Chapter 12

Processor Structure and Function

| Central Processing Unit	
3.1	CPU architecture, Register organization, Instruction formats and addressing modes(Intel processor).,Basic instruction cycle. Control unit Operation ,Micro operations : Fetch, Indirect, Interrupt , Execute cycle Control of the processor, Functioning of micro programmed control unit, Micro instruction Execution and Sequencing, Applications of Micro programming
3.2	RISC v/s CISC processors, RISC and CISC Architecture, RISC pipelining, Case study on SPARC

CPU Structure

- CPU must:

  - Fetch instructions

  - Interpret instructions

  - Fetch data

  - Process data

  - Write data

CPU With Systems Bus

ALU

Registers

Control Unit

Control Bus   Data Bus   Address Bus

System Bus
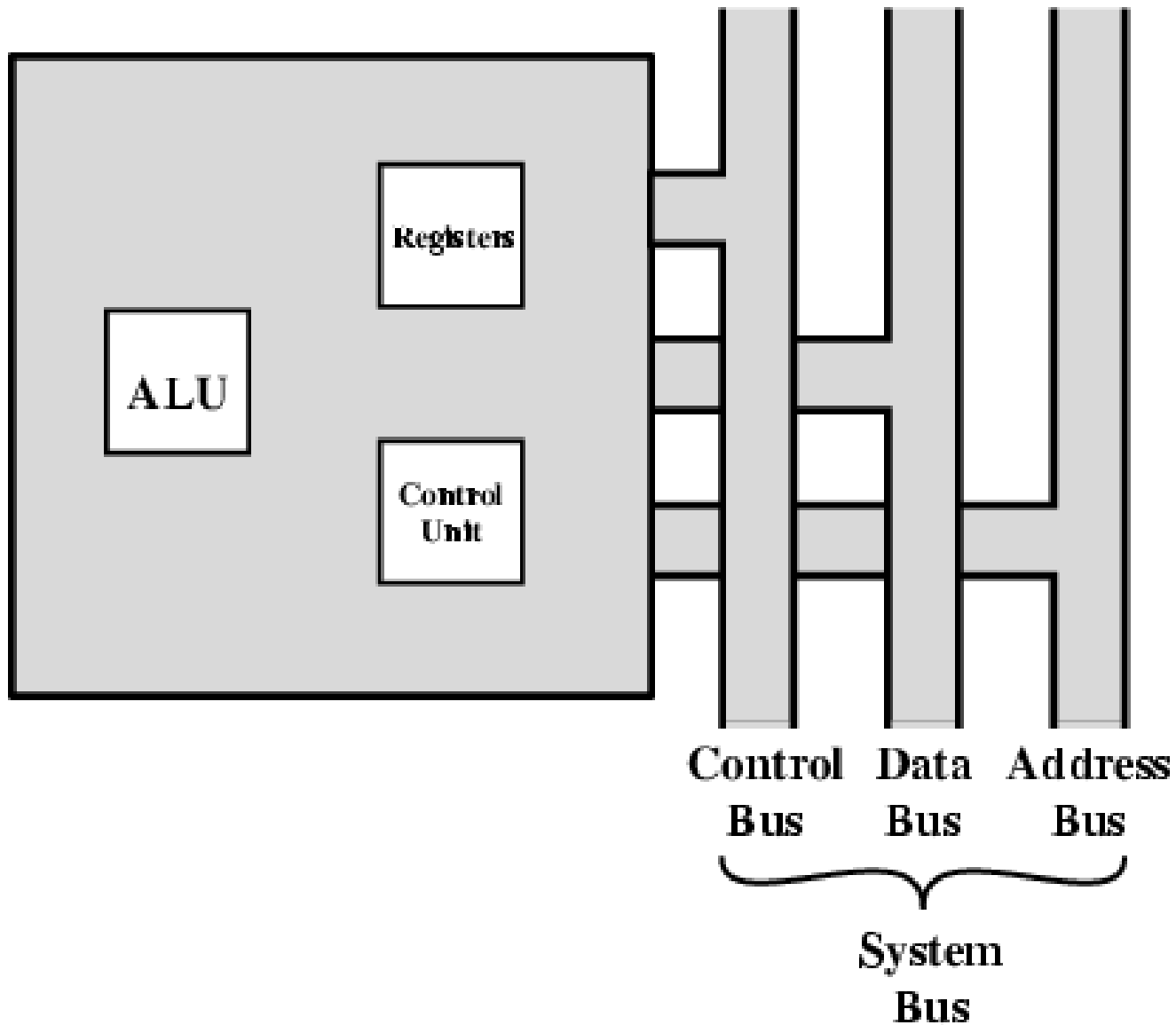
CPU Internal Structure
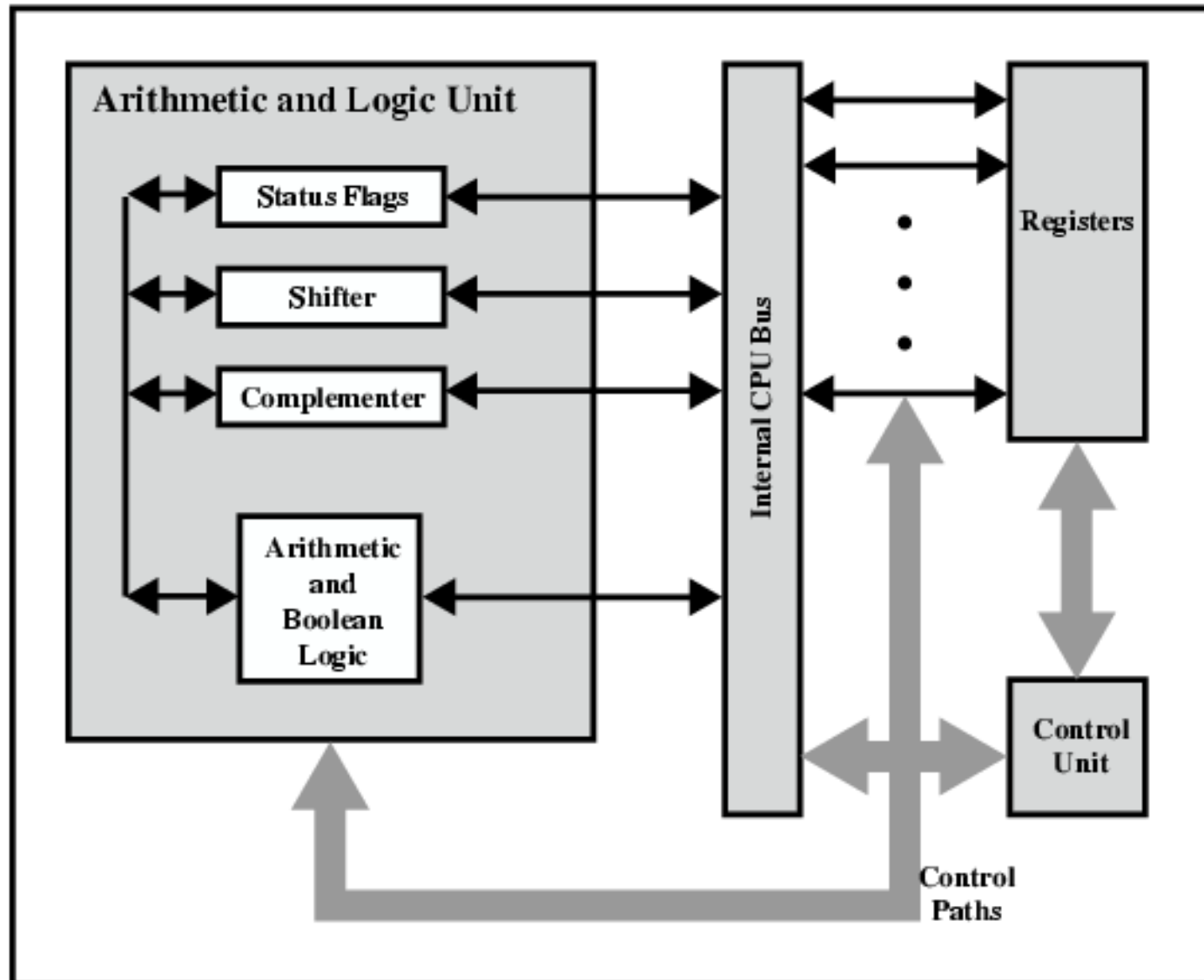
Registers

- 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

User Visible Registers

- General Purpose

- Data

- Address

- Condition Codes

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

General Purpose Registers

- May be true general purpose

- May be restricted

- May be used for data or addressing

- Data

  - Accumulator

- Addressing

  - Segment

- Why make them general purpose?

  – Increase flexibility and programmer options

  – Increase instruction size & complexity

How big?

- Large enough to hold full address

- Large enough to hold full word

- Often possible to combine two data registers

  - C programming

  - double int a;

  - long int a;

Condition Code Registers(Flag Reg)

- 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

Control & Status Registers

- Program Counter (PC)

- Instruction Decoding Register(IR)

- Memory Address Register(MAR)

- Memory Buffer Register(MBR)

Registers

- **Memory Address Register (MAR)**
  - Connected to address bus
  - Specifies address for read or write op

- **Memory Buffer Register (MBR)**
  - Connected to data bus
  - Holds data to write or last data read

- **Program Counter (PC)**
  - Holds **address of next** instruction to be fetched

- **Instruction Register (IR)**
  - Holds last instruction fetched/current instruction being executed

Program Status Word

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

General Registers

‣ **AX is the primary accumulator**; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

‣ **BX is known as the base register**, as it could be used in indexed addressing.

‣ **CX is known as the count register**, as the ECX, CX registers store the loop count in iterative operations.

‣ **DX is known as the data register**. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

Pointer Registers

- **Instruction Pointer (IP)** − The 16-bit IP register <mark>stores the offset address of the next instruction</mark> to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

- **Stack Pointer (SP)** − The 16-bit SP register provides the <mark>offset value within the program stack.</mark> SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.

- **Base Pointer (BP)** − The 16-bit BP register mainly <mark>helps in referencing the parameter variables passed to a subroutine.</mark> The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

Index Registers

SI and DI, are used for indexed addressing and sometimes used in addition and subtraction.

There are two sets of index pointers −

Source Index (SI) − It is used as source index for string operations.

Destination Index (DI) − It is used as destination index for string operations.

Control Registers

- The 32-bit instruction pointer register and the 32-bit flags register combined are considered as the control registers.

- Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

- The common flag bits are:

FLAG REGISTER 8086
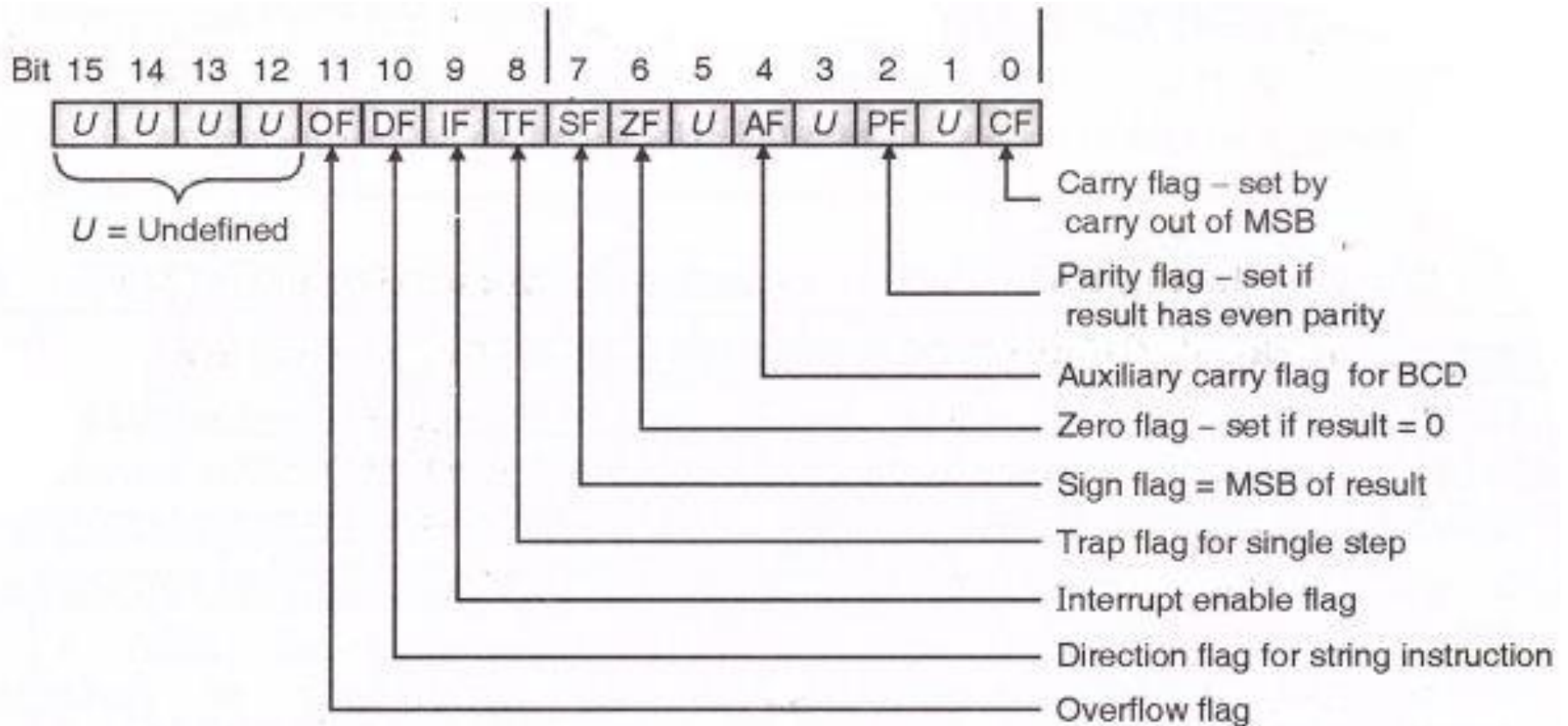
8086 flag register format

- **Overflow Flag (OF)** – It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.

- **Direction Flag (DF)** – It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

- **Trap Flag (TF)** − It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

- **Sign Flag (SF)** − It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

- **Zero Flag (ZF)** − It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.

- **Interrupt Flag (IF)** − It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.

- **Auxiliary Carry Flag (AF)** − It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

- **Parity Flag (PF)** − It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.

- **Carry Flag (CF)** − It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a *shift* or *rotate* operation.

Segment Registers

- Segments are specific areas defined in a program for containing data, code and stack. There are three main segments –

- **Code Segment** – It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.

- **Data Segment(DS,ES)** – It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.

- **Stack Segment** – It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

INSTRUCTION FORMAT (PENTIUM)

- Instruction :- MOV A (Destination),B(Source)
- Examples:

MOV AX,BX

Add AX,4

JMP

MUL 3,5

- Opcodes-Operation Code
- Operands-Data

→Instruction=Opcodes+Operands.

Instruction Formats

- Layout of bits in an instruction
- Includes **opcode**
- Includes (implicit or explicit) **operand(s)**
- Usually more than one instruction format in an instruction set
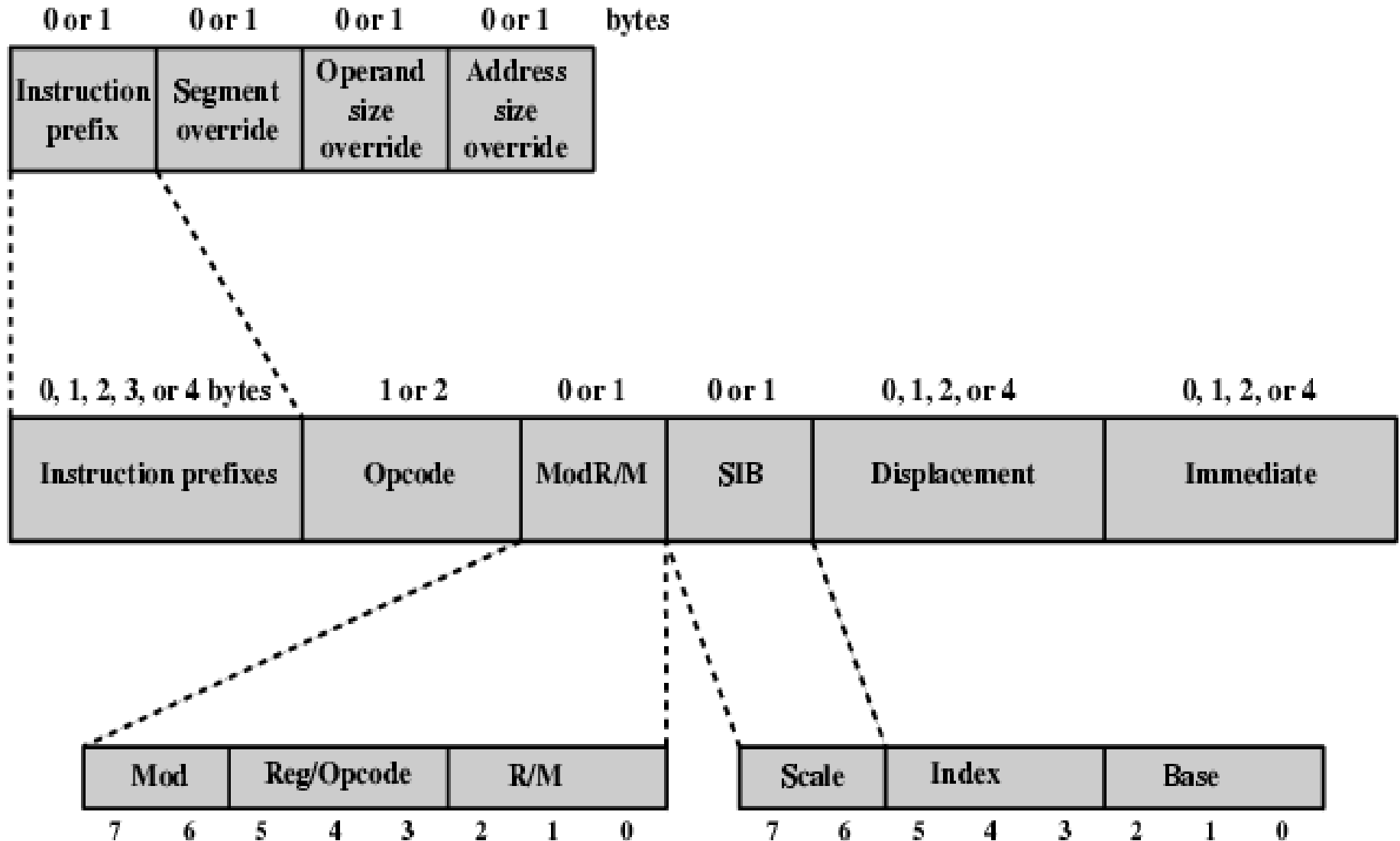
Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed

User wants more opcodes, operands, addressing modes , address range

Allocation of Bits

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets
- Address range
- Address Granularity

Pentium Instruction Format

Instruction prefix	Segment override	Operand size override	Address size override	bytes
0 or 1	0 or 1	0 or 1	0 or 1	

Instruction prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
0, 1, 2, 3, or 4 bytes	1 or 2	0 or 1	0 or 1	0, 1, 2, or 4	0, 1, 2, or 4

Mod	Reg/Opcode	R/M
7 6	5 4 3	2 1 0

Scale	Index	Base
7 6	5 4 3	2 1 0

PENTIUM INSTRUCTION FORMAT

- **Instruction Prefixes**-**LOCK** prefix or one of the **REPEAT** prefixes(REPE,REPNE,REPZ,REPNZ….)

- **Segment Override**-explicitly specifying segment register

- **Address Size**-16 or 32 bit (switch)

- **Operand Size**-16 or 32 bit (switch)

PENTIUM INSTRUCTION FORMAT

- **Opcode**
- **Mod R/m**-addressing
  - Mod+ r/m (combined info of registers and addressing modes)
  - Register-register or opcode info
- **SIB**
  - **Scale-**scale factor for scaled indexing
  - **Index**-specifies index register(SI,DI)
  - **Base**-specifies base register(BX)
- **Displacement**-8 or 16 or 32 bit
- **Immediate**-provides the value of 8/16/32 bit operand

PENTIUM INSTRUCTION FORMAT

- **Instruction Prefixes**-LOCK prefix or one of the REPEAT prefixes

- **Segment Override**-explicitly specifying segment register

- **Address Size**-16 or 32 bit (switch)

- **Operand Size**-16 or 32 bit (switch)

William Stallings
Computer Organization and Architecture
6th Edition

Chapter 11

Instruction Sets:

Addressing Modes and Formats

Addressing Modes

- **Immediate**
- **Direct**
- **Indirect**
- **Register**
- **Register Indirect**
- **Displacement (Indexed)**
- **Stack**

Add   AX, [1234h]

Add   [1234h], AX
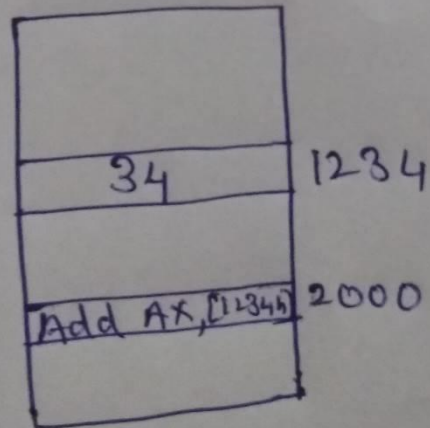
memory (code + data)

Addressing Modes

- 8086 accesses data operands in different ways.

- When 8086 executes an instruction , it performs specific function on data(operand).

- Operands may be contained in registers, within the instruction itself, in memory or in I/O ports.

- To access these different types of operands ,8086 has to address memory or I/O.

- The address of memory and I/O can be calculated in several different ways, referred as Addressing Modes.

Immediate Addressing

- In this mode data is present in address field of instruction .

- Designed like one address instruction format
  - Note:Limitation in the immediate mode is that the range of constants are restricted by size of address field.

- **FEATURES**
  - **Operand is part of instruction**
  - **Operand = address field**
  - **No memory reference to fetch data**
  - **Fast**
  - **Limited range**

Instruction

| | Operand |

(a) Immediate
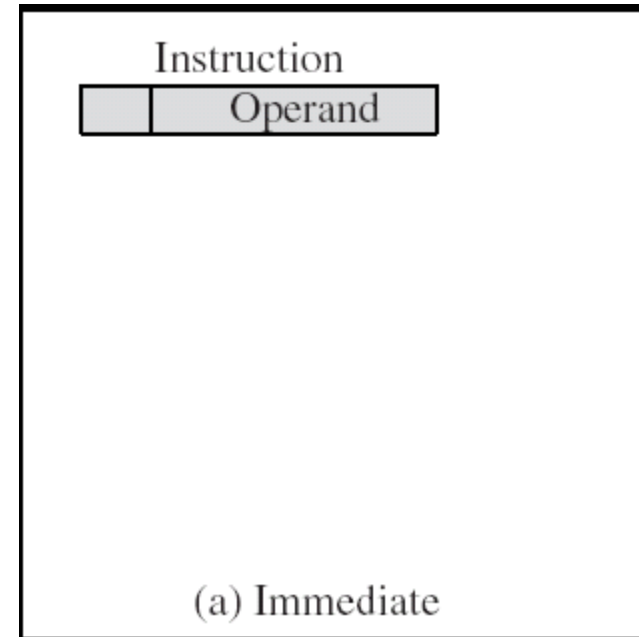
MOV AX, #2000

MOV CL, 0Ah

ADD AL, #45

ADD AX, 0000

MOV CX, 4929 H

ADD AX, 2387 H,

MOV AL, FFH

Opcode	Address

Data is directly stored here.

Direct(M) Addressing Diagram

(b) Direct
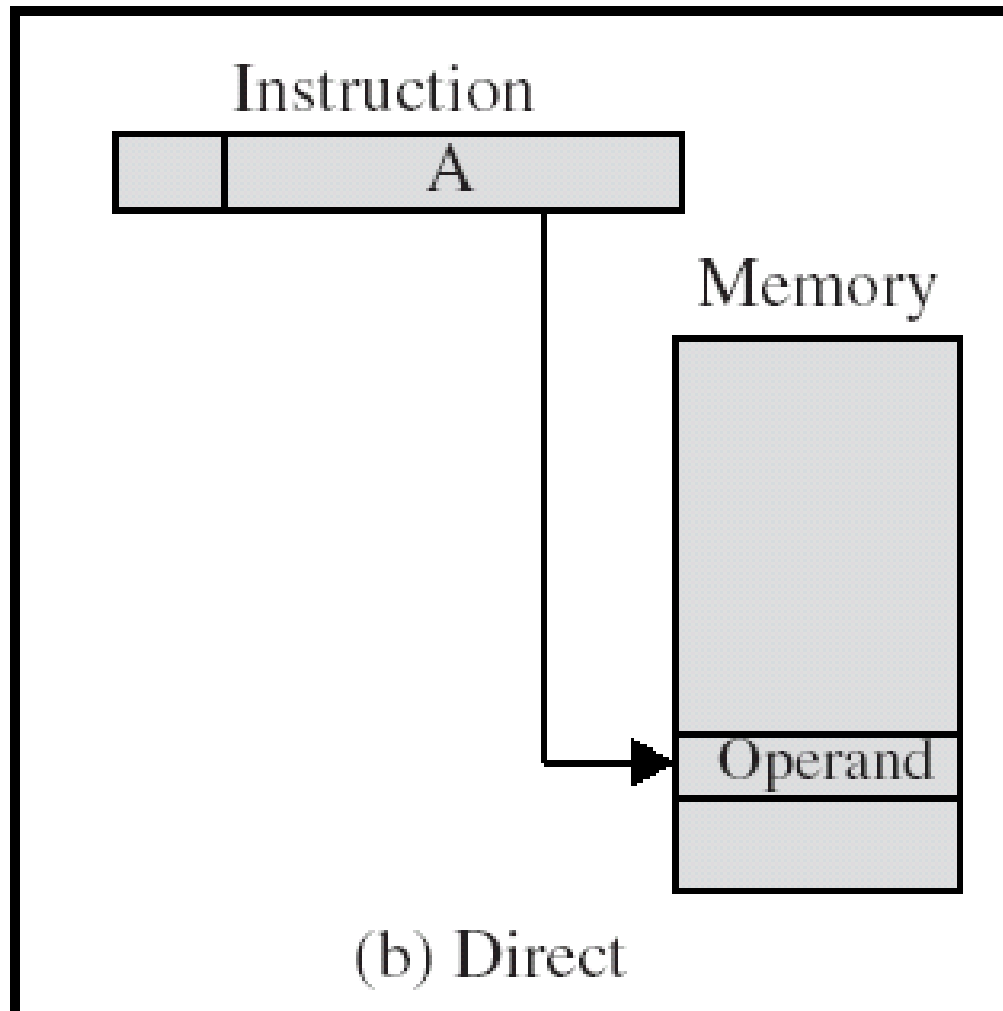
MOV AX, [200h]

ADD [1234H],AX

MOV AX, [0500h]

ADD AL,[0301]

Instruction

Memory

| Effective address | → | Data |

- The operand's offset is given in the instruction as an 8 bit or 16 bit displacement element.

- In this addressing mode the 16 bit effective address of the data is the part of the instruction.

- *Here only one memory reference operation is required to access the data.*

Direct Addressing

- **Address field contains address of operand**

- **EFFECTIVE ADDRESS EA = address field (A)**

  - **Look in memory at address value for operand**

- **Single memory reference to access data**

- **No additional calculations to work out effective address**

- **Limited address space**

Indirect Addressing Diagram

(c) Indirect
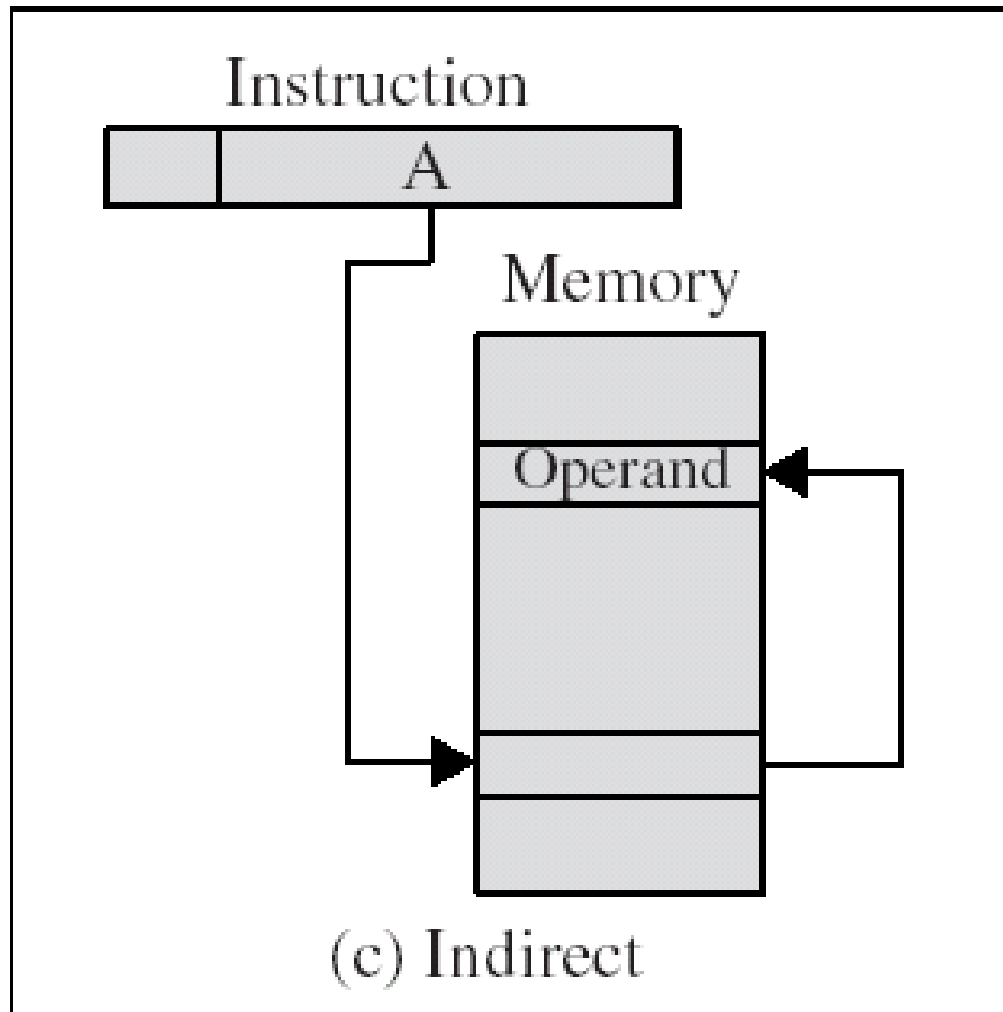
- In this mode address field of instruction contains the address of effective address.

- Here two references are required. 1st reference to get effective address. 2nd reference to access the data.

-

Based on the availability of Effective address, Indirect mode is of two kind:

- <u>REGISTER INDIRECT</u>: In this mode effective address is in the register, and corresponding register name will be maintained in the address field of an instruction.

    - *Here one register reference,one memory reference is required to access the data.*

- <u>MEMORY INDIRECT</u>: In this mode effective address is in the memory, and corresponding memory address will be maintained in the address field of an instruction.

  – *Here two memory reference is required to access the data*

Indirect Addressing (1/2)

- Memory cell pointed to by address field contains the address

  of (pointer to) the operand

- EA =(A)

  - Look in A, find address (A) and look there for operand

- e.g. ADD AX, (A)

  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing (2/2)

- Large address space

- 2^n where n = word length

- May be nested, multilevel, cascaded

  - e.g. EA = (((A)))

    - Draw the diagram yourself

- Multiple memory accesses to find operand

- Hence slower

Register Addressing Diagram
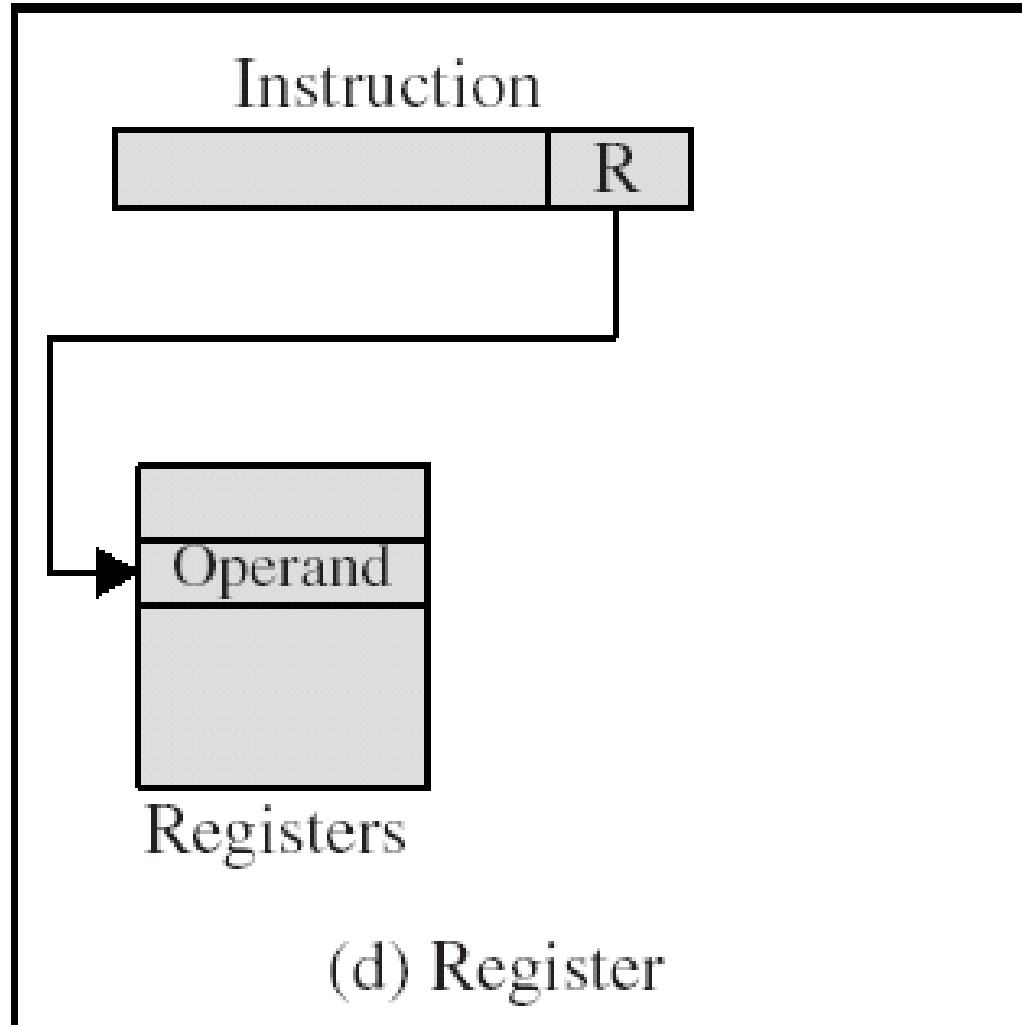
- In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers.

- The data is in the register that is specified by the instruction.

  – *Here one register reference is required to access the data.*

Instruction

Register

→

Register

Data

Register Addressing (1/2)

- Operand is held in register named in address filed

- EA = R

- Limited number of registers

- Very small address field needed

  - Shorter instructions

  - Faster instruction fetch

  - MOV AX, BX

  - ADD AX, BX

Register Addressing (2/2)

- No memory access

- Very fast execution

- Very limited address space

- Multiple registers helps performance

  - Requires good assembly programming or compiler writing

  - N.B. C programming

    - register int a;

- c.f. Direct addressing

Register Indirect Addressing Diagram
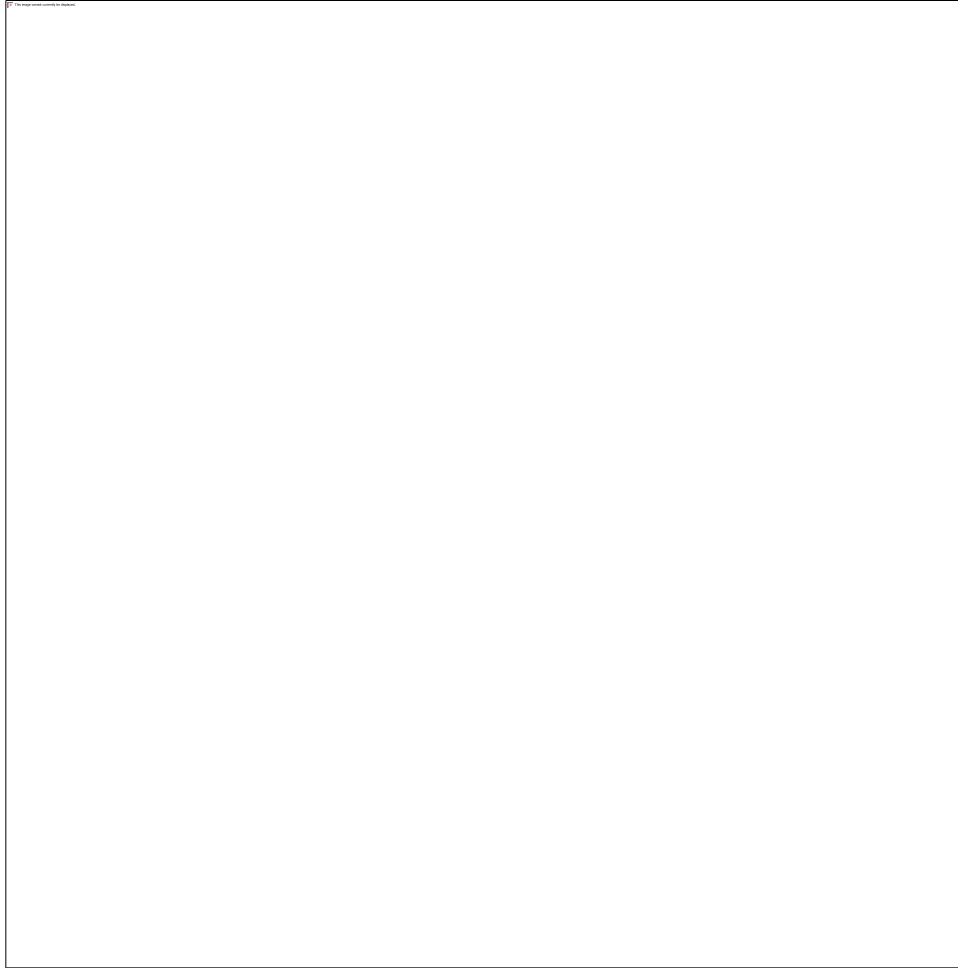
- In this addressing the operand's offset is placed in any one of the registers BX,BP,SI,DI as specified in the instruction.

- The effective address of the data is in the base register or an index register that is specified by the instruction.

  - *Here two register reference is required to access the data.*

Instruction	Register	Memory
Register	Effective Address	Data

Register Indirect Addressing

MOV AX, [BX]

(move the contents of memory location s addressed by the register BX to the register AX)

MOV AX, [DX]

ADD AL, [BX]

MOV [AX], CX

- Operand is in memory cell pointed to by contents of register R

- Large address space (2^n)

- One fewer memory access than indirect addressing

Displacement Addressing Diagram
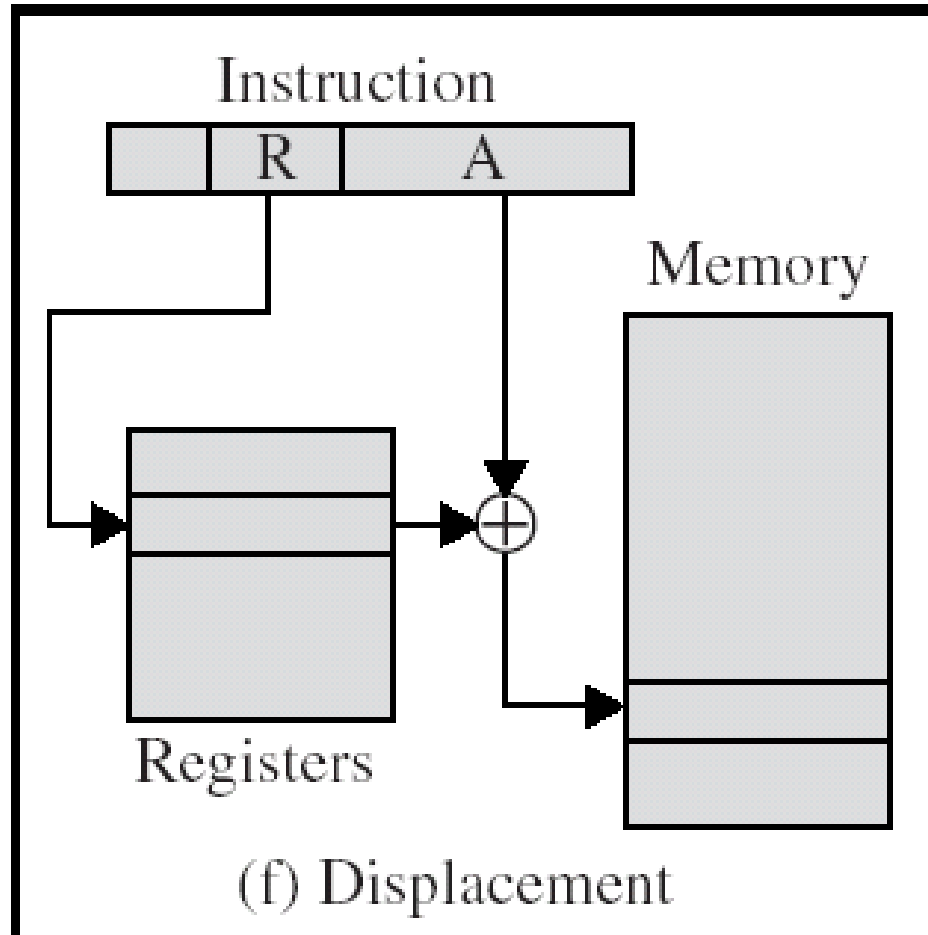
(f) Displacement

Displacement Addressing

- EA = A + (R)

- Effective address=start address + displacement

- Effective address=Offset + (Segment Register)

- Use direct and register indirect

- Address field hold two values

  - A = base value

  - R = register that holds displacement

  - or vice versa

Base-Register Addressing

- Base register addressing mode is used to implement inter segment transfer of control.

- In this mode effective address is obtained by adding base register value to address field value.

- ADD AX,[BX][SI] Base indexed addressing mode

- ADD AX,[BX][SI + 20] Base index with displacement

Memory: Indexed Addressing

The operand's offset is the sum of the content of an

index register SI or DI and an 8 bit or 16 bit

displacement.

   mov AL,[SI + 5]

- MOV AX, [SI +05]

Stack Addressing

- Operand is (implicitly) on top of stack

- e.g.
  - ADD   Pop top two items from stack and add and push

61
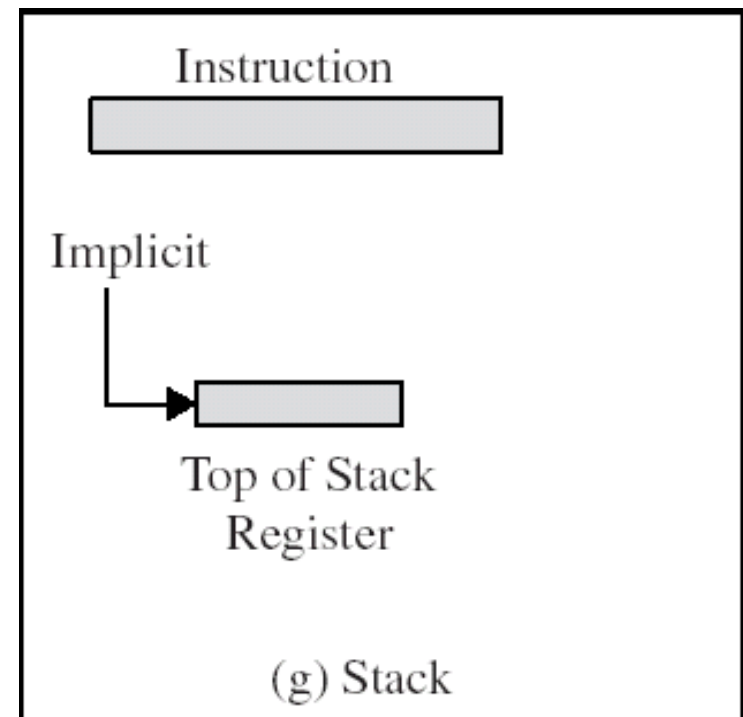
Instruction Cycle

Instruction Cycle

- Two steps:
  - Fetch
  - Execute
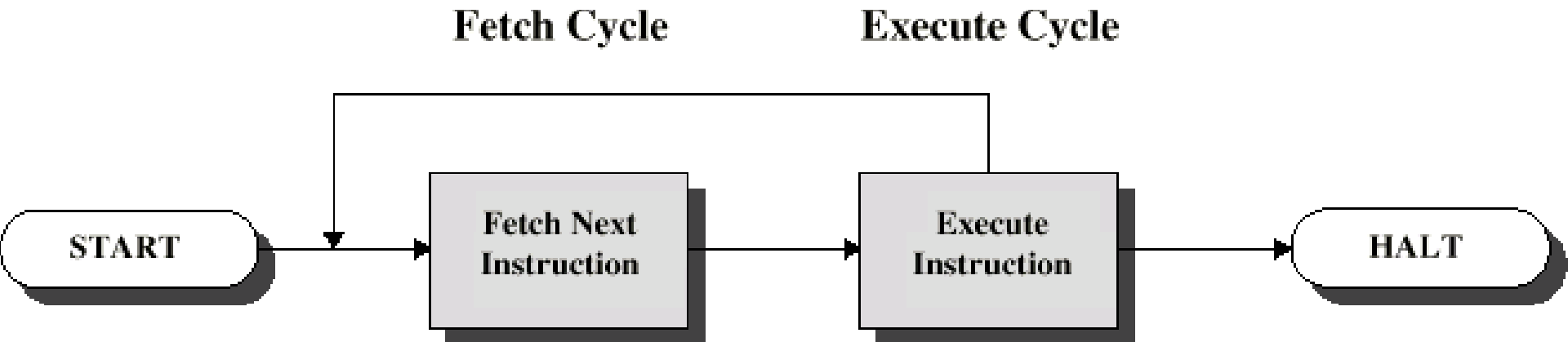
Instruction Cycle with Interrupts
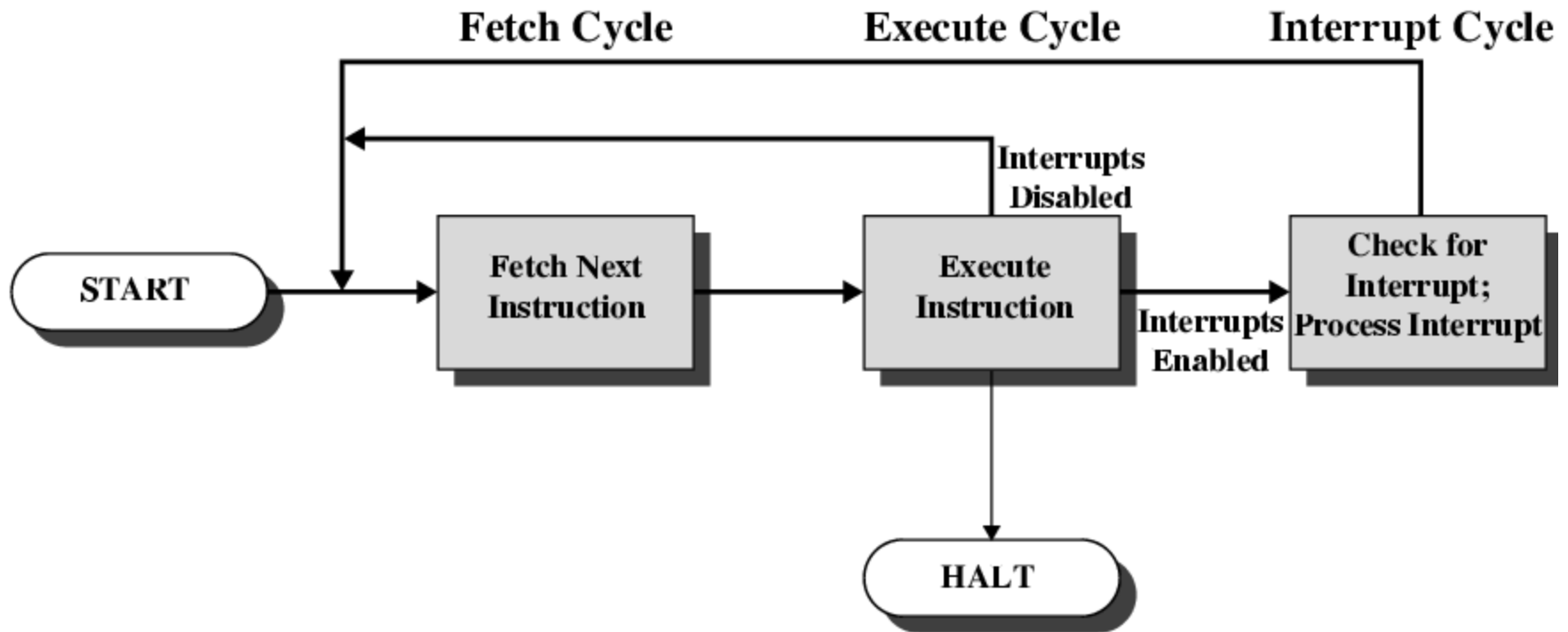
Instruction Cycle

- It is the time in which a single instruction is fetched from memory, decoded, and executed

- An **Instruction Cycle** requires the following subcycle:

  - **FETCH**

  - **EXECUTE**

  - **INDIRECT**

  - **INTERRUPT**

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

Instruction Cycle
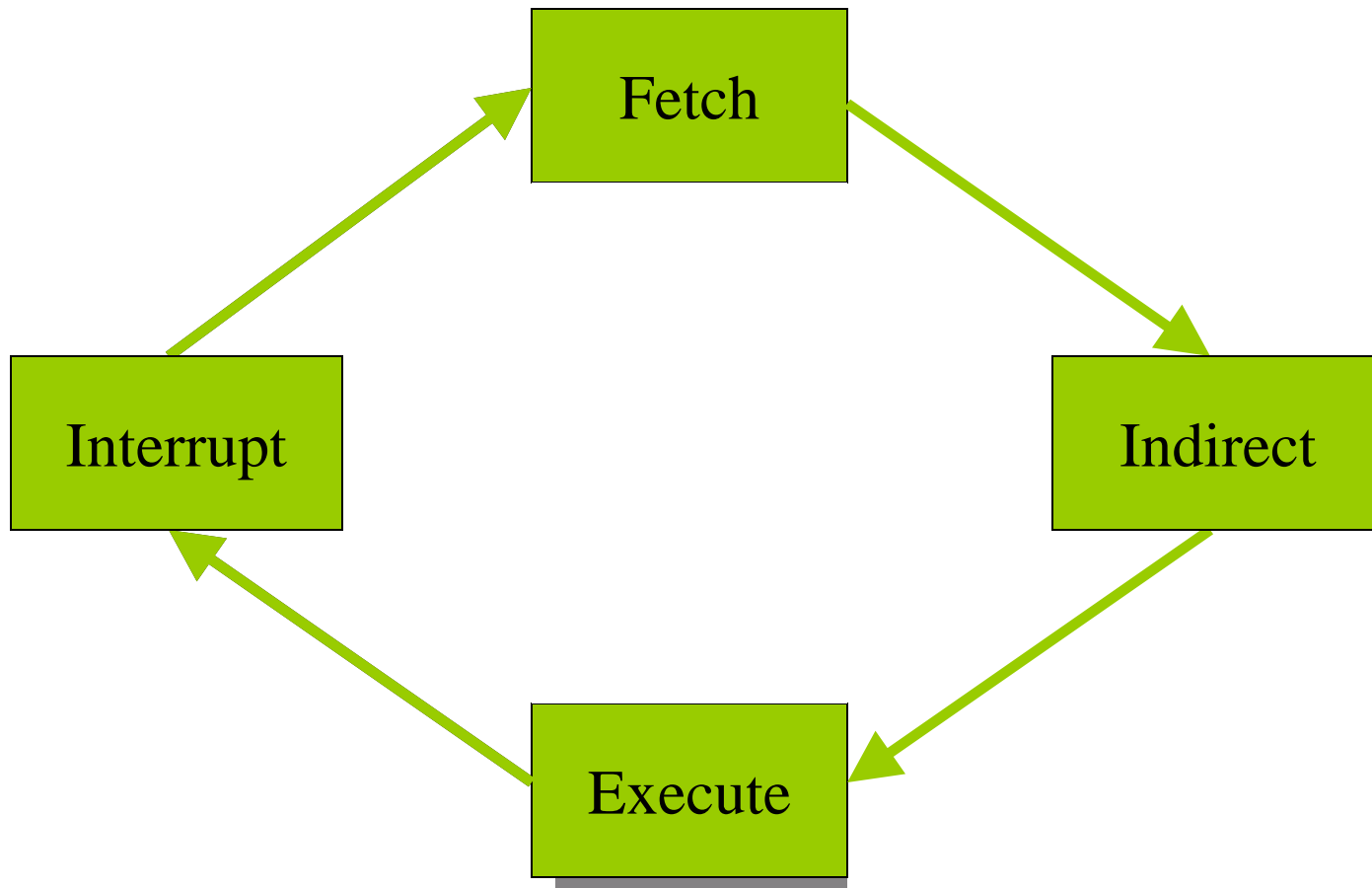
Instruction Cycle State Diagram
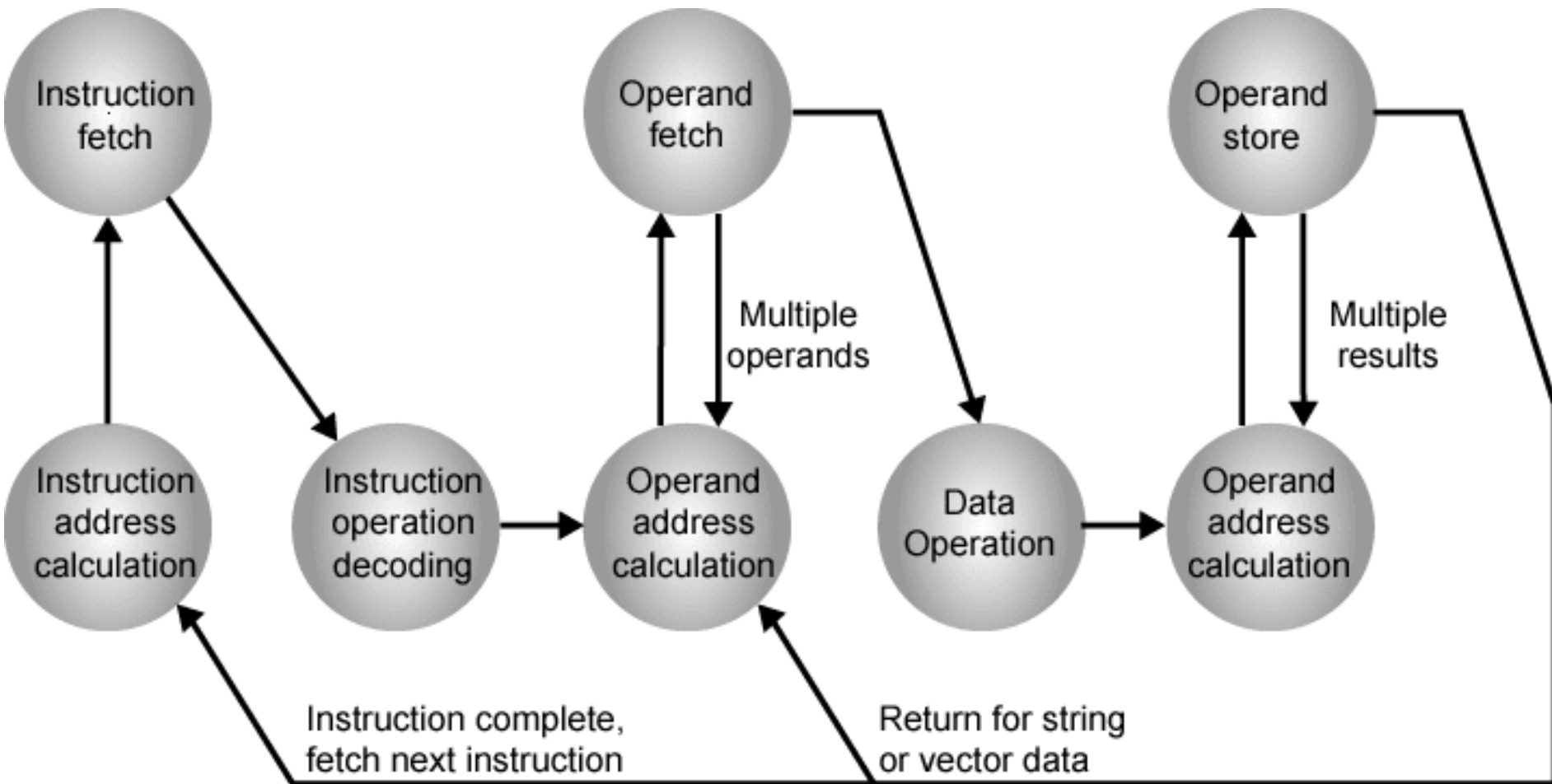
Figure 10.1   Instruction Cycle State Diagram

Instruction Cycle State Diagram

- This illustrates more correctly the nature of the instruction cycle.

- Once an instruction is fetched, its operand specifiers must be identified.

- Each input operand in memory is then fetched, and this process may require indirect addressing.

- Register-based operands need not be fetched.

- Once the opcode is executed, a similar process may be needed to store the result in main memory.

Registers

- **Memory Address Register (MAR)**

  – Connected to address bus

  – Specifies address for read or write op

- **Memory Buffer Register (MBR)**

  – Connected to data bus

  – Holds data to write or last data read

- **Program Counter (PC)**

  – Holds address of next instruction to be fetched

- **Instruction Register (IR)**

  – Holds last instruction fetched/current instruction being executed

Fetch Cycle

- **Program Counter (PC)** holds address of next instruction to be fetched

- Processor fetches instruction from memory location pointed to by PC

- Increment PC
  - Unless told otherwise

- Instruction loaded into Instruction Register (IR)

- Processor interprets instruction and performs required actions

Fetch Sequence (symbolic)

t1 : MAR ← PC

t2 : MBR ← MEMORY

PC ← (PC) + I

t3 : IR ← (MBR)

- **The Fetch Cycle –**
  At the beginning of the fetch cycle, the address of the next instruction to be executed is in the *Program Counter*(PC).

MAR	
MBR	
PC	0000000001100100
IR	
AC	
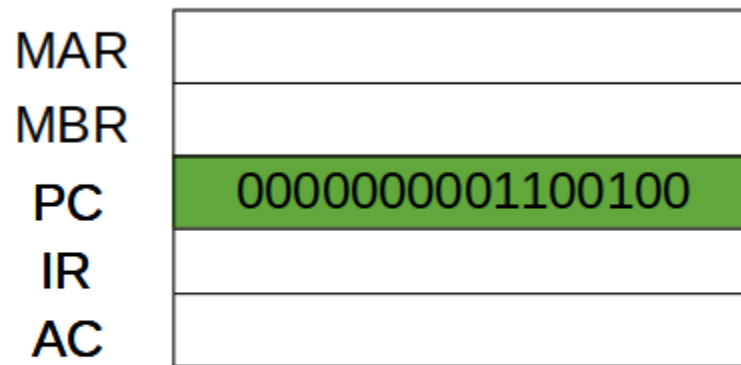
BEGINNING

Step 1:

- The address in the program counter is moved to the memory address register(MAR), as this is the only register which is connected to address lines of the system bus.

MAR	0000000001100100
MBR	
PC	0000000001100100
IR	
AC	

FIRST STEP

Step 2:

- The address in MAR is placed on the address bus, now the control unit issues a READ command on the control bus, and the result appears on the data bus and is then copied into the memory buffer register(MBR).

- Program counter is incremented by one, to get ready for the next instruction.(These two action can be performed simultaneously to save time)

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100101
IR	
AC	

SECOND STEP

Step 3:

- The content of the MBR is moved to the instruction register(IR)

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100100
IR	0001000000100000
AC	

THIRD STEP

Indirect Cycle

t1 : MAR ← (IR(ADDRESS))

t2 : MBR ← MEMORY

t3 : IR(ADDRESS) ← (MBR(ADDRESS))

Step 1:

The address field of the instruction is transferred to the MAR.

This is used to fetch the address of the operand.
Step 2:

The address field of the IR is updated from the MBR.(So that it now contains a direct addressing rather than indirect addressing)
Step 3:

The IR is now in the state, as if indirect addressing has not been occurred.

Interrupt Cycle

- At the completion of the Execute Cycle, a test is made to determine whether any enabled interrupt has occurred or not.

- If an enabled interrupt has occurred then Interrupt Cycle occurs.

- The nature of this cycle varies greatly from one machine to another.

t1 : MBR ← (PC)

t2 : MAR ← SAVE_ADDRESS

PC ← ROUTINE_ADDRESS

t3 : MEMORY ← (MBR)

- Step 1: Contents of the PC is transferred to the MBR, so that they can be saved for return.

  Step 2: MAR is loaded with the address at which the contents of the PC are to be saved.

  PC is loaded with the address of the start of the interrupt-processing routine.

  Step 3: MBR, containing the old value of PC, is stored in memory.

  - **Note:** In step 2, two actions are implemented as one micro-operation. However, most processor provide multiple types of interrupts, it may take one or more micro-operation to obtain the save_address and the routine_address before they are transferred to the MAR and PC respectively.a

Execute Cycle (ADD)

Different for each instruction

e.g. ADD R1,X - add the contents of location X to Register 1 , result in R1

$$t_1 : IR \rightarrow MAR$$

$$t_2 : MEMORY \rightarrow MDR$$

$$t_3 : MDR + R1 \rightarrow R1$$

Constituent Elements of Program Execution
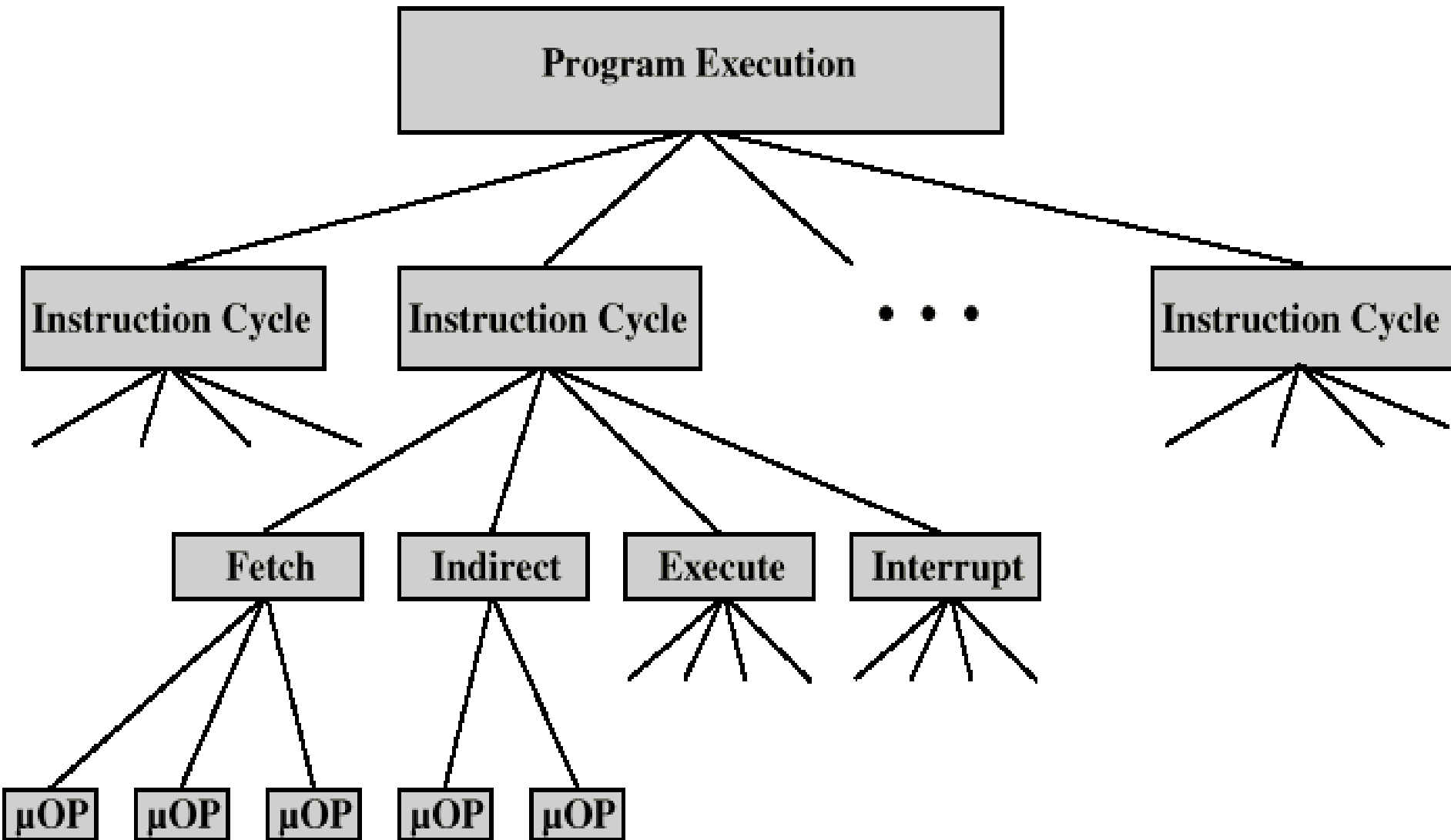
Micro-Operations

- A computer executes a program

- Fetch/execute cycle

- Each cycle has a number of steps

- Called micro-operations

- Each step does very little

- Atomic operation of CPU

Types of Micro-operation

- Transfer data between registers

- Transfer data from register to external

- Transfer data from external to register

- Perform arithmetic or logical ops

William Stallings
Computer Organization and Architecture
8th Edition

Control Unit Operation

Functional Requirements(of Control Unit)

- Define basic elements of processor

- Describe micro-operations processor performs

- Determine functions control unit must

  perform

Registers

- Memory Address Register (MAR)
  - Connected to address bus
  - Specifies address for read or write op
- Memory Buffer Register (MBR)
  - Connected to data bus
  - Holds data to write or last data read
- Program Counter (PC)
  - Holds address of next instruction to be fetched
- Instruction Register (IR)
  - Holds last instruction fetched/current instruction being executed

Model of Control Unit
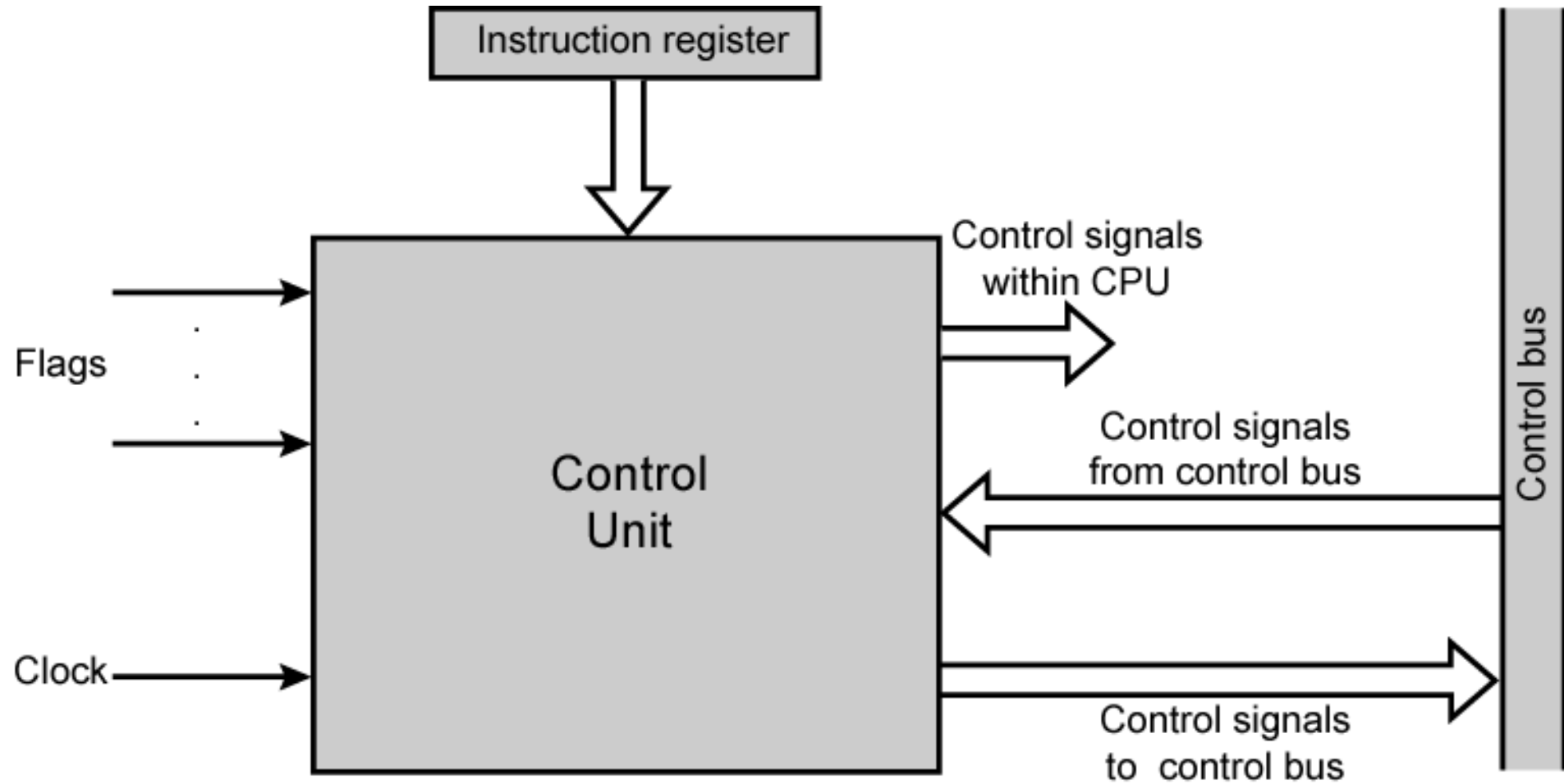
Functions of Control Unit

- **Sequencing**

  ◦ Causing the CPU to step through a series of micro-operations

- **Execution**

  ◦ Causing the performance of each micro-op

- This is done using Control Signals→→

Control Signals(input)

- **Clock**
  - One micro-instruction (or set of parallel micro-instructions) per clock cycle
- **Instruction register**
  - Op-code for current instruction
  - Determines which micro-instructions are performed
- **Flags**
  - State of CPU
  - Results of previous operations
- **From control bus**
  - Interrupts
  - Acknowledgements

Control Signals - output

- Within CPU

  ◦ Cause data movement

  ◦ Activate specific functions

- Via control bus

  ◦ To memory

  ◦ To I/O modules

Control Unit Organization

Implementation

- All the control unit does is generate a set of **control signals**

- Each control signal is **on** or **off**

- Represent each control signal by a **bit**

- Have a **control word** for each micro-operation

- Have a **sequence of control words** for each machine code instruction

- Add an **address** to specify the next micro-instruction, depending on conditions

Chapter 16

Micro-programmed Control

William Stallings
Computer Organization and Architecture
8th Edition

Functioning of Micro programmed Control Unit
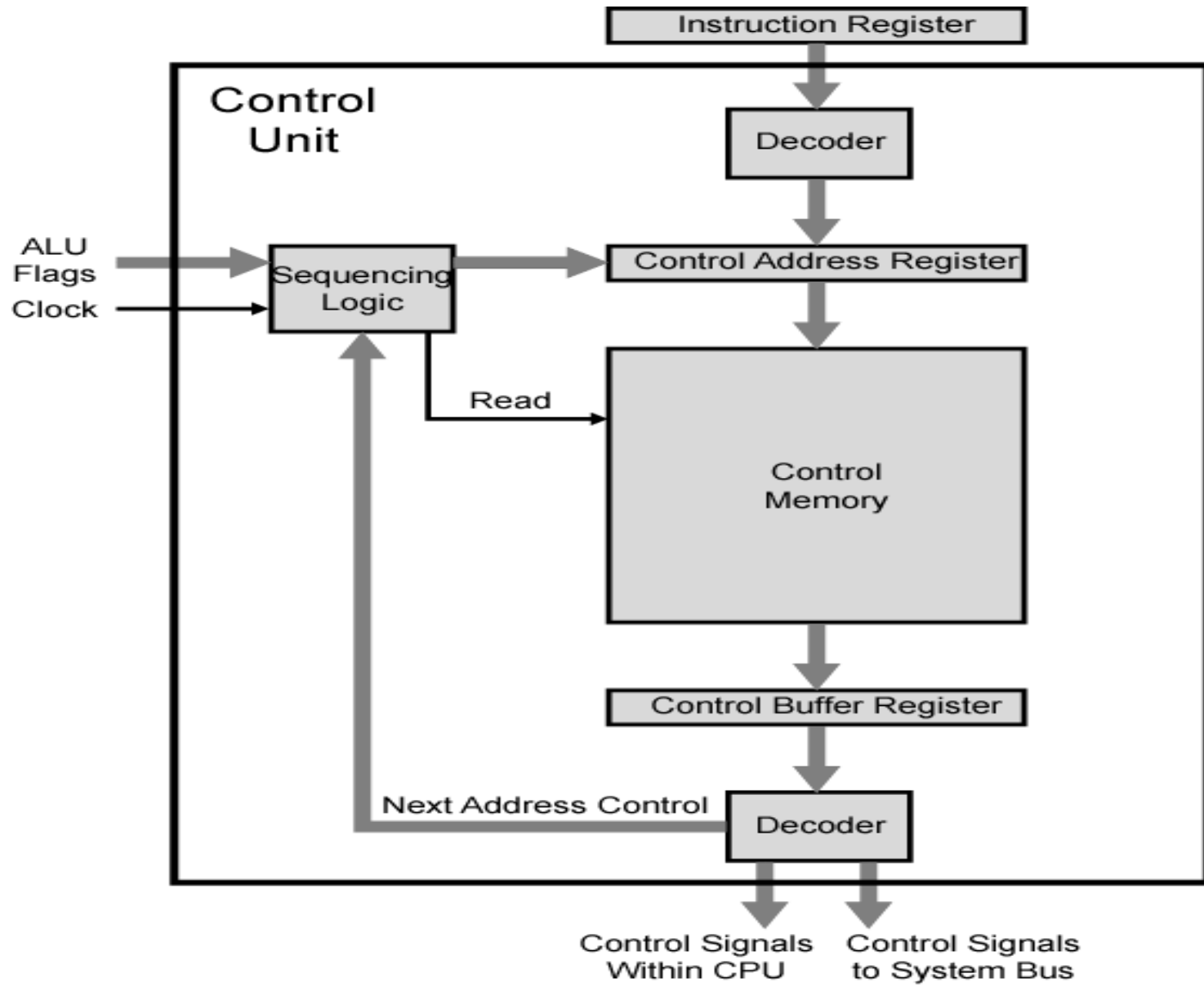
Micro programmed Control Unit Func

- Sequence logic unit issues read command

- Word specified in control address register is read into control

  buffer register

- Control buffer register contents generates control signals and next

  address information

- Sequence logic loads new address into control buffer register

  based on next address information from control buffer register

  and ALU flags

Next Address Decision

- Depending on ALU flags and control buffer register
  - Get next instruction
    - Add 1 to control address register
  - Jump to new routine based on jump microinstruction
    - Load address field of control buffer register into control address register
  - Jump to machine instruction routine
    - Load control address register based on opcode in IR

Advantages and Disadvantages of Microprogramming
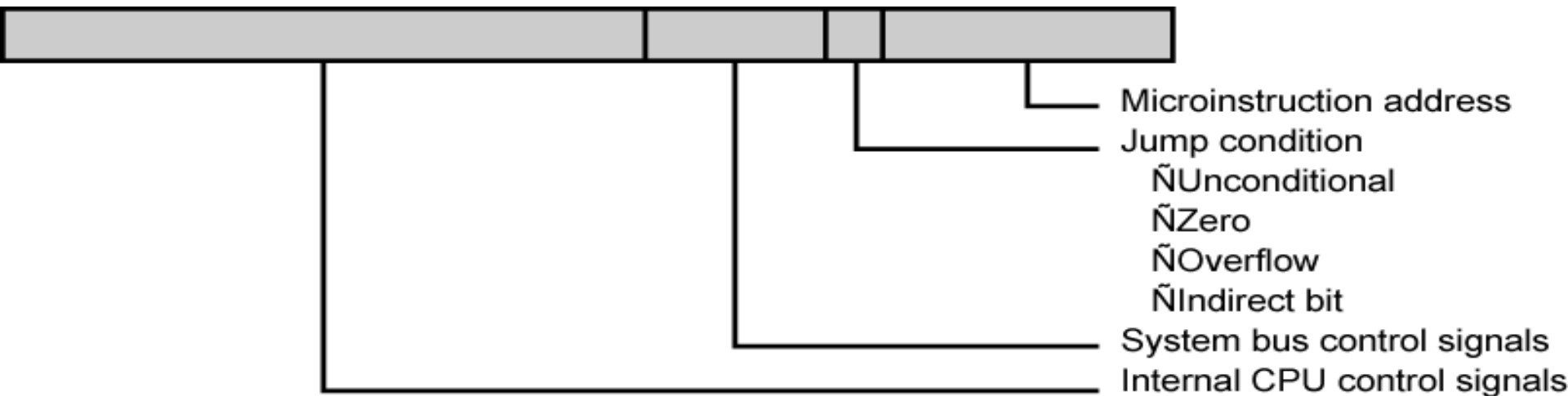
- Simplifies design of control unit
  - Cheaper
  - Less error-prone
- Slower

Tasks Done By Microprogrammed Control Unit

- Microinstruction sequencing
- Microinstruction execution
- Must consider both together

Micro-instruction Types

- Each micro-instruction specifies many **different**

  micro-operations to be performed in **parallel**

  - (*horizontal* **micro-programming**)

- Each micro-instruction specifies **single** (or few)

  micro-operations to be performed

  - (*vertical* **micro-programming**)

Typical Microinstruction Formats

(a) Horizontal microinstruction

- Microinstruction address
- Jump condition
  - ÑUnconditional
  - ÑZero
  - ÑOverflow
  - ÑIndirect bit
- System bus control signals
- Internal CPU control signals

(b) Vertical microinstruction

- Microinstruction address
- Jump condition
- } Function codes
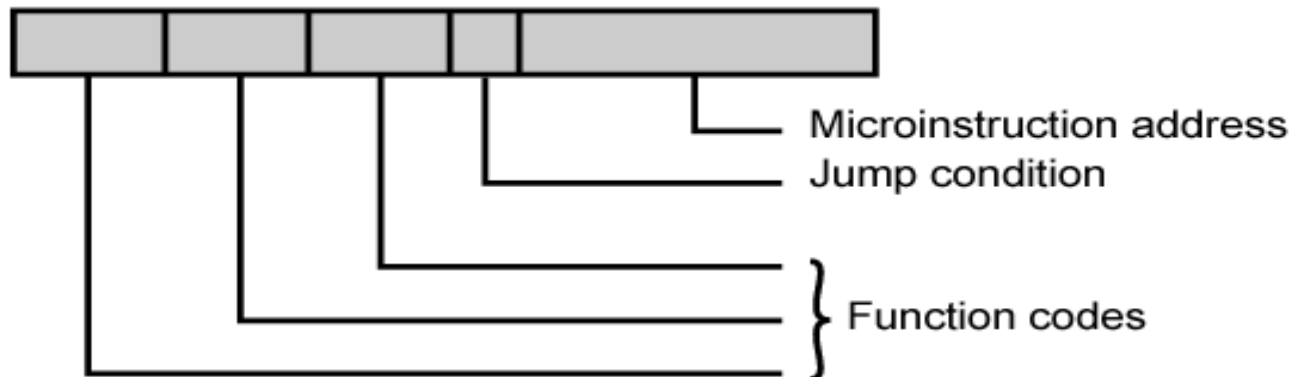
Vertical Micro-programming

- **Width** is **narrow**

- **Limited** ability to express **parallelism**

- Considerable encoding of control information requires

  external memory word decoder to identify the exact

  control line being manipulated

Horizontal Micro-programming

- **Wide** memory word

- **High degree of parallel** operations possible

- Little encoding of control information

William Stallings
Computer Organization
and Architecture
Chapter 13
7ᵗʰ Edition

Reduced Instruction Set Computers

v/s

Complex Instruction Set Computers

Introduction

- **THE ARCHITECTURAL DESIGN** of the **CPU** is RISC & CISC.

- Hardware fused with software (**Intel v/s Apple**)

- **Intel's** hardware oriented approach →CISC while that of **Apple** is RISC

- **Instruction Set Architecture- Interface** to allow easy communication

  between the programmer and the hardware.

- ISA- execution of data, copying data, deleting it, editing

- Instruction Set , Addressing Modes,

RISC-**Reduced Instruction Set Computer**

- RISC processor design has separate **digital circuitry** in the control unit

- Signals needed for the execution of each instruction in the instruction set of the processor.

- Examples of RISC processors:
  - IBM RS6000, MC88100
  - DEC's Alpha 21064, 21164 and 21264 processors

CISC-**Complex Instruction Set Computer**

- Control unit →micro-electronic circuitry

  - generates a set of control signals →activated by a micro-code

- The primary goal of CISC architecture is to complete a task in as few lines of assembly code as possible.

- **Examples of CISC processors are:**

  – Intel 386, 486, Pentium, Pentium Pro, Pentium II, Pentium III

  – Motorola's 68000, 68020, 68040, etc.

CISC processor features

- Instruction set with 120-350 instructions

- Variable instruction/data formats

- Small set of general purpose registers(8-24)

- A large number of addressing modes

- High dependency on micro program

- Complex instructions to support HLL features

CISC processor features

- Complex pipelining

- Many functional chips needed to design a computer using CISC

- Difficult to design a superscalar processor

RISC processor features

- Instruction set with limited number of instructions

- Simple instruction format

- Large set of CPU registers

- Very few addressing modes

- Easy to construct a superscalar processor

RISC processor features

- Hardwired control unit for sequencing

  microinstructions

- Supports on chip cache memory

- All functional units on a single chip

- Simple pipelining

Example for RISC vs. CISC

Consider the the program fragments:

```
         mov ax, 0
         mov bx, 10
         mov cx, 5
CISC  mov ax, 10        RISC   Begin   add ax, bx
      mov bx, 5                        loop Begin
      mul bx, ax
```

The total clock cycles for the CISC version might be:

(2 movs × 1 cycle) + (1 mul × 30 cycles) = 32 cycles

While the clock cycles for the RISC version is:

(3 movs × 1 cycle) + (5 adds × 1 cycle) + (5 loops × 1 cycle) = 13 cycles

PIPELINING-Ex Laundry Analogy
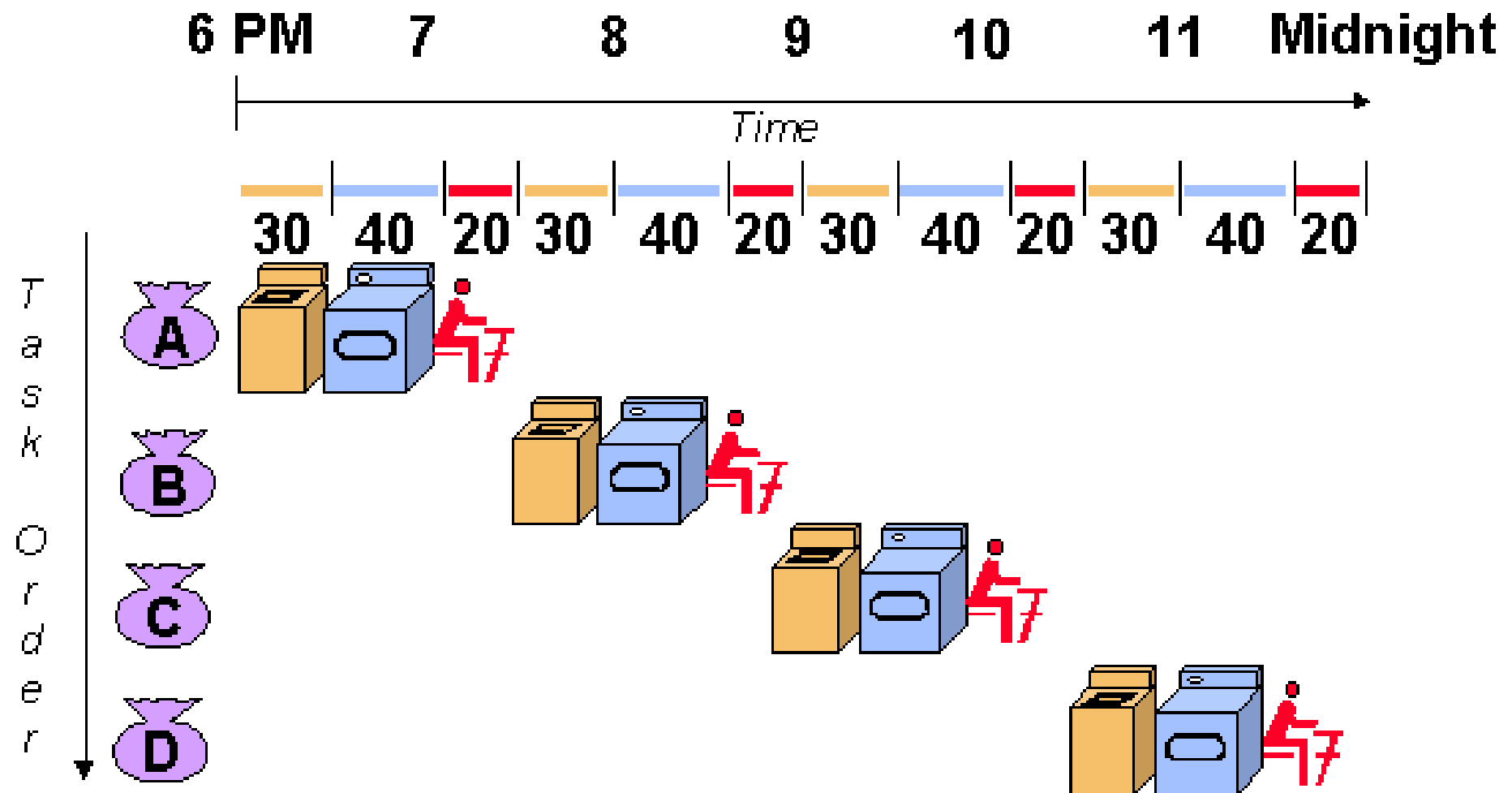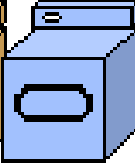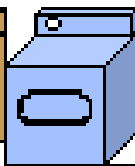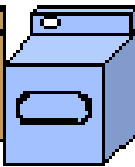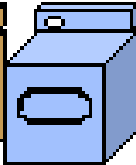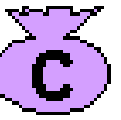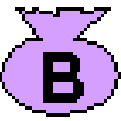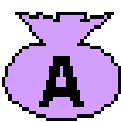
Problems with pipeline

- Stalling of pipeline
  - Data Dependency
  - Branch,etc...

RISC Pipelining

- Most instructions are **register to register**

- Two phases of execution, **I E**
  - I: Instruction fetch
  - E: Execute
    - ALU operation with register input and output

- For **load and store**(**memory**),**I E D**
  - I: Instruction fetch
  - E: Execute
    - Calculate memory address
  - D: Memory
    - Register to memory or memory to register operation

		I	E	D											
Load	rA ← M	I	E	D											
Load	rB ← M				I	E	D								
Add	rC ← rA + rB							I	E						
Store	M ← rC									I	E	D			
Branch X													I	E

(a) Sequential execution

Load	rA ← M
Load	rB ← M
Add	rC ← rA + rB
Store	M ← rC
Branch X	
NOOP	

(b) Two-stage pipelined timing

NOP is typically used to generate a delay in execution or to reserve space in code memory.

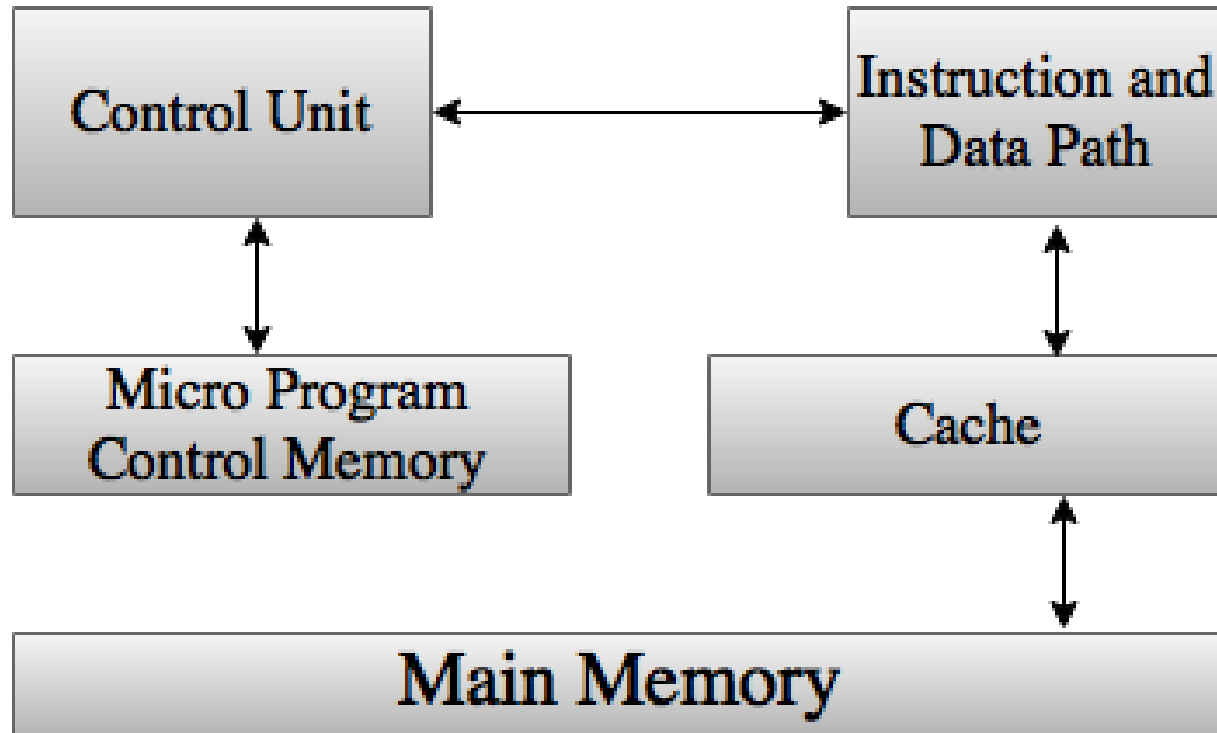| Load | rA ← M | I | E | D | | | | | |
Load	rB ← M		I	E	D				
NOOP				I	E				
Add	rC ← rA + rB				I	E			
Store	M ← rC					I	E	D	
Branch X							I	E	
NOOP								I	E

(c) Three-stage pipelined timing

RISC Architecture

- 9 **functional units** interconnected by multiple data paths with width ranging from 32-128 bits

- All internal- external buses are 32 bit wide

- Separate instruction (4KB)and data cache(8KB)

- **MMU-** implements paged virtual memory structure

- **RISC integer unit** executes load,store,fetch etc

- 2 floating point units , multiplier unit and adder unit

- **Graphics unit** to support 3D drawing

CISC Architecture

Fig. CISC Architecture