CPU Architecture & Boolean Math

Background:

Microcontrollers are a type of computer which is designed for controlling devices, such as toasters, vacuum cleaners, etc. Most are built around a microcomputer with several features added:

- Memory is typically incorporated within the microcontroller, allowing a single-chip design.
- Timers are added, and
- Analog inputs / output are often added.

For example, the microcompuer and corresponding microcontroller are

- Intel: 8080 / 8051
- Motorola: 6800 / 6812

Microchip is a small company which specializes in microcontrollers for start-up companies. We're using them in ECE 376 simply because they are inexpensive, they work well, and the tools (compilers, boot loaders, etc.) are available and free. Free is good. Other microcontrollers (Intel 8051, Motorola 6812 etc.) all behave about the same. Once you are comfortable with one you have a pretty good idea for how all the others work.

Microcontrollers are becoming increasingly popular due to cost. Anything you can do in software you can do in hardware and visa versa. If you design a circuit around a microcontroller, the function is determined by the code you write. If you need a different function, you simply need to change the program. This flexibility can greatly reduce the cost of devices for companies. If you change hardware, you may be talking hundreds of thousands of dollars to revamp the assembly line, rectify the circuit board, change the packaging, etc. If you change the software, the cost is (in theory) zero. Plus, microcontrollers are becoming so inexpensive they are reasonable solutions to many problems. A low-end PIC microcontroller, for example, costs $0.67 in quantity. You don't need a computer to run a toaster, but at $0.67, why not?

CPU Architecture

All computers have five main sections:

- Program Memory
- Data Memory
- Stack Memory (to store data such as the return address when you call a subroutine)
- Registers (to store data which is being manipulated), and
- Arithmetic Logic Unit (ALU) which does the addition, subtraction, etc.

All five of these are incorporated into microcontrollers, which is both a blessing and a curse. On the good side, you can design an embedded system with a single chip. On the bad side, you're stuck with whatever is inside that chip.

Computers are 'rated' as being 4-bit, 8-bit, 16-bit, etc. This refers to the size of the registers on the computer and the size of the data bus. For an 8-bit microcontroller, a read / write / or / etc. function manipulates 8 bits at a time.

There are two main styles for building microcontrollers. In Von-Neuman architectures (Motorola, Intel), all memory is the same size. In the 6812, for example, all memory is 8-bits (one byte). This is an advantage in a sense since you can interchange program memory with data memory with stack memory, etc. It also allows you to use the stack to store data. This is often how you pass data to a subroutine (the data is placed on the stack when the subroutine is called.)

Since the memory can be allocated at will, you often need to define what section of memory is allocated to the program, to data, and to the stack. This is termed a memory map. The default for a 6812 is shown below as an example.

Address	Allocation	Memory Type
0	Data	RAM
4,095		
4,096	Stack	RAM
16,583		
16,584	Program	RAM or FLASH EPROM
65,536		

It has a disadvantage in that you only have 8-bits for program memory, allowing at most 256 instructions. To get around this, Motorola and Intel use several bytes to code each instruction. For example, to read the data at address 0x1000 into register A, three bytes are used:

- Byte 1: Memory read to A
- Byte 2: High byte of address (0x10)
- Byte 3: Low byte of address (0x00)

In addition, it can take several clocks to execute one instruction. The 6812, for example, takes up to 12 clocks to execute a single line of assembler code. This makes the speed of these devices somewhat difficult to define.

PIC microcontrollers us a Harvard architecture instead. With a Harvard architecture, program memory, data memory, and stack memory are separate and cannot be interchanged. This allows you to optimize each. For example, the microcontroller we're using (the PIC18F4626) has

- 32768 lines of 16-bit program memory
- 3968 lines of 8-bit data memory, and
- 31 lines of 15-bit stack memory[1]

By using 16-bit program memory, each instruction takes one line of code and each instruction executes in one clock cycle. Hence a PIC running at 5MHz is faster than a 6812 running at the same speed. Since the dimensions are different, however, you cannot exchange memory types. If you have an application which needs less program memory but more data memory, you can make this swap with a Von Neuman architecture. You cannot make this swap with a Harvard architecture.

[1] $2^{15} = 32768$. 15 bits are required to address 32768 lines of program memory

ROM / RAM / Stack:

ROM is used for

- Vectors (Tell the CPU where to go on certain events, such as power on, 10ms clock tic, etc.)
- Tables (information the program will use. Feedback gains, wait times, etc. This allows you to change the program by changing data in the table.)
- Main Routine (tells the CPU what to do)
- Subroutines (small parts of the mainline routine)
- Interrupt Service Routines (programs that are called by hardware events, like a switch closing or 10ms clock tic).

All of this must fit in the 32k ROM.

It's preferred if you keep these blocks together to simplify debugging. (The address tells you what the data means. In a table, it's data. In the main routine, it's an instruction, etc.)

RAM is used for

- Sending and receiving data from the microcontroller
- Saving data from program execution.

RAM split into banks of 4096 bytes.

- Data RAM is located in the low bank at address 0x0000 to 0x0FFF (0 to 4095).
- Special functions are located 0xFF00 to 0xFFFF.

For us, this doesn't mean much: you can have a single array that's 3198 bytes of 3198 variables, each of which is one byte. It doesn't matter.

These banks will cause problems with future upgrades, however. If a future version of this chip has 32k or RAM, for example, there are only enough bits in the instruction to address 4k of that RAM. To adderss all 32k, you need an extra 3 bits somewhere else. These extra 3 bits create 8 banks of 4k RAM. At any time, you can access only one of these banks.

This is a common practice in microprocessors and microcontrollers. It allows you to upgrade previous designs without changing the design too much. (Adding more bits to each instruction means a completely new processor design.) It causes great pains to programmers, however, since accessing different banks of RAM becomes difficult.

Pipeline & Program Timing

A PIC processor has a 2-level pipeline

- Level 1: The next instruction in the program (moved to Level 2)
- Level 2: The present instruction being executed.

This results in 1 instruction being executed each clock: the first level pre-fetches the next instruction to be executed so that it is available one clock in the future for execution. The exception is when a jump occurs. When this happens, the wrong instruction was preloaded into the stack (in Level #1). This wastes one clock to load the right instruction into the pipeline (i.e. jumps take 2 cycles to execute, every other instruction takes just one.)

Registers

Registers are where data is stored when you want to add two numbers, compare two numbers, etc. They're generally very useful and the more you have the better. The Motorola 6812 has six registers for your use. This allows you to use one for a counter, another for data, a third for a result, etc. All PIC microcontrollers only have **one** register (W). This makes programming a little simpler - you have no choice. Everything must go through the W register. This makes programming rather tedious, however. You sometimes have to use convoluted logic when you have only one register to play with.

Boolean Math

In a computer, everything is binary: data is only stored as ones and zeros. Likewise, all math is done using binary arithmetic.

Definitions:

- Bit: 1 or 0. A single flip flop or capacitor whose output is 5V (1) or 0V (0).
- Nibble: 4 bits.
- Byte: Eight bits.
- Word: More than one bit. 'Word' has no specific size in general.
- Binary: Base 2 arithmetic. 0b01010 means 'binary 01010'
- Decimal: Base 10 arithmetic Default is base 10.
- Hexadecimal: Base 16 arithmetic. 0x1234 means 'hexadecimal 1234'

Base N Numbers:

In base-10, we represent numbers in powers of 10. For example, 1,234 means

- 1×10^3
- $+2 \times 10^2$
- $+3 \times 10$
- $+4$

Similarly, in base 2, the number 0b110101 means

- 1×2^5
- $+ 1 \times 2^4$
- $+ 0 \times 2^3$
- $+ 1 \times 2^2$
- $+ 0 \times 2$
- $+ 1$

In base 16, 0x1234 means

- 1×16^3
- $+2 \times 16^2$
- $+3 \times 16$
- $+ 4$

The range of numbers you can represent with N bits using base X is

$$Range = 0 \text{ to } X^N - 1$$

For example,

- With 3 digits in base 10, you can represent numbers from 0 to 999.
- With 8 bits in base 2, you can represent numbers from 0 to 255 (2^8-1)
- With 16 bits in base 2, you can represent numbers from 0 to 65,535 $(2^{16}-1)$

Hexadecimal

Hexadecimal is a little more convenient than binary. The standard notation is as follows:

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

The advantage of using hexadecimal notation is

- You can represent 4 bits with a single number
- It's easy to convert to and from binary in hexadecimal.

Example: Convert the number 0x1234 to binary.

Solution: Go nibble by nibble:

Hexadecimal	1	2	3	4
Binary	0001	0010	0011	0100

So

0x1234 = 0b0001 0010 0011 0100

(The spaces are just to make the number easier to read.)

Example: Convert the binary number 0b00101010100100101010 to hexadecimal.

Solution: Separate into groups of 4 bits (nibbles)

```
0b      0010 1010 1001 0010 1010
0x       2    A    9    2    A
```

The answer is 0x2A92A.

To convert from base 10 to base 16, one method is to keep dividing by 16 and taking the remainder. For example, convert 2009 to hexadecimal:

2009 / 16 = 125 remainder 9 (9)

125 / 16 = 7 remainder 13 (D)

7 / 16 = 0 remainder 7 (7)

So, the answer is

2009 = 0x7D9

To convert from base 16 to base 10, use the definition

$$0x7D9 = 7 \times 16^2 + 13 \times 16^1 + 9 = 2009$$

Boolean Math

Addition: Add just like you do in base 10. Just remember to carry a 2 (base 2) or carry a 16 (hexadecimal)

Example:

Carry	(1)		(1)	
	0x4	A	2	6
+	0x9	C	8	D
	14	22 (16 + 6)	11	19 (16 + 3)
Result	0xD	6	B	3

0x4A26 + 0x9C8D = 0xD6B3

Subtraction: Again, just like base 10 but you borrow a 16

Borrow	(-1)	-1 + 16	-1 + 16	16
	0x4	A	2	6
-	0x2	C	8	D
	3 - 2 = 1	25 - 12 = 13	17 - 8 = 9	22 - 13 = 9
Result	0x1	D	9	9

0x4A26 - 0x2C8D = 0x1D99

Logical Operations:

And, Or, Xor operate on each bit independent of the other bits. For a byte, they behave as eight separate Boolean functions.

The truth table for and, or, xor are as follows:

A	B	A & B (and)	Comment
0	0	0	Bit clear 0 & X = 0
0	1	0	
1	0	0	no change 1 & X = X
1	1	1	

A	B	A | B (or)	comment
0	0	0	no change 0 | X = X
0	1	1	
1	0	1	set 1 | X = 1
1	1	1	

A	B	A ^ B (xor)	comment
0	0	0	no change 0 ^ X = X
0	1	1	
1	0	1	toggle 1 ^ X = not X
1	1	0	

If you have more than one bit, convert to binary and go bit by bit.

Example: Solve 0x12 & 0x34

Solution: Convert to binary

	1st nibble				2nd nibble			
0x12	0	0	0	1	0	0	1	0
0x34	0	0	1	1	0	1	0	0
and	0	0	0	1	0	0	0	0
hex	0x1				0			

Solution: 0x12 & 0x34 = 0x10

2's Compliment Notation

Suppose you want to represent a negative number in binary. How do you do that?

One idea is to add another bit - the sign bit. If it's a zero, the number is positive. If it's 1, the number is negative.

The problem with this idea is you need to change how you do addition depending upon whether the numbers you are adding are positive or negative. For example, find the result of 1+1 using 8-bit binary numbers:

```
    0000 0001
+   0000 0001
=   0000 0010
```

or 1 + 1 = 2. So far so good. Now, add +1 + (-1):

```
    0000 0001   (+1)
+   1000 0001   (-1)
=   1000 0010   (-2)
```

+1 + (-1) = -2. This didn't work out. There needs to be another way to represent negative numbers.

Suppose you have an 8-bit register. With 8 bits, you can only count from 0 to 255.

Suppose the number stored is 255. What happens when you add one?

```
255          1111 1111
+ 1          0000 0001
=            0000 0000
```

The result will be zero with a carry out. Ignoring the carry, the result is zero.

If you define (-1) as being whatever number produces zero when you add one to it, then 255 behaves like -1.

What's happening is similar to angles. With angles, you can always add or subtract 360 degrees and get the same heading. Likewise, +270 degrees = -90 degrees.

With 8-bit numbers, you can always add or subtract 256 and get the same result.

$$255 - 256 = -1$$

$$255 = -1$$

This results in slightly odd representation of numbers. Addition (and subtraction) all work out with 2's compliment notation, however.

Problem: What is -0x1234? Assume 16 bit data (4 nibbles).

Solution #1: Find what you have to add to +0x1234 to get 0x0000. That number will be -0x1234

```
              1   2   3   4
        +    (14)(13)(12)(12)
        = 1   0   0   0   0
```

-0x1234 = 0xEDCC

Solution #2: Toggle all the bits and add one. This method is easy for computers (where it knows the bit values). It's hard for people though.

```
 -0x1234 = -0b(0001 0010 0011 0100)
         =  0b(1110 1101 1100 1011) + 1
         =  0b(1110 1101 1100 1100)
         =  0x  E     D    C    C
```

With 2's compliment notation, half of the numbers are positive and half are negative (similar to representing angles from -180 degrees to +180 degrees.)

- If the first bit is a zero, the number is positive. The amplitude is the normal binary number.
- If the first bit is a one, the number is negative. The amplitude is whatever it takes to get to zero.

Problem: The number 0x1374 is stored in memory. What number does this represent?

Solution: The first bit is zero. The result is the amplitude of the number:

$$1 \times 16^3 + 3 \times 16^2 + 7 \times 16 + 4 = 4{,}980$$

Problem: The number 0xA374 is stored in memory. What number does this represent?

Solution: The first bit is a one.

If the number is unsigned, this represents:

$$10 \times 16^3 + 3 \times 16^2 + 7 \times 16 + 4 = 41{,}844$$

If the number is signed, this represents a negative number

$$41{,}844 - 2^{16} = -23{,}692$$

(Similar to angles where you can add or subtract 360 degrees, add or subtract 2^N where N is the number of bits being used.)

It's up to the programmer to know what the number means. The computer could care less. That's the strength of 2's compliment numbers: the computer it able to treat the number the same regardless of whether it represents +41,844 or -23,692.

Note: You'll see this later in the semester. When you display +1, the LCD display will show 00001. When you display -1, it will display 65,535 (in base 10) or 0xFFFF (in hexadecimal). In 2's compliment notation, -1 = +65,535 = 0xFFFF.