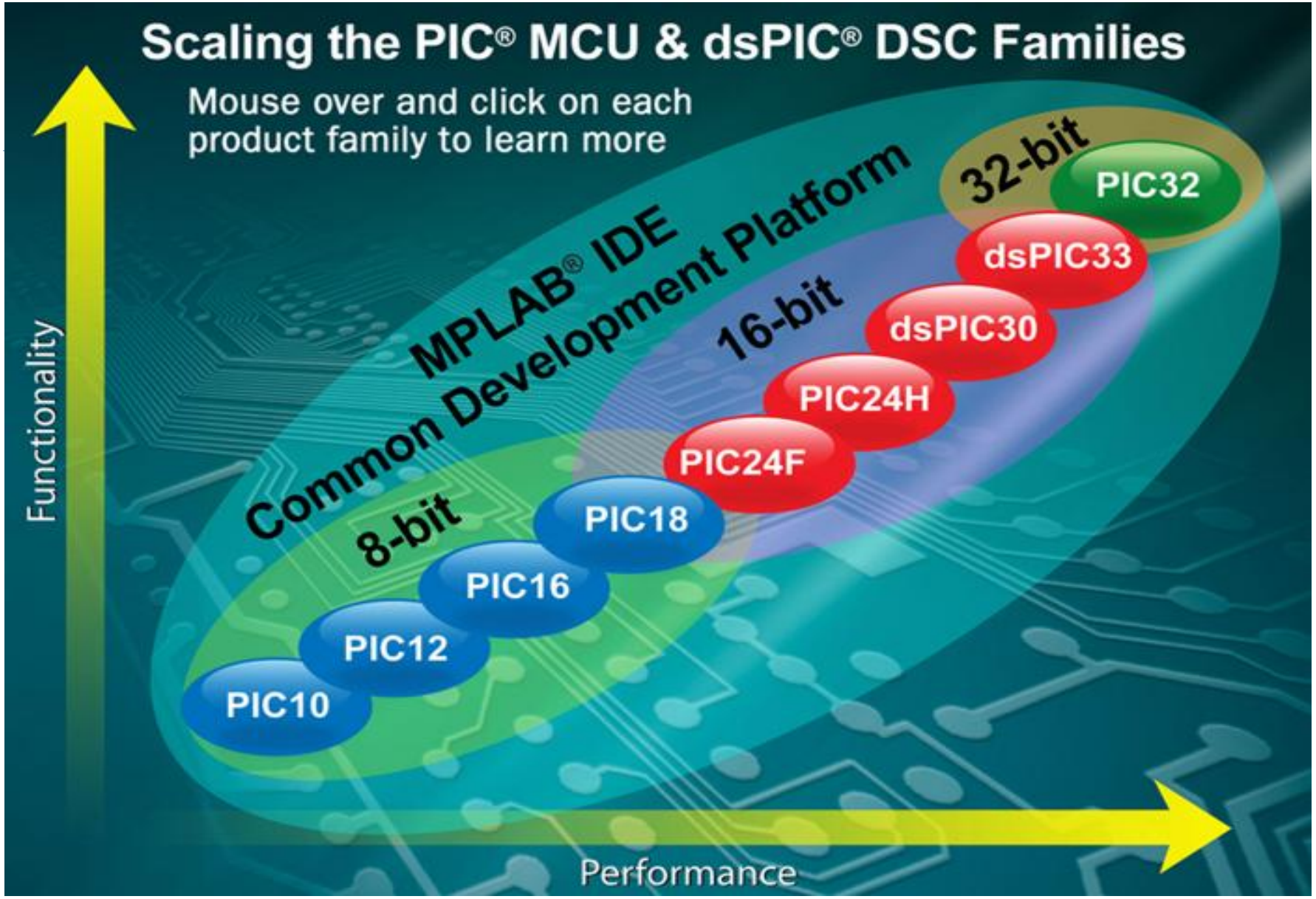
# CENG-336 Introduction to Embedded Systems Development

## PIC 18F Architecture

Spring 2018




http://www.microchip.com

# PIC ARCITECHTURE

- PIC general
  - Features, Advantages, Harvard/Von Neumann Architecture, RISC/CISC
- PIC Families
  - Program memory, Control registers, Speed, Peripherals
- PIC 18F8722
  - Pin Diagram, Core Features, Peripheral Features, Block Diagram
- PIC18 Programmers' Model
  - Registers, Memory (program and data), Addressing modes, Special Function registers

# PIC general-features

- Range of low end 8 bit microcontrollers to high end 32 bit microcontrollers.
- Smallest: have only 6 pins, largest 144 pins.
- Cheap, you can pick them up at less than $1-$10 each.
- Targeted at a various applications: consumer products, automotive, home appliance, connectivity,  etc.

# PIC general-Typical Applications

- Baseline
  - Replace discrete logic functions
    - Gates, simple state machines, encoders/decoders, etc.
  - Disposable electronics
    - Drug / pregnancy testers, dialysis monitor, etc
- Mid-Range
  - Digital sensors, displays, controllers, telecom equipment
  - Glucose / blood pressure set
- PIC18
  - Integration with peripherals + networks
    - USB, Ethernet, MCU-to-MCU, etc
  - Higher level analog peripherals, industrial control, major appliances
- PIC24 / dsPIC30 16-bit ALU with integrated DSP
  - Portable EGK PIC32
- General purpose RISC microprocessor + controller
  - MRI

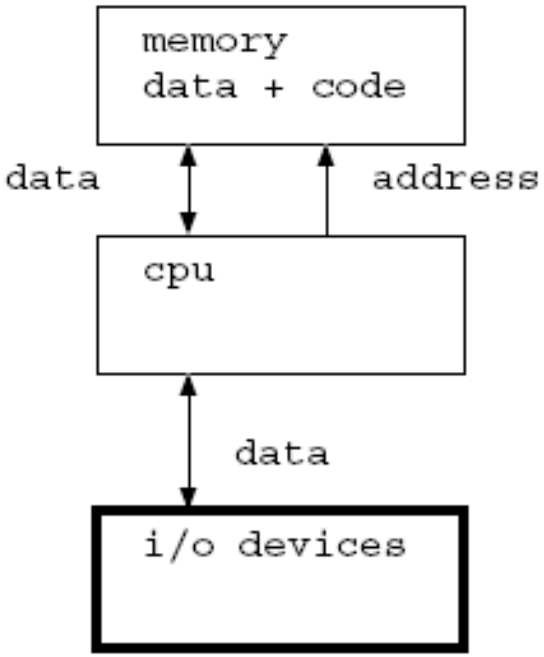
# PIC general-Advantages

- Harvard architecture
- It is a RISC (Reduced Instruction Set Computer) design
- Small instruction set to learn (35-75 instructions to remember)
- It is low cost and has high clock speed

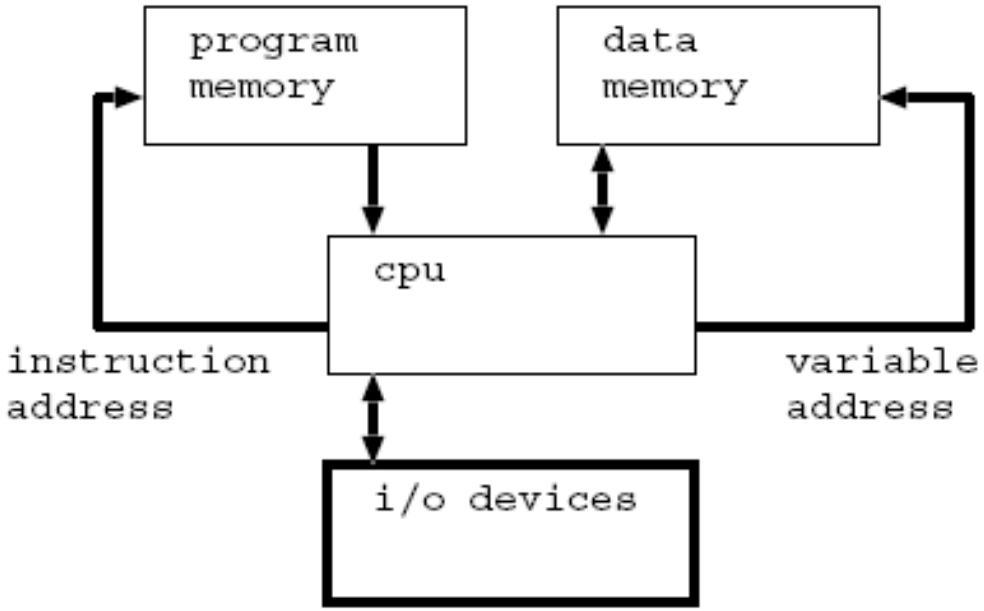
# PIC general-Harvard Architecture

- Like many microcontrollers the PIC is a Harvard (not a von-Neumann) machine
- Simpler and faster
- Separate program bus and data bus: can be different widths!
- For example, PICs use:
  - Data memory (RAM): a small number of 8-bit registers
  - Program memory (ROM): 12-bit, 14-bit or 16-bit wide (in EPROM, FLASH, or ROM)

# PIC general-Comparison




Von Neumann machine




Harvard Machine

# PIC general-Harvard Architecture

- Harvard architecture is a newer concept than Von-Neumann's. It rose out of the need to speed up the work of a microcontroller.

- In Harvard architecture, Data Access and Address Access are seperate. Thus a greater flow of data is possible through the central processing unit.

- PIC18F8722 uses 16 bits for instructions (which allows for all instructions to be one word instructions).

# PIC general-Advantages of Harvard Model

- An add operation of the form “a := b + c” must fetch 2 operands from memory and write 1 operand to memory. In addition it is likely to have to fetch 3 instructions from memory.

- With a single memory this will take 6 cycles. With 2 memories, we can fetch the instructions in parallel with the data and do it in 3 cycles.

- We have different word lengths for instructions and data – 8 bit data and 16 bit instructions.

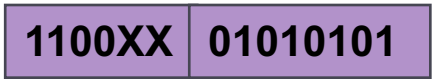
# PIC general-Von Neumann Architecture

- Used in: 80X86 (PCs), 8051, 68HC11, etc.)
- Only one bus between CPU and memory
- data and program memory share the same bus and the same memory, and so must have the same bit width
- Bottleneck: Getting instructions interferes with accessing RAM

# PIC general-RISC Architecture

- PICs and most Harvard chips are "RISC"
  - Reduced Instruction Set Computer (RISC)
  - Used in: SPARC, ALPHA, Atmel AVR, etc.
  - Few instructions (usually < 50)
  - Only a few addressing modes
  - Executes 1 instruction in 1 internal clock cycle (Tcyc)
  - Example:

**PIC16CXXX:** **MOVLW 0x55**

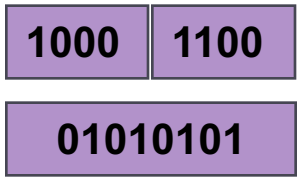



**1 word, 1 cycle**

# PIC general-CISC Architecture

- Traditionally, CPUs are "CISC"
  - Complex Instruction Set Computer (CISC)
  - Used in: 80X86, 8051, 68HC11, etc.
  - Many instructions (usually > 100)
  - Many, many addressing modes
  - Usually takes more than 1 internal clock cycle (Tcyc) to execute
  - Example:

**MC68HC05:** **LDAA 0x55**




**2 bytes, 2 cycles**

# PIC ARCITECHTURE

- PIC general
  - Features, Advantages, Harvard/Von Neumann Architecture, RISC/CISC
- PIC Families
  - Program memory, Control registers, Speed, Peripherals
- PIC 18F8722
  - Pin Diagram, Core Features, Peripheral Features, Block Diagram
- PIC18 Programmers' Model
  - Registers, Memory (program and data), Addressing modes, Special Function registers

# PIC Family

- 8 bit PIC MCU
  - PIC10
  - PIC12
  - PIC16
  - PIC18 (some has 16 bit MCU)
- 16 bit PIC MCU
  - PIC24E
  - PIC24F
  - PIC24H
- 16 bit dsPIC DSC
  - dsPIC30F
  - dsPIC33F
  - dsPIC33E
- 32 bit PIC MCU
  - PIC32

# PIC Family

| Architecture | Family | | Data Width | Instruction Width |
|---|---|---|---|---|
| 8-bit MCU | PIC10 / PIC12 / PIC16 | Baseline | 8 bits | 12-bits |
| | | Mid-Range | | 14-bits |
| | PIC18 | | | 16-bits |
| 16-bit MCU | PIC24 | | 16 bits | 16 bits |
| | dsPIC30 | Integrated DSP | | |
| 32-bit MCU | | | 32 bits | 32 bits |

**Data width**

8 / 16/ 32 bits

Wider integer ⇒ higher precision arithmetic

**Instruction width**

12 / 14 / 16 / 32 bits

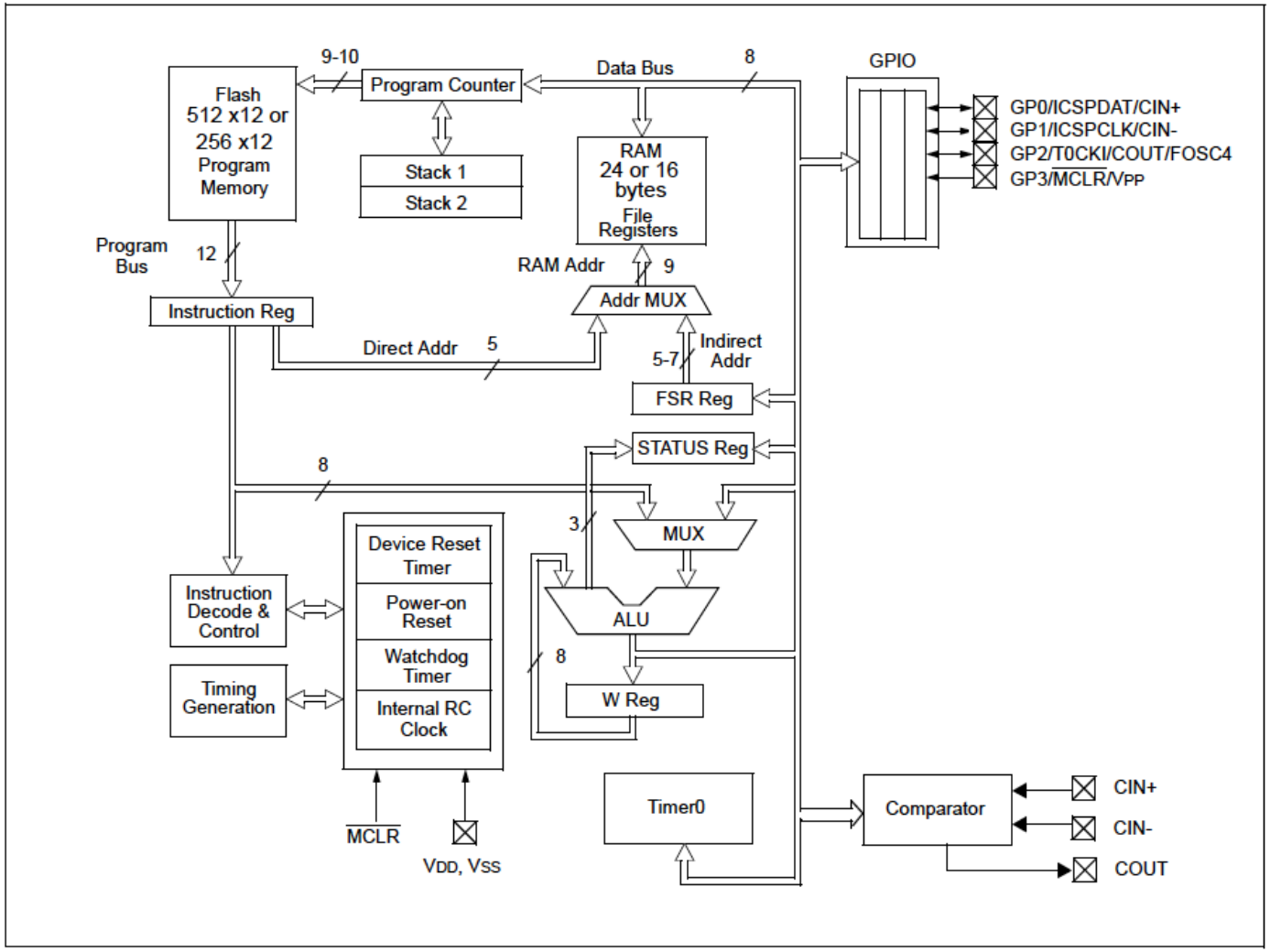
Wider instruction ⇒ more complex instructions + higher precision arithmetic

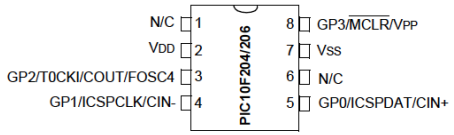
# PIC Family-Samples

| | PIC10F2xx | PIC12F5xx | PIC16F5xx | PIC10F3xx<br>PIC12F6xx<br>PIC16F6xx | PIC18F5xx |
|---|---|---|---|---|---|
| Instruction word | 12 bits | 12 bits | 12 bits | 14 bits | 16 bits |
| Instructions | 33 | 33 | 33 | 35 | 83 |
| Program memory | 256 – 512 words | 512 – 1024 words | 1024 – 2048 words | 256 – 8192 words | 2 Kwords – 64 Kwords |
| ROM | Flash | Flash | Flash | Flash | Flash |
| Data memory (bytes) | 16 – 24 | 25 – 41 | 25 – 134 | 56 – 368 | 256 – 4K |
| Interrupts | 0 | 0 | 0 | int / ext | int / ext |
| Pins | 6 | 8 | 14 – 40 | 6 – 64 | 18 – 100 |
| I/O pins | 4 | 6 | 12 – 32 | 4 – 54 | 16 – 70 |
| Stack | 2 levels | 2 levels | 2 levels | 8 levels | 31 levels |
| Timers | 1 | 1 | 1 | 2 – 3 | 2 – 5 |
| Bulk price | $0.35 | $0.50 | $0.50 – $0.85 | $0.35 – $2.50 | $1.20 - $8.50 |

# PIC Family-Example-PIC 10F204 Block Diagram

**FIGURE 3-2: PIC10F204/206 BLOCK DIAGRAM**



# PIC Family-Program Memory

PICs have two different types of program storage:

1. FLASH
   - Re-writable (even by chip itself)
   - Much faster to develop on!
   - Finite number of writes (~100k Writes)

2. EPROM (Erasable Programmable Read Only Memory)
   - Needs high voltage from a programmer to program (~13V)
   - Needs windowed chips and UV light to erase
   - Note: One Time Programmable (OTP) chips are EPROM chips, but with no window!

# PIC Family-Control Registers

PICs use a series of "special function registers" for controlling peripherals and PIC behaviors.

Some examples are:

| | |
|---|---|
| STATUS | Bank select bits, ALU bits (zero, borrow, carry) |
| INTCON | Interrupt control: interrupt enables, flags, etc. |
| TRIS | Tristate control for digital I/O: which pins are 'floating' |
| TXREG | UART transmit register: the next byte to transmit |

# PIC Family-Speed

- PICs require a clock to work.
- Can use crystals, clock oscillators, or even an RC circuit.
- Some PICs have a built in 4MHz RC clock
  - Not very accurate, but requires no external components!
- Instruction speed = 1/4 clock speed (Tcyc = 4 * Tclk)
- All PICs can be run from DC to their maximum spec'd speed:
  - 12F and 16F -> 32MHz
  - 18F -> 64MHz and 80MHz
  - 32F -> 40MHz, 50MHz, 80MHz, 100MHz, 200MHz

# PIC Family-Peripherals

Different PICs have different on-board peripherals

Some common peripherals are:

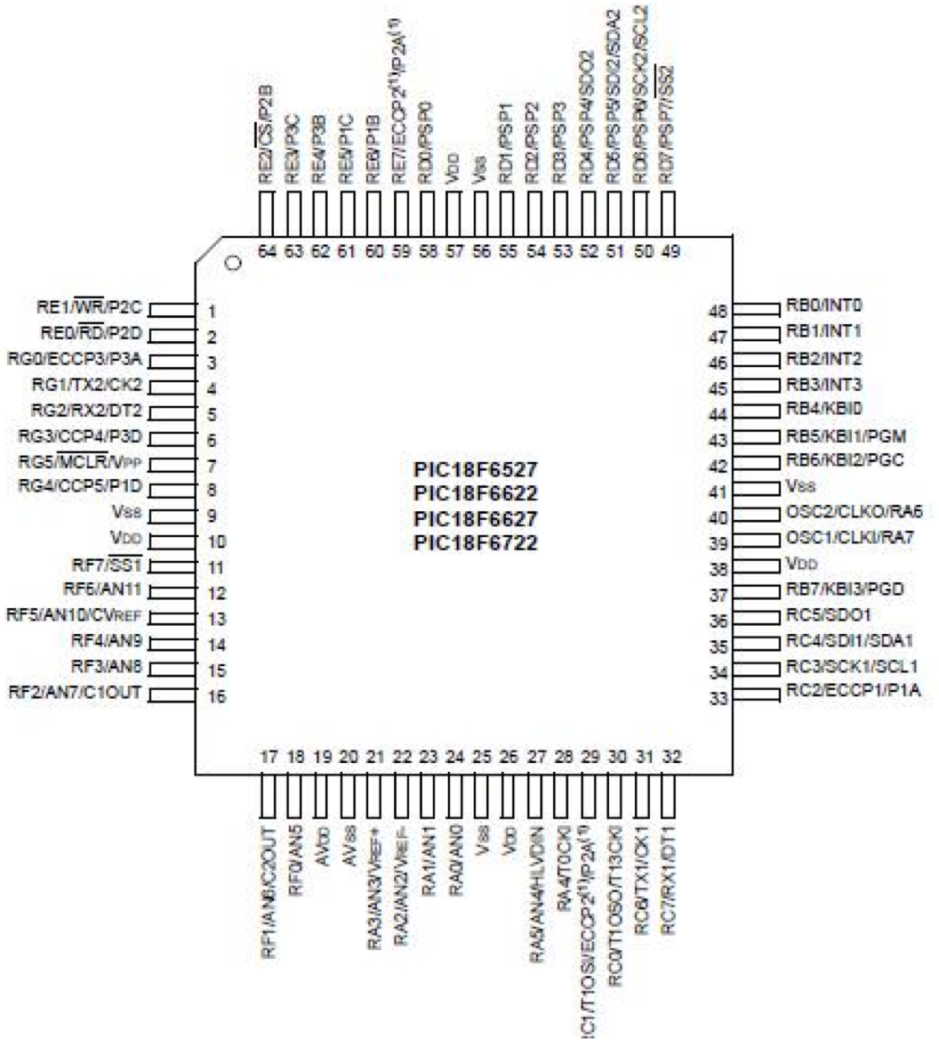
- Tri-state ("floatable") digital I/O pins
- Analog to Digital Converters (ADC) (8, 10 and 12bit, 50ksps)
- Serial communications: UART (RS-232C), SPI, I2C, CAN
- Pulse Width Modulation (PWM) (10bit)
- Timers and counters (8 and 16bit)
- Watchdog timers, Brown out detect, LCD drivers

# PIC ARCITECHTURE

- PIC general
  - Features, Advantages, Harvard/Von Neumann Architecture, RISC/CISC
- PIC Families
  - Program memory, Control registers, Speed, Peripherals
- PIC 18F8722
  - Pin Diagram, Core Features, Peripheral Features, Block Diagram
- PIC18 Programmers' Model
  - Registers, Memory (program and data), Addressing modes, Special Function registers

# PIC 18F-Pin Diagram

- I/O Ports: R{A,B,C,D,E,F,H,J}0-R{A,B,C,D,E,F,H,J}7 and RG0-RG5
- A/D: AN0-AN15
- Ints: INT0-INT3
- Oscillator inputs



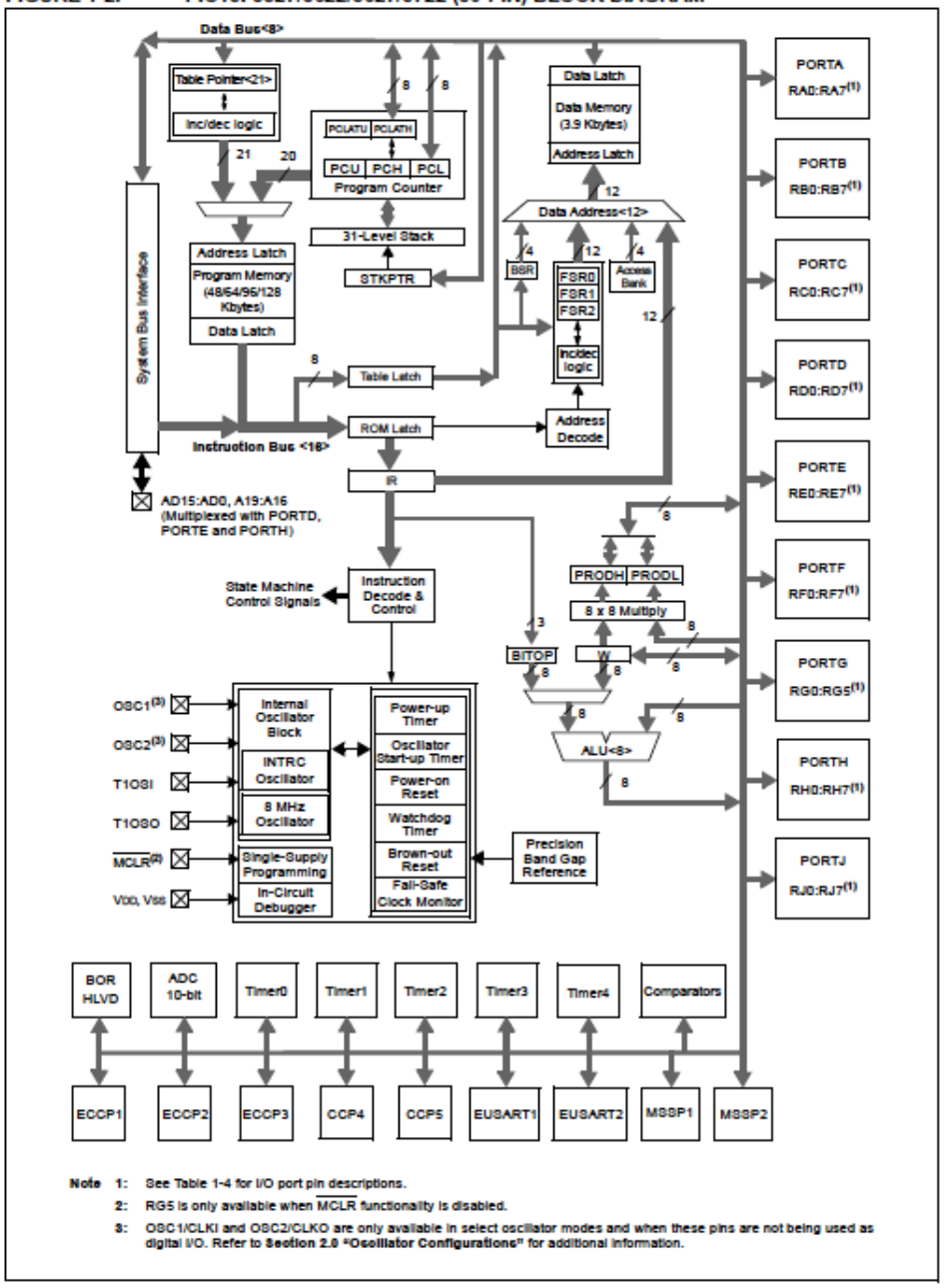
# PIC 18F8722-Core Features

- Accumulator Based Machine
- Harvard Architecture Memory (separate program and data memory)
  - 2Mx16bit Flash Based Instruction Memory
  - 4096x8bit Static Ram Based Data Memory (16 banks, File Registers)
- 75+8 Instructions (fixed length encoding - 16-bit)
- 4 Addressing Modes (inherent, direct, indirect, relative)
- 31x21bit Hardware Stack
- Execution Speed
  - Overlapped Instruction Fetch and Instruction Execution
  - 1 cycle/instruction (non-branching)
  - 2 cycles/instruction (branching)
  - 1 cycle period = 4/CLK_IN (ex. 40Mhz CLK_IN -> 100ns cycle period)

# PIC 18F8722-Peripheral Features

- Up to 9 I/O ports available
- 5 Timer/counters (programmable prescalars)
- 2 Capture/Compare/PWM modules
- 10-bit up to16 channel analog-to-digital converter
- Synchronous serial port
- 2 USART
- 8-bit Parallel Slave Port - function allow another processor to read from a data buffer in the PIC.
- 1024 bytes of EEPROM Memory
- Interrupts
  - Three Programmable External Interrupts
  - Four Input Change Interrupts

FIGURE 1-2: PIC18F8527/8622/8627/8722 (80-PIN) BLOCK DIAGRAM




**Note 1:** See Table 1-4 for I/O port pin descriptions.

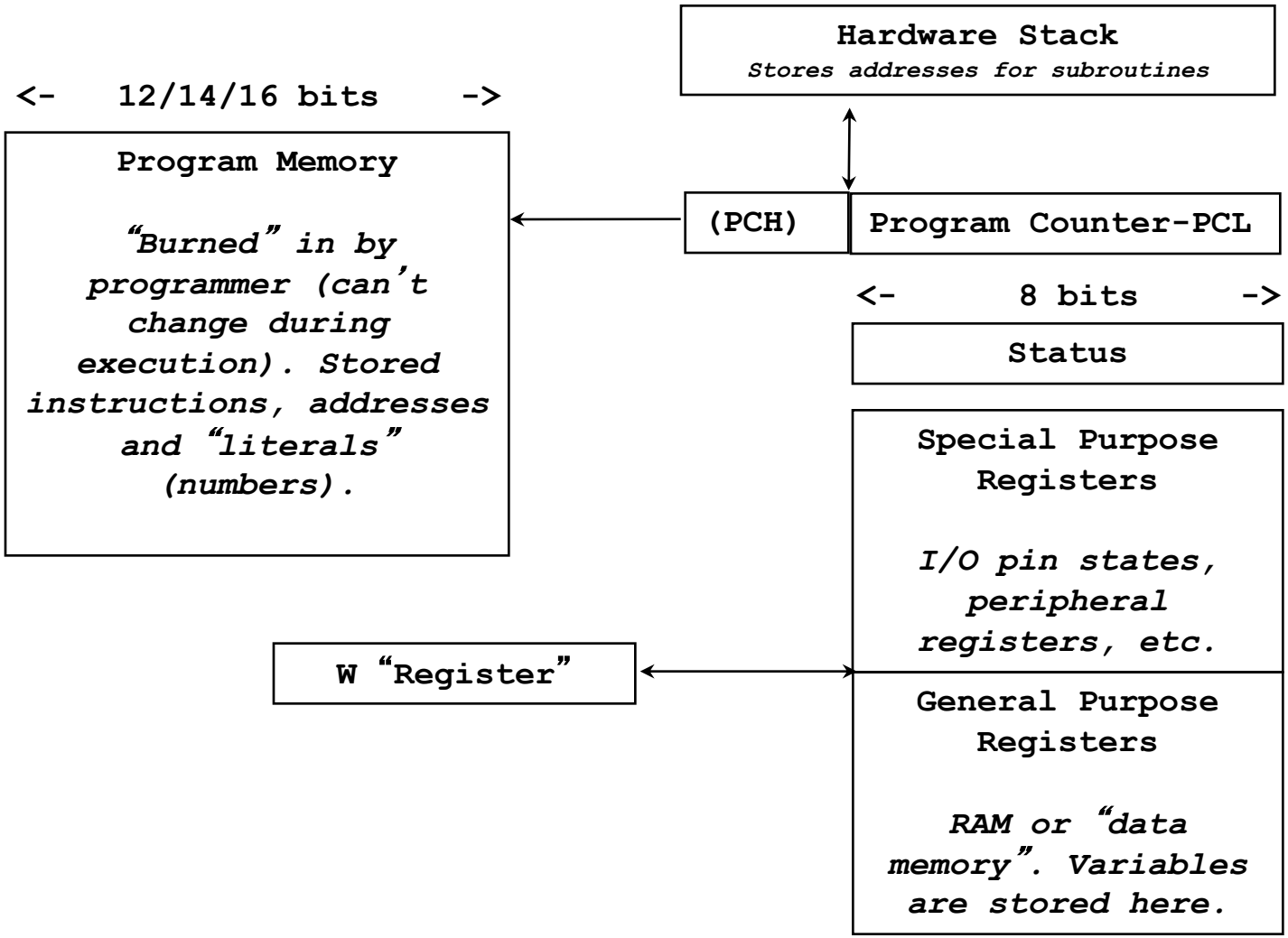
**2:** RG5 is only available when MCLR functionality is disabled.

**3:** OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

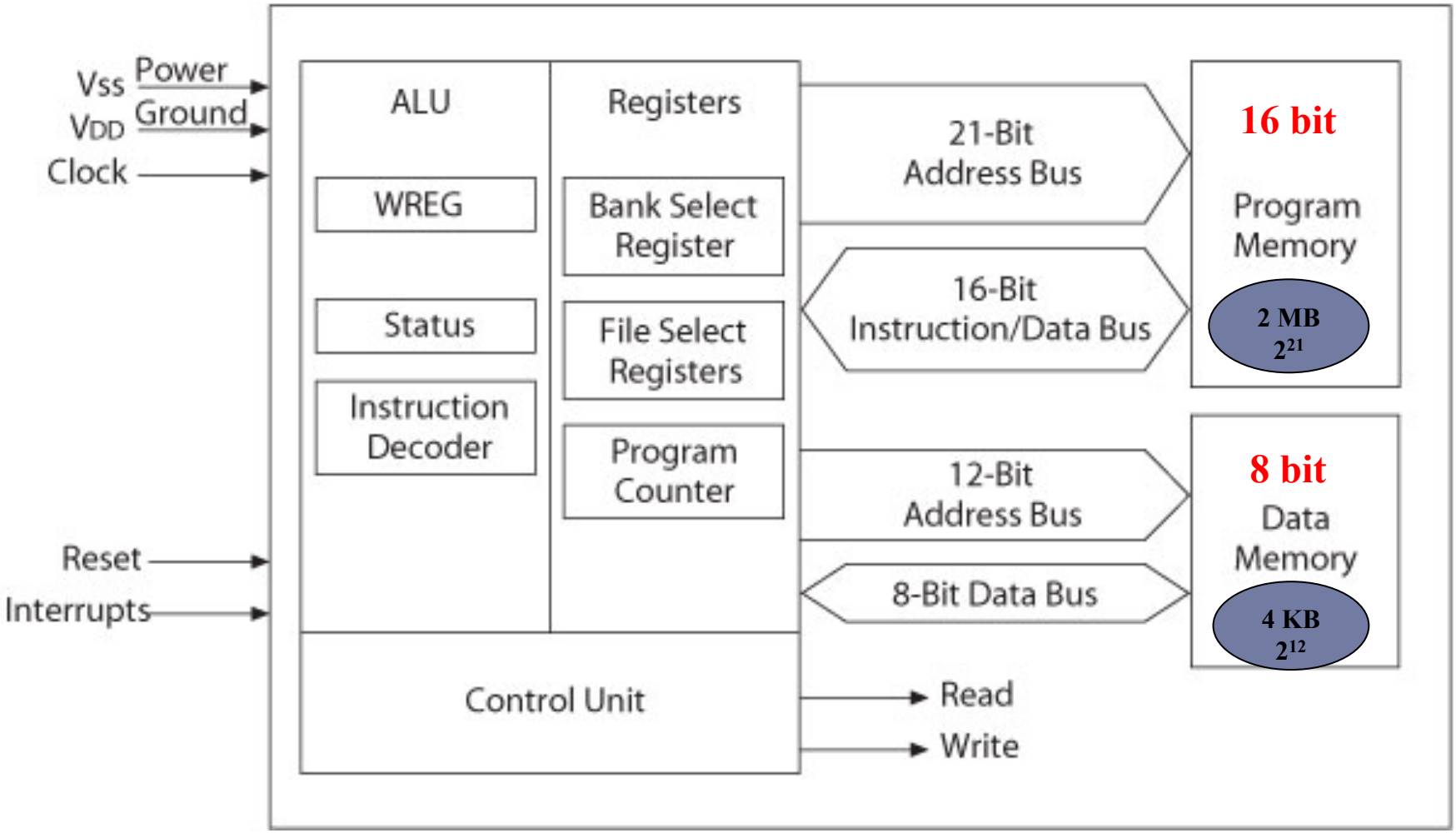
# PIC ARCITECHTURE

- PIC general
  - Features, Advantages, Harvard/Von Neumann Architecture, RISC/CISC
- PIC Families
  - Program memory, Control registers, Speed, Peripherals
- PIC 18F8722
  - Pin Diagram, Core Features, Peripheral Features, Block Diagram
- PIC18 Programmers' Model
  - Registers, Memory (program and data), Addressing modes, Special Function registers

# Programmer's Model

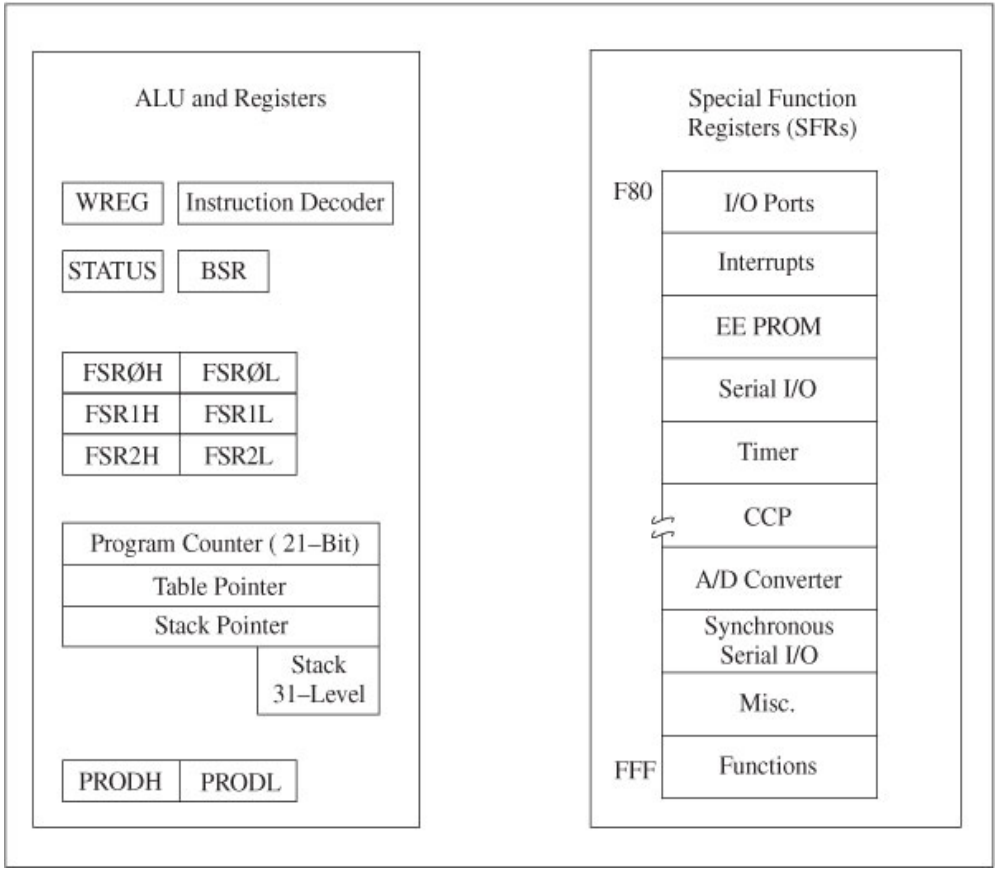


# Programmer's Model-Memory

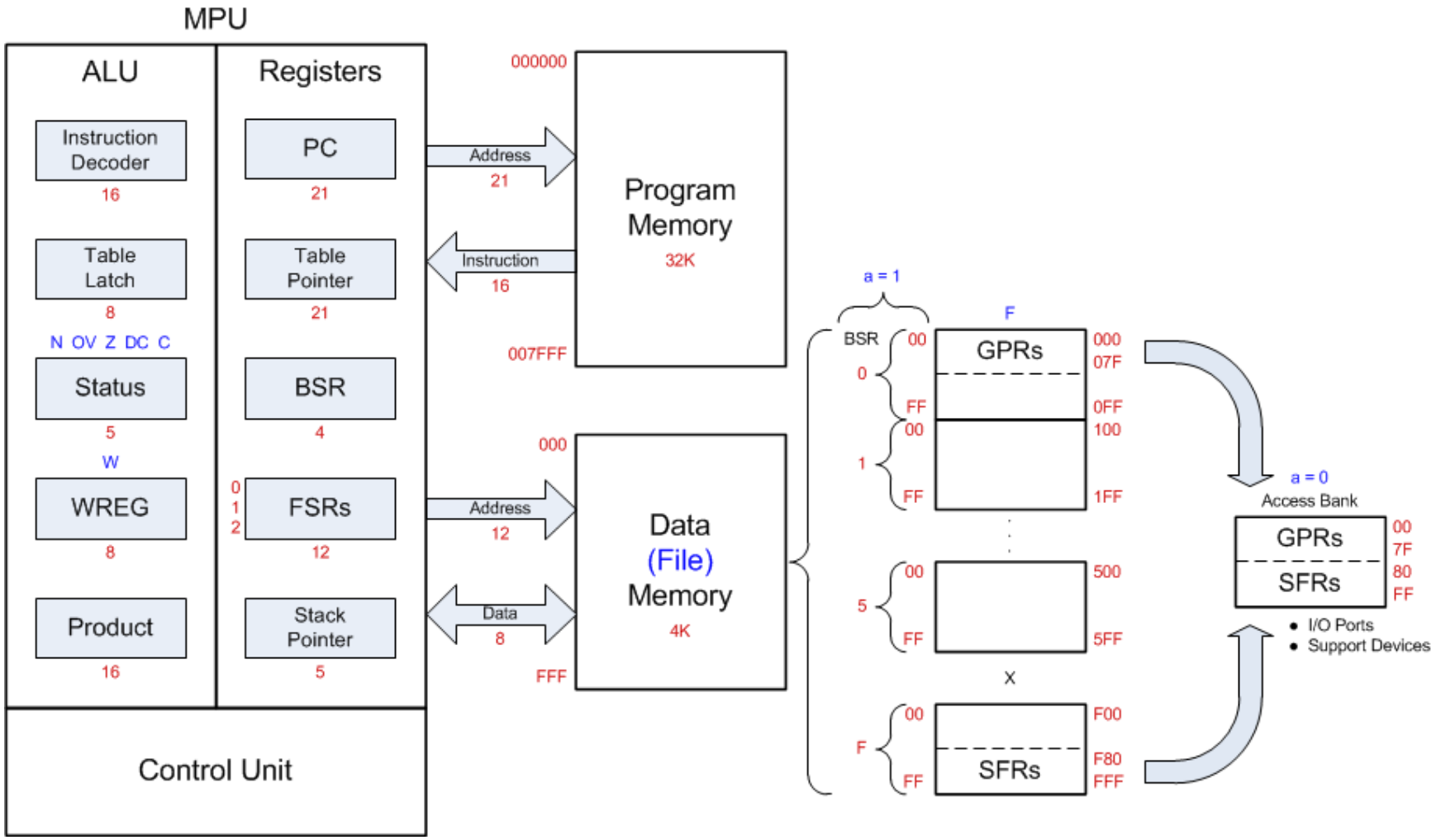


# Programmer's Model

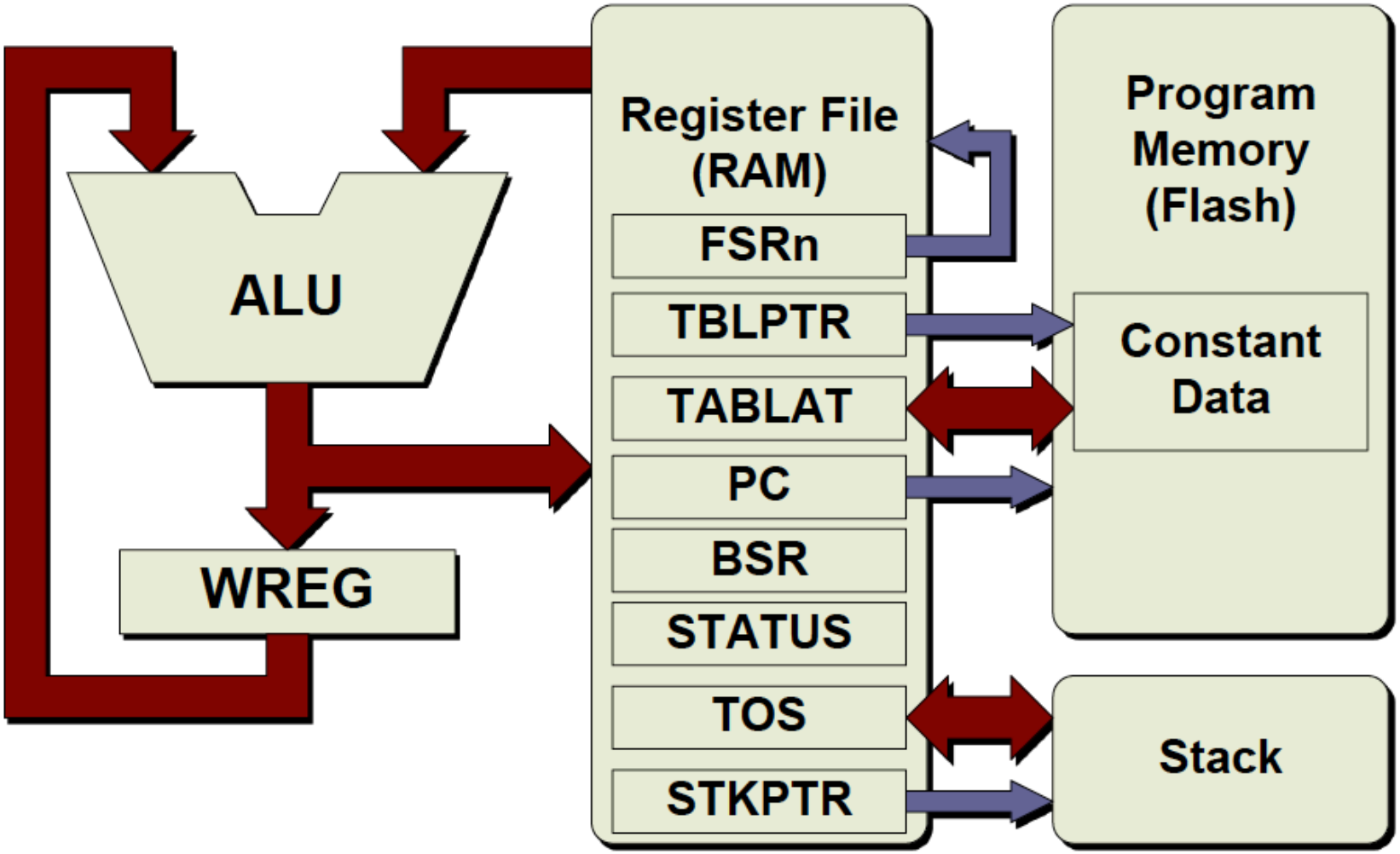
- The representation of the internal architecture of a microprocessor, necessary to write assembly language programs
- Divided into two groups
  - ALU Arithmetic Logic Unit (ALU) and the core
  - Special Function Registers (SFRs) from data memory
- Uses memory-mapped I/O



# Programmer's Model



# Programmer's Model



# Programmer's Model-Registers

- WREG
  - 8-bit Working Register (equivalent to an accumulator)
- BSR: Bank Select Register
  - 4-bit Register (0 to F)
    - Only low-order four bits are used to provide MSB four bits of a12-bit address of data memory.
- STATUS: Flag Register

# Programmer's Model-Registers

## Accumulator (Working Register)

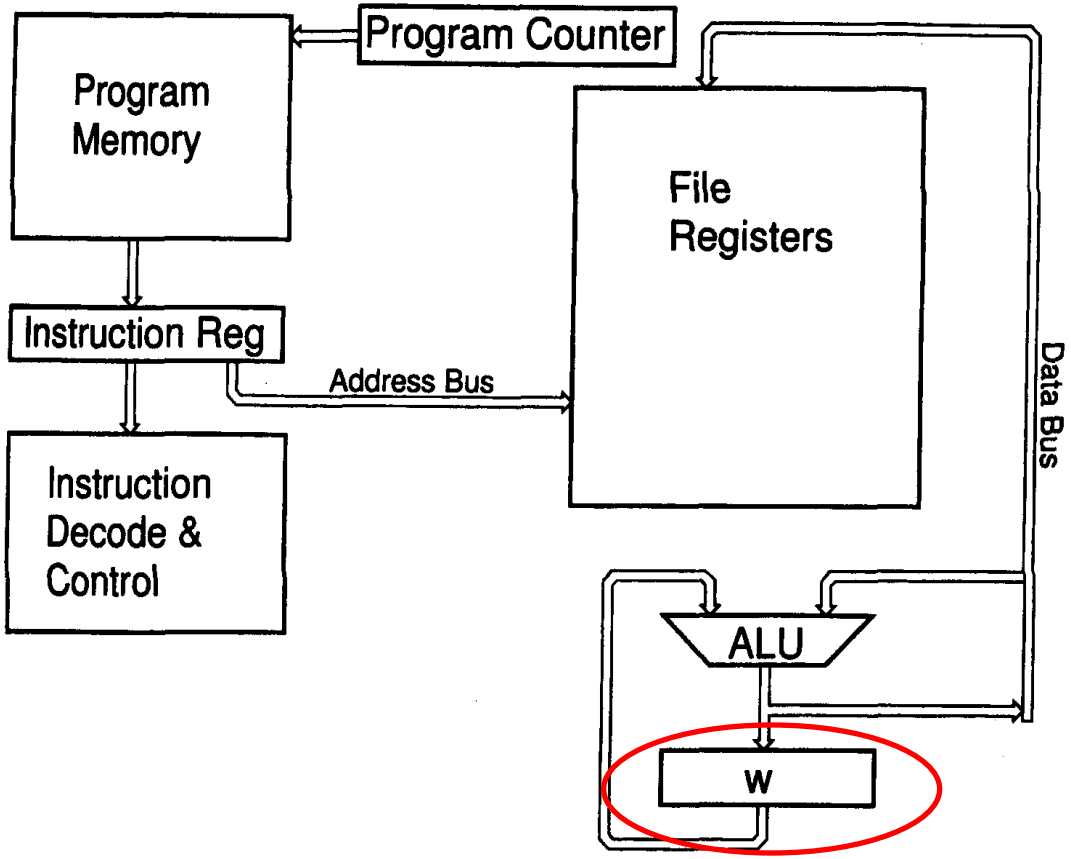
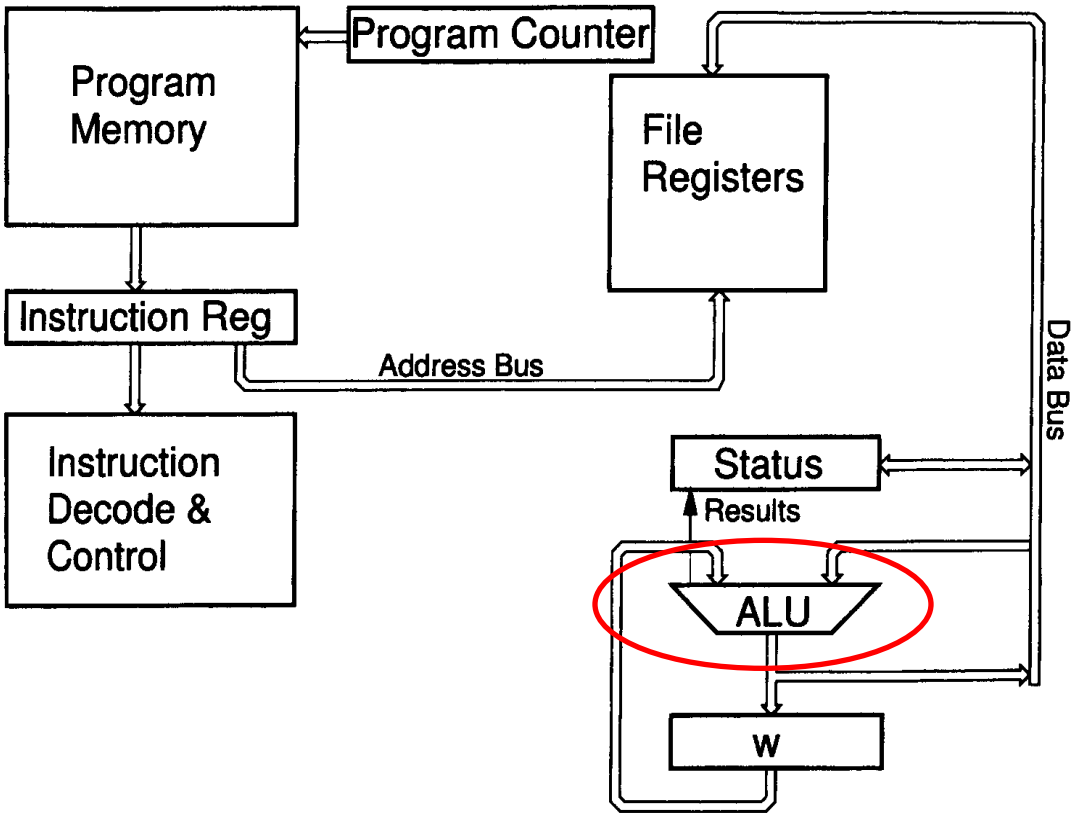



**Figure 3-4** PICmicro® MCU Processor with "w" register as an "accumulator"

- to add two numbers together
  - first move the contents of one file register into the W register
  - then add the contents of the second file register to W
  - the result can be written to W or to the second file register

# Programmer's Model-Registers

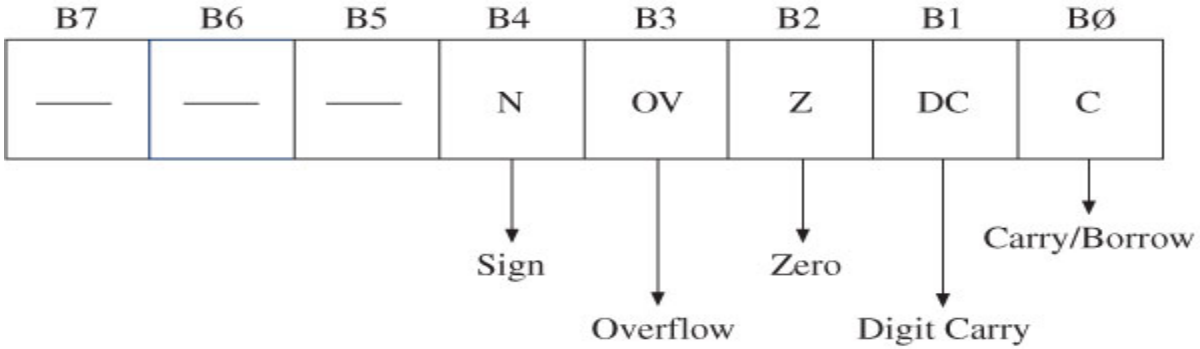
## Status Register




- the STATUS register stores 'results' of the operation
- three of the bits of the STATUS register are set based on the result of an arithmetic or bitwise operation

# Programmer's Model-Registers

## Status Register




Example: 9F+52 =F1
1001 1111
0101 0010
---
1111 0001
N=1,OV=0, Z=0, C=0

- C (Carry/Borrow Flag):
  - set when an addition generates a carry and a subtraction generates a borrow
- DC (Digit Carry Flag):
  - also called Half Carry flag; set when carry generated from Bit3 to Bit4 in an arithmetic operation
- Z (Zero Flag):
  - set when result of an operation is zero
- OV (Overflow Flag):
  - set when result of an operation of signed numbers goes beyond seven bits
- N (Negative Flag):
  - set when bit B7 is one of the result of an arithmetic /logic operation

# Programmer's Model-Registers

File Select Registers (FSR)

- There are three registers:
  - FSR0, FSR1, and FSR2
  - Each register composed of two 8-bit registers (FSRH and FSRL)
- Used as pointers for data registers
- Holds 12-bit address of data register

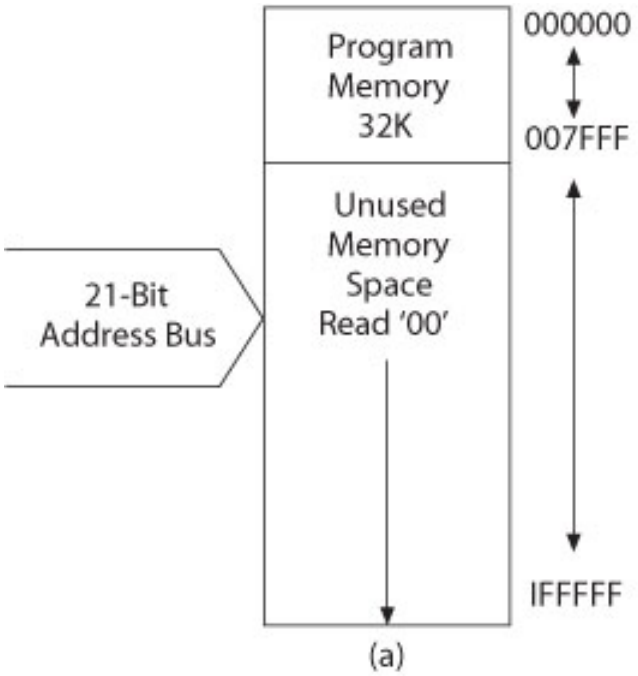
# Programmer's Model-Registers

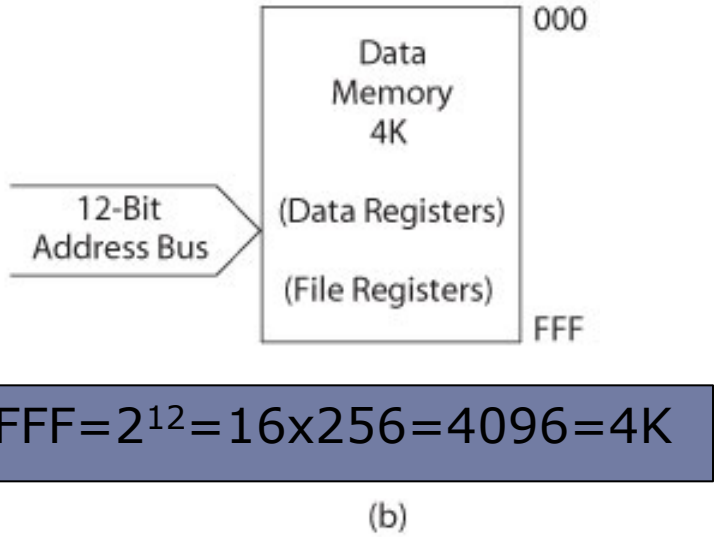
## Other Registers

- Program Counter (PC)
  - 21-bit register functions as a pointer to program memory during program execution
- Table Pointer
  - 21-bit register used as a memory pointer to copy bytes between program memory and data registers
- Stack Pointer (SP)
  - Register used to point to the stack
- Stack
  - 31 word-sized registers used for temporary storage of memory addresses during execution of a program
- Special Function Registers (SFRs):
  - Data registers associated with I/O ports, support devices, and processes of data transfer

# Programmer's Model-Memory

- Program memory with addresses (Flash)
- Data memory with addresses



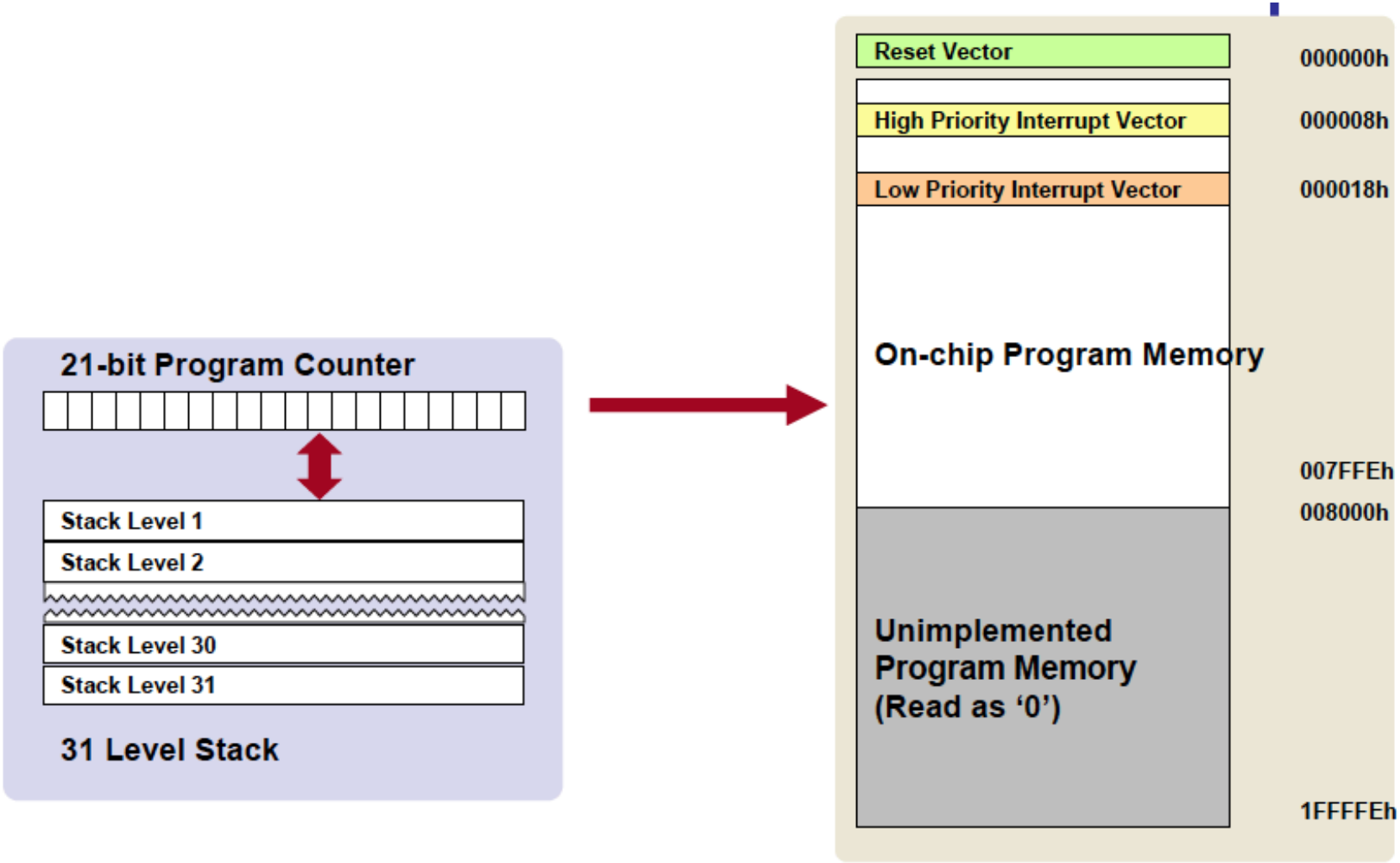




$FFF=2^{12}=16x256=4096=4K$

# Programmer's Model-Program Memory

- Used for storing compiled code
- Each location is 16 bits long (Every instruction is coded as a 16 bit word)
- Addresses H’0000’ (reset), H’0008’ (high priority interrupt) and H'0018 (low priority interrupt) are treated in a special way
- PC can address up to 2M addresses

# Programmer's Model-Program Memory



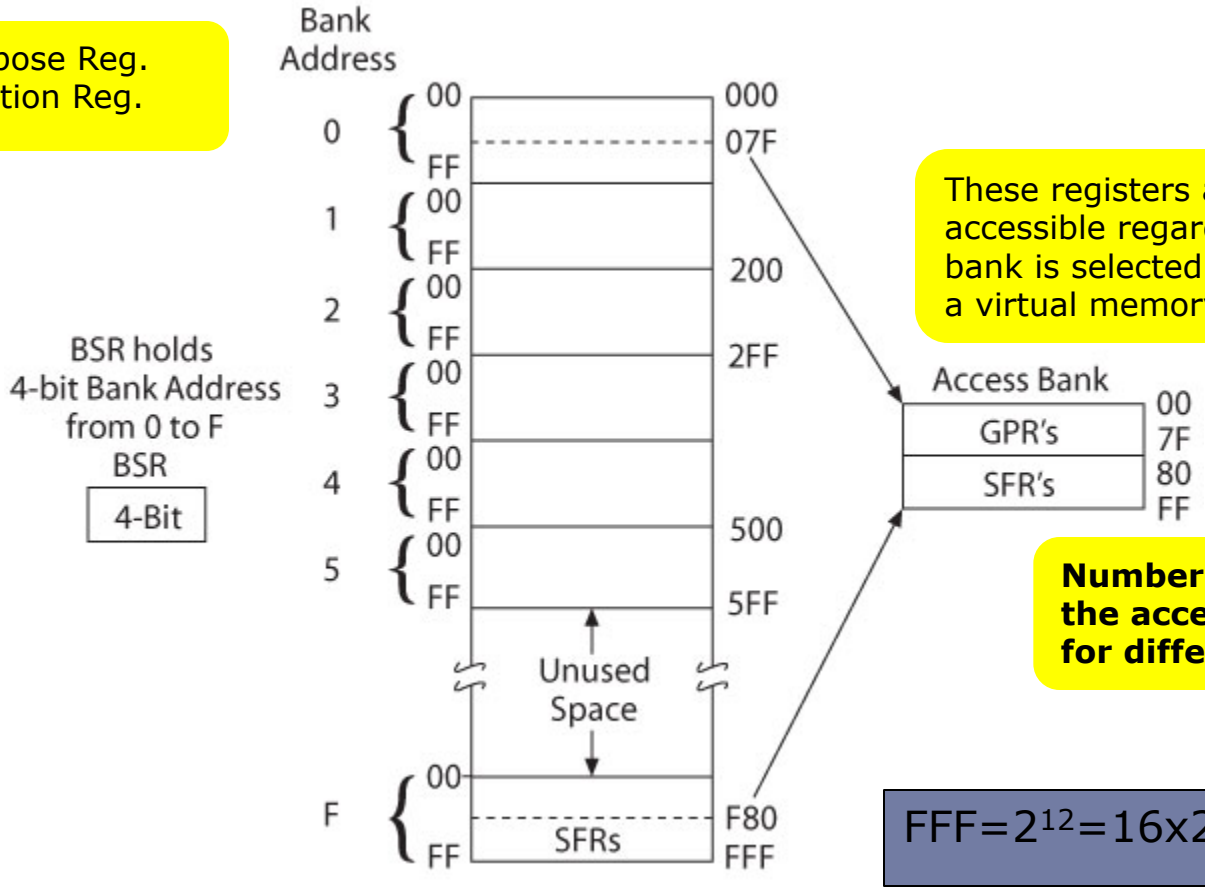
# Programmer's Model-Data Memory

- Consist of 2 Components
  - General Purpose Register (GPR) Files (RAM)
  - Special Purpose Register (SPR) files

- This portion of memory is separated into banks of 256 bytes long

# Programmer's Model-Data Memory with Access Banks

GPR=General Purpose Reg.
SFR=Special Function Reg.



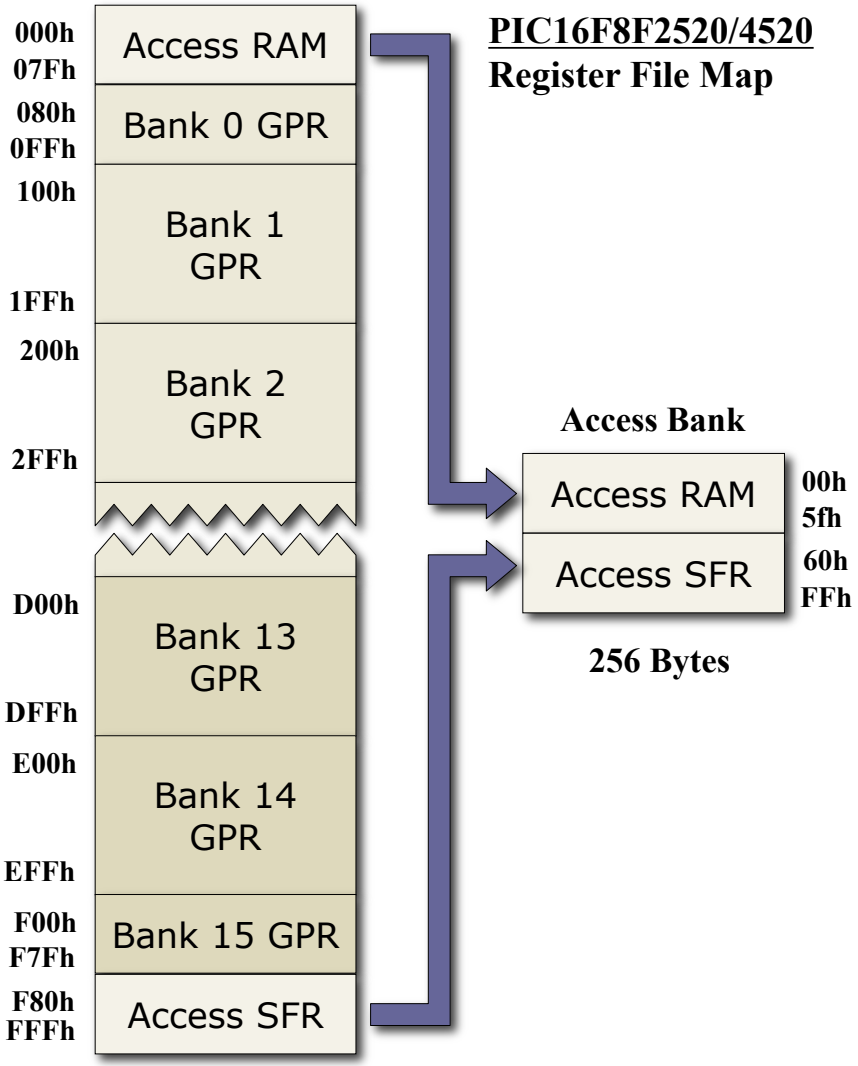

These registers are always accessible regardless which bank is selected – acting as a virtual memory

**Number of GPR's in the access bank differs for different PIC18's**

FFF=$2^{12}$=16x256=4096=4K

# Programmer's Model-Data Memory Organization

- Data Memory up to 4k bytes
- Divided into 256 byte banks
- Half of bank 0 and half of bank 15 form a virtual bank that is accessible no matter which bank is selected



# PIC18F Special Function Registers

| address | Name | Description |
|---|---|---|
| 0xFFF | TOSU | Top of stack (upper) |
| 0xFFE | TOSH | Top of stack (high) |
| 0xFFD | TOSL | Top of stack (low) |
| 0xFFC | STKPTR | Stack pointer |
| 0xFFB | PCLATU | Upper program counter latch |
| 0xFFA | PCLATH | High program counter latch |
| 0xFF9 | PCL | Program counter low byte |
| 0xFF8 | TBLPTRU | Table pointer upper byte |
| 0xFF7 | TBLPTRH | Table pointer high byte |
| 0xFF6 | TBLPTRL | Table pointer low byte |
| 0xFF5 | TABLAT | Table latch |
| 0xFF4 | PRODH | High product register |
| 0xFF3 | PRODL | Low product register |
| 0xFF2 | INTCON | Interrupt control register |
| 0xFF1 | INTCON2 | Interrupt control register 2 |
| 0xFF0 | INTCON3 | Interrupt control register 3 |
| 0xFEF | INDF0 (1) | Indirect file register pointer 0 |
| 0xFEE | POSTINC0 (1) | Post increment pointer 0 (to GPRs) |
| 0xFED | POSTDEC0 (1) | Post decrement pointer 0 (to GPRs) |
| 0xFEC | PREINC0 (1) | Pre increment pointer 0 (to GPRs) |
| 0xFEB | PLUSW0 (1) | Add WREG to FSR0 |

| address | Name | Description |
|---|---|---|
| 0xFEA | FSR0H | File select register 0 high byte |
| 0xFE9 | FSR0L | File select register 0 low byte |
| 0xFE8 | WREG | Working register |
| 0xFE7 | INDF1 (1) | Indirect file register pointer 1 |
| 0xFE6 | POSTINC1 (1) | Post increment pointer 1 (to GPRs) |
| 0xFE5 | POSTDEC1 (1) | Post decrement pointer 1 (to GPRs) |
| 0xFE4 | PREINC1 (1) | Pre increment pointer 1 (to GPRs) |
| 0xFE3 | PLUSW1 (1) | Add WREG to FSR1 |
| 0xFE2 | FSR1H | File select register 1 high byte |
| 0xFE1 | FSR1L | File select register 1 low byte |
| 0xFE0 | BSR | Bank select register |
| 0xFDF | INDF2 (1) | Indirect file register pointer 2 |
| 0xFDE | POSTINC2 (1) | Post increment pointer 2 (to GPRs) |
| 0xFDD | POSRDEC2 (1) | Post decrement pointer 2 (to GPRs) |
| 0xFDC | PREINC2 (1) | Pre increment pointer 2 (to GPRs) |
| 0xFDB | PLUSW2 (1) | Add WREG to FSR2 |
| 0xFDA | FSR2H | File select register 2 high byte |
| 0xFD9 | FSR2L | File select register 2 low byte |
| 0xFD8 | STATUS | Status register |

Note 1. This is not a physical register

# Programmer's Model-Register Addressing Modes

- There are 4 types of addressing modes in PIC
  - Inherent
  - Literal (Immediate) Addressing
  - Direct Addressing
  - Indirect Addressing
  - An additional addressing mode, Indexed Literal Offset, is available when the extended instruction set is enabled (XINST Configuration bit = 1)

# Programmer's Model-Inherent Addressing

- Many PIC18 control instructions do not need any argument at all
- they either perform an operation that globally affects the device or they operate implicitly on one register.
- This addressing mode is known as Inherent Addressing
- Examples include
  - SLEEP
  - RESET
  - DAW

# Programmer's Model-Literal (immediate) Addressing

- Some instructions require an additional explicit argument in the opcode.
- This is known as Literal Addressing mode because they require some literal value as an argument.
- Examples include

  ADDLW and MOVLW,

  which respectively, add or move a literal value to the W register.
- Other examples include CALL and GOTO, which include a 20-bit program memory address.
- movlw H'0F'

# Programmer's Model-Direct Addressing

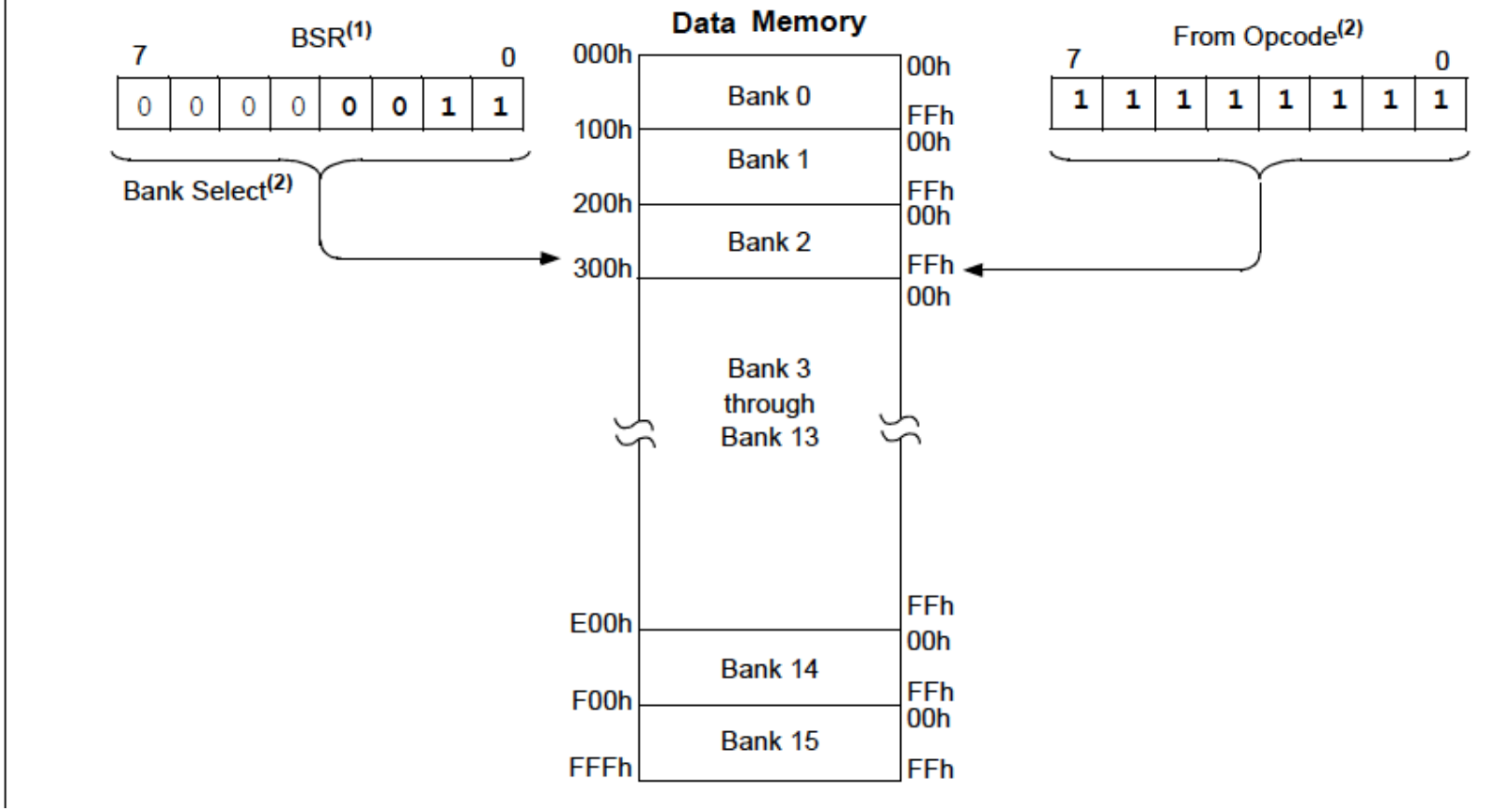
- Direct Addressing specifies all or part of the source and/or destination address of the operation within the opcode itself.
- Uses 8 bits of 16 bit instruction to identify a register file address
- This address specifies
  - either a register address in one of the banks of data RAM
  - or a location in the Access Bank

  as the data source for the instruction

# Programmer's Model-Direct Addressing

- a register address in one of the banks of data RAM

**FIGURE 5-7: USE OF THE BANK SELECT REGISTER (DIRECT ADDRESSING)**



# Programmer's Model-Indirect Addressing

- Full 12 bit address is written one of the four special function registers FSRx (LFSR instruction for PIC18)
- One of the following registers is used to get the content of the address pointed by FSR:
  - INDFx, POSTINCx, POSTDECx, PREINCx, PLUSWx
- Example : A sample program to clear RAM Bank1

# Indirect Addressing - Example

```
        LFSR        FSR0, 100h  ; Load FSR0

NEXT    CLRF        POSTINC0    ; Clear INDF
                                ; and increment FSR0
        BTFSS       FSR0H, 1    ; Test bit1 of FSR0
                                ; high byte, skip next
                                ; instruction if set
        BRA         NEXT        ; Loop until done

CONTINUE    ...
```

# Indirect Addressing - Example

- It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing.

- Indirect addressing is very convenient for manipulating data arrays located in GPR registers.
  - In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.

Write a program to copy the value 55H into RAM memory locations 40H to 45H using
(a) Direct addressing mode.
(b) Register indirect addressing mode without a loop.
(c) A loop.

**Solution:**

(a)

```
        MOVLW   0x55        ;load WREG with value 55H
        MOVWF   0x40        ;copy WREG to RAM location 40H
        MOVWF   0x41        ;copy WREG to RAM location 41H
        MOVWF   0x42        ;copy WREG to RAM location 42H
        MOVWF   0x43        ;copy WREG to RAM location 43H
        MOVWF   0x44        ;copy WREG to RAM location 44H
```

(b)

```
        MOVLW   55H         ;load with value 55H
        LFSR    0,0x40      ;load the pointer. FSR0 = 40H
        MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
        INCF    FSR0L,F     ;increment pointer. Now FSR0 = 41H
        MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
        INCF    FSR0L,F     ;increment pointer. Now FSR0 = 42H
        MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
        INCF    FSR0L,F     ;increment pointer. Now FSR0 = 43H
        MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
        INCF    FSR0L,F     ;increment pointer. Now FSR0 = 44H
        MOVWF   INDF0       ;copy W to RAM loc FSR0 points to
```

(c)

```
        COUNT   EQU 0x10    ;location 10H for counter
        MOVLW   0x5         ;WREG = 5
        MOVWF   COUNT       ;load the counter, Count = 5
        LFSR    0,0x40      ;load pointer. FSR0 = 40H, RAM address
        MOVLW   0x55        ;WREG = 55h value to be copied
B1      MOVWF   INDF0       ;copy WREG to RAM loc SFR0 points to
        INCF    FSR0L,F     ;increment FSR0 pointer
        DECF    COUNT,F     ;decrement the counter
        BNZ     B1          ;loop until counter = zero
```

# Introduction to the PIC18 Instruction Set

# PIC18 Instruction Set

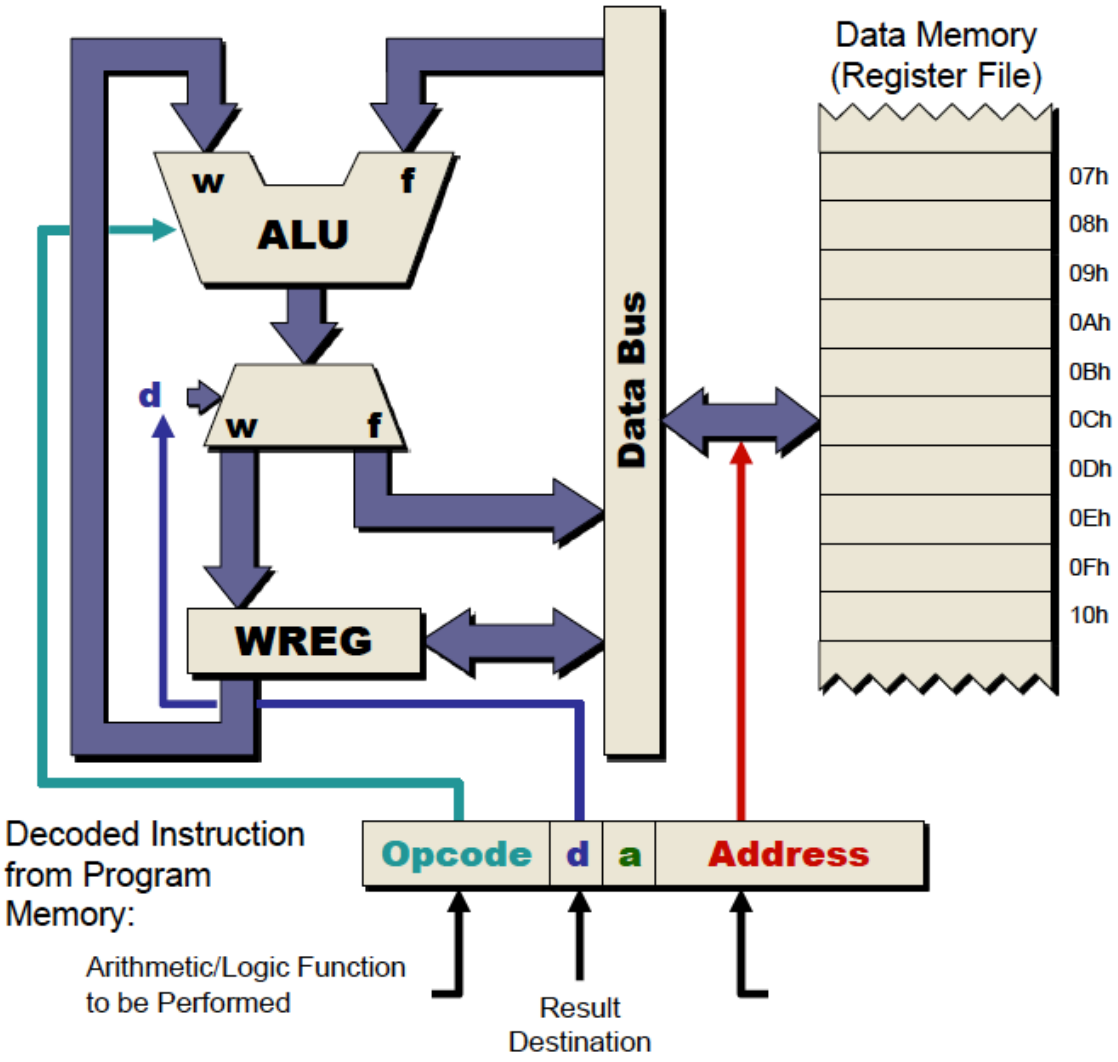
- Instruction format
- Instruction set
- Assembly Programming
- Example 1Example 2
- How it works?

# PIC18 Instruction Set

- Includes 77 instructions; 73 one word (16-bit) long and remaining 4 two words (32-bit) long

- Divided into seven groups
  - Move (Data Copy) and Load
  - Arithmetic
  - Logic
  - Program Redirection (Branch/Jump)
  - Bit Manipulation
  - Table Read/Write
  - Machine Control

# Instruction Format



# Instruction Format-*movwf* format

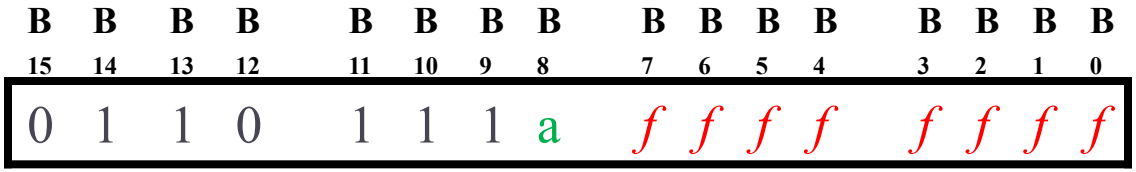
- "Write contents of W register to data memory location floc". General form:
  - movwf floc[,a] ; floc←(w)
    - floc (8 bits) is a memory location in the file registers (data memory)
    - W is the working register
    - a is data memory access bit, 'ACCESS' (0) use Access Bank --ignore Bank Select Register (BSR), 'BANKED' (1), use BSR. (will talk more about this later), [..] means optional usage.

- When floc is destination, means "modify memory location floc".
  - movwf 0x70 ;0x70 ←(w) (writes W to location 0x70)
  - Since a is not specified, it defaults to ACCESS (more later)

# *Instruction Format-movwf* format

movwf *floc* [,a]

*floc* ← (*w*)




*'ffff ffff'* lower 8-bit of *floc* address

a = 1 use Bank Select Register (BANKED)

a = 0 ignore BSR, just use (ACCESS)

machine code

| | |
|---|---|
| movwf 0x070, 0 | 0110 1110 0111 0000 = 0x6e70 |
| movwf 0x070, 1 | 0110 1111 0111 0000 = 0x6f70 |

# Instruction Format-Bank Select Register

- movwf 0x070, 1  also written as: movwf 0x070, **BANKED**

- The execution of the above instruction depends on the value in the Bank Select Register.

- If BSR = 0, then location 0x070 is addressed.
- If BSR = 1, then location 0x170 is addressed.
- If BSR = 2, then location 0x270 is addressed....etc.

- movwf 0x070, 0  also written as: movwf 0x070, **ACCESS**

- The execution of the above instruction does NOT depend on the value in the Bank Select Register, only the 8 bits in the machine code is used for the address location.
- Location 0x070 is always addressed.

# Instruction Format-example for *movwf*

**movwf 0x170**
**For this to work, BSR must be 0x1!**

**movwf 0x270**
**For this to work, BSR must be 0x2!**

The instruction mnemonics are different, but the machine code is the same! That is because machine code only uses lower 8-bits of the address!!!

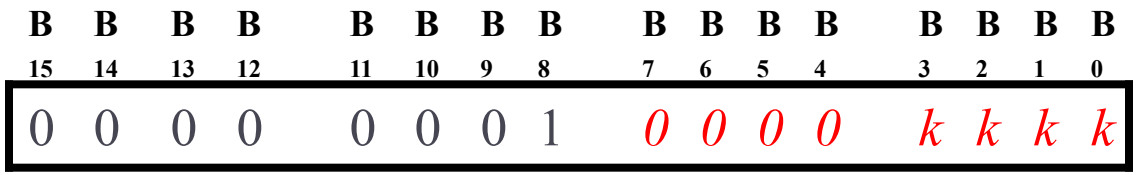
| mnemonic | | Machine code | |
|---|---|---|---|
| movwf | 0x070 | 0110 1110 0111 0000 = 0x6e70 | (a=0) |
| movwf | 0x170 | 0110 1111 0111 0000 = 0x6f70 | (a=1) |
| movwf | 0x270 | 0110 1111 0111 0000 = 0x6f70 | (a=1) |
| movwf | 0xF90 | 0110 1111 1001 0000 = 0x6e90 | (a=0) |

***By default (after processor reset), BSR = 0x0 !!!!.***

# *Instruction Format-movlb* Move Literal to Bank Reg.

movlb *k*

*BSR* ← *k*




*Move 4-bit literal k into BSR (only 16 banks, hence 4-bits)*

machine code

movlb 2 0000 0001 0000 0010= 0x0102

Example usage:

Selects bank 2

```
movlb 2
movwf 0x270
```

Causes the value stored in W to be written to location 0x270

# *Instruction Format-movf* Move to File

Copies a value from data memory to *w* or back to data memory.

movwf *floc* [,d[,a]

*d* ← (*floc*)

| B 15 | B 14 | B 13 | B 12 | B 11 | B 10 | B 9 | B 8 | B 7 | B 6 | B 5 | B 4 | B 3 | B 2 | B 1 | B 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | d | a | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* |

*'ffff ffff'* lower 8-bit of *floc* address

'd' : 0 = *W* reg, 1 = *f*

The second example looks useless as it just moves the contents of a memory location back onto itself. However, it **is useful, as will be seen later.**

| machine code | Instructions | |
|---|---|---|
| 0x501D | movf 0x01D,w | w ← [0x01D] |
| 0x521D | movf 0x01D,f | [0x1D] ← [0x01D] |

# Instruction Format-Copying Data Between Banks

Assume you want to copy data from location 0x1A0 to location 0x23F. Location 0x1A0 is in bank1, location 0x23F is in bank 2.

The HARD way:

```
movlb   0x1          ; select bank1
movf    0x1A0, w     ; w<-----[0x1A0]
movlb   0x2          ; select bank2
movwf   0x23F        ; [0x23F]<-----(w)
```

The EASY way:

```
movff   0x1A0, 0x23F    ; [0x23f]<----- [0x1A0]
```

The *movff* instruction copies the contents of a source location to a destination location

# Instruction Format-*movff* Move From File to File

movff $f_s, f_d$

$[f_d] \leftarrow [f_s]$

| B 15 | B 14 | B 13 | B 12 | B 11 | B 10 | B 9 | B 8 | B 7 | B 6 | B 5 | B 4 | B 3 | B 2 | B 1 | B 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *(src)* |
| 1 | 1 | 1 | 1 | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *f* | *(dest)* |

*Move contents of location $f_s$ to location $f_d$*

| machine code | Instructions |
|---|---|
| 0xC1A0<br>0xF23F | movff 0x1A0, 0x23F [0x23F] ⟵ [0x01D] |

Requires two instruction words (4 bytes). Only *movff, goto, call, lfsr* instructions take two words; all others take one word.

# *Instruction Format-addwf* Add W to File

General form:

addwf ***floc*** [, d[, a]  d←(floc)+ (w)

| | |
|---|---|
| *floc* | is a memory location in the file registers (data memory) |
| *w* | is the working register |
| *d* | is the destination, can either be the literal ‘f’ (1, default) or ‘w’ (0) |
| *a* | is data memory access bit |

```
addwf 0x070,w;w ←(0x070) + (w)
addwf 0x070,f  ;0x070 ←(0x070) + (w)
```

# *Instruction Format-addwf* Example

Assume Data memory contents on the right

***W*** register contains 0x1D

| Location | Contents |
|---|---|
| 0x058 | 0x2C |
| 0x059 | 0xBA |
| 0x05A | 0x34 |
| 0x05B | 0xD3 |

ALWAYS specify these in your instructions!!!!!!!

Execute: addwf 0x059, w  w ←[0x059] + (w)

w = (0x059) + (w) = 0xBA + 0x1D = 0xD7

After execution w = 0xD7, memory unchanged.

Execute: addwf 0x059, f  [0x059] ←[0x059] + (w)

[0x059] = [0x059] + (w) = 0xBA + 0x1D = 0xD7

After execution, location 0x059 contains 0xD7, w is unchanged.

# *Instruction Format-subwf* Subtract W from File

General form:

subwf ***floc*** [, d[, a]  d←(floc) - (w)

*floc* is a memory location in the file registers (data memory)

*w* is the working register

*d* is the destination, can either be the literal 'f' (1, default) or 'w' (0)

*a* is data memory access bit

```
subwf 0x070,w    ;w ←(0x070) - (w)
subwf 0x070,f    ;0x070 ←(0x070) - (w)
```

# Summarizing key points so far…

almost every instruction has two operands, source and destination:

| op-code | source | destination |
|---|---|---|
| movwf<br>addwf<br>movlb<br>etc…. | W register<br>Memory location (floc)<br>literal (immediate value) | W register<br>Memory location (floc) |

The memory location specified in the instructions (floc) is 8-bits only, so we need extra 4-bits from the BSR register.

The Data memory splitted into two parts:

BANKED; used for storing data during run time
ACCESS BANK; used when you need to access any SFR.

# PIC18 Instruction Set

- Instruction format
- Instruction set
- Assembly Programming
- How it works?

# PIC18 Instruction Set

- Includes 77 instructions; 73 one word (16-bit) long and remaining four two words (32-bit) long

- Divided into seven groups
  - Move (Data Copy) and Load
  - Arithmetic
  - Logic
  - Program Redirection (Branch/Jump)
  - Bit Manipulation
  - Table Read/Write
  - Machine Control

# Move and Load Instructions

| Op-Code | Description |
|---|---|
| movlw 8-bit | Load an 8-bit literal in WREG |
| Example : movlw 0 x F2 | |
| movwf F, a | Copy WREG in File (Data) Reg.<br>If a = 0, F is in Access Bank<br>If a = 1, Bank is specified by BSR |
| Example : movwf 0x25, 0 ;Copy W in Data Reg.25H | |
| movff fs, fd | Copy from one Data Reg. to another Data Reg. |
| Example : movff 0x20,0x30 ;Copy Data Reg. 20 into Reg.30 | |

# Arithmetic Instructions (1 of 3)

| Op-Code | Description |
|---|---|
| ADDLW 8-bit | Add 8-bit number to WREG |
| Example : ADDLW 0x32 | ;Add 32H to WREG |
| ADDWF F, d, a | Add WREG to File (Data) Reg.<br>Save result in W if d =0<br>Save result in F if d = 1 |
| Example : addwf 0x20, 1 | ;Add WREG to REG20 and save result in REG20 |
| Example : addwf 0x20, 0 | ;Add WREG to REG20 and save result in WREG |

# Arithmetic Instructions (2 of 3)

| Op-Code | Description |
|---|---|
| addwfc f, d, a | Add WREG to File Reg. with Carry and save result in W or F |
| sublw 8-bit | Subtract WREG from literal |
| subwf f, d, a | Subtract WREG from File Reg |
| subwfb f, d, a | Subtract WREG from File Reg. with Borrow |
| incf f, d, a | Increment File Reg. |
| decff, d, a | Decrement File Reg. |
| comf f, d, a | Complement File Reg. |
| negff, a | Take 2's Complement-File Reg. |

# Arithmetic Instructions (3 of 3)

| Op-Code | | Description |
|---|---|---|
| *mullw* | *8-bit* | ***Multiply 8-bit and WREG save result in PRODH-PRODL*** |
| *mulwf* | *f, a* | ***Multiply WREG and File Reg. save result in PRODH-PRODL*** |
| *daw* | | ***Decimal adjust WREG for BCD operations*** |

# Logic Instructions

| Op-Code | | Description |
|---|---|---|
| andlw | 8-bit | AND literal with WREG |
| andwf a | f, d, | AND WREG with File Reg. and save result in WREG/ File Reg |
| | | |
| iorlw | 8-bit | Inclusive OR literal with WREG |
| iorwf a | f, d, | Inclusive OR WREG with File Reg. and save result in WREG/File Reg. |
| | | |
| xorlw | 8-bit | Exclusive OR literal with WREG |
| xorwf a | f, d, | Exclusive OR WREG with File Reg. and save result in WREG/File Reg. |

# Branch Instructions

| Op-Code | Description |
|---|---|
| bc n | Branch if C flag = 1 within + or – 64 Words |
| bnc n | Branch if C flag = 0 within + or – 64 Words |
| bz n | Branch if Z flag = 1 within + or – 64 Words |
| bnz n | Branch if Z flag = 0 within + or – 64 Words |
| bn n | Branch if N flag = 1 within + or – 64 Words |
| bnn n | Branch if N flag = 0 within + or – 64 Words |
| bov n | Branch if OV flag = 1 within + or – 64 Words |
| bnov n | Branch if OV flag = 0 within + or – 64 Words |
| goto address | Branch to 20-bit address unconditionally |

# Call and Return Instructions

| Op-Code | Description |
|---|---|
| rcall nn | Call subroutine within +or – 512 words |
| call 20-bit, s | Call subroutine. If s = 1, save W, STATUS, and BSR |
| return, s | Return subroutine .If s = 1, retrieve W, STATUS, and BSR |
| retfie, s | Return from interrupt. If s = 1, retrieve W, STATUS, and BSR |

# Bit Manipulation Instructions

| Op-Code | Description |
|---|---|
| bcf f, b, a | Clear bit b of file register. b = 0 to 7 |
| bsf f, b, a | Set bit b of file register. b = 0 to 7 |
| btg f, b, a | Toggle bit b of file register. b = 0 to 7 |
| rlcf f, d, a | Rotate bits left in file register through carry and save in W or F register |
| rlncf f, d, a | Rotate bits left in file register and save in W or F register |
| rrcf f, d, a | Rotate bits right in file register through carry and save in W or F register |
| rrncf f, d, a | Rotate bits right in file register and save in W or F register |

# Test and Skip Instructions

| Op-Code | Description |
|---|---|
| btfsc f, b, a | Test bit b in file register and skip next instruction if bit is cleared (b =0) |
| btfss f, b, a | Test bit b in file register and skip next instruction if bit is set (b =1) |
| cpfseq f, a | Compare F with W, skip if F = W |
| cpfsgt f, a | Compare F with W, skip if F > W |
| cpfslt f, a | Compare F with W, skip if F < W |
| tstfsz f, a | Test F; skip if F = 0 |

# Incr/Decr and Skip Next Instrs.

| Op-Code | Description |
|---|---|
| decfsz f, b, a | Decrement file register and skip the next instruction if F = 0 |
| decfsnz f, b, a | Decrement file register and skip the next instruction if F ≠ 0 |
| incfsz f, b, a | Increment file register and skip the next instruction if F = 0 |
| incfsnz f, b, a | Increment file register and skip the next instruction if F ≠ 0 |
| | |
| | |

# Table Read/Write Instrs. (1 or 2)

| Op-Code | Description |
|---|---|
| tblrd* | Read Program Memory pointed by TBLPTR into TABLAT |
| tblrd*+ | Read Program Memory pointed by TBLPTR into TABLAT and increment TBLPTR |
| tblrd*- | Read Program Memory pointed by TBLPTR into TABLAT and decrement TBLPTR |
| tblrd+* | Increment TBLPTR and Read Program Memory pointed by TBLPTR into TABLAT |

# Table Read/Write Instrs. (2 or 2)

| Op-Code | Description |
|---|---|
| tblwt* | Write TABLAT into Program Memory pointed by TBLPTR |
| tblwt*+ | Write TABLAT into Program Memory pointed by TBLPTR and increment TBLPTR |
| tblwt*- | Write TABLAT into Program Memory pointed by TBLPTR and decrement TBLPTR |
| tblwt+* | Increment TBLPTR and Write TABLAT into Program Memory pointed by TBLPTR |

# Machine Control Instructions

| Op-Code | Description |
|---|---|
| CLRWDT | Clear Watchdog Timer |
| RESET | Reset all registers and flags |
| SLEEP | Go into standby mode |
| NOP | No operation |

# PIC18 Instruction Set

- Instruction format
- Instruction set
- Assembly Programming
- How it works?

# Assembly Programming

- *Byte ordering issues*
  - Two conventions for ordering the bytes within a word
    - Big Endian: the most significant byte of a variable is stored at the lowest address
    - Little Endian: the least significant byte of a variable is stored at the lowest address (PIC processors use little endian)

***Ex***: double word 87654321 (hex) in memory

Little

| | |
|---|---|
| 00003 | 87 |
| 00002 | 65 |
| 00001 | 43 |
| 00000 | 21 |

Big

| | |
|---|---|
| 00003 | 21 |
| 00002 | 43 |
| 00001 | 65 |
| 00000 | 87 |

# Assembly Programming-Example 1

**Example 1** Write a program that adds the three numbers stored in data registers at 0x20, 0x30, and 0x40 and places the sum in data register at 0x50.

**Pseudo Algorithm:**

**Step 1**

Load the number stored at 0x20 into the WREG register.

**Step 2**

Add the number stored at 0x30 and the number in the WREG register and leave the sum in the WREG register.

**Step 3**

Add the number stored at 0x40 and the number in the WREG register and leave the sum in the WREG register.

**Step 4**

Store the contents of the WREG register in the memory location at 0x50.

# Assembly Programming-Example 1

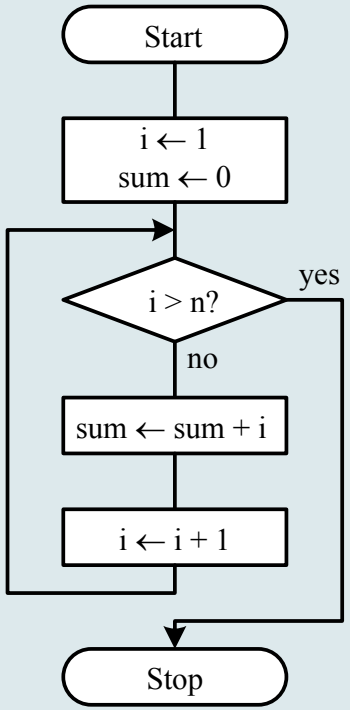
**The program that implements this algorithm is as follows:**

```
            #include <p18F8720.inc>   ; can be other processor
            org       0x00
            goto      start
            org       0x08
            retfie
            org       0x18
            retfie
start       movf      0x20,W,A        ; WREG ← [0x20]
            addwf     0x30,W,A        ; WREG ← [0x20] + [0x30]
            addwf     0x40,W,A        ; WREG ← [0x20] + [0x30] + [0x40]
            movwf     0x50,A          ; 0x50 ← sum (in WREG)
            end
```

# Assembly Programming-Example 2

**Example 2** Write a program to compute 1 + 2 + 3 + … + n and save the sum at 0x00 and 0x01.

**Program Logic:**



# Assembly Programming-Example 2

**Program** of Example 2 (in **for i = n1 to n2** construct)

```
            #include <p18F8720.inc>

            radix       dec

n           equ         D'50'
sum_hi      set         0x01        ; high byte of sum
sum_lo      set         0x00        ; low byte of sum
i           set         0x02        ; loop index i

            org         0x00        ; reset vector
            goto        start

            org         0x08
            retfie

            org         0x18
            retfie
```

# Assembly Programming-Example 2

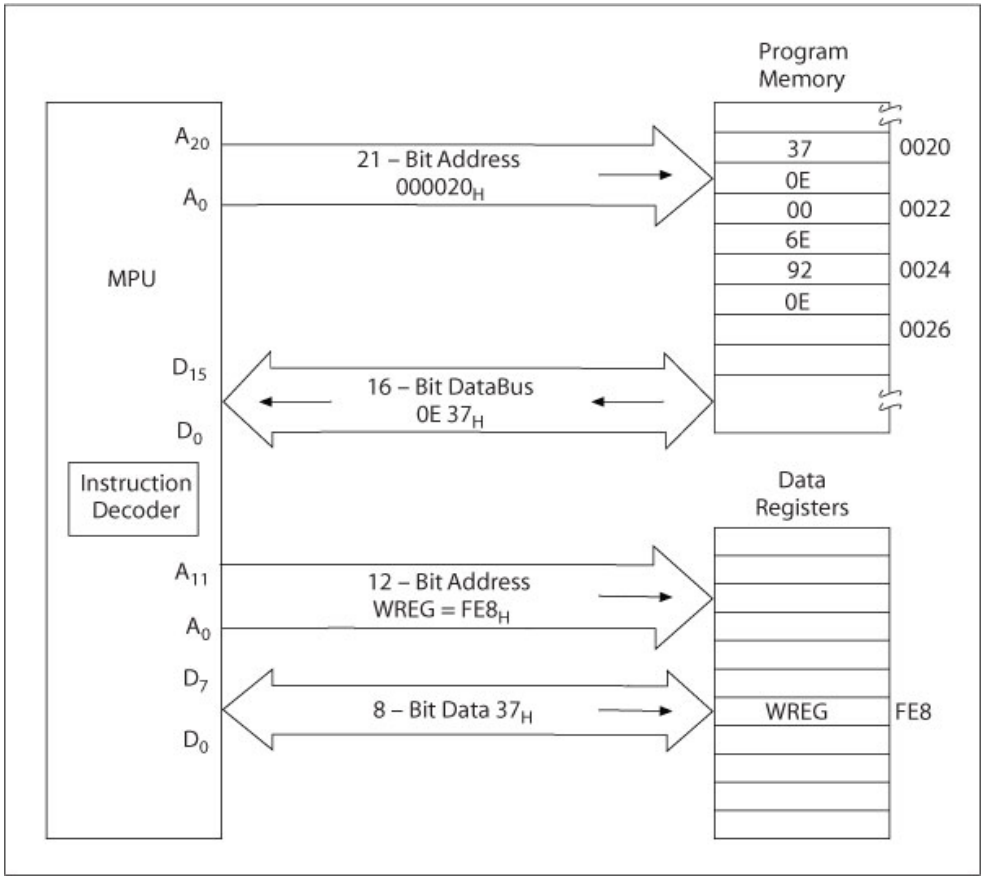
```
start     clrf    sum_hi          ; initialize sum to 0
          clrf    sum_lo          ;
          clrf    i               ; initialize i to 0
          incf    i,F             ; i starts from 1
sum_lp    movlw   n               ; place n in WREG
          cpfsgt  i               ; compare i with n and skip if i > n
          bra     add_lp          ; perform addition when i ≤ 50
          bra     exit_sum      ; it is done when i > 50
add_lp    movf    i,W             ; place i in WREG
          addwf   sum_lo,F        ; add i to sum_lo
          movlw   0
          addwfc  sum_hi,F        ; add carry to sum_hi
          incf    i,F             ; increment loop index i by 1
          bra     sum_lp
exit_sum  nop
          end
```

# PIC18 Instruction Set

- Instruction format
- Instruction set
- Assembly Programming
- How it works?

# Bus Contents During Execution

- Execution of MOVLW 0x37 instruction
- machine code
- 0x0E37



# Pipeline Fetch and Execution

Instruction Cycles:

| | Instruction Cycle 0 | Instruction Cycle 1 | Instruction Cycle 2 | Instruction Cycle 3 |
|---|---|---|---|---|
| | $Q_1 Q_2 Q_3 Q_4$ | $Q_1 Q_2 Q_3 Q_4$ | $Q_1 Q_2 Q_3 Q_4$ | $Q_1 Q_2 Q_3 Q_4$ |
| Fetch | $0E\ 37_H$ | $6E\ 00_H$ | $0E\ 92_H$ | |
| Execute | | $37_H \rightarrow W$ | $W \rightarrow Reg0$ | $92_H \rightarrow W$ |
| | Fetch 1 | Fetch 2 | Fetch 3 | |
| | | Execute 1 | Execute 2 | Execute 3 |

# Clock Cycles vs. Instruction Cycles

- The clock signal used by a PIC18 μC to control instruction execution can be generated by an off-chip oscillator (with a maximum clock frequency of 40 MHz).

An **instruction** cycle is four clock cycles.

- A PIC18 instruction takes 1 or 2 instruction cycles, depending on the instruction (see Table 20-2, PIC18Fxx2 data sheet).
- If an instruction causes the program counter to change, that instruction takes 2 instruction cycles. An add instruction takes 1 instruction cycle. How much time is this if the clock frequency is 20 MHz ?
  - 1/frequency = period, 1/20 MHz = 50 ns
  - Add instruction @ 20 MHz takes 4 * 50 ns = 200 ns (or 0.2 us).
- By comparison, a Pentium IV add instruction @ 3 GHz takes 0.33 ns (330 ps). A Pentium IV could emulate a PIC18Fxx2 faster than a PIC18Fxx2 can execute! But you can't put a Pentium IV in a toaster, or buy one from Digi-Key for $5.00.