Text Script for "PICmicro® x14 Architecture"

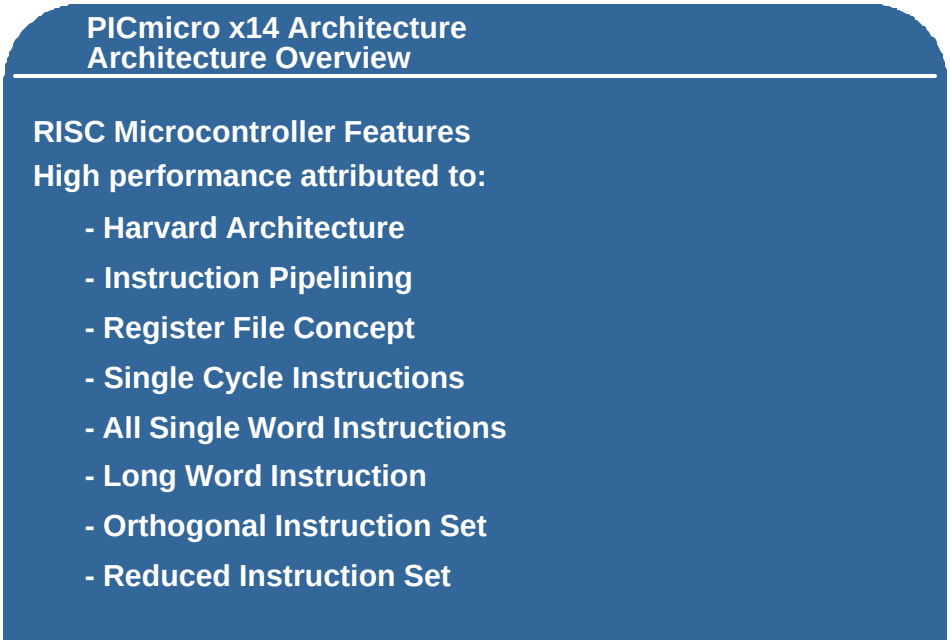
Table of Contents

Slide 1: Title Slide

Thank you for joining the Microchip Technology Inc. PICmicro x14 Architecture class.

This class is intended for those new to PICmicro products or requiring a refresher on PICmicro MCU architecture. It is recommended you have an electrical engineering background or some experience and knowledge of basic electronics. Knowledge of microcontrollers is beneficial.

Slide 2: Architecture Overview



The PICmicro MCU family is based on a RISC architecture, RISC standing for Reduced Instruction Set Computer. High performance is accomplished as a result of the Harvard architecture, which has separate data and address busses and two separate memory spaces.

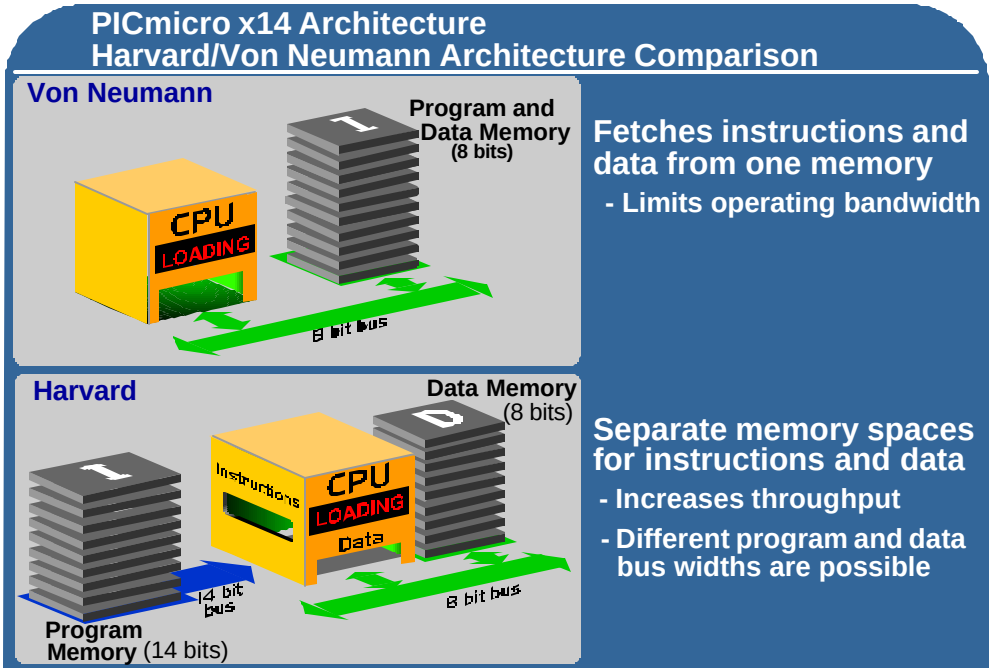
Instruction pipelining gives the capability of higher speed while the register file concept allows you to do the same thing with the register as you would with a piece of RAM.

Single cycle instructions allow high speed code execution, and single word instructions improve the throughput and reduce the amount of program memory space required.

The long word instruction gives the capability of holding immediate data in the same line of code as the instruction itself, which leads to a reduced instruction set which is very orthogonal or symmetric. This means that for every Set instruction there is a Clear instruction, for every Increment there is a Decrement and so on.

You can perform all instructions with any of the registers, so there are no special functions or special instructions for doing operations on the carry or handling any I/O peripherals. They all use the same instruction set.

Slide 3: Comparison between Harvard and Von Neumann Architectures



Lets compare the Harvard architecture to the Von Neumann architecture, which is common in other 8-bit microcontrollers.

The Von Neumann architecture has only one memory space to store both program and data memory. This means that it is necessary to fetch instructions and data from the same memory space, which limits your bandwidth because you can only transmit one piece of data or one instruction at one time.

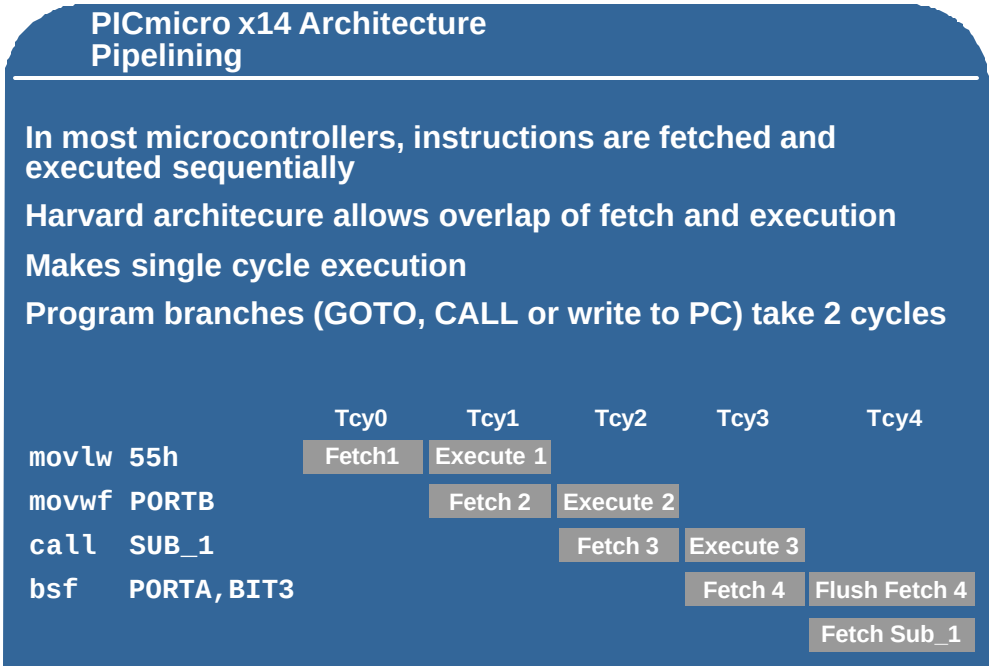
With the Harvard architecture, there are two separate memory spaces, one for instructions and one for data. You can increase your throughput because while fetching the next instruction you can still be writing the result from the previous instruction. The other advantage is that since this is an 8-bit microcontroller, the data memory is 8-bits wide, but the program memory can be any width you choose. On existing PICmicro architectures, the program memory used is either 12-bits, 14-bits, or 16-bits wide. The architecture we will be discussing today is the x14 architecture, which has a 14 bit wide program memory space.

Slide 4: Knowledge Check 1

Question: Instructions on the PICmicro x14 architecture:

1) All execute in one instruction cycle
2) **Only take one location in program memory**
3) Are multi byte
4) Have great mnemonics

Slide 5: Pipelining



Let's look first at instruction pipelining. In most 8 bit microcontrollers, instructions have to be fetched and executed sequentially because there is only one data path. Pipelining provides the capability of overlapping the fetch and execute steps to increase the speed at which the instructions are executed. We will go through an example here that shows how we can get single cycle execution and overlap both fetching and executing instructions.

In this example, we will execute 4 instructions, and we'll observe how the instructions are fetched and executed across five instruction cycles. The instruction cycles are shown as Tcy0 through Tcy4. You will see that each of the 4 instructions has to be fetched first, but they are actually pre-fetched and then executed during the following instruction cycle.

Every instruction takes one instruction cycle, except for instructions that modify the program counter. This occurs with the jump and jump to sub-routine instructions, or any instruction that writes directly to the program counter. These instructions will all take two instruction cycles, as we will see in the example.

The first instruction we will perform is MOVLW 55h. Since this is the first instruction to be executed, there is nothing in the pipeline to pre-fetch, so we perform a fetch to get the instruction into the instruction register during Tcy0.

The MOVLW instruction will move the literal contents of the instruction, (in this case 55h) into the W register. While this instruction is executing during Tcy1, we also pre-fetch the next instruction. This is a Move W to File (MOVWF) instruction which in this case will move the contents of the W register into the register PORTB, which is an I/O port. At the end of Tcy1 the W register will have the value 55h in it loaded from the instruction.

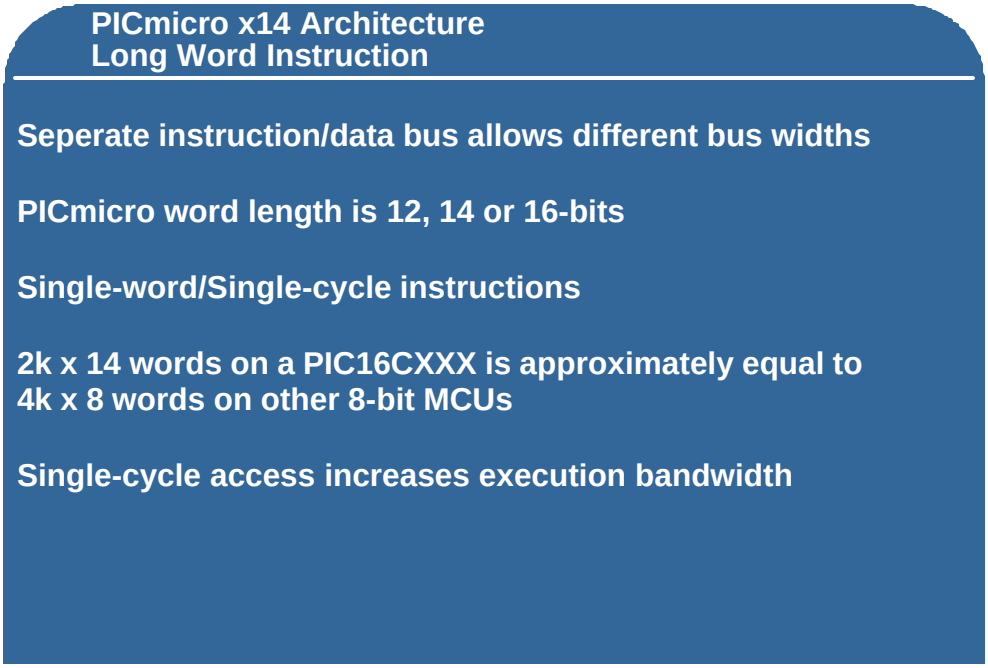
During Tcy2, we execute the MOVWF instruction which will copy the contents of the W register to PORTB. During this same instruction cycle, we pre-fetch the next instruction which is the third instruction in the sequence. Note that at the end of Tcy2, the W register still contains 55h and PORTB also contains 55h. The MOVWF always retains the original value in the W register. The next instruction pre-fetched in Tcy2 is CALL SUB_1 which tells the program to jump to the sub-routine where the label of the sub-routine is SUB_1.

During Tcy3, we execute the jump to sub-routine instruction and at the same time, we pre-fetch the next instruction in the sequence. In this case, the next instruction is Bit Set File Port A, Bit 3. In other words take Bit 3 of PORTA and set it high. However, while we are pre-fetching that instruction we are also completing the execution of the CALL SUB_1 instruction. This reloads the program counter with the address location SUB_1, which will start a different instruction sequence.

In Tcy4, we start the new instruction sequence which means we have to throw away or flush the fourth instruction we fetched which was BSF PORT A, BIT3. By doing this we are reloading the pipeline and the BSF instruction is *not* executed by the ALU while the instruction at location SUB_1 is fetched.

This shows that whenever you do anything that modifies the program counter, whether it's a jump to sub-routine or just a plain jump, the ALU will automatically flush the instruction that you pre-fetched and throw it away. When a subroutine is used, the RETURN instruction will cause the program to return to the original program sequence pointing at the instruction after the CALL, in this case the BSF PORTA, BIT3.

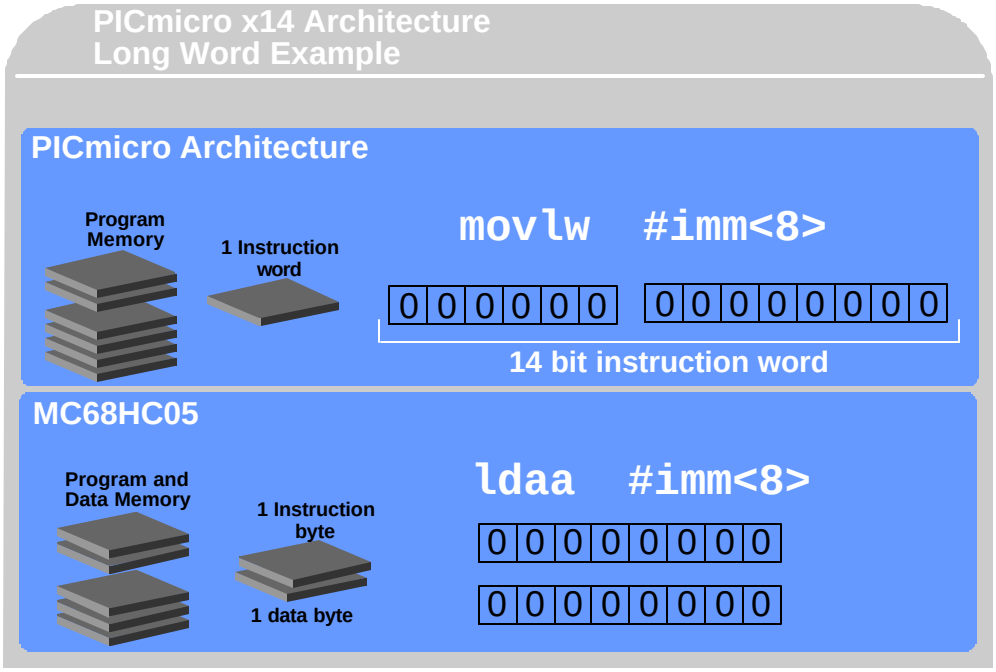
Slide 6: Long Word Instruction



Now let's look at the Long Word Instruction. The Long Word Instruction allows different bus widths to be used since the instruction and data busses are separate. As we said earlier, the PICmicro family has either 12, 14 or 16 bit wide instruction busses, each with single word, single cycle instructions.

Because of the single word instruction the power is increased in the instruction itself. As a rule of thumb, two 2k x 14 words of program memory on a PIC16CXX device is approximately equivalent to 4k x 8 words on other 8-bit microcontrollers. The single cycle access increases the execution bandwidth so that we can do things very quickly without using a lot of program memory space.

Slide 7: Example of Long Word Instruction



This shows an example of the Long Word Instruction, the MOVLW #IMM, which loads the immediate value designated here as #IMM into W. In this architecture, the immediate value is always an 8-bit number. You can see the constant here is 8-bits so we can store 8-bit immediate values in the 14-bit wide instruction word. The 4-bits used for the op-code in this case, MOVLW, translate into 1100b.

If you look at the same instruction on a Motorola 68HC05, the instruction equivalent would be LDAA #IMM. LDAA is defined as 1000 0110 which is the op code for Load Accumulator A. Then the next byte in the memory space stores the immediate value or the constant value that is being used. You can see this now uses 2 lines of program memory instead of 1, albeit these 2 lines are only 8-bits long instead of 1 word at 14-bits long.

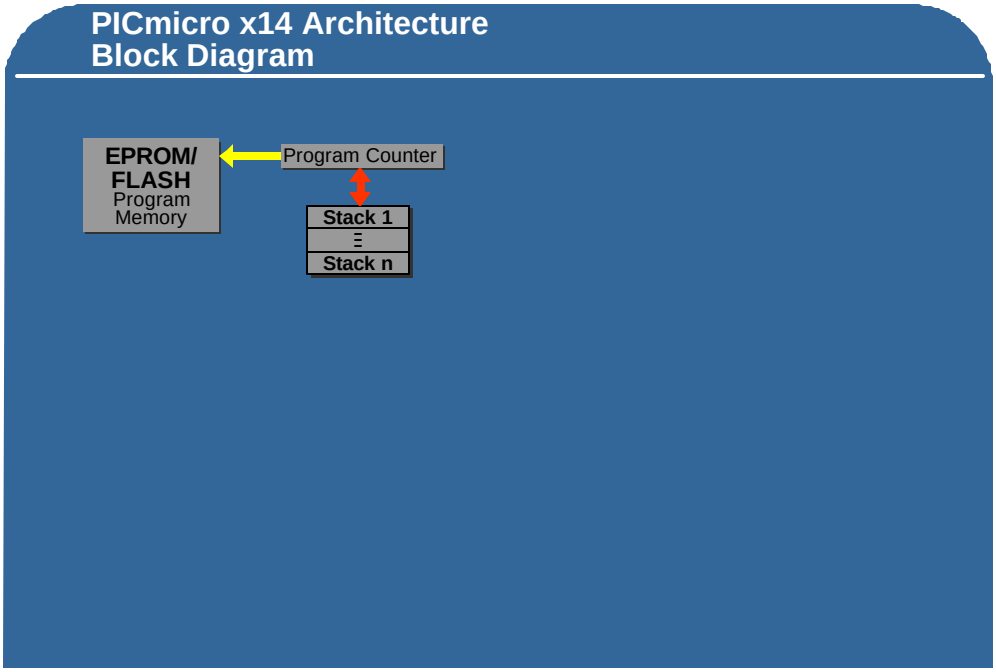
When you count the number of instruction locations used to perform a specific function, this would count as 2 where on the PICmicro family it would only count as 1. This is why 2k x 14 words on the PICmicro MCU can be equivalent to almost 4k x 8 words on the 68HC05, depending on the application being developed.

Slide 8: Knowledge Check 2

Question: **Which one of the following instructions takes two instruction cycles to execute?**

1) `movlw 0X23`
2) `addwf temp,W`
3) **`goto loop_1`**
4) `bcf PORTA,BIT1`

Slide 9: Block Diagram



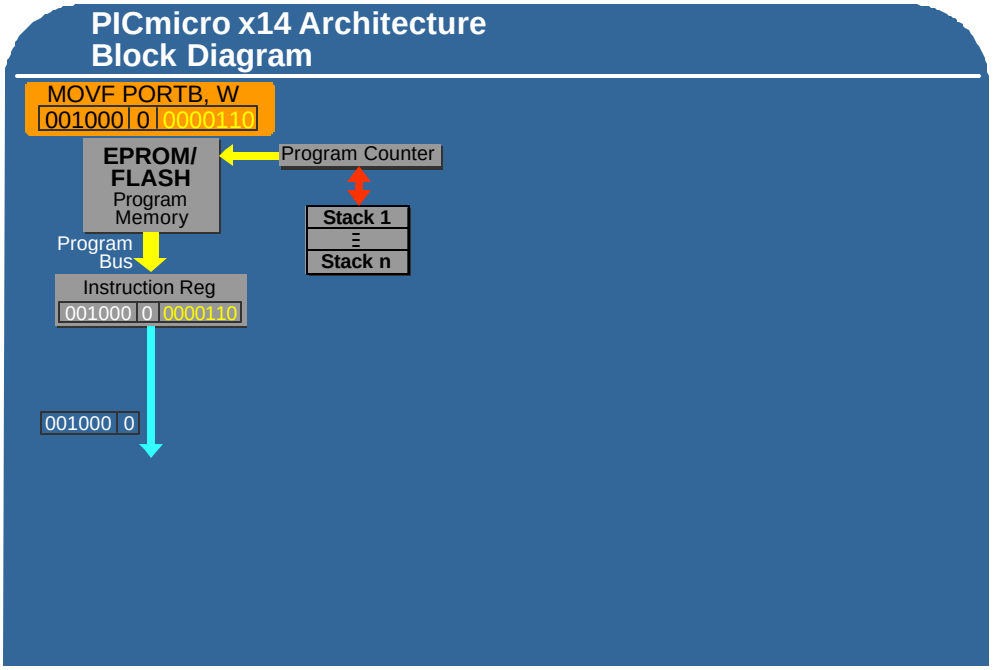
Let's now take a look at the block diagram of the PICmicro x14 architecture starting with the program memory space. EPROM memory or Flash EEPROM program memory space is addressed from the program counter. The program counter on the x14 architecture is 13 bits long.

You will notice the program counter has a stack associated with it. This is a hardware stack and on the x14 architecture it is 8 levels deep. Note that this is *not* a stack used to pass parameters. This stack can only be used to store the program counter when you jump to a sub-routine or when an interrupt occurs.

When either of these events takes place, the current program counter plus 1 is pushed into the stack. This allows the program counter itself to be reloaded with the address of the new instruction that is to be executed.

In the case of a jump to subroutine instruction this address is embedded in the CALL instruction. For the case of the interrupt, the program counter is set to 04h which is the interrupt vector address.

Slide 10: Block Diagram



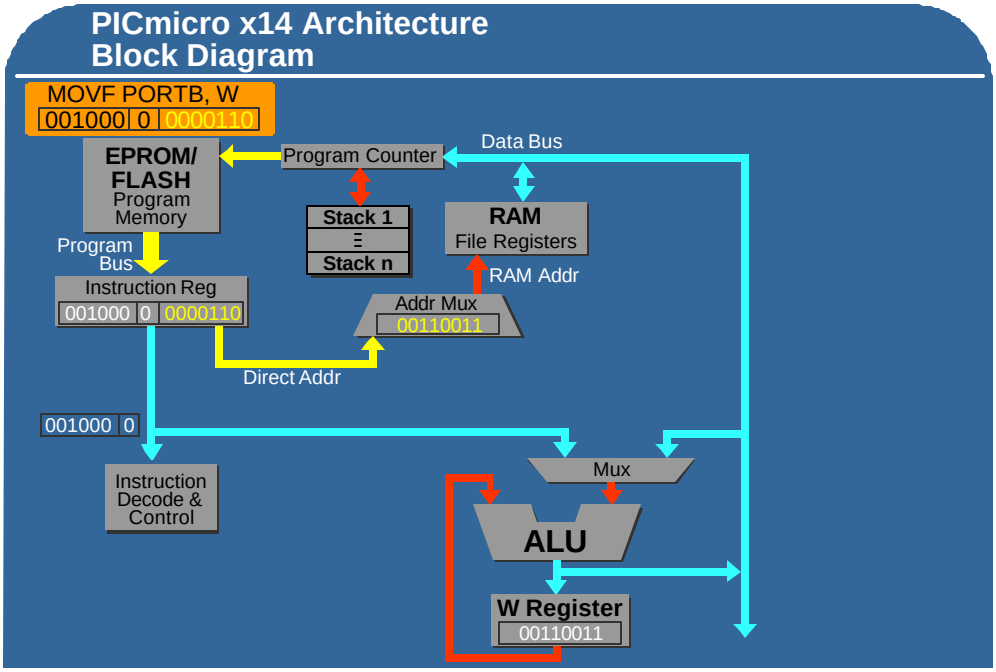
We will now follow the instruction MOVF PORTB,W through the architecture. This instruction will take the contents of PORTB and copy them into the W register. You will see the op code underneath the instruction and the most significant 6 bits of op code defines the MOVF instruction. The next bit is the destination bit, which in this case is a 0 and means we will store the results in W. The least significant 7-bits define the location of the register that we are going to get the data from that we move to W. In this case, the value is address 0x06 which is PORTB on the device we are using.

The destination bit defines where results of the instruction will go. Since there is only one bit, there are only two options for the destination address, either into the W register or back into the register defined in the instruction itself, in this case PORTB.

The 14 bit op code is contained in the instruction that the program counter was pointing to, and is the information that comes out of the program bus to the instruction register.

Notice that the address (highlighted in yellow) is the address of the **data memory space** that the operation will be carried out on and is either SRAM or EEPROM. The **program memory** is EPROM, Flash EEPROM or could even be ROM based. These two memory spaces are totally separate from each other and should not be confused.

Slide 11: Block Diagram

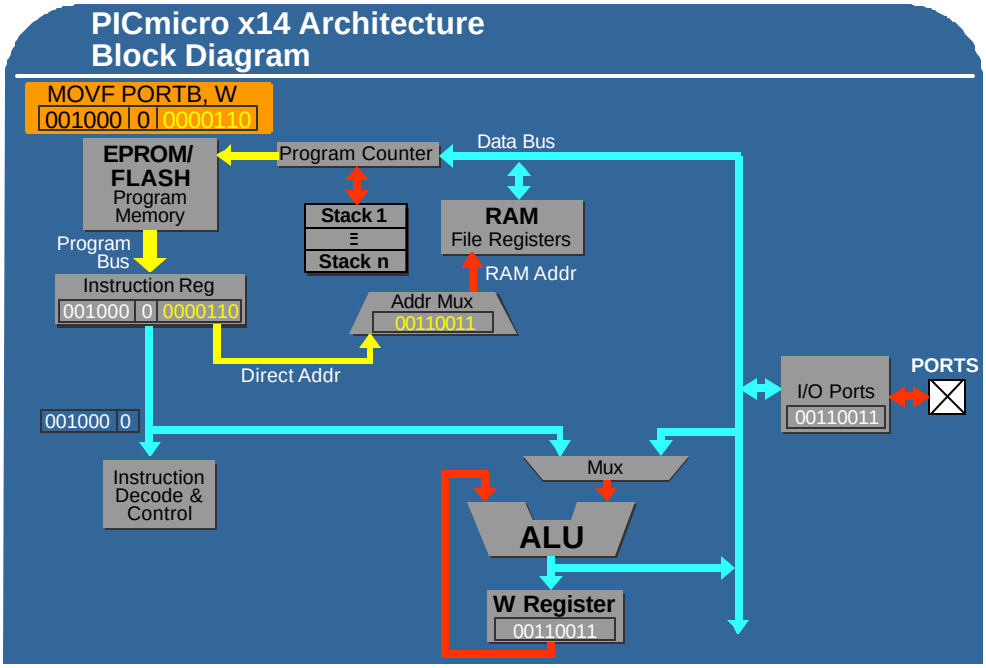


In this case the top 7-bits of the instruction are decoded and tell the processor that it needs to do a move-file instruction and put the result in the W register. The address goes into the address multiplexer and since we are using direct addressing in this example, PORTB is used to address the RAM or the file register space. Notice the W register is a part of the ALU on the x14 architecture. It does not have a separate address space in this architecture.

On the x16 architecture it appears as a data memory register in the memory map as well as part of the ALU. However on the x14 and x12 architectures the W register is buried inside the ALU. Therefore in order to move data into W, you must go through the ALU. Conversely to move data from W to the data memory space you must go through the ALU using the red feedback line into the ALU itself.

The immediate instructions take the 8-bit literal value from the instruction along the blue line into the multiplexer above the ALU using the path that takes data from the instruction register. This path then takes the 8-bit literal value through the ALU and it deposits it in the W register. This gives a path of data from program memory into W which is on the data memory side of the architecture.

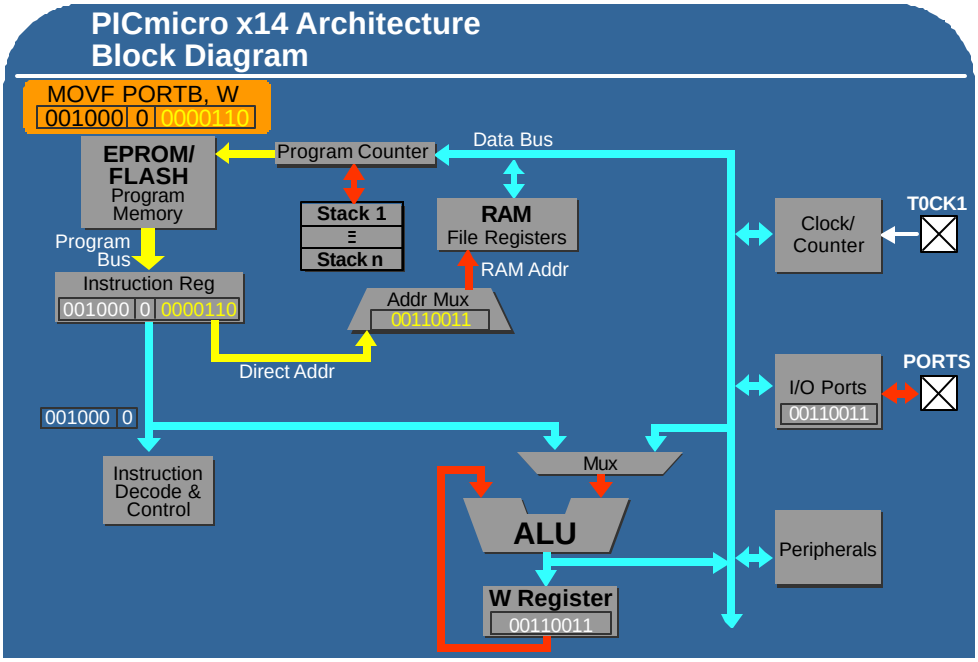
Slide 12: Block Diagram



Since the I/O ports sit on the same data bus as the file registers and RAM locations, you can address I/O ports and RAM using exactly the same instructions that you would for file registers. Just change the address and the instruction will execute on the RAM location or the I/O port, because they sit on the same 8-bit data bus.

Notice the 01 pattern on the pins of I/O PORTB are copied into the W register because the data has been read from the I/O pins onto the data bus, through the multiplexer, into the ALU, and hence into the W register.

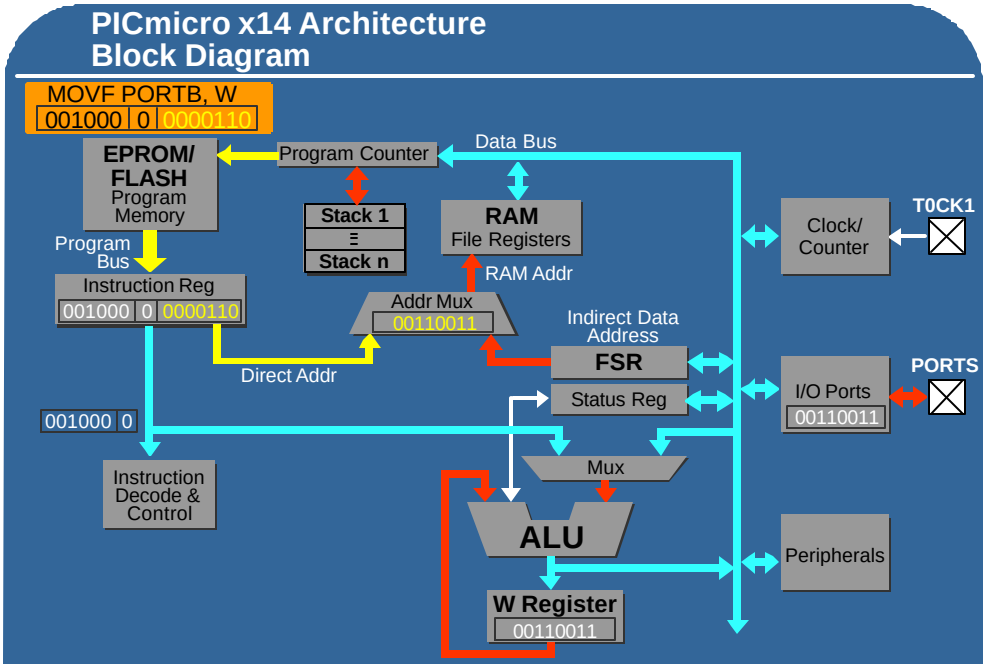
Slide 13: Block Diagram



This completes this particular instruction, but to look at the remainder of the block diagram, you can see all the peripherals including the timers, clocks and counters sit on the same data bus, although they all have different addresses.

The peripherals are accessed through the same data bus as the RAM, file registers and I/O ports, and even the program counter sits on that same data bus. Now you can take the contents of the W register and move them to the program counter. Since it's an 8-bit data bus you can only do that on the lower 8-bits of the program counter and we will discuss this a little later when we look at doing computed gotos.

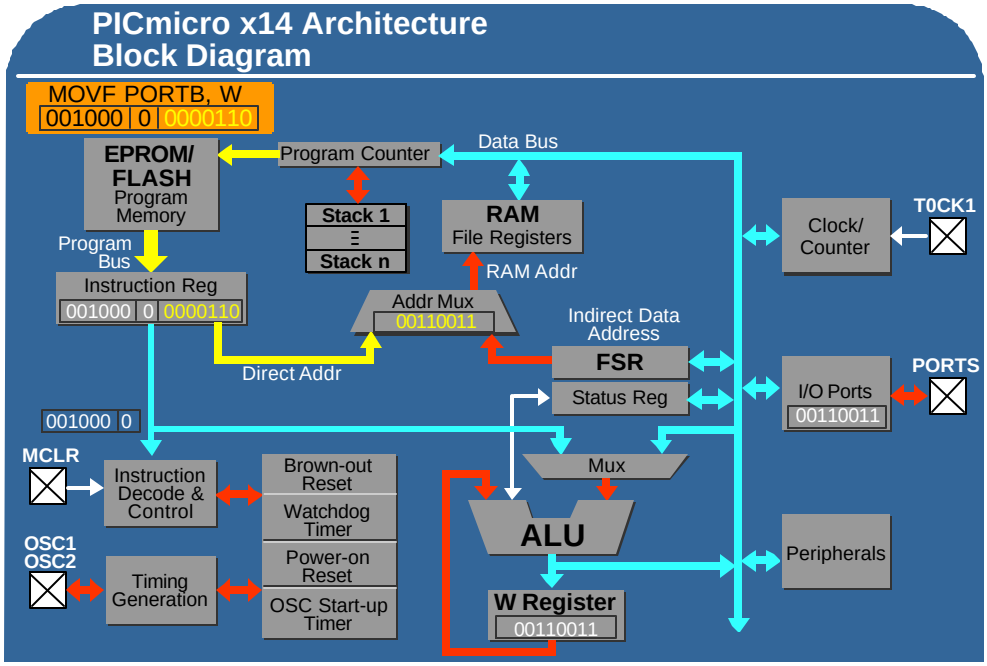
Slide 14: Block Diagram



The address multiplexer requires either a direct address or an indirect address and the output of the address multiplexer addresses a file register, a RAM location, an I/O port, a peripheral, a clock counter or even the program counter itself. You will notice there is another input to the address multiplexer, the address does not have to come from the instruction, it can come from the file select register, or FSR. This is the indirect data address pointer. We can take the contents of the file select register and use that as the address for the RAM, the I/O ports, the peripherals or the clock counters.

The other block that appears here is the status register. The status register is another register that sits on the data bus and gives the status of the ALU. We will discuss a few of the bits a little bit later in the presentation, but these bits define if the result of the last ALU operation was a 0, whether there was an overflow from a subtract or an add instruction, etc.

Slide 15: Block Diagram



Before we leave this section, lets briefly discuss the auxiliary features that are built into the PICmicro x14 architecture. These include the brown out reset, watch dog timer, power on reset and oscillator startup timer, all of which are generic functions available in the PICmicro x14 family of products.

The timing generation block generates the necessary timing signals for the device, which are derived from an external crystal oscillator, ceramic resonator or an internal RC oscillator inside the chip itself. This frequency is internally divided by four to generate the Tcy clock cycles we discussed earlier, four of which make one instruction cycle.

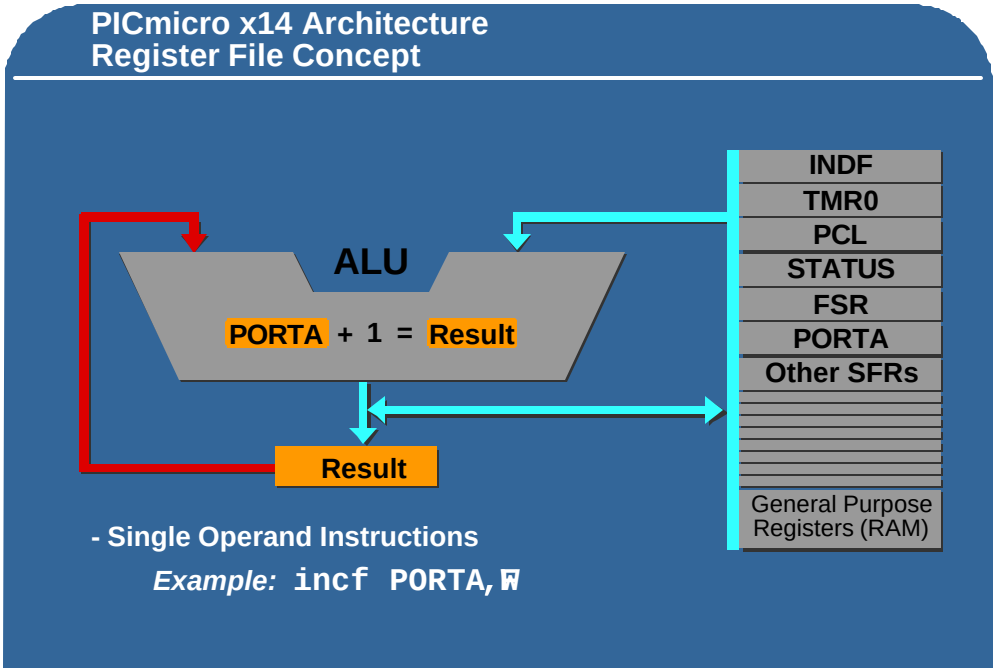
These functions, although described in the architecture, are not necessarily available on every part, so consult the product line card or the individual data sheet for each product to determine which items are available on specific devices.

Slide 16: Knowledge Check 3

Question: The hardware stack on the x14 architecture:

1) Can be as deep as you specify
2) Is very useful to pass parameters to a subroutine
3) **Can only be used for storing return addresses**
4) Is easy to read

Slide 17: Register File Concept



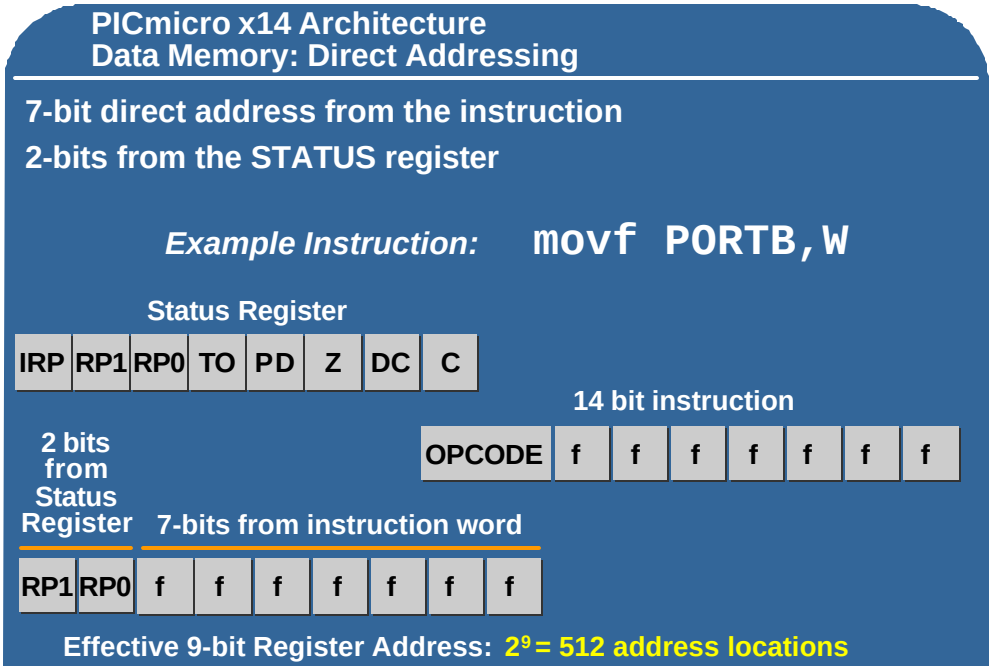
One of the features in the PICmicro architecture that makes it so powerful is the use of register files. The register file concept means that all RAM locations, peripherals and I/O ports are organized as a simple bank of registers. All instructions can operate on any register and you don't have to load things from RAM into a register to do functions on them.

For example, you can take the contents of the W register and add that to any other register in one instruction. It can be added to a general purpose register or it can be added to an I/O port. This is referred to as a two-operand instruction, which is discussed in more detail in the instruction set module. When using two operand instructions, one operand always comes from the file register bank and the other operand is always the W register.

If you are using a single operand instruction such as increment file (INCF), you can take the contents of the file you define into the ALU, increment the contents and write the result back out to the file itself, all in one instruction.

As discussed earlier, the destination bit can be used with most instructions to determine if the results of the instruction are stored back into the file register itself or stored in the W register. As an example, if we add the "comma F" to the end of our increment f instruction, the results of the instruction are are stored back into the file register, which is the same result had we just left the ", F" off the instruction. However, if add "comma W" onto the end of the instruction, this would take the contents of the file you defined, increment it and put the result in the W register without affecting the register you first read. This gives you two different options for storing the result of the instructions.

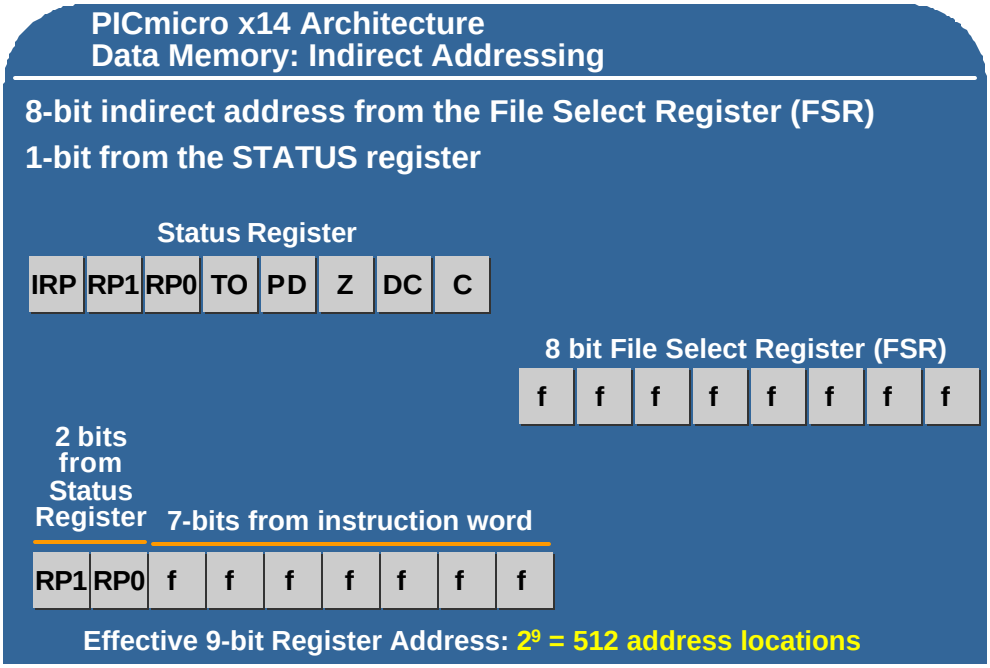
Slide 18: Data Memory: Direct Addressing



Let's look at direct addressing of the data memory space. An example of direct is the the MOVFW instruction that we saw earlier. You can see in the 14-bit instruction there are 7-bits of data that come from the instruction word itself. That gives us up to 128 different registers we can access directly.

If we want to access additional registers as we do on the products that have more than 128 bytes of RAM or registers available, then 2 bits in the status register are used. These 2 bits are referred to as RP0 and RP1. These 2 bits give us the capability of up to 4 different pages of RAM. By using these 2 bits, we increase the addressable region of our file set to 9 bits, which allows us to access four times the number of registers that the 7 bit field in the instruction itself gives us.

Slide 19: Data Memory: Indirect Addressing



Now we will discuss indirect addressing, which allows you to increment a pointer and very easily go through a table or list of data held in the data memory space.

When using indirect addressing, you have 9 bits of address space available. The lower eight bits of the indirect address come from the file select register or FSR, while the 9th bit comes from the Indirect Register Bank Pointer bit (IRP) in the status register. The IRP bit effectively gives you 2 'pages' of 8 bit address which allows access to 9 bits worth of address space. When using indirect addressing, you need to be sure that the IRP bit in the status register is set to the correct page.

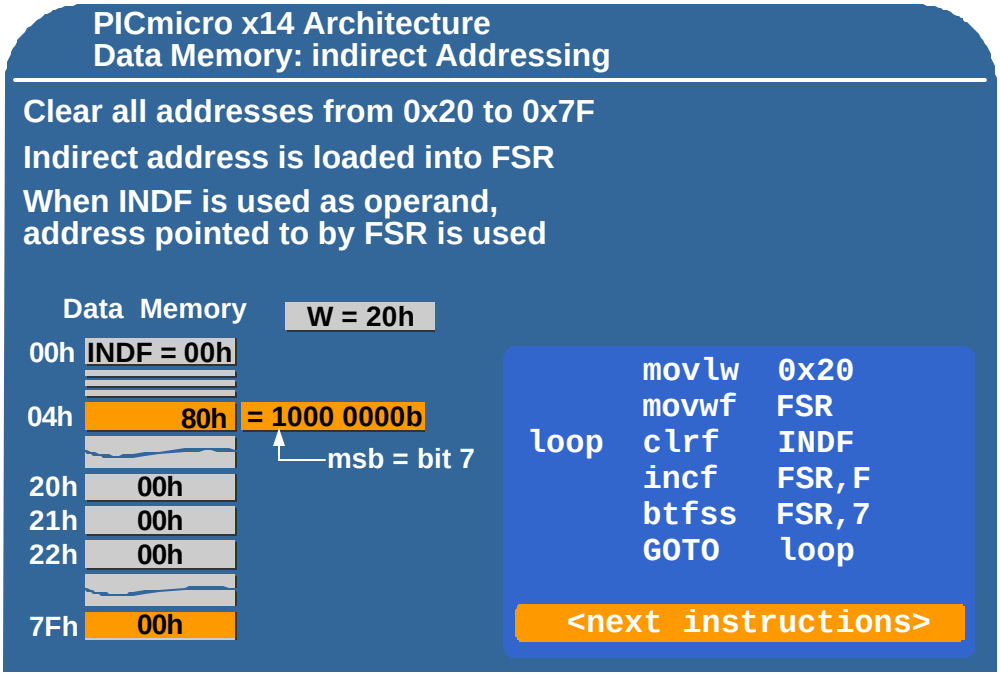
If you look at the status register, there are some other bits of importance. We have already looked at IRP bit, the register page one (RP1) and register page 0 (RP0) bits. TO is the time out bit and PD is the power down bit. These 2 bits are used in conjunction with one another to define what caused a reset condition, whether it was the result of a watchdog timer timeout, a reset signal on the master clear pin (MCLR) or a power down sequence. Likewise, we have the zero bit, the digit carry and the carry all used to indicate the status of the ALU at the end of each instruction.

Slide 20: Knowledge Check 4

Question: After an operation is completed, the result:

1) Always goes into W
2) Overwrites the instruction
3) **Is stored in either W or the source file**
4) Stays in the accumulator

Slide 21: Data Memory: Indirect Addressing Example



In this example, we've been assigned the problem of clearing all RAM locations from 0x20 to 0x7F. Because all the memory locations we need to clear are sequential, we can easily do this using indirect memory addressing.

We use the indirect addressing pointer to do this by loading the first address that we need to clear into the file select register, the FSR.

The FSR is loaded using two instructions, the first being the MOVLW 0x20. This instruction moves 20h into the W register because 20h is the first address that we are going to clear. The second instruction is the move w to file FSR, which takes the 20h currently in the W register and copies to the FSR. It should be noted here that although we are using the acronym FSR in our instruction, it is really the same as the absolute address 04h in the case of the x14 architecture.

Now we are going to go through the loop that actually clears the contents of all the registers between 0x20 and 0x7F.

The first instruction in the loop is the clear-file instruction and it clears the Indirect File Register, or INDF. Whenever INDF is used as the register name in an instruction, the decoder translates this to be the contents of register location 0x04 which, as we just pointed out, is the FSR. In other words, if you use the register name INDF in an instruction, the processor will go and get the contents of the FSR and use *that* value as the register address.

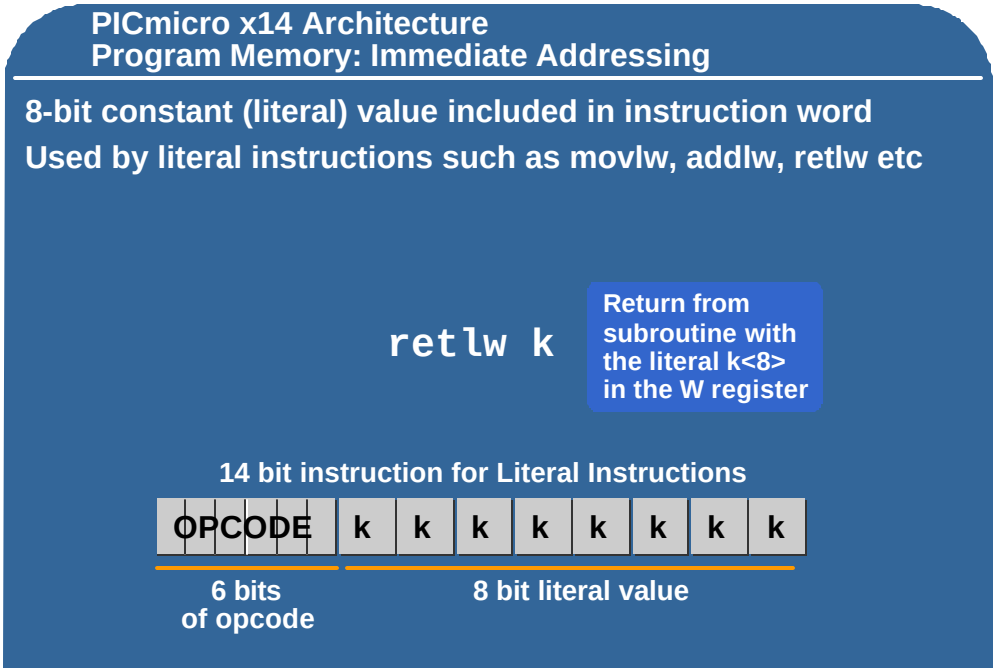
The processor now takes the contents of 0x20, throws it away and replaces it with 0x009. RAM location 20h now has 0 in it because when the instruction CLRF INDF was executed, the contents of the FSR was 0x20.

The next instruction is the increment file FSR with the destination set to f which will place the results back in the file register. This increments the FSR from 20h to 21h and puts the result back into the FSR, overwriting the previous contents.

The next instruction we execute is the BTFSS FSR,7. This instruction tests the most significant bit of the FSR to see if it has gone high yet. When this bit *does* go high, it indicates we have reached 80 hex indicating we have gone through and cleared all the registers we need to. If bit 7 is *not* a 1, it will execute the instruction immediately following the BTFSS instruction, which is a GOTO instruction. This will cause the program counter to jump back to the label "loop" and we will execute our loop all over again.

Now if you recall, the FSR currently has 0x21 in it so now when we execute the CLRF INDF instruction, we clear the contents of address 0x21 to all zeros. We then increment the FSR register again with the results going back into the FSR making it equal to 0x22 , then 0x23 and so on each time we pass through the loop. Eventually, the FSR will be incremented all the way up to 0x80 when bit 7 will go high. When that happens the Bit Test instruction will be true and we will skip over the GOTO instruction which causes us to exit the loop. At this point, we have cleared all the memory locations from 0x20 to 7F hex and are ready to execute the next instruction in our program.

Slide 22: Immediate Addressing

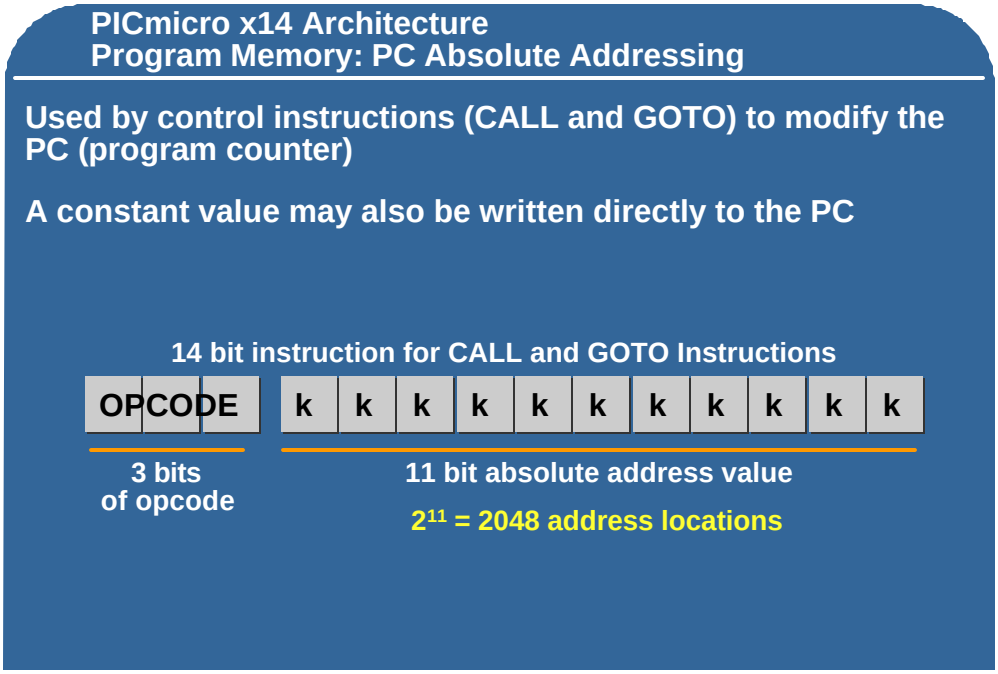


As mentioned earlier in this presentation, Immediate addressing in program memory space is done using the 8 bit immediate value included in the instruction.

You can see in the 14-bit instruction we have a 6-bit op code and an 8-bit literal value “k”. These instructions are used to manipulate the value in W with the 8-bit literal value. There are several instructions available that use the literal value. As an example, move literal to W (MOVLW) is a frequently used instruction that simply loads a specific value into W. This example shows how the value 0x79 is loaded immediately into the W register.

One very important instruction for table look-ups is return with a literal value in the W register (RETLW), which allows you to return from a subroutine with a literal value placed in W.

Slide 23: PC Absolute Addressing



Absolute addressing within the program memory space allows the programmer to insert control instructions. Control instructions are defined as GOTO or jump instructions, CALL or jump to subroutine instructions, or any other instruction that modifies the program counter itself. This includes instructions that write directly to the program counter, which will be discussed later in more detail.

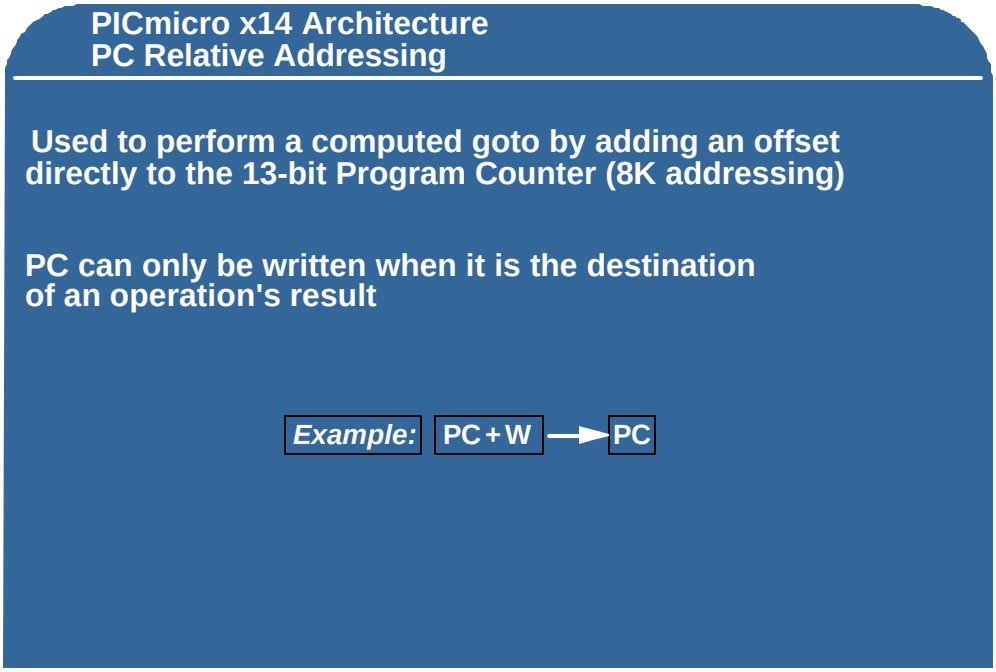
You will notice here in the 14-bit instruction, the absolute value is 11-bits with a 3-bit op code. This gives us the capability of jumping anywhere in a 2k address space within the program memory.

Slide 24: Knowledge Check 5

Question: Indirect addressing is used:

1) To move data from program memory to data memory
2) On program memory only
3) When the programmer is lost
4) **Only on the data memory space**

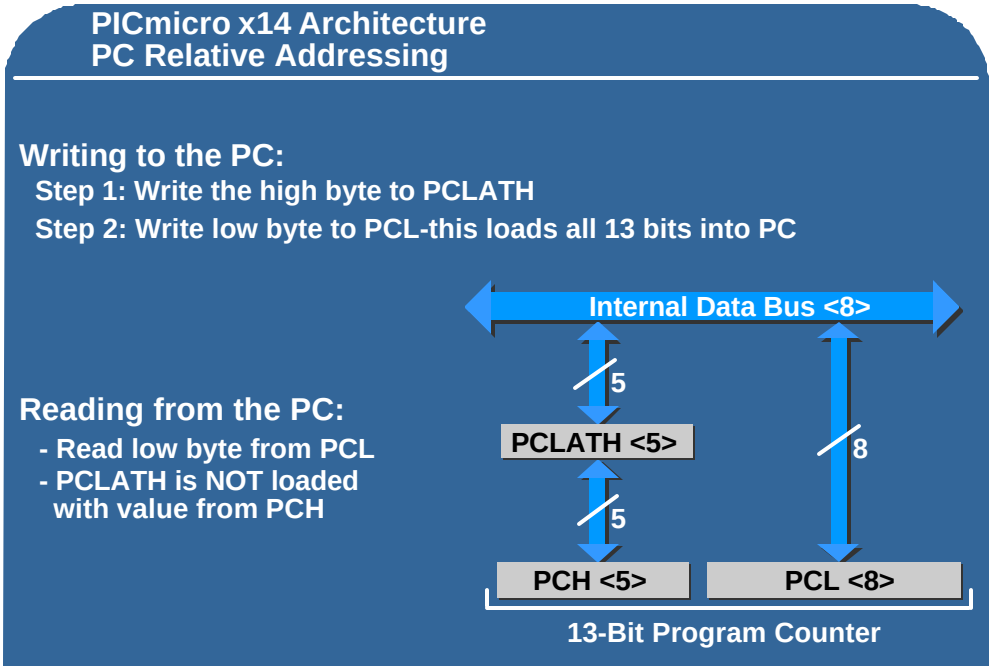
Slide 25: PC Relative Addressing



Relative addressing is done via the program counter itself. It is primarily used to do a computed goto where we take the 13-bit program counter and add an offset to it. In order to do this, however, you have to be very careful in how you write to the program counter because you only have 8-bits available to you. We can only do this when the program counter is the destination of an operations result.

For example, you could add the contents of W to the program counter and store the result back into the program counter, causing a two cycle instruction which is really a computed goto. This will be shown later in an example.

Slide 26: PC Relative Addressing



Lets go through the steps required to write to the program counter when using relative addressing.

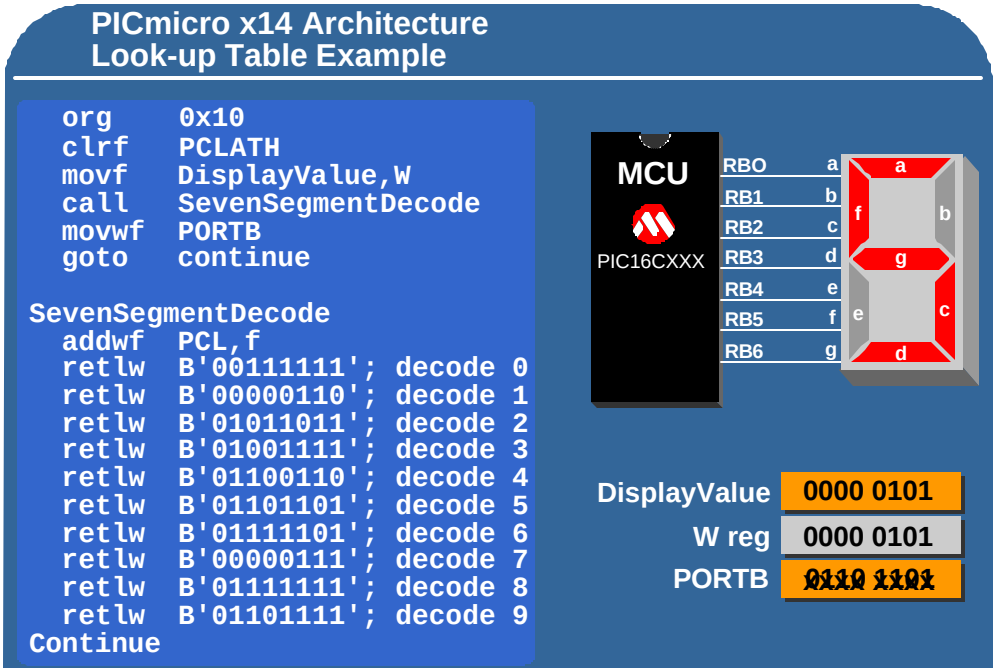
First of all, you have to write to Program Counter Latch High, which is defined as PCLATH. You will notice that we cannot write directly to PC high because if you did, the next instruction fetched would be in an unexpected location within the program memory space. All 13 bits of the PC must be written to simultaneously.

Next we write the low byte to PC low (PCL). When you write to PCL, the processor will always take the contents of PCLATH and load that into the top 5 bits of PC high, and then write all 13-bits of the program counter at one time.

If you forget to write PCLATH first and you just write to PCL, then whatever happens to be in PCLATH will go into pc high which could be erroneous. In order to read the PC, you read the low byte from PCL. For example, you can use the instruction MOVF PC,W to take the lower 8-bits of the program counter and move them into the W register.

It's important to note at this point that when you read the PCL, the 5 upper bits of PCH are NOT loaded into PCLATH. The PCLATH register can only be loaded from the internal bus by an instruction that defines PCLATH as a destination register.

Slide 27: Look-up Table Example



Let's look at an example of using relative addressing. This example shows a lookup table being used to convert BCD (binary coded decimal) numbers to values that are used to light the segments on a 7 segment display.

The BCD value that we are going to convert to is stored in a register called DisplayValue. Once we have converted this value into the 7 segment equivalent, we write it out to I/O PORTB which will turn on the correct segments.

In this example the PIC16CXXX is running at 4 Mhz, and bits 0 through 6 of I/O PORTB drive the 7-bits of the 7 segment decode. The assembler directive ORG 10 hex tells the assembler to start this program at location 0x10. The interrupt vector address is location 0x04 in program memory and we need to make sure we start well away from the interrupt vector address and also from the reset vector which is at location zero.

When you are using relative addressing to do the table look up, you need to make sure that there is enough room within the page to add the offset to the program counter. Be sure that the result of the Add instruction will not cause the eight bits of PC-Low to rollover, which would require the software engineer to modify PC-latch-high prior to the Add instruction.

The first instruction executed here is to clear the PCLATH register. Because we know we are at absolute location 0x10, then PCLATH needs to be low.

The next instruction moves the contents of DisplayValue, which is our BCD value that needs to be converted, into the W register. The next instruction executes the CALL to a subroutine called SevenSegmentDecode.

Remember that a CALLl instruction takes 2 cycles to execute and when completed, we will end up at the SevenSegmentDecode label. The first instruction in the subroutine is ADDWF PCL, f. This takes the contents of W and adds it to PCL, and stores the result back into PCL.

If we make the assumption the display value register has 0x05 in it then when we do the add instruction, we will add the contents of W, which is 0x05 to PCL and then jump to the 5th element in the table. In actual fact it jumps to the 6th element. The reason for this is when you perform the ADDWF PCL instruction the program counter has already been incremented in order to pre-fetch the next instruction.

As you can see, the program counter is pointing to the line with the comment 'decode 0'. When we add 0x05 to that, we end up at the line with the comment 'decode 5'. That instruction is the return from a sub-routine with a literal value in the W register (RETLW) defined in the next 8 bits, which in this case is 6Dh (01101101b) which is the 7 segment equivalent to make a 5 on the display.

Now we return with the 7 segment value in the W register to the location after the subroutine call which is move W to file PORTB (MOVWF PORTB). This takes the contents of W, which we know is the equivalent to a 0x05 decoded into 7 segments and outputs it to PORTB lighting up the appropriate segments of the LED. The RETLW instruction, the CALL instruction and the ADDWF instruction each have taken 2cycles because they all modified the program counter.

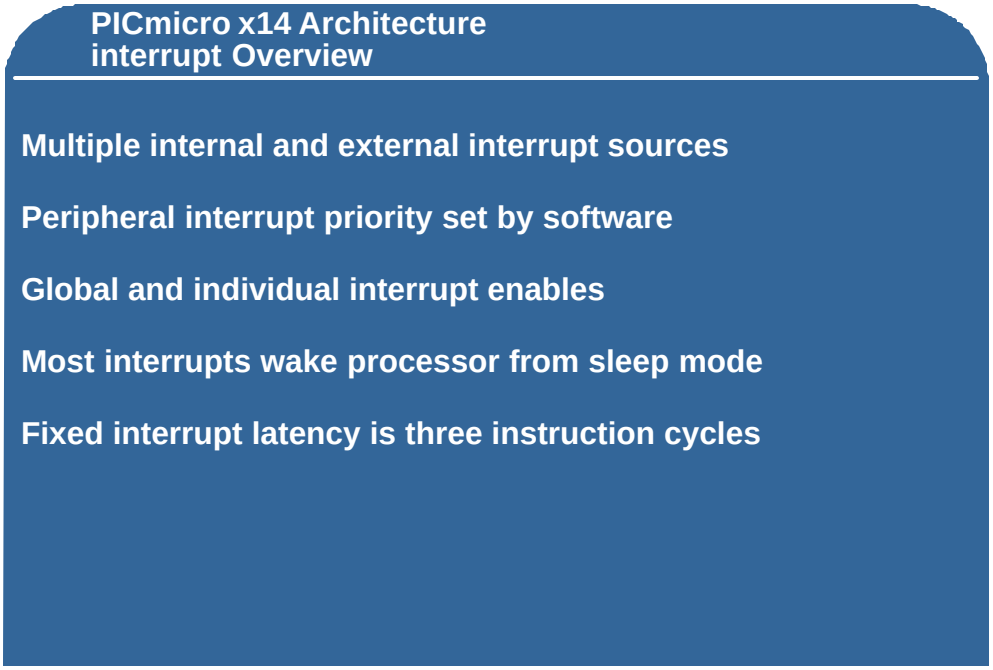
Now we continue with the rest of the program because we have successfully lit up the correct segments in the 7 segment display.

Slide 28: Knowledge Check 6

Question: Relative addressing is useful for:

1) **Looking up information held in tables**
2) Finding your long lost relative
3) Going to the next part in the program sequence
4) Accessing subroutines

Slide 29: Interrupt Overview

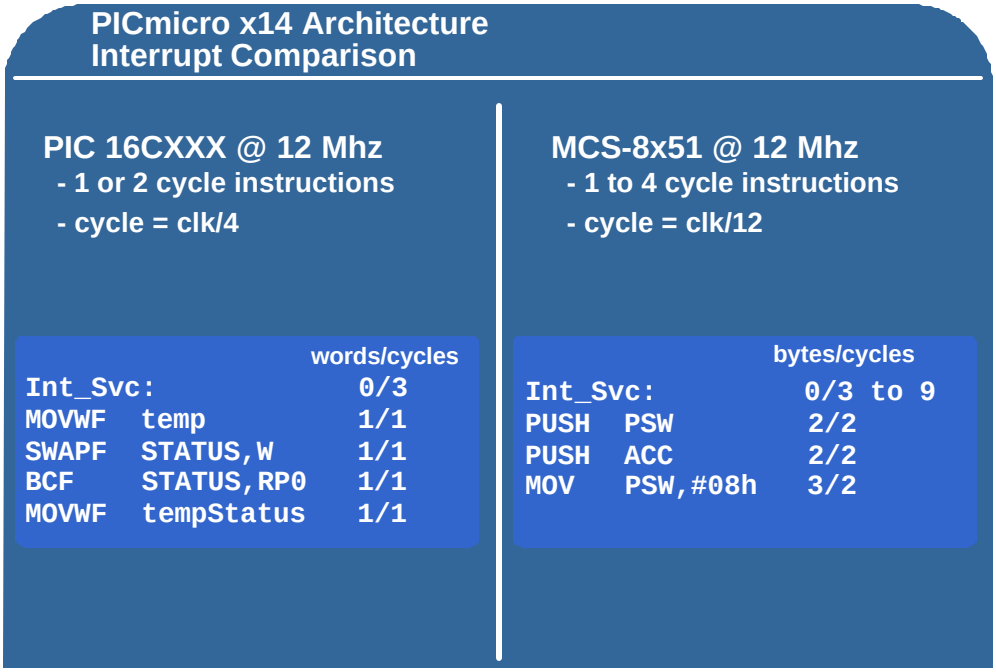


This is an overview of interrupts on the x14 architecture. There are multiple interrupts with both internal and external sources. Peripheral interrupt priorities are set by software using the sequence in which the interrupt request bits are tested. The interrupt you wish to have the highest priority should be tested first.

There are both individual and global interrupt disable bits so you can individually enable or disable individual interrupts or you can enable or disable all interrupts in one instruction using the global interrupt control bit.

Most interrupts will wake the processor from sleep and one of the keys to this architecture is that all interrupts have a latency of 3 instructions. This is very important when you are working with critical timing because you always know how long an interrupt will take.

Slide 30: Interrupt Comparison



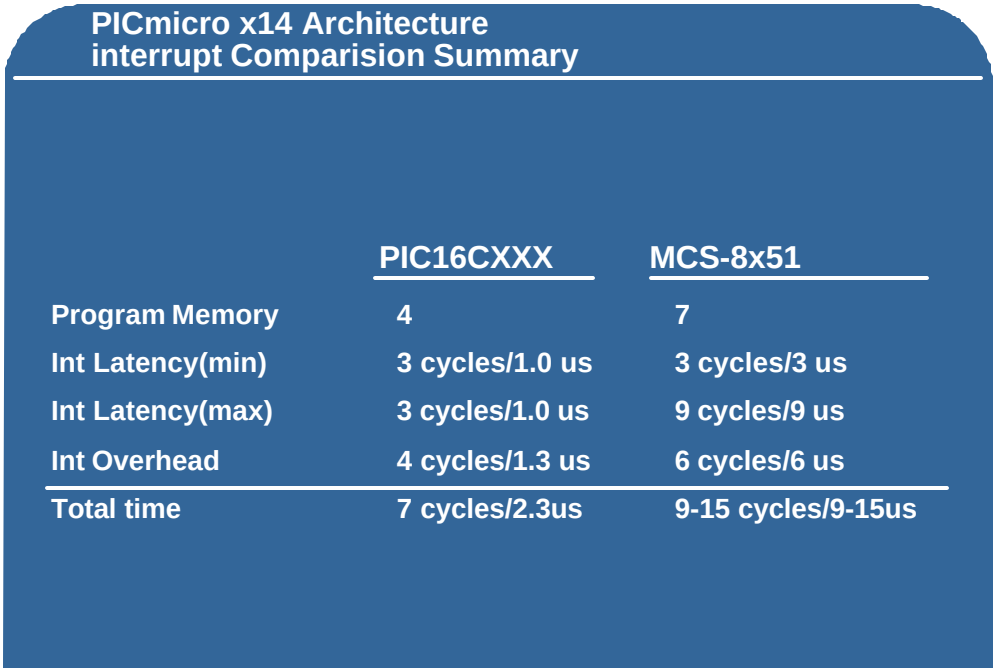
If we look at this comparison and consider the PIC16CXX running at 12 Mhz. It does not matter if the interrupt occurs during a 1 cycle or 2 cycle instruction. The instruction cycle is defined as clock divided by 4, which is the oscillator frequency divided by 4.

To get the program counter to the interrupt service routine takes no words of program memory and 3 cycles. Here we are saving the contents of the W register, so we MOVWF temp and since we are saving the contents of status, we swap status to W, so that it doesn't modify the contents of the status register when we restore it. We clear status RP0 to make sure we are in the correct bank of data registers. We use the BCF instruction to do that and then we move W to the file called temp-status.

When you come to look at the instruction set module you will learn there that it is important to look at what status bits are affected by which instructions so that you can limit or minimize the execution time of your interrupt service routine. So to save W and the status has taken 4 instructions or 4 lines of program memory and 4 instruction cycles to execute.

Performing the same function on the 8x51 product family takes anywhere from 3 to 9 cycles to get to the interrupt service routine. The Push Processor-Status-Word instruction is used to save the status word, which uses 2 bytes of program memory and takes 2 cycles to execute. Next is the push accumulator instruction which saves the accumulator, which uses 2 words and takes 2 cycles. Lastly we move 0x08 into the processor status word to set up for the interrupt service routine. This takes 3 bytes of program memory and 2 cycles to execute.

Slide 31: Interrupt Comparison Summary



In summary this shows that the program memory of the PIC16CXX interrupt service routine, to do the same function, is 4 lines of code against 7 on the MCS8x51 architecture.

Interrupt latency is a constant of 3 cycles on the PICmicro architecture or 1 microsecond where on the 8x51 architecture the interrupt latency can be anywhere from 3 cycles or 3 microseconds to 9 cycles or 9 microseconds.

The overhead of instructions to save the status and the temporary accumulator value is 4 cycles or 1.3 microseconds on the PICmicro and 6 cycles or 6 microseconds on the 8x51 architecture.

Looking that total for both architectures, we see that we 7 cycles and 2.3 microseconds for the PICmicro architecture and anywhere from 9 to 15 cycles or 9 to 15 microseconds for the 8x51 architecture.

You can see that if you have a time critical application, it's much more efficient and consistent to use the PICmicro architecture then the 8x51, which is slower and has a lot of variation.

Slide 32: Knowledge Check 7

Question: Interrupts on the PICmicro x14 architecture:

1) Are prioritized via the different vector addresses
2) Use the accumulator to store return address values
3) **Have a fixed constant response time**
4) Do not operate if the device is in Sleep mode

Slide 33: Closing Slide

This concludes the PICmicro x14 Architecture course. 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.