# Block diagram of 8086

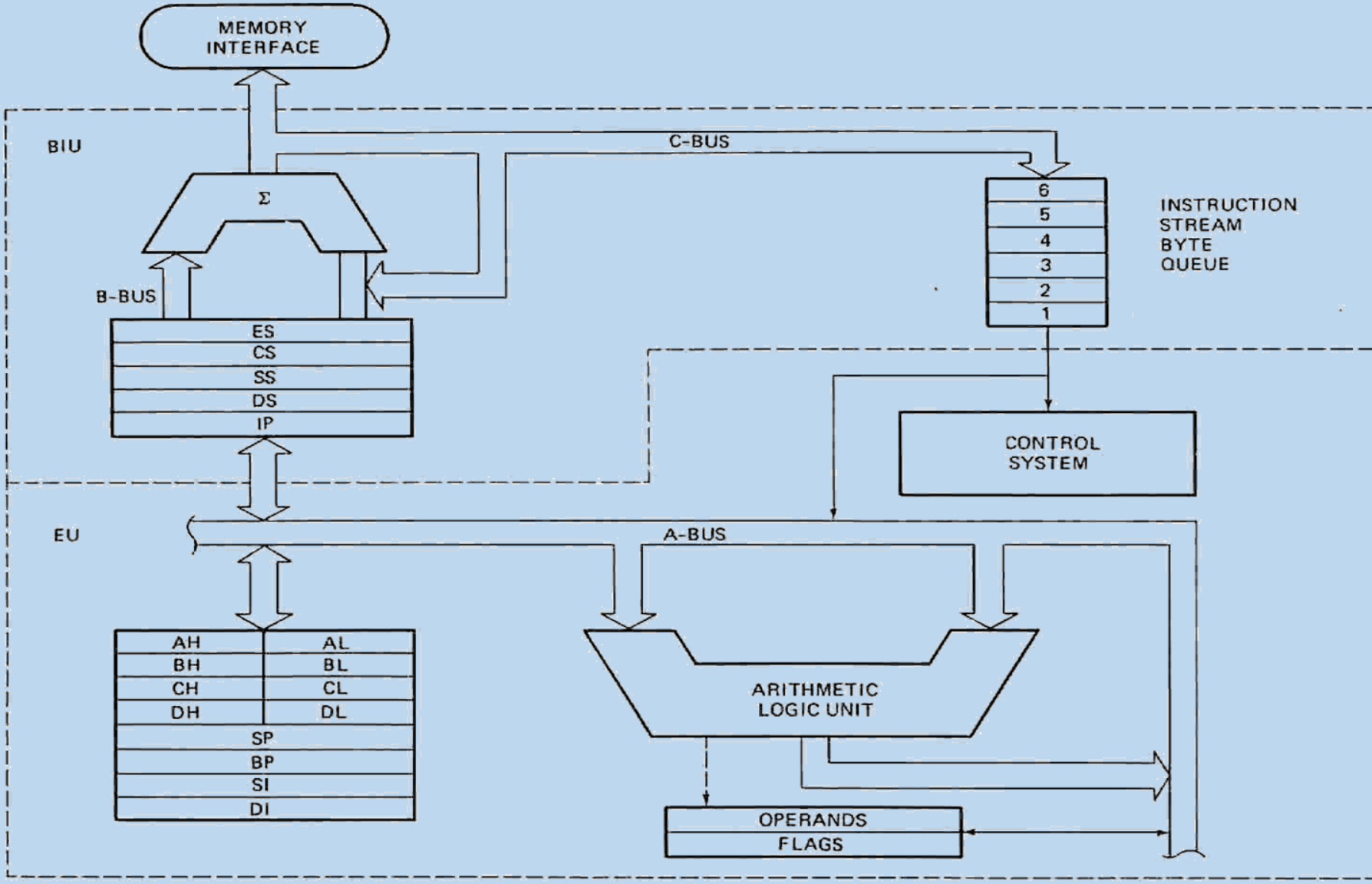
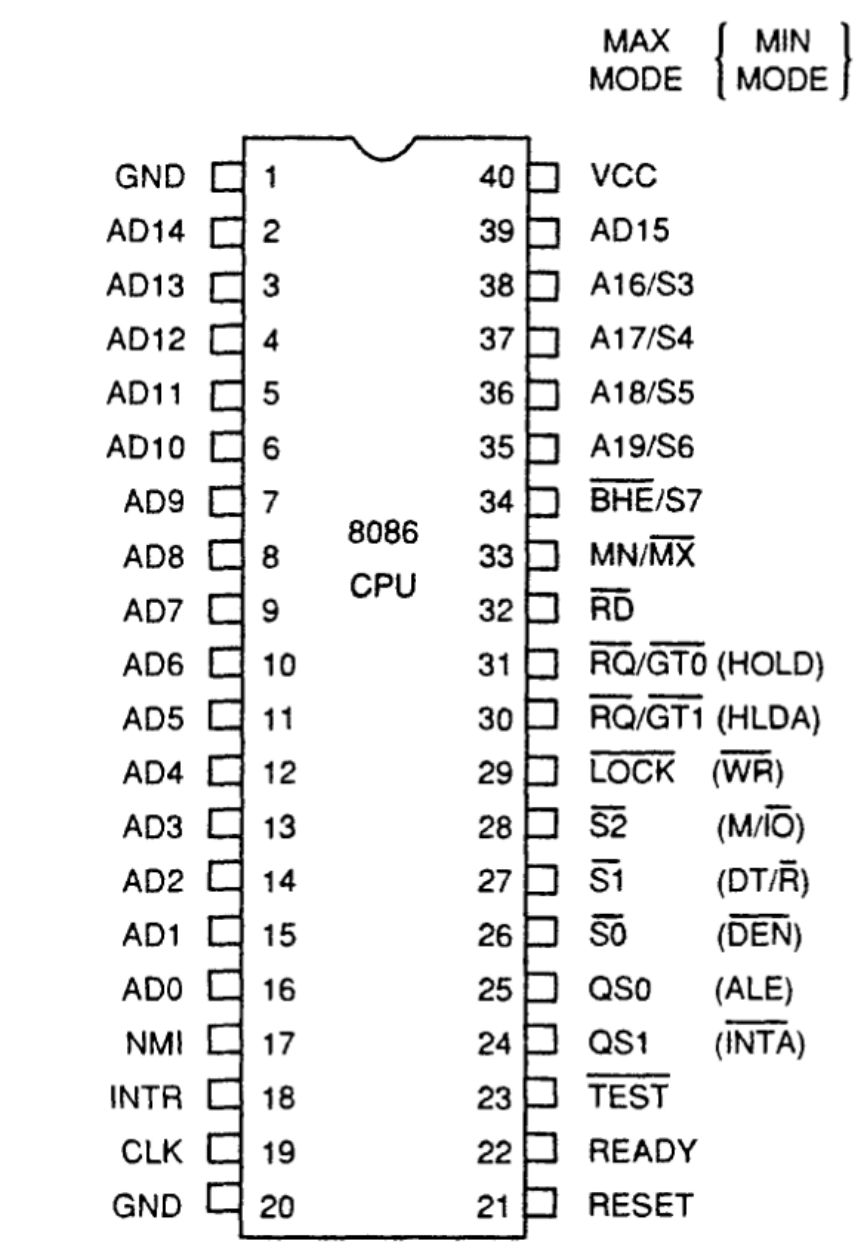
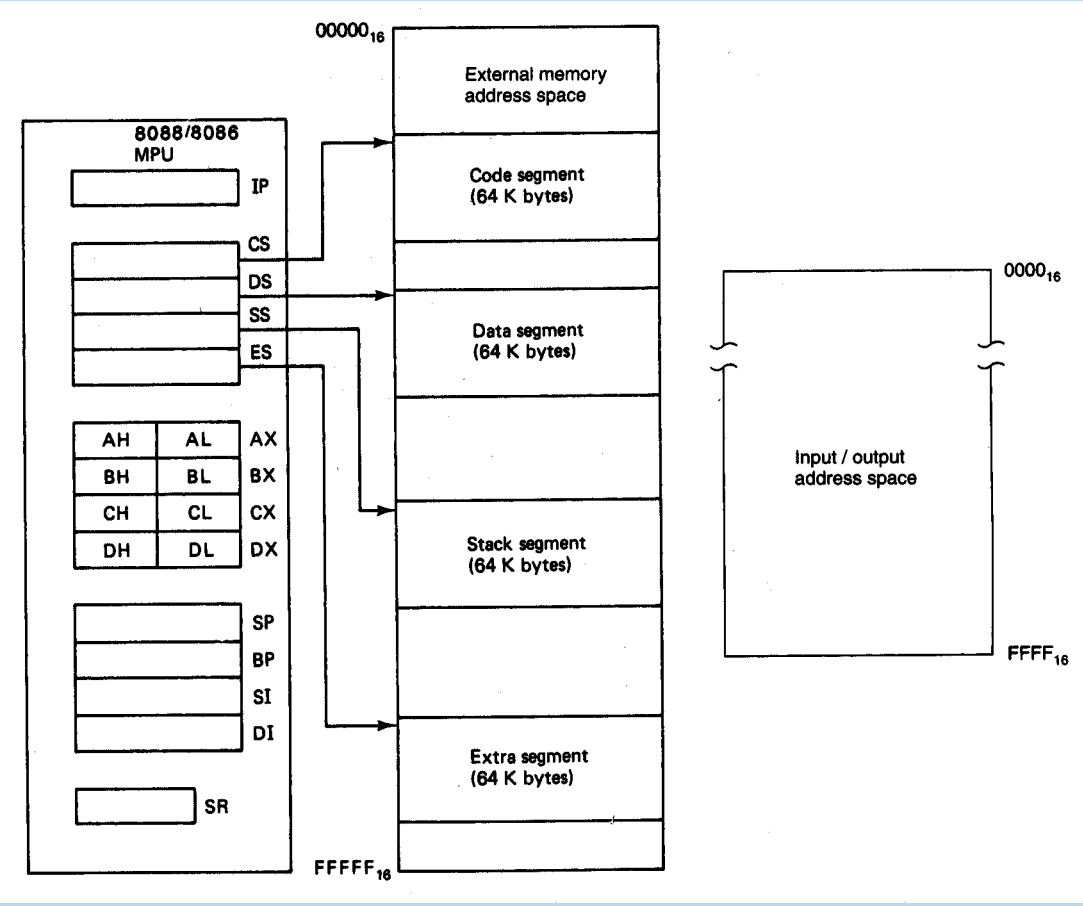



FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

# INTEL 8086 - Pin Diagram



# Software Model of the 8086 Microprocessors



# 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

# General Purpose Registers

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

# General Purpose Registers

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

# How Many GP Registers?

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

# 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

- 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
- Instruction Decoding Register
- Memory Address Register
- Memory Buffer Register

- Revision: what do these all do?

# Program Status Word

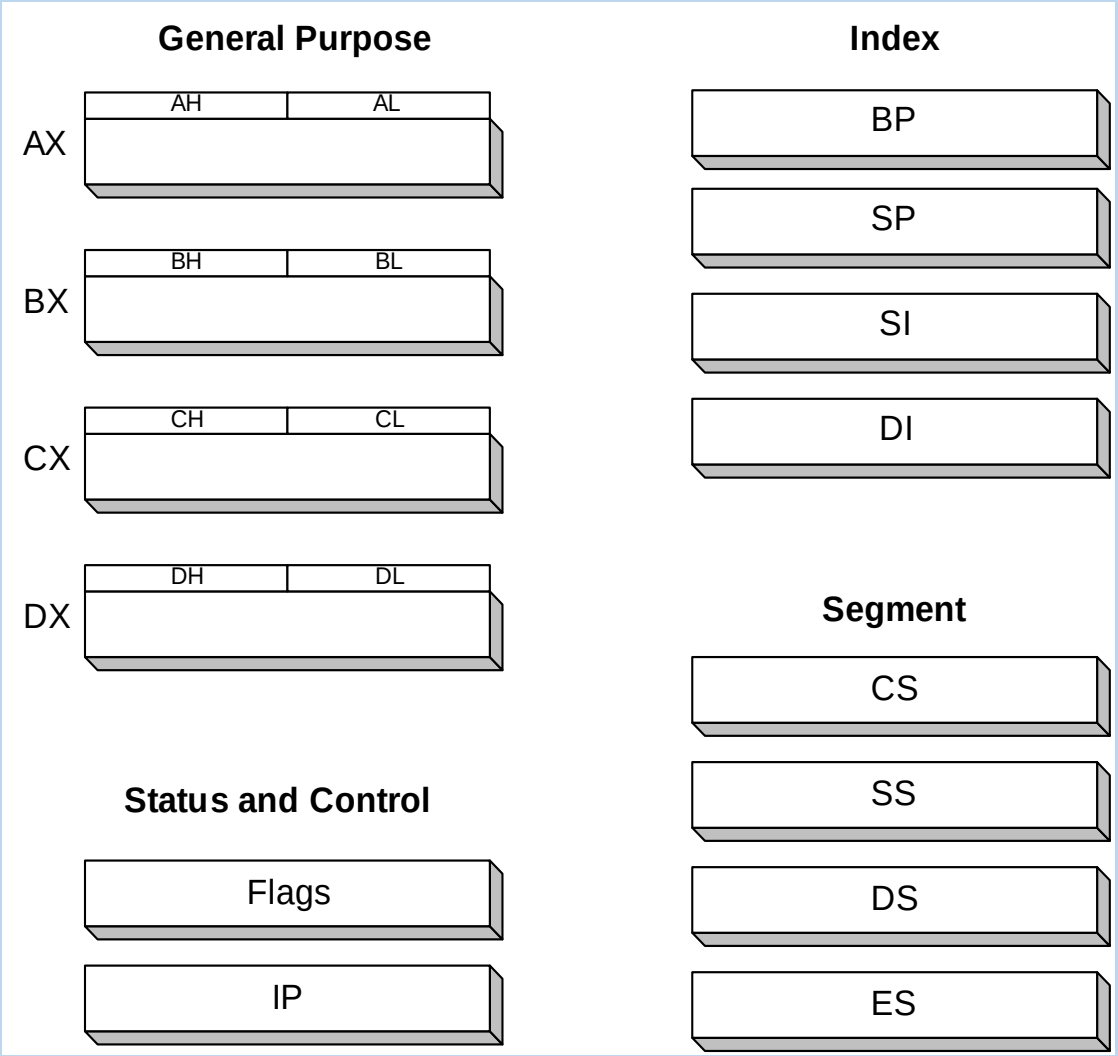
- A set of bits
- Includes Condition Codes
- Sign of last result
- Zero
- Carry
- Equal
- Overflow
- Interrupt enable/disable
- Supervisor

# Other Registers

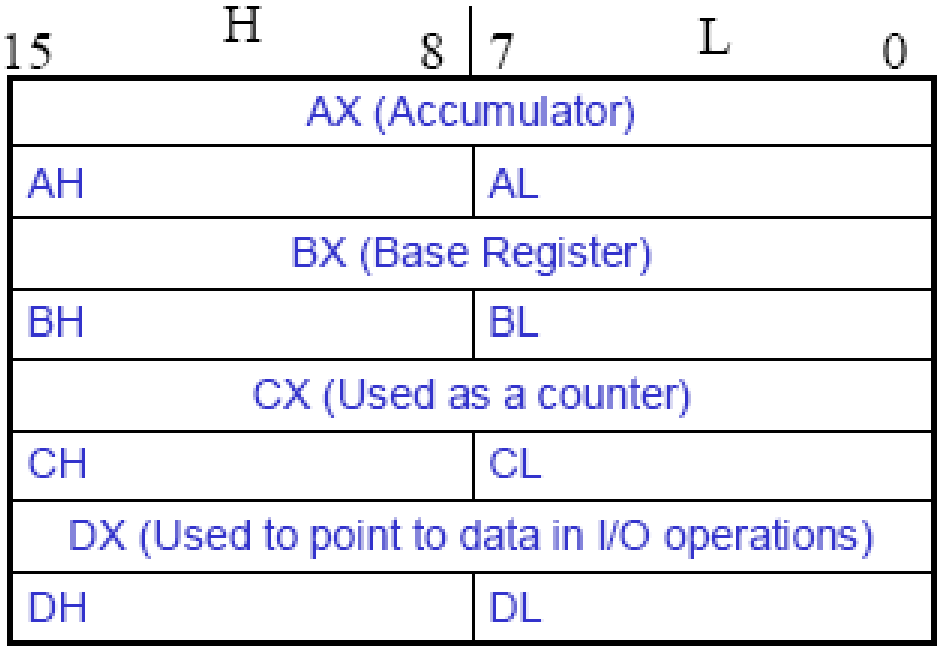
- May have registers pointing to:
  - Process control blocks (see O/S)
  - Interrupt Vectors (see O/S)

- N.B. CPU design and operating system design are closely linked

# 8086 Registers



# General Purpose Registers-8086




**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-8086

- **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-8086

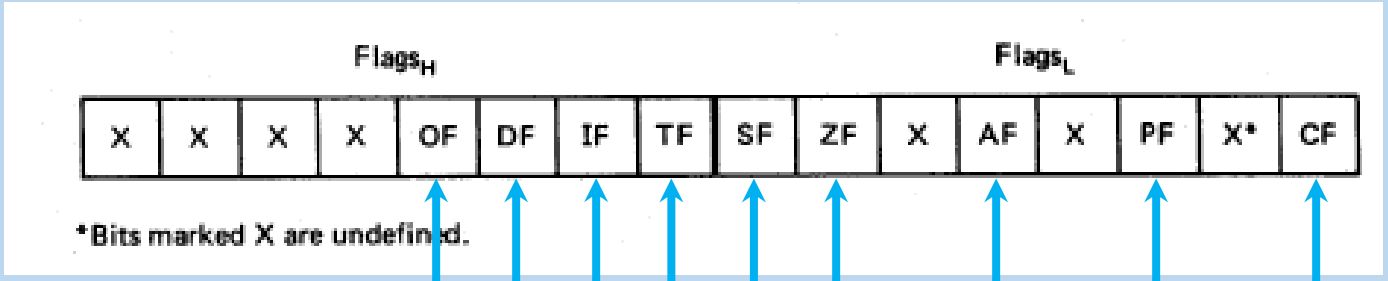
- **CX**
  - Count register
  - Used as a loop counter
  - Used in shift and rotate operations
- **DX**
  - Data register
  - Used in multiplication and division
  - Also used in I/O operations

# Pointer and Index Registers-8086

| | |
|---|---|
| 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




Overflow

Direction

Interrupt enable

Trap

Sign

Zero

Auxiliary Carry

Parity

Carry

6 are status flags

3 are control flag

# 8086 Programmer's Model

BIU registers
(20 bit adder)

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

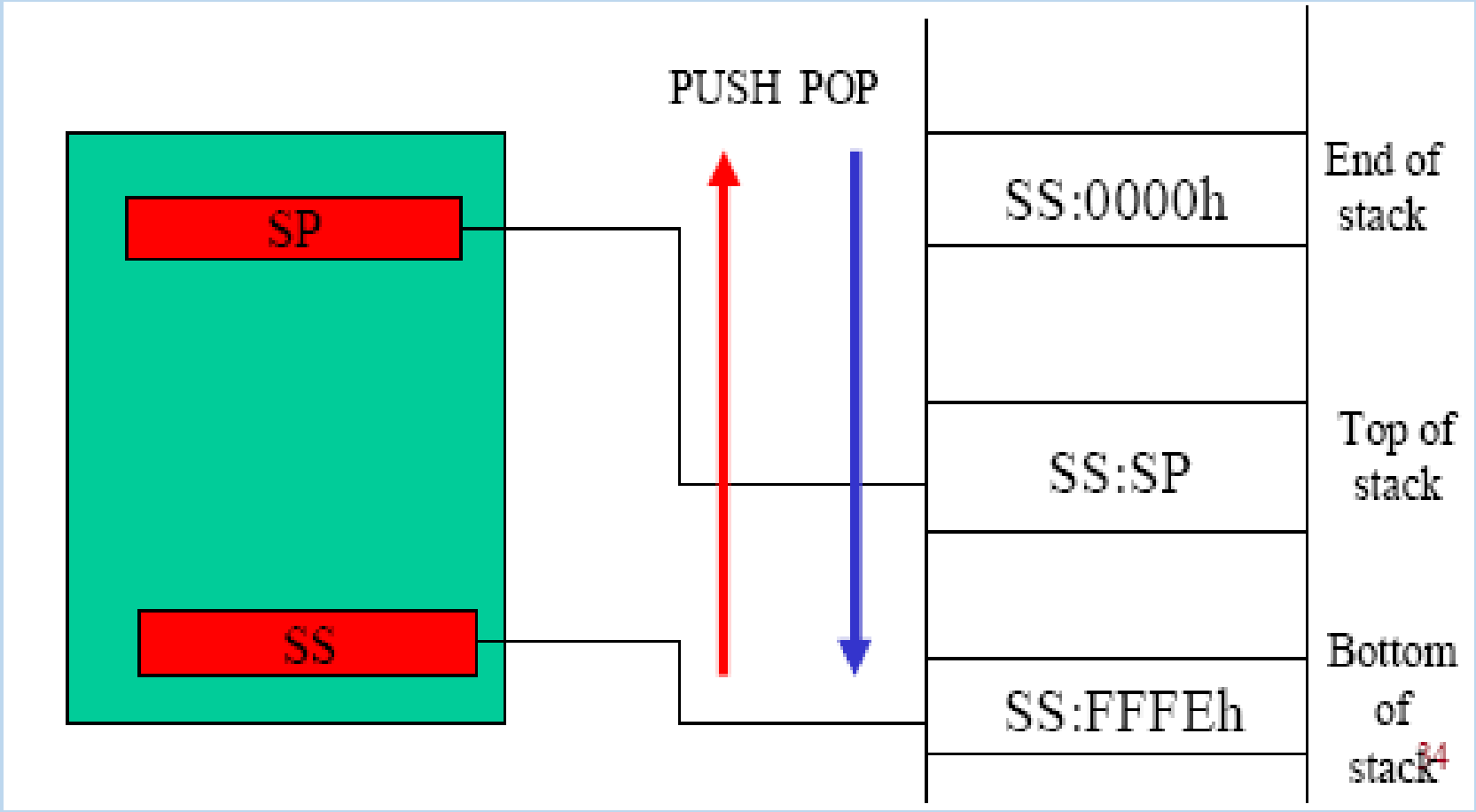
EU registers

| | | | |
|---|---|---|---|
| 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 | | |

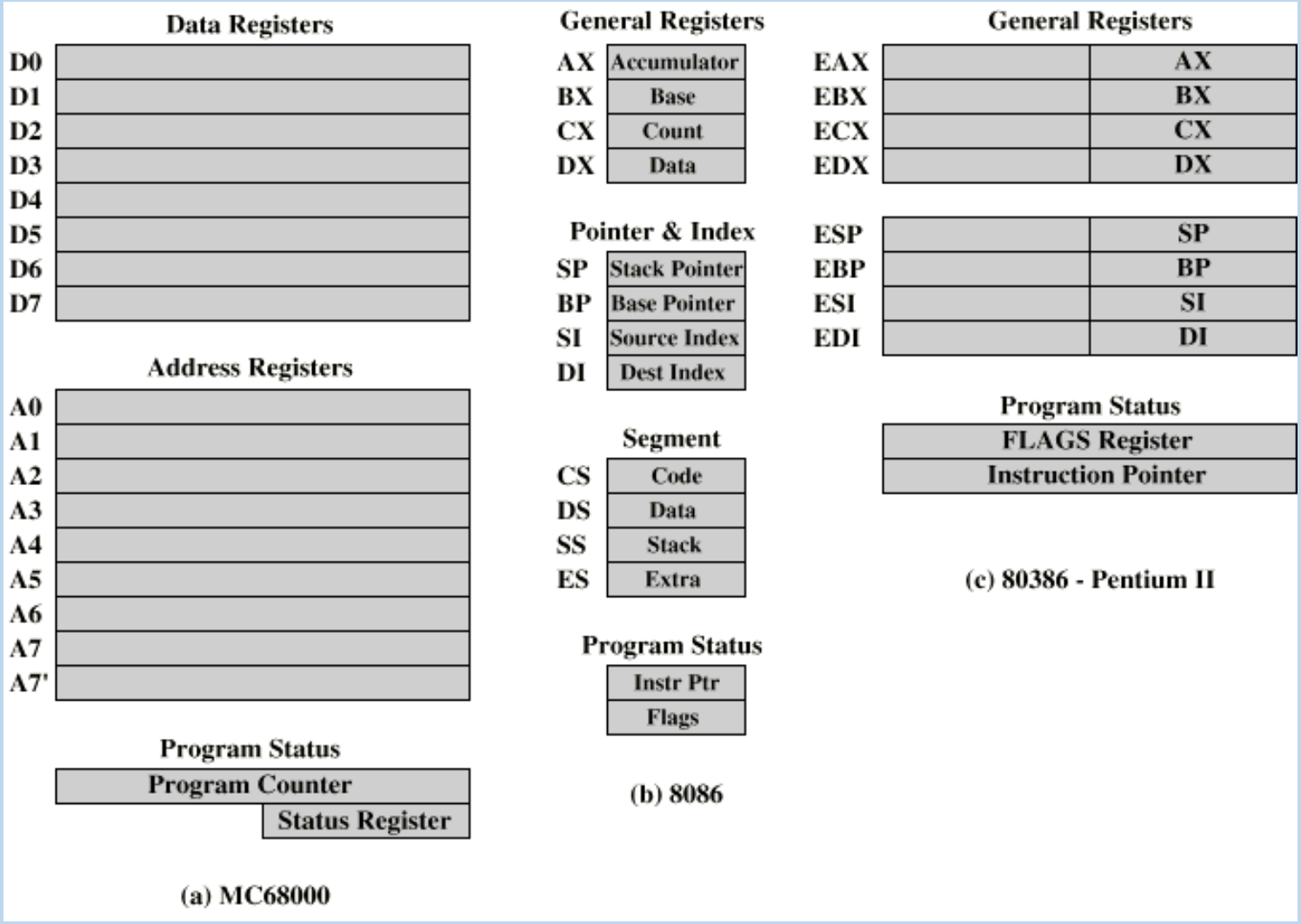
# The Stack-8086

- 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-8086



# Examples: Register Organizations



# Instruction format:8086

- Instructions are represented in memory by a series of “opcode bytes.”
- A variance in instruction size means that disassembly is position specific.
- Most instructions take zero, one, or two arguments:

```
instruction destination, source
```

For example: add AX, BX

is equivalent to the expression AX = AX + BX

# 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

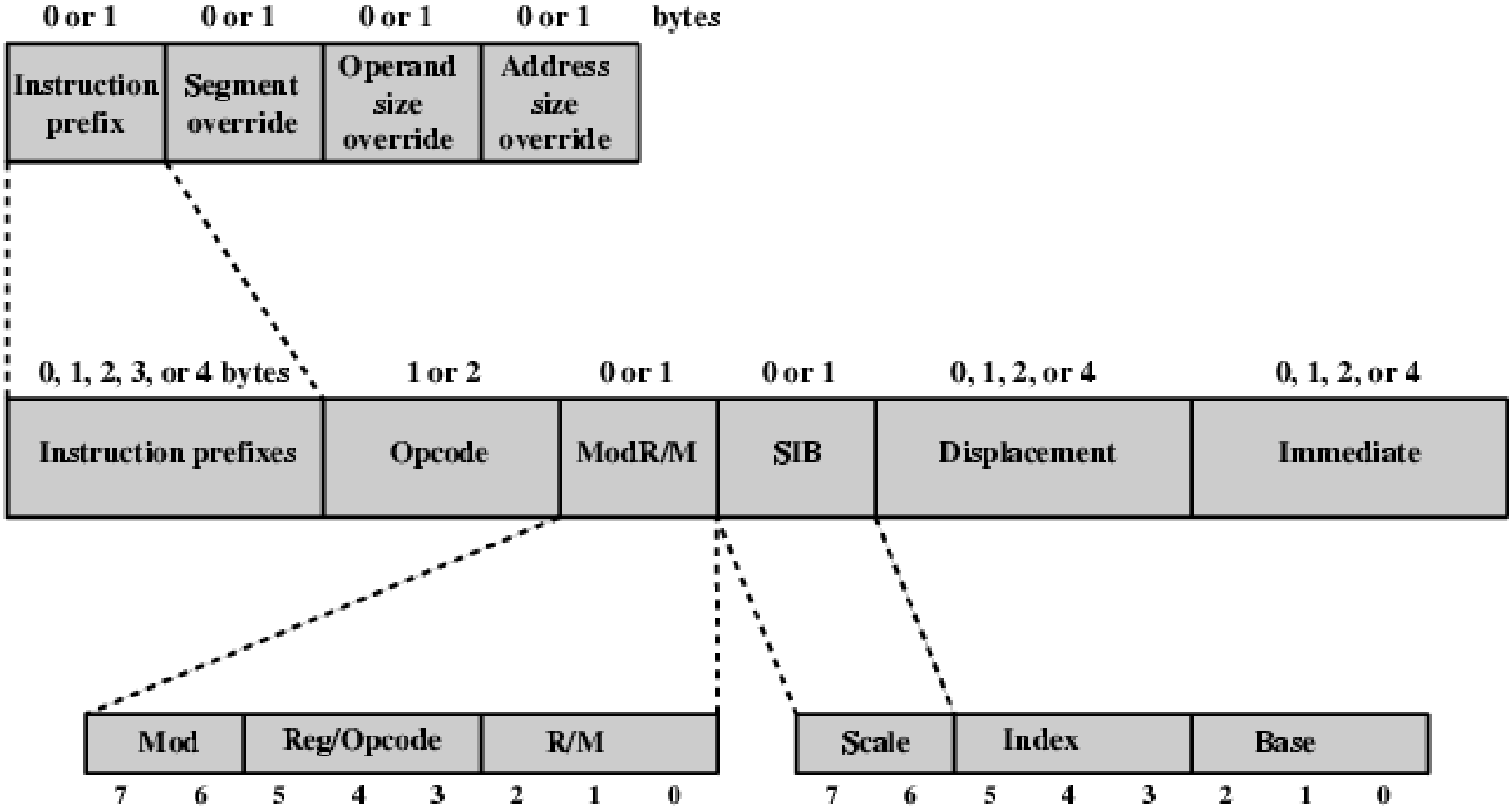
# Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed
- Trade off between powerful instruction repertoire and saving space

# Allocation of Bits

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

# x86 Instruction Format



# Instruction format:8086

*Assembly Instruction format*

General format

| mnemonic | operand(s) | ;comments |
|---|---|---|

| MOV | destination,source | ;copy source operand to destination |
|---|---|---|

Example:

```
MOV    DX,CX
```

Example 2:

```
MOV    CL,55H
MOV    DL,CL
MOV    AH,DL
MOV    AL,AH
MOV    BH,CL
MOV    CH,BH
```

| | |
|---|---|
| AH | AL |
| BH | BL |
| CH | CL |
| DH | DL |

# Instruction format:8086

## What if …

**MOV AL,DX**

Rule #1:
moving a value that is too large into a register will cause an error

```
MOV   BL,7F2H        ;Illegal: 7F2H is larger than 8 bits
MOV   AX,2FE456H     ;Illegal
```

Rule #2:
Data can be moved directly into nonsegment registers only
(Values cannot be loaded directly into any segment register.
To load a value into a segment register, first load it to a
nonsegment register and then move it to the segment register.)

```
MOV   AX,2345H        MOV   DI,1400H
MOV   DS,AX           MOV   ES,DI
```

Rule #3:
If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

MOV BX, 5

BX =0005
BH = 00, BL = 05

# Program Segments: Code segment

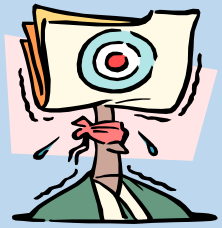
The 8086 fetches the instructions (opcodes and operands) from the code segments.

The 8086 address types:

- Physical address
- Offset address
- Logical address

- Physical address
  - 20-bit address that is actually put on the address pins of 8086
  - Decoded by the memory interfacing circuitry
  - A range of 00000H to FFFFFH
  - It is the actual physical location in RAM or ROM within 1 MB mem. range
- Offset address
  - A location within a 64KB segment range
  - A range of 0000H to FFFFH
- Logical address
  - consist of a segment value and an offset address

# Program Segments: Code segment

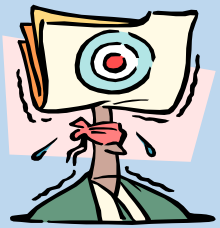


Define the addresses for the 8086 when it fetches the instructions (opcodes and operands) from the code segments.

- Logical address:
  - Consist of a **CS** (code segment) and an **IP** (instruction pointer)
    format is **CS:IP**
- Offset address
  - **IP** contains the offset address
- Physical address
  - generated by shifting the **CS** left one hex digit and then adding it to the **IP**
  - the resulting 20-bit address is called the physical address

*give me some numbers…ok*

# Program Segments: Code segment



Suppose we have:

| | |
|---|---|
| CS | 2500 |
| IP | 95F3 |

- Logical address:
  - Consist of a **CS** (code segment) and an **IP** (instruction pointer)
    format is **CS:IP** **2500:95F3H**

- Offset address
  - **IP** contains the offset address which is **95F3H**

- Physical address
  - generated by shifting the **CS** left one hex digit and then adding it to the **IP**
    **25000 + 95F3 = 2E5F3H**

# Endian conversion

- Little endian conversion:

  In the case of 16-bit data, the low byte goes to the low memory location and the high byte goes to the high memory address. (Intel, Digital VAX)

- Big endian conversion:

  The high byte goes to low address. (Motorola)

*Example:*

Suppose DS:6826 = 48, DS:6827 = 22,

Show the contents of register BX in the instruction **MOV BX,[6826]**

**Little endian conversion: BL = 48H, and BH = 22H**

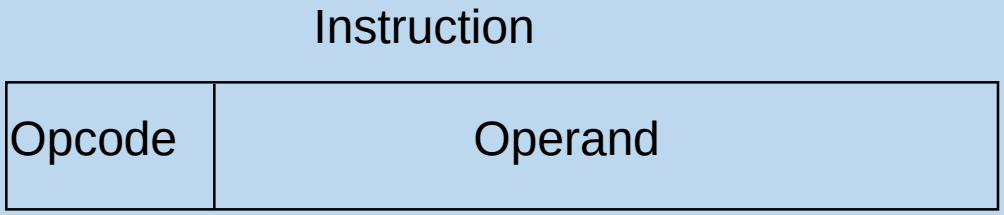
# Addressing Modes

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

# Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

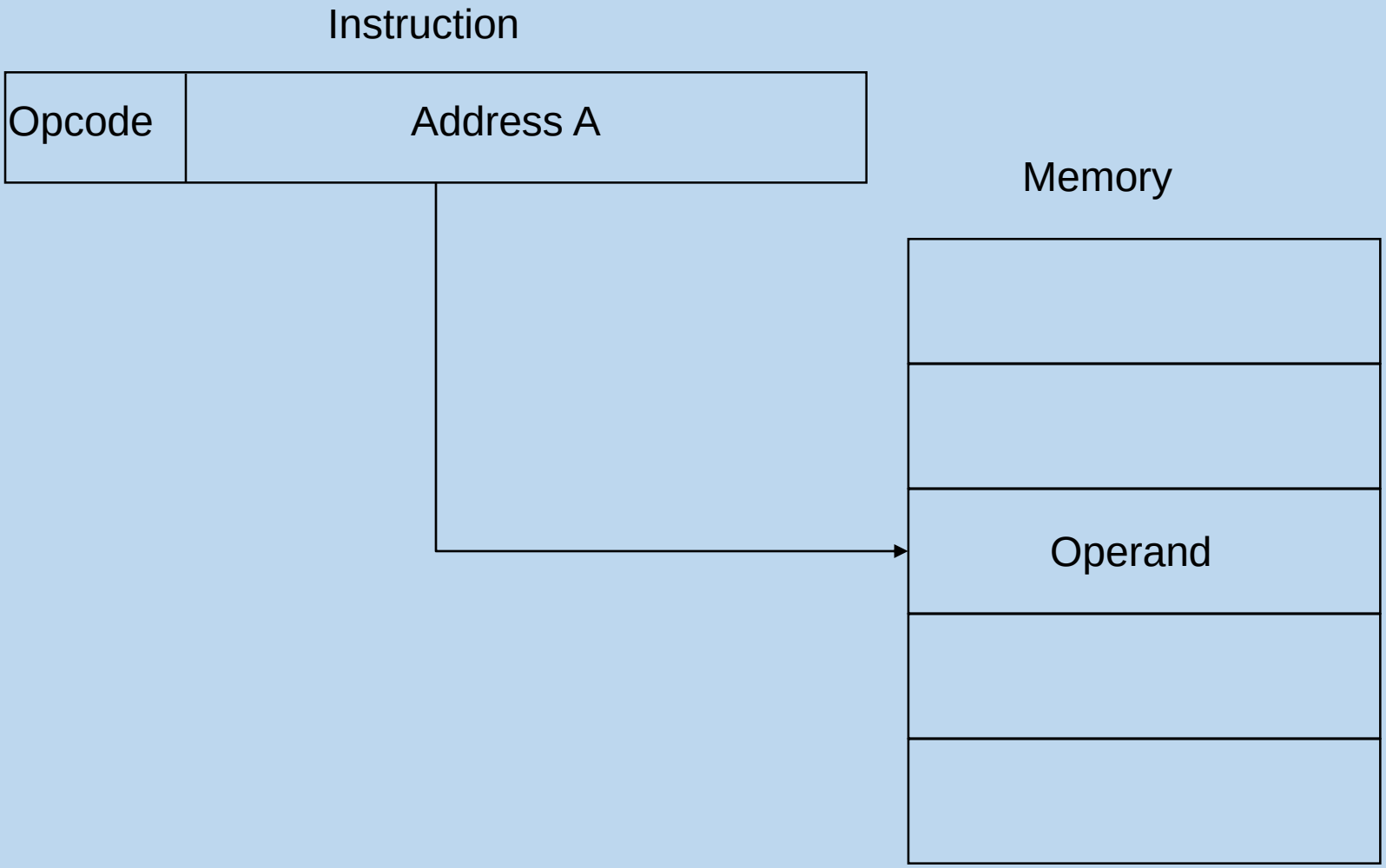
# Immediate Addressing Diagram



# Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g.  ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

# Direct Addressing Diagram



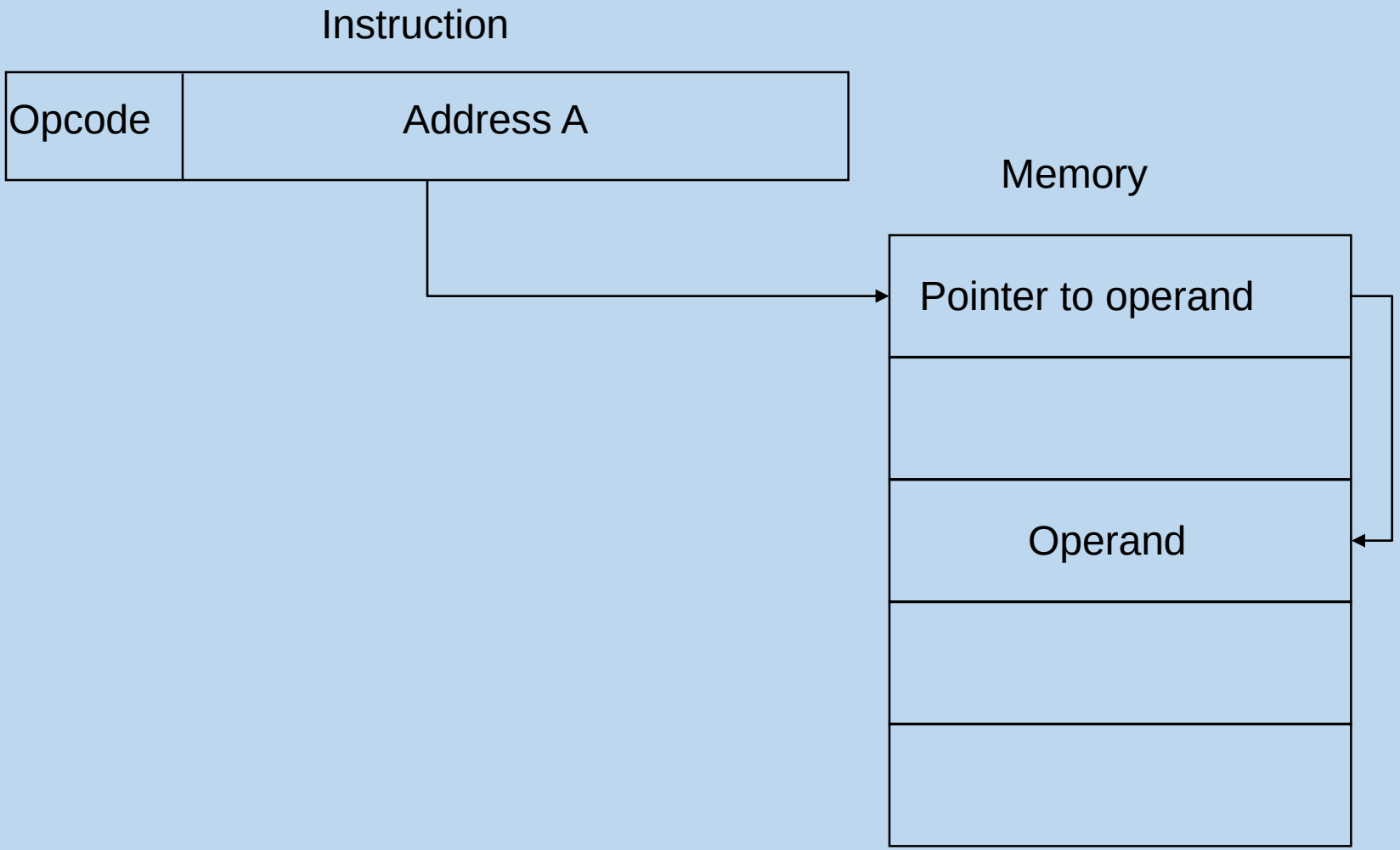
# Indirect Addressing (1)

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

# Indirect Addressing (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

# Indirect Addressing Diagram



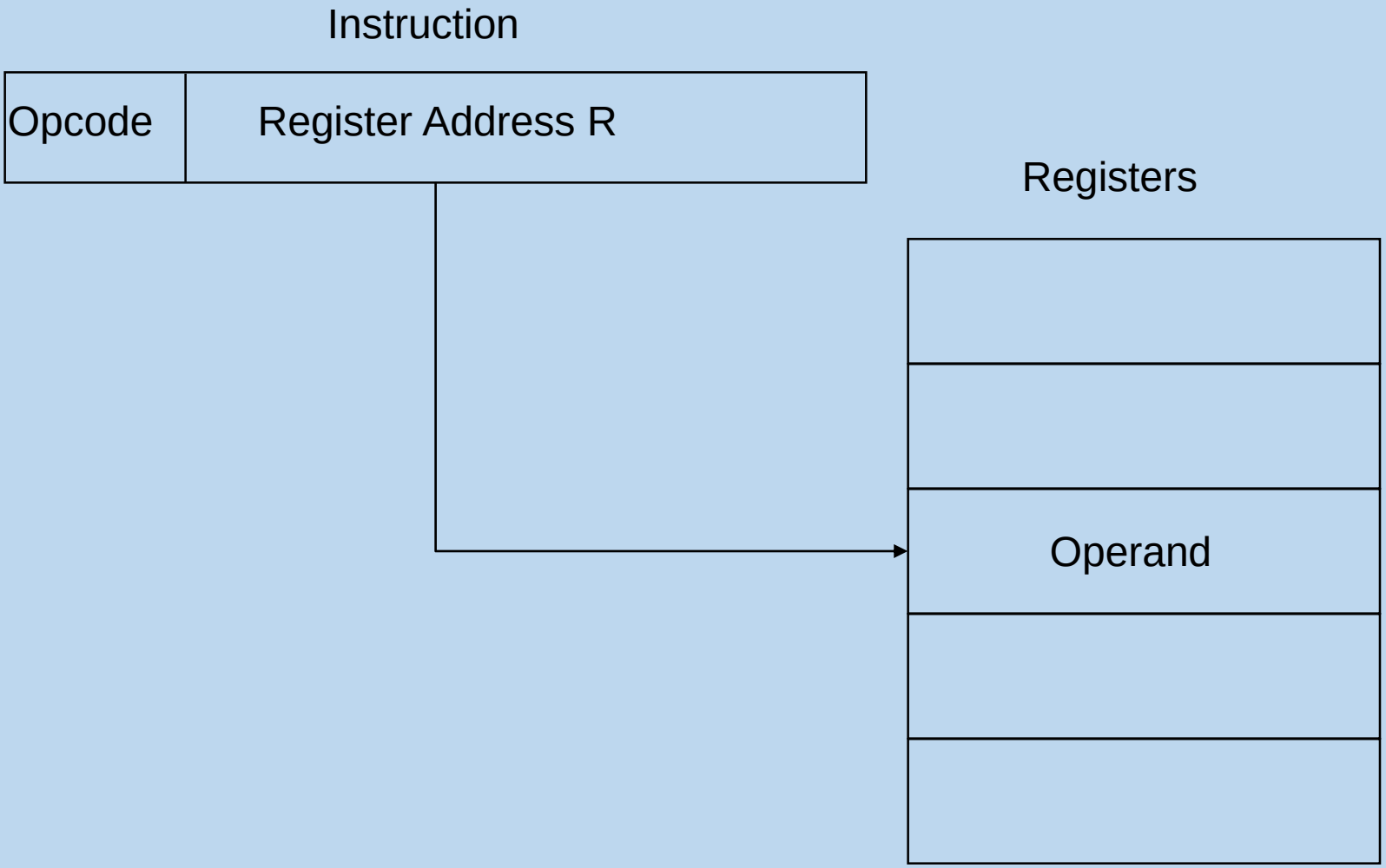
# Register Addressing (1)

- 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

# Register Addressing (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

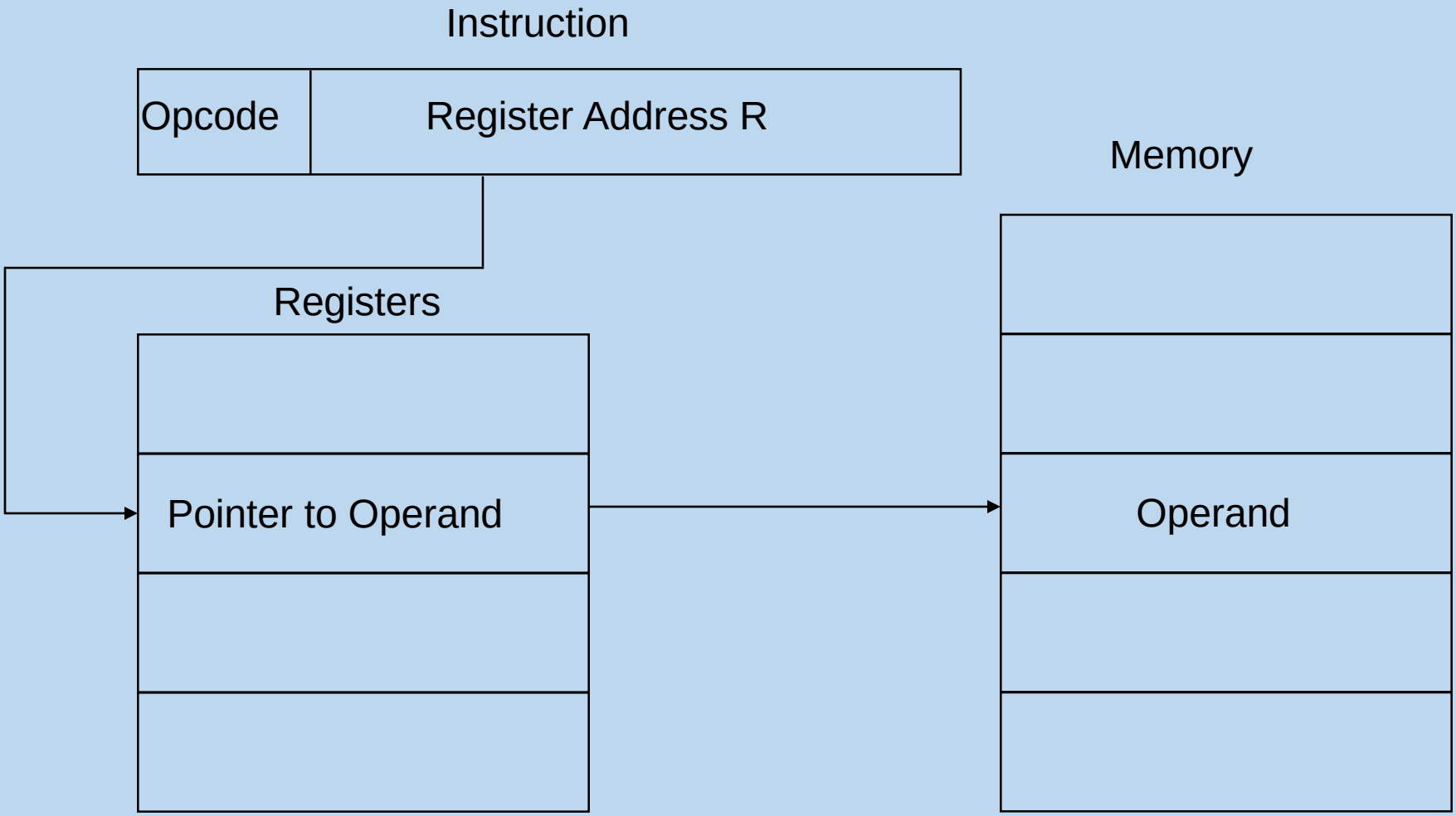
# Register Addressing Diagram



# Register Indirect Addressing

- C.f. indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space ($2^n$)
- One fewer memory access than indirect addressing

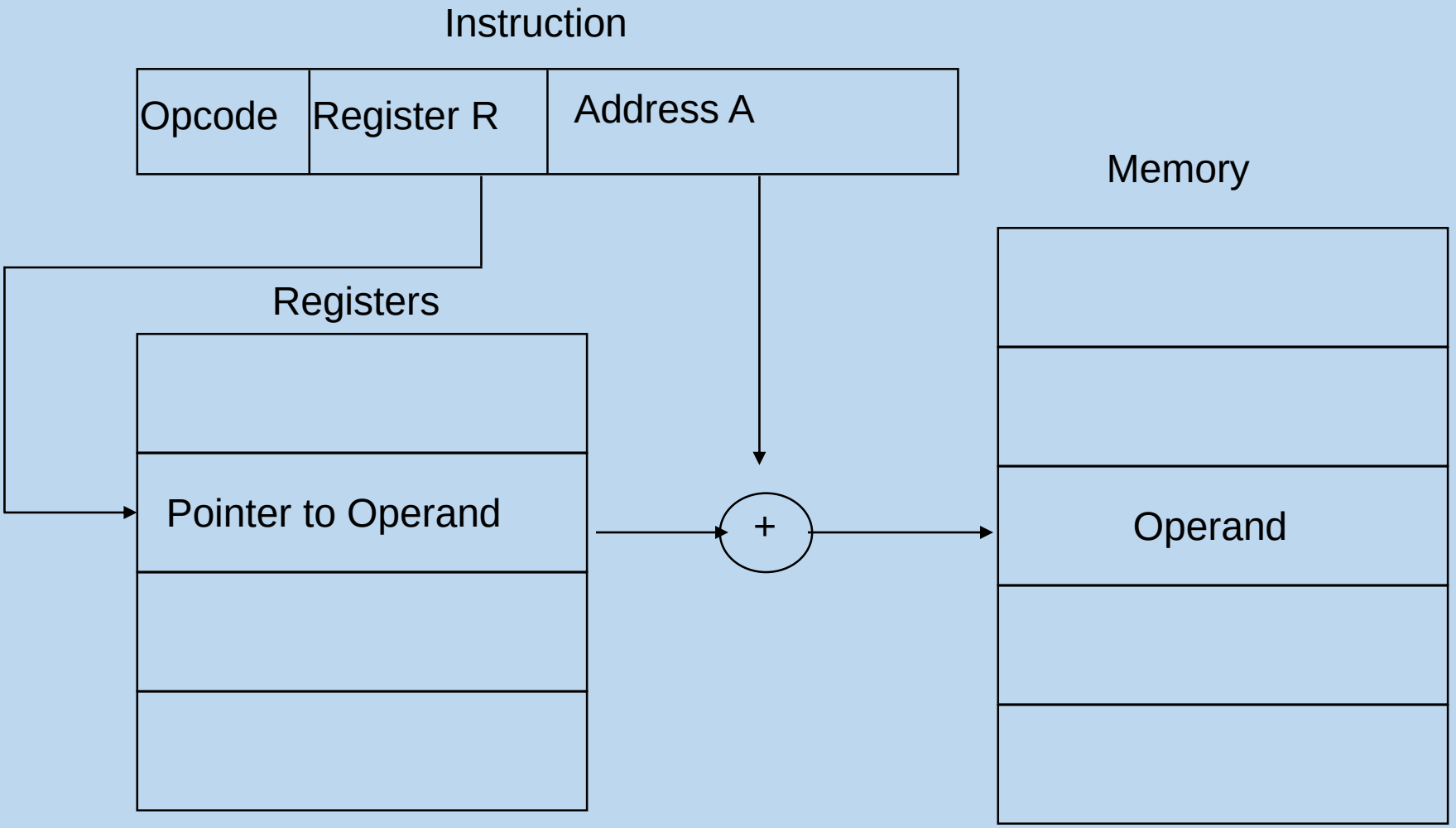
# Register Indirect Addressing Diagram



# Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

# Displacement Addressing Diagram



# Relative Addressing

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e. get operand from A cells from current location pointed to by PC
- c.f locality of reference & cache usage

# Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86

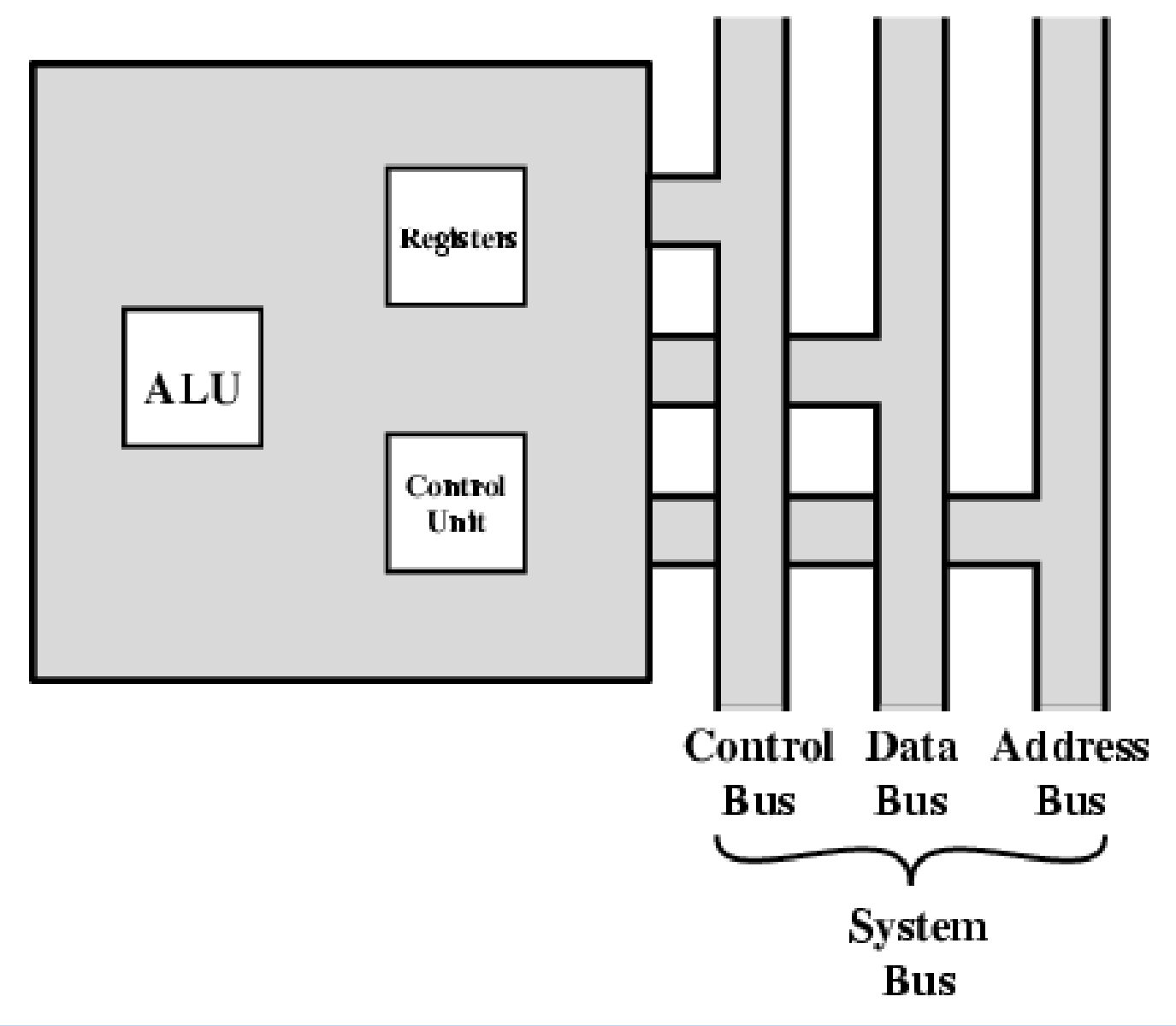
# Indexed Addressing

- A = base
- R = displacement
- EA = A + R
- Good for accessing arrays
  - EA = A + R
  - R++

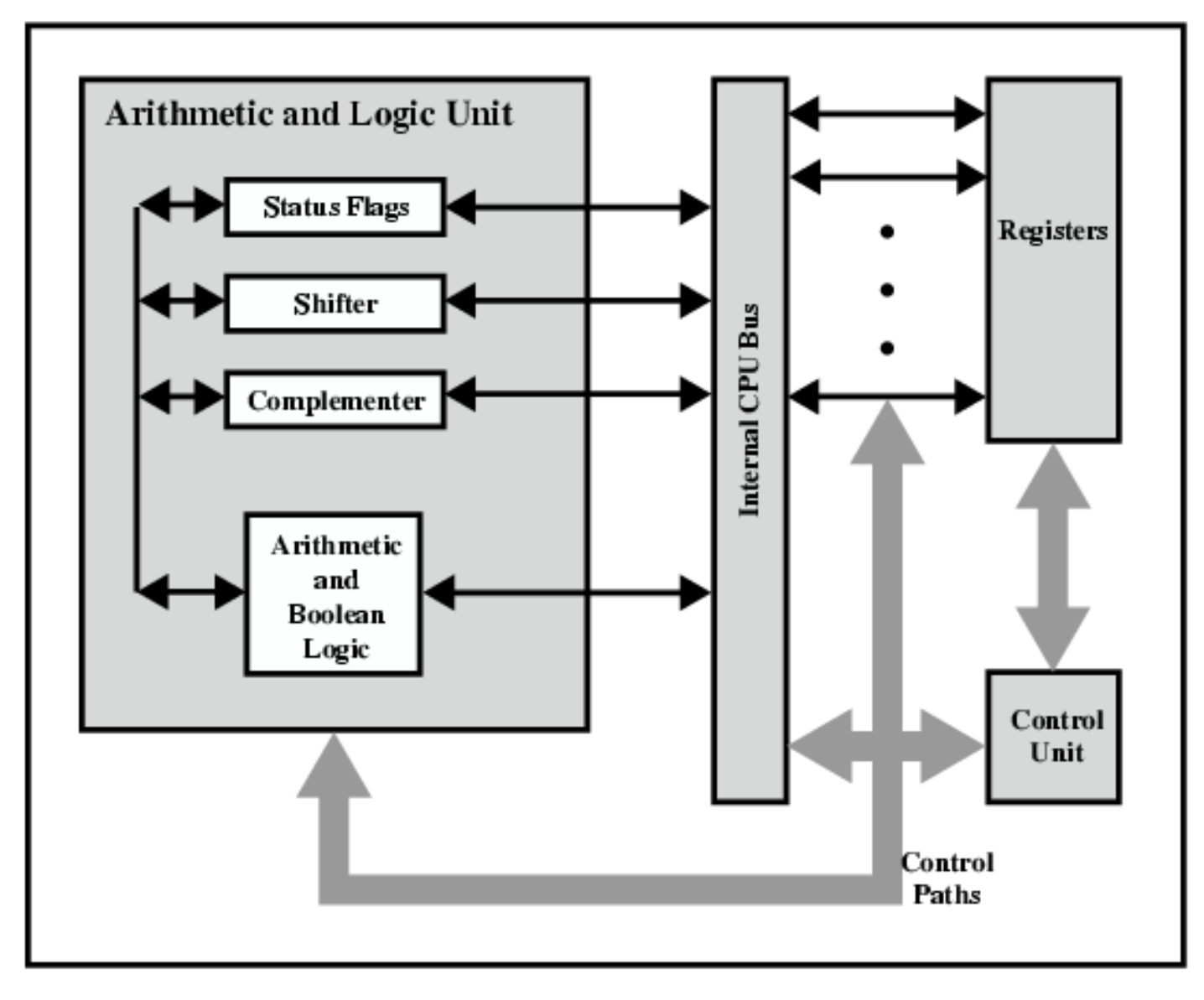
# CPU Structure

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

# CPU With Systems Bus



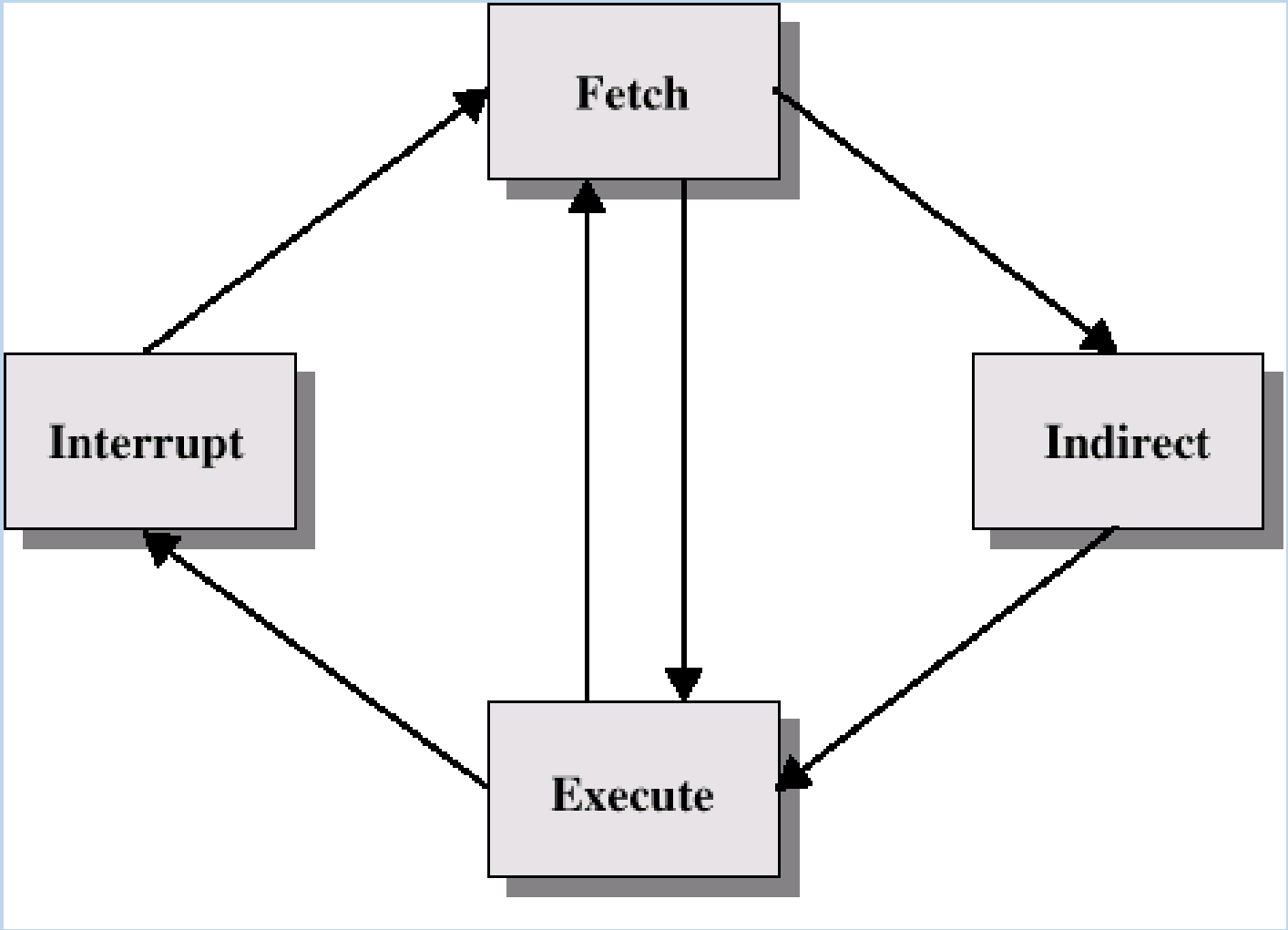
# CPU Internal Structure



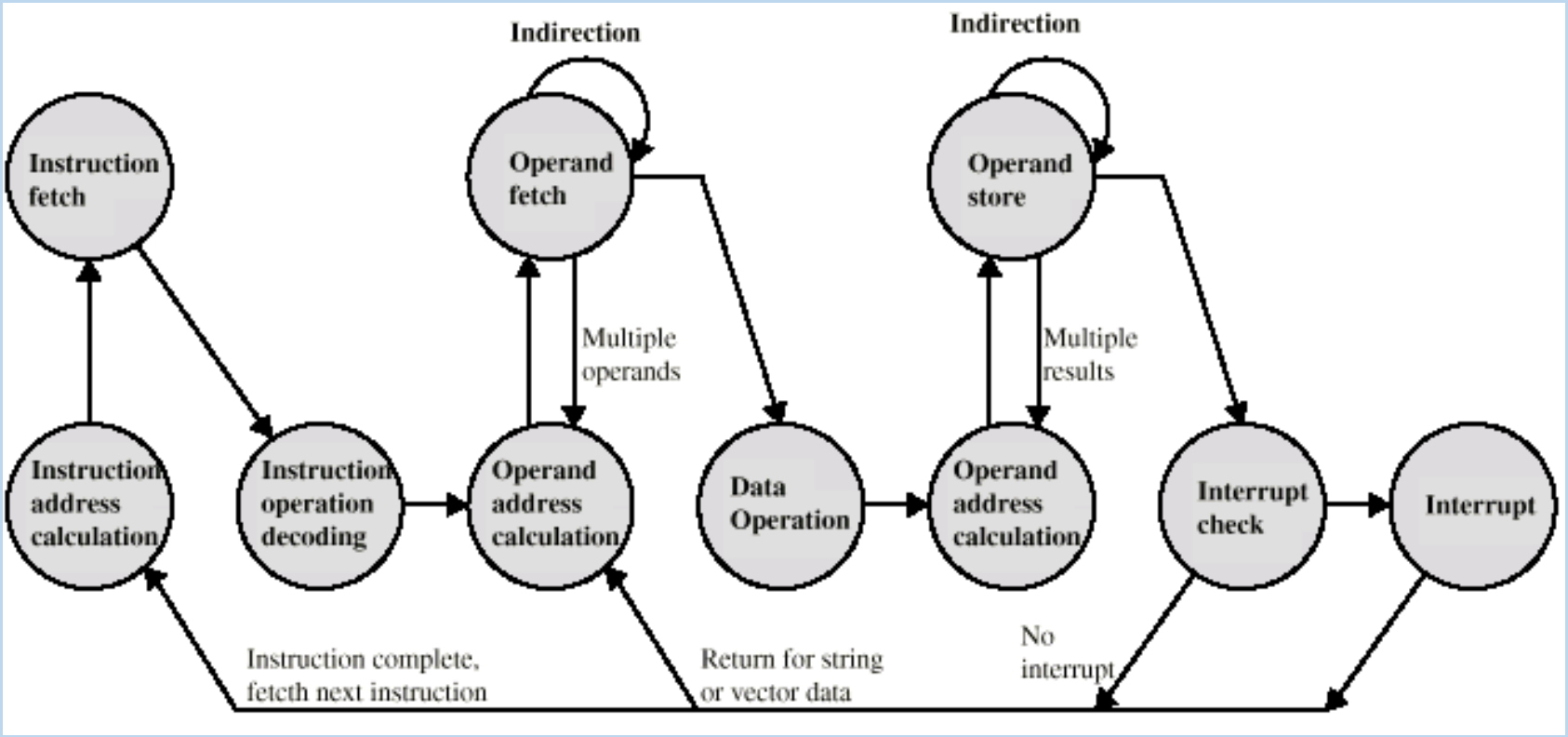
# Indirect Cycle

- 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)

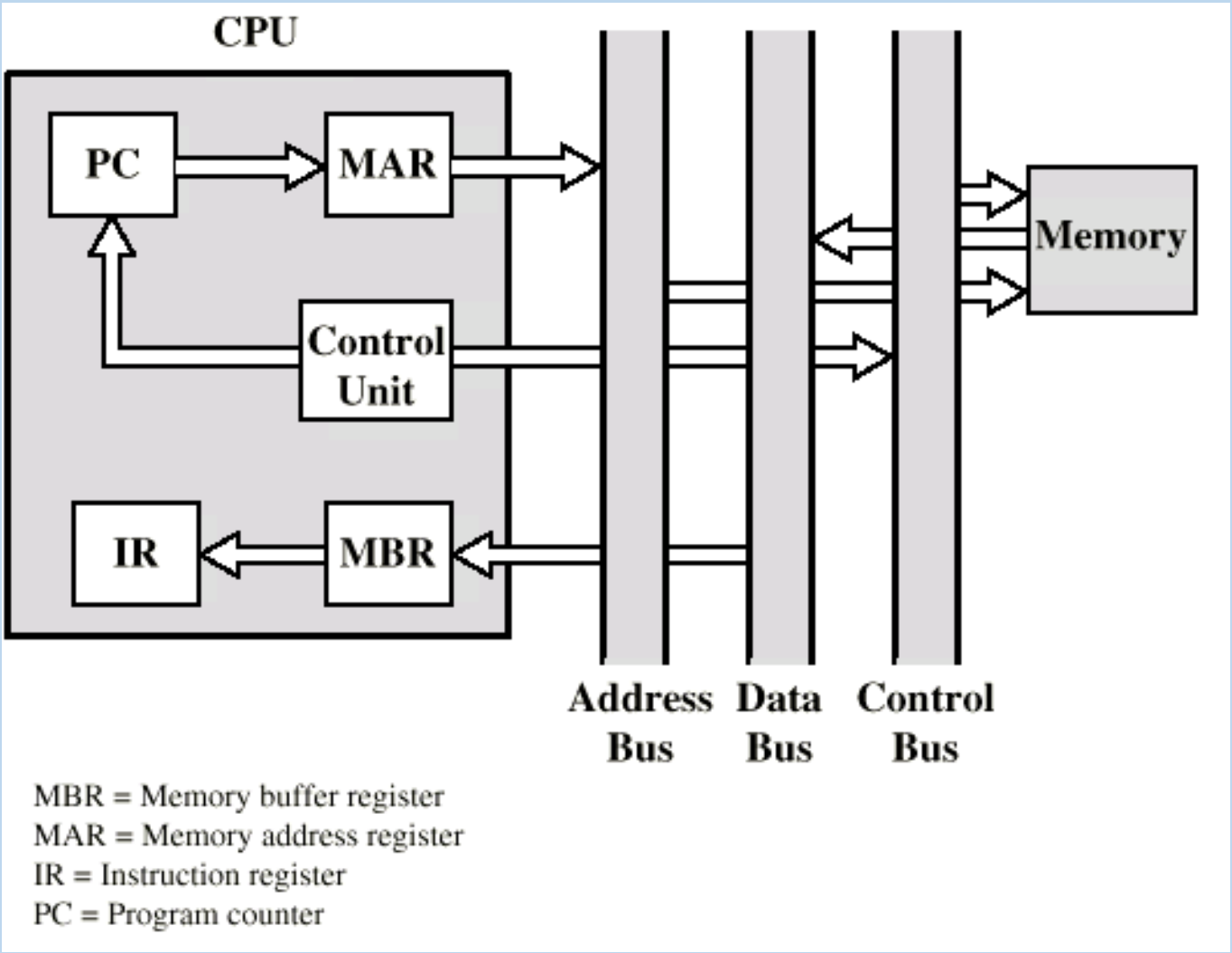
- Depends on CPU design
- In general:

- 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

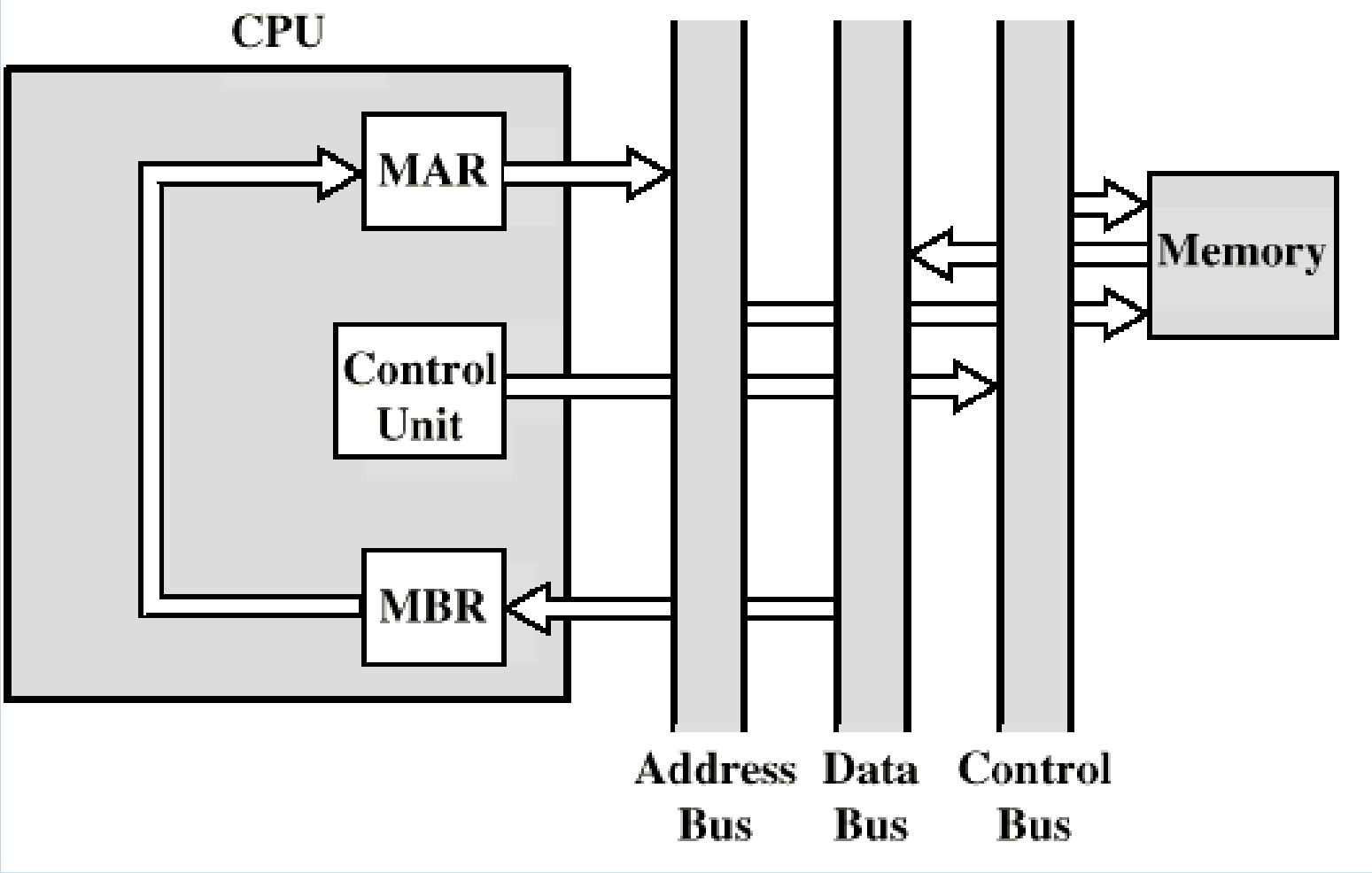
# Data Flow (Data Fetch)

- 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)



# Data Flow (Indirect Diagram)



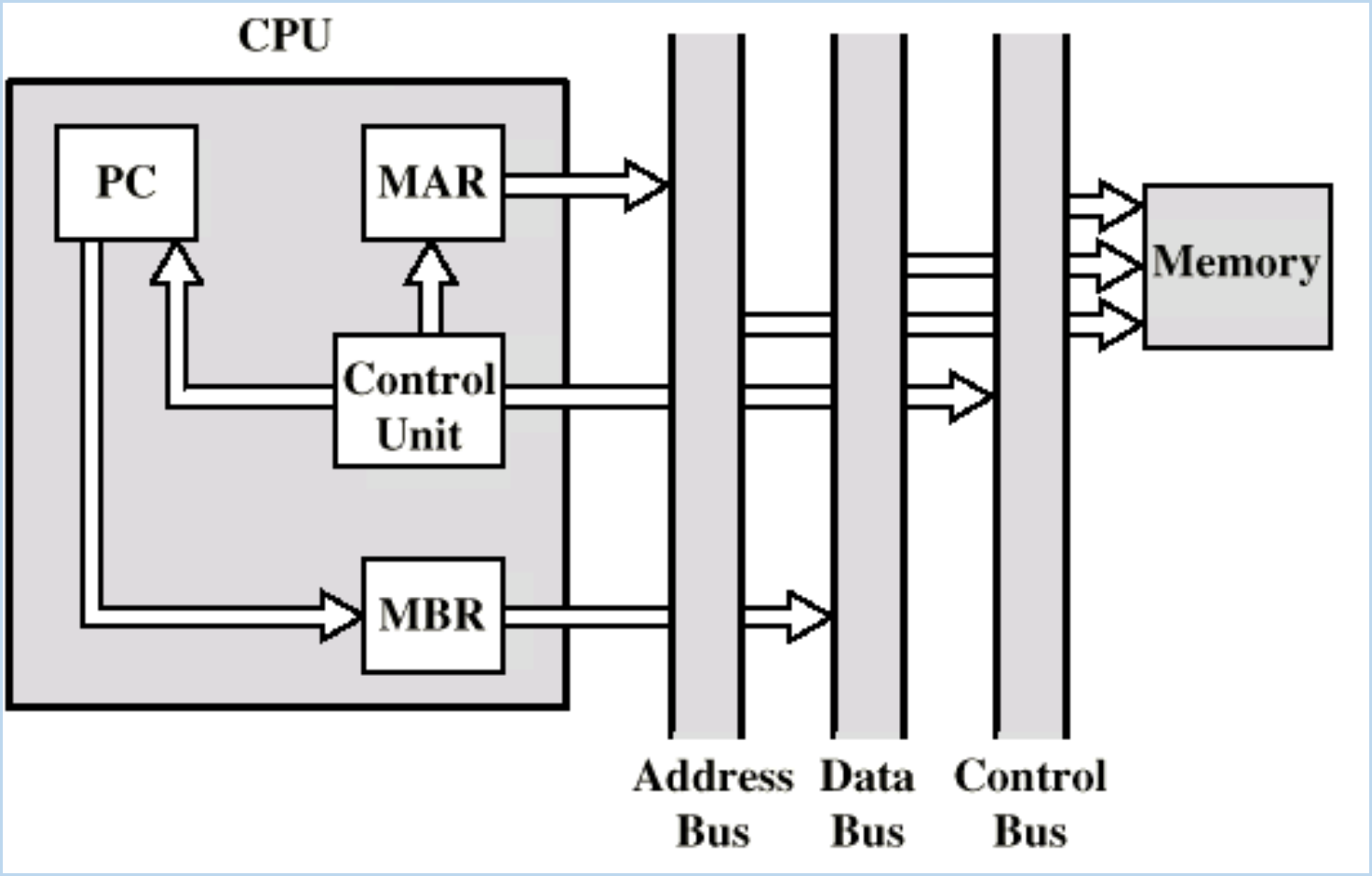
# Data Flow (Execute)

- 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)



# Prefetch

- Fetch accessing main memory
- Execution usually does not access main memory
- Can fetch next instruction during execution of current instruction
- Called instruction prefetch