Explanation of PIC 16F84A processor data sheet -- Part 1: overview of the basics

This report is the first of a three part series that discusses the features of the PIC 16F94A processor. The reports will refer to the PIC 16F84A data sheet from www.microchip.com. Each report takes about 30-60 minutes to read (this report will take 60 minutes). In this report, we first review the basics of computer organization and technologies, and then cover some of the basic features of the PIC processor in the data sheet. Then, as an example, we present a simple application of the processor. Finally, we will discuss bit-wise logic operations since they will be used in Labs 2.1, 2.2, and 2.3.

Introduction and Computer Terminology

The PIC processor is a 16-pin computer in a chip with a

- Central processing unit (CPU), which runs the program.
  - Recall, a program is a collection of instructions that does some task.
- Memory, which stores the program and holds variable values
- Input/output (IO), which is how the chip interacts with the rest of the world.

A program can be written using a high level language like C, such as in Figure 1.

```c
main( )
{ int i;
  int n;
  n = 0;
  for (i=0; i<10; i++) n = n*(n+1);
  return();
}
```

**Figure 1.** Simple C language program.

This program has C language instructions such as “for” and “return”; variables such as “i” and “n”; and data constants such as “10” and “0”. Note that variables can be read and written to.

This program cannot be executed by a processor. A processor is a digital circuit which runs on digital signals, which in turn represent bits (0s and 1s). A C program is “high-level” and cannot be understood by a processor directly. The program must be compiled to another type of program called an executable code or program (or machine code or program). The executable code can be executed by the processor. A C program is called source code, and is eventually converted to executable code. Compiling can be done manually or by using software called a compiler. Compiling by hand is tedious but can sometimes be more efficient than compilers.

The executable code is made up of instructions of another language called the machine language. The machine language is made up of machine instructions that are strings of bits that can be understood by the processor. The machine language is specific to each processor. So the PIC has a machine language. Other processors, such as Intel’s Pentium, Qualcomm’s Snapdragon, and Texas Instrument’s OMAP, have their own machine language. Thus, a C
language program can be compiled by different compilers that are specific to different “target” processors. This will result in different executable programs.

Since machine instructions are strings of bits, they are difficult to read by humans. Another way to represent a machine program is by using assembly language. In assembly language, each machine instruction has a mneumonic. The mneumonic is a representation of an instruction that is easier to read by humans. As an example, the PIC machine instruction 00000011000001 has the mneumonic CLRF 1. This instruction clears a variable “1”. Note that the mneumonic CLRF 1 is easier to read than the machine instruction 0000011000001. Assembly language programs are written in text, and they need to be assembled (similar to compiling) to convert them to executable programs.

Programs and their variables are stored in the memory of the processor. Programs are stored in instruction memory or program memory, while variables and their data are stored in data memory.

There are different types of memory technologies:

- **Read only memory (ROM)** can be read but not written to. Some ROM technologies can be written to (or programmed) only once, at the factory.
- **Programmable ROMs (PROMs)** can be programmed by the user. Many PROM technologies allow reprogramming, though the reprogramming is relatively slow.
- **Erasable PROMs (EPROMS)** can be erased by ultra violet light and then reprogrammed. You may have used EPROMS in EE 260.
- **Electrically erasable PROMs (EEPROMs)** are erasable using electricity. They are more convenient to reprogram since they do not require ultra violet light. A special type of EEPROM is flash memory, used in thumb drives.
- **Random access memory (RAM)** can be read and written to.

PROMs are known as non-volatile memory since the circuit can be turned off and the data is still retained. RAM is referred to as volatile memory since when turned off, all the data is lost.

**PIC Data Sheet**

Figure 2 is a snapshot of the first page of the PIC data sheet. The left side has a quick overview of the chip, and the right side has the pin descriptions for two different package configurations: PDIP (or SOIC) or SSOP. We will focus on the 16 pin package. Read Figure 2 before continuing reading below.

The left side of Figure 2 has a list of the processor’s features. There are 35 machine instructions and each instruction (word) is 14 bits. The program memory can store up to 1024 machine instructions. Though it is not indicated in the figure, the program memory will be non-volatile flash memory.

Each data is 8 bits (a byte). Note that data is either a variable or data constant. Data is often referred to as an operand, and instructions as operations. There are 68 bytes of RAM and 64 bytes of EEPROM. The RAM can be used for variables that hold temporary values. For example in Figure 1, the dummy variable “i” in the for-loop can be implemented in RAM. The EEPROM can store values that must be kept over long periods of time. For example, a program that collects scientific data (e.g., temperature values sampled hourly) can store them in EEPROM.
Figure 2 has the PIC’s operating speed which is the clock rate. The PIC like all processors is a synchronous digital circuit and operates on a clock signal. The PIC has multiple options for clock signals. It can use an internally generated clock signal or one from outside. The internal signal is convenient but inaccurate. An external signal can be generated using a crystal. For this lab, you will use an oscillator crystal. Figure 3 shows example crystals. The figure also shows the configuration of the crystal with the processor.

**High Performance RISC CPU Features:**
- Only 35 single word instructions to learn
- All instructions single-cycle except for program branches which are two-cycle
- Operating speed: DC - 20 MHz clock input DC - 200 ns instruction cycle
- 1024 words of program memory
- 66 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 9-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
  - External RB0/INT pin
  - TMR0 timer overflow
  - PORTB<7:4> interrupt-on-change
  - Data EEPROM write complete

**Peripheral Features:**
- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - 25 mA sink max. per pin
  - 25 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Figure 2. PIC 16F84A.
Figure 3. Oscillator crystals and the oscillator configuration from the PIC data sheet (see Sect 6.2 of data sheet).

The pin diagram of Figure 2 shows the power and ground pins (VSS and VDD), oscillator pins (OSC), clear or reset pin (MCLR), and IO pins (RA and RB). The table in Figure 4 has a summary of the pins.

Figure 4 has a reference to Schmidt trigger circuits. Schmidt triggers are useful for cleaning up digital signals, and in particular if a digital signal has slow rise and fall times. The Schmidt trigger will sharpen these edges at its output. These circuits have high and low threshold voltages. If the input signal goes above the high threshold, the output signal goes high, and similarly for the low threshold. Figure 5 shows the schematic of a Schmidt trigger, and what happens to the signals. You can find more information about Schmidt triggers on the Internet:

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>PDIP No.</th>
<th>SOIC No.</th>
<th>SSOP No.</th>
<th>I/O/P Type</th>
<th>Buffer Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSC1/CLKIN</td>
<td>16</td>
<td>16</td>
<td>18</td>
<td>I</td>
<td>ST/CMOS(1)</td>
<td>Oscillator crystal input/external clock source input.</td>
</tr>
<tr>
<td>OSC2/CLKOUT</td>
<td>15</td>
<td>15</td>
<td>19</td>
<td>O</td>
<td>—</td>
<td>Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. In RC mode, OSC2 pin outputs CLKOUT, which has 1/4 the frequency of OSC1 and denotes the instruction cycle rate.</td>
</tr>
<tr>
<td>MCLR</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>I/P</td>
<td>ST</td>
<td>Master Clear (Reset) input/programming voltage input. This pin is an active low RESET to the device.</td>
</tr>
<tr>
<td>RA0</td>
<td>17</td>
<td>17</td>
<td>19</td>
<td>I/O</td>
<td>TTL</td>
<td>PORTA is a bi-directional I/O port.</td>
</tr>
<tr>
<td>RA1</td>
<td>10</td>
<td>18</td>
<td>20</td>
<td>I/O</td>
<td>TTL</td>
<td></td>
</tr>
<tr>
<td>RA2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>I/O</td>
<td>TTL</td>
<td></td>
</tr>
<tr>
<td>RA3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>I/O</td>
<td>TTL</td>
<td>Can also be selected to be the clock input to the TMR0 timer/counter. Output is open drain type.</td>
</tr>
<tr>
<td>RA4/TOCKI</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>I/O</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>RB0/INT</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>I/O</td>
<td>TTL/ST(1)</td>
<td>PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs.</td>
</tr>
<tr>
<td>RB1</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>I/O</td>
<td>TTL</td>
<td>RBO/INT can also be selected as an external interrupt pin.</td>
</tr>
<tr>
<td>RB2</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>I/O</td>
<td>TTL</td>
<td></td>
</tr>
<tr>
<td>RB3</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>I/O</td>
<td>TTL</td>
<td></td>
</tr>
<tr>
<td>RB4</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>I/O</td>
<td>TTL</td>
<td>Interrupt-on-change pin.</td>
</tr>
<tr>
<td>RB5</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>I/O</td>
<td>TTL</td>
<td>Interrupt-on-change pin.</td>
</tr>
<tr>
<td>RB6</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>I/O</td>
<td>TTL/ST(2)</td>
<td>Interrupt-on-change pin.</td>
</tr>
<tr>
<td>RB7</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>I/O</td>
<td>TTL/ST(2)</td>
<td>Serial programming data.</td>
</tr>
<tr>
<td>VSS</td>
<td>5</td>
<td>5</td>
<td>5,6</td>
<td>P</td>
<td>—</td>
<td>Ground reference for logic and I/O pins.</td>
</tr>
<tr>
<td>VDD</td>
<td>14</td>
<td>14</td>
<td>15,16</td>
<td>P</td>
<td>—</td>
<td>Positive supply for logic and I/O pins.</td>
</tr>
</tbody>
</table>

Legend: 1= input  O = Output  I/O = Input/Output  P = Power  — = Not used  TTL = TTL input  ST = Schmitt Trigger input

Note: 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: This buffer is a Schmitt Trigger input when configured in RC oscillator mode and a CMOS input otherwise.

Figure 4. Table 1-1 from the PIC data sheet.
Example Application: Blinking LED

The program on in Figure 1 is useless for a number of reasons but one very critical reason is that it has no input/output (IO). The CPU is executing the program but since there is no IO, the processor does not affect the outside world.

The following program called the “Blinking LED program” is a simple example that has IO. It assumes that an LED is attached to pin RA1, and it turns the LED on and off. The RA1 pin is part of the 5-bit wide PORTA. The bits of PORTA are RA4, RA3, ..., RA0. PORTA can be treated as a variable in software.

```
main( ) // BLINKING LED program
{
    // Set PORTA so that RA3 and RA2 are inputs, and RA4, RA1, and RA0 are outputs.
    TRISA = 0b01100; // Prefix “0b” means binary
    while(1) { // Loop forever
        PORTA = 0b00000; // Output 0 to RA1
        PORTA = 0b00010; // Output 1 to RA1
    }
}
```

PORTA is bidirectional which means its bits can be configured to be inputs or outputs. These bits can be configured in software by another variable TRISA. Setting a bit in TRISA to 0 will make the corresponding bit in PORTA an output. Setting the bit to 1 will make the bit an input.

The Blinking LED program sets RA1 to be an output. It also sets the other PORTA bits as follows: RA3 and RA2 to inputs, and RA4 and RA0 to outputs. (Note that the configuration of RA0, RA2-RA4 is irrelevant to this example since we do not use them.)

TRISA is implemented by a hardware register. Such registers are often called control registers. To the C compiler for the PIC, “TRISA” and “PORTA” are variables with special names.

The Blinking LED program first configures PORTA to inputs and outputs. Then it goes into an infinite loop, continually setting pin RA1 to the value 0 and then 1. There is a problem with this program. Since the processor is running so fast,
Gal Sasaki, Department of Electrical Engineering, University of Hawaii

the LED will blink very quickly, too fast for us humans to detect. We will just see an LED that seems continually on. This can be solved by adding delay as shown in the next program, “Blinking LED 2”.

```c
main( ) // BLINKING LED 2 program
{
    // Set PORTA so that RA3 and RA2 are inputs, and RA4, RA1, and RA0 are outputs.
    TRISA = 0b01100; // Prefix “0b” means binary TRISA = 0b01100;

    while(1) { // Loop forever
        PORTA = 0b00000; // Output 0 to RA1
        delay1sec(); // Delay 1 second
        PORTA = 0b00010; // Output 1 to RA1
        delay1sec(); // Delay 1 second
    }
}
```

The Blinking LED 2 program uses a function “delay1sec”, which provides a 1 second delay. Next, we will discuss implementing this function. The following implementation is naive approach. It assumes a 1 MHz clock. The function has a for-loop that goes through 1 million passes.

```c
void delay1sec( ) // A delay of 1 second
{
    int n;
    n = 0;
    while (n < 1000000) { // 1000000 = 1 million passes of the while loop
        n++;
    }
}
```

If each pass takes one clock cycle (1 us) then the for-loop will cause 1 second of delay. However, this function will not work for the following reasons:

- Each line of C will translate to multiple lines of machine instructions. It is difficult to estimate the number of instructions without compiling.
- Machine instructions can take multiple clock cycles, and some instructions will take different clock cycles depending on the situation.
- The variable “n” is stored in a memory cell (register), and this cell has a limited size. Operands are 1 byte and its value ranges from -128 to +127 (using two's complement arithmetic; if you don’t know two's complement, we will discuss this in lecture later). Thus, “n” cannot be implemented correctly in the for-loop, which requires it to range up to 1 million.
Next, we will implement the 1 second delay using two functions. The first function “delay1sec” calls a second function “delay1ms”, that has a 1 millisecond delay. The function “delay1sec” calls “delay1ms” for 1000 times.

The following is “delay1sec”. The C instruction “for” and calculations such as “n++” take clock cycles too but their contribution to percentage of clock cycles is small compared to “delay1ms”.

```c
void delay1sec( ) // A delay of 1 second
{ int n,m;
   for (n=0; n<10; n++) {
      for (m=0; m<100; m++) delay1ms( );
   }
}
```

Figure 6 has the “delay1ms” C function which also has assembly language instructions. The compiler for the PIC allows a program to have a mix of C language and assembly language. By writing most of the function in assembly language, we can control the number of cycles it takes.

Figure 6 shows how the assembly language portion is delimited. It also explains the “DELAY_LOOP” label. The assembly language portion is a delay loop. It is composed of the assembly language instructions CLRWDT, NOP, DECFSZ, and GOTO. (The PIC has 35 machine instructions, which are presented in Section 7 in the data sheet. Figure 7 shows some of these instructions.)
Next, we will explain the machine instructions.

**NOP:** The simplest instruction is NOP (for “no operation), which does nothing except to use up a clock cycle. Figure 7 shows that “NOP” is the assembly language mnemonic, and that the machine instruction is the 14-bit string 00 0000 0xx0 0000. The figure also shows that it uses 1 clock cycle.

**GOTO:** The instruction GOTO has the syntax GOTO <label> . The <label> indicates the position of another instruction. For the delay10us( ) function, the label DELAY LOOP corresponds to the CLRWDT instruction. When the PIC processor executes this GOTO instruction, it will jump to the DELAY LOOP label.
### Mnemonic, Operands | Description | Cycles | 14-Bit Opcode | Status Affected | Notes |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDWF f, d</td>
<td>Add W and f</td>
<td>1</td>
<td>00 0111 dfff ffff</td>
<td>C,DC,Z</td>
<td>1,2</td>
</tr>
<tr>
<td>ANDWF f, d</td>
<td>AND W with f</td>
<td>1</td>
<td>00 0101 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>CLRF f</td>
<td>Clear f</td>
<td>1</td>
<td>00 0001 1fff ffff</td>
<td>Z</td>
<td>2</td>
</tr>
<tr>
<td>CLRW -</td>
<td>Clear W</td>
<td>1</td>
<td>00 0001 xxxx xxxx</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>COMF f, d</td>
<td>Complement f</td>
<td>1</td>
<td>00 1001 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>DECF f, d</td>
<td>Decrement f</td>
<td>1</td>
<td>00 0111 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>DECFSZ f, d</td>
<td>Decrement f, Skip if 0</td>
<td>1 (2)</td>
<td>00 1010 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>INCF f, d</td>
<td>Increment f</td>
<td>1</td>
<td>00 1100 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>INCFSZ f, d</td>
<td>Increment f, Skip if 0</td>
<td>1 (2)</td>
<td>00 1111 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>IORWF f, d</td>
<td>Inclusive OR W with f</td>
<td>1</td>
<td>00 0100 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>MOV f, d</td>
<td>Move f</td>
<td>1</td>
<td>00 1000 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>MOVWF f</td>
<td>Move W to f</td>
<td>1</td>
<td>00 0000 1fff ffff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP -</td>
<td>No Operation</td>
<td>1</td>
<td>00 0000 0000 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLF f, d</td>
<td>Rotate Left f through Carry</td>
<td>1</td>
<td>00 1101 dfff ffff</td>
<td>C</td>
<td>1,2</td>
</tr>
<tr>
<td>RRF f, d</td>
<td>Rotate Right f through Carry</td>
<td>1</td>
<td>00 1100 dfff ffff</td>
<td>C</td>
<td>1,2</td>
</tr>
<tr>
<td>SUBWF f, d</td>
<td>Subtract W from f</td>
<td>1</td>
<td>00 0010 dfff ffff</td>
<td>C, DC, Z</td>
<td>1,2</td>
</tr>
<tr>
<td>SWAPF f, d</td>
<td>Swap nibbles in f</td>
<td>1</td>
<td>00 1110 dfff ffff</td>
<td></td>
<td>1,2</td>
</tr>
<tr>
<td>XORWF f, d</td>
<td>Exclusive OR W with f</td>
<td>1</td>
<td>00 0110 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
</tbody>
</table>

**Figure 7.** Some of the PIC instructions.

**CLRWDT:** The instruction CLRWDT clears the *Watch Dog Timer*. The timer is a module within the PIC. Watch Dog Timers are useful for computers that need to keep running forever. These timers basically run on their own, keeping track of how much time has elapsed. When these timers have elapsed for a pre-specified period, they *time out*. This causes the computer to reset.

Watch dog timers are useful for computers that should be running all the time. Typically, these computers will run a single program that continually does a particular task. A simple example is a computer and its program controlling a traffic light. Note that such a computer is expected to run on its own without any human to monitor it. Now if the computer gets stuck for some reason, there is no one to reset it. The computer could get stuck because the program may have bugs, or there may be a “soft error”, which is a temporary hardware error.

To avoid getting stuck forever, a computer can use Watch Dog Timer to automatically reset it. Then the program must be written so that it periodically resets the timer to keep from timing out and resetting prematurely. This is sometimes referred to as “petting the dog”. The analogy is that you must regularly pet the dog to keep it happy. Our delay function will pet the dog every pass through the loop. Here is more information about the Watch Dog Timer:

http://en.wikipedia.org/wiki/Watchdog_timer

A related machine instruction of CLRWDT is CLRF shown in Figure 6. Here, “F” (or “f”) is a variable. which is realized by a hardware register. This instruction clears the variable F. CLRF has the machine instruction 00 0001 1fff ffff, where “fff ffff” is a binary number that indicates the register F. CLRWDT is a separate function because the watch dog timer is special hardware.
DECFSZ: The final machine instruction in the delay loop is DECFSZ _delay,1. Figure 8 has a description of DECFSZ f,d, where “f” is a variable (which holds data or operand), and “d” is a value which is either 0 or 1. In our case, “f” is the global variable “delay”. Note that in assembly language the C variable “delay” is “_delay”. In our case, “d” is equal to 1.

According to Figure 7, when the processor executes “DECFSZ _delay,1”, it first decrements the variable “delay” by 1. Since “d” has been set to 1, the decremented value is stored back into the global variable “delay”. Next, the processor checks the result. If the result is 0 (i.e., global variable “delay” is zero) then the next instruction (GOTO) is skipped. Otherwise, the processor proceeds to execute the next instruction.

<table>
<thead>
<tr>
<th>DECFSZ</th>
<th>Decrement f, Skip if 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>[label] DECFSZ f,d</td>
</tr>
<tr>
<td>Operands:</td>
<td>0 ≤ f ≤ 127</td>
</tr>
<tr>
<td></td>
<td>d ∈ {0,1}</td>
</tr>
<tr>
<td>Operation:</td>
<td>(f) - 1 → (destination);</td>
</tr>
<tr>
<td></td>
<td>skip if result = 0</td>
</tr>
<tr>
<td>Status Affected:</td>
<td>None</td>
</tr>
<tr>
<td>Description:</td>
<td>The contents of register ‘f’ are decremented. If ‘d’ is 0, the result is placed in the V register. If ‘d’ is 1, the result is placed back in register ‘f’. If the result is 1, the next instruction is executed. If the result is 0, then a NOP is executed instead, making it a 2Tcy instruction.</td>
</tr>
</tbody>
</table>

*Figure 8.* The description of DECFSZ from the data sheet.

Figure 9 shows the function “delay1ms” again with an explanation of the number of clock cycles it uses. It has the following components:

- The C instruction “delay = 100” initializes the variable “delay”.
- The assembly language portion is a delay loop. Each time the delay loop is passed, the instruction DECFSZ decrements “delay”. Thus, the loop goes through 100 passes.

Figure 7 shows the number of cycles per instruction.
Let us consider the delay in Figure 9. Each pass will execute CLRWD T once, NOP six times, DECFSZ once, and GOTO once (though GOTO is skipped on the 100th pass). From Figure 7, CLRWD T, NOP, and GOTO each take one clock cycle. DECFSZ takes one cycle for the first 99 passes but 2 cycles on the last pass.

Each pass uses 10 clock cycles. More specifically, for the first 99 passes, each pass is 10 clock cycles, composed of CLRWD T (1 cycle), NOP (6 cycles), DECFSZ (1 cycle), and GOTO (1 cycle). The 100th pass also uses 10 clock cycles, composed of CLRWD T (1 cycle), NOP (6 cycles), and DECFSZ (2 cycles).

Since there are 100 passes, the total number of clock cycles in the delay loop is (10 cycles per pass) x (1000 cycles) = 1000 clock cycles, which is a 1 millisecond delay.

This accounts for most of the clock cycles but not all. There are clock cycles to execute “delay = 100” and clock cycles to implement housekeeping operations for the function, e.g., returning from the function call. But the number of these cycles is small compared to 1000 clock cycles of the delay loop. Thus, we ignore them and assume that the function has a delay of “approximately” 1 ms.

Note that the number of NOPs in the function was determined as part of its design. The number of NOPS used is just enough so that the function has a 1 ms delay. If we wanted more or less delay we would add or delete NOPs.
Bit-Wise Logic Operations

Let us review logic operations. Recall that there are three basic logic operations: AND, OR, and COMPLEMENT (sometimes called NOT). Their truth tables are shown in Figure 10.

![Truth Tables](image)

Figure 10. Logic operation truth tables.

The C language has bit-wise logic operators:

- "&" (AND)
- "|" (OR)
- "~" (COMPLEMENT)

To understand what these operators do, let us discuss an example. Consider the C language instruction

```c
m = n & 6;
```

and assume that all the operands are 8 bits. Let the 8 bits of m be m7, m6, ..., m0; and the 8 bits of n be n7, n6, ..., n0. Also note that the decimal value 6 is 00000110 in binary. The bit-wise AND operation "&" will do eight AND operations, and in particular,

```c
m7 = n7 AND 0
m6 = n6 AND 0
m5 = n5 AND 0
m4 = n4 AND 0
m3 = n3 AND 0
m2 = n2 AND 1
m1 = n1 AND 1
m0 = n0 AND 0
```

Similarly,

```c
m = n | 6;
```

results in
m7 = n7 OR 0
m6 = n6 OR 0
m5 = n5 OR 0
m4 = n4 OR 0
m3 = n3 OR 0
m2 = n2 OR 1
m1 = n1 OR 1
m0 = n0 OR 0

The bit-wise complement operation “~” will just flip all the bits, e.g., \( m = \sim n \) will flip the 8 bits in \( n \) and store them in \( m \).

The logic operations can be very useful in single chip computers like the PIC. As we saw earlier, the PIC has certain variables such as PORTA and TRISA (which are hardware registers in the PIC) that need to be set to certain values. In many cases, we would like to set one bit to a certain value and leave the rest unchanged. These bit-wise logic operations can help us.

To illustrate this, let review the operation \( m = n \& 6 \), which is really

m7 = n7 AND 0
m6 = n6 AND 0
m5 = n5 AND 0
m4 = n4 AND 0
m3 = n3 AND 0
m2 = n2 AND 1
m1 = n1 AND 1
m0 = n0 AND 0

Recall that if you AND a value \( x \) with 0 then the result is 0; and if you AND the value \( x \) with 1 then the result is \( x \). You can check this with the truth table of the AND operation in Figure 10. Figure 11 summarizes this as a function table for the AND (also shown is the function table for OR). It assumes that the two inputs of the AND is a variable \( x \) and an input called the “mask”. If the mask = 0 then the output is 0, and if the mask = 1 then the output is \( x \).

<table>
<thead>
<tr>
<th>AND</th>
<th>mask</th>
<th>input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>mask</th>
<th>input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 11.** Function tables of the AND and OR assuming inputs “\( x \)” and “mask”.

In the example \( m = n \& 6 \), the bit pattern 00000110 is a mask, where 0s result in 0s, and 1s result in unchanged values. Thus, bits 7, 6,..., 3 and 0 are cleared to 0, while bits 2 and 1 remain unchanged, i.e.,
Thus, the bit-wise AND operation “ & “ can be used to clear bits. Similarly, the bit-wise OR operation “ | “ can be used to set bits to 1. Here, 0s result in unchanged values, while 1s result in bits being set to 1. This corresponds to the OR function table in Figure 11.

Some other useful C language operators are the bit shifting operators “<<” and “>>”, which are left and right shifts respectively. Let us go over the left shift operation first. Consider the C language instruction

\[ m = n << 2 \]

This means that the value of \( n \) (\( = n_7, n_6, ..., n_0 \)) should be shifted to the left by 2 bit positions. This results in

\[
\begin{align*}
m_7 &= n_5 \\
m_6 &= n_4 \\
m_5 &= n_3 \\
m_4 &= n_2 \\
m_3 &= n_1 \\
m_2 &= n_0 \\
m_1 &= 0 \\
m_0 &= 0
\end{align*}
\]

Notice that the bits of \( n \) were shifted to the left by 2, and 0s were shifted into the lower bits \( m_1 \) and \( m_0 \). Another example is \( m = n << 6 \), which results in

\[
\begin{align*}
m_7 &= n_1 \\
m_6 &= n_0 \\
m_5 &= 0 \\
m_4 &= 0 \\
m_3 &= 0 \\
m_2 &= 0 \\
m_1 &= 0 \\
m_0 &= 0
\end{align*}
\]

The right shift “>>” works in a similar way except the bits are shifted to the right.

Now as a review, we will give some examples that combine all of the operations above. Consider the C language instructions
d = 1 << 3;
m = m | d;

The first line “d = 1 << 3” will shift the value “1” to the left by 3 bit positions. Recall that “1” in binary is 00000001. Shifting this to the left by 3 bit positions leaves 00001000. Thus, d = 00001000. This is also our mask.

The second line “m = m | d” will bit-wise OR the mask with variable m. Since the mask has value 1 at bit position 3, m3 will be set to 1, while the rest of variable m will be unchanged. We can rewrite this more compactly by the single line

\[ m = m | (1 << 3); \] or alternatively \[ m |= (1 << 3); \]

Note that these lines will set m3 to 1, and leave the rest of variable m unchanged. If instead we wanted to set m6 to 1, we could write

\[ m |= (1 << 6); \]

Now we will show how the bit-wise AND operation can be used to clear bits. Consider the C language instructions

d = 1 << 3;
d = ~d;
m = m & d;

The first line “d = 1 << 3” will leave d = 00001000. The second line “d = ~d” will leave d = 11110111. This is our mask. The third line “m = m & d” will bit-wise AND the mask with variable m. Since the mask has value 0 at bit position 3, m3 will be cleared to 0, while the rest of variable m will be unchanged. We can rewrite this more compactly by the single line

\[ m = m & (~1 << 3); \] or alternatively \[ m &= (~1 << 3); \]

Note that these lines will clear m3 to 0, and leaving the rest of variable m unchanged. If instead we wanted to clear m6, we could write

\[ m &= (~1 << 6); \]

Finally, note that we use the term “clear” to mean setting a value to 0, and the term “set” to mean setting a value to 1. Though this is not a standard terminology, it is used in technical documents frequently. Sometimes documents will just state “set” and assume you know that it means “set to the value 1”, and “clear” to mean “clear to the value 0”. This does not happen all the time but be aware of it.