Galen Sasaki, Department of Electrical Engineering, University of Hawaii

Explanation of PIC 16F648A processor data sheet -- Part 2: More on the PIC

This is the second of the three part overview of the PIC 16F648A processor. We will first discuss the timer module and prescalar. Then we will discuss further about the PIC architecture (especially the *instruction processing cycle*), and finally give another tutorial about MPLAB and breakpoints.

Timer Module and Prescalar

Many applications of the PIC require that the processor fulfill timing requirements. For example, in the testlab1.c example from Lab 2.1, the program required a quarter-second delay function. This function was implemented by having it call another function, which had a 1ms delay, which was written so that it uses a certain number clock cycles given the clock rate. The problem with this approach is that the programmer must carefully count the number of machine instructions.

An alternative approach is to use a timer circuit which indicates how much time has elapsed. The PIC has three timer circuits. We will focus our attention on one of these circuits called TIMER0 (you can find the description of TIMER0 in the PIC documentation on page 47). It is an 8-bit timer/counter, and in many cases can be accessed like a variable. It can be configured so that it will increment every clock cycle. Next is a naive example implementation of using TIMER0. It is a delay function of 100 clock cycles. However, there are problems with this implementation as we shall see. Also note that TIMER0 can be accessed using the C language with the variable name TMR0.

```
void delayNaive( ) // A delay of (approximately) 100 clock cycles
{
TMR0 = 0;  // Clears the TIMER0 circuit
// From this point, the TIMER0 circuit runs on its own.
// It increments with each clock cycle.
while (TMR0 < 100); // This checks when 100 clock cycles have elapsed
}
```

If we wanted to increase the delay to more clock cycles, e.g., 2000 clock cycles, we cannot do it because TIMER0 is limited to 8-bits, and therefore limited to a maximum value of 255.

To get a longer delay, we can use the *prescalar*. The *prescaler* slows TIMER0 down by reducing the clock signal rate to TIMER0. The options of rates are 1:2, 1:4, 1:8,...., or 1:256. Note that the step-down values are all powers of two.

The next delay function “delayNaive2” (see below) has a delay of approximately 10,000 clock cycles. First, we configure the prescalar to 1:256. Thus, the timer will increment every 256 clock cycles. The function will terminate when TIMER0 = 40, i.e., after 40 x 256 = 10,240 clock cycles. Thus, “delayNaive2” has a delay of approximately 10,000 clock cycles.

```
#include <htc.c>

void delayNaive2();

main() {
.
.
TOCS = 0;
PSA = 0;
PS2 = 1;
PS1 = 1;
PS0 = 1;
.
.  The rest of the program include function-calls to delayNaive2()
.
}


void delayNaive2( ) // A delay of (approximately) 10000 clock cycles
{
TMR0 = 0;
while (TMR0 < 40);
}
```

The main( ) function initializes the following bits:

- TOCS (bit 5): TOCS determines if TIMER0 is in Timer or Counter Mode, where TOCS = 0 means Timer Mode.
- PSA (bit 3): The prescalar can be assigned to TIMER0 or the watch dog timer. Clearing PSA to 0 will assign the prescalar to TIMER0. PSA = 1 assigns TIMER0 to the watch dog timer. We want PSA = 0 for our delay function.
- PS2-PS0 (bits 2-0): Determines the step down rate of the prescalar. PS2-PS0 = 111 means 1:256, while PS2-PS0 = 000 means 1:2.

These bits are part of the OPTION register, which a programmer can use to configure the PIC. See Figure 1 for a portion of a table in the PIC documentation describing the register. Figure 2 has a more detailed description of the OPTION register from the documentation.

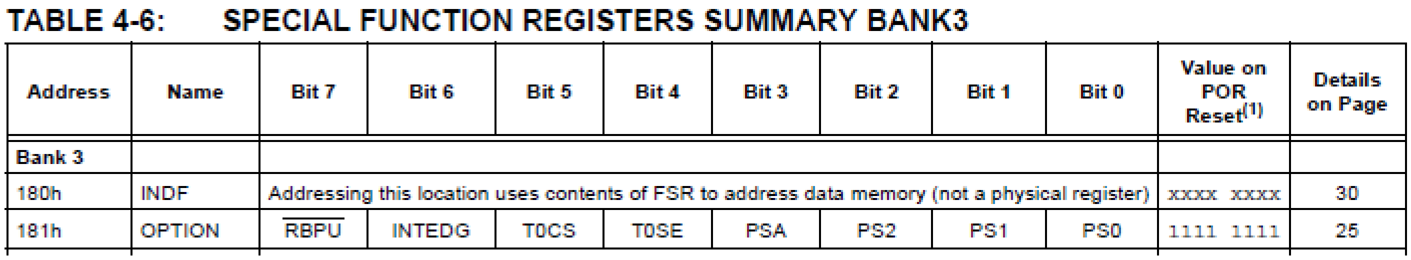
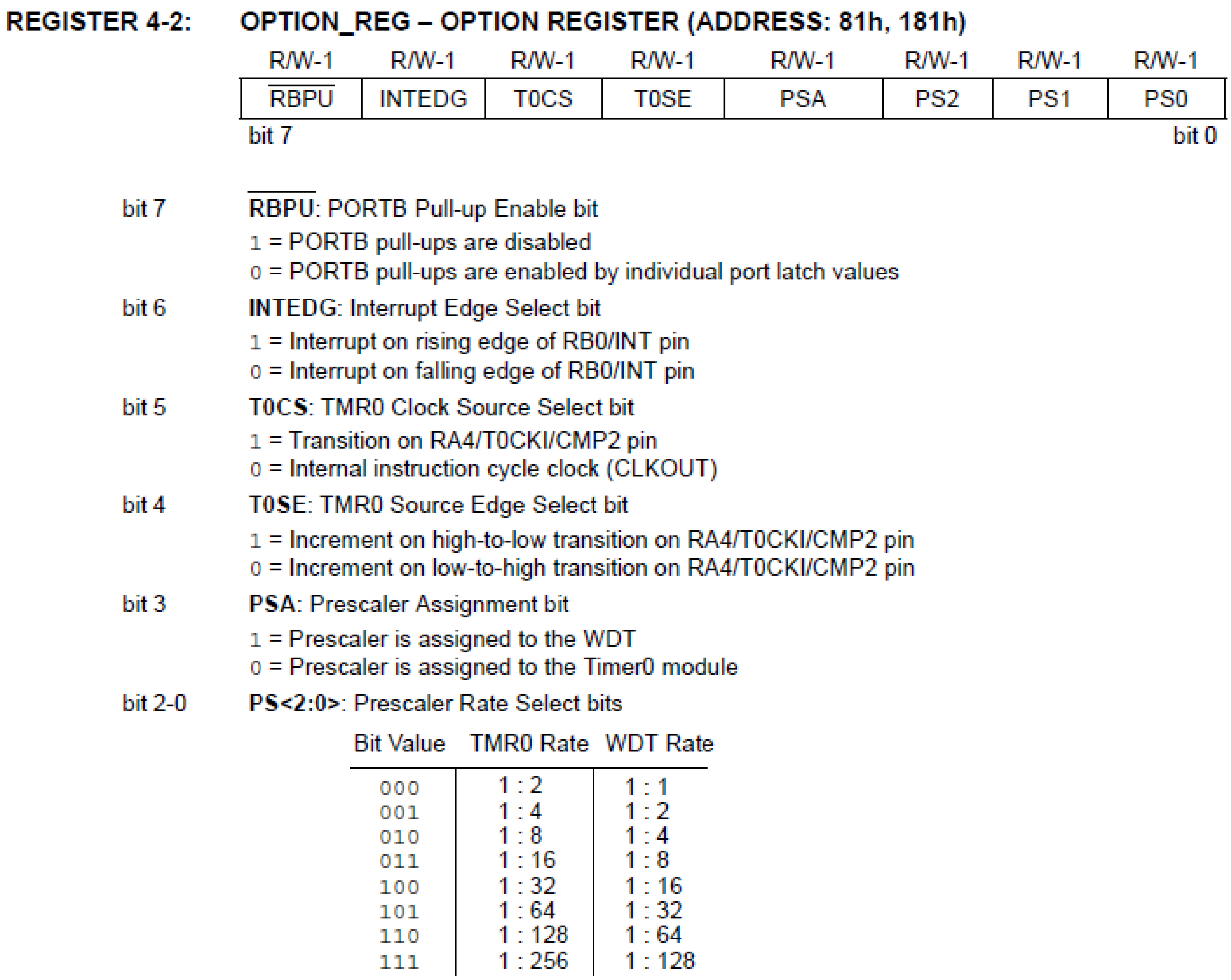


TABLE 4-6: SPECIAL FUNCTION REGISTERS SUMMARY BANK3

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR Reset[1]	Details on Page
Bank 3											
180h	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								xxxx xxxx	30
181h	OPTION	$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	25

Figure 1. Table 4-6 from the data sheet for the PIC on page 23.



REGISTER 4-2: OPTION_REG – OPTION REGISTER (ADDRESS: 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI/CMP2 pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI/CMP2 pin
0 = Increment on low-to-high transition on RA4/T0CKI/CMP2 pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS<2:0>**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Figure 2. From page 25 of the PIC documentation, a description of the OPTION register. Legend: W = writable, R = readable, U = unimplemented, x = unknown.

The other bits of the OPTION register are irrelevant for our purposes. (For example, TOSE (bit 4) is used when TIMER0 is in Counter Mode, but we use Timer Mode.)

Note that the default value of the bits is 1 – see "Value on POR Reset" in Figure 1. Thus, PS2, PS1, and PS0 are already set to 1, and our setting them in the program is redundant. We do it anyway to be safe.

The following is an example of a modification of testlab1.c using TIMER0. There are no assembly language instructions.

```
/* This will blink an LED on-and-off */

#include <htc.h>                       // Header file for PIC processor library files
void delay_10us(unsigned char t);      // t*10us delay
void delay_ms(long t);                 // t ms delay

main(void)
{
PSA = 0;        // Use TIMER0 rather than Watch Dog Timer
TOCS = 0;       // TIMER0 in timer mode rather than counter mode
PS2 = 1; PS1 = 1; PS0 = 1; // Prescale rate of 256:1

TRISB = 0x00; /* All ports of PORTB are outputs */

while(1) { /* Turn RB0 and RB1 on-and-off */
        RB0 = 0;
        delay_ms(250); // Delay of a quarter-second
        RB0 = 1;
        delay_ms(250); // Delay of a quarter-second
} /* End while-loop */
} /* End main */

void delay_ms(long t) // delays t ms (approximately) assuming 1 instruction per clock cycle
{
long k;
for (k=0;  k<t;  k++) {
        TMR0 = 0;
        while(TMR0 < 4);  /* This while-loop waits until TMR0 = 4, since 4 x 256 is about 1000 */
        }
}
```

Now we discuss some details. First notice the difference in the delay loops of Lab 2.1 and the one above. In Lab 2.1, the delay loop was a loop of instructions. The clock cycles of these instructions had to be counted to ensure that the loop provided the exact delay. The delay loop above is done by the TIMER0 circuit. The program "polls" the TIMER0 to determine its value and whether the delay has expired.

T0IF bit. This is a "flag" that indicates when the TIMER0 circuit has an "overflow", which means that the value has become so large that it can't fit into the TIMER0. In particular, note that TIMER0 has values from 0 through 255 (which is 0xFF in hexadecimal). When TIMER0 reaches the value 255, it cannot get any larger. So it "wraps around" and goes back to zero. Whenever this wrap around occurs, the T0IF flag is set to 1. You can manually clear the T0IF flag with the C language instruction "T0IF = 0;". An alternative to the delay_ms function above is the following:

```
void delay_ms(long t) // delays t ms (approximately) assuming 1 instruction per clock cycle
{
long k;
for (k=0;  k<t;  k++) {
        TMR0 = 256 – 4;         // Initialize TIMER0 so that it increments 4 times before wraparound
        T0IF = 0;               // Clear T0IF flag
        while(T0IF == 0);       // Wait until T0IF flag becomes 1
        }
}
```

This is an indirect way to implement the delay. However, it is an example of using the T0IF flag. The flag can also be used with "interrupts" but we have not covered interrupts yet – we will in the next lab.

Finally, note that the T0IF flag is bit 2 of the INTCON register.

Instruction Cycle. There is a *clock cycle* and an *instruction cycle*. The clock cycle comes from the clock signal, such as the crystal or internal clock circuitry. The instruction cycle is made up of four clock cycles. The clock cycles of an instruction cycle are labeled Q1, Q2, Q3, and Q4. Each instruction typically takes two instruction cycles to execute: the first is fetching the instruction from memory, and the second is the actual execution of the instruction. The PIC has a pipelined architecture, so typically a new instruction is fetched per instruction cycle. That means that the rate at which instructions are being executed is typically one instruction per instruction cycle. The relationship between clock cycle and instruction cycle is explained on page 15 of the data sheet of the PIC. Figure 3 shows a timing diagram relating the clock cycles and instruction cycles.

FIGURE 3-2: CLOCK/INSTRUCTION CYCLE

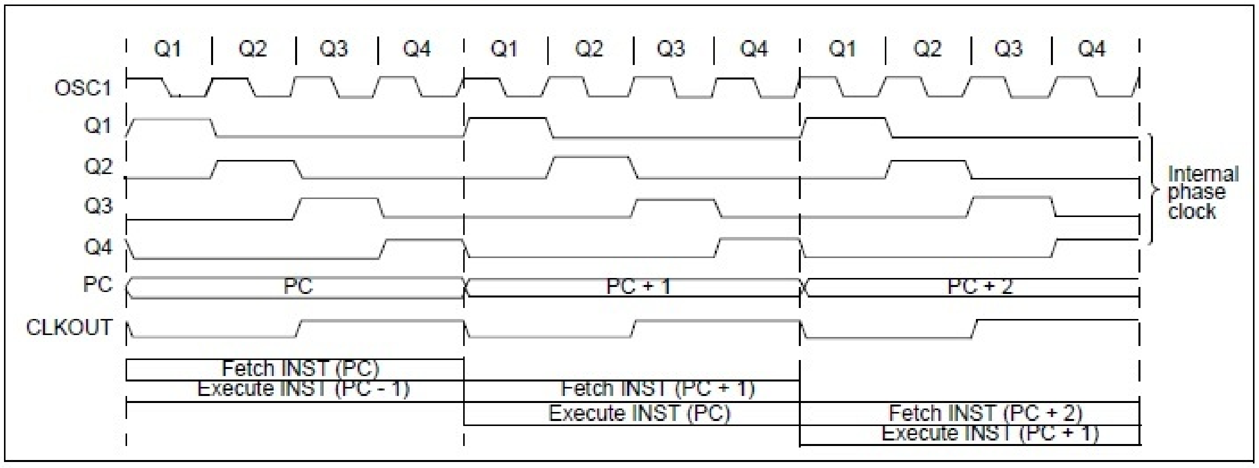


Figure 3. Clock cycle and instruction cycle relationship from page 15 of the data sheet.

TIMER0, without the prescalar, will increment on every instruction cycle. Of course, with the prescalar, TIMER0 will reduce the rate of incrementing according to the chosen prescalar value, e.g., 1:16.

In this lab, we use a 4 MHz external crystal oscillator. Thus, the rate of the instruction cycles is 1 MHz.

Polling. The following is another way to implement delay_ms( ).

```
void delay_ms(long t) // delays t ms (approximately) assuming 1 instruction per clock cycle
{
long k=0;
TMR0 = 256-4;
T0IF = 0;

while(1) {

  if (T0IF) {  // 1 ms has elapsed
    k++;      // Keep track of number of elapsed milliseconds
    TMR0 = 256 – 4;  // Reset TIMER0
    T0IF = 0
    }

  if (k >= t) return;  // If t ms has elapsed then stop the delay
}
}
```

Note that the while-loop keeps checking if T0IF became "1" because this means that 1 ms has elapsed. Regularly checking a variable (or some other device) is known as "polling". Also note that both if-statements are run during every pass of the while-loop. However, they actually do something only occasionally.

OPTION Register. In the previous examples, bits in the OPTION register had their own names, e.g., PSA, TOCS, PS2, PS1, and PS0. These names can be used as variables and the corresponding bits can be set by using an assignment instruction, e.g., PS2 = 0.

The OPTION register has its own variable name, called OPTION_REG. So the entire register can have its value initialized with one assignment instruction, e.g., OPTION_REG = `0b00000111` (here, "0b" is a prefix for a binary number). We can clear the bits of PSA and TOCS in one instruction, and set the bits of PS2, PS1, and PS0 in another instruction (in the example below, "0b" is the prefix for a binary number):

```
// In the following instruction, we clear TOCS (bit 5) and PSA (bit 3).
// Note that bits 5 and 3 are ANDed to "0", which will clear them; while
// the other bits are ANDed to "1", which leaves them alone.
n = OPTION_REG & 0b11010111;

// The following will set the prescalar to 256:1 by setting the bits
// PS2 (bit 2), PS1 (bit 1), and PS0 (bit 0) to "1".  This is done by
// ORing the bits to "1".  The other bits are left alone by ORing them to "0".
OPTION_REG = n | 0b00000111;
```

Hardware of Prescalar. Figure 3 shows the schematic diagram of the TIMER0 module and prescalar. FOSC/4 is the instruction clock of the PIC. Notice how it is routed through

- Two multiplexers (controlled by PSA and TOCS)
- The prescalar (which is an 8-bit counter). This counter has 8 outputs, each generates a periodic signal. We will discuss this output shortly.
- One of the counter's outputs is selected by an 8:1 multiplexer (controlled by PS2, PS1, PS0), and that signal is routed to the TIMER0 register.

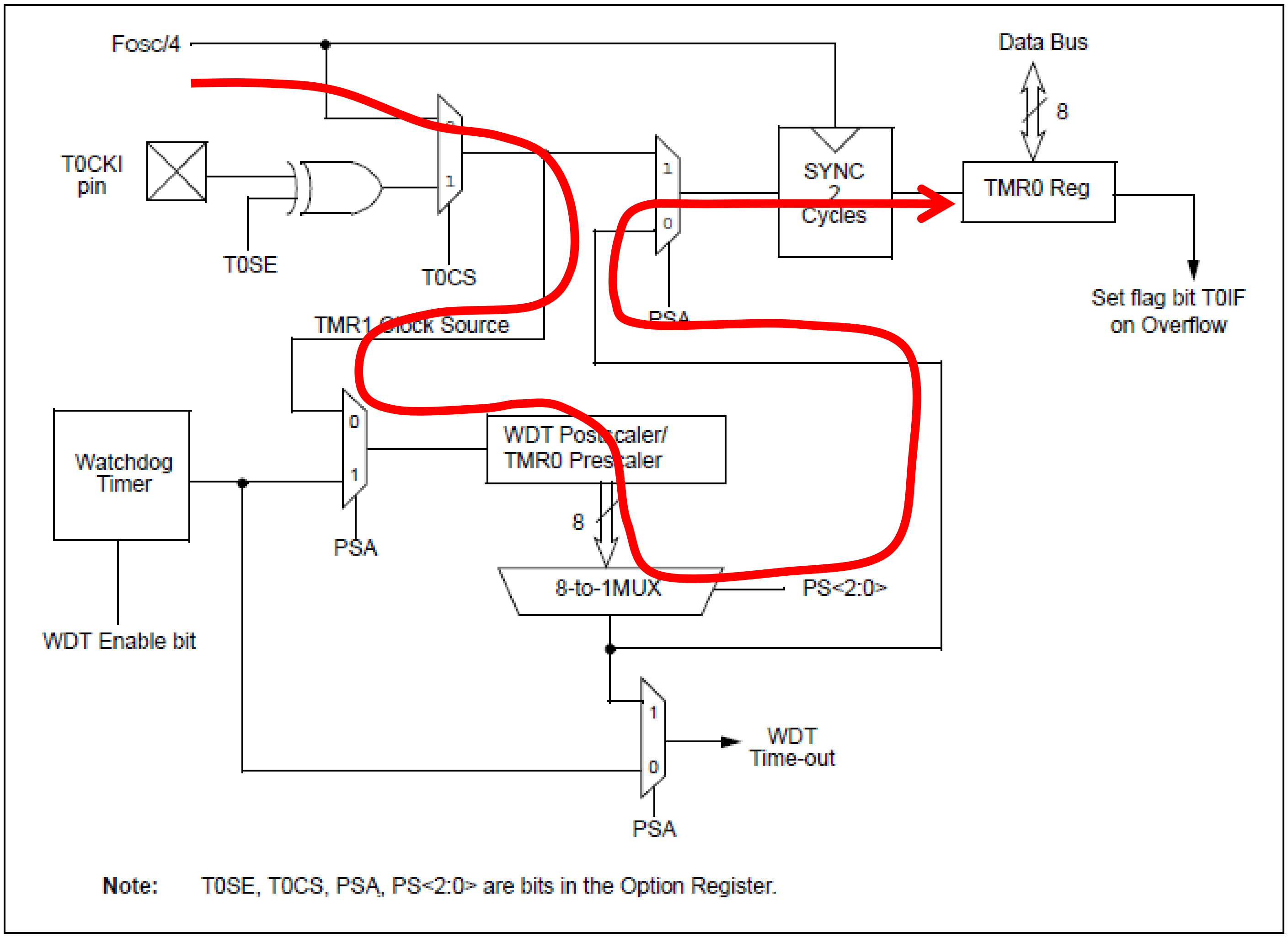


Figure 3. TIMER0 and prescaler circuitry from Figure 6-1 on page 48 of the datasheet of the PIC.

To understand the output of the prescalar counter, consider its values with each instruction cycle as shown in the next table.

Cycle	Prescalar Output	Bit 1	Bit 0
0	00000000	0	0
1	00000001	0	1
2	00000010	1	0
3	00000011	1	1
4	00000100	0	0
5	00000101	0	1
6	00000110	1	0
7	00000111	1	1

The table has the value of the prescalar output with each instruction cycle. It also shows Bits 1 and 0 of the output in isolation. Notice that Bit 0 is a periodic signal with a period of 2 cycles, and Bit 1 is a periodic signal with a period of 4 cycles. Thus, Bits 0 and 1 have step-down rates of 1:2 and 1:4, respectively. The prescalar is also known as a "step down counter" since it provides clock signals at a slower rate.

Here's a couple of tips about choosing the step-down rate. The first tip is in regards to our delay_ms example above. Recall that PS2, PS1, and PS0 were chosen so that the step-down rate was 1:256. This means that the T0IF flag goes to "1" every 256 instruction cycles. Whenever, T0IF goes to "1", let's call that a "tick" of the clock. Actually, all of these "ticks" correspond to 256 instruction cycles, except possibly the first. The first tick may be less, and perhaps much less, than 256 instruction cycles. That's because when the CPU jumps into the program that checks T0IF, it may at a time during the middle rather than at the beginning of a tick.

The delay_ms is basically a delay loop that counts 4 ticks. However, the first may be less than a full tick. Since there are only four ticks in the delay loop, the inaccuracy of the first tick may lead to significant error in the overall delay.

The second tip is a possible problem if the step-down rate is too small, e.g., 1:2. Then that segment of the program that checks for ticks may be too long compared to a tick. In other words, the program segment is too slow to detect each tick.

Instruction Processing Cycle

Like all computers, the PIC follows an *instruction processing cycl*e, sometimes known as the *Von Neumann processing cycle* (named after a famous mathematician and computer scientist John Von Neumann). This cycle is basically what computers do continually. Figure 4 shows this cycle. It is a loop of:

1. Fetch instruction (from program memory).
2. Decode instruction (so that the CPU understands the instruction and what's to be done).
3. Execute instruction.

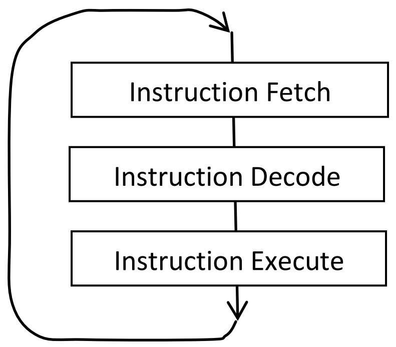
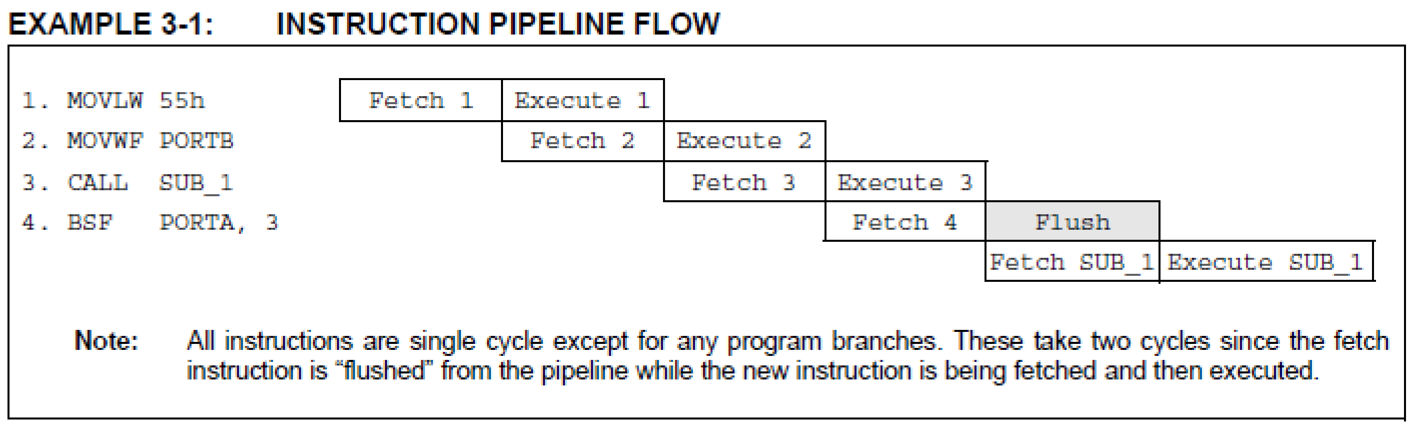


Figure 4. Instruction processing cycle.

The PIC is organized so that the Instruction Fetch and Decode is done in one instruction cycle, and Execute is done in another instruction cycle. Fetch/Decode and Execute can be pipelined as shown in Figure 5.


EXAMPLE 3-1: INSTRUCTION PIPELINE FLOW

```
1. MOVLW 55h
2. MOVWF PORTB
3. CALL  SUB_1
4. BSF   PORTA, 3
```

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

Figure 5. The pipeline in the PIC from page 15 of the data sheet.

Figure 6 has the portion of the block diagram of the PIC that is involved with fetch, decode, and execute instruction. To fetch the instruction, the CPU will use the Program Counter, which points to the next instruction to execute. The fetched instruction is stored in the Instruction Register. The output of the Instruction Register is connected to the circuitry called Instruction Decode and Control. This circuitry will first decode the instruction so that the CPU understands what the instruction is. Then it will send control signals to the datapath of the computer to execute the instruction (note that the control lines are not shown, probably because it would make the diagram very messy to look at).

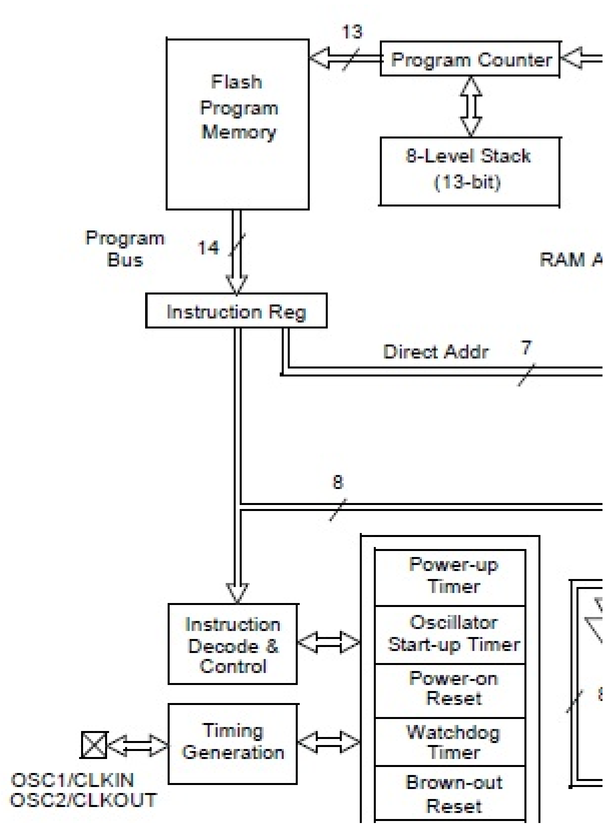


Figure 6. Instruction processing hardware of the PIC from page 12 of the data sheet.

Tutorial on Breakpoints for MPLAB

In Lab 2.1, we were introduced to MPLAB and its simulator. (By the way, MPLAB is available in the HP Labs in POST second floor, POST 208.)

To run the simulator, we used single stepping. Another technique is to use *breakpoints*. A breakpoint is a location in your program, where whenever the processor is at that location, the simulator halts. At that point, values in registers, memory, flags can be checked and discover any bugs.

In this tutorial, we will use breakpoints. Included with this tutorial is a simple program called testlab2.c.

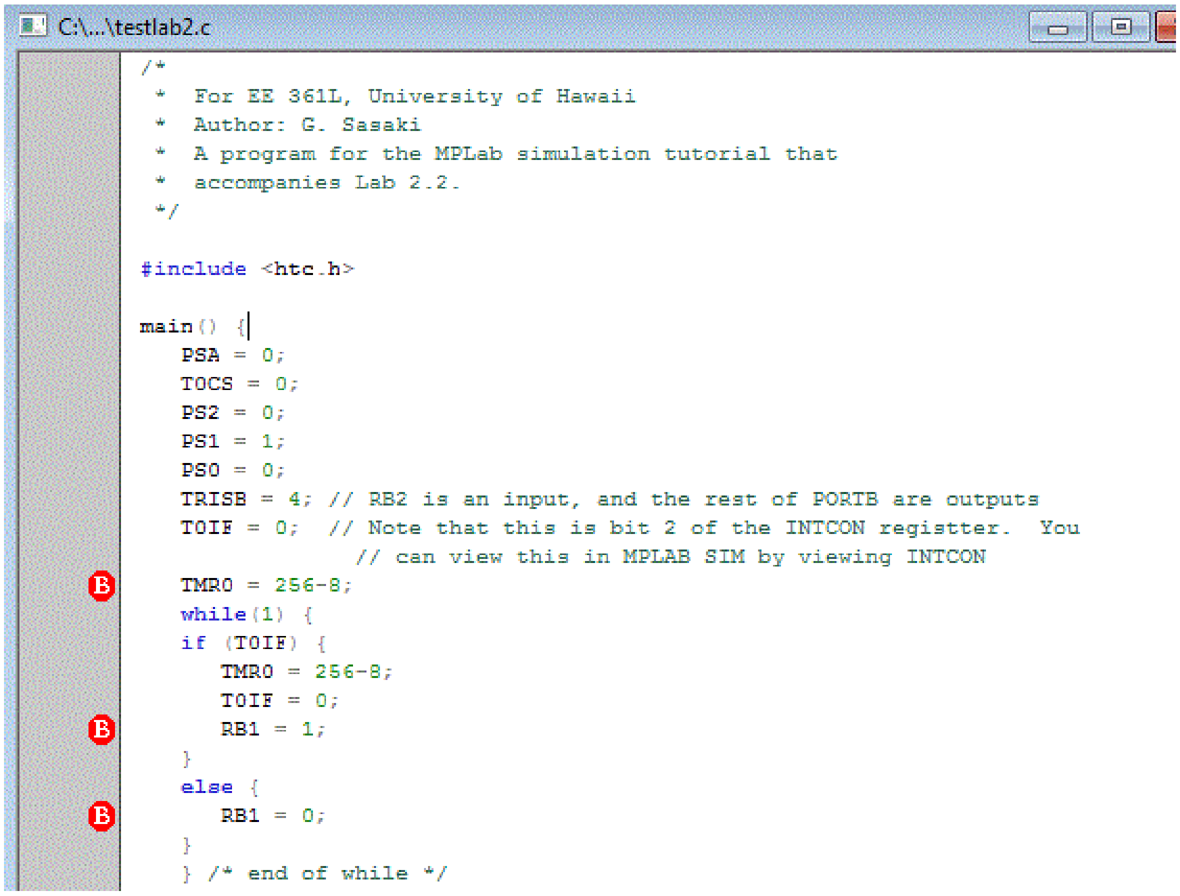
Create a project for this program, and load the program. Assume the same configurations as in Lab 2.1.

Set a breakpoint by moving the cursor to a location where you want to set a breakpoint. Then right-click, and select "Set Breakpoint". Set the three breakpoints as shown in Figure 7. The breakpoints are indicated by the red-colored "B". Then whenever the processor gets to a breakpoint, the simulation will halt. You can resume execution by Running the simulation.

Check all the breakpoints are enabled by selecting the Debugger menu and the Breakpoint option.

Build (or rebuild) the project. Reset and then run. If you get an error, you probably didn't set the breakpoint at a proper location. Choose a line that has an instruction. Other locations may work too.

Keep running the simulation until it goes through the if-statement a few times. You can observe how RB1 changes in the Simulator Logic Analyzer (from the View menu) as shown in Figure 8. (Don't forget to select RB1 as a channel in your Logic Analyzer.) You can keep track of how much time has elapsed with the Stop Watch under the Debugger menu.



```
/*
 *  For EE 361L, University of Hawaii
 *  Author: G. Sasaki
 *  A program for the MPLab simulation tutorial that
 *  accompanies Lab 2.2.
 */

#include <htc.h>

main() {
   PSA = 0;
   TOCS = 0;
   PS2 = 0;
   PS1 = 1;
   PS0 = 0;
   TRISB = 4; // RB2 is an input, and the rest of PORTB are outputs
   TOIF = 0;  // Note that this is bit 2 of the INTCON registter.  You
                 // can view this in MPLAB SIM by viewing INTCON
   TMR0 = 256-8;
   while(1) {
   if (TOIF) {
      TMR0 = 256-8;
      TOIF = 0;
      RB1 = 1;
   }
   else {
      RB1 = 0;
   }
   } /* end of while */
```

Figure 7. Breakpoints in MPLAB Debugger.

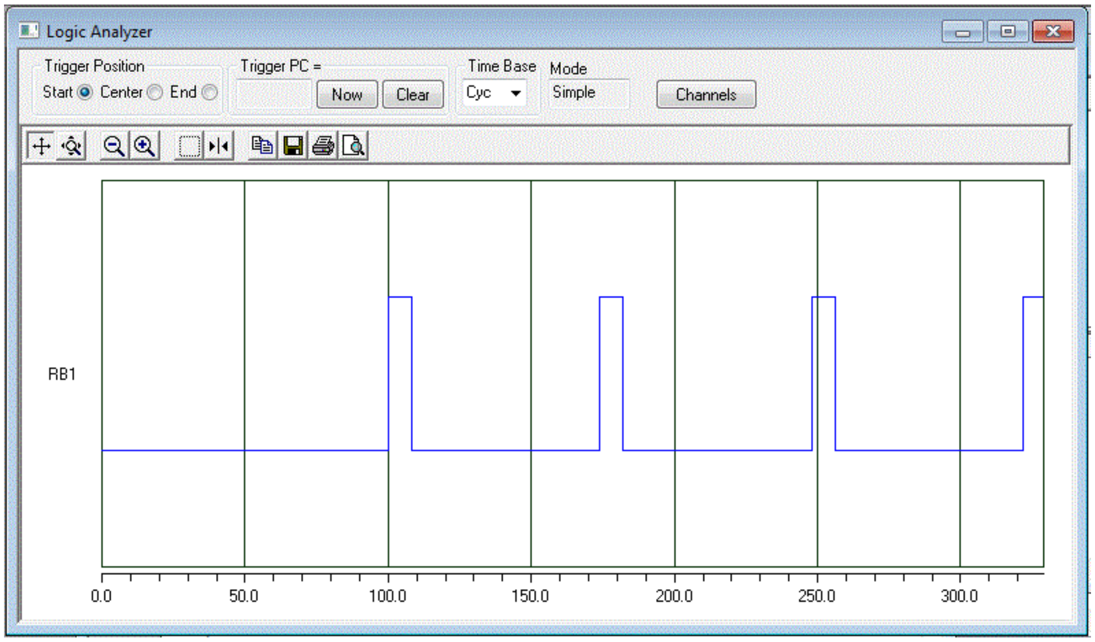


Figure 8. Logic Analyzer.