03 - Instruction set and machine programming

- Programming model: instruction set and addressing modes
- Control structures, flags
- Subroutines; hardware stack: mechanisms, calling conventions
- Interrupt service Routines (hardware and software interrupts):
  - hardware and software mechanisms, IR vector table; special case: boot
- User and Supervisor mode of processors

An Overview of Programming Languages and Examples of Their Standards

- The hardware components within an embedded system can only directly transmit, store, and execute **machine code**, a basic language consisting of ones and zeros.
  - Machine code was used in earlier days to program computer systems, which made creating any complex application a long and tedious ordeal.
- In order to make programming more efficient, machine code was made visible to programmers through the creation of a hardware-specific set of instructions, where each instruction corresponded to one or more machine code operations.
- These hardware- specific sets of instructions were referred to as **assembly language.**

An Overview of Programming Languages and Examples of Their Standards

- Over time, other programming languages, such as C, C++, Java, etc., evolved with instruction sets that were (among other things) more hardware-independent.
  - These are commonly referred to as **highlevel languages** because
    - they are semantically further away from machine code,
    - they more resemble human languages, and
    - are typically independent of the hardware.
  - This is in contrast to a **low-level language**, such as assembly language, which more closely resembles machine code.
- Unlike high-level languages, low-level languages are hardware dependent, meaning there is a unique instruction set for processors with different architectures.

Evolution of programming languages

	Language	Details
1st Generation	Machine code	Binary (0,1) and hardware dependent.
2nd Generation	Assembly language	Hardware-dependent representing corresponding binary machine code.
3rd Generation	HOL (high-order languages)/procedural languages	High-level languages with more English-like phrases and more transportable, such as C, Pascal
4th Generation	VHLL (very high level languages)/ nonprocedural languages.	“Very” high-level languages: object-oriented languages (C++, Java,…), database query languages (SQL), etc
5th Generation	Natural languages	Programming similar to conversational languages, typically used in artificial intelligence (AI). Still in the research and development phases in most cases—not yet applicable in mainstream embedded systems.

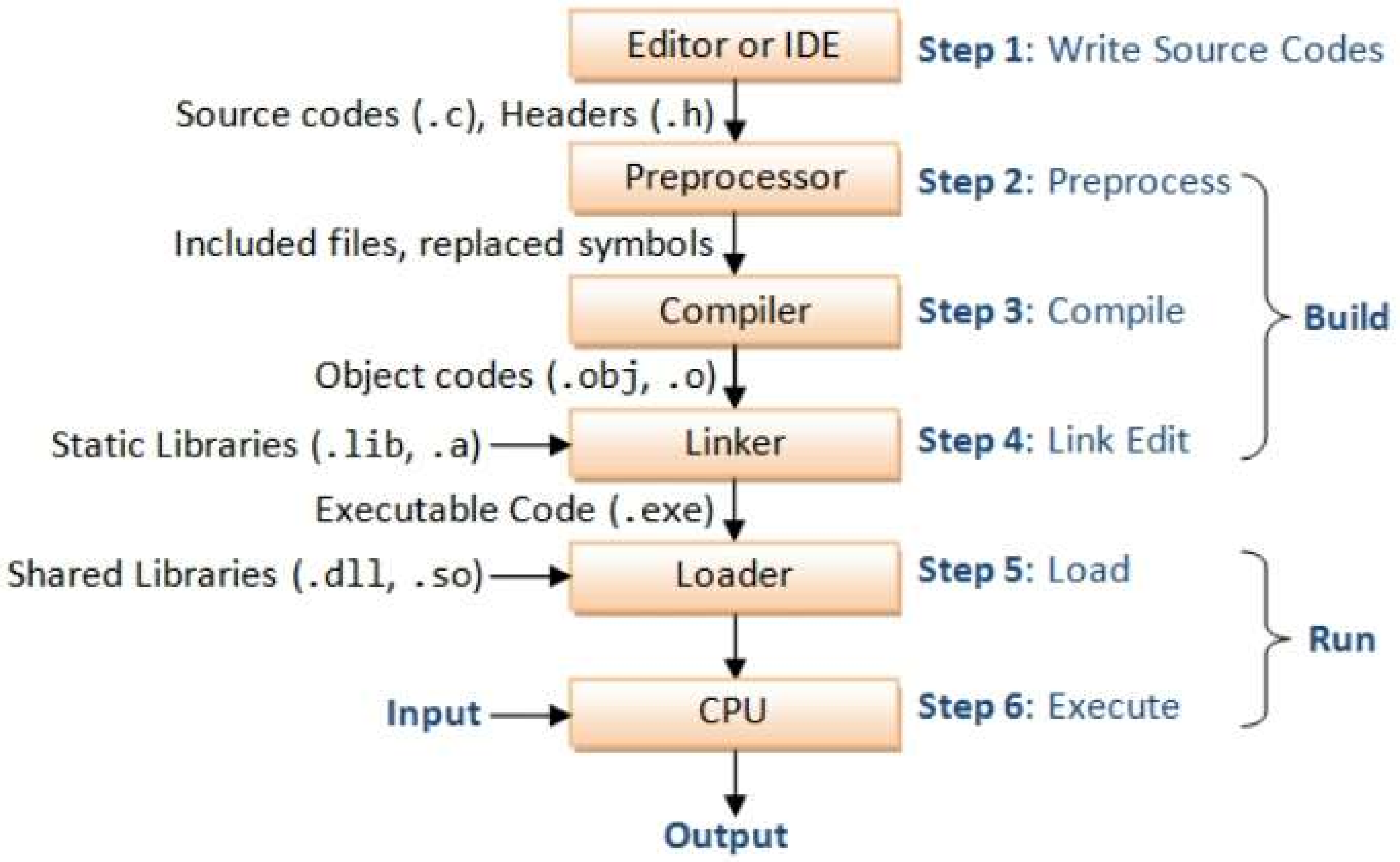
Machine Code Evolution

- Because machine code is the only language the hardware can directly execute, all other languages need some type of mechanism to generate the corresponding machine code.
- This mechanism usually includes one or some combination of
  - **preprocessing**,
  - **translation**
  - **interpretation**.
- Depending on the language, these mechanisms exist
  - **on the programmer's host system** (typically a non-embedded development system, such as a PC or Sparc station), or
  - **On the target system** (the embedded system being developed).

Preprocessor

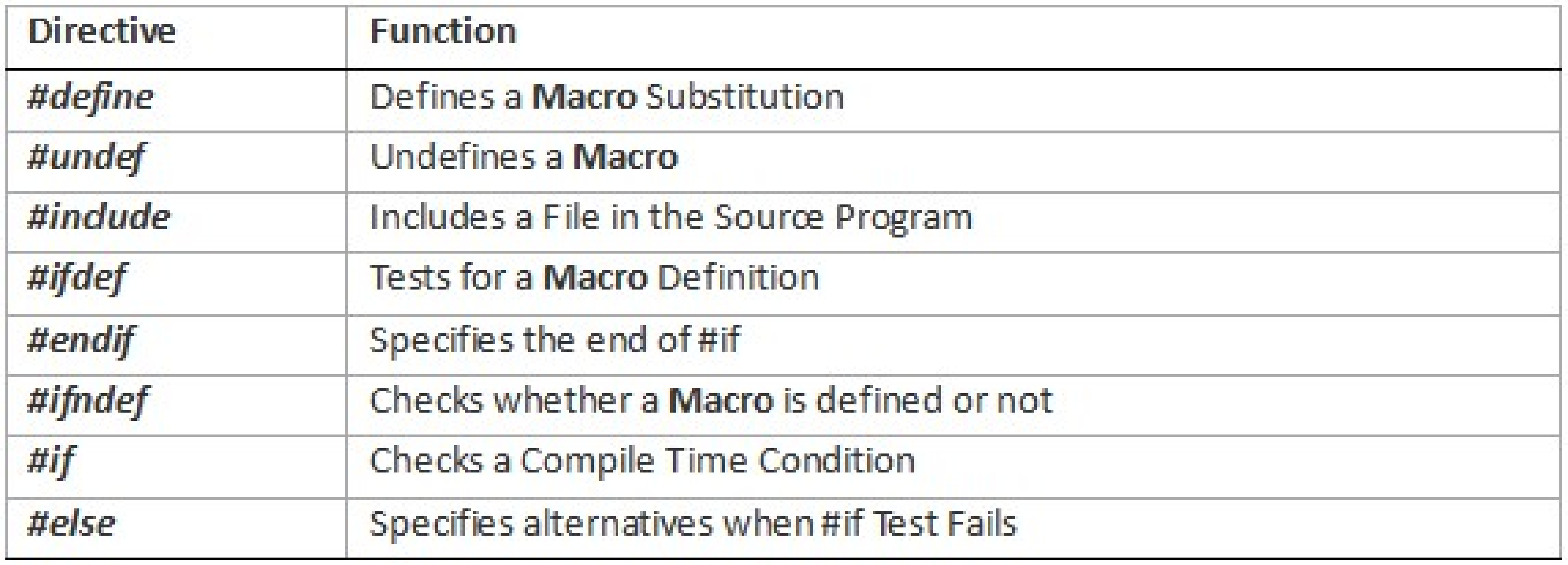
- Preprocessing is an optional step that occurs before either the translation or interpretation of source code, and whose functionality is commonly implemented by a **preprocessor.**
- The preprocessor's role is to organize and restructure the source code to make translation or interpretation of this code easier.
    - As an example, in languages like C and C++, it is a preprocessor that allows the use of named code fragments, such as ***macros***, that simplify code development by allowing the use of the macro's name in the code to replace fragments of code.
    - The preprocessor then replaces the macro name with the contents of the macro during preprocessing.
- The preprocessor can exist as a separate entity, or can be integrated within the translation or interpretation unit.

The Preprocessor – How it works



Preprocessor directives

- All preprocessor commands begin with a hash symbol (#).
- It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.



Directive	Function
#define	Defines a **Macro** Substitution
#undef	Undefines a **Macro**
#include	Includes a File in the Source Program
#ifdef	Tests for a **Macro** Definition
#endif	Specifies the end of #if
#ifndef	Checks whether a **Macro** is defined or not
#if	Checks a Compile Time Condition
#else	Specifies alternatives when #if Test Fails

Host and target system diagram

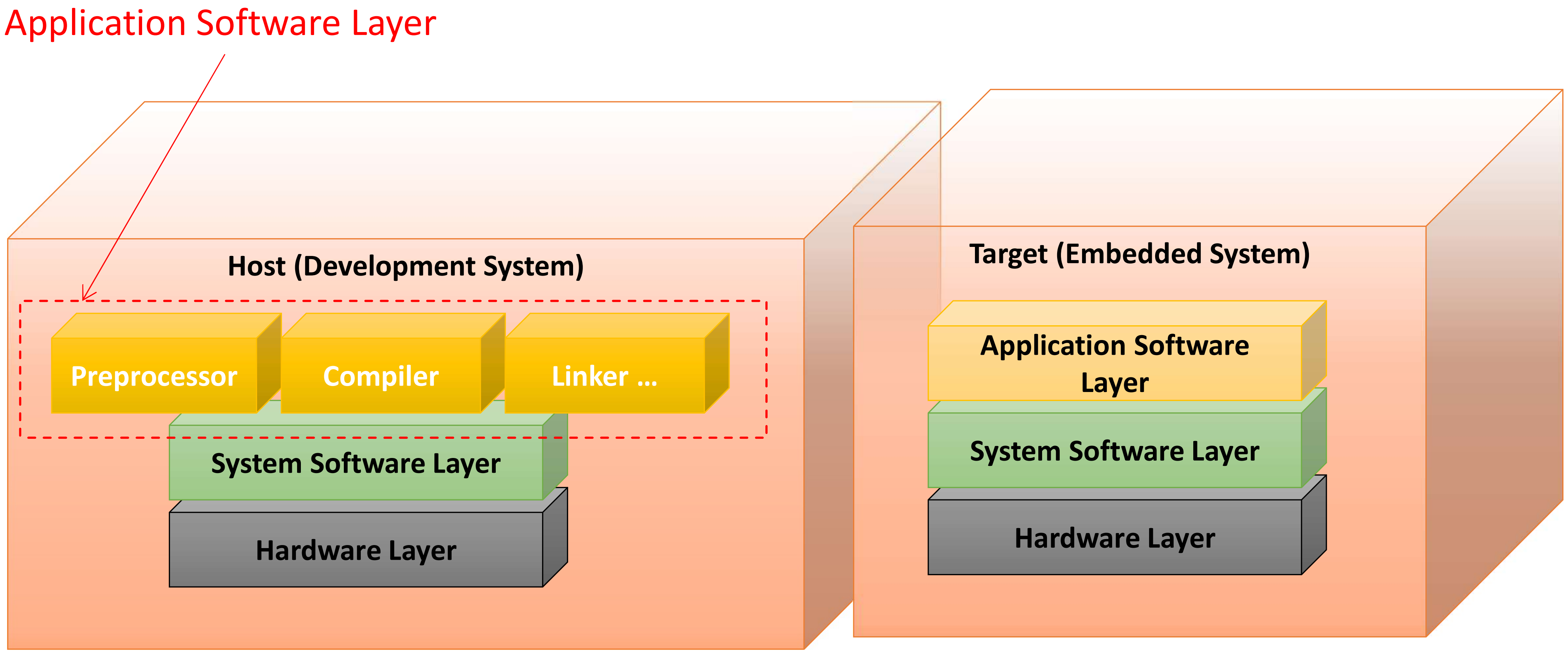


Figure 1: Host and target system diagram

Compiler

- Many languages convert source code, either directly or after having been preprocessed through use of a compiler
  - A compiler is a program that generates a particular target language—such as machine code and Java byte code — from the source language

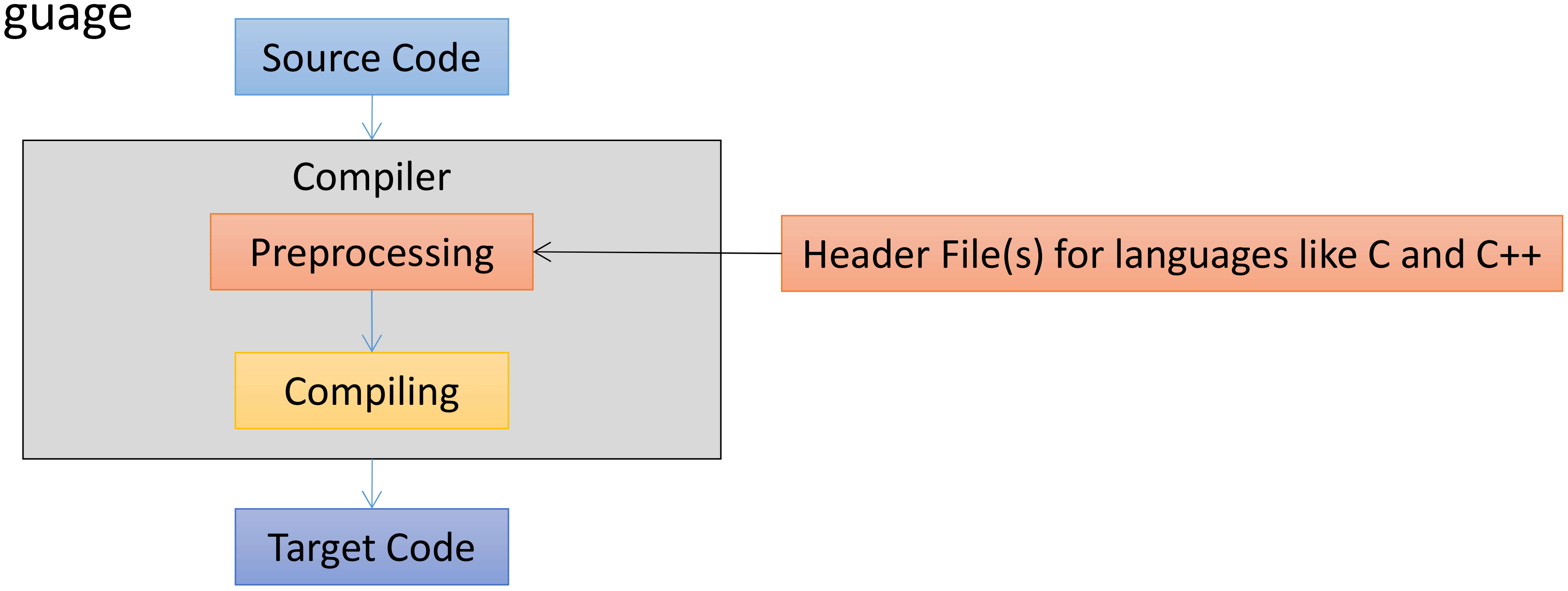


Figure 2: Compilation diagram

Compilers

- A compiler “translates” all of the source code to some target code at one time.
- Usually compilers are located on the programmer’s host machine and generate target code for hardware platforms that differ from the platform the compiler is actually running on.
  - These compilers are commonly referred to as **cross-compilers.**
- In the case of assembly language, the compiler is simply a specialized cross-compiler referred to as an **assembler**, and it always generates machine code.

Compilers

- Other high-level language compilers are commonly referred to by the language name plus the term “compiler,” such as “Java compiler” and “C compiler.”
- High-level language compilers vary widely in terms of what is generated.
  - Some generate machine code, while
  - others generate other high-level code, which then requires what is produced to be run through at least one more compiler or interpreter.
  - Other compilers generate assembly code, which then must be run through an assembler.

Linker

- After all the compilation on the programmer’s host machine is completed, the remaining target code file is commonly referred to as an **object file**, and can contain anything from machine code to Java byte code, depending on the programming language used.
- As shown in figure 3, after linking this object file to any system libraries required, the object file, commonly referred to as an **executable**, is then ready to be transferred to the target embedded system’s memory.

Linking

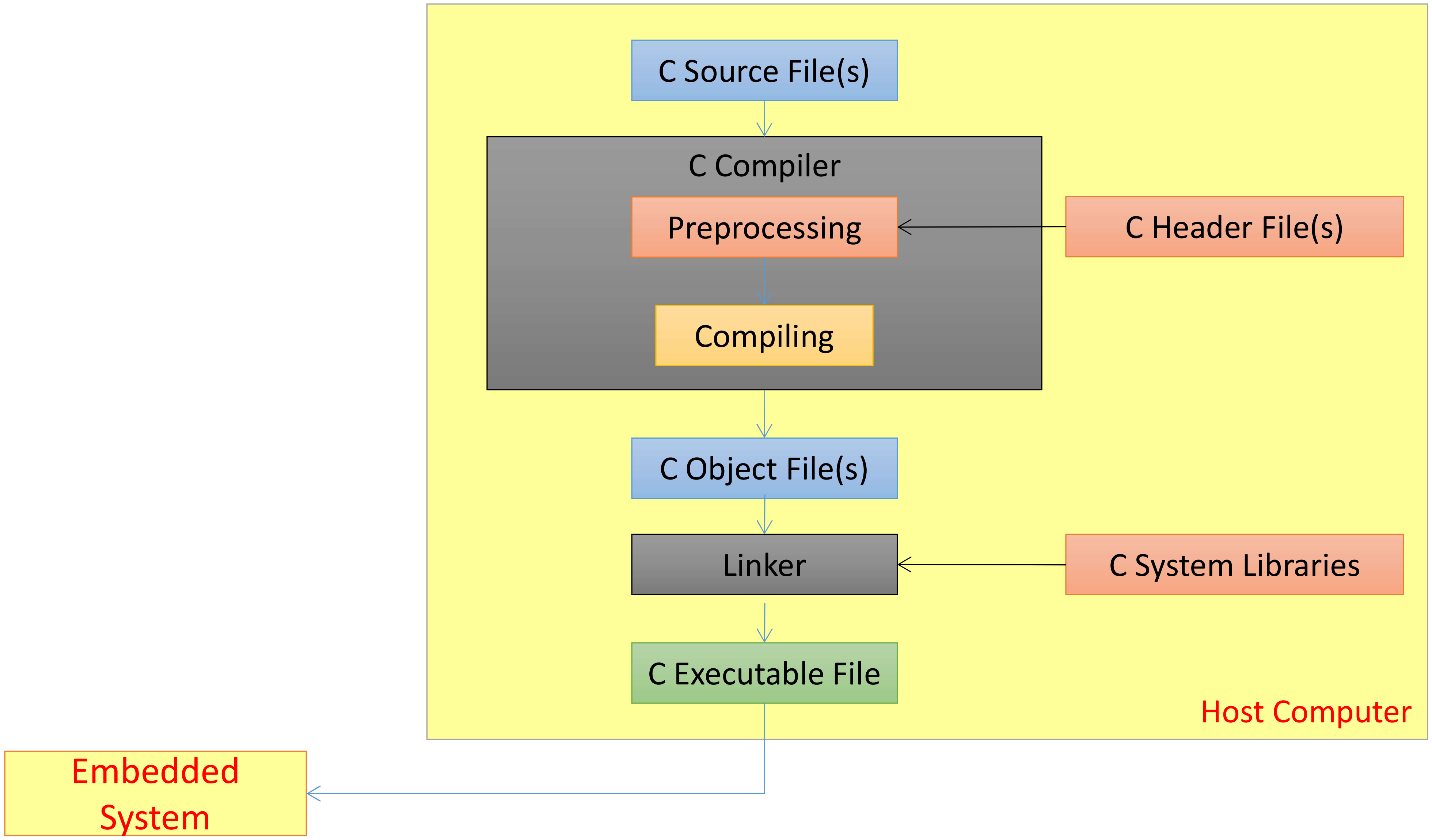


Figure 3: C Example compilation/linking steps and object file results

Interpreter

- With embedded platforms that support programs written in a scripting language, an additional component—an **interpreter**—must be included in the embedded system's architecture to allow for "on-the-fly" processing of code.
  - Such is the case with the embedded system architectural software stack shown, where an internet browser can contain both an HTML and JavaScript interpreter to process downloaded web pages.

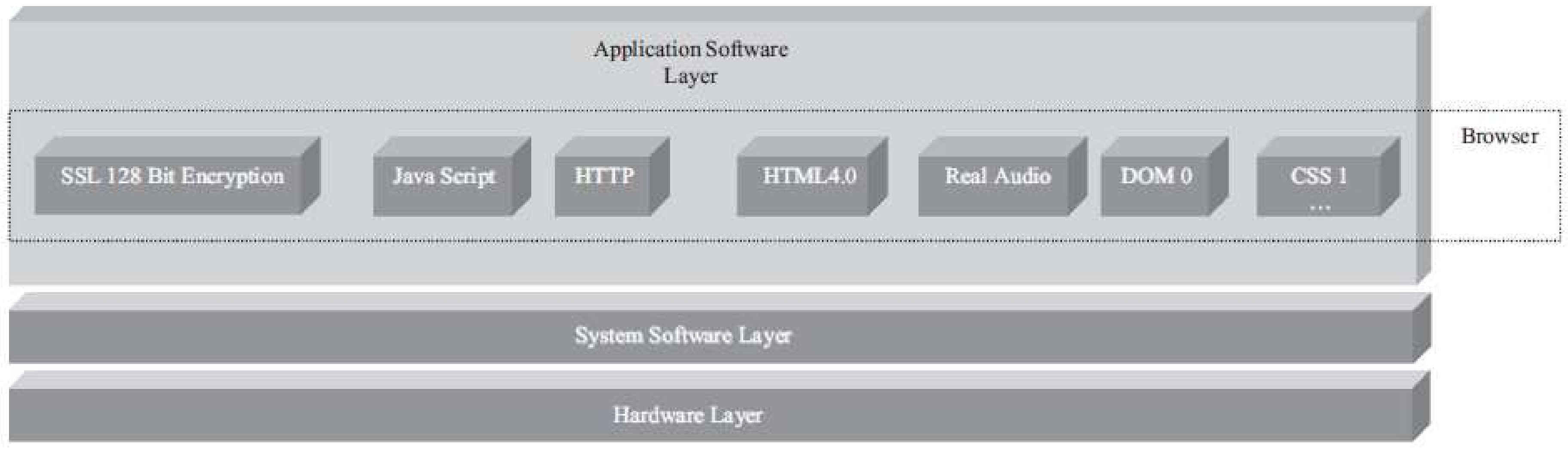


Figure 4: HTML and Javascript in the application layer

Flags

- All CPUs are equipped with a **special register** called status register or flag register
  - Reflects the current processing status
  - All the individual bits (called **flags**) are meaningful during the execution of logic and arithmetic operations
- The flags can be
  - Read and Written
- by
  - the CPU and/or the user (but in a restricted way)
- Not all the instructions affect all the flags (depends mainly on the philosophy of the manufacturer)
  - The common sense does not always work in this case

Flags

- The most common flags are:
  - Zero (affected when the result is zero)
  - Carry (affected if a carry or borrow) appeared.
    - Carry is in some cases seen as a 1 bit accumulator and usually has distinct instruction (Set_Cy, Clear_Cy, Add_with_Cy, etc)
  - Half carry or digit carry or auxiliary carry (affected if there has been a carry of the low-order of the nibble of the operand, 4th bit)
  - Negative (negative result⇔ MSb = 1)
  - Interrupt specific
  - Reset specific

STATUS Register – PIC16x

bit 7							bit 0
IRP	RP1	RP0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

IRP:	Register Bank Select (used for Indirect addressing)
	0 = Bank 0, 1 1 = Bank 2, 3
RP1:RP0:	Register Bank Select Bits (used for direct addressing)
	00 = Bank 0, 01 = Bank 1, 10 = Bank 2, 11 = Bank 3
$\overline{\text{TO}}$:	Time-out bit
	0 = A WDT time-out occurred
$\overline{\text{PD}}$:	Power-down bit
	0 = SLEEP instruction executed
Z:	Zero bit
	1 = Result of arithmetic operation is zero
DC:	Digit cary / $\overline{\text{borrow}}$ bit
	1 = Carry out of 4th low order bit occurred / No borrow occurred
C:	Carry / $\overline{\text{borrow}}$ bit
	1 = Carry out of MSb occurred / No borrow occurred

PIC

Other CPUs

8086 Flags

bit 15 bit 0

-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

- **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**.
- **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bits in result.
- **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
- **Zero Flag (ZF)** - set to **1** when result is **zero**.
- **Sign Flag (SF)** - set to **1** when result is **negative**.
  - (This flag takes the value of the most significant bit.)

8086 Flags (continued)

bit 15															bit 0
-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

- **Trap Flag (TF)** - Used for on-chip debugging.

- **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.

- **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.

- **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**.

Carry flag vs. Borrow flag

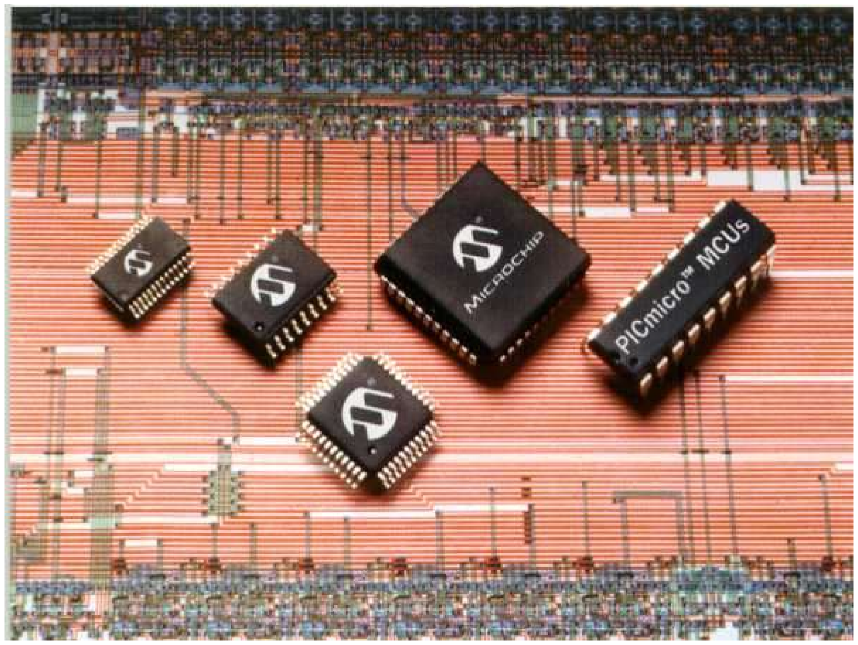
- While the sense of the carry flag is well-defined in addition, there are two popular ways to interpret the carry flag when subtracting.

1. Cy used as a **borrow flag**, setting it if a<b when computing a−b, and a borrow must be performed.
   - A subtract with borrow (SBB) instruction will compute a−b−C = a−(b+C), while a subtract without borrow (SUB) acts as if the borrow bit were clear.
   - The 68k, and x86 family of processors use a borrow bit.
2. The other takes advantage of the identity that −x = not(x)+1 and computes a−b as a+not(b)+1. Cy is set according to this addition, and subtract with Cy computes a+not(b)+C, while subtract without carry acts as if the carry bit were set.
   - The 6502 and PowerPC processors use this convention.

- The modern convention is to refer to the first alternative as a "borrow bit", while the second is called a "carry bit". However, there are exceptions in both directions.

PIC Microcontrollers - Architecture

- The high performance of the PIC microcontrollers can be attributed to the following architectural features:
  - Harvard Architecture
  - Instruction Pipelining
  - Large Register File
  - Single Cycle Instructions
  - Single Word Instructions
  - Long Word Instructions
  - Reduced Instruction Set
  - Orthogonal Instruction Set

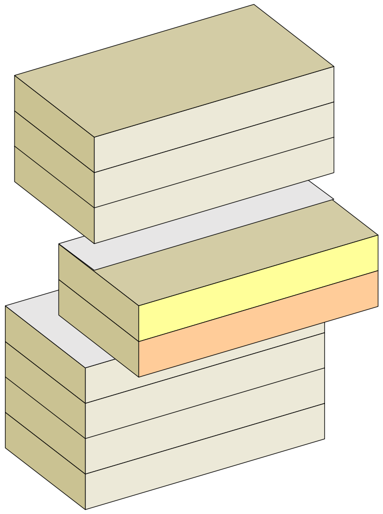


PIC

Other CPUs

Long Word Instruction

8-bit Program Memory



8-bit Instruction on typical 8-bit MCU

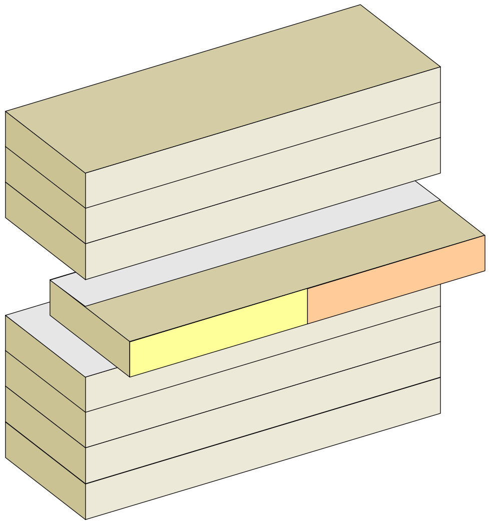
Example: Freescale 'Load Accumulator A':
- 2 Program Memory Locations
- 2 Instruction Cycles to Execute

```
ldaa  #k
```

1	0	0	0	0	1	1	0
k	k	k	k	k	k	k	k

- Limits Bandwidth
- Increases Memory Size Requirements

14-bit Program Memory



14-bit Instruction on PIC16 8-bit MCU

Example: 'Move Literal to Working Register'
- 1 Program Memory Location
- 1 Instruction Cycle to Execute

```
movlw  k
```

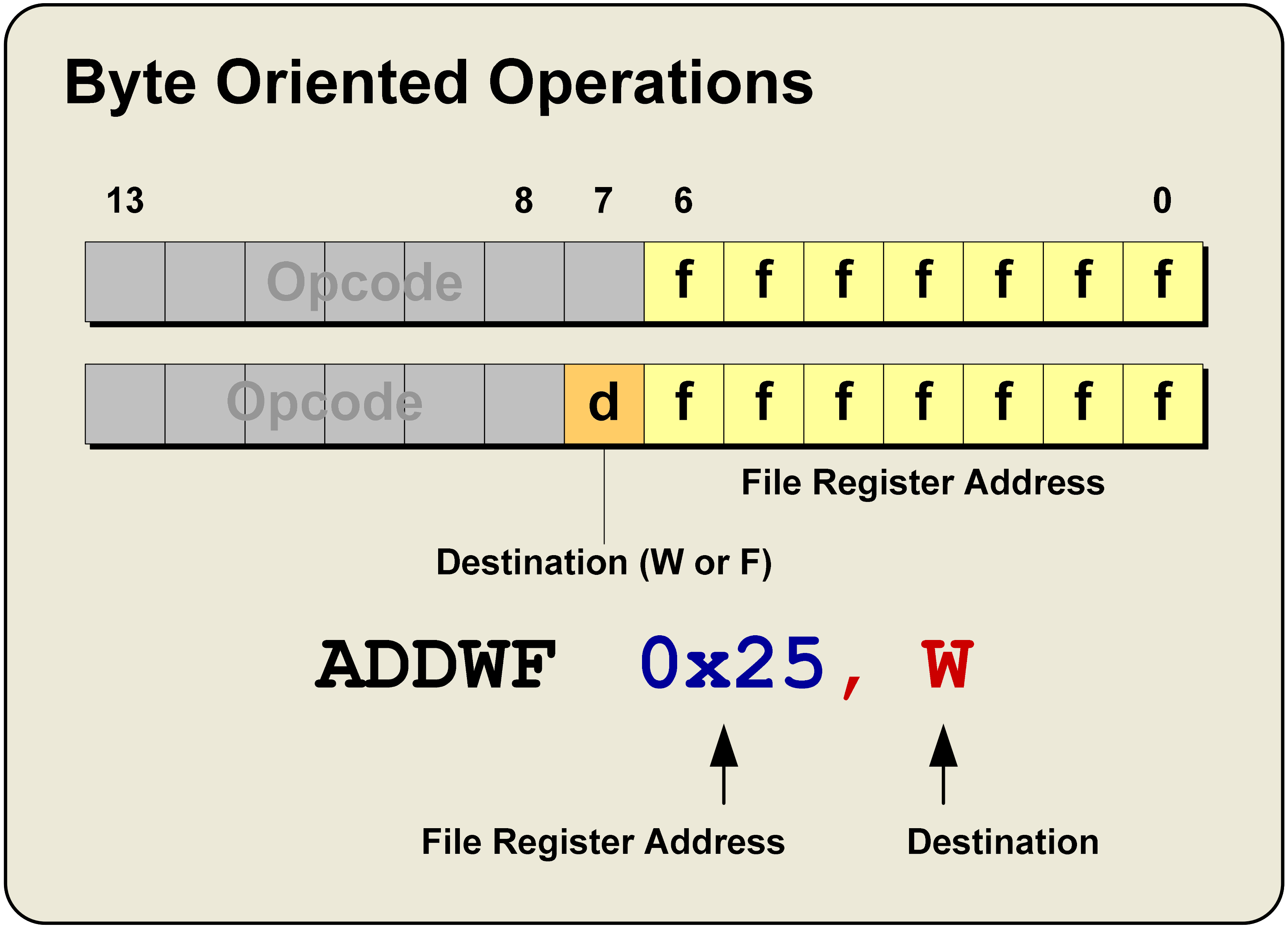
1	1	0	0	0	0	k	k	k	k	k	k	k	k

- Separate busses allow different widths
- 2k x 14 is roughly equivalent to 4k x 8

PIC

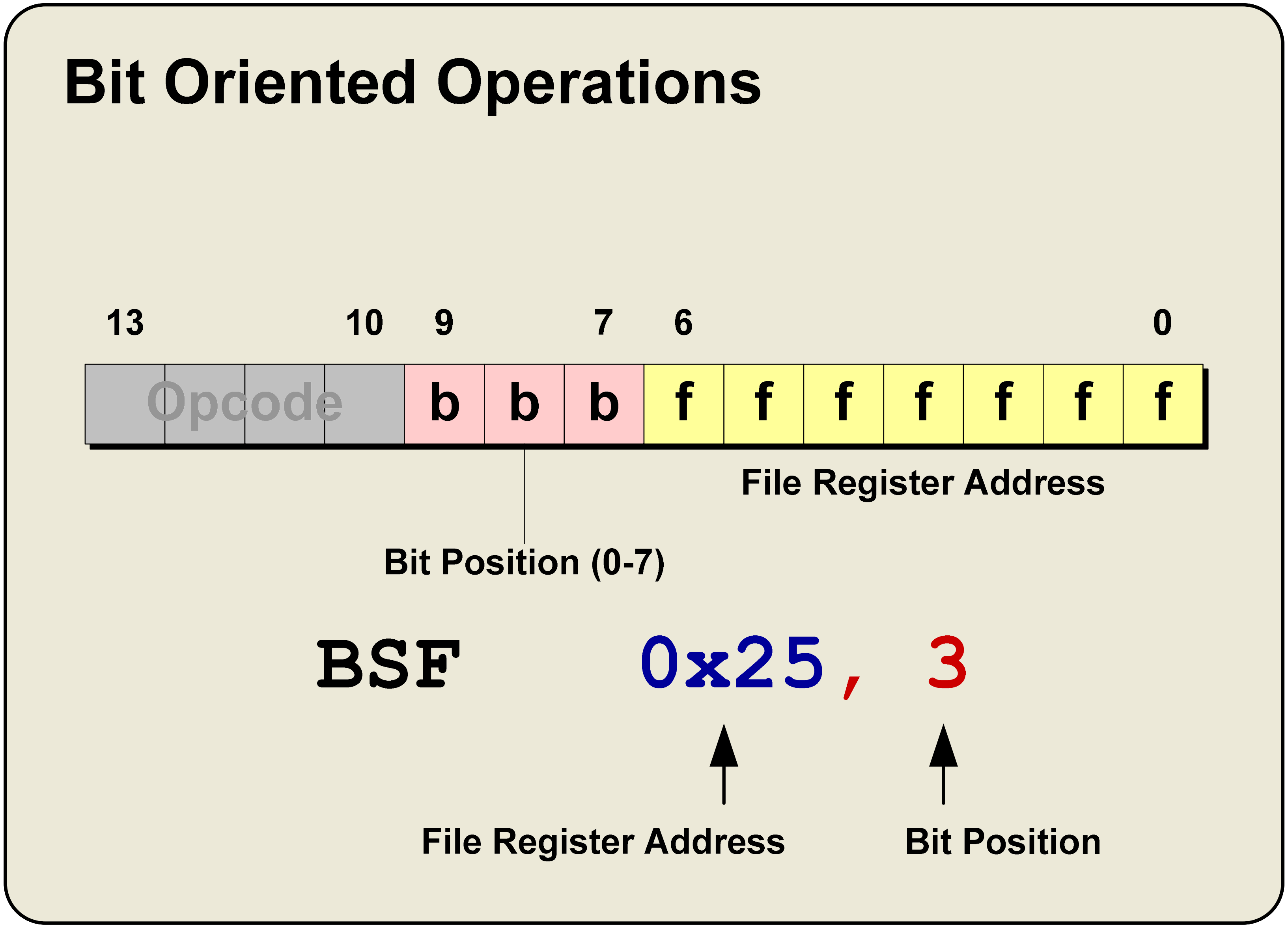
Other CPUs

Instruction Set Overview



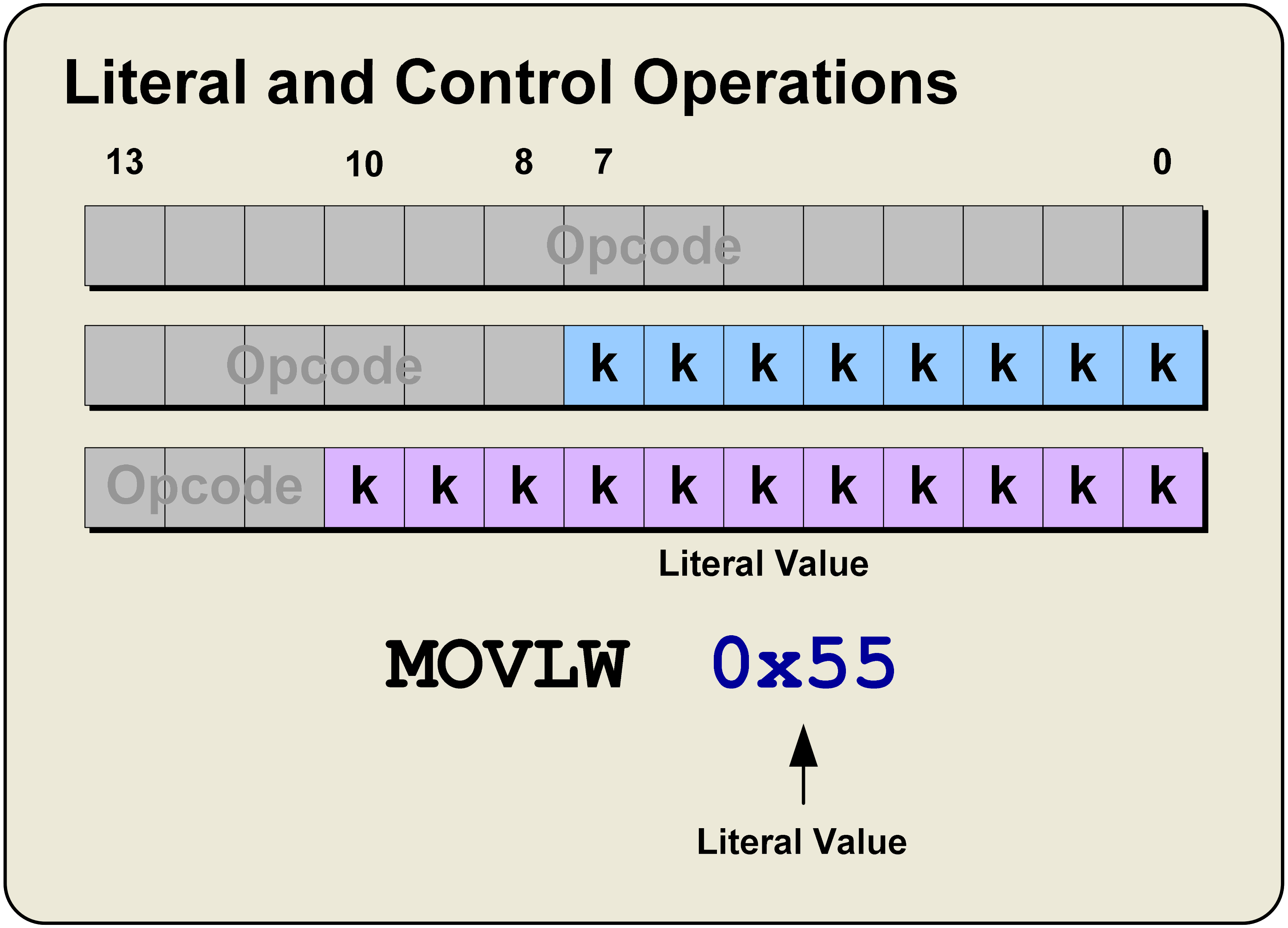
PIC

Instruction Set Overview



PIC

Instruction Set Overview



PIC

PIC16 Instruction Set

Byte Oriented Operations		
addwf	f,d	Add W and f
andwf	f,d	AND W with f
clrf	f	Clear f
clrw	-	Clear W
comf	f,d	Complement f
decf	f,d	Decrement f
decfsz	f,d	Decrement f, Skip if 0
incf	f,d	Increment f
incfsz	f,d	Increment f, Skip if 0
iorwf	f,d	Inclusive OR W with f
movf	f,d	Move f
movwf	f	Move W to f
nop	-	No Operation
rlf	f,d	Rotate Left f through Carry
rrf	f,d	Rotate Right f through Carry
subwf	f,d	Subtract W from f
swapf	f,d	Swap nibbles in f
xorwf	f,d	Exclusive OR W with f

Bit Oriented Operations		
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		
addlw	k	Add literal and W
andlw	k	AND literal with W
call	k	Call subroutine
clrwdt	-	Clear Watchdog Timer
goto	k	Go to address
iorlw	k	Inclusive OR literal with W
movlw	k	Move literal to W
retfie	-	Return from interrupt
retlw	k	Return with literal in W
return	-	Return from Subroutine
sleep	-	Go into standby mode
sublw	k	Subtract W from literal
xorlw	k	Exclusive OR literal with W

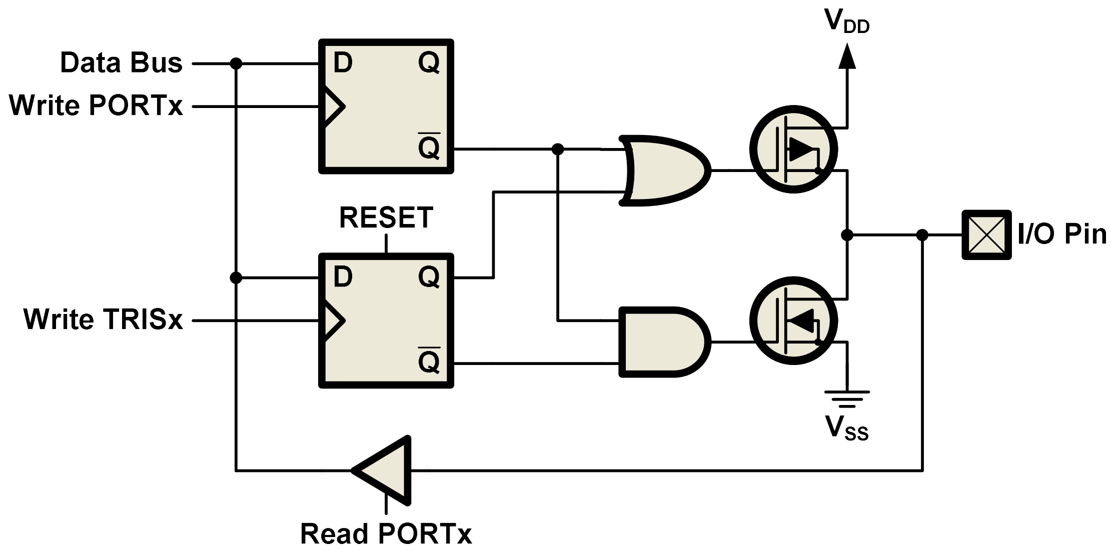
PIC

12-BIT/14-BIT Pseudo Instruction Mnemonics

Mnemonic		Descrip.	Equivalent Operation(s)		Status
ADDCF	f,d	Add Carry to File	BTFSC INCF	3,0 f,d	Z
ADDDCF	f,d	Add Digit Carry to File	BTFSC INCF	3,1 f,d	Z
B	k	Branch	GOTO	k	-
BC	k	Branch on Carry	BTFSC GOTO	3,0 k	-
BDC	k	Branch on Digit Carry	BTFSC GOTO	3,1 k	-
BNC	k	Branch on No Carry	BTFSS GOTO	3,0 k	-
BNDC	k	Branch on No Digit Carry	BTFSS GOTO	3,1 k	-
BNZ	k	Branch on No Zero	BTFSS GOTO	3,2 k	-
BZ	k	Branch on Zero	BTFSC GOTO	3,2 k	-
CLRC		Clear Carry	BCF	3,0	-
CLRDC		Clear Digit Carry	BCF	3,1	-
CLRZ		Clear Zero	BCF	3,2	-
LCALL	k	Long Call	BCF/BSF BCF/BSF CALL	0x0A,3 0x0A,4 k	
LGOTO	k	Long GOTO	BCF/BSF BCF/BSF GOTO	0x0A,3 0x0A,4 k	

Mnemonic		Descrip.	Equivalent Operation(s)		Status
MOVFW	f	Move File to W	MOVF	f,0	Z
NEGF	f,d	Negate File	COMF INCF	f,1 f,d	Z
SETC		Set Carry	BSF	3,0	-
SETDC		Set Digit Carry	BSF	3,1	-
SETZ		Set Zero	BSF	3,2	-
SKPC		Skip on Carry	BTFSS	3,0	-
SKPDC		Skip on Digit Carry	BTFSS	3,1	-
SKPNC		Skip on No Carry	BTFSC	3,0	-
SKPNDC		Skip on No Digit Carry	BTFSC	3,1	-
SKPNZ		Skip on Non Zero	BTFSC	3,2	-
SKPZ		Skip on Zero	BTFSS	3,2	-
SUBCF	f,d	Subtract Carry from File	BTFSC DECF	3,0 f,d	Z
SUBDCF	f,d	Subtract Digit Carry from File	BTFSC DECF	3,1 f,d	Z
TSTF	f	Test File	MOVF	f,1	Z

I/O Ports

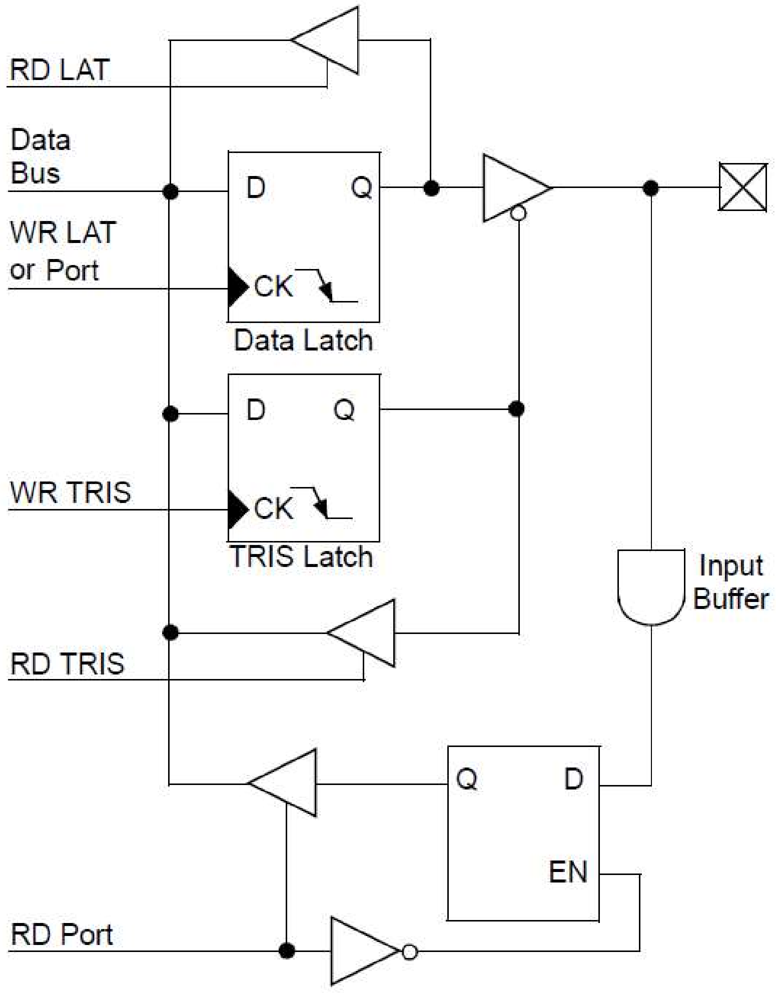


- High Drive Capability
- Can directly drive LEDs
- Direct, single cycle bit manipulation
- Each pin has individual direction control under software
- All pins have ESD protection diodes
- Pin RA4 is usually open drain
- All I/O pins default to inputs (high impedance) on startup
- All pins multiplexed with analog functions default to analog inputs on startup

PIC

Other CPUs

Generic I/O Port Operation



Addressing Modes

- specifies how to calculate the effective memory address of an operand by using
  - information held in registers and/or
  - constants contained within a machine instruction or elsewhere
- addressing modes are primarily of interest to
  - compiler writers and
  - to those who write code directly in assembly language.
- there is no generally accepted way of naming the various addressing modes.
- different authors and computer manufacturers may
  - give different names to the same addressing mode, or
  - the same names to different addressing modes.
- Furthermore, an addressing mode which, in one given architecture, is treated as a single addressing mode may represent functionality that, in another architecture, is covered by two or more addressing modes

How many addressing modes?

- Different computer architectures vary greatly as to the number of addressing modes they provide in hardware.
  - There are some benefits to eliminating complex addressing modes and using only one or a few simpler addressing modes, even though it requires a few extra instructions, and perhaps an extra register.
  - It has proven much easier to design pipelined CPUs if the only addressing modes available are simple ones.
- Most RISC machines have only about five simple addressing modes, while CISC machines have over a dozen addressing modes, some of which are quite complicated.
  - The IBM System/360 mainframe had only three addressing modes; a few more have been added for the System/390.

Addressing Modes

- Operands can be referenced in a variety of manners, called addressing modes, depending on the type of instruction and the type of operand.
- Some types of instructions inherently use only one addressing mode, and some types have multiple modes.
- The manners of referencing operands can be categorized into six basic addressing modes:
    - implied
    - immediate
    - direct
    - relative
    - indirect
    - indexed

Implied addressing

- Implied addressing specifies the operand of an instruction as an inherent property of that instruction.
- For example, CLRA implies the accumulator by definition. No additional addressing information following the opcode is needed.

```
         .ORIG   $100
BEGIN    CLRA
INC_LOOP INCA
         CMPA   #$1E        ; compare ACCA = $1E
         BNE    INC_LOOP    ; if <>, go back
         LDAA   #'Z'        ; else, load ASCII 'Z'
         BSR    SEND_CHAR   ; send to serial port
         BRA    BEGIN       ; start over again
```

Immediate addressing

- Immediate addressing places an operand’s value literally into the instruction sequence.

```
LDAA   #'Z'
```

- has its primary operand immediately available following the opcode.
  - An immediate operand is indicated with the **#** prefix in some assembly languages.
  - Eight-bit microprocessors with eight-bit instruction words cannot fit an immediate value into the instruction word itself and, therefore, require that an extra byte following the opcode be used to specify the immediate value.
  - More powerful 32-bit microprocessors can often fit a 16-bit or 24-bit immediate value within the instruction word. This saves an additional memory fetch to obtain the operand.

Direct addressing

- Direct addressing places the address of an operand directly into the instruction sequence.
- Instead of specifying

```
LDAA    #'Z' ;Immediate addressing
```

- the programmer could specify

```
LDAA $1234     ;Direct addressing
```

- This version of the instruction would tell the microprocessor to read memory location $1234 and load the resulting value into the accumulator.
- The operand is directly available by looking into the memory address specified just following the instruction.

Direct addressing

- Direct addressing is useful when there is a need to read a fixed memory location.
- Usage of the direct addressing mode has a slightly different impact on various microprocessors.
  - A typical 8-bit microprocessor has a 16-bit address space, meaning that two bytes following the opcode are necessary to represent a direct address. The 8-bit microprocessor will have to perform two additional 8-bit fetch operations to load the direct address.
  - A typical 32-bit microprocessor has a 32-bit address space, meaning that 4 bytes following the opcode are necessary. If the 32-bit microprocessor has a 32-bit data bus, only one additional 32-bit fetch operation is required to load the direct address.

Relative addressing

- Relative addressing places an operand's relative address into the instruction sequence.
  - A relative address is expressed as a signed offset relative to the current value of the PC.
- Relative addressing is often used by branch instructions, because the target of a branch is usually within a short distance of the PC, or current instruction.

```
BNE  INC_LOOP
```

  - results in a branch-if-notequal backward by two instructions.
- The assembler automatically resolves the addresses and calculates a relative offset to be placed following the `BNE` opcode.
  - This relative operation is performed by adding the offset to the PC.
  - The new PC value is then used to resume the instruction fetch and execution process.

Relative addressing

- Relative addressing can utilize both positive and negative deltas that are applied to the PC.
- A microprocessor’s instruction format constrains the relative range that can be specified in this addressing mode.
    - For example, most 8-bit microprocessors provide only an 8- bit signed field for relative branches, indicating a range of +127/–128 bytes.
    - The relative delta value is stored into its own byte just after the opcode.
- Many 32-bit microprocessors allow a 16-bit delta field and are able to fit this value into the 32-bit instruction word, enabling the entire instruction to be fetched in a single memory read.

Relative addressing

- Limiting the range of a relative operation is generally not an excessive constraint because of software's *locality property.*
- Locality in this context means that the set of instructions involved in performing a specific task are generally relatively close together in memory.
  - The locality property covers the great majority of branch instructions.
  - For those few branches that have their targets outside of the allowed relative range, it is necessary to perform a short relative branch to a long jump instruction that specifies a direct address.
  - This reduces the efficiency of the microprocessor by having to perform two branches when only one is ideally desired, but the overall efficiency of saving extra memory accesses for the majority of short branches is worth the trade-off.

Indirect addressing

- Indirect addressing specifies an operand’s direct address as a value contained in another register.
  - The other register becomes a pointer to the desired data.
- For example, a microprocessor with two accumulators can load ACCA with the value that is at the address in ACCB.

```
LDAA  (ACCB)
```

  - would tell the microprocessor to put the value of accumulator B onto the address bus, perform a read, and put the returned value into accumulator A.
- Indirect addressing allows writing software routines that operate on data at different addresses.
- If a programmer wants to read or write an arbitrary entry in a data table, the software can load the address of that entry into a microprocessor register and then perform an indirect access using that register as a pointer.
- Some microprocessors place constraints on which registers can be used as references for indirect addressing.

Indexed addressing

- Indexed addressing is a close relative (no pun intended) of indirect addressing, because it also refers to an address contained in another register.
- However, indexed addressing also specifies an offset, or index, to be added to that register base value to generate the final operand address: base + offset = final address.
- Some microprocessors allow general accumulator registers to be used as base-address registers, but others provide special index registers for this purpose.

Indexed addressing

- In many 8-bit microprocessors, a full 16-bit address cannot be obtained from an 8-bit accumulator serving as the base address.
  - Therefore, one or more separate index registers are present for the purpose of indexed addressing.
- In contrast, many 32-bit microprocessors are able to specify a full 32-bit address with any general-purpose register and place no limitations on which register serves as the index register.
- Indexed addressing builds upon the capabilities of indirect addressing by enabling multiple address offsets to be referenced from the same base address.

Indexed addressing

```
LDAA  (X+$20)
```

would tell the microprocessor to add $20 to the index register, X, and use the resulting address to fetch data to be loaded into ACCA.

- One simple example of using indexed addressing is a subroutine to add a set of four numbers located at an arbitrary location in memory.
  - Before calling the subroutine, the main program can set an index register to point to the table of numbers.
  - Within the subroutine, four individual addition instructions use the indexed addressing mode to add the locations X+0, X+1, X+2, and X+3.
  - When so written, the subroutine is flexible enough to be used for any such set of numbers.
- Because of the similarity of indexed and indirect addressing, some microprocessors merge them into a single mode and obtain indirect addressing by performing indexed addressing with an index value of zero.

Comments

- Each individual microprocessor applies these addressing modes differently.
  - Some combine multiple modes into a single mode (e.g., indexed and indirect),
  - some will create multiple submodes out of a single mode.
- The exact variation depends on the specifics of an individual microprocessor's architecture.
- With the various addressing modes modifying the specific opcode and operands that are presented to the microprocessor, the benefits of using assembly language over direct binary values can be observed.
- The programmer does not have to worry about calculating branch target addresses or resolving different addressing modes.
- Each mnemonic can map to several unique opcodes, depending on the addressing mode used.
  - For example, the LDAA instruction already discussed could easily have used *extended* addressing by specifying a full 16-bit address at which the ASCII transmit-value is located.
- Because labels are resolved each time the program is assembled, small changes to the program can be made that add or remove instructions and labels, and the assembler will automatically adjust the resulting addresses accordingly.

Comments

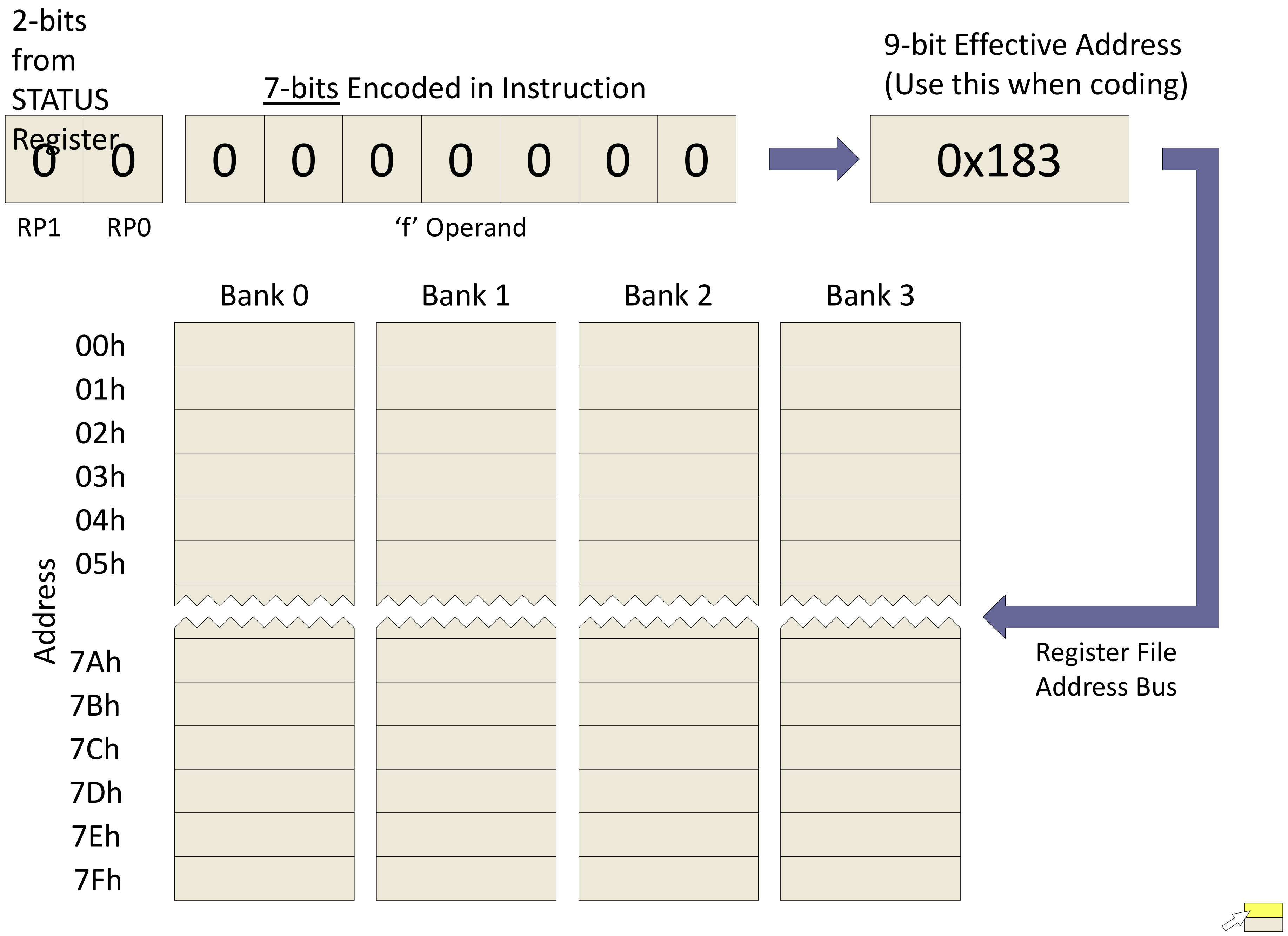
- Programming in assembly language is different from using a high-level language, because one must think in smaller steps and have direct knowledge about the microprocessor’s operation and architecture.
- Assembly language is processor-specific instead of generic, as with a high-level language
- Therefore, assembly language programming is usually restricted to special cases such as boot code or routines in which absolute efficiency and performance are demanded.
  - A human programmer will usually be able to write more efficient assembly language than a high-level language compiler can generate.
  - In large programs, the slight inefficiency of the compiler is well worth the trade-off for ease of programming in a high-level language.
  - However, time-critical routines such as I/O drivers or ISRs may benefit from manual assembly language coding.

PIC16 Addressing Modes

- Data Memory Access:
  - Direct `addwf <data_addr>, <d>`
  - Indirect `addwf INDF, <d>`
  - Immediate (Literal) `movlw <constant>`
- Program Memory Access:
  - Absolute `goto <program_addr>`
  - Relative `addwf PCL, f`

PIC

Register Direct Addressing



2-bits from STATUS Register
7-bits Encoded in Instruction
9-bit Effective Address (Use this when coding)
0
0
0
0
0
0
0
0
0
RP1
RP0
'f' Operand
0x183
Bank 0
Bank 1
Bank 2
Bank 3
Address
00h
01h
02h
03h
04h
05h
7Ah
7Bh
7Ch
7Dh
7Eh
7Fh
Register File Address Bus

Register Direct Addressing

Example: Initialize bits 0-3 of PORTB as outputs

W Register:

F0

9-Bit Effective Address:

RP1	RP0	7-bits from Instruction						
0	1	0	0	0	0	0	0	0

```
bsf     STATUS,RP0
movlw   b'11110000'
movwf   TRISB
bcf     STATUS,RP0
clrf    PORTB
```

PIC

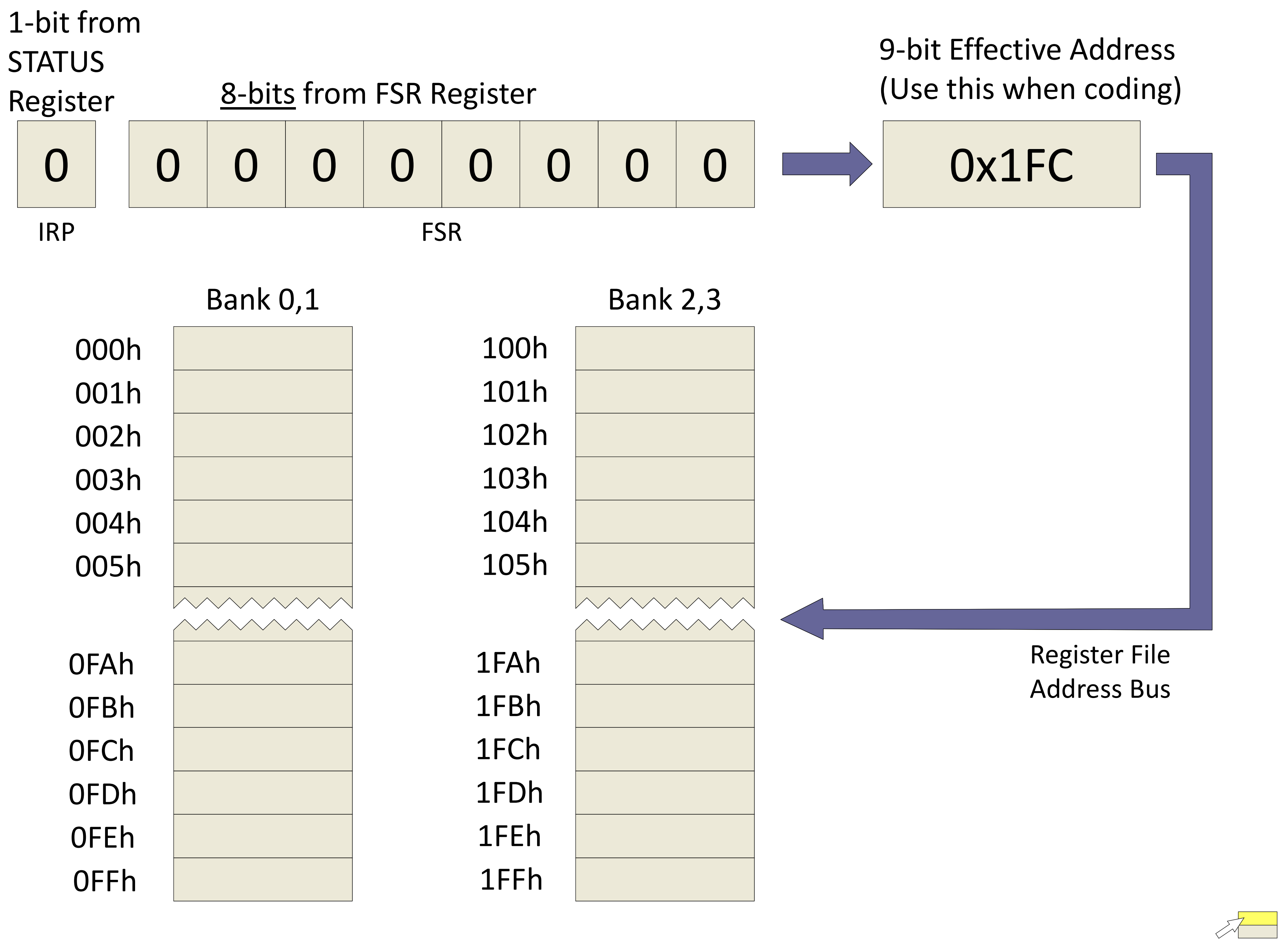
Register File

Address	Bank 0	Bank 1	Address
INDF: 00h	FF	FF	80h : INDF
TMR0: 01h	FF	FF	81h : OPTION
PCL : 02h	FF	FF	82h : PCL
STATUS: 03h	38	38	83h : STATUS
FSR: 04h	FF	FF	84h : FSR
PORTA: 05h	FF	FF	85h : TRISA
PORTB: 06h	FF	FF	86h : TRISB
PORTC: 07h	FF	FF	87h : TRISC
20h	FF	FF	A0h
21h	FF	FF	A1h
22h	FF	FF	A2h
23h	FF	FF	A3h

Bin [✓ Hex] Dec

PIC

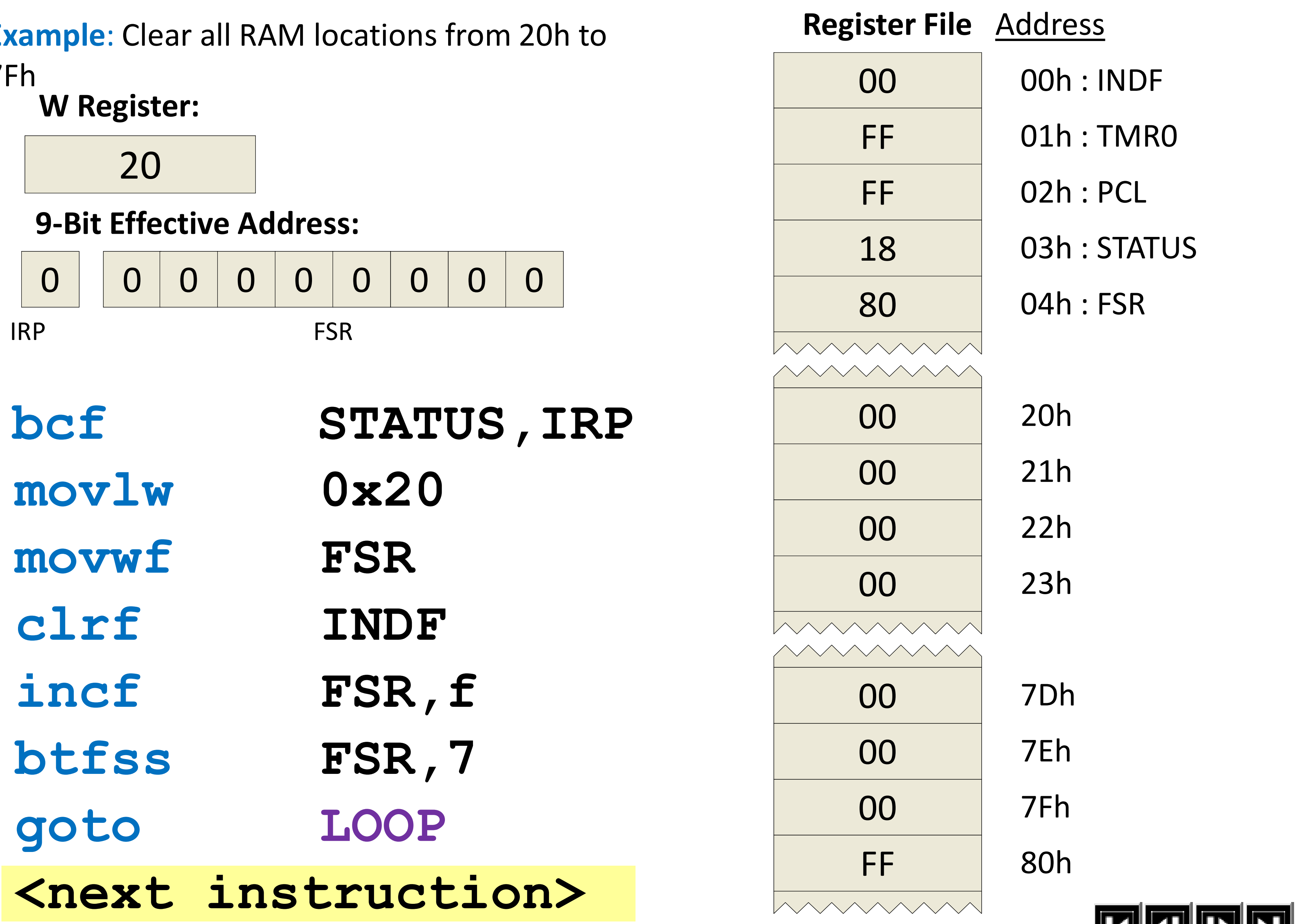
Register Indirect Addressing



PIC

Register Indirect Addressing

Example: Clear all RAM locations from 20h to 7Fh



Register File	Address
00	00h : INDF
FF	01h : TMR0
FF	02h : PCL
18	03h : STATUS
80	04h : FSR
00	20h
00	21h
00	22h
00	23h
00	7Dh
00	7Eh
00	7Fh
FF	80h

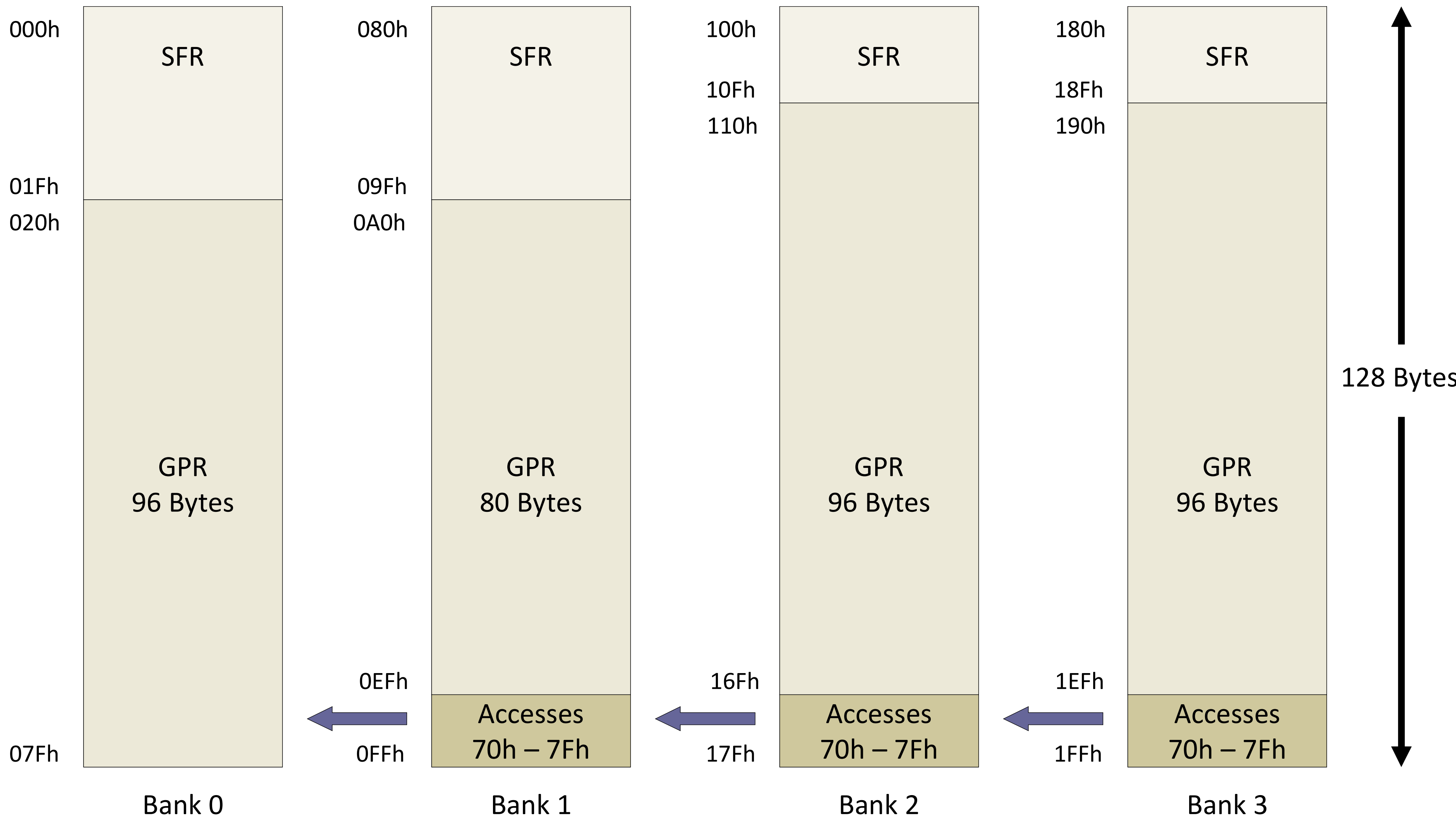
```
        bcf       STATUS,IRP
        movlw     0x20
        movwf     FSR
LOOP    clrf      INDF
        incf      FSR,f
        btfss     FSR,7
        goto      LOOP
        <next instruction>
```

PIC

Data Memory Organization

PIC16F876/877 Register File Map

368 Bytes of General Purpose RAM Plus Special Function Registers



PIC

Data Memory Organization

	Bank 0		Bank 1		Bank 2		Bank 3
000	INDF	080	INDF	100	INDF	180	INDF
001	TMR0	081	OPTION_REG	101	TMR0	181	OPTION_REG
002	PCL	082	PCL	102	PCL	182	PCL
003	STATUS	083	STATUS	103	STATUS	183	STATUS
004	FSR	084	FSR	104	FSR	184	FSR
005	PORTA	085	TRISA	105		185	
006	PORTB	086	TRISB	106	PORTB	186	TRISB
007	PORTC	087	TRISC	107		187	
008	PORTD	088	TRISD	108		188	
009	PORTE	089	TRISE	109		189	
00A	PCLATH	08A	PCLATH	10A	PCLATH	18A	PCLATH
00B	INTCON	08B	INTCON	10B	INTCON	18B	INTCON
00C	PIR1	08C	PIE1	10C	EEDATA	18C	EECON1
00D	PIR2	08D	PIE2	10D	EEADR	18D	EECON2

Device Specific Registers

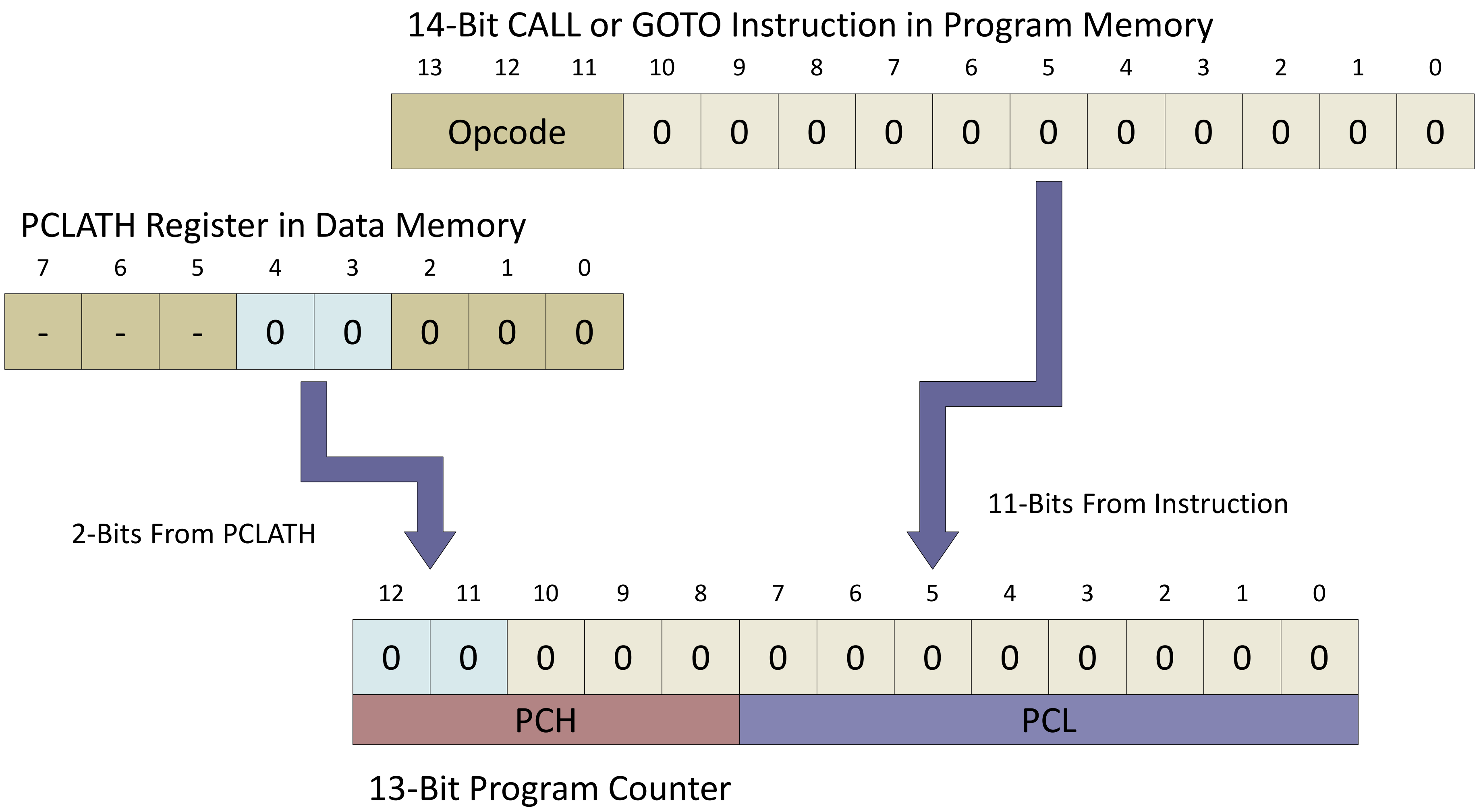
PC Absolute Addressing

CALL and GOTO Instructions:

13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode			0	0	0	0	0	0	0	0	0	0	0

- PC Absolute Addressing (Program Memory)
  - Jump to another program memory location out of PC sequence
  - Call a subroutine
- Used by the CALL and GOTO instructions
  - 11-bits of the required 13 address bits are encoded in the instruction
  - 2 additional bits will come from the PCLATH register
- Used when performing Computed Goto operation
  - Address to jump to is calculated by the program
  - Computed address is written directly into the Program Counter

PC Absolute Addressing



PC Absolute Addressing

Example: Jumping to code located in a different program memory page.

PCLATH Register

7	6	5	4	3	2	1	0
-	-	-	0	0	0	0	0

CALL Instruction in Program Memory

13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode			0	0	0	0	0	0	0	0	0	0	0

W Register

FF

Program Counter - PCH:PCL

-	0	0	0	0	0	0	0	0	0	0	0	0	0

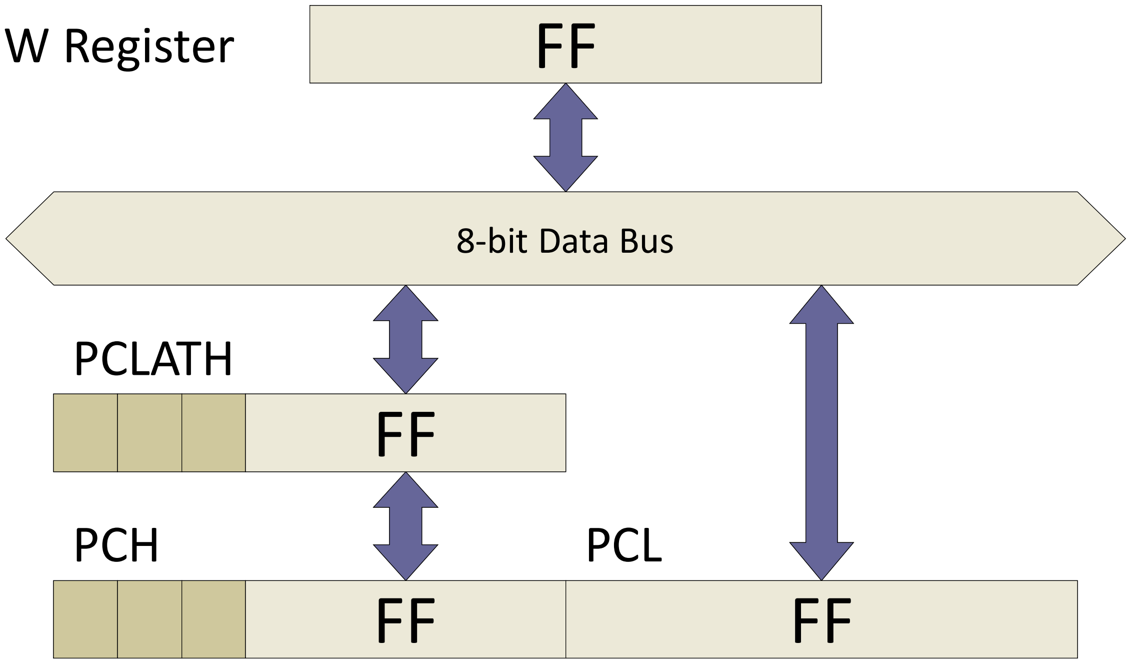
```
                org     0x0020
                movlw   HIGH MySubroutine
                movwf   PCLATH
                call    MySubroutine
                …
                org     0x1250
MySubroutine    <do something useful>
                …
                return
```

PC Relative Addressing

To write to PC:

❶ Write high byte to PCLATH

❷ Write low byte to PCL
(PCH will be loaded with value from PCLATH)



```
movlw   HIGH 0x1250
movwf   PCLATH
movlw   LOW 0x1250
movwf   PCL
```

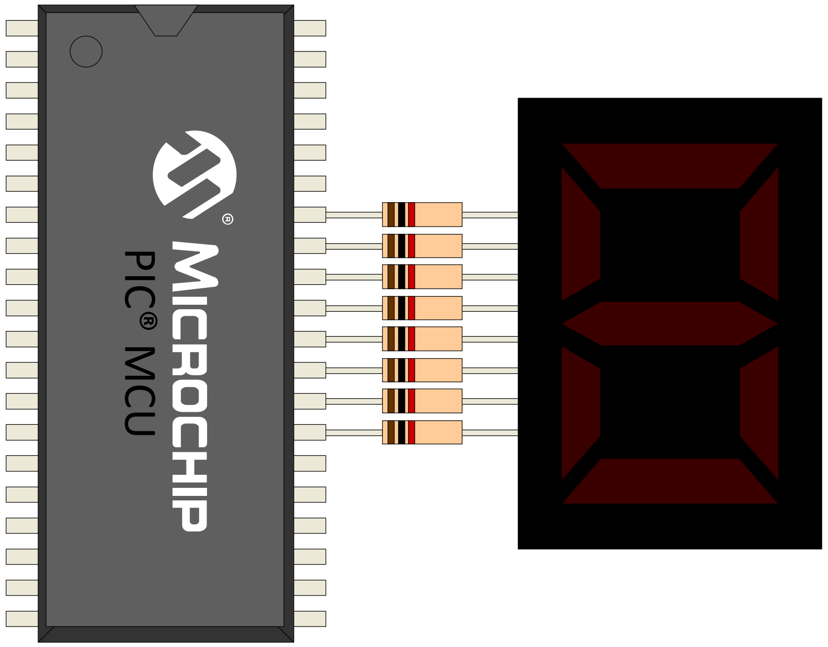
PC Relative Addressing: Lookup Table

Example: Use a lookup table with relative addressing to retrieve the bit pattern to display a digit on a 7-segment LED

```
        ORG     0x0020              ;Page 0
        movlw   HIGH SevenSegDecode
        movwf   PCLATH
        movlw   .5
        call    SevenSegDecode
        movwf   PORTB

        …

        ORG     0x1800              ;Page 3
SevenSegDecode:
        addwf   PCL,f
        retlw   b'00111111'    ;0
        retlw   b'00000110'    ;1
        retlw   b'01011011'    ;2
        retlw   b'01001111'    ;3
        retlw   b'01100110'    ;4
        retlw   b'01101101'    ;5
        retlw   b'01111101'    ;6
        retlw   b'00000111'    ;7
        retlw   b'01111111'    ;8
        retlw   b'01101111'    ;9
```



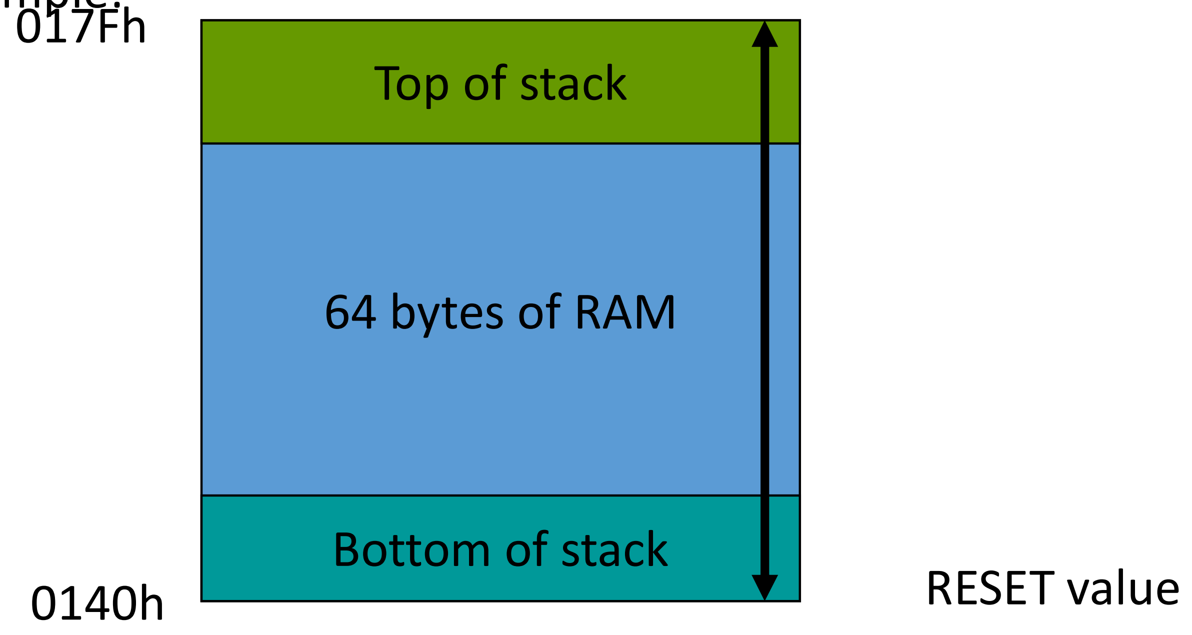
The Stack

- a dynamic RAM area
  - return addresses are **pushed** when you call a routine or an interrupt handler is invoked,
  - off which they are **popped** when these routines finish
- The stacks are
  - hardware and
  - software

The Stack

Stack: return address from subroutine are stored by a CALL instruction and retrieved by a RET instruction.

Example:



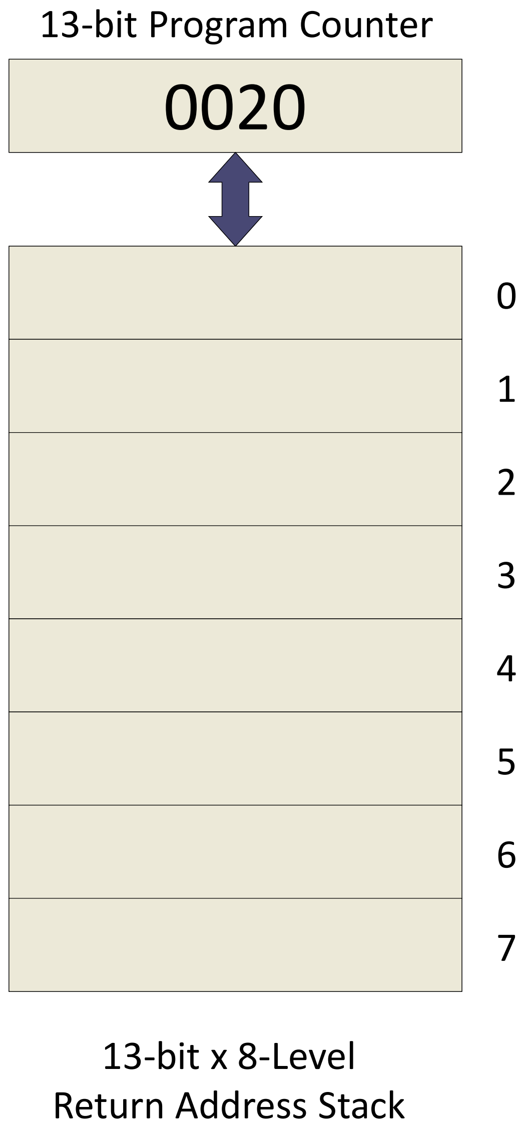
Managing the stack

- This is just a reminder, since it applies anywhere in the program. The service routine must track the usage it makes of the stack, so as to pop at the end as many bytes as it had pushed at the beginning.
- This may look trivial, but if some pushes occur in some conditions and not others (i.e. the service routine has conditional statements somewhere), the popping must occur in exactly the reverse way, taking into account the same conditions as those that produced the pushing.
- This may not be very obvious to code.

PIC

CALL / RETURN Stack

```
0020            movlw   HIGH MySub1
0021            movwf   PCLATH
0022            call    MySub1
0023            call    MySub4
0024            bsf     PORTB,7
 ...            ...
1000    MySub1  bsf     PORTB,0
1001            call    MySub2
1002            return
1003    MySub2  bsf     PORTB,1
1004            call    MySub3
1005            return
1006    MySub3  bsf     PORTB,2
1007            return
1008    MySub4  bsf     PORTB,3
1009            call    MySub2
100A            return
```



Vectors

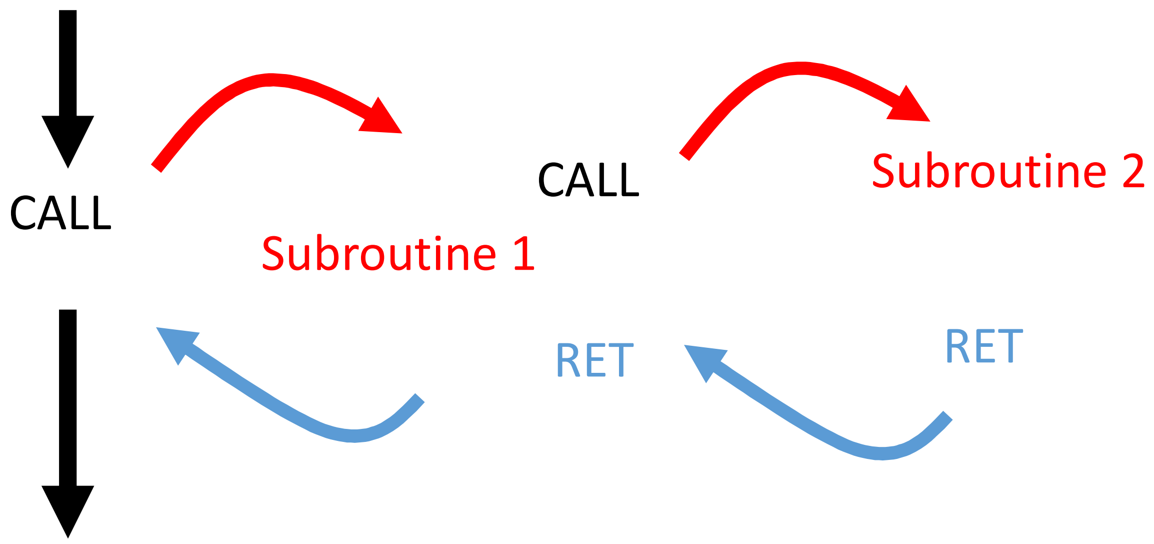
- a special program (code) location
- used after
  - Reset
  - Interrupt
- May appear like
  - a JUMP instruction
  - an address (an entry point)
- Typically all the vectors are together in one area, called the *vector table*

Subroutines

Subroutine:

Is a part of program that is used frequently and make sense to separate from rest of the program and call when is necessary.

Example of subroutines



Interrupts – The Cat Story

1. Watching the TV. The doorbell rings.
2. Treating: talking with the postman. The phone rings. Interrupting the talk.
3. Treating: talking with the mother in law. Ending the call. Coming back to the door.
4. Ending the postman story. Back to the TV! Promice not to be interrupted any longer!
5. The cat, the milk bottle, shortcircuit saga. Changing the fuse.
6. Watching the TV.

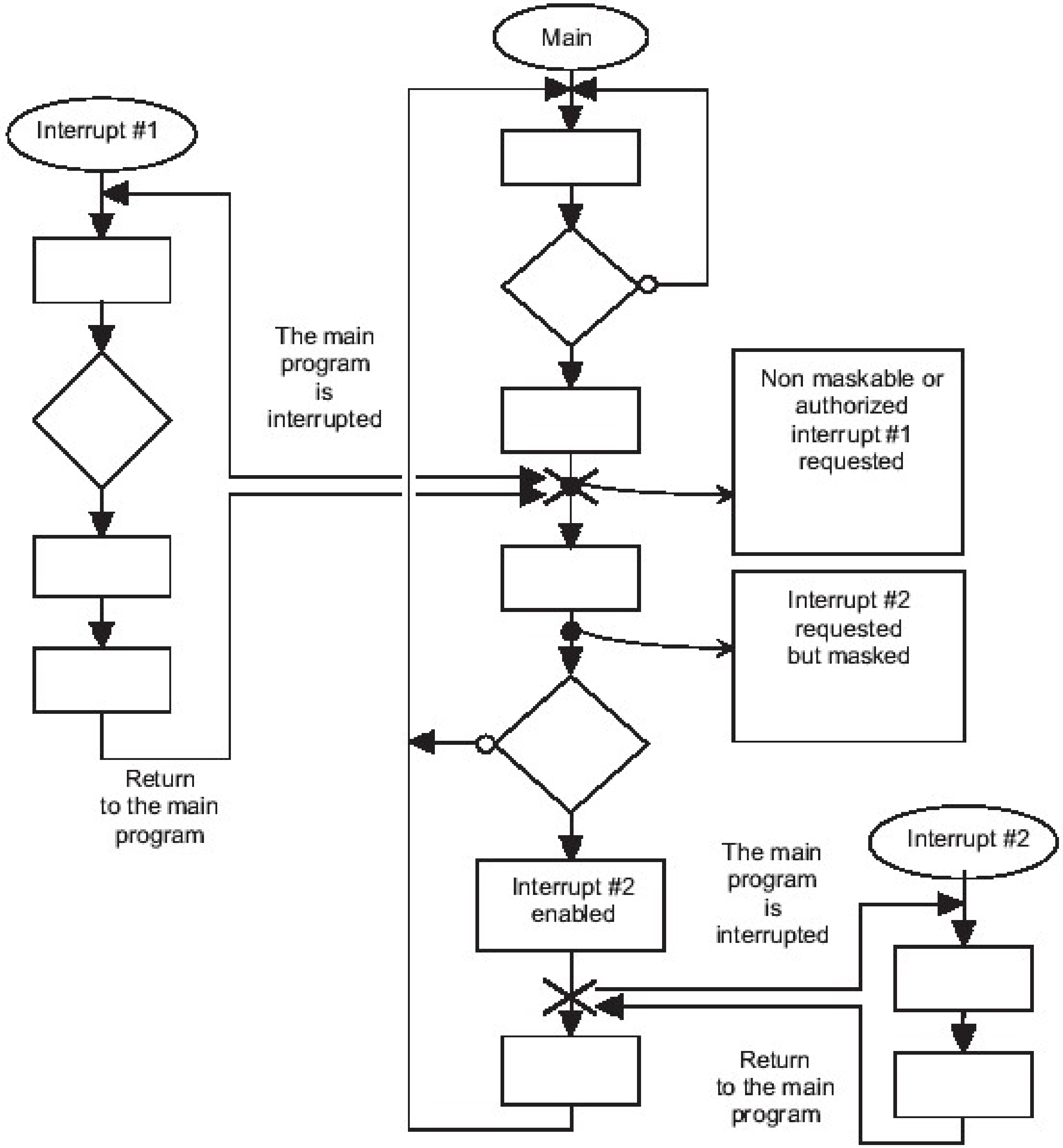
Interrupts - continued

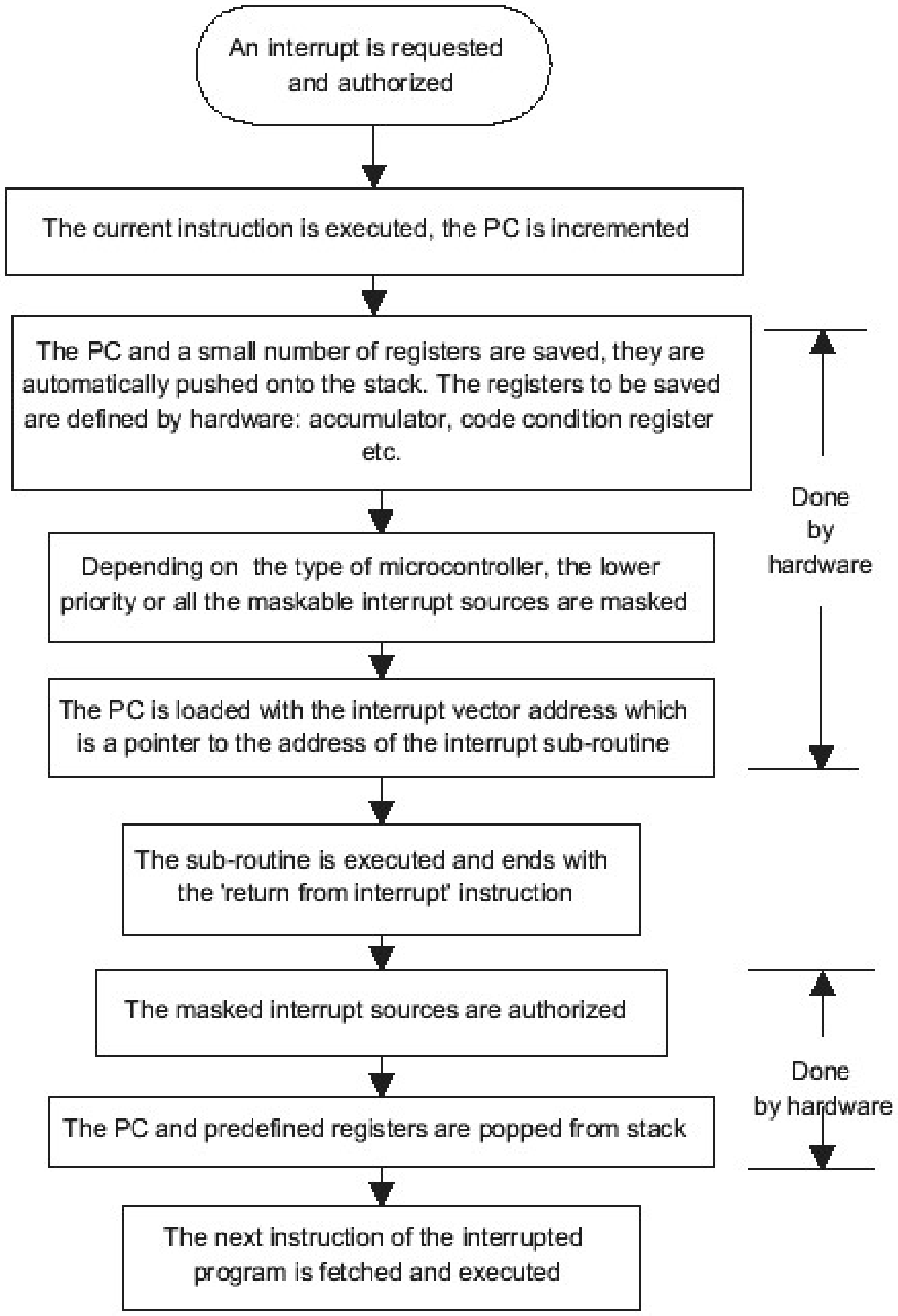
- Interrupts are the most common means of altering the normal course of the program, when occurs
  - an unexpected event
  - an expected one but occurring at an unexpected time.
- It may be more or less complicated according to the features it provides.
- Features provided may include
  - queuing of interrupt requests
  - handling requests according to their priorities,
  - or even modification of priorities to increase the chance that low-priority requests will be eventually processed in a context where there are numerous requests.

- At the point an IRQ of a master processor receives a signal that an interrupt has been raised, the interrupt is processed by the interrupt handling mechanisms within the system.
- These mechanisms are made up of a combination of both hardware and software components. In terms of hardware, an ***interrupt controller can be integrated onto a board or within a processor*** to mediate interrupt transactions in conjunction with software.
- Architectures that include an interrupt controller within their interrupt handling schemes include
  - the 268/386 (x86) architectures that use two PICs (Intel’s Programmable Interrupt Controller);
  - MIPS32, which relies on an external interrupt controller;
  - and the MPC860, which integrates two interrupt controllers, one in the CPM and one in its SIU.
- For systems with no interrupt controller (such as the Mitsubishi M37267M8 TV microcontroller), the interrupt request lines are connected directly to the master processor, and interrupt transactions are controlled via software and some internal circuitry (registers, counters, etc.).

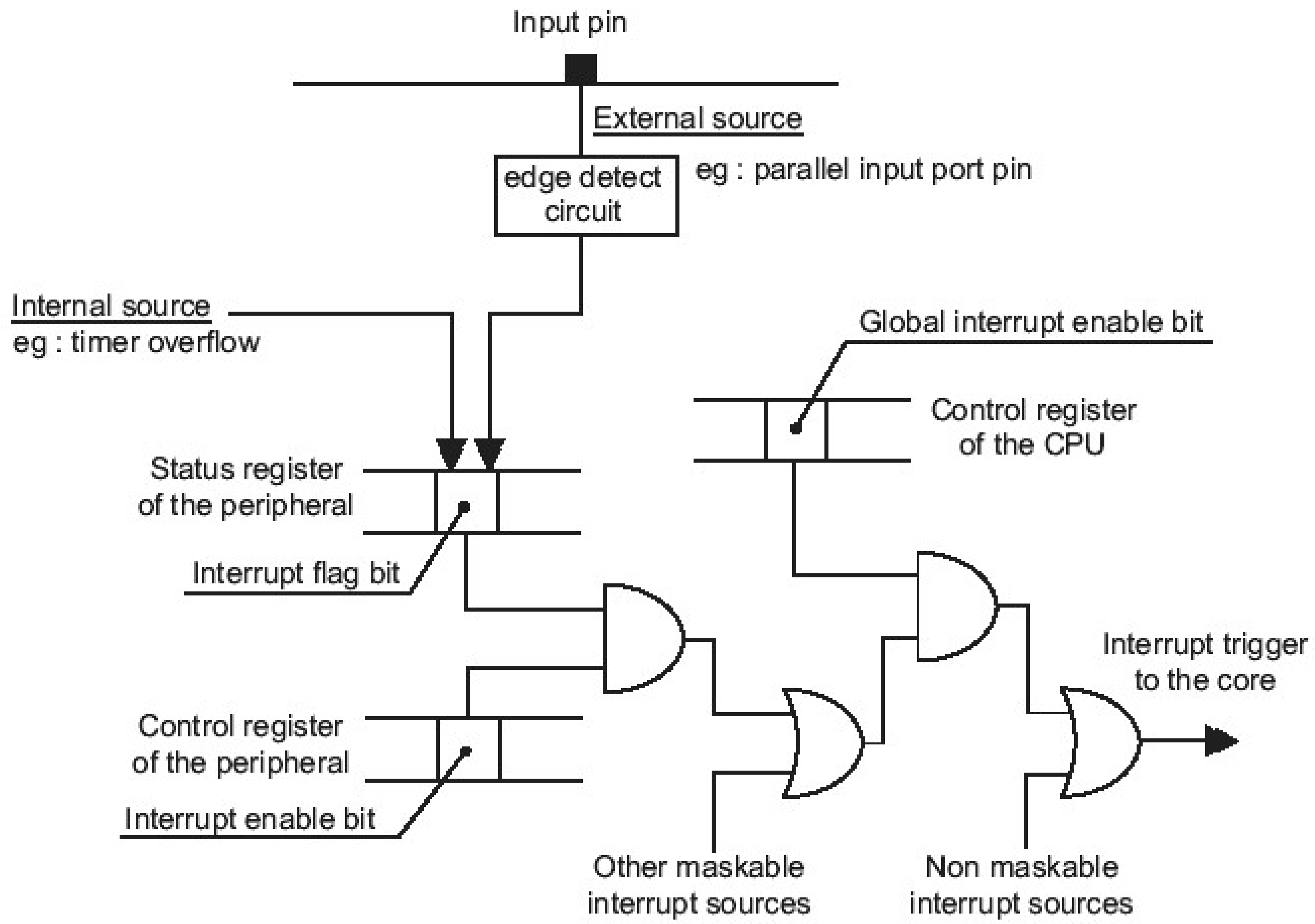
Interrupts

- With respect to the external device triggering an interrupt, the interrupt scheme depends on whether that device can provide an ***interrupt vector (a place in memory that holds the address of*** an interrupt’s ISR).
- For devices that cannot provide the interrupt vector, master processors implement an ***auto-vectored interrupt scheme and acknowledgment is done via software.***


Main
Interrupt #1
The main program is interrupted
Non maskable or authorized interrupt #1 requested
Interrupt #2 requested but masked
Return to the main program
Interrupt #2
Interrupt #2 enabled
The main program is interrupted
Return to the main program


An interrupt is requested and authorized
The current instruction is executed, the PC is incremented
The PC and a small number of registers are saved, they are automatically pushed onto the stack. The registers to be saved are defined by hardware: accumulator, code condition register etc.
Depending on the type of microcontroller, the lower priority or all the maskable interrupt sources are masked
The PC is loaded with the interrupt vector address which is a pointer to the address of the interrupt sub-routine
Done by hardware
The sub-routine is executed and ends with the 'return from interrupt' instruction
The masked interrupt sources are authorized
The PC and predefined registers are popped from stack
Done by hardware
The next instruction of the interrupted program is fetched and executed

The hardware behind the interrupt mechanism


Input pin
External source
edge detect circuit
eg : parallel input port pin
Internal source
eg : timer overflow
Global interrupt enable bit
Control register of the CPU
Status register of the peripheral
Interrupt flag bit
Interrupt trigger to the core
Control register of the peripheral
Interrupt enable bit
Other maskable interrupt sources
Non maskable interrupt sources

Interrupt Vector

- An interrupt vectored scheme is implemented to support peripherals that can provide an interrupt vector over a bus, and acknowledgment is automatic.
  - Some IACK register on the master CPU informs the device, requesting the interrupt to stop requesting interrupt service, and provides the master processor with what it needs to process the correct interrupt (such as the interrupt number, vector number, and so forth).
  - Based upon the activation of an external interrupt pin, an interrupt controller's interrupt select register, a device's interrupt select register, or some combination of these, the master processor can determine which ISR to execute.
  - After the ISR completes, the master processor resets the interrupt status by adjusting the bits in the processor's status register or an interrupt mask in the external interrupt controller.
  - The interrupt request and acknowledge mechanisms are determined by the device requesting the interrupt (since it determines which interrupt service to trigger), the master processor, and the system bus protocols.

Auto-vectored scheme

- PowerPC architectures implement an auto-vectored scheme, with no interrupt vector base register.
- The 68000 architecture supports both auto-vectored and interrupt vectored schemes, whereas MIPS32 architectures have no IACK cycle, and so the interrupt handler handles the triggered interrupts.

Interrupt Priorities

- All available interrupts within a processor have an associated interrupt level, which is the priority of that interrupt within the system.
  - Typically, interrupts starting at level “1” are the highest priority within the system, and incrementally from there (2,3,4,…) the priorities of the associated interrupts decrease.
  - Interrupts with higher levels (priorities) have precedence over any instruction stream being executed by the master processor.
  - This means that not only do interrupts have precedence over the main program, but they also have precedence over interrupts with lower priorities as well.

Interrupts - Trigerring

- Interrupts are signals triggered by some event during the execution of an instruction stream by the master processor.
  - This means they can be initiated asynchronously, for external hardware devices, resets, power failures, or
  - synchronously for instruction-related activities such as system calls, or illegal instructions.
- These signals cause the master processor to stop executing the current instruction stream and start the process of handling (processing) the interrupt.
- The three main types of interrupts are
  - software,
  - internal hardware, and
  - external hardware.

Main Types Of Interrupts –

- The three main types of interrupts are
  - software,
  - internal hardware
  - external hardware.

Software Interrupts

- Software interrupts are explicitly triggered internally by some instruction within the current instruction stream being executed by the master processor.

Internal Hardware Interrupts

- Internal hardware interrupts, on the other hand, are initiated by an event that is a result of a problem with the current instruction stream that is being executed by the master processor because of the features (or limitations) of the hardware, such as
  - illegal math operations like overflow or divide-by-zero,
  - debugging (single-stepping, breakpoints),
  - invalid instructions (opcodes),
  - and so on.
- Interrupts that are raised (requested) by some internal event to the master processor (basically, software and internal hardware interrupts) are also commonly referred to as exceptions or traps (depending on the type of interrupt).

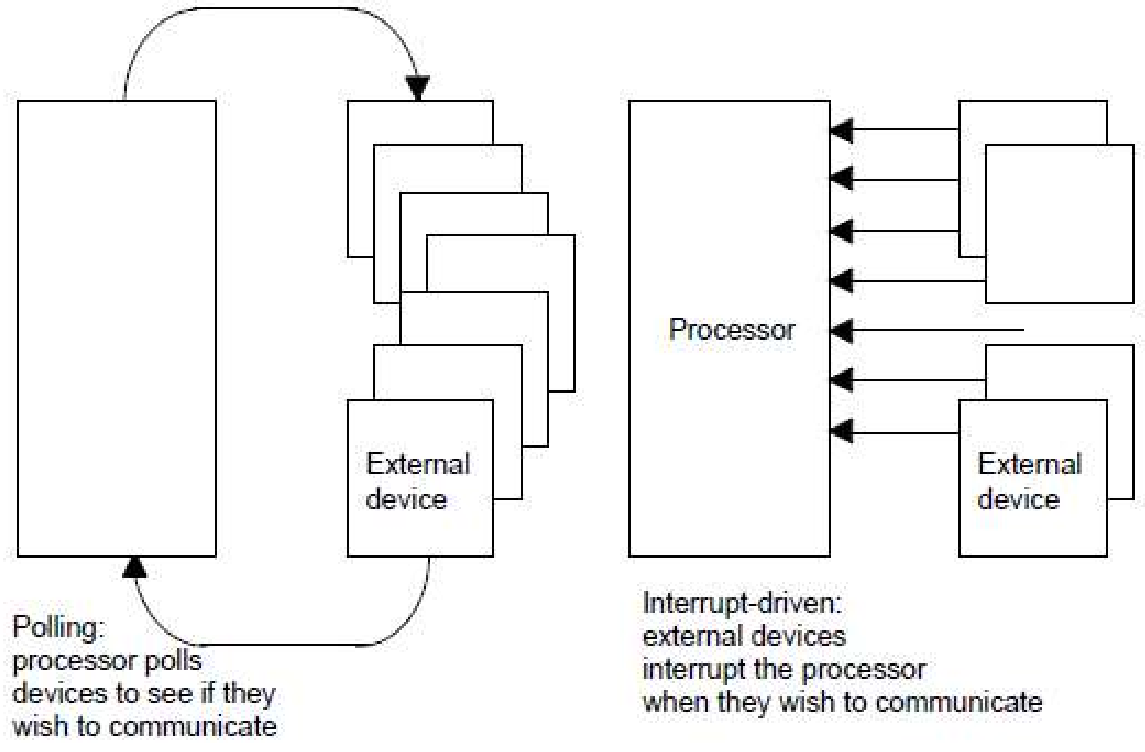
Hardware interrupts

- Computer systems either use
  - **polling** or
  - **interrupt-driven**

software to service external equipment.

- With polling the computer continually monitors a status line and waits for it to become active, whereas an interrupt-driven device sends an interrupt request to the computer, which is then serviced by an interrupt service routine (ISR).
- Interrupt-driven devices are normally better in that the computer is thus free to do other things, whereas
- polling slows the system down as it must continually monitor the external device.
  - Polling can also cause problems in that a device may be ready to send data and the computer is not watching the status line at that point.

Polling or interrupt-driven communications



Polling or interrupt-driven communications

interrupts

- As illustrated in the next figure, the generation of an interrupt can occur by
  - hardware or
  - software,
- If a device wishes to interrupt the processor, it informs the programmable interrupt controller (PIC).
- The PIC then decides whether it should interrupt the processor.
  - If there is a processor interrupt then the processor reads the PIC to determine which device caused the interrupt.
  - Then, depending on the device that caused the interrupt, a call to an ISR is made.

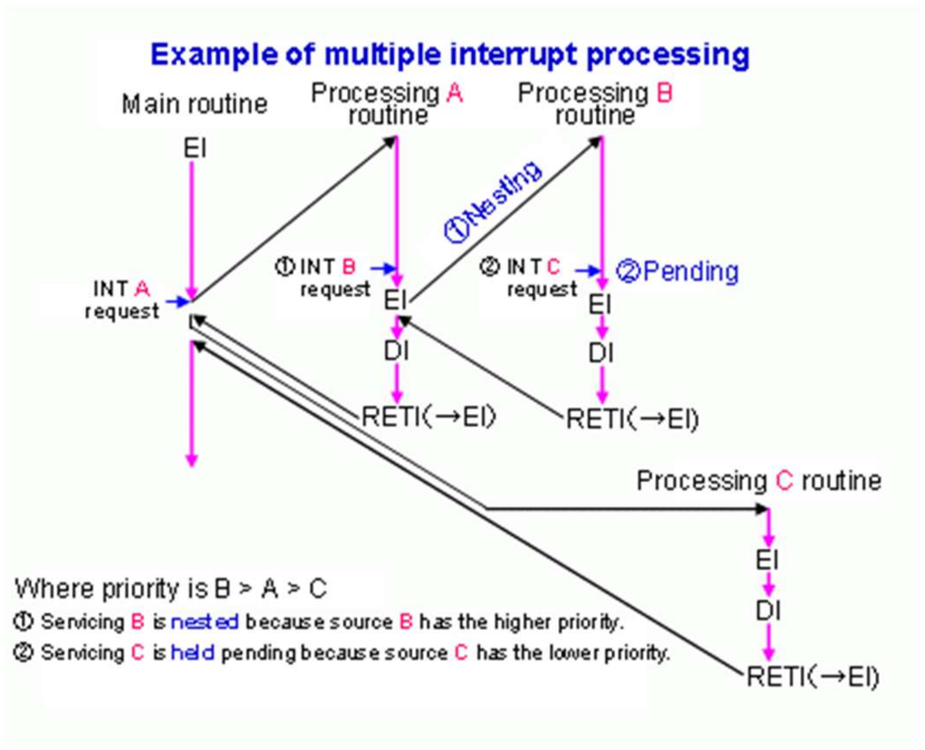
Software and Hardware interrupts

- The ISR then communicates with the device and processes any data. When it has finished the program execution returns to the original program.
- A software interrupt causes the program to interrupt its execution and goes to an interrupt service routine.
  - Typical software interrupts include reading a key from the keyboard, outputting text to the screen and reading the current date and time.
- Hardware interrupts allow external devices to gain the attention of the processor.

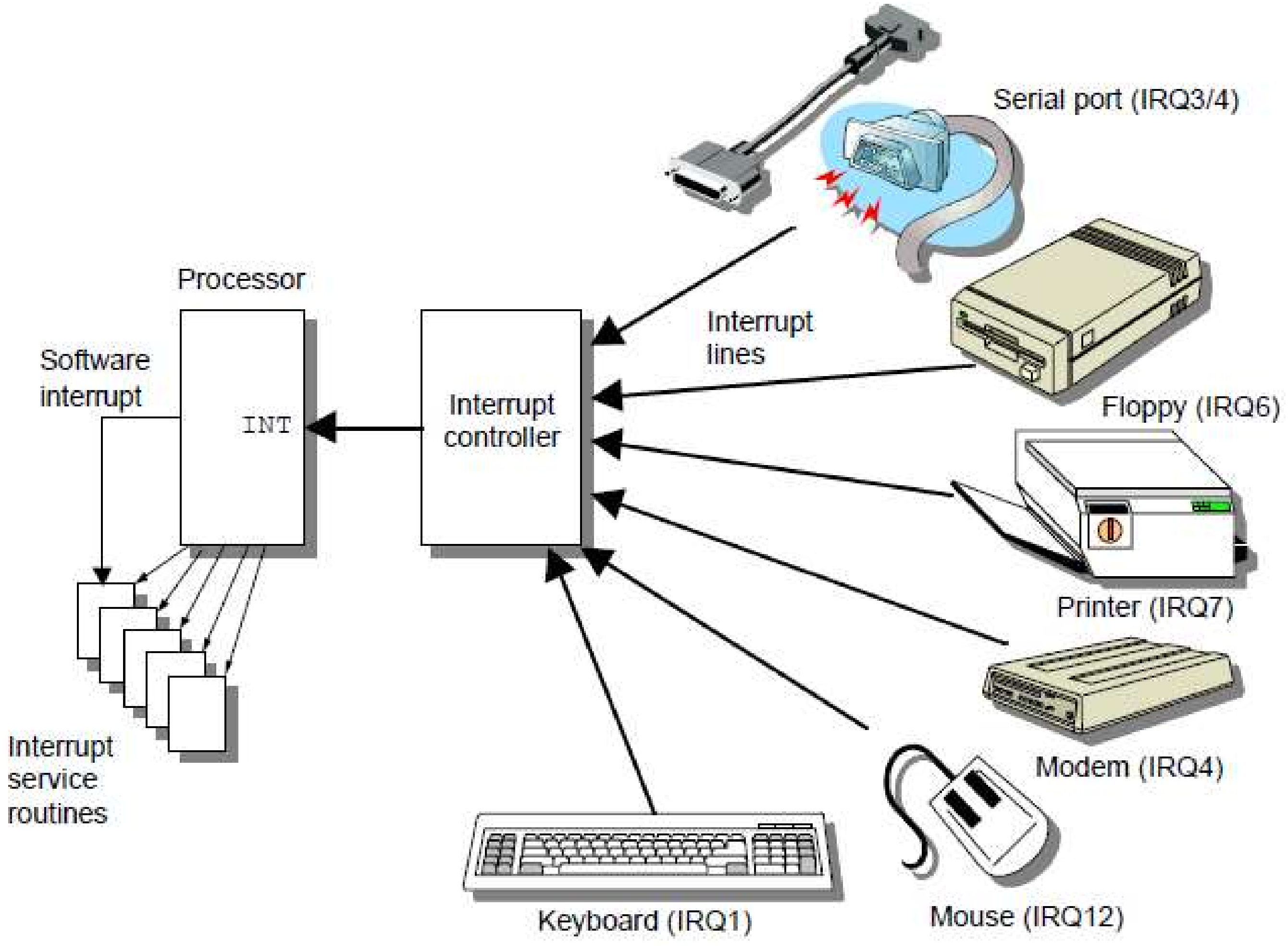
Interrupt service routine (ISR)

- Depending on the type of interrupt the processor leaves the current program and goes to a special program called an interrupt service routine (ISR).
  - This program communicates with the device and processes any data.
  - After it has completed its task then program execution returns to the program that was running before the interrupt occurred.
  - Examples of interrupts include the processing of keys from a keyboard and data from a sound card.
- As previously mentioned, a device informs the processor that it wants to interrupt it by setting an interrupt line on the PC.
- Then, depending on the device that caused the interrupt, a call to an ISR is made.
- Each PIC allows access to eight interrupt request lines.
  - Most PCs use two PICs which gives access to 16 interrupt lines.

Multiple Interrupt Processing


Example of multiple interrupt processing
Main routine
Processing A routine
Processing B routine
EI
INT A request
① INT B request
①Nesting
② INT C request
②Pending
EI
DI
RETI(→EI)
EI
DI
RETI(→EI)
Processing C routine
EI
DI
RETI(→EI)
Where priority is B > A > C
① Servicing B is nested because source B has the higher priority.
② Servicing C is held pending because source C has the lower priority.

Interrupt handling



Interrupt handling

External Hardware Interrupts

- external hardware interrupts are interrupts initiated by hardware other than the master CPU (i.e., board buses, I/O, etc.).
- What actually triggers an interrupt is typically determined by the software via register bits that activate or deactivate potential interrupt sources in the initialization device driver code.

External Events Interrupts

- For interrupts that are raised by external events, the master processor is either wired
  - via an input pin(s), called an IRQ (Interrupt Request Level) pin or port, to outside intermediary hardware (i.e., interrupt controllers),
  - or directly to other components on the board with dedicated interrupt ports that signal the master CPU when they want to raise the interrupt.
- These types of interrupts are triggered in one of two ways:
  - level-triggered or
  - edge-triggered.

Level-triggered Interrupts

- A level-triggered interrupt is initiated when the interrupt request (IRQ) signal is at a certain level (i.e., HIGH or LOW—see Figure 30).
- These interrupts are processed when the CPU finds a request for a level-triggered interrupt when sampling its IRQ line, such as at the end of processing each instruction.

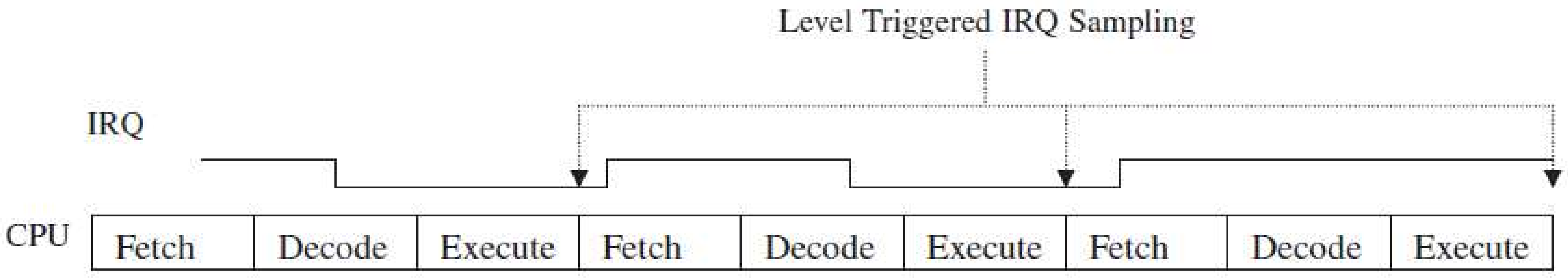


Figure 30: Level-triggered interrupts

Edge-triggered Interrupts

- Edge-triggered interrupts trigger when a change occurs on its IRQ line (from LOW to HIGH/rising edge of signal or from HIGH to LOW/falling edge of signal—see Figure 31).
- Once triggered, these interrupts latch into the CPU until processed.

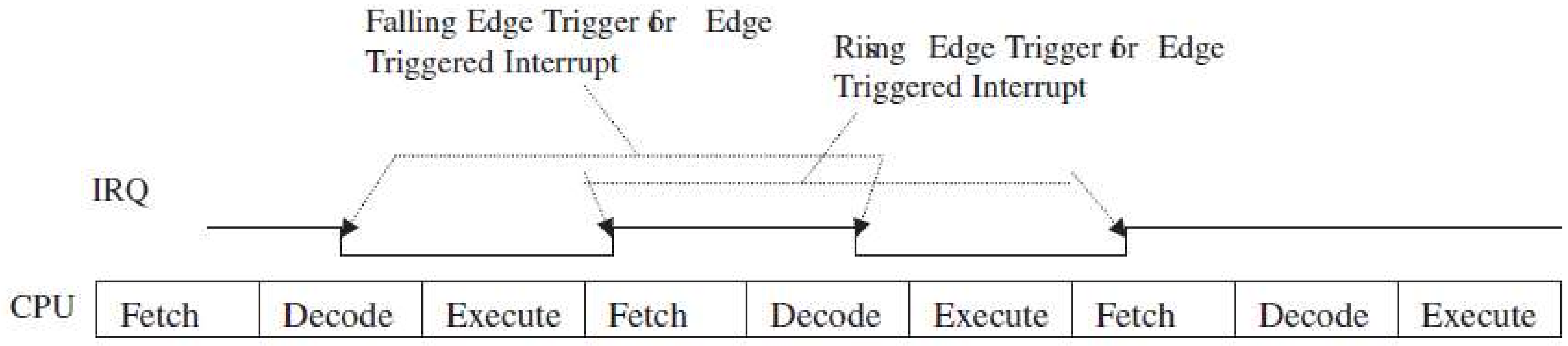

Falling Edge Trigger for Edge Triggered Interrupt
Rising Edge Trigger for Edge Triggered Interrupt
IRQ
CPU
Fetch
Decode
Execute
Fetch
Decode
Execute
Fetch
Decode
Execute

Figure 31: Edge-triggered interrupts

Interrupts - Strengths And Drawbacks

- With a level-triggered interrupt, as shown in Figure 32, if the request is being processed and has not been disabled before the next sampling period, the CPU would try to service the same interrupt again.
- On the flip side, if the level-triggered interrupt were triggered and then disabled before the CPU's sample period, the CPU would never note its existence and would therefore never process it.

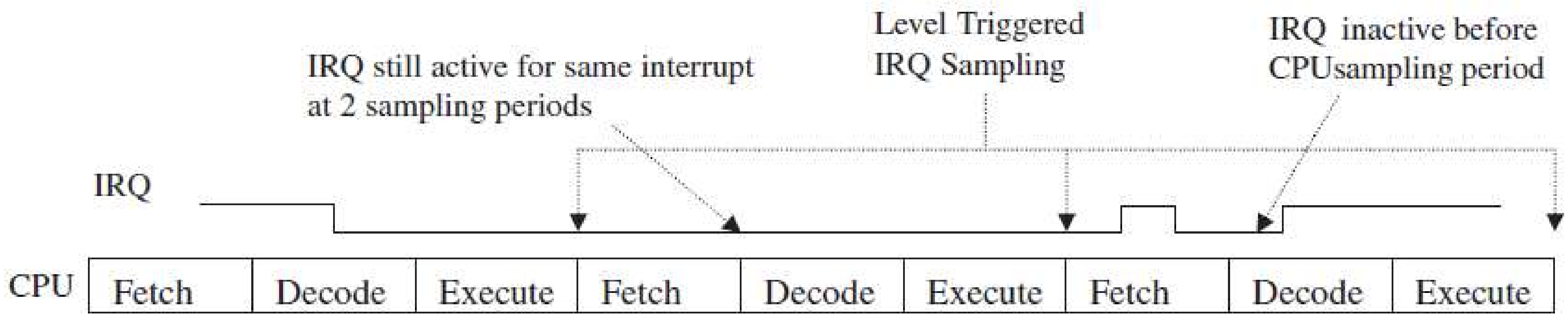

IRQ still active for same interrupt at 2 sampling periods
Level Triggered IRQ Sampling
IRQ inactive before CPUsampling period
IRQ
CPU
Fetch
Decode
Execute
Fetch
Decode
Execute
Fetch
Decode
Execute

Figure 32: Level-triggered interrupts drawbacks

Interrupts - Strengths And Drawbacks

- Edge level interrupts can have problems if they share the same IRQ line, if they are triggered in the same manner at about the same time (say before the CPU could process the first interrupt), resulting in the CPU being able to detect only one of the interrupts (see Figure 33).

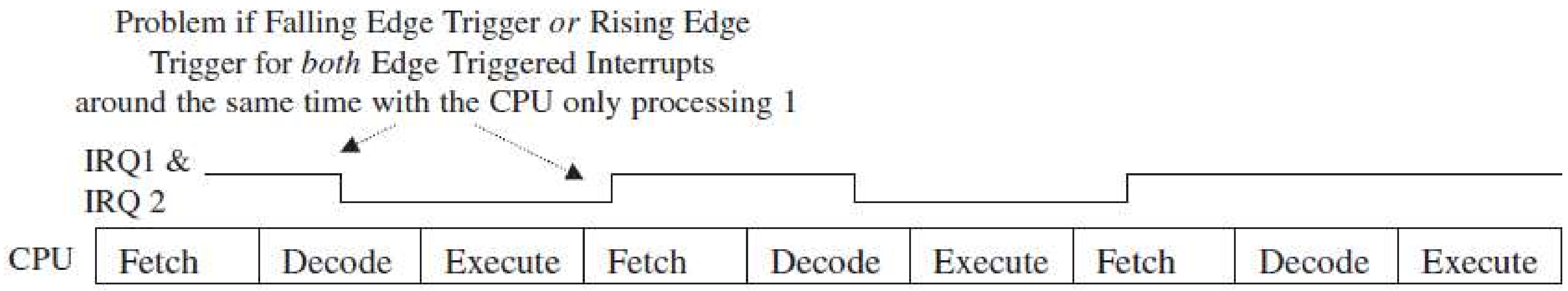

Problem if Falling Edge Trigger *or* Rising Edge Trigger for *both* Edge Triggered Interrupts around the same time with the CPU only processing 1
IRQ1 & IRQ 2
CPU
Fetch
Decode
Execute
Fetch
Decode
Execute
Fetch
Decode
Execute

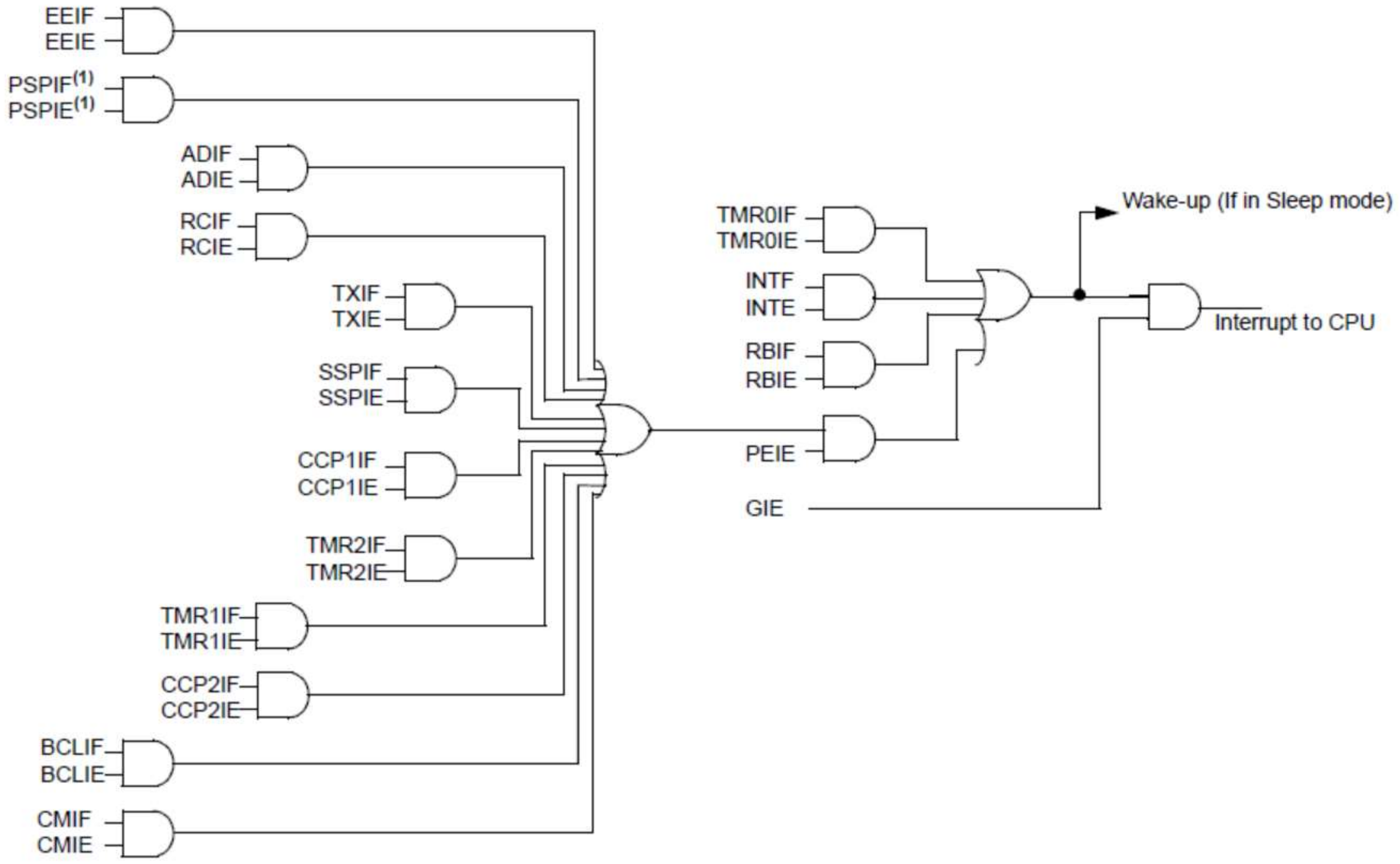
Figure 33: Edge-triggered interrupts drawbacks

Interrupts - Conclusions

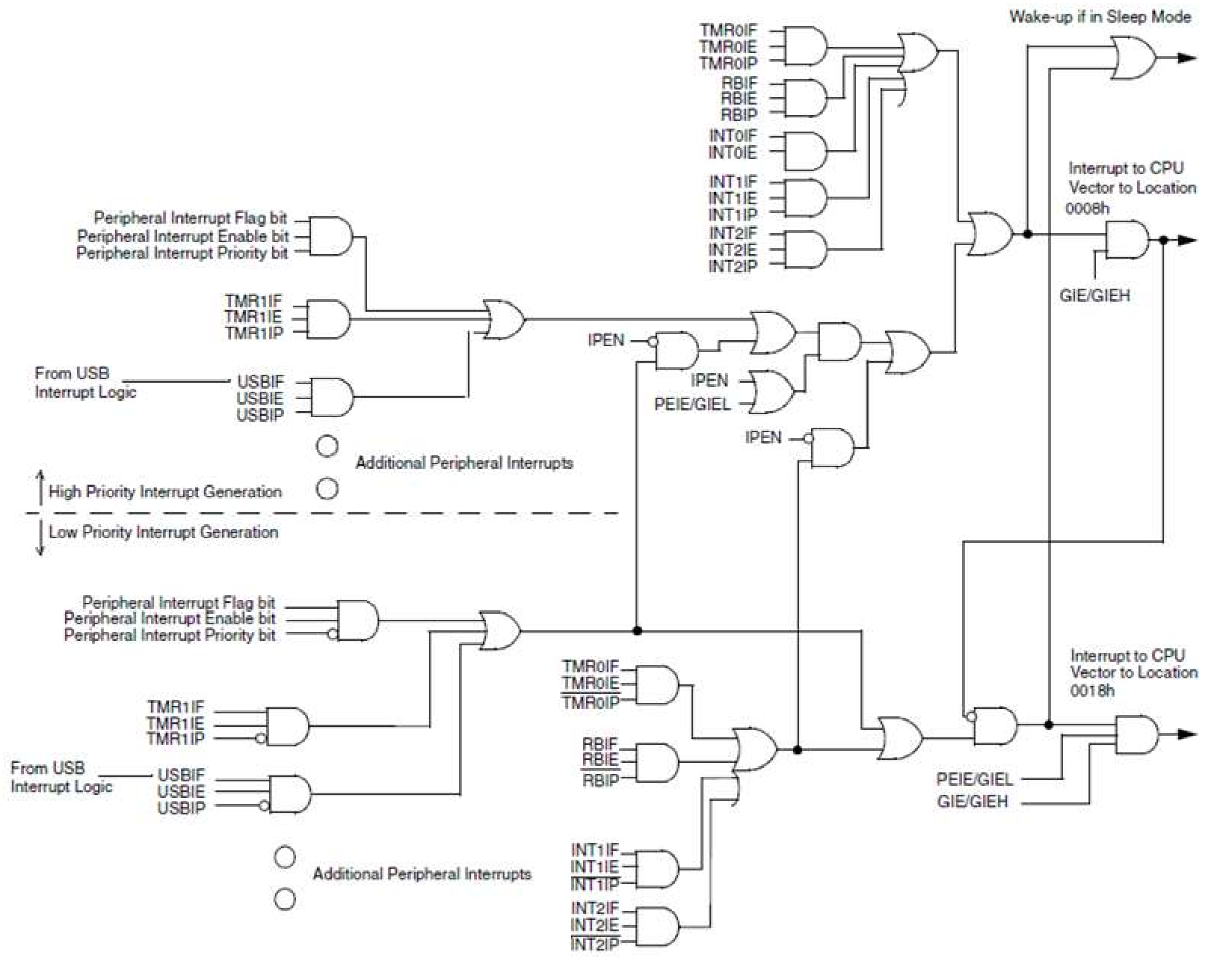
- Because of these drawbacks
  - level-triggered interrupts are generally recommended for interrupts that share IRQ lines,
  - edge-triggered interrupts are typically recommended for interrupt signals that are very short or very long.

Interrupt Details (for Microcontrolles)

PIC16F877A Interrupt Logic



PIC18F Interrupt Logic



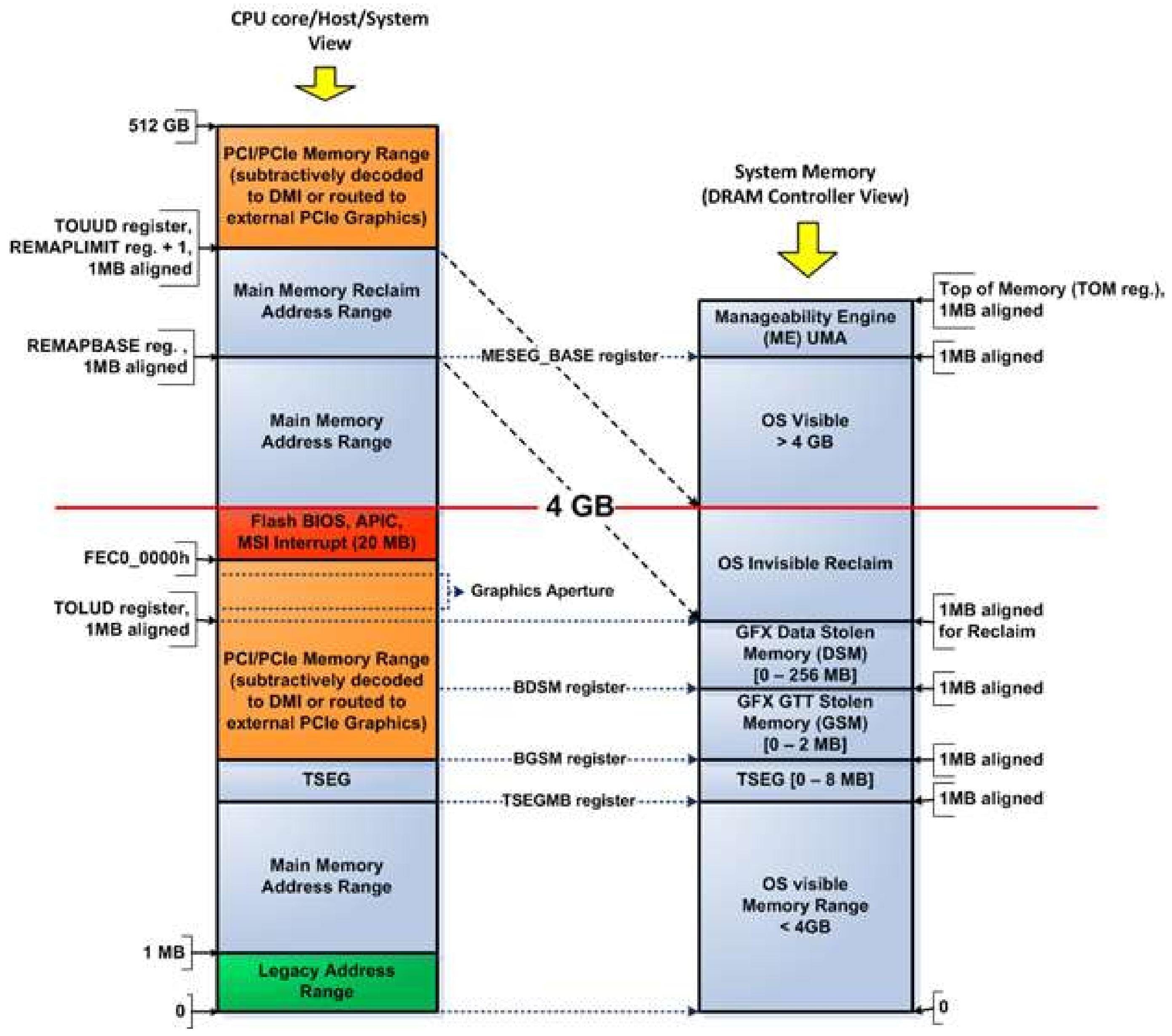
Wake-up if in Sleep Mode
TMR0IF
TMR0IE
TMR0IP
RBIF
RBIE
RBIP
INT0IF
INT0IE
INT1IF
INT1IE
INT1IP
INT2IF
INT2IE
INT2IP
Interrupt to CPU Vector to Location 0008h
GIE/GIEH
Peripheral Interrupt Flag bit
Peripheral Interrupt Enable bit
Peripheral Interrupt Priority bit
TMR1IF
TMR1IE
TMR1IP
IPEN
IPEN
PEIE/GIEL
IPEN
From USB Interrupt Logic
USBIF
USBIE
USBIP
Additional Peripheral Interrupts
High Priority Interrupt Generation
Low Priority Interrupt Generation
Peripheral Interrupt Flag bit
Peripheral Interrupt Enable bit
Peripheral Interrupt Priority bit
TMR0IF
TMR0IE
TMR0IP
TMR1IF
TMR1IE
TMR1IP
RBIF
RBIE
RBIP
Interrupt to CPU Vector to Location 0018h
From USB Interrupt Logic
USBIF
USBIE
USBIP
PEIE/GIEL
GIE/GIEH
INT1IF
INT1IE
INT1IP
Additional Peripheral Interrupts
INT2IF
INT2IE
INT2IP

PC (8086 Based) Interrupts

BIOS and the operating system

- The Basic Input/Output System (BIOS) communicates directly with the hardware of the computer.
- It consists of a set of programs which interface with devices such as
  - keyboards, displays,
  - printers, serial ports and disk drives.
- These programs allow the user to write application programs that contain calls to these functions, without having to worry about controlling them or which type of equipment is being used.
  - Without BIOS, the computer system would simply consist of a bundle of wires and electronic devices.

Haswell System Address Map (System Memory >= 4GB)



CPU core/Host/System View
512 GB
PCI/PCIe Memory Range (subtractively decoded to DMI or routed to external PCIe Graphics)
TOUUD register, REMAPLIMIT reg. + 1, 1MB aligned
Main Memory Reclaim Address Range
REMAPBASE reg. , 1MB aligned
MESEG_BASE register
Main Memory Address Range
4 GB
Flash BIOS, APIC, MSI Interrupt (20 MB)
FEC0_0000h
Graphics Aperture
TOLUD register, 1MB aligned
PCI/PCIe Memory Range (subtractively decoded to DMI or routed to external PCIe Graphics)
BDSM register
BGSM register
TSEG
TSEGMB register
Main Memory Address Range
1 MB
Legacy Address Range
0
System Memory (DRAM Controller View)
Top of Memory (TOM reg.), 1MB aligned
Manageability Engine (ME) UMA
1MB aligned
OS Visible > 4 GB
OS Invisible Reclaim
1MB aligned for Reclaim
GFX Data Stolen Memory (DSM) [0 – 256 MB]
1MB aligned
GFX GTT Stolen Memory (GSM) [0 – 2 MB]
1MB aligned
TSEG [0 – 8 MB]
1MB aligned
OS visible Memory Range < 4GB
0

BIOS Components

- There are two main parts to BIOS.
- The first is the part permanently stored in a ROM (the ROM BIOS).
  - It is this part that starts the computer (or bootstap) and contains programs which communicate with resident devices.
- The second stage is loaded when the operating system is started. This part is non-permanent.
  - An operating system allows the user to access the hardware in an easy-to-use manner. It accepts commands from the keyboard and displays them to the monitor.
  - The Disk Operating System, or DOS, gained its name from its original purpose of providing a controller for the computer to access its disk drives.
  - The language of DOS consists of a set of commands which are entered directly by the user and are interpreted to perform file management tasks, program execution and system configuration. It makes calls to BIOS to execute these.
  - The main functions of DOS are to run programs, copy and remove files, create directories, move within a directory structure and to list files.
  - Microsoft Windows calls BIOS programs directly.

Interrupt vectors

- Interrupt vectors are addresses which inform the interrupt handler as to where to find the ISR.
- All interrupts are assigned a number from 0 to 255. The interrupt vectors associated with each interrupt number are stored in the lower 1024 bytes of PC memory.
- For example, interrupt 0 is stored from 0000:0000 to 0000:0003, interrupt 1 from 0000:0004 to 0000:0007, and so on.
- The first two bytes store the offset and the next two store the segment address.
- Each interrupt number is assigned a predetermined task, as outlined in the next Table.
- An interrupt can be generated either by
  - external hardware,
  - software, or
  - by the processor.
- Interrupts 0, 1, 3, 4, 6 and 7 are generated by the processor.
- Interrupts from 8 to 15 and interrupt 2 are generated by external hardware.
- These get the attention of the processor by activating a interrupt request (IRQ) line.
- The IRQ0 line connects to the system timer, the keyboard to IRQ1, and so on.
- Most other interrupts are generated by software.

Interrupt vectors

- An interrupt can be generated either by
  - external hardware,
  - software, or
  - by the processor.
- Interrupts 0, 1, 3, 4, 6 and 7 are generated by the processor.
- Interrupts from 8 to 15 and interrupt 2 are generated by external hardware.
  - These get the attention of the processor by activating a interrupt request (IRQ) line.
  - The IRQ0 line connects to the system timer, the keyboard to IRQ1, and so on.
- Most other interrupts are generated by software.

Interrupt handling

Interrupt	***Name***	***Generated by***	***Interrupt***	***Name***	***Generated by***
00 (00h)	Divide error	processor	14 (0Eh)	Floppy disk controller	HW via IRQ6
01 (00h)	Single step	processor	15 (0Fh)	Parallel printer	HW via IRQ7
02 (02h)	NMI	external equipment	16 (10h)	BIOS - Video access	software
03 (03h)	Breakpoint	processor	17 (11h)	BIOS - Equipment check	software
04 (04h)	Overflow	processor	18 (12h)	BIOS – Memory size	software
05 (05h)	Print screen	Shft-PrtScr key stroke	19 (13h)	BIOS – Disk operations	software
06 (06h)	Reserved	processor	20 (14h)	BIOS – Serial communic.	software
07 (07h)	Reserved	processor	22 (16h)	BIOS – Keyboard	software
08 (08h)	System timer	HW via IRQ0	23 (17h)	BIOS – Printer	software
09 (09h)	Keyboard	HW via IRQ1	25 (19h)	BIOS – Reboot	software
10 (0Ah)	Reserved	HW via IRQ2	26 (1Ah)	BIOS – Time of day	software
11 (0Bh)	Serial (COM2)	HW via IRQ3	28 (1Ch)	BIOS – Ticker timer	software
12 (0Ch)	Serial (COM1)	HW via IRQ4	33 (21h)	DOS – DOS services	software
13 (0Dh)	Reserved	HW via IRQ5	39 (27h)	DOS – Terminate and stay resident	software

Interrupt vectors

- Each device that requires to be ‘interrupt-driven’ is assigned an IRQ (interrupt request) line.
- Each IRQ is active high. The first eight (IRQ0–IRQ7) map into interrupts 8 to 15 (08h–0Fh)
- and the next eight (IRQ8–IRQ15) into interrupts 112 to 119 (70h–77h). Table 2.6 outlines the usage of each of these interrupts.
- When IRQ0 is made active, the ISR corresponds to interrupt vector 8. IRQ0 normally connects to the system timer, the keyboard to IRQ1, and so on.
- The system timer interrupts the processor 18.2 times per second and is used to update the system time.
- When the keyboard has data, it interrupts the processor with the IRQ1 line.

Interrupt handling

Interrupt	***Name***	***Gener. by***	***Interrupt***	***Name***	***Gener. by***
08 (08h)	System timer	IRQ0	112 (70h)	Real-time clock	IRQ8
09 (09h)	Keyboard	IRQ1	113 (71h)	Redirection of IRQ2	IRQ9
10 (0Ah)	Reserved	IRQ2	114 (72h)	Reserved	IRQ10
11 (0Bh)	COM2	IRQ3	115 (73h)	Reserved	IRQ11
12 (0Ch)	COM1	IRQ4	116 (74h)	Reserved	IRQ12
13 (0Dh)	Parallel port (LPT2:)	IRQ5	117 (75h)	Math co-processor	IRQ13
14 (0Eh)	Floppy disk controller	IRQ6	118 (76h)	Hard disk controller	IRQ14
15 (0Fh)	Parallel printer (LPT1:)	IRQ7	119 (77h)	Reserved	IRQ15

Typical uses of interrupts:

- **IRQ0: System timer** The system timer uses IRQ0 to interrupt the processor 18.2 times per second and is used to keep the time-of-day clock updated.
- **IRQ1: Keyboard data ready** The keyboard uses IRQ1 to signal to the processor that data is ready to be received from the keyboard. This data is normally a scan code.
- **IRQ2: Redirection of IRQ9** The BIOS redirects the interrupt for IRQ9 back here.
- **IRQ3: Secondary serial port (COM2:)** The secondary serial port (COM2:) uses IRQ3 to interrupt the processor. Typically, COM3: to COM8: also use it, although COM3: may use IRQ4.
- **IRQ4: Primary serial port (COM1:)** The primary serial port (COM1:) uses IRQ4 to interrupt the processor. Typically, COM3: also uses it.
- **IRQ5: Secondary parallel port (LPT2:)** On older PCs the IRQ5 line was used by the fixed disk. On newer systems the secondary parallel port uses it. Typically, it is used by a sound card on PCs which have no secondary parallel port connected.
- **IRQ6: Floppy disk controller** The floppy disk controller activates the IRQ6 line on completion of a disk operation.
- **IRQ7: Primary parallel port (LPT1:)** Printers (or other parallel devices) activate the IRQ7 line when they become active. As with IRQ5 it may be used by another device, if there are no other devices connected to this line.
- **IRQ9 Redirected to IRQ2 service routine.**

CPU modes

- CPU modes (also called processor modes, CPU states, CPU privilege levels and other names) are operating modes for the central processing unit of some computer architectures that place restrictions on the type and scope of operations that can be performed by certain processes being run by the CPU.
    - This design allows the operating system to run with more privileges than application software.
- Ideally, only highly-trusted kernel code is allowed to execute in the unrestricted mode;
    - everything else (including non-supervisory portions of the operating system) run in a restricted mode and must use a system call to request the kernel perform on its behalf any operation that could damage or compromise the system, making it impossible for untrusted programs to alter or damage other programs (or the computing system itself).

CPU modes

- In practice, however, system calls take time and can hurt the performance of a computing system, so it is not uncommon for system designers to allow some time-critical software (especially device drivers) to run with full kernel privileges.
- Multiple modes can be implemented—allowing a hypervisor to run multiple operating system supervisors beneath it, which is the basic design of many virtual machine systems available today.

Mode types

- At a minimum, any CPU architecture supporting protected execution will offer two distinct operating modes; at least one of the modes must allow completely unrestricted operation of the processor.
    - The unrestricted mode is often called **kernel mode**, but many other designations exist (master mode, supervisor mode, privileged mode, supervisor state, etc.).
    - Restricted modes are usually referred to as **user modes**, but are also known by many other names (slave mode, user mode, problem state, etc.).

Kernel Mode

- In kernel mode, the CPU may perform any operation allowed by its architecture;
  - any instruction may be executed,
  - any I/O operation initiated,
  - any area of memory accessed, and so on.

Kernel Mode Execution

- can execute privileged instructions
  - able to perform I/O operations
  - interrupt enable/disable/return, load PS
  - instructions to change processor mode
- can access privileged address spaces
  - access data structures inside the OS
  - access other process's address spaces
  - change and create address spaces
- may have alternate registers, alternate stack

User Mode

- In the other CPU modes, certain restrictions on CPU operations are enforced by the hardware.
  - Typically, certain instructions are not permitted (especially those—including I/O operations—that could alter the global state of the machine), some memory areas cannot be accessed, etc.
- User-mode capabilities of the CPU are typically a subset of those available in kernel mode but in some cases, such as hardware emulation of non-native architectures, they may be significantly different from those available in standard kernel mode.

User Mode Execution

- able to use all of the “normal” instructions
  - load and store general registers from/to memory
  - arithmetic, logical, test, compare, data copying
  - branches and subroutine calls
- able to address some subset of memory
  - what is controlled by a Memory Management Unit
- not able to perform privileged operations
  - I/O operations, update the MMU
  - interrupt enables, enter supervisor mode

References

- PICmicro® MCU Mid-Range Family Reference Manual (DS33023A), *Microchip Technology*
- Programming and Customizing PICmicro Microcontrollers, *by Myke Predko*
- Design with PIC® Microcontrollers *by John B. Peatman*
- 123 PIC® Microcontroller Experiments for the Evil Genius by Myke Predko