Module 3

3	**Title**	**Processor Organization and Control Unit**		**9**
	3.1	CPU Architecture, Register Organization Instruction formats, basic instruction cycle. Instruction interpretation and sequencing, Case Study of 8086 architecture and Register Organization.	1,2,4	
	3.2	Control Unit: Soft wired (Micro-programmed) and hardwired control unit design methods. Microinstruction sequencing and execution. Micro operations	2,4	
	3.3	RISC and CISC: Introduction to RISC and CISC architectures and design issues.	2,4	

Processor Structure and Function

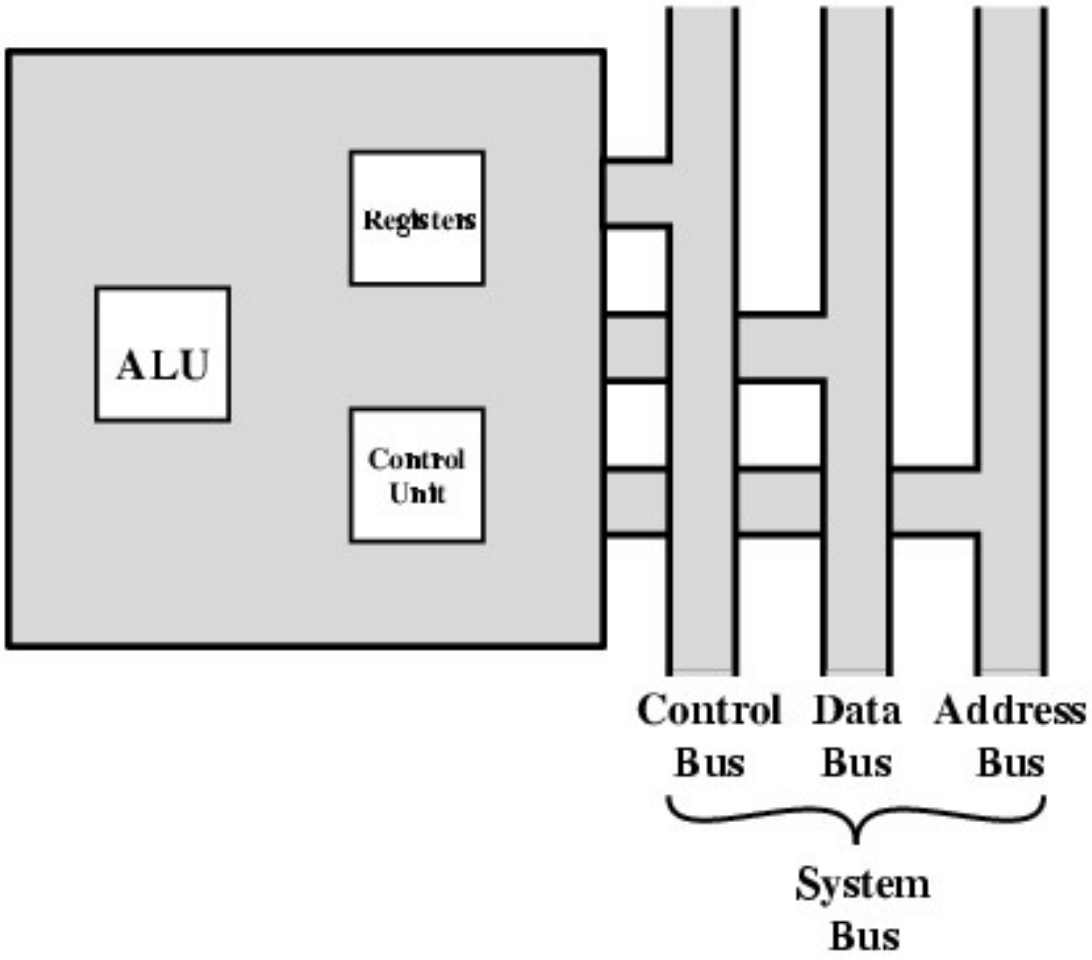
CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data

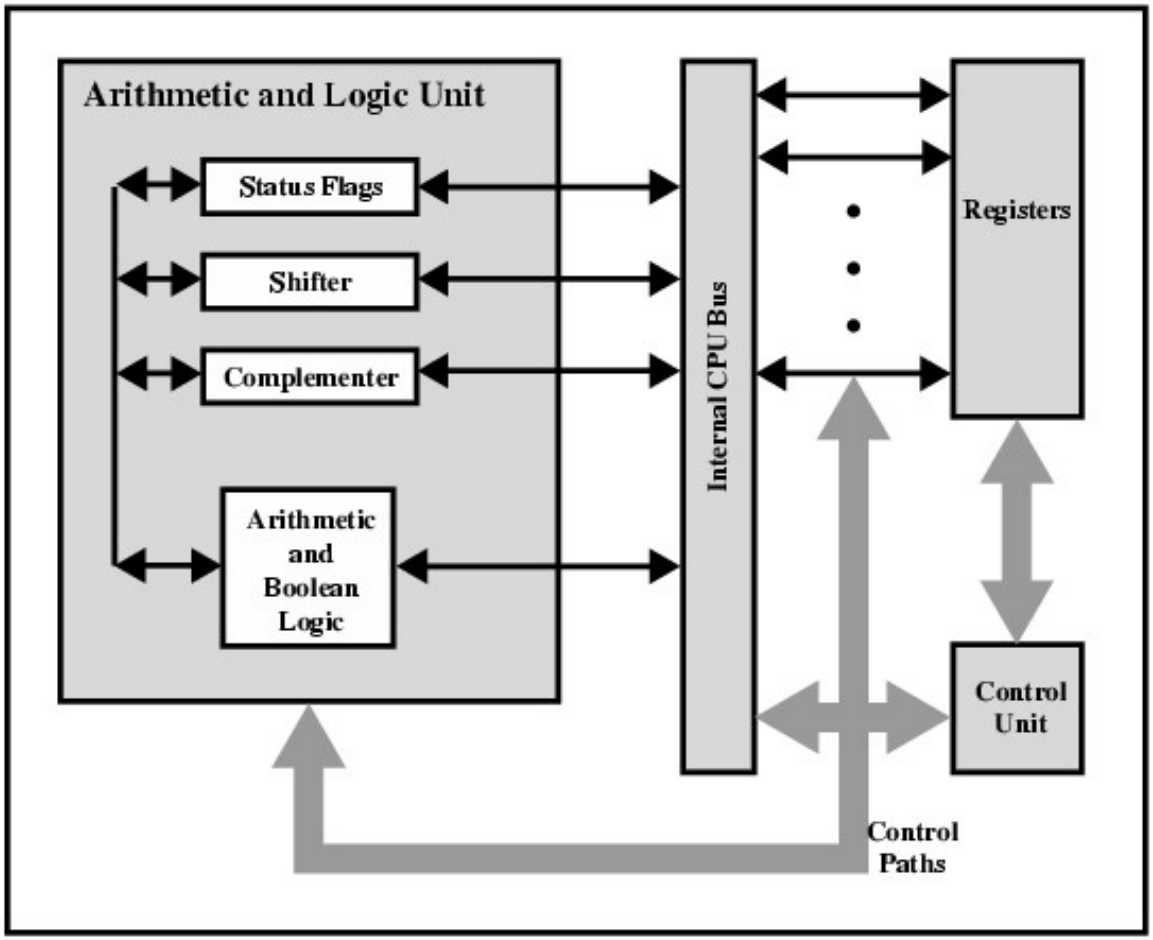
Remember….

- The processor needs to store some data temporally.
- It must remember the location of the last instruction so that it can know where to get the next instruction.
- It needs to store instructions and data temporally while an instruction is being executed.

CPU With System Bus



CPU Internal Structure



Registers

- CPU must have some working space (temporary storage)
- Called registers
- Number and function vary between processor designs
- One of the major design decisions
- Top level of memory hierarchy

How big?

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

How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More does not reduce memory references and takes up processor real estate

Register Organization

- User-visible registers: These enables the machine - or assembly-language programmer to minimize main memory reference by optimizing use of registers.

- Control and status registers: These are used by the control unit to control the operation of the CPU. Operating system programs may also use these in privileged mode to control the execution of program.

User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

- General Purpose Registers
  - May be true general purpose
  - can be assigned to a variety of functions by the programmer
  - May be used for data or addressing
- Data
  - used to hold only data and cannot be employed in the calculation of an operand address.
- Addressing
  - Segment pointers
  - Index registers
  - Stack Pointer

Condition Code Registers

- Sets of individual bits
  - e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - e.g. Jump if zero
- Can not (usually) be set by programs

Program Status Word

- A set of bits Includes Condition Codes
- Sign of last result
- Zero
- Carry
- Equal
- Overflow
- Interrupt enable/disable
- Supervisor

Supervisor Mode

- Kernel mode
- Allows privileged instructions to execute
- Used by operating system
- Not available to user programs

Control & Status Registers

- Program Counter
- Instruction Decoding Register
- Memory Address Register
- Memory Buffer Register

- Revision: what do these all do?

Block diagram of 8086

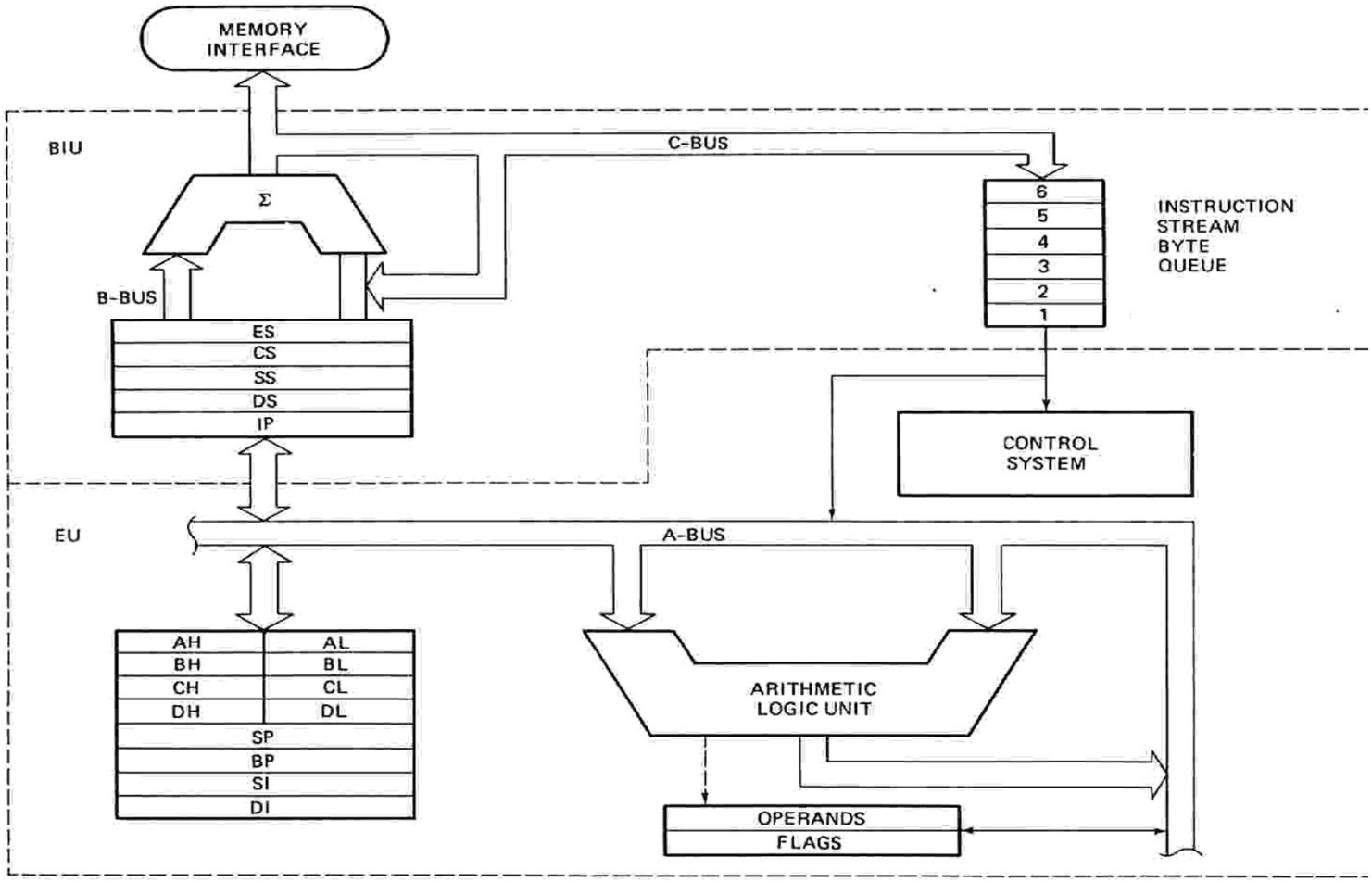
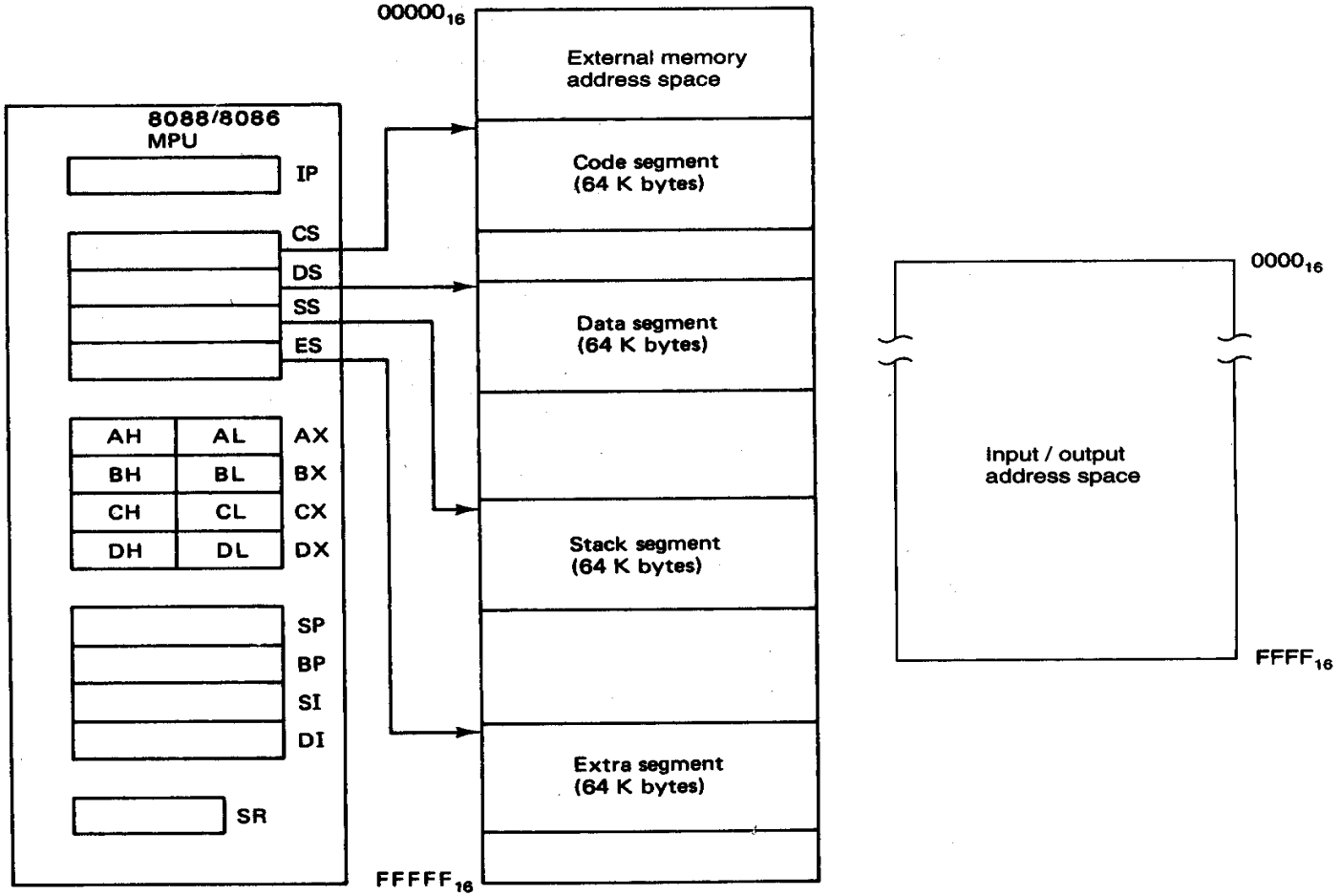
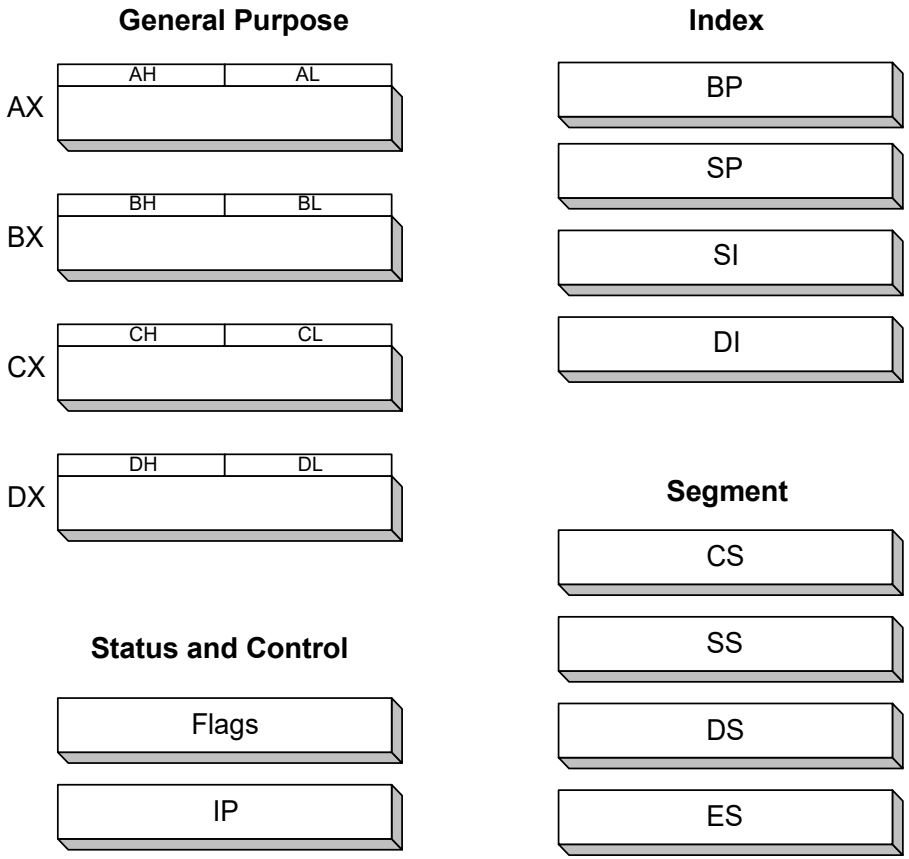


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

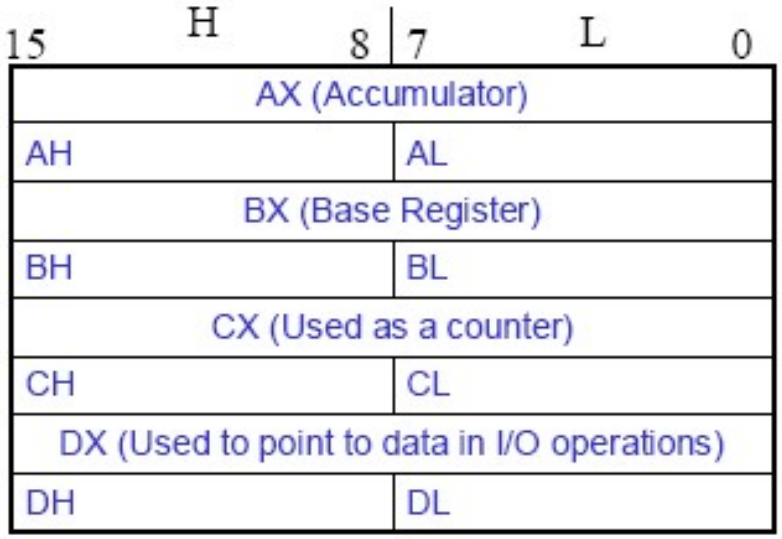
Software Model of the 8086 Microprocessors



8086 Registers



General Purpose Registers



AX - the Accumulator
BX - the Base Register
CX - the Count Register
DX - the Data Register

- Normally used for storing temporary results
- Each of the registers is 16 bits wide **(AX, BX, CX, DX)**
- Can be accessed as either 16 or 8 bits AX, AH, AL

General Purpose Registers

- **AX**
  - Accumulator Register
  - Preferred register to use in arithmetic, logic and data transfer instructions because it generates the shortest Machine Language Code
  - Must be used in multiplication and division operations
  - Must also be used in I/O operations

- **BX**
  - Base Register
  - Also serves as an address register

General Purpose Registers

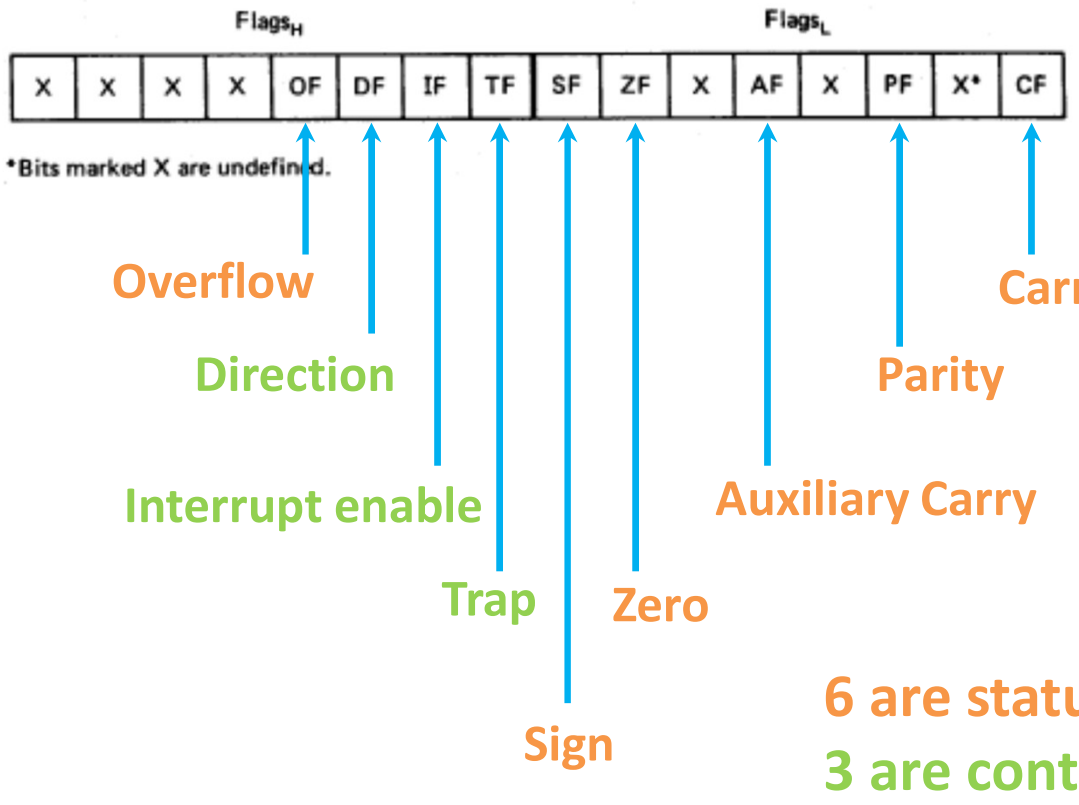
- **CX**
  - Count register
  - Used as a loop counter
  - Used in shift and rotate operations
- **DX**
  - Data register
  - Used in multiplication and division
  - Also used in I/O operations

Pointer and Index Registers

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index
IP	Instruction Pointer

- All 16 bits wide, L/H bytes are not accessible
- Used as memory pointers
  - Example: MOV AH, [SI]
    - *Move the byte stored in memory location whose address is contained in register SI to register AH*
- IP is not under direct control of the programmer

Flag Register



8086 Programmer's Model

BIU registers
(20 bit adder)

ES		Extra Segment
CS		Code Segment
SS		Stack Segment
DS		Data Segment
IP		Instruction Pointer

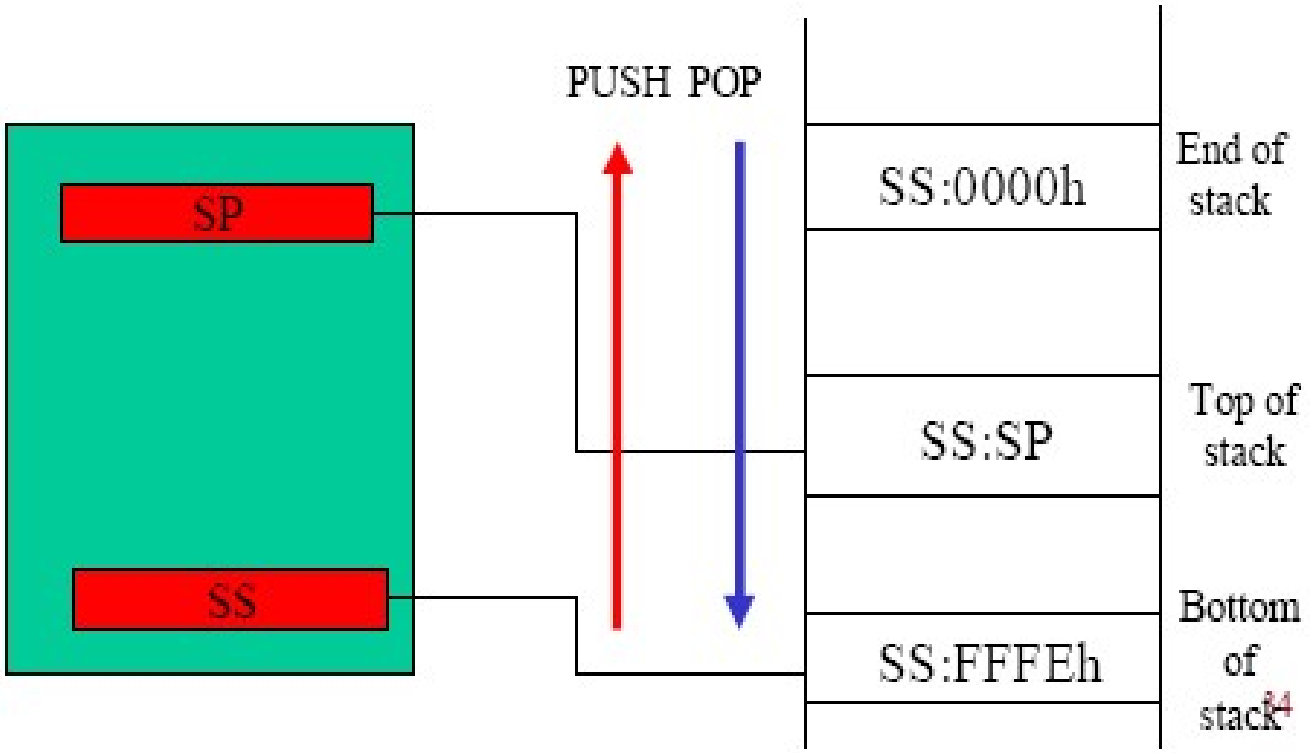
EU registers

AX	AH	AL	Accumulator
BX	BH	BL	Base Register
CX	CH	CL	Count Register
DX	DH	DL	Data Register
	SP		Stack Pointer
	BP		Base Pointer
	SI		Source Index Register
	DI		Destination Index Register
	FLAGS		

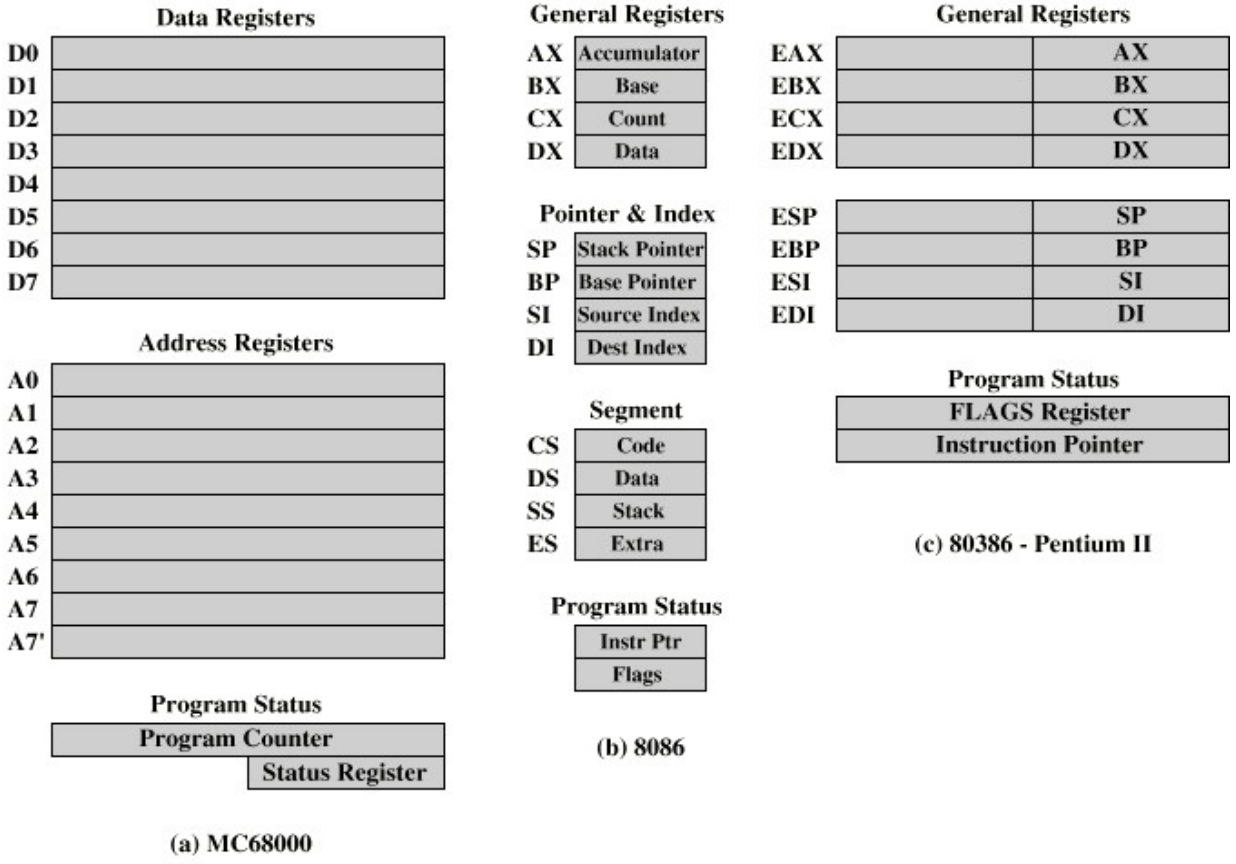
The Stack

- The stack is used for temporary storage of information such as data or addresses.
- When a **CALL** is executed, the 8086 automatically **PUSH**es the current value of CS and IP onto the stack.
- Other registers can also be pushed
- Before return from the **subroutine**, **POP** instructions can be used to pop values back from the stack into the corresponding registers.

The Stack


PUSH POP
SP
SS
SS:0000h
End of stack
SS:SP
Top of stack
SS:FFFEh
Bottom of stack

Example Register Organizations



Features of an ISA

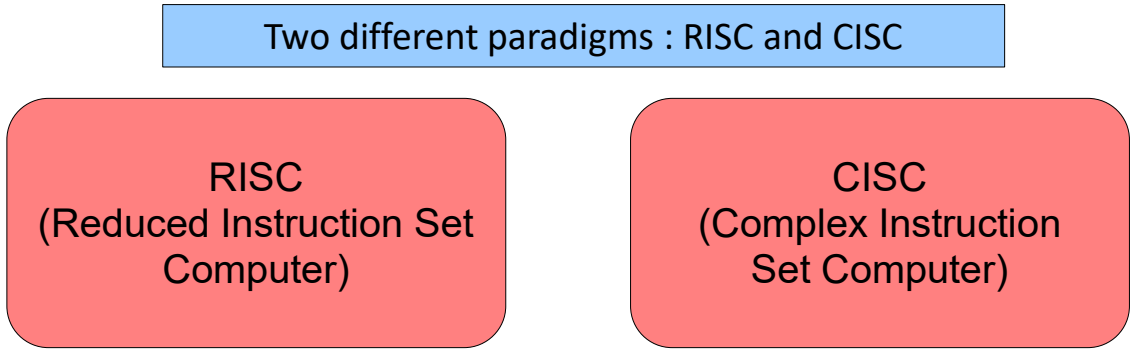
* Example of instructions in an ISA
  * Arithmetic instructions : add, sub, mul, div
  * Logical instructions : and, or, not
  * Data transfer/movement instructions
* Complete
  * It should be able to implement all the programs that users may write.

Features of an ISA – II

* Concise
  * The instruction set should have a limited size. Typically an ISA contains 32-1000 instructions.
* Generic
  * Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized
* Simple
  * Should not be very complicated.

Designing an ISA

* Important questions that need to be answered :
  * How many instructions should we have ?
  * What should they do ?
  * How complicated should they be ?

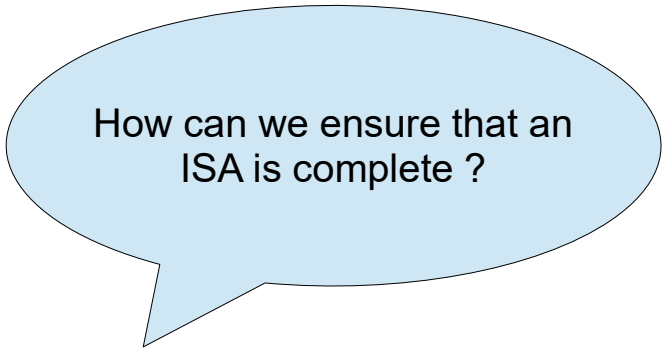


RISC vs CISC

A reduced instruction set computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number (64 to 128). Examples: ARM, IBM PowerPC, HP PA-RISC

A complex instruction set computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large (typically 500+). Examples: Intel x86, VAX

Completeness of an ISA



* Complete means :
  * Can implement all types of programs
  * For example, if we just have add instructions, we cannot subtract (NOT Complete)

Completeness of an ISA – II

How to ensure that we have just enough instructions such that we can implement every possible program that we might want to write ?

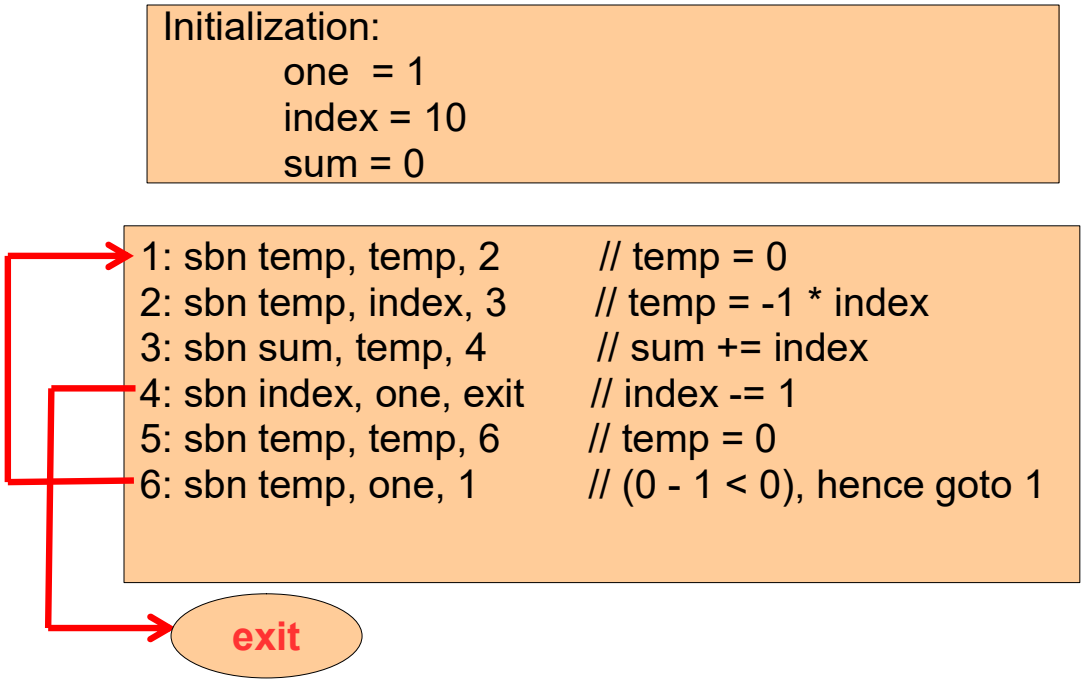
Let us now design an ISA ...

* Single Instruction ISA
  * sbn – subtract and branch if negative
* Add (a + b) (assume temp = 0)

```
1: sbn temp, b, 2
2: sbn a, temp, exit
```

Single Instruction ISA - II

* Add the numbers – 1 … 10



Multiple Instruction ISA

* Arithmetic Instructions
  * add, subtract, multiply, divide
* Logical Instructions
  * or, and, not
* Move instructions
  * Transfer values between memory locations
* Branch instructions
  * Move to a new program location, based on the values of some memory locations

CISC

- It consists of a large set of instructions with variable formats (Typically 16 to 64 bits per instruction)
- It has higher number of addressing modes, typically 12 to 24.
- It consists of complex instructions that take multiple cycles to execute
- Instructions are not pipelined or less pipelined
- Complexity lies in microprogram

RISC

- It consists of small set of instructions with fixed format and these instructions are of register based instructions.
- It has a limited number of addressing modes, typically 3 to 5.
- It consists of simple instructions that take single cycle to execute.
- Instructions are pipelined
- Complexity lies the compiler

Instruction Cycle

- **Fetch**

  Read next instruction from memory into the processor

- **Indirect Cycle (Decode Cycle)**

  May require memory access to fetch operands, therefore more memory accesses.

- **Interrupt**

  Save current instruction and service the interrupt

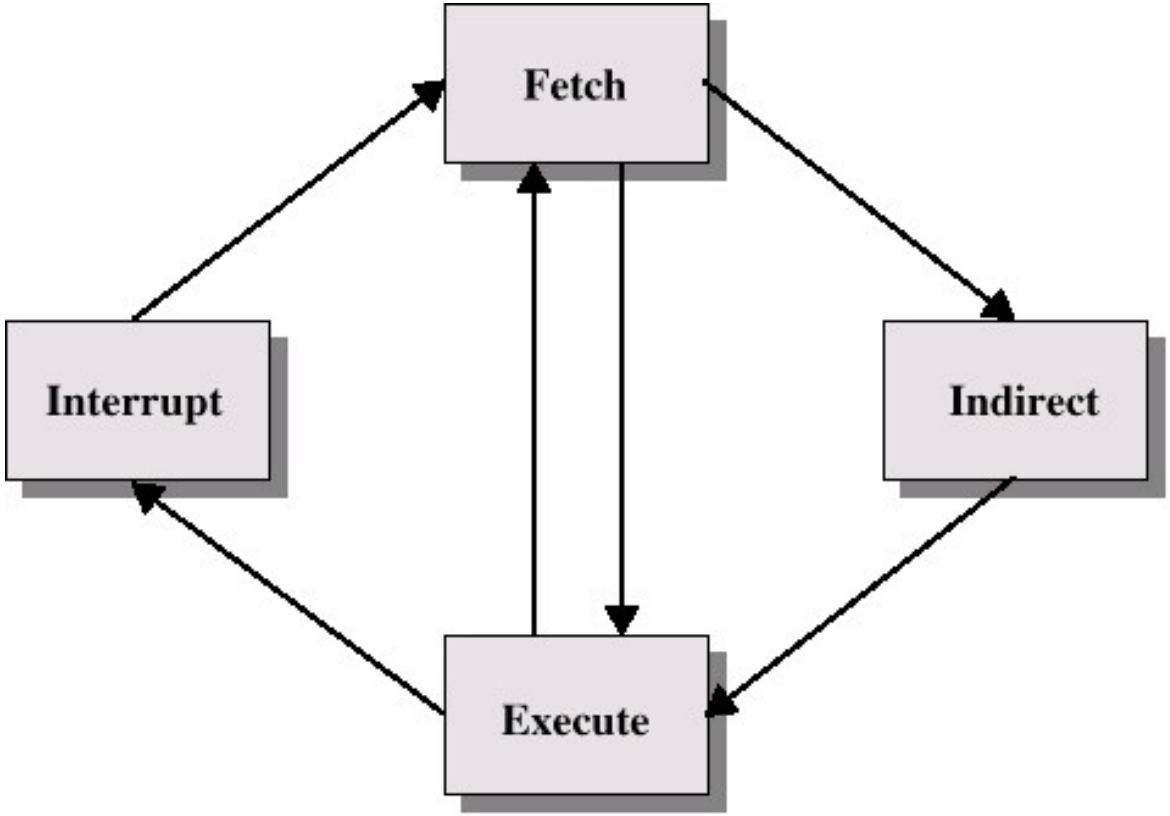
- **Execute**

  Interpret the opcode and perform the indicated operation

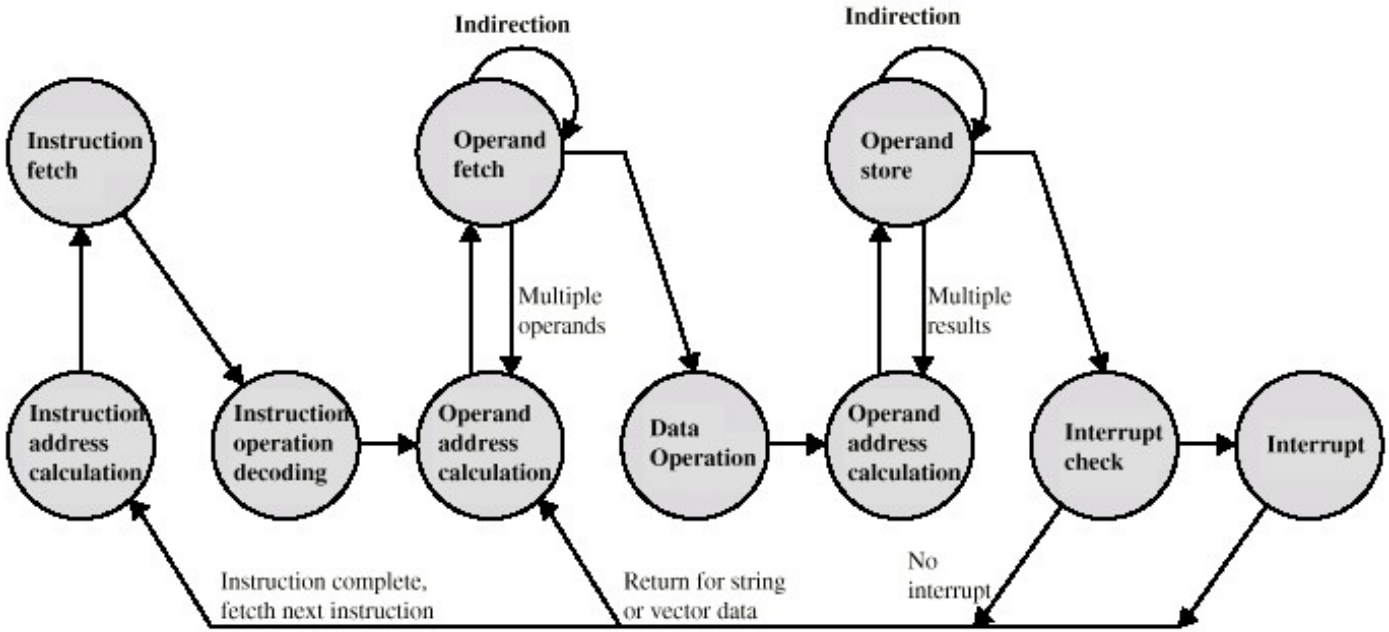
Indirect Cycle

- May require memory access to fetch operands
- Indirect addressing requires more memory accesses
- Can be thought of as additional instruction subcycle

Instruction Cycle with Indirect



Instruction Cycle State Diagram



Data Flow (Instruction Fetch)

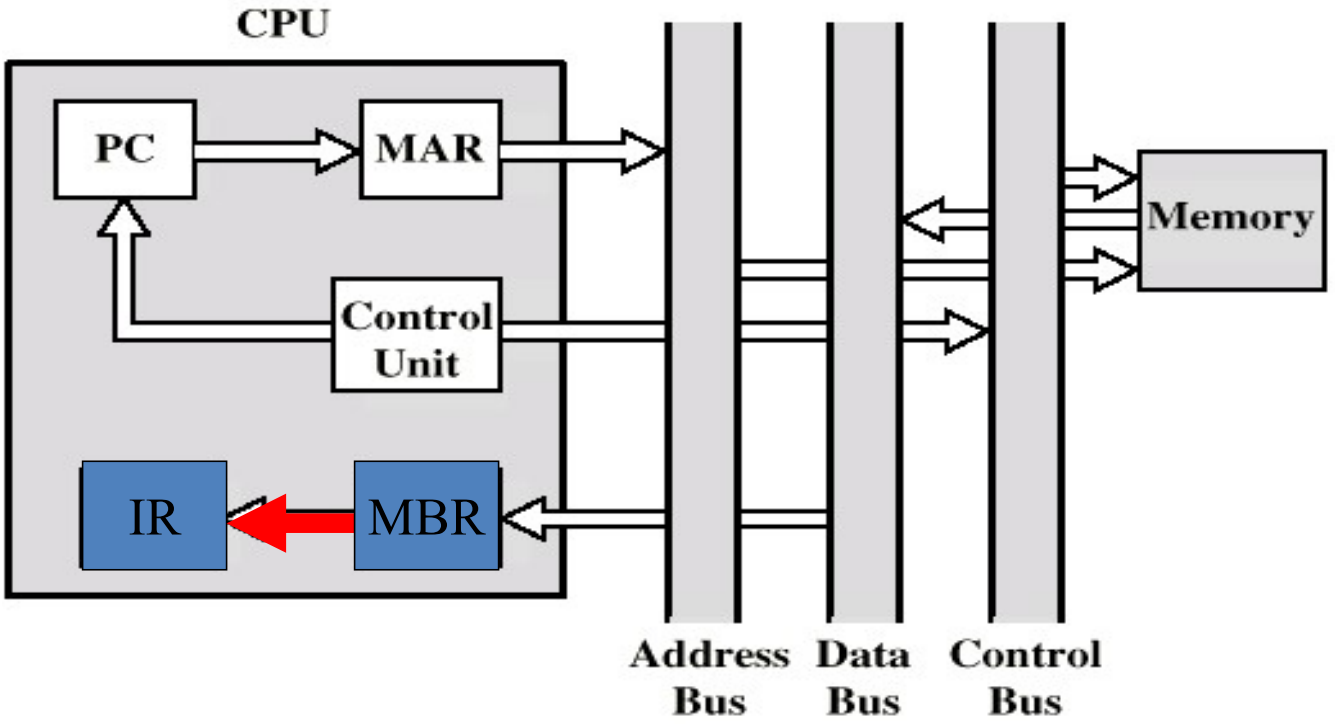
- Depends on CPU design
- In general:

- Fetch
  - PC contains address of next instruction
  - Address moved to MAR
  - Address placed on address bus
  - Control unit requests memory read
  - Result placed on data bus, copied to MBR, then to IR
  - Meanwhile PC incremented by 1

Data Flow (Data Fetch)

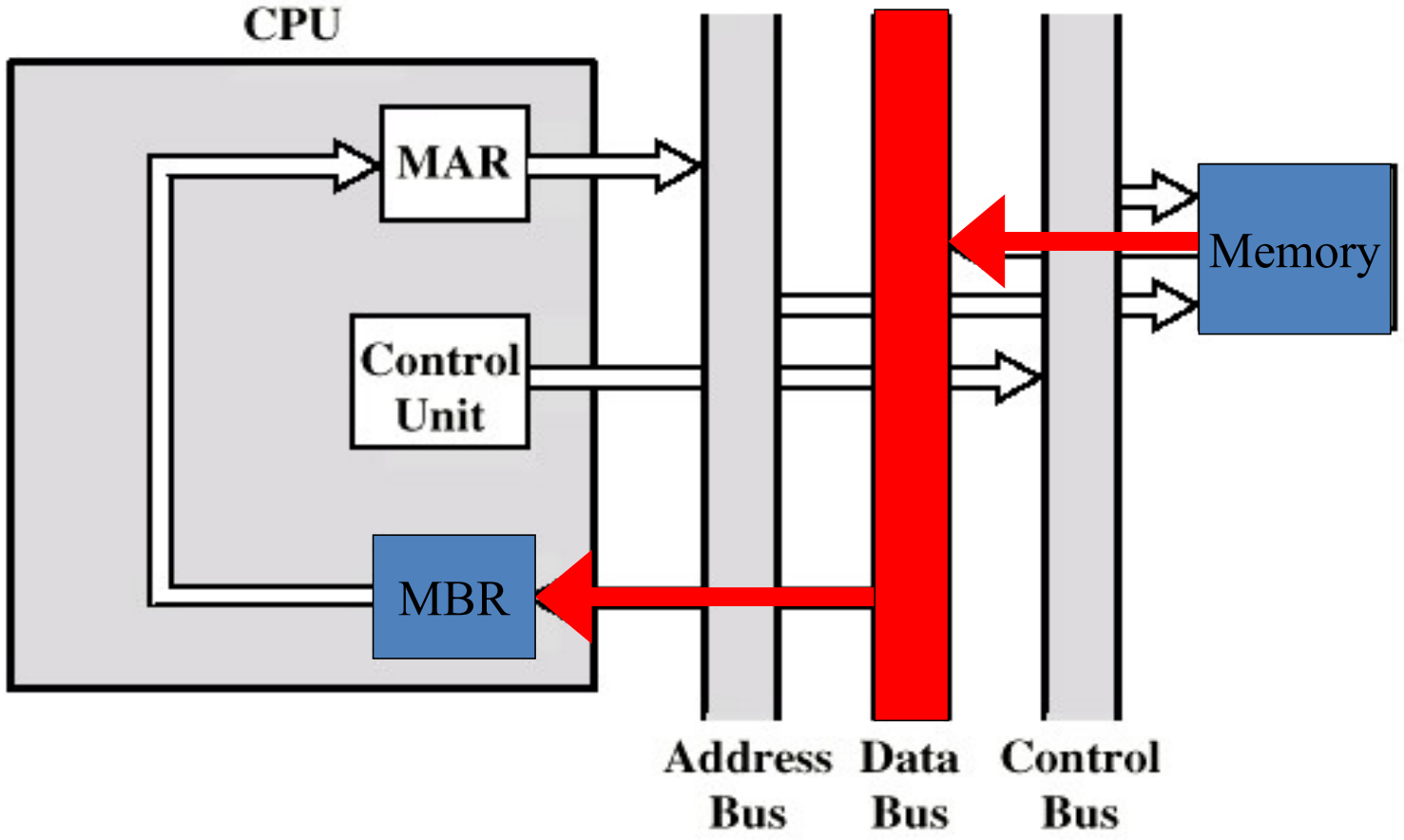
- IR is examined
- If indirect addressing, indirect cycle is performed
  - Right most N bits of MBR transferred to MAR
  - Control unit requests memory read
  - Result (address of operand) moved to MBR

Data Flow (Fetch Diagram)



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Data Flow (Indirect Diagram)



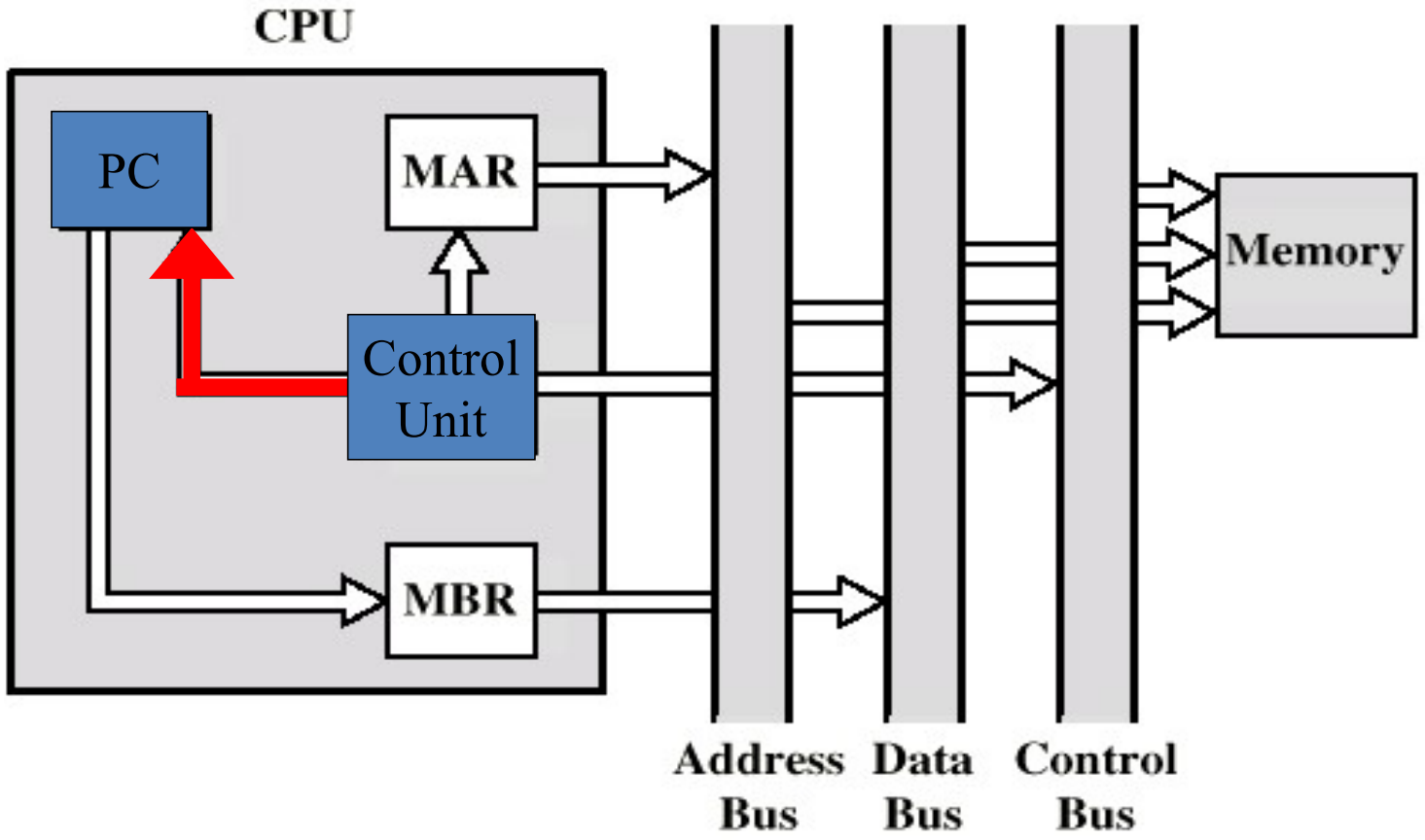
Data Flow (Execute)

- May take many forms
- Depends on instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

Data Flow (Interrupt)

- Simple
- Predictable
- Current PC saved to allow resumption after interrupt
- Contents of PC copied to MBR
- Special memory location (e.g. stack pointer) loaded to MAR
- MBR written to memory
- PC loaded with address of interrupt handling routine
- Next instruction (first of interrupt handler) can be fetched

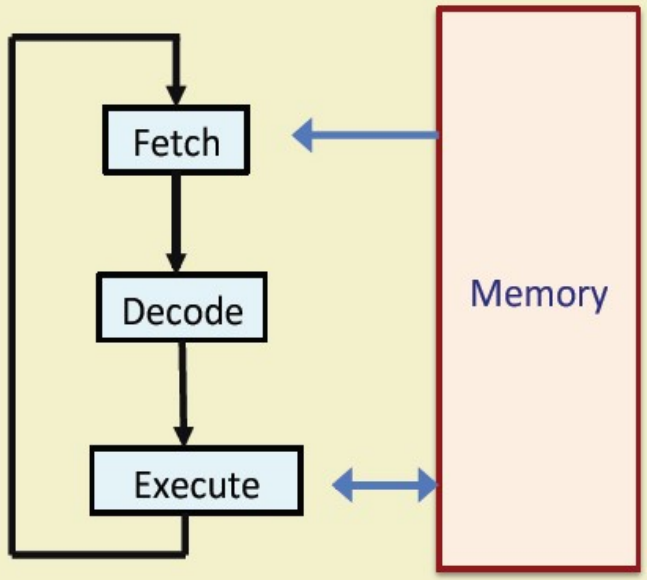
Data Flow (Interrupt Diagram)



Control Unit Design

How an instruction Gets Executed?

```
repeat forever
       // till power off or
       // system failure
{
    Fetch instruction
    Decode instruction
    Execute instruction
}
```



The Fetch-Execute Cycle

- Fetch the next instruction from memory.
- Decode the instruction.
- Execution Cycle:
  - Gets data from memory if needed (data not available in the processor)
  - Perform the required operation on the data.
  - May also store the result back in memory or register.

Registers: IR and PC

- Program Counter (PC) holds the address of the memory location containing the next instruction to be executed.
- Instruction Register (IR) contains the current instruction being executed.
- Basic processing cycle to be implemented:
  - Instruction Fetch (IF)

    $IR \leftarrow Mem\,[PC]$
  - Considering the word length of the machine is 32 bit, the PC is incremented by 4 to point to the next instruction.

    $PC \leftarrow PC + 4$
  - Carry out the operations specified in IR.

Example: Add R1, R2

Address	Instruction
1000	ADD R1, R2
1004	MUL R3, R4

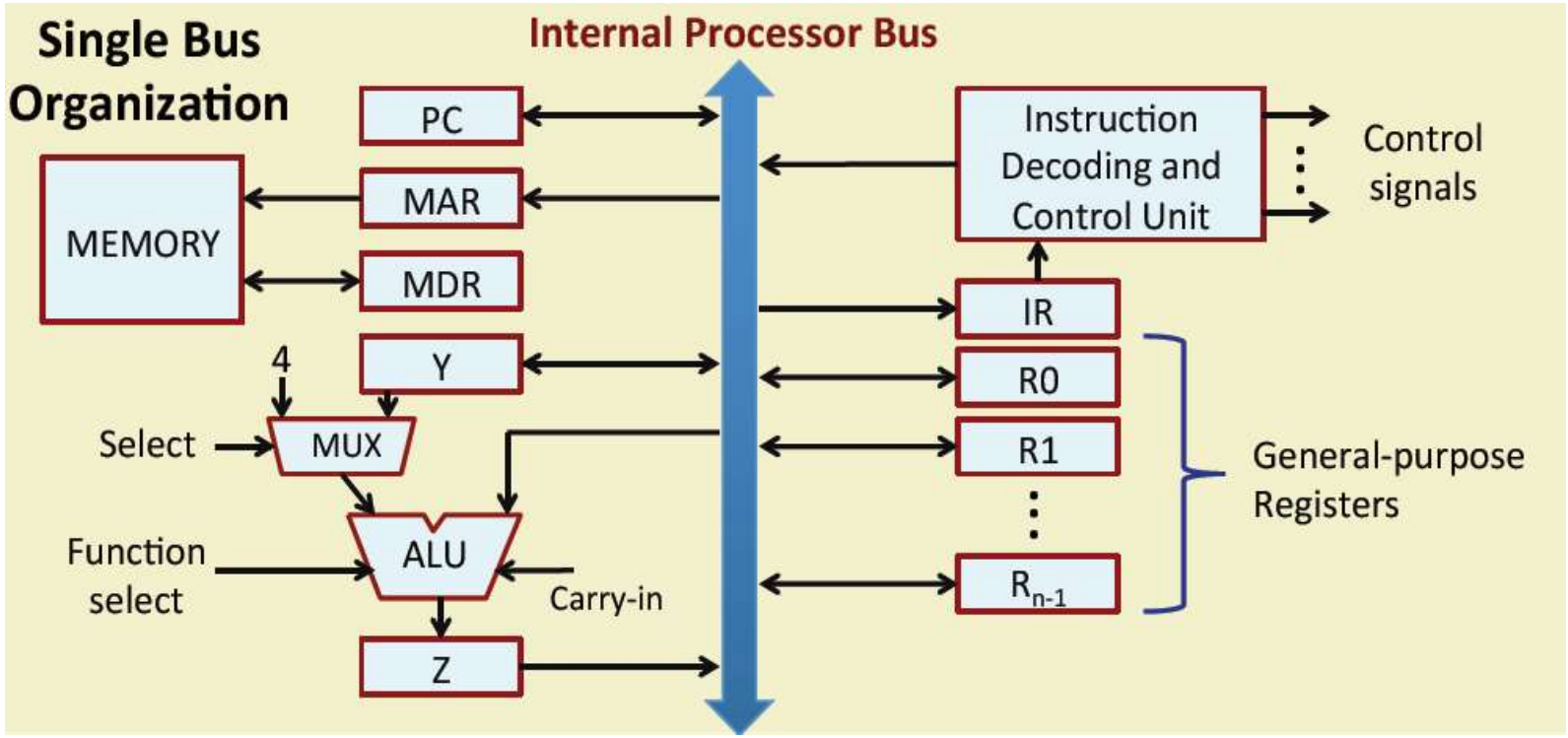
a) PC = 1000
b) MAR = 1000
c) PC = PC + 4 = 1004
d) MDR = "ADD R1, R2"
e) IR = "ADD R1, R2"
(Decode and finally execute)
f) R1 = R1 + R2

Requirement for Instruction Execution

- The necessary registers must be present.
- The internal organization of the registers must be known.
- The data path must be known.
- For instruction execution, a number of micro-operations are carried out on the data path.
  - May involve movement of data.

Kinds of Data Movement

- Broadly three types:
  a) Register to Register
  b) Register to ALU
  c) ALU to Register
- Data movement is supported in the data path by:
  - The Registers
  - The Bus (single or multiple)
  - The ALU temporary Register (Z)


Single Bus Organization
Internal Processor Bus
PC
MAR
MDR
MEMORY
Y
4
Select
MUX
Function select
ALU
Carry-in
Z
Instruction Decoding and Control Unit
Control signals
IR
R0
R1
R_{n-1}
General-purpose Registers

Single Internal Bus Organization

- All the registers and various units are connected using a single internal bus.
- Registers R_0-R_{n-1} are general-purpose registers used for various purposes.
- Y and Z are used for storing intermediate results and never used by instructions explicitly.
- The multiplexer selects either a constant 4 or output of register Y.
  - When PC is incremented, a constant 4 has to be added.

- The instruction decoder and control unit is responsible for performing the actions specified by the instruction loaded into IR.
- The decoder generates all the control signals in the proper sequence required to execute the instruction specified by the IR.
- The registers, the ALU and the interconnecting bus are collectively referred to as the *datapath*.

Kinds of Operations

- Transfer of data from one register to another.

  MOVE R1, R2

- Perform arithmetic or logic operation on data loaded into registers.

  ADD R1, R2

- Fetch the content of a memory location and load it into a register.

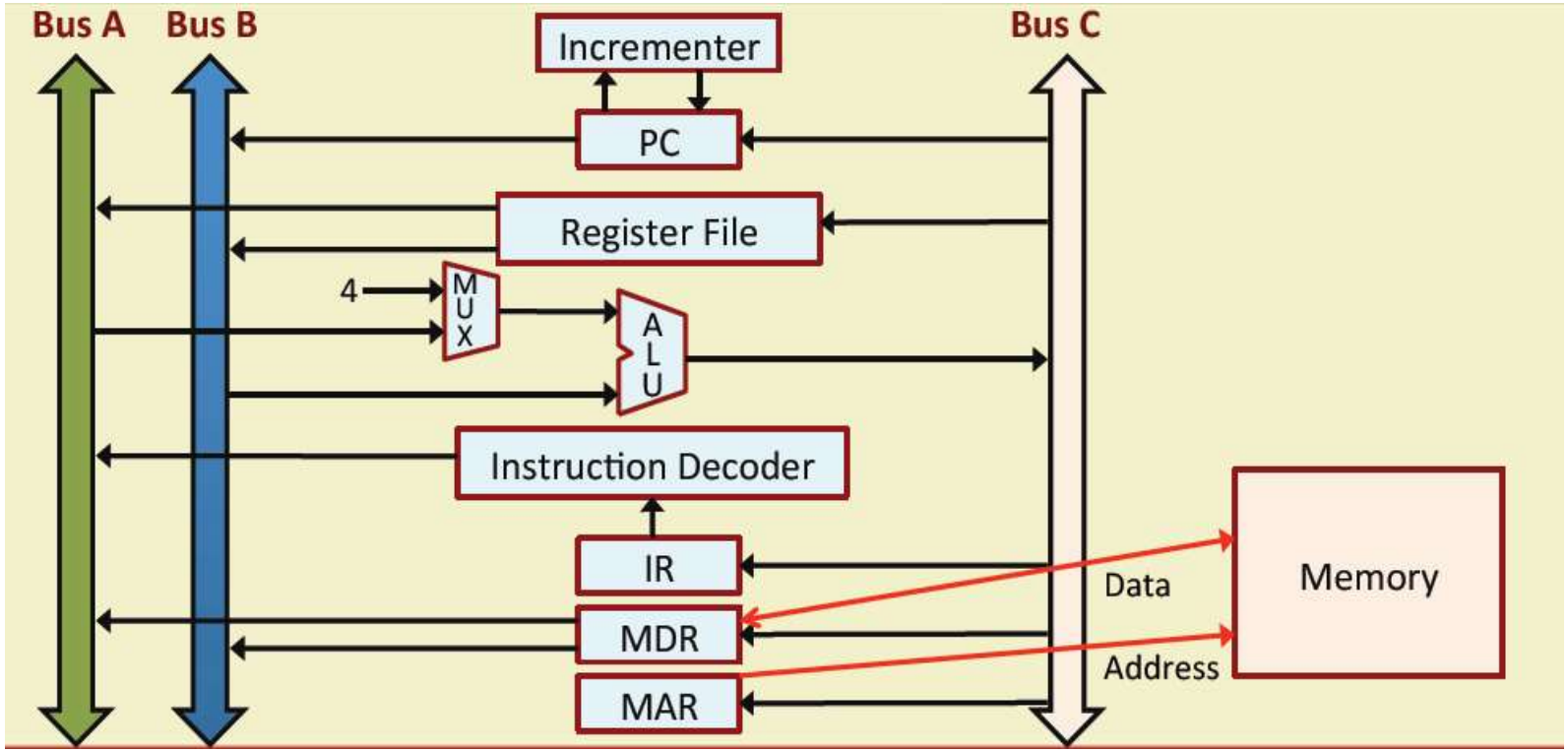
  MOVE R1, LOCA

- Store a word of data from a register into a given memory location.

  MOVE LOCA, R1

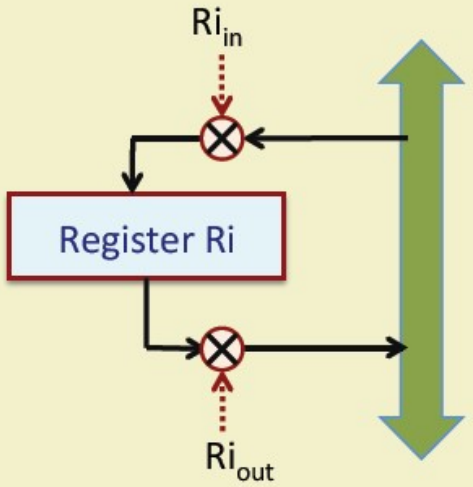
Three Bus Organization

- A typical 3-bus architecture for the processor datapath is shown in the next slide.
  - The 3-bus organization is internal to the CPU.
  - Three buses allow three parallel data transfer operations to be carried out.
- Less number of cycles required to execute an instruction compared to single bus organization.

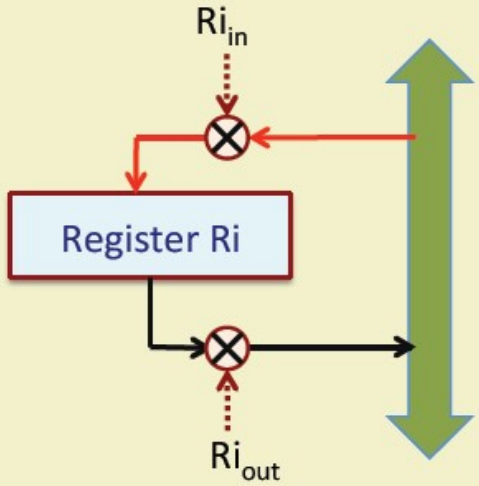

Bus A
Bus B
Incrementer
Bus C
PC
Register File
4
MUX
ALU
Instruction Decoder
IR
MDR
MAR
Data
Address
Memory

Organization of a Register

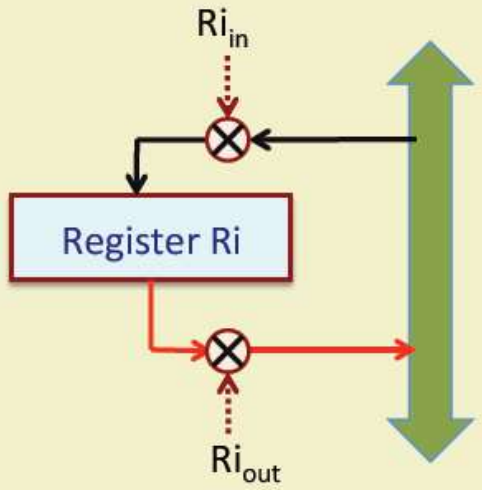
- A register is used for temporary storage of data (parallel-in, parallel-out, etc.).
- A register Ri typically has two control signals.
  - Ri_{in} : used to load the register with data from the bus.
  - Ri_{out} : used to place the data stored in the register on the bus.
- Input and output lines of the register Ri are connected to the bus via controlled switches.



- When (Ri_{in} = 1), the data available on bus is loaded into Ri.



- When (Ri_{out} = 1), the data from register Ri are placed on the bus.

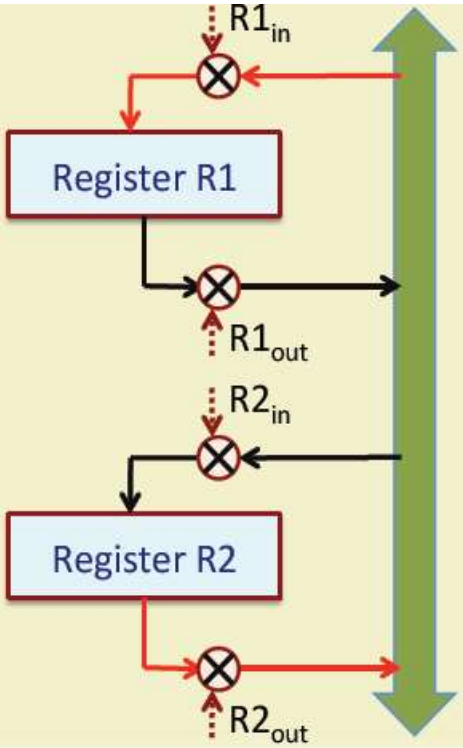


Register Transfer

MOVE R1, R2 // R1 ← R2

- Enable the output of R2 by setting $R2_{out}$ = 1.
- Enable the input of register R1 by setting $R1_{in}$ = 1.
- All operations are performed in synchronism with the processor clock.
  - The control signals are asserted at the start of the clock cycle.
  - After data transfer the control signals will return to 0.
- We write as *T1: $R2_{out}$, $R1_{in}$*

Time Step — Control Signals



ALU Operation

ADD R1, R2 // R1=R1 + R2

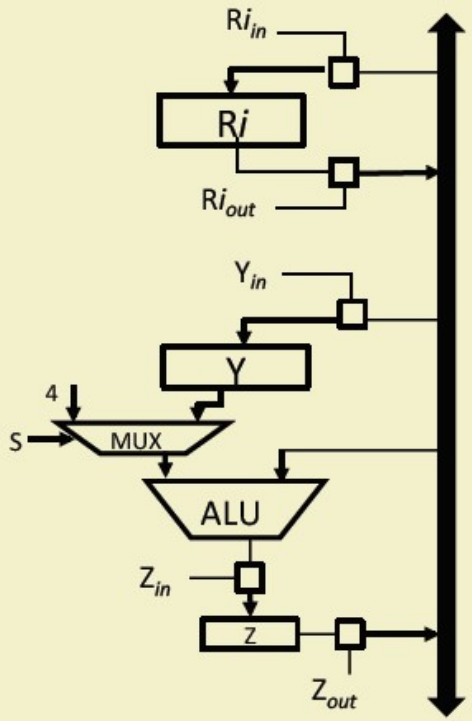
- Bring the two operands (R1 and R2) to the two inputs of the ALU.

  One through Y (R1) and another (R2) directly from internal bus.
- Result is stored in Z and finally transferred to R1.

  T1: $R1_{out}$, Y_{in}

  T2: $R2_{out}$, SelectY, ADD, Z_{in}

  T3: Z_{out}, $R1_{in}$



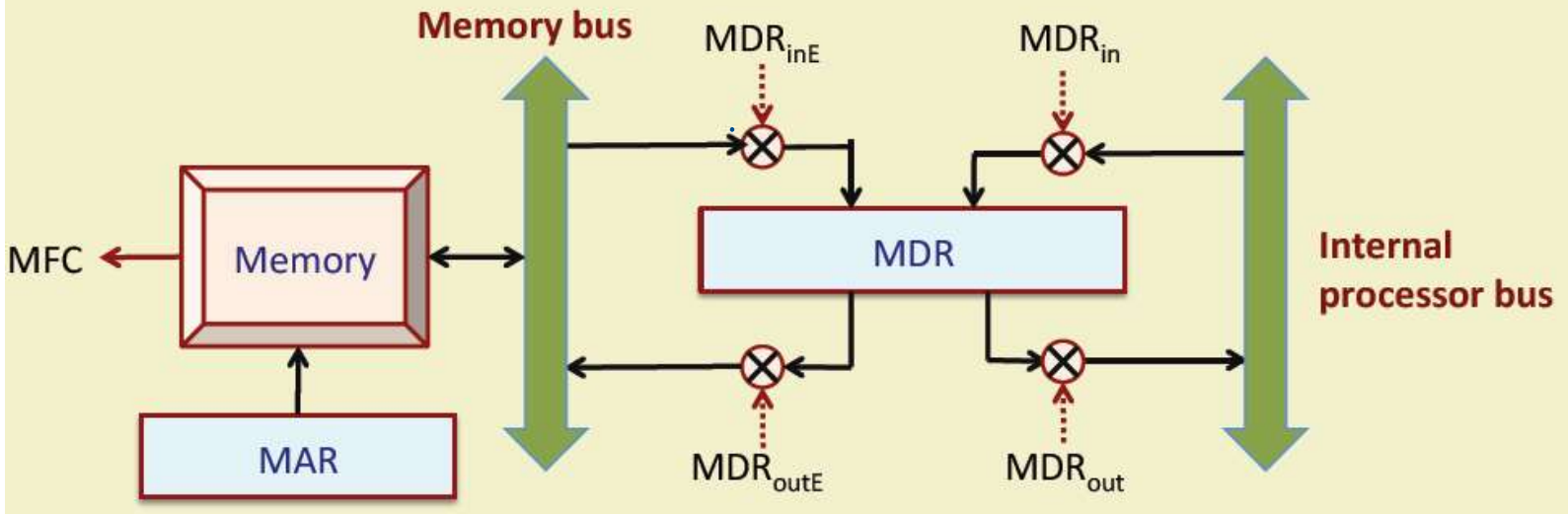
Fetching a Word from Memory

- The steps involved to fetch a word from memory:
  - The processor specifies the address of the memory location where the data or instruction is stored.
  - The processor requests a read operation.
  - The information to be fetched can either be an instruction or an operand of the instruction.
  - The data read is brought from the memory to MDR.
  - Then it can be transferred to the required register or ALU for further operation.

Storing a Word into Memory

- The steps involved to store a word into the memory:
  - The processor specifies the address of the memory location where the data is to be written.
  - The data to be written in loaded into MDR.
  - The processor requests a write operation.
  - The content of MDR will be written to the specified memory location.

Connecting MDR to Memory Bus and Internal Bus



- Memory read/write operation:
  - The address of memory location is transferred to MAR.
  - At the same time a read/write control signal is provided to indicate the operation.
  - For read the data from memory data bus comes to MDR by activating MDR_{inE}.
  - For write the data from MDR goes to memory data bus by activating the signal MDR_{outE}.

- When the processor sends a read request, it has to wait until the data is read from the memory and written into MDR.
- To accommodate the variability in response time, the process has to wait until it receives an indication from the memory that the read operation has been completed.
- A control signal called *Memory Function Complete* (MFC) is used for this purpose.
  - When this signal is 1, indicates that the content of the specified location is read and are available on the data line of the memory bus.
  - Then the data can be made available to MDR.

Fetch a word: MOVE R1, (R2)

1. MAR ← R2
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory
5. R1 ← MDR

Control steps:

a) $R2_{out}$, MAR_{in}, Read
b) MDR_{inE}, WMFC
c) MDR_{out}, $R1_{in}$

Store a word: MOVE (R1), R2

1. MAR ← R1
2. MDR ← R2
3. Start a Write operation on the memory bus
4. Wait for the MFC response from the memory

Control steps:

a) $R1_{out}$, MAR_{in}

b) $R2_{out}$, MDR_{in}, Write

c) MDR_{outE}, WMFC

Execution of a Complete Instruction

ADD R1, R2 // R1 = R1 + R2

T1: PC_{out}, MAR_{in}, Read, Select4, ADD, Z_{in}

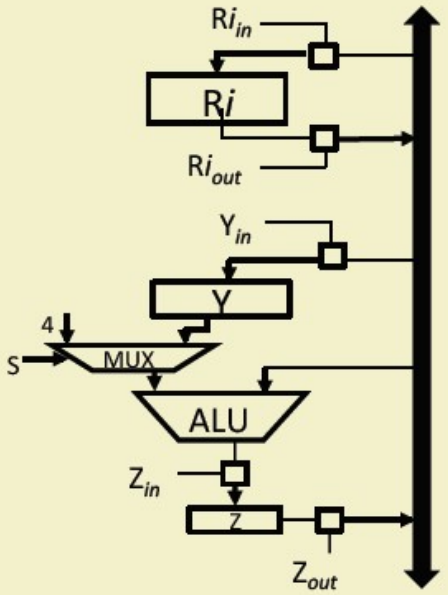
T2: Z_{out}, PC_{in}, Y_{in}, WMFC

T3: MDR_{out}, IR_{in}

T4: $R1_{out}$, Y_{in}, SelectY

T5: $R2_{out}$, ADD, Z_{in}

T6: Z_{out}, $R1_{in}$



Example for a Three Bus Organization

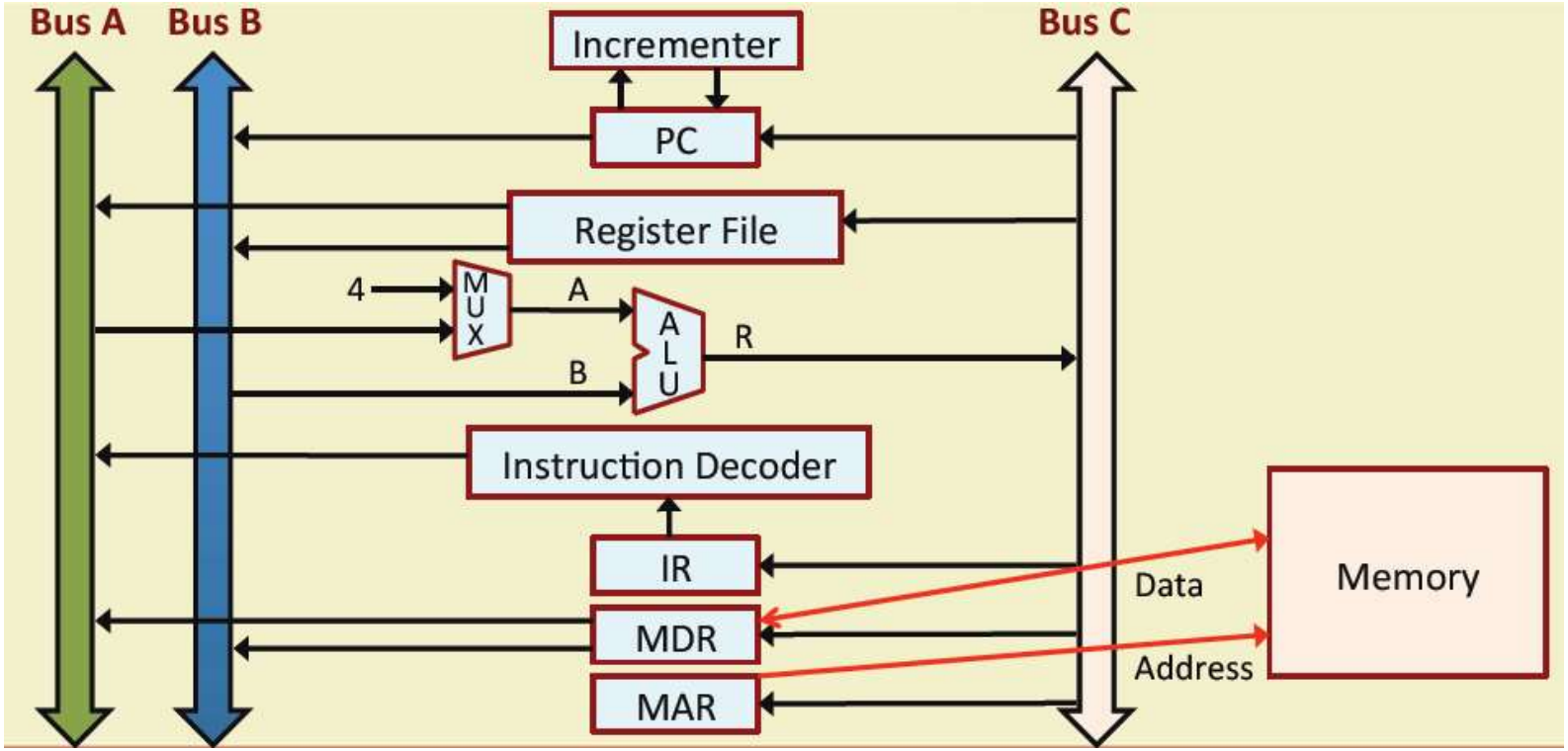
SUB R1, R2, R3 // R1 = R2 – R3

T1: PC_{out}, R = B, MAR_{in}, READ, IncPC

T2: WMFC

T3: MDR_{outB}, R = B, IR_{in}

T4: $R2_{outA}$, $R3_{outB}$, SelectA, SUB, $R1_{in}$, End


Bus A
Bus B
Incrementer
Bus C
PC
Register File
4
MUX
A
ALU
R
B
Instruction Decoder
IR
MDR
MAR
Data
Address
Memory

Various instructions: Control sequence

1.	ADD R1, R2	// R1 = R1+R2
2.	ADD R1, LOCA	// R1 = R1 + Mem[LOCA]
3.	LOAD R1, LOCA	// R1 = Mem[LOCA]
4.	STORE LOCA, R1	// Mem[LOCA] = R1
5.	MOVE R1, R2	// R1 = R2
6.	MOVE R1, #10	// R1 = 10
7.	BR LOCA	// PC = LOCA
8.	BZ LOCA	// PC = LOCA if Zero flag is set
9.	INC R1	// R1 = R1 + 4
10.	DEC R1	// R1 = R1 – 4
11.	CMP R1, R2	// R1 – R2
12.	HALT	// Machine Halt

1. ADD R1, R2 (R1 = R1+R2)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Add, Z_{in}
6	Z_{out}, $R1_{in}$, End

2. ADD R1, LOCA (R1 = R1 + Mem[LOCA])

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

3. LOAD R1, LOCA (R1 = Mem[LOCA])

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	WMFC
6	MDR_{out}, $R1_{in,}$ END

4. STORE LOCA, R1 (Mem[LOCA] = R1)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}
5	$R1_{out}$, MDR_{in}, Write
6	MDR_{outE}, WMFC, End

5. MOVE R1, R2 (R1 = R2)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R2_{out}$, $R1_{in,}$ END

6. MOVE R1, #10 (R1 = 10)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Immediate field of IR_{out}, $R1_{in}$, END

7. BRANCH Label (PC = PC + offset)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
5	Z_{out}, PC_{in}, End

8. BZ Label (if Z=1 PC = PC + offset)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}, If Z=0 then End
5	Z_{out}, PC_{in}, End

9. INC R1 (R1 = R1 + 4)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Select4, Add, Z_{in}
5	Z_{out}, $R1_{in}$, End

10. DEC R1 (R1 = R1 – 4)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Select4, SUB, Z_{in}
5	Z_{out}, $R1_{in}$, End

11. CMP R1, R2

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Sub, Z_{in}, End

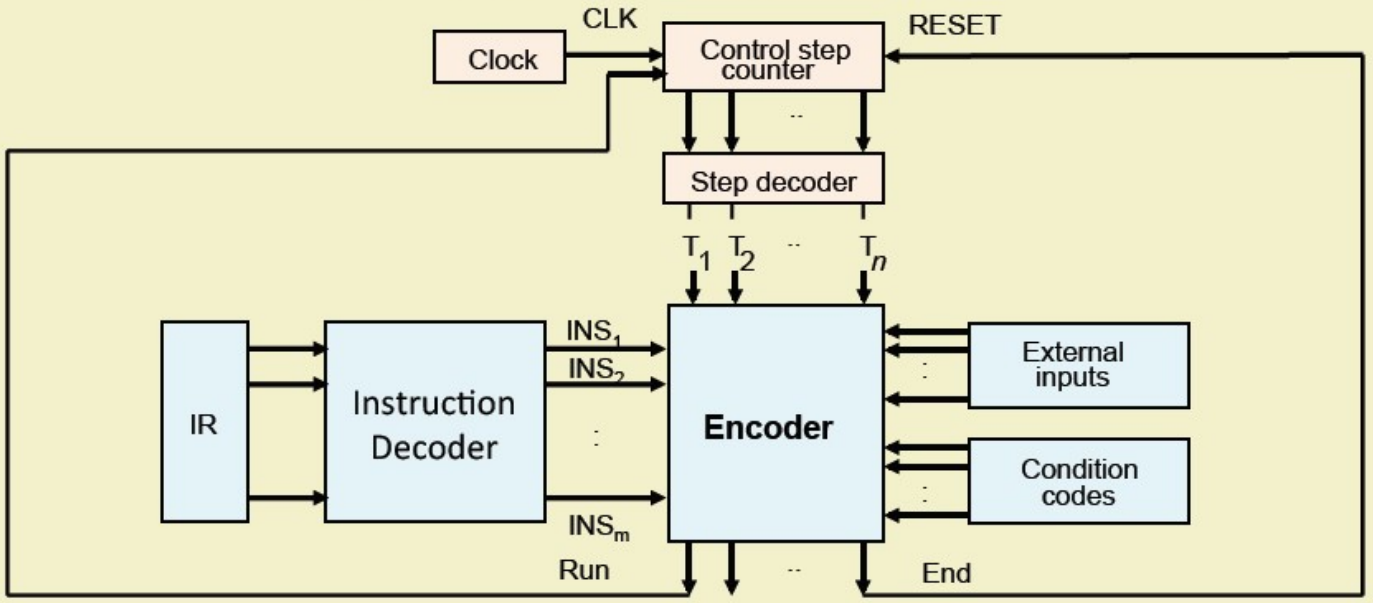
12. HALT

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	End

Introduction

- To execute an instruction, the processor must generate control signals for the data path in proper sequence.
  - Example: ADD R1, R2
    a) $R1_{out}$, Y_{in}, SelectY
    b) $R2_{out}$, ADD, Z_{in}
    c) Z_{out}, $R1_{in}$
- Two alternate approaches:
  1. Hardwired control unit design
  2. Microprogrammed control unit design

Hardwired Control unit



Sequence of control signals for ADD R1, LOCA

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IR_{out}, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Hardwired Control Unit Design

- Assumption:
  - Each step in this sequence is completed in one clock cycle.
- A counter is used to keep track of the time step.
- The control signals are determined by the following information:
  - Content of control step counter
  - Content of instruction register
  - Content of conditional code flags
  - External input signals such as MFC (Memory Function Complete)

- The encoder/decoder circuit is a combinational circuit which generates control signals depending on the inputs provided.
- The step decoder generates separate signal line for each step in the control sequence (T_1, T_2, T_3, etc.).
  - Depending on maximum steps required for an instruction, the step decoder is designed.
  - If a maximum of 10 steps are required, then a 4 x 16 step decoder is used.
- Among the total set of instructions, the instruction decoder is used to select one of them. (That particular line will be 1 and rest will be 0).
  - If a maximum of 100 instructions are present in the ISA then a 7 x 128 instruction decoder is used.

- At every clock cycle the RUN signal is used to increment the counter by one.
  - When RUN is 0 the counter stops counting.
  - This signal is needed when WMFC is issued.
- END signal starts a new instruction.
  - It resets the control step counter to its starting value.
- The sequence of operations carried out by the control unit is determined by the wiring of the logic elements and hence it is named *hardwired*.
- This approach of control unit design is fast but limited to the complexity of instruction set that is implemented.

Generation of Control Signals

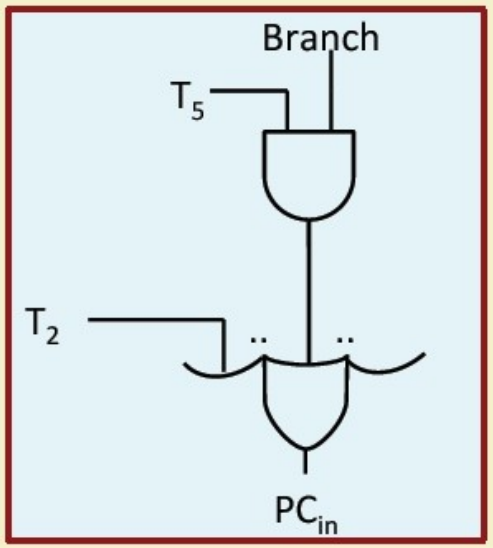
	ADD R1, R2
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Add, Z_{in}
6	Z_{out}, $R1_{in}$, End

	ADD R1, LOCA
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

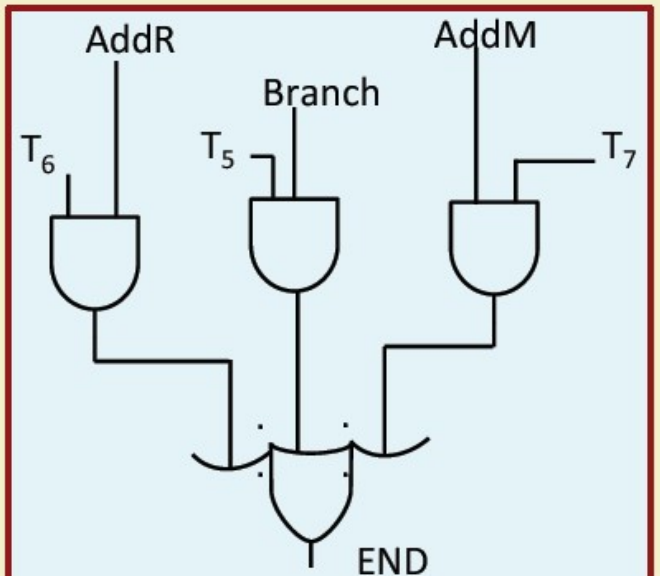
	BRANCH Label
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Generation of PC_{in} and END

$PC_{in} = T_2 + T_5.Branch$

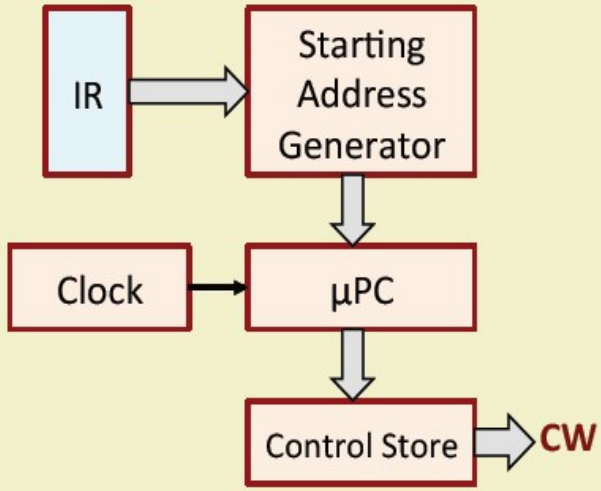


$END = T_6.ADDR + T_5.Branch + T_7.AddM$

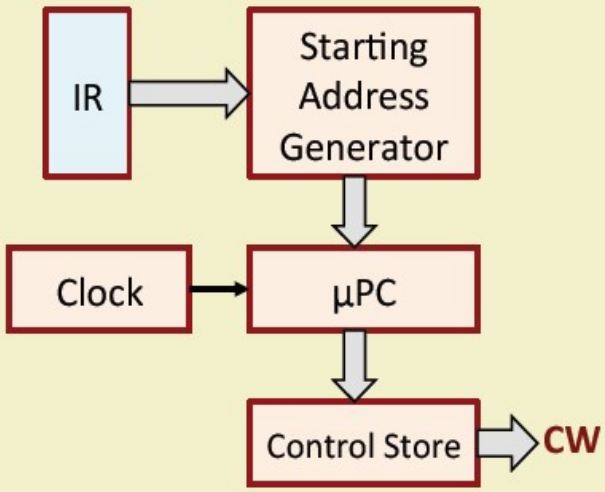


Microprogrammed Control Unit Design

- Control signals are generated by a program similar to machine language program.
- The *Control Store* (CS) stores the microroutines for all instructions of an ISA.
- The sequence of steps corresponding to the control sequence of a machine instruction is the *microroutine*.
- Each sequence of steps is a *control word* (CW) whose individual bits represent the various control signals.
- Individual control words in a microroutine are called *microinstructions*.



- Control-unit generates the control signals for an instruction by sequentially reading CWs of corresponding micro routine from CS.
- The *µPC* is used to read CWs sequentially from CS.
- Every time a new instruction is loaded into IR, output of Starting Address Generator is loaded into µPC.
- Then, µPC is automatically incremented by clock causing successive microinstructions to be read from CS.



Control Store for "ADD R1, R2"

Micro-instr.	...	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R2_{out}$	WMFC	End	...
1	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
3	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0

Control Store for "BRANCH LOCN"

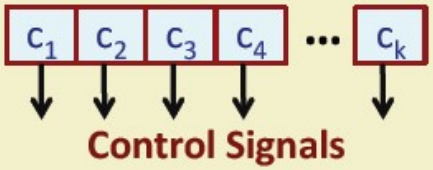
Micro-instr.	...	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	IR_{out}	WMFC	End	...
1	0	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0
3	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	**0**	1	1	0	1	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0

Horizontal versus Vertical Microinstruction Encoding

- Broadly there are two alternate schemes to code the control signals in the control memory.
  a) **Horizontal Micro-instruction Encoding**
     - Each control signal is represented by a bit in the micro-instruction.
     - Fewer control store words, with more bits per word.
  b) **Vertical Micro-instruction Encoding**
     - Each control word represents a single micro-instruction in encoded form.
     - k-bit control word can support up to 2^k micro-instructions.
     - More control store words, with fewer bits per word

- There can be a tradeoff between horizontal and vertical micro-instruction encoding.
  - Sometimes referred to as **Diagonal Micro-instruction Encoding**.
  - The control signals are grouped into sets S_1, S_2, etc., such that the control signals within a set are mutually exclusive.
- Summary:
  - Horizontal encoding supports unlimited parallelism among micro-instructions.
  - Vertical encoding supports strictly sequential execution of micro-instructions.
  - Diagonal encoding does not sacrifice the required level of parallelism, but uses less number of bits per control word as compared to horizontal encoding.

(a) Horizontal Micro-instruction Encoding



- Suppose that there are *k* control signals: $c_1, c_2, \ldots, c_k$.
- In horizontal encoding, every control word stored in control memory (CM) consists of *k* bits, one bit for every control signal.
- Several bits in a control word can be 1:
  - Parallel activation of several micro-operations in a single time step.

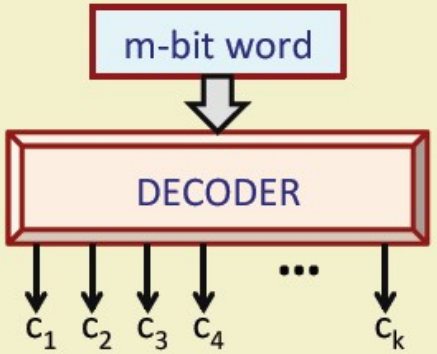
0	1	0	1	1	0

→ c_2, c_4 and c_5 are activated together

- Advantage:
  - Unlimited parallelism is possible in the activation of the micro-operations.
- Disadvantage:
  - Size of the control memory is large (word size is much longer).
  - Cost of implementation is higher.

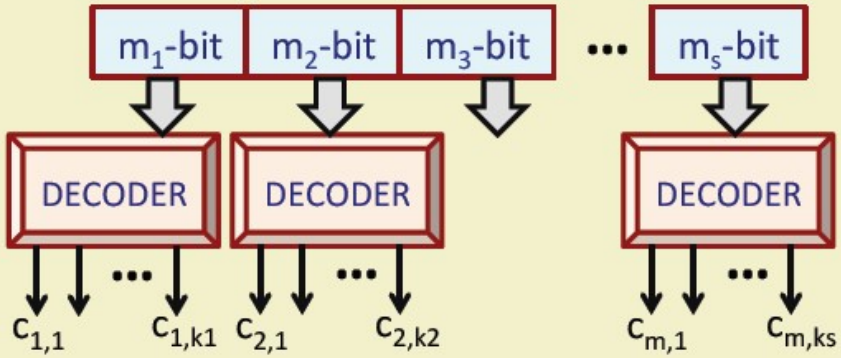
(b) Vertical Micro-instruction Encoding

- Again consider that there are k control signals: c_1, c_2, …, c_k.
- We encode the control signals in a m-bit word in the control memory, where $k \leq 2^m$.
- Depending on the m-bit control word, exactly one control signal will be activated (= 1), while all others will remain de-activated (= 0).
  - At most one control signal can be activated in a time step.



- Advantage:
  - Requires much smaller word size in control memory.
  - Low cost of implementation.
- Disadvantage:
  - More than one control signals cannot be activated at a time.
  - Requires sequential activation of the control signals, and hence more number of time steps.

(c) Diagonal Micro-instruction Encoding



- Suppose we group the set of k control signals into s groups, containing $k_1, k_2, \ldots, k_s$ signals.
- We encode the control signals in groups as shown, where $k_i \leq 2^{mi}$.
  - Within a group, at most one control signal can be activated in a time step.
  - Parallelism across groups is allowed.

- Advantages:
  - Maximum parallelism as required by the micro-programs can be supported.
  - Word size of control memory is less than that for horizontal encoding.
  - Used in practice.
- Disadvantages:
  - Multiple decoders (though smaller in sizes) are required.

Example 1

- Suppose there are 100 control signals in a processor data path.

  a) For horizontal encoding, control word size = 100 bits.

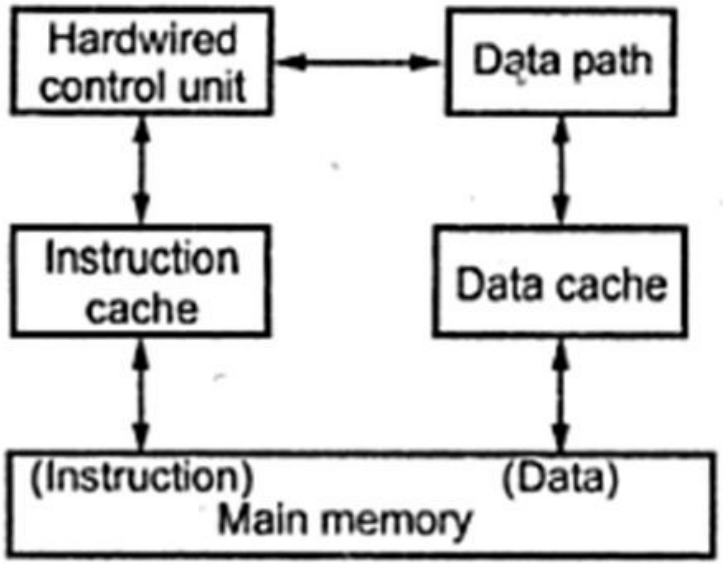
  b) For vertical encoding, control word size = $\lceil \log_2 100 \rceil$ = 7 bits.

  c) For diagonal encoding, suppose after analysis of the micro-programs, we divide the control signals into 5 groups, containing 25, 15, 40, 5 and 15 control signals respectively.
     - We have: $m_1 = 5, m_2 = 4, m_3 = 6, m_4 = 3, m_5 = 4$
     - Control word size = 5 + 4 + 6 + 3 + 4 = 22 bits.

$25 \leq 2^5$ $\quad 15 \leq 2^4$
$40 \leq 2^6$ $\quad 5 \leq 2^3$
$15 \leq 2^4$

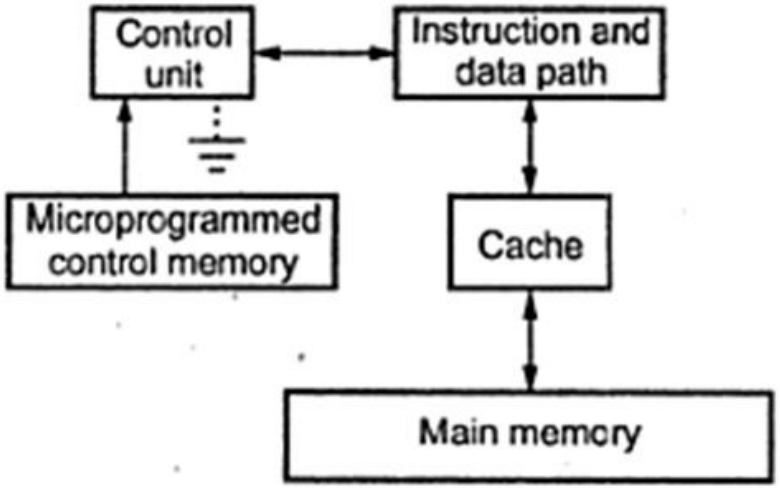
DESIGN ISSUES OF RISC PROCESSOR

RISC Architecture



(a) The RISC architecture with hardwired control and split instruction cache and data cache

CISC Architecture



(b) The CISC architecture with microprogrammed control and unified cache

Characteristic of RISC

- Although a variety of different approaches to reduce Instruction set architecture have been taken, certain characteristics are common to all of them:
  - One instruction per cycle.
  - Register–to–register operations.
  - Simple addressing modes.
  - Simple instruction formats.

Characteristic of RISC cont.d

1. One machine instruction per machine cycle :

- A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.
  With simple, one-cycle instructions there is little or no need of microcode, the machine instructions can be hardwired. Hardware implementation of control unit executes faster than the microprogrammed control, because it is not necessary to access a microprogram control store during instruction execution.
- One Cycle Execution Time. RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU

2. Register –to– register operations

- With register–to–register operation, a simple LOAD and STORE operation is required to access the memory, because most of the operation are register–to-register. Generally we do not have memory–to–memory and mixed register/memory operation.

Characteristic of RISC cont.d

3. Simple Addressing Modes

- Almost all RISC instructions use simple register addressing. For memory access only, we may include some other addressing, such as displacement and PC-relative. Once the data are fetched inside the CPU, all instruction can be performed with simple register addressing.

4. Simple Instruction Format

- Generally in most of the RISC machine, only one or few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed.
  With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit.

CISC

- Intel IA-32, belongs to the Complex Instruction Set Computer (CISC) design.
- The obvious reason for this classification is the “complex” nature of its Instruction Set Architecture (ISA). The motivation for designing such complex instruction sets is to provide an instruction set that closely supports the operations and data structures used by Higher-Level Languages (HLLs).
- However, the side effects of this design effort are far too serious to ignore.

Addressing Modes in CISC

- The decision of CISC processor designers to provide a variety of addressing modes leads to variable-length instructions. For example, instruction length increases if an operand is in memory as opposed to in a register.
  a) This is because we have to specify the memory address as part of instruction encoding, which takes many more bits.
  b) This complicates instruction decoding and scheduling. The side effect of providing a wide range of instruction types is that the number of clocks required to execute instructions varies widely.
  c) This again leads to problems in instruction scheduling and pipelining.

- Because CISC and RISC have their advantages and disadvantages, modern processors take features from both classes. For example, the PowerPC, which follows the RISC philosophy, has quite a few complex instructions.

RISC Vs CISC: An Example

- Multiplying Two Numbers in Memory. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4.
- The execution unit is responsible for carrying out all computations.
- However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F).
- Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3

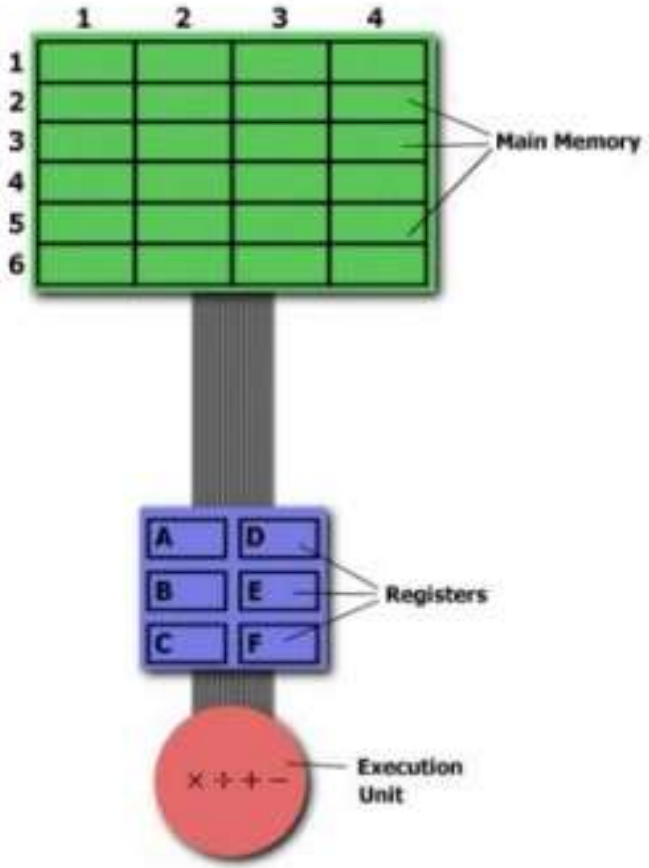


Figure 4 **Representation of Storage Scheme for a Generic Computer**

- The CISC Approach. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations.

- For this particular task, a CISC processor would come prepared with a specific instruction (say "MUL").

  – When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register.
  – Thus, the entire task of multiplying two numbers can be completed with one instruction:

MUL 2:3, 5:2

- MUL is what is known as a "complex instruction."
- It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.
- It closely resembles a command in a higher level language.
- For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a x b."

- Advantage.
- One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly.
- Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

- RISC processors only use simple instructions that can be executed within one clock cycle.
- Thus, the "MUL" command described above could be divided into three separate commands:
  - "LOAD," which moves data from the memory bank to a register,
  - "PROD," which finds the product of two operands located within the registers
  - "STORE," which moves data from a register to the memory banks.

LOAD A, 2:3

LOAD B, 5:2

PROD A, B

STORE 2:3, A

Analysis

- At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions.
- The compiler must also perform more work to convert a high-level language statement into code of this form.
- Advantage of RISC. However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MUL" command.
- These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers.
- Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.
- (1) Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform.
- (2) After a CISC-style "MUL" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register.
- In RISC, the operand will remain in the register until another value is loaded in its place.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Table 1 **Comparison of CISC and RISC Architectures**[vii]

RISC	CISC
Used by Apple	Used by Intel and AMD processors
Requires less registers therefore it is easier to design	Slower than RISC chips when performing instructions
Faster than CISC	More expensive to make compared to RISC
Reduced Instruction Set Computer	Complex Instruction Set Architecture
Pipelining can be implemented easily	Pipelining implementation is not easy
Direct addition is not possible	Direct addition between data in two memory locations. Ex.8085
Fewer, simpler and faster instructions	Large amount of different and complex instructions
RISC architecture is not widely used	Atleast 75% of the processor use CISC architecture
RISC chips require fewer transistors and cheaper to produce. Finally, it's easier to write powerful optimized compilers.	In common CISC chips are relatively slow (compared to RISC chips) per instruction, but use little (less than RISC) instructions.
RISC puts a greater burden on the software. Software developers need to write more lines for the same tasks.	In CISC, software developers no need to write more lines for the same tasks
Mainly used for real time applications	Mainly used in normal PC's, Workstations and servers
Large number of registers, most of which can be used as general purpose registers	CISC processors cannot have a large number of registers.
RISC processor has a number of hardwired instructions.	CISC processor executes microcode instructions.
Instructions are executed by hardware	Instructions are executed by micro program
Fixed format instructions	variable format instructions
Few instructions	Many instructions
Multiple instruction set	Single instruction set
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity is in the microprogram

The Performance Equation

- The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

  - CISC Approach. The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
  - RISC Approach. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.