original author: Feisal Mohammed
updated: 4th March 2002– CLR

C Solutions & study tips

The solutions presented here should be consulted only AFTER you have attempted the 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: [Har01]

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 [Har01]:
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 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"[Har01].

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!

References

[Har01] Hope J. Hartman. *Developing Students' Meta-cognitive Knowledge and Skills*, chapter 3, pages 33–68. Kluwer Academic Publishers, 2001.

[PIC01] PIC16F87X Data Sheet: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. Technical Reference Document 30292c, MicroChip Technology Inc., 2001.

[Pre01] Myke Preko. *PICmicro Microcontroller Pocket Reference*. McGraw-Hill, 2001.

[Sta00] William Stallings. *Computer Architecture and Organization*. Prentice-Hall, Inc., 5th edition, 2000.

page I 5

1. Differentiate between:

   (a) Von Neumann and Harvard Architectures.
   The primary difference is that the Harvard architecture uses separate memory banks (and buses) for program instructions and program data.

   (b) bus arbitration and multiplexing
   Arbitration is the process by which multiple bus masters negotiate who has permission to use the bus. Multiplexing is the use of the same physical bus lines (by the bus master) for different signals (e.g. address and data) at different points in the bus cycle.

   (c) instruction cycle and machine cycle
   The instruction cycle, is the time it takes the processor to complete all 5 stages of instruction processing i.e. fetch, decode, execute, write and interrupt. The machine cycle is the time it takes the processor to complete an indivisible sequence of micro-operations e.g. place values on ALU inputs, trigger ALU operation ,copy ALU outputs. Machine cycles will be 1 or more clock cycles depending on the processor.

   (d) machine instruction and a micro-operation
   A machine instruction is the smallest action that the programmer can request(e.g add, move). The duration of the instruction execution will depend on the processor and the instruction. A micro-operation is a minimal CU operation (turn on/off a control signal), performed during a single clock pulse.

   (e) RISC and CISC processor architectures
   RISC processor architectures are characterised by smaller, faster instruction sets. CISC processor architectures offer larger, more flexible (for the programmer) instruction sets.

   (f) microprocessor architecture and organization
   Microprocessor architecture refers to those aspects of the design which will affect the way programs are written, or the results when they are executed. (e.g. will the processor have a stack?) Microprocessor organization refers to design/implementation decisions which do not affect the programmer.(e.g. will the ALU have 1 or 2 adders?)

   (g) an accumulator and a register
   Registers are memory locations which are internal to the CPU. The accumulator is a special function register used in single-operand-instruction type processors to hold the implicit second operand. e.g. add 5 presumes that 5 is added to the accumulator value.

   (h) ALU and CU
   Both are components of the CPU. The ALU performs the arithmetic and logic functions. The CU is responsible for coordinating the actions of the CPU i.e. micro-operations.

2. (a) Design a 4 bit decoder using standard logic gates.
Similar diagram of a 3 bit decoder is shown in Stallings page 688.

   (b) Draw a memory array using 16 1-bit memory cells, 4 addressing lines, and standard logic gates.
Presuming we want a single bit output for each address, some possible configurations are:
1)Tie each output of a 4 bit decoder to the Select Line of a memory cell, and tie all the outputs of the memory cells together. The address lines are the 4 input lines for the decoder. The data comes from the shared output.
OR
2) Place the memory cells in 4 rows of 4; for each row, tie the select lines of all the memory cells to one of the outputs of a 2 bit decoder; for each column, tie the outputs of all the memory cells to one of the inputs of 4 input multiplexor. Use two address lines as input to the decoder, and the other two to control which column is output from the multiplexor. The data comes from the multiplexor output.
If you used logic gates to implement the decoders/multiplexor, you will notice that the number of gates required for a two bit decoder, and a two bit multiplexor, is much less than that required for a four bit decoder.

   (c) How many memory cells can be addressed using 8 bit addresses? $2^8 = 256$
Explain the terms kilo-byte, mega-byte, giga-byte.
kilobyte=$2^{10} = 1,024$bytes; megabyte =$2^{20} = 1,048,576$ bytes; giga-byte=2^{30} bytes. Note that the prefixes are not the same as the standard metric ones 10^3, 10^6,10^9.

   (d) Suggest reasons why we need offset addresses.
Within a memory chip, each memory location has a physical address. The chip offset address is the number used to specify that this particular chip is being accessed. For example a 128(0x80) byte RAM is located at offset address 0x10. To access the physical location 0x25 within the RAM chip, the system uses the address 0x25+0x10=0x35. Offset addressing allows us to use multiple chips in the same address space. By using multiple chips, gaps may be purposely left, for use in memory mapping peripherals, or CPU special registers.

   (e) Suggest reasons that RAMs traditionally have been organized as only one bit per chip whereas ROMs are usually organized with multiple bits per chip. [4.1][Sta00]
Quoting from Stallings:

> In the memory hierarchy as a whole, [i.e. static/dynamic RAM; ROM/EEPROM] we saw that there are tradeoffs amoung speed, capacity and cost. These tradeoffs also exist when we consider the organization of memory cells and functional logic on a chip. For semiconductor memories, one of the key design issues is the number of bits of data that may be read/written at one time.

We note that traditional static RAM cells are much larger than ROM or even the newer DRAM cells. Therefore it was more reasonable for designers to choose a memory layout with 1 bit per chip for Static RAM.

   (f) Discuss the advantages and disadvantages of storing program and data in the same memory. from: http://smokey.homeip.net/Archive/Papers/scithought.html
Disadvantages: 1) Instruction and data fetches cannot be performed simultaneously.
2) Instructions may inadvertently be overwritten (if intentionally overwritten – advantage)
Advantages: 1) Memory can be allocated as necessary to data/program storage.

3. (a) Contrast the following with reference to the different micro-operations required within the CPU:

   i. RISC and CISC philosophy
   RISC instructions will typically require fewer micro-operations than CISC instructions. RISC philosophy is based on simple instructions, CISC philosophy is based on complex instructions.

   ii. separate vs. shared address/data buses
   For shared use (multiplexing) of the address/data lines, the CU must place the address and data on the same lines at different times. For separate address/data lines, the CU can perform the required micro-operations simultaneously.

   iii. Von Neumann vs. Harvard architectures
   The instruction fetch, decode and write stages will all use the same bus for the Von Neumann architecture. It will therefore not be possible to conduct micro-operations for instruction fetch and operand fetch (decode) simultaneously, thus preventing the easy implementation of pipelining. The Harvard architecture with it's sperate buses, will readily support simultaneous micro-operations for instruction and operand fetches, and hence pipelining.

   (b) Draw the timing diagram for a memory fetch across a synchronous bus with shared address and data lines.
   This will look the same as the synchronous fetch from Stallings Fig 3.19, except that the address and data traces would be replaced with a single address/data trace which shows two activity periods.

   (c) A simple stack may be implemented using registers to hold the addresses of the stack pointer (SP), and stack bottom (SB). The CPU may then have instructions POP and PUSH. Show the sequence of micro-operations for popping and pushing the accumulator value onto the stack.
   PUSH: [W]→push_value; [SP]→address lines; request read; data lines→W; increment [W]; [W]→address lines; push_value→data lines; request write; [SP]→address lines; [W]→data lines; request write.
   POP: [SP]→address lines; request read; data lines→W; [W]→address lines; request read; data lines→popped_value; [W]-1→holding register; [SB]→address lines; request read; [W]-data lines→W; if W is 0 stop; [SP]→ address lines; holding register→data lines; request write; popped_value→W.

   (d) For an accumulator-based CPU with a pipeline depth of 4, list the micro-operations required to perform successive ADD operations.
   Pipeline depth of 4 means that 4 stages may be processed simultaneously.
   Cycle 1: FetchA
   Cycle 2: FetchB, DecodeA
   Cycle 3: FetchC, DecodeB, ExecuteA
   Cycle 4: FetchD, DecodeC, ExecuteB, WriteA
   Cycle 5: FetchE, DecodeD, ExecuteC, WriteB
   FETCH: [PC]→address lines; request read; increment [PC]; data lines→IR
   DECODE: [IR]<n:x> → (addressing mode) → ALU inputs; [IR]$< 0 : n >$ → ALU control inputs
   EXECUTE: [W]→ALU inputs; trigger ALU; ALU outputs→[W]
   WRITE: [IR]$< x : q >$ → (addressing mode) → address lines; ALU outputs → data lines; request write.

page I 9

1. (a) 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?

   (b) 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?

   In both cases it is a matter of efficiency/code size. General branch and add instructions require an additional argument, which skip, and increment do not (implicit argument is 1). In a variable length instruction set this would mean that skip and increment instructions occupied less program memory. Furthermore, the CPU may be designed to perform increment operations (which occur frequently) differently and more efficiently than general purpose addition.

   (c) The nop(no operation) instruction has no effect other than incrementing the program counter. Suggest some uses of the nop instruction. [Sta00, 9.5]
   The nop instruction can be used to introduce a known delay, or to pad out the program (fill the blank locations).

   (d) What is the difference between a program branch, and a subroutine call?
   A branch (goto, skip) simply changes the contents of the program counter. A subroutine call (function) stores the value of the program counter before it changing it. This is so that when the subroutine 'return's the value of the program counter can be restored.

   (e) How can CPU use a stack for any purpose, if there are no push and pop operations in the instruction set? [Sta00, 9.7]
   Push and Pop operations in the instruction set let the programmer manipulate the stack explicitly. Even if there are no explicit instructions, the CPU may use a stack to perform it's internal functions. Two examples: to hold ALU arguments in a zero-address machine; to hold the program counter values during subroutine calls.

2. (a) What are the relative advantages and disadvantages of each of the addressing modes? [Sta00, Table 10.1]

   Reproducing the table:

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA=A	Simple	Limited address space
Indirect	EA=*A	Large Address Space	Multiple Memory references
Register	Operand=R	No memory reference	Limited address space
Register indirect	EA=R	Large address space	Extra memory reference
Displacement	EA=A+R	Flexibility	Complexity
Stack	EA=top of stack	No memory reference	Limited applicability

   EA= effective address
   A= contents of an operand in the instruction
   R= contents of a register indicated by an operand in the instruction.

(b) Given the following memory values and a one-address machine with an accumulator, what values do the following instructions load into the accumulator? [Sta00, 10.2]

i. LOAD IMMEDIATE 20
ii. LOAD DIRECT 20
iii. LOAD INDIRECT 20
iv. LOAD IMMEDIATE 30
v. LOAD DIRECT 30
vi. LOAD INDIRECT 30

Address	20	30	40	50
Data	40	50	60	70

The answers are 20, 40, 60, 30, 50, 70.

(c) How would you perform an indirect addressing operation, using direct and immediate addressing mode instructions?
Using Register Direct addressing and immediate mode addressing we get LOAD INDIRECT x is the same as LOAD DIRECT x, ACCUMULATOR→REGISTER, LOAD REGISTER.
Using direct and immediate addressing only, we would have to alter the following instruction. LOAD INDIRECT x is the same as LOAD DIRECT x, ACCUMULATOR→next instruction argument n, LOAD DIRECT n.

(d) 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. [Sta00, 9.4] 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:

i. Data transfer: Location X to accumulator, accumulator to location X
Presuming we keep locations Y,Z as scratch space which we always leave as zero, we can move X to the accumulator using

```
subs Z ; zero accumulator
subs Z
subs X ; negate X
subs Z ; zero accumulator
subs Z
subs X ; restore X and copy to accumulator
```

and the accumulator to X using

```
subs Z ; store accumulator in Z
subs X ; zero accumulator and X
subs X
subs Z ; accumulator is negative of original
subs X ; now negative value is in X
subs Z ; put zero back in Z
subs Z
subs X ; original value back in X and accumulator
```

ii. Addition: Add contents of location X to accumulator

Addition is the same as negating, the negative accumulator subtract X.

```
subs Y ; store accumulator in Y (presumes Y was zeroed previously)
subs Z ; zero out accumulator and Z
subs Z
subs X ; X and accumulator are negative original X value
subs Y ; Y and accumulator are negative the sum of original values
subs Z
subs Z
subs X ; restore the X value
subs Z
subs Z
subs Y ; Y and accumulator are now the sum of the values
```

iii. Conditional branch

Consider a straight branch to the line which is the value in the accumulator below the jump instruction (presuming that the architecture manipulates n-1 bit numbers). Simply add to the default jmp 0 address.

```
subs Y     ; store accumulator in Y (presumes Y was zeroed previously)
subs Z     ; zero out accumulator and Z
subs Z
subs BCH   ; X and accumulator are negative original X value
subs Y     ; Y and accumulator are negative the sum of original values
subs BCH
subs BCH
subs Y
subs BCH   ; final modification
subs Y
subs Y     ; clear Y
BCH jump BCH+1 ; jump instruction which was modified
```

Conditional branch can be done by placing either a particular value or 0 in the accumulator prior to the branch.

iv. Logical OR

(A OR B) is A+B minus the carry of A+B plus the carry of A+B (shifted right). Any thoughts on extracting the carry byte?

3. (a) Differentiate between fixed length and fixed format instruction sets. What are their respective implications for CPU operation.
A fixed length instruction set, is one in which all instructions have the same number of bits. A fixed format instruction set is one in which the fields (i.e opcode,operands) of all the instructions have the same lengths. A fixed length instruction set simplifies the instruction fetch stage of the instruction cycle, as the amount of data to be retrieved from the program memory is known. A fixed format instruction set simplifies the decode stage of the instruction cycle, as operands will always be at the same location within the instruction; particular locations can thus be hardwired to the ALU/data bus, simplifying the CPU micro-operations.

   (b) 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? [Sta00, 10.9]
The maximum number of instruction variants is 2^{16}. The K instructions which take 2 operands will have a maximum of $K * 2^6 * 2^6$ variants. The L instructions with no operands will have a maximum of L variants. The remaining variants are:

$$2^{16} - L - (K * 2^{12})$$

If all these variants were available for 1 operand instruction variants, the number of one operand instructions would be

$$\frac{2^{16} - L - (K * 2^{12})}{2^6}$$

   (c) Is there any possible justification for an instruction with two opcodes? [Sta00, 10.12]
The answer depends on what you mean by two opcodes. There are cases where additional bits in the instruction, may be used to further specify the operation to be performed. Those additional bits could either be thought of as part of an extended opcode, or as a separate opcode. One example is the Pentium II Instruction Format (see diagram Stallings page 398) where the general opcode is optionally extended by 3 additional bits, which are not physically next to the original bits.

4. (a) Differentiate between the host and target machines with reference to the IDE, the compiler, and the generated machine code.
The IDE and compiler run on the host machine. The compiler generates machine code for the target machine. The IDE may simulate or help emulate the operations of the target machine. The IDE may be used interface with/ download machine code to the target machine.

   (b) Why do we need a simulator, when we can debug directly on the system?
1) The hardware may not be complete. Imagine working on a group project where some people are responsible for software, and others for hardware. The software people can get on with testing their code without waiting for the hardware people to finish. 2) It's faster/easier to debug in the simulator. Downloading code to the target takes time, and in some cases may the target may have a finite number of programming cycles. It would therefore be more expedient to remove bugs in simulation. The simulator may also give you more access to the internal state of the simulated target, than the debug facilities can give for the real target.

   (c) Why do we need debug facilities when we can just use the simulator?
The simulator runs much more slowly, and (unless you have a circuit-level simulator) performs each instruction as a unit. This means that simulating asynchronous events, or events which occur and finish between instructions, is not possible. In such cases, the code must be run on the target machine, and the debugger can be used to monitor the state of the target machine.

   (d) Suggest some possible advantages of programming in assembly language rather than in high level languages (e.g. C).
In assembly language, you have direct control over how operations are performed. You can therefore make code optimizations SPECIFIC to your particular application, that the compiler would not be able to make(e.g. removal of the carry check in addition/subtraction for Booth's). You also have better control over hardware/ peripherals. A final year student had a problem programming his PIC16F877 in C, because he was sharing pins on a particular port. When he called the routine provided for configuring a particular set of pins, it cleared the settings for the other pins on the port.

page I 12

1. (a) What are the advantages and disadvantages of memory-mapped I/O as compared to isolated I/O?
   Memory-mapped I/O advantage:
   1) No dedicated instructions required to read the I/O devices.
   2) DMA can easily be used to move information from I/O module to memory as they share the same bus lines.
   Memory mapped I/O disadvantage:
   1) Address assignment/offsets for I/O modules use space that could have been used by memory.
   2) I/O and fetch from memory cannot occur simultaneously.

   (b) Differentiate between bus arbitration and DMA with reference to the CPU's ability to use the bus.
   In arbitration, the CPU requests and waits for permission to utilise the bus; once granted the CPU has full control of the bus until it completes it's current transaction(regardless of the length of the transaction). In DMA, if there ia a data word to be transferred across the bus, the CPU is suspended just prior to performing a transaction on the bus. Arbitration occurs between different bus masters. DMA steals cycles from the CPU (bus master).

   (c) Outline how I/O modules could use the interrupt line to move data to memory as soon as it is available.
   When data is received, the I/O module raises the CPU interrupt line. The CPU will process the interrupt at the end of the instruction cycle. The interrupt routine will fetch the data from the I/O module, and then store it in memory.
   What are the relative advantages/disadvantages compared to DMA.
   I/O interrupt Disadvantage:
   1) Processor is managing the I/O transfer from module to memory.
   2) Large overhead associated with interrupts for each small amount of data received.
   I/O Interrupt Advantage:
   1) Processor can manipulate/repsond to the received data immediately.

   (d) 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? [Sta00, 6.5]
   Presuming that the data width of the system bus is a byte, the DMA module will steal a cycle each time a byte has arrived. If the device receives 9600 bits per second, 1200 bytes will be received every second. The CPU will slow from 1 MIPS to (1,000,000-1200) Instructions per second.

2. (a) 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?
   Multiple interrupts can either be prioritised, or serviced on a first-come first-served basis. If they are being serviced in order of arrival, then it is impractical to have them interrupt each other, as there will be no service guarantee for the individual handler. In priority based schemes, there are only a limited number of higher priority interrupts which could be serviced first, thus the handler execution time is bounded. Interrupts with individual handlers may be either polled, or vectored. For polled interrupts, the main interrupt handler, determines who triggered the interrupt, and calls the relevant handler. For vectored interrupts, the address of the handler is placed on the bus when the interrupt is triggered, so that the CPU can go there directly.

   (b) 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?
   The clock signal could be used as input to the counter. When queried the counter returns the current value. The counter will take no system resource to update, unlike a memory update from interrupt.

3. 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 [Sta00, 6.9].
   This problem is intended to illustrate different I/O mechanisms. In analogy, Apple Eater is the I/O module, and Apple Server is the CPU.

   (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.
   The alarm clock goes off at regular intervals. When it rings, Apple-server sends over an apple and Apple-eater knows when to expect it. Analogous to blind cycle/timer interrupt polling of I/O.

   (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.
   Apple Server keeps looking for the flag, and throws an apple when he sees it. Apple Server must always be looking for the flag, unlike the previous case, where he could wait until the alarm clock rang, and was certain about the interval. Analogous to busy polling.

(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 (3b) using the string.
Apple Server ties an apple to the string whenever it appears. Apple Eater pulls the string when he is ready to eat, and replaces it as soon as he is done. The advantage is that Apple Server has the entire interval to determine that the string is in place. Analogous to DMA.

4. List the issues to be considered in the design/choice of a CPU for a particular application.
Speed, bus lines, clock/power requirements, accessible i/o and address space, instruction set (will reflect architecture).

page II 3

1. (a) Which is better: a microcontroller-based system or a microprocessor-based system?
   It depends on the application and your definition of "better" in relation to the application. Better can mean smaller, cheaper, less power-hungry, more reliable, more revision/expansion capability.

   What applications is a microcontroller (as opposed to a microprocessor) typically used for?
   Microcontrollers are generally used for large quantity embedded applications which do not require a lot of power, must interact with sensors/actuators/displays, and need to be cheaply mass-produced with many variants e.g. remote controls, microwaves, calculators etc.

   (b) Differentiate between the terms "file register" and "register file".
   File register - single GPR or SFR; Register file - collective term for all registers.

   (c) When pipelining with static branch prediction, is there an advantage in assuming that the branch is always taken?
   There is no inherent advantage. The number of pipeline flushes will depend on the frequency with which a given branch occurs.

   (d) Suggest an application where the PIC Parallel Slave Port mode would be useful.
   Quoting from the data sheet:

   > The Parallel Slave Port can directly interface to an 8-bit microprocessor data bus. The external microprocessor can read or write the PORTD latch as an 8-bit latch.

   In this mode, the PIC can be used as an I/O module, which receives/returns settings/data through the PSP.

   (e) After the addition of 9 and 8, which of the flags (Carry, Zero and Digit Carry) would be reset(i.e value 0)?
   Carry is reset, Zero is reset, Digit Carry is set.

page II 5

1. (a) Explain the concept of banking memory space.
   Banking is the practice of subdividing the address space, so that only the addresses in the currently selected bank can be accessed. You may also see this referred to as paging.

   (b) Give reasons why memory banks may be used in a microcontroller.
   1) Insufficient space for the full address in the instruction.
   2) Deliberate isolation of particular registers.

   (c) How/Where are memory banks used in the PIC 16F877?
   Memory Banks are used to directly address the register file (bits RP1,RP0), indirectly address the register file (IRP bit), and to directly address the program memory in goto/call statements(PCLATH).

   (d) Certain general purpose and special function registers are mapped to multiple addresses in the data memory space of the PIC 16F877. What possible advantage could such a scheme offer?
   By mapping frequently used registers to multiple banks, the frequency of bank switching (and hence code size) is reduced.

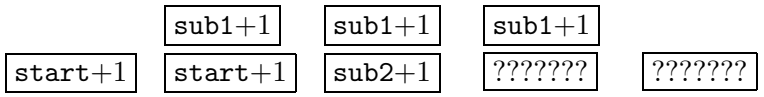
(e) For the PIC16F877, what special considerations must be made, when performing/returning from a long subroutine call (i.e. outside the current program bank)?
Before making the call, PCLATH must be set to reflect the subroutine bank. After returning, PCLATH must be set to reflect the current bank.

2. (a) The PIC16F877 has both data EEPROM and static data RAM (general purpose registers) included in the microcontroller. Suggest reasons (using application examples) why the data EEPROM is also included in the microcontroller.
Strictly speaking, in a Harvard architecture processor, there is no way to access the program memory (there is a work-around for the PIC but let's ignore it for now). If there was only the Program FLASH and the static RAM, there would be no place to store constant or semi-constant values which could be used in the program. Examples of such values would be: PID setpoints in a digital controller, or a lookup table for conversion from sensor reading to display value.

(b) 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.
The behaviour of the PIC 16F877 is such that when the stack is overrun, the first element is overwritten, and when underrun, an indeterminate value will be returned. For example, a program makes three nested subroutine calls:

```
start   call    sub1
        sleep
sub1    call    sub2
        return
sub2    call    sub3
        return
sub3    return
```

If the stack is only 2 levels deep, the call to `sub3` will cause the original stack entry `(start+1)` to be overwritten by `(sub2+1)`. When the program tries to return from sub1, it will not be able to do so correctly.



(c) The SFRs discussed so far include: STATUS, PCL, PCLATH, FSR, INDF, EEADR, EEDATA and EECON1. For each of the above, using the data sheet[PIC01]

i. locate all the mapped addresses in the data memory space

ii. describe the meaning of the register/individual bits as appropriate

STATUS	03h,83h,103h, 183h	Contains the bank bits and the ALU flags
PCL	02h, 82h, 102h, 182h	Bottom 8 bits of the program counter
PCLATH	0Ah 8Ah 10Ah 18Ah	Holding register for the upper bits of the program counter
FSR	04h 84h 104h 184h	Holds address for indirect addressing
INDF	00h 80h 100h 180h	References data at address contained in FSR
EEDATA	10Ch	Holds the data to send or that was retrieved from EEPROM
EEADR	10Dh	Holds the address to be referenced on the EEPROM
EECON1	18Ch	Contains flags to initiate read/write, and report an error.

page II 11

1. (a) Explain (and illustrate using pieces of code) the direct and indirect mechanisms of addressing Special Function Registers.
The special function registers are all mapped into the data memory space. Therefore they may be addressed using the same mechanisms that are used for data RAM (i.e. using the mapped address for direct addressing; placing the mapped address in FSR, then using INDF for indirect addressing). The following examples address the option register directly and indirectly(respectively) to set the T0CS (Timer 0 clock source select internal/external) bit. Both examples ignore the 9th bit (IRP or RP0 in STATUS) but this is OK as the option register is mapped into both Bank1 and Bank3.

```
bsf     STATUS,RP0      ; jump to bank 1/3
bsf     OPTION_REG,T0CS

movlw   OPTION_REG
movwf   FSR
bsf     INDF,T0CS
```

(b) What happens when the assembly language statement `clrf STATUS` is executed?
When the `clrf` instruction is executed on a general purpose register, or on most special function registers, it clears all 8 bits of the register. Because the STATUS register contains bits which are determined by external conditions, and by the ALU, all bits cannot be cleared. Quoting from the data sheet:

> CLRF STATUS will clear the upper three bits and set the Z bit. This leaves the STATUS register as 000u u1uu (where u = unchanged).

(c) Explain how the INDF register behaves when it is indirectly addressed for read/write.
The INDF register is an imaginary register. The contents read/written in an instruction which uses INDF will be those of the location specified by the FSR register, and the IRP bit of the STATUS register. Because the INDF register is imaginary, indirectly addressing INDF is an invalid act. Quoting from the data sheet:

> Reading the INDF register itself, indirectly (FSR = 0) will read 00h. Writing to the INDF register indirectly results in a no operation (although status bits may be affected).

(d) Differentiate between the indirect addressing mechanisms for data RAM and data EEPROM.
The indirect addressing mechanism for data RAM uses the FSR register to hold the bottom 8 address bits, and the IRP bit of the STATUS register for the uppermost address bit. The data can be accessed by reading/writing the INDF register in the following instruction cycle. The indirect addressing mechanism for data EEPROM uses the EEADR register to hold all 8 address bits (the data EEPROM only has 256 locations). The data is accessed by reading or writing the EEDATA register. Note that unlike the data RAM, the EEPROM is not actually read/written when the EEADR/EEDATA register is altered. To read: Set the RD bit – the data can then be accessed in the following instruction cycle.

2. (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).

```
movlw   0x10
movwf   PCLATH
goto    0x10
```

(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`).

```
swapf   Reg,F
swapf   Reg,W
swapf   Reg,F
```

(c) 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.
Quoting from [Pre01]:

> At the end of the routine, bit 0 of "X" will have the "Even" Parity bit of the original number. "Even" Parity means that if all the 1's in the byte are summed along with the Parity bit, and even number will be produced.

```
swapf   X,W
xorwf   X,F
rrf     X,W
xorwf   X,F
btfsc   X,2
incf    X,F
```

(d) 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.
The code in the notes tested for bit 4 being set after starting from 0x20. In this case, we wish to test for bit 3 being set after starting at 0x40. Finally we need to save and restore the accumulator when initialising FSR.

```
    movwf   W_sav
    movlw   0x40
    movwf   FSR
    movf    W_sav,W
Lp  clrf    INDF
    incf    FSR,F
    btfss   FSR,3
    goto    Lp
```

3. 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.

   (a) When the microcontroller is reset, does `stkptr` need to be initialized? Yes.
   If so, what should its initial value be? Choose the lower end of a free block of 16 registers. In the case of the PIC16F877, an appropriate block could be 0x20 to 0x2F.

   (b) Assuming that the stack is never popped when empty, is it necessary that it be cleared when initialized? No.

   (c) 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.
   For the "push" routine, it is assumed that the value to be pushed is in the working register and that there is a variable, `temp`, that is used for temporary storage.

```
temp    EQU     0x0D      ; temporary storage

; stkptr has been assigned elsewhere Push    movwf  temp       ;
save the parameter
        incf   stkptr,F   ; point stkptr to new location
        movf   stkptr,W   ; get the stkptr into FSR
        movwf  FSR        ; ... to do indirect addressing
        movf   temp,W     ; now get the parameter
        movwf  INDF       ; store it in location stkptr points to
        return
```

   (d) Do the same for the above for the pop routine.
   Here there is no need for a temporary variable, and the popped parameter is returned in the working register.

```
Pop     movf   stkptr,W   ; get the stkptr into FSR
        movwf  FSR        ; ... to do indirect addressing
        movf   INDF,W     ; retrieve the value
        decf   stkptr,F   ; point to the new location
        return
```

4. 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.

   (a) When the CPU is reset, how should each of the 19 bytes be initialized, if at all? `inptr` and `outptr` should have the same value i.e. the address at the beginning of the block of memory, and `qcount` should be initialised to 0. The initial values of the 16 queue locations is irrelevant.

(b) 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.
Assuming that the queue variables have all been initialized and that the queue ranges from 0xB0 to 0xBF then the put routine can be as follows with the working register having the parameter to be stored.

```
temp    equ     0x0D
starq   equ     0xB0        ; for information only
endq    equ     0xBF        ; ditto

; assume qcount has been assigned elsewhere
Put     movwf   temp        ; Save the parameter
        incf    qcount,F    ; update number in queue
        movf    inptr,W     ; get the inptr ready for indirect
        movwf   FSR         ; ... addressing
        movf    temp,W
        movwf   INDF        ; store parameter
        incf    inptr,F     ; point to new storage locn
        return
```

(c) 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.
Assume that the same definitions exist as for the queue Put routine. The value from the queue is returned in the working register.

```
Get     decf    qcount,F    ; reduce number in queue
        movf    outptr,W    ; setup for indirect addressing
        movwf   FSR         ; ...using outptr
        incf    outptr,F    ; point to new output locn
        movf    INDF,W      ; get value from queue
        return
```

(d) 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.

(e) In the queue get routine, add a check to prevent retrieval of data if the queue is empty.

(f) In the queue put routine, add a check to prevent insertion of data if the queue is full.

```
Put     movwf   temp        ; Save the parameter

        movf    qcount,W
        sublw   0x10
        btfsc   STATUS,Z    ; check if subtraction from 16 is zero
        return              ; if so, leave early

        incf    qcount,F    ; update number in queue
        movf    inptr,W     ; get the inptr ready for indirect
        movwf   FSR         ; ... addressing
        movf    temp,W
        movwf   INDF        ; store parameter
        incf    inptr,F     ; point to new storage locn

        movf    inptr,W
        sublw   endq+1
        btfsc   STATUS,Z
        movf    inptr,starq ; if inptr is past endq change to starq
        return

Get     movf    qcount,F
        btfsc   STATUS,Z    ; leave get early if qcount is 0
        return
        decf    qcount,F    ; reduce number in queue
        movf    outptr,W    ; setup for indirect addressing
        movwf   FSR         ; ...using outptr
        movf    INDF,W      ; get value from queue
        movwf   temp        ; store temporarily

        incf    outptr,F    ; point to new output locn
        movf    outptr,W
        sublw   endq+1
        btfsc   STATUS,Z
        movf    outptr,starq ; if outptr is past endq change to starq

        movf    temp,W      ; return with the gotten value
        return
```

5. 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. 2-stage pipeline

   (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? Because pipeline is flushed during branch.

6. (a) 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?
   Lookup tables must be either coded into the program (i.e. case statements), or set up in the data memory by the program at the start. Using a lookup table in the prior case would require a subroutine/function call. In both cases, program memory is inefficiently used, as there is instruction overhead, as well as the actual data values.

   (b) The PIC16F877 allows an indirect read of the program memory to be performed. Describe the mechanism used, using an assembler code example.
   The mechanism is similar to that used to indirectly address the data EEPROM. The primary differences are that the address and data each occupy two registers (EEADRH and EEADR, EEDATH and EEDATA) and that a delay of two instruction cycles is required between setting the RD bit and reading the data registers.

page III 11

1. Using the techniques practiced above, set up a program (Program first in C and then in assembler) which will:

   (a) add two global variables `b` and `d` and places the result in global variable `r`

```
int b,d,r;

void main() {
    b=1;
    d=2;
    r=b+d;
}
```

```
main    movlw 1
        movwf b
        movlw 2
        movwf d
        addwf b,W
        movwf r
        sleep
```

   (b) call a subroutine called `addbd` that adds two global variables, `b` and `d` and places the result in global variable `r`. Call the subroutine from the main program.

```
int b,d,r;

void addbd() {
    r=b+d;
}

void main() {
    b=1;
    d=2;
    addbd();
}
```

```
main    movlw 1
        movwf b
        movlw 2
        movwf d
        call addbd
        sleep

addbd   movf b,W
        addwf d,W
        movwf r
        return
```

(c) Place an infinite loop around the code. Input `b` and `d` from `portb` and `portd` respectively. Use the stimulus (asynchronous or files) to test the code is working.

```
int b,d,r;

void addbd() {
    r=b+d;
}

void main() {
    for (;;)
    {
        b=input_b();
        d=input_d();
        addbd();
    }
}
```

```
main    clrf b
        clrf d
loop    movf PORTB,W
        movwf b
        movf PORTD,W
        movwf d
        call addbd
        goto loop

addbd   movf b,W
        addwf d,W
        movwf r
        return
```