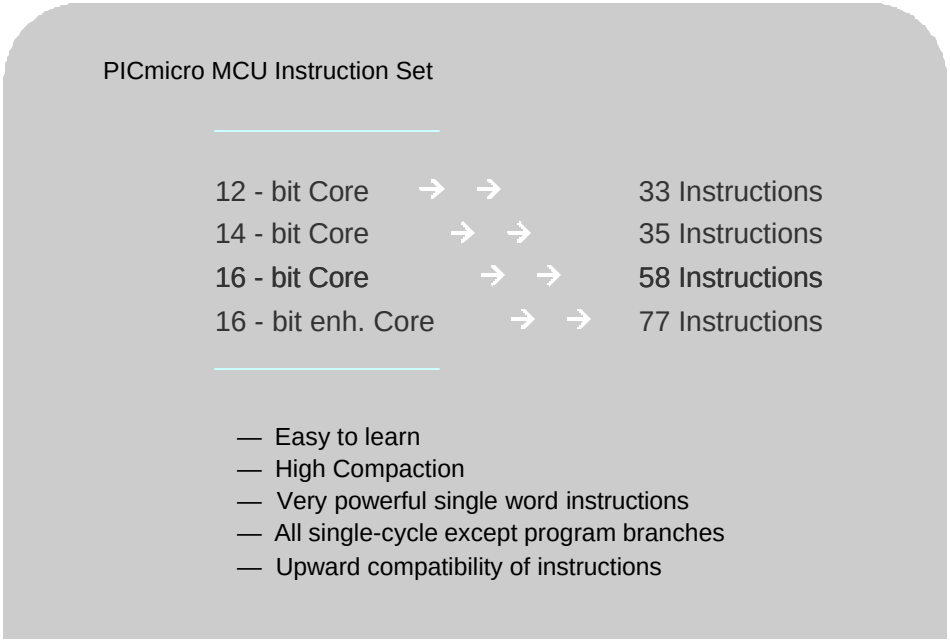
Text Script for "PICmicro® x14 Instruction Set"

Slide 1: Title Slide

PICmicro® Instruction Set (x14)

Thank you for joining the Microchip Technology Inc. PICmicro x14 Instruction set training class. For this presentation, we assume that you have already attended the x14 architecture class or are already somewhat familiar with the PICmicro x14 architecture.

Slide 2. PICmicro MCU Instruction Set

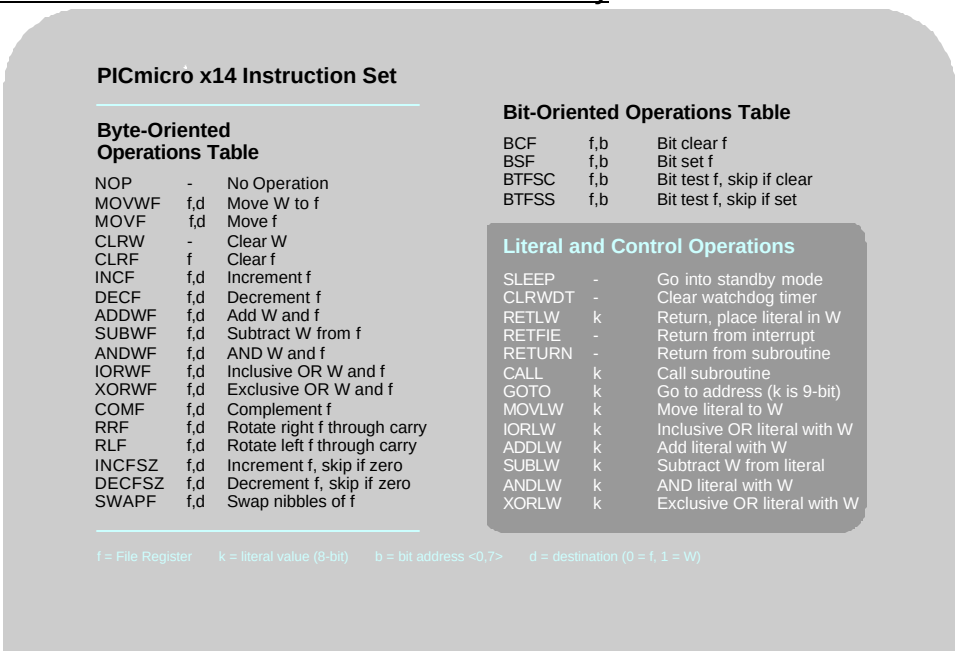


The instruction sets for Microchips different PICmicro architectures range from 33 instructions for the x12 bit core to 7 instructions for the x16 bit enhanced core. The x14 bit PICmicro family as you can see has 35 instructions. When we say "x14", this indicates that the instruction word is 14 bits wide as opposed to the x12 architecture which is 12 bits wide or the x16 architectures which are 16bits wide.

As mentioned before, there are only 35 instructions in the x14 instruction set, so it's very easy to learn and there is a high level of code compaction. All instructions are single word, single cycle instructions, which makes it very easy to use and to count timing loops. The only exceptions to this rule are instructions that modify the program counter which take two cycles. For example, program branches are two cycle instructions because they disturb the pipe line.

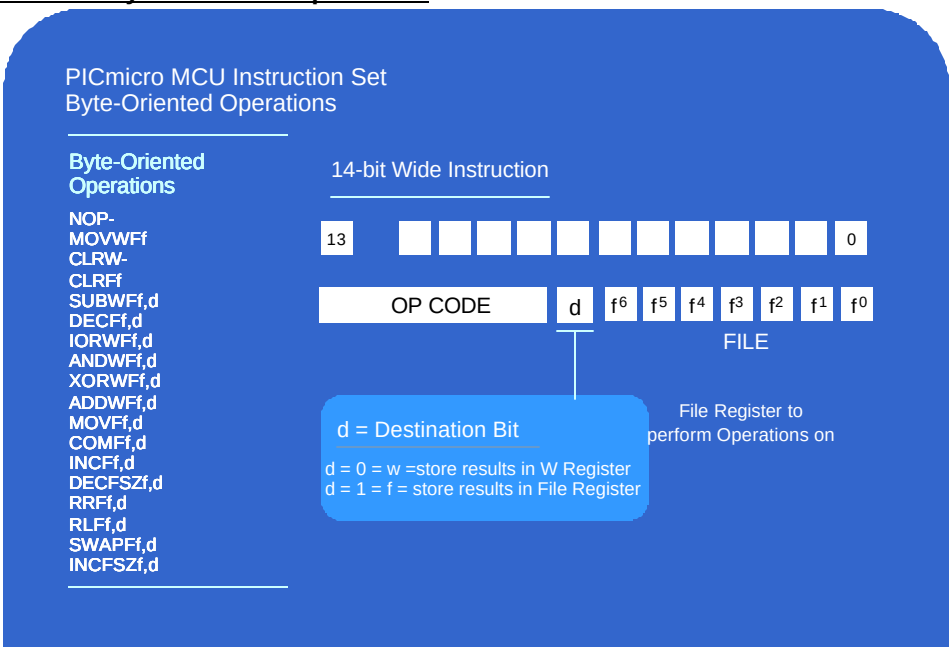
On the PICmicro families, one of the more important things to remember is that all of the instructions are upward compatible. This means that the 33 instructions on the 12 bit core are also available as part of the instruction sets in the PICmicro family all the way up to the 16 bit enhanced core. This is important because it makes it very easy for you to migrate your code from one architecture to another.

Slide 3. PICmicro MCU Instruction Set Summary



This slide gives us a summary of the 35 instructions in the x14 bit wide instruction set The instruction set is broken up into 3 major groups; Byte oriented operations, Bit-Oriented operations and the Literal and Control operations. We will go through each one of these blocks in more detail in the following slides.

Slide 4. Byte-Oriented Operations



First of all let's take a look at the byte oriented operations. (As the name implies, these instructions all move or manipulate an entire byte of data.) The encoding format of 14 bit instruction is shown here.

Each operation starts with a 6 bit op code which is the Mnemonic that determines which instruction will be executed. As you can see in the table, there are 18 instructions that are considered Byte-Oriented instructions.

Most but not all of the instructions have a destination bit which determines where the result of the instruction is stored. If the destination bit is set to a 0, then after the instruction is executed, the result will be stored in the W register and the File register will be left unchanged. If the destination bit is set to a 1, then the result of the instruction will be stored back into the file register and will overwrite the previous contents.

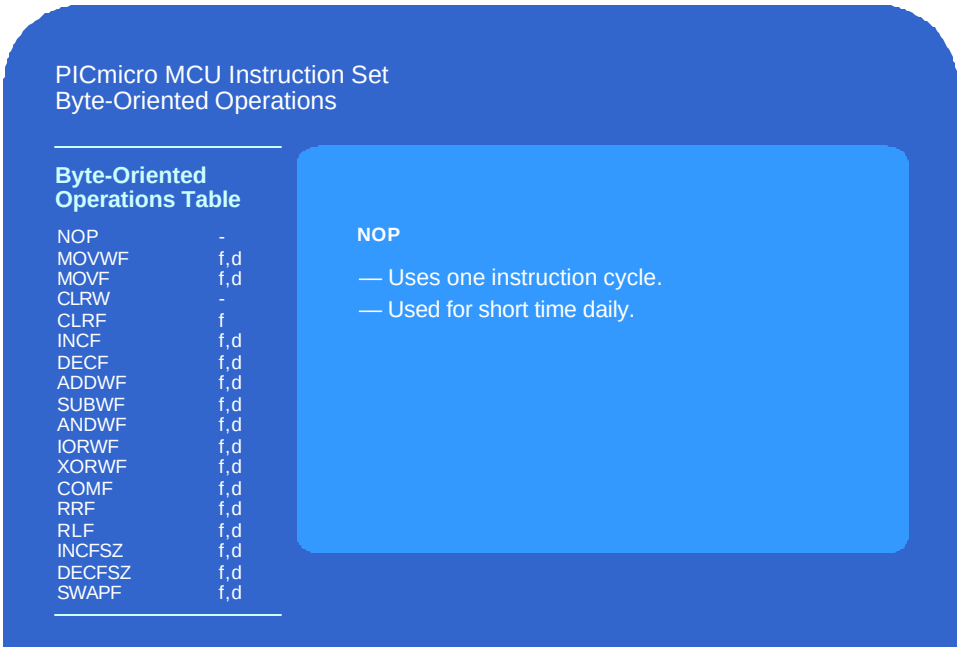
It should also be noted that for ease of programming, you do not have to remember when to use 0 and 1 for your destination bits. If you use the standard include files provided my Microchip for each controller, you can use a w when you want the result stored in the W register and an f when you want the result put back into the file register.

The lower 6 bits of the instruction contain the file register which for most commands determines what register location is acted upon by the instruction.

In actual use in a program, the standard format of these instructions is to have the OP Code first, followed by the file register and the destination bit separated by a comma. For example, lets look at one of the simpler instructions, the MOVF instruction. We can use this command to take the contents of the file register defined by the 7 file register bits and move it into the W register. In the example here, were are going to move the contents of the file register at location 0x05hex and you can see here that we are using a w as our destination bit, so we will move the file register contents into the W register.

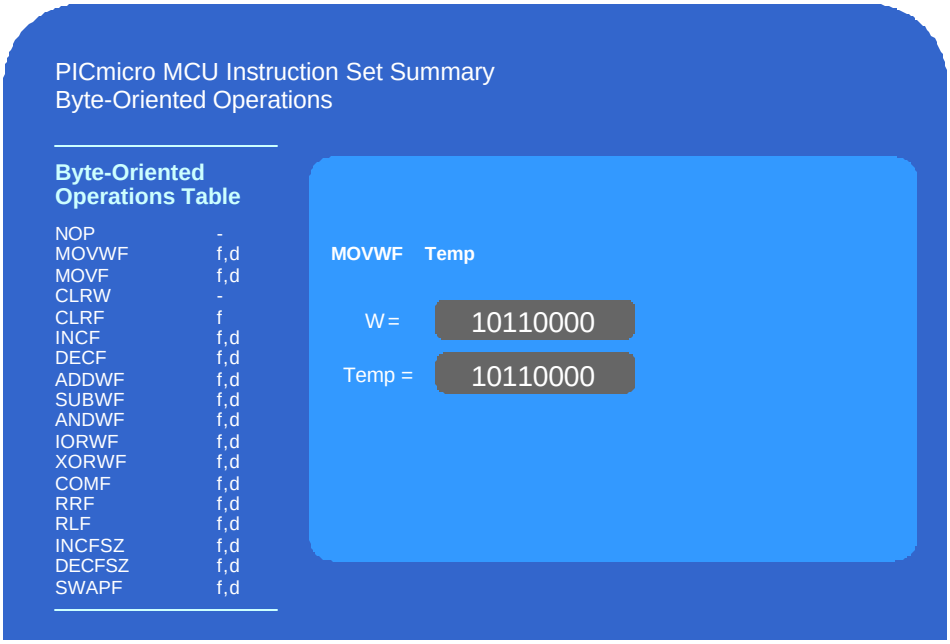
The following group of slides show a brief explanation of each of the Byte-oriented instructions

Slide 5. NOP Instruction



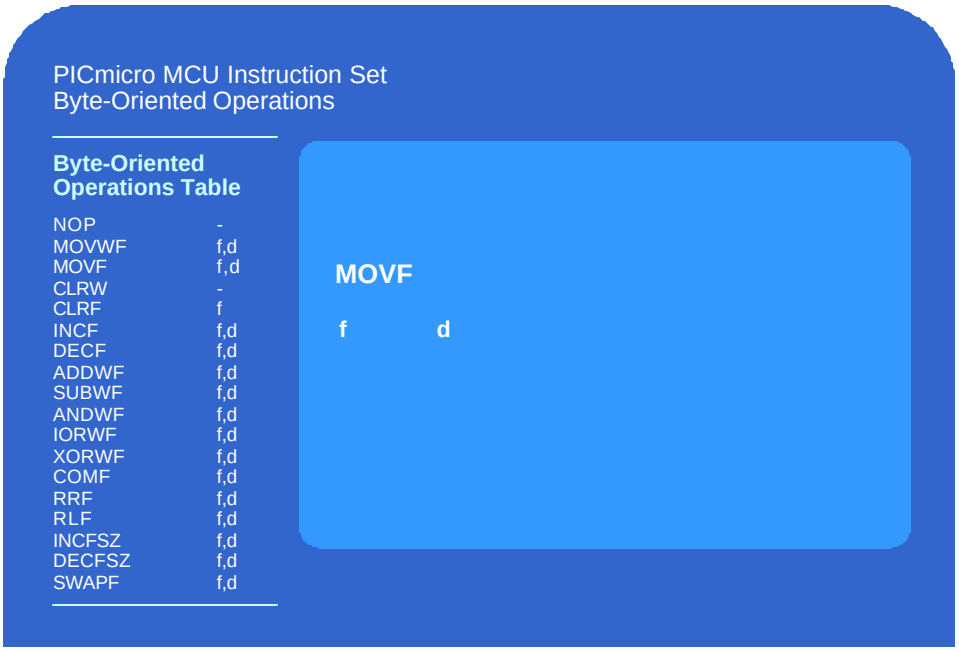
The first instruction in the table is the No-operation or no-op [NOP], which essentially does nothing but use up one instruction cycle . The NOP is typically used when short time delays are required.

Slide 6. MOVWF Instruction



The second instruction in the table is the move W to File instruction [MOVWF]. This instruction allows you to move the contents of the W register into the file register that's defined in the 7 bit file register address. For example, lets assume we have a register that we have defined as Temp, and we execute the MOVWF command on this register with the hex value 0xC0 in the W register. When we execute the command, the value 0xC0 is transferred from the W register into the register called Temp.

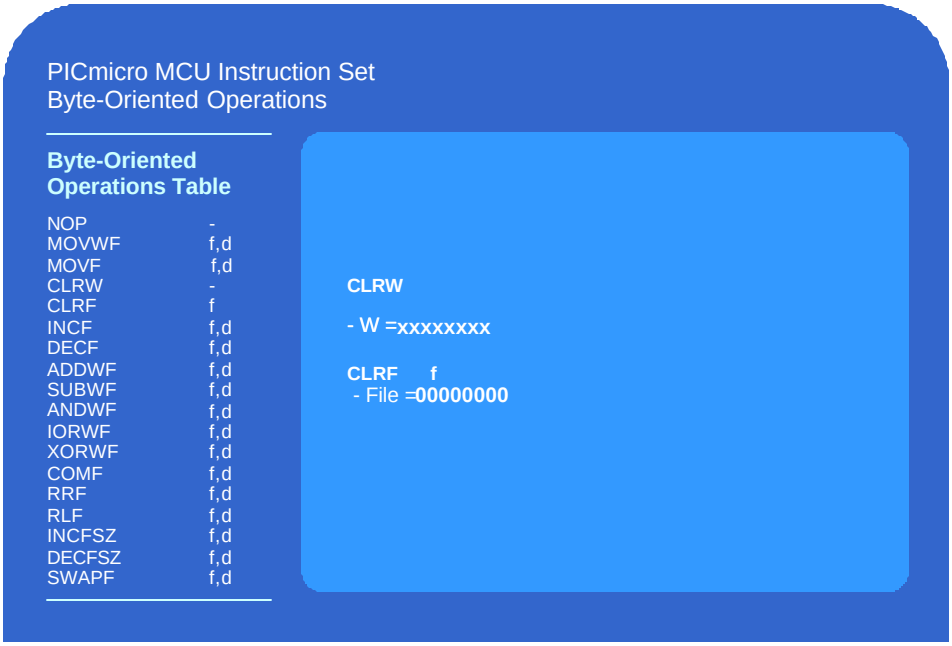
Slide 7. MOVF Instruction



Let's take a look at the move File instruction [MOVF] which we talked about briefly a minute ago when we were talking about the standard instruction format. You will notice that unlike the MOVWF instruction that we just talked about, the MOVF has both a file register designator *and* a destination bit.

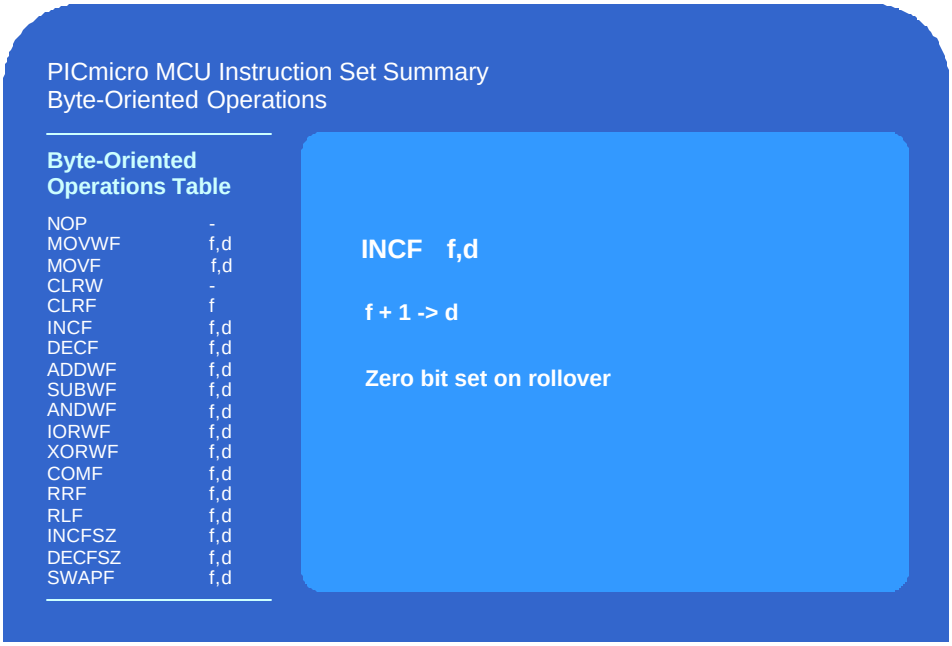
Now you may wonder what the purpose of the destination bit is for this instruction, because if you set the destination as the file register, it would appear that this instruction has no purpose. All it would do is take the contents of the file register, and move it back into the same register. This is essentially what happens and although this appears to do nothing, it actually performs an important task which is to change the contents of the "zero bit" in the status register accordingly. This instruction will *set* the zero bit if the file register has the value zero in it, and *clear* the zero bit if the file register contains a non zero value. Using the MOVF instruction in this manner allows you to easily identify if the contents of the defined register contains zero or not.

Slide 8. CLRW/CLRF Instruction



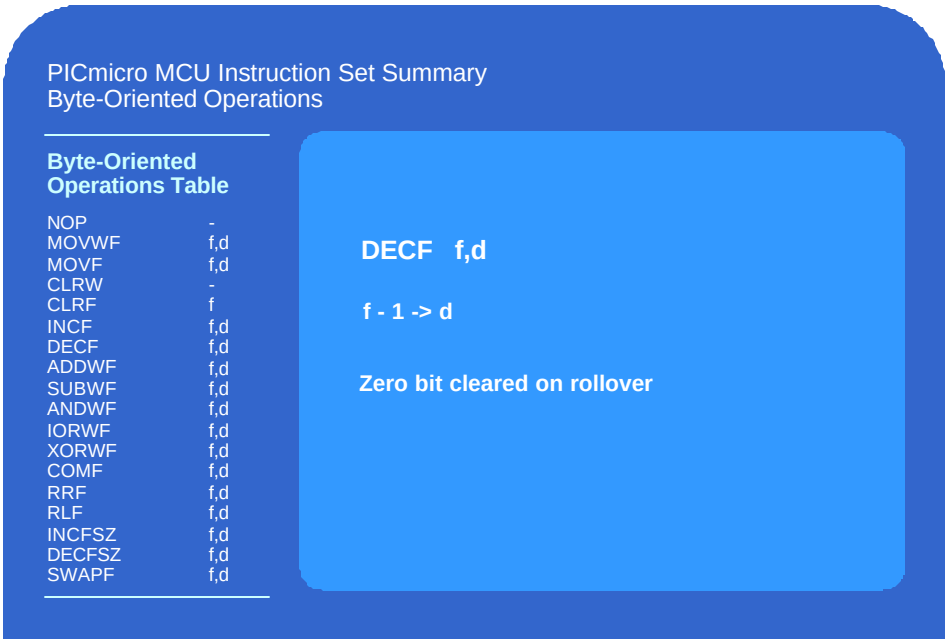
The Clear W [CLRW] instruction clears the contents of the W register to all zeros and similarly, the Clear File instruction [CLRF] clears the file that you identify in the 7 bit file register address.

Slide 9. INCF Instruction



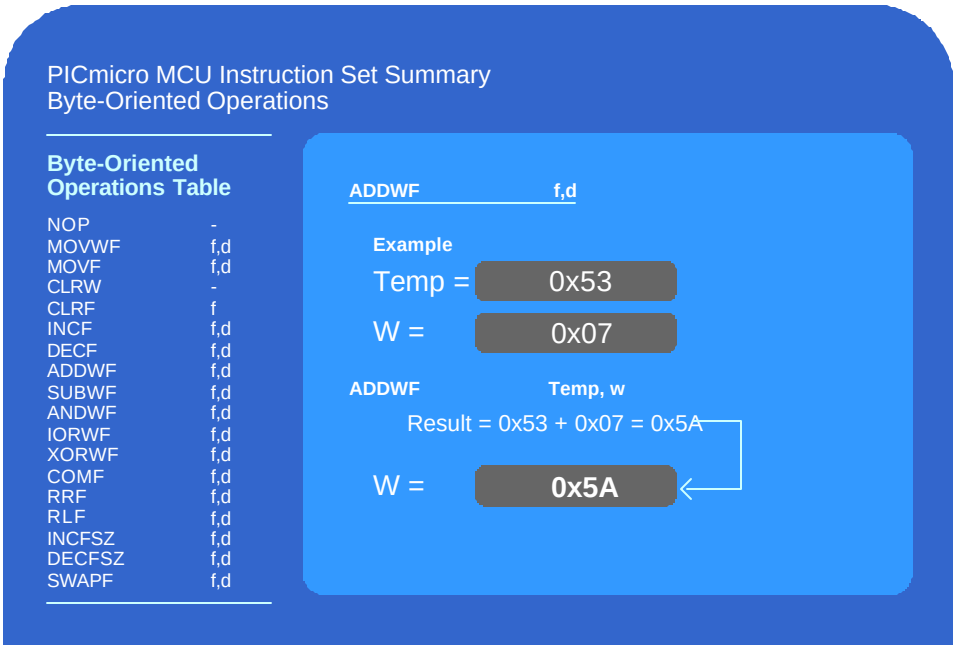
The Increment file [INCF] instruction simply increments the file by one. It is important to note that if a register with all ones in it is incremented, it will roll over to all zeros and the zero status bit will be set.

Slide 10. DECF Instruction



The Decrement file instruction [DECF] operates the same as the Increment File instruction except that it decrements the register by one instead of incrementing. If a register containing all zeros[2] is decremented, it will rollover to all ones.

Slide 11. ADDWF Instruction



Let's look at some arithmetic functions now, the first one being the Add W File instruction [ADDWF]. This instruction allows you to add two 8 bit numbers together and produce an 8 bit result.

Lets look at an example where we have defined a register called temp and we have put the value 0x53 hex into temp. Lets also assume that the W register contains the value 0x07 hex before get to this portion of our code. When we execute the instruction, the result will be the value of Temp

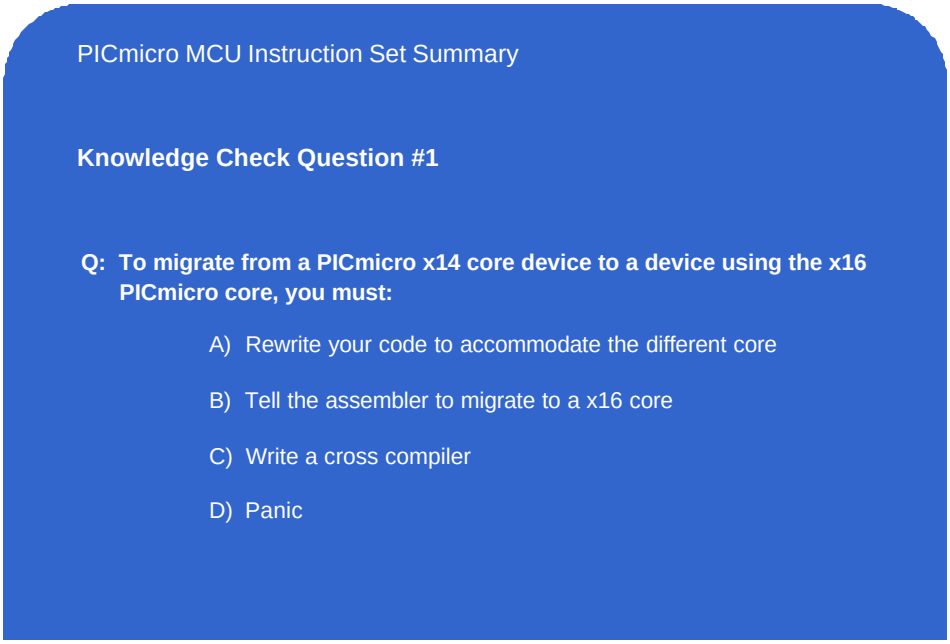
added to the value of W which in this case is 0x5A hex. Now you will notice that for this example, we used the "w" character for our destination bit, which means that the result is going to be put into the W register and Temp will remain with its original value If we had used "f" instead of "w" in the destination bit then the result would have gone into the Temp register.

Alright, now lets do another example. We are going to add two more numbers, but we are going to use bigger numbers this time. Our value of Temp is going to be 0xA3h and we are going to assume that our W register contains the value 0xB7h.

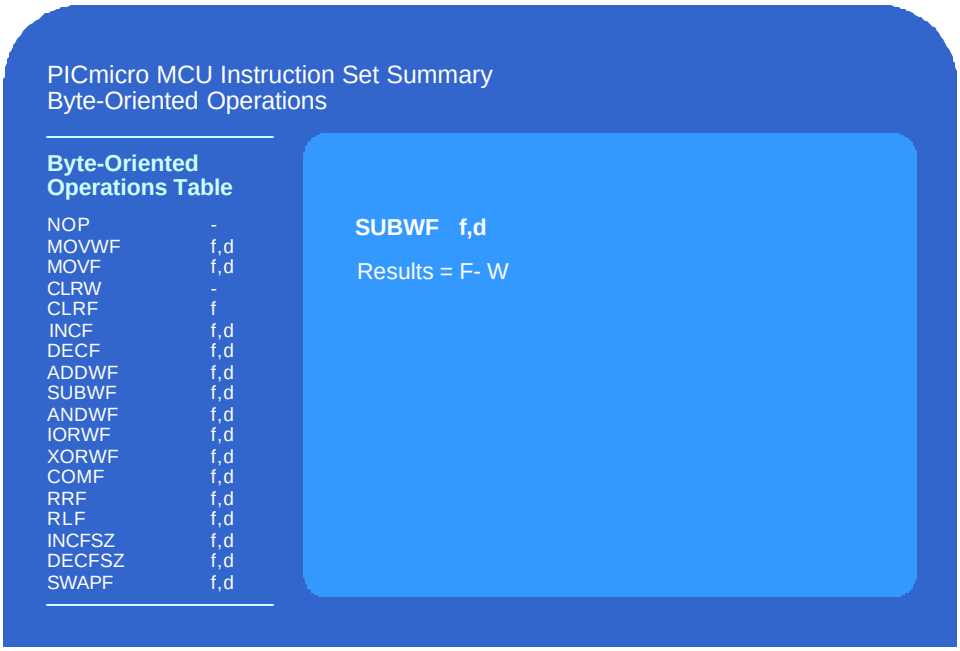
Just like the last time, when we execute the instruction, the result will be the value of Temp added to the value of W which in this case is 0x15A hex. Now you notice that this time, we were adding larger numbers and the result was larger than 255, therefore it requires 9 bits instead of 8 like our last example. So our result goes into the W register just like the last example, but the W register is only 8 bits so only the lower 8 bits go here. And since our result was greater than 255, the carry bit in the status register is also set.

The lesson here is to make sure you take into account the state of the carry bit after you execute an add instruction.

Slide 12. Knowledge Check 1

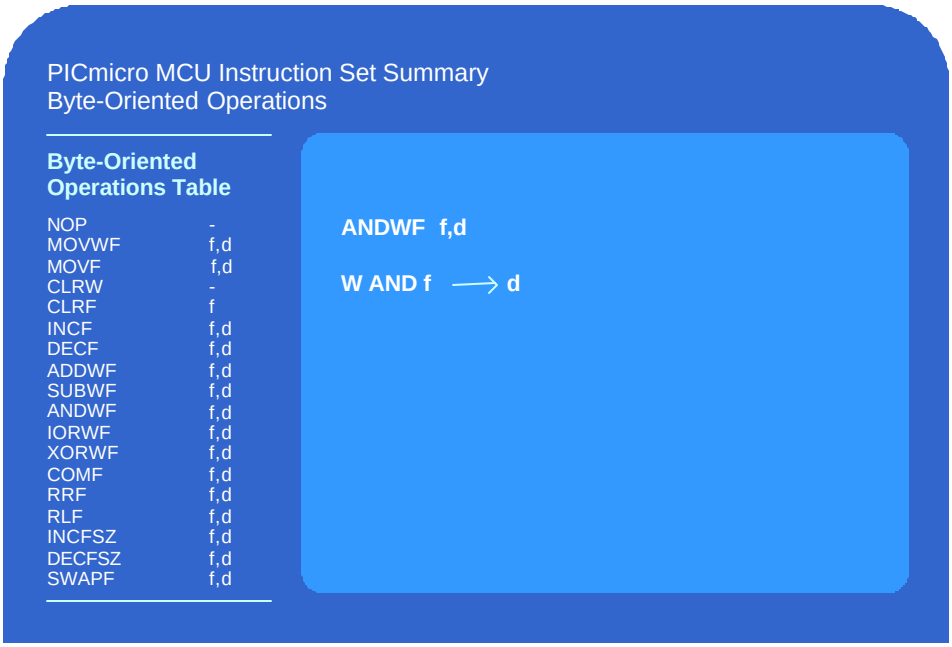


Slide 13. SUBWF Instruction



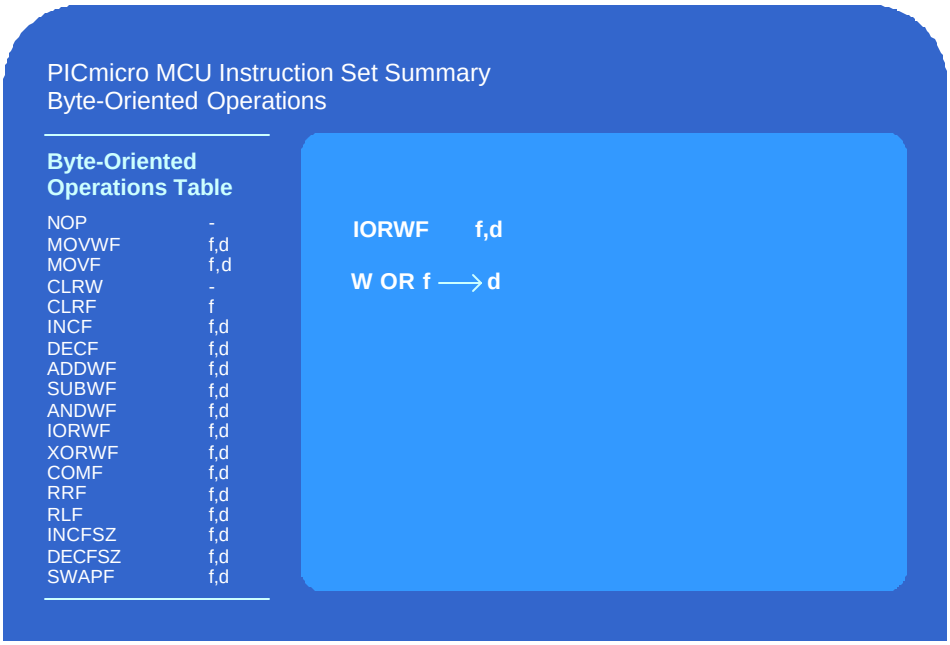
In a similar fashion to the Add instruction, we can also do a subtraction using the Subtract W File instruction [SUBWF]. This instruction subtracts W from the file defined in the 7 bit register address. You have to be careful how you execute the subtract since it is a twos compliment subtract so make sure this has been considered.

Slide 14. ANDWF Instruction



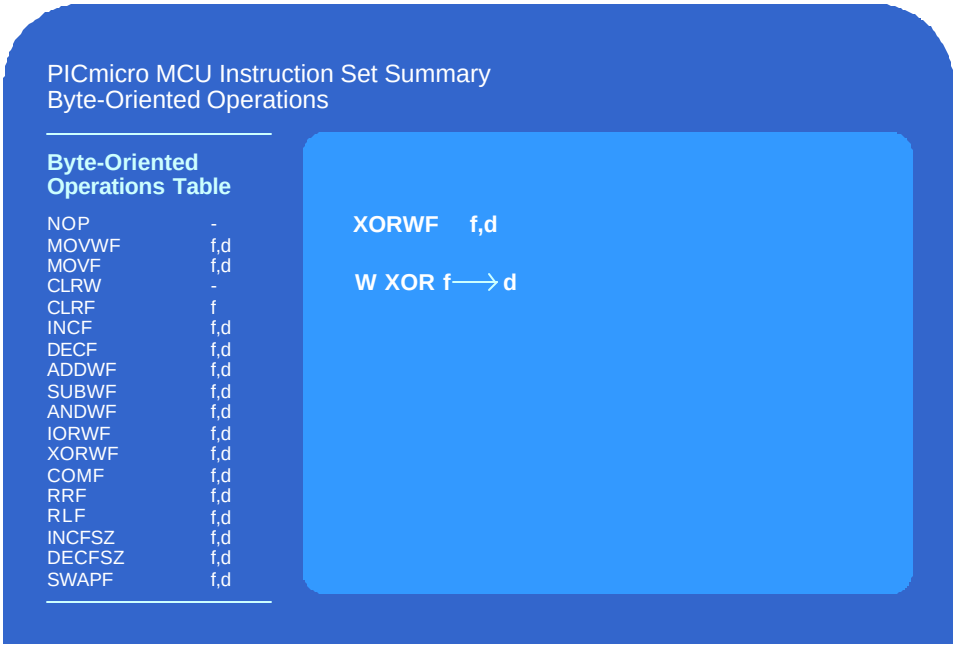
The And W File instruction [ANDWF] will execute a logical AND with the contents of the W register and the file register. This command is often used when it is necessary to clear or 'mask out' certain bits in a register. This is done by ANDing the register with a value that has zeros in the bit positions that need to be cleared and ones in the bit positions that need to be maintained.

Slide 15. IORWF Instruction



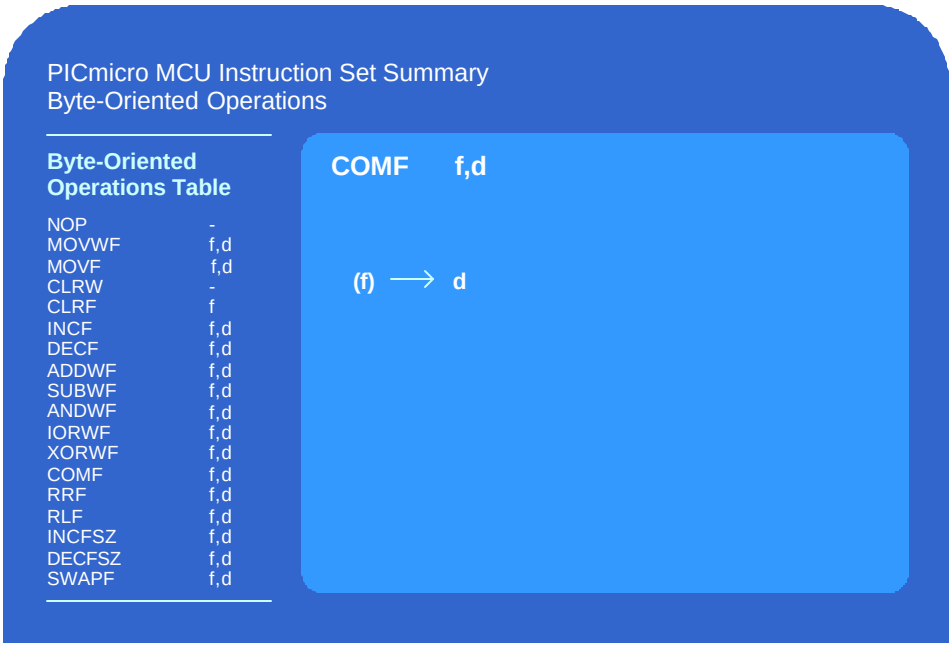
The Inclusive OR W File instruction [IORWF] will perform a bit-by-bit inclusive OR function with the W register and the File register.

Slide 16. XORWF Instruction



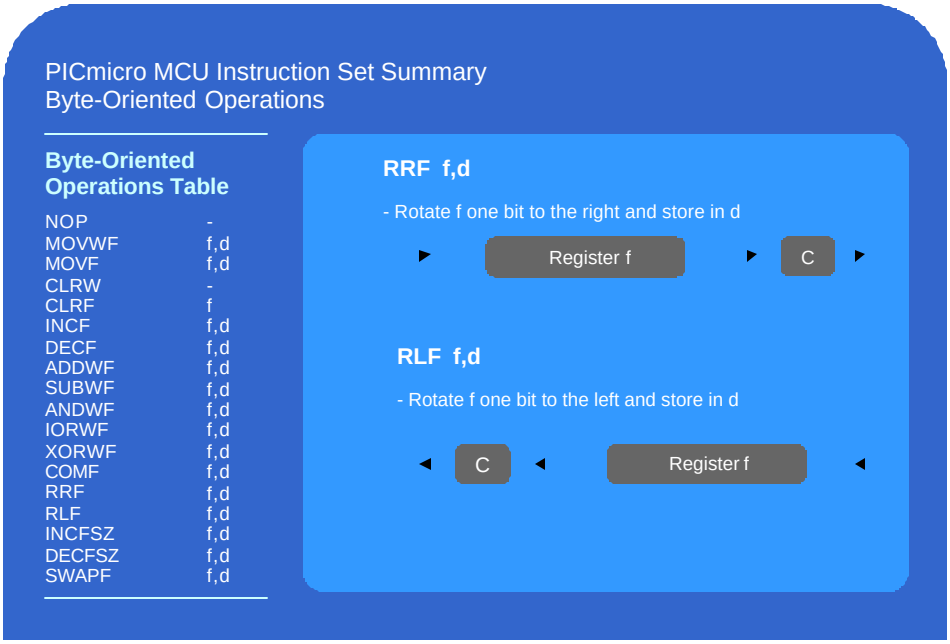
Likewise, the Exclusive OR W File instruction [XORWF] operates in a similar fashion except that the function is the exclusive OR instead of the Inclusive OR .

Slide 17. COMF Instruction



The compliment file instruction [COMF] instruction simply executes a one's compliment on the contents of the file register.

Slide 18. RRF/RLF Instructions



The rotate right command (RRF) instruction will rotate the contents of the file register one bit to the right. Note that the contents of the carry bit at the time of this instruction will be rotated into the msb position and the lsb position will be rotated into the carry bit. The rotate left [RLF] instruction operates exactly the same except that the register is rotated one bit to the left.

Slide 19. Knowledge Check 2

PICmicro MCU Instruction Set Summary

Knowledge Check Question #2

Q: The PICmicro instructions

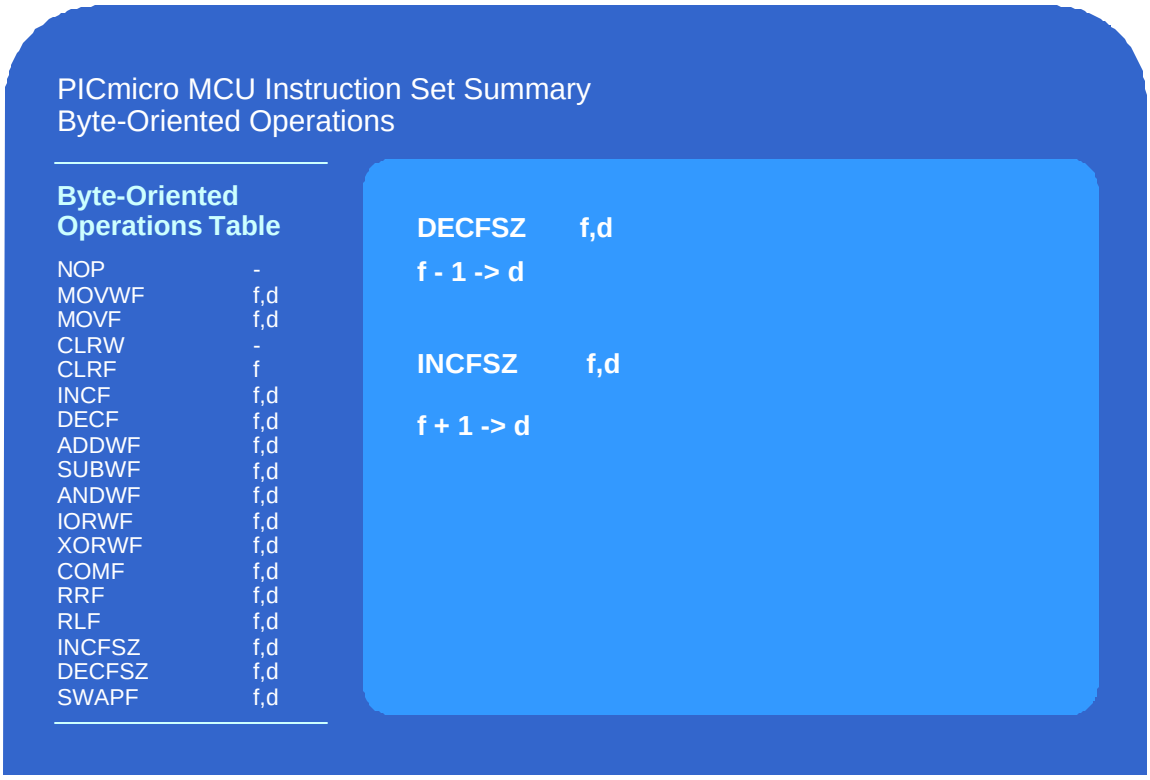
A) Take up to 3 lines of code each

B) Execute in four instruction cycles

C) Take only one line of code each

D) Are very hard to understand

Slide 20. DECFSZ/INCFSZ Instructions



Now we come two instructions that are used for loop control, the increment file, skip if zero instruction [INCFSZ] and the decrement file, skip if zero instruction [DECFSZ]. These instructions actually carry out multiple functions. We will talk about the decrement instruction first.

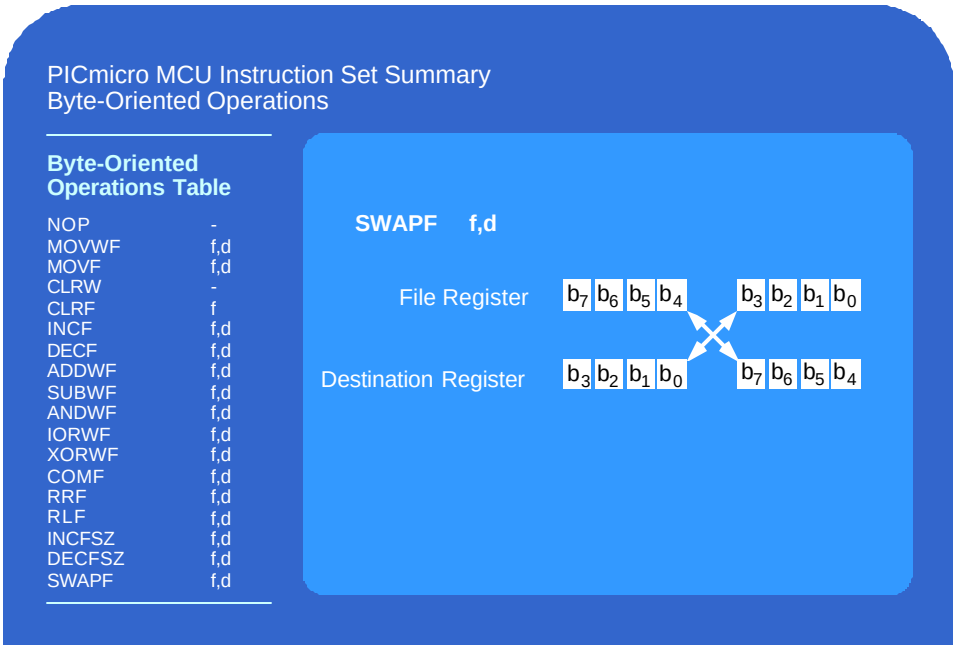
The first thing that happens with this instruction is that the file register defined by the 7 file register bits is decremented by one. After this decrement has been executed, the value if the file register is checked to see if it has decremented all the way to down to zero. If it *is* zero, the next instruction is skipped and the program will continue from there. If the result was *not* zero, then the program will just execute the next instruction.

Lets look at an example to see how this instruction is used for loop control. In our example we have a very simple delay loop. We have a variable defined as “count” that has been set to the value of earlier in our program. Lets put our program counter indictor on the top line of our loop, which is just a NOP instruction used to consume more time. If we wanted a longer delay for our loop, we could add more No op instructions or increase the value of our “count” variable or both. So lets go to the next instruction in our program now, which is our decrement file skip if zero instruction. Notice that we have the destination bit set to “f” so each time we execute the instruction the new result will be stored back into the variable “count” .

It should also be noted here that with these instructions, the “zero bit” in the status register will **not** be affected. As you can see, when we executed this instruction the count variable was decremented from 3 down to 2. Since we are not down to zero yet, the next instruction will be executed and that will take us back to the top of our loop. We will execute the NOP and then land on the decrement command again. You can see that our “count” variable has now moved from 2 down to 1. We are still not down to zero, so we will execute the next instruction, and once more go back to the top of our loop. We will execute the NOP and then land on the decrement command again. You can see that our “count” variable has now reached zero, therefore we will skip over the next instruction and we are now finished with our delay loop.

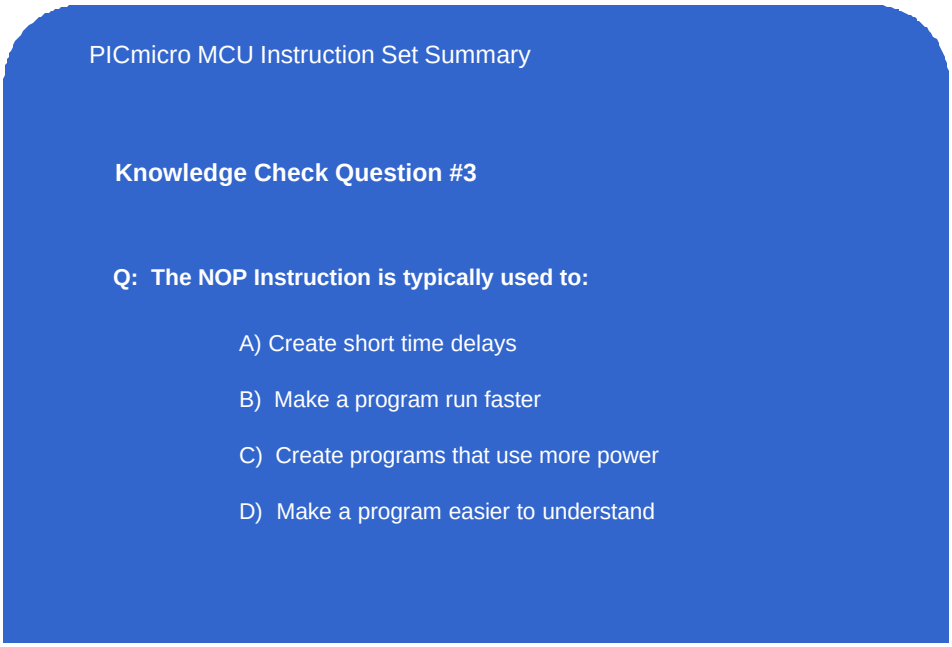
The INCFSZ instruction works the same way expect that the skip will not occur until the count variable goes all the way up to 255 and then rolls over to 0. These two instructions are very important for loop and allows you to use a minimum of code space to create a loop.

Slide 21. SWAPF Instruction

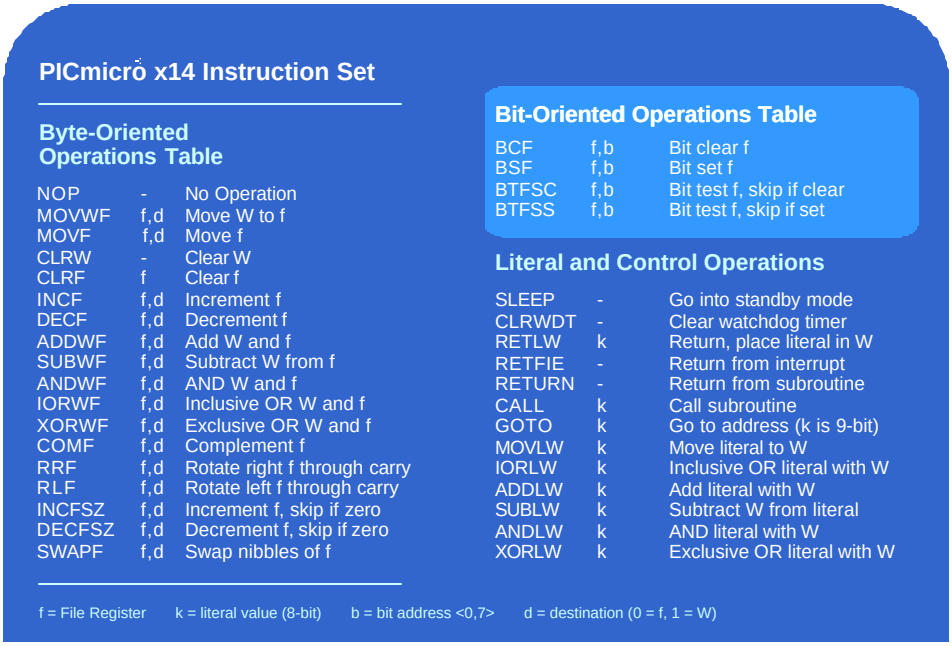


The next instruction we will discuss is the swap file instruction [SWAPF]. This instruction is a nibble swap, which takes the lower 4 bits and swaps them with the upper 4 bits. As an example, lets assume we have a register that we have defined as Temp, and in temp we have placed the binary value 10110000. When we execute the SWAPF instruction with the destination bit set to a one, the high and low nibbles are swapped and we end up with the binary value 00001011 stored in Temp.

Slide 22. Knowledge Check 3

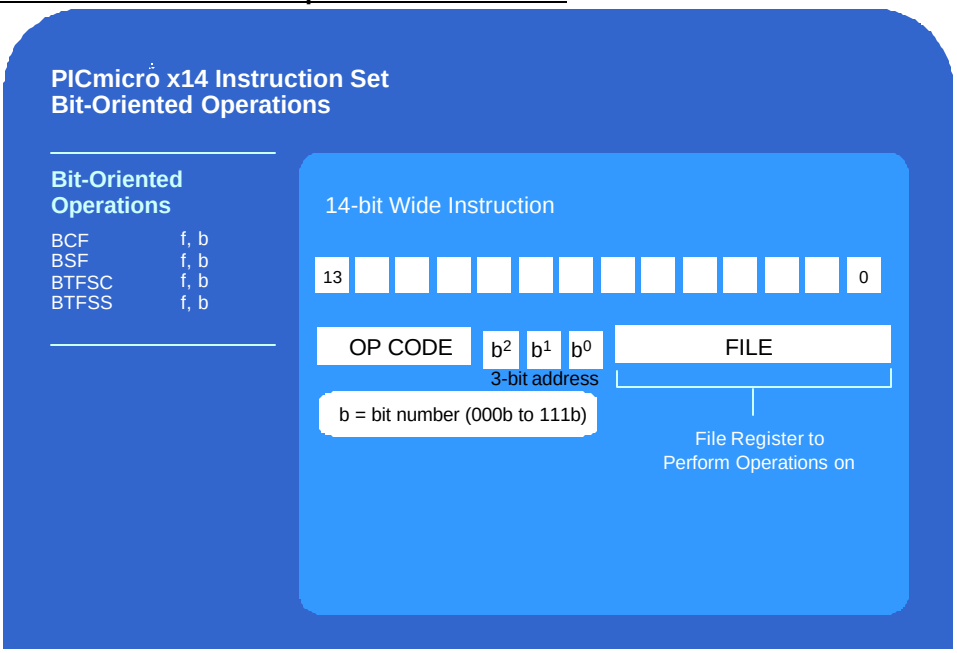


Slide 23. Bit-Oriented Operations



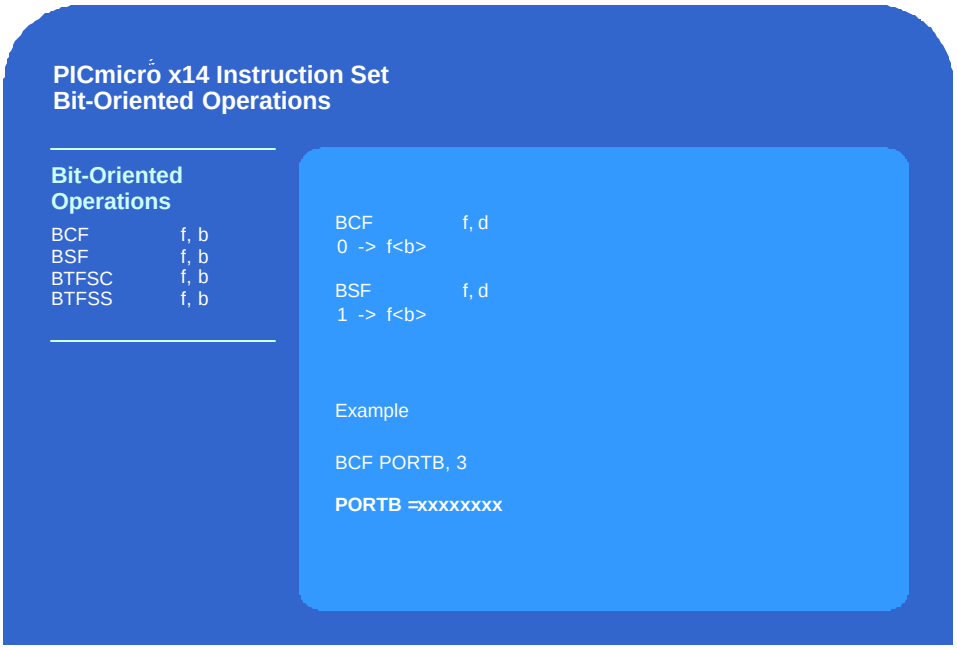
We have now finished our discussion with the Byte oriented operations and we are going to move onto the Bit-Oriented Operations which is one of the most powerful groups of instructions in this architecture .

Slide 24. Bit-Oriented Operation Instructions



The encoding format of the bit-oriented instructions is shown here. Each instruction has of a 4 bit opcode, and as you can see here, there are only 4 bit-oriented instructions. Because these are bit-oriented instructions, we have the 7 bit file register that determines which register location we will be acting upon, but we *also* have a 3 bit position address which determines which bit position is to be acted upon.

Slide 25. BCF/BSF Instruction

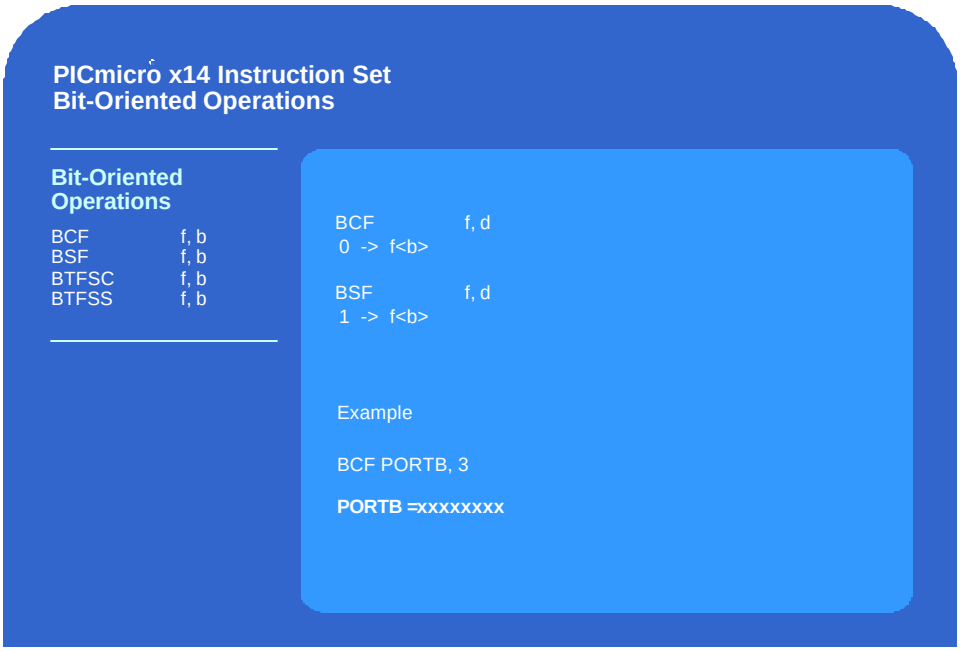


Let's look at the bit clear file [BCF] and bit set file [BSF] instructions first. These instructions allow you to clear or set a specific bit in a register. The file register address in the instruction determines which register we are looking at, and the bit address determines which bit in that register we clear or set.

As an example, lets say you want to set bit 3 on PORTB (RB3) low. If we use the portb definition in our include file, the instruction would look like this. When this instruction is executed, the processor will write a "zero" to the RB3 bit and the I/O port pin will be pulled low. Notice that since this is a bit operation, none of the other bits in the register are disturbed. So that's how the Bit Clear instruction works, and the Bit Set instructions works exactly the same way, except that the pin will be set high instead of low. So as you can see, these instructions give you the capability of setting or clearing a pin with just one instruction.

Keep in mind that you can use the Bit Clear and Bit Set instructions on any register, not just I/O port registers. If you need to keep track of status flags in your program, you could define a register as "flags" where each bit in the register was a flag. You would then use the Bit Set and Bit Clear instructions to manipulate the individual bits or flags in the register.

Slide 26. BTFSC/BTFSS Instructions



Now let's look at the other bit oriented instructions, which are the Bit test file-skip if clear [BTFSC] and the Bit test file-skip if set [BTFSS] instructions. These instructions are similar to the Bit set and Bit clear instructions in that they also have both a file address and a bit address. However, the function of these instructions is to test the state of a particular bit in a register, then execute the next instruction or *skip over* the next instruction based on the state of the bit. These instructions allow you to do conditional branches very easily.

If you look at the example here, we have a simple piece of code. You can see in the first line of code we are using the Bit test file-skip if clear instruction. The register we are looking at is the status register and we are testing the carry bit, which is defined in our include file as the letter "C When the program counter gets to this instruction the state of the carry bit is tested and if the carry bit is set it would execute the next instruction so you could say that this instruction is the "branch if carry" and the next instruction would be the address that you would branch to.

If the carry bit was clear when it was tested, the next instruction will be skipped and we will jump to the "branch if clear" portion of code that handles this condition.

As you can see, these instructions give you the capability of jumping to another location depending on the condition of an I/O pin or a register bit, including the status bits in the status register.

Slide 27. Knowledge Check 4

PICmicro MCU Instruction Set Summary

Knowledge Check Question #4

Q: To change the state of an I/O pin on a PICmicro x14 core device, you must:

A) Read the port, perform a logical function using a mask on the pin then rewrite the port

B) Use a multi-cycle bit oriented instruction

C) Use the special I/O instruction for controlling I/O pins

D) Use the same single word, single cycle instruction you use on RAM locations

Slide 28. Literal and Control Operations

PICmicro x14 Instruction Set

Byte-Oriented Operations Table

NOP	-	No Operation
MOVWF	f,d	Move W to f
MOVF	f,d	Move f
CLRW	-	Clear W
CLRF	f	Clear f
INCF	f,d	Increment f
DECF	f,d	Decrement f
ADDWF	f,d	Add W and f
SUBWF	f,d	Subtract W from f
ANDWF	f,d	AND W and f
IORWF	f,d	Inclusive OR W and f
XORWF	f,d	Exclusive OR W and f
COMF	f,d	Complement f
RRF	f,d	Rotate right f through carry
RLF	f,d	Rotate left f through carry
INCFSZ	f,d	Increment f, skip if zero
DECFSZ	f,d	Decrement f, skip if zero
SWAPF	f,d	Swap nibbles of f

Bit-Oriented Operations Table

BCF	f,b	Bit clear f
BSF	f,b	Bit set f
BTFSC	f,b	Bit test f, skip if clear
BTFSS	f,b	Bit test f, skip if set

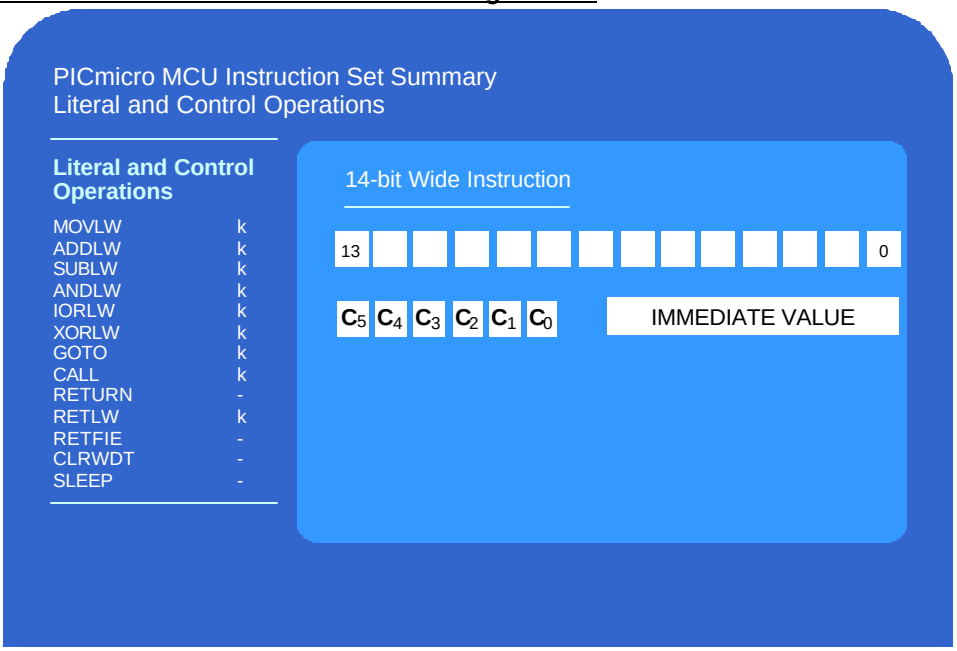
Literal and Control Operations

SLEEP	-	Go into standby mode
CLRWDT	-	Clear watchdog timer
RETLW	k	Return, place literal in W
RETFIE	-	Return from interrupt
RETURN	-	Return from subroutine
CALL	k	Call subroutine
GOTO	k	Go to address (k is 9-bit)
MOVLW	k	Move literal to W
IORLW	k	Inclusive OR literal with W
ADDLW	k	Add literal with W
SUBLW	k	Subtract W from literal
ANDLW	k	AND literal with W
XORLW	k	Exclusive OR literal with W

f = File Register k = literal value (8-bit) b = bit address <0,7> d = destination (0 = f, 1 = W)

We have finished discussing the Byte and Bit-oriented instructions, and now we're going to move on to the final group of instruction, which are the Literal and Control operations.

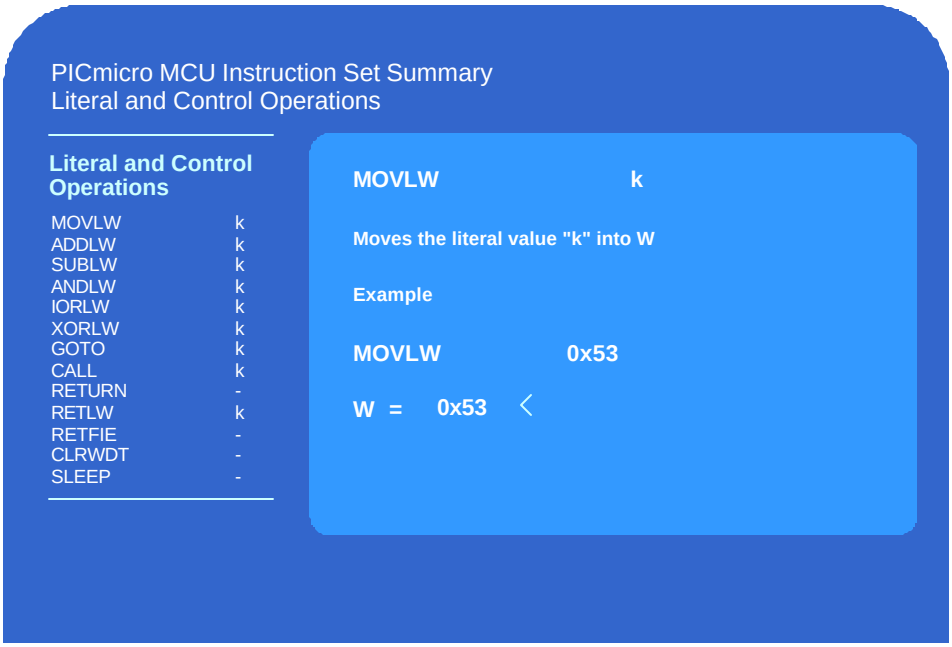
Slide 29. Literal Instructions: Encoding Format



There are 13 literal and control instructions and the bit encoding format is shown here. Each operation starts with a 6 bit op code. Most, but not all of these instructions also have an 8 bit immediate or literal value embedded in the instruction.

Lets quickly go over these instructions as well.

Slide 30. MOVLW Instruction



The first instruction in this group we will discuss is the Move Literal to W instruction [MOVLW]. This instruction will take the 8 bit literal and put it in W register. As an example, we have the instruction MOVLW 0x53 hex, which means we want to move the literal value 0x53 hex into W. When this instruction is executed, the value 0x53 will be moved into the W register.

Slide 31. Knowledge Check 5

PICmicro MCU Instruction Set Summary

Knowledge Check Question #5

Q: The BTFSC instruction is typically used for:

A) Moving data from one register location to another based on the contents of a register.

B) Decide if the program should jump to another location based on the state of a register bit.

C) Making programs run faster

D) Toggling I/O pins based on the contents of register bits

Slide 32. ADDLW Instruction

PICmicro MCU Instruction Set Summary
Literal and Control Operations

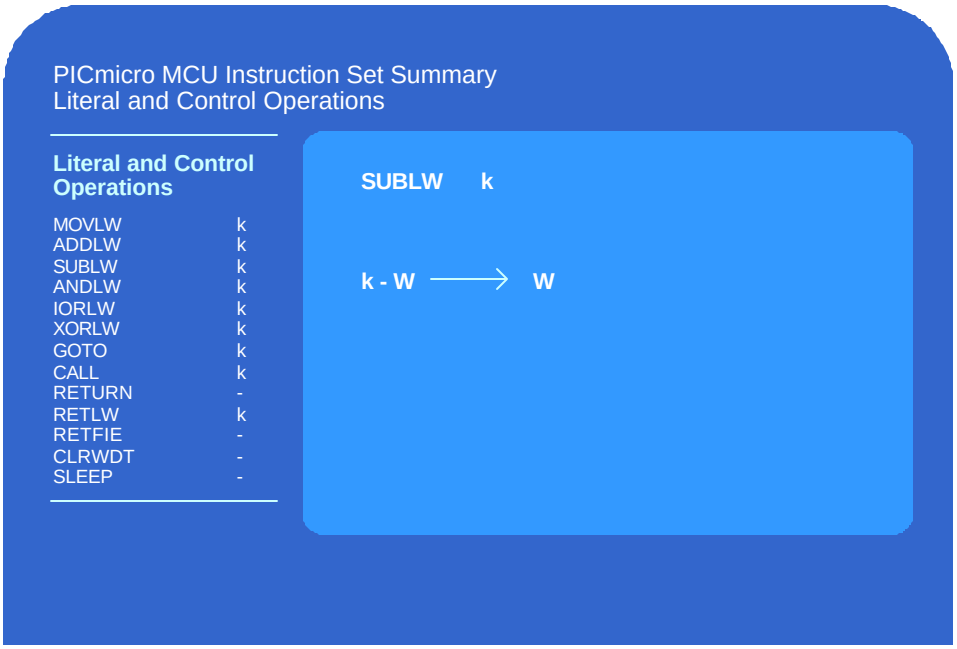
Literal and Control Operations	
MOVLW	k
ADDLW	k
SUBLW	k
ANDLW	k
IORLW	k
XORLW	k
GOTO	k
CALL	k
RETURN	-
RETLW	k
RETFIE	-
CLRWDT	-
SLEEP	-

ADDLW k

W + k ⟶ W

Next we have the Add Literal to W instruction [ADDLW] which takes the 8 bit literal value and adds it to the current contents of the W register.

Slide 33. SUBLW Instruction

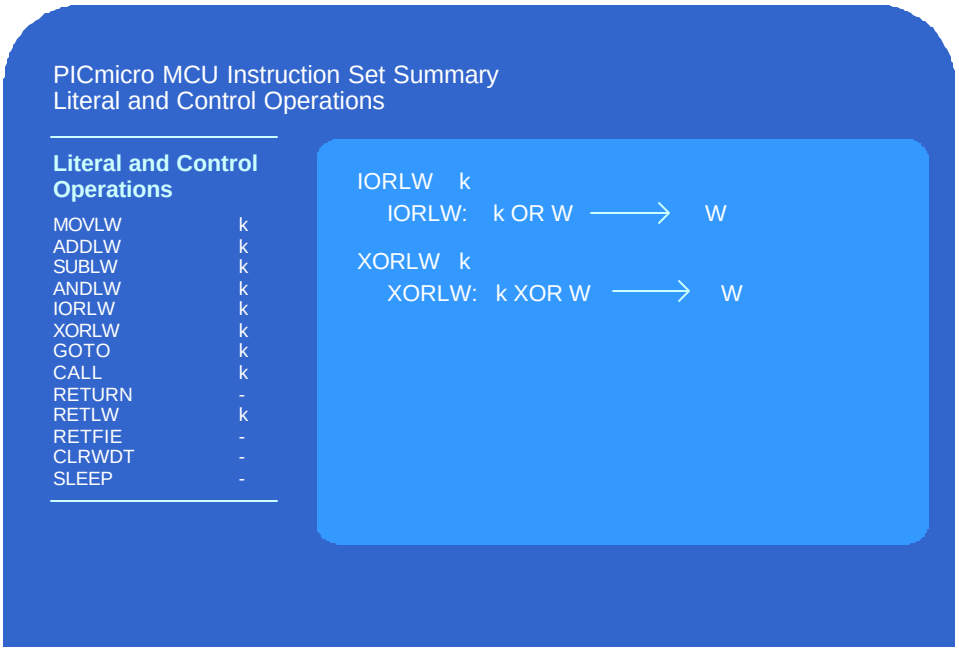


In a similar fashion, we have the Subtract Literal from W instruction [SUBLW] which subtracts the current contents of the W register from the literal value and stores the results back into W.

Slide 34. ANDLW Instruction

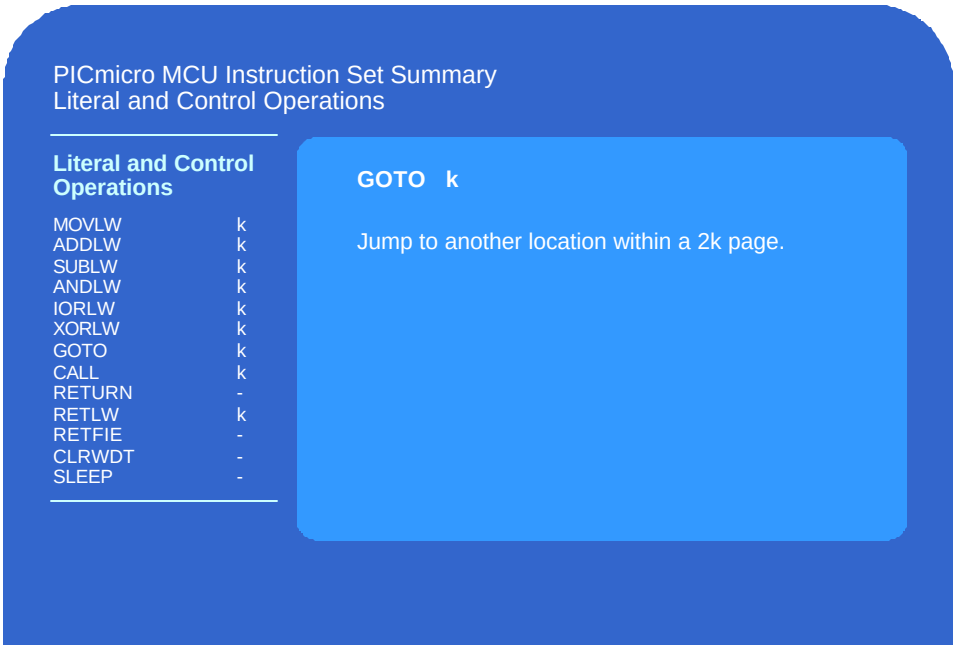
Another of the arithmetic operations is the AND Literal with W instruction [ANDLW] which executes a logical AND of the literal value and the contents of the W register and places the result back into W.

Slide 35. IORLW/XORLW Instructions



Next up are the Inclusive OR literal and W instruction [IORLW] and Exclusive OR Literal and W instructions [XORLW], which will execute the inclusive and exclusive OR functions, respectively, on the literal value and the current contents of the W register. In both cases, the result is placed into the W register.

Slide 36. GOTO Instruction



The GOTO instruction allows you to jump to any other location in a 2K page.

Slide 37. Knowledge Check 6

PICmicro MCU Instruction Set Summary

Knowledge Check Question #6

Q: The "MOVLW 0x37" instruction would:

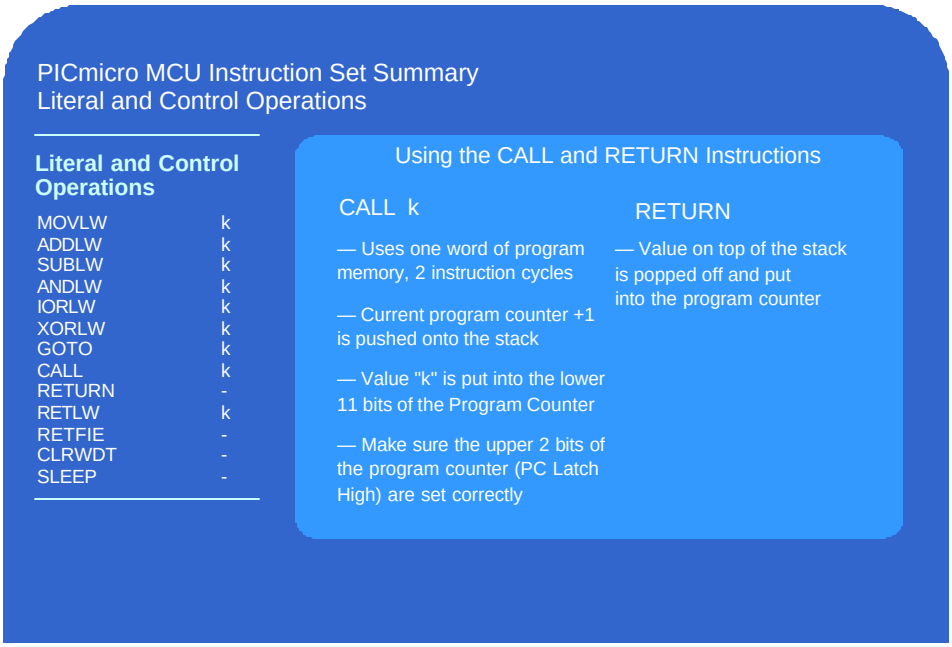
A) Move the contents of the W register into the literal register at address 0x37

B) Move the literal value 0x37 into the W register

C) Move the W register into the address defined by the contents of address 0x37

D) Move the literal value 0x37 into the program counter and clear the W register

Slide 38. CALL/RETURN Instructions



Now we move on to some of the instructions that we use for subroutines. The first is the CALL instruction, which gives you the capability of jumping to a subroutine anywhere in a 2K address space.

As for returning from a subroutine call, there are actually 3 instructions that you can use. The first one we're going to talk about is the Return from Subroutine [RETURN] instruction.

Lets take a closer look at how the CALL and RETURN instructions work together. As previously mentioned, the CALL instruction is used to call a subroutine and has the capability of calling anywhere in a 2K address space. It takes 1 word of program memory, but because it is one of the instructions that disturbs the pipeline, it takes 2 instruction cycles to execute.

When the CALL instruction is executed, there are 2 things that happen: the first operation that results is the current program counter value plus one is pushed onto the top of the stack. The second part of this instruction is when the constant value K from the instruction is put into the lower 11 bits of the program counter. This defines any address within a 2K memory space. If you are using a device with more than 2K of program memory, you need to ensure that you are pointing to the correct page before executing the CALL instruction. For more information on paging, please refer to the PICmicro MCU Midrange Family Reference Manual section 6.2.6.

The Return instruction is the simplest of the 3 instructions that can be used to return from a subroutine. When the RETURN instruction is executed, the value on the top of the stack is popped off and put into the program counter to be used as the return address.

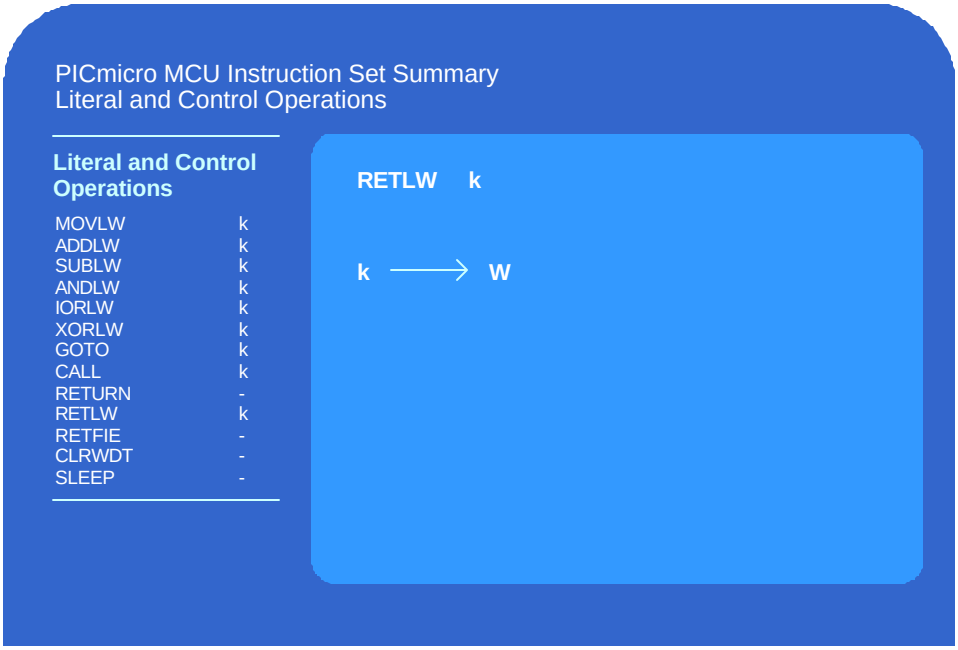
Lets go ahead and look at an example where both the Call and the Return instructions are used.

The example we have here is a very simple code segment that starts at address 11 and a very simple subroutine that starts at address 27. The purpose of this part of the program is rather arbitrary, but for the sake of clarity, it is meant to take a value starting at zero, call a subroutine that outputs the value on the I/O port B. Each time the program returns from the subroutine, the value is incremented by one and the entire process repeats forever. Over on the left hand side we have our 8 level stack and to the right of that we will be monitoring our program counter. Below the code space we will be monitoring 3 registers: W, Temp and PortB.

So lets get started on our code here. The first line of code loads the literal value of zero into the W register and then we move that into a variable that we have defined as Temp. Notice that as we move through our code the program counter value matches the address of each instruction. The next instruction we get to is the call to our subroutine. When this instruction executes, 2 things are going to happen: the first is that the current program counter plus 1, which in this case is address 0014, is going to be pushed onto the stack. You will notice that any values already on the stack are pushed down one level. The second thing that happens is that the immediate value in our instruction, which in this case is the address of our subroutine called Output is moved into the program counter. For our example here, the subroutine called Output starts at address 0027 so this value will be loaded into the program counter. We have now finished executing the CALL instruction and we have jumped to the first line of the subroutine.

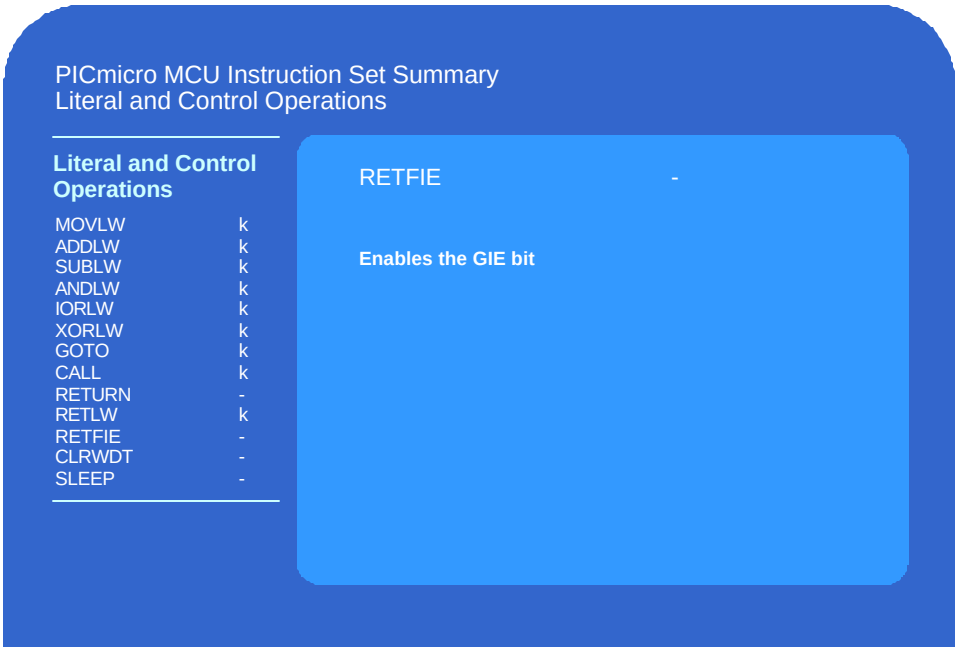
This instruction takes the value we have currently in the variable called temp and moves it into the W register, and the next instruction moves it out to the I/O port called port B. Our subroutine function is now finished, so we come the to the RETURN instruction. When this instruction is executed, we pop the value off the top of the stack, which is our return address, and load this value into the program counter. We then return to our main routine and you can see that this line increments our Temp value by one and then we come to the Goto instruction that takes us back to the top of the loop and starts the entire process over again.

Slide 39. RETLW Instruction



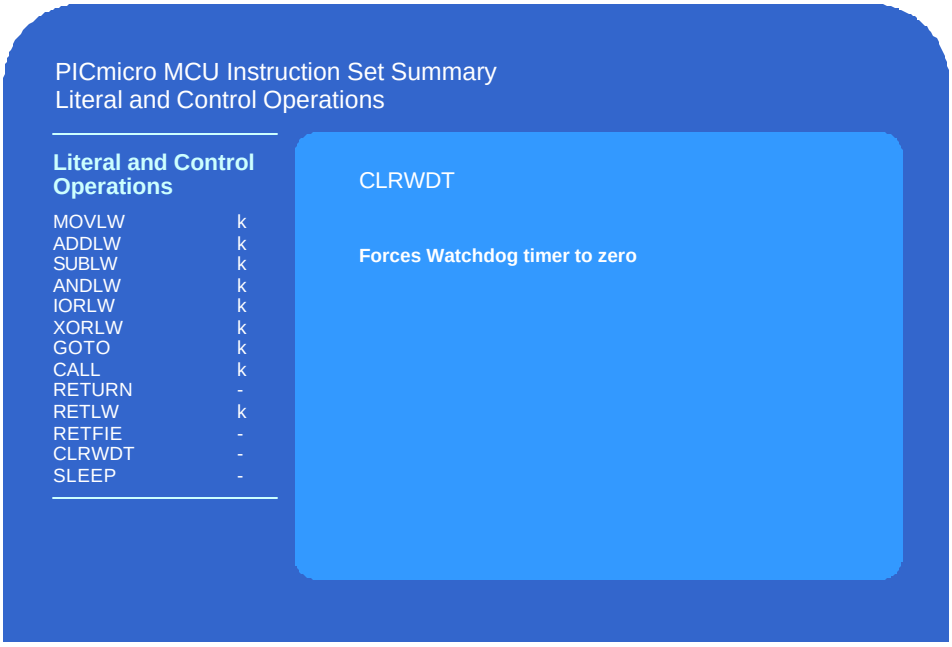
The second method of returning from a subroutine is to use the Return with a Literal in W instruction [RETLW]. This instruction works the same as the previous instruction, except that a literal value is returned from the subroutine stored in the W register. The literal value is an 8 bit value and is very useful for doing table lookups.

Slide 40. RETFIE Instruction



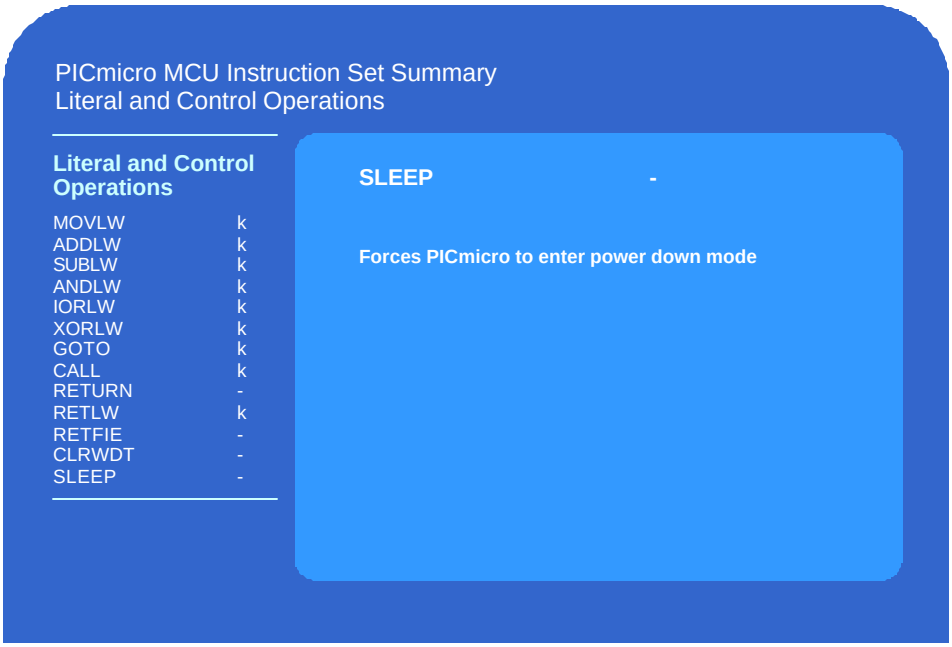
The last return instruction is the Return from Subroutine with Interrupt Enabled [RETFIE]. This instruction enables the global interrupt enable [GIE] bit upon execution

Slide 41. CLRWDT Instruction



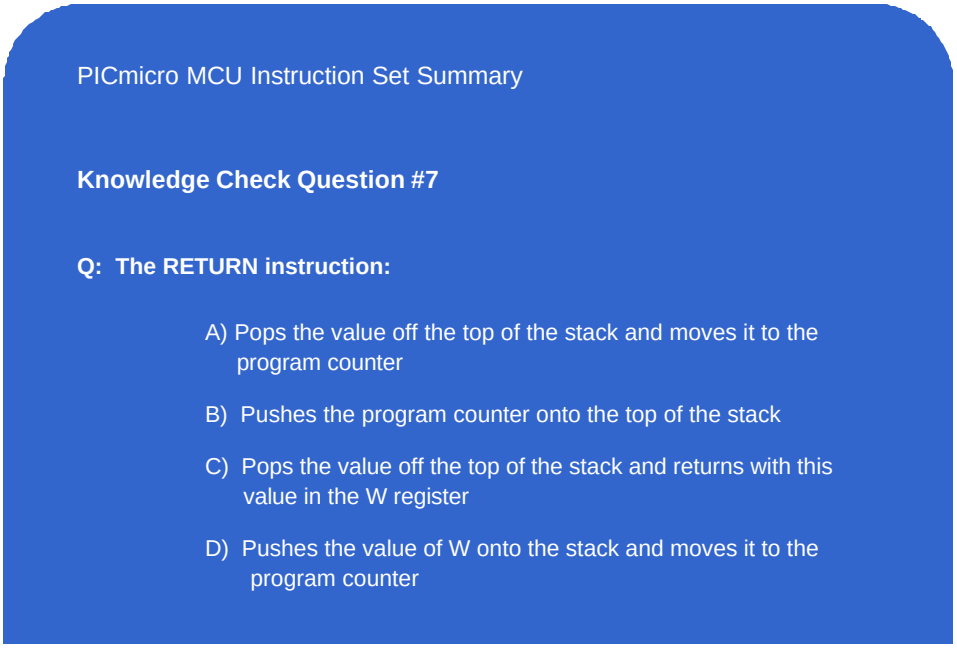
The Clear Watchdog timer instruction [CLRWDT] resets the watch dog timer back to zero.

Slide 42. SLEEP Instruction

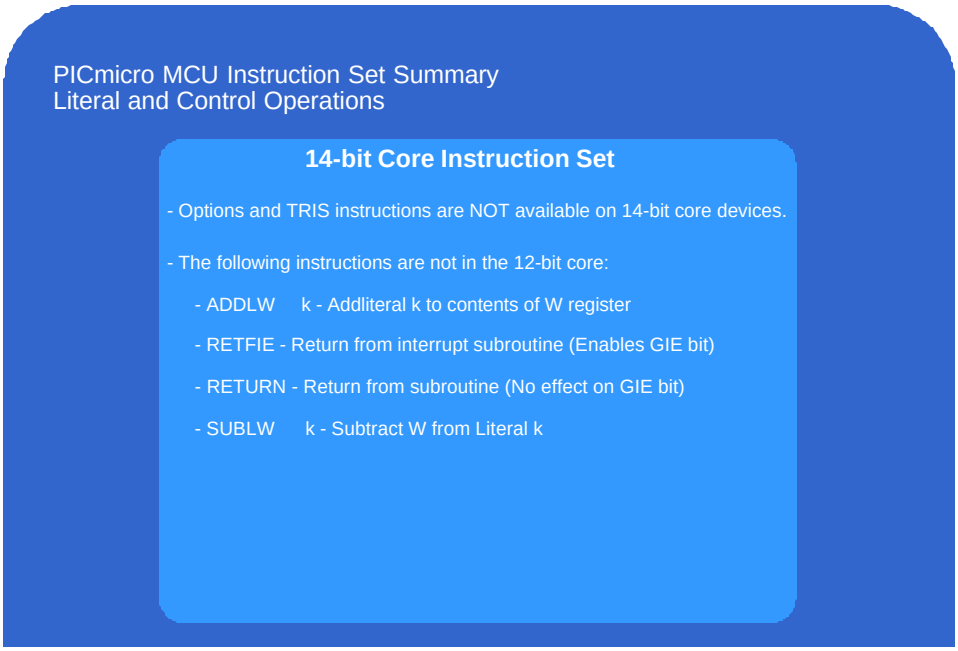


The SLEEP instruction will cause the PICmicro to enter the power down mode with the oscillator stopped.

Slide 43. Knowledge Check 7



Slide 44. 14-Bit Core Instructions



Now lets talk about some of the things you need to be aware of if you have been writing code with the x12 bit architecture and are migrating to the x14 instruction set. There are two instructions on the x12 architecture, the Option instruction [OPTION] and the Tristate instruction [TRIS] which are not implemented in hardware on the x14 architecture, but they are available as interpreted commands via the assembler. Because of this, it is recommended that you *not* use these instructions on x14 devices as these interpreted commands may not be implemented in future devices. The reason for these differences is that these 2 instructions access 'buried registers' on the x12 architecture, which are the Option register and the Tristate register. By 'buried registers' we mean registers that are not accessible through the standard register map.

On the x12 architecture, the tri state or TRIS instruction allows you to make an I/O pin an input or an output on the fly based on the contents of the W register. On the x14 core, as you may have heard in the x14 architecture presentation, the tri state registers and the Option register are not buried, but are actually regular registers within the architecture. Therefore, you can access the tri state registers and the Option register just by moving W to them directly or executing bit set or bit clear instructions on individual bits just like you can on any other register.

Now lets look at the 4 new instructions that were gained on the x14 instruction set by going from 33 to 35 instructions and removing the OPTION and Tri-State instructions.

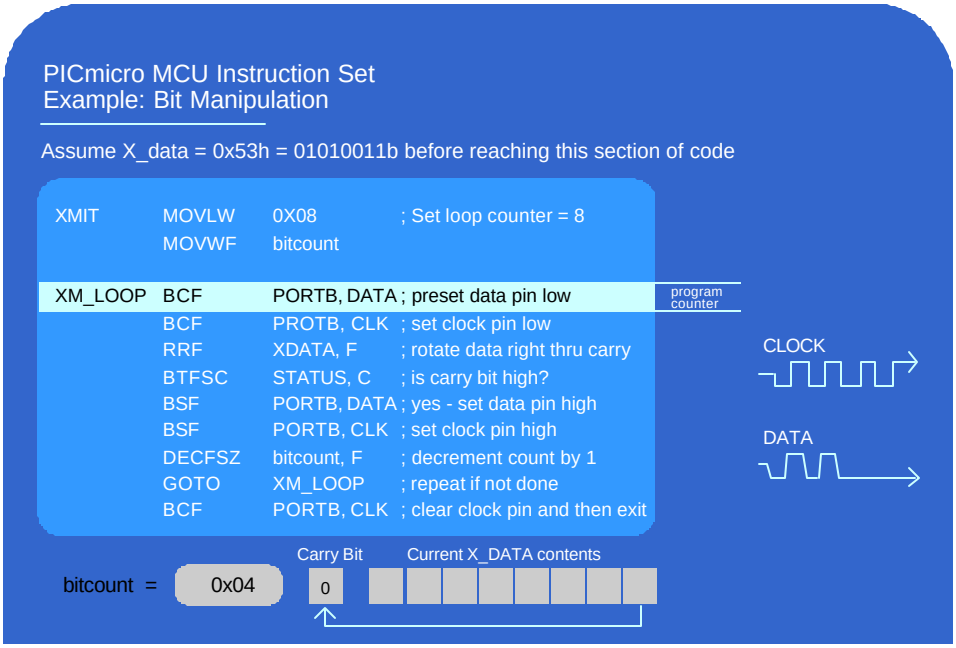
First is the Add Literal to W instruction [ADDLW], which takes the 8 bit literal value 'k' in the instruction and adds it to the contents of W.

next is the Return From Subroutine with Interrupt Enabled instruction [RETFIE] which allows return from subroutine or from an interrupt and at the same time enable the interrupts which enables the GIE bit in the interrupt status register.

Next is the [RETURN] instruction, which is a generic return you from subroutine instruction and has no effect on the GIE bit and no effect on the W register.

And finally we have subtract W from Literal [SUBLW], which takes contents of W and subtracts it from the literal value "k" and then puts the result back in the W register.

Slide 45. Bit Manipulation Example



Now were going to go through an example in detail that shows the power of the PICmicro instruction set. Our example will demonstrate how to execute the synchronous serial transmission of 8 bits of data. The term “synchronous transmission” means that we will use both a clock and a data line to transmit the data and the device at the receiving end will wait for the rising edge of the clock line to read the state of the data line.

The 8 bits of data that we want to transmit are stored a register that we have defined as X_data, and for this example we are going to use the value 0x53h as the contents of X_data. We are going to transmit the contents of X_data using two of the I/O pins that we have defined as CLK and DATA.

The first two instructions at the label X_mit sets up the program to go through the loop 8 times by moving the literal value of 0x08h into the W register and then moving W into the file register called bit_count. This is the register we are going to use to make sure we go through the loop 8 times.

The next line has the label XM _LOOP which is the top of the loop that will take each bit of X_data in sequence and transmit it using the CLK and DATA pins. The first thing we do in the loop is preset the DATA pin and the CLK pin to zero using the Bit Clear instructions on these two pins, both of which we have defined in port B. It should be noted here that we set the data pin low at this point for every bit that we transmit, even though we will set it back high later in the loop for data bits that are a logic ones. Because of this, even if you transmit two ones in a row, you will still see the data line go low in-between the two bits being transmitted. You could easily change the implementation of this loop so this does not occur, but it would take more code to do it. And since this is a synchronous transmission, the fact that the data line goes low before each data bit is transmitted does not matter as long as the data line is at the proper state and stable before the clock line goes high.

Next we get to the rotate right instruction [RRF] where we are going to rotate the current contents of the X_data register one bit to the right and then store the new results back into X_data. As you can see in our X_data register that bit 0 from X_data is pushed off the right and is rotated around into the carry bit.

Now we are going to do a bit test on the carry bit within the status register by using BTFSC instruction. Since the carry bit is now a one, we will not skip the next instruction and we drop down to the next line where we set the Data pin high using the BSF instruction.

We now have valid data on the data pin, so we go to the next instruction where we use the BSF instruction to set the clock pin to a one. The purpose of this is to tell the device receiving this data that the signal on the data pin is now valid, so the receiver can read this data bit and determine if it is a zero or a one. The fact that the receiver waits for the rising clock to sample the data line is what makes this a synchronous transmission.

Now we have transmitted a bit of data, the next thing to do is to see if there are any more bits of data that need to be transmitted. We use the DECFSZ instruction on the register we called bit_count which we're using to keep track of how many bits we have transmitted. Notice that we used the letter "F" as our destination bit so after we decrement the bit_count register, the result will be stored back into bit_count instead of the W register. So at this point, we see that the value of the bit_count register is 0x07h which is not zero, so we will not skip the next instruction and instead we land on the GOTO instruction, which will take us back to the top of the loop.

So, we set our data line low, then the clock line low and come to our rotate instruction again. Each bit is rotated to the right again and just like last time, we have a one that gets rotated around into the carry bit. We check the carry bit, and just like last time, the bit is high so we don't skip the next instruction. We set the DATA pin high and then set the CLK pin high, and we come to the decrement instruction and where our bit count variable goes from 7 to 6. We have still not finished all our loops so we will not skip the next instruction and instead we land on the GOTO instruction, and again we go back to the top of the loop.

This time, after the rotate is finished, you see that we now have a zero in the carry bit, so we *do* skip over the next instruction which means that the DATA pin will stay low for this bit. We decrement the bit_count variable down to 5 and go to the top of the loop once again.

The DATA pin is already low so it will stay low, and after the rotate is finished, you see that again we now have a zero in the carry bit, so we skip over the next instruction which means that the DATA pin will continue to stay low. We decrement the bit_count variable down to 4 and go to the top of the loop once again.

At this point we have transmitted 4 of our 8 bits, and eventually, the loop counter will decrement all the way down zero, at which time the program will skip the GOTO instruction and execute the last bit clear instruction, which pulls the clock pin low allowing us to exit and go forward to the next instruction in our program.

Slide 46. Knowledge Check 8

PICmicro MCU Instruction Set Summary

Knowledge Check Question #8

Q: In the Bit Manipulation Example that we just went through, the DECFSZ instruction is used to:

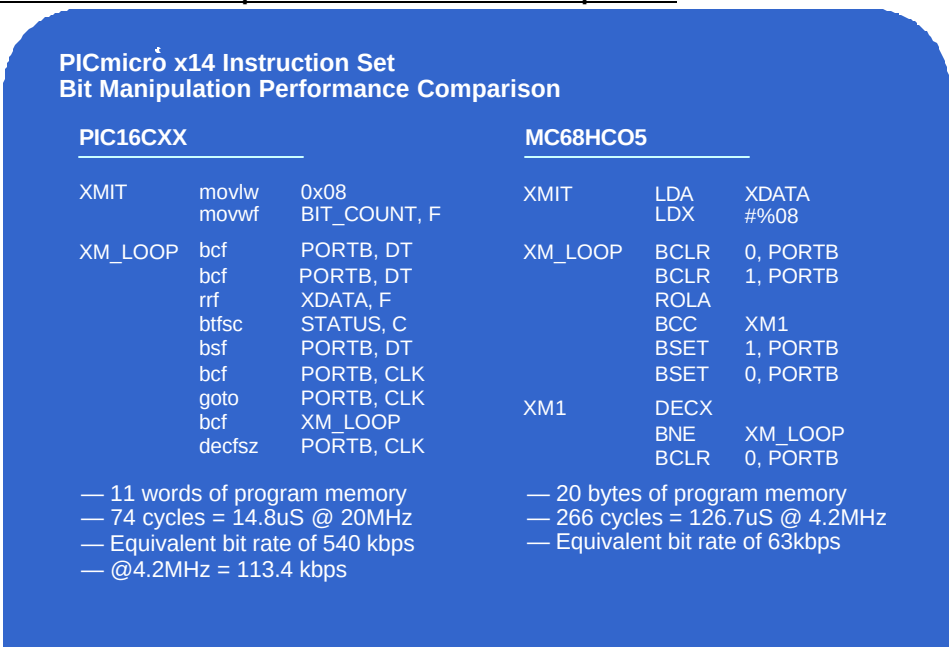
A) Is used to create a delay loop

B) Is used to determine if a 1 or a 0 should be transmitted each time through the loop

C) Is used to set the data pin low each time we go through the loop

D) Is used to control how many bits are transmitted

SLIDE 47. Bit Manipulation Performance Comparison



In order to demonstrate the power and performance of the PICmicro instruction set, we will show a comparison between the PICmicro x14 instruction set on the left and on the right, the 68HC05 which is another 8 bit microcontroller architecture. The code segments for each of these controllers accomplishes exactly the same thing, in fact we are using the example that we just went through which is the synchronous transmission of 8 bits.

If you look at the summary tables under each of the sections of code, you can see that by using the PICmicro instruction set, it takes 11 words of program memory to vs. 20 bytes on 68HC05.

On the PICmicro side, after you go through the loop 8 times, these 11 words of program memory take 74 cycles, which at a clock rate of 20 MHz is 14.8 microseconds. That's an equivalent bit rate of 540 kilobits per second. You can see the power and performance you get from the instruction set using single word and single cycle instructions, not only reducing the amount of program memory required, but also making things happen a lot faster.

We can't do a side by side comparison at this clock rate because at the time of creating this presentation, the fastest speed available on the 68HC05 was 4.2Mhz, so lets do the comparison at that speed. As you can see in the summary table under the 68HC05 code that this controller requires 266 cycles to execute the program. At a clock rate of 4.2 MHz, this equates to 126.7 microseconds or an equivalent bit rate of 63 kilobits per second. If you were to run the PICmicro at the same clock rate of 4.2MHz, the equivalent bit rate would be 113.4 kilobits per second.

If you run the PICmicro architecture and decrease the clock rate down to 2.33Mhz, you can achieve the same bit rate of 63 kilobits per second that you did by running the 68HC05 at 4.2Mhz. This means you can achieve the same speed performance with a lot less power and hence a lot less noise in your system.

Slide 48. Closing Slide

Thank you for your time looking into the PICmicro x14 instruction set. We hope you found this presentation interesting and worthwhile. If you have comments about this presentation or any other topic concerning Microchip's eLearning Program, you can send your comments by clicking on the "Feedback" link on the left side of this screen.