

# **Unit-5**

## **CENTRAL PROCESSING UNIT**

**By**  
**Er. Sachita Nand Mishra**  
**M.E. in Computer and Electronics**  
**Engineering**

# CENTRAL PROCESSING UNIT

- **Introduction**
- **General Register Organization**
- **Stack Organization**
- **Instruction Formats**
- **Addressing Modes**
- **Data Transfer and Manipulation**
- **Program Control**
- **Reduced Instruction Set Computer**

# Introduction

- **Part of computer that performs the bulk of data processing operations is called the Central processing Unit(CPU). It Consists of 3 major parts:**
  - **Register set:** stores intermediate data during execution of an instruction .
  - **ALU:** performs various microoperations required
  - **Control unit:** supervises register transfers and instructs ALU

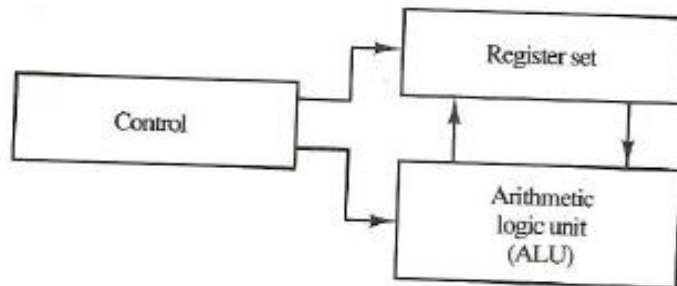


Fig: Major components of CPU

# MAJOR COMPONENTS OF CPU

- **Storage Components**

Registers

Flags

- **Execution (Processing) Components**

Arithmetic Logic Unit(ALU)

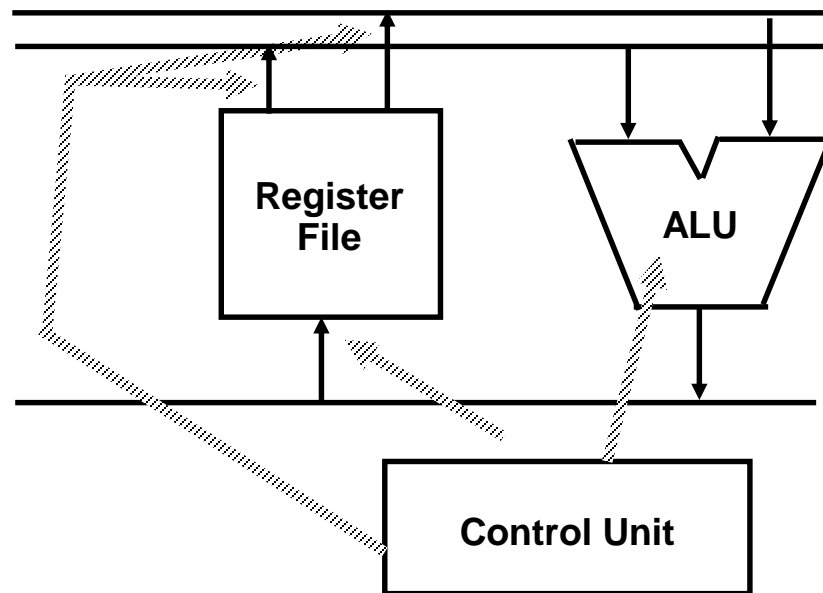
Arithmetic calculations, Logical computations, Shifts/Rotates

- **Transfer Components**

Bus

- **Control Components**

Control Unit

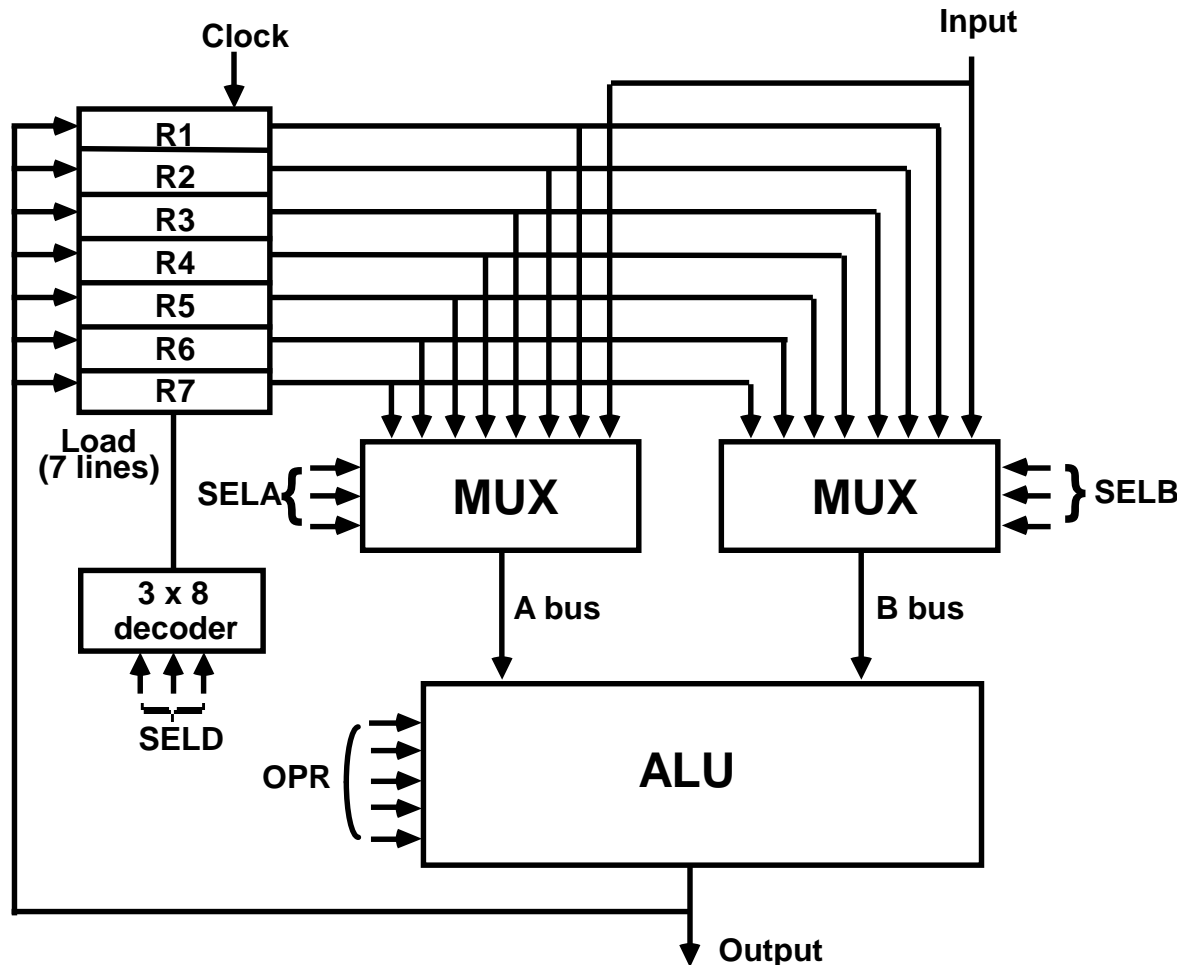


# REGISTERS

- In Basic Computer, there is only one general purpose register, the Accumulator (AC)
- In modern CPUs, there are many general purpose registers
- It is advantageous to have many registers
  - Transfer between registers within the processor are relatively fast
  - Going “off the processor” to access memory is much slower
- ***Why we need CPU registers?***
  - During instruction execution, we could store pointers, counters, return addresses, temporary results and partial products in some locations in RAM, but having to refer memory locations for such applications is time consuming compared to instruction cycle. So for convenient and more efficient processing, we need processor registers (connected through common bus system) to store intermediate results.

# GENERAL REGISTER ORGANIZATION

A bus organization of seven CPU registers is shown below:



**Fig: Block diagram(register organization)**

➤ All registers are connected to two multiplexers (MUX) that select the registers for bus A and bus B.

➤ Registers selected by multiplexers are sent to ALU.

➤ Another selector (OPR) connected to ALU selects the operation for the ALU.

➤ Output produced by ALU is stored in some register and this destination register for storing the result is activated by the destination decoder (SELD).

# OPERATION OF CONTROL UNIT

## The control unit

Directs the information flow through ALU by

- Selecting various *Components* in the system
- Selecting the *Function* of ALU

Example:  $R1 \leftarrow R2 + R3$

[1] MUX A selector (SELA):  $BUS\ A \leftarrow R2$

[2] MUX B selector (SELB):  $BUS\ B \leftarrow R3$

[3] ALU operation selector (OPR): ALU to ADD

[4] Decoder destination selector (SELD):  $R1 \leftarrow Out\ Bus$

## Control Word

3	3	3	5
SELA	SELB	SELD	OPR

- There are 14 binary selection inputs in the unit and their combined value Specifies a control word.
- Combination of all selection bits of a processing unit is called control word.
- Control Word for above CPU is as above:
- The 14 bit control word when applied to the selection inputs specify a particular microoperation.
- It consists of 4 fields.
- The 3 bits of SELA select a source register for the A input of ALU.
- The 3 bits of SELB select a register for B input of ALU.
- The 3 bits of SELD select a destination register using the decoder and its seven load outputs.
- The 5 bits of OPR select one of the operation in the ALU

# OPERATION OF CONTROL UNIT

- **Encoding of register selection fields**

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

- When SELA or SELB is 000, corresponding multiplexer selects the external Input data.
- When SELD is 000, no destination register is selected but the content of Output bus are available in the external output.



# ALU CONTROL

## Encoding of ALU operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

## Examples of ALU Microoperations

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	-	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	-	R7	TSFA	001 000 111 00000
$\text{Output} \leftarrow R2$	R2	-	None	TSFA	010 000 000 00000
$\text{Output} \leftarrow \text{Input}$	Input	-	None	TSFA	000 000 000 00000
$R4 \leftarrow \text{shl } R4$	R4	-	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

# STACK ORGANIZATION

- A Stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
- This is useful last-in, first-out (LIFO) list (actually storage device) included in most CPU's.
- Stack in digital computers is essentially a memory unit with a stack pointer (SP).
- The register that holds the address of stack is called stack pointer(SP) because its value always points at the top item in the stack.
- SP is simply an address register that points stack top.
- Two operations of a stack are the insertion (push) and deletion (pop) of items.
- In a computer stack, nothing is pushed or popped; these operations are simulated by incrementing or decrementing the SP register.
- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation

# REGISTER STACK ORGANIZATION

## Register Stack

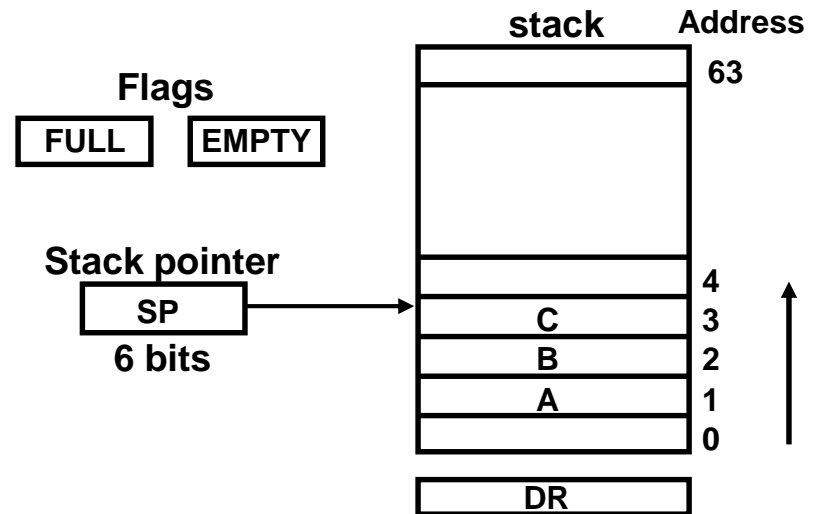
It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of stack

Diagram shows 64-word register stack.

6-bit address SP points stack top.

Currently 3 items are placed in the stack: A, B and C do that content of SP is now 3 (actually 000011).

1-bit registers FULL and EMTY are set to 1 when the stack is full and empty respectively. DR is data register that holds the binary data to be written into or read out of the stack.



## Push, Pop operations

*/\* Initially, SP = 0, EMPTY = 1, FULL = 0 \*/*

### PUSH

$SP \leftarrow SP + 1$

$M[SP] \leftarrow DR$

If  $(SP = 0)$  then  $(FULL \leftarrow 1)$

$EMPTY \leftarrow 0$

### POP

$DR \leftarrow M[SP]$

$SP \leftarrow SP - 1$

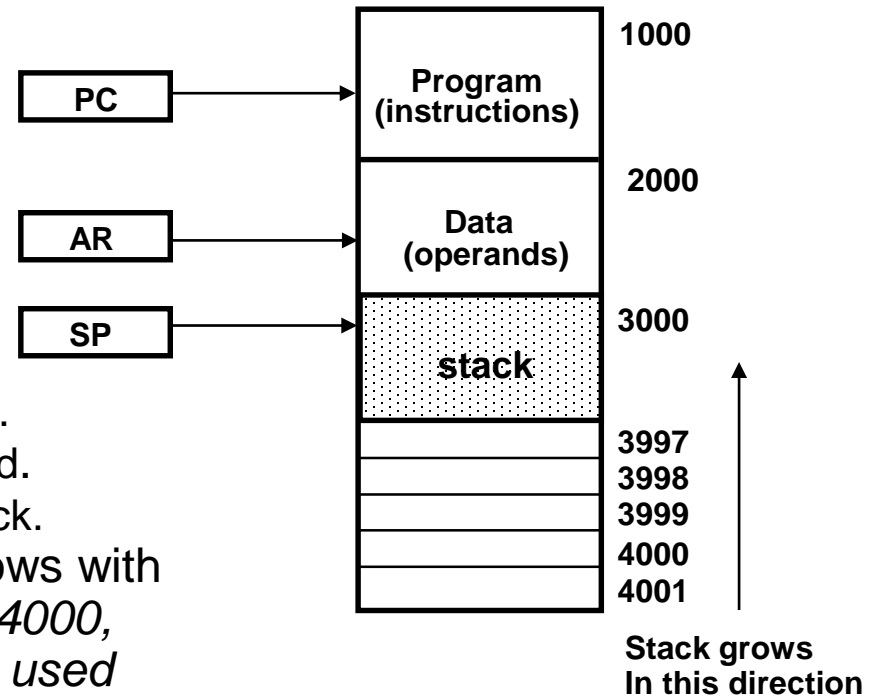
If  $(SP = 0)$  then  $(EMPTY \leftarrow 1)$

$FULL \leftarrow 0$

# MEMORY STACK ORGANIZATION

-A portion of memory is used as a stack with a processor register as a stack pointer

-Figure shows a portion of memory Partitioned into 3 parts: Program, Data, and Stack Segments



**PC:** used during fetch phase to read an instruction.

**AR:** used during execute phase to read an operand.

**SP:** used to push or pop items into or from the stack.

Here, initial value of SP is 4001 and stack grows with *decreasing addresses*. First item is stored at 4000, second at 3999 and last address that can be used is 3000. No provisions are available for stack limit checks.

- **PUSH:**  $SP \leftarrow SP - 1$   
 $M[SP] \leftarrow DR$
- **POP:**  $DR \leftarrow M[SP]$   
 $SP \leftarrow SP + 1$

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) → must be done in software

# PROCESSOR ORGANIZATION

- In general, most processors are organized in one of 3 ways

- Single register (Accumulator) organization

- » Basic Computer is a good example

- » Accumulator is the only general purpose register

Example:     ADD X       // AC  $\leftarrow$  AC + M[X]

              LDA Y       // AC  $\leftarrow$  M[Y]

- General register organization

- » Used by most modern computer processors

- » Any of the registers can be used as the source or destination for computer operations

Example:

              ADD R1, R2, R3       // R1  $\leftarrow$  R2 + R3

              ADD R1, R2       // R1  $\leftarrow$  R1 + R2

              MOV R1, R2       // R1  $\leftarrow$  R2

              ADD R1, X       // R1  $\leftarrow$  R1 + M[X]

- Stack organization

- » All operations are done using the hardware stack

- » For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

Example:     PUSH X     // TOS  $\leftarrow$  M[X]

              ADD       // TOS = TOP(S) + TOP(S)

# INSTRUCTION FORMAT

- **Instruction Fields**

**OP-code field - specifies the operation to be performed**

**Address field - designates memory address(es) or a processor register(s)**

**Mode field** - determines how the address field is to be interpreted (to get effective address or the operand)

- **The number of address fields in the instruction format depends on the internal organization of CPU**

- **The three most common CPU organizations:**

### Single accumulator organization:

**ADD    X                    /\* AC ← AC + M[X] \*/**

## General register organization:

**ADD    R1, R2, R3            */\* R1 ← R2 + R3 \*/***

**ADD    R1, R2            /\* R1  $\leftarrow$  R1 + R2 \*/**

**MOV     R1, R2                    /\* R1 ← R2 \*/**

**ADD    R1, X                    /\* R1 ← R1 + M[X] \*/**

## Stack organization:


**PUSH X**                      **/\* TOS ← M[X] \*/**

## ADD


# THREE-ADDRESS INSTRUCTIONS

- On the basis of no. of address field we can categorize the instruction as below:
- **Three-Address Instructions**
  - Computers with three address instruction formats can use each address field to specify either a processor register or a memory operand.

Program to evaluate  $X = (A + B) * (C + D)$  :



ADD	R1, A, B	/* $R1 \leftarrow M[A] + M[B]$	*/
ADD	R2, C, D	/* $R2 \leftarrow M[C] + M[D]$	*/
MUL	X, R1, R2	/* $M[X] \leftarrow R1 * R2$	*/

- Results in short programs when evaluating arithmetic expressions.
  - Instruction becomes long (many bits)
  - The binary coded instructions require too many bits to specify three addresses.
- 

# TWO-ADDRESS INSTRUCTIONS

- **Two-Address Instructions**

- These instructions are much common in commercial computers.
- Each address field can specify either a processor register or a memory word.

Program to evaluate  $X = (A + B) * (C + D)$  :

MOV	R1, A	/* R1 ← M[A]	*/
ADD	R1, B	/* R1 ← R1 + M[A]	*/
MOV	R2, C	/* R2 ← M[C]	*/
ADD	R2, D	/* R2 ← R2 + M[D]	*/
MUL	R1, R2	/* R1 ← R1 * R2	*/
MOV	X, R1	/* M[X] ← R1	*/

- Tries to minimize the size of instruction
- Size of program is relatively larger.



# ONE ADDRESS INSTRUCTIONS

- One-Address Instructions

- Use an implied AC register for all data manipulation
- Program to evaluate  $X = (A + B) * (C + D)$  :

LOAD	A	/* AC $\leftarrow$ M[A]	*/
ADD	B	/* AC $\leftarrow$ AC + M[B]	*/
STORE	T	/* M[T] $\leftarrow$ AC	*/
LOAD	C	/* AC $\leftarrow$ M[C]	*/
ADD	D	/* AC $\leftarrow$ AC + M[D]	*/
MUL	T	/* AC $\leftarrow$ AC * M[T]	*/
STORE	X	/* M[X] $\leftarrow$ AC	*/

# ZERO-ADDRESS INSTRUCTIONS

- Zero-Address Instructions

- Can be found in a stack-organized computer
- A stack organized computer does not use an address field for instruction ADD and MUL.
- The PUSH and POP instructions however need an address field to specify the operand that communicates with the stack.

- Program to evaluate  $X = (A + B) * (C + D)$  :

PUSH	A	/* TOS $\leftarrow$ A	*/
PUSH	B	/* TOS $\leftarrow$ B	*/
<u>ADD</u>		/* TOS $\leftarrow$ (A + B)	*/
<u>PUSH</u>	C	/* TOS $\leftarrow$ C	*/
<u>PUSH</u>	D	/* TOS $\leftarrow$ D	*/
<u>ADD</u>		/* TOS $\leftarrow$ (C + D)	*/
<u>MUL</u>		/* TOS $\leftarrow$ (C + D) * (A + B)	*/
<u>POP</u>	X	/* M[X] $\leftarrow$ TOS	*/

The name “zero-address” is given to this type of computer because Of the absence of an address field in the computational instructions.

# ADDRESSING MODES

- Addressing Modes

❖ Operation field of an instruction specifies the operation that must be executed on some data stored in computer register or memory words”.

The way operands (data) are chosen during program execution depends on the addressing mode of the instruction

- \* Addressing mode Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

- \* Variety of addressing modes

- to give programming flexibility to the user
- to use the bits in the address field of the instruction efficiently

# TYPES OF ADDRESSING MODES

- **Implied Mode**

Address of the operands are specified implicitly in the definition of the instruction

- No need to specify address in the instruction
- EA = AC, or EA = Stack[SP]
- Examples from Basic Computer  
CLA, CME, INP

- **Immediate Mode**

Instead of specifying the address of the operand, **operand itself is specified**

- No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, **require more bits than the address**
  - **Fast** to acquire an operand
- Example **R4,#3**

ex `mov A, #h6u8 , LDA`

# TYPES OF ADDRESSING MODES

## • Register Mode

- In this mode **operands are in registers** that reside within the CPU.
  - Address specified in the instruction is the **register address**
  - Designated operand need to be in a register
  - **Shorter address** than the memory address
  - Saving address field in the instruction
  - **Faster to acquire an operand than the memory addressing**
  - **EA = IR(R)** (IR(R): Register field of IR)

## • Register Indirect Mode

Instruction **specifies a register** which contains the memory address of the operand

- In other words the selected **register contains address of operand rather than operand itself.**
- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA = [IR(R)] ([x]: Content of x)

# TYPES OF ADDRESSING MODES

- **Autoincrement or Autodecrement Mode**
  - When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically.
  - It is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

# TYPES OF ADDRESSING MODES

- **Direct Address Mode**

Instruction specifies the memory address which can be used directly to access the memory

- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory space
- $EA = IR(addr)$  ( $IR(addr)$ : address field of IR)

- **Indirect Addressing Mode**

The address field of an instruction specifies the address of a memory location that contains the address of the operand

- When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
- Slow to acquire an operand because of an additional memory access
- $EA = M[IR(address)]$

# TYPES OF ADDRESSING MODES

- **Relative Addressing Modes**

The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

- Address field of the instruction is short
- Large physical memory can be accessed with a small number of address bits
- $EA = f(IR(address), R)$ , R is sometimes implied

3 different Relative Addressing Modes depending on R;

- \* **PC Relative Addressing Mode** (R = PC)

- $EA = PC + IR(address)$

- \* **Indexed Addressing Mode** (R = IX, where IX: Index Register)

- $EA = IX + IR(address)$

- \* **Base Register Addressing Mode**

(R = BAR, where BAR: Base Address Register)

- $EA = BAR + IR(address)$



# ADDRESSING MODES - EXAMPLES -

PC = 200

R1 = 400

XR = 100

AC

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

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

# **Data Transfer and Manipulation**

- **Computers provide extensive set of instructions to give the user the flexibility to carryout various computational tasks.**
- **The actual operations in the instruction set are not very different from one computer to another although binary encodings and symbol name (operation) may vary.**
- **So, most computer instructions can be classified into 3 categories:**
  - 1. Data transfer instructions**
  - 2. Data manipulation instructions**
  - 3. Program control instructions**

# DATA TRANSFER INSTRUCTIONS

- Data transfer instructions causes transfer of data from one location to another without changing the binary information content.
- The most common transfers are:
  - ☐ between memory and processor registers
  - ☐ between processor registers and I/O
  - ☐ between processor register themselves
- Table below lists 8 data transfer instructions used in many computers.

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

**Typical Data Transfer Instructions**

# DATA TRANSFER INSTRUCTIONS

- Instructions described above are often associated with the variety of addressing modes.
- Assembly language uses special character to designate the addressing mode. E.g. # sign placed before the operand to recognize the immediate mode. (Some other assembly languages modify the mnemonics symbol to denote various addressing modes, e.g. for load immediate: LDI).
- Example: consider *load to accumulator instruction when used with 8 different addressing modes*:

- Data Transfer Instructions with 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)	$R1 \leftarrow R1 - 1, AC \leftarrow M[R1]$

# **DATA MANIPULATION INSTRUCTIONS**

- **Data manipulation instructions perform operations on data and provide computational capabilities for the computer.**
- **Three Basic Types:**
  - Arithmetic instructions**
  - Logical and bit manipulation instructions**
  - Shift instructions**

# DATA MANIPULATION INSTRUCTIONS

- **Arithmetic Instructions**

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

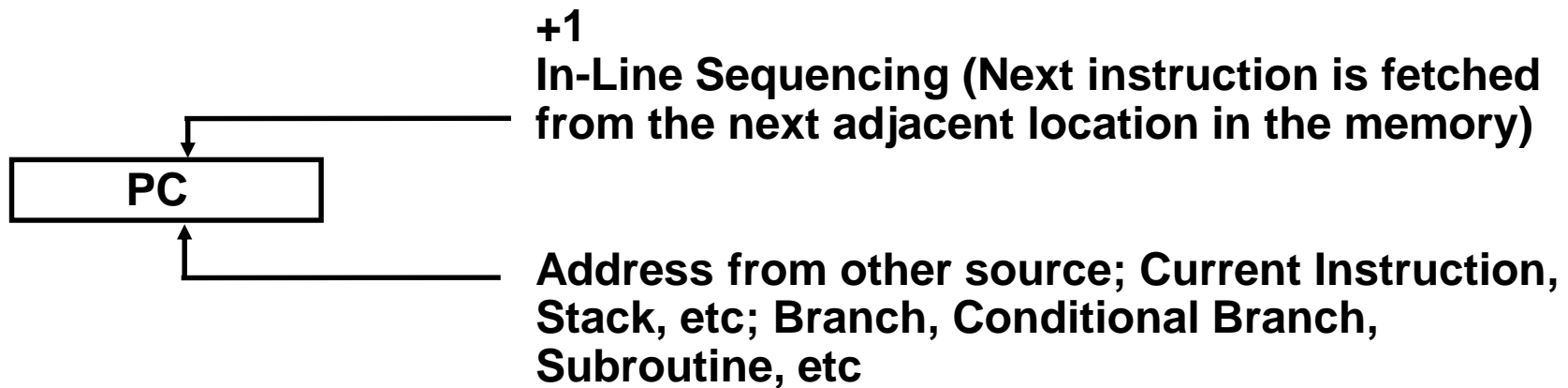
- **Logical and Bit Manipulation Instructions**

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

- **Shift Instructions**

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

# PROGRAM CONTROL INSTRUCTIONS



## • Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by – )	CMP
Test(by AND)	TST

\* CMP and TST instructions do not retain their results of operations ( – and AND, respectively). They only set or clear certain Flags.

# PROGRAM INTERRUPT

## Types of Interrupts

### External interrupts

External Interrupts initiated from the outside of CPU and Memory

- I/O Device → Data transfer request or Data transfer complete
- Timing Device → Timeout
- Power Failure
- Operator

### Internal interrupts (traps)

Internal Interrupts are caused by the currently running program

- Register, Stack Overflow
- Divide by zero
- OP-code Violation
- Protection Violation

### Software Interrupts

Both External and Internal Interrupts are initiated by the computer HW.

Software Interrupts are initiated by the executing an instruction.

- Supervisor Call → Switching from a user mode to the supervisor mode  
→ Allows to execute a certain class of operations  
which are not allowed in the user mode



# INTERRUPT PROCEDURE

## Interrupt Procedure and Subroutine Call

- The interrupt is usually initiated by an internal or an external signal rather than from the execution of an instruction (except for the software interrupt)
- The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction
- An interrupt procedure usually stores all the information necessary to define the state of CPU rather than storing only the PC.

The state of the CPU is determined from;

Content of the PC

Content of all processor registers

Content of status bits

Many ways of saving the CPU state

depending on the CPU architectures

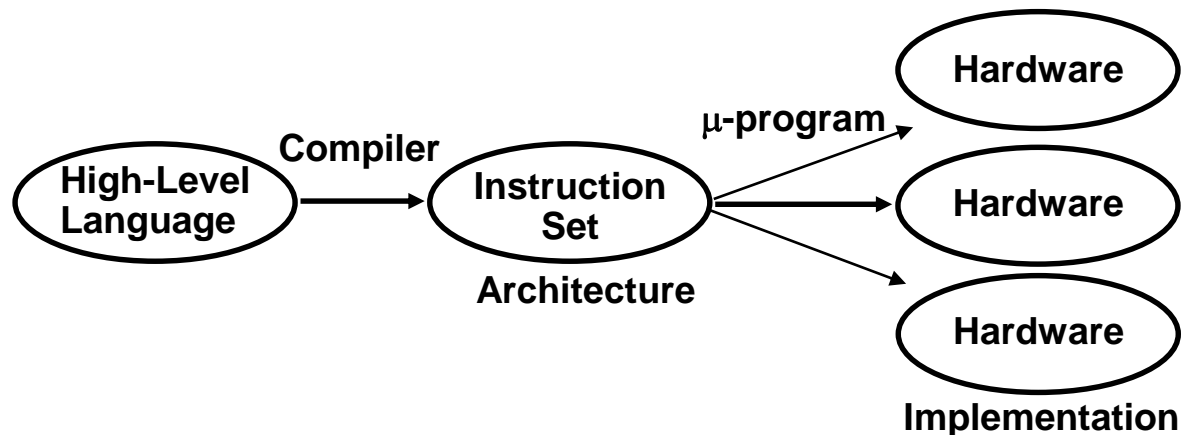
# RISC: Historical Background

all vvi risc n cisc

RISC  
C

## IBM System/360, 1964

- The real beginning of modern computer architecture
- Distinction between *Architecture* and *Implementation*
- Architecture: The abstract structure of a computer seen by an assembly-language programmer



- Continuing growth in semiconductor memory and microprogramming
  - ⇒ A much richer and complicated instruction sets
  - ⇒ CISC(Complex Instruction Set Computer)

# ARGUMENTS ADVANCED AT THAT TIME

---


- Richer instruction sets would simplify compilers
- Richer instruction sets would alleviate the software crisis
  - move as much functions to the hardware as possible
- Richer instruction sets would improve *architecture quality*

# ARCHITECTURE DESIGN PRINCIPLES - IN 70's -

- **Large microprograms would add little or nothing to the cost of the machine**
  - ← Rapid growth of memory technology
  - ⇒ Large General Purpose Instruction Set
- **Microprogram is much faster than the machine instructions**
  - ← Microprogram memory is much faster than main memory
  - ⇒ Moving the software functions into microprogram for the high performance machines
- **Execution speed is proportional to the program size**
  - ← Architectural techniques that led to small program
  - ⇒ High performance instruction set
- **Number of registers in CPU has limitations**
  - ← Very costly
  - ⇒ Difficult to utilize them efficiently



# COMPLEX INSTRUCTION SET COMPUTER

- These computers with **many instructions** and addressing modes came to be known as Complex Instruction Set Computers (CISC)
  - One goal for CISC machines was to have a machine language instruction to match each high-level language statement type
- 

# CISC: VARIABLE LENGTH INSTRUCTIONS

- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
- The **large number of instructions** means a greater number of bits to specify them
- In order to manage this large number of opcodes efficiently, they were encoded with different lengths:
  - **More frequently** used instructions were encoded using **short opcodes**.
  - **Less frequently** used ones were assigned **longer opcodes**.
- Also, multiple operand instructions could specify **different addressing modes** for each operand
  - For example,
    - » Operand 1 could be a directly addressed register,
    - » Operand 2 could be an indirectly addressed memory location,
    - » Operand 3 (the destination) could be an indirectly addressed register.
- All of this led to the need to have different length instructions in different situations, depending on the opcode and operands used

# VARIABLE LENGTH INSTRUCTIONS

- For example, an instruction that only specifies register operands may only be two bytes in length
  - One byte to specify the instruction and addressing mode
  - One byte to specify the source and destination registers.
- An instruction that specifies memory addresses for operands may need five bytes
  - One byte to specify the instruction and addressing mode
  - Two bytes to specify each memory address
    - » Maybe more if there's a large amount of memory.
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex

# COMPLEX INSTRUCTION SET COMPUTER

- Another characteristic of CISC computers is that they have instructions that act directly on memory addresses
  - For example, ADD L1, L2, L3  
that takes the contents of  $M[L1]$  adds it to the contents of  $M[L2]$  and stores the result in location  $M[L3]$
- An instruction like this takes three memory access cycles to execute
- That makes for a potentially very long instruction execution cycle
- The problems with CISC computers are
  - The complexity of the design may slow down the processor,
  - The complexity of the design may result in costly errors in the processor design and implementation,
  - Many of the instructions and addressing modes are used rarely, if ever



# SUMMARY: CRITICISMS ON CISC

## High Performance General Purpose Instructions

- **Complex Instruction**

- Format, Length, Addressing Modes
- Complicated instruction cycle control due to the complex decoding HW and decoding process

- **Multiple memory cycle instructions**

- Operations on memory data
- Multiple memory accesses/instruction

- **Microprogrammed control is necessity**

- Microprogram control storage takes substantial portion of CPU chip area
- Semantic Gap is large between machine instruction and microinstruction

- **General purpose instruction set includes all the features required by individually different applications**

- When any one application is running, all the features required by the other applications are extra burden to the application

# REDUCED INSTRUCTION SET COMPUTERS

- In the late '70s and early '80s there was a reaction to the shortcomings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as an alternative
- The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time
- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

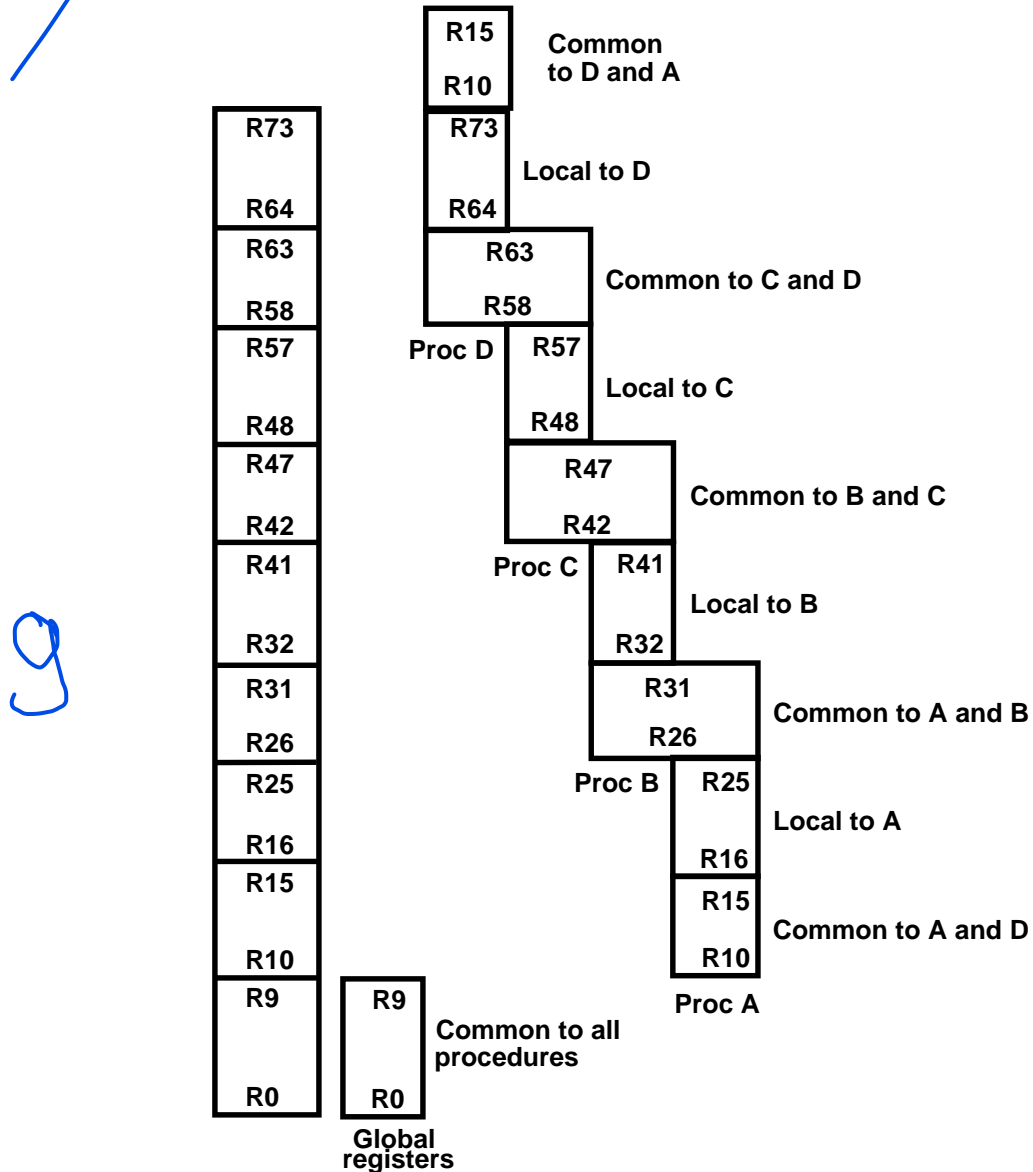
# REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed
- By having all instructions the same length, reading them in is easy and fast
- The fetch and decode stages are simple, looking much more like Mano's Basic Computer than a CISC machine
- The instruction and address formats are designed to be easy to decode
- Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously
- The control logic of a RISC processor is designed to be simple and fast
- The control logic is simple because of the small number of instructions and the simple addressing modes
- The control logic is hardwired, rather than microprogrammed, because hardwired control is faster

# REDUCED INSTRUCTION SET COMPUTERS

- By simplifying the instructions and addressing modes, there is space available on the chip or board of a RISC CPU for more circuits than with a CISC processor
- Use of overlapped-register windows to speed procedure call and return
- This extra capacity is used to
  - Pipeline instruction execution to speed up instruction execution
  - Add a large number of registers to the CPU

# OVERLAPPED REGISTER WINDOWS



# OVERLAPPED REGISTER WINDOWS

- There are three classes of registers:
  - Global Registers
    - » Available to all functions
  - Window local registers
    - » Variables local to the function
  - Window shared registers
    - » Permit data to be shared without actually needing to copy it
- Only one register window is active at a time
  - The active register window is indicated by a pointer
- When a function is called, a new register window is activated
  - This is done by incrementing the pointer
- When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window
- This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

# OVERLAPPED REGISTER WINDOWS

- System has a total of 74 registers (Just an example)
    - R0 – R9 = global registers (hold parameters shared by all procedures)
    - Other 64 registers are divided into 4 windows to accommodate procedures A, B, C and D.
    - Each register window consists of 10 local registers and two sets of 6 registers common to adjacent windows.
    - Common overlapped registers permit parameters to be passed without the actual movement of data
    - Only one register window is activated at any time with a pointer indicating the active window.
    - Four windows have a circular organization with A being adjacent to D
-

# OVERLAPPED REGISTER WINDOWS

- In general, the organization of register windows will have following relationships:
- Number of global registers =  $G$
- Number of local register in each window =  $L$
- Number of registers common to windows =  $C$
- Number of windows =  $W$
- Now,
- Window size =  $L + 2C + G$
- Register file =  $(L+C)W + G$  (total number of register needed in the processor)
- Example: In above fig,  $G = 10$ ,  $L = 10$ ,  $C = 6$  and  $W = 4$ . Thus window size =  $10+12+10 = 32$  registers and register file consists of  $(10+6)*4+10 = 74$  registers.



# CHARACTERISTICS OF RISC

- **RISC Characteristics**

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format
- Single-cycle instruction format
- Hardwired rather than microprogrammed control

- **Advantages of RISC**

- VLSI Realization
- Computing Speed
- Design Costs and Reliability
- High Level Language Support

# ADVANTAGES OF RISC

- VLSI Realization

Control area is considerably reduced

Example:

RISC I: 6%

RISC II: 10%

MC68020: 68%

general CISCs: ~50%

⇒ RISC chips allow a large number of registers on the chip

- Enhancement of performance and HLL support
- Higher regularization factor and lower VLSI design cost

The GaAs VLSI chip realization is possible

- Computing Speed

- Simpler, smaller control unit ⇒ faster
- Simpler instruction set; addressing modes; instruction format  
⇒ faster decoding
- Register operation ⇒ faster than memory operation
- Register window ⇒ enhances the overall speed of execution
- Identical instruction length, One cycle instruction execution  
⇒ suitable for pipelining ⇒ faster

# ADVANTAGES OF RISC

## • Design Costs and Reliability

- Shorter time to design
  - ⇒ reduction in the overall design cost and reduces the problem that the end product will be obsolete by the time the design is completed
- Simpler, smaller control unit
  - ⇒ higher reliability
- Simple instruction format (of fixed length)
  - ⇒ ease of virtual memory management

## • High Level Language Support

- A single choice of instruction
  - ⇒ shorter, simpler compiler
- A large number of CPU registers
  - ⇒ more efficient code
- Register window
  - ⇒ Direct support of HLL
- Reduced burden on compiler writer