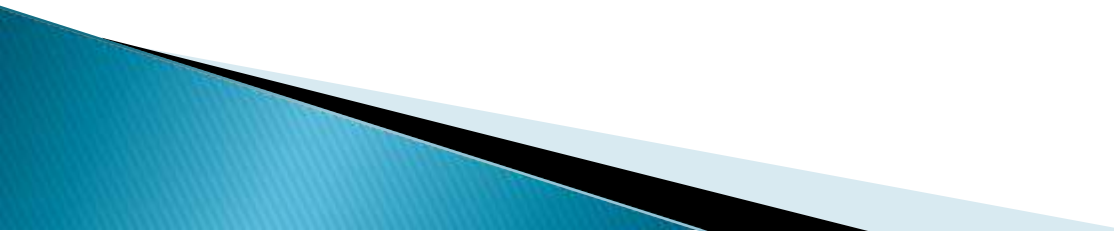


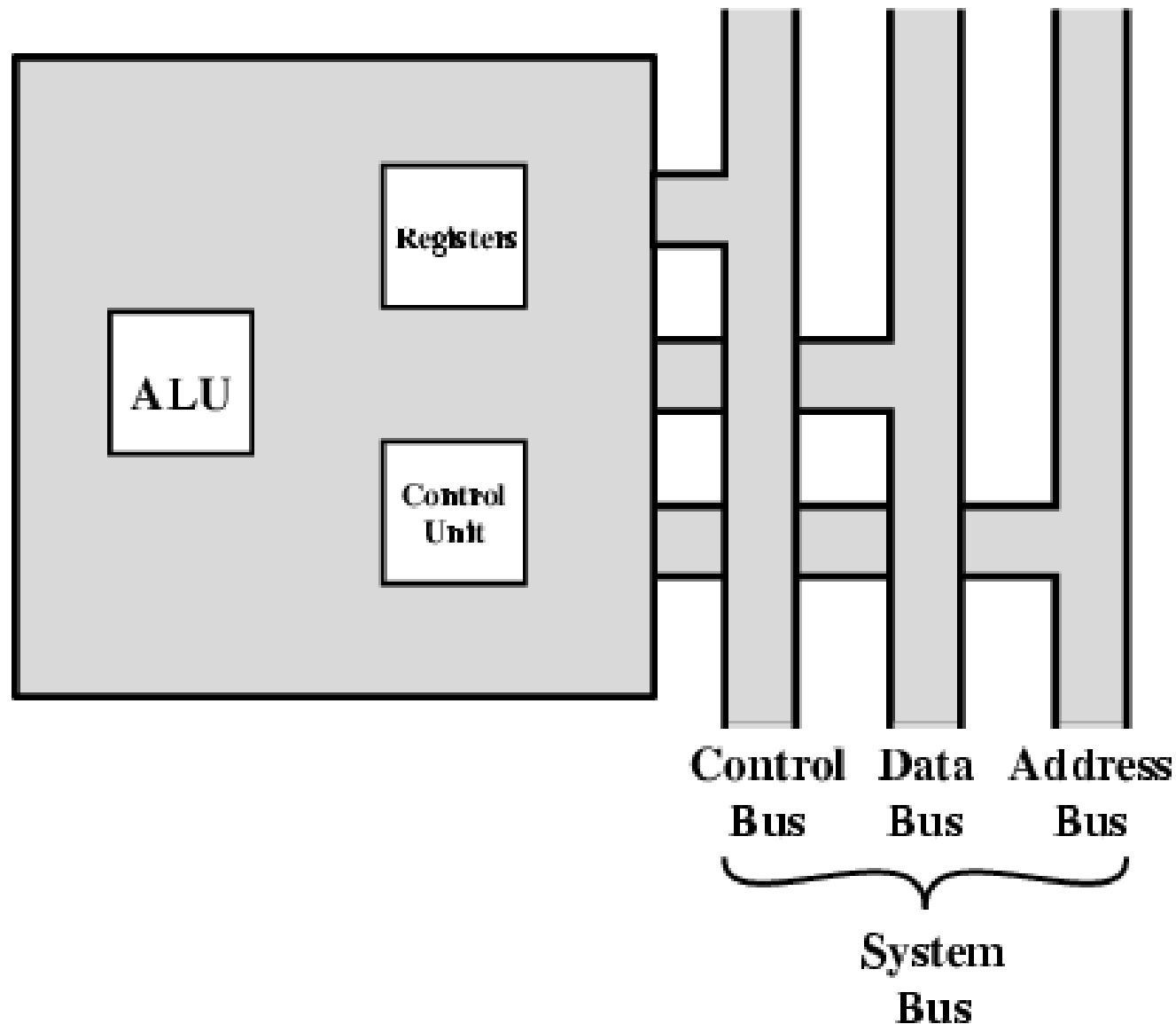
# Central Processing Unit

3	Central Processing Unit		10	CO 2
	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 pipelining		
		Self learning: RISC and CISC Architecture, Case study on SPARC		

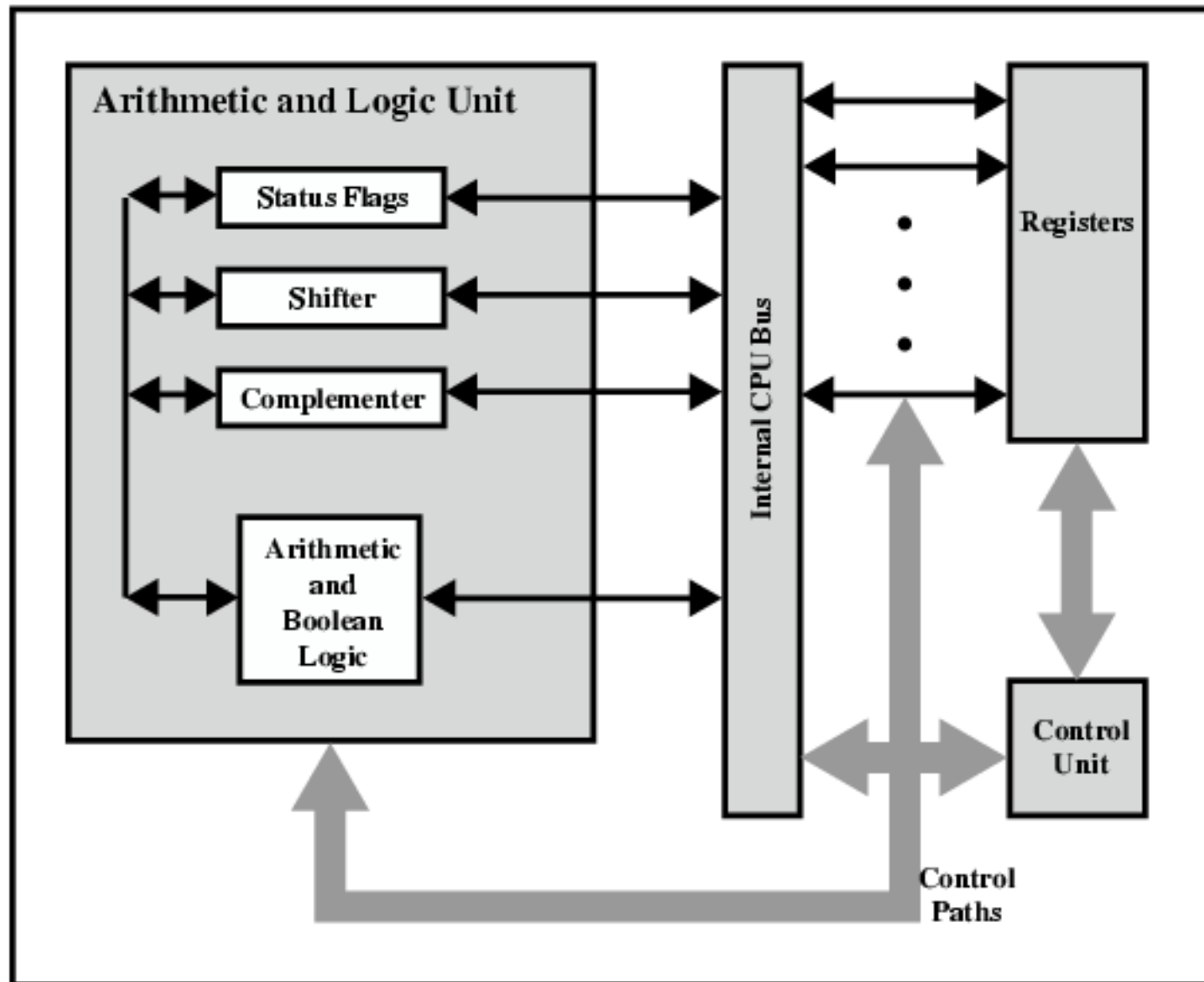
# CPU Structure

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

# CPU With Systems Bus

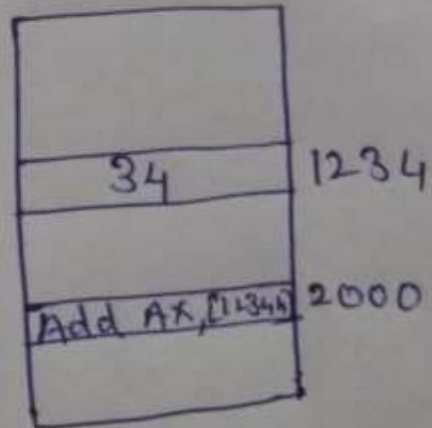


# CPU Internal Structure



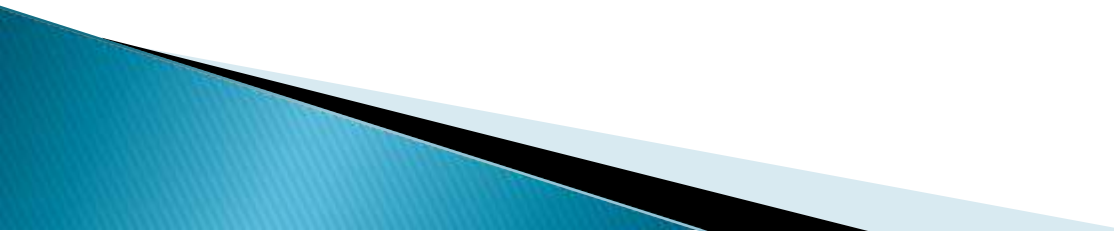
Add AX, [1234h]

Add [1234h], AX



memory (code + data)

# 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

# User Visible Registers

File Edit View Run Break Options Window Help

execute the assembly program using TASM

cs:0000 B8AE48	mov	ax,48AE	ax 2887	c=1
cs:0003 8ED8	mov	ds,ax	bx 9DC2	z=0
cs:0005 B90000	mov	cx,0000	cx 0001	s=0
cs:0008 A10E00	mov	ax,[000E]	dx 0000	o=0
cs:000B 8B1E1000	mov	bx,[0010]	si 0000	p=0
cs:000F 03C3	add	ax,bx	di 0000	a=0
cs:0011 7301	jnb	0014	bp 0000	i=1
cs:0013 41	inc	cx	sp 0000	d=0
cs:0014 A31200	mov	[0012],ax	ds 48AE	
cs:0017 890E1400	mov	[0014],cx	es 489D	
cs:001B CD21	int	21	ss 48AC	
			cs 48AD	
			ip 001B	

[ ]=Dump 2=[↑][↓]

ds:0000 C3 73 01 41 A3 12 00 89	ds:0008 0E 14 00 CD 21 00 C5 8A	ds:0010 C2 9D 87 28 01 00 00 00	ds:0018 00 00 00 00 00 00 00 00
es:0018 01 03 01 00 02 FF FF FF			

ss:0002 6474  
ss:0000 0000



# 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

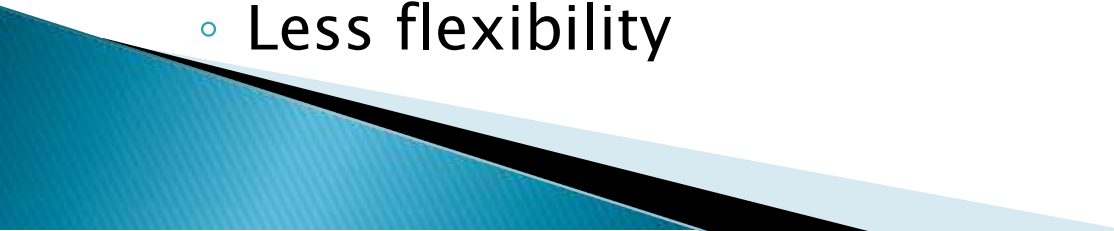
FLAGS Register
Instruction Pointer

(c) 80386 - Pentium II

## General Purpose Registers (1)

- ▶ May be true general purpose
- ▶ May be restricted
- ▶ May be used for data or addressing
- ▶ Data
  - Accumulator
- ▶ Addressing
  - Segment

## General Purpose Registers (2)

- ▶ Why make them general purpose?
    - Increase flexibility and programmer options
    - Increase instruction size & complexity
  - ▶ Make them specialized
    - Smaller (faster) instructions
    - Less flexibility
- 

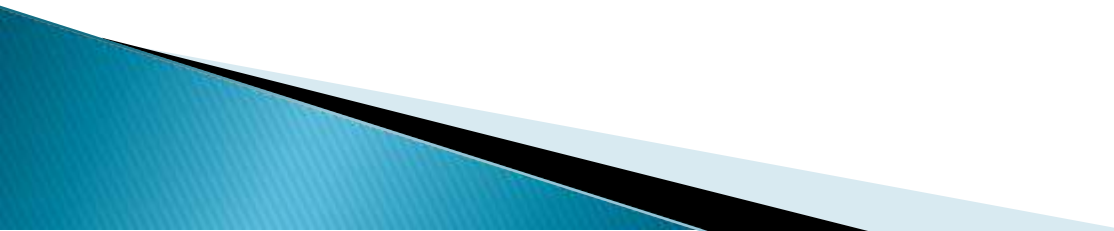
# How big?

- ▶ Large enough to hold full address
- ▶ Large enough to hold full word
- ▶ Often possible to combine two data registers
  - C programming
  - `long int a;`

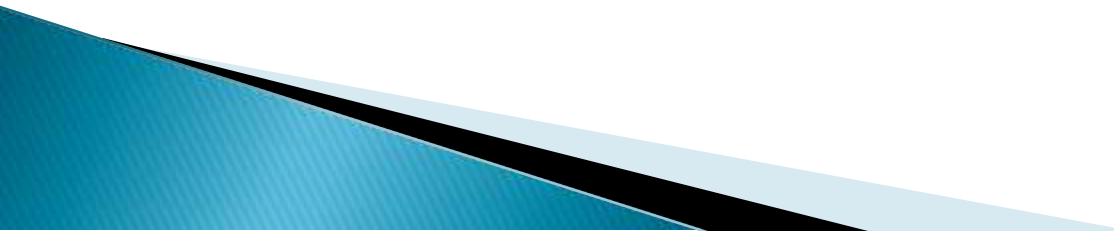
# Condition Code Registers

- ▶ 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

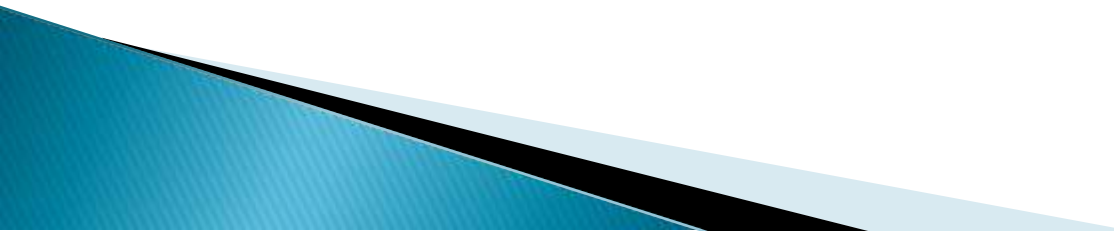
# Control & Status Registers

- ▶ Program Counter (PC)
  - ▶ Instruction Register(IR)
  - ▶ Memory Address Register(MAR)
  - ▶ Memory Buffer Register(MBR)
- 

# Program Status Word

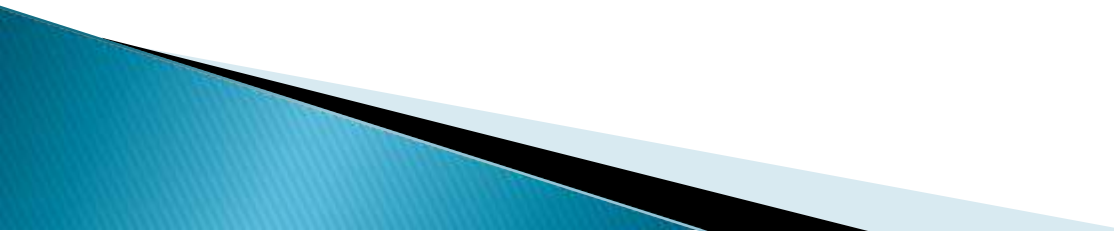
- ▶ A set of bits
  - ▶ Includes Condition Codes
  - ▶ Sign of last result
  - ▶ Zero
  - ▶ Carry
  - ▶ Equal
  - ▶ Overflow
  - ▶ Interrupt enable/disable
  - ▶ Supervisor
- 

# 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 stores the offset address of the next instruction 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 offset value within the program stack. 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 helps in referencing the parameter variables passed to a subroutine. 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.
- 

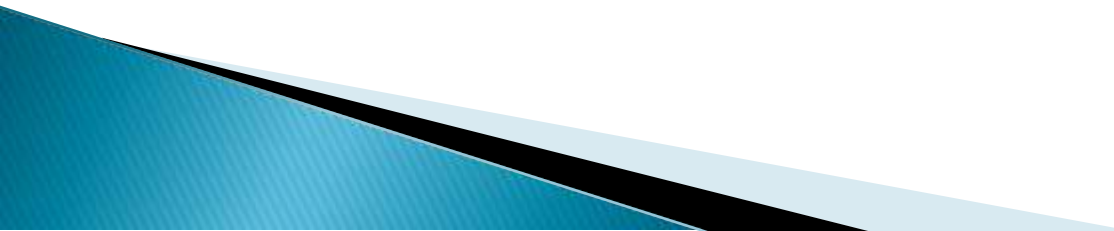
# Index Registers

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

There are two sets of index pointers –

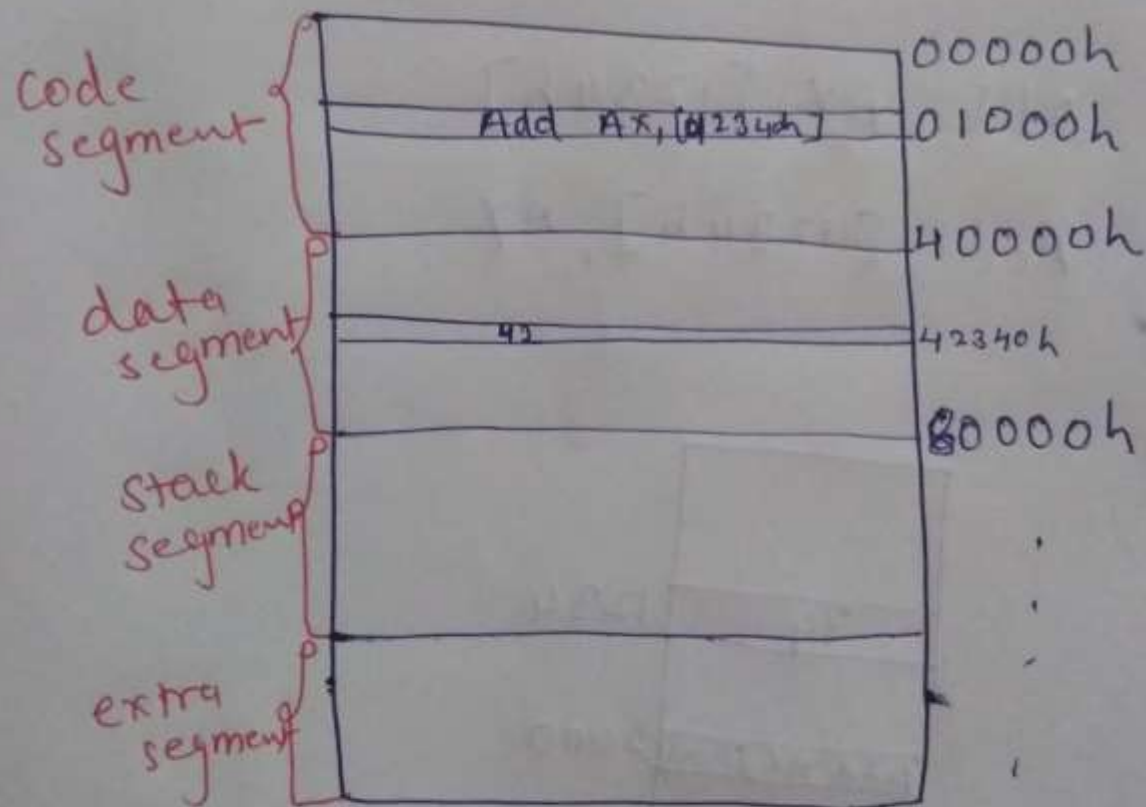
**Source Index (SI)** – It is used as source index for string operations.

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



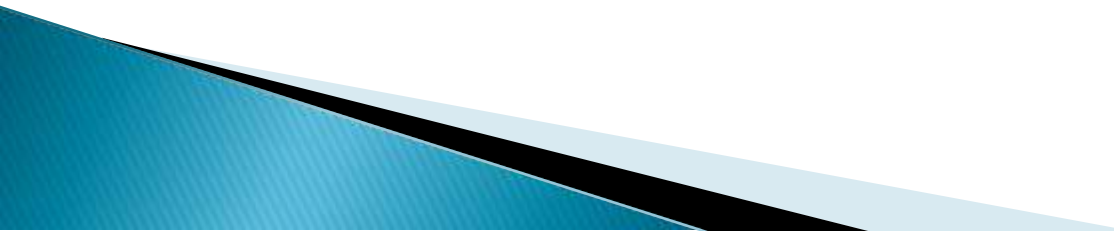
CS 0000

IP 1000h



Memory  
(instr + data)

# 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

- Flag Register (status register)
  - 16-bit register
  - Conditional flags: CF, PF, AF, ZF, SF, OF
  - Control flags: TF, IF, DF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

R = reserved

U = undefined

OF = overflow flag

DF = direction flag

IF = interrupt flag

TF = trap flag

SF = sign flag

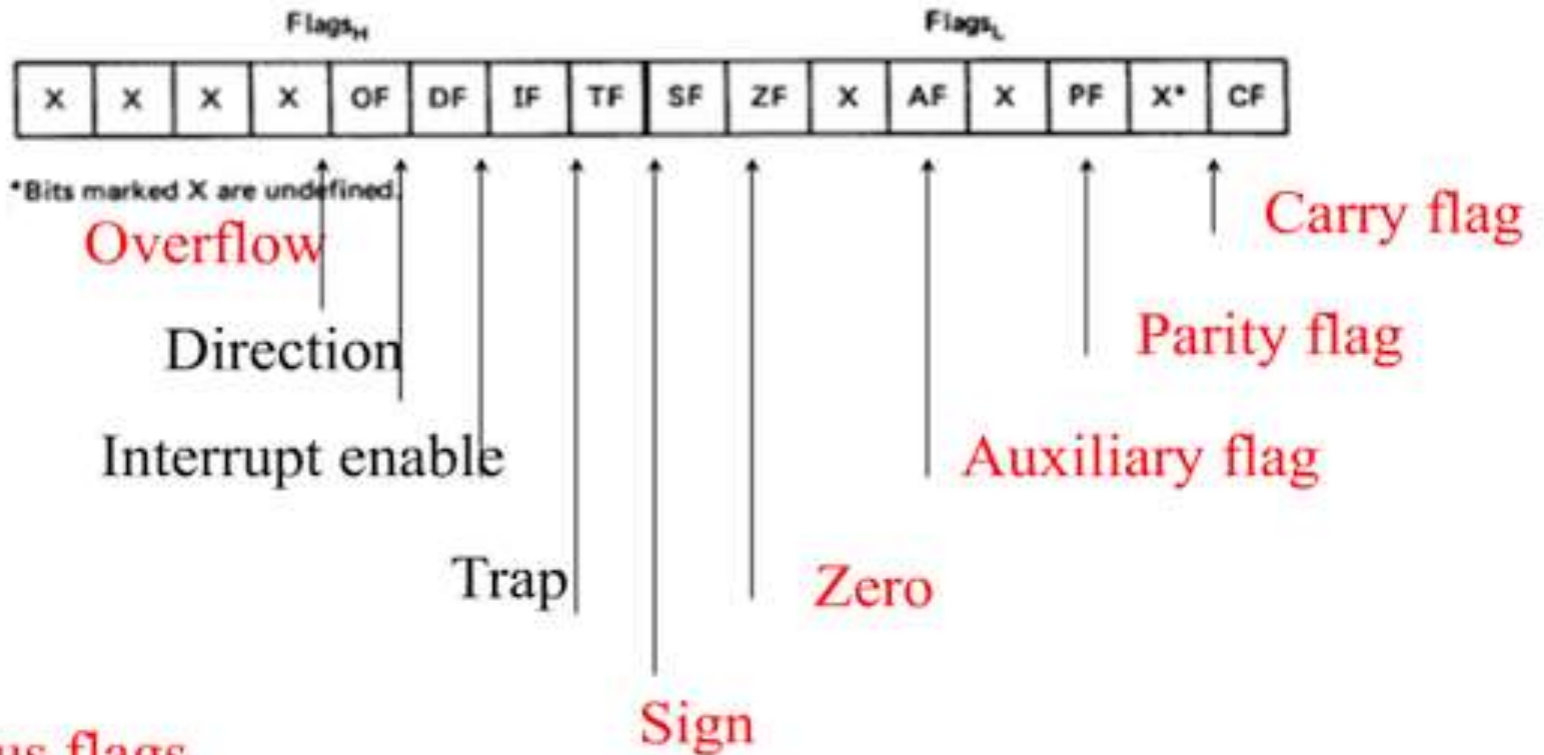
ZF = zero flag

AF = auxiliary carry flag

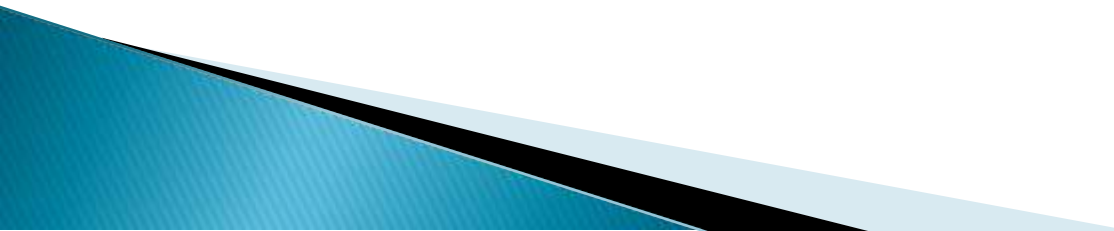
PF = parity flag

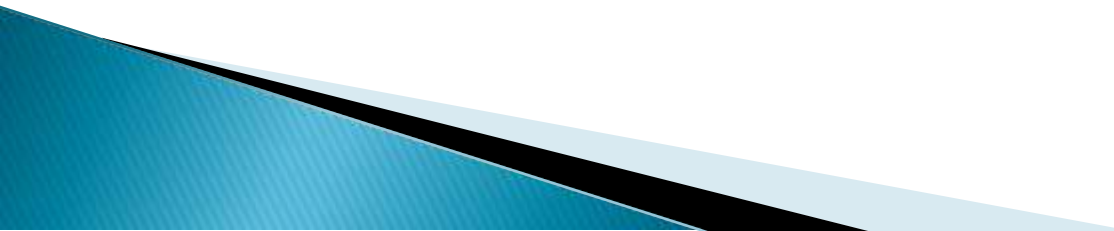
CF = carry flag

# Flags



6 are status flags  
3 are control flag

- ▶ **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.

# Segment Registers

## Addressing Modes

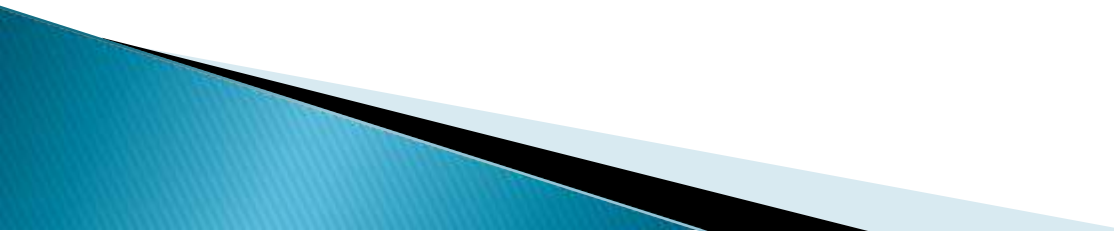
Pointer Register(offset)	Default segment register
BX	DS
SI	DS
DI	DS(ES for string)
SP	SS
BP	SS
IP	CS

# Addressing Modes

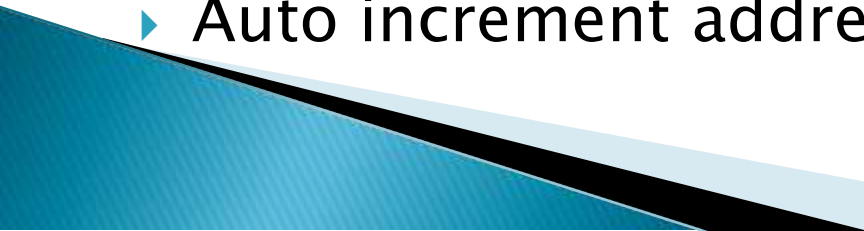
# **Addressing Modes**



# 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.
- 

# Addressing Modes

- ▶ Immediate addressing mode
  - ▶ Register addressing mode
  - ▶ Memory addressing mode
    - Direct memory addressing mode
    - Register Indirect addressing mode
    - Based addressing mode
    - Indexed addressing mode
    - Based Indexed addressing mode
    - Based Indexed with displacement addressing mode
  - ▶ String addressing mode
  - ▶ Auto increment addressing mode
- 

# Addressing Modes

Pointer Register(offset)	Default segment register
BX	DS
SI	DS
DI	DS(ES for string)
SP	SS
BP	SS
IP	CS



# Immediate Addressing

- ▶ Operand is part of instruction
- ▶ Operand = address field
- ▶ e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand

Mov CL,#30

- ▶ No memory reference to fetch data
- ▶ Fast
- ▶ Limited range

# Immediate Addressing Diagram

Instruction



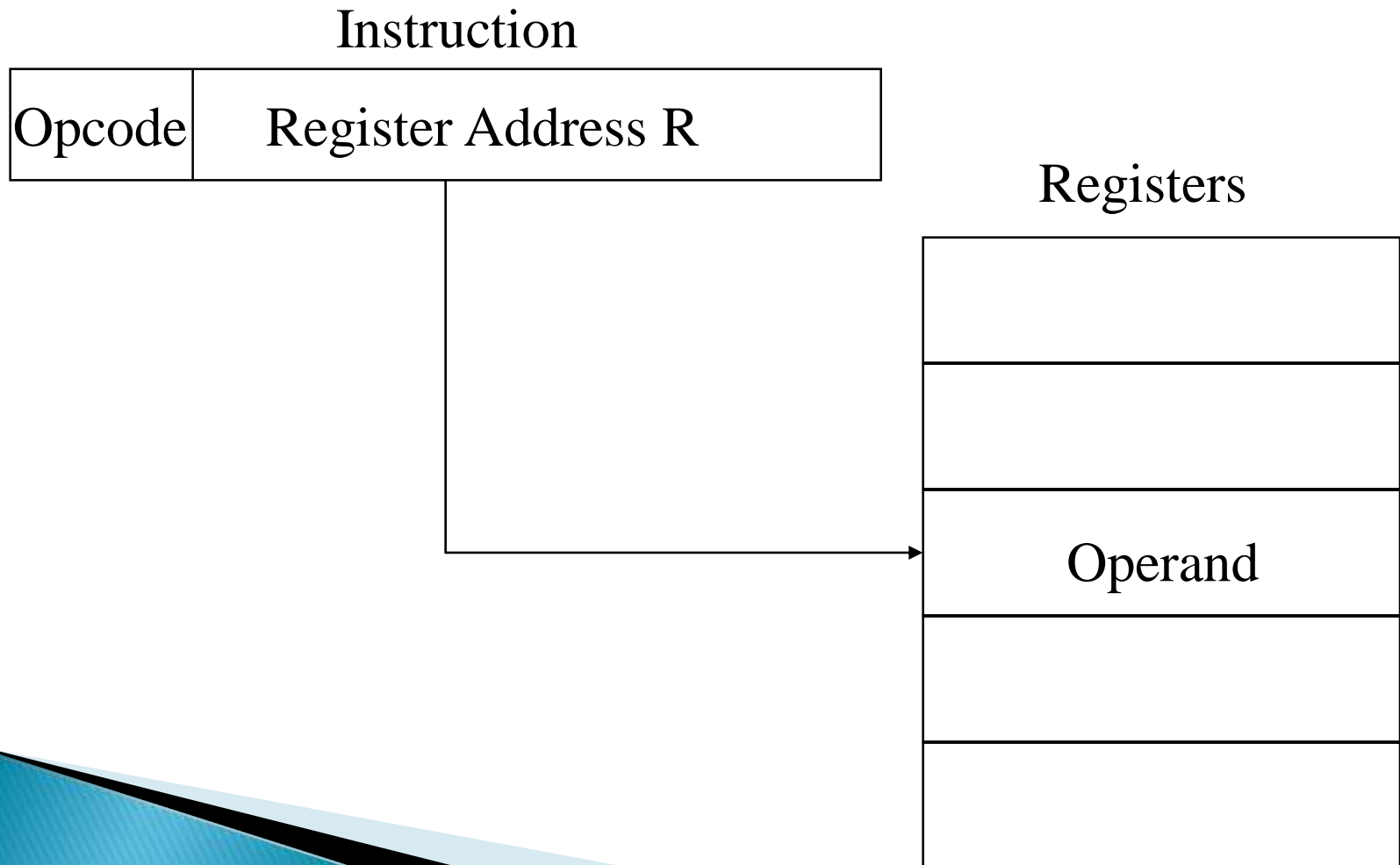
# Register Addressing (1)

- ▶ Operand is held in register named in address field
- ▶  $EA = R$
- ▶ Limited number of registers
- ▶ Very small address field needed
  - Shorter instructions
  - Faster instruction fetch

# Register Addressing (2)

- ▶ No memory access
- ▶ Very fast execution
- ▶ Very limited address space
- ▶ Multiple registers helps performance
  - Requires good assembly programming or compiler writing
- ▶ e.g. `MOV AX,BX`
- ▶ `ADD AX,BX`

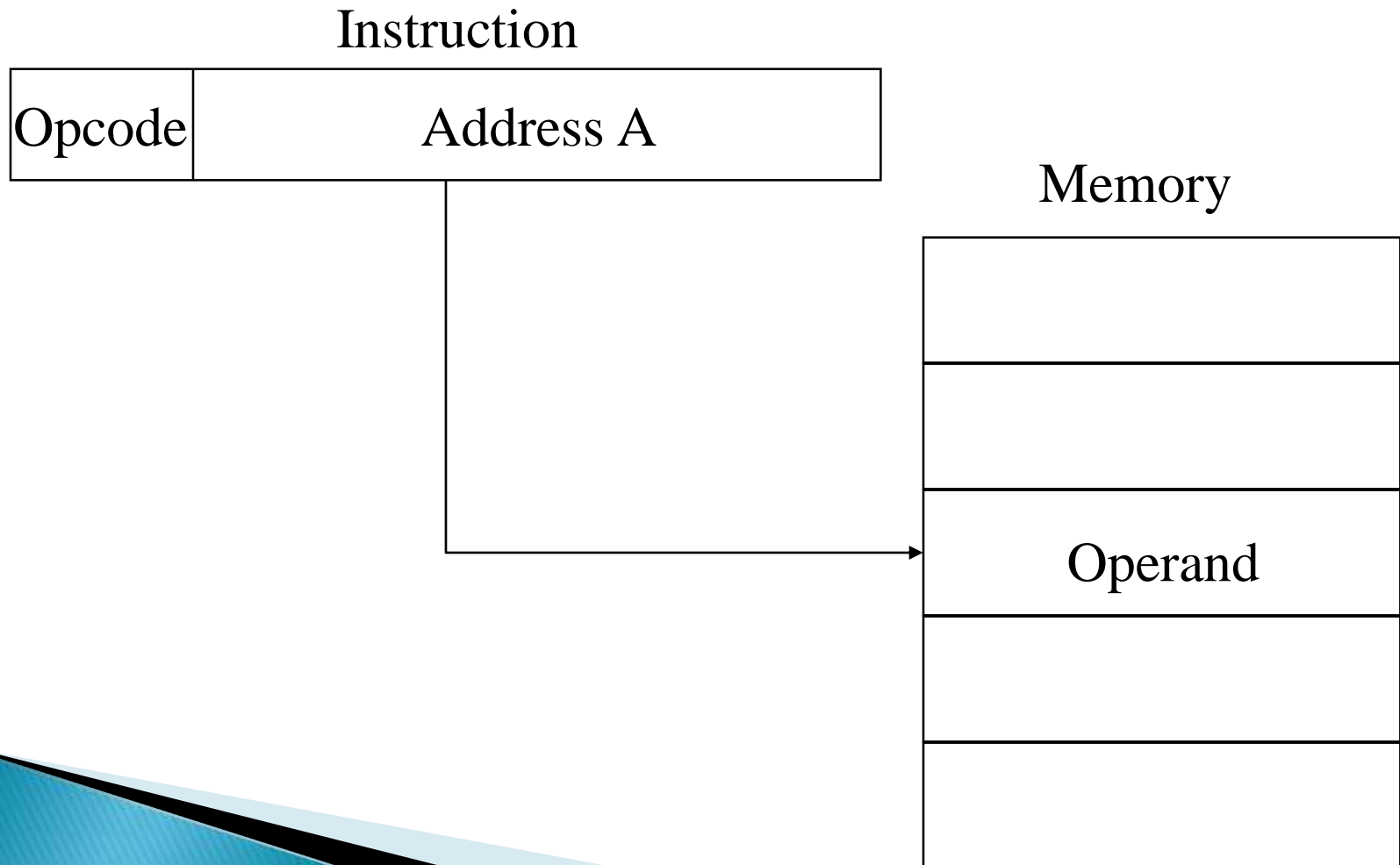
# Register Addressing Diagram



# Memory: Direct Addressing

- ▶ Address field contains address of operand
- ▶ Effective address (EA) = address field (A)
- ▶  $PA = \text{segment:EA}$
- ▶ e.g. `ADD A`
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- ▶ `ADD AX,[1234h]`
- ▶ Single memory reference to access data
- ▶ No additional calculations to work out effective address
- ▶ Limited address space

# Direct Addressing Diagram

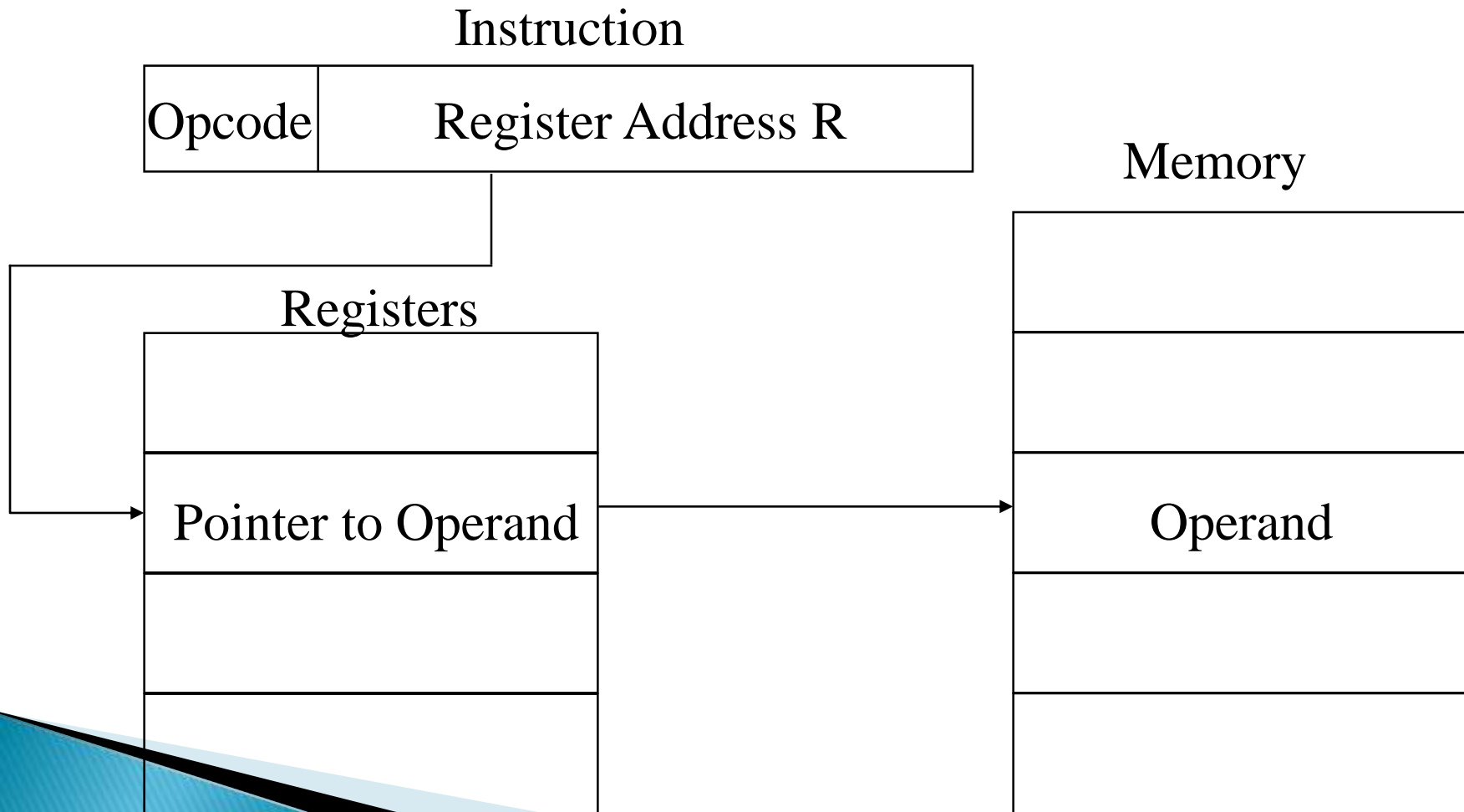


# Memory: Register Indirect Addressing

- ▶ Memory cell pointed to by address field contains the address of (pointer to) the operand
- ▶  $EA = \{BX, BP, SI, DI\}$
- ▶  $PA = \text{segment} : EA$
- ▶ e.g. `MOV [SI], AL`
- ▶ `ADD AL, [SI]`



# Register Indirect Addressing Diagram



# Memory: Based Addressing Mode

- Same as register indirect ,difference is an 8 or 16 bit displacement may be included in the operand field.
- $PA = \text{segment}:\{BX, BP\}$ 
  - E.g. `mov AL,[BX+5]`

# Memory: Indexed Addressing Mode

- Like based addressing ,indexed addressing allows the use of signed displacement.
- Index registers SI or DI must be used in the operand field.
- PA=segment:{SI/DI}
- E.g. `mov AL,[SI + 5]`

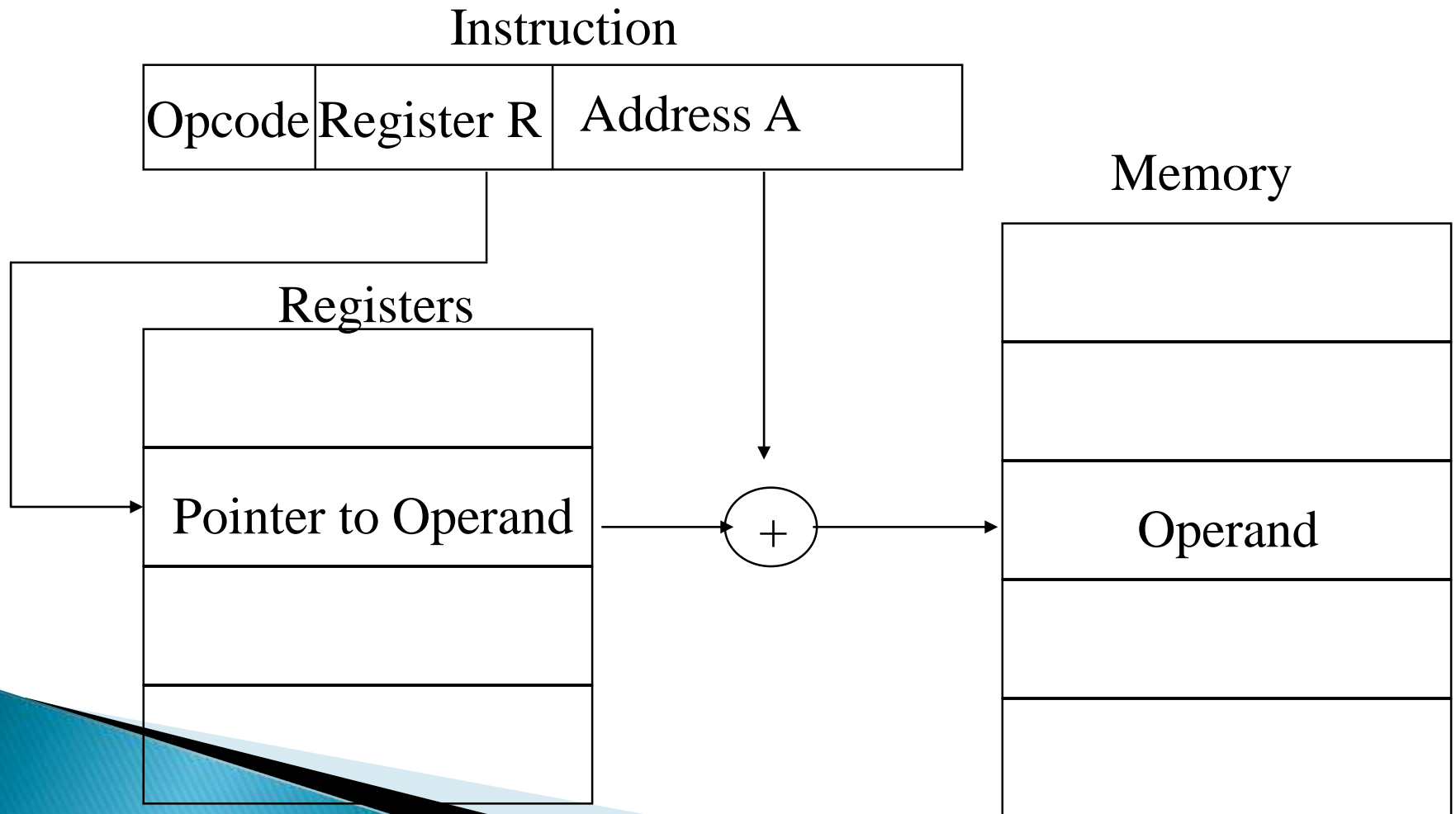
# Memory: Based Indexed Addressing Mode

- Contains the features of based and indexed addressing but does not allow use of displacement.
- $PA = \text{segment} : \{BX/BP\} + \{SI/DI\}$
- E.g. `ADD AX,[BX][SI]`

# Memory: Based Indexed with displacement Addressing Mode

- The EA of the operand is found by adding the contents of the base register and index register with 8 or 16 bit displacement.
- $PA = \text{segment} : \{BX/BP\} + \{SI/DI\} + \{8/16 \text{ bit displacement}\}$
- E.g. `ADD AX,[BX][SI + 20]`

# Displacement Addressing Diagram



# String Addressing

- ▶ The instructions that handle strings do not use any of the standard addressing modes.
- ▶ E.g MOVSB
- ▶ In this the processor knows that SI points to the source string in the data segment and DI points to the first string of the destination string in the extra segment.

# Stack Addressing/Auto increment addressing

- ▶ Operand is (implicitly) on top of stack
- ▶ After every PUSH operation , SP is decremented by two.
- ▶ Every POP operation, the stack pointer is incremented by 2.
- ▶ e.g.
  - ADD      Pop top two items from stack and add



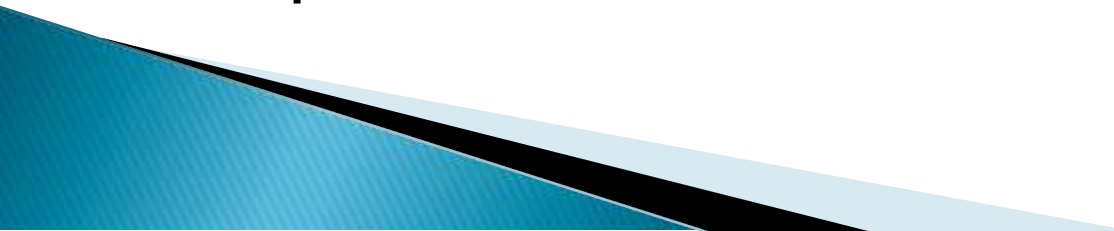
# x86 Addressing Modes

- ▶ Virtual or effective address is offset into segment
  - Starting address plus offset gives linear address
  - This goes through page translation if paging enabled
- ▶ 12 addressing modes available
  - Immediate
  - Register operand
  - Displacement
  - Base
  - Base with displacement
  - Scaled index with displacement
  - Base with index and displacement
  - Base scaled index with displacement
  - Relative

# Instruction Cycle



# Fetch Cycle

- ▶ Program Counter (PC) holds address of next instruction to fetch
  - ▶ 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 Cycle

T1:  $MAR \leftarrow (PC)$

T2:  $MBR \leftarrow \text{Memory}$

T3:  $PC \leftarrow (PC) + I$

$IR \leftarrow (MBR)$

# 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

# Execute Cycle

- ▶ Processor-memory
  - data transfer between CPU and main memory
- ▶ Processor I/O
  - Data transfer between CPU and I/O module
- ▶ Data processing
  - Some arithmetic or logical operation on data
- ▶ Control
  - Alteration of sequence of operations
  - e.g. jump
- ▶ Combination of above

# Execute Cycle

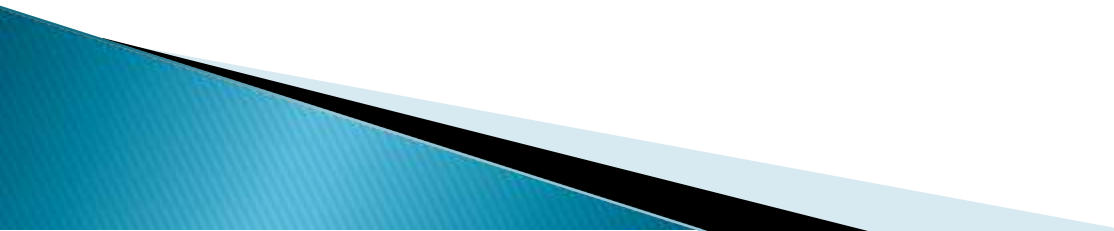
ADD R1,x

T1:  $MAR \leftarrow (IR(address))$

T2:  $MBR \leftarrow Memory$

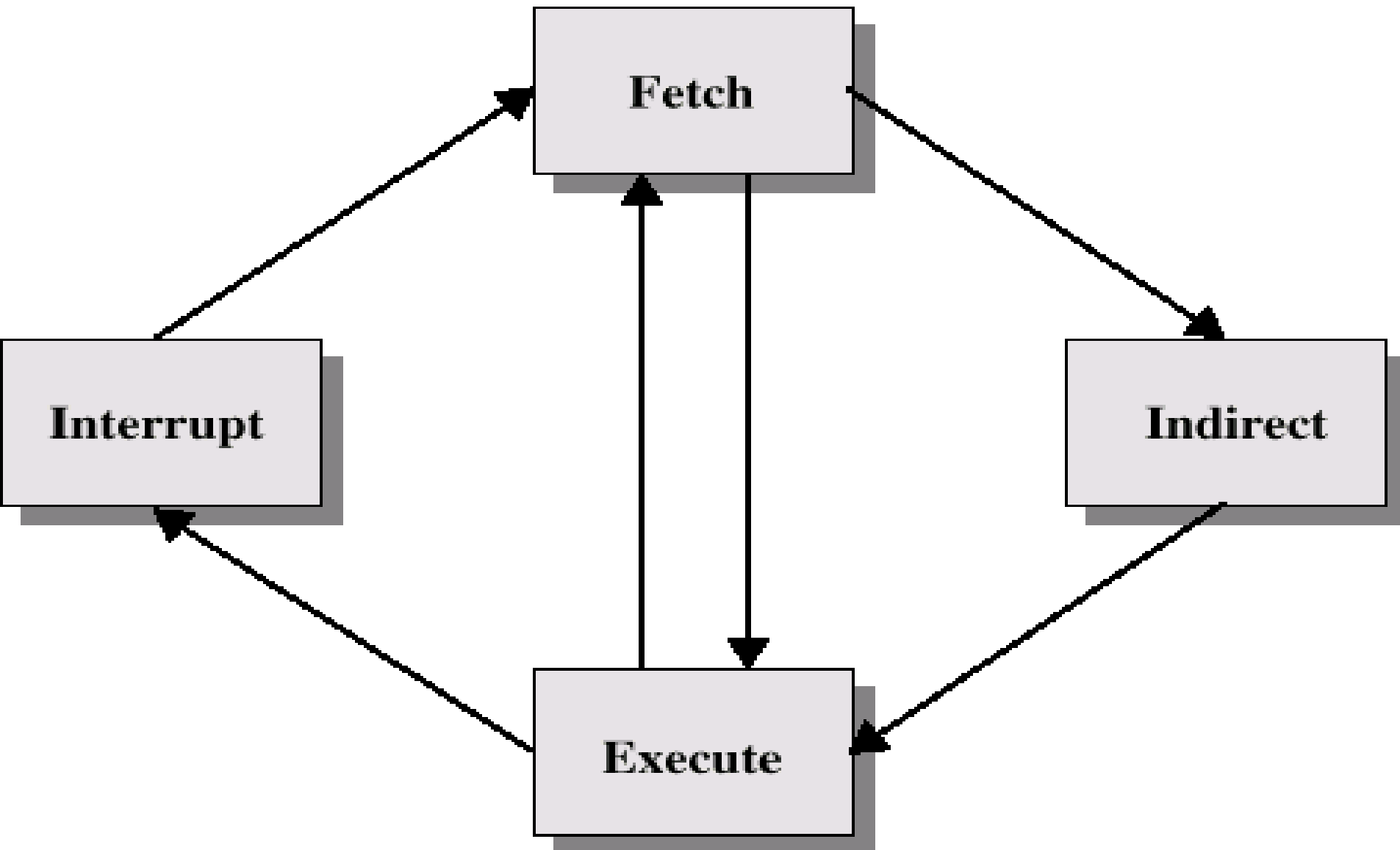
T3:  $R1 \leftarrow (R1) + (MBR)$

# Data Flow (Execute)

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



# Instruction Cycle with Indirect



# Indirect Cycle

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

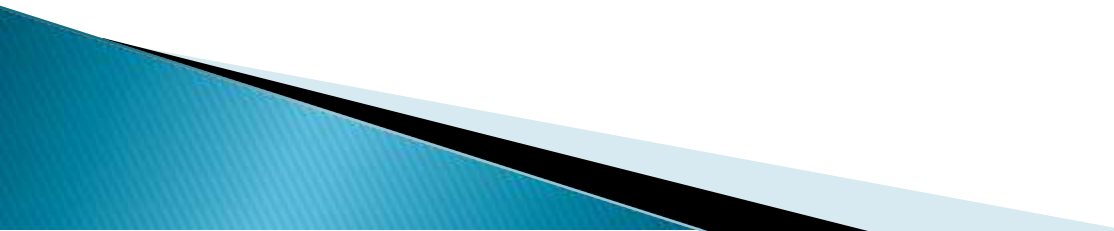
# Indirect Cycle

T1:  $MAR \leftarrow (IR(address))$

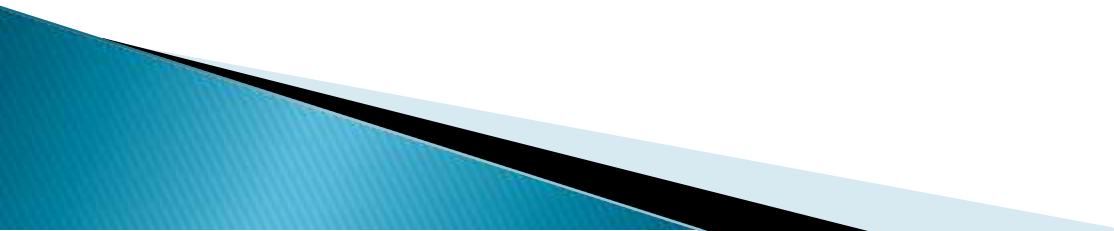
T2:  $MBR \leftarrow Memory$

T3:  $IR(Address) \leftarrow (MBR(Address))$

# Interrupts

- ▶ Mechanism by which other modules (e.g. I/O) may interrupt normal sequence of processing
  - ▶ Program
    - e.g. overflow, division by zero
  - ▶ Timer
    - Generated by internal processor timer
    - Used in pre-emptive multi-tasking
  - ▶ I/O
    - from I/O controller
  - ▶ Hardware failure
    - e.g. memory parity error
- 

# Interrupt Cycle

- ▶ Added to instruction cycle
  - ▶ Processor checks for interrupt
    - Indicated by an interrupt signal
  - ▶ If no interrupt, fetch next instruction
  - ▶ If interrupt pending:
    - Suspend execution of current program
    - Save context
    - Set PC to start address of interrupt handler routine
    - Process interrupt
    - Restore context and continue interrupted program
- 

# Interrupt Cycle

T1:  $MBR \leftarrow (PC)$

T2:  $MAR \leftarrow \text{Save\_Address}$

$PC \leftarrow \text{Routine\_Address}$

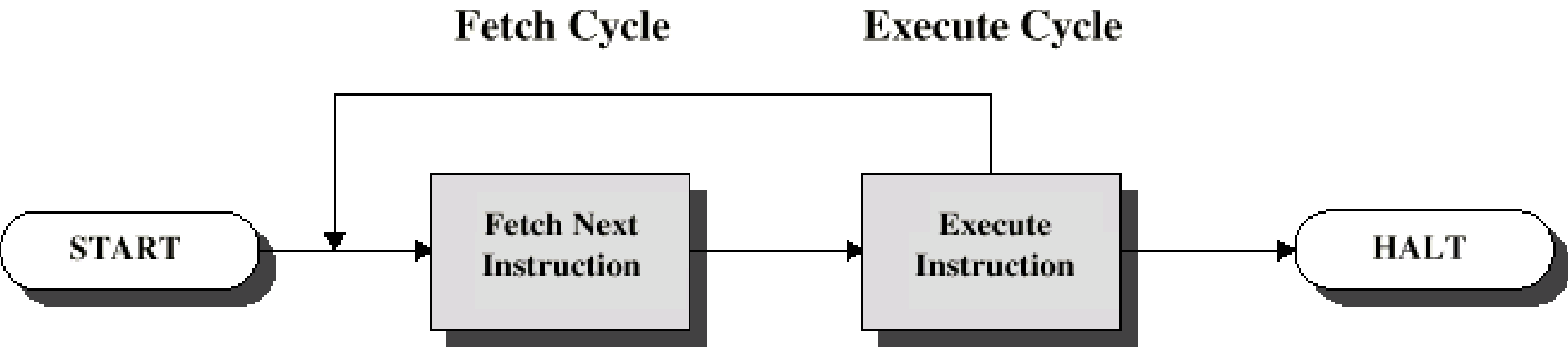
T3:  $\text{Memory} \leftarrow (MBR)$

# 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

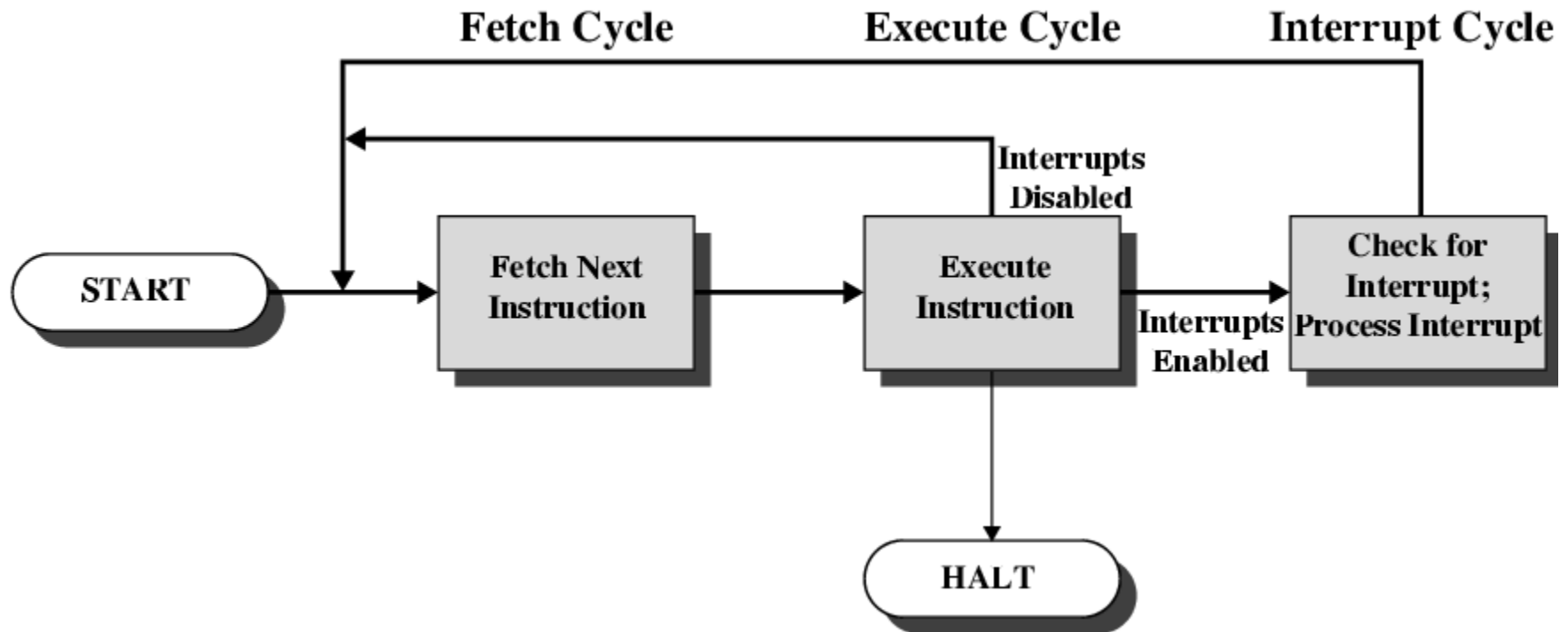
# Instruction Cycle

- ▶ Two steps:
  - Fetch
  - Execute

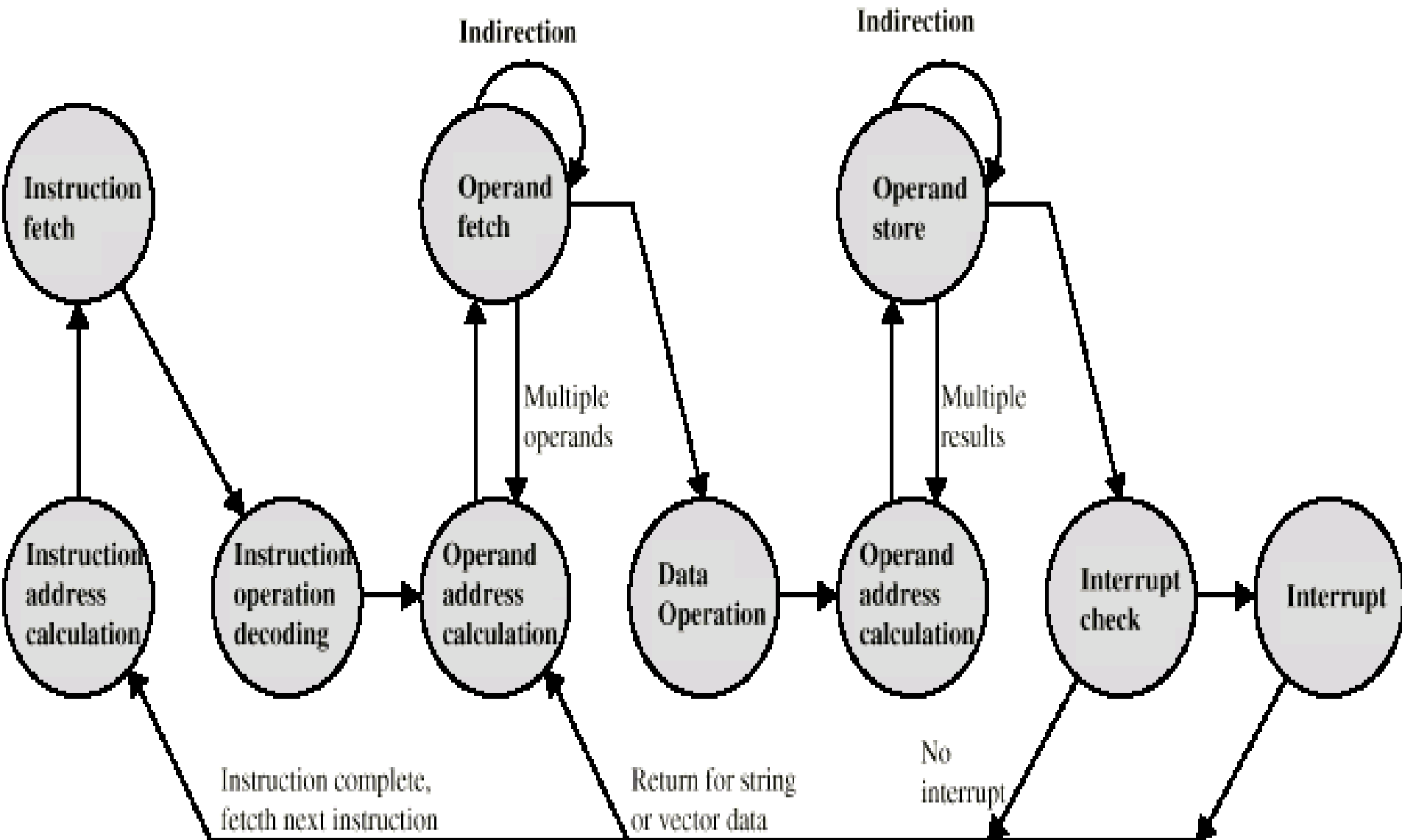




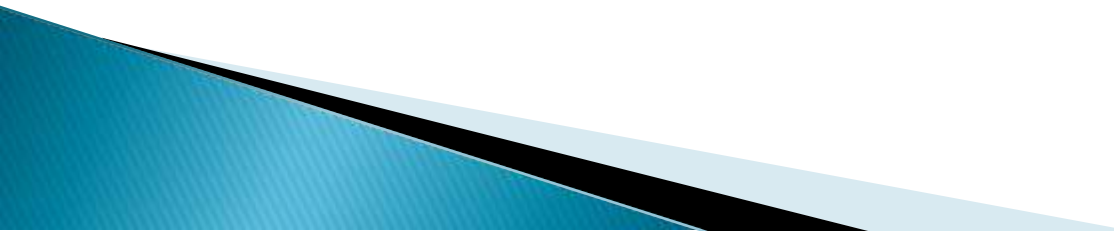
# Instruction Cycle with Interrupts



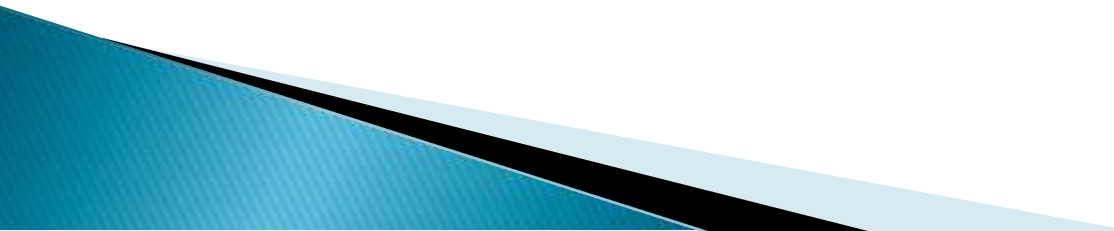
# Instruction Cycle State Diagram



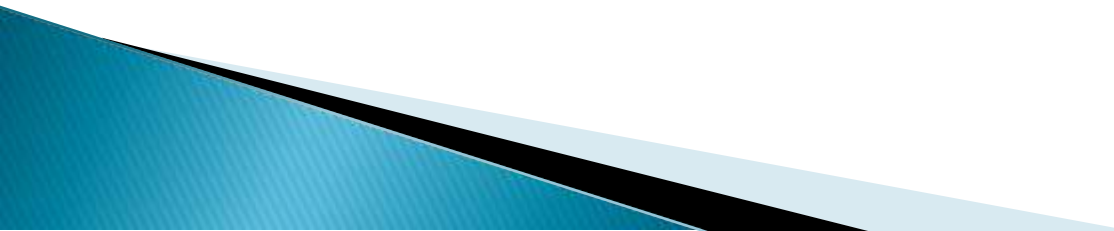
# Instruction:

- ▶ An instruction consist of an op-code and one or more operands.
  - ▶ These operands could be explicitly specified in an instruction or they could be assumed.
  - ▶ An instruction format is used to define the layout of the bits allocated to these elements of instructions.
  - ▶ The instruction format indicates the addressing modes used for each operand in that instruction.
- 

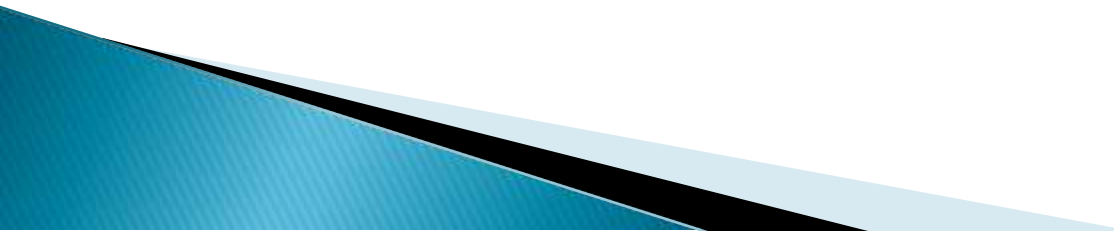
# Factors for Instruction Length

- ▶ Affected by and affects:
    - Memory size(larger memory more bits)
    - Memory organization(Virtual memory needs more memory range)
    - Bus structure(instruction should be of bus length or multiple of bus length)
    - CPU speed(data transfer rate of memory should be equal to the processor speed)
- 

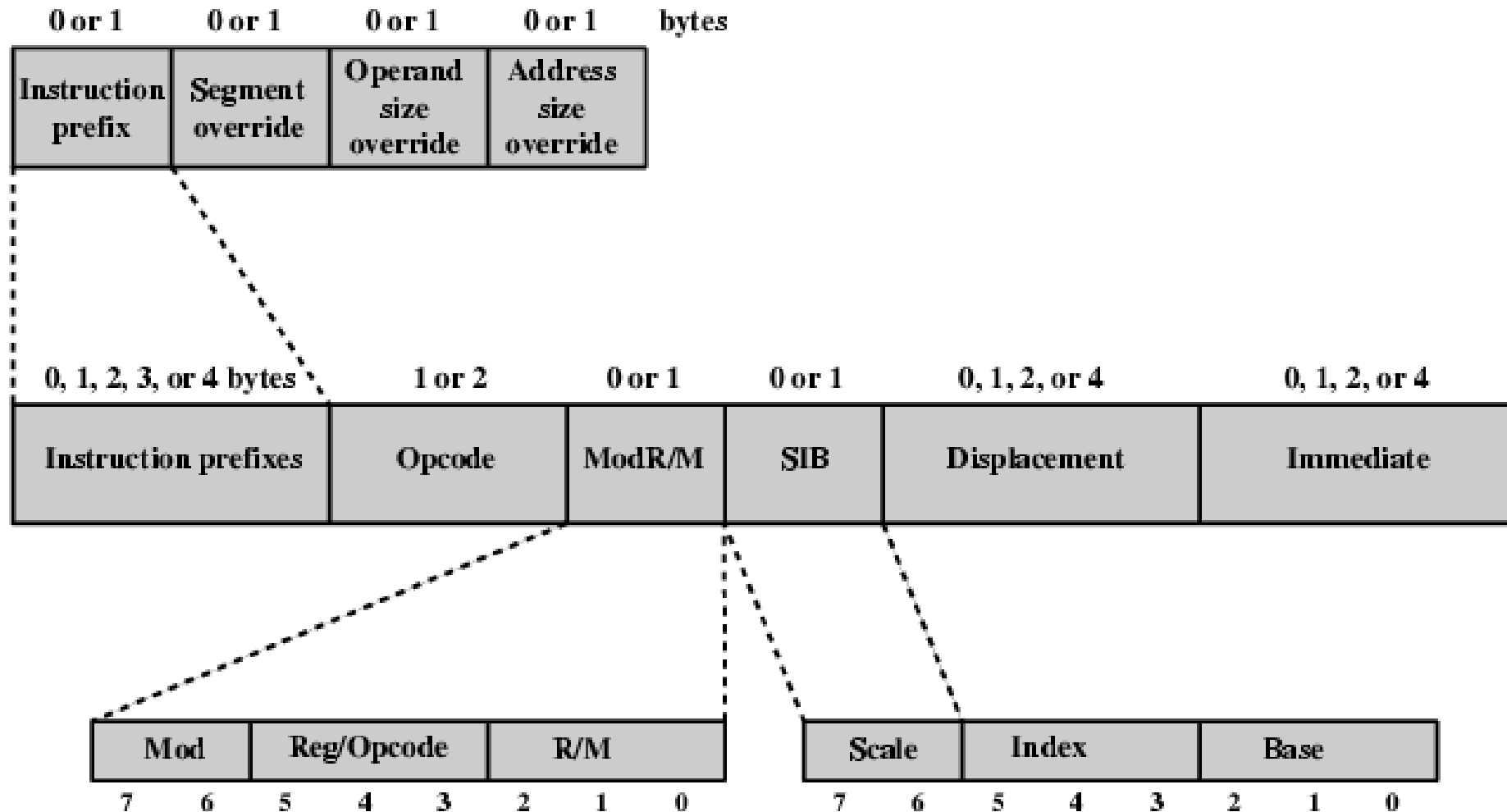
# Allocation of Bits

- ▶ Number of addressing modes
  - ▶ Number of operands
  - ▶ Register versus memory
  - ▶ Number of register sets
  - ▶ Address range(e.g. direct addressing)
  - ▶ Address granularity
- 

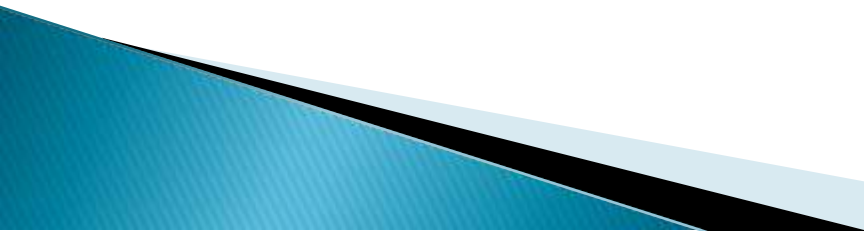
# Instruction Formats

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

# x86 Instruction Format



# Instruction Format:(optional prefixes)

- ▶ **Instruction Prefix:** LOCK or REP prefix.
  - ▶ **Segment Override:** Explicitly specifies which segment register an instruction should use, overriding the default segment-register selection generated by the processor for the instruction.  
e.g. MOV AX,CS:[SI]
  - ▶ **Operand size:** default operand size is 16 or 32 bit. Switches between 16 or 32 bit.
  - ▶ **Address size:** determines the displacement size in instructions and the size of address offsets generated during EA calculation.(16 bit or 32 bit)
- 



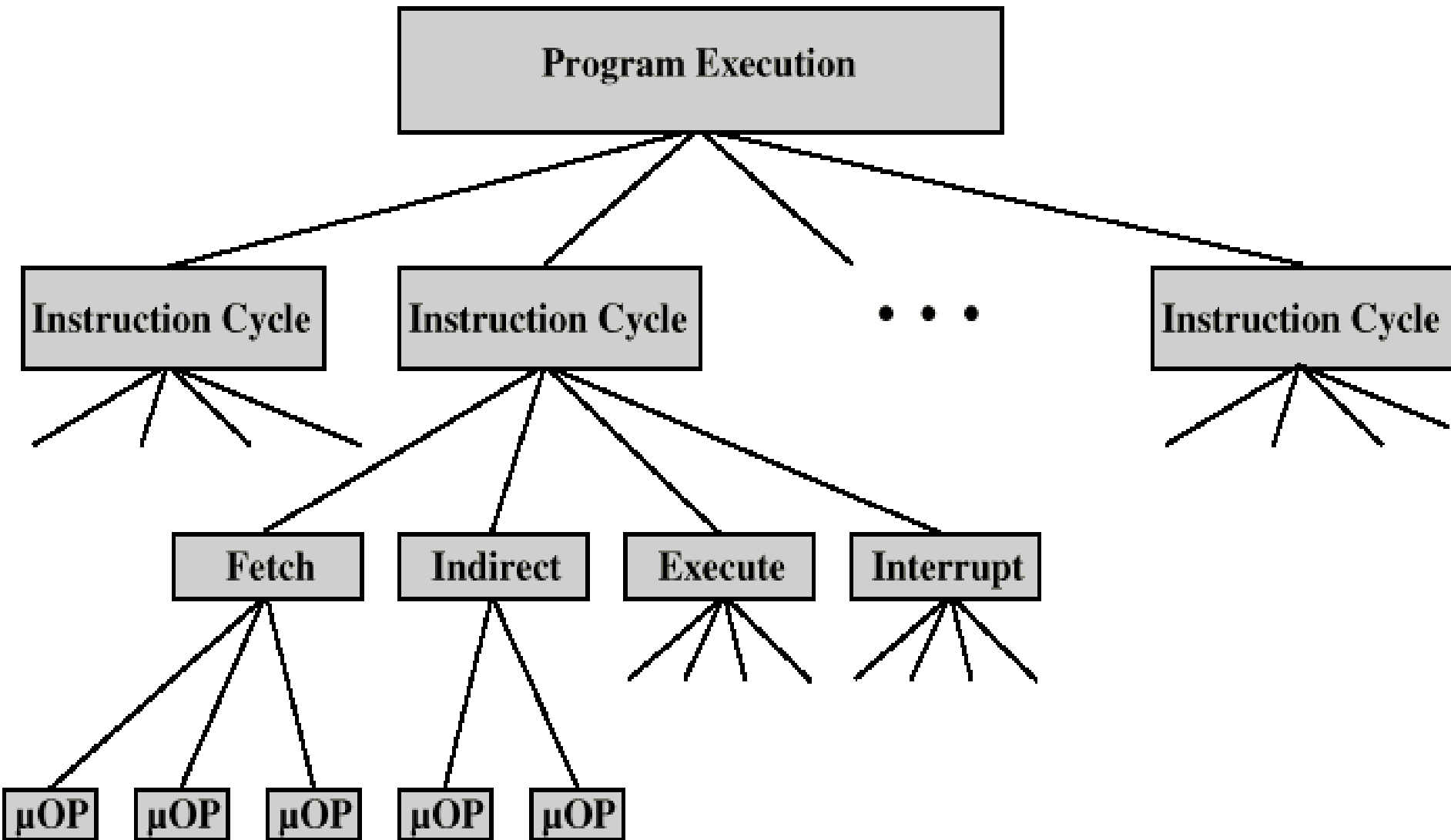
# Instruction Format:(General instruction format)

- ▶ **Opcode:** it specifies
  - If data are byte or full size.
  - Direction of data operation(to or from memory)
  - Whether immediate data field must be sign extended.
- ▶ **Mod R/m:** specifies whether operand is in a register or in memory.
  - Mod: combines with r/m field to form 32 possible values(8 registers and 24 indexing modes)
  - Reg/opcode:specifies either register number or 3 more bits of opcode information.
  - R/m: can specify a register location or can form a part of mod field.

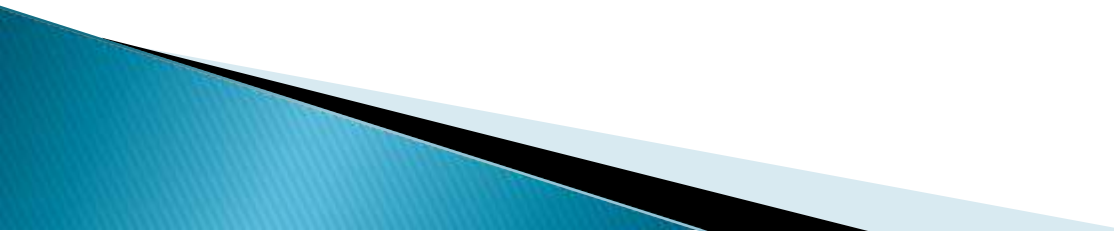
# Instruction Format:

- ▶ **SIB:** 3 fields
  - Scale(2 bits): specifies scale factor for scaled indexing.
  - Index(3 bits): specifies index register
  - Base(3 bits): specifies base register
- ▶ **Displacement** : displacement 8,16 or 32 bit
- ▶ **Immediate**: provides immediate value 8,16 or 32 bit.

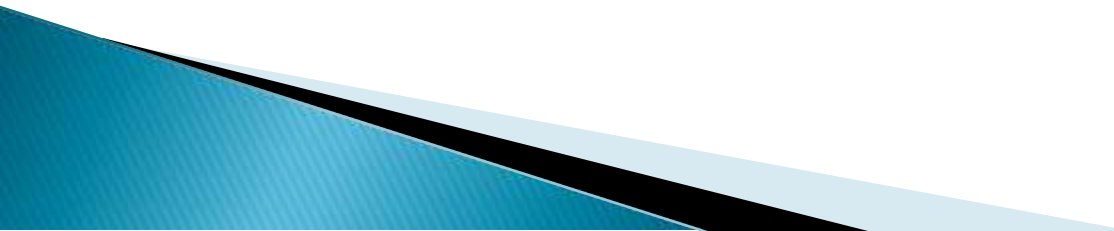
# Constituent Elements of Program Execution




# 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
- 

- ▶ **Microoperation:** A micro-operation is a primitive action performed by a machine on the data stored in the registers.
  - ▶ **Microinstruction:** Every bit of a control word stands for a control signal . Each microinstruction is defined by a set of control signals.
- 

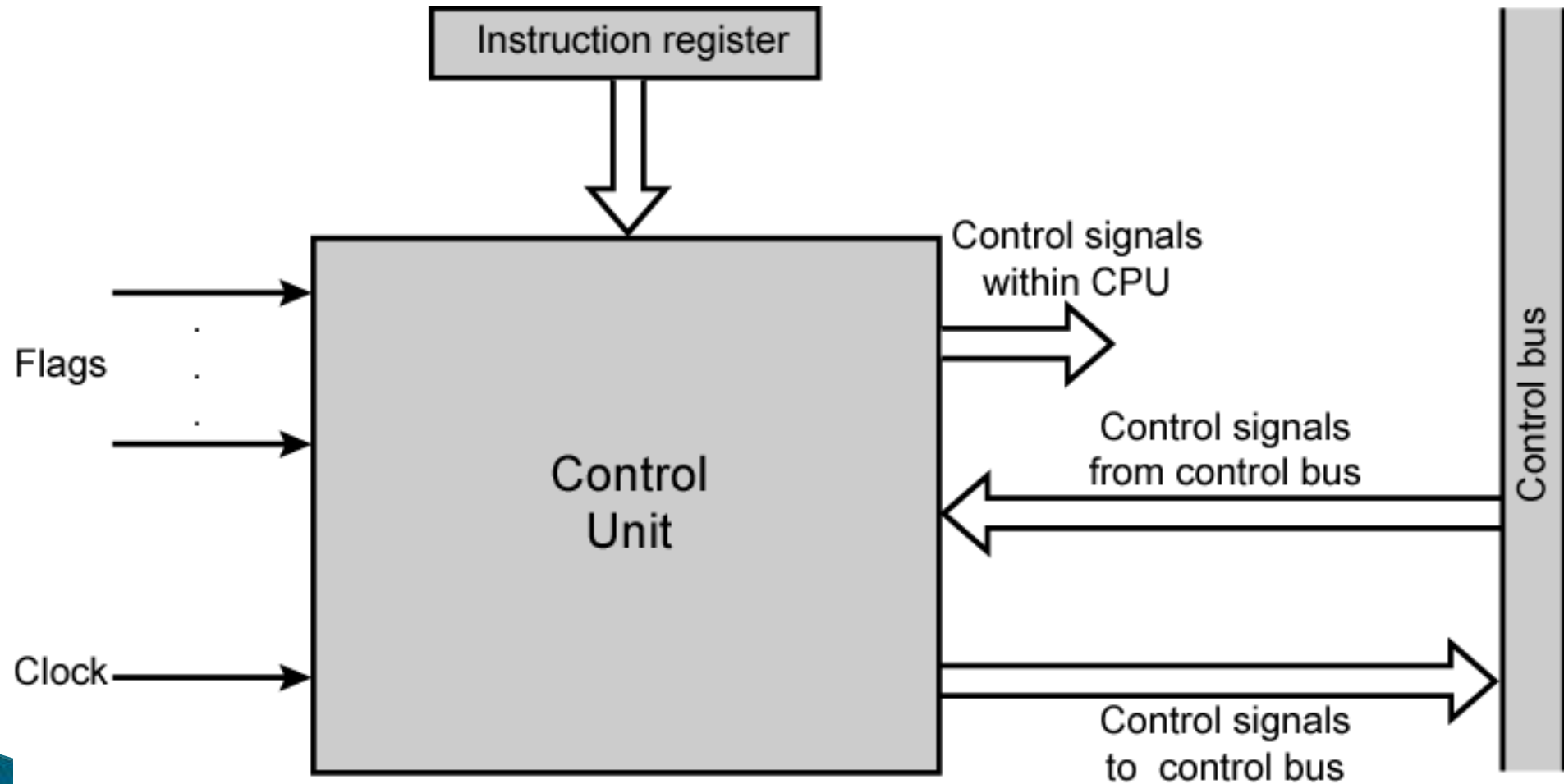
- ▶ Microprogram: A sequence of microinstructions is known as a microprogram.
- ▶ Microcode: A logically coherent portion of microprogram is called microcode.

# Control Unit





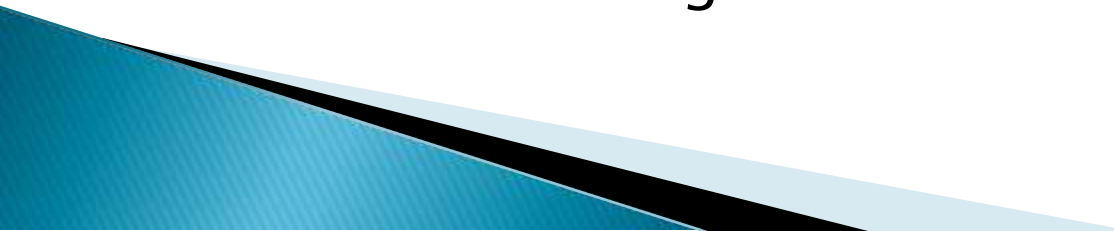
# Model of Control Unit




# 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

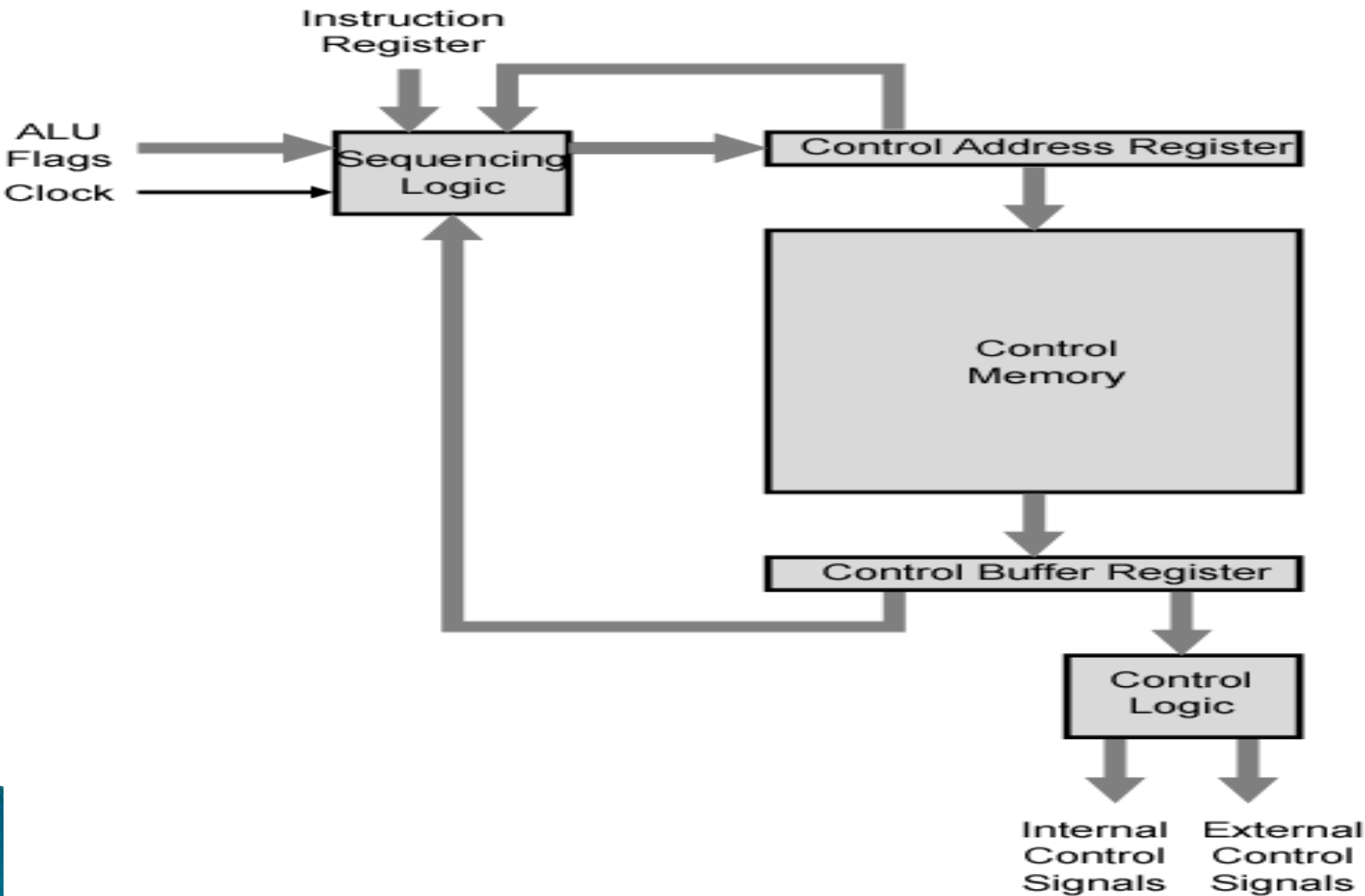
- **Within CPU**
    - Cause data movement
    - Activate specific functions
  - **Via control bus**
    - To memory
    - To I/O modules
- 

# Control Unit


- Hardwired control unit
- Microprogrammed control unit



# 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
- 

Execution of a microinstruction is responsible for generation of a set of micro-operations. A single microinstruction can cause execution of one or more micro-operations. A sequence of microinstructions (a micro-program) can cause execution of an instruction.

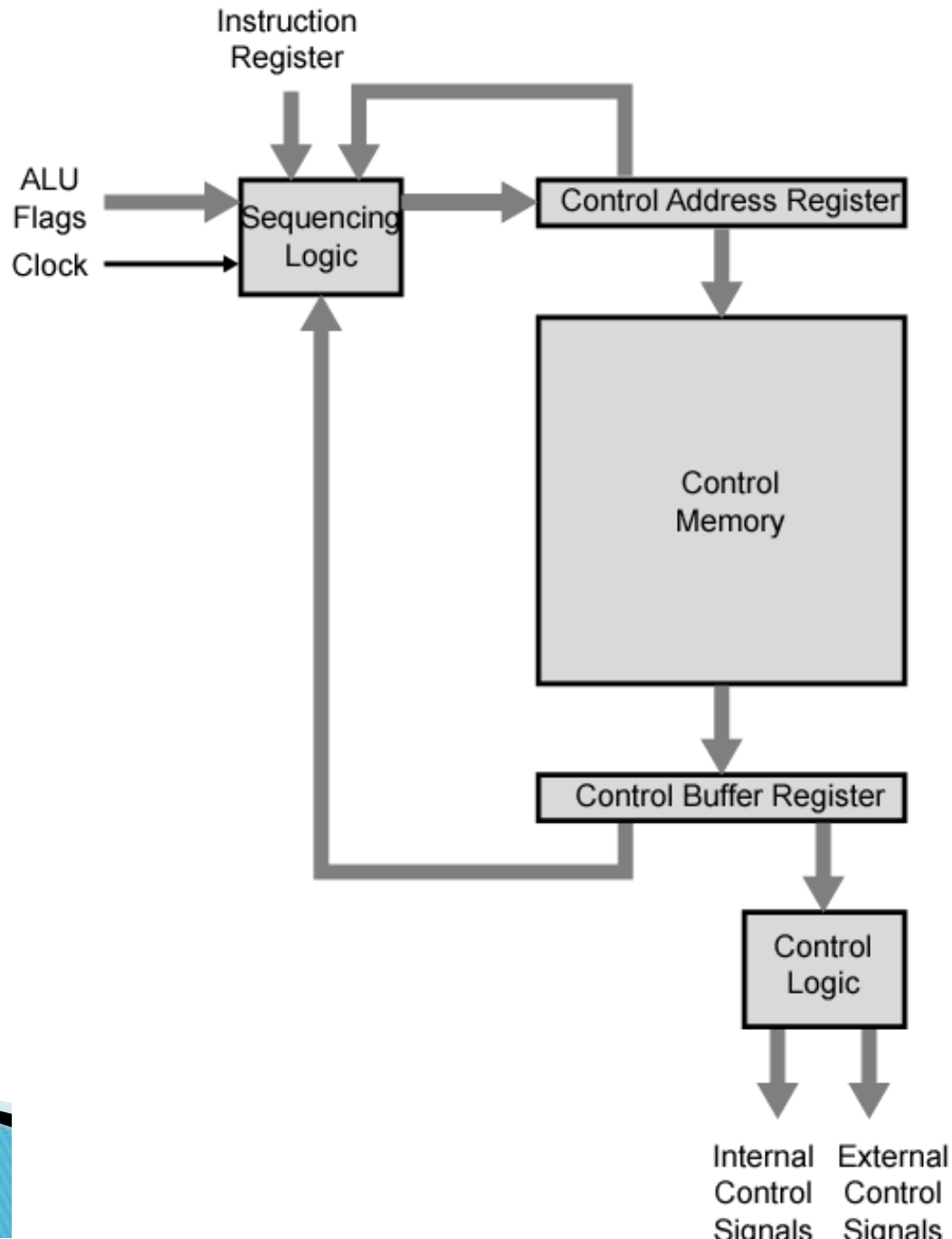
Memory address		Control field										Address field	
		$C_0$	$C_1$	$C_2$	$C_3$	$C_4$	$C_5$	$C_6$	$C_7$	$C_8$	$C_9$	$C_{10}$	
1.	0000	0	1	1	0	0	1	0	1	1	0	0	0001
2.	0001	1	0	0	1	1	1	0	0	1	0	1	0010
3.	0010	1	1	0	0	0	1	1	0	0	1	0	0011
	⋮												

**Fig. 3.35 (a) : Micro-program**


Execution of microinstructions at memory address 0000,  $C_1$ ,  $C_2$ ,  $C_3$  is shown. Address of the next instruction is provided by the address field. A microinstruction consists of:



# Control Unit Organization



# Implementation

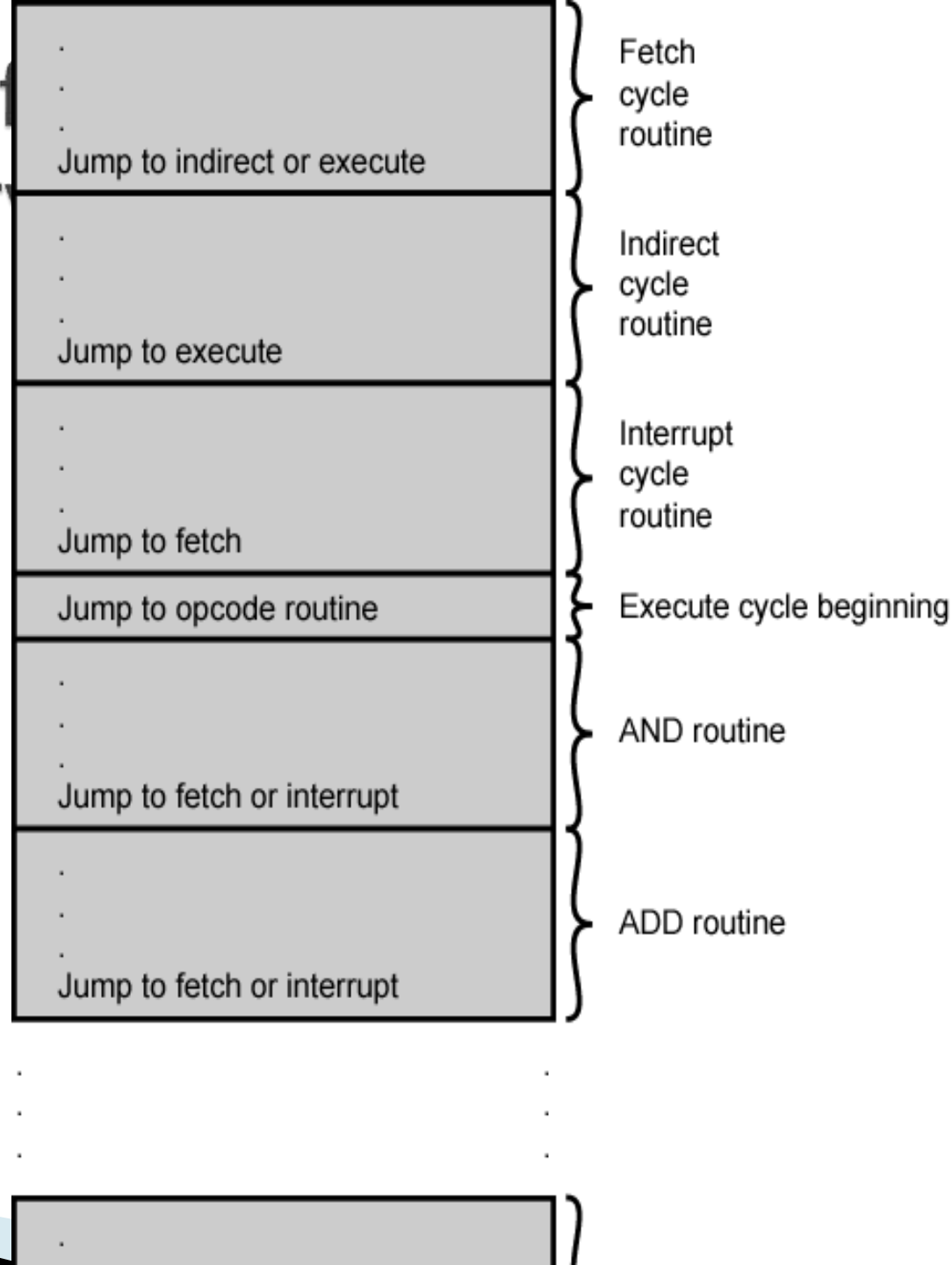
- ▶ All the control unit does is generate a set of **control signals**
  - ▶ Each control signal is **on** or **off**
  - ▶ Represent each control signal by a **bit**
  - ▶ Have a **control word** for each micro-operation
  - ▶ Have a **sequence of control words** for each machine code instruction
  - ▶ Add an **address** to specify the next micro-instruction, depending on conditions
- 

# Micro-programmed Control

- ▶ Use sequences of instructions to control complex operations
- ▶ Called micro-programming or firmware



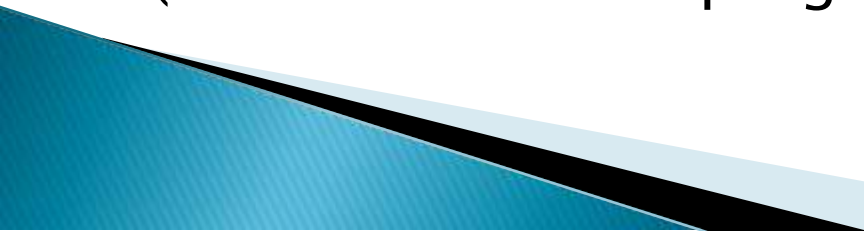
# Organization of Control Memory



# 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

# Micro-instruction Types

- ▶ Each micro-instruction specifies **single (or few)** micro-operations to be performed
    - (***vertical*** micro-programming)
  - ▶ Each micro-instruction specifies **many** different micro-operations to be performed in parallel
    - (***horizontal*** micro-programming)
- 

## 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

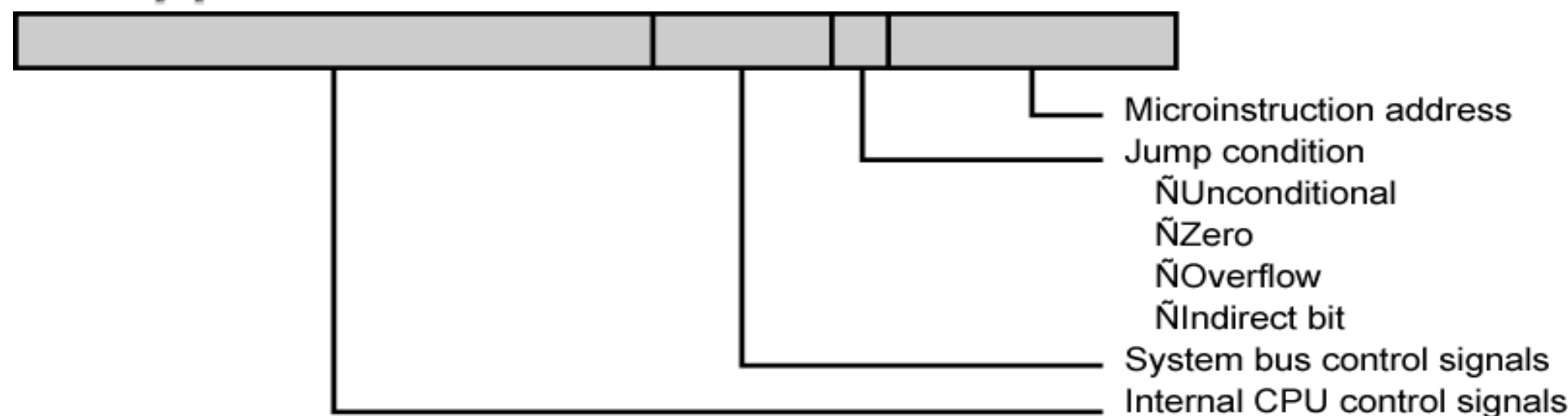
## Horizontal Micro-programming

- ▶ **Wide** memory word
- ▶ **High degree of parallel** operations possible
- ▶ Little encoding of control information

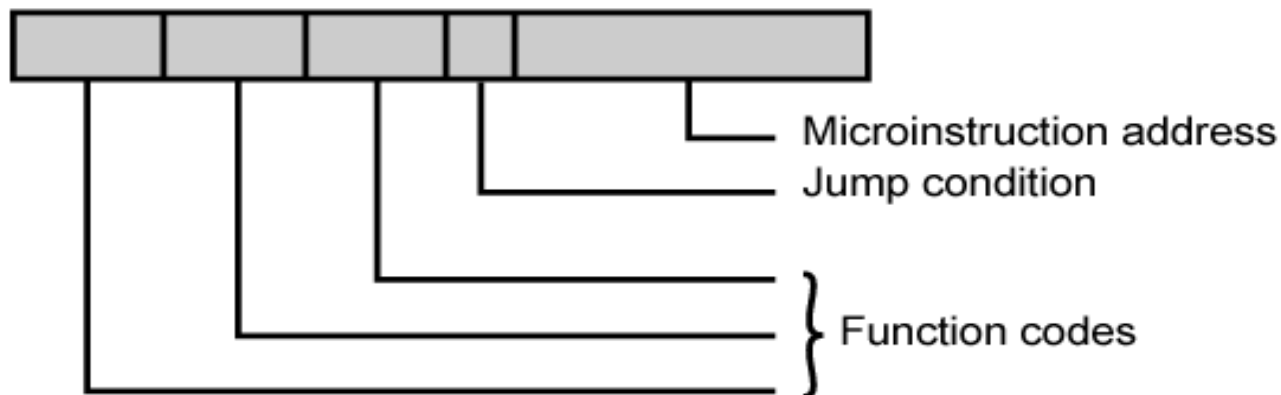




# Typical Microinstruction Formats

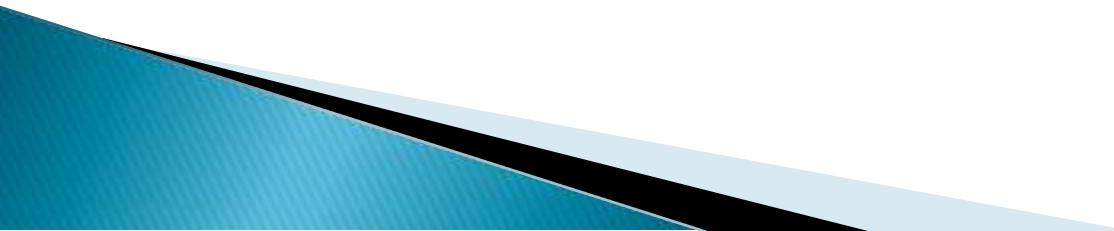


(a) Horizontal microinstruction

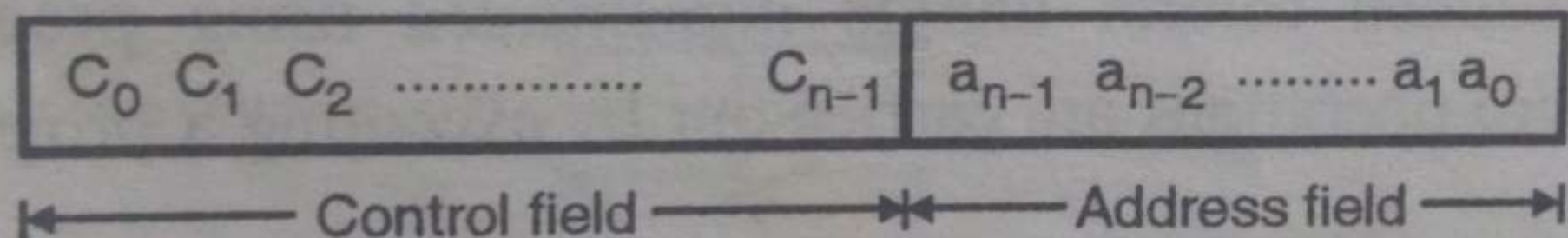


(b) Vertical microinstruction

# Wilkes Control

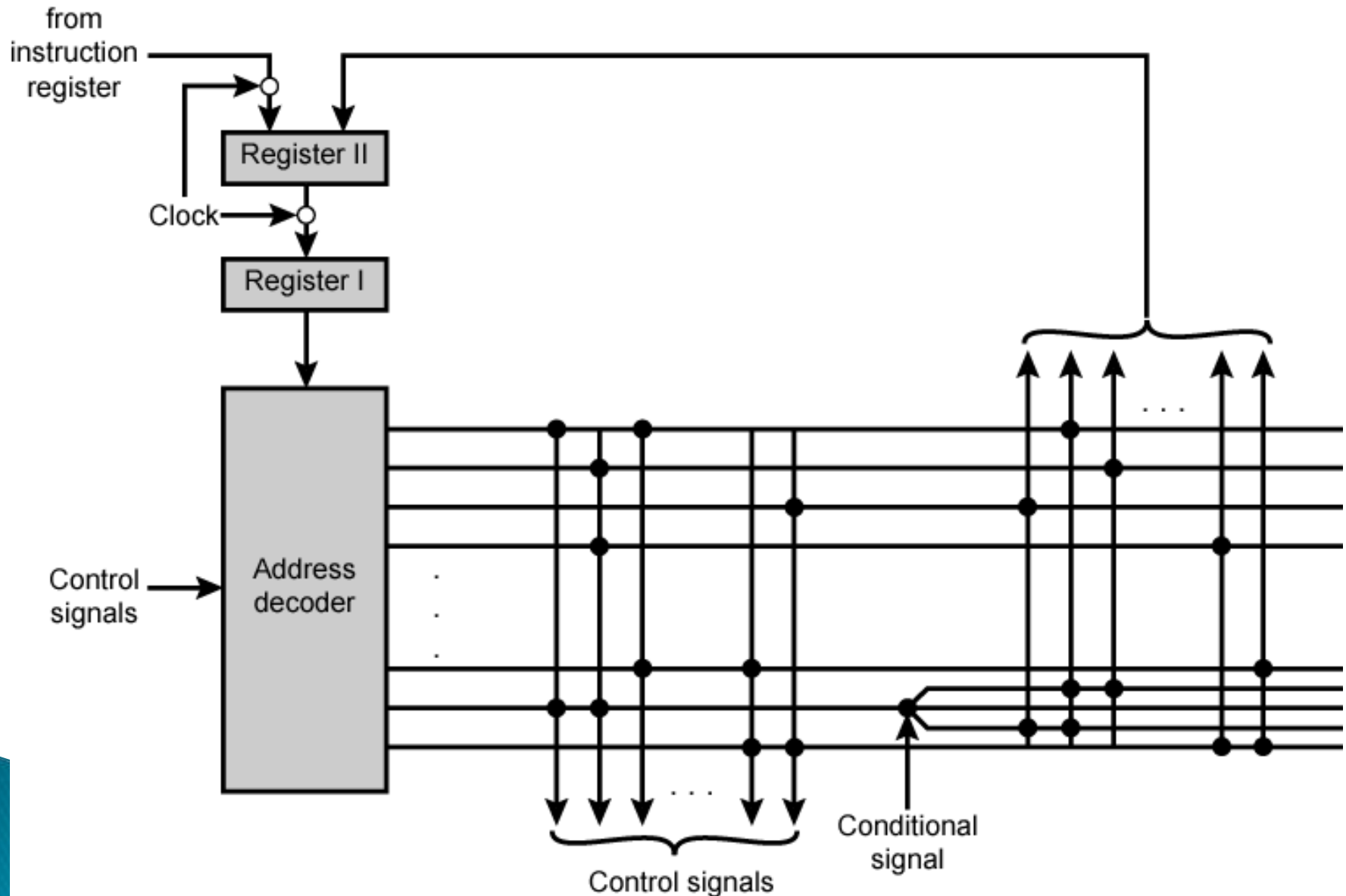
- ▶ 1951
  - ▶ A microinstruction has two major components: Control field & Address field
  - ▶ Matrix partially filled with diodes
  - ▶ During cycle, one row activated
    - Generates signals where diode present
    - First part of row generates control
    - Second generates address for next cycle
- 

model of a micro-programmed control unit was proposed. In this design, a microinstruction has two major components :



**Fig. 3.35 (b) : A typical microinstruction**

# Wilkes's Microprogrammed Control Unit



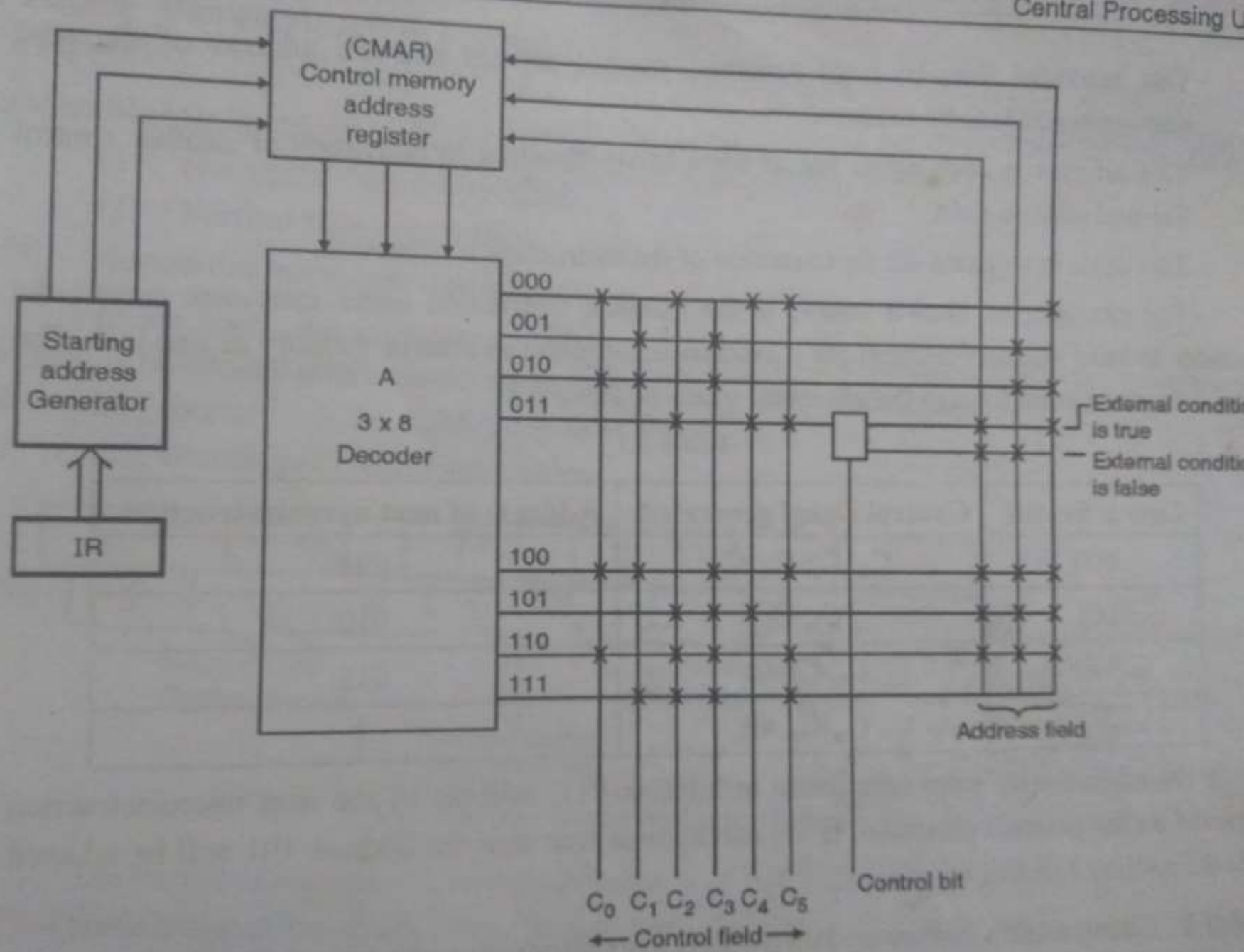


Fig. 3.35 (c) : Wilkes control

This activated line, in turn, generates control signals and the address of the microinstruction to be executed.

This address is once again fed to the CMAR resulting in activation of another control line and address field.

This cycle is repeated till the execution of the instruction is achieved.

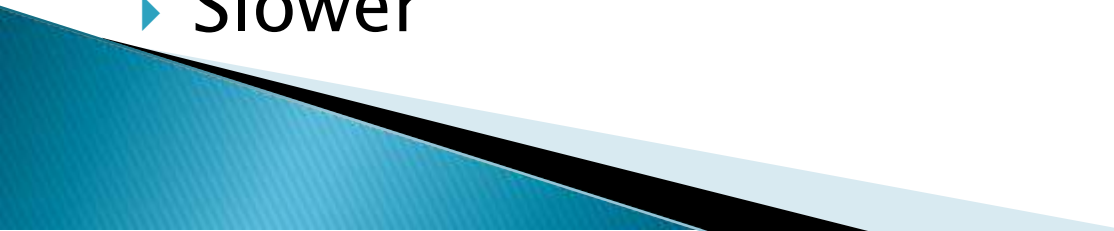
For example, as shown below, if the machine instruction under execution causes the microinstruction decoder to have an entry address for a machine instruction in control memory at line 000, the microinstruction decoder activates the lines in the sequence given in Table 3.1 :

**Table 3.1**

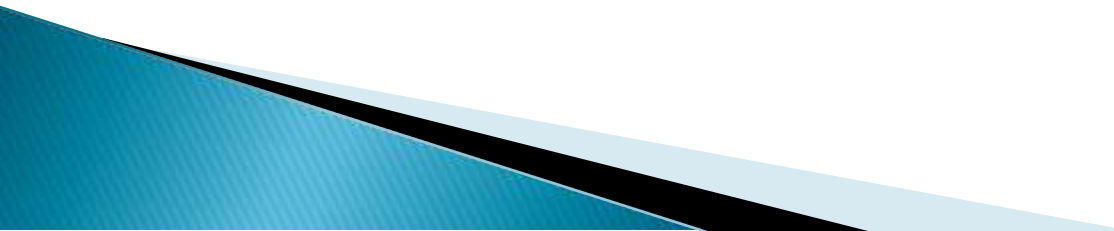
Line activated	Control signal generated	Address of next microinstruction
000	$C_0, C_2, C_4, C_5$	001
001	$C_1, C_3$	010
010	$C_0, C_1, C_3$	011
011	$C_2, C_4, C_5$	2

On execution of microinstruction at address 011, address of the next microinstruction depends on the external condition. If the condition is true then the address 101 will be selected. If the condition is false, the address 110 will be selected.

# Advantages and Disadvantages of Microprogramming

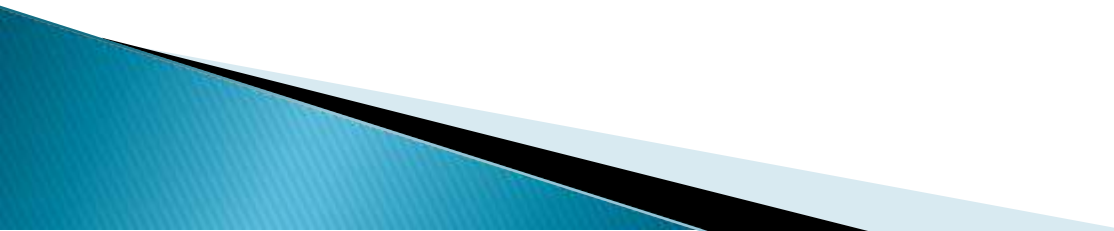
- ▶ Simplifies design of control unit
    - Cheaper
    - Less error-prone
  - ▶ Slower
- 

# Tasks Done By Microprogrammed Control Unit

- ▶ Microinstruction sequencing
  - ▶ Microinstruction execution
  - ▶ Must consider both together
- 



# Design Considerations

- ▶ Size of microinstructions
  - ▶ Address generation time
    - Determined by instruction register
      - Once per cycle, after instruction is fetched
    - Next sequential address
      - Common in most designed
    - Branches
      - Both conditional and unconditional
- 

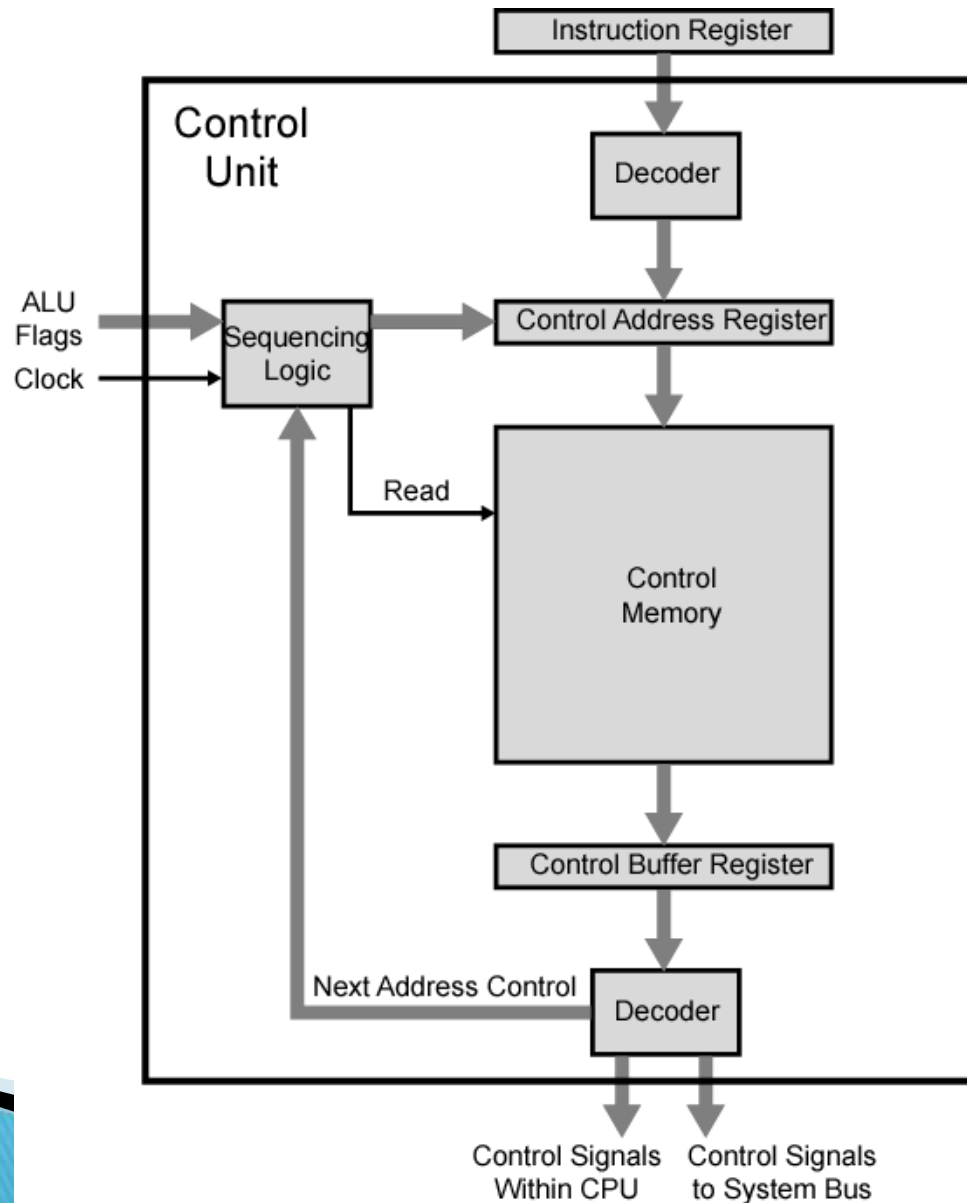
# Microinstruction Sequencing

A micro-program control unit has two parts:

- ▶ The control memory that stores the microinstructions.
- ▶ Sequencing circuit that controls the generation of the next address.

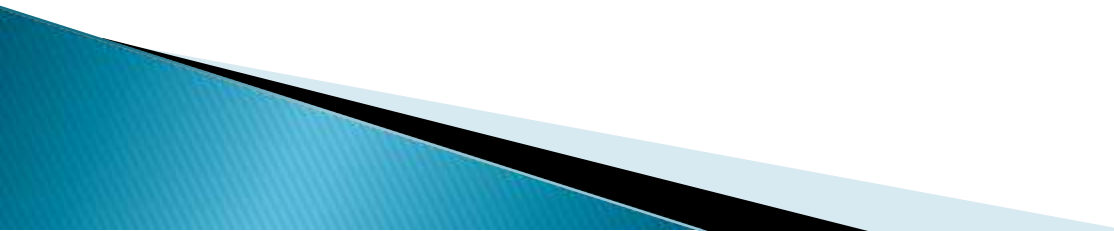


# Microprogrammed Control Unit




# Microinstruction Sequencing

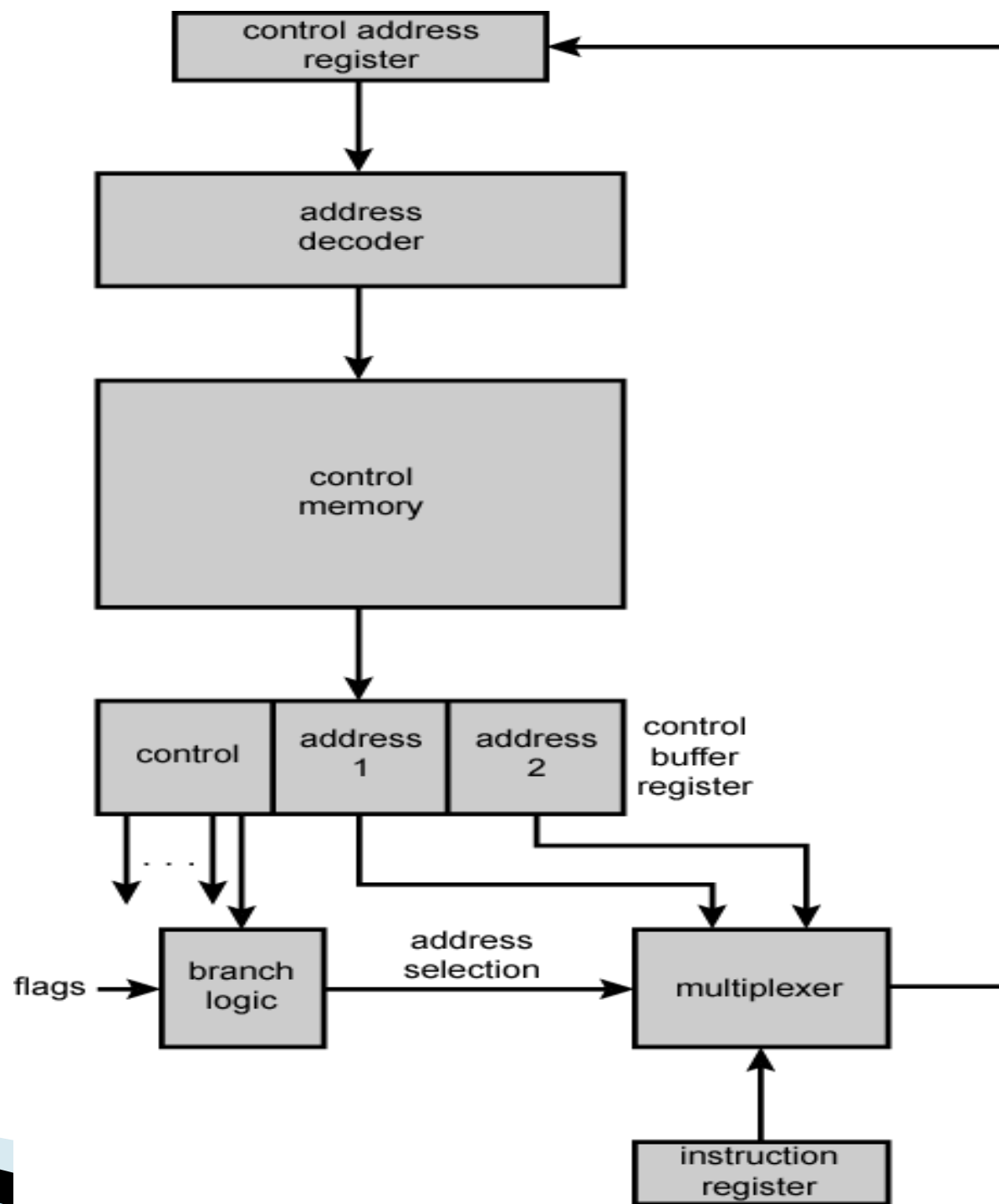
A sequencer has following sequencing capabilities:

- ▶ Increment the present address for control memory.
  - ▶ Branches to an address as specified by the address field of the microinstruction.
  - ▶ Branch to a given address if a specified status bit is equal to 1.
  - ▶ Transfer control to a new address as specified by an external source(IR)
  - ▶ Has a facility for subroutine calls and returns.
- 

# Sequencing Techniques

- ▶ Based on current microinstruction, condition flags, contents of IR, **control memory address** must be generated
  - ▶ Based on format of address information
    - Two address fields
    - Single address field
    - Variable format
- 

# Branch Control Logic: Two Address Fields




# Two address field

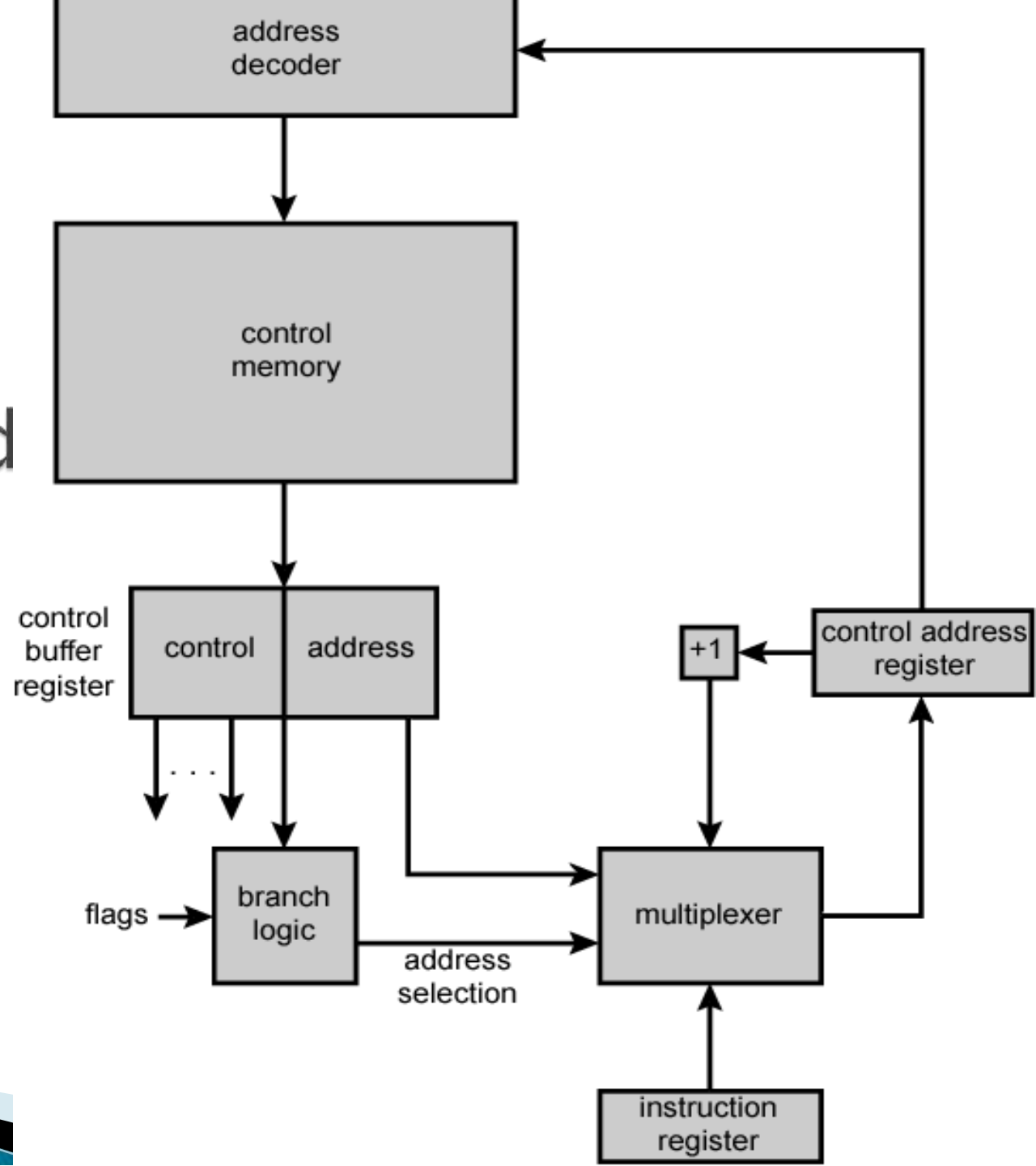
A multiplexer is provided to select:

- ▶ Address from the first address field
- ▶ Address from the second address field
- ▶ Starting address based on the Opcode field of the current instruction.

The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.



# Branch Control Logic: Single Address Field





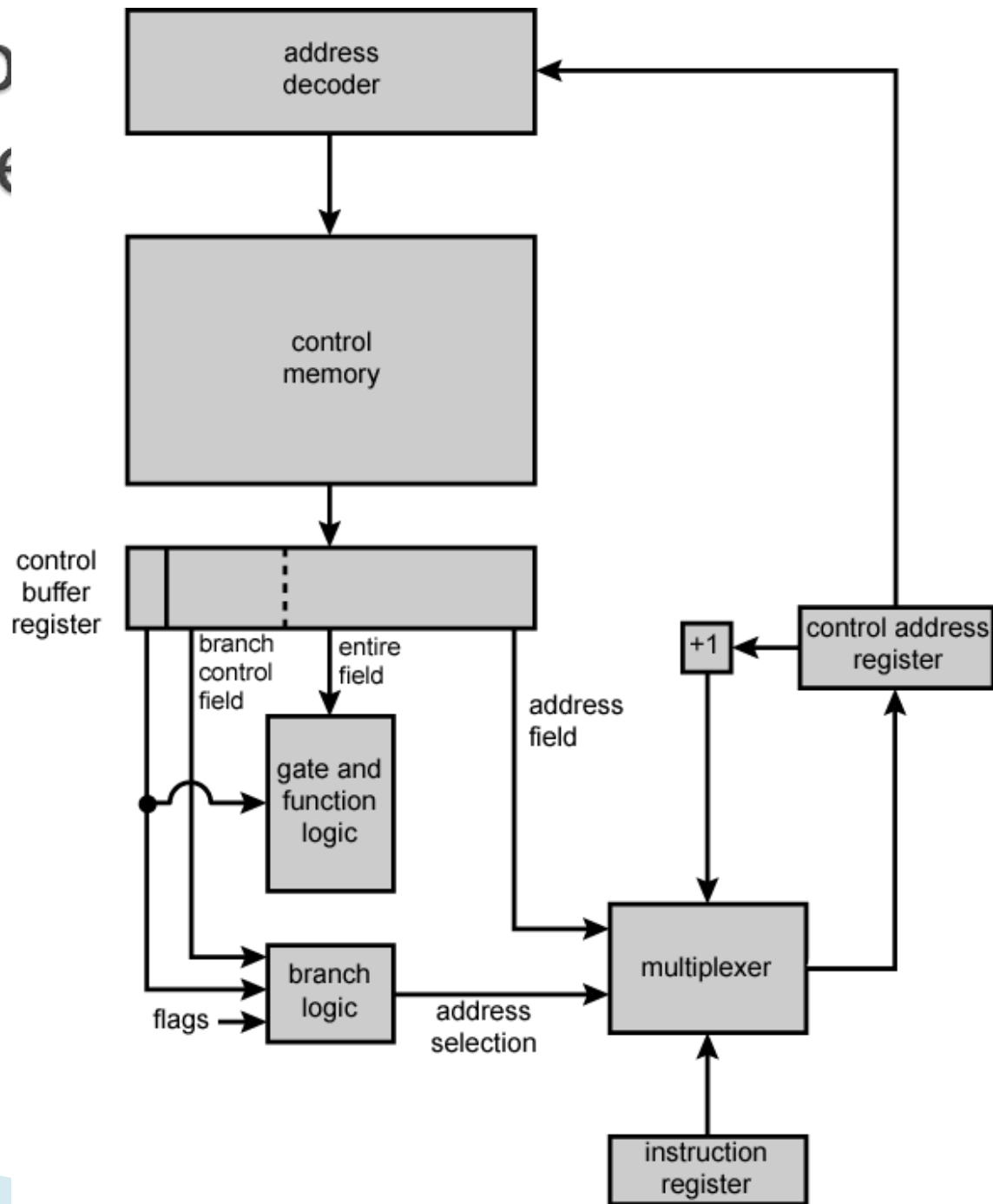
# Single address field

Two address approach is simple but it requires more bits in the microinstruction.

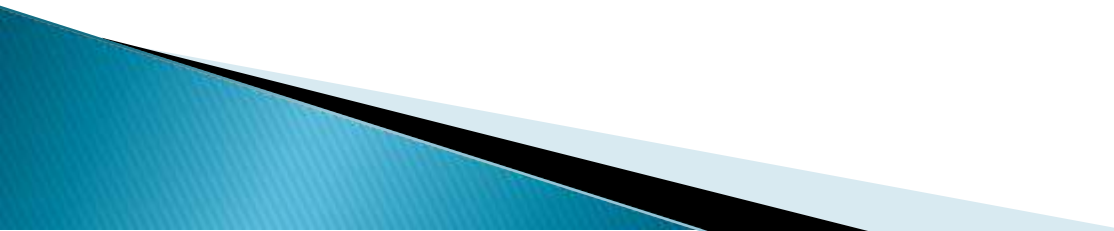
A multiplexer is provided to select next address using:

- ▶ Address field
- ▶ Based on OPcode in IR
- ▶ Next sequential address

# Branch Control Logic: Variable Format



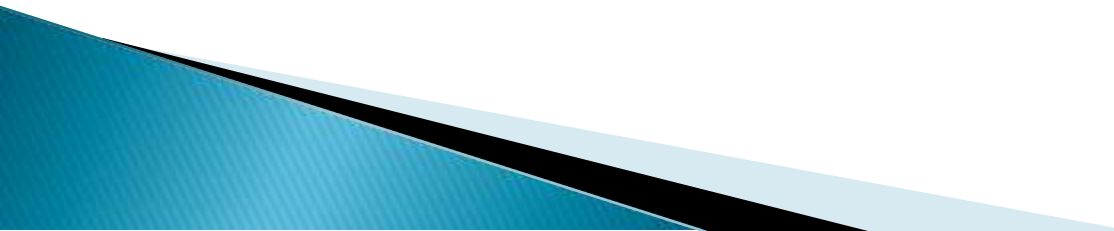
# Variable format

- ▶ In this approach, two entirely different microinstruction formats.
  - ▶ One bit designates which format is being used.
  - ▶ In first format the remaining bits are used to activate control signals. The next address is either the next sequential address or the address derived from the IR.
  - ▶ In the second format, some bits drive the branch logic module and the remaining bits provide the address. Branch is either conditional or unconditional.
- 

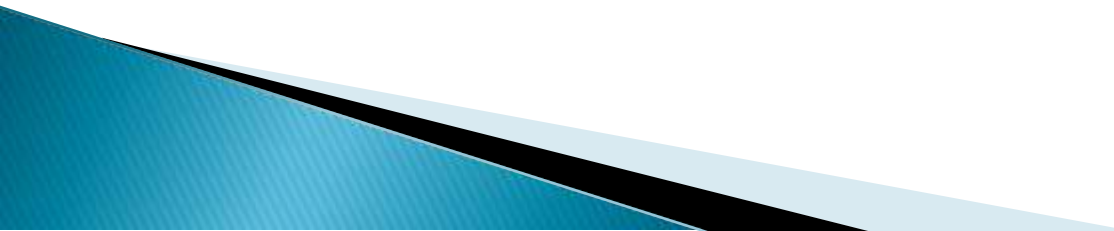
# Difference in Hardwired control unit and Microprogrammed control

u Hardwired Control	Microprogrammed control
Speed is fast	Speed is slow
Cost of implementation is more	Cost of implementation is less
In implementation sequential circuit is used	In implementation programming is used
Not flexible	Flexible
Ability to handle complex instructions is difficult	Ability to handle complex instructions is easier
Design process is complicated	Design process is systematic
Decoding and sequential logic is complex	Decoding and sequential logic is easy
Used in RISC	Used in CISC
Instn set size is small	Instn set size is large
Control memory is absent	Control memory is present
Chip area required is less	Chip area required is more.

# Functional Requirements(of Control Unit)

- Define basic elements of processor
  - Describe micro-operations processor performs
  - Determine functions control unit must perform
- 

# Applications of microprogramming

- ▶ The applications of Microprogramming are:
  - ▶ **In Realization of control unit: Microprogramming** is used widely now for implementing the control unit of computers
  - ▶ **In Emulation:** Emulation refers to the use of a microprogram on one machine to execute programs originally written for another machine. This is used widely as an aid for users in migrating from one computer to another.
  - ▶ **In Operating system:** Microprograms can be used to implement some of the primitives of operating system. This simplifies operation system implementation and also improves the performance of the operating system.
  - ▶ **In High-Level Language support:** In High-Level language various sub functions and data types can be implemented using microprogramming. This makes compilation into an efficient machine language form possible.
  - ▶ **In Microdiagnostics:** Microprogramming can be used for detection isolation monitoring and repair of system errors. This is known as microdiagnostics and they significantly enhance system maintenance. e. g. high speed multiplier.
  - ▶ **In User Tailoring:** By using RAM for implementing control memory (CM), it is possible to tailor the machine to different applications.
- 

## Difference between Hardwired control unit and Microprogrammed Control Unit

Attribute	Hardwired control	Microprogrammed control
speed	Fast	slow
Cost of imple.	more	cheaper
Imple. approach	Sequential circuit	programming
flexibility	Not flexible	flexible
Ability too handle complex instn	difficult	easier
Design process	complicated	systematic
Decoding & sequencing logic	complex	easy

## Difference between Hardwired control unit and Microprogrammed Control Unit

Attribute	Hardwired control	Microprogrammed control
Applications	RISC mP	CISC mP
Instruction set size	small	Large
Control memory	absent	Present
Chip area required	less	More