In lesson 1, we made GP1 high, and left it that way. To make it flash, we need to set it high, then low, and then repeat. You may think that you could achieve this with something like:

```
flash
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO
        movlw   b'000000'       ; set GP1 low
        movwf   GPIO
        goto    flash           ; repeat forever
```

If you try this code, you'll find that the LED appears to remain on continuously. In fact, it's flashing too fast for the eye to see.

Our PIC is using an internal RC oscillator[1], clocked at a nominal 4 MHz. Each instruction executes in four clock cycles, or 1 µs – except instructions which branch to another location, such as '`goto`', which require two instruction cycles, or 2 µs[2].

This loop takes a total of 6 µs, so the LED flashes at 1/(6 µs) = 166.7 kHz. That's much to fast to see!

To slow it down to a more sedate (and visible!) 1 Hz, we have to add a delay. But before looking at delays, we can make a small improvement to the code.

To flip, or toggle, a single bit – to change it from 0 to 1 or from 1 to 0, you can exclusive-or it with 1.

That is:

0 XOR 1 = 1

1 XOR 1 = 0

So to repeatedly toggle GP1, we can read the current state of GPIO, exclusive-or the bit corresponding to GP1, then write it back to GPIO, as follows:

```
        movlw   b'000010'       ; bit mask to toggle GP1 only
flash
        xorwf   GPIO,f          ; toggle GP1 using mask in W
        goto    flash           ; repeat forever
```

The '`xorwf`' instruction exclusive-ors the W register with the specified register – "e**x**clusive-**or W** with **f**ile register", and writes the result either to the specified file register (GPIO in this case) or to W.

Note that there is no need to set GP1 to an initial state; whether it's high or low to start with, it will be successively flipped.

Many of the PIC instructions, like `xorwf`, are able to place the result of an operation (e.g. add, subtract, or in this case XOR) into either a file register or W. This is referred to as the instruction destination. A '`,f`' at the end indicates that the result should be written back to the file register; to place the result in W, use '`,w`' instead.

This single instruction – '`xorwf GPIO,f`' – is doing a lot of work. It reads GPIO, performs the XOR operation, and then writes the result back to GPIO.

The read-modify-write problem

And therein lays a potential problem. You'll find it referred to as the *read-modify-write* problem. When an instruction reads a port register, such as GPIO, the value that is read back is not necessarily the value that

[1] The 12F508 has been configured (using the `__config` directive) to use its internal RC oscillator, while the 10F200 can only use an internal RC oscillator; there is no other choice.

[2] Assuming a 4 MHz processor clock

you originally wrote to it. When the PIC reads a port register, it doesn't read the value in the "output latch" (i.e. the value you wrote to it). Instead, it reads the pins themselves – the voltages present in the circuit.

Normally, that doesn't matter. When you write a '1', the corresponding pin (if configured as an output) will go to a high voltage level, and when you then read that pin, it's still at a high voltage, so it reads back as a '1'. But if there's excessive load on that pin, the PIC may not be able to drive it high, and it will read as a '0'. Or capacitance loading the output line may mean a delay between the PIC's attempt to raise the voltage and the voltage actually swinging high enough to register as a '1'. Or noise in the circuit may mean that a line that normally reads as a '1', sometimes (randomly) reads as a '0'.

In this simple case, particularly when we slow the flashing down to 1 Hz, you'll find that this isn't an issue. The above code will usually work correctly. But it's good to get into good habits early. For the reasons given above, it is considered "bad practice" to assume a value you have previously written is still present on an I/O port register.

It's better to keep a copy of what the port value is supposed to be, and operate on that, then copy it to the port register. This is referred to as using a *shadow register*.

We could use W as a shadow register, as follows:

```
        movlw   b'000000'       ; start with W zeroed
flash
        xorlw   b'000010'       ; toggle W bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; and write to GPIO
        goto    flash           ; repeat forever
```

Each time around the loop, the contents of W are updated and then written to the I/O port.

The 'xorlw' instruction exclusive-ors a literal value with the W register, placing the result in W – "e**x**clusive-**or l**iteral to **W**".

Normally, instead of 'movlw b'000000'' (or simply 'movlw 0') you'd use the 'clrw' instruction – "**cl**ear **W**".

'clrw' has the same effect as 'movlw 0', except that 'clrw' sets the 'Z' (zero) status flag, while the 'movlw' instruction doesn't affect any of the status flags, including Z.

Status flags are bits in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	-	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

Certain arithmetic or logical operations will set or clear the Z, DC or C status bits, and other instructions can test these bits, and take different actions depending on their value. We'll see examples of testing these flags in later lessons.

We're not using Z here, so we can use clrw to make the code more readable:

```
        clrw                    ; use W to shadow GPIO - initially zeroed
flash
        xorlw   b'000010'       ; toggle W bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; and write to GPIO
        goto    flash           ; repeat forever
```

It would be very unusual to be able to use W as a shadow register, because it is used in so many PIC instructions. When we add delay code, it will certainly need to be able to change the contents of W, so we'll have to use a file register to hold the shadow copy of GPIO.

In lesson 1, we saw how to allocate data memory for variables (such as shadow registers), using the UDATA and RES directives. In this case, we need something like:

```
;***** VARIABLE DEFINITIONS
        UDATA
sGPIO   res 1                    ; shadow copy of GPIO
```

The flashing code now becomes:

```
        clrf    sGPIO           ; clear shadow register
flash
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO
        goto    flash           ; repeat forever
```

That's nearly twice as much code as the first version, that operated on GPIO directly, but this version is much more robust.

There are two new instructions here.

'clrf' clears (sets to 0) the specified register – "**cl**ear **f**ile register".

'movf', with ',w' as the destination, copies the contents of the specified register to W – "**mov**e **f**ile register to destination". This is the instruction used to read a register.

'movf', with ',f' as the destination, copies the contents of the specified register to itself. That would seem to be pointless; why copy a register back to itself? The answer is that the 'movf' instruction affects the Z status flag, so copying a register to itself is a sneaky way to test whether the value in the register is zero.

Delay Loops

To make the flashing visible, we need to slow it down, and that means getting the PIC to "do nothing" between LED changes.

The baseline PICs do have a "do nothing" instruction: 'nop' – "**n**o **op**eration". All it does is to take some time to execute.

How much time depends on the clock rate. Instructions are executed at one quarter the rate of the processor clock. In this case, the PIC is using the internal RC clock, running at a nominal 4 MHz (see lesson 1). The instructions are clocked at ¼ of this rate: 1 MHz. Each instruction cycle is then 1 µs.

Most baseline PIC instructions, including 'nop', execute in a single cycle. The exceptions are those which jump to another location (such as 'goto') or if an instruction is conditionally skipped (we'll see an example of this soon). So 'nop' provides a 1 µs delay – not very long!

Another "do nothing" instruction is 'goto $+1'. Since '$' stands for the current address, '$+1' is the address of the next instruction. Hence, 'goto $+1' jumps to the following instruction – apparently useless behaviour. But all 'goto' instructions executes in two cycles. So 'goto $+1' provides a 2 µs delay in a single instruction – equivalent to two 'nop's, but using less program memory.

To flash at 1 Hz, the PIC should light the LED, wait for 0.5 s, turn off the LED, wait for another 0.5 s, and then repeat.