you have any errors, ask your lecturer/TA for assistance.

   (a) Once your file has compiled, assembled and linked correctly, look at your folder. List the names and extensions of the ADDITIONAL files which have been created by `gpasm` and `gplink`. For each file: Indicate whether it is an ASCII text file, and if it is, write out one/two lines from the file. (Try to choose interesting ones!)

Filename	Filesize	ASCII?	Extract
looplogm.hex			
looplogm.o			

Show your lecturer or TA and have them sign/stamp your script here. *2 marks*

4. Compare the files generated by the two processes from the same original C file. Are they identical? Did you expect them to be? Explain why you did or did not think so.

Show your lecturer or TA and have them sign/stamp your script here. *4 marks*

Build Tools Roundup - Make Utility

5. (a) The batch file works well for a single file, but where we want to establish rules for compiling files it is often more productive to use a makefile. Using this makefile we can specify different targets, and how they come together to form the final executable. We will only use a simple makefile here (an electronic copy of `makefile` is available).

```
%.asm:  %.c
                sdcc -S -V -mpic14 -p16F877 $<
%.o:  %.asm
                gpasm -c $<
%.hex:   %.o
                gplink -m  -o $@ $< \\
                D:\Program Files\MicroChip\Third Party\SDCC\lib\pic\libsdcc.lib
```

Each rule starts in the first column (no space or TAB) and specifies the Target. Our targets all start with % which is a wildcard i.e. the rule will match any file which has the same suffix. This is followed by a list of dependencies (files that must exist in order to produce the target) . Then the following lines (each starting with a TAB) list the command(s) used to make the target ($@) from the dependencies($<).

- Check the rules in the file called "makefile" (no extension) in the SDCC directory.
- Open a command window and add the paths for `sdcc`, `gputils`, and `mingw\bin` to the search path.
- Make the SDCC directory the active directory.
- Test your makefile is syntactically correct by using the command:`mingw32-make`. If you have any errors, ask your lecturer/TA for assistance.
- Type the following command: `mingw32-make myfile.hex`

Delete `myfile.hex` and try it again. *4 marks*

Show your lecturer or TA and have them sign/stamp your script here.

You should notice that make automatically builds only the files that need to be updated. This is one advantage over using scripts or batch files.

(b) Challenge(Bonus 10 marks): Place a makefile in the CCSC sub-directory that could be used to build a hex file using the `ccsc` compiler. Write your rule(s) down here.

Show your lecturer or TA and have them sign/stamp your script here.

Development Tools

We have looked at tools used to build the hex file from the source code. Now we will look at tools which assist in the development of correctly functioning programs. The first of the development tools we will look at is the simulator.

Even though we have compiled the code on a HOST PC, the output file needs to be loaded onto a TARGET microprocessor in order to run.

The HOST PC cannot normally run this file by itself. A simulator is a program which will allow the HOST PC to interpret the machine code intended for the TARGET processor.

This is a convenient alternative for the developer, as software development can start even though the hardware is not available.

6. GPSIM is a Windows based simulator for the PIC16F877. To start gpsim,

   - Open a new command window.
   - Change the active directory to SDCC (or CCSC for next question!).
   - Add the search paths for gpsim, and gputils.
   - Use the following command to start gpsim: `gpsim -p pic16f877 looplogm.hex`

   You will see text printed in the command window and a button bar will pop up.

   (a) Open the Program Memory window, using the menu on the button bar. This has two tabs, one tab shows the assembly language code, with each instruction next to it's stored address, and equivalent instruction code, while the other shows the bytes of instruction code stored in the program memory in a matrix form. What range of instruction memory locations are occupied by `looplogm.hex`. ______________
   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (b) Open the RAM window, using the menu on the button bar. This is a matrix view of the data memory.
   You can step through the assembly language code using the step button. The command window reports the status of the processor with each step. By looking at the RAM window while you step, you can see how information moves from one register to another in response to the instructions.
   How many data memory locations are being used by `looplogm.hex`?______________
   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (c) It is inconvenient to keep stepping through a program. To move more quickly, set a BreakPoint by double clicking on a program line in the Program Window. Reset the processor (using the button on the button bar), then run. The processor should stop before executing the line you have indicated.
   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (d) Open the StopWatch Window. This is used to determine how many cycles the code takes to run. Play with the Stopwatch and the breakpoint to see how long the code takes to execute.
   How many cycles are being used by `looplogm.hex`?______________
   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*
   Close GPSIM when you are done.

7. Repeat the process described in 6 to measure the number of instruction memory locations, data memory locations, and cycles required by the version of `looplogm.hex` generated using CCSC (instead of the one generated using SDCC). *3 marks*

   Show your lecturer or TA and have them sign/stamp your script here.

8. The PIC16F877 Visual Simulation was written to illustrate what goes on inside the micro-controller on each clock cycle i.e. it is a cycle-level simulator. GPSIM is an instruction-level simulator. It does not show us what goes on within the instruction-cycle.

   When you launch the PIC16F877 Visual Simulation, the main window will look like the layout diagram shown in your data-sheet book.

   Just like the GPSIM simulator, this simulator loads in the Hex file output by the compiler.

   - Launch the PIC16F877 Visual Simulation from the Windows Menu.
   - To load in a hex file, Click on **TASKS**, **DECODE HEX File**; a window named "Decode Hex File" will come up; you should Choose **Select File to Decode**. Locate either of your compiled hex files and press OK. In the "Hex Data" Field, you will see the actual contents of the HEX file.
   - When parsed (using the Intel Hex format) the byte-codes to be loaded at particular addresses are displayed in the "Parsed data" Field.
   - Click the **DECODE** button, to interpret the instruction op-codes & operands as mnemonics.

   (a) Does this match the GPSIM views of program memory? Is that what you expected? Explain your answer. *1 mark*

   (b) Click the **Back to the interface** button and run the simulator by Clicking the **start processing** button. You will see two dots moving around the screen.

   - Blue Dot represents information flow due to the fetch phase of the pipeline.
   - Red Dot represents information flow due to the execute phase of the pipeline.

   Each phase is made up of 4 sections, Q1 to Q4, the duration of which is set by the clock. The currently executing phase is displayed on the left. Pay particular attention to what happens when a `goto` instruction is executed. How is pipeline execution affected? *2 marks*

You should close/pause the PIC16F877 Visual Simulation when you are finished.

Build & Development Tools Roundup - The Integrated Development Environments(IDE)

An IDE allows us to perform all the tasks we have looked at from a single interface. Some IDE's are supplied with their own tool-chain, while others contain no tools and must be configured to work with your tools. Environmental variables, batch files, scripts, makefiles, and command line parameters can all be used with an IDE to make it operate seamlessly with a tool-chain. Instead of manually calling separate programs to translate from `.c` to `.asm` to `.o` to `.exe`; we can automatically call appropriate programs.

The MPLAB IDE is an example of an IDE that comes with it's own tools, but allows you the option to choose other build tools (like the CCS compiler), and customize environmental variables, and command line parameters.

9. (a) Start the MPLAB IDE from the menu. Like most Windows based IDE's there is a menu, toolbar, workspace and a status bar. Create a new project in your CCSC subfolder.

   - Click **PROJECT** on the menu bar and select **PROJECT WIZARD**. In the **PROJECT WIZARD** box click **NEXT**. From the **DEVICE** drop down menu select **PIC16F877** and click **NEXT**. Select **CCS C COMPLIER** form the **ACTIVE TOOLSUITE** dropdown box and then click **NEXT.**
   - Enter "**myproject**" in the box named **PROJECT NAME**. Then browse and find the CCSC folder you created for the **PROJECT DIRECTORY** box and click **NEXT**.
   - A window will now come up with the buttons **ADD** and **REMOVE**, add your file `looplogm.c` from the `CCSC` directory and then click **FINISH** in the new window.
   - Then go to **WINDOW** on the menu bar and select "**project1.mcw**". You will see "**looplogm.c**" . Now double click the "**looplogm.c**" link and your program will appear.

   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

   (b) The MPLAB IDE has a built in editor. It can be used to view and edit any ASCII-type file. Try opening different files in the CCSC directory.

   Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(c) The MPLAB IDE also has a built in simulator. Go to the menu bar and select **DEBUGGER > SELECT TOOL > MPLAB SIM**. This means that you will be executing your code on the simulator. You should be able to use the files which have already been generated. To display the Program Memory click on the **VIEW > PROGRAM MEMORY** menu item. The buttons at the bottom left of the window, allow you to access three different views of the Simulator Program Memory. State the name of each the view, and identify how the views differ from each other. Are these views similar to the ones we had in the other two simulators? *2 marks*

(d) The PC value displayed in the status bar is set to 0x00. Let us investigate how to advance the PC to the next instruction location. To advance to the next instruction location click on the **DEBUGGER** menu and choose **STEP OVER** from the menu. What do we see happening? Is this similar to what happened in the other two simulators?

Show your lecturer or TA and have them sign/stamp your script here. *2 marks*

(e) The MPLAB IDE project has been set up to use the same CCSC compiler we used at the command line. Click on the **PROJECT > BUILD ALL** menu item (or press F10), to compile the project. A build window should appear stating the build was successful. Have a close look. Can you see the command used to launch the compiler? Write it below.

Show your lecturer or TA and have them sign/stamp your script here. *1 mark*

(f) In the GPSIM simulator we used breakpoints, a view of RAM and a stopwatch. Locate the equivalent functions in MPLAB.

Show your lecturer or TA and have them sign/stamp your script here. *3 marks*

Performance Comparison

So far we have compared the sizes of files, the number of instruction and data memory locations used, and the number of cycles that the program requires to run.

The performance of the final `*.hex` file depends on:

- the algorithm implemented
- the compiler/assembler used to generate the `*.hex` file
- the language and data types used for implementation

We will investigate each of these in turn, using the skills we have acquired during this lab. The comparison criteria to be used are:

I the size of the source code file in bytes

II the number of instruction memory locations used by the final program

III the number of data memory locations used by the final program

IV the number of instruction cycles required to execute the program on test case A

V the number of instruction cycles required to execute the program on test case B

The test cases to be used are A: $\log_3 81$ **and B:** $\log_5 25$**.**

10. Challenge Question: (Bonus 10 marks) Use the test cases to discuss WHY the algorithm you researched for the pre-lab is likely to perform better or worse than the algorithm implemented in `looplogm.c`. In addition to the specified comparison criteria, you could also consider the accuracy/suitability of the final solution.

Item	I	II	III	IV	V
Units	bytes	instruction-words	data-words	cycles	cycles
`looplogm.c` with CCSC					
`looplogm.c` with SDCC					
`liblogm.c` with CCSC					
`liblogm.c` with SDCC					
`asmlogm.asm`					

Table 1: Performance comparison results

11. Use data from previous pages, and modify the programs and re-run them to obtain data to update the first two rows in Table 3. *2 marks*

    Demonstrate the changes made to your TA/lecturer, who will sign/stamp your script as confirmation.

12. The program `liblogm.c` uses the compiler provided math libraries to calculate the answer. Compile and/or assemble programs as necessary to fill the second two rows of the Table 3. *10 marks*

    Demonstrate this to your TA/lecturer, who will sign/stamp your script as confirmation.

13. Make a new project in MPLAB to assemble `asmlogm.asm`. You may use either the MPASM or the GPASM assembler. Make the measurements needed to update Table 3. *5 marks*

    Demonstrate this to your TA/lecturer, who will sign/stamp your script as confirmation.

14. Use your data to identify which is the most effective implementation of logarithm. Justify your choice, and explain whether this is what you had expected. *3 marks*

Total marks 100.
This exercise is worth 5% of your ECNG2005 lab mark.

Unit 10 Write the number you have been assigned here______.
Answer the following questions for the TARGET platform whose number you were assigned.

1. What is the name of the target platform? ______________.
2. Where did you find the information? ______________.
3. Who is the manufacturer of the tool-chain? ______________.
4. What is the price of this tool-chain? ______________.
5. Which HOST platform/operating system(s) can use this toolchain? ______________.
6. Which of the following items does the tool-chain contain?

- IDE ☐
- make utility OR makefiles ☐
- batch/shell files OR other scripts ☐
- assembler ☐
- disassembler ☐
- linker ☐
- compiler ☐
- pre-defined header files ☐
- pre-defined library files OR librarian ☐
- simulator ☐
- downloader ☐
- debug kernel + monitor ☐
- other ☐

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this lab exercise.
  - utilize the MPLAB IDE, CCS compiler, and other software tools, to develop software for the MicroChip PIC16Cxxx series of microcontroller, in C/C++ and assembly language.
- Which aspect of this lab exercise did you have the most difficulty understanding?
- Which aspect of this lab exercise did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this lab exercise.
- Identify one way in which this lab exercise could be improved.

PIC16 Introduction

In this section, we have looked at the basic features of the PIC16F877(memory layout, instruction cycle, instruction set), and the MPLAB/CCS tool-chains (directives and keywords). While these are specific to the particular items, they are typical of the features you will find available in many microprocessors/microcontrollers, and their associated tool-chains.

In examining these features we have furthered the first learning objective, as you can now: **"given relevant diagrams and specifications for a particular system, identify, and describe the role of, the components of a microprocessor, a microprocessor-based system, a development suite (hardware, software) for a microprocessor-based system."**

In the following section, we will look at how data of various types can be represented and manipulated by microprocessors.

Part IV
Numbers & Data

There are several issues connected with the manipulation of real numbers, strings and other forms of data by computers.

The first is how such data should be represented; the second is the means by which the data should be manipulated; and the third is the bit/byte/word/record-ordering of data within memory to accommodate the representation.

In this section, we will address these issues for two's complement, binary-coded decimal, and floating point representations of numbers, ASCII representation of strings, and storage of composite records.

11 Real numbers

At the end of this unit the student will be able to:

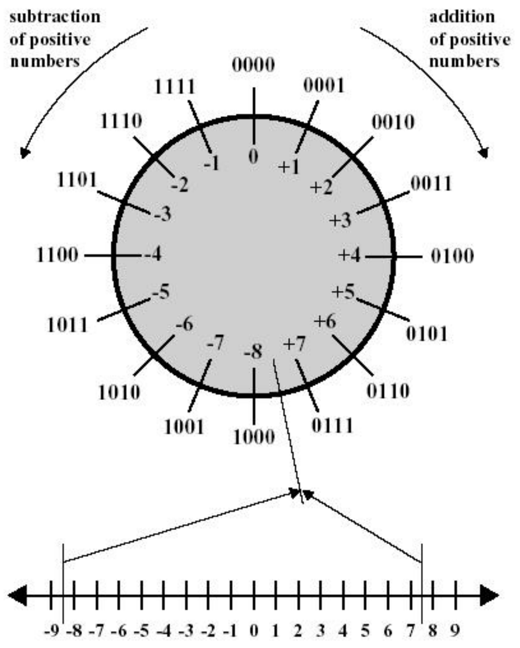
represent real numbers using fixed and floating point binary representations.

Two's Complement The most commonly found representation of integer numbers in computers is the two's complement representation. In this representation, the full range of binary numbers (2^n for n bits) is split between positive and negative numbers (-2^{n-1} to $2^{n-1}-1$). Positive numbers are the same as their unsigned representations. To find the representation of a negative number, invert the bits of the unsigned binary representation and then add 1. Using this representation, all subtractive operations become additions with a negative number. Because the uppermost bit of a two's complement number is always 0 for positive numbers[22], and 1 for negative numbers, this bit is often referred to as the sign bit.

Changing the bit width of a two's complement number can be done by sign reduction or extension i.e. removing locations which are the same as the sign bit, or copying the sign bits to any additional bit locations. In practice, representation of a number using more bits involves the use of multiple data words.

Binary Coded Decimal Binary coded decimal representation of numbers, uses a 4 bit binary number to represent the decimal digits 0-9. It is clearly an inefficient representation, as 6 of the numbers which could be represented by 4 bits are never used. The advantage of using such a scheme lies in the fact that most human I/O involves the use of decimal numbers. The use of BCD reduces the overhead of conversion at the interface-level. The disadvantage is that mathematical operations become more complex, as the manipulation of individual binary digits may yield an invalid 4 bit BCD code. Further there is no accommodation for signed quantities. Subsequently BCD arithmetic

[22]Zero is assumed to be a positive number.



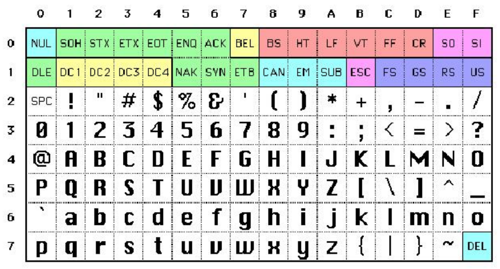
BCD

- Decimal: Facilitates easy conversion to/from human input
- BCD: Each digit represented by an n-bit binary number
- Packed BCD: 2 digits represented in a single byte

$$\underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}}_{5}\underbrace{\boxed{1}\boxed{0}\boxed{0}\boxed{1}}_{9}$$

From: http://www.mindspring.com/~jc1/serial/Resources/ASCII.html

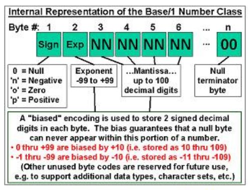
ASCII



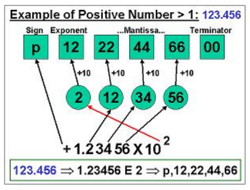
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

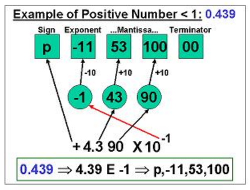
- Visible Character Codes
- Control Codes
  - Physical Communication Control
  - Logical Communication Control
  - Physical Device Control
  - Physical Device Format Effects
  - Code Extenders
  - Information Separators

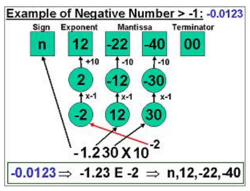
http://www.boic.com/numrep.htm

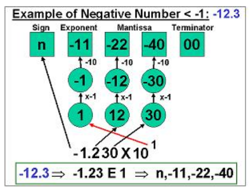

Internal Representation of the Base/1 Number Class
Byte #: 1 2 3 4 5 6 ... n
Sign Exp NN NN NN NN ... 00
0 = Null
'n' = Negative
'o' = Zero
'p' = Positive
Exponent -99 to +99
...Mantissa... up to 100 decimal digits
Null terminator byte

A "biased" encoding is used to store 2 signed decimal digits in each byte. The bias guarantees that a null byte can never appear within this portion of a number.
• 0 thru +99 are biased by +10 (i.e. stored as 10 thru 109)
• -1 thru -99 are biased by -10 (i.e. stored as -11 thru -109)
(Other unused byte codes are reserved for future use, e.g. to support additional data types, character sets, etc.)


Example of Positive Number > 1: 123.456
Sign Exponent ...Mantissa... Terminator
p 12 22 44 66 00
+10 +10 +10 +10
2 12 34 56
+1.23456 X 10^2
123.456 ⇒ 1.23456 E 2 ⇒ p,12,22,44,66


Example of Positive Number < 1: 0.439
Sign Exponent ...Mantissa... Terminator
p -11 53 100 00
-10 +10 +10
-1 43 90
+4.3 90 X 10^-1
0.439 ⇒ 4.39 E -1 ⇒ p,-11,53,100


Example of Negative Number > -1: -0.0123
Sign Exponent Mantissa Terminator
n 12 -22 -40 00
+10 -10 -10
2 -12 -30
x-1 x-1 x-1
-2 12 30
-1.2 30 X 10^-2
-0.0123 ⇒ -1.23 E -2 ⇒ n,12,-22,-40


Example of Negative Number < -1: -12.3
Sign Exponent Mantissa Terminator
n -11 -22 -40 00
-10 -10 -10
-1 -12 -30
x-1 x-1 x-1
1 12 30
-1.2 30 X 10^1
-12.3 ⇒ -1.23 E 1 ⇒ n,-11,-22,-40

```
The Base/1 Number Class uses a proprietary
internal representation to support arithmetic on
very large numbers and exact fractions of up to
100 decimal digits. Base One's design makes it
possible to store high precision numbers
compactly without sacrificing computational
efficiency. (U.S. Patent Number 6,384,748)
```

operations are generally restricted to addition and subtraction of unsigned quantities. In *packed BCD*, two digits are continued within a single byte. Other representations *pad* the extra bits in the byte/word (i.e. leave the upper bits of the data word either unused or filled with a particular prefix) e.g. the ASCII[23] character set defines a representation of characters as 7 bit numbers. The digit characters 0 through 9 are represented as $011\ 0000_2$ thru $011\ 1001_2$.

Fixed point representations Integer representations are specific cases of fixed point representations i.e. numeric representations in which the position of the decimal point is assumed to be fixed; the digits above and below the decimal point are specified.

Floating point representations represent numbers using an assumed numeric *base* i.e. all numbers in a particular system, are presumed to have the same base. Numbers are represented by an *exponent* of the base, multiplied by a normalised number called the *mantissa* or *significand.* i.e. $\pm S \times B^{\pm E}$. Normalised numbers are justified so that the first significant digit appears at a fixed place e.g. 0.00001 and 0.1 and 100 may be expressed as 1×10^{-5}, 1×10^{-1} and 1×10^{2}.

The presence of sign in the significand or exponent may be represented separately, or by use of a *biased* numeric representation. A biased numeric representation is one which includes a fixed offset e.g. for an 8 bit number with a bias of 0x7F, the number -4 would be represented as 0x7B and the number 4 would be represented by 0x83.

The base used, the use of biased numbers vs. explicit sign representation, the number of bits used to represent the mantissa and exponent, and the ordering of bits within the representation depends on the particular system/designer/programmer. In computer systems the conventions are:

- the base is usually 2,
- the exponent is biased, and
- the sign of the significand is explicitly represented.

All fixed/floating point representations, regardless of the details, share the following flaws:

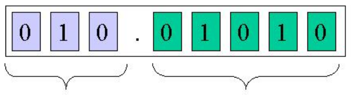
Representation error The precision with which a floating point number can be represented is determined by the number of bits used in the exponent and significand. Where the number of bits is insufficient, the number be be approximated by roundoff, or truncation. In either case there is some discrepancy between the number and it's representation.

Error propagation Arithmetic operations often magnify the effects of errors in the input quantities. e.g. 2.51 * 2.32 = 5.8232 BUT 2.5 * 2.3 = 5.75.

Non-unique 0 The floating point representation scheme is such that there is a choice of representations for zero.

[23] American Standard Code for Information Interchange

Fixed point

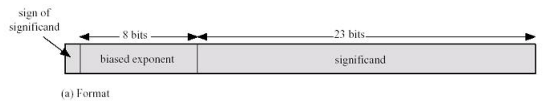


- "Artificial" "binary point"
  - within the data-word
  - placement MUST be specified
- Bits on either side of "binary point" may be interpreted as integers or BCD
- Right side
  - BCD lists decimal fraction numerals
  - Integer uses fraction of maximum integer; in effect listing the binary fraction numerals e.g. .01 is 1 of 2; .010 is 2 of 4

Fixed point translation

Signed fixed point	Two's complement representation of signed fixed point
$+01001.001_2$	01001001_2
$-\ 01001.001_2$	10110111_2
$+9.2_{16}$	49_{16}
$-\ 9.2_{16}$	$B7_{16}$
$+9.125_{10}$	73_{10}
$-\ 9.125_{10}$	183_{10}

When translating a negative fixed point number, keep in mind that the fraction is always positive e.g. 10110111_2 where we know we have 3 bits after the point.
The integer portion is 10110_2 in a 5 bit two's complement representation scheme i.e. $-\ 01010_2$ which can be written as -10_{10}
The fractional portion is 111_2, i.e. $^7/_8 = 0.875_{10}$
$-\ 10_{10} + 0.875_{10} = -\ 9.125_{10}$



(a) Format

```
 0.11010001   2^10100  = 0 10010011 10100010000000000000000
-0.11010001   2^10100  = 1 10010011 10100010000000000000000
 0.11010001   2^-10100 = 0 01101011 10100010000000000000000
-0.11010001   2^-10100 = 1 01101011 10100010000000000000000
```

(b) Examples

Figure 8.18 Typical 32-Bit Floating-Point Format

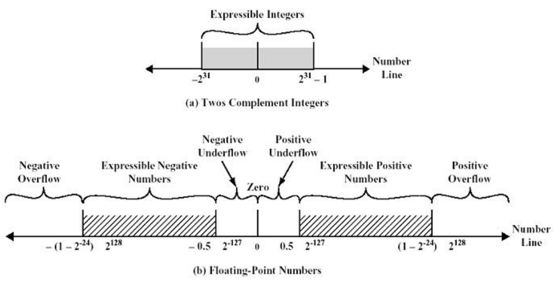
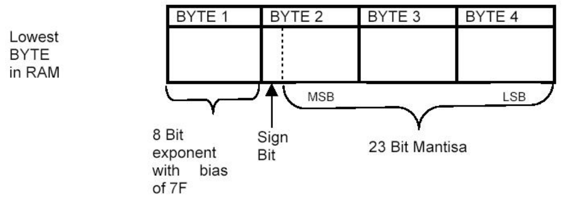


Figure 8.19 Expressible Numbers in Typical 32-Bit Formats



Example Number				
0	00	00	00	00
1	7F	00	00	00
-1	7F	80	00	00
10	82	20	00	00
100	85	47	00	00
123.45	85	48	E6	66
123.45E20	C8	27	4E	53
123.45 E-20	43	36	2E	17

Lowest BYTE in RAM

Changing bit-widths

- 2's complement
  - extend/reduce the sign bit
  - arithmetic shift in context
- BCD
  - fill with 0's at front
- ASCII
  - defined as 7-bit
- Floating point
  - fill significand (mantissa) at end
  - add the change in offset to exponent

Interpretation

1010 0011 1101 0101 0011 0111 0011 1010

- Unsigned binary number:
  $A3D5373A_{16}=2748659514_{10}$
- Signed binary number:
  $-5C2AC8C6_{16}= -1546307782_{10}$
- Unsigned fixed point(16/16):
  $A3D5.373A_{16}=41941.2157\ldots_{10}$
- Floating point number(IEEE):
  $-\ 0.\ 110101010011011100111010_2 \times 2^{0100\ 0111_2}$
- Visual C representation:
  2.74866e+009 [A3D53700]
- Questions:
  - How can we tell what type the data is?
  - How can we convert from one data type to another?
  - How can we interpret the data as another type?

Representation

Unsigned integer:
- 01001001_2 written as a binary #
- 49_{16} written as a hexadecimal #
- 73_{10} written as a decimal #

Signed integer:
- $+01001001_2$ written as a signed binary #
- $+49_{16}$ written as a signed hexadecimal #
- $+73_{10}$ written as a signed decimal #
- $-\ 01001001_2$ written as a signed binary #
- $-\ 49_{16}$ written as a signed hexadecimal #
- $-\ 73_{10}$ written as a signed decimal #

Two's complement of an unsigned integer is <u>also an unsigned integer</u>

Two's complement:
- 10110111_2 written as a binary #
- $B7_{16}$ written as a hexadecimal #
- 183_{10} written as a decimal #

Two's complement <u>representation</u> of a signed integer is <u>not</u> the same as the two's complement of an unsigned integer

Two's complement representation of signed integers and the Sign bit

In <u>some</u> representations there is a sign bit which is used to represent the plus + or minus - signs. The two's complement representation scheme does <u>NOT</u> have an <u>EXPLICIT</u> sign bit, but a feature of the scheme is that the first bit will <u>IMPLICIT</u>-ly reflect the sign of the number.

0000	0	positive sign bit
1001	-7	negative sign bit
0111	7	positive sign bit

Signed Integer		Two's complement Representation
$+01001001_2$	$\longleftrightarrow$	01001001_2
$-\ 01001001_2$	$\longleftrightarrow$	10110111_2
$+49_{16}$	$\longleftrightarrow$	49_{16}
-49_{16}	$\longleftrightarrow$	$B7_{16}$
73_{10}	$\longleftrightarrow$	73_{10}
-73_{10}	$\longleftrightarrow$	183_{10}

Review Questions

1. What range of unsigned integer numbers can be represented in 6 bits?
   (a) -2^5 to $2^5 - 1$ ☐
   (b) $-2^5 + 1$ to $2^5 - 1$ ☐
   (c) -2^6 to $2^6 - 1$ ☐
   (d) $-2^6 + 1$ to $2^6 - 1$ ☐
   (e) 0 to $2^6 - 1$ ☐

2. If the byte 11010110_2 represented a packed BCD number, the BCD number would be:
   (a) $-2A_{16}$ ☐
   (b) -56_{10} ☐
   (c) $D6_{16}$ ☐
   (d) invalid ☐
   (e) none of the above ☐

3. If the dataword 11100010 represented a signed integer (two's complement representation scheme), the number would be:
   (a) $+1E_{16}$ ☐
   (b) $+E2_{16}$ ☐
   (c) -17_{16} ☐
   (d) $-1E_{16}$ ☐
   (e) none of the above ☐

4. Convert the following numbers (show your working):
   (a) 25_{10} to 2's complement representation, written as a hexadecimal number.
   (b) -73_{10} to 2's complement representation, written as a binary number.
   (c) -37_{16} to 2's complement representation, written as a binary number.
   (d) -12_{16} to 2's complement representation, written as a hexadecimal number.
   (e) -121_{10} to 2's complement representation, written as a hexadecimal number.
   (f) 97_{16} read as 2's complement representation, to a signed hexadecimal number.
   (g) 5.6_{10} to a binary fixed point representation with 3 bits (integer), and 5 bits (fraction).
   (h) $2.A_{16}$ to a binary fixed point representation with 3 bits (integer), and 5 bits (fraction).
   (i) 97_{16} read as signed binary fixed point representation with 2 bits (integer), and 6 bits (fraction); to a signed decimal number.
   (j) 97_{16} read as unsigned binary fixed point representation with 2 bits (integer), and 6 bits (fraction); to a signed hexadecimal number.

$$1101010111110010_2$$

5. Show how you would derive the base 10 equivalent of the number shown above, using diagrams, and/or calculations, if we interpret the value as:

   (a) an unsigned 16 bit integer number

   (b) a signed 16 bit integer number

   (c) a signed 16 bit fixed point number (9,13)

   (d) a 16 bit floating point number (explicit sign, biased exponent(base 2), significand normalised just after numeric point)

6. A PIC16F877 program needs to use certain non-integer numbers. For each number, suggest an appropriate way in which the number can be represented in a single file register, write down your representation, and determine what error there is between the actual and represented values.

   (a) 2.5

   (b) 0.333

   (c) 10000.345

7. You need to sort a list of signed integer numbers on a PIC16F877. Suggest an appropriate numeric representation for numbers in the list, and justify your answer if:

   (a) we wish to sort numbers by magnitude. e.g. (-5,4,3,-2,1,0)

   (b) we wish to sort from most negative to most positive. e.g.(-5,-2,0,1,3,4)

8. Any fixed/floating point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A' is the stored value approximating the real value A then the relative error r is expressed as: $r = \frac{A-A'}{A}$.(Stallings 2000; 8.21–8.24)

   (a) Represent the decimal quantity +0.7 in the following formats, and determine the relative error for each representation.

      i. floating point format: base = 2; exponent: biased 4 bits; significand 7 bits.

      ii. fixed point format: 2 bits integer; 6 bits fraction.

   (b) If $A = 1.427$, find the relative error if A is truncated to 1.42 and if it is rounded to 1.43.

   (c) Numerical values A and B are stored in the computer as approximations A' and B'. Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors in the factors.

   (d) One of the most serious errors in computer calculations occurs when two nearly equal numbers are subtracted. Consider $A = 0.22288$ and $B = 0.2221$. The computer truncates all values to four decimal digits. Thus $A' = 0.2228$ and $B' = 0.2221$.

      i. What are the relative errors for A' and B'?

      ii. What is the relative error for $C' = A' - B'$?

9. Typically floating point numbers are represented using a minimum of 32 bits.

   The IEEE standard 754 standard defines a ”single” format 32 bit floating point number as a sign bit followed by an 8 bit biased exponent, followed by a 23 bit significand (representing the normalised 24 bit binary fraction 0.1xxx xxxx xxxx xxxx xxxx xxxx).

   Floating point numbers in the CCS compiler for the PIC16F877, place a biased 8 bit exponent first, followed by the sign bit and then a 23 bit significand (representing the normalised 24 bit binary fraction 0.1xxx xxxx xxxx xxxx xxxx xxxx).

   The difference is in the location of the sign bit. Suggest at least TWO possible advantages/disadvantages of placing the sign bit in either position.

10. Role Play/Challenge: Prove that

    (a) it is possible to write down the binary/octal equivalent of a hexadecimal number by individually translating each digit.

    (b) the n’s complement can be calculated for any base-n number (Hint: 2’s complement is for base 2 numbers).

    (c) it is possible to write down the 2’s/8’s complement of a hexadecimal number by individually translating each digit of the 16’s complement.

    (d) (Stallings 2000; 8.29) every real number with a terminating binary representation (finite number of digits to the right of the binary point) also has a terminating decimal representation (finite number of digits to the right of the decimal point)

Assignment B [24]

ID# ____________

1. Choose two different representation schemes for real numbers. **Explain** how each scheme operates, using the number -8.46_{10} as an example. Compare your chosen representation schemes in terms of the possible representation error and storage space requirements. *12 marks*

[24]Students are advised that there is no online version of this assignment, and that electronic submissions will not be accepted without the explicit permission of the course lecturer.

2. Draw a diagram to show how to, and then write a routine (snippet) for the PIC16F877 which will, interpret the upper nibble of the working register as a BCD digit, and return (end) with the digit value in the lower nibble of the working register. Your code should test for/reject impossible values, in which case the value in the working register should be set to 0x0F. Please comment your code appropriately. *12 marks*

3. Draw a diagram to show how to, and then write a routine (snippet) for the PIC16F877 which will, add two floating point numbers, where the floating point numbers are represented using 4 bits (lower nibble) for the unsigned mantissa, and 4 bits (upper nibble) for the signed exponent. Please explain any choices/assumptions you make about your representation. Please comment your code appropriately. *16 marks*

This assignment is worth 2% of your ECNG2006 mark. It contains a total of 40 marks.

Unit 11 Your lecturer will write an 8-bit binary number, and a decimal number on the board. Write the letter you have been assigned here________. The representation schemes are:

A: signed integer (2's cmplt)

B: BCD unsigned integer

C: fixed point (4 bits before point; 4 bits after point)

D: floating point (1 bit sign; 3 bits biased exponent; 4 bits significand)

1. Interpret the binary number using your assigned representation scheme.
2. Express the decimal number using your assigned representation scheme.
3. What range of numbers can you represent using this scheme?
4. What is the smallest difference you can represent using this scheme?

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - represent real numbers using fixed and floating point binary representations.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

12 Integer arithmetic

At the end of this unit the student will be able to:

perform integer arithmetic operations using binary representations of the operands.

Addition Two or three bits can be added together to generate a sum and a bit-carry; two unsigned binary numbers can be added together by considering each pair of bits in turn, and adding them together with the bit-carry from the previous pair.

Subtraction Two's complement notation was introduced in the previous unit. Using this notation, both positive and negative numbers can be expressed. The convenience of this notation, is

- that addition of two's complement numbers can be accomplished using the same method as for unsigned binary numbers.
- that subtraction of two's complement numbers can be accomplished by negating one number, and then performing an addition.

Overflow is not Carry The bit-carry from the final pair of bits is often used to set the carry flag. The interpretation of a set carry flag differs depending on the arithmetic operation carried out. If an unsigned addition generates a carry, it implies that the data-word was not big enough to hold the result (i.e. overflow).

For two's complement numbers, it is not so simple:

- for valid two's complement numbers of *opposing* sign, the carry and sign bits of the addition result will always be of opposing polarity, and overflow will never occur.
- for valid two's complement numbers of the *same* sign, the carry and sign bits of the addition result will be of opposing polarity *iff* overflow has occurred, and of the same polarity *iff* no overflow has occured. (as shown below)

Sign Bit	No Carry	Carry
Reset	No Overflow	Overflow
Set	Overflow	No Overflow

As a result, some processors keep separate carry and overflow flags, others define just one which behaves differently depending on the instruction carried out. Other variants of the carry flag, are the mid-word carry flags. These are set to reflect particular bit-carry's in the middle of the data-word.

Carry & Overflow

- Carry
  - bit that "falls off" after addition

0111 0011 1010 1111 1 0010 0010

- Digit Carry
  - bit that "falls out" of lower nibble during addition

1

0111 0011 1010 1111 1 0010 0010

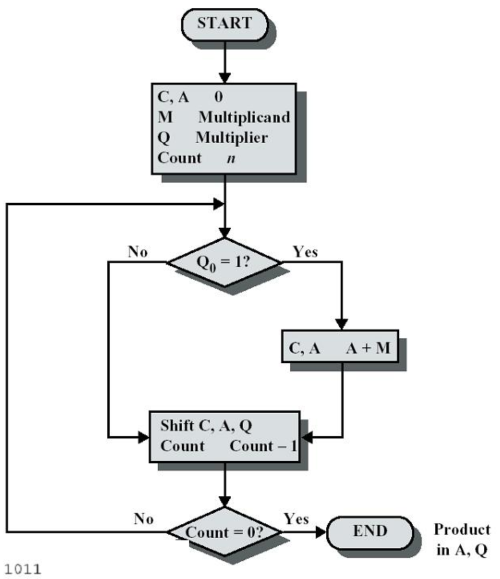
- Overflow (<u>not the carry!</u>)
  - bit that indicates that the data-word cannot contain the result of the addition

Unsigned

0111 0011 1010 1111 1 0010 0010

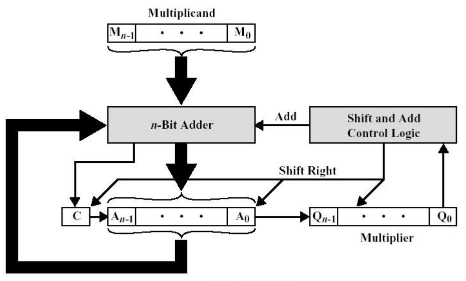
0111 0011 1010 1111 1 0010 0010

Signed



```
    1011
   ×1101
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
10001111
```

Figure 8.9 Flowchart for Unsigned Binary Multiplication

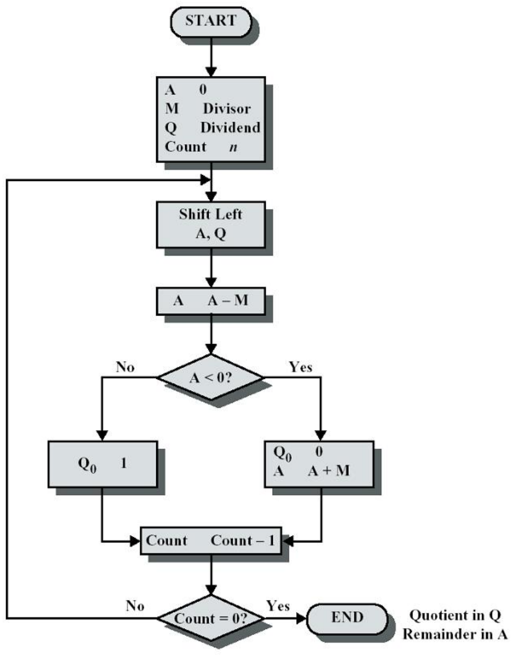


(a) Block Diagram

```
C   A     Q     M
0   0000  1101  1011    Initial Values

0   1011  1101  1011    Add   } First
0   0101  1110  1011    Shift } Cycle

0   0010  1111  1011    Shift } Second
                              } Cycle

0   1101  1111  1011    Add   } Third
0   0110  1111  1011    Shift } Cycle

1   0001  1111  1011    Add   } Fourth
0   1000  1111  1011    Shift } Cycle
```

(b) Example from Figure 8.7 (product in A, Q)

Figure 8.8 Hardware Implementation of Unsigned Binary Multiplication



Division

Multiplication There are two ways to implement the multiplication of two unsigned numbers, one is by repeated addition and the other is via partial sums (works like primary-school arithmetic). Unlike addition and subtraction, the unsigned methods will not work for multiplication involving negative two's complement numbers (Stallings 2000; Ch. 8). They will however work for pairs of positive two's complement numbers. One procedure is:

- check the sign of the result (same sign: result is positive, different sign: result is negative),
- make all numbers positive,
- perform an unsigned multiplication
- negate the result if necessary

Alternatively, Booth's algorithm, a variant of the partial sums method, may be utilised. This method exploits the fact that transitions from 0->1 and 1->0 (reading right to left) within the stream of bits of a two's complement number reflect a subtraction and addition (respectively) of a power of 2.

Division is the opposite of multiplication. In it's simplest form, for unsigned numbers, the divisor may be repeatedly subtracted from the dividend and the quotient incremented each time. Unsigned numbers may also be divided using a partial remainder method. Two's complement numbers may be converted to unsigned form for division (note that in addition to adjusting the quotient sign, the reminder sign needs to be adjusted), or the partial remainder method may be modified for use on two's complement numbers (note that the method involves an element of "back-tracking").

Multi-byte operations Because the ALU is generally structured for single data word operations, the manipulation of multiple-word integer numbers i.e. 16-bit, 24-bit, etc. on an 8-bit ALU requires extra work. Addition and subtraction may be accomplished simply by propagating the carry to the addition of the lowest bits of the next data-word pair. Multi-byte multiplication may be accomplished using a partial sums method, which involves shifting by words instead of by bits before adding. Multi-byte division may be accomplished by starting with the highest order word, shifting the result, and propagating the remainder.

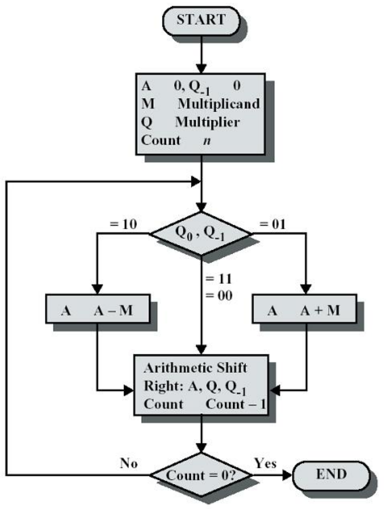
BCD operations Binary coded decimal numbers are specified so that individual digits are spaced just as far apart as their normal representations, in the increasing direction only. It follows that two digits can be added/subtracted as long as we ensure that we have not generated an invalid BCD as a result. If an invalid BCD is detected, it should be corrected, and a carry/borrow set for use by the next pair of digits.

Multiplication/Division of BCD numbers is best done by conversion to binary and then reconversion to BCD on completion. An alternative is to use lookup tables to generate the desired result.

Booth's Theory

$$\underbrace{00111110}_{2^5+2^4+2^3+2^2+2^1 \,\equiv\, 2^6-2^1}$$

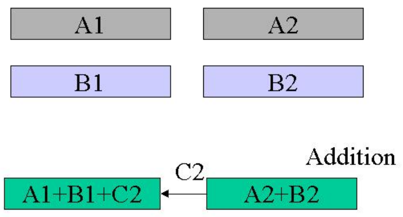
- Read from right to left
- Look for transitions from 0->1 (minus) and 1->0 (add)
- Uses a "phoney" bit -1 (i.e. to the right of bit 0)
- Assumes two's complement signed numbers
  - extend bit width of unsigned numbers if necessary.

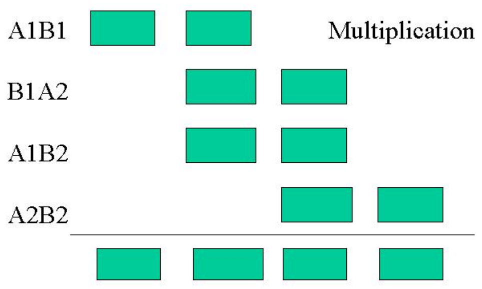


Booths Algorithm

```
1001   0011   0   0111    A    A - M }  First
1100   1001   1   0111    Shift       }  Cycle

1110   0100   1   0111    Shift       }  Second
                                         Cycle

0101   0100   1   0111    A    A + M }  Third
0010   1010   0   0111    Shift       }  Cycle

0001   0101   0   0111    Shift       }  Fourth
                                         Cycle
```

Multi-word operations





BCD translation/carry

- Observations
  - Valid BCD digits have the same binary representation as their binary counterparts.
  - Successive BCD digits represent a factor of 10 increase
    - * 10 = * 2 (l.shift once) + * 8 (l.shift 3 times)
  - An upper BCD digit has an inherent factor of 16 (l.shift 4 times)
  - If a BCD digit is invalid (i.e. $\geq A_{16}$), adding 6_{16} will generate a digit carry.
  - Digit carry's will propagate "naturally" into the next digit.
- Action
  - to translate a packed BCD byte
    - store the bottom-most digit
    - take the top digit (clear out bottom bits)
      - shift it once,
      - add to stored value,
      - shift it twice more
      - add it to the stored value
  - to add two packed BCD bytes
    - add the bytes using integer addition
    - add 66_{16} , carry flags will now be ok.
    - subtract 6_{16} from each nibble, iff no (digit) carry

Tutorial Group page IV-6 Q. 6

To divide an x-bit dividend by a q-bit divisor using Booth's like division

- make an (q+x) bit number for the q-bit remainder and x-bit quotient
- initialise the uppermost q-bits with the sign bit of the dividend, and the lower x-bits with the dividend
- Loop for x times:
  - arithmetic left shift the (q+x) bit number
  - if the sign is the same as(different from) the divisor
    - subtract(add) the divisor from(to) the uppermost q bits
    - if the result sign is different from the original sign; undo
    - if the result sign is the same as the original sign OR the whole (q+x) number is 0; set the LSB of the (q+x) bit number to 1
- Remainder is top q-bits. If the divisor and dividend signs are same (different), the answer is the (complement of) bottom x-bits

Review Exercises

1. The working shows an attempt to perform addition on 8 bit signed numbers, using 4 bit unsigned addition.

$$
\begin{gathered}
-12_{10} + 22_{10} \\
-0001100_2 + 0010110_2 \\
10001100_2 + 00010110_2 \\
10010010_2 \\
-0010010_2 \\
-18_{10}
\end{gathered}
$$

   Identify the flaw (if any) in the method shown:

   (a) arithmetic error in lower nibble ☐
   (b) arithmetic error in upper nibble ☐
   (c) carry was not transferred from lower nibbles to upper nibbles ☐
   (d) incorrect representation ☐
   (e) none of the above ☐

2. The working shows an attempt to perform subtraction using two's complement representation.

$$
\begin{array}{l}
F_{16} - 4_{16} \\
1111_2 - 0100_2 \\
1111_2 + 1100_2 \\
11011_2 \\
-00100_2 \\
-4_{16}
\end{array}
$$

   Identify the flaw (if any) in the method shown:

   (a) arithmetic error ☐
   (b) incorrect extension of the representation ☐
   (c) incorrect interpretation of result ☐
   (d) insufficient bits used to represent the number ☐
   (e) none of the above ☐

3. Using two's complement arithmetic, calculate the result of the following expressions and determine the values of the carry and overflow flags (presuming an 8 bit data-word). Show all working.

   (a) $10_{10} + 16_{10}$
   (b) $7A_{16} - 10_{16}$
   (c) $-82_{10} - 167_{10}$
   (d) $22_{16} - 84_{16}$
   (e) $6_{16} - 2A_{16}$

4. "Many CPU's provide logic for performing arithmetic on packed decimal numbers. Although the rules for decimal arithmetic are similar to those for binary operations, the decimal results may require some corrections to the individual digits if binary logic is used. Consider the decimal addition of two unsigned numbers. If each number consists of N digits, then there are $4N$ bits in each number. The two numbers are to be added using a binary adder.

   - Suggest a simple rule for correcting the result.
   - Perform addition in this fashion on the numbers 1698_{10} and 1786_{10}.

   " (Stallings 2000; 9.1)

5. "The tens complement of the decimal number X is defined to be $10N - X$, where N is the number of digits in the number.

   - Describe the use of tens complement representation to perform decimal subtraction.
   - Illustrate the procedure by subtracting $(0326)_{10}$ from $(0736)_{10}$." (Stallings 2000; 9.2)
   - What purpose does ten's complement arithmetic serve in BCD arithmetic?

6. To perform the multiplication of two 8-bit unsigned integers, we can employ:

   (a) Booth's algorithm
   (b) partial sums
   (c) repeated addition
   (d) all of the above
   (e) none of the above

7. Booths algorithm is carried out on two 4 bit numbers. This will involve:

   (a) 4 arithmetic shifts and 4 or less add/subtract operations
   (b) 4 arithmetic shifts and a total of 8 add/subtract operations
   (c) 4 logical shifts and 4 or less add/subtract operations
   (d) 5 arithmetic shifts and a total of 4 add/subtract operations
   (e) 8 logical shifts and less than 4 add/subtract operations

8. (a) Using repeated addition, determine: $B_{16} * -9_{16}$. Show all working.
   (b) Using partial sums, determine: $F_{16} * -4_{16}$. Show all working.
   (c) Using Booth's algorithm, determine: $F_{16} * -A_{16}$. Show all working.

9. Explain why -7/3 and 7/-3 yield the same quotient but different remainders.

10. Role Play/Challenge:

    (a) Write programs for the PIC16F877 to perform integer multiplication using shifting, and addition operations only!
    (b) Role Play/Challenge: Suggest a method to correctly add and subtract two ASCII digits.

Tutorial Exercise 5A Offline Version[25]

ID# ____________

1. Explain the difference between ”carry” and ”overflow”. *2 marks*

2. Name and explain one alternative to Booth’s algorithm for multiplication of **signed** numbers. *2 marks*

3. Use Booth’s algorithm to determine $8_{10} * -2_{10}$. *6 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student’s work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

[25] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 12 Write the letter you have been assigned here________.

The numbers assigned to each letter are: A: $F_{16}, -3_{16}$ B: $2_{16}, F_{16}$ C: $2_{16}, -3_{16}$ D: $8_{16}, 2_{16}$

Use Booth's algorithm to multiply the numbers provided.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - perform integer arithmetic operations using binary representations of the operands.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

13 Real number arithmetic

At the end of this unit the student will be able to:

> *perform fixed/floating point arithmetic operations using binary representations of the operands.*

Any real number X may be re-expressed as

$$X = X_s \times B^{X_E}$$

where: X_s is the significand or mantissa, B is the base and X_E is the exponent.
The relevant arithmetic operations on such numbers may be written as:

$$\begin{aligned} X \pm Y &= \left(X_s \times B^{X_E - Y_E} \pm Y_s\right) \times B^{Y_E}; \forall X_E \leq Y_E \\ X \times Y &= (X_s \times Y_s) \times B^{X_E + Y_E} \\ X \div Y &= (X_s \div Y_s) \times B^{X_E - Y_E} \end{aligned}$$

Fixed point, same format Fixed point numbers can be interpreted as scaled integers (i.e. the two's complement integer multiplied by some factor; the base $B = 2$, the significand X_s is the two's complement integer and the exponent X_E is a negative integer).
If two fixed point numbers are of the same format, it follows that standard integer addition and subtraction will generate the correct fixed point result. Multiplication and division however will requires correction in order to increase/reduce the factor. This can be accomplished by the use of *arithmetic* shifts[26] *after* performing the standard integer operation.

Fixed point, different formats Fixed point numbers of different formats will always be of the same base $B = 2$, and must be arithmetically shifted to the same format, *prior* to addition/subtraction. Multiplication/division however, may be carried out with the unaltered numbers, and the required shift performed *afterwards* on the result.

Floating point Arithmetic using floating point numbers (assuming the same base) involves manipulating both the exponent and the significand as for fixed point notations. However, because floating point formats differ not only in the number of bits assigned to each component (sign, significand, exponent), but also in the order in which they appear in the representation, in the formats used to represent each component (biased, two's complement, fixed point), and in the assumed base, it is not possible to manipulate floating point numbers using standard integer arithmetic methods. In order to perform floating point arithmetic, the number must first be decomposed into it's components, the components manipulated separately as integer or fixed point, and then the floating point number is re-assembled. As a result, floating point manipulation takes large amounts of code/time.

[26]Arithmetic shifts differ from logical shifts in that the sign bit is maintained.

Fixed/Floating point arithmetic

$\pm S \times B^{\pm E}$; B is known

$$\left.\begin{aligned} X+Y &= \left(X_S \times B^{X_E-Y_E} + Y_S\right)\times B^{Y_E} \\ X-Y &= \left(X_S \times B^{X_E-Y_E} - Y_S\right)\times B^{Y_E} \end{aligned}\right\} X_E \le Y_E$$

$$X \times Y = \left(X_S \times Y_S\right)\times B^{X_E+Y_E}$$

$$\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right)\times B^{X_E-Y_E}$$

Should we use floating point with the PIC?

- VERY LONG programs, which are processor intensive
  - try CCS assembler

```
float c=2.3;
c=2.3;
c=2.3*1.25;
```

- Possible Alternatives
  - fixed point representation (factored arithmetic)
  - lookup tables

See:
http://www.phanderson.com/PIC/16C84/calc_disc_1.html

Review Exercises

1. In order to add two real numbers using floating point representation/arithmetic, we:
   (a) add the exponents, and subtract the significands.
   (b) equate the exponents, and multiply the significands.
   (c) equate the exponents, and then add the significands.
   (d) equate the significands, and add the exponents.
   (e) subtract the exponents, and add the significands.
2. In order to multiply two real numbers using floating point representation/arithmetic, we:
   (a) add the exponents, and multiply the significands.
   (b) equate the exponents, and multiply the significands.
   (c) equate the significands, and add the exponents.
   (d) multiply the exponents, and add the significands.
   (e) multiply the exponents, and then multiply the significands.

3. $10.45 * 10^{12} \times 3.47 * 10^{-7} =$
   (a) $(10.45 \times 3.47) * 10^{12+7}$
   (b) $(\frac{10.45}{3.47}) * 10^{12+7}$
   (c) $(\frac{10.45}{3.47}) * 10^{12-7}$
   (d) $(10.45 \times 3.47) * 10^{12-7}$

4. $\left(\frac{5.2\times8^2}{1.02\times8^{14}}\right)^{-1} =$
   (a) $\frac{5.2}{1.02} \times 10^{2-14}$
   (b) $\frac{1.02}{5.2} \times 8^{2-14}$
   (c) $\frac{5.2}{1.02} \times 8^{14-2}$
   (d) $\frac{1.02}{5.2} \times 8^{14-2}$

5. Show how the following fixed/floating point calculations are performed (where significands are truncated to 4 decimal digits) (Stallings 2000; 8.25–8.27)
   (a) $0.5566 \times 10^3 + 0.7777 \times 10^3$
   (b) $0.3344 \times 10^2 + 0.8877 \times 10^{-1}$
   (c) $0.7744 \times 10^{-2} - 0.6666 \times 10^{-2}$
   (d) $0.8844 \times 10^{-2} - 0.2233 \times 10^0$
   (e) $(0.2255 \times 10^2) \times (0.1234 \times 10^1)$
   (f) $(0.8833 \times 10^3) \div (0.5555 \times 10^5)$

6. Perform the following fixed/floating point calculations using addition, increment, shift, and complement operations(show your working).
   (a) $0.1123_{10} - 1.56_{10}$
   (b) $0.001_2 + 1.01_2$
   (c) $0.101_2 \times 2^{11_2} - 0.11_2 \times 2^{10_2}$
   (d) $0.101_2 \times 2^{11_2} * 0.11_2 \times 2^{10_2}$

7. For a fractional number F, the relative error of its representation F' is expressed as $\frac{F-F'}{F}$. If the fractional numbers $A = 0.236$ and $B = 0.135$ are represented by the computer as truncated numbers $A' = 0.23$ and $B' = 0.13$; What is the relative error between the calculated value $C = A - B$ and it's represented value C' ?

8. Role Play/Challenge: With regard for the ease of performing calculations and manipulations using the representation, which representation of real numbers is to be preferred: fixed or floating point?Explain your answer, using sample code for the PIC16F877.

9. Role Play/Challenge: With regard for the accuracy/precision with which a non-integer number can be represented, which representation of real numbers is to be preferred: fixed or floating point?Explain your answer, using sample code for the PIC16F877.

10. Role Play/Challenge: Write an assembly language program for the PIC16F877, to perform floating point addition of a number with a sign bit, and 2 bit exponent.

Tutorial Exercise 5B Offline Version[27]

ID#

Write an assembly-language routine for the PIC16F877, which will multiply two 8-bit fixed point numbers, where the point is located between bits 6 and 5.

- You may call any of the "standard" code snippets for multiplication of integers – there is no need to re-write them.
- The routine should return the answer as a fixed point number with the point located between bits 6 and 5.
- On overflow the routine should return 0, with the Carry flag set.
- Your answer should show how your routine would work with the numbers 01.101001_2 and 01.001000_2. *10 marks*

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

[27] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 13 Write the letter you have been assigned here________.
The values and calculations are: Q: 536.5 Z: 23.875 A:(Q*Z) B:(Q/Z) C:(Q+Z) D:(Q-Z)
What steps would you have to go through in order to perform this calculation:

1. using a floating point representation of the numbers.

2. using a fixed point binary representation of the numbers.

Challenge: can you write a PIC16F877 routine to do so?

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - perform fixed/floating point arithmetic operations using binary representations of the operands.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

14 Data structures

At the end of this unit the student will be able to:

> *describe how data (and data structures) can be represented and manipulated, with reference to alignment and byte/bit ordering issues.*

Multiple data-words may be used to represent a single number (e.g. 8 bit machine may have 16 bit integers and 32 bit floating point) or a single data-word may represent multiple digits (e.g. packed BCD). It follows that the ordering of bits and bytes may affect the interpretation of a particular data structure.

The first issue is bit numbering: Is the lowest numbered bit presumed to be bit 0 or bit 1? Are the bits numbered from left to right or right to left? There are four possible interpretations: 76543210, 01234567, 12345678, 87654321. We have been using the first by default (as bit n represents 2^n within the unsigned binary number), however the others do occur.

The second is bit ordering: Is the least significant bit assigned to the lowest numbered bit or the highest numbered bit? For example, do we represent the sequence of bits for the number 0xFFA2 as 1111 1111 1010 0010 or 0100 0101 1111 1111. We have been using the first by default, but the other is also valid.

A related issue is byte ordering. Both of the solutions presented above, presume that the byte and bit orders are the same i.e. if the least significant bit appears first, the least significant byte appears first. This is also not necessarily the case, especially where structures span multiple data words.

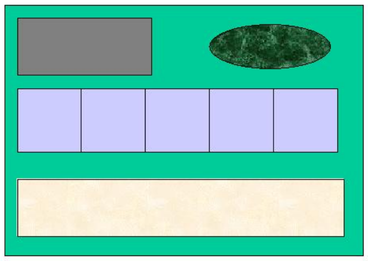
The issues of ordering of bits in the byte, bytes in a data structure, and data structures in arrays or compound structures are referred to as ”endian” issues.

Finally, the issue of alignment concerns cases where the data structure to be stored is smaller than the word size (e.g. a digit in Packed BCD). In order to facilitate the efficient retrieval of such data from memory, it is stored so that data ”chunks” start on word boundaries. One example is in storage of video data. Single lines of pixel data are stored, such that each new line starts with a new data word. Any space at the end of the last data word is left unused.

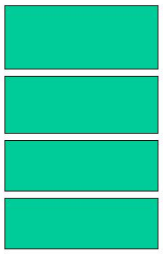
Why are these issues important?

- Serial transfer data between computers which have different endian and alignment conventions, will result in data being read differently
- Reinterpretation of data (as a different type) will result in a different values being obtained if different endian and alignment conventions are observed.
- Indexing into a block of memory (or lookup table) requires that the programmer know not the inherent size of the structure, but the actual size produced after observing alignment.

Records (Struct, Class)



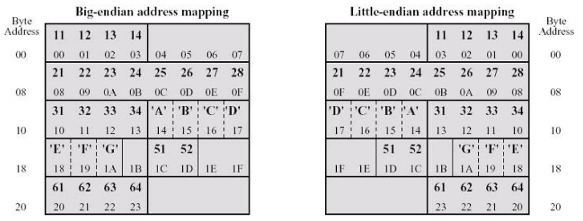
Array/Table



Memory Wars

- Bit Numbering
- Bit Ordering
- End-ian-ness
- Data Alignment

```
struct{
   int     a;     //0x1112_1314                word
   int     pad;   //
   double  b;     //0x2122_2324_2526_2728      doubleword
   char*   c;     //0x3132_3334                word
   char    d[7];  //'A'.'B','C','D','E','F','G'  byte array
   short   e;     //0x5152                     halfword
   int     f;     //0x6161_6364                word
} s;
```



Big-endian address mapping

Byte Address								
	11	12	13	14				
00	00	01	02	03	04	05	06	07
	21	22	23	24	25	26	27	28
08	08	09	0A	0B	0C	0D	0E	0F
	31	32	33	34	'A'	'B'	'C'	'D'
10	10	11	12	13	14	15	16	17
	'E'	'F'	'G'		51	52		
18	18	19	1A	1B	1C	1D	1E	1F
	61	62	63	64				
20	20	21	22	23				

Little-endian address mapping

								Byte Address
				11	12	13	14	
07	06	05	04	03	02	01	00	00
21	22	23	24	25	26	27	28	
0F	0E	0D	0C	0B	0A	09	08	08
'D'	'C'	'B'	'A'	31	32	33	34	
17	16	15	14	13	12	11	10	10
		51	52		'G'	'F'	'E'	
1F	1E	1D	1C	1B	1A	19	18	18
				61	62	63	64	
				23	22	21	20	20

Review Exercises

1. For the following data structures, draw the big-endian and little-endian layouts, using the format of Figure 9.18 (see handout) and comment on the results. (Stallings 2000; 9.17)

   (a)
   ```
   struct {
       double i; // 0x1112131415161718
   } s1;
   ```
   (b)
   ```
   struct {
       int i; // 0x11121314
       int j; // 0x15161718
   } s2;
   ```
   (c)
   ```
   struct {
       short i; // 0x1112
       short j; // 0x1314
       short k; // 0x1516
       short l; // 0x1718
   } s3;
   ```

2. Based on the above, write a small C program which determines the endianness of a machine and reports the results. (Stallings 2000; 9.19)

3. It is possible to store a data structure which is either wider or narrower than the data word size, by simply storing the bits of the data structure consecutively. What are the implications for retrieving data from memory?

4. Boundary alignment is the practice of starting new blocks of data in a new data-word block, regardless of the space remaining in the previous data-word block. What possible advantage does such a scheme offer?

5. A particular data structure contains an 10 bit unsigned integer, two 7 bit characters, and two status flags. Draw a diagram to show how the 26-bit data structure can be stored in consecutive 14-bit data-words using:

   (a) packing with boundary alignment (i.e. 2 data words per structure)

   (b) packing with no boundary alignment (i.e. overlapping datawords)

   (c) individual program words for each structure element (i.e. 5 data words per structure)

6. A certain C-struct was defined as containing a character, an integer array of size 3, and a precision integer number, in that order. The structure is made part of a program, which is compiled to run on a machine. The machine in question:

   - aligns it's data on 4-byte boundaries,
   - stores the most significant byte at the least numbered address, and
   - stores the most significant bit in a byte at bit position 7.

   Presuming that:

   - a character requires 1 byte of storage space,
   - an integer requires 2 bytes of storage space,
   - a precision (long) requires 4 bytes of storage space;

   Draw a memory layout showing how the record {'A'(0x41), {10(0x000A),15(0x000F),299(0x012B)}, 1250(0x000004E2)} when stored at location 0x20 will appear.

7. Draw a diagram to illustrate how **you** would choose to store a set of 3-bit fields on the PIC16F877. You should justify your choice by making reference to the amount of fields stored, and ease of storage/retrieval.

8. (a) Draw a diagram to illustrate how the order of bits in a byte can be reversed in the PIC16F877 using not more than two file registers, and the rotate operations.

   (b) Write a program for the PIC16F877, which performs the algorithm you illustrated in your diagram for reversing the order of bits in a byte.

9. Role Play: Describe the operation of the four different methods of implementing lookup tables in the PIC16F877

   (a) PCL manipulation with DT/retlw (no bank crossing)

   (b) PCL manipulation with DT/retlw (bank safe)

   (c) data EEPROM

   (d) program EEPROM

10. Role Play/Challenge: How would you implement record storage and retrieval on the PIC16F877?

Tutorial Exercise 6A Offline Version[28]

ID#

One hundred 3-bit fields need to be stored in Bank 0 of the register file in the PIC16F877.

1. Which is the most appropriate storage method? *1 mark*
   - (a) each group of 2 fields stored in a file register ☐
   - (b) each group of 8 fields stored in 3 consecutive file registers ☐
   - (c) each field stored in it's own file register ☐
2. Justify the choice you made in Question 1 *3 marks*
3. For your chosen storage method (Question 1), write a routine to retrieve the n^{th} 3-bit field where n is between 0 and 99. n is value in the working register when the routine is called. The answer should be placed in bits 2-0 of the working register when the routine returns. Your answer should show how your routine will work for n=57. *6 marks*

[28] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 14 Write the letter you have been assigned here________.
The concepts are: A: bit numbering B: bit ordering C: byte ordering D: alignment.

1. Write a sentence which explains this concept.

2. A C structure contains a character, and an integer, in that order. The structure is part of a program. When compiled:

   - records are aligned on 2-byte boundaries,
   - the most significant byte is stored at the least numbered address, and
   - the least significant bit in a byte is stored at bit position 0
   - a character requires 1 byte, and an integer requires 2 bytes of storage space

   Show how the record 'A'(0x41), 410 is stored starting at address 0x10. Use X to indicate the unused bits.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0x10:								
0x11:								
0x12:								
0x13:								
0x14:								

3. Use another color pen/pencil to amend the table, thus illustrating your concept.

Unit 14

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - describe how data (and data structures) can be represented and manipulated, with reference to alignment and byte/bit ordering issues.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

Numbers & Data

This section has partially addressed the second learning objective for this course i.e. you should now be able to " **illustrate how data can be represented in (and accessed from) memory**". It should be clear at this stage that there are no "hard-and-fast" rules to representation, however you need to be aware of what the representation is, in order to make any sense of the data. Further the rules for manipulation (arithmetic in particular) are dependent upon the chosen representation, and finally that the job of manipulating data can be made significantly easier or more difficult by the choice of representation.

In the next section we will look at code for implementing integer arithmetic and common algorithms/ data handling techniques on the PIC16 series of microcontroller.

original author: Feisal Mohammed; updated: March 11, 2008– CLR

Part V
Algorithms on PIC16

So far we have been discussing assembly language programming on microprocessors in terms of generic instructions. This section looks at implementing these manipulations on the PIC16 series of microcontrollers. It contains sample code for basic operations which have already been discussed, how pieces of code can be compared, and the limitations of compiler generated code.

15 Basic operations

At the end of this unit the student will be able to:

implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.

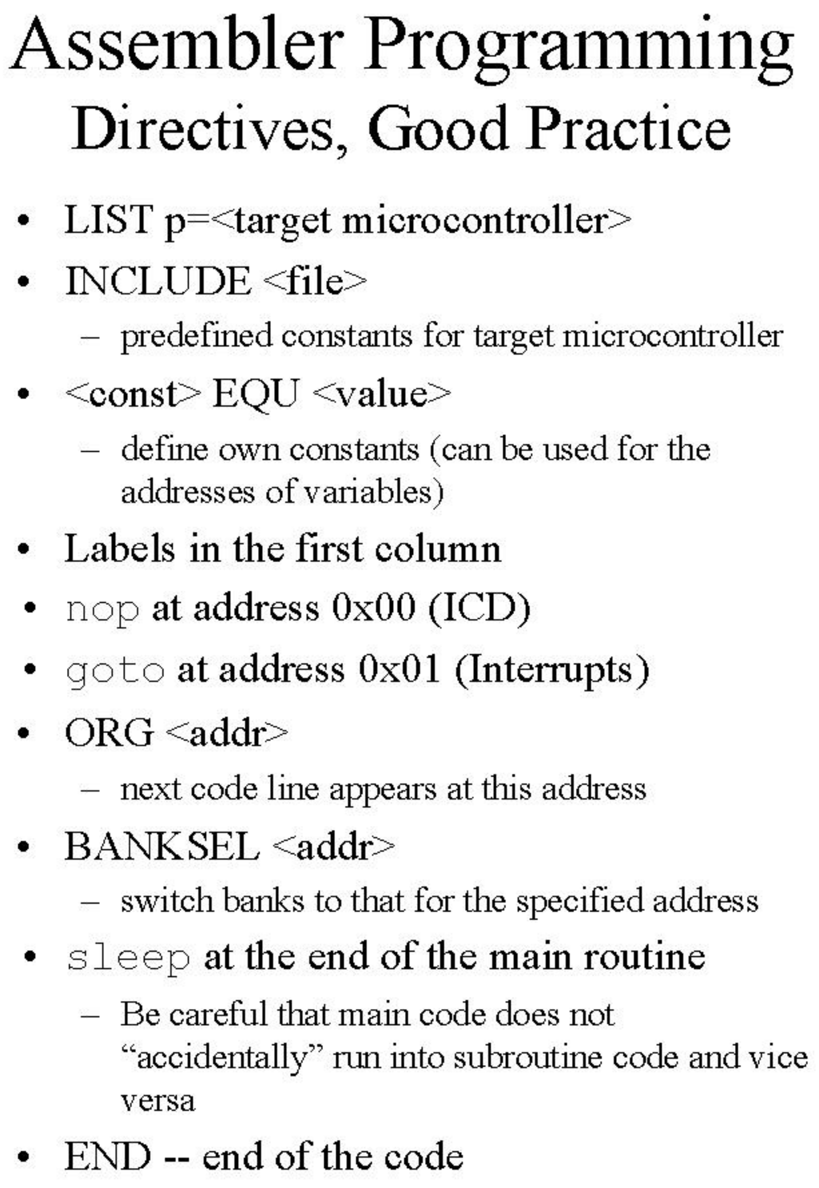
The code presented in this section has been gathered from a multitude of sources. It is important to recognise that they are "snippets". They will not work by themselves; they are intended to be used in a program, in which variables have been appropriately declared, and they have been appropriately placed in the program banks. Thus you should exercise caution in trying to "cut-and-paste" code. A more appropriate use of these snippets, would be to appreciate the underlying operations, so that you can reproduce, or improve on, the algorithms in your own code.

Standard Operations

Indirect addressing The following bit of code[29] shows how to clear a section of RAM from 20h to 2Fh using indirect addressing.

```
     movlw 0x20    ; put the starting address into W
     movwf FSR     ; ...then into the FSR
Next clrf  INDF    ; clear the address pointed to by the FSR
     incf  FSR,F   ; point to the next address
     btfss FSR,4   ; does the FSR now contain 30h? (is bit 4 set?)
     goto  Next    ; not set, repeat
done  ......       ; execution jumps to here when bit 4 is set
```

[29]Please note that the code presented in this section uses two features of the PIC assembler which are not actually translated into opcodes. The first is the use of labels to represent the addresses of the following instruction (e.g. `Next` and `done`). The second is the use of labels for the Special Function Registers (e.g. `FSR`, `INDF`, `F`). These are pre-defined in the include file for the microcontroller using the `EQU` preprocessor keyword.



Assembler Programming
Directives, Good Practice

- LIST p=<target microcontroller>
- INCLUDE <file>
  - predefined constants for target microcontroller
- <const> EQU <value>
  - define own constants (can be used for the addresses of variables)
- Labels in the first column
- `nop` at address 0x00 (ICD)
- `goto` at address 0x01 (Interrupts)
- ORG <addr>
  - next code line appears at this address
- BANKSEL <addr>
  - switch banks to that for the specified address
- `sleep` at the end of the main routine
  - Be careful that main code does not "accidentally" run into subroutine code and vice versa
- END -- end of the code

Swapping data The case where data must be swapped between W and a File register seems simple enough, just use a few `mov` instructions and a temporary RAM location. However upon examining the instructions, it can be seen that both the `movwf` and the `movf` instructions would require the use of the W register. That is there is no instruction that would move data between two File registers. Therefore, coding the swap routine requires *two* temporary locations and six instructions.

```
movwf   tempA
movf    data, W
movwf   tempB
movf    tempA, W
movwf   data
movf    tempB, W
```

An alternative routine swaps the working register, W, with a file register, `data`, it uses the fact that two exclusive ORs of the same value cancel out.

```
xorwf   data,W
xorwf   data,F
xorwf   data,W
```

Similarly, to swap data between two file registers where the working register contents are not needed, the direct method requires a temporary register and 6 instructions.

```
movf    dataX, W
movwf   temp
movf    dataY, W
movwf   dataX
movf    temp, W
movwf   dataY
```

This can be reduced to 4 instructions, and no temporary location as follows:

```
movf    dataX, W
subwf   dataY, W
addwf   dataX, F
subwf   dataY, F
```

Subtraction The subtract instructions (`subwf` and `sublw`) work differently than most people expect. `subwf` subtracts W *from* the contents of the register, and `sublw` subtracts W *from* the literal. Additionally the Carry flag behaves as $\overline{\text{Borrow}}$ flag when subtraction occurs i.e. if a borrow occurs the Carry flag is *not* set and vice-versa.

If you want to subtract a literal from W, it is easiest to use the `addlw` instruction with the two 's complement of the literal. For example:

```
addlw   0feh    ; W := W - 2
```

The MicroChip assembler allows this to be written as:

```
      addlw   -2
```

There is no instruction to take the two's complement of W (like the `neg` instruction on Motorola processors), but because of the way the subtract instructions work you can use:

```
      sublw   0
```

Comparison operations Comparing two numbers need to be treated with care because of the non-intuitive nature of the Carry flag.

```
        movlw   2           ; W = 2
        subwf   Q,W         ; W = Q - 2
        btfss   STATUS,C
        goto    Less        ; Q < 2
        goto    Gr_eq       ; Q >= 2
```

If `Q` is less than 2, there will be a borrow therefore the Carry flag will *not* be set. The implementation of the subtraction is done by forming the two's complement of the subtrahend and then adding. If `Q` is 1 then the operation looks like the following

```
  0000 0001     ; Q = 1
  1111 1110     ; two's complement of W (= 2)
  ---------
0 1111 1111 ^---------------; Carry not set, i.e. borrow
```

If however `Q` is 3 then

```
  0000 0011     ; Q = 3
  1111 1110     ; two's complement of W (= 2)
  ---------
1 0000 0001 ^---------------; Carry set, i.e. no borrow
```

If now it is required to branch to one spot for `Q` less than or equal to W, then the subtraction will have to be turned around, for example

```
        movf    Q,W         ; W = Q
        sublw   2           ; W = 2 - W (Q)
        btfss   STATUS,C
        goto    Great       ; Q > 2
        goto    Less_eq     ; Q <= 2
```

When `Q` is equal to 3, then the operation looks like this

```
  0000 0010     ; 2
  1111 1101     ; two's complement of W (= 3)
  ---------
0 1111 1111 ^---------------; Carry not set, i.e. borrow
```

and if `Q` is equal to 1

```
  0000 0010     ;
  1111 1111     ; two's complement of W (= 1)
  --------
1 0000 0001 ^---------------; Carry set, i.e. no borrow
```

The above examples illustrate that care must be taken when using the subtraction routines. It is better to work out what actually happens than make assumptions based on other CPUs.

Comparison operations form the basis of if-then-else structures and loop termination tests.

Bit masking It is not generally necessary to perform single-bit masking operations on the PIC16 series of microcontrollers, because of the presence of the bit-test-skip instructions. If however you need to match a multiple-bit pattern, then you will need to make and use a bit mask.

```
chkallset   andwf   MASK,W  ; to extract just the bits we want
            xorwf   MASK,W  ; to test if all masked bits are set
            btfss   STATUS,Z;       == NOT (any masked bits are clear)
            retlw   0       ; the bits were not all set
            retlw   1       ; the bits were all set

chkallclr   andwf   MASK,W  ; to test if all masked bits are clear
            btfss   STATUS,Z;       == NOT (any masked bits were set)
            retlw   0       ; the bits were not all clear
            retlw   1       ; the bits were all clear

setbits     iorwf   MASK,W  ;to set all masked bits in W
            return

clearbits   andwf   NMASK,W ;to clear all masked bits in W
            return

togglebits  xorwf   MASK,W  ;to toggle all masked bits in W
            return

setupmask   movwf   MASK    ;to change MASK to the contents of W, and work out NMASK
            comf    MASK,W
            movwf   NMASK
            movf    MASK,W
            return
```

Multi-word operations

Extending bit width 8bit − > 16bit Extension consists of copying the relevant bit into all the positions of the new upper byte.

```
E8to16  movlw 255       ; make the accumulator 1111 1111
        clrf Out_hi     ; clear out the upper result byte
        btfsc In, 7     ; if the top bit of the input byte is 1
        movwf Out_hi    ; put all ones in the upper result byte
        movf In,W       ; copy the input byte to
        movwf Out_lo    ; the lower result byte
        return
```

Reducing bit width 16bit − > 8bit Reduction should only occur if the bits of the upper byte all match the uppermost bit of the lower byte.

```
R16to8  movlw 255       ; make a byte with the
        btfss In_lo,7   ; inverted upper bit of the lower byte
        clrw
        xorwf In_hi,W   ; xor with the upper byte to check if all bits are the same
        btfss STATUS,Z  ; return an error (non-zero W) if not the same
        retlw 255       ;
        movf  In_lo,W   ; otherwise truncate and return
        movwf Out
        retlw 0
```

Decrementing a 16-bit variable Given a 16-bit variable with its low byte in `countl` and high byte in `counth`.

```
Decr   movf     countl,F   ; set the Zero flag if countl is zero
       btfsc    STATUS,Z   ; countl zero? decrement counth (next instr)
       decf     counth,F
       decf     countl,F   ; always decrement countl
```

Test a 16-bit variable for zero Using the same 16-bit variable as before, test it for zero.

```
Iszero movf     countl,F   ; test for zero in low byte
       btfsc    STATUS,Z   ; if not, don't test high byte
       movf     counth,F   ; test high byte
       btfsc    STATUS,Z
       goto     Bothzero   ; Zero? branch to some appropriate place
       ...
```

16-bit Addition A simple work around for 16-bit numbers is to simply increment one of the numbers if the Carry is set. For example:

```
; This routine adds the 16-bit number in datalo and datahi, to the
; 16-bit number is resultlo and resulthi, leaving the answer in ;
resultlo and resulthi
 Add16  movf    datalo,W     ; W = datalo
        addwf   resultlo,F   ; resultlo = resultlo + W
        movf    datahi,W     ; W = datahi

        btfsc   STATUS,C     ; If no carry skip
        incf    resulthi,F   ; If carry, add 1 to result
        addwf   resulthi,F   ; resulthi = resulthi + W
```

There is a problem with this method because the Carry bit does not accurately reflect the results of the last addition. In particular, if `resulthi` was 0xFF and the Carry bit was set from the low bytes then after the last addition the Carry bit would be clear and it should be set. This means that this method cannot be scaled up to be used with 24-bit and larger numbers.

A slightly longer, but scalable routine that correctly sets the Carry bit on termination is shown below. You should work out the sequence of operations to ensure that you understand the routine.

```
Addgood  movf   datalo,W     ; W = datalo
         addwf  resultlo,F   ; resultlo = resultlo + W
         movf   datahi,W     ; W = datahi

         btfsc  STATUS,C     ; If no carry ...
         incfsz resulthi,F   ; if resulthi is 0 after the increment
                             ; ... don't do the addition
         addwf  resulthi,W
         movwf  resulthi     ; resulthi = resulthi + W
```

16-bit Subtraction Subtraction is very similar to addition, with the inclusion of the borrow. The only difference to look out for is that the PIC16xxx treats the carry bit as a $\overline{\text{Borrow}}$. Assume that the operation required is R = Q_1 - Q_2, where Q_1 and Q_2 are two 16-bit numbers. This can be accomplished by the following code:

```
Subr    movf    Q2_lo,W
        subwf   Q1_lo,W     ; W = Q1_lo - Q2_lo
        movwf   R_lo
        btfss   STATUS,C    ; need to borrow?
        decf    Q1_hi,F     ; get it from hi byte
        movf    Q2_hi,W
        subwf   Q1_hi,W     ; W = Q1_hi - Q2_hi
        movwf   R_hi
        return
```

Note that the carry flag at the end of the routine will not be set correctly if Q1_hi is 0. An improved routine is as follows

```
Subrgd  movf    Q2_lo,W
        subwf   Q1_lo,W     ; W = Q1_lo - Q2_lo
        movwf   R_lo
        movf    Q2_hi,W
        btfss   STATUS,C    ; need to borrow?
        goto    Borrow

        subwf   Q1_hi,W     ; W = Q1_hi - Q2_hi
        movwf   R_hi
        goto    Done

Borrow  subwf   Q1_hi,W     ; W = Q1_hi - Q2_hi
        movwf   R_hi
        decf    R_hi,F      ; do the decrement after
Done    return
```

Integer Multiplication and Division

8-bit by 8-bit Unsigned Multiplication There are two ways to implement the multiplication of two 8-bit numbers, one is by repeated addition and the other is via partial sums. The first example multiplies M1 by M2 leaving the result in R_hi and R_lo. It loops through the addition of M2 to itself M1 times.

```
    ; 8x8 unsigned multiply routine.
    ; No checks made for M1 or M2 equal to zero

        clrf    R_hi      ; clear result location
        clrf    R_lo
        clrw

M8x8    addwf   M2,W      ; add M2 to itself
        btfsc   STATUS,C  ; if carry set
        incf    R_hi,F    ; ... increment high byte
        decfsz  M1,F
        goto    M8x8
        movwf   R_lo
        ...
```

This second routine, does the same thing but loops only 8 times instead of M1 times as the previous case. M1 is added to the result which is then shifted. It makes use of the fact that multiplying a number by one results in the same number so all that is required is either addition if the bit, in M2, was one or no addition if it was zero.

```
    ; 8x8 unsigned multiply routine.
    ; No checks made for M1 or M2 equal to zero

        clrf    R_hi      ; Clear result location
        clrf    R_lo

        movlw   0x08      ; setup loop count
        movwf   cntr
        movf    M1,W      ; initialize W with M1

Umul    rrf     M2,F      ; rotate into carry to check bits
        btfsc   STATUS,C  ; if carry set i.e. bit was 1
        addwf   R_hi,F    ; ... we add
        rrf     R_hi,F    ; shift over for the next addition
        rrf     R_lo,F
        decf    cntr,F    ; check the loop count
        btfss   STATUS,Z
        goto    Umul
        ...
```

16-bit by 8-bit Unsigned Division

```
    ; 16-bit unsigned division. Quot = (Div_hi, Div_lo) / Divsor

        clrf    Quot

Divide  movf    Divsor,W    ; setup for subtraction
        subwf   Div_lo,F    ; Div_lo = Div_lo - Divisor
        btfss   STATUS,C
        goto    Borrow
        goto    Div_2

Borrow  movlw   0x01
        subwf   Div_hi,F    ; use subtract instead of decf
        btfss   STATUS,C    ; ... because it sets the carry
        goto    Done        ; generated a borrow so finish

Div_2   incf    Quot,F      ; add one and loop again
        goto    Divide
Done    ....
```

Booth's algorithm: 8-bit by 8-bit signed numbers It is presumed that the result will be 16 bits wide, and that the arguments are valid.

```
    ; we wish to multiply M*Q. Result will appear in A:Q

bthloop movf    Q,W
        andlw   0x01
        xorwf   STATUS,F    ; check the pair of bits
        btfss   STATUS,C
        goto    arshft
        movlw   M

        btfsc   Q,0         ; if the Q_0 bit is 1 then subtract
sub8    subwf   A,F

        btfss   Q,0
add8    addwf   A,F

arshft  bcf     STATUS,C
        btfsc   A,7
        bsf     STATUS,C
        rrf     A,F
        rrf     Q,F
        decfsz  count,F     ; check if we are done
        goto    bthloop
        ....
```

BCD operations

Packed BCD Validity checking Adding 6 to a 4-bit nibble produces a carry out of that nibble if the nibble is an invalid BCD code. The following returns 1 in the working register if either code in `Reg` are invalid.

```
PBCDchk movlw   0x66
        addwf   Reg,W
        btfsc   STATUS,C
        retlw   1
        btfsc   STATUS,DC
        retlw   1
        retlw   0
```

Packed BCD Single Digit extraction To extract a single digit from a BCD number, a bit mask may be used:

```
PBCDext movlw 0xF0
        andwf In,W
        movwf Nibble_hi
        movlw 0x0F
        andwf In,W
        movwf Nibble_lo
        return
```

Packed BCD Addition The addition of BCD numbers involves consideration of the decimal carry. The addition plus 6 to each nibble will produce a carry exactly when the decimal addition of two digits in the operands would produce a carry, and these are the carries we are concerned with. But the digits in positions that did not produce a carry will contain an excess 6. The problem, then, is to remove the extra 6 from each digit that did not generate a carry. The following routine leaves the carry set if the upper digit overflows:

```
PBCDadd movlw   0x66
        addwf   RegA,W
        addwf   RegB,W
        rrf     Out,F
        btfss   STATUS,DC
        addlw   -0x06
        btfss   Out,7
        addlw   -0x60
        rlf     Out,F
        movwf   Out
        return
```

– From: Scott Dattalo (http://www.piclist.com/techref/microchip/math/add/bcdp2.htm)

Packed BCD Subtraction Similarly to two's complement arithmetic for binary number, subtraction of decimal numbers can be performed by adding the ten's complement (subtract from all nines and add 1) e.g.

$$\begin{array}{rc} 8_{10} - 2_{10} & 8_{10} - 16_{10} \\ 08_{10} + 98_{10} & 08_{10} + 84_{10} \\ 1\ \ 06_{10} & 92_{10} \\ 6_{10} & -8_{10} \end{array}$$

Note that 0 to 99 represents the numbers +49 to -50 in ten's complement notation. The following code produces the ten's complement of a packed BCD number:

```
PBCDCmp movlw 0x99   ;
        movwf Out    ;
        movf  In, W  ;
        subwf Out,F  ; Out=0x99-In
        incf  Out,F
        return
```

Conversion Packed BCD to unsigned binary To convert from BCD to an unsigned binary, multiply the upper nibble by ten and add the lower nibble. This can be done by shifting the upper nibble right 3 places (multiply by 2), adding the upper nibble shifted 1 place (multiply by 8) and adding the lower nibble.

```
PBCDbin rrf   In,W
        andlw 0x78 ; 0111 1000
        movwf Out
        bcf   STATUS,C
        rrf   Out,F
        rrf   Out,F
        addwf Out,F
        movlw In
        andlw 0x0F
        addwf Out,F
        return
```

Conversion unsigned binary to Packed BCD To convert binary to BCD, check that the 7th bit is not used, and set the carry if the number exceeds 99(i.e. between 99 and 128).

```
BinPBCD btfss In,7  ; check if > 128
        retlw 1     ;   return an error
        clrf  Out
        movlw 156   ; Check if the number will overflow
        addwf In,W  ;
        rrf   Out,F ; Store the Carry in Out
        movlw 100   ; subtract 100 if needed
        btfss Out,7 ;
        subwf In,W  ;
        iorwf Out,F ; Store the reduced number

        swapf Out,W ; sort out the bottom bits
        addwf Out,W
        andlw 0x0F
        btfsc STATUS,DC ; correct for the DC
        addlw 0x10
        btfsc Out,4 ; correct for bit 4
        addlw 0x06
        btfsc Out,7 ; correct for the Carry stored in bit 8
        sublw 0x08
        addlw 6 ; make sure lower nibble is 0-9
        btfss STATUS,DC
        addlw -6

        btfsc Out,6 ; now the upper bits
        addlw 0x60  ; If the 64 bit is on store BCD 60
        btfsc Out,5
        addlw 0x30  ; If the 32 bit is on add BCD 30
        btfsc Out,4
        addlw 0x10  ; If the 16 bit is on add BCD 10

        rlf Out,F
        movwf Out  ; restore the carry, set Out and return
        return
```

Data Storage

MPASM recognises the following formats for numbers/ASCII characters:

Type	Syntax	Example
Decimal	D'<digits>'	D'100'
Hexadecimal	H'<hex_digits>' , 0x<hex_digits>	H'9f', 0x9f
Octal	O'<octal_digits>'	O'777'
Binary	B'<binary_digits>'	B'00111001'
ASCII	'<character>', A'<character>'	'C', A'C'
String	"<character_string>"	"Cat"

Any of the above formats will automatically be translated into the appropriate binary code by the assembler.

Data values may be stored as part of the program i.e. explicitly loaded. Where it is necessary to assume a sequence of values, a lookup table may be implemented using a sub-routine with multiple return values. On some occasions, it is necessary to retrieve constant values from either the program or data ROM's.

Table Lookup When there is a write to the PCL, for example in a table lookup, the PC is moved to point to a particular place in memory based on some input parameter. In the following section of code, the routine returns different binary numbers based on the value of the `W` register. `W` can range from 1 to 4. The routine shown will increment the program counter with the value in W, thus jumping to the relevant number of locations forward. The next instruction will return the 8-bit value (`retlw` has room for an 8-bit operand) at that position of the table.

```
table  addwf PCL, F      ; add W to the PCL and store it in the PCL
       retlw B'00000001' ; return 1 in W if W = 0
       retlw B'00000011' ; return 3 in W if W = 1
       retlw B'00000110' ; return 6 in W if W = 2
       retlw B'00000010' ; return 2 in W if W = 3
       end
```

Please note, that this example presumes that no program bank switching is required. Since the PCL is only 8-bits wide the table has a restriction in that it cannot cross a 256 word boundary. The PCL and PCLATH are another pair of Special Function registers that are also assigned multiple addresses so that access can occur regardless of the value of the RP0, RP1 bits.

A similar example follows where the data to be retrieved is an ASCII string.

```
BatLkup addwf PCL,F
        retlw A‘B’
        retlw A‘a’
        retlw A‘t’
        retlw 0x00
        retlw 0x04
        retlw 0x06
```

The DT directive may be used as a shorthand. For this directive, each value is prefixed with `retlw` and stored in the program memory. Strings are stored as a sequence of ASCII character codes.

The above example could be re-written as:

```
BatLkup addwf PCL,F
        DT  "Bat",0x00,0x04,0x06
```

This method is inefficient because, the upper 6 bits of program memory go unused in each entry, and an extra location is required for the addition instruction for the table.

Lookup Table data values may be stored into any location of the program memory using the DA directive. Because the program memory is 14 bits wide, this directive will store ASCII (7 bits per character) strings with characters packed two-to-a-word. Numeric data will be stored one value to a word. For example:

```
ORG 0x20 DA "abcdef",0xFFFF,0x80
```

will put the following values in the program memory:

Address	Value
0x20	30E2
0x21	31E4
0x23	32E6
0x24	3380
0x25	3FFF
0x26	0080

where the ASCII codes for ’a’ thru ’f’ are 0x61 thru 0x66, and the data value 0xFFFF is truncated to fit 14 bits.

To read a single data word from the program memory, an indirect read must be performed (note that this feature while available on the PIC is not normally available on Harvard architectures). From the data sheet (PIC 2001)

> The steps to reading the FLASH program memory are:
>
> 1. Write the address to EEADRH:EEADR. Make sure that the address is not larger than the memory size of the PIC16F87X device.
> 2. Set the EEPGD bit to point to FLASH program memory.
> 3. Set the RD bit to start the read operation.
> 4. Execute two NOP instructions to allow the microcontroller to read out of program memory.
> 5. Read the data from the EEDATH:EEDATA registers.

```
BSF STATUS, RP1     ;
BCF STATUS, RP0     ;Bank 2
MOVF ADDRL, W       ;Write the
MOVWF EEADR         ;address bytes
MOVF ADDRH,W        ;for the desired
MOVWF EEADRH        ;address to read
BSF STATUS, RP0     ;Bank 3
BSF EECON1, EEPGD   ;Point to Program memory
BSF EECON1, RD      ;Start read operation
NOP                 ;Required two NOPs
NOP                 ;
BCF STATUS, RP0     ;Bank 2
MOVF EEDATA, W      ;DATAL = EEDATA
MOVWF DATAL         ;
MOVF EEDATH,W       ;DATAH = EEDATH
MOVWF DATAH         ;
```

The indirect read of the EEPROM method for storing tables, while more efficient on storage space, takes longer to access and requires more overhead code to read. It is best used for particularly large tables/data values.

Similar methods may be used to store and read data from the Data EEPROM. The directive in this case is `DE` which stores each character or value in a single byte. When using the DE directive be sure to initialise the address to 0x2100 (programming address of the Data EEPROM).

Review Exercises

1. What does the following code-fragment for the PIC 16F877 do:

```
swapf   Reg,W
andlw   0x0F
movwf   Reg
```

   (a) swap the higher and lower nibbles of `Reg`
   (b) perform a signed division `Reg` by 8
   (c) logical right shift `Reg` 4 bit-places
   (d) rotate `Reg` 4 bit-places

2. Label the statements as True or False, in relation to the following code for the PIC16F877:

```
Lp  movlw   255
    btfss   In_lo,7
    clrw
    xorwf   In_hi,W
    btfss   STATUS,Z
    retlw   255
    movf    In_lo,W
    movwf   Out
    retlw   0
```

   (a) This code can be initiated by the statement `call Lp`.
   (b) This code only returns the values 0 or 255 in `W`.
   (c) This code produces the 8-bit equivalent of the 16-bit number, only if possible, otherwise it returns 255.
   (d) This code tests the sign of the 16 bit number `In`, and returns 0 for positive numbers, and 255 for negative numbers.
   (e) This code uses the values `In_lo`, `In_hi` and modifies `Out`.

3.
```
xorwf   data,0
xorwf   data
xorwf   data,0
```

   What is the function of the above code fragment for the PIC16F877?

   (a) clear the working register, and the file register `data`
   (b) indirectly address the location whose address is in file register `data`
   (c) logical XOR the working register and file register `data` and leave the result in `data`
   (d) move the value from file register `data` into the working register without affecting the flags
   (e) swap the values in the working register and file register `data`

4.
```
BatLkup addwf   PCL,F
        retlw   A'B'
        retlw   A'a'
        retlw   A't'
        retlw   0x00
```
When the subroutine `BatLkup` is called with the value 2 in the working register, the value returned in the working register is:

   (a) 0x00
   (b) 0x02
   (c) A'B'
   (d) A'a'
   (e) A't'

5. What does the following code fragment for the PIC16F877 do?
```
        movlw   0x80
        movwf   FSR
Next    movwf   INDF
        incf    FSR,F
        btfss   FSR,6
        goto    Next
```
   (a) initialises memory locations 0x20 to 0x2F with the value 0x00
   (b) initialises memory locations 0x80 to 0xBF with the value 0x80
   (c) initialises memory locations 0x80 to 0xFF with the value 0x80
   (d) initialises memory locations 0x80 to 0xBF with the value 0x00
   (e) initialises memory locations 0x80 to 0xFF with the value 0x00

6.
```
my_sub  movlw   0x66
        addwf   Reg,W
        btfsc   STATUS,C
        retlw   1
        btfsc   STATUS,DC
        retlw   1
        retlw   0
```
Label the statements as True or false. The above piece of code:

   (a) adds 0x66 to the contents of the working register.
   (b) returns 1 if there is a carry or digit carry.
   (c) returns with 0x66 in the working register.
   (d) should be run with the instruction `goto my_sub`.
   (e) will be useful for checking the validity of BCD numbers.

7. Basic programming practice:

   (a) Write a piece of assembler code to branch to program location 1010h, presuming that the location is outside of the current program bank. (Note: the relevant bits must be set in PCLATH before the branch).

   (b) The `movf` instruction copies a byte from a register to the accumulator and affects the Zero flag. Write a piece of assembler code which will copy a byte from a register to accumulator and does not affect any of the status bits. (Hint: use `swapf`).

   (c) The notes include a code example which clears all memory locations between 20h and 30h. Write a similar piece of code which will set the values of memory locations 40h to 60h with the value that was in the accumulator at the start of the code.

   (d) Role Play/Challenge: The parity of a byte is even '0' if there is an even number of bits set, and odd '1' if there is an odd number of bits set. Write a piece of assembler code that will find the parity of a byte in a register, and leave the result in another register.

8. Lookup table/Indirect addressing practice:

   (a) Explain the operation of the indirect addressing mechanism on the PIC16F877 using the code snippet provided in the notes.

   (b) The notes contain example code for a lookup table using the DT directive, which presumes that there was no need to switch banks within the table. Write similar code which will operate correctly even when the table spans a program bank boundary.

   (c) Write a program containing a lookup table for multiplication by 7 in the data EEPROM. The lookup table should be accessed with a subroutine called `mul7` which will take the value in the working register as it's argument, and return the answer in the working register. Assume that `mul7` is only called with inputs from 0 to 15.

   (d) Role Play/Challenge:

      i. One of the drawbacks of the Harvard architecture is that instructions cannot directly execute reads of program memory. How does this affect the implementation of lookup tables, and the performance of lookups?

      ii. The PIC16F877 allows an indirect read of the program memory to be performed. Describe the mechanism used, using an assembler code example.

9. Representation & Mathematical practice:

   (a) BCD is not the only means of representing decimal numbers. The ASCII[30] character set defines a representation of characters as 7 bit numbers. The digit characters 0 through 9 are represented as 011 0000 thru 011 1001. Using this information write routines to correctly add and subtract two ASCII digits. (Note: The carry flag should be set for overflow, or no borrow respectively)

   (b) Presuming a two's complement 8-bit integer representation, write an algorithm and implement it in PIC assembly language to count the number of negative integers in a set of twenty numbers starting at address 30h. [Past Paper 2001, No. 1(c)]

   (c) Write an algorithm and implement it in PIC assembly language to subtract two 24-bit numbers, Q_1 and Q_2, and store the result in R. Ensure that the carry flag reflects the result of the subtraction. [Past Paper 2001, No. 2]

   (d) Write code to decrement a 16-bit variable and test the result for zero, branching to `Bothzero` if the result is zero. Can this be done using fewer lines of code and/or faster execution than simply concatenating the two routines shown previously?

   (e) Design an algorithm and implement it in PIC assembly language to perform unsigned division of a 24-bit number by an 8-bit number and produce a 16-bit quotient and 8-bit remainder.

   (f) Design an algorithm that works more efficiently than repeated subtraction for unsigned 8-bit by 8-bit division. Implement it in PIC assembly language.

   (g) Role Play/Challenge: Implement Booth's algorithm for two signed 8 bit numbers in PIC assembly language. It is presumed that the result will be 16 bits wide.

   (h) Role Play/Challenge: Using PIC assembly, implement Booth's algorithm for the multiplication of two 16-bit signed integers.

   (i) Role Play/Challenge: Design a Booth's-like algorithm for the division of a signed 16 bit number by a signed 8 bit number, and implement it in PIC assembly language.

[30] American Standard Code for Information Interchange

10. Data Structure practice

 (a) Role Play: It is necessary to implement a stack in software using indirect addressing. It must be 16 bytes long and can be located in any part of the GPR space. A variable `stkptr` is allocated as a pointer to the stack.

  i. When the microcontroller is reset, does `stkptr` need to be initialized? If so, what should its initial value be?

  ii. Assuming that the stack is never popped when empty, is it necessary that it be cleared when initialized?

  iii. Assuming that more than 16 items are never pushed onto the stack, write an algorithm for the push routine and implement it in PIC assembly language.

  iv. Do the same for the above for the pop routine.

 (b) Role Play: Another type of data structure is the queue i.e. a first-in, first-out data structure. The size of the queue in RAM is to be 16 bytes and an additional three bytes (variables) are allocated for tracking the data in the queue. The three variables are

 `inptr` Points to the next available input location.

 `outptr` Points to the next available output location.

 `qcount` Number of bytes of data in the queue.

  i. When the CPU is reset, how should each of the 19 bytes be initialized, if at all?

  ii. Assume , simplistically, that more than 16 bytes will never be stored in the queue. Write an algorithm for the `put` routine that puts a byte of data on the queue and implement it in assembly language.

  iii. Write and implement an algorithm for the `get` routine that retrieves a byte of data from the queue. It may be assumed that a `get` will never be executed unless `qcount` is non-zero.

  iv. In the queue put routine, add additional functionality that checks if the end of the queue has been reached and "wraps" the input and output pointers back to the beginning of the queue.

  v. In the queue get routine, add a check to prevent retrieval of data if the queue is empty.

  vi. In the queue put routine, add a check to prevent insertion of data if the queue is full.

Lab 2

ID# ______________________

You should complete the pre-lab before coming to your lab session. Your Teaching Assistant/Demonstrator may refuse to allow you into your lab session if your pre-lab is incomplete, or if you are unduly late.
You will have 3 hours in the lab to complete the exercises. Answers should be written on this lab-sheet in PEN. Please do not attach any extraneous pieces of paper unless SPECIFICALLY asked to do so.
Please bring your PIC16F877 datasheet book (provided in ECNG2006) to the lab with you. You may need to refer to it in order to complete the pre-lab and lab exercises.
Your lab submissions will be checked for unwarranted collusion, and unreferenced use of Intenet-available/other resources.

At the end of this unit the student will be able to:

> *implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.*

Pre-Lab

1. Please identify
   - your lab group letter (E,F,G,H): ______________________
   - your ECNG2006 uP group designator (e.g. A3): ______________________
2. Review basic operations/instructions for the PIC16F877 and answer the following questions about the PIC16F877:
   (a) What is a flag? *1 mark*

   (b) Where are the flags stored? *1 mark*

   (c) Give ONE example of when the Carry flag is cleared for an addition instruction. *1 mark*

   (d) Give ONE example of when the Carry flag is set for a subtraction instruction. *1 mark*

(e) Identify another arithmetic flag (apart from the carry flag) and give ONE example of when it is either set(Lab Groups E,F) OR cleared (Lab Groups G,H) by an addition instruction. *2 marks*

(f) Use an example to explain the behaviour of your assigned conditional instruction: Lab Group E: `btfss`, Lab Group F: `btfsc`, Lab Group G: `decfsz`, Lab Group H: `incfsz`. Your answer should state the conditions which exist BEFORE the instruction is executed, the operands of the instruction, and the conditions which exist AFTER the instruction is executed. *3 marks*

(g) Use the number 25_{16} to illustrate the difference between the `swapf`, `rlf`, and `rrf` instructions and explain how these instructions can be used to implement multiplication and division by powers of 2. *4 marks*

(h) Use the number 25_{16} and the bit mask 00000001_2 to illustrate the difference between the `andlw`, `iorlw`, and `xorlw` instructions and explain how these instructions can be used to extract and/or alter the state of a single bit (without affecting the state of other bits in the register). *4 marks*

3. Review the basic functionality of the MPASM assembler (use the manual/help files), paying particular attention to the topics discussed below:

   (a) What is a directive? Is it the same as an instruction? Explain your answer in your own words. *1 mark*

   (b) Differentiate between the EQU and ORG directives, in your own words. *1 mark*

   (c) In your own words, explain why 8-bit registers cannot be used to perform two's complement arithmetic with 8-bit numbers. Use an example for clarity. *1 mark*

   (d) Does MPASM automatically do two's complement arithmetic? Explain your answer. *1 mark*

   (e) MPASM is able to recognize numbers written in several different bases. Illustrate how the number equivalent to 27_9 would be written in an MPASM assembly language program using the base you have been assigned:
   Lab Grp E: base-10, Lab Grp F: base-16, Lab Grp G: base-2, Lab Grp H : base-8. *1 mark*

   (f) How does MPASM interpret a number whose base is not specified? *1 mark*

   (g) C compilers allow the programmer to re-use variable-names inside a subroutine, without interfering with data values of the same name held outside the subroutine. Does MPASM similarly allow us to have two variables of the same name? Explain your answer. *1 mark*

4. In lab 1, you were provided with C code in `looplogm.c` and assembly language code `asmlogm.asm` which each calculated the base-n logarithm of a number, *by repeatedly comparing the number with an increasing power of n*. The code is shown on pages 26 and 25.

   (a) The following questions about the C language code should be answered by annotating the code supplied on page 25.

      i. Circle places in the C code where there are loops and function calls. *2 marks*

      ii. Underline and label, C language instruction(s) used to implement loop initalization, loop completion testing, value change for loop variable, and loop repetition. *4 marks*

      iii. Underline and label, C language instruction(s) used to define, call, and return from a function? *3 marks*

      iv. Draw arrows to indicate the values to be changed if we wanted to change the logarithm being calculated from $\log_3 81$ to $\log_5 25$. *2 marks*

      v. How many C-language instructions are in the program? *1 mark*

      vi. Can the range(s) of data memory locations being used be determined from the source code? If so, what are they? *1 mark*

   (b) The following questions about the assembly language code should be answered by annotating the code supplied on page 26.

      i. Circle places in the assembly language code where there are loops and function calls. *2 marks*

      ii. Underline and label, assembly language instruction(s) used to implement loop initalization, loop completion testing, value change for loop variable, and loop repetition? *4 marks*

      iii. Underline and label, assembly language instruction(s), MPASM directive(s) and MPASM labels used to define, call and return from a function. *3 marks*

      iv. Draw arrows to indicate the values to be changed if we wanted to change the logarithm being calculated from $\log_3 81$ to $\log_5 25$. *2 marks*

      v. How many assembly language instructions are in the program? *1 mark*

      vi. Can the range(s) of data memory locations being used be determined from the source code? If so, what are they? *1 mark*

```
unsigned char logn(unsigned char x, unsigned char n)
{
            unsigned char i,j;

            j = 0;
            if(x==0) return 0xFF;            // error - 0 has no logarithm

            for (i=n; i<=x; i=i*n) j++;      // keep comparing with increasing powers

            return j;
}

int main()
{
            unsigned char a,b, answer;
            a = 81;
            b = 3;
            answer = logn(a,b);
}
```

Figure 6: `looplogm.c`

```
          INCLUDE   <P16f877.inc>
          LIST      p=16f877

          ORG       0x00

Num1      EQU       0x51
Num2      EQU       0x03

X         EQU       0x20
N         EQU       0x21
j         EQU       0x22
i         EQU       0x23
i2        EQU       0x27
count     EQU       0x24
check     EQU       0x25
check2    EQU       0x26
Answer    EQU       0x28

          ORG       0x20
main
          movlw     Num1
          movwf     X
          movlw     Num2
          movwf     N
          call      log
          movwf     Answer
          sleep

log       movf      X,F
          btfsc     STATUS,Z
          retlw     0xFF
          clrf      j
          movf      N,W
          movwf     i

compare   subwf     X,W
          btfss     STATUS,C
          goto      stop
          incf      j,F

next      movlw     0x04
          movwf     check
          clrf      i2
          movf      N,W
          movwf     check2
          movf      i,W
          movwf     count

next2     btfsc     check2,0
          addwf     i2,F
          bcf       STATUS,C
          rlf       count, F
          movf      count, W
          bcf       STATUS, C
          rrf       check2, F
          decfsz    check, F
          goto      next2
          movf      i2,W
          movwf     i
          goto      compare

stop      movf      j,W
          return

          END
```

Figure 7: asmlogm.asm

5. *(Bonus: 5 marks)* Algorithms are descriptions of how a function can be achieved. The algorithm used in the code provided on pages 26 and 25 is based on the principle of comparing the number to an incrementally increasing exponent of the base. This is not the only possible algorithm.

   Groups A1,B1,C1,D1,C4 Borcharts algorithm
   (e.g. http://www.dattalo.com/technical/theory/logs.html)

   Groups A2,B2,C2,D2,B4 Exploits floating point representation
   (e.g http://wehner.org/fpoint/)

   Groups A3,B3,C3,D3,A4,D4 Uses a lookup-table
   (e.g. http://www.dattalo.com/technical/software/pic/piclog.html)

   You were previously asked to read/describe the alternative algorithm you have been assigned in Lab 1. You should now implement it using the C language and build the code for the PIC16F877, using the techniques you practiced during Lab 1. Use the calculation of $\log_3 81$ $\log_5 25$ to test your code.

   Write down your initial try at the C language routine in the space below. Please use appropriate comments.

   Bring an electronic copy of the file to the lab. Filename is `L2Chlnge.c`. The first line of the file should contain a comment with your name and ID #.

Helpsheet: Writing and Testing Assembly Language code

When writing assembly language code, try to follow the following "Code formatting guidelines"

- labels in first column;
- code in second column;
- operands (where appropriate) in 3rd column;
- brief comments (where appropriate) in the 4th column
- directives in capital letters;
- assembly language in common letters;
- appropriate variable names/labels lower case (optional first capital letter).
- separate routines with a blank line

What is the purpose of these guidelines? Assembly language is inherently cryptic, and difficult to read. Following the guidelines "strange" instructions, or directives (e.g lacking an operand or placed in the wrong order) will be more easily spotted. This makes it easier to trouble shoot and maintain the code.

In keeping with the need to easily maintain your code, you should choose your variable (register) labels with care. It is a lot easier to find loops whose loop variables are named `loopvari` than those named `temp`, `myvar` or `fred`.

You will notice that **brief** comments should be included where appropriate. If you have chosen variable (register) labels appropriately, most code lines will not need explanation. You should use comments to explain the general intent of the program rather than the specifics of a particular instruction. Remember that at a later time, you will still be able to figure out what the instruction does, but you will have no idea why you placed it there. For example: andlw 0x0F ; bit-wise AND with 0x0F does not express intent, but andlw 0x0F ; extract lower nibble has clear meaning. Using comments in this way has the advantage that you can spot errors if the comment does not match the instruction – for example: andlw 0xF0 ; extract lower nibble clearly needs to be changed whereas andlw 0xF0 ; bit-wise AND with 0xF0 will not help.

Finally, when you are using test cases for your code, define the test values using labels at the top of the program, then use the label throughout. This way you can change the test cases in one place, without having to search for all the places that need to be changed.

In the lab

Before you start, please ensure that the computer is set up to show file extensions:

- Double click on My Computer then select the following from the menu(s) and pop-up windows: `Tools -- Folder Options -- View`.
- Ensure that the checkbox labelled `Hide file extensions` is clear.
- Click OK.

Next, create a directory on the `D:` drive . This is the directory you will work with during the lab. If you create this directory on the `D:` drive, please delete it at the end of the lab (you can copy it to your Z: drive or a key first!) :

- In the My Computer window, double-click on the icon for the drive.
- Right click and select New Folder.
- Name the directory `uPxxxxxxxx` where `xxxxxxxx` is your ID number.

During the lab you should create subdirectories for each question. The subdirectories should be named according to the lab and the question for example: `Lab1Q1`
Next, open MPLAB IDE. You will be asked to create new projects throughout this lab.
In Lab 1 you learnt to create a C language project. Refer to those instructions if needed.
To create an assembly language project:

- Click **PROJECT** on the menu bar and select **PROJECT WIZARD**. In the **PROJECT WIZARD** box click **NEXT**. From the **DEVICE** drop down menu select **PIC16F877** and click **NEXT**. Select **MPASM** or GPASM from the **ACTIVE TOOLSUITE** dropdown box and then click **NEXT.**
- Enter an appropriate name, for example "**Lab1Q1**" in the box named **PROJECT NAME**. Then for the **PROJECT DIRECTORY** box, browse and find the sub-directory you created for the question and click **NEXT**.
- A window will now come up with the buttons **ADD** and **REMOVE**. If you have already typed and/or saved your assembly language file, you may add your file. If not then you will need to crate it and add it to the project later. Click **FINISH** in the new window.
- Go to the menu bar and select **DEBUGGER** > **SELECT TOOL** > **MPLAB SIM**. This means that you will be executing your code on the simulator.
- To display the Program Memory click on the **VIEW** > **PROGRAM MEMORY** menu item.

Electronic copies of the code in this script are provided. Please note that there is no guarantee that the code will operate as required. Troubleshooting code is a skill which you are expected to use during this lab.
Call your TA or lecturer for assistance if you are stuck at any time.

PIC Basics

1. This question will investigate how to test if the value in a certain register is zero or not and how it may be used to our advantage. Create a new assembly language project. Use the MPSIM simulator windows to examine the data memory and the instruction memory. Build and step through the code below, using the MPLAB simulator:

```
        LIST p=16f877
        INCLUDE <P16F877.inc>

q       EQU     0x29

        ORG     0x00
        goto    main

        ORG     0x30
main    movlw   24
        movwf   q

loop    movf    q,F
        btfsc   STATUS,Z
        sleep
        goto    loop

        END
```

   (a) Use the simulator windows to identify ONE difference between the data and the instruction memories. *1 mark*

   (b) What location in the data memory is the variable `q` stored at? Suggest an alternative and explain your choice. *2 marks*

   (c) What location in the instruction memory is the instruction indicated by the label 'main' stored at? Suggest an alternate value and explain your choice. *2 marks*

   (d) Did the Zero (Z) flag change in response to `movlw` or `movwf` instructions? Explain why. *2 marks*

(e) Was the Z flag set(1) or reset(0) by the `movf` instruction in the program? Explain why. *1 mark*

(f) Describe how the registers, flags and program counter change when you step through the program. *1 mark*

(g) Replace the line `movlw 24` with `movlw 0`. Run the program again. Was the Z flag set or reset by the `movf` instruction? *1 mark*

____

(h) Describe how the registers, flags and program counter change when you step through the program. Has anything changed? *1 mark*

(i) Change the line `btfsc STATUS,Z` to `btfss STATUS,Z` and explain why the program behaves the way it does. *1 mark*

(j) Change the line `btfss STATUS,Z` to `btfss q,Z` and explain why the program behaves the way it does. *1 mark*

(k) Is the value specified on the line `movlw 24` a decimal value, or a hexadecimal value? Explain how you checked this. *1 mark*

(l) How would you write the instruction to *ensure* the assembler interprets the value as a hexadecimal number? *1 mark*

Basic Addition and Subtraction

2. This question will investigate how addition & subtraction instructions are performed. We will begin by attempting to add to simple unsigned 8 bit numbers. Let us try to add the numbers 19_{16} and 17_{16} and store the result in a register.

   Firstly perform the addition manually in the space given. *1 mark*

   In the code shown below, choose appropriate values for `q EQU` ______ and `ORG` ______. *2 marks*

   Choose values which are different to those given in the previous question.

   Create a new assembly language project. Enter the code, and step through it in the MPLAB simulator. Use watch windows to observe the working register, and the variable register `q`. Make sure you understand what the code does.

```
        LIST p=16f877
        INCLUDE <P16F877.inc>

q       EQU     -------

        ORG     0x00
        goto    main

        ORG     -------
main    movlw   17
        movwf   q
        movlw   19
        addwf   q,0
        sleep

        END
```

   (a) Explain how you chose an appropriate value for `q`. *1 mark*

   (b) Explain how you chose an appropriate value for the `ORG` statement. *1 mark*

   (c) In the line `addwf q,0`, what does the "0" mean? *1 mark*

(d) Run the program and check the result of the addition. Is the value what you expected? If not why not? How do you fix the program so it behaves as expected?

(e) Change the line `addwf q,0` to `addwf q,1`. How does this affect the program behaviour? *1 mark*

(f) Replace the `addwf` instruction with `subwf`, step through the code, and examine what happens. How does this affect the program behaviour? *2 marks*

(g) Underline the correct definition of `subwf`.
((value in w) - (value in f)) ((value in f) - (value in w))
and explain what w and f are.

2 marks

(h) Replace the `subwf` instruction with `sublw`, step through the code, and examine what happens. How does this affect the program behaviour? *2 marks*

(i) Replace the `sublw` instruction with `addlw`, step through the code, and examine what happens. How does this affect the program behaviour? *2 marks*

Arithmetic routines

3. Two's complement arithmetic can be used to perform subtraction, by adding the two's complement representations of

   - the minuend and
   - the negated subtrahend

   (a) The following code will convert the 7 bit number in `q` to the two's complement representation of q's negated value:

```
        LIST p=16f877
        INCLUDE <P16F877.inc>

q       EQU     _______

        ORG     0x00
        goto    main

        ORG     _______
main    movlw   _______
        movwf   q
        movf    q,W
        sublw   0xFF
        addlw   0x01
        movwf   q

        sleep

        END
```

   Fill in the blanks in the code, create a new assembly language project, enter the code, build the project, and step through the program to see the final value of `q`. Use the space at the side below to verify that the answer the program delivers is correct.

   Demonstrate your program to the lecturer/TA who will sign & mark your effort. *2 marks*

   (b) Let us move our code into a short routine called `tcmp` which will take the 7-bit value contained in the working register, and return with the 8-bit two's complement representation of the negated value. Fill in **appropriate** comments. *4 marks*

```
tcmp    andlw   0x7F    ;___________________________________________________

        sublw   0xFF    ;___________________________________________________

        addlw   0x01    ;___________________________________________________

        return          ;___________________________________________________
```

   (c) Let us write a program which will perform subtraction by calling the subroutine `tcmp`.

   The program must start with:

```
       LIST     ________________

       INCLUDE  ________________

Minud  EQU      ________________

Subtd  EQU      ________________

       ORG     _______

       goto    _______

       ORG     _______

______ movlw   Subtd

       call   tcmp

       addlw   Minud

       sleep

       ; place tcmp subroutine here!

       END
```

Fill in the blanks in the code. Create a new assembly language project, enter the code, build and step through the program to see the final value in the working register. Demonstrate your program to the lecturer/TA who will sign & mark your effort. *2 marks*

(d) `tcmp` is a routine which ends with a `return` instruction. Try using `goto tcmp` to call the routine. Did it cause a problem? Explain your answer. *2 marks*

(e) You were told to place the routine AFTER the `sleep` instruction, but before the `END` directive. Play around with other places for the routine. Could placing the routine incorrectly cause a problem? Explain your answer. *2 marks*

(f) In this program, are `Subtd` and `Minud` used as literal (values) or register addresses? Can the EQU directive only be used to "name" registers? Explain your answer. *2 marks*

4. Now we shall investigate unsigned 8-bit by unsigned 8-bit multiplication. When an 8-bit by 8-bit multiplication is performed, the result can potentially fill 16 bits. Therefore the answer must be stored in two separate registers. The code below will multiply two numbers M1 and M2. The results will be stored in RESHIGH and RESLOW. This code is based on the principle of repeated addition, and therefore involves a loop.

```
        LIST p=16f877
        INCLUDE <P16F877.inc>

        RESHIGH EQU     0x56
        RESLOW  EQU     0x55
        M1      EQU     0x0F
        M2      EQU     0xA9

        ORG     0x00
        goto    main

        ORG     0x20
main    clrf    RESHIGH
        clrf    RESLOW

        movlw   M1
        movwf   RESLOW
        movlw   M2

loop    decf    RESLOW,F
        movf    RESLOW,F
        btfsc   STATUS,Z
        goto    finish
        addlw   M2
        btfsc   STATUS,C
        incf    RESHIGH,F
        goto    loop

finish  movwf   RESLOW

        sleep
        END
```

(a) What values are being multiplied? ________ and ________ *1 mark*

(b) Circle the loop in the code *1 mark*

(c) Write in **appropriate** comments for the code *2 marks*

(d) Create a new assembly language project, enter the code, build and step through the program. What values ultimately end up in RESHIGH ________ and RESLOW ________ *1 mark*

(e) Does your code work? Yes or No (circle) If No, explain why your code does not work.

(f) Are there any values for which your program will fail? How did you find these values(if any), or ensure that there were none? *1 mark*

Demonstrate your findings to the lecturer/TA who will sign & mark your effort.

Binary Coded Decimal (BCD) Validity Testing. ***Bonus 10 marks***

5. (a) BCD uses a 4 bit number to represent the digits 0 (0000_2)through 9(1001_2). Any number higher than 9 will be considered an invalid BCD digit. Fill out the table below:

Binary Nibble	Decimal Equivalent	Valid BCD digit?
0011	3	Valid
1101	13	Invalid
0001		
1010		

(b) We already know that the carry flag is set when an operation causes the result byte to overflow. Similarly, the digit carry flag is set when an operation cause the lower nibble of the result byte to overflow into the upper nibble. The name, digit carry, stems from the usefulness of this flag in the manipulation of BCD numbers.

The highest valid BCD digit is 1001_2 (9) . What is the smallest number that can be added to 1010_2 (10) that will cause the DC flag to be set?

Express this number in BINARY ________ and DECIMAL ________.

(c) The return with value `retlw` instruction places the operand into the working register BEFORE executing a normal return. Just like the regular `return` it will be the last statement *executed* in a routine, but a single routine can have multiple places at which it `return`'s. Which `return` or `retlw` is used depends on the order in which the instructions within the routine are executed, and the values in the registers at the time the routine is executed.

The computed-goto operates by using an arithmetic operation whose *destination* is the program counter e.g. `iorwf PCL,F`, `addwf PCL,F`. In this way we change the address of the next instruction that will be executed. The exact value of the new instruction address will depend upon the value in the working register at the time of execution.

The `retlw` and `addwf PCL,F` instructions are often combined to form a lookup table. The lookup table will return a value from a list of values, depending on the index passed in the working register. The first item in the list will correspond to index value 0. To make a lookup table, you must identify an appropriate answer for EVERY possible value of the index. Masking is often used to ensure that *invalid* index values are not used.

The 5 times table for a 2-bit index could be implemented as:

```
times5     andlw     _____________

lkup       addwf     PCL, ________

           retlw     d'0'

           retlw     _____________

           retlw     _____________

           retlw     d'15'
```

(d) The routines `BCDChkA` and `BCDChkB` will check if the *lower nibble* of the register `Reg` contains a valid BCD digit. If it does, the working register is set to 0, otherwise the working register is set to 1. The implementations of the routines however are different. One uses the digit carry-test, and the other uses a lookup table. Create a new assembly language project, and a main program which can be used to compare the two versions.

i. Did your program compile and work straight away? If it didn't what kinds of errors did you have to fix in order to get it to work? How did you track down these errors?

ii. Choose two test cases and run them using each version. Did your program give the correct answer for each test case? If it didn't what kinds of errors did you have to fix in order to get it to work? How did you track down these errors?

iii. Print the final version of the main program and attach it to this script. Record your comparison of the two routines in the space below.

Units	instruction-words	Case A cycles	Case B cycles
BCDChkA			
BCDChkB			

Table 2: BCDChk Performance comparison results

Demonstrate your program to the lecturer/TA who will sign & note on your printout, whether the program compiled, whether it worked on each test case, and whether you can explain how each routine works.

```
BCDchkA   movlw     0x0F
          andwf     Reg,W
          movlw     0x06
          addwf     Reg,W
          btfsc     STATUS,DC
          retlw     1
          retlw     0
```

```
BCDChkB   movlw     0x0F
          andwf     Reg,W
lkup      addwf     PCL,F
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     0
          retlw     1
          retlw     1
          retlw     1
          retlw     1
          retlw     1
```

Logarithms again ....*Bonus: 10 marks*

6. Create a new C language project, enter, build and test the C language code you wrote for your pre-lab exercise, using the MPSIM simulator.

   (a) Did your program compile and work straight away? If it didn't what kinds of errors did you have to fix in order to get it to work? How did you track down these errors?

   (b) Did your program give the correct answer for each test case? If it didn't what kinds of errors did you have to fix in order to get it to work? How did you track down these errors?

   (c) Print the final version of the code and attach it to this script. Demonstrate your program to the lecturer/TA who will sign & note on your printout, whether the program compiled, whether it worked on each test case, and whether you can explain the underlying algorithm.

   (d) Compare your code to previous programs supplied by taking appropriate measurements to complete the following table, which is based on the same performance criteria used in Lab 1. Indicate which Build Tool (GPASM/MPASM/SDCC/CCSC) was used in each case:

Item	Build Tool	I	II	III	IV	V
Units		bytes	instruction-words	data-words	cycles	cycles
`looplogm.c`						
`asmlogm.asm`						
`L2Chlnge.c`						

Table 3: Logarithms – Performance comparison results

   (e) Use your data to identify the "best" performing implementation. How did the language, the build tool, and the algorithm implemented, contribute to make this the "best"?

Total marks 100. This exercise is worth 5% of your ECNG2005 lab mark.

Unit 15 Write the number you have been assigned here________.

Answer the following questions for the TARGET platform whose number you were assigned.

1. What is the name of the target platform? ____________________.
2. What are the key architectural features of the target platform? ____________________.
3. List all available addition and subtraction instructions.
4. List all call/return/branch instructions.
5. Write a routine to find the two's complement of a number that is in a specified register when the routine is called. The answer should be in the same register at the end of the routine.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this lab exercise.
  - utilize the MPLAB IDE and CCS compiler, to develop software for the MicroChip PIC16Cxxx series of microcontroller, in C, C++ and assembly language.
  - implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.
- Which aspect of this lab exercise did you have the most difficulty understanding?
- Which aspect of this lab exercise did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this lab exercise.
- Identify one way in which this lab exercise could be improved.

16 Code comparison

At the end of this unit the student will be able to:

> *contrast alternate programming techniques in terms of size/speed of the code produced for the PIC16Cxxx series of microcontrollers.*

When assessing assembly language code, there are three major considerations: is it syntactically correct? ;is it logically correct? ;is it optimally sized/fast?

Syntax errors are generally picked up by the assembler and reported with a message indicating what the assembler "thinks" the problem is. These messages while useful, can sometimes be misleading; remember that they reflect the most likely cause of the error, but not the actual cause. For example, placing a label on the same line as the BANKSEL pre-processor directive causes MPASM to erroneously report that the label is not defined.

Logical errors occur when the program is written to do something that you didn't intend to do.

> Typical Assembly Language bugs include:
>
> - Improper transfer to subroutines
> - Using subroutines that wipe out registers
> - Transposed commands
> - Not initializing registers [or checking initial conditions]
> - Modifying condition code before conditional branch
> - Using the wrong conditional instruction
> - Not stopping the program properly
> - The use/change of "magic" numbers
>
> – summarized from (Cady 1997)

Two pieces of code for the same processor may be compared in terms of the number of program memory locations needed to store the code, the number of data memory or register locations required to store constants/variables, and the time the code takes to execute. For simple processors with a fixed pipeline recovery policy (like the PIC16F877), we can perform rudimentary performance prediction by counting instructions (and hence instruction cycles). Very often performance of code can be improved, simply by making a few simple adjustments. For example, when testing for the end of a loop, there are many failures of the loop tests, and only one success. Therefore, and improvement in the code in the failure branch will have a greater impact on program execution time.

Review Exercises

Questions 1–5 refer to the following code snippets:

```
Snippet A                Snippet B
----------               --------------
                         movwf   tmpA
```

Syntax error

```
        LIST  p=16c64

Dest    EQU     H'0B'
        ORG     H'01FF'
        goto  Start

        ORG     H'0000'
Start
        BANKSEL  0x00
        movlw  0
        movwf  Dest
        movf  Dest,F
        goto  Start
        END
```

```
                      00001         LIST p=16c64
                      00002
  0000000B            00003 Dest    EQU      H'0B'
01FF                  00004         ORG      H'01FF'
Error[113]  : Symbol not previously defined (Start)
01FF   2800           00005         goto Start
                      00006
0000                  00007         ORG      H'0000'
Error[121]  : Illegal label (Start)
0000   1283           00008 Start   BANKSEL 0x00
0001   3000           00009         movlw 0
0002   008B           00010         movwf Dest
0003   088B           00011         movf Dest,F
Error[113]  : Symbol not previously defined (Start)
0004   2800           00012         goto Start
                      00013         END
```

Logical error

- Improper transfer to subroutines
- Using subroutines that wipe out registers
- Transposed commands
- Initialising registers/ checking initial value
- Modifying condition code before conditional branch
- Using the wrong conditional instruction
- Not stopping the program properly
- The use/change of "magic" numbers

```
N       EQU     0x14

        ORG 0x00
        goto main

        ORG 0x20
main    movlw   N
        addwf   q,F       ; q=N
        goto    addnum    ; w=q+num
        btfss   STATUS,C   ; if C flag is clear
        sleep             ; stop program
        call main         ; otherwise loop to main

addnum  ; add num to q and return result in W
        movlw   4
        addwf   q,F
        movf    q,W
        btfss   STATUS,Z ; if Z flag set
        nop               ; stop executing program
        return
END
```

Decrementing a 16 bit number 2001/2, Page IV-6, Q.1

```
movf    countl,F
btfsc   STATUS,Z
decf    counth,F
decf    countl,F
movf    countl,F
btfsc   STATUS,Z
movf    count,F
```

Redundant

s

```
btfsc   STATUS,Z
goto    Bz
goto    Nbz
```

Original Code:

- 1 case: 2+8+2
- 2^{16}-1 cases: 2+9+2
- Avg. Cycles: 9
- Max. Cycles: 9

Modified Code:

- Remove 1 line
- Swap 2 lines
- 1 case: 2+8+2
- 2^{16}-1 cases: 2+7+2
- Avg. Cycles: 7
- Max. Cycles: 8

2001/2, Page IV-6, Q.1 Tutorial Group Solution

```
movlw   1
subwf   countl,F
btfss   STATUS,C
decfsz  counth,F
btfss   STATUS,Z
goto    Nbz
movf    counth,W
btfss   STATUS,Z
goto    Nbz
goto    Bz
```

Subtraction of 1 can never result in both a borrow and a zero result i.e. if the carry flag is clear, the zero flag is never set.

Early Exit

- $\overline{C}$, $\overline{Z}$ 256 cases: 2+5+2
- C, $\overline{Z}$ 256*254 cases: 2+5+2

Remaining C, Z

- 255 cases 2+8+2
- 1 case: 2+9+2

- Avg. Cycles: 5
- Max. Cycles: 9

```
                         movf    data,W
xorwf data,W             movwf   tmpB
xorwf data,F             movf    tmpA,W
xorwf data,W             movwf   data
return                   movf    tmpB,W
                         return
```

1. What do these snippets do?
   (a) swap the contents of `data` and `tmpA`
   (b) apply the bit mask in `W` to `data`
   (c) swap the upper and lower nibbles of `data`
   (d) swap the contents of `data` and `W`
   (e) invert all the bits of `data`

2. The number of instruction cycles these snippets will take to run (disregarding the incoming branch/call instruction) are:
   (a) A: 4 B: 7
   (b) A: 4 B: 8
   (c) A: 5 B: 8
   (d) A: 5 B: 7

3. The number of file registers required by each snippet:
   (a) A: 2 B: 4
   (b) A: 1 B: 3
   (c) A: 1 B: 4

4. The instruction storage required by each snippet:
   (a) A: 3 B: 6
   (b) A: 7 B: 4
   (c) A: 4 B: 7

5. The flags affected by each snippet are:
   (a) A: Zero flag only B: Zero flag only
   (b) A: none B: Zero flag only
   (c) A: Zero flag only B: none

Questions 6–8 refer to the following code snippets:

```
Snippet A                Snippet B
---------------          --------------


movf    datalo,W         movf    datalo,W
```

```
addwf   resultlo,F          addwf   resultlo,F
movf    datahi,W            movf    datahi,W
btfsc   STATUS,C            btfsc   STATUS,C
incf    resulthi,F          incfsz  resulthi,F
addwf   resulthi,F          addwf   resulthi,W
return                      movwf   resulthi
                            return
```

6. Label the following statements as True or False.

   (a) Snippet A will always take 8 instruction cycles to run; Snippet B will always take 9 instruction cycles

   (b) These snippets perform 16 bit addition of `datahi:datalo` to `resulthi:resultlo`, and leave the answer in the working register

   (c) Snippet A may leave the carry flag in a state which does not reflect the 16 bit addition.

   (d) Both snippets use the same number of data registers.

   (e) Both snippets return with the low byte of the result in the working register.

7. The number of program memory locations required by each snippet:

   (a) A: 4 B: 4

   (b) A: 7 B: 8

   (c) A: 8 B: 7

   (d) A: 8 B: 9

   (e) A: 9 B: 8

8. The flags affected by each snippet are:

   (a) A: Carry Flag only B: Carry Flag only

   (b) A: Carry flag only B: Zero and Carry flags only

   (c) A: Zero and Carry flags only B: Zero and Carry Flags only

   (d) A: Zero, Carry and Digit Carry Flags B: Zero, Carry and Digit Carry Flags

9. The following code changes the sign of Reg, so that it is always interpreted as a positive number in the two's complement representation scheme. How many cycles does the following code for the PIC16F877 take for negative and positive numbers respectively?

```
        movf   Reg,W
        btfss  Reg,7
        goto   Done
        sublw  0
Done    movwf  Reg
```

   (a) 7 cycles; 5 cycles
   (b) 7 cycles; 4 cycles
   (c) 5 cycles; 5 cycles
   (d) 5 cycles; 4 cycles
   (e) none of the above

10. You are writing a program which loops continuously. Within the loop, you are required to alternately set and clear bit 4 in an 8-bit register. The shortest way to successfully do so at each loop iteration is to:

   (a) test the bit; if true bit-wise AND with EF_{16}; if false bit-wise OR with 10_{16}
   (b) test the bit; if true bit-wise OR with 10_{16}; if false bit-wise AND with EF_{16}
   (c) test the bit; if true bit-wise AND with 10_{16}; if false bit-wise OR with 10_{16}
   (d) bitwise XOR with EF_{16}
   (e) bitwise XOR with 08_{16}

11. Role Play/Challenge: Programming Scenarios.

   TIP: The following exercises are best attempted on paper FIRST, then on the computer. The key is to think about what is right/wrong rather than make "mad" changes.

(a) As part of a design team, you have been asked to complete a piece of assembly language code for the PIC16F877 that will count the number of capital letters in a null-terminated variable length string. You are given an initial piece of code, which may have syntactical and logical errors in it.

i. Identify all syntactical and logical errors in the code, and re-write the routine if necessary. Use instruction counting to determine how many cycles your routine will take.

```
        #include <P16F877.inc>

Cnt     EQU H'0B'

        ORG     G'0000'
        goto    Start

        ORG 0x0020
Start   BANKSEL Cnt
        clrw
loop    goto    Table           ; get next character from Table
        btfsc   STATUS,Z        ; if the value is 0 we are done
        goto    Done
        sublw   'A'             ; if the ASCII value is less than A
        btfsc   STATUS,C        ; goto loop
        goto    loop
        subwl   'Z'-'A'         ; if the ASCII value is between A and Z
        btfsc   STATUS,C        ; increment Cnt register
        incf    Cnt,W
        goto    loop

Table   addwf   PCL,F           ; Lookup table to return the string characters
        DT      "6 Char", D'0'  ; Null-terminated string.

Done    sleep

    END
```

ii. Optimize the code for the string given. Print the code, and illustrate how many cycles your improved code will take.

iii. Optimize the code presuming that the ratio of capital to common letters in a given string is 10:20. Print the code, and illustrate how many cycles your improved code will take.

(b) "A simple digital low pass filter can be implemented using the algorithm: $[A_{n-1}] = \frac{S_n}{4} + \frac{S_{n-1}}{2} + \frac{S_{n-2}}{4}$ where S_n is the n^{th} sample from an eight bit analog to digital converter located at Port B." (Katzen 2003; p. 134)

As part of a design team, you have been asked to complete a piece of assembly language code for a low pass filter on the PIC16F877. At any time the filtered value should be placed on Port C. You are given an initial piece of code.

i. Identify all syntactical and logical errors (if any) in the code.

```
        #include <P16F84.inc>

Sn      EQU     H'0B'
Snm1    EQU     H'0C'
Snm2    EQU     H'0D'

        ORG     G'0000'
        goto    Start

        ORG     0x0020
Start
        BANKSEL PORTB
        clrw

Loop    movf    PORTB,W         ; get value from PortB
        movwf   Sn              ; store the value

        addwf   Snm2,W          ; add Sn to Snm2
        rrf     W               ; divide by 2

        addwf   Snm1,W          ; add Snm1 to (Snm2+Sn)/2
        rrf     W               ; divide by 2

        movwf   PORTC           ; put out the output

        movf    Sn              ; move Sn to Sn-1
        movwf   Snm1

        movf    Snm1,W          ; move Sn-1 to Sn-2
        movwf   Snm2

        goto    Loop

Done    sleep

        END
```

ii. Fix the errors and compile the code. Explain how you tested this code. Print the code, and illustrate using the instruction counting technique, how many cycles this code will take to run.

iii. Optimize the code presuming that the A/D converter never generates values greater than 5 bits wide. Print the code, and illustrate how many cycles your improved code will take.

(c) As part of a design team you have been asked to complete a piece of assembly language code which produces a "continuous" histogram of sampled data values. The data is 8 bits wide and is sampled at intervals from PORTB. The histogram has 8 bins. Every 128 samples all bins in the histogram are halved. This ensures that no bin overflows, and that the most recent updates are reflected most strongly in the histogram. The histogram bin values are stored in 8 consecutive locations in memory.

You are given an initial piece of code(over) which may have syntactical and logical errors in it.

i. Identify all syntactical and logical errors you can initially see in the code.
ii. Fix the errors and compile the code. Check that it works using the simulator (you should explain how you tested this code). Print the code, and illustrate using the instruction counting technique, how many cycles this code will take to run. Run the program and use the stopwatch to verify your answer.
iii. Optimize the code presuming that the sampled value is never greater than 7 bits wide, and must still be divided into 8 histogram bins. Print the code, and illustrate how many cycles your improved code will take.

```
        LIST    p=16f877
        INCLUDE <P16F687.inc>

counter EQU     0x10
sample  EQU     0x11
array   EQU     0x12


        ORG     0x00
        goto    Start

        ORG     0x20
Start
        clrf    counter

LpSmpl  movf    PORTB,W         ; get value from PORTB
        movwf   sample          ; store in sample
        swapf   sample,W        ; derive bin # from sample value
        rrf     sample,F
        movlw   B'00000111'
        iorwf   sample,F
        movlw   array           ; access the relevant bin & increment it
        movwf   FSR
        movf    sample,W
        addwf   FSR,W
        incf    INDF,F
        incf    counter,F       ; add 2 to counter -- if no overflow repeat the sampling
        incfsz  counter,F
        goto    LpSmple

        movlw   array           ; initialise FSR
        movf    FSR
LpRot   rrf     INDF,F          ; divide each bin by 2
        incf    FSR,F
        movlw   array+7         ; is this the last bin?
        subwf   FSR,W
        btfsc   STATUS,Z
        goto    LpRot
        goto    LpSmpl

        sleep

        END
```

(d) Question is based on (Wilmhurst 2001; Q3.8,Q3.9).

As part of a design team you have been asked to complete a piece of assembly language code for a PIC16F877 with an external Digital-to-Analogue Converter connected to $PORTC < 5 : 0 >$. The PIC16F877 must send values to the DAC such that a ramp waveform is generated. The frequency of the waveform is determined by two switches connected to $PORTC < 7 : 6 >$, as follows:

$PORTC < 7 >$	$PORTC < 6 >$	Frequency
0	0	0 Hz
0	1	0.25 kHz
1	0	1 kHz
1	1	4 kHz

A 4MHz oscillator provides the clock signal to the PIC16F877.

You are given an initial piece of code(over) which may have syntactical and logical errors in it.

i. Identify (using comments on the code sheet) all the syntactical and logical errors you can initially see in the code.

ii. Fix the errors and compile the code. Check that it works using the simulator (you should explain how you tested this code). Print the code, and illustrate using the instruction counting technique, how many cycles this code will take to run. Run the program and use the stopwatch to verify your answer.

iii. Improve the code so that the ACTUAL ramp frequency is closer to the SPECIFIED frequency and/or the code takes less space. Print the code, and illustrate how many cycles your improved code will take.

```
        LIST    p=16f877
        INCLUDE <P16F678.inc>

        ORG 0x00
        goto init

        ORG 0x20
int
        BANKSEL PORTC       ; configure PORTC
        movlw   0xC0
        movwf   TRISC
        BANKSEL PORTC

loop    decf    Cnt         ; wait until Cnt is 0 then skip
        goto    lop

        decf    Out,F       ; ramp up output by 1 notch
        movf    Out,W
        movwf   PORTC

chk     movf    PORTC,F     ; check to see what switches are set to
        andlw   0xC0
        btfsc   STATUS,Z
        goto    no_out

        movwf   Swt         ; get switch bits into lsb's of W
        rrf     Swt,W
        rlf     Swt,F
        swapf   Swt,W
        call    tbl         ; retrieve the correct count
        movwf   Cnt
        goto    loop

no_out  clrf    PORTC       ; zero all output to DAC
        goto    chk

tbl     addwf   PCL,W       ; lookup table
        retlw   0           ; no counting for 0Hz
        addlw   16          ; count 16 for 0.25kHz;
        return  4           ; count 4 for 1 kHz
        retlw   1           ; count 1 for 4 kHz

        END
```

Assignment C

ID# ____________

An engineering design company is developing an embedded system involving a non-linear pressure sensor and a PIC16F877 with a 4MHz clock signal.

The pressure sensor is activated by a falling edge on PORTC<1>. The output of the pressure sensor on PORTC<2> is held high for a length of time.

As a member of the design team you have been asked to complete a piece of code for the PIC16F877.

The code should determine the width of the logic-high pulse in microseconds (μs), and then convert it to a pressure reading in pounds-per-square-inch (psi). The code will be assembled using the MPASM assembler.

The pressure can be determined from the time using: $p = 4 \times \log_2 t$
where t is the pulse width in μs, and p is the pressure in psi.

The design team has made an initial decision to represent the pressure reading using a 4 bit unsigned integer.

A correctly functioning pressure sensor will output logic-high for between 1 and 15 microseconds after being activated.

Your initial attempt at the code is shown on page 53.

1. Identify (using comments or a separate text file) all the syntactical and logical errors you can initially see in the code. *12 marks*

2. Fix the errors and compile the code. Check that it works using the simulator (you should explain how you tested this code). Illustrate using the instruction counting technique,(with comments or a in separate text file) how many cycles this code will take to run. Run the program and use the stopwatch to verify your answer. *16 marks*

3. Improve the code so that the accuracy of the RECORDED pressure is improved and/or the code takes less space. Illustrate (with comments or a in separate text file) how many cycles/storage locations your improved code will take. You should also identify how accurate the pressure reading is. *12 marks*

```
        LIST        p=16f877
        INCLUDE     <P18F452.inc>

Cntr    EQU         0x12

        ORG         0x10
        goto        init           ; at the reset vector, branch to start of code

        ORG         0x80
init
        BANKSEL     TRISC          ; configure PORTC<2> as an input, and PORTC<1> as an output
        bcf         TRISC, 2
        bcf         TRISC,1
        BANKSEL     PORTC

main    call        read           ; repeat the reading
        goto        man
        sleep

read    clrf        Cntr           ; clear counter
        bcf         PORTC,1        ; output falling edge on PORTC<1>
        bsf         PORTC,1

loop    incf        Cntr,W         ; count up until the signal goes low
        btfss       PORTC,1
        goto        loop

        goto        convert        ; call routine to convert time to pressure
        movwf       PrRdg
        return

convert iorlw       0x0F           ; mask to extract lower bits of byte
        addwf       PCL,F
        retlw       0xFF           ; return error value
        retlw       b'0000'        ; 4*log_2(1 microseconds)= 0 psi
        retlw       b'0100'        ; 4*log_2(2 microseconds)= 4 psi
        retlw       b'0110'        ; 4*log_2(3 microseconds)= 6 psi
        retlw       b'1000'        ; 4*log_2(4 microseconds)= 8 psi
        retlw       b'1001'        ; 4*log_2(5 microseconds)= 9 psi
        retlw       b'1010'        ; 4*log_2(6 microseconds)=10psi
        retlw       b'1011'        ; 4*log_2(7 microseconds)=11psi
        retlw       b'1100'        ; 4*log_2(8 microseconds)=12psi
        retlw       b'1100'        ; 4*log_2(8 microseconds)=12psi
        retlw       b'1101'        ; 4*log_2(8 microseconds)=13psi
        retlw       b'1101'        ; 4*log_2(8 microseconds)=13psi
        retlw       b'1110'        ; 4*log_2(8 microseconds)=14psi
        retlw       b'1110'        ; 4*log_2(8 microseconds)=14psi
        retlw       b'1111'        ; 4*log_2(8 microseconds)=15psi
        retlw       b'1111'        ; 4*log_2(8 microseconds)=15psi

        END
```

Assignment C Submission Guidelines:

You will be awarded (up to a maximum of 4) bonus marks, for the use of exceptionally efficient methods. An exceptionally efficient method is one which is at least 20% shorter or 5% more accurate than the median program length/accuracy of ALL submissions. i.e. you will compete with your peers. You will be penalised (up to a maximum of 2) marks if you violate ANY of the program formatting guidelines.

All submitted printout/programs MUST follow the specified format i.e.

- labels in first column;
- code in second column;
- operands (where appropriate) in 3rd column;
- comments (where appropriate) in the 4th column
- directives in capital letters;
- assembly language in common letters;
- variable names/labels lower case (optional first capital letter).

Your files should be named Cqnnnnnnnn.* where nnnnnnnn is your ID Number and q is the question number.

Electronic Submission:

Please type your answers into plain ASCII text files (no Word documents please!) with the same name(s) as your other files, and the extension .asc . Alternatively you may write your answers in the *.asm files as comments.

The *.asc, *.hex, *.asm and *.lst files for all questions should be zipped into a single archive named Cnnnnnnnn.zip where nnnnnnnn is your ID Number.

The archive should be uploaded to Moodle using the appropriate link.Please remember to electronically submit AHEAD of the deadline in order to avoid problems with system overload.

Unit 16 Write the letter you have been assigned here________.
Letters refer either to the code in the figure opposite, or which your lecturer will put up.

1. What does your code do?
   (a) swap the values in W and a file register named data
   (b) lookup a value at the Wth position in the table
   (c) determine if all the masked bits are set
   (d) add two 16 bit values
2. Using the initial values provided, write down the line #'s for Q in order of execution.
3. Using the initial values provided, what is the final value in the accumulator after Q executes
4. Using the initial values provided, what are the final values of the Z, C, and DC flags (write unchanged if they have not been altered in the code) after Q executes?
5. How many program memory locations will be occupied by each piece of code R & S.
6. On average, how many instruction cycles will it take to run R & S
7. Which piece of code is better, R or S? Explain your answer.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - contrast alternate programming techniques in terms of size/speed of the code produced for the PIC16Cxxx series of microcontrollers.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

	Q	R	S
A			
	1: BatLkup addwf PCL,F	BatLkup addwf PCL,F	BatLkup addwf PCL,F
	2: retlw A‘B’	retlw A‘B’	DT "Bat",0x00
	3: retlw A‘a’	retlw A‘a’	
	4: retlw A‘t’	retlw A‘t’	
	5: retlw 0x00	retlw 0x00	
	Initial value:		
	W=0x01		
B			
	1: chkset andwf MASK,W	chkset andwf MASK,W	chkset andwf MASK,W
	2: xorwf MASK,W	xorwf MASK,W	xorwf MASK,W
	3: btfss STATUS,Z	btfss STATUS,Z	btfsc STATUS,Z
	4: retlw 0	retlw 0	retlw 1
	5: retlw 1	retlw 1	retlw 0
	Initial values:		
	W=0xA7;		
	MASK=0x13;		
C			
	1: Swap xorwf data,W	Swap movwf tempA	Swap xorwf data,W
	2: xorwf data,F	movf data, W	xorwf data,F
	3: xorwf data,W	movwf tempB	xorwf data,W
	4: return	movf tempA, W	return
		movwf data	
	Initial values:	movf tempB, W	
	W=0xFD;	return	
	data=0x00		
D			
	1: Add16 movf datalo,W	Add16 movf datalo,W	Add16 movf datalo,W
	2: addwf resultlo,F	addwf resultlo,F	addwf resultlo,F
	3: movf datahi,W	movf datahi,W	movf datahi,W
	4: btfsc STATUS,C		
	5: incf resulthi,F	btfsc STATUS,C	btfsc STATUS,C
	6: addwf resulthi,F	incf resulthi,F	incfsz resulthi,F
	7: return	addwf resulthi,F	addwf resulthi,W
	Initial values:	return	movwf resulthi
	W=0xF4;		return
	datalo=0x40;		
	datahi=0xCD;		
	resulthi=0x00;		
	resultlo=0XFA		

17 Compiler limitations

At the end of this unit the student will be able to:

> *give examples of the limitations placed on compiler-generated code, for the PIC16Cxxx series of microcontrollers, by the need for generality.*

The purpose of a higher level language, is to offer the programmer an environment which will behave in a standard way regardless of the platform the program is compiled for. This places constraints on assumptions the compiler can make when translating your code. If you are aware of the assumptions which the compiler makes in translating code, you can exploit them to obtain the best possible performance from your C code. Similarly, if you are aware of the assumptions the compiler makes, you can avoid the generation of invalid code.

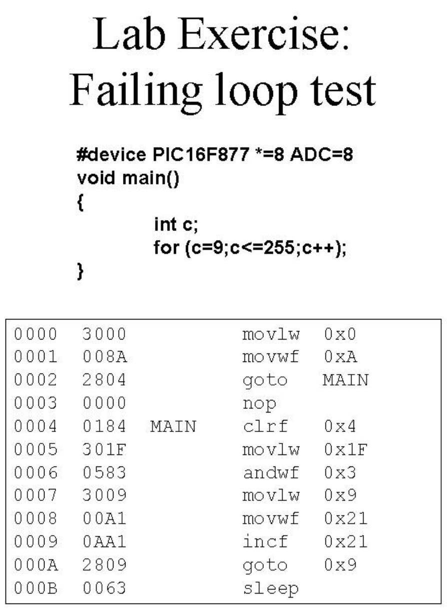
Typically problems occur with:

- boundary checking
- excessive code – type translation
- code spanning program banks
- variables located in different data banks

Some tips gathered from `http://www.microchipc.com/` for PIC16Cxxx C programming, appear below. These apply to assembly as well, but tend to be forgotten/ignored when programming in C.

- Use unsigned integers and/or chars where possible.
- Use byte (not word) arithmetic where possible.
- Some variables will not need initialization; you can save code if you leave it out.
- Order operations to take advantage of values already in the accumulator.
- Perform loop termination tests against zero where possible. When doing a multi-word count-down loop, you only need to test the uppermost byte.
- Avoid multiplication and especially division – replace with shifts where possible
- Initialise/use variables in the same bank together before doing variables located in another bank.
- Use of functions can reduce your code size, but, may incur a speed penalty.
- Zero, increment and decrement will always generate more efficient code than value assignment.
- For large projects, there are three approaches to producing code:
  - compile one C file;
  - compile one main C file which includes all the other files;
  - compile each C files separately, and then link them together

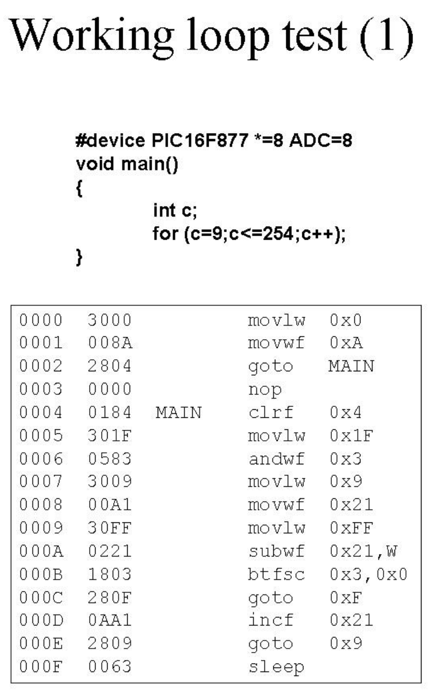
  Debugging support for latter two is not as extensive, but they allow the development/use of re-usable code modules. First two generate a smaller final binary – less space wasted in-page. The last one offers incremental compilation.



Lab Exercise: Failing loop test

```
#device PIC16F877 *=8 ADC=8
void main()
{
        int c;
        for (c=9;c<=255;c++);
}
```

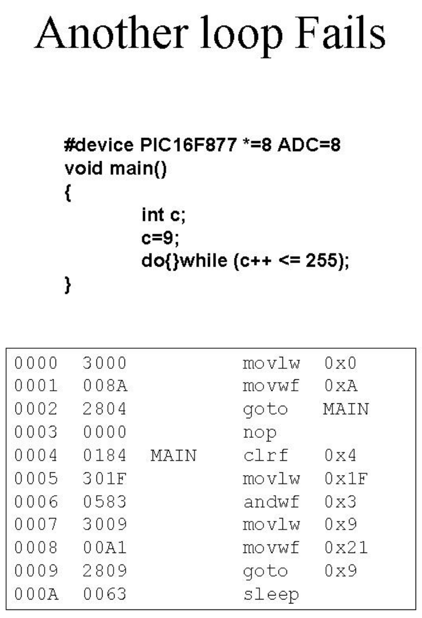
```
0000  3000          movlw  0x0
0001  008A          movwf  0xA
0002  2804          goto   MAIN
0003  0000          nop
0004  0184  MAIN    clrf   0x4
0005  301F          movlw  0x1F
0006  0583          andwf  0x3
0007  3009          movlw  0x9
0008  00A1          movwf  0x21
0009  0AA1          incf   0x21
000A  2809          goto   0x9
000B  0063          sleep
```



Working loop test (1)

```
#device PIC16F877 *=8 ADC=8
void main()
{
        int c;
        for (c=9;c<=254;c++);
}
```

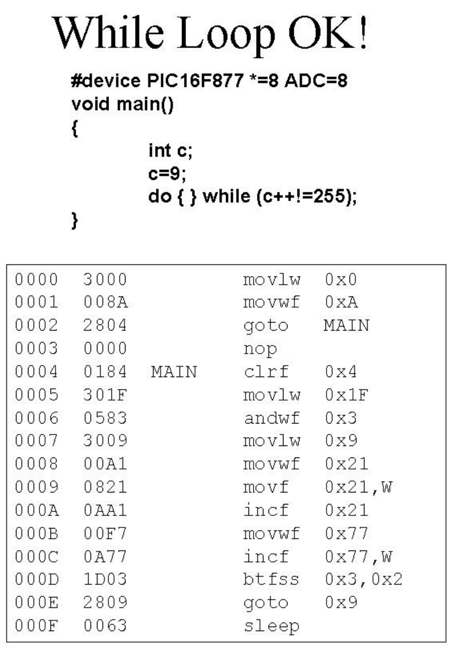
```
0000  3000          movlw  0x0
0001  008A          movwf  0xA
0002  2804          goto   MAIN
0003  0000          nop
0004  0184  MAIN    clrf   0x4
0005  301F          movlw  0x1F
0006  0583          andwf  0x3
0007  3009          movlw  0x9
0008  00A1          movwf  0x21
0009  30FF          movlw  0xFF
000A  0221          subwf  0x21,W
000B  1803          btfsc  0x3,0x0
000C  280F          goto   0xF
000D  0AA1          incf   0x21
000E  2809          goto   0x9
000F  0063          sleep
```



Another loop Fails

```
#device PIC16F877 *=8 ADC=8
void main()
{
        int c;
        c=9;
        do{}while (c++ <= 255);
}
```

```
0000  3000          movlw  0x0
0001  008A          movwf  0xA
0002  2804          goto   MAIN
0003  0000          nop
0004  0184  MAIN    clrf   0x4
0005  301F          movlw  0x1F
0006  0583          andwf  0x3
0007  3009          movlw  0x9
0008  00A1          movwf  0x21
0009  2809          goto   0x9
000A  0063          sleep
```



While Loop OK!

```
#device PIC16F877 *=8 ADC=8
void main()
{
        int c;
        c=9;
        do { } while (c++!=255);
}
```

```
0000  3000          movlw  0x0
0001  008A          movwf  0xA
0002  2804          goto   MAIN
0003  0000          nop
0004  0184  MAIN    clrf   0x4
0005  301F          movlw  0x1F
0006  0583          andwf  0x3
0007  3009          movlw  0x9
0008  00A1          movwf  0x21
0009  0821          movf   0x21,W
000A  0AA1          incf   0x21
000B  00F7          movwf  0x77
000C  0A77          incf   0x77,W
000D  1D03          btfss  0x3,0x2
000E  2809          goto   0x9
000F  0063          sleep
```

Addition/Subtraction

```
#device PIC16F877 *=8 ADC=8
void main()
{
        signed int c;
        signed int b;
        c=-4;
        b=c+2;
}
```

```
0000  3000         movlw  0x0
0001  008A         movwf  0xA
0002  2804         goto   MAIN
0003  0000         nop
0004  0184  MAIN   clrf   0x4
0005  301F         movlw  0x1F
0006  0583         andwf  0x3
0007  30FC         movlw  0xFC
0008  00A1         movwf  0x21
0009  3002         movlw  0x2
000A  0721         addwf  0x21,W
000B  00A2         movwf  0x22
```

CCS Compiler Integer multiplication

- Repeated addition
  - store result sign
  - convert both integers to unsigned
  - store one integer in W
  - For each bit in integer;
    - if set add W to the result
    - rotate result
  - if result sign is negative, convert the result

Multiplication & Division

goto vs. call?

```
#device PIC16F877 *=8 ADC=8
void main()
{
        signed int c;
        signed int b;
        c=-6;
        b=c*5;
        b=c/3;
}
```

```
0063  0184  MAIN   clrf   0x4
0064  301F         movlw  0x1F
0065  0583         andwf  0x3
0066  30FA         movlw  0xFA
0067  00A1         movwf  0x21
0068  0821         movf   0x21,W
0069  00A3         movwf  0x23
006A  3005         movlw  0x5
006B  00A4         movwf  0x24
006C  2804         goto   0x4
006D  0878         movf   0x78,W
006E  00A2         movwf  0x22
006F  0821         movf   0x21,W
0070  00A3         movwf  0x23
0071  3003         movlw  0x3
0072  00A4         movwf  0x24
0073  283B         goto   0x3B
0074  0878         movf   0x78,W
0075  00A2         movwf  0x22
0076  0063         sleep
```

CCS floating point example

```
#device PIC16F877 *=8 ADC=8
void main()
{
        float c;
        c=1.25;
}
```

```
0000  3000         movlw  0x0
0001  008A         movwf  0xA
0002  2804         goto   MAIN
0003  0000         nop
0004  0184  MAIN   clrf   0x4
0005  301F         movlw  0x1F
0006  0583         andwf  0x3
0007  307F         movlw  0x7F
0008  00A1         movwf  0x21
0009  3020         movlw  0x20
000A  00A2         movwf  0x22
000B  3000         movlw  0x0
000C  00A3         movwf  0x23
000D  00A4         movwf  0x24
000E  0063         sleep
```

Review Exercises

1. The code listed below was part of a program entered into the PIC16F877.

```
loop    incfsz  Var,F
        goto    loop
```

   The loop never terminates. Which of the explanations are plausible?

   (a) Var is set to a special function register. ☐
   (b) The initial value stored at location Var was not set correctly. ☐
   (c) The register (Var) cannot hold a number greater than 255. ☐
   (d) The register value is being changed by an interrupt. ☐
   (e) The status flags are being changed by an interrupt. ☐

2. Compiler generated assembly code, is sometimes larger (in both program and data space) than hand-generated assembly language code. Which of the following (if any) contribute to the inflated code size:

   (a) standard sizes of different variable types
   (b) robust translation of loop-testing conditions
   (c) failure to recognize(optimize for) supported assembly instruction-operations
   (d) placement of code/data in different pages/banks
   (e) entry and exit code (subroutine stack treatment)

3. You are writing a C program for the PIC16F877 which initialises locations in memory to the same default value. You are wondering which program would generate the most efficient assembly code when compiled. Your alternate choices are:

[A:]	`for (i=0;i<5;i++) c[i]=-1;`
[B:]	`for (i=4;i!=0;i--) c[i]=0;`
[C:]	`c[0]=0;c[1]=0;c[2]=0;c[3]=0;`

   Label the following statements TRUE/FALSE if they represent a valid/invalid reason for choosing the respective C code:

   (a) Code [A:] because we have an increment instruction, but no decrement instruction.
   (b) Code [C:] only a few locations need to be cleared, so it will take the least space when compiled.
   (c) Code [B:] or Code [C:] clearing a location is easily done with `clrf`.
   (d) Code [B:] testing for the ZERO flag is easily done with `btfsc STATUS,Z`.
   (e) Code [A:] or Code [B:] indirect addressing can be used in a loop.

4. The loop shown was encoded by the compiler as shown below.

```
void main()
{
    int cbuff[3]={1,2,3};
    int i;
    for (i=0;i<5;i++) cbuff[i]=0;
}
```

When the C program is compiled for the PC, it runs 5 times. When the C program is compiled for the PIC16F877, it runs continuously. The best explanation for this observation is that:

(a) Integers are represented differently by the two compilers.
(b) The code is being optimized for the PIC, and not for the PC.
(c) The compiled program is too big for the PIC16F877 program memory.
(d) The compiled program accesses too much data memory in the PIC16F877.
(e) The compiled program runs out of time on the PIC16F877.

5. Three sets of C code have been written to perform the multiplication of 255 and 27 using repeated addition.

5-I

```
void main()
{
    int k,M1,M2;
    long int R;

    R=0;
    M2=27;
    M1=255;
    for (k=0;k<=M1;k++)
    {
        R=R+M2;
    }
}
```

5-II

```
void main()
{
    int k,M1,M2;
    long int R;

    R=0;
    M2=27;
    M1=255;
    for (k=M1;k>0;k++)
    {
        R=R+M2;
    }
}
```

5-III

```
void main()
{
    int k,M1,M2;
    long int R;

    R=M2=27;
    M1=254;
    for (k=0;k<=M1;k++)
    {
        R=R+M2;
    }
}
```

For the compiler used, integers are represented using 8 bits, and long integers are represented using 16 bits. When run, Program 5-I never stops. Programs 5-II and 5-III, however, run as expected. The MOST PLAUSIBLE explanation for the failure of Program 5-I is that:

(a) R is not big enough to hold the result of the operation.
(b) R should also be declared as an integer.
(c) the number 255 cannot be represented using 8 bits.
(d) the termination test requires representing the number 256 in 8 bits.
(e) there is a flaw in the compiler – it generates bad code sometimes.

6. We wish to extract bits 6 and 7 of a byte called `Data`, and place them in bits 0 and 1 of the working register, respectively. The following code snippets have been suggested.

P-I	**P-II**	**P-III**

`swapf Data,F`
`nop`
`andlw 0x0C`
`rrf Data,F`
`rrf Data,W`
	`movlw 0xC0`
	`andwf Data,F`
	`swapf Data,F`
	`rrf Data,F`
	`rrf Data,W`
		`movlw 0xC0`
		`andwf Data,F`
		`bcf STATUS,C`
		`rlf Data,F`
		`rlf Data,F`
		`rlf Data,W`

(a) Which of the snippets will perform this task correctly?

i. P-I only
ii. P-I and P-II only
iii. P-I and P-III only
iv. P-II and P-III only
v. P-I, P-II and P-III

(b) Presuming that they are all correct, which snippet will take the shortest time to run?

i. P-I
ii. P-II
iii. P-III
iv. P-I and P-II will take the same (shorter) time
v. P-I, P-II and P-III will all take same time

(c) Presuming that they are all correct, these snippets could have been generated by a compiler from C program code. Which of the following tasks is MOST likely to be the task performed by the C program code?

i. clear a flag contained in a byte (Data)
ii. divide an unsigned number (Data) by 64
iii. extract a packed BCD number from a byte (Data)
iv. multiply a number (Data) by 8
v. retrieve an integer from a packed-array (Data)

7. All questions refer to the following C code, and generated assembly language for the PIC16F877:

```
void main()
{
    int c;
    c=9;
    do{}while (c++ <= 255);
}
```

```
0000  3000         movlw  0x0
0001  008A         movwf  0xA
0002  2804         goto   MAIN
0003  0000         nop
0004  0184  MAIN   clrf   0x4
0005  301F         movlw  0x1F
0006  0583         andwf  0x3
0007  3009         movlw  0x9
0008  00A1         movwf  0x21
0009  2809         goto   0x9
000A  0063         sleep
```

(a) What is the address of the variable c? ____________

(b) **Highlight** (or circle) statements which comprise loops in the C and assembly code.

(c) What do you expect the C code **loop** to do?

(d) What will the generated assembly code **loop** actually do?

(e) In your own words, explain why **this** assembly language code has been generated.

(f) Write your own equivalent assembly language for the C code.

8. The C code in Figure 8 on page 65 was compiled; the generated assembly language for the PIC16F877 is also shown:

   (a) What is the address of the variable cbuff?

   (b) What is the address of the variable i?

   (c) What is the address of the return value?

   (d) View/Add comments which explain the generated assembly code.

   (e) **Highlight** (or circle) statements which comprise the loops in the C and assembly code.

   (f) What do you expect the C code **loop** to do?

   (g) What will the generated assembly code **loop** actually do?

   (h) In your own words, explain why **this particular piece of** assembly language code has been generated from the C code.

   (i) Write your own equivalent assembly language code for the C code, in the space next to the code.

9. The C code shown in Figure 9 on page 66 was compiled; the generated assembly language for the PIC16F877 is also shown:

   (a) What is the address of the variable q? ________________

   (b) **Highlight** (or circle) statements which comprise loops in the C and assembly code.

   (c) What do you expect the C code **loop** to do?

   (d) Add comments which explain the generated assembly code. What will the generated assembly code **loop** actually do?

   (e) In your own words, explain why **this** assembly language code has been generated.

   (f) Would you expect similar assembly language code to be generated if we divided by 3 in each iteration of the loop? Explain your answer.

10. Role Play/Challenge: Find all possible ways in which a "for" loop test can fail when the loop variable is a floating point number.

Figure 8: Code sample and generated assembly for Question 8

```
#include <16f877.h>

int main()
{
    int cbuff[3]={1,2,3};
    int i;

    for (i=0;i<5;i++)   cbuff[i]=0;
    return 0;
}

0000  3000        movlw  0x0          ; return to Program Bank 0 (clear PCLATH)
0001  008A        movwf  0xA
0002  2804        goto   main
0003  0000        nop
0004  0184  main  clrf   0x4          ; clear FSR
0005  301F        movlw  0x1F
0006  0583        andwf  0x3          ; clear out all the banking bits in STATUS by using a mask
0007  3006        movlw  0x6
0008  1683        bsf    0x3,0x5      ; move to data memory bank 1
0009  009F        movwf  0x1F         ; configure ADCON1 so that shared pins are digital
000A  3001        movlw  0x1          ; first value for initialising the array
000B  1283        bcf    0x3,0x5      ; back to data memory bank 0
000C  00A1        movwf  0x21         ; start initialisation
000D  3002        movlw  0x2
000E  00A2        movwf  0x22
000F  3003        movlw  0x3
0010  00A3        movwf  0x23         ; all initialised
0011  01A4        clrf   0x24         ; for loop -- initial state i=0
0012  0824        movf   0x24,W       ; for loop -- tst <5: subtrt 4, tst Carry (borrow-bar) flag
0013  3C04        sublw  0x4
0014  1C03        btfss  0x3,0x0
0015  281C        goto   0x1C         ; leave the loop if we fail the test
0016  3021        movlw  0x21         ; put address of cbuff in W
0017  0724        addwf  0x24,W       ; add the value of i to get the address of cbuff[i]
0018  0084        movwf  0x4          ; place address in the FSR
0019  0180        clrf   0x0          ; clear the location indicated by FSR (cbuff[i])
001A  0AA4        incf   0x24         ; for loop -- increment
001B  2812        goto   0x12         ; go and re-test condition
001C  3000        movlw  0x0
001D  00F8        movwf  0x78         ; return value stored
001E  0063        sleep
```

Figure 9: Code sample and generated assembly for Question 9

```
#include<16F877.h>

int main()
{
    char q;
    int i;

    q=83;
    for (i=0;i<8;i++)
        q=q/4;
    return 0;
}
```

```
0000  3000        movlw  0x0
0001  008A        movwf  0xA
0002  2804        goto   main
0003  0000        nop
0004  0184  main  clrf   0x4
0005  301F        movlw  0x1F
0006  0583        andwf  0x3
0007  1683        bsf    0x3,0x5
0008  141F        bsf    0x1F,0x0
0009  149F        bsf    0x1F,0x1
000A  151F        bsf    0x1F,0x2
000B  119F        bcf    0x1F,0x3
000C  3053        movlw  0x53
000D  1283        bcf    0x3,0x5
000E  00A1        movwf  0x21
000F  01A2        clrf   0x22
0010  0822        movf   0x22,W
0011  3C07        sublw  0x7
0012  1C03        btfss  0x3,0x0
0013  281A        goto   0x1A
0014  0CA1        rrf    0x21
0015  0CA1        rrf    0x21
0016  303F        movlw  0x3F
0017  05A1        andwf  0x21
0018  0AA2        incf   0x22
0019  2810        goto   0x10
001A  3000        movlw  0x0
001B  00F8        movwf  0x78
001C  0063        sleep
```

Tutorial Exercise 6B Offline Version[31]

ID# ________

The C code shown overleaf was compiled and the generated assembly for the PIC16F877 is shown:

1. What addresses are used to store the variables q and i? ________ *2 marks*
2. **Highlight** (or circle) statements which comprise loops in the C and assembly code. *2 marks*
3. What do you expect the C code **loop** to do? *1 mark*
4. What will the generated assembly code **loop** actually do? *1 mark*
5. In your own words, explain why **this** assembly language code has been generated. *1 mark*
6. How would the code change if we incremented q in each iteration of the loop? Write changes next to the code overleaf. Explain your answer below. *3 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

[31]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

```
#include <16F877.H>
int main()
{
    long q;
    int i;

    q=26;
    for (i=0;i<4;i++)
        q=q*2;
    return 0;
}
```

```
0000  3000 MOVLW 0
0001  008A MOVWF 0xa
0002  2804 GOTO 0x4
0003  0000 NOP
0004  0184 CLRF 0x4
0005  301F MOVLW 0x1f
0006  0583 ANDWF 0x3, 0x1
0007  300F MOVLW 0xf
0008  1683 BSF 0x3, 0x5
0009  009F MOVWF 0x1f
000A  1283 BCF 0x3, 0x5
000B  01A2 CLRF 0x22
000C  301A MOVLW 0x1a
000D  00A1 MOVWF 0x21
000E  01A3 CLRF 0x23
000F  0823 MOVF 0x23, 0
0010  3C03 SUBLW 0x3
0011  1C03 BTFSS 0x3, 0
0012  2818 GOTO 0x18
0013  1003 BCF 0x3, 0
0014  0DA1 RLF 0x21, 0x1
0015  0DA2 RLF 0x22, 0x1
0016  0AA3 INCF 0x23, 0x1
0017  280F GOTO 0xf
0018  3000 MOVLW 0
0019  00F8 MOVWF 0x78
001A  0063 SLEEP
```

Unit 17 Write the letter you have been assigned here________.

Answer the following questions making reference to the two pieces of C code corresponding to your letter.

1. Identify one difference between the two pieces of code.

2. What will this difference mean when the code is compiled?

3. Choose one piece of code, and explain the circumstances under which this would be the better choice.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - give examples of the limitations placed on compiler-generated code, for the PIC16Cxxx series of microcontrollers, by the need for generality.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

A

```
void main()
{
   int b,c;
   b=5;
   c=4;
   c+=b;
}
```

```
void main()
{
   long int b,c;
   b=5;
   c=4;
   c+=b;
}
```

B

```
void main()
{
   int i,c[10];
   for (i=0;i<10;i++)
       c[i]=i*4;
}
```

```
void main()
{
   int i,c[10];
   for (i=9;i!=0;i--)
       c[i]=i*4;
   c[0]=0;
}
```

C

```
void main()
{
   int i,c[10];
   for (i=0;i<10;i++)
       c[i]=i*4;
}
```

```
void main()
{
   int i,c[10];
   for (i=0;i<10;i++)
       c[i]=i<<2;
}
```

D

```
void main()
{
   int i,c[10];
   for (i=0;i<10;i++)
       c[i]=i*2;
}
```

```
void main()
{
   int i,c[10];
   for (i=0;i!=10;i++)
       c[i]=i*2;
}
```

Algorithms on PIC16

In this section we have explored assembly code snippets for the PIC16F877, discussed some of the ways in which both assembly and C code can be manipulated to improve performance. In doing so, we have partially satisfied the second course objective; to "implement arithmetic and data manipulation operations in assembly language, and assess how well code will perform in a given context, given the architecture and instruction set of a microprocessor".

This represents the mid-way point in the course. You should fill out the mid-course review at this stage. After this, we will be paying more attention to the ways in which the software can be made to control or work with hardware devices. To facilitate this, the group mini-projects will be issued at this stage. Then the next section will deal with the primary means of external interaction, the I/O port and other peripherals.

Group Mini-project

Objectives

By the end of this project, your group will have achieved/performed the following objectives.

- made an appropriate sub-division of tasks within the group, and assessment of how well group members performed their assigned tasks
- consideration of an applied μP problem and production of an appropriate hardware/software design outline;
- interpretation of information presented on data sheets, and application of that information in microprocessor interfacing;
- development of assembly language code, which utilises at least 2 of the following:
  - Interrupts,
  - Timing,
  - A/D and/or D/A techniques,
  - Communications,
  - Peripherals: possibly to accomplish any of the above.
- identification and elimination of circuit and design problems using a prototype, standard lab equipment, and troubleshooting procedures.
- documentation of the final design/prototype, and identification of salient features/flaws.

Protocol/Guidelines

The entire class will be divided into groups, with each group consisting of 4-6 persons. The group will appoint a group leader. Each group will be required to attempt one of the specified projects. The group leader will "pick from a hat" to determine which project the group will perform. A preliminary package of parts/datasheets will be supplied to each group leader. Groups should utilise their group forum to communicate with each other about the project and their group wiki to update the class on their progress. Each group session, randomly selected group(s) will be called upon to present their work-to-date.

Please note:

- Group leaders will be responsible for obtaining any additional parts required, and for ensuring that parts are not damaged.
- The group is collectively responsible for making sure that all members participate in the exercise.
- Any group with damaged parts will be required to carry on using the damaged parts (therefore resulting in loss of marks in demonstration and implementation)

Peer Assessment

The mark which you will receive for this project will be calculated based on the group mark, and the peer assessment made by the rest of your group. The peer assessment form is on page V-74

$$r = \frac{nP}{\sum_{i=1}^{n} P_i}$$

$$final\_mark = \begin{cases} rG & ; r < 2 \\ 2G & ; r \geq 2 \end{cases}$$

where P is the peer assessment mark you received (between 1 and 5), n persons are in the group, G is the group mark and $final\_mark$ will be recorded. No-one will be allowed to receive more than twice the group mark.

Completed peer assessment forms should be returned to the technician, TA, or lecturer, anytime up to the Tuesday following the submission deadline. If you do not complete the peer assessment exercise, you will not receive any marks for the group project.

Demonstration

The final design prototype should be observed in operation by either the TA or lecturer, prior to submission (please arrange a mutually convenient time). If you make any changes to the operation of the system before submission, you will need to repeat this.

Presentation

On a selected day, during the regular lecture slot, a randomly chosen member of your group will be required to make a 1 minute presentation to the class about your project, and answer 2 minutes of questions. All group members should be present and prepared, as anyone may be called on the day.

Peer Assessment Form

Based on: SAMPLE PEER ASSESSMENT FRAMEWORK
`http://www.oaa.pdx.edu/CAE/FacultyFocus/spring96/bulman.html`

Assessment of others is an important skill. You should take time to complete this assessment form, forcing yourself to be objective and unbiased. **Your responses will be kept confidential.**

For each member of the group (except yourself), award a mark from 1 to 5 (or X if not applicable) for your group-mate's contribution/performance in each of the tasks identified. The marking scheme considers both the quality of your peers' work (hardware, software, report) and the cooperative-ness of your peers (participation).

For Participation, consider things like:

- Did this person attend scheduled meetings / keep appointments?
- Did this person meet all assigned deadlines?
- Did this person accept constructive criticism and act on it?
- Did this person contribute and share ideas?
- Did this person do a fair share of the more tedious work (i.e. photocopying, typing, formatting, testing, debugging etc.)?

Name & ID# ____ Group #______________

First Name	Surname	ID Number	Hardware	Software	Report	Participation

Please use the following scale:

1 - did not contribute to this task even though a contribution was expected

2 - willing, but not very successful

3 - average, did the basic work successfully but no special effort

4 - above average, willing, able, successful, and made a special effort

5 - outstanding: extra effort, and critical to success of task

X - was not asked to contribute to that task

Marks reported by your group-mates will be averaged and rounded to the nearest integer to give you an individual peer assessed mark (from 1 and 5).

Group Assessment

The group will be assessed on the following basis:

5 marks Presentation/Demonstration

5 marks Report

10 marks Design

10 marks Implementation

Up to four (4) extra bonus points may be awarded, at the discretion of the lecturer or TA. One (1) in each of the specified categories, for exceptional performance in that category.

The group report should be submitted in a manila folder. The group number should be marked on the tab of the manila folder. A disk or CD containing the *.hex, *.asm and *.lst files for the software should also be securely fixed inside the folder. One method: you seal the disk in an envelope, and staple the envelope to the folder.

Please write the course code, the coursework title, and your Group number on both the folder and the disk/CD. A list of the group members ID#'s should also appear on the report cover.

The following declaration should appear on the first page of the report, and MUST be signed by all group members.

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- oral or written discussion, clarification and/or modification of the project goals
- oral or written discussion of alternate solutions
- oral or written assistance in troubleshooting a particular solution.
- oral or written advice on report presentation/structure, and Technical English grammar/sentence construction.

Department of Electrical and Computer Engineering, UWI

PLAGIARISM Plagiarism is the presentation by a student-group of an assignment which has in fact been copied in whole or in part from another student-group's work, or from any other source (e.g. published books or periodicals), without due acknowledgement.

COLLUSION Collusion is the presentation by a student-group of an assignment as their own which is in fact the result in whole or part of unauthorised collaboration with persons who are not members of the student-group.

DECLARATION We declare that this assignment is our own work and does not involve plagiarism or collusion. We have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed:	Date:
Signed:	Date:
Signed:	Date:
Signed:	Date:
Signed:	Date:

(Department of Electrical and Computer Engineering)

Report Format

Details of the report format are shown on the following pages.

Part of the Report	Sub-Section	Content	Format	Pagination
The Front Matter	**General Information**	**Quotations** of more than two lines are single spaced and indented .5" from the left Margin. **Tables or Figures** should appear closely following the text where they are discussed. No further than a page following. Tables and Figures must have descriptive titles.	1.5 to double line spacing **Single line** spacing used in Quotations, Appendices, References **Margins-** top and bottom 1"; Left 2" and Right 1" **Typeface** – 10-12 Pitch fonts Preferably Serif font (TNR) for body text and Sans Serif font (Arial) for Headings Equations and Formulae must be typed, not hand written	Roman Numerals Lowercase **Page numbers** on Top or Bottom Right of the page **Tables and Figures** are numbered using Arabic numbering and separate sequences
	Title Page	Students' Name & ID Number Lecturer's Name Title of Report Date of Report	Centered Horizontally and Vertically on the Page and spaced evenly	No Page Numbering
	Abstract	An Informative Abstract is required, no longer than 2 paragraphs. This must include: objectives, scope, method, findings, recommendations and conclusions.	Single Line Spaced	Lower case Roman Numerals, Page starting from 2
	Table of Contents	Includes all significant parts of the report. Does NOT include the TOC	Generated from Document Headings	All front matter continues lower case Roman Numerals, numbered consecutively off the abstract
	Glossary (optional)			
	List of Abbreviations (optional)			
	List of Figures/Tables		Generated from Captions	

Part of the Report	Sub-Section	Content	Format	Pagination
The Body	Overview	Summarize the behavior and set the goals of the system.	Each section on a new page	New Sequence, Arabic numbering
	Hardware Layout and/or Circuit diagram	Block and circuit diagrams. Calculations, and component choices along with explanations. Please do not include specification sheets.		
	Software structure and/or Algorithm	State charts, Flow Charts, Use Case diagrams, Pseudo code (as appropriate)		
	Testing	A list of tests developed based on system requirements, and the recorded result of each test on the final system.		
	Discussion	Analysis of how well the objectives were met, details of design tradeoffs and **<u>significant</u>** problems encountered during development, suggestions for correction/improvement.		
The End Matter	**References**	Chicago Manual of Style 15th Ed. (http://www.mainlib.uwi.tt/divisions/eps/guides/epschicagocite.pdf)		Continue sequence of Arabic numbers
	Appendices - CODE	All submitted printout/programs MUST follow the specified format i.e. • labels in first column; • code in second column; • operands (where appropriate) in 3rd column; • comments (where appropriate) in the 4th column • directives in capital letters; • assembly language in common letters; • variable names/labels lower case (optional first capital letter).		

Description

The mini-project for ECNG2006 in 2007/8 will be related to a system for a remotely operated vehicle, suitable for exploring hazardous environments. The vehicle will require a number of sub-systems:

1. GP1: Direction Monitor (4)
2. GP2: Motor Control (3)
3. GP3: Voltage Monitor (5)
4. GP4: Remote Communication (4)

Students are expected to locate datasheets for the parts marked *****.

GP1 - Direction Monitor

Initial Requirements

- The compass direction should be obtained using I2C interface (compass address 0xC0)
- The LED's should be arranged so as to give a visual indication of the direction in which the compass is oriented: the user should be able to visually distinguish between the 8 directions: N, NE, E, SE, S, SW, W, NW.
- The user should be able to specify the desired direction using the keypad. The keypad keys should be appropriately labeled to facilitate this task.
- The system should beep at intervals - the interval between beeps should increase as the compass direction approaches the desired direction.

Parts

- breadboard, PIC/header, zener diode, crystal oscillator & decoupling capacitor

* Devantech magnetic compass mounted on CD w/cable

- 2 Yellow and 2 Red LED's
- 8 Ohm Loudspeaker

GP2 - Motor Control

Initial Requirements

- The DC Motor speed should be controlled using Pulse Width Modulation (PWM).
- The user should be able to select different DC motor speeds using the 4-bit DIP switch; at least one switch setting will stop the motor.
- A visual indication of the current motor speed (from 0 to 999 revs per minute), should be displayed on the common anode 7-segment displays.
- The motor speed should be stored in the serial EEPROM (address 0xA4) every 1 second after startup, until the EEPROM memory is full. After the EEPROM memory is full, stopping the motor should result in playback of the motor readings to the 7-segment displays.

Parts

- breadboard, PIC/header, zener diode, crystal oscillator & decoupling capacitor
- 4-DIP switch

* US Digital Encoder S1-1000-NT mounted on a DC Motor

- 3 common-anode 7-segment displays

* I2C Serial EEPROM 24C02

GP3 - Power monitor

Initial Requirements

- The Regulator should be driven with a variable voltage input between 8 and 12 Volts. It should independently power the joystick and the PIC16F877, so that the joystick can be connected/disconnected from the power source, using a signal from the PIC16F877.
- The joystick and a voltage monitoring line for the regulator input voltage, should be appropriately connected to different channels on the A/D converter. **Please ensure that you do not exceed the allowable input voltages for the PIC16F877.** The joystick should be disconnected from the regulator if the regulator input voltage drops below 9 volts.
- The joystick and 10 segment LED bar should be oriented so that the LED Bar is aligned with one of the joystick axes. Moving the joystick along the aligned axis should result in the pattern displayed on the 10-segment LED changing in the direction indicated.
- The battery voltage reading should be stored every 1 second after startup, using the serial EEPROM (address 0xA2) as a circular buffer. Dropping the regulator input voltage below 9 volts, should result in playback of the preceding 30 seconds of voltage readings to the 10 segment LED bar. Dropping the regulator input voltage below 8 V should result in the PIC16F877 going to 'sleep'. The PIC16F877 should be reset using a button press or any other feasible circuitry.

Parts

- breadboard, PIC/header, zener diode, crystal oscillator & decoupling capacitor
- Joystick

* 5 Volt Regulator

- 10 segment LED bar

* I2C Serial EEPROM 24C02

GP4 - Remote Communication

Requirements

- The PIC16F877 should be able to send ASCII strings to/receive ASCII strings from a PC using the 9-pin serial port and the level-shifting IC. The protocol settings to be used are 2400 baud, 8 data bits, 1 stop bit, no parity.
- The 7-segment displays should be used to display a 24-hour clock which increments at 1 minute intervals - e.g. 23:00 represents 11pm at night.
- The clock value can be reported at any time using a serial port command "GET". The PIC16F877 should return a string indicating the current time e.g. `Current time is nn:nn` where `nn:nn` is the current clock setting. The clock value can be changed at any time using a serial port command `SET nn:nn` where `nn:nn` is the new clock setting. Invalid commands/input times should be ignored.
- Initial access to the PIC16F877 should be controlled using a password, which the user is prompted for. The password should be stored in the PIC16F877 Data EEPROM memory in an encrypted form. The user should be able to change the password from the PC using a serial command.

Parts

- breadboard, PIC/header, zener diode, crystal oscillator & decoupling capacitor
- 4 common anode 7-segment displays

* RS232- voltage level shift IC (IN232) + 9-pin male connector to ribbon

Part VI
Interfacing Peripherals

There are several features of computers/microprocessors/microcontrollers which have not been formally explored. In particular the interfacing functions, which let the processor interact with the external environment, and typical peripherals including I/O Ports, Timers, Interrupts, A/D conversion, PWM and Serial communications.

The most generic peripheral is the I/O port. All other peripheral functions can be simulated using an I/O port and dedicated code. The use of dedicated peripherals allows the microprocessor to perform multiple functions simultaneously. One of the features that differentiates a microcontroller from a microprocessor is the presence of built-in I/O ports and peripherals. The following sections address I/O ports and peripherals in general, and discuss how they operate on the PIC16F877 in particular.

18 Interfacing Idioms

At the end of this unit the student will be able to:

> *identify alternate ways in which components may be interfaced with a microprocessor (i.e. common idioms for interacting with hardware): single bit (with debounce), matrix, map into memory space, etc.*

The registers which appear in the data memory space of the PIC16F877 may be roughly classified into 4 sub-groups(Predko 2001):

- processor registers (STATUS, PCL, PCLATH, FSR, INDF),
- variable memory,
- shared (shadowed) variable memory,
- i/o port, or peripheral hardware registers.

So far we have used the processor registers, and the variable memory (un-shared). Because these are all in Bank 0 we also did not need to do any bank switching. This will change when dealing with peripheral registers. Peripheral registers may be a different size to the processor data-word size, and may be of two types: control, and data. Although the registers may have different addresses, they work in tandem to affect peripheral behaviour.

Input/Output Ports

The most common/flexible peripheral is the I/O port, also referred to as the Parallel Interface Adaptor (PIA) or Parallel Port Interface (PPI). The parallel I/O port acts as a buffer for things on the bus. It can be configured for input or output (uses tri-state on external side) using the tri-state or data direction control register; in many cases bits/nibbles can be separately configured. In response to the peripheral data address, data from the bus is latched for conveyance to external devices, or external input is latched for placement on the bus. I/O ports may be either mapped into the memory space or they may exist in a separate I/O space. The most commonly used I/O port peripheral on the PC is the Intel 8255 (and variants), which provides two ports.

The PIC16F877 has five I/O ports PORTA thru PORTE. Each port has a direction register (also referred to as the tri-state register TRISx) associated with it.

Particular port pins may be internally configured to facilitate interfacing. Three facilities found in the PIC16F877 are the use of Schmitt trigger inputs (debouncing), internal pull-ups (allow unpowered switch connections), and open drain outputs (require external pull-up – will not power the connection). Some of the port pins are multiplexed with other peripheral inputs/outputs.

> PORTA is a 6-bit wide, bi-directional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a Hi-Impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin). – (PIC 2001)

PORTA is at address 0x05 and its direction register, TRISA, is at address 0x85, which means that PORTA is in Bank 0 and TRISA is in Bank 1. Therefore either indirect addressing or bank switching must be performed to access the direction register. The following example shows how PORTA can be initialized:

```
; assume that we are in Bank 0
    clrf    PORTA       ; Set the output latches of PORTA to zero
    bsf     STATUS,RP0  ; Switch to Bank 1
    movlw   0x0F        ; Value for data direction
    movwf   TRISA       ; set RA<3:0> as inputs and RA4 as output
    bcf     STATUS,RP0  ; switch back to Bank 0
```

Alternatively using indirect addressing:

```
    clrf    PORTA       ; Set the output latches of PORTA to zero
    movlw   TRISA       ; Setup for indirect addressing
    movwf   FSR         ; ...by moving address to FSR
    movlw   0x0F        ; Value for data direction
    movwf   INDF        ; set RA<3:0> as inputs and RA4 as output
```

All other ports may be initialised in a similar manner.

The BANKSEL pre-processor directive for the MPASM assembler, may be used as shorthand for bank switching. The direct addressing example listed above may be re-written as:

```
; assume that we are in Bank 0
    clrf    PORTA       ; Set the output latches of PORTA to zero
    BANKSEL TRISA   ; Switch to Bank 1
    movlw   0x0F        ; Value for data direction
    movwf   TRISA       ; set RA<3:0> as inputs and RA4 as output
    BANKSEL PORTA   ; switch back to Bank 0
```

PORTB is very similar to PORTA. It is an 8-bit wide bi-directional port with the pin directions being controlled by the TRISB register. Each of the PORTB pins can have a weak internal pull-up, which can all be turned on by a single control bit. This bit being the $\overline{\text{RBPU}}$ (OPTION_REG<7>) bit. The weak pull-up is automatically disabled when the port pin is configured as an output. The pull-ups are also disabled by a power-on reset. (PIC 2001)

PORTC, and PORTD are 8-bit wide, bi-directional ports, while PORTE is a 3 bit wide bi-directional port.

I/O Programming Considerations

Any instruction which writes, operates internally as a read followed by a write operation. For example, the bsf and bcf instructions first read a register into the CPU, execute the bit clear/set operation and write the result back to the register. Care must be taken when these instructions are applied to a port that is configured for both input and output.

A bsf operation on bit 5 of PORTB will cause all eight bits of PORTB to be read into the CPU. Then the bit set operation takes place on bit 5 and the result is written to the output latches. If another bit, say bit 0, of PORTB is defined as an input at this time, the input signal present on the pin itself would be read into the CPU and rewritten to the data latch of this particular pin, overwriting the previous content. As long as the pin stays in the input mode, no problem occurs. However when bit 0 is switched into output mode later on, the content of the data latch is unknown.

An example is shown in the code below.

```
; Initial port settings: PORTB<7:4>Input   PORTB<3:0> Output
; PORTB<7:6> have external pullups and are not connected
; to any other circuitry
;
;                              Port Latch   Port pins
;                              ----------   ---------
1       bcf  PORTB,7         ; 01pp pppp    11pp pppp
2       bcf  PORTB,6         ; 10pp pppp    11pp pppp
3       bsf  STATUS,RP0      ;
4       bcf  TRISB,7         ; 10pp pppp    11pp pppp
5       bcf  TRISB,6         ; 10pp pppp    10pp pppp
```

At line 2, it would seem that the port latch's high byte should be 00pp but when the instruction is executed, the port pins are read again, thus the bit clear acts on 11pp pppp.

The actual write to an I/O port happens at the end of an instruction cycle, whereas for reading, the data must be valid at the beginning of the instruction cycle. Therefore, care must be exercised if a write followed by a read operation is carried out on the same I/O port-pin. The sequence of instructions should be such to allow the pin voltage to stabilize (load dependent) before the next instruction which causes that file to be read into the CPU is executed. Otherwise, the previous state of that pin may be read into the CPU rather than the new state. When in doubt, it is better to separate these instructions with a NOP or another instruction which does not access this I/O port.

Simple I/O Interfacing

A single bit of an I/O port may be used to interface with a number of devices. Each bit of an I/O port typically corresponds to a single IC pin. Circuits connected to I/O port pins should be configured in accordance with the current sink/source limits of the IC. There are typically three limits: maximum current sunk/sourced by a) the pin b) all pins in the port c) all ports on the IC. As IC's can generally sink more current than they can source, "sink" configurations are preferable.

The simplest output we can visualise is an LED as an indicator. This will pass current in one direction only (diode), once the potential difference exceeds a known value. As a result of the passing current, the LED will light. An appropriately oriented LED in series with a resistor, can be tied between either 5V or ground, and an I/O port pin. The resistor value will depend on the current requirements of the LED and current limits of the I/O pin/port/IC. Depending on the orientation, the LED will light when the port bit is configured as an output, and either a 1 or 0(depending on the LED orientation, and tie voltage) is placed on the port bit.

Similarly, the simplest input we can visualise is a single switch. Again it can be configured so that it either pulls the output high, or grounds the input. The input port will read '1' when the voltage exceeds the low-high voltage threshold, and '0' when the voltage is below the high-low voltage threshold. When configured as an input, the IC will draw a minimal amount of current (high impedance), however a protective resistance/capacitance may be used to "debounce" inputs.

It follows that multiple inputs/outputs can be simultaneously controlled from a port: e.g. a 7 segment display can use pins of a 8 pin port, all configured as outputs. Where the number of pins available on the port is limited, inputs/outputs may be combined, or certain pins may be used to select the active device e.g. matrix keypads/displays.

Simple I/O Applications

- LEDs (single bit)
  - source/sink (check current!)
- 7 segment displays
  - common anode (sink)
  - common cathode (source)
  - multiple digits: cycle
- Switches (single bit)
  - source/sink (typical)
  - software debounce
  - hardware debounce (Schmitt Trigger)
- Matrix/Parallel bus
  - Read/write line
- Shared pin for I/O
  - DO NOT enable the external I/O device inputs when reading the external outputs.
  - Be careful of current/voltage requirements

Common Interfacing Idioms

- Registers on the PIC16F877. There are different sub-sets mapped into the same memory space.
  - processor registers,
    - STATUS, PCL, PCLATH, FSR, INDF
  - variable memory (Bank 0)
  - shared (shadowed) variable memory
  - peripheral hardware registers
- Parallel Interface Adapter(PIA) -- parallel i/o port (Intel 8255)
  - Register(latch) acts as a buffer
    - Take device off-bus
    - Replace parallel bus
    - May be smaller than data-word
  - can be configured for input or output using tri-state or data-direction register
    - Active high/low
    - Logic high/low
    - Vdd/Gnd
    - thresholds
  - input and output registers (latches) may have same address

Peripheral Device Characteristics

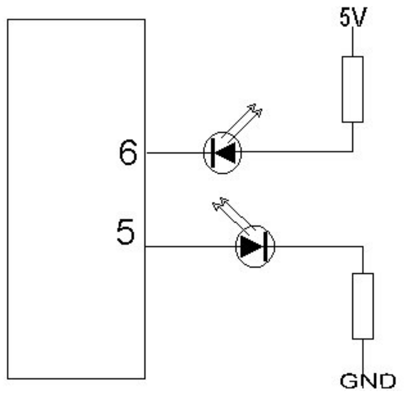
- internal latches/registers accessible by uP. There are 4 categories of registers/latches/flag bits:
  - **control** (telling the peripheral what to do),
  - **configuration**(telling the peripheral how to do it),
  - **state** (reporting what the peripheral is presently doing) and
  - **data** (containing the data the peripheral must send/receive to the microprocessor)
- indicate state to uP through the use of flags (internal or maintained by a peripheral controller), or output signals.
- (in conjunction with a Programmable Interrupt Controller) trigger maskable interrupts i.e. the interrupt signal may be individually enabled/disabled.

Peripherals on PIC16F877

- Parallel I/O port
  - PORTx -- data
  - TRISx -- configuration
- Timer0
  - TMR0 -- data
  - T0CS -- control/configuration (bit)
  - T0IE -- control (bit)
  - T0IF -- state

Review Exercises

1. PIC16F877 I/O ports have two associated registers `TRISx` and `PORTx`. Pins 5 and 6 of a particular port are connected to LED's as shown in the diagram.



   If the value in the TRISx register is 0x3F and the value in the PORTx register is 0x4F then:

   (a) the LED connected to port pin 5 is ON and the LED connected to port pin 6 is OFF.
   (b) the LED connected to port pin 5 is ON and the LED connected to port pin 6 is ON.
   (c) the LED connected to port pin 5 is OFF and the LED connected to port pin 6 is ON.
   (d) the LED connected to port pin 5 is OFF and the LED connected to port pin 6 is OFF.

2. A student interfaces an LED (Voltage drop 1.8V, Max. Current 5mA)to the PIC16F877. Which of the following is NOT an appropriate method of doing so:

   (a) tied to 5V through a 100 Ohm resistor
   (b) tied to ground through a 1 kilo-Ohm resistor.
   (c) tied to 5V through a 1 kilo-Ohm resistor.
   (d) tied to 3V through a 220 Ohm resistor.
   (e) tied to ground through a 2 kilo-Ohm resistor.

3. A normally open push button switch needs to be interfaced to the PIC16F877. Which one of the following methods (if any) is NOT a viable alternative:

   (a) tie the switch between PORTB pin and ground with internal pull-ups enabled; switch-off == pin-high
   (b) tie the switch between PORTB pin and ground with an external pull-up resistor; switch-off == pin-high
   (c) tie the switch between PORTB pin and V_{DD} with a pull-down resistor to ground; switch-off == pin-low
   (d) tie the switch between PORTB pin and V_{DD} with a pull-down resistor to ground; switch-off == pin-high
   (e) none of the above

4. We have a circuit in which a switch is connected to `PORTB<6>` and to ground. If `PORTB` pull-ups are enabled, all `PORTB` pins are configured as inputs, and no other `PORTB` pins are connected to circuitry, what value is placed in the working register by the instruction `movf PORTB,W` when the switch is ON?

   (a) 0xFF

   (b) 0xFE

   (c) 0x7F

   (d) 0xBF

   (e) 0xDF

5. The MPASM assembler has the BANKSEL preprocessor directive, which switches banks to the bank of the specified register.

   (a) Write code to configure PortB for all inputs using the BANKSEL directive.

   (b) Using the BANKSEL directive, write an assembly language routine which will
      - initialize `PORTB<3>` as an input, and `PORTB<6>` as an output.
      - within a loop: read `PORTB<3>` and set `PORTB<6>` to the value read.

   (c) Identify the pins associated with `PORTB<3>` and `PORTB<6>` on the PIC16F877 in 40 pin DIP package.

   (d) Draw circuitry to connect a simple switch to `PORTB<3>` and an LED to `PORTB<6>`, so that when your program from 5b is run, the LED will light when the switch is off, and vice versa.

6. (a) ”A certain [PIC16F877] system needs to be able to both activate eight LEDs and to read the state of eight normally-open (N.O.) push switches. It has been proposed that ... Port B might be able to combine these functions – the former when set to output and the latter when set to input. Can you devise a suitable circuit?” (Katzen 2003; Q. 11.4, p. 299)

   (b) For the circuit you devised, write a program which will repeatedly read the switches, and display the two’s complement value of the equivalent byte on the LED’s.

7. A dairy farmer has asked you to develop an ”intelligent” milk vat which can report the level, pH and viscosity of the liquid it contains, when it is remotely polled from a ”master” computer. Each reading should also be displayed as a histogram on a 10 bit LED bar display. You are given linear sensors which each generate a voltage between 0-5V for ranges of (0 - 100%full, pH 10 - pH 4, 0 - 50 mPas), and a PIC16F877 with a 4MHz oscillator.

   (a) Draw appropriate diagrams (and specify components and pin numbers where necessary) for circuitry such that:
      - The three sensors are connected to analog channels AN0, AN1, AN2.
      - The bar displays are individually turned on by transistors connected to `PORTB<3:1>`.
      - Bar display LED’s are switched on when the relevant output is high.
      - The upper two bits of the bar display are connected to `PORTB<5:4>` and the lower eight bits of the bar display are connected to `PORTC<7:0>`.

(b) Use diagrams to describe, and then write a subroutine called `send_port` which will place the upper 6 bits of the 10 bit value contained in registers `Val_hi<1:0>`, `Val_lo<7:0>` onto the Parallel Slave Port `<5:0>`, without altering bits `<7:6>` of the Parallel Slave Port.

(c) Use diagrams to describe, and then write a subroutine called `display` which will display the 10 bit value contained in registers `Val_hi<1:0>`, `Val_lo<7:0>` as a histogram or "moving bar" on the active LED bar (your subroutine does not need to activate the bar).

8. You are a member of a development team which is working on an automated parking system to help people locate their car in the supermarket parking lot.

Persons entering the car park will be given two battery powered wireless parking "cards": one to be left in the car, and one which they carry while they shop. The pair of cards must be returned on exiting the car park.

The cards each contain an array of DIP switches which are set to the same ID#.

The carry-card has a button, a circle of 8 LED's, a wireless transmitter, and an array of 4 directional wireless receivers. When the button is pressed, a query-message containing the ID# is transmitted.

The car-card has a single wireless transmitter/reciever. When the car-card receives the query message, it transmits a reply-message.

The carry-card receives reply-message(s) with varying strength on each of the directional receivers; if a reply-message with the correct ID# is received, the signal strength at the receiver is recorded. After waiting an appropriate time, the carry-card lights the LED closest to the receiver(s) which received the reply (i.e. in the direction of the car).

(a) Draw circuit(s) to show:

- how a normally-closed push-button can be interfaced to the PIC16F877 pin PORTB$< 0 >$ so that when the button is pressed, the PIC16F877 detects a high signal.
- how an LED can be interfaced to the PIC16F877 pin PORTD$< 0 >$, so that it lights when the PIC16F877 sends a low signal.

Your answers should include details of how you would configure the PIC16F877 port pins, and choose appropriate component values.

(b) The 8 LED's are connected to the pins of PORTD. In order to light a single LED, you need to write a byte with one bit set to PORTD.

Your team is discussing whether to generate this byte from the bit # by using a lookup table, or by rotating the number 0x01 the appropriate number of bits.

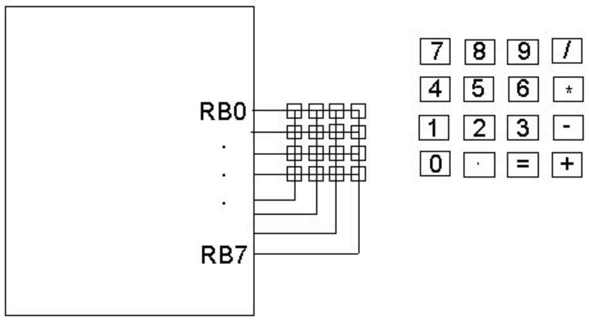
Write subroutines which will take the bit# in the working register, and return the appropriate byte in the working register, using each of these methods. Your code should return 0x00 if the number passed in the working register is not between 0x00 and 0x07.

(c) You are using a module independent of the PIC16F877 to detect the RF ID# for each receiver. Each of these modules has an output that will stay high for a time proportional to the signal strength received. The 4 module outputs are tied to PORTB$< 7 : 4 >$.

i. Describe using a diagram how you will determine which LED needs to be lit based on the outputs of the 4 modules.

ii. Implement this algorithm using polling on the PIC16F877.

9. You would like to interface a keypad to the PIC16F877 as shown in the figure below.



(a) Write a routine called `KeyInit` which will configure PORTB with pull-ups enabled, RB0:RB3 as inputs, RB4:RB7 as outputs, and clear PORTB. You should **`return`** at the end of your routine.

(b) After this initialisation routine has been executed:

  i. if no keys are pressed, what value will be transferred to the working register by the instruction `movf PORTB,W`.

  ii. if the '5' key is pressed, what value will be transferred to the working register by the instruction `movf PORTB,W`.

  iii. identify the three other keys that could be pressed to give the same answer as in 9(b)ii

(c) Presuming that internal pull-ups were not available, how could we interface a keypad to the I/O port, so that the keypad would function in the same way? Design an appropriate circuit showing one of the keypad buttons. Give adequate consideration to the voltage and current requirements/loads of the PIC16F877.

(d) The keypad columns must be individually activated in order to uniquely determine which key is pressed. Write a routine called `KeyScan` which loops through all four columns until a keypress is detected, and then returns with the value from PORTB in the working register.

(e) Draw up a table with the following headings in your exam booklet:

- Key Pressed
- ASCII value
- `KeyScan` return value

and fill it in for all 16 keys.
ASCII values: '*' 0x2A; '+' 0x2B; '-' 0x2D; '/' 0x2F; '.' 0x2E; '=' 0x3D; '0' to '9' 0x30 to 0x39

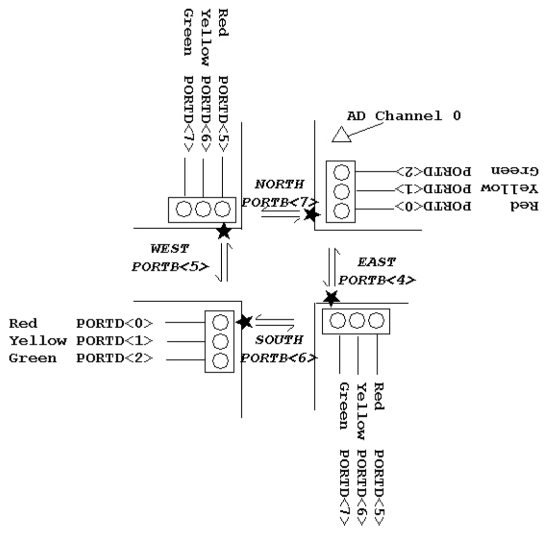
(f) Comment on how your routine `KeyScan` will work if two keys are pressed simultaneously.

(g) Suggest a suitable strategy for converting the `KeyScan` values to ASCII codes, and write a new routine `KeyConvert` which will convert the value in the working register, and return the converted value in the working register.

10. You have been asked to design a PIC16F877 based traffic light controller for an intersection.
Requirements

- There are 2 sets of traffic lights. One facing North/South and the other facing East/West. Each set consists of a pair of units. Each unit has a red, yellow, and green light. Each light requires 0.5A, and runs at 12V. Lights of the same color in a set are wired in parallel with each other.
- Six output lines (North/South `PORTD<7:5>`; East/West `PORTD<2:0>`) from the PIC16F877 are used to switch the lights on/off.
- The traffic lights are controlled by four reflective infrared sensors which detect waiting cars. The sensor deliver a $5V$ signal when a car is detected, and $0V$ when no car is detected. The sensors are connected to `PORTB<7:4>`.
- An analogue light sensor is connected to the A/D converter channel 0. which is configured for a left justified result. During the night, `ADRESH` falls below $0x80$.
- During the night, the traffic lights switch (in either direction) if a car has been waiting for more than one (1) minute, otherwise they remain green in the same direction.
- During the day, the lights are green for 5 minutes in one direction before switching.
- The sequence of lights is red, yellow (10 seconds), green, yellow (10 seconds), red.
- The traffic light controller also acts as a survey system counting the number of cars that pass each sensor. The four 16-bit counts are written to the data EEPROM, and then cleared, every half-hour.

Conceptual Design



```
main:      configure PORTD pins for outs (lights)
           configure A/D channel 0
           configure PORTB pins for ins (sensors)

loop:      if daytime
              for 5 minutes
                  for each sensor
                      if change from 0 to 1,
                          increment car count
              switch lights
           else
               for each sensor
                      if change from 0 to 1,
                          increment car count
               if waiting more than 1 minute
                  switch lights
           if 30 minutes have passed
              write counts to EEPROM
           goto loop
```

(a) Design a circuit to interface a PIC16F877 to a pair of traffic lights (e.g. Red on North/South to `PORTD<0>`). You should explain/show calculations for component values.

(b) Write a routine named `LgtSwitch` that will switch the lights. You should return at the end of your routine.

(c) i. Draw a diagram to explain how to increment a 16 bit integer number, stored in successive bytes, using indirect addressing.

ii. Suggest an alternate method of storing counts that would require less than 16 bits. What are the disadvantages of your method?

(d) Write a routine named `CntStore` which stores the four 16-bit counts in the data EEPROM. You should return at the end of your routine.

(e) Write an instruction-based delay routine named `dlym10` which will supply a delay, in multiples of 10 seconds, based on the value in the Working register (e.g. calling this routine with a value of 2 in the working register will supply a 20 second delay).You should return at the end of your routine.

Your answer should explain how you determined/verified all values, presuming that the PIC16F877 is being driven by a 4MHz clock signal.

(f) Write a routine named `RdAD` which will start, wait for, then read the A/D converter Channel 0, and return the value of ADRESH in the working register.You should return at the end of your routine.

(g) Identify and explain one way in which the counter/timer peripheral could be used in this application.

(h) Criticise the system you have just produced, and subsequently suggest at least two improvements.

(i) Write an assembly language program for the PIC16F877 which performs the algorithm provided. Your program should indicate where the preceding routines `RdAD`, `CntStore`, `dlym10`, `LgtSwitch`, are to be placed, but you do NOT need to rewrite them.

You should also indicate where the configuration routine for the ports and A/D converter `Cfg` should be placed, but you do NOT need to write this routine.

You should also indicate where the ISR, and handler for PORTB should be placed, but you do NOT need to write them.

Your main program should use appropriate preprocessor directives.

Tutorial Exercise 7AOffline Version[32]

ID# ______

The Bank of Nowhere has recently installed a Passive Infra-Red sensor at the doorway to their bank vault. The sensor generates a 5V signal when a person is standing in front of the doorway, and 0V when no-one is at the doorway. The bank wishes to use a PIC16F877 to ring a bell for as long as someone is standing in the doorway. The sensor is connected to `PORTB<0>`. The bell is connected to `PORTC<2>` via some interface circuitry. The bell rings when it receives a signal greater than 3V, and draws a peak current of 100mA.

1. Draw an appropriate circuit to interface the bell to the PIC16F877 pin `PORTC<2>`. Explain why you chose/designed this circuit. *4 marks*

2. For your circuit, write a routine called `RingChg` which will toggle `PORTC<2>`. Your code should be appropriately commented. *3 marks*

3. Write a code snippet which will repeatedly reads the pin `PORTB<0>` and call `RingChg` ONLY when a person enters/leaves the doorway. Your code should be appropriately commented. *3 marks*

[32] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 18 Write the letter you have been assigned here________.

A: Count the number of times a single pole double throw switch is closed.

B: Turn an output pin on/off when a non-latching push-button is pressed.

C: LED flashes at a constant frequency

D: Speaker beeps at a constant frequency

For the task which you have been assigned:

1. Draw the circuitry you would use

2. Describe how the software would work (flow charts or code snippets may be used)

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - identify alternate ways in which components may be interfaced with a microprocessor (i.e. common idioms for interacting with hardware): single bit (with debounce), matrix, map into memory space, etc.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

19 Interrupt Basics

At the end of this unit the student will be able to:

> *explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.*

Computer systems do not exist in isolation but are usually interfaced to the external world via I/O devices. The problem with I/O devices is that they are usually much slower than the computer system, therefore it is necessary to synchronize the interactions between the I/O devices and the computer system. With an input device, such as a keyboard, there will be times when the CPU has to wait for new data to be ready. Only when the data is ready, can it be read by the CPU and processed. If the time to create new data is longer than the processing time then the system is said to be *I/O bound*. If the data is ready before the CPU is ready, then the system is *CPU bound*. Systems can be *unbuffered* where the I/O writes/reads directly to the CPU or *buffered* where a buffer or memory storage exists between the I/O device and the CPU. There are several ways through which synchronization can be accomplished.

- Blind cycle
- Busy polling
- Interrupts
- Direct Memory Access (DMA)

Blind Cycle

This type of synchronization is appropriate when the delay due to I/O is fixed and known. It is called blind because it provides no feedback from the I/O device back to the computer. For example, if a printer can print at 40 characters/sec but cannot inform the computer when the last character has been printed. Figure 10 shows a diagram of how this synchronization is achieved. Blind cycle synchronization is simple and predictable, however if the output/input rate is variable

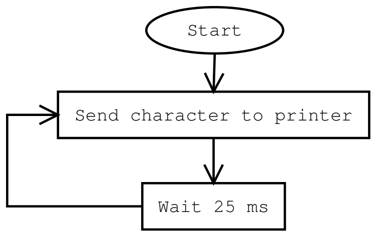


Figure 10: Blind cycle flowchart for a printer

or unknown then this method cannot be used. Computing time is wasted while the CPU is waiting and, because no feedback is available, error checking cannot be performed nor can special conditions be handled.

Busy polling

With this method of synchronization, the I/O device has a status bit that can be checked by the CPU to determine if data transfer can occur. Figure 11 shows how this type of synchronization is achieved. Busy polling can accommodate variable or unknown output/input rates, however the

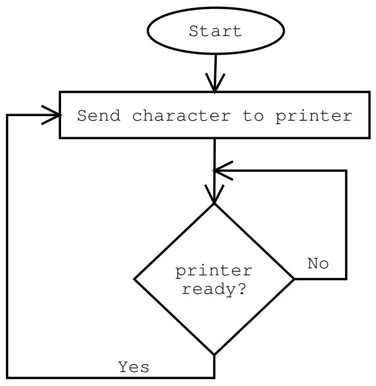


Figure 11: Busy wait flowchart for a printer

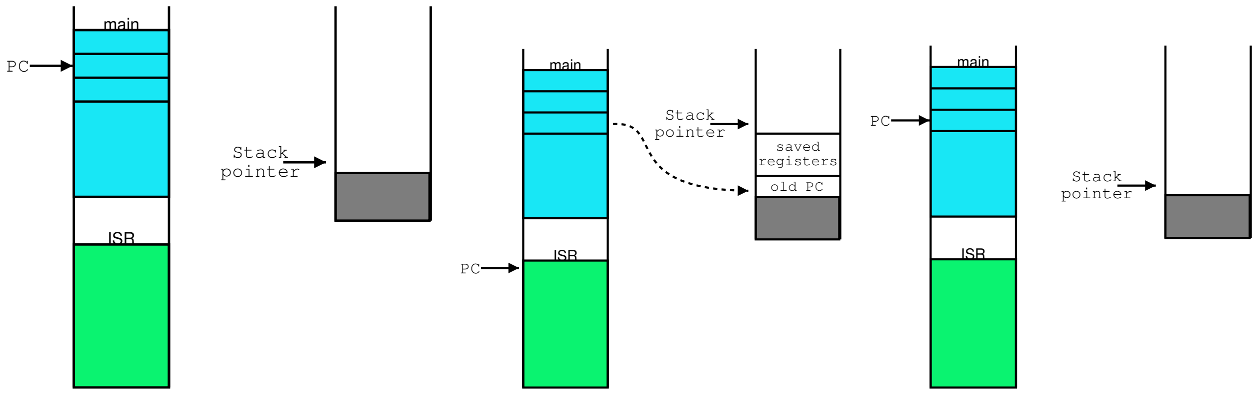
CPU still wastes time waiting for the I/O device to become ready. Therefore, the CPU is unable to do any other task while polling is taking place. A better method is to have the external device signal the CPU when servicing is required i.e. interrupts.

Direct Memory Access (DMA)

All the interfaces, seen so far, between the I/O device and the CPU have been controlled by software. In this type of control, if it is required to transfer data from an input device to RAM, the data must first be transferred to the CPU registers, then from the registers to RAM. Performance can be greatly improved if the data can be transferred directly between I/O devices and RAM. Direct Memory Access (DMA)allows this type of direct transfer without CPU intervention. DMA can be read or write and in both cases, the CPU is halted and the data transferred between memory and I/O device. The PIC16F877 does not support DMA.

Interrupts

Interrupts are a way of signaling the CPU in an asynchronous fashion that servicing is required. When an interrupt occurs the CPU finishes execution of its current instruction, saves any needed registers and the program counter (PC) on the stack and transfers execution to the Interrupt Service Routine (ISR). The CPU then executes the ISR and when finished, restores the PC and continues execution of the main program. This sequence is shown in Figure 12. The CPU is only interrupted when a service request is received, so no waiting is takes place and useful work can be done between interrupts. Interrupts can be *vectored* which means that each interrupt has a unique interrupt vector, and therefore a separate ISR. With vectored interrupts, the CPU will automatically execute the appropriate ISR when an interrupt occurs and there is no need to determine which device generated the service request. PC's have a standard interrupt vector structure. An interrupt subroutine may be ”installed” by changing the instruction at the appropriate vector. The PIC16F877 does not



(a) Before interrupt (b) During interrupt handling (c) After interrupt handling

Figure 12: Main memory and stack when an interrupt occurs

support multiple vectored interrupts, but the same effect can be achieved by *polling* the interrupt sources to determine who raised the interrupt and then executing an appropriate sub-routine.

When multiple interrupts occur simultaneously, the CPU must have some way of determining which interrupt to service first, i.e. which interrupt has *priority*. Further most CPU's have a single interrupt line, which is shared by all the interrupt sources. To implement hardware priority ordering, a peripheral interrupt controller (PIC) may be used. This is in turn a peripheral, which accepts all the hardware interrupt signals, and asserts the CPU interrupt line iff an *enabled* signal is received. The PIC control registers are used enable signals, and set their relative priorities. The PIC data registers report which of the many signals have been exerted, and the location of the ISR vector. If a PIC is not used (or does not support priority ordering) then the priority of the interrupt is determined by the order in which the interrupt sources are polled in the ISR, and the sequence in which the interrupts are enabled/disabled.

The interrupts which we have discussed, are all triggered by a hardware signal. It is also possible to trigger an interrupt handler from software (typically using a special interrupt instruction). In essence, this is the same as a subroutine call. Typically hardware interrupts are generated by peripheral reporting task completion or requesting CPU service. Software interrupts may be used to access BIOS (and other low level) functions, as well as to allow programs to access shared memory and peripherals in a structured or "known" fashion.

Software interrupts are not supported by the PIC16F877.

The PIC16F877 family has 14 sources of interrupts, including Timer Overflow, and signals/changes on Input Ports. The interrupt control register (INTCON) records individual interrupt requests in flag bits (Note that the flag bits will be set whether or not the interrupt is enabled). It also has individual and global interrupt enable bits. A global interrupt enable bit, GIE (`INTCON<7>`) enables (if set) all unmasked interrupts, or disables (if cleared) all interrupts. When bit GIE is enabled, and an interrupts flag bit and mask bit are set, the interrupt will vector immediately.

Individual interrupts can be disabled through their corresponding enable bits in various registers. Individual interrupt bits are set, regardless of the status of the GIE bit. The GIE bit is cleared on RESET. The "return from interrupt" instruction, RETFIE, exits the interrupt routine, as well as sets the GIE bit, which re-enables interrupts.

The RB0/INT pin interrupt, the RB port change interrupt, and the TMR0 overflow interrupt flags are contained in the INTCON register. The peripheral interrupt flags are contained in the special function registers, PIR1 and PIR2. The corresponding interrupt enable bits are contained in special function registers, PIE1 and PIE2, and the peripheral interrupt enable bit is contained in special function register INTCON.

When an interrupt is responded to, the GIE bit is cleared to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with 0004h. Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits.

It is the responsibility of the ISR to clear the interrupt flag bit corresponding to the interrupt that caused the ISR to be executed, otherwise the CPU would assume that another interrupt of the same type had occurred during execution of the ISR. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid recursive interrupts.

For external interrupt events, such as the INT pin or PORTB change interrupt, the interrupt latency will be three or four instruction cycles. The exact latency depends on when the interrupt event occurs. The latency is the same for one or two-cycle instructions. Individual interrupt flag bits are set, regardless of the status of their corresponding mask bit, PEIE bit, or GIE bit.

If interrupt occurs during the execution of the ISR it is not serviced until after the ISR completes and the GIE bit re-enabled. At this point, if the postponed interrupt's enable bit is set, it is serviced. – (PIC 2001)

Timing & Timers

One use to which the PIC can be put to is to control the times when events happen. A typical example is reading a port or A/D at specified intervals. If polling or blind cycle synchronization is used then what is required is some means of generating a delay to control the loop that reads the port or A/D.

There are several ways in which fixed delay times can be obtained. One of the most straightforward is by using the times taken by instructions to create the delay. Another is to use the built-in timer facility available.

It can be assumed for the rest of this discussion that a 10 ms delay time is required. If the clock frequency is 4 Mhz, then the number of cycles required is 10,000[33]. Let us see how to generate smaller delays, for example 1 ms. This can be done by counting down a value in the W register until it becomes zero. A first pass at the delay routine can be:

```
delay_1ms   movlw   <unknown>
:Loop4      addlw   0xFF     ; add -1, cannot use decrement
            btfss   STATUS,Z
            goto    :Loop4
            return
```

[33]The cycle frequency is the 1/4 the clock frequency

Peripherals on PIC16F877

- Parallel I/O port
  - PORTx -- data
  - TRISx -- configuration
- Timer0
  - TMR0 -- data
  - T0CS -- control/configuration (bit)
  - T0IE -- control (bit)
  - T0IF -- state

Using Peripheral registers/flags

- Four categories
  - control
  - configuration
  - state
  - data
- Use once

```
action:
    configure (maybe once)
    set control
    wait for state (optional)
    read/write data
    wait for state (optional)
```

- Polling
  - blind (keep looping)

```
repeat forever
    action
    delay (optional)
```

  - busy (loop using state)

```
repeat forever
    do something
    if state ok
        action
```

Timing/Delay

- Count the instructions (know the clock frequency)

```
delay_1ms     movlw 0xF9      ; 0xF9 = 249
              nop
usec4         addlw 0xFF      ; add -1
              btfss STATUS,Z
              goto usec4
              return
```

- Use the timers
  - Watchdog
    - nominal time-out period of 18 ms. Period varies with temperature, VDD
    - Scale up to 1:128 (max. 2.3 seconds)
    - reset/wake-up on timeout
  - Timer0/1/2
    - period determined by external oscillators or the instruction cycle
    - pre/post scaling
    - poll for timeout/interrupt on timeout

Software Interfacing

- CPU vs. I/O Bound
- Buffered vs. Unbuffered
- Synchronization
  - Blind cycle ★
  - Busy polling ★
  - Interrupts ★
    - vectored
  - Direct Memory Access (DMA)
- I/O Modules/Ports
  - isolated
  - memory mapped ★

★ Supported by PIC16F877

Now to count how long it takes to do the `:Loop4` loop. The `addlw` instruction takes 1 cycle, the `btfss` takes 1 and the `goto` takes 2. So a single traversal of the loop takes 4 cycles. The last traversal of the loop will take one more cycle when the bit test jumps to the `return`. Since it is only one cycle in a millisecond, the error is small. The final part is the setting of the `<unknown>` value.

The call to the delay routine will take 2 cycles and the loading of W will take another cycle, and since the loop takes 4 cycles, we need to round out the overhead to 4 cycles and compensate for it in the initial value of W.

The final routine that generates 1 ms delay is show below

```
delay_1ms   movlw   0xF9     ; 0xF9 = 249
            nop
usec4       addlw   0xFF     ; add -1, cannot use decrement, why?
            btfss   STATUS,Z
            goto    usec4
            return
```

This routine can be expanded through the use of a variable to allow times in multiples of 1 ms. For example

```
; routine is passes the amount of ms to delay in W ; it uses an
additional register variable msec_count

delay_ms    movwf   msec_count
mseccloop   movlw   0xF8     ; 0xF8 = 248
            call    usec4    ; 248 * 4 + 2 (call) = 994
            nop
            nop
            decfsz  msec_count,F
            goto    msecloop
            return
```

The overhead for the call to `delay_ms` is neglected as well saving W in `msec_count`.

Go through the routine carefully making sure that you understand its operation.

Using built-in timers

The Timer 0 of the PIC will generate an interrupt, if enabled, when an overflow occurs. This can be an advantage over the previous type of delay because the CPU can be doing other useful work during the waiting. In non-time critical applications, the use of ISR's are not necessary and simple polling will suffice. It is fairly straightforward to generate small delays by initializing Timer 0 to specific values and letting it timeout while continuously checking for the timeout. Normally Timer 0 runs continuously whenever the PIC is turned on, but there may be times when the user may want to set the time when it starts. This can be done by ensuring that the RA4/T0CK1 pin is grounded and setting Timer 0 in counter mode. Since the input to the counter is zero, no counting will take place and initialization of Timer 0 can take place without timing being started. When timing is required Timer 0 can then be switched to timer mode.

Simple timing (no interrupts)

The following section of code illustrates how simple timing can be done.

```
Init    bsf     STATUS,RP0      ; bank 1
        movlw   B'00100110'     ; set in counter mode, prescaler = 128
        movwf   OPTION_REG
        bcf     STATUS,RP0      ; back to bank 0
        bcf     INTCON,T0IF     ; clear to be sure
        .
        call    Delay
        .
Delay   movlw   106             ; time for 256 - 106 = 250 cycles
        movwf   TMR0
        bsf     STATUS,RP0      ; switch to bank 1 to access OPTION_REG
        bcf     OPTION_REG,T0CS ; start timing
        bcf     STATUS,RP0
Loop    btfss   INTCON,T0IF     ; keep checking for overflow
        goto    Loop
        bcf     INTCON,T0IF     ; clear flag
        return
```

Writing an Interrupt Service Routine (ISR)

The PIC16F877 has only one interrupt vector, located at 0x004, and as a result, polling must be done at the start of the ISR to determine which interrupt source generated the interrupt. Once the source is determined the appropriate handler can be executed. Within the handler, the interrupt flag bit must be cleared before re-enabling interrupts. Using the template given previously, an ISR designed to handle Timer 0, external and Port B change interrupts looks like the following.

```
ISR     movwf   w_temp       ; save W and STATUS
        movf    STATUS,W     ;
        movwf   status_temp  ;
Poll    btfsc   INTCON,T0IF  ; test if TMR0 overflow occurred
        call    Timer_hndlr  ; call handler for TMR0
        btfsc   INTCON,INTF  ; test if external interrupt occurred
        call    Extern_hndlr ; call handler for external interrupt
        btfsc   INTCON,RBIF  ; test if PORTB change interrupt occurred
        call    Change_hndlr ; ...etc
ISRDne  movf    status_temp,W ; restore pre-isr STATUS register
        movwf   STATUS        ;
        swapf   w_temp,F      ; restore pre-isr W register contents
        swapf   w_temp,W      ; ... without affecting the zero flag
        retfie                ; return from interrupt

Timer_hndler
        bcf     INTCON,T0IF   ; clear the overflow flag
        ; do processing
        return
Extern_hndlr
        ; clear appropriate flag etc
        return
Change_hndlr
        ; clear appropriate flag etc
        return
```

Simple timing (with interrupts)

If a 10 ms delay is required then there is no combination of TMR0 and prescaler that will give that time repeatedly so other methods must be devised. It can be seen that using a prescale value of 8, Timer 0 will overflow every 2.048 ms, and counting this five times will give 10.24 ms or approximately 10 ms. The following routine illustrates how this can be done.

```
        COUNT   EQU     0x05
        clrf    cntr
        .
Delay   btfss   INTCON,T0IF
        goto    Delay
        bcf     INTCON,T0IF
        incf    cntr,F
        movlw   COUNT
        xorwf   cntr,W
        btfss   STATUS,Z
        goto    Delay
        clrf    cntr
        return
```

Instead of polling the T0IF flag, an ISR can be used that is triggered by the overflow interrupt, in addition the counter, `cntr` is changed by the ISR. So the routine can now be broken into two parts and also can be made simpler by doing the check for the five times overflow of TMR0 by decrementing instead of incrementing `cntr`.

```
COUNT   equ     0x05 ; ISR
        ; initialization of ISR etc
Poll    btfsc   INTCON,T0IF
        goto    Timer
        btfsc   INTCON,.
        .
Timer   decf    cntr
        bcf     INTCON,T0IF
        goto    Poll

        ; end of ISR
        ; Main routine
Init    movlw   COUNT
        movwf   cntr
        .
Delay   btfss   cntr,7
        goto    Delay
        movlw   COUNT
        movwf   cntr
        return
```

More precise timing

The time delay achieved in the preceding section is only an approximation to the desired value of 10 ms which means that TMR0 must overflow every 2 ms. Now the total count in TMR0 with a prescaler of 8 is $256 \times 8 = 2048$ and $2000 = 256 \times 8 - 6 \times 8$. Therefore if TMR0 is incremented by 6 every time it overflows then the cycle time will be 2 ms. This can be done in the `Timer` handler of the ISR, for example,

```
Offset  equ     6

Timer   movlw   Offset
        addwf   TMR0,F
        decf    cntr
        bcf     INTCON,T0IF
        goto    Poll
```

Note that whenever a write is done to TMR0, it waits for two cycles to re-synchronize before counting starts again. In this case the wait is not significant and the error in 10 ms is only 0.1%.

Review Exercises

1. Label the following statements about interrupts True/False:
   (a) Interrupts may be triggered either by an electrical signal, or a special software instruction.
   (b) Before handling an interrupt, all processor registers which may be affected by the handler must be saved, so they can be restored later.
   (c) When an interrupt occurs the instruction which is being processed is stopped, and re-executed after the interrupt has been handled.
   (d) It is possible for an interrupt handler to be interrupted by another interrupt.
   (e) Interrupt prioritisation may be achieved either by using a dedicated peripheral, or by checking the interrupt sources in a pre-determined order.

Interrupts vs. Polling

Initialise -- **configuration**	Initialise -- **configuration**
Initialise -- **flags**	
Initialise -- **mask**	
Command -- **control**	
Repeat	Repeat
Job	**Command -- **control**
	Do
	Job
	Until -- **state**
	Read -- **data
ISR:	
save context	
Command -- **control	
Read -- **data	
Clear -- **flags	
restore context	

Software Interrupts

- Trigger does not come from an external device – it is an instruction (which may indicate the appropriate vector).
- These are used to:
  - facilitate BIOS and other low level functions
  - allow programs to access memory and peripherals in a structured "known" fashion

Prioritizing hardware interrupts

- Priority in software handler
  - Ordering of the handler calls
  - Disabling/Enabling other triggers
- Priority using an external peripheral: Peripheral interrupt controller (PIC)
  - Only allows one interrupt to be serviced at a time.
  - Priority is either determined by the connection position or can be programmed using a peripheral register.

Typical applications

- Hardware Interrupts
  - Peripheral reporting completion
    - A/D conversion done
    - Interval expired
    - DMA request complete
  - Peripheral requesting service
    - Key-press
    - Clock tick
- Software Interrupts
  - Time triggered actions
    - System clock update
  - Protected mode functions
    - BIOS function call (joystick read)

2. All PIC16F877 ISR routines must save the working register before anything else is done. This is because:

   (a) using the `retfie` instruction will corrupt the working register.

   (b) after 10 instruction cycles without use, the Working register will lose it's value.

   (c) the ISR routine is likely to change the Working register, and the main program needs the original value.

   (d) we can't poll the interrupt sources unless the Working register is empty.

   (e) none of the above

3. Choose the word combination which best completes the following sentence(s):

   The interrupt ____3I____ is the address of the instruction executed when the interrupt occurs. Interrupts are typically used when the ____3II____ needs to communicate information to the ____3III____ .

   (a) 3I: service routine; 3II:CPU ; 3III:peripheral

   (b) 3I: vector; 3II:CPU ; 3III:peripheral

   (c) 3I: service routine; 3II:peripheral; 3III:CPU

   (d) 3I: vector; 3II:peripheral; 3III:CPU

   (e) 3I: handler; 3II:peripheral; 3III:CPU

4. (a) In your own words, explain what happens when an interrupt occurs, and when the main program resumes.

   (b) In your own words, explain what is meant by saving and restoring the ”context”, and why these processes are needed in interrupt procedures.

5. (a) Differentiate between hardware and software interrupts.

   (b) Which is preferable: software or hardware prioritisation of interrupts? Justify your answer.

   (c) Student A tells Student B that hardware interrupts are interrupts that are prioritised using hardware. Is student A correct? Explain your answer.

6. (a) What is the reason for having the interrupt sources in a particular order?

   (b) When the ISR is entered in response to one specific interrupt, is it possible that a different source might be serviced first? Discuss your answer.

   (c) If while one interrupt source is being serviced, a second interrupt source requests service, will the second source be serviced before the `retfie` instruction at the end of the ISR? Discuss your answer.

7. Identify one source of interrupts on the PIC 16F877 (apart from the timers), and locate the flag and enable bits for your chosen interrupt source.

8. "The speed of a rotating shaft can be measured by using a coded disk to generate a pulse on each angular advance of 10°, which can be used to interrupt a PIC. If the top speed is 20,000 revolutions per minute, what is the absolute maximum duration of the ISR in this worst case situation to avoid missing pulses? You may assume a crystal frequency of 4MHz. "(Katzen 2003; Q7.4 p.193)

   Where code is requested, it is sufficient to simply include the relevant snippets/fragments. Use comments to indicate any assumptions you make about the rest of the code.

9. In a typical ISR, part of the initialization saves the W and STATUS registers, however this assumes that the main code that will be interrupted will never be in Bank 1 at the time of the interrupt. If the ISR ever switches banks then the values of W and STATUS that are restored would be incorrect[34]. In processors where the Bank 0 addresses are different from the Bank 1 addresses several things can be done to relax this assumption.

   - Interrupts can be disabled before switching to Bank 1 and re-enabled after switching back to Bank 0.
   - Use two addresses, one in each bank, to hold the temporary value of W, i.e. `W_TEMP`. For example,

     ```
     W_TEMP  equ    0x20
     W_TEMP_ equ    0xA0
     ```

     Now when W is stored in `W_TEMP` using direct addressing at the beginning of the ISR it will go in either of two locations: 0x20 or 0xA0, depending on which bank is active at the moment the interrupt occurs.

   (a) Rewrite the initialization of the ISR that saves W and STATUS in this case, switching to Bank 0 for the duration of the ISR. There would be need only for one location for `STATUS_TEMP` defined in Bank 0.

   (b) Rewrite the ending of the ISR to restore the STATUS and W.

10. You are responsible for writing an application for a PIC16F877 with an 8MHz external clock signal, which reads 8-bit information from a device. The device is configured so that the most recent byte appears on PortB every 40ms, and stays there until the next byte is available. When the byte changes, the device raises the READY signal line for 5ms and then drops it.

    (a) **In your own words**, explain the difference(s) between using blind cycle polling, busy-wait polling, and interrupts to accomplish your task.

    (b) **In your own words**, explain how indirect addressing may be used to implement a circular buffer.

    (c) For each of the three approaches(blind cycle polling, busy-wait polling, and interrupts)
      - explain where the READY signal line needs to be connected,
      - write a program for the PIC16F877 which will store the 8 most recently read bytes in a circular buffer,
      - determine the number of program memory locations needed to store the code and the number of data memory or register locations required to store constants/variables.

[34]In the PIC16F8x, Bank 1's General purpose registers are mapped to Bank 0, so that this argument does not hold

Tutorial Exercise 7BOffline Version[35]

ID# ____________

The Bank of Nowhere has recently installed a Passive Infra-Red sensor at the doorway to their bank vault. The sensor generates a 5V signal when a person is standing in front of the doorway, and 0V when no-one is at the doorway. The sensor is connected to `PORTB<0>`. The bank wishes to maintain a count of the number of people who ARRIVE at the vault doorway.

1. Write a routine called `Init` which will enable the appropriate interrupts. Your code should be appropriately commented. *4 marks*

2. Write an interrupt handler which will increment a register named `Cnt` when a person arrives. *6 marks*

[35] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 19 Write the letter you you have been assigned here________.

A: Count the number of times a single pole double throw switch is closed.

B: Turn an output pin on/off when a non-latching push-button is pressed.

C: LED flashes at a constant frequency

D: Speaker beeps at a constant frequency

For the task which you have been assigned:

1. Describe one way in which interrupts could be used to accomplish your task.

2. Identify 2 issues you need to consider in order to decide if using interrupts is appropriate for your task.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

20 Typical peripherals

At the end of this unit the student will be able to:

> *explain the operation of commonly used peripherals: PPI/PIA, A/D, Timers/Counters, PWM, D/A.*

In the previous units we looked at the operation of two peripherals on the PIC16F877, the port and the timer. Peripherals share the following characteristics:

- they have internal latches/registers which can be accessed by the microprocessor. These registers and latches are of various types: **control** (telling the peripheral what to do), **configuration** (telling the peripheral how to do it), **state** (reporting what the peripheral is presently doing) and **data** (containing the data the peripheral must send/recieve to the microprocessor)
- they are able to indicate their state to the microprocessor through the use of flags (internal or maintained by a peripheral controller), or output signals.
- (in conjunction with a PIC) they are able to trigger **maskable** interrupts i.e. the interrupt signal may be individually enabled/disabled.

There are a variety of peripherals which are typically used in conjunction with microprocessors(Horowitz and Hill 1989; 11.11 Peripheral LSI chips). In this unit we will look at the methods of implementation, and typical operation of these peripherals.

Please note that all peripherals operate asynchronously w.r.t. the microprocessor i.e. they are not guaranteed to respond/perform the operation within an instruction cycle, and **timing** should be carefully considered in your application. In particular, sampling/generation of rapidly changing signals will be limited.

Parallel I/O Ports

There is a data register which may either be read (to reflect input values) or written to (to set output values). If the port can be direction configured:

- there will be some method of specifying which bits are input and which are output. This may vary from a data direction register, where each bit reflects the direction of the corresponding port bit, to bits within a register which indicate a pre-defined pattern of inputs/outputs.
- there will be two separate data registers, one for input, and one for output. These registers are frequently mapped to the same internal address for external access.

Ports may also contain configuration registers for interfacing circuitry (e.g. internal pull-ups/ground).

Interrupts generated by ports are generally used to indicate a change of state on the port (i.e. inputs have changed)

Common Peripherals

All peripheral actions can be achieved using the PIA and software. Dedicated hardware, allows the CPU to do other things.

- ★ PPI/PIA/PSP
- ★ A/D
- ★ Timers/Counters
- ★ PWM
- D/A
- ★ USART
- PIC
- RTC

★ Built into PIC16F877

External Parallel Bus

- Bus lines
  - selection/address lines
  - data lines
  - read, write lines
- Master
  - PIC controls access to I/O devices/modules
  - any I/O port may be used
  - PIC must adhere to timing
- Slave
  - Another microprocessor accesses data on the PIC
  - PSP port must be used.
  - Master must adhere to timing
  - Interrupt on read/write

A/D conversion techniques

- Parallel encoding
  - Compare the input voltage to a series of equally spaced reference voltages
  - Each comparator outputs a single bit
  - The bit pattern can be translated to a binary number using a lookup ROM.
  - FAST (all bit simultaneous) but large circuit
- Successive approximation
  - Use a DAC, and binary search until the DAC output matches the input voltage.
  - Digital value is the final DAC input
  - Time proportional to number of bits
- Dual slope integration
  - Allow capacitor to charge for a fixed time
  - Digitally time the discharge for a constant current output
  - Discharge time is proportional to input voltage

See "Art of Electronics" Horowitz pp.622-624

D/A conversion techniques

- Scaled resistors
  - connect all bits via different resistor values to an op-amp
  - each resistor should be twice the value of the resistor for the next most significant bit.
- R-2R ladder
  - instead of inputting scaled 0/1 voltages, voltage inputs are switched from a "ladder" made up of 2 resistor values
- Pulse Width Modulation
  - a train of pulses is generated, whose on-width per period is determined by the digital value
  - the average voltage value of the pulse train is proportional to the digital value.
  - A low pass filter (RC) can be used to produce an averaged signal.

See "Art of Electronics" Horowitz pp.612-621

Timer/Counter

This peripheral has an internal register to hold the present value, and two inputs: one which increments the register value, and the other which resets the register (note that the register reset value may not always be 0). Generally, overflow of the internal register also triggers a reset. Optionally, the value of the register prior to reset is latched into another internal register. Both, or just one of the above registers may be externally accessible.

The peripheral as described is dual-purpose: When the increment is connected to a regular oscillating signal, and the reset connected to an irregular signal, the register will increment at regular intervals; indicating the amount of time passed between resets. When the signals are reversed, the register will count the number of cycles occurring in a particular interval. (Horowitz and Hill 1989; 15.10 Frequency, period and time-interval measurements)

The current state of the counter/timer (running, stopped, overflow occurred) is generally specified by flags in an accessible internal register.

Control registers associated with the timer/counter peripheral, start/stop the counter, and may determine the reset value. Configuration registers optionally choose scalers (these are divide-by counters which reduce the counted/reported values), enable or disable default signals, choose to count in one-shot or continuous modes.

Watchdog timers

A watchdog timer is a special timer which has pre-determined increment and reset inputs, no accessible internal registers, and outputs a signal on overflow (rather then flag and reset). The increment input is generally the microprocessor clock signal (possibly scaled), and the reset input must be tied to an instruction generated signal.

Control & Configuration registers enable/disable the timer, and choose the increment source and the scale values (where appropriate).

A/D and D/A conversion

The purpose of A/D conversion is to generate a digital value, which is proportional to, the analog offset from a known reference. The analog value which needs to be converted is generally a DC voltage; however DC current, and AC current, voltage, frequency levels may also be converted. The purpose of D/A conversion is to generate an analog value (DC current/voltage; AC current/voltage/frequency), whose offset from a known reference, is proportional to the original digital value.

A/D and D/A conversion involve some element of time over which the analog signal must be kept constant, therefore the control registers must make provision for starting/stopping the sampling and/or holding of the analog signal. A/D and D/A conversion peripherals have configuration registers which may be used to specify the reference input(s) to be used and the time associated with signal acquisition. They will also have data registers which contain the digital value, and the current peripheral state: i.e. (acquiring), converting, conversion done, (holding). The resolution of the A/D or D/A process will determine the effective bit-width of the data, and hence the number of data registers required to hold the result.

Please note that resolution and accuracy are **not** the same thing.

> ”Accuracy is often confused with resolution. Resolution relates the smallest signal(or noise) to the full-scale value. Accuracy relates the smallest signal to the measured signal. Accuracy is given as a percent and describes how close the measurement is to the actual value. *The signal is accurate to within* $\frac{V_{RESOLUTION}}{V_{SIGNAL}} \times 100\%$” – (Cady 1997; p 207)

Another characteristic property of A/D and D/A converters, is their linearity, i.e. how well they approximate a straight line relation between analog and digital values.

A/D and D/A conversion circuits may be based on several principles. The most commonly used with microprocessors are those that deal with DC voltage as the analogue signal. Popular types include: successive approximation A/D, dual-slope (charge–discharge) A/D, parallel (flash) A/D converters (Horowitz and Hill 1989; 9.20 Analog-to-digital converters pp. 622-624), scaled (binary weighted) resistors D/A, and R-2R ladder D/A (Horowitz and Hill 1989; 9.16 Digital-to-analog converters). One final form of D/A which we will consider separately is Pulse Width Modulation (PWM).

PWM

This frequency based system is used for data transmission as well as D/A conversion.

> ”In this technique the digital input code is used to generate a train of pulses of fixed frequency, with width proportional to the input count ... a simple low pass filter can be used to generate an output voltage proportional to the average time spent in the HIGH state i.e. proportional to the input code .... this sort of D/A conversion is used when the load is itself a slowly responding system; the pulse-width modulator then generates precise parcels of energy, averaged by the system connected ..” (Horowitz and Hill 1989; p.618)

Examples of usage are to control motor speeds, and display brightness.

Given a particular PWM frequency, PWM has resolution (like any other D/A or A/D technique) i.e. the smallest possible change in pulse width. Where the digital value has n possible values, the best use of PWM will map a change of 1 in the digital value, to the smallest possible pulse width change (PWM period$p$; smallest possible change $\frac{p}{n}$).

PWM peripherals typically offset the digital range, to make use of the fact that a digital value of 0, is equivalent to turning off PWM. This means that the maximum expressible digital value, can be used to represent a constant ”on” (pulse width=PWM period).

Frequency sampling and re-construction

The only signal a port pin is capable of directly **output**ting is a square wave. The maximum frequency at which a signal which can be output from a port or other peripheral depends not only on the processing/response time of the peripheral, but also on the speed with which the voltage rises/falls on the pin. While this is primarily determined by the pin circuitry, it is also influenced by the load which the output pin is driving. The signal which is output will have frequency components determined by the rise/fall times and the nature of the load (i.e. not a perfect square wave).

If a continuous signal is to be generated using a D/A converter, please note that the hold time will also limit the maximum frequency of the constructed signal (period >= 2*hold time), and introduce high frequency components which can be removed using a low-pass filter.

The maximum frequency at which change can be detected on an **input** pin will depend on how quickly the pin circuitry can respond to the signal. Realistically, however this signal is not recognised by the microprocessor until the peripheral has communicated with the microprocessor, and this may further reduce the detection frequency.

If a continuous signal is to be sampled using an A/D converter, the aperture (how long the signal is sampled over) and conversion (how long the converter takes to operate) times will also affect the maximum sampling frequency. The sampling frequency (Nyquist Frequency; Shannon Sampling Theorem) must be at least twice that of the input signal. A low pass *anti-aliasing* filter may be used on the input to guarantee this.

Programming peripherals

The programs presented so far, have addressed the programming needs of ports, timers and ISR's in an ad-hoc fashion. In programs which utilise more than one peripheral, a more rigorous approach makes it easier to ensure that the peripherals operate correctly.(** optional)

```
Blind Polling:

Initialise -- configuration
Repeat
    **Command -- control
    Wait for time
    **Read  -- data
```

```
Busy-wait Polling:

Initialise -- configuration
Repeat
    **Command -- control
    do
        Job
    Until -- state
    **Read  -- data
```

```
Interrupts:

Initialise -- configuration
UnMask -- interrupts
Command -- control
Repeat
    Job
ISR:
    save context
    **Command -- control
    **Read -- data
    **Clear -- Flags
    restore context
```

Review Exercises

1. (a) Identify the types of registers associated with peripherals, and explain the circumstances under which each type of register needs to be accessed.

   (b) Peripherals may be built into a microcontroller, or external peripherals may be interfaced via a generic parallel I/O port. Which is better? Why?

2. Choose the word combination which best completes the following sentence(s):

   ______2I______ Modulation may be considered as a means of ______2II______.
   The frequency/period of the signal is independent of the ______2I______.
   The ratio of ______2I______ to period is referred to as the ______2III______.

   (a) 2I:pulse width; 2II:digital-to-analog conversion; 2III:duty cycle
   (b) 2I:pulse width; 2II:analog-to-digital conversion; 2III:duty cycle
   (c) 2I:pulse wave; 2II:digital-to-analog conversion; 2III:duty cycle
   (d) 2I:pulse width; 2II:digital-to-analog conversion; 2III:cycle time
   (e) 2I:pulse width; 2II:analog-to-digital conversion; 2III:cycle time

3. The timer/counter peripheral can be used for all of the following functions except:

   (a) counting transitions (lo->hi or hi->lo) on a signal line
   (b) estimating frequency of a signal
   (c) generating a PWM signal
   (d) (integer) frequency division of a signal
   (e) determining the time between successive transitions(lo->hi or hi->lo) of a signal

4. The A/D converter sampling time is the sum of the acquisition and conversion time where:

   (a) the acquisition time is determined by the circuit properties, and the conversion time is constant.
   (b) the conversion time is determined by the A/D clock, and the acquisition time is determined by the circuit properties
   (c) the acquisition time is determined by the A/D clock, and the conversion time is determined by the circuit properties
   (d) the conversion time is constant, and the acquisition time is determined by the A/D clock.
   (e) the acquisition time is constant, and the conversion time is determined by the circuit properties.

5. (a) In your own words, differentiate between the accuracy and resolution of an A-D/D-A converter.

   (b) In your own words, explain how PWM may be used as a form of D-A conversion.

   (c) "An A/D converter is required to digitize a 1kHz sinusoidal waveform. What is the maximum allowable conversion time for the A/D? Assume a sample and hold circuit is being used to give the correct aperture time." –(Cady 1997; 11.8). Explain your answer.

6. An 8 bit timer/counter is being used with an oscillating signal of 4MHz, and a second source of indeterminate frequency. The frequency of the second source can be estimated by using the counter/timer in either the counter or timer configurations. What are the minimum and maximum frequencies that can be detected in (a) counter mode, (b) timer mode. Show all working.

7. Based on (Horowitz and Hill 1989; Exercise 9.4) A 10kHz PWM signal is being output based on an 8 bit binary value.

   (a) What is the length of the PWM period? Explain your answer.

   (b) What will the pulse width be if the binary value is 01110110_2? Explain your answer.

8. For the PIC16F877:

   (a) Write an initialization routine so that when a falling edge occurs on the RB0/INT pin, an interrupt will occur.

   (b) Write an RB0_INT interrupt handler that increments a 1-byte variable, `CountINT`. If `CountINT` is incremented to 100, set bit 5 of a variable `Flags`[36], then clear `CountINT` and finish by branching back to the polling routine.

9. On the PIC16F877 we can use Timer0 as a counter to count input pulses on the RA4/T0CKI pin. Write a program such that Timer0 generates an interrupt when 100 pulses have occurred. Within the interrupt handler set bit 5 of a variable, `Flags` and then reinitialize Timer0 so that another interrupt will occur when 100 more pulses have occurred.

   Compare the pros and cons of this approach for counting external events to the approach of Problem 8.

10. Challenge: For the PIC16F877:

    (a) If the PIC is clocked at 10 MHz, TMR0 can be used to obtain a 10 ms loop-time by counting the number of times it overflows. Write an interrupt handler that uses TMR0 and makes it overflow when 250 cycles have elapsed. When 10,000 interrupts have occurred then 10 ms would have elapsed. Code a main routine that waits until 10,000 interrupts have occurred.

    (b) Why is adding to TMR0 better than loading a value into it?

    (c) Note that in Problem (10a), it would seem that adding *six* to TMR0 after each overflow changes it into 250 cycle timer. However the normal increment of TMR0 is bypassed by the addition, suggesting that *seven* is more appropriate. In addition to which whenever TMR0 is written to, incrementing is inhibited for the following two instruction cycles because of the need to re-synchronize the timer with the clock. What would be an appropriate value to add to TMR0?

    (d) The approach used in Problem (10a) makes no use of the prescaler. How can the prescaler be used to cut down on the number of interrupts per 10-ms interval? What is the effect of each write to TMR0 clearing the prescaler on the long-term accuracy of the 10-ms delay time?

[36]This is to indicate that 100 input edges have been seen.

Tutorial Exercise 8AOffline Version[37]

ID# ______

The Bank of Nowhere has recently installed a security system at the doorway to their bank vault. The bank wishes to use a PIC16F877 to ring a bell for as long as someone is standing in the doorway. The volume of the bell should increase from minimum (after 10 seconds) to maximum (after 3 minutes). The bell is connected to `PORTC<2>` via some interface circuitry, such that it is ON when the pin is at a voltage corresponding to LOGIC-HIGH.

1. Describe how Timer1 could be used to determine how long the person has been standing in front of the doorway. You may use diagrams if needed. *3 marks*

2. Explain how PWM could be used to change the volume of the bell. You may use diagrams if needed. *3 marks*

3. Write a snippet to indicate how the Timer1 value would be used to affect the volume. You may assume that all peripherals have been appropriately configured. *Bonus 10 marks for an interrupt-based snippet.* *4 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

[37] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 20 Write the letter you have been assigned here________.

A: Analog to Digital Converter.

B: PWM Peripheral.

C: Digital to Analog Converter.

D: Counter/Timer.

1. Write a sentence to explain the function/operation of the peripheral in a μP based system.

2. Identify 1 timing constraint/specification associated with the peripheral.

3. Explain how the peripheral could be replaced by an I/O port and a small program

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - explain the operation of commonly used peripherals: PPI/PIA, A/D, Timers/Counters, PWM, D/A.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

21 Digital troubleshooting

At the end of this unit the student will be able to:

> *select and apply techniques for troubleshooting digital circuitry using standard laboratory equipment*

When you run into problems with your circuit: –
summarised from `http://learnat.sait.ab.ca/ict/digi290_godin/ppt/lab.ppt`

1. "Check the obvious" – see common errors
2. "Visually inspect the components and the wiring. Look for clues. Start with points that contain multiple connections. Check to ensure there are no floating inputs." – floating inputs tend to show up as transient faults. Just because your circuit was working fine before doesn't mean that a floating input was not the cause. Floating grounds are a particular problem!
3. "Check those things that are the quickest" – you can reduce your troubleshooting time by doing the quicker tests first.
4. "Analyze the inputs and outputs: Using the logic probe, look at the inputs and outputs starting with those that have the greatest effect on the circuit. Probe the contacts on the ICs, not the breadboard." – you can also do this with a scope/meter, and a shielded lead (or if you have to a bit of wire). Be careful not to short anything!!
5. "Rule of Halves: Where possible, split the circuit in half. ... Apply external signals where needed."

Common circuit errors –
summarised from `http://learnat.sait.ab.ca/ict/digi290_godin/ppt/lab.ppt`
See also (Predko 2001; Ch. 14 – PIC Micro MCU Application Debugging Checklist).

Multiple Outputs "Do not wire 2 or more outputs together. This will damage your components and is not proper practice." – this is a useful rule. On occasion you may wish to drive outputs simultaneously in order to increase the current drive capability. It is not worth the trouble ... switch using a transistor if necessary.

Driving LED's "You must use resistors in series with your LEDs. [Minimum] Recommended is 330 ohms. 7-segment displays are especially vulnerable to blow when no resistor is used to limit current. You must limit current for each segment."

Mis-wired Switches Unless the I/O port is equipped with an internal pull-up to V_{dd} or pull-down to V_{ss}, then you should have a resistor in series with the switch and tap off the resistor at the port.

DIP packaged switches/resistors SIL switches are wired as for single switches. Two way switches in DIP packages may look the same. Be careful!

Inadvertent component damage "CMOS components are susceptible to damage by static electricity. Use grounding precautions such as: Static Bags, static foam, etc Grounding straps, grounded work area, etc". Also when troubleshooting a live circuit with a meter, scope or probe, be careful not to short circuit any pins with the leads.

Incorrect power/signal connections Make sure that power and ground are correctly connected (i.e. no accidental reversal; polarity is correct). Try to use a convention for which rails are power and ground, and a particular color of wire for external power/signal connections. Make sure all signals have the correct peak-peak and DC offset voltages, at source and at connection.

Wiring When wiring component packages we often see: off by one; mirrored wiring; chip pin-out or pin assignment read incorrectly (look for the dot/notch (DIP) or the flat side (T092)).

Input/Output timing The input/output may be set to the right value, but at the wrong time. Use the scope to make sure that the timing on your signals occur in the correct sequence and is in-line with the specifications.

Meeting specifications "Using the wrong specification sheet or device (ICs, capacitors, resistors, etc)", "Applying the wrong input frequency or waveform", "Defective component, bent contact, or device operating outside of specification"

Troubleshooting induced faults You may be testing at the wrong point! Try to test as close to the signal source as possible, and then work your way out to check the integrity of signal connections Also note that test equipment can load circuits, and may discharge/charge capacitors. Choose your test points carefully.

Noise Noise sources can be external, or internal. Common internal sources of noise can be eliminated by decoupling lines, keeping wiring runs short, running wiring in parallel (or using twisted/shielded cables), and isolating high voltage or rapidly switching components.

Inter-Connections For breadboards: "Wires stripped improperly (too short)." – this may leave the input floating, or give a partial connection. You should also look out for shorts between neighbouring breadboard tracks, breaks along a continuous track, and jumpers between rails that you want tied together. For soldered boards: bad solder joints are often difficult to spot. Make sure you have continuity across the joint.

Fan-out "Exceeding fan-out, the number of gate inputs an output can handle (use a 1kohm pull-up or pull-down resistor as a temporary fix)."

In the event that your circuit is functioning as it should, there may be something wrong with your hardware/software design, or your understanding of how the hardware/software design should work.

" Six-step troubleshooting procedure:

- Identifying the problem
- Symptom elaboration
- Identifying the probable faulty functions
- Isolating the faulty function
- Isolating the faulty circuit
- Failure analysis

"– `http://www.but.be/WLN/Industrial%20Computer/DETDE.htm`

In order to follow this troubleshooting procedure, you will need more than the meter and oscilloscope you used to detect circuit errors. You will need to use your simulator, ICD, and logic analyser as well. You should also consider using simple indicators/triggers like LED's and switches to indicate what is going on when your program is running, and allow you to simulate inputs.

Identifying the problem You are observing some anomalous behaviour ... try to express the observed behaviour as clearly and unambiguously as possible. For example: "the circuit doesn't work" .... is not helpful. "there is a voltage of 1.6V on a pin which is outputting a low" ... clarifies the problem.

Symptom elaboration Does the problem only occur in particular circumstances or is it all the time? Does the problem appear when you step through the code, and disappear when you run the code (or vice versa)? The problem appears in which combination of the following: the simulator, the debug system, the live system? Can you do anything which changes/affects the problem temporarily? For example: "It only occurs on the debug system, not the simulation. The voltage changes when the pot is turned, even though the pot is on another circuit. A high is transmitted on the pin just fine."

Identifying the probable faulty functions List the possible causes. For example: "The pin circuity could be inappropriate, or badly configured. The system ground could be floating. I could be feeding a voltage back to the pin from my circuit. My instruction to put a 0 on the pin might not be working as I intended it to. The pin might be configured as an input. "

Isolating the faulty function Eliminate from your list, one at a time, by investigating using all the tools at your disposal. If you don't find the fault, try the next step or go back to the previous step! For example: "I haven't got anything intentionally connected to that output pin which should feedback. I haven't got any accidental cross-connections or breadboard shorts. Even when I disconnect the pin from the circuit, it is still happening. The debug screen shows a 0 on the pin, and that the pin is configured as an output. The pin circuitry looks ok. The system ground tests at 0 Volts on the chip. "

Isolating the faulty circuit (if appropriate) Disconnect the suspect circuitry, and see if the problem disappears. If it doesn't go back to the previous steps! For example: " I disconnected the circuit and the output went to 0V. Something must be loading/feeding back onto the pin"

Failure analysis Think about what caused this circuit/function to behave improperly. Re-think your hardware/software design in light of your observations. For example: " Where is the signal coming from? Can I move a connection or component, so it doesn't feedback? Does my circuit design allow a signal to feedback in a way I hadn't though about? "

See also (Predko 2000a; Ch. 12 – Debugging your Applications). **Finally**: Remember that troubleshooting is a very time-consuming process (especially if a re-design is needed), and plan your time accordingly.

Review Exercises

1. You have built a circuit to connect an LED between a PIC16F877 port pin and ground (via a resistor). You have written a program which toggles the port pin which the LED is connected

to. When you run the program on the PIC16F877 the LED does not appear to flash. Which ONE of the following statements is NOT a plausible explanation.

(a) The LED is faulty or is connected backwards.
(b) The port pin was configured as an input.
(c) The port pin cannot sink sufficient current to light the LED.
(d) The port pin has open-drain output circuitry, and cannot exert a "true" high value.
(e) The port pin is faulty.

2. You have a circuit in which a counter is connected to the output pin of the PIC16F877. The counter should count on each rising edge; however you notice that it counts twice on each rising edge. In order to discover the cause of the problem, the **most appropriate** action is to:

(a) use a pull-up resistor on the output pin signal
(b) connect an oscilloscope to the output pin and observe the waveform
(c) replace the PIC16F877
(d) replace the counter
(e) use a meter to check the supply voltage

3. You have built the circuit described in question 4 on page 7: " a circuit in which a switch is connected to `PORTB<6>` and to ground ... `PORTB` pull-ups are enabled, all `PORTB` pins are configured as inputs, and no other `PORTB` pins are connected to circuitry."

However, when you press the switch, and attempt to read `PORTB`, the value returned is 0x00. Which of the following statements is NOT a plausible explanation for this observation?

(a) PORTB was not configured correctly as inputs.
(b) The switch is faulty, and never makes contact.
(c) The pull-ups were not enabled.
(d) The program instruction is being executed from the wrong bank.
(e) All the pins of PORTB have inadvertently been shorted (or the port input circuitry is faulty).

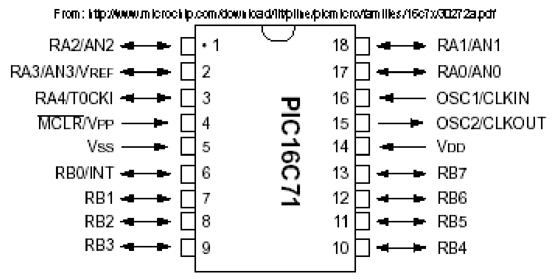
4. You have written a program for the PIC16F877 which uses interrupts from Timer0 to toggle the state of a port pin connected to an LED. When you start the program, the LED goes from off to on and then stays on. The port-pin is the only one which is configured as an output. Which of the following is NOT a plausible explanation:
   (a) The pin is toggling so fast that you can't see the LED flashing.
   (b) The ISR routine used the `return` instruction instead of the `retfie` instruction and only ran once.
   (c) The wrong bit-logical instruction was used to toggle the port-pin.
   (d) The LED is connected to the wrong port-pin.
   (e) The ISR did not clear the Timer0 flag.

5. You have built a circuit to drive a DC motor using PWM. The DC motor is connected to the power supply, and is switched on and off using a transistor which grounds the other side of the motor. The base of the transistor is connected to the PIC via a resistor. There are no other components. Your circuit works, but the transistor and/or the power supply fuse blows every now and again. Explain what the problem might be, and suggest a fix.

6. You have built a PIC16F877 based remote-controlled car which is battery powered. There is an intermittent fault, where where the car stops working. Switching the car off and on again corrects the problem. You observe that the problem happens when the car is driving into an obstacle, and when you attempt to accelerate strongly. Suggest a possible cause of the problem, and an appropriate solution.

7. A programmer writing an ISR for the PIC16F877 accidentally uses the `return` instruction instead of the `retfie` instruction. Presuming that interrupts are initially enabled, what would you expect to happen when the program runs and interrupt triggers occur? – Based on (Katzen 2003; Q. 14.2 p.422)

8. "Microchip recommend that where possible channel 0 RA0/AN0 should not be used in the PIC16C71 due to possible noise problems. Can you see why this is so and can you think of any way around this problem?"(Katzen 2003; Q. 14.9 p. 423)



9. "A certain PIC running at 20MHz has its Port C connected to LEDs tied high through a 1$k\Omega$ resistor with a 300pF capacitance to ground. All LEDs are off and the programmer attempts to turn on LED 7 and LED0 as follows:

```
bcf     PORTC,7    ;   Turn on LED7
bcf     PORTC,0    ;   Turn on LED0
```

   However only LED 0 actually turns on. What is happening? "(Katzen 2003; Q. 11.2 p. 299)

   Please draw the circuit, and identify at least two possible sources of the problem.

10. You have designed some hardware/software to interface the PIC16F877 with a device which will pull a signal line low for a given time (short 10 microseconds or long 20 microseconds), once it sees a falling edge (within 0.2 microseconds) on it's input.

    LED's are connected between `PORTC<1:0>` and the 5V rail. The device input is connected to `PORTC <2>`. The device output is connected to `PORTC <3>`. The PIC is wired to +5V supply and Ground, and is clocked using a 1MHz clock signal.

    The program should check the device, and then light one light-emitting diode for short, and two light-emitting diodes for long.

```
        ....
start   BANKSEL TRISC   ; configure TRISC
        movlw   0xF4
        movwf   TRISC
        BANKSEL PORTC
        clrf    Cnt
        mowlw   0x08
        movwf   PORTC
        clrw    PORTC   ; falling edge

loop    incf    Cnt,F
        btfsc   PORTC,3
        goto    loop

light   movf    Cnt,W   ;light diodes
        sublw   4
        btfss   STATUS,Z
        bsf     PORTC,1
        bsf     PORTC,0
        goto done
        ...
```

    When you run the program, neither diode lights.

    (a) Identify the pins used on the PIC16F877, and draw the circuits described.

    (b) Identify 3 possible causes of this problem, and outline how you would attempt to eliminate each possible cause.

Tutorial Exercise 8BOffline Version[38]

ID#

The Bank of Nowhere has recently installed a security system at the doorway to their bank vault. The bank wishes to use a PIC16F877 to ring a bell for as long as someone is standing in the doorway. The volume of the bell should increase from minimum (after 10 seconds) to maximum (after 3 minutes). Timer 0 is used to determine how long the person has been present. The PWM peripheral is used to vary the volume. The sensor is connected to `PORTB<0>`, and triggers an interrupt only when the person ARRIVES. Departure is detected by polling in the main loop.

Indicate ONE possible cause of EACH of the following symptoms, and indicate how you would test, and/or eliminate it.

1. The bell never rings. *3 marks*

2. When a person is in the doorway, the bell rings intermittently at 10 second intervals. *3 marks*

3. When a person leaves the doorway, after standing there for more than 3 minutes, the bell does not stop ringing. *3 marks*

[38]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 21 Write the letter you have been assigned here________.

Symptoms:

A LED is off; nothing happens when button is pressed

B LED is off; when button is pressed LED comes on and remains on despite further button pressing

C LED is on; nothing happens when button is pressed

D LED is on; when button is pressed LED goes off and remains off despite further button pressing

A PIC16F877-based system contains a switch-button that is connected (with appropriate circuitry) to `PORTA<4>`. The button state is polled from the main loop, and is used to decide if to light/out the LED connected (with appropriate circuitry) to `PORTC<0>`. You can assume there is nothing wrong with the code. Draw up a troubleshooting plan/flowchart for your assigned symptom.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - select and apply techniques for troubleshooting digital circuitry using standard laboratory equipment
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

22 PIC16 peripherals

At the end of this unit the student will be able to:

utilize built-in peripherals of the PIC16F877 microcontroller in simple applications.

The PIC16F877 has several built-in peripheral interface features which are multiplexed with the parallel i/o port pins. Please note: Each of these features could be implemented using a PIC without peripheral interfaces, (possibly augmented with custom hardware). For the sake of efficiency however dedicated hardware has been provided. Each will be reviewed in turn (NB. UART and PSP later in course) All information is from (PIC 2001) and (PIC 1997).

Port overrides

PORTC is an 8-bit wide, bi-directional port. The corresponding data direction register is TRISC. PORTC is multiplexed with several peripheral functions namely: Serial Communications, Pulse Width Modulation(PWM), Capture and ComPare (CCP), and oscillator input/ouput for Timer1. When enabling peripheral functions, care should be taken in defining TRIS bits for each PORTC pin. Some peripherals override the TRIS bit to make a pin an output, while other peripherals override the TRIS bit to make a pin an input.

PORTD is an 8-bit port; each pin is individually configurable as an input or output. PORTD can be configured as an 8-bit wide microprocessor port (parallel slave port) by setting control bit PSPMODE (`TRISE<4>`).

PORTE has three pins (RE0/RD/AN5, RE1/WR/AN6, and RE2/CS/AN7) which are individually configurable as inputs or outputs. The PORTE pins become the I/O control inputs for the microprocessor port when bit PSPMODE (`TRISE<4>`) is set. In this mode, the user must make certain that the `TRISE<2:0>` bits are set, and that the pins are configured as digital inputs(as PORTE pins are multiplexed with analog inputs which are activated by ADCON1). TRISE controls the direction of the RE pins, even when they are being used as analog inputs.

Watchdog timer

The WatchDog timer generates a device reset whenever it overflows and is normally used to recover from any malfunction that corrupts the contents of the program counter. For example, in unattended operations if a failure occurs in the program and the device gets into a perpetual loop, the watchdog timer can be used to reset the microcontroller and thus exit the loop. It can also be used in low-power battery applications to “awaken” the CPU periodically.

The Watchdog Timer (WDT) is a free running on-chip RC oscillator which does not require any external components. This oscillator is separate from any other oscillator on the microcontroller, which means that it will continue running even if the other oscillators have been stopped, for example during a `sleep` instruction. During normal operation a WDT time-out generates a device RESET. If the device is in SLEEP mode, a WDT Wake-up causes the device to wake-up and continue with normal operation.

The same prescaler that is used with Timer 0 can also be assigned to the WDT but it is used as a postscaler instead. The `clrwdt` and `sleep` instructions clear the WDT and the postscaler (if assigned to the WDT) and prevent it from timing out and generating a device RESET condition.

Timer 0

Timer 0 is an 8 bit timer with the following features:

- Internal or external clock select
- Edge select for external clock
- 8-bit software programmable prescalar
- Interrupt on overflow from 0xFF to 0x00

When TMR0 uses the internal clock it is said to operate in *timer mode* and it uses an external clock it is said to operate in *counter mode*. Timer mode is selected by clearing the T0CS bit (`OPTION_REG<5>`). In timer mode, the TMR0 will increment on every instruction cycle. Counter mode is selected by setting the T0CS bit. In this mode, TMR0 will increment either on every rising or falling edge of pin RA4/T0CK1. The incrementing edge is determined by the T0 Source Edge select bit, T0SE (`OPTION_REG<4>`). Clearing bit T0SE selects the rising edge.

Timer 0 and the Watchdog timer both share a single programmable prescaler which can be used to extend their ranges. The prescaler assignment is controlled by bit PSA (`OPTION_REG<3>`). Clearing bit PSA will assign the prescaler to Timer 0. When this is done the prescale value is software selectable using the bits PS2:PS0 (`OPTION_REG<2:0>`). The prescaler is neither readable or write-able. When the prescaler is assigned to Timer 0, all instructions that write to the TMR0 register will clear the contents of the prescaler. When it is assigned to the WDT, a `clrwdt` instruction will clear the contents of the prescaler along with the Watchdog Timer.

The prescaler is like an additional counter that is placed in series with Timer 0. Assume that the prescaler is set to 1:4 and both it and the Timer 0 are initialized to zero. The first clock pulse will increment the prescaler to 1, while the timer will remain on zero. Subsequent clock pulses will continue incrementing the prescaler only, until it contains 3. The next clock pulse will cause the prescaler to rollover to 0 and the timer will increment to 1. Since the prescaler cannot be read, a timer reading of say 10 with a prescale value of 1:4 can correspond to number of clock ticks from anywhere between 40 and 43. In order to avoid an unintended device reset the following two routines must be used when switching the assignment from Timer 0 to the WDT and vice versa. These are only needed if the WDT is used.

```
; Changing from Timer 0 to WDT (assumes Bank 0 i.e. RP0 cleared)
    clrf   TMR0         ; clear TMR0 and prescaler
    bsf    STATUS,RP0   ; switch to Bank 1
    clrwdt              ; clear Watchdog timer
    movlw  b'xxxxx1xxx' ; select new assignment and prescale value
    movwf  OPTION_REG
    bcf    STATUS,RP0   ; back to Bank 0

; Changing from WDT to Timer 0 (assumes in Bank 0)
   clrwdt               ; clear WDT and prescaler
   bsf     STATUS,RP0   ; switch to Bank 1
   movlw   b'xxxx0xxx'  ; select Timer 0
   movwf   OPTION_REG
   bcf     STATUS, RP0  ; back to Bank 0
```

The TMR0 interrupt is generated when the TMR0 register overflows from 0xFF to 0x00. This overflow sets the TOIF bit (`INTCON<2>`). The interrupt can be masked by clearing the TOIE bit (`INTCON<5>`). The TOIF bit must be cleared in software, by the Timer 0 interrupt service routine, before re-enabling this interrupt.

Timer 1

Timer 1 is a 16-bit timer whose values are kept in two readable and writeable 8-bit registers, TMR1H and TMR1L. Timer 1 is controlled by the T1CON (Timer1 Control) register1 and can operate in one of two modes, timer or counter mode. The operating mode is set by bit TMR1CS (`T1CON<1>`). Similar to Timer 0, Timer 1 increments on every instruction cycle in timer mode and on every rising edge of the external clock in counter mode. Timer mode is selected by clearing TMR1CS (`T1CON<1>`). Counter mode There are two sub-modes available, synchronized and asynchronous mode. In synchronous mode, the external clock input is synchronized with the internal phase clock. In this configuration, Timer 1 will not increment during SLEEP mode, even if the external clock is present, because synchronization circuit is shut off. When the bit T1SYNC (`T1CON<2>`) is set, synchronization is not done and Timer 1 will continue to operate during SLEEP mode. There is a special consideration that must be taken into account when reading Timer 1 because of its 16-bit value. Since it takes two reads to get its value, the timer may have incremented between the reads, thus, the value read may not reflect the actual value in the timer.

Timer 2

Timer 2 is an 8-bit timer with both a prescaler and a postscaler. It can be used as the PWM time-base in addition to normal timing activities. There is no external clock input so it is driven solely by the internal processor clock. The prescaler functions just like the prescaler of the other two timers, Timer 0 and Timer 1, and can have prescale values of 1:1, 1:4, and 1:16. The postscaler is used in a different fashion and gives the timer added flexibility. The other two timers overflow when they reach 0xFF (Timer 0) and 0xFFFF (Timer 1). Timer 2, on the other hand, overflows when its value matches the value in the PR2 register. The PR2 register, which defaults to 0xFF on reset, is readable and writeable. The output of the comparison between Timer 2 and PR2 is then sent to the postscaler which can have postscale values from 1:1 to 1:16. This gives the timer much more flexibility to time events, unlike Timer 0 where it was necessary to offset it on each overflow when timing events could not be mapped to a multiple of 0xFF timer ticks.

A/D Converter

The built-in analog-to-digital(A/D) converter of the PIC16F877 has 8 channels and these are multiplexed across ports A and E. The first five channels being on Port A<5,3:0> and the rest on Port E<2:0>. The ADCON0 register, controls the operation of the A/D module. The ADCON1 register, configures the functions of the port pins. The port pins can be configured as analog inputs, voltage reference, or as digital I/O(whose behaviour is controlled by TRISx and PORTx registers). The analog input channels must have their corresponding TRIS bits selected as an input.

The A/D module has high and low voltage reference inputs that are software selectable to some combination of VDD(device voltage), VSS(ground reference), RA2, or RA3. Ideally, the digital output value is the fraction of the difference between the reference voltages in which the analog voltage falls. The A/D converter produces a 10-bit result.

$$\frac{V_A - V_{REF}^-}{V_{REF}^+ - V_{REF}^-} * 2^{10}$$

After the A/D module has been configured as desired, the selected channel must be acquired before the conversion is started. The analog input charges a sample and hold capacitor. There is a minimum acquisition time which must be observed, to ensure that the capacitor charges to within 0.5 LSB of the actual input voltage. This is estimated in the data sheet as a worst case value of $20\mu s$, however this value is dependent on the temperature, the source impedance, and the supply voltage related switching resistance. For alternate conditions this acquisition time may be reduced.

During conversion, the sample and hold capacitor is disconnected from the input, and the converter then generates a digital result of the analog level on the capacitor via successive approximation. The A/D conversion of the analog input signal results in a corresponding 10-bit digital number. Some additional wait time is needed between conversions. The minimum wait time is $2T_{AD}$ where T_{AD} is the conversion time per bit.

The minimum conversion time per bit is dictated by the internal circuitry of the successive approximation A/D converter. For the PIC16F877 microcontroller this time must be at least $1.6\mu s$. The signal used to clock the A/D converter is software selectable as either an internal RC oscillator with a period of between 2 and $6\mu s$, or can be derived from the chip clock frequency (1/2, 1/8, 1/32). For example, a 20 MHz chip clock signal, has a period of $50ns$. It follows that the A/D conversion clock can only be set to 1/32 of the clock frequency, otherwise the $1.6\mu s$ limit will be violated. Total conversion time is $12T_{AD}$ for 10 bits, plus overhead.

Since the A/D is a 10-bit device, two bytes are needed to store the converted result. These are ADRESH (A/D RESult High register) and ADRESL (A/D RESult Low register). The extra 6 bits are not used. The A/D can be instructed to have the result left-justified. where the six least-significant bits of ADRESL are read as 0 or right-justified, where the six most-significant bits of ADRESH are read as 0. When the A/D conversion is complete, the result is loaded into the ADRES register, the GO/DONE bit (`ADCON0<2>`) is cleared, and A/D interrupt flag bit, ADIF, is set.

To recap, acquisition time is the time that the A/D modules holding capacitor is connected to the external voltage level. Then there is the conversion time of $12T_{AD}$, which is started when the GO bit is set. The sum of these two times is the effective sampling time. Please note that relative to the instruction cycle, this sampling time is slow, and should not be used to measure rapidly changing signals.

Capture/Compare/PWM

One function which is frequently performed by microcontrollers is the timing of external input signals, or the regular switching of external output signals. Both can be implemented using interrupt routines: either record the time when the event occurs or when the timer expires switch the external signal. The PIC16F877 has dedicated hardware which will perform such switching without the need for software. This peripheral has 2 channels on each of which it can perform one of three functions:

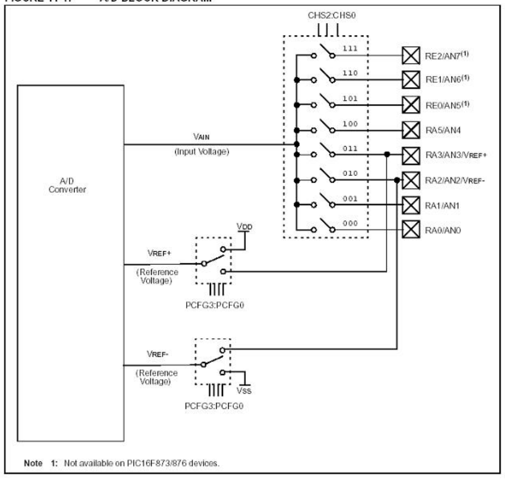
- automatic capture of the timer value when an external input occurs (CAPTURE)

  In Capture mode, CCPR1H:CCPR1L captures the 16-bit value of the TMR1 register when an event occurs on pin RC2/CCP1. An event is defined as one of the

PIC 16F877 A/D Block diagram

8 channels; 10 bit resolution; left/right justification; reference voltage configuration ADCON1

FIGURE 11-1: A/D BLOCK DIAGRAM



Note 1: Not available on PIC16F873/876 devices.

16F877 A/D Timing

Figure 22-2: A/D Conversion Sequence

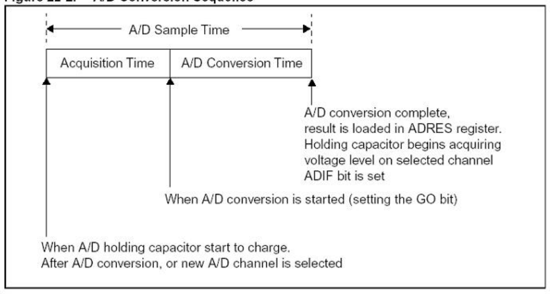
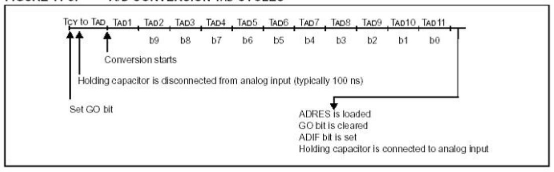

A/D Sample Time
Acquisition Time
A/D Conversion Time
A/D conversion complete, result is loaded in ADRES register. Holding capacitor begins acquiring voltage level on selected channel ADIF bit is set
When A/D conversion is started (setting the GO bit)
When A/D holding capacitor start to charge. After A/D conversion, or new A/D channel is selected

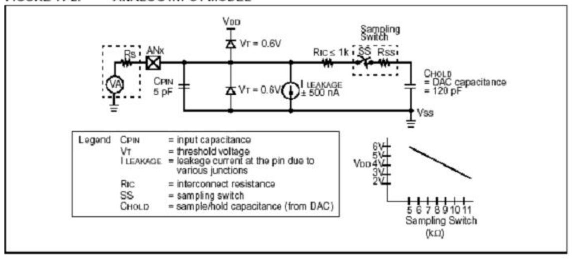
FIGURE 11-3: A/D CONVERSION TAD CYCLES


TCY to TAD
TAD1 TAD2 TAD3 TAD4 TAD5 TAD6 TAD7 TAD8 TAD9 TAD10 TAD11
b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Conversion starts
Holding capacitor is disconnected from analog input (typically 100 ns)
Set GO bit
ADRES is loaded
GO bit is cleared
ADIF bit is set
Holding capacitor is connected to analog input

Sampling Time

=Acquisition Time+ Conversion Time

FIGURE 11-2: ANALOG INPUT MODEL


VDD
VT = 0.6V
RIC ≤ 1k
Sampling Switch
SS RSS
Rs ANx
VA
CPIN 5 pF
VT = 0.6V
I LEAKAGE ± 500 nA
CHOLD = DAC capacitance = 120 pF
VSS
Legend CPIN = input capacitance
VT = threshold voltage
I LEAKAGE = leakage current at the pin due to various junctions
RIC = interconnect resistance
SS = sampling switch
CHOLD = sample/hold capacitance (from DAC)
6V 5V 4V 3V 2V
VDD
5 6 7 8 9 10 11
Sampling Switch (kΩ)

- For correct A/D conversions, the A/D conversion clock (T_{AD}) must be selected to ensure a minimum T_{AD} time of 1.6 μs.
- Between A/D conversions or after switching channels, the minimum sampling time must be observed (to avoid bad readings).

```
        LIST p=16f877

      ; Include file, change directory if needed
        INCLUDE "p16f877.inc"

      ; Start at the reset vector
        ORG     0x000
        nop

Start   BANKSEL PORTD
        clrf    PORTD       ;Clear PORTD
        movlw   B'01000001' ;Fosc/8, A/D enabled, Sample Channel
        movwf   ADCON0

        BANKSEL OPTION_REG
        movlw   B'10000111' ;TMR0 prescaler, 1:256
        movwf   OPTION_REG
        clrf    TRISD       ;PORTD all outputs
        movlw   B'00001110' ;Left justify, 1 analog channel
        movwf   ADCON1      ;VDD and VSS references

        BANKSEL PORTD

Main    bcf     INTCON,T0IF

Loop    btfss   INTCON,T0IF ;Wait for Timer0 to timeout
        goto    Loop

        bsf     ADCON0,GO   ;Start A/D conversion for channel 0

Wait    btfss   PIR1,ADIF   ;Wait for conversion to complete
        goto    Wait

        movf    ADRESH,W    ;Write A/D result to PORTD
        movwf   PORTD       ;LEDs
        bcf     PIR1,ADIF   ;Clear the completion flag

        goto    Loop        ;Do it again

        END
```

ICD tutorial

following:

- Every falling edge
- Every rising edge
- Every 4th rising edge
- Every 16th rising edge

The type of event is configured by control bits CCP1M3:CCP1M0 (CCPxCON$< 3 : 0 >$). When a capture is made, the interrupt request flag bit CCP1IF (PIR1$< 2 >$) is set. The interrupt flag must be cleared in software. If another capture occurs before the value in register CCPR1 is read, the old captured value is overwritten Timer1 must be running in Timer mode, or Synchronized Counter mode, for the CCP module to use the capture feature. In Asynchronous Counter mode, the capture operation may not work. (PIC 2001)

- automatic switching of an external signal when the timer value matches a pre-set value (COMPARE)

  In Compare mode, the 16-bit CCPR1 register value is constantly compared against the TMR1 register pair value. When a match occurs, the RC2/CCP1 pin is:

  - Driven high
  - Driven low
  - Remains unchanged

  The action on the pin is based on the value of control bits CCP1M3:CCP1M0 (CCP1CON$< 3 : 0 >$). At the same time, interrupt flag bit CCP1IF is set. Timer1 must be running in Timer mode, or Synchronized Counter mode, if the CCP module is using the compare feature. In Asynchronous Counter mode, the compare operation may not work. (PIC 2001)

- automatic toggling at a pre-defined signal frequency (PWM) Pulse-Width-Modulation is one means by which a digital value can be converted into an analog output.

  A PWM signal is a repeating signal that is on for a set ... time, [which] is proportional to the [desired output voltage] (Predko 2000c).

  i.e. the average voltage/power delivered by such a signal will be proportional to the "on" time.

The "on" time of the PWM signal is referred to as the pulse width or duty cycle, which may be expressed either in time, or as a percentage of the PWM period. The latter notation is more convenient, as the PWM frequency can be independently changed without re-specifying the duty cycle.

The choice of PWM frequency depends on two factors. Firstly, the frequency must be high enough that the device does not visibly/audibly/mechanically go on/off. Secondly, the frequency must be low enough that within the PWM period, there are sufficient "steps" available to express all the digital values which will be converted. This "step" size is referred to as the granularity of the PWM signal. The PWM frequency and granularity together limit the resolution of the PWM signal i.e. the maximum number of digital values which may be converted.

In the PIC16F877, although the Capture and Compare functions use Timer1 to perform their functions, the PWM function uses Timer2. The period is determined by Timer2 overflow value held in `PR2`. The pulse width is determined by the value held in `CCPR1L:CCPR1CON<5:4>`. The built-in PWM function granularity is determined by the chip oscillation period. (PIC 2001)

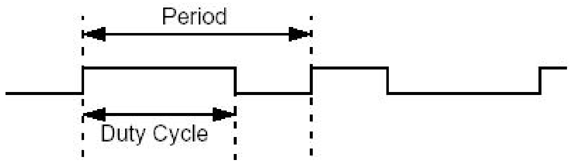
Simulator Limitations

The MPLAB simulator is an instruction-level simulator, i.e. it does not concern itself with events occurring between instructions. The following list itemizes the functions and peripherals among the entire PICmicro MCU family of microcontrollers that are affected by simulation on instruction cycle boundaries(MPL 2000):

- Clock pulse inputs smaller than one cycle can not be simulated even though timer pre-scalers are capable of accepting clock pulse inputs smaller than one cycle.
- PWM output pulse resolution less than one [instruction] cycle is not supported.
- Compares greater than 8-bits are not supported.
- In unsynchronised counter mode, clock inputs smaller than one cycle can not be used.
- The oscillator waveform on RC0/RC1 pins can not be shown.
- MPLAB-SIM does not simulate serial I/O.
- Timer0 (and the interrupt it can generate on overflow) is fully supported by the MPLAB-SIM simulator, and will increment by the internal or external clock. Clock input must have a minimum high time of 1 TCY and a minimum low time of 1 TCY due to stimulus requirements.
- Timer1 in its various modes is supported by the MPLAB-SIM simulator, except when running in counter mode by an external crystal.
- The external oscillator on RC0/RC1 is not simulated, but a clock stimulus can be assigned to those pins.
- PWM output (resolution greater than 1 TCY only) are not supported in this version of the MPLAB-SIM simulator.
- The Synchronous Serial Port is supported in SPI mode only. The MPLAB-SIM simulator currently does not support the I2C mode.
- The simulator does not load any meaningful value into A/D result register (ADRES) at the end of a conversion.

PWM in PIC16F877

- Software: Timer interrupt
- Hardware: PWM module
- Calculate of period and duty cycle times:
  - instruction frequency 1MHz (4MHz clock)
  - instruction cycle time T_{CY}=1µs
  - **pre-scale ratio 1:8 (LAB2 formula)**
  - For scale: 1:64; timer overflow cycle time 64 µs
  - Desired frequency 60Hz, 16.6ms, 260 ticks
  - Duty Cycle: Period (ticks) by %on-time
  - For an A/D reading 0-255 we can presume that the duty cycle is equal to the reading for a 60Hz PWM.
- Points to note:
  - minimum/maximum frequencies:eye/ear
  - frequency granularity: # steps
  - signal noise -- the CCP/PWM outputs are near to oscillator input lines!
  - current switching: isolate through transistor.



Built-In PWM

- Find the instruction cycle time:
  - F_{OSC} =16MHhz
  - F_{CYC} =16/4=4MHz
  - T_{CYC} =0.25µs
- Period 1ms
  = 1ms/0.25 µs
  = 4000 instruction cycles
- Choose TMR2 (8 bit) pre-scale:
  - 1:1 -- max. 256 instruction cycles
  - ~~1:4 -- max. 1024 instruction cycles~~
  - ~~1:16 -- max. 4096 instruction cycles~~
- PR2 Value = 4000/16 -1=249
- Resolution
  - TMR2 will count in increments of 16*0.25 µs = 4 µs
  - CCPR1L Value = OnTime (µs)/4
  - To get 1 µs resolution we need all ten bits e.g. 963 µs

963/4 = 240.75
11110000 → CCPR1L
11 → CCPCON<5:4>

Software -- PWM by force

- Find the instruction cycle time:
  - F_{OSC} =16MHhz
  - F_{CYC} =16/4=4MHz
  - T_{CYC} =0.25µs
- Period 1ms
  = 1ms/0.25 µs
  = 4000 instruction cycles
- Choose which timer and what pre-scale value:
  - Timer0: 8 bits; pre-scaler 1:~~1/2/4/8~~/16/32/64/128/256
  - Timer1: 16 bits; pre-scaler 1:1/2/4/8
  - Timer2: 8 bits; pre-scaler 1:~~1/4~~/16
  - Timers 0,2 will give a 4 µs increment; to get 1 µs resolution we will need to implement a software fraction
  - Timer 1 can give use a 0.25 µs increment; with a pre-scaler of 4 this will automatically be a 1 µs increment.

In Circuit Debugger (ICD) Limitations

The MicroChip ICD module is supported by the IDE, and provides programming and debugging facilities. Programming uses the In Circuit Serial Programming (ICSP) facilities of the Microchip family; debugging operates by adding a debug executive (kernel) to the code programmed on target. This means that some resources are not available to the system designer/programmer when using the ICD (ICD 2000):

- MCLR/VPP shared for programming
- RB3 reserved for programming
- RB6 and RB7 reserved for programming and in-circuit debugging
- Six (6) general purpose file registers reserved for debug monitor: 0x70, 0x1EB to 0x1EF
- First program memory location (address 0x000) must be a NOP instruction
- Program memory 0x1F00-0x1FFF reserved for Debug Code
- One(two) stack levels not available

Review Exercises

1. In order to use the Parallel Slave Port on the PIC16F877:
   (a) the PSPMODE bit in register TRISD must be cleared
   (b) the PSPMODE bit in register TRISE must be cleared
   (c) the PSPMODE bit in register TRISE must be set
   (d) the PSPMODE bit in register PORTE must be cleared
   (e) the PSPMODE bit in register TRISD must be set
2. The A/D module may be used to read from one of eight channels. Which of the following values, will allow the program to sample from analog channel 3, and deliver a right justified result if it is written to ADCON1 using

```
movlw value
movwf ADCON1
```

   (a) value EQU B'1011 0011'
   (b) value EQU B'1000 1110'
   (c) value EQU B'1000 0110'
   (d) value EQU B'1010 0001'
   (e) value EQU B'1111 0000'
3. In order to configure PORTB so that pins ¡4:3¿ are outputs and pins ¡2:0¿ are inputs, we can:
   (a) move the literal B'00011000' into TRISB
   (b) move the literal B'00011000' into PORTB
   (c) move the literal B'00000111' into PORTB
   (d) bit-wise OR TRISB with B'00000111' then bit-wise AND TRISB with B'11100111'
   (e) bit-wise OR PORTB with B'00000111' then bit-wise AND PORTB with B'11100111'
4. (a) Differentiate between the host and target machines with reference to the IDE, the compiler, and the generated machine code.
   (b) List features of simulators, and debuggers using MPLAB as an example. What are some of the disadvantages of using simulators to develop application code?
   (c) Why do we need a simulator, when we can debug directly on the system?
   (d) Why do we need debug facilities when we can just use the simulator?
   (e) Suggest some possible advantages of programming in assembly language rather than in high level languages (e.g. C).
5. One useful feature of the PIC16F877 microcontroller is the fact that it can report what condition caused it to re-start: Power-on, Brown-out, Watchdog timer reset or Physical reset (MCLR pin). Are these features supported in the simulator? Can the simulator be used to generate each of these conditions, in order to test any associated code?

6. The A/D converter on the PIC16F877 can accept reference voltages.
   (a) What range of voltages are appropriate for use as reference voltages?
   (b) What advantage could using different reference voltages offer?

7. Write a routine called `PWMSetup` for the PIC16F877 with a 1MHz clock, which will configure/initialise the PWM peripheral on CCP1, and set the initial PWM duty cycle to 1520 μs. What is the minimum pulse width change that your routine can make in μs?

8. What value must be loaded into `T2CON` in order to configure the PIC16F877 as follows: Timer 2 on, prescaler 1:16, postscaler 1:5.

9. (a) For the example code for time delays on page VI 19, calculate the percentage error caused by the neglected overhead, and the extra cycle on return for delays of 1 ms, 100 ms and 255 ms.
   (b) For the example on VI 20, the code timed 250 cycles, with the prescaler set to 128. For a 4MHz clock, what is the delay in ms?
   (c) Another method of obtaining a delay of 10 ms is to count for exactly 10,000 cycles. Now if the prescaler with TMR0 is set to 1:64 then it will overflow every 16384 cycles. Now $10,000 = 16384 - (100 \times 64 - 16)$. Design a main delay routine that tests for the overflow flag, clears it and bypasses exactly 6384 cycles, so that when it returns, at total of 10,000 cycles have been executed. An outline is shown below. As long as the main routine takes less than 10 ms to execute then the loop will be done in exactly 10,000 cycles.

```
        ORG 0x000

Init    ; initialize Timer0 etc

main    ; main routine
        call    Delay
        goto    main

Delay   ; test for overflow flag
        ; clear overflow flag
        ; bypass 6384 cycles
        return
```

   (d) Challenge: Devise a routine using TMR0 to get time delays of 15 ms.

10. Challenge: Identify a pair of conflicting peripherals on the PIC16F877 (i.e. the peripherals utilise the same pins). For your conflicting peripherals, suggest a work-around appropriate for an application which requires functions provided by both peripherals.

Lab 3 ID# ____________

You should complete the pre-lab before coming to your lab session. Your Teaching Assistant/Demonstrator may refuse to allow you into your lab session if your pre-lab is incomplete, or if you are unduly late.
You will have 6 hours in the lab to complete the exercises. All answers should be written on this lab-sheet in PEN. Please do not attach any extraneous pieces of paper unless SPECIFICALLY asked to do so.
Please bring your PIC16F877 datasheet book (provided in ECNG2006) to the lab with you. You may need to refer to it in order to complete the pre-lab and lab exercises. Your lab submissions will be checked for unwarranted collusion, and unreferenced use of Intenet-available/other resources.
At the end of this unit the student will be able to:

> *utilize built-in peripherals of the PIC16F877 microcontroller in simple applications.*

Pre-Lab

1. In general, a peripheral will have 4 types of information associated with it: configuration, control, state, and data. The information may be a whole register (as in the case of the parallel I/O port); it may also be a single bit, or a multi-bit pattern. For each of the following peripherals:

   - PortB
   - A/D
   - PWM

   locate the description of the peripheral in your datasheet, and identify the associated registers/flags. Classify each register, flag, or group of bits (as appropriate) as:

   - configuration – tells peripheral *how to do* task it will do later
   - state – peripheral uses to *report what doing/did*
   - control – tells peripheral *what to do* now
   - data –*information* the peripheral either provides or will work with

   Use the table below to summarize your findings. *3 marks*

Peripheral	Page #'s	Configuration	Control	State	Data
PortD		`TRISD, TRISE (PSPMODE)`		`TRISE(IBF, OBF, IBOV)`	`PORTD`
PortB					
Timer0	`p. 47-9`	`PSA, PS2:0`	`T0IE, T0CS`	`T0IF`	`TMR0`
A/D					
PWM					

Ports

2. All (parallel input-output) ports have at least two registers associated with them: a data register PORTx and a configuration (data-direction) register TRISx. PORTB is a register and is inside the PIC16F877. RB0 to RB7 are labels for pins external to the PIC16F877 IC. Pins RB0 to RB7 are connected to the register PORTB inside the PIC16F877. Since PORTB holds a byte it is logical to assume that each pin (RB0 to RB7) reads/writes one bit from/to PORTB. By default, PORTB pins are configured as inputs

   (a) What are the pin #'s on the 40 pin DIP package which correspond to RB0 to RB7? *1 mark*

   (b) If ideal high/low voltages are applied to pins RB0:RB7 as shown in the table, when PORTB is configured as all inputs, what bit pattern will appear in PORTB? *1 mark*

Pin Label	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
Pin #								
Pin Voltage/V	0	5	5	0	0	5	0	0
PortB Bit #	<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
Bit Value								

3. If we want to get pins RB0 to RB7 to reflect the values which we write to PORTB, first we must configure the pins of PORTB as outputs.

   (a) The registers that configure the PORT pins as INPUTS or OUTPUTS are the TRIS registers. What bank(s) are the TRIS registers located in?____________________ *1 mark*

   (b) How does one select a different bank? (Note: BANK0 is the default bank.)

      i. use the preprocessor directive `BANKSEL`.
      To change to the bank that contains TRISB what line of code would you use?
      ____________________ *1 mark*

      ii. explicitly set or clear the banking bits in the STATUS register.
      To change to the bank that contains TRISB what line of code would you use?
      ____________________ *1 mark*

   (c) If the TRIS bit is set '0' then the corresponding Port Pin acts as an output. If the TRIS bit is clear '1' then the corresponding Port Pin acts as an input. Complete the code below so that it changes all the PORTB Pins to outputs. Try to do it in as few instructions as possible. *3 marks*

```
BANKSEL _____    ; switch banks
_____________    ; put your code here
_____________    ;
_____________    ;
BANKSEL PORTB
```

4. In order to connect a simple switch as an input on register-bit PORTB$< 1 >$ (pin RB1). Determine the following:

   - the pin # on the PIC16F877 40 pin package that we need to connect to is ________. *1 mark*
   - the bit-mask and bit-wise-logical operation to change the relevant bit in TRISB to an input without affecting the other pins *1 mark*
   - the input voltage (logic-high threshold) & current requirement(s) of this pin are ________. *1 mark*
   - If the maximum current the switch can carry is 100 mA and the resistance of the switch is 0.2 Ohms design an appropriate circuit so that the pin is logic-high when the switch is ON, and logic-low when the switch is OFF *2 marks*
   - For the same conditions, design an appropriate circuit so that the pin is logic-high when the switch is OFF, and logic-low when the switch is ON *2 marks*

5. PORTB supports internal pull-ups. What are these and what are they used for? *2 marks*

6. PortB pins RB3, RB6 and RB7 are multiplexed with functions used by the ICD2 to communicate with the IDE i.e. for programming, receiving instructions (run, stop, step) and for returning register values. What do you think will happen if we connect circuitry to these pins while we are using the ICD2 to "debug" the PIC16F877? *2 marks*

Instruction Timing

The clock cycle time for the PIC16F877 is determined by the oscillator frequency; $Q = T_{OSC} = \frac{1}{f_{OSC}}$ The instruction cycle time for the PIC16F877 is 8 clock cycles; however due to pipelining it appears as 4 clock cycles. $T_{instruction} = 4 * T_{OSC}$

7. (a) For a 4MHz oscillator, what will be *actual* PIC16F877 instruction cycle time? ________. *1 mark*

   (b) For a 4MHz oscillator, what wil PIC16F877 instruction cycle time *appear* to be? ________. *1 mark*

Timers(Counters)

The **timer** (also known as the counter) is a type of peripheral. A *basic* timer has two inputs **increment**, and **reset**, and two outputs **current count** and **overflow signal**. Timers(Counters) overflow and generates a signal when the value held matches the overflow value. Where no overflow value is specified, this is the maximum value that the timer can hold (i.e. 8 bits – 0xFF). Counter/Timers take signals either from an external source, or from the instruction cycle time $T_{instruction}$.

A timer whose **increment** input is connected to a fixed frequency, will overflow at regular intervals, determined by the input frequency and the overflow value. Sometimes the frequency of the increment input signal is reduced by a preliminary stage (pre-scaler) which overflows at a small scale factor. The value held by the timer is then incremented on the rising/falling edge of the scaled signal. The value of the scale factor is determined by setting bits in the timer configuration register. Available scale factors are always powers of 2(i.e. 1:2; 1:4; 1:8).

8. (a) What value will Timer0 overflow at? ________. *1 mark*

   (b) What pre-scale factors are available for Timer0? ________. *1 mark*

A/D and PWM

The PIC16F877 has several additional peripherals. We will look at the built in Analog to Digital converter, and the built in Pulse-Width-Modulation generator. These peripherals automatically use timing, to sample/generate signals on one or more channels.

9. How many channels does the A/D converter support? Can signals on these channels be sampled simultaneously? If so, under what conditions. *2 marks*

10. How many channels does the PWM generator support? Can signals on these channels be generated simultaneously? If so, under what conditions. *2 marks*

Helpsheet – How to use the ICD2

Code should be compiled and tested using the simulator BEFORE you attempt to program/debug the PIC16F877-based application. When you are ready to change from Simulator mode to ICD mode, you should do the following:

1. If you built the circuit yourself, or made any changes, then get your circuit checked by the TA, technician, or lecturer.
2. **After your circuit has been checked!!** After your circuit has been checked!! Connect one end of the USB cable to the computer, and the other to the ICD2 module. Then plug the header into the ICD2 module using the RJ12 cable, and power up your circuit.
3. To use the ICD2 instead of the simulator, choose Debugger > Select Tool > ICD2. In some cases, a wizard will appear to help you set up the ICD2. If no wizard appears, goto the next step. To use the wizard, Click Next.

   The next form allows you to select a communications method. Under the Com Port select "USB". Click NEXT.

   The next form allows you to select the ICD power source. Select "Target has own power supply". Click NEXT.

   The next form allows MPLAB to autoconnect to the ICD2 at startup. **Do not check the box provided.** Click NEXT.

   This form allows MPLAB to automatically download the ICD2 operating systems when needed. Check the box provided and click NEXT. On the next screen click FINISH.
4. To connect to the ICD2 choose Debugger > Connect. A window should now show some messages; the final message should read

   ```
   MPLAB ICD 2 Ready
   ```

   Please ask your lecturer/lab assistant for help if any problems are encountered.
5. The ICD2 must now be enabled as the programmer. To do this choose Programmer>Select Programmer>MPLAB ICD2.
6. The configuration bits must now be set before programming the device. Select Configure. Now ensure that the following selections are made:

Oscillator	XT
Watchdog Timer	Off
Power Up Timer	Off
Brown Out Detect	Off
Low Voltage Program	Disabled
Flash Program Write	Enabled
Background Debug	Enabled
Data EE Read Protect	Off
Code Protect	Off

7. The device can now be programmed. Click Debugger>Program, to download the hex-file and debug code into the device. Programming may take a couple of minutes. During the process, the status bar should indicate that the device is being programmed then verified. When verification is complete the MPLAB ICD2 windows should indicate that the programming was successful and that the device is ready. be off/disabled.
8. The program can now be executed using Debugger>Run. The Status Bar should now show that the program is being run. The run process is halted by selecting Debugger>Halt. The circuit is now running your program.

Helpsheet – A/D and PWM – calculations & programming

A/D Timing

A/D sampling time = min'm acquisition time ($\approx 20\mu s$) + # bits (10) * conversion time/bit ($T_{AD} > 1.6\mu s$) + min'm wait time ($2T_{AD}$) where:

- Acquisition time is the time that the A/D module's holding capacitor is connected to the external voltage level.
- Conversion time is started when the GO bit is set, and includes conversion time for each bit, and a minimum wait time.

The signal used to determine T_{AD} is software selectable as either an internal RC oscillator with a period of between 2 and $6\mu s$, or can be derived from the chip clock frequency T_{OSC} with scale factor (1/2, 1/8, 1/32). Relative to the instruction cycle, this sampling time is slow, and the A/D should not be used to measure rapidly changing signals.

How to calculate the voltage represented by ADRESH (w/left-justified result):

The A/D answer is 10 bits long. If the user reads the 8 most significant bits (i.e. the 8 most left most bits in `ADRESH`): 255 will represent V_{REF+} volts and 0 will represent V_{REF-} volts.
In general, `ADRESH` will represent $\frac{V_{REF+} - V_{REF-}}{256} \times$`ADRESH` volts.

How to calculate the PWM frequency and duty cycle (pulse width)

"The PWM frequency is defined as $\frac{1}{[PWM period]}$." PIC (2001) To set the PWM period, we write values to PR2, and set the TMR2 prescaler (bit pattern set in the two LSB of T2CON).

`PWM period = [(PR2)+1]`$\times 4 \times T_{OSC} \times$ `(prescale value)`

To set the duty cycle we write values to CCP1RL and two lower bits. If we ignore the lower 2 bits of the duty cycle then: `PWM duty cycle = CCPR1L`$\times 4 \times T_{OSC} \times$ `(prescale value)`

Writing a program to interface with a peripheral

When writing a program which uses a peripheral you will probably want to employ a program which has the following structure:

```
configure
wait for and/or reset state
write data
assert control
wait for and/or reset state
read data
to repeat,loop to appropriate point
```

Sometimes these steps are not possible or not appropriate for particular peripherals, but the general structure holds in all cases. Where a wait is required we have several options:

busy wait loop until condition detected (nothing else happens)

instruction-based delay loop for a pre-set # of cycles (a specially timed thing happens (if any))

timer-based delay occasionally test if timer has expired (anything else may take place)

Helpsheet - 7 segment display patterns

A seven segment display consists of seven/eight elements arranged as shown:

```
 _
|_|
|_|.
```

Banks of seven segment display modules are typically used to display numeric information e.g. on calculator displays. A seven segment LED display module uses an individual LED to illuminate each of the segments. It typically has 10 connection lines: 8 which individually connect to the LED's, and 2 which connect to the LED common anode OR cathode.

Multiple seven segment LED display modules can be connected in parallel, with each module individually activated by a transistor which lets current flow from power (common anode) OR to ground (common cathode). If more than one module is activated simultaneously, they will both receive and display the same patterns. To make it look as though we are simultaneously displaying a **different** pattern on each display, we must switch rapidly between the displays. Read the Chapter from Predko (2000d) in your databook to clarify.

In this lab, the following patterns are used to display numbers, characters, and frequently used patterns on a 7 segment display.

```
0      1      2      3      4      5      6      7      8      9

 _             _      _             _      _      _      _      _
| |      |     _|     _|    |_|    |_     |_     | |    |_|    |_|
|_|      |    |_      _|      |     _|    |_|      |    |_|      |

a      b      c      d      e      f      g      h      i      j

 _                          _      _      _
 _|    |_             _|   |_|    |_     |_|    |_      _      _
|_|    |_|    |_     |_|   |_     |      _|    | |      |     _|

k      l      m      n      o      p      q      r      s      t

               _                   _      _
|_             _      _      _    |_|    |_|     _    |_     |_
|      |_     | |    | |    |_|   |        |.   |       |    |_

u      v      w      x      y      z

               _
       |_|           |_|    |_|     _|
|_|     _     |_|    | |     _|    |

dot    dash   minus  full   blank  vbars  hbars  hi-o   mu

                      _                    _      _
                     |_|           | |     _     |_|    |_|
   .    _      _     |_|.          | |     _            |
```

In the Lab

In this lab exercise, we will learn about bank switching, changing ports from inputs to outputs, reading switches and lighting simple LED displays. Then we will practice using three built in peripherals: Timer, A/D converter, and PWM module. Finally, we will create a scrolling alphanumeric display with user adjustable scroll rate and brightness level.

You will be given a circuit that looks like the circuit shown in Figure 13.

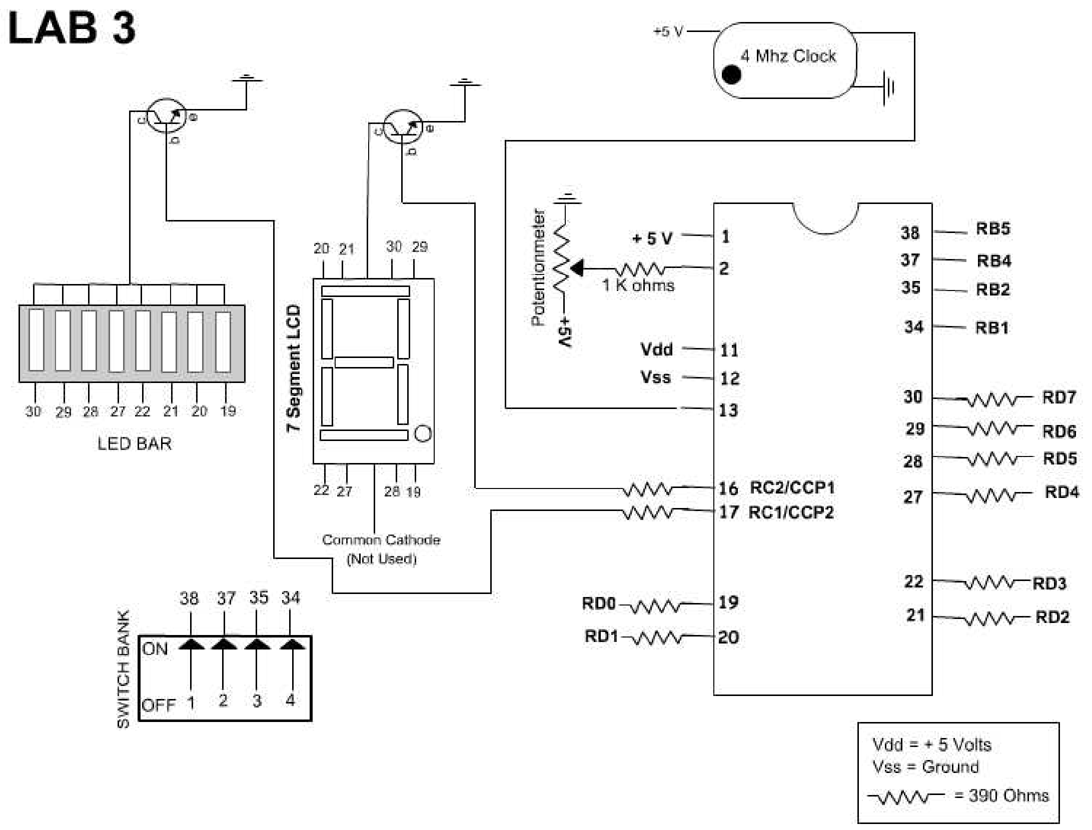


Figure 13: Lab 3 Circuit Diagram

Before starting each numbered question, you should read through the entire question, and make note of any queries that you may have.

Electronic copies of the code in this script are provided. Please note that there is no guarantee that the code will operate as required. Troubleshooting code is a skill which you are expected to use during this lab. Use the MPLAB ICD and MPLAB IDE debugging functions, to examine registers, find bugs and fix application software. As you work, try to make use of the IDE features, i.e. breakpoints, watch windows etc.

Call your TA or lecturer for assistance if you are stuck at any time.

ID# ____________________

Please identify

- your lab group letter (E,F,G,H): ____________________
- your ECNG2006 uP group designator (e.g. A3): ____________________

Micro-controller Circuit

1. The first step in using a microcontroller is to double check the circuit connections. The circuit you will be using in the lab is shown on page 63.

   (a) Are the pin numbers for all devices correct? Make adjustments to the diagram if needed. *1 mark*

   (b) Does the circuit diagram match the circuit? Make adjustments to the diagram if needed. *1 mark*

   (c) You will notice that there is a switch bank with 4 switches. One side of the switch is connected to GROUND and the other is connected to the PIC16F877 on PORTB.

      i. When a switch is OFF the input pin is: high low floating? (Circle correct answer) *1 mark*

      ii. When a switch is ON the input pin is: high low floating? (Circle correct answer) *1 mark*

   (d) The LED's have been connected so they light when the associated pin is at 5 Volts. Assuming that a typical LED has forward voltage drop of 2V and an 8mA min'm current requirement. Determine the following for the LED connected to PORTD<7>:

      i. the pin # on the PIC16F877 40 pin package is ________. *1 mark*

      ii. the maximum current that the pin can source/sink is ________. *1 mark*

      iii. the resistance connected in series with the LED is: ________. *1 mark*

      iv. Verify that the current requirements of the pin and LED are both met. *1 mark*

   (e) Each LED modules devices is grounded through a transistor (common cathode). The bases of the transistors are connected to pins RC1 and RC2 respectively. Explain how

      i. pins RC1 and RC2 can be used to control whether the LED devices are ON/OFF. *1 mark*

      ii. the circuit should be modified for common-anode LED devices. *1 mark*

Hook up your board and ICD2 now.
N.B Please let the lab technician/demonstrator check your circuit before you proceed.

Accessing ports in C and Assembly Language

2. To input/output signals from a port, the relevant registers/bits must be changed. We can do so using either an assembly language program or a C language program

   (a) Complete the following code, so that the LED Bar is switched ON.

```
        LIST    p=16f877
        INCLUDE <p16F877.inc>

test    EQU     0xFF

        ORG     0x00
        nop
        goto    main

        ORG     0x20
main
        BANKSEL TRISD      ; Select the Bank that contains TRISD and TRISC
        movlw   0x00       ; Make PORTD pins all
        movwf   TRISD      ; outputs
        clrf    TRISC      ; Make PORTC pins all outputs
        BANKSEL ______     ; Select the bank that contains PORTC and PORTD
        movlw   ______     ; Literal specifies pin connected to base of LED Bar transistor
        MOVWF   PORTC      ; Move value to PORTC
loop    movlw   test       ; Move this value to PORTD
        _______________    ; to prove that the LED bars were chosen
        goto    loop       ; Keep repeating

done    sleep
        END
```

   Create a new assembly language project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works.

   Next PROGRAM the PIC16F877 and ensure that the code works with the circuit.

   Did the LED's on the LED Bar light up with a bit pattern corresponding to `test`?

   Yes or No (Circle your answer)

   If it didn't what was the problem? How did you fix it?

   Use the MPLAB IDE and/or command line tools to determine:

   i. The size (in bytes) of the source code file:________. *1 mark*

   ii. The number of instruction memory locations used by the code:________. *1 mark*

   iii. The number of instruction cycles required to perform the loop:________. *1 mark*

   Demonstrate to the lecturer/demonstrator/TA who will sign your script.

(b) We can achieve the same effect using a PICC compiler C language program. Complete the following code, so that the LED Bar is switched ON.

```
#include <16F877.h>
#define test   0xFF
int main()
{
   SET_TRIS_D(0x00);    // Make Port D pins all outputs
   SET_TRIS_C(0x00);    // Make Port C pins all outputs
   OUTPUT_C(_______);   // Literal specifies pin connected to base of LED Bar transistor
   for (;;)             // keep repeating
   {
    OUTPUT_D(test);     // move this value to PORTD to prove that LED Bars were chosen
   }
}
```

Create a new PICC project for the PIC16F877 in the MPLAB IDE Compile the code and ensure that it is built successfully. Simulate it and see if it works.

Next PROGRAM the PIC16F877 and ensure that the code works with the circuit.

Did the LED's on the LED Bar light up with a bit pattern corresponding to `test`?

Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

Use the MPLAB IDE and/or command line tools to determine:

i. The size (in bytes) of the source code file:_________. *1 mark*

ii. The number of instruction memory locations used by the code:_________. *1 mark*

iii. The number of instruction cycles required to perform the loop:_________. *1 mark*

Demonstrate to the lecturer/demonstrator/TA who will sign your script.

(c) We can combine C and assembly language using multiple source files which are separately compiled/assembled and then linked. We can also use the `#asm` keyword to include assembly language in a C program.
Complete the following code, so that the LED Bar is switched ON.

```
#include <16F877.h>
#define test   0xFF
int main()
{
   SET_TRIS_D(0x00);    // Make Port D pins all outputs
   SET_TRIS_C(0x00);    // Make Port C pins all outputs
   OUTPUT_C(_______);   // Literal specifies pin connected to base of LED Bar transistor
   for (;;)             // keep repeating
   {
     // move this value to PORTD to prove that LED Bars were chosen
    #define PORTD_Address 0x08
    #asm
      movlw test
      movwf PORTD_Address
    #endasm
   }
}
```

Create a new PICC project for the PIC16F877 in the MPLAB IDE Compile the code and ensure that it is built successfully. Simulate it and see if it works.

Next PROGRAM the PIC16F877 and ensure that the code works with the circuit.

Did the LED's on the LED Bar light up with a bit pattern corresponding to `test`?

Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

Use the MPLAB IDE and/or command line tools to determine:

i. The size (in bytes) of the source code file:________. *1 mark*

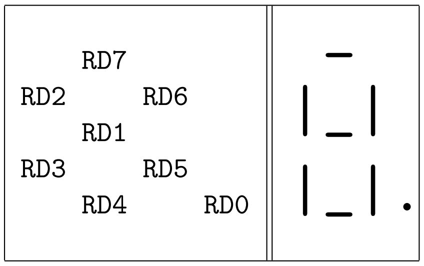
ii. The number of instruction memory locations used by the code:________. *1 mark*

iii. The number of instruction cycles required to perform the loop:________. *1 mark*

Demonstrate to the lecturer/demonstrator/TA who will sign your script.

(d) Which of the three implementations do you prefer? Justify your answer. *1 mark*

3. (a) Now we will attempt to make the 7 Segment LED active while making the LED Bar inactive. Before we look at the code we need to know a few things. How are the PIC16F877 pins connected to the Seven Segment LED?



The patterns for displaying digits and characters were provided in the helpsheet on page 62. Complete the table below which shows the logical values will let you display hexadecimal digits between `0` and `F`, and the decimal point. *2 marks*

RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	Display digit	Hex Value
1	1	1	1	1	1	0	0	0	0xFC
0	1	1	0	0	0	0	0	1	0x60
1	1	0	1	1	0	1	0	2	
1	1	1	1	0	0	1	0	3	0xF2
								4	
1	0	1	1	0	1	1	0	5	0xB6
								6	
1	1	1	0	0	0	0	0	7	0xE0
1	1	1	1	1	1	1	0	8	0xFE
								9	0xE6
1	1	1	0	1	1	1	0	A	0xEE
								b	0x3E
0	0	0	1	1	0	1	0	c	0x1A
0	1	1	1	1	0	1	0	d	0x7A
1	1	0	1	1	1	1	0	e	0xDE
1	0	0	0	1	1	1	0	f	0x8E
0	0	0	0	0	0	0	1	.	0x01

(b) Use the values from the table, and Complete the code shown on the following page. *2 marks*
Create a new assembly language project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works, then PROGRAM the PIC16F877 and ensure that the code works with the circuit.

Did Seven Segment LED light up with a number? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

What value is displayed on the Seven Segment LCD? ____________

Change the digit displayed on the Seven Segment LCD. Which line of code did you alter?

Demonstrate to the lecturer/demonstrator/TA who will sign your script. *1 mark*

```
        LIST    p=16f877
        INCLUDE <p16F877.inc>

        ORG     0x00
        nop
        goto    main

        ORG     0x20
main
        BANKSEL TRISD       ; Select Bank 1
        clrf    TRISD       ;  Configure PORTD and PORTC pins
        clrf    TRISC       ;  all as outputs
        BANKSEL PORTC       ; Return to bank 0
        movlw   ________    ; Switch on the 7-segment display
        MOVWF   PORTC

loop    movlw   0x08
        _____   lookup      ; convert digit to the bit pattern
        BANKSEL PORTD
        _____   PORTD       ; put bit pattern on PORTD
        goto    loop

done    sleep

                                ; Data EEPROM lookup table
lookup  andlw   0x0F            ; Mask to ensure this is a hex digit
        BANKSEL EEADR           ;
        movwf   EEADR           ; Specify lookup address
        BANKSEL EECON1
        bcf     EECON1, EEPGD   ; Trigger lookup from Data Memory
        bsf     EECON1, RD      ;
        BANKSEL EEDATA          ;
        movf    EEDATA, W       ; Retrieve Data into W
        return

        ORG 0x2100
        DE 0xFC , 0x60  , _____ , 0xF2  , _____ , 0xB6  , _____ , 0xE0
        DE 0xFE , 0xE6  , 0xEE  ,  0x3E  , 0x1A , 0x7A  , 0xDE  , 0x8E
END
```

(c) In Lab 2, you used a different kind of lookup table (using `retlw`) to check whether a BCD value was valid. Re-write this code, using a `retlw` lookup table to convert numbers to appropriate outputs for the seven segment display.

Create a new assembly language project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works, then PROGRAM the PIC16F877 .

Did Seven Segment LED light up with a number? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

What value is displayed on the Seven Segment LCD? ____________

Change the digit displayed on the Seven Segment LCD. Which line of code did you alter?

Use the MPLAB IDE and/or command line tools to:

i. Print your new program, and attach the printout to this script. *2 marks*

ii. Determine the number of instruction memory locations used by the code:________. *1 mark*

iii. Determine the number of instruction cycles required to perform the loop:________. *1 mark*

iv. Did the lookup change make the code either shorter or faster?

Shorter:	Yes	or	No	(Circle your answer)
Faster:	Yes	or	No	(Circle your answer)

v. Which lookup table method do you prefer? Explain your answer. *1 mark*

Demonstrate to the demonstrator/TA who will sign your script & printout.

(d) Challenge (Bonus 5 marks): Write a PICC program which will perform the same task using a pre-defined array. Write down the final program here:

Create a new PICC project for the PIC16F877 in the MPLAB IDE Compile the code and ensure that it is built successfully. Simulate it and see if it works. Next PROGRAM the PIC16F877 and ensure that the code works with the circuit.

Did Seven Segment LED light up with a number? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

What value is displayed on the Seven Segment LCD? ____________________

Change the digit displayed on the Seven Segment LCD. Which line of code did you alter?

If it didn't what was the problem? How did you fix it?

Use the MPLAB IDE and/or command line tools to determine:

i. The size (in bytes) of the source code file:________.

ii. The number of instruction memory locations used by the code:________.

iii. The number of instruction cycles required to perform the loop:________.

Demonstrate to the lecturer/demonstrator/TA who will sign your script.

4. (a) So far we have looked at outputting data (i.e. data moves from the PIC16F877 to the LED's). Let us try it the other way around (i.e. data moves from outside to inside the PIC16F877). Take a look at your circuit.

The table below shows the relationship between the switches and the PIC Pins.

Switch	PIC Pin
1	RB5
2	RB4
3	RB2
4	RB1

The following program is going to use the LED Bar to display the value read in from the inputs. We will be able to see how the inputs are changing. Echoing inputs to a simple display can sometimes be a useful technique when troubleshooting a μP-based application.

```
        LIST    p=16F877
        INCLUDE <p16F877.inc>

        ORG     0x00
        nop
        goto    main

main
        BANKSEL TRISB          ; Switch to bank 1
        movlw   0xFF           ; Set PORTB to
        movwf   TRISB          ; an input port
        bcf     OPTION_REG,7   ; enable pull-ups
        clrf    TRISD          ; PortD all outputs
        bcf     TRISC,2        ; Port C transistor is an output
        BANKSEL PORTB          ; Move back to bank 0
        bsf     PORTC,2        ; turn on transistor
loop    movf    PORTB,W
        andlw   B'00110110'    ; bit mask out only bits 5,4,2,1
        movwf   PORTD
        goto    loop           ; Create a loop effect

        sleep
        END
```

Compile and check your program in the simulator. PROGRAM the PIC16F877 and RUN the program. Play with the switches.

Describe what this program does, in your own words. *2 marks*

Demonstrate to the lecturer/demonstrator/TA who will sign your script.

(b) Let us change the program so that Switches 1-4 control the digit displayed on the 7-Segment display.

You will need one of the `lookup` functions used earlier. Paste it in the appropriate place in the code.

Because the switches are not connected to consecutive bits on PortB, we will need to use a file register to temporarily store the input, and make up the corrected output in the working register. Fill in the missing mask values in the code.

```
        LIST    p=16F877
        INCLUDE <p16F877.inc>

Inp     EQU     0x20

        ORG     0x00
        nop
        goto    main

main
        BANKSEL TRISB           ; Switch to bank 1
        movlw   0xFF            ; Set PORTB to
        movwf   TRISB           ; an input port
        bcf     OPTION_REG,7    ; enable pull-ups
        clrf    TRISD           ; PortD all outputs
        bcf     TRISC,2         ; Port C transistor is an output
        BANKSEL PORTB           ; Move back to bank 0
        bsf     PORTC,2         ; turn on transistor

loop    movf    PORTB,W
        movwf   Inp
        clrw
        btfsc   Inp,1           ; test bit 1, switch 4
        iorlw   ___________     ;    if set, then set bit 3
        btfsc   Inp,1           ; test bit 2, switch 3
        iorlw   ___________     ;    if set, then set bit 2
        btfsc   Inp,1           ; test bit 4, switch 2
        iorlw   ___________     ;    if set, then set bit 1
        btfsc   Inp,1           ; test bit 5, switch 1
        iorlw   ___________     ;    if set, then set bit 0
        call    lookup
        movwf   PORTD
        goto    loop            ; Create a loop effect

        sleep

; place either lookup function here!
        END
```

Compile and check your program in the simulator. PROGRAM the PIC16F877 and RUN the program. Play with the switches.

Does it work as you expected? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

i. Describe what this program does, in your own words. *2 marks*

ii. Is it possible to use a mathematical algorithm, to convert the bits from PORTB? If so, would it be faster/shorter than the lookup table? Explain your answer(s). *1 mark*

End of CORE material. ALL students expected to reach here within 3 hours.

Timers

5. (a) If the pre-scaler is used with Timer0, with a rate of 1:128, and T0CS is **clear**, what frequency will the timer overflow at (assume 4MHz clock signal)? *2 marks*

(b) Let's use Timer0 to slowly count **upwards** on the 7-segment LED display.

Use the code on the next page. You will need one of the `lookup` functions used earlier. Paste it in the appropriate place in the code. Verify that the code is consistent with the comments. Check the program in the simulator, then program the PIC16F877 and run the program.

Does it work as you expected? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

Show your results to the TA/lecturer who will sign your script. *4 marks*

(c) i. What **frequency** does the digit change at? ____________________ *2 marks*

Is this frequency what you expected? Yes or No (Circle your answer)

If it isn't, what was the problem? Use the space below to clarify your thoughts.

ii. What type(s) of delay did we use in the program: busy-wait, instruction-based, or timer based delay(s)? Explain your answer. *2 marks*

```
        LIST    p=16F877
        INCLUDE <p16F877.inc>
Cnt     EQU     0x22
Dlys    EQU     0x23

        ORG     0x00
        nop
        goto    main

main    bsf     STATUS,RP0      ; bank 1
        movlw   ___________     ; set T0CS=1, prescaler = 128
        movwf   OPTION_REG
        clrf    TRISD           ; PortD all ouputs
        bcf     TRISC,2         ; Port C transistor for 7-segment display is an output
        BANKSEL PORTB           ; Move back to bank 0
        bsf     PORTC,2         ; turn on transistor for 7-segment display
        clrf    Cnt

mloop   movlw   .250            ; set the number of Delays we want
        movwf   Dlys
        movf    Cnt,W           ; display the current count
        call    lookup
        movwf   PORTD
        incf    Cnt,F           ; increment the Count

iloop   call    Delay           ; call 250 Delay's
        decfsz  Dlys
        goto    iloop

        goto    mloop
        sleep                   ; in case we run past the goto, stop here

; Now for the delay routine
Delay   bcf     INTCON,T0IF     ; clear to be sure
        BANKSEL TMR0            ;
        clrf    TMR0

        bsf     STATUS,RP0      ; switch to bank 1 to access OPTION_REG
        bcf     OPTION_REG,T0CS ; start timing
        bcf     STATUS,RP0

Loop    btfss   INTCON,T0IF     ; keep checking for overflow
        goto    Loop

        bsf     STATUS,RP0      ; switch to bank 1 to access OPTION_REG
        bsf     OPTION_REG,T0CS ; stop timing
        bcf     STATUS,RP0
        return

; place either lookup function here!

        END
```

6. We can change the rate at which the digits change, by modifying the pre-scaler used by Timer0.

   Re-write the program so that the initial switch settings (RB5,RB4,RB2,RB1) are used to select the pre-scaler setting for Timer0 (PSA,PS2,PS1,PS0). The pre-scaler value should remain constant until the PIC16F877 is reset.

   Create a new assembly language project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works, then PROGRAM the PIC16F877 .

   Does it work as you expected? Yes or No (Circle your answer)

   If it didn't what was the problem? How did you fix it?

   Use the MPLAB IDE and/or command line tools to:

   (a) Print your new program, and attach the printout to this script. *4 marks*

   (b) Record the **frequency** at which the digit changes for: *4 marks*

      i. one switch on,

      ii. two switches on,

      iii. three switches on,

      iv. all switches on.

   (c) Does the pre-scaler perform as you expected? Comment on the accuracy and resolution of your results. *2 marks*

Demonstrate to the demonstrator/TA who will sign your script & printout.

7. Challenge (Bonus 10 marks): We can display a message on the Seven segment LED display, by displaying successive characters at regular intervals.

   Write a program (either assembly or C) which will display a message which is stored in memory (a lookup table (assembly), or an array (C)). The first byte stored in memory is the number of characters in the message. Each of the following bytes in memory is a byte which must be sent directly to the Seven Segment LED display module.

   You should write your program so that the message can be easily changed for demonstration. You should assume that messages will always be less than 180 characters long. Test your program with the following data:
   d'63', 0x1E, 0x2E, 0xDE, 0x00, 0xDE, 0x8E, 0x8E, 0xDE, 0x1A, 0x1E, 0x22, 0x56, 0xDE, 0x00, 0x22, 0x2A, 0x26, 0x1E, 0x0A, 0x38, 0x1A, 0x1E, 0x22, 0x3A, 0x2A, 0x00, 0x1A, 0x5E, 0x1A, 0x18, 0xDE, 0x00, 0xB8, 0x22, 0x1E, 0x2E, 0x00, 0xFA, 0x00, 0x66, 0xAA, 0x2E, 0x4A, 0x00, 0x3A,0x26, 0x1A, 0x22, 0x18, 0x18, 0xFA, 0x1E, 0x3A, 0x0A, 0x00, 0x22, 0x26, 0x00, 0x60, 0x4E, 0x26, 0x01, 0x00

   Create a new project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works, then PROGRAM the PIC16F877 .

   Does it work as you expected? Yes or No (Circle your answer)

   If it didn't what was the problem? How did you fix it?

   Attach a printout of the program to the script.

   Did this code work? Yes or No (Circle your answer)

   Demonstrate to the demonstrator/TA who will sign your script & printout.

8. This question will allow the student to investigate and develop code that converts Analog Data into Digital Data using the A/D peripheral. Configure `ADCON0`, `ADCON1` so that the A/D meets the following specifications:

A/D Conversion Clock	$\frac{F_{osc}}{8}$
Analog Channel	Channel 0 selected
State	A/D Conversion not in progress, A/D On
Result	rightt justified
Reference voltages	V_{dd}, V_{ss}
Port Configuration	AN0 as the ONLY analog input

Bit	<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
ADCON0								
ADCON1								

1 mark

The result produced by the A/D module is 10 bits long and not 8 bits nor 16 bits. This means that the answer is put into two registers named `ADRESH` and `ADRESL`. When the answer is left justified, valid bits are in `ADRESH<7:0>, ADRESL<7:6>`. When the answer is right justified, valid bits are in `ADRESH<1:0>, ADRESL<7:0>`. We will only be using `ADRESH`. The code appears on the following page. Fill in the values from the table above.

You will need one of the `lookup` functions used earlier. Paste it in the appropriate place in the code.

Check that the code is consistent with the comments before moving on. If not, what changes did you make?

In the code, Timer0 is used to set the sampling rate. If Timer0 has just overflowed, how long will it take to overflow again? Show your working.

2 marks

Compile the code and ensure that it builds successfully.

Did your code compile? Yes or No (Circle your answer)

If not, what changes did you make?

If you have problems ask the TA for assistance.

```
        LIST    p=16f877
        INCLUDE "p16f877.inc"
        ORG     0x00
        nop
        goto    Start           ; Start at the reset vector + 1 for ICD

        ORG     0x20
Start
        BANKSEL PORTD
        clrf    PORTD           ;Clear PORTD

        movlw   ___________     ;Fosc/8, A/D enabled, Sample Channel 0
        movwf   ADCON0

        BANKSEL OPTION_REG
        movlw   B'10000111'     ;TMR0 prescaler, 1:256
        movwf   OPTION_REG
        clrf    TRISD           ;PORTD all outputs
        clrf    TRISC           ;PORTC all outputs
        movlw   ___________     ;Right justify, 1 analog channel
        movwf   ADCON1          ;VDD and VSS references

        BANKSEL PORTD           ;Turn on 7-segment display
        movlw   _______
        MOVWF   PORTC

Main    bcf     INTCON,T0IF
Loop    btfss   INTCON,T0IF     ;Wait for Timer0 to timeout
        goto    Loop

        bsf     ADCON0,GO       ;Start A/D conversion for channel 0

Wait    btfss   PIR1,ADIF       ;Wait for conversion to complete
        goto    Wait

        movf    ADRESH,W        ;Write A/D result to PORTD
        call    lookup
        movwf   PORTD           ;LEDs
        bcf     PIR1,ADIF       ;Clear the completion flag

        goto    Main            ;Do it again
        sleep

; place either lookup function here!

        END                     ;this is the last line in the file
```

Does the program work as you expected? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

Demonstrate to the demonstrator/TA who will sign your script.

Program the 16F877, run the program and use a voltmeter to read the actual input voltage to the PIC from the potentiometer. Note the voltage and the value displayed by the 7-segment display in the chart below. Take 4 readings between 0 and 5 Volts, then work out the equivalent voltages for the readings you observed on the 7-segment display.

Input Voltage	PIC LED's	Equivalent Voltage

Comment on the accuracy and resolution of your results. *1 mark*

Would accuracy and resolution improve if we used more bits of the A/D result? How could we modify the code to achieve this? Try it! Explain what you changed, and take new readings. *1 mark*

Input Voltage	PIC LED's	Equivalent Voltage

Demonstrate to the demonstrator/TA who will sign your script.

Pulse-Width Modulation

9. We shall now investigate Pulse Width Modulation (PWM). Compile the code below and PROGRAM the PIC16F877. Then vary the POT.

   If the circuit is working properly the LED's in the LED Bar vary their brightness. Answer the following questions based on the code on the following page, and your observations.

   (a) Fill in appropriate comments at the places indicated. *2 marks*

   (b) What is the PWM frequency? Show your working. *2 marks*

   (c) Explain in your own words why the LEDS vary their brightness. *2 marks*

   (d) What happens if you change the PWM frequency specified by the Timer2 pre-scaler and re-compile and re-run the program? Your answer should state how you changed the Timer2 pre-scaler. *2 marks*

   (e) Based on your observations, is PWM an appropriate technique for D/A conversion? Explain your answer. *2 marks*

```
        LIST    p=16F877
        INCLUDE <p16F877.inc>
        ORG     0x00
        goto    main

        ORG     0x20
main
        BANKSEL PR2             ; ________________________________
        movlw   0xFF            ; ________________________________
        movwf   PR2             ; ________________________________

        BANKSEL T2CON           ; ________________________________
        movlw   B'00000110'     ; ________________________________
        movwf   T2CON           ; ________________________________

        movlw   B'00001100'     ; ________________________________
        movwf   CCP1CON         ; ________________________________

        BANKSEL TRISD           ;
        clrf    TRISD           ;Configure PORTD -- all output pins.
        bcf     TRISC,2         ;Configure PORTC<2> as an output pin.

        BANKSEL ADCON0
        movlw   B'01000001'     ;Fosc/8, A/D enabled, Sample Channel 0
        movwf   ADCON0
        BANKSEL OPTION_REG
        movlw   B'10000111'     ;TMR0 prescaler, 1:256
        movwf   OPTION_REG
        movlw   B'00001110'     ;Left justify, 1 analog channel
        movwf   ADCON1          ;VDD and VSS references

        BANKSEL PORTD
        movlw   0xFF            ;put an indicator value on the LED's
        movwf   PORTD


Loop    bcf     INTCON,T0IF
Here    btfss   INTCON,T0IF     ;Wait for Timer0 to timeout
        goto    Here

        bcf     PIR1,ADIF       ;Clear the completion flag
        bsf     ADCON0,GO       ;Start A/D conversion for channel 0

Wait    btfss   PIR1,ADIF       ;Wait for conversion to complete
        goto    Wait

        movf    ADRESH,W        ;_____________________________________
        movwf   CCPR1L
        goto    Loop            ;Do it again
        sleep
        END
```

Doing it yourself

10. Challenge (Bonus 10 marks): Attempt one of the following challenges in either assembly language or C:

    (a) Write a program which will light up the segments on the LED bar, in sequence, at regular intervals. The interval is determined using the switches, and the brightness of the display is determined by the pot.

    OR

    (b) Write a program which will simultaneously display a count which increments/decrements on the hex digit, as the segments on the LED bar light up in sequence.

    Create a new project for the PIC16F877 in the MPLAB IDE. Compile the code and ensure that it is built successfully. Simulate it and see if it works, then PROGRAM the PIC16F877.

    Does it work as you expected? Yes or No (Circle your answer)

    If your program didn't work on the first try what were the problems? How did you fix it?

    Attach a printout of the program to the script.

    Did this code work? Yes or No (Circle your answer)

    If so, **Show the demonstrator/TA who will sign your script & printout.**

Total marks 100.
This exercise is worth 5% of your ECNG2005 lab mark.

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts
- assistance in troubleshooting circuitry and/or using/operating the IDE/debugger

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

Unit 22 Write the letter you have been assigned here________.
Letters refer to the following peripherals located on the PIC16F877:
A: A/D, B: PWM, C: PSP, D: Data EEPROM

1. Which datasheet page(s) have information related to this peripheral? ________.
2. Identify and classify the registers associated with this peripheral:

Peripheral	Configuration	Control	State	Data

3. Identify ONE electrical and/or timing constraints associated with this peripheral.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this lab exercise.
  - utilize the MPLAB IDE and CCS compiler, to develop software for the MicroChip PIC16Cxxx series of microcontroller, in C, C++ and assembly language.
  - implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.
  - utilize built-in peripherals of the PIC16F877 microcontroller in simple applications.
- Which aspect of this lab exercise did you have the most difficulty understanding?
- Which aspect of this lab exercise did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this lab exercise.
- Identify one way in which this lab exercise could be improved.

Interfacing Peripherals

An understanding of peripheral operation is necessary in order to effectively interface the microprocessor to the external environment. This section has reviewed peripherals in general and the PIC peripherals in particular, as well as the basics of interrupts, and reading peripheral registers. It has also looked at some of the problems that may arise in protoyping/testind a system involving peripherals. As such it has contributed to the third course objective. You should now be able to "design and implement an appropriate interface between a microprocessor-based device and a peripheral device".

©DECE, UWI, St. Augustine, Trinidad
original author: Feisal Mohammed; updated: March 11, 2008– CLR

Part VII
System Issues

The previous part of the course looked at basic operation of the microprocessor in an applied context. This requires correctness of not just the program, but also the interface and circuitry.

In this part of the course, we look at both hardware and software issues which, while they do not directly affect system design, can influence system performance. These issues are related to interrupts, timing, signals and microprocessor choice. Design guidelines are presented on the basis of some consideration of these issues.

23 Interrupt issues

At the end of this unit the student will be able to:

> *understand the implications of (nested/multiple) interrupts for execution times, and the need for context saving.*

It is possible to predict the execution time of code, on the basis of the instructions which make up the program, and knowledge of how the pipeline (if it exists) performs. This was the basis on which instruction generated delays were constructed. For the PIC16F877 in particular, all instructions take 1 machine cycle, except for branches (goto, call, return) which take 2 cycles, conditional skips (bit test, increment/decrement) which take 2 cycles if the skip is performed, and the initial instruction executed on reset. The 2 cycle duration for certain instructions is due to the use of the pipeline. For example, the following code takes 3 (1+2) cycles for each iteration, and 4 (2+2) for the final stage:

```
Loop    decfsz  count,F
        goto Loop
        goto Done
```

There are many other considerations when attempting to predict the performance of code on a microprocessor. These include:

- overhead due to memory error detection/correction
- overhead due to memory cache
- overhead due to bus timing
- overhead due to writing peripheral/special function processor registers
- overhead due to interrupts

The last of these will be considered in the next section. The discussion of performance is followed by a review of two applications of the PIC.

Timing considerations with interrupts

When an interrupt source sends an interrupt signal to the CPU, it is signaling that it is ready to be serviced. Along with the signal, is the implication that this servicing needs to occur in a timely manner otherwise a failure may occur. For example, if characters are being received by a serial port, the UART will signal the CPU that a character has arrived and needs to be removed from its buffer. If the UART has a two byte FIFO buffer then the CPU must respond to the interrupt before two more characters arrive and overwrite the first character received. If the characters arrive at 9600 baud then the CPU must respond within roughly 2 ms. In order to design interrupt service routines that can service interrupts in a timely manner, the length of time that the ISR takes must be determined by counting the number of cycles the ISR takes.

Entering/leaving the ISR

The PIC inserts two additional cycles between the execution of the main instruction and the start of execution of the ISR. This is shown in Figure (14). When the `retfie` instruction is executed, the

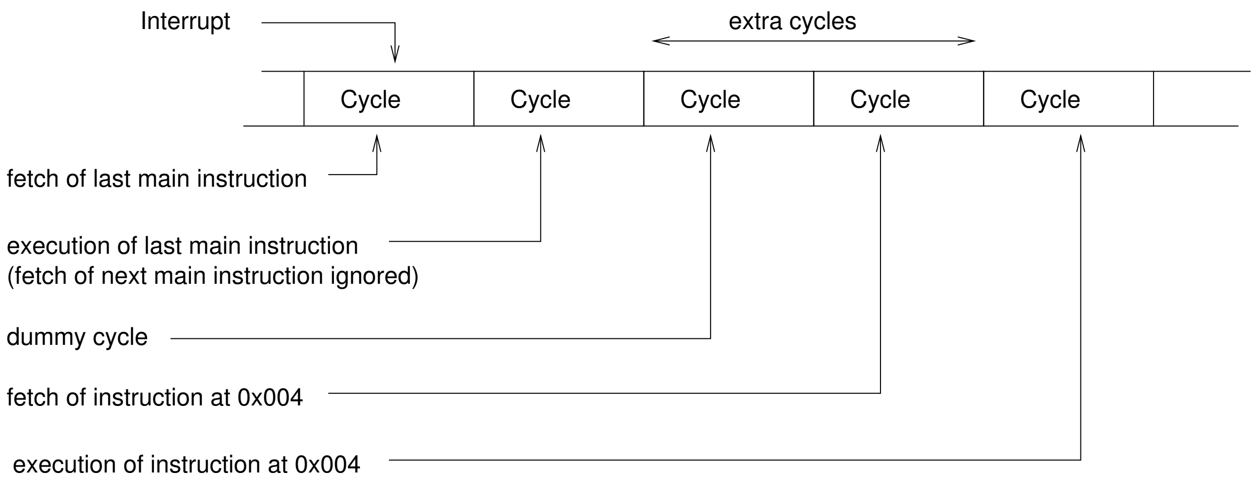

Interrupt
extra cycles
Cycle
Cycle
Cycle
Cycle
Cycle
fetch of last main instruction
execution of last main instruction
(fetch of next main instruction ignored)
dummy cycle
fetch of instruction at 0x004
execution of instruction at 0x004

Figure 14: Timing for start of interrupt

CPU does not insert any extra cycles as it returns to execution of the main program (Figure (15)). In addition, there are two more cycles to be taken into account since the actual ISR does not start

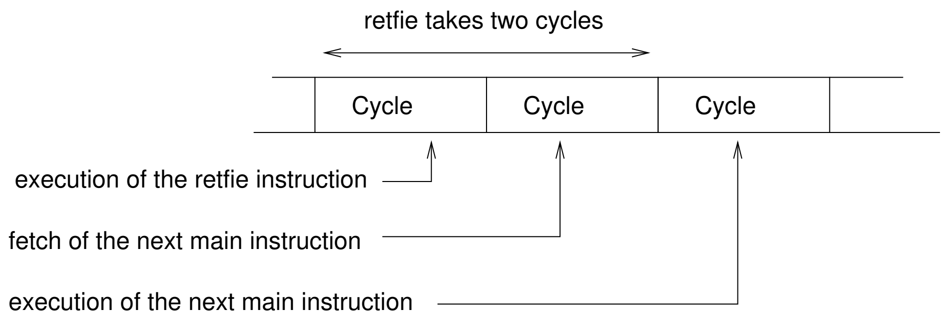

retfie takes two cycles
Cycle
Cycle
Cycle
execution of the retfie instruction
fetch of the next main instruction
execution of the next main instruction

Figure 15: Timing at end of interrupt

at 0x004 but is located elsewhere and must be reached by a `goto ISR` located at the 0x0004.

The ISR

Using the sample ISR (given previously in Part V) as a base, the number of cycles for execution can be determined. If there are N interrupt sources and the ith source takes T_i time for the execution of its interrupt handler then the execution time for each event is

Event	**Cycles**
Extra cycles inserted after execution of last main instruction	2
`goto ISR` instruction	2
Save W and STATUS registers	3
Polling routine	$2N$ - 1
Time for the ith handler	T_i
Restore W and STATUS	4
`retfie` instruction	2
Total	$12 + 2N + T_i$

These numbers can be represented as an overhead, (O_n), where the subscript represents the number of interrupt sources plus the time for one handler or $O_n + T_i$. The following table gives a listing of the overhead as the number of interrupt sources increases.

N interrupt sources (with only one pending)	Overhead (O_n)
1	14
2	16
3	18
4	20
.	.
.	.

Another time, P_i, can also be defined as the time it takes from the start of the ISR to the test of the ith interrupt source. The following table shows the values of P_i for various values of i.

N i	P_i
1	8
2	10
3	12
4	14
.	.
.	.

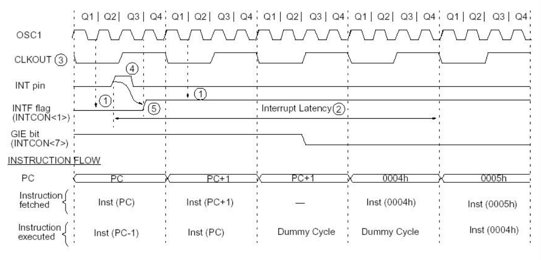
PIC16F877 ISR Points to note

- Interrupt Triggers
  - Individual interrupt flag bits are set, regardless of the status of their corresponding mask bit, PEIE bit, or GIE bit.
  - When bit GIE is enabled, and an interrupt's flag bit and mask bit are set, the interrupt will vector immediately.
- Interrupt Process
  - When an interrupt is responded to, the GIE bit is cleared to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with 0004h.
  - Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits.
  - The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid recursive interrupts.
  - The "return from interrupt" instruction, RETFIE, exits the interrupt routine, as well as sets the GIE bit, which re-enables interrupts.

PIC16F877 ISR Points to note

- Register Save/Restore
  - During an interrupt, only the return PC value is saved on the stack. Typically, users may wish to save key registers during an interrupt, (i.e., W register and STATUS register). This will have to be implemented in software.
  - Since the upper 16 bytes of each bank are common in the PIC16F876/877 devices, temporary holding registers e.g. W_TEMP, should be placed in here. These 16 locations don't require banking and therefore, make it easier for context save and restore when the program and ISR are in different banks.
- Interrupt latency
  - time from the interrupt event (the interrupt flag bit gets set) to the time that the instruction at the interrupt vector starts execution.
  - Synchronous interrupts (typically internal), the latency is $3T_{CY}$ (instruction cycles)..
  - Asynchronous interrupts (typically external), the will be 3 - $3.75T_{CY}$ The exact latency depends upon when the interrupt event occurs in relation to the instruction cycle.
  - Latency is the same for both one and two cycle instructions (by design).

Figure 8-2: INT Pin and Other External Interrupt Timing



Note 1: INTF flag is sampled here (every Q1).
2: Interrupt latency = 3-4 T_{CY} where T_{CY} = instruction cycle time.
Latency is the same whether Instruction (PC) is a single cycle or a 2-cycle instruction.
3: CLKOUT is available only in RC oscillator mode.
4: For minimum width of INT pulse, refer to AC specs.
5: INTF is enabled to be set anytime during the Q4-Q1 cycles.

Multiple Interrupts

- $O_n = 12 + 2N$ <u>(14+2N)</u>
  - the overhead time for an N interrupt source ISR
- $P_i = 6 + 2i$ <u>(8 +2i)</u>
  - from the start of the ISR
  - to the test of the ith interrupt source (presuming no other interrupt)
- T_i -- time for the ith handler
- How long:
  - will "Main" be "asleep"?
  - between event flag and ISR end?
  - Analysis considers all possible arrival patterns
- Consider worst-cases:
  - ISR(1) - high priority interrupt arrives long after low priority interrupt (84; 76)
  - ISR(1) - high priority interrupt arrives after low priority interrupt; but before even lower priority interrupt (118;116;108)
  - ISR(2) - (78; 109; 117)

Multiple interrupts

When multiple interrupts are pending at the same time, there are two things that are of interest, how long does the CPU take from the time it leaves the main code to when it returns and how long is the delay between the occurrence of an interrupt to when it is actually serviced. If the test for an interrupt source is higher up in the polling sequence then it is said to be of higher priority than one whose test is lower. Under normal circumstances a high priority interrupt will be handled before one of low priority but there are instances where this is not the case.

Consider a case where there are two interrupt sources, 1 and 2, where 1 is of a higher priority than 2 and that $T_1 = T_2 = 30$ cycles. The interrupts overlap with 2 occurring before 1. The time that the CPU will stop the running of the main program to when it continues execution depends upon whether source 1 or source 2 is serviced first. If source 2 is serviced first then the time will be

$$(O_2 + T_1) + (O_2 + T_2) = (16 + T_1) + (16 + T_2) = 92 \text{ cycles}$$

On the other hand, if source 1 is serviced first then the time will be

$$(O_2 + T_1) + (O_2 + T_2) - O_2 = (16 + T_1) + (16 + T_2) - 16 = 76 \text{ cycles}$$

The difference occurs because after servicing source 1, the ISR will then be able to poll source 2 and carry out servicing without having to return to the main program.

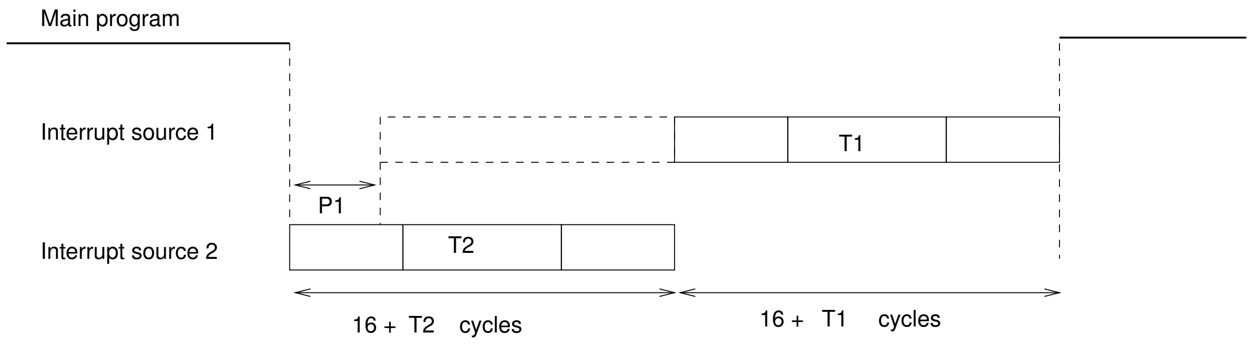


Figure 16: Interrupt 1 arriving too late to be serviced

It is instructive to determine the worst case time for the request for service, servicing of the interrupt and return to the main program ready to service a request by the same source. Figure (16) shows the sequence when the interrupt from source 1 arrives too late to be serviced, this gives the worst case time for source 1. Therefore the worst case time for source 1, T_{w1} is

$$T_{w1} = (O_2 + T_2) + (O_2 + T_1) - P_1$$

If from before $T_1 = T_2 = 30$ cycles then T_{w1} is 84 cycles.
The worst case time for source 2, T_{w2}, is then

$$(O_2 + T_2) + (O_2 + T_1) - O_2$$

Using the same assumptions from before $T_{w2} = 76$ cycles. Which gives the paradoxical result of the lower priority interrupt servicing actually completing quicker than the higher priority interrupt's servicing.

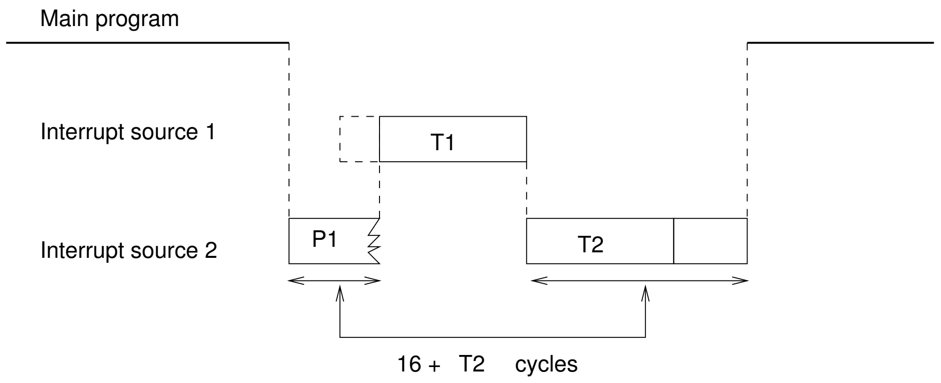


Figure 17: Interrupt 1 arriving early enough to be serviced

If now the situation is expanded to three interrupt sources with each handler taking 30 cycles, and the priorities are in the order 1, 2, and 3. Then the worst case timing for each of the three can be illustrated in Figures (18), (19), and (20). In Figure (18) the worst time, T_{w1} is

$$T_{w1} = (O_3 + T_2) + (O_3 + T_3) + (O_3 + T_1) - P_1 - O_3 = 118 \text{ cycles}$$

Similarly, the worst case time for source 2, T_{w2}, is

$$T_{w2} = (O_3 + T_3) + (O_3 + T_1) + (O_3 + T_2) - P_2 - O_3 = 116 \text{ cycles}$$

and finally

$$T_{w3} = (O_3 + T_1) + (O_3 + T_2) + (O_3 + T_3) - O_3 - O_3 = 108 \text{ cycles}$$

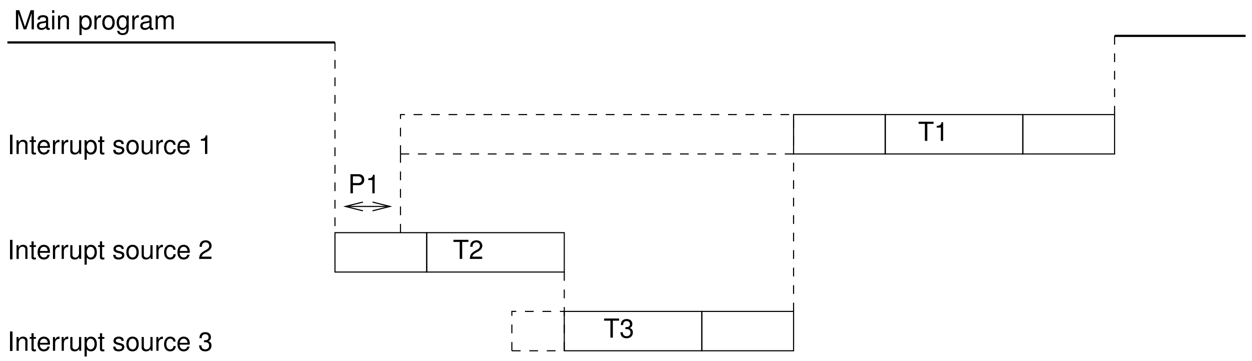


Figure 18: Worst case timing for source 1

This results in the following inequality

$$T_1 + T_2 + \cdots + T_n + O_n \leq \text{ worst case time} < T_1 + T_2 + \cdots + T_n + 2O_n \qquad (1)$$

Equation (1) shows that in the worst case, the priority of the interrupt source does not matter. The problem lies in the fact that on completion of the handler for a specific interrupt source, the polling routine is continued instead of restarted. Because of this continuation, all other pending interrupt sources are tested and serviced before attending to the higher priority interrupt source.

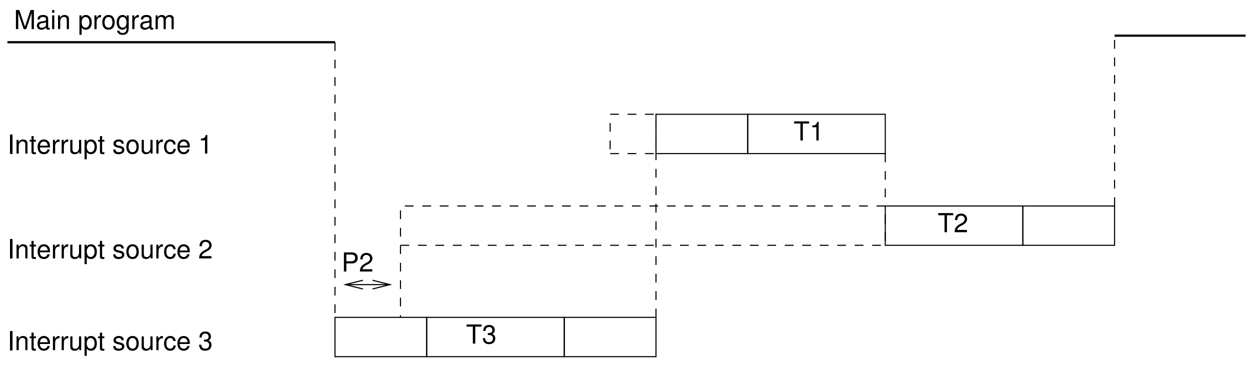


Figure 19: Worst case timing for source 2

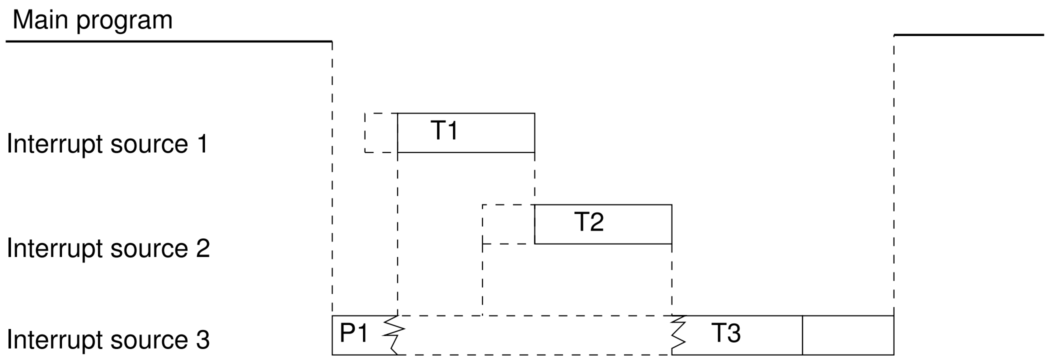


Figure 20: Worst case timing for source 3

If instead the ISR is written as follows, where at the end of each handler the poll is done again, then the interrupt from the higher priority gets the opportunity to be tested and serviced.

```
        ; save W and STATUS registers

        ; Polling sequence
Poll
        btfsc   INTCON,...   ; test if interrupt 1 occurred
        goto    Hnd1         ; jump to handler 1
        btfsc   INTCON,...   ; test if interrupt 2 occurred
        goto    Hnd2         ; jump to handler 2


Hnd1    ; code for handler 1
        goto    Poll         ; back to polling sequence

Hnd2    ; code for handler 2
        goto    Poll         ; back to polling sequence

        ; restore W and STATUS registers
        retfie                ; return from interrupt
```

The worst times for the modified ISR is shown in Figures (21), (22) and (23). The worst case time

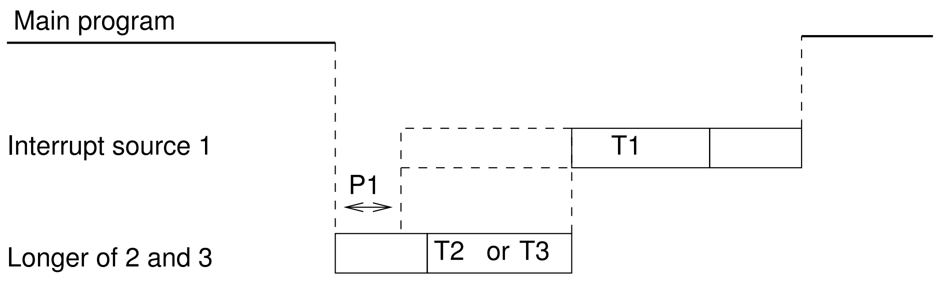


Figure 21: Worst case timing for source 1 (new ISR)

for source 1 is

$$T_{w1} = O_3 + (2 \times 3 - 1 + T_3) + (2 \times 1 - 1 + T_1) - P_1 = 78 \text{ cycles}$$

Here it is assumed that T_3 is the longer of T_2 and T_3. The $(2 \times 3 - 1)$ term is the time taken by the polling sequence to test all sources until it gets to source 3, whose flag is set. The 1 cycle in the $(2 \times 1 - 1)$ term is the time taken to test for source 1 when its flag is set.

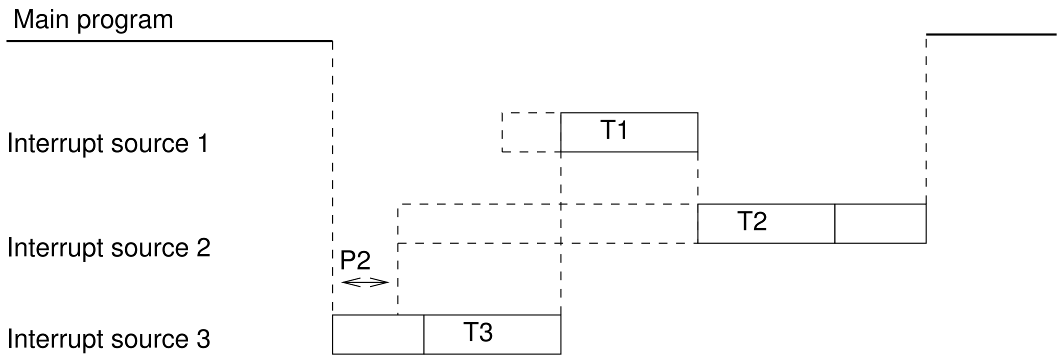


Figure 22: Worst case timing for source 2 (new ISR)

The worst case time for source 2 is

$$T_{w2} = O_3 + (2 \times 3 - 1 + T_3) + (2 \times 1 - 1 + T_1) + (2 \times 2 - 1 + T_2) - P_2 = 109 \text{ cycles}$$

The worst case time for source 3 is

$$T_{w2} = O_3 + (2 \times 2 - 1 + T_2) + (2 \times 1 - 1 + T_1) + (2 \times 3 - 1 + T_3) = 117 \text{ cycles}$$

Comparing the results with those of the previous ISR, it can be seen that there is significant improvement for the highest priority interrupt, source 1. The next highest, source 2, is slightly improved also and the lowest priority, source 3 is slightly worsened. Now the T_{w1} depends only upon the longest handler of low priority, T_3 and, of course, T_1 instead of the times of all the handlers as before.

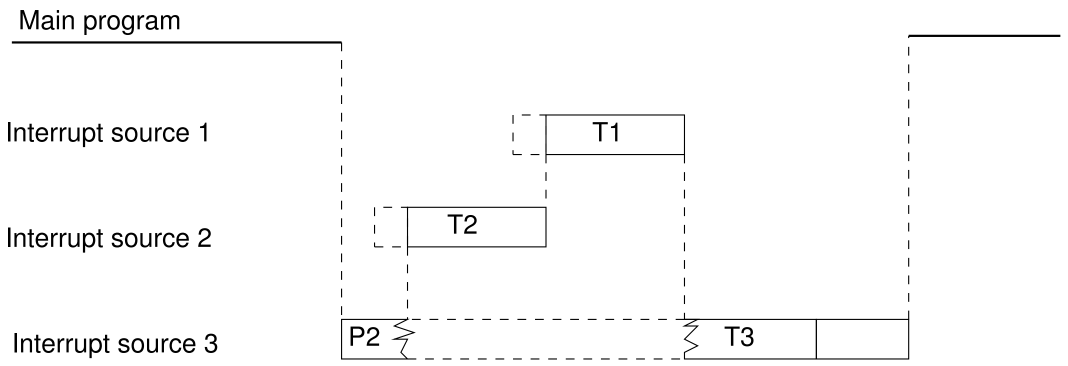


Figure 23: Worst case timing for source 3 (new ISR)

Race conditions

Whenever there are interrupts the occasion may arise where there is data that is shared between the main program and an interrupt handler. If the data (register(s)) is modified by either the main program or the handler then a race condition exists and the final result depends upon which code runs and precisely when it runs. In order to avoid this condition is to ensure that reading or writing to the shared data is done by only one part of code at any time. In other words, mutual exclusion must exist. This is normally accomplished by identifying critical sections, which are sections of code where reading or writing takes place of the shared data. When the main program is in a critical section it must not be interrupted. As an example, assume that the main program decrements a 16-bit variable,`count` = (`counth, countl`) while the interrupt handler increments it. When the code is being executed the variable is being decremented from 0x0100 to 0x00ff.

```
movf    countl,F     ; line 1
btfsc   STATUS,Z     ;  "   2
decf    counth,F     ;  "   3
decf    countl,F     ;  "   4
```

If an interrupt occurs between lines 3 and 4, the interrupt handler will find `count` equal to 0x0000 and it will return to the main program after changed it 0x0001. The main program will then execute line 4 and `count` will have the final value of 0x0000 instead of the correct value of 0x00ff. The critical section can be protected in the following manner.

```
Again   bcf     INTCON,GIE  ; disable interrupts
        btfsc   INTCON,GIE  ; check that an interrupt did not occur
        goto    Again       ; ...in the meanwhile and set GIE back to 1
                            ; Start Critical region
                            ;   code
                            ; End Critical region
        bsf     INTCON,GIE ; re-enable interrupts
```

After the instruction to disable interrupts has executed it is necessary to check that the GIE flag is clear because an interrupt could have occurred just as the `bcf` instruction was executed. If this happened then the interrupt would be serviced before the next instruction leaving the GIE bit set.

Race Conditions

- who gets there first?
  - final result depends on the order of events
- Occurs when
  - two separate routines (main, ISR) *actively* access the same variables

    AND/OR
  - successive commands *must* be executed without interruption

Critical Sections

- the code in a critical section is never interrupted
  - implemented on the PIC16F877 by disabling/re-enabling GIE
  - test for GIE *after* it is disabled
  - can be used to avoid race conditions

Sample ISR -- 21 Cycles

```
                       ORG    0x04
1,2                    goto   ISR

3      ISR             movwf  _w
4                      movf   STATUS,w
5                      movwf  _status

6,7                    btfsc  PWM
7,8                    goto   PWM_ON

8      PWMOff          nop
9                      bsf    PWM
10                     movf   Ontime,W
11,12                  goto   PWM_Done

9      PWMOn           bcf    PWM
10                     movf   Ontime,W
11                     subwf  Period,W
12                     nop

13     PWMDone         sublw  0
14                     movf   TMR0
15                     bcf    INTCON,T0IF
16                     movf   _status,W
17                     movwf  STATUS
18                     swapf  _w,F
19                     swapf  _w,W

20,21                  retfie
```

TMR0 - 1:16 prescaler

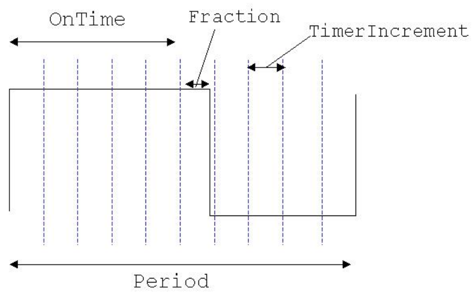
`Period` = (1ms/0.25μs - 2*Interrupt Run time)/16
`OnTime` = (pulse width/0.25μs - Interrupt Run time)/16

Ideal Interrupt Run Time is a multiple of 16.
Modify the routine using additional `nop` to meet the requirements.

Resolution is limited only by the clock period 0.25 μs.
Define `Fraction` as the number of additional instruction cycles required -- it will be used to delay the onset of the OFF time in the ISR.

`Fraction` =

$$\lfloor(\text{pulse width}/0.25\ \mu s)/16\rfloor * 16 - (\text{pulse width}/0.25\ \mu s)$$



A possible revised ISR

```
PWMOn  1  movf     Fraction,W    PWMOff    nop
       2  sublw    0                     1 nop
       3  addwf    PCL, F                2 nop
          nop                            3 nop
          nop                            4 bsf     PWM
          nop                            5 nop
          nop                            6 nop
          nop                            7 nop
          nop                            8 nop
          nop                            9 nop
          nop                           10 nop
          nop                           11 nop
          nop                           12 nop
          nop                           13 nop
          nop                           14 nop
          nop                           15 nop
          nop                           16 nop
          nop                           17 nop
       4  bcf      PWM                  18 nop
       5  movf     Fraction,W           19 nop
       6  addwf    PCL, F               20 nop
       7  nop                           21 movf    OnTime,W
       8  nop                        22,23 goto    PWMDone
       9  nop
      10  nop
      11  nop
      12  nop
      13  nop
      14  nop
      15  nop
      16  nop
      17  nop
      18  nop
      19  nop
      20  nop
      21  nop
      22  movf     OnTime,W
      23  subwf    Period,W
```

ISR call cycles	2
Pre-branch cycles	8
Branch cycles	23
Post-branch cycles	9
	42
Nearest 16 multiple	48
∴`nop`s required before `retfie`	6

Rotary Pulse Generator (RPG)

One of the biggest uses of microcontrollers such as the PIC chips is in embedded control and the range of applications is quite extensive. One such use is in digital instruments. In contrast to fully digital systems, analog systems would use potentiometers to vary values e.g. voltage output from a power supply.

In digital systems, use of a potentiometer would require an A/D to convert the signal from the potentiometer. It would be better to have the control to be fully digital. One such method is through the use of a Rotary Pulse Generator (RPG).

A rotary pulse generator is an optical rotary incremental encoder. It is commonly used by instruments that require the ability to vary setup parameters, such as voltage, frequency, etc. The RPG has two outputs, channel A and channel B and by comparing the both outputs, it can be determined if the RPG is being turned in a clockwise or anticlockwise direction. Figure (24) shows the waveforms when it is turned in either direction. If the rising edge of channel A is compared with the signal level on channel B, it can be seen that for clockwise motion, the rising edge of A coincides with a low on B. For counter-clockwise motion, the rising edge of A coincides with a high on B. Thus the number of pulses can be determined in addition to the direction of motion.

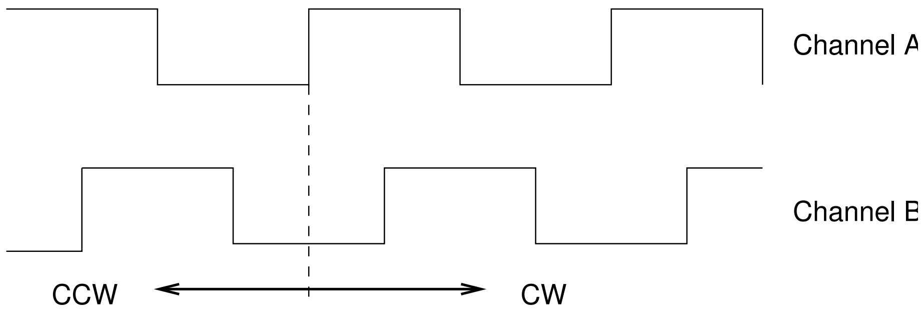


Figure 24: Encoder output

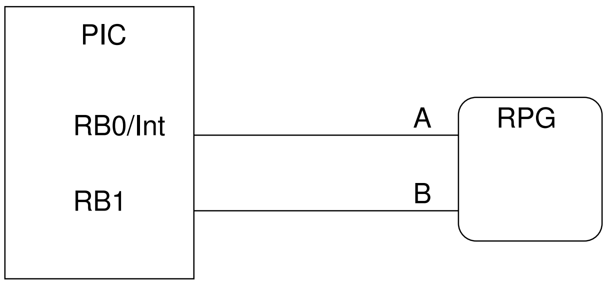


Figure 25: Connection between the PIC and RPG

Suppose the RPG is connected to the PIC as shown in Figure (25). Channel A is used to generate external interrupts at each rising edge and an thus an Interrupt Service Routine (ISR) can be used to determine the direction of rotation. The use of a RPG makes it possible to have additional features such as coarse/fine control of parameters using a single control knob, something which is not possible if a potentiometer is used. Assume that the RPG is being used to control the voltage of a voltage supply. When the control knob is turned quickly, the voltage is to be changed in 1.0V

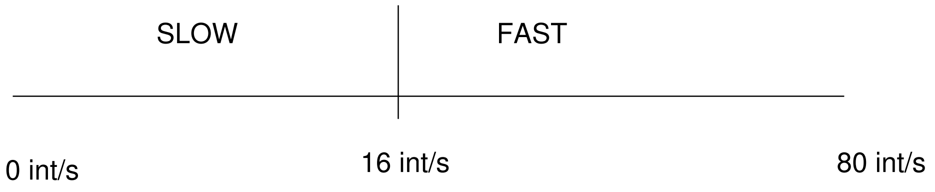


Figure 26: Differentiation between fast and slow speeds

steps and when the control knob is turned slowly the voltage steps are 0.1V. Therefore in addition to knowing whether to increase or decrease the voltage, the speed of rotation must also be known. The maximum rate of rotation of the RPG is 2.5 rev/sec and if it generates 32 cycles per revolution, there will be 80 interrupts generated in one second or one every 12.5 ms. This represents the high limit of rotation with the lower limit being 0. A reasonable threshold between fast and slow is 0.5 rev/sec, which translates to 16 interrupts/sec or one every 62.5 ms (Figure (26). The built-in timer of the PIC can be used to detect whether the RPG is turning fast or slow by having the timer timing out when turning slowly and not timing out when turning fast. The test that determines this is as follows:

```
; This is called at every interrupt
if ( TMR0 has timed out )
    then RPG is turning slowly
else
    RPG is turning fast
Reset TMR0
```

It is important to reset TMR0 back to zero at every interrupt because it is the time between interrupts that is of interest. Now it is necessary to setup the timer and prescaler such that it times out in approximately 62.5 ms. A setting of 1:256 in the prescaler, when the PIC is driven by a 4MHz clock signal, will cause timeouts every 65.5 ms which is reasonably close enough to 62.5 ms. The main program now looks like the following:

```
        ORG     0x000
        goto    Init
        ORG     0x004
        goto    ISR
; The Init section configures TMR0, the prescaler and PORTB
Init
        bsf     STATUS,RP0  ; switch to Bank 1
        movlw   B'11000111' ; bit 7 pull up disable
                            ; bit 6 interrupt on rising edge
                            ; bit 5 use internal clock for TMR0
                            ; bit 3 assign prescaler to TMR0
                            ; bits 2,1,0 prescaler scaler set at 1:256
        movwf   OPTION_REG
        bsf     TRISB,1     ; Line 1 of PORTB set as input
        bcf     STATUS,RP0  ; back to Bank 0
        bsf     INTCON,INTE ; enable external interrupt
        bsf     INTCON,GIE  ; enable Global interrupt
main    ....
        goto    main
```

The ISR now implements the algorithm

```
ISR     movwf   w_temp        ; save W and STATUS
        movf    STATUS,W      ;
        movwf   status_temp  ;

Poll    btfsc   INTCON,INTF
        goto    RPG            ; test for only one interrupt source
        movf    status_temp,W ; restore W and STATUS
        movwf   STATUS         ;
        swapf   w_temp,F       ;
        swapf   w_temp,W       ;
        retfie                 ; return from interrupt

RPG     bcf     INTCON,INTF    ; clear the extern interrupt flag
        clrf    TMR0           ; reset the timer
        btfsc   INTCON,T0IF    ; check if timer overflowed
        goto    update_small   ; overflowed... turning slowly
        goto    update_large   ; turning fast

update_small
        bcf     INTCON,T0IF    ; reset the overflow flag
        btfss   PORTB,1        ; check channel B of RPG
        goto    inc_small      ; low? clockwise
        goto    dec_small      : high? counterclockwise
update_large
        btfss   PORTB,1        ; check channel B
        goto    inc_large
        goto    dec_large

inc_small
        ; increase the voltage by 0.1 V
        call    increase_voltage_small
        goto    Poll
dec_small
        ; decrease the voltage by 0.1 V
        call    decrease_voltage_small
        goto    Poll
inc_large
        ; increase the voltage by 1.0 V
        call    increase_voltage_large
        goto    Poll
dec_large
        ; decrease the voltage by 1.0 V
        call    decrease_voltage_large
        goto    Poll
```

While the system as shown will work, much of the processing is being done in the ISR which can cause the ISR take too long to complete its processing. A better solution is to have the ISR determine what change in voltage is needed and have the main program implement the change of voltage. What is required now is a means of informing the main routine that a change is required. This can be accomplished through the use of flags encoded as bit positions in a file register e.g. `upd_m`. The meaning of the various bits are

Bit	Action
0	decrease small
1	decrease large
2	increase small
3	decrease large

The ISR would then determine what change is required, set the appropriate flag and exit. The main routine polls the variable, `upd_m`, for the requested change. The main routine then becomes

```
decS    equ     0
decL    equ     1
incS    equ     2
incL    equ     3

        org     0x000
        goto    Init

        org     0x004
        goto    ISR

Init
            ; Initialize TMR0, the prescaler and PORTB
            ; ... as before

main    movf    upd_m,F
        btfsc   STATUS,Z
        goto    main            ; if zero keep polling
        btfsc   upd_m,decS
        call    decrease_voltage_small
        btfsc   upd_m,decL
        call    decrease_voltage_large
        btfsc   upd_m,incS
        call    increase_voltage_small
        btfsc   upd_m,incL
        call    increase_voltage_large
        goto    main
```

The ISR can now execute much faster with the removal of the subroutine calls to change the voltage.

```
ISR     ; save W and STATUS etc

Poll    btfsc   INTCON,INTF
        goto    RPG            : test for only one interrupt source
        ; restore W and STATUS
        retfie                 ; return from interrupt

RPG     bcf     INTCON,INTF    ; clear the extern interrupt flag
        clrf    TMR0           ; reset the timer
        clrf    upd_m          ; clear prior to setting flag
        btfsc   INTCON,T0IF    ; check if timer overflowed
        goto    update_small   ; overflowed... turning slowly
        goto    update_large   ; turning fast

update_small
        bcf     INTCON,T0IF    ; reset the overflow flag
        btfss   PORTB,1        ; check channel B of RPG
        goto    inc_small      ; low? clockwise
        goto    dec_small      : high? counterclockwise

update_large
        btfss   PORTB,1        ; check channel B
        goto    inc_large
        goto    dec_large

inc_small
        ; increase the voltage by 0.1 V
        bsf     upd_m,incS
        goto    Poll

dec_small
        ; decrease the voltage by 0.1 V
        bsf     upd_m,decS
        goto    Poll

inc_large
        ; increase the voltage by 1.0 V
        bsf     upd_m,incL
        goto    Poll

dec_large
        ; decrease the voltage by 1.0 V
        bsf     upd_m,decL
        goto    Poll
```

Frequency measurement

Another use of the timer/counter is to measure frequencies. For this purpose, a number of cycles must be counted over a known time. The timer/counter cannot be used to generate this time reference, so an external time reference must be used. For simplicity it can be assumed that an external time reference exists with a base of 1 sec, this can be fed into the external interrupt line so that interrupts will be generated every second. At the first interrupt, the counter will be cleared and at the next interrupt the value in the counter will be the frequency. If the counter is used without the prescaler, the range of frequencies would be from 0 to 255 Hz. If the prescaler, with a scale of 1:256, is used, the range of frequencies is now 0 to 65280 Hz, but since the prescaler cannot be read there can be an error in that measurement which can be as much as 255 cycles. This error can be significant if the same 1:256 scale is used for low values of frequency. Therefore it is better to use an additional file register to count the number of times the counter has overflowed and keep the prescaler at low values e.g. 1:2 where the error is low or even not to use the prescaler at all.

The ISR responds to two sources, counter overflow and time reference rising clock edge. When the counter overflows the file register, say `ncycles` is incremented by one. The actions for the time reference interrupt were given above. For flexibility, the measuring system can be assigned four states, so that measurements can be taken at specified intervals or on a continuous basis.

- Idle
- Measurement requested
- Measurement in progress
- Measurement complete

These four states can be encoded as bits within a file register as

00	Idle
01	Measurement requested
11	Measurement in progress
10	Measurement complete

The main routine as show below requests a measurement and loops until it is done and can be easily changed to take continuous measurements. At the end of the measurement the frequency is in the 16-bit variable (Freq_hi, Freq_lo).

```
DONE    equ     B'00000010'

        org     0x000
        goto    Init

        org     0x004
        goto    ISR

Init    bsf     STATUS,RP0  ; switch to Bank 1
        movlw   B'11100000' ; bit 7 pull up disable
                            ; bit 6 interrupt on rising edge
                            ; bit 5 use external clock for TMR0
                            ; bit 4 increment on rising edge
                            ; bit 3 assign prescaler to TMR0
                            ; bits 2,1,0 prescaler scaler set at 1:2
        movwf   OPTION_REG
        bcf     STATUS,RP0  ; back to Bank 0
        bsf     INTCON,INTE ; enable external interrupt
        bsf     INTCON,T0IE ; enable overflow interrupt
        bsf     INTCON,GIE  ; enable Global interrupt


main    bsf     Flags,0     ; request a single measurement

Wait    movlw   DONE        ; pattern to check for in W
        xorwf   Flags,W     ; xor will give 0 if pattern found
        btfss   STATUS,Z
        goto    Wait        ; not found? loop
        clrf    Flags       ; go to idle mode, then show measurement
; the frequency = Ncycles*256*prescale + TMR0*prescale
        clrf    Freq_hi
        rlf     Freq_lo,F   ; shift left by 1 i.e. mult by 2 (prescale)
        rlf     Freq_hi,F   ; excess into hi byte
        movf    Ncycles,W   ; add the rest
        addwf   Freq_hi,F
        rlf     Freq_hi,F   ; times 2 again (prescale)
        call    Display_frequency
        end
```

The ISR has to respond to two sources, the external interrupt and the overflow interrupt from the counter.

```
INPROG  equ     B'00000011'

ISR     ; save W and STATUS


Poll    btfsc   INTCON,INTF
        goto    Measure
        btfsc   INTCON,T0IF
        goto    Counter
        ; restore W and STATUS
        retfie

Measure bcf     INTCON,INTF
        btfss   Flags,0
        goto    Poll
        btfss   Flags,1
        goto    Start_read
        goto    Stop_read

Start_read
        clrf    TMR0        ; initialize counter and Ncycles
        clrf    Ncycles
        bsf     Flags,1     ; set to 11
        goto    Poll

Stop_read
        movf    TMR0,W      ; save contents
        movwf   Freq_lo
        bcf     Flags,0     ; set to 10
        goto    Poll

Counter bcf     INTCON,T0IF ; clear interrupt flag
        movlw   INPROG
        xorwf   Flags,W     ; check if measurement in progress
        btfsc   STATUS,Z
        incf    NCycles,F   ; yes? increment
        goto    Poll
```

Review Exercises

1. A student writes a PIC16F877 ISR in which he checks for the TMR0 interrupt, the external interrupt, and the Serial RX interrupt **in that order**. He observes that if the TMR0 interrupt and the Serial RX interrupt occur at the **same time**, sometimes the Serial RX interrupt gets processed first. This is probably because:

   (a) Sometimes the interrupt signals occur when the ISR is already handling an external interrupt.

   (b) Sometimes the interrupt handlers, interrupt each other.

   (c) Sometimes TMR0 malfunctions and overflows.

   (d) Sometimes the Serial peripheral receives alot of data.

   (e) Sometimes the PIC16F877 malfunctions and resets itself.

2. A rotary pulse generator is connected to the INT line of a PIC16F877. If the generator produces pulses such that the INTF flag goes high multiple times while executing the handler, we can ensure that the program does not get stuck in the ISR by any of the following means EXCEPT:

   (a) moving lengthy sections of code from the handler to the main program.

   (b) clearing the INTF bit just before leaving the handler

   (c) clearing the INTE bit before leaving the handler

   (d) clearing the GIE bit before leaving the handler

   (e) none of the above

3. You are attempting to stop ALL interrupts from interfering with a section of your code. Which ONE of the following code wrapper snippets will do the job:

(a)
```
redo    bsf     INTCON, GIE
        btfsc   INTCON, GIE
        goto    redo
        ......
        bcf     INTCON, GIE
```

(b)
```
redo    bcf     INTCON, GIE
        btfsc   INTCON, GIE
        goto    redo
        ......
        bsf     INTCON, GIE
```

(c)
```
redo    bcf     INTCON, PEIE
        btfsc   INTCON, PEIE
        goto    redo
        ......
        bsf     INTCON, PEIE
```

(d)
```
redo    bcf     INTCON, GIE
        btfss   INTCON, GIE
        goto    redo
        ......
        bsf     INTCON, GIE
```

(e)
```
redo    bsf     INTCON, GIE
        btfss   INTCON, GIE
        goto    redo
        ......
        bcf     INTCON, GIE
```

4. The Working register and status register must be stored/restored in any ISR. Which of the following code fragments, demonstrate how this is correctly done from ANY data bank?

(a)
```
_W      EQU     0x21
_STATUS EQU     0x22
isr     movwf   _W
        movf    STATUS,W
        clrf    STATUS
        movwf   _STATUS
        .
        .
        movf   _STATUS,W
        movwf   STATUS
        movf    _W
```

(b)
```
_W      EQU     0x71
_STATUS EQU     0x22
isr     movwf   _W
        movf    STATUS,W
        clrf    STATUS
        movwf   _STATUS
        .
        .
        movf   _STATUS,W
        movwf   STATUS
        swapf   _W,F
        swapf   _W,W
```

(c)
```
_W      EQU     0x71
_STATUS EQU     0x72
isr     movwf   _W
        movf    STATUS,W
        clrf    STATUS
        movwf   _STATUS
        .
        .
        movf   _STATUS,W
        movwf   STATUS
        movf    _W
```

(d)
```
_W      EQU     0x21
_STATUS EQU     0x72
isr     movwf   _W
        movf    STATUS,W
        clrf    STATUS
        movwf   _STATUS
        .
        .
        movf   _STATUS,W
        swapf   _W,F
        swapf   _W,W
```

(e)
```
_W      EQU     0x21
_STATUS EQU     0x22
isr     movwf   _W
        movf    STATUS,W
        clrf    STATUS
        movwf   _STATUS
        .
        .
        movf   _STATUS,W
        movwf   STATUS
        swapf   _W,F
        swapf   _W,W
```

5. Consider a PIC application that has four interrupt sources, with T_1 =10 cycles, T_2 =20 cycles, T_3 =40 cycles and, T_4 =80 cycles

   (a) Using the inequality

   $$T_1 + T_2 + \cdots + T_n + O_n \leq \text{ worst case time} < T_1 + T_2 + \cdots + T_n + 2O_n, \qquad (2)$$

   determine the worst-case response of the CPU to any of these interrupts given the polling scheme where subroutine calls are used for each handler.

   (b) What does this mean for the minimum period between any of these interrupts?

6. Consider the timing diagrams from Figures (8), (9) and (10) in the notes titled Input/Output.

   (a) Why does the worst-case time for source 1 begin at the end of the time labeled P_1 instead at the time before this when the CPU actually stops execution of the main program?

   (b) Why does the worst-case time for source 2 begin at the end of the time labeled P_2 instead at the time before this when the CPU actually stops execution of the main program?

   (c) Why does the worst-case time for source 3, the lowest priority interrupt, begin when the CPU actually stops execution of the main program rather at the end of P_3 or P_1?

7. Using the new polling structure for ISRs i.e. using `goto` for the handlers instead of `call`, consider an application with four interrupt sources having times exactly like those in Problem 5 and priority T_1, T_2, T_3, and T_4.

   Draw the worst-case timing diagram for the response of the CPU to interrupts from source 1. Also determine the time TP_1, the minimum time between source 1 interrupts such that the CPU will assuredly execute source 1's handler and be ready to handle another source 1 interrupt. Repeat this exercise for source 3.

8. When an interrupt occurs, the global interrupt enable bit, GIE, is automatically cleared by the CPU. Sometimes it is necessary to disable interrupts for a short time in the main program in order to execute a short sequence that can give an incorrect result if interrupted and then re-enable the interrupt after the sequence.

   A problem could arise if the instruction to disable the interrupts is being executed at the exact moment that an interrupt occurs. Describe the effect if the instruction to disable the interrupts is indeed executed, but the interrupt is acknowledged and handled by the CPU before the program instruction is able to take effect.

9. The code presented for frequency measurement should produce a 16-bit estimate for the switching frequency of the signal connected to Timer0. Comment on the accuracy of this measurement. Hints: Check the minimum duration change detected on the Timer0 input, and the ISR execution time.

10. Challenge:

    (a) Using the code developed for the Rotary Pulse Generator, modify it to increment a 2-byte variable, `Amplitude` if the RPG has been rotated CW and to decrement it if the rotation is CCW. Do not increment past 65,535 or decrement past zero.

    (b) Modify the solution to Problem (10a) to change the variable `Amplitude` by one-eighth of its value or by one count, whichever is larger when fast turning of the RPG is detected.

Assignment D [39]

ID# ______

The notes contain two sets of code for the RPG (Rotary Pulse Generator). In the first set (page 13) the subroutines are called from within the ISR, and in the second set (page 14) the subroutines are called from the main program. You should presume that EACH subroutine: takes n instruction cycles to execute, requires p instructions, and uses f file registers.

1. Determine how many program instructions and file registers are required for each version of the code. Show all reasoning. *6 marks*
2. In each case, presuming that the ISR completes before the next interrupt occurs, determine the time it takes to from the occurrence of an interrupt event to: *12 marks*
   (a) the completion of the ISR
   (b) the completion of the relevant subroutine
3. In each case, determine the largest value of n which will allow: *6 marks*
   (a) the ISR to complete BEFORE the next interrupt event occurs.
   (b) the relevant subroutine to complete BEFORE the next interrupt occurs
4. Explain (for each case), whether the RPG will continue to function correctly if another interrupt event occurs before:
   (a) the previous interrupt event has been *handled*.
   (b) the previous interrupt event has exited the ISR.

   Where the RPG may malfunction, suggest amendment(s) which will correct the problem. *6 marks*
5. In the second case, a variable called `updm` is used to communicate the ISR result to the main routine. There is a flaw in the main routine, in that `updm` needs to be cleared before the last line of the main routine i.e. `goto main`. In your own words, explain why this is so. *2 marks*
6. The correction introduced in the last question solves one problem, and introduces another according to where the main routine is interrupted. Find, and explain the problem, and suggest a solution. *4 marks*
7. Based on the above, which RPG program would you prefer to use? In your own words, justify your decision. *4 marks*

This assignment is worth 2% of your ECNG2006 mark. It contains a total of 40 marks.

[39] Students are advised that there is no online version of this assignment, and that electronic submissions will not be accepted without the explicit permission of the course lecturer.

Unit 23 A PIC16F877 program samples the value on the upper bits of `PORTB<7:4>` whenever a change in those values is detected. The most recently sampled value is output to `PORTD` whenever a button press is detected at `PORTB<0>`. Interrupts are ONLY generated by activity on `PORTB`. The main loop polls the variable `MyFlags` and updates `PORTD` as required.

```
Loop      btfss     MyFlags,0  ; If the button was pressed
          goto      Loop
          bcf       MyFlags,0
          movf      SampleA,W  ; Copy the new value to PORTD
          movwf     PORTD
          goto      Loop
```

The interrupt handler fragments are executed by the main ISR routine, and return to the main ISR routine to retest all interrupt flags, on completion. RBIF is tested for before INTF.

```
RBIFInt   bcf       INTCON,RBIF; if input value changed, store it
          movf      PORTB,W
          andlw     0xF0
          movwf     SampleA
          swapf     SampleA,F  ; place input bits PortB<7:4> in SampleA<3:0>
          goto      Poll       ; back to polling sequence
INTFInt   bcf       INTCON,INTF; if button pressed set the flag
          bsf       MyFlags,0
          goto      Poll       ; back to polling sequence
```

Write the name of the ISR handler you have been assigned here ________________. The handler takes ________ cycles, and has a worst case response time of ________ cycles.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - understand the implications of (nested/multiple) interrupts for execution times, and the need for context saving.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

24 μP interface/timing issues

At the end of this unit the student will be able to:

- *interpret and recognize the implications of, microprocessor timing and interfacing requirements.*
- *determine the noise, voltage, and current considerations due to microprocessors and peripheral fan-in/out.*

In the world of the ideal μP, inputs and outputs switch instantly; outputs can source/sink infinite amounts of current, and inputs never sink/source current. Unfortunately not one of these statements is true. When designing a μP-based system, it is therefore important that we not only check that our design are functionally correct, we must also check that the components are capable of achieving the demands being made, and will do so reliably. In practice this means reading electrical characteristics, and timing diagrams, and performing worst case analyses. (Arnold 2001; Ch. 3)

Timing diagrams

> " The most important timing specifications for interfacing components to a bus-oriented design are:
>
> - Rise/fall time
> - Propagation delay time
> - Setup time
> - Hold time
> - Tri-state enable and disable delays
> - Pulse width
> - Clock frequency
>
> " – (Arnold 2001; Ch.3)

Rise and fall times refer to the time it takes for a signal to move from logic high to logic low, and vice versa. These have maximum acceptable values (inputs) and minimum achievable values (outputs).

Propagation delay time refers to the time it takes between a signal changing on one line, to a response being observed on another line. This may not necessarily be the same for high-low and low-high transitions.

Setup and hold times refer to the times that the signal must be stable, pre/post a transition on another signal line (generally clock).

Tri-state circuitry has an inherent delay between the change of input signal, and the change of output voltage.

Some signals must be held for a time within a fixed range, in order to communicate a particular value. The time from leading to falling edge is the pulse width.

Clock frequency determines the rate at which micro-operations are performed (Q cycles), and hence determines how long the signal must be held in order to ensure that the relevant micro-operation was performed.

It is important to ascertain whether any of the timing parameters are being violated, as transient "glitches" can result if timing is violated.

Electrical characteristics

The device pin may be modelled as a load with input and output resistance and capacitance. It may also be modelled as a voltage source with a current limitation. It follows that any signals passing between devices will be affected by the electrical characteristics of the pins at either side of the connection.

Noise margin analysis – input voltage limits should not lie "outside" of output voltage limits.

AC load analysis – the sum of the input capacitances should not exceed the load capacitance.

DC load analysis – current sourced/sunk should not exceed the specified maximum values.

Varied Behaviour

Temperature, pressure and operating frequency can affect the electrical characteristics and the operating voltage ranges. Curves are normally provided to allow you to predict how the device behaviour will change as these deviate from nominal values.

Compensating Strategies

If your analysis reveals a problem, a number of strategies can be employed to fix it.

- Signals which do not rise/fall fast enough may be pulled high/low using a resistor – this will reduce the transition times to this default state.
- Clock frequencies (generally externally supplied) can be reduced/increased in order to allow more time between events.
- Handshaking or other arbitration protocols may be used to ensure that no contention occurs on signal lines.
- Loading/noise margin issues can generally be fixed by using appropriate logic buffers.

Additional strategies appear below.

" Forms of digital signal degradation.

Signal degradation	Possible solution at input
Slow edge	Schmitt trigger
Voltage attenuation	Schmitt trigger
EM interference – data error	Analogue filter + Schmitt trigger, digital filter
EM interference – ... equipment damage	Current limiting resistor + diode opto-isolate, other voltage clamp
Switch bounce	Debounce in hardware/software
Earth differential	Opto-isolate

"

– (Wilmhurst 2001; Table 9.1, pg 253)

Reference Notes for the PIC16F877

Output Pin Load resistance	R_L	464Ω
Output Pin Load capacitance	C_L	50 pF
Input Pin capacitance	C_{in}	5 pF
Max. External Clock rise/fall time	T_{os}	25ns
Max. Port output rise/fall time	T_{io}	40ns
Max. Delay port output	$T_{os}H2_{io}$	255ns
Min. Input Hold time	$T_{os}H2_{io}I$	100 ns
Max. current sunk/sourced by output (ignoring totals)		20mA
Max. input leakage current	I_{IL}	$\pm 1\mu$A
Output Low Voltage	V_{OL}	0.6V
Output High Voltage	V_{OH}	$V_{DD} - 0.7$ Volts
Max. Input Low Voltage	V_{IL}	$0.15V_{DD}$ Volts
Min. Input High Voltage	V_{IH}	$0.25V_{DD} + 0.8$ Volts

Machine cycle (4Q)	Q1	Q2	Q3	Q4
(in 1 instruction cycle)				
Pipeline – Fetch stage (Interrupt)	PC on bus	Check Interrupts Disable GIE	PC=0x04 Latch Stack	IR=0x00
Pipeline – Fetch stage (No Interrupt)	PC on bus	Check interrupts	Increment PC Trigger Program memory	Latch IR
Pipeline – Execute stage	Decode IR	Register Read	Execute/Process	Register Write
Peripheral actions	Output signal	Input signal		Set flags (interrupt/status)

All voltages must lie between V_{dd} and V_{ss}. All ratings taken at 25°.(PIC 2001)

Review Exercises

Use the parameters for the PIC16F877 supplied in the notes, to answer the following questions.

1. A PIC16F877 powered at 4.9V is required to switch a digital signal line. The load at the end of the line sinks 40mA, and has a load capacitance of 0.1 pF. The load reads a logic high at V=2.5V and a logic low at 1.8V. Required rise/fall times for the signal are comparable to the nominal values for the PIC16F877 I/O pins. Label the following statements as True/False:

   (a) A pull-up resistor may be used to maintain the high bus voltage at appropriate levels.

   (b) The PIC16F877 pin will be able to supply the load current.

   (c) The interface can be implemented if we use a buffer, or switch a transistor in order to drive the load.

   (d) The rise and fall times for the signal voltage will be greater than those cited in the data-sheet, because the load capacitance is too great.

   (e) The signal voltage will be less than the nominal values, and therefore may not meet the load requirements.

2. A PIC16F877 with a 4MHz oscillator is used with an **external** A/D converter which responds with the latest digital reading on `PORTD<3:0>` $1\mu s$ after the write line on `PORTD<4>` is raised. Assuming that `PORTD` has already been appropriately configured, which piece of code will successfully read the converter in the least instruction cycles?

(a)
```
bsf   PORTD,4
movf  PORTD,W
andlw 0x0F
movwf VAL_IN
```

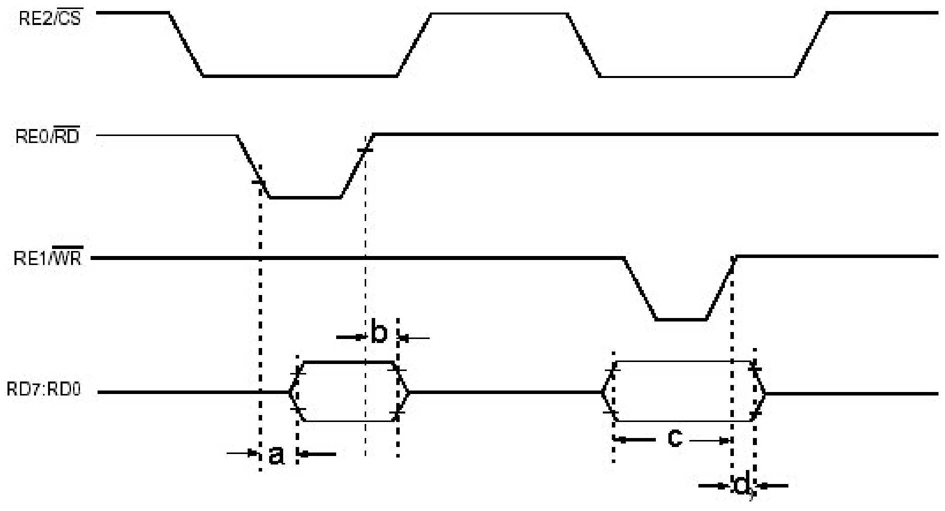
(b)
```
bsf   PORTD,4
nop
movf  PORTD,W
andlw 0x0F
movwf VAL_IN
```

(c)
```
bsf   PORTD,4
movf  PORTD,W
andlw 0x0F
movf  VAL_IN
```

(d)
```
bsf   PORTD,4
movlw 0x0F
andwf PORTD,W
movwf VAL_IN
```

(e)
```
bsf   PORTD,4
movf  PORTD,W
nop
andlw 0x0F
movwf VAL_IN
```

3. The timing diagram is for the PIC16F877 Parallel Slave Port.



Which time reflects the **setup** time for data which needs to be written to the PIC16F877?

(a) a
(b) b
(c) c
(d) d
(e) none of the above

4. A PIC16F877 with a 4MHZ oscillator is used to communicate with a device which requires:

- enable line is held high for at least 3.5μs,
- draws 10mA from the enable line, and has an input capacitance of 3pF
- interprets 0.5 V as input-low, and 2.1 volts as input-high

Which piece(s) of code (if any) will achieve the appropriate timing for the enable line?

(I) `bsf PORTA,E`
`nop`
`bcf PORTA,E`
	(II) `bsf PORTA,E`
	`nop`
	`nop`
	`bcf PORTA,E`
		(III) `bsf PORTA,E`
		`nop`
		`nop`
		`nop`
		`bcf PORTA,E`

(a) none of these will achieve the required timing
(b) (III) only
(c) (III) and (II) only
(d) (III), (II) and (I)

5. A PIC16F877 with a 4MHZ oscillator is used to communicate with a device which requires:
   - enable line is held high for at least $3.5\mu s$,
   - draws 10mA from the enable line, and has an input capacitance of 3pF
   - interprets 0.5 V as input-low, and 2.1 volts as input-high

   Once the timing is appropriate, will the communication be successful, and if not why not?

   (a) Yes, communication will be successful.
   (b) No, we have violated the voltage restrictions.
   (c) No, we have violated the current restrictions.
   (d) No, we have violated the capacitance restrictions.
   (e) No, we have violated two or more of the restrictions.

6. What happens when the load capacitance of an IC signal pin is exceeded?

   (a) The signal voltage will never get to the expected maximum/minimum values.
   (b) On transition, the signal will exceed the specified rise/fall times.
   (c) The signal pin will not be able to deliver the specified currents.
   (d) The IC will stop working, and there will be no signal at all.
   (e) None of the above.

7. A PIC16F877 is to be interfaced to a device which has the following electrical characteristics:

Max. current sunk/sourced by output (ignoring totals)	$-10\mu A$ / $5\mu A$
Max. input leakage current	$\pm10\mu$A
Output Pin Load capacitance	25 pF
Input Pin capacitance	10 pF
Max. Output Low Voltage	1V
Min. Output High Voltage	3.5 Volts
Max. Input Low Voltage	2.5 Volts
Min. Input High Voltage	4 Volts

All ratings taken at 25°.

   Which one of the following statements (if any) is NOT a correct analysis:

   (a) DC analysis PIC16F877 output to device input: OK $\pm10\mu$A $<$ $\pm$20mA
   (b) DC analysis device output to PIC16F877 input: OK $\pm1\mu$A $< -10/+5\mu$A
   (c) AC analysis PIC16F877 output to device input: OK $10pF < 50pF$
   (d) AC analysis device output to PIC16F877 input: OK $5pF < 25pF$
   (e) none of the above

8. Two PIC16F877 microcontrollers (A and B) are to be interfaced with each other.

   (a) What is the maximum number of inputs on B, which can be driven from a single output on A? Show your working. Hint: Consider current load and load capacitance on A's output pin.

   (b) In your own words, explain what happens if the load capacitance on A's output pin is exceeded.

   (c) A and B communicate across a single signal line. A needs to output a high for 10 time units, and a low for 20 time units. We write a subroutine `Output` which we call with the appropriate value (high/low) already in W. The sequence of instruction operations is as follows:

```
    ...
Output  movf  Port          ;A places a value on the signal line (output)
        btfss Port, SigBit  ;A tests the value it just put out
        call  delay10       ; and delays for an appropriate time
        call  delay10
        return
    ....
```

   When this code is executed on the simulator it works. In each of the following scenarios, identify the cause of this problem, and suggest a solution.

      i. When this code is executed on the PIC with the pin open, the oscilloscope shows that it stays high for 10 time units and low for 20 time units. When B is connected, the oscilloscope shows that it stays high for 10 time units and low for 10 time units.

      ii. When this code is executed on the PIC with the pin open, the oscilloscope shows that it stays high for 10 time units and low for 10 time units.

9. You are given a device whose I/O pins have the following characteristics:

Max. current sunk/sourced by output (ignoring totals)	$-10\mu A$ / $5\mu A$
Max. input leakage current	$\pm10\mu$A
Input Pin capacitance	10 pF
Output Low Voltage	1V
Max. Input Low Voltage	2.5 Volts
Min. Output High Voltage	3.5 Volts
Input signal hold time (after Write Line)	$4\mu s$
Input signal setup time (before Write Line)	$2\mu s$
Output signal response time (after Write Line)	$2\mu s$

Your task is to interface this device to a PIC16F877 with a 4MHz clock. The pins from the device are connected directly to those of PortD on the PIC16F877, and the code to access the device is as follows:

```
Read_in     clrf   PORTD             ; set the Write Line Low
            movf   PORTD,W
            andlw  0b00001100        ; mask out all but the recv bits
            movwf  VAL_IN
            return

Send_out    movf    VAL_OUT,W
            andlw   0b00000011        ; mask out all but the send bits
            iorlw   0b00010000        ; set the Write Line high
            movwf   PORTD
            nop
            clrf    PORTD
            return
```

Perform timing, noise, AC and DC analyses, and suggest corrective actions where appropriate.

10. Challenge: The specifications for the PIC16F877 given in the notes are for a nominal temperature of 25°. Explain (in your own words) how you would cater for operation at 40°. (Hint: you may wish to check the data-sheet for temperature curves.)

Tutorial Exercise 9AOffline Version[40] ID#

1. You are asked to interface an HD77380-compatible LCD display (controller) with a PIC16F877 which is powered at $5V$, and uses a 20MHz oscillator. The electrical and timing characteristics for the PIC16F877 as well as the bus timing characteristics for the HD77380 are given in your "green" databook. In addition you are given the following electrical information for the HD44780:

Item	Symbol	Min	Typ	Max	Unit
Input high voltage	V_{IH}	2.2		V_{CC}	V
Input low voltage	V_{IL}	0.3		0.6	V
Output high voltage	V_{OH}	$0.9V_{CC}$		-	V
Output low voltage	V_{OL}			$0.1V_{CC}$	V
Output Current Sunk/Sourced	I_{OH}	-	-	0.04	mA
Input leakage current	I_{LI}	1		1	μA
Input Pin capacitance	C_{IN}	-	10	-	pF
Capacitive Load on Output Pins	C_{IO}	-	-	25	pF

and the code snippet used to "trigger" an LCD character send is

```
bcf      LCD_CNTL, LCD_RW
bcf      LCD_CNTL, LCD_RS
movwf    LCD_DATA
bsf      LCD_CNTL, LCD_E
bcf      LCD_CNTL, LCD_E
```

Perform timing, noise, AC and DC analyses. State any assumptions that you make, note any missing information required, and suggest corrective actions if appropriate. *16 marks*

[40] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 24 Write the letter you have been assigned here________.

A Timing analysis

B AC analysis

C DC analysis

D Noise analysis

Perform the analysis you have been assigned for the system described in Review Question 9.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - interpret and recognize the implications of, microprocessor timing and interfacing requirements.
  - determine the noise, voltage, and current considerations due to microprocessors and peripheral fan-in/out
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

25 Power/Signal issues

At the end of this unit the student will be able to:

- *determine the noise, power considerations due to microprocessors and peripheral fan-in/out*
- *determine how different logic families (as well as inverted voltage logic) may be used in conjunction in a design.*

The previous unit looked at some of the problems which arise when the functional design falls outside of voltage, current, load and timing specifications for the components. This unit will investigate factors which are specific to a particular implementation of the design.

Power source

All of the preceding discussions assumed an ideal power source, which provides a constant supply of the ideal voltage level, and can source an infinite amount of current. In practice, these assumptions are not true.

Power conditioning refers to production/maintenance of the required voltage. It incorporates voltage regulation (maintaining output DC voltage as long as the input voltage lies within a specified range), voltage conversion (stepping a DC voltage up/down), and voltage inversion (changing polarity of DC signal). Special circuitry may be used to achieve these functions (Horowitz and Hill 1989), prior to feeding the microprocessor based system. It is advisable to check which methods are being used to generate the supplied voltages, and whether they will meet load requirements, and noise tolerances.

Power supply supervision refers to the pro-active monitoring and/or control of the power supply for the microprocessor based system. Typical functions include control of battery charging, switching between power supplies, monitoring of the supply voltage, switching power on/off.(Wilmhurst 2001; Ch. 10) These functions may be directly carried out by the microprocessor (watchdog timer, brown-out detection), facilitated by a microprocessor controlled peripheral, or may be done entirely independently of the microprocessor-based system. The decision about which method to use (if any) will depend on the application requirements and the consequences of power failure.

Low power circuits, are those explicitly designed to minimise the amount of current drawn (thereby prolonging the run time on a battery-supply, or reducing the current requirements for a mains-converter).(Horowitz and Hill 1989)

> “ the guidelines for minimising power consumption in CMOS circuits, loosely in order of descending importance, are as follows:
>
> - define all inputs clearly as logic 0 or 1
> - minimise clock frequencies
> - minimise the power supply voltage
> - ensure fast logic transitions
> - minimise load and interconnection capacitances
>
> ”–(Wilmhurst 2001; pp 312–313)

This can be achieved through selection of components for their low power characteristics, switching off unused circuitry, and designing interfaces to use the widest possible noise/load margins.

Routing Wires/Tracks

Load capacitance and resistance are not just determined by the device at the end of the line, the wire connecting the devices also has an effect on the signal behaviour.

Ground bounce, occurs when large current spikes are returned through the common ground line. The effects can be reduced by using a larger conductor or track to reduce the resitance/inductance of the ground line, and by the use of decoupling/bypass capacitors.(Horowitz and Hill 1989; Ch.9).

Where a *mechanical connection* needs to be made between two conductors, it should have as much surface exposed to contact as possible, in order to reduce the contact impedance to a minimum.

All efforts should be made to keep signal wires away from "known" noise sources, either by passing them elsewhere, or by *shield*ing the noise source/signal wire. A single shield may be used for multiple signal wires.

When multiple signals physically run along parallel (proximate) wires/tracks (which happens in interfacing many devices), it is likely that they will be exposed to the same EM noise, therefore a *differential signal* will always be received correctly; the disadvantage is that more connections will be needed to send a single signal.

Interfacing different logic families

> "The three things that can keep you from connecting any pair of logic chips together are (a) input logic-level incompatibility (b) output drive capability, and (c) supply voltages "–(Horowitz and Hill 1989; Section 9.03)

Where logic families use the same supply voltages, and the input logic levels are compatible (or can be made so with a pull-up resistor) there may be reduced fan-in/out for the devices. Further, while the logic levels are ok, the fast edge-triggered devices may not respond (or may respond multiple times – ringing) to slow-moving edges. Where they cannot be made compatible, *level translators/shifters* (Wakerly; Ch. 3) must be used. These are specialised IC's which perform level conversion. (Horowitz and Hill 1989; see Table 9.2 Logic Family Connections)

Overall voltage/current requirements

The naive way to determine the maximum current and voltage requirements for the system is to find the sum/max of the requirements for all components. This does not take transient requirements into account. Be sure to verify that the power supply can handle sudden demands if your system requires them.

Review Exercises

1. Write definitions for all the italicised terms in this unit.

2. (a) Question from (Wilmhurst 2001; 10.5) "A microcontroller can operate from 3V to 6V. Under a particular set of operating conditions it acts to the power supply as a load of $1k\Omega$.

It is powered from a 5V supply and the total circuit decoupling capacitance is 220μF. If the supply is disconnected, approximately how much time does the microcontroller have to perform a controlled system shutdown if it draws its power during this time only from the decoupling capacitance? ” Hint: Draw the circuit model, look at the stored charge, and check to see how long the voltage takes to drop below 3V.

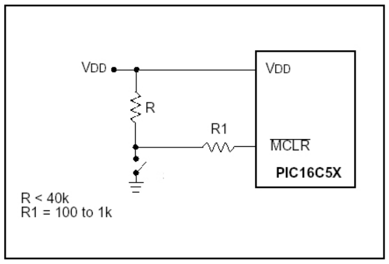
(b) For 2a, what is the approximate current drawn from the power supply during normal operation? Estimate the lifetime of an “ideal” 1Ah (amp-hour) 5V battery.

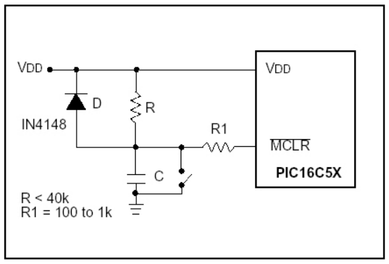
(c) For 2a, what is the **maximum** current drawn from the power supply during normal operation? Suggest a way to ensure that the power supply can provide the required maximum current.

3. Question from (Katzen 2003; 10.3) “The current consumption of a PIC operating at 4MHz and a V_{DD} of $5V$ is measured as $550\mu A$ with no loading at the port pins. What will be the current consumption if the device were to be clocked at 100 kHz and powered by a 4V supply?”(Hint: current consumption is proportional to operating frequency, and the square of the operating voltage. You may ignore the quiescent current consumed by the PIC when operating frequency is 0.)

4. Question from (Wilmhurst 2001; 10.6) “Each line of an 8-bit data bus is connected to four ICs, each input of which has an estimated capacitance to ground of $6pF$. Each line of the bus is switching between 0V and 3.3.V at a frequency of 2MHz. Estimate the power drain due to the activity of the bus alone.”

5. (a) Question from (Katzen 2003; 10.2) “In an attempt to reduce the current consumption of the circuit when reset, a student has used a $1M\Omega$ pull-up resistor in the Manual Reset circuit ... Why does the PIC not come out of reset?” What is the maximum value of pull-up resistor that should be used?





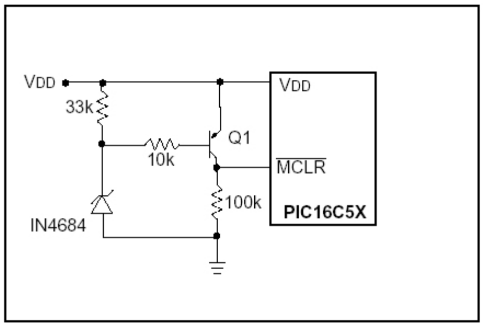
Based on (Mitra 1997): Manual Reset   Modified Manual Reset (slow rise time)

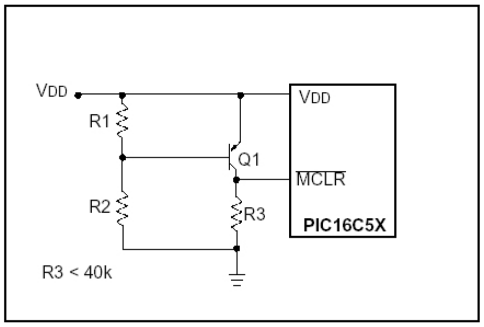
(b) In your own words, explain why the modified reset circuitry is needed for slow-rise time power supplies, and how it’s component values are chosen.

6. (a) Explain in your own words, the term “brown-out” as it relates to microprocessors/digital circuitry.

(b) Identify 1 possible consequence of brown-out occurring in microprocessors/digital circuitry.

(c) Question based on (Katzen 2003; 10.4, Fig 10.8). The circuits shown have been proposed as brown-out protection circuits. Discuss their operation, and how the component values are chosen.





Taken from (Mitra 1997)

(d) Can the brown-out protection circuitry be combined with the reset circuitry? Explain your answer.

7. (a) Can the PIC16F877 operate with all pins simultaneously sinking/sourcing their individual maximum currents? Explain your answer.

   (b) Suggest one way in which we can ensure that the current drawn by the PIC16F877 never exceeds specification.

8. A PIC16F877 is powered at 5V and connected to a 4MHz clock. A N.O. push-button switch changes an input (port pin 0) voltage from high to low when pressed. A counter connected to an output (port pin 4), will increment on the rising edge of the signal. The code for the PIC16F877 is as follows:

```
        ; ... appropriate configuration ...
        clrw
main    btfsc   PORTB,0
        goto    main
        xorlw   0b00010000
        movwf   PORTB
        goto    main
```

   (a) Describe what you would expect this code to do, and recommend any changes to make this more effective.

   (b) What is the maximum frequency at which the output pin will change?

   (c) For the circuit and code described/shown, predict anomalous behaviour you would expect (and suggest a solution) if:

      i. the pin-output capacitances (device) are large

      ii. the pin-input voltage (switch) varies between 0.7V and 1V

      iii. the PIC16F877 outputs are TTL level and the counter requires CMOS level inputs

      iv. the power supply voltage is disturbed by a motor connected to the same source causing a ±0.5V swing

      v. the tracks for the signal and power lines are not EM-isolated from each other

9. Challenge: Question based on (Wilmhurst 2001; worked example 10.3): A PIC16F877 microcontroller is used as part of a data logger system, which makes a measurement every second and then records two bytes of data in ***external*** RAM. It takes 50 instruction cycles to make this measurement. It requires 5 instruction cycles to write the data to the external RAM, and enables the external RAM chip for only 1 of these instruction cycles. When the system is powered at $V_{dd} = 5V$, with a 4MHz oscillator,

   - the PIC16F877 microcontroller draws $4mA$ when active, and $42\mu A$ when asleep.
   - the external memory draws $20mA$ when enabled, and $150\mu A$ otherwise.

   No other tasks are required of the microcontroller when logging. You may assume that the Watchdog Timer can be utilised without additional current consumption.

   (a) Describe how the sleep mode, and the Watchdog timer, could be used to minimize the current consumption of the system.

   (b) What would be the resulting current consumption?

10. Challenge: Question from (Wilmhurst 2001; 10.2) "What are the relative advantages of linear and switching voltage regulators/converters, particularly when applied to lower power applications?"

Tutorial Exercise 9BOffline Version[41]

ID# ______

1. In your own words, describe the phenomenon known as brown-out, and differentiate between brown-out, and power-on-reset. *6 marks*

2. An active low signal is connected to a PIC16F877 input pin, $PORTx < n >$. The input pin is read as '1' when a logic-high is on the input, and '0' when a logic-low is on the input. During operation, the signal is active approximately 30% of the time.

   Indicate whether the following statements are True or False. *4 marks*

   (a) According to the following snippet, the PIC16F877 will execute `CRoutine` when the signal is active.

   ```
   btfss   PORTx,n
   call    CRoutine
   ```

   (b) The device sending the signal will use logic-low when the signal condition is true.

   (c) Using an active-low signal will lower overall power consumption of the system.

   (d) Using an active-low signal will make the system less susceptible to input-noise.

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

[41] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 25 Write the letter you have been assigned here________.

A Power Conditioning

B Power Supply Supervision

C EM noise reduction

D Power budget

Your lecturer will propose a microprocessor-based system:________.
Discuss the requirements of the system for your assigned topic.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - determine the noise and power considerations due to microprocessors and peripheral fan-in/out
  - determine how different logic families (as well as inverted voltage logic) may be used in conjunction in a design
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

26 μP choice/comparison

At the end of this unit the student will be able to:

compare alternate microprocessors in terms of:

- *the instruction set,*
- *available enhancement features (such as cache, pipelining etc.)*
- *best performance, which may be facilitated by the instruction set and features.*

Unit 8 looked at some criteria for deciding between different microprocessor families and forms.This unit will look at a procedure for quantifying the importance of conflicting criterion, balancing tradeoffs and documenting the decision process.

"Device Selection Process

In selecting a device from a field of several devices, there is more to be considered than just the speed of the processor. Some factors, such as the availability of secondary suppliers may be an absolute requirement in some applications. In order to make a systematic evaluation and selection of the best alternative, the following method has proved to be valuable, particularly when the selection process must be documented and justified. The process consists of three major steps: eliminating the alternatives that are completely inappropriate, ranking the remaining options, and evaluating the adverse consequences of a catastrophic event. The three decision matrices are:

1) Pass/fail criteria for elimination of non-conforming alternative.

2) Weighted scoring of parametric values to rank options.

3) Consideration of adverse consequences, including their probability and severity.

The first matrix consists of a table with all the options on one axis and all the "must have" criteria on the other axis. Each criterion is checked off for each option. The second matrix consists of the surviving options from the first matrix on one axis of a table, and a list of quantitative measures on the other axis, along with a weighting factor for each measure, indicating its relative importance. Each option receives a weighted score allowing them to be ranked. Finally, each of the top ranking options is evaluated with respect to probability of occurrence" – (Arnold 2001; p. 207–208).

When ranking, items pay careful attention to make sure that items are ranked in the same order (i.e. high score == better ranking for all items OR low score == better ranking for all items).

It is possible to use scales for non-numeric or range-numeric data e.g. the price of a processor may be expressed using the scale (0- cheap 1- average 2- expensive), or as it's $ value.

Weights are assigned on a subjective basis according to the perceived relative importance of the ranking metrics. The weights do not necessarily need to sum to 1.

An example follows for a decision amongst 4 processors. They must have a particular instruction in their instruction set, and two other features (e.g. clock speed or a certain #/type of in-built peripherals). Only two processors meet these criteria. The remaining processors are then ranked in terms of cost (using a defined scale), performance, code size, and the presence of a non-essential peripheral (using a binary yes/no). In this example, a lower total reflects a higher rank.

Processor	InstructionA	FeatureB	FeatureC
i	$\surd$	$\surd$	$\surd$
j	$\surd$	$\times$	$\times$
k	$\surd$	$\surd$	$\surd$
l	$\times$	$\surd$	$\surd$

Criterion	Weighting	Processor i	Processor k
Cost (ranked 0(cheap), 1(ok), 2(expensive))	0.2	2	0
Performance (benchmark results/s)	0.6	4	2
Code Size (benchmark results/bytes)	0.1	2	4
Built-in A/D converter (0, present; 1: not present)	0.1	0	1
Total		3	1.7
Ranking		# 2	#1

Processor k is preferred unless there are extenuating circumstances such as supply problems.

Review Exercises

1. You are told to choose a microcontroller based on it's ability to perform 16 bit multiplication. The PIC16F877 is under consideration. You could rule out the PIC16F877 because of which of the following (if any):
   (a) The PIC16F877 has 14 bit instructions
   (b) The PIC16F877 does not have a multiply instruction
   (c) The PIC16F877 instructions operate on 8 bit data
   (d) The PIC16F877 has Harvard architecture
   (e) none of the above

2. The 80C51 supports register-register addressing. This means that compared to the PIC16F877:
   (a) It will always run faster when moving data from one register to another.
   (b) It will take less instruction cycles to move data from one register to another.
   (c) It will take less machine cycles to move data from one register to another.
   (d) It will take less clock cycles to move data from one register to another.
   (e) All of the above statements are correct.

3. You are told that the 80C51 supports register-register instructions, while the PIC16F877 only supports load/store to accumulator. Based on this information, which of the following statements is correct:
   (a) The 80C51 does not have an accumulator.
   (b) We can move information between registers using 1 instruction on the 80C51.
   (c) We can move information between registers using 1 instruction on the PIC16F877.
   (d) We can move information between registers in 1 machine cycle on the 80C51.
   (e) We can move information between registers in 1 machine cycle on the PIC16F877.

4. You are told that the 80C51 and PIC16F877 both have data transfer instructions.The 80C51 can move a literal to any register using one instruction: `MOV Reg, #6`.

   The PIC16F877 can move a literal to the working register, and subsequently to any register using two instructions: `movlw 6` followed by `movwf Reg`.

   Based on this information ONLY, we can assume that an 80C51 program involving several literal-to-register data transfers will probably:

   4-I have less lines of code
   4-II require less program storage space
   4-III run faster

   Which combination of stated assumptions is appropriate?
   (a) none of the above assumptions are appropriate.
   (b) 4-I only
   (c) 4-I and 4-II only
   (d) 4-I and 4-III only
   (e) 4-I, 4-II and 4-III

5. Five microprocessor devices are being compared on the basis of cost, and execution speed. The device selection matrix is shown below. Which device should we choose?

Name	Scaled Cost	Scaled exec'n speed	Rating = 1*cost + 2*speed
A	1	1	3
B	2	5	12
C	0.2	1.5	3.2
D	0.1	0.6	1.3
E	5	0.1	5.2

6. From: (Wilmhurst 2001; 1.5) “Three microprocessors, A, B, C have maximum clock speeds respectively of 10MHz, 24 MHz and 20 MHz. Processor A divides its clock by 4 to give one machine cycle, processor B by 12, and processor C by 8. A and B take two machine cycles to perform an 'add accumulator to immediate data' instruction, while C takes 3 [machine] cycles. Place the processors in order of the speed with which they can perform this instruction.”

7. Due to lack of availability, we need to replace the PIC16C5x in our embedded system, with one of the alternatives investigated in AN520(Palmer 1997a). Presuming that:

   - the loop and shift benchmarks are relevant for our application, and
   - that loops currently account for 20% of our execution time and 5% of the code space
   - that shifts currently account for 60% of our execution time and 10% of the code space

   Use the reported relative code size, and relative execution times to develop a ranking scheme, and draw a ranking table for the COP800, ST62, and 8051 processors, where

   (a) code efficiency is weighted at 0.8, and speed efficiency is weighted at 0.2
   (b) code efficiency is weighted at 0.2, and speed efficiency is weighted at 0.8

   In each case, explain your chosen ranking scheme, and identify the best-ranked processor.

8. We need to rank the processors A and B to make a decision, where code size is weighted twice as heavily as cost. On Processor A, an algorithm requires 80 instructions, and on Processor B the same algorithm takes 40 instructions; however Processor A costs 20 money units, and Processor B costs 300 money units. Select the table which correctly ranks the processors.

(a)

	Cost	Relative cost	Weighted Cost	Code Size	Relative Code Size	Weighted Size	Total
A	20	1	2	80	1	1	3
B	300	15	30	40	0.5	0.5	30.5

Weighting factors: 2 (cost), 1(code size). Choice Processor B.

(b)

	Cost	Relative cost	Weighted Cost	Code Size	Relative Code Size	Weighted Size	Total
A	20	1	1	80	1	2	3
B	300	15	15	40	0.5	1	16

Weighting factors: 1 (cost), 2(code size). Choice Processor A.

(c)

	Cost	Relative cost	Weighted Cost	Code Size	Relative Code Size	Weighted Size	Total
A	20	1	2	80	1	1	3
B	300	15	30	40	0.5	0.5	30.5

Weighting factors: 2 (cost), 1(code size). Choice Processor A.

(d)

	Cost	Relative cost	Weighted Cost	Code Size	Relative Code Size	Weighted Size	Total
A	20	1	1	80	1	2	3
B	300	15	15	40	0.5	1	16

Weighting factors: 1 (cost), 2(code size). Choice Processor B.

9. Based on (Wilmhurst 2001; 1.9) "How long do the longest and the shortest instructions take to execute for each of the 16F877, ... 68HC05[, AT90S8535], when each is operating at

   - it's fastest clock frequency?
   - and external clock frequency of 2MHz?

   "

10. Challenge: Find data-sheets/specifications for a microcontroller based on a 16 or 32-bit data-word processor.

    (a) Identify features on your chosen microcontroller which are NOT available on the 8-bit data-word processors we have been studying.

    (b) Identify a task which your chosen microcontroller would be better suited for, and perform appropriate ranking to justify your choice.

    (c) Identify a real-world product in which your chosen microcontroller is used.

Tutorial Exercise 10AOffline Version[42]

ID# ____________

Your team is developing an embedded system. You have been asked to choose between the PIC16F877 and the 80C51, each clocked at 4MHz.

1. Write a program for each of the microcontrollers, which matches the following pseudo-code:

```
loop    read switch
        if switch off
            turn on light
        else
            turn off light
        goto loop
```

How many instruction cycles, program locations and data storage locations are required in each case? *8 marks*

[42] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

2. On the basis of how well the microcontroller performs light-switching (weight 0.2 program space; weight 0.2 data space; 0.6 execution speed) choose a microcontroller. *2 marks*

PLAIGIARISM DECLARATION:
For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Unit 26 Write the letter you have been assigned here________.

A instruction set size, # operands supported, architecture

B built-in peripherals

C electrical and frequency specifications

D support for debugging/development & packaging options

Compare the PIC16F877 and the 80C51 in your assigned area.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  Compare alternate microprocessors in terms of:
  - the instruction set,
  - available enhancement features (such as cache, pipelining etc.)
  - best performance, which may be facilitated by the instruction set and features.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

27 Case study

At the end of this unit the student will be able to:

- *recognize the issues and tradeoffs involved in the design of a microprocessor based (embedded) system, and*
- *justify the role of a microprocessor(-based system) in an applied context.*

"Design is the task of defining a system's functionality and converting that functionality into a physical implementation, while satisfying certain constrained design metrics, and optimizing other design metrics. Design is hard ...Not only is getting the functionality right hard, but creating a physical implementation that satisfies constraints is also very difficult, because there are so many competing, tightly constrained metrics ...."(Vahid and Givargis 2002; pp 281, Fig 11.1). " A design metric is a measurable feature of a system's implementation. Commonly used metrics include:

- NRE cost (nonrecurring engineering cost) ...
- Unit cost ...
- Size ...
- Performance ...
- Power ...
- Flexibility ...
- Time-to-prototype ...
- Time-to-market ...
- Maintainability ...
- Correctness ...
- Safety ...

Metrics typically compete with one another: Improving one often leads to worsening of another. " (Vahid and Givargis 2002; pp4-5)

The design process may be subdivided into four main stages: clarification of need, conceptual design, embodiment design and detail design (Wilmhurst 2001; Figure 12.1, Table 12.1).

The first stage begins with a "*single-phrase function statement* – a statement which summarises the main function of the product without ... saying how that function is to be achieved. Following this a *Requirements List* should be drawn up, showing all the required features and functions of the product. These can be divided into Demands (i.e. essential requirements) and Wishes (i.e. those that would enhance the product but are not essential)."(Wilmhurst 2001; p.354) To complete the first design stage, a specification is drawn up. The specification should address all of the above design metrics, as well as: functionality, user interaction, physical form, operating environment, signals, and qualification test(s) (Wilmhurst 2001; Table 12.3) (Predko 2000b). The specification may be enhanced as the design progresses.

The second stage: conceptual design involves the generation of alternate functional solutions which will meet the specification and requirements from the first design stage. One means of doing this is to identify alternate ways of achieving each requirement function, and then combine the blocks to give alternate solutions, and identify the preferred combination. (Wilmhurst 2001; p. 357)

The third stage, embodiment design is where we will make the decisions regarding hardware parts/software breakdown, and the processes which will be used to manufacture and test the prototype/product. It will include consideration of several issues including:

Do functions in hardware or software? "The choice of hardware versus software for a particular function is simply a trade-off among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference between what hardware or software can implement" (Vahid and Givargis 2002; p21)

Choice of microprocessor/microcontroller; allocation of resources The microprocessor is chosen so that it meets the conceptual design function requirements, and is ranked according to the design metrics. Once the microprocessor or microcontroller is chosen, we must decide on peripheral locations in the memory map, and pin allocations to specific functions.

Layout and communication Identification of the way in which different modules will communicate with each other, and their relative positions within the system (if applicable).

Choice of programming language; tool-chain; libraries These will be constrained by prior experience, and the hardware chosen, as well as the design metrics.

Identification of test strategies These will be used to test the final system. By considering them at this stage, we can tailor the design to facilitate testings e.g. placing LED's/other circuitry on the board to facilitate BIST.

The final stage detail design involves the production of software flowcharts/code, and hardware layouts. A bill of materials and a work-plan should also be generated. This design stage is often the most time-consuming. " In response to low production rates, the design community has focused much effort and resources to developing design technologies that improve productivity ...

1. *Automation* is the task of using a computer program to replace a manual design effort.
2. *Reuse* is the process of using predesigned components (whether designed by humans or computers) rather than designing those components oneself.
3. *Verification* is the task of ensuring the correctness and completeness of each design step

" (Vahid and Givargis 2002; pp 282, Fig 11.1). The prototype can be implemented directly from the results of this final design stage.

Do we really need a microprocessor? In closing we should warn against the typical engineer's tendency to use technology unnecessarily. In many instances a microprocessor-based system simply performs a task at similar, or not much better levels than a non-microprocessor-based solution. For example while we can design a system for maintaining water levels in a tank (μP, level/pressure sensor, relay – valve control), the task can be done just as effectively (probably cheaper and more robust) by a float attached to a mechanical lever.

Review Exercises

1. The design of an embedded system starts with the __1-I__, which is followed by the __1-II__ and then the __1-III__.

(a) 1-I: conceptual design; 1-II: function statement; 1-III: requirements list
(b) 1-I: conceptual design; 1-II: requirements list; 1-III: function statement
(c) 1-I: function statement; 1-II: conceptual design; 1-III: requirements list
(d) 1-I: function statement; 1-II: requirements list; 1-III: conceptual design
(e) 1-I: requirements list; 1-II: function statement; 1-III: conceptual design

2. From (Vahid and Givargis 2002; 1.4): "List a pair of design metrics that may compete with one another, providing an intuitive explanation of the reason behind the competition."

3. From (Wilmhurst 2001; 12.5) Write the function statement for a microprocessor-based system "that would genuinely improve the quality of life for a person or group of people. Consider devices which would give independence to the disabled or vulnerable, enhance education or save energy."

   (a) For your idea, list one alternative method of accomplishing the task without using a microprocessor-based system.
   (b) Identify one benefit gained by using a microprocessor-based system in this application.
   (c) Identify one disadvantage of using a microprocessor-based system in this application.
   (d) Write the requirements list, and specification for your proposed system.

4. Which ONE of the following statements (if any) about the microprocessor-based system design process is NOT correct?

   (a) The single phrase function statement establishes the product function.
   (b) The requirements list, contains two types of information, those requirements which are necessary, and those which are desirable.
   (c) A design metric is some property of the system implementation which can be measured, and used for comparison. with other implementations.
   (d) During the design process, decisions about hardware and software can be independently made.
   (e) All of the above statements are correct.

5. Your team is developing an embedded system which requires a digital reading of a potentiometer value. The design requires an 8 bit RISC microcontroller in a 40 pin DIP, which accepts a 4MHz clock signal. You have been asked to choose between the PIC16F877 and the ATMEL AVR AT90S8535. Apply your knowledge of CPU architecture, and use the data-sheets to determine at least **five** functional differences between the two micro-controllers.

6. Your company is designing an independent device for monitoring RS232 traffic. The device should passively monitor electrical signals on the TX/RX lines of an RS232 serial cable, and display current and cumulative estimates of communication traffic. You have been asked to choose between the PIC16F877 and the 80C51.

   (a) Use your knowledge of the PIC16F877 characteristics and the data-sheets provided to identify at least 3 functional differences which would be relevant for this application. Based on the functional differences which you have previously identified, recommend one of the microcontrollers to your team.

   (b) After selecting the processor, the team discusses the support tools which will be needed for this project. Identify at least five items which you believe are necessary in order to get this project working. Justify your answers.

7. You are a member of a development team which is developing a battery-powered refrigerator monitoring system. The system checks the actual current being drawn by the refrigerator twice an hour, and keeps an estimate of the kWh drawn to date. The system has a reset button and a display button. The system is connected to 7-segment LED display digits which show the current kWh estimate when the display button is pressed. The estimate is reset to 0 when the reset button is pressed. Your group has decided to use a 4MHz oscillator and a 40-pin DIP package.

   (a) You have been asked to choose between the PIC16F877 and the Motorola 68000. Use your knowledge of the PIC16F877 characteristics and the data-sheets provided to identify at least 3 functional differences which would be relevant for this application.

   (b) Based on the functional differences which you have previously identified, recommend one of the microcontrollers to your team, and justify your recommendation (presume that cost, availability, and prior experience are not of concern).

   (c) The reading from the current probe, needs to be scaled and added to the previous kWh estimate. The kWh estimate must then be displayed on the 7 segment display. Presuming that you are using the PIC16F877, suggest an appropriate representation for the estimate and justify your answer.

8. You are a member of a development team which is working on a new microprocessor-controlled refrigerator, using a PIC16F877 with a 700 Hz clock. The refrigerator consists of a cooling unit, a defrost unit, and a thermostat, which are connected to `PORTD` pins `<2>`, `<1>` and `<0>` respectively.

   - The cooling unit is switched on when `PORTD<2>` is high.; the defrost unit is switched on when `PORTD<1>` is low.
   - `PORTD<0>` has an external pull-up resistor. The thermostat pulls `PORTD<0>` low when the fridge temperature is above $c + 2$ degrees Celsius, and keeps the line low until the fridge temperature drops below $c - 2$ degrees Celsius; where c is the temperature setting for the fridge.
   - The defrost unit is activated for approximately 6 minutes each day.

   (a) Write the pseudo-code for your proposed system.

   (b) Suggest an appropriate test strategy for your proposed system.

9. You have been asked to design a gate opener for a new apartment complex which has 6 apartments. Each apartment owner, and the security guard, are given a unique entry code.

   (a) Write the function statement, the requirements list, and specification for your proposed system.

   (b) Write the pseudo-code for your proposed system.

   (c) Suggest an appropriate test strategy for your proposed system.

10. You wish to make a microprocessor-based coin-sorter which can accommodate multiple currencies.

    (a) List alternative method(s) of accomplishing the task without using a microprocessor-based system.

    (b) Identify benefit(s) gained by using a microprocessor-based system in this application.

    (c) Identify one disadvantage of using a microprocessor-based system in this application.

    (d) Write the function statement, the requirements list, and specification for your proposed system.

    (e) Write the pseudo-code for your proposed system.

    (f) Suggest an appropriate test strategy for your proposed system.

Tutorial Exercise 10BOffline Version[43]

ID# ____________

Your task is to design a controller, for a mobile robot which driven by two DC motors. The motors will be driven by a PWM signal. The controller will receive (EIA)RS-232 compatible serial commands at 9600 baud. The three commands are: start/stop, left speed and right speed.

1. Write the function statement, the requirements list, and specification for your system. *6 marks*

2. Draw a block diagram to illustrate how your controller will function. *4 marks*

[43]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 27 Write the letter you have been assigned here________.

A function statement

B requirements list

C specification

D conceptual design

1. Write a sentence which explains your assigned term in the context of microprocessor-based system design.

2. Produce an example of your assigned term, for a refrigerator temperature controller.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - recognize the issues and tradeoffs involved in the design of a microprocessor based (embedded) system, and
  - justify the role of a microprocessor(-based system) in an applied context.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

28 Design guidelines

At the end of this unit the student will be able to:

remember and follow a checklist of design guidelines, in order to perform a simple design for a microprocessor based system.

Processes in Embedded system Design (Wilmhurst 2001; Table 12.1)

A design model for embedded systems (Wilmhurst 2001; Figure 12.2)

Checklist for embedded system specification preparation (Wilmhurst 2001; Table 12.3)

Functionality All functions the product is meant to perform

User interaction Controls, displays

Physical form Dimensions, weight, materials, aesthetics

Safety Legal requirements, special considerations, results of product failure

Operating Environment Target environment, e.g. temperature and humidity ranges, EMC

Signals Inputs, Outputs, response times

Power Supply Source of power, power consumption

Maintenance Reliability, time between failures

Costs Development costs, product cost

Schedules Product time scales, delivery times

Hardware Design Checklist (Arnold 2001; Appendix A, Hardware Design checklist)

1. Define Power Supply Requirements
2. Verify Voltage Level Compatibility
3. Check DC Fan-out: Output Current Drive vs. Loading
4. AC (capacitive) Output Drive vs. Capacitive Load and De-rating
5. Verify Worst case timing conditions
6. Determine if transmission line termination is required
7. Clock Distribution
8. Power and Ground Distribution
9. Asynchronous inputs
10. Guarantee Power-On Reset State
11. Programmable Logic Devices
12. Deactivate Interrupt and other requests on power-up
13. Electromagnetic compatibility issues
14. Manufacturing and test issues

A Baker's Dozen Rules to Help Avoid Application Software Problems (Predko 2000d; pp 520–521,Ch. 9 PICmicro ®MCU Assembly-Language Software-Development Techniques)

1. Always initialize your variables
2. Indent conditionally executing code after a skip instruction
3. Let the compiler/assembler do the calculations for you
4. Use Microchip's register definition files without modification in all your applications
5. Keep your code as simple as possible
6. Develop your application in terms of functional blocks and interfaces
7. Establish a plan to test and confirm your code is correct
8. Use `CBLOCK` or other built-in tools to allocate variables instead of defining them manually
9. Avoid changing the register bank unless it is absolutely necessary
10. Don't allow code to go over page boundaries
11. Use the `__CONFIG` statement always in your source code and use a programmer that programs the configuration information automatically (not one that requires manual intervention)
12. Simulate as much of your application as possible
13. Keep subroutine calling to a minimum

Review Exercises

1. Write one sentence to justify each of the listed design guidelines.
2. A PWM motor controller is under development. We need to choose between a microcontroller with a PWM peripheral, and a microcontroller with a timer. Both microcontrollers are based around the same CPU. Which of the following issues(if any) are appropriate for consideration?
    (a) the PWM frequency and resolution requirements
    (b) the program and data memory requirements for bit-banging PWM using the timer
    (c) cost/availability of the microcontrollers
    (d) clock speed/packaging for the microcontrollers
3. When writing routines for a microprocessor, you have been advised to always initialize your registers, even if the default value is the value you want. Label the following statements as valid/invalid reasons for this guideline:
    (a) The microprocessor may malfunction and/or reset without powering up
    (b) The microprocessor may not power-up correctly
    (c) The routine may attempt to access more memory than originally intended/allocated
    (d) The routine may be called after the value was changed subsequent to power-up
    (e) The routine may be interrupted while it is running.
4. When designing a microprocessor based system for the PIC16F877, you have been advised to simulate as much of your application as possible. Which of the following statements is an INVALID reason for this guideline?
    (a) Flash EEPROM has a finite number of write cycles; simulation avoids using them up needlessly.
    (b) Simulation is faster for debugging, as it avoids download time.
    (c) Software testing can begin before hardware is available.
    (d) We can avoid inadvertently starting up harmful equipment.
    (e) We can use the simulator to troubleshoot noise problems.
5. Based on your experiences in the lab, review the lists of design guidelines, and add/remove guidelines, to form your own compiled list.
6. Challenge: Investigate the operation of the IPod Music Player. Can you design a similar product?
7. Challenge/Role Play: Find/read stories about the Therac-25 accidents (try: `http://sunnyday.mit.edu/therac-25.html`).
    - Identify the design/testing guidelines that could have averted these problems.
    - Identify the key persons whose decisions may have contributed to these problems. What would they have said in their defence?

8. You have been asked to design a microprocessor controlled cooling system for equipment mounted in an industry standard 19 inch rack.

   - The rack is fitted with a power supply which delivers 30V for fans and equipment.
   - The rack is fitted with DC cooling fans which require 30V.
   - At 20 degrees Celsius the fans should be off, and at 50 degrees Celsius they should be running at full speed.
   - There is a thermocouple mounted in the equipment rack which delivers a signal between 0 and 2V for temperatures between 20 and 60 degrees Celsius. The thermocouple impedance is 200 ohms.
   - When the rack temperature rises above 60 degrees Celsius, it automatically shuts off all power. The power supply must subsequently be manually reset.
   - The microprocessor should report the current rack temperature on an Hitachi 44780 LCD display mounted at the front.
   - The microprocessor should asynchronously report (via RS232 cable) the present temperature to a remote PC at 9600 baud, no parity, no stop bit.

   Your boss wants you to use the PIC16F877 with an 8MHz crystal oscillator because he happened to have them left over from a previous project. [44]

   (a) What voltage will you operate your system at and how will you deliver power to the microprocessor/oscillator? (Keep in mind that the fans or other equipment need to be isolated for noise)

   (b) The thermocouple voltage will be sampled using the A/D converter. Determine an appropriate setting for the A/D conversion clock when the chip oscillation frequency is 8MHz, and the minimum acquisition time for this thermocouple. What is the overall sampling time?

   (c) Temperature change in a rack is generally a slow process, so it does not matter in this case; but what is the maximum frequency we should attempt to sample with the above sampling time?

   (d) Choose reference voltages for the A/D conversion. What range of digital values will be obtained from the thermocouple?

   (e) The fan motors will be driven using the built-in PWM module and a transistor. The duty cycle will be determined from the range of digital values from the thermocouple. Choose an appropriate PWM frequency, and a method of calculating duty cycle such that each 10-bit A/D value obtained for the thermocouple generates a different duty cycle.

   (f) The LCD display may be interfaced to the PIC by using the I/O port pins to make a parallel bus. Read the timing diagram for the LCD panel, and design subroutines which will communicate with the LCD. (Note: you do not need to write any code!!)

   (g) With no parity, the remote PC has no means of verifying if individual data bytes have been corrupted. Suggest a possible scheme for checking whether data has been corrupted en-route.

[44] Yes I do realize this is a very far-fetched/unrealistic scenario but it serves our purposes.

(h) At 9600 baud, is it possible to send every reading to the remote PC? If not, suggest one method of reducing the data to be sent.

(i) There are many available software communication protocols for sending data from the PIC to the remote PC. Suggest an alternative which might be better suited to an industrial environment than RS232, and justify your choice.

(j) Your boss runs into your office while you are still designing, and says he wants to have one microcontroller system do temperature control for 8 racks simultaneously i.e. 8 PWM fan outputs; 8 LCD displays; 8 thermo-couples. Suggest one way in which you could do it.

(k) Draw flow-chart; block diagrams; and write algorithms to illustrate your final design.

(l) Run through the checklists; which issues still need to be addressed?

(m) Draw up a testing procedure for the final system.

9. You have been asked to design a gate opener for a new apartment complex which has 6 apartments. Each apartment owner, and the security guard, are given a unique 8-bit entry code. Each valid entry attempt is logged by incrementing the 16-bit count associated with that entry code. There is a separate count for invalid entry attempts.

   (a) 16-bit counts are stored in memory at consecutive locations, hi-byte first, starting at location `CtrArray`. Write a routine called `UpdArray` which uses indirect addressing to increment the counter specified by the value in the working register.

   (b) Write a routine which will compare an 8 bit code stored in `CodeRead` to the entries in a "regular" lookup table `CodeTbl` and return with either

      - the index of the code in the table, or
      - the value 7 if the code is not found in the table

      in the working register.

   (c) Challenge: modify your routine so that the data values are stored in the data EEPROM.

   (d) Write a routine which reads 4 switches connected to `PORTD<3:0>` and displays the appropriate histogram bin value on the LED bar connected to `PORTC<7:0>`.

   (e) Challenge: modify your routine so that it displays a level and not just the binary number.

   (f) Identify an appropriate means for the user to enter the code, and justify your choice. Design circuitry, and code such that the entry code is stored in `CodeRead` when the user attempts to enter the complex.

   (g) Challenge: modify your routine to use interrupts.

   (h) Write a main loop, add all your routines, compile and verify that your system works in simulation.

   (i) Criticise the system you have just produced, suggest at least two improvements.

   (j) Outline how you would verify that the **final** system was functioning **as required**.

10. You have been asked to produce a 5 words-per-minute (= 250 counts-per-minute) Morse Code Generator, using a PIC16F877 with an 8MHz clock, which will accept a string of characters from a keyboard, and produce the corresponding morse code tones using a speaker which is switched on/off by circuitry connected to PORTB pin 5.

    (a) Write a timing routine which will delay for 1 Morse code count.

    (b) Using your timing routine, write routines which will:

        i. play a short tone

        ii. play a long tone

    (c) Using your wait routine, write routines which will wait for 3 and 7 counts respectively (Suggestion: Keep a counter and check when the appropriate bit goes high).

    (d) Decide on an appropriate means of representing morse code in a register in the PIC16F877. (Hint: Is it sufficient to use 0's for dots, and 1's for dashes? How will we know how many codes are used to represent a character?)

    (e) Using your representation for morse codes, write a routine and a lookup table, such that the routine will translate the ASCII value in the working register into the the appropriate code, and return the code in the working register.

    (f) Write a routine which will "play" the morse character tones using your tone routines, based on the representation returned from the lookup table routine.

    (g) Write a routine which will "play" an indirectly addressed null-terminated string. (the address of the start of the string is either passed in the working register, or set up prior to calling the routine).

    (h) Assemble and test all your routines. You should specify how you tested your routines, and what modifications you subsequently made to the routines/representation (if any).

Morse code/ASCII Summary:
(dot, di) short tone: 1 count
(dash, da)long tone: 3 count
space between tones in a character: 1 count
space between characters in a word: 3 counts
space between words in a message (ASCII code 0x20): 7 counts

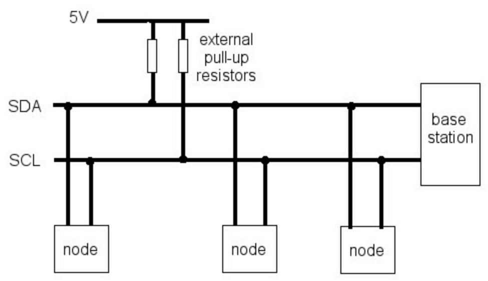
Char	ASCII	Morse	Char	ASCII	Morse	Char	ASCII	Morse
A	0x41 (0x61)	. -	B	0x42 (0x62)	- . . .	C	0x43 (0x63)	- . - .
D	0x44 (0x64)	- . .	E	0x45 (0x65)	.	F	0x46 (0x66)	. . - .
G	0x47 (0x67)	- -.	H	0x48 (0x68)	. . . .	I	0x49 (0x69)	. .
J	0x4A (0x6A)	.- - -	K	0x4B (0x6D)	- . -	L	0x4C (0x6C)	. - . .
M	0x4D (0x6D)	- -	N	0x4E (0x6E)	- .	O	0x4F (0x6F)	- - -
P	0x50 (0x70)	. - - .	Q	0x51 (0x71)	- - . -	R	0x52 (0x72)	. - .
S	0x53 (0x73)	. . .	T	0x54 (0x74)	-	U	0x55 (0x75)	. . -
V	0x56 (0x76)	. . . -	W	0x57 (0x77)	. - -	X	0x58 (0x78)	- . . -
Y	0x59 (0x79)	- . - -	Z	0x5A (0x7A)	- - . .			
0	0x30	- - - - -	1	0x31	. - - - -	2	0x32	. . - - -
3	0x33	. . . - -	4	0x34	. . . . -	5	0x35	. . . . .
6	0x36	-	7	0x37	- - . . .	8	0x38	- - - . .
9	0x39	- - - -.						
. Fullstop	0x2E	. - . - . -	, Comma	0x2C	- - . . - -	? Query	0x3F	. . - - . .

Assignment E

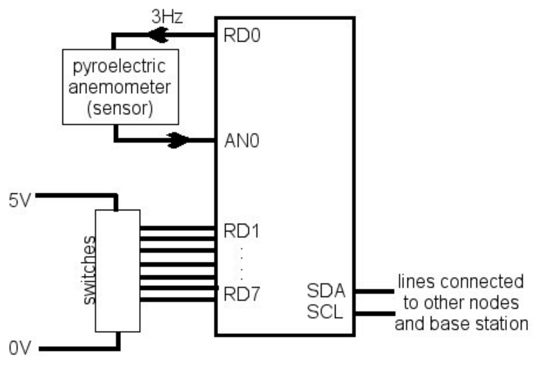
ID# ______________

You are a member of a team of engineers working on a wired sensor array to monitor the fluid flow distribution in a natural gas pipeline.

- The sensor array consists of several nodes which can each communicate with a base station.
- Each node consists of:
  - a PIC16F877, using a 4MHz external clock signal, powered at 5V.
  - a pyroelectric anemometer, to sense flow rate.
  - 7 switches which specify the node identifier.
- A pyroelectric anemometer is a sensor which detects fluid flow rate at a location in the pipeline. It requires an AC input signal. It generates an AC output signal of the same frequency as the input signal, but whose amplitude is proportional to the rate of fluid flow at the sensor.
- The input of the pyroelectric anemometer is connected to digital output RD0 of the PIC16F877. A software generated 3Hz; 0 to 5Volt; square wave is output from the PIC16F877 on digital output RD0. The 3Hz signal is generated using Timer1, using a 1:8 pre-scaler, and initialised with an appropriate value p.
- The output of the pyroelectric anemometer is an AC signal varying between 0 and 5V. It is connected to analog input 0 (AN0) of the PIC16F877. The resulting 10-bit readings are stored for later reporting to the base-station.
- For communication, the base station acts as the master and polls each (slave) node periodically. Upon receiving a request from the (master) base station, the (slave) node responds with the most recent 10-bit output of the amplifier software.
- Nodes & base station are attached to an I2C bus, consisting of two lines (SCL,SDA) with external pull-ups.
- Each node should respond to a unique 7-bit I2C address. This address is configured at node startup. The node I2C address should be set by the 7 switches connected to RD1-RD7.



System Overview


3Hz
RD0
pyroelectric anemometer (sensor)
AN0
5V
switches
RD1
RD7
SDA
SCL
lines connected to other nodes and base station
0V

Node

Pseudocode

```
main:        call Cfg
             enable global interrupts
             goto loop

loop:        if Timer1 has overflowed
                      toggle RD0 pin state
                      place initial value $p$ in Timer1
             goto loop

Cfg:         configure pins, Timer1, A-D & I2C
             enable A-D & I2C interrupts
             initialize all variables
             place initial value $p$ in Timer1
             start A-D and Timer1

MainISR:     store context
             if I2C interrupt; do I2C Handler
             if A-D interrupt; do A-D Handler
             restore context

I2C Handler: if request for this node
                      report upper 8 bits of Amplifier output
             clear I2C flag

A-D Handler: retrieve 10 bit A-D output
             reading = (upper 4 A-D bits) - 8
             call Amplifier
             clear A-D flag
             start next A-D sample
```

---

1. Write an assembly language routine named `average` for the PIC16F877 which will return the average value of two 16-bit signed two's complement representation numbers (contained in registers `Ahi:Alo` and `Bhi:Blo`) in the registers `Chi:Clo`. *12 marks*
2. Write an assembly language routine named `multiply` for the PIC16F877 which will return the product of two 8-bit signed two's complement representation numbers (contained in registers `M1`, and `M2`) in the registers `Qhi:Qlo`. Please state the mathematical principle on which your routine is based. *12 marks*
3. Use the psuedo-code provided to write a main loop, a main interrupt-subroutine, add all your routines, compile and verify that your system works in simulation. Explain how you tested your code. *8 marks*
4. Criticise the system as specified, and suggest at least two improvements. *4 marks*
5. Outline how you would verify that the **final** system was functioning **as required**. *4 marks*

Submission Guidelines

You will be penalised (up to a maximum of 4) marks if you violate ANY of the program formatting guidelines. All submitted programs MUST follow the specified format i.e.

- labels in first column;
- code in second column;
- operands (where appropriate) in 3rd column;
- comments (where appropriate) in the 4th column
- directives in capital letters;
- assembly language in common letters;
- variable names/labels lower case (optional first capital letter).

Your files should be named Eqnnnnnnnn.* where nnnnnnnn is your ID Number and q is the question number.

Please type your answers into plain ASCII text files (no Word documents please!) with the same name(s) as your other files, and the extension .asc . Alternatively you may write your answers in the *.asm files as comments.

The *.asc, *.hex, *.asm and *.lst files for all questions should be zipped into a single archive named Ennnnnnnn.zip where nnnnnnnn is your ID Number.

The archive should be uploaded to myELearning. Please remember to electronically submit AHEAD of the deadline in order to avoid problems with system overload.

This assignment is worth 2% of your ECNG2006 mark. It contains a total of 40 marks.

Unit 28 Write the letter you have been assigned here________.

The guideline is:

What is the worst thing that could happen if we ignore this guideline:

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - remember and follow a checklist of design guidelines, in order to perform a simple design for a microprocessor based system.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

System Issues

In this section we have looked at issues which affect overall system performance. These include the effects of integrating hardware and software modules (which may have been developed and tested independently), a comparison of components for selection and a methodology for system design. In doing so we have fulfilled the fourth learning objective "select (and critique the selection of) a microprocessor-based system for an application, given relevant datasheets and application requirements."

Part VIII
Communication

In Part VI, we examined ways in which peripherals could be interfaced with the microprocessor. In this section, we will elaborate on peripherals whose function is to facilitate communication between the microprocessor and other devices; firstly we will look at standard communication protocols/interfaces which such peripherals may support.

29 Communication protocols

At the end of this unit the student will be able to:

- *understand commonly used parallel and serial communication protocols (I2C, CAN, RS232, USB, Firewire)*
- *interpret descriptions of proprietary protocols (Dallas 1-wire)*

In the notes so far, we have used the term *bus* to describe the physical means by which:

- the CU communicates with other parts of the CPU,
- the CPU communicates with other parts of the microcontroller,
- the microprocessor/microcontroller communicates with other parts of the microprocessor-based system.

In effect a bus is any transport mechanism which moves information between devices. The term can also be used to describe the physical means by which two independent microprocessor-based systems communicate with each other.

Communication between two microprocessor-based systems can be achieved in many different ways. We have already noted that internal bus transactions may be either synchronous or asynchronous. In order to demarcate the bounds of an asynchronously transmitted message specific message formats are used: e.g. start/stop bits, frames with Header and EOF.

Another distinction which we can make is between *bit-serial* (only one bit is transmitted at a time) and *bit-parallel* (more than 1 bit transmitted at a time, may also be called byte-serial) transfers. We can also differentiate between *simplex* (where a signal is only sent in one direction on a particular signal line) *half-duplex* (where a signal is sent in both directions (at different times) along the same signal line) and *full-duplex* (where signals are sent in both directions along different lines). Signals sent along the bus may be referenced to the system ground (*unbalanced* or *single-ended*), a single reference line (*signal ground*), or individual return lines (*differential* or *balanced*).

We can look at whether the bus connects only 2 devices together (*point-to-point*) or if several devices have independent physical access to the bus(*network*). "Virtual" buses (as opposed to physical buses) can be created by relaying information from one physical bus to another. Where several devices are communicating on the bus, the message must contain information which indicates who the sender/recipient of the information is (internal to the microprocessor-based system – this is addressing).

Information may be transmitted across any type of bus using different *line coding* techniques. So far, we have been using two voltage levels to represent the state of the bit, and the same voltage levels always represent the same state. We may also use the transition (from hi to lo or lo to hi), the presence or absence of a pulse, and the pulse width to represent the state of a bit. Furthermore, consecutive bits of the same value need not necessarily be represented by the same signal, and multi-bit patterns may be represented using 3 or more voltage levels.

The use of such line coding techniques improves the robustness of the bus communication. Where additional surety is required, some form of error detection/correction (e.g. parity, checksums, message length) may be employed. This requires additional information to be passed across the bus. The receiver can check that the additional information is consistent with the message, and thereby determine that the information is the same as that which left the source. The rate at which information is passed across a bus will depend not only on the signal transaction speed of the bus, but also on the line coding technique employed, and the amount of additional information that is transmitted (i.e. destination, source, error check).

Communication protocols define the means in which communication will take place, and generally define:

- the number of signal wires, and the voltage tolerances
- transaction timing
- message format and the form of error detection/correction to be used (if any).
- the form of synchronization to be used (if any)
- standard interface connections (if any)

The standard methods of communication which we will examine in this unit, all utilise different ways of dealing with these issues and are summarised in the following tables.

I2C(I2C 2000)

Operating mode Synchronous, bit-serial, (multi-)mastered

Signal Level unbalanced signal; bus lines floats high by default; low voltage 0V, high voltage $\approx V_{dd}$ (different voltage devices can communicate)

Line coding NRZ level code (hi - 1; lo - 0)

Speed 100kHz (400kHz – fast mode)

Max. # devices "The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF."(I2C 2000)

Transmission distance This is limited by the maximum bus capacitance. This can be overcome by reducing the frequency of transactions, using I2C drivers, or adding pull-up resistors. "wiring must be so chosen that crosstalk and interference to/from the bus lines is minimized. The bus lines are most susceptible to crosstalk and interference at the HIGH level because of the relatively high impedance of the pull-up devices.

If the length of the bus lines on a PCB or ribbon cable exceeds 10 cm and includes the VDD and VSS lines, the wiring pattern must be: SDA VDD VSS SCL

If only the VSS line is included, the wiring pattern must be: SDA VSS SCL

These wiring patterns also result in identical capacitive loads for the SDA and SCL lines. The VSS and VDD lines can be omitted if a PCB with a VSS and/or VDD layer is used.

If the bus lines are twisted-pairs, each bus line must be twisted with a VSS return. Alternatively, the SCL line can be twisted with a VSS return, and the SDA line twisted with a VDD return. In the latter case, capacitors must be used to decouple the VDD line to the VSS line at both ends of the twisted pairs.

If the bus lines are shielded (shield connected to VSS), interference will be minimized. However, the shielded cable must have low capacitive coupling between the SDA and SCL lines to minimize crosstalk." (I2C 2000)

# Communication Lines 2

# Signal Lines 1 (SDA)

# Handshaking / Clock lines 1 (SCL)

Arbitration Distributed – monitor line – 0 bit has priority – drop off and wait for stop.

Message/Transaction Format MSB first;

- (Master) Start Condition,
- (Master) Receiver Address (7 bits) ,
- (Master) Read/Write Request (1 bit),
- (Device) Acknowledge (1 bit),
- (Master - Write; Device - Read) Data Byte (8 bits),
- (Device) Acknowledge
- (Master) Optional: additional Start (read/write), or Data Byte (write only)
- (Master) Stop Condition

Standard Connector(s)/Cable(s) – none –

Controller Area Network (CAN) ISO 11898

Operating mode asynchronous

Signal Level 0 - 16 V; “The bus can have one of two complementary logical values: ‘dominant’ or ’recesive’. During simultaneous transmission of ‘dominant’ and ‘recessive’ bits, the resulting bus value will be ’dominant’.”(CAN 1991)

Line coding NRZ

Speed 200 kbps (nominal) – 1Mbps

Max. # devices “The CAN serial communication link is a bus to which a number of units may be connected. This number has no theoretical limit. Practically, the total number of units will be limited by delay times and/or electrical loads on the bus line.”(CAN 1991)

Transmission distance dependent on speed 40m @1Mbps. Point-to point cabling.

# Communication lines 2 lines

# Signal lines 1 differential pair

# Handshaking/Clock lines – none –

Arbitration Distributed – monitor line – 0 bit has priority – drop off and wait for stop.

Message/Transaction Format “

SOF	Start of Frame, A single Dominant Bit
Identifier	11 or 19 Bit Message Identifier
RTR	This Bit is set if the transmitter is also TXing Data
r1/r0	Reserved Bits, Should always be Dominant
DLC	Four Bits indicating the number of bytes that follow
Data	Zero to 8 Bytes of Data, Sent MSB First
CRC	15 bits pf CRC data followed by a recessive bit
Ack	Two Bit field, Dominant/ Recessive Bits
EOF	End of Frame, at least 7 Recessive Bits

”

(Predko 2000c)

Standard Connector(s)/Cable(s) 9 pin D-type connector, STP, UTP or ribbon.

GPIB/HPIB (IEEE488,IEC60625)

Operating mode Asynchronous (handshaking)

Signal Level TTL level; active low

Line coding NRZ

Speed limited to speed of slowest device on bus (daisy-chain); 1Mbps

Max. # devices 20

Transmission distance 20m total – maximum separation 2m (point-to-point)

# Communication lines 24 lines

# Signal lines 8 data lines + 5 interface lines

# Handshaking/Clock lines 3 handshaking lines

Arbitration Central – all devices must request permission from the controller

Message/Transaction Format "When the Controller or a Talker wishes to transmit data on the bus, it sets the DAV line high (data not valid), and checks to see that the NRFD and NDAC lines are both low, and then it puts the data on the data lines. When all the devices that can receive the data are ready, each releases its NRFD (not ready for data) line. When the last receiver releases NRFD, and it goes high, the Controller or Talker takes DAV low indicating that valid data is now on the bus.

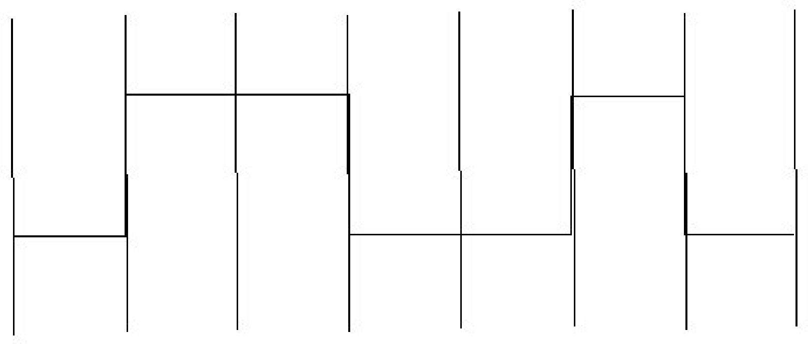
In response each receiver takes NRFD low again to indicate it is busy and releases NDAC (not data accepted) when it has received the data. When the last receiver has accepted the data, NDAC will go high and the Controller or Talker can set DAV high again to transmit the next byte of data. "

– from HTBasic GPIB Tutorial, http://www.techsoft.de/htbasic/tutgpib.htm

Standard Connector(s)/Cable(s) Centronics-type 24 pin connectors, shielded 24 conductor cable.

Review Exercises

1. The signal shown in the figure appears in a communication scheme that uses NRZ coding, a logic '1' appears as a negative voltage, and a logic '0' appears as a positive voltage. What 7-bit pattern is being transmitted?



   (a) 0110010
   (b) 0110011
   (c) 1001100
   (d) 1001101
   (e) none of the above

2. Which of the following statements does NOT describe the RS232 protocol:

   (a) Signal voltages: ±12Volts
   (b) Hardware, software or no handshaking
   (c) Error correction through the use of parity bits
   (d) Supported speeds: 1200/second, 9600 bits/second
   (e) Start and Stop bits with each byte; LSB first transmission

3. Serial communication is being carried out across a channel which can support a maximum of n transitions per second. The line coding scheme used has not more than 2 transitions per bit, and 7 bit data-words are transmitted with an additional 3 bits of overhead. The effective bit-rate at which data can be transferred across this channel is:

   (a) n bits/second
   (b) $n \times \frac{7}{(3+7)}$ bits/second
   (c) $\frac{n}{2} \times \frac{7}{(3+7)}$ bits/second
   (d) $\frac{n}{2} \times \frac{3}{7}$ bits/second
   (e) $\frac{n}{2} \times \frac{3}{(3+7)}$ bits/second

4. The PC has standard ports which can be used for interfacing different devices. What is the advantage of having standard (as opposed to proprietary ports/interface cards) for external devices?

5. Identify one serial/parallel interface standard for PC's which was not identified in the unit.

6. Differentiate between simplex and half-duplex communication.

7. What would happen if two senders simultaneously tried to transmit over a half-duplex line?

8. NRZ line coding involves maintaining an active high or low for the entire bit-period. Manchester line coding involves a signal transition during each bit period (hi− >lo) for 1 and (lo− >hi) for 0. In the Dallas 1 wire protocol, a 1 is encoded by pulling the line low for a short time, and a 0, by pulling the line low for a longer time. Draw the line signals for the nibble 0101 transmitted using NRZ, Manchester and Dallas 1-wire encoding schemes respectively.

9. A particular serial interface has the following message format: 2 bits header, 8 bits information (LSB first), 1 bit checksum, 1 bit end-of frame. The channel supports n transitions/second. What is the true rate at which information is transferred across the channel if we use each of the following line coding schemes?
   (a) NRZ coding
   (b) Manchester encoding

10. Challenge: Research and draw up similar tables for one of the following communication standards: RS232, USB, Firewire, IEEE1294, ModBUS RTU, DNP3, Ethernet, AS-I (Siemens)

Tutorial Exercise 11AOffline Version[45]

ID#

You have been asked to design an RFID (Radio Frequency ID) stock tracking system which can be integrated into the shelves of a retail store. You are given the PIC16F877 with a 16 MHz clock and the Texas Instruments Series 2000 Micro-reader module (RI-STU-MRD1). (see `http://www.ti.com/tiris/docs/manuals/refmanuals/micro_8.pdf`) Describe the communication protocol for communication between the module and the PIC16F877 in the space below. *12 marks*

Operating mode .

Signal Level .

Line coding .

Speed .

Max. # devices .

Transmission distance .

# Communication lines .

# Signal lines .

# Handshaking/Clock lines .

Arbitration .

Message/Transaction Format .

Standard Connector(s) .

[45] Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed: Date:

(Department of Electrical and Computer Engineering)

Unit 29

Dallas 1-wire (Micro-LAN)

Operating mode .

Signal Level .

Line coding .

Speed .

Max. # devices .

Transmission distance .

# Communication lines .

# Signal lines .

# Handshaking/Clock lines .

Arbitration .

Message/Transaction Format .

Standard Connector(s) .

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - understand commonly used parallel and serial communication protocols (I2C, CAN, RS232, USB, Firewire)
  - interpret descriptions of proprietary protocols (Dallas 1-wire)
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

30 Bit-banging vs. hardware

At the end of this unit the student will be able to:

> *differentiate between bit-banging and dedicated hardware interfaces for serial/parallel communication (PPI/PIA vs ACIA/UART, PPC)*

In the previous unit we talked about serial/parallel communication protocols. Software implementations of standard communication protocols are often described as bit-banging i.e. they bang out bits repeatedly.

> "The bit banging method is a way that does not require assistance from any external circuitry to send or receive data stream. At the sending or receiving device, normally there is a controller handles everything during the communication period. The task of sending/checking on the clock and sending data/waiting for response from the other end on a bit-by-bit basis requires constant attention during the communication period that leaves heavy burden on the controller on both parties. There is little room and time for the controller to perform tasks during the sending/receiving data period. It has little error detecting that the controller can perform since it is constantly busy in dealing with data transmission on every single bit."

– from http://www.21centuryengineer.com/issues/fall2002/siung.htm
Previous units have shown how dedicated peripherals can be used to off-load work from the microprocessor. This can also be done for communications. However because of the continuous and potentially complex nature of communication, the peripherals must be particularly "smart". Features which appear in communication peripherals include:

- peripheral buffering (TX and/or RX) w/programmable interrupt levels, and transmission speeds
- checksum/parity generation/checking
- transmission/recieve error detection/notification
- automatic hardware/software handshaking
- automatic message re-transmission/re-request
- automatic reset of interrupt flags
- automatic RX filtering: e.g. address/message type recognition

Ideally, the presence of any subset of these features on a peripheral will make it a more efficient alternative to bit-banging software. In practice, peripherals tend to be customised to accommodate a certain protocol(s), and bit-banging software offers more flexibility and control over the communication process.

Bit-serial communications, in particular, are very dependent on timing; in order to pass large quantities of data, individual bits must be switched/detected on the signal lines at fast rates. This means that either the microprocessor (bit-banging) or the peripheral must sample the line at a higher speed than the signal transition rate. The sample buffer can then be examined for “glitches” and “bounce”, and these effects ignored. The “true” bit can be used to synchronize the receive and transmitter clocks. Certain line/message coding techniques (e.g. start/stop bits, Manchester encoding, bit stuffing) facilitate easier synchronization. Because of these characteristics the receive is very sensitive to anything which affects rise/fall time on the signal line, and mismatched tx/rx speeds while detecting data, will not be reconstructed properly.

One particular limitation of peripherals is their ability to accommodate only a finite # of sampling speeds. The sampling clock is generally derived from the microprocessor clock, and therefore must have a sampling period which is an integer multiple of the clock period. Where the desired sampling period is not an exact multiple, synchronisation must be used to correct for any error.

Review Exercises

1. Which of the following (if any) are advantages of using bit-banging software to perform communication, rather than a dedicated peripheral device?
   (a) ability to perform communication in parallel with other tasks
   (b) ability to perform error detection
   (c) ability to service communications on demand
   (d) ability to support/adapt to multiple protocols
2. You are given the choice of using bit-banging or a dedicated peripheral to communicate with the device. Label the following statements True/False:
   (a) If we use a peripheral, then our program will need to provide the peripheral with commands about the required function.
   (b) If we use bit-banging, then our program will need to explicitly set and clear bits at the correct times.
   (c) Monitoring communications for bit-banging may occupy most of the processor time.
   (d) Peripherals are always faster than bit-banging.
   (e) We may use built-in timers and timer interrupts to perform bit-banging.
3. A key difference between the use of bit banging and dedicated communication peripherals is:
   (a) the peripheral can perform buffering, and communication handshaking independently of the processor.
   (b) the processor cannot communicate as fast as a dedicated peripheral can.
   (c) the processor cannot use interrupts to do bit-banging.
   (d) the processor has less work to do with bit-banging than with a dedicated communication peripheral.
4. "Why are transmit and receive clocks usually a factor of 16 or 64 times the data rate?" (Cady 1997; 10.3)
5. Draw a diagram to illustrate the problems which may arise when the clock is not properly recovered from an asynchronously transmitted signal.

Tutorial Exercise 11BOffline Version[46]

ID#

You have been asked to design an RFID (Radio Frequency ID) stock tracking system which can be integrated into the shelves of a retail store. You are given the PIC16F877 with a 16 MHz clock and the Texas Instruments Series 2000 Micro-reader module (RI-STU-MRD1). (see `http://www.ti.com/tiris/docs/manuals/refmanuals/micro_8.pdf`)

In order to communicate with this module, you decide to use the built-in serial communications peripheral. Write appropriate initialisation and termination routines for the peripheral, presuming that you will be using TX and RX interrupts. You should note any requirements/presumptions you make concerning the PIC16F877 support circuitry. *16 marks*

[46]Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 30 Write the letter you have been assigned here________.

	Serial	Parallel
Bit-banging	A	B
Peripheral	C	D

1. Describe how your assigned combination operates in the performing communication tasks.

2. Identify one advantage of your combination over bit-banging/peripheral use for communication.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
  - differentiate between bit-banging and dedicated hardware interfaces for serial/parallel communication (PPI/PIA vs ACIA/UART, PPC)
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

31 I/O on the PC

At the end of this unit the student will be able to:

perform I/O programming on the PC in C/Assembly using a C/C++ compiler

The modern-day Intel Pentium(or compatible) PC is based on the original 8086/88. In order to appreciate how instructions work in the 8086-type architecture, we need to understand a bit about the architecture. "There are four general purpose registers: AX, BX, CX, and DX, each of which can be used to manipulate a whole 16-bit word .... Each of these registers can be used as two 8-bit registers, for example, AL represents an 8-bit register that is the lower half of AX, and AH ... the upper half of AX. Each of the ... registers has one or more implied functions ... AX is the accumulator, and is used for all input/output operations ... DX is the data register, and is used for some input/output ..." (Buchanan and Wilson 2001; p.97)

"The move [assembly language] instruction (`mov`) moves either a byte (8 bits) or a word (16 bits) from one place to another. There are three possible methods:

- moving data from a register to a memory location. [`mov [ax],bx`]
- moving data from a memory location to a register. [`mov ax,[bx]`]
- moving data from one register to another. [`mov ax,bx`]

....

```
mov cx,20           ; moves decimal 20 into cx
mov ax,10h          ; moves 10 hex into ax
mov ax,01110110b    ; moves binary 01110110 into ax
mov bx,200h
mov [bx],50h        ;load 50 hex into memory location 200h
```

" (Buchanan and Wilson 2001; p.105)

I/O devices on the PC are mapped into a separate 16-bit address space. "I/O address are called ports. ... The I/O address space is from 0000h to FFFFh ... split into two main areas: 00h – FFh: embedded I/O address ... 100h–FFFFh: expansion I/O addresses ... Some I/O addresses are index or data register addresses. The data written to an index register indicates to the device which internal register is to be accessed. The data written to or read from a data register is the actual data value to be placed in the register or the data contained in the register, indicated by the preceding write to the index register. "(Buchanan and Wilson 2001; p. 181)

The `in` and `out` assembly language instructions are used to access ports. The address to be accessed can be specified in the instruction, or may be loaded into the DX register, as shown below

```
in  al,03f8h
in  al,dx
out 03f8h, al
out dx,al
```

"Most modern C++ development systems use an inline assembler that allows assembly language code to be embedded with C++ code. This code can use any C variable or function name that is in scope. The `__asm` keyword invokes the inline assembler, and can appear wherever a C statement is legal. "

```
void main()
{
    unsigned int i=5;

    __asm mov ax,[i];
    __asm
    {
        inc ax
        i=ax;
    }
}
```

In C/C++ development tool-chains, frequently used assembly language instructions are often wrapped into C-style functions provided in header/library files. For example the `_inp` and `_outp` C-functions may be used instead of assembly language calls to `in` and `out`.

A couple of provisos to the preceding notes:

- When programming to access a device in the I/O space, it is necessary to ascertain whether the device actually exists at the specified location. The result of accessing an unspecified device may vary from machine to machine, and compiler to compiler.
- This technique should not be viewed as an alternative to having a separately assembled assembly language module which is linked with the compiled code. It is best used when the assembly language functionality required is limited to simple operations.
- Variations exist between compilers: the actual C functions may be named differently e.g. `_asm` or `inportb`; and pre-processors may not correctly interpret assembly language syntax.

Review Exercises

1. Which of the following is NOT a valid comparison between the PIC16F877 microcontroller and Intel x86 processors?
   (a) Intel x86: CISC PIC16F877: RISC
   (b) Intel x86: Harvard PIC16F877: Von Neumann
   (c) Intel x86: Isolated I/O PIC16F877: Memory Mapped I/O
   (d) Intel x86: Vectored Interrupts PIC16F877: Single Interrupt
   (e) Intel x86: Identical data/address widths PIC16F877: Different data/address widths
2. If we need to communicate between a PC and this device, we could interface individual lines to the Parallel Port. To access the various lines of the Parallel Port, we use the `inp` and `outp` instructions. These instructions refer to AL (Register A). However, when moving information between register A and other registers we refer to AX. This is because of the following reasons EXCEPT:
   (a) AX is a 16 bit register, and AL is the lower byte.
   (b) The parallel port peripheral registers are 8 bits wide.

(c) The `outp` instruction format only has room for 8 bits of data.

(d) Of the bus lines used to communicate with the parallel port peripheral, only 8 are data lines.

(e) The PC processor is backwards compatible with earlier members of the x86 family.

3. An assembly language program which accesses peripheral registers on the PC, uses the `inp` and `outp` assembly language instructions to access them. Data registers on the PC are accessed using the `mov` instructions. The PIC16F877 uses the same `mov` instructions to access both data and peripheral registers. This is because:

   (a) The PIC16F877 is a RISC processor and the PC is a CISC processor.

   (b) The PIC16F877 has Harvard architecture and the PC has Von Neumann architecture.

   (c) The PIC16F877 has a memory-mapped I/O system and the PC has an isolated I/O system.

   (d) all of the above

   (e) none of the above

4. PC's maintain a system variable containing the number of seconds which have elapsed since the PC was switched on. Outline how this could be implemented using the clock signal and a digital counter. Is the counter really necessary?

5. Choose a PC C/C++ compiler, and determine

   (a) the name/format of the keyword for including assembly language statements within the C program

   (b) the name of the wrapper functions for the I/O instructions `in` and `out`

6. Choose one of the following devices (Serial Port COM3, Programmable Interrupt Controller, Programmable System Timer, Lan-on-Motherboard Controller), and find out

   (a) where it is located in the PC I/O memory map (i.e. base address)

   (b) the number of registers associated with the device,their functions, and their respective offset addresses.

32 PC to μP serial link

At the end of this unit the student will be able to:

> *produce assembly language programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and PPC datasheets/details, and respective instruction sets.*

The PIC16F877 does not have dedicated peripherals which perform parallel communication. The 8259 parallel port controller performs this function on the PC, and was originally intended for communication with a printer. We used this peripheral in the previous unit's lab exercise.

The PIC16F877 includes peripherals which will independently perform serial communications: The Master Synchronous Serial Port (MSSP) for specific synchronous communication protocols (i.e. I2C and SPI), and the Universal Synchronous Asynchronous Receiver Transmitter (USART) for general synchronous/asynchronous serial communications.

"An example of a late 1980s UART was the Intel 8450. In the 1990s, newer UARTs were developed with on-chip buffer space for data. This allowed higher transmission speed without data loss and without requiring such frequent attention from the computer. For example, the Intel 16550 has a 16 byte FIFO." (FOL 1993)

Like all other peripherals, communication peripherals have 4 types of peripheral registers:

data generally gives access to the RX and TX buffers

state report errors, medium state, completion of TX/RX, state of TX/RX buffers

configuration/control protocol settings; interrupt masks; TX/RX enabled/disabled

Because serial communications must be processed in a relatively quick manner, there are generally multiple interrupt triggers associated with a serial communication peripheral. Further, certain interrupt flags are automatically cleared (in order to save machine cycles) for e.g. the RX flag might be cleared as soon as the received data is accessed. A U(S)ART may be used to implement RS232 communications between microprocessors (when combined with appropriate level shifter(s)).

" (UART) An integrated circuit used for serial communications, containing a transmitter (parallel-to-serial converter) and a receiver (serial-to-parallel converter), each clocked separately.

The parallel side of a UART is usually connected to the bus of a computer. When the computer writes a byte to the UART's transmit data register (TDR), the UART will start to transmit it on the serial line. The UART's status register contains a flag bit which the computer can read to see if the UART is ready to transmit another byte. Another status register bit says whether the UART has received a byte from the serial line, in which case the computer should read it from the receive data register (RDR). If another byte is received before the previous one is read, the UART will signal an "overrun" error via another status bit.

The UART may be set up to interrupt the computer when data is received or when ready to transmit more data.

The UART's serial connections usually go via separate line driver and line receiver integrated circuits which provide the power and voltages required to drive the serial line and give some protection against noise on the line.

Data on the serial line is formatted by the UART according to the setting of the UART's control register. This may also determine the transmit and receive baud rates if the UART contains its own clock circuits or "baud rate generators". If incorrectly formatted data is received the UART may signal a "framing error" or "parity error".

Often the clock will run at 16 times the baud rate (bits per second) to allow the receiver to do centre sampling - i.e. to read each bit in the middle of its allotted time period. This makes the UART more tolerant to variations in the clock rate ("jitter") of the incoming data. "(FOL 1993)

In RS232 communication, 9 way D-type connectors and "25-way D-type connectors are common but often only three wires are connected - one ground (pin 7) and one for data in each direction. The other pins are primarily related to hardware handshaking between sender and receiver and to carrier detection on modems, inoperative circuits, busy conditions etc.

The standard classifies equipment as either Data Communications Equipment (DCE) or Data Terminal Equipment (DTE). DTE receives data on pin 3 and transmits on pin 2 (TD). A DCE EIA-232 interface has a female connector. DCE receives data from DTE on pin 2 (TD) and sends that data out the analog line. Data received from the analog line is sent by the DCE on pin 3(RD).

Originally DCE was a modem and DTE was a computer or terminal. The terminal or computer was connected (via EIA-232) to two modems, which were connected via a telephone line.

The above arrangement allows a computer or terminal to be connected to a modem with a straight-through (2-2, 3-3) cable. It is common, however, to find equipment with the wrong sex connector or with pins two and three reversed, requiring the insertion of a cable or adaptor wired as a gender mender or null modem. Such an adaptor is also required when connecting a computer directly to a terminal or to another computer without the use of modems. "(FOL 1993)

" A logic '1' ranges from -3V to -25V, but will typically be around -12V. A logical '0' ranges from 3V to 25V, but will typically be around +12V. Any voltage between -3V and +3V has an indeterminate logical state. If no pulses are present on the line the voltage level is equivalent to a high level, that is -12V. A voltage level of 0V at the receiver is interpreted as a line break or a short circuit. ... In the transmission of data there can either be no handshaking, hardware handshaking, or software handshaking. If no handshaking is used, then the receiver must be able to read the received characters before the transmitter sends another. ... Hardware handshaking involves the transmitter asking the receiver if it is ready to receive data. If the receive buffer is empty it will inform the transmitter that it is ready to receive data. Once the data is transmitted .... the transmitter is informed not to transmit any more characters .... The main hardware handshaking lines used for this purpose are:

- CTS - Clear to Send
- RTS - Ready to Send
- DTR - Data terminal ready
- DSR - Data set ready

... Software handshaking involves sending special control characters. ... There are two ASCII characters that start and stop communications. These are X-ON (^S, Cntrl-S or ASCII 11) and X-OFF (^Q, Cntrl-Q or ASCII 13). When the transmitter receives an X-OFF character it ceases communications until an X-ON character is sent. "(Buchanan and Wilson 2001)

Review Exercises

1. The parity bit in RS232 serial communications may be used for error detection. When Even parity is selected, the parity bit is set/cleared so that when combined with the data bits, there is an even number of '1' bits. In 7 bit, even parity communication, which of the following should have the parity bit set?

   (a) 1111 101

   (b) 1010 100

   (c) 1010 101

   (d) 1100 110

   (e) 1011 111

2. A device uses a proprietary synchronous communication protocol that requires 3 lines, Signal, Ground, and an externally controlled clock. The maximum rate at which data can be transmitted is 2400 bits/s. A message transaction involves 8 data bits and 3 overhead bits (start, stop and parity).

   We need to communicate between the PIC16F877 and this device. It has been suggested that we could use the UART to do so. Which of the following is a valid configuration value for the TXSTA register?

   (a) 1010 1111

   (b) 1100 1111

   (c) 1101 1110

   (d) 1010 1110

   (e) 0001 1111

3. We wish to configure the PC UART for 9600 baud communication. The crystal frequency typically used with these UART's is is 1.8432 MHz. Given that the UART samples the serial line 16 times per bit period, the divisor value must be set to:

   (a) 0x0C

   (b) 0x0F

   (c) 0x06

   (d) 0x40

   (e) 0x01

4. The PIC16F877 has built-in peripherals which support synchronous and asynchronous communication. The asynchronous communication peripheral can be used to interface with an RS232 signal (presuming appropriate signal level conversion). The peripheral does not support automatic parity generation/checking or flow control.

   For an application that needs to communicate using even parity, and software flow control, choose between bit-banging and using this peripheral, and justify your choice.

5. Your company is designing an independent device for monitoring RS232 traffic. The device should passively monitor electrical signals on the TX/RX lines of an RS232 serial cable, and display current and cumulative estimates of communication traffic. The team has decided to use the UART on a PIC16F877 to passively monitor the lines.

   Your job is to specify the electrical connection, and then write an appropriately commented code module which will allow the information transmitted in both directions to be stored in a circular buffer.

Lab 4 ID# ______

You should complete the pre-lab before coming to your lab session. Your Teaching Assistant/Demonstrator may refuse to allow you into your lab session if your pre-lab is incomplete, or if you are unduly late.
You will have 12 hours in the lab to complete the exercises. All answers should be written on this lab-sheet in PEN. Please do not attach any extraneous pieces of paper unless SPECIFICALLY asked to do so.
Please bring your PIC16F877 datasheet book (provided in ECNG2006) to the lab with you. You may need to refer to it in order to complete the pre-lab and lab exercises. Your lab submissions will be checked for unwarranted collusion, and unreferenced use of Intenet-available/other resources.
At the end of this unit the student will be able to:

- *produce programs to interface external peripherals/devices to the PIC16F877 microcontroller.*
- *produce programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and/or PPC datasheets/details, and respective language/instruction sets.*

This lab exercise will look at four tasks commonly performed during embedded application programming: multiplexing, interrupts, master/slave communication, PC communication.

Pre-Lab

1. Let's start with some general questions about parallel and serial communication.

   (a) What is the difference between parallel and serial communications. *3 marks*

   (b) What is a UART? In your own words, provide a brief description of it's function. *2 marks*

   (c) Each UART has 3 pins: transmit, receive, and ground. Draw a diagram to show connections between two UART's so that they can exchange information. *3 marks*

(d) When establishing a connection across an RS232 link, we must set the **baud rate**, **flow control** setting, **parity**, **# data bits** and **# stop bits** for the UART. In your own words (and using diagrams if needed), explain each of these terms. *5 marks*

(e) What would you expect to happen if two UARTS were configured differently, and they tried to communicate? Explain your answer. *2 marks*

(f) "Serial peripherals can be **poll**ed by the CPU **or** they can **interrupt** the CPU when they need service". Explain the highlighted terms in the context of the sentence, and list the relative advantages/disadvantages of the two methods. *4 marks*

2. Next, some questions specific to the PIC16F877. The USART allows the PIC to make and answer requests from another microprocessor. The Parallel Slave Port on the PIC16F877 allows an external "master" to retrieve information from, or send information to the PIC16F877.

   (a) Read the PIC16F877 datasheet (10.2.1, 10.2.2) and identify the data, configuration, state and control bits/registers for the USART operating in asynchronous mode. *5 marks*

Processor	Peripheral	Configuration	Control	State	Data
PIC16F877	USART				

   (b) The PIC16F877 has several banks in the data memory map. The bottom bytes of each bank are all mapped onto the same memory locations. Explain, in your own words, why it is useful to have memory mapped into the same location of each bank. *2 marks*

   (c) Identify the pins associated with the PSP, and the PC connections (for these pins) which exist in your circuit. *11 marks*

   There are switches connected to some of the pins. Choose two of these pins. What voltage do you expect to be applied to pin - ______ when:

   - the switch is on? ______ *1 mark*
   - the switch is off? ______ *1 mark*

   What voltage do you expect to be applied to pin - ______ when:

   - the switch is on? ______ *1 mark*
   - the switch is off? ______ *1 mark*

3. Let us start with a basic delay routine which we can use to build standard times. We will use instruction based delays. We can write a general delay, which will call a 1 millisecond delay routine `Dly1ms` as many times as specified in the working register `W`.

```
Dly         call    Dly1ms
            addlw   0xFF
            btfss   STATUS,Z
            goto    Dly
            return
```

4. We have used **macros** in the preceding code. The preprocessor replaces the macro name with the code included between the MACRO and ENDM directives. What are the names of the two macros? What does the code included in each macro do? *2 marks*

5. In the second macro, what is `dval`? *1 mark*

6. Identify at least **two** differences between using a macro and using a subroutine. *2 marks*

7. What possible advantage is there to using macros in an assembly language program? *1 mark*

8. Finally let us write/consider some support routines/algorithms which you will use later in the lab.

   (a) Write a 1 millisecond instruction-based delay routine for the PIC16F877 named `Dly1ms` whose last line will `return` on completion, which does **not** affect the value of the working register. You should assume a 4MHz clock. *6 marks*

   (b) Write a routine to configure and start Timer0 on the PIC16F877 so that Timer0 will count the number of **falling** edges it sees from an **external** input. *4 marks*

Helpsheet: Connecting to Bluetooth Modules

What is Bluetooth?

Ensure board is powered. Drop pin in program so that it is ready to receive connections Check Blue Soliel - make sure that connection is established to your module - check the ID number under module if unsure. Use the default pin 0000. Open PC - start service for serial port - note COM port Proceed to open Hyperterminal for that COM port.

HelpSheet: Hyperterminal and the PC serial port

Hyperterminal is the terminal emulation program supplied with the Windows Operating system. It may be used to communicate across any serial channel, including the RS232 communications ports (COMn:).

To start Hyperterminal:

- Click on the Start menu, and then select:
  Programs>Accesories>Communications>Hyperterminal.
- Click cancel on the box that is presented. You should now have a window entitled ”New Connection”.
- Choose File, Properties, this will bring up a communications box.
- Choose the relevant COM port from the connect using box.
- Click the Configure button – this will bring up a dialogue box which will allow you to choose the baud rate, data bits, parity and stop bits, and the flow control. Once you have chosen your settings, click OK twice. The settings should now appear in the status bar at the bottom of the window.
- Click Call> Call to start communicating. (You'll need to enter a name)
- Click Call> Disconnect to stop communicating.
- If you change the settings while connected, you must disconnect and reconnect for them to take effect.

HelpSheet: Frequently Asked Questions: Embedded Application Programming

multiplex ing parallel i/o devices

Q: Why do we use multiplexors when we can just connect each device to different pins?

A: Micro-controller (packages) have a limited number of i/o ports (pins). Multiplexing allows us to control more devices with the same number of pins.

Q: There must be some tradeoff ...

A: Yes - the effective response/control time of the devices will be reduced.

interrupts for device i/o handling

Q: What is an interrupt?

A: An interrupt is a

Q: Interrupts! They are so confusing! Won't polling be easier?

A: Polling only works well when events will ALWAYS occur in a known sequence. If the sequence is not known, a polling program runs the risk of getting stuck polling for one event while another one is occurring.

master /slave synchronous serial communications

Q: What does synchronous mean?

A: The word synchronous refers to things that happen at the same time. For communications that means that the sender and the receiver have an agreed clock signal, using which they send/recieve at agreed times.

Q: So does the slave do all the work (sending and receiving) in master/slave communications?

A: No. The master controls/generates the clock signal which is used to synchronize communication, however they are both sending and receiving - in fact the master may do more sending/recieving than the slave.

Personal Computer (PC) asynchronous serial communications

Q: Why would anyone want to communicate between the PC and a microcontroller?

A: Lot's of reasons ... you may want the PC to act as a storage device, or as a terminal emulator, or the microcontroller may be providing updated values for a PC program (e.g. MATLAB).

Q: Haven't we been using a serial connection all along?

A: Yes there was a USB serial cable for communication between the ICD and the MPLAB IDE.

Q: So what's the difference?

A: This time you will have two serial connection – one for MPLAB-ICD and a wireless Bluetooth serial connection between the PC and the PIC16F877 (without passing through the ICD). In practice, once you have finished developing your application, the ICD cable will no longer be necessary.

In the Lab ID# ____________

You are given the circuit diagram for a system board containing

- four 7-segment LED displays,
- four DIP switch package.
- an opto-switch,
- a joystick connected to an Analog-to-Digital converter,
- a Bluetooth module with antenna,

The board is connected via a ribbon cable to a board on which the PIC16F877 is mounted. The pin connection from the PIC16F877 to the system board are as follows:
Check that the circuit matches the circuit diagram on page 51; if there are any differences, make appropriate corrections to the diagram and/or circuit.
Please let the lab technician check your circuit before you proceed.

7 segment output w/timer to scroll between different outputs

Switches input - parallel port

Optoswitch - interrupt source - count up

Joystick - serial SPI - using the interrupts to drive an o around your segments.

Bluetooth - serial async - info to PC.

Challenge - drive the o around from the PC -arrow keys.

The PIC16F877 can also control a parallel bus connection. We are going to use the LCD functions we wrote in the pre-lab to write a simple program which will display a string, and a value.

1. Let us start with a basic program which will use the include file, and simply initialise the LCD. Make a new project, a new file, and enter the following code (N.B. the LCD include file must be in the same directory).

```
        INCLUDE <P16f877.inc>
        LIST P=16F877

        ORG     0x00
        nop
        goto    main

        ORG     0x10
main    bsf     STATUS,RP0
        movlw   B'00000110' ; we need this for PORTA
        movwf   ADCON1      ; to operate properly as digital pins
        bcf     STATUS,RP0
        call    LCDSetup

loop    goto loop
        sleep

        ORG     0x200       ; place this somewhere where it won't clash with the program

LCD_DATA        EQU PORTB
LCD_DATA_TRIS   EQU TRISB
LCD_CNTL        EQU PORTA
LCD_CNTL_TRIS   EQU TRISA
LCD_E           EQU 3
LCD_RW          EQU 2
LCD_RS          EQU 1
LCD_Vars        EQU 0x20     ; we need 4 locations starting from LCD_Vars in Block 0

        INCLUDE <LCDASM.INC>

        END
```

Compile and run all code, and make sure it works in the simulator.

Program and run the code on the cirucit using the ICD. If the program is working your LCD will initialise and display `Ok`. If it doesn't work, apply your troubleshooting skills to locating the problem. Use the space below to record your troubleshooting investigations.

Demonstrate working code to your TA/lecturer who will sign your script. *4 marks*

2. Choose an appropriate place in your code to place the following lookup tables.

```
lookup_nibble   addwf   PCL,F
                DT  "0123456789ABCDEF\0"
lookup_string   addwf   PCL,F
                DT  "The value is: \0"
```

Where did you put them? Explain why you chose that position. *2 marks*

Now, alter the main loop so that it uses a register named **indx** which starts from 0 and increments until we get to the end of the string. On each iteration, fetch the character from **lookup_string** associated with **indx**, and display it on the LCD using the LCD_SEND_CHAR routine.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *5 marks*

3. Strings can also be stored in RAM. Write a routine that will take the following 16-byte string:

```
MyStrInit   addwf   PCL,F
            DT      "The value is:  ",0
```

and copy it using indirect addressing to locations starting from `MyStr.` You should define the 16 spaces for `MyStr` using the `CBLOCK` directive.

Alter the main loop so that it will convert the number stored in a register `num` into the ASCII equivalent of the hexadecimal number, and store it in `MyStr` at positions 13 and 14, before displaying `MyStr` on the LCD display.

(Hint: Use the `lookup_nibble` table to translate each nibble of `num` separately.)

Initialise `num` with a value.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *10 marks*

4. Now alter the main loop so that:

   - the motor runs at a speed determined by the value written to the PSP from the PC,
   - Timer0 is configured as a counter, whose value is continuously copied to `num` for display.

   As the motor turns we should see the LCD displaying the number of encoder pulses counted by Timer0.

   Compile and test your code. Does it work as you expected? Make changes until it does.

   If your program didn't work on the first try what were the problems? How did you fix it?

   Write down the final version of your main loop here.

   Did this code work? Yes or No (Circle your answer)

   If so, demonstrate the working code to your TA/lecturer who will sign your script. *16 marks*

We would like to write a program for the PIC16F877, which communicates with the PC over an RS232 link, instead of across the PSP.

Disconnect the parallel cable and connect the serial cable.

Firstly, let us locate all the serial ports on the computer that you are using. Look at the back of the computer. How many serial ports do you see? ______________
(N.B. remember that the computer is DTE, so you are looking for **male** D-type connectors, either 9 pin or 25 pin.)
The computer "labels" it's serial ports COM1, COM2, COM3, COM4 etc Which serial port are you connected to? ______________
The circuit you have been provided with has a PIC in a header, and an RS232 driver IC. What is the IC part #? ______________
You have already located the registers of the USART peripheral in the PIC (as part of your pre-lab exercises). We will start with a simple program which polls for a character, and then echoes the character received to the transmit register. We will use the following communication settings:

- 2400 baud
- 8 data bits
- no parity bits
- no flow control
- 1 stop bit

The structure of the program will be:

```
configure and control
loop:
    wait for state
    read data
    write data
    wait for state
goto loop
```

Fill in appropriate settings, and register/flag names in the code below. *7 marks*

```
            LIST P=16F877
            INCLUDE <P16F877.inc>

            ORG     0x00
            nop
            goto    main

            ORG     0x10
main        call    UART_Cnfg
            goto    UART_poll

UART_Cnfg   bsf     STATUS,RP0 ; Go to Bank1
            movlw   __________ ; Set Baud rate to 2400
            movwf   SPBRG
            movlw   __________ ; 8-bit transmit, transmitter enabled,
            movwf   TXSTA      ; asynchronous mode, low speed mode
            bcf     STATUS,RP0 ; Go to Bank 0
            movlw   __________ ; 8-bit receive, receiver enabled,
            movwf   RCSTA      ; serial port enabled
            return

UART_poll   clrw
loop        btfss   ______,RCIF; wait on char
            goto    loop
nchar       movf    _________,W; transmit what you rx (NB reading clears the flag)
            movwf   ___________
wait        btfss   ______,TXIF; flag clears when character is gone from buffer
            goto    wait
            goto    loop
            sleep

            END
```

Start up MPLAB, and create a new project and file for the code. Compile your code, and make sure there are no errors. Power up your circuit, and download the code.

Start Hyperterminal (as per helpsheet) and choose the required communication settings. Connect to the communication channel, and then run the program on the PIC16F877.

What happens when you type in the Hyperterminal window? Is it what you expected? If not, why not, and/or what did you do to fix it?

What happens if you disconnect, change the communication settings, and then reconnect and start typing? Is that also what you expected? If not, why not, and/or what did you do to fix it?

Explain your observations. *4 marks*

Change the settings in HyperTerminal back to their original values before proceeding.

We can modify the previous example, so that instead of just repeating what it received, the micro-controller changes the content somehow.

Add the routine `lookup_string` (that you used earlier) between `return` and `UART_poll`.

Change the original program by inserting a `call` to your routine, on a new line just after the `nchar` labelled line.

Compile your code, and make sure there are no errors. Download the code and test your program by typing different characters in HyperTerminal. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your routine here. *5 marks*

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script.

So far we have been polling. Let us convert the polling program to an interrupt driven one. We can rewrite the main part of the code as:

configure and **control**

loop:

– do something else

goto loop

and then write an interrupt service routine, and handlers for the individual interrupts.

Let us start with the interrupt subroutine. Choose appropriate locations to store `_W`, `_STATUS` and `_PCLATH`, so that the code will work in all banks. Justify your choices. *3 marks*

```
_W          EQU     _______
_STATUS     EQU     _______
_PCLATH     EQU     _______

isr                           ; save all registers
            movwf   _W        ; Copy W to TEMP register
            swapf   STATUS,W  ; Swap status to be saved into W
            clrf    STATUS    ; bank 0, clears IRP,RP1,RP0
            movwf   _STATUS   ; Save status to bank zero STATUS_TEMP register
            movf    PCLATH, W ; Only required if using pages 1, 2 and/or 3
            movwf   _PCLATH   ; Save PCLATH into W
            clrf    PCLATH    ; Page zero, regardless of current page

           ; no need to clear flags as they change automatically for this device
            btfsc   PIR1, RCIF
            goto    rx_handler

rtn_isr     movf    _PCLATH, W ; Restore PCLATH
            movwf   PCLATH    ; Move W into PCLATH
            swapf   _STATUS,W ; Swap STATUS_TEMP register into W
                              ; (sets bank to original state)
            movwf   STATUS    ; Move W into STATUS register
            swapf   _W,F      ; Swap W_TEMP
            swapf   _W,W      ; Swap W_TEMP into W
                              ; all registers restored
            retfie
```

Next we have the individual handlers. We would like to transmit as soon as we receive, so the transmit handler can be empty for now.

```
rx_handler ; on rx, tx the next character
            movf    ________,W ; read the byte to clear the flag
            movwf   ___________
            goto    rtn_isr
```

Now for the associated main loop, fill in the blanks in the following code . *2 marks*

```
            LIST P=16F877
            INCLUDE <P16F877.inc>

            ORG     0x00
            nop
            goto    main

            ORG     0x04
            goto    isr

            ORG     0x10
main        call    UART_Cnfg
            goto    otherloop

UART_Cnfg   bsf     STATUS,RP0 ; Go to Bank1
            movlw   __________ ; Set Baud rate to 2400
            movwf   SPBRG
            movlw   __________ ; 8-bit transmit, transmitter enabled,
            movwf   TXSTA      ; asynchronous mode, low speed mode
            bsf     PIE1, RCIE ; enable receive interrupts
            bcf     STATUS,RP0 ; Go to Bank 0
            movlw   __________ ; 8-bit receive, receiver enabled,
            movwf   RCSTA      ; serial port enabled
            bsf     INTCON, ___; enable peripheral & general interrupts
            bsf     ______, PEIE
            bsf     RCSTA , CREN
            return

otherloop   goto    otherloop
            sleep

            END
```

Type in all your code, compile and run it. Test your program by using HyperTerminal.

Does it work as you expected? If not, why not, and/or what did you do to fix it?

Demonstrate the working code to your TA/lecturer who will sign your script. *4 marks*

Previously, we adjusted motor speed from the PC using the PSP. Write a program which will change motor speed based on a single digit typed in HyperTerminal, where `'0'`means stop, `9` means full speed, and digits `1` through `8` adjust speed proportionally between no and full speed.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *10 marks*

Modify your program so that it in response to a single character, it sends the 3-character string '`OK\n\0`'. You should use TX interrupts, and indirect addressing in a way which will allow you to send any null-terminated string. *15 marks*

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script.

Modfy your program to report the measured motor speed(Timer0) across the serial link when the character `S` is typed in HyperTerminal. *20 marks*

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script.

Challenge: The pre-lab asked you to explain an algorithm for manipulating regularly sampled data.

1. Use either instruction-based delays or Timer1 (with/without interrupts) to regularly sample the distance turned by the motor (Timer0 counts). *20 marks*
2. All results should be reported serially using RS232. *10 marks*
3. Bonus(30 marks): For the sampled data, implement the algorithm you explained in the pre-lab.

Compile and test your code. Does it work as you expected? Make changes until it does.
If your program didn't work on the first try what were the problems? How did you fix it?

Attach a printout of your final code.

Did this code work? Yes or No (Circle your answer)
If so, demonstrate the working code to your TA/lecturer who will sign/stamp your script/printout.

Total marks 194.
This exercise is worth 15% of your ECNG2007 lab mark.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this lab exercise.
  - produce programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and/or PPC datasheets/details, and respective language/instruction sets.
- Which aspect of this lab exercise did you have the most difficulty understanding?
- Which aspect of this lab exercise did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this lab exercise.
- Identify one way in which this lab exercise could be improved.

33 μP Parallel Interface

At the end of this unit the student will be able to:

> *interface external peripherals/devices to the PIC16F877 microcontroller using a parallel bus, and interface the PIC16F877 microcontroller as an external device on a parallel bus (memory mapped).*

A bus is a collection of lines used to transfer information between two devices. The bus master is responsible for initiating bus transactions, and may also provide address, clock and/or enable signals for the external (slave) device.

Typically a bit-parallel bus has selection/address lines, data lines, and read/write control lines. While some bit-parallel bus devices support addressing, most devices do not include such functionality. Bit-parallel bus addressing can be implemented using logic circuitry and/or decoders with inputs from the parallel bus address lines, and outputs tied to the enable lines (or clock signals/local address lines) of the respective slave devices.

A bit-parallel bus, for which the PIC16F877 is the bus master/slave, may be simulated using the I/O ports. There is no peripheral available in the PIC16F877 which supports parallel bus mastering, however the Parallel Slave Port peripheral will independently handle receipt and transmission of data, as demanded by an external master.

A bit-parallel bus may be synchronous or asynchronous. In either case, the software/peripheral on the master must adhere to the read/write timing specifications of the external (slave) device. The circuit designer should verify that the required signal rise/fall/hold times of the slave can be met by the master.

When the Parallel Slave Port is used, the master should observe the PIC's response timing. The CU responds to external signals of the PSP in Q2 of the instruction cycle. Therefore to ensure that the signals are properly processed, the master must hold them for 5 clock oscillations (1.25 instruction cycles) to guarantee that a Q2 has passed. The PIC sets it's internal event flags for the PSP at the start of Q4 following the Q2 in which the event occurred. The PSP flags may either be polled in software, or used as interrupt triggers.

A slave port may be simulated using a standard I/O port by either:

- polling the simulated enable/chip select/clock line at regular intervals (use a timer interrupt?)
- or interrupting on change on the enable/chip select/clock line

The PIC16F877 may also be used to master other devices across a bit-parallel bus. Where there are insufficient port pins available on the PIC16F877 for all the bus lines, special-purpose interfaces, or even general purpose serial-to-parallel converters may be employed. The PIC16F877 then pulses out the bits one at a time, before triggering the bus operation. Some devices also operate at different data-word widths.

One such device is the standard character display LCD panel. The controller for such panels are all instruction-compatible with the Hitachi 44780 IC, however there may be some variations in the timing specifications. You should always check the timing specifications for your particular LCD controller. Operation of a typical panel is described in (Predko 2000c).

The minimum cycle time for a transaction across the bus is specified as $1000ns == 1\mu s$ for the HD44780. Therefore a PIC with a 4MHz clock i.e. instruction cycle $1\mu s$, can change the bus signals with each instruction. Delays in the code will be necessary if a faster clock signal (e.g. 20MHz; instruction cycle $250ns$) is used.

Because there is no peripheral available to perform these operations, mastering a parallel bus on the PIC16F877 can consume significant resources (and cannot be interrupted for fear of violating timing).

An alternative to bit-banging in the main loop, is to model the system using state machines and use the timer interrupts to generate the state transitions. This leads to rather complicated interrupt software, but will allow the PIC to perform non-time critical tasks in the main loop.

The data transfer rate across a bit-parallel bus is generally faster than across a bit-serial bus. Use of a reduced bit-width bus, or a serial-to-parallel converter, will reduce the data transfer rate. With slaves which have significantly slower timing than the microprocessor however, there will be little to no slow-down, at the cost of the additional circuitry, and more complicated software. One way of countering increased software complexity, is to use a serial peripheral (e.g. USART) to independently send out the bit-stream.

Review Exercises

1. A PIC16F877 is used as the controller on an interface card built for a personal computer. The interface card has a 10MHz oscillator which generates the clock signal for the PIC16F877. The computer processor communicates with the PIC16F877 using Parallel Slave Port. In order to ensure that timing requirements are met, the computer processor must hold the Chip Select signal line low for:

   (a) $5\mu s$

   (b) $1.25\mu s$

   (c) $0.5\mu s$

   (d) $0.25\mu s$

   (e) none of the above

2. A device uses a proprietary synchronous communication protocol that requires 3 lines, Signal, Ground, and an externally controlled clock. The maximum rate at which data can be transmitted is 2400 bits/s. A message transaction involves 8 data bits and 3 overhead bits (start, stop and parity).

   Four of these devices are to be interfaced to the PIC16F877. It is suggested that a parallel bus could be used to send/receive data from all devices simultaneously. Which of the following is NOT a viable alternative?

   (a) Clock lines: tied together; Ground lines: tied together; Data lines: tied together

   (b) Clock lines: individual; Ground lines: tied together; Data lines: tied together

   (c) Clock lines: tied together; Ground lines: tied together; Data lines: individual

   (d) Clock lines: tied together; Ground lines: individual; Data lines: individual

(e) Clock lines: individual; Ground lines: individual; Data lines: tied together

3. Two 2-line LCD displays use different LCD controllers. The controllers have compatible instructions, and the same transaction sequence across the parallel bus, but they have different timing requirements.

Controller	Rise/Fall Time	Setup Time	Hold Time	Wait Time
A	40ns	100 ns	50 ns	40 μs
B	20ns	80 ns	80 ns	20 μs

What timing should you use in your system to ensure that it will work for either LCD display?

	Rise/Fall Time	Setup Time	Hold Time	Wait Time
(a)	20 ns	100 ns	80 ns	40 μs
(b)	40 ns	80 ns	50 ns	20 μs
(c)	20 ns	80 ns	80 ns	20 μs
(d)	40 ns	100 ns	50 ns	20 μs
(e)	20 ns	80 ns	80 ns	20 μs

4. An external microprocessor is to be used to access the PIC16F877, through the Parallel Slave Port.

   (a) What connections need to be made between the PIC and the microprocessor bus, so that the PIC16F877 will be mapped into the external microprocessor's memory address space?

   (b) With reference to the timing diagrams, identify the timing constraints which the *external* microprocessor should observe for reliable operation.

   (c) The deadline for the project is so tight that you cannot wait for the hardware to be completed before starting software development. Discuss the software tool-chain and hardware tool requirements needed to facilitate this.

5. Your company is designing an independent device for monitoring RS232 traffic. The device should passively monitor electrical signals on the TX/RX lines of an RS232 serial cable, and display current and cumulative estimates of communication traffic. The team has decided to use an LCD display module to show the traffic estimates. Your job is to write the module which will send a single character to an LCD display which was initialised in 4-wire mode. Write appropriately commented code for the PIC16F877.

Lab 4 Alternate–developed A. Sinanan, Mar. 03, modified Mar. 07

In this lab exercise we will look at performing serial communication between the PIC16F877 microcontroller and the PC using the RS232 serial port. We will also look at performing bit-parallel communication between the PIC16F877 microcontroller and other devices. Let's clear up some questions you may have:

Q: Why would anyone want to communicate between the PC and a microcontroller?

A: Lot's of reasons ... you may want the PC to act as a storage device, or as a terminal emulator, or the microcontroller may be providing updated values for a PC program (e.g. MATLAB).

Q: Haven't we been using a serial connection all along?

A: Yes there was a USB serial cable for communication between the ICD and the MPLAB IDE.

Q: So what's the difference?

A: This time you will have two serial cables – one for MPLAB-ICD and the other between the PC and the PIC16F877 (without passing through the ICD). In practice, once you have finished developing your application, the ICD cable will no longer be necessary.

Q: What does parallel communication involve?

A: It involves using multiple signal lines simultaneously, some of the lines will carry data, and others will carry additional information which will facilitate the transfer (like address, clock, enable etc.).

Q: Which peripheral on the PIC16F877 does parallel communication?

A: The Parallel Slave Port can be used for parallel communication which is initiated by another system. For parallel communication which is initiated by the PIC16F877 you must use one of the standard I/O ports.

Q: The PIC16F877 and the Intel x86 PC both have UART peripherals. So why doesn't the x86 PC have a slave port, or the PIC16F877 have a parallel port?

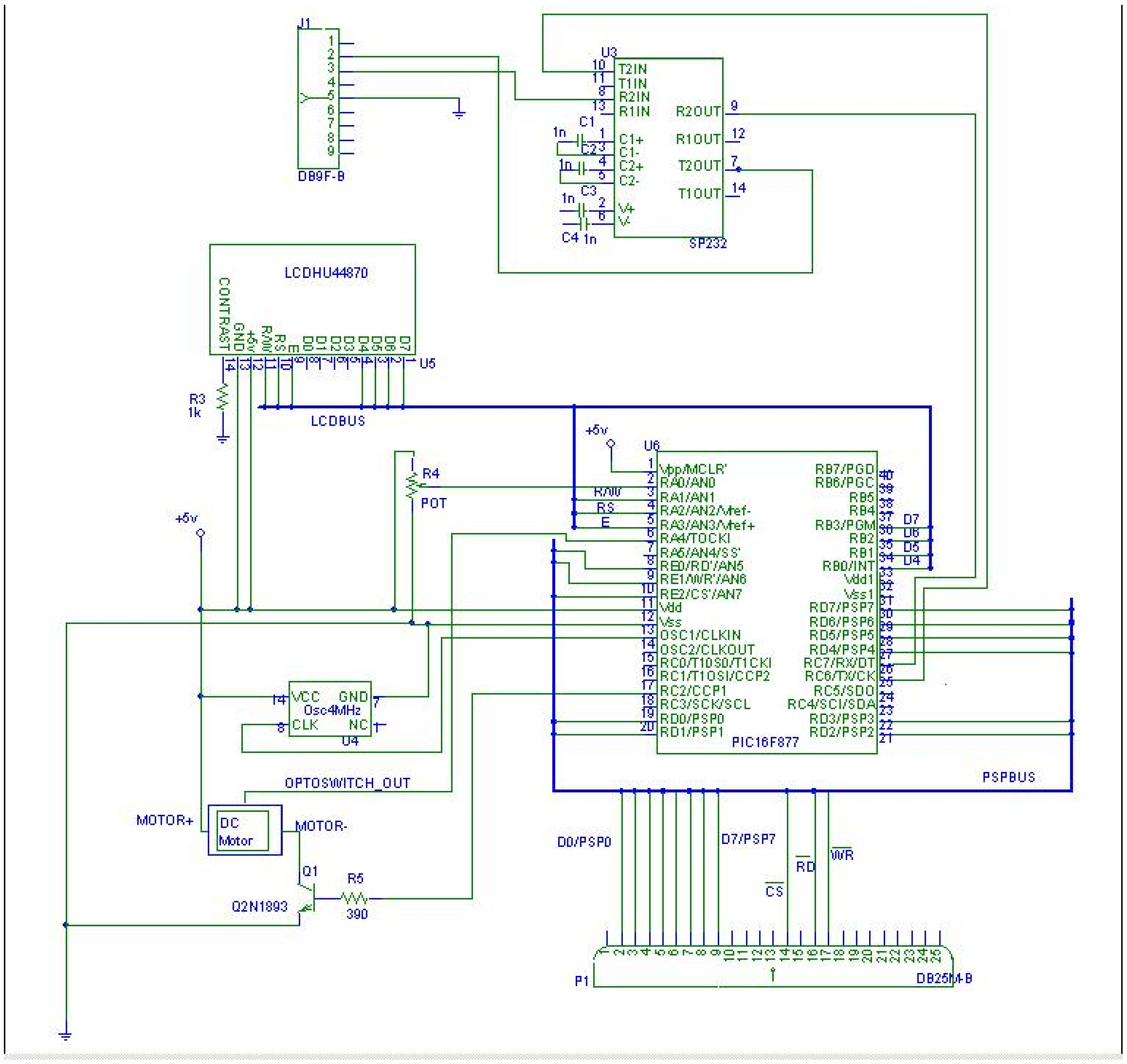
A: The PIC family of microcontrollers were originally designed to function as peripherals within a PC. The slave port feature is so that they can respond to the PC processor. The PC doesn't need that feature. The PC parallel port, was intended for the original display device: printer/tele-type. The PIC microcontrollers never needed that feature.

Q: So why do we have to consider parallel communication when serial communication can do the same job with less pins over longer distances?

A: The primary reason is speed. Given a constant rise/fall time for signals on a line, data can be transferred much more quickly over a parallel bus.

You are given the circuit diagram for a system containing

- a PIC16F877,
- a potentiometer,
- an LCD display,
- a DC motor fitted with switching transistor & incremental encoder disc + opto-switch,
- a connection from the PIC16F877 UART to the serial port of a PC,
- a connection from the PIC16F877 PSP to the parallel port of a PC.



Pre-Lab ID# ______

You should complete the pre-lab before coming to the lab. You may not be allowed in the lab if your pre-lab is incomplete. You will require 6-12 hours in the lab to complete the exercises. Unless otherwise specified, answers should be written on this lab-sheet. Please do not attach any extraneous pieces of paper. You may keep the help-sheets if you would like to.

At the end of this unit the student will be able to:

- *perform I/O programming on the Personal Computer (PC)*
- *produce programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and/or PPC datasheets/details, and respective language/instruction sets.*
- *interface external peripherals/devices to the PIC16F877 microcontroller using a parallel bus, and interface the PIC16F877 microcontroller as an external device on a parallel bus (memory mapped).*

1. Let's start with some general questions about parallel and serial communication.

   (a) What is the difference between parallel and serial communications. *3 marks*

   (b) What is a UART? In your own words, provide a brief description of it's function. *2 marks*

   (c) Each UART has 3 pins: transmit, receive, and ground. Draw a diagram to show connections between two UART's so that they can exchange information. *3 marks*

(d) When establishing a connection across an RS232 link, we must set the **baud rate**, **flow control** setting, **parity**, **# data bits** and **# stop bits** for the UART. In your own words (and using diagrams if needed), explain each of these terms. *5 marks*

(e) What would you expect to happen if two UARTS were configured differently, and they tried to communicate? Explain your answer. *2 marks*

(f) The MAX232 IC and its clones are often used in serial communication circuits. In your own words, briefly describe what it does. *2 marks*

(g) "Serial peripherals can be **poll**ed by the CPU **or** they can **interrupt** the CPU when they need service". Explain the highlighted terms in the context of the sentence, and list the relative advantages/disadvantages of the two methods. *4 marks*

2. Next, some questions specific to the PIC16F877. The USART allows the PIC to make and answer requests from another microprocessor. The Parallel Slave Port on the PIC16F877 allows an external "master" to retrieve information from, or send information to the PIC16F877.

   (a) Read the PIC16F877 datasheet (10.2.1, 10.2.2) and identify the data, configuration, state and control bits/registers for the USART operating in asynchronous mode. *5 marks*

Processor	Peripheral	Configuration	Control	State	Data
PIC16F877	USART				

   (b) The PIC16F877 has several banks in the data memory map. The bottom bytes of each bank are all mapped onto the same memory locations. Explain, in your own words, why it is useful to have memory mapped into the same location of each bank. *2 marks*

   (c) The parallel slave port timing diagrams for the PIC16F877 appear in the datasheet. For a PIC16F877 which is clocked at 4MHz, determine an appropriate signal sequence in order for the PSP value to be correctly read. (The PSP should be turned off at the end of the sequence). *5 marks*

   (d) Identify the pins associated with the PSP, and the PC connections (for these pins) which exist in your circuit. *11 marks*

   There are switches connected to some of the pins. Choose two of these pins. What voltage do you expect to be applied to pin - ________ when:

   - the switch is on? ________ *1 mark*
   - the switch is off? ________ *1 mark*

   What voltage do you expect to be applied to pin - ________ when:

   - the switch is on? ________ *1 mark*

- the switch is off? ______________ *1 mark*

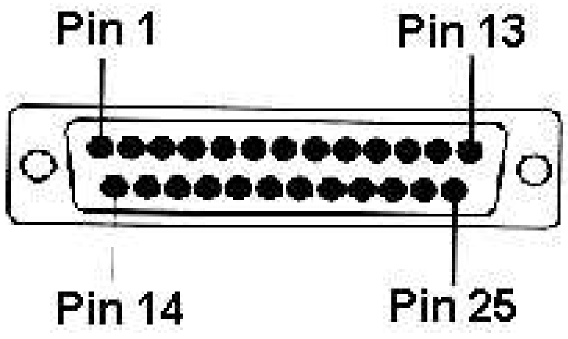
3. Now some questions specific to the PC.

(a) A PC can support more than one serial port or parallel port peripheral. These are typically labelled COMx or LPTx. Each port has it's own **base address**. In your own words, what is the **base address**? *2 marks*

(b) Now let us look at the parallel port on the PC. Below is the diagram of the male connector which will fit into the parallel port on the PC.

i. Draw one circuit with a switch for illustration; such that when the switch is ON the voltage at the pin is 0 volts and 5 volts when it is off. Use one of the Input Pins on the parallel port. What is the polarity of the signal connected to your chosen pin? *3 marks*

ii. Draw one circuit with an LED for illustration; such that the LED is ON when the voltage at the pin is 5 Volts and 0 Volts when it is off. Use one of the Output Data Lines on the Parallel Port. What is the polarity of the signal connected to your chosen pin? *3 marks*



Pins	Description
2 – 9	Output Data Lines
18 – 25	Ground
10	Acknowledge
11	Busy
12	Out of Paper
13	Select
15	Error
1	Strobe

iii. The parallel port designations are shown above. Use this information to complete the table below for the circuit provided: *4 marks*

Switch Number	Pin Number	Pin Name
1		
2		
3		
4		

4. The LCD display is a device which must controlled by an external "master" across a parallel bus. LCD displays typically use a standard controller chip (e.g. HD44780) which can communicate with the master. The HD44780 LCD character display controller can be operated in two interface modes: 4 bit (7 wire) and 8 bit (11 wire); where the number of bits indicates the number of data lines.

   (a) The KS066U is an HD44780 instruction-compatible controllers which has different timing under certain conditions. The initialization sequence for 4 bit mode appears in each datasheet. Write the sequence of instructions, and recommend initialization timings which would be valid for both controllers so that either would initialise in 4 bit mode with the block cursor on. *4 marks*

   (b) We will attempt to write routines which will drive an HD44780 compatible display. Let us start with a basic delay routine which we can use to build standard times. We will use instruction based delays. We can write a general delay, which will call a 1 millisecond delay routine `Dly1ms` as many times as specified in the working register `W`.

```
Dly         call    Dly1ms
            addlw   0xFF
            btfss   STATUS,Z
            goto    Dly
            return
```

   Using this routine, we can write a 4-bit mode initialization routine for the LCD. Assume that the 4 data bits of the LCD (`D7:D4`) are connected to the lower half of a PORT named `LCD_DATA`, and that the LCD control lines (`RW`, `RS`, `E`) are connected to the bits (`LCD_RW`, `LCD_RS`, `LCD_E`) of a port named `LCD_CNTL`. The data direction registers associated with the ports are `LCD_DATA_TRIS` and `LCD_CNTL_TRIS`. It is assumed that LCD code is always executed from Bank 0. Compare the following code to the datasheet timings.

```
LLCDTrigger MACRO
                bsf     LCD_CNTL, LCD_E
                bcf     LCD_CNTL, LCD_E
                movlw   D'15'
                call    WAIT_MS
            ENDM

SLCDTrigger MACRO
                bsf     LCD_CNTL, LCD_E
```

```
        bcf     LCD_CNTL, LCD_E
        movlw   D'1'
        call    WAIT_MS
    ENDM
```

```
LCDTrigger  MACRO
                bsf     LCD_CNTL, LCD_E
                bcf     LCD_CNTL, LCD_E
            ENDM

SetData     MACRO dval
                movlw   0xF0
                andwf   LCD_DATA,W
                iorlw   dval
                movwf   LCD_DATA
            ENDM

LCDInit         ; Init LCD
            movlw   LCD_CNTL_MASK
            iorwf   LCD_CNTL,F    ; Clear port and wait for settling
            movlw   D'150'
            call    WAIT_MS

            movlw   B'0011'       ; Trigger port for first time
            movwf   LCD_DATA
            LLCDTrigger

            movlw   B'0011'       ; Trigger port for first time
            movwf   LCD_DATA
            LLCDTrigger

            movlw   B'0010'       ; Trigger port for first time
            movwf   LCD_DATA
            LLCDTrigger

            movlw   B'0010'       ; Trigger port for second time
            movwf   LCD_DATA
            movlw   B'1000'       ; Follow with format
            LCDTrigger
            movwf   LCD_DATA
            SLCDTrigger

            movlw   B'00001110'     ; display and cursor on
            call    SEND_LCD_CMD
            movlw   B'00000001' ; OK now clear display
            call    SEND_LCD_CMD
            return
```

i. Use the space below to draw a flowchart to match the preceding code for 4-bit initialisation. *6 marks*

ii. Compare your flowchart to the one in the datasheet. Are there any differences? Which algorithm (as illustrated in the two flowcharts) is preferable? Explain your answer. *2 marks*

iii. We have used **macros** in the preceding code. The preprocessor replaces the macro name with the code included between the MACRO and ENDM directives. What are the names of the two macros? What does the code included in each macro do? *2 marks*

iv. In the second macro, what is `dval`? *1 mark*

v. Identify at least **two** differences between using a macro and using a subroutine. *2 marks*

vi. What possible advantage is there to using macros in an assembly language program? *1 mark*

vii. Now let us try simple routines to:

- determine if the controller is busy
- send a command
- send a character

We will use two storage variables for these routines: `LCD_CHAR` to store the char or command sent/recieved and `LCD_TEMP` to store the LCD status byte (including busy flag).

Let us start with a routine to determine if the controller is busy. Read the datasheet for the LCD commands, and fill in appropriate comments for the code below. *9 marks*

```
Busy_check  bsf      STATUS, RP0
            movlw    0x0F
            iorwf    LCD_DATA_TRIS,F     ; _______________________________
            bcf      STATUS, RP0

            bcf      LCD_CNTL, LCD_RS
            bsf      LCD_CNTL, LCD_RW    ; _______________________________

            bsf      LCD_CNTL, LCD_E
            nop
            swapf    LCD_DATA, W
            bcf      LCD_CNTL, LCD_E     ; _______________________________

            andlw    0xF0
            movwf    LCD_TEMP            ; _______________________________

            bsf      LCD_CNTL, LCD_E
            nop
            movf     LCD_DATA, W
            bcf      LCD_CNTL, LCD_E     ; _______________________________

            andlw    0x0F
            iorwf    LCD_TEMP, F         ; _______________________________

            btfsc    LCD_TEMP, 7
            goto     Busy_check          ; _______________________________

            bcf      LCD_CNTL, LCD_RW    ; _______________________________

            bsf      STATUS, RP0
            movlw    0xF0
            andwf    LCD_DATA_TRIS,F
            bcf      STATUS, RP0         ; _______________________________

            return
```

viii. Our next routine sends a command to the LCD. It will check if the LCD is busy before attempting to send the command. Read the LCD datasheet to determine the signal line settings needed to send a command. Insert appropriate comments in the following code. *2 marks*

```
SEND_LCD_CMD    movwf    LCD_CHAR
                call     BUSY_CHECK

                swapf    LCD_CHAR, W  ; ____________________
                andlw    0x0F
                movwf    LCD_DATA
                bcf      LCD_CNTL, LCD_RW
                bcf      LCD_CNTL, LCD_RS
                movf     LCD_CHAR, W  ; ____________________
                andlw    0x0F
                LCDTrigger
                movwf    LCD_DATA
                LCDTrigger
                return
```

ix. Modify the above routine so that it will send a character. Call your modified routine `SEND_LCD_CHAR`, and write it in the space below. *1 mark*

x. Now, use the MPASM help, and explain how the `CBLOCK` directive works *1 mark*

Finally, some general routines to clear the display, and setup the associated ports.

```
LCDClr      movlw   B'00001110'   ; display and cursor on
            call    SEND_LCD_CMD
            movlw   B'00000001'   ; OK now clear display
            call    SEND_LCD_CMD
            return
LCDSetup    clrf    LCD_CNTL
            clrf    LCD_DATA
            bsf     STATUS, RP0
            movlw   LCD_CNTL_MASK
            andwf   LCD_CNTL_TRIS,F
            movlw   0xF0
            andwf   LCD_DATA_TRIS,F
            bcf     STATUS, RP0
            call    LCDInit
            movlw   'L'
            call    SEND_LCD_CHAR
            movlw   'C'
            call    SEND_LCD_CHAR
            movlw   'D'
            call    SEND_LCD_CHAR
            movlw   ' '
            call    SEND_LCD_CHAR
            movlw   'O'
            call    SEND_LCD_CHAR
            movlw   'K'
            call    SEND_LCD_CHAR
            return
```

Place all of this code for the LCD in an include file named `lcdasm.inc`. At the top of the file, place the following declarations/comments:

```
; based on http://www.sonic.net/~schlae/lcd.asm
; assumes that: we are connected to the lower 4 bits of the Data Port
; level triggered LCD; code called from Data Bank 0; any bits of CNTL port used
; all variables are in block 0 and that the following are defined e.g.:
;LCD_DATA        EQU PORTB
;LCD_DATA_TRIS   EQU TRISB
;LCD_CNTL        EQU PORTA
;LCD_CNTL_TRIS   EQU TRISA
;LCD_E           EQU 3
;LCD_RW          EQU 2
;LCD_RS          EQU 1
; we need 4 locations starting from LCD_Vars in Block 0; define LCD_Vars accordingly
        CBLOCK LCD_Vars
            LCD_Count
            LCD_Count2
            LCD_CHAR
            LCD_TEMP
        ENDC

LCD_CNTL_MASK    EQU ~((1<<LCD_E) | (1<<LCD_RW) | (1<<LCD_RS))
```

5. Finally let us write/consider some support routines/algorithms which you will use later in the lab.

   (a) Write a 1 millisecond instruction-based delay routine for the PIC16F877 named `Dly1ms` whose last line will `return` on completion, which does **not** affect the value of the working register. You should assume a 4MHz clock. *6 marks*

   (b) Write a routine to configure and start Timer0 on the PIC16F877 so that Timer0 will count the number of **falling** edges it sees from an **external** input. *4 marks*

   (c) Shift and rotate operations can be used to perform fast division by powers of 2. Using a diagram, explain how this works. *2 marks*

(d) Masking operations can be used to perform modulo (reminder) operations by powers of 2. Using a diagram, explain how this works. *2 marks*

(e) Identify and describe an algorithm for: (any ONE of the following)

- estimating the first order derivative,
- filtering to remove random or periodic noise,
- integrating,
- predicting the next reading,

from regularly sampled data values. *3 marks*

Helpsheet – A/D and PWM calculations

How to calculate the voltage represented by ADRESH (w/left-justified result):

The A/D answer is 10 bits long.
The user reads the 8 most significant bits (i.e. the 8 most left most bits in `ADRESH`).
Let 255 represent V_{REF+} volts and 0 represent V_{REF-} volts.
`ADRESH` will represent $\frac{V_{REF+}-V_{REF-}}{256} \times ADRESH$ volts.

How to calculate the PWM frequency and pulse width

"The PWM frequency is defined as 1/[PWM period]." PIC (2001)

To set the PWM period, we write values to PR2, and set the TMR2 prescaler (bit pattern set in the two LSB of T2CON).

PWM period = [(PR2)+1]$\times 4 \times T_{OSC} \times$ (prescale value)

To set the duty cycle we write values to CCP1RL and two lower bits. If we ignore the lower 2 bits of the duty cycle then

PWM duty cycle = CCPR1L$\times 4 \times T_{OSC} \times$ (prescale value)

A/D Timing

A/D sampling time = min'm acquisition time ($\approx 20\mu s$) + # bits (10) * conversion time/bit ($T_{AD} > 1.6\mu s$) + min'm wait time ($2T_{AD}$)

Acquisition time is the time that the A/D modules holding capacitor is connected to the external voltage level. Conversion time is started when the GO bit is set, and includes conversion time for each bit, and a minimum wait time. The signal used to determine T_{AD} is software selectable as either an internal RC oscillator with a period of between 2 and $6\mu s$, or can be derived from the chip clock frequency T_{OSC} with scale factor (1/2, 1/8, 1/32). Relative to the instruction cycle, this sampling time is slow, and the A/D should not be used to measure rapidly changing signals.

HelpSheet: Hyperterminal and the PC serial port

Hyperterminal is the terminal emulation program supplied with the Windows Operating system. It may be used to communicate across any serial channel, including the RS232 communications ports (COMn:).

To start Hyperterminal:

- Click on the Start menu, and then select:
  Programs>Accesories>Communications>Hyperterminal.
- Click cancel on the box that is presented. You should now have a window entitled "New Connection".
- Choose File, Properties, this will bring up a communications box.
- Choose the relevant COM port from the connect using box.
- Click the Configure button – this will bring up a dialogue box which will allow you to choose the baud rate, data bits, parity and stop bits, and the flow control. Once you have chosen your settings, click OK twice. The settings should now appear in the status bar at the bottom of the window.
- Click Call> Call to start communicating. (You'll need to enter a name)
- Click Call> Disconnect to stop communicating.
- If you change the settings while connected, you must disconnect and reconnect for them to take effect.

HelpSheet: Connecting circuits to the Serial port

Level conversion Strictly speaking, driver circuitry must be designed/used in order to convert signals from RS232 signal levels to/from TTL level UART signals. " The types of driver ICs used in serial ports can be divided into three general categories:

- Drivers which require plus (+) and minus (-) voltage power supplies such as the 1488 ...
- Low power drivers which require one +5 volt power supply. This type of driver has an internal charge pump for voltage conversion ....
- Low voltage (3.3V) and low power drivers which meet the EIA-562 standard ...

... To determine the type of driver used on a serial port without looking at the ICs, put a 3kohm load on the driver output line to signal ground and measure the voltage. If possible, measure the voltage under both (+) and (-) voltage conditions. These voltage measurements should give values as shown below.

1488 Driver	±9 volts $< V_{out} < \pm11$ volts
Charge Pump driver	±7.5 volts $< V_{out} < \pm8$ volts
EIA562 driver	±3.7 volts $< V_{out} < \pm5$ volts

" – from Tips for using Port Powered Converters, B&B electronics, Technical Article #2, May 1995, `http://www.bb-europe.com/bb-euro/literature/tech/b&b2.pdf`

Some alternative strategies to using driver IC's:
(see `http://www.efplus.com/techref/io/serial/ttl-rs232.htm`)

TTL to RS232 – use two TTL outputs (a single output and it's inverted value) – connect one output to the RS232 GND and the other to the RS232 RX. Outputs will then appear as ±5V (Be careful ... this won't work if signal and chassis grounds are connected!).

TTL to RS232 – use the TTL level signal to switch a transistor or opto-coupler, and steal ±12V from other RS232 lines.

RS232 to TTL – use the RS232 level signal to switch a transistor, and provide a 5V supply line.

RS232 to TTL – use a series resistor to reduce the voltage, clamp with a diode, and use an inverting TTL input/buffer.

Bleeding power "There is one way to get some power out of serial port: steal it from signal lines. When you are developing you own circuit which connects only the PC, then the only line which can be used are the output signal lines from PC serial port: DTS, RTS and TD.

In normal operation situation DTR and RTS are raised so they give positive voltage output (about +12V when not loaded). TD pin is in logic 1 when no data is sent which means that it is most time at negative voltage (-12V when not loaded) most of the time. The voltage at these outputs drop quite fast when load current is increase, because they are designed to drive normally only RS-232 input circuits (3-7 kohm resistance). " `http://www.hut.fi/Misc/Electronics/circuits/rspower.html`

Surge protection/Opto-isolation "RS-232C serial port lines are quite prone to be damaged by [high voltages]... Protecting RS-232 serial port against overvoltage is usually quite easy. The standard says that the signal can be voltage between +25 and -25 volts, so anything outside this range is not allowed. Typical RS-232C receiver chips can handle well voltages up to +/-30V. So an overvoltage protection [device] which cuts out everything above 25V and acts fast is the solution. Suitable components for this are varistors (VDR), zener diodes and other fast semiconductor based overvoltage protection devices.

Another question is what needs to be protected. The answer is that every communication line wire which is connected between computers must be protected. In simplest case there are only transmit data, receive data and ground lines used. In this case only protection components are needed: one between transmit data and ground and another between receive data and ground. If more lines are used, then all of then must be protected with a protection component between signal wire and signal ground.

In [potentially] bad overvoltage situations is is recommended to use galvanic isolators in serial port lines or use serial data converters which provide isolation and protection (short haul modems, current loop adapters etc.)."

`http://www.epanorama.net/circuits/rs232_protector.html`

8250/16550 UART registers The UART base address is either fixed, set using jumpers, or by using the BIOS. There are 4 "standard" addresses assigned to the UART in a PC 03f8h, 02f8h, 03e8h, 02e8h however other addresses may also be used. The COM port typically generates an interrupt at IRQ # 3 (COM2/COM4) or 4(COM1/COM3), however to avoid conflicts, some UART's can also be configured to use the "free" interrupts: 10 or 11. The UART registers which will be used in this lab exercise are:

Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Offset
Transmit Register	TX	Write-only, DLAB=0 – data byte for TX								+0x00
Receive Register	RX	Read-only, DLAB=0 – data byte RX'd								+0x00
Line Status Register	LSR		TX Buffer empty						RX Data Ready	+0x05
Line Control Register	LCR	Divisor Latch Ac-cess Bit DLAB			0-even 1-odd	parity? 0-no 1-yes	stop bits? 0-1 1-1.5	00 – 5 bits 01 – 6 bits 10 – 7 bits 11 – 8 bits		+0x03
Divisor Latch Low	DLL	Read/Write, DLAB=1 – Low Byte of 16 bit divisor								+0x00
Divisor Latch High	DLH	Read/Write, DLAB=1 – High Byte of 16 bit divisor								+0x01

The 8250/16550 UART transmit and receive circuits clock/sample 16 times per bit. Transmissio/Reception of the data-word is LSB–first. The crystal frequency typically used with these UART's is is 1.8432 MHz.

$$Baudrate = \frac{clockfrequency}{16 \times DLH{:}DLL}$$

Common divisors

Baud Rate	DLH	DLL
9600	00	0C
4800	00	18
1200	00	60

Helpsheet: Interfacing w/Parallel Port

Overview The parallel port is a byte-serial bus, originally designed to provide an interface between the computer and printer. The standard interface uses a 25 pin D-type connector on the PC side, and a Centronics-type connector on the printer side. In the standard parallel port, data is sent in one direction only, from the computer to the printer. Handshaking involves signalling in both directions. At least three registers are associated with any parallel port: the data, status and control registers each contain bits which correspond to lines on the port connector, as shown in the table below:

Signal Name	Register bit	Signal Pin	Ground Return
Data bit 0	D0	2	19
Data bit 1	D1	3	19
Data bit 2	D2	4	20
Data bit 3	D3	5	20
Data bit 4	D4	6	21
Data bit 5	D5	7	21
Data bit 6	D6	8	22
Data bit 7	D7	9	22
$\overline{ERROR}$ ($\overline{FAULT}$)	S3	15	23
$SELECT$	S4	13	24
$PAPEREND$	S5	12	24
$\overline{ACK}$	S6	10	24
$BUSY$	S7	11	23
$\overline{STROBE}$	C0	1	18
$\overline{AUTOLF}$	C1	14	25
$\overline{INIT}$	C2	16	25
$\overline{SELECTIN}$	C3	17	25

extracted from http://www.lvr.com/files/pppinout.pdf

When communicating with the printer, the parallel port peripheral " ... sends data to the data lines, and then sets the $\overline{STROBE}$ LOW and then HIGH. These then latch the data to the printer ... [i.e.]

1. Data is written to the data register.
2. The program reads from the status register to test whether the BUSY signal is LOW (that is, the printer is not busy)
3. If the printer is not busy, then the program sets the $\overline{STROBE}$ line active LOW.
4. Program then makes the $\overline{STROBE}$ line HIGH by de-asserting it.

" Buchanan and Wilson (2001)

A note about locating signals on D-type connectors. Male and female connectors are designed to mate with each other, so the pin and it's mating socket are assigned the same numbers. This means that if the connectors are facing you, pin 1 is located on the right of the female connector, and on the left of the male connector i.e. the layouts are mirrored. When making up a cable, always verify by checking the numbers marked on the connector.

Power at the parallel port .
See `http://www.hut.fi/Misc/Electronics/circuits/lptpower.html`
and `http://www.hut.fi/Misc/Electronics/circuits/parallel_output.html`.
The parallel port utilises TTL-level signals, and is generally capable of:

- sourcing 0.5mA reliably on the data lines (out),
- sinking 5mA reliably on the data lines (in),
- sourcing 1 mA on the control/signal lines (out),
- sinking 7mA on the control/signal lines (in).

These are only guidelines, actual values will depend on the internal PC circuitry used for the parallel port. Conservative guidelines are used, because, if the port specifications are exceeded the voltage supplied will drop (source), or the port will blow (sink).

If your external circuitry can tolerate the voltage drop associated with higher current draw, then you may exceed the sourcing guidelines (up to 2.6mA @ 2.4V). *Note: If you connect multiple output pins together in an attempt to source additional current, Use protective diodes on each output to prevent accidental shorts (blow port).*

Apart from circuitry sinking too much current on data/signal lines, unexpected spikes may be fed back into the computer along the ground lines. The only way to prevent this is to use current limiting resistors to connect grounds or, opto-isolate the port from the external circuitry.

As a general rule:

try NOT to use a parallel port which is built into the motherboard.
These cannot be replaced if something goes wrong.

In the Lab

ID# ____

Check that the circuit matches the circuit diagram on page 51; if there are any differences, make appropriate corrections to the diagram and/or circuit.

Please let the lab technician check your circuit before you proceed.

1. This question will allow the student to investigate and develop code that converts Analog Data into Digital Data using the A/D peripheral. Configure `ADCON0`, `ADCON1` so that the A/D meets the following specifications:

A/D Conversion Clock	$\frac{F_{osc}}{8}$
Analog Channel	Channel 0 selected
State	A/D Conversion not in progress, A/D On
Result	rightt justified
Reference voltages	V_{dd}, V_{ss}
Port Configuration	AN0 as the ONLY analog input

Bit	<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
ADCON0								
ADCON1								

2 marks

The result produced by the A/D module is 10 bits long and not 8 bits nor 16 bits. This means that the answer is put into two registers named `ADRESH` and `ADRESL`. When the answer is left justified, valid bits are in `ADRESH<7:0>, ADRESL<7:6>`. When the answer is right justified, valid bits are in `ADRESH<1:0>, ADRESL<7:0>`. We will only be using `ADRESH`. The code appears on the following page. Fill in the values from the table above.

Check that the code is consistent with the comments before moving on. If not, what changes did you make?

In the code, Timer0 is used to set the sampling rate. If Timer0 has just overflowed, how long will it take to overflow again? Show your working.

2 marks

Compile the code and ensure that it builds successfully.

Did your code compile? Yes or No (Circle your answer)

If not, what changes did you make?

If you have problems ask the TA for assistance.

```
        LIST    p=16f877
        INCLUDE "p16f877.inc"
        ORG     0x00
        nop
        goto    Start           ; Start at the reset vector + 1 for ICD

        ORG     0x20
Start
        BANKSEL PORTD
        clrf    PORTD           ;Clear PORTD

        movlw   ___________     ;Fosc/8, A/D enabled, Sample Channel 0
        movwf   ADCON0

        BANKSEL OPTION_REG
        movlw   B'10000111'     ;TMR0 prescaler, 1:256
        movwf   OPTION_REG
        clrf    TRISD           ;PORTD all outputs
        movlw   ___________     ;Right justify, 1 analog channel
        movwf   ADCON1          ;VDD and VSS references

Main    bcf     INTCON,T0IF
Loop    btfss   INTCON,T0IF     ;Wait for Timer0 to timeout
        goto    Loop

        bsf     ADCON0,GO       ;Start A/D conversion for channel 0

Wait    btfss   PIR1,ADIF       ;Wait for conversion to complete
        goto    Wait

        movf    ADRESH,W        ;Write A/D result to PORTD
        movwf   PORTD           ;LEDs
        bcf     PIR1,ADIF       ;Clear the completion flag

        goto    Main            ;Do it again
        sleep

        END                     ;this is the last line in the file
```

Does the program work as you expected? Yes or No (Circle your answer)

If it didn't what was the problem? How did you fix it?

Program the 16F877, run the program and use a voltmeter to read the actual input voltage to the PIC from the potentiometer. Note the voltage and the binary value displayed on the LED's in the chart below. Take 5 readings between 0 and 5 Volts, then work out the equivalent voltages for the readings you observed on the LED's.

Input Voltage	PIC LED's	Equivalent Voltage

Comment on the accuracy and resolution of your results. *2 marks*

Would it improve if we used more bits of the A/D result? How could we modify the code to achieve this? Justify your answers. *2 marks*

2. We shall now investigate Pulse Width Modulation (PWM). Compile the code below and PROGRAM the PIC16F877. Then vary the POT.

   If the circuit is working properly the Motor speed should change as you turn the POT.

   If you have problems ask the TA for assistance.

   Answer the following questions based on the code on the following page, and your observations.

   (a) Fill in appropriate comments at the places indicated. *4 marks*

   (b) What is the PWM frequency? Show your working. *4 marks*

   (c) Explain, in your own words, why the motor speed changes as you vary the POT. *2 marks*

   (d) What happens if you change the PWM frequency specified by the Timer2 pre-scaler and re-compile and re-run the program? Your answer should state how you changed the Timer2 pre-scaler for this investigation. *4 marks*

   (e) Based on your observations, is PWM an appropriate technique for D/A conversion? Explain your answer. *2 marks*

```
        LIST    p=16F877
        INCLUDE <p16F877.inc>
        ORG     0x00
        goto    main

        ORG     0x20
main
        BANKSEL PR2             ; __________________________________
        movlw   0xFF            ; __________________________________
        movwf   PR2             ; __________________________________

        BANKSEL T2CON           ; __________________________________
        movlw   B'00000110'     ; __________________________________
        movwf   T2CON           ; __________________________________

        movlw   B'00001100'     ; __________________________________
        movwf   CCP1CON         ; __________________________________

        BANKSEL TRISD           ;
        clrf    TRISD           ;Configure PORTD -- all output pins.
        bcf     TRISC,2         ;Configure PORTC<2> as an output pin.

        BANKSEL ADCON0
        movlw   B'01000001'     ;Fosc/8, A/D enabled, Sample Channel 0
        movwf   ADCON0
        BANKSEL OPTION_REG
        movlw   B'10000111'     ;TMR0 prescaler, 1:256
        movwf   OPTION_REG
        movlw   B'00001110'     ;Left justify, 1 analog channel
        movwf   ADCON1          ;VDD and VSS references

        BANKSEL PORTD
        movlw   0xFF            ;put an indicator value on the LED's
        movwf   PORTD


Loop    bcf     INTCON,T0IF
Here    btfss   INTCON,T0IF     ;Wait for Timer0 to timeout
        goto    Here

        bcf     PIR1,ADIF       ;Clear the completion flag
        bsf     ADCON0,GO       ;Start A/D conversion for channel 0

Wait    btfss   PIR1,ADIF       ;Wait for conversion to complete
        goto    Wait

        movf    ADRESH,W        ;_______________________________________
        movwf   CCPR1L
        movwf   PORTD
        goto    Loop            ;Do it again
        sleep
        END
```

3. We will move on to investigate the parallel slave port (PSP).

   Examine the following program and answer the questions.

```
            INCLUDE <P16f877.inc>
            LIST P=16F877

            ORG   0x00
            nop
            goto  main

            ORG     0x20
main        bsf     STATUS,RP0     ; Line A
            movlw   B'00000111'
            movwf   ADCON1
            bsf     TRISE,PSPMODE  ; Line B
            bcf     STATUS,RP0     ; Line C
            movlw   0x56           ; Line D
            movwf   PORTD          ; Line E
loop        goto loop
            sleep
            END
```

   - What does Line B do? ______________________________ *1 mark*
   - What does Line E do? ______________________________ *1 mark*
   - What is the value in TRISD when this program runs? __________ *1 mark*
   - Can you tell (just from the code) whether the value in the PORTD output latch will appear on the output pins? Explain your answer. *2 marks*

   Type in the above program. Compile it and download it to the PIC16F877. Run the program, and record what happens when the switches are ALL on. *1 mark*

   If you have problems ask the TA for assistance.

   In your own words, explain how the PSP can be used by an external microprocessor, to access data within the PIC16F877. *2 marks*

Turn off all switches before proceeding.

4. This question will familiarize us with the parallel port on the PC[47].

   (a) What kind of PC are you running this lab on? ____________________ *1 mark*

   (b) What operating system is loaded on the machine? ____________________ *1 mark*

   (c) How do you tell the difference between a parallel port and a serial port? *1 mark*

   (d) How many **parallel** ports are there on the machine? ____________________ *mark*

   (e) To be safe we should use a parallel port that is NOT on the motherboard; this way we would avoid accidentally blowing the computer. If this is not possible then our circuit should have appropriate protection circuitry.

      - Choose a parallel port on your computer, and write down the designator (LPTx), the associated interrupt number, the port address(es) and describe where the port connector is located. *3 marks*

      - For your circuit, identify the protection circuitry which will prevent excess current being fed into the parallel port. *3 marks*

[47]To get information about the PC, Right click on the "My Computer" icon, and select Properties from the menu. Some information can be found on the General Tab. Other information can be found on the Hardware Tab (you will need to press the device manager button, and find the parallel port under the "Ports" tree-item)

Connect the Parallel Port Cable before proceeding.

5. We are going to use the Data Acquisition Toolbox in MATLAB to control the pins of the parallel port.

   The registers are contained at the addresses you wrote down for your chosen parallel port. The lowest number (e.g. hex 0378) is referred to as the *BASE_ADDRESS* of the parallel port. All parallel registers have an address which is offset from the BASE_ADDRESS.

   The *data register* is located at BASE_ADDRESS+0x00, and contains bits which correspond to the values output on pins 2 thru 9.

   The *status register* is located at BASE_ADDRESS+0x01, and contains bits which correspond to the values input from the Error,Select, Out of Paper, Acknowledge and Busy lines (see helpsheet to determine which is which).

   The *control register* is located at BASE_ADDRESS+0x02. It contains the bits corresponding to the values output on the Strobe, LineFeed, Initialise, and Select lines (see helpsheet to determine which is which).

   (a) Start MATLAB on your computer. Use the `daqhwinfo` command to access information about ports on your machine, which MATLAB can recognise.

   ```
   parportinfo=daqhwinfo('parallel');
   parportinfo.InstalledBoardIds
   ```

   Ensure that your chosen port is available to MATLAB.

   (b) Create a MATLAB digital Input/output object for your chosen port (be sure to replace `x` with the appropriate number):

   ```
   parport = digitalio('parallel','LPTx');
   ```

   (c) Examine the hardware information for the object you have just created.

   ```
   hwinfo = daqhwinfo(parport);
   hwinfo.Port(1)
   hwinfo.Port(2)
   hwinfo.Port(3)
   ```

   Relate the information to that given in the helpsheet. Use the space below to make notes.

   If you have problems ask the TA for assistance.

(d) Add the following lines to your digital input/output object (corresponding to the parallel port lines you use in your circuit):

```
addline(parport,0:7,0,'out');
addline(parport,1:3,2,'out');
parport
```

We can set and clear individual bits using the command **putvalue**. For example

```
putvalue(parport.pin2,1);
```

Use the LED's or a meter to determine whether each of the 11 lines is active high, or active low. If you have problems ask the TA for assistance.

We can also set multiple lines simultaneously for example:

```
putvalue(parport,75);
```

What value(s) would you have to use, and in what sequence, in order to get the value 23_{10} into the Parallel Slave Port of the PIC16F877? *3 marks*

Run the PIC program, use the appropriate commands in MATLAB and then check (use watch window) to see if you were successful. If you have problems ask the TA for assistance.

(e) Write an assembly language program which varies motor speed according to value received from PSP.

Compile and test your code using MATLAB to change the parallel port output from the PC. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *10 marks*

6. The PIC16F877 can also control a parallel bus connection. We are going to use the LCD functions we wrote in the pre-lab to write a simple program which will display a string, and a value.

   (a) Let us start with a basic program which will use the include file, and simply initialise the LCD. Make a new project, a new file, and enter the following code (N.B. the LCD include file must be in the same directory).

```
        INCLUDE <P16f877.inc>
        LIST P=16F877

        ORG     0x00
        nop
        goto    main

        ORG     0x10
main    bsf     STATUS,RP0
        movlw   B'00000110' ; we need this for PORTA
        movwf   ADCON1      ; to operate properly as digital pins
        bcf     STATUS,RP0
        call    LCDSetup

loop    goto loop
        sleep

        ORG     0x200       ; place this somewhere where it won't clash with the program

LCD_DATA        EQU PORTB
LCD_DATA_TRIS   EQU TRISB
LCD_CNTL        EQU PORTA
LCD_CNTL_TRIS   EQU TRISA
LCD_E           EQU 3
LCD_RW          EQU 2
LCD_RS          EQU 1
LCD_Vars        EQU 0x20     ; we need 4 locations starting from LCD_Vars in Block 0

        INCLUDE <LCDASM.INC>

        END
```

   Compile and run all code, and make sure it works in the simulator.

   Program and run the code on the cirucit using the ICD. If the program is working your LCD will initialise and display `Ok`. If it doesn't work, apply your troubleshooting skills to locating the problem. Use the space below to record your troubleshooting investigations.

   Demonstrate working code to your TA/lecturer who will sign your script. *4 marks*

(b) Choose an appropriate place in your code to place the following lookup tables.

```
lookup_nibble   addwf   PCL,F
                DT  "0123456789ABCDEF\0"
lookup_string   addwf   PCL,F
                DT  "The value is: \0"
```

Where did you put them? Explain why you chose that position. *2 marks*

Now, alter the main loop so that it uses a register named **indx** which starts from 0 and increments until we get to the end of the string. On each iteration, fetch the character from `lookup_string` associated with **indx**, and display it on the LCD using the `LCD_SEND_CHAR` routine.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *5 marks*

(c) Strings can also be stored in RAM. Write a routine that will take the following 16-byte string:

```
MyStrInit   addwf   PCL,F
            DT      "The value is:  ",0
```

and copy it using indirect addressing to locations starting from `MyStr`. You should define the 16 spaces for `MyStr` using the `CBLOCK` directive.

Alter the main loop so that it will convert the number stored in a register `num` into the ASCII equivalent of the hexadecimal number, and store it in `MyStr` at positions 13 and 14, before displaying `MyStr` on the LCD display.

(Hint: Use the `lookup_nibble` table to translate each nibble of `num` separately.)

Initialise `num` with a value.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *10 marks*

(d) Now alter the main loop so that:

- the motor runs at a speed determined by the value written to the PSP from the PC,
- Timer0 is configured as a counter, whose value is continuously copied to `num` for display.

As the motor turns we should see the LCD displaying the number of encoder pulses counted by Timer0.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *16 marks*

7. We would like to write a program for the PIC16F877, which communicates with the PC over an RS232 link, instead of across the PSP.

   Disconnect the parallel cable and connect the serial cable.

   Firstly, let us locate all the serial ports on the computer that you are using. Look at the back of the computer. How many serial ports do you see? ______________
   (N.B. remember that the computer is DTE, so you are looking for **male** D-type connectors, either 9 pin or 25 pin.)

   The computer "labels" it's serial ports COM1, COM2, COM3, COM4 etc Which serial port are you connected to? ______________

   The circuit you have been provided with has a PIC in a header, and an RS232 driver IC. What is the IC part #? ______________

   You have already located the registers of the USART peripheral in the PIC (as part of your pre-lab exercises). We will start with a simple program which polls for a character, and then echoes the character received to the transmit register. We will use the following communication settings:

   - 2400 baud
   - 8 data bits
   - no parity bits
   - no flow control
   - 1 stop bit

   The structure of the program will be:

```
configure and control
loop:
    wait for state
    read data
    write data
    wait for state
goto loop
```

Fill in appropriate settings, and register/flag names in the code below. *7 marks*

```
            LIST P=16F877
            INCLUDE <P16F877.inc>

            ORG     0x00
            nop
            goto    main

            ORG     0x10
main        call    UART_Cnfg
            goto    UART_poll

UART_Cnfg   bsf     STATUS,RP0 ; Go to Bank1
            movlw   __________ ; Set Baud rate to 2400
            movwf   SPBRG
            movlw   __________ ; 8-bit transmit, transmitter enabled,
            movwf   TXSTA      ; asynchronous mode, low speed mode
            bcf     STATUS,RP0 ; Go to Bank 0
            movlw   __________ ; 8-bit receive, receiver enabled,
            movwf   RCSTA      ; serial port enabled
            return

UART_poll   clrw
loop        btfss   ______,RCIF; wait on char
            goto    loop
nchar       movf    _________,W; transmit what you rx (NB reading clears the flag)
            movwf   ___________
wait        btfss   ______,TXIF; flag clears when character is gone from buffer
            goto    wait
            goto    loop
            sleep

            END
```

Start up MPLAB, and create a new project and file for the code. Compile your code, and make sure there are no errors. Power up your circuit, and download the code.

Start Hyperterminal (as per helpsheet) and choose the required communication settings. Connect to the communication channel, and then run the program on the PIC16F877.

What happens when you type in the Hyperterminal window? Is it what you expected? If not, why not, and/or what did you do to fix it?

What happens if you disconnect, change the communication settings, and then reconnect and start typing? Is that also what you expected? If not, why not, and/or what did you do to fix it?

Explain your observations. *4 marks*

Change the settings in HyperTerminal back to their original values before proceeding.

8. We can modify the previous example, so that instead of just repeating what it received, the microcontroller changes the content somehow.

   Add the routine `lookup_string` (that you used earlier) between `return` and `UART_poll`.

   Change the original program by inserting a `call` to your routine, on a new line just after the `nchar` labelled line.

   Compile your code, and make sure there are no errors. Download the code and test your program by typing different characters in HyperTerminal. Does it work as you expected? Make changes until it does.

   If your program didn't work on the first try what were the problems? How did you fix it?

   Write down the final version of your routine here. *5 marks*

   Did this code work? Yes or No (Circle your answer)

   If so, demonstrate the working code to your TA/lecturer who will sign your script.

9. So far we have been polling. Let us convert the polling program to an interrupt driven one. We can rewrite the main part of the code as:

   configure and **control**

   loop:

   – do something else

   goto loop

   and then write an interrupt service routine, and handlers for the individual interrupts.

   Let us start with the interrupt subroutine. Choose appropriate locations to store `_W`, `_STATUS` and `_PCLATH`, so that the code will work in all banks. Justify your choices. *3 marks*

```
_W          EQU     _______
_STATUS     EQU     _______
_PCLATH     EQU     _______

isr                             ; save all registers
            movwf   _W          ; Copy W to TEMP register
            swapf   STATUS,W    ; Swap status to be saved into W
            clrf    STATUS      ; bank 0, clears IRP,RP1,RP0
            movwf   _STATUS     ; Save status to bank zero STATUS_TEMP register
            movf    PCLATH, W   ; Only required if using pages 1, 2 and/or 3
            movwf   _PCLATH     ; Save PCLATH into W
            clrf    PCLATH      ; Page zero, regardless of current page

           ; no need to clear flags as they change automatically for this device
            btfsc   PIR1, RCIF
            goto    rx_handler

rtn_isr     movf    _PCLATH, W ; Restore PCLATH
            movwf   PCLATH      ; Move W into PCLATH
            swapf   _STATUS,W   ; Swap STATUS_TEMP register into W
                                ; (sets bank to original state)
            movwf   STATUS      ; Move W into STATUS register
            swapf   _W,F        ; Swap W_TEMP
            swapf   _W,W        ; Swap W_TEMP into W
                                ; all registers restored
            retfie
```

Next we have the individual handlers. We would like to transmit as soon as we receive, so the transmit handler can be empty for now.

```
rx_handler ; on rx, tx the next character
            movf    ________,W ; read the byte to clear the flag
            movwf   ___________
            goto    rtn_isr
```

Now for the associated main loop, fill in the blanks in the following code . *2 marks*

```
            LIST P=16F877
            INCLUDE <P16F877.inc>

            ORG     0x00
            nop
            goto    main

            ORG     0x04
            goto    isr

            ORG     0x10
main        call    UART_Cnfg
            goto    otherloop

UART_Cnfg   bsf     STATUS,RP0 ; Go to Bank1
            movlw   __________ ; Set Baud rate to 2400
            movwf   SPBRG
            movlw   __________ ; 8-bit transmit, transmitter enabled,
            movwf   TXSTA      ; asynchronous mode, low speed mode
            bsf     PIE1, RCIE ; enable receive interrupts
            bcf     STATUS,RP0 ; Go to Bank 0
            movlw   __________ ; 8-bit receive, receiver enabled,
            movwf   RCSTA      ; serial port enabled
            bsf     INTCON, ___; enable peripheral & general interrupts
            bsf     ______, PEIE
            bsf     RCSTA , CREN
            return

otherloop   goto    otherloop
            sleep

            END
```

Type in all your code, compile and run it. Test your program by using HyperTerminal.

Does it work as you expected? If not, why not, and/or what did you do to fix it?

Demonstrate the working code to your TA/lecturer who will sign your script. *4 marks*

10. Previously, we adjusted motor speed from the PC using the PSP. Write a program which will change motor speed based on a single digit typed in HyperTerminal, where `'0'`means stop, `9` means full speed, and digits `1` through `8` adjust speed proportionally between no and full speed.

Compile and test your code. Does it work as you expected? Make changes until it does.

If your program didn't work on the first try what were the problems? How did you fix it?

Write down the final version of your main loop here.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign your script. *10 marks*

11. Modify your program so that it in response to a single character, it sends the 3-character string '`OK\n\0`'. You should use TX interrupts, and indirect addressing in a way which will allow you to send any null-terminated string. *15 marks*

    Compile and test your code. Does it work as you expected? Make changes until it does.

    If your program didn't work on the first try what were the problems? How did you fix it?

    Write down the final version of your main loop here.

    Did this code work? Yes or No (Circle your answer)

    If so, demonstrate the working code to your TA/lecturer who will sign your script.

12. Modfy your program to report the measured motor speed(Timer0) across the serial link when the character `S` is typed in HyperTerminal. *20 marks*

    Compile and test your code. Does it work as you expected? Make changes until it does.

    If your program didn't work on the first try what were the problems? How did you fix it?

    Write down the final version of your main loop here.

    Did this code work? Yes or No (Circle your answer)

    If so, demonstrate the working code to your TA/lecturer who will sign your script.

13. Challenge: The pre-lab asked you to explain an algorithm for manipulating regularly sampled data.

    (a) Use either instruction-based delays or Timer1 (with/without interrupts) to regularly sample the distance turned by the motor (Timer0 counts). *20 marks*

    (b) All results should be reported serially using RS232. *10 marks*

    (c) Bonus(30 marks): For the sampled data, implement the algorithm you explained in the pre-lab.

    Compile and test your code. Does it work as you expected? Make changes until it does.

    If your program didn't work on the first try what were the problems? How did you fix it?

Attach a printout of your final code.

Did this code work? Yes or No (Circle your answer)

If so, demonstrate the working code to your TA/lecturer who will sign/stamp your script/printout.

Total marks 300.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this lab exercise.
  - perform I/O programming on the Personal Computer (PC)
  - produce programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and/or PPC datasheets/details, and respective language/instruction sets.
  - interface external peripherals/devices to the PIC16F877 microcontroller using a parallel bus, and interface the PIC16F877 microcontroller as an external device on a parallel bus (memory mapped).
- Which aspect of this lab exercise did you have the most difficulty understanding?
- Which aspect of this lab exercise did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this lab exercise.
- Identify one way in which this lab exercise could be improved.

Communication

In this section we have looked at commercially available methods of interfacing peripherals with microprocessors. In doing so, we have completed the third learning objective "design and implement an appropriate interface between a microprocessor-based system and another microprocessor-based system, given relevant data-sheets and suitable parts".

The course is now complete.

Final Feedback

This form should be filled in together with the standard course evaluation form.
Please re-read all the course/unit objectives, and respond to the following questions.
At the end of this unit/course the student will be able to:

√	Objective	Description
	A	in general, or (given relevant diagrams and specifications) for a particular system, identify, and describe the role of, the components of a microprocessor, a microprocessor-based system, a development suite (hardware, software) for a microprocessor-based system.
	B	illustrate how data can be represented in (and accessed from) memory, implement arithmetic and data manipulation operations in assembly language, and assess how well code will perform in a given context, given the architecture and instruction set of a microprocessor
	C	design and implement an appropriate interface between a microprocessor-based system and a peripheral device (or another microprocessor-based system), given relevant datasheets and suitable parts
	D	select (and critique the selection of) a microprocessor-based system for an application, given relevant datasheets and application requirements
	1	identify the components of a typical microprocessor (ALU, registers, stack), describe the operation of the components of a typical microprocessor, describe the representation of instructions in the operation of the instruction cycle, and illustrate how these may be implemented in digital logic.
	2	identify the components of a microprocessor-based system (CPU, bus, memory/layout, peripherals), describe the operation of typical components of a microprocessor-based system, and discuss issues involved in the design/choice of the various components of the microprocessor(-based system)
	3	describe (and/or illustrate) how bus addressing, conflict resolution and memory systems may be implemented using digital logic, and explain the function of basic support circuitry for microprocessors such as clocks, reset circuitry, watch dog timer, capacitors to ground etc. .
	4	differentiate between organization and architecture, and relate instruction set design to the programmers model of the microprocessor,.
	5	explain the operation of typical machine instructions, addressing modes, status bits, and infer the consequences of sequences of generic instructions.
	6	explain the roles of the compiler, linker, assembler, simulator, emulator, debugger in the software development tool chain, and the role of scripting languages, make/configure-like utilities and IDEs in the software development process.
	7	explain the role of firmware, operating systems, oscilloscopes, logic probes/analyzers, terminals, disk drives, external memory, debug protocols e.g. JTAG and host computers in the development and support of microprocessor based systems.

√	Objective	Description
	8	recognize and differentiate between the different commercially available families/forms of microprocessor based systems (Motorola, AVR, PIC chip/component packages – SOC, microcontroller, PC, backplane bus, PC104, standalone, embedded) based on their pictures/properties/descriptions and supported development tools.
	9	explain the memory layout, operation of the instruction cycle, and the basic instructions (move, add, subtract, shifts) for the PIC16F877 microcontroller.
	10	utilize the MPLAB IDE and CCS compiler, to develop software, for the MicroChip PIC16Cxxx series of microcontroller, in C, C++ and assembly language.
	11	represent real numbers using fixed and floating point binary representations.
	12	perform integer arithmetic operations using binary representations of the operands.
	13	perform fixed/floating point arithmetic operations using binary representations of the operands.
	14	describe how data (and data structures) can be represented and manipulated, with reference to alignment and byte/bit ordering issues.
	15	implement basic programming operations (loops, swaps, lookups), integer arithmetic, and other computation/numerical methods, in assembly language on the PIC16Cxxx series of microcontrollers.
	16	contrast alternate programming techniques in terms of size/speed of the code produced for the PIC16Cxxx series of microcontrollers.
	17	give examples of the limitations placed on compiler-generated code, for the PIC16Cxxx series of microcontrollers, by the need for generality.
	18	identify alternate ways in which components may be interfaced with a microprocessor (i.e. common idioms for interacting with hardware): single bit (with debounce), matrix, map into memory space, etc. .
	19	explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.
	20	explain the operation of commonly used peripherals: PPI/PIA, A/D, Timers/Counters, PWM, D/A.
	21	select and apply techniques for troubleshooting digital circuitry using standard laboratory equipment.
	22	utilize built-in peripherals of the PIC16F877 microcontroller in simple applications.
	23	understand the implications of (nested/multiple) interrupts for execution times, and the need for context saving.
	24	interpret and recognize the implications of, microprocessor timing and interfacing requirements.
	25	determine the noise, voltage, current and power considerations due to microprocessors and peripheral fan-in/out and how different logic families (as well as inverted voltage logic) may be used in conjunction in a design.

$\surd$	Objective	Description
	26	compare alternate microprocessors in terms of the instruction set, and features(such as cache pipelining etc.) and the best performance, which may be facilitated by the instruction set and features.
	27	recognize the issues and tradeoffs involved in the design of a microprocessor based (embedded) system, and justify the role of a microprocessor(-based system) in an applied context.
	28	remember and follow a checklist of design guidelines, in order to perform a simple design for a microprocessor based system.
	29	understand commonly used parallel and serial communication protocols (I2C, CAN, RS232, USB, Firewire) and interpret descriptions of proprietary protocols (Dallas 1-wire).
	30	differentiate between bit-banging and dedicated hardware interfaces for serial/parallel communication (PPI/PIA vs ACIA/UART, PPC)
	31	perform I/O programming on the PC in C/Assembly using the Visual C++ compiler
	32	produce assembly language programs for communication between a microprocessor and a PC using the parallel and/or serial port, given UART and PPC datasheets/details, and respective instruction sets.
	33	interface external peripherals/devices to the PIC16F877 microcontroller using a parallel bus, and interface the PIC16F877 microcontroller as an external device on a parallel bus (memory mapped).

- Do you think that you have achieved all **four** of the course objectives? (Yes/No)
- Tick the objectives you feel that you have achieved, and identify the objectives you have not achieved, by writing NA next to them.
- Identify the **three** learning objectives which you had the most difficulty in achieving, by writing D next to them.
- Identify the **three** learning objectives which you liked best, by writing B next to them.

- Do you have any comments/suggestions regarding the **order** in which material was presented in this course?

- Do you have any comments/suggestions regarding the **depth** with which material was covered in this course?

- Do you have any comments/suggestions regarding the **content** covered in this course?

- Do you feel that you were adequately prepared to take this course, and if not, why not?

- Do you feel that this course has adequately prepared you to take final year courses? Please give a **brief** explanation of your answer.

Part IX
Appendices

A Data Sheets

Selected pages from documents listed below are included in the data-sheets used in this course. ** documents do not appear in the electronic copy

- *A Comparison of Low End 8 Bit Microcontrollers*, M. Palmer, Microchip Application Note 520, 1997. pp 1-9.
- *PIC16F87x 20/40 Pin 8-bit CMOS FLASH Microcontrollers*, Microchip Data Sheet DS30292C, 2001. pp 1, 6, 8, 9, 11-13, 15-17, 26-27, 29-39, 41-49, 51-63, 95-102, 111-117, 129-133, 135-142, 149, 159-163, 165-167, 173, 175.
- Chapter 6 from *Programming and Customizing PICmicro^R Microcontrollers*, M. Predko, 2001.
- *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver)*, Hitachi datasheet, pp. 24,25,33,46,52,58.
- *KS0066U 16COM / 40SEG DRIVER & CONTROLLER FOR DOT MATRIX LCD*, Samsung Electronics datasheet,pp. 1-2,18-19,26-27, 31-32
- *8XC51FX CHMOS SINGLE-CHIP 8-BIT MICROCONTROLLERS*, Intel Preliminary DataSheet, April 1996. pp 1, extract from 2-3, extract from 8-9, 4-5
- ***80C51 External Memory Interfacing*, Phillips Semiconductors Application Note AN457, May 1996. p 2
- *8051 Compact Assembly Reference* from `http://www.stack.nl/~wvengen/uni/elec/8051ref.pdf`, page 1
- ***Converting from 8051 to Microchip Assembler: A Quick Reference*, G. Kavaiya, Microchip Application Note 880, 2003. pp 1-8.
- *ATMEL AT91 Electrical Characteristics M55800A*, pp 1,2,3,11
- *ATMEL AT91 Summary M55800A*, p. 9
- *ARM 7 Datasheet* Document Number: ARM DDI 0020C Issued: Dec 1994 , pp 13-15
- *ARM7TDMI Datasheet* Document Number: ARM DDI 0084D Issued: Chapter 4 Arm Instruction Sept. pp. 2-4
- *Z8 CPU User Manual UM001602-0904*, ZiLOG Sept. 2004, pp.1-3, 7-9, 201-204

- ***MC68HC705C8A Rev. 2.1 Technical Data*, 2001. pp 22,25,35,36,38,44-48,57-58,154,164-170
- *Micro-reader: RI-STU-MRD1 Reference Manual*, Texas Instruments Series 2000 Reader System, May 2000. pp 6,8,15-19.
- *MicroLAN Design Guide*,Tech Brief 1,Dallas Semiconductor. pp1-2
- *Intel Hexadecimal Object File Format Specification*,Intel, Revision A, 1988. pp 1-10.

B Glossary

Where definitions have been taken from other documents, the references are cited.

access time The average time interval between a storage peripheral (usually a disk drive or semiconductor memory) receiving a request to read or write a certain location and returning the value read or completing the write. (FOL 1993)

accumulator The name of the CPU register in a single-address instruction format. The accumulator, or AC, is implicitly one of the two operands for the instruction.(Stallings 2000) Also known as the working register.

addressing modes One of a set of methods for specifying the operand(s) for a machine code instruction. Different processors vary greatly in the number of addressing modes they provide. The more complex modes described below can usually be replaced with a short sequence of instructions using only simpler modes.
The most common modes are ”register” - the operand is stored in a specified register; ”absolute”(direct) - the operand is stored at a specified memory address; and ”immediate” - the operand is contained within the instruction.
Most processors also have indirect addressing modes, e.g. ”register indirect”, ”memory indirect” where the specified register or memory location does not contain the operand but contains its address, known as the ”effective address”. For an absolute (direct) addressing mode, the effective address is contained within the instruction. (FOL 1993)

ALU Arithmetic and Logic Unit. A part of a CPU that performs arithmetic operations, logic operations and associated operations.(Stallings 2000)

analog-to-digital converter

anti-aliasing

arbiter

architecture

arithmetic shift

ASCII American Standard for Information Interchange. ASCII is a 7-bit code used to represent numeric, alphabetic, and special printable characters. It also includes codes for *control characters* which are not printed or displayed but specify some control function.(Stallings 2000)

assembler A language tool that translates a users assembly source code (.asm) into machine code.(MPL 2000)

assembly language A programming language that is once removed from machine language. Machine languages consist entirely of numbers and are almost impossible for humans to read and write. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names (mnemonics) instead of numbers.(MPL 2000)

assert

asynchronous bus timing A technique in which the occurrence of one event on a bus follows and depends upon the occurrence of a previous event.(Stallings 2000)

Automation

background debug

balanced

banking/paging

base/radix In a positional representation of numbers, that integer by which the significance of one digit place must be multiplied to give the significance of the next higher digit place. Conventional decimal numbers are radix ten, binary numbers are radix two.(FOL 1993)

BCD Binary Coded Decimal.

benchmark A standard program or set of programs which can be run on different computers to give an inaccurate measure of their performance.
"In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks."
A benchmark may attempt to indicate the overall power of a system by including a "typical" mixture of programs or it may attempt to measure more specific aspects of performance, like graphics, I/O or computation (integer or floating-point). Others measure specific tasks like rendering polygons, reading and writing files or performing operations on matrices. The most useful kind of benchmark is one which is tailored to a user's own typical tasks. While no one benchmark can fully characterise overall system performance, the results of a variety of realistic benchmarks can give valuable insight into expected real performance.
Benchmarks should be carefully interpreted, you should know exactly which benchmark was run (name, version); exactly what configuration was it run on (CPU, memory, compiler options, single user/multi-user, peripherals, network); how does the benchmark relate to your workload? (FOL 1993)

biased

binary

bit mask A pattern of binary values which is combined with some value using bitwise AND with the result that bits in the value in positions where the mask is zero are also set to zero.(FOL 1993)

bit In the pure binary numeration system, either 0 or 1.

bit-parallel

bit-serial

bit-width

Booth's algorithm

boot To load and initialise the operating system on a computer. (FOL 1993)

boot-loader (from "bootstrap" or "to pull oneself up by one's bootstraps") A short program that was read in from cards or paper tape, or toggled in from the front panel switches, which read in a more complex program to which it gave control.

On early computers the bootstrap loader was always very short (great efforts were expended on making it short in order to minimise the labour and chance of error involved in toggling it in), but was just smart enough to read in a slightly more complex program (usually from a card or paper tape reader), to which it handed control; this program in turn was smart enough to read the application or operating system from a magnetic tape drive or disk drive. Thus, in successive steps, the computer "pulled itself up by its bootstraps" to a useful operating state. Nowadays the bootstrap is usually found in ROM or EPROM, and reads the first stage in from a fixed location on the disk, called the "boot block". When this program gains control, it is powerful enough to load the actual OS and hand control over to it.(FOL 1993)

branch prediction A mechanism used by the processor to predict the outcome of a program branch prior to it's execution.(Stallings 2000)

breakpoint In hardware, an event whose execution will cause a halt. In software, an address where execution of the firmware will halt.(MPL 2000)

buffered

bug An unwanted and unintended property of a program or piece of hardware, especially one that causes it to malfunction. The identification and removal of bugs in a program is called "debugging".(FOL 1993)

built-in hardware support for debugging

bus arbitration The process of determining which competing bus master will be permitted access to the bus.(Stallings 2000)

bus cycle

bus A shared communications path consisting of one or a collection of lines. In some computer systems, CPU, memory, and I/O components are connected by a common bus. Since the lines are shared by all the components, only one component at a time can successfully transmit.(Stallings 2000)

byte (a collective term for) 8 bits.

CCS

cell

CISC Complex Instruction Set Computer. Inspired by a desire to improve performance by reducing the number of instruction fetches, CISC processers feature rich instruction sets, and smaller program sizes.

clock (cycle) (period) The relative execution times of instructions on a computer are usually measured by number of clock cycles rather than seconds. One good reason for this is that clock rates for various models of the computer may increase as technology improves, and it is usually the relative times one is interested in when discussing the instruction set.(FOL 1993)

clock speed The fundamental rate in cycles per second at which a computer performs its most basic operations such as adding two numbers or transfering a value from one register to another.(FOL 1993)

CMOS

Communication protocols A set of formal rules describing how to transmit data, especially across a network. Low level protocols define the electrical and physical standards to be observed, bit- and byte-ordering and the transmission and error detection and correction of the bit stream. High level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages etc.(FOL 1993)

compiler A language tool that translates a users C source code into machine code.(MPL 2000)

contiguous

control characters

control signals

counter mode

Counter/Timer

CPU bound

CPU Central Processing Unit. The portion of a computer that fetches and executes instructions. It consists of an arithmetic and logic unit (ALU), a control unit, and registers. Often simply referred to as a *processor*.(Stallings 2000)

CU Control Unit. That part of the CPU that controls CPU operations, including ALU operations, the movement of data within the CPU, and the exchange of data and control signals across external interfaces (e.g. the system bus).(Stallings 2000)

cycle stealing

data

data-word

debug kernel

debugger

decouple

decrement decrease by 1 unit.

density

Development boards

differential line (signal) A kind of electrical connection using two wires, one of which carries the normal signal (V) and the other carries an inverted version the signal (-V). A differential amplifier at the receiver subtracts the inverted signal from the normal signal to yield a signal proportional to V. This subtraction is intended to cancel out any noise induced in the wires, on the assmption that the same level of noise will have been induced in both wires. Twisted pair wiring is often used to try to ensure that this is the case.(FOL 1993)

digit In the decimal system, 0,1,2,3,4,5,6,7,8,9. Often used to refer to the numbers of other numeric bases e.g. hexadecimal digits, binary digits.

digital-to-analog converter

directive

disassembling

discrete event simulator

DMA Direct Memory Access. A form of I/O in which a special module, called a DMA module, controls the exchange of data between main memory and an I/O module. The CPU sends a request for the transfer of a block of data to the DMA module, and is interrupted only after the entire block has been transferred.(Stallings 2000)

EEPROM Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

emulator Hardware that performs the process of executing software (loaded into emulation memory) as if the firmware resided on the microcontroller device under development.(MPL 2000)

enabled

EPROM

executable

exponent

fan- in/out

fetch

file register Register within the register file.

file storage system

firmware

fixed point representation system. A radix numeration system in which the radix point is implicitly fixed in the series of digit places by some convention upon which agreement has been reached.(Stallings 2000)

flags (Z,C,DC, and others??)

Flash A type of EEPROM where data is written or erased in blocks instead of bytes.

floating point representation system. A numeration system in which a real number is represented by a pair of distinct numerals, the real number being the product of the fixed point part, one of the numerals, and a value obtained by raising the implicit floating point base to a power denoted by the exponent in the floating point representation, indicated by the second numeral.(Stallings 2000)

full-duplex A type of duplex communications channel which carries data in both directions at once. (FOL 1993)

general purpose register

Ground bounce

half-duplex A type of communication channel using a single circuit which can carry data in either direction but not both directions at once.(FOL 1993)

Harvard architecture A computer architecture in which program instructions are stored in different memory from data. Each type of memory is accessed via a separate bus, allowing instructions and data to be fetched in parallel.(FOL 1993)

hexadecimal

hold time

host machine The machine on which the IDE, compiler, assembler and other development tools are run to produce the hex file for the target machine.

I/O bound

I/O module One of the major components of a computer. It is responsible for the control of one or more external devices (peripherals) and for the exchange of data between those devices and main memory and/or CPU registers.(Stallings 2000)

IDE An application that has multiple functions for firmware development. The MPLAB IDE integrates a compiler, an assembler, a project manager, an editor, a debugger, a simulator, and an MPLAB IDE Users Guide assortment of other tools within one Windows application. A user developing an application can write code, compile, debug, and test an application without leaving the MPLAB IDE desktop.(MPL 2000)

in-circuit debug

in-circuit programming PIC16F87X microcontrollers can be serially programmed while in the end application circuit. This is simply done with two lines for clock and data and three other lines for power, ground, and the programming voltage. This allows customers to manufacture boards with unprogrammed devices, and then program the microcontroller just before shipping the product. This also allows the most recent firmware, or a custom firmware to be programmed. (PIC 2001)

in-circuit

increment increase by 1 unit.

indirect (decode)

instruction cycle – 5 stages

instruction format fixed/var

instruction width/length fixed/var

instruction register A register that is used to hold an instruction for interpretation.

interrupt A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.(ATI 2001)

interrupt-driven

IR see instruction register.

isolated I/O A method of addressing I/O modules and external devices. The I/O address space is treated separately from main memory address space. Specific I/O machine instructions must be used. Compare *memory-mapped I/O*.(Stallings 2000)

ISR

JTAG

jump/goto/branch

kernel

level translators/shifters

library

line coding

linker scripts

linker A language tool that combines object files and libraries to create executable code.(MPL 2000)

load-and-store

logic analyser

logic high

logic low

logical shift

bitwise-and

bitwise-or

bitwise-xor

machine cycle The (minimum) time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.(Stallings 2000)

machine instruction

makefile

mantissa For a real number x, the mantissa is defined as the positive fractional part.(WOL) The part of a floating point number which, when multiplied by its radix raised to the power of its exponent, gives its value. The mantissa may include the number's sign or this may be considered to be a separate part.(REF)

map

masked/unmasked; set/reset; set/clear

mechanical connection

memory mapped I/O A method of addressing I/O modules and external devices. A single address space is used for both main memory and I/O addresses, and the same machine instructions are used both for memory read/write and for I/O.(Stallings 2000)

micro-code

microcontroller A highly integrated chip that contains all the components comprising a controller. Typically this includes a CPU, RAM, some form of ROM, I/O ports, and timers. Unlike a general-purpose computer, which also includes all of these components, a microcontroller is designed for a very specific task - to control a particular system. As a result, the parts can be simplified and reduced, which cuts down on production costs.(MPL 2000)

micro-operation an elementary CPU operation, performed during 1 clock pulse.(Stallings 2000)

microprocessor A processor whose elements have been miniaturized into one or a few integrated circuits.(Stallings 2000)

mnemonics Instructions that are translated directly into machine code. Mnemonics are used to perform arithmetic and logical operations on data residing in program or data memory of a microcontroller. They can also move data in and out of registers and memory as well as change the flow of program execution. Also referred to as Opcodes.(MPL 2000)

MPLAB

multiplex (time)

network

nibble (a collective term for) 4 bits.

object files

opcode Operation Code. A code used to represent the operations of a computer.(Stallings 2000)

open drain output

open-collector

open-drain (output)

operand An entity on which an operation is performed.(Stallings 2000)

operating system

organization

oscilloscope

packed BCD

pad

parity

peripheral In a computer system, with respect to a particular processing unit, any equipment that provides the processing unit with outside communication.(Stallings 2000)

PIC Peripheral Interrupt Controller

PIC16F877

pipeline branch prediction

pipeline a processor organization in which the processor consists of a number of stages, allowing multiple instructions to be executed concurrently.(Stallings 2000)

point-to-point

polling

Power conditioning

Power supply supervision

preprocessor A program invoked by various compilers to process code before compilation. For example, the C preprocessor, cpp, handles textual macro substitution, conditional compilation and inclusion of other files. (FOL 1993)

priority

processor

program branch

program counter PC. A special purpose register used to hold the address of the next instruction to be executed.(Stallings 2000) Also known as the instruction address register (IAR).

program The noun "program" describes a single, complete and more-or-less self-contained list of instructions, often stored in a single file.(FOL 1993)

project file

project nodes

PROM

push-pull (output)

PWM

RAM Random Access Memory. Memory in which each addressable location has a unique addressing mechanism. The time to access a given location is independent of the sequence of prior access.(Stallings 2000) Variants: static/dynamic.

register file Collection of registers within the CPU. Also used to refer to RAM within a microcontroller which can be accessed in a register-like fashion.

register(s) High speed memory internal to the CPU. Some registers are user visible; that is available to the programmer via the machine instruction set. Other registers are used only by the CPU, for control purposes.(Stallings 2000)

relocatable code A section of code whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation. Relocation is a process in which absolute addresses are assigned to relocatable sections and all identifier symbol definitions within the relocatable sections are updated to their new addresses.(MPL 2000)

Requirements List

Reuse

RISC Reduced Instruction Set Computer(ing). Generally characterised by a large number of general purpose registers, a limited, simple instruction set and an emphasis on optimizing the instruction pipeline.(Stallings 2000)

ROM Read-Only Memory. Semiconductor memory whose contents cannot be altered, except by destroying the storage unit. Non-erasable memory.(Stallings 2000) Variants: EEPROM, Flash.

rotate

Schmitt trigger

setup time

shield

sign extension/reduction In order to represent a two's complement number using more/less bits, it is necessary to consider the sign bit. For example the number negative 8 may be expressed as 1000 in 4 bits, 1111 1000 in 8 bits or 1111 1111 1111 1000 in 16 bits. The uppermost bit is extended to fill all empty bits. Similarly to reduce the number of bits, all bits may be removed up until there is a change. For example 0000 0111 may be reduced to 0111, and no further.

signal ground

significand

simplex

simulator A software program that models the operation of a microprocessor.(MPL 2000)

single-ended

single-phrase function statement

special function register

stack – different uses

static branch prediction

static RAM

step The Single Step command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution.(MPL 2000)

stimulus Data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register. (MPL 2000)

subroutine/call

synchronous bus timing A technique in which the occurrence of events on a bus is determined by a clock. The clock defines equal-width time slots, and events begin only at the beginning of a time slot.(Stallings 2000)

synchronous

target (machine) The machine on which the hex file (produced by a compiler, assembler or other development tools) is intended to run.

ten's complement

terminal

time multiplexed

timer mode

timing diagrams

tool chain

trace

transducer A device for converting sound, temperature, pressure, light or other signals to or from an electronic signal.(FOL 1993)

transition

tri-state (output)

two's complement Used to represent binary integers. A positive integer is represented as the unsigned binary number. A negative number is represented by the two's complement of the positive number. The two's complement may be found by adding one to the complement(inverted bits) of the positive number.(Stallings 2000)

unbalanced

unbuffered

uni/bi-directional

universal or chip programmer

vectored

Verification

visible to the programmer

volatile

von Neumann architecture A computer architecture conceived by mathematician John von Neumann, which forms the core of nearly every computer system in use today (regardless of size). In contrast to a Turing machine, a von Neumann machine has a random-access memory (RAM) which means that each successive operation can read or write any memory location, independent of the location accessed by the previous operation.
A von Neumann machine also has a central processing unit (CPU) with one or more registers that hold data that are being operated on. The CPU has a set of built-in operations (its instruction set) that is far richer than with the Turing machine, e.g. adding two binary integers, or branching to another part of a program if the binary integer in some register is equal to zero (conditional branch).
The CPU can interpret the contents of memory either as instructions or as data according to the fetch-execute cycle.(FOL 1993)

Watch Watch Variables are variables that you may monitor during a debugging session in a watch window. Watch windows contain a list of watch variables that are updated at each break point.(MPL 2000)

Watchdog timers

working register see accumulator.

write

C Study tips

The solutions should be consulted only AFTER you have attempted the review exercises.

If you read the solutions without attempting the questions, you will have LOST the opportunity to assess/reflect on your performance, and make an impact on your memory (i.e. you'll forget faster).

One technique for making best use of solutions is to perform an error analysis. For each question you get wrong, ask yourself the following questions: (Hartman 2001)

1. What answer did I have? AND What was the answer really?
   OR
   What did I do wrong? AND What should I have done?
2. Why did I choose the wrong answer?
   OR
   Why did I do it wrong?
3. How will I remember what I now know is the correct answer?
   OR
   How will I make sure I don't make the same mistake again?

Try to focus on the specific content involved in the errors, rather than on general causes like not studying enough, or feting the night before.

Another study technique from (Hartman 2001):
When reviewing your notes, or reading any reference books, ask yourself:

1. • What do I already know about this topic?
   • Is there anything I don't understand?
   • Did I understand and remember everything?
2. • What am I expected to learn from this reading?
   • Can I figure it out on my own?
   • Which ideas are most important?
3. • How much time should it take me to read this?
   • How can I remember what I have read so far?
   • How can I read with better understanding next time?

If you are working/studying in pairs or groups, you may wish to try the Think Aloud technique while attempting the questions/projects. This technique involves one person (thinker) saying all their thoughts, feelings etc. out loud, as they solve the problem (no swear words please!). The listener(s) then pay(s) attention to what the thinker says, examines the accuracy, points out errors, and keep(s) the thinker talking aloud.

Any people/groups who are interested can pass by my office or access
`http://www.ccny.cuny.edu/ctl/handbook/hartman.html`

to get a copy of the guidelines for using the Think Aloud technique. This technique may help you to discover, "errors, misconceptions, disorganization, and other impediments to intellectual performance"(Hartman 2001).

The Think Aloud technique can also be useful if you are studying alone, but don't be surprised if passers-by give you strange looks!

Happy studying!