Microprocessor Instructions & Communication

Chapter 2

Microprocessor Instructions & Communication

- **Instruction** Set ,Mnemonics
- Basic Instruction Types
- Addressing modes
- Microprocessor I/O connecting I/O put to Microprocessor
- Polling and Interrupts
- Interrupt and DM Controllers.

Microprocessor Instruction Set Classification

- Transfer
  - Move
- Arithmetic
  - Add / Subtract
  - Mul / Div, etc.
- Control
  - Jump
  - Call / Return, etc.

Data transfer : Move

- MOV Dest, Src
  - `MOV reg, reg` `reg <- reg`
  - `MOV reg, mem` `reg <- mem`
  - `MOV mem, reg` `mem <- reg`
  - `MOV reg, imm` `reg <- imm`
  - `MOV mem, imm` `mem <- imm`

- There is no move mem<-mem instruction.

Move limitation

- Both operand must be in the same size.
- There is no instruction to put immediate value directly to segment register. Have to use accumulator (AX) to accomplish this.
- To put immediate value directly to memory, we have to specify its size. (Byte/Word PTR)

Move *(MOV)* Example

- `MOV AX,100h`
- `MOV BX,AX`
- `MOV DX,BX`

- `MOV AX,1234h`
- `MOV DX,5678h`
- `MOV AL,DL`
- `MOV BH,DH`

MOV Example

- `MOV AX,1000h`
- `MOV [100h],AX`
- `MOV BX,[100h]`

- `MOV BYTE PTR [200h],10h`
- `MOV WORD PTR [300h],10h`

- `MOV AX,2300h`
- `MOV DS,AX`

MOV : 16 / 8 Bit register

```
MOV  AX,1000h
```

AX	AH	AL
1000h	10h	00h

```
MOV  AL,3Ah
```

AX	AH	AL
103Ah	10h	3Ah

```
MOV  AH,AL
```

AX	AH	AL
3A3Ah	3Ah	3Ah

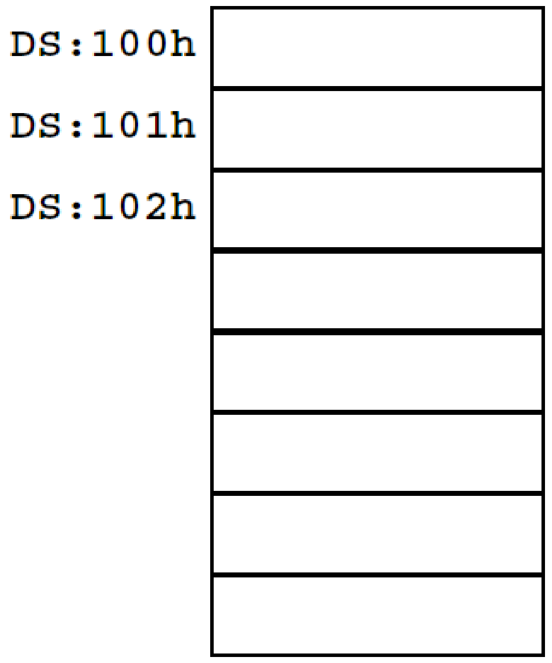
```
MOV  AX,234h
```

AX	AH	AL
234h	02h	34h

- To move value between registers, their size must be the same.

MOV : Memory

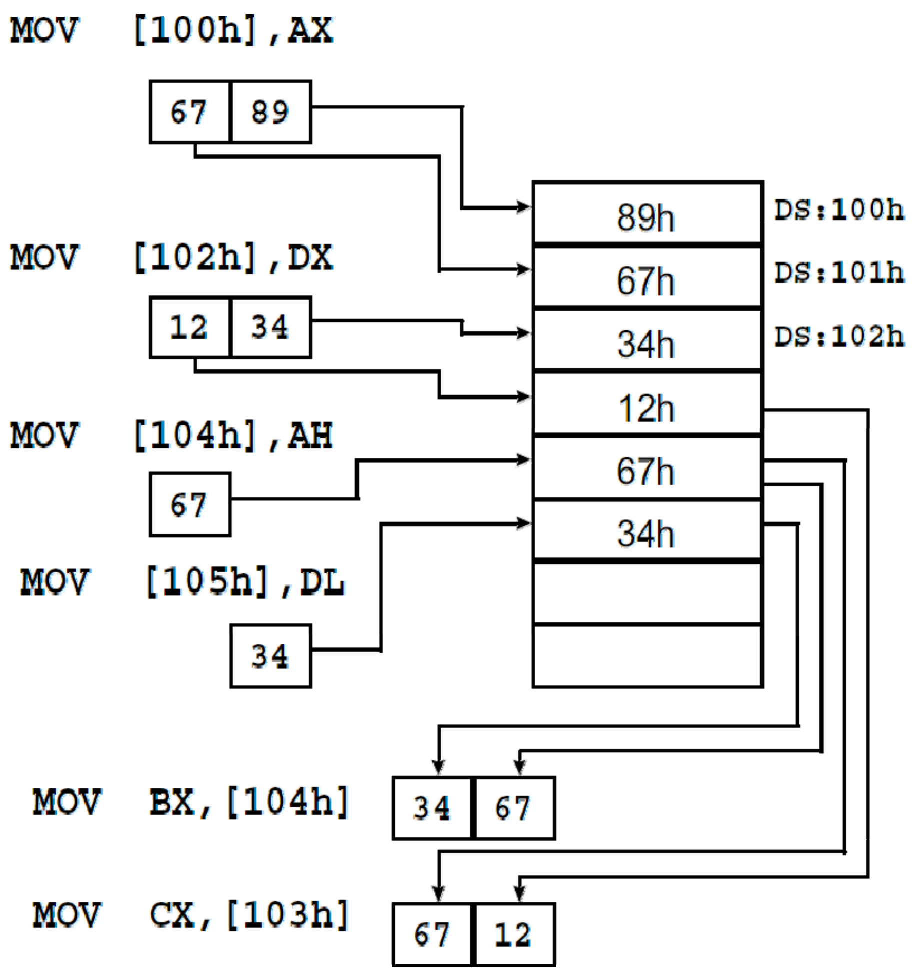
```
MOV   AX,6789h
MOV   DX,1234h
MOV    [100h],AX
MOV    [102h],DX
MOV    [104h],AH
MOV    [105h],DL
MOV   BX,[104h]
MOV   CX,[103h]
MOV    [106h],CL
```



- Given only offset where to put value, it will be automatically select DS as the segment register.

Byte ordering : Little endian

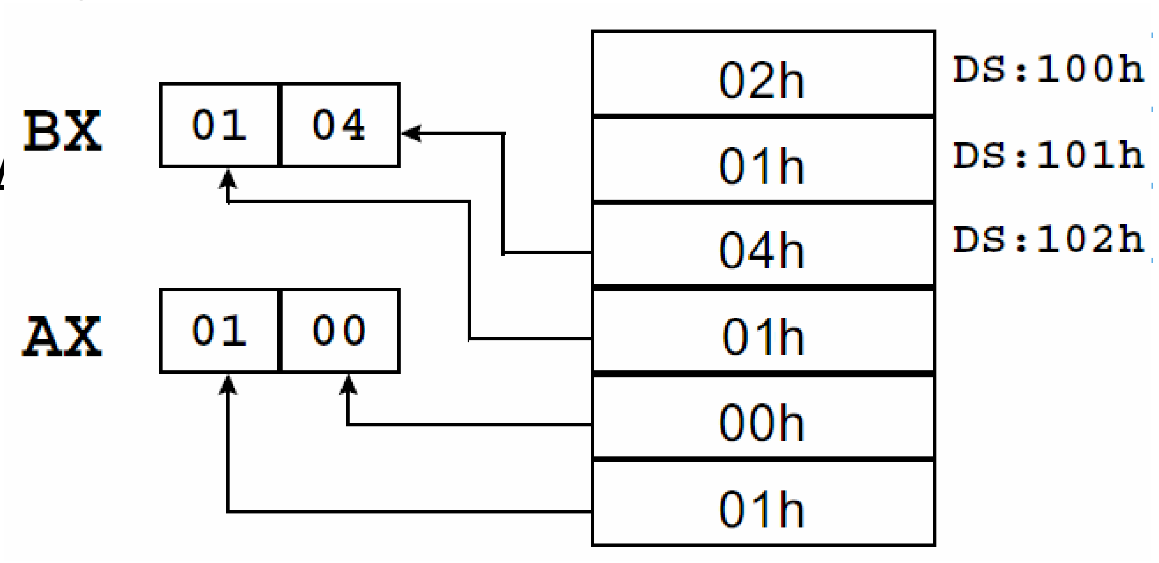
- Since, x86's byte ordering is little endian.
- Therefore, the LSB will be placed at lowest address and MSB will be placed at highest address.



Displacement

- We can use BX (Base) register to point a place of memory.
- Both register direct or displacement.
- A

```
MOV   AX,102h
MOV   BX,100h
MOV   CX,4004h
MOV   DX,1201h
MOV   [BX],AX
MOV   [BX+2],CX
      [BX+3],DX
      [BX+4],BX
      BX,[102h]
      AX,[BX]
```



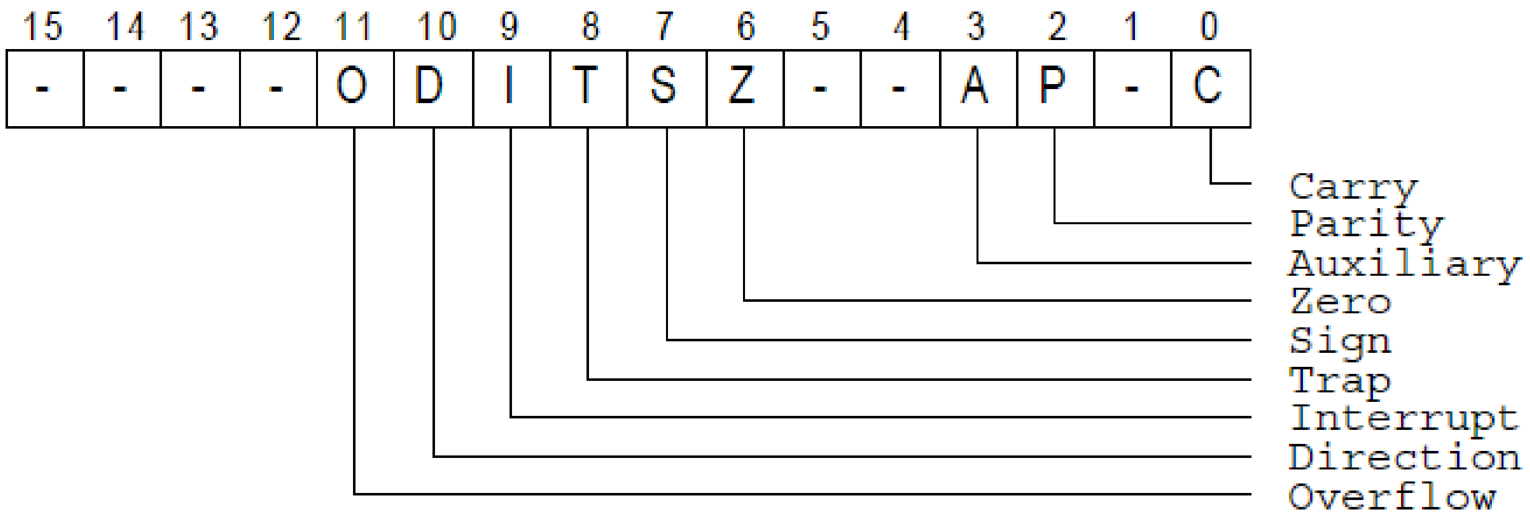
What is the result of …

- MOV [100h] , 10h
- Address 100 = 10h
- What about address 101?
- Word or Byte?
  - `MOV WORD PTR [100h], 10h`
  - `MOV BYTE PTR [100h], 10h`

- What about MOV [100h], AX ?

Flag

- 8086 has 16 bit flag to indicate the status of final arithmetic result.



Zero Flag

- The zero flag will be set (1) whenever the result is zero.

EX

```
MOV   AL,10h        Z=?
ADD   AL,E0h        Z=1      AL=0
ADD   AL,20h        Z=0      AL=20h
SUB   AL,10h        Z=0      AL=10h
SUB   AL,10h        Z=1      AL=0
```

Parity flag

- The parity flag will be set whenever the number of bit “1” are even.

EX

```
MOV   AL,14h        P=?
ADD   AL,20h        P=0       AL=34h
ADD   AL,10h        P=1       AL=44h
SUB   AL,8h         P=1       AL=3Ch
SUB   AL,10h        P=0       AL=2Ch
```

Carry flag

- Carry flag will be set whenever there is a carry or borrow (only with unsigned operation).

EX

```
MOV   AL,77h        C=?
ADD   AL,50h        C=0      AL=C7h
ADD   AL,50h        C=1      AL=17h
SUB   AL,A0h        C=1      AL=77h
ADD   AL,27h        C=0      AL=9Eh
```

Overflow flag

- Overflow flag will be set whenever the result is overflow (with signed operation).

EX

```
MOV   AL,77h       O=?
ADD   AL,50h       O=1      AL=C7h
ADD   AL,50h       O=0      AL=17h
SUB   AL,A0h       O=0      AL=77h
ADD   AL,27h       O=1      AL=9Eh
```

Sign flag

- Sign flag will be set whenever the result is negative with signed operation.

EX

```
MOV   AL,77h       S=?
ADD   AL,50h       S=1      AL=C7h
ADD   AL,50h       S=0      AL=17h
SUB   AL,A0h       S=0      AL=77h
ADD   AL,27h       S=1      AL=9Eh
```

More flag

- Auxiliary flag will be set when the result of BCD operation need to be adjusted.
- Direction flag is used to specify direction (increment/decrement index register) in string operation.
- Trap flag is used to interrupt CPU after each operation.
- Interrupt flag is used to enable/disable hardware interrupt.

Flag set/reset instructions

Carry flag	STC / CLC
Direction flag	STD / CLD
Interrupt flag	STI / CLI

Increment - Decrement

- INC / DEC
  - INC register DEC register
  - INC memory DEC memory

- EX.
  - INC AX
  - DEC BL
  - How can we increment a byte of memory?
    - INC ??? [100]

Add

• ADD reg, imm	ADC reg, imm
• ADD reg, mem	ADC reg, mem
• ADD reg, reg	ADC reg, reg
• ADD mem, imm	ADC mem, imm
• ADD mem, reg	ADC mem, reg

EX. ADD

```
• MOV AL, 10h
• ADD AL, 20h          ;AL =   30h
• MOV BX, 200h         ;BX = 0200h
• MOV WORD PTR [BX], 10h
• ADD WORD PTR [BX], 70h
• MOV AH, 89h          ;AX = 8930h
• ADD AX, 9876h        ;AX = 21A6h
• ADC BX, 01h          ;BX = 0202h ?
```

Subtract

• SUB reg, imm	SBB reg, imm
• SUB reg, mem	SBB reg, mem
• SUB reg, reg	SBB reg, reg
• SUB mem, imm	SBB mem, imm
• SUB mem, reg	SBB mem, reg

Ex. SUB

```
• MOV AL, 10h
• ADD AL, 20h            ;AL =   30h
• MOV BX, 200h           ;BX = 0200h
• MOV WORD PTR [BX], 10h
• SUB WORD PTR [BX], 70h
• MOV AH, 89h            ;AX = 8930h
• SBB AX, 0001h          ;AX = 892Eh ?
• SBB AX, 0001h          ;AX = 892Dh
```

Compare

- CMP reg, imm
- CMP reg, mem
- CMP reg, reg
- CMP mem, reg

- There is no “CMP mem, imm”

Ex. CMP

- `MOV CX, 10h`
- `CMP CX, 20h   ;Z=0,S=1,C=1,O=0`
- `MOV BX, 40h`
- `CMP BX, 40h   ;Z=1,S=0,C=0,O=0`
- `MOV AX, 30h`
- `CMP AX, 20h   ;Z=0,S=0,C=0,O=0`

Negation

- Compute 2’complement.
- Carry flag always set.

- Usage
  - NEG reg
  - NEG mem

Ex. NEG

- `MOV CX, 10h`
- `NEG CX            ; CX = 0FFF0h`
- `MOV AX,0FFFFH`
- `NEG AX            ; AX = 1`
- `MOV BX,1H`
- `NEG BX            ; BX = 0FFFFh`

Multiplication

- IMUL (Integer multiplication) unsigned multiplication
- MUL (Multiplication) signed multiplication.

– MUL reg	IMUL reg
– MUL mem	IMUL mem

- Always perform with accumulator.
- Effected flag are only over and carry flag.

8 bit multiplication

- AL is multiplicand
- AX keep the result

- MOV AL,10h ; AL = 10h
- MOV CL,13h ; CL = 13h
- IMUL CL ; AX = 0130h

16 bit multiplication

- AX is multiplicand
- DX:AX keep the result

- MOV AX,0100h ; AX = 0100h
- MOV BX,1234h ; BX = 1234h
- IMUL BX ; DX = 0012h
  ; AX = 3400h

Division

- IDIV (Integer division) unsigned division.
- DIV (Division) signed division.
  - DIV reg IDIV reg
  - DIV mem IDIV mem
- Always perform with accumulator.
- Effected flag are only over and carry flag.

8 bit division

- AL is dividend
- AL keep the result
- AH keep the remainder

- MOV AX, 0017h
- MOV BX, 0010h
- DIV BL ; AX = 0701

16 bit multiplication

- DX:AX dividend.
- AX keep the result, DX keep the remainder.

```
• MOV AX,4022h                 ;
• MOV DX,0000h                 ;
• MOV CX,1000h                 ;
• DIV CX                  ;AX = 0004
                               ;DX = 0022
```

Conversion

- Byte to Word : CBW
  - Signed convert AL -> AX

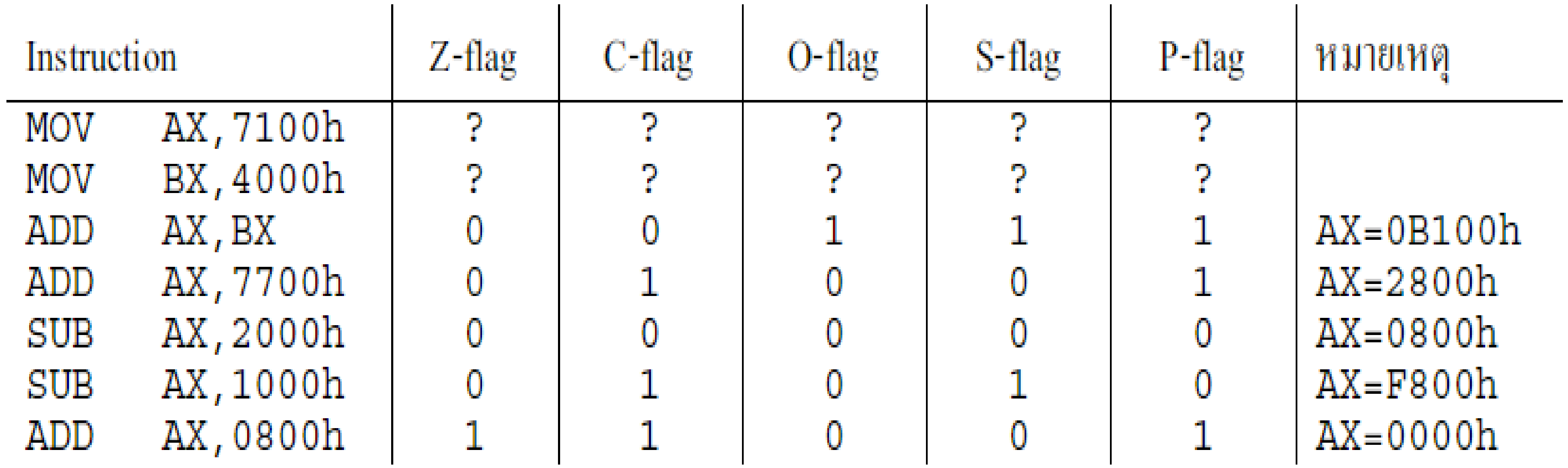
- Word to Double word : CWD
  - Signed convert AX -> DX:AX

Ex. Conversion

```
• MOV AL,22h
• CBW                ; AX=0022h
• MOV AL,F0h
• CBW                ; AX=FFF0h
• MOV AX, 3422h
• CWD                ; DX=0000h
                     ; AX=3422h
```

Instruction	Flag affected				
	Z-flag	C-flag	S-flag	O-flag	A-flag
`ADD`	*yes*	*yes*	*yes*	*yes*	*yes*
`ADC`	*yes*	*yes*	*yes*	*yes*	*yes*
`SUB`	*yes*	*yes*	*yes*	*yes*	*yes*
`SBB`	*yes*	*yes*	*yes*	*yes*	*yes*
`INC`	*yes*	***no***	*yes*	*yes*	*yes*
`DEC`	*yes*	***no***	*yes*	*yes*	*yes*
`NEG`	*yes*	*yes*	*yes*	*yes*	*yes*
`CMP`	*yes*	*yes*	*yes*	*yes*	*yes*
`MUL`	*no*	*yes*	*no*	*yes*	*no*
`IMUL`	*no*	*yes*	*no*	*yes*	*no*
`DIV`	*no*	*no*	*no*	*no*	*no*
`IDIV`	*no*	*no*	*no*	*no*	*no*
`CBW`	*no*	*no*	*no*	*no*	*no*
`CWD`	*no*	*no*	*no*	*no*	*no*

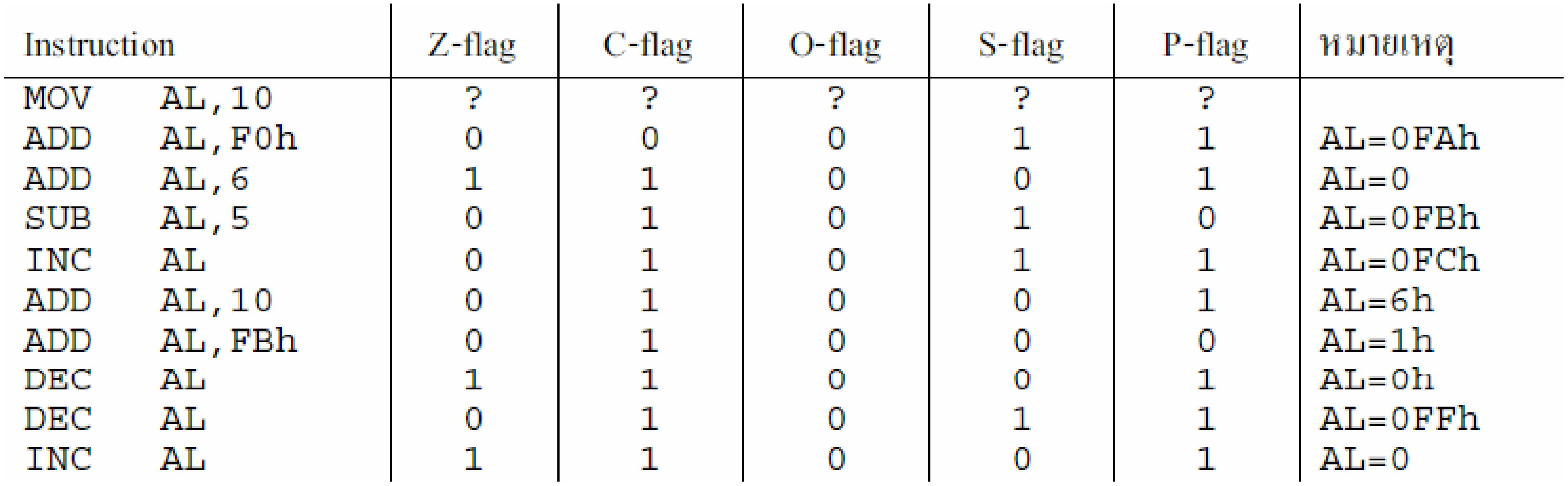
Example about flag with arithmetic



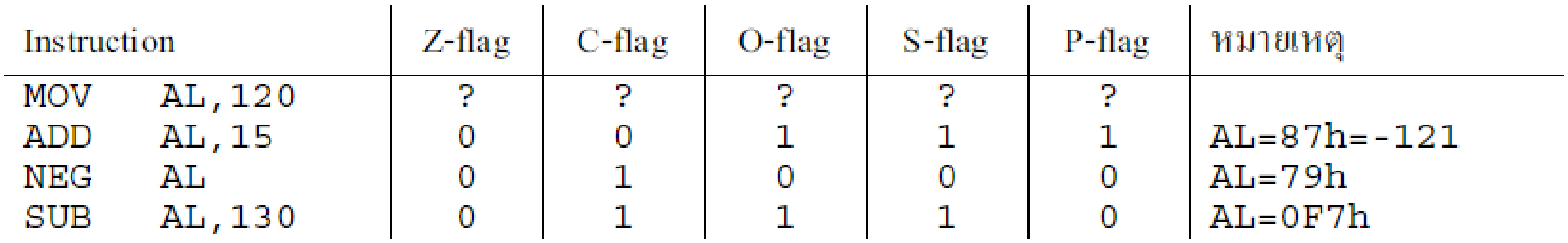
Instruction	Z-flag	C-flag	O-flag	S-flag	P-flag	หมายเหตุ
MOV AX,7100h	?	?	?	?	?	
MOV BX,4000h	?	?	?	?	?	
ADD AX,BX	0	0	1	1	1	AX=0B100h
ADD AX,7700h	0	1	0	0	1	AX=2800h
SUB AX,2000h	0	0	0	0	0	AX=0800h
SUB AX,1000h	0	1	0	1	0	AX=F800h
ADD AX,0800h	1	1	0	0	1	AX=0000h

- เอกสารอ้างอิง
  - เอกสารประกอบการสอนวิชา 204221 องค์ประกอบคอมพิวเตอร์และภาษาแอสเซมบลี้ มหาวิทยาลัยเกษตรศาสตร์, 1994

Example about flag with arithmetic



Instruction	Z-flag	C-flag	O-flag	S-flag	P-flag	หมายเหตุ
MOV AL,10	?	?	?	?	?	
ADD AL,F0h	0	0	0	1	1	AL=0FAh
ADD AL,6	1	1	0	0	1	AL=0
SUB AL,5	0	1	0	1	0	AL=0FBh
INC AL	0	1	0	1	1	AL=0FCh
ADD AL,10	0	1	0	0	1	AL=6h
ADD AL,FBh	0	1	0	0	0	AL=1h
DEC AL	1	1	0	0	1	AL=0h
DEC AL	0	1	0	1	1	AL=0FFh
INC AL	1	1	0	0	1	AL=0



Instruction	Z-flag	C-flag	O-flag	S-flag	P-flag	หมายเหตุ
MOV AL,120	?	?	?	?	?	
ADD AL,15	0	0	1	1	1	AL=87h=-121
NEG AL	0	1	0	0	0	AL=79h
SUB AL,130	0	1	1	1	0	AL=0F7h

NEG -> Carry flag always 1, INC/DEC does not effect any flag

Jump and Loops

- Control structures
  - Selection
  - Repetition / Loop

- Jxx Label

If ah=10, if ah>=10

```
       cmp    ah,10         ;เปรียบเทียบ ah กับ 10
       jz     lab1          ;ถ้าเท่ากันให้กระโดดไปที่lab1
       mov    bx,2
lab1:  add    cx,10

       cmp    ah,10         ;เปรียบเทียบ ah กับ 10
       jge    tenup         ;ถ้ามากกว่าหรือเท่ากับให้กระโดดไปที่tenup
       add    dl,'0'
       jmp    endif         ;กระโดดไปที่ endif
tenup:
       add    dl,'A'
endif:
```

Get ‘Q’ and Print ASCII code.

```
getonechar:
        mov     ah,1            ; ใช้บริการหมายเลข 1 : อ่านอักขระ
        int     21h
        cmp     al,'Q'          ; เปรียบเทียบ al กับ 'Q'
        jne     getonechar      ; ถ้าไม่เท่ากันให้กระโดดไปที่getonechar (กลับไปรับตัวอักษรใหม่)

        mov     ah,02           ; บริการหมายเลข 2 : พิมพ์อักขระ
        mov     dl,32           ; เริ่มที่ ช่องว่าง ASCII = 32 ('   ')
printloop:
        cmp     dl,128          ; เปรียบเทียบ dl กับ 128 (ASCII สุดท้าย)
        ja      finish          ; ถ้ามากกว่ากระโดดไปที่finish
        int     21h             ; พิมพ์อักขระที่มี ASCII = dl
        inc     dl              ; เพิ่ม dl
        jmp     printloop
finish:
```

Loop

- Base on CX (Counter register) to count the loop.
- Instructions :
  - LOOP ; Dec CX … 0
  - LOOPZ ;CX<>0 and Z=1
  - LOOPNZ ;CX<>0 and Z=0
  - JCXZ ; Jump if CX=0, used with LOOP to determine the CX before loop.

LOOP

```
        mov     cx,20                   ; ทำซ้ำ 20 ครั้ง
        mov     bl,1                    ; เริ่มจาก 1
        mov     dx,0                    ; กำหนดค่าเริ่มต้นให้กับผลรวม
addonenumber:
        add     dl,bl                   ; บวก 8 บิตล่าง
        adc     dh,0                    ; รวมตัวทด
        inc     bl                      ; ตัวถัดไป
        loop    addonenumber            ; ถ้า CX ลดลงแล้วไม่เท่ากับ 0  ทำซ้ำต่อไป
```

JCXZ

```
          initialization
          jcxz   endloop                    ; CX =0 ?
label1:
          actions
          loop   label1                     ; loop
endloop:
```

Example finding first character

```
        cmp     byte ptr [bx],' '       ; พิจารณาอักษรตัวแรก
        jnz     found                   ; อักษรตัวแรกไม่ใช่ช่องว่าง
        mov     cx,100                  ; ทำซ้ำ 100 ครั้ง
findnotspace:
        inc     bx                      ; BX ชี้ไปยังอักษรตัวถัดไป
        cmp     byte ptr [bx],' '       ; เปรียบเทียบ
        loopz   findnotspace            ; ทำซ้ำถ้ายังพบช่องว่างและยังไม่ครบข้อมูล
        jnz     found                   ; ค้นเจออักษรตัวแรก

        ; not found                     ; ข้อความมีแต่ช่องว่าง
        …

found:
        ; found                         ; พบอักษรตัวแรก
        …
```

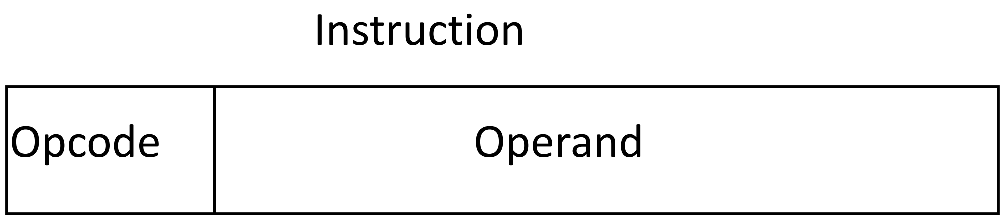
Addressing Modes

- 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

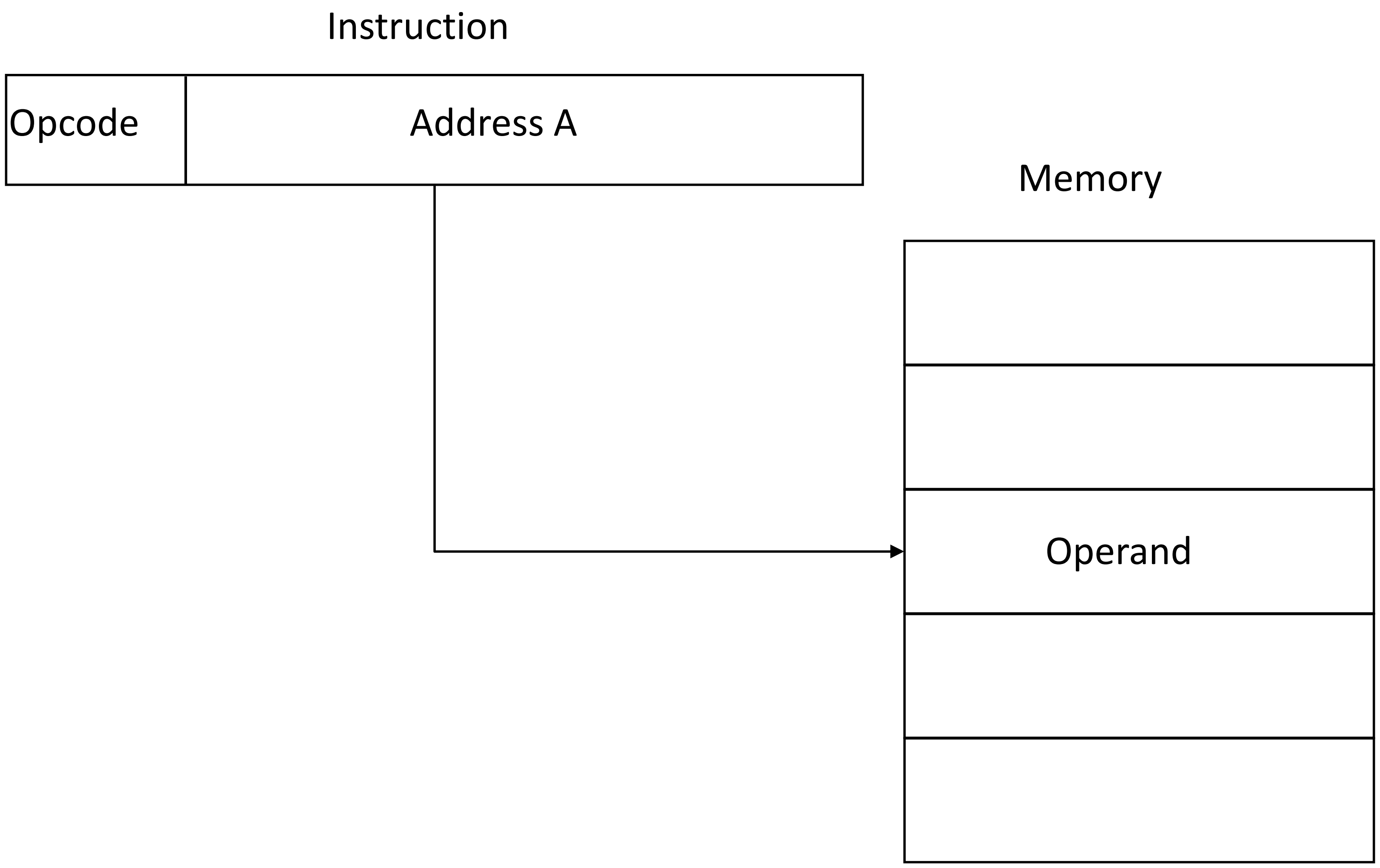
Immediate Addressing Diagram



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



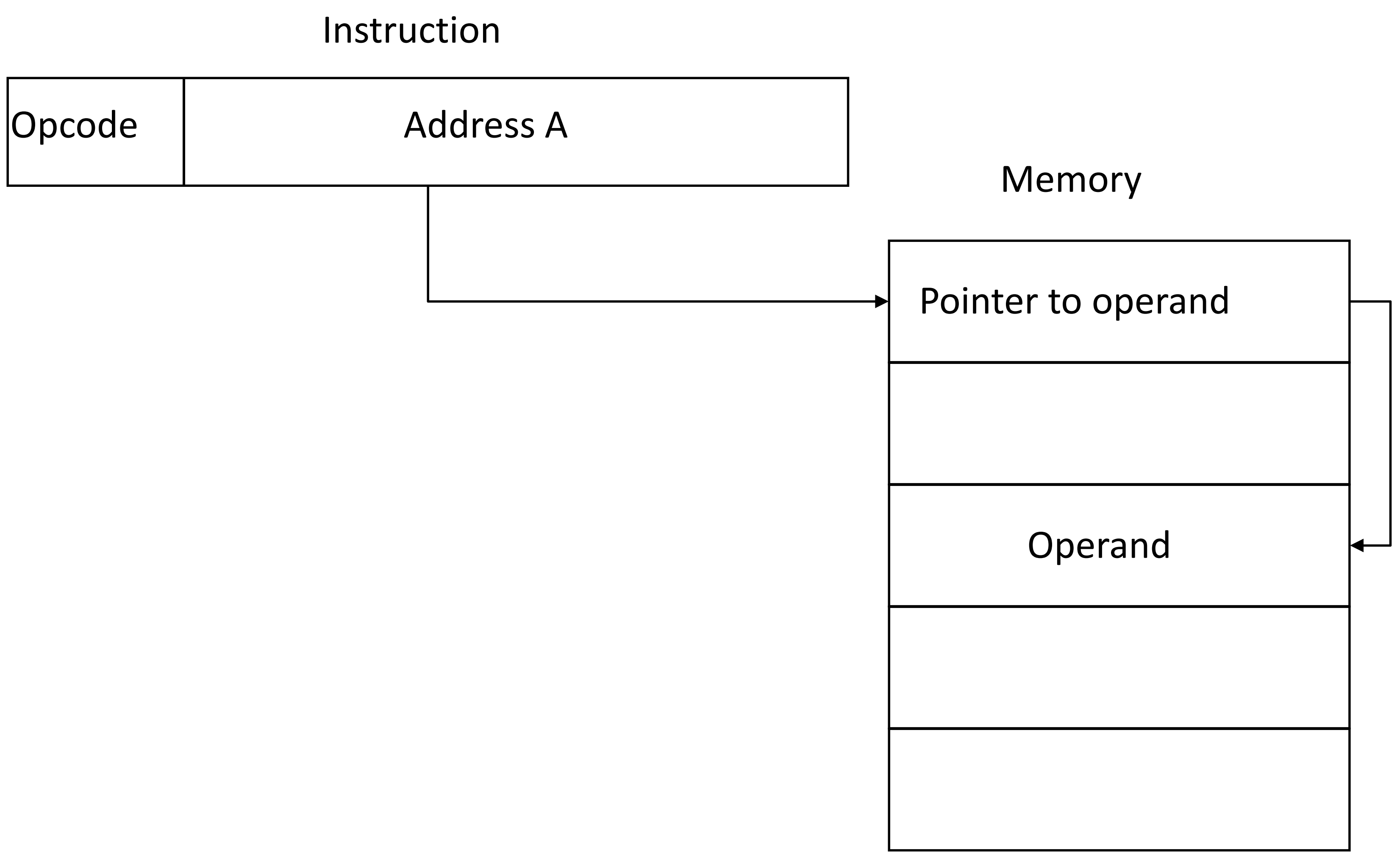
Indirect Addressing (1)

- 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



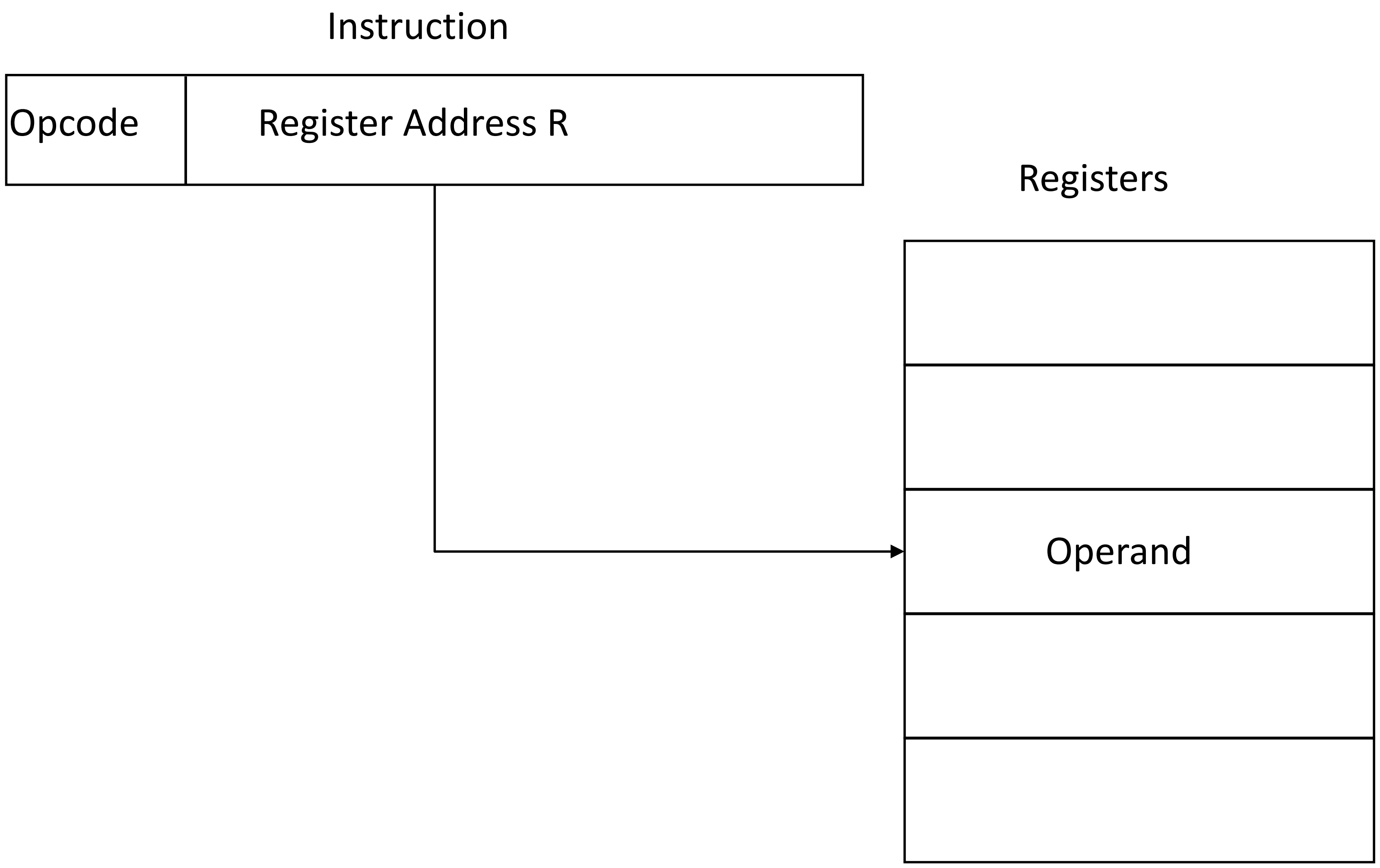
Register Addressing (1)

- 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

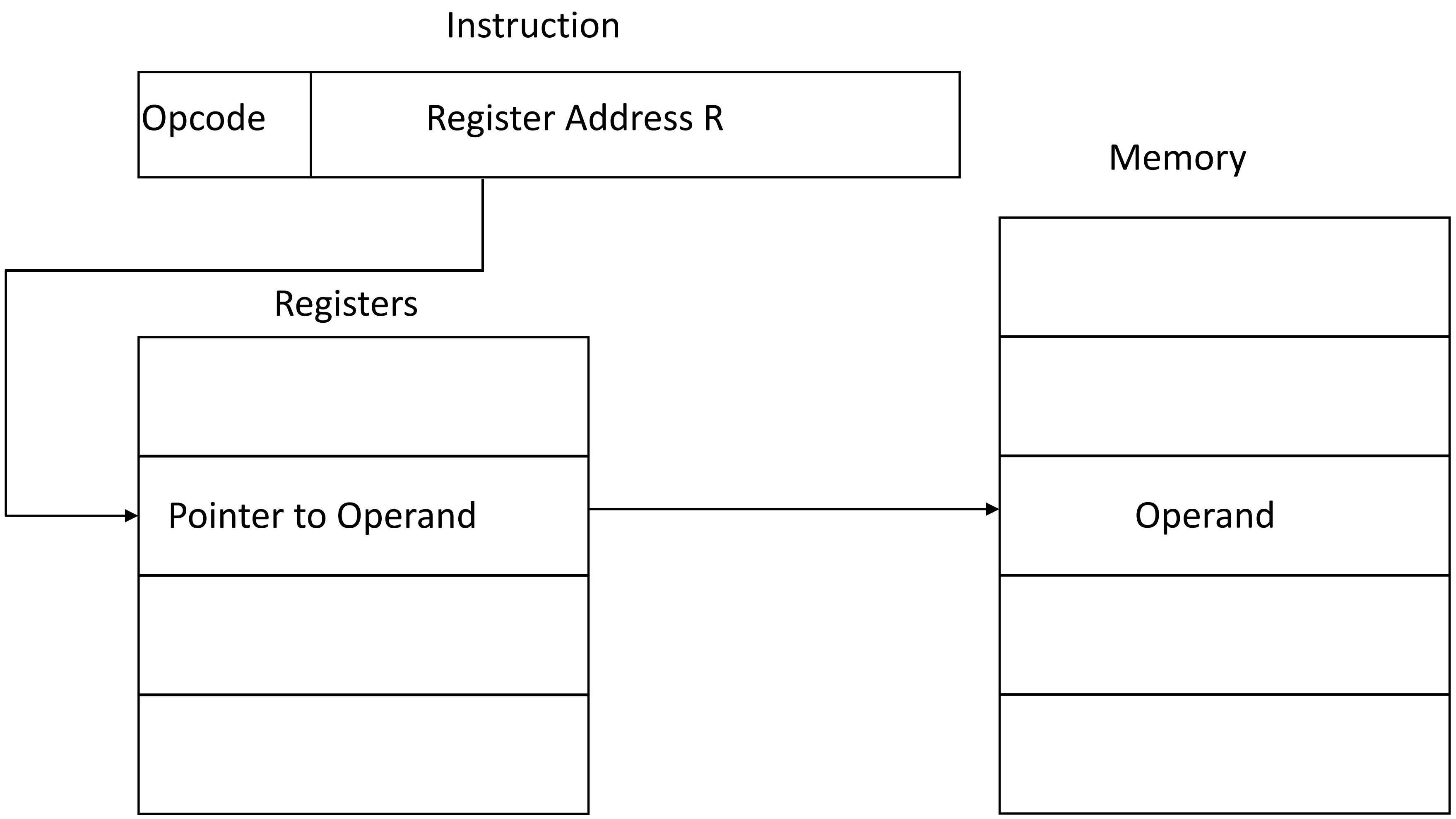
Register Addressing Diagram



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

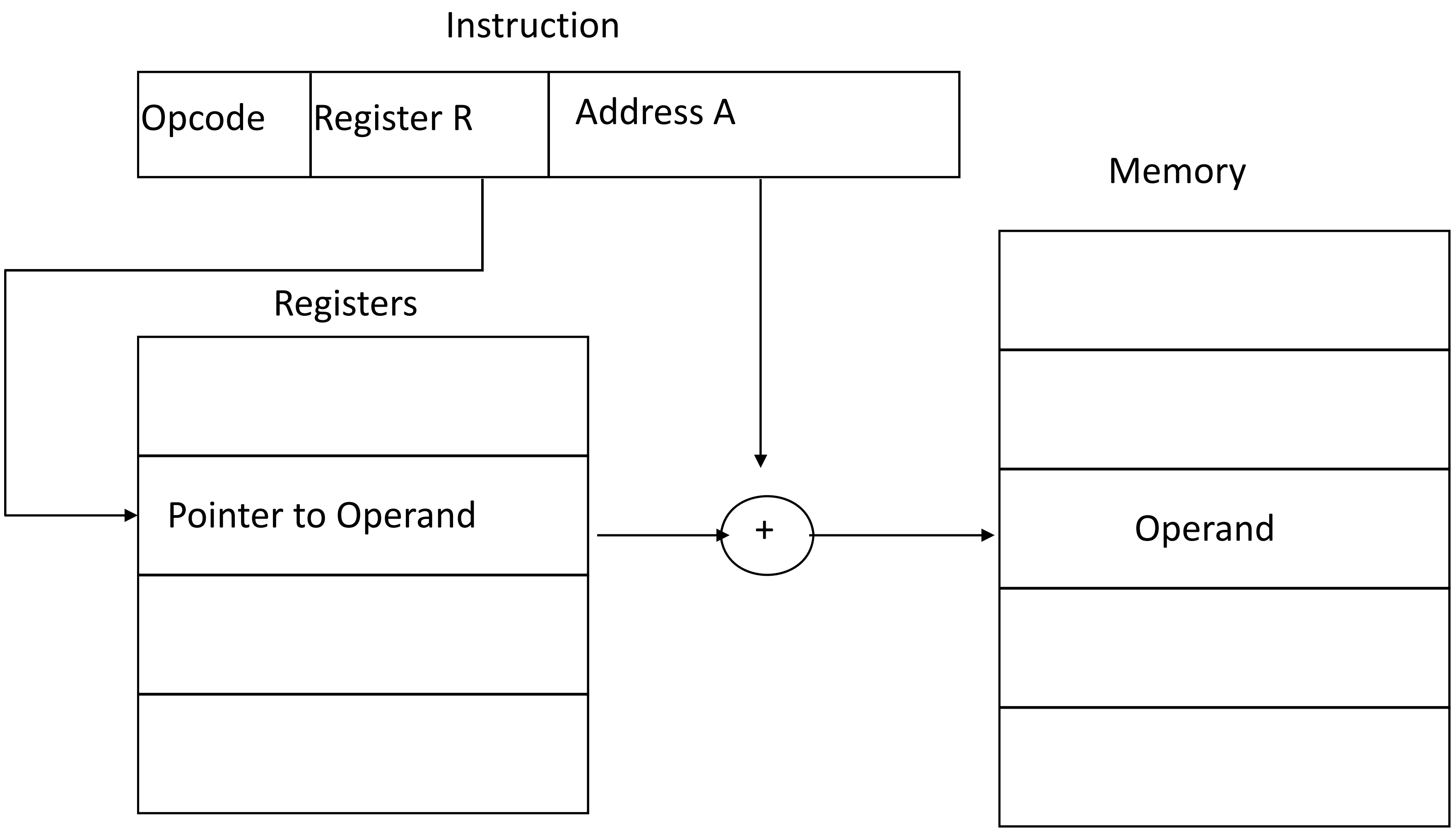
Register Indirect Addressing Diagram



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++

Combinations

- Postindex
- EA = (A) + (R)

- Preindex
- EA = (A+(R))

- (Draw the diagrams)

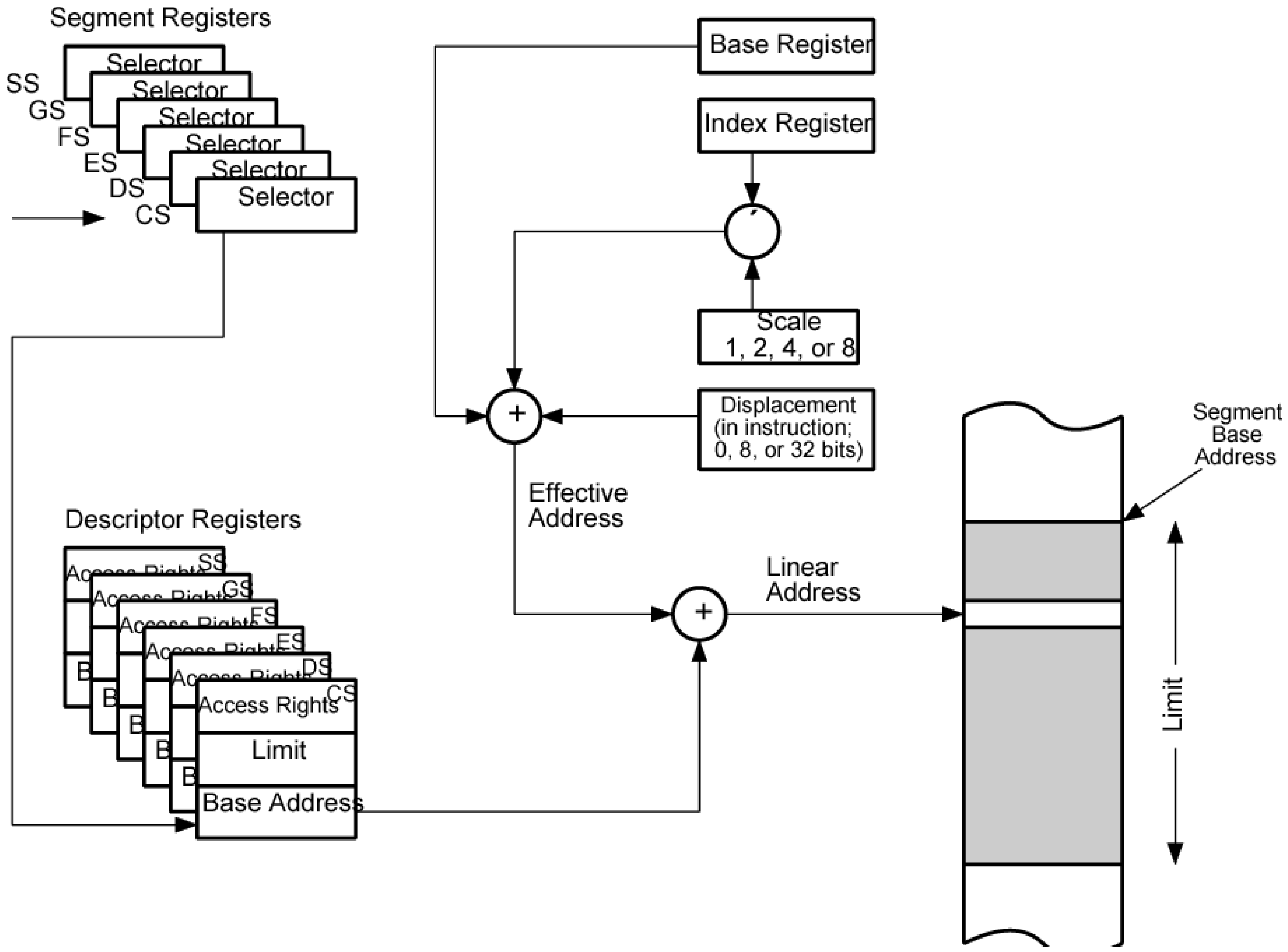
Stack Addressing

- Operand is (implicitly) on top of stack
- 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

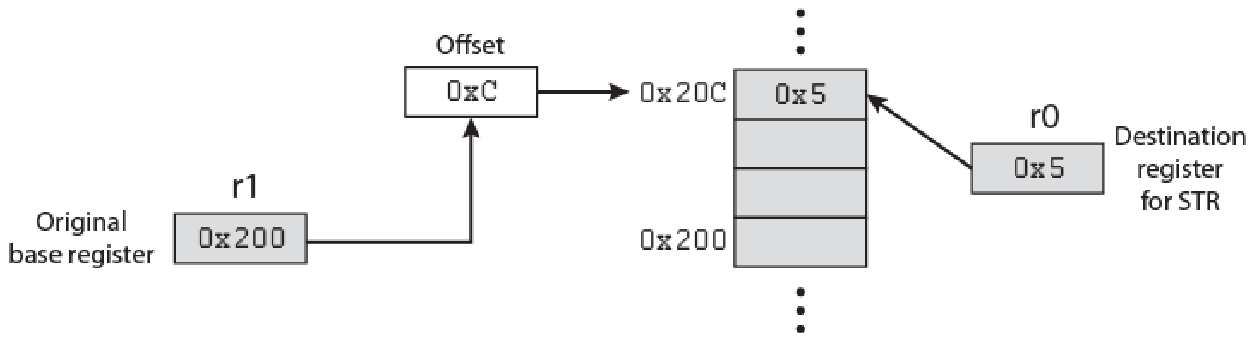
x86 Addressing Mode Calculation



ARM Addressing Modes Load/Store

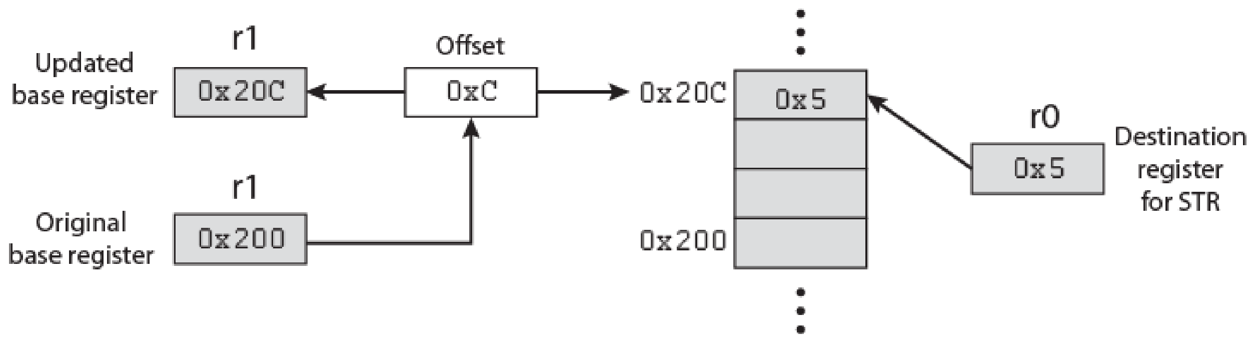
- Only instructions that reference memory
- Indirectly through base register plus offset
- Offset
  - Offset added to or subtracted from base register contents to form the memory address
- Preindex
  - Memory address is formed as for offset addressing
  - Memory address also written back to base register
  - So base register value incremented or decremented by offset value
- Postindex
  - Memory address is base register value
  - Offset added or subtracted
    Result written back to base register
- Base register acts as index register for preindex and postindex addressing
- Offset either immediate value in instruction or another register
- If register scaled register addressing available
  - Offset register value scaled by shift operator
  - Instruction specifies shift size

```
STRB r0, [r1, #12]
```



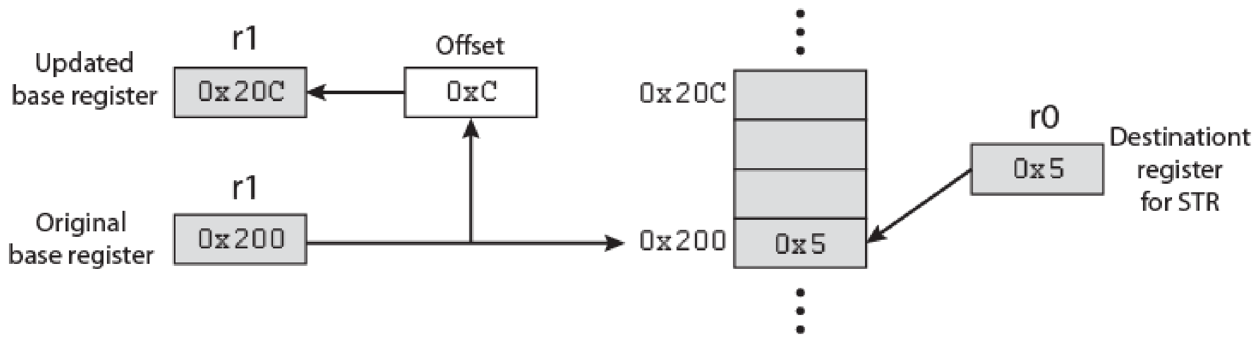
(a) Offset

```
STRB r0, [r1, #12]!
```



(b) Preindex

```
STRBv r0, [r1], #12
```



(c) Postindex

ARM Data Processing Instruction Addressing & Branch Instructions

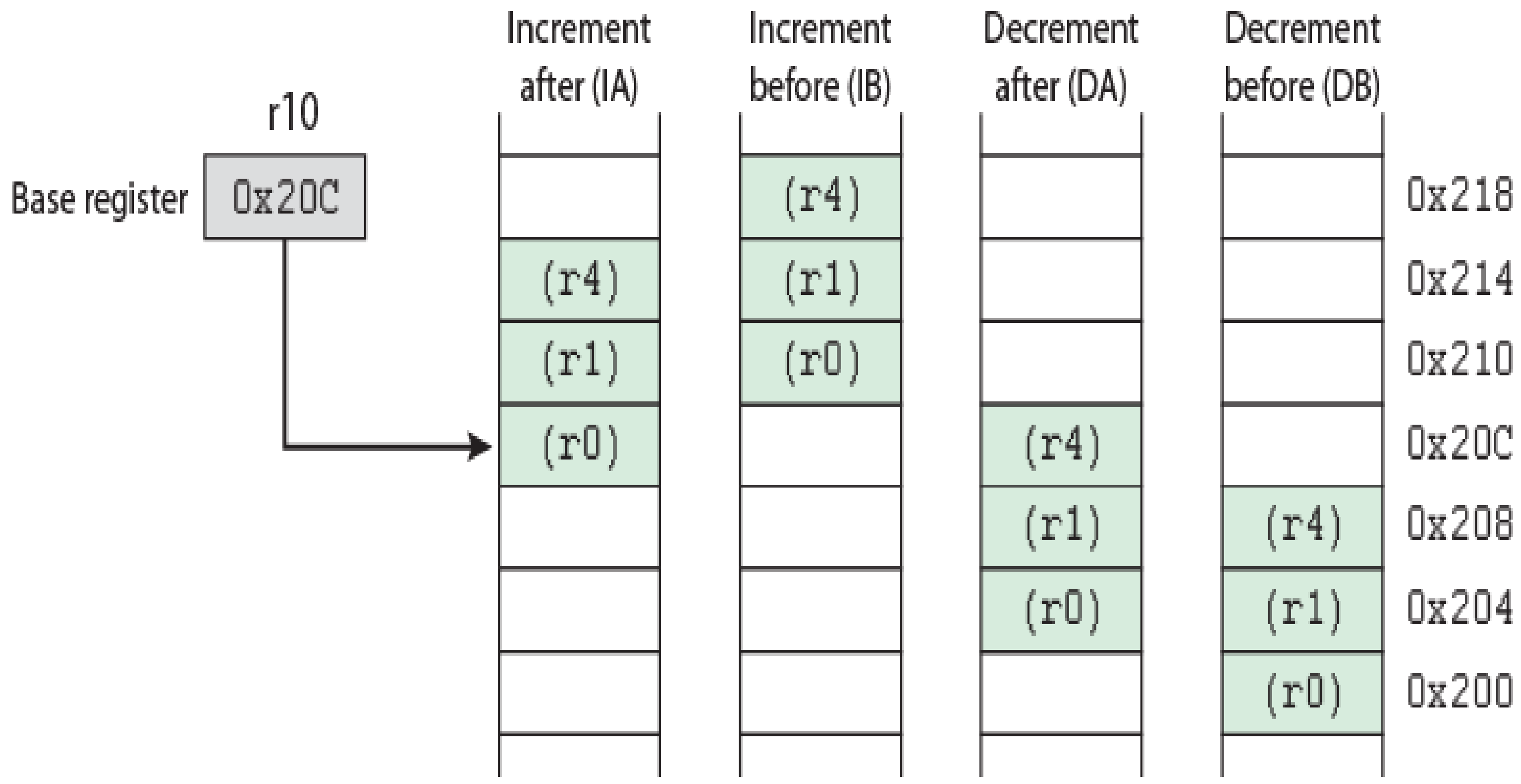
- Data Processing
  - Register addressing
    - Value in register operands may be scaled using a shift operator
  - Or mixture of register and immediate addressing
- Branch
  - Immediate
  - Instruction contains 24 bit value
  - Shifted 2 bits left
    - On word boundary
    - Effective range +/-32MB from PC.

ARM Load/Store Multiple Addressing

- Load/store subset of general-purpose registers
- 16-bit instruction field specifies list of registers
- Sequential range of memory addresses
- Increment after, increment before, decrement after, and decrement before
- Base register specifies main memory address
- Incrementing or decrementing starts before or after first memory access

ARM Load/Store Multiple Addressing Diagram

```
LDMxx r10, {r0, r1, r4}
STMxx r10, {r0, r1, r4}
```



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

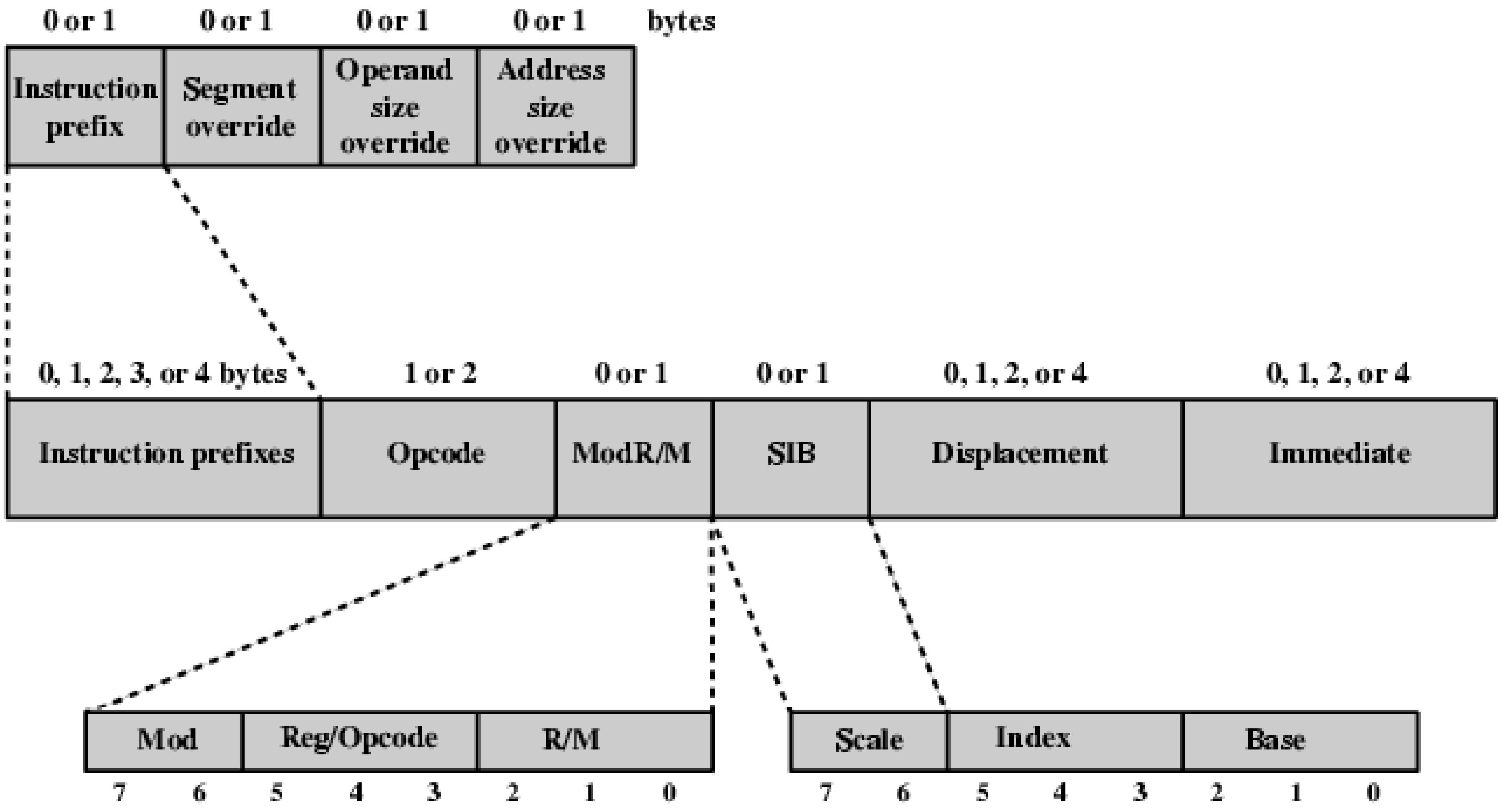
Instruction Length

- 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



Symbolic Addresses

- First field (address) now symbolic
- Memory references in third field now symbolic
- Now have assembly language and need an assembler to translate
- Assembler used for some systems programming
  - Compliers
  - I/O routines

Symbolic Program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

Assembler Program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

Interrupts

- ❑ An interrupt is an event that occurs while the processor is executing a program
- ❑ The interrupt temporarily suspends execution of the program and switch the processor to executing a special routine (interrupt service routine)
- ❑ When the execution of interrupt service routine is complete, the processor resumes the execution of the original program
- ❑ Interrupt classification

Hardware Interrupts	Software Interrupts
— Caused by activating the processor's interrupt control signals (NMI, INTR)	— Caused by the execution of an INT instruction — Caused by an event which is generated by the execution of a program, such as division by zero

- ❑ 8088 can have 256 interrupts

Input/Output Organization

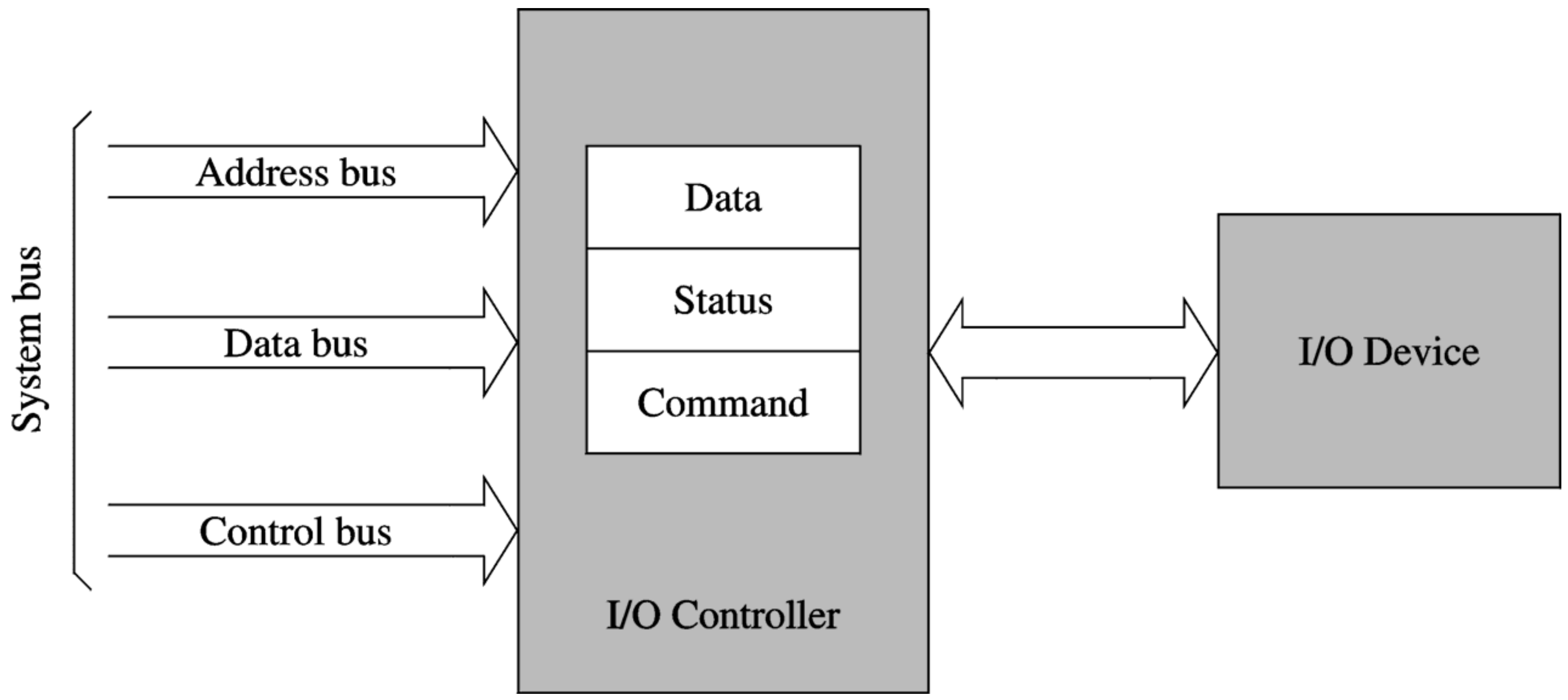
Outline

- Introduction
- Accessing I/O devices
- An example I/O device
  - Keyboard
- I/O data transfer
  - Programmed I/O
  - DMA
- Error detection and correction
  - Parity encoding
  - Error correction
  - CRC
- External interface
  - Serial transmission
  - Parallel interface
- USB
  - Motivation
  - USB architecture
  - USB transactions
- IEEE 1394
  - Advantages
  - Transactions
  - Bus arbitration
  - Configuration

Introduction

- I/O devices serve two main purposes
  - To communicate with outside world
  - To store data
- I/O controller acts as an interface between the systems bus and I/O device
  - Relieves the processor of low-level details
  - Takes care of electrical interface
- I/O controllers have three types of registers
  - Data
  - Command
  - Status

Introduction (cont’d)



Introduction (cont'd)

- To communicate with an I/O device, we need
  - Access to various registers (data, status,…)
    - This access depends on I/O mapping
      - Two basic ways
        - » Memory-mapped I/O
        - » Isolated I/O
  - A protocol to communicate (to send data, …)
    - Three types
      - Programmed I/O
      - Direct memory access (DMA)
      - Interrupt-driven I/O

Accessing I/O Devices

- I/O address mapping
  - Memory-mapped I/O
    - Reading and writing are similar to memory read/write
    - Uses same memory read and write signals
    - Most processors use this I/O mapping
  - Isolated I/O
    - Separate I/O address space
    - Separate I/O read and write signals are needed
    - Pentium supports isolated I/O
      - 64 KB address space
        - » Can be any combination of 8-, 16- and 32-bit I/O ports
      - Also supports memory-mapped I/O

Accessing I/O Devices (cont’d)

- Accessing I/O ports in Pentium
  - Register I/O instructions

    ```
    in    accumulator, port8   ; direct format
    ```

      - Useful to access first 256 ports

    ```
    in    accumulator,DX        ; indirect format
    ```

      - DX gives the port address
  - Block I/O instructions
    - **`ins`** and **`outs`**
      - Both take no operands---as in string instructions
    - **`ins`**: port address in DX, memory address in ES:(E)DI
    - **`outs`**: port address in DX, memory address in ES:(E)SI
    - We can use **`rep`** prefix for block transfer of data

An Example I/O Device

- Keyboard
  - – Keyboard controller scans and reports
      - – Key depressions and releases
    - Supplies key identity as a scan code
      - – Scan code is like a sequence number of the key
        - » Key’s scan code depends on its position on the keyboard
        - » No relation to the ASCII value of the key
  - – Interfaced through an 8-bit parallel I/O port
    - Originally supported by 8255 programmable peripheral interface chip (PPI)

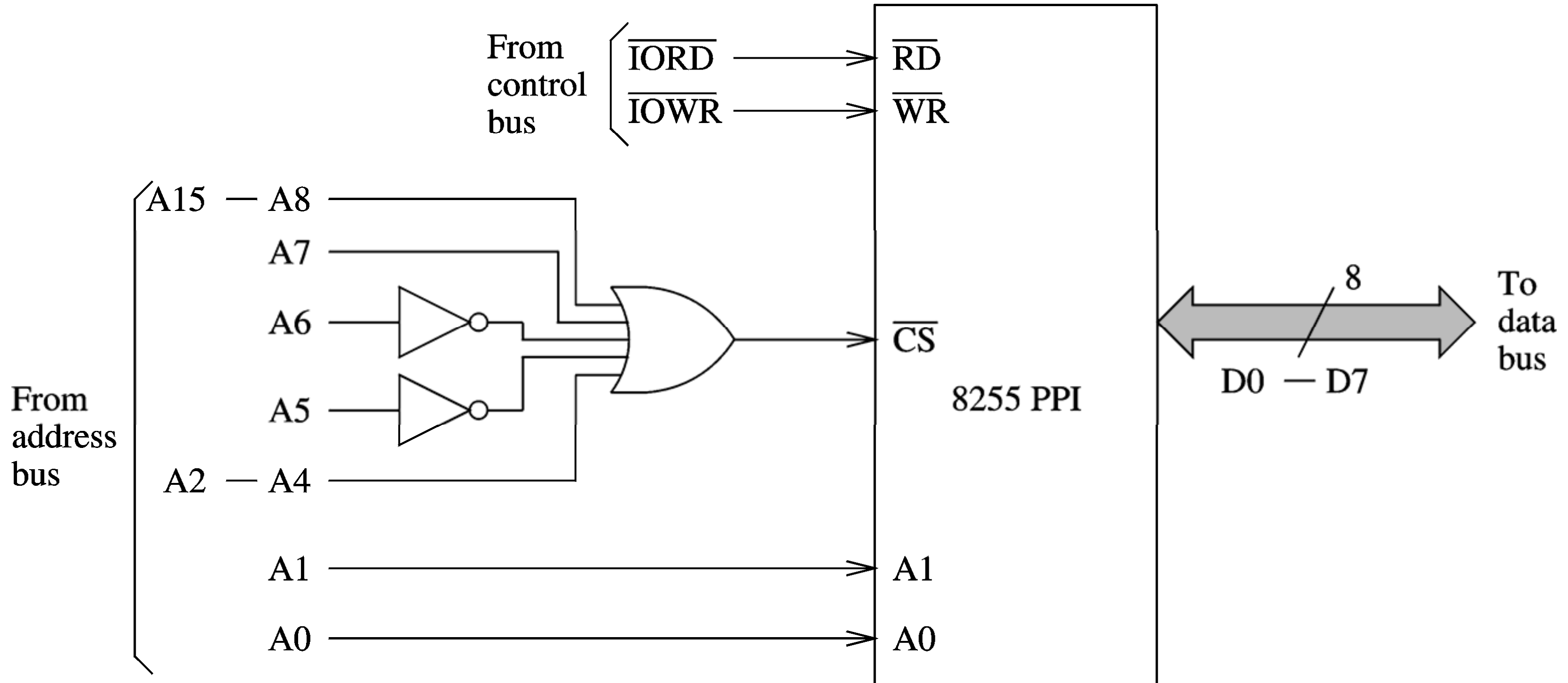
An Example I/O Device (cont'd)

- 8255 PPI has three 8-bit registers
    - Port A (PA)
    - Port B (PB)
    - Port C (PC)
  - These ports are mapped as follows

8255 register	Port address
PA (input port)	60H
PB (output port)	61H
PC (input port)	62H
Command register	63H

An Example I/O Device (cont'd)

Mapping of 8255 I/O ports



An Example I/O Device (cont’d)

- Mapping I/O ports is similar to mapping memory
  - Partial mapping
  - Full mapping
    - See our discussion in Chapter 16
- Keyboard scan code and status can be read from port 60H
  - 7-bit scan code is available from
    - PA0 – PA6
  - Key status is available from PA7
    - PA7 = 0 – key depressed
    - PA0 = 1 – key released

I/O Data Transfer

- Data transfer involves two phases
  - A data transfer phase
    - It can be done either by
      - Programmed I/O
      - DMA
  - An end-notification phase
    - Programmed I/O
    - Interrupt
- Three basic techniques
  - Programmed I/O
  - DMA
  - Interrupt-driven I/O (discussed in Chapter 20)

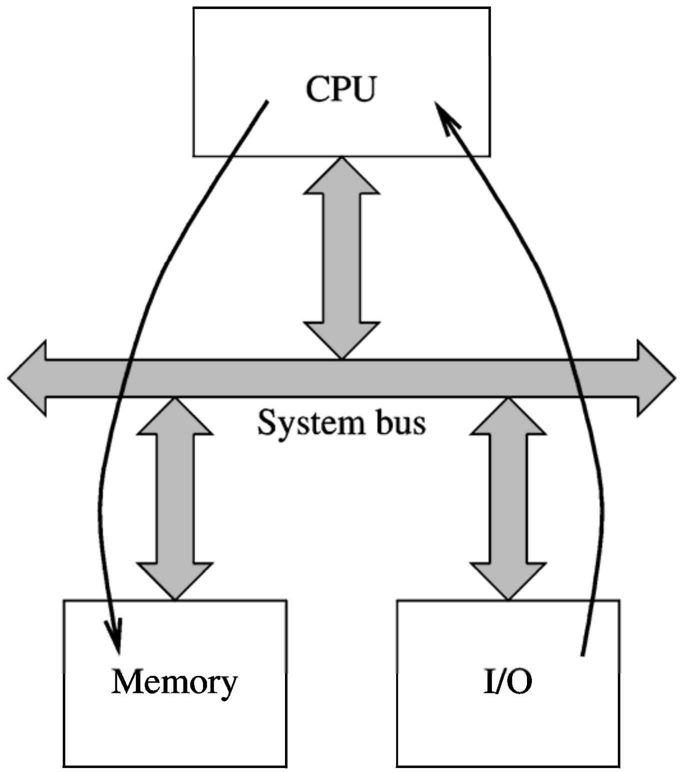
I/O Data Transfer (cont'd)

- Programmed I/O
  - Done by busy-waiting
    - This process is called ***polling***
- Example
  - Reading a key from the keyboard involves
    - Waiting for PA7 bit to go low
      - Indicates that a key is pressed
    - Reading the key scan code
    - Translating it to the ASCII value
    - Waiting until the key is released
  - Program 19.1 uses this process to read input from the keyboard

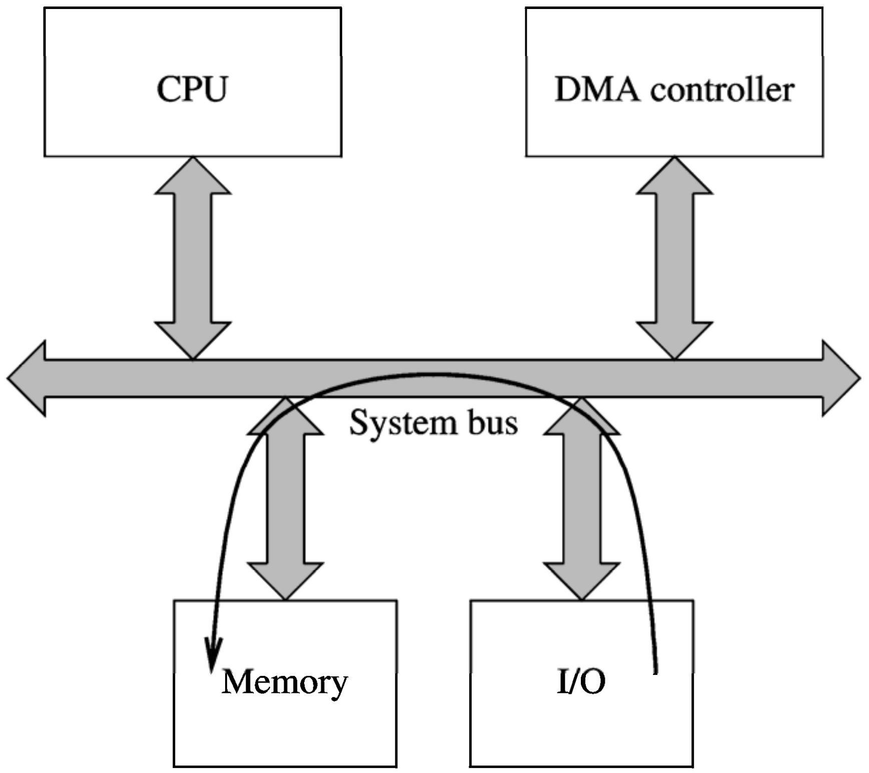
I/O Data Transfer (cont'd)

- Direct memory access (DMA)
  - Problems with programmed I/O
    - Processor wastes time polling
      - In our example
        - » Waiting for a key to be pressed,
        - » Waiting for it to be released
    - May not satisfy timing constraints associated with some devices
      - Disk read or write
  - DMA
    - Frees the processor of the data transfer responsibility

I/O Data Transfer (cont'd)



(a) Programmed I/O transfer



(b) DMA transfer

I/O Data Transfer (cont'd)

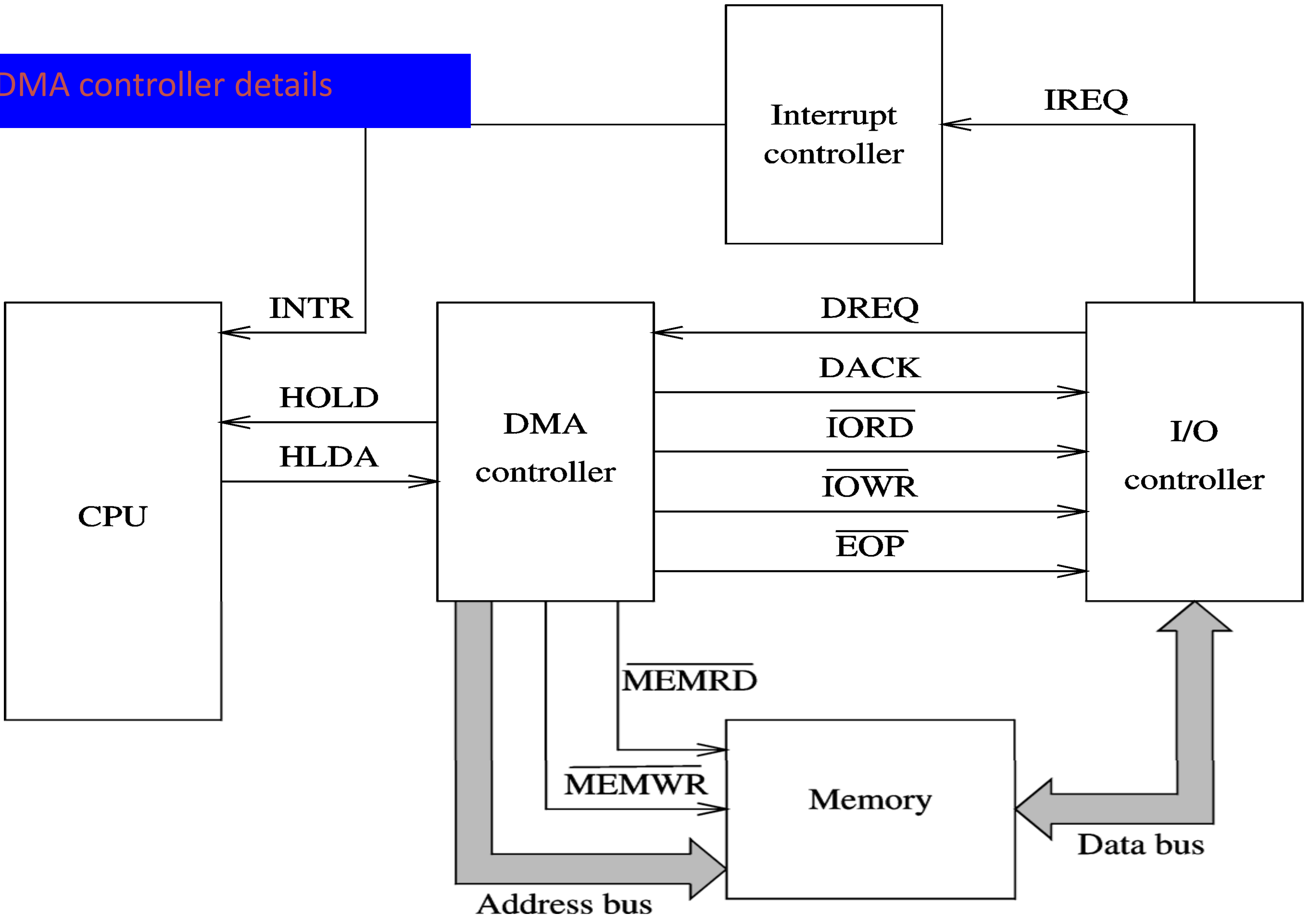
- DMA is implemented using a DMA controller
  - DMA controller
    - Acts as slave to processor
    - Receives instructions from processor
    - Example: Reading from an I/O device
      - Processor gives details to the DMA controller
        - » I/O device number
        - » Main memory buffer address
        - » Number of bytes to transfer
        - » Direction of transfer (memory $\rightarrow$ I/O device, or vice versa)

I/O Data Transfer (cont'd)

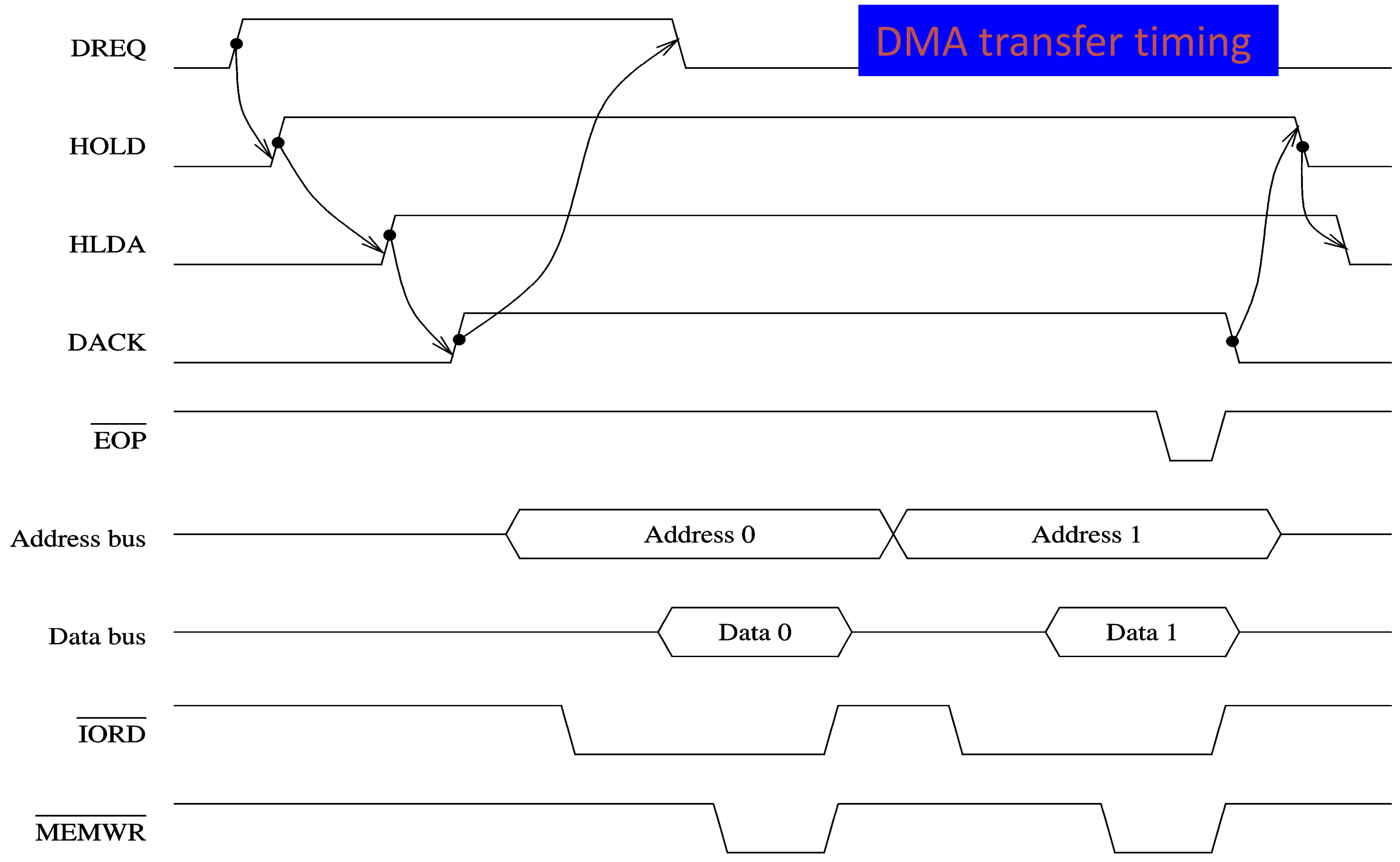
- Steps in a DMA operation
  - Processor initiates the DMA controller
    - Gives device number, memory buffer pointer, …
      - Called ***channel initialization***
    - Once initialized, it is ready for data transfer
  - When ready, I/O device informs the DMA controller
    - DMA controller starts the data transfer process
      - Obtains bus by going through bus arbitration
      - Places memory address and appropriate control signals
      - Completes transfer and releases the bus
      - Updates memory address and count value
      - If more to read, loops back to repeat the process
  - Notify the processor when done
    - Typically uses an interrupt

I/O Data Transfer (cont'd)

DMA controller details

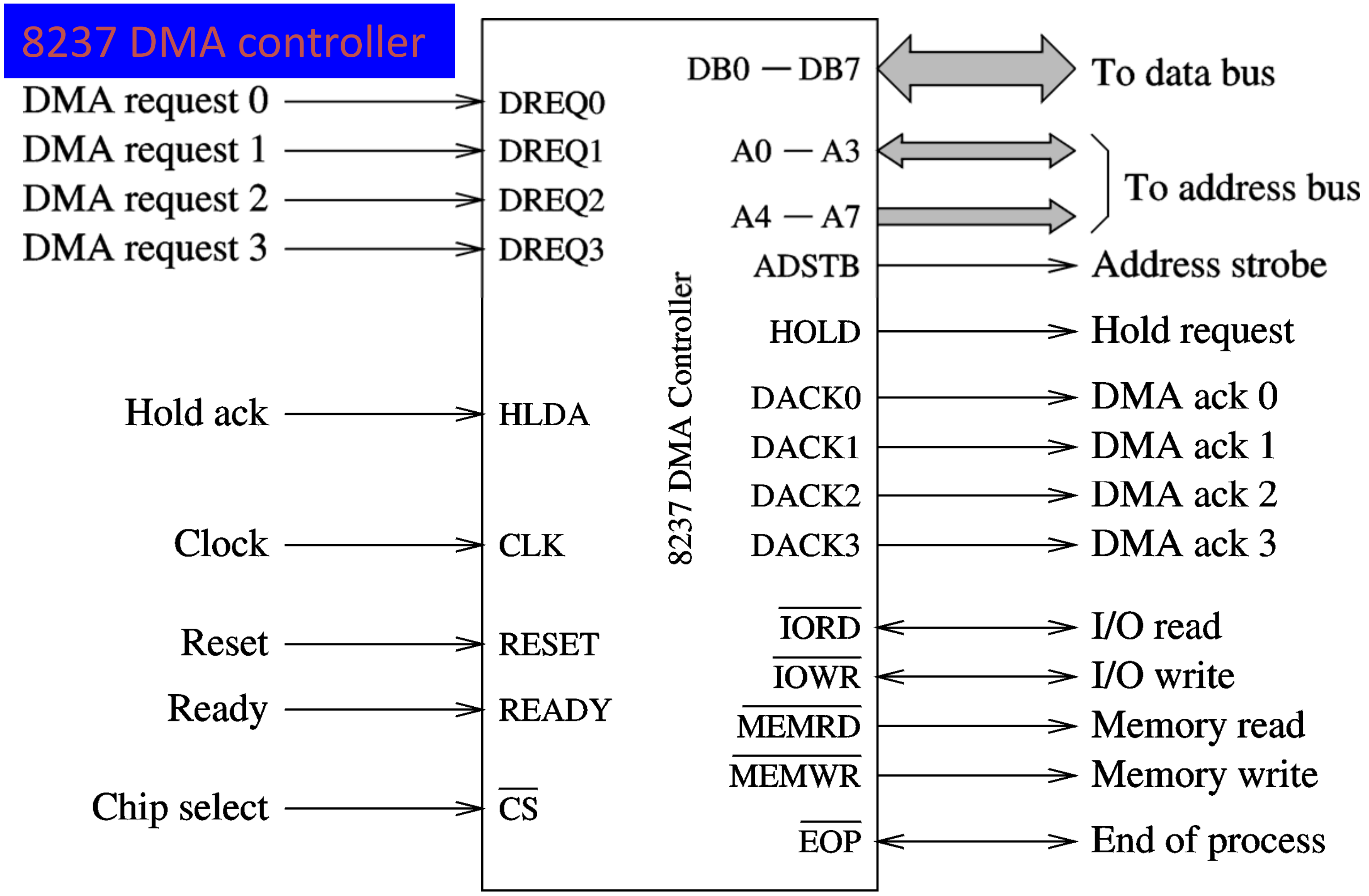


I/O Data Transfer (cont'd)



I/O Data Transfer (cont'd)

8237 DMA controller



I/O Data Transfer (cont’d)

- 8237 supports four DMA channels
- It has the following internal registers
  - Current address register
    - One 16-bit register for each channel
    - Holds address for the current DMA transfer
  - Current word register
    - Keeps the byte count
    - Generates terminal count (TC) signal when the count goes from zero to FFFFH
  - Command register
    - Used to program 8257 (type of priority, …)

I/O Data Transfer (cont’d)

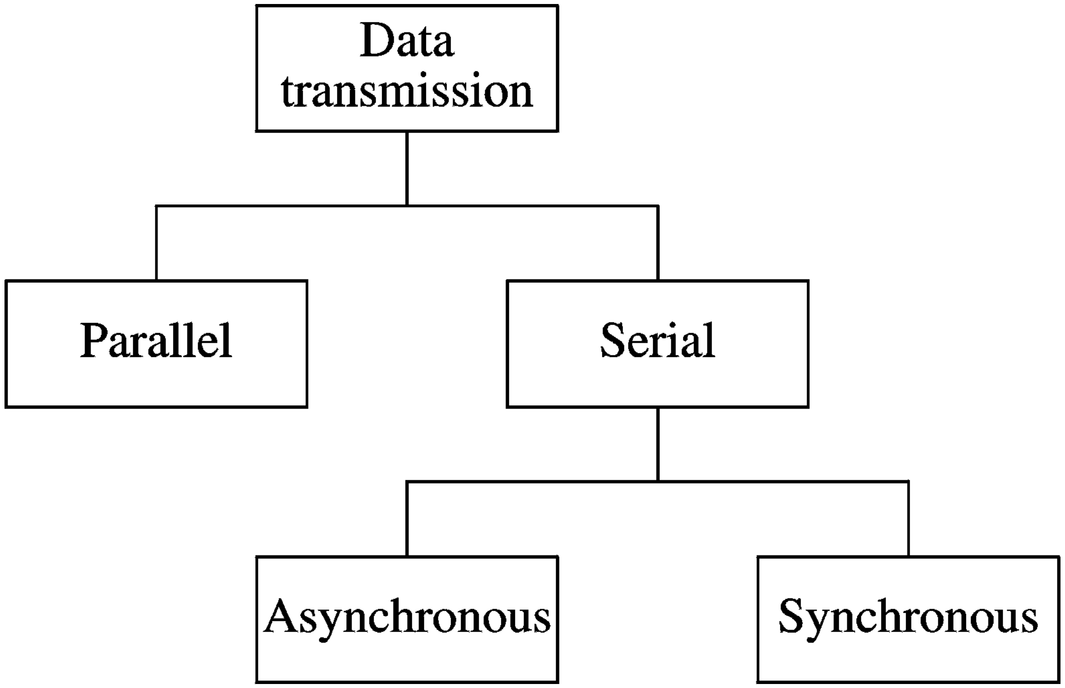
- Mode register
  - Each channel can be programmed to
    - Read or write
    - Autoincrement or autodecrement the address
    - Autoinitialize the channel
- Request register
  - For software-initiated DMA
- Mask register
  - Used to disable a specific channel
- Status register
- Temporary register
  - Used for memory-to-memory transfers

I/O Data Transfer (cont’d)

- 8237 supports four types of data transfer
  - Single cycle transfer
    - Only single transfer takes place
    - Useful for slow devices
  - Block transfer mode
    - Transfers data until TC is generated or external EOP signal is received
  - Demand transfer mode
    - Similar to the block transfer mode
    - In addition to TC and EOP, transfer can be terminated by deactivating DREQ signal
  - Cascade mode
    - Useful to expand the number channels beyond four

External Interface

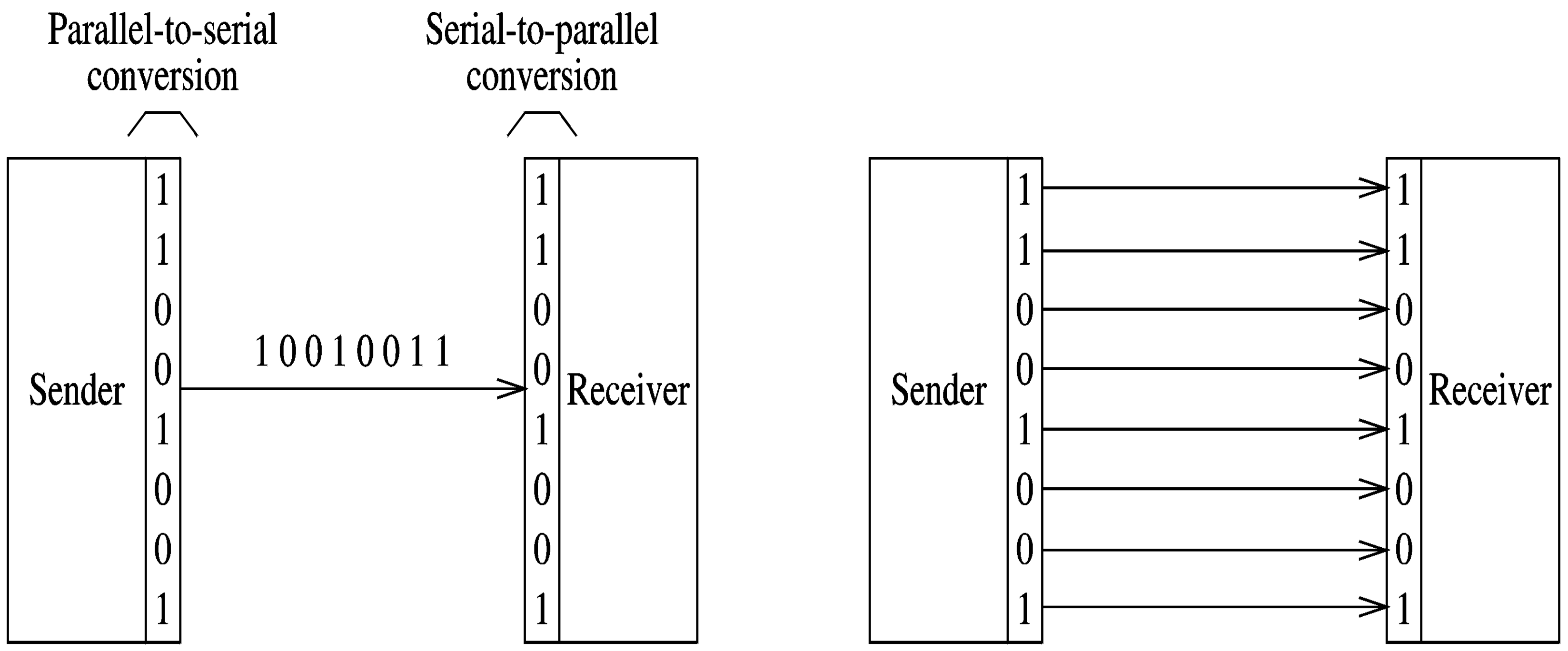
- Two ways of interfacing I/O devices
  - Serial
    - Cheaper
    - Slower
  - Parallel
    - Faster
    - Data skew



Limited to small distances

External Interface (cont'd)

Two basic modes of data transmission



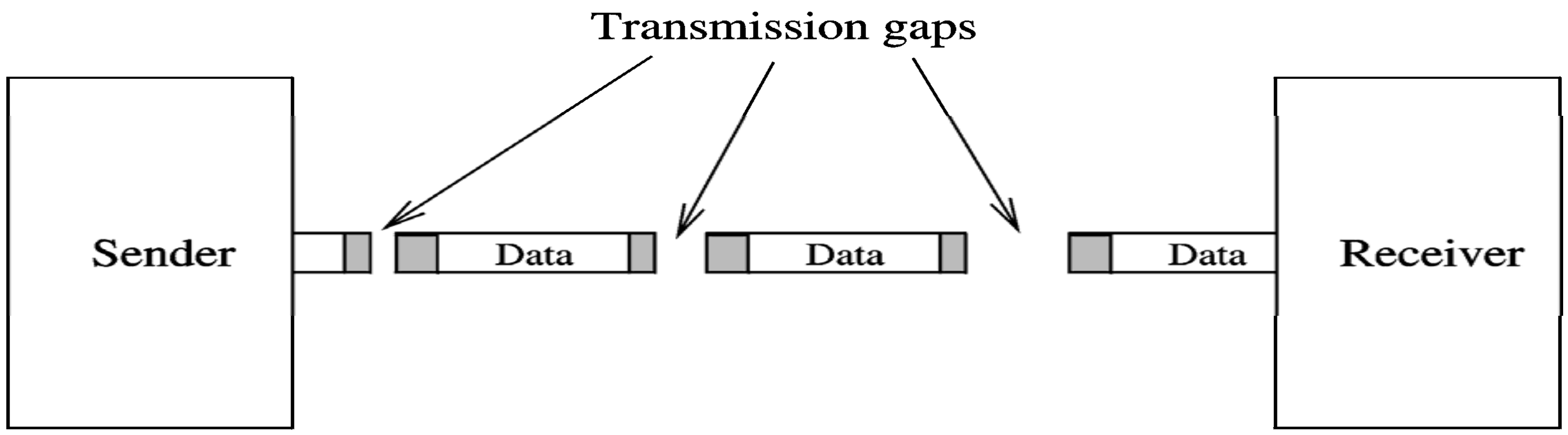
(a) Serial transmission

(b) Parallel transmission

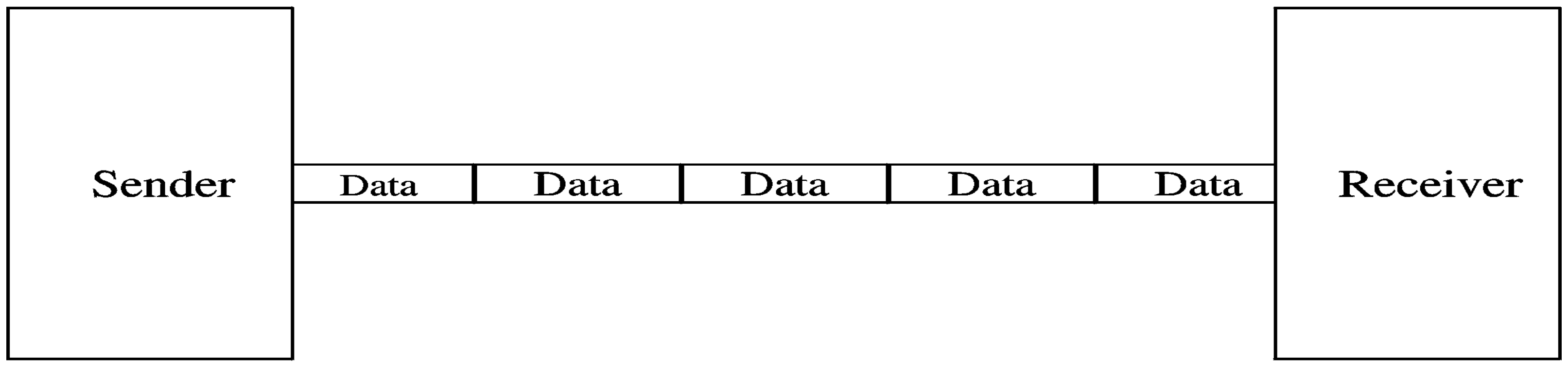
External Interface (cont’d)

- Serial transmission
  - Asynchronous
    - Each byte is encoded for transmission
      - Start and stop bits
    - No need for sender and receiver synchronization
  - Synchronous
    - Sender and receiver must synchronize
      - Done in hardware using phase locked loops (PLLs)
    - Block of data can be sent
    - More efficient
      - Less overhead than asynchronous transmission
    - Expensive

External Interface (cont'd)



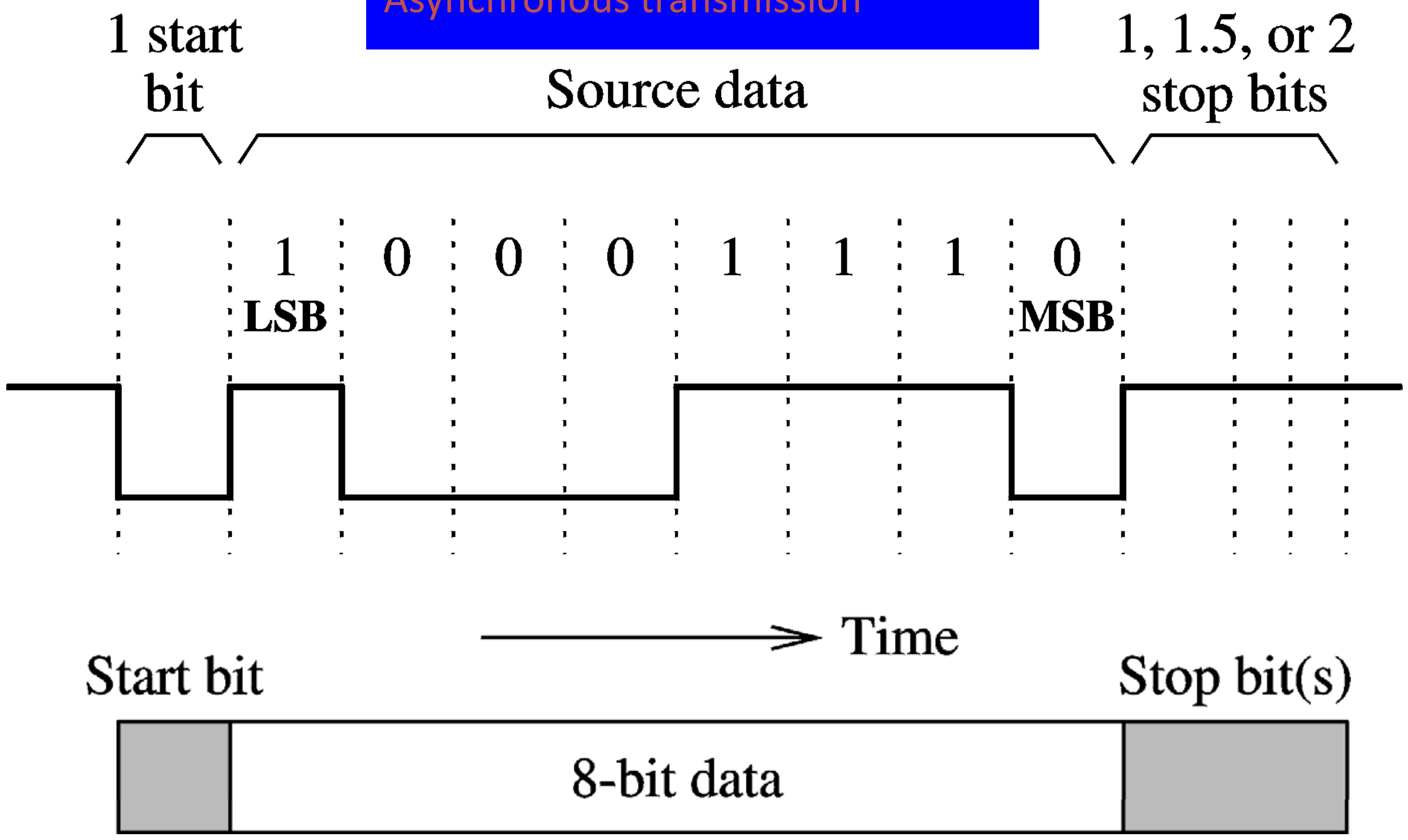
(a) Asynchronous transmission



(b) Synchronous transmission

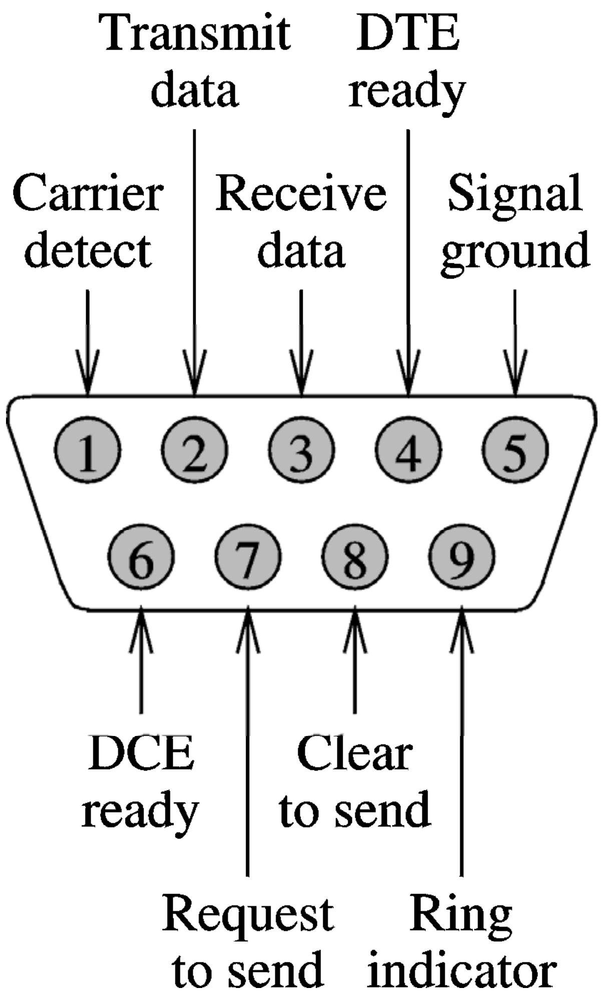
External Interface (cont'd)

Asynchronous transmission



External Interface (cont'd)

- EIA-232 serial interface
  - Low-speed serial transmission
  - Adopted by Electronics Industry Association (EIA)
    - Popularly known by its predecessor RS-232
  - It uses a 9-pin connector DB-9
    - Uses 8 signals
  - Typically used to connect a modem to a computer



External Interface (cont’d)

- Transmission protocol uses three phases
  - Connection setup
    - Computer A asserts DTE Ready
      - Transmits phone# via Transmit Data line (pin 2)
    - Modem B alerts its computer via Ring Indicator (pin 9)
      - Computer B asserts DTE Ready (pin 4)
      - Modem B generates carrier and turns its DCE Ready
    - Modem A detects the carrier signal from modem B
      - Modem A alters its computer via Carrier Detect (pin 1)
      - Turns its DCE Ready
  - Data transmission
    - Done by handshaking using
      - request-to-send (RTS) and clear-to-send (CTS) signals
  - Connection termination
    - Done by deactivating RTS

External Interface (cont'd)

- Parallel printer interface
  - A simple parallel interface
  - Uses 25-pin DB-25
    - 8 data signals
      - Latched by strobe (pin 1)
    - Data transfer uses simple handshaking
      - Uses acknowledge (CK) signal
        - After each byte, computer waits for ACK
    - 5 lines for printer status
      - Busy, out-of-paper, online/offline, autofeed, and fault
    - Can be initialized with INIT
      - Clears the printer buffer and resets the printer

DMA Controller 8237

- DMA Operation
- DMA controller architecture
- DMA transfer types and modes
- DMA 8237 controller pin diagram
- DMA 8237 controller block diagram

Direct memory access

- Direct Memory Access (DMA) allows devices to transfer data without subjecting the processor a heavy overhead. Otherwise, the processor would have to copy each piece of data from the source to the destination. This is typically slower than copying normal blocks of memory since access to I/O devices over a peripheral bus is generally slower than normal system RAM. During this time the processor would be unavailable for any other tasks involving processor bus access. But it can continue to work on any work which does not require bus access. DMA transfers are essential for high performance embedded systems where large chunks of data need to be transferred from the input/output devices to or from the primary memory.

DMA Controller

- A DMA controller is a device, usually peripheral to a CPU that is programmed to perform a sequence of data transfers on behalf of the CPU. A DMA controller can directly access memory and is used to transfer data from one memory location to another, or from an I/O device to memory and vice versa. A DMA controller manages several DMA channels, each of which can be programmed to perform a sequence of these DMA transfers. Devices, usually I/O peripherals, that acquire data that must be read (or devices that must output data and be written to) signal the DMA controller to perform a DMA transfer by asserting a hardware DMA request (DRQ) signal.

A DMA request signal for each channel is routed to the DMA controller. This signal is monitored and responded to in much the same way that a processor handles interrupts. When the DMA controller sees a DMA request, it responds by performing one or many data transfers from that I/O device into system memory or vice versa. Channels must be enabled by the processor for the DMA controller to respond to DMA requests. The number of transfers performed, transfer modes used, and memory locations accessed depends on how the DMA channel is programmed. A DMA controller typically shares the system memory and I/O bus with the CPU and has both bus master and slave capability.

- Figure shows the DMA controller architecture and how the DMA controller interacts with the CPU. In bus master mode, the DMA controller acquires the system bus (address, data, and control lines) from the CPU to perform the
- DMA transfers. Because the CPU releases the system bus for the duration of the transfer, the process is sometimes referred to as cycle stealing.
- In bus slave mode, the DMA controller is accessed by the CPU, which programs the DMA controller's internal registers to set up DMA transfers. The internal registers consist of source and destination address registers and transfer count registers for each DMA channel, as well as control and status registers for initiating, monitoring, and sustaining the operation of the DMA controller.

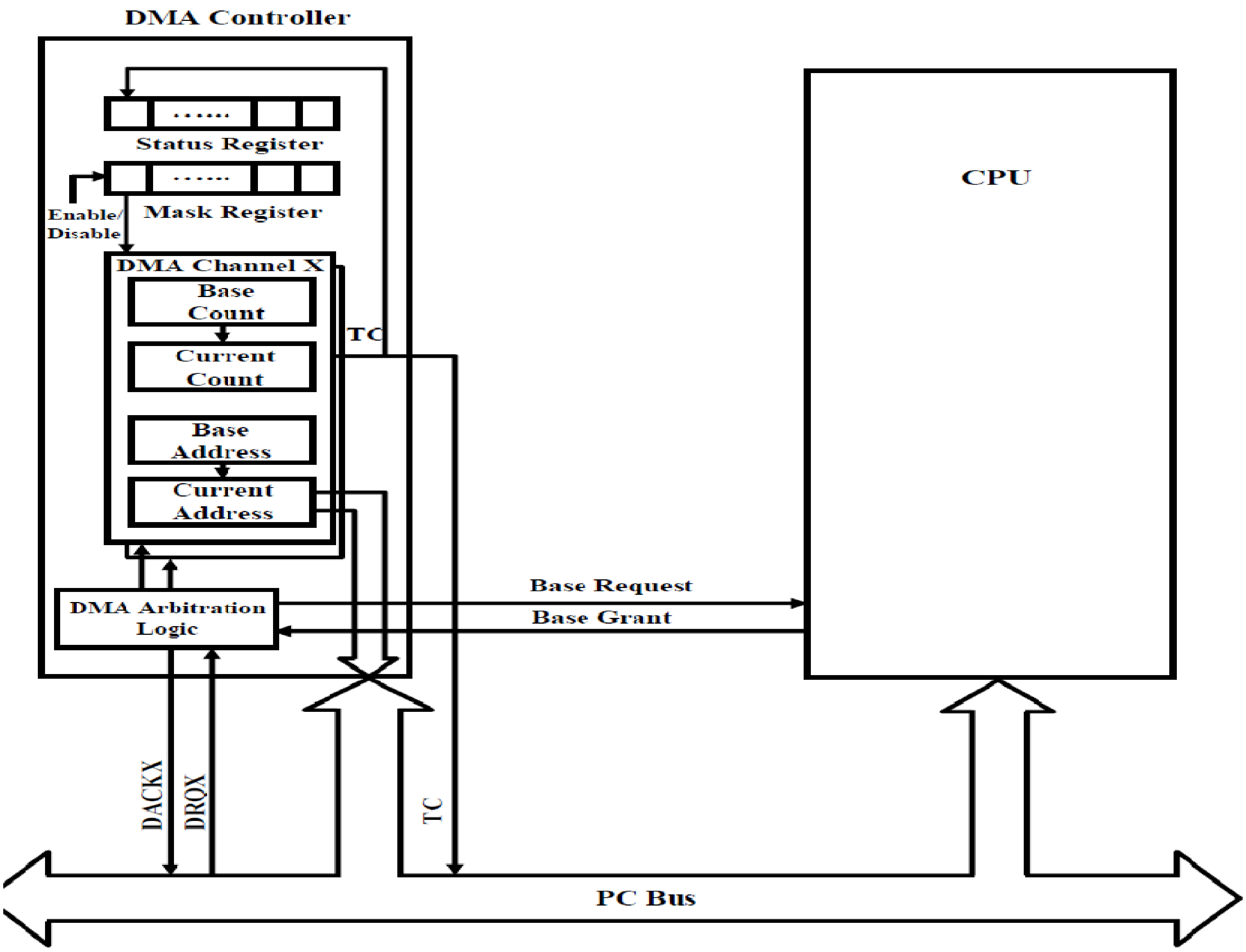
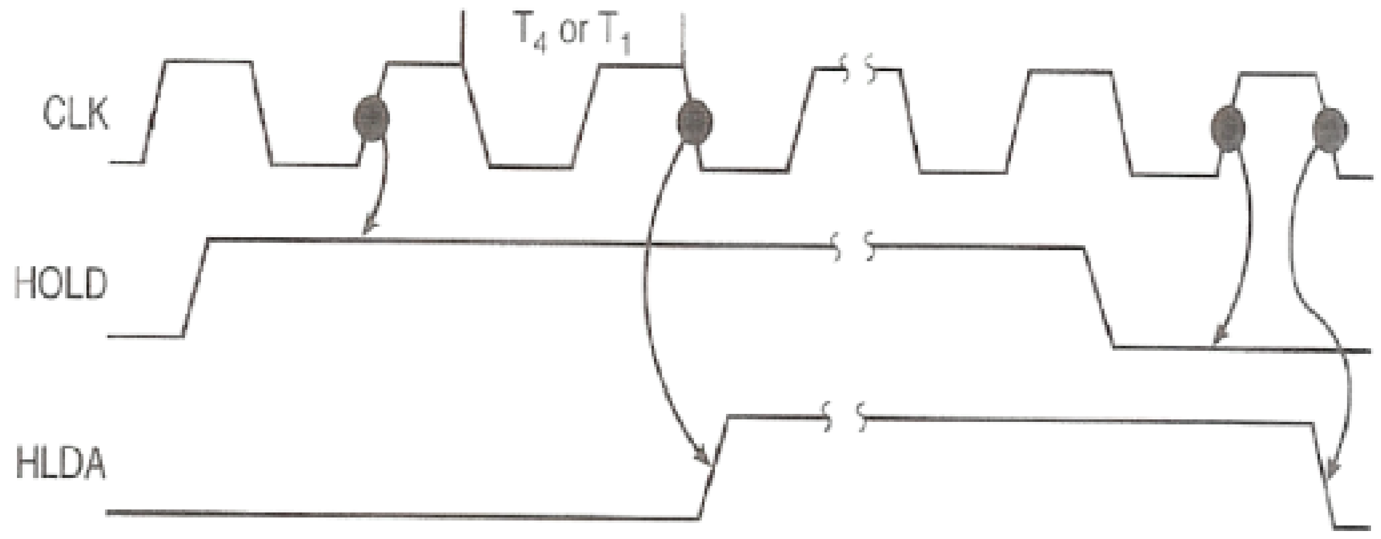


Fig. 16.1 The DMA controller architecture

- During a DMA access the microprocessor is turned off by placing a logic one on the HOLD input. After placing a logic one on HOLD, the microprocessor issues a logic one on the HLDA to indicate a hold is in effect.
- During a HOLD, the microprocessor stops running the program and it places its address, data, and control bus connections at their impedance state.
- This in effect is the same as removing the microprocessor from its socket!
- While the microprocessor is held, other devices are free to gain access to its memory and I/O space to directly transfer data.

- HOLD has a higher priority than interrupts and HOLD takes effect in a clock or two.



DMA Transfer Types and Modes

- DMA controllers vary as to the type of DMA transfers and the number of DMA channels they support.

- The two types of DMA transfers are
    - i) Flyby DMA transfers
    - ii) Fetch-and-deposit DMA transfers.

Flyby Transfer type

- The fastest DMA transfer type is referred to as a single-cycle, single-address, or flyby transfer. In a flyby DMA transfer, a single bus operation is used to accomplish the transfer, with data read from the source and written to the destination simultaneously. In flyby operation, the device requesting service asserts a DMA request on the appropriate channel request line of the DMA controller. The DMA controller responds by gaining control of the system bus from the CPU and then issuing the pre-programmed memory address. Simultaneously, the DMA controller sends a DMA acknowledge signal to the requesting device.

- This signal alerts the requesting device to drive the data onto the system data bus or to latch the data from the system bus, depending on the direction of the transfer. In other words, a flyby DMA transfer looks like a memory read or write cycle with the DMA controller supplying the address and the I/O device reading or writing the data. Because flyby DMA transfers involve a single memory cycle per data transfer, these transfers are very efficient. Fig.16.2 shows the flyby DMA transfer signal protocol.

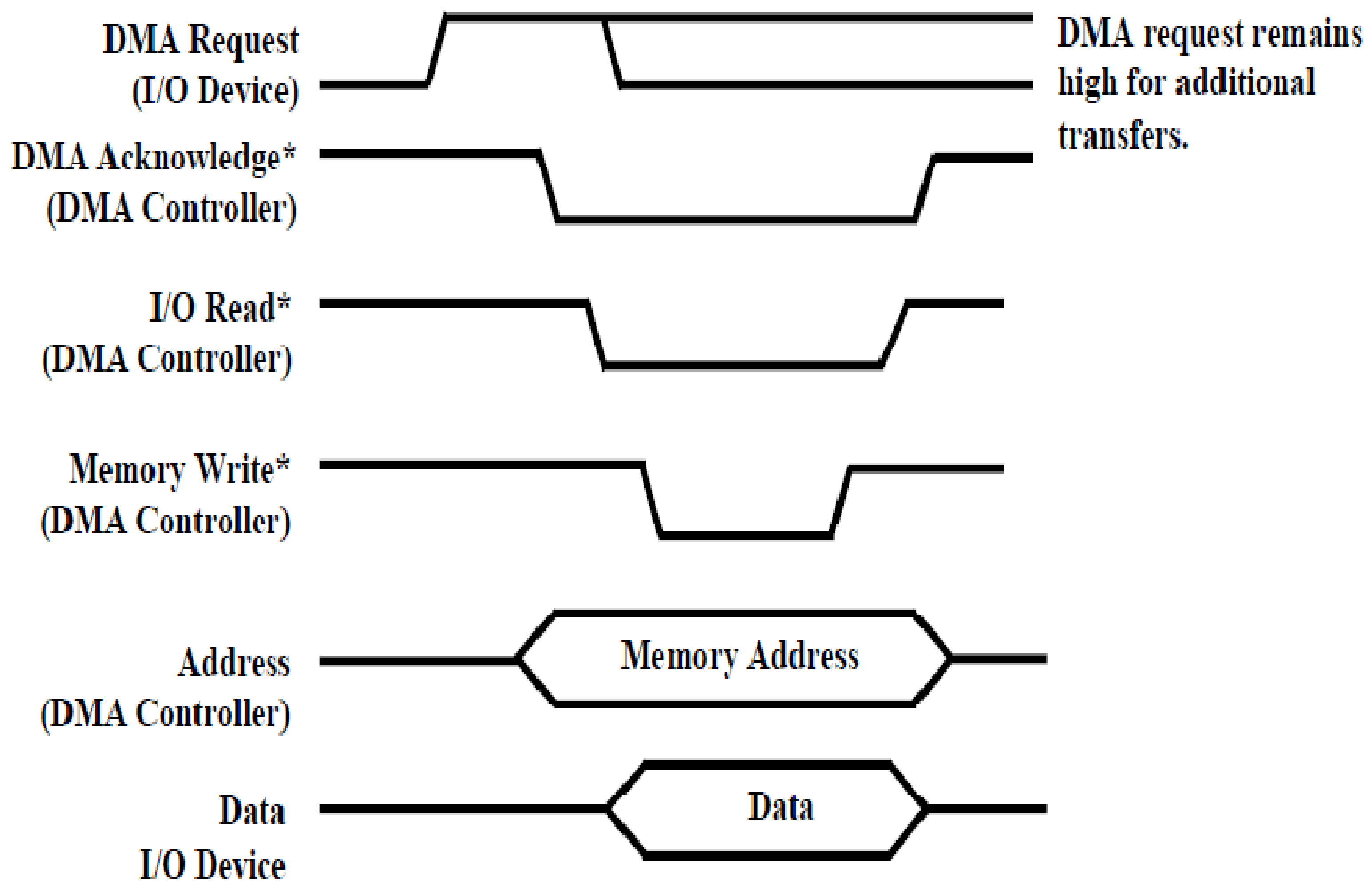


Fig. 16.2 Flyby DMA transfer

Fetch-and-Deposit DMA transfer

- this type of transfer involves two memory or I/O cycles. The data being transferred is first read from the I/O device or memory into a temporary data register internal to the DMA controller. The data is then written to the memory or I/O device in the next cycle. Fig.16.3 shows the fetch-and-deposit DMA transfer signal protocol. Although inefficient because the DMA controller performs two cycles and thus retains the system bus longer, this type of transfer is useful for interfacing devices with different data bus sizes. For example, a DMA controller can perform two 16-bit read operations from one location followed by a 32-bit write operation to another location.

- A DMA controller supporting this type of transfer has two address registers per channel (source address and destination address) and bus-size registers, in addition to the usual transfer count and control registers.
- Unlike the flyby operation, this type of DMA transfer is suitable for both memory-to-memory and I/O transfers.
- Single, block, and demand are the most common transfer modes. Single transfer mode transfers one data value for each DMA request assertion. This mode is the slowest method of transfer because it requires the DMA controller to arbitrate for the system bus with each transfer. This arbitration is not a major problem on a lightly loaded bus, but it can lead to latency problems when multiple devices are using the bus.

- Block and demand transfer modes increase system throughput by allowing the DMA controller to perform multiple DMA transfers when the DMA controller has gained the bus. For block mode transfers, the DMA controller performs the entire DMA sequence as specified by the transfer count register at the fastest possible rate in response to a single DMA request from the I/O device. For demand mode transfers, the DMA controller performs DMA transfers at the fastest possible rate as long as the I/O device asserts its DMA request. When the I/O device unasserts this DMA request, transfers are held off.

DMA modes

- **Demand mode**
- Till EOP or inactive DREQ

- **Single mode**
- Release hold after each byte transfer

- **Block mode**
- Transfer all bytes in count, DREQ not required tobe active Cascade mode
- More than one DMA

DMA Controller Operation

- For each channel, the DMA controller saves the programmed address and count in the base registers and maintains copies of the information in the current address and current count registers, as shown in Fig.16.1. Each DMA channel is enabled and disabled via a DMA mask register. When DMA is started by writing to the base registers and enabling the DMA channel, the current registers are loaded from the base registers. With each DMA transfer, the value in the current address register is driven onto the address bus, and the current address register is automatically incremented or decremented.

- The current count register determines the number of transfers remaining and is automatically decremented after each transfer. When the value in the current count register goes from 0 to -1, a terminal count (TC) signal is generated, which signifies the completion of the DMA transfer sequence. This termination event is referred to as reaching terminal count. DMA controllers often generate a hardware TC pulse during the last cycle of a DMA transfer sequence. This signal can be monitored by the I/O devices participating in the DMA transfers. DMA controllers require reprogramming when a DMA channel reaches TC. Thus, DMA controllers require some CPU time, but far less than is required for the CPU to service device I/O interrupts.

- When a DMA channel reaches TC, the processor may need to reprogram the controller for additional DMA transfers. Some DMA controllers interrupt the processor whenever a channel terminates. DMA controllers also have mechanisms for automatically reprogramming a DMA channel when the DMA transfer sequence completes. These mechanisms include auto initialization and buffer chaining. The auto initialization feature repeats the DMA transfer sequence by reloading the DMA channel's current registers from the base registers at the end of a DMA sequence and re-enabling the channel. Buffer chaining is useful for transferring blocks of data into noncontiguous buffer areas or for handling double-buffered data acquisition. With buffer chaining, a channel interrupts the CPU and is programmed with the next address and count parameters while DMA transfers are being performed on the current buffer

- Some DMA controllers minimize CPU intervention further by having a chain address register that points to a chain control table in memory. The DMA controller then loads its own channel parameters from memory. Generally, the more sophisticated the DMA controller, the less servicing the CPU has to perform.
- A DMA controller has one or more status registers that are read by the CPU to determine the state of each DMA channel. The status register typically indicates whether a DMA request is asserted on a channel and whether a channel has reached TC. Reading the status register often clears the terminal count information in the register, which leads to problems when multiple programs are trying to use different DMA channels.

Steps in a Typical DMA cycle

- Processor completes the current bus cycle and then asserts the bus grant signal to the device.
- The device then asserts the bus grant ack signal.
- The processor senses in the change in the state of bus grant ack signal and starts listening to the data and address bus for DMA activity.
- The DMA device performs the transfer from the source to destination address.
- During these transfers, the processor monitors the addresses on the bus and checks if any location modified during DMA operations is cached in the processor.

- If the processor detects a cached address on the bus, it can take one of the two actions:
  - Processor invalidates the internal cache entry for the address involved in DMA write operation
  - Processor updates the internal cache when a DMA write is detected
- Once the DMA operations have been completed, the device releases the bus by asserting the bus release signal.
- Processor acknowledges the bus release and resumes its bus cycles from the point it left off.

8237 DMA Controller

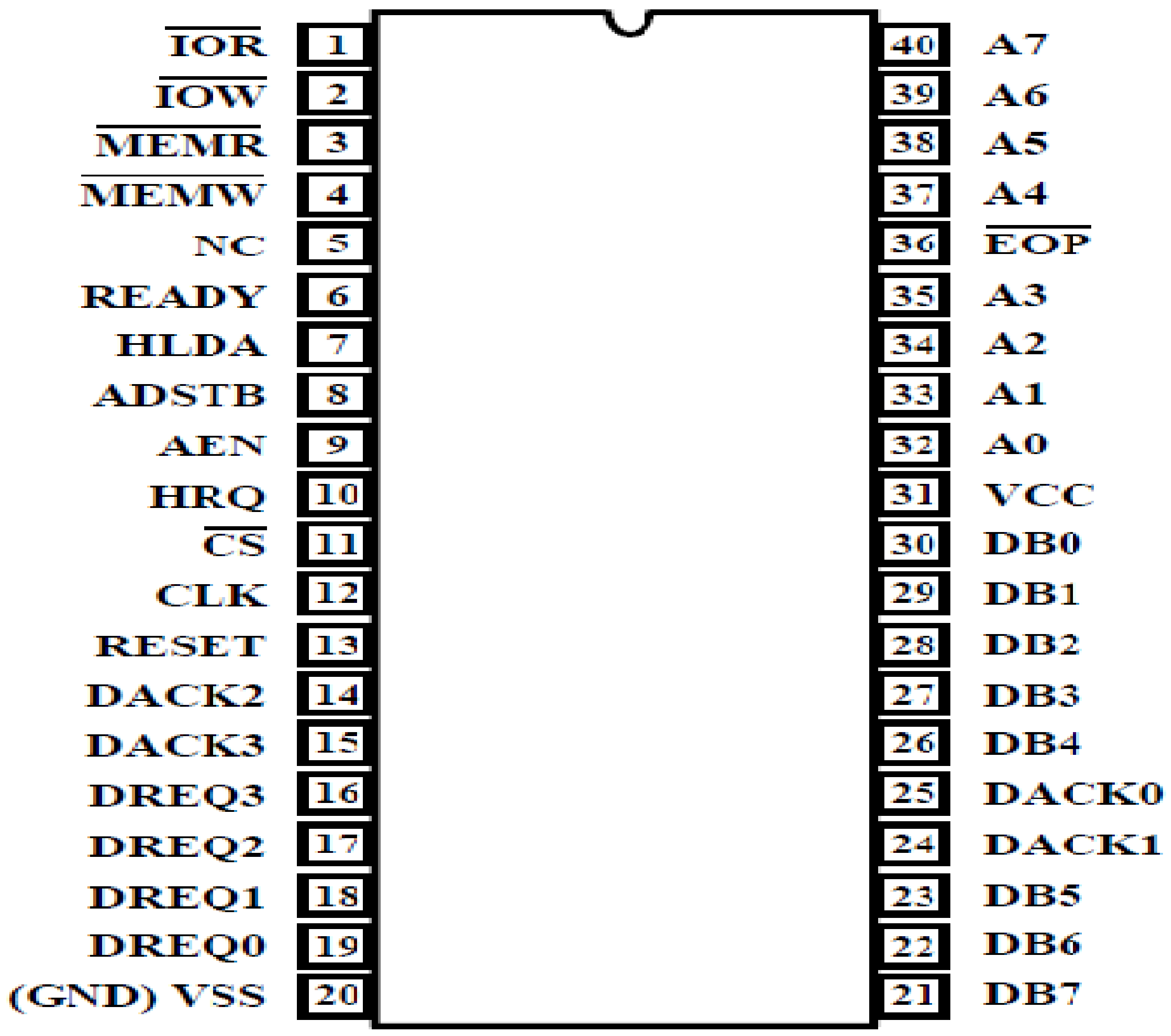


Fig. 16.4 The DMA pin-out

Functional Description

- **VCC**: is the +5V power supply pin
- **GND** Ground
- **CLK**: CLOCK INPUT: The Clock Input is used to generate the timing signals which control 82C37A operations.
- **CS**: CHIP SELECT: Chip Select is an active low input used to enable the controller onto the data bus for CPU communications.
- **RESET**: This is an active high input which clears the Command, Status, Request, and Temporary registers, the First/Last Flip-Flop, and the mode register counter. The Mask register is set to ignore requests. Following a Reset, the controller is in an idle cycle.
- **READY**: This signal can be used to extend the memory read and write pulses from the 82C37A to accommodate slow memories or I/O devices.

- **HLDA**: HOLD ACKNOWLEDGE: The active high Hold Acknowledge from the CPU indicates that it has relinquished control of the system busses.
- **DREQ0-DREQ3**: DMA REQUEST: The DMA Request (DREQ) lines are individual asynchronous channel request inputs used by peripheral circuits to obtain DMA service. In Fixed Priority, DREQ0 has the highest priority and DREQ3 has the lowest priority. A request is generated by activating the DREQ line of a channel. DACK will acknowledge the recognition of a DREQ signal. Polarity of DREQ is programmable. RESET initializes these lines to active high. DREQ must be maintained until the corresponding DACK goes active. DREQ will not be recognized while the clock is stopped. Unused DREQ inputs should be pulled High or Low (inactive) and the corresponding mask bit set.

- **DB0-DB7**: DATA BUS: The Data Bus lines are bidirectional three-state signals connected to the system data bus. The outputs are enabled in the Program condition during the I/O Read to output the contents of a register to the CPU. The outputs are disabled and the inputs are read during an I/O Write cycle when the CPU is programming the 82C37A control registers. During DMA cycles, the most significant 8-bits of the address are output onto the data bus to be strobed into an external latch by ADSTB. In memory-to-memory operations, data from the memory enters the 82C37A on the data bus during the read-from-memory transfer, then during the write-to-memory transfer, the data bus outputs write the data into the new memory location.
- **IOR:** READ: I/O Read is a bidirectional active low three-state line. In the Idle cycle, it is an input control signal used by the CPU to read the control registers. In the Active cycle, it is an output control signal used by the 82C37A to access data from the peripheral during a DMA Write transfer.

- **DB0-DB7**: DATA BUS: The Data Bus lines are bidirectional three-state signals connected to the system data bus. The outputs are enabled in the Program condition during the I/O Read to output the contents of a register to the CPU. The outputs are disabled and the inputs are read during an I/O Write cycle when the CPU is programming the 82C37A control registers. During DMA cycles, the most significant 8-bits of the address are output onto the data bus to be strobed into an external latch by ADSTB. In memory-to-memory operations, data from the memory enters the 82C37A on the data bus during the read-from-memory transfer, then during the write-to-memory transfer, the data bus outputs write the data into the new memory location.
- **IOR:** READ: I/O Read is a bidirectional active low three-state line. In the Idle cycle, it is an input control signal used by the CPU to read the control registers. In the Active cycle, it is an output control signal used by the 82C37A to access data from the peripheral during a DMA Write transfer.

- **IOW:** WRITE: I/O Write is a bidirectional active low three-state line. In the Idle cycle, it is an input control signal used by the CPU to load information into the 82C37A. In the Active cycle, it is an output control signal used by the 82C37A to load data to the peripheral during a DMA Read transfer.
- **EOP:** END OF PROCESS: End of Process (EOP) is an active low bidirectional signal. Information concerning the completion of DMA services is available at the bidirectional EOP pin. The 82C37A allows an external signal to terminate an active DMA service by pulling the EOP pin low. A pulse is generated by the 82C37A when terminal count (TC) for any channel is reached, except for channel 0 in memory-to-memory mode. During memory-to-memory

- transfers, EOP will be output when the TC for channel 1 occurs. The EOP pin is driven by an open drain transistor on-chip, and requires an external pull-up resistor to VCC. When an EOP pulse occurs, whether internally or externally generated, the 82C37A will terminate the service, and if auto-initialize is enabled, the base registers will be written to the current registers of that channel. The mask bit and TC bit in the status word will be set for the currently active channel by EOP unless the channel is programmed for autoinitialize. In that case, the mask bit remains clear.
- **A0-A3:** ADDRESS: The four least significant address lines are bidirectional three-state signals. In the Idle cycle, they are inputs and are used by the 82C37A to address the control register to be loaded or read. In the Active cycle, they are outputs and provide the lower 4-bits of the output address.

- **A4-A7:** ADDRESS: The four most significant address lines are three-state outputs and provide 4-bits of address. These lines are enabled only during the DMA service.
- **HRQ**: HOLD REQUEST: The Hold Request (HRQ) output is used to request control of the system bus. When a DREQ occurs and the corresponding mask bit is clear, or a software DMA request is made, the 82C37A issues HRQ. The HLDA signal then informs the controller when access to the system busses is permitted. For stand-alone operation where the 82C37A always controls the busses, HRQ may be tied to HLDA. This will result in one S0 state before the transfer.
- **DACK0-DACK3**: DMA ACKNOWLEDGE: DMA acknowledge is used to notify the individual peripherals when one has been granted a DMA cycle. The sense of these lines is programmable. RESET initializes them to active low.

- **AEN:** ADDRESS ENABLE: Address Enable enables the 8-bit latch containing the upper 8 address bits onto the system address bus. AEN can also be used to disable other system bus drivers during DMA transfers. AEN is active high.
- **ADSTB:** ADDRESS STROBE: This is an active high signal used to control latching of the upper address byte. It will drive directly the strobe input of external transparent octal latches, such as the 82C82. During block operations, ADSTB will only be issued when the upper address byte must be updated, thus speeding operation through elimination of S1 states. ADSTB timing is referenced to the falling edge of the 82C37A clock.
- **MEMR:** MEMORY READ: The Memory Read signal is an active low three-state output used to access data from the selected memory location during a DMA Read or a memory-to-memory transfer.

- **MEMW** MEMORY WRITE: The Memory Write signal is an active low three-state output used to write data to the selected memory location during a DMA Write or a memory-to-memory transfer.
- **NC:** NO CONNECT: Pin 5 is open and should not be tested for continuity.

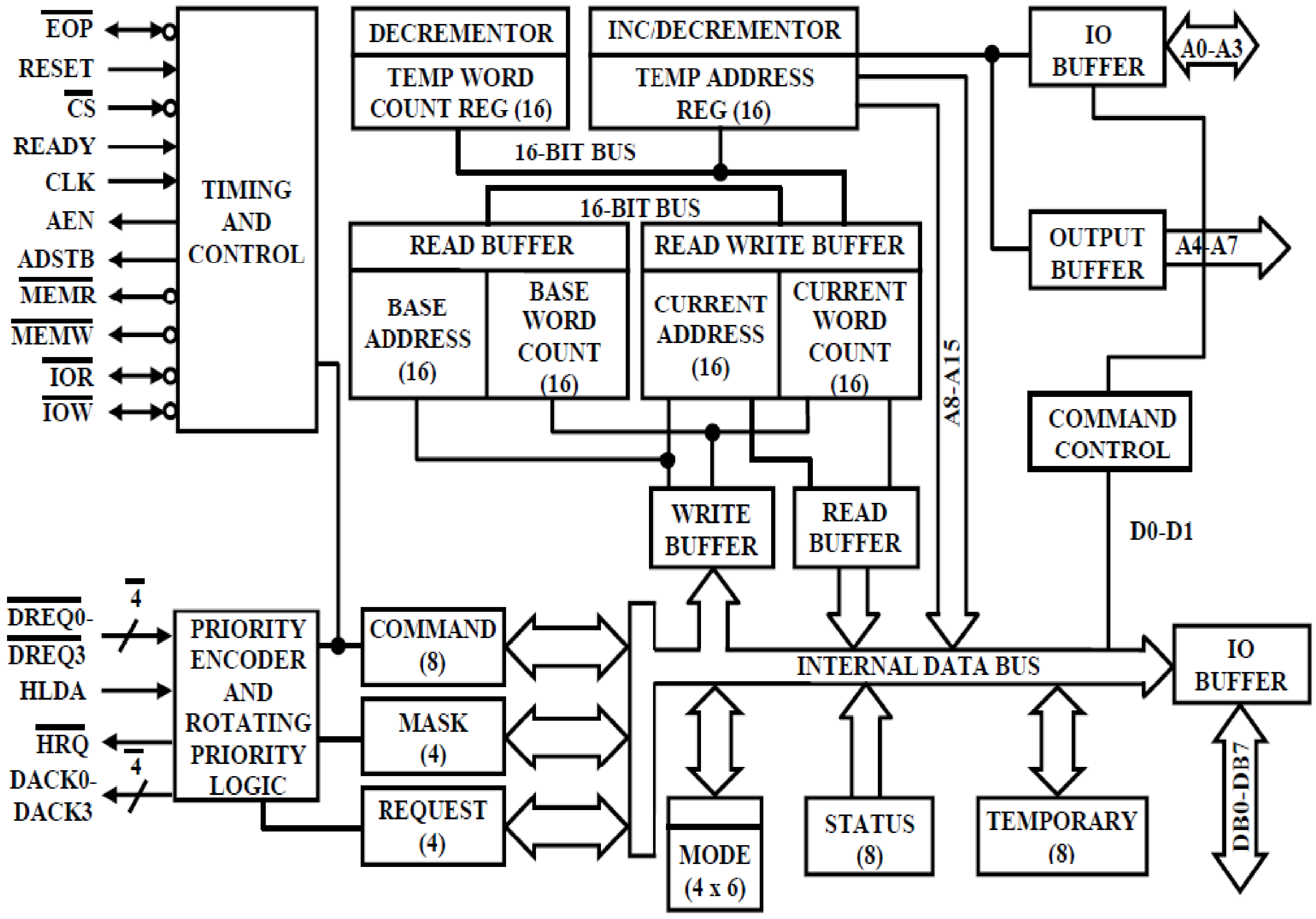
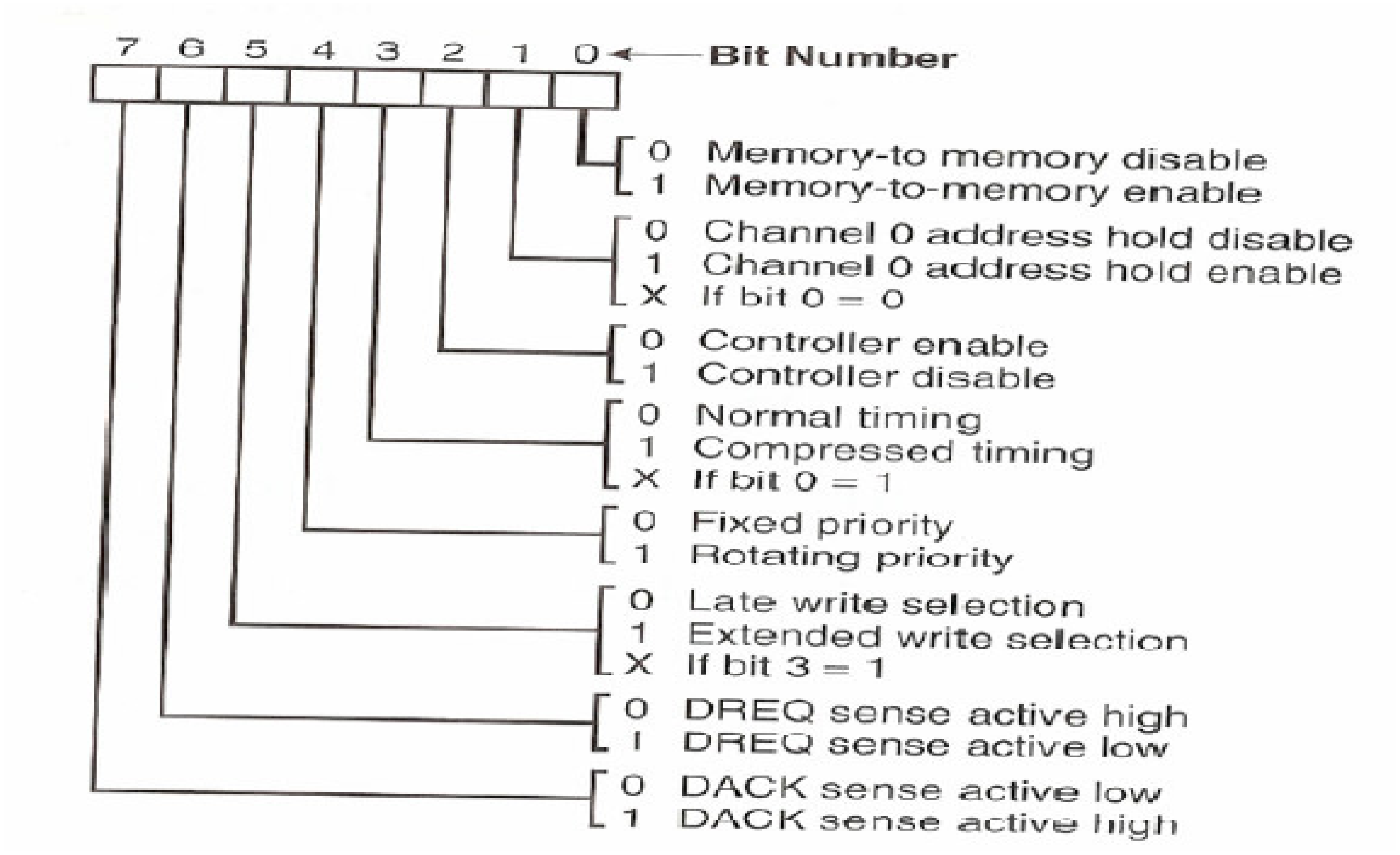


Fig. 16.5 The 8237 Architecture

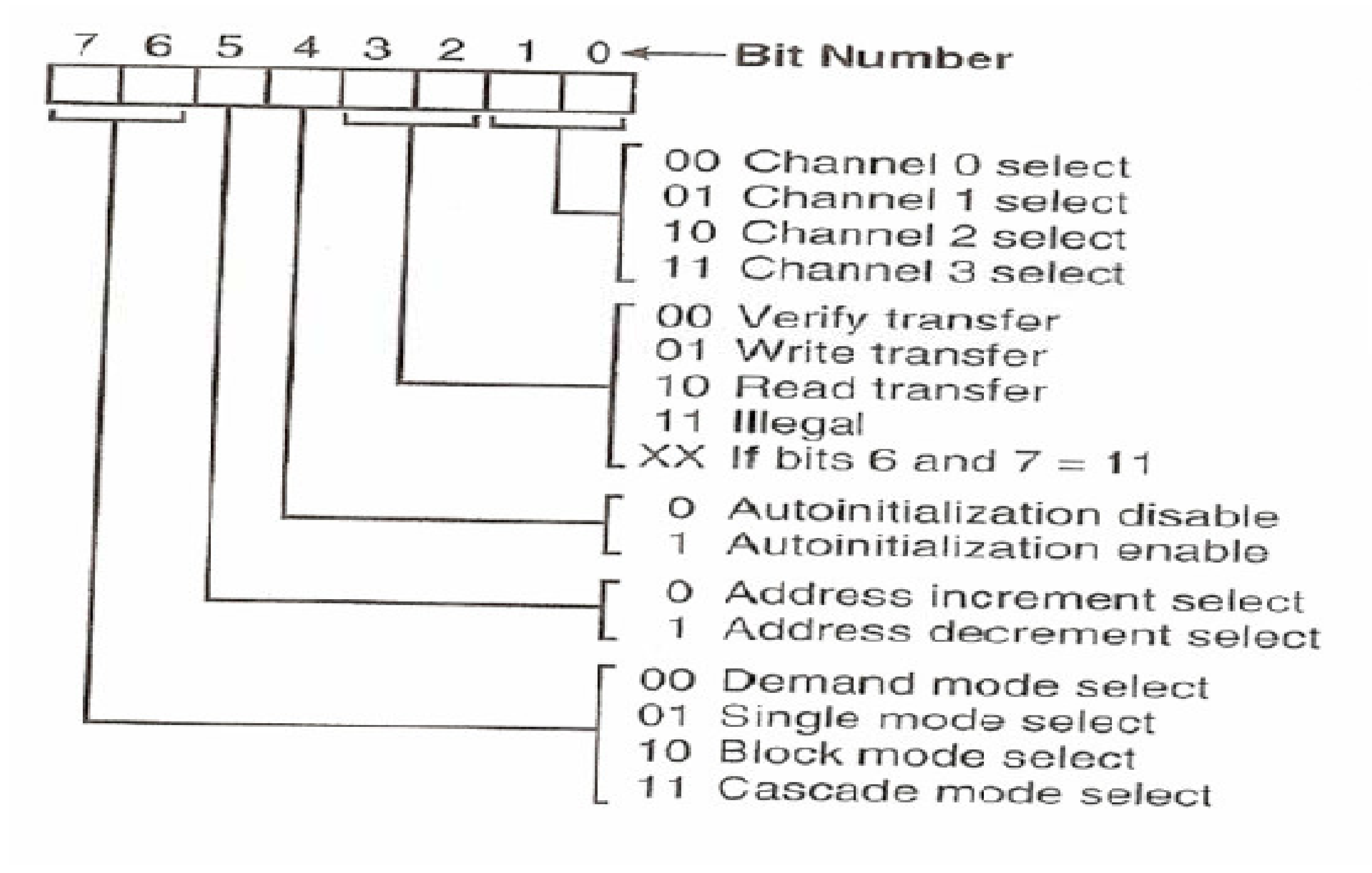
Internal Registers

- **CAR: Current Address Register**
- 16-bit for each channel
- Auto increment or decrement
- **CWCR:Current Word Count Register**
- Number loaded is one less than count
- **BA and BWC: base registers**
- Used in auto initialization mode
- Reload the original values

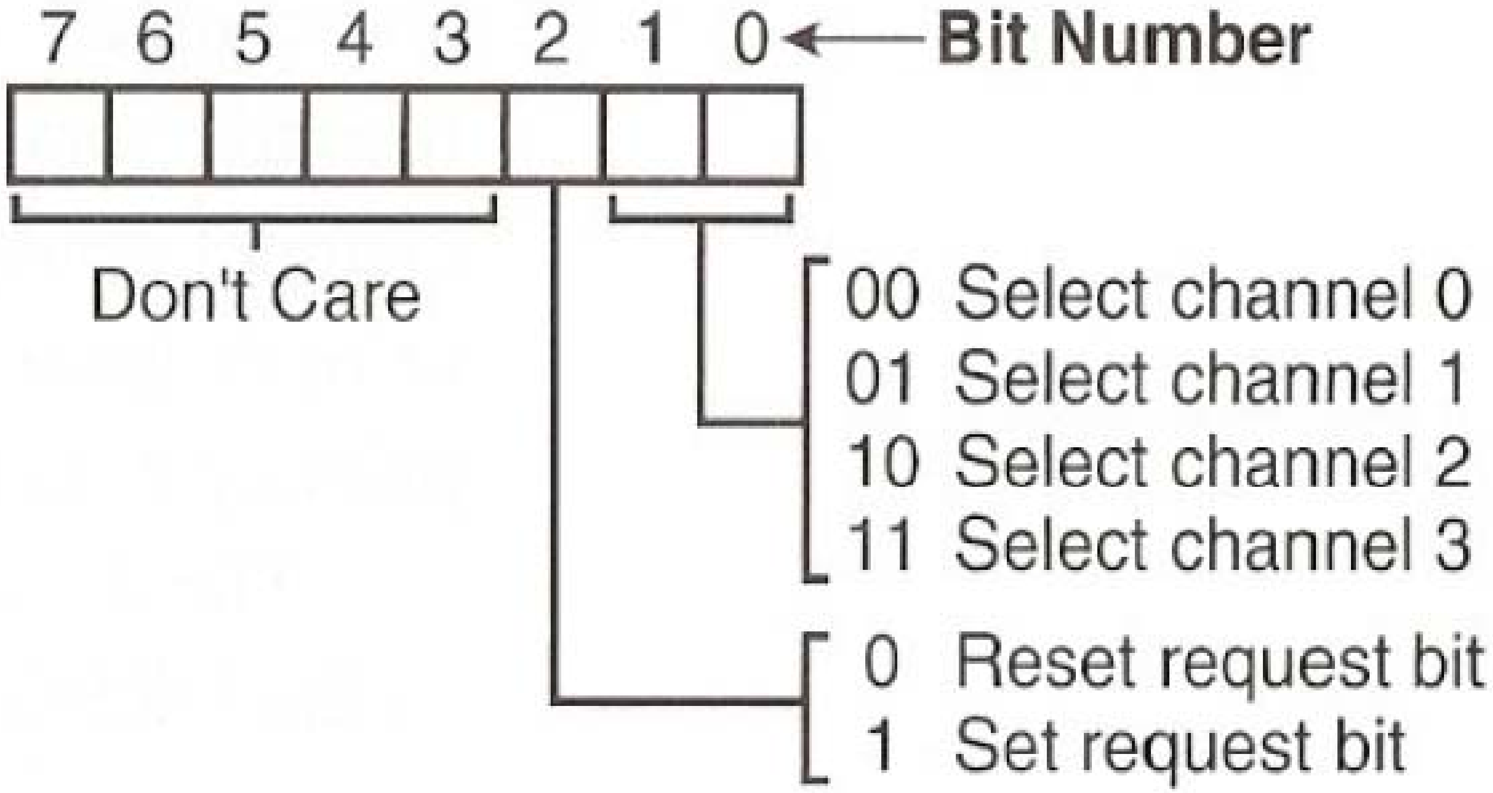
DMA Command Register



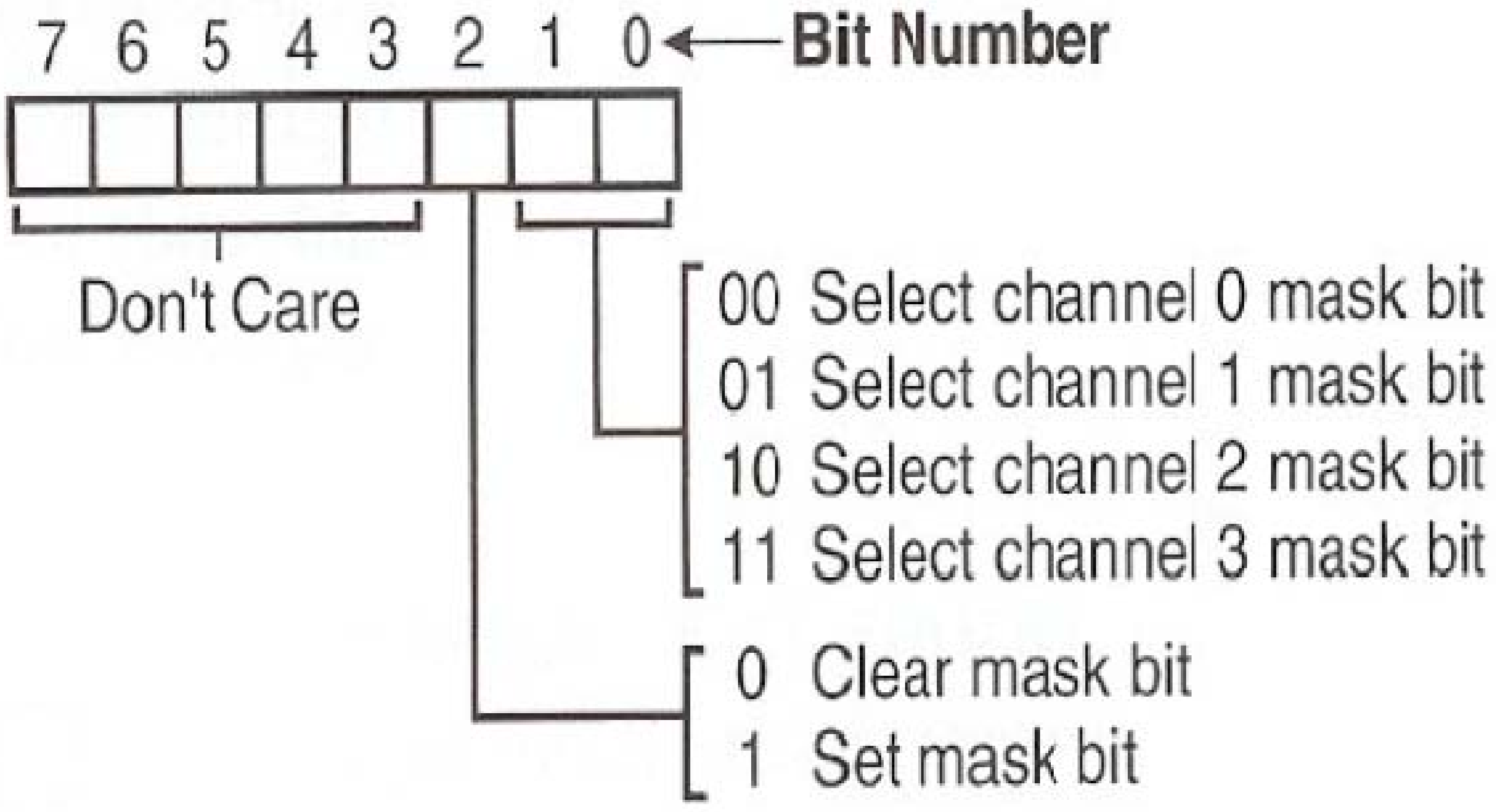
DMA Mode register



DMA Request Register

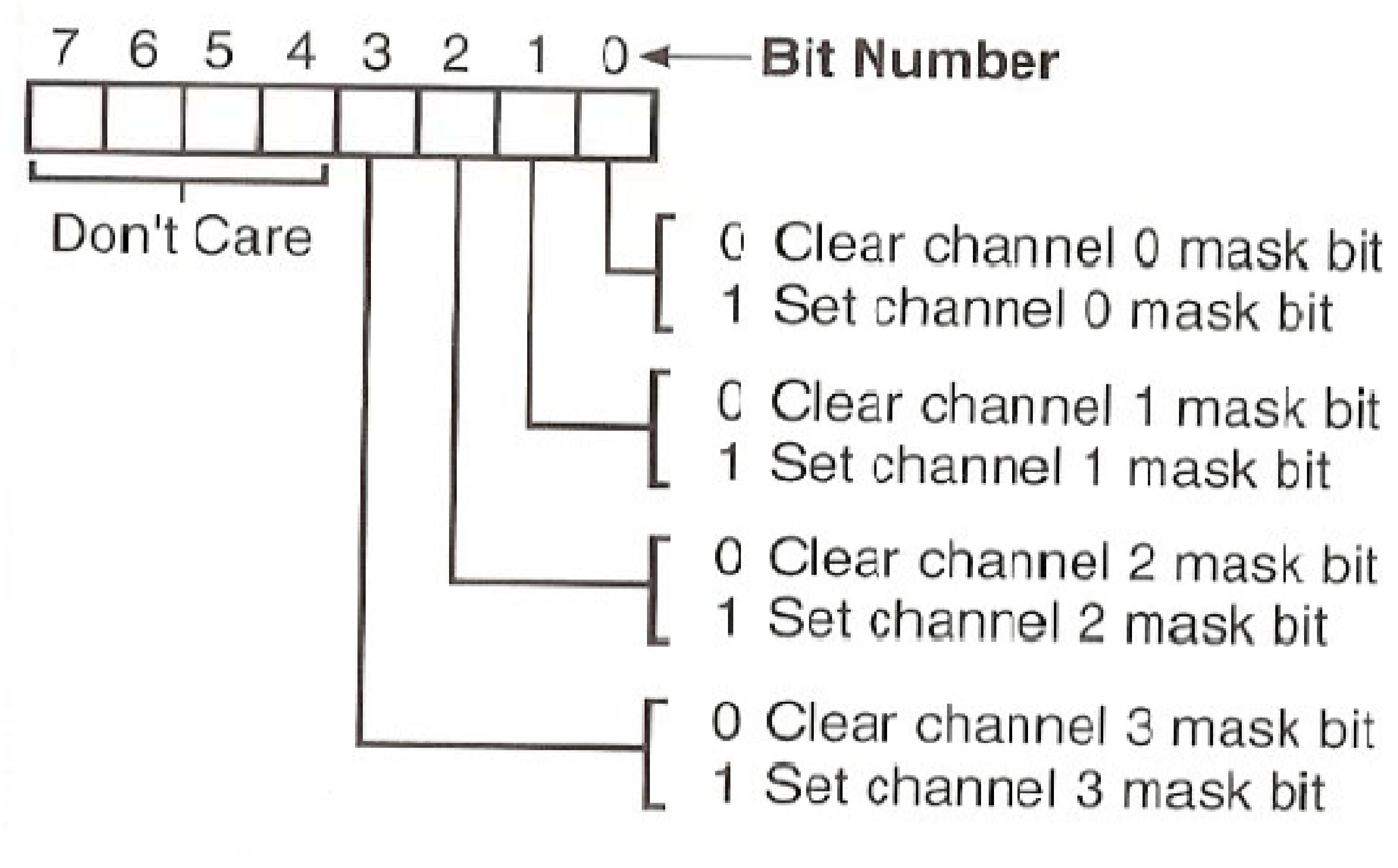


Mask Set/Rest Register

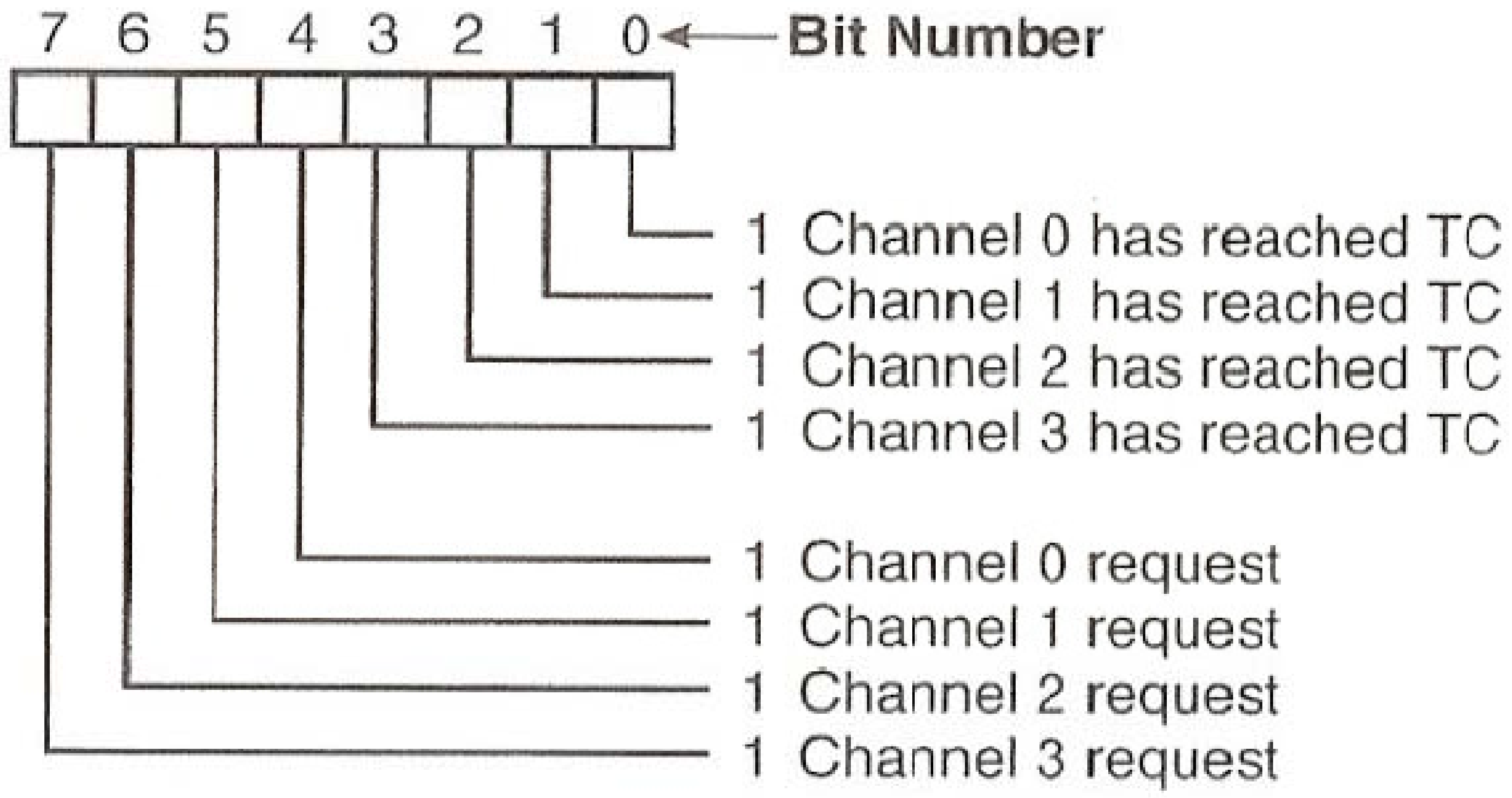


7 6 5 4 3 2 1 0
Bit Number
Don't Care
00 Select channel 0 mask bit
01 Select channel 1 mask bit
10 Select channel 2 mask bit
11 Select channel 3 mask bit
0 Clear mask bit
1 Set mask bit

DMA Mask register



Status Register


7 6 5 4 3 2 1 0
Bit Number
1 Channel 0 has reached TC
1 Channel 1 has reached TC
1 Channel 2 has reached TC
1 Channel 3 has reached TC
1 Channel 0 request
1 Channel 1 request
1 Channel 2 request
1 Channel 3 request