



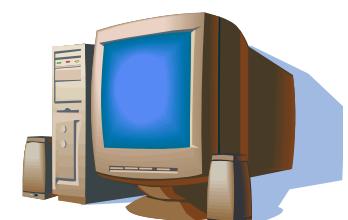
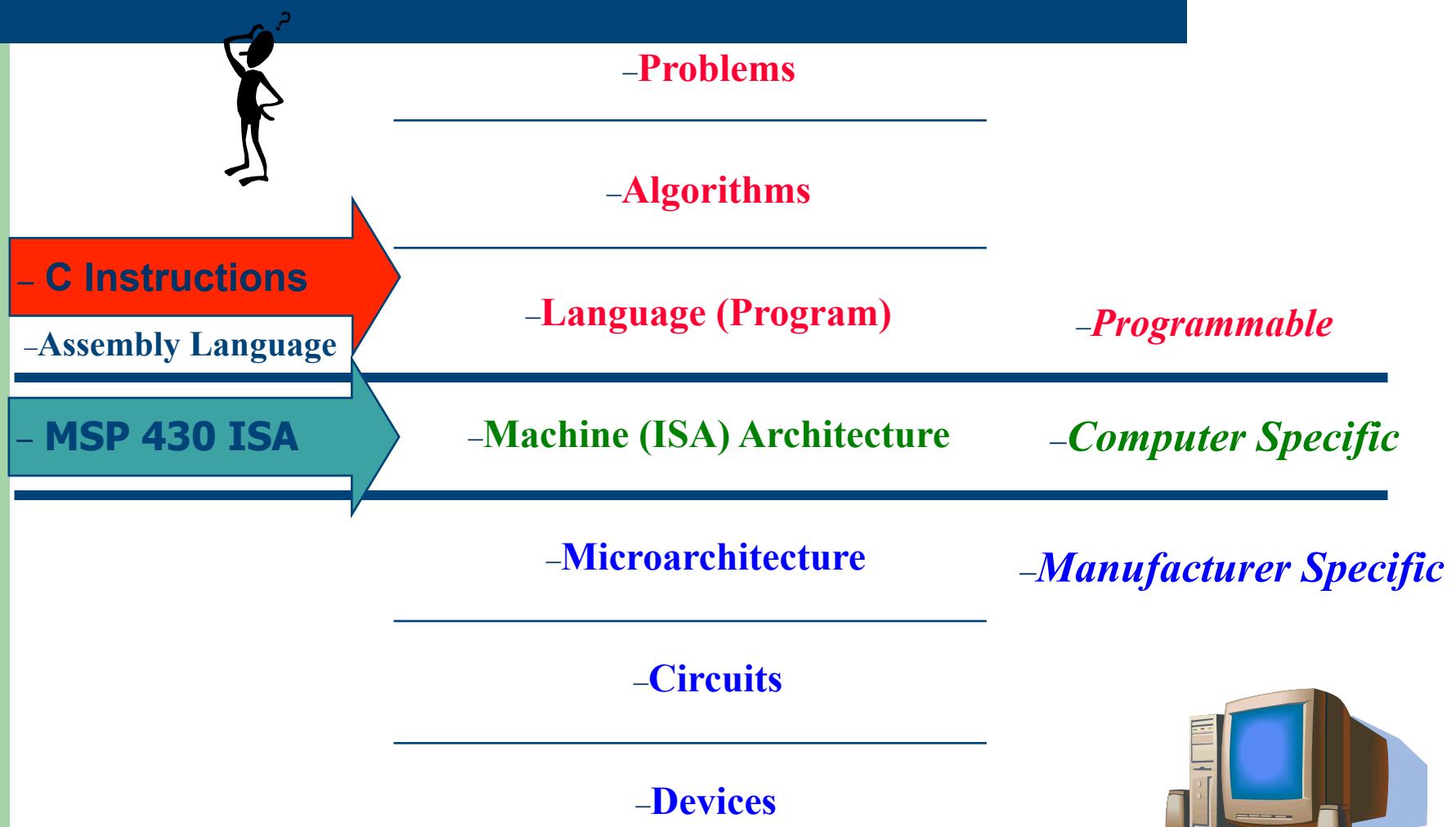
Instruction Set Architecture

EE3376

Topics to Cover...

- MSP430 ISA
- MSP430 Registers, ALU, Memory
- Instruction Formats
- Addressing Modes
- Double Operand Instructions
- Single Operand Instructions
- Jump Instructions
- Emulated Instructions
 - http://en.wikipedia.org/wiki/TI_MSP430

Levels of Transformation



-Adapted from notes from BYU ECE124

Instruction Set Architecture

- The computer ISA defines all of the *programmer-visible* components and operations of the computer
 - memory organization
 - address space -- how many locations can be addressed?
 - addressability -- how many bits per location?
 - register set (a place to store a collection of bits)
 - how many? what size? how are they used?
 - instruction set
 - Opcodes (operation selection codes)
 - data types (data types: byte or word)
 - addressing modes (coding schemes to access data)
- ISA provides all information needed for someone that wants to write a program in machine language (or translate from a high-level language to machine language).

MSP430 Instruction Set Architecture

- MSP430 CPU specifically designed to allow the use of modern programming techniques, such as:
 - the computation of jump addresses
 - data processing in tables
 - use of high-level languages such as C.
- 64KB memory space with 16 16-bit registers that reduce fetches to memory.
- Implements RISC architecture with 27 instructions and 7 addressing modes.

MSP430 16-bit RISC

- Orthogonal architecture with every instruction usable with every addressing mode.
- Full register access including program counter, status registers, and stack pointer.
- Single-cycle register operations.
- 16-bit address bus allows direct access and branching throughout entire memory range.
- 16-bit data bus allows direct manipulation of word-wide arguments.
- Word and byte addressing and instruction formats.

MSP430 Registers

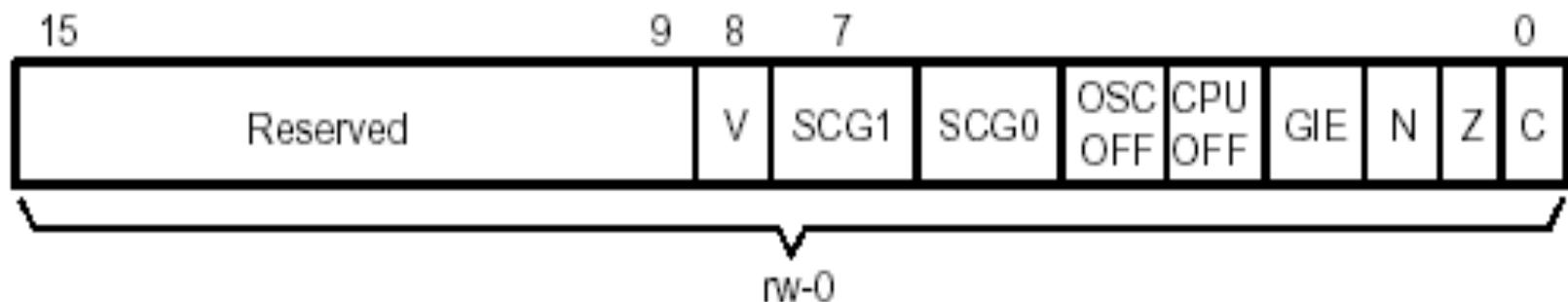
- The MSP430 CPU has 16 registers
 - Large 16-bit register file eliminates single accumulator bottleneck
 - High-bandwidth 16-bit data and address bus
- R0 (PC) – Program Counter
 - This register always points to the next instruction to be fetched
 - Each instruction occupies an even number of bytes. Therefore, the least significant bit (LSB) of the PC register is always zero.
 - After fetch of an instruction, the PC register is incremented by 2, 4, or 6 to point to the next instruction.

MSP430 Registers

- R1 (SP) – Stack Pointer
 - The MSP430 CPU stores the return address of routines or interrupts on the stack
 - User programs store local data on the stack
 - The SP can be incremented or decremented automatically with each stack access
 - The stack “grows down” thru RAM and thus SP must be initialized with a valid RAM address
 - SP always points to an even address, so its LSB is always zero

MSP430 Registers

- R2 (SR/CG1) – Status Register
 - The status of the MSP430 CPU is defined by a set of bits contained in register R2
 - This register can only be accessed through register addressing mode - all other addressing modes are reserved to support the constants generator
 - The status register is used for clock selection, interrupt enable/disable, and instruction result status



R2 (SR) – Status Register

V	Overflow bit – set when arithmetic operation overflows the signed-variable range.
SCG1	System clock generator 1 – turns off the SMCLK.
SCG0	System clock generator 0 – turns off the DCO dc generator.
OSCOFF	Oscillator off – turns off the LFXT1 crystal oscillator.
CPUOFF	CPU off – turns off the CPU.
GIE	General interrupt enable – enables maskable interrupts.
N	Negative bit – set when the result of a byte or word operation is negative.
Z	Zero bit – set when the result of a byte or word operation is 0.
C	Carry bit – set when the result of a byte or word operation produces a carry.

R2 (SR) – Status Register

- R2 (SR/CG1), R3 (CG2) – Constant Generators
 - Six different constants commonly used in programming can be generated using the registers R2 and R3, without adding a 16-bit extension word of code to the instruction

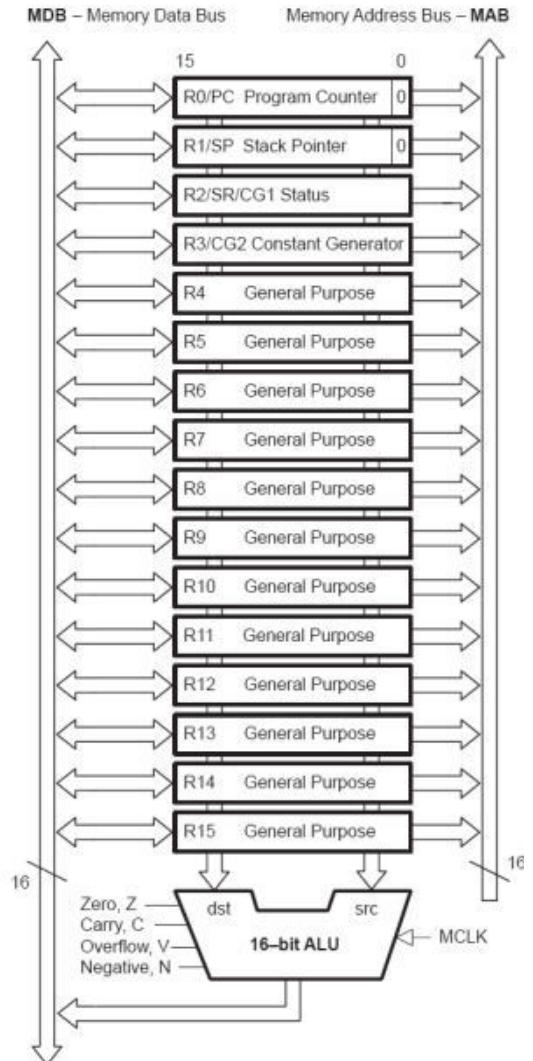
Register	As	Constant	Remarks
R2	00	-	Register mode
R2	01	(0)	Absolute mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	-1, word processing

MSP430 Registers

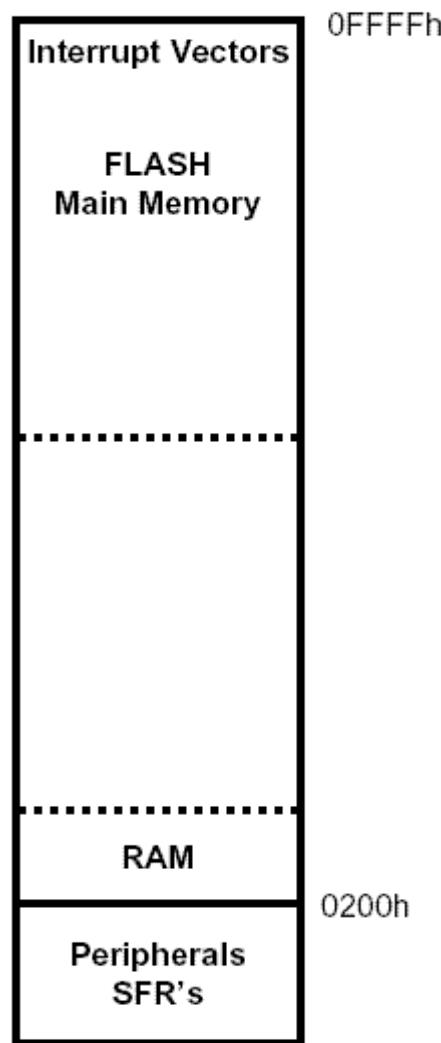
- R4-R15 – General Purpose registers
 - The general purpose registers R4 to R15 can be used as data registers, data pointers and indices.
 - They can be accessed either as a byte or as a word
 - Instruction formats support byte or word accesses
 - The status bits of the CPU in the SR are updated after the execution of a register instruction.

MSP430 ALU

- 16 bit Arithmetic Logic Unit (ALU).
 - Performs instruction arithmetic and logical operations
 - Instruction execution affects the state of the following flags:
 - Zero (Z)
 - Carry (C)
 - Overflow (V)
 - Negative (N)
 - The MCLK (Master) clock signal drives the CPU.



MSP430 Memory



- Unified 64KB continuous memory map
- Same instructions for data and peripherals
- Program and data in Flash or RAM with no restrictions
- Designed for modern programming techniques such as pointers and fast look-up tables

Anatomy of an Instruction

- Opcode
 - What the instruction does – verb
 - May or may not require operands – objects
- Source Operand
 - 1st data object manipulated by the instruction
- Destination Operand
 - 2nd data object manipulated by the instruction
 - Also where results of operation are stored.
- Addressing Modes

Instruction Format

- There are three formats used to encode instructions for processing by the CPU core
 - Double operand
 - Single operand
 - Jumps
- The instructions for double and single operands, depend on the suffix used, (.W) word or (.B) byte
- These suffixes allow word or byte data access
- If the suffix is ignored, the instruction processes **word data** by default

Instruction Format

- The source and destination of the data operated by an instruction are defined by the following fields:
 - **src**: source operand address, as defined in As and S-reg
 - **dst**: destination operand address, as defined in Ad and D-reg
 - **As**: addressing bits used to define the addressing mode used by the source operand
 - **S-reg**: register used by the source operand
 - **Ad**: Addressing bits used to define the addressing mode used by the destination operand
 - **D-reg**: register used by the destination operand
 - **b/w**: word or byte access definition bit.

MPS430 Instruction Formats

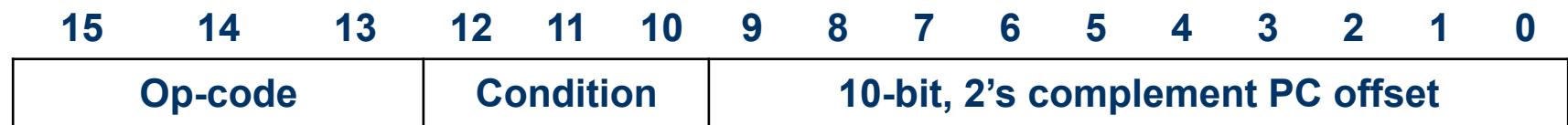
- Format I: Instructions with two operands:



- Format II: Instruction with one operand:



- Format II: Jump instructions:



3 Instruction Formats

; Format I Source and Destination

Op-Code	Source-Register	Ad	B/W	As	Destination-Register
---------	-----------------	----	-----	----	----------------------

5405	add.w	R4 , R5			; R4+R5=R5 xxxx
5445	add.b	R4 , R5			; R4+R5=R5 00xx

; Format II Destination Only

Op-Code	B/W	Ad	D/S- Register
---------	-----	----	---------------

6404	rlc.w	R4	;
6444	rlc.b	R4	;

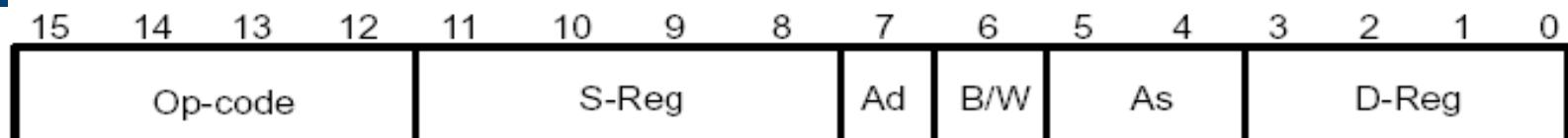
; Format III There are 8 (Un)conditional Jumps

Op-Code	Condition	10-bit PC offset
---------	-----------	------------------

3c28	jmp	Loop_1	; Goto Loop_1
------	-----	--------	---------------

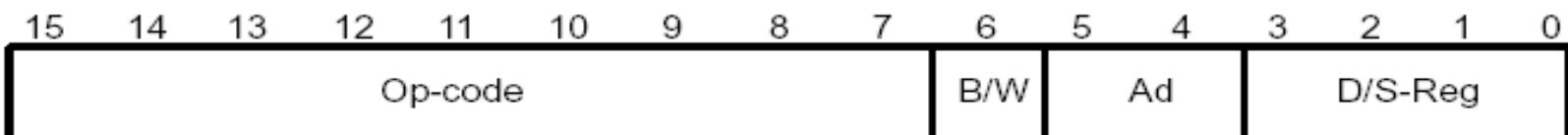
-Adapted from notes from BYU ECE124

Double Operand Instructions



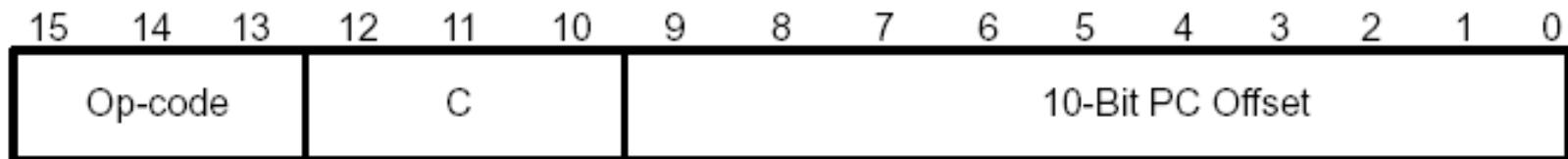
Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV (.B)	src, dst	src → dst	—	—	—	—
ADD (.B)	src, dst	src + dst → dst	*	*	*	*
ADDC (.B)	src, dst	src + dst + C → dst	*	*	*	*
SUB (.B)	src, dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC (.B)	src, dst	dst + .not.src + C → dst	*	*	*	*
CMP (.B)	src, dst	dst – src	*	*	*	*
DADD (.B)	src, dst	src + dst + C → dst (decimally)	*	*	*	*
BIT (.B)	src, dst	src .and. dst	0	*	*	*
BIC (.B)	src, dst	.not.src .and. dst → dst	—	—	—	—
BIS (.B)	src, dst	src .or. dst → dst	—	—	—	—
XOR (.B)	src, dst	src .xor. dst → dst	*	*	*	*
AND (.B)	src, dst	src .and. dst → dst	0	*	*	*

Single Operand Instruction



Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC (.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA (.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH (.B)	src	SP – 2 → SP, src → @SP	—	—	—	—
SWPB	dst	Swap bytes	—	—	—	—
CALL	dst	SP – 2 → SP, PC+2 → @SP dst → PC	—	—	—	—
RETI		TOS → SR, SP + 2 → SP TOS → PC,SP + 2 → SP	*	*	*	*
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

Jump Instructions



Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if $(N \text{ .XOR. } V) = 0$
JL	Label	Jump to label if $(N \text{ .XOR. } V) = 1$
JMP	Label	Jump to label unconditionally

Source Addressing Modes

- The MSP430 has four basic modes for the source address:
 - Rs - Register
 - $x(Rs)$ - Indexed Register
 - $@Rs$ - Register Indirect
 - $@Rs+$ - Indirect Auto-increment
- In combination with registers R0-R3, three additional source addressing modes are available:
 - **label** - PC Relative, $x(PC)$
 - **&label** – Absolute, $x(SR)$
 - **#n** – Immediate, $@PC+$

Destination Addressing Modes

- There are two basic modes for the destination address:
 - Rd - Register
 - $x(Rd)$ - Indexed Register
- In combination with registers R0/R2, two additional destination addressing modes are available:
 - **label** - PC Relative, $x(PC)$
 - **&label** – Absolute, $x(SR)$

Register Mode (Rn)

- The most straightforward addressing mode and is available for both source and destination
 - **Example:**
`mov.w r5,r6 ; move word from r5 to r6`
- The registers are specified in the instruction; no further data is needed
- Also the fastest mode and does not require an addition cycle
- Byte instructions use only the lower byte, but clear the **upper byte when writing**

0	1	0	0	0	1	0	1	0	0	0	0	1	1	0
Op-code				S-reg			Ad	b/w	As	D-reg				

Indexed Mode x(Rn)

- The address is formed by adding a constant (index) to the contents of a CPU register
 - Example:**

```
mov.b 3(r5),r6 ; move byte from
                  ; M(310+r5) to r6
```
- Indexed addressing can be used for source and/or destination, value in r5 is unchanged.
- The index is located in the memory word following the instruction and requires an additional memory cycle
- There is no restriction on the address for a byte, but words must lie on even addresses

0	1	0	0	0	1	0	1	0	1	0	1	1	0
Op-code		S-reg		Ad	b/w	As	D-reg						

Symbolic Mode (PC Relative)

- The address is formed by adding a constant (index) to the program counter (PC)
 - Example:** (mov.w x(PC), r6 where x=Cnt-PC)
- mov.w Cnt,r6** ; move word
 ; M(Cnt+PC) to r6
- The PC relative index is calculated by the assembler
 - Produces position-independent code, but rarely used in the MSP430 because absolute addressing can reach all memory addresses
 - Note: this is NOT an appropriate mode of addressing when referencing fixed locations in memory such as the special function registers (SFR's)

0	1	0	0	0	0	0	0	0	0	1	0	1	1	0
Op-code			S-reg			Ad	b/w	As	D-reg					

Absolute Mode (&label)

- The address is formed directly from a constant (index) and specified by preceding a label with an ampersand (&)
 - Example:** (mov.w x(SR), r6 where 0 is used for SR)

**mov.w &Cnt,r6 ; move word
; M(Cnt) to r6**

- Same as indexed mode with the base register value of 0 (by using the status register SR as the base register)
- The absolute address is stored in the memory word following the instruction and requires an additional cycle
- Note: this is the preferred mode of addressing when referencing fixed locations in memory such as the special function registers (SFR's)

0 1 0 0 0 1 0 0 0 1 0 1 1 0

Op-code	S-reg	Ad	b/w	As	D-reg
---------	-------	----	-----	----	-------

- Adapted from notes from BYU ECE124

Indirect Register Mode (@Rn)

- The address of the operand is formed from the contents of the specified register

– Example:

**mov.w @r5,r6 ; move word
; M(r5) to r6**

- Only available for source operands
- Same as indexed mode with index equal to 0, but does not require an additional instruction word
- The value of the indirect register is unchanged

0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Op-code

S-reg

Ad

b/w

As

D-reg

Indirect Autoincrement Mode (@Rn+)

- The address of the operand is formed from the contents of the specified register and afterwards, the register is automatically incremented by 1 if a byte is fetched or by 2 if a word is fetched
 - Example:**

**mov.w @r5+,r6 ; move word
; M(r5) to r6
; increment r5 by 2**

- Only available for source operands.
- Usually called **post-increment** addressing.
- Note: All operations on the first address are fully completed before the second address is evaluated**

0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Op-code	S-reg	Ad	b/w	As	D-reg
---------	-------	----	-----	----	-------

Immediate Mode (#n)

- The operand is an immediate value
 - Example** (mov.w @PC+, r6)

mov.w #100,r6 ; 100 -> r6
- The immediate value is located in the memory word following the instruction
- Only available for source operands
- The immediate mode of addressing is a special case of auto-increment addressing that uses the program counter (PC) as the source register.
- The PC is automatically incremented after the instruction is fetched; hence points to the following word

0	1	0	0	0	0	0	0	0	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Op-code	S-reg	Ad	b/w	As	D-reg
---------	-------	----	-----	----	-------

Constant Generators

- The following source register/addressing mode combinations result in a commonly used constant operand value
- Do not require an additional instruction word

Register	As	Constant	Remarks
R2	00	-----	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	-1, word processing

Addressing Summary

ADDRESS MODE	S	D	SYNTAX	EXAMPLE	OPERATION
Register	●	●	MOV Rs,Rd	MOV R10,R11	R10 --> R11
Indexed	●	●	MOV X(Rn),Y(Rm)	MOV 2(R5),6(R6)	M(2+R5)--> M(6+R6)
Symbolic (PC relative)	●	●	MOV EDE,TONI		M(EDE) --> M(TONI)
Absolute	●	●	MOV &MEM,&TCDAT		M(MEM) --> M(TCDAT)
Indirect	●		MOV @Rn,Y(Rm)	MOV @R10,Tab(R6)	M(R10) --> M(Tab+R6)
Indirect autoincrement	●		MOV @Rn+,Rm	MOV @R10+,R11	M(R10) --> R11 R10 + 2--> R10
Immediate	●		MOV #X,TONI	MOV #45,TONI	#45 --> M(TONI)

NOTE: S = source D = destination

Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/-	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/-	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/-	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Format I: Double Operand

- Double operand instructions:

Mnemonic	Operation	Description
Arithmetic instructions		
ADD(.B or .W) src,dst	src+dst→dst	Add source to destination
ADDC(.B or .W) src,dst	src+dst+C→dst	Add source and carry to destination
DADD(.B or .W) src,dst	src+dst+C→dst (dec)	Decimal add source and carry to destination
SUB(.B or .W) src,dst	dst+.not.src+1→dst	Subtract source from destination
SUBC(.B or .W) src,dst	dst+.not.src+C→dst	Subtract source and not carry from destination
Logical and register control instructions		
AND(.B or .W) src,dst	src.and.dst→dst	AND source with destination
BIC(.B or .W) src,dst	.not.src.and.dst→dst	Clear bits in destination
BIS(.B or .W) src,dst	src.or.dst→dst	Set bits in destination
BIT(.B or .W) src,dst	src.and.dst	Test bits in destination
XOR(.B or .W) src,dst	src.xor.dst→dst	XOR source with destination
Data instructions		
CMP(.B or .W) src,dst	dst-src	Compare source to destination
MOV(.B or .W) src,dst	src→dst	Move source to destination

Example: Double Operand

- Copy the contents of a register to another register
 - Assembly: **mov.w r5,r4**
 - Instruction code: **0x4504**

<u>Op-code</u> <i>mov</i>	<u>S-reg</u> <i>r5</i>	<u>Ad</u> <i>Register</i>	<u>b/w</u> <i>16-bits</i>	<u>As</u> <i>Register</i>	<u>D-reg</u> <i>r4</i>
0 1 0 0	0 1 0 1	0	0	0 0	0 1 0 0

- One word instruction
- The instruction instructs the CPU to copy the 16-bit 2's complement number in register **r5** to register **r4**

Example: Double Operand

- Copy the contents of a register to a PC-relative memory address location
 - Assembly: **mov.w r5,TONI**
 - Instruction code: **0x4580**

<u>Op-code</u> <i>mov</i>	<u>S-reg</u> <i>r5</i>	<u>Ad</u> <i>Symbolic</i>	<u>b/w</u> <i>16-bits</i>	<u>As</u> <i>Register</i>	<u>D-reg</u> <i>PC</i>
0 1 0 0	0 1 0 1	1	0	0 0	0 0 0 0
2's complement PC-relative destination index					

- Two word instruction
- The instruction instructs the CPU to copy the 16-bit 2's complement word in register **r5** to the memory location whose address is obtained by adding the **PC** to the memory word following the instruction

Example: Double Operand

- Copy the contents of a PC-relative memory location to another PC-relative memory location
 - Assembly: **mov.b EDEN,TONI**
 - Instruction code: **0x40d0**

<u>Op-code</u> <i>mov</i>	<u>S-reg</u> <i>PC</i>	<u>Ad</u> <i>Symbolic</i>	<u>b/w</u> <i>8-bits</i>	<u>As</u> <i>Symbolic</i>	<u>D-reg</u> <i>PC</i>
0 1 0 0	0 0 0 0	1	1	0 1	0 0 0 0
2's complement PC-relative source index					
2's complement PC-relative destination index					

- Three word instruction
- The CPU copies the 8-bit contents of EDEN (pointed to by **source index + PC**) to TONI (pointed to by **destination index + PC**)

Format II: Single Operand

- Single operand instructions:

Mnemonic	Operation	Description
Logical and register control instructions		
RRA(.B or .W) dst	MSB→MSB→... LSB→C	Roll destination right
RRC(.B or .W) dst	C→MSB→...LSB→C	Roll destination right through carry
SWPB(or .W) dst	Swap bytes	Swap bytes in destination
SXT dst	bit 7→bit 8...bit 15	Sign extend destination
PUSH(.B or .W) src	SP-2→SP, src→@SP	Push source on stack
Program flow control instructions		
CALL(.B or .W) dst	SP-2→SP, PC+2→@SP dst→PC	Subroutine call to destination
RETI	@SP+→SR, @SP+→SP	Return from interrupt

Example: Single Operand

- Logically shift the contents of register **r5** to the right through the status register carry
 - Assembly: **rrc.w r5**
 - Instruction code: **0x1005**

<u>Op-code</u> <i>rrc</i>	<u>b/w</u> <i>16-bits</i>	<u>Ad</u> <i>Register</i>	<u>D-reg</u> <i>r5</i>
0 0 0 1 0 0 0 0	0	0 0	0 1 0 1

- One word instruction
- The CPU shifts the 16-bit register **r5** one bit to the right (divide by 2) – the carry bit prior to the instruction becomes the MSB of the result while the LSB shifted out replaces the carry bit in the status register

Example: Single Operand

- Arithmetically shift the contents of absolute memory location **P2OUT** to the right through the SR carry
 - Assembly: **rra.b &P2OUT**
 - Instruction code: **0x1152**

<u>Op-code</u> <i>rra</i>	<u>b/w</u> 8-bits	<u>Ad</u> <i>Indexed</i>	D-reg <i>r2</i>
0 0 0 1 0 0 0 1 0	1	0 1	0 0 1 0
Absolute memory address (P2OUT)			

- Two word instruction
- The CPU arithmetically shifts the 8-bit memory location **P2OUT** one bit to the right (divide by 2) – MSB prior to the instruction becomes the MSB of the result while the LSB shifted out replaces the carry bit in the SR

Jump Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code		Condition			10-bit, 2's complement PC offset										

- Jump instructions are used to direct program flow to another part of the program.
- The condition on which a jump occurs depends on the Condition field consisting of 3 bits:
 - 000: jump if not equal
 - 001: jump if equal
 - 010: jump if carry flag equal to zero
 - 011: jump if carry flag equal to one
 - 100: jump if negative ($N = 1$)
 - 101: jump if greater than or equal ($N = V$)
 - 110: jump if lower ($N \neq V$)
 - 111: unconditional jump

Jump Instruction Format

- Jump instructions are executed based on the current PC and the status register
- Conditional jumps are controlled by the status bits
- Status bits are not changed by a jump instruction
- The jump off-set is represented by the 10-bit, 2's complement value:

$$PC_{new} = PC_{old} + 2 + PC_{offset} \times 2$$

- Thus, the range of the jump is -511 to +512 words, (-1022 to 1024 bytes) from the current instruction
- Note: Use a BR instruction to jump to any address

Example: Jump Format

- Continue execution at the label **main** if the carry bit is set
 - Assembly: jc main
 - Instruction code: **0x2fe4**

<u>Op-code</u> JC	<u>Condition</u> Carry Set	<u>10-Bit, 2's complement PC offset</u> -28
0 0 1	0 1 1	1 1 1 1 1 0 0 1 0 0

- One word instruction
- The CPU will add to the **PC** (R0) the value **-28 x 2** if the carry is set

Emulated Instructions

- In addition to the 27 instructions of the CPU there are 24 emulated instructions
- The CPU coding is unique
- The emulated instructions make reading and writing code easier, but do not have their own op-codes
- Emulated instructions are replaced automatically by instructions from the CPU
- There are no penalties for using emulated instructions.

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Arithmetic instructions			
ADC(.B or .W) dst	dst+C→dst	ADDC(.B or .W) #0,dst	Add carry to destination
DADC(.B or .W) dst	d s t + C → d s t (decimally)	DADD(.B or .W) #0,dst	Decimal add carry to destination
DEC(.B or .W) dst	dst-1→dst	SUB(.B or .W) #1,dst	Decrement destination
DECD(.B or .W) dst	dst-2→dst	SUB(.B or .W) #2,dst	Decrement destination twice
INC(.B or .W) dst	dst+1→dst	ADD(.B or .W) #1,dst	Increment destination
INCD(.B or .W) dst	dst+2→dst	ADD(.B or .W) #2,dst	Increment destination twice
SBC(.B or .W) dst	dst+0FFFFh+C→dst dst+0FFh→dst	SUBC(.B or .W) #0,dst	Subtract source and borrow /.NOT. carry from dest.

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Logical and register control instructions			
INV(.B or .W) dst	.NOT.dst→dst	XOR(.B or .W) #0(FF)FFh,dst	Invert bits in destination
RLA(.B or .W) dst	C←MSB←MSB-1 LSB+1←LSB←0	ADD(.B or .W) dst,dst	Rotate left arithmetically
RLC(.B or .W) dst	C←MSB←MSB-1 LSB+1←LSB←C	ADDC(.B or .W) dst,dst	Rotate left through carry
Program flow control			
BR dst	dst→PC	MOV dst,PC	Branch to destination
DINT	0→GIE	BIC #8,SR	Disable (general) interrupts
EINT	1→GIE	BIS #8,SR	Enable (general) interrupts
NOP	None	MOV #0,R3	No operation
RET	@SP→PC SP+2→SP	MOV @SP+,PC	Return from subroutine

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Data instructions			
CLR(.B or .W) dst	0→dst	MOV(.B or .W) #0,dst	Clear destination
CLRC	0→C	BIC #1,SR	Clear carry flag
CLRN	0→N	BIC #4,SR	Clear negative flag
CLRZ	0→Z	BIC #2,SR	Clear zero flag
POP(.B or .W) dst	@SP→temp SP+2→SP temp→dst	MOV(.B or .W) @SP +,dst	Pop byte/word from stack to destination
SETC	1→C	BIS #1,SR	Set carry flag
SETN	1→N	BIS #4,SR	Set negative flag
SETZ	1→Z	BIS #2,SR	Set zero flag
TST(.B or .W) dst	dst + 0FFFFh + 1 dst + 0FFh + 1	CMP(.B or .W) #0,dst	Test destination

Example: Emulated Instructions

- Clear the contents of register R5:
-CLR R5

- Instruction code: 0x4305

<u>Op-code</u> <i>mov</i>	<u>S-reg</u> <i>r3</i>	<u>Ad</u> <i>Register</i>	<u>b/w</u> <i>16-bits</i>	<u>As</u> <i>Register</i>	<u>D-reg</u> <i>r5</i>
0 1 0 0	0 0 1 1	0	0	0 0	0 1 0 1

- This instruction is equivalent to **MOV R3 ,R5**, where R3 takes the value #0.

Example: Emulated Instructions

- Increment the content of register R5:

- INC R5

- Instruction code: 0x5315

<u>Op-code</u> add	<u>S-reg</u> r3	<u>Ad</u> Register	<u>b/w</u> 16-bits	<u>As</u> Indexed	<u>D-reg</u> r5
0 1 0 1	0 0 1 1	0	0	0 1	0 1 0 1

- This instruction is equivalent to ADD 0 (R3) , R5 where R3 takes the value #1.

Example: Emulated Instructions

- Decrement the contents of register R5:

-DEC R5

- Instruction code: 0x8315

<u>Op-code</u> sub	<u>S-reg</u> r3	<u>Ad</u> Register	<u>b/w</u> 16-bits	<u>As</u> Indexed	<u>D-reg</u> r5
1 0 0 0	0 0 1 1	0	0	0 1	0 1 0 1

- This instruction is equivalent to SUB 0 (R3) ,R5 where R3 takes the value #1.

Example: Emulated Instructions

- Decrement by two the contents of register R5:

-DECD R5

- Instruction code: 0x8325

<u>Op-code</u> sub	<u>S-reg</u> r3	<u>Ad</u> Register	<u>b/w</u> 16-bits	<u>As</u> Indirect	<u>D-reg</u> r5
1 0 0 0	0 0 1 1	0	0	1 0	0 1 0 1

- This instruction is equivalent to **SUB @R3 ,R5**, where R3 points to the value #2.

Example: Emulated Instructions

- Do not carry out any operation:

– NOP

- Instruction code: 0x4303

<u>Op-code</u> mov	<u>S-reg</u> r3	<u>Ad</u> Register	<u>b/w</u> 16-bits	<u>As</u> Register	<u>D-reg</u> r5
0 1 0 0	0 0 1 1	0	0	0 0	0 0 1 1

- This instruction is equivalent to **MOV R3 ,R3** and therefore the contents of R3 are moved to itself.

Example: Emulated Instructions

- Add the carry flag to the register R5:
-ADC R5

- Instruction code: 0x6305

<u>Op-code</u> addc	<u>S-reg</u> r3	<u>Ad</u> Register	<u>b/w</u> 16-bits	<u>As</u> Register	<u>D-reg</u> r5
0 1 1 0	0 0 1 1	0	0	0 0	0 1 0 1

- This instruction is equivalent to ADDC R3 ,R5, where R3 takes the value #0.

Assembly to Machine Code

-Memory Location	-Machine code instruction	-Machine code information	-Assembly code
-0x8000:	4031	0300	MOV.W #0x0300,SP
-0x8004:	40B2	5A80 0120	MOV.W #0x5a80,&Watchdog_Timer_WDTCTL
-0x800a:	D0F2	000F 0022	BIS.B #0x000f,&Port_1_2_P1DIR
-0x8010:	430E		CLR.W R14
-			
Mainloop:			
-0x8012:	4EC2	0021	MOV.B R14,&Port_1_2_P1OUT
-0x8016:	531E		INC.W R14
-0x8018:	F03E	000F	AND.W #0x000f,R14
-			
Wait:			
-0x801c:	401F	000E	MOV.W Delay,R15
-0x8020:	120F		PUSH R15
-			
L1:			
-0x8022:	8391	0000	DEC.W 0x0000(SP)
-0x8026:	23FD		JNE (L1)
-0x8028:	413F		POP.W R15
-0x802a:	3FF3		JMP (Mainloop)
-			
Delay:			
-0x802c:	0002		.word 0x0002

Machine Code in the Memory

	Memory Location	Machine Code	Description
-# for immediate value			-Require 1 extra word to store the immediate value 0x0300
-& for absolute address			-Require 2 extra words to store the immediate value 0x5A80 and the absolute address WDTCTL
-Symbol			-Require 2 extra words to store the immediate value 0x000F and the absolute address Port_1_2_P1DIR
-Index value			-Require 1 extra word to store the immediate value 0x000F
-Label			-Require 1 extra word to store the symbolic info to get Delay
			-Require 1 extra word to store the index value 0x0000
	0x8000:	4031	MOV.W #0x0300, SP
	0x8002:	0300	
	0x8004:	40B2	MOV.W #0x5a80, &Watchdog_Timer_WDTCTL
	0x8006:	5A80	
	-0x8008:	0120	
	0x800a:	D0F2	BIS.B #0x000f, &Port_1_2_P1DIR
	0x800c:	000F	
	0x800e:	0022	
	0x8010:	430E	CLR.W R14
	Mainloop:0x8012:	4EC2	MOV.B R14, &Port_1_2_P1OUT
	0x8014:	0021	
	0x8016:	531E	INC.W R14
	-0x8018:	F03E	AND.W #0x000f, R14
	0x801a:	000F	
	-Wait:	0x801c:	401F MOV.W Delay, R15
		0x801e:	000E
		0x8020:	120F PUSH R15
	-L1:	0x8022:	8391 DEC.W >0x0000(SP)
		0x8024:	0000
		0x8026:	23FD JNE >L1
		0x8028:	413F POP.W R15
		0x802a:	3FFF JMP Mainloop
	-Delay:	0x802c:	.word 0x0002

Memory Location Offset

-	0x8000:	4031	MOV.W	#0x0300, SP	
-	0x8002:	0300			
-	0x8004:	40B2	MOV.W	#0x5a80, &Watchdog_Timer	W _{0x8028-0x8022=0x0006} jump -6 bytes or -3 words)
-	0x8006:	5A80			
-	-0x8008:	0120			
-	0x800a:	D0F2	BIS.B	#0x000f, &Port_1_2_P1DIR	JNE 0000 0011 1111 1101
-	0x800c:	000F			
-	0x800e:	0022			
-	0x8010:	430E	CLR.W	R14	
-	Mainloop: 0x8012:	4EC2	MOV.B	R14, &Port_1_2_P1OUT	New PC value
-	0x8014:	0021			
-	0x8016:	531E	INC.W	R14	
-	-0x8018:	F03E	AND.W	#0x000f, R14	
-	-0x801a:	000F			
-	Wait: 0x801c:	401F	MOV.W	Delay, R15	
-	0x801e:	000E			(14) 0x802c-0x801e =
0x000E					
-	0x8020:	120F	PUSH	R15	
-L1:	0x8022:	8391	DEC.W	0x0000 (S)	New PC value
-	0x8024:	0000			
-	0x8026:	23FD	JNE	L1	-0x802c-0x8012=0x001a jump -26 bytes or -13 words)
-	0x8028:	413F	POP.W	R15	
-	0x802a:	3FF3	JMP	Mainloop	JMP 0001 1111 1111 0011
-Delay:	0x802c:	0002	.word	0x0002	

Machine Code in the Memory

	0x8000:	MOV.W #0x0300,SP
	0x8002: 0300	
	0x8004:	MOV.W #0x5a80,&Watchdog_Timer_WDTCTL
	0x8006: 5A80	
	_0x8008: 0120	
	0x800a:	BIS.B #0x000f,&Port_1_2_P1DIR
	0x800c: 000F	
	0x800e: 0022	
	0x8010:	CLR.W R14
Mainloop:	0x8012:	MOV.B R14,&Port_1_2_P1OUT
	0x8014: 0021	
	0x8016:	INC.W R14
	_0x8018:	AND.W #0x000f,R14
	_0x801a: 000F	
Wait:	0x801c:	MOV.W Delay,R15
	0x801e: 000E	
	0x8020:	PUSH R15
L1:	0x8022:	DEC.W 0x0000(SP)
	0x8024: 0000	
	0x8026:	JNE L1
	0x8028:	POP.W R15
	0x802a:	JMP Mainloop
Delay:	0x802c: 0002	.word 0x0002

Machine Code in the Memory

```

-      0x8000:  4031 0100 0000 0011 0001      MOV.W   #0x0300,SP
-
-      0x8002:  0300 0000 0011 0000 0000
-
-      0x8004:  40B2 0100 0000 1011 0010      MOV.W
#0x5a80,&Watchdog_Timer_WDTCTL
-
-      0x8006:  5A80 0101 1010 1000 0000
-
-      -0x8008: 0120 0000 0001 0010 0000
-
-      0x800a:  D0F2 1101 0000 1111 0010      BIS.B   #0x000f,&Port_1_2_P1DIR
-
-      0x800c:  000F 0000 0000 0000 1111
-
-      0x800e:  0022 0000 0000 0010 0010
-
-      0x8010:  430E 0100 0011 0000 1110      CLR.W   R14      ;(MOV.W #0X0000, R14)
-
_Mainloop:0x8012: 4EC2 0100 1110 1100 0010      MOV.B   R14, &Port_1_2_P1OUT
-
-      0x8014:  0021 0000 0000 0010 0001
-
-      0x8016:  531E 0101 0011 0001 1110      INC.W   R14      ;(ADD.W #0X01, R14)
-
-      -0x8018: F03E 1111 0000 0011 1110      AND.W   #0x000f,R14
-
-      -0x801a: 000F 0000 0000 0000 1111
-
_Wait:   0x801c:  401F 0100 0000 0001 1111      MOV.W   Delay,R15
-
-      0x801e:  000E 0000 0000 0000 1110
-
-      0x8020:  120F 0001 0010 0000 1111      PUSH    R15
-
_L1:     0x8022:  8391 1000 0011 1001 0001      DEC.W   0(SP) ;(SUB.W #0X01, 0(SP))
-
-      0x8024:  0000 0000 0000 0000 0000
-
-      0x8026:  23FD 0010 0011 1111 1101      JNE     L1
-
-      0x8028:  413F 0100 0001 0011 1111      POP.W   R15      ;(MOV.W @SP+, R15)
-
-      0x802a:  3FF3 0011 1111 1111 0011      JMP    Mainloop
-
_Delay:  0x802c:  0002 0000 0000 0000 0010      .word   0x0002

```

Practice:

- Disassemble the following MSP430 instructions:

<u>Address</u>	<u>Data</u>	
-0x8010:	4031 0100 0000 0011 0001	-mov.w #0x0600,r1
-0x8012:	0600	
-0x8014:	40B2 0100 0000 1011 0010	-mov.w #0x5a1e,&0x0120
-0x8016:	5A1E	
-0x8018:	0120	
-0x801a:	430E 0100 0011 0000 1110	-mov.w #0,r14
-0x801c:	535E 0101 0011 0101 1110	-add.b #1,r14
-0x801e:	F07E 1111 0000 0111 1110	-and.b #0x0f,r14
-0x8020:	000F	
-0x8022:	1230 0001 0010 0011 0000	-push #0x000e
-0x8024:	000E	
-0x8026:	8391 1000 0011 1001 0001	-sub.w #1, 0(r1)
-0x8028:	0000	
-0x802a:	23FD 0010 0011 1111 1101	-jne 0x8026 (0x802C-3x2)
-0x802c:	413F 0100 0001 0011 1111	-mov.w @r1+,r15 (pop.w r15)
-0x802e:	3FF6 0011 1111 1111 0110	-jmp 0x801c (0x8030-2x10)



SEMESTER: FALL 2020-2021

COURSE CODE & NAME: CSE2001 COMPUTER ARCHITECTURE AND ORGANIZATION

SLOTE: B1+TB1/ B2+TB2

Expected Course Outcomes

- CO1. Differentiate Von Neumann, Harvard, and CISC and RISC architectures. Analyze the performance of machines with different capabilities.
- CO2. Illustrate binary format for numerical and characters. Validate efficient algorithm for arithmetic operations.
- CO3. Construct machine level program for given expression on n-address machine. Analyze and calculate memory traffic for a program execution. Design an efficient data path for an instruction format for a given architecture.
- CO4. Explain the importance of hierarchical memory organization. Able to construct larger memories. Analyze and suggest efficient cache mapping technique and replacement algorithms for given design requirements. Demonstrate hamming code for error detection and correction.
- CO5. Understand the need for an interface. Compare and contrast memory mapping and IO mapping techniques. Describe and Differentiate different modes of data transfer. Appraise the synchronous and asynchronous bus for performance and arbitration.
- CO6. Understand the structure and read write mechanisms for different storage systems. Illustrate and suggest appropriate use of RAID levels. Assess the performance of IO and external storage systems.
- CO7. Classify parallel machine models. Illustrate typical 6-stage pipeline for overlapped execution. Analyze the hazards and solutions.

Assessment Methods, Rubrics and Other Guidelines

Assessment type	Date	Max. Marks	Weightage	Remarks	Course Outcomes
Quiz 1	Almost every week	30 - 40	10	Five minute tests conducted, during the class hours, till CAT – I	CO1 CO2 CO3
Quiz 2	Almost every week	30 - 40	10	Five minute tests conducted,	CO4 CO5

				during the class hours, till CAT – II	
Digital Assignment	Submission before 10-10-2020	10	10	Question and rubrics given separately	CO2 CO3 CO4 CO5
CAT – I	As per the announcement by the University	15	Schedule will be announced by the University	CO1 CO2 CO4	
CAT – II					CO4 CO5
FAT		40		All	

Digital Assignment

Prepare a Latex document on assigned topics (in Computer Architecture and Organization.) Marks will be awarded based on the Rubrics given.

Rubrics for Digital Assignment

Sl. No.	Expected Deliverable	Max. Marks
1	Clarity and coverage of concept.	15%
2	Error free and Plagiarism free.	10%
3	Inclusion of sufficient number of relevant figures and tables.	10%
4	Use of standard notations/ terms in content.	10%
5	Novelty of examples used in the document.	20%
6	Organization of document.	10%
7	Level of content delivery (Abstract or in-depth)	10%
8	Early bird submission (on or before 5th OCT. 2020). Expecting final submission of PDF and Latex documents. .tex	15%

Introduction

Lijo V. P.

Asst. Professor
SCOPE, VIT

Why this subject?

- To understand the functional components, characteristics, performance and interactions of a computer system.
- Need to understand computer architecture in order to structure a program so that it runs more efficiently on a real machine.
- To understand the tradeoff among various components such as CPU clock speed vs. memory size.

Introduction

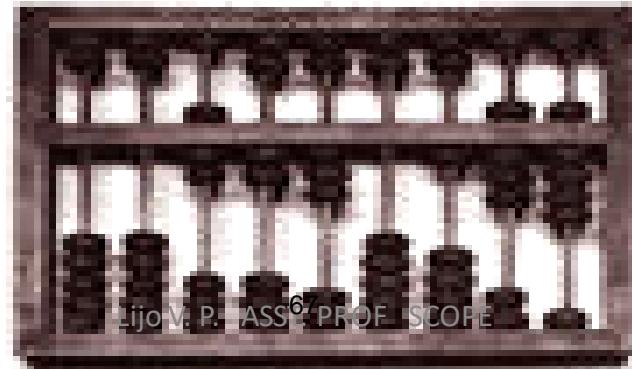
- What is computer?
 - A **computer** is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format.

History of Computers

ABACUS -4th Century B.C.

☞ The abacus, a simple counting aid, may have been invented in Babylonia (now Iraq) in the fourth century B.C.

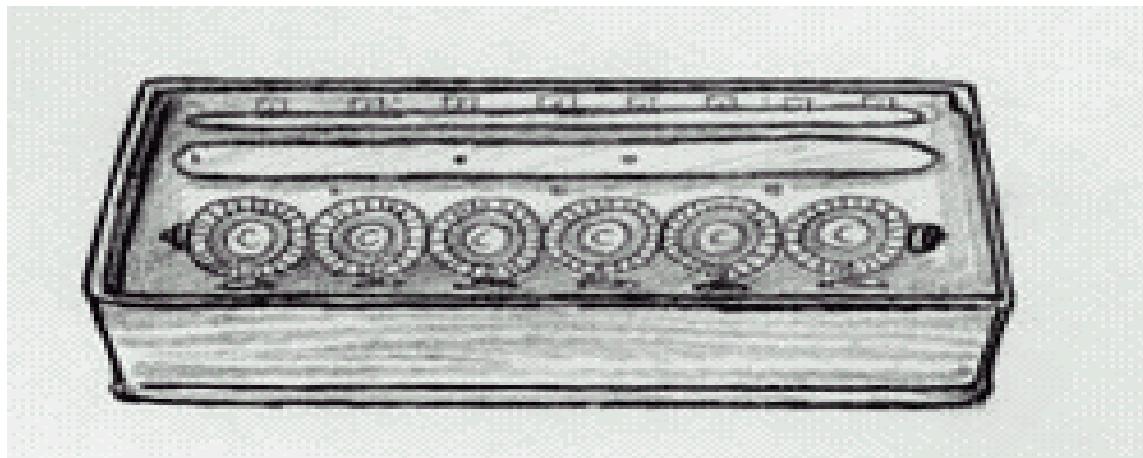
☞ This device allows users to make computations using a system of sliding beads arranged on a rack.



BLAISE PASCAL

(1623 - 1662)

💡 In 1642, the French mathematician and philosopher Blaise Pascal invented a calculating device that would come to be called the "Adding Machine".



CHARLES BABBAGE

(1791 - 1871)

- 💡 Born in 1791, Charles Babbage was an English mathematician and professor.
- 💡 In 1822, he persuaded the British government to finance his design to build a machine that would calculate tables for logarithms.
- 💡 With Charles Babbage's creation of the "Analytical Engine", (1833) computers took the form of a general purpose machine.

HOWARD AIKEN

(1900 - 1973)

 Aiken thought he could create a modern and functioning model of Babbage's Analytical Engine.

 He succeeded in securing a grant of 1 million dollars for his proposed Automatic Sequence Calculator; the Mark I for short. From IBM.

 In 1944, the Mark I was "switched" on. Aiken's colossal machine spanned 51 feet in length and 8 feet in height. 500 meters of wiring were required to connect each component.

- The Mark I *did* transform Babbage's dream into reality and *did* succeed in putting IBM's name on the forefront of the burgeoning computer industry. From 1944 on, modern computers would forever be associated with digital intelligence.

ENIAC 1946



Electronic Numerical Integrator And Computer

Under the leadership of **J. Presper Eckert (1919 - 1995)** and **John W. Mauchly (1907 - 1980)** the team produced a machine that computed at speeds 1,000 times faster than the Mark I was capable of only 2 years earlier.

Using 18,000-19,000 vacuum tubes, 70,000 resistors and 5 million soldered joints this massive instrument required the output of a small power station to operate it.

It could do nuclear physics calculations (in two hours) which it would have taken 100 engineers a year to do by hand.

The system's program could be changed by rewiring a panel.

ENIAC

1946



TRANSISTOR 1948

💡 In the laboratories of Bell Telephone, John Bardeen, Walter Brattain and William Shockley discovered the "transfer resistor"; later labelled the transistor.

💡 Advantages:

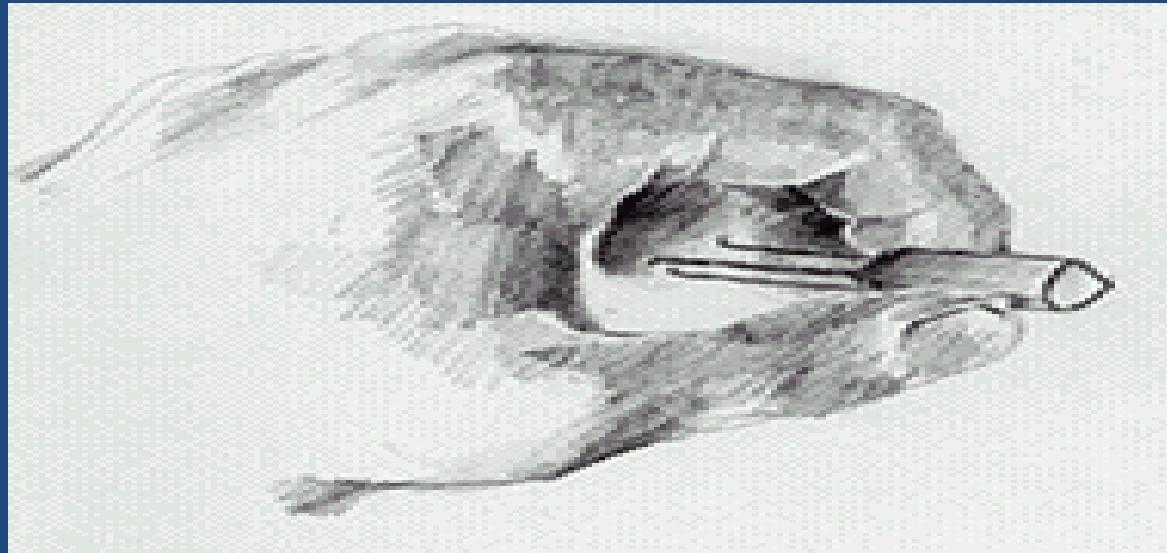
- 💡 increased reliability
- 💡 1/13 size of vacuum tubes
- 💡 consumed 1/20 of the electricity of vacuum tubes
- 💡 were a fraction of the cost

TRANSISTOR

1948



This tiny device had a huge impact on and extensive implications for modern computers. In 1956, the transistor won its creators the Noble Peace Prize for their invention.



ALTAIR 1975



The invention of the transistor made computers smaller, cheaper and more reliable. Therefore, the stage was set for the entrance of the computer into the domestic realm. In 1975, the age of personal computers commenced.



Under the leadership of Ed Roberts the Micro Instrumentation and Telemetry Company (MITS) wanted to design a computer 'kit' for the home hobbyist.

Based on the Intel 8080 processor, capable of controlling 64 kilobytes of memory, the MITS Altair - as the invention was later called - was debuted on the cover of the January edition of *Popular Electronics* magazine.

Presenting the Altair as an unassembled kit kept costs to a minimum. Therefore, the company was able to offer this model for only \$395. Supply could not keep up with demand.

1981

💡 On August 12, 1981 IBM announced its own personal computer.

💡 Using the 16 bit Intel 8088 microprocessor, allowed for increased speed and huge amounts of memory.

💡 Unlike the Altair that was sold as unassembled computer kits, IBM sold its "ready-made" machine through retailers and by qualified salespeople.

IBM (PC)

1981

-  To satisfy consumer appetites and to increase usability, IBM gave prototype IBM PCs to a number of major software companies.
-  For the first time, small companies and individuals who never would have imagined owning a "personal" computer were now opened to the computer world.

Computer Generations

FIRST GENERATION

(1945-1956)



💡 First generation computers were characterized by the fact that operating instructions were made-to-order for the specific task for which the computer was to be used.

💡 Each computer had a different binary-coded program called a machine language that told it how to operate. This made the computer difficult to program and limited its versatility and speed.

💡 Other distinctive features of first generation computers were the use of vacuum tubes (responsible for their breathtaking size) and magnetic drums for data storage.

First Generations

- Vacuum Tubes
- Magnetic Drum
- 4,000 bits
- Hard Wire Programs in computers
- IBM 650, Univac I
- ENIAC

Vacuum Tubes



SECOND GENERATION (1956-1963)



Throughout the early 1960's, there were a number of commercially successful second generation computers used in

- business,
- universities, and
- government from companies such as Burroughs, Control Data, Honeywell, IBM, Sperry-Rand, and others.

These second generation computers were also of solid state design, and contained transistors in place of vacuum tubes.

THIRD GENERATION

(1965-1971)



Though transistors were clearly an improvement over the vacuum tube, they still generated a great deal of heat, which damaged the computer's sensitive internal parts.

The quartz rock eliminated this problem. Jack Kilby, an engineer with Texas Instruments, developed the integrated circuit (IC) in 1958.

The IC combined three electronic components onto a small silicon disc, which was made from quartz.

Scientists later managed to fit even more components on a single chip, called a semiconductor.

As a result, computers became ever smaller as more components were squeezed onto the chip. Another third-generation development included the use of an operating system that allowed machines to run many different programs at once with a central program that monitored and coordinated the computer's memory.

FOURTH GENERATION

(1971-Present)

-  In 1981, IBM introduced its personal computer (PC) for use in the home, office and schools.
-  The 1980's saw an expansion in computer use in all three arenas as clones of the IBM PC made the personal computer even more affordable.
-  The number of personal computers in use more than doubled from 2 million in 1981 to 5.5 million in 1982.

FOURTH GENERATION

(1971-Present)

USB icon **Ten years later, 65 million PCs were being used. Computers continued their trend toward a smaller size, working their way down from desktop to laptop computers (which could fit inside a briefcase) to palmtop (able to fit inside a pocket).**

USB icon **In direct competition with IBM's PC was Apple's Macintosh line, introduced in 1984. Notable for its user-friendly design, the Macintosh offered an operating system that allowed users to move screen icons instead of typing instructions**

FIFTH GENERATION

Many advances in the science of computer design and technology are coming together to enable the creation of fifth-generation computers.

Two such engineering advances are **parallel processing**, which replaces von Neumann's single central processing unit design with a system harnessing the power of many CPUs to work as one.

Another advance is **superconductor technology**, which allows the flow of electricity with little or no resistance, greatly improving the speed of information flow.

FIFTH GENERATION

(Future)

 Computers today have some attributes of fifth generation computers.

 For example, expert systems assist doctors in making diagnoses by applying the problem-solving steps a doctor might use in assessing a patient's needs.

 It will take several more years of development before expert systems are in widespread use.

Introduction and Overview of Computer Architecture and Organization

WHY STUDY COMPUTER ORGANIZATION AND ARCHITECTURE?

- The computer lies at the heart of computing.
- Without it most of the computing disciplines today would be a branch of theoretical mathematics.
- To be a professional in any field of computing today, one should not regard the computer as just a black box that executes programs by magic.
- All students of computing should acquire some understanding of a computer system's functional components, their characteristics, their performance, and their interactions.

WHY STUDY COMPUTER ORGANIZATION AND ARCHITECTURE?

- Students need to understand computer architecture in order to structure a program so that it runs more efficiently on a real machine.
- In selecting a system to use, they should be able to understand the tradeoff among various components, such as CPU clock speed vs. memory size.

Computer

- Takes Input
 - Data
- Processes it according to stored instructions
 - Instructions: Software, Programs
- Produces results as output
 - Information (numbers, words, sounds, images)

Types of Computers

- Computer
 - Special Purpose (Embedded Systems)
 - Oven
 - Television
 - Mobile
 - General Purpose (User-programmable)
 - Personal Computers
 - Mainframes or enterprise systems
 - Workstations
 - Notebook
 - Supercomputers

Computer Architecture

- **Computer architecture** refers to
 - those attributes of a system visible to a programmer
 - those attributes that have a direct impact on the logical execution of a program.
- Examples
 - Instruction set
 - The number of bits used to represent various data types (e.g., numbers, characters)
 - I/O mechanisms, and techniques for addressing memory.
- It is an architectural design issue whether a computer will have a multiply instruction.

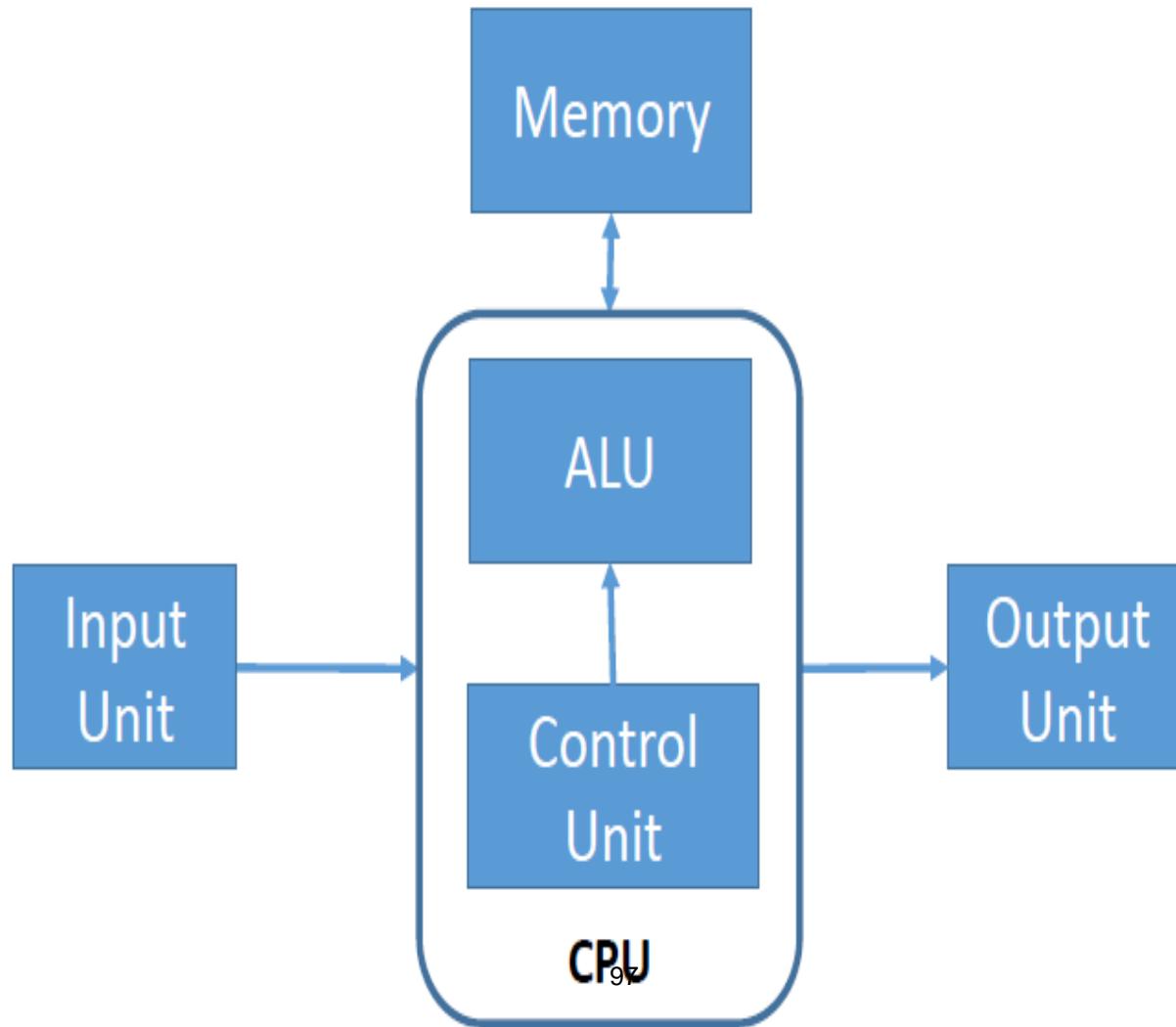
Computer Organization

- **Computer organization** refers to the operational units and their interconnections that realize the architectural specifications.
- Organizational attributes include those hardware details transparent to the programmer
- Examples
 - Control signals;
 - Interfaces between the computer and peripherals;
 - The memory technology used.
- It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system.

Computer Architecture and Organization

- A particular architecture may span many years and encompass a number of different computer models, its organization changing with changing technology.
- IBM System/370 architecture - This architecture was first introduced in 1970 and included a number of models.
- The customer with modest requirements could buy a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model

Functional Components of a Computer



Input Unit

Mouse



Keyboard



Joystick



Light Pen



Touch Pad



Microphone



Track Ball



Scanner



Digital Camera



Memory – Physical Device to store programs or data

- This memory is of two fundamental types:
 - Main memory (Primary Memory)
 - Volatile – loses information when power is removed.
 - Main Storage
 - Secondary memory.
 - Non-volatile
 - Secondary or Mass Storage

Main Memory

- Closely connected to the processor.
- Stored data are quickly and easily changed.
- Holds the programs and data that the processor is actively working with.
- Interacts with the processor millions of times per second.
- Needs constant electric power to keep its information.
- Fast
- Expensive
- Low Capacity
- Works directly with the processor

Main Memory

ROM

RAM



Secondary Memory

- Connected to main memory through the bus and a controller.
- Stored data are easily changed, but changes are slow compared to main memory.
- Used for long-term storage of programs and data.
- Before data and programs can be used, they must be copied from secondary memory into main memory.
- Does not need electric power to keep its information.
- Slow
- Cheap
- Large Capacity
- Not connected directly to the processor

Secondary Memory



Output Devices



Webcam



Webcam



Microphone



Monitor



Speakers

Printer



Printer



Headphone



CPU – Central Processing Unit



Interconnection of Components

- The functional components of a computer need to communicate with each other to perform a task.
- Therefore, computer is a network of components.
- The collection of paths connecting the various components is called the *interconnection structure*. The design of this structure will depend on the exchanges that must be made among modules.

Input and output of a Memory Component

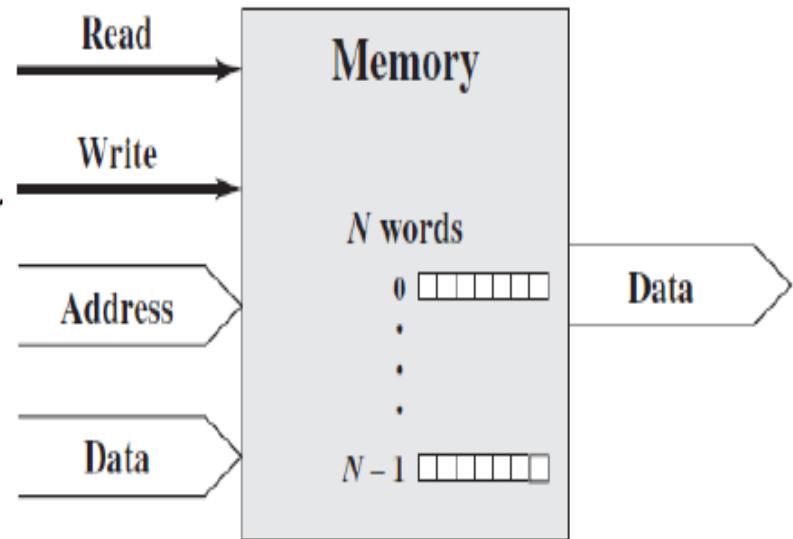
- Two Operations

- Read

- Input
 - Enable Read Control signal
 - Provide the address on address bus from where data to read
 - Output
 - Data from the given address

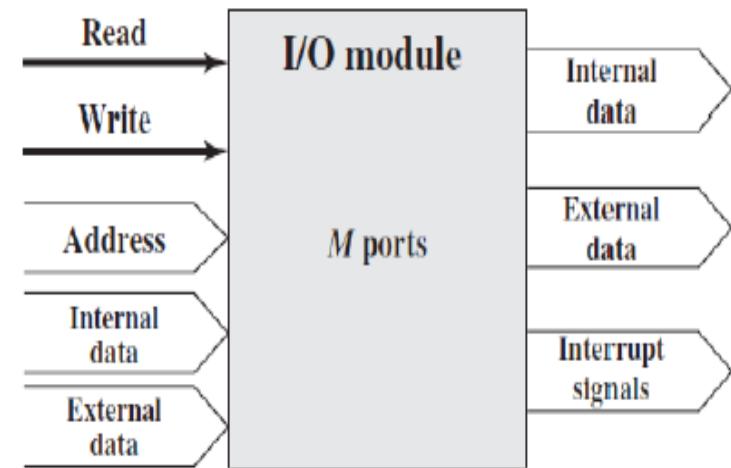
- Write

- Input
 - Enable Write control signal
 - Provide the address on address bus where the data to be written
 - Provide the data to be written on data bus
 - Output



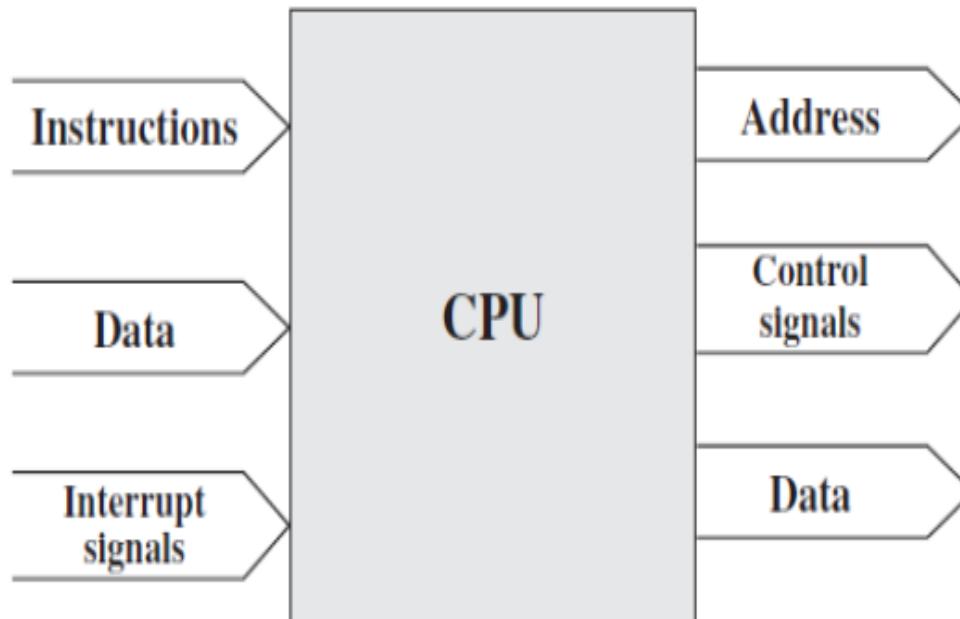
Input and Output of I/O Module

- I/O Module may control more than one external device.
- Each of the interface to an external device is referred as a *port* and given each a unique address (e.g., 0, 1, . . . , M-1)
- Finally, an I/O module may be able to send interrupt signals to the processor.



Input and Output of Processor component

- The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system.
- It also receives interrupt signals.



Interconnections

- The interconnection structure must support the following types of transfers:
 - Memory \leftrightarrow Processor
 - I/O \leftrightarrow Processor
 - I/O \leftrightarrow Memory (DMA)

Bus Interconnection



- It is a pathway connecting two or more devices.
- A bus consists of multiple communication pathways, or lines.
- Each line is capable of transmitting signals representing binary 1 and binary 0.
- Transmitting a word across a single line (bit by bit serially)
- Several lines of a bus can be used to transmit binary digits simultaneously (in parallel).
 - For example, an 8-bit unit of data can be transmitted over eight bus lines.
- A bus that connects major computer components (processor, memory, I/O) is called a *system bus*

Bus

- The lines can be classified into three functional groups: **data, address, and control lines**
- **Data Lines – Data bus** – provide a path for moving data among system modules.
 - The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the *width* of the data bus.
 - If the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.

Address Bus

- **Address Lines – Address Bus** – Used to designate the source or destination of the data on the data bus.
- For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines.
- Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module.
- For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module.

Control Bus

- The **control lines** are used to control the access to and the use of the data and address lines.
- Because the data and address lines are shared by all components, there must be a means of controlling their use.
- Control signals transmit both command and timing information among system modules.
- Timing signals indicate the validity of data and address information.
- Command signals specify operations to be performed.

The operation of the bus

- If one module wishes to send data to another, it must do two things:
 - Obtain the use of the bus, and
 - Transfer data via the bus.
- If one module wishes to request data from another module, it must
 - Obtain the use of the bus, and
 - Transfer a request to the other module over the appropriate control and address lines.
 - It must then wait for that second module to send the data.

I/O Bus and Memory Bus

- *Functions of Buses*
 - *MEMORY BUS* is for information transfers between CPU and the Main Memory
 - *I/O BUS* is for information transfers between CPU and I/O devices through their I/O interface
- Many computers use a common single bus system for both memory and I/O interface units
 - separate control lines for each function
 - common address and data lines for both functions
- Some computer systems use two separate buses,
 - one to communicate with memory and the other with I/O interfaces

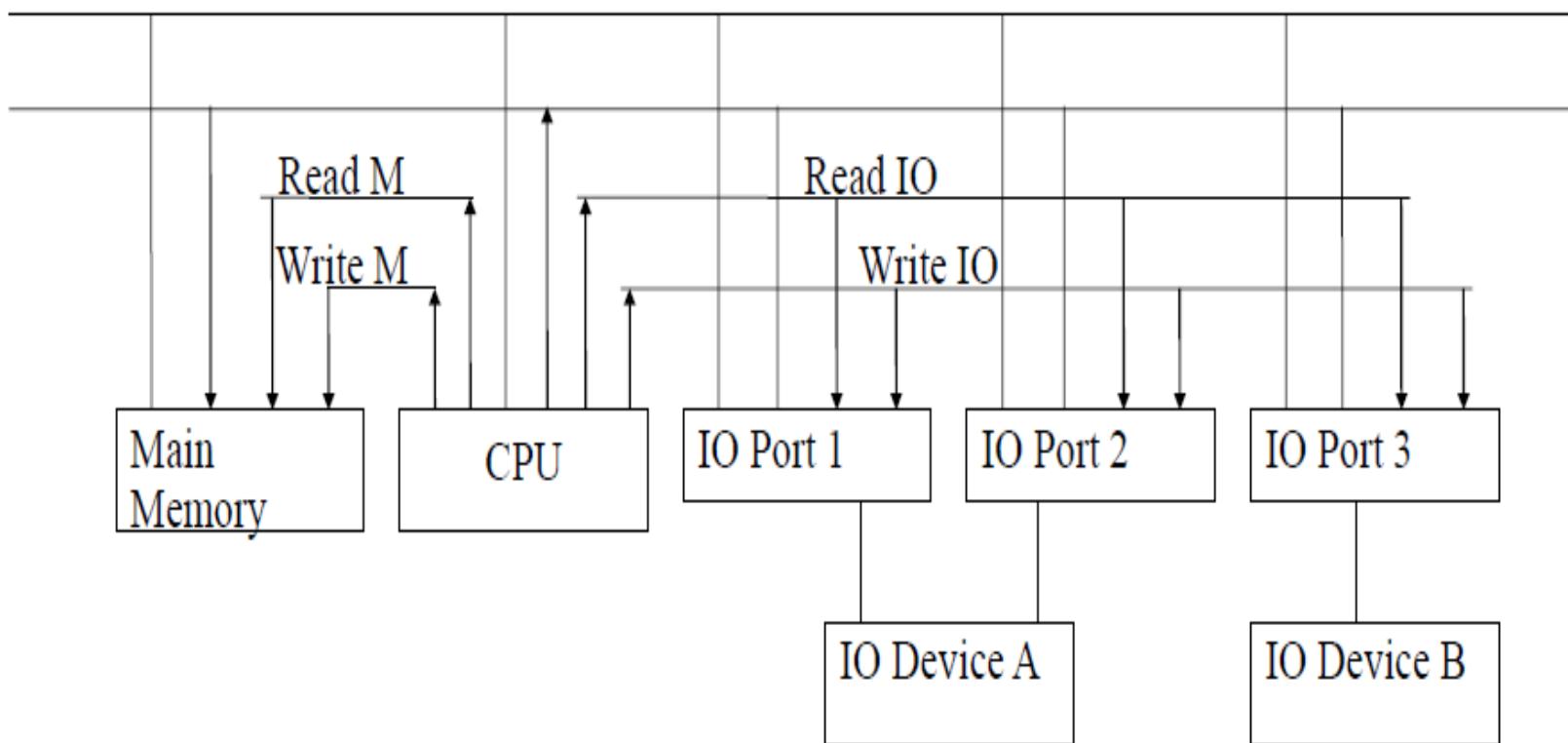
Isolated I/O – I/O Mapped I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions
- When CPU fetches and decodes the opcode of an I/O instruction, it places the address into the common address lines.
- Also enables read/write control lines => the address in the address lines is for interface register and not for a memory word.

Isolated I/O

Data

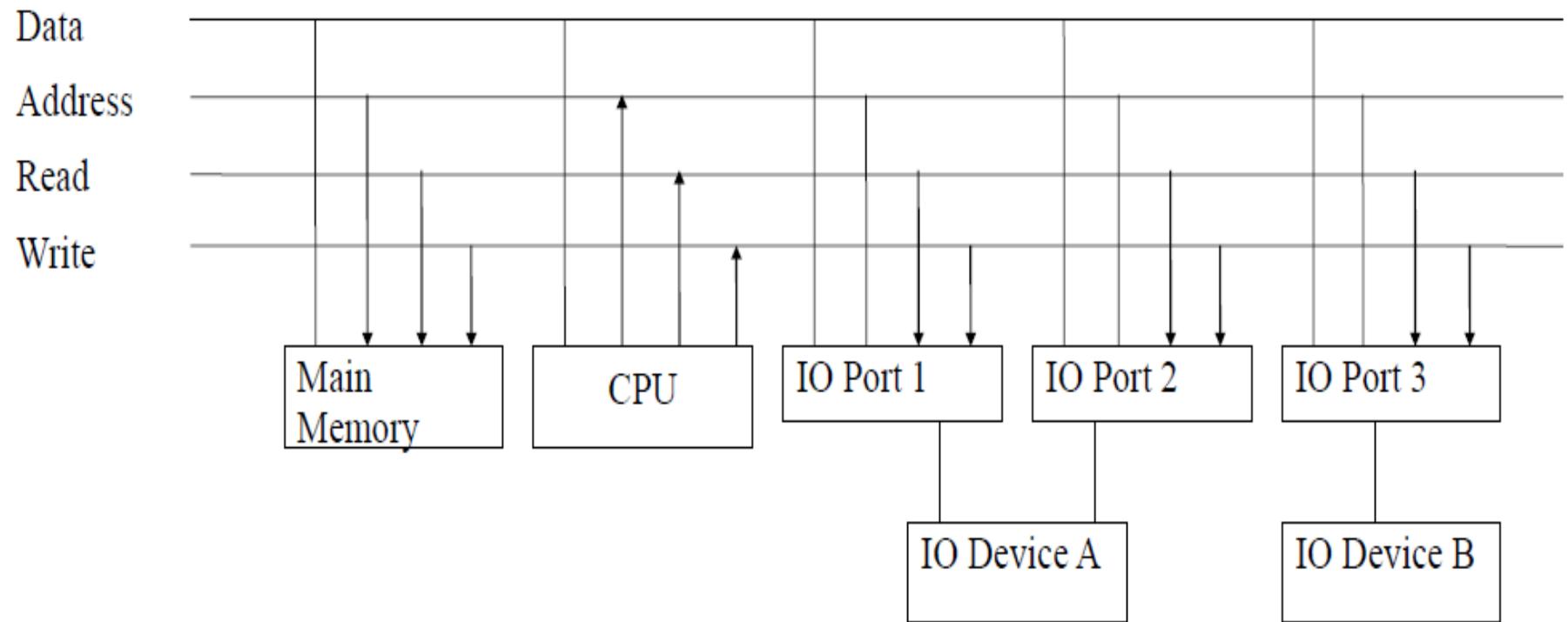
Address



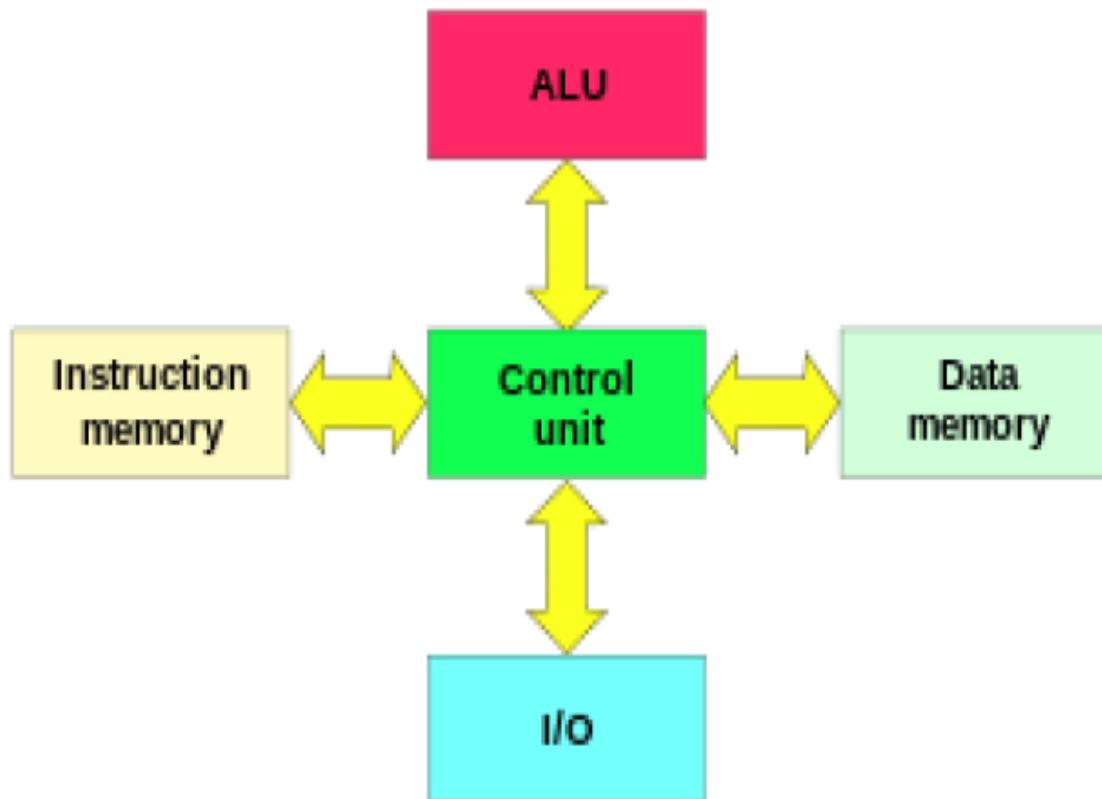
Memory Mapped I/O

- A single set of read/write control lines
- No distinction between memory and I/O transfer
- Memory and I/O addresses share the common address space
 - reduces memory address range available
 - Memory addresses xffff0000 and above are used for I/O devices
- No specific input or output instruction
- The same memory reference instructions can be used for I/O transfers
- when the bus sees certain addresses, it knows they are not memory addresses, but are addresses for accessing I/O devices.

Memory Mapped I/O



Harvard Architecture



Harvard architecture

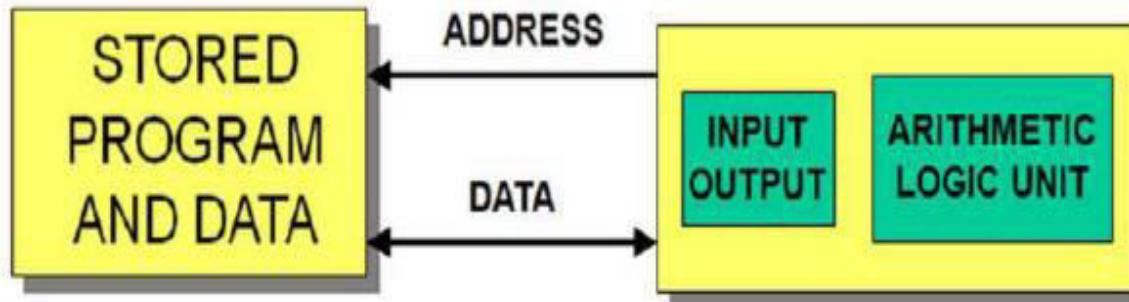
- The **Harvard architecture** is a computer architecture with physically separate storage
- These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data.
- Programs needed to be loaded by an operator; the processor could not initialize itself.

Harvard Architecture

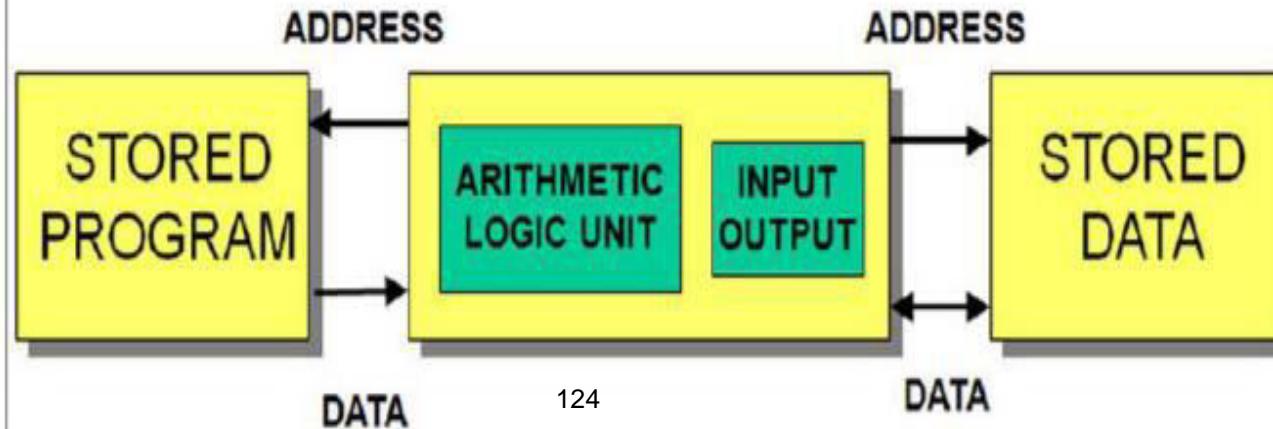
- In a Harvard architecture, there is no need to make the two memories share characteristics.
- In particular, the word width, timing, implementation technology, and memory address structure can differ.
- Instructions – read-only memory
- Data memory – read-write memory.
- In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.
- Ex: Mark1

Von-Neumann vs Harvard Architecture

Von Neuman



Havard



Von-Neumann vs Harvard Architecture

Von-Neumann Architecture	Harvard Architecture
The data and program are stored in the same memory	The data and program memory are separate. Both memories can use different sizes.
The code is executed serially and takes more clock cycles.	The code is executed in parallel
The programs can be optimized in lesser size	The programs tend to grow big in size
Parallel access to data and program is not possible.	Two memories with two buses allow parallel access to data access and instructions
One bus is simpler for the control unit design	Control unit for 2 buses is more complicated and more expensive
Error in program can rewrite instruction and crash program execution	Program can't write itself

Modified Harvard Architecture

- A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses.
- The most common modification includes separate instruction and data caches backed by a common address space.
- While the CPU executes from cache, it acts as a pure Harvard machine.
- When accessing backing memory, it acts like a von Neumann machine
- Ex: x86 processors

References

- **Text Book** – William Stallings, “Computer Organization and Architecture, Designing for performance”, 8th edition, Prentice Hall.
- Internet Sources.

Introduction

Lijo V. P

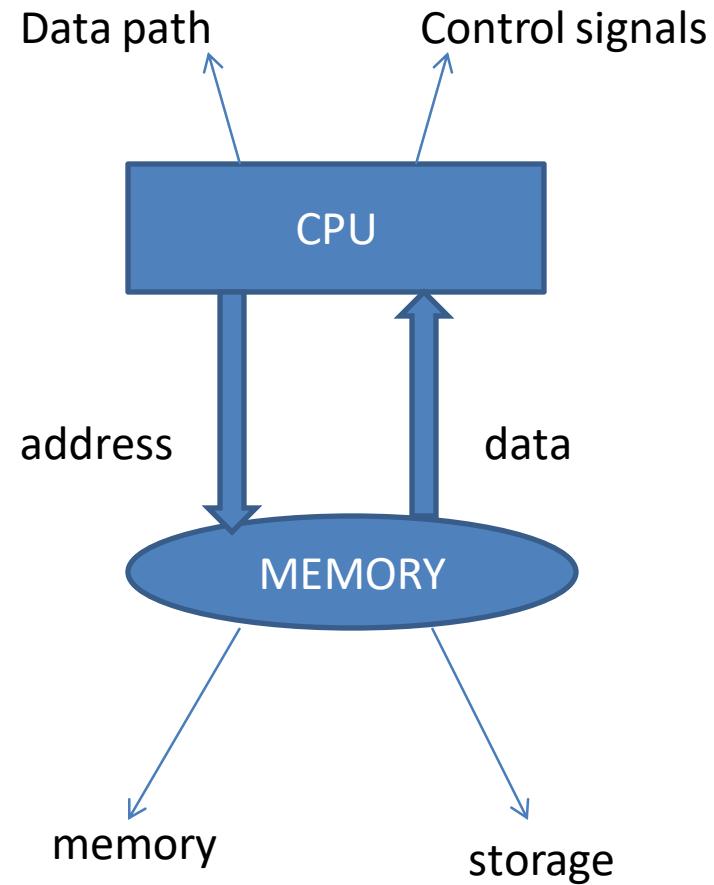
Asst. Professor
SCOPE, VIT

Fundamental Aspects

- Differentiate Computer Organization and Architecture.
- C.O:-
 - User's point of view
 - Study from s/w point of view
 - Eg. Car, How can be used? Drive?
- C.A:-
 - Designer's point of view
 - Study from h/w point of view
 - Eg. Car, How can be implemented? Repair?

Root of computing

- Started with numbers.
- Processing the numbers.
- Any computing system,
 - Processor- process data
 - Memory- store data
 - I/O- user interacts with system
 - (extended memory for real data)

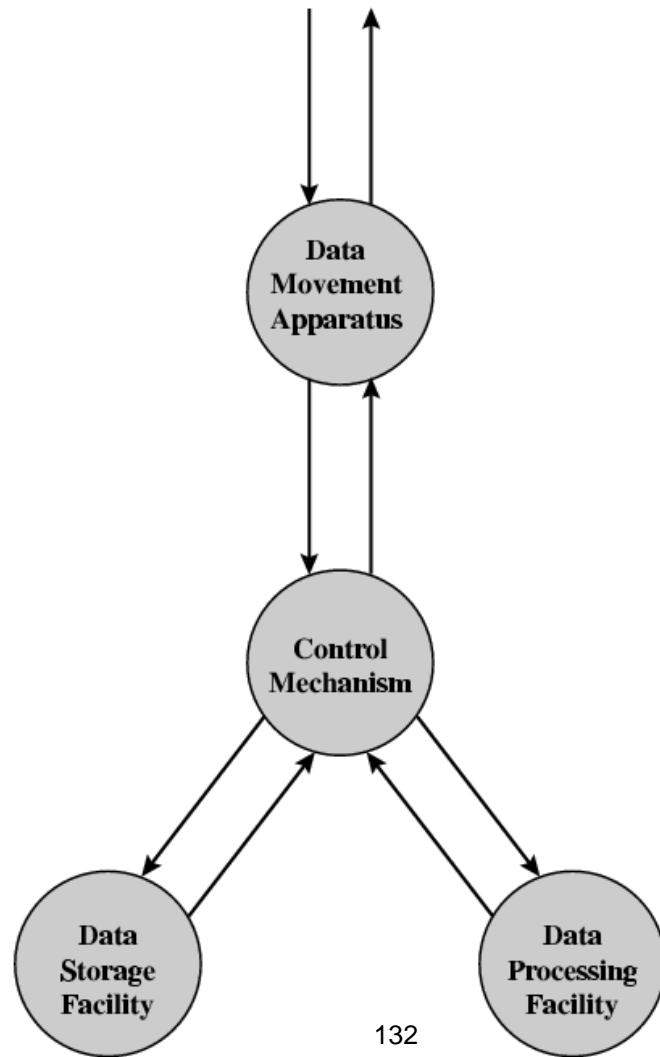


Structure and Function

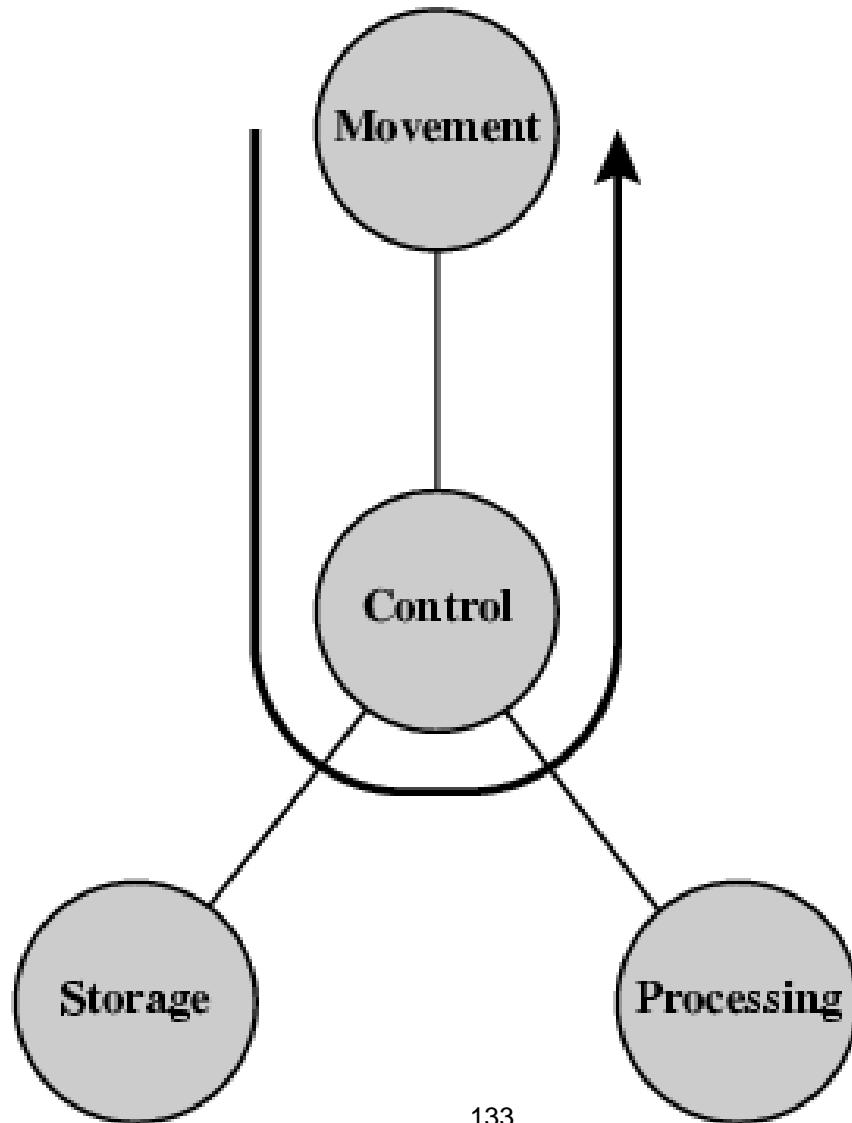
- **Structure**:- The way in which the components are interrelated.
 - CPU:- Registers, ALU, Control unit, Buses
 - Memory
 - I/O
- **Function**:- Function is the operation of individual components as part of the structure
 - Data processing
 - Data storage
 - Data movement
 - Control

Functional view

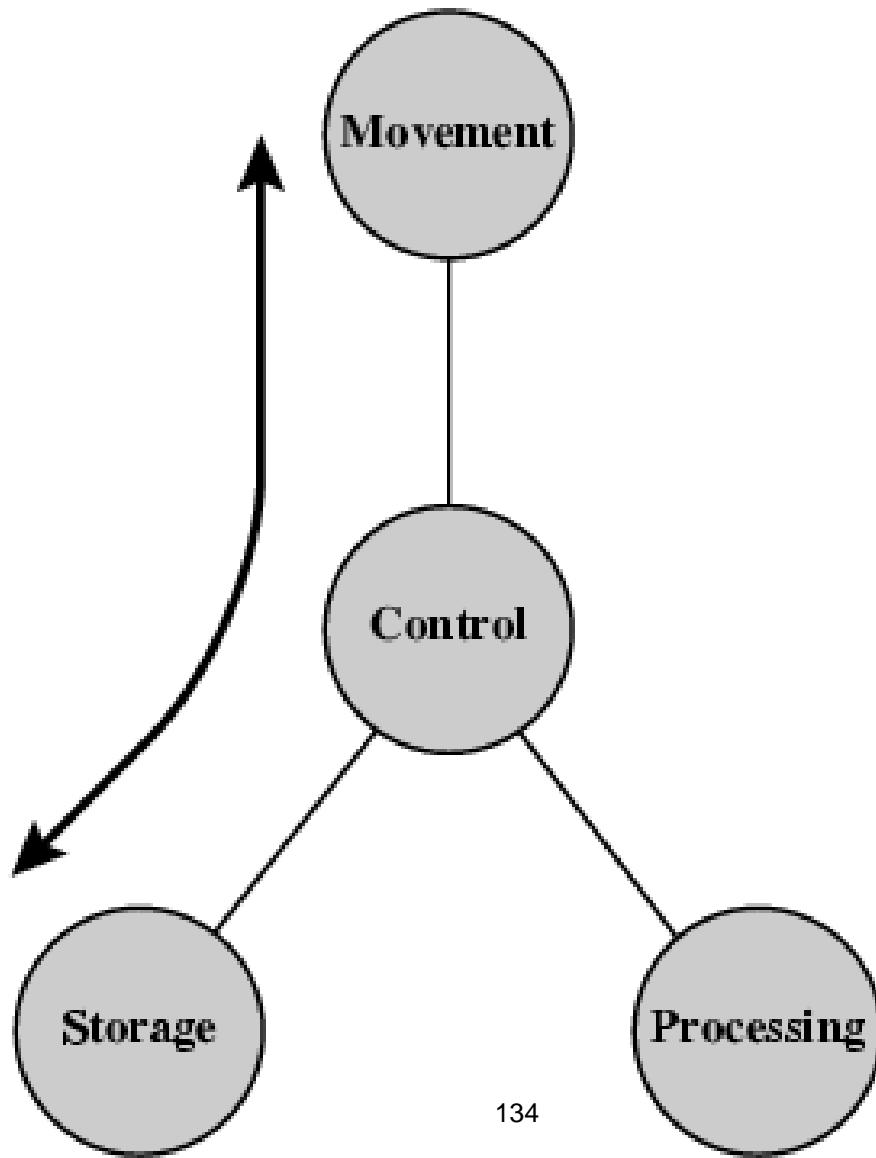
Operating Environment
(source and destination of data)



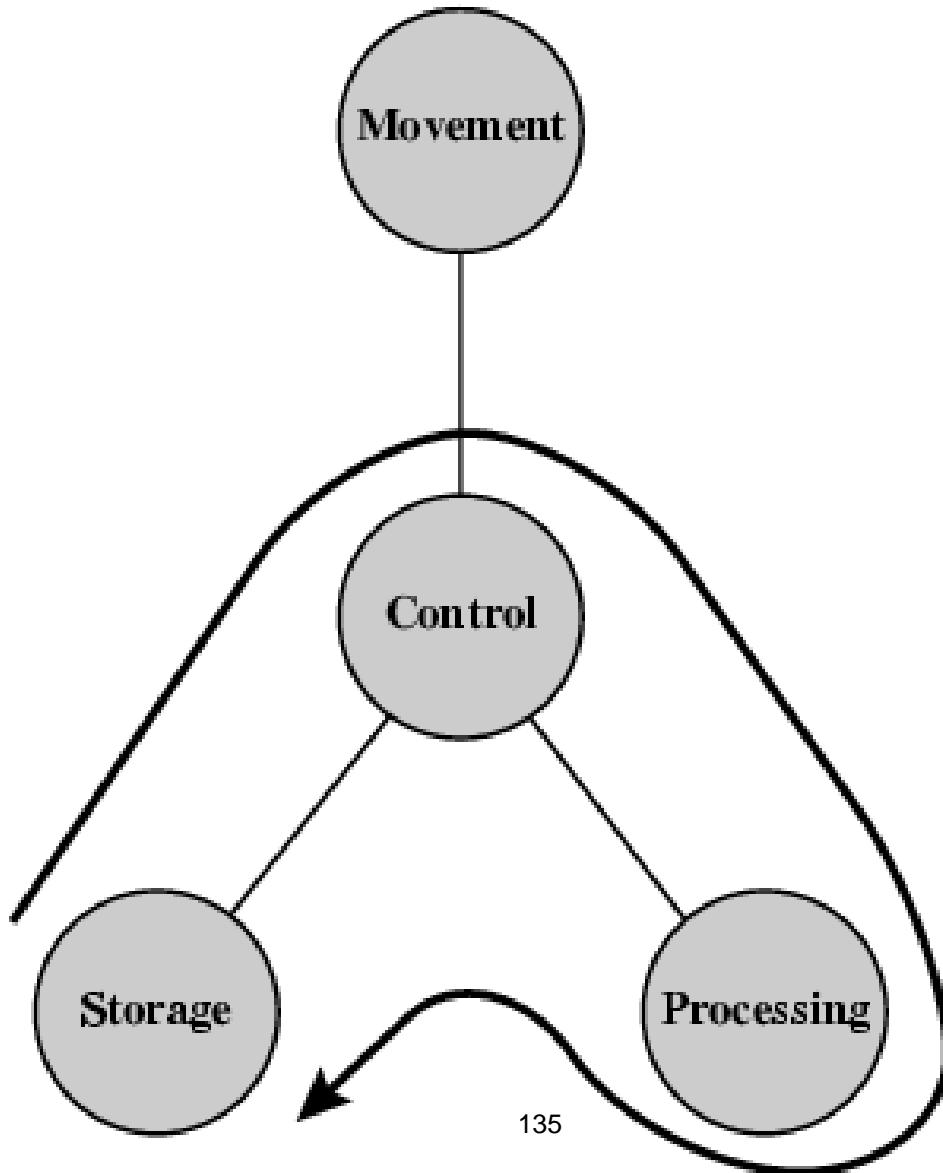
Operations (1) Data movement



Operations (2) Storage

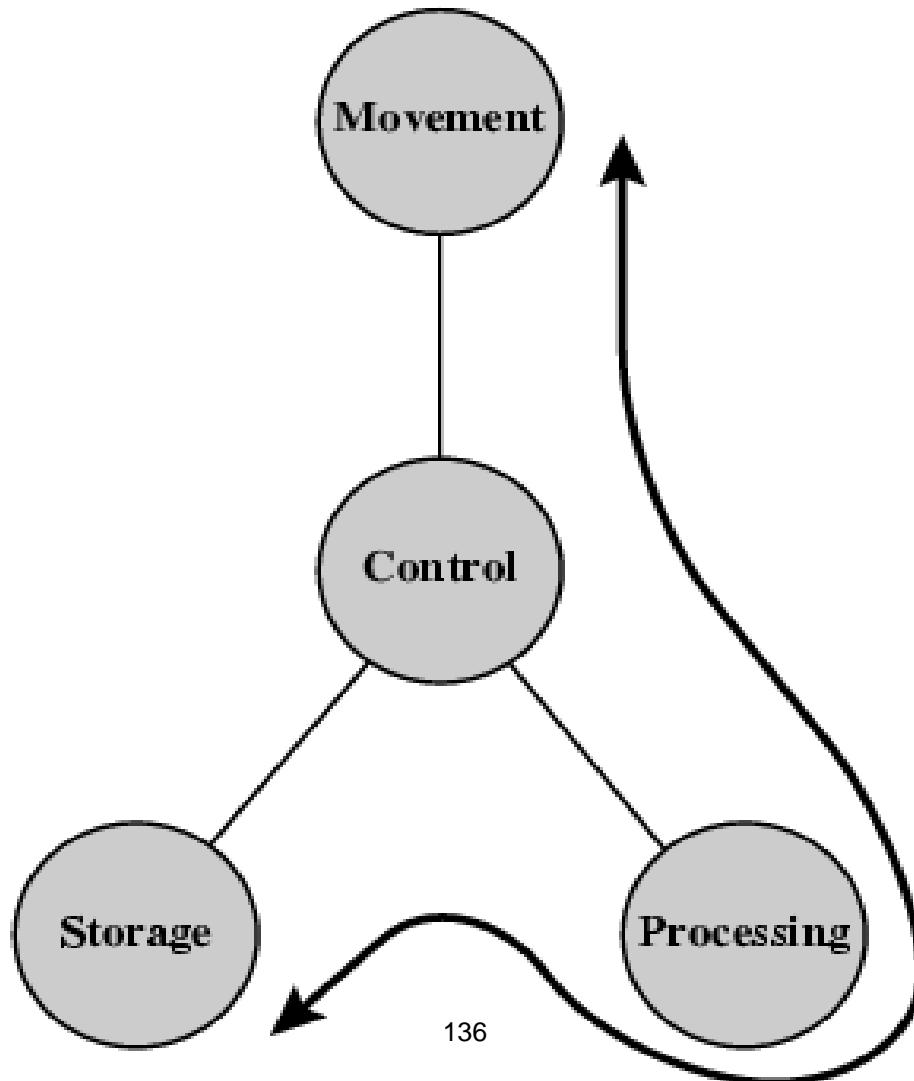


Operation (3) Processing from/to

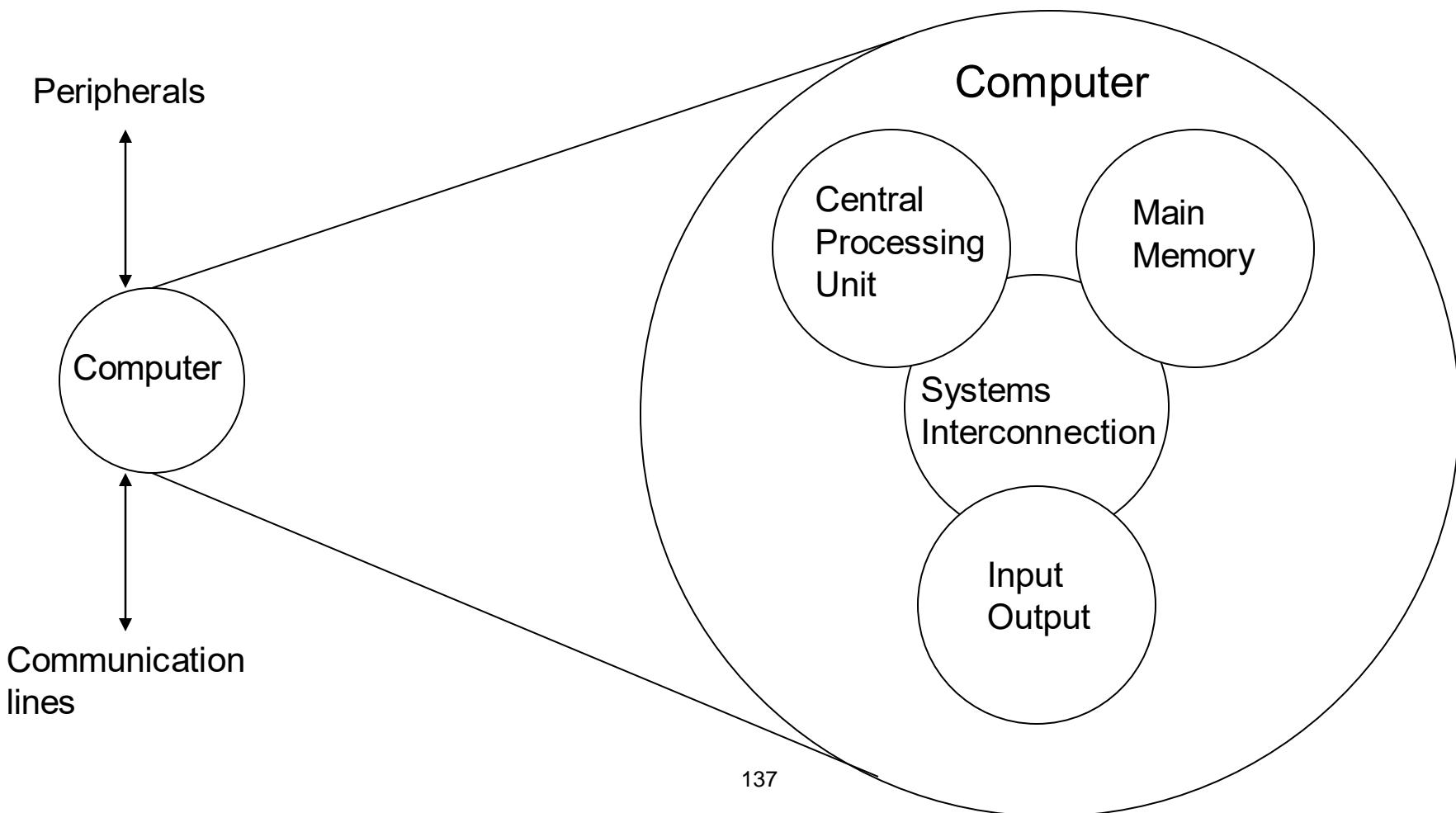


Operation (4)

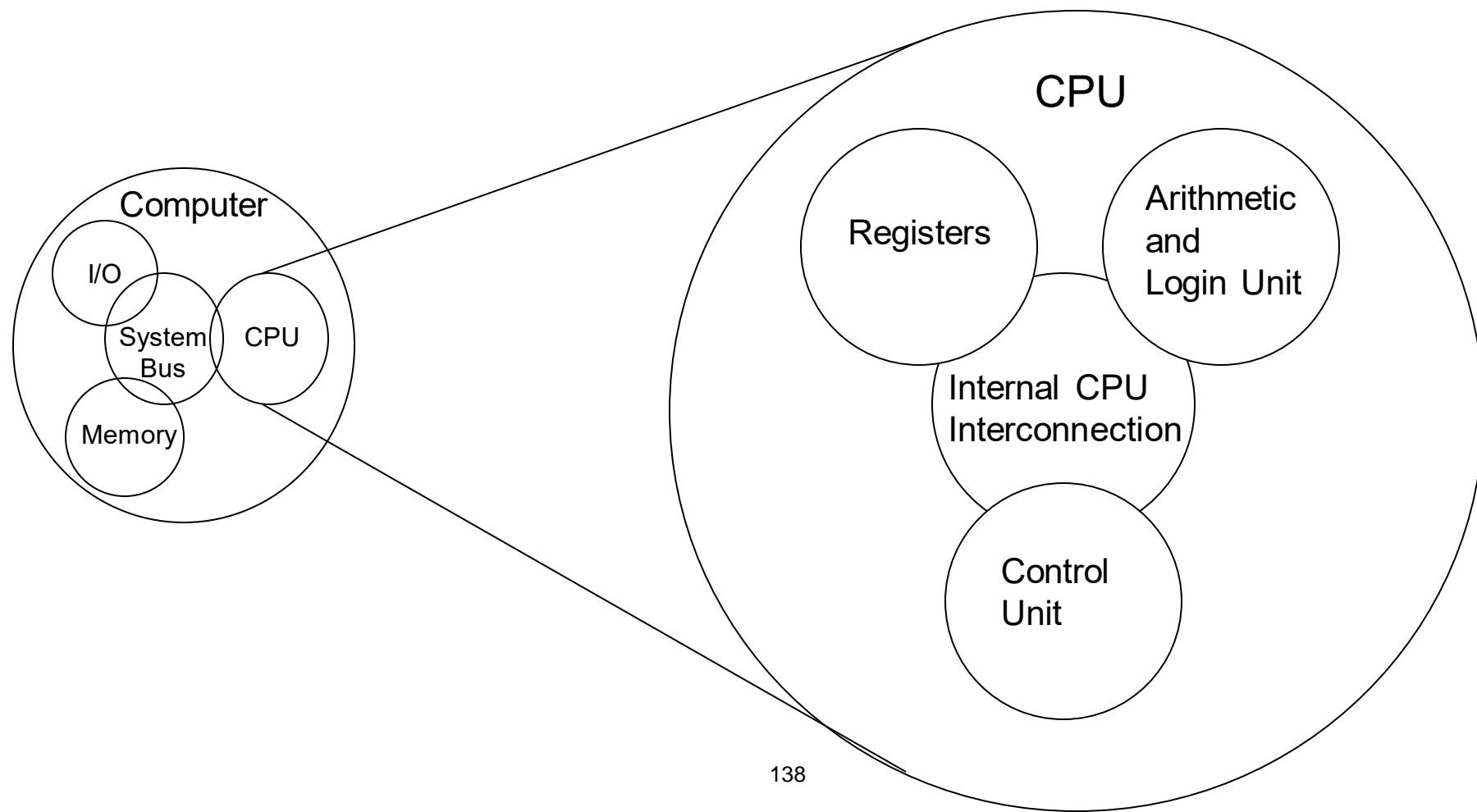
Processing from storage to I/O



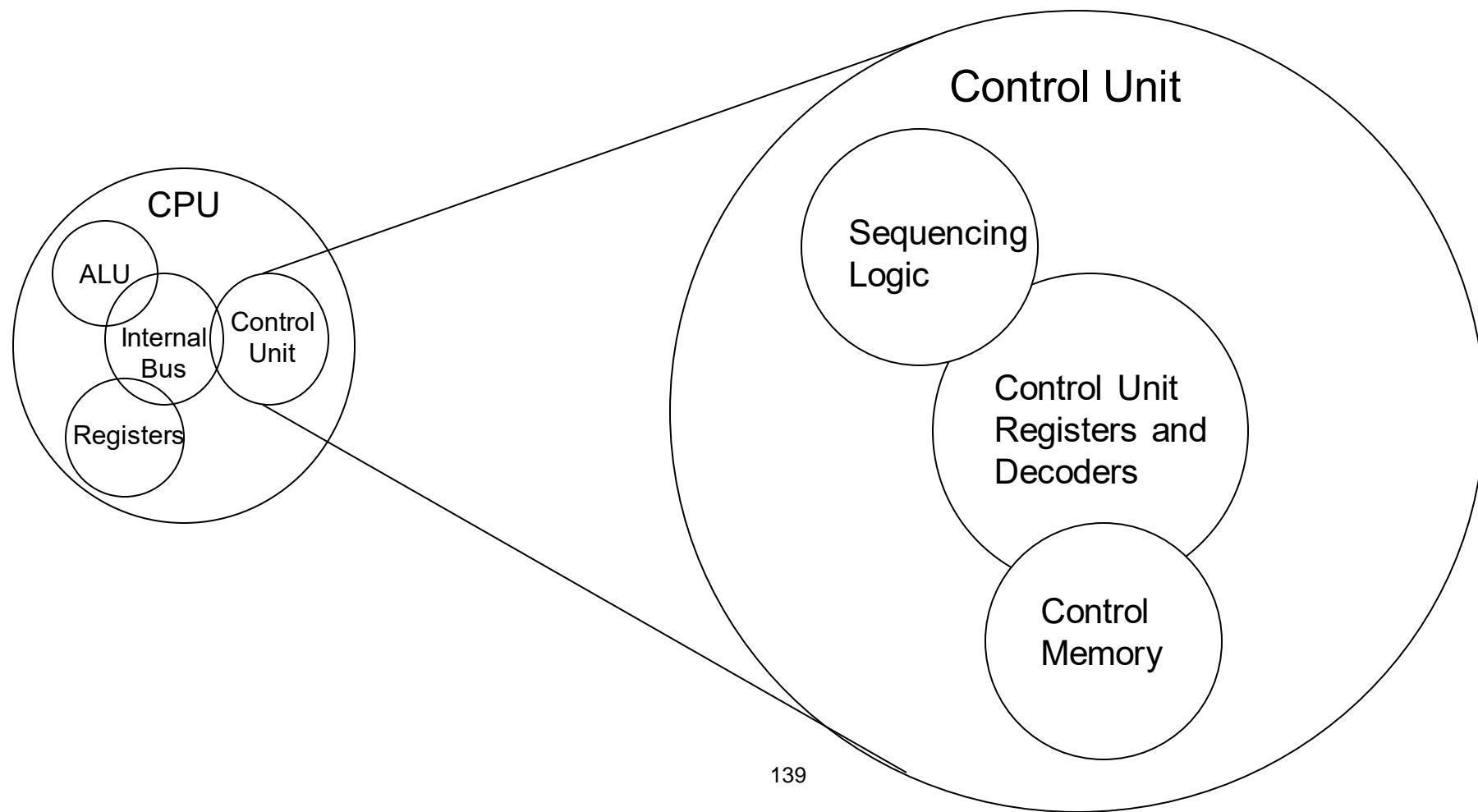
Structure - Top Level



Structure - The CPU

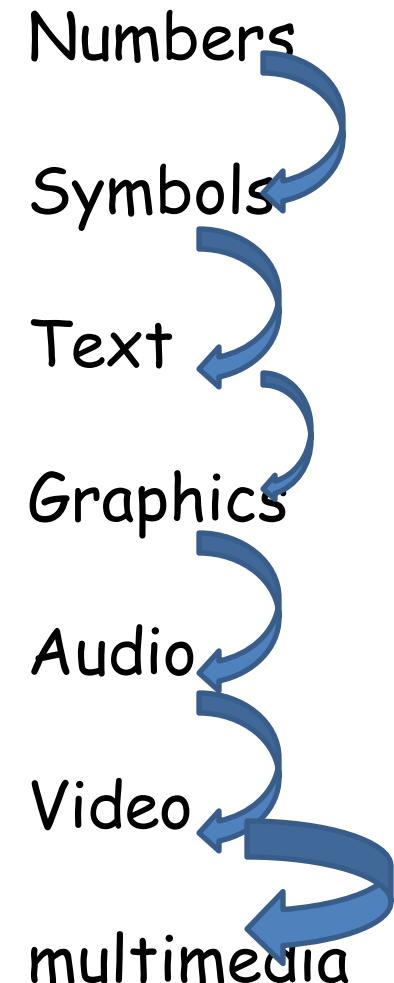


Structure - The Control Unit



Computing and Communicating

- Computing
 - Numbers → Arithmetic processing
 - Symbols → Logic processing
 - ALU → core of CPU.
- Communication
 - Line communication
 - Voice communication
 - Voice + coded data



H/W and S/W

- H/W:- core of the system, hard to understand.
- S/W:- soften the hard aspects of the system.
- LLL
- ALL
- HLL
- Translator:- Compiler, Interpreter, Assembler

? What is device driver?

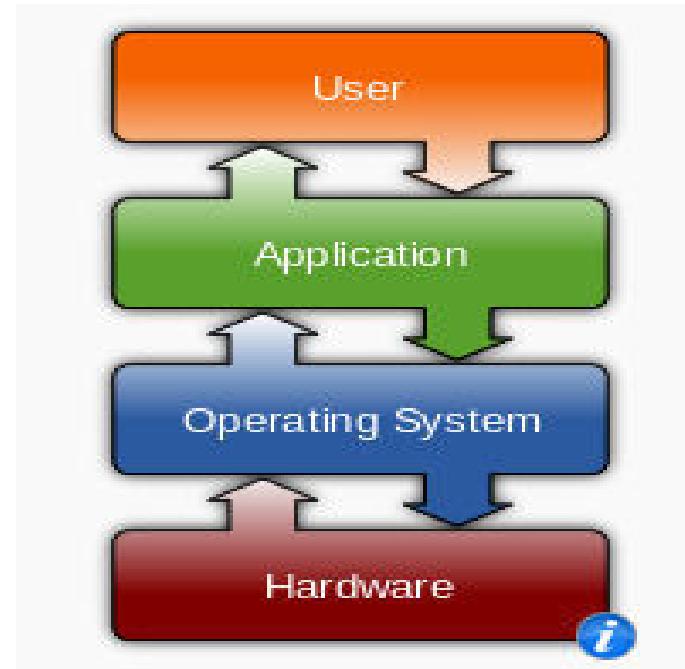
? Differentiate System S/W and Application S/W.

? Example for each translators.

Operating System

- Manages computer hardware resources.

- Provide platform for Application software.
- It's a system software.



- Interface between application and the hardware.
- ? What are the popular modern operating systems?
- ? Is single processor with multiple OS possible?

IAS Machine

LIJO V P
SCOPE
VIT, VELLORE

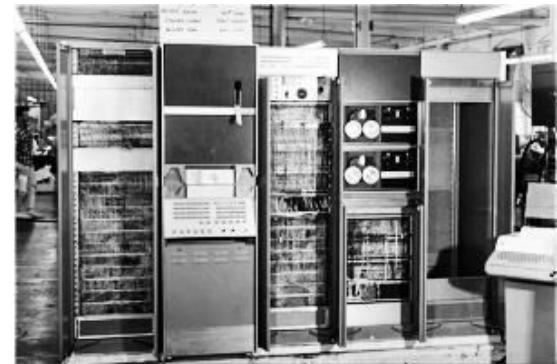
The Von Neumann Architecture

1945: John von Neumann(Princeton university)

- wrote a report on the **stored program** concept, known as the *First Draft of a Report on EDVAC*
- The basic structure proposed in the draft became known as the “von Neumann machine”.
 - a memory, containing instructions and data
 - a processing unit, for performing arithmetic and logical operations
 - a control unit, for interpreting instructions

The Von Neumann Architecture

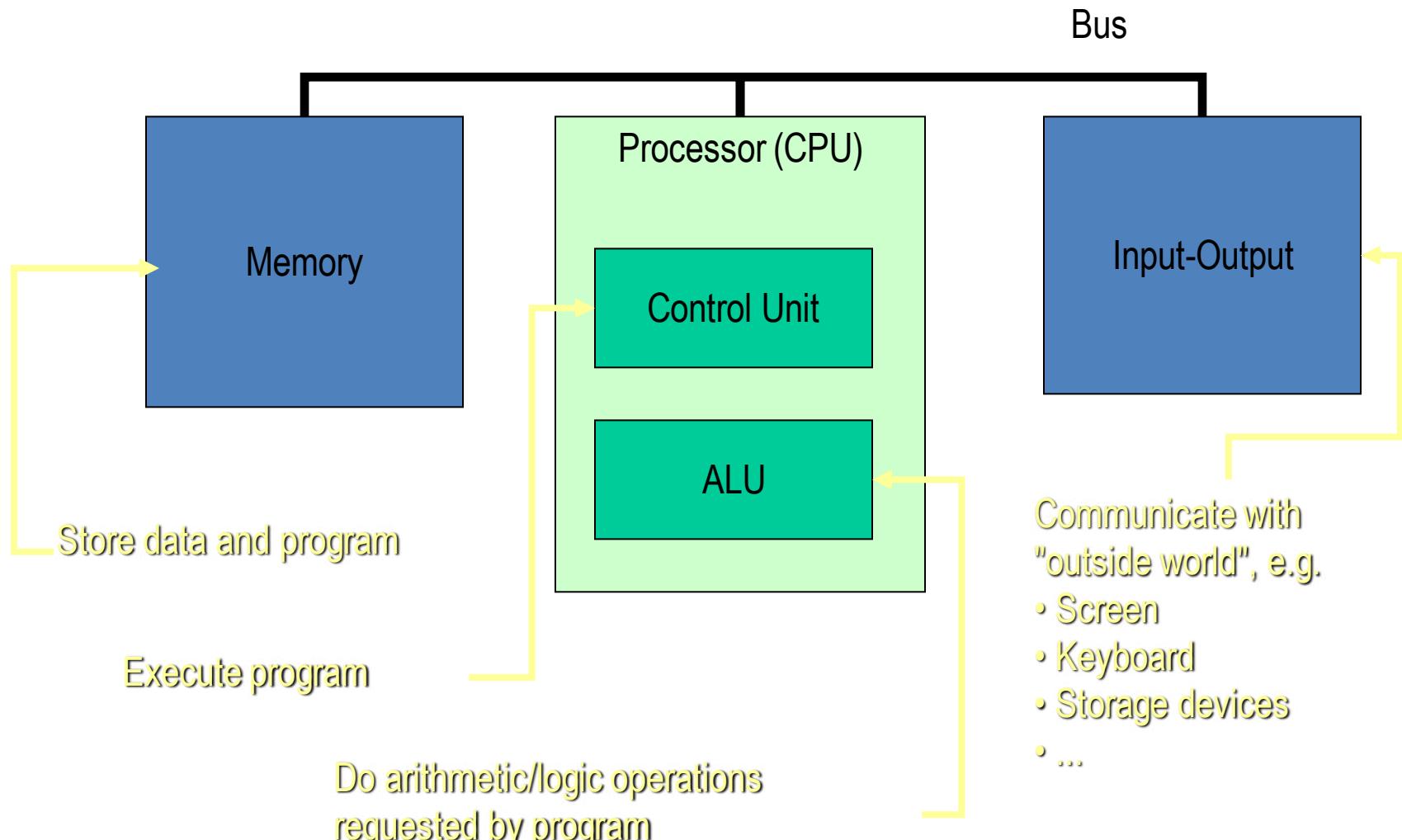
- All computers more or less based on the same basic design, the Von Neumann Architecture!



The Von Neumann Architecture

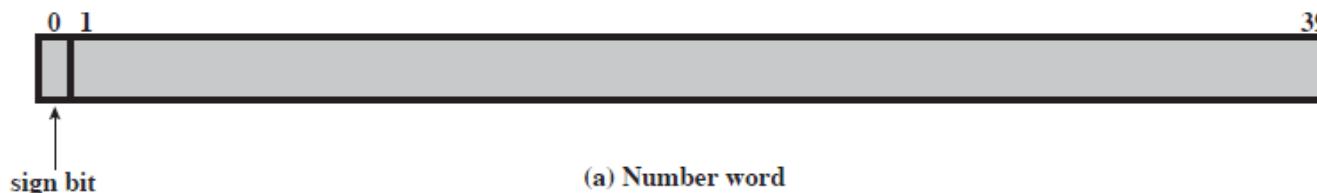
- Model for designing and building computers, based on the following three characteristics:
 - 1) The computer consists of four main sub-systems:
 - Memory
 - ALU (Arithmetic/Logic Unit)
 - Control Unit
 - Input/Output System (I/O)
 - 2) Program is stored in memory during execution.
 - 3) Program instructions are executed sequentially.

The Von Neumann Architecture

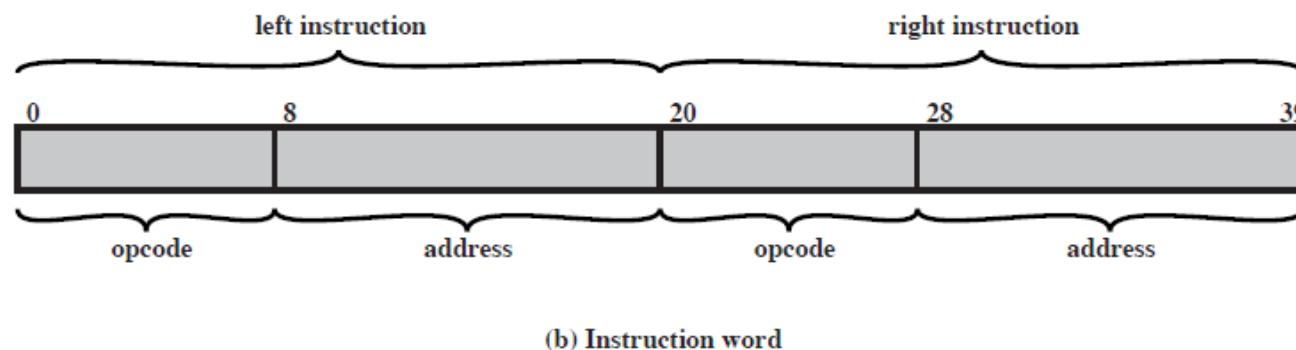


IAS – memory format

- 1000 location with the capacity of 40 bit.
- Both data and instructions are stored in the same memory.

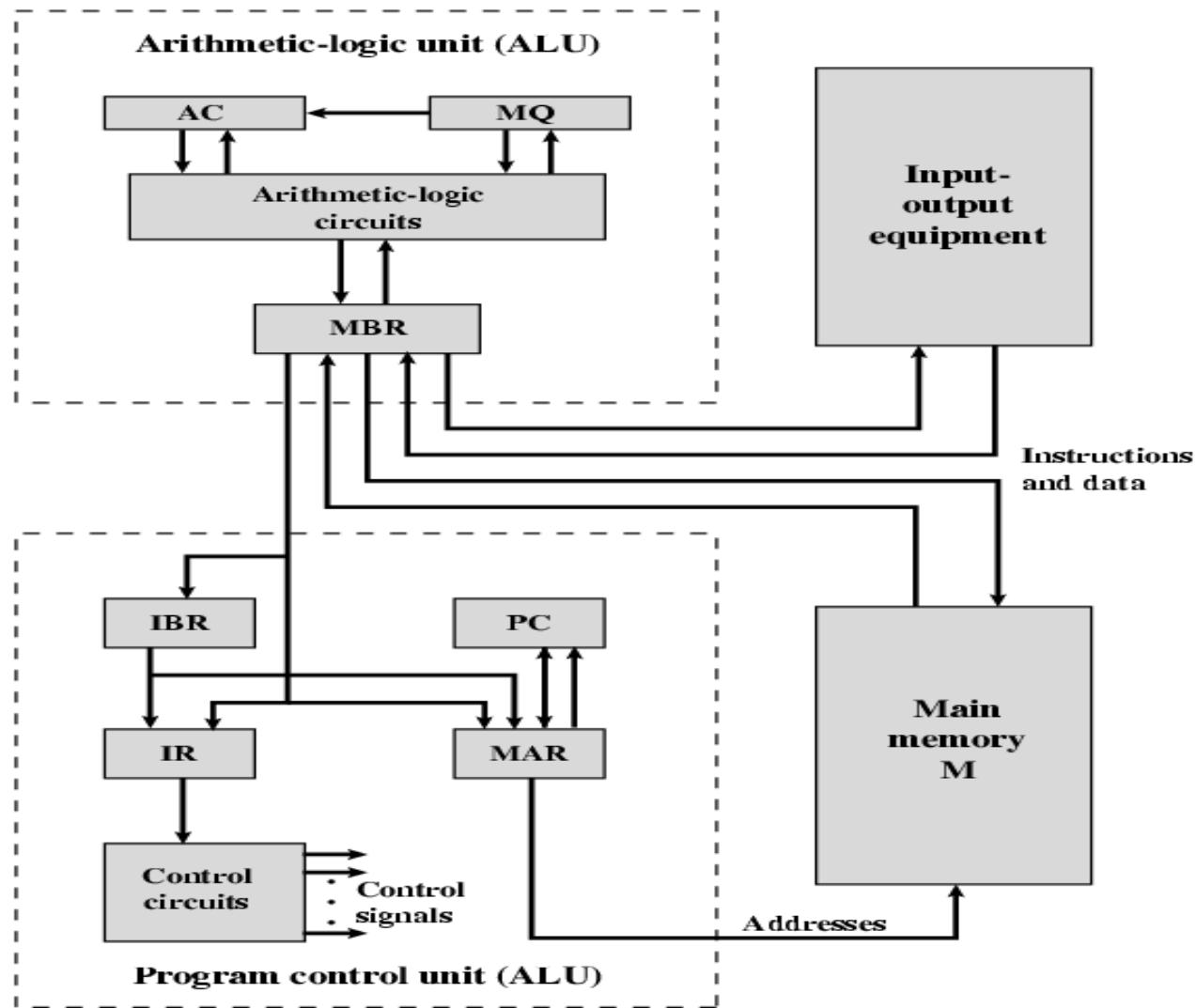


(a) Number word



(b) Instruction word

Expanded Structure of IAS



Expanded structure of IAS computer

- **Set of registers (storage in CPU)**
 - **Memory Buffer Register (MBR)**
 - Contains a word to be stored in memory or sent to the I/O unit, or it is used to receive a word from memory or from the I/O unit.
 - **Memory Address Register (MAR)**
 - Specifies the address in memory of the word to be written from or read into the MBR.
 - **Instruction Register (IR)**
 - Contains the 8 bit opcode instruction being executed.
 - **Instruction Buffer Register (IBR)**
 - Employed to hold temporarily the right hand instruction from a word in memory
 - **Program Counter (PC)**
 - Contains the address of the next instruction pair to be fetched from memory
 - **Accumulator (AC) & Multiplier Quotient (MQ)**
 - Employed to hold temporarily the operands and results of ALU operations. For eg. The result of multiplying two 40 bit numbers is an 80 bit number, the most significant 40 bits are stored in the AC and the least significant in the MQ.

IAS Computer

$MAR \leftarrow PC$

$MBR \leftarrow M[MAR]$

$IBR \leftarrow MBR<20..39>$ $IBR \leftarrow MBR<20..39>$

$IR \leftarrow MBR<0..7>$

$MAR \leftarrow MBR<8..19>$

$MBR \leftarrow M[MAR]$

$AC \leftarrow MBR$

$IR \leftarrow IBR<0..7>$

$MAR \leftarrow IBR<8..19>$

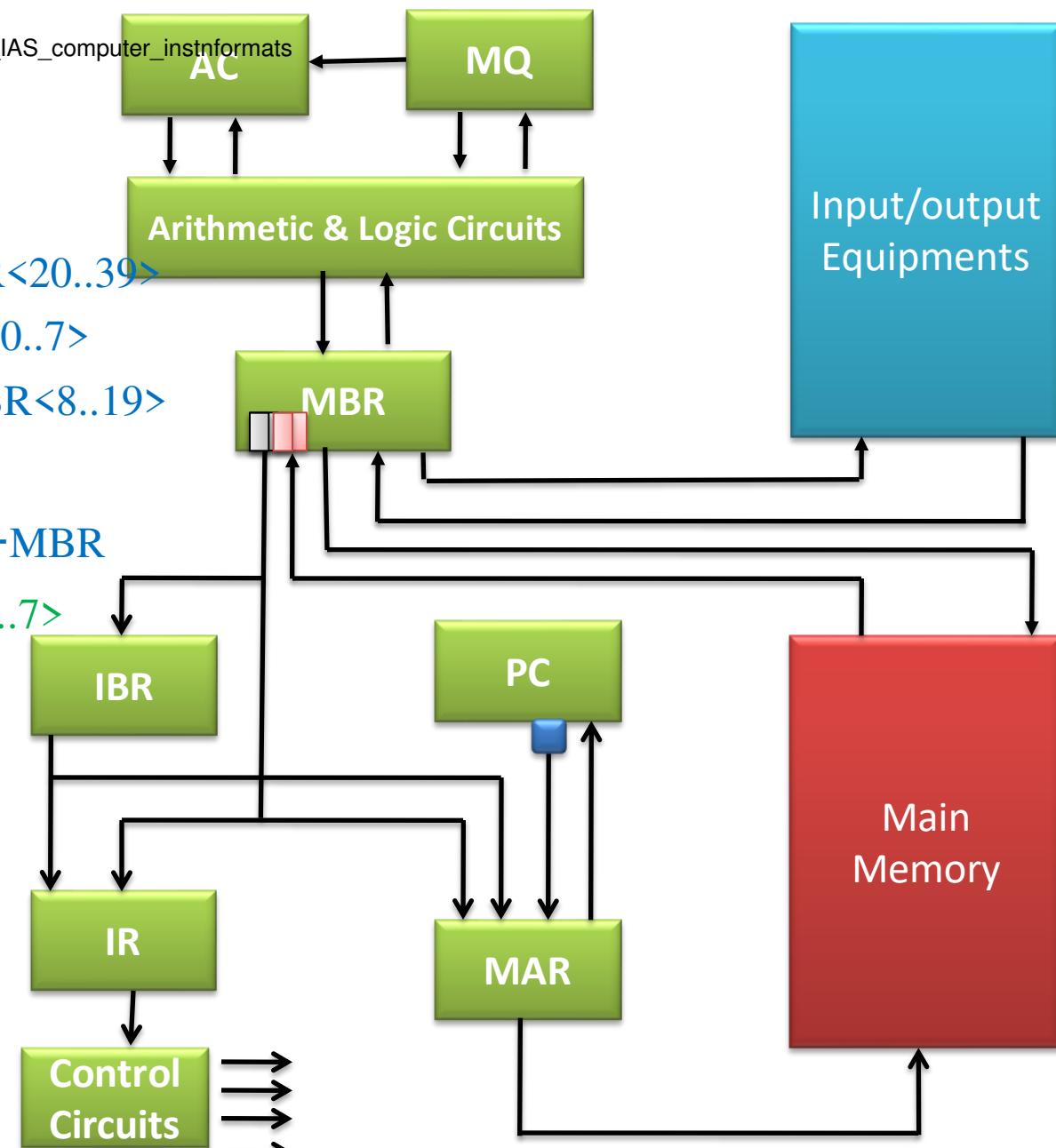
$MBR \leftarrow M[MAR]$

$AC \leftarrow AC + MBR$

$PC \leftarrow PC+1$

$MAR \leftarrow PC$

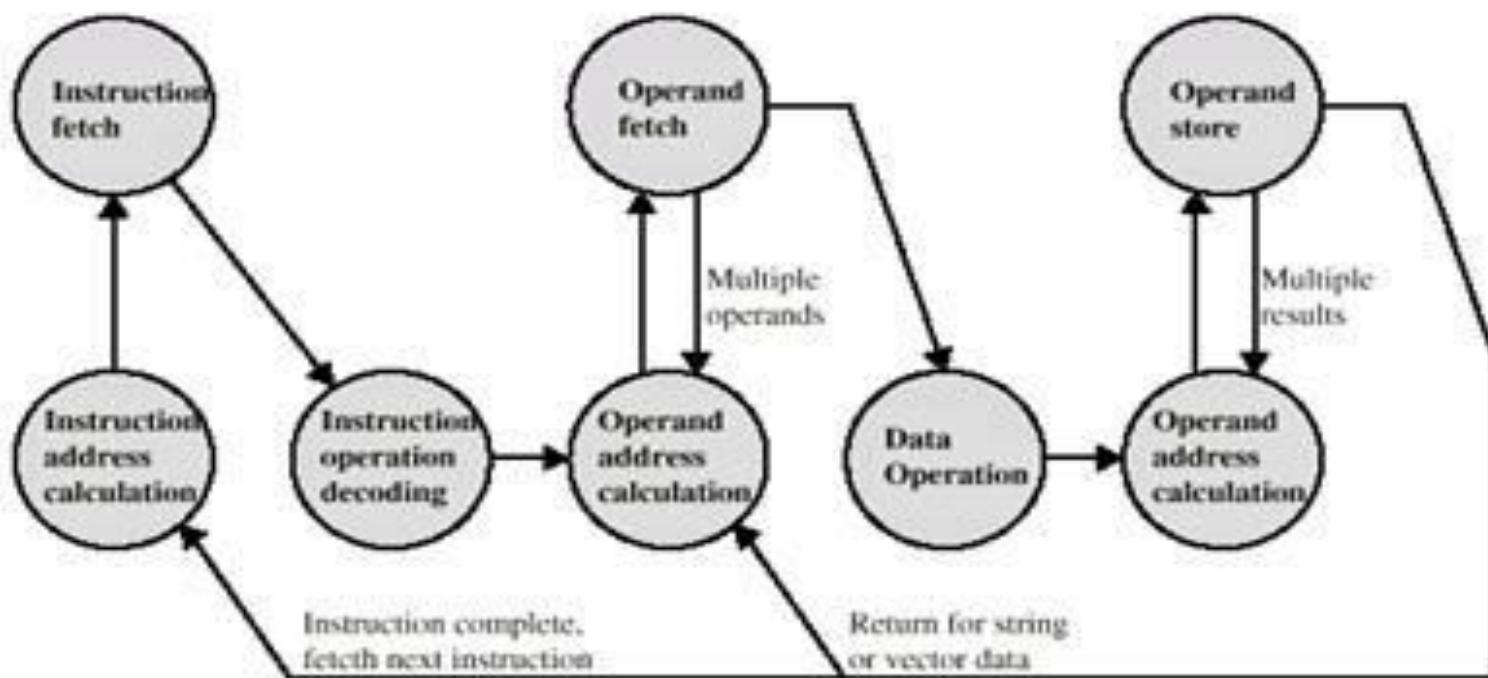
$MBR \leftarrow M[MAR]$



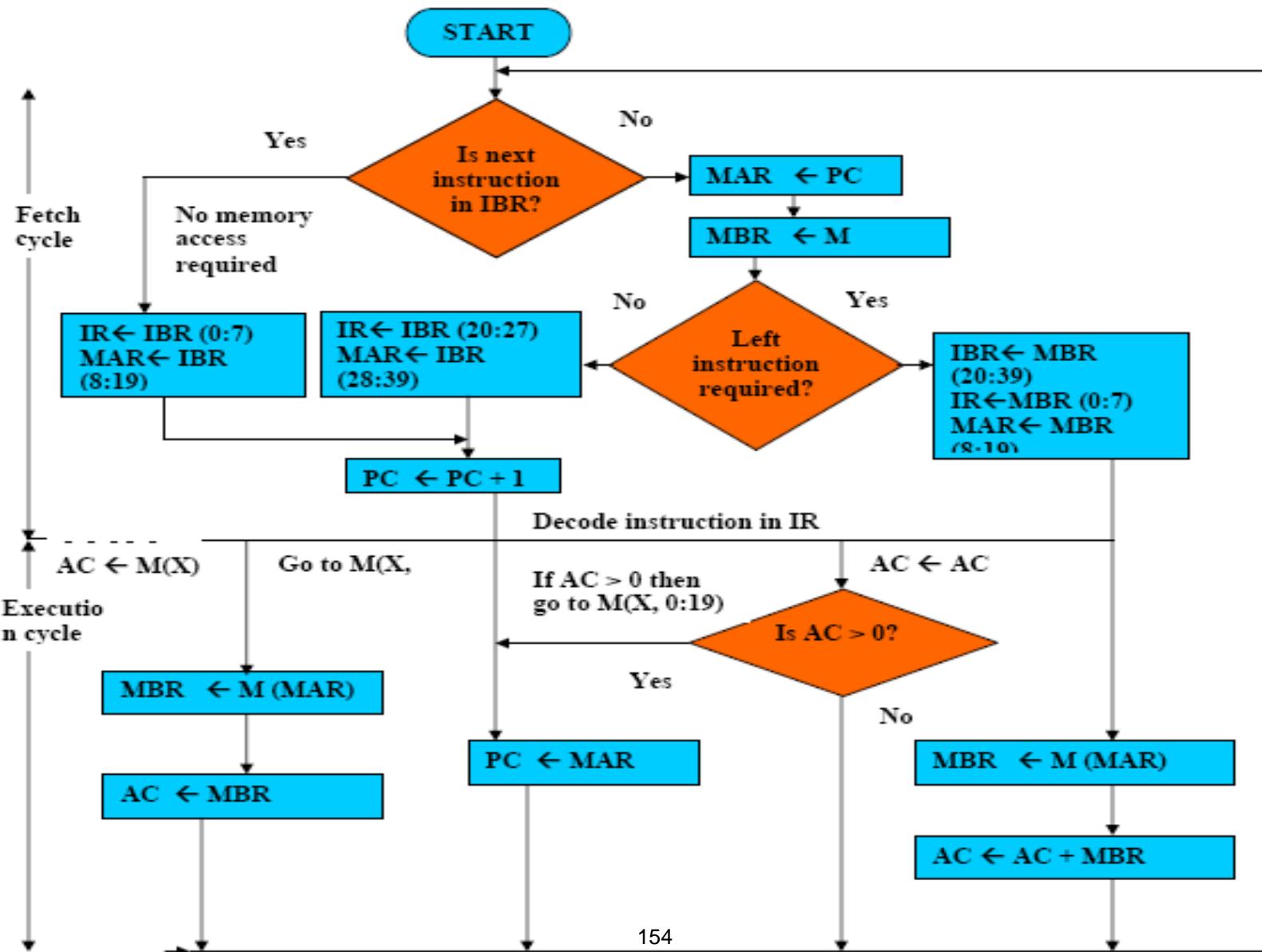
Register transfer operation for addition operation

- Assume PC = 1
- [1] \leftarrow LOAD M(X) ; ADD M(X)
 - MAR \leftarrow PC
 - MBR \leftarrow M[MAR]
 - IBR \leftarrow MBR[20:39]
 - IR \leftarrow MBR[0:7]
 - MAR \leftarrow MBR[8:19]
 - MBR \leftarrow M[MAR]
 - AC \leftarrow MBR
 - IR \leftarrow IBR[0:7]
 - MAR \leftarrow IBR[8:19]
 - MBR \leftarrow M[MAR]
 - AC \leftarrow AC + MBR

Instruction Cycle State Diagram



Fetch / Execute Cycle



Instruction set: Collection of instructions that the CPU can execute

What an Instruction set should specify?

- Which Operation to perform (Opcode)
- Where to find the operand or operands (CPU registers, main memory or I/O port)
- Where to put the result, if there is result
- Where to find the next instruction

IAS – Contd..

IAS – Total 21 Instructions

- Data Transfer
- Unconditional Branch Instruction
- Conditional Branch Instruction
- Arithmetic
- Address Modify Instruction

Instruction Format

The instruction is divided into two fields

- Opcode field
- Operand field

This operand field further divided into one to four fields.

Simple instruction format

Opcode	Operand Address1	Operand Address2	Result Address1	Next Instruction
--------	------------------	------------------	-----------------	------------------

Instruction Set is categorized into types based on

- Operation performed
- number of operand addresses
- and addressing modes

Based on Operation

- Data Movement

Memory : LOAD, STORE, MOV

I\O Instructions: IN, OUT

- Data Processing

Arithmetic : Add, Sub, MUL

Logic Instructions: AND, OR,

- Control Instructions

Conditional : JNZ, JZ

Un Conditional: Jump

Based on number of **operand address** in the instruction.

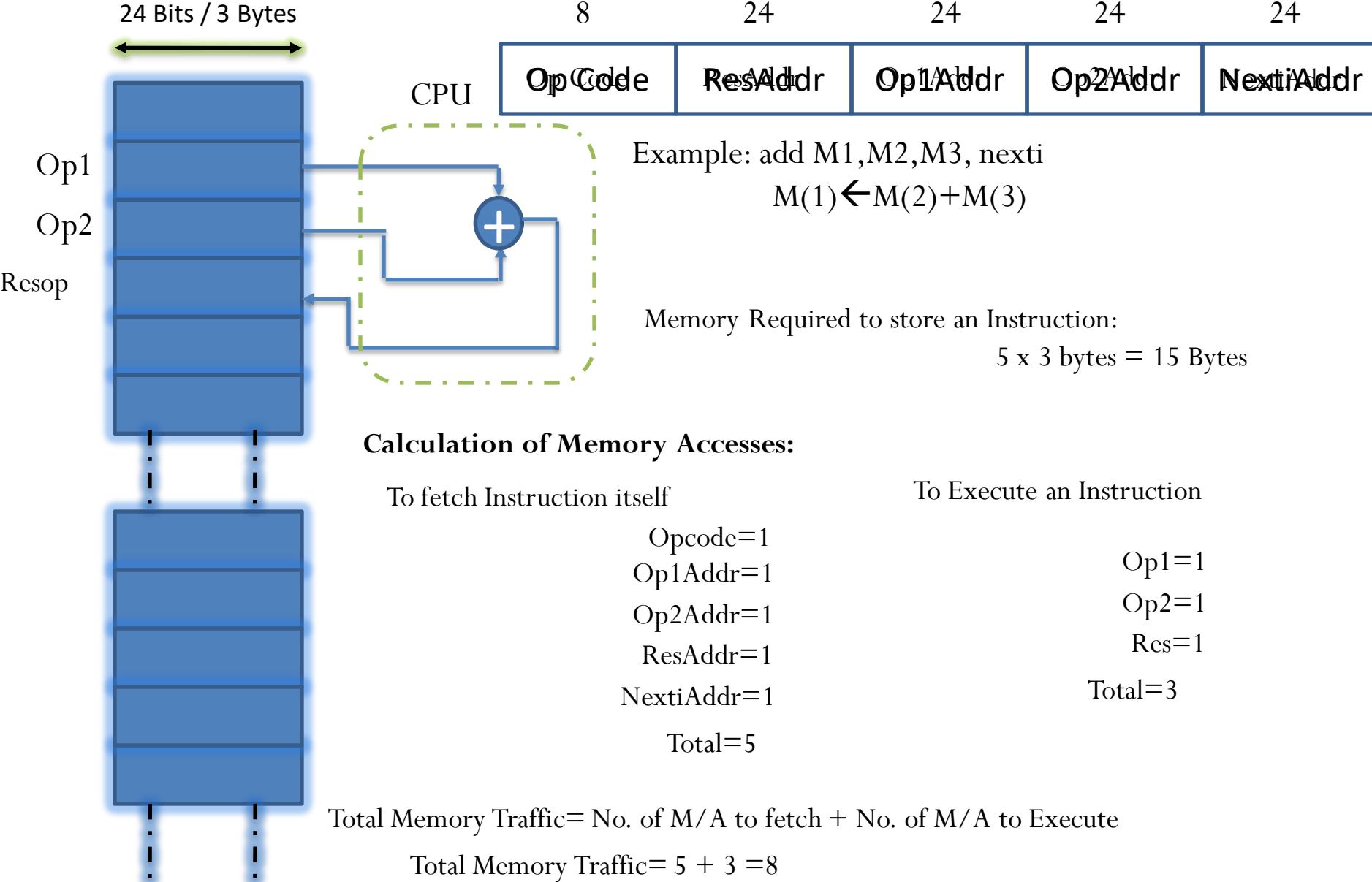
- 4 Address Instruction
- 3 Address Instruction
- 2 Address Instruction
- 1 Address Instruction
- 0 Address Instruction

For a two-operand arithmetic instruction, five items need to be specified

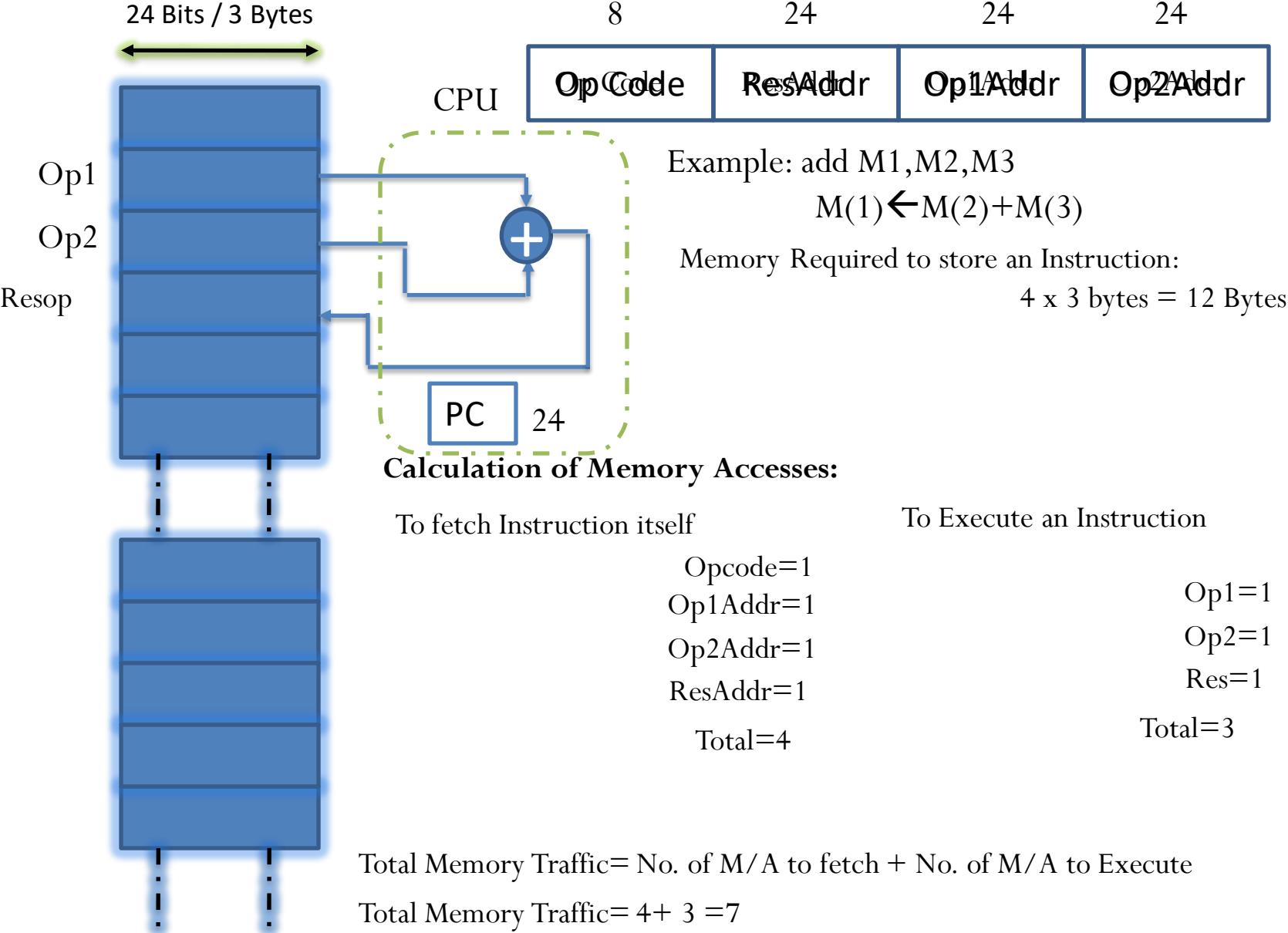
1. Operation to be performed (opcode)
2. Location of the first operand
3. Location of the second operand
4. Place to store the result
5. Location of next instruction to be executed

Assumptions
24-bit memory address (3 bytes)
8-bit opcode

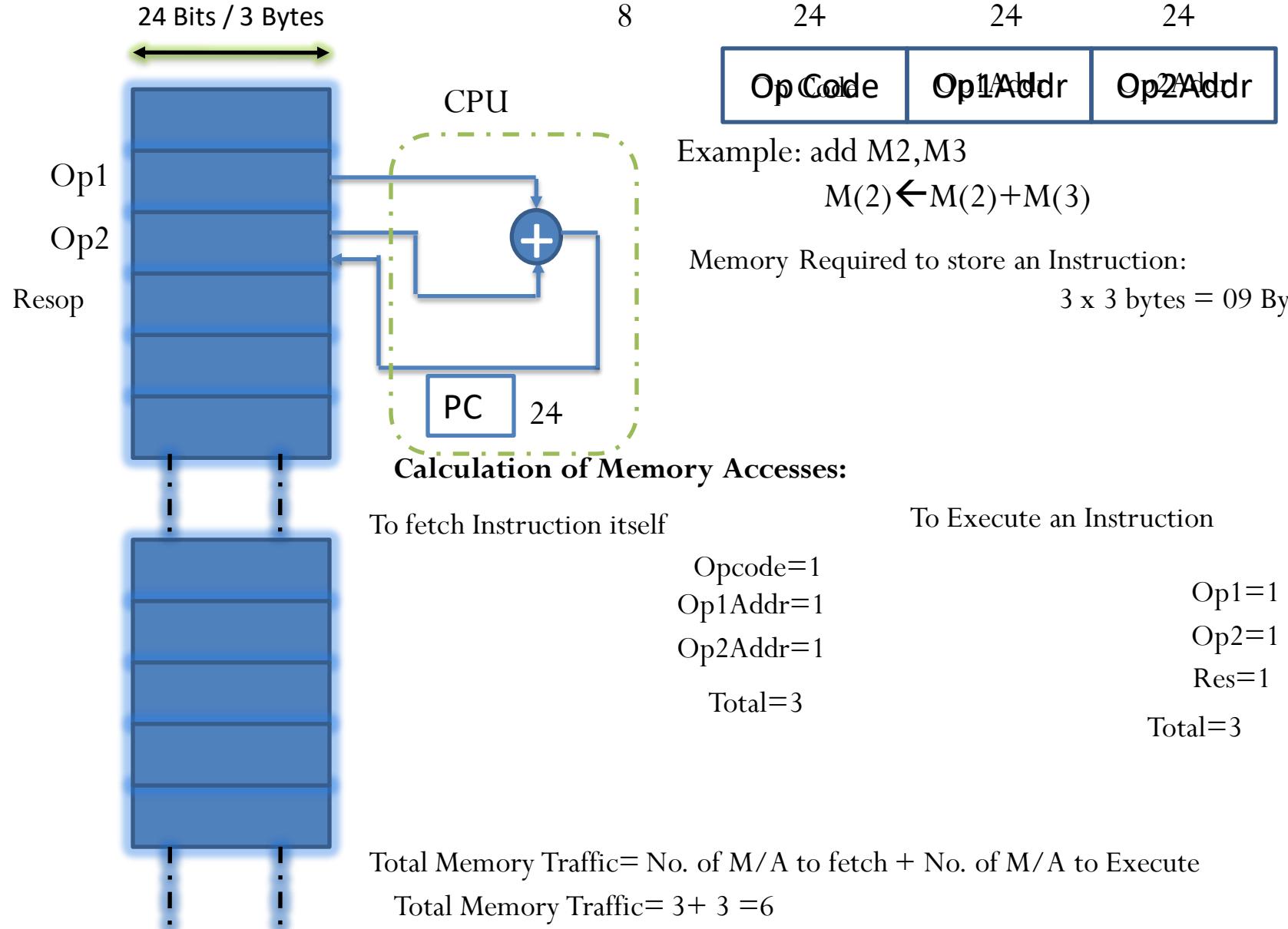
4- Address Instruction



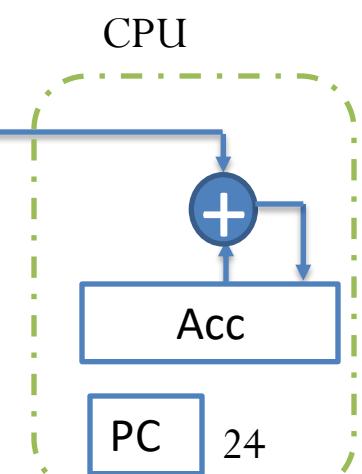
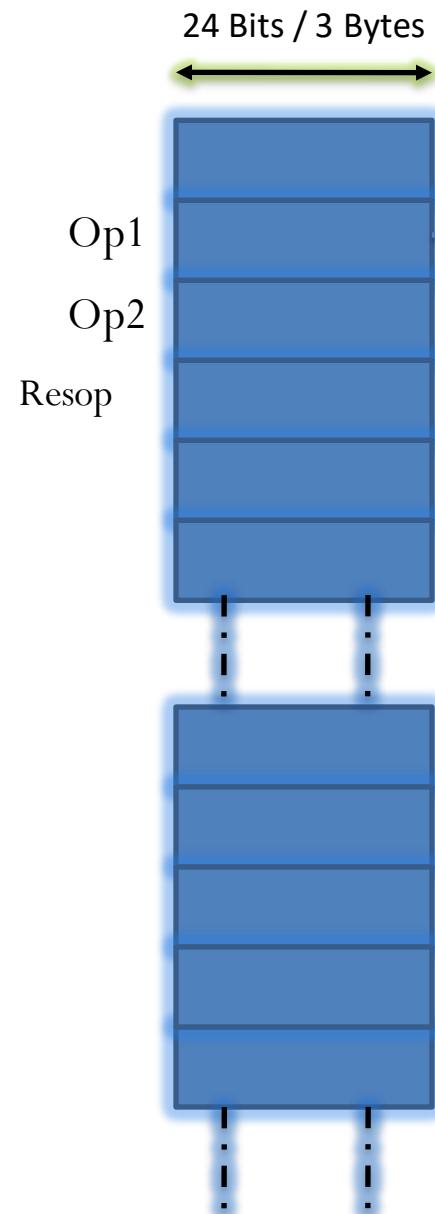
3- Address Instruction



2- Address Instruction



1- Address Instruction



Example: add M2

Memory Required to store an Instruction:

$$2 \times 3 \text{ bytes} = 06 \text{ Bytes}$$

Calculation of Memory Accesses:

To fetch Instruction itself

Opcode=1
Op1Addr=1
Total=2

To Execute an Instruction

Op1=1
Total=1

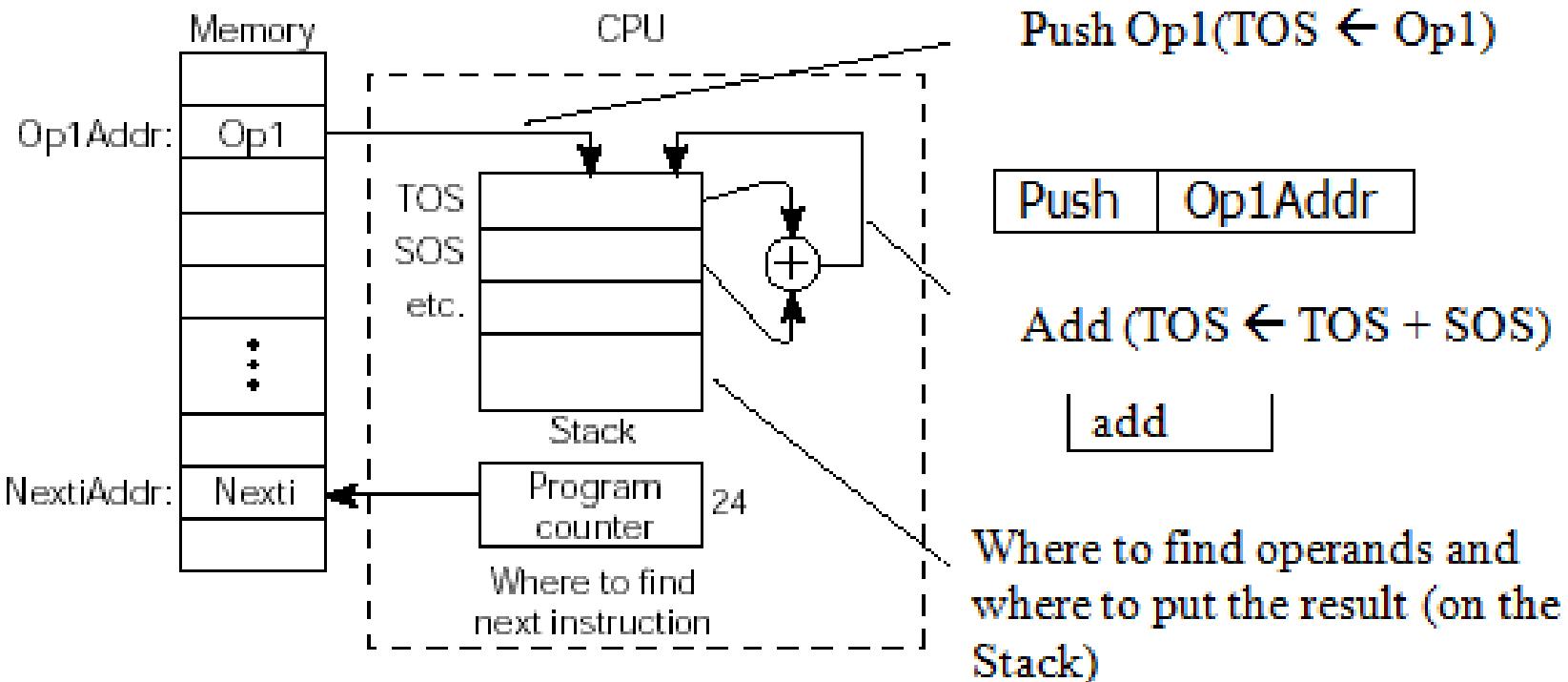
Total Memory Traffic = No. of M/A to fetch + No. of M/A to Execute

$$\text{Total Memory Traffic} = 2 + 1 = 3$$

0-Address Instruction

- Uses a push down stack in CPU
- Arithmetic uses stack for both operands and the result

Computer must have a 1-address instruction to push and pop operands to and from the stack



Comparisons

Instruction Type	Memory To Store in Bytes	Memory To Encode in Bytes	M/As to fetch an Instruction	M/As to Execute an Instruction	Memory Traffic
4-address	$5 \times 3 = 15$	$1 + (4 \times 3) = 13$	5	3	$5+3=8$
3-Address	$4 \times 3 = 12$	$1 + (3 \times 3) = 10$	4	3	$4+3=7$
2-Address	$3 \times 3 = 09$	$1 + (2 \times 3) = 07$	3	3	$3+3=6$
1-Address	$2 \times 3 = 06$	$1 + (1 \times 3) = 04$	2	1	$2+1=3$
0-Address	$1 \times 3 = 03$	$1 + (0 \times 3) = 01$	1	0	$1+0=1$

Evaluate $a = (b+c)*d - e$

3-Address

add a, b, c	$a \leftarrow b+c$
mpy a, a, d	$a \leftarrow a*d$
sub a, a, e	$a \leftarrow a-e$

0-Address

push b
push c
add
push d
mpy
push e
sub

2-Address

load a, b	$a \leftarrow b$
add a, c	$a \leftarrow a+c$
mpy a, d	$a \leftarrow a*d$
sub a, e	$a \leftarrow a-e$

pop a

1-Address

load b	$Acc \leftarrow b$
add c	$Acc \leftarrow Acc + c$
mpy d	$Acc \leftarrow Acc * d$
sub e	$Acc \leftarrow Acc - e$
store a	$a \leftarrow Acc$

		Memory to Store	Memory to encode	M/As to Fetch	M/As to Execute	Memory Traffic
add a, b, c	$a \leftarrow b+c$	$4*3=12$	$1+(3*3)=10$	4	3	$4+3=7$
mpy a, a, d	$a \leftarrow a*d$	$4*3=12$	$1+(3*3)=10$	4	3	$4+3=7$
sub a, a, e	$a \leftarrow a-e$	$4*3=12$	$1+(3*3)=10$	4	3	$4+3=7$
		36	30	12	9	21

		Memory to Store	Memory to encode	M/As to Fetch	M/As to Execute	Memory Traffic
load a, b	$a \leftarrow b$	$3*3=9$	$1+(2*3)=7$	3	2	$3+2=6$
add a, c	$a \leftarrow a+c$	$3*3=9$	$1+(2*3)=7$	3	3	$3+3=6$
mpy a, d	$a \leftarrow a*d$	$3*3=9$	$1+(2*3)=7$	3	3	$3+3=6$
sub a, e	$a \leftarrow a-e$	$3*3=9$	$1+(2*3)=7$	3	3	$3+3=6$
		36	28	12	11	23

		Memory to Store	Memory to encode	M/As to Fetch	M/As to Execute	Memory Traffic
load b	Acc ← b	2*3=6	1+(1*3)=4	2	1	2+1=3
add c	Acc ← Acc+c	2*3=6	1+(1*3)=4	2	1	2+1=3
mpy d	Acc ← Acc*d	2*3=6	1+(1*3)=4	2	1	2+1=3
sub e	Acc ← Acc-e	2*3=6	1+(1*3)=4	2	1	2+1=3
store a	a ← Acc	2*3=6	1+(1*3)=4	2	1	2+1=3
		30	20	10	5	15
push b		6	4	2	1	3
push c						
add		3	1	1	0	1
push d						
mpy						
push e						
sub						
pop a						
		39	23	13	5	18

6-IAS Machine-16-Summary of 3, 2, 1, and 0-Address instruction programming

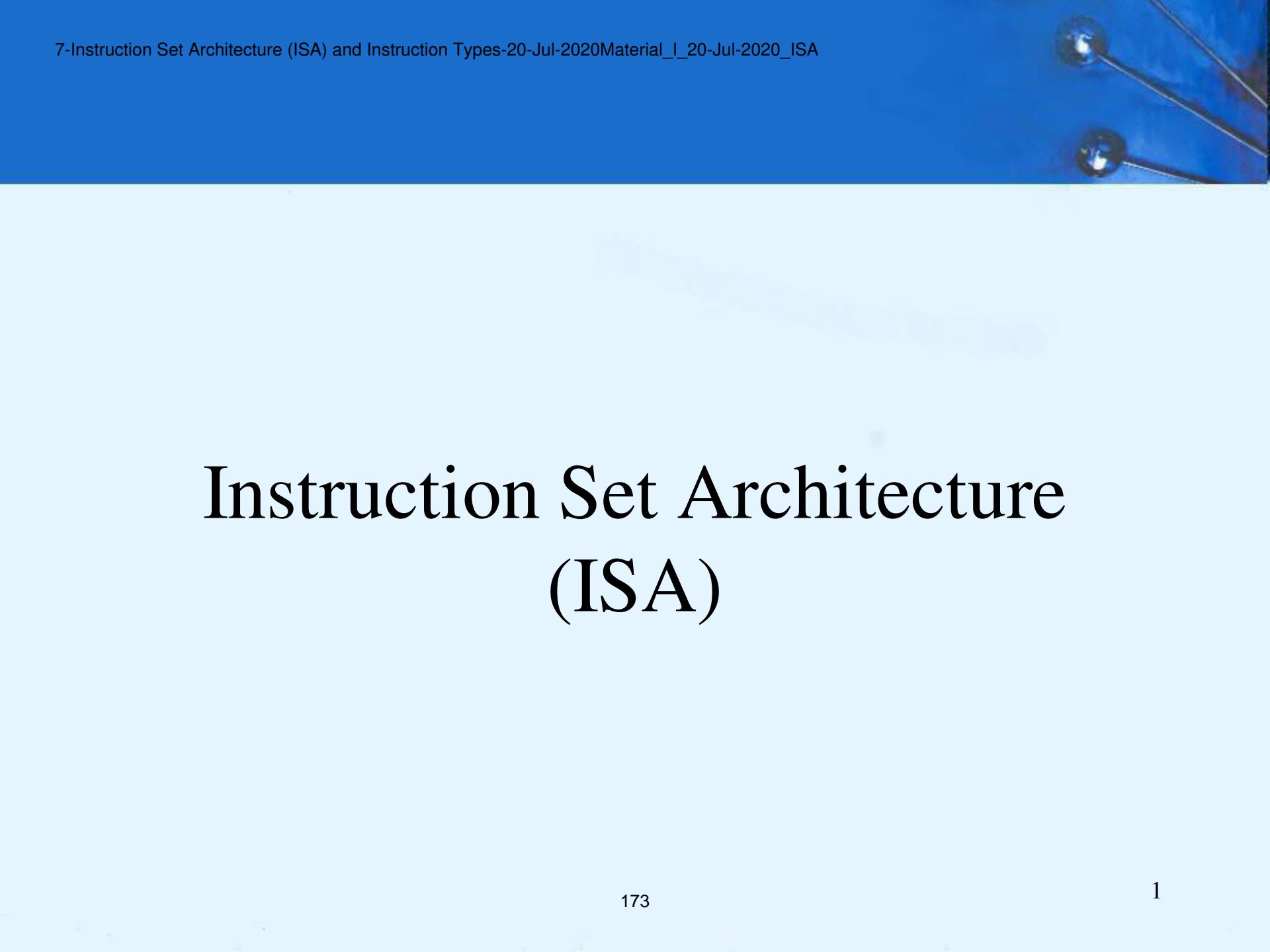
Instruction Format	Memory to Store	Memory to Encode	M/As to Fetch	M/As to Execute	Memory Traffic
4-Address	45	39	24	9	33
3-Address	36	30	12	9	21
2-Address	36	28	12	11	24
1-Address	30	20	10	5	15
0-Address	8	8	4	0	4

- Evaluate $X = (A + B) * (C + D)$
- Evaluate $Y = (A - B) / [C + (D / E)]$

References

Reference Book

- Vincent .P. Heuring, Harry F. Jordan “Computer System design and Architecture” Pearson, 2nd Edition, 2003
- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- J. P. Hayes, Computer system architecture, McGraw Hill,2000



Instruction Set Architecture (ISA)

5.1 Introduction

- This chapter builds upon the ideas in Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

5.2 Instruction Formats

In designing an instruction set, consideration is given to:

- Instruction length.
 - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
 - Whether byte- or word addressable.
- Addressing modes.
 - Choose any or all: direct, indirect or indexed.

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
 - In *little endian* machines, the least significant byte is followed by the most significant byte.
 - *Big endian* machines store the most significant byte first (at the lower address).

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.2 Instruction Formats

- Big endian:
 - Is more natural.
 - The sign of the number can be determined by looking at the byte at address offset 0.
 - Strings and integers are stored in the same order.
- Little endian:
 - Makes it easier to place values on non-word boundaries.
 - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

5.2 Instruction Formats

- The next consideration for architecture design concerns how the CPU will store data.
- We have three choices:
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

5.2 Instruction Formats

- In a stack architecture, instructions and operands are implicitly taken from the stack.
 - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
 - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
 - Faster than accumulator architecture.
 - Efficient implementation for compilers.
 - Results in longer instructions.

5.2 Instruction Formats

- Most systems today are GPR systems.
- There are three types:
 - Memory-memory where two or three operands may be in memory.
 - Register-memory where at least one operand must be in a register.
 - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

5.2 Instruction Formats

- Stack machines use one- and zero-operand instructions.
- **LOAD** and **STORE** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **PUSH** and **POP** operations involve only the stack's top element.
- Binary instructions (e.g., **ADD**, **MULT**) use the top two items on the stack.

5.2 Instruction Formats

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: $Z = X + Y$.
- Stack arithmetic requires that we use *postfix* notation: $Z = XY+$.
 - This is also called *reverse Polish notation*, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

5.2 Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

$$Z = (X \times Y) + (W \times U),$$

becomes:

$$Z = X Y \times W U \times +$$

in postfix notation.

5.2 Instruction Formats

- In a stack ISA, the postfix expression,

$Z = X Y \times W U \times +$

might look like this:

PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z

Note: The result of a binary operation is implicitly stored on the top of the stack!

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

$$Z = X \times Y + W \times U$$

looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

5.2 Instruction Formats

- In a two-address ISA, (e.g., Intel, Motorola), the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
LOAD R1,X  
MULT R1,Y  
LOAD R2,W  
MULT R2,U  
ADD R1,R2  
STORE Z,R1
```

Note: One-address ISAs usually require one operand to be a register.

5.2 Instruction Formats

- With a three-address ISA, (e.g.,mainframes),
the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
MULT R1,X,Y  
MULT R2,W,U  
ADD Z,R1,R2
```

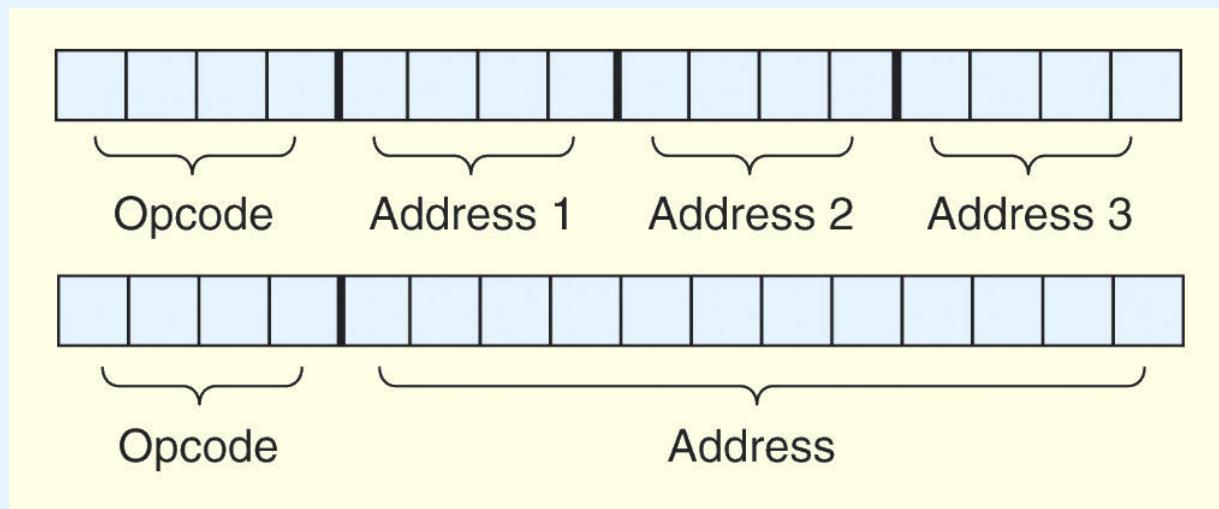
Would this program execute faster than the corresponding (longer) program that we saw in the stack-based ISA?

5.2 Instruction Formats

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as **HALT**, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

5.2 Instruction Formats

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

0000	R1	R2	R3	15 3-address codes
1110	R1	R2	R3	
1111 0000	R1	R2	14 2-address codes	
1111 1101	R1	R2		
1111 1110 0000	R1	31 1-address codes		
1111 1111 1110	R1			
1111 1111 1111 0000	16 0-address codes			
1111 1111 1111 1111				

Is there something missing from this instruction set?

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

Can you think of some examples of each of these?

5.4 Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

5.4 Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the address of the data.

5.4 Addressing

- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

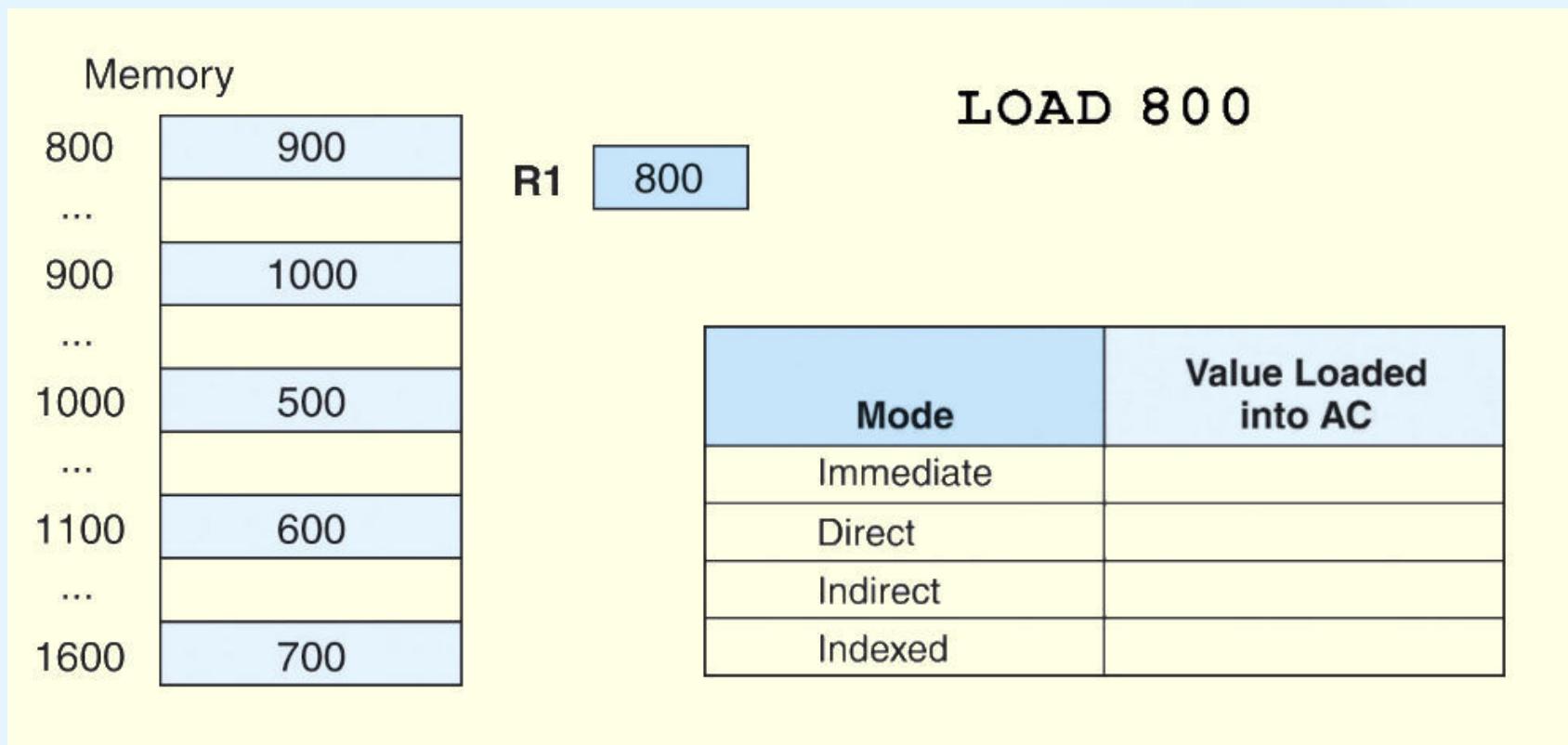
5.4 Addressing

- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
 - Indirect indexed.
 - Base/offset.
 - Self-relative
 - Auto increment - decrement.
- We won't cover these in detail.

Let's look at an example of the principal addressing modes.

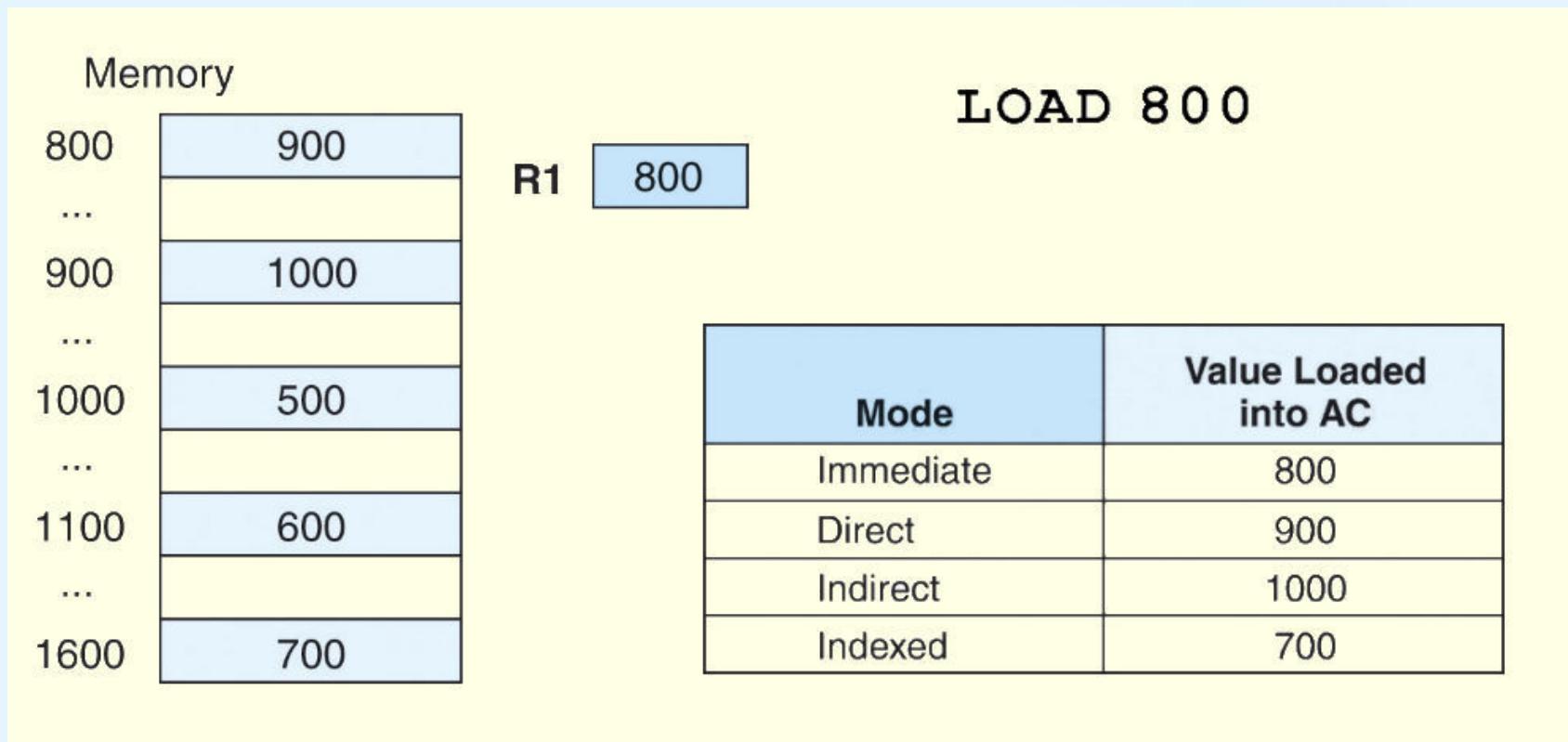
5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



5.4 Addressing

- These are the values loaded into the accumulator for each addressing mode.



5.5 Instruction-Level Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction-level pipelining*.
- This term is sometimes abbreviated *ILP* in the literature.

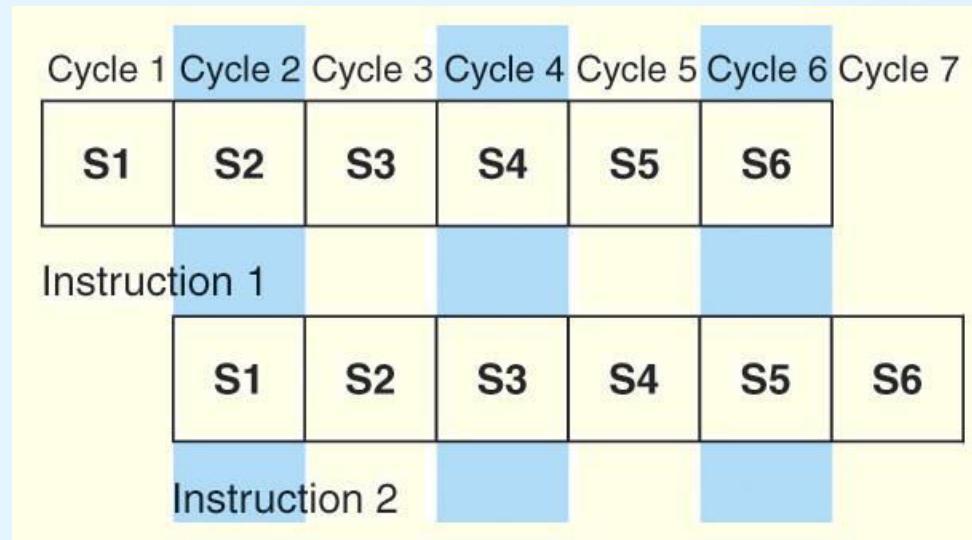
The next slide shows an example of instruction-level pipelining.

5.5 Instruction-Level Pipelining

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:
 1. Fetch instruction.
 2. Decode opcode.
 3. Calculate effective address of operands.
 4. Fetch operands.
 5. Execute instruction.
 6. Store result.
- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

5.5 Instruction-Level Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



S1. Fetch instruction.
 S2. Decode opcode.
 S3. Calculate effective address of operands.

S4. Fetch operands.
 S5. Execute.
 S6. Store result.

5.5 Instruction-Level Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

Let t_p be the time per stage. Each instruction represents a task, T , in the pipeline.

The first task (instruction) requires $k \times t_p$ time to complete in a k -stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

Thus, to complete n tasks using a k -stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

5.5 Instruction-Level Pipelining

- If we take the time required to complete n tasks without a pipeline and divide it by the time it takes to complete n tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1) t_p}$$

- If we take the limit as n approaches infinity, $(k + n - 1)$ approaches n , which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

5.5 Instruction-Level Pipelining

- Our neat equations take a number of things for granted.
- First, we have to assume that the architecture supports fetching instructions and data in parallel.
- Second, we assume that the pipeline can be kept filled at all times. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.

5.5 Instruction-Level Pipelining

- An instruction pipeline may stall, or be flushed for any of the following reasons:
 - Resource conflicts.
 - Data dependencies.
 - Conditional branching.
- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

5.6 Real-World Examples of ISAs

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two five-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.
- The Itanium (IA-64) has only a 10-stage pipeline.

5.6 Real-World Examples of ISAs

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this chapter.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

5.6 Real-World Examples of ISAs

- MIPS was an acronym for *Microprocessor Without Interlocked Pipeline Stages*.
- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Like Intel, the pipeline size of the MIPS processors has grown: The R2000 and R3000 have five-stage pipelines.; the R4000 and R4400 have 8-stage pipelines.

5.6 Real-World Examples of ISAs

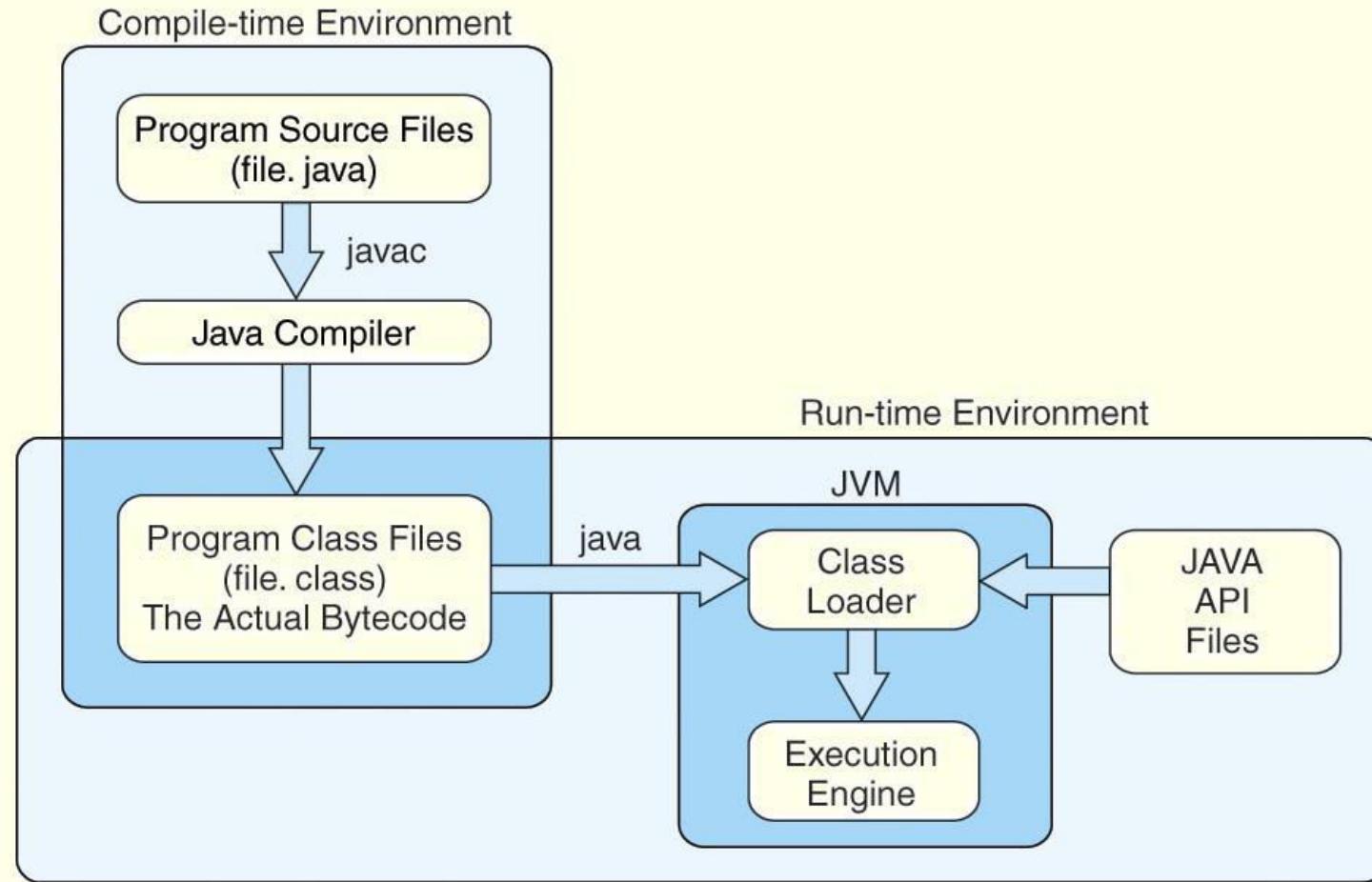
- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for **LOAD/STORE** instructions.
- In all MIPS ISAs, only the **LOAD** and **STORE** instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

5.6 Real-World Examples of ISAs

- The Java programming language is an interpreted language that runs in a software machine called the *Java Virtual Machine* (JVM).
- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- Like a real machine, the JVM has an ISA all of its own, called *bytecode*. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

The next slide shows how the pieces fit together.

5.6 Real-World Examples of ISAs



5.6 Real-World Examples of ISAs

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for platform interoperability, not performance!

Chapter 5 Conclusion

- ISAs are distinguished according to their bits per instruction, number of operands per instruction, operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store data based on
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.

Chapter 5 Conclusion

- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor. We looked at:
 - Immediate
 - Register
 - Indirect
 - Based
 - Direct
 - Register Indirect
 - Indexed
 - Stack

Chapter 5 Conclusion

- A k -stage pipeline can theoretically produce execution speedup of k as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.
- The Intel, MIPS, and JVM architectures provide good examples of the concepts presented in this chapter.

Assembly Language Programming

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
• LOAD MQ	00001010	AC \leftarrow MQ
• LOAD MQ, M(X)	00001001	MQ \leftarrow M(X)
• STOR M(X)	00100001	M(X) \leftarrow AC
• LOAD M(X)	00000001	AC \leftarrow M(X)
• LOAD - M(X)	00000010	AC \leftarrow - M(X)
• LOAD M(X)	00000011	AC \leftarrow M(X)
• LOAD - M(X)	00000100	AC \leftarrow - M(X)

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
• JUMP M(X,0:19)	00001101	next instruction M(X,0:19)
• JUMP M(X,20:39)	00001110	next instruction M(X,20:39)

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
• JUMP +M(X,0:19)	00001111	IF AC ≥ 0 , then next instruction M(X,0:19)
• JUMP +M(X,20:39)	00010000	IF AC ≥ 0 , then next instruction M(X,20:39)

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
• ADD M(X)	00000101	$AC \leftarrow AC + M(X)$
• ADD M(X)	00000111	$AC \leftarrow AC + M(X) $
• SUB M(X)	00000110	$AC \leftarrow AC - M(X)$
• SUB M(X)	00001000	$AC \leftarrow AC - M(X) $
• MUL M(X)	00001011	$AC, MQ \leftarrow MQ \times M(X)$
• DIV M(X)	00001100	$MQ, AC \leftarrow MQ / M(X)$
• LSH	00010100	$AC \leftarrow AC \times 2$
• RSH	00010101	$AC \leftarrow AC / 2$

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
• STOR M(X,8:19)	00010010	M(X,8:19) ← AC(28:39)
• STOR M(X,28:39)	00010011	M(X,28:39) ← AC(28:39)

ALP

Write an appropriate assembly language code for the following operation and interpret to Von Neumann IAS architecture

$$X = Y * Z$$

Where $Z \rightarrow 40$ bit data and $Y \rightarrow 40$ bit data
Result would be more than 40 bit.

Assume that data variables 'Y' & 'Z' available at memory locations 801 & 802 respectively. And X will be stored 803 onwards.

LOAD MQ, M(801) $MQ \leftarrow M[801]$

MUL M(802) $Ac \leftarrow MQ * M[802]$

STOR M(803) $M[803] \leftarrow Ac$

LOAD MQ $Ac \leftarrow MQ$

STOR M(804) $M[804] \leftarrow Ac$

Write an assembly language program to perform
A= (B-C) * D

Assumption: Inputs are available 801 onwards

Load M(801)

Sub M(802)

Stor M(804)

Load MQ, M(804)

Mul M(803)

Stor M(804)

Load MQ

Stor M(805)

- Write an ALP for the following
- If $a > 0$

$$a = a - b$$

else

$$a = a + b$$

Assume:

$$[100] \leftarrow a \quad [101] \leftarrow b \quad [100] \leftarrow a+b/a-b$$

- 1 LOAD M(100), JMP +M(3,0:19)
- 2 ADD M(101), JMP M(3,20:39)
- 3 SUB M(101), STOR M(100)

Write an assembly language program to perform

$$X = (A + B) * (C + D)$$

$$Y = (A - B) / [C + (D/E)]$$

Instruction Types

Classification of computer instructions

- Most computer instructions can be classified into three categories:
 1. Data transfer instructions
 2. Data manipulation A & L Instructions
 3. Program control instructions
 4. I/O Instructions

1. Data Transfer Instruction

- Move data from one place to another without changing the data content in the computer.
- Different data transfers:
 - Memory \leftrightarrow processor registers
 - Processor registers \leftrightarrow input or output
 - Processor register \leftrightarrow processor register

Data Transfer

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

1. Data Transfer Instruction cont..

- **Load**: transfer from memory to a processor **register**, usually an **AC** (*memory read*)
- **Store**: transfer from a processor **register** into memory (*memory write*)
- **Move**: transfer from **one register** to **another register**
- **Exchange**: swap information between **two registers** or a **register and a memory word**
- **Input/Output**: transfer data among processor **registers** and **input/output device** (**I/O instructions**)
- **Push**: transfer data between processor **registers** and a **memory stack**
- **Pop** : transfer data from **stack** to processor **registers**.

LOAD Instructions in Different Addressing Modes

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD \$ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$
Autodecrement	LD -(R1)	$R1 \leftarrow R1 - 1, AC \leftarrow M[R1]$

2. Data Manipulation Instruction

- Perform operations on data and provide the computational capabilities for the computer.
 - I. Arithmetic,
 - II. Logical and bit manipulation,
 - III. Shift Instruction

I. Arithmetic Instructions cont..

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

II. Logical and Bit Manipulation Instructions cont..

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

III. Shift Instructions cont..

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

- **Logical shift left**

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Shift by 1 bit towards left

0

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

After shifting two times

0

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

After shifting three times

- Logical shift Right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Shift by 1 bit towards right

0

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

After shifting two times

0

0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

After shifting three times

Arithmetic shift left

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Sign bit
↑

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Sign bit
↑

0

Shift by 1 bit towards left

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

Sign bit
↑

0

After shifting two times

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

Sign bit
↑

0

After shifting three times

Overflow occurs as sign bit changes

- **Arithmetic shift Right:** -ve values are in 2's complement form

1	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

1	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

Shift by 1 bit towards right

1	1	0	0	1	1	0	1
---	---	---	---	---	---	---	---

↑
Sign bit

After shifting two times

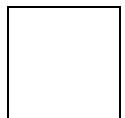
1	1	1	0	0	1	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

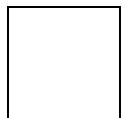
After shifting three times

- **Rotate left**

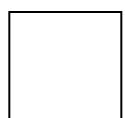
0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

**Buffer**

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Rotate by 1 bit towards left**Buffer**

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

After rotating two times**Buffer**

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

After rotating three times

- **Rotate right**

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

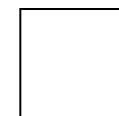
rotate by 1 bit towards right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---



Buffer

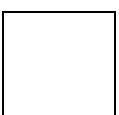
0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---



After rotating two times

Buffer

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

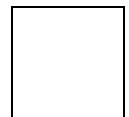


After rotating three times

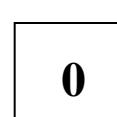
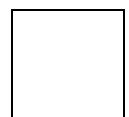
Buffer

- Rotate left through carry

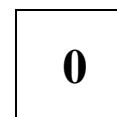
0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

**Buffer**

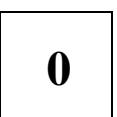
0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Rotate by 1 bit towards left**Carry****Buffer**

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

After rotating two times**Carry****Buffer**

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

After rotating three times**Carry**

- **Rotate right through carry**

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0

Carry

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

rotate by 1 bit towards right

--

Buffer

0

Carry

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

--

After rotating two times

Buffer

0

Carry

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

--

After rotating three times

Buffer

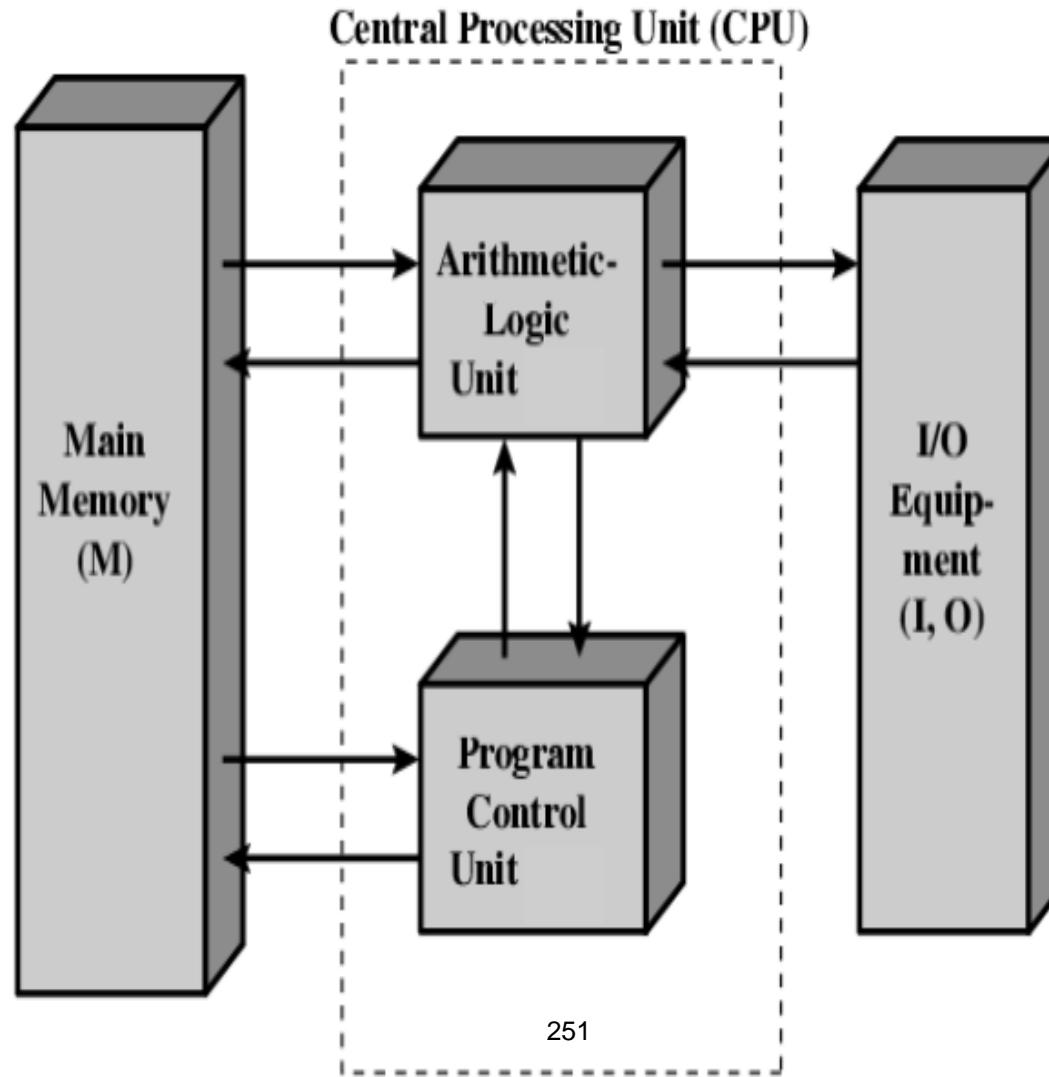
3. Program control Instructions

Operation Name	Description
Jump Unconditional	Unconditional transfer
Jump	Test specified condition
Jump to subroutine	Jump to specified address
Return	Replace the content of PC
Execute	Execute instructions
Skip	Increment PC to skip next Instruction
Skip conditi	Test conditon for skip
Halt	Stop program execution
Wait (hold)	Stop program execution and resume when condition satisfied
No operation	No operation performed but program execution continued

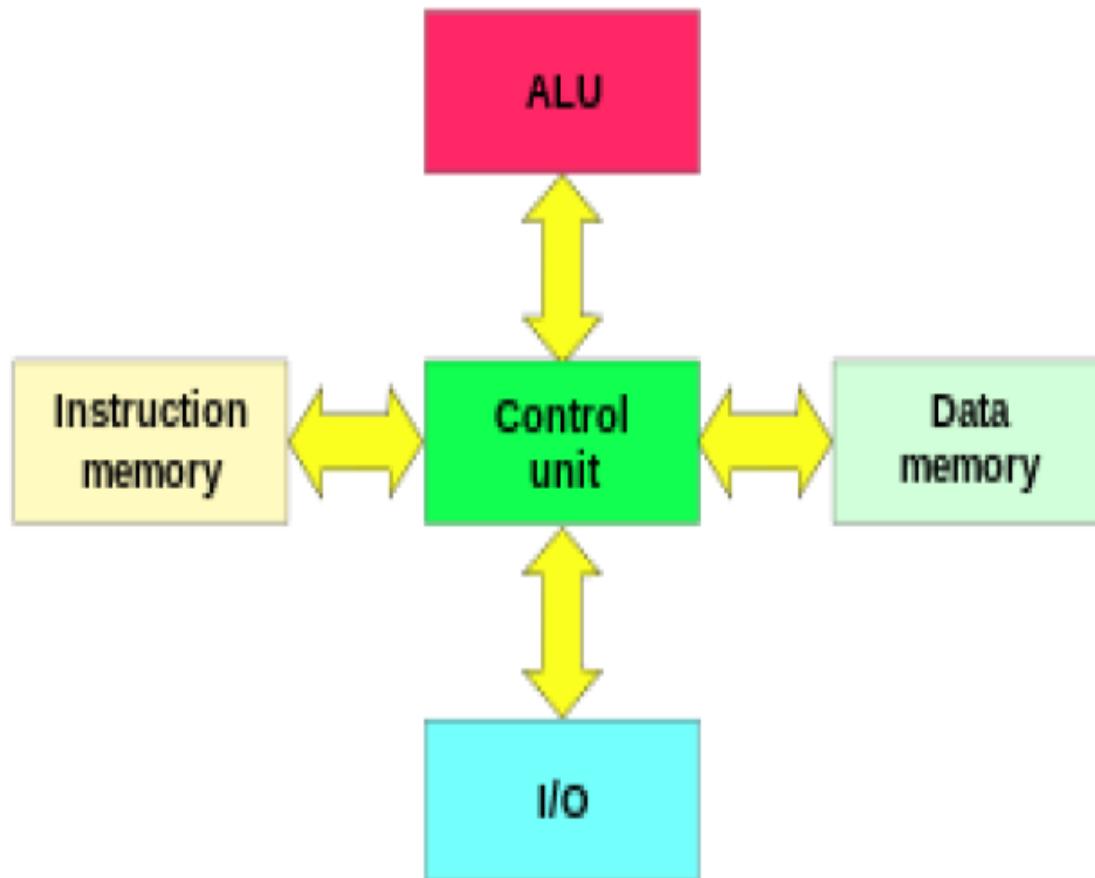
Von-Neumann and Harvard Architecture

LIJO V P
SCOPE
VIT, VELLORE

Structure of Von Neumann Machine



Harvard Architecture



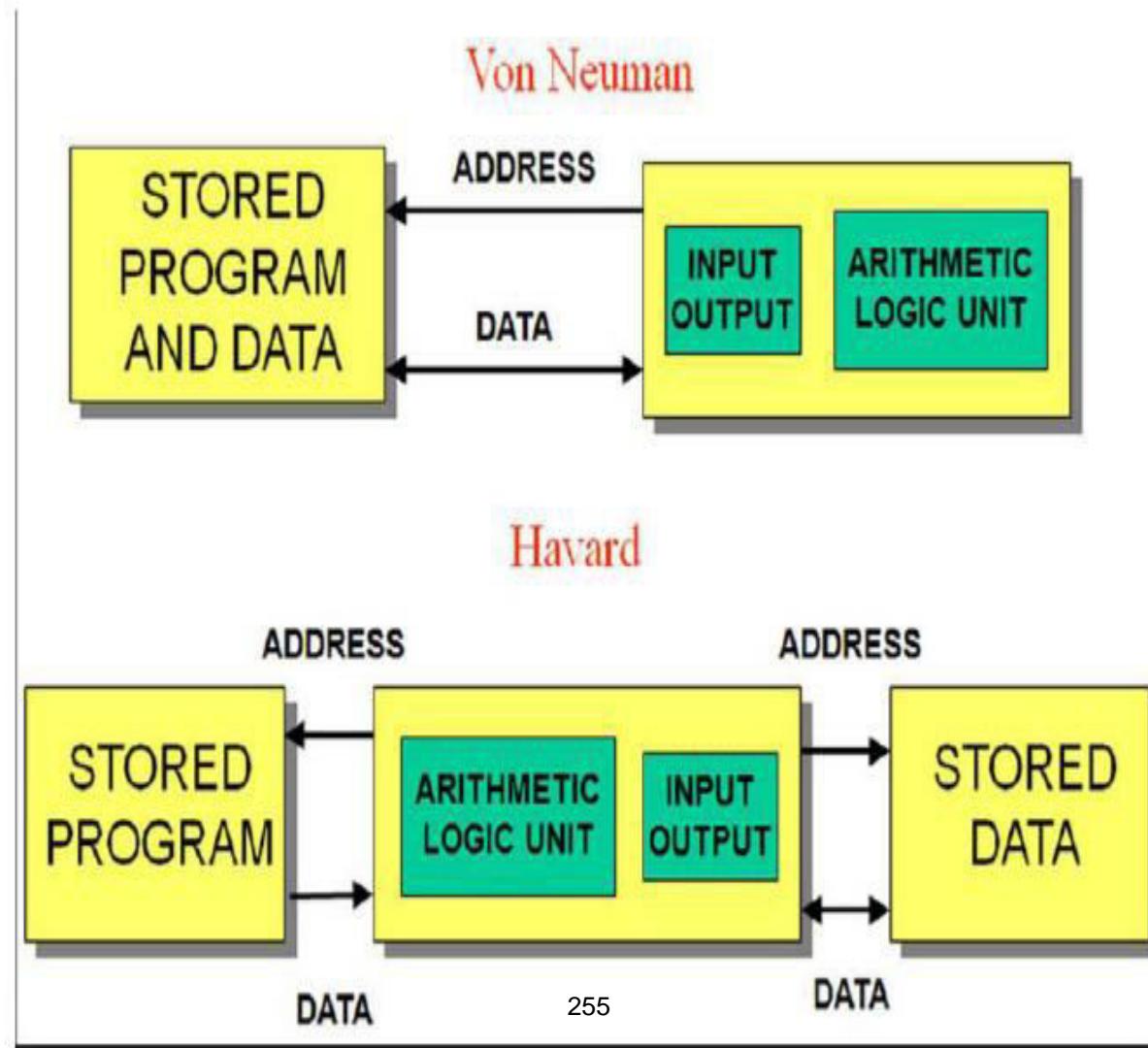
Harvard architecture

- The **Harvard architecture** is a computer architecture with physically separate storage
- These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data.
- Programs needed to be loaded by an operator; the processor could not initialize itself.

Harvard Architecture

- In a Harvard architecture, there is no need to make the two memories share characteristics.
- In particular, the word width, timing, implementation technology, and memory address structure can differ.
- Instructions – read-only memory
- Data memory – read-write memory.
- In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.
- Ex: Mark1

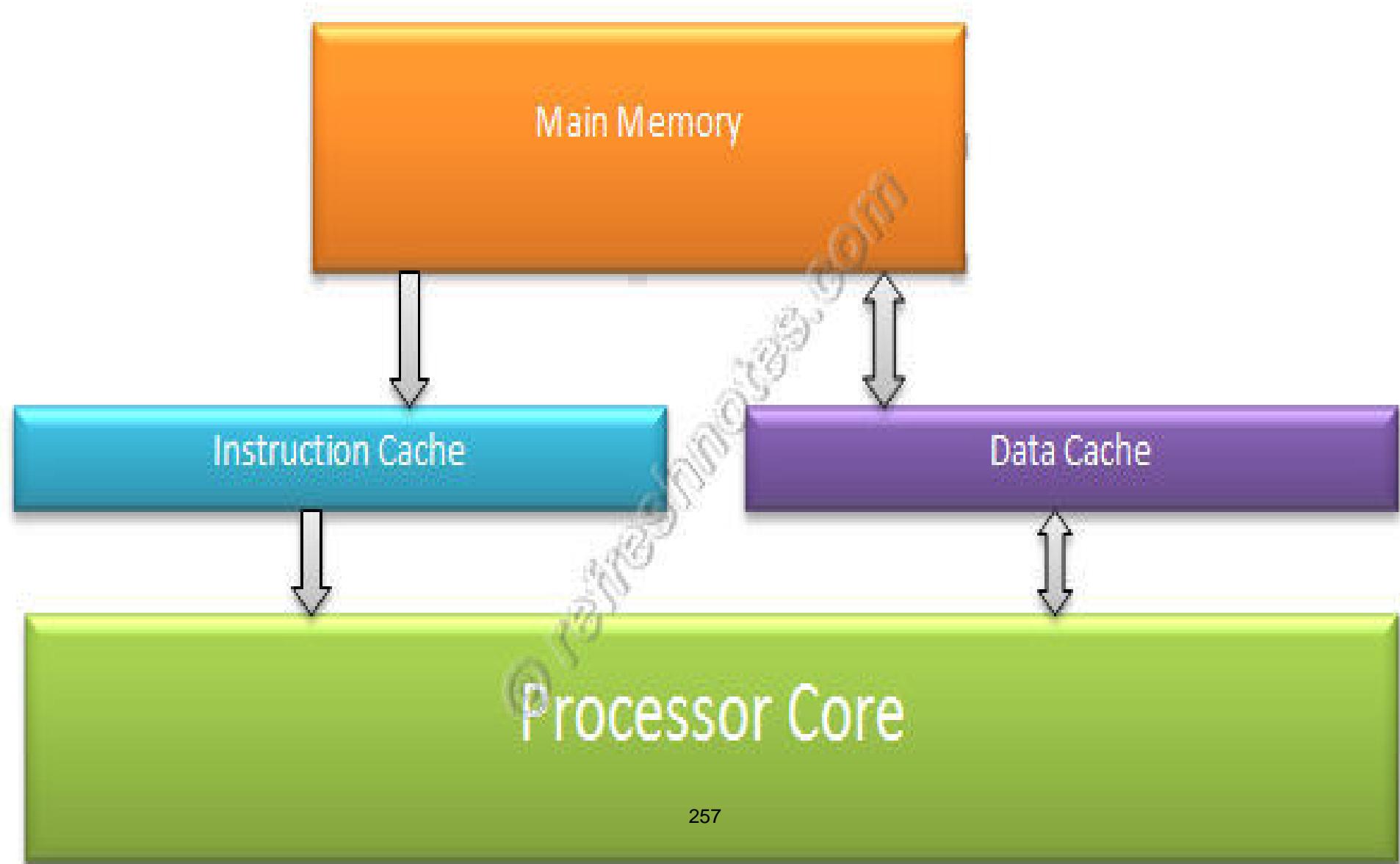
Von-Neumann vs Harvard Architecture



Von-Neumann vs Harvard Architecture

Von-Neumann Architecture	Harvard Architecture
The data and program are stored in the same memory	The data and program memory are separate. Both memories can use different sizes.
The code is executed serially and takes more clock cycles.	The code is executed in parallel
The programs can be optimized in lesser size	The programs tend to grow big in size
Parallel access to data and program is not possible.	Two memories with two buses allow parallel access to data access and instructions
One bus is simpler for the control unit design	Control unit for 2 buses is more complicated and more expensive
Error in program can rewrite instruction and crash program execution	Program can't write itself

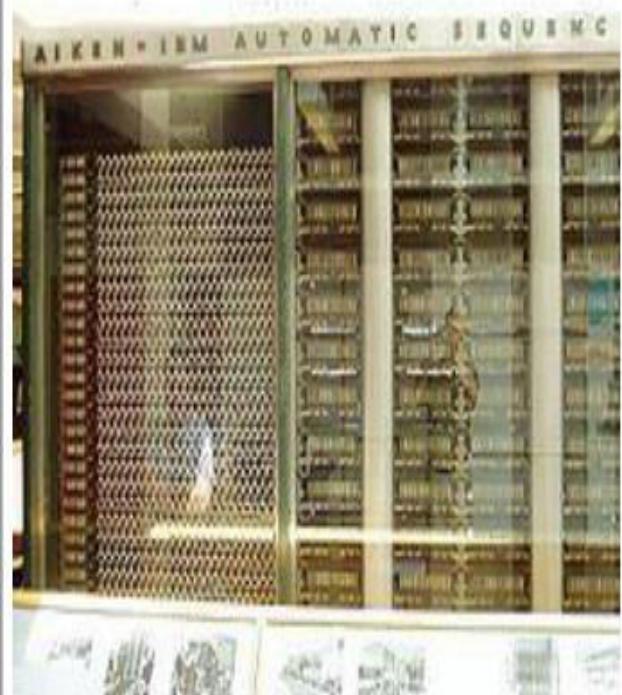
Modified Harvard Architecture



Modified Harvard Architecture

- A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses.
- The most common modification includes separate instruction and data caches backed by a common address space.
- While the CPU executes from cache, it acts as a pure Harvard machine.
- When accessing backing memory, it acts like a von Neumann machine
- Ex: x86 processors

Harvard Mark1



The left end consisted of electromechanical computing components

The right end included data and program readers, and automatic typewriters

Closeup of input/output and control readers

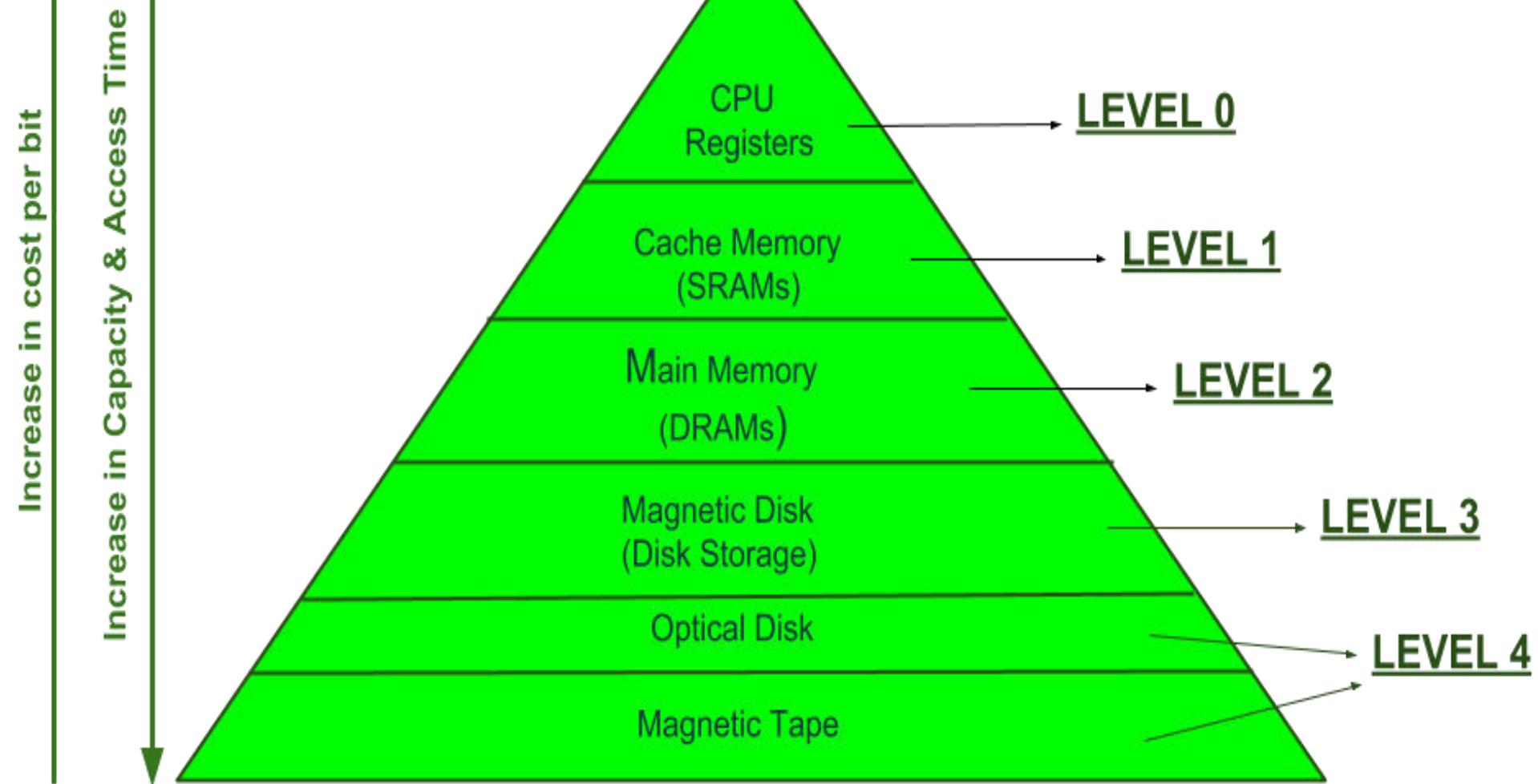
Registers and Register File

Outline

- Registers
 - User Visible Register
 - Control and Status register
- Register Files

Introduction

- CPU must have some working space (temporary storage) called registers
- Number and function vary between processor designs
- Top level of memory hierarchy



MEMORY HIERARCHY DESIGN

- User visible register
 - Used by the Programmer
 - To minimize the memory reference by optimizing the use of registers.
- Control and Status register
 - Used by:
 - The control unit to control the operations of the processor
 - The OS to control execution of programs.

User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

General Purpose Registers

- May be true general purpose
- May be restricted-(dedicated registers-floating point or stack)
- Make them general purpose
 - Increase flexibility and programmer options
- Make them specialized
 - Smaller (faster) instructions
 - Less flexibility

How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More-does not reduce memory references

How big to be the GP register?

- Large enough to hold full address
- Large enough to hold full word
- Often possible to combine two data registers

Special Purpose Registers

- Data
 - Accumulator, MQ (IAS Machine)
- Addressing
 - Segment pointer, Index, stack pointer

Control Registers

- Not visible to the user
- May be visible in a control or operating system mode (supervisor mode)
- Registers essential to instruction execution:
 - Program Counter (PC)
 - Instruction Register (IR)-Contains the inst most recently fetched
 - Memory Address Register (MAR) – contains the addr of loc in mem
 - Memory Buffer Register (MBR) – contains a word of data to be written to mem or the word most recently read

Condition Code Registers

- Bits set by hardware processor as a result of some operations.
- Sets of individual bits
 - e.g. result of last operation was zero
- Can be read (implicitly) by programs
 - e.g. Jump if zero –simplifies branch taking.

Program Status Word(PSW)

- Registers or set of register is known as PSW
- Contains status information
- Includes Condition Codes
 - Sign of last result
 - Zero
 - Carry
 - Equal
 - Overflow
 - Interrupt enable/disable
 - Supervisor

Example Register Organizations

Data Registers	
D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	

Address Registers	
A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	
A7'	

Program Status	
Program Counter	
Status Register	

(a) MC68000

General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

Program Status

Instr Ptr
Flags

(b) 8086

General Registers

EAX	AX
EBX	BX
ECX	CX
EDX	DX

ESP	SP
EBP	BP
ESI	SI
EDI	DI

Program Status

FLAGS Register
Instruction Pointer

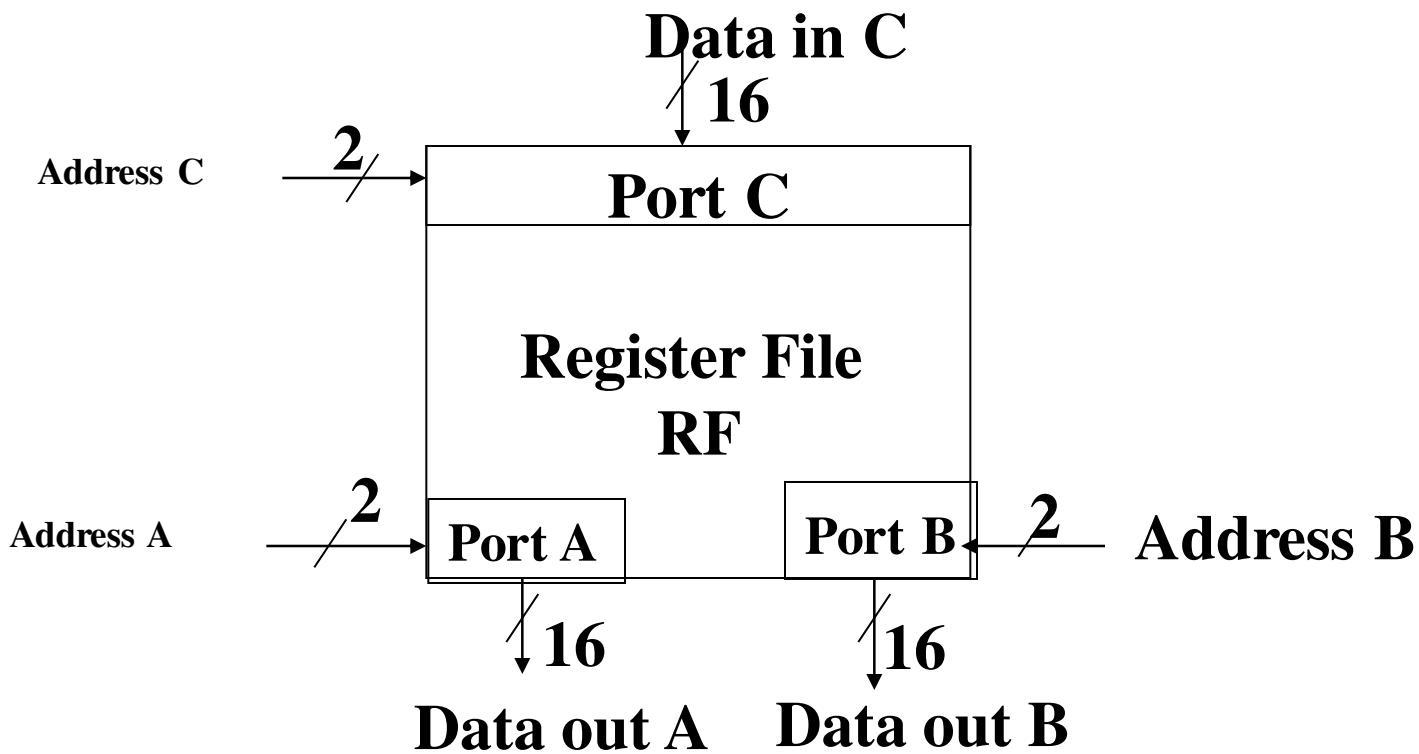
(c) 80386 - Pentium II

Understatement: There is no universally accepted philosophy for organizing the processor registers

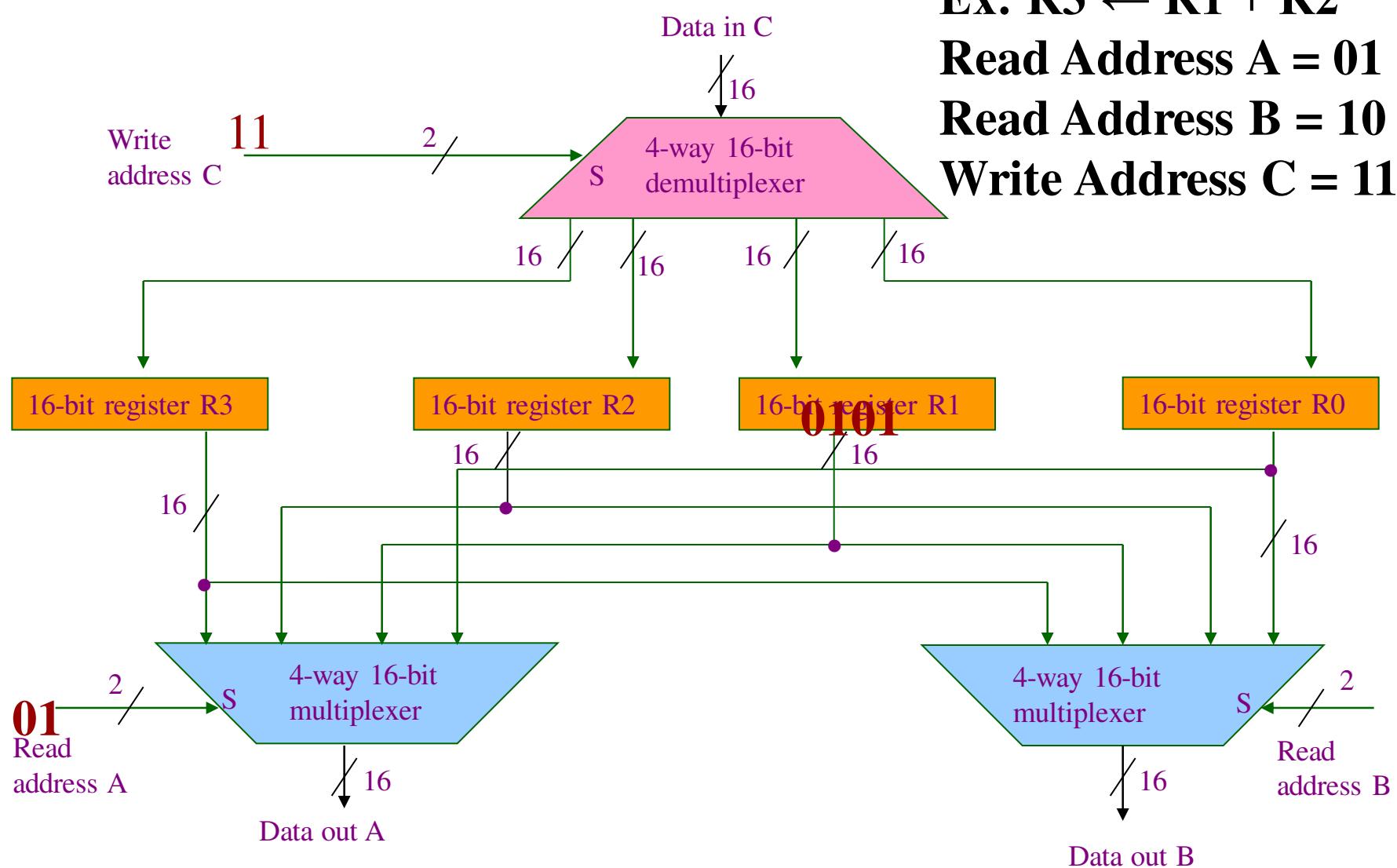
Register Files (RF)

- Set of general purpose registers.
- It functions as small RAM and implemented using fast RAM technology.
- RF needs several access ports for simultaneously reading from or writing to several different registers. Hence RF is realized as **multiport RAM**.
- A standard RAM has just one access port with an associated address bus and data bus.

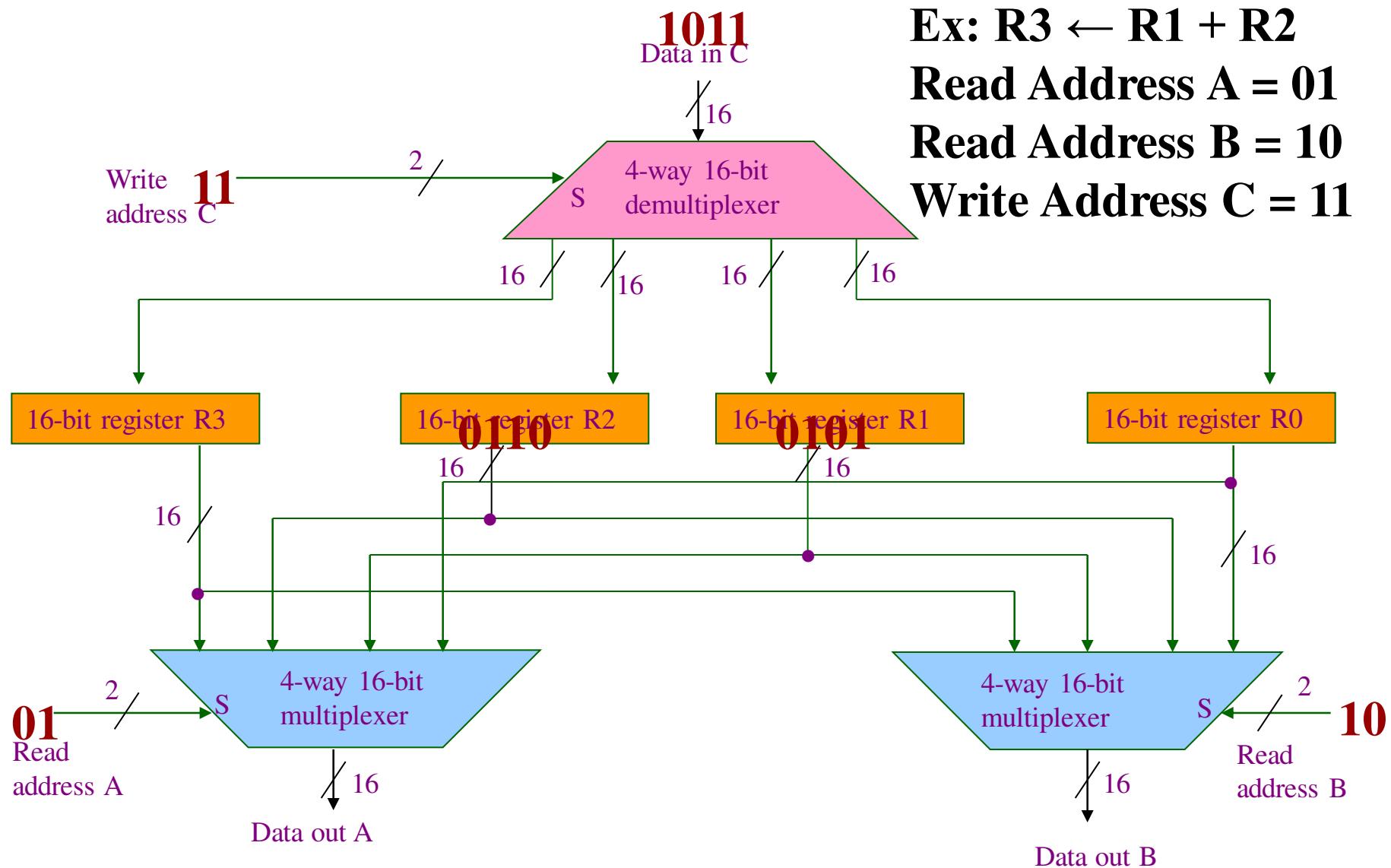
A register file with three access ports - symbol



A Register File with three access ports – logic diagram



A Register File with three access ports – logic diagram



Exercise 1:

If 8 registers are used

- How many bits are needed for read/write address?
- What is the size of the de-multiplexer and multiplexer required?
- If 4 multiplexers are used, how many parallel reads can be performed?
- Give an example with 4 parallel reads and 1 write.
- List all types of registers for the processor MC6800 and explain them briefly.
- Ref: Vincent .P. Heuring, Harry F. Jordan “ Computer System design and Architecture” Pearson, 2nd Edition, 2003.

Exercise 2: Draw your design of a register file:

- Three registers, each is 2-bits wide
 - Two source buses, one destination bus
- How many & what size:
 - Muxes did you use?
 - Demuxes did you use?

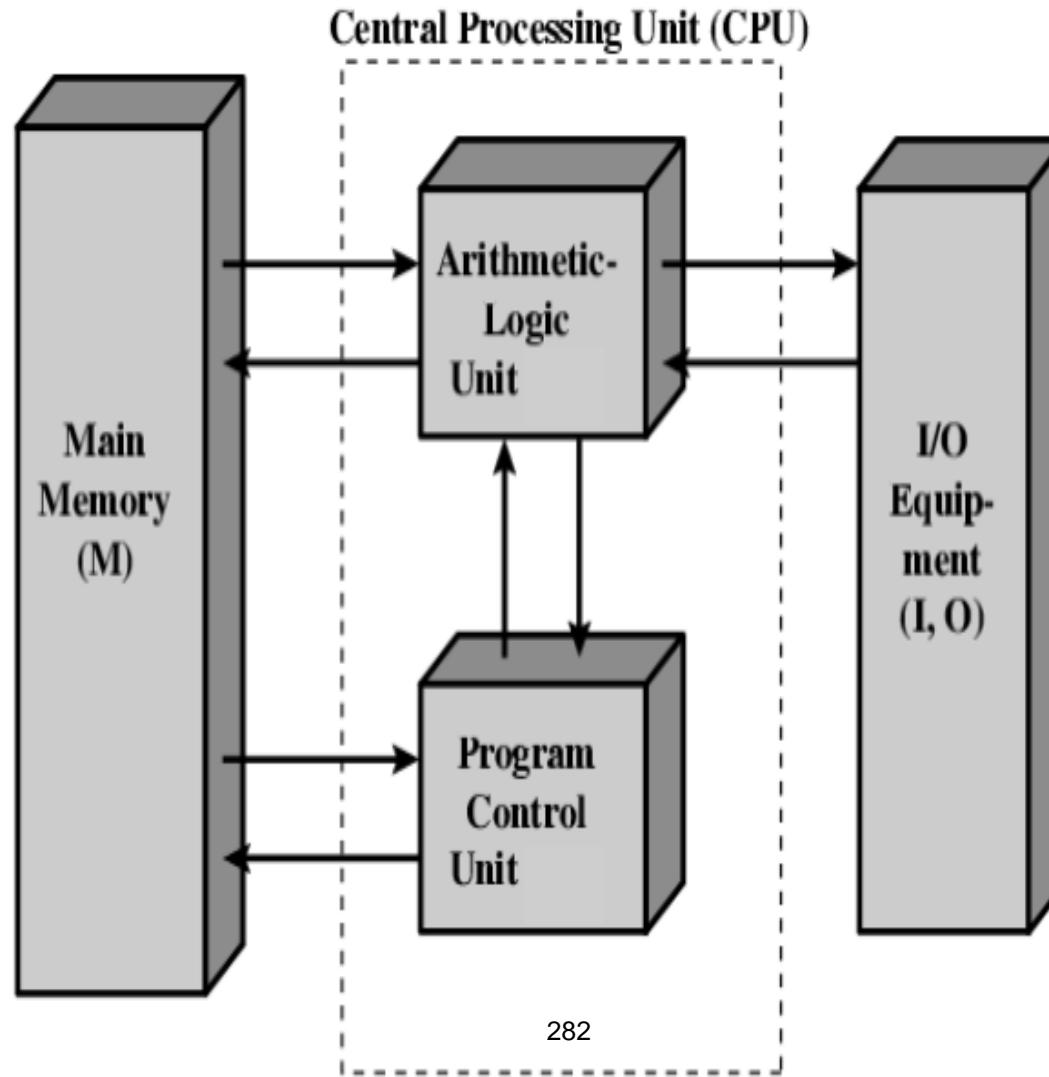
References

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- J. P. Hayes, Computer system architecture, McGraw Hill

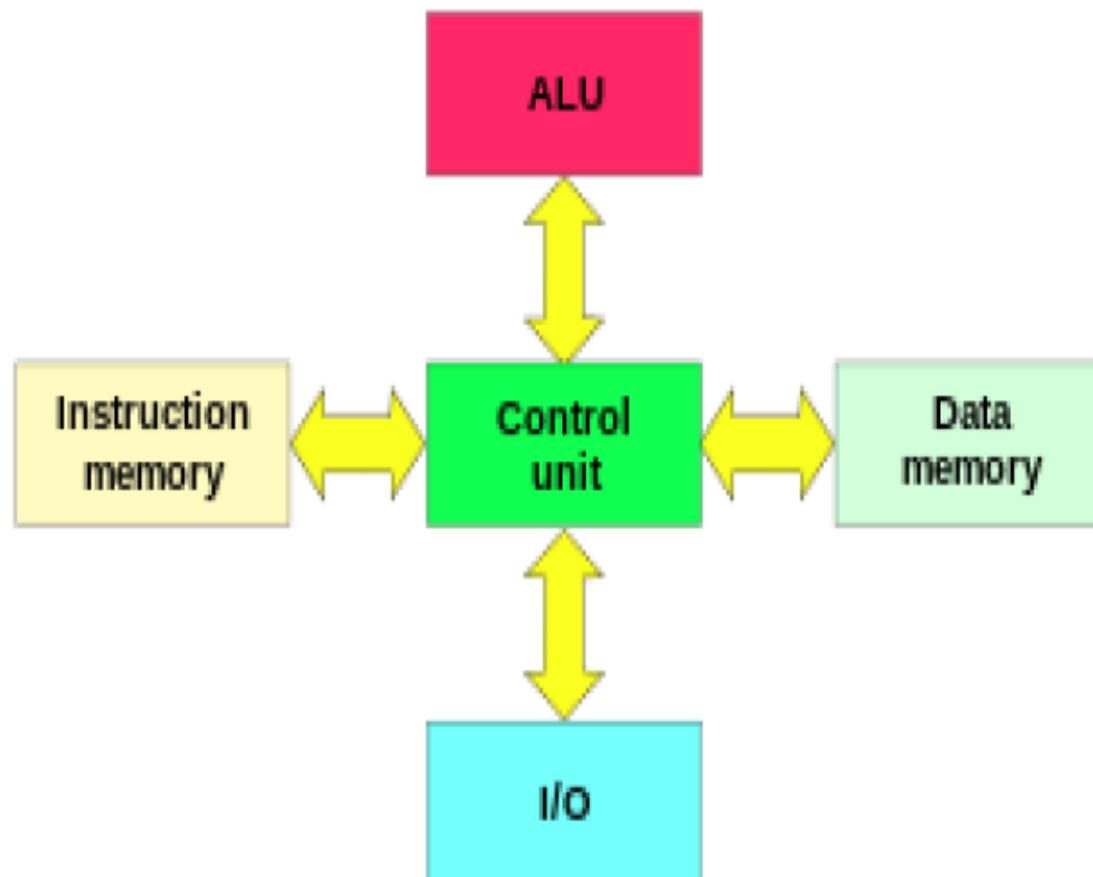
Von-Neumann and Harvard Architecture

LIJO V P
SCOPE
VIT, VELLORE

Structure of Von Neumann Machine



Harvard Architecture



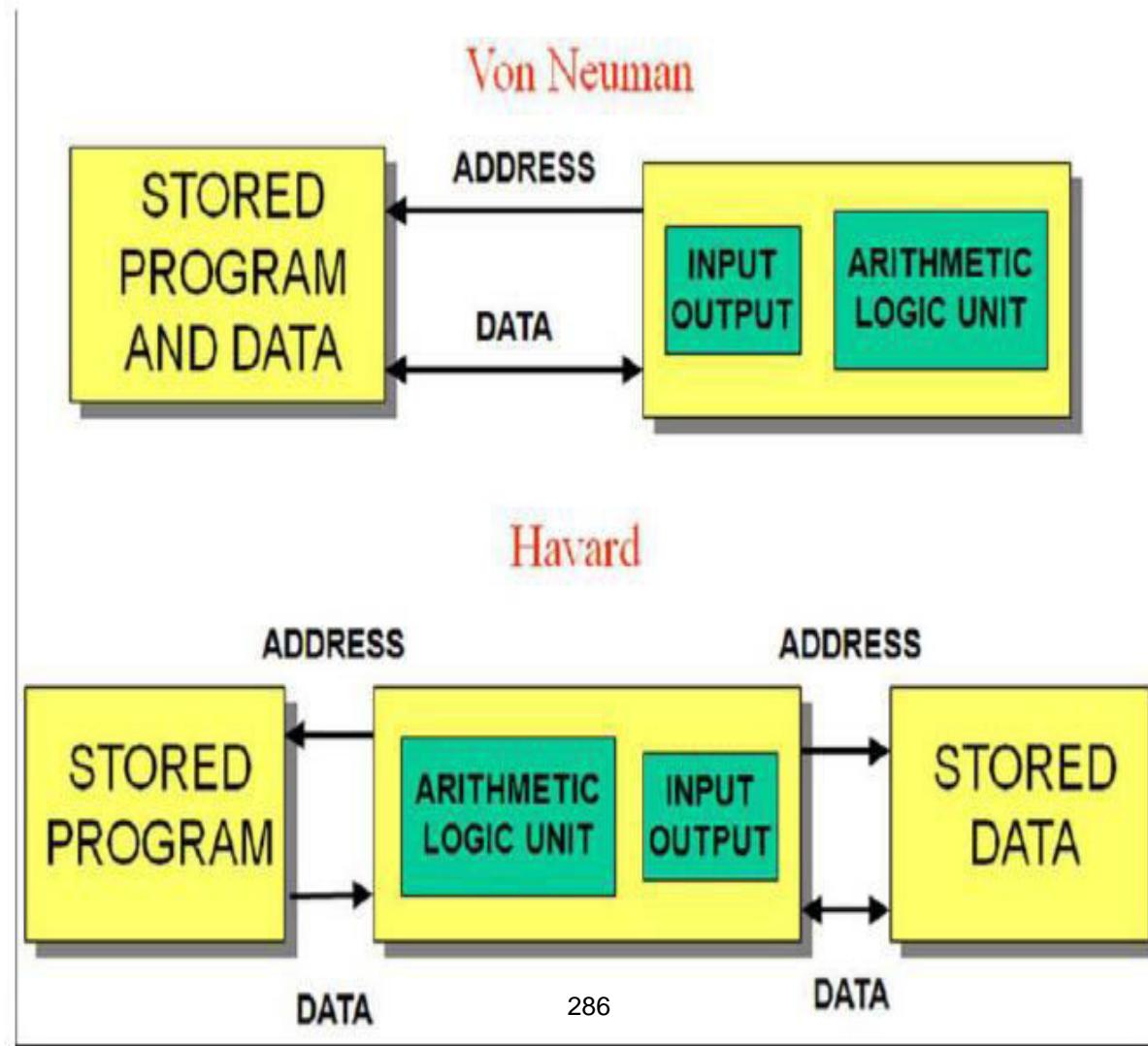
Harvard architecture

- The **Harvard architecture** is a computer architecture with physically separate storage
- These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data.
- Programs needed to be loaded by an operator; the processor could not initialize itself.

Harvard Architecture

- In a Harvard architecture, there is no need to make the two memories share characteristics.
- In particular, the word width, timing, implementation technology, and memory address structure can differ.
- Instructions – read-only memory
- Data memory – read-write memory.
- In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.
- Ex: Mark1

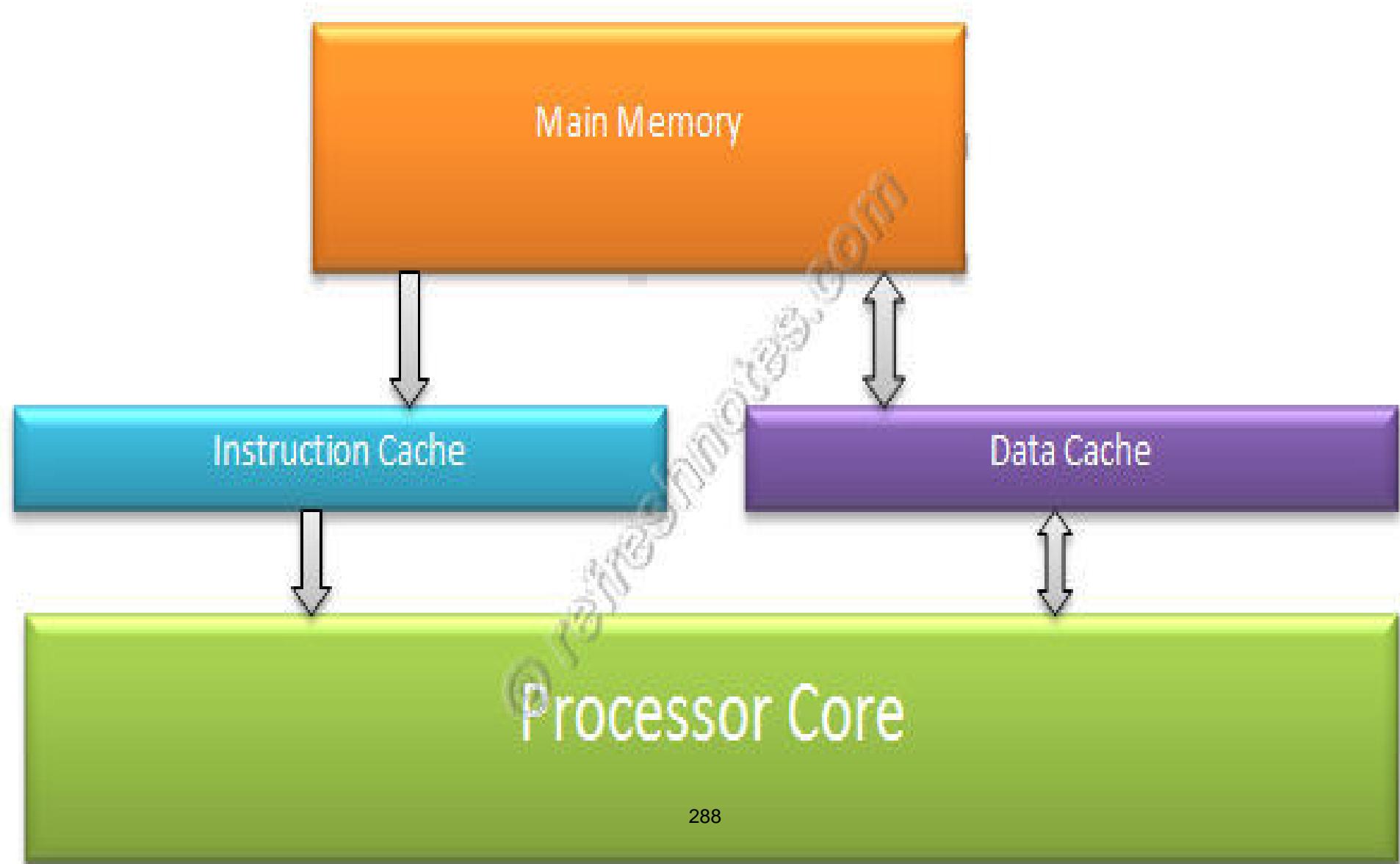
Von-Neumann vs Harvard Architecture



Von-Neumann vs Harvard Architecture

Von-Neumann Architecture	Harvard Architecture
The data and program are stored in the same memory	The data and program memory are separate. Both memories can use different sizes.
The code is executed serially and takes more clock cycles.	The code is executed in parallel
The programs can be optimized in lesser size	The programs tend to grow big in size
Parallel access to data and program is not possible.	Two memories with two buses allow parallel access to data access and instructions
One bus is simpler for the control unit design	Control unit for 2 buses is more complicated and more expensive
Error in program can rewrite instruction and crash program execution	Program can't write itself

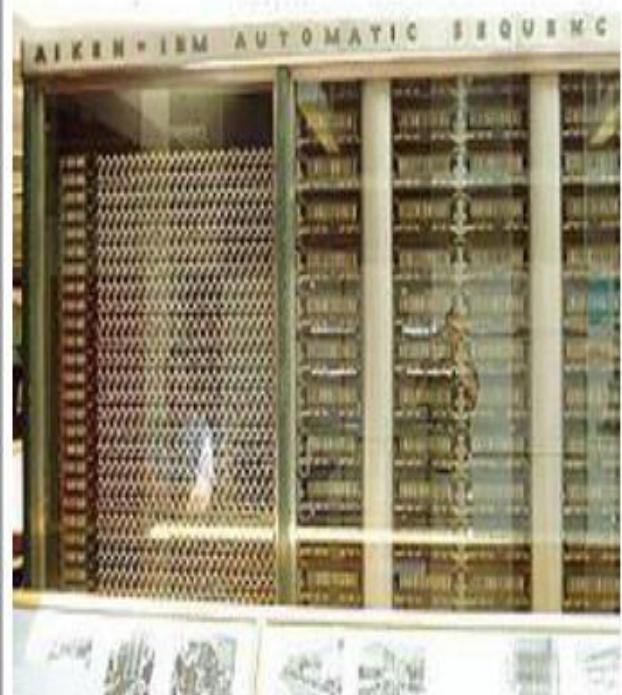
Modified Harvard Architecture



Modified Harvard Architecture

- A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses.
- The most common modification includes separate instruction and data caches backed by a common address space.
- While the CPU executes from cache, it acts as a pure Harvard machine.
- When accessing backing memory, it acts like a von Neumann machine
- Ex: x86 processors

Harvard Mark1



The left end consisted of electromechanical computing components

The right end included data and program readers, and automatic typewriters

Closeup of input/output and control readers

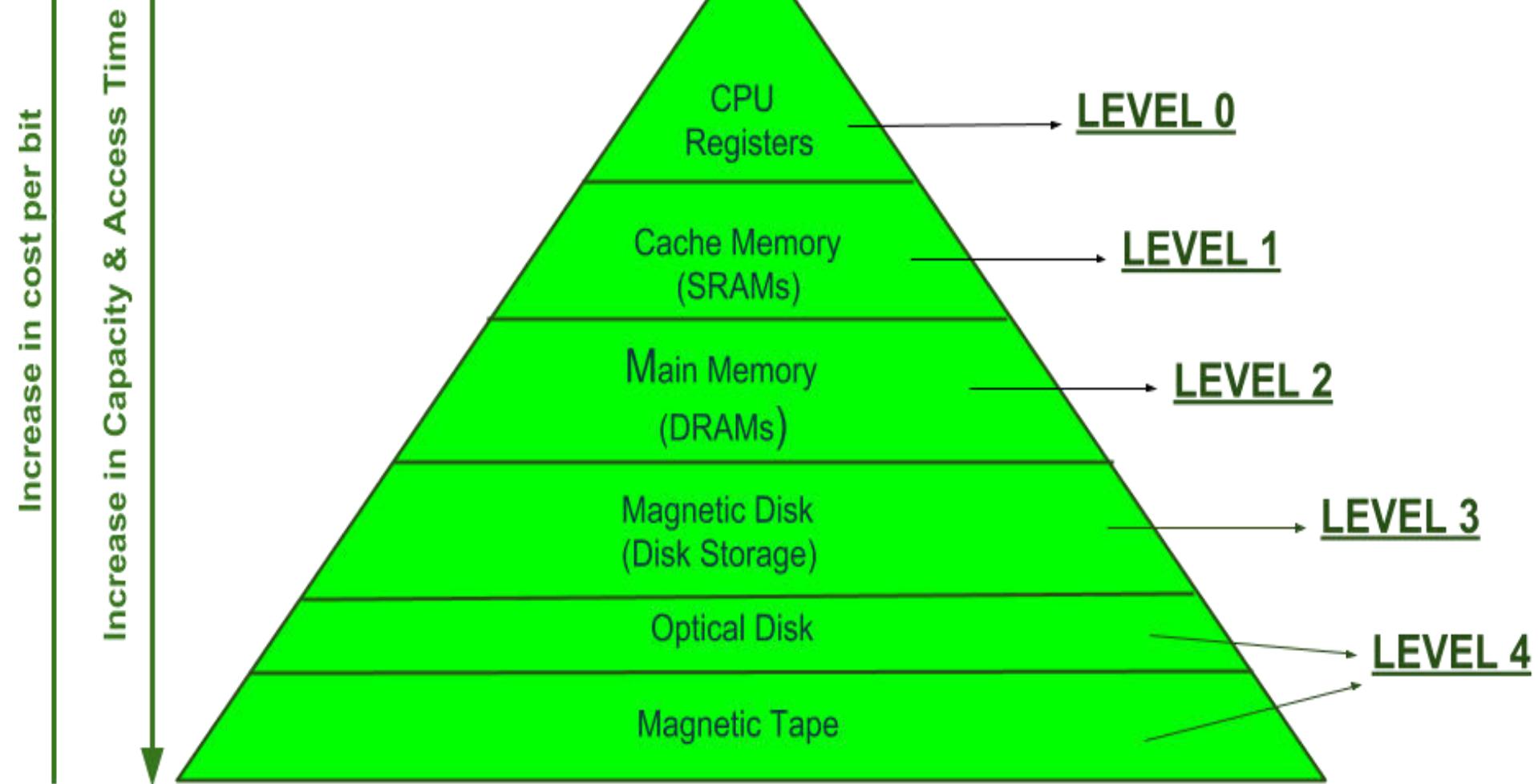
Registers and Register File

Outline

- Registers
 - User Visible Register
 - Control and Status register
- Register Files

Introduction

- CPU must have some working space (temporary storage) called registers
- Number and function vary between processor designs
- Top level of memory hierarchy



MEMORY HIERARCHY DESIGN

- User visible register
 - Used by the Programmer
 - To minimize the memory reference by optimizing the use of registers.
- Control and Status register
 - Used by:
 - The control unit to control the operations of the processor
 - The OS to control execution of programs.

User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

General Purpose Registers

- May be true general purpose
- May be restricted-(dedicated registers-floating point or stack)
- Make them general purpose
 - Increase flexibility and programmer options
- Make them specialized
 - Smaller (faster) instructions
 - Less flexibility

How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More-does not reduce memory references

How big to be the GP register?

- Large enough to hold full address
- Large enough to hold full word
- Often possible to combine two data registers

Special Purpose Registers

- Data
 - Accumulator, MQ (IAS Machine)
- Addressing
 - Segment pointer, Index, stack pointer

Control Registers

- Not visible to the user
- May be visible in a control or operating system mode (supervisor mode)
- Registers essential to instruction execution:
 - Program Counter (PC)
 - Instruction Register (IR)-Contains the inst most recently fetched
 - Memory Address Register (MAR) – contains the addr of loc in mem
 - Memory Buffer Register (MBR) – contains a word of data to be written to mem or the word most recently read

Condition Code Registers

- Bits set by hardware processor as a result of some operations.
- Sets of individual bits
 - e.g. result of last operation was zero
- Can be read (implicitly) by programs
 - e.g. Jump if zero –simplifies branch taking.

Program Status Word(PSW)

- Registers or set of register is known as PSW
- Contains status information
- Includes Condition Codes
 - Sign of last result
 - Zero
 - Carry
 - Equal
 - Overflow
 - Interrupt enable/disable
 - Supervisor

Example Register Organizations

Data Registers	
D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	

Address Registers	
A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	
A7'	

Program Status	
Program Counter	
Status Register	

(a) MC68000

General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

Program Status

Instr Ptr
Flags

(b) 8086

General Registers

EAX	AX
EBX	BX
ECX	CX
EDX	DX

ESP	SP
EBP	BP
ESI	SI
EDI	DI

Program Status

FLAGS Register
Instruction Pointer

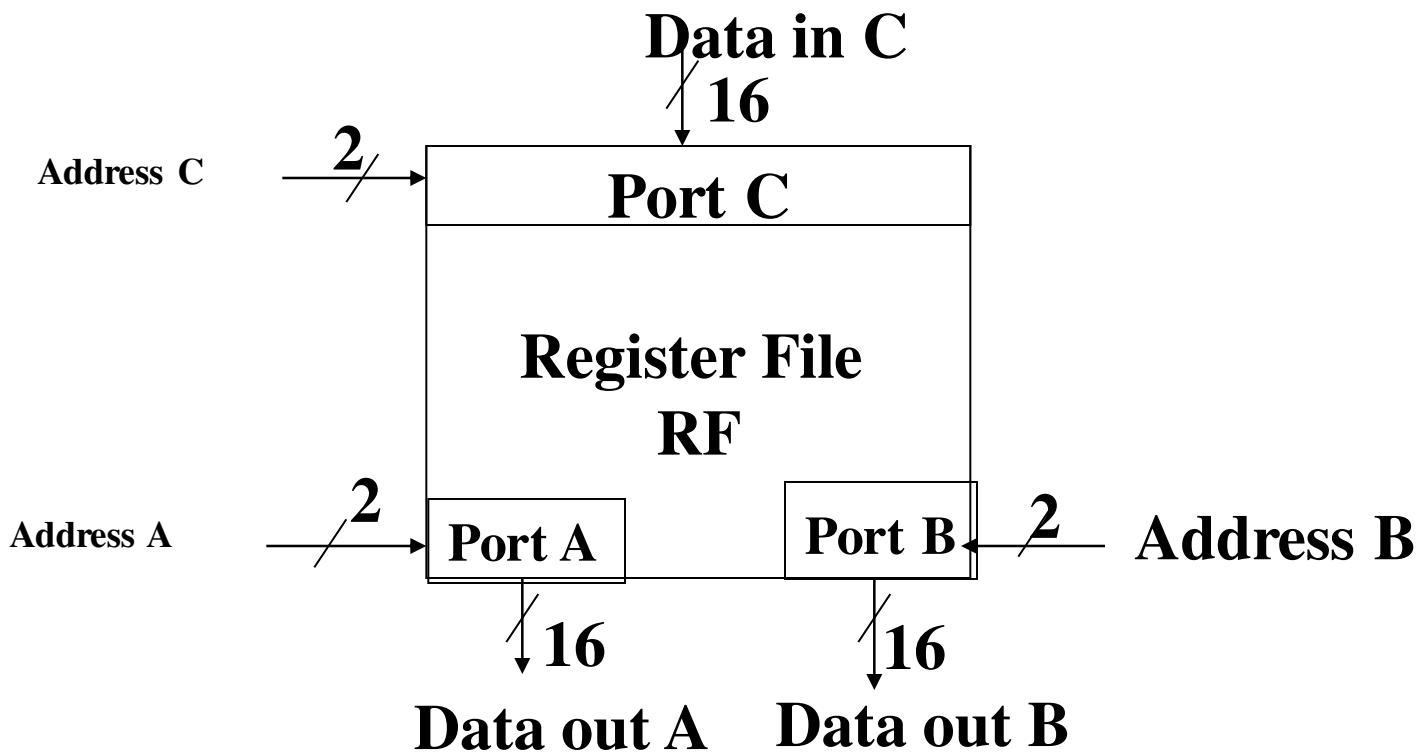
(c) 80386 - Pentium II

Understatement: There is no universally accepted philosophy for organizing the processor registers

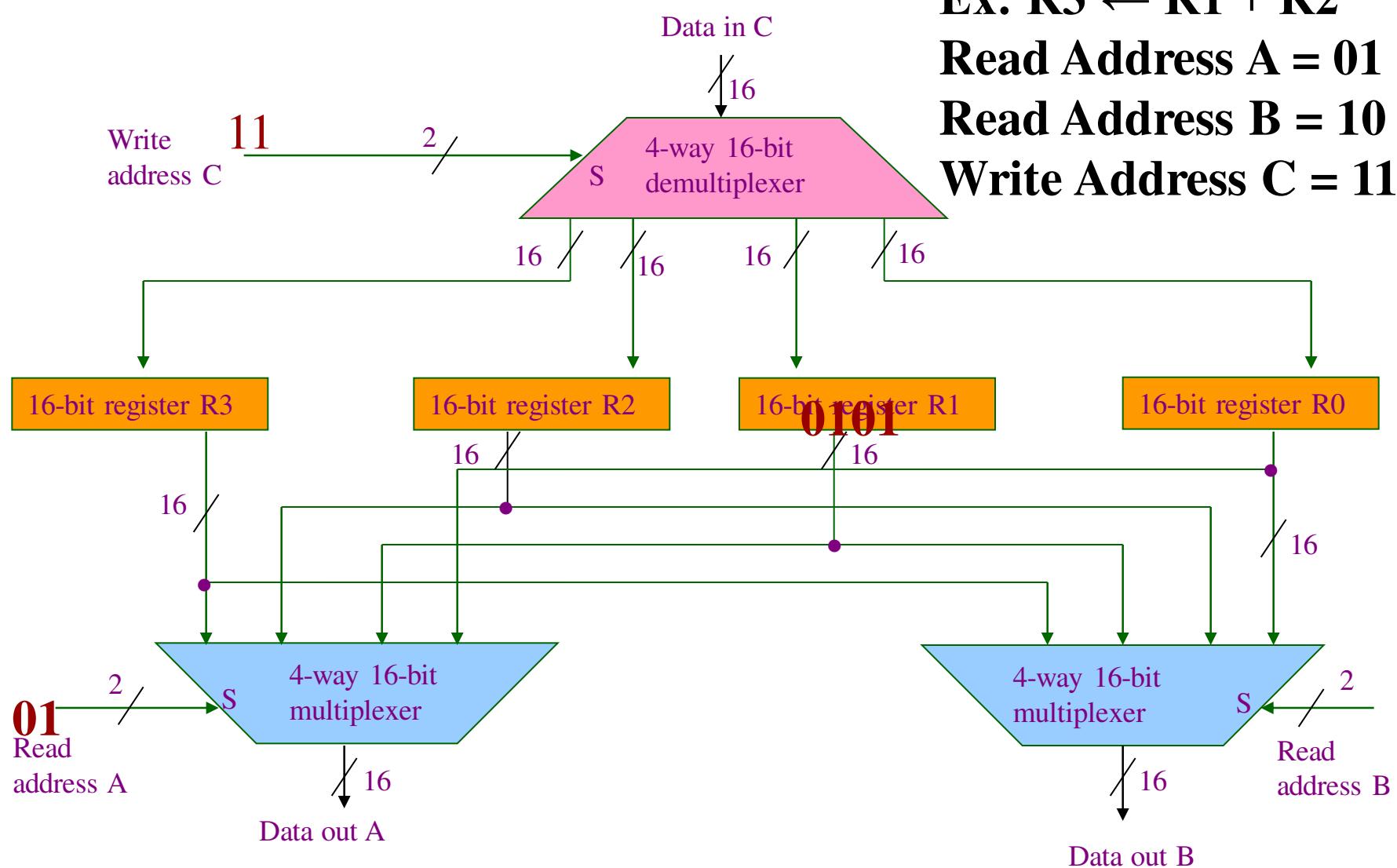
Register Files (RF)

- Set of general purpose registers.
- It functions as small RAM and implemented using fast RAM technology.
- RF needs several access ports for simultaneously reading from or writing to several different registers. Hence RF is realized as **multiport RAM**.
- A standard RAM has just one access port with an associated address bus and data bus.

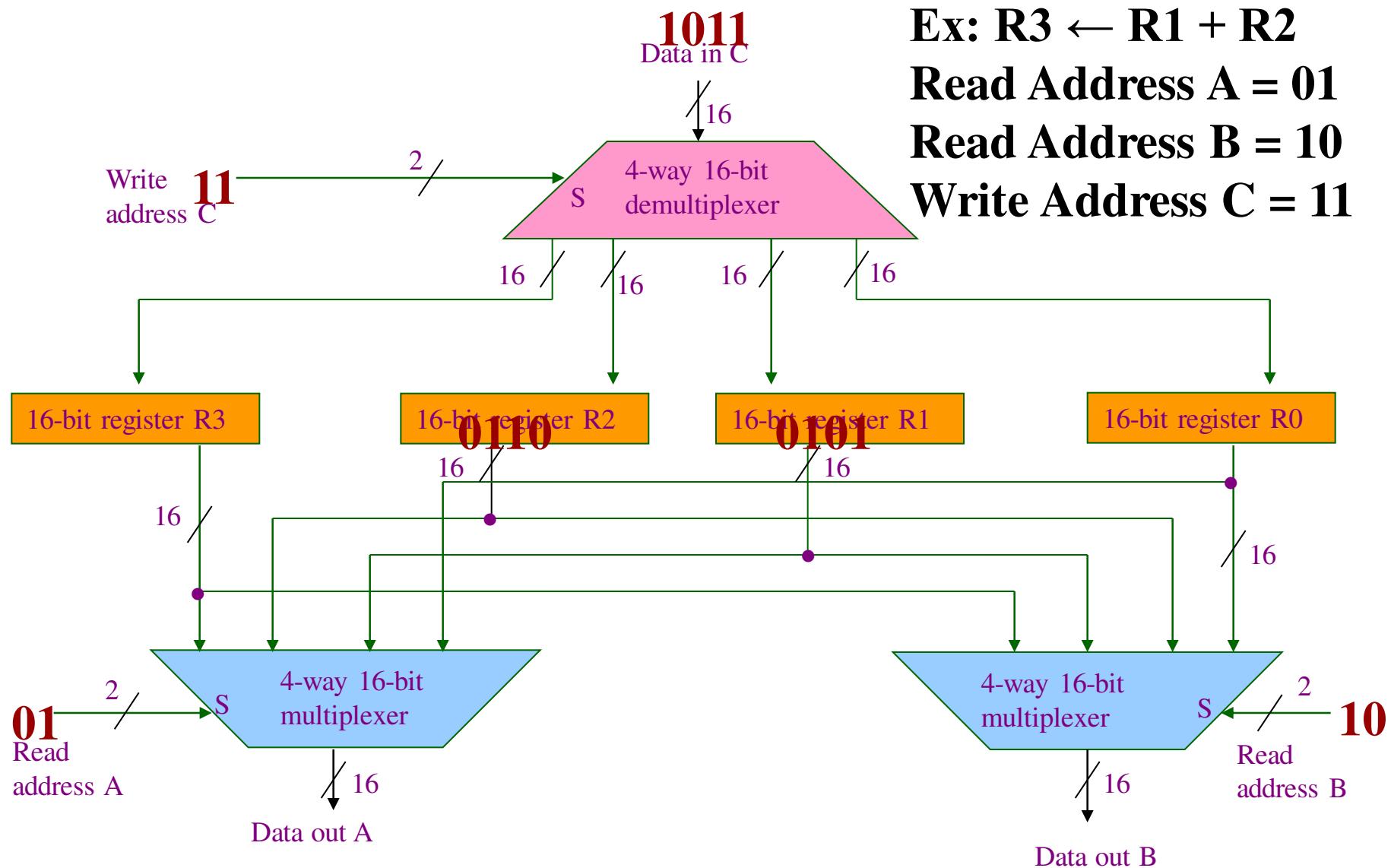
A register file with three access ports - symbol



A Register File with three access ports – logic diagram



A Register File with three access ports – logic diagram



Exercise 1:

If 8 registers are used

- How many bits are needed for read/write address?
- What is the size of the de-multiplexer and multiplexer required?
- If 4 multiplexers are used, how many parallel reads can be performed?
- Give an example with 4 parallel reads and 1 write.
- List all types of registers for the processor MC6800 and explain them briefly.
- Ref: Vincent .P. Heuring, Harry F. Jordan “ Computer System design and Architecture” Pearson, 2nd Edition, 2003.

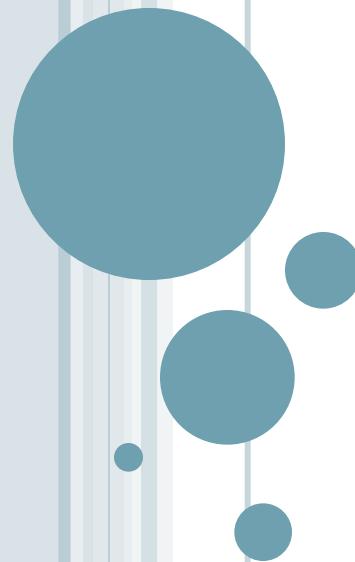
Exercise 2: Draw your design of a register file:

- Three registers, each is 2-bits wide
 - Two source buses, one destination bus
- How many & what size:
 - Muxes did you use?
 - Demuxes did you use?

References

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- J. P. Hayes, Computer system architecture, McGraw Hill

SUBROUTINE CALL & RETURN



LIJO V. P
SCOPE
VIT, VELLORE

SUBROUTINE

- Subroutine is a self-contained sequence of instructions that performs a given computational task.
- Call Subroutine.
- When called,
 - Push [PC] to TOS
 - $[PC] \leftarrow$ 1st address of subroutine
 - Execute SUB
 - Finally execute RET instruction
 - Pop TOS to PC
 - Continue with the instruction which is next to SUB call.

LOCATIONS TO STORE THE RETURN ADDRESS

- First memory location of the subroutine
- Fixed location in memory
- Processor registers
- Memory stack – best option
 - Adv: In the case of sequential calls to subroutines. So, the top of the stack always has the return address of the subroutine which to be returned first.

MICRO-OPERATIONS

Call:

```
SP ← SP – 1           // decrement stack pointer  
M[SP] ← PC           // push content of PC onto the stack  
PC ← effective address /* transfer control to the subroutine */
```

Return:

```
PC ← M[SP] // pop stack and transfer to PC  
SP ← SP + 1 // increment stack pointer
```

- **Recursive SUB:-** Subroutine that calls itself
- **Subroutine nesting:-** One subroutine that calls another.
- If only one register or memory location is used to hold the return address, when subroutine is called recursively, it destroys the previous return address.
- So, stack is the good solution for this problem.

REFERENCES

Text Book

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

Addressing Modes

LIJO V P
SCOPE
VIT, VELLORE

Addressing Mode

- The Way the operands are specified in the instruction
 - Operands can be in registers, memory or embedded in the instruction
- The operation to be performed is indicated by the opcode.

Addressing Mode -Classification

- Implied Addressing Mode
- Immediate Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Register Direct Addressing Mode
- Register Indirect Addressing Mode
- Displacement or Indexed Addressing Mode
- Base Relative Addressing Mode
- Auto Increment and Auto Decrement Addressing Mode

Implied Addressing Mode

- Operand is implied / specified implicitly in the instruction
- Effective Address (EA) = AC or Stack[SP]
- CMA, CLC, STC

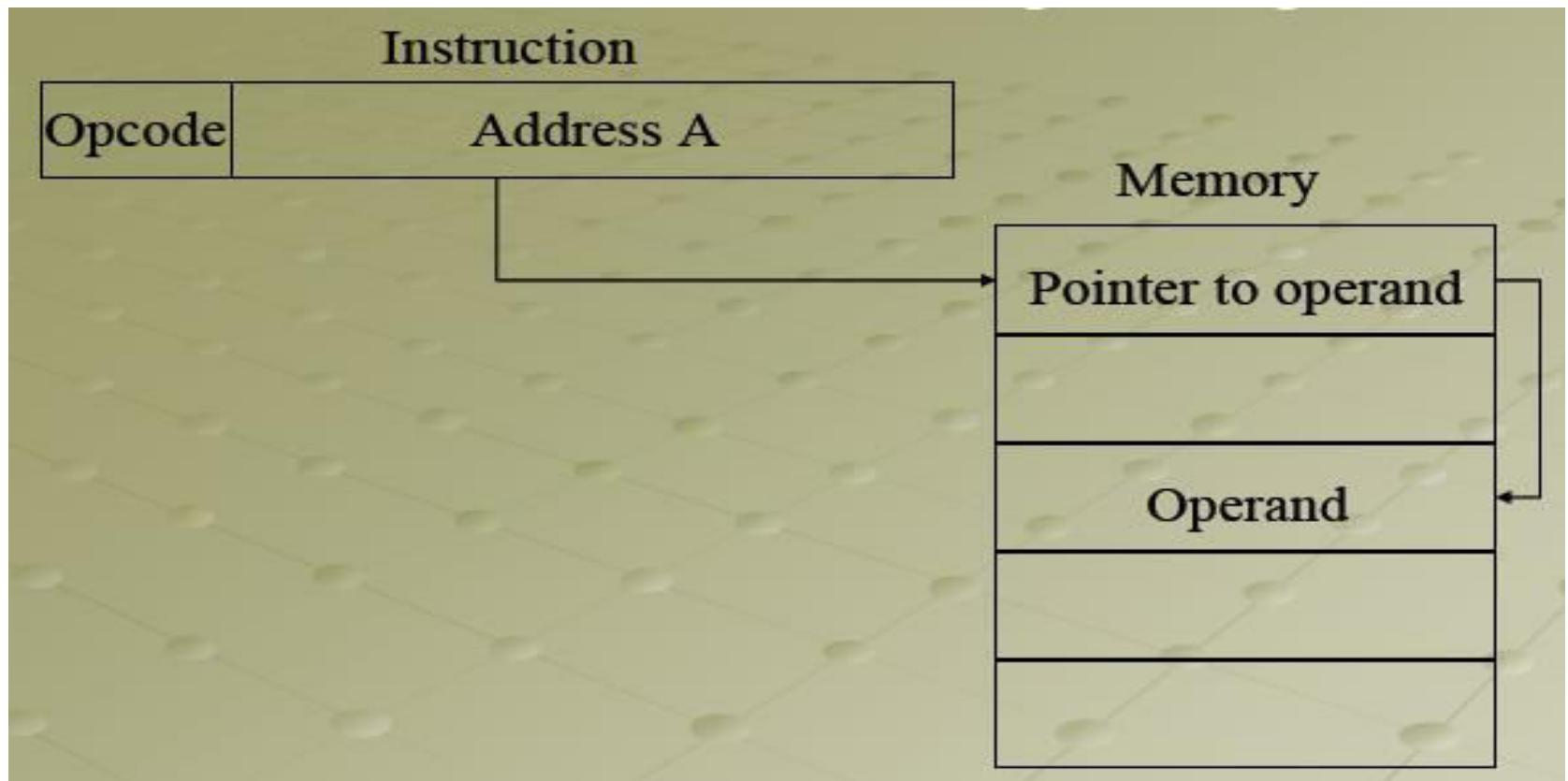
Immediate Addressing mode

- Operand embedded in the instruction
- Used during initialization
- MVI A, 23h
- No memory reference to fetch data
- Fast

Direct Addressing mode

- Effective address is a part of the instruction.
- MOV A, 3000h
- Single memory reference to access data
- No additional calculations to work out effective address

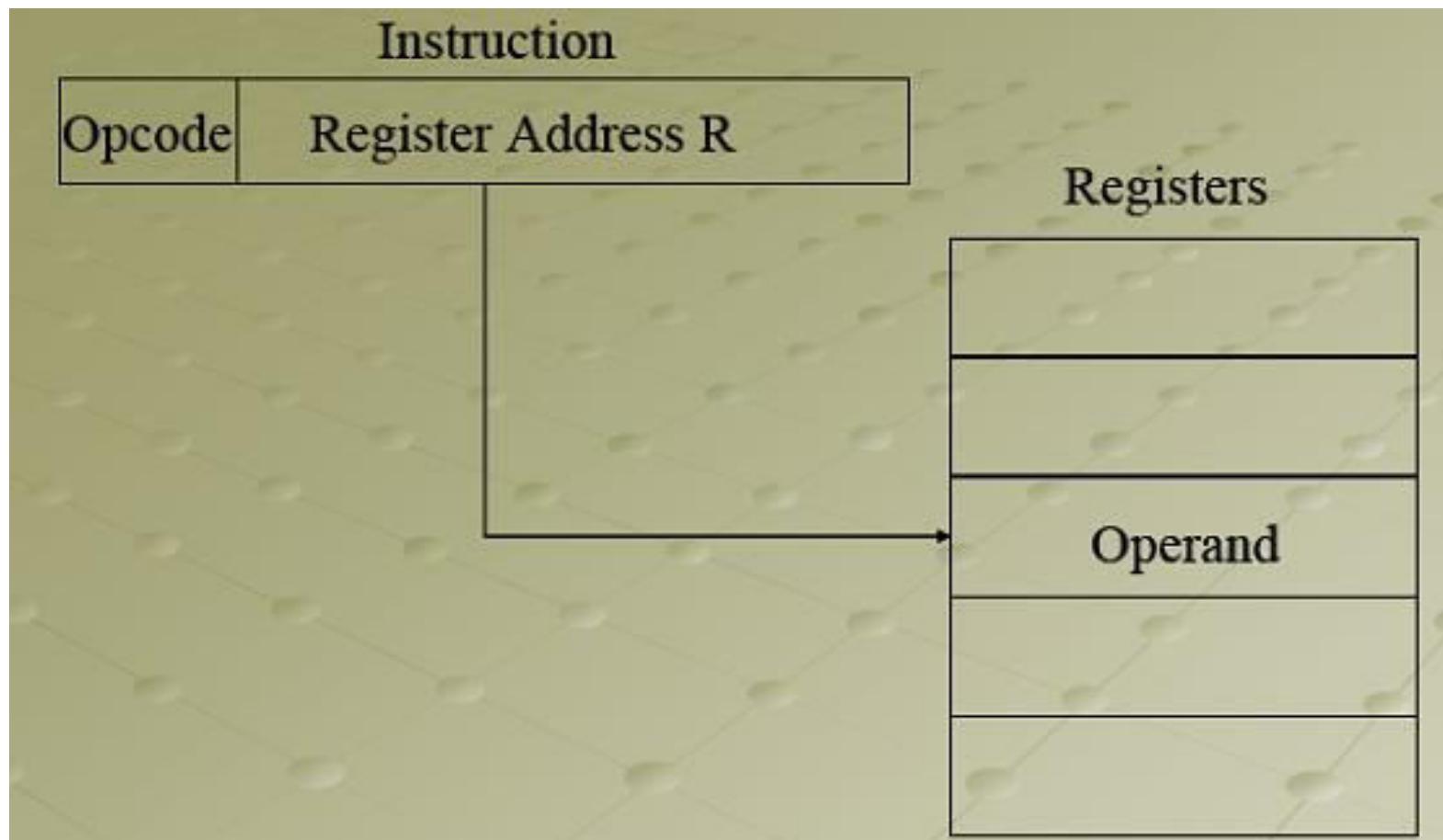
Indirect Addressing Mode Diagram



Indirect Addressing mode

- The address field of the instruction gives the address where the effective address of the operand stored in the memory.
- Ex: Move CX, [4200H]
- Multiple memory accesses to find operand
- Hence slower

Register Direct Addressing Diagram

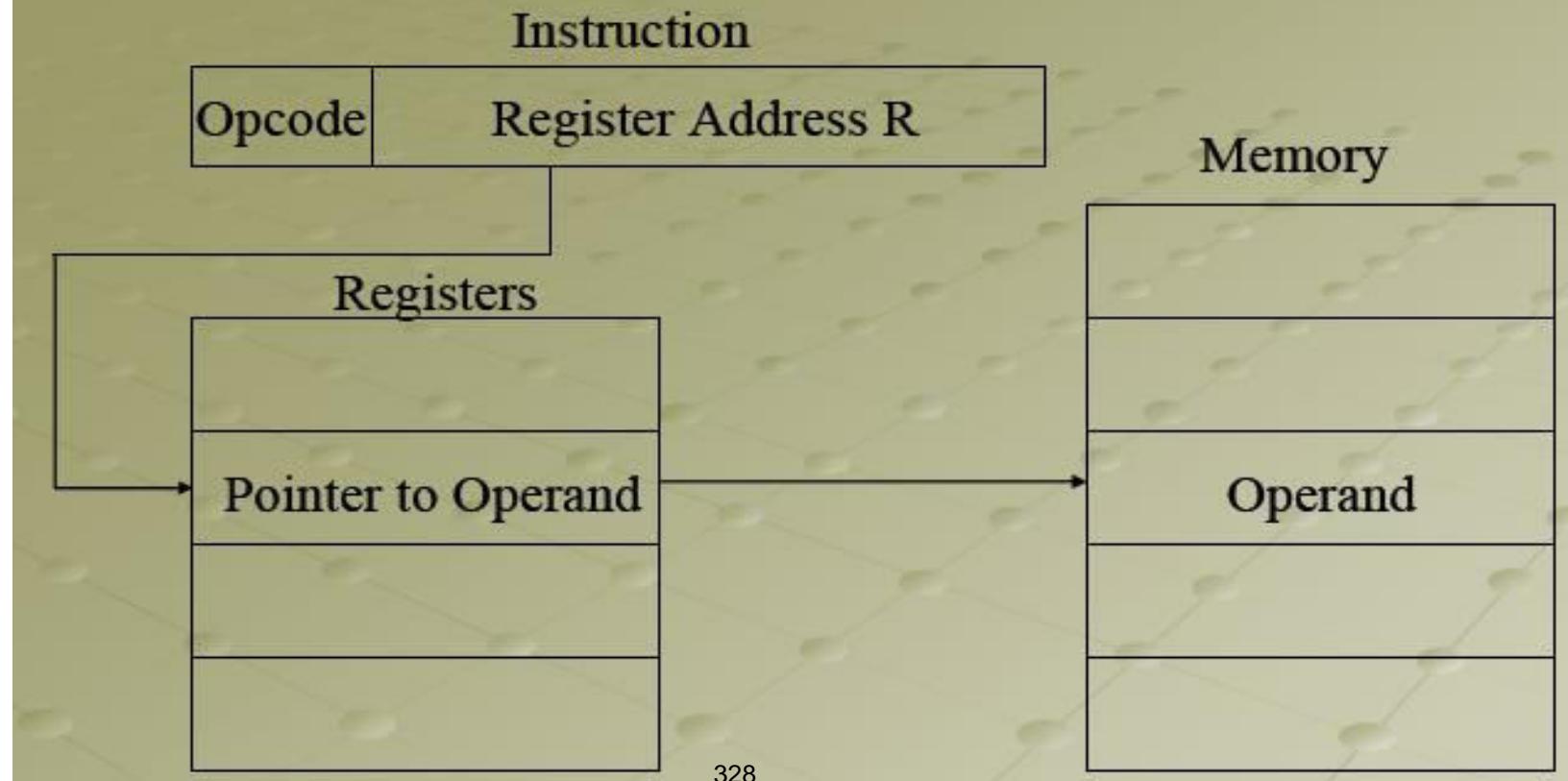


Register Direct

- Operand is in the register specified in the address part of the instruction
- $EA = R$
- Ex: `Mov AX, BX`
- Special case of direct addressing
- Faster register access.
- No memory access.

Register Indirect

Register Indirect Addressing Diagram



Register Indirect

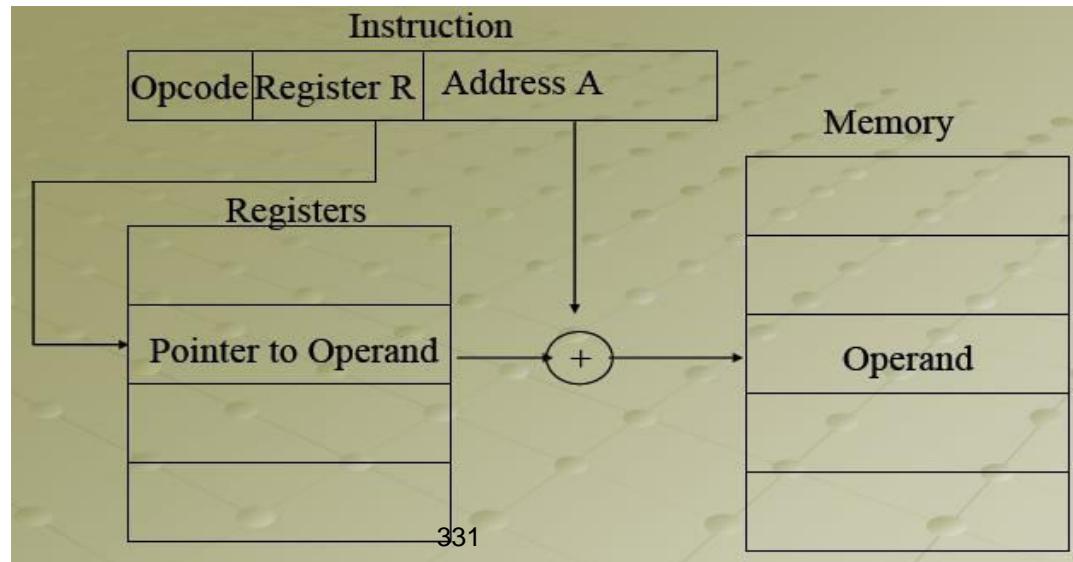
- Address part of the instruction specifies the register which gives the address of the operand in memory.
- Special case of indirect addressing
- $EA = (R)$
- Ex: Mov BX, [DX]

Auto Increment and Auto Decrement Addressing Modes

- Extension to Register indirect addressing mode.
- Register incremented or decremented after accessing memory
- Useful while transferring large chunks of contiguous data.
- Add R1, (R2)+
 - $R1 \leftarrow R1 + m[R2]$
 - $R2 \leftarrow r2 + d$; d is the size of an element.

Displacement Addressing Mode

- $EA = A + (R)$
- Add R4, 100(R1)
 - $R4 \leftarrow R4 + M[R1+100]$



Relative addressing mode

- It is often used in branch (conditional and unconditional) instructions, locality of reference and cache usage.
- **Effective address =Address part of the instruction +PC value after fetch cycle**
- JMP 24

Base register Addressing mode

- Used to facilitate the **relocation of programs in memory.**
- Uses Base register.
- **EA = Address part of the instruction +base value.**
- Only the register [base register] will be updated to reflect the beginning of a new memory segment.

Indexed Addressing mode

- Used in performing iterative operations
- Address field holds the starting address of the data array.
- Index register hold the actual index of the operand in the array.
- **EA = starting address(address part of an inst) + index register value**

Basic Addressing Modes Differences

Basic Addressing Modes differences :

	Mode	Algorithm	Advantage	Disadvantage
1	Immediate	Operand=1	No memory Reference	Limited operand magnitude
2	Direct	$EA = A$	Simple	Limited address space
3	Indirect	$EA = (A)$	Large Address space	Multiple Memory References
4	Register	$EA = R$	No memory Reference	Limited address space
5	Register Indirect	$EA = (R)$	Large address space	Extra memory space
6	Displacement	$EA = A + (R)$	Flexibility	Complexity
7	Stack	$EA = \text{Top of Stack}$	No memory Reference	Limited Applicability

Problems

- Find the effective address and the content of AC for the given data.

Address	Memory	
	Load to AC	Mode
200		
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

PC = 200
R1 = 400
XR = 100
AC

Addressing Mode	Effective Address		Content of AC
Direct Address	500	$AC \leftarrow (500)$	800
Immediate operand	201	$AC \leftarrow 500$	500
Indirect address	800	$AC \leftarrow ((500))$	300
Relative address	702	$AC \leftarrow (PC + 500)$	325
Indexed address	600	$AC \leftarrow (XR + 500)$	900
Register	-	$AC \leftarrow R1$	400
Register Indirect	400	$AC \leftarrow (R1)$	700
Autoincrement	400	$AC \leftarrow (R1)+$	700
Autodecrement	399	$AC \leftarrow -(R1)$	450

- An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate (c) relative (d) register indirect; (e) index with R1 as the index register.
- Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has the address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is
 - (a) direct (b) indirect (c) PC relative (d) indexed?

References

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- M. M. Mano, Computer System Architecture, Prentice-Hall

Processor Performance

Performance of a Processor - Instruction Execution Rate

- A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ , where $\tau = 1/f$.
- Instruction count – I_c – The number of machine instructions executed for that program until it runs to completion or for some defined time interval
- CPI - Average cycles per instruction(CPI)for a program.
- If all instructions required the same number of clock cycles, then CPI would be a constant value for a processor.
- However, on any give processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on.

Instruction Execution Rate

- i – Instruction type
- CPI_i - The number of cycles required
- I_i – The number of executed instructions of type i for a given program.
- Then we can calculate an overall CPI as follows

$$CPI = \frac{\sum_{i=1}^n (CPI_i \times I_i)}{I_c}$$

Processor Time

- The processor time T needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

- During the execution of an Instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory.
- In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time.
- We can rewrite the preceding equation as

$$T = I_c \times [p + (m \times k)] \times \tau$$

- Where,
- p – the number of processor cycles needed to decode and execute the instruction,
- m – the number of memory references needed
- k – The ratio between memory cycle time and processor cycle time

MIPS

- A common measure of performance for a processor is the rate at which instructions are executed, expressed as Millions of Instructions Per Second (MIPS), referred to as the **MIPS rate**.
- We can express the MIPS rate in terms of the clock rate and CPI as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

MIPS Example

- Consider the execution of a program which results in the execution of 2 million instructions on a 400-MHz processor.
- The instruction mix and the CPI for each instruction type are given below based on the result of a program trace experiment:

Instruction Type	CPI	Instruction Mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

$$CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24.$$

MIPS rate is $(400 \times 10^6) / (2.24 \times 10^6) \approx 178$.

MFLOPS

- Floating point performance is expressed as millions of floating-point operations per second (MFLOPS), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating-point operations in a program}}{\text{Execution time} \times 10^6}$$

Benchmarks

- Measures such as MIPS and MFLOPS have proven inadequate to evaluate the performance of processors.
- Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures.
- For example, consider this high-level language statement:

$$A = B + C$$

- With the traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

add mem(B), mem(C), mem (A)

Benchmarks

- On a typical RISC machine, the compilation would look something like this:

```
load mem(B), reg(1);
load mem(C), reg(2);
add reg(1), reg(2), reg(3);
store reg(3), mem (A)
```
- Because of the nature of the RISC architecture, both machines may execute the original high-level language instruction in about the same time.
- If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS.
- But both do the same amount of high-level language work in the same amount of time.

Benchmarks

- Measure the performance of systems using a set of benchmark programs.
- The same set of programs can be run on different machines and the execution times compared.
- Collection of benchmark suites is defined and maintained by the System Performance Evaluation Corporation (SPEC), an industry consortium.
- Ex: SPEC CPU2006, SPECjvm98, SPECweb99

Amdahl's Law

- Amdahl's law was first proposed by Gene Amdahl in [AMDA67] and deals with the potential speedup of a program using multiple processors compared to a single processor.
- Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.
- Let T be the total execution time of the program using a single processor.

Speedup

- Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\begin{aligned}\text{Speedup} &= \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} \\ &= \frac{T(1 - f) + Tf}{T(1 - f) + \frac{Tf}{N}} = \frac{1}{(1 - f) + \frac{f}{N}}\end{aligned}$$

The fraction of code can be executed in parallel = f

Fraction of code needs to be executed serial manner = $(1 - f)$

So, total time for execution, in single machine $T(1 - f) + T(f)$

Speedup

- Two important conclusions can be drawn:
 - When f is small, the use of parallel processors has little effect.
 - As N approaches infinity, speedup is bound by $1/(1 - f)$, so that there are diminishing returns for using more processors.

Speedup

- Consider any enhancement to a feature of a system that results in a speedup.
- The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}}$$

- Suppose that a feature of the system is used during execution a fraction of the time f , before enhancement, and that the speedup of that feature after enhancement is SU_f . Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

Speedup – Example

- Suppose that a task makes extensive use of floating-point operations, with 40% of the time is consumed by floating-point operations.
- With a new hardware design, the floating-point module is speeded up by a factor of K.
- Then the overall speedup is:

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{K}}$$

- Thus, independent of K, the maximum speedup is 1.67.

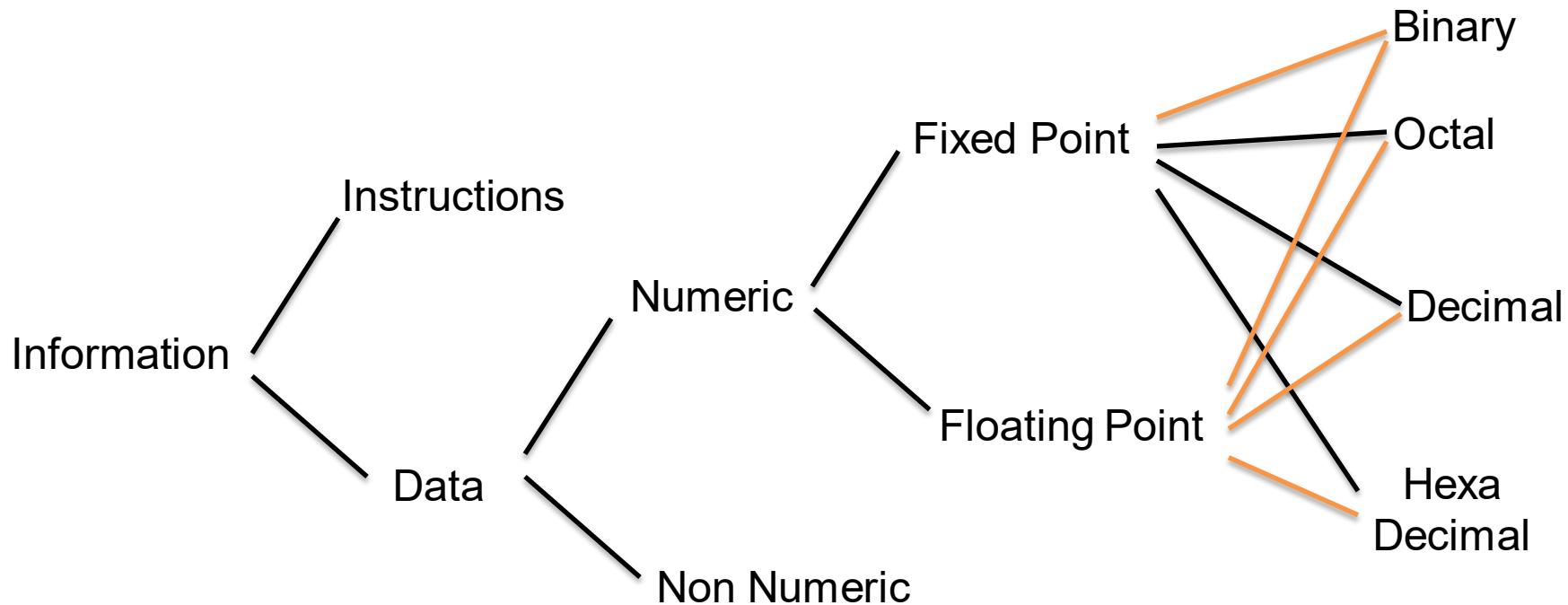
References

- **Text Book** – William Stallings, “Computer Organization and Architecture, Designing for performance”, 8th edition, Prentice Hall.
- Internet Sources.

Data Representation

**LIJO V. P.
VIT, VELLORE**

Data Representation



Integer /Fixed point representations

There are 4 commonly known (1 not common) integer representations.

All have been used at various times for various reasons.

1. unsigned
2. sign magnitude
3. one's complement
4. two's complement
5. biased (not commonly known)

Question: virtually all modern computers operate based on 2's complement representation. why?

1. hardware is faster
2. hardware is simpler (which makes it faster)

UNSIGNED

The standard binary encoding already given only positive values range:
 0 to $2^n - 1$ for n bits

Example: 4 bits, values 0 to 15; $n=4$, $2^4 - 1$ is 15; in normal binary representation, the **largest number possible** with n bits is $2^n - 1$.

Binary	decimal	hex	binary	decimal	hex
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	a
0011	3	3	1011	11	b
0100	4	4	1100	12	c
0101	5	5	1101	13	d
0110	6	6	1110	14	e
0111	7	7	1111	15	f

Sign magnitude Representation

Use 1 bit to represent the sign of the integer

sign bit - 0 -> positive, 1 -> negative.

The remaining bits are used to represent the magnitude

example: 4 bits (MSB – Sign bit, 3 bits for magnitude

0 101 is 5

1 101 is -5

To get the additive inverse of a number, just flip (not, invert, complement, negate) the sign bit.

The hardware that does arithmetic on sign magnitude integers is not fast, and it is more complex than the hardware that does arithmetic on 1's comp. and 2's comp. integers.

Sign magnitude Representation

Range: $-2^{n-1} + 1$ to $2^{n-1} - 1$
where n is the number of bits

Example: $n = 4$

Range:	$-2^3 + 1$	to	$+2^3 - 1$
	$-8 + 1$	to	$+8 - 1$
	-7	to	+7

Things to Notice:

Because of the sign bit, there are 2 representations for 0. This is a problem for hardware. . .

0 000 is +0, 1 000 is -0

Sign magnitude Multiplication

General Rules

- Multiplication involves the generation of partial products – one for each digit in the multiplier.
- Partial products are summed to produce the final product.
- Partial products are very simple to define for binary multiplication. If the digit is a ‘one’ the partial product is the multiplicand, otherwise the partial product is zero.
- The total product is the sum of the partial products. Each successive partial product is shifted one position to the left.
- The multiplication of two n-bit binary numbers results in a product of up to $2n$ bits in length.
- m bits \times n bits = $m+n$ bit product

• Series of successive shift and add operations

• 23 10111 Multiplicand

19 x 10011 Multiplier

 10111

 10111

 00000

 00000

10111 +

437 110110101 Product

1 0 0 0 two = 8₍₁₀₎ multiplicand

1 0 0 1 two = 9₍₁₀₎ multiplier

1 0 0 0	
0 0 0 0	
0 0 0 0	
1 0 0 0	

1 0 0 1 0 0 0 two = 72₍₁₀₎

Paper and pencil Method
(unsigned or Signed-Magnitude Representation excluding Sign)

1011
x 1101

1011

partial products

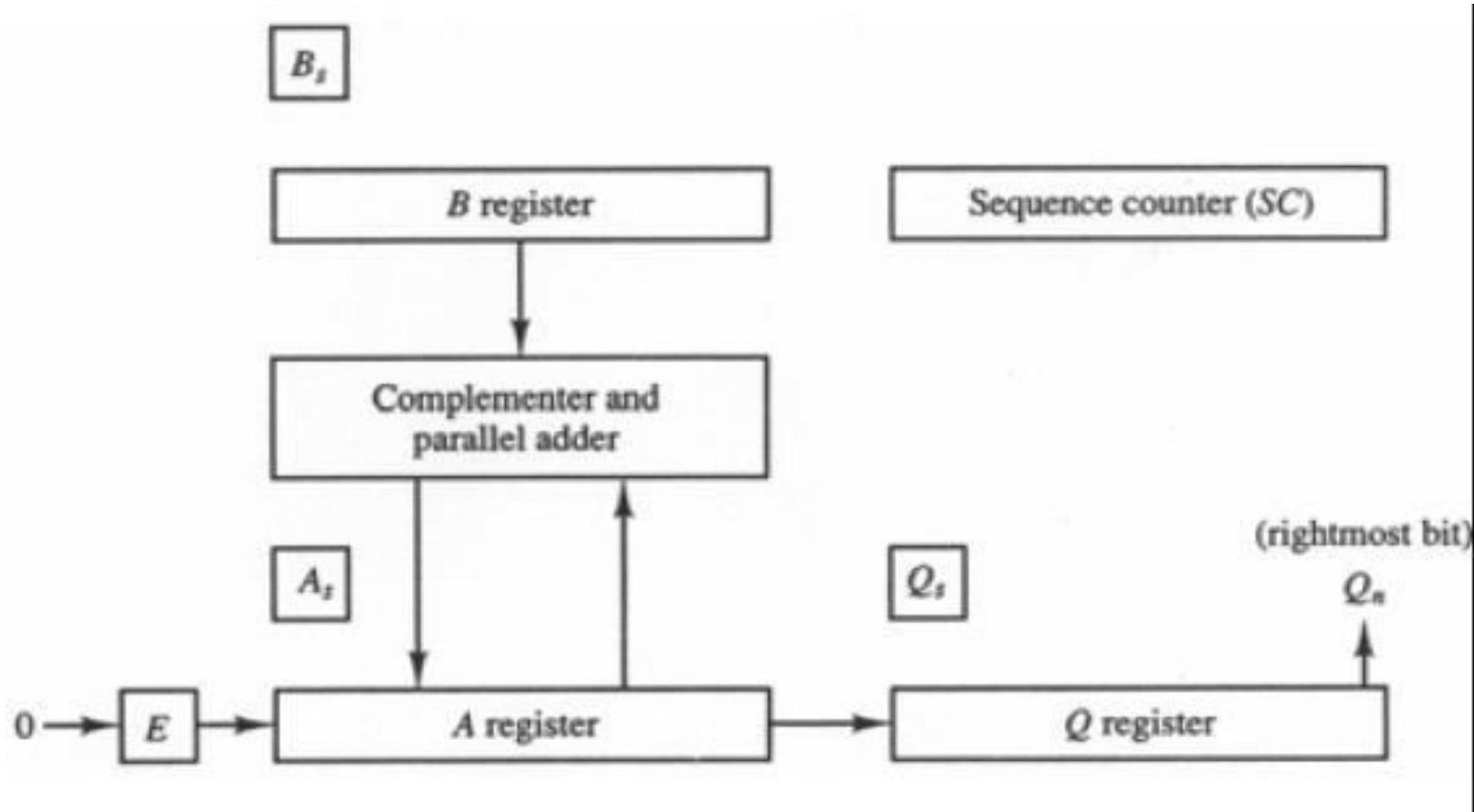
0000
1011
1011

10001111

Simplifying Multiplication

- The processor can keep a running product rather than summing at the end.
- For each ‘1’ in the multiplier we can apply an add and a shift.
- For each ‘0’ only a shift is needed.

Hardware Implementation for Signed-Magnitude Multiplication



Description

- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
 - If low-order bit of Q is 1
 - $A \leftarrow A + B$
 - Shift right EAQ
 - Product is in AQ

Flow chart for Multiplication (Signed Magnitude Representation)

14-Data Representation-04-Aug-2020Material_I_04-Aug-2020_2.1DataRepresentation

Multiply Operation

Multiplicand in B
Multiplier in Q

```

 $A_S \leftarrow Q_S + B_S$ 
 $Q_S \leftarrow Q_S + B_S$ 
 $A \leftarrow 0, E \leftarrow 0$ 
 $SC \leftarrow n - 1$ 

```

= 0 = 1

$EA \leftarrow A + B$

Shr EAQ
 $SC \leftarrow SC - 1$

0 = 0

END
(Product is in AQ)³⁶⁸

$$23 \times 19 = 437$$

$$Q = 19 (10011), B = 23 (10111)$$

	E	A	Q	SC
Add	0	00000 10111	10011	101
Shift Add	0	01011 10111	11001	100
Shift	1	00010	01100	011
Shift	0	10001	10110	010
Shift Add	0	01000 00100 10111	01011	001
Shift	0	11011	10101	000

Example: 11×13

E	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101		Add } First
0	0101	1110		Shift } Cycle
0	0010	1111		Shift } Second
0	1101	1111		Shift } Cycle
0	0110	1111		Add } Third
1	0001	1111		Shift } Cycle
0	1000	1111		Add } Fourth
				Shift } Cycle

Group Exercise:**Group 1: 3×7 , Group 2: -3×7 , Group 3: 3×-7 , Group 4: -3×-7**

Exercise

- Show the contents of registers E, A, Q and SC during the process of multiplication of two binary numbers, 11111 (multiplicand) and 10101 (multiplier). The signs are not included.

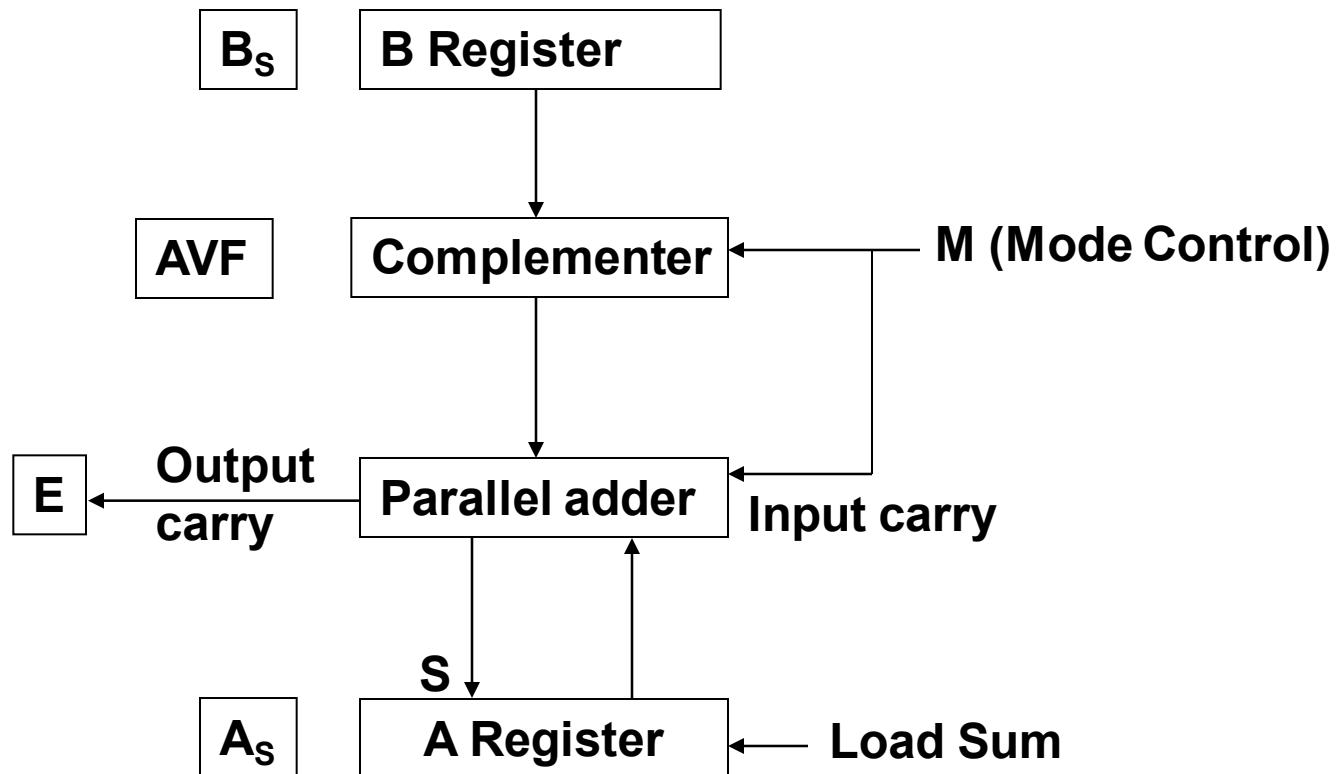
Addition (subtraction) with signed magnitude data

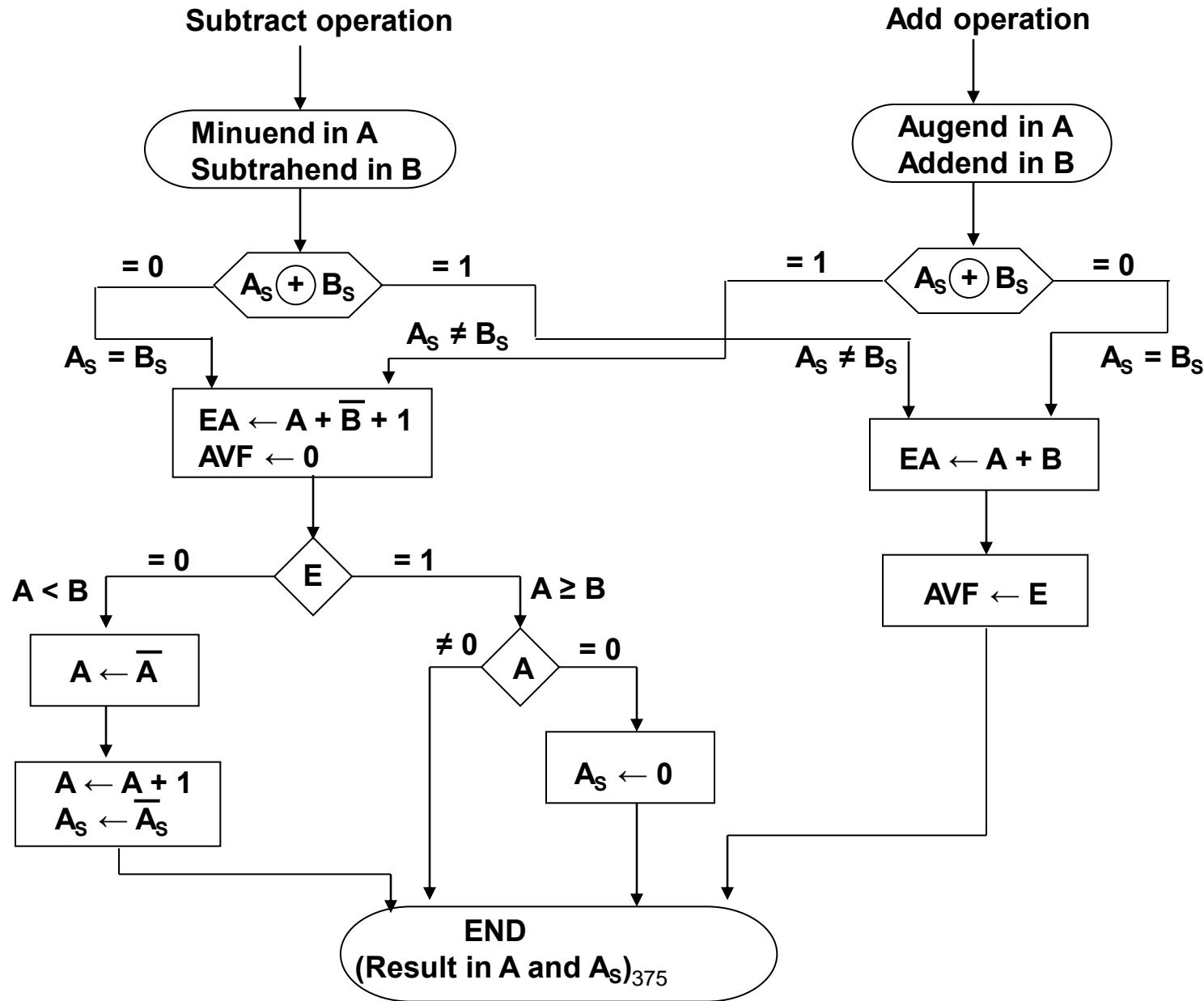
- When signs of A and B
 - Are identical (different)
 - Add magnitudes
 - Are different (identical)
 - Compare the magnitudes
 - if $A > B$, subtract B from A, put the sign of A for the result
 - If $A < B$, subtract A from B, put the complement of the sign of A for the result.

Addition and Subtraction of signed magnitude numbers

Operation	Magnitudes	Subtract Magnitudes		
		Add	When $A > B$	When $A < B$
$(+A) + (+B)$	$+ (A + B)$			
$(+A) + (-B)$		$+ (A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$- (A + B)$			
$(+A) - (+B)$		$+ (A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+ (A + B)$			
$(-A) - (+B)$	$- (A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Hardware for signed magnitude addition and subtraction





Problems

- Mark each individual path in the flow chart by a number and then indicate the overall path that the algorithm takes when the following signed magnitude numbers are computed. In each case give the value of AVF. The left most bit in the following numbers represents the sign bit
 - 0 1001101 + 0 011111
 - 1 011111 + 1 101101
 - 0 101101 – 0 011111
 - 0 101101 – 0 101101
 - 1 011111 – 0 101101

Signed magnitude addition

$$\begin{array}{r} \textcolor{blue}{1} \ 1 \ 1 \ 1 \ 1 \\ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\ + \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \\ \hline \textcolor{blue}{0} \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \end{array}$$

$$E = AVF = 1$$

ONE'S COMPLEMENT

In the past, early computers built by Seymour Cray were based on 1's comp. integers.

Historically important, and we use this representation to get 2's complement integers.

Now, nobody builds machines which are based on 1's comp. integers.

positive integers use the same representation as unsigned.

00000 is 0; 00111 is 7, etc.

Negation (finding an additive inverse) is done by taking a bit wise complement of the positive representation.

EXAMPLES: 11100

This must be a negative number.

To find out which, find the additive inverse!

00011 is +3 by sight,

so 11100 must be -3

Things to notice:

1. any negative number will have a 1 in the MSB.
2. there are 2 representations for 0,

00000 and 11111.

TWO's COMPLEMENT

A variation on 2's complement is that it does NOT have 2 representations for 0.

This makes the arithmetic calculation faster than the other representations.

Range: -2^{n-1} to $2^{n-1} - 1$

A 3 bit example:

bit pattern: 100 101 110 111 000 001 010 011

1's comp: -3 -2 -1 0 0 1 2 3

2's comp.: -4 -3 -2 -1 0 1 2 3

Two's Complement

EXAMPLE:

What decimal value does the two's complement 110011 represent?

It must be a negative number, since the most significant bit is a 1. Therefore, find the additive inverse:

110011 (2's comp. ?)

001100 (after taking the 1's complement)

+ 1

001101 (2's comp. +13)

Therefore, the given value must be -13.

Pros and cons of integer representation

- Signed magnitude representation:
 - 2 representations for 0
 - Simple
 - 255 different numbers can be represented.
 - Need to consider both sign and magnitude in arithmetic
 - Different logic for addition and subtraction
- 1's complement representation:
 - 2 representations for 0
 - Complexity in performing addition and subtraction
 - 255 different numbers can be represented.
- 2's complement representation:
 - Only one representation for 0
 - 256 different numbers can be represented.
 - Arithmetic works easily
 - Negating is fairly easy

Floating Point Number representation

- * The location of the fractional point is not fixed to a certain location
- * The range of the representable numbers is wide

$$F = EM$$

m_n	$e_k e_{k-1} \dots e_0$	$m_{n-1} m_{n-2} \dots m_0 \cdot m_{-1} \dots m_{-m}$
sign	exponent	mantissa

- Mantissa

Signed fixed point number, either an integer or a fractional number

- Exponent

Designates the position of the radix point

Floating Point Number Representation

Example

sign	
0	.1234567
mantissa	

$$\Rightarrow +.1234567 \times 10^{+04}$$

sign	
0	04
exponent	

Note:

In Floating Point Number representation, only Mantissa(M) and Exponent(E) are explicitly represented. The Radix(R) and the position of the Radix Point are implied.

Example

A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa)

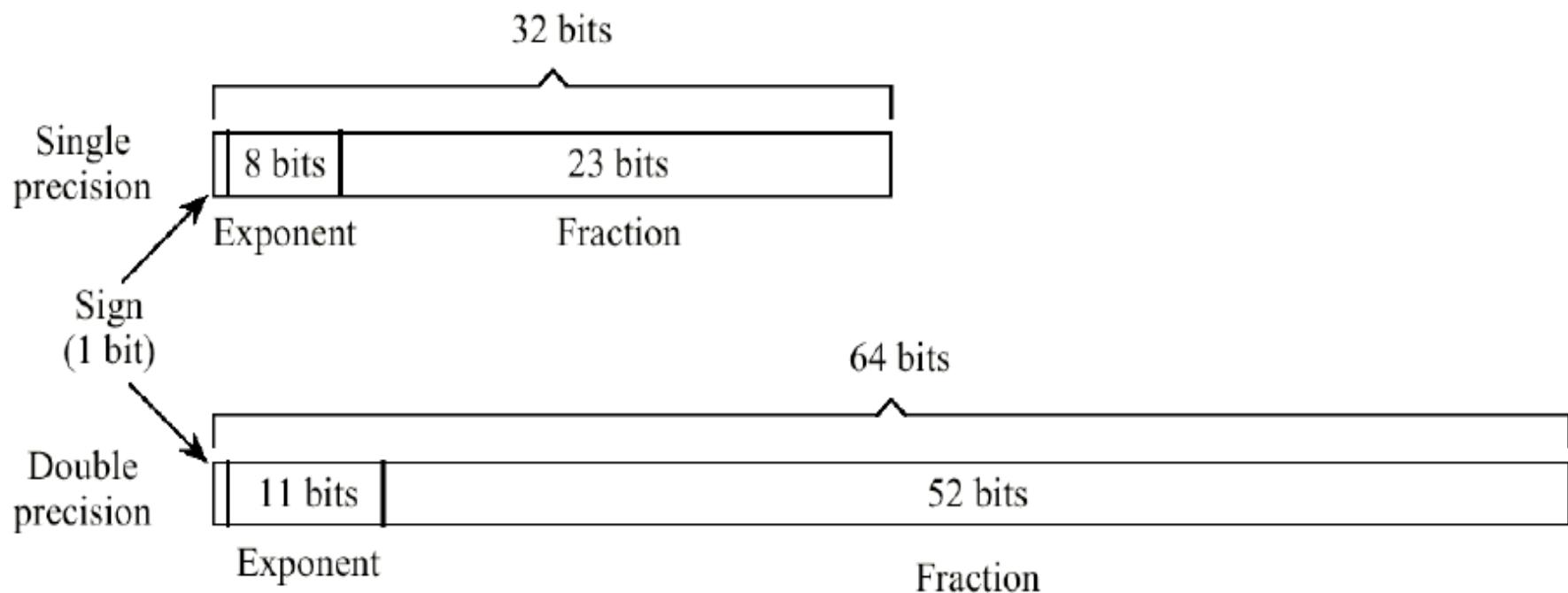
or

	0	0 00100	100111000
	Sign	Exponent	Mantissa
	0	0 00101	010011100 384

Normal Form

- There are many different floating point number representations of the same number
 - Need for a unified representation in a given computer
- *the most significant position of the mantissa contains a non-zero digit*

IEEE-754 Floating Point Formats



IEEE-754 Examples

	Value	Bit Pattern		
		Sign	Exponent	Fraction
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	-1.01011×2^{-126}	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	$+0$	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	-0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	$+NaN$	0	1111 1111	011 0111 0000 0000 0000 0000

IEEE-754 Conversion Example

Represent -12.62510 in single precision IEEE-754 format.

- Step #1: Convert to target base. $-12.62510 = -1100.101_2$
- Step #2: Normalize. $-1100.101_2 = -1.100101_2 \times 2^3$
- Step #3: Fill in bit fields. Sign is negative, so sign bit is 1.
Exponent is

in excess 127 (not excess 128!), so exponent is represented as
the

unsigned integer $3 + 127 = 130$. Leading 1 of significant is hidden,
so

final bit pattern is:

1 1000 0010 . 1001 0100 0000 0000 0000 000

Character Representation ASCII

ASCII (American Standard Code for Information Interchange) Code

		MSB (3 bits)							
		0	1	2	3	4	5	6	7
LSB (4 bits)	0	NUL	DLE	SP	0	@	P	'	P
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	m	n	~
	F	SI	US	389 /	?	O	n	o	DEL

Control Character Representation (ASCII)

NUL	Null	DC1	Device Control 1
SOH	Start of Heading (CC)	DC2	Device Control 2
STX	Start of Text (CC)	DC3	Device Control 3
ETX	End of Text (CC)	DC4	Device Control 4
EOT	End of Transmission (CC)	NAK	Negative Acknowledge (CC)
ENQ	Enquiry (CC)	SYN	Synchronous Idle (CC)
ACK	Acknowledge (CC)	ETB	End of Transmission Block (CC)
BEL	Bell	CAN	Cancel
BS	Backspace (FE)	EM	End of Medium
HT	Horizontal Tab. (FE)	SUB	Substitute
LF	Line Feed (FE)	ESC	Escape
VT	Vertical Tab. (FE)	FS	File Separator (IS)
FF	Form Feed (FE)	GS	Group Separator (IS)
CR	Carriage Return (FE)	RS	Record Separator (IS)
SO	Shift Out	US	Unit Separator (IS)
SI	Shift In	DEL	Delete
DLE	Data Link Escape (CC)		

(CC) Communication Control

(FE) Format Effector

(IS) Information Separator

The EBCDIC character code, shown with hexadecimal indices

00	NUL	20	DS	40	SP	60	-	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	'	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(6D	-	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F		6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	
1B	CU1	3B	CU3	5B	\$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	.	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	:	7E	394	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	-	7F	"	9F		BF		DF		FF	

The EBCDIC control character representation

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

References

Text Book

- M. M. Mano, Computer System Architecture, Prentice-Hall, 2004
- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

Booth Multiplication Algorithm

Booth Algorithm

- An efficient way to multiply two signed binary numbers expressed in 2's complement notation :
- Reduces the number of operations by relying on blocks of consecutive 1's
- Example:
- $Y \times 0111110 = Y \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1)$.
- $Y \times 0111110 = Y \times (01000000 - 00000010) = Y \times (2^6 - 2^1)$.
One addition and one subtraction

- Search for a run of ‘1’ bits in the multiplier
 - E.g. ‘0110’ has a run of 2 ‘1’ bits in the middle
 - Multiplying by ‘0110’ (6 in decimal) is equivalent to multiplying by 8 and subtracting two, since $6 \times m = (8 - 2) \times m = 8m - 2m$, $(2^{k+1} - 2^n)$ where $k = 2$ and $n = 1$

Description and Hardware for Booth Multiplication

- QR multiplier
- Q_n least significant bit of QR
- Q_{n+1} previous least significant bit of QR
- BR multiplicand
- $AC = 0$
- SC number of bits in multiplier

Algorithm

Do SC times

$$Q_n Q_{n+1} = 10$$

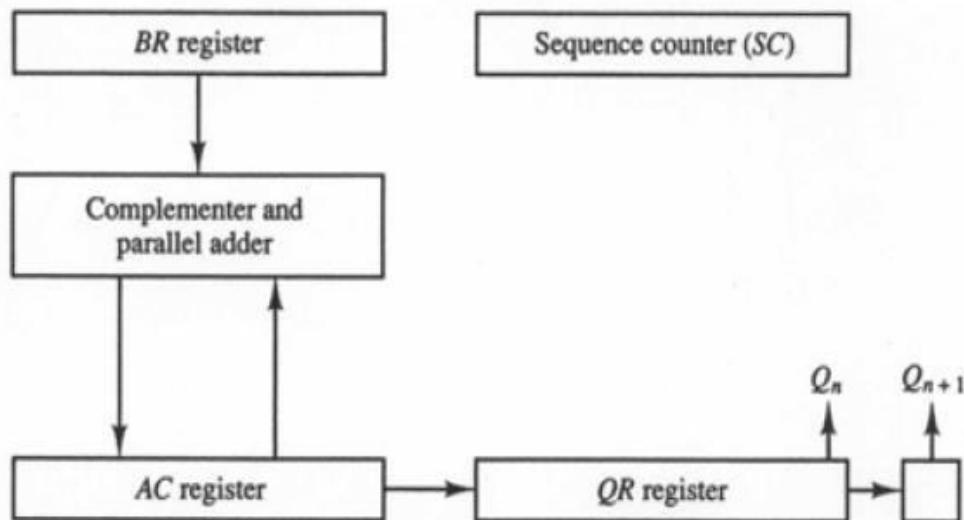
$$AC \leftarrow AC + \overline{BR} + 1$$

$$Q_n Q_{n+1} = 01$$

$$AC \leftarrow AC + BR$$

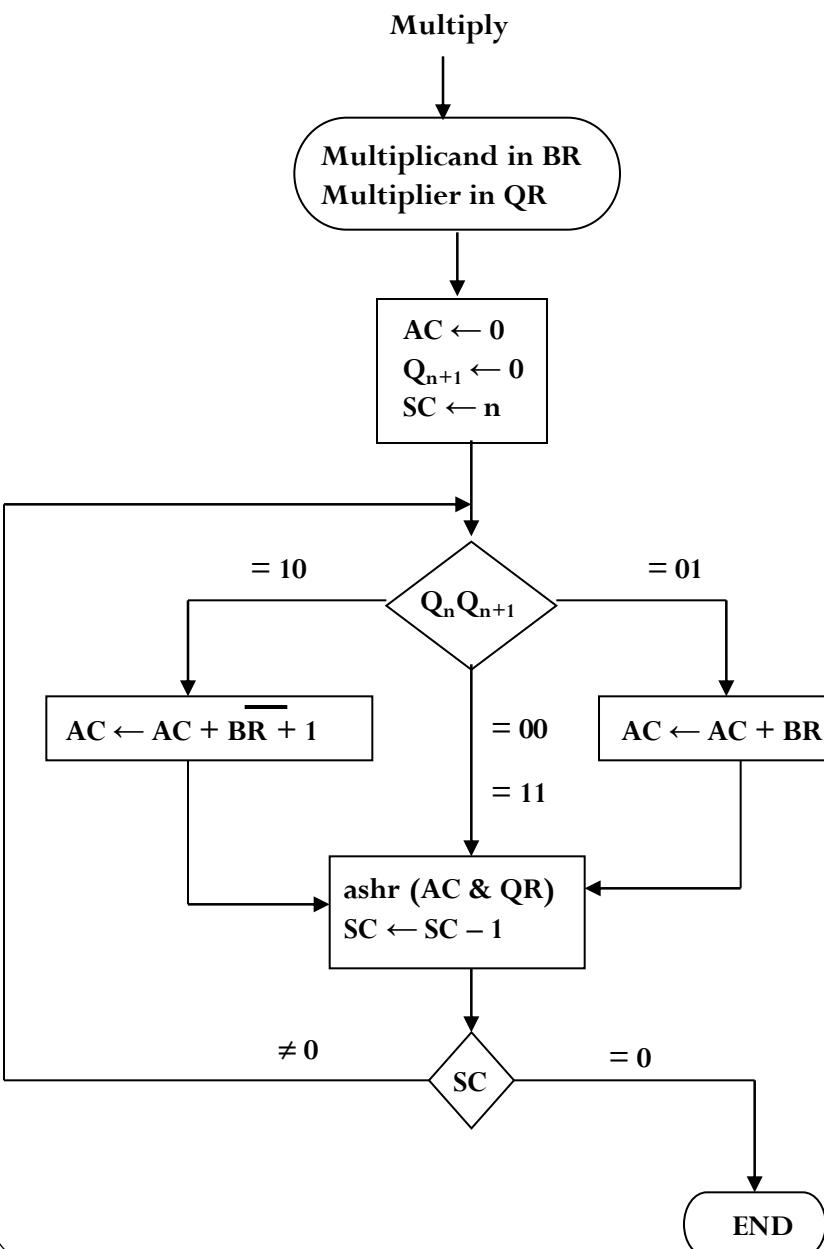
Arithmetic shift right AC & QR

$$SC \leftarrow SC - 1$$



- For our example, and multiply **(-9) x (-13)**
 - The numerically larger **operand (13) would require 4 bits** to represent in binary (1101). So we must use **AT LEAST 5 bits** to represent the operands, to allow for the sign bit.

Flowchart for Booth Multiplication



Example: $-9 \times -13 = 117$
 $BR = 10111, \overline{BR} + 1 = 01001$

Comment	AC	QR	Q _{n+1}	SC
	00000	10011	0	5
Subtract BR	01001			
	01001			
Ashr	00100	11001	1	4
Ashr	00010	01100	1	3
Add BR	10111			
	11001			
Ashr	11100	10110	0	2
Ashr	11110	01011	0	1
Subtract BR	01001			
	00111			
Ashr	00011	10101	1	0

Multiply 7×3 using above signed 2's compliment binary multiplication.

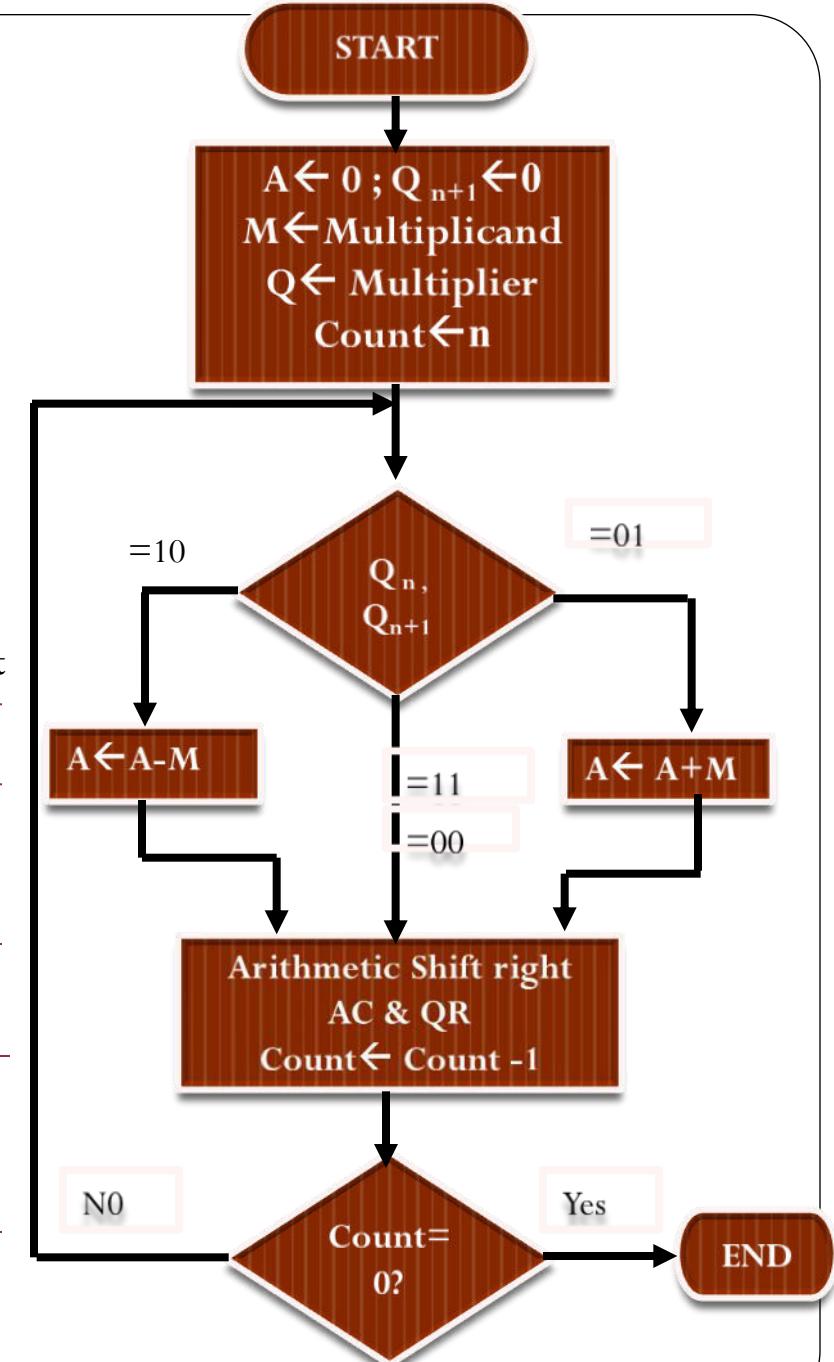
Multiplicand = 7 → Binary equivalent is 0111 → M

Multiplier = 3 → Binary equivalent is 0011 → Q

-7 → Binary equivalent is 1001 → -M

$$\begin{array}{r}
 \begin{array}{r} A \ 0 \ 0 \ 0 \ 0 \\ + M \ 1 \ 0 \ 0 \ 1 \\ \hline A \ 1 \ 0 \ 0 \ 1 \end{array}
 \quad
 \begin{array}{r} A \ 1 \ 1 \ 1 \ 0 \\ + M \ 0 \ 1 \ 1 \ 1 \\ \hline A \ 0 \ 1 \ 0 \ 1 \end{array}
 \end{array}$$

Step	A	Q	Q_{n+1}	Action	Count
1	0 0 0 0	0 0 1 1	0	Initial	4
2	1 0 0 1	0 0 1 1	0	$A \leftarrow A - M$	
2	1 1 0 0	1 0 0 1	1	Shift	3
3	1 1 1 0	0 1 0 0	1	Shift	2
4	0 1 0 1	0 1 0 0	1	$A \leftarrow A + M$	
4	0 0 1 0	1 0 1 0	0	Shift	1
5	0 0 0 1	0 1 0 1	0	Shift	0



Examples

- Multiply the following using Booth's algorithm

$$7 \times -3$$

$$-7 \times 3$$

$$-7 \times -3$$

$$11 \times 13$$

$$-11 \times 13$$

$$11 \times -13$$

$$-11 \times -13$$

-11 x 13

$$\begin{array}{r} -11 = 10101 \\ +11 = 01011 \end{array}$$

13 =01101

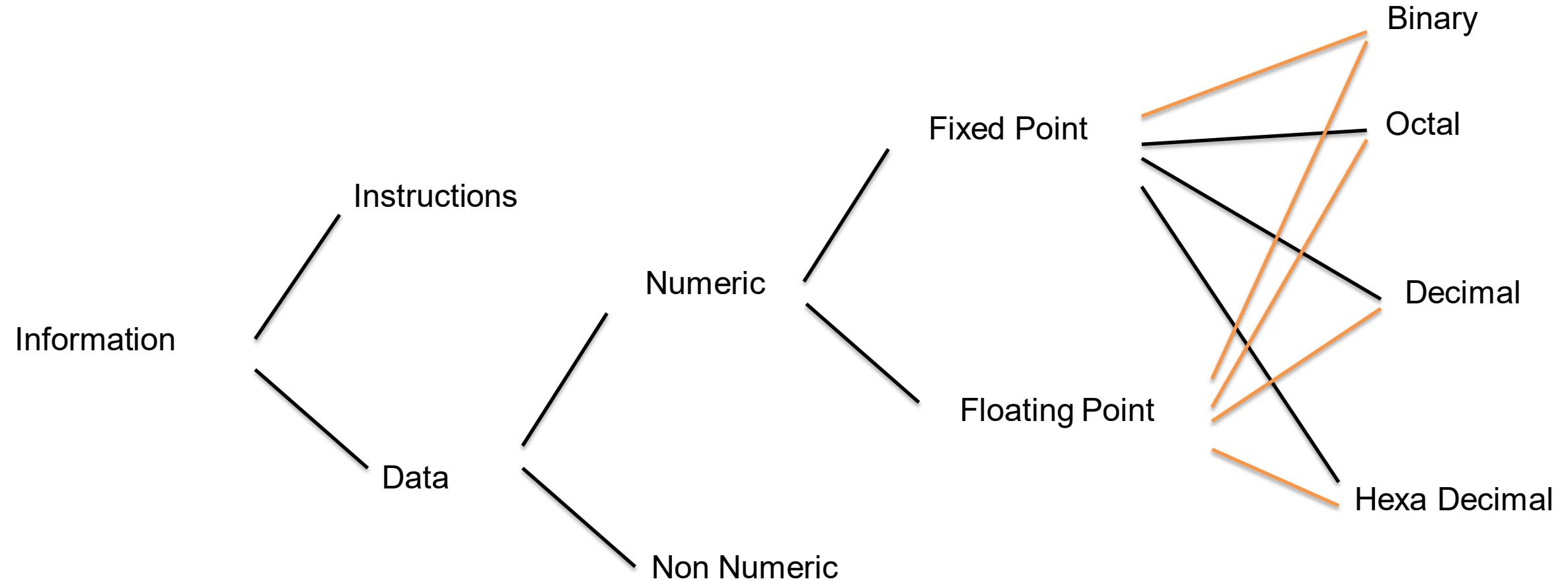
OP	AC	Q	Q _{n+1}	SC
	00000	01101	0	5
• SUB	01011	01100	0	5
• SHR	00101	10110	1	4
• ADD	11010	10110	1	4
• SHR	11101	01011	0	3
• SUB	01000	01011	0	3
• SHR	00100	00101	1	2
• SHR	00010	00010	1	1
• ADD	10111	00010	1	1
• SHR	11011	10001	0	0

Reference

- Morris Mano, “Computer System Architecture”, Pearson Education, 3rd edition (Chapter 10)

Data Representation and Computer Arithmetic

Data Representation



Integer /Fixed point representations

- There are 4 commonly known (1 uncommon) integer representations.
- All have been used at various times for various reasons.
 - unsigned
 - sign magnitude
 - one's complement
 - two's complement
 - biased (not commonly known)
- Question: virtually all modern computers operate based on 2's complement representation. why?
 - hardware is faster
 - hardware is simpler (which makes it faster)

Unsigned Representation

- In general if an n-bit sequence of binary digits $a_{n-1}a_{n-2}\dots a_1a_0$ is it $A = \sum_{i=0}^{n-1} 2^i a_i$ as an uns
- | Binary | decimal | hex | binary | decimal | hex |
|--------|---------|-----|--------|---------|-----|
| 0000 | 0 | 0 | 1000 | 8 | 8 |
| 0001 | 1 | 1 | 1001 | 9 | 9 |
| 0010 | 2 | 2 | 1010 | 10 | a |
| 0011 | 3 | 3 | 1011 | 11 | b |
| 0100 | 4 | 4 | 1100 | 12 | c |
| 0101 | 5 | 5 | 1101 | 13 | d |
| 0110 | 6 | 6 | 1110 | 14 | e |
| 0111 | 7 | 7 | 1111 | 15 | f |
- Range: 0 to $2^n - 1$ for |
 - Ex: 0 to 15 for 4 bits
($n=4$, $2^4 - 1$ is 15)

Sign magnitude Representation

- The simplest form of representation that employs a sign bit is the sign-magnitude representation.
- In an n-bit word, the rightmost bits hold the magnitude of the integer.
- The most significant (leftmost) bit in the word as a sign bit.
- Sign bit = 0 => positive
- Sign bit = 1 => negative.
- Ex: + 18 = 00010010; - 18 = 10010010
- Range: $-2^{n-1} + 1$ to $2^{n-1} - 1$
- To get the additive inverse of a number, just flip (not, invert, complement, negate) the sign bit.

Sign magnitude representation

- The general case car

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

- Drawbacks

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation.
- There are two representations of 0

1's Complement Representation

- Historically important, and we use this representation to get 2's complement integers
- In the past, early computers built by Seymour Cray (while at CDC) were based on 1's comp. integers.
- Positive integers use the same representation as unsigned; 00000 is 0; 00111 is 7;
- Negation (finding an additive inverse) is done by taking a bit wise complement of the positive representation.
- EXAMPLES: 11100 (This must be a negative number)
- 00011 is +3, so 11100 must be -3
- Things to notice:
 - Any negative number will have a 1 in the MSB.
 - There are 2 representations for 0; 00000 and 11111.

2's Complement

- The most significant bit represents a sign of the number
- **Range:** -2^{n-1} through $2^{n-1} - 1$
- **Number of Representations of Zero:** One
- **Negation:** Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
- **Expansion of Bit Length:** Add additional bit positions to the left and fill in with the value of the original sign bit.
- **Overflow Rule:** If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.

Pros and cons of integer representation

- Signed magnitude representation:
 - 2 representations for 0
 - Simple
 - 255 different numbers can be represented.
 - Need to consider both sign and magnitude in arithmetic
 - Different logic for addition and subtraction
- 1's complement representation:
 - 2 representations for 0
 - Complexity in performing addition and subtraction
 - 255 different numbers can be represented.
- 2's complement representation:
 - Only one representation for 0

2's Complement Subtraction

- Represent negative numbers in 2's complement form
- Subtract the given two numbers
 - Take 2's complement of subtrahend (second number) and add to the minuend (first number)
- If carry
 - Discard it
 - Answer is positive
- Else
 - Answer is negative
 - So, answer is in 2's complement form

Overflow

- Overflow in unsigned addition – result occupies more number of bits.
- Overflow in signed-magnitude / 2's complement addition
 - result occupies more number of bits
 - addition of two positive number resulting negative number
 - sum of two negative numbers resulting in positive number.
- There will no overflow during unsigned subtraction / signed – magnitude subtraction.
- Overflow bit in 2's complement subtraction is ignored

Overflow / Underflow Problem

- Overflow – when the addition of two binary numbers yields a result that is greater than the maximum possible value
- Underflow – when the addition/subtraction of two binary numbers yields a result that is less than the minimum possible value
- Assume 4-bit restriction and 2's complement
 - Maximum possible value: $2^{4-1} - 1 = 7$
 - Minimum possible value: $-(2^{4-1}) = -8$

$$6_{10} + 3_{10} = 9_{10}$$

$$\begin{array}{r}
 0110_2 \quad 6_{10} \\
 +0011_2 \quad +3_{10} \\
 \hline
 1001_2 \quad -7_{10} \quad \leftarrow \text{not good!}
 \end{array}$$

$$-5_{10} + -5_{10} = -10_{10}$$

$$\begin{array}{r}
 1011_2 \quad -5_{10} \\
 +1011_2 \quad + -5_{10} \\
 \hline
 0110_2 \quad 6_{10} \quad \leftarrow \text{not good!}
 \end{array}$$

Overflow of Signed Numbers

$$\begin{array}{r}
 1101 \text{ } (-3) \\
 +1011 \text{ } (-5) \\
 \hline
 1000 \text{ } (-8)
 \end{array}$$

carry generated,
but no overflow

$$\begin{array}{r}
 1 \\
 1101 \text{ } (-3) \\
 +1010 \text{ } (-6) \\
 \hline
 1 \ 0111 \text{ } (+7)
 \end{array}$$

carry and
overflow

$$\begin{array}{r}
 1101 \text{ } (-3) \\
 +0010 \text{ } (+2) \\
 \hline
 0 \ 1111 \text{ } (-1)
 \end{array}$$

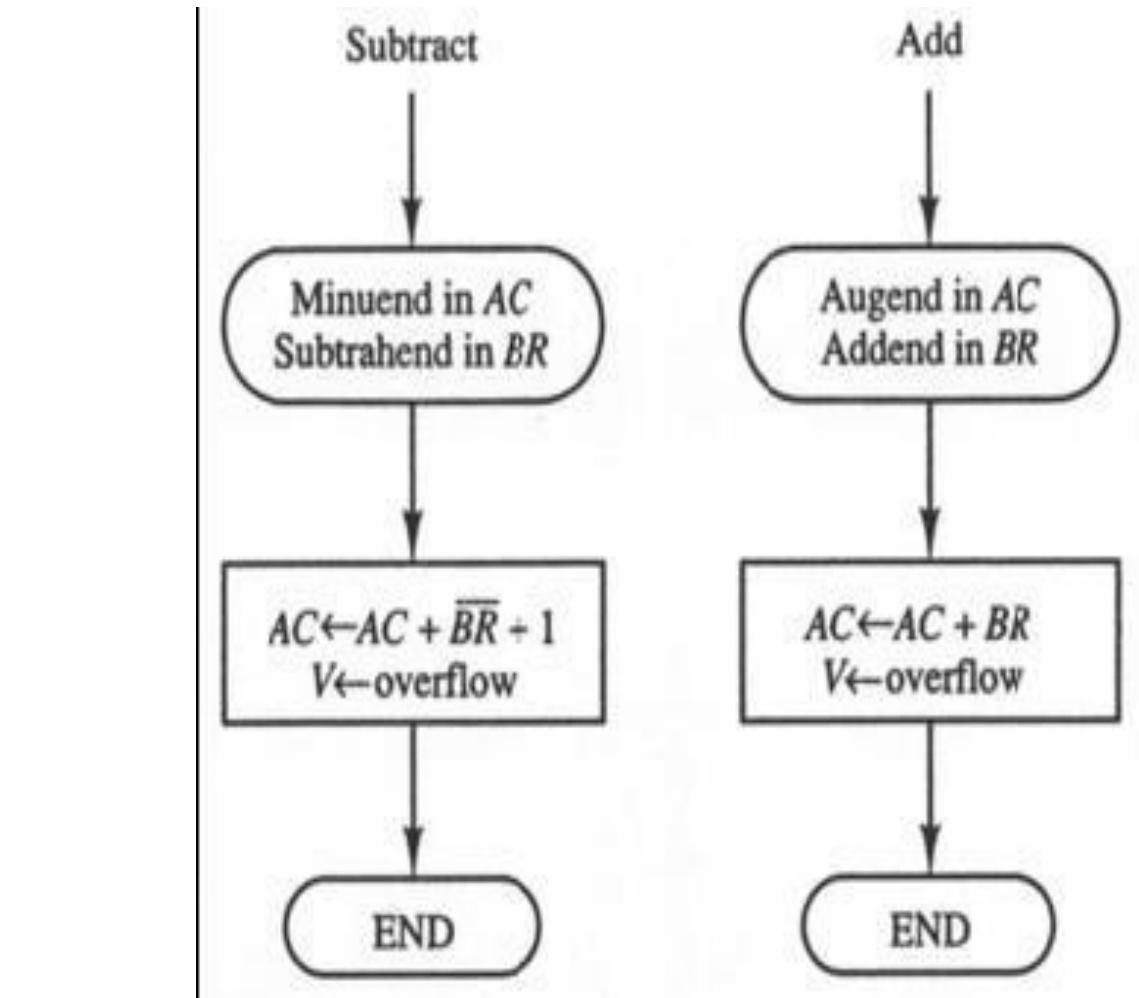
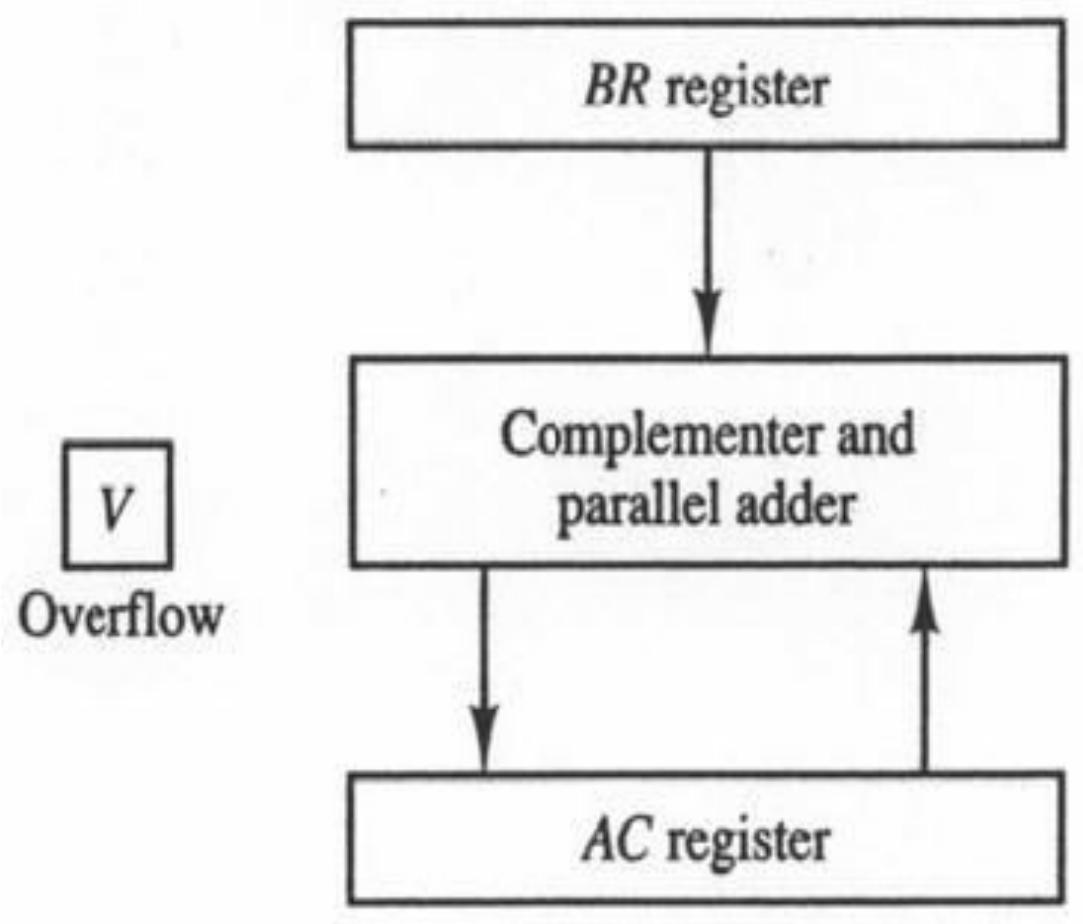
no carry and
no overflow

$$\begin{array}{r}
 0111 \text{ } (+7) \\
 +0011 \text{ } (+3) \\
 \hline
 0 \ 1010 \text{ } (-6)
 \end{array}$$

no carry and
overflow

$\text{Cin} \oplus \text{Cout} = 1 \Rightarrow \text{Overflow}$

2's Complement Addition and Subtraction Hardware



Problem

- Perform the arithmetic operations below with binary numbers and with negative numbers in signed 2's complement representation. Use seven bits to accommodate each number together with its sign. In each case, determine if there is an overflow by checking the carries into and out of the sign bit position

$$(+35) + (+40)$$

$$(-35) + (-40)$$

$$(-35) - (+40)$$

Multiplication - Using paper-pencil

- Using paper – pencil method

$ \begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ \hline 1011 \\ \hline 10001111 \end{array} $	Multiplicand (11) Multiplier (13) Partial products Product (143)
--	---

- Inferences

- Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the previous partial product.

Comparison of Multiplication of Unsigned and Twos Complement Integers

- We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers.
- Unfortunately, this simple scheme will not work for multiplication.
- Ex: Multiply 11 (1011) by 13 (1101) to get 143 (10001111).
- If we interpret these as twos complement numbers, we have -5 (1011) times -3 (1101) equals -113 (10001111).
- This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative.
- In fact, it will not work if either the multiplicand or the multiplier is negative

Comparison of Multiplication of Unsigned and Twos Complement Integers

$$\begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ \hline 00011011 \quad (27) \end{array}$$

Unsigned integers

$$\begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ \hline 11101011 \quad (-21) \end{array}$$

Twos complement integers

Booth's Multiplication Algorithm

$$M \times (00011110)$$

$$= M \times (2^4 + 2^3 + 2^2 + 2^1)$$

$$= M \times (16 + 8 + 4 + 2)$$

$$= M \times 30$$

The number of operations can be reduced to two if we observe that:

$$2^n + 2^{n-1} + \dots + 2^{n-k} = 2^{n+1} - 2^{n-k}$$

$$M \times (00011110) = M \times (2^5 - 2^1)$$

$$= M \times (32 - 2)$$

$$= M \times 30$$

$$\begin{aligned} 14 = 01110 &= 2^4 - 2^1 \\ &= 16 - 2 \end{aligned}$$

$$\begin{aligned} 25 = 011001 &= -2^0 + 2^1 - 2^3 + 2^5 = -1 + 2 - 8 + 32 \\ &= 1 - 8 + 32 = -7 + 32 \end{aligned}$$

Booth's Multiplication Advantage

Serial addition

00010100	20
$\times \underline{00011110}$	30
00000000	
00010100	
00010100	
00010100	
00010100	
00000000	
00000000	
<hr/>	
0000010011000	600

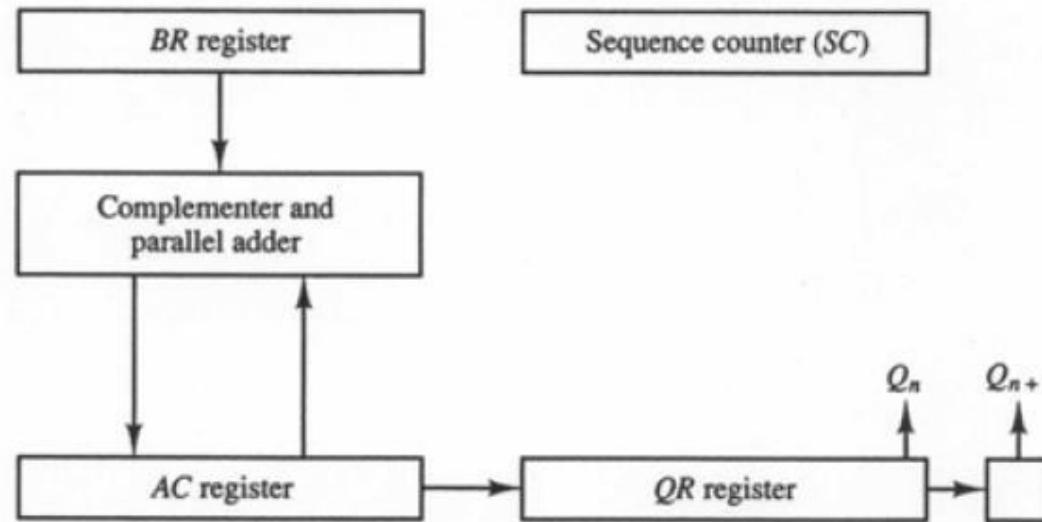
Four partial product additions

Booth algorithm

00010100	20
$\times \underline{00011110}$	30
11111111101100	
00000010100	
<hr/>	
0000010011000	600

Two partial product additions

Hardware for Booth's Multiplication



Description and Algorithm for Booth Multiplication

Description

Q - multiplier

Q_0 - least significant bit of QR

Q_{-1} - previous least significant bit of QR

M - multiplicand

A - 0

$Count$ - number of bits in multiplier

Algorithm

Do Count times

$$Q_0 Q_{-1} = 10 \quad -$$

$$A \leftarrow A + M + 1$$

$$Q_0 Q_{-1} = 01$$

$$A \leftarrow A + M$$

Arithmetic shift right A, Q, Q_{-1}

$$Count \leftarrow Count - 1$$

Booth Multiplier Recoding Table

Multiplier		Version of Multiplicand selected by bit i
Bit i	Bit i - 1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Exercise

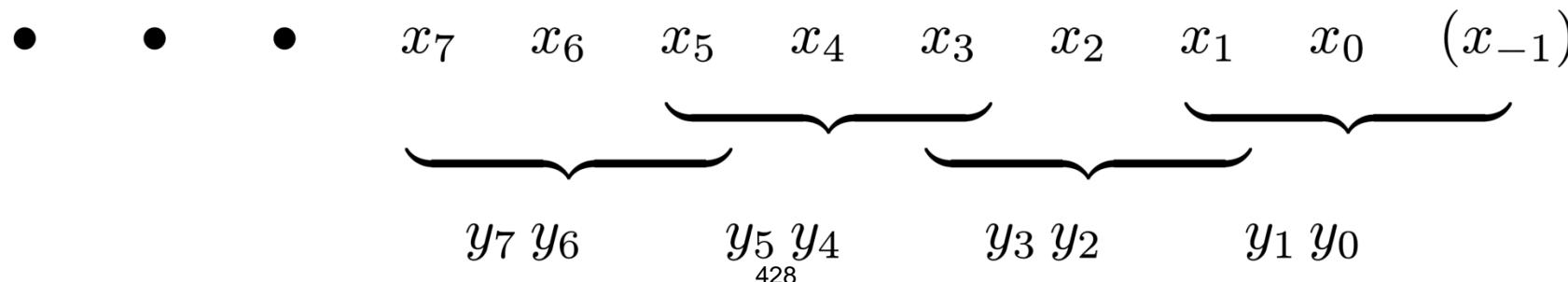
- Show the step by step multiplication process using Booth algorithm when the following binary numbers are multiplied. Assume 5-bit registers that hold signed numbers. The multiplicand in both cases is +15.

$$(+15) \times (+13)$$

$$(+15) \times (-13)$$

Modified Booth's Algorithm

- Guarantees that the maximum number of summands that must be added is $n/2$ for n -bit operands.
 - Bit pair recoding technique
 - Observe the following:
 - The pair $(+1, -1)$ is equivalent to the pair $(0, +1)$
 - $(+1, 0)$ is equivalent to $(0, +2)$
 - $(-1, +1)$ is equivalent to $(0, -1)$
 - Examines 3 bits at a time.



Modified Booth's Algorithm

$$\begin{array}{r} 101101 \\ \times -1 \ 1 \\ \hline \end{array}$$

$$\begin{array}{r} 1101101 \\ 010011 \\ \hline \end{array}$$

$$\begin{array}{r} 10010011 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \times \quad -1 \\ \hline \end{array}$$

$$\begin{array}{r} 010011 \\ \hline \end{array}$$

$$\begin{array}{r} 0010011 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \times 1 \ -1 \\ \hline \end{array}$$

$$\begin{array}{r} 0010011 \\ 101101 \\ \hline \end{array}$$

$$\begin{array}{r} 1101101 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \times \quad 1 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \hline \end{array}$$

$$\begin{array}{r} 1101101 \\ \hline \end{array}$$

Modified Booth's Algorithm

$$\begin{array}{r} 101101 \\ \times 10 \\ \hline 0000000 \\ 101101 \\ \hline 1011010 \end{array}$$

$$\begin{array}{r} 101101 \\ \times +2 \\ \hline 1011010 \\ 1011010 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \times -10 \\ \hline 0000000 \\ 010011 \\ \hline 0100110 \end{array}$$

$$\begin{array}{r} 101101 \\ \times -2 \\ \hline 0100110 \\ 0100110 \\ \hline \end{array}$$

Table of Multiplicand and Selection decisions

Multiplier Bit-Pair		Multiplier bit on the right	Booths Represenation	Multiplicand selected at position i
i + 1	i	i - 1		
0	0	0	0	0 $\times M$
0	0	1	0	+1 $\times M$
0	1	0	1	-1 $\times M$
0	1	1	1	+2 $\times M$
1	0	0	-1	-2 $\times M$
1	0	1	-1	-1 $\times M$
1	1	0	0	-1 $\times M$
1	1	1	0	0 $\times M$

Select Line (encoding)	Partial Products (operation)
000	add 0
001	add multiplicand
010	add multiplicand
011	add 2^* multiplicand
100	subtract 2^* multiplicand
101	subtract multiplicand
110	subtract multiplicand
111	subtract 0

Multiplication requiring only $n/2$ summands

$$\begin{array}{r} 0 \ 1 \ 1 \ 0 \ 1 \ (+13) \\ \times 1 \ 1 \ 0 \ 1 \ 0 \ (-6) \\ \hline \end{array}$$

$$\begin{array}{r}
 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & -1 & +1 & -1 & 0 \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
 \end{array}$$

$$\begin{array}{r}
 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & -1 & -2 \\
 \hline
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0
 \end{array}$$

Modified Booth's Multiplication - Example

Example: $-9 \times -13 = 117$

$$M = 110111, \overline{M} + 1 = 001001$$

Comment	A	Q	Q ₋₁	SC
	000000	1100	11 0	3
Subtract M	001001			
	001001			
Ashr	000100	111001	1	
Ashr	000010	011100	1	2
Add M	<u>110111</u>			
	111001			
Ashr	111100	101110	0	
Ashr	111110	0101	11 0	1
Subtract M	<u>001001</u>			
	000111			
Ashr	000011	101011	1	
Ashr	000001	110101	1	0

Modified Booth's Multiplication - Example

Example: $13 \times 9 = 117$

$M = 001101$, $\overline{M} + 1 = 110011$

Comment	A	Q	Q ₋₁	SC
	000000	001001	0	3
Add M	001101			
	001101			
Ashr	000110	100100	1	
Ashr	000011	010010	0	2
Add shifted 2's	100110			
	101001			
Ashr	110100	101001	0	
Ashr	111010	010100	1	1
Add M	001101			
	000111			
Ashr	000011	101010	0	
Ashr	000001	110101	0	0

Restoring Division Algorithm and Flowchart

- Input:
 - M – positive divisor (n-bit)
 - Q – positive dividend (n-bit)
- Output:
 - Q – Quotient
 - A – Remainder
- Begin
 - $A \leftarrow 0$
 - Do n times
 - Shift A and Q left one binary position
 - $A \leftarrow A - M$
 - If sign of A is 1
 - $q_0 \leftarrow 0$ and $A \leftarrow A + M$ (Restore A)
 - Else
 - $q_0 \leftarrow 1$
 - End

References

Text Book

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- Carl Hamacher, Zvonko Vranesic, Sofwat Zaky, “Computer Organization”, 5th edition, Mc Graw Hill.
- <http://courses.cs.tamu.edu/rabi/cpsc321/lectures/lec06.ppt>

Restoring and Non-restoring Division Algorithms

Unsigned division

- Division is a more tedious process than multiplication.
- For the unsigned case, there are two standard approaches:
 - 1.) Restoring division. 2.) Non restoring division.

$$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ 26 \\ \hline 14 \\ 13 \\ \hline 1 \end{array}$$

Try dividing 13 into 2.

Try dividing 13 into 26.

$$\begin{array}{r} 10101 \\ 1101 \overline{) 10001001} \\ 01101 \\ \hline 10000 \\ 1101 \\ \hline 1110 \\ 1101 \\ \hline 1 \end{array}$$

Try dividing 1101 into 1, 10, 100, 1000 and 10001.

Restoring division

How do we know when the divisor has gone into part of the dividend correctly?

$$\begin{array}{r} 10101 \\ 1101 \overline{)10001001} \\ 01101 \\ \hline 10000 \\ 1101 \\ \hline 1110 \\ 1101 \\ \hline 1 \end{array}$$

Subtract 1101 from 1, result is negative
Subtract 1101 from 10, result is negative.
Subtract 1101 from 100, result is negative
Subtract 1101 from 1000, result is negative.
Subtract 1101 from 10001, result is positive.

Restoring division

Strategy for unsigned division:

Shift the dividend one bit at a time starting from MSB into a register.
Subtract the divisor from this register.

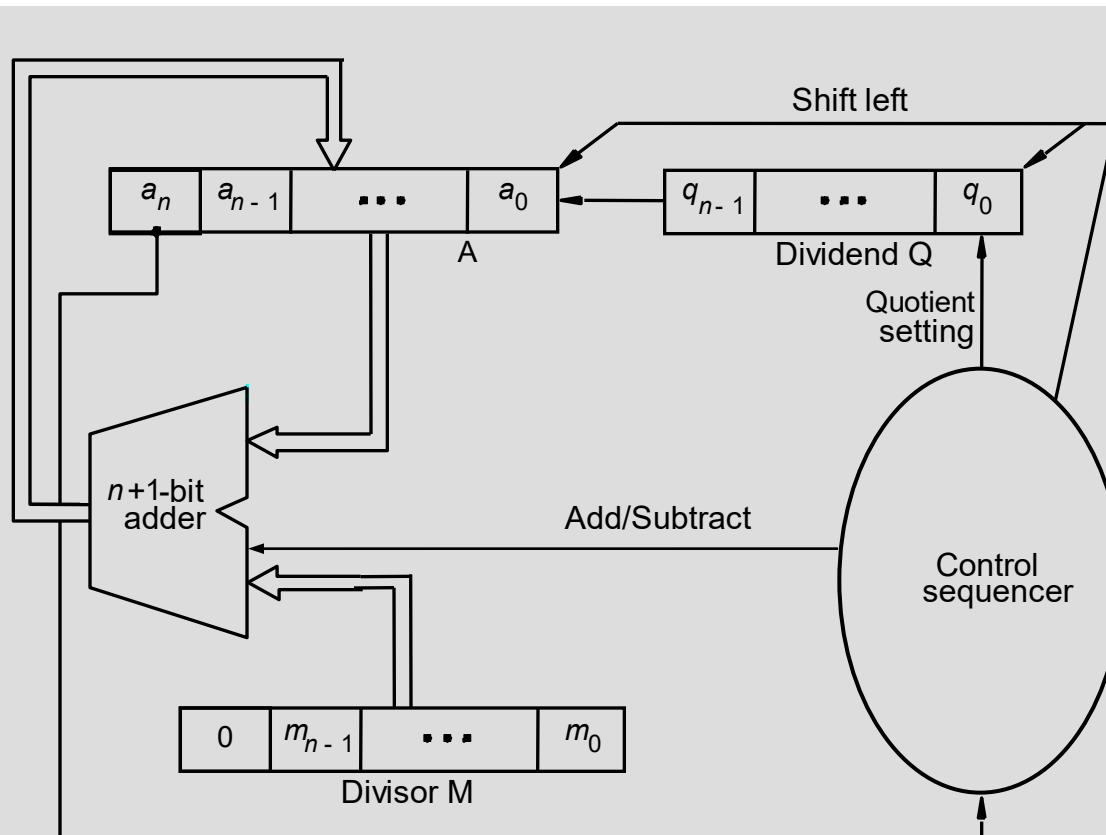
If the result is negative ("didn't go"):

- Add the divisor back into the register.
- Record 0 into the result register.

If the result is positive:

- Do not restore the intermediate result.
- Set a 1 into the result register.

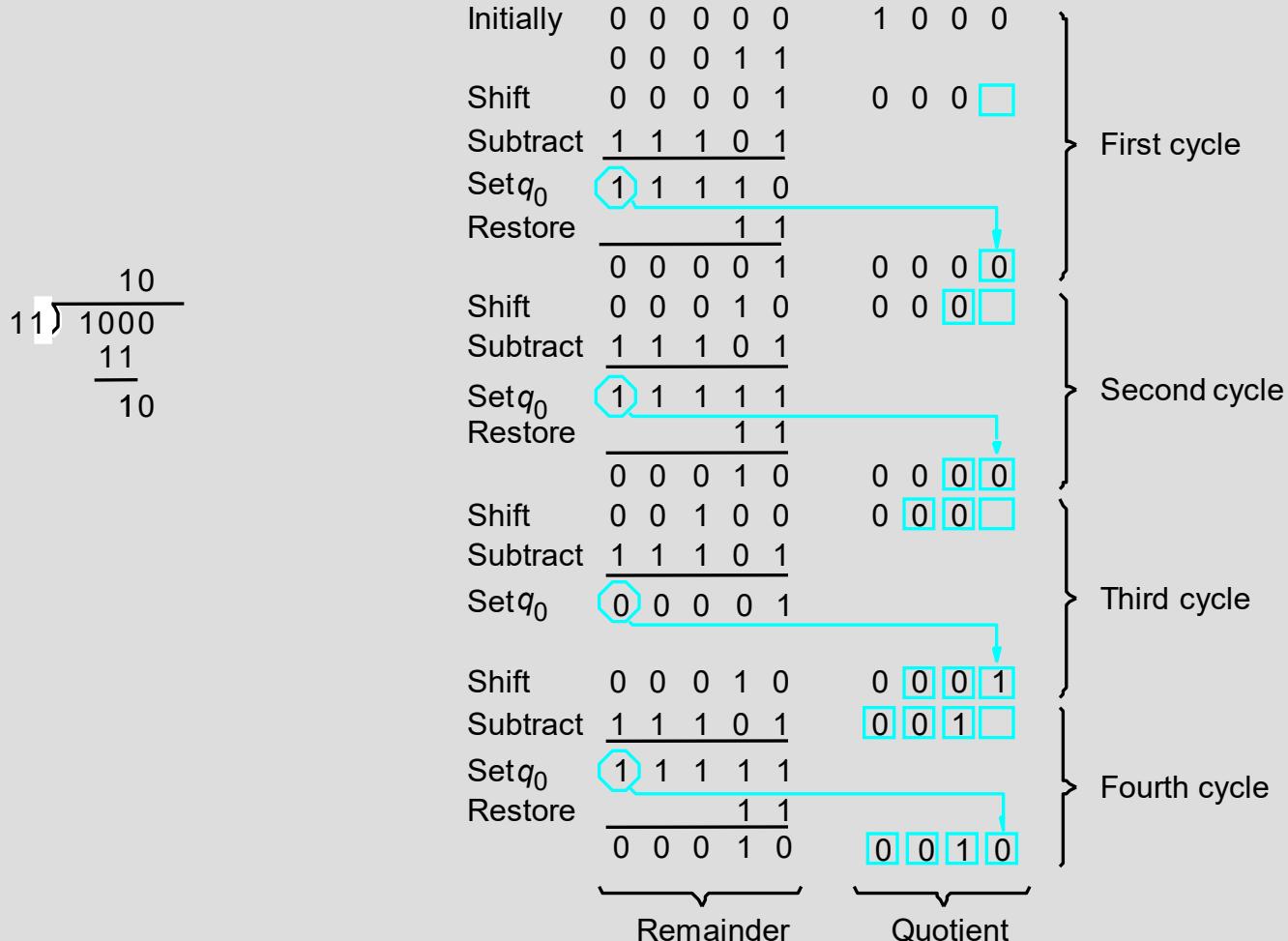
Restoring division (contd..)



*Set Register A to 0.
Load dividend in Q.
Load divisor into M.
Repeat n times:
- Shift A and Q left one bit.
- Subtract M from A.
-Place the result in A.
-If sign of A is 1, set q_0 to 0 and add M back to A.
Else set q_0 to 1.*

*End of the process:
- Quotient will be in Q.
- Remainder will be in A.*

Restoring division (contd..)



Non-restoring division

Restoring division can be improved using non-restoring algorithm

The effect of restoring algorithm actually is:

If A is positive, we shift it left and subtract M , that is compute $2A - M$

If A is negative, we restore it $(A + M)$, shift it left, and subtract M , that is, $2(A + M) - M = 2A + M$.

Set q_0 to 1 or 0 appropriately.

Non-restoring algorithm is:

Set A to 0.

Repeat n times:

If the sign of A is positive:

Shift A and Q left and subtract M . Set q_0 to 1.

Else if the sign of A is negative:

Shift A and Q left and add M . Set q_0 to 0.

If the sign of A is 1, add A to M .

Non-restoring division (contd..)

Initially	0 0 0 0 0	1 0 0 0 0	First cycle
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 0	0 0 0 □ 0	
Shift	1 1 1 0 0	0 0 0 □ □	Second cycle
Add	0 0 0 1 1		
Set q_0	1 1 1 1 1	0 0 0 □ 0	
Shift	1 1 1 1 0	0 □ 0 □ □	Third cycle
Add	0 0 0 1 1		
Set q_0	0 0 0 0 1	0 □ 0 □ 1	
Shift	0 0 0 1 0	□ □ 0 1 □	Fourth cycle
Subtract	1 1 1 0 1	□ □ 0 1 □	
Set q_0	1 1 1 1 1	□ □ 0 1 0	
Quotient			
Add	1 1 1 1 1		Restore remainder
	0 0 0 1 1		
	0 0 0 1 0		
Remainder			

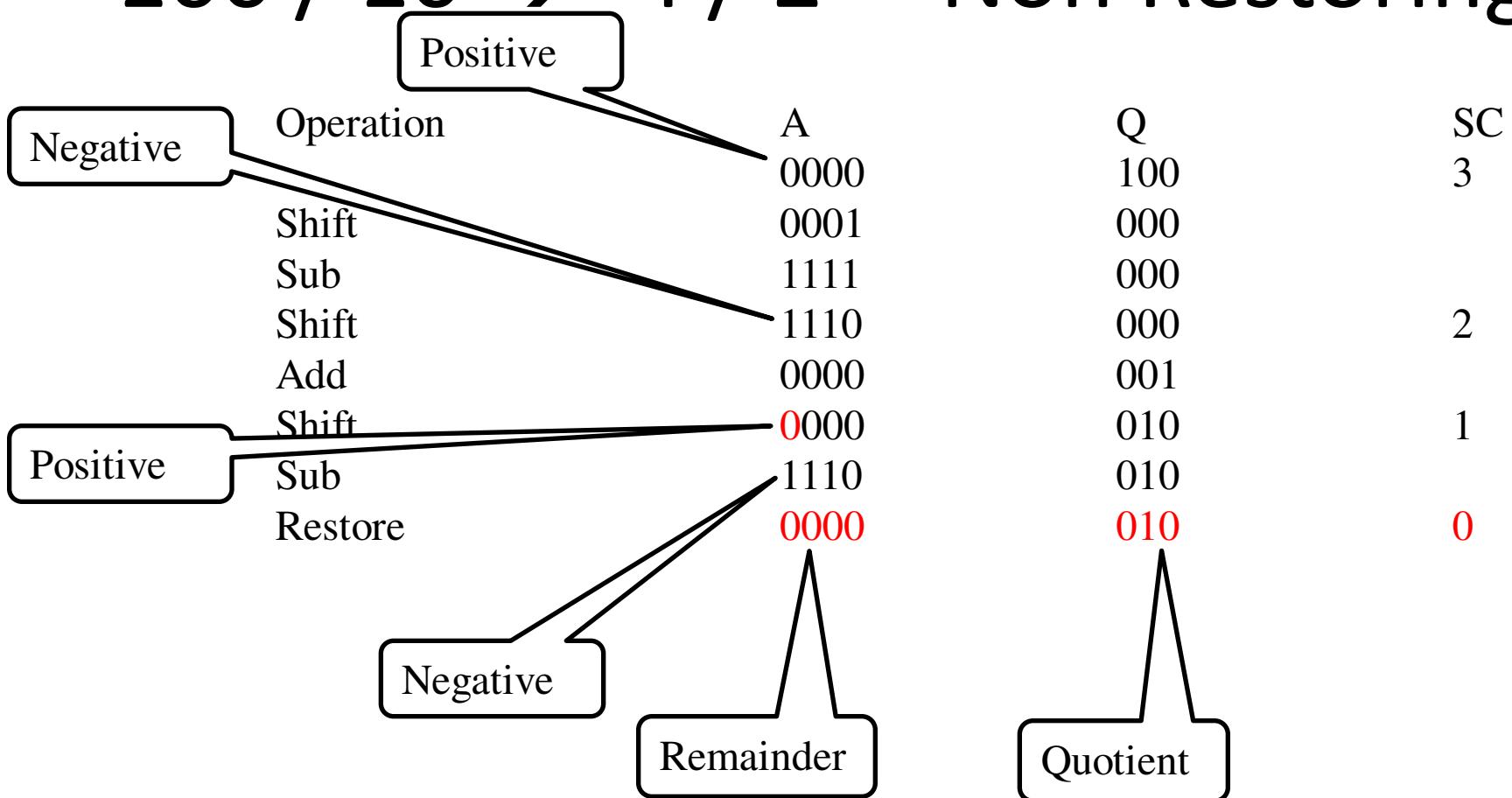
11) 1000
 11
 —
 10

$100 / 10 \rightarrow 4 / 2$ Restoring

Operation	A	Q	SC
	0000	100	3
Shift	0001	000	
Sub	1111	000	
Restore	0001	000	2
Shift	0010	000	
Sub	0000	001	1
Shift	0000	010	
Sub	1110	010	
Restore	0000	010	0

The diagram illustrates the flow of bits from the 'Positive' row to the final results. An arrow points from the 'Positive' row to the 'Remainder' box. Another arrow points from the 'Positive' row to the 'Quotient' box. The 'Remainder' box contains the value '0000'. The 'Quotient' box contains the value '010'.

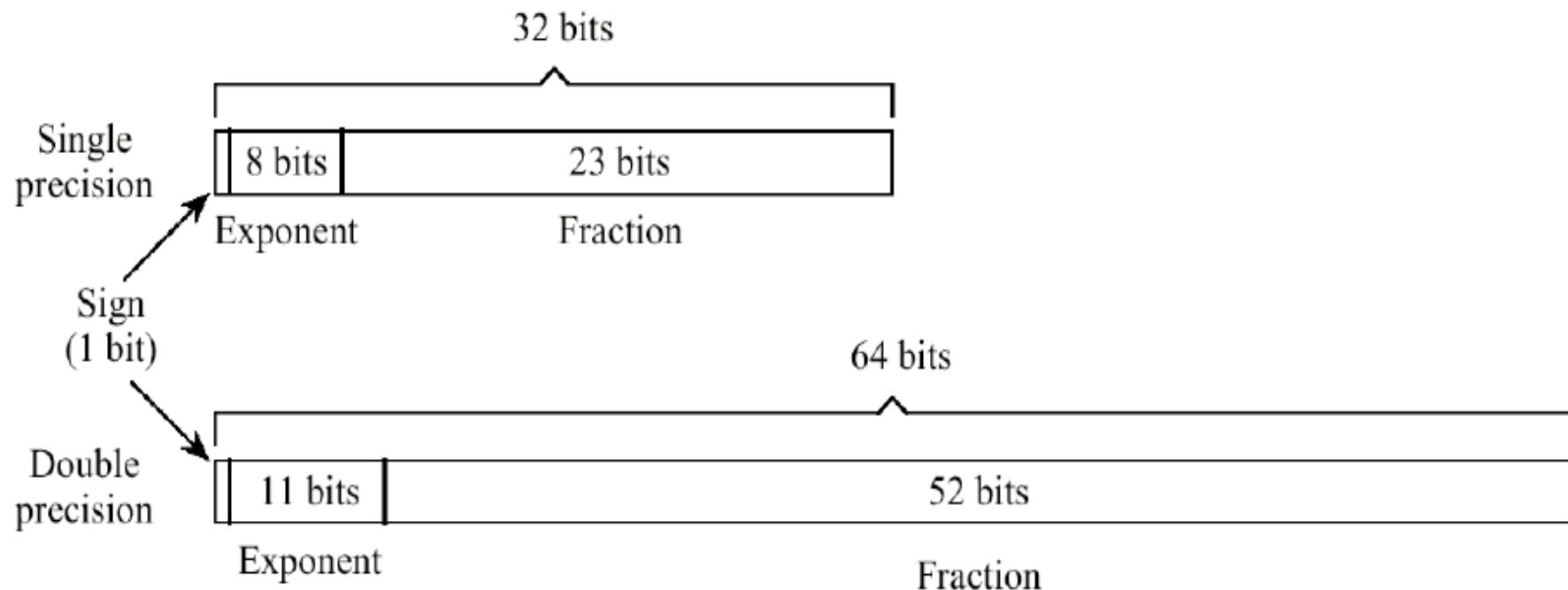
$100 / 10 \rightarrow 4 / 2$ Non Restoring



FLOATING POINT NUMBER REPRESENTATION AND ARITHMETIC OPERATIONS

LIJO V P
SCOPE
VIT, VELLORE

IEEE-754 Floating Point Formats



IEEE Floating-point Format –More explanations with examples

- IEEE has introduced a standard floating-point format for arithmetic operations in mini and microcomputer, which is defined in IEEE Standard 754
- In this format, the numbers are normalized so that the significand or mantissa lie in the range $1 \leq F < 2$, which corresponds to an integer part equal to 1
- An IEEE format floating-point number X is formally defined as:

$$X = -1^S \times 2^{E-B} \times 1.F$$

where S = sign bit [0 → +, 1 → -]

E = exponent biased by B

F = fractional mantissa

IEEE-754 Conversion Example

Represent -12.62510 in single precision IEEE-754 format.

- Step #1: Convert to target base. $-12.62510 = -1100.101_2$
- Step #2: Normalize. $-1100.101_2 = -1.100101_2 \times 2^3$
- Step #3: Fill in bit fields. Sign is negative, so sign bit is 1.
Exponent is in excess 127 (not excess 128!), so exponent is represented as the
unsigned integer $3 + 127 = 130$. Leading 1 of significant is hidden,
so

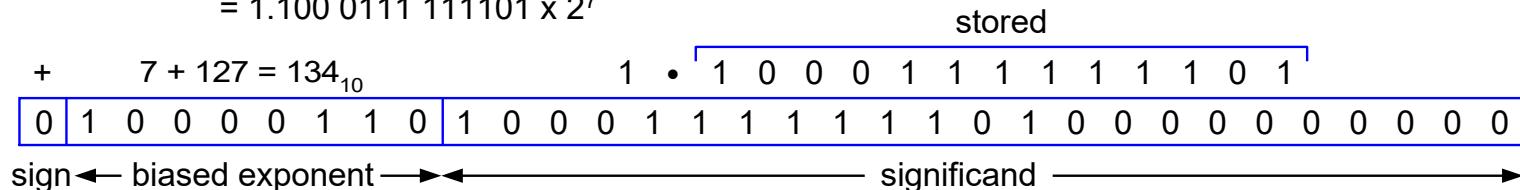
final bit pattern is:

1 1000 0010 . 1001 0100 0000 0000 0000 000

Example

Convert these number to IEEE single precision format:

$$(a) 199.953125_{10} = 1100\ 0111.111101_2 \\ = 1.100\ 0111\ 111101 \times 2^7$$



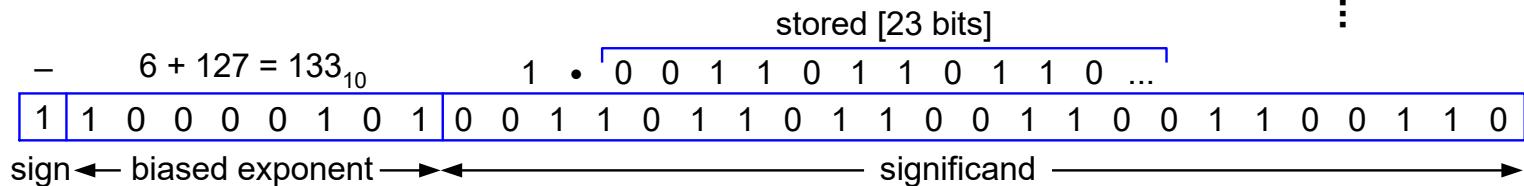
$$(b) -77.7_{10} = -100\ 1101.10110\ 0110_2 \dots \\ = -1.00\ 1101\ 101100110 \dots \times 2^6$$

$$77_{10} = 100\ 1101_2$$

$$0.7_{10} \Rightarrow 0.7 \times 2 \rightarrow \underline{1.4} \\ 0.4 \times 2 \rightarrow \underline{0.8} \\ 0.8 \times 2 \rightarrow \underline{1.6} \\ 0.6 \times 2 \rightarrow \underline{1.2} \\ 0.2 \times 2 \rightarrow \underline{0.4} \\ 0.4 \times 2 \rightarrow \underline{0.8} \\ 0.8 \times 2 \rightarrow \underline{1.6} \\ 0.6 \times 2 \rightarrow \underline{1.2} \\ 0.2 \times 2 \rightarrow \underline{0.4}$$

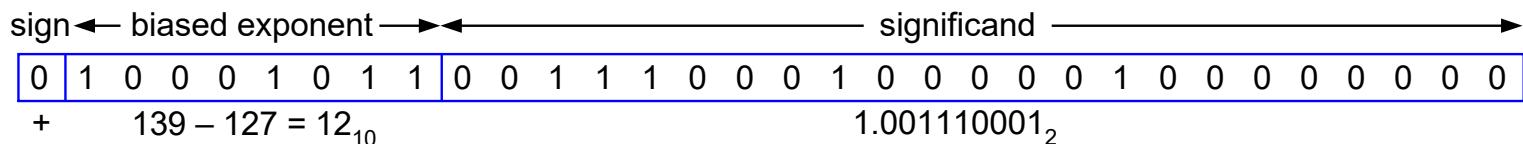
⋮

Slides adapted from tan
wooi haw's lecture notes
(FOE)



Convert these IEEE single precision floating-point numbers to their decimal equivalent:

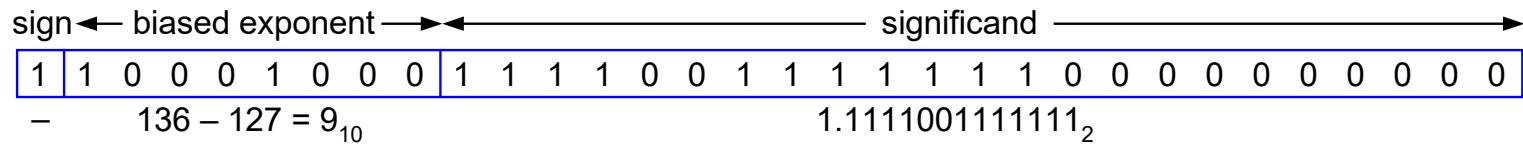
(a) 0100 0101 1001 1100 0100 0001 0000 0000₂



$$1.001110001000001_2 \times 2^{12} = 1001110001000.001_2$$

$$= 5000.125_{10}$$

(b) 1100 0100 0111 1001 1111 1100 0000 0000₂



$$-1.1111001111111_2 \times 2^9 = -1111100111.1111_2$$

$$= -999.9375_{10}$$

Slides adapted from tan
wooi haw's lecture notes
(FOE)

Character Representation ASCII

ASCII (American Standard Code for Information Interchange) Code

		MSB (3 bits)							
		0	1	2	3	4	5	6	7
LSB (4 bits)	0	NUL	DLE	SP	0	@	P	'	P
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	m	n	~
	F	SI	US	454 /	?	O	n	o	DEL

Control Character Representation (ASCII)

NUL	Null	DC1	Device Control 1
SOH	Start of Heading (CC)	DC2	Device Control 2
STX	Start of Text (CC)	DC3	Device Control 3
ETX	End of Text (CC)	DC4	Device Control 4
EOT	End of Transmission (CC)	NAK	Negative Acknowledge (CC)
ENQ	Enquiry (CC)	SYN	Synchronous Idle (CC)
ACK	Acknowledge (CC)	ETB	End of Transmission Block (CC)
BEL	Bell	CAN	Cancel
BS	Backspace (FE)	EM	End of Medium
HT	Horizontal Tab. (FE)	SUB	Substitute
LF	Line Feed (FE)	ESC	Escape
VT	Vertical Tab. (FE)	FS	File Separator (IS)
FF	Form Feed (FE)	GS	Group Separator (IS)
CR	Carriage Return (FE)	RS	Record Separator (IS)
SO	Shift Out	US	Unit Separator (IS)
SI	Shift In	DEL	Delete
DLE	Data Link Escape (CC)		

(CC) Communication Control

(FE) Format Effector

(IS) Information Separator

The EBCDIC character code, shown with hexadecimal indices

00	NUL	20	DS	40	SP	60	-	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	'	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(6D	-	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F		6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	
1B	CU1	3B	CU3	5B	\$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	.	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	:	7E	456	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	-	7F	"	9F		BF		DF		FF	

The EBCDIC control character representation

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

References

Text Book

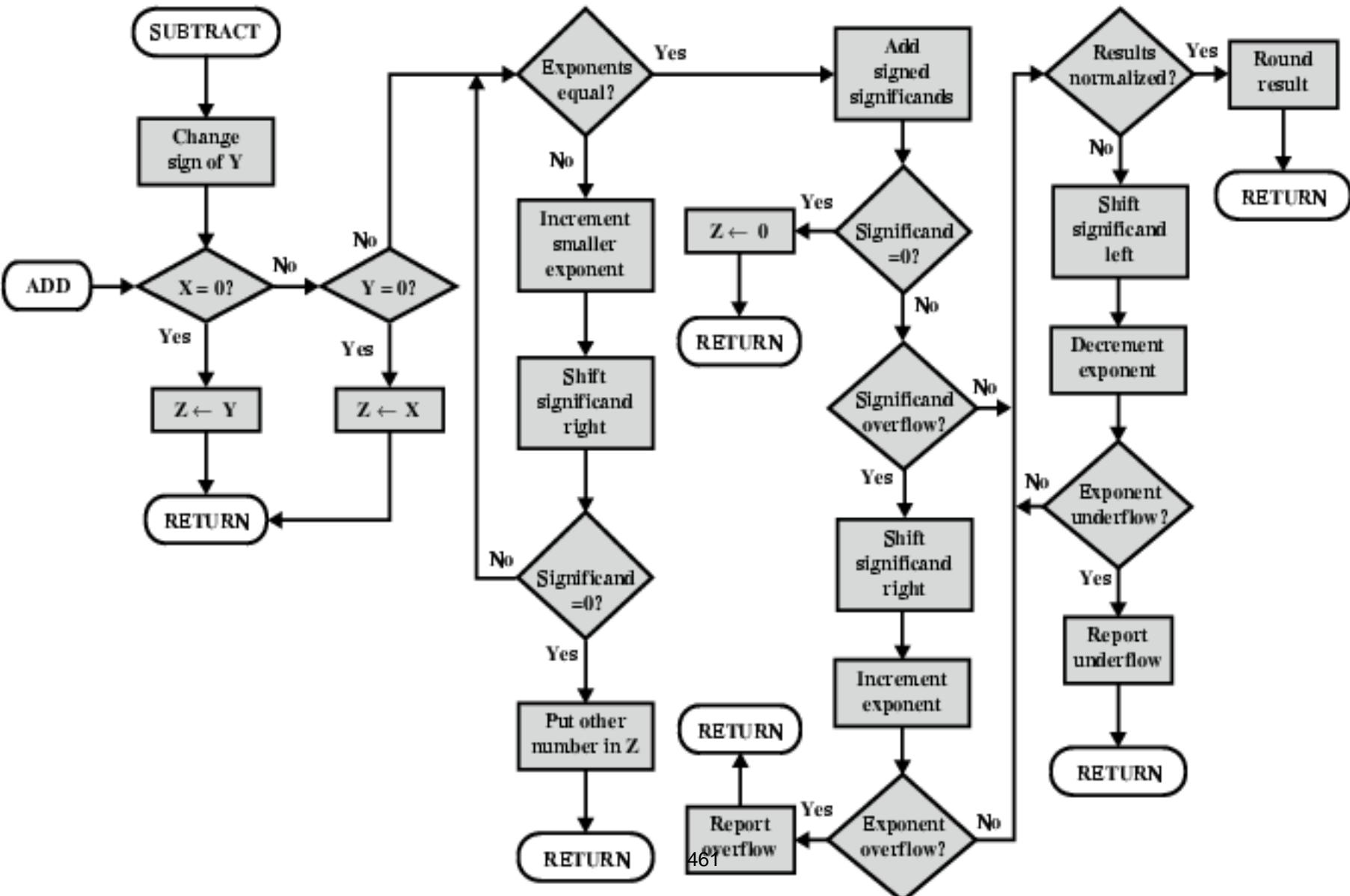
- M. M. Mano, Computer System Architecture, Prentice-Hall, 2004
- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

Floating Point Operations

FP Arithmetic +/-

- Check for zeros
- Align Mantissa (adjusting exponents)
- Add or subtract Mantissa's
- Normalize the result

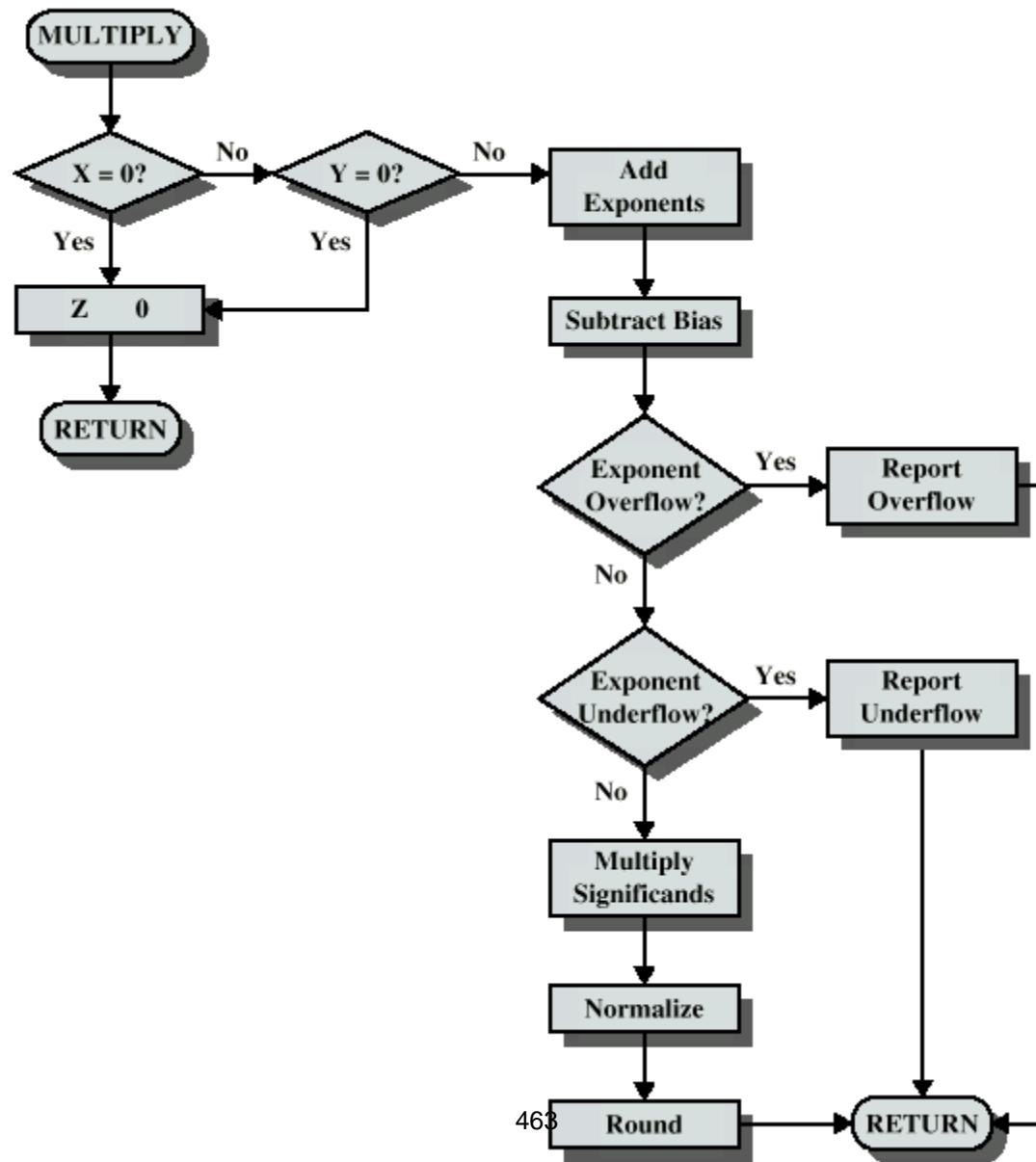
Floating Point Addition & Subtraction Flowchart



Floating Point Multiplication

- Check for zero
- Add exponents
- Multiply Mantissa's
- Normalize
- Round
- All intermediate results should be in double length storage

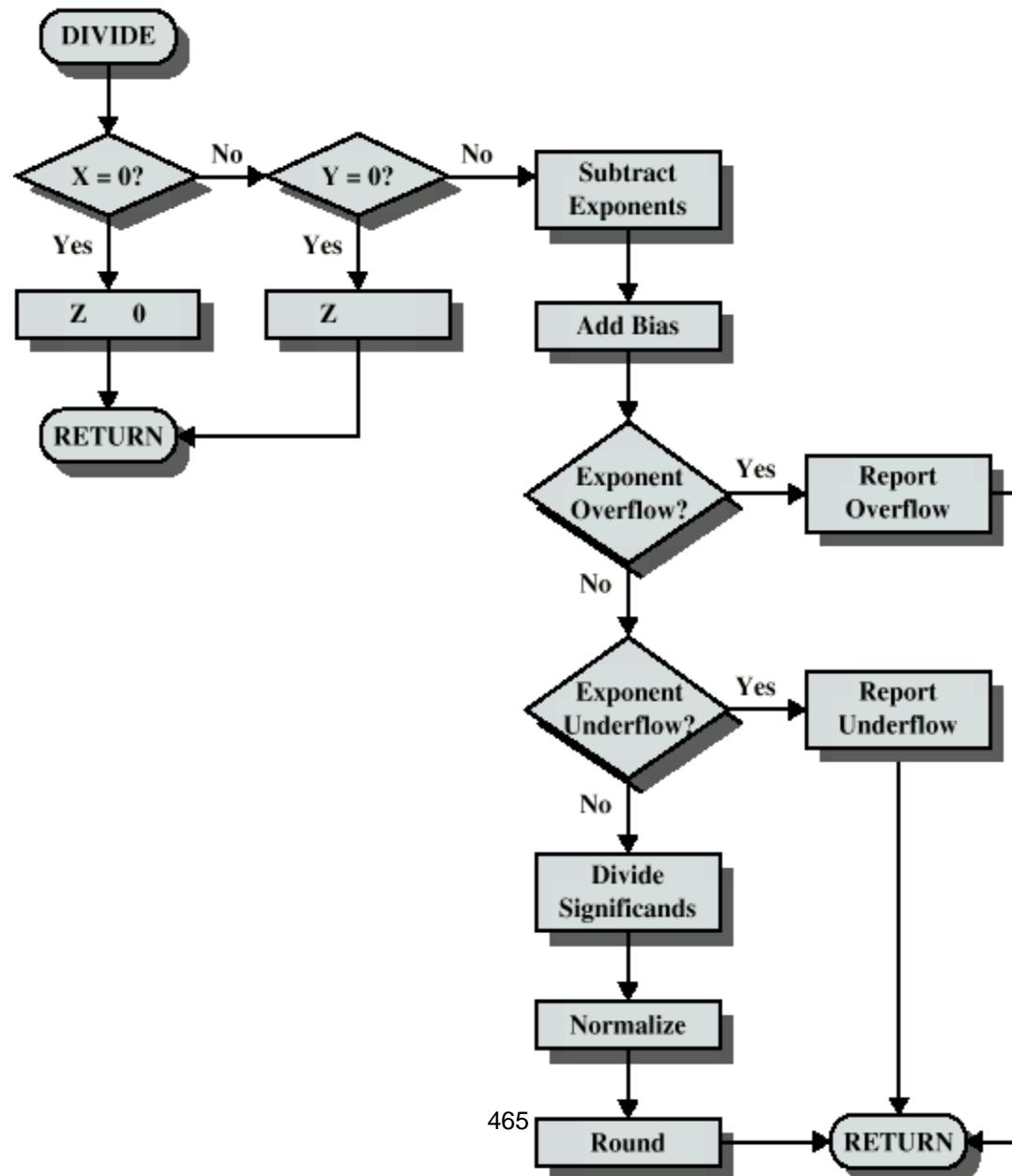
Floating Point Multiplication



Floating Point Division

- Check for zero
- Subtract exponents
- Divide Mantissa's
- Normalize
- Round

Floating Point Division



Thank You

IEEE Floating-point Format –More explanations with examples

- IEEE has introduced a standard floating-point format for arithmetic operations in mini and microcomputer, which is defined in IEEE Standard 754
- In this format, the numbers are normalized so that the significand or mantissa lie in the range $1 \leq F < 2$, which corresponds to an integer part equal to 1
- An IEEE format floating-point number X is formally defined as:

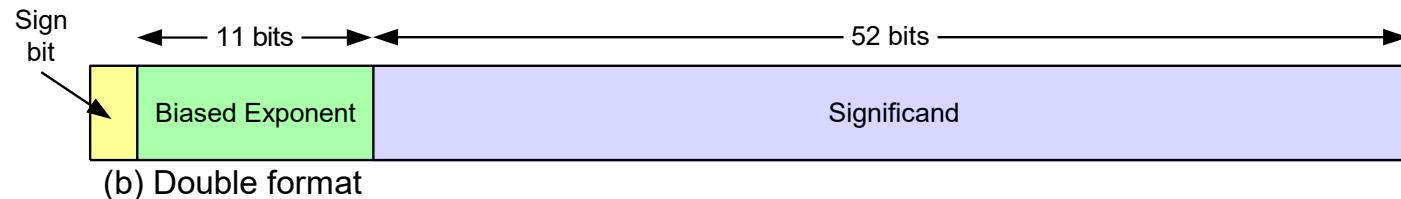
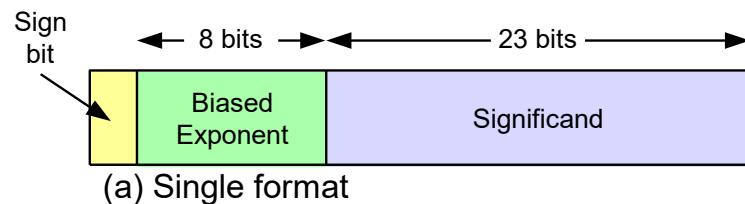
$$X = -1^S \times 2^{E-B} \times 1.F$$

where S = sign bit [0 → +, 1 → -]

E = exponent biased by B

F = fractional mantissa

- Two basic formats are defined in the IEEE Standard 754
- These are the 32-bit single and 64-bit double formats, with 8-bit and 11-bit exponent respectively



- A sign-magnitude representation has been adopted for the mantissa; mantissa is negative if $S = 1$, and positive if $S = 0$

Floating Point Examples



negative

(a) Format

20

$$127 + 20 = 147$$

1.1010001 X	$2^{10100} = 0$	10010011 10100010000000000000000
-1.1010001 X	$2^{10100} = 1$	10010011 10100010000000000000000
1.1010001 X	$2^{-10100} = 0$	01101011 10100010000000000000000
-1.1010001 X	$2^{-10100} = 1$	01101011 10100010000000000000000

normalized (b) Examples

-20

$$127 - 20 = 107$$

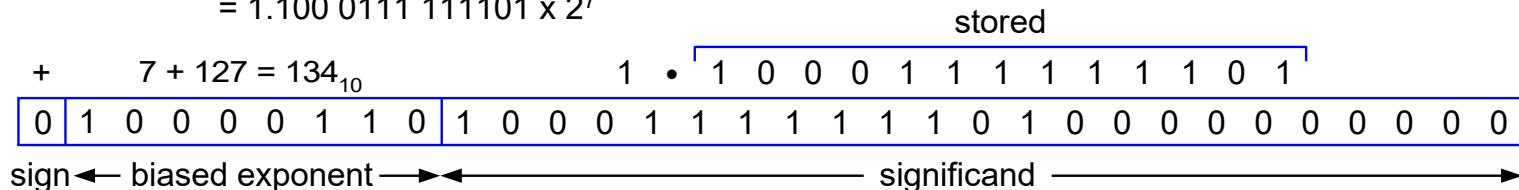
$$\begin{aligned}
 &= 1.638125 \times 2^{20} \\
 &= -1.638125 \times 2^{20} \\
 &= 1.638125 \times 2^{-20} \\
 &= -1.638125 \times 2^{-20}
 \end{aligned}$$

The bias equals to $(2^{K-1} - 1) \rightarrow 2^{8-1} - 1 = 127$

Example

Convert these number to IEEE single precision format:

$$(a) 199.953125_{10} = 1100\ 0111.111101_2 \\ = 1.100\ 0111\ 111101 \times 2^7$$

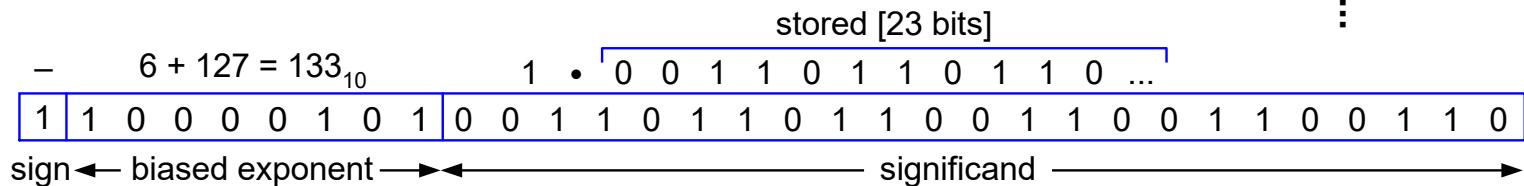


$$(b) -77.7_{10} = -100\ 1101.10110\ 0110_2 \dots \\ = -1.00\ 1101\ 101100110 \dots \times 2^6$$

Slides adapted from tan
wooi haw's lecture notes
(FOE)

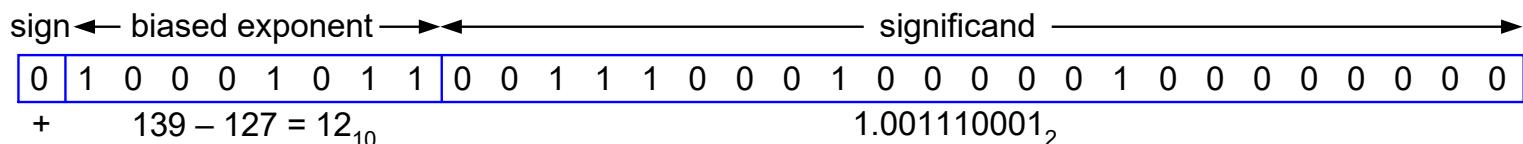
$$77_{10} = 100\ 1101_2$$

$$\begin{aligned} 0.7_{10} &\Rightarrow 0.7 \times 2 \rightarrow \underline{1.4} \\ 0.4 \times 2 &\rightarrow \underline{0.8} \\ 0.8 \times 2 &\rightarrow \underline{1.6} \\ 0.6 \times 2 &\rightarrow \underline{1.2} \\ 0.2 \times 2 &\rightarrow \underline{0.4} \\ 0.4 \times 2 &\rightarrow \underline{0.8} \\ 0.8 \times 2 &\rightarrow \underline{1.6} \\ 0.6 \times 2 &\rightarrow \underline{1.2} \\ 0.2 \times 2 &\rightarrow \underline{0.4} \end{aligned}$$



Convert these IEEE single precision floating-point numbers to their decimal equivalent:

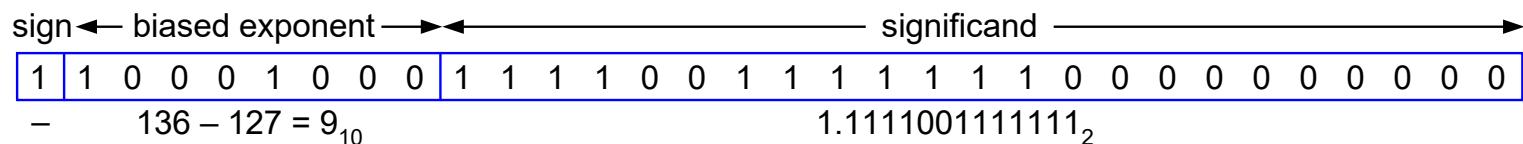
(a) 0100 0101 1001 1100 0100 0001 0000 0000₂



$$1.001110001000001_2 \times 2^{12} = 1001110001000.001_2$$

$$= 5000.125_{10}$$

(b) 1100 0100 0111 1001 1111 1100 0000 0000₂



$$-1.1111001111111_2 \times 2^9 = -1111100111.1111_2$$

$$= -999.9375_{10}$$

Slides adapted from tan
wooi haw's lecture notes
(FOE)

FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

FP Arithmetic x/\div

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Floating-point Arithmetic (cont.)

Table 9.5 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_s \times B^{X_E}$ $Y = Y_s \times B^{Y_E}$	$X + Y = \left(X_s \times B^{X_E - Y_E} + Y_s \right) \times B^{Y_E}$ $X - Y = \left(X_s \times B^{X_E - Y_E} - Y_s \right) \times B^{Y_E} \quad X_E \leq Y_E$ $X \times Y = (X_s \times Y_s) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_s}{Y_s} \right) \times B^{X_E - Y_E}$

Examples: Some basic floating-point arithmetic operations are shown in the table

$$X = 0.3 \times 10^3 = 30$$

$$Y = 0.2 \times 10^2 = 200$$

$$X + Y = (0.3 \times 10^{3-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$

$$X - Y = (0.3 \times 10^{3-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$

$$X \times Y = (0.3 \times 0.2) \times 10^{3+2} = 0.06 \times 10^5 = 6000$$

$$X \div Y = (0.3 \div 0.2) \times 10^{3-2} = 1.5 \times 10^{-1} = 0.15$$

Floating-point Arithmetic (cont.)

- For addition and subtraction, it is necessary to ensure that both operand exponents have the same value
- This may involves shifting the radix point of one of the operand to achieve alignment

Floating-point Arithmetic (cont.)

- Some problems that may arise during arithmetic operations are:
 - i. Exponent overflow: A positive exponent exceeds the maximum possible exponent value and this may leads to $+\infty$ or $-\infty$ in some systems
 - ii. Exponent underflow: A negative exponent is less than the minimum possible exponent value (eg. 2^{-200}), the number is too small to be represented and maybe reported as 0
 - iii. Significand underflow: In the process of aligning significands, the smaller number may have a significand which is too small to be represented
 - iv. Significand overflow: The addition of two significands of the same sign may result in a carry out from the most significant bit

FP Arithmetic +/-

- Unlike integer and fixed-point number representation, floating-point numbers cannot be added in one simple operation
- Consider adding two decimal numbers:

$$A = 12345$$

$$B = 567.89$$

If these numbers are normalized and added in floating-point format, we will have

$$\begin{array}{r} 0.12345 \times 10^5 \\ + 0.56789 \times 10^3 \\ \hline \textcolor{red}{?????} \times 10^? \end{array}$$

Obviously, direct addition cannot take place as the exponents are different

FP Arithmetic +/- (cont.)

- Floating-point addition and subtraction will typically involve the following steps:
 - i. Align the significand
 - ii. Add or subtract the significands
 - iii. Normalize the result
- Since addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation
- The floating-point numbers can only be added if the two exponents are equal
- This can be done by aligning the smaller number with the bigger number [increasing its exponent] or vice-versa, so that both numbers have the same exponent

FP Arithmetic +/- (cont.)

- As the aligning operation may result in the loss of digits, it is the smaller number that is shifted so that any lost will therefore be of relatively insignificant

$$1.1001 \times 2^9 \xrightarrow{\text{shift left}} \underline{1}10010000 \times 2^1 \quad 1 \times 2^9 \text{ is lost}$$

$$1.0111 \times 2^1 \longrightarrow 1.0111000 \times 2^1$$

- Hence, the smaller number are shifted right by increasing its exponent until the two exponents are the same
- If both numbers have exponents that differ significantly, the smaller number is lost as a result of shifting

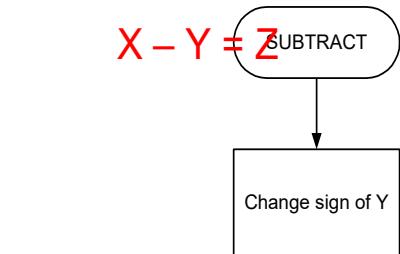
$$1.1001001 \times 2^9 \longrightarrow 1.1001001 \times 2^9$$

$$1.0110001 \times 2^1 \xrightarrow{\text{shift right}} 0.0000000 \times 2^9$$

FP Arithmetic +/- (cont.)

$$\begin{array}{r}
 1.1101 \times 2^4 \\
 + 0.0101 \times 2^4 \\
 \hline
 10.0010 \times 2^4 \rightarrow 1.0001 \times 2^5
 \end{array}$$

- After the numbers have been aligned, they are added together taking into account their signs
- There might be a possibility of significand overflow due to a carry out from the most significant bit
- If this occurs, the significand of the result is shifted right and the exponent is incremented
- As the exponents are incremented, it might overflows and the operation will stop
- Lastly, the result is normalized by shifting significand digits left until the most significant digit is non-zero
- Each shift causes a decrement of the exponent and thus could cause an exponent underflow
- Finally, the result is rounded off and reported

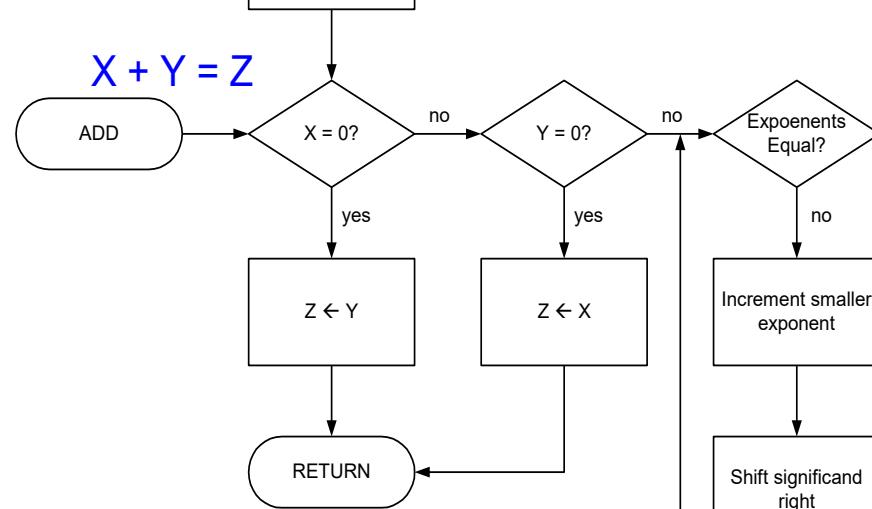


$$X = 1.01101 \times 2^7$$

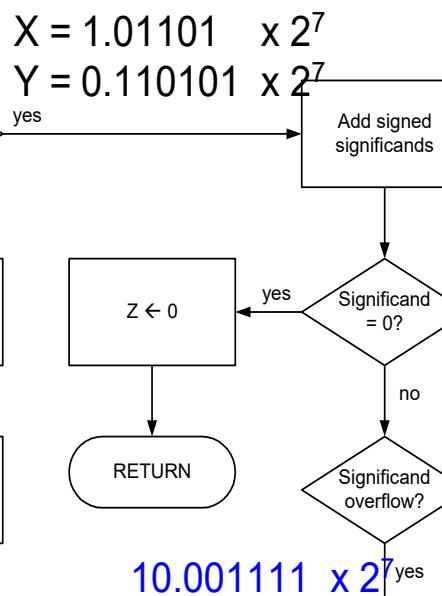
$$Y = 1.10101 \times 2^6$$

$$\begin{array}{r} 1.01101 \times 2^7 \\ + 0.110101 \times 2^7 \\ \hline 10.001111 \times 2^7 \end{array}$$

$$\begin{array}{r} 1.01101 \times 2^7 \\ - 0.110101 \times 2^7 \\ \hline 0.100101 \times 2^7 \end{array}$$



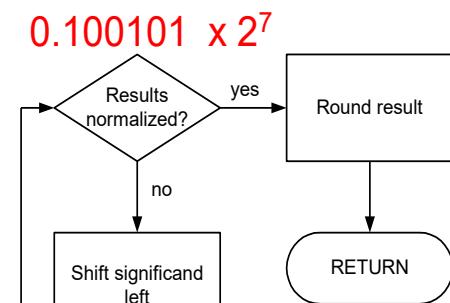
$$Y = 0.110101 \times 2^6$$



$$\begin{array}{r} 1.0001111 \times 2^8 \\ \text{RETURN} \\ \hline \end{array}$$

$$\begin{array}{r} \text{Report overflow} \\ \text{yes} \\ \hline \end{array}$$

$$\begin{array}{r} \text{Exponent overflow?} \\ \text{no} \\ \hline \end{array}$$



Slides adapted from tan wooi haw's lecture notes (FOE)

FP Arithmetic +/- (cont.)

- Some of the floating-point arithmetic will lead to an increase number of bits in the mantissa
- For example, consider adding these 5 significant bits floating-point numbers:

$$A = 0.11001 \times 2^4$$

$$B = 0.10001 \times 2^3$$

$$A = 0.11001 \times 2^4$$

$$\begin{array}{r} B = 0.010001 \times 2^4 \\ \hline 1.000011 \times 2^4 \end{array} \xrightarrow{\text{normalize}} 0.10000\underline{11} \times 2^5$$

- The result has two extra bit of precision which cannot be fitted into the floating point format
- For simplicity, the number can be truncated to give 0.10000×2^5

FP Arithmetic +/- (cont.)

- Truncation is the simplest method which involves nothing more than taking away the extra bits
- A much better technique is rounding in which if the value of the extra bits is greater than half the least significant bit of the retained bits, 1 is added to the LSB of the remaining digits
- For example, consider rounding these numbers to 4 significant bits:
 - i. 0.1101101
extra bits → 0.0000101
LSB of retained bits → 0.0001

$$\begin{array}{r}
 0.1\ 1\ 0\ 1\ 1\ 0\ 1 \\
 \text{more than half} \\
 \hline
 \end{array}
 \rightarrow
 \begin{array}{r}
 0.1101 \\
 + \quad \quad \quad 1 \\
 \hline
 0.1110
 \end{array}$$

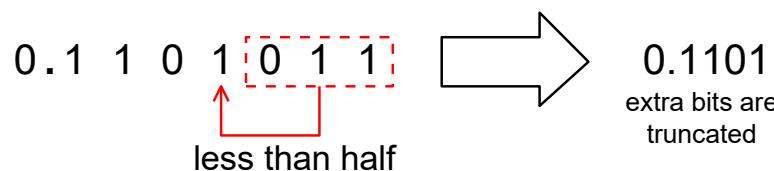
add 1 to the
LSB

FP Arithmetic +/- (cont.)

ii. 0.1101011

extra bits \rightarrow 0.0000011

LSB of retained bits \rightarrow 0.0001



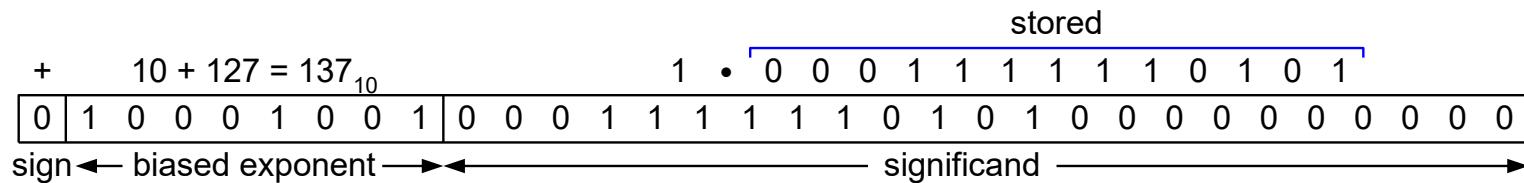
- Truncation always undervalues the result, leading to a systematic error, whereas rounding sometimes reduces the result and sometimes increases it
- Rounding is always preferred to truncation partly because it is more accurate and partly it gives rise to an unbiased error
- Major disadvantage of rounding is that it requires a further arithmetic operation on the result

Example

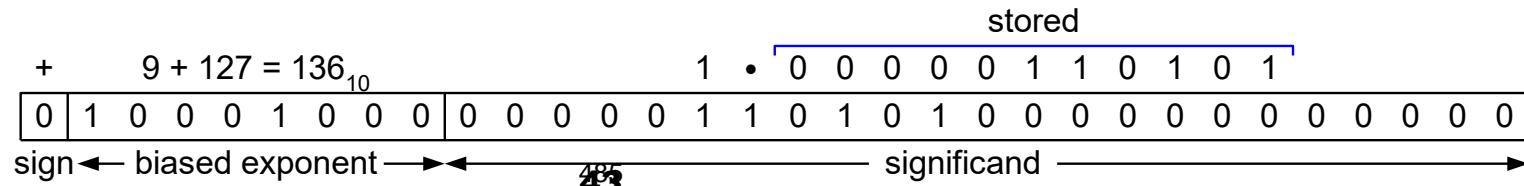
Perform the following arithmetic operation using floating point arithmetic, In each case, show how the numbers would be stored using IEEE single-precision format

i. $1150.625_{10} - 525.25_{10}$

$$\begin{aligned} 1150.625_{10} &= 100\ 0111\ 1110.\ 101_2 \\ &= 1.\ 0001\ 1111\ 10101 \times 2^{10} \end{aligned}$$



$$\begin{aligned} 525.25_{10} &= 10\ 0000\ 1101.01_2 \\ &= 1.\ 0000\ 0110\ 101 \times 2^9 \end{aligned}$$



continue ...

As these numbers have different exponents, the smaller number is shifted right to align with the larger number

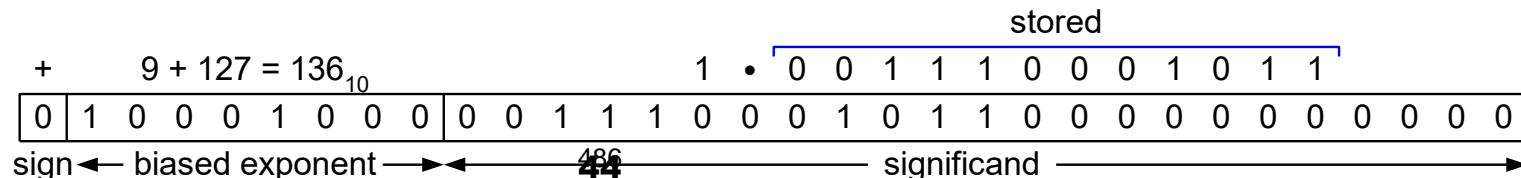
$$\begin{array}{rccccc}
 1000 & 1000 & 1.00000110101 & \rightarrow & 1000 & 1001 & 0.100000110101 \\
 \text{exponent} & & \text{mantissa} & & \text{exponent} & & \text{mantissa}
 \end{array}$$

Subtract the mantissa

$$\begin{array}{r}
 1.000111110101 \\
 - 0.100000110101 \\
 \hline
 0.1001110001011
 \end{array}$$

Normalize the result

$$\begin{array}{rccccc}
 1000 & 1001 & 0.1001110001011 & \rightarrow & 1000 & 1000 & 1.001110001011 \\
 \text{exponent} & & \text{mantissa} & & \text{exponent} & & \text{mantissa}
 \end{array}$$



continue ...

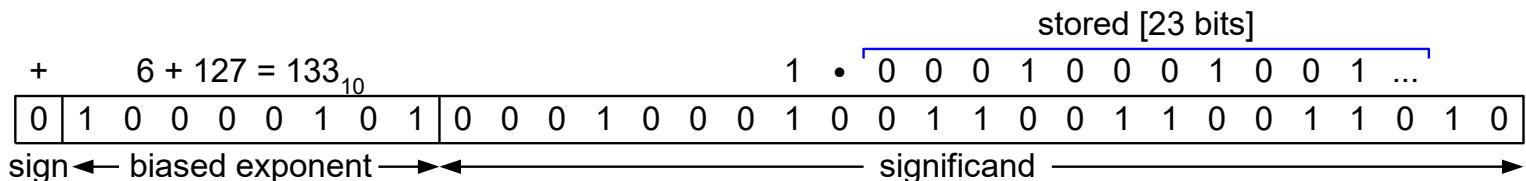
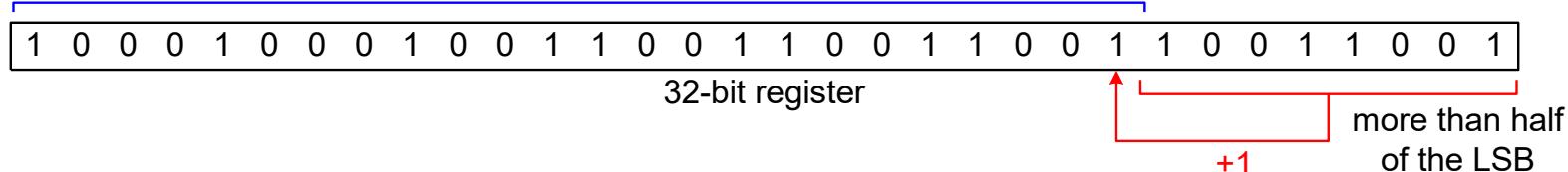
ii. $68.3_{10} + 12.2_{10}$

$$\begin{aligned} 68.3_{10} &= 100\ 0100.01001\ 1001 \dots \\ &= 1.00\ 0100\ 01001\ 1001 \dots \times 2^6 \end{aligned}$$

$$\begin{aligned} 68_{10} &= 100\ 0100_2 \\ 0.3_{10} &\Rightarrow 0.3 \times 2 \rightarrow \underline{0.6} \\ &\quad 0.6 \times 2 \rightarrow \underline{1.2} \\ &\quad 0.2 \times 2 \rightarrow \underline{0.4} \\ &\quad 0.4 \times 2 \rightarrow \underline{0.8} \\ &\quad 0.8 \times 2 \rightarrow \underline{1.6} \\ &\quad 0.6 \times 2 \rightarrow \underline{1.2} \end{aligned}$$

⋮

only 24 bits can be stored

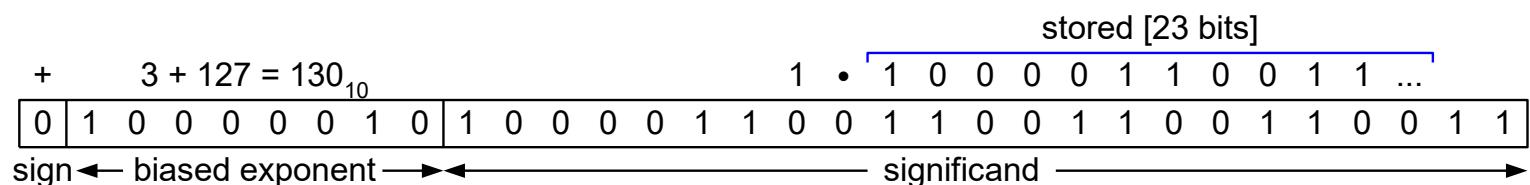
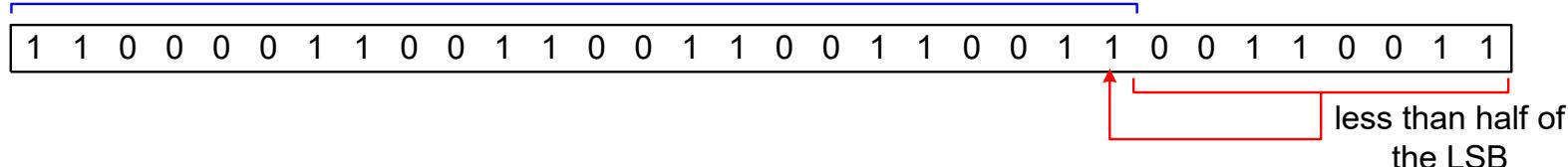


continue ...

$$\begin{aligned}12.2_{10} &= 1100.0011\ 0011\dots \\&= 1.100\ 0011\ 0011\dots \times 2^3\end{aligned}$$

$$\begin{aligned}12_{10} &= 1100_2 \\0.2_{10} \Rightarrow & 0.2 \times 2 \rightarrow 0.4 \\& 0.4 \times 2 \rightarrow 0.8 \\& 0.8 \times 2 \rightarrow 1.6 \\& 0.6 \times 2 \rightarrow 1.2 \\& 0.2 \times 2 \rightarrow 0.4 \\&\vdots\end{aligned}$$

only 24 bits can be stored



continue ...

Align the smaller number with the larger number by shifting it to the right [increasing the exponent]

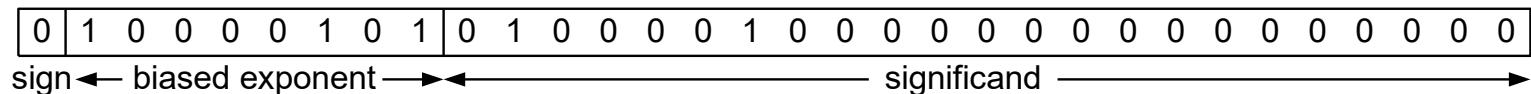
1000 0010	1.1000011001100110011	→	1000 0101	0.0011000011001100110011
exponent	mantissa		exponent	mantissa

ADD the mantissa

$$\begin{array}{r}
 1.00010001001100110011010 \\
 + 0.0011000011001100110011 \\
 \hline
 1.01000010000000000000000011
 \end{array}$$

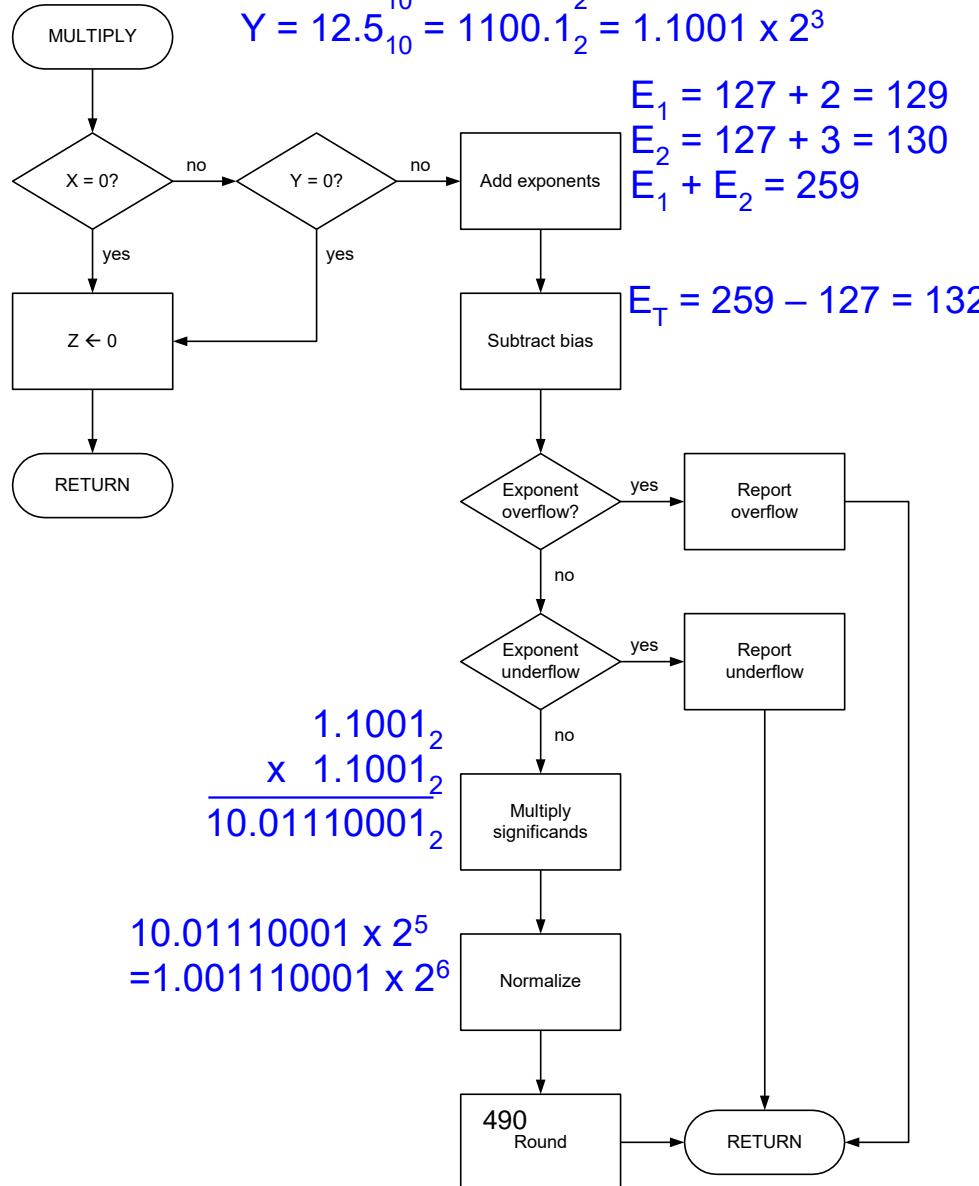
 less than half
 of the LSB

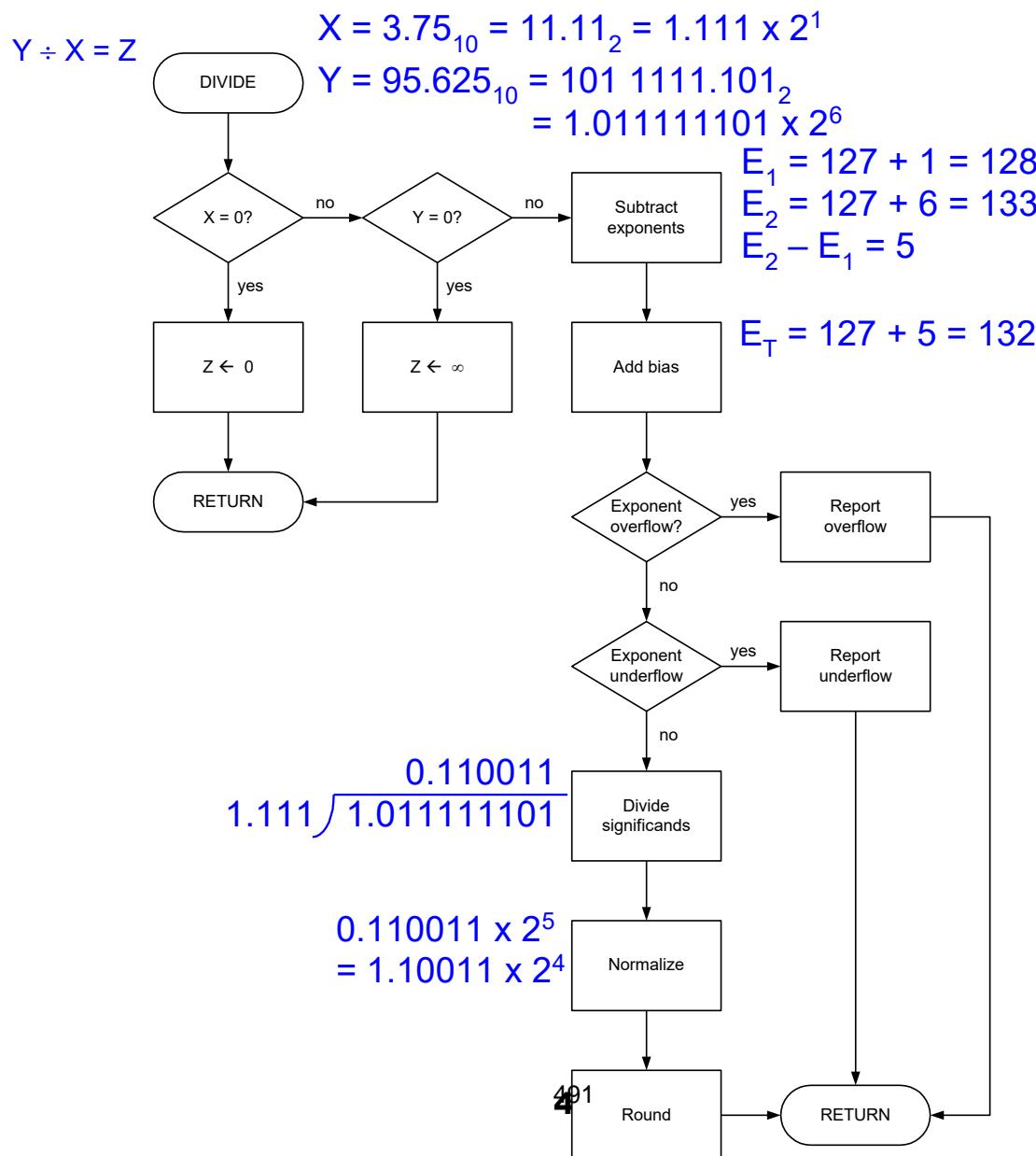
Store the result in IEEE single-precision format



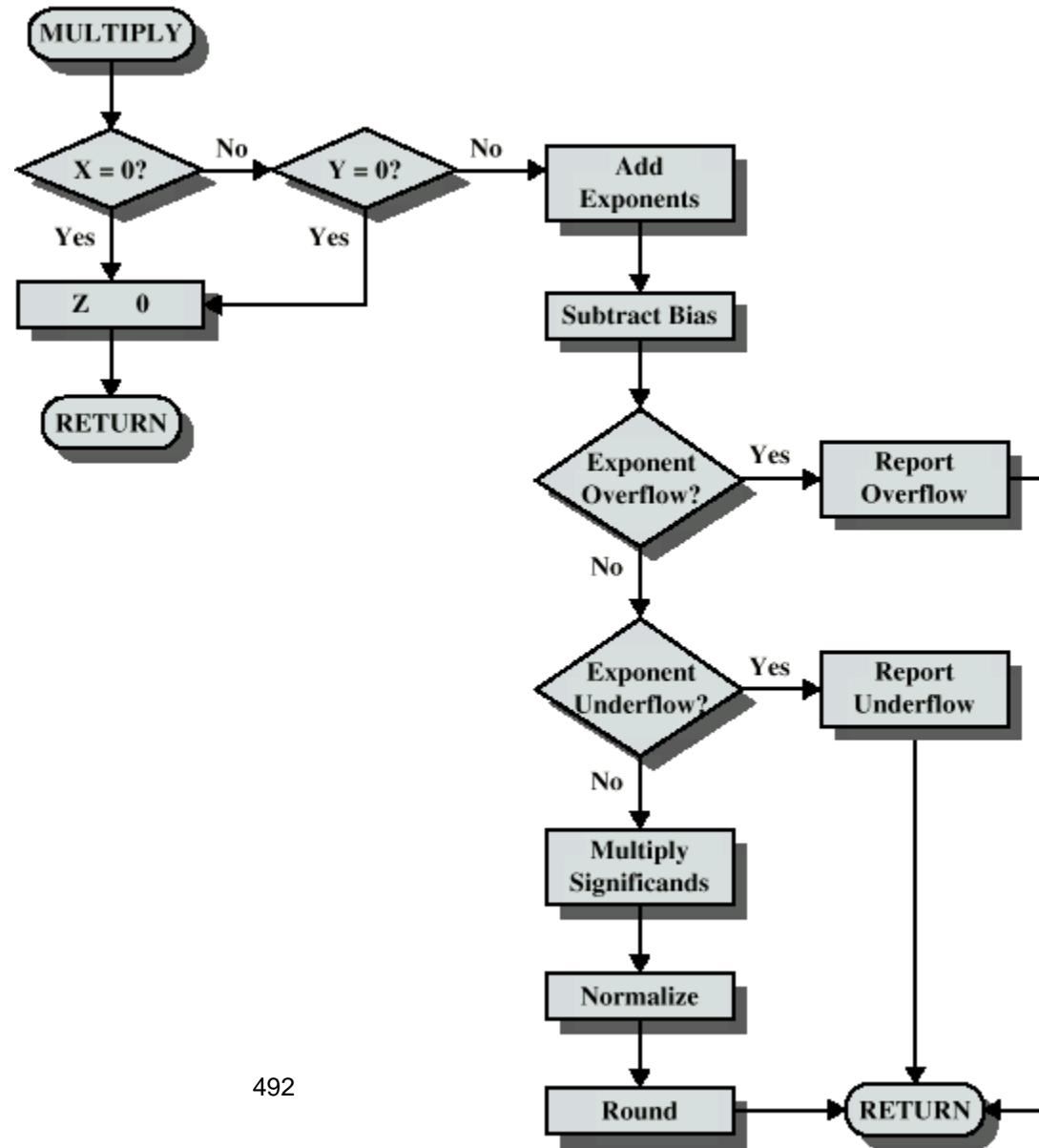
Floating-point Multiplication

$$X \times Y = Z$$

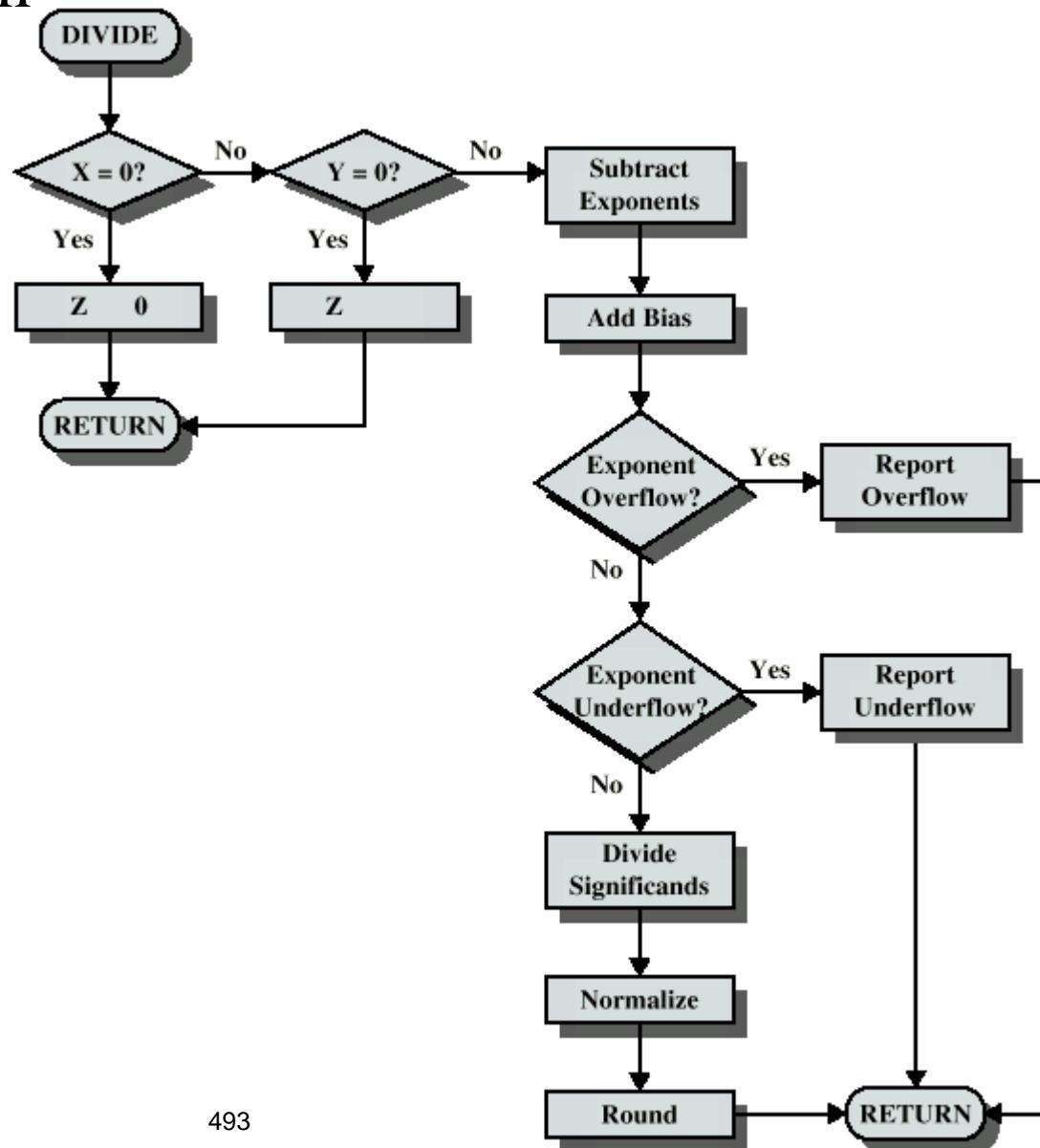




Floating Point Multiplication



Floating Point Division



PROBLEM (1)

- Express the number - $(640.5)_{10}$ in IEEE 32 bit and 64 bit floating point format

SOLUTION (1)....

- IEEE 32 BIT FLOATING POINT FORMAT

MSB	8 bits	23 bits
sign	Biased Exponent	Mantissa/Significand (Normalized)

Step 1: Express the given number in binary form

$$(640.5) = 1010000000.1 * 2^0$$

Step 2: Normalize the number into the form 1.bbbbbbb

$$1010000000.1 * 2^0 = 1.010000001 * 2^9$$

Once Normalized, every number will have 1 at the leftmost bit. So IEEE notation is saying that there is no need to store this bit. Therefore significand to be stored is 0100 0000 0100 0000 0000 000 in the allotted 23 bits

SOLUTION (1).....

- Step 3: For the 8 bit biased exponent field, the bias used is

$$2^{k-1}-1 = 2^{8-1}-1 = 127$$

Add the bias 127 to the exponent 9 and convert it into binary in order to store for 8-bit biased exponent.

$$127 + 9 = 136 \quad ($$

$1000 \ 1000)$

- Step 4: Since the given number is negative, put MSB as 1
- Step 5: Pack the result into proper format(IEEE 32 bit)

1	1000 1000	0100 0000 0010 0000 0000 000
---	-----------	------------------------------

SOLUTION (1).....

- IEEE 64 BIT FLOATING POINT FORMAT

MSB	11 bits	52 bits
sign	Biased Exponent	Mantissa/Significand (Normalized)

Step 1: Express the given number in binary form

$$(640.5) = 1010000000.1 * 2^0$$

Step 2: Normalize the number into the form 1.bbbbbbb

$$1010000000.1 * 2^0 = 1.010000001 * 2^9$$

Once Normalized, every number will have 1 at the leftmost bit. So IEEE notation is saying that there is no need to store this bit. Therefore significand to be stored is 0100 0000 0100 0000 0000 0000 0000 0000 0000 0000 in the allotted 52 bits

SOLUTION (1)...

- Step 3: For the 11 bit biased exponent field, the bias used is

$$2^{k-1}-1 = 2^{11-1}-1 = 1023$$

Add the bias 1023 to the exponent 9 and convert it into binary in order to store for 11-bit biased exponent. $1023 + 9$

1023 + 9

=1032 (1000 0001 000)

- Step 4: Since the given number is negative, put MSB as 1
 - Step 5: Pack the result into proper format(IEEE 64 bit)

References

Text Book

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- <http://courses.cs.tamu.edu/rabi/cpsc321/lectures/lec06.ppt>

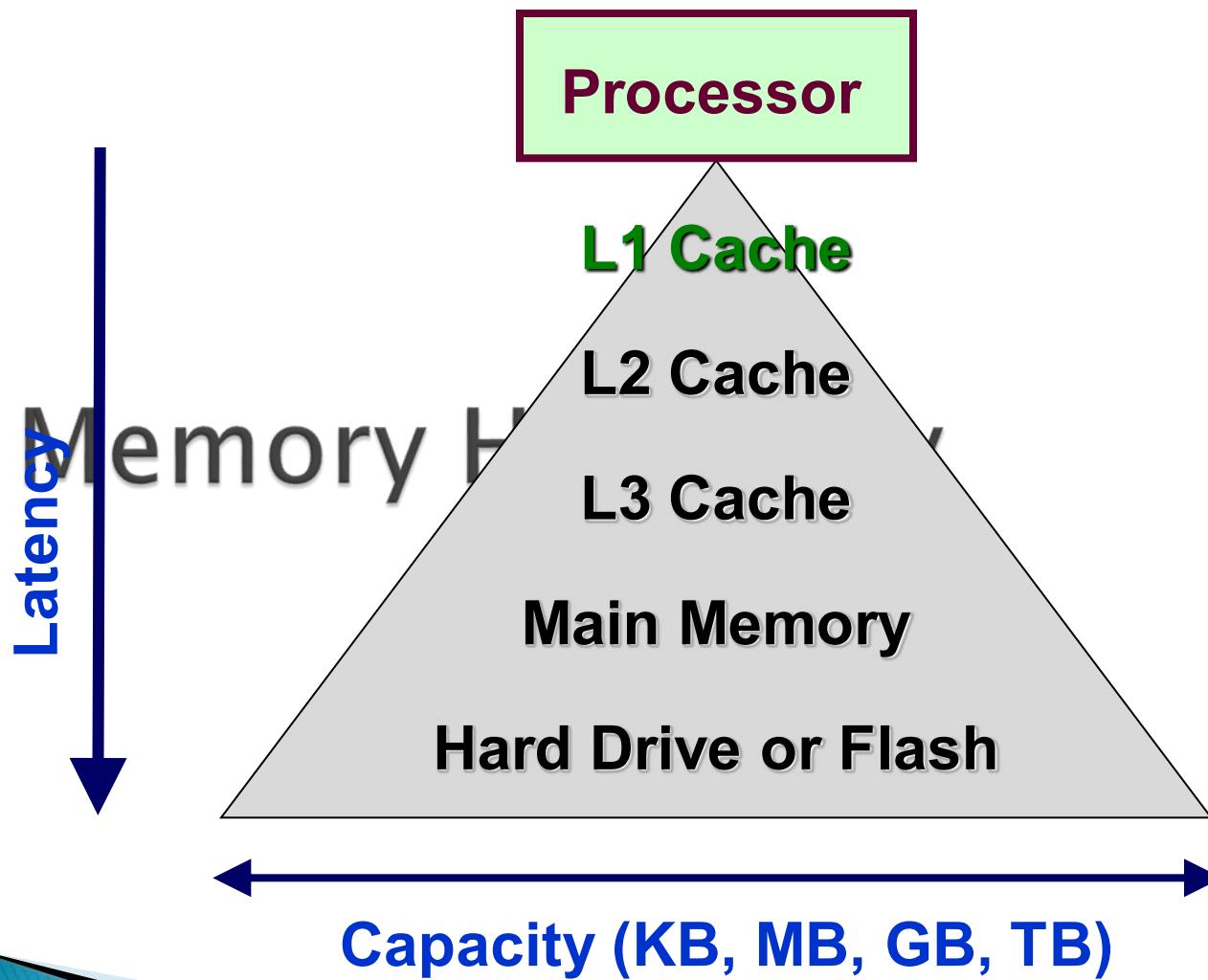
MEMORY ORGANIZATION

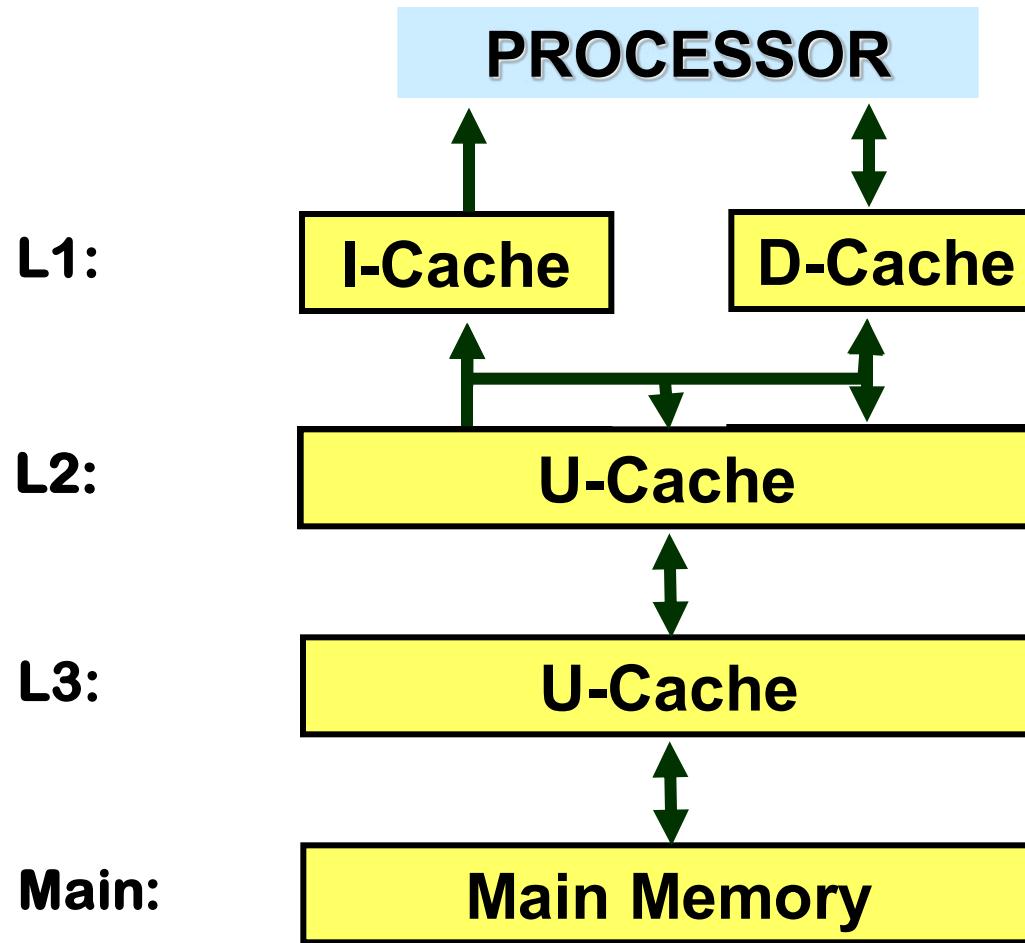
LIJO V. P

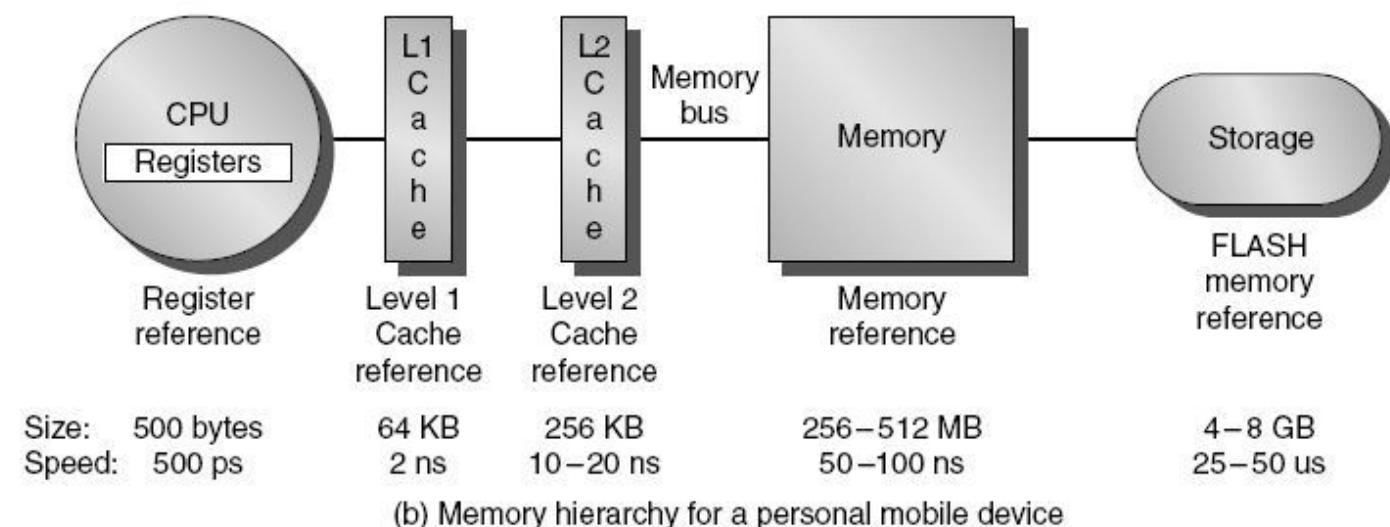
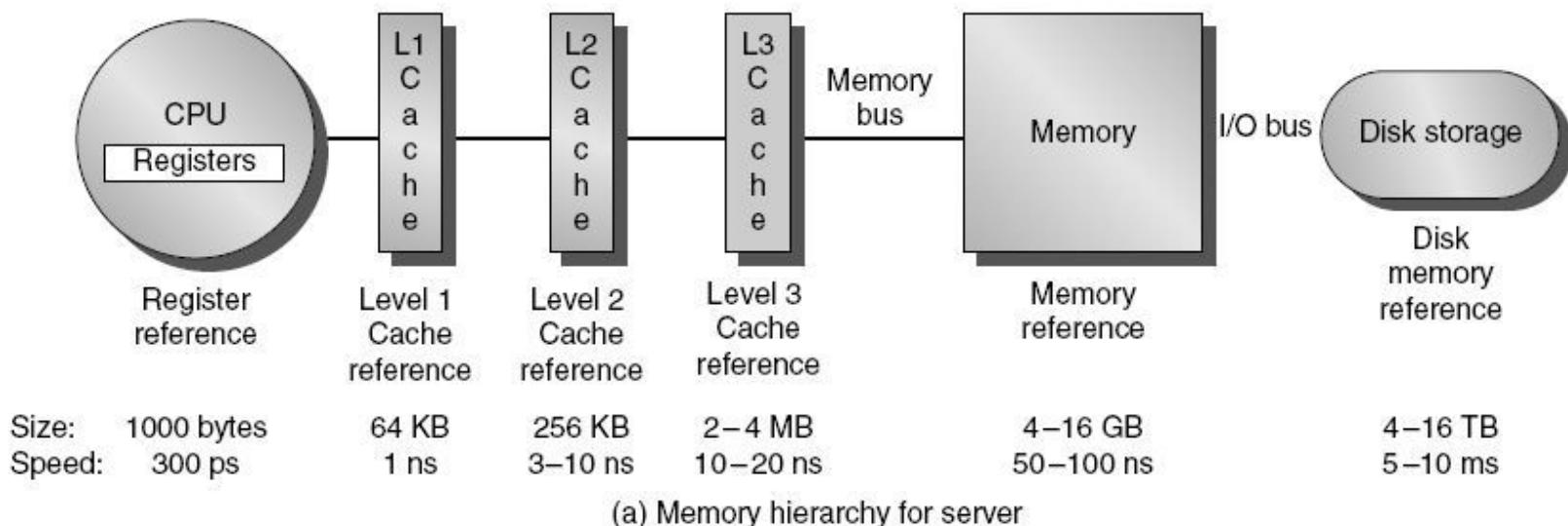
Introduction

- ▶ Programmers want **very large memory** with **low latency**
- ▶ **Fast memory** technology is more **expensive** per bit than slower memory
- ▶ Solution: organize memory system into a hierarchy
 - Entire addressable memory space available in largest, slowest memory
 - Incrementally smaller and faster memories, **each containing a subset of the memory below it**, proceed in steps up toward the processor
- ▶ Temporal and spatial locality insures that nearly all references can be found in smaller memories
 - Gives the allusion of a large, fast memory being presented to the processor

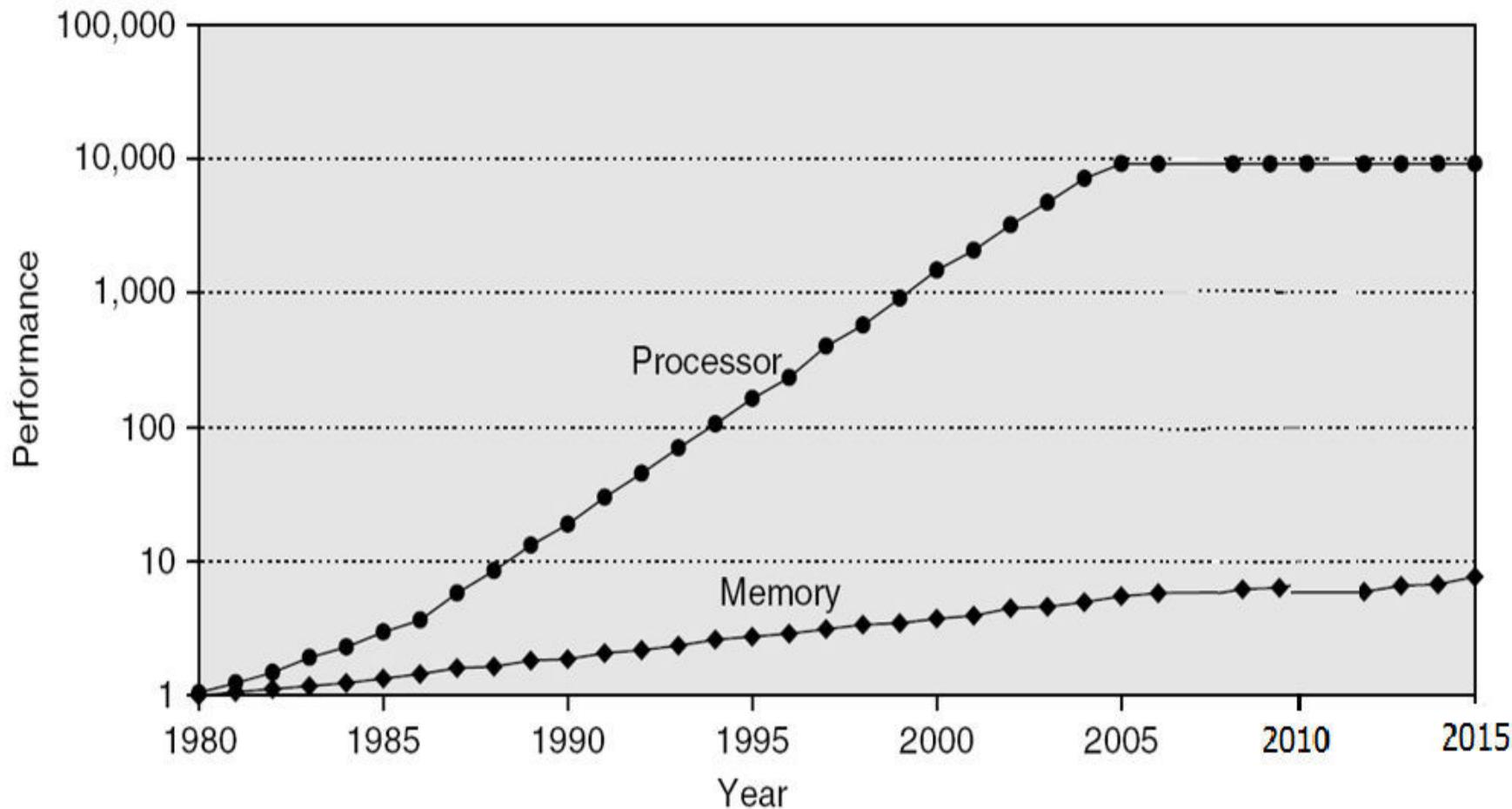
Memory Hierarchy







Memory Performance Gap



Memory organization

Byte Ordering

Ordering of bytes within a multi-byte data item

Types:

- Big-endian
- Little-endian

Byte Storage Methods (Byte Ordering)

Big-Endian

- Assigns MSB to least address and LSB to highest address
- Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	DE
Base Address + 1	AD
Base Address + 2	BE
Base Address + 3	EF

Byte Storage Methods contd.,

- ▶ Little Endian
 - Assigns MSB to highest address and LSB to least address
 - Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	EF
Base Address + 1	BE
Base Address + 2	AD
Base Address + 3	DE

Example

- ▶ **Example:** Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?

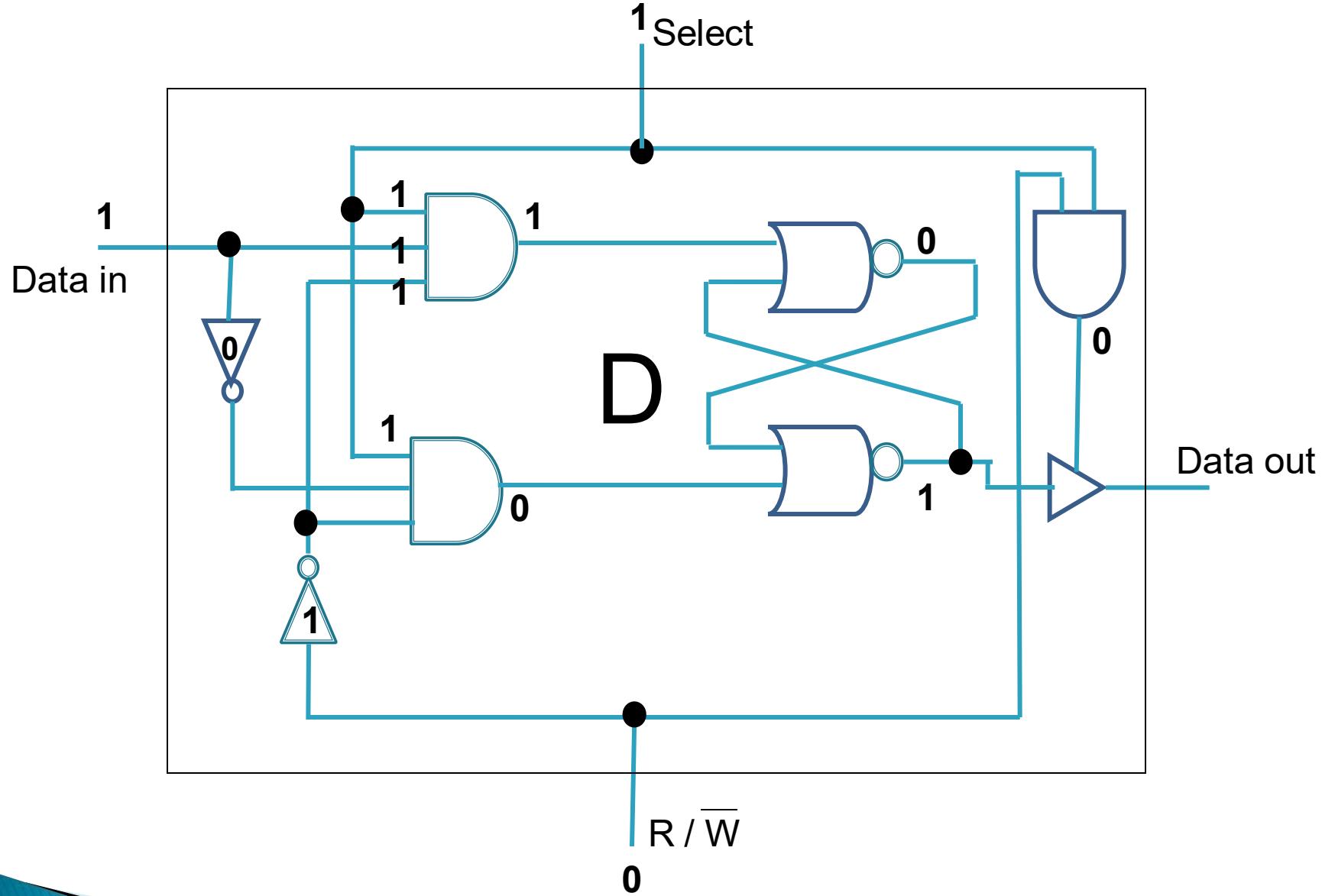
Big Endian				Little Endian					
MSB	----->			LSB	MSB	----->			LSB
24	25	26	27		27	26	25	24	
Word 24	12	2E	5F	01	Word 24	12	2E	5F	01

Some points about endian-ness

- Computer systems, in use today are split between those that are big-endian, and those that are little-endian.
- This leads to problems when a big-endian computer wants to transfer data to a little-endian computer.
- Some architectures, for example the PowerPC and ARM, allow the **endian-ness** of the architecture to be changed programmatically.

Conceptual memory cell – static RAM cell

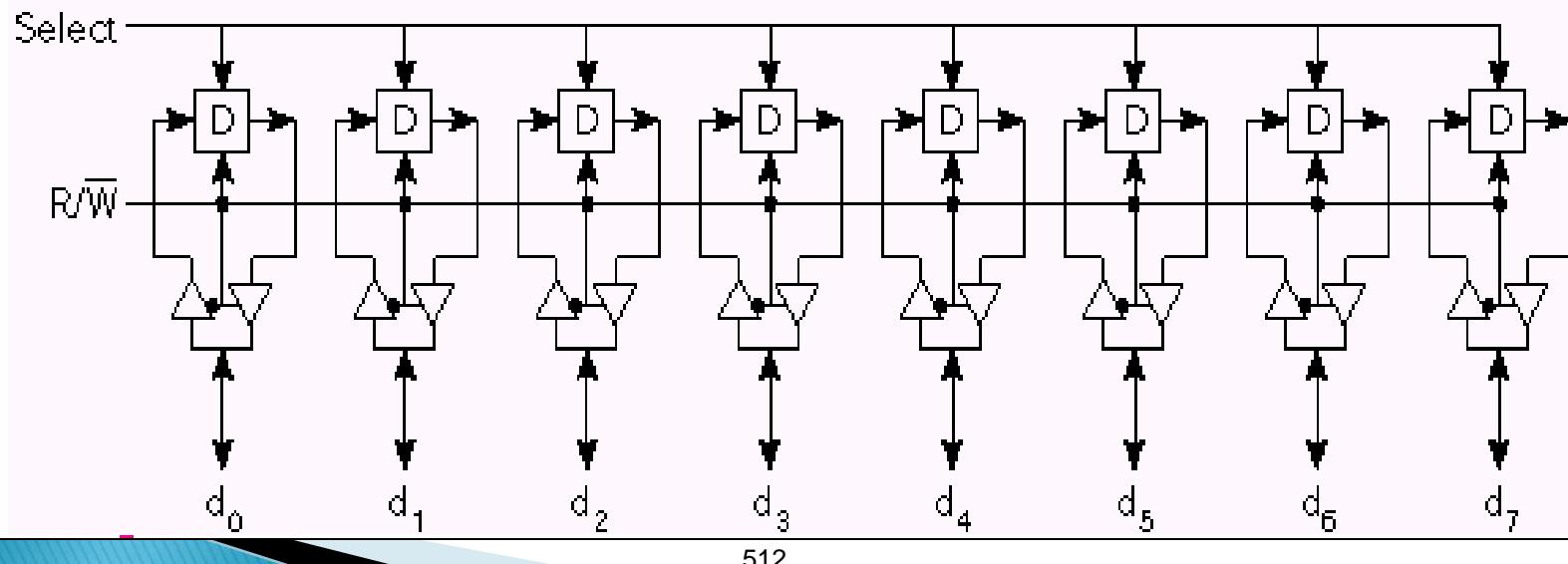
19-Memory Organization-01-Sep-2020Material_I_01-Sep-2020_Memoryorganisation



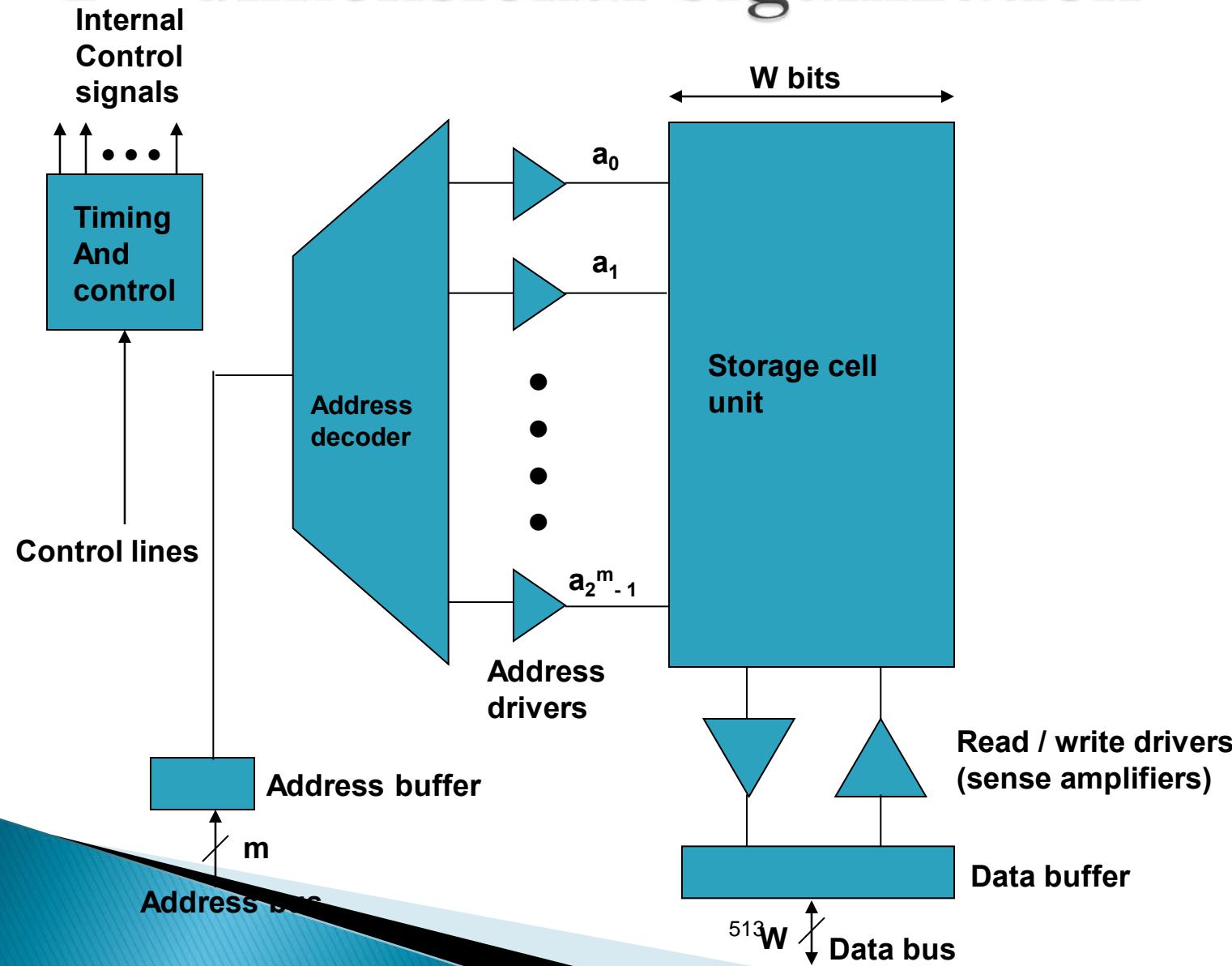
So, Tri-state buffer is in high impedance state and buffers the value

1 – dimensional organization

- ▶ Buffers would not be required, since the output of each cell is already buffered. They are shown to indicate buffering of the register from the data bus



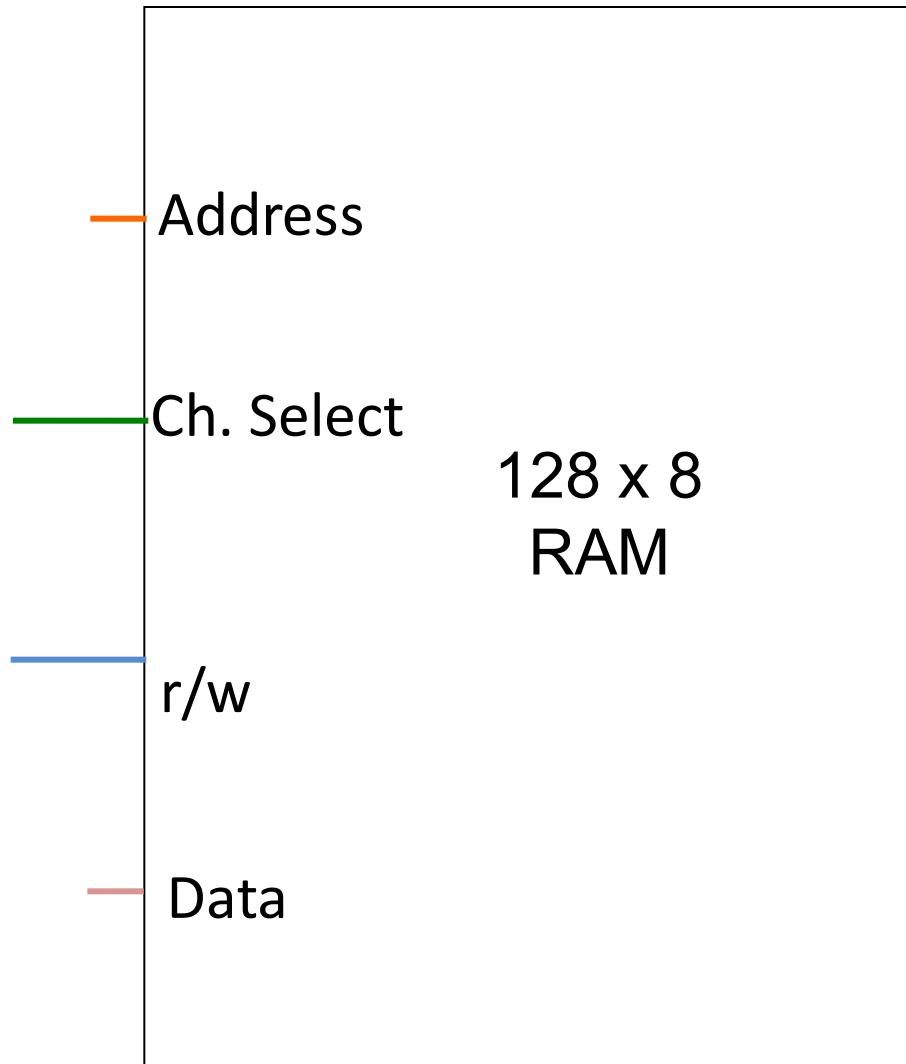
1 – dimensional organization



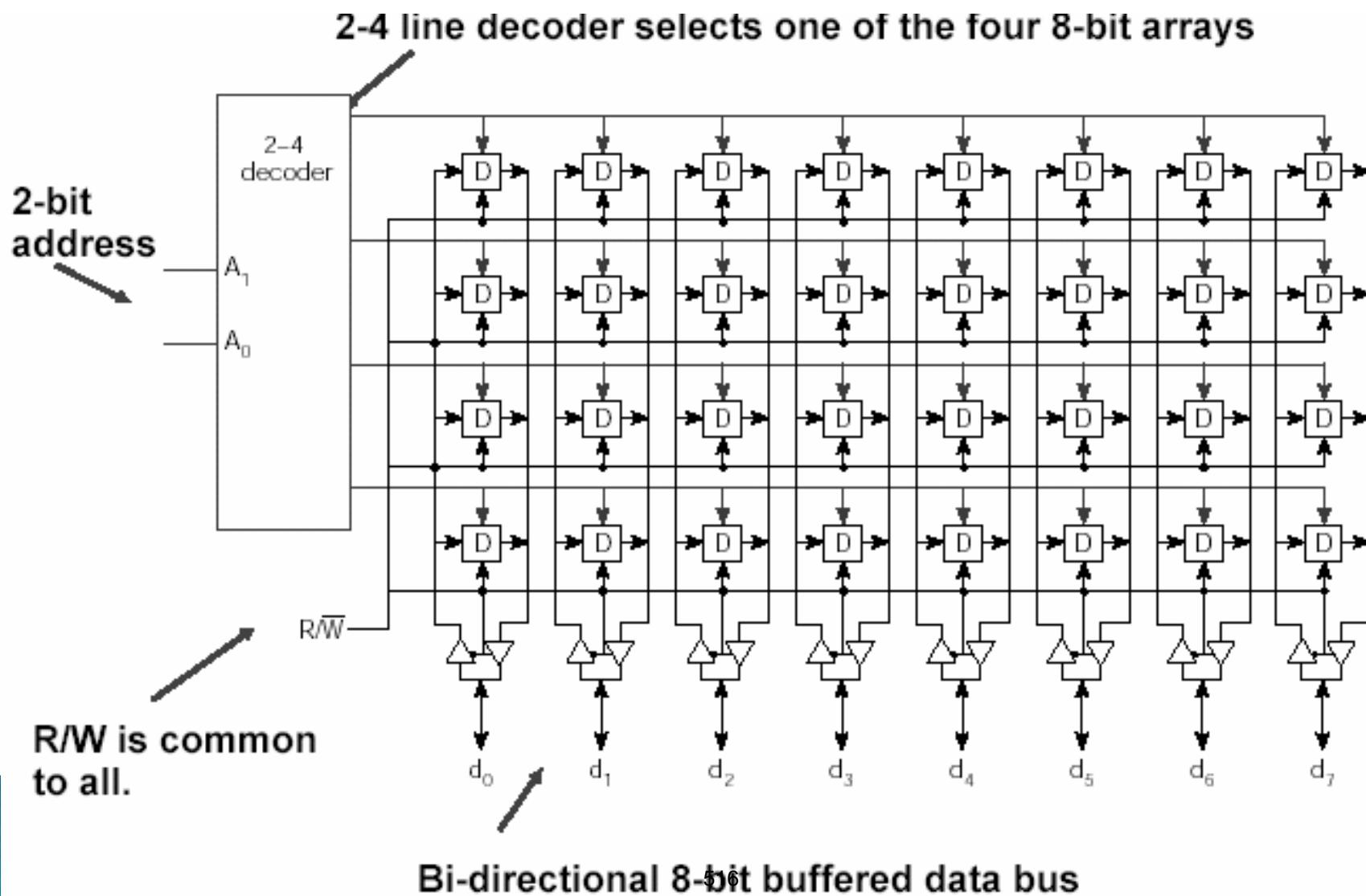


128x 8

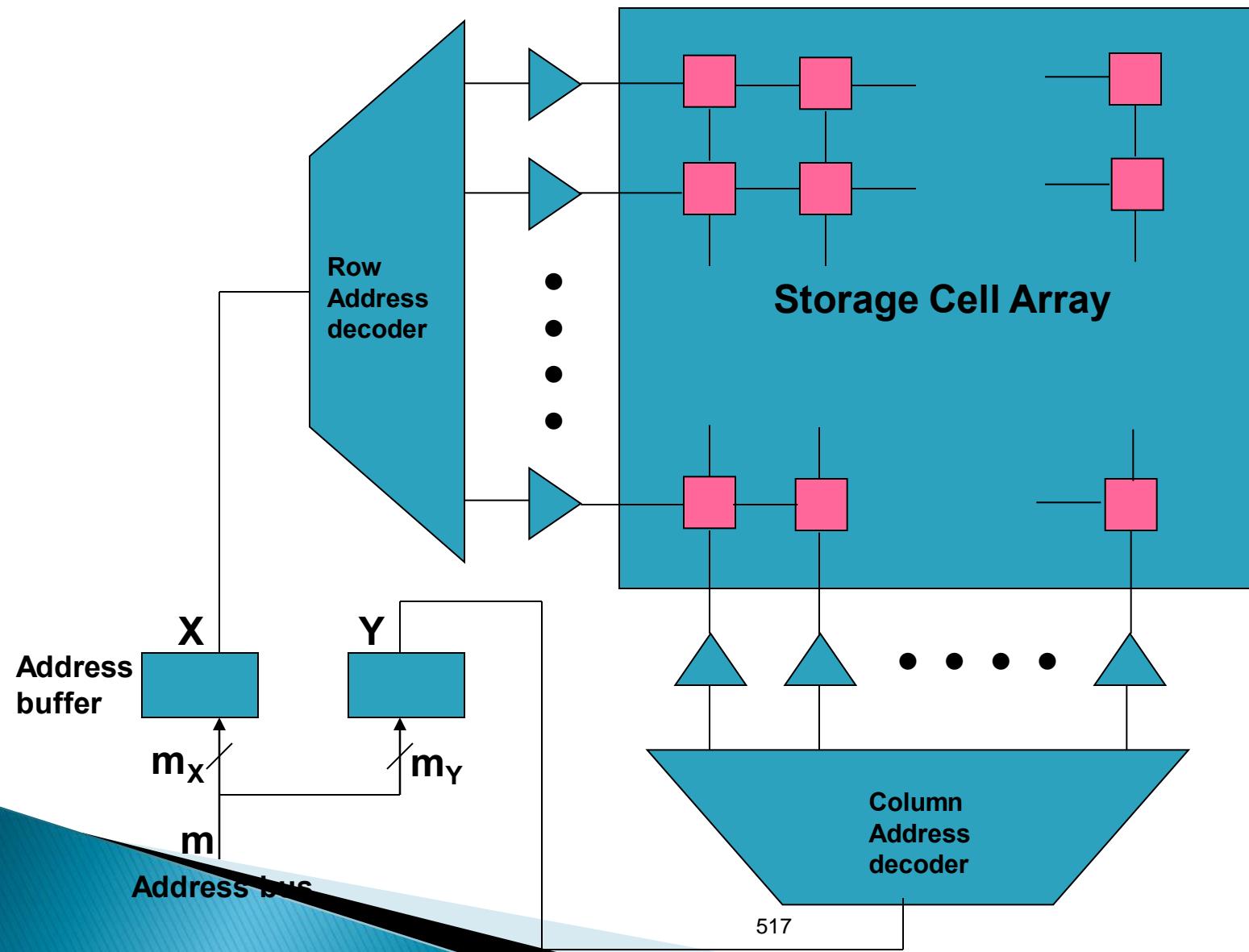
RAM



1 – dimensional organization



2 – dimensional organization



References

Text Book

- ▶ William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- ▶ J. P. Hayes, Computer system architecture, McGraw Hill,2000

Memory Key Characteristics

Introduction to Memory

- **Charles Babbage started Difference engine in 1821 but failed its test in 1833, Why?**

Due to unavailability of Memory

- **What is Memory?**
- **A single separate storage structure that holds information in the form of bits called as Memory**
- **The binary information may be instructions and data**
- **Stored program concept was introduced with the advent of vacuum tubes by John Von Neumann 1940**



Memory Capacity

- Number of bytes that can be stored

Term	Normal Usage	Usage as Power of 2
K (Kilo)	10^3	$2^{10} = 1,024$
M (Mega)	10^6	$2^{20} = 1,048,576$
G (Giga)	10^9	$2^{30} = 1,073,741,824$
T (Tera)	10^{12}	$2^{40} = 1,099,511,627,776$

Key Characteristics



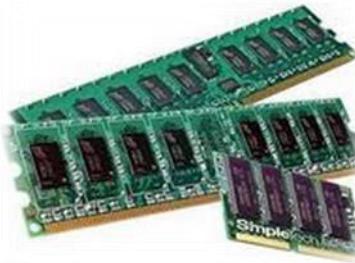
- Location
 - CPU
 - Internal (main)
 - External (secondary)
- Capacity
 - Word size
 - Number of words
- Unit of transfer
 - Word
 - Block
- Access methods
 - Sequential access
 - Direct access
 - Random access
 - Associative access
- Performance
 - Access time
 - Cycle time
 - Transfer rate

Key Characteristics contd.,

- Physical Type
 - Semiconductor
 - Magnetic surface
 - Optical
- Physical Characteristics
 - Volatile / Non-Volatile
 - Erasable / Non-erasable
- Organization

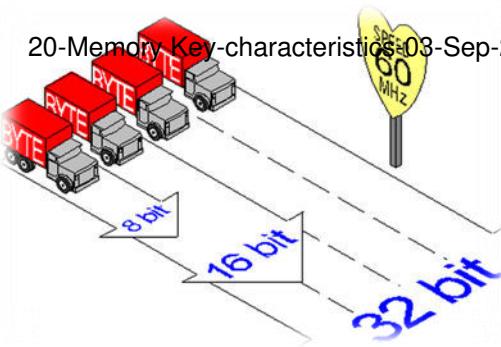
Location

- Three locations of memories
 - CPU
 - Registers – used by CPU as its local memory
 - Internal memory
 - Main memory
 - Cache memory
 - External memory
 - Peripheral devices – disk, tape – accessible to CPU via I/O controllers



Capacity

- Internal memory capacity is expressed in terms of bytes or words.
- External memory capacity is expressed in terms of blocks (depends on words in memory)
- Total memory = number of words × word length
- Number of words = $2^{\text{address bus width}}$
- Word length = Data bus width



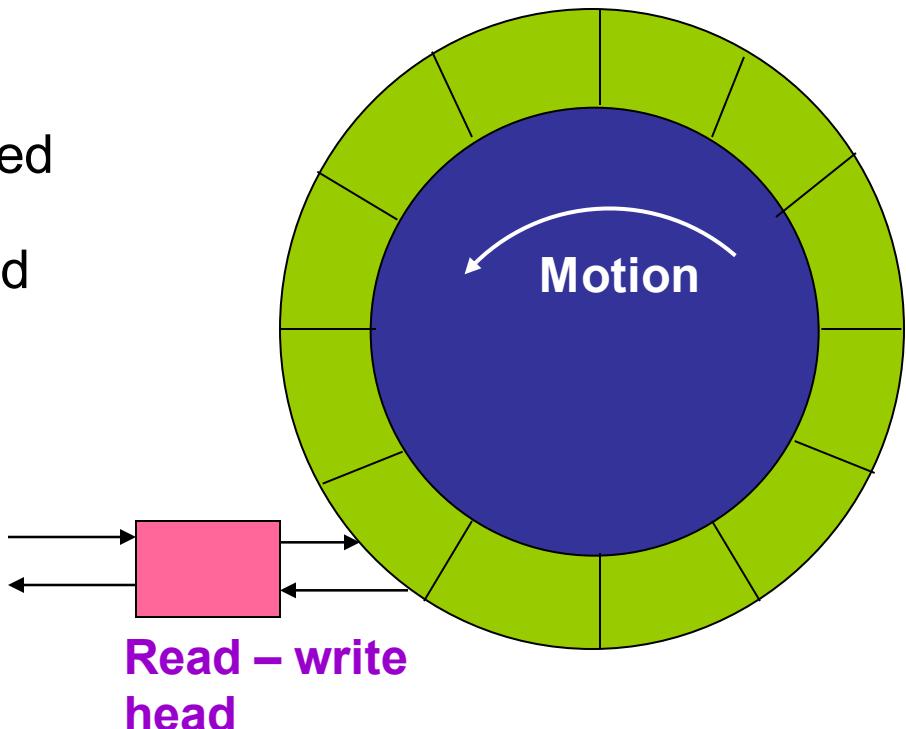
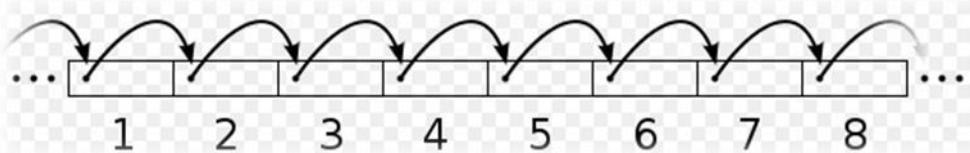
Unit of transfer

- Internal memory – number of data lines into and out of the main memory module
- External memory – blocks – longer units than a word

Access Methods

- Four types
 - Sequential Access

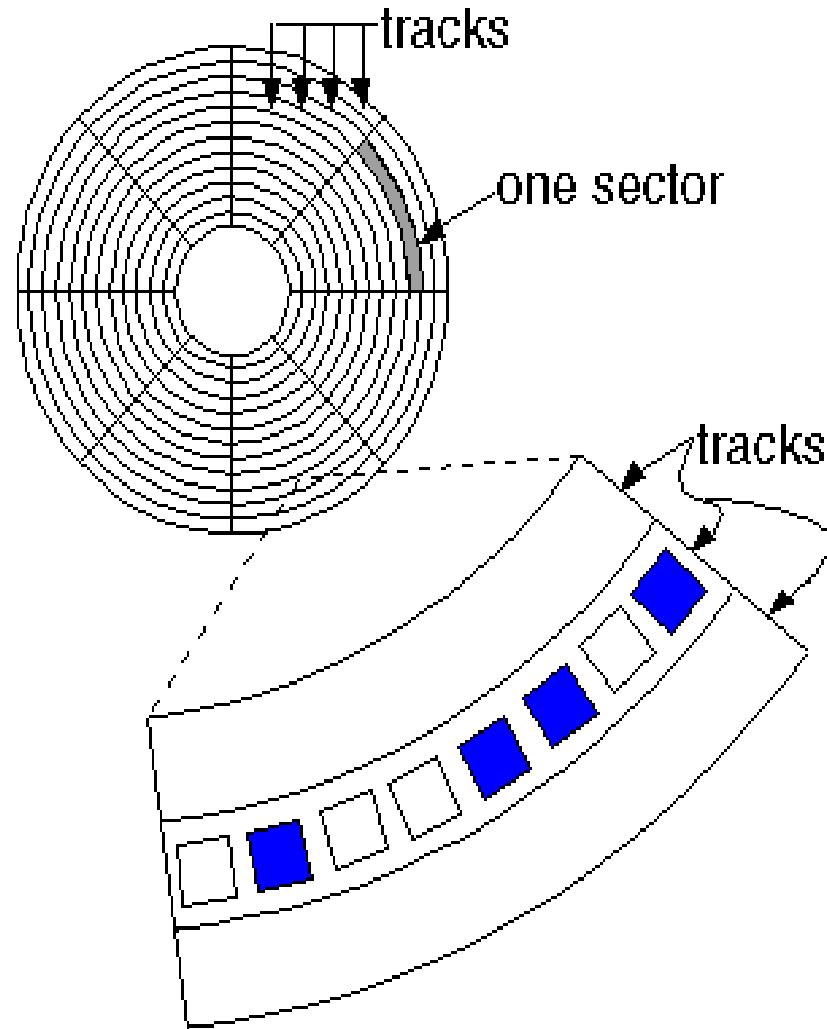
- Accesses the memory in predetermined sequence
- Shared read/write head is used, and this must be moved its current location to the desired location, passing and rejecting each intermediate record.
- So, the time to access an arbitrary record is highly variable
- Slower than random access memory
- Ex: Magnetic Tapes, data in memory array



Access Methods contd.,

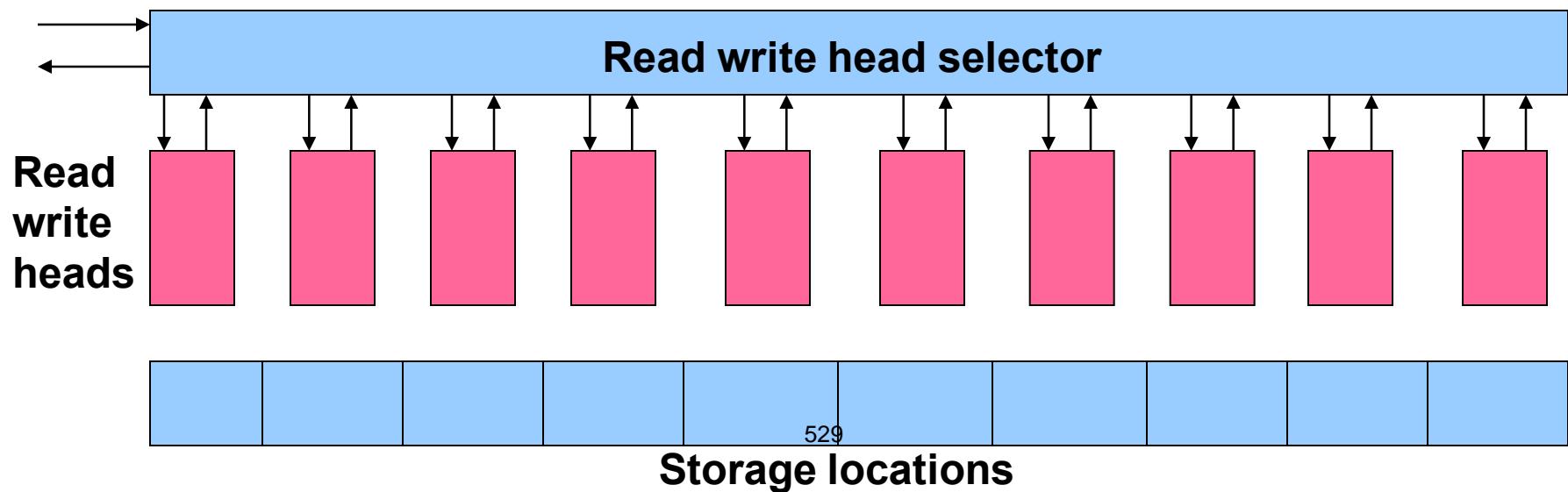
From Computer Desktop Encyclopedia
© 1998 The Computer Language Co., Inc.

- Direct access
 - Also referred as semi random access memory
 - Access time is variable
 - The track is accessed randomly but access within each track is serial
 - Access is accomplished by general access to reach a general vicinity plus sequential searching, counting, waiting to reach the final location.
 - Ex: Magnetic Disk



Access methods contd.,

- Random Access
 - Each addressable location in memory has unique, physically wired – in addressing mechanism
 - Time to access a location is independent of the sequences of prior access and is constant
 - Main memory systems are a random access
 - Storage locations can be accessed in any order
 - Semiconductor memories



Access Methods contd.,



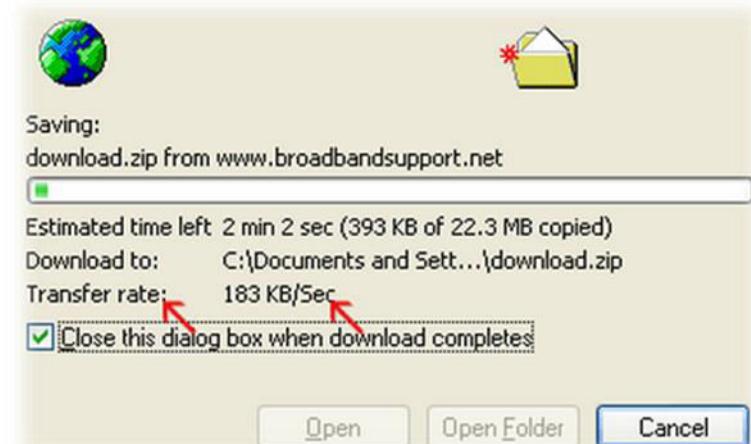
- Associate Access
 - Word is retrieved based on portion of its contents rather than its address
 - This enables one to make a comparison of desired bit locations within a word for specific match
 - Has own addressing mechanism
 - Retrieval time is constant
 - Access time is independent of location or prior access patterns
 - Cache memories

Performance

- Access time
 - The time required to read / write the data from / into desired record
 - Depends on the amount of data to be read / write
 - If the amount data is uniform for all records then the access time is same for all records.
 - Time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use.
- Memory Cycle time
 - Access time + time required before a second access can commence
 - For Random access method ,this memory cycle time is same for all records
 - The sequential access and direct access ,the memory cycle time is different

Performance contd.,

- Transfer rate / Throughput
 - Rate at which the data can be transferred into or out of a memory unit
 - Random access memory
 - 1/cycle time
 - Non-Random access memory
 - $T_n = T_a + (N/R)$, where
 - T_n – average time to read or write N bits
 - T_a – average access time
 - N – Number of bits
 - R – Transfer rate,
in bits per second (BPS)



Physical type

Semiconductor

Semiconductor memory uses semiconductor-based integrated circuits to store information.



Magnetic surface



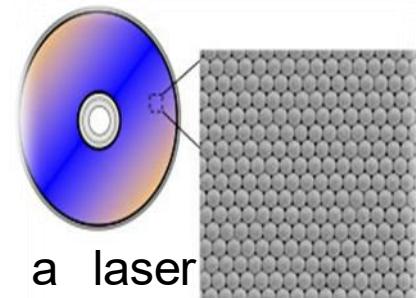
Magnetic storage uses different patterns of magnetization on a magnetically coated surface to store information.

Example:

Magnetic disk, Floppy disk, Hard disk drive

Optical

The typical optical disc, stores information in deformities on the surface of a circular disc and reads this information by illuminating the surface with a laser diode and observing the reflection.



Physical characteristics

- Volatile memory
 - Information decays naturally or lost when electrical power is switched off
- Non-volatile memory
 - Once recorded is retained until deliberately changed
 - No electrical power is needed to retain information
 - Magnetic surface memories
- Semiconductor memories may be either volatile or non-volatile
 - A type of non-volatile semiconductor memory known as flash memory
 - A type of volatile semiconductor memory is random access memory
- Non-erasable memory
 - Cannot be altered, except by destroying the storage unit (ROM)
 - A practical non-erasable memory must also be non-volatile
 - Ex: CD-R, Flash Memories
- Erasable memory
 - Erase the stored information by writing new information
 - Ex: Magnetic storage is erasable

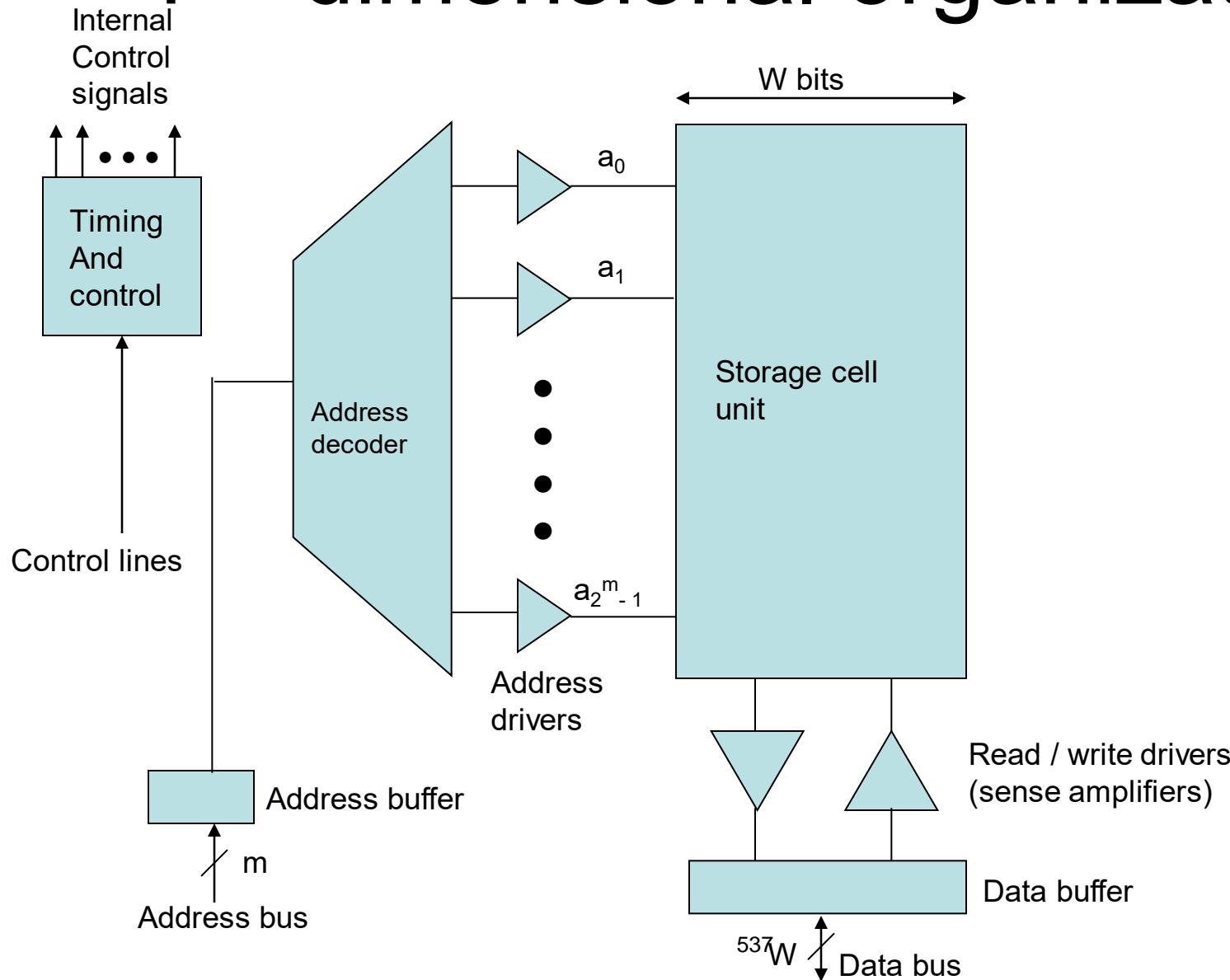
Organization

- Physical arrangement of bits to form words
- 2 types
 - 1 dimensional
 - 2 dimensional

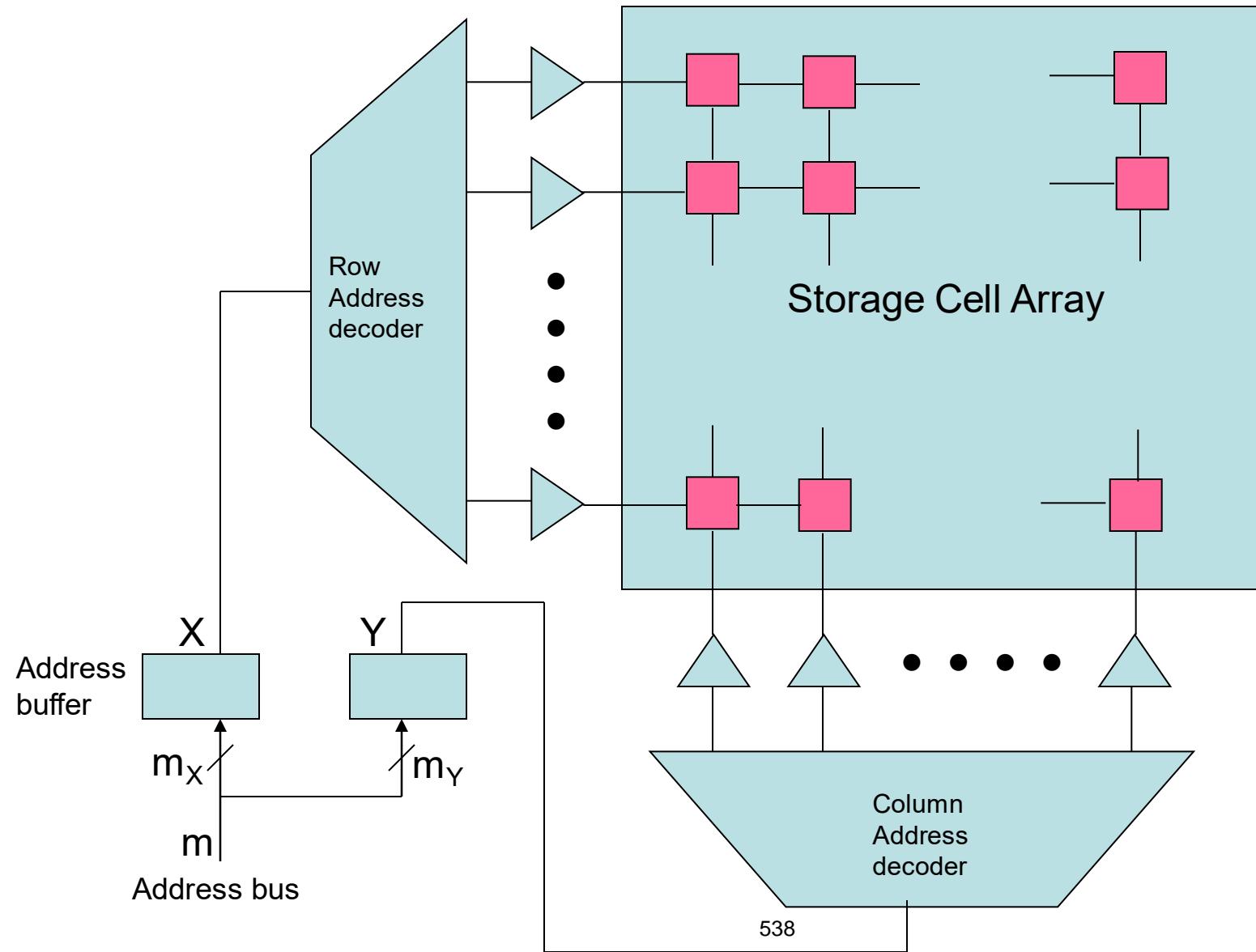
Memory Organization

- Basic element = memory cell
- Properties of Memory cell:
 - They exhibit two stable states, which can be used to represent binary 1 and 0.
 - They are capable of being written into (atleast once) to set the state.
 - They are capable of being read to sense the state.

1 – dimensional organization



2 – dimensional organization



Byte Storage Methods

- Big-Endian
 - Assigns MSB to least address and LSB to highest address
 - Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	DE
Base Address + 1	AD
Base Address + 2	BE
Base Address + 3	EF

Byte Storage Methods contd.,

- Little Endian
 - Assigns MSB to highest address and LSB to least address
 - Ex: 0 × DEADBEEF

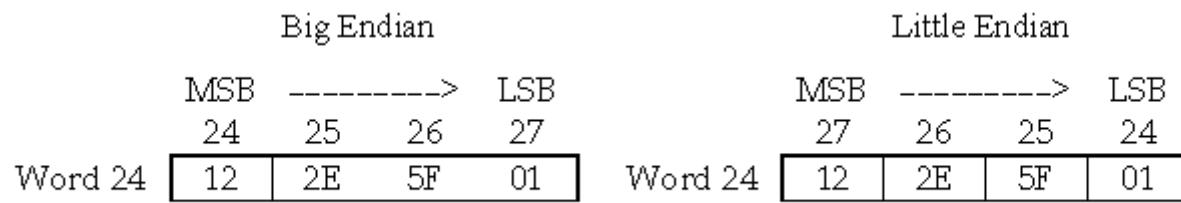
Memory Location	Value
Base Address + 0	EF
Base Address + 1	BE
Base Address + 2	AD
Base Address + 3	DE

Byte Storage Methods contd.,

- LittleEndian
 - Intel x 86 family
 - Digital equipment corporation architectures (PDP – 11, VAX, Alpha)
- BigEndian
 - Sun SPARC
 - IBM 360 / 370
 - Motorola 68000
 - Motorola 88000
- Bi-Endian
 - Power PC
 - MIPS
 - Intel's 64 IA - 64

Example

- **Example:** Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?



References

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

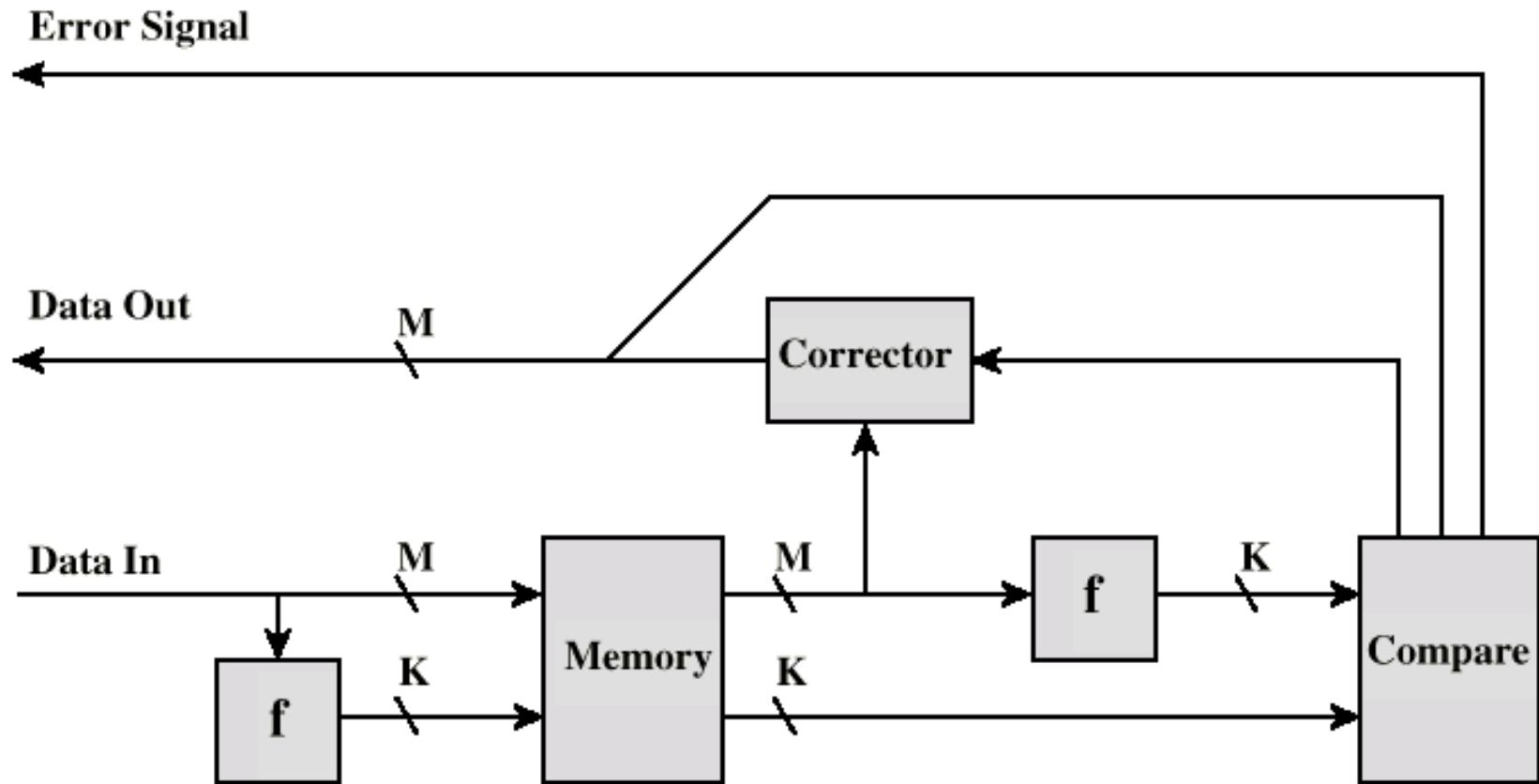
ERROR DETECTION AND CORRECTION

LIJO V P
SCOPE
VIT, VELLORE

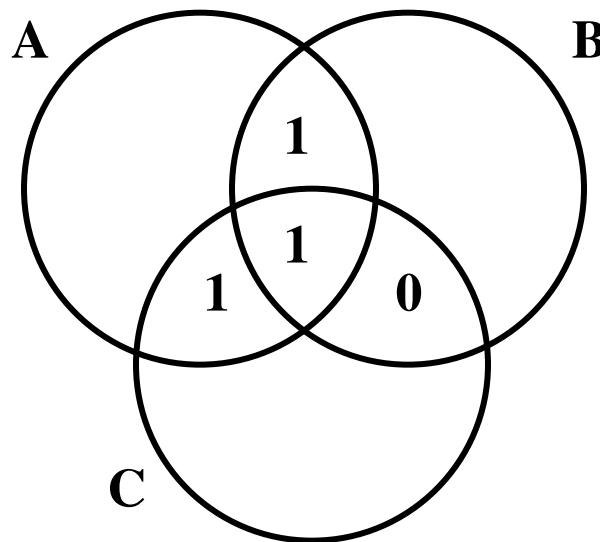
Error Correction

- A semiconductor memory system is subject to errors.
- Hard failures – permanent physical defects
Environmental abuse, manufacturing defects, etc.
- Soft error
Power supply problems etc.
- Need logic for detecting and correcting errors.
- Basic technique
 - Prior to storing data a code is generated from the bits in the word.
 - Code is stored along with the word in memory.
 - Code used to identify and correct errors.
 - When the word is fetched a new code is generated and compared to the stored code.
 - No error (normal case)
 - Correctable error is detected and corrected.
 - Non-fixable error is detected and reported.

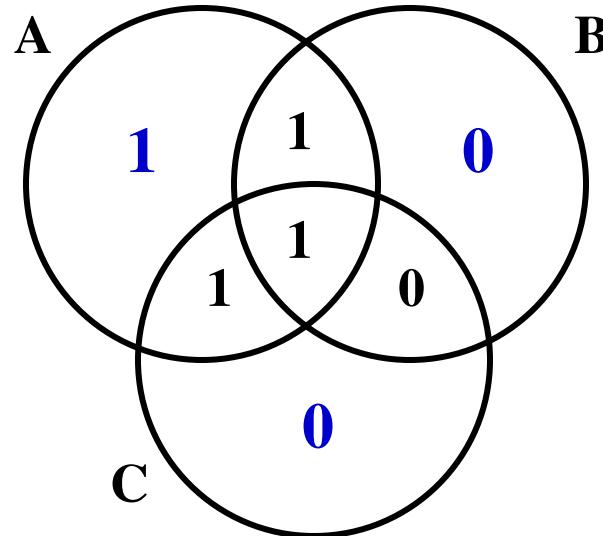
Error Correcting Code Function



Hamming Code



Assign data bits to the inner compartments.

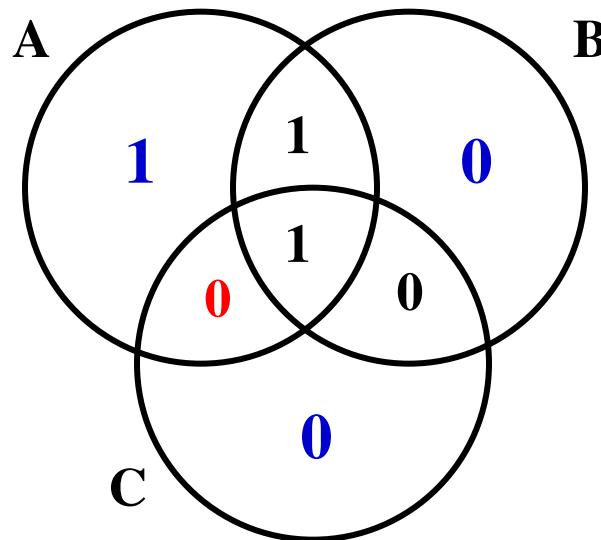
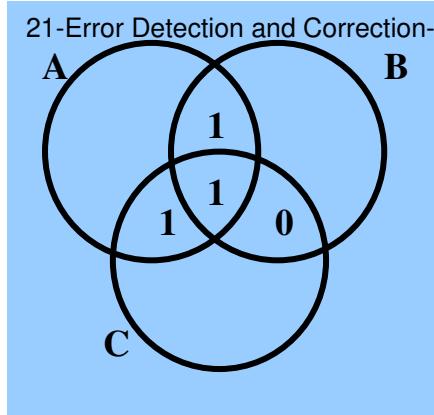


Fill the remaining compartments with parity bits.

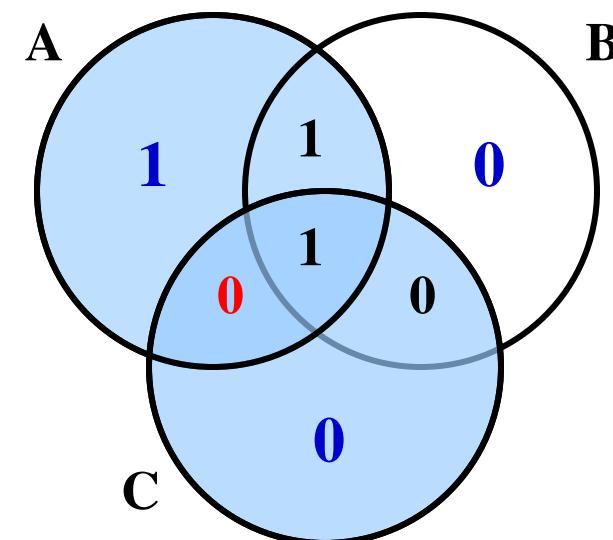
The total number of bits in a circle must be even.

For example: The data bits in A = $1+1+1 = 3$. This is odd – therefore add an additional 1.
548

Hamming Code



If a bit gets erroneously changed, the parity bits in that circle will no longer add up to 1.



Errors are found in A and C – and the shared bit in A and C is in error and can be fixed.

Single Bit Errors in 8-bit words

- 8 data bits
- $2^K - 1 \geq M + K$ is used to find the value of K (number of check bits).
- No errors \Rightarrow code = 0.
- More than one bit set to ‘1’ \Rightarrow the numerical value of the syndrome indicates the position of the data bit in error.
- $2^K - 1 \geq 8 + K \Rightarrow K = 4$

Single Bit Errors in 8-bit words

- Data and check bits arranged into a 12-bit word.
- Bit positions numbered from 1 to 12.
- Bit positions representing position numbers that are powers of 2 are designated as check bits.
- Check bits calculated as follows:

$C_1 = D_1 \oplus D_2 \oplus$	$D_4 \oplus D_5 \oplus$	D_7
$C_2 = D_1 \oplus$	$D_3 \oplus D_4 \oplus$	$D_6 \oplus D_7$
$C_4 =$	$D_2 \oplus D_3 \oplus D_4 \oplus$	D_8
$C_8 =$	$D_5 \oplus D_6 \oplus D_7 \oplus$	D_8

- Data and check bits arranged into a 12 bit syndrome word:



Calculating check bits

Bit Position	Binary	Type	
1	0001	C1	\oplus All D's with a 1 in bit 1
2	0010	C2	\oplus All D's with a 1 in bit 2
3	0011	D1	
4	0100	C4	\oplus All D's with a 1 in bit 3
5	0101	D2	
6	0110	D3	
7	0111	D4	
8	1000	C8	\oplus All D's with a 1 in bit 4
9	1001	D5	
10	1010	D6	
11	1011	D7	
12	1100	D8	

$$C_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

Each check bit works on every data bit who shares the same bit position

Example

- Input word: 00111001
Data bit D1 in rightmost position

- Calculate check bits:

$$C_1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_3 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

$$C_4 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Stored word = 001101001111

- If data bit 3 sustains an error (001101101111)

$$C_1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_3 = 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_4 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

- Calculate syndrome word:

0110 = bit position 6.

- D3 resides in bit position 6.

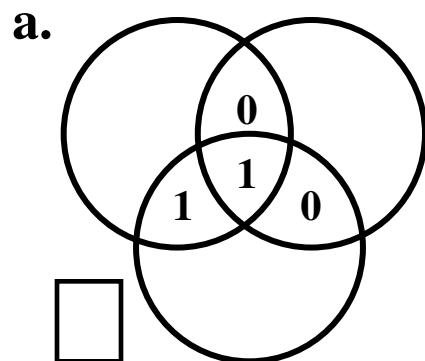
C C C C

8 4 2 1

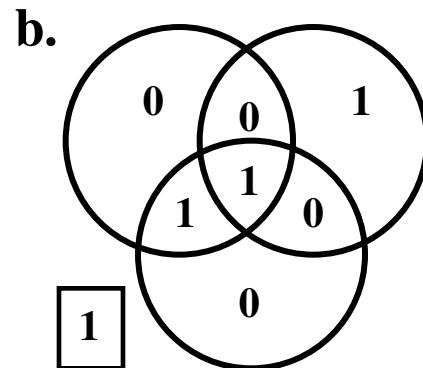
0 1 1 1

$$\begin{array}{r}
 \oplus \\
 0 \quad 0 \quad 0 \quad 1 \\
 \hline
 0 \quad 1 \quad 1 \quad 0
 \end{array}$$

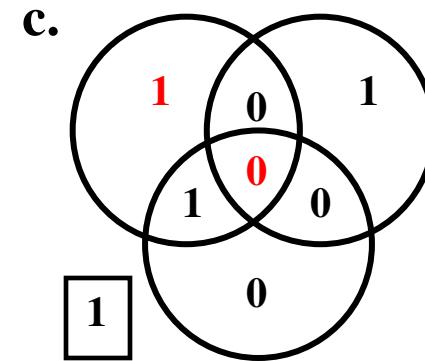
- Previous example is Single-Error-Correcting code.
- Semiconductor memory is usually equipped with SEC-DED (Single-error-correcting, double-error-detecting code). SEC-DED requires an extra bit.



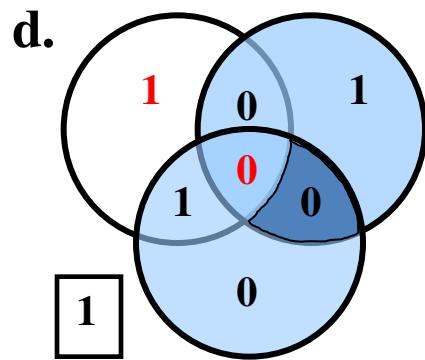
Fill in data bits.



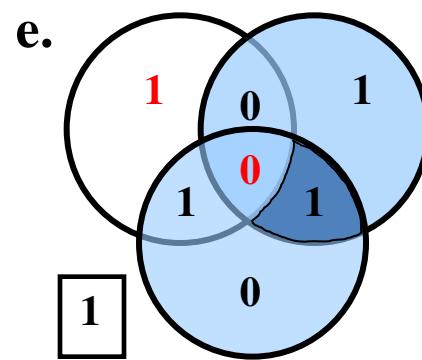
Calculate check bits.



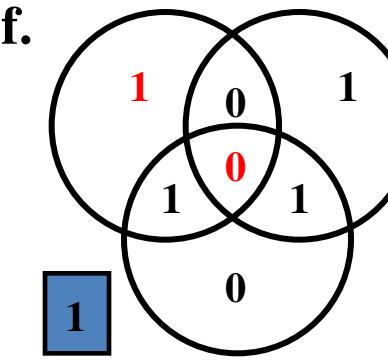
Two errors are introduced



SEC identifies the wrong bit



The extra bit checks for even parity.
554



The double error is detected!

Problems

- Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.
- For the 8-bit word 00111001, the check bits stored with it would be 0111. Suppose when the word is read from memory, the check bits are calculated to be 1101. What is the data word that was read from memory?
- How many check bits are needed if the hamming error correction code is used to detect single bit errors in 1024 bit data word
- Generate the code for the data word 0101000000111001. show that the code will correctly identify an error in data bit 5.

References

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- Internet: PPT slides, Author: Jane and Harold Huang

Types of Memories

Memory

Main memory consists of a number of storage locations, each of which is identified by a **unique address**

The ability of the CPU to identify each location is known as its **addressability**

Each location stores a **word** i.e. the number of bits that can be processed by the CPU in a single operation.

Word length may be typically 16, 24, 32 or as many as 64 bits.

A large word length **improves system performance**, though may be less efficient on occasions when the full word length is not used



Types of Main Memories

- Read Only Memory (**ROM**)
- Random Access Memory (**RAM**)

RAM

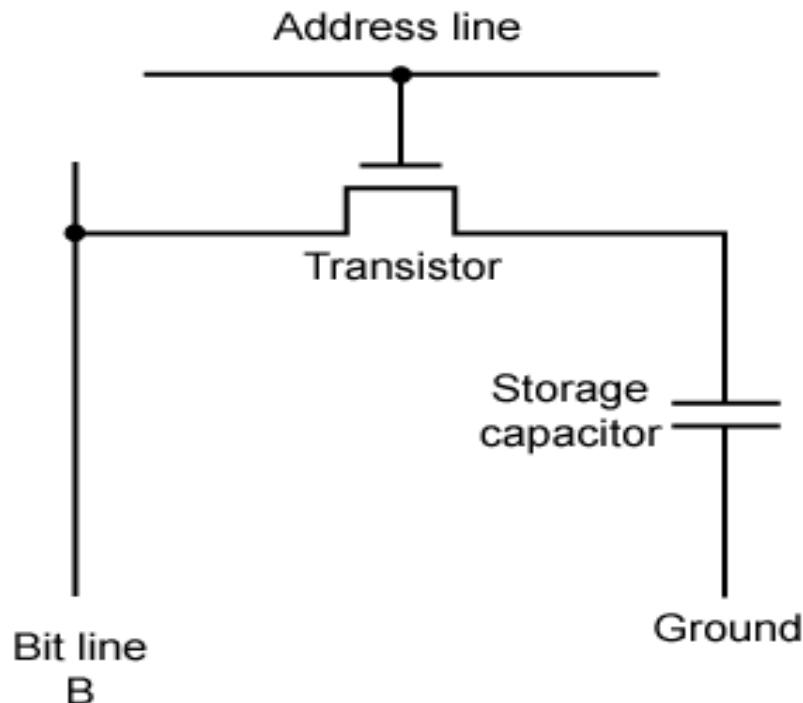
- Reading and writing is possible
- Both are accomplished using electrical signals.
- Volatile – so, used only as temporary storage.
- Two technologies
 - Static RAM (SRAM)
 - Dynamic RAM (DRAM)

Dynamic RAM

- Stores data as charge on capacitors
- If capacitor is charged
 - Data is 1
- Else
 - Data is 0.
- Needs refreshing cycle as capacitors have a tendency of discharging.
- The term *dynamic* refers to this tendency of the stored charge to leak away, even with power continuously applied.
- Volatile
- When read, data is lost. So, restoring need to be done.

Dynamic RAM

- DRAM cell
 - Consists of a transistor and a capacitor.
 - Transistor acts a switch
 - If transistor is closed
 - Allows current to flow
 - Else
 - No current flows



Dynamic RAM

- Write
 - Voltage signal is applied to the bit line.
 - High voltage – 1
 - Low voltage – 0
 - Address line is activated allowing the charge to be transferred to the capacitor
- Read
 - Address line is activated
 - Charge on capacitor is fed out onto a bit line and to a sense amplifier.
 - Sense amplifier compares with reference value and determines if the cell contains 0 or 1.
 - The value is restored
- Used for large memory requirements

Static RAM

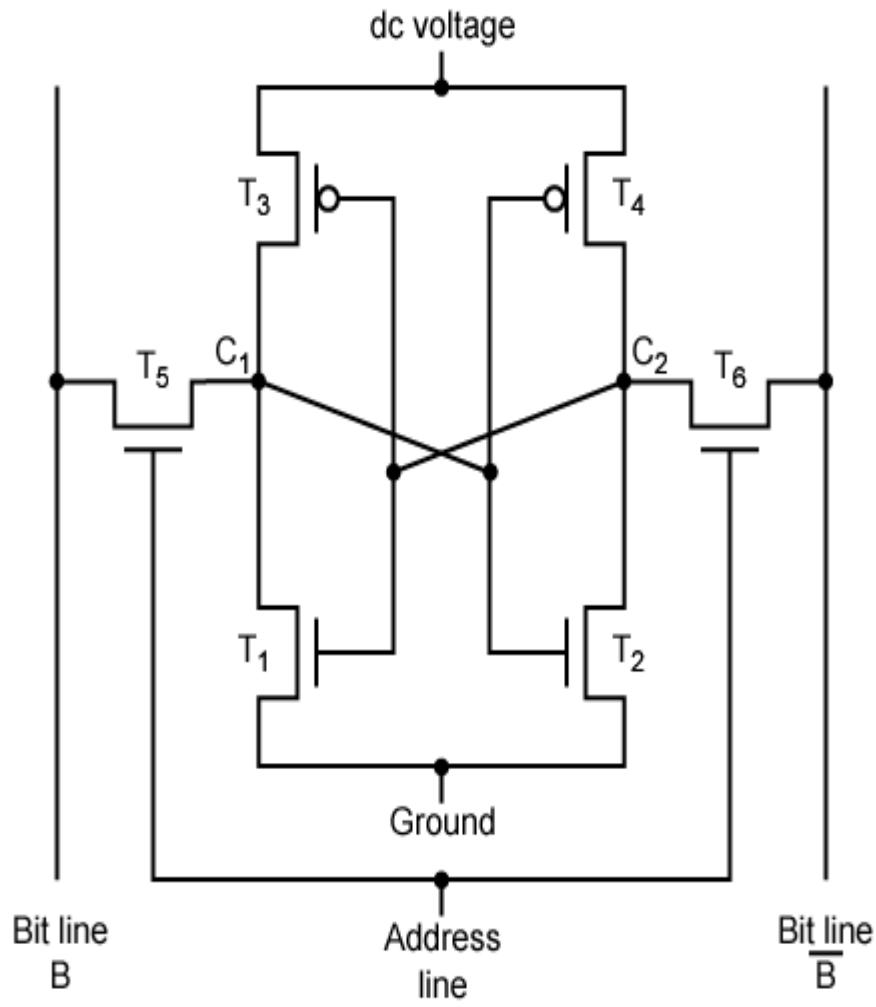
- Holds the data as long as the power is supplied
- Read operation don't destroy the original data.
- Expensive than DRAM but shorter cycle times.
- Used for faster small memories like cache memory.
- Uses 4 – 6 transistors to store a single bit of data
- Less power consumption than DRAM
- Complex construction
- Digital
 - Use flip-flops

See the Demo through the following reference

<http://tams-www.informatik.uni-hamburg.de/applets/sram/index.html>

SRAM

- Transistor arrangement gives stable logic state
- Logic State 1
 - C_1 high, C_2 low
 - $T_1 T_4$ off, $T_2 T_3$ on
- Logic State 0
 - C_2 high, C_1 low
 - $T_2 T_3$ off, $T_1 T_4$ on
- Address line controls two transistors $T_5 T_6$.
- When signal is applied to address line, T_5 and T_6 are on.
- Write – apply value to B & complement to \bar{B}
- Read – bit value is read from line B



SRAM

- Volatile
- Faster
- smaller memory units
- Complex construction
- Don't require
refreshing circuit
- Cache memory
- Larger per bit
- Digital

DRAM

- volatile
- slower
- larger memory units
- simpler to build
- require refresh
- Main memory
- smaller per bit
- analog

Types of RAM

1. Dynamic Random Access Memory (DRAM)

- Contents are constantly refreshed 1000 times per second
- Access time 60 – 70 nanoseconds

Note: a **nanosecond** is one **billionth** of a second!

2. Synchronous Dynamic Random Access Memory (SDRAM)

- Quicker than DRAM
- Access time less than 60 nanoseconds

3. Direct Rambus Dynamic Random Access Memory (DRDRAM)

- New type of RAM architecture
- Access time 20 times faster than DRAM
- More expensive

Types of RAM

4. Static Random Access Memory (SRAM)

- Doesn't need refreshing
- Retains contents as long as power applied to the chip
- Access time around 10 nanoseconds
- Used for **cache** memory
- Also for **date and time** settings as powered by small battery

5. Cache memory

- Small amount of memory typically 256 or 512 kilobytes
- Temporary store for often used instructions
- **Level 1** cache is built within the CPU (internal)
- **Level 2** cache may be on chip or nearby (external)
- Faster for CPU to access than main memory

Types of RAM

6. Video Random Access memory

- Holds data to be displayed on computer screen
- Has two data paths allowing READ and WRITE to occur at the same time
- A system's amount of VRAM relates to the number of colours and resolution
- A graphics card may have its own VRAM chip on board

7. Virtual memory

- Uses backing storage e.g. hard disk as a **temporary location** for programs and data where insufficient RAM available
- Swaps programs and data between the hard-disk and RAM as the CPU requires them for processing
- A cost-effective method of running large or many programs on a computer system
- Cost is speed: the CPU can access RAM in nanoseconds but hard-disk in milliseconds (Note: a **millisecond** is a **thousandth** of a second)
- Virtual memory is much slower than RAM

Read only memory (ROM)

- ✓ ROM holds programs and data **permanently** even when computer is switched off
- ✓ The contents of ROM are fixed at the time of manufacture
- ✓ Stores a program called the **bootstrap loader** that helps start up the computer
- ✓ Access time of between 10 and 50 nanoseconds

Types of ROM

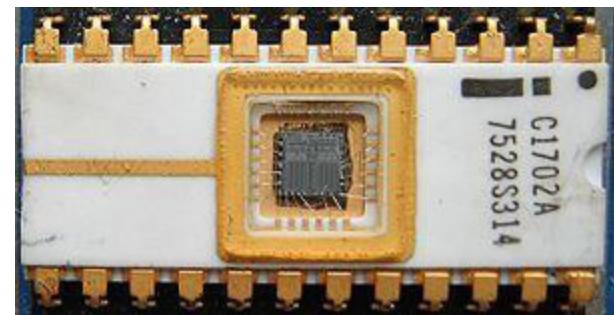
1. Programmable Read Only Memory (PROM)

- Empty of data when manufactured
- May be permanently programmed by the user

2. Erasable Programmable Read Only Memory (EPROM)

- Can be programmed, erased and reprogrammed
- The EPROM chip has a small window on top allowing it to be erased by shining ultra-violet light on it
- After reprogramming the window is covered to prevent new contents being erased
- Access time is around 45 – 90 nanoseconds

- The first EPROM, an Intel 1702, with the die and wire bonds clearly visible through the erase window.



Types of ROM

3. Electrically Erasable Programmable Read Only Memory (EEPROM)

- Reprogrammed electrically **without** using ultraviolet light
- Must be removed from the computer and placed in a special machine to do this
- Access times between 45 and 200 nanoseconds

4. Flash ROM

- Similar to EEPROM
- However, can be reprogrammed while still in the computer
- Easier to upgrade programs stored in Flash ROM
- Used to store programs in devices e.g. modems
- Access time is around 45 – 90 nanoseconds

5. ROM cartridges

- Commonly used in games machines
- Prevents software from being easily copied

Comparison of different types of main memories

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	
Programmable ROM (PROM)				
Erasable PROM (EPROM)		UV light, chip-level	Electrically	Nonvolatile
Electrically Erasable PROM (EEPROM)	Read-mostly memory	Electrically, byte-level		
Flash memory		Electrically, block-level		

References

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- <http://sites.google.com/site/mrajasekharababu/Home/academic/09m-tech-cse/mtech09> (7.8InstructorMeterial-Unit3-MainMemory.pdf)

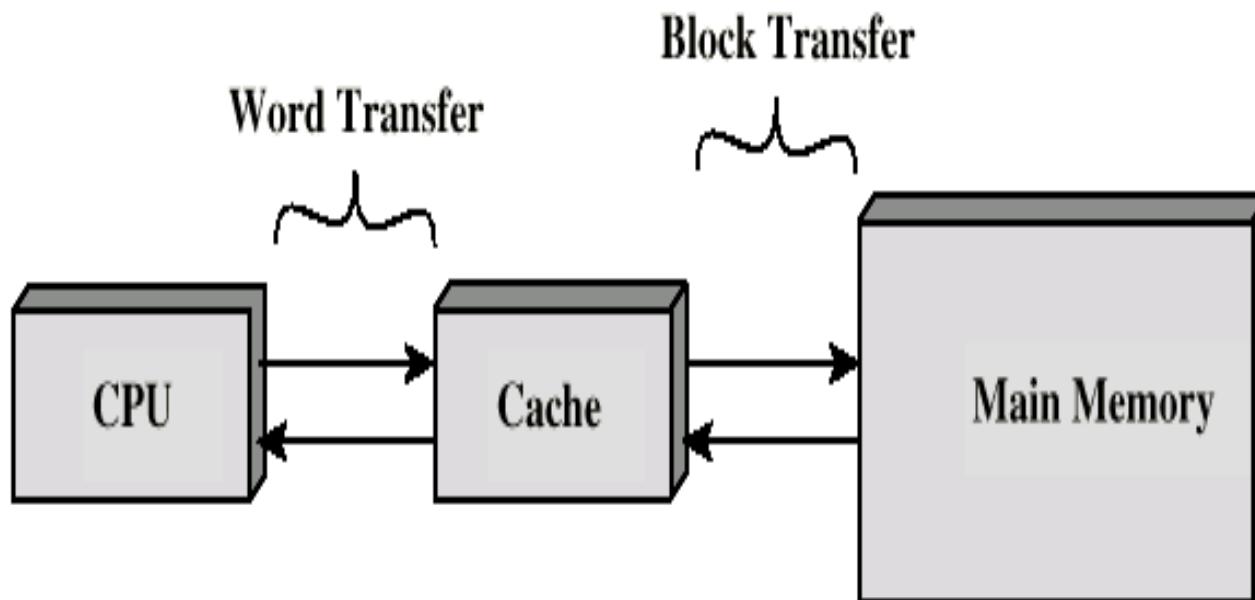
CACHE MEMORY

LIJO V. P.
SCOPE
VIT, VELLORE

Cache Memory

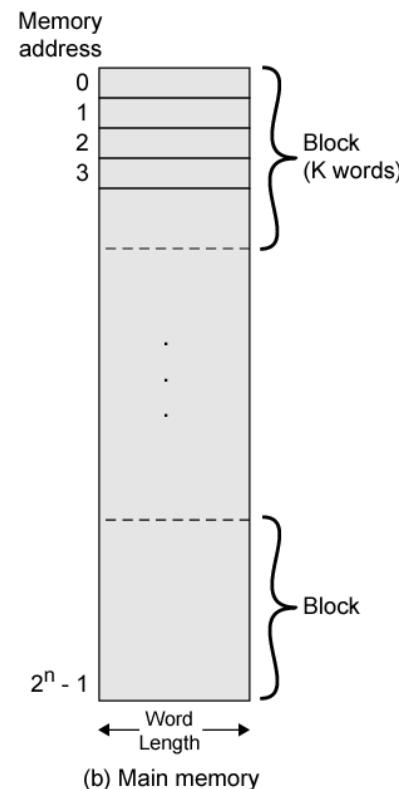
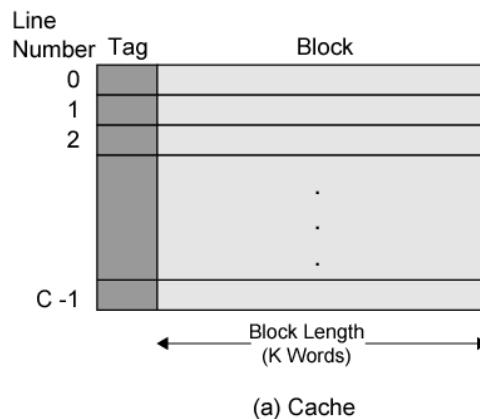
A small amount of high speed memory between the main memory and the processor.

May be located on processor chip or separate module



Cache memory Vs Main memory structure

- Cache is partitioned into **lines** (also called **blocks**).
- Each line has 4-64 bytes in it. During data transfer, a whole line is read or written.
- Cache includes **tags** to identify which block of main memory is in each cache line.



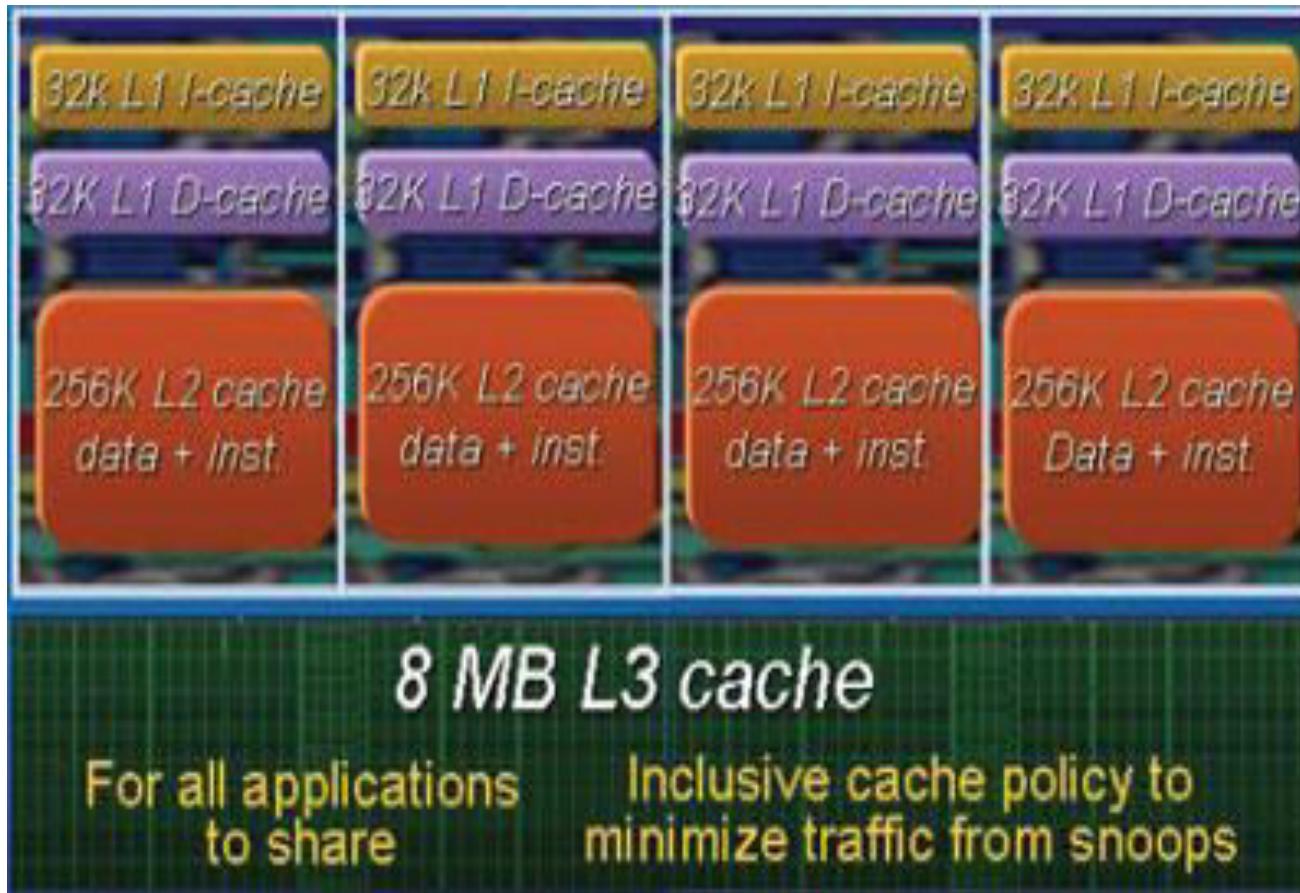
Cache organization

- Split cache
 - Separate caches for instructions and data
 - I-cache (Instruction) – mostly accessed sequentially
 - D-cache (data) – mostly random access
- Unified cache
 - Same cache for instruction and data
- Higher hit rate for unified cache as it balances between instruction and data
- Split caches eliminate contention for cache between the instruction processor and the execution unit – used for pipelining processes

Multilevel caches

- The penalty for a cache miss is the extra time that it takes to obtain the requested item from main memory.
- One way in which this penalty can be reduced is to provide another cache, the secondary cache, which is accessed in response to a miss in the primary cache.
- The primary cache is referred to as the L1 (level 1) cache and the secondary cache is called the L2 (level 2) cache.
- Most high-performance microprocessors include L2 and L3 cache which is often located off-chip, whereas the L1 cache is located on the same chip as the CPU.
- With a two-level cache, central memory has to be accessed only if a miss occurs in both caches.

Nehalem's micro-architecture showing different cache memory levels



Principle of locality

- Property of a program to reuse the data and instructions they have used recently.
- we need to predict what instructions and data a program will use in the near future based on its accesses in the recent past.
- The principle of locality also applies to data accesses, though not as strongly as to code accesses.
- **Types of locality**
 - Temporal locality: recently accessed items are likely to be accessed in the near future.
 - Spatial locality: items whose addresses are near one another tend to be referenced close together in time.

Try it yourself: Can you think of some practical cases of temporal and spatial locality?

Parameters of cache memory

i) Cache hit

Data found in cache.

Results in data transfer at maximum speed.

ii) Cache miss

Data not found in cache. Processor loads data from memory and copies into cache (miss penalty).

iii) Hit ratio

Ratio of number of hits to total number of references =>
number of hits/(number of hits + number of Miss)

iv) Miss penalty

Additional cycles required to serve the miss

Time required for the cache miss depends on both the latency and bandwidth

v) Latency – time to retrieve the first word of the block

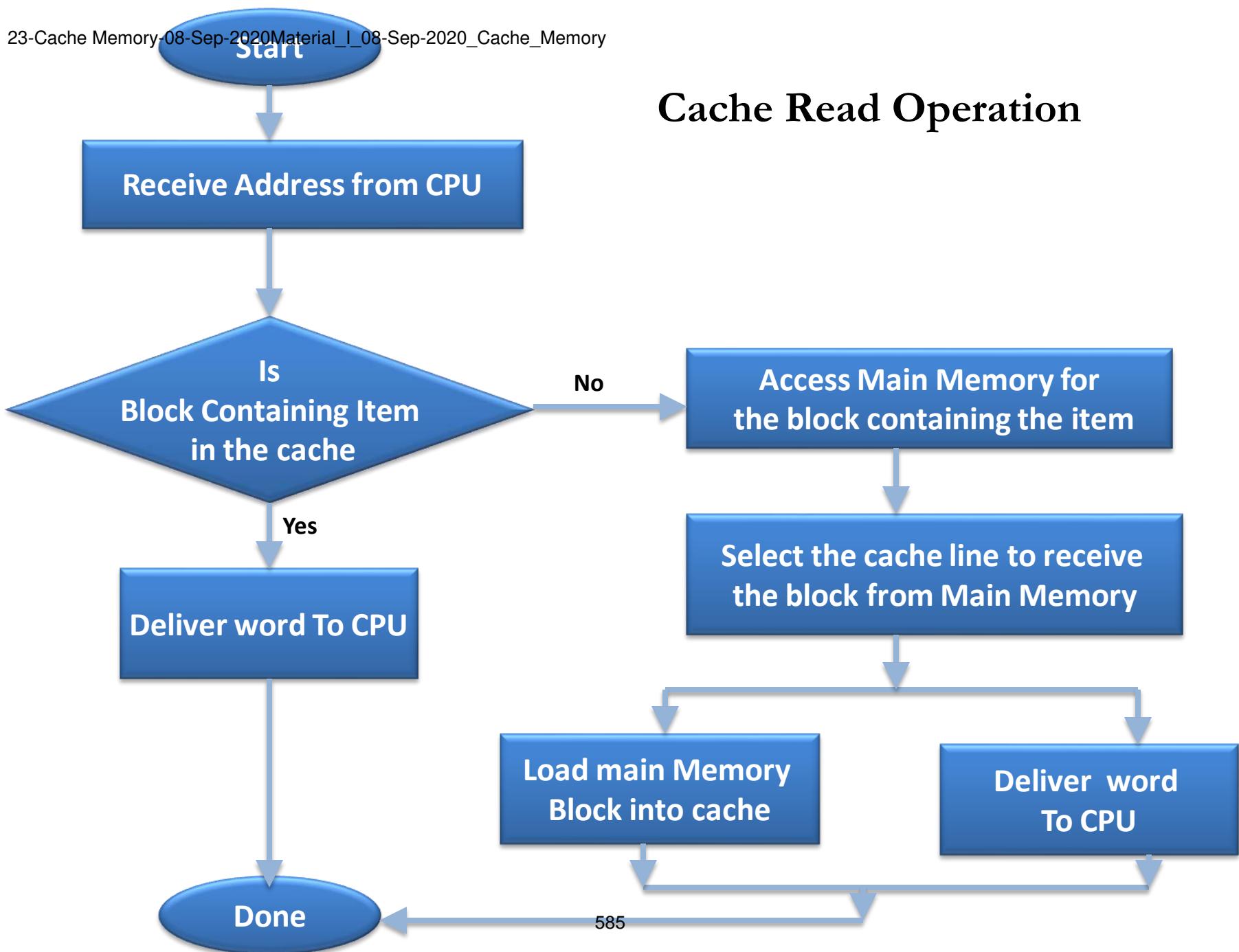
vi) Bandwidth – time to retrieve the rest of this block

Sources of Cache Misses (Three C's)

Compulsory Misses: These are misses that are caused by the cache being empty initially or by the first reference to a location in memory . Sometimes referred to as **Cold misses**.

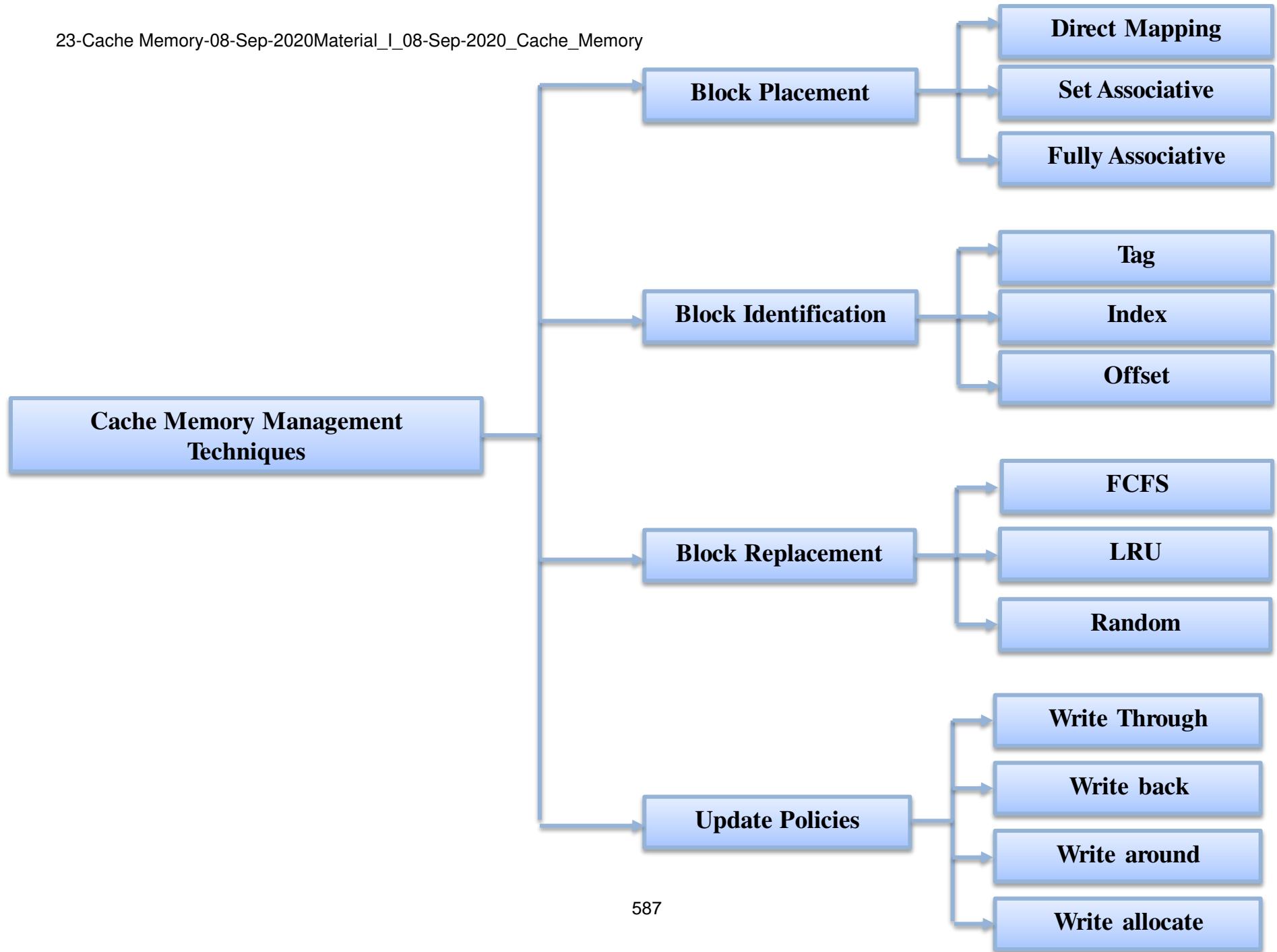
Capacity Misses : If the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

Conflict Misses: If the cache mapping is such that multiple blocks are mapped to the same cache entry. Common in set associative or direct mapped block placement, where a block can be discarded and later retrieved if too many blocks map to its set. Also called collision or interference misses.



Some Processor models and their Cache level specifications

Processor	Type	Year of Introduction	L1 Cache ^a	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 KB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 kB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA ^b	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 kB/32 kB	4 MB	—
Itanium 2	PC/server	2002	32 kB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24-48 MB



Example

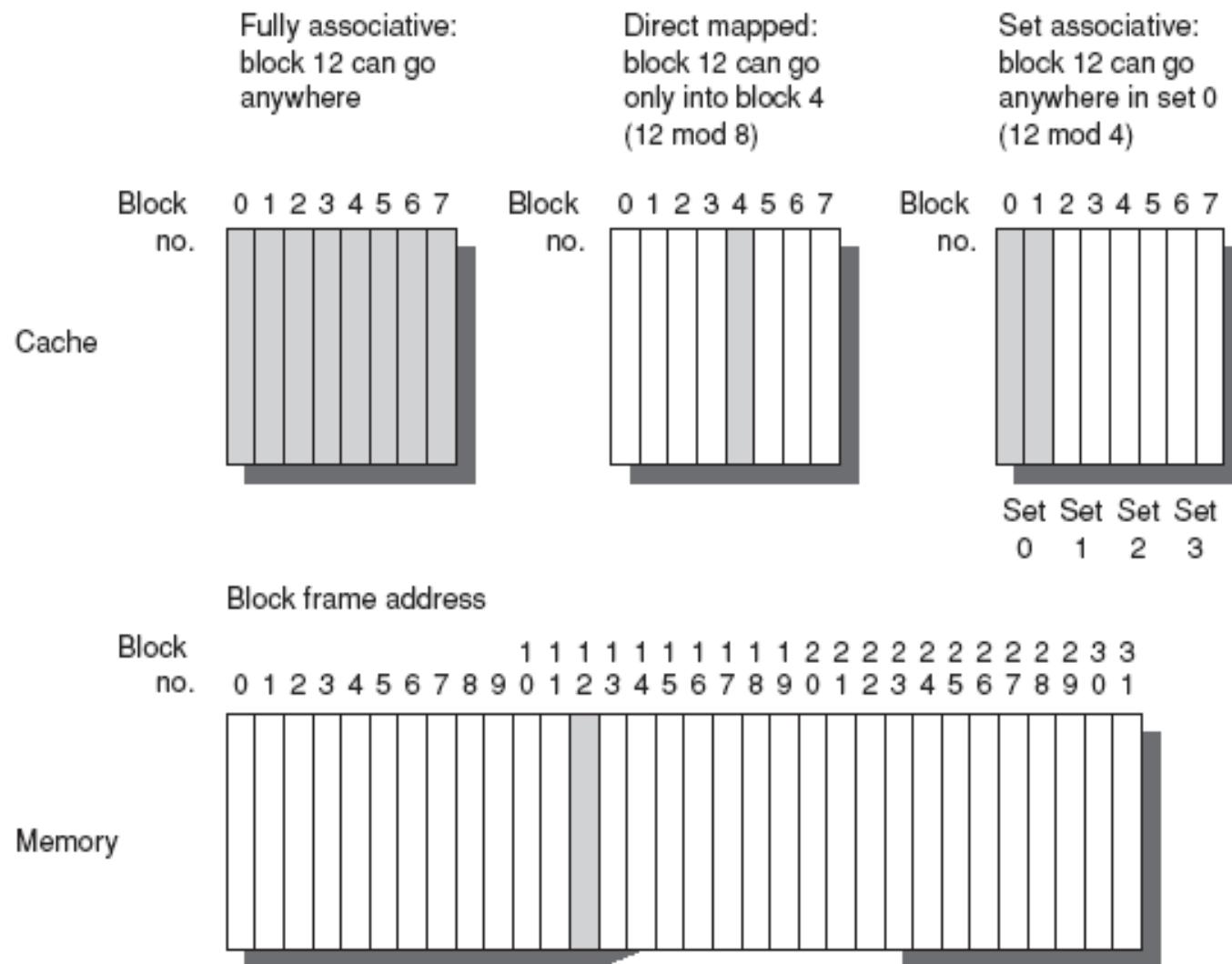
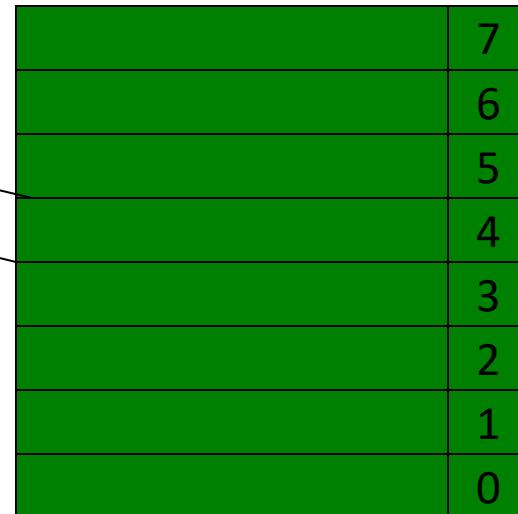


Figure C.2 This example cache has eight block frames and memory has 32 blocks.

Direct Mapping

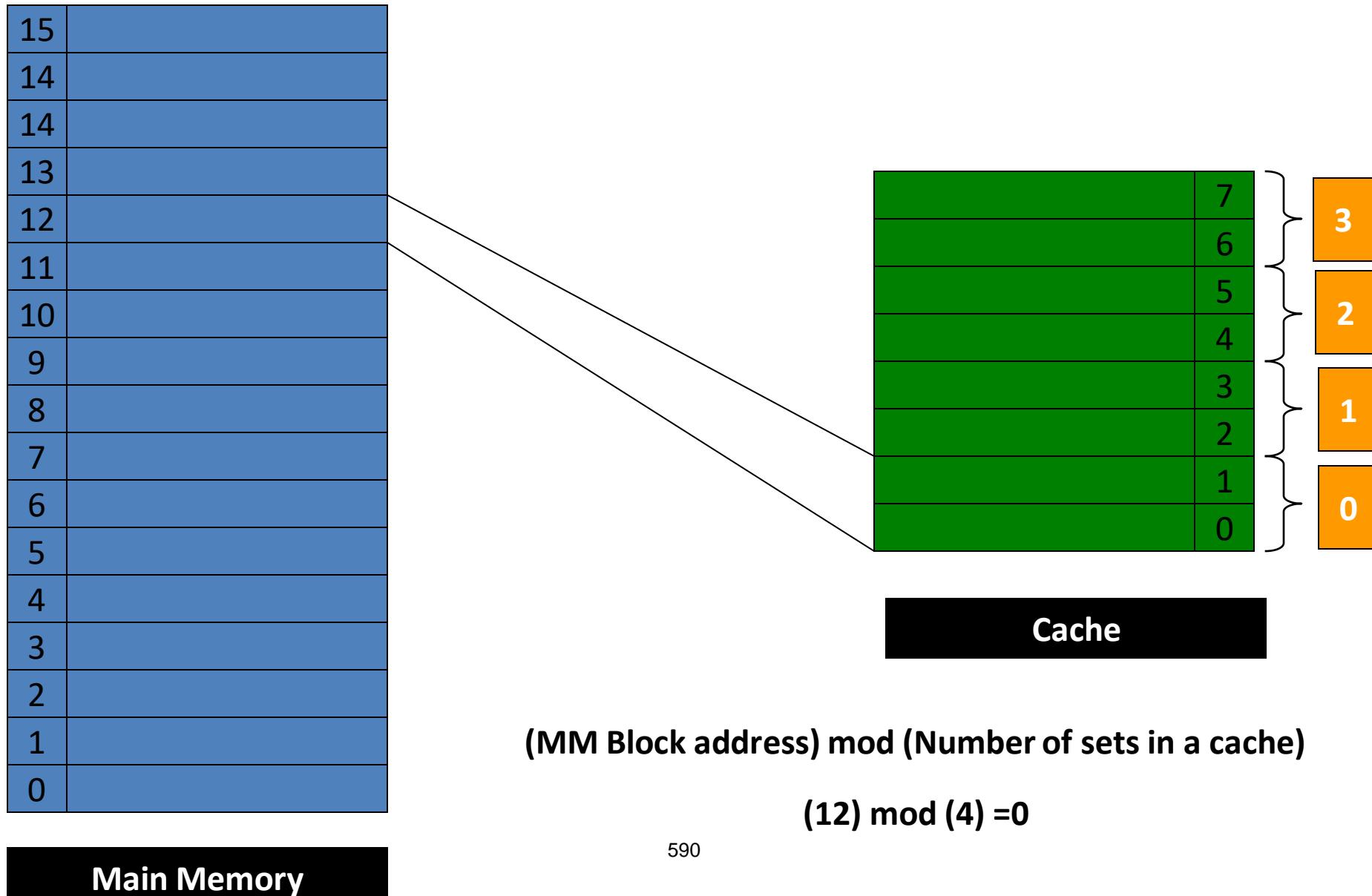


Cache

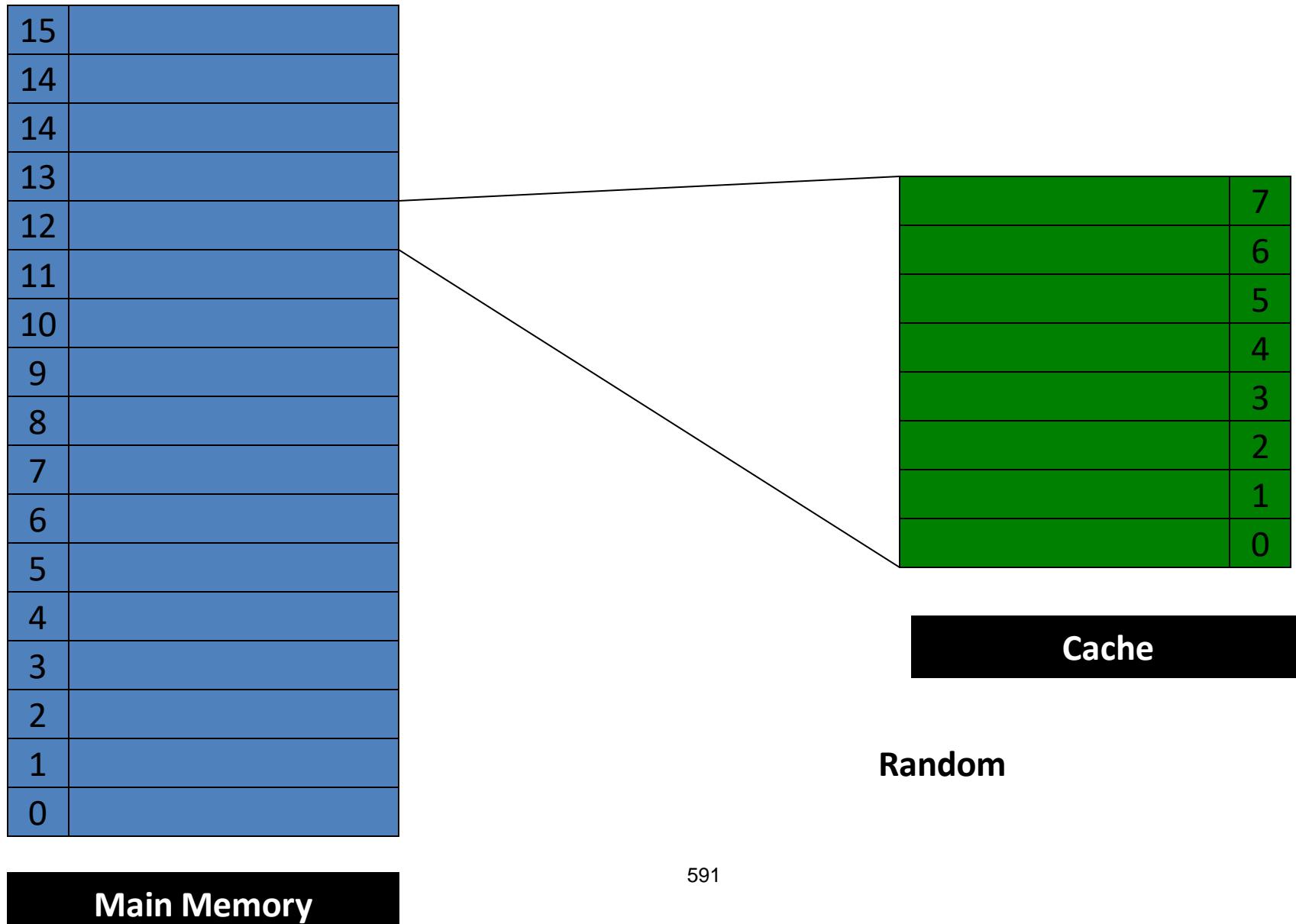
(MM Block address) mod (Number of lines in a cache)

$$(12) \bmod (8) = 4$$

Set Associative Mapping

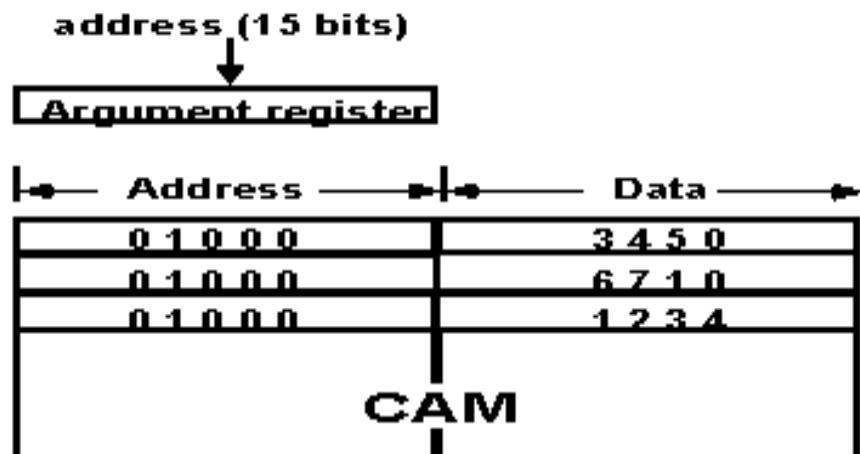


Fully Associative Mapping



Associative Mapping

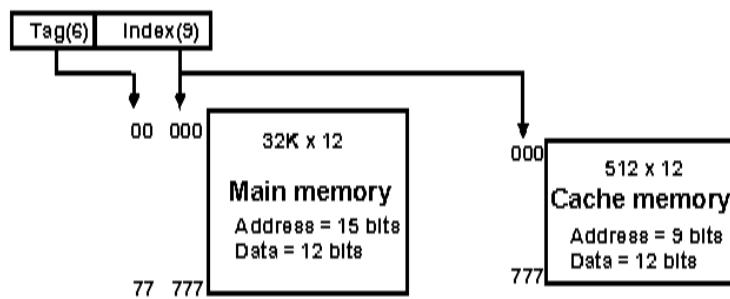
- Fastest, most flexible but very expensive
- Any block location in cache can store any block in memory
- Stores both the address and the content of the memory word
- CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address
- If found data is read and sent to the CPU else main memory is accessed.
- CAM – content addressable memory



Direct Mapping

- N-bit CPU memory address – k bits index field and (n-k) bits tag field
- Index bits are used to access the cache
- Each word in cache consists of data word and its associated tag

Addressing Relationships

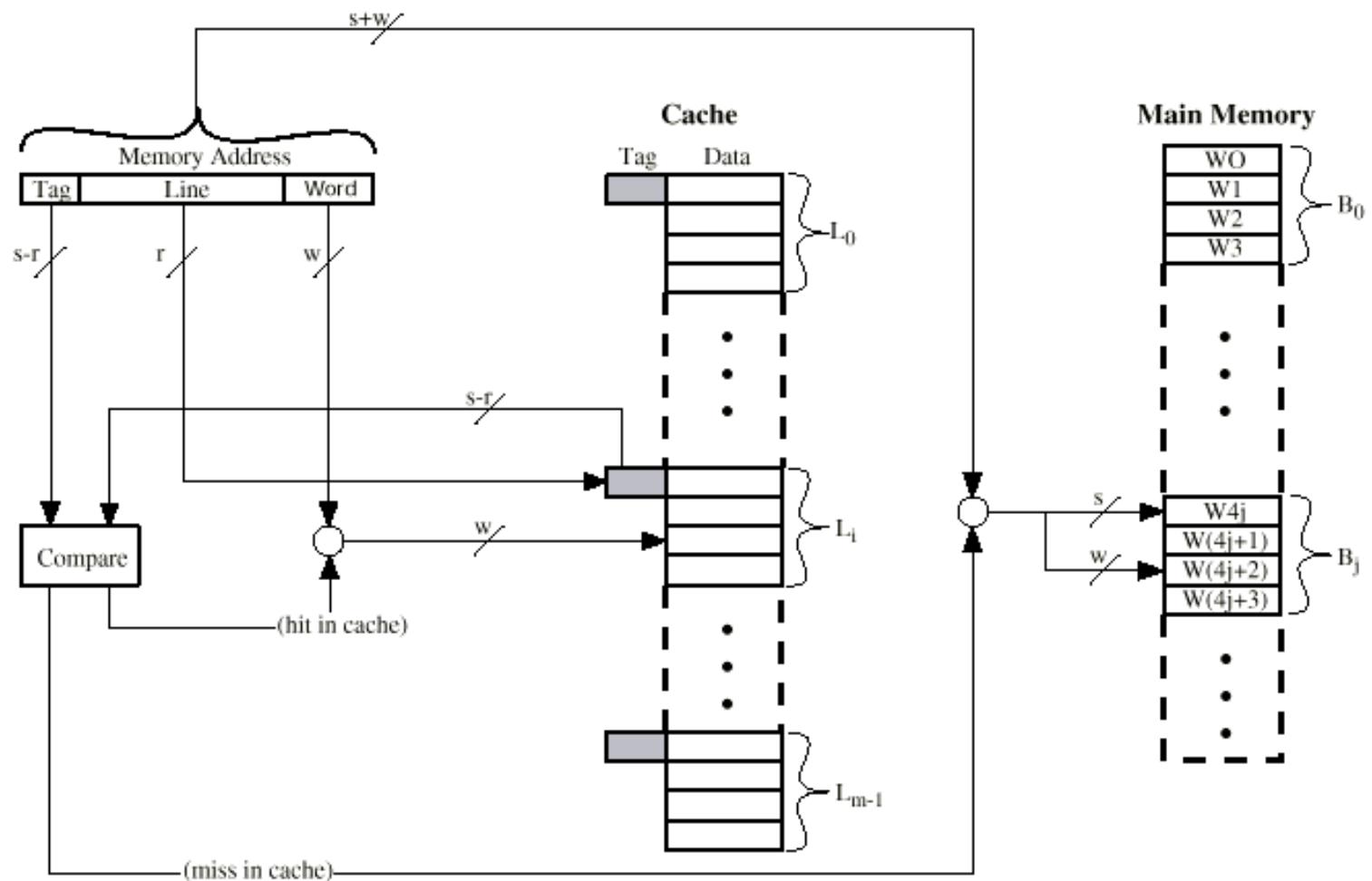


Memory address	Memory data
00000	1 2 2 0
00777	2 3 4 0
01000	3 4 5 0
01777	4 5 6 0
02000	5 6 7 0
02777	6 7 1 0

Index address	Tag	Data
000	0 0	1 2 2 0
777	0 2	6 7 1 0

On memory request, index field is used to access the cache. Tag field of CPU address is compared with the tag in the word read in the cache

Direct Mapping Cache organization



If match then there is hit

Else miss – word is read from the memory

It is then stored in the cache together with the new tag replacing the previous value

Disadvantage: hit ratio drops if 2 or more words with same index but different tags are accessed repeatedly.

When memory is divided into blocks of words, index field – block field and word field

Ex: 512 words cache – 64 blocks of 8words each – block field (6bits) and words field (3bits)

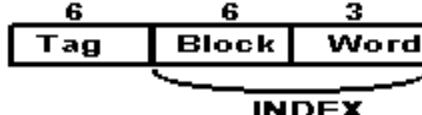
Why comparators in MMU?

- Comparators are using to compare the Tag present in the cache and tag bits in the address given.
- The number of comparators used is based on number of cache lines/ number of Tag field in cache.

- Tags within the block are same.

Direct Mapping with block size of 8 words

	Index	tag	data
Block 0	000	0 1	3 4 5 0
	007	0 1	6 5 7 8
Block 1	010		
	017		
Block 63	770	0 2	
	777	0 2	6 7 1 0



The INDEX field is divided into three parts: Tag (6 bits), Block (6 bits), and Word (3 bits). The word part is labeled 'Word' and has a curved arrow pointing to it. The index part is labeled 'INDEX'.

- When a miss occurs, entire block is transferred from main memory to cache.
- It is time consuming but improves hit ratio because of the sequential nature of programs.

Set-Associative Mapping

- Same index address of cache can represent 2 or more words of memory.

Set Associative Mapping Cache with set size of two

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

The comparison logic is done by an associative search of the tags in the set similar to an associative memory search, thus the name “Set-Associative”

- Hit ratio increases as the set size increases but more complex comparison logic is required when number of set size of cache increases
- When a miss occurs and set is full, one of tag-data items are replaced using block replacement policy

Problem 1

- A set associative cache consists of 64 lines or slots, divided into four line sets. Main memory consists 4k blocks of 128 words each. Show the format of main memory addresses.

The cache is divided into 16 sets of 4 lines each. Therefore, 4 bits are needed to identify the set number. Main memory consists of $4K = 2^{12}$ blocks. Therefore, the set plus tag lengths must be 12 bits and therefore the tag length is 8 bits. Each block contains 128 words. Therefore, 7 bits are needed to specify the word.

Main memory address =	TAG	SET	WORD
	8 600	4	7

Problem 2

- A two-way set associative cache has lines of 16 bytes and a total size of 8k bytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.

There are a total of $8 \text{ kbytes} / 16 \text{ bytes} = 512$ lines in the cache. Thus the cache consists of 256 sets of 2 lines each. Therefore 8 bits are needed to identify the set number. For the 64-Mbyte main memory, a 26-bit address is needed. Main memory consists of $64\text{-Mbyte} / 16 \text{ bytes} = 2^{22}$ blocks. Therefore, the set plus tag lengths must be 22 bits, so the tag length is 14 bits and the word field length is 4 bits.

	TAG	SET	WORD
Main memory address =	14	8	4

Try it yourself: Solve more problems on finding the format of memory address using Direct mapped and associative caches and see how the format differs for each case.

Block Replacement

- **Least Recently Used: (LRU)**

Replace that block in the set that has been in the cache longest with no reference to it.

- **First Come First Out: (FIFO)**

Replace that block in the set that has been in the cache longest.

- **Least Frequently Used: (LFU)**

Replace that block in the set that has experienced the fewest references

Update Policies - Write Through

- Update main memory with every memory write operation
- Cache memory is updated in parallel if it contains the word at specified address.
- Advantage: main memory always contains the same data as the cache
- It is important during DMA transfers to ensure the data in main memory is valid
- Disadvantage: slow due to memory access time

Write Back

- Only cache is updated during write operation and marked by flag. When the word is removed from the cache, it is copied into main memory
- Memory is not up-to-date, i.e., the same item in cache and memory may have different value

Update policies – Contd..

- Write-Allocate
 - update the item in main memory and bring the block containing the updated item into the cache.
- Write-Around or Write-no-allocate
 - correspond to items not currently in the cache (i.e. write misses) the item is updated in main memory only without affecting the cache.

Cache Reference Policies

- Look through: The cache is checked first for a hit, and if a miss occurs then the access to main memory is started.
- Look aside: access to main memory in parallel with the cache lookup.

Performance analysis

- **Look through**

$$T_A = T_c + (1-h)*T_m$$

T_A – Average read access time

T_c is the average cache access time

T_m is the average main memory access time

- **Look aside**

$$T_A = h*T_c + (1-h)*T_m$$

number of references found in the cache

- hit ratio $h = \frac{\text{number of references found in the cache}}{\text{total number of memory references}}$

- **Miss Ratio $m=(1-h)$**

Example: assume that a computer system employs a cache with an access time of 20ns and a main memory with an access/cycle time of 200ns. Suppose that the hit ratio for reads is 90%,

- a) what would be the average access time for reads if the cache is a look through-cache?

The average read access time (T_A) = $T_c + (1-h)*T_m$

$$20\text{ns} + 0.10*200\text{ns} = 40\text{ns}$$

- b) what would be the average access time for reads if the cache is a “look-Aside” cache?

The average read access time in this case (T_A)

$$= h*T_c + (1-h)*T_m = 0.9*20\text{ns} + 0.10*200\text{ns} = 38\text{ns}$$

Problem 1

- Consider a memory system with $T_c = 100\text{ns}$ and $T_m = 1200\text{ns}$. If the effective access time is 10% greater than the cache access time, what is the hit ratio H in look-through cache?

$$\Rightarrow T_A = T_C + (1-h)*T_M$$

$$\Rightarrow 1.1 T_c = T_c + (1-h)*T_M$$

$$\Rightarrow 0.1 T_C = (1-h) * T_M$$

$$\Rightarrow 0.1 * 100 = (1-h) * 1200$$

$$\Rightarrow 1-h = 10/1200$$

$$\Rightarrow h = 1190/1200$$

Problem 2

- A computer system employs a write-back cache with a 70% hit ratio for writes. The cache operates in look-aside mode and has a 90% read hit ratio. Reads account for 80% of all memory references and writes account for 20%. If the main memory cycle time is 200ns and the cache access time is 20ns, what would be the average access time for all references (reads as well as writes)?

Total Reference

The average access time for reads

$$= 0.9*20\text{ns} + 0.1*200\text{ns} = 38\text{ns}.$$

The average write time

$$= 0.7*20\text{ns} + 0.3*200\text{ns} = 74\text{ns}$$

Hence the overall average access time for combined reads and writes is

$$= 0.8*38\text{ns} + 0.2*74\text{ns} = 45.2\text{ns}$$

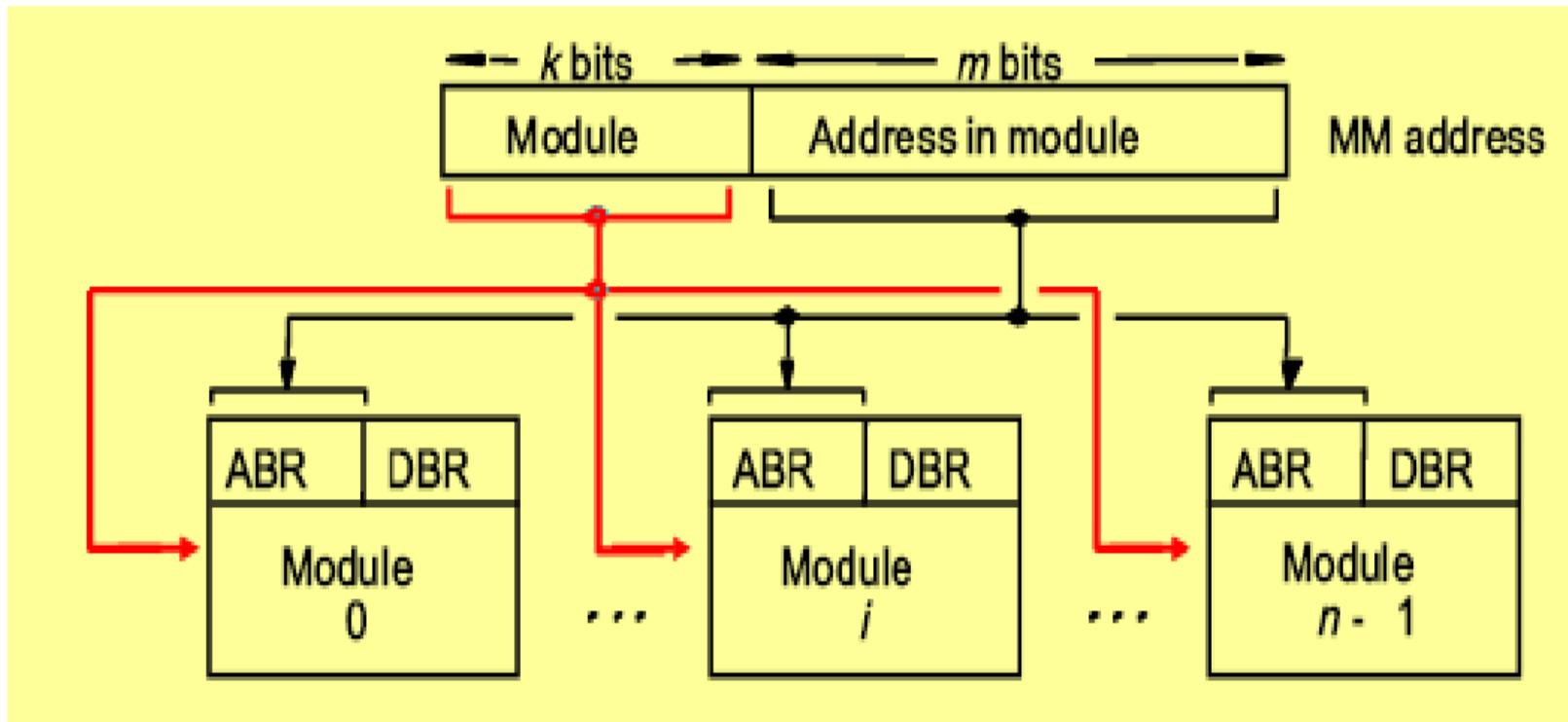
References

- D.A. Patterson & J. L. Hennessy, Computer Organization and Design: The hardware/software interface, Fifth Edition, Morgan Kaufman, 2011.

Memory Interleaving

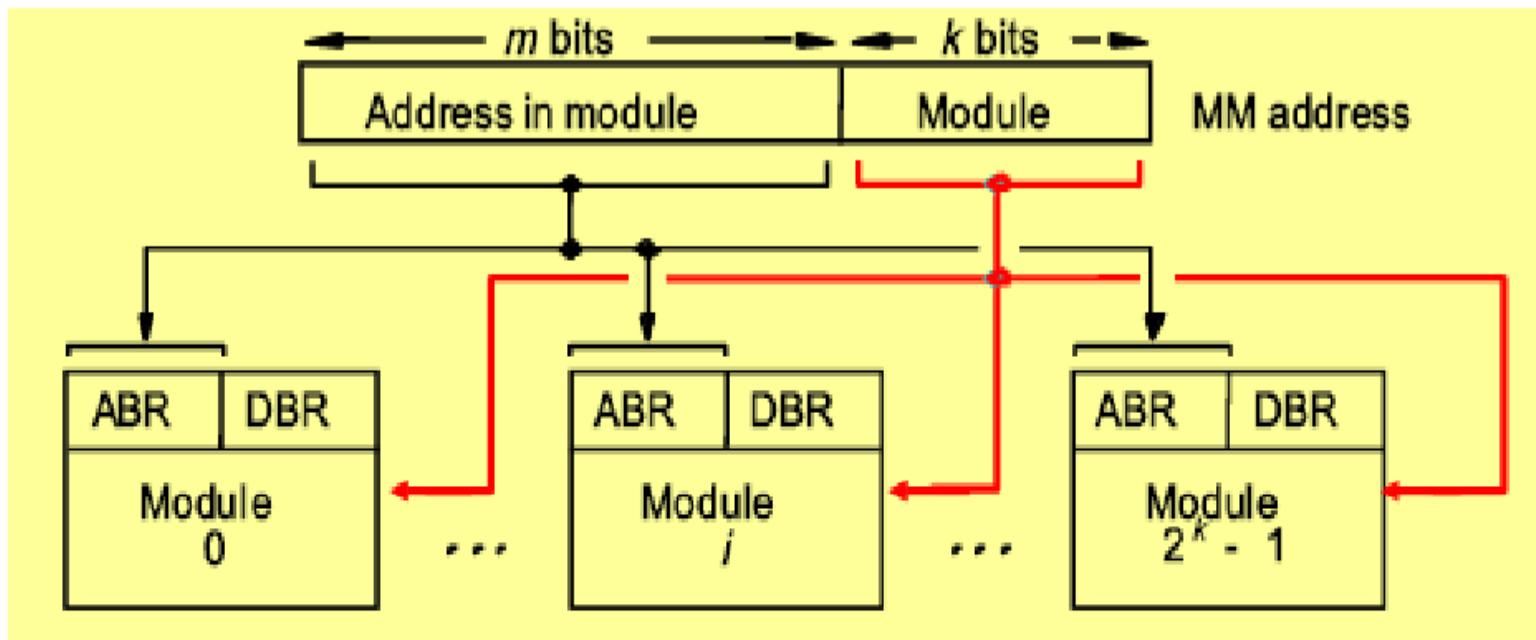
- If the main memory is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time.
- Hence, aggregate rate of transmission of words to and from the memory can be increased.
- Two methods of distribution of words among modules
 - Consecutive words in a module
 - Consecutive words in consecutive modules

Consecutive words in a module



- When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved.

Consecutive words in consecutive modules



- This method is called memory interleaving
- Parallel access is possible. Hence, faster
- Higher average utilization of the memory system

Cache coherence

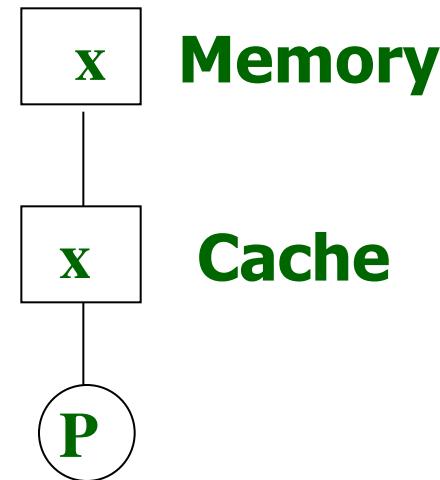
Single Processor caching

Hit: data in the cache

Miss: data is not in the cache

Hit rate: h

Miss rate: $m = (1-h)$

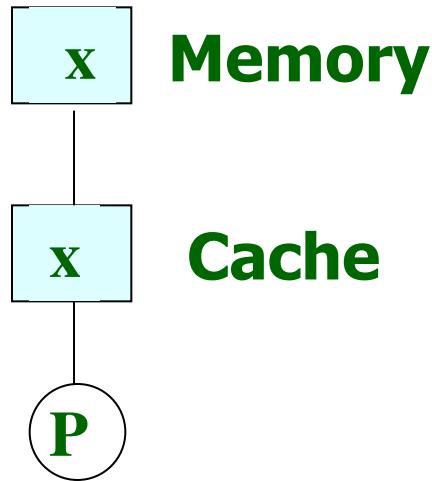


Cache Coherence Policies

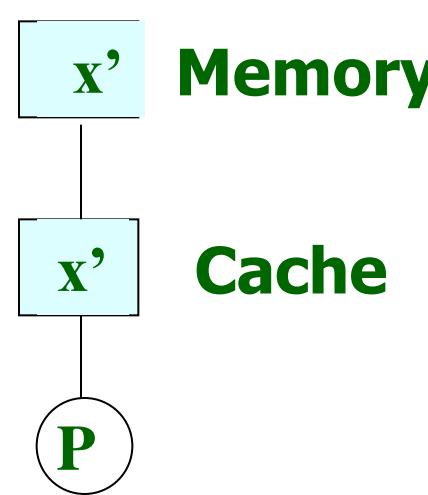
► Writing to Cache in 1 processor case

- Write Through
- Write Back

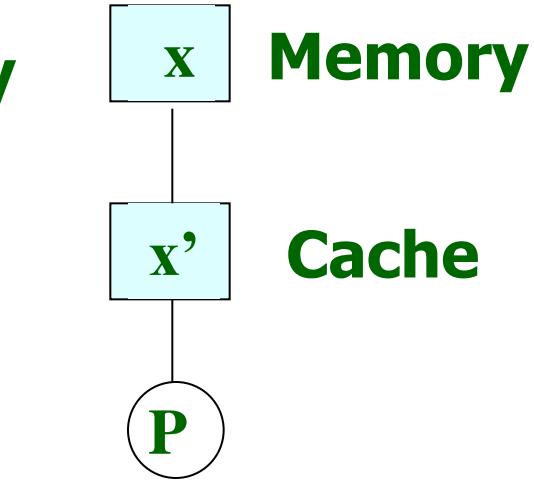
Writing in the cache



Before

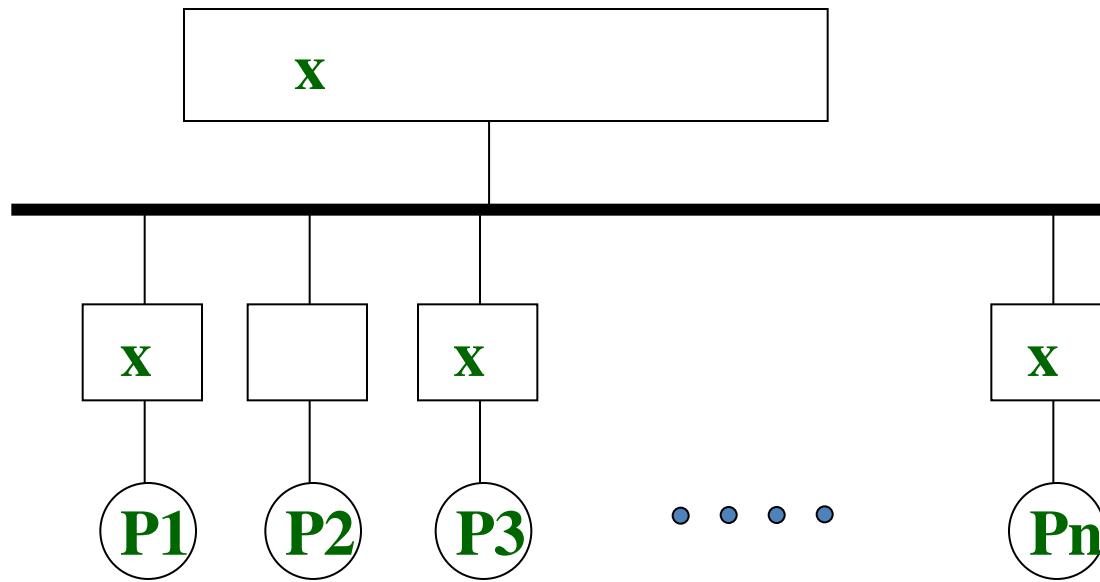


Write through



Write back

Cache Coherence

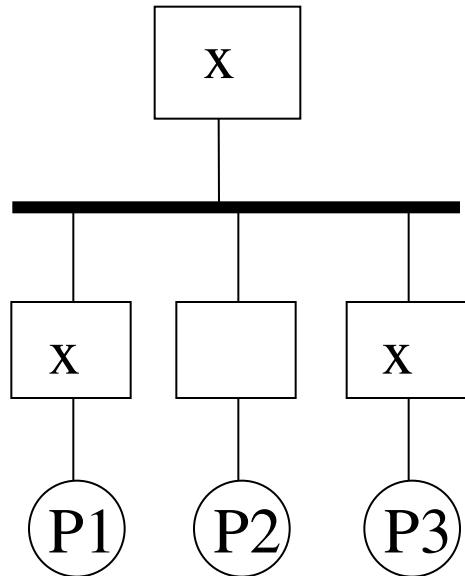


- Multiple copies of x
- What if P1 updates x?

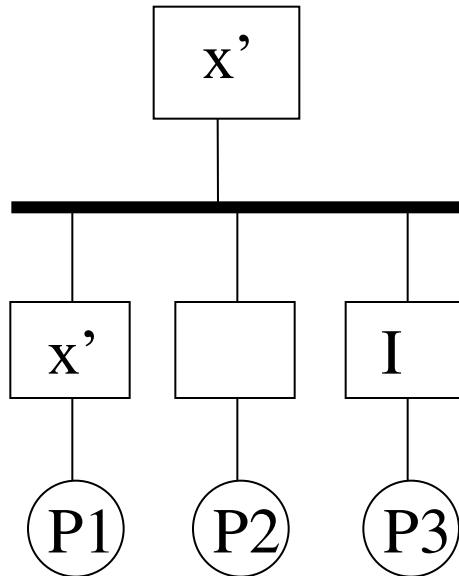
Cache Coherence Policies

- Writing to Cache in n processor case
 - Write Update - Write Through
 - Write Invalidate - Write Back
 - Write Update - Write Back
 - Write Invalidate - Write Through

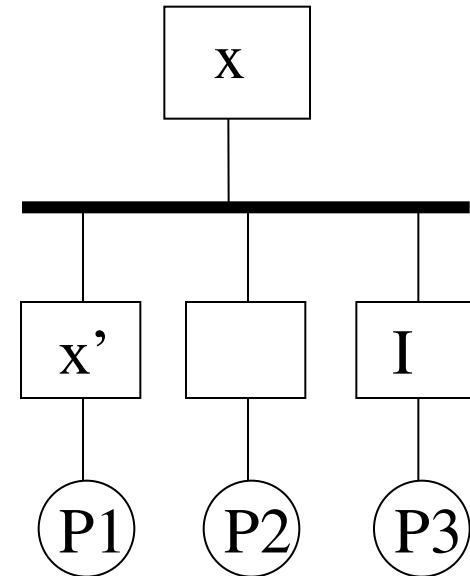
Write-invalidate



Before

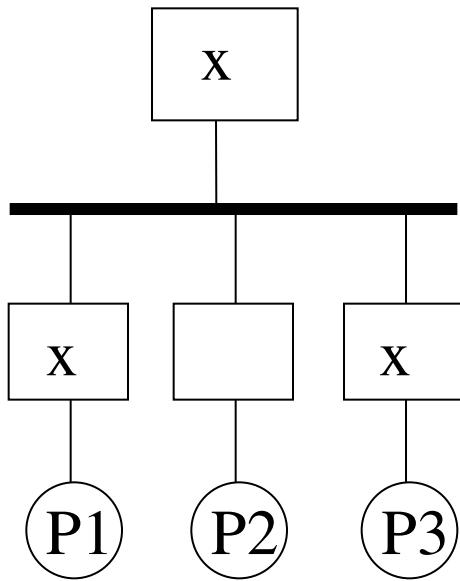


Write Through

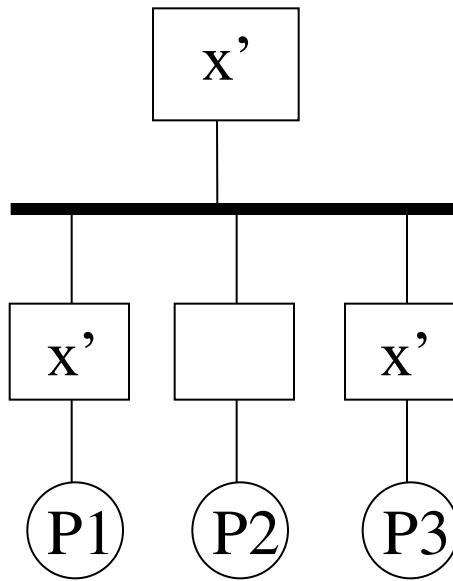


Write back

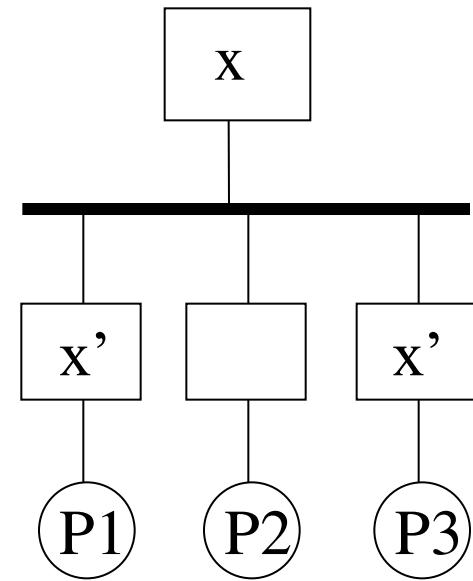
Write-Update



Before



Write Through



Write back

Snooping Protocols

- Snooping protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary.
- Global memory is moved in blocks, and each block has a state associated with it, which determines what happens to the entire contents of the block.
- The state of a block might change as a result of the operations Read-Miss, Read-Hit, Write-Miss, and Write-Hit.

Thank you

Write Invalidate Write Through

- **Multiple processors can read block copies from main memory safely until one processor updates its copy.**
- **At this time, all cache copies are invalidated and the memory is updated to remain consistent.**

Write Through- Write Invalidate (cont.)

State	Description
Valid [VALID]	The copy is consistent with global memory
Invalid [INV]	The copy is inconsistent

Write Through-Write Invalidate (cont.)

Event	Actions
Read Hit	Use the local copy from the cache.
Read Miss	Fetch a copy from global memory. Set the state of this copy to Valid.
Write Hit	Perform the write locally. Broadcast an Invalid command to all caches. Update the global memory.
Write Miss	Get a copy from global memory. Broadcast an invalid command to all caches. Update the global memory. Update the local copy and set its state to Valid.
Replace	Since memory is always consistent, no write back is needed when a block is replaced.

Write Back- Write Invalidate (ownership)

- A valid block can be owned by memory and shared in multiple caches that can contain only the shared copies of the block.
- Multiple processors can safely read these blocks from their caches until one processor updates its copy.
- At this time, the writer becomes the only owner of the valid block and all other copies are invalidated.

Write Back- Write Invalidate (cont.)

State	Description
Shared (Read-Only) [RO]	Data is valid and can be read safely. Multiple copies can be in this state
Exclusive (Read-Write) [RW]	Only one valid cache copy exists and can be read from and written to safely. Copies in other caches are invalid
Invalid [INV]	The copy is inconsistent

Ownership (cont.)

Event	Action
Read Hit	Use the local copy from the cache.
Read Miss:	If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (Read-Only) in both caches.

Ownership (cont.)

Write Hit	If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).
Write Miss	Get a copy from either a cache with an Exclusive (Read-Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).
Block Replacement	If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write back is needed when a block is replaced.

Write Once

This write-invalidate protocol, which was proposed by Goodman in 1983 uses a combination of write-through and write-back. Write-through is used the very first time a block is written. Subsequent writes are performed using write back.

Write Once (cont.)

State	Description
Invalid [INV]	The copy is inconsistent.
Valid [VALID]	The copy is consistent with global memory.
Reserved [RES]	Data has been written exactly once and the copy is consistent with global memory. There is only one copy of the global memory block in one local cache.
Dirty [DIRTY]	Data has been updated more than once and there is only one copy in one local cache. When a copy is dirty, it must be written back to global memory

Write Once

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

(Cont.)

Event	Actions
Read Hit	Use the local copy from the cache.
Read Miss	If no Dirty copy exists, then supply a copy from global memory. Set the state of this copy to Valid. If a dirty copy exists, make a copy from the cache that set the state to Dirty, update global memory and local cache with the copy. Set the state to VALID in both caches.

Write Once

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

(Cont.)

Write Hit	If the copy is Dirty or Reserved, perform the write locally, and set the state to Dirty. If the state is Valid, then broadcast an Invalid command to all caches. Update the global memory and set the state to Reserved.
Write Miss	Get a copy from either a cache with a Dirty copy or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Dirty.
Block Replacement	If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid, Reserved, or Invalid states, no write back is needed when a block is replaced.

Write update and partial write through

In this protocol an update to one cache is written to memory at the same time it is broadcast to other caches sharing the updated block. These caches snoop on the bus and perform updates to their local copies. There is also a special bus line, which is asserted to indicate that at least one other cache is sharing the block.

Write update and partial write through (cont.)

State	Description
Valid Exclusive [VAL-X]	This is the only cache copy and is consistent with global memory
Shared [SHARE]	There are multiple caches copies shared. All copies are consistent with memory
Dirty [DIRTY]	This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Copy ownership)

Write update and partial write through (cont.)

Event	Action
Read Hit	Use the local copy from the cache. State does not change
Read Miss:	If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared in both caches. If the cache copy was in a Dirty state, the value must also be written to memory.

Write update and partial write through (cont.)

Write Hit	<p>Perform the write locally and set the state to Dirty. If the state is Shared, then broadcast data to memory and to all caches and set the state to Shared. If other caches no longer share the block, the state changes from Shared to Valid Exclusion.</p>
Write Miss	<p>The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update and update all other caches that share the block and global memory. Set the state to Shared. If the copy comes from memory, perform the write and set the state to Dirty.</p>
Block Replacement	<p>If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive or Shared states, no write back is needed when a block is replaced.</p>

Write Update Write Back

This protocol is similar to the previous one except that instead of writing through to the memory whenever a shared block is updated, memory updates are done only when the block is being replaced.

Write Update Write Back (cont.)

State	Description
Valid Exclusive [VAL-X]	This is the only cache copy and is consistent with global memory
Shared Clean [SH-CLN]	There are multiple caches copies shared.
Shared Dirty [SH-DRT]	There are multiple shared caches copies. This is the last one being updated. (Ownership)
Dirty [DIRTY]	This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Ownership)

Write Update Write Back (cont.)

Event	Action
Read Hit	Use the local copy from the cache. State does not change
Read Miss:	If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared Clean. If the supplying cache copy was in a Valid Exclusion or Shared Clean, its new state becomes Shared Clean. If the supplying cache copy was in a Dirty or Shared Dirty state, its new state becomes Shared Dirty.

Write Update Write Back (cont.)

Write Hit	If the state was Valid Exclusive or Dirty, Perform the write locally and set the state to Dirty. If the state is Shared Clean or Shared Dirty, perform update and change state to Shared Dirty. Broadcast the updated block to all other caches. These caches snoop the bus and update their copies and set their state to Shared Clean.
Write Miss	The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update, set the state to Shared Dirty, and broadcast the updated block to all other caches. Other caches snoop the bus, update their copies, and change their state to Shared Clean. If the copy comes from memory, perform the write and set the state to Dirty.
Block Replacement	If a copy is in a Dirty or Shared Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive, no write back is needed when a block is replaced.

Directory Based

Protocols

Due to the nature of some interconnection networks and the size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical .

Examples??

Cache coherence protocols that somehow store information on where copies of blocks reside are called directory schemes.

What is a directory?

A directory is a data structure that maintains information on the processors that share a memory block and on its state. The information maintained in the directory could be either centralized or distributed.

Centralized vs.

Distributed

A Central directory maintains information about all blocks in a central data structure.

Bottleneck, large search time!

The same information can be handled in a distributed fashion by allowing each memory module to maintain a separate directory.

Categorization

- **Full Map Directories**

- **Limited Directories**

- **Chained Directories**

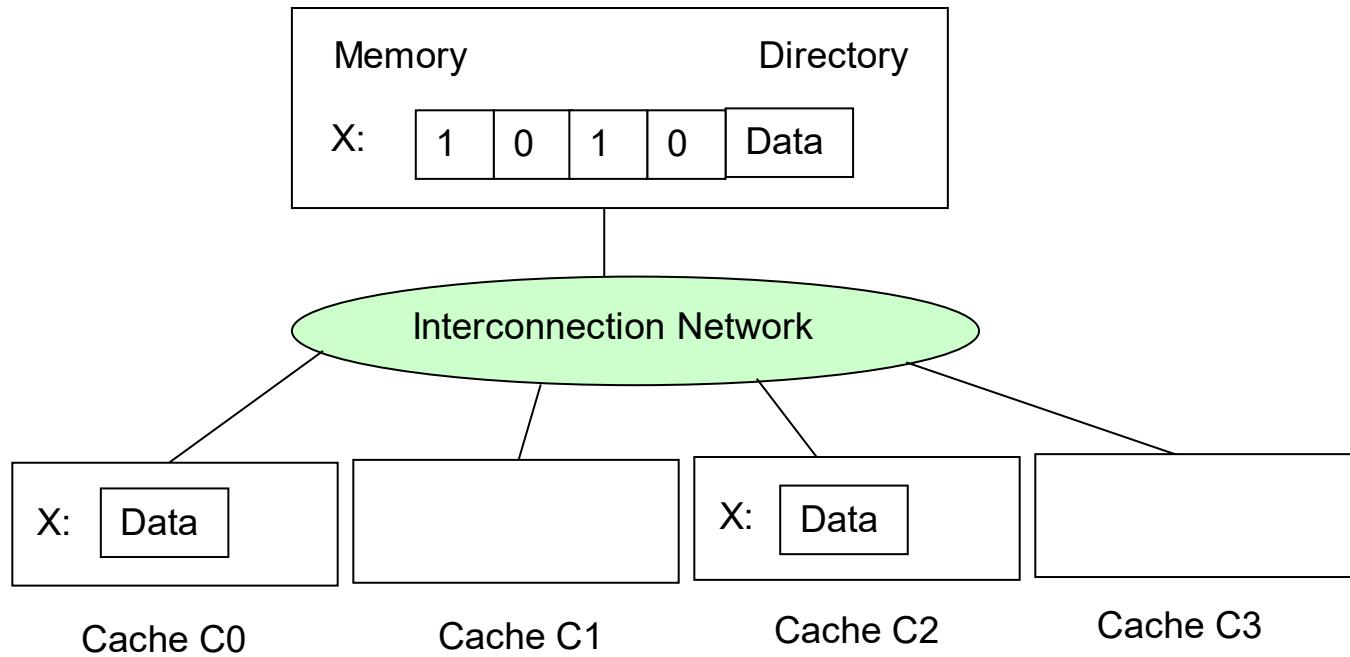
Full Map Directory

- Each directory entry contains N pointers, where N is the number of processors.
- There could be N cached copies of a particular block shared by all processors.
- For every memory block, an N bit vector is maintained, where N equals the number of processors in the shared memory system. Each bit in the vector corresponds to one processor.

Full Map

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

Directory

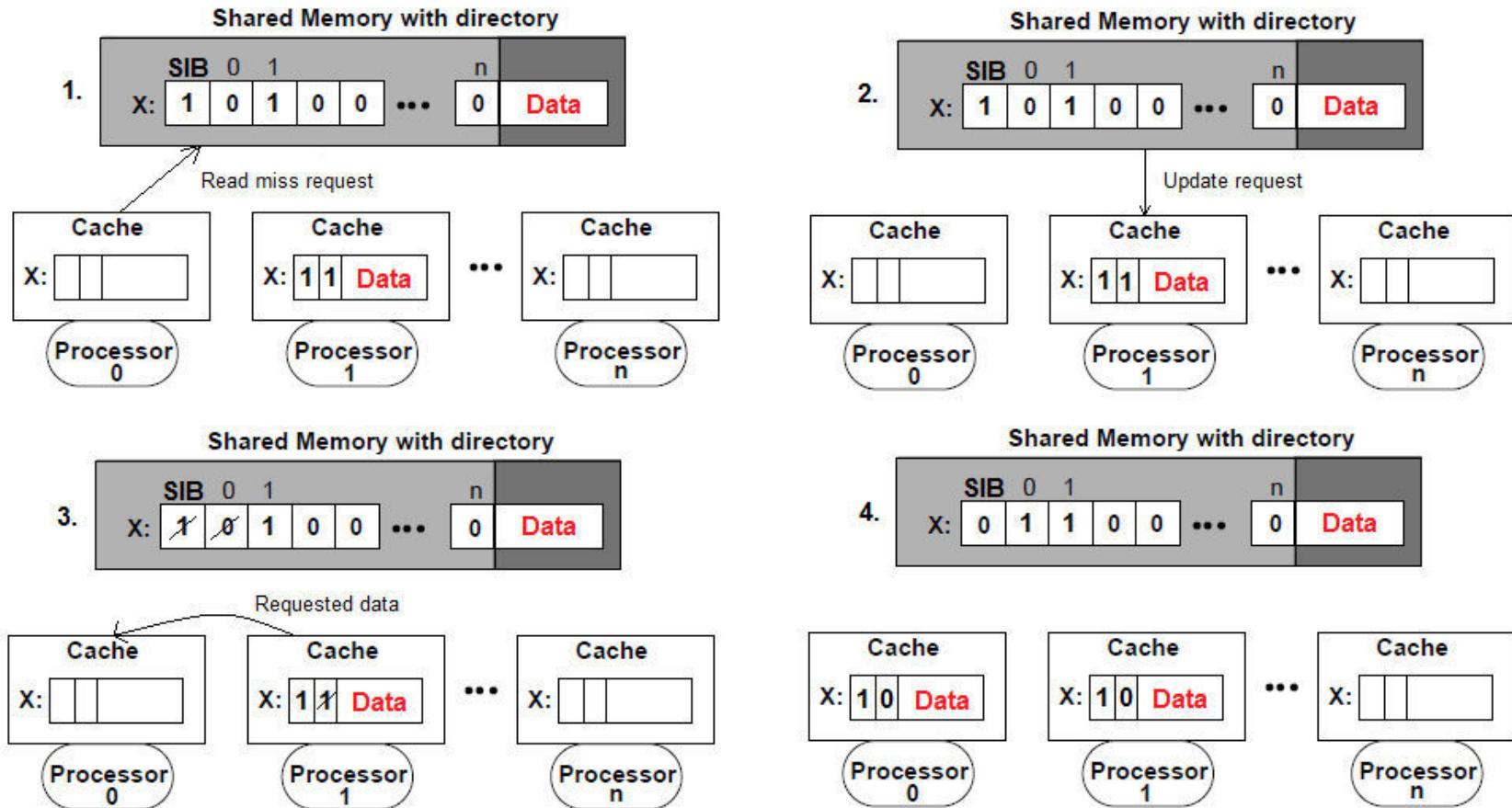


Full Map Directory

- Read miss
 - Requester sends read miss request to memory
 - Single inconsistent bit:
 - 0: One of the sharing cache sends data to requester or data comes from memory
 - 1: Memory send update request to other private cache. Private cache sends the data to the requester and to memory for update
 - Memory update directory state
 - Requester CPU reads the data from its cache

Full Map Directory

- Read miss (S.I.B. = 1)

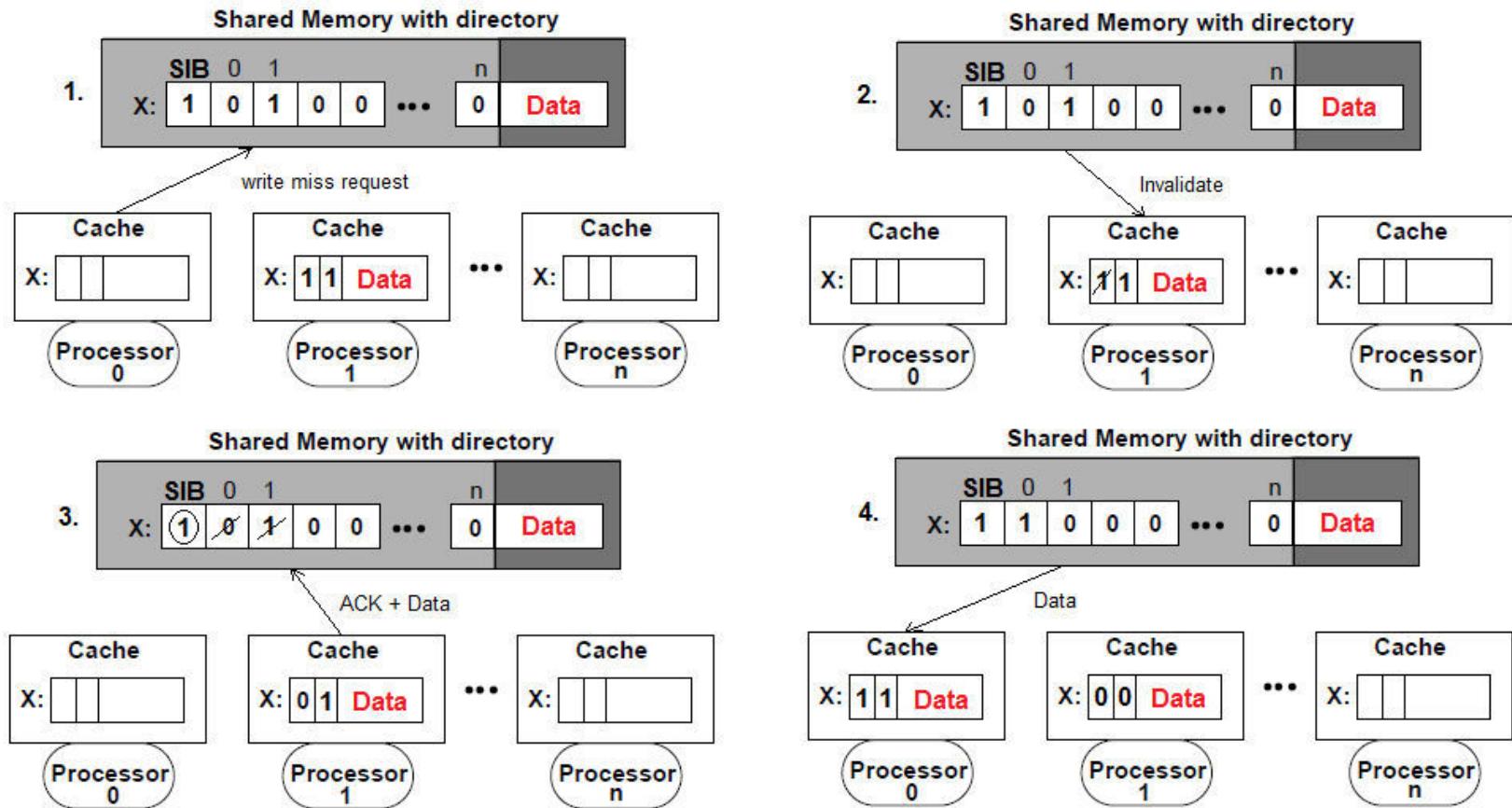


Full Map Directory

- Write miss
 - Requester sends write miss request to memory
 - Memory sends invalidate signal to sharing caches
 - Sharing caches sends ACK signal
 - Memory updates directory state
 - Requester has private access to data

Full Map Directory

- Write miss



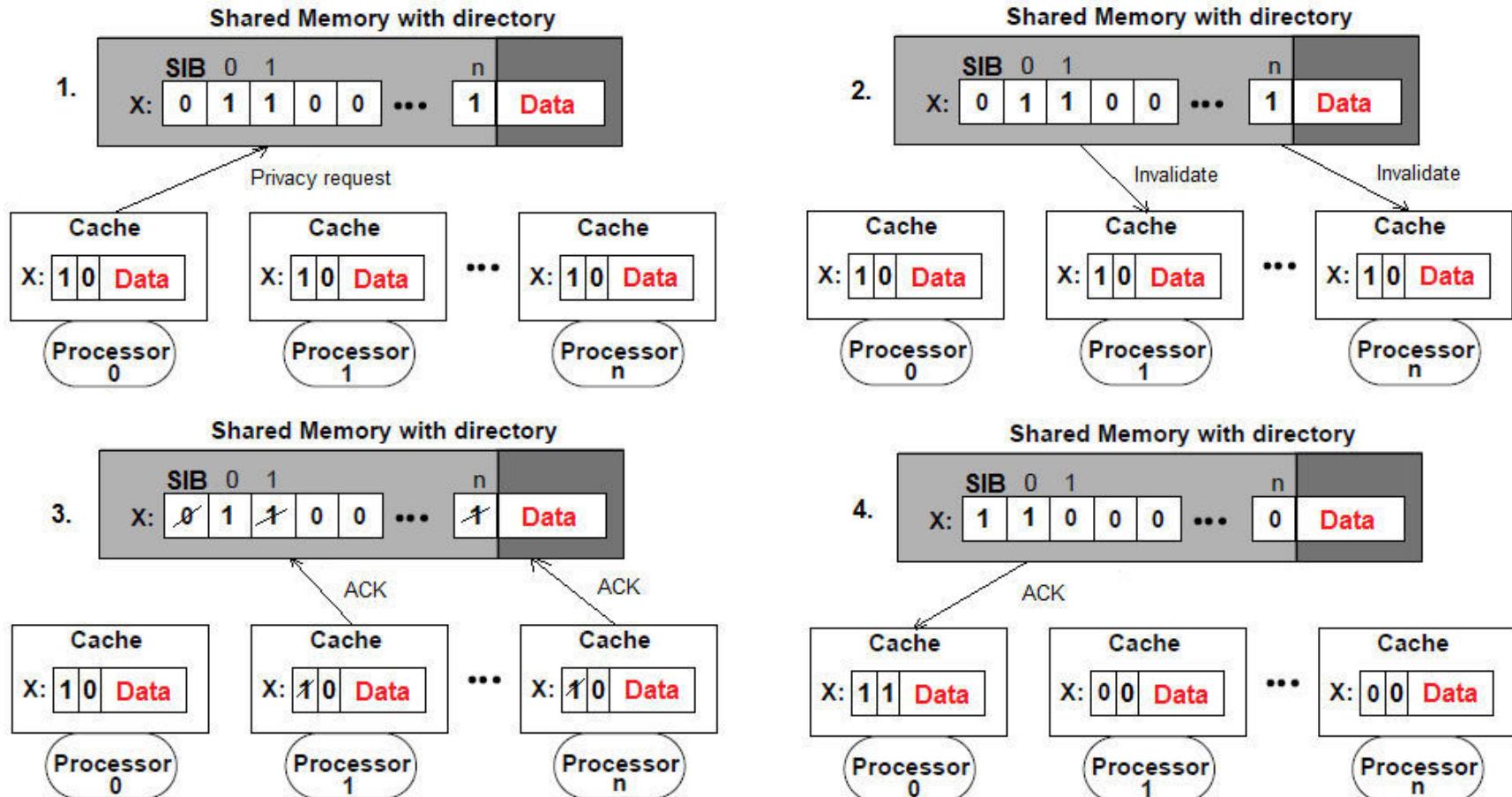
Full Map Directory

- Write hit

- Requester sends privacy request to memory
- Memory sends invalidate signals to other sharers
- Invalidated caches send ACK signal to memory
- Shared memory sends ACK signal to requester
- Requester CPU can write to data exclusively

Full Map Directory

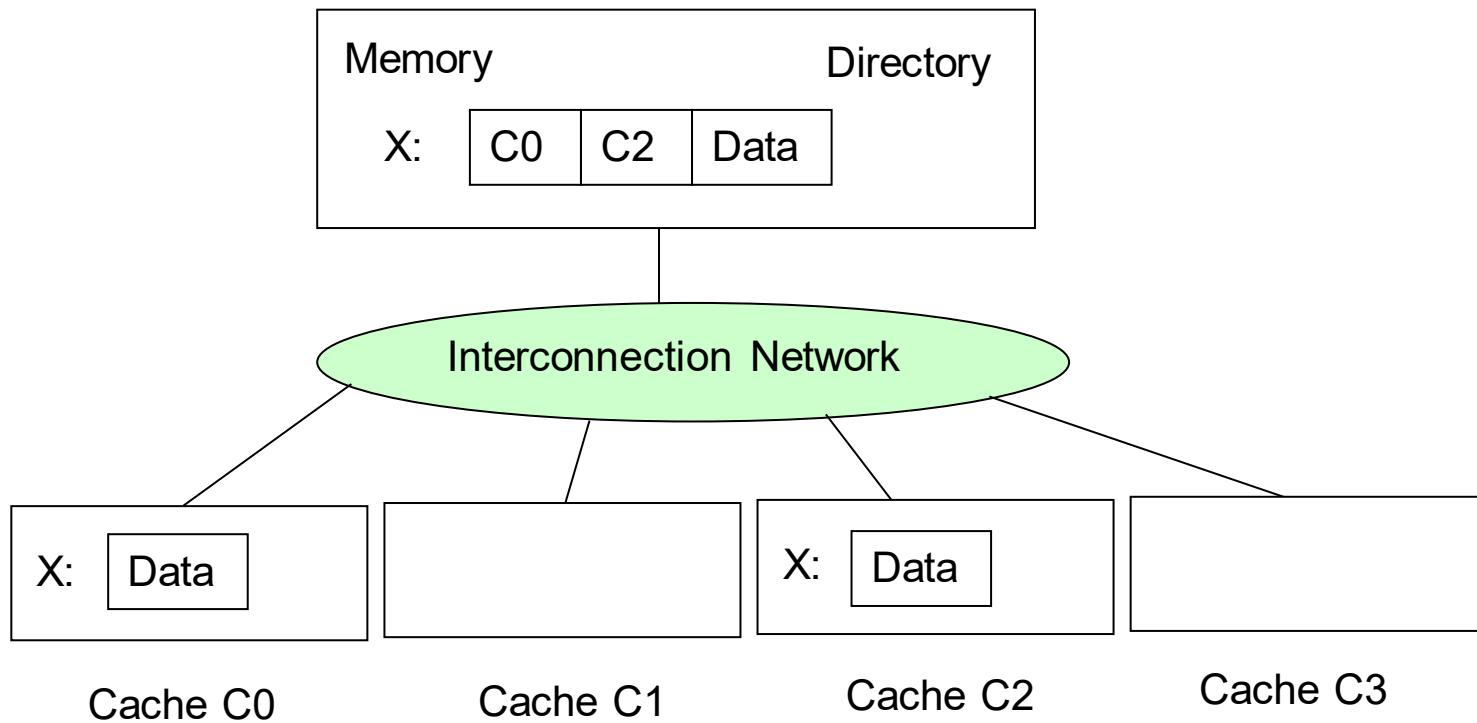
- Write hit



Limited Directory

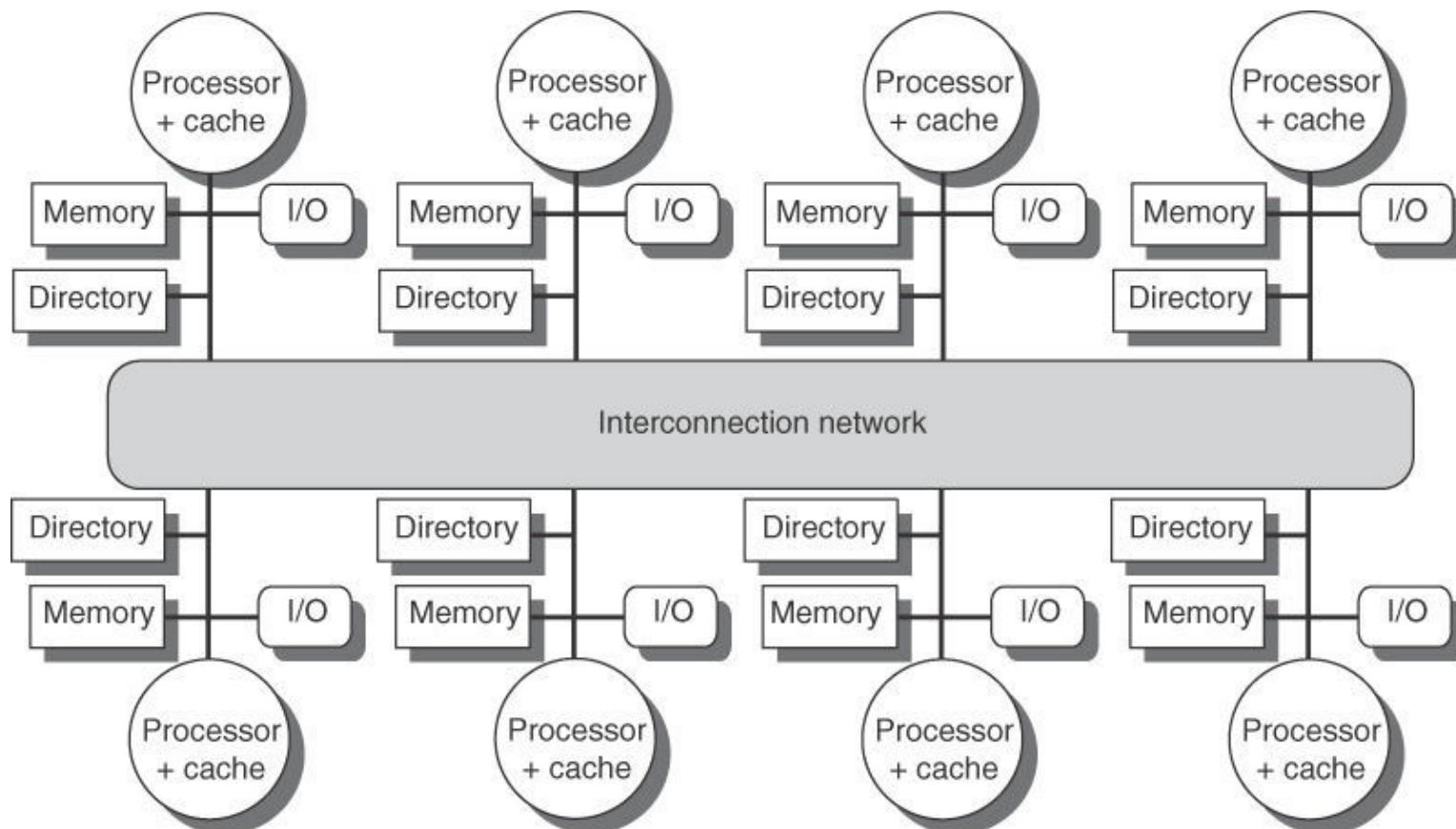
- Fixed number of pointers per directory entry regardless of the number of processors.
- Restricting the number of simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories. .

Limited Directory



Distributed Directory

- Distributed directory

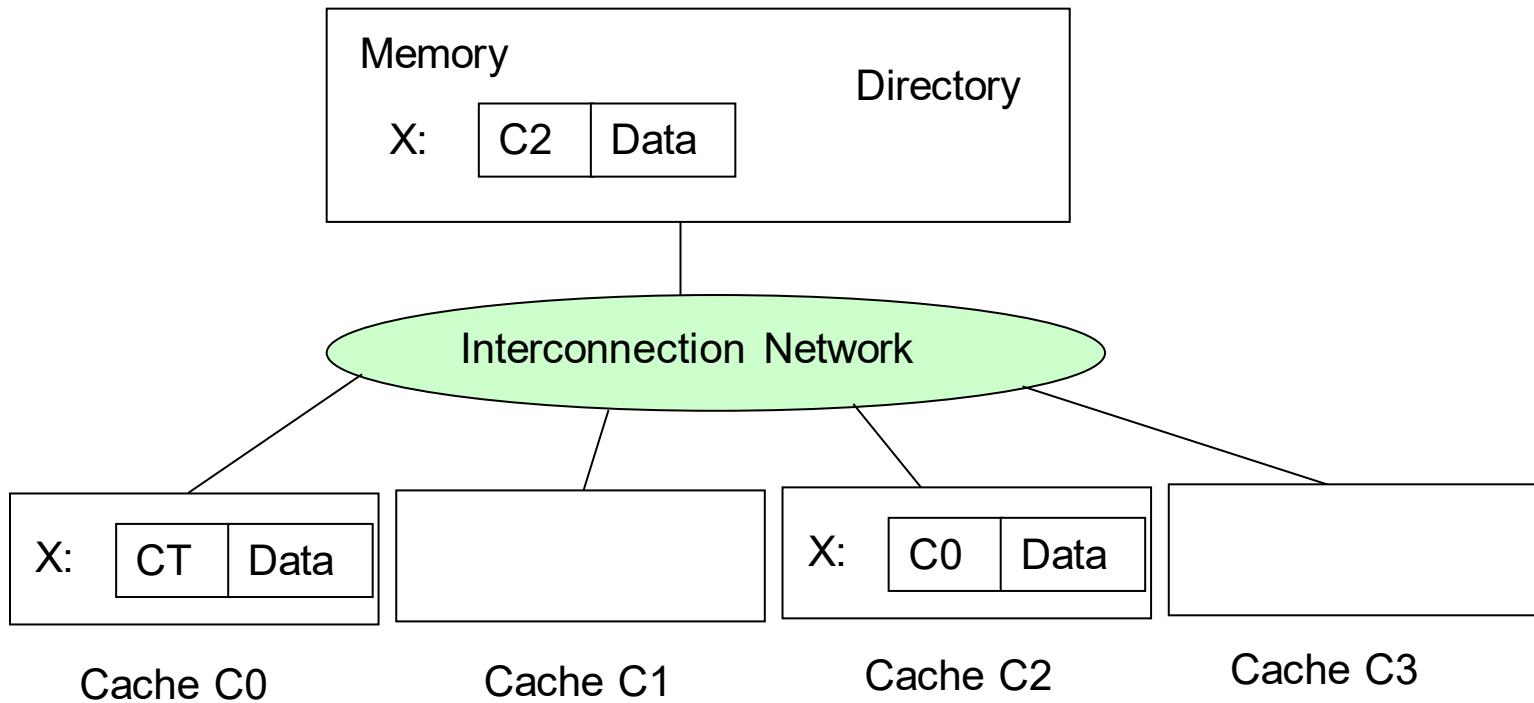


© 2007 Elsevier, Inc. All rights reserved.

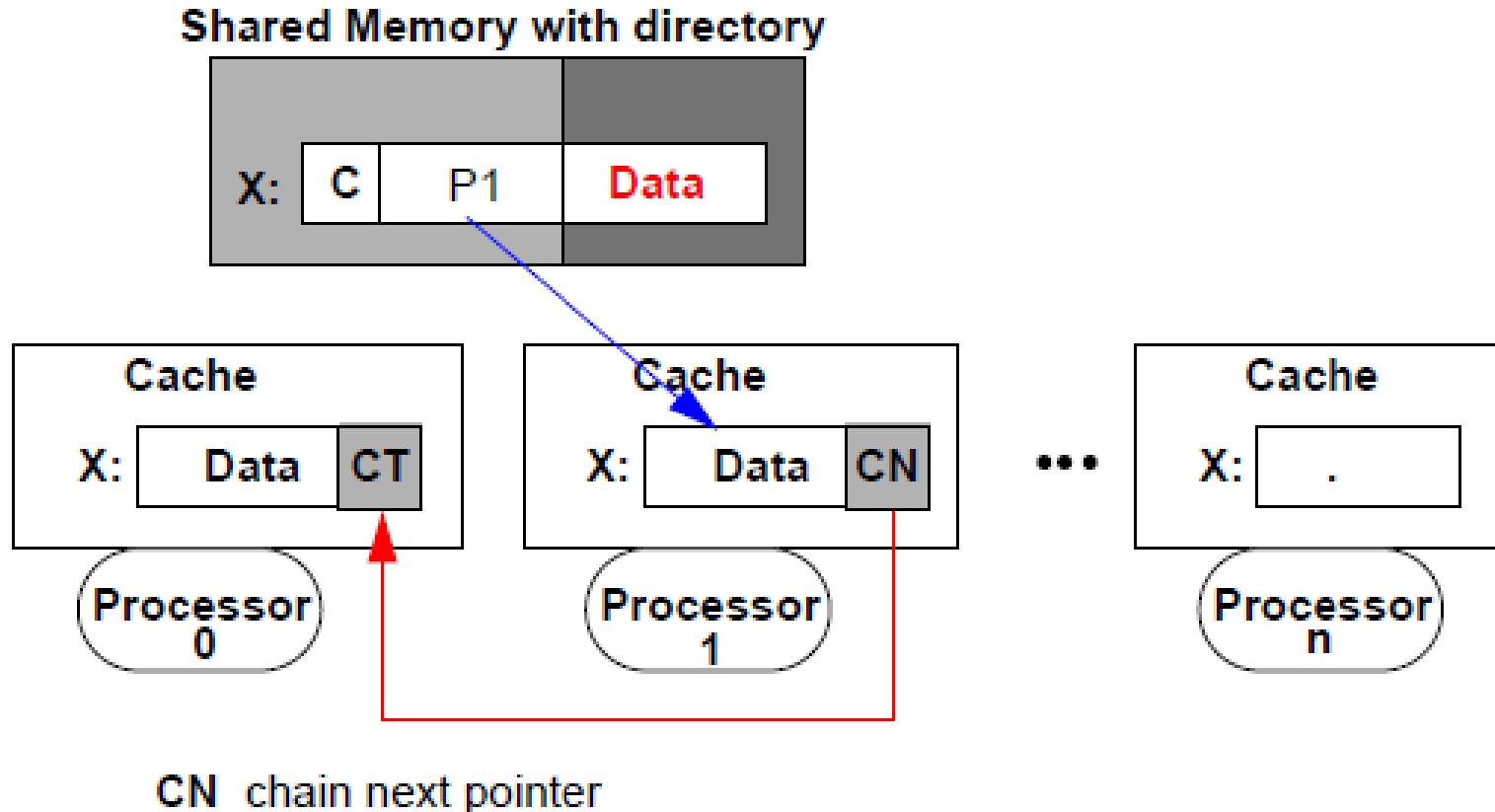
Chained Directory

- Chained directories emulate full-map by distributing the directory among the caches.
- Solving the directory size problem without restricting the number of shared block copies.
- Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers.

Chained Directory



Chained Directory



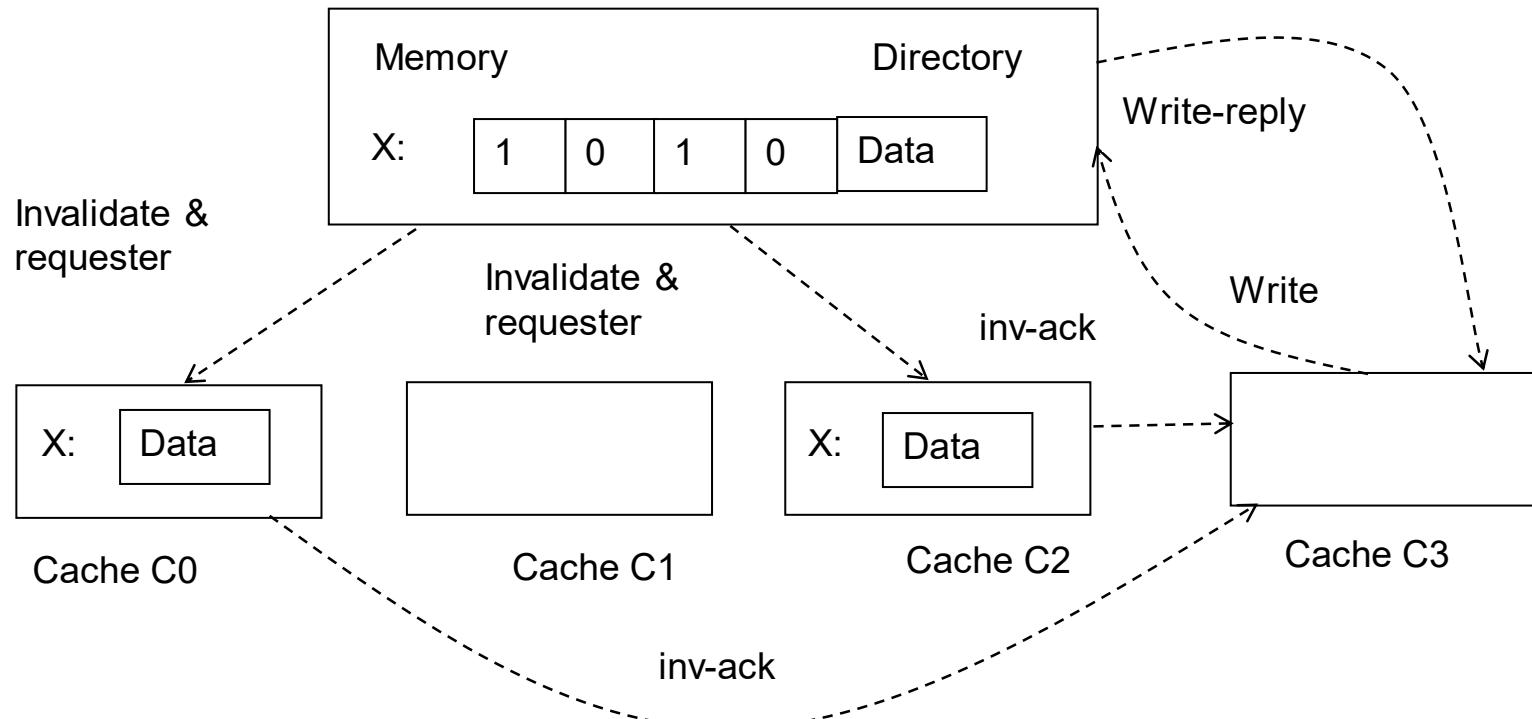
Invalidate

In invalidating signals and a pointer to the requesting processor are forwarded to all processors that have a copy of the block.

Each invalidated cache sends an acknowledgment to the requesting processor.

After the invalidation is complete, only the writing processor will have a cache with a copy of the block.

Write by P3



Scalable Coherent Interface (SCI)

Doubly linked list of distributed directories.

Each cached block is entered into a list of processors sharing that block.

For every block address, the memory and cache entries have additional tag bits. Part of the memory tag identifies the first processor in the sharing list (the head). Part of each cache tag identifies the previous and following sharing list entries.

Scenarios

Initially memory is in the uncached state and cached copies are invalid.

A read request is directed from a processor to the memory controller. The requested data is returned to the requester's cache and its entry state is changed from invalid to the head state.

This changes the memory state from uncached to cached.

SCI Scenarios (Cont.)

When a new requester directs its read request to memory, the memory returns a pointer to the head.

A cache-to-cache read request (called Prepend) is sent from the requester to the head cache.

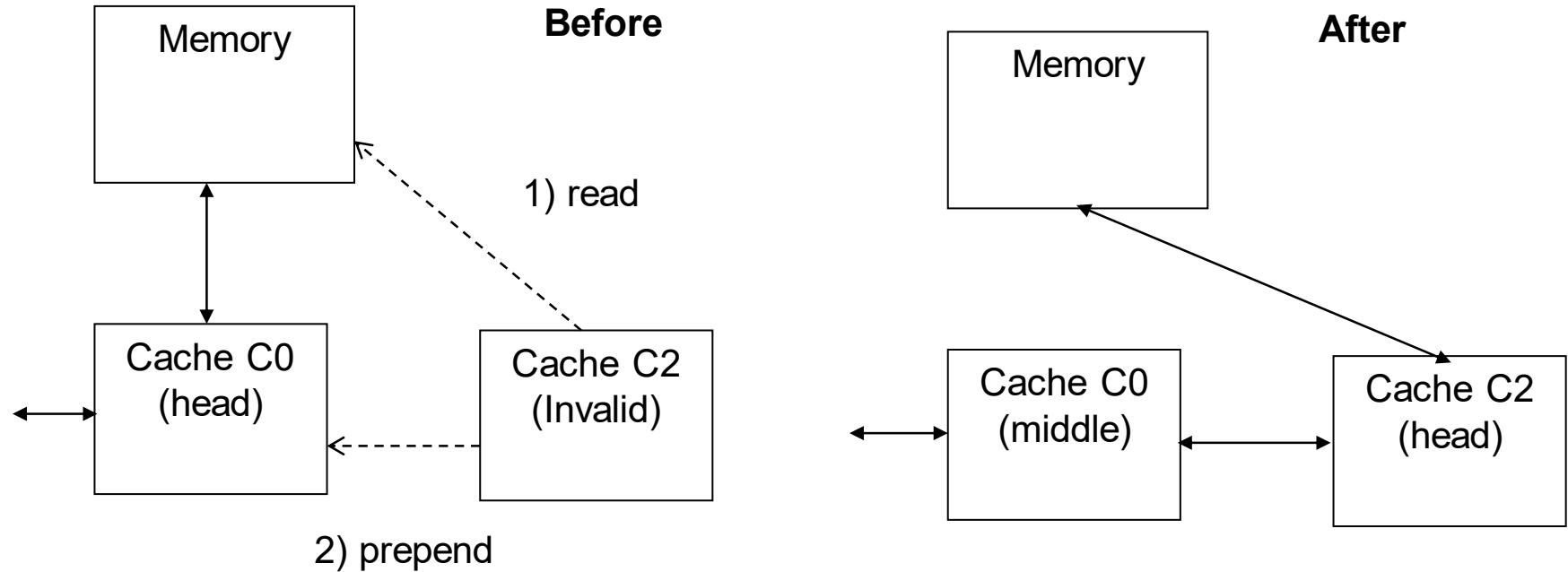
On receiving the request, the head cache sets its backward pointer to point to the requester's cache.

The requested data is returned to the requester's cache and its entry state is changed to the head state.

SCI – Sharing List

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

Addition



SCI Scenarios

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

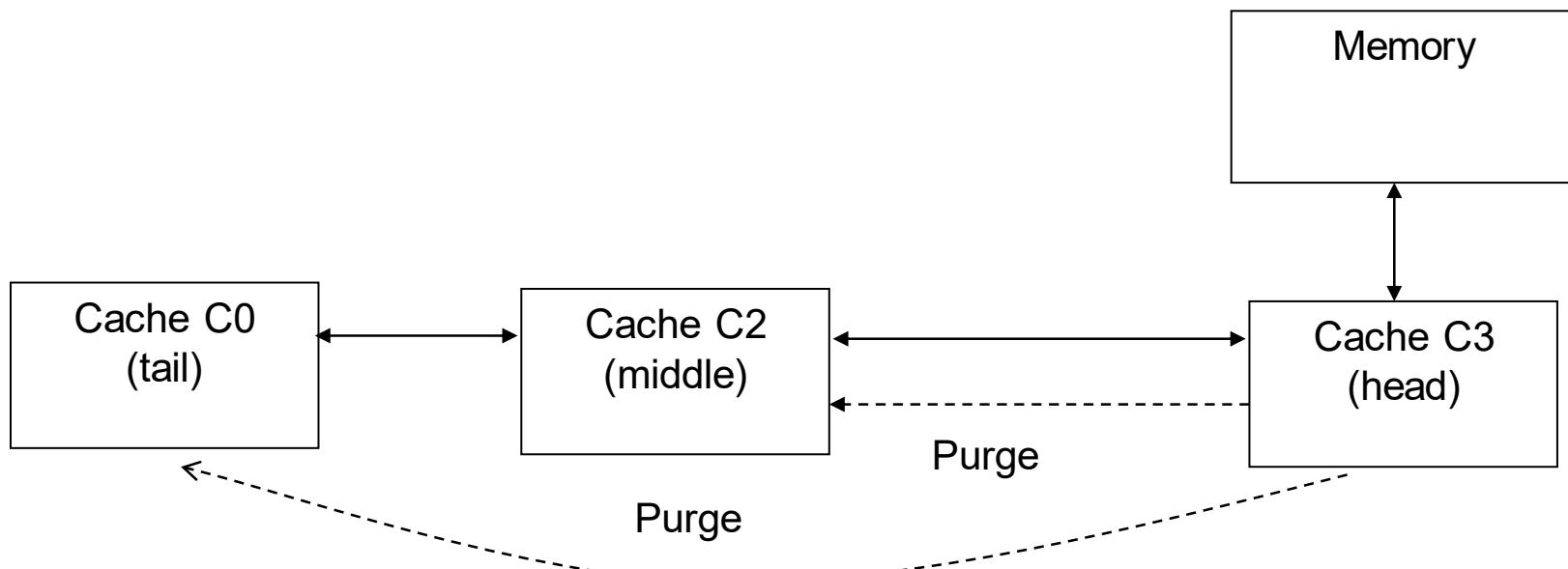
(Cont.)

The head of the list has the authority to purge other entries in the list to obtain an exclusive (read-write) entry.

SCI-- Head Purging

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide

Other Entries



Directory (SDD)

A singly linked list of distributed directories.

Similar to the SCI protocol, memory points to the head of the sharing list.

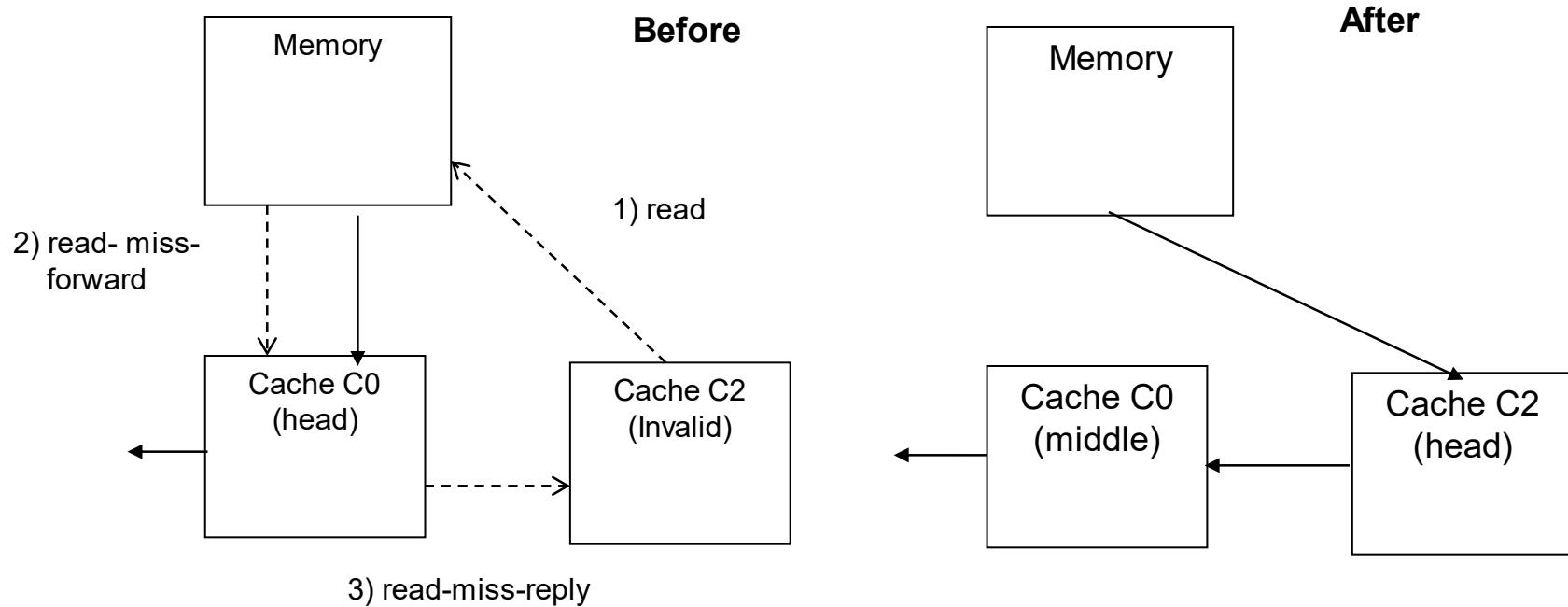
Each processor points only to its predecessor.

The sharing list additions and removals are handled different from the SCI protocol .

Scenarios

On a read miss, a new requester sends a read-miss message to memory. The memory updates its head pointers to point to the requester and send a read-miss-forward signal to the old head. On receiving the request, the old head returns the requested data along with its address as a read-miss-reply. When the reply is received, at the requester's cache, the data is copied and the pointer is made to point to the old head .

Addition



(cont.)

On a write miss, a requester sends a write-miss message to memory. The memory updates its head pointers to point to the requester and sends a write-miss-forward signal to the old head. The old head invalidates itself, returns the requested data as a write-miss-reply-data signal, and send a write-miss-forward to the next cache in the list.

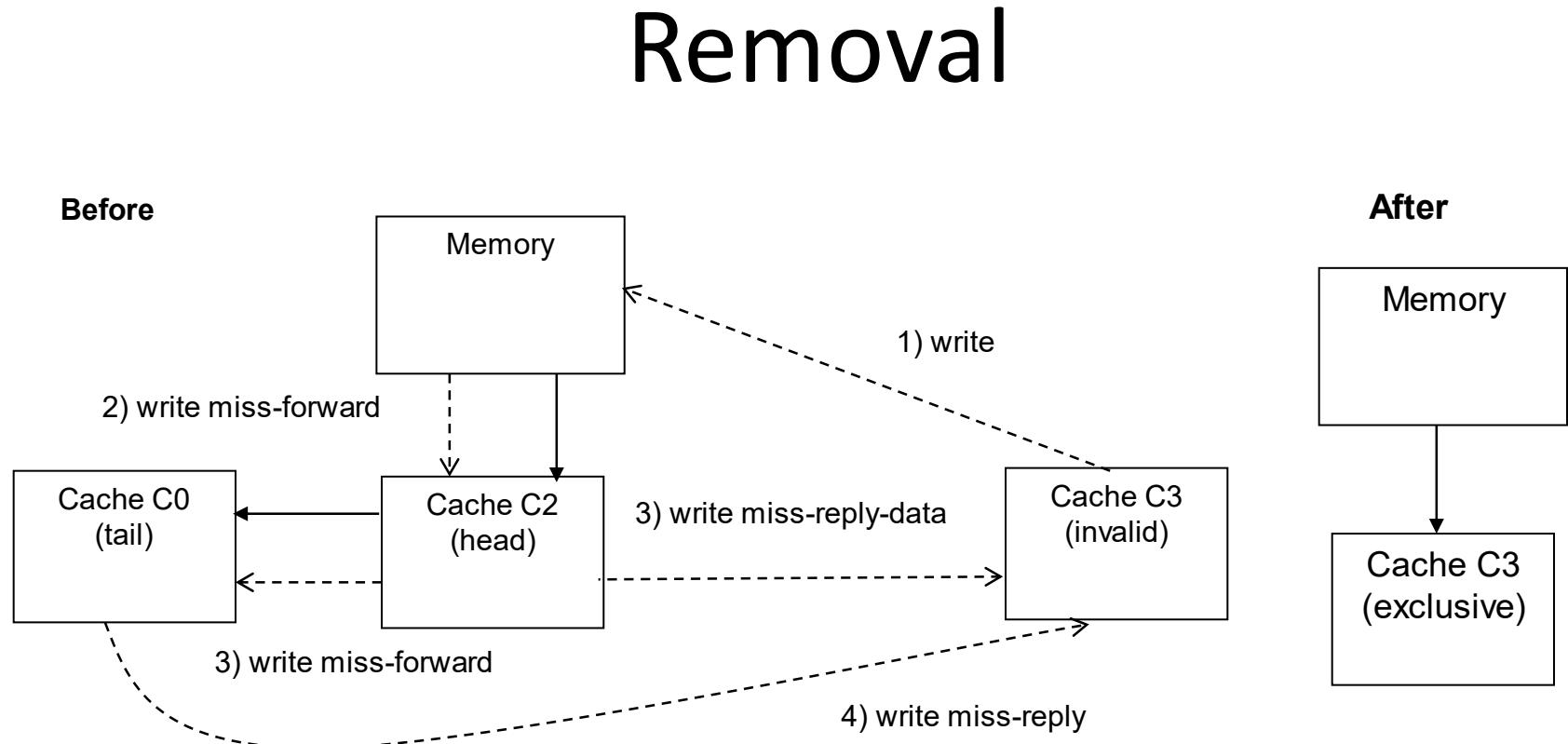
SDD Scenarios

(cont.)

When the next cache receives the write-miss-forward signal, it invalidates itself and sends a write-miss-forward to the next cache in the list. When the write-miss-forward signal is received by the tail or by a cache that no longer has copy of the block, a write-miss-reply is sent to the requester. The write is complete when the requester receives both write-miss-reply-data and write-miss-reply.

SDD- Write Miss List Removal

25-Memory Interleaving; Cache Coherence-10-Sep-2020Material_II_10-Sep-2020_Cache_coherence-good-slide



VIRTUAL MEMORY

LIJO V P
SCOPE
VIT, VELLORE

What is a Virtual memory?

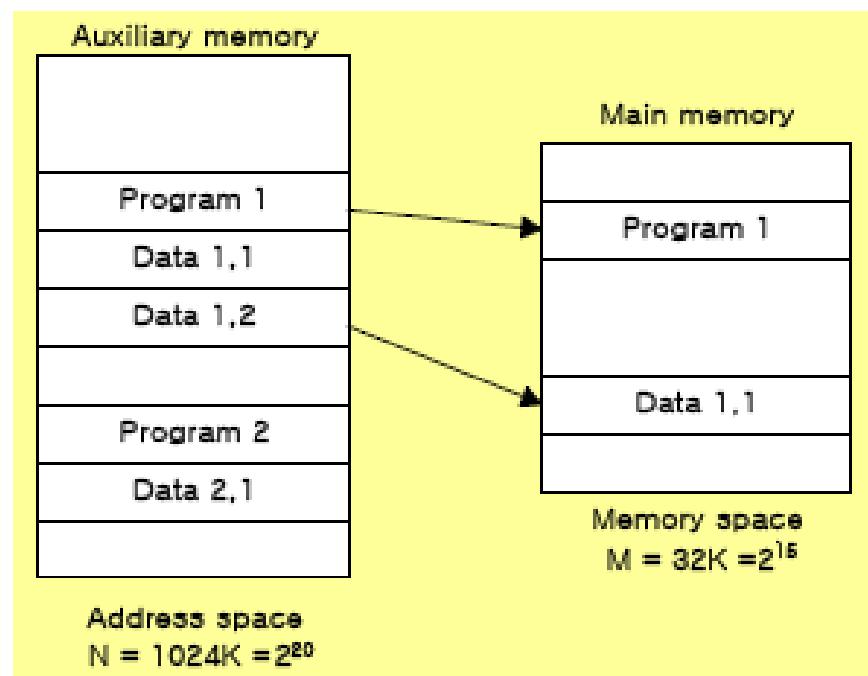
- Virtual memory permits the user to construct programs as though a large memory space were available.
- It gives the programmer **an illusion** that they have a very large memory.
- It provides a mechanism for translating program generated addresses into correct main memory locations by means of mapping table.
- Virtual memory architecture – combination of hardware mapping mechanism + memory management software

Some important terms

- **Virtual address** – address used by a programmer
- **Address space** - Set of virtual addresses
- **Physical address** – address in main memory
- **Memory space** – set of physical addresses
- Memory mapping - process of translating virtual addresses into physical addresses

Relation between address and memory space in virtual memory system

In multi-program system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU.



Paged Virtual Memory

- **Block/ Frame** - Organization of memory space
- **Page** – blocks of contiguous virtual memory addresses.
 - Organization of address space
- Pages are moved from auxiliary memory to main memory in records equal to size of a page (usually around 4 KB).
- A *page table* is a data structure used for translating virtual addresses (seen by application) to physical addresses (used by hardware) to process instructions.
- The hardware that handles this specific translation is called as **Memory Management Unit (MMU)**.

Page Table & Address Translation

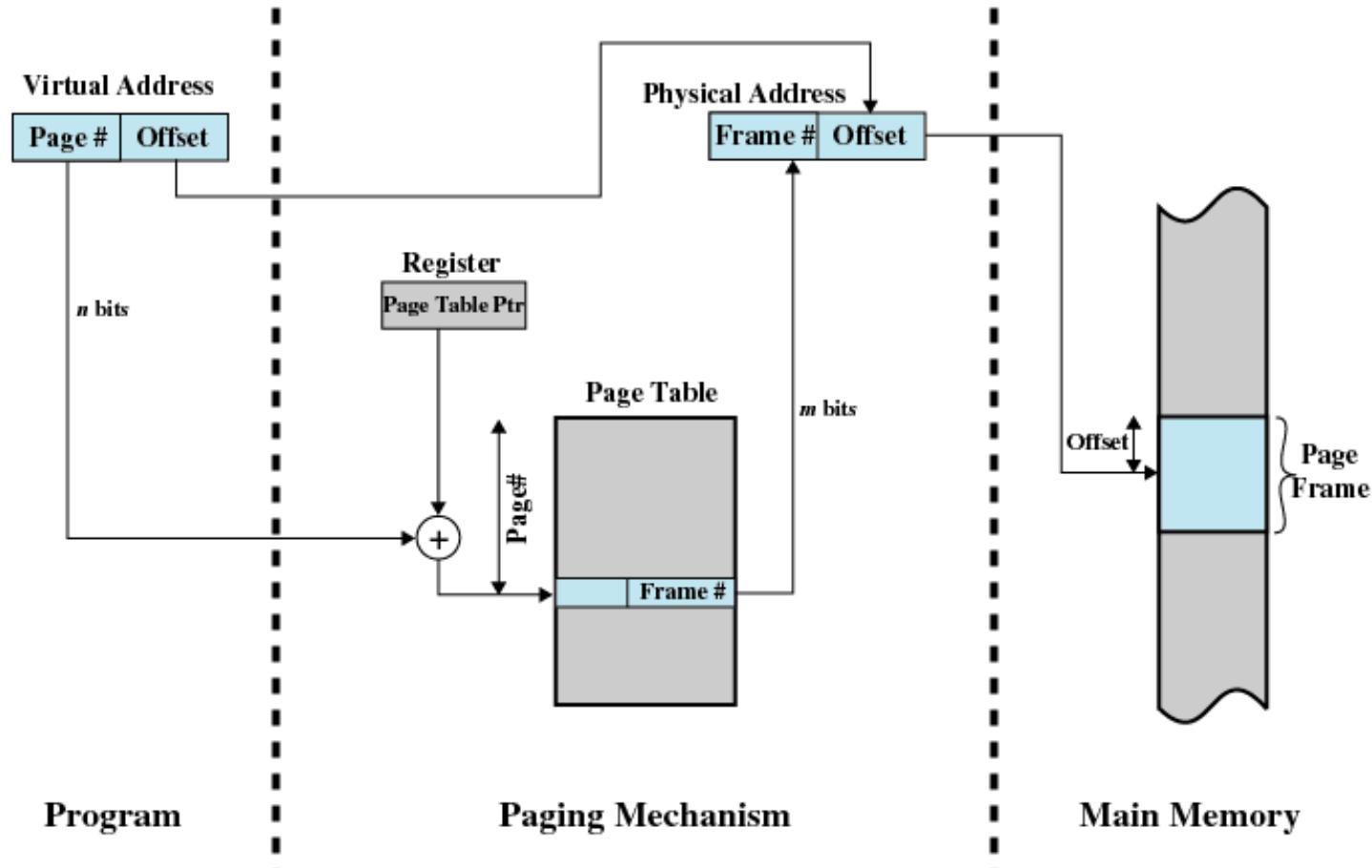


Figure 8.3 Address Translation in a Paging System

Translation Look-aside Buffer

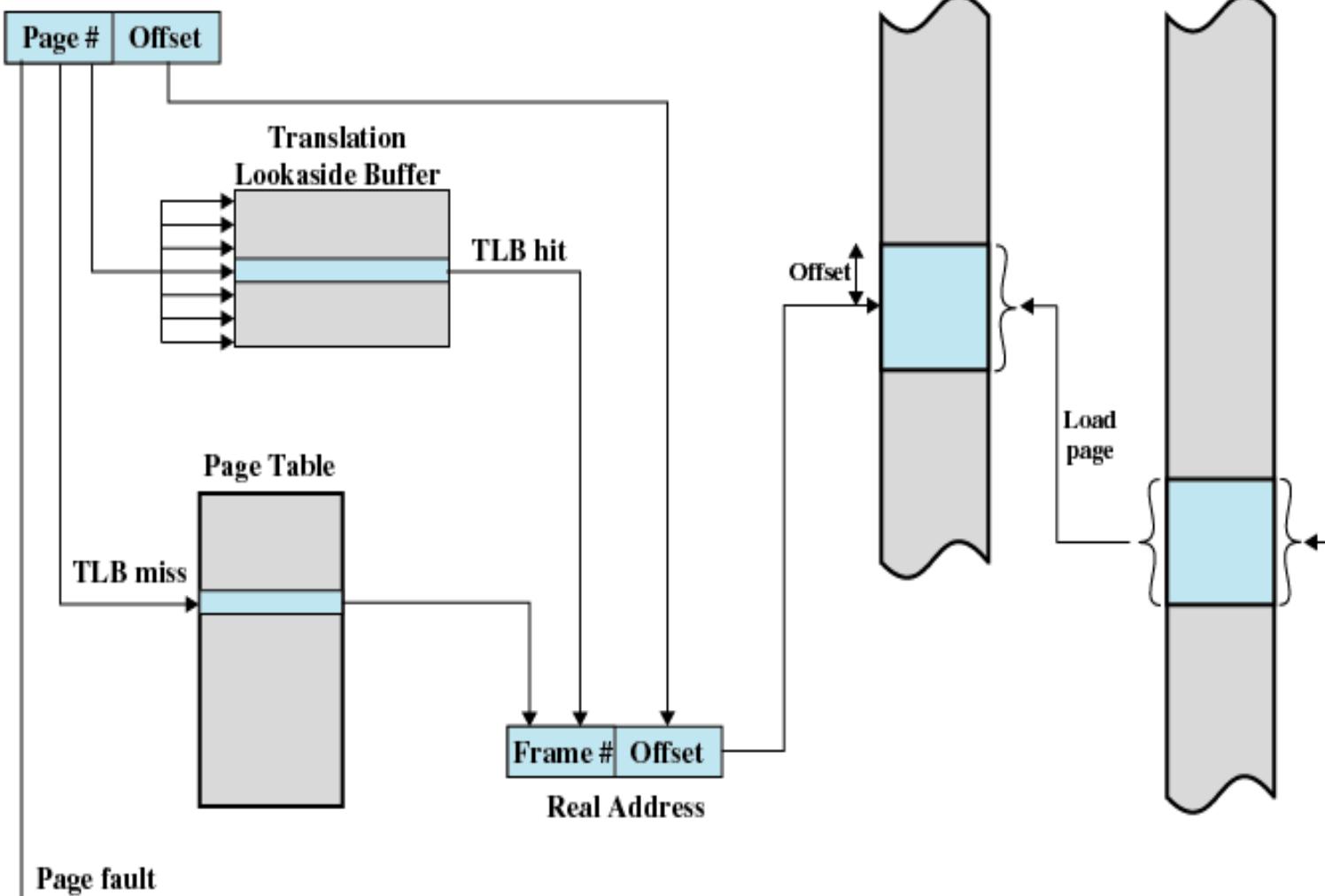
- Each virtual memory reference can cause **two physical memory accesses**
 - One to **fetch the page table**
 - One to **fetch the data**
- To overcome this problem a high-speed cache is set up for page table entries
 - Called a **Translation Look-aside Buffer (TLB)**
 - TLB Contains page table entries that have been most recently used

Translation Look-aside Buffer

- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table

TLB Continue.....

- First checks if page is already in main memory
 - If not in main memory a **page fault** is issued
- The TLB is updated to include the new page entry

Virtual Address**Figure 8.7 Use of a Translation Lookaside Buffer**

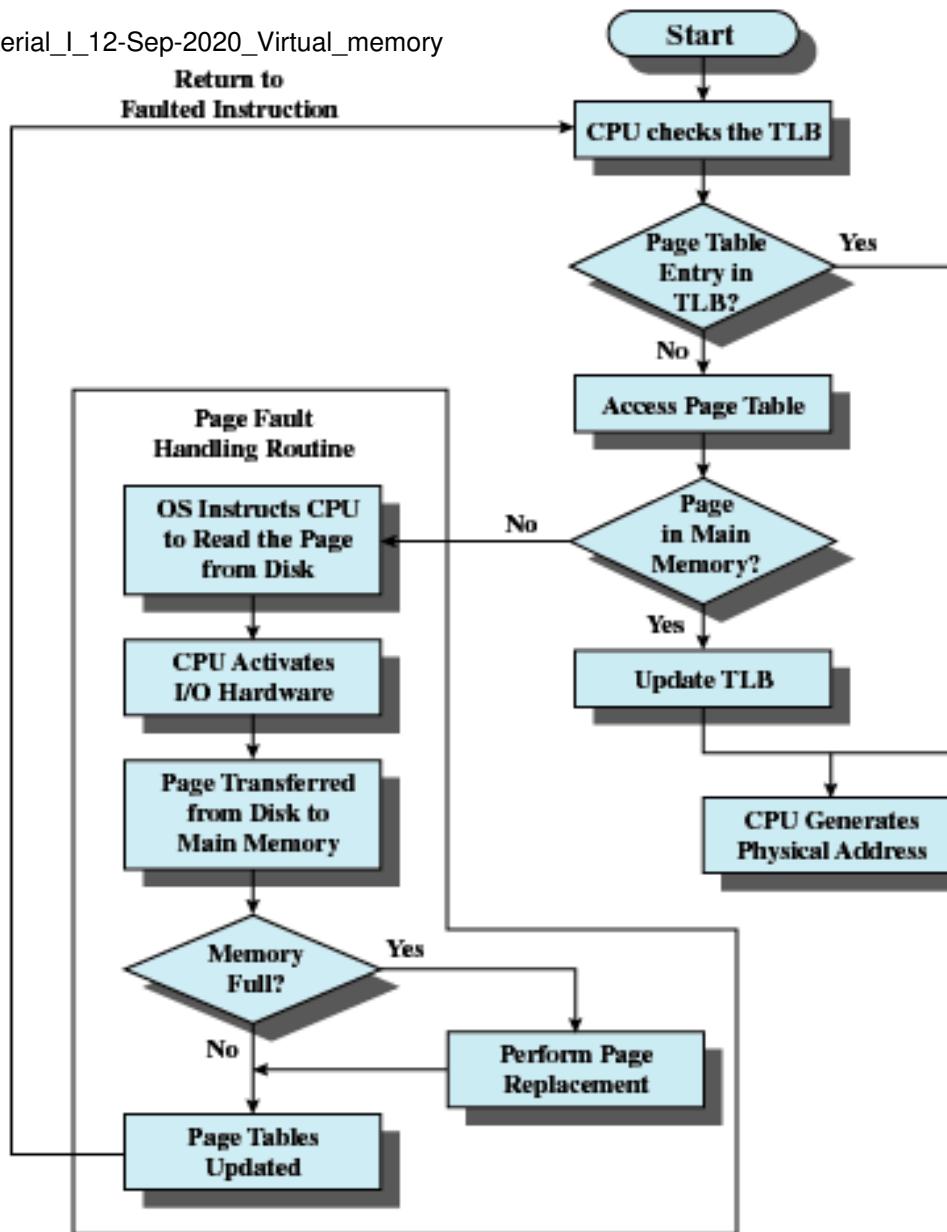


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

Segmentation

- May be **unequal, dynamic** size
- Simplifies handling of growing data structures
- Allows programs to be altered and recompiled independently

Segment Tables

- Corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

Segment Table Entries

Virtual Address



Segment Table Entry



(b) Segmentation only

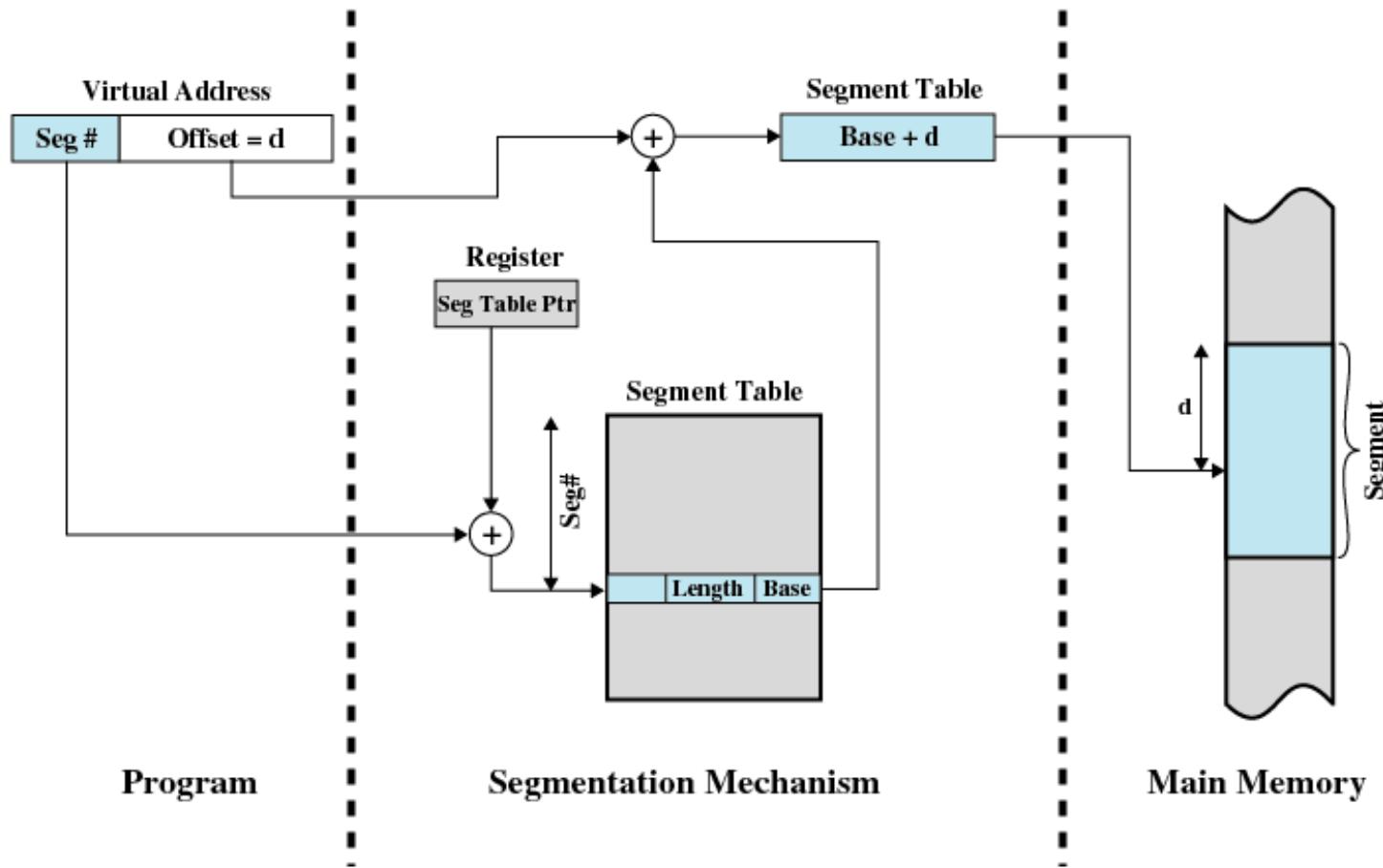


Figure 8.12 Address Translation in a Segmentation System

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

Combined Segmentation and Paging

Virtual Address



Segment Table Entry



Page Table Entry



P= present bit
M = Modified bit

(c) Combined segmentation and paging

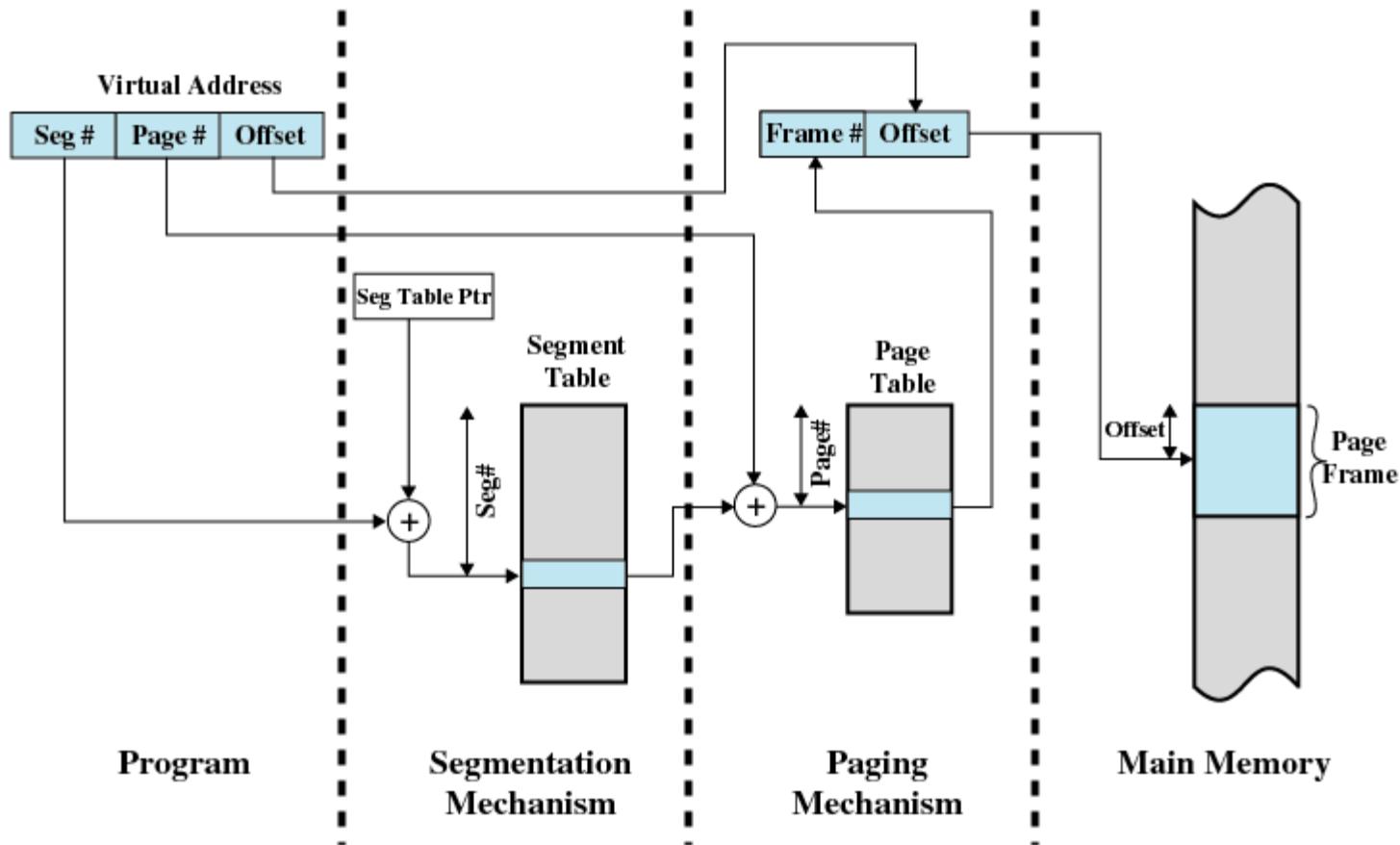


Figure 8.13 Address Translation in a Segmentation/Paging System

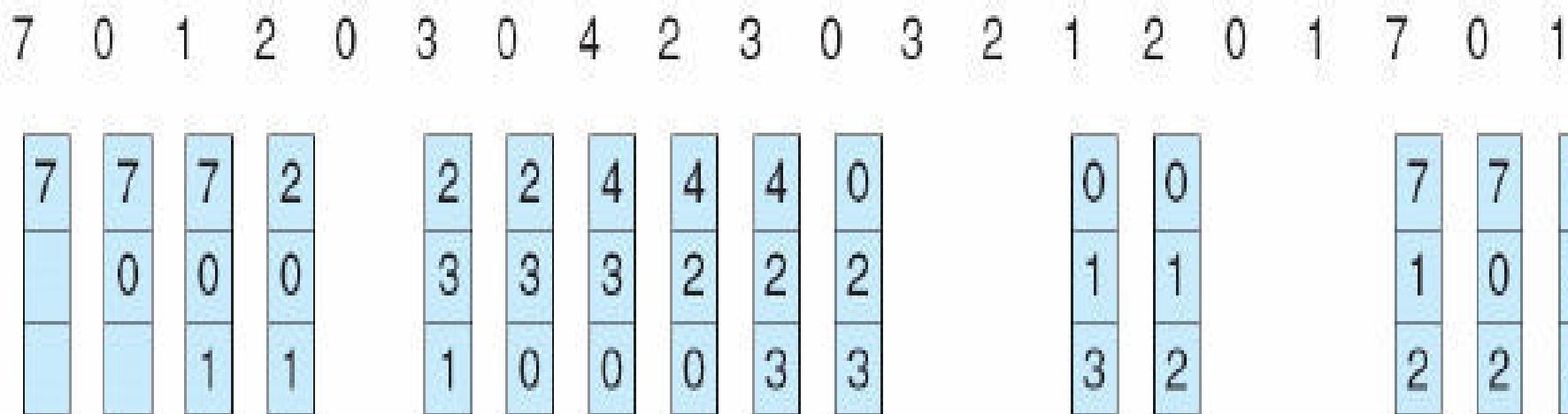
Page replacement Algorithms

- Which page need to be replaced when page fault occurs?
- Page Replacement algorithms
 - FIFO (First-In First-Out)
 - Optimal page replacement
 - LRU (Least Recently Used)

FIFO

- When a page must be replaced, the oldest page is chosen

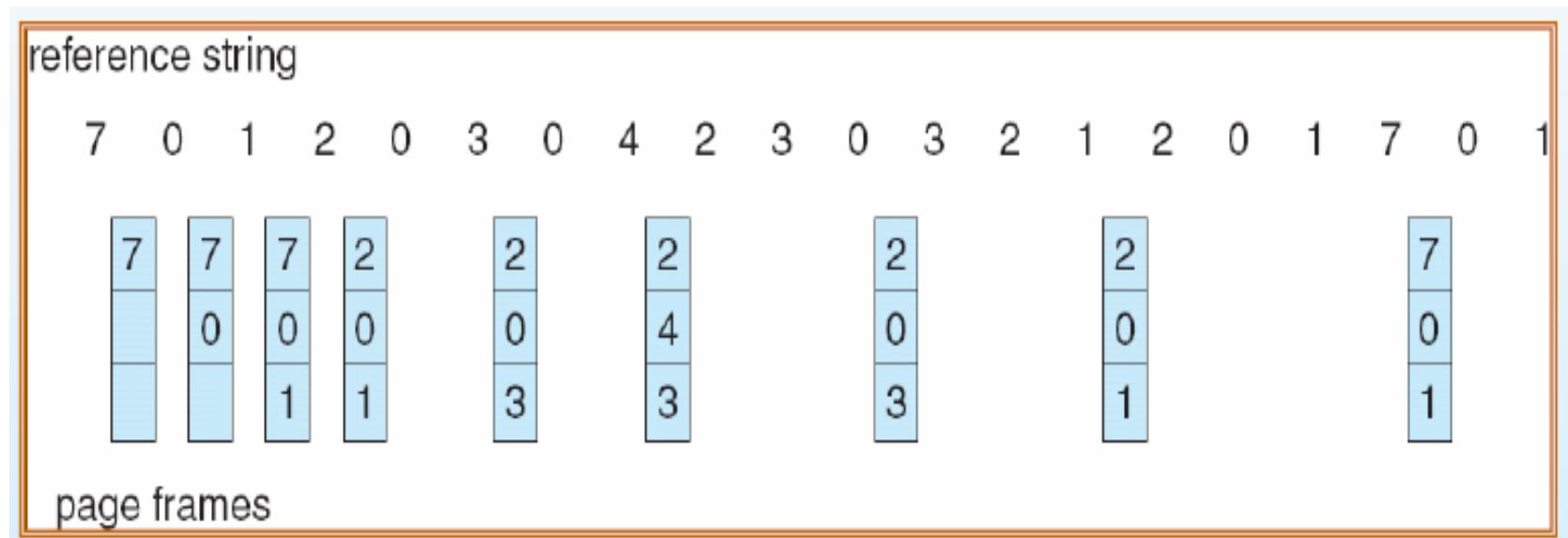
reference string



page frames

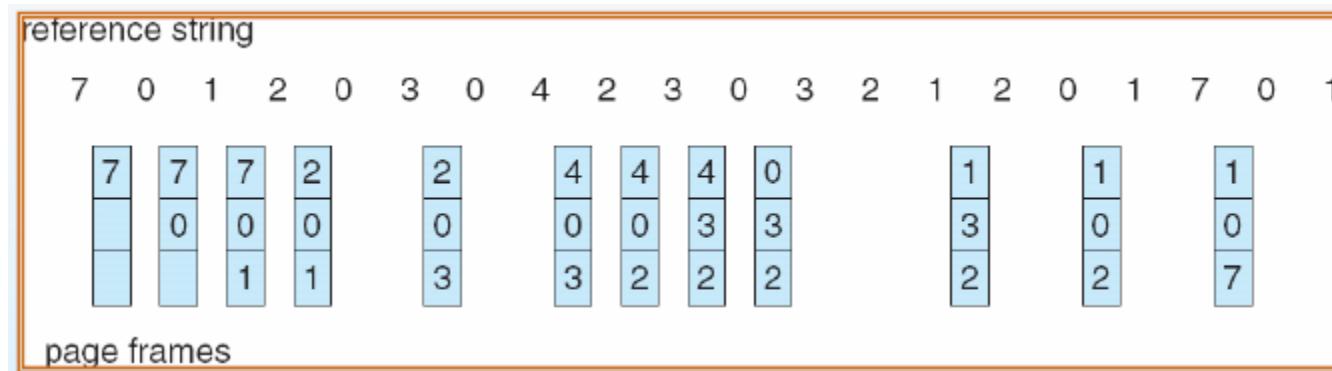
Optimal Page-Replacement Algorithm

Replace the page that won't be needed for the longest time in the future



Least-recently-used (LRU) algorithm

Replace the page that hasn't been referenced for the longest time



Problem 1

A virtual memory system has an address space of 8K words, a memory space of 4K words and page and block sizes of 1K words. The following page reference changes occur during a given time interval.

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm is (a) FIFO (b) LRU (c) Optimal page replacement algorithm. Compare the performance of the algorithms using the page hit ratio.

4 3 2 0 1 2 0 4 3 5 2 0 1 2 3

Problem 2

The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory:

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

Determine the four pages that are resident in main memory after each page reference. Calculate the hit ratio considering LRU as the replacement policy adopted. Identify the type of miss when occurred and total number of misses in each case.

Thank You

I/O FUNDAMENTALS

EXAMPLES OF I/O DEVICES

A personal computer

- Keyboard
- Mouse
- Joystick
- Keypad
- Scanner
- Monitor
- Printer

I/O INTERFACE

Resolves the *differences* between the computer and peripheral devices

Peripherals - Electromechanical Devices

CPU or Memory - Electronic Device

- Data Transfer Rate

Peripherals - Usually slower

CPU or Memory - Usually faster than peripherals

Some kinds of Synchronization mechanism may be needed

- Unit of Information

Peripherals - Byte

CPU or Memory - Word

- Operating Modes

Peripherals - Autonomous, Asynchronous

CPU or Memory - Synchronous

I/O BUS AND MEMORY BUS

Functions of Buses

MEMORY BUS is for information transfers between CPU and the Main Memory

- * *I/O BUS* is for information transfers between CPU and I/O devices through their I/O interface
- * Many computers use a common single bus system for both memory and I/O interface units
 - Use one common bus but separate control lines for each function
 - Use one common bus with common control lines for both functions
- * Some computer systems use two separate buses,
one to communicate with memory and the other with I/O interfaces

ISOLATED I/O

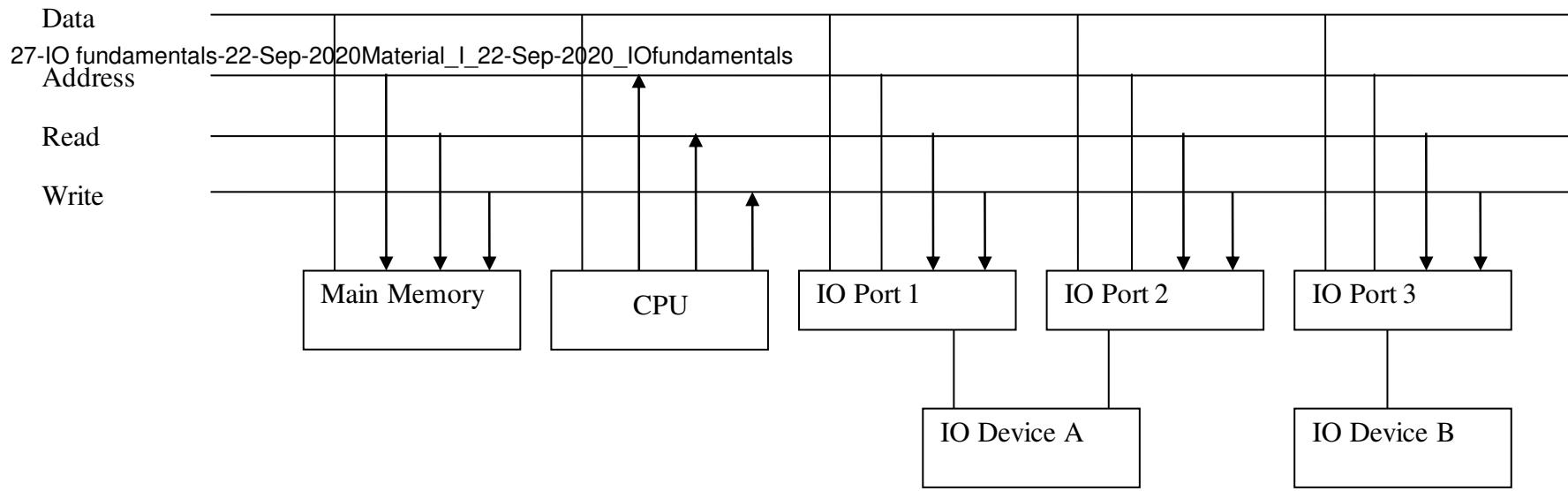
Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions
- When CPU fetches and decodes the opcode of an I/O instruction, it places the address into the common address lines.
- Also enables read/write control lines => the address in the address lines is for interface register and not for a memory word.

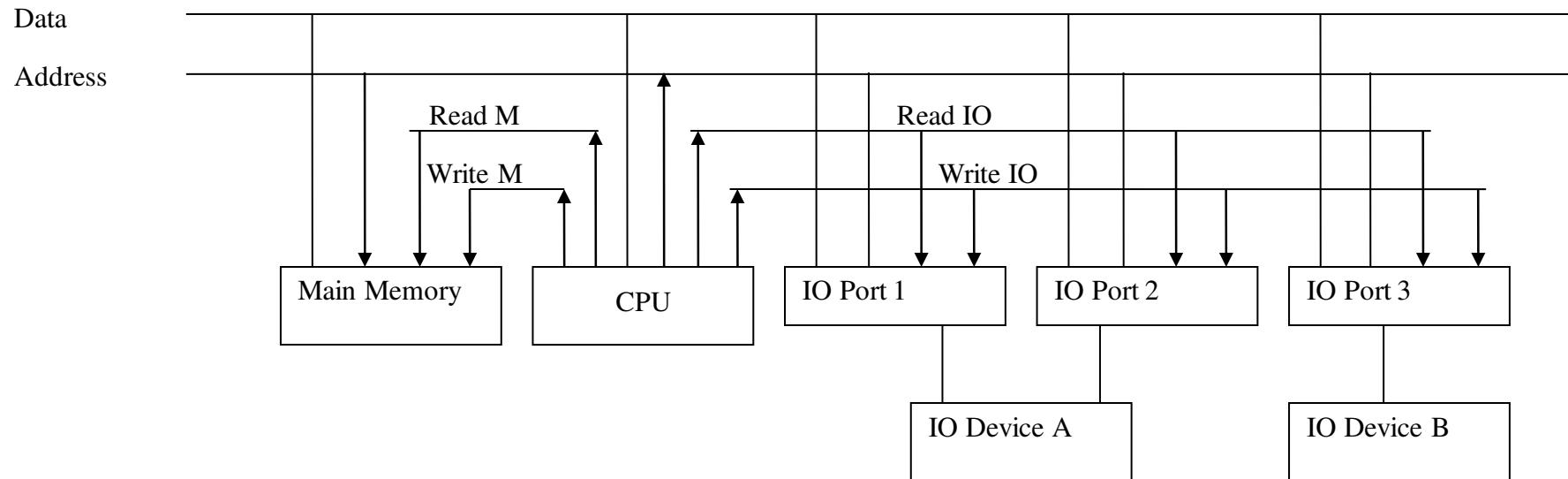
MEMORY MAPPED I/O

Memory mapped I/O

- A single set of read/write control lines
 - (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space
 - > reduces memory address range available
 - > Memory addresses xffff0000 and above are used for I/O devices
- No specific input or output instruction
 - > The same memory reference instructions can be used for I/O transfers
 - > when the bus sees certain addresses, it knows they are not memory addresses, but are addresses for accessing I/O devices.
- Considerable flexibility in handling I/O operations



Memory-Mapped I/O



I/O Mapped I/O

ASYNCHRONOUS DATA TRANSFER

Synchronous and asynchronous operations

line Synchronous - All devices derive the timing information from common clock

Asynchronous - No common clock

Asynchronous Data Transfer

Asynchronous data transfer between two independent units requires that *control signals* be transmitted between the communicating units *to indicate the time at which data is being transmitted*

Two Asynchronous Data Transfer Methods

Strobe pulse

- A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur

Handshaking

- A control signal is accompanied with each data being transmitted to indicate the presence of data

- The receiving unit responds with another control signal to acknowledge receipt of the data

STROBE CONTROL

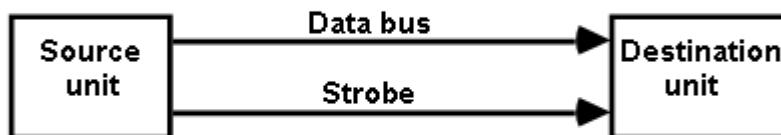
Employs a single control line to time each transfer

The strobe may be activated by either the source or the destination unit

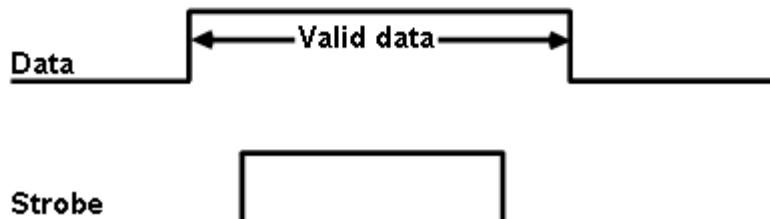
Data bus carries the binary information from source to destination

Source-Initiated Strobe for Data Transfer

Block Diagram

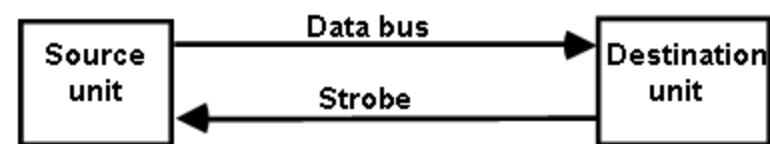


Timing Diagram

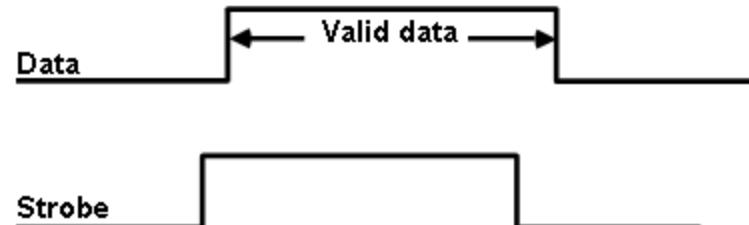


Destination-Initiated Strobe for Data Transfer

Block Diagram



Timing Diagram



SOURCE INITIATED STROBE

It could be a memory write control signal from the CPU to a memory unit.

Source is the CPU, places a word on the data bus and informs the memory unit which is destination, that this is a write operation

Disadvantage – no way of knowing whether the destination unit has actually received data

DESTINATION INITIATED STROBE

It could be a memory read control signal.

CPU, the destination initiates the read operation to inform the memory which is the source to place a selected word into the data bus.

Disadvantage - no way of knowing whether the source has actually placed the data on the bus

HAND SHAKING

To solve the disadvantages of strobe, the *HANDSHAKE* method introduces a second control signal to provide a *Reply* to the unit that initiates the transfer

2 types

- Source initiated transfer
- Destination initiated transfer

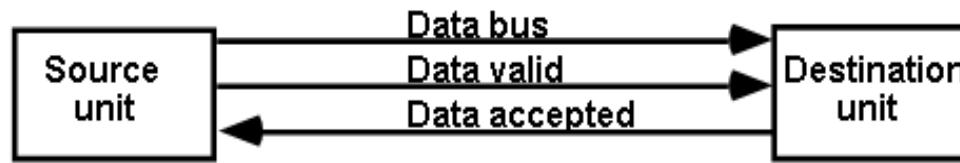
Provides high degree of flexibility and reliability

Incompletion of data transfer can be detected by means of a timeout mechanism

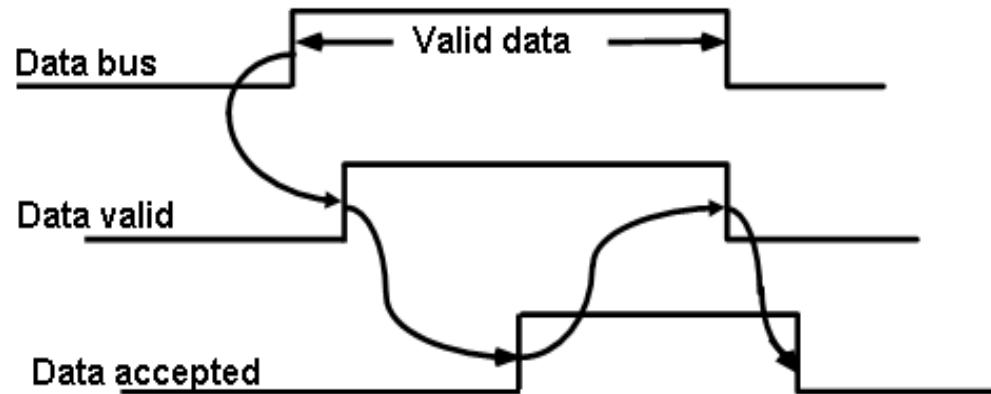
The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.

SOURCE-INITIATED TRANSFER USING HANDSHAKE

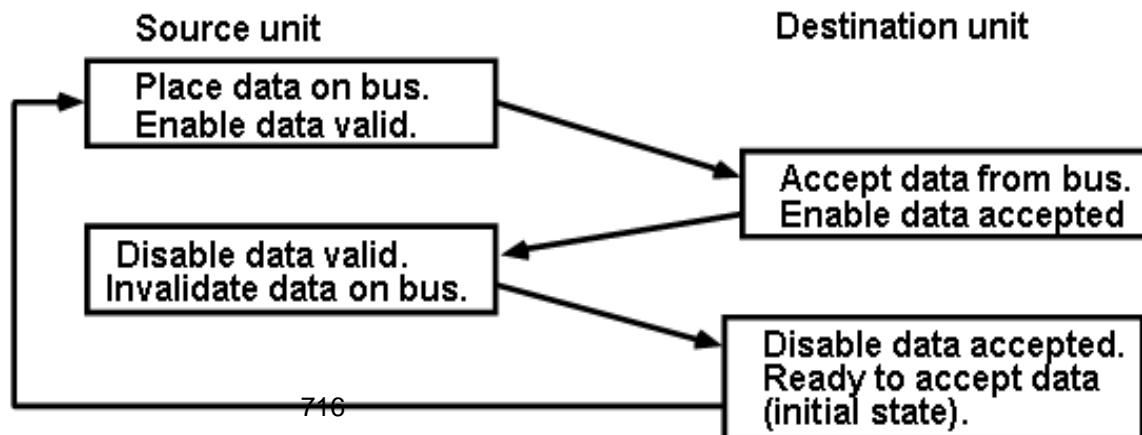
Block Diagram



Timing Diagram

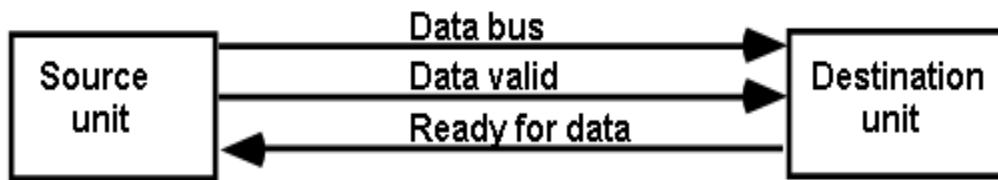


Sequence of Events

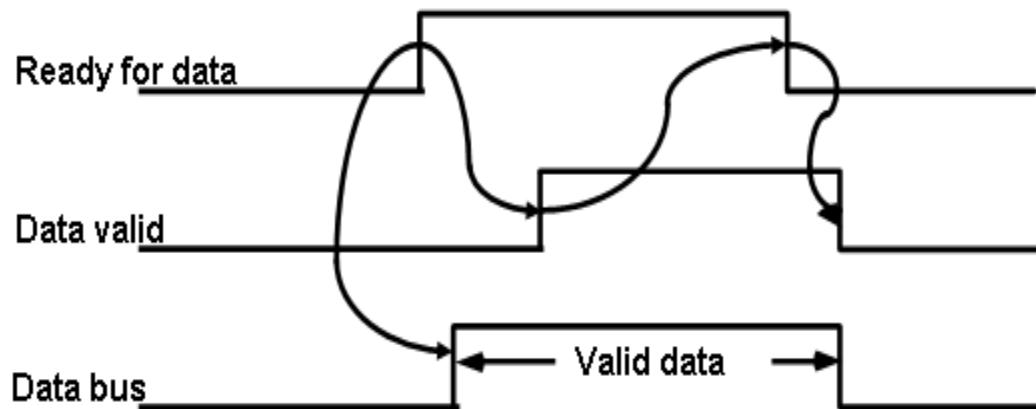


DESTINATION-INITIATED TRANSFER USING HANDSHAKE

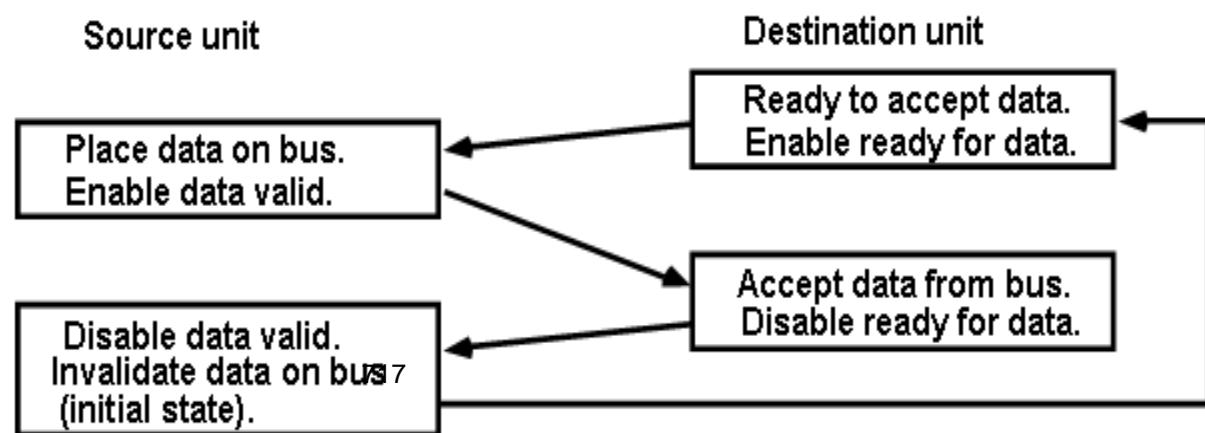
Block Diagram



Timing Diagram



Sequence of Events



REFERENCES

M. M. Mano, Computer System Architecture, Prentice-Hall

I/O techniques: programmed I/O, interrupt-driven I/O, DMA

Lijo V P
SCOPE
VIT, VELLORE

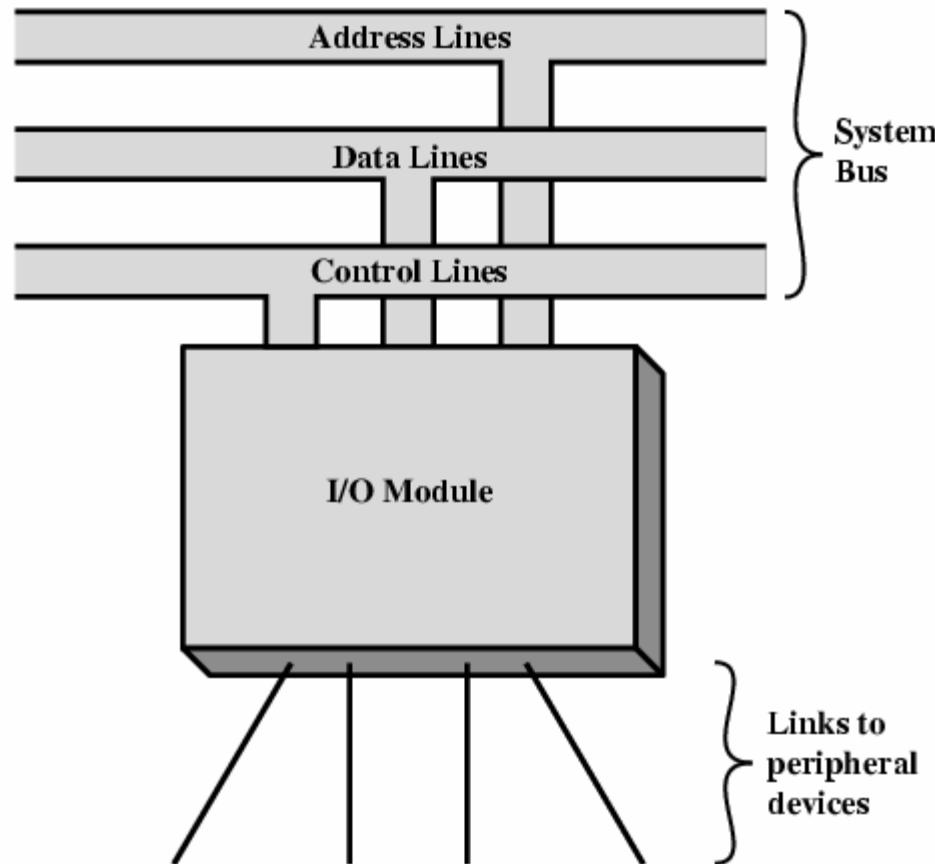
Input/Output Problems

- Wide variety of peripherals
- Delivering different amounts of data
- At different speeds
- In different formats
- All slower than CPU and RAM
- Need I/O modules

Input/Output Module

- Interface to CPU and Memory
- Interface to one or more peripherals

Generic Model of I/O Module



I/O Module Function

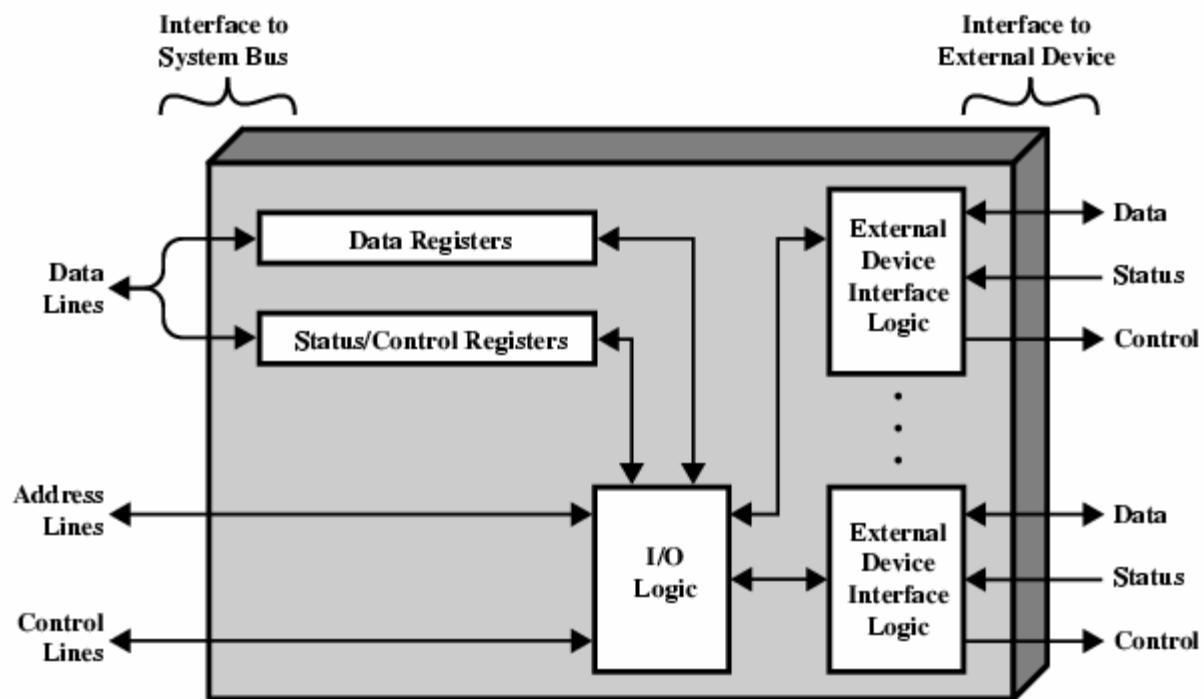
- Control & Timing
- CPU Communication
- Device Communication
- Data Buffering
- Error Detection

I/O Steps

Steps needed to transfer data to or from external device to CPU:

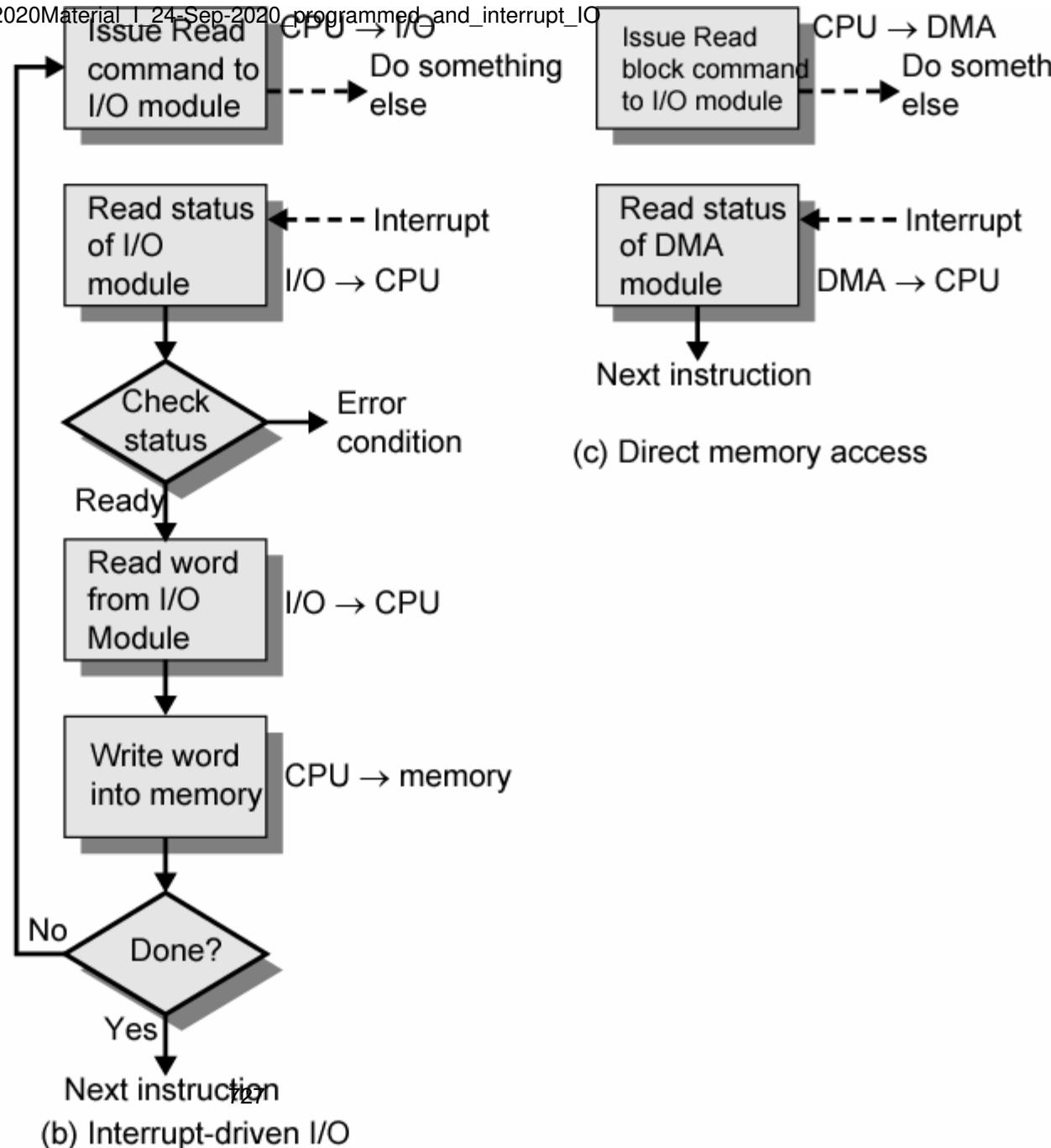
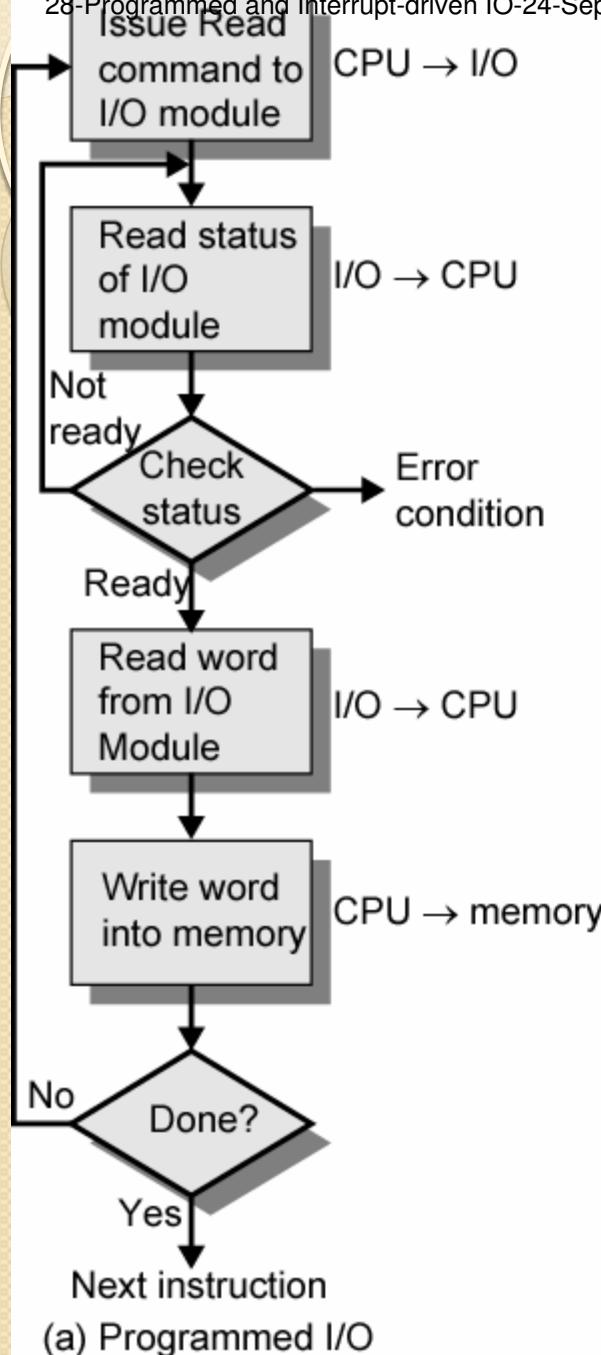
1. CPU checks I/O module device status
2. I/O module returns status
3. If ready, CPU requests data transfer
4. I/O module gets data from device
5. I/O module transfers data to CPU

I/O Module Diagram



Input Output Techniques/ mode of transfer

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)



Programmed I/O

- CPU has direct control over I/O
 - Sensing status
 - Read/write commands
 - Transferring data
- CPU waits for I/O module to complete operation
- Wastes CPU time

Programmed I/O - detail

- CPU requests I/O operation
- I/O module performs operation and sets status bits after completion
- CPU checks status bits periodically.
- I/O module does not inform CPU directly that is it does not interrupt CPU
- CPU must wait

Interrupt Driven I/O

- Overcomes CPU waiting
- No repeated CPU checking of device
- I/O module interrupts when ready

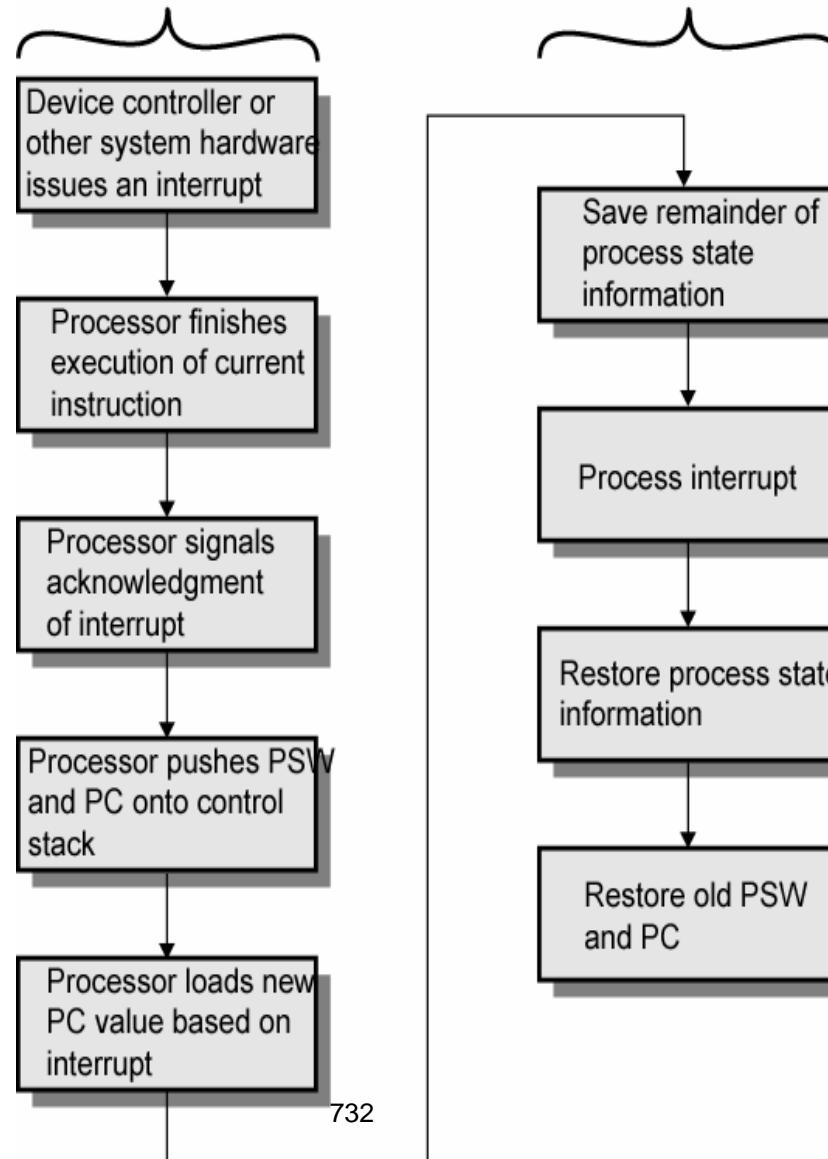
Interrupt Driven I/O Basic Operation

- CPU issues read command
- I/O module gets data from peripheral while CPU does other work
- I/O module interrupts CPU after completion
- CPU requests data
- I/O module transfers data

Simple Interrupt Processing

Hardware

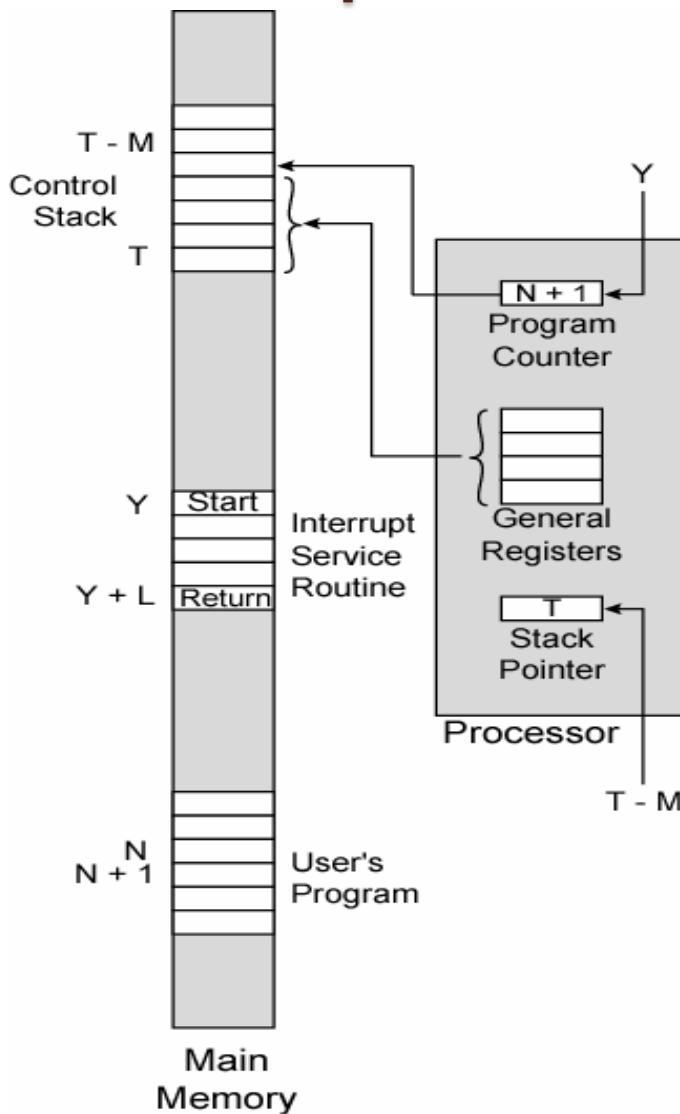
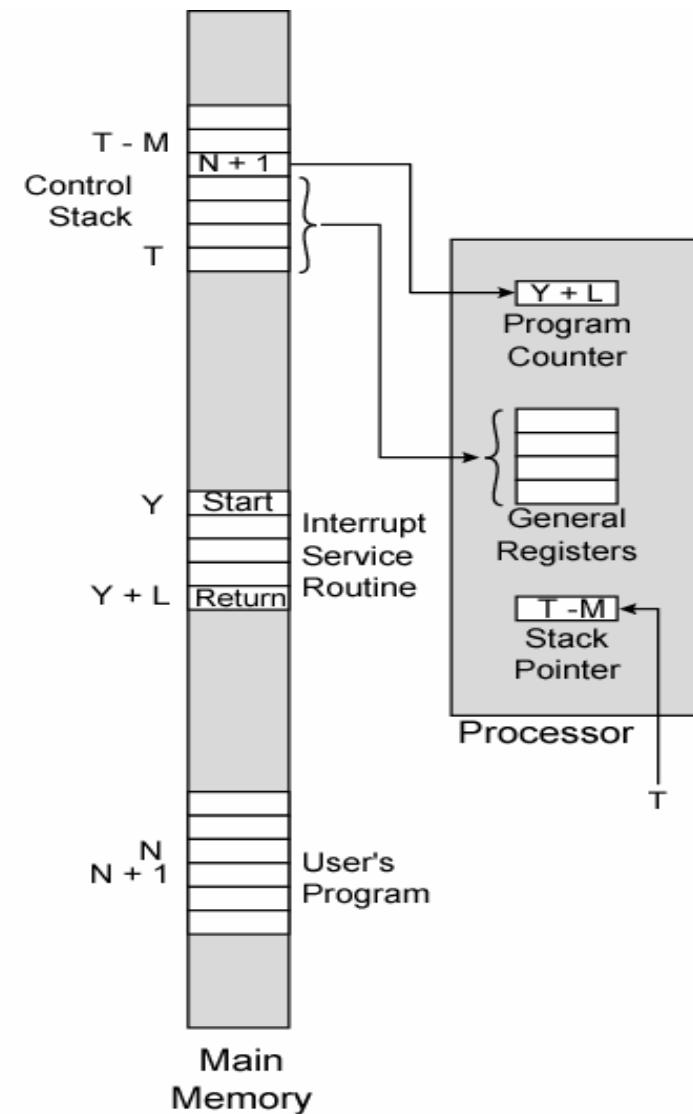
Software



CPU Viewpoint

- Issue read command
- Do other work
- Check for interrupt at end of each instruction cycle
- If interrupted:-
 - – Save context (registers)
 - – Process interrupt
- Fetch data & store

Changes in Memory and Registers for an Interrupt

(a) Interrupt occurs after instruction at location N 

(b) Return from interrupt

I/O fundamentals

PERIPHERAL DEVICES

Input Devices

- Keyboard
- Optical input devices
 - Card Reader
 - Paper Tape Reader
 - Bar code reader
 - Digitizer
 - Optical Mark Reader
- Magnetic Input Devices
 - Magnetic Stripe Reader
- Screen Input Devices
 - Touch Screen
 - Light Pen
 - Mouse
- Analog Input Devices

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Plotter
- Analog
- Voice

Introduction

- The I/O subsystem provides an efficient mode of communication between CPU and outside environment
- Devices that are under the direct control of computer are said to be connected on-line
- Input or output devices attached to the computer are also referred as peripherals.
- I/O interface or I/O Module provides a method for transferring information between internal storage and external i/o devices.

I/O interface

Resolves the *differences* between the computer and peripheral devices

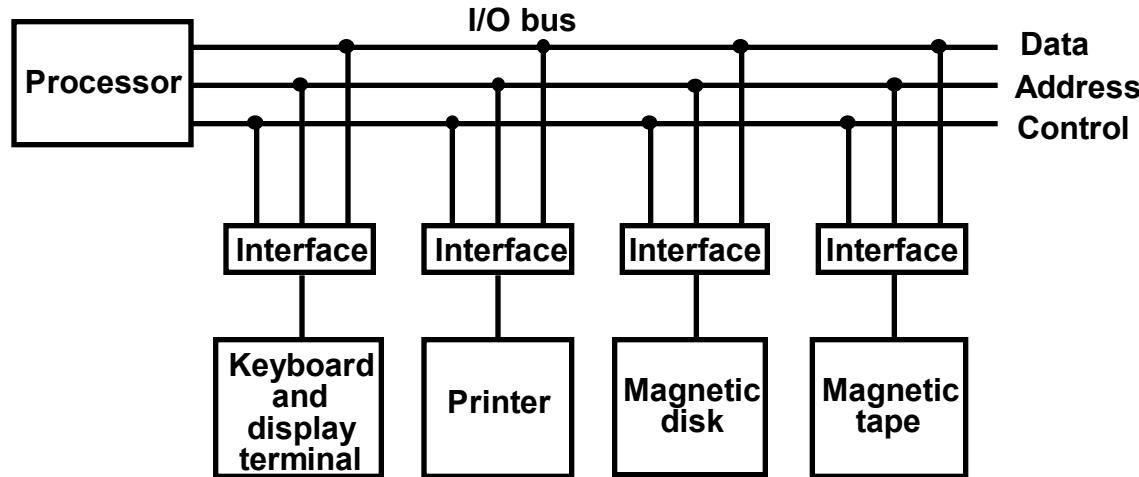
- **Peripherals - Electromechanical Devices**
- **CPU or Memory - Electronic Device**

- **- Data Transfer Rate**
 - **Peripherals - Usually slower**
 - **CPU or Memory - Usually faster than peripherals**
- **Some kinds of Synchronization mechanism may be needed**

- **- Unit of Information**
 - **Peripherals - Byte**
 - **CPU or Memory - Word**

- **- Operating Modes**
 - **Peripherals - Autonomous, Asynchronous**
 - **CPU or Memory - Synchronous**

I/O BUS AND INTERFACE MODULES



Each peripheral has an interface module associated with it

Interface

- Decodes the device address (device code)
- Decodes the commands (operation)
- Provides signals for the peripheral controller
- Synchronizes the data flow and supervises the transfer rate between peripheral and CPU or Memory

Typical I/O instruction

Op. code	Device address	Function code
(Command)		

I/O Bus and Memory Bus

Functions of Buses

MEMORY BUS is for information transfers between CPU and the Main Memory

- * **I/O BUS** is for information transfers between CPU and I/O devices through their I/O interface
- * Many computers use a common single bus system for both memory and I/O interface units
 - Use one common bus but separate control lines for each function
 - Use one common bus with common control lines for both functions
- * Some computer systems use two separate buses, one to communicate with memory and the other with I/O interfaces

Isolated I/O

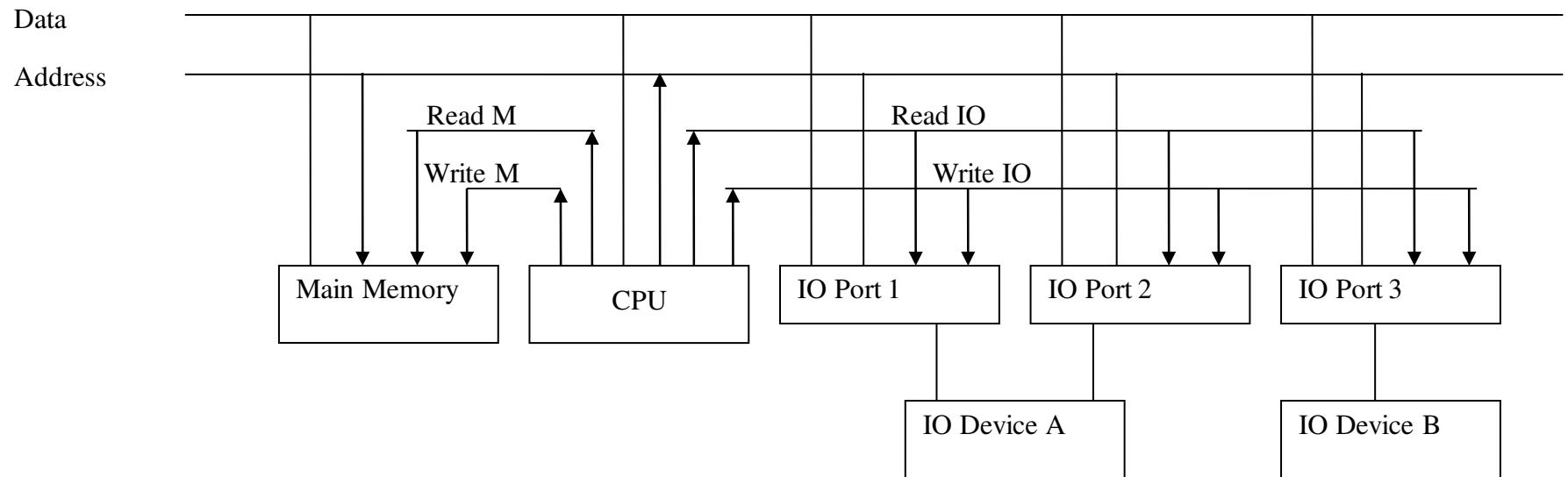
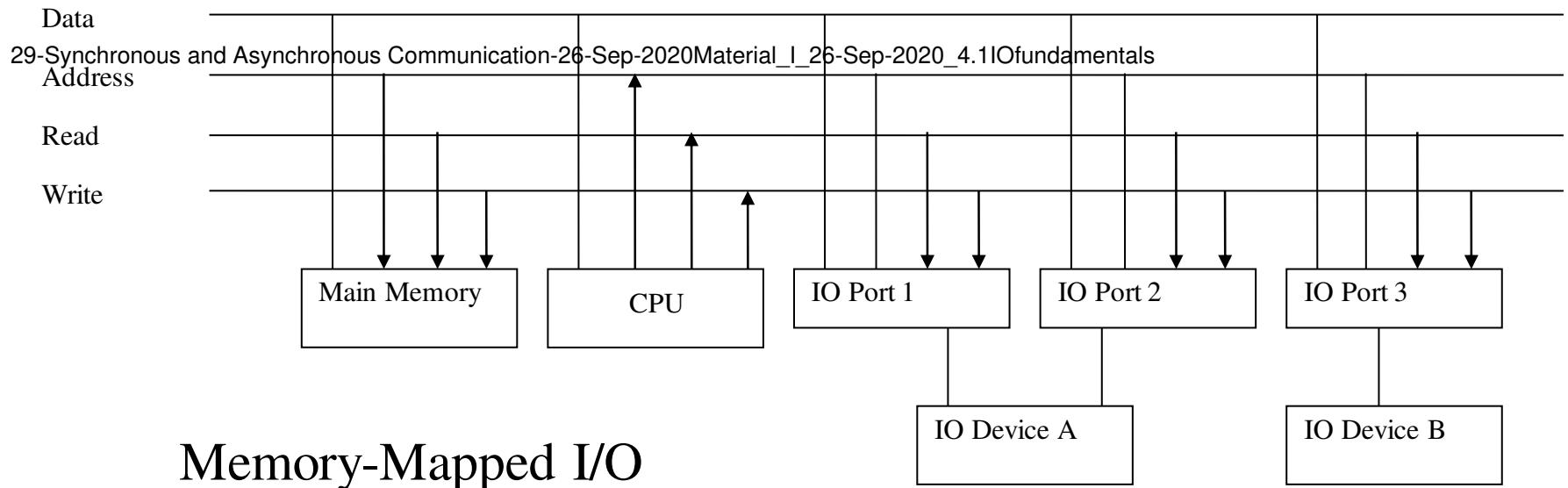
Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines.
- Separate (isolated) memory and I/O address spaces.
- Distinct input and output instructions.
- When CPU fetches and decodes the opcode of an I/O instruction, it places the address into the common address lines. Also enables read/write control lines => the address in the address lines is for interface register and not for a memory word.

Memory Mapped I/O

Memory mapped I/O

- A single set of read/write control lines
 - (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space
 - > reduces memory address range available
 - > Memory addresses xffff0000 and above are used for I/O devices
- No specific input or output instruction
 - > The same memory reference instructions can be used for I/O transfers
 - > when the bus sees certain addresses, it knows they are not memory addresses, but are addresses for accessing I/O devices.
 - Considerable flexibility in handling I/O operations



I/O Mapped I/O

Asynchronous Data Transfer

Synchronous and asynchronous operations

Synchronous - All devices derive the timing information from common clock line

Asynchronous - No common clock

Asynchronous Data Transfer

Asynchronous data transfer between two independent units requires that *control signals* be transmitted between the communicating units *to indicate the time at which data is being transmitted*

Two Asynchronous Data Transfer Methods

Strobe pulse

- A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur

Handshaking

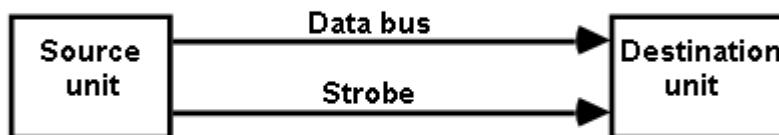
- A control signal is accompanied with each data being transmitted to indicate the presence of data
- The receiving unit responds with another control signal to acknowledge receipt of the data

Strobe Control

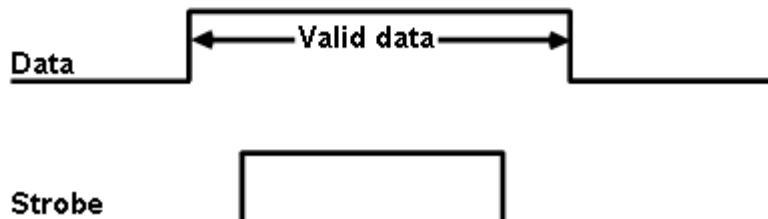
- **Employs a single control line to time each transfer**
- **The strobe may be activated by either the source or the destination unit**
- **Data bus carries the binary information from source to destination**

Source-Initiated Strobe for Data Transfer

Block Diagram

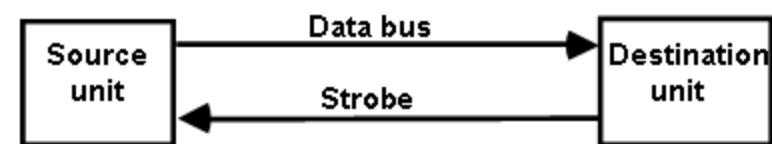


Timing Diagram

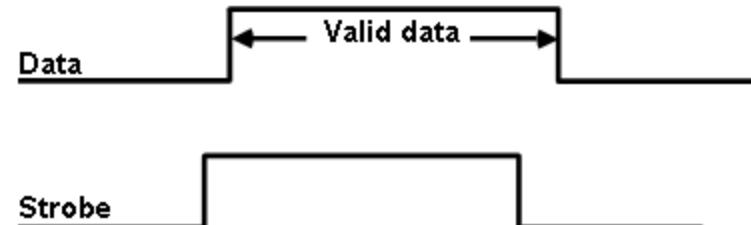


Destination-Initiated Strobe for Data Transfer

Block Diagram



Timing Diagram



Source initiated strobe

- It could be a memory write control signal from the CPU to a memory unit.
- Source is the CPU, places a word on the data bus and informs the memory unit which is destination, that this is a write operation
- Disadvantage – no way of knowing whether the destination unit has actually received data

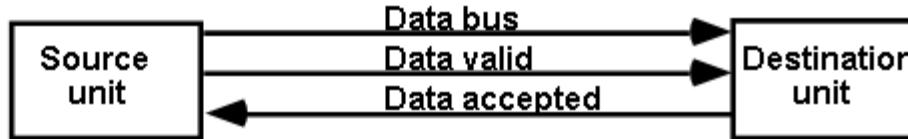
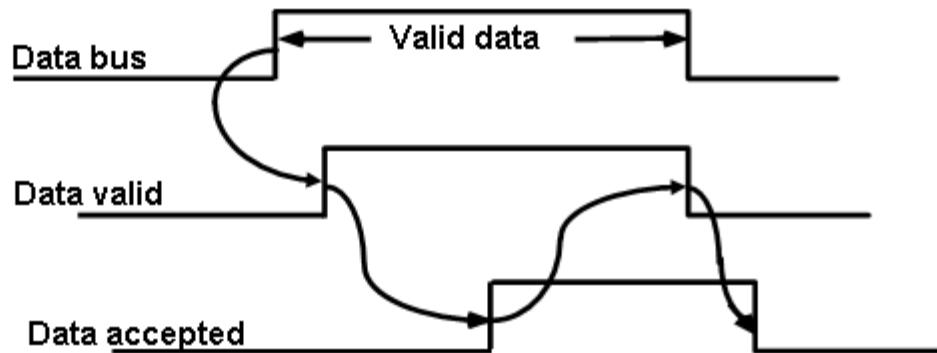
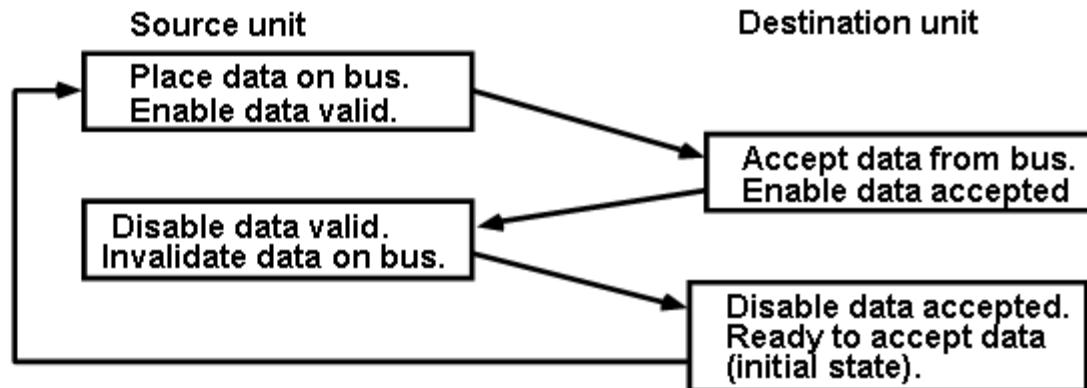
Destination initiated strobe

- It could be a memory read control signal.
- CPU, the destination initiates the read operation to inform the memory which is the source to place a selected word into the data bus.
- Disadvantage - no way of knowing whether the source has actually placed the data on the bus

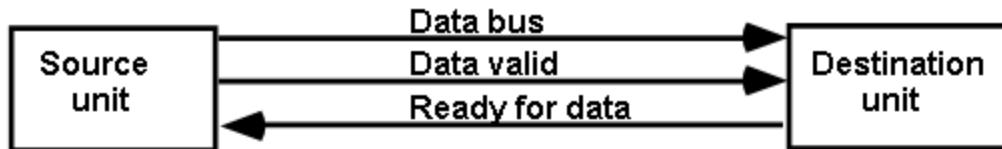
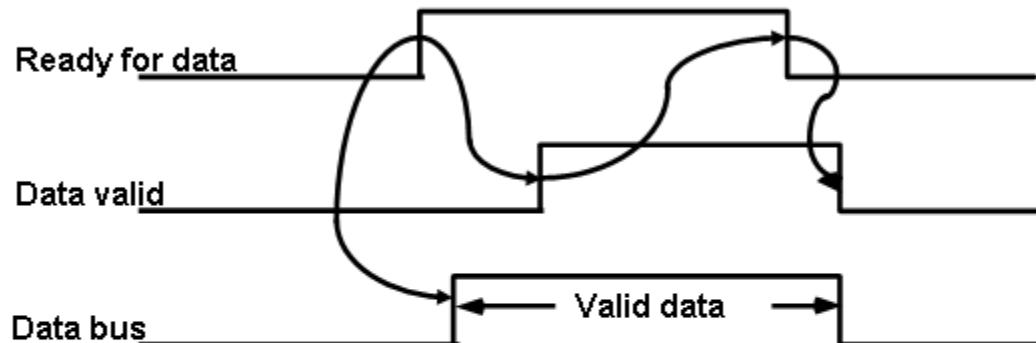
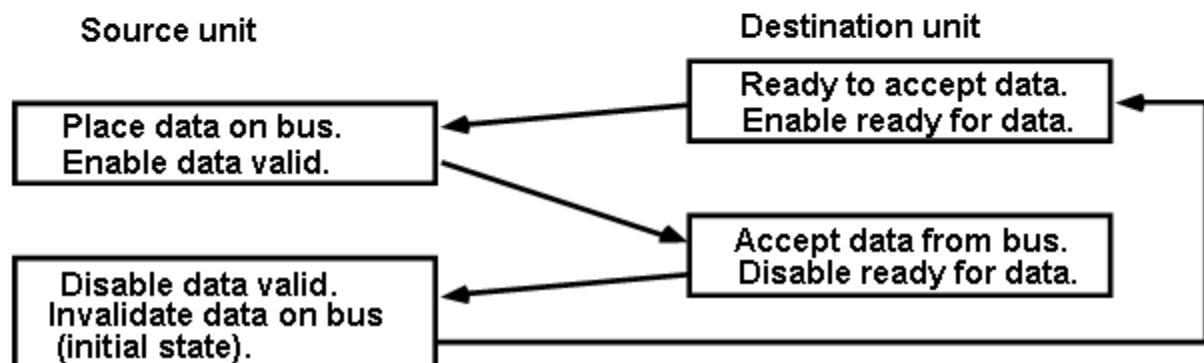
Hand Shaking

- To solve the disadvantages of strobe, the *HANDSHAKE* method introduces a second control signal to provide a *Reply* to the unit that initiates the transfer.
- 2 types
 - Source initiated transfer
 - Destination initiated transfer
- Provides ***high degree of flexibility and reliability***
- Incompletion of data transfer can be detected by means of a ***timeout mechanism***
- The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.

SOURCE-INITIATED TRANSFER USING HANDSHAKE

Block Diagram**Timing Diagram****Sequence of Events**

DESTINATION-INITIATED TRANSFER USING HANDSHAKE

Block Diagram**Timing Diagram****Sequence of Events**

Modes of Transfer

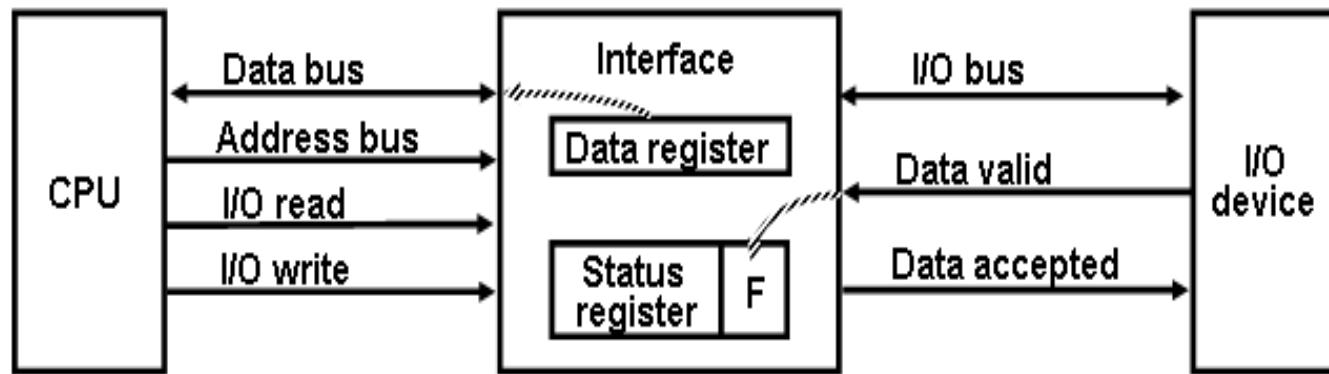
- 3 different **Data Transfer Modes** between the central computer (CPU or Memory) and peripherals;
 - Program-Controlled I/O or programmed I/O
 - Interrupt-Initiated I/O or Interrupt driven I/O
 - Direct Memory Access (DMA)
- **Programmed I/O**
 - Transferring data under program control requires constant monitoring of the peripheral by CPU.
 - I/O device does not have direct access to memory
 - Programmed I/O is time consuming process since it keeps the processor busy needlessly.

Programmed I/O

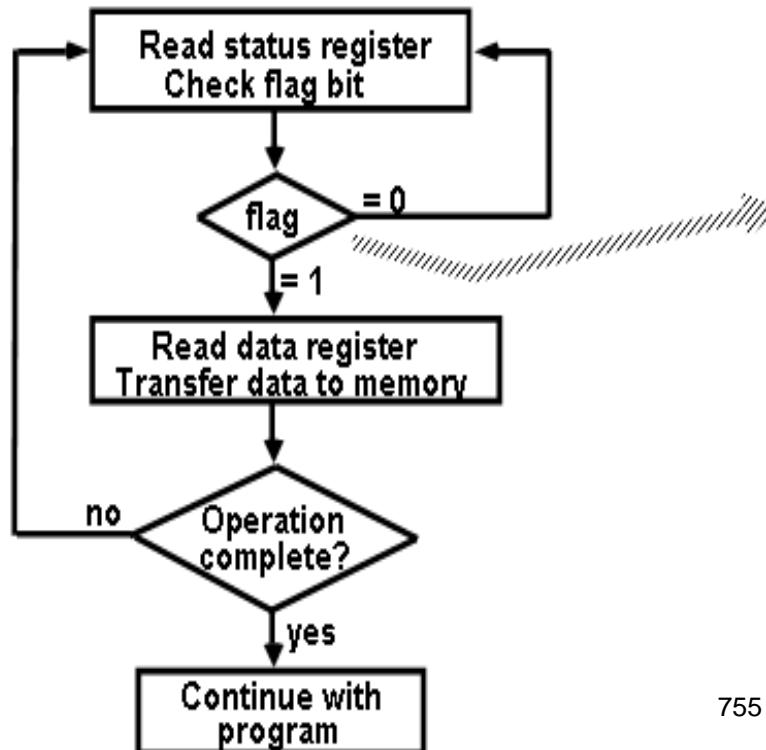
- More instructions may be required during the transfer of data from I/O device to memory
 - Input instruction: from I/P device to CPU
 - Store instruction: CPU->memory
 - To verify the data availability
 - To count the number of words transferred
- Useful in small low speed systems or in dedicated systems to monitor a device continuously.

Programmed I/O

Data Transfer from I/O device to CPU



Flow chart for CPU program to input data



Polling or Status Checking

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

Interrupt-Initiated I/O

- When the interface determines that the I/O device is ready for data transfer, it generates an *Interrupt Request* to the CPU .
- Upon detecting an interrupt, CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing

Vectored and Non-Vectored

- 2 methods in which the CPU chooses the branch address of the service routine
 - Non-vectored interrupt
 - *Non-Vectored Interrupts* are those in which vector address is not predefined. The interrupting device gives the address of sub-routine for these interrupts.
 - *INTR* is the only non-vectored interrupt in 8085 microprocessor.
 - Vectored interrupt
 - Interrupt supplies the branch info (direct or indirect address of ISR) to the CPU
 - Processor automatically generates the subroutine address

Vectored Interrupts

INTERRUPT	VECTOR ADDRESS
TRAP (RST 4.5)	24 H
RST 5.5	2C H
RST 6.5	34 H
RST 7.5	3C H

INTERRUPT	VECTOR ADDRESS
RST 0	00 H
RST 1	08 H
RST 2	10 H
RST 3	18 H
RST 4	20 H
RST 5	28 H
RST 6	30 H
RST 7	38 H

Maskable and Non-Maskable Interrupts

- ***Maskable Interrupts*** are those which can be disabled or ignored by the microprocessor.
 - *INTR, RST 7.5, RST 6.5, RST 5.5* are maskable interrupts in 8085 microprocessor.
- ***Non-Maskable Interrupts*** are those which cannot be disabled or ignored by microprocessor.
 - *TRAP* is a non-maskable interrupt.

PRIORITY INTERRUPT

Priority

- Determines which interrupt is to be served first when two or more requests are made simultaneously
- Also determines which interrupts are permitted to interrupt the computer while another is being serviced
- Higher priority interrupts can make requests while servicing a lower priority interrupt

Priority Interrupt by Software(Polling)

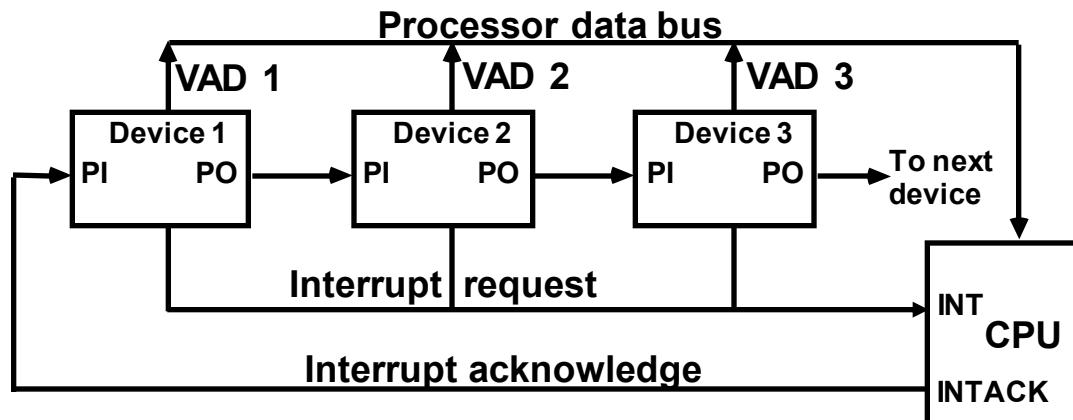
- Priority is established by the order of polling the devices(interrupt sources)
- Flexible since it is established by software
- Low cost since it needs a very little hardware
- Very slow

Priority Interrupt by Hardware

- Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request
- Fast since identification of the highest priority interrupt request is identified by the hardware
- Fast since each interrupt source has its own interrupt vector to access directly to its own service routine

HARDWARE PRIORITY INTERRUPT - DAISY-CHAIN

26-Synchronous and Asynchronous Communication-26-Sep-2020 Material I _ 26-Sep-2020 _ 4.10 fundamentals



- * Serial hardware priority function
- * Interrupt Request Line
 - Single common line
- * Interrupt Acknowledge Line
 - Daisy-Chain

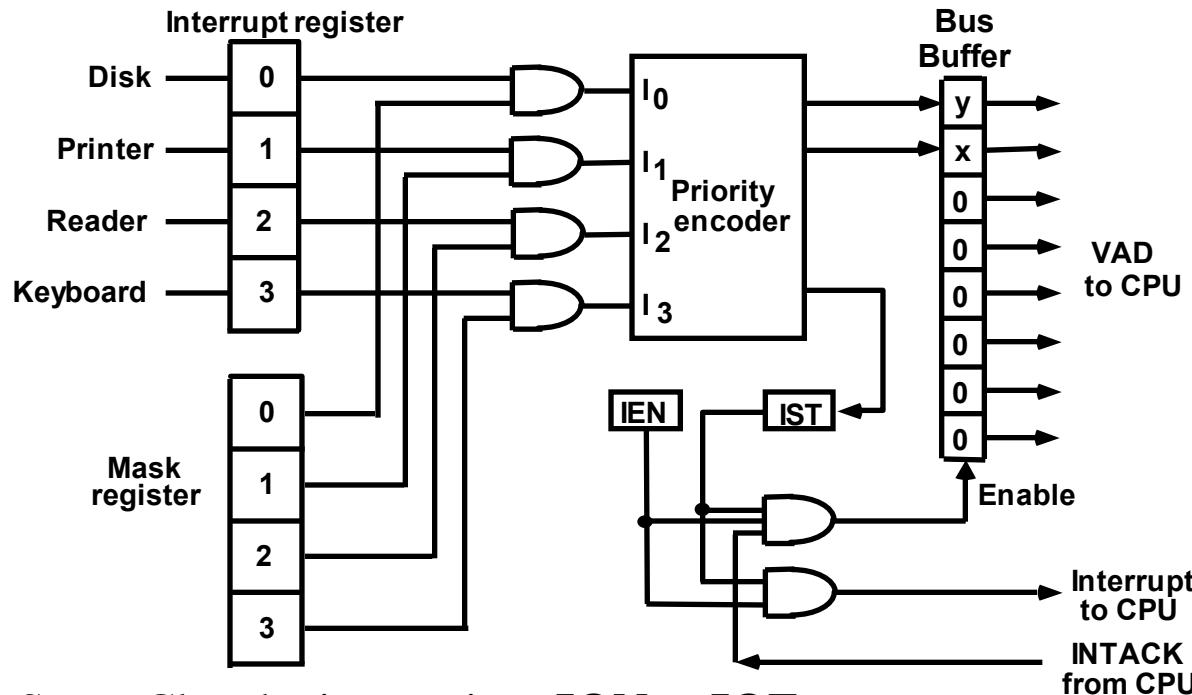
Interrupt Request from any device(>=1)

-> CPU responds by **INTACK** $\leftarrow 1$

-> Any device receives signal (**INTACK**) 1 at **PI** puts the **VAD** on the bus

Among interrupt requesting devices the only device which is physically closest to **CPU** gets **INTACK=1**, and it blocks **INTACK** to propagate to the next device

PARALLEL PRIORITY INTERRUPT



IEN: Set or Clear by instructions **ION** or **IOF**

IST: Represents an unmasked interrupt has occurred. **INTACK** enables tristate Bus Buffer to load **VAD** generated by the Priority Logic

Interrupt Register:

- Each bit is associated with an Interrupt Request from different Interrupt Source
 - different priority level
- Each bit can be cleared by a program instruction

Mask Register:

- Mask Register is associated with Interrupt Register
- Each bit can be set or cleared by an Instruction
- Control the status of each interrupt req, disable low priority device while a high priority device is being served

INTERRUPT PRIORITY ENCODER

Determines the highest priority interrupt when more than one interrupts take place

Priority Encoder Truth table

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	d	d	d	0	0	1	
0	1	d	d	0	1	1	
0	0	1	d	1	0	1	$x = I_0' \cdot I_1'$
0	0	0	1	1	1	1	$y = I_0' \cdot I_1 + I_0 \cdot I_2'$
0	0	0	0	d	d	0	$(IST) = I_0 + I_1 + I_2 + I_3$

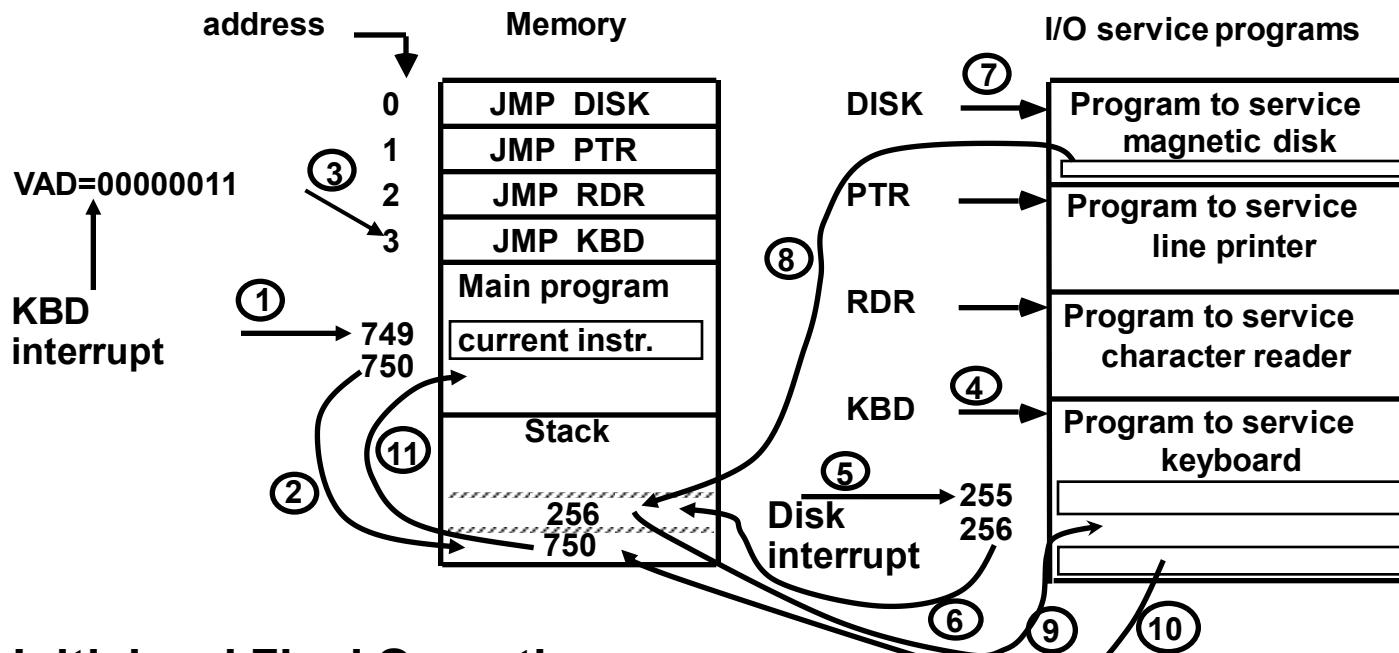
INTERRUPT CYCLE

At the end of each Instruction cycle

- CPU checks IEN and IST
- If $IEN \bullet IST = 1$, CPU \rightarrow Interrupt Cycle

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push PC into stack
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow VAD$	Transfer vector address to PC
$IEN \leftarrow 0$	Disable further interrupts
Go To Fetch	to execute the first instruction in the interrupt service routine

INTERRUPT SERVICE ROUTINE



Initial and Final Operations

Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system

Initial Sequence

- [1] Clear lower level Mask reg. bits
- [2] IST <- 0
- [3] Save contents of CPU registers
- [4] IEN <- 1
- [5] Go to Interrupt Service Routine

Final Sequence

- [1] IEN <- 0
- [2] Restore CPU registers
- [3] Clear the bit in the Interrupt Reg
- [4] Set lower level Mask reg. bits
- [5] Restore return address, IEN <- 1

Role of CPU in transfer of information

peripheral device -> CPU -> memory

⇒ CPU limits the speed of transfer

Peripheral device -> memory

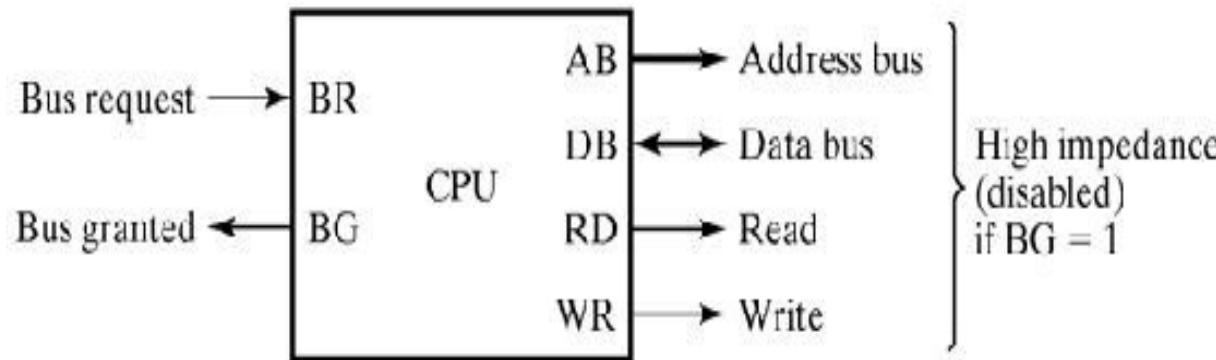
⇒ Transfer speed increases

⇒ Peripheral device manage the memory bus directly.

⇒ DMA (Direct memory Access)

⇒ CPU is idle

CPU Bus Signals for DMA Transfer

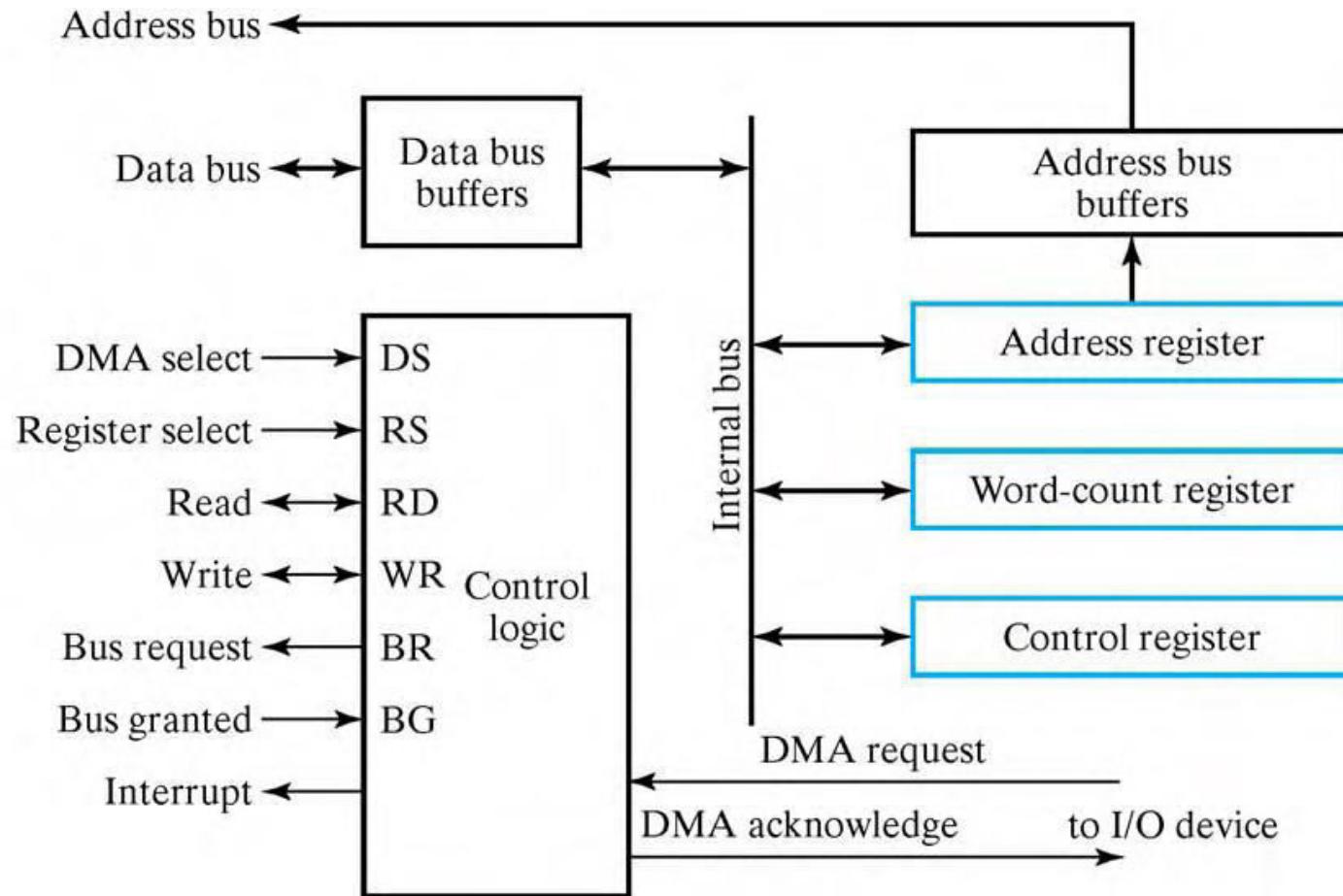


- DMA Controller sends **Bus Request** to CPU
- CPU stops execution of current instruction and places address bus, data bus, read and write lines into high impedance state.
- CPU activates **Bus Grant**
- After transfer DMA disables BR
- CPU continues normal operation

Different ways of DMA transfer

- Burst transfer – a block sequence consisting of a number of memory words is transferred in a continuous burst => need for fast devices
- Cycle stealing – transfer one word at a time, after which it must return control of the buses to the CPU.
 - CPU delays 1 cycle to allow Direct Memory I/O transfer

DMA Controller



DMA Controller contd.,

- Three registers
 - Address registers – to specify the desired location in memory
 - Incremented after each word transfer
 - Word count registers – numbers of words to be transferred
 - Decrement after each word transfer and tested for 0.
 - Control register – specifies mode of transfer

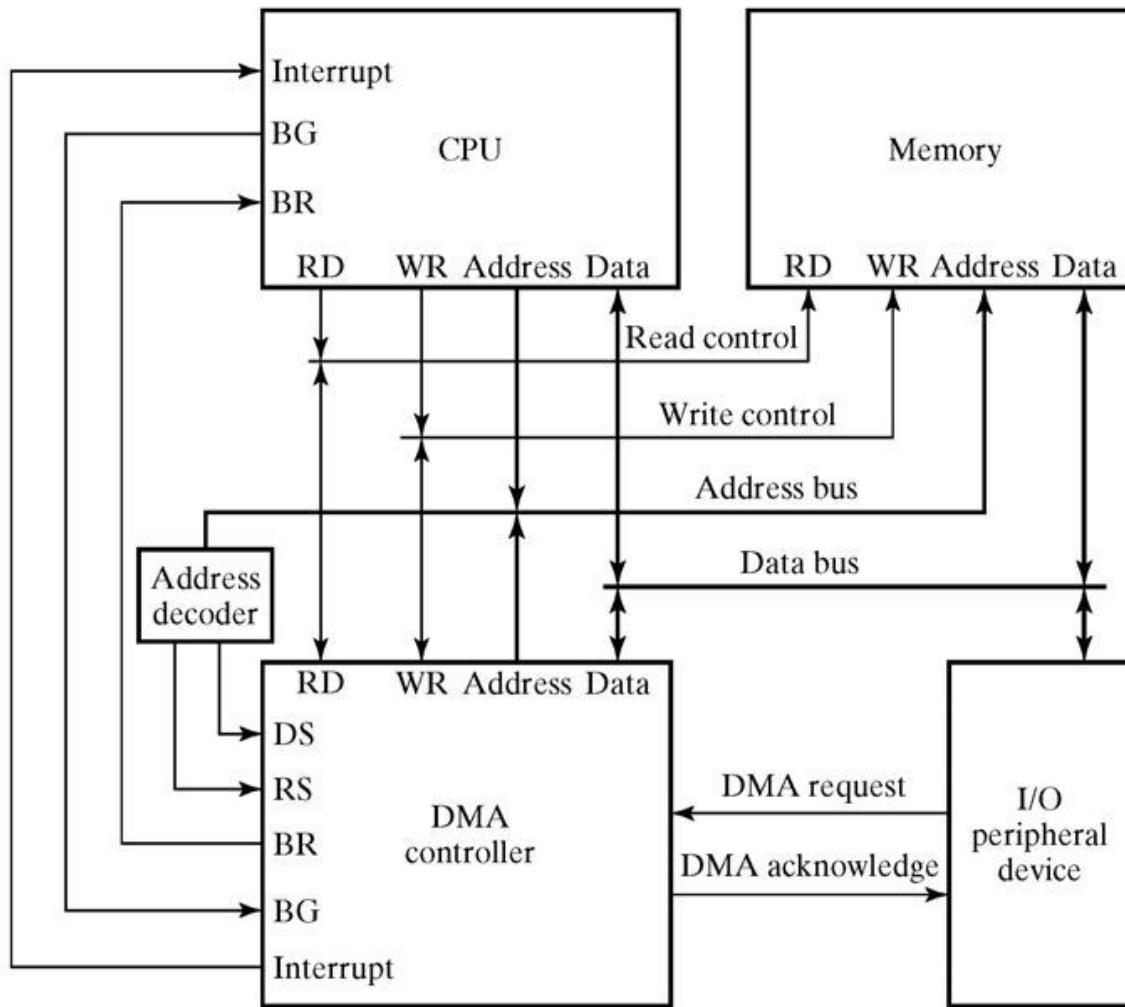
DMA Controller

- Data bus and control lines => used to communicate with CPU
- Registers in DMA are selected by CPU through address bus by enabling DS and RS inputs.
- BG = 0 => CPU reads from or writes to the DMA registers
- BG = 1 => DMA directly communicates with Memory by specifying address in address bus and activates the RD or WR control.
- Request and acknowledge signals are used to communicate with peripheral devices

Initialization of DMA

- CPU initializes DMA by sending the following information through data bus
 - Starting address of the memory block where data are available (for read) or where data are to be stored (for write)
 - The word count
 - Control – to read or write
 - A control to start the DMA transfer
- After initialization, CPU stops communicating with DMA unless it receives an interrupt signal or if it wants to checks how many words have been transferred.

DMA Transfer



DMA Transfer

- Peripheral device $\xrightarrow{\text{DMA Request}}$ DMA controller $\xrightarrow{\text{Bus Request}}$
CPU $\xrightarrow{\text{Bus grant}}$ DMA controller
- DMA controller puts the current value of its address register onto address bus, activates RD or WR signal, and $\xrightarrow{\text{DMA Acknowledge}}$ peripheral device
- RD and WR are bidirectional
 - BR = 0 => CPU communicates with the internal DMA registers
 - BR = 1 => RD and WR are output lines from DMA controller to the RAM to specify read or write operation for data

DMA contd.,

- When the peripheral device receives a DMA acknowledge, it puts a word onto data bus or receives a word from data bus
- For each word transfer, DMA increments address register and decrements word count register.
- If WC = 0, DMA disables Bus request.
- Termination is informed to CPU by means of interrupt signal.
- When CPU receives interrupt signal, it checks value of word count register.

References

- M. M. Mano, Computer System Architecture, Prentice-Hall

DIRECT MEMORY ACCESS

OVERVIEW

- ▶ Introduction
- ▶ Implementing DMA in a computer system
- ▶ Data transfer using DMA controller
- ▶ Internal configuration of a DMA controller
- ▶ Process of DMA transfer
- ▶ DMA transfer modes

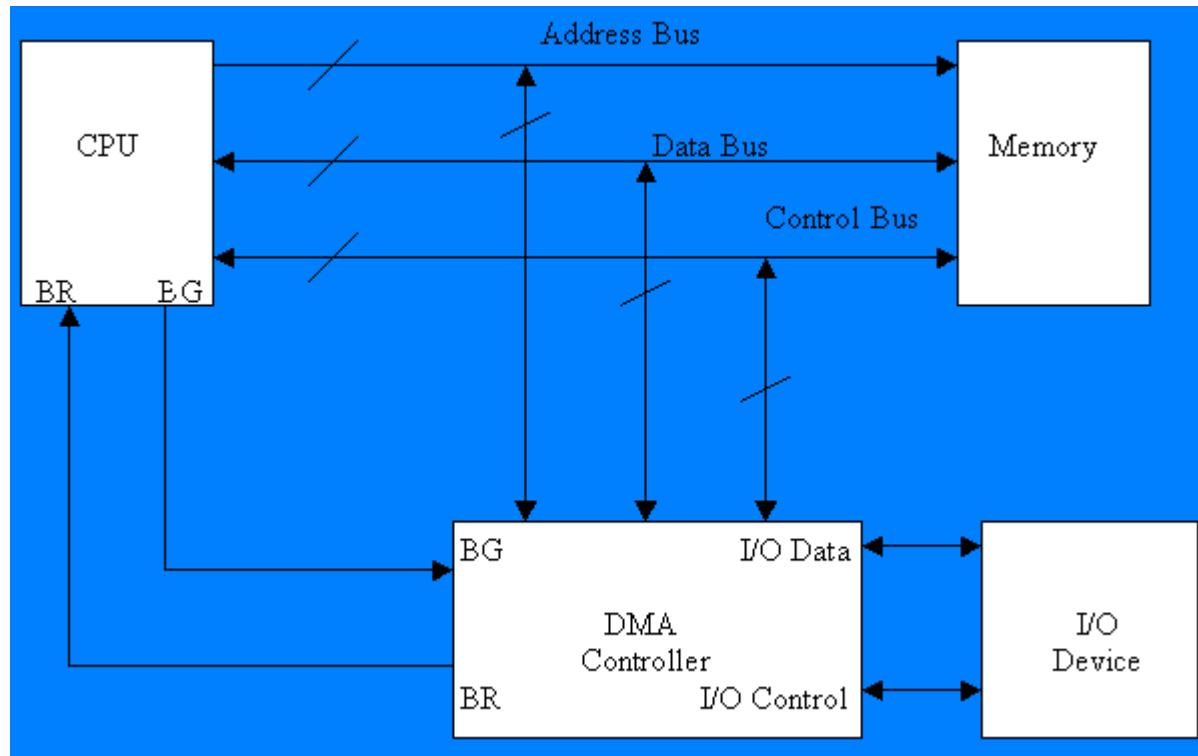
Direct Memory Access

▶ Introduction

- An important aspect governing the Computer System performance is the transfer of data between memory and I/O devices.
- The operation involves loading programs or data files from disk into memory, saving file on disk, and accessing virtual memory pages on any secondary storage medium.

cont→

Computer System with DMA



- ▶ Consider a typical system consisting of a CPU, memory and one or more input/output devices as shown in fig. Assume one of the I/O devices is a disk drive and that the computer must load a program from this drive into memory.
- ▶ The CPU would read the first byte of the program and then write that byte to memory. Then it would do the same for the second byte, until it had loaded the entire program into memory.

cont→

- ▶ This process proves to be inefficient. Loading data into, and then writing data out of the CPU significantly slows down the transfer. The CPU does not modify the data at all, so it only serves as an additional stop for data on the way to its final destination.
- ▶ The process would be much quicker if we could bypass the CPU & transfer data directly from the I/O device to memory.
- ▶ Direct Memory Access does exactly that.

Implementing DMA in a Computer System

- ▶ A DMA controller implements direct memory access in a computer system.
- ▶ It connects directly to the I/O device at one end and to the system buses at the other end. It also interacts with the CPU, both via the system buses and two new direct connections.
- ▶ It is sometimes referred to as a channel. In an alternate configuration, the DMA controller may be incorporated directly into the I/O device.

Data Transfer using DMA Controller

- ▶ To transfer data from an I/O device to memory, the DMA controller first sends a Bus Request to the CPU by setting BR to 1. When it is ready to grant this request, the CPU sets its Bus grant signal, BG to 1.
- ▶ The CPU also tri-states its address, data, and control lines thus truly granting control of the system buses to the DMA controller.
- ▶ The CPU will continue to tri-state its outputs as long as BR is asserted.

cont→

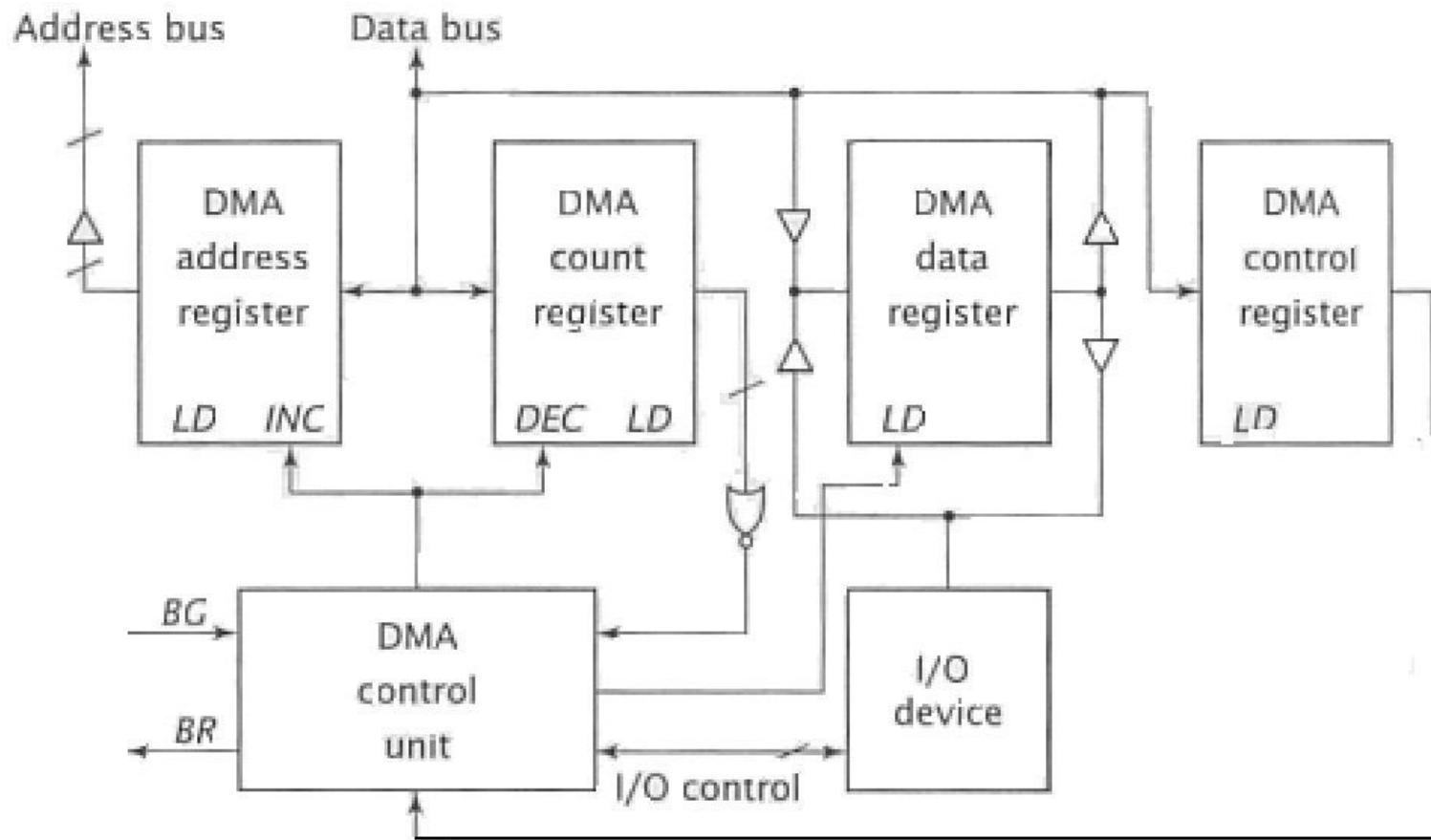
Internal Configuration

- ▶ The DMA controller includes several registers :-
 - The **DMA Address Register** contains the memory address to be used in the data transfer. The CPU treats this signal as one or more output ports.
 - The **DMA Count Register**, also called Word Count Register, contains the no. of bytes of data to be transferred. Like the DMA address register, it too is treated as an O/P port (with a diff. Address) by the CPU.
 - The **DMA Control Register** accepts commands from the CPU. It is also treated as an O/P port by the CPU.

cont→

- ▶ Although not shown in this fig., most DMA controllers also have a **Status Register**. This register supplies information to the CPU, which accesses it as an I/P port.

Internal Configuration of DMA Controller



Process of DMA Transfer

- ▶ To initiate a DMA transfer, the CPU loads the address of the first memory location of the memory block (to be read or written from) into the DMA address register. It does this via an I/O output instruction, such as the OTPT instruction for the relatively simple CPU.
- ▶ It then writes the no. of bytes to be transferred into the DMA count register in the same manner.
- ▶ Finally, it writes one or more commands to the DMA control register.

cont→

- ▶ These commands may specify transfer options such as the DMA transfer mode, but should always specify the direction of the transfer, either from I/O to memory or from memory to I/O.
- ▶ The last command causes the DMA controller to initiate the transfer. The controller then sets BR to 1 and, once BG becomes 1, seizes control of the system buses.

DMA Transfer Modes

Modes vary by how the DMA controller determines when to transfer data, but the actual data transfer process is the same for all the modes.

- **BURST mode**
 - Sometimes called Block Transfer Mode.
 - An entire block of data is transferred in one contiguous sequence. Once the DMA controller is granted access to the system buses by the CPU, it transfers all bytes of data in the data block before releasing control of the system buses back to the CPU.
 - This mode is useful for loading programs or data files into memory, but it does render the CPU inactive for relatively long periods of time.

cont→

- CYCLE STEALING Mode

- Viable alternative for systems in which the CPU should not be disabled for the length of time needed for Burst transfer modes.
- DMA controller obtains access to the system buses as in burst mode, using BR & BG signals. However, it transfers one byte of data and then deasserts BR, returning control of the system buses to the CPU. It continually issues requests via BR, transferring one byte of data per request, until it has transferred its entire block of data.

cont→

- By continually obtaining and releasing control of the system buses, the DMA controller essentially interleaves instruction & data transfers. The CPU processes an instruction, then the DMA controller transfers a data value, and so on.
- The data block is not transferred as quickly as in burst mode, but the CPU is not idled for as long as in that mode.
- Useful for controllers monitoring data in real time.

- **TRANSPARENT Mode**

- This requires the most time to transfer a block of data, yet it is also the most efficient in terms of overall system performance.
- The DMA controller only transfers data when the CPU is performing operations that do not use the system buses.
- For example, the Relatively simple CPU has several states that move or process data solely within the CPU:

NOP1: (No operation)

LDAC5: AC←DR

cont→

JUMP3: $PC \leftarrow DR, TR$

CLAC1: $AC \leftarrow 0, Z \leftarrow 1$

- Primary advantage is that CPU never stops executing its programs and DMA transfer is free in terms of time.
- Disadvantage is that the hardware needed to determine when the CPU is not using the system buses can be quite complex and relatively expensive.

Summary

► Advantages of DMA

- Computer system performance is improved by direct transfer of data between memory and I/O devices, bypassing the CPU.
- CPU is free to perform operations that do not use system buses.

► Disadvantages of DMA

- In case of Burst Mode data transfer, the CPU is rendered inactive for relatively long periods of time.

VECTORED AND PRIORITY INTERRUPTS

LIJO V P
SCOPE
VIT, VELLORE

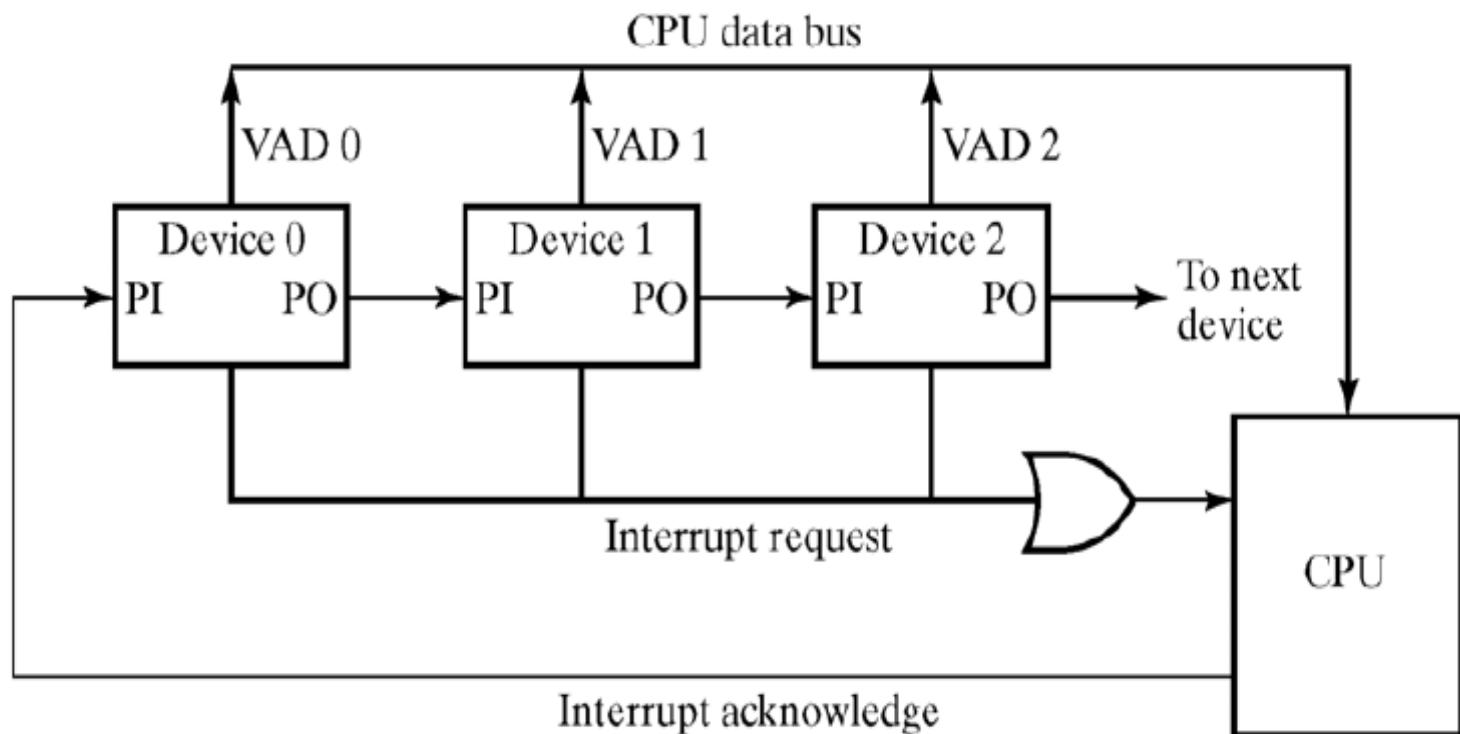
Priority Interrupt - polling

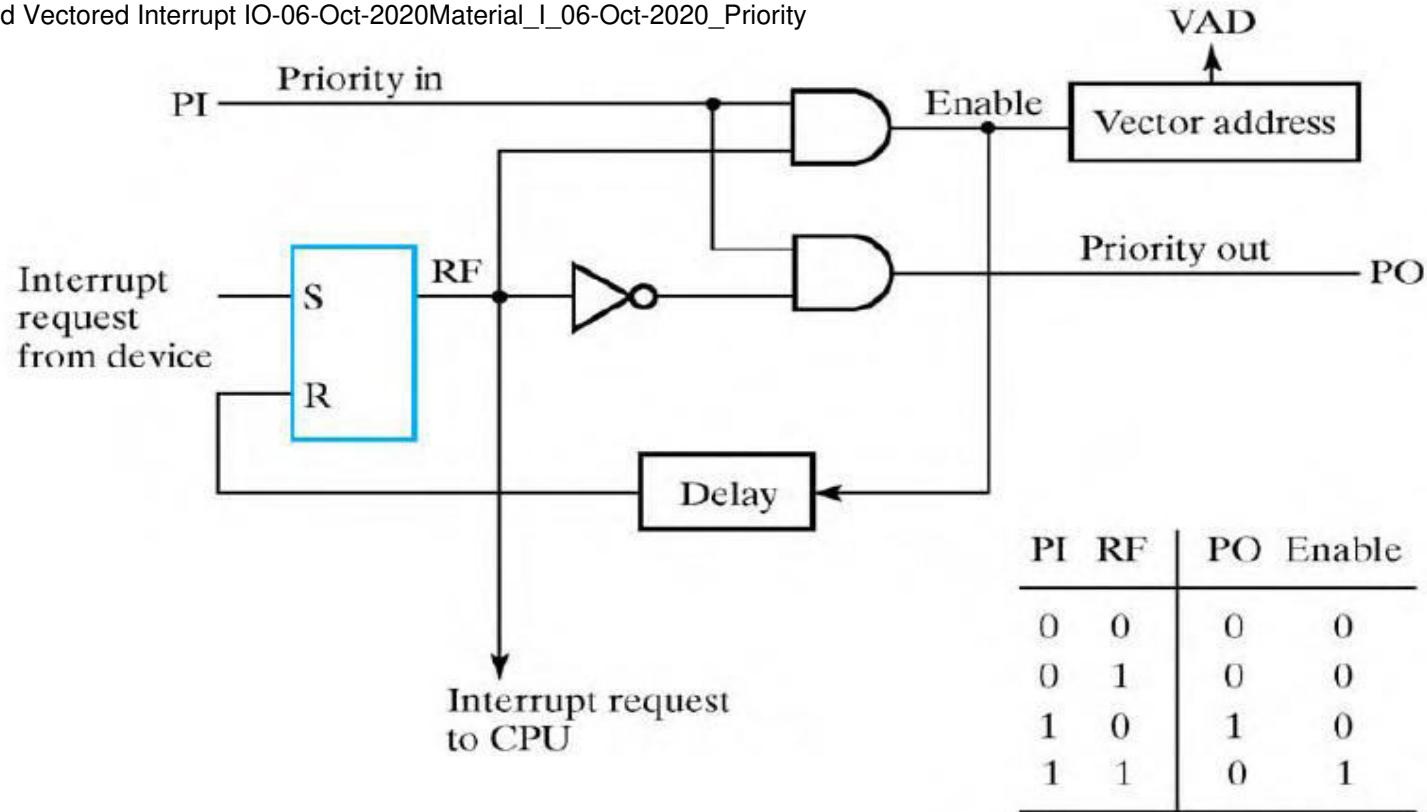
- Establishes priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.
- The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced.
- Priority can be established by software or hardware
- Highest priority source is tested first and if its interrupt signal is on, control branches to a service routine for this source

- Otherwise, the next-lower priority source is tested and so on.
- **Disadvantage – if many interrupts**, time required to poll them may exceed the time available to service the I/O device.
- Then, hardware priority interrupt unit can be used to speed up the operation.
- Hardware priority interrupts – accepts interrupt request from many sources, determines which of the incoming requests has the highest priority and issues an interrupt request to the computer.
- Here, each interrupt source has its own interrupt vector to access its own service routine directly.

Daisy chaining

- Hardware priority interrupts – serial or parallel connection of interrupt lines.
- Serial – daisy chaining





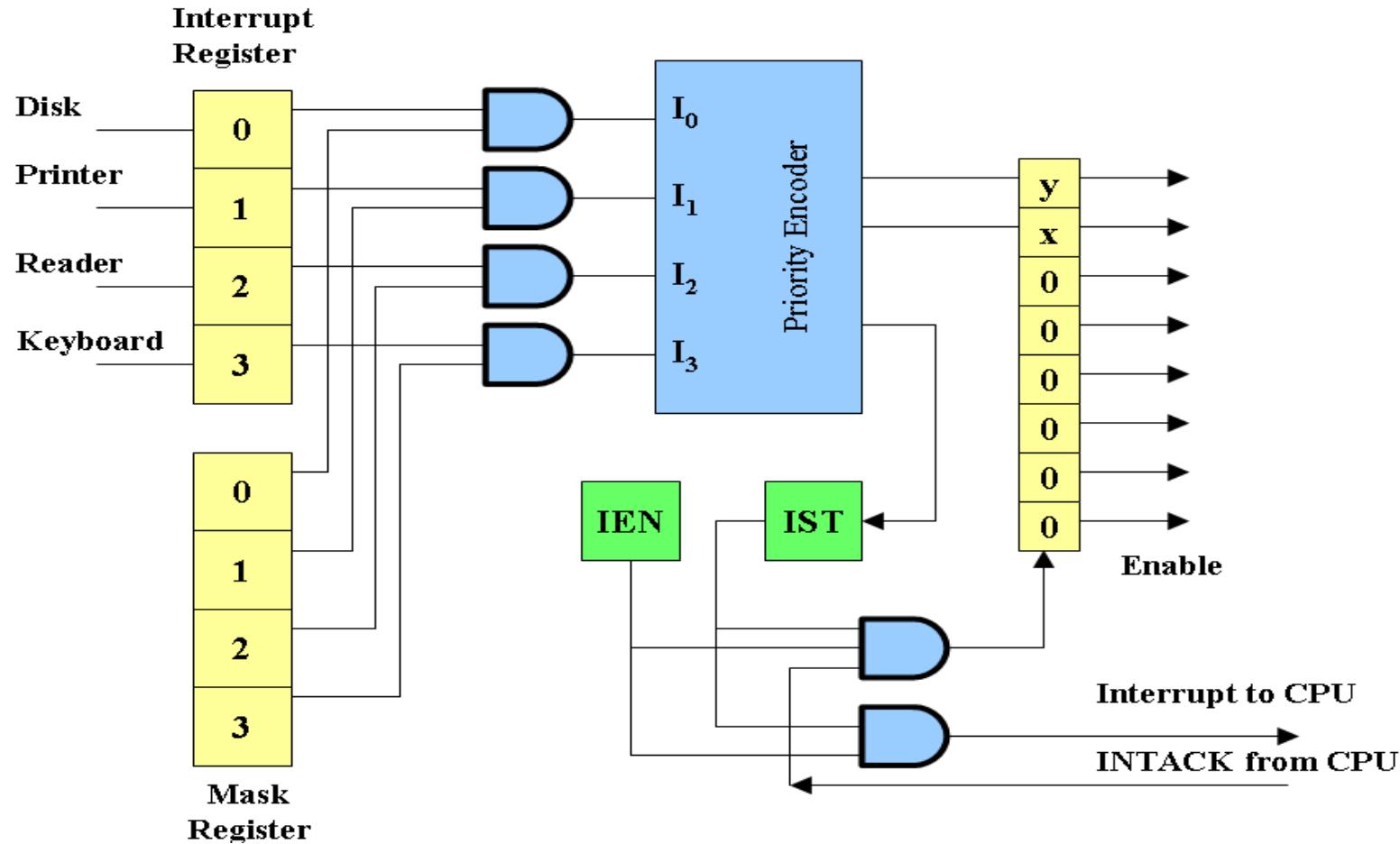
One stage of daisy chain priority interrupt scheme

If PI = 0, both PO and the enable line to VAD (vector address) are equal to 0, irrespective of RF.

- If $PI = 1$ and $RF = 0$, then $PO = 1$ and the vector address is disabled \Rightarrow passes the acknowledge signal to the next device through PO .
- The device is active when $PI = 1$ and $RF = 1$. This condition places a 0 in PO and enables the vector address for the data bus.
- The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

Parallel Priority Interrupt

- Uses a register whose bits are set separately by the interrupt signal from each device.
- Priority is established according to the position of the bits in the register.
- Mask register is used to disable lower priority interrupts while a higher priority device is being serviced.
- It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower priority device is being serviced



- Mask register has a same number of bits as the interrupt register.
- By means of program instructions, it is possible to set or reset any bit in the mask register.

- The priority encoder generates two bits of the vector address, which is transferred to the CPU.
- Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs.
- The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- IST ANDed with IEN provide a common interrupt signal for the CPU.
- The INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

Priority encoder

Circuit that implements the priority function.

Logic – if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

Boolean functions

$$X = I'_0 I'_1$$

$$Y = I'_0 I_1 + I'_0 I'_2$$

$$IST = I_0 + I_1 + I_2 + I_3$$

Inputs			
I₀	I₁	I₂	I₃
1	d	d	d
0	1	d	d
0	0	1	d
0	0	0	1
0	0	0	0

Outputs		
d	Y	IST
0	0	1
0	1	1
1	0	1
1	1	1
d	d	0

Interrupt

- An interrupt is a signal from a device attached to a computer or from a program within the computer that causes the CPU to stop its normal program execution and perform service related to the event.
- Examples of interrupts :I/O completion, divide-by-0, etc.
- **Maskable Interrupt:** It is a hardware interrupt that may be ignored by setting a bit in an interrupt mask register's (IMR) bit-mask.
- **Non-maskable Interrupt:** is a hardware interrupt that does not have a bit-mask associated with it - meaning that it can never be ignored. NMIs are often used for timers, especially watchdog timers

Interrupt Cycle

The Interrupt enable flip-flop (IEN) can be set or cleared by program instructions.

A programmer can therefore allow interrupts (clear IEN) or disallow interrupts (set IEN)

At the end of each instruction cycle the CPU checks IEN and IST. If either is equal to zero, control continues with the next instruction. If both = 1, the interrupt is handled.

Interrupt micro-operations:

$SP \leftarrow SP - 1$ (Decrement stack pointer)

$M[SP] \leftarrow PC$ Push PC onto stack

$INTACK \leftarrow 1$ Enable interrupt acknowledge

$PC \leftarrow VAD$ Transfer vector address to PC

$IEN \leftarrow 0$ Disable further interrupts

Go to fetch next instruction

Interrupt Service Routine

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated.

Interrupt Overhead

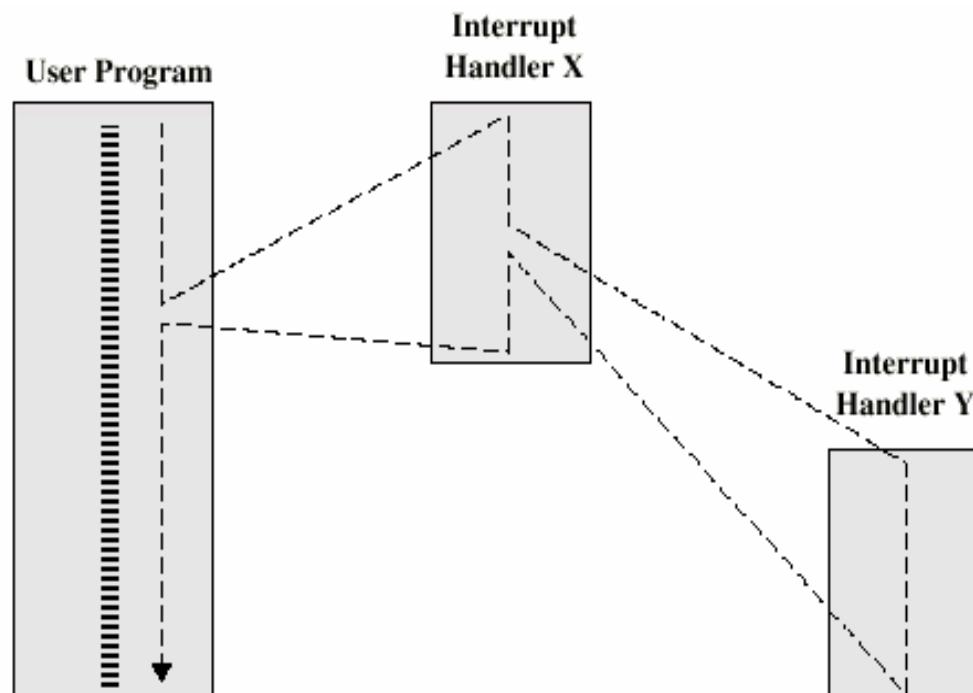
The interrupt overhead is caused by context switching (storing and restoring the state of CPU)

On interrupt handler entry, the context of the current process and its thread must be saved. On exit, it must be restored.

On handler entry, memory locations different from the memory locations in the cache are used, and therefore cache updates are required.

Interrupt nesting

- An interrupt can happen while executing an ISR. This is called *interrupt nesting*.



References

- M. M. Mano, Computer System Architecture, Prentice-Hall

Buses: bus protocols and bus arbitration

Buses

- There are a number of possible interconnection systems
- Single and multiple BUS structures are most common
- e.g. Control/Address/Data bus (PC)
- e.g. Unibus (DEC-PDP)

What is a Bus?

- A communication pathway connecting two or more devices
- Usually broadcast
- Often grouped
 - A number of channels in one bus
 - e.g. 32 bit data bus is 32 separate single bit channels
- Power lines may not be shown

Data Bus

- Carries data
 - Remember that there is no difference between “data” and “instruction” at this level
- Width is a key determinant of performance
 - 8, 16, 32, 64 bit

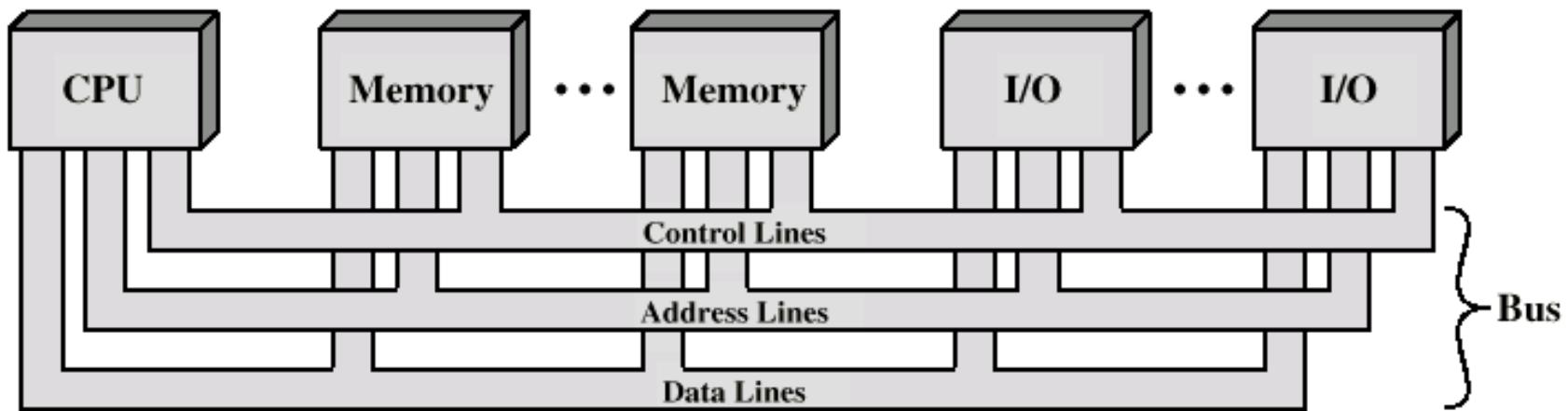
Address bus

- Identify the source or destination of data
- e.g. CPU needs to read an instruction (data) from a given location in memory
- Bus width determines maximum memory capacity of system
 - e.g. 8080 has 16 bit address bus giving 64k address space

Control Bus

- Control and timing information
 - Memory read/write signal
 - Interrupt request
 - Clock signals

Bus Interconnection Scheme



Bus Types

- Dedicated
 - Separate data & address lines
- Multiplexed
 - Shared lines
 - Address valid or data valid control line
 - Advantage - fewer lines
 - Disadvantages
 - More complex control
 - Ultimate performance

Bus Arbitration

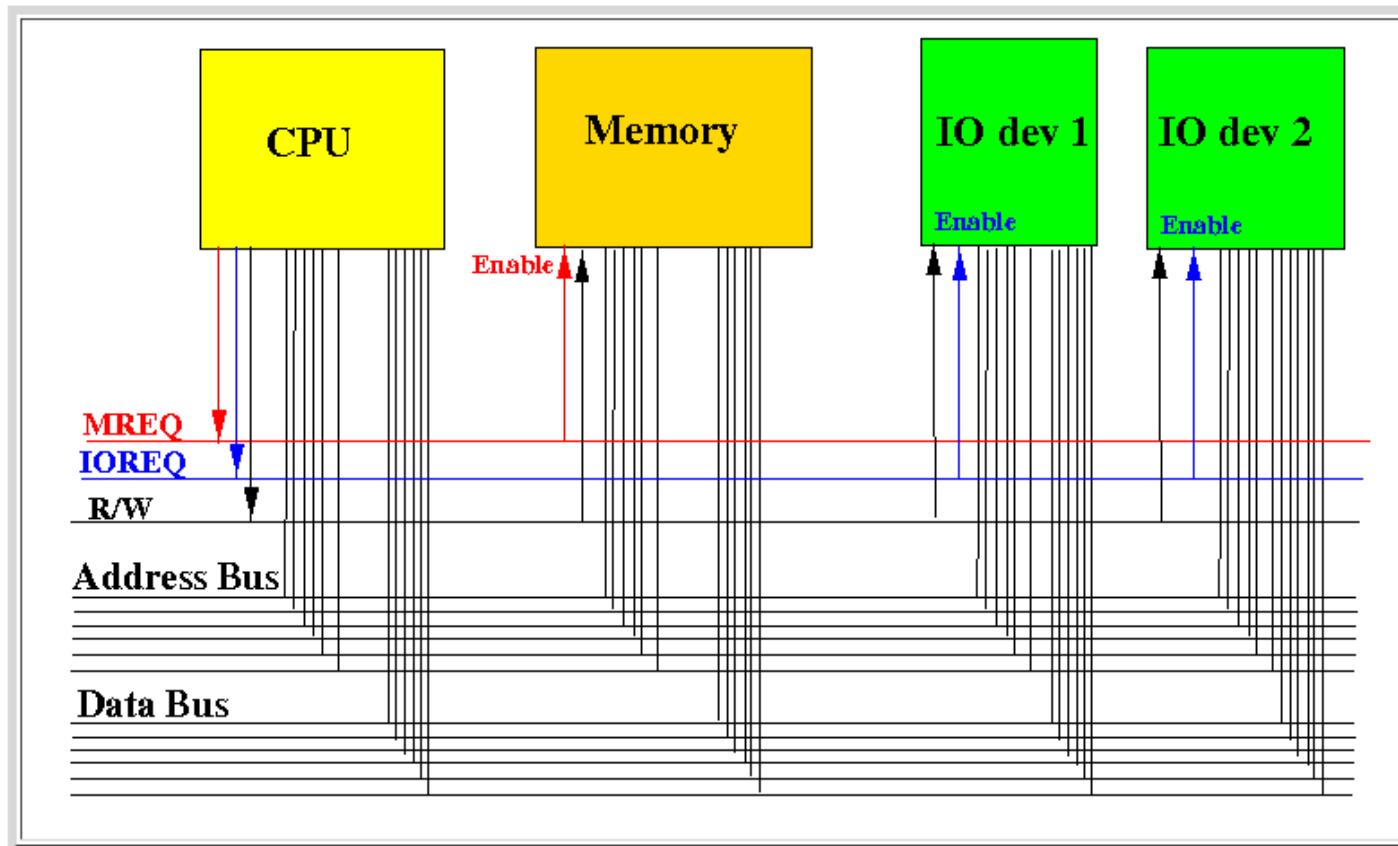
- More than one module controlling the bus
- e.g. CPU and DMA controller
- Only one module may control bus at one time
- Arbitration may be centralised or distributed

Centralised or Distributed Arbitration

- Centralised
 - Single hardware device controlling bus access
 - Bus Controller
 - Arbiter
 - May be part of CPU or separate
- Distributed
 - Each module may claim the bus
 - Control logic on all modules

Bus System

- How devices are connected inside a compute:



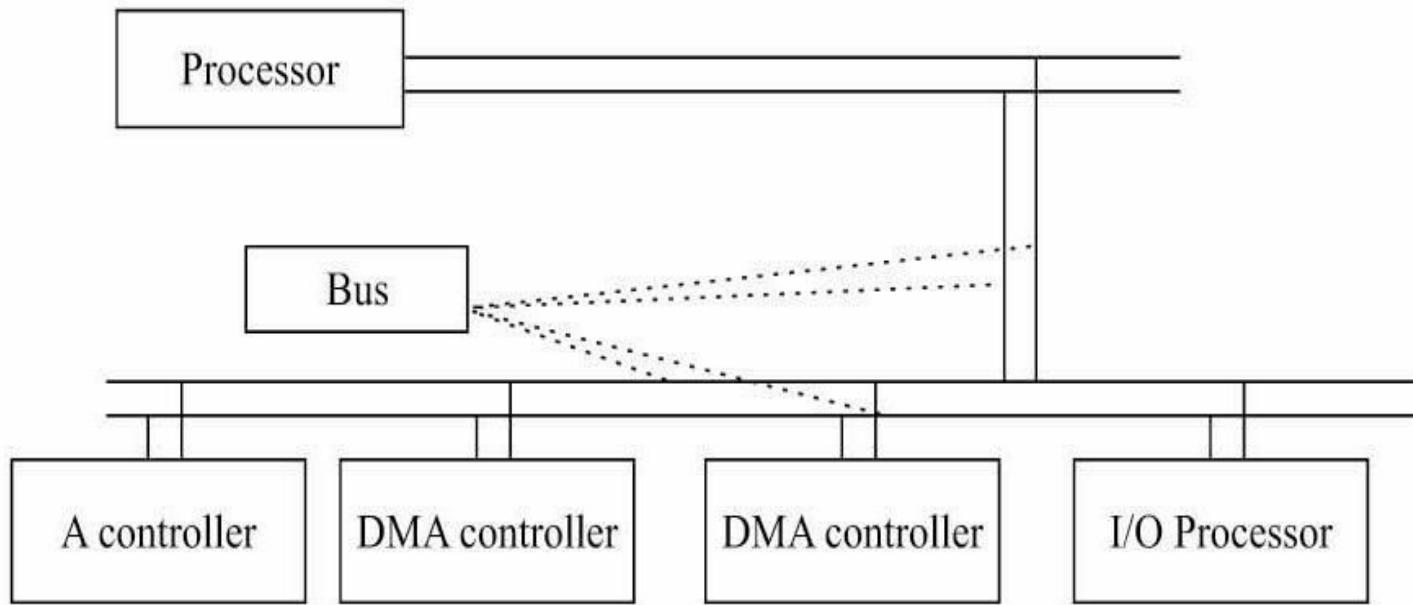
Synchronous Bus

- Synchronous bus (e.g., processor-memory buses)
 - Includes a clock in the control lines and has a fixed protocol for communication that is **relative** to the clock
 - Advantage: involves very little logic and can run very fast
 - Disadvantages:
 - Every device communicating on the bus must use same clock rate

Asynchronous Bus

- Asynchronous bus (e.g., I/O buses)
 - It is not clocked, so requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
 - Advantages:
 - Can accommodate a wide range of devices and device speeds
 - Disadvantage: slow(er)

Bus Arbitration



A bus (any bus) **cannot** be used by **more than one device** at one time due to electrical properties of the devices.

Bus Arbitration

- Before any device is allowed to perform a read/write operation using the system bus, it must first **obtain permission**.
- A device that **starts a read/write operation** is called a **master device**
- The device that it "talks" to is called the **slave device**

Bus Arbitration

- Only one processor or controller can be **bus master**
- The bus master— the controller that has access to a bus at an instance.
- Any one controller or processor can be the bus master at the given instance (s).

Bus Arbitration Methods

- **Centralized**

Centralized bus arbitration requires hardware (arbiter) that will grant the bus to one of the requesting devices. This hardware can be part of the CPU or it can be a separate device on the motherboard.

- Single hardware device controlling bus access
 - Bus Controller
 - Arbiter

- **Decentralized**

Decentralized arbitration there isn't an arbiter, so the devices have to decide who goes next. This makes the devices more complicated, but saves the expense of having an arbiter.

- Each module may claim the bus
- Control logic on all modules

Centralized Bus Arbitration

- Centralized One Level Bus Arbiter
 - This method of arbitration uses one centralized bus controller that all devices can query.
- Centralized Two Level Bus Arbiter
 - Uses a Bus Request Line and Bus Grant Line for each Level

3 BUS ARBITRATION METHODS

1. Daisy Chain
2. Independent Bus Requests and Grant
3. Polling

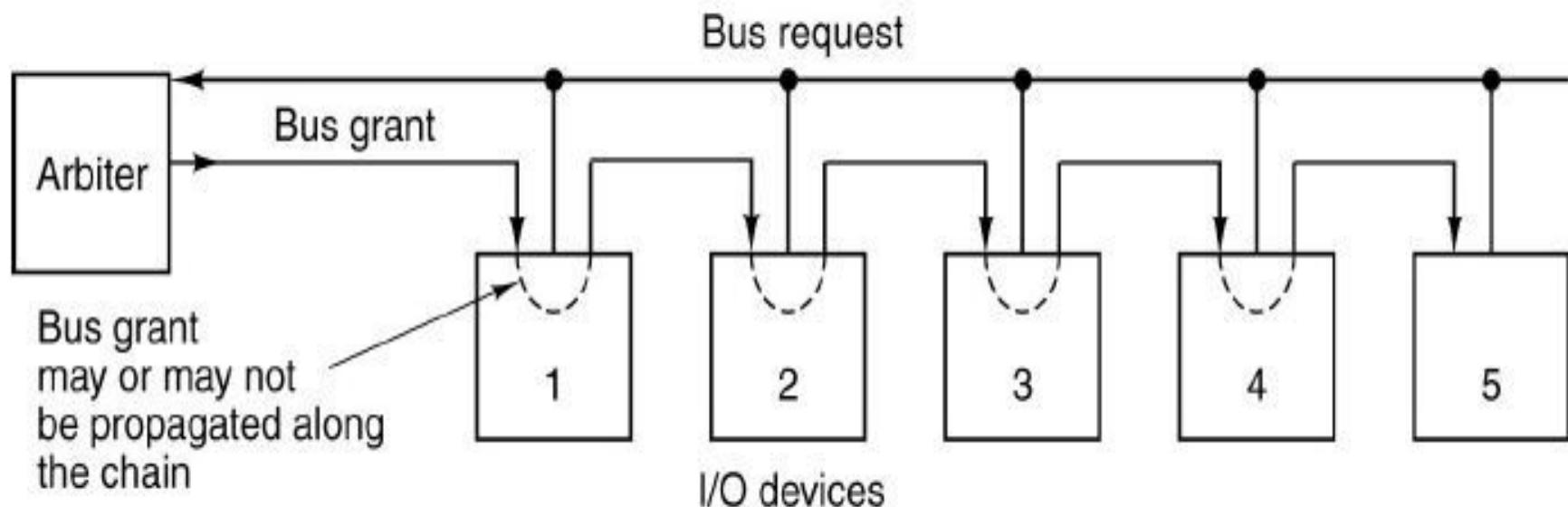
Centralized One Level Bus Arbitration

This method of arbitration uses one centralized bus controller that all devices can query. There are 2 lines that are used:

1. Bus Request Line – A wired ‘OR’ that the controller knows a request was made, but does not know which device made the request.
2. Bus Grant Line – First a signal is propagated to all devices. The Bus Grant Line is asserted to the first device in the chain. If that device made the request it takes a hold of the bus and leaves the Bus Grant Line negated for the next device in the chain. If that device didn’t make the request then the Bus Grant Line is asserted for the next device in the chain.

If two devices make a request for the bus at the same time then the device closer to the controller gets the bus. This is called *daisy chaining*

A centralized bus arbiter using daisy chaining



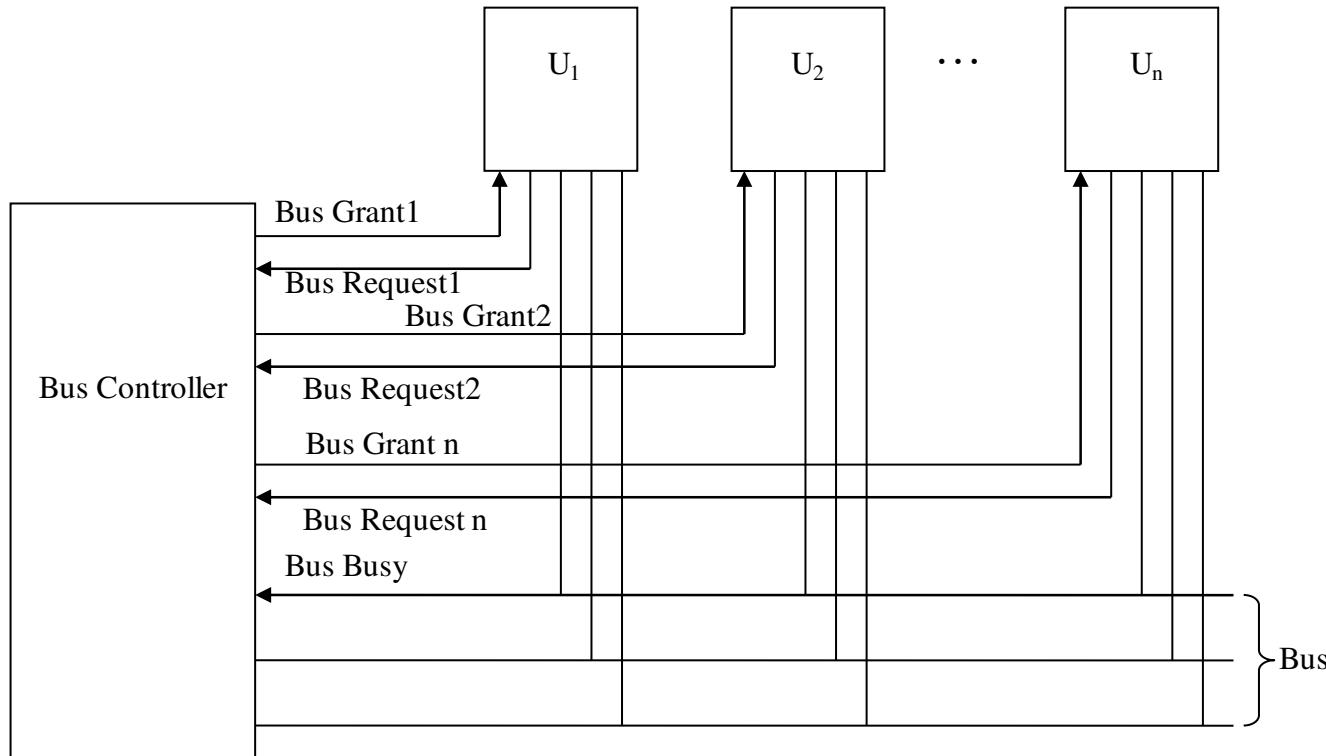
Centralized Two Level Bus Arbitration

- Centralized Two Level Bus Arbiter

- Bus Request line: one for each level
 - Bus Grant line: one for each level

This Helps to alleviate the problem of the closest device to the controller getting control of the bus. If more than one request comes in at one time, control is granted based on priority. One major advantage to this is when a lower priority device has control of the bus, a higher priority device cannot ‘steal’ the bus from that device.

Independent Requesting

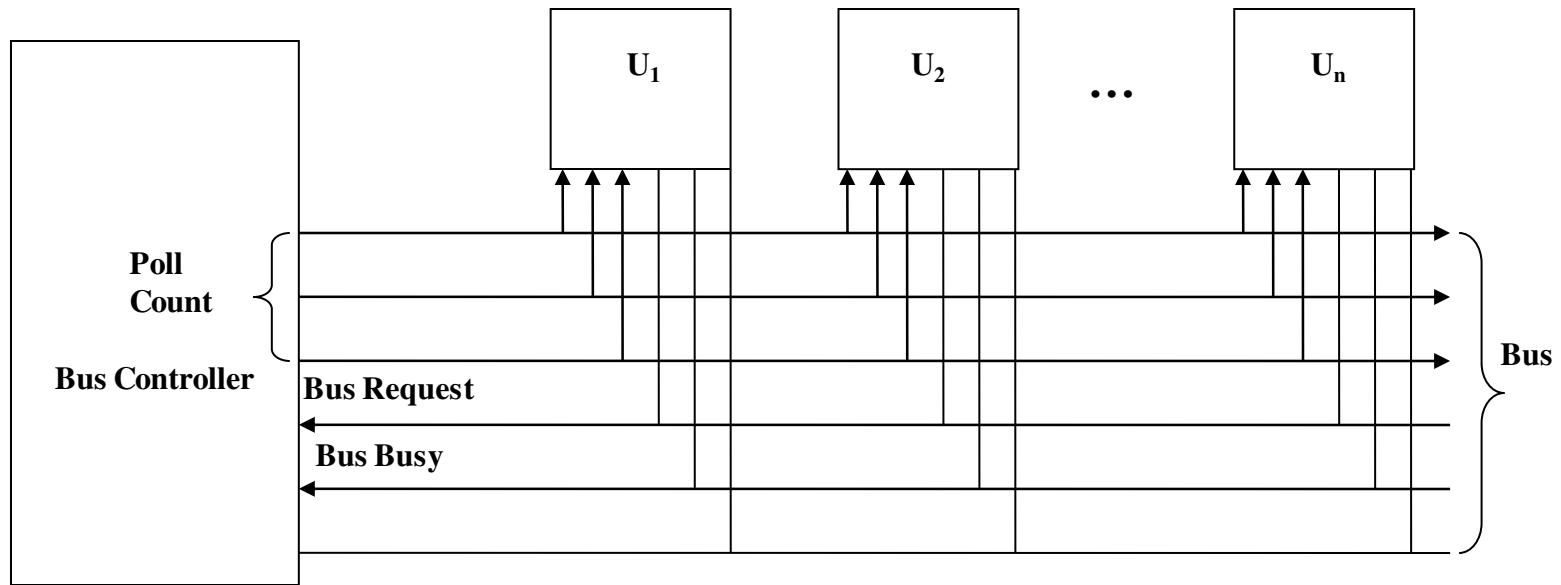


- The bus controller determines priority, which is programmable.
- Drawback: $2n$ Bus request and bus grant lines to control n devices, whereas daisy chaining requires 2 such lines and polling requires $\log_2 n$ lines approximately.

polling

- Unit requests the bus via BUS request line.
- In response, the Bus controller proceeds to generate a sequence of numbers on the poll-count lines.
- Each unit compares to the unique address assigned to it.
- When requesting unit finds the match, Bus Busy signal is activated.
- In response, bus controller terminates the polling process and U_i connects to the bus.
- Advantage: failure of one unit need not affect the other units.
- Disadvantage: expensive because of more control lines. Number of units are limited based on the poll-count lines capability

Polling



Daisy chaining, the centralized controller always sending bus control to highest priority, which passes it to next if bus access not required.

Independent request method, the centralized controller listens to requests of each device individually and grant access to the bus on resolving its priority.

Polling method, the centralized controller does the polling of the devices and grant access to that bus which requests it on receiving the poll address.

References

- William Stallings “Computer Organization and architecture”, Prentice Hall, 7th edition, 2006.
- M. M. Mano, Computer System Architecture, Prentice-Hall

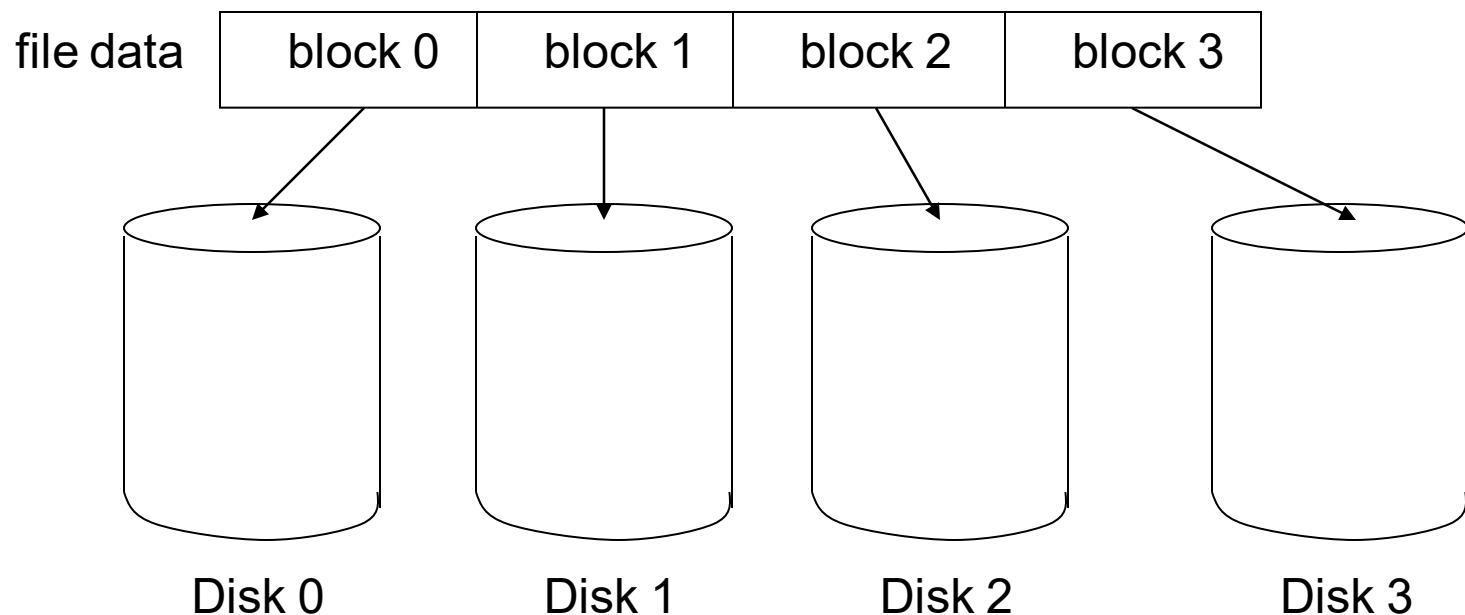
RAID ARCHITECTURE

RAID (Redundant Array of Independent Disks)

- Redundant array of inexpensive disks
- Multiple disk database design
- Set of physical disk drives viewed by the OS as a single logical drive
- Data are distributed across the physical drives of an array
- Improve access time and improve reliability
 - large storage capacity
 - redundant data
 - 7 levels (6 levels in common use)
 - differing levels of redundancy, error checking, capacity, and cost

Striping

- Take file data and map it to different disks
- Allows for reading data in parallel



Parity

- Way to do error checking and correction
- Add up all the bits that are 1
 - if even number, set parity bit to 0
 - if odd number, set parity bit to 1
- To actually implement this, do an exclusive OR of all the bits being considered
- Consider the following 2 bytes

<u>byte</u>	<u>parity</u>
10110011	1
01101010	0

- If a single bit is bad, it is possible to detect it

Mirroring

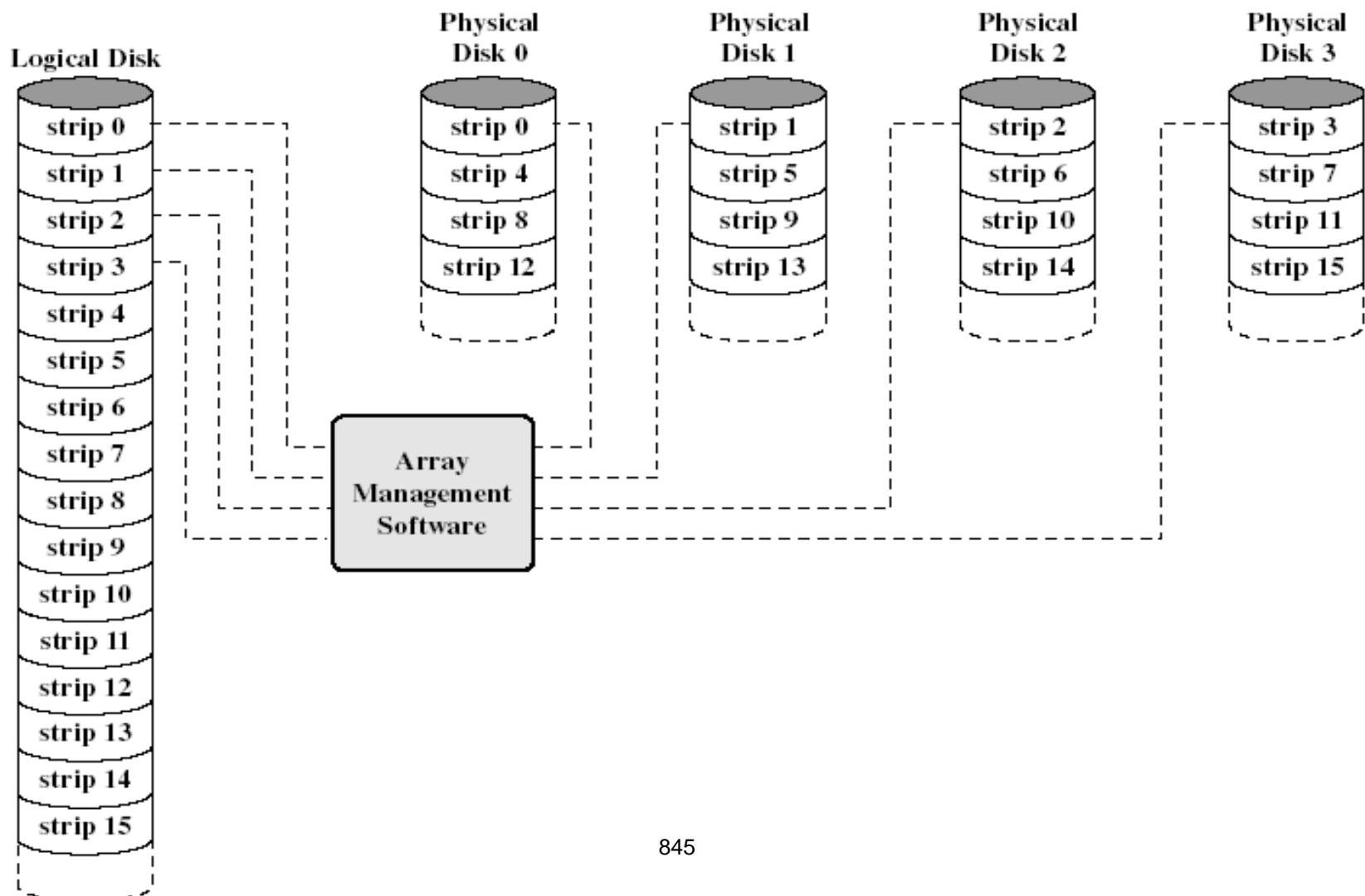
- Keep two copies of data on two separate disks
- Gives good error recovery
 - if some data is lost, get it from the other source
- Expensive
 - requires twice as many disks
- Write performance can be slow
 - have to write data to two different spots
- Read performance is enhanced
 - can read data from file in parallel

RAID Level 0

- Not a true member of the RAID family - does not include redundancy to improve performance.
- User and system data distributed across all disks in the array in strips.
- Imagine a large logical disk containing ALL data. This is divided into strips (physical blocks or sectors) that are mapped ‘round robin’ to the strips in the array.
- A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a stripe.
- + If two different I/O requests are pending for two different blocks of data – then there is a good chance that the data will be on different disks and can be serviced in parallel.
- + If a single I/O request is for multiple logically continuous strips – up to n strips can be handled in parallel.

Data Mapping for RAID Level 0

- This software may execute either in the disk sub system or in a host computer



RAID -0

- **Recommended Applications**

Very good performance, but at a price...

- Video Production and Editing
- Image Editing
- Any application requiring high bandwidth

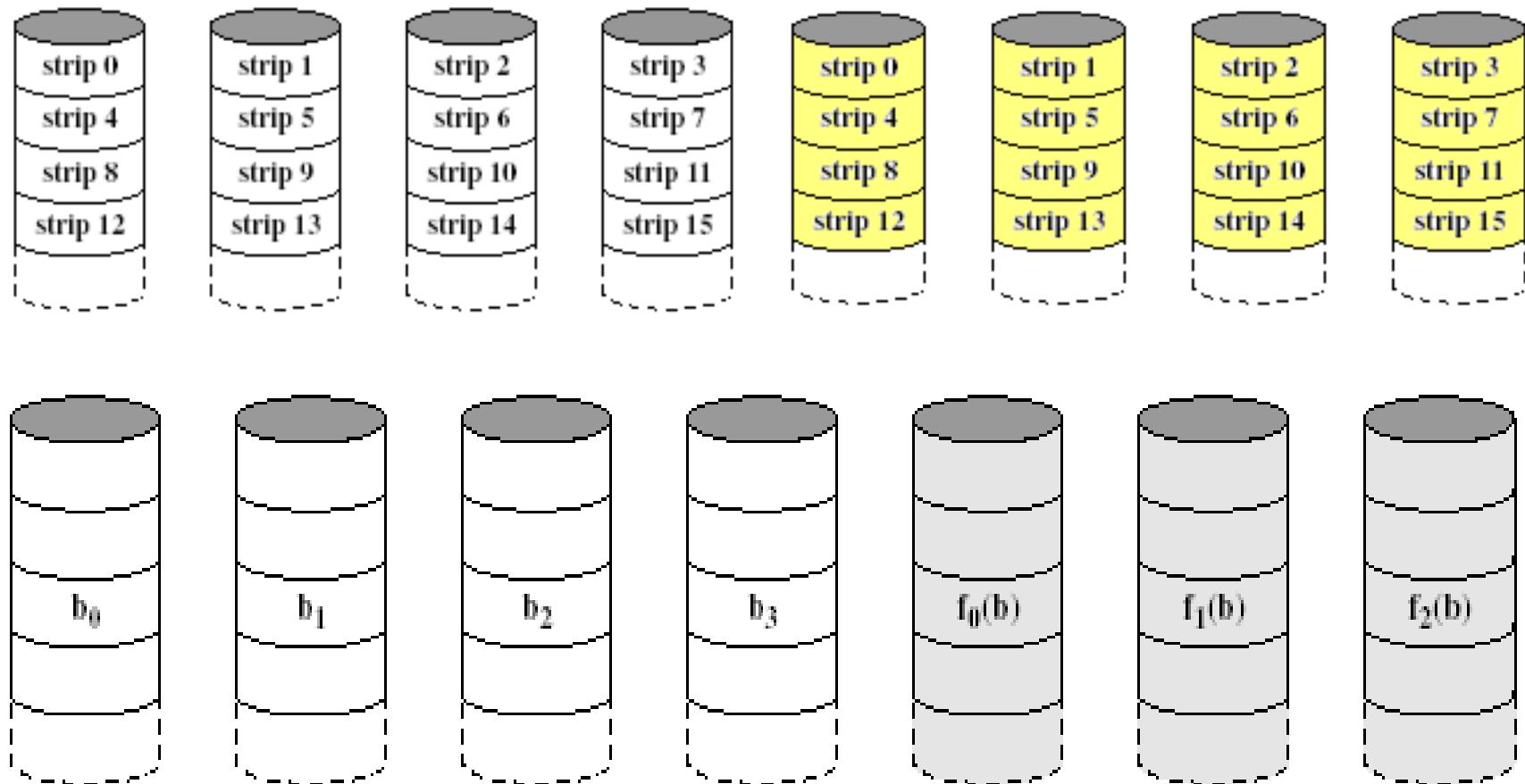
RAID Level 1

- Redundancy achieved through duplicating complete file on disks.
- Data stripping is similar to RAID level 0.
- Each logic strip is mapped to two physical disks.
 - Read request can be serviced from either of available 2 disks, which ever involves the minimum seek time and rotational latency
 - Write request requires both disks to be updated – but this can be done in parallel. (Slower write dictates overall speed).
 - Recover from failure is simple! (data may still be accessed from the second drive)
- Disadvantage:
 - Cost
 - requires twice the disk space
 - Configuration is limited, so used only for system software and other highly critical files.
- Improvement occurs if the application can split each read request so that both disk members participate

RAID - 1

- **Recommended Applications**
 - Accounting
 - Payroll
 - Financial
 - Any application requiring very high availability

RAID Level 1 (Mirroring)



(e) RAID 2 (redundancy through Hamming code)

RAID Level 2

- ▶ Utilizes parallel access techniques - All disks participate in the execution of every I/O request.
- ▶ Spindles of individual drives are synchronized so that each disk head is in the same position on each disk at any given time.
- ▶ Data striping – very small strips (bit level striping).
- ▶ Error correcting code is calculated across corresponding bits on each disk, and the code bits are stored in corresponding bit positions on multiple parity disks.
- ▶ For Hamming Code – number of parity (redundant) disks is proportionate to the log of the number of data disks.
- ▶ On a single read, all disks are simultaneously accessed. The requested data and the associated error correcting code are delivered to the array controller. Array controller can detect and fix single bit errors.
- ▶ For write – all disks must be accessed.
- ▶ Adv: Requires fewer disks than level 1
- ▶ Big problem is performance
 - must read data plus ECC code from other disks
 - for a write, have to modify data, ECC, and parity disks

RAID Level 3

Byte level striping with parity

Similar to RAID 2 – parallel access with data distributed in small strips.

Requires only a single redundant disk because it uses a single parity bit for the set of individual bits in the same position on all of the data disks.

If drives X0-X3 contain data, and X4 contains parity bits.

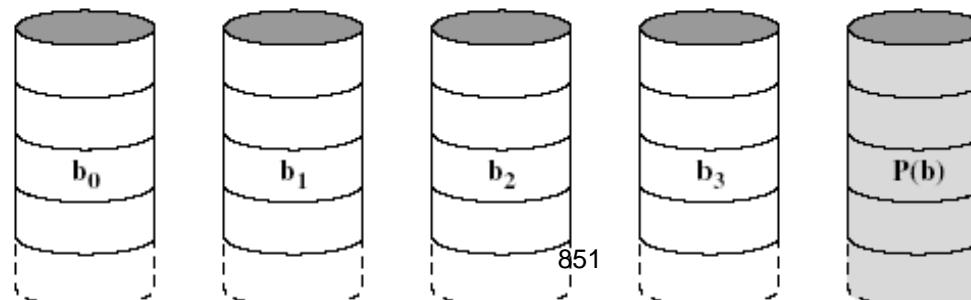
$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

Redundancy – in the case of disk failure, the data can be reconstructed.

If drive X1 fails – it can be reconstructed as:

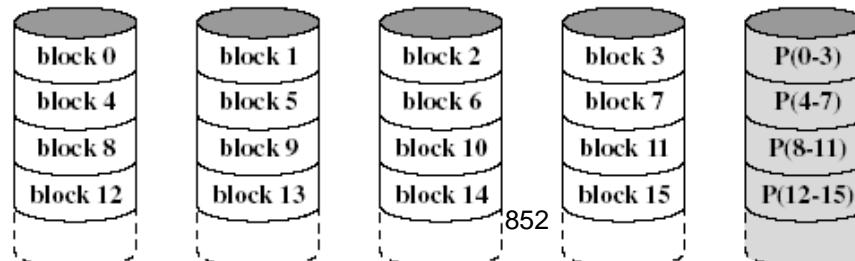
$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Performance – can achieve high transfer rates, but only one I/O request can be executed at one time. (Better for large data transfers in non transaction-oriented environments).



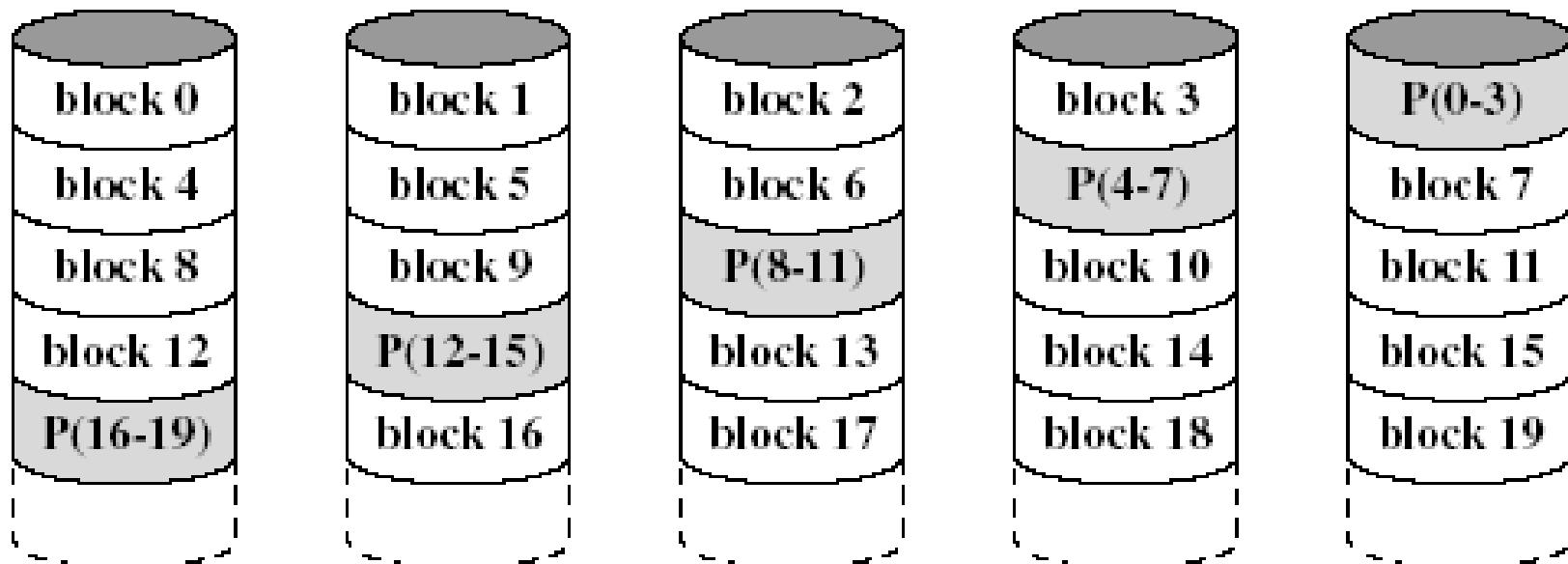
RAID Level 4

- ▶ Each disk operates independently - Separate I/O requests satisfied in parallel.
- ▶ Suitable for applications with high I/O request rates and NOT well suited for those requiring high data transfer rates.
- ▶ Data striping – Block level. (Strips are larger than in lower RAIDs) .
- ▶ Bit-by-bit parity strip is calculated across corresponding strips on each data disk, and stored in corresponding strip on the parity disk.
- ▶ Performance – write penalty when I/O request is small size. Write must update user data + corresponding parity bits.
- ▶
$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$
- ▶ If $X1(i)$ is changed to $X1'(i)$
- ▶
$$X4(i) = X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) = X4(i) \oplus X1(i) \oplus X1'(i)$$
- ▶ To calculate new parity, the old user data and old parity strips must be read. Then it can update these two strips with the new data and the newly calculated parity. Thus each strip write involves two reads and two writes.
- ▶ **Adv:** Level-4 interleaves file blocks



RAID Level 5

- Same as RAID 4 – but parity strips are distributed across all disks.
- Typical allocation uses round-robin.
- For an n-disk array, the parity strip is on a different disk for the first n strips and the pattern then repeats.
- Avoids potential bottleneck found in RAID 4.

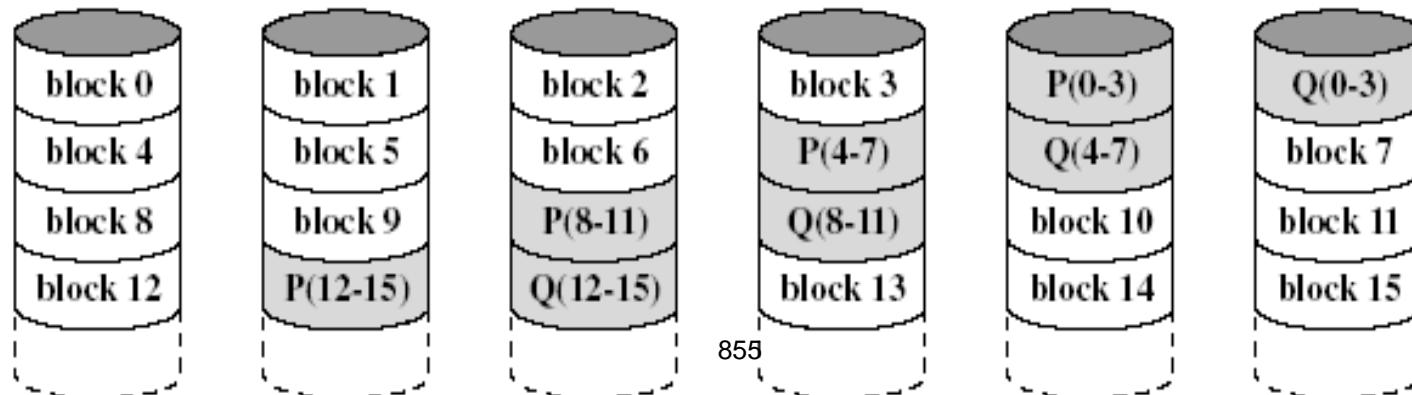


RAID - 5

- **Recommended Applications**
 - File and Application servers
 - Database servers
 - WWW, E-mail, and News servers
 - Intranet servers
 - Most versatile RAID level

RAID Level 6

- Two different parity calculations are carried out and stored in separate blocks on different disks.
 - Example: XOR and an independent data check algorithm => makes it possible to regenerate data even if two disks containing user data fail.
- No. of disks required = $N + 2$ (where N = number of disks required for data).
- Provides HIGH data availability.
- Incurs substantial write penalty as each write affects two parity blocks.
- Three disks would have to fail within MTTR (mean time to repair) interval to cause data to be lost



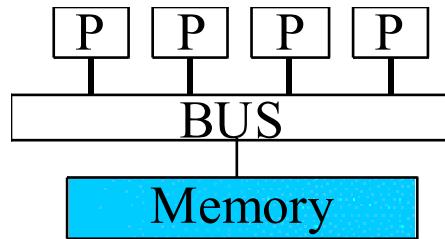
Comparison of RAID Levels

Category	Level	Description	I/O request rate (Read/Write)	Data Transfer Rate (Read/Write)	Typical application
Stripping	0	Nonredundant	large strips: excellent	small strips: excellent	applications requiring high performance for non-critical data
Mirroring	1	Mirrored	Good/Fair	Fair/fair	System drives; critical files
Parallel access	2	Bit-level striping with hamming code parity	poor	excellent	
	3	Byte-level striping with dedicated parity	poor	excellent	large I/O request size applications, such as imaging, CAD
Independent access	4	Block level striping with dedicated parity	excellent/fair	fair/poor	
	5	Block-level striping with distributed parity	excellent/fair	fair/poor	high request rate, read intensive, data lookup
	6	block-level striping with double distributed parity	excellent/poor	fair/poor	applications requiring extreamly high availability

References

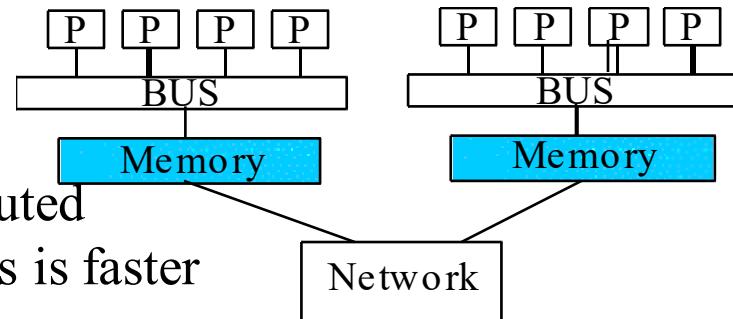
- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

Shared Memory: UMA and NUMA



Uniform memory access (UMA)
 Each processor has uniform access time
 to memory - also known as
 symmetric multiprocessors (SMPs)
 (example: SUN ES1000)

Non-uniform memory access (NUMA)
 Time for memory access depends on
 location of data; also known as Distributed
 Shared memory machines. Local access is faster
 than non-local access. Easier to scale
 than SMPs (example: SGI Origin 2000)



SMPs: Memory Access Problems

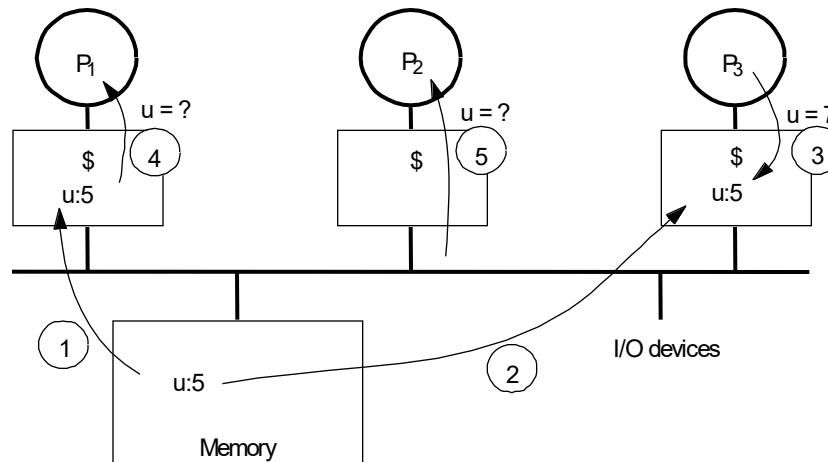
- **Conventional wisdom is that systems do not scale well**
 - Bus based systems can become saturated
 - Fast large crossbars are expensive

SMPs: Memory Access Problems

- **Cache Coherence** : When a processor modifies a shared variable in local cache, different processors have different value of the variable. Several mechanism have been developed for handling the cache coherence problem
- **Cache coherence problem**
 - Copies of a variable can be present in multiple caches
 - A write by one processor may not become visible to others
 - They'll keep accessing stale value in their caches
 - Need to take actions to ensure visibility or cache coherence

Cache Coherence Problem

- Processors see different values for u after event 3
- Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!



Snooping-Based Coherence

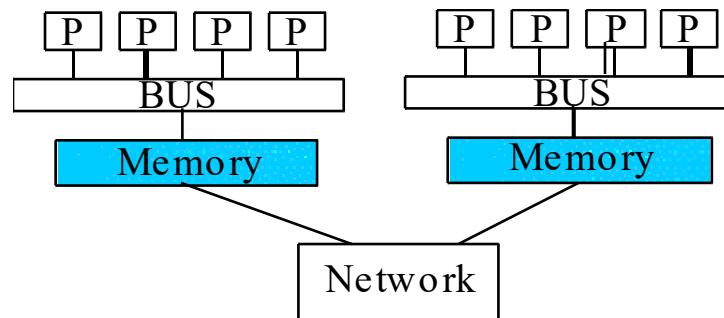
- **Basic idea:**
 - Transactions on memory are visible to all processors
 - Processor or their representatives can snoop (monitor) bus and take action on relevant events
- **Implementation**
 - When a processor writes a value a signal is sent over the bus
 - Signal is either
 - Write invalidate - tell others cached value is invalid and then update
 - Write broadcast/update - tell others the new value continuously as it is updated

SMP Machines

- **Various Suns such as the E10000/HPC10000**
- **Various Intel and Alpha servers**
- **Crays: T90, J90, SV1**

“Distributed-shared” Memory (NUMA)

- **Consists of N processors and a global address space**
 - All processors can see all memory
 - Each processor has some amount of local memory
 - Access to the memory of other processors is slower
 - Cache coherence ('cc') must be maintained across the network as well as within nodes
- **NonUniform Memory Access**



cc-NUMA Memory Issues

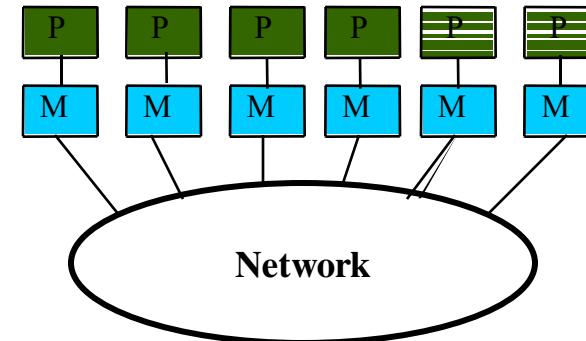
- Easier to build with same number of processors because of slower access to remote memory (crossbars/buses can be less expensive than if all processors are in an SMP box)
- Possible to created much larger shared memory system than in SMP
- Similar cache problems as SMPs, but coherence must be also enforced across network now!
- Code writers should be aware of data distribution
 - Load balance
 - Minimize access of "far" memory

cc-NUMA Machines

- **SGI Origin 2000**
- **HP X2000, V2500**

Distributed Memory

- **Each of N processors has its own memory**
- **Memory is not shared**
- **Communication occurs using messages**
 - Communication networks/interconnects
- **Custom**
 - Many manufacturers offer custom interconnects
- **Off the shelf**
 - Ethernet
 - ATM
 - FIBER Channel
 - FDDI



Types of Distributed Memory Machine Interconnects

- **Fully connected**
- **Array and torus**
 - Paragon
 - T3E
- **Crossbar**
 - IBM SP
 - Sun HPC10000
- **Hypercube**
 - Ncube
- **Fat Trees**
 - Meiko CS-2

Multi-tiered/Hybrid Machines

- **Clustered SMP nodes ('CLUMPS'): SMPs or even cc-NUMAs connected by an interconnect network**
- **Examples systems**
 - All new IBM SP
 - Sun HPC1000s using multiple nodes
 - SGI Origin 2000s when linked together
 - HP Exemplar using multiple nodes
 - SGI SV1 using multiple nodes

Reasons for Each System

- **SMPs (Shared memory machine): easy to build, easy to program, good price-performance for small numbers of processors; predictable performance due to UMA**
- **cc-NUMAs (Distributed Shared memory machines) : enables larger number of processors and shared memory address space than SMPs while still being easy to program, but harder and more expensive to build**

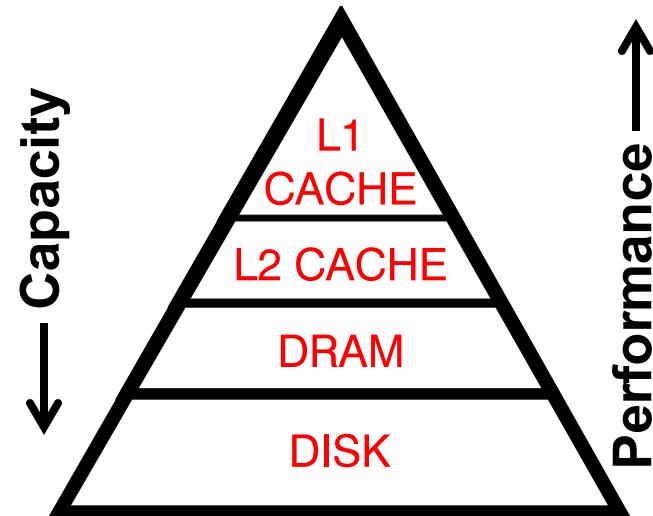
Magnetic Disk

Disk-based storage in computers

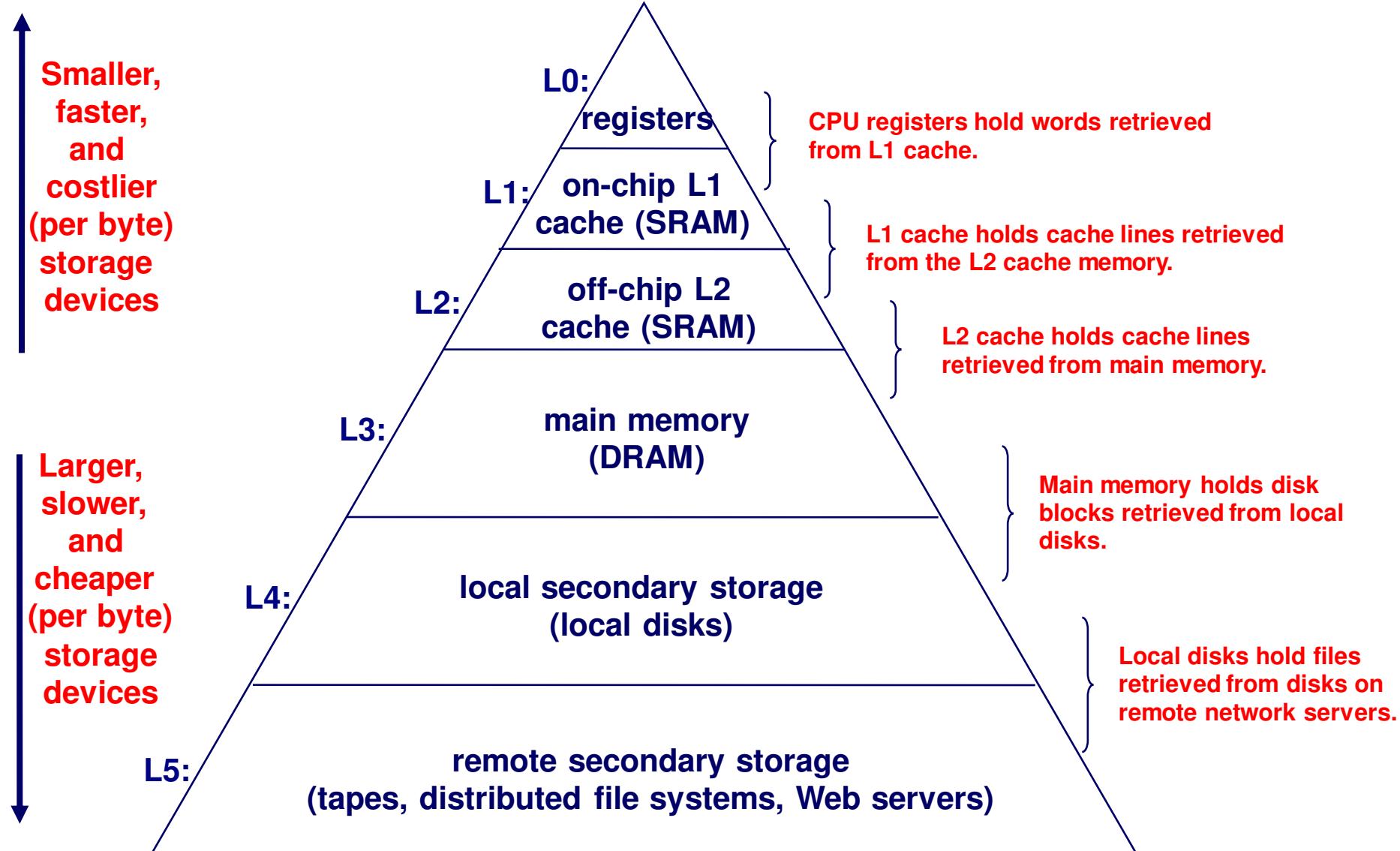
- **Memory/storage hierarchy**
 - Combining many technologies to balance costs/benefits
 - Recall the memory hierarchy and virtual memory lectures

Memory/storage hierarchies

- **Balancing performance with cost**
 - Small memories are **fast but expensive**
 - Large memories are **slow but cheap**
- **Exploit locality to get the best of both worlds**
 - locality = re-use/nearness of accesses
 - allows most accesses to use small, fast memory



An Example Memory Hierarchy



Disk-based storage in computers

- **Memory/storage hierarchy**
 - Combining many technologies to balance costs/benefits
 - Recall the memory hierarchy and virtual memory
- **Persistence**
 - Storing data for lengthy periods of time
 - DRAM/SRAM is “volatile”: contents lost if power lost
 - Disks are “non-volatile”: contents survive power outages
 - To be useful, it must also be possible to find it again later
 - this brings in many interesting data organization, consistency, and management issues

What's Inside A Disk Drive?

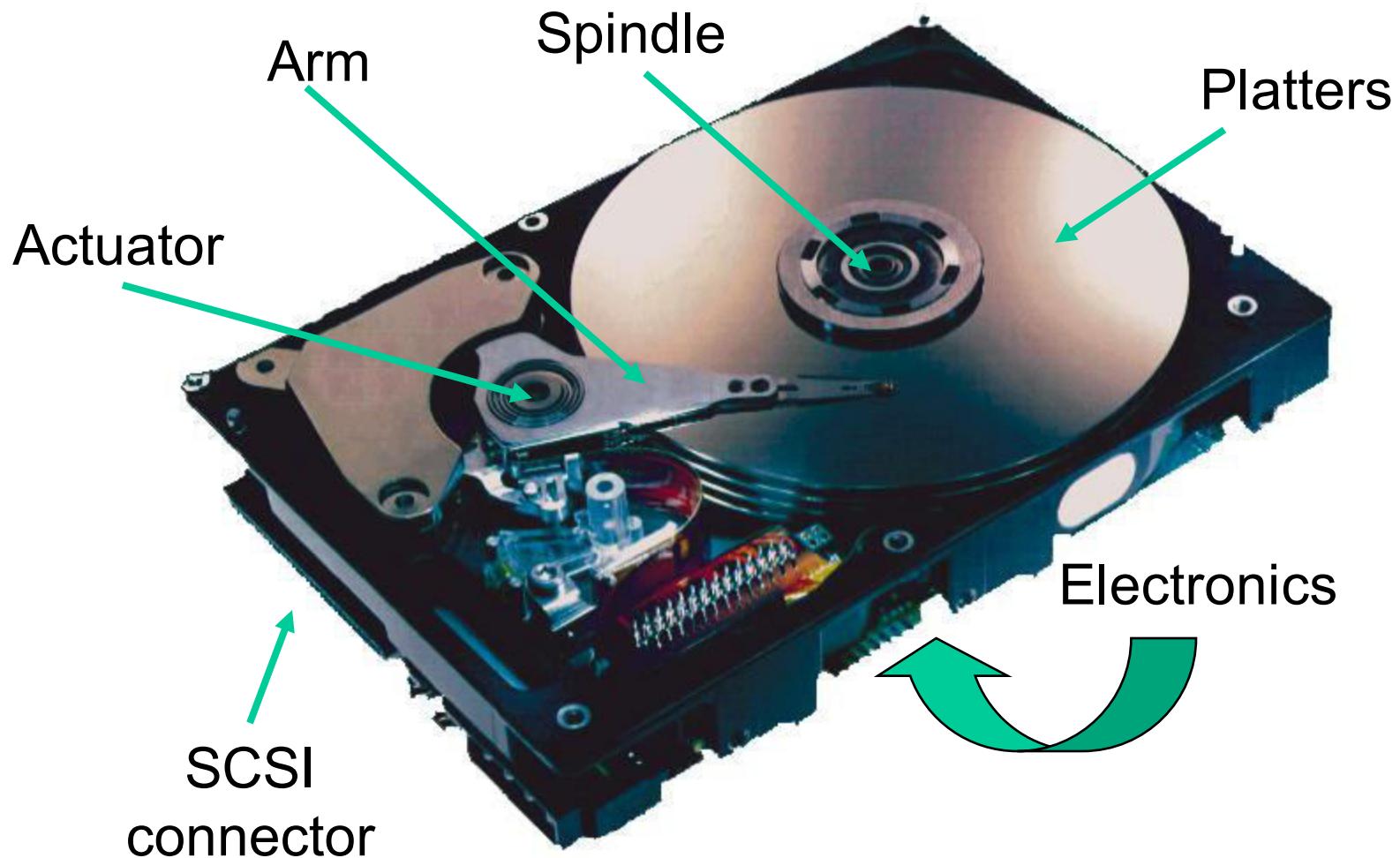
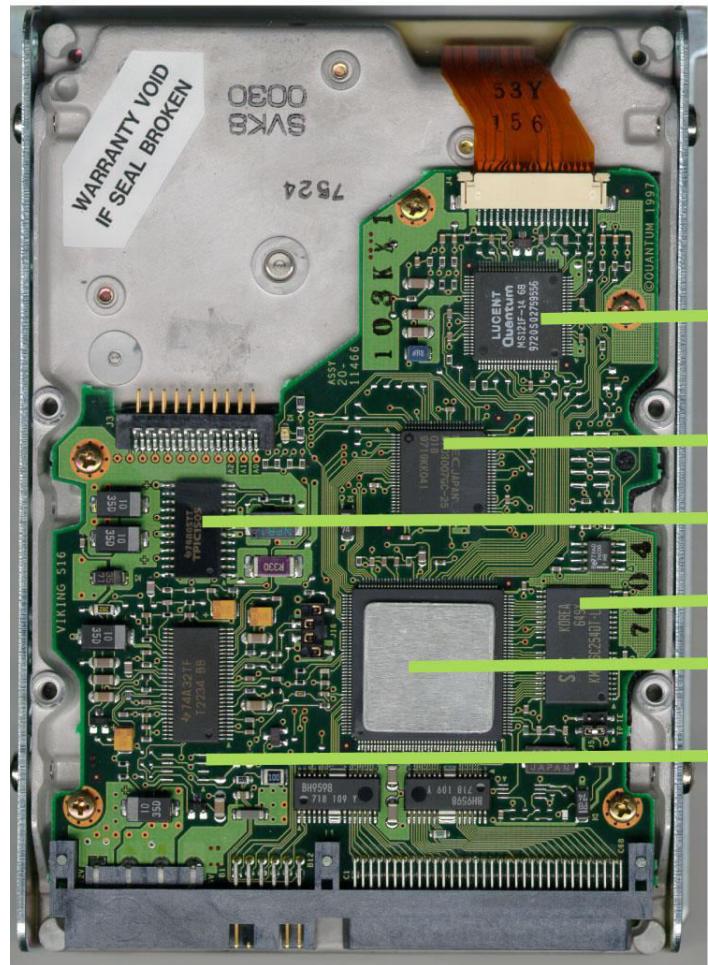


Image courtesy of Seagate Technology

Disk Electronics

Quantum Viking (circa 1997)



6 Chips

R/W Channel

uProcessor

32-bit, 25 MHz

Power Array

2 MB DRAM

Control ASIC
SCSI, servo, ECC

Motor/Spindle

Just like a small computer – processor, memory, network iface

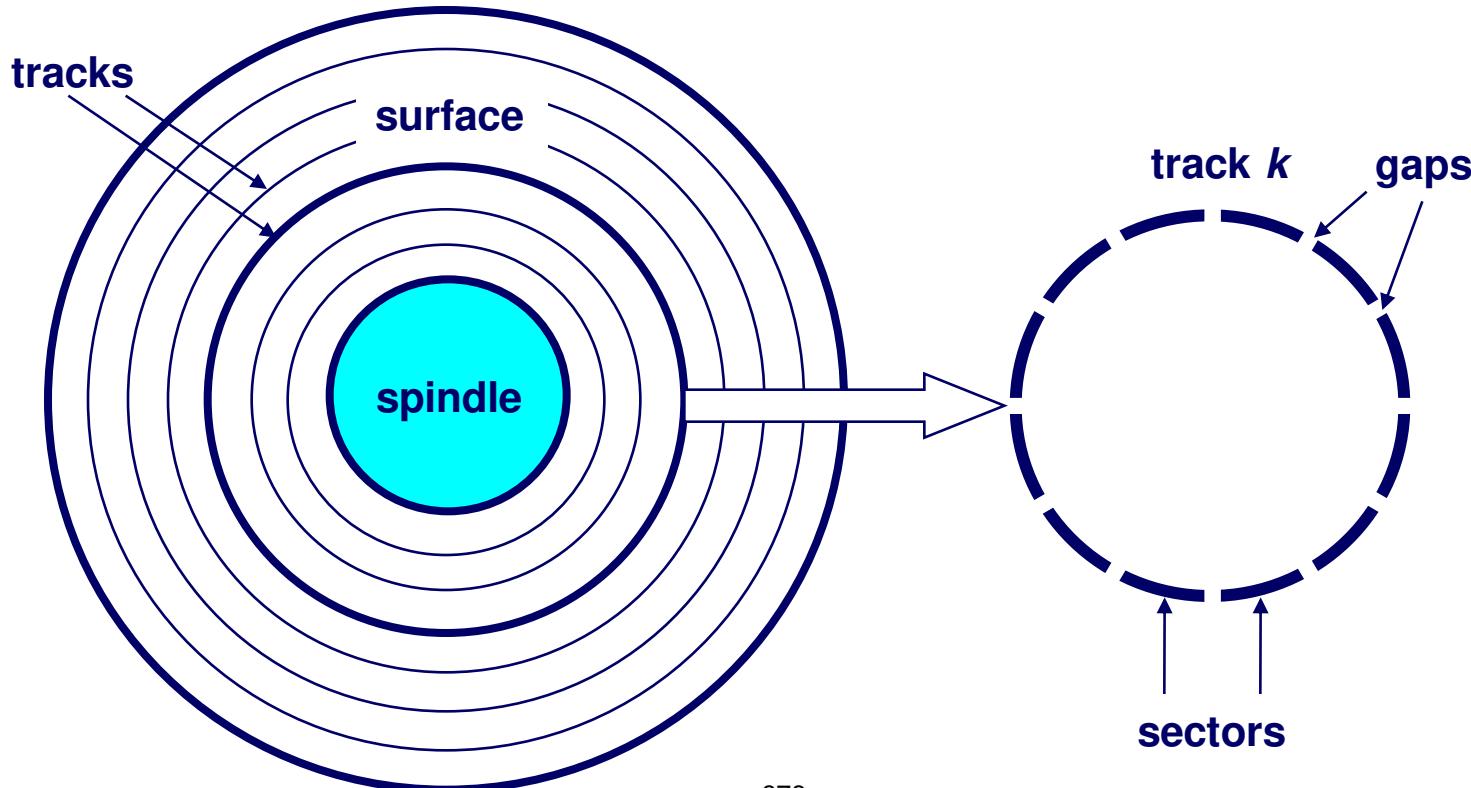
- Connect to disk
- Control processor
- Cache memory
- Control ASIC
- Connect to motor

Disk “Geometry”

Disks contain platters, each with two surfaces

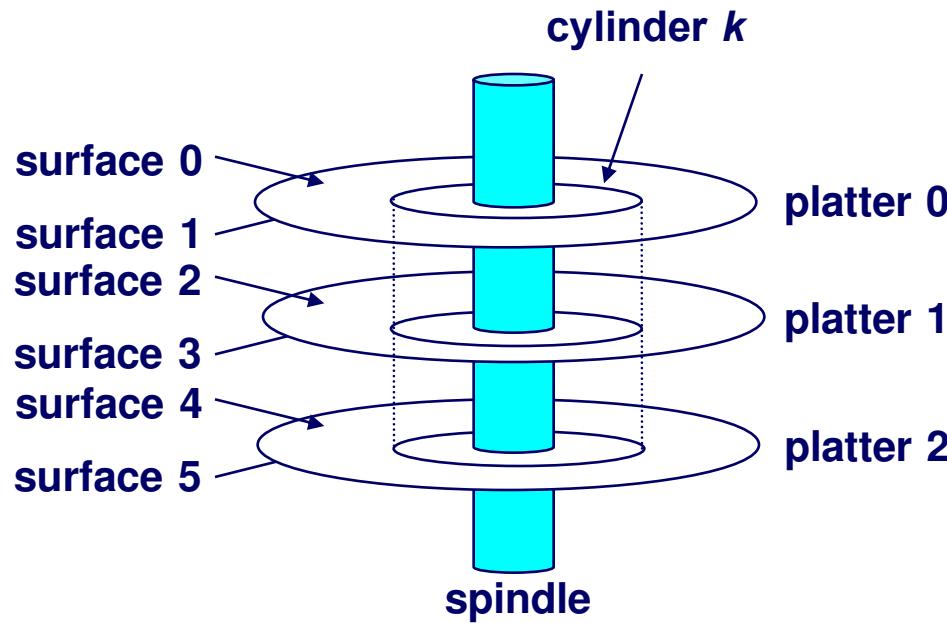
Each surface organized in concentric rings called tracks

Each track consists of sectors separated by gaps

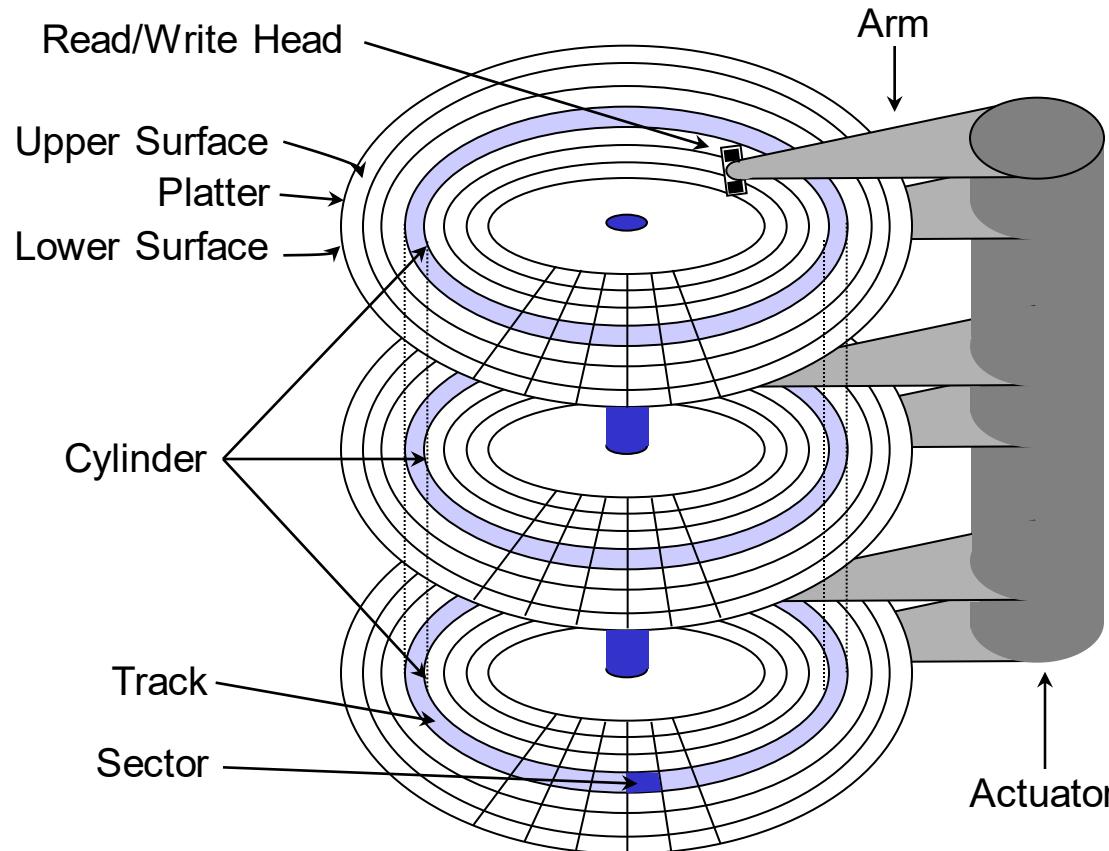


Disk Geometry (Multiple-Platter View)

Aligned tracks form a cylinder

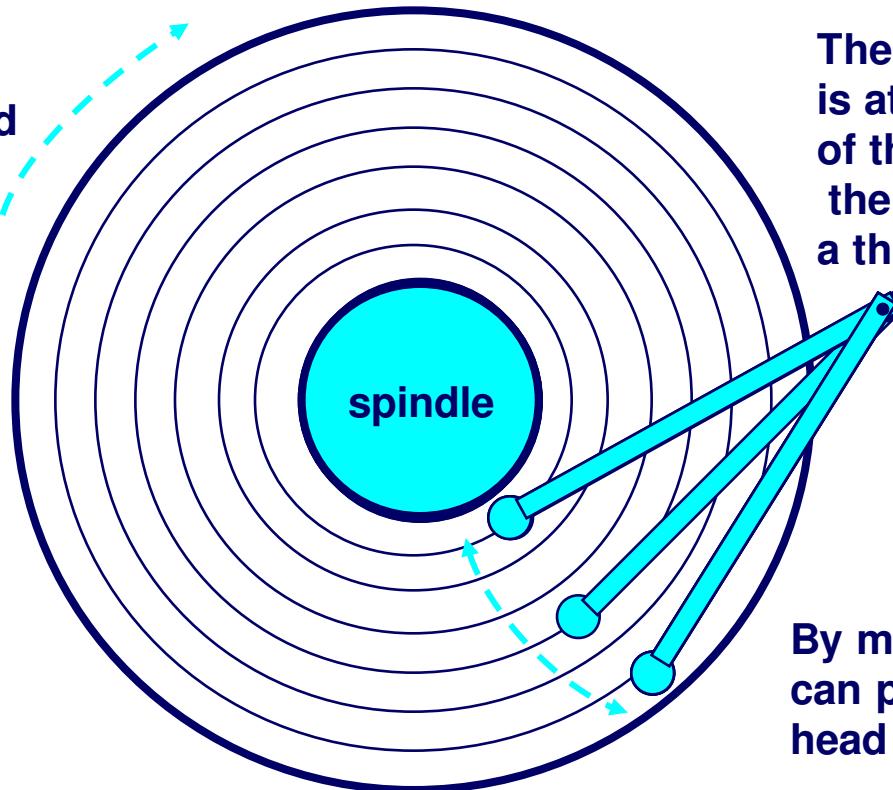


Disk Structure



Disk Operation (Single-Platter View)

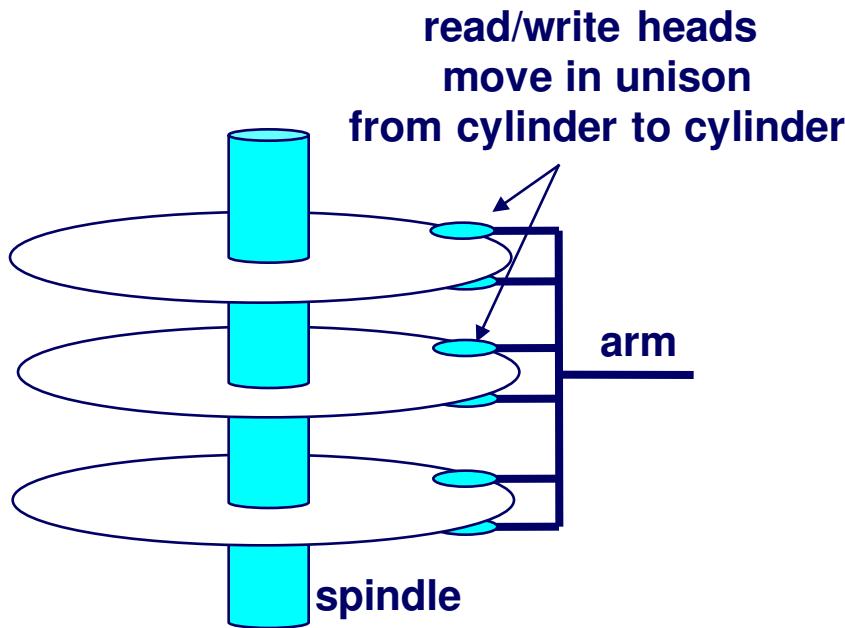
The disk surface spins at a fixed rotational rate



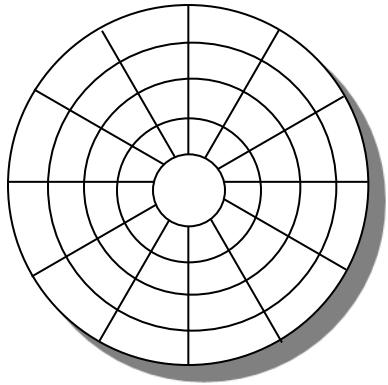
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air

By moving radially, the arm can position the read/write head over any track

Disk Operation (Multi-Platter View)



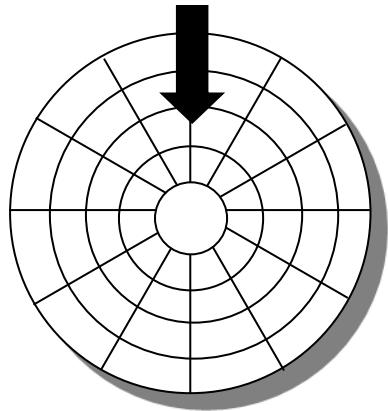
Disk Structure - top view of single platter



Surface organized into tracks

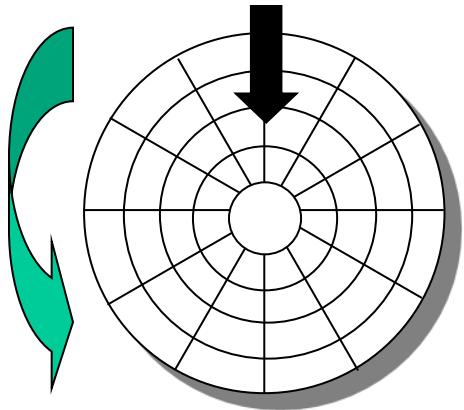
Tracks divided into sectors

Disk Access



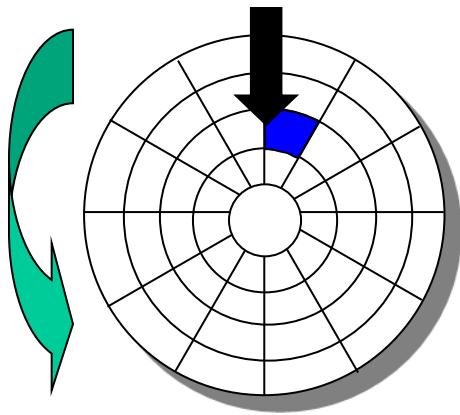
Head in position above a track

Disk Access



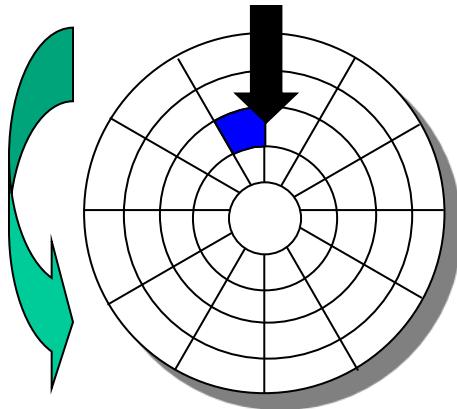
Rotation is counter-clockwise

Disk Access – Read



About to read blue sector

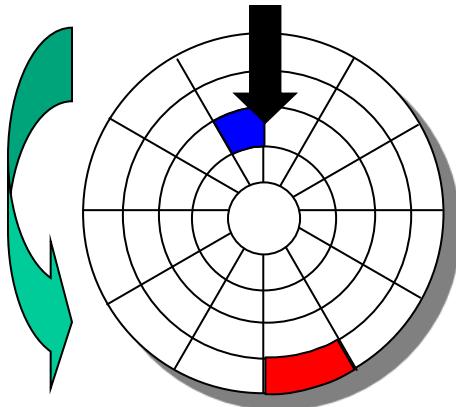
Disk Access – Read



After BLUE read

After reading blue sector

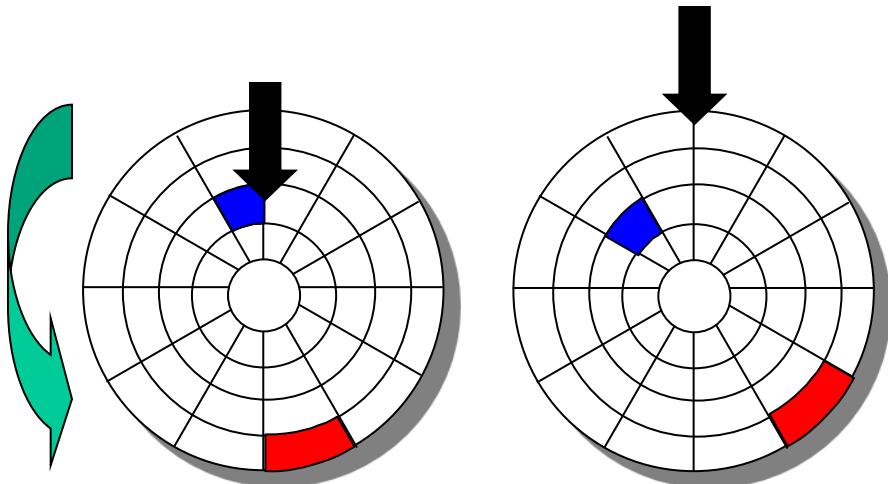
Disk Access – Read



After **BLUE** read

Red request scheduled next

Disk Access – Seek

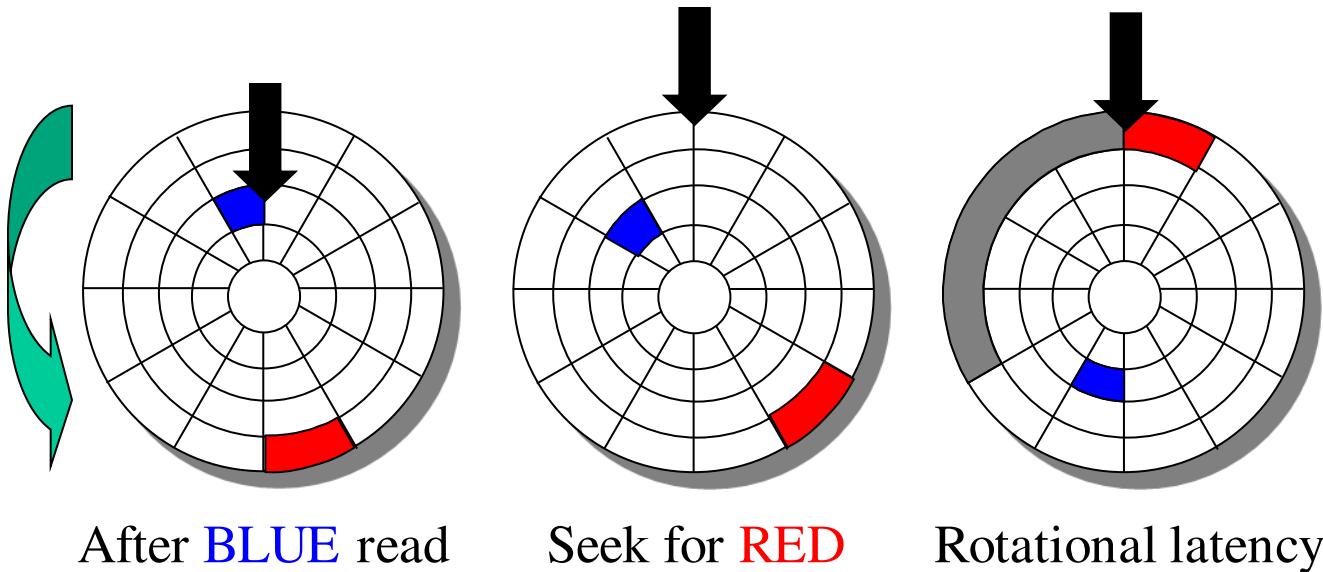


After **BLUE** read

Seek for **RED**

Seek to red's track

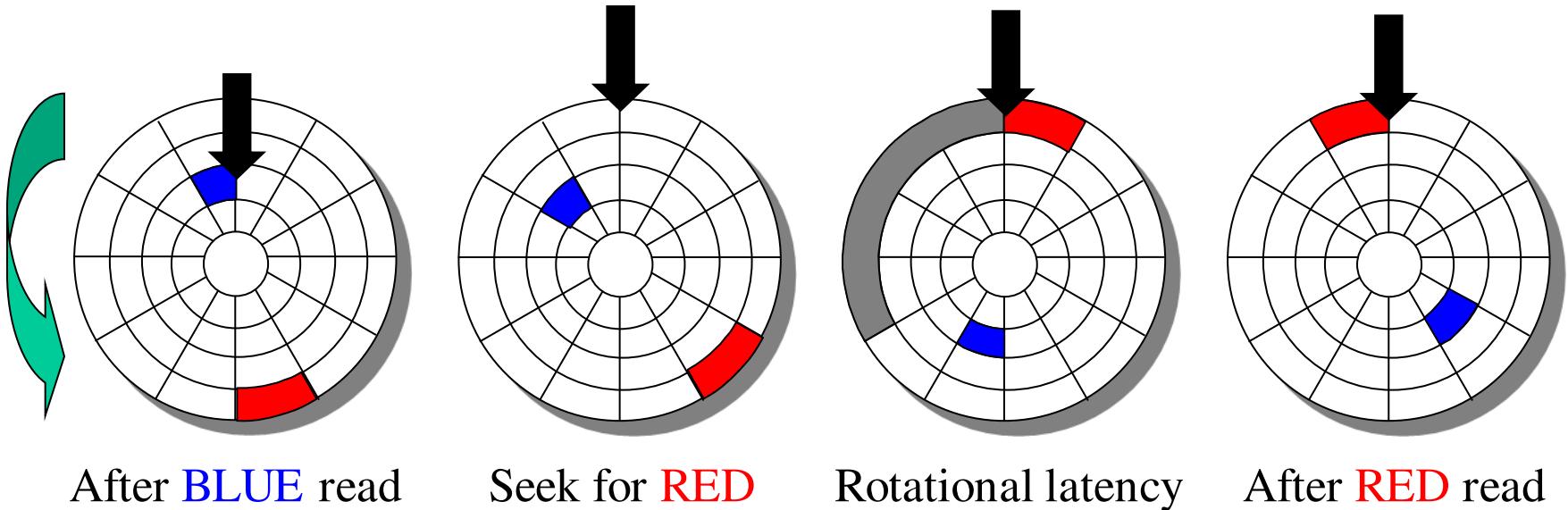
Disk Access – Rotational Latency

After **BLUE** readSeek for **RED**

Rotational latency

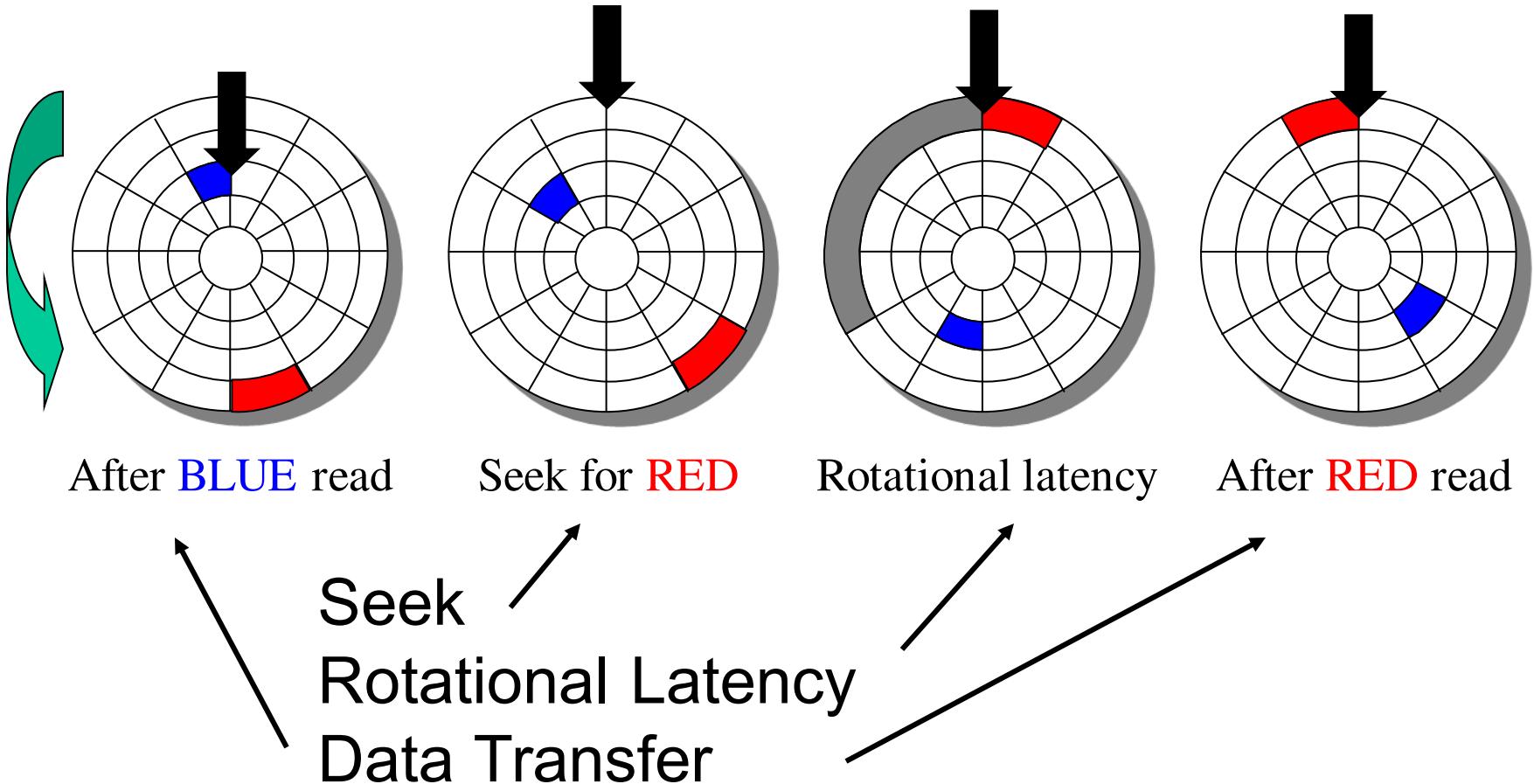
Wait for red sector to rotate around

Disk Access – Read



Complete read of red

Disk Access – Service Time Components



Disk Access Time

Average time to access a specific sector approximated by:

- $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$

Seek time (T_{avg} seek)

- Time to position heads over cylinder containing target sector
- Typical $T_{avg\ seek} = 3-5\ ms$

Rotational latency (T_{avg} rotation)

- Time waiting for first bit of target sector to pass under r/w head
- $T_{avg\ rotation} = 1/2 \times 1/\text{RPMs} \times 60\ sec/1\ min$
 - e.g., 3ms for 10,000 RPM disk

Transfer time (T_{avg} transfer)

- Time to read the bits in the target sector
- $T_{avg\ transfer} = 1/\text{RPM} \times 1/(\text{avg\ # sectors/track}) \times 60\ secs/1\ min$
 - e.g., 0.006ms for 10,000 RPM disk with 1,000 sectors/track
 - given 512-byte sectors, ~85 MB/s data transfer rate

Disk Access Time Example

Given:

- Rotational rate = 7,200 RPM
- Average seek time = 5 ms
- Avg # sectors/track = 400

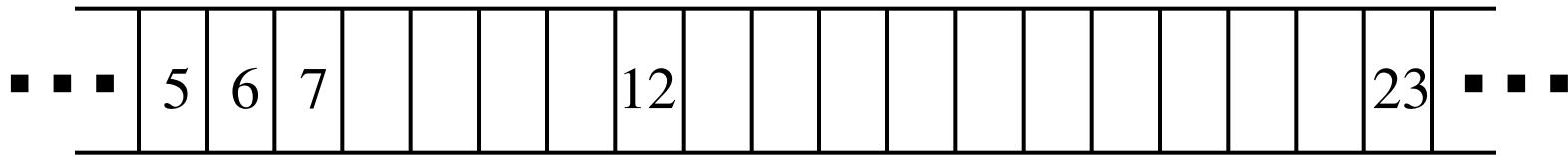
Derived average time to access random sector:

- $T_{avg\ rotation} = 1/2 \times (60\ secs/7200\ RPM) \times 1000\ ms/sec = 4\ ms$
- $T_{avg\ transfer} = 60/7200\ RPM \times 1/400\ secs/track \times 1000\ ms/sec = 0.008\ ms$
- $T_{access} = 5\ ms + 4\ ms + 0.008\ ms = 9.008\ ms$
 - Time to second sector: 0.008 ms

Important points:

- Access time dominated by seek time and rotational latency
- First bit in a sector is the most expensive, the rest are free
- SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
 - ~100,000 times longer to access a word on disk than in DRAM

Disk storage as array of blocks



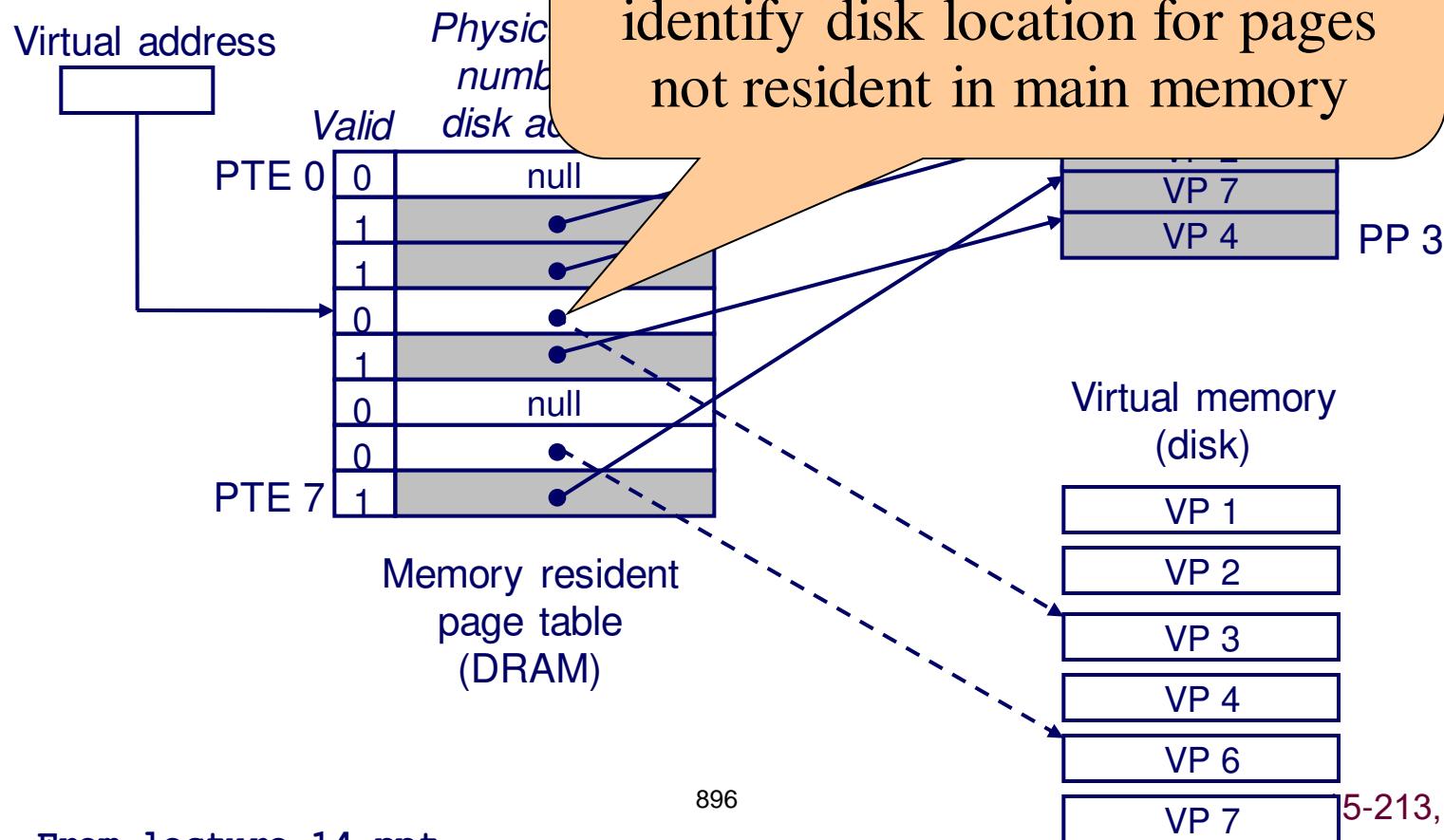
OS's view of storage device

- **Common “logical block” size: 512 bytes**
- **Number of blocks: device capacity / block size**
- **Common OS-to-storage requests defined by few fields**
 - R/W, block #, # of blocks, memory source/dest

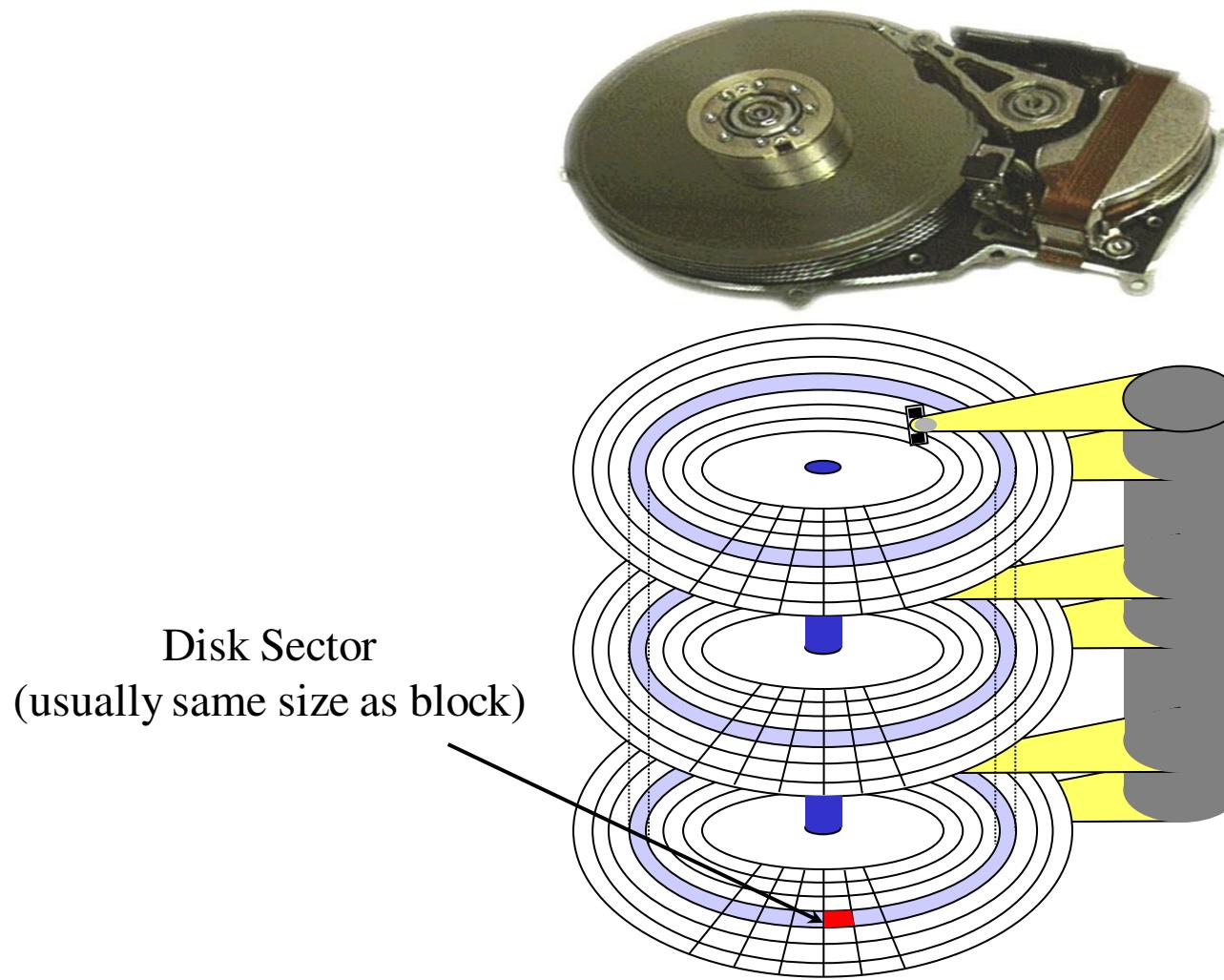
Page Faults

A *page fault* is caused by a reference to a VM word that is not in physical (main) memory

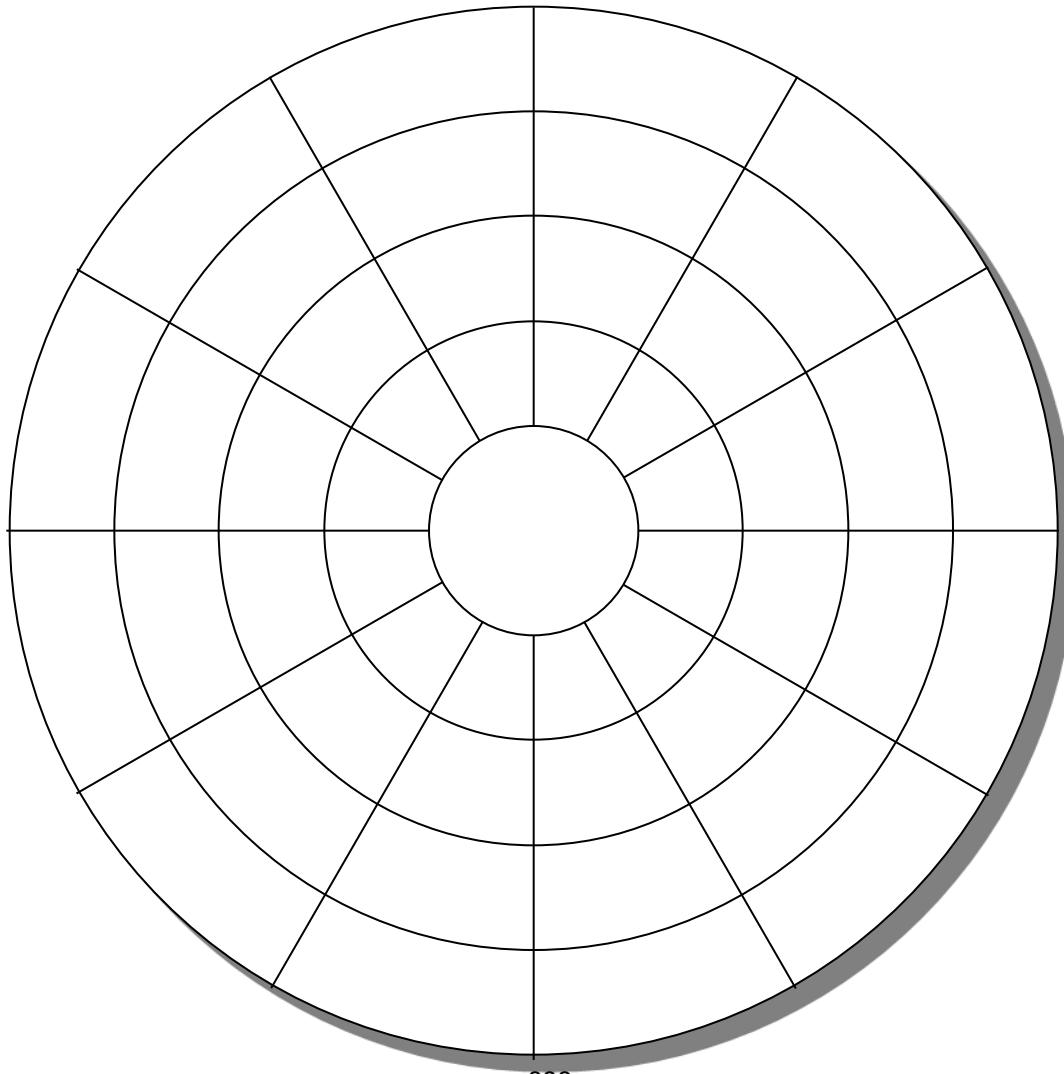
- Example: An instruction reference that triggers a page fault e



In device, “blocks” mapped to physical store



Physical sectors of a single-surface disk



Disk Capacity

Capacity: maximum number of bits that can be stored

- Vendors express capacity in units of gigabytes (GB), where
 $1 \text{ GB} = 10^9 \text{ Bytes}$ (Lawsuit pending! Claims deceptive advertising)

Capacity is determined by these technology factors:

- **Recording density (bits/in):** number of bits that can be squeezed into a 1 inch linear segment of a track
- **Track density (tracks/in):** number of tracks that can be squeezed into a 1 inch radial segment
- **Areal density (bits/in²):** product of recording and track density

Computing Disk Capacity

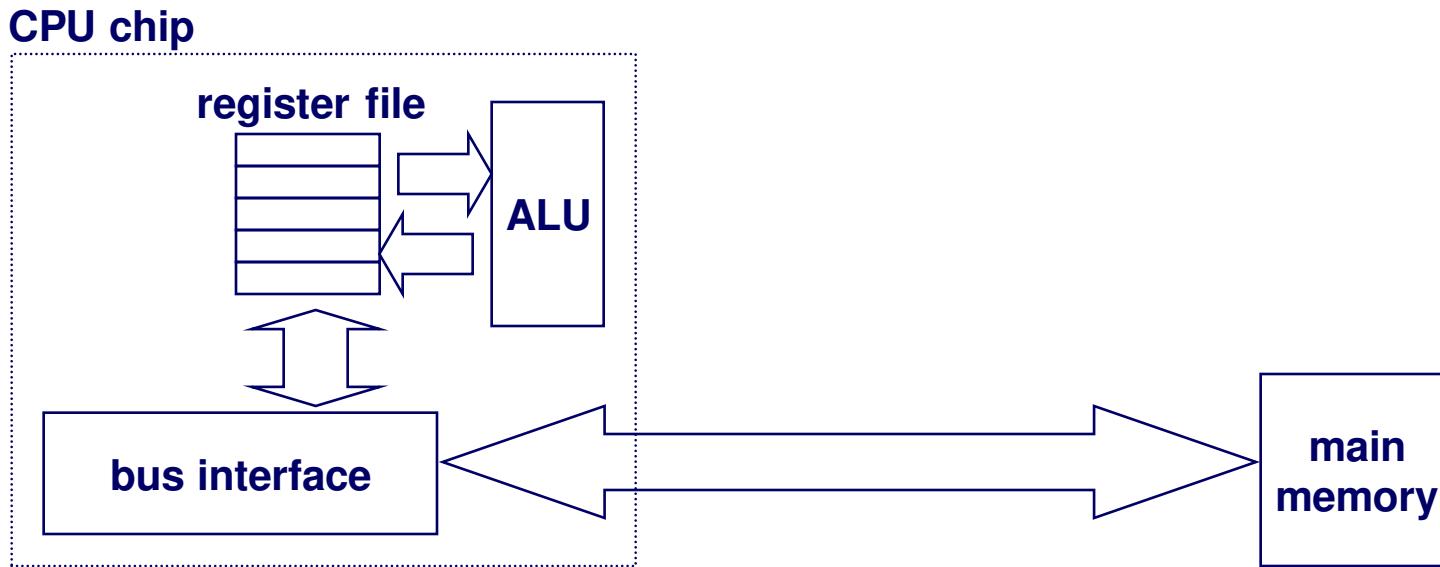
**Capacity = (# bytes/sector) x (avg. # sectors/track) x
 (# tracks/surface) x (# surfaces/platter) x
 (# platters/disk)**

Example:

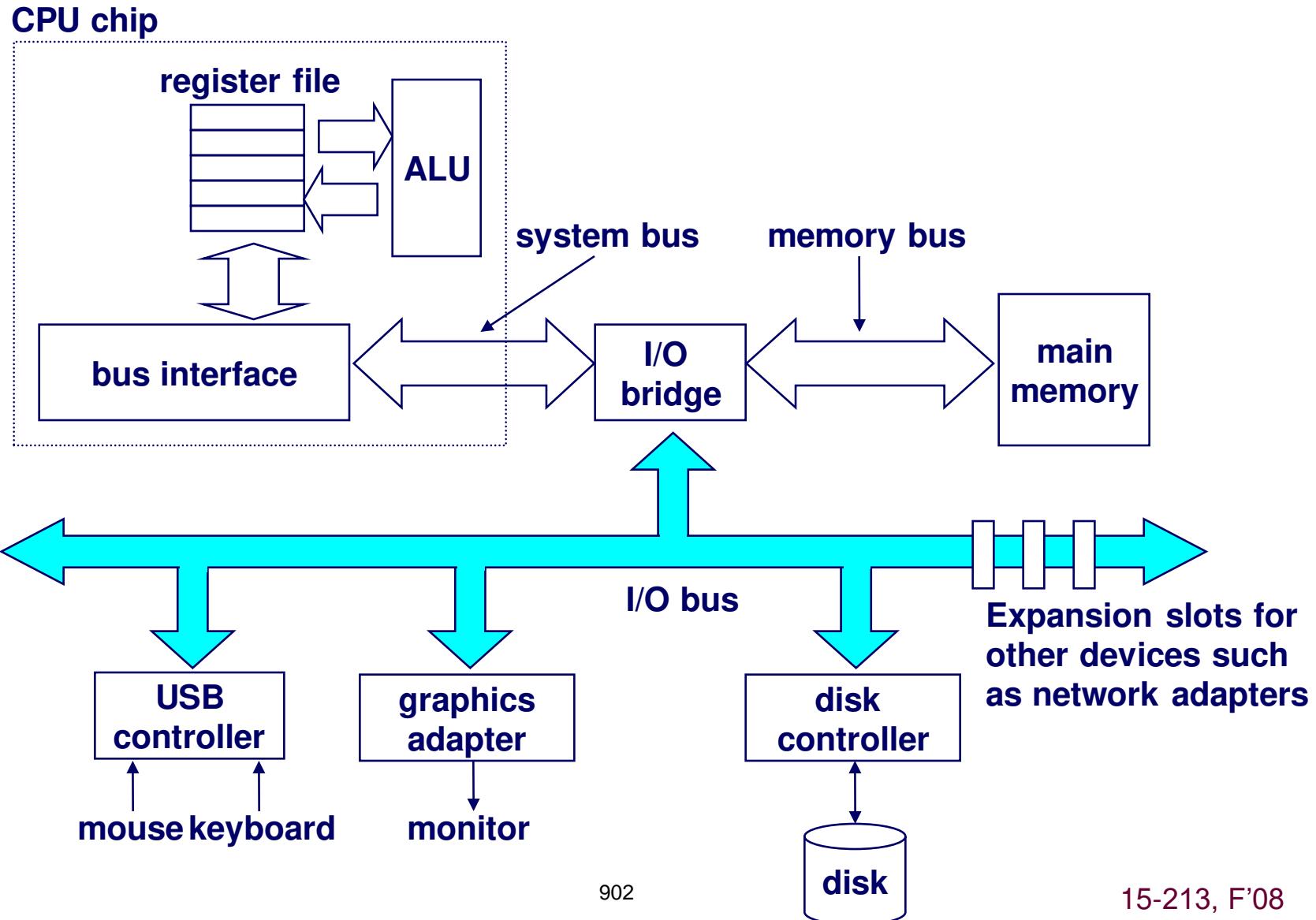
- 512 bytes/sector
- 1000 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

$$\begin{aligned}\text{Capacity} &= 512 \times 1000 \times 80000 \times 2 \times 5 \\ &= 409,600,000,000 \\ &= 409.6 \text{ GB}\end{aligned}$$

Looking back at the hardware

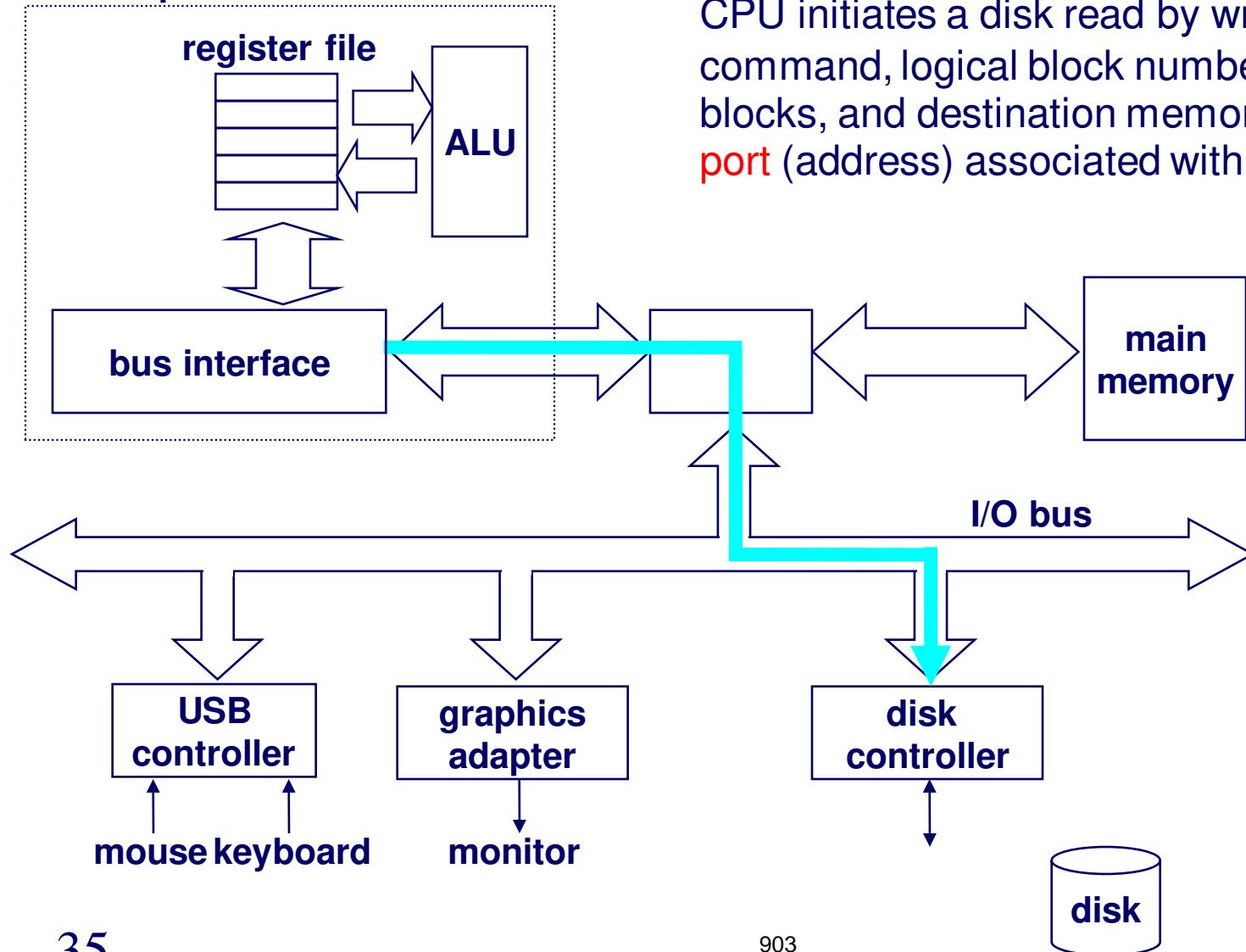


Connecting I/O devices: the I/O Bus



Reading from disk (1)

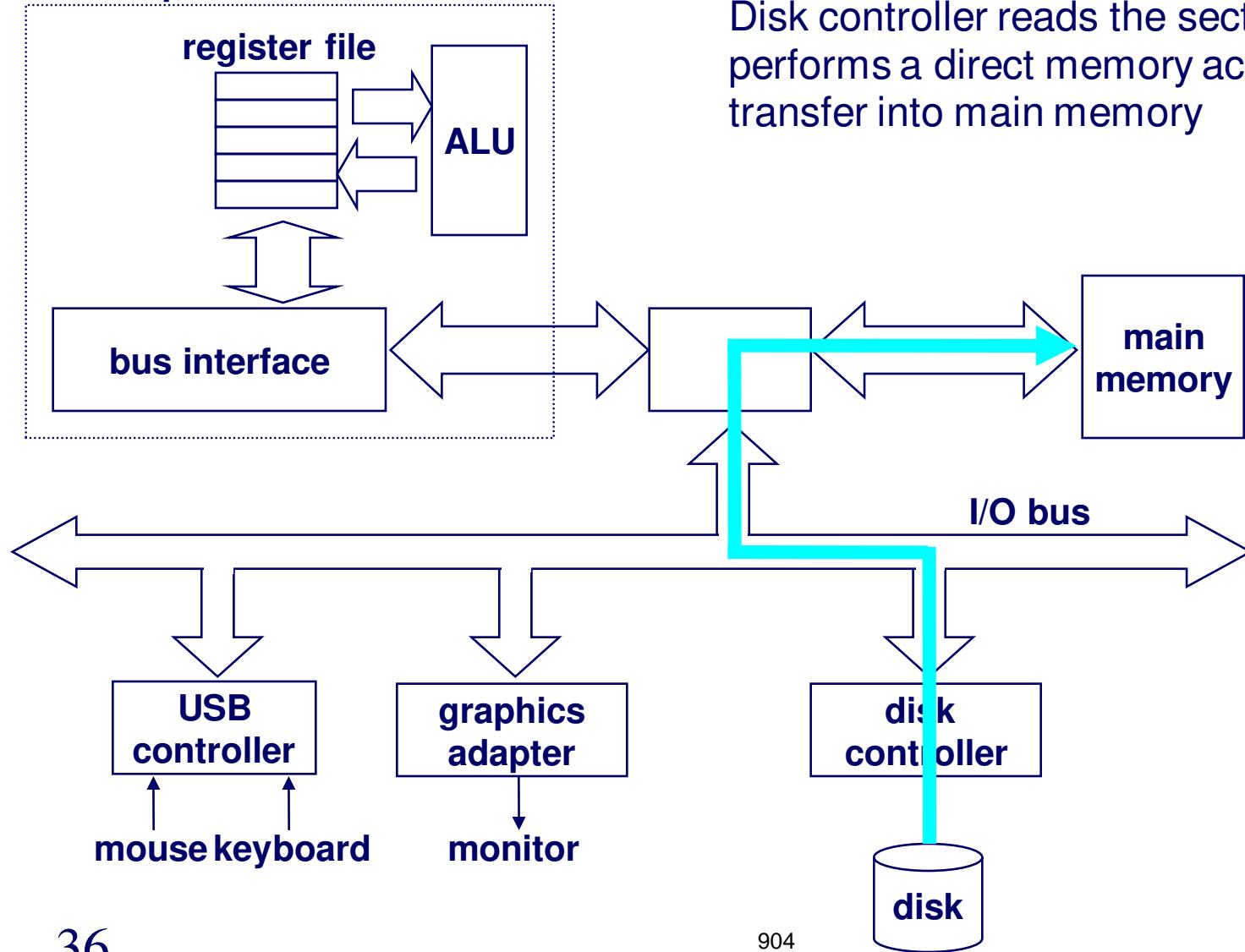
CPU chip



CPU initiates a disk read by writing a READ command, logical block number, number of blocks, and destination memory address to a **port** (address) associated with disk controller

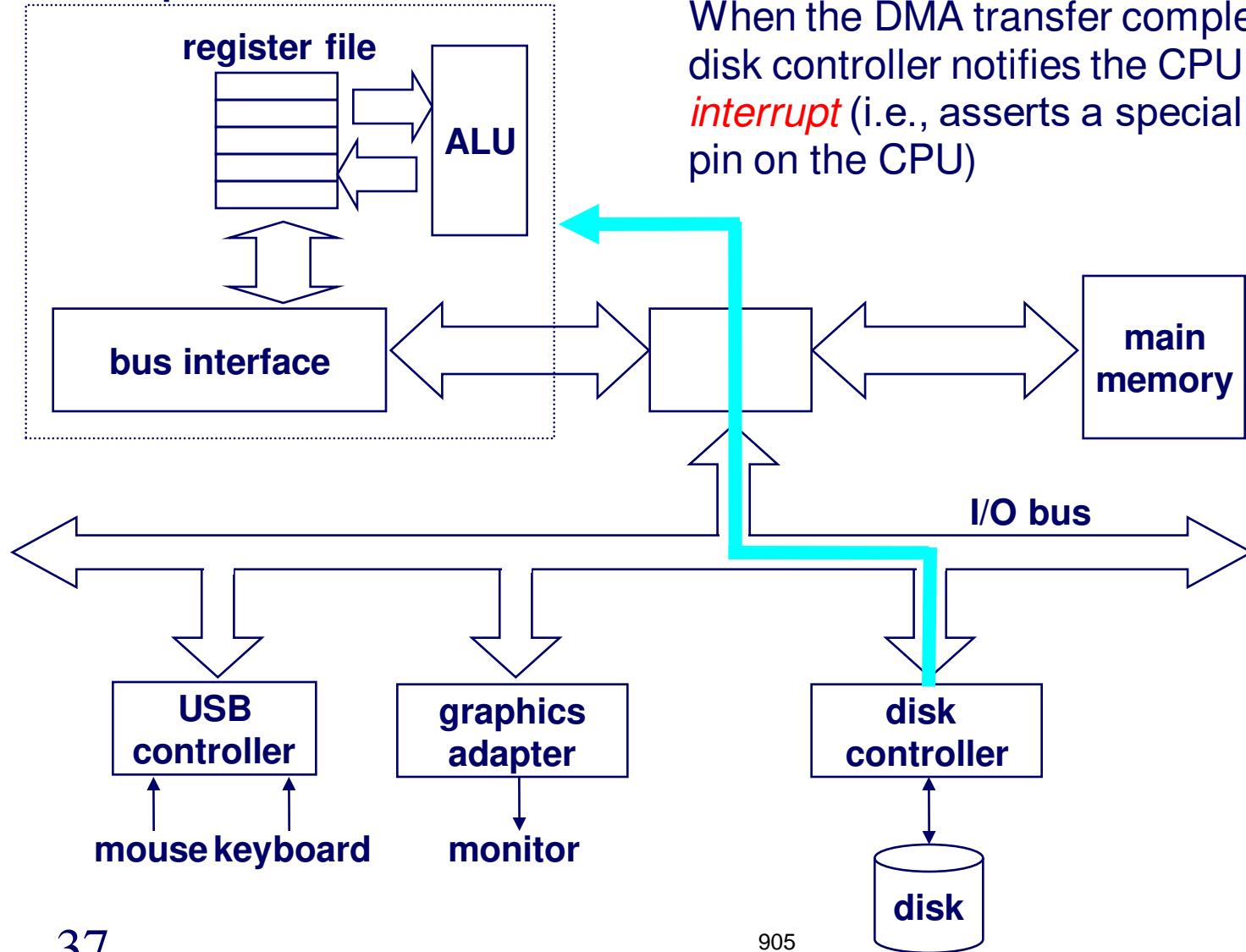
Reading from disk (2)

CPU chip



Reading from disk (3)

CPU chip



OPTICAL MEMORY:

A computer memory that uses optical techniques which generally involve an addressable laser beam, a storage medium which responds to the beam for writing and sometimes for erasing, and a detector which reacts to the altered character of the medium when it uses the beam to read out stored data..

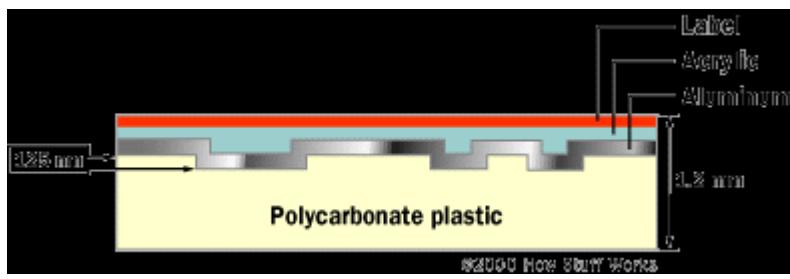
In 1983, one of the most successful consumer products of all time was introduced: the compact disk (CD) digital audio system. The CD is a nonerasable disk that can store more than 60 minutes of audio information on one side. The huge commercial success of the CD enabled the development of low-cost optical-disk storage technology that has revolutionized computer data storage. A variety of optical-disk systems have been introduced (Table 6.4). We briefly review each of these.

Table 6.4 Optical Disk Products

CD	Compact Disk. A nonerasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.
CD-ROM	Compact Disk Read-Only Memory. A nonerasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.
CD-R	CD Recordable. Similar to a CD-ROM. The user can write to the disk only once.
CD-RW	CD Rewritable. Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.
DVD	Digital Video Disk. A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8- and 12-cm diameters are used, with a double-sided capacity of up to 17 Gbytes. The basic DVD is read-only (DVD-ROM).
DVD-R	DVD Recordable. Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.
DVD-RW	DVD Rewritable. Similar to a DVD-ROM. The user can write to the disk multiple times. Only one-sided disks can be used.

Structure of optical device:

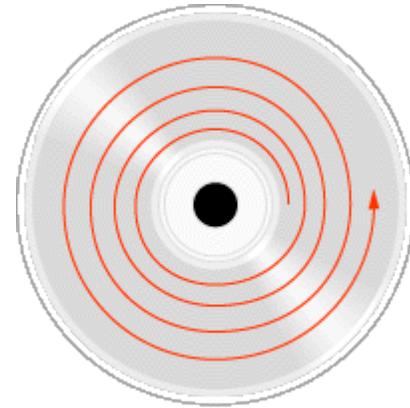
A CD is a fairly simple piece of plastic, about four one-hundredths (4/100) of an inch (1.2 mm) thick. Most of a CD consists of an **injection-molded piece of clear polycarbonate plastic**. During manufacturing, this plastic is impressed with microscopic bumps arranged as a single, continuous, extremely long spiral track of data. We'll return to the bumps in a moment. Once the clear piece of polycarbonate is formed, a thin, reflective aluminum layer is sputtered onto the disc, covering the bumps. Then a thin acrylic layer is sprayed over the aluminum to protect it. The label is then printed onto the acrylic. A cross section of a complete CD (not to scale) looks like this:



Understanding the CD: The Spiral

A CD has a single spiral track of data, circling from the inside of the disc to the outside. The fact that the spiral track starts at the center means that the CD can be smaller than 4.8 inches (12 cm) if desired, and in fact there are now plastic baseball cards and business cards that you can put in a CD player. CD business cards hold about 2 MB of data before the size and shape of the card cuts off the spiral.

What the picture on the right does not even begin to impress upon you is how incredibly small the data track is -- it is approximately 0.5 microns wide, with 1.6 microns separating one track from the next. (A micron is a millionth of a meter.) And the bumps are even more minuscule...

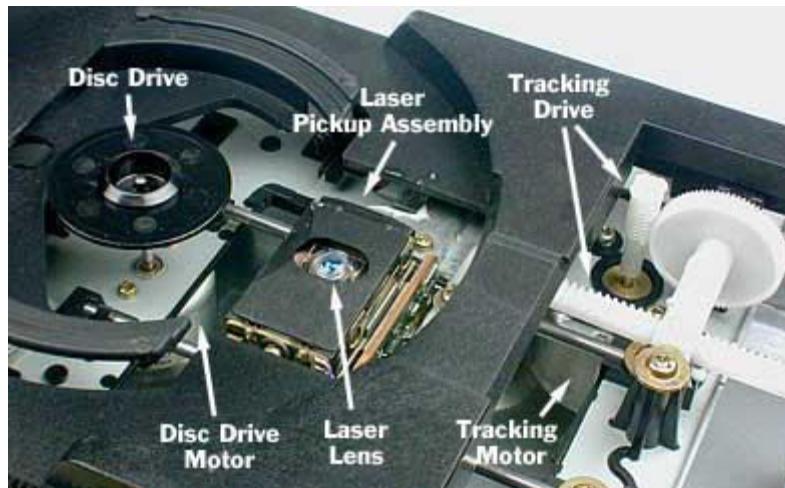


TO READ THE DATA WE NEED OPTICAL PLAYER:

read something this small you need an incredibly precise disc-reading mechanism. Let's take a look at that.

OPTICAL Player Components: The CD player has the job of finding and reading the data stored as bumps on the CD. Considering how small the bumps are, the CD player is an exceptionally precise piece of equipment. The drive consists of three fundamental components:

- A **drive motor** spins the disc. This drive motor is precisely controlled to rotate between 200 and 500 rpm depending on which track is being read.
- A **laser** and a **lens system** focus in on and read the bumps.
- A **tracking mechanism** moves the laser assembly so that the laser's beam can follow the spiral track. The tracking system has to be able to move the laser at micron resolutions.



Now we will see how reading and writing is done in optical devices:

Writing:

Writing to the cell was accomplished by an external cathode ray tube (CRT) arranged in front of the photoemissive side of the grid. Cells were selected by using the deflection coils in the CRT to pull the beam into position in front of the cell, lighting up the front of the tube in that location. This initial pulse of light, focussed through a lens, would set the cell to the "on" state. Due to the way the photoemissive layer worked, focusing light on it again when it was already "lit up" would overload the material, stopping electrons from flowing out the other side into the interior of the cell. When the external light was then removed, the cell was dark, turning it "off".

Reading:

Reading the cells was accomplished by a grid of photocells arranged behind the phosphorescent layer, which emitted photons non-directionally. This allowed the cells to be read from the back of the device, as long as the phosphorescent layer was thin enough. To form a complete memory the system was arranged to be regenerative, with the output of the photocells being amplified and sent back into the CRT to refresh the cells periodically.

ADVANTAGES:

- The optical disk together with the information stored on it can be mass replicated inexpensively—unlike a magnetic disk. The database on a magnetic disk has to be reproduced by copying one disk at a time using two disk drives.
- The optical disk is removable, allowing the disk itself to be used for archival storage. Most magnetic disks are nonremovable. The information on non-removable magnetic disks must first be copied to tape before the disk drive/disk can be used to store new information.

The disadvantages of CD-ROM are as follows:

- It is read-only and cannot be updated.
- It has an access time much longer than that of a magnetic disk drive, as much as half a second.

Parallel Processing (Multiprocessing)

Its All About *Increasing Performance*

- Processor performance can be measured by the rate at which it executes instructions

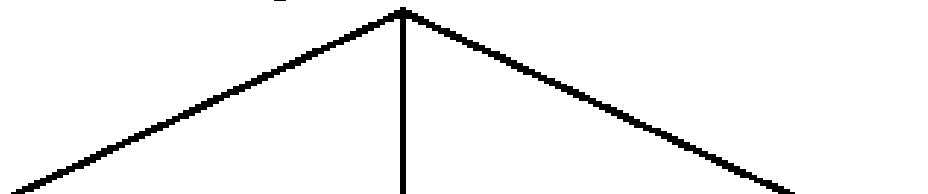
MIPS rate = $f * IPC$ (Millions of Instructions per Second)

- f is the processor clock frequency, in MHz
- IPC the is average Instructions Per Cycle

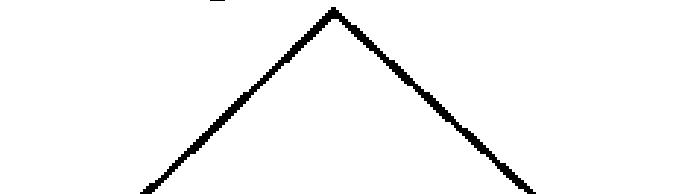
- Increase performance by
 - increasing clock frequency and
 - increasing instructions that complete during cycle
 - May be reaching limit ☺
 - + Complexity
 - + Power consumption

Computer Organizations

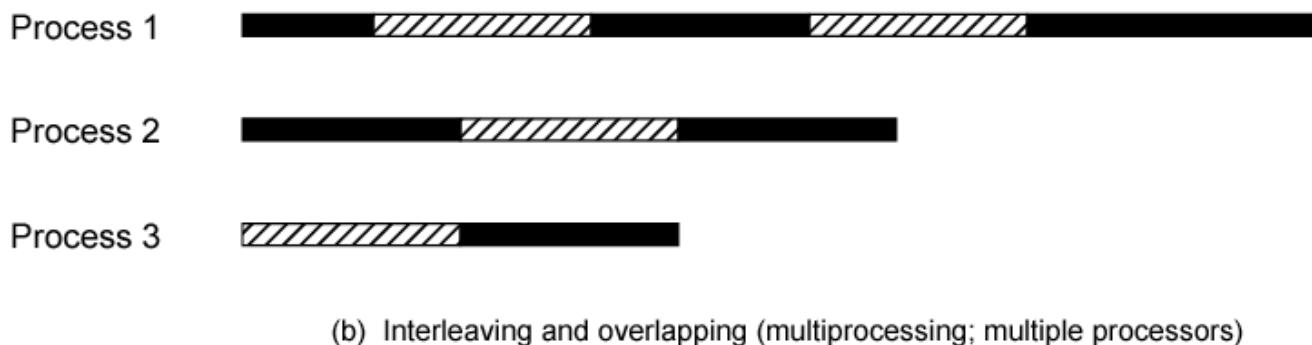
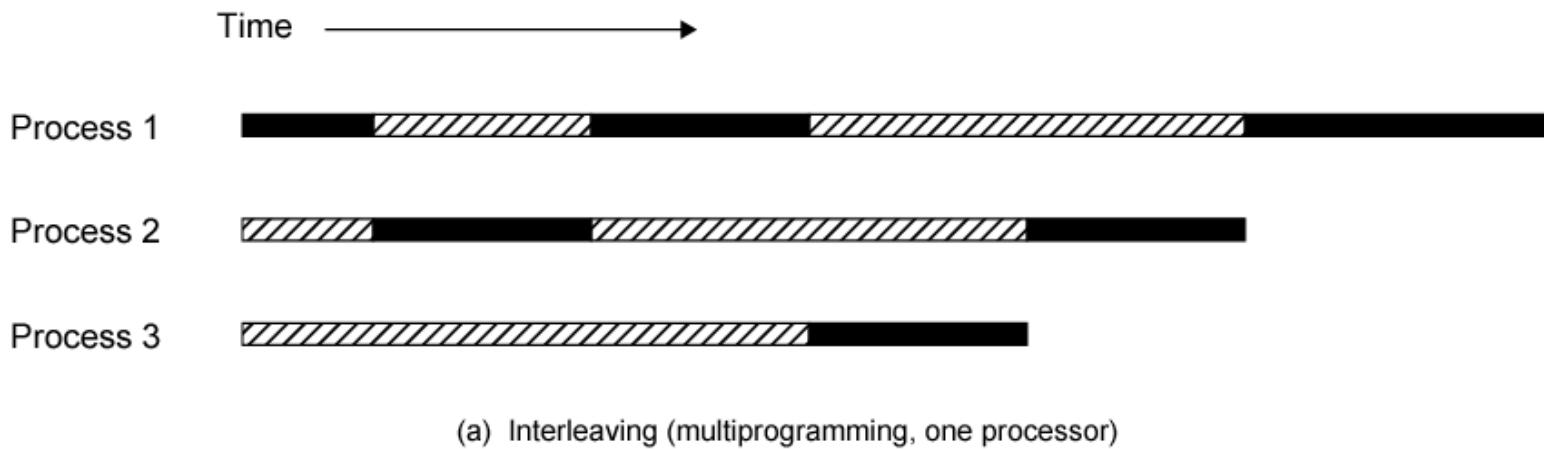
Single Control Unit



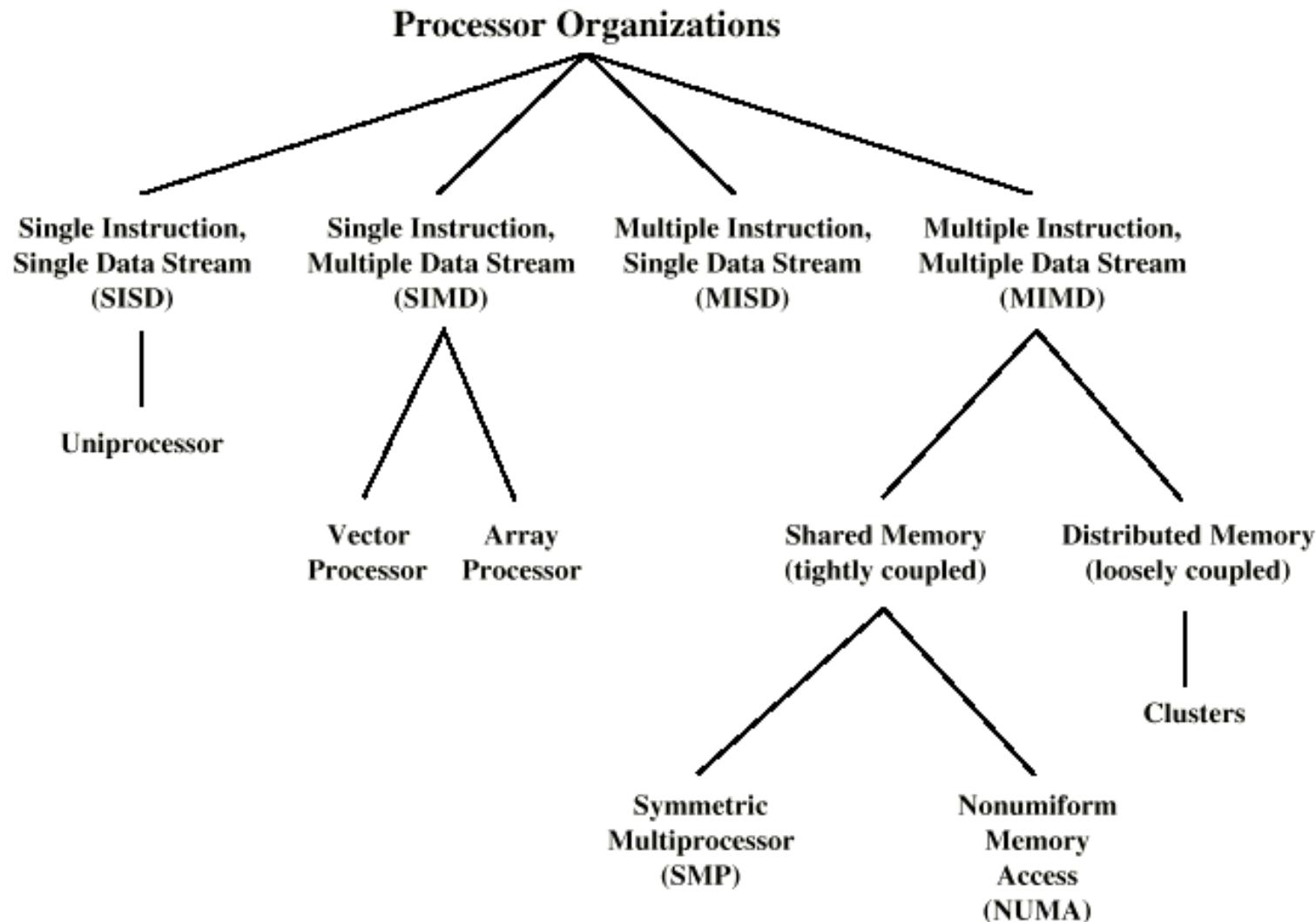
Multiple Control Units



Multiprogramming and Multiprocessing



Taxonomy of Parallel Processor Architectures

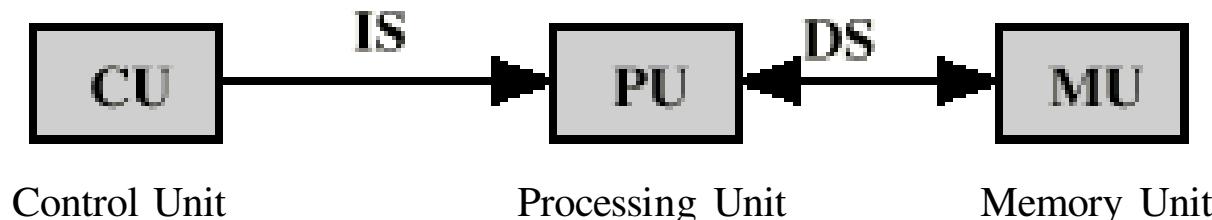


Multiple Processor Organization

- SISD - Single instruction, single data stream
- SIMD - Single instruction, multiple data stream
- MISD - Multiple instruction, single data stream
- MIMD - Multiple instruction, multiple data stream

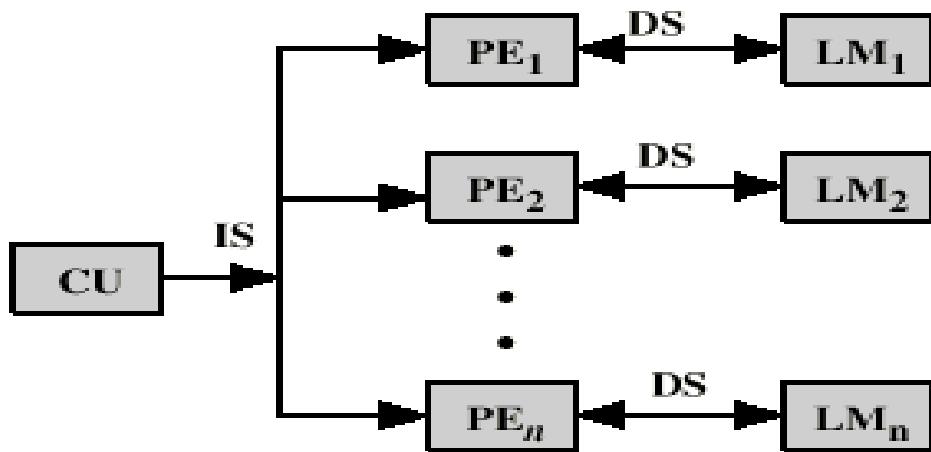
SISD - Single Instruction, Single Data Stream

- Single processor
- Single instruction stream
- Data stored in single memory
- Uni-processor



SIMD - Single Instruction, Multiple Data Stream

- Single machine instruction
- Number of processing elements
- Each processing element has associated data memory
- Each instruction simultaneously executed on different set of data by different processors



Typical Application - Vector and Array processors

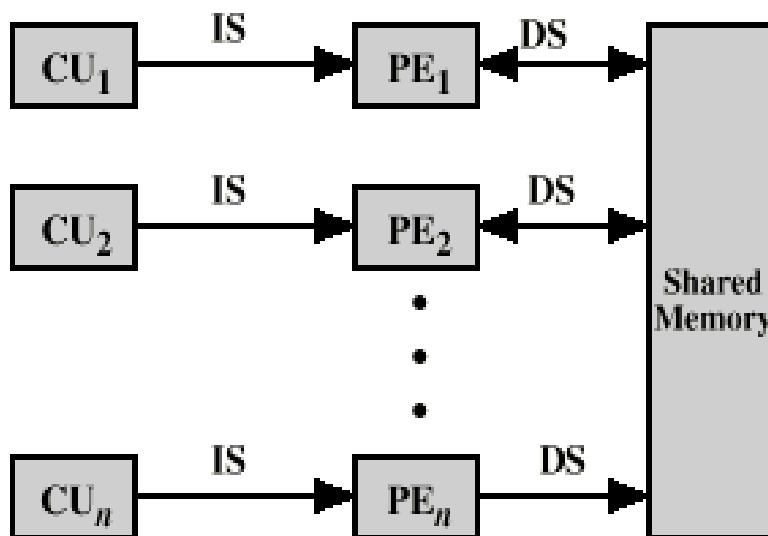
MISD Multiple Instruction, Single Data Stream

- One sequence of data
- A set of processors
- Each processor executes different instruction sequence

Not much practical application

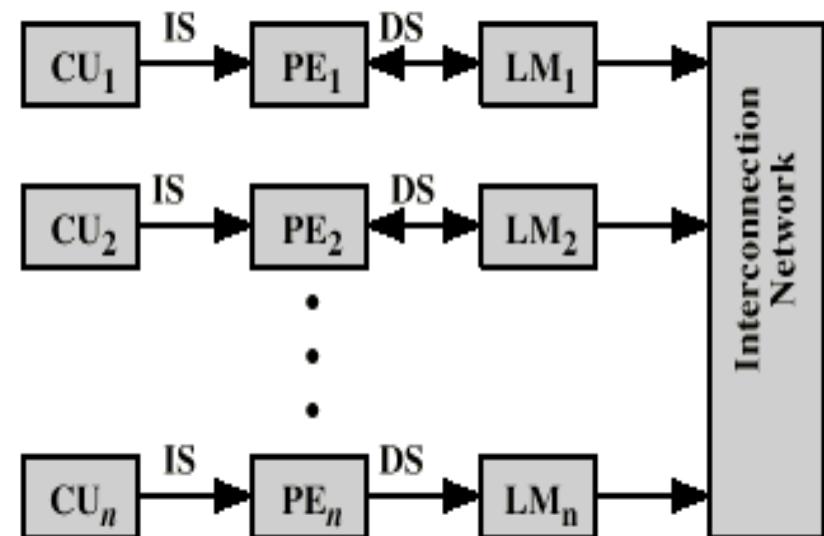
MIMD - Multiple Instruction, Multiple Data Stream

- Set of processors
- Simultaneously execute different instruction sequences
- Different sets of data
 - SMPs (Symmetric Multiprocessors)
 - NUMA systems (Non-uniform Memory Access)
 - Clusters (Groups of “partnering” computers)



Shared memory (SMP or NUMA)

919



Distributed memory (Clusters)

MIMD - Overview

- General purpose processors
- Each can process all instructions necessary
- Further classified by method of processor communication & memory access

MIMD - Tightly Coupled

- Processors share memory
- Communicate via that shared memory

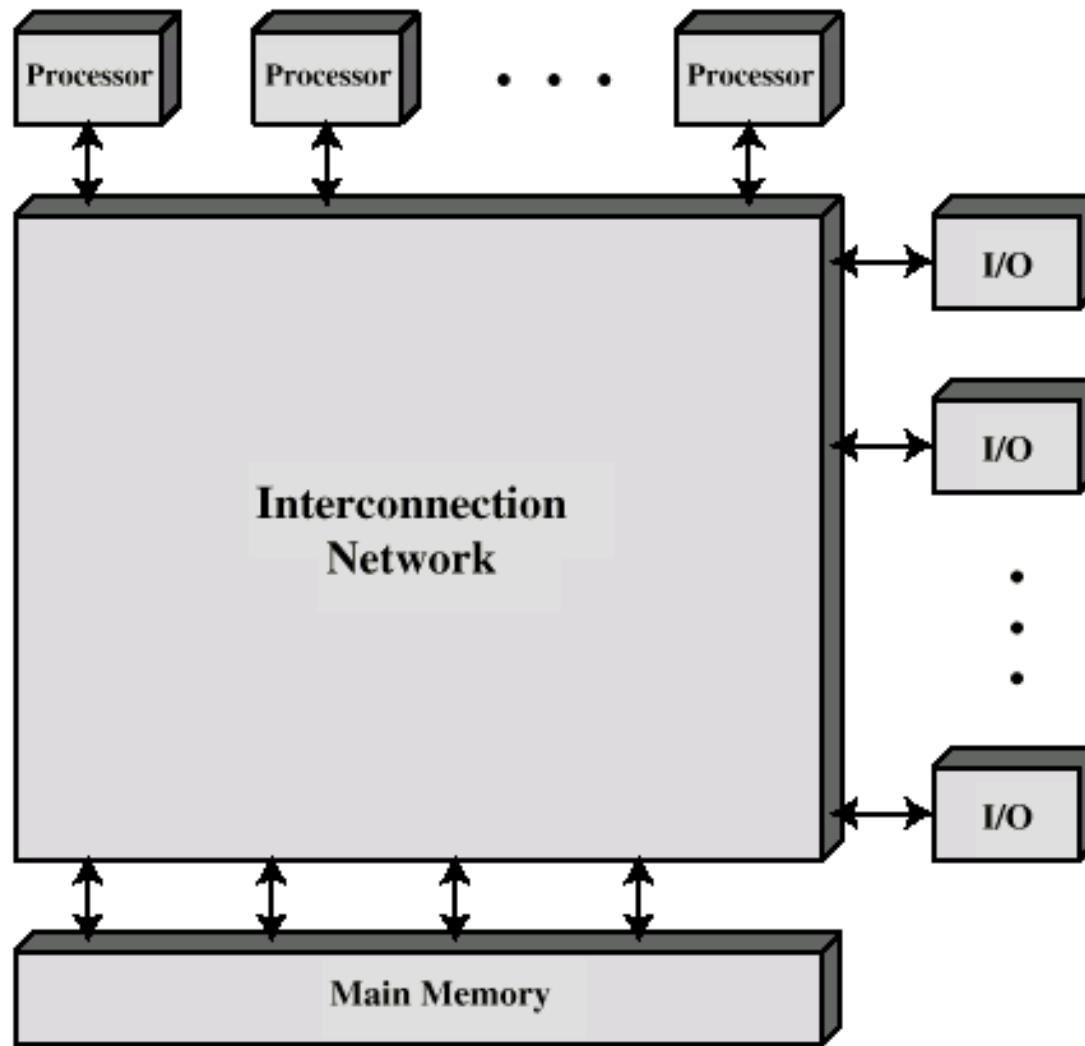
Symmetric Multiprocessor (SMP)

- Share single memory or pool
- Shared bus to access memory
- Memory access time to given area of memory is approximately the same for each processor

Nonuniform memory access (NUMA)

- Access times to different regions of memory may differ

Block Diagram of Tightly Coupled Multiprocessor



MIMD - Loosely Coupled

Clusters

- Collection of independent uniprocessors
- Interconnected to form a cluster
- Communication via fixed path or network connections

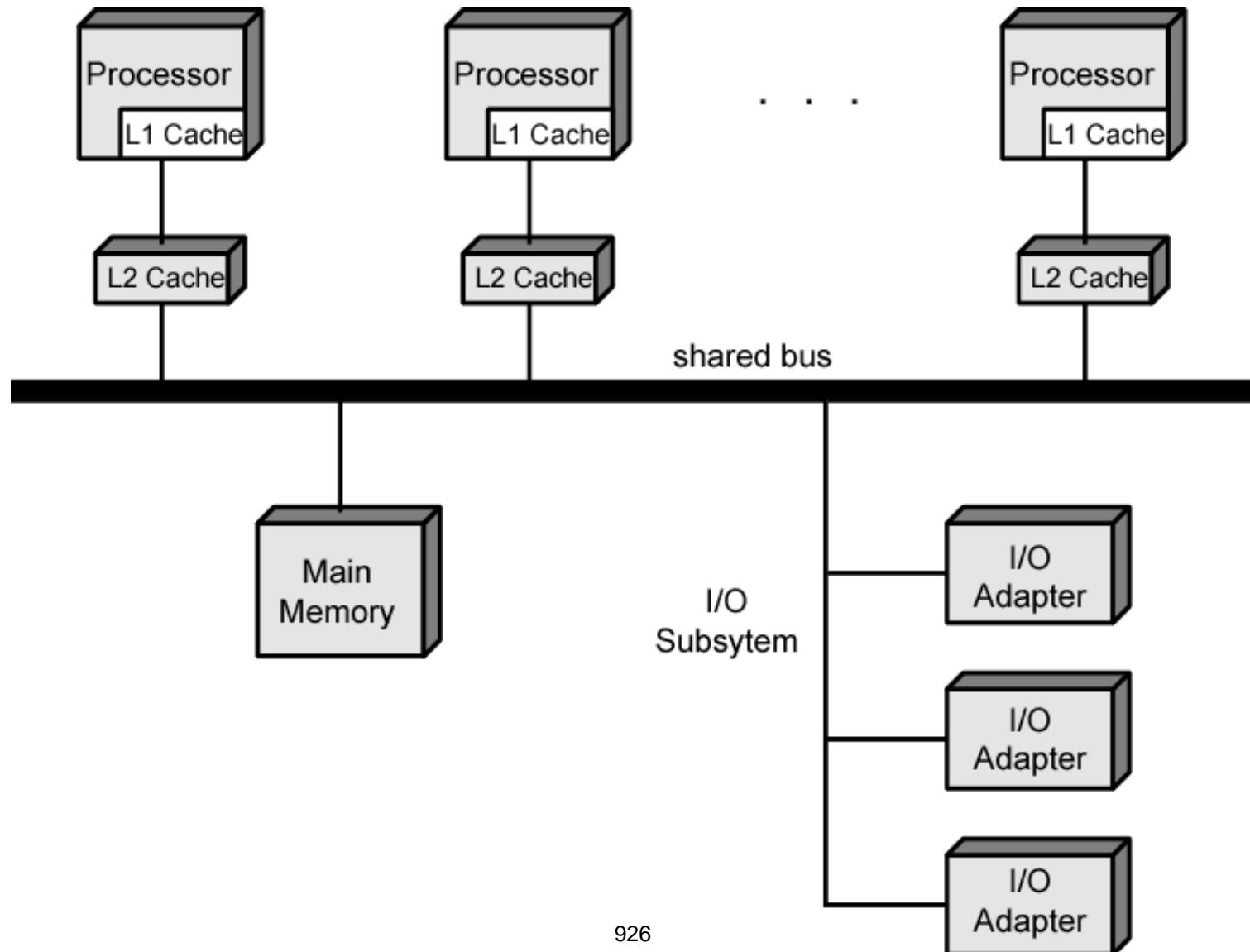
Symmetric Multiprocessors (SMP)

- A stand alone “computer” with the following characteristics:
 - Two or more similar processors of comparable capacity
 - All processors can perform the same functions (hence symmetric)
 - Processors share same memory and I/O access
 - Memory access time is approximately the same for each processor (time shared bus or multi-port memory)
 - Processors are connected by a bus or other internal connection
 - System controlled by integrated operating system
 - Providing interaction between processors
 - Providing interaction at job, task, file and data element levels

SMP Advantages

- Performance
 - If some work can be done in parallel
- Availability
 - Since all processors can perform the same functions, failure of a single processor does not halt the system
- Incremental growth
 - User can enhance performance by adding additional processors
- Scaling
 - Vendors can offer range of products based on number of processors

Symmetric Multiprocessor Organization



Time Shared Bus (vs Multiport memory)

- Simplest form
- Structure and interface similar to single processor system
- Following features provided
 - **Addressing** - distinguish modules on bus
 - **Arbitration** - any module can be temporary master
 - **Time sharing** - if one module has the bus, others must wait and may have to suspend

Time Share Bus - Advantages

Advantages:

- Simplicity
- Flexibility
- Reliability

Disadvantages

- Performance limited by bus cycle time
- Each processor must have local cache
 - Reduce number of bus accesses
- Leads to problems with cache coherence

Operating System Issues

- Simultaneous concurrent processes
- Scheduling
- Synchronization
- Memory management
- Reliability and fault tolerance
- Cache Coherence

Cache Coherence

- Problem - multiple copies of same data in different caches
- Can result in an inconsistent view of memory
 - Write back policy can lead to inconsistency
 - Write through can also give problems unless caches monitor memory traffic

→ MESI Protocol (Modify - Exclusive - Shared - Invalid)

Software Solution to Cache Coherence

Compiler and operating system deal with problem

- Overhead transferred to compile time
- Design complexity transferred from hardware to software
 - Analyze code to determine safe periods for caching shared variables
 - However, software tends to (must) make conservative decisions
 - → Inefficient cache utilization

Hardware Solution to Cache Coherence

Cache coherence hardware protocols

- Dynamic recognition of potential problems
- Run time solution
 - More efficient use of cache
- Transparent to programmer / Compiler

Implemented with:

- Directory protocols
- Snoopy protocols

Directory & Snoopy Protocols

Directory Protocols

Effective in large scale systems with complex interconnection schemes

- Collect and maintain information about copies of data in cache
 - Directory stored in main memory
- Requests are checked against directory
 - Appropriate transfers are performed

Creates central bottleneck

Snoopy Protocols

Suited to bus based multiprocessor

- Distribute cache coherence responsibility among cache controllers
- Cache recognizes that a line is shared
- Updates announced to other caches

Increases bus traffic

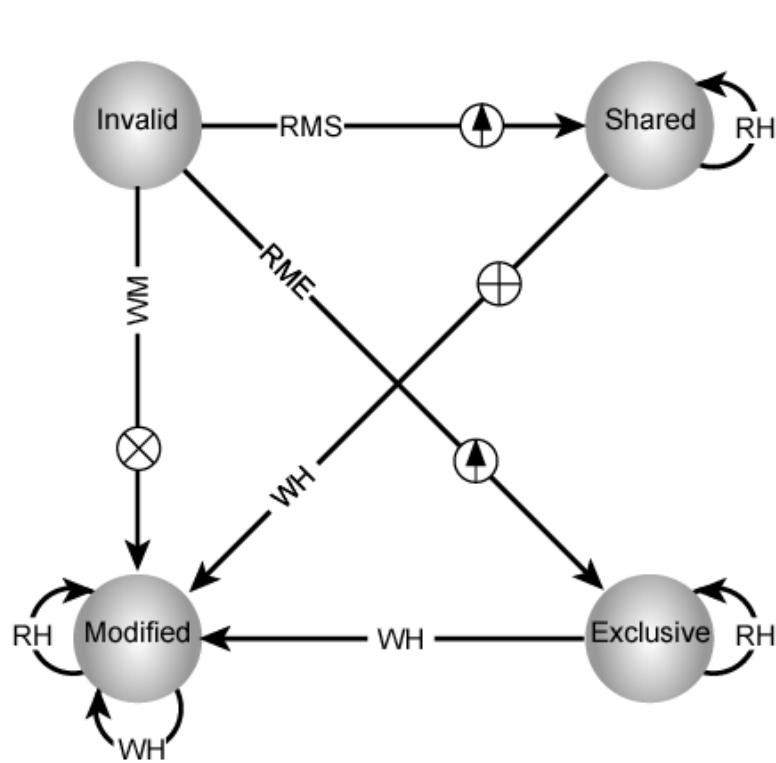
Snoopy Protocols

- Write Update Protocol (Write Broadcast)
 - Multiple readers and writers
 - Updated word is distributed to all other processors
 - Multiple readers, one writer
- Write Invalidate protocol (MESI)
 - When a write is required, all other caches of the line are invalidated
 - Writing processor then has exclusive (cheap) access until line is required by another processor

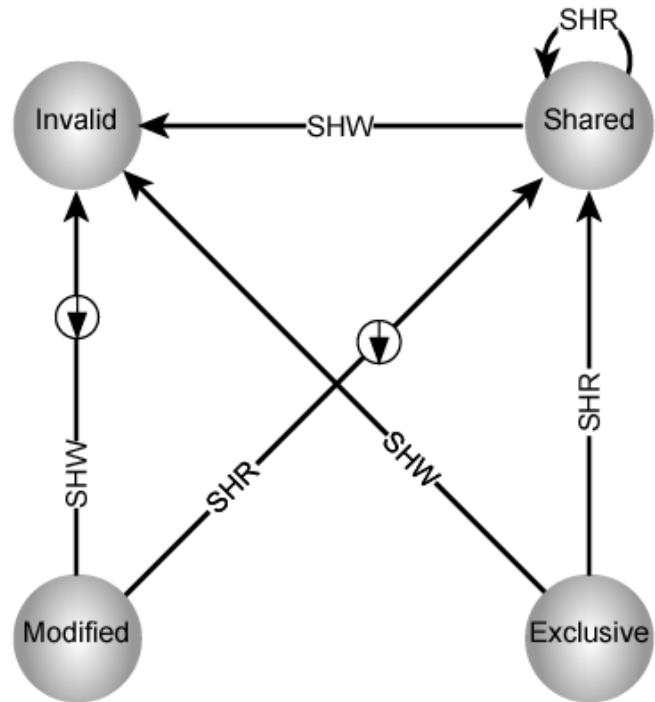
MESI Protocol - State of every line is marked as Modified, Exclusive, Shared or Invalid

- two bits are included with each cache tag

MESI State Transition Diagram



(a) Line in cache at initiating processor



(b) Line in snooping cache

RH	Read hit
RMS	Read miss, shared
RME	Read miss, exclusive
WH	Write hit
WM	Write miss
SHR	Snoop hit on read
SHW	Snoop hit on write or read-with-intent-to-modify

- | | |
|---|----------------------------|
| ▼ | Dirty line copyback |
| ⊕ | Invalidate transaction |
| ⊗ | Read-with-intent-to-modify |
| ▲ | Cache line fill |

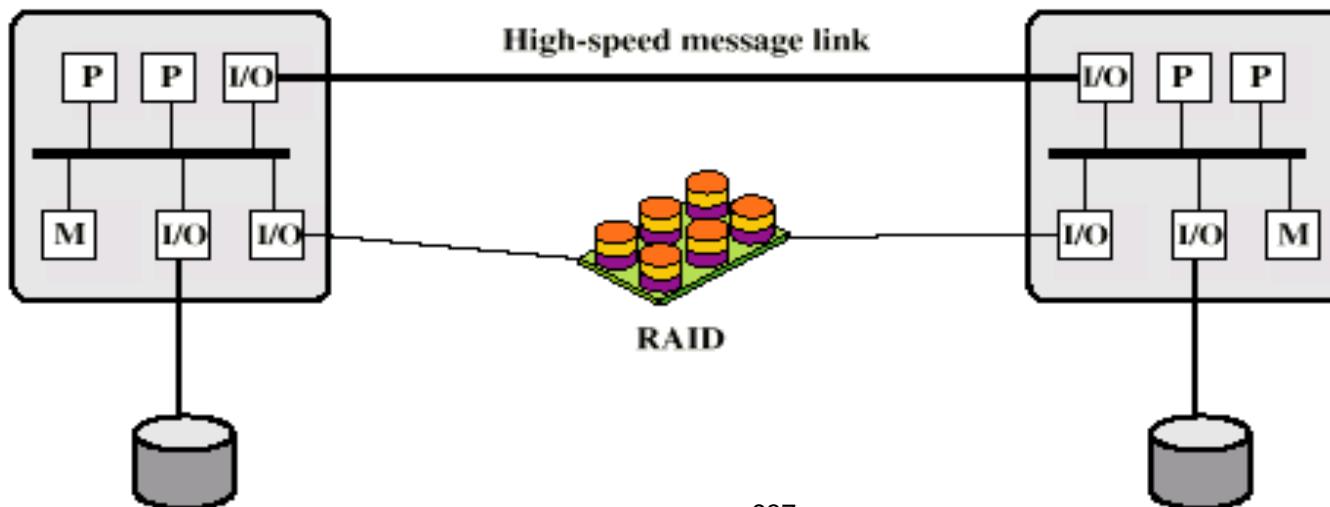
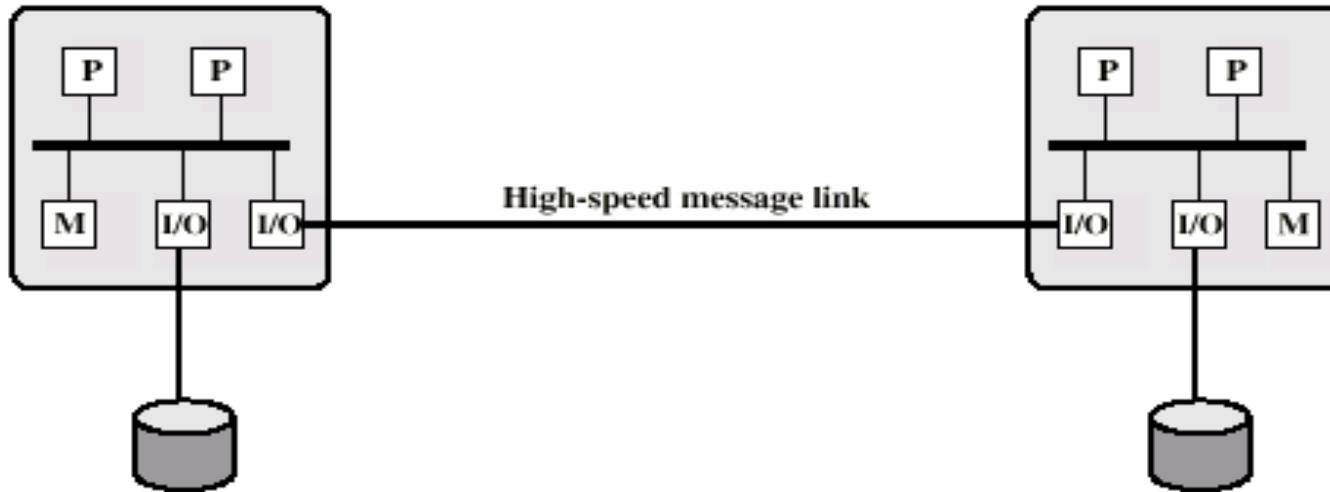
Clusters (Alternative to SMP)

- A group of interconnected whole computers (or SMP's)
- Working together as unified resource
- Illusion of being one machine
- Each computer called a node

Benefits:

- Absolute scalability
- Incremental scalability
- High availability
- Superior performance/price

Cluster Configurations



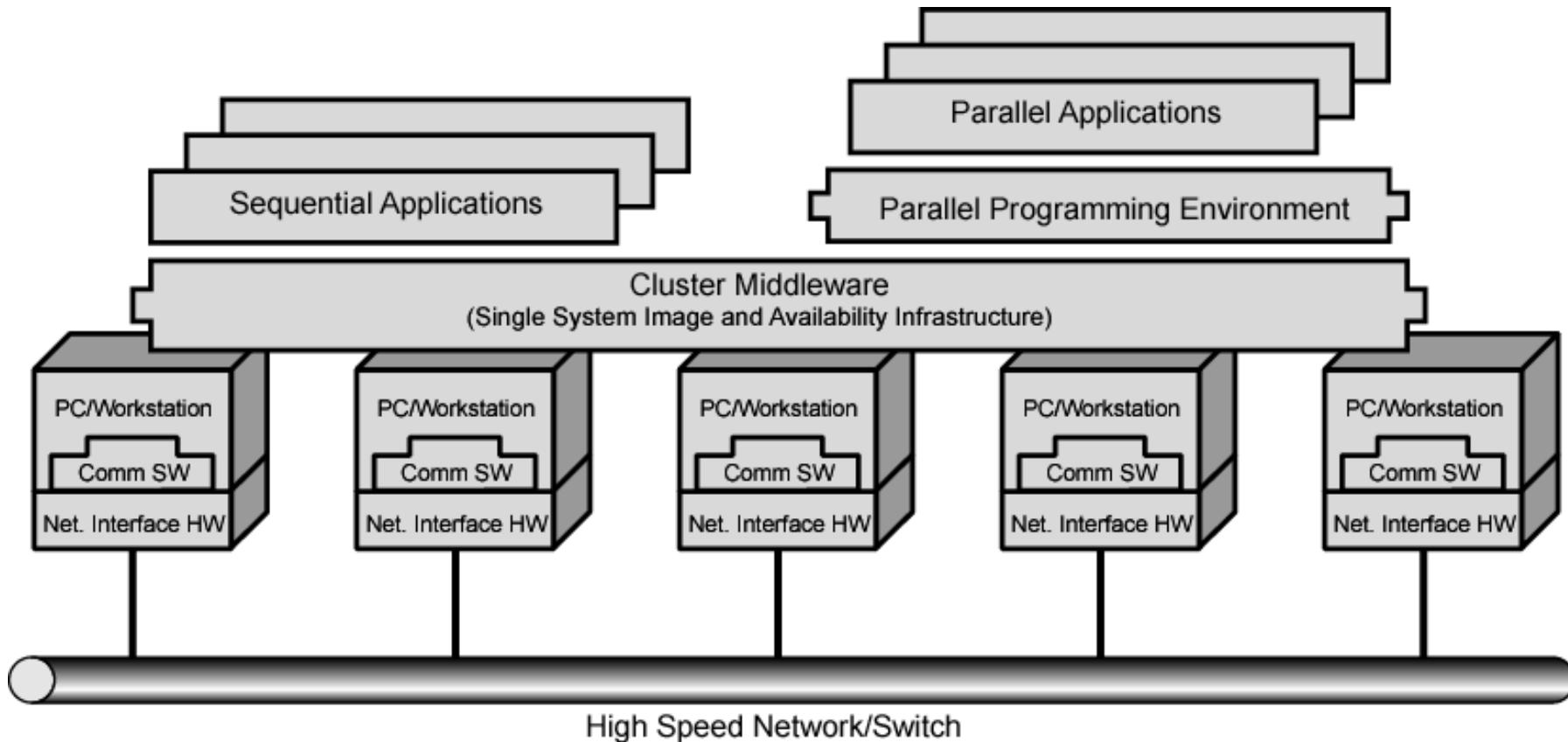
Operating Systems Design Issues

- Failure Management
 - High available clusters
 - Fault tolerant clusters
 - Failover
 - Switching applications & data from failed system to alternative within cluster
 - Fallback
 - Restoration of applications and data to original system after problem is fixed
- Load balancing
 - Incremental scalability
 - Automatically include new computers in scheduling
 - Middleware needs to recognise that processes may switch between machines

Parallelizing Computation

- Single application is executing, in parallel, on a number of machines in a cluster
 - Complier Approach
 - Determine at compile time which parts can be executed in parallel
 - Split off parts to different computers
 - Application Approach
 - Application written from scratch to be parallel
 - Message passing used to move data between nodes
 - + Difficult to program, but best end result
 - Parametric Computing Approach
 - If a problem is repeated execution of algorithm on different sets of data, e.g. simulation using different scenarios
 - + Needs effective tools to organize and run

Cluster Computer Architecture



Cluster Middleware

Everything LOOKS like a single machine to the user

- Single point of entry
- Single file hierarchy
- Single control point (one management computer?)
- Single virtual networking (distributed control)
- Single (virtual) memory space
- Single job management system
- Single user interface
- Single I/O space
- Single process space
- Checkpointing symmetric
- Process migration

Cluster v. SMP

- Both provide multiprocessor support to high demand applications.
- Both available commercially
 - SMP for longer time
- SMP:
 - Easier to manage and control
 - Inherently closer to single processor system
 - Less physical space
 - Lower power consumption
- Clustering:
 - Superior incremental & absolute scalability
 - Superior availability
 - Redundancy

Nonuniform Memory Access (NUMA)

Alternative to SMP & Clustering

- Uniform memory access
 - All processors have access to all parts of memory
 - Using load & store
 - Access time to all regions of memory is the same
 - Access time to memory for different processors same
 - As used by SMP
- Nonuniform memory access
 - All processors have access to all parts of memory
 - Access time of processor differs depending on region of memory
 - Different processors access different regions of memory at different speeds

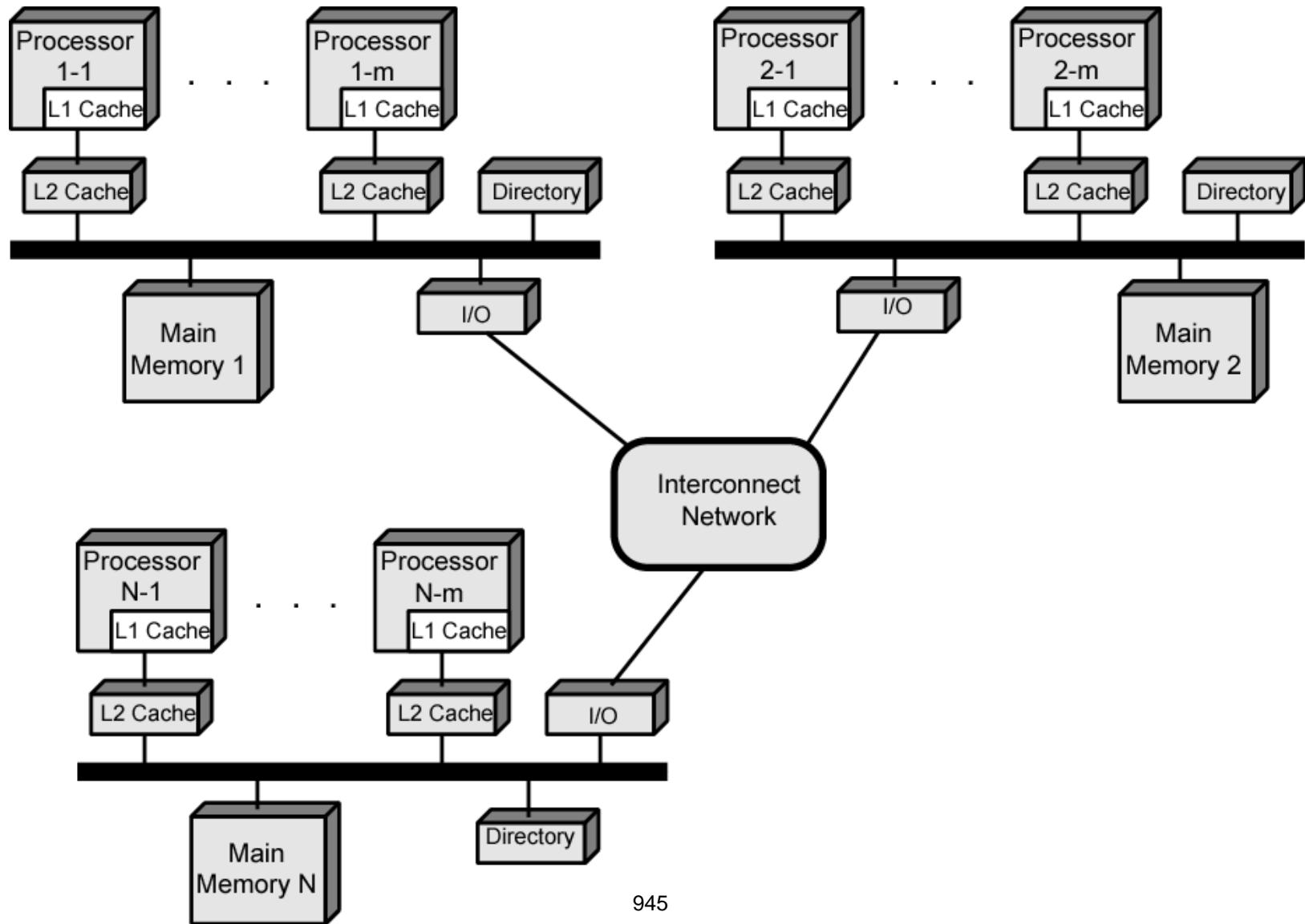
Cache coherent NUMA - CC-NUMA

- Cache coherence is maintained among the caches of the various processors

Motivation for CC-NUMA

- SMP has practical limit to number of processors
 - Bus traffic limits to between 16 and 64 processors
- In clusters each node has own memory
 - Apps do not see large global memory
 - Coherence maintained by software not hardware
- NUMA retains SMP flavor while providing large scale multiprocessing
- Objective is to maintain transparent system wide memory while permitting multiprocessor nodes, each with own bus or internal interconnection system

CC-NUMA Organization



CC-NUMA Operation

- Each processor has own L1 and L2 cache
- Each node has own main memory
- Nodes connected by some networking facility
- Each processor sees single addressable memory space
- Memory request order:
 - L1 cache (local to processor)
 - L2 cache (local to processor)
 - Main memory (local to node)
 - Remote memory
 - Delivered to requesting (local to processor) cache
- Automatic and transparent

CC-NUMA Memory Access Sequence

- Each node maintains directory of location of portions of memory and cache status
- e.g. node 2 processor 3 (P2-3) requests location 798 which is in memory of node 1
 - P2-3 issues read request on snoopy bus of node 2
 - Directory on node 2 recognises location is on node 1
 - Node 2 directory requests node 1's directory
 - Node 1 directory requests contents of 798
 - Node 1 memory puts data on (node 1 local) bus
 - Node 1 directory gets data from (node 1 local) bus
 - Data transferred to node 2's directory
 - Node 2 directory puts data on (node 2 local) bus
 - Data picked up, put in P2-3's cache and delivered to processor

Cache Coherence

- Node 1 directory keeps note that node 2 has copy of data
- If data modified in cache, this is broadcast to other nodes
- Local directories monitor and purge local cache if necessary
- Local directory monitors changes to local data in remote caches and marks memory invalid until writeback
- Local directory forces writeback if memory location requested by another processor

NUMA Pros & Cons

- Effective performance potentially at higher levels of parallelism than SMP for many applications
- Less major software changes
- Performance can breakdown if there is too much access to remote memory
 - Can be avoided by:
 - Good spatial locality of software
 - L1 & L2 cache design reducing memory access
 - + Need good temporal locality of software
 - Virtual memory management moving pages to nodes that are using them most
- Not transparent
 - Page allocation, process allocation and load balancing changes needed
- Availability is a question₉₄₉

Vector Computation (SIMD)

- Math problems involving physical processes present different challenges for computation
 - Aerodynamics, seismology, meteorology
- High precision
- Repeated floating point calculations on large arrays of numbers
 - Supercomputers handle these types of problem
 - Hundreds of millions of flops
 - \$10-15 million
 - Optimised for calculation rather than multitasking and I/O
 - Limited market
 - + Research, government agencies, meteorology
- Array processor
 - Alternative to supercomputer
 - Configured as peripherals to mainframe & mini
 - Just run vector portion of problems

Vector Computation Approaches

- General purpose computers rely on iteration to do vector calculations
- Parallel processing
 - Independent processes functioning in parallel
 - Use FORK N to start individual process at location N
 - JOIN N causes N independent processes to join and merge following JOIN
 - O/S Co-ordinates JOINS
 - Execution is blocked until all N processes have reached JOIN
- Vector processing
 - Assume possible to operate on one-dimensional vector of data
 - All elements in a particular row can be calculated in parallel

Processor Designs

- Pipelined ALU
 - Within operations
 - Across operations
- Parallel ALUs
- Parallel processors

Multiprocessor Systems

Organization of Multiprocessor Systems

❑ Flynn's Classification

- Was proposed by researcher Michael J. Flynn in 1966.
- It is the most commonly accepted taxonomy of computer organization.
- In this classification, computers are classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets.

Taxonomy of Computer Architectures

Simple Diagrammatic Representation

		Single	DATA STREAM	Multiple
		Single	SISD	SIMD
		Multiple	MISD	MIMD
INSTRUCTION STREAM	Single	Single Instruction Single Data		Single Instruction Multiple Data
	Multiple	Multiple Instruction Single Data		Multiple Instruction Multiple Data

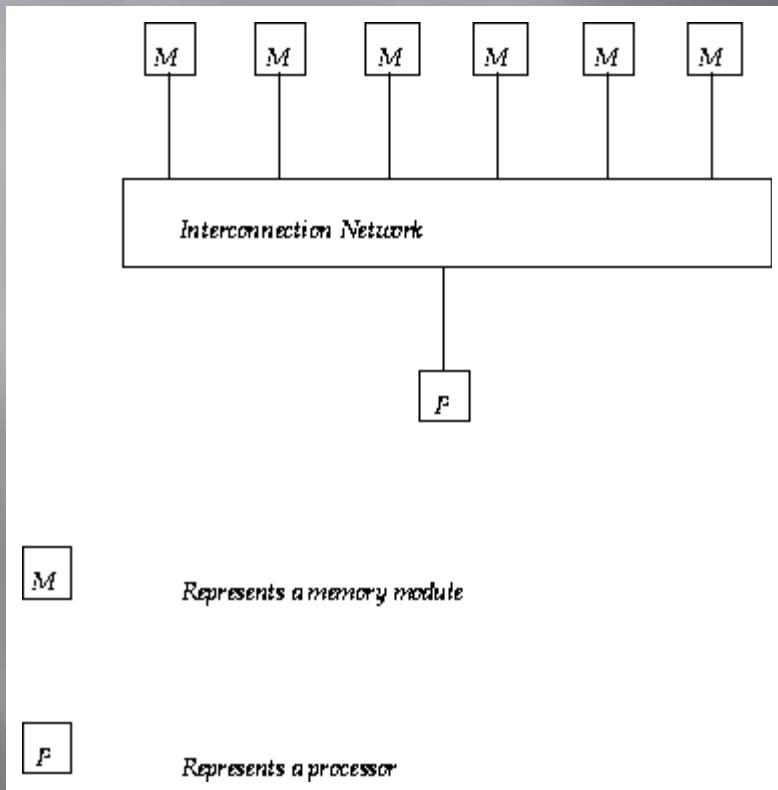
4 categories of Flynn's classification of multiprocessor systems by their instruction and data streams

Single Instruction, Single Data (SISD)

- ❑ SISD machines executes a single instruction on individual data values using a single processor.
- ❑ Based on traditional Von Neumann uniprocessor architecture, instructions are executed sequentially or serially, one step after the next.
- ❑ Until most recently, most computers are of SISD type.

SISD

Simple Diagrammatic Representation

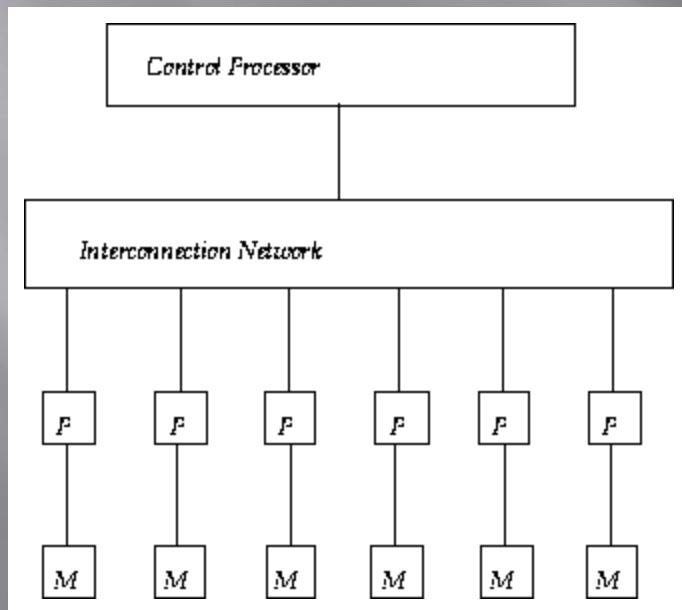


Single Instruction, Multiple Data (SIMD)

- ❑ An SIMD machine executes a single instruction on multiple data values simultaneously using many processors.
- ❑ Since there is only one instruction, each processor does not have to fetch and decode each instruction. Instead, a single control unit does the fetch and decoding for all processors.
- ❑ SIMD architectures include array processors.

SIMD

Simple Diagrammatic Representation

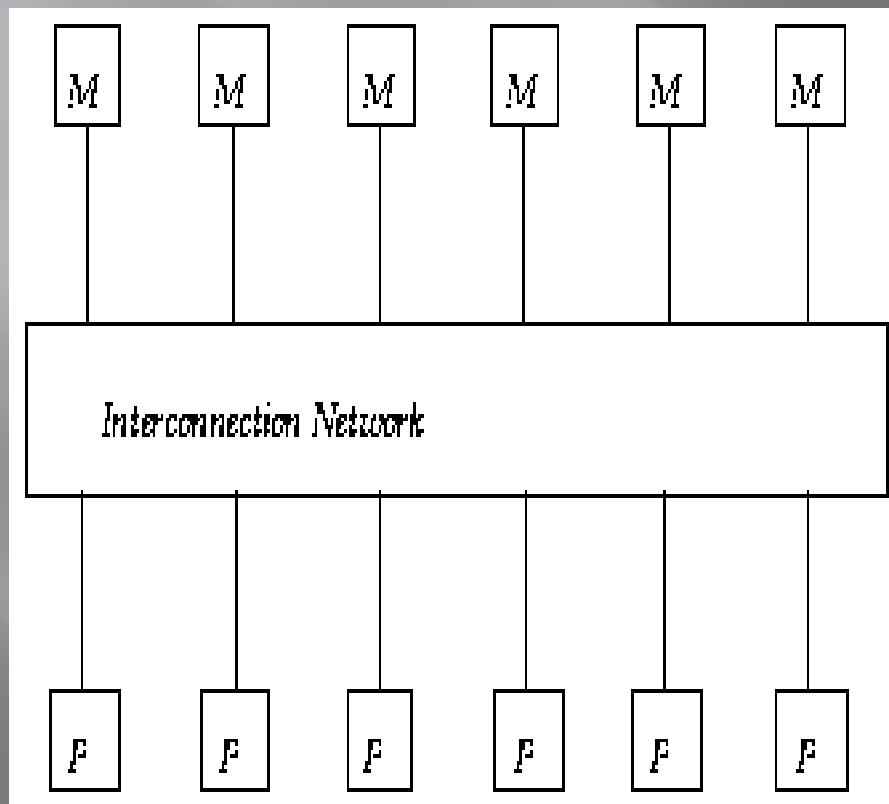


Multiple Instruction, Multiple Data (MIMD)

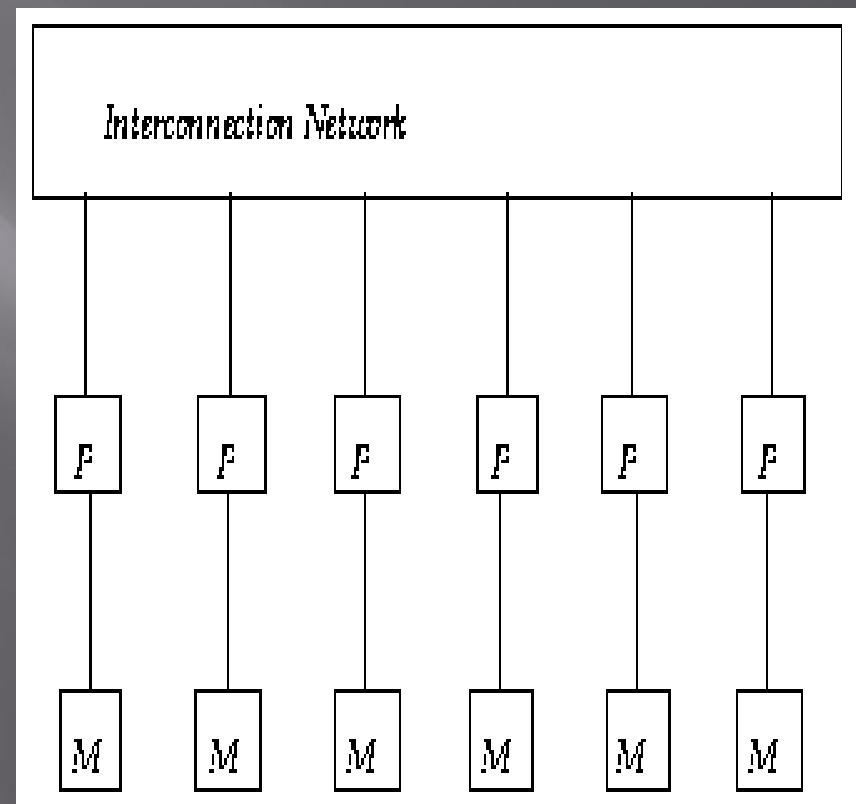
- ❑ MIMD machines are usually referred to as multiprocessors or multicollectors.
- ❑ It may execute multiple instructions simultaneously, contrary to SIMD machines.
- ❑ Each processor must include its own control unit that will assign to the processors parts of a task or a separate task.
- ❑ It has two subclasses: Shared memory and distributed memory

MIMD

Simple Diagrammatic Representation
(Shared Memory)



Simple Diagrammatic Representation(DistributedMemory)



Multiple Instruction, Single Data (MISD)

- ❑ This category does not actually exist. This category was included in the taxonomy for the sake of completeness.

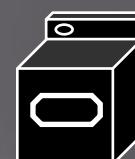
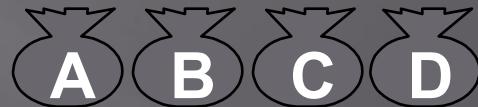
Pipelining

What is Pipelining?

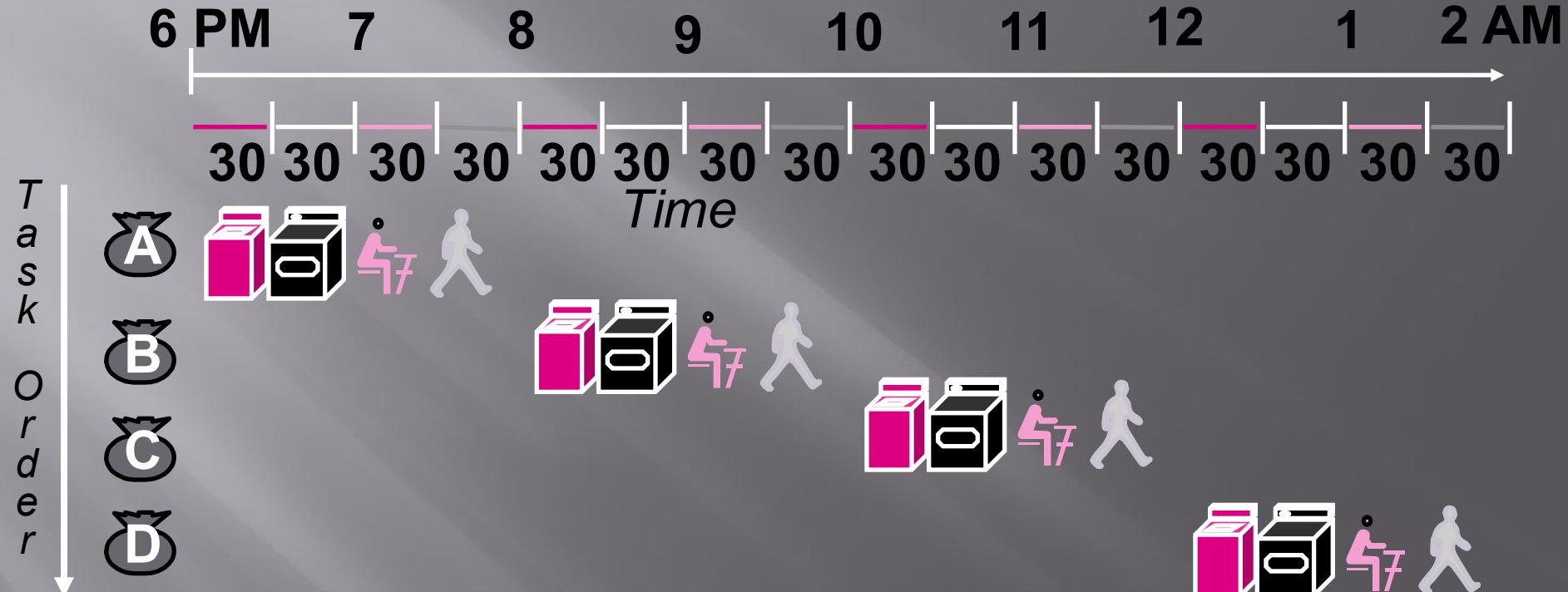
- A way of speeding up execution of instructions
- *Key idea:*
overlap execution of multiple instructions

The Laundry Analogy

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers

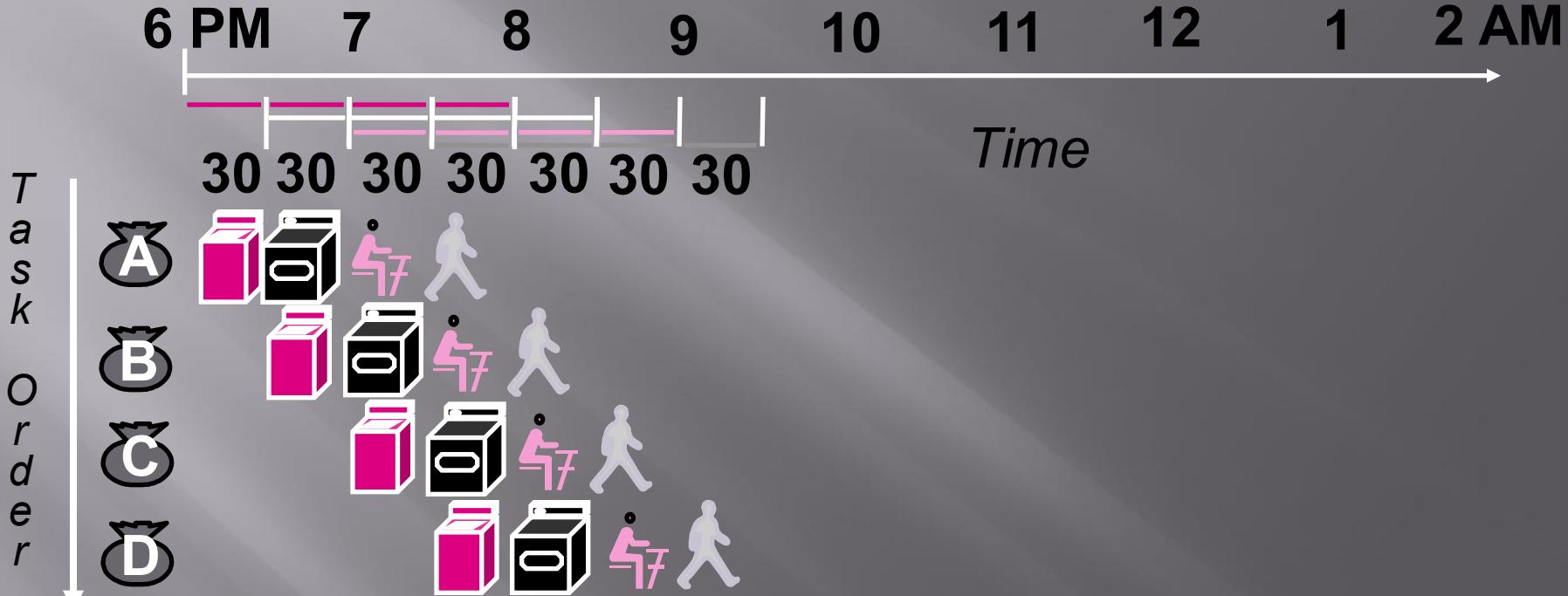


If we do laundry sequentially...



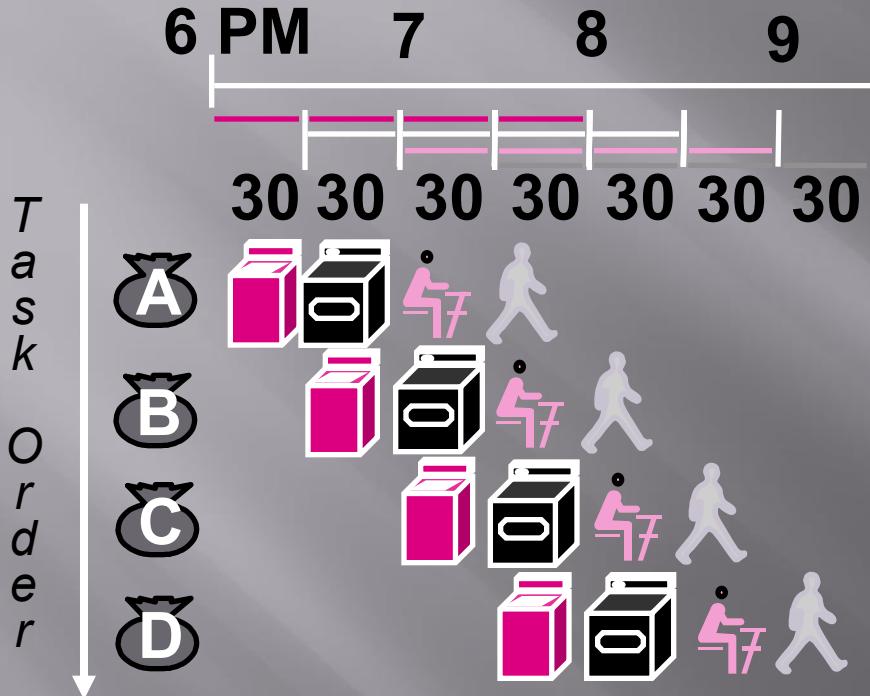
- Time Required: 8 hours for 4 loads

To Pipeline, We Overlap Tasks



- Time Required: 3.5 Hours for 4 Loads

To Pipeline, We Overlap Tasks



Time

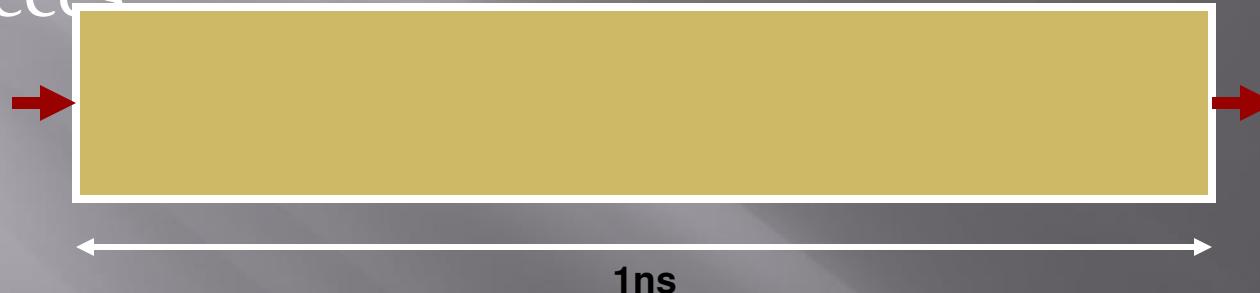
- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

Pipelining a Digital System

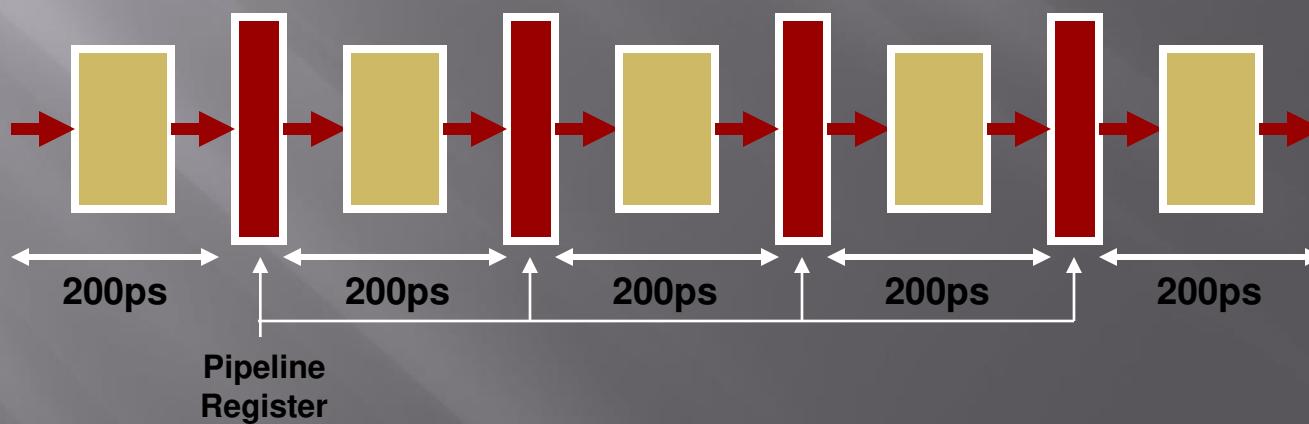
1 nanosecond = 10^{-9} second

1 picosecond = 10^{-12} second

- Key idea: break big computation up into pieces

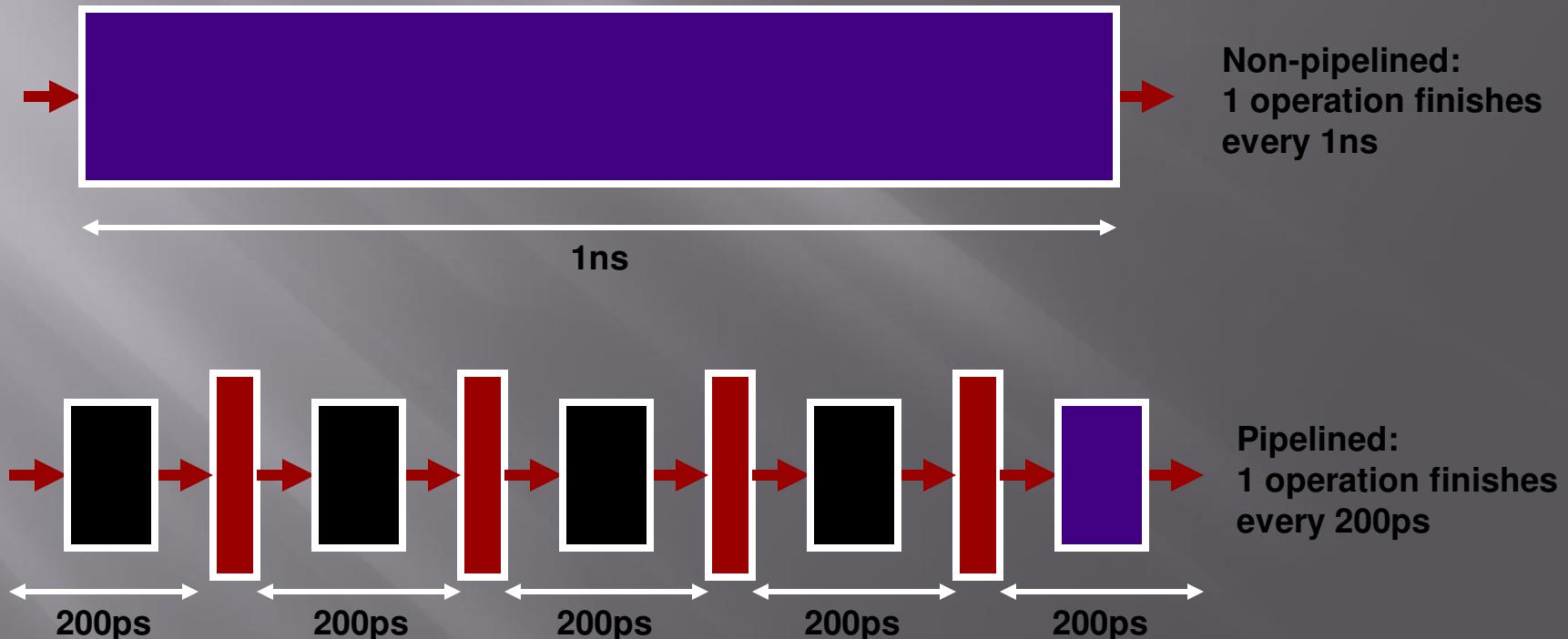


Separate each piece with a pipeline register



Pipelining a Digital System

- Why do this? Because it's faster for repeated computations



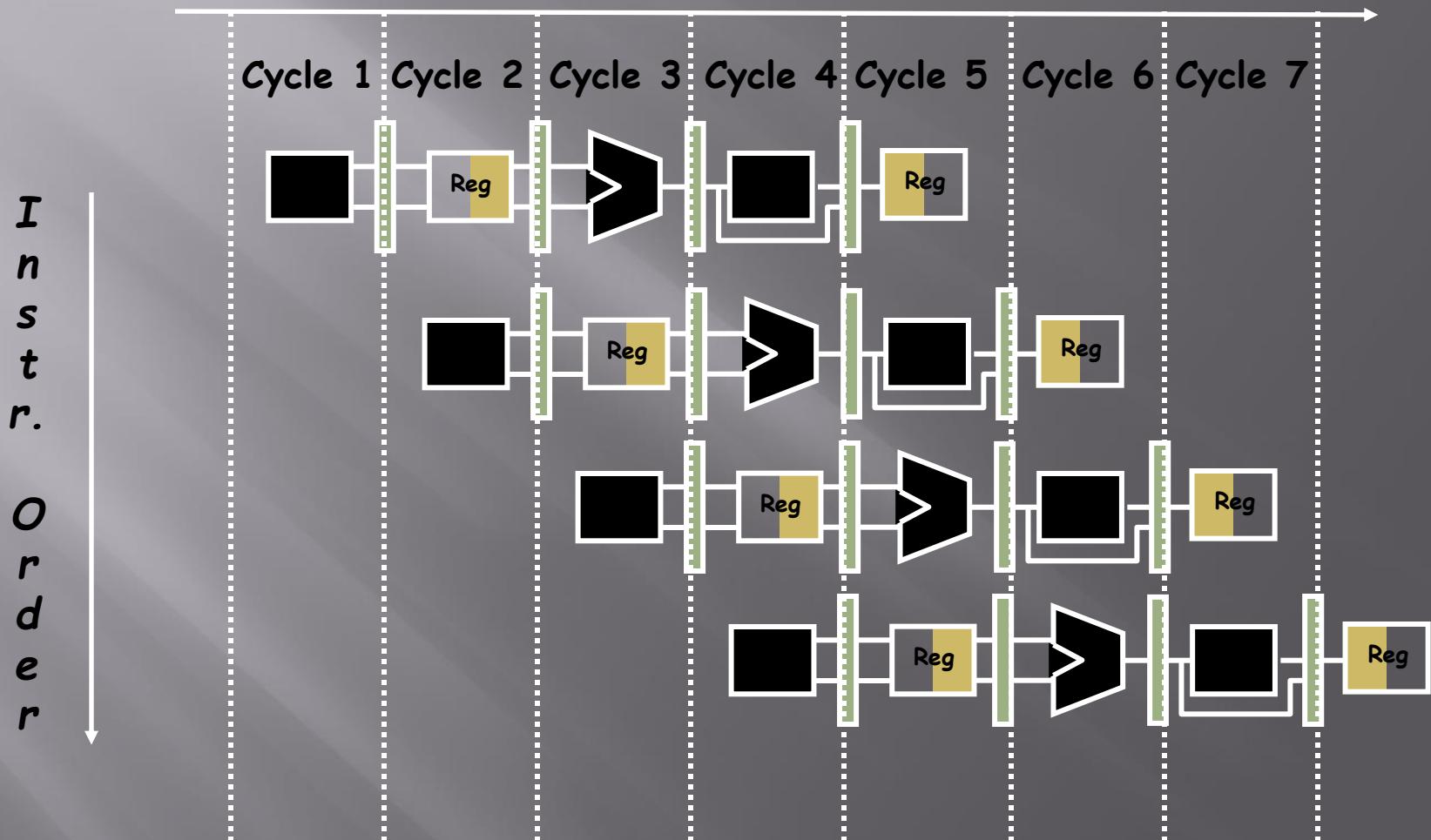
Comments about pipelining

- ❑ Pipelining increases **throughput**, but not **latency**
 - Answer available every 200ps, BUT
 - A single computation still takes 1ns
- ❑ Limitations:
 - Computations must be divisible into stage size
 - Pipeline registers add overhead

Pipelining a Processor

- ❑ Recall the 5 steps in instruction execution:
 1. Instruction Fetch (**IF**)
 2. Instruction Decode and Register Read (**ID**)
 3. Execution operation or calculate address (**EX**)
 4. Memory access (**MEM**)
 5. Write result into register (**WB**)
- ❑ Review: Single-Cycle Processor
 - All 5 steps done in a single clock cycle
 - Dedicated hardware required for each step

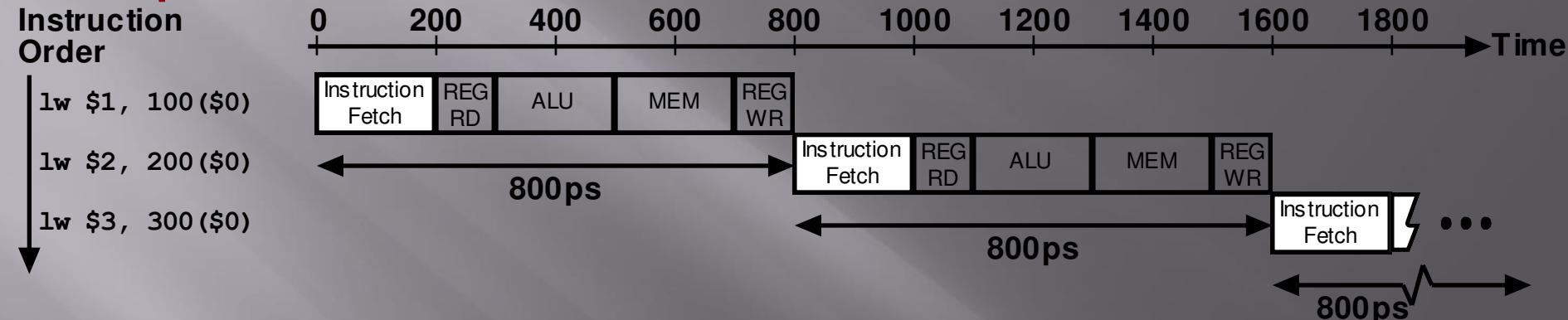
The Basic Pipeline For MIPS



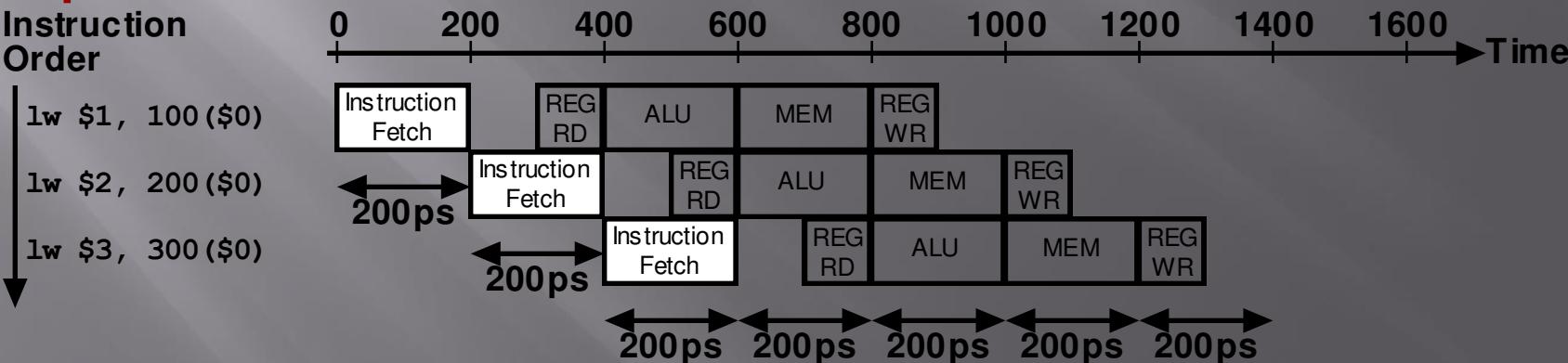
What do we need to add to actually split the datapath into stages?

Single-Cycle vs. Pipelined Execution

Non-Pipelined



Pipelined



Speedup

- Consider the unpipelined processor introduced previously. Assume that it has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations, assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Average instruction execution time

$$\begin{aligned} &= 1 \text{ ns} * ((40\% + 20\%) * 4 + 40\% * 5) \\ &= 4.4 \text{ ns} \end{aligned}$$

Speedup from pipeline

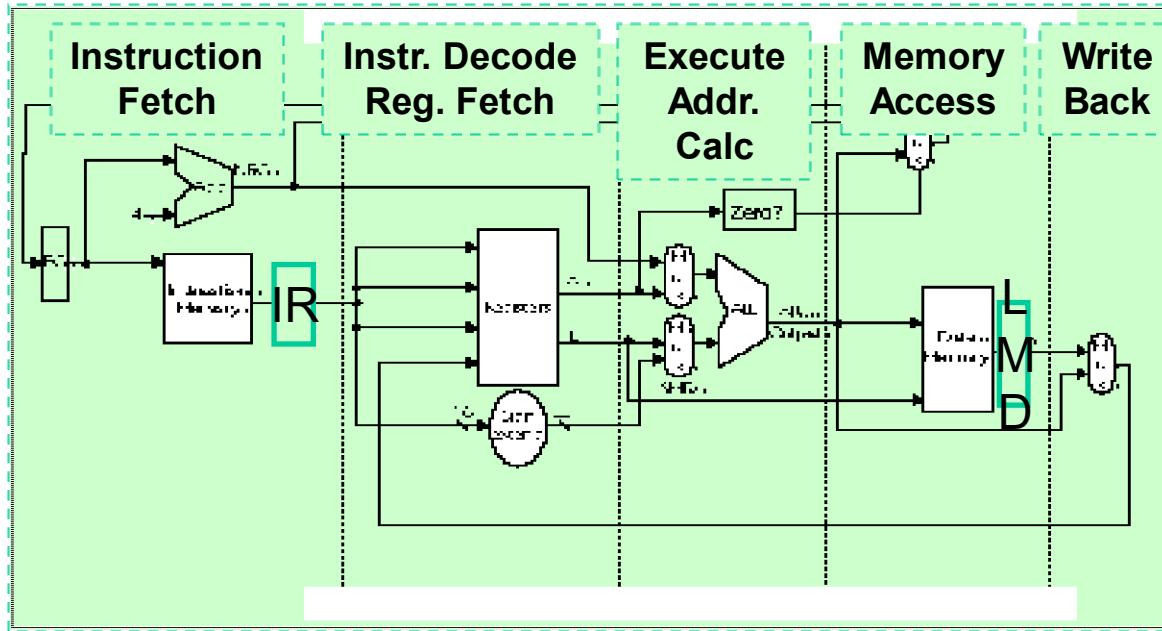
$$\begin{aligned} &= \text{Average instruction time unpiplined} / \text{Average instruction time pipelined} \\ &= 4.4 \text{ ns} / 1.2 \text{ ns} = 3.7 \end{aligned}$$

Comments about Pipelining

- The good news
 - Multiple instructions are being processed at same time
 - This works because stages are isolated by registers
 - Best case speedup of N
- The bad news
 - Instructions interfere with each other - hazards
 - Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
 - Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipelining

MIPS Functions



Passed To Next Stage

IR <- Mem[PC]
NPC <- PC + 4

Instruction Fetch (IF):

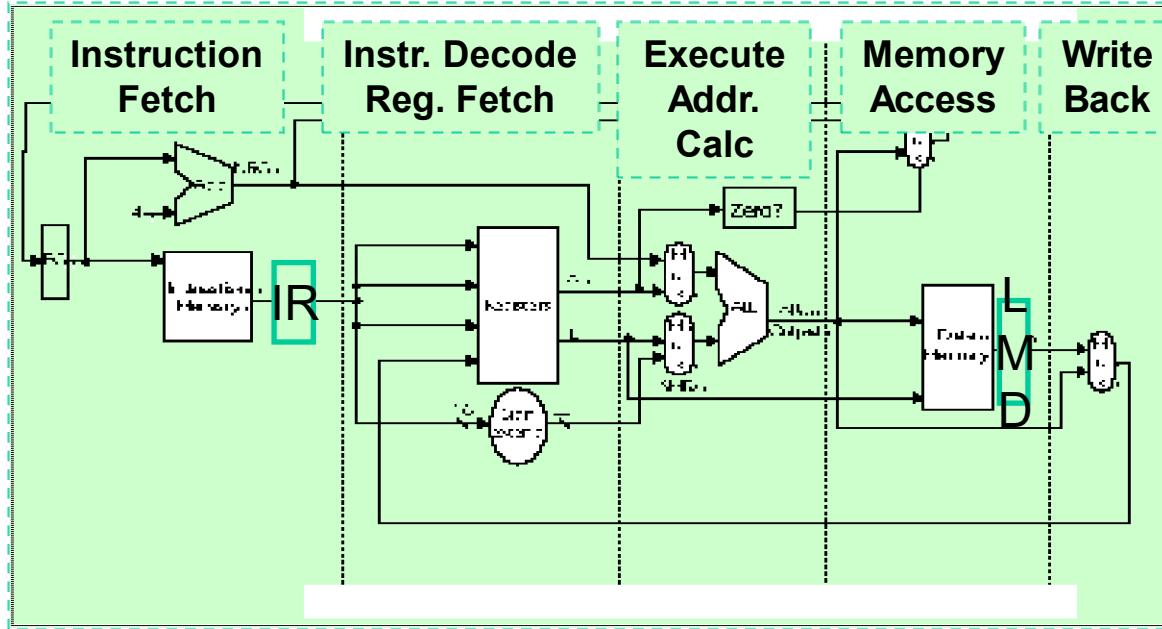
Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

IR holds the instruction that will be used in the next stage.

NPC holds the value of the next PC.

Pipelining

MIPS Functions

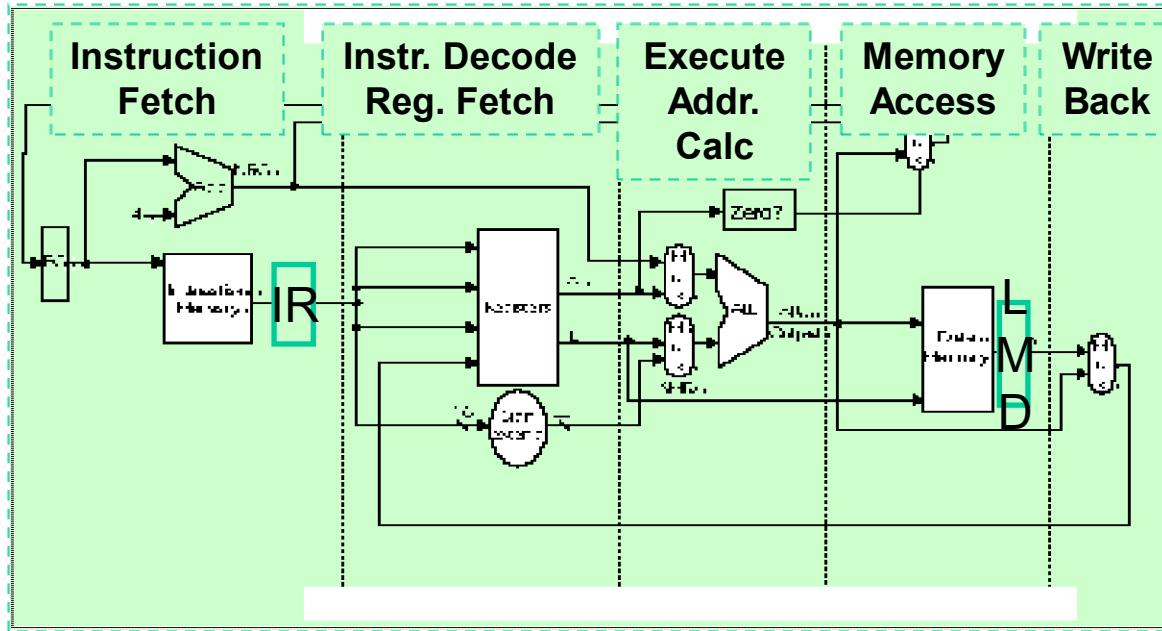


Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers. The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles. We extend the sign of the lower 16 bits of the Instruction Register.

Pipelining

MIPS Functions



Passed To Next Stage
A <- A func. B
cond = 0;

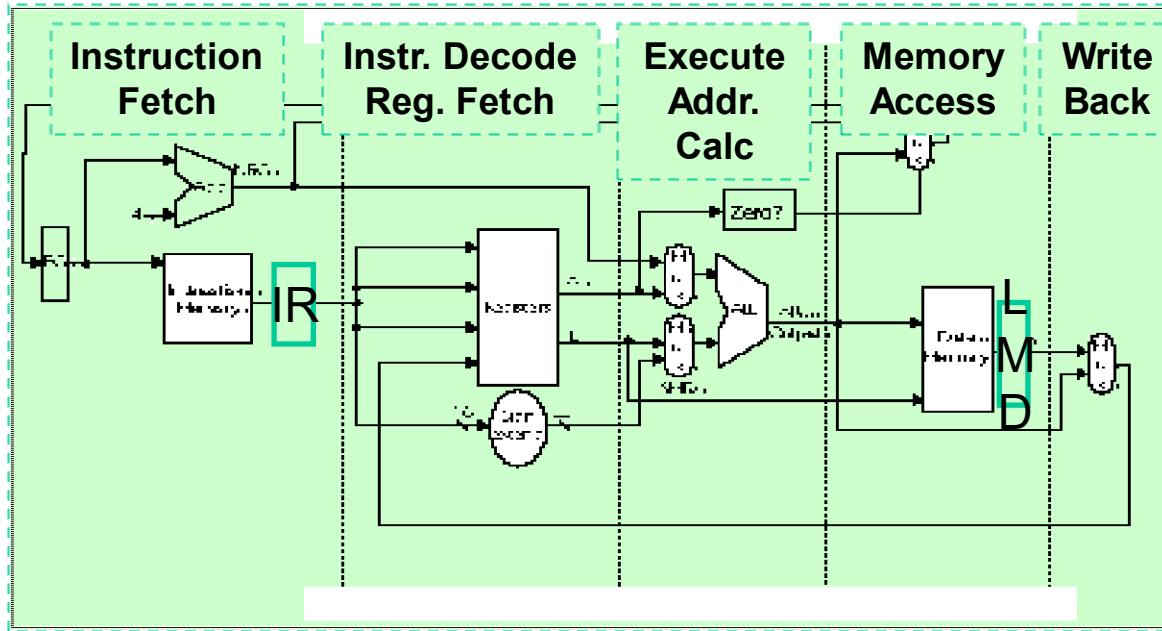
Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

Pipelining

MIPS Functions



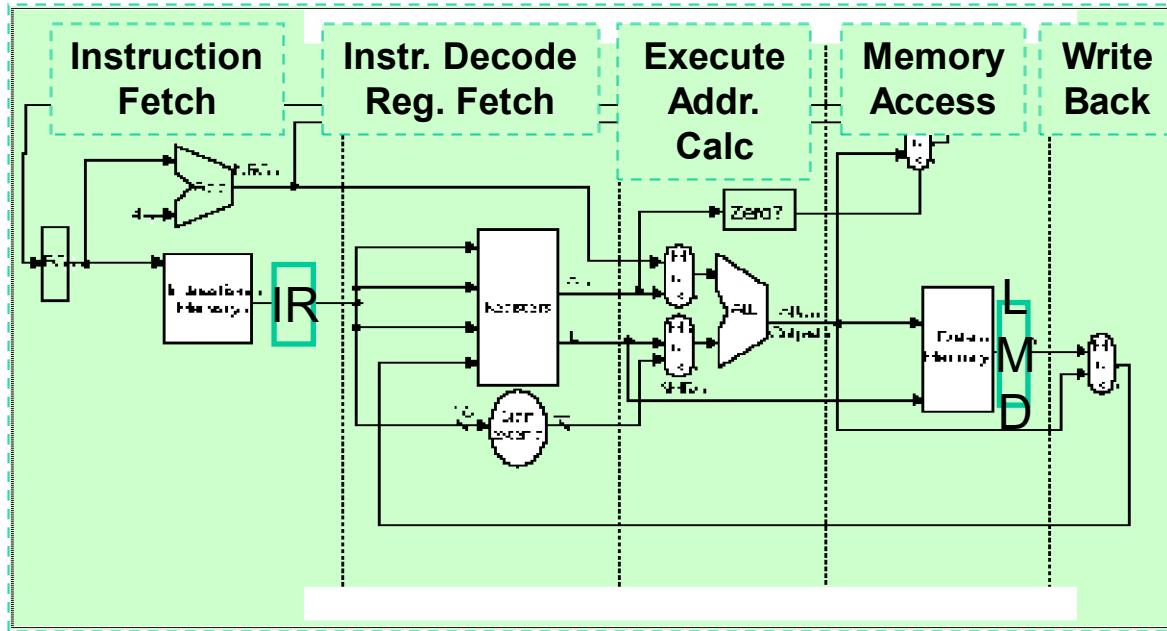
Passed To Next Stage
 $A = \text{Mem}[\text{prev. } B]$
 or
 $\text{Mem}[\text{prev. } B] = A$

MEMORY ACCESS (MEM):

If this is an ALU, do nothing.
 If a load or store, then access memory.

What Is Pipelining

MIPS Functions



WRITE BACK (WB):

Update the registers from either the ALU or from the data loaded.

Pipeline Hazards

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
 - Structural hazards: two different instructions use same h/w in same cycle
 - Data hazards: Instruction depends on result of prior instruction still in the pipeline
 - Control hazards: Pipelining of branches & other instructions that change the PC

Structural Hazards

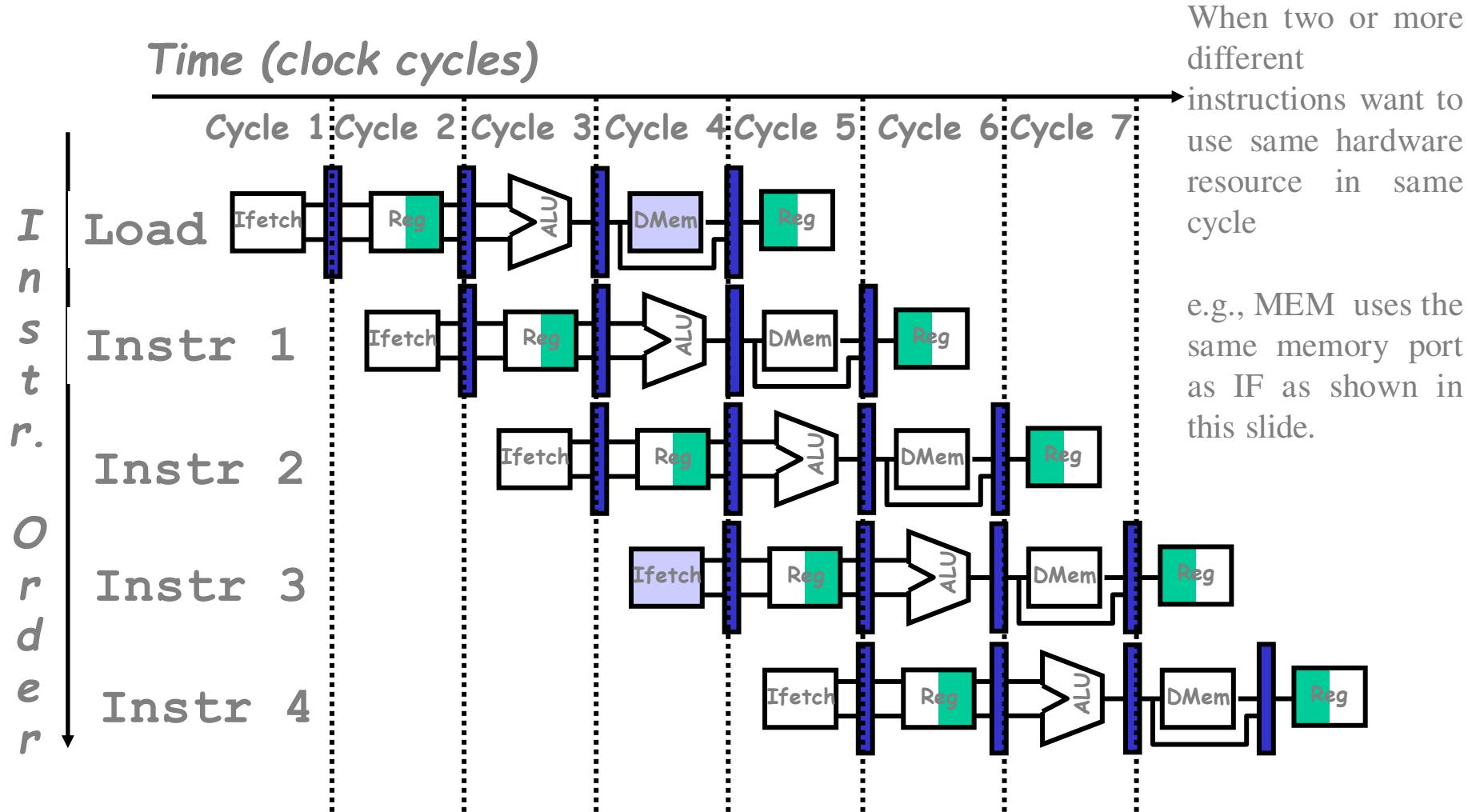


Figure 3.6

Structural Hazards

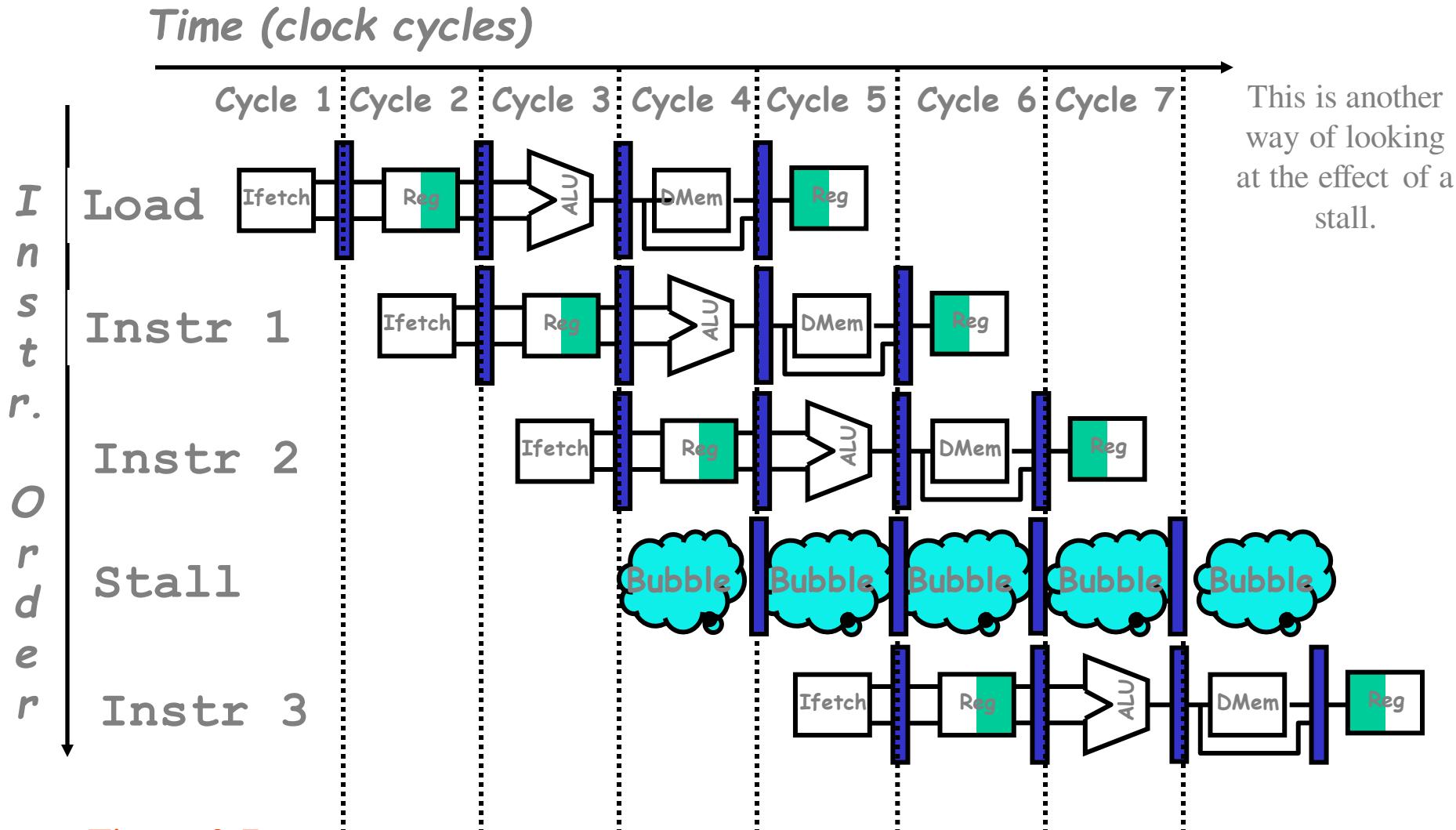


Figure 3.7

Structural Hazards

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction j + 1	IF	ID	EX	MEM	WB					
Instruction j + 2		IF	ID	EX	MEM	WB				
Instruction j + 3			stall	IF	ID	EX	MEM	WB		
Instruction j + 4				IF	ID	EX	MEM	WB		
Instruction j + 5					IF	ID	EX	MEM		
Instruction j + 6						IF	ID	EX		

This is another way to represent the stall we saw on the last few pages.

Structural Hazards

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

Structural hazards are reduced with these rules:

- Each instruction uses a resource at most once
- Always use the resource in the same pipeline stage
- Use the resource for one cycle only

Many RISC ISA's designed with this in mind

Sometimes very complex to do this. For example, memory of necessity is used in the IF and MEM stages.

Some common Structural Hazards:

- Memory - we've already mentioned this one.
- Floating point - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.

Data Hazards

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining- Structural Hazards

- Structural Hazards
 - Data Hazards
 - Control Hazards

A.3 How is Pipelining Implemented

A.4 What Makes Pipelining Hard to Implement?

A.5 Extending the MIPS Pipeline to Handle Multi-cycle Operations

These occur when at any time, there are instructions active that need to access the same data (memory or register) locations.

Where there's real trouble is when we have:

instruction A
instruction B

and B manipulates (reads or writes) data before A does. This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.

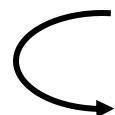
Data Hazards

Execution Order is:

Instr_I
Instr_J

Read After Write (RAW)

Instr_J tries to read operand before Instr_I writes it



I: add r1, r2, r3
J: sub r4, r1, r3

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards

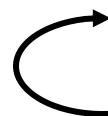
Execution Order is:

Instr_I
Instr_J

Write After Read (WAR)

Instr_J tries to write operand before Instr_I reads it

- Gets wrong operand



I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7

- Called an “anti-dependence” by compiler writers.
This results from reuse of the name “r1”.
- **Can't happen in MIPS 5 stage pipeline because:**
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Data Hazards

Execution Order is:

Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand before Instr_I writes it

- Leaves wrong result (Instr_I not Instr_J)



```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an “output dependence” by compiler writers
This also results from the reuse of name “r1”.
- Can't happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5

Data Hazards

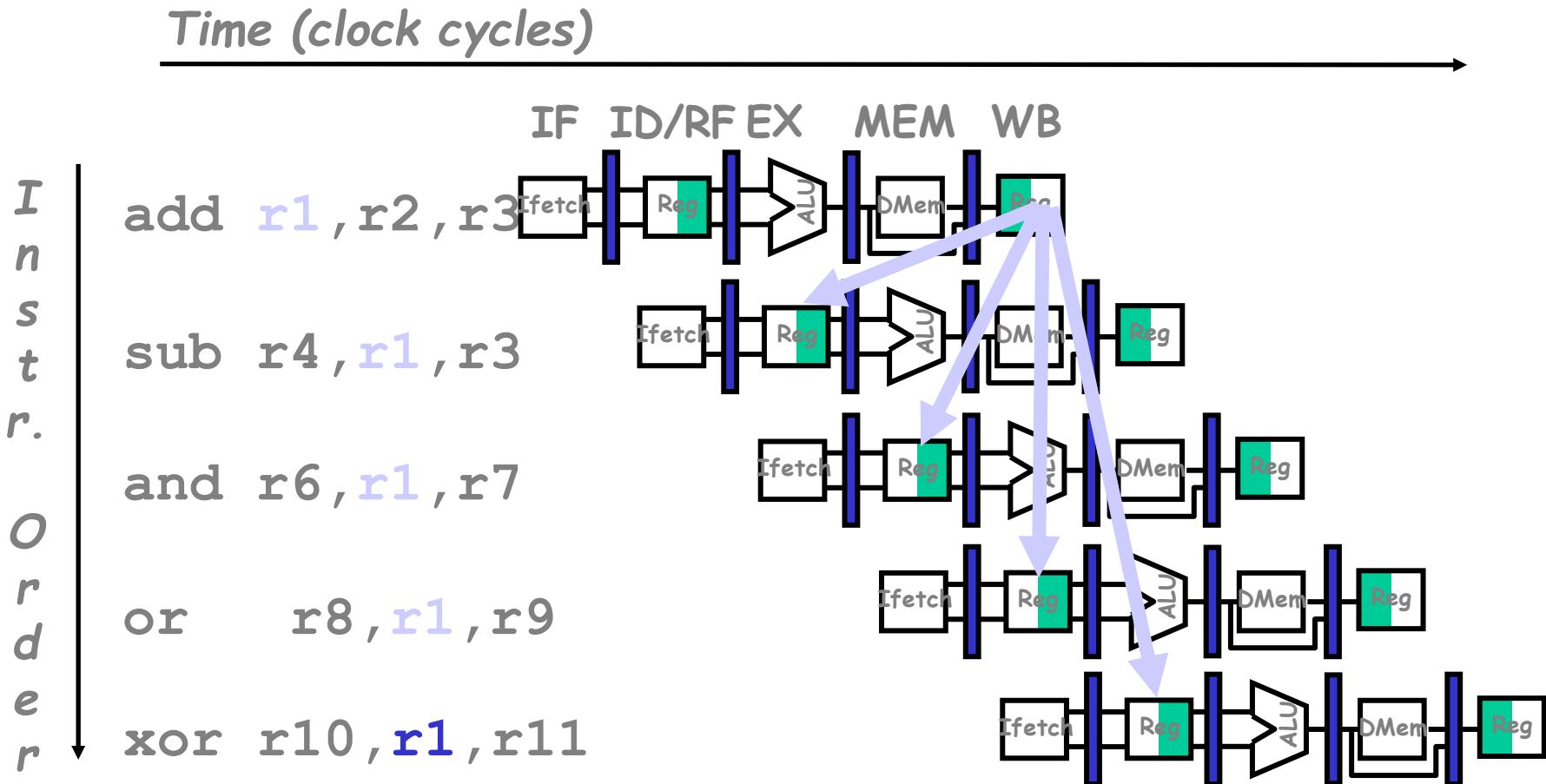
Simple Solution to RAW

- Hardware detects RAW and stalls
- Assumes register written then read each cycle
 - + low cost to implement, simple
 - reduces IPC
 - Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
 - Use data before it is in the register
 - + reduces/avoids stalls
 - complex
 - Crucial for common RAW hazards

Data Hazards



The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Figure 3.9

Data Hazards

Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

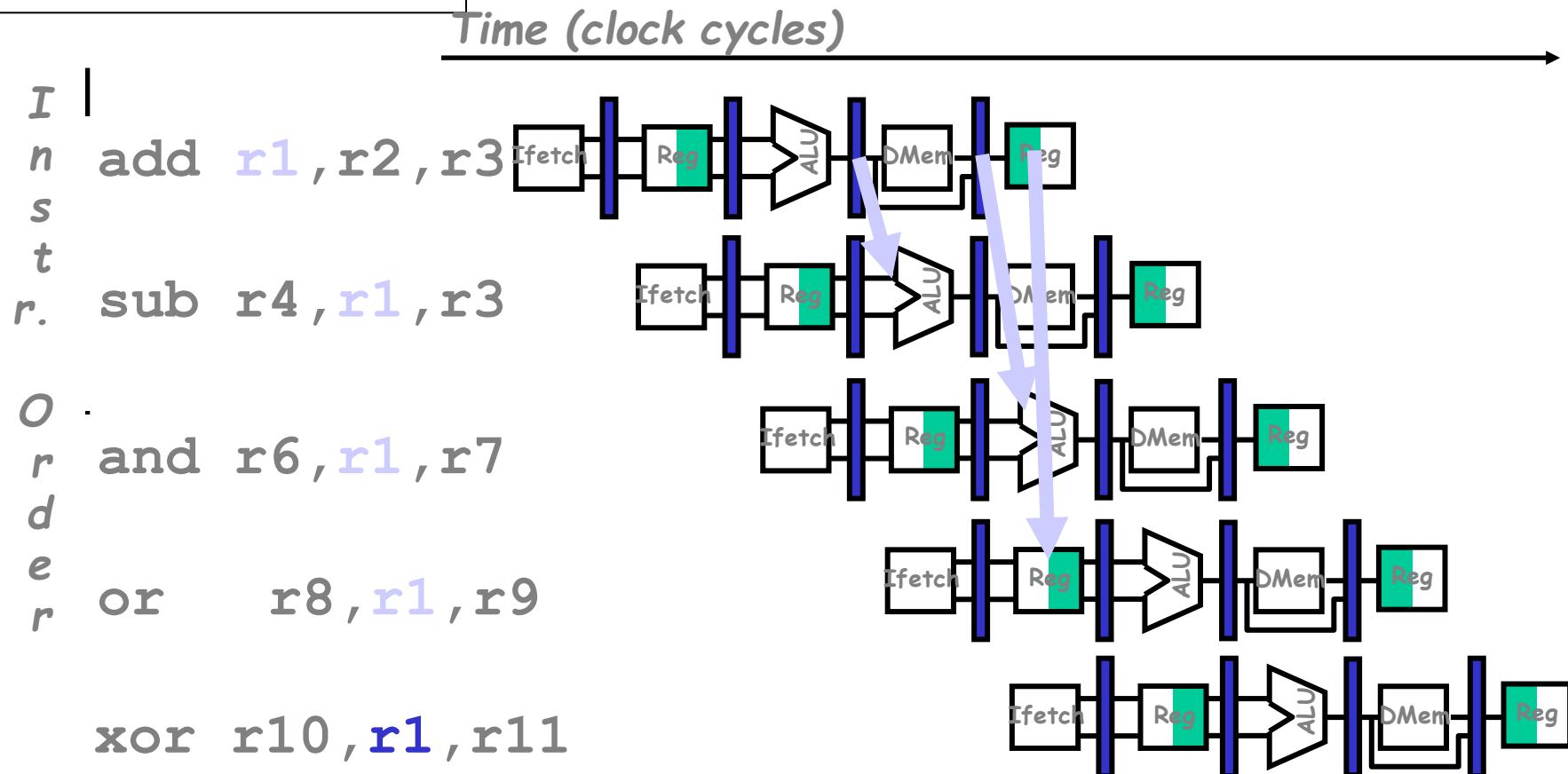
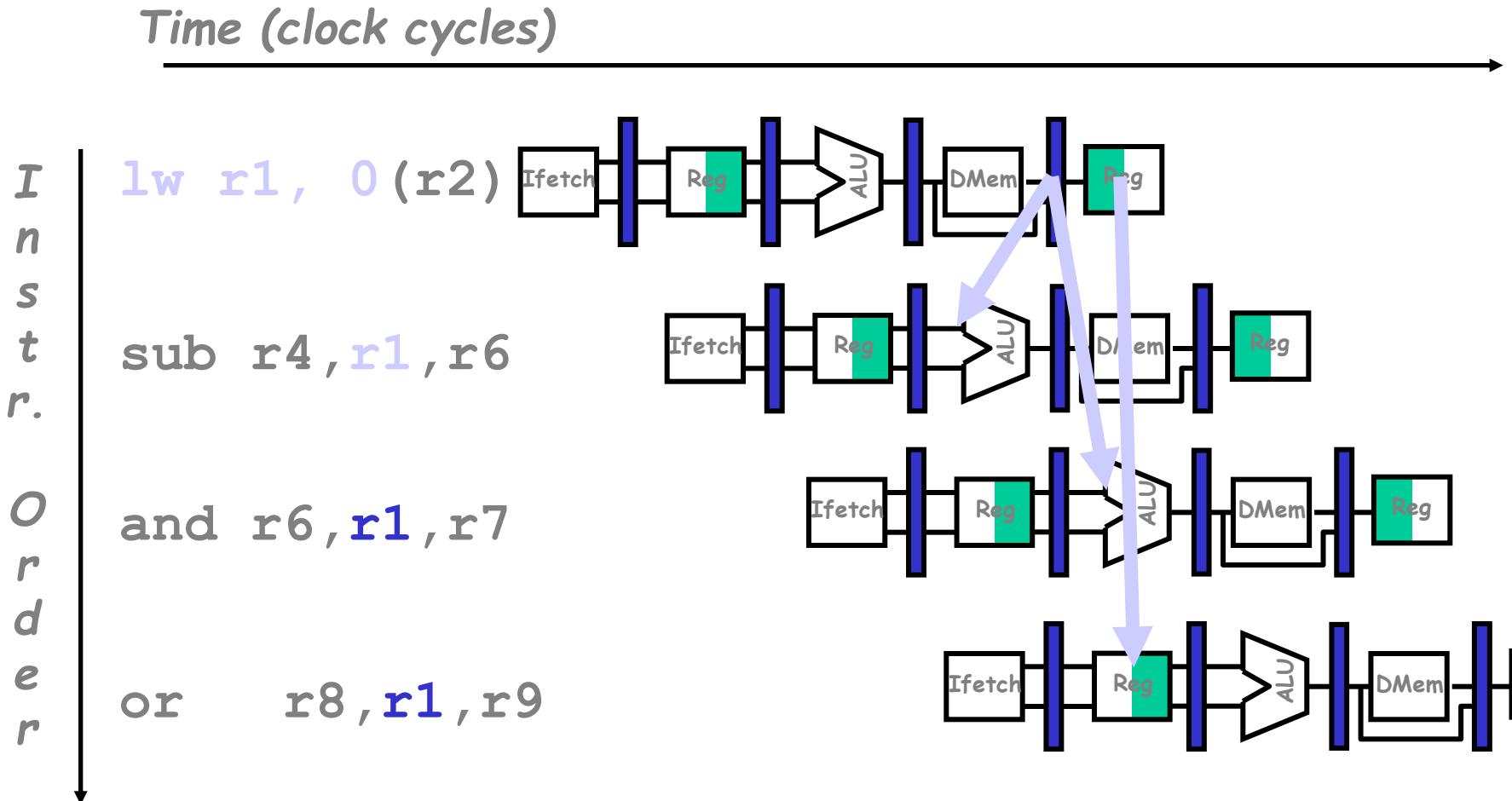


Figure 3.10

The data isn't loaded until after the MEM stage.

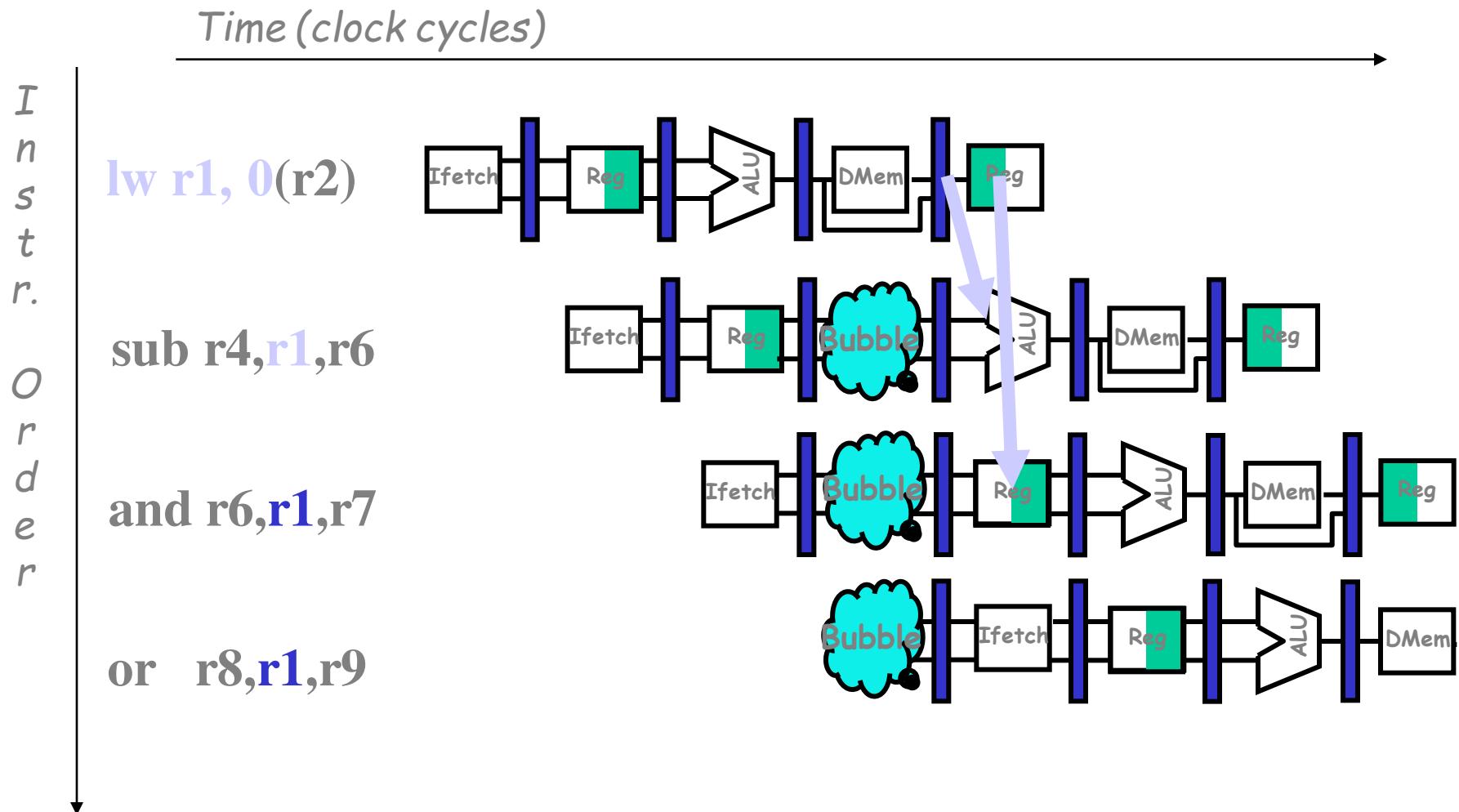


There are some instances where hazards occur, even with forwarding.

Figure 3.12

Data Hazards

The stall is necessary as shown here.



There are some instances where hazards occur, even with forwarding.

Figure 3.13

Data Hazards

This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB	
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB
OR R8, R1, R9				stall	IF	ID	EX	MEM

Control Hazards

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining- Structural Hazards

- Structural Hazards
- Data Hazards
- Control Hazards

A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

Control Hazards

Control Hazard on Branches Three Stage Stall

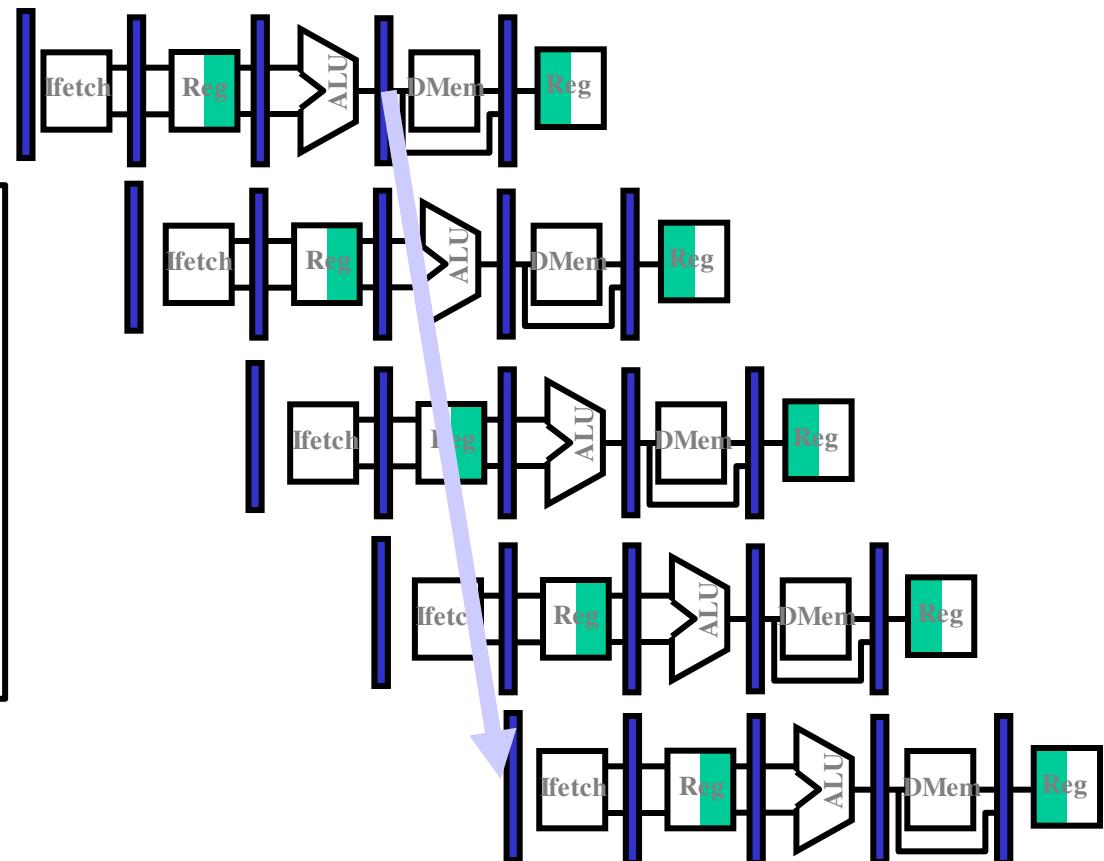
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



Control Hazards

Five Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
 - MIPS still incurs 1 cycle branch penalty
 - Other machines: branch target known before outcome

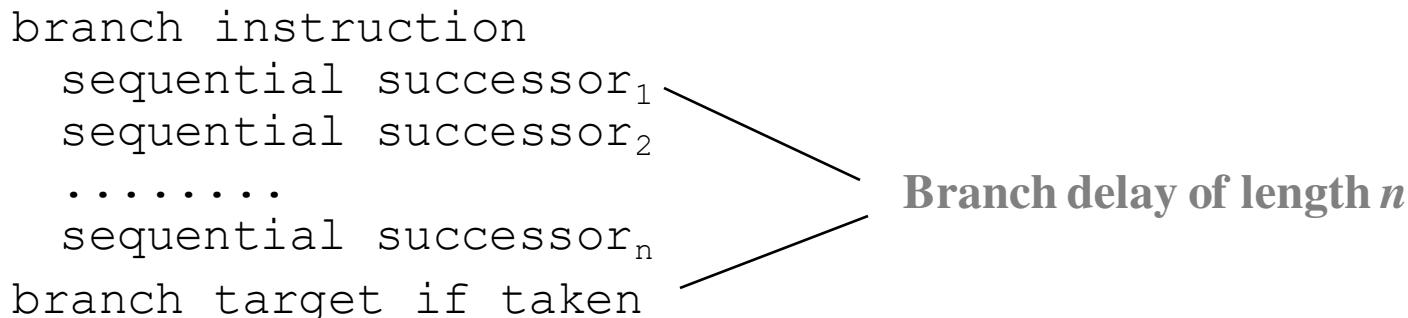
Control Hazards

Five Branch Hazard Alternatives

#4: Execute Both Paths

#5: Delayed Branch

- Define branch to take place **AFTER** a following instruction



- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Control Hazards

Delayed Branch

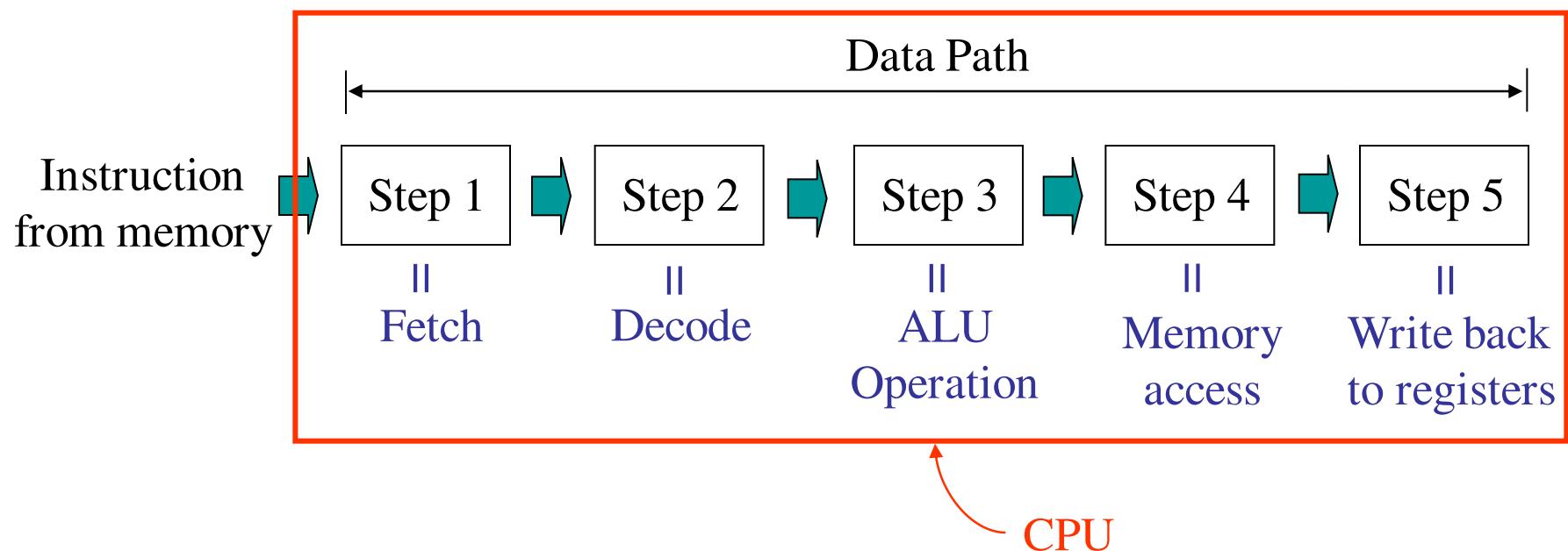
- **Where to get instructions to fill branch delay slot?**
 - Before branch instruction
 - From the target address: only valuable when branch taken
 - From fall through: only valuable when branch not taken
- **Compiler effectiveness for single branch delay slot:**
 - About 80% of instructions executed in branch delay slots useful in computation

Thank you

Data Path:
The way instructions are being
executed inside a processor

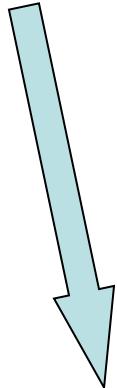
Question: how machine instructions are executed in a processor?

Answer: There are five steps to execute a machine instruction.



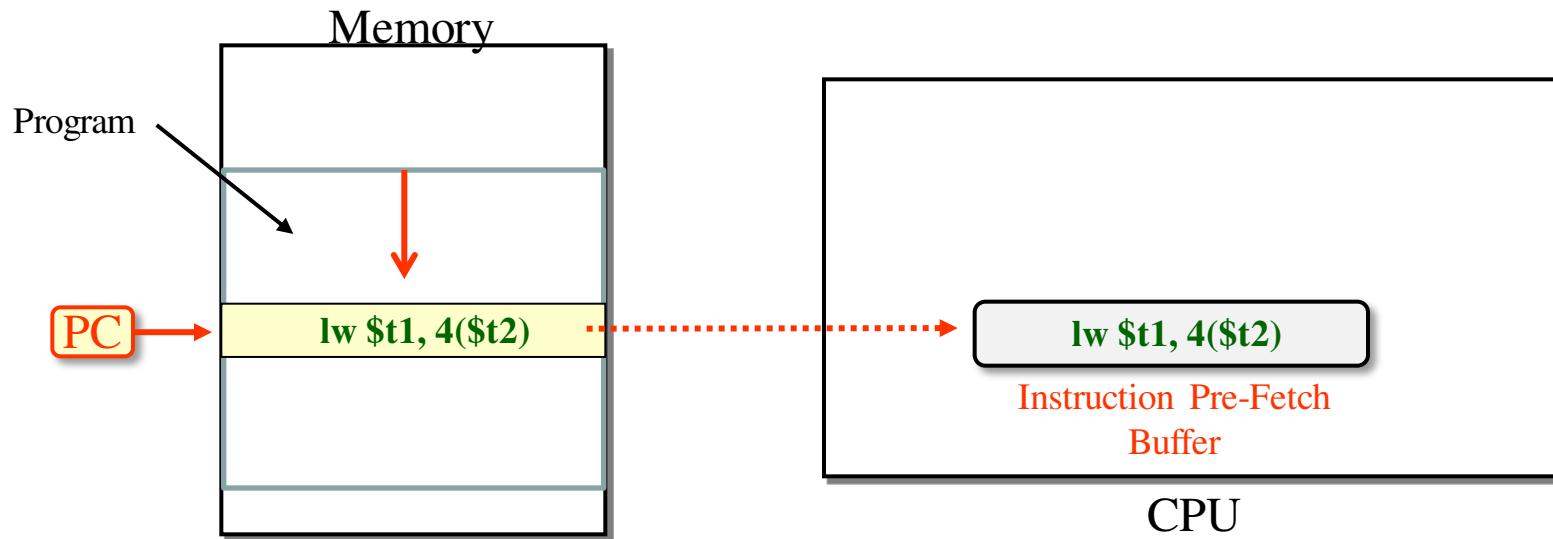
Datapath: a sequence of CPU circuits that execute CPU instructions step by step

Example: Data path for the “load” instruction



lb \$t1, 0(\$t2) # Load one byte from the memory pointed by t2 register

Step 1: Instruction Fetch (IF)



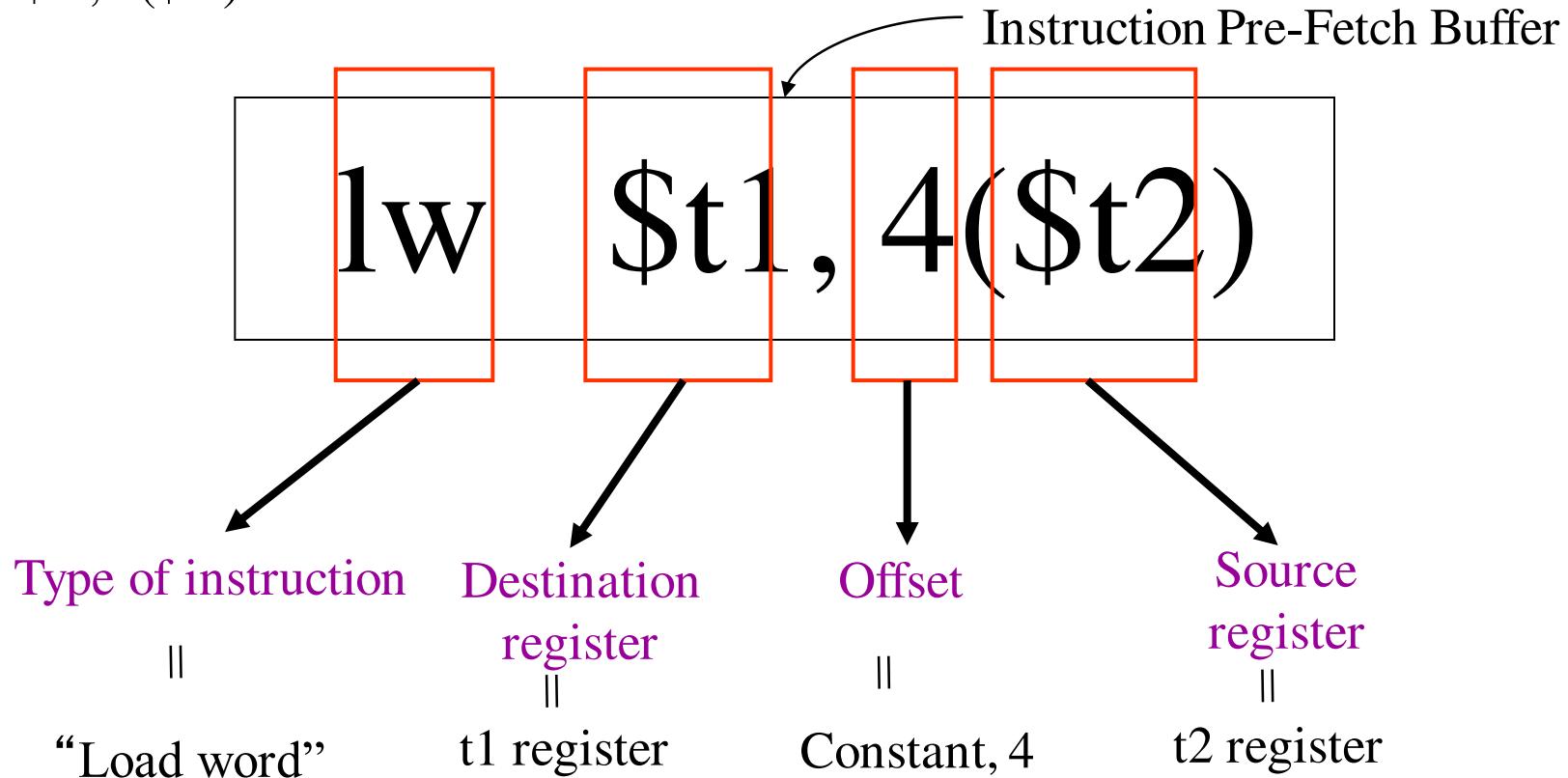
Instruction Fetch (IF):

Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC (by 4) to address the next sequential instruction.

IR holds the instruction that will be used in the next stage.

Step 2: Instruction Decoding (ID)

lw \$t1, 4(\$t2):



Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers. The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.

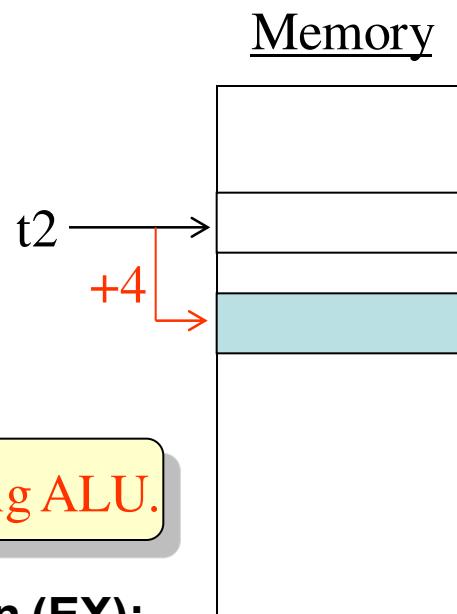
We extend the sign of the lower 16 bits of the Instruction Register.

Step 3: ALU Operation (EX)

lw \$t1, 4(\$t2):

OPERATIONS

- (1) Memory pointed by $t2$
- (2) +4 bytes offset



We need to calculate $t2 + 4$ using ALU.

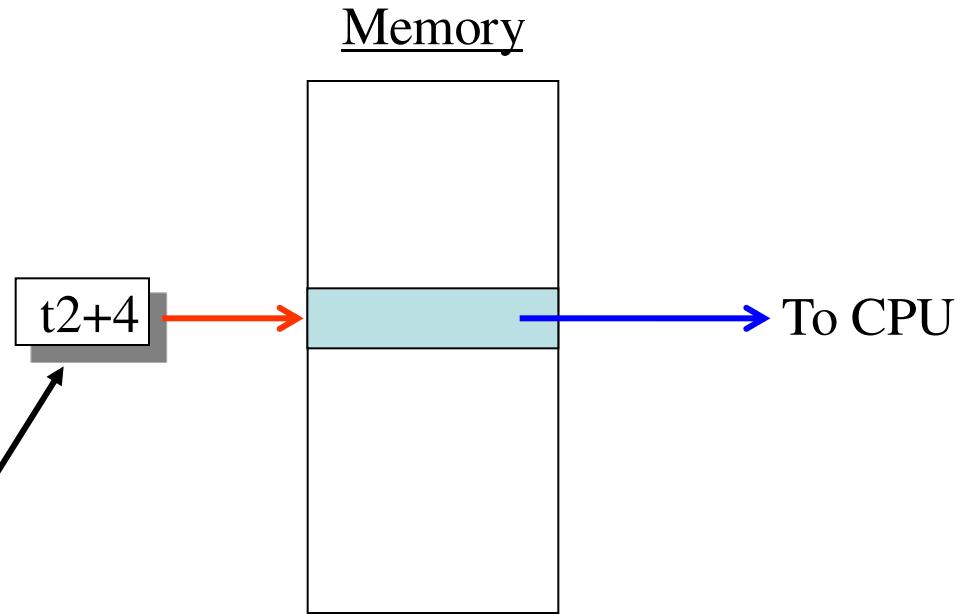
Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

Step 4: Memory Access (ME)

lw \$t1, 4(\$t2):



Calculated by the previous step (= Step 3)

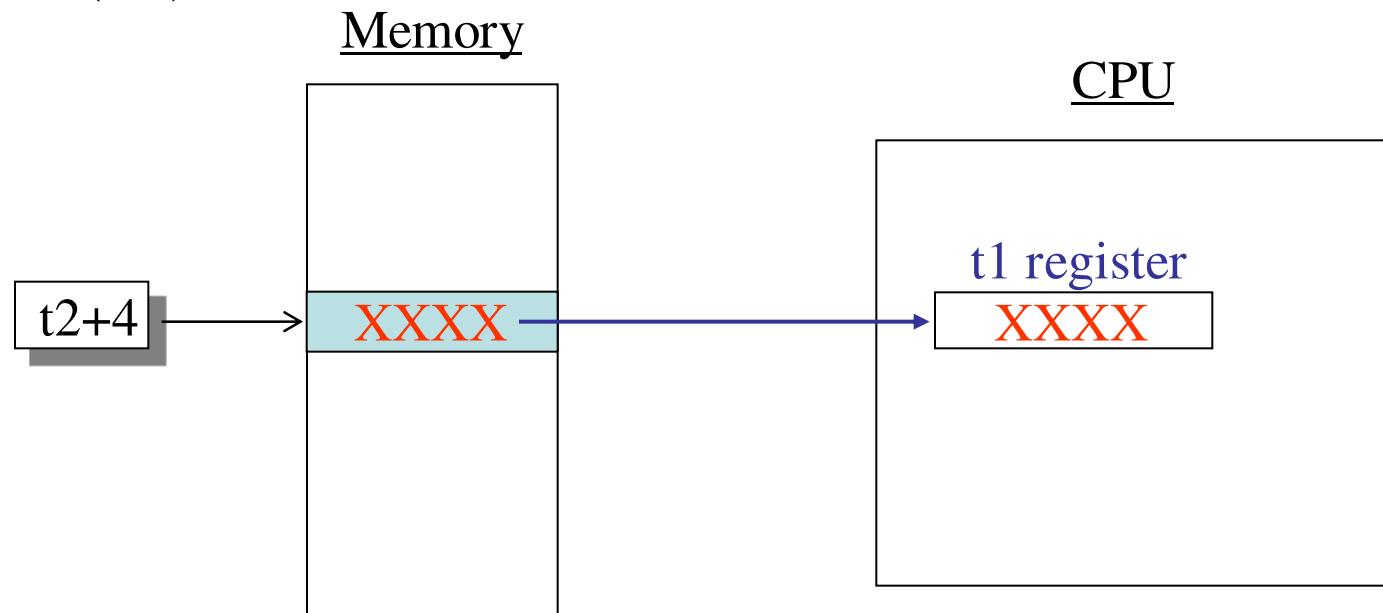
MEMORY ACCESS (MEM):

If this is an ALU, do nothing.

If a load or store, then access memory.

Step 5: Output to the register (WB)

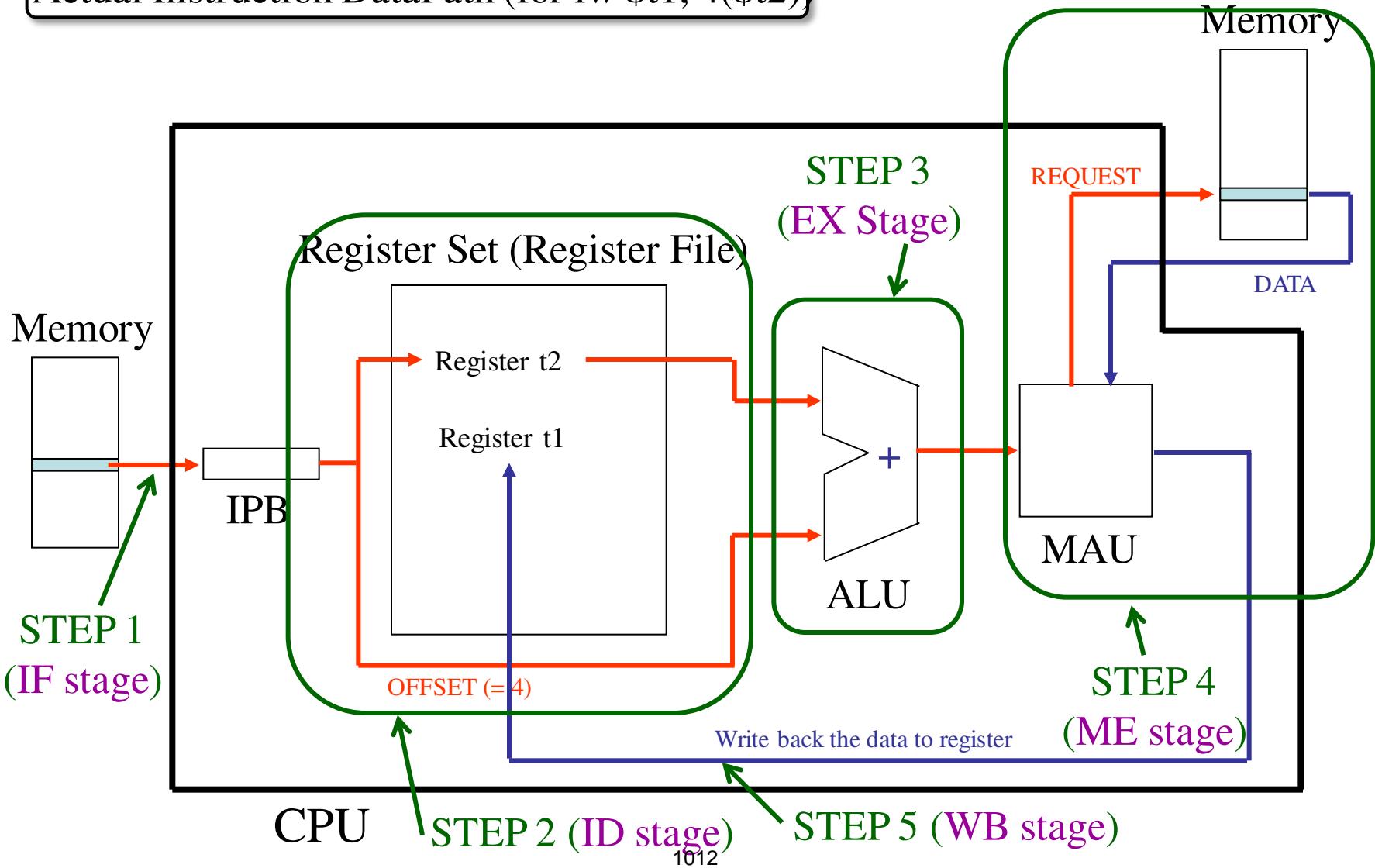
lw \$t1, 4(\$t2):



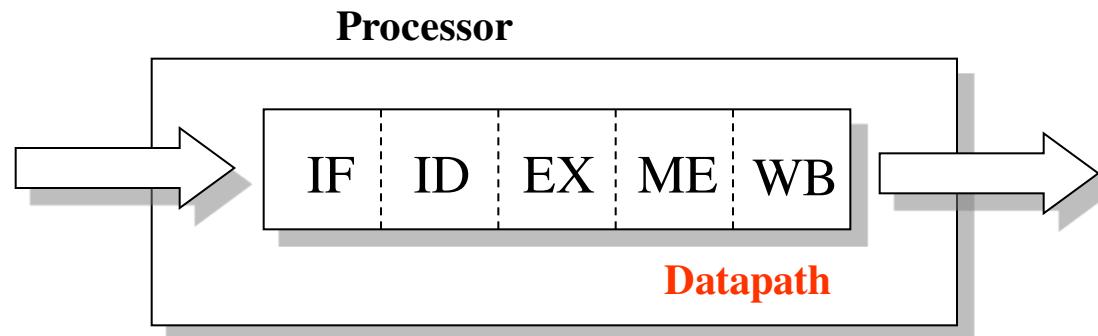
WRITE BACK (WB):

Update the registers from either the ALU or from the data loaded.

Actual Instruction DataPath (for lw \$t1, 4(\$t2))



- Datapath executes processor instruction
- The essential datapath phases are: IF, ID, EX, ME, and WB
- Each datapath phase requires a processor one processor clock cycle

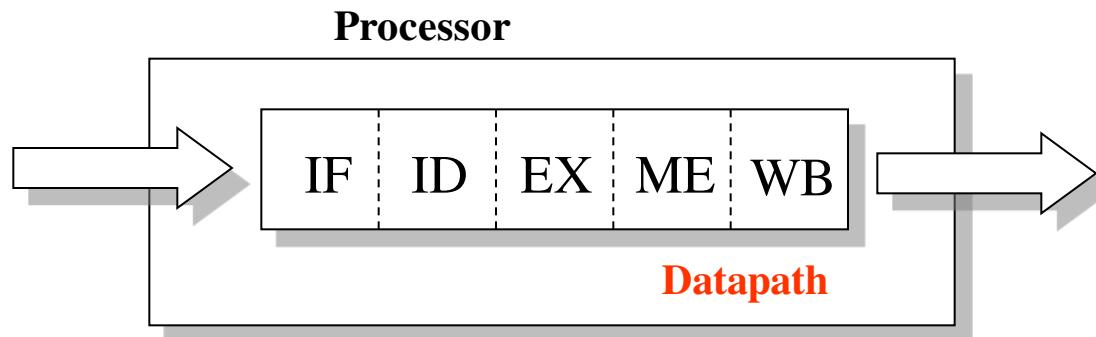


IF: Instruction Fetch **ID:** Instruction Decode **EX:** Execution

ME: Memory access **WB:** Write Back to registers

1. Scalar Datapath Processors

- The datapath includes the five circuit units
- All five units are implemented as single monolithic unit
- When an instruction is being executed, no other instruction can enter the datapath



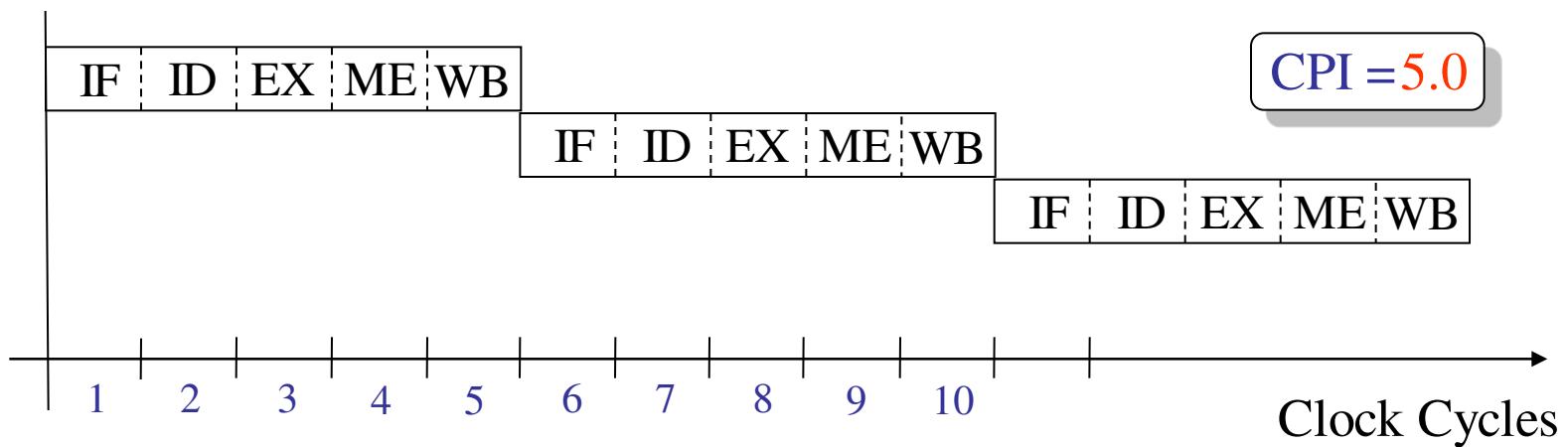
IF: Instruction Fetch **ID:** Instruction Decode **EX:** Execution

ME: Memory access **WB:** Write Back to registers

1. Scalar Datapath Processors

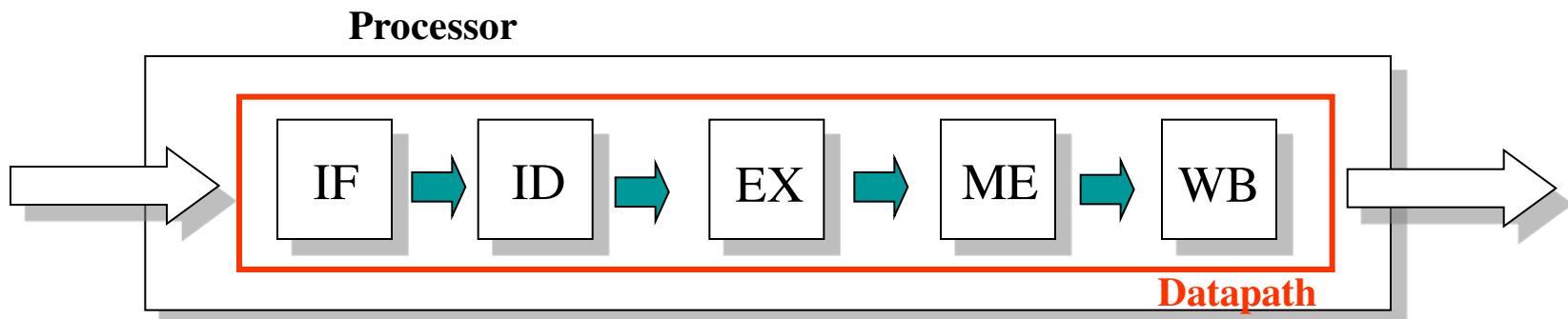
Representative processors in this generation

i4004, i8080, i8086, i80816, Z80, MC68000



2. Pipeline Datapath Processors

- All five units are implemented as independent units
- When an instruction is completed in a unit, the instruction can be forwarded to the next unit
- All five units can be occupied by different instructions



2. Pipeline Datapath Processors (continued)

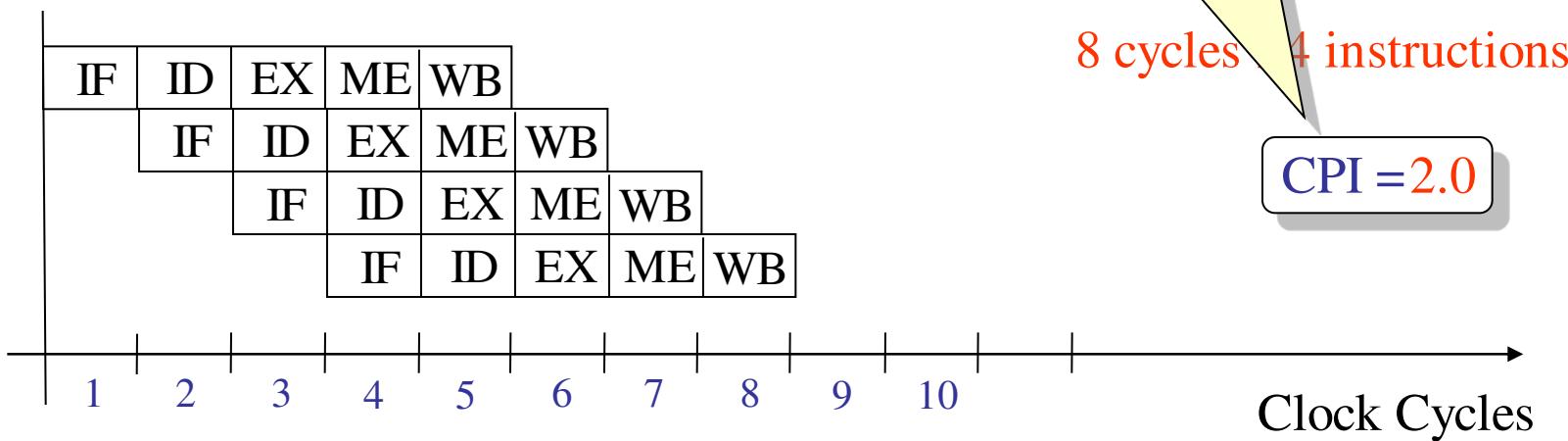
Representative processors in this generation

i80386, i40846, MC68040,

CPI for these only for
these four instructions

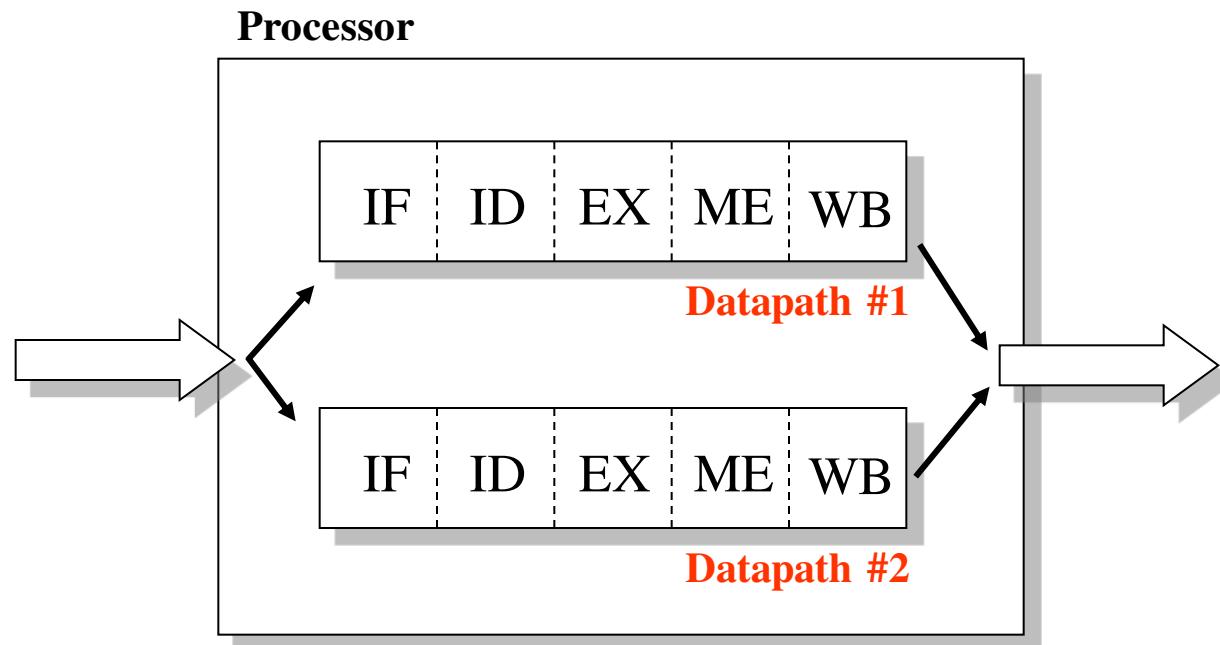
8 cycles 4 instructions

CPI = 2.0

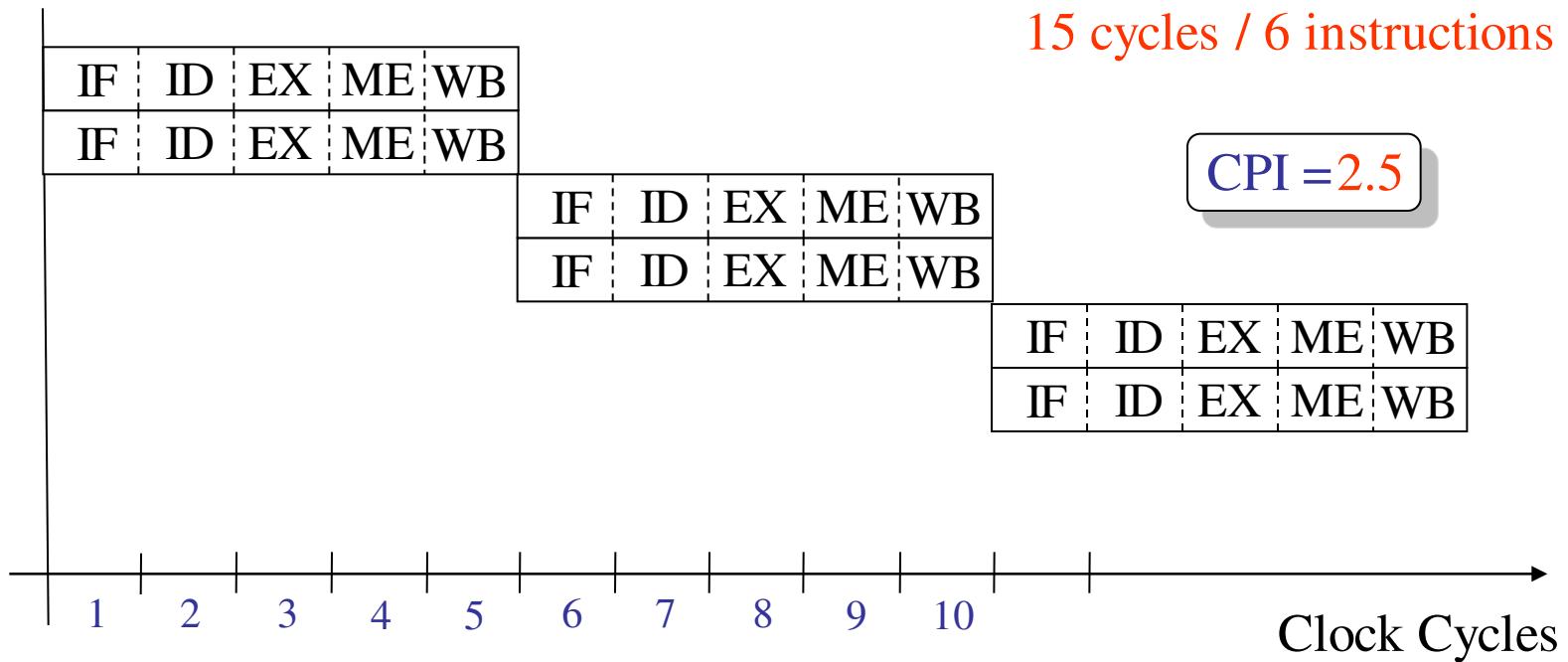


3. Super-Scalar Datapath Processors

- Multiple Scalar Datapath

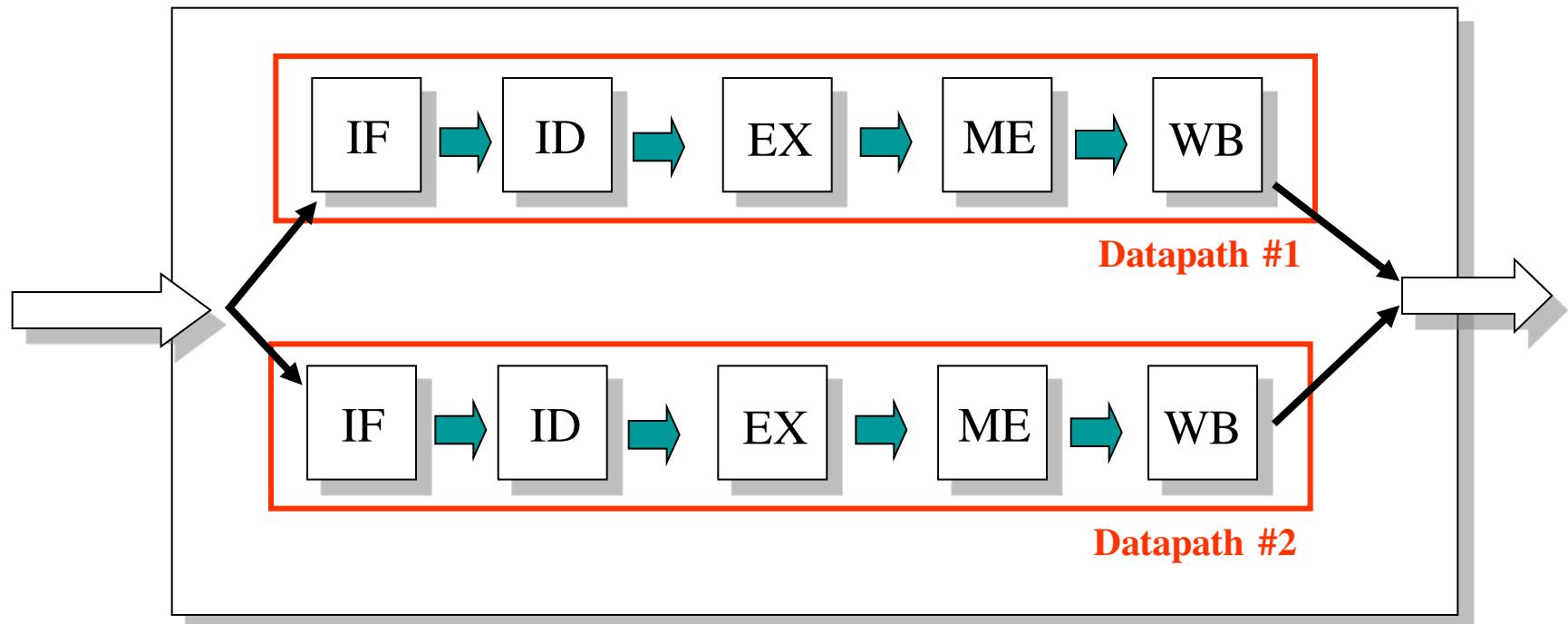


3. Super-Scalar Datapath Processors (continued)



4. Super-Pipeline Datapath Processors

- A combination of super-scalar and pipeline Processor



4. Super-Pipeline Datapath Processors (continued)

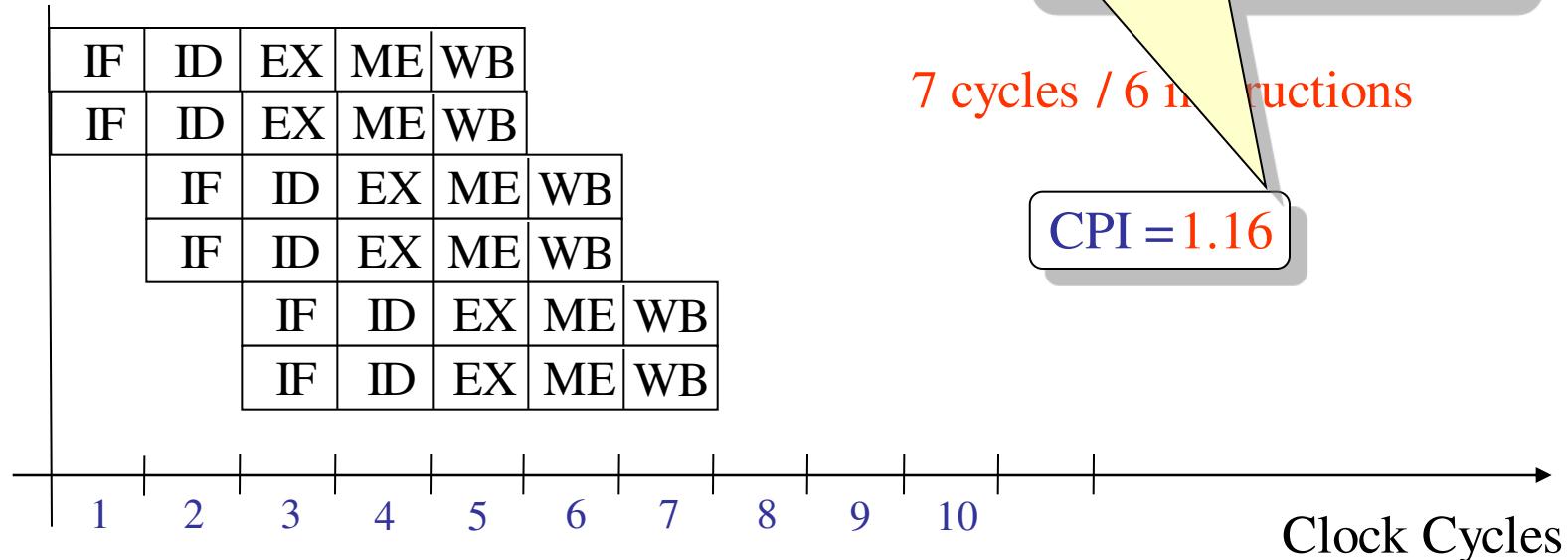
Representative processors in this generation

Pentiums (54, P55C), MIPS

CPI for these only for
these six instructions

7 cycles / 6 instructions

$\text{CPI} = 1.16$



SPEED UP IN PIPELINE

- **For a pipeline processor:**

- *k-stage* pipeline processes with a clock cycle time t_p is used to execute n tasks.
- The time required for the first task T_1 to complete the operation = $k * t_p$ (if k segments in the pipe)
- The time required to complete $(n-1)$ tasks = $(n-1) * t_p$
- Therefore to complete n tasks using a k -segment pipeline requires $=k + (n-1)$ clock cycles.

- **For the non-pipelined processor :**

- Time to complete each task = t_n
- Total time required to complete n tasks = $n * t_n$
- Speed up = non pipelining processing/pipelining processing

$$S = \frac{T_1}{T_K} = \frac{n t_n}{(k + (n - 1)) t_p}$$