Computer Organization

Chap1: Introduction

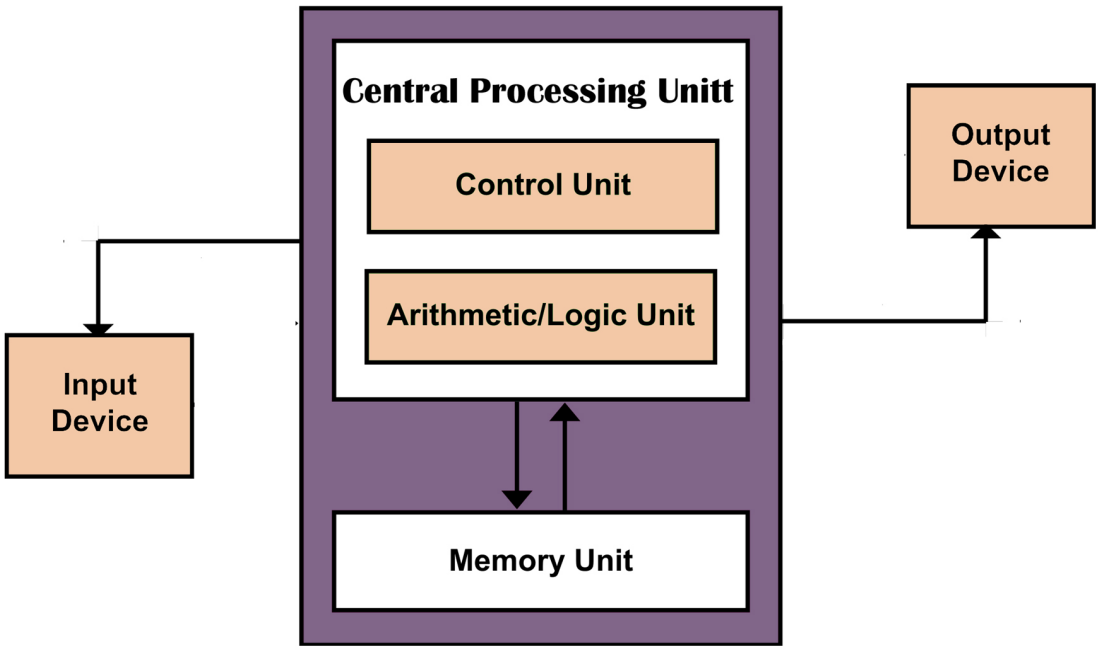
Goal

- Understand the computer organization
- Understand how different computer components interact
- Understand the programming interface of computer

Computer Organization

Von Neumann Architecture

- IO devices: Input&Output
- Memory: Memory
- CPU: ALU&Control unit



Von Neumann Architecture

Key **Abstraction**

- Data
- Instruction
- Sequential execution model

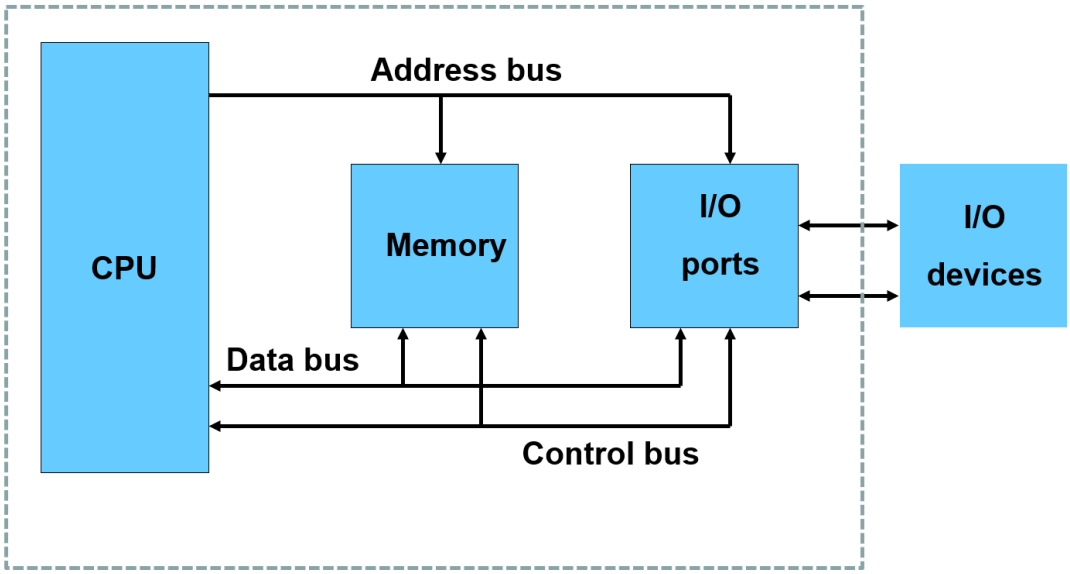
Key **Concepts**

- Instruction and Data stored in single R/W memory
- Contents of memory addressable by location, regardless of data type
- Sequential execution

Harvard Architecture: split single memory into data memory and instructions memory

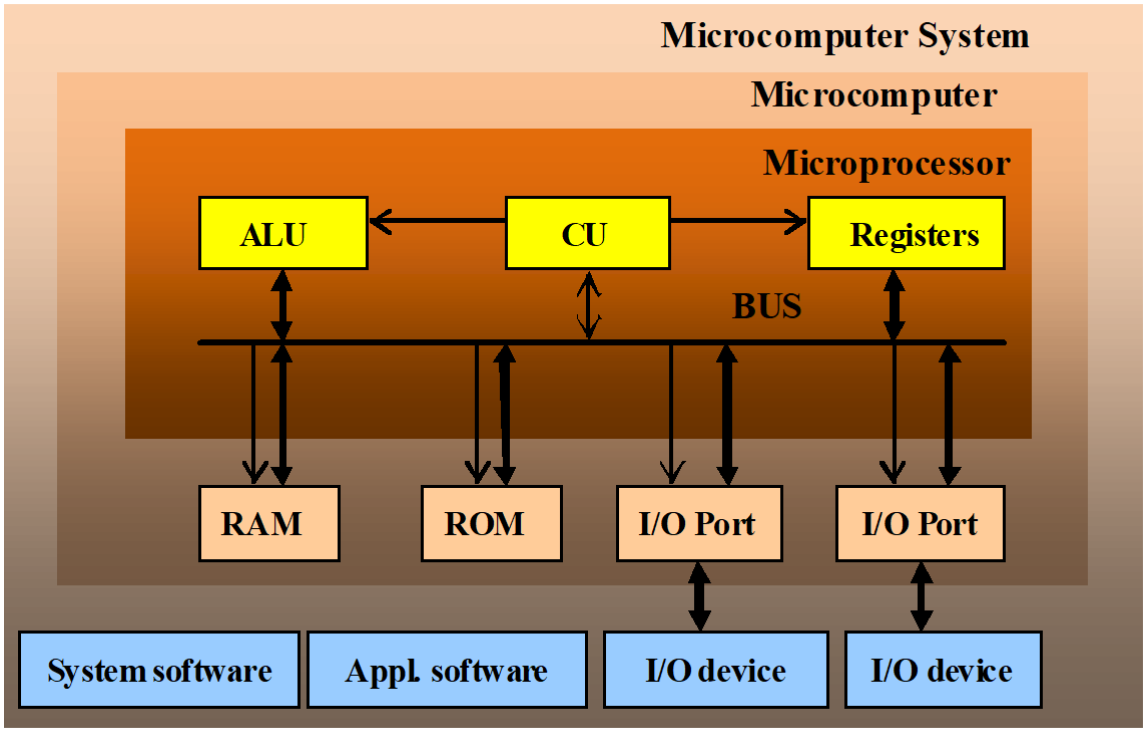
Microcomputer

- CPU: processes information stored in the memory
- Memory: stores both instructions and data.
- IO ports: provide a means of communicating with the CPU
- BUS: interconnecting all parts together
    - Address bus
    - Data bus
    - Control bus



Microcomputer **System**

- Microcomputer
- Peripheral I/O devices
- System and Application software



CPU

Transistor ⟶ Logical gate ⟶ Instruction Set Architecture (ISA)

Key **Concepts**

- core (each core has ALU, CU and register)
- clock
- ISA (x86/ARM/RISC-V)

ALU

multifunctional calculator

CU

instruction decoder + program counter

Instruction Set

- CISC (complex)

  e.g. x86

  - variable instruction length
  - variable execution time
  - more formats
  - upwardly compatible (兼容)
- RISC (reduced)

  e.g. RISC-V, ARM

  - fixed size
  - fixed execution time and easy to pipeline
  - less formats

Memory

Word:# bits that a CPU can process at one time. It depends on the width of registers and data bus. (16 bits in 8086)

Bit extension (width): need bigger unit of transfer than that of given memory chips

Word extension (number): need larger number of memory words than that of given memory chips

Memory Hierarchy: cache ⟶ RAM ⟶ disk

Bus

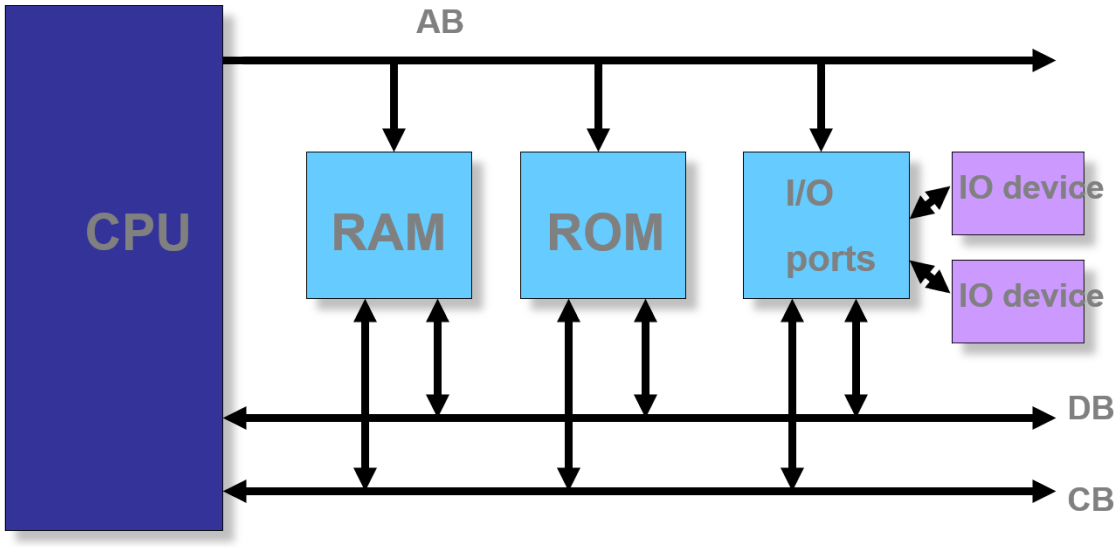
Bus is a communication pathway connecting two or more devices.

A **shared** transmission medium: one device at a time

System bus: connects major computer components (processor, memory, I/O)

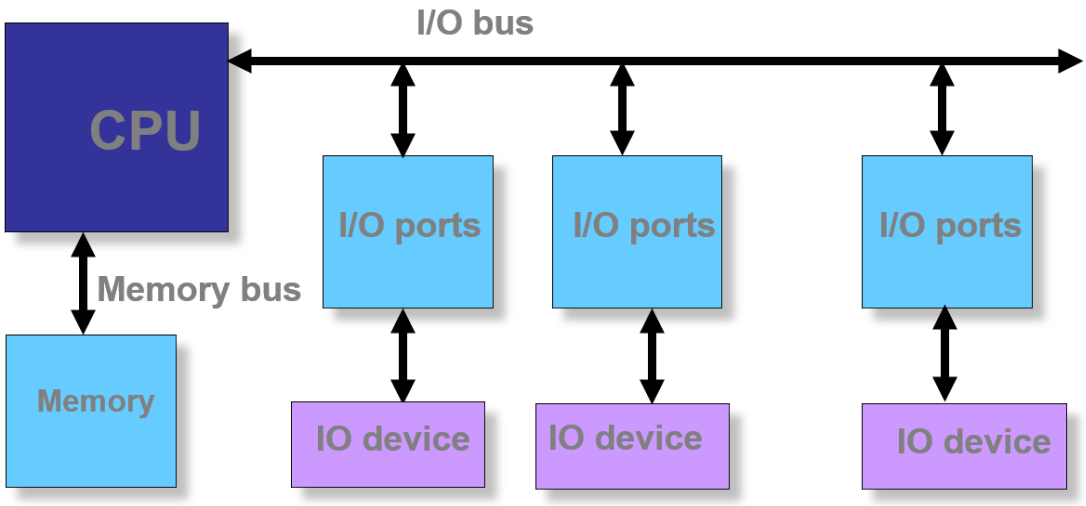
Single-bus Structure

A bus connects all modules. Poor throughput.

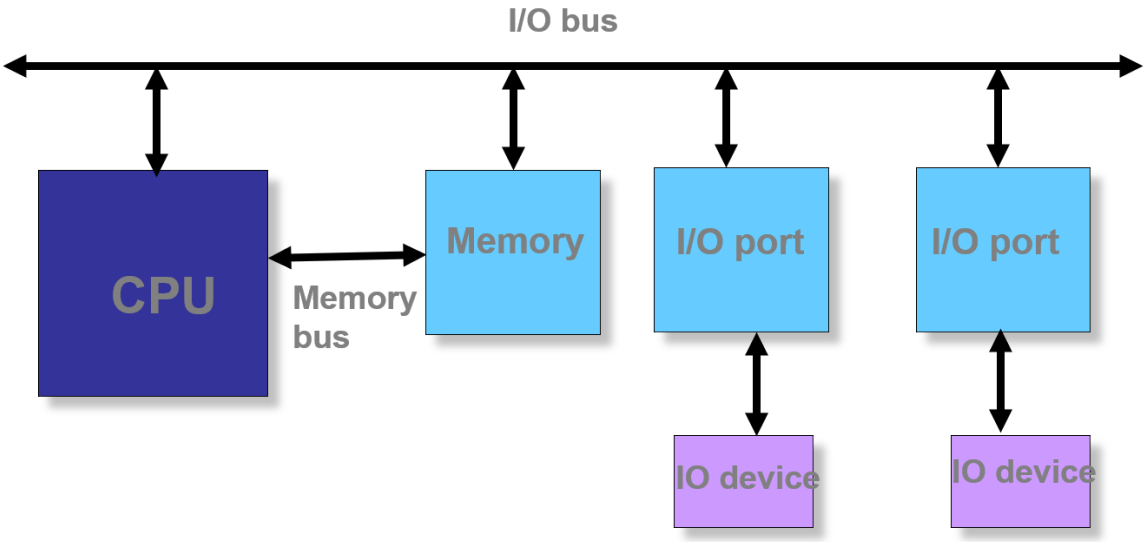


CPU-Central Dual-Bus Structure

A dedicated bus between CPU and memory, and a dedicated bus between CPU and I/O devices. Information between memory and IO has to go through CPU.



Memory-Central Dual-Bus Structure



Data Bus

- Used to provide a path for moving data between system modules.
- Bidirectional: CPU $\longleftrightarrow$ memory
- The width of data bus is usually the same as register. (i.e. the width of a **word**)

  Determines how much data the processor can read or write in one memory or I/O cycle

Address Bus

- Used to designate the source or destination of the data on the data bus
- Unidirectional: CPU $\longrightarrow$ memory/IO
- The width of address bus n determines the total number of addressable memory locations, which is 2^n.

Control Bus

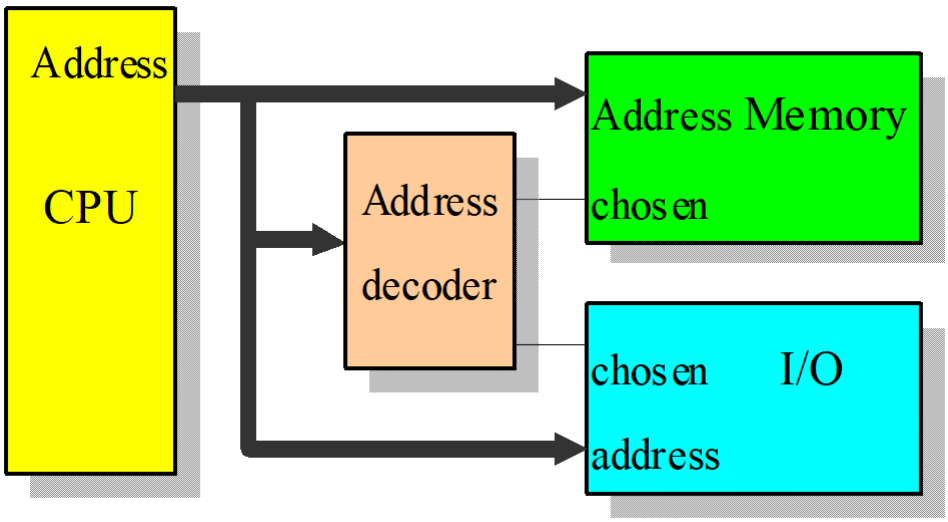
- Used to control each module and the use of data and address buses

  Mainly used to send command signal.
- Two sets of unidirectional control signals
  - Command signal: CPU $\longrightarrow$ memory/IO
  - State signal: memory/IO $\longrightarrow$ CPU

IO

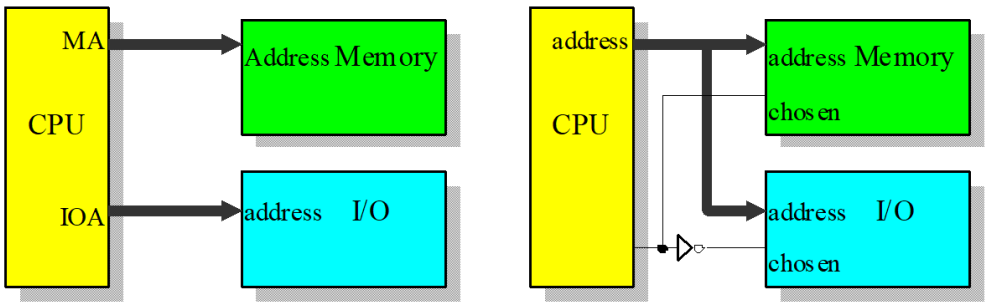
Memory-mapped IO

- Single address space for both memory and I/O
- Status and data registers of I/O modules are treated as memory locations
- Using the same machine instructions to access both



Isolated IO

- Two separate address spaces for memory and I/O modules
- Using different sets of accessing instructions



Dedicated address lines **Multiplexing address lines**

Embedded System

Microcontroller

A microcontroller has a CPU in addition to a fixed amount of RAM, ROM, I/O ports on one single chip.

Microcontroller is ideal for applications in which cost and space are critical.

Embedded System

An embedded system uses a microcontroller or a microprocessor to do one task and one task only.

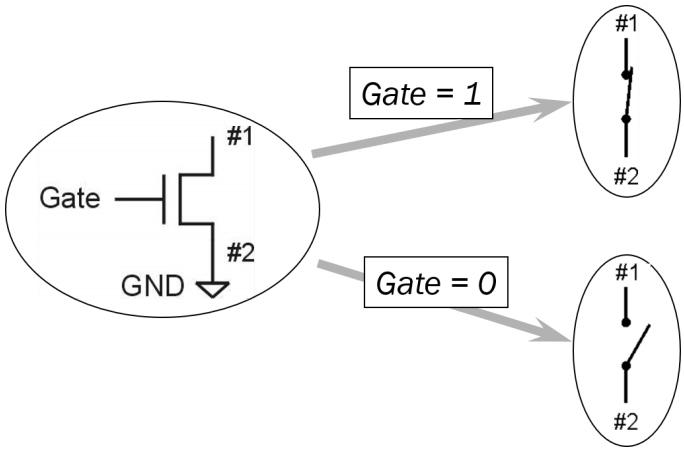
Chap2: Digital Logic

Transistor ⟶ logic gates (AND, OR, NOT) ⟶ higher-level structures (adder, multiplexer, decoder, register...) ⟶ processor

MOS

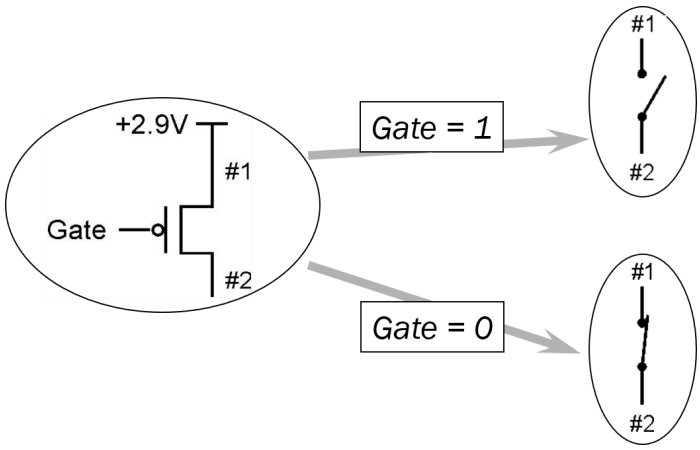
n-type MOS

- Attached to GND
- Pulls output voltage DOWN when input is 1



p-type MOS

- Attached to + voltage
- Pulls output voltage UP when input is zero



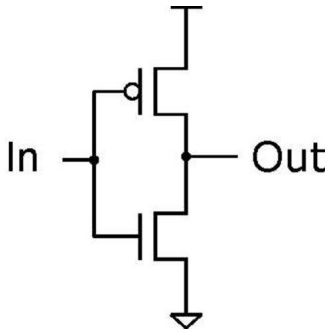
Logic Gates

Use switch behavior of MOS to implement logical functions

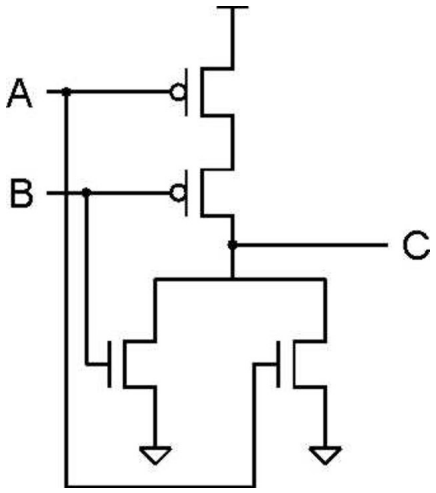
CMOS(Complementary MOS): use both n-type and p-type MOS

Note: For all inputs, make sure that output is either connected to GND or to +, but not both!

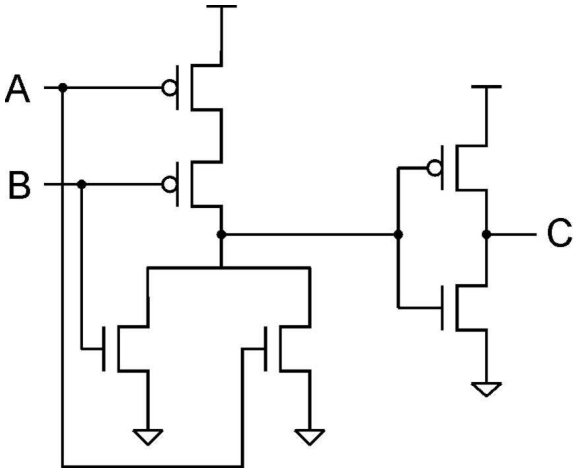
NOT



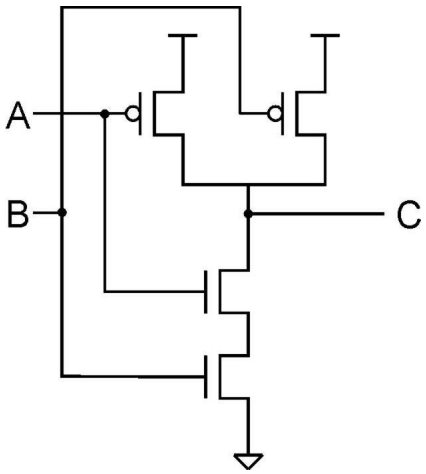
NOR



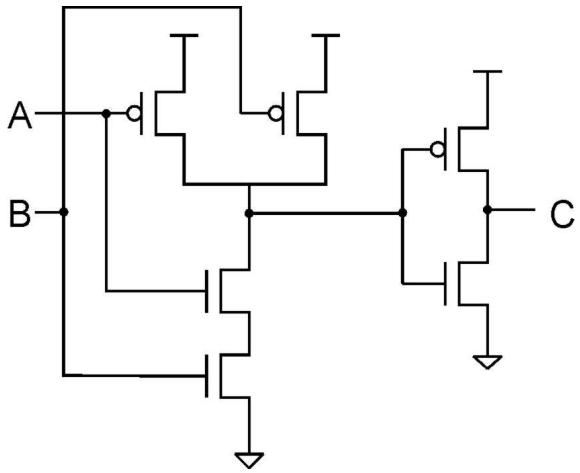
OR



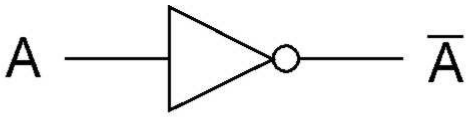
NAND



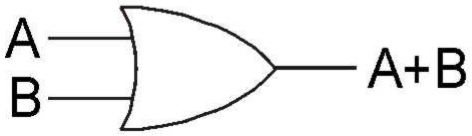
AND



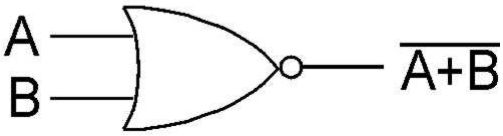
Notations



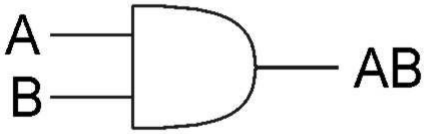
NOT



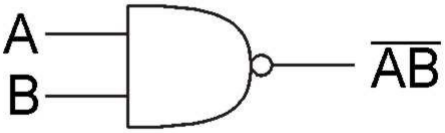
OR



NOR



AND



NAND

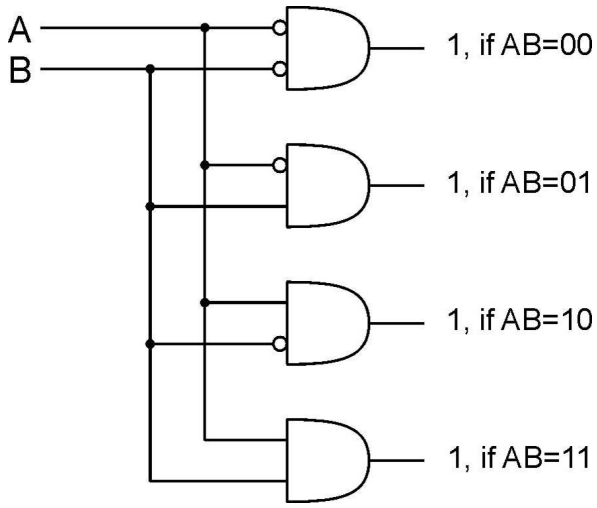
High Level Structure

- Combinational Logic Circuit
  - output depends only on the current inputs
  - stateless
- Sequential Logic Circuit
  - output depends on the sequence of inputs (past and present)
  - stores information (state) from past inputs

Combinational Logic Examples

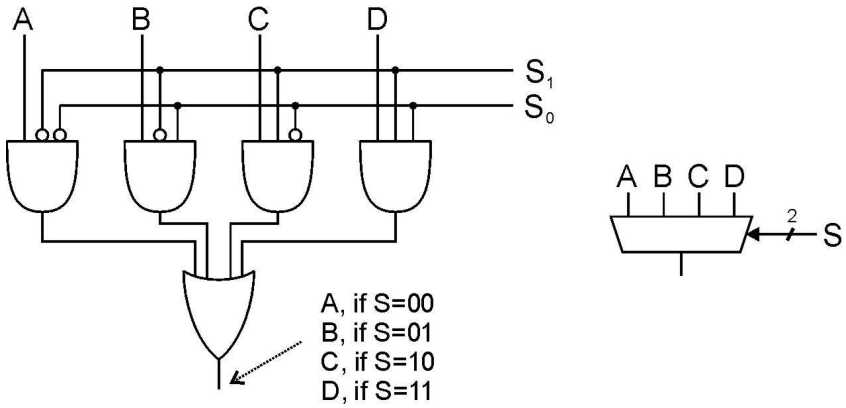
Decoder

exactly one output is 1 for each possible input pattern



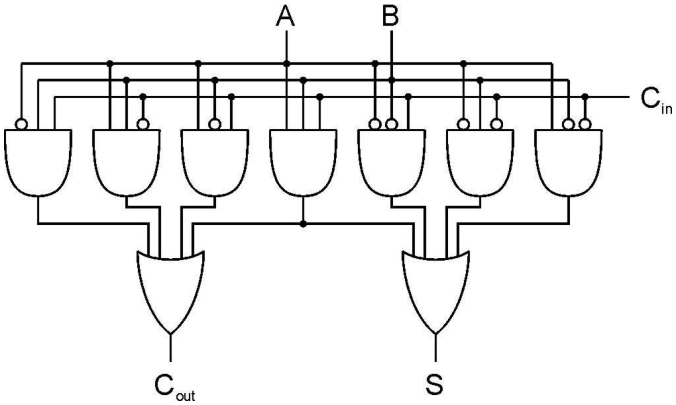
Multiplexer(MUX)

input + decoder(selector)



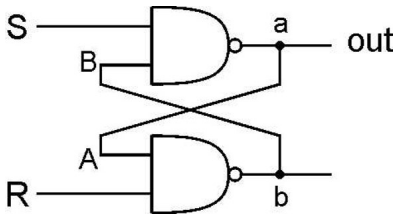
Full Adder

Add 2 bits and carry-in, produce one-bit sum and carry-out.



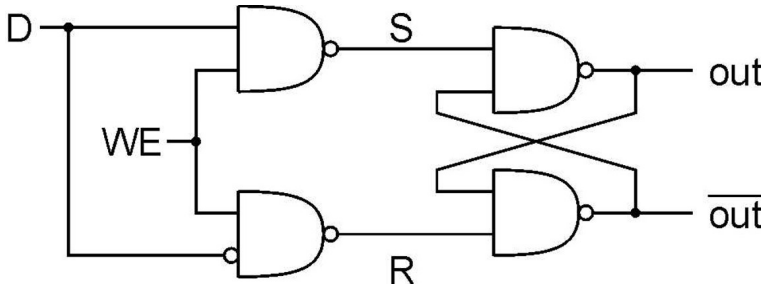
Sequential Logic Examples

R-S Latch(锁存)



R	S	Out
0	0	hold
0	1	set to 1
1	0	set to 0
1	1	x

D-Latch



inputs: D(data) and WE(write enable)

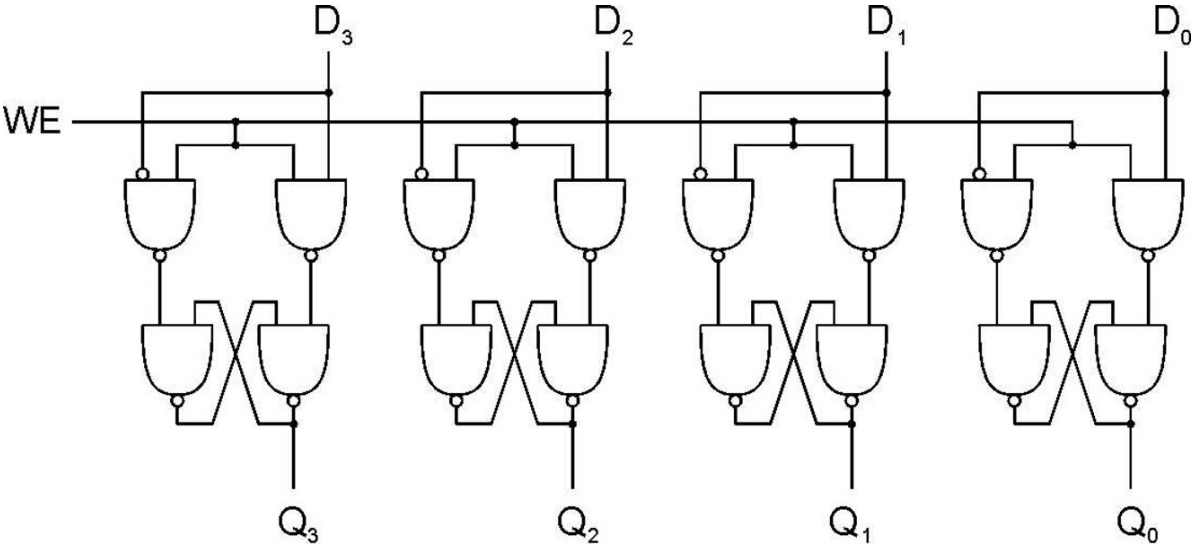
- WE=1, latch is set to D

  S=not D, R=D
- WE=0, latch holds

  S=R=1

Register

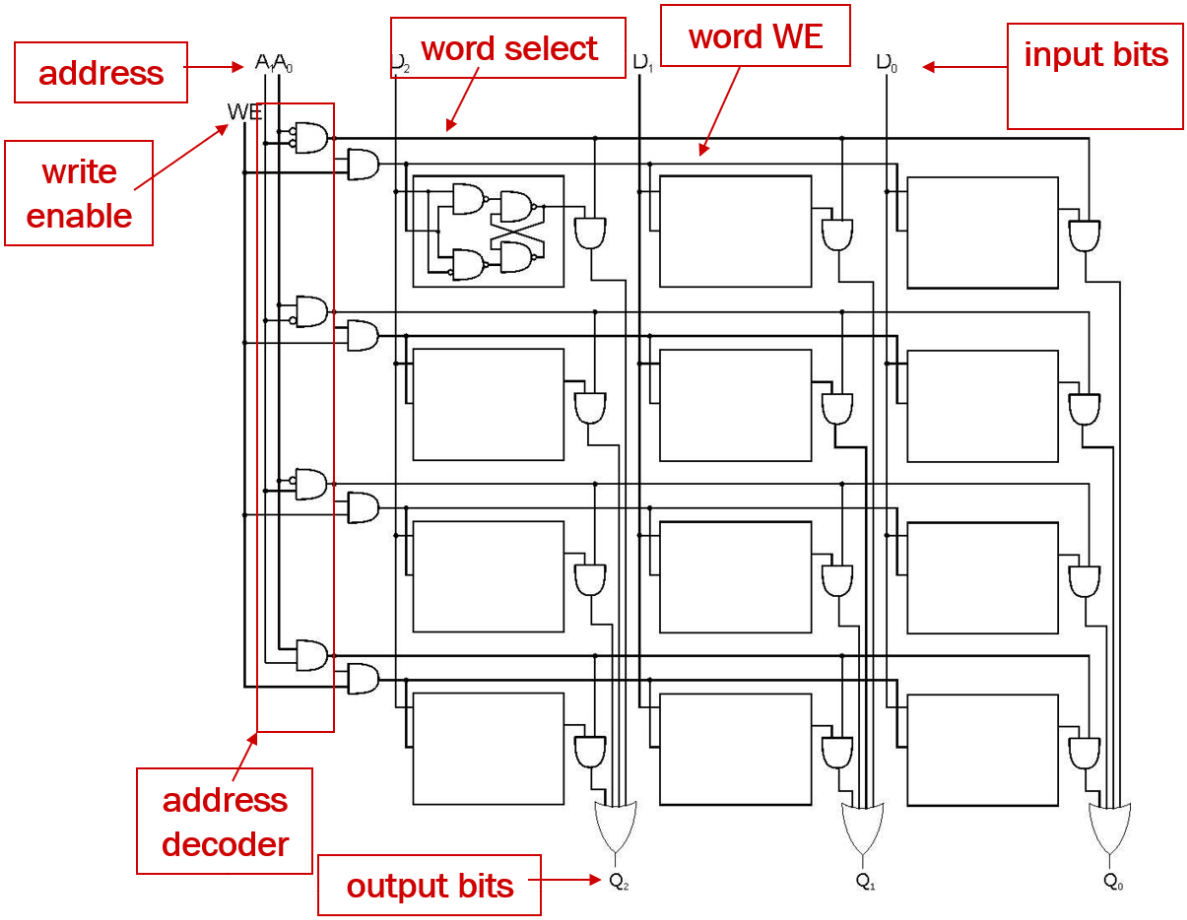
A collection of D-latches



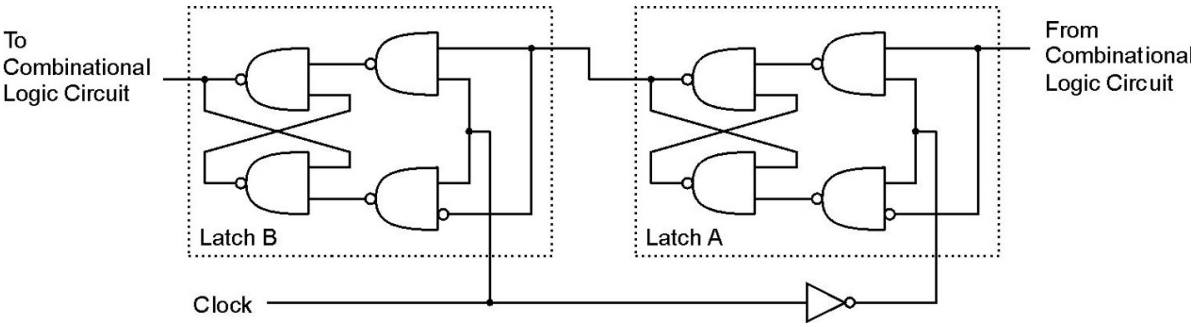
Memory

A $2^n \times m$ array of D-latches. Decoder + Register

Not the way actual memory is implemented.



Master-Slave Flipflop (主从触发器)



clock=0, In → A; clock=1, A → B

write data to flipflop at positive edge of clock (上升沿触发)

Chap3a: Memory

Memory Taxonomy & Characteristics

Location

- CPU: registers
- internal: cache, main memory
- external: disks, CD and DVD

Unit of Transfer

- Internal: usually a word, governed by the data bus width
- External: usually a block which is much larger than a word.

  memory transfer takes much more time

Addressable unit

Smallest location which can be uniquely addressed.

- Normally a byte for internal memory
- Cluster of disks

Access Methods

- Sequential

  Access start at the beginning and read through in order. (tape)

- Direct

  Individual blocks have unique address. Access is by jumping to vicinity plus sequential search. (disk)

- Random

  Individual addresses identify locations exactly. Access time is independent of location or previous access. (RAM)

- Associative

  Data is located based on a portion of its contents rather than its address. Access time is independent of location or previous access. (cache)

Performance Metrics

- Access time

  time between presenting the address and getting the valid data.

- Memory cycle time: time may be required for the memory to "recover" before next access.

  cycle time = access + recovery

- Transfer rate

  rate at which data can be moved. (unit: transfer per second)

- Transfer bandwidth

  equals to transfer rate * transfer unit size (unit: bytes per second)

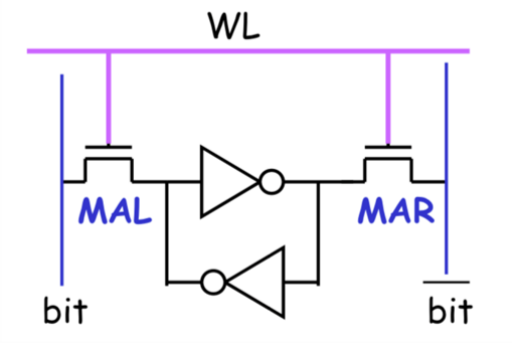
Memory Basics

RAM (Random Access) & ROM (Read Only)

- Volatility of Memory（易失性）
  - Volatile memory loses data over time or when power is removed. (RAM)
  - Non-volatile memory stores date even when power is removed. (ROM)
  - Static: holds data as long as power is applied. (SRAM)
  - Dynamic: will lose data unless refreshed periodically. (DRAM)

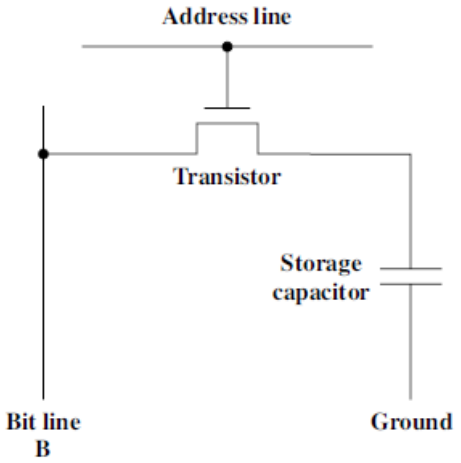
SRAM

No refreshing needed. More complex construction. 6T-SRAM (6 transistors)

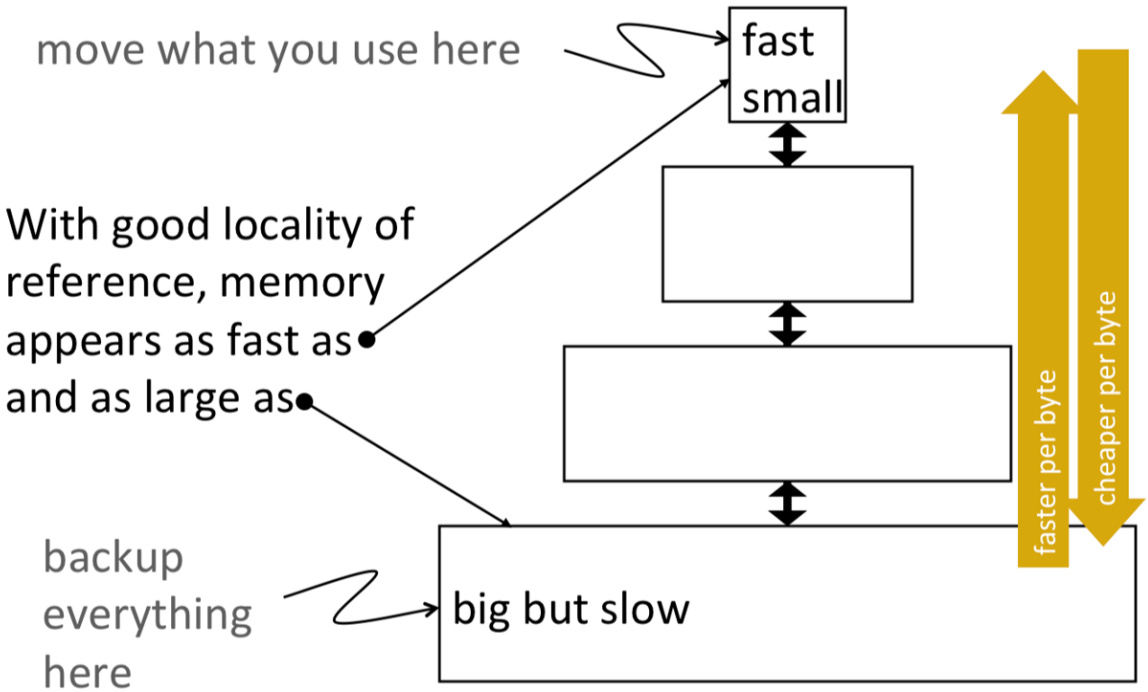


DRAM

Bits stored as charge in capacitors. Charges leak. Need refresh circuits. Simpler construction. Need 'recovery' part



Memory Hierarchy in Computer System

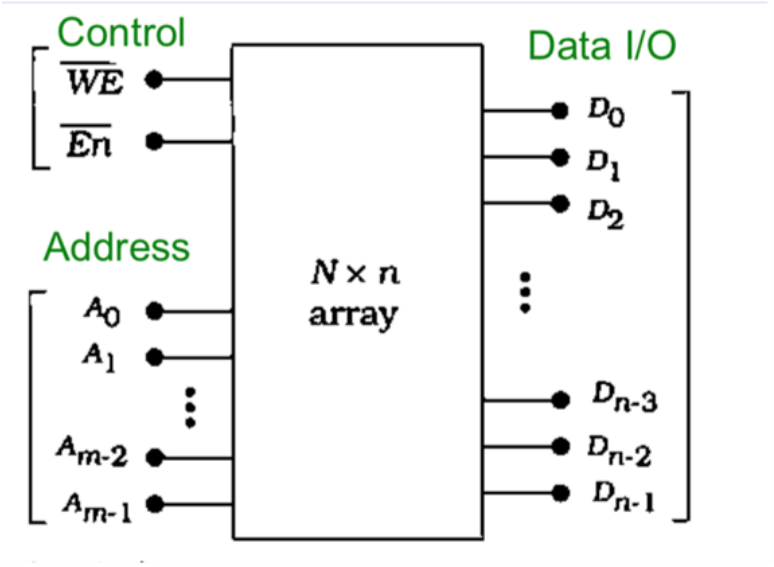


RAM organization

Memory Chip

N x n memory chip

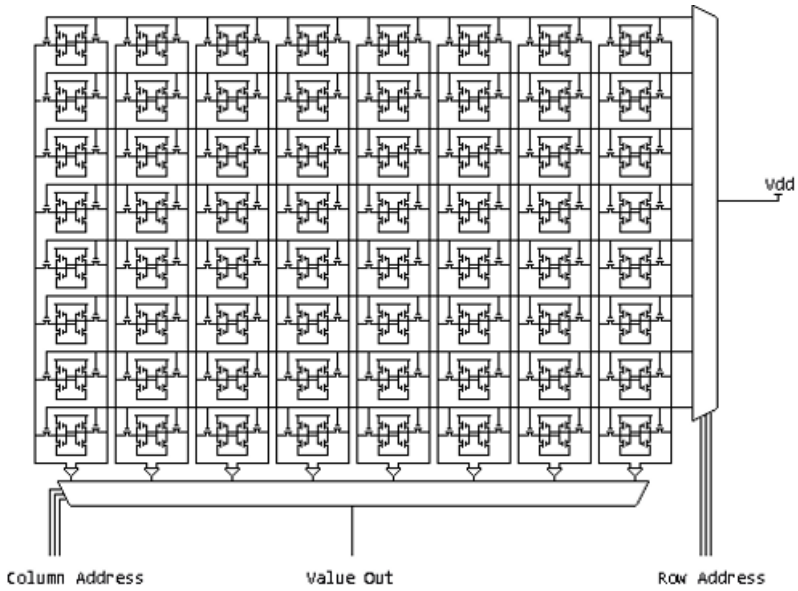
- n = chip word width. (not CPU word)
- N = number of n-bit words.



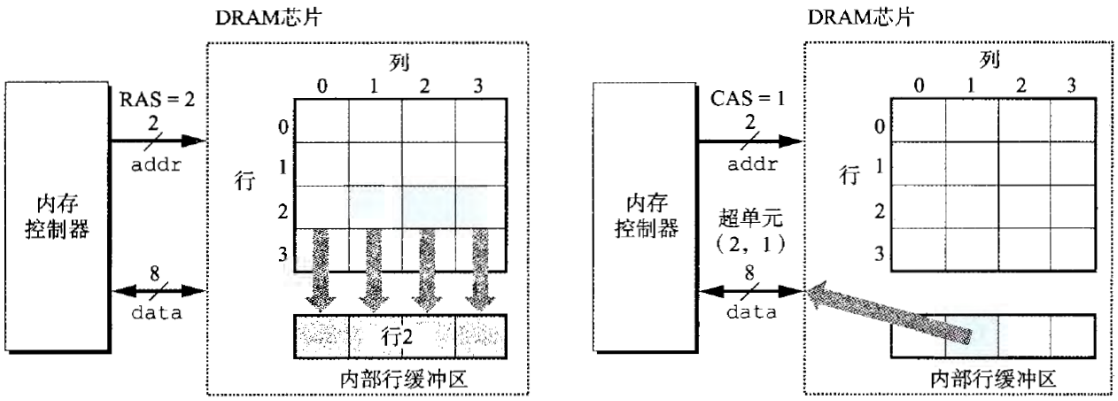
- Data: $D_i \cdots$
- Address: $A_i \cdots$
- Control
  - WE(write enable). Read: WE=1; Write: WE=0
  - En: block enable (assert low)

Physical view of memory chip

$N_R \times N_C$ array of 1-bit cells



Multiplex(复用) row address and column address to reduces # pins.



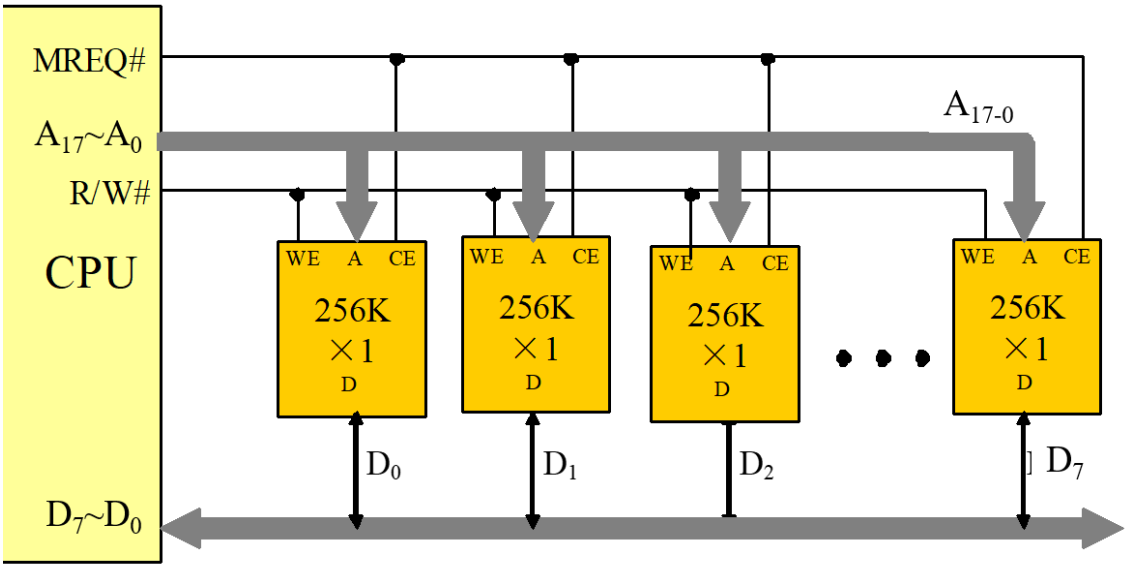
a）选择行2（RAS请求）　　b）选择列1（CAS请求）

Memory Module Extension

Bit Extension

needs bigger unit of transfer than that of given memory chips

Use 256Kx1-bit chips to build 256Kx8-bit memory

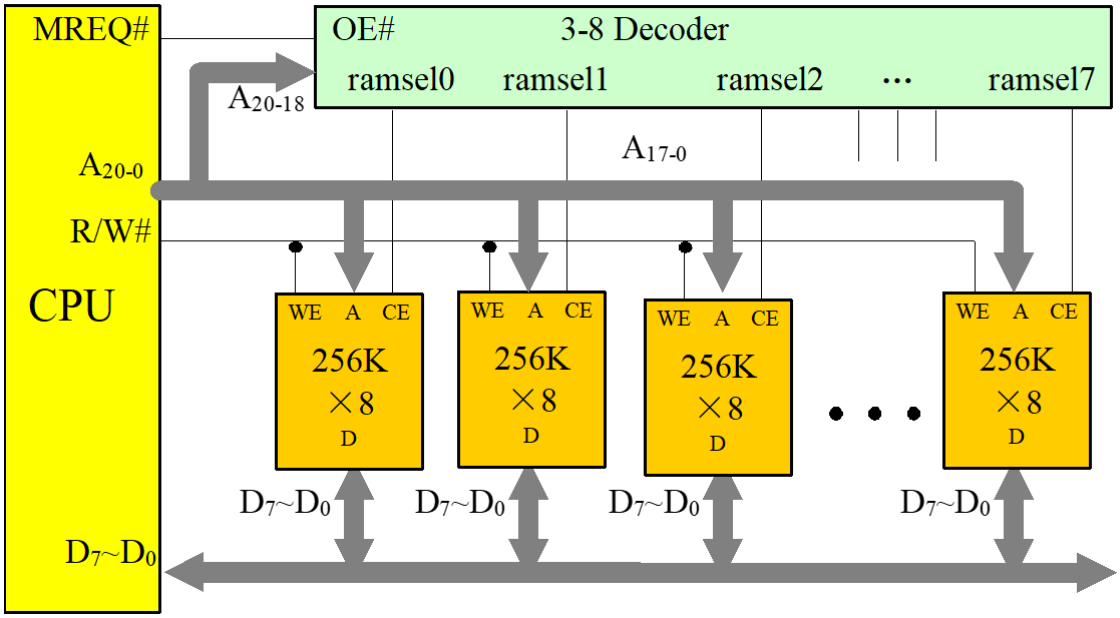


Word Extension

needs larger number of memory words than given memory chips

Use 256Kx8-bit chips to build 2Mx8-bit memory

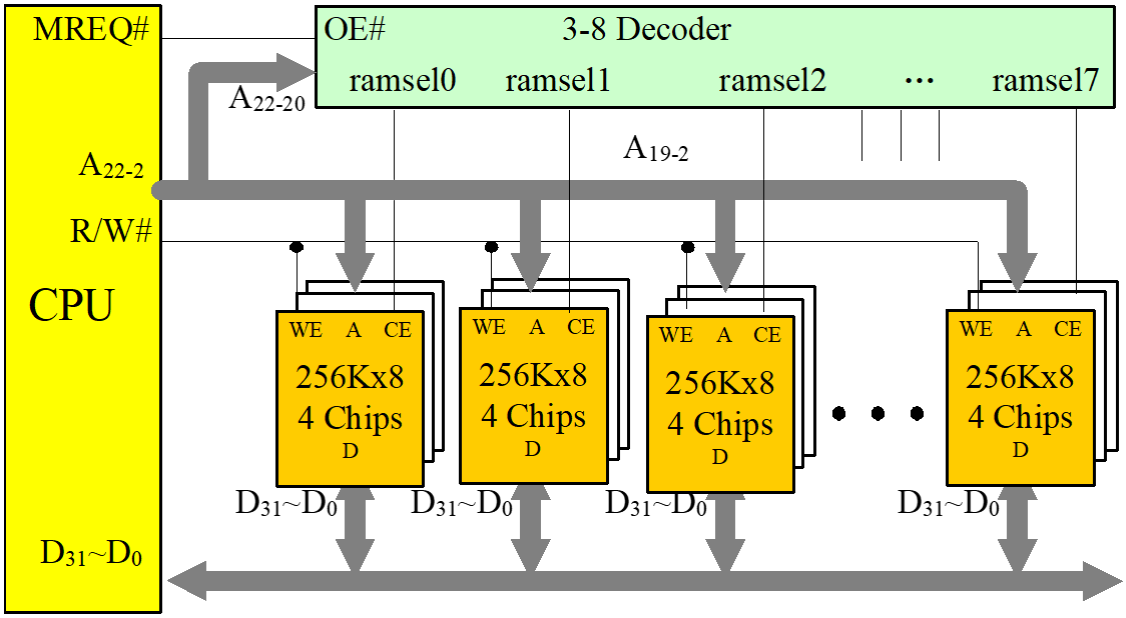
Use 3-8 decoder to select one of the chips.



Word and Bit Extension Example

Use 256Kx8-bit chips to build 2Mx32-bit memory. Addressable unit is still one byte.

A[1:0] is no use. Must access 32 bits a time (larger bandwidth)



Summary

CPU wants $N_c \times n_c$

memory has $N_m \times n_m$

- Word extension ($N_c > N_m$)
  - connect data bus of CPU and all memory chips directly.
  - require a extra decoder.
- Bit extension ($n_c > n_m$)
  - connect address bus of CPU and all memory chips directly

    May discard some address lines if $n_c >$ CPU addressable unit

- Assemble memory chips' data buses to connect with CPU.

Chap3b: IO Modules

There are a wide variety of peripherals(外设)

- Different operation logic

  Impractical for CPU to control all kinds of devices

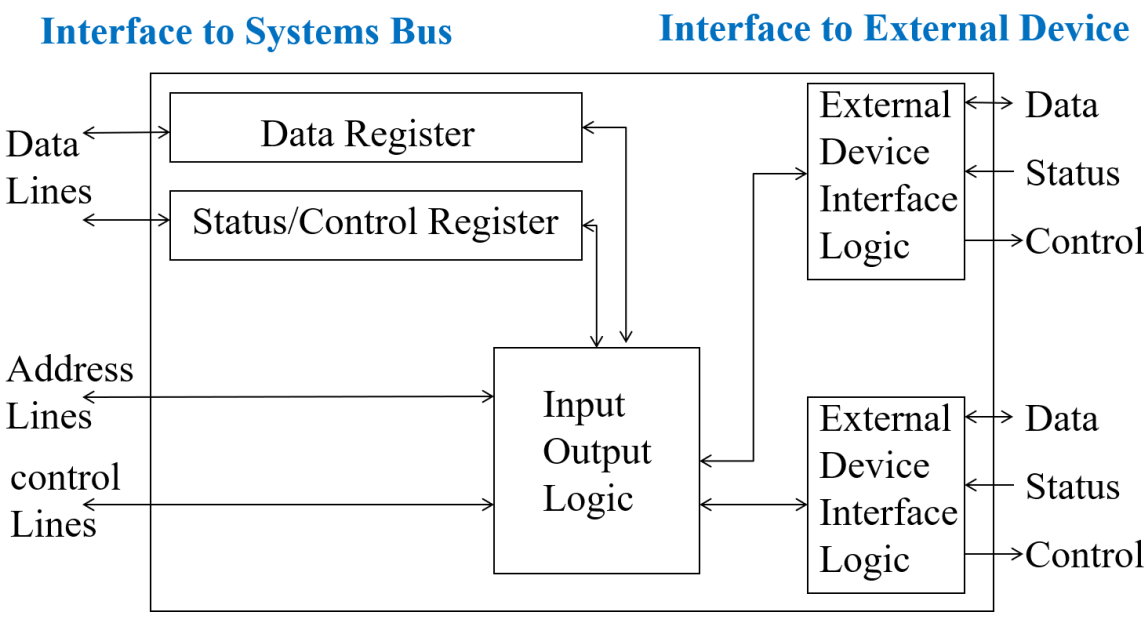
- Speak different 'languages' (serial / parallel, different speed)

  Not practical for CPU to understand

- Slower than CPU and RAM
  Not practical to directly connect devices with highspeed system bus.

IO Modules(ports)

- interface to CPU and memory
- interface to one or more peripherals



I/O Module Function

- Control & Timing
- Communication (CPU & Device)
- Data Buffering
- Error Detection

IO Steps

Transfer the data from external device to processor (CPU).

1. CPU checks I/O module for device status
2. I/O module returns the device status
3. If the device is ready, CPU requests data transfer
4. I/O module gets a unit of data from device
5. I/O module transfers the data to CPU

Variations for output, DMA, etc.

IO Techniques

Programmed IO

CPU executes a program that gives it direct control of I/O operation.

IO module does not inform or interrupt CPU.

1. CPU requests I/O operation
2. I/O module performs operation
3. I/O module sets status bits
4. CPU checks status bits periodically
5. Once the I/O module is ready, CPU transfers data (like memory access)

Simple but time consuming.

IO Commands

CPU issues address: to identify module.

CPU issues different command: control, test (check), read/write.

Interrupt Driven IO

I/O Module interrupts when ready. No repeated CPU checking of device

Issue read, do other work, check for interrupt every instruction cycle.

1. CPU requests I/O operation
2. I/O module performs operation while CPU does other work
3. I/O module informs CPU when it is ready by interrupting CPU
4. CPU deals with the event (CPU transfers data)

Interrupt: New event needs CPU to handle first but CPU needs to go back to previous work after that.

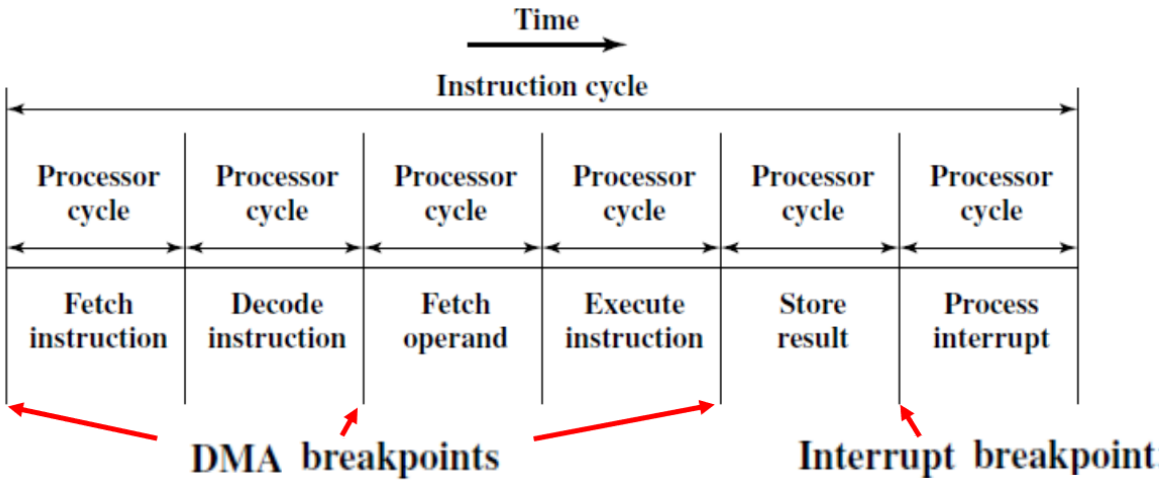
Direct Memory Access (DMA)

Additional Module (hardware) on bus. DMA controller takes over from CPU for I/O

1. CPU tell DMA controller:
    - read/write
    - device address
    - starting address of memory block
    - amount of data to be transferred
2. CPU carries on with other work
3. DMA controller deals with transfer
4. DMA controller sends interrupt when finished.

DMA Transfer Cycle Stealing: In an instruction cycle, the processor may be suspended due to DMA operation (i.e. DMA controller takes over bus while CPU wants to access it). Slow down CPU but not as much as CPU doing transfer.

DMA can have breakpoints after each stage, but interrupt breakpoint can only appear after WB stage.



DMA Configurations: refer to slides for details.

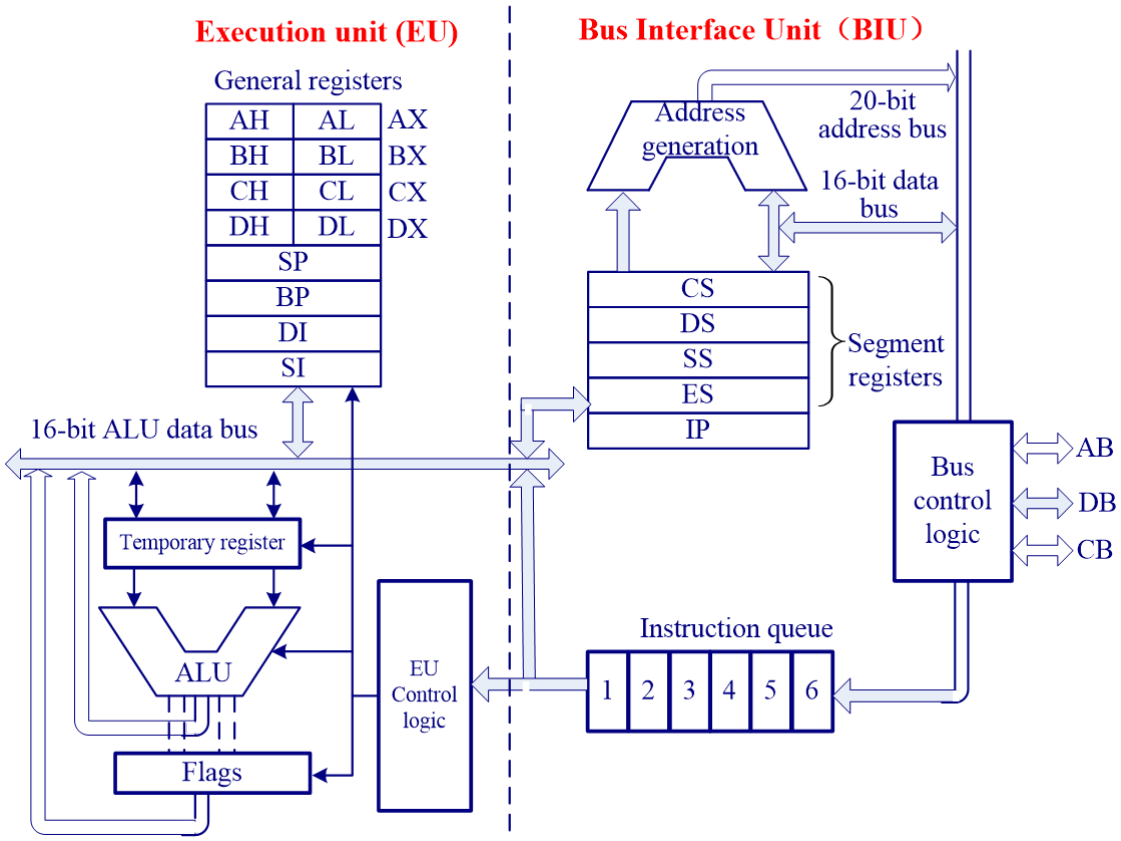
Summary

Interaction with I/O device	Programmed I/O	Interrupt driven I/O	DMA
wait for device	software (CPU instructions)	hardware (interrupt)	hardware
transfer data to memory	software	software	hardware

Chap4: 80x86 Microprocessor

Internal Organization of 8086

8086: 16-bit microprocessor, 20-bit address data bus (1MB memory), BIU + EU, pipelined(fetch&exec)



Bus interface unit (BIU)

access memory and peripherals

Two types of data: instruction&operand

consists of:

- 16-bit segment registers: CS(code), DS(data), ES(extra), SS(stack)
- 16-bit instruction pointer: IP
- 20-bit address adder (e.g. CS*16+IP)
- 6-byte instruction queue

While EU is executing an instruction, BIU will fetch the next several instructions

Execution unit (EU)

executes instructions

consists of:

- 16-bit general registers: AX(accumulator), BX(base), CX(count), DX(data)
- 16-bit pointer registers: SP(stack pointer), BP(base pointer)
- 16-bit index registers: SI(source index), DI(destination index)
- 16-bit flag register: PSW(processor status word), 9/16 bits are used
- ALU

Registers

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note:
The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

Flag Register

- Control Flags
  - **DF**(direction flag)

    indicates the direction of string operations
  - **IF**(interrupt enable flag)

    when set it enables external maskable interrupts
  - **TF**(trap flag)

    when set it allows the program to single-step for debugging
- Conditional Flags
  - **CF**(carry flag)

    set whenever there is a carry out (加法进位,减法借位). unsigned number
  - **PF**(parity flag)

    the parity of the operation result's low-order byte (i.e. AL when AX is result)

    set when the byte has an even number of 1
  - **AF**(auxiliary carry flag)

    set if there is a carry from d3 to d4, used by BCD-related arithmetic.
  - **ZF**(zero flag)

    set when the result is zero
  - **SF**(sign flag)

    copied from the sign bit (the most significant bit) after operation.
  - **OF**(overflow flag)

    set when the result of a signed number operation is too large

Note: BCD code, use 4 binary bits to represent one decimal digital

Signed Number

- Original value (原码)
- Sign-magnitude (符号+数值)
- One's complement (反码)
- Two's complement (补码)

  sign bit: MSB(most significant bit)

  CF is used for unsigned arithmetic operations

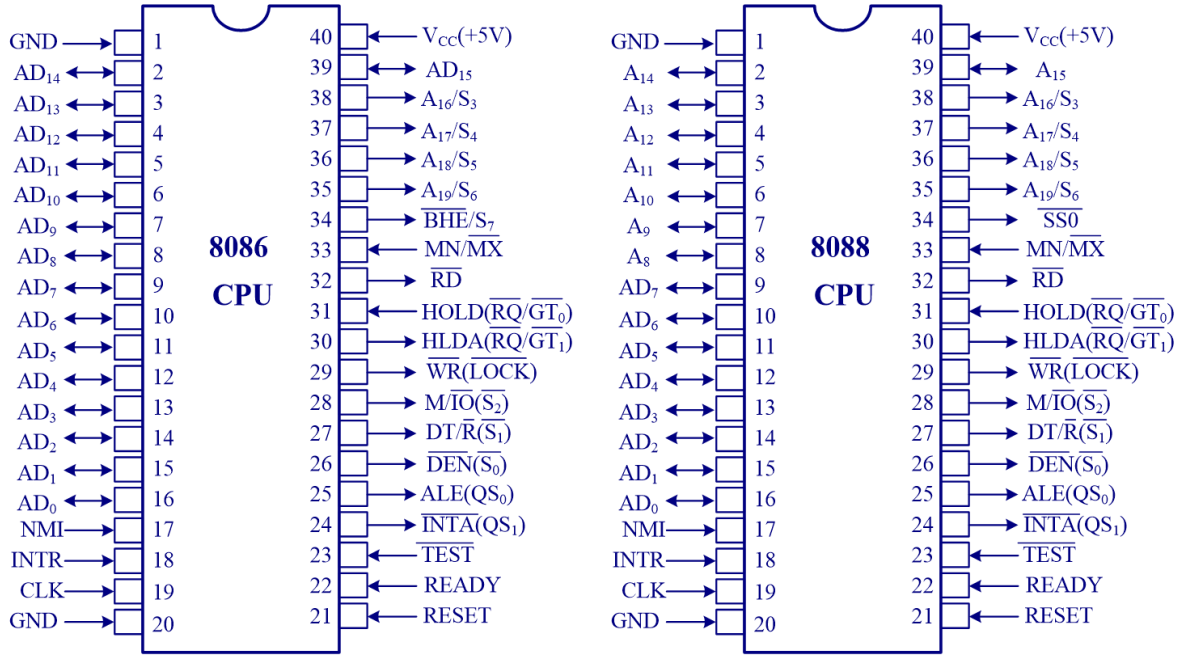
  OF is used for signed arithmetic operations

  1. positive operands, negative result; negative operands, positive result
  2. for 8-bit operation, carry from d6 to d7 and no carry from d7; carry from d7 and no carry from d6 to d7

Chip Interface of 8086

Note: A means A=1 enable, $\overline{A}$ means A=0 enable

(e.g. $MN/\overline{MX}$, 0: minimum mode; 1: maximum mode)



Work Mode

- Minimum mode: $MN/\overline{MX} = 1$

  Single CPU

- Maximum mode: $MN/\overline{MX} = 0$

  Multiple CPUs (8086 + 8087)

Control Signals

- $MN/\overline{MX}$: minimum mode (high level), maximum mode (low level)
- $\overline{RD}$: output, the CPU is reading from memory or I/O
- $\overline{WR}$: output, the CPU is writing to memory or I/O
- $M/\overline{IO}$: output, CPU is accessing memory (high level) or I/O (low level)

8086 uses isolated I/O

- $READY$: input, memory or I/O is ready for data transfer
- $\overline{DEN}$: output, used to enable data transceivers
- $DT/\overline{R}$: output, used to inform the data transceivers the direction of data transfer

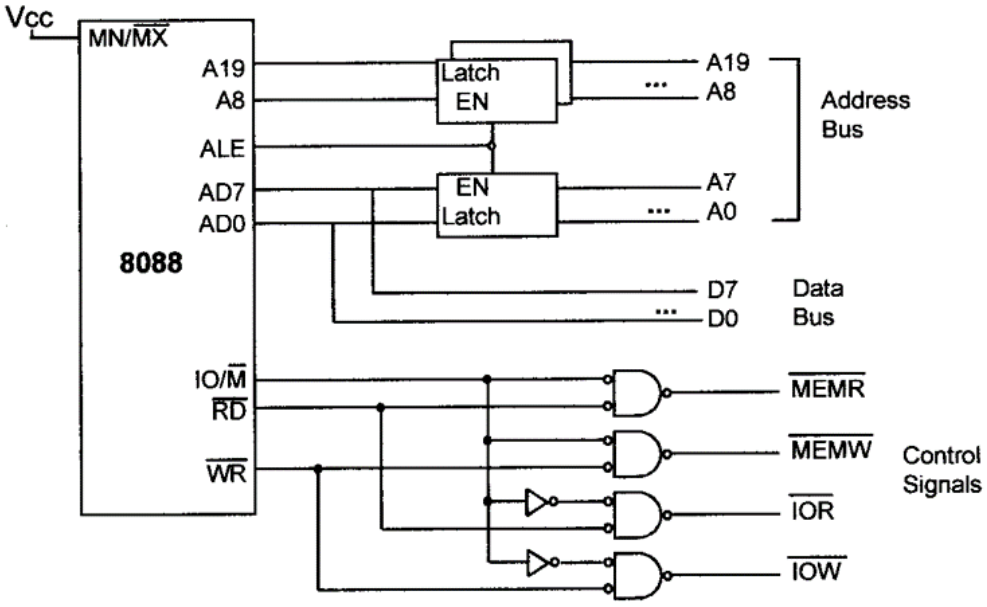
  sending: high; receiving: low
- $\overline{BHE}$: output. When $\overline{BHE} = 0$, AD_8 to AD_{15} are used. (bank high enable)
- ALE: output, used as the latch enable signal of the address latch
- $HOLD$: input signal, hold the bus request
- $HLDA$: output signal hold request ack
- $INTR$: input, interrupt request from 8259 interrupt controller, maskable by clearing the IF in the flag register
- $INTA$: output, interrupt ack
- NMI: input, non-maskable interrupt, CPU is interrupted after finishing the current instruction

  cannot be masked by software
- $RESET$: input signal, reset the CPU

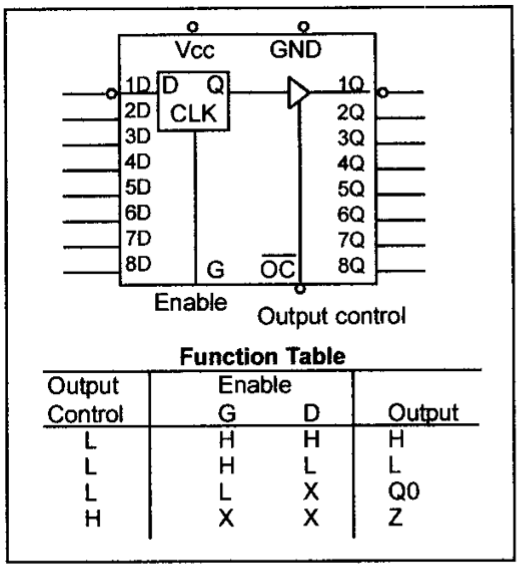
  IP, DS, SS, ES and instruction queue are cleared, CS=FFFFH

  first instruction to be execute is FFFF0H, where BIOS locates

Address/Data Demultiplexing & Address latching



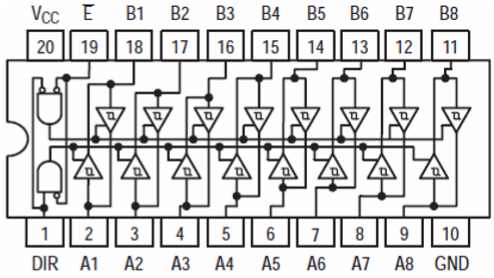
CPU first activate ALE(Address Latch Enable), then transfer the address signals, then the signal will be stored in D latch (aka address latches). After that the CPU can deactivate ALE and transfer the data signals to the data bus, and address bus can read the address from D latch.



Function Table

Output Control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

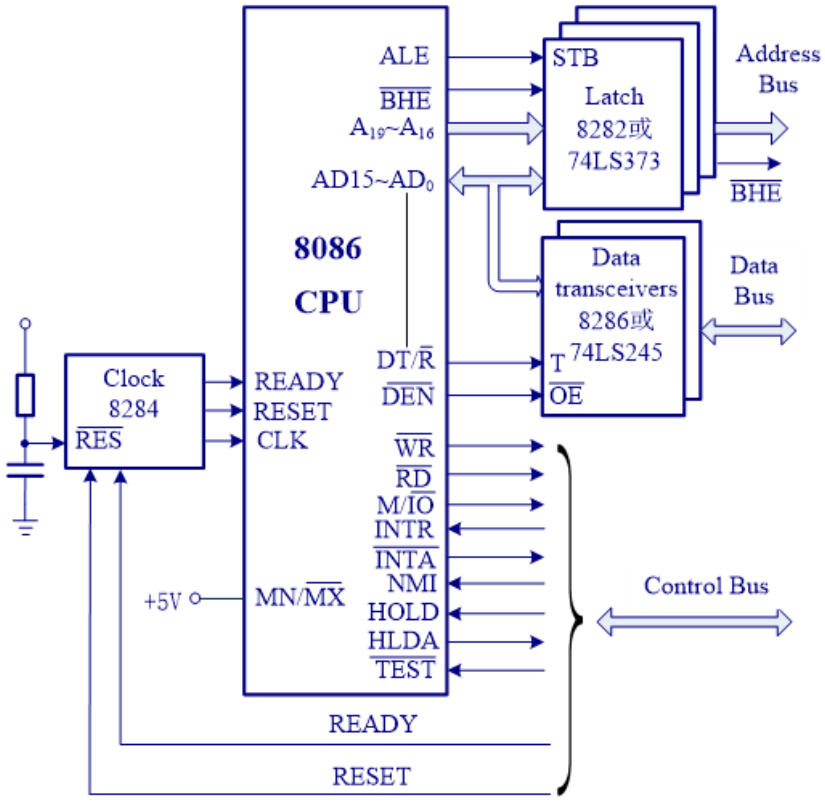
74LS373 D Latch



INPUTS		OUTPUT
Ē	DIR	
L	L	Bus B Data to Bus A
L	H	Bus A Data to Bus B
H	X	Isolation

74LS245

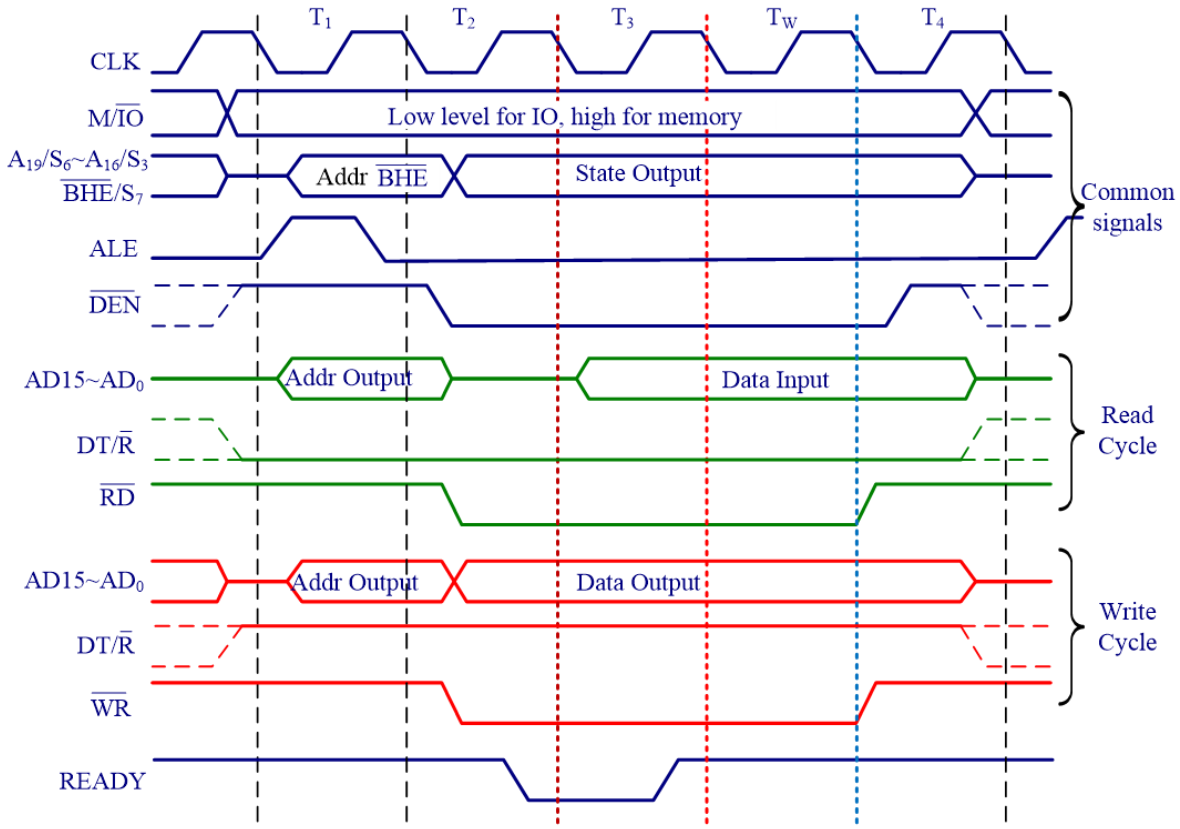
74LS373 is a D latch. 74LS245 is data transceiver, which uses tri-gates (三态门) to control the direction of data.



8086 Bus Cycle

the cycle or time required to make a single read or write transaction

latch address ⟶ issue read ⟶ execute read command ⟶ read data bus



Memory Management in 8086

A typical program on 8086 consists of at least 3 segments

- Code segment: contains instructions that accomplish certain tasks.
- Data segment: store information to be processed.
- Stack segment: store information temporarily.

Segment

A memory block includes up to 64KB. Begins on an address evenly divisible by 16.

Physical Address

- 20-bit address that is actually put on the address bus

  A range of 1MB from 00000H to FFFFFH

- Actual physical location in memory

Logical Address

- Consists of a segment value (determines the beginning of a segment) and an offset address (a relative location within a 64KB segment)

  e.g. CS (code segment register): IP (instruction pointer)

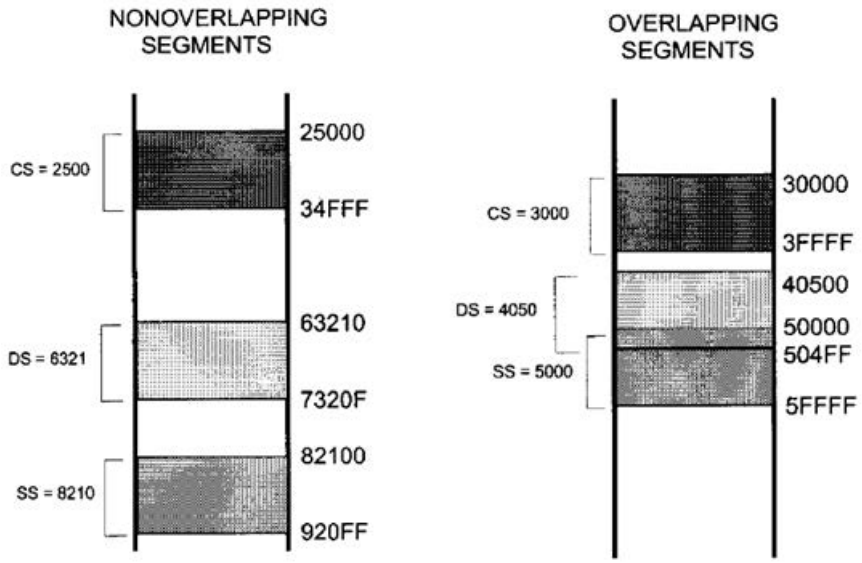
- physical address = 16 * segment value + offset address

  many logical addresses can map into one physical address

Wrap-around (反转)

When adding the offset to the shifted segment value results in an address beyond the maximum value FFFFFH, then it wraps around from 00000H.

Segment Overlapping



Code Segment

- Logical address of an instruction CS:IP
- If the instructions are physically located beyond the current code segment, then change the CS value so that those instructions can be located using new logical addresses.

Data Segment

- Logical address of a piece of data DS:offset
  - offset value
  - offset registers: BX, SI, DI
- If the data are physically located beyond the current data segment, then change the DS value so that those data can be located using new logical addresses

Data Representation in Memory

- Little endian (8086)

  the low byte of the data goes to the low memory location.

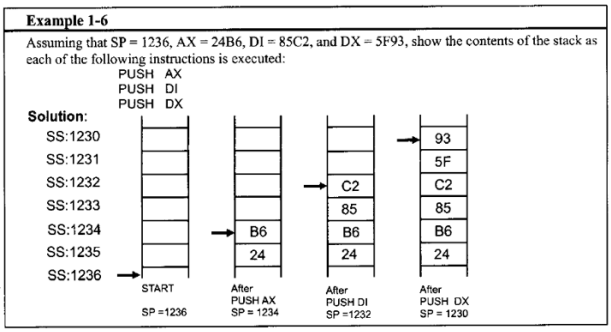
- Big endian

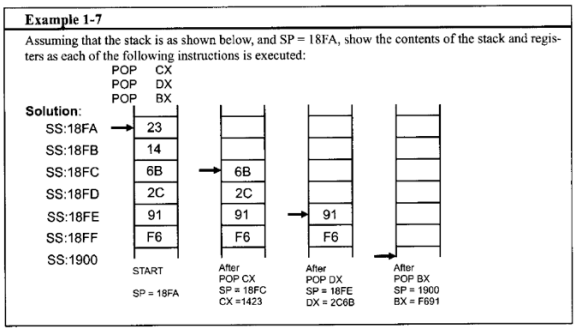
  the high byte of the data goes to the low memory location.

Stack Segment

- Logical address of a piece of data SS:SP (special applications with BP)
- Most registers (except segment registers and SP) inside the CPU can be stored in the stack and brought back into CPU from the stack using push and pop respectively
- Grows downward from upper addresses to lower addresses

example: use little endian





Extra Segment

- Logical address of a piece of data ES:offset
  - offset value
  - offset registers: BX, SI, DI
- essential for string operations

Segment Address Summary

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

BIOS Function

The first instruction to be execute after reset is FFFF0H, where BIOS locates.

- Tests all devices connected to the PC when powered on and reports errors if any
- Load DOS from disk into RAM
- Hand over control of the PC to DOS

Addressing mode in 8086

Default Segment Registers

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

- **Register**

  ```
  MOV BX,AX
  ```

  Data can be moved among ALL registers except CS (can not be set) and IP (cannot be accessed by MOV)

- **Immediate**

  ```
  MOV AX,500H
  ```

  Immediate numbers CANNOT be moved to segment registers

- **Direct**

  Data is stored in memory and the address is given in instructions

  Segment address in the data segment (DS) by default

  ```
  MOV DL,[2400]   ;move DS:2400H into DL
  ```

- **Register indirect**

  Data is stored in memory and the address is held by a register

  Segment address in the data segment (DS) by default

  Registers for this purpose are SI, DI, and BX

```
MOV AL,[BX]     ;move DS:BX into AL
```

- **Based relative**

  Data is stored in memory and the address can be calculated with base registers BX and BP as well as a displacement value

  The default segment is data segment (DS) for BX, stack segment (SS) for BP

```
MOV CX,[BX]+10  ;move DS:BX+10 into CX
```

- **Indexed relative**

  Data is stored in memory and the address can be calculated with index registers DI and SI as well as a displacement value

  The default segment is data segment (DS)

```
MOV DX,[SI]+5
```

- **Based indexed relative**

  Combines based and indexed addressing modes, one base register and one index register are used

  The default segment is data segment (DS) for BX, stack segment (SS) for BP

```
MOV CL,[BX][DI]+8   ;PA=DS*16+BX+DI+8
```

Segment Overrides

allows the program to override the default segment registers

Instruction	Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Chap5: Assembly Language Programming (1)

Assembly Programming Language

Statements

- Assembly language instructions: perform the real work of program (for CPU).
- Directives (pseudo-instructions): Give instructions for the assembler program about how to translate the program into machine code.

Consists of multiple segments

Statement

Note: immediate CAN NOT begin with letter, otherwise it might be a label (e.g. FFH ⟶ 0FFH)

[label:] mnemonic [operands] [;comment]

- label is a reference to this statement

  Rules for names: each label must be unique; letters, 0-9, (?), (.), (@), (_), and ($); first character cannot be a digit

- ":" is needed if it is an instruction otherwise omitted
- ";" leads a comment, the assembler omits anything on this line following a semicolon

Model Definition

size	
SMALL	code <=64KB, data <=64KB
MEDIUM	code >64KB, data <=64KB
COMPACT	code <=64KB, data >64KB
LARGE	data>64KB but single set of data<64KB, code>64KB
HUGE	data>64KB, code>64KB
TINY	code + data<64KB

Simplified Segment Definition

- Only three segments can be defined: `.CODE`, `.DATA`, `.STACK`
- Automatically correspond to the CPU's `CS`, `DS`, `SS`
- DOS determines the `CS` and `SS` segment registers automatically

  `DS` (and `ES`) has to be manually specified

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DB      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                ADD     AL,BL           ;add the operands
                MOV     SUM,AL          ;store the result in location SUM
                MOV     AH,4CH          ;set up to return to DOS
                INT     21H             ;
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Code segment

- Write your statements
- Procedures definition

```
label PROC [FAR|NEAR]
...
    ret
label ENDP
```

NEAR means the procedure is in the same code segment; FAR means the procedure is in another code segment.

Entrance proc should be FAR

Full Segment Definition

```
label SEGMENT
...
label ENDS
```

- You name those labels, and segments can be as many as needed.
- DOS assigns CS and SS ; program assigns (manually) DS and ES.

Program Execution

- Program starts from the entrance
- Program ends whenever calls 21H interrupt with AH=4CH

Control Transfer Instructions

- `JUMP` instruction

  SHORT, NEAR: intra-segment; FAR: inter-segment

  - conditional jump

    SHORT jumps. ABOVE&BELOW: unsigned, GREAT&LESS: signed

Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

  - unconditional jump

    NEAR by default

    `JMP [SHORT|NEAR|FAR PTR] label`

- `CALL` instruction

NEAR: same code segment; FAR: outside current code segment

- Calling a NEAR procedure

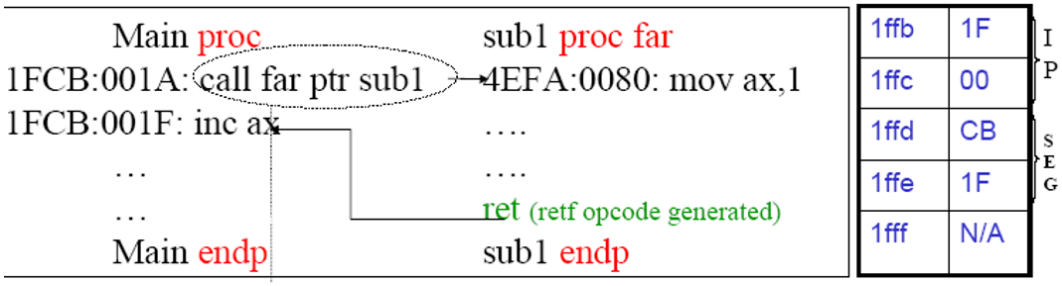
  save current IP on the stack

  load subroutine's offset into IP

- Calling a FAR procedure

  save current CS and IP on the stack

  load subroutine's CS and offset into IP



Data Type & Definition

CPU can process either 8-bit or 16-bit. 32-bit data needs more words.

Directives

- `ORG`

  indicates the beginning of the offset address (可用于对齐)

```
...     ;can be anything, may overlap
ORG 100H
...     ;begin offset is 100H
```

- Define variables

```
x DB 12
```

  DB: byte; DW: word, 2 bytes; DD: double word, 4 bytes; DQ quadword, 8 bytes

- `EQU`

```
NUM     EQU 234 ;define a constant
```

- `DUP`

  duplicate a given number of characters

```
x   DB 6 DUP(23H)       ;array memory:  23  23  23  23  23  23
```

More about Variables

Variable names have three attributes

- Segment value
- Offset address
- Type (e.g. byte-wise)

```
;get the segment value
MOV     AX, SEG xxxx        ;1
MOV     AX, @DATA           ;2  if x is first variable in DATA SEG
;get the offset address
MOV     AX, OFFSET  xxxx    ;1
LEA     AX, xxxxx           ;2
```

More about Labels

Labels have three attributes

- Segment value
- Offset address
- Type: range for jumps: NEAR, FAR

Label Definition

- Implicitly

  `AGAIN:  ADD AX,1`

- Use `LABEL`

  ```
  AGAIN LABEL FAR
  ADD AX, 03423H
  ```

PTR Directive

Temporarily change the type (range) attribute of a variable (label).

- To guarantee that both operands in an instruction match

  ```
  DATA1   DB 10H,20H,30H
  DATA2   DW 4023H, 0A845H
  ...
  MOV     BX, WORD PTR DATA 1 ; 2010H -> BX
  MOV     AL, BYTE PTR DATA 2 ; 23H -> AL
  MOV     WORD PTR [BX], 10H; [BX], [BX+1] <- 0010H
  ```

- To guarantee that the jump can reach a label

  ```
  JMP FAR PTR aLabel
  ```

Chap6: Assembly Language Programming (2)

Arithmetic Instructions

CPU will treat all the values as unsigned value

Addition

```
ADD     dest,src    ;dest=dest+src
```

- dest can be a register or in memory
- src can be a register, in memory or immediate
- No mem-to-mem operations in 8086
- Change ZF, SF, AF, CF, OF, PF

```
ADC     dest,src    ;dest=dest+src+CF
```

- For multi-byte numbers
- If there is a carry from last addition, adds 1 to the result
- Change ZF, SF, AF, CF, OF, PF

```
INC     dest        ;dest=dest+1
```

- dest can be a register or in memory
- dest cannot be an immediate
- Change ZF, SF, AF, OF, PF
- DOES NOT change CF

Subtraction

```
SUB     dest,src    ;dest=dest-src
```

- dest can be a register or in memory
- src can be a register, in memory or an immediate
- No mem-to-mem operations in 8086
- Change ZF, SF, AF, CF, OF, PF

```
SBB     dest,src    ;dest=dest-src-CF   CF表示借位
```

- For multi-byte numbers
- If there is a borrow from last subtraction, subtracts 1 from the result
- Change ZF, SF, AF, CF, OF, PF

```
DEC     dest        ;dest=dest-1
```

- dest can be a register or in memory
- dest cannot be an immediate
- Change ZF, SF, AF, OF, PF
- DOES NOT change CF

CF in subtraction

CF indicates borrow

1. take the 2's complement of the src
2. add it to the dest
3. invert the carry

Note: CF=0, positive result; CF=1, negative result

Multiplication

```
MUL operand
```

- operand can be a register, in memory (not an immediate)
- Change OF, CF; Unpredictable: SF, ZF, AF, PF
- byte * byte: one implicit operand is AL, the other is operand, result is stored in AX
- word * word: one implicit operand is AX, the other is operand, result is stored in DX(high)&AX(low)
- word * byte: AL hold the byte and AH=0 (case 2), the word is the operand, result is stored in DX(high)&AX(low)

Division

```
DIV denominator
```

- denominator can be a register or in memory
- Denominator cannot be zero. Quotient cannot be too large for the assigned register
- word / byte: numerator in AX; quotient in AL (max 0FFH), remainder in AH
- double-word / word: numerator in DX&AX, quotient in AX (max 0FFFFH), remainder in DX
- byte / byte: numerator in AL, clear AH (case 1); quotient in AL, remainder in AH
- word / word: numerator in AX, clear DX (case 2); quotient in AX, remainder in DX

Logical Instructions

Bit-wise Operations

```
AND     dest,src
OR      dest,src
XOR     dest,src
```

- dest can be a register or in memory
- src can be a register, in memory, or immediate
- Update SF, ZF, PF; AF is undetermined
- Clear CF and OF (set to 0)

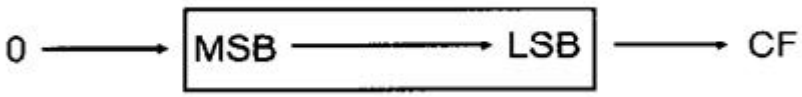
```
NOT     operand
```

- operand can be a register or in memory
- DOES NOT change the flag register

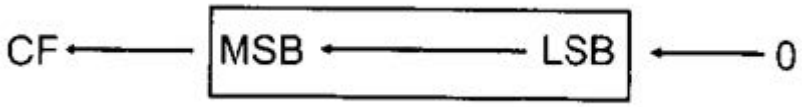
Logical SHIFT

dest can be a register or in memory

```
SHR     dest, times
```




```
SHL     dest, times
```

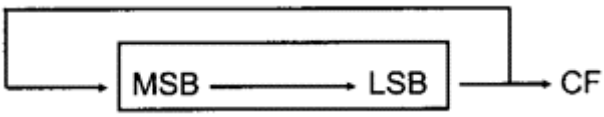



```
;times=1
SHR     xxx, 1
;times>1
MOV     CL, times
SHR     xx, CL      ;can not use imm
;same in rotate shift
```

Rotate SHIFT

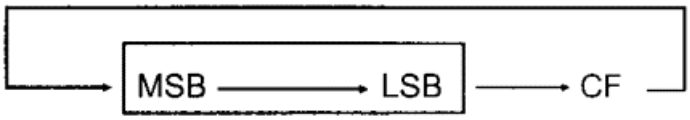
dest can be a register, or in memory

```
ROL     dest,times
ROR     dest,times
```




```
RCL     dest,times
RCR     dest,times
```

Note: CF is involved.



Compare Instruction

```
CMP     a,b
;   unsigned
;   case    CF  ZF
;   a>b     0   0
;   a=b     0   1
;   a<b     1   0
;   signed
;   case
;   a>b     OF=SF & ZF=0
;   a=b     ZF=1
;   a<b     OF!=SF
```

Flags affected as `a-b` but operands remain unchanged

- Jump based on unsigned

  Above and Below (e.g. JA, JAE, JB, JBE)

- Jump based on signed

  Great and Less (e.g. JG, JGE, JL)

Chap7: Memory Address Decoding

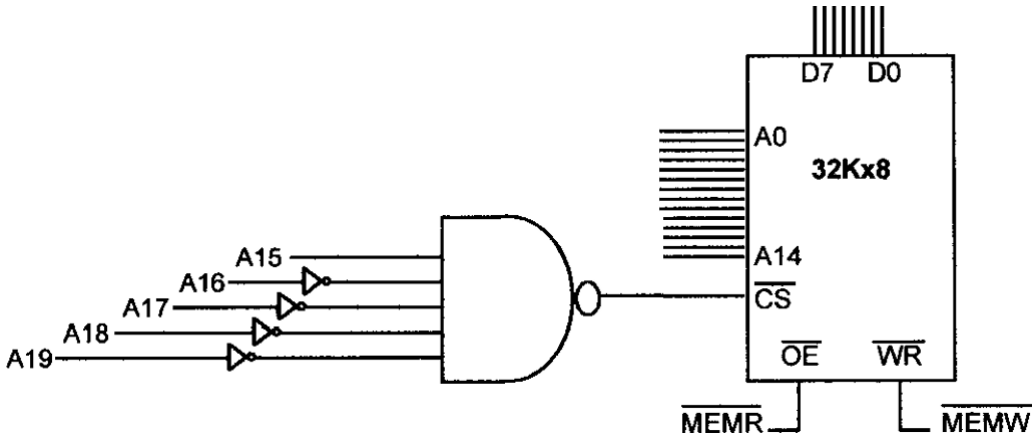
CPU calculates the physical address of the operand and put corresponding signals on the address bus.

Memory address decoding circuitry locates the specific memory chip that stores the desired data.

Examples

Logic gates

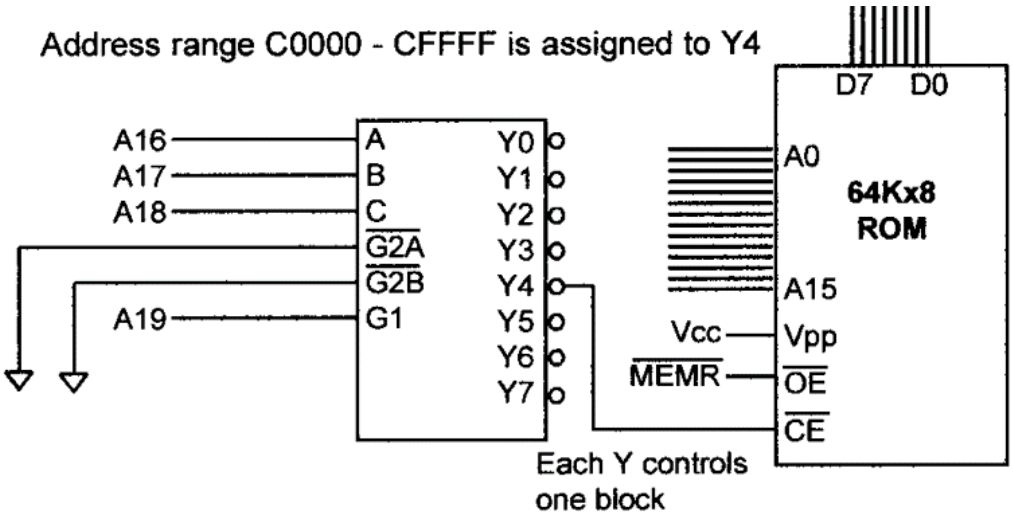
locate 09012H: 0000 1001 0000 0001 0010



This memory chip contains: 08000H~0FFFFH (0000 1.....)

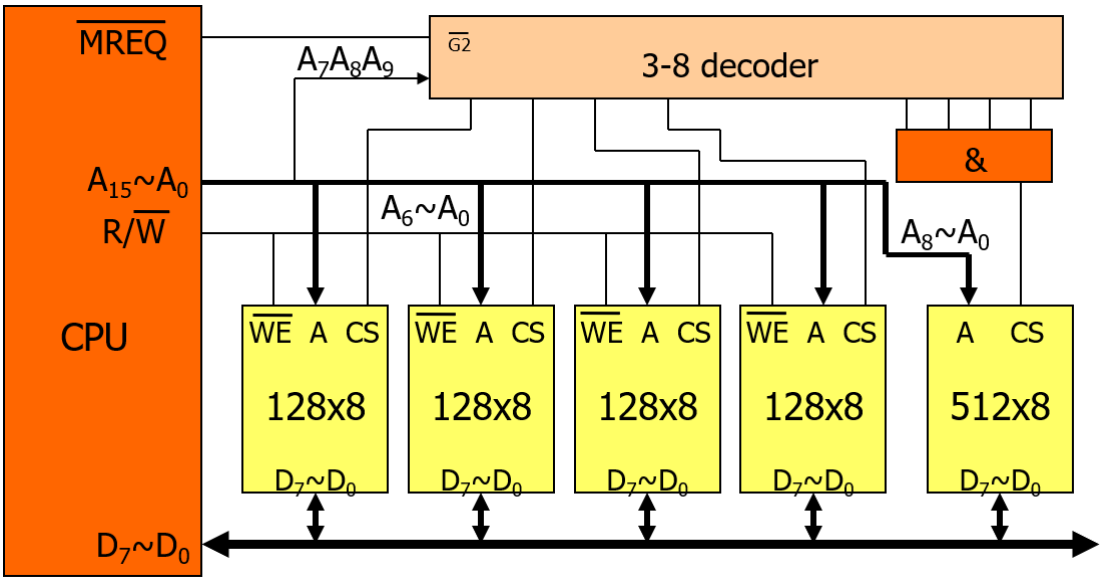
74LS138 decoder chip

location: 1100 ....



Build Memory

512 bytes RAM and 512 bytes ROM. Use 128x8 chips for RAM and 512x8 chips for ROM



More on Decoding

- Absolute address decoding (全译码)

  All address lines are decoded

- Linear select decoding (部分译码)

  Only selected lines are decoded, cheap, but with aliases (same memory unit (I/O port) with multiple addresses)

Data Integrity

- Checksum byte for ROM

  Check the integrity of a series of bytes

  1. Add all bytes together and drop all carries
  2. Take the two's complement of the sum (negation)

  check the integrity by adding data and checksum, the result is 0

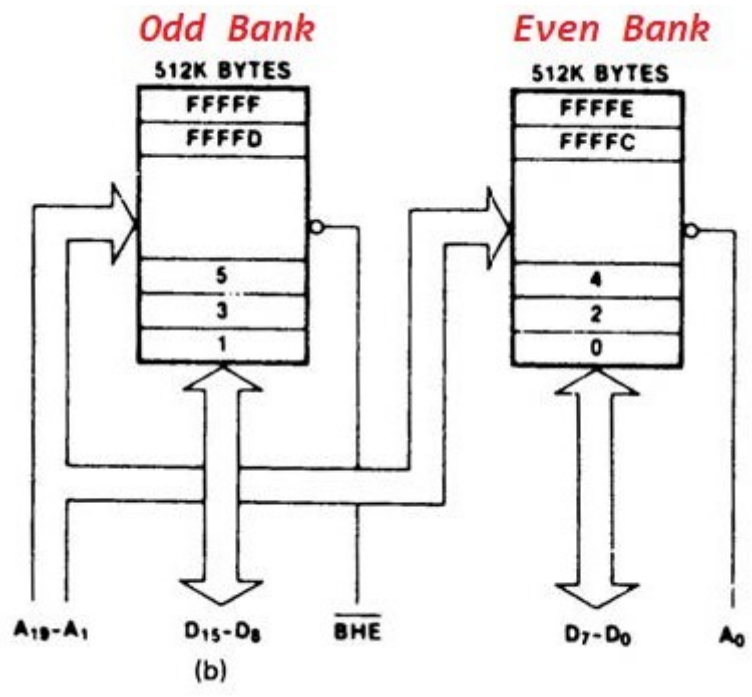
- Parity bit (奇偶校验码) for DRAM

Check the integrity of a series of bits

odd parity: if data have even number of 1s, the parity bit is set to 1. (in 8086)

- CRC for disks and the Internet

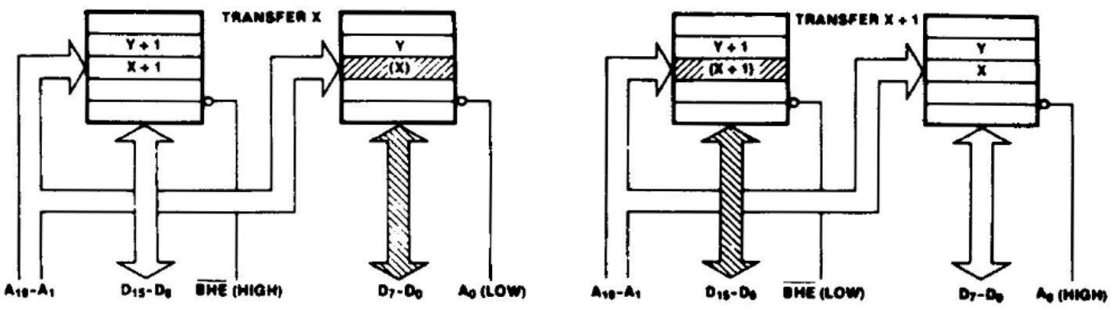
Memory Organization in 8086



Address bits `A[19:1]` select the storage location that is to be accessed. They are applied to both banks in parallel. `A[0]` and `bank high enable` ($\overline{BHE}$) are used as bank-select signals.

BHE	A0		
0	0	Even word	D0 - D15
0	1	Odd byte	D8 - D15
1	0	Even byte	D0 - D7
1	1	None	

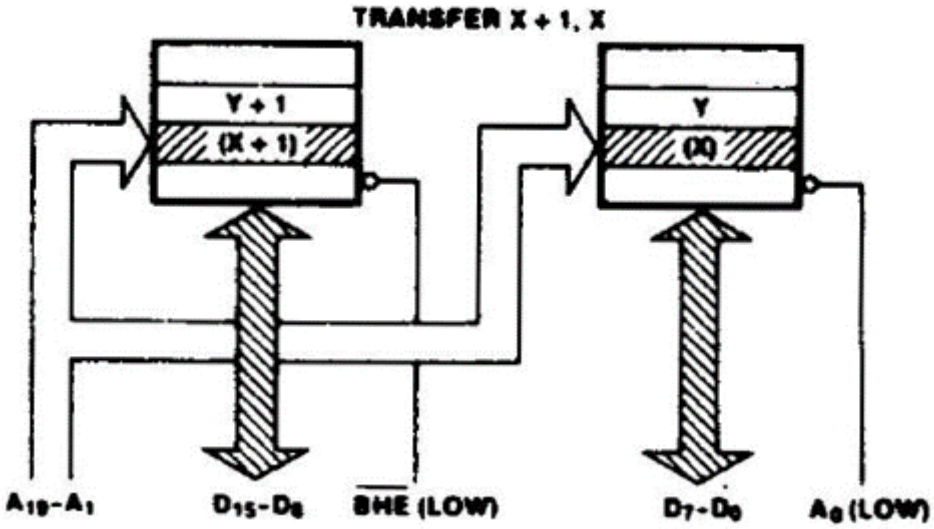
- Byte Memory Operation



Even: only D0~D7 have data; Odd: only D8~D15 have data

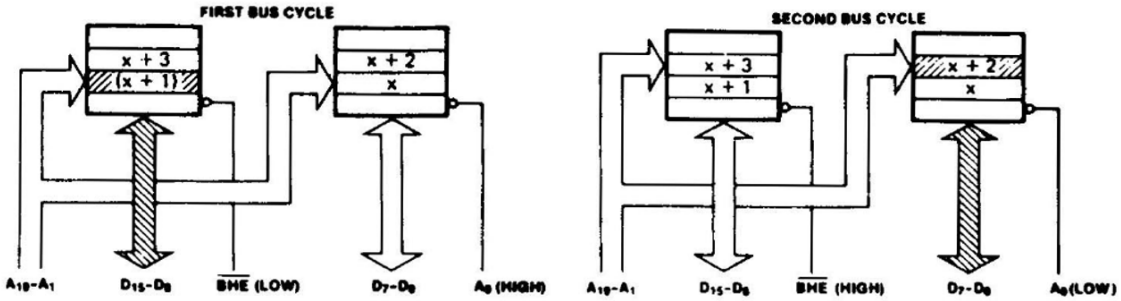
- Aligned Word

  Need 1 bus cycle.



- Misaligned Word

  Need 2 bus cycle.



I/O in X86 Family

- x86 microprocessors have an I/O space in addition to memory space (i.e. isolated I/O)
- Use special I/O instructions accessing I/O devices at ports(i.e. addresses for I/O)
- Memory can contain machine codes and data, I/O ports only contain data.

I/O Instructions

Note: no segment concept for port addresses

- Direct

  port# ranges from 00h to 0ffh, 256 ports in total

```
IN      AL,port
OUT     port,AL
;can only use AL,AX
```

- Indirect

  port# ranges from 0000h to 0ffffh, 65536 ports in all

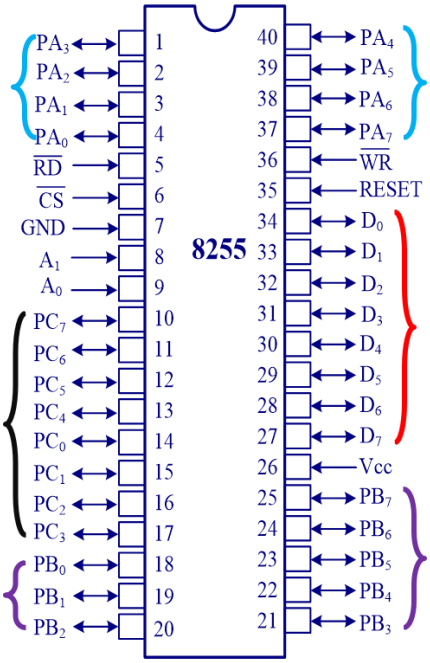
  use a 16-bit address that resides in the DX register

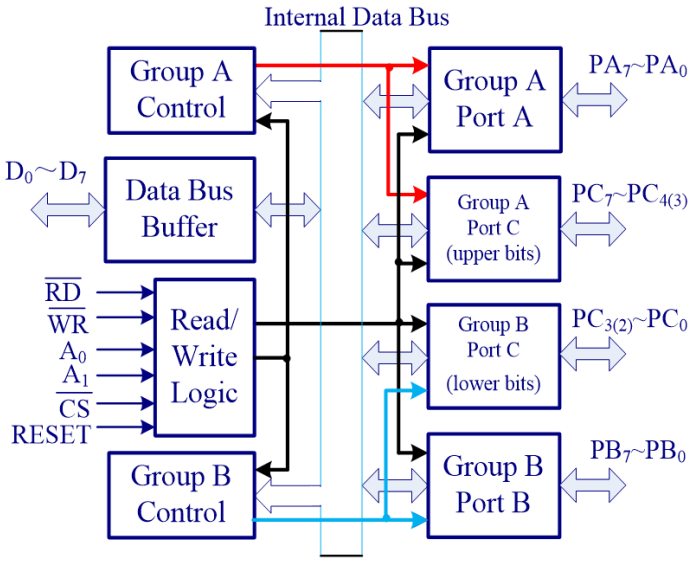
```
MOV     DX,port
IN      AL,DX
OUT     DX,AL
```

Chap8: 8255 PPI Chip

外设管理芯片

PPI: Programmable Parallel Interface (I/O module)





Internal Structure and Pins

Data Ports

- Port A&B

  can be programmed all as input/output

- Port C

  can be split into two separate parts PCU and PCL; any bit can be programmed individually.

Control Registers (CR)

A 8-bit internal register, used to setup the chip, selected when A[0]=1&A[1]=1

Groups

- Group A: PA & PCU(upper)
- Group B: PB & PCL(lower)

Read/Write Control Logic

~CS	A_1	A_0	~RD	~WR	Function
0	0	0	0	1	PA->Data bus
0	0	1	0	1	PB->Data bus
0	1	0	0	1	PC->Data bus
0	0	0	1	0	Data bus->PA
0	0	1	1	0	Data bus->PB
0	1	0	1	0	Data bus->PC
0	1	1	1	0	Data bus->CR
1	×	×	1	1	D_0~D_7 in float

- Internal (CR) and external control signals
- RESET: high-active, clear control register, all ports are set as input
- A[1], A[0]: port selection

Operation Modes

- **Input/Output (IO) modes**

  can only send data in the unit of byte

  - Mode 0: simple I/O mode

    PA, PB, PC: PCU{PC4 ~ PC7}, PCL{PC0 ~ PC3}.

    No handshaking (negotiation between two entitles before communication)

    Each port can be programmed as input/output port

  - Mode 1

    PA, PB can be used as input/output ports with handshaking

    PCU{PC3 ~ PC7}, PCL{PC0 ~ PC2} are used as handshake lines for PA and PB respectively

  - Mode 2

    Only **PA** can be used for bidirectional handshake data transfer

    PCU{PC3 ~ PC7} are used as handshake lines for PA
- **Bit Set/Reset (BSR) mode**

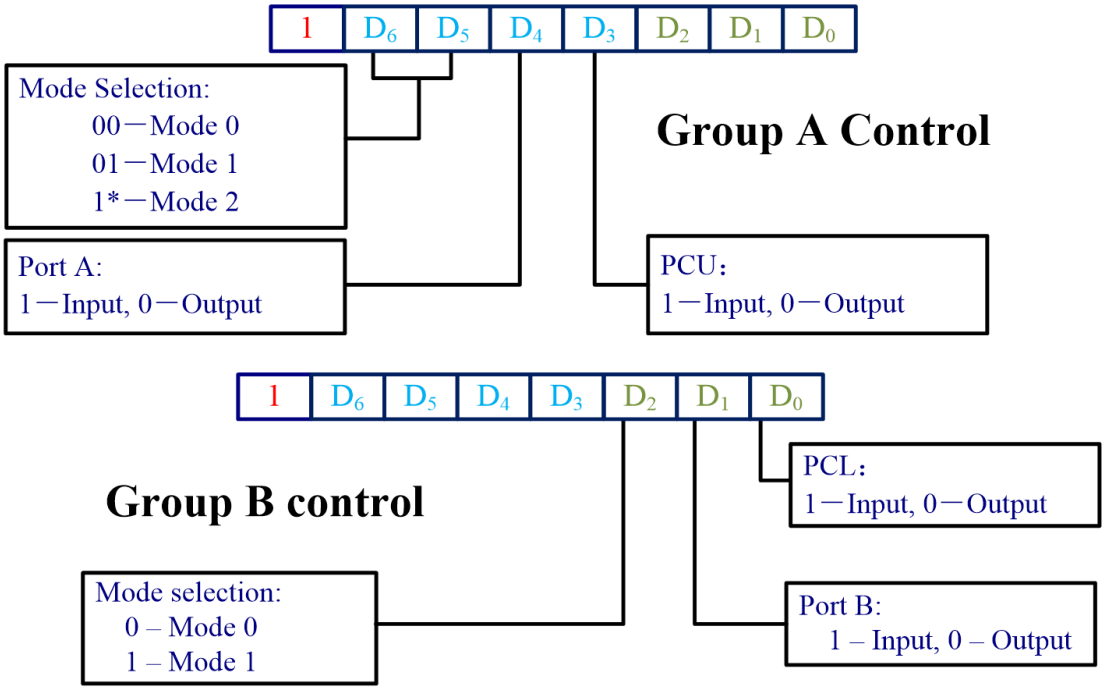
  Only **PC** can be used as output port

  Each line of PC can be set/reset individually

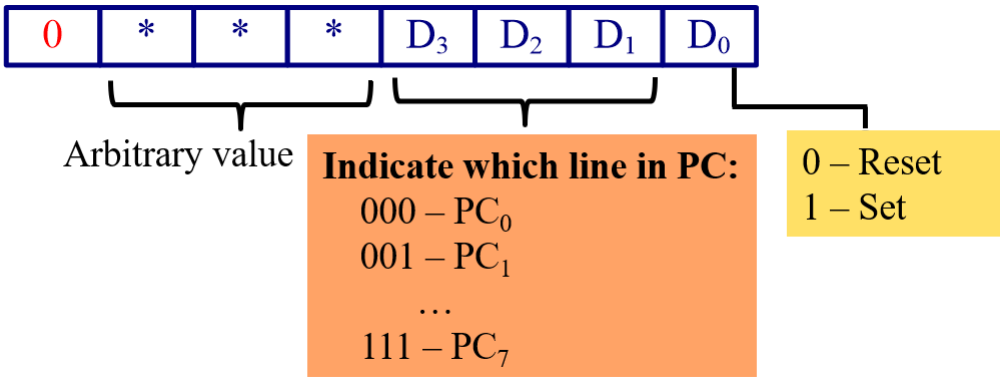
Control Register & Operation Modes

CR contains control word.

- IO Modes



- BSR Mode



Example

```
;BSR mode e.g.
;generate a positive pulse
MOV     AL,00001011b
OUT     CtrPort,AL
MOV     AL,00001010b
OUT     CtrPort,AL
```

IO Modes

Mode 0 (Main)

for simple input/output scenario

Any port of PA, PB and PC can be programmed as input or output port independently

CPU directly read from or write to a port using IN and OUT instructions

Input data are not latched, but output data are latched

Example

(1) send control word to CR first. (2) execute IO operations

The 8255 shown in Figure 11-13 is configured as follows: port A as input, B as output, and all the bits of port C as output.
(a) Find the port addresses assigned to A, B, C, and the control register.
(b) Find the control byte (word) for this configuration.
(c) Program the ports to input data from port A and send it to both ports B and C.

Solution:
(a) The port addresses are as follows:

$\overline{CS}$	A1	A0	Address	Port
11 0001 00	0	0	310H	Port A
11 0001 00	0	1	311H	Port B
11 0001 00	1	0	312H	Port C
11 0001 00	1	1	313H	Control register

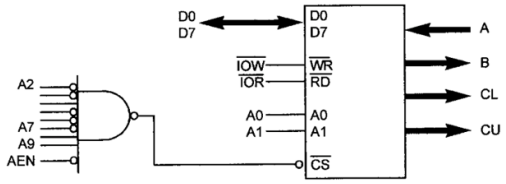


Figure 11-13

(b) The control word is 90H, or 1001 0000.

(c) One version of the program is as follows:

```
MOV   AL,90H      ;control byte PA=in, PB=out, PC=out
MOV   DX,313H     ;load control reg address
OUT   DX,AL       ;send it to control register
MOV   DX,310H     ;load PA address
IN    AL,DX       ;get the data from PA
MOV   DX,311H     ;load PB address
OUT   DX,AL       ;send it to PB
MOV   DX,312H     ;load PC address
OUT   DX,AL       ;and to PC
```

Mode 1

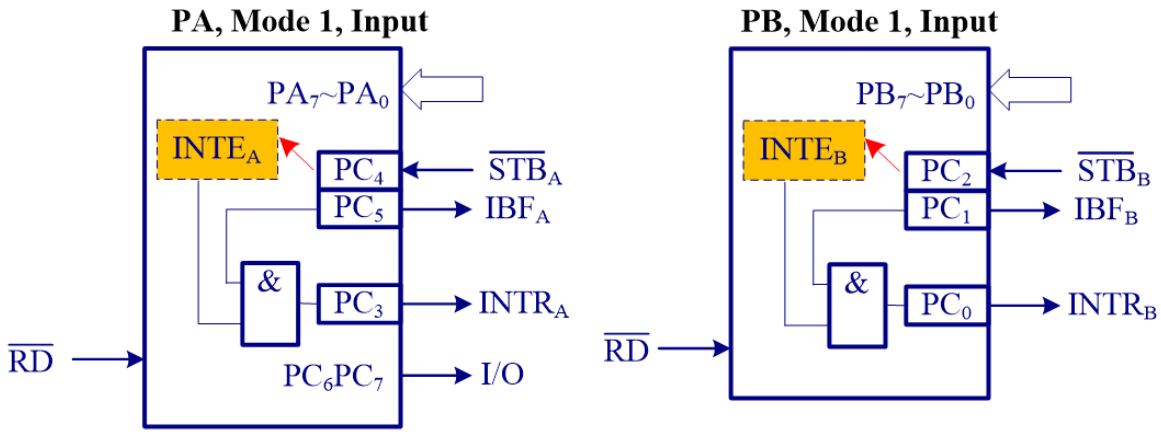
for handshake input/output scenario

Both input data and output data are latched.

- PA and PB can be used as input or output ports
- PC3~PC5 used as handshake lines for PA
- PC0~PC2 used as handshake lines for PB

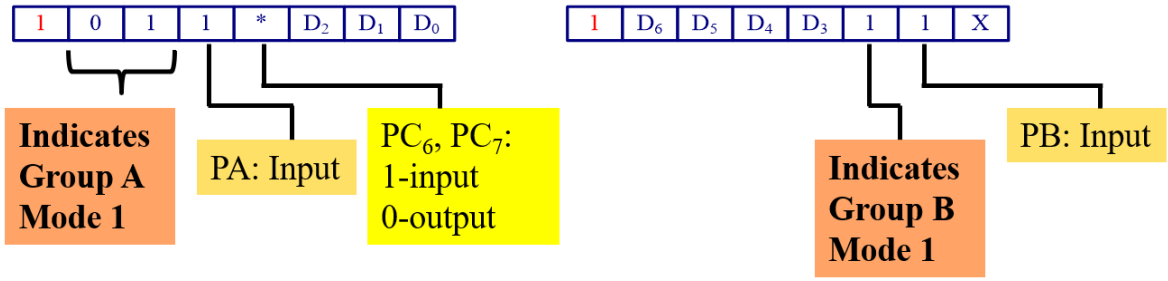
Input

PC6&PC7 can be used as separate I/O lines for any purpose (controlled by D3)

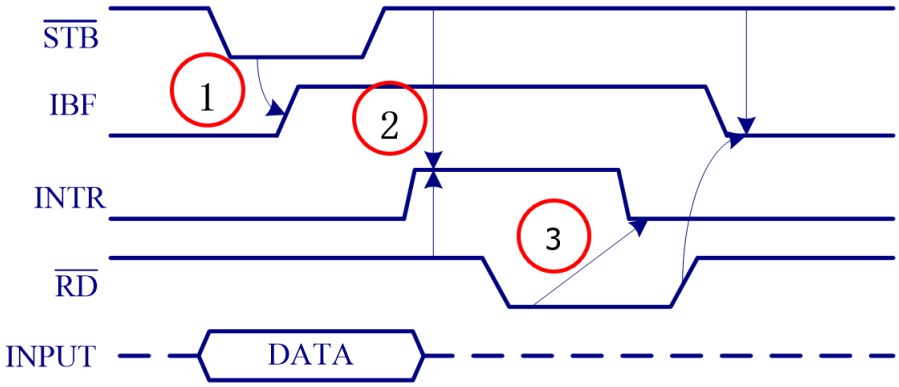


- $\overline{STB}$ (input): input from device, device loads data into the port latch (low - load) (选通)
- IBF (output): input buffer full (high - full), output to device
- $INTR$ (output): interrupt request is an output to CPU that requests an interrupts (Interrupted IO)
- $INTE$ (internal): interrupt enable signal.

  programmed via the PC4 (port A) or PC2 (port B) with BSR mode before configuration (1 - allowed)



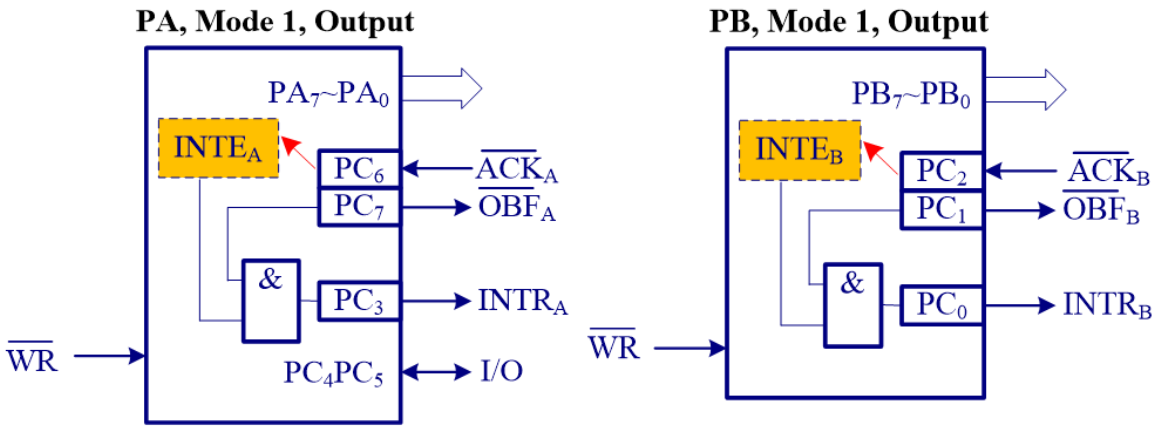
Input Timing



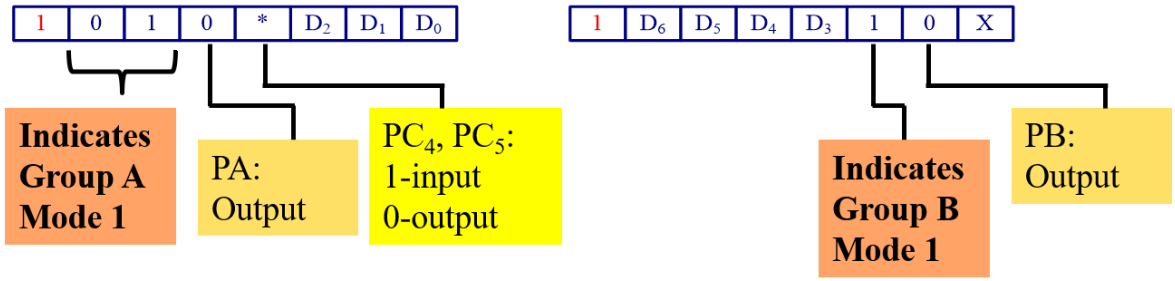
1. device inputs data until buffer is full
2. 8255 inform CPU by interrupt
3. CPU responds to interrupt, clear INTR and read data
4. clear IBF after CPU finishes reading

Output

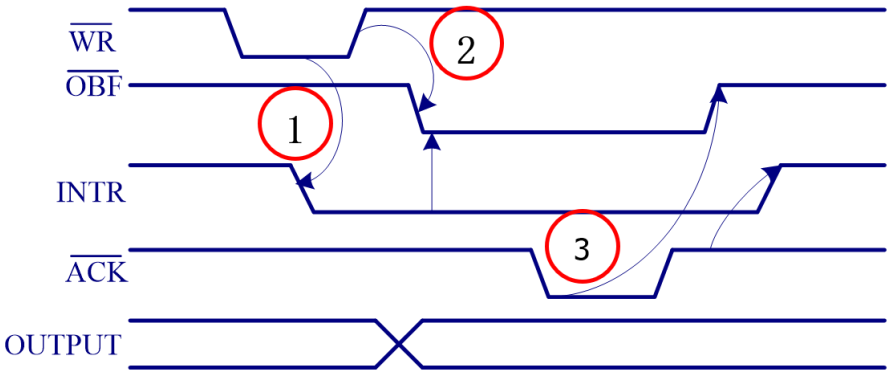
PC4&PC5 can be used as separate I/O lines for any purpose (controlled by D3)



- $\overline{ACK}$ (input): acknowledge, indicates that the external device has taken the data (low - taken)
- $\overline{OBF}$ (output): output buffer full, indicates the data has been latched in the port (low - full)
- $INTR$: Interrupt request is an output to CPU that requests an interrupts
- $INTE$: programmed via the PC6 (port A) or PC2 (port B) with BSR mode before configuration



Output Timing



1. CPU responds interrupt and writes data to buffer until buffer is full
2. 8255 activates $\overline{OBF}$ to inform device
3. device takes data, activates $\overline{ACK}$ and clear $\overline{OBF}$
4. 8255 set INTR to inform CPU to write data

Mode 2

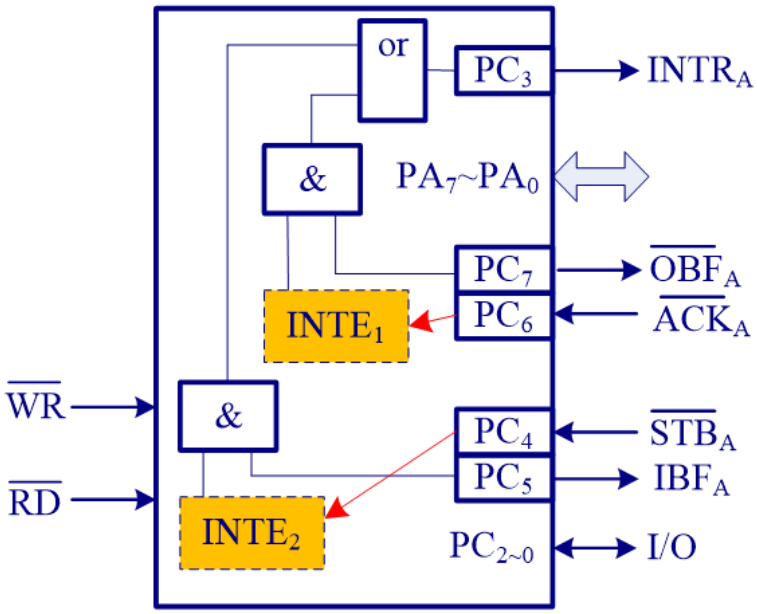
Mode 1 Input + Mode 1 Output

for bidirectional handshake input/output scenario

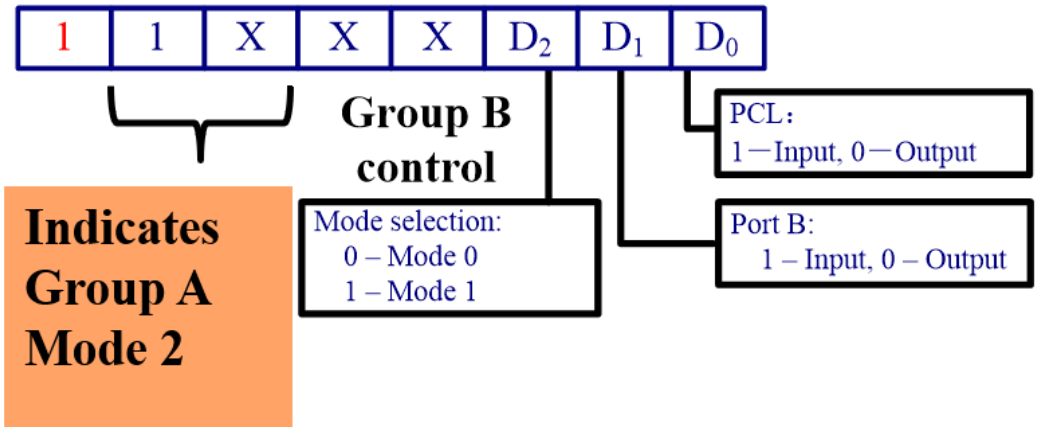
Both input data and output data are latched

Only can PA be used as both input and output port.

PC3~PC7, used as handshake lines for PA



When CPU responds to an interrupt, the interrupt handler must check the $\overline{OBF}$ and IBF in order to tell whether the input process or the output process is generating the interrupt.

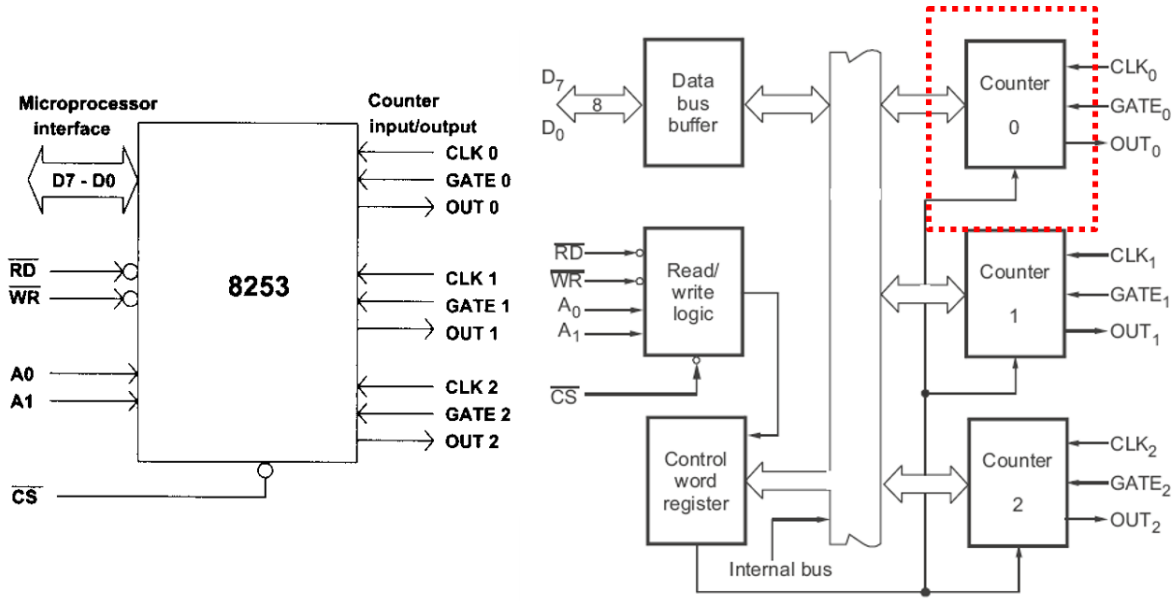


Chap9: 8253 PIT Chip

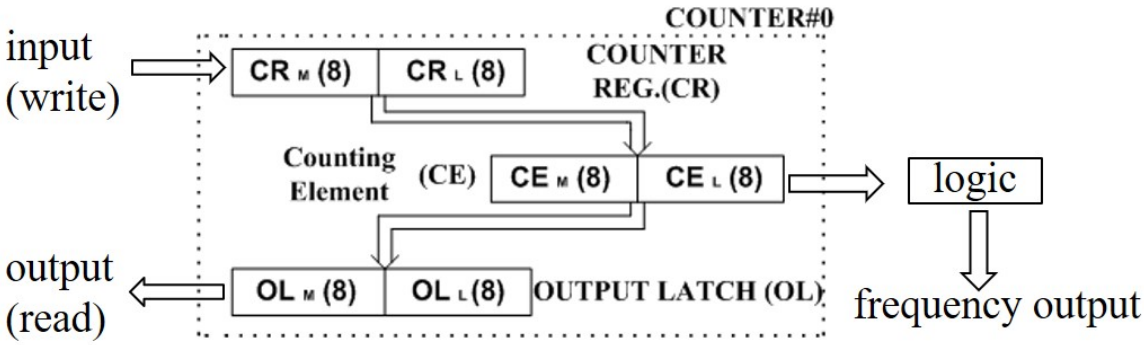
计时器

The 8253/54 Programmable Interval Timer is used to generate a lower frequency for various uses (e.g. event counter & accurate time delays)

software delay needs CPU busy running and not so accurate.



- 3 independent counters
- Gate is used to enable (High) or disable (Low) the counter
- CLK pin connects to the initial clock signal
- The new clock signal is generated in OUT pin
- input frequency is 1~65536 (BCD: 1~10000)
- Shape of output frequency
  - Square-wave
  - One-shot
  - Square-wave with various duty cycles (占空比,可用于驱动电机等)



- There is a Counting Register (CR) inside each counter, which stores initial count value
- There is a Counting Element (CE) inside each counter.
- CE begins to decrement until it reaches 0
- generate frequency output accordingly

Features of 8253

- 8253 takes one CLK pulse to convey the count from CR to CE
- On every CLK pulse's rising edge (0-to-1), 8253 will check the GATE
- On every CLK pulse's falling edge (1-to-0), 8253 will count down

Internal Structure and Pins

/CS	/RD	/WR	A1A0	FUNCTION
0	1	0	00	Write counter0 (to CR0)
0	1	0	01	Write counter1 (to CR1)
0	1	0	10	Write counter2 (to CR2)
0	1	0	11	Write control port
0	0	1	00	Read counter0 (from OL0)
0	0	1	01	Read counter1 (from OL1)
0	0	1	10	Read counter2 (from OL2)
0	0	1	11	Read control port (for *8254*)
1	X	X	XX	Not available

Control Word Register

Selected when A[0]=1&A[1]=1, used to specify which counter to be used, its mode, and a read or write operation.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC_1	SC_0	RW_1	RW_0	M_2	M_1	M_0	BCD

SC - Select counter

SC_1	SC_0	
0	0	Select counter 0
0	1	Select counter 1
1	0	Select counter 2
1	1	Illegal for 8253 Read -Back command for 8254 (See Read operations)

RW - Read /Write

RW_1	RW_0	
0	0	Counter latch command (See Read operations)
0	1	Read / Write least significant byte only
1	0	Read / Write most significant byte only
1	1	Read / write least significant byte first, then most significant byte

M - Mode

M_2	M_1	M_0	
0	0	0	Mode 0
0	0	1	Mode 1
x	1	0	Mode 2
x	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD :

0	Binary counter 16 - bits
1	Binary coded decimal (BCD) Counter (4 Decades)

Counters

- Each consists a single, 16-bit, pre-settable, down counter
- Can operate in either binary and BCD
- Input, gate and output are configured by the selection of modes
- Reading from a counter does not disturb the actual count in process.

Write/Read operations

Write

1. Write a control word into control register
2. Load the low-order byte of a count in the counter register
3. Load the high-order byte of a count in the counter register

Read

- Simple Read

  two I/O read operations: first one for low-order byte and last one for the high-order byte

- Counter Latch Command

  (1) One I/O write operation used to write a control word to the control register to latch a count in the output latch

  (2) Two I/O read operations are used to read the latched count as in Simple Read

Example

CS		A1A0	Port	Port address (hex)
1001	01	00	Counter 0	94
1001	01	01	Counter 1	95
1001	01	10	Counter 2	96
1001	01	11	Control register	97

(a) counter 0 for binary count of mode 3 (square wave) to divide CLK0 by number 4282 (BCD)
(b) counter 2 for binary count of mode 3 (square wave) to divide CLK2 by number C26A hex
(c) Find the frequency of OUT0 and OUT2 in (a) and (b) if CLK0 =1.2 MHz, CLK2 = 1.8 MHz.

Solution:

(a) To program counter 0 for mode 3, we have 00110111 for the control word. Therefore,

```
MOV   AL,37H      ;counter 0, mode 3, BCD
OUT   97H,AL      ;send it to control register
MOV   AX,4282H    ;load the divisor (BCD needs H for hex)
OUT   94H,AL      ;send the low byte
MOV   AL,AH       ;to counter 0
OUT   94H,AL      ;and then the high byte to counter 0
```

(b) By the same token:

```
MOV   AL,B6H      ;counter2, mode 3, binary(hex)
OUT   97H,AL      ;send it to control register
MOV   AX,C26AH    ;load the divisor
OUT   96H,AL      ;send the low byte
MOV   AL,AH       ;to count 2
OUT   96H,AL      ;send the high byte to counter 2
```

(c) The output frequency for OUT0 is 1.2MHz divided by 4282, which is 280 Hz. Notice that the program in part (a) used instruction "MOV AX,4282H" since BCD and hex numbers are represented in the same way, up to 9999. For OUT2, CLK2 of 1.8 MHz is divided by 49770 since C26AH = 49770 in decimal. Therefore, OUT2 frequency is a square wave of 36 Hz.

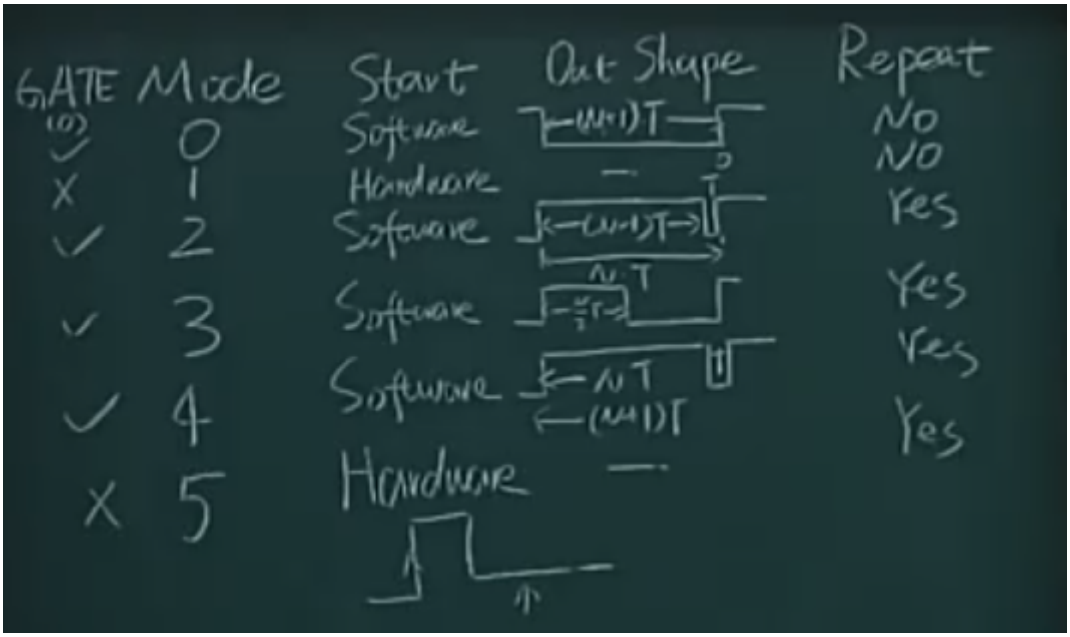
Operation Modes

可以通过级联来更多地降低频率

Main: mode 0, mode 2 and mode 3

Key concepts

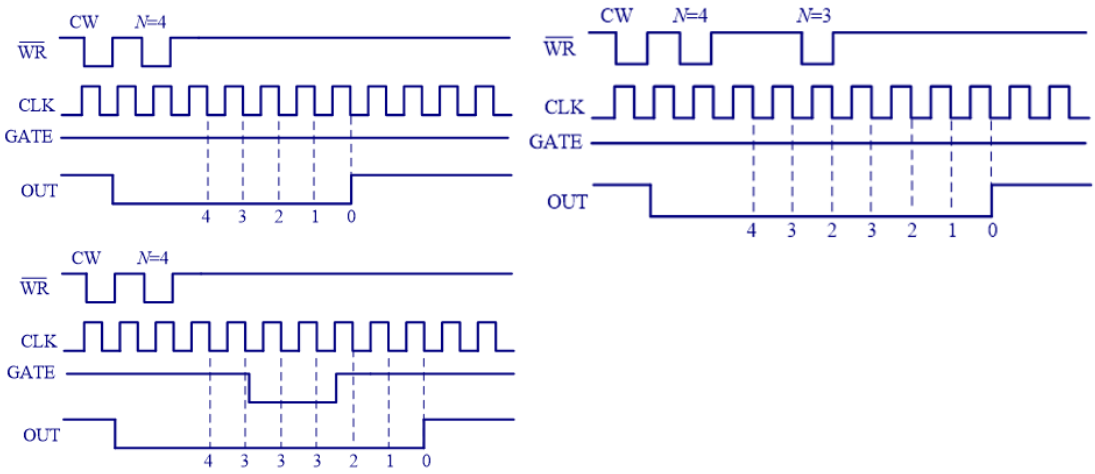
1. When to start counting? Software or hardware(GATE)
2. Output shape, square-wave or pulse
3. Does it repeat?



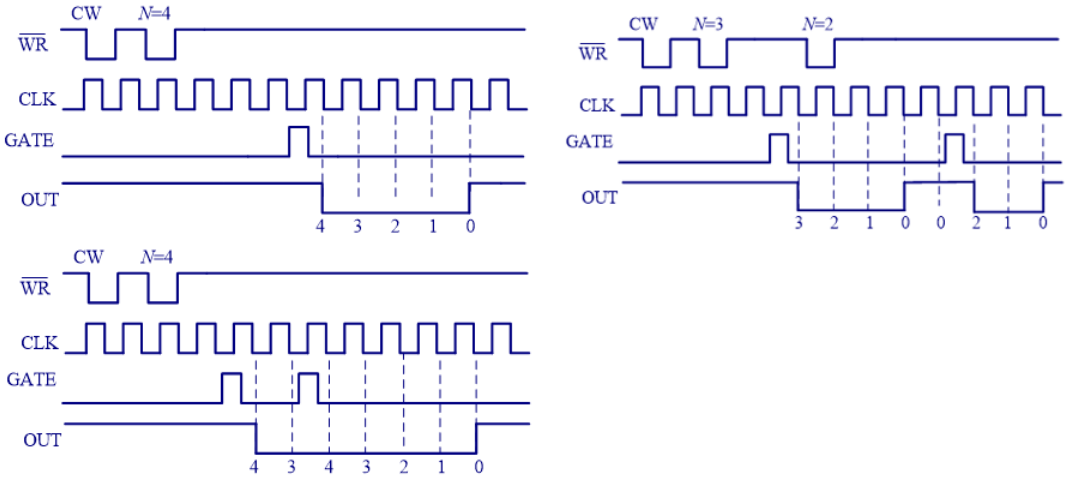
Mode	Start	Out Shape	Repeat	Stall(GATE=0)
0	soft	H | L (N+1)T | H	N	Y
1	hard	Same as mode 0	N	N
2	soft	L | H (N-1)T | L T | H	Y	Y
3	soft	L | H NT/2 | L NT/2 | H	Y	Y
4	soft	L |H NT |L T |H	N	Y
5	hard	Same as mode 4	N	N

Note: Refer to slides for details.

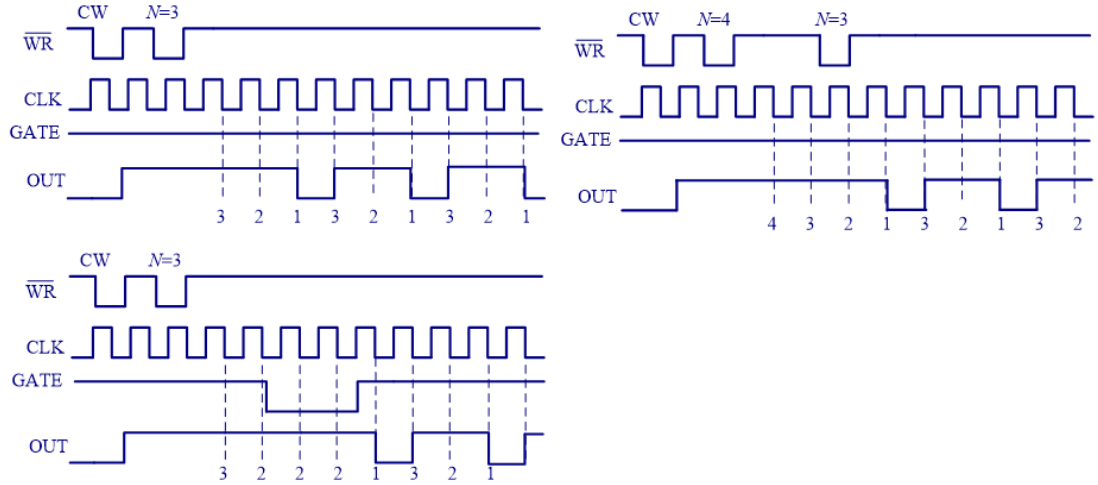
Mode 0: Interrupt on Terminal Count



Mode 1: Hardware Re-triggerable One-shot



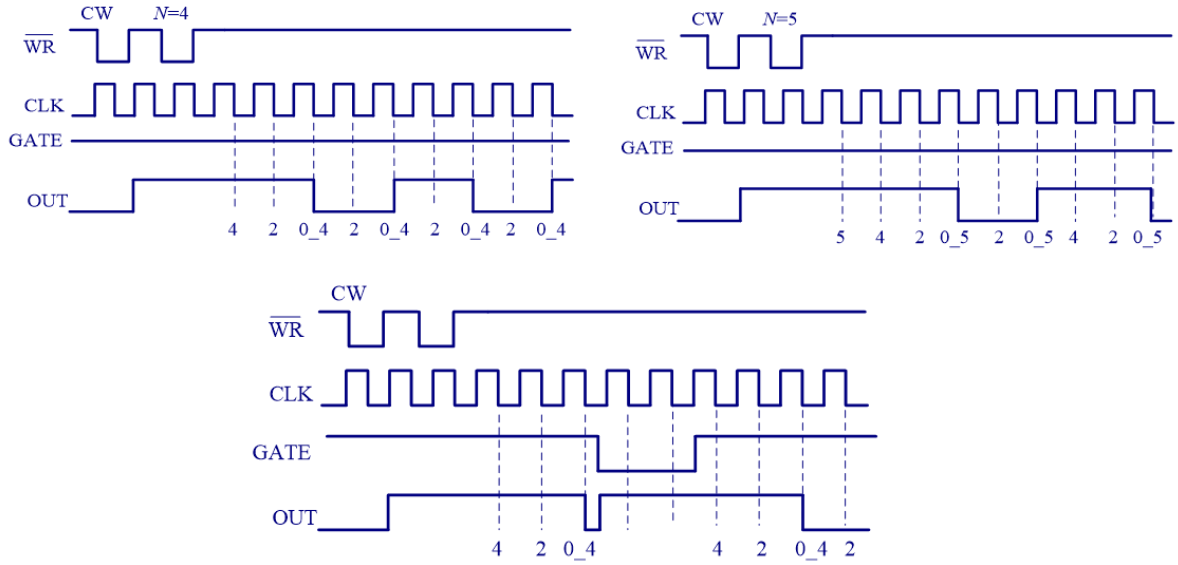
Mode 2: Rate Generator



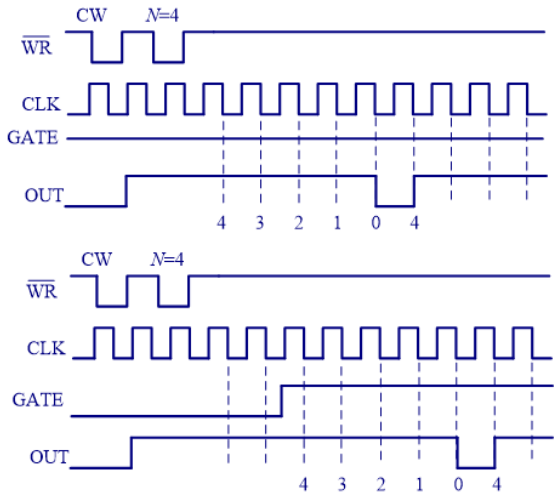
Mode 3: Square Wave Rate Generator

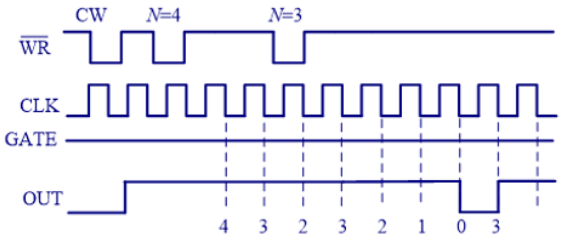
counter is decreased by two every time

If count is odd, output will be high for (n+1)/2 clock and low for (n-1)/2 clock

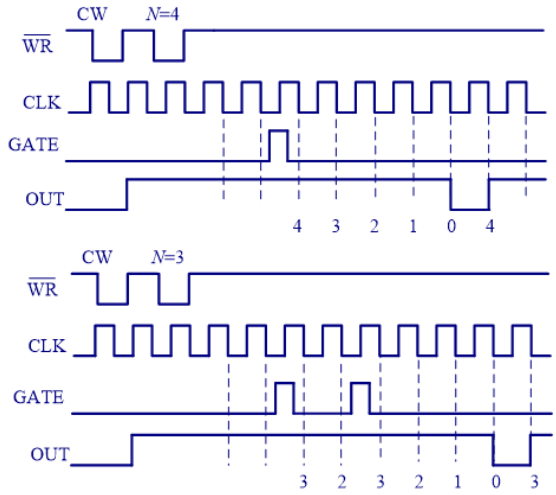


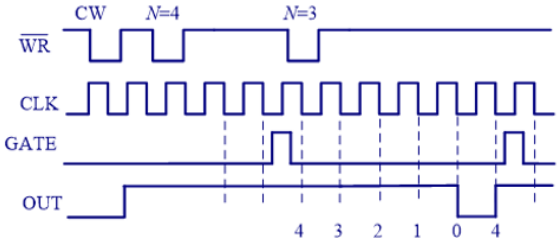
Mode 4: Software Triggered Strobe





Mode 5: Hardware Triggered Strobe





Chap10: Interrupts & 8259

中断管理

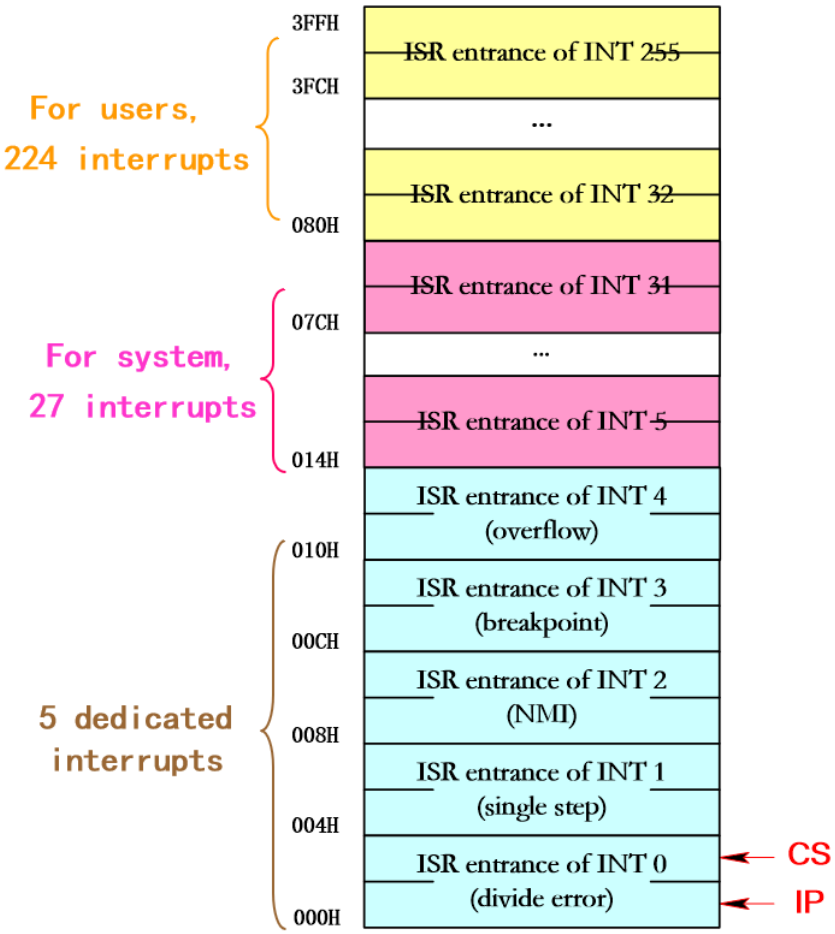
Interrupts

256 interrupt types (INT 00 ~ INT 0FFH)

Interrupt vector

- CS + IP, needs 4 bytes
- points the entrance address of the corresponding interrupt service routine (ISR)

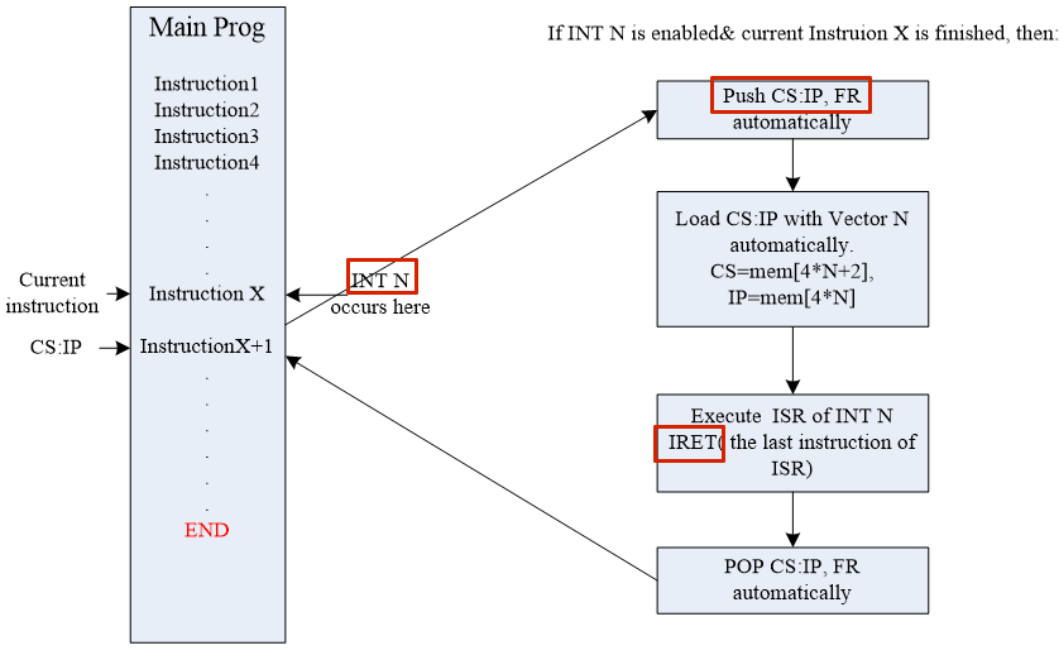
Interrupt vector table: The first 1KB is used to store the interrupt vectors



Main Program and ISR

Interrupt can happen anytime and anywhere. Thus, ISR should not change flag registers. (e.g. between CMP&JNZ)

Manually PUSH and POP other registers if used.



Categories of Interrupts

- Hardware interrupts (external interrupts)
  - Maskable (可屏蔽的) (from INTR)
  - Non-maskable (不可屏蔽的) (from NMI)
- Software interrupts (internal interrupts)
  - Using INT instructions
  - Predefined conditional (exception) interrupts

Hardware Interrupts

- Non-maskable interrupt
  - Trigger: NMI pin, input-signal, rising edge and two-cycle high activate
  - Type: INT 02
  - not affected by the IF (interrupt flag)
  - Reasons: e.g. RAM parity check error
- Maskable interrupt
  - Trigger: INTR pin, input-signal, high activate
  - Type: No predefined type (read from data bus)
  - IF = 1 , enable; IF = 0 , disable (use STI to set IF , and use CLI to clear IF)
  - Reasons: Interrupt requests of external I/O devices

Processing Maskable Interrupts

- CPU responds to INTR interrupt requests
  1. External I/O devices send interrupt requests to CPU
  2. CPU will check INTR pin on the last cycle of an instruction

     if the INTR is high and IF = 1, CPU responds to the interrupt request
  3. CPU sends two negative pulses on the ~INTA (INT answer) pin to the I/O device
  4. After receiving the second ~INTA , I/O device sends the interrupt type N on the data bus
- CPU executes the ISR of INT N
  1. CPU reads N from data bus
  2. Push the FR in the stack
  3. Clear IF (automatically disable nested interrupts) and TF (trap flag)
  4. Push the CS and IP of the next instruction in stack
  5. Load the ISR entrance address and moves to the ISR
  6. At the end of ISR, IRET will pop IP and CS and FR in turn, CPU returns to previous instruction

Software Interrupts

- INT xx instruction (e.g. system call)
  - An ISR is called upon instruction
  - CPU always responds and goes execute the corresponding ISR

    not affected by IF
  - can call any ISR by using INT instructions.
- Predefined conditional interrupts
  - INT 00 : (divide error)

    dividing a number by zero, or quotient is too large.

- INT 01 : (single step)

  If TF = 1 , CPU will generates an INT 1 interrupt after executing each instruction for debugging.

  ```
  ;How to clear TF ?
  PUSHF
  POP     AX
  AND     AX, 0FEFFH
  PUSH    AX
  POPF
  ```

- INT 03 : (breakpoints)

  When CPU hits the breakpoint set in the program, CPU generates INT 3 interrupt for debugging.

- INT 04 : (signed number overflow)

  INTO instruction (manual): check the OF after an arithmetic instruction

  ```
  do something
  INTO
  ```

Difference between INT and CALL

- CALL can jump anywhere; INT jumps to a fix location (corresponding ISR)
- CALL is in the sequence of instructions; an external interrupt can come in at any time
- CALL cannot be masked (disabled); an external interrupt can be masked
- CALL FAR saves CS:IP of next instruction; INT saves FR + CS:IP
- last instruction: RET v.s. IRET

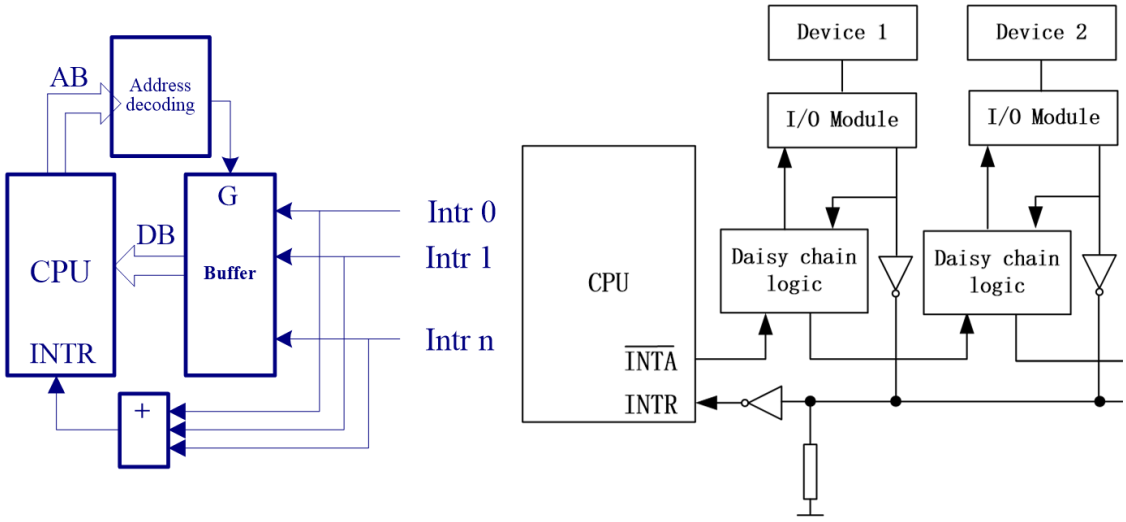
Interrupt Priority

INT > NMI > INTR

Different strategies for different external interrupt requests

- Software polling

  the sequence of checking determines the priority



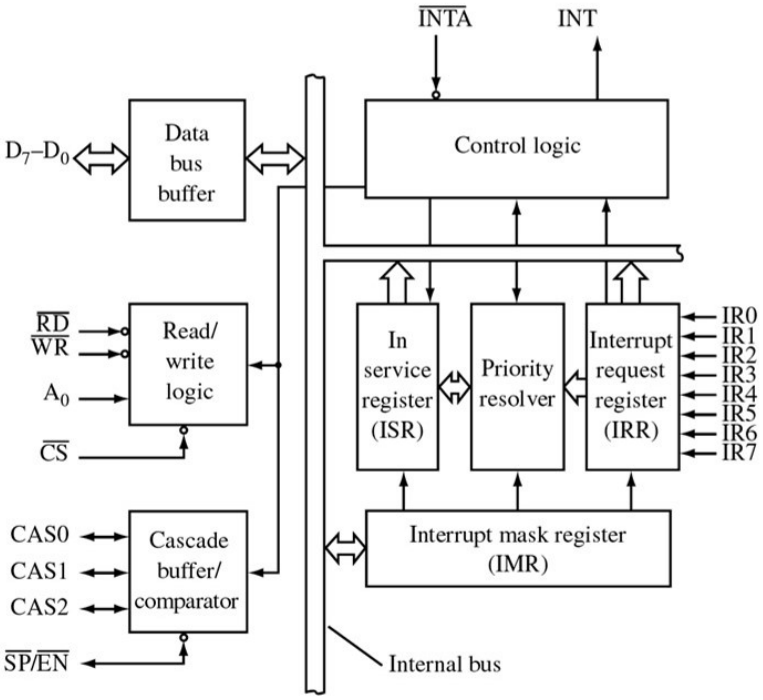
- Hardware checking

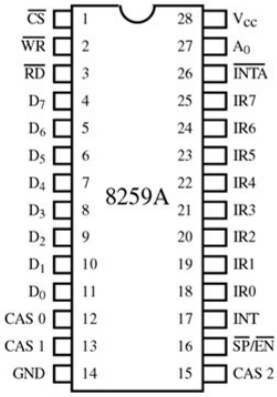
  the location in the daisy chain counts

- Interrupt Controller: 8259

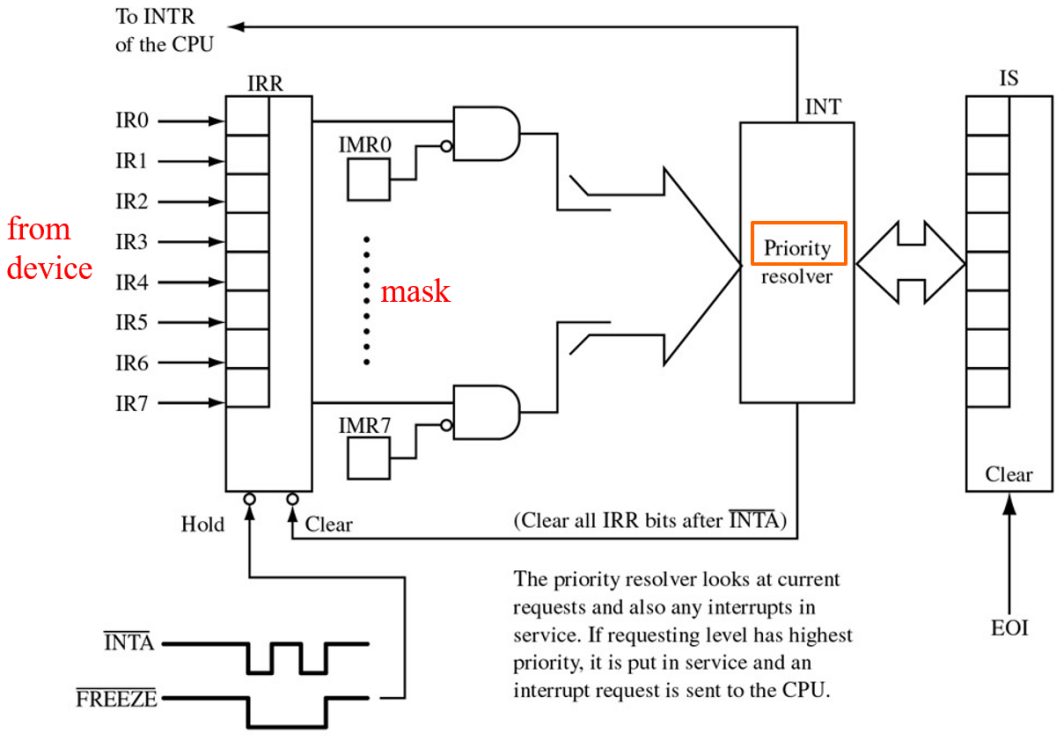
8259 Programmable Interrupt Controller (PIC)

- PIC can deal with up to 64 interrupt inputs
- interrupts can be masked
- various priority schemes can also programmed



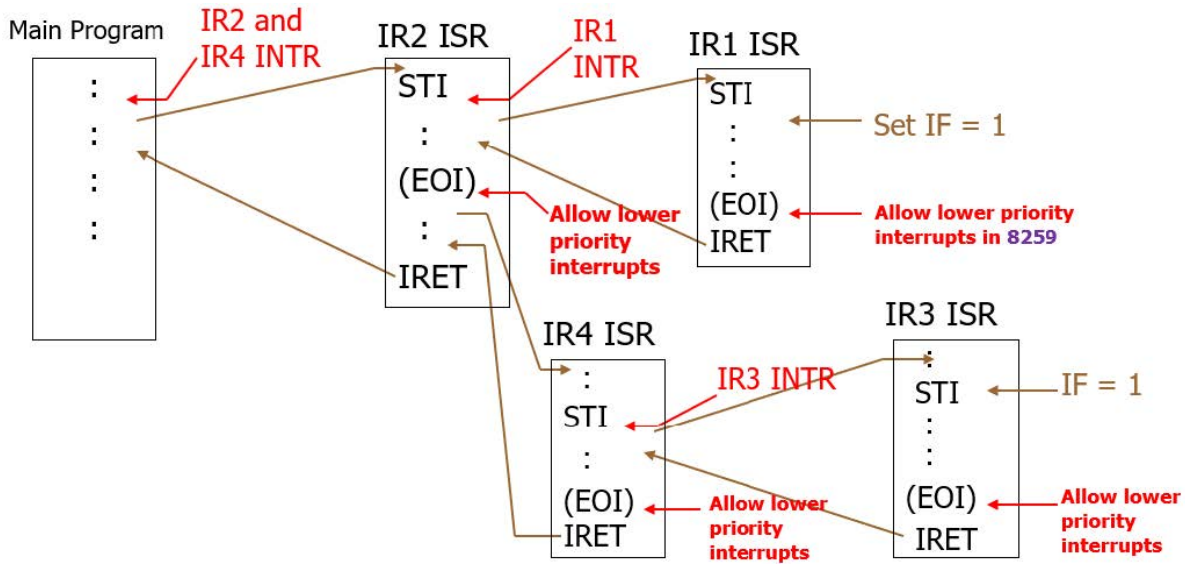


- 8 input interrupts (IR0~IR7), choose one and send it to CPU
- can set priority of IR
- can enable/disable IR[i] respectively



Interrupt Nesting

Higher priority interrupts can interrupt lower interrupts



STI is used to re-open the interrupt flag which is closed automatically.

```
...
(EOI)       ; end of interrupt
; can respond to other interrupt earlier
POP     ..
POP     ..
...
IRET
```

Chap11: BIOS and DOS Programming

Just need to understand the concepts. Refer to slides for details.

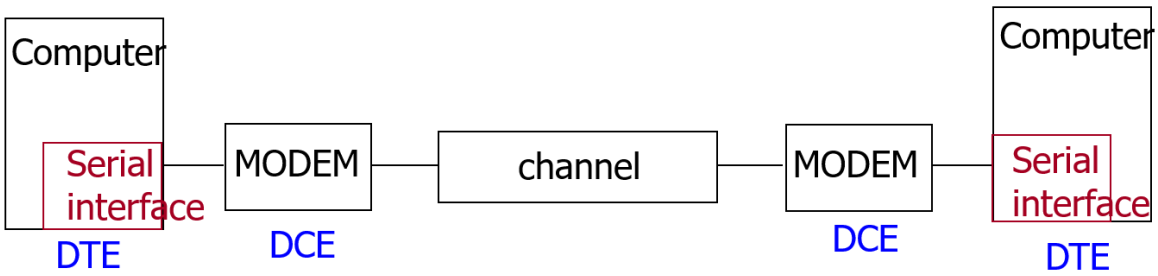
Chap12: Serial Data Communication & 8251

Just need to understand the concepts. Refer to slides for details.

Data transmission is the transfer of data from point to point often represented as an electromagnetic signal over a physical communication channel

A **communication channel** (信道) refers to the medium used to convey information from a sender (or transmitter) to a receiver.

Parallel data transfers VS Serial data transfers



Serial Communication

Data transfer rate

- Symbol rate

  The number of distinct symbol or pulse changes (signaling events) made to the transmission medium per second, quantified using the baud unit (波特率)

  Each symbol can represent one (binary encoded signal) or several bits of data (e.g. use 0/0.25/0.5/0.75 to represent 2 bits)

- Bit rate

  the number of bits that are conveyed or processed per unit of time, quantified using the bits per second

Synchronization Methods

- Asynchronous communication

  The starting of each byte is asynchronous

- Synchronous communication

  The sender and the receiver are synchronized at the beginning of data transfer using synch characters (e.g. use same CLK)

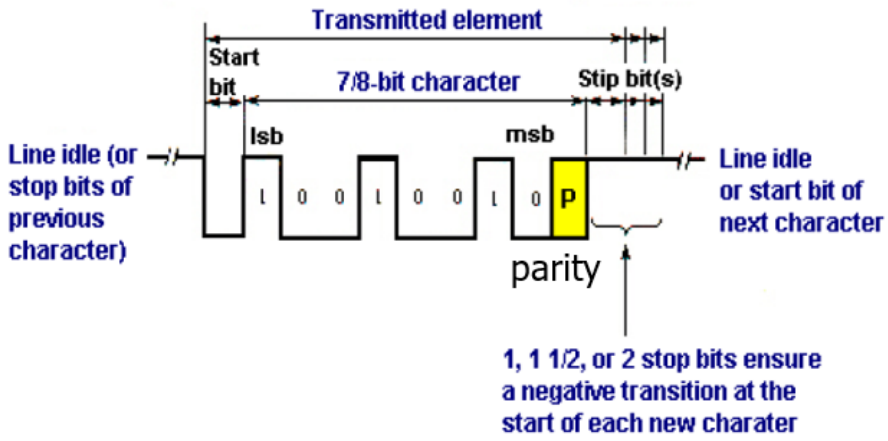
Asynchronous Communication

Need protocol.

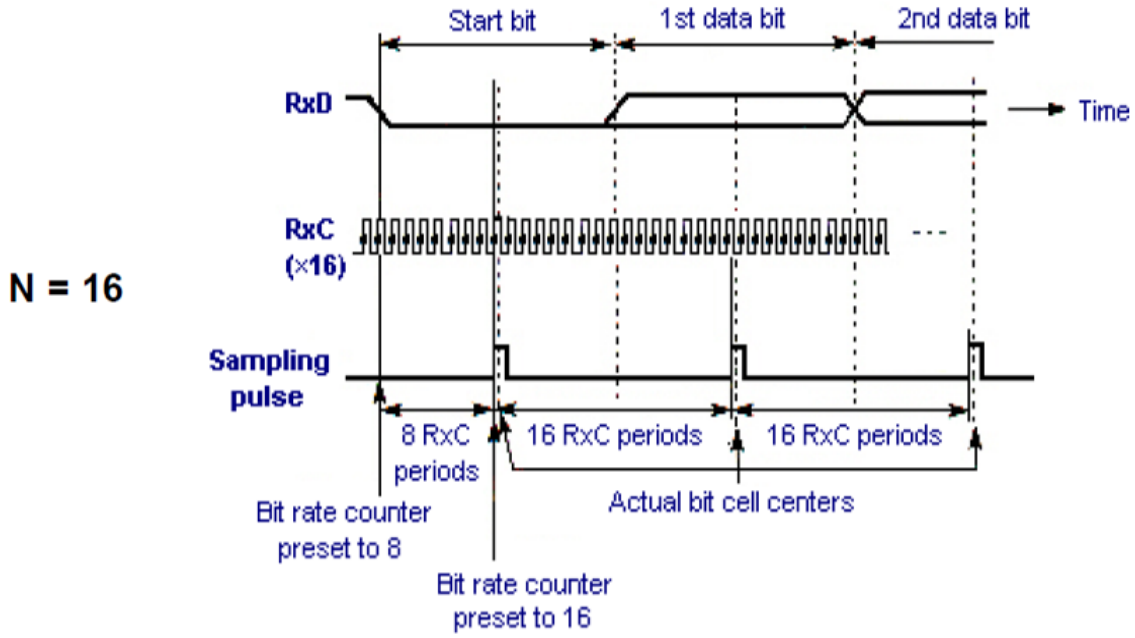
Each character (byte) is encapsulated between an additional start bit (1 $\rightarrow$ 0) and one or more stop bits (2 high bit)

The state of transmission line between character is idle state.

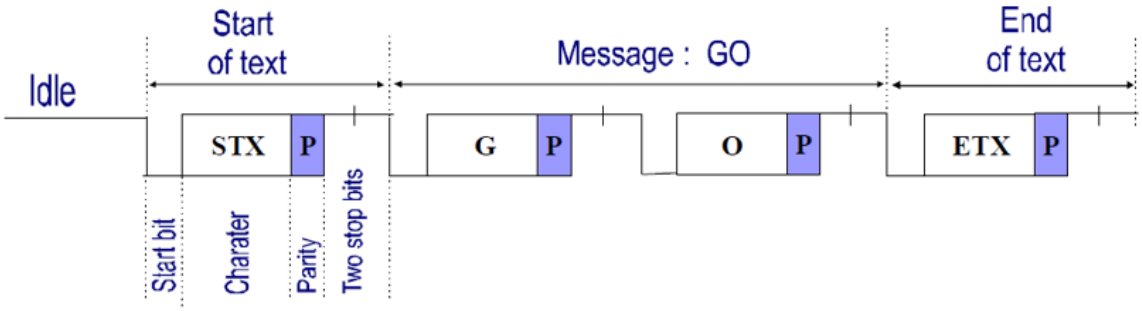
> **Example**
>
> Under this transmit method
>
> 100 character per second $\Rightarrow$ baud rate 100*(1+8+1+2)=1200bps



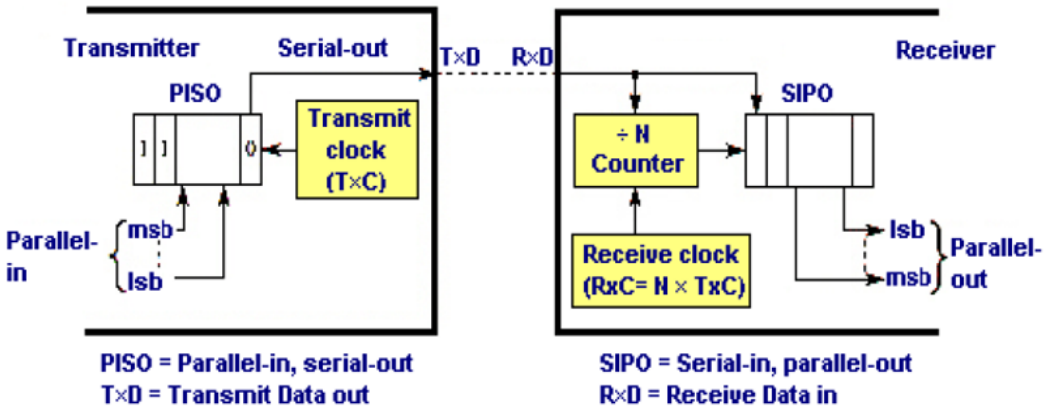
Each bit is sampled at the center. The receiver clock is N (baud factor) times the transmitted bit rate.



frame consists of bytes and start&end of text

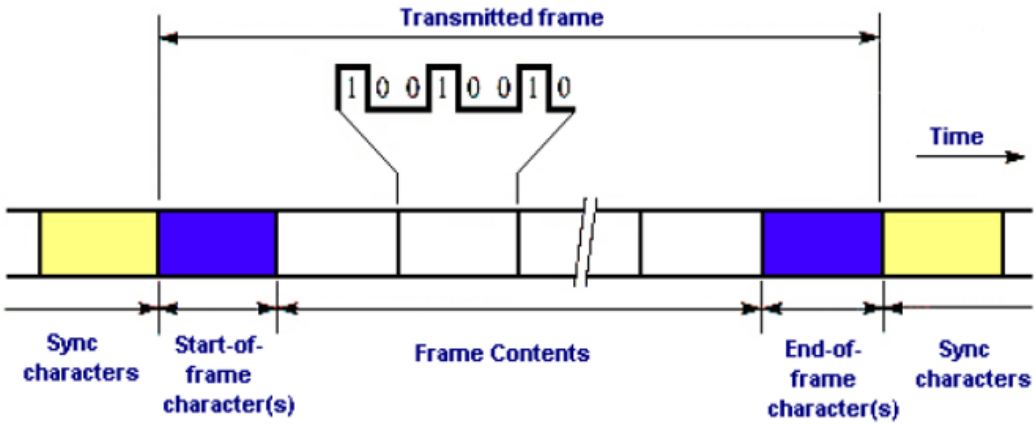


summary

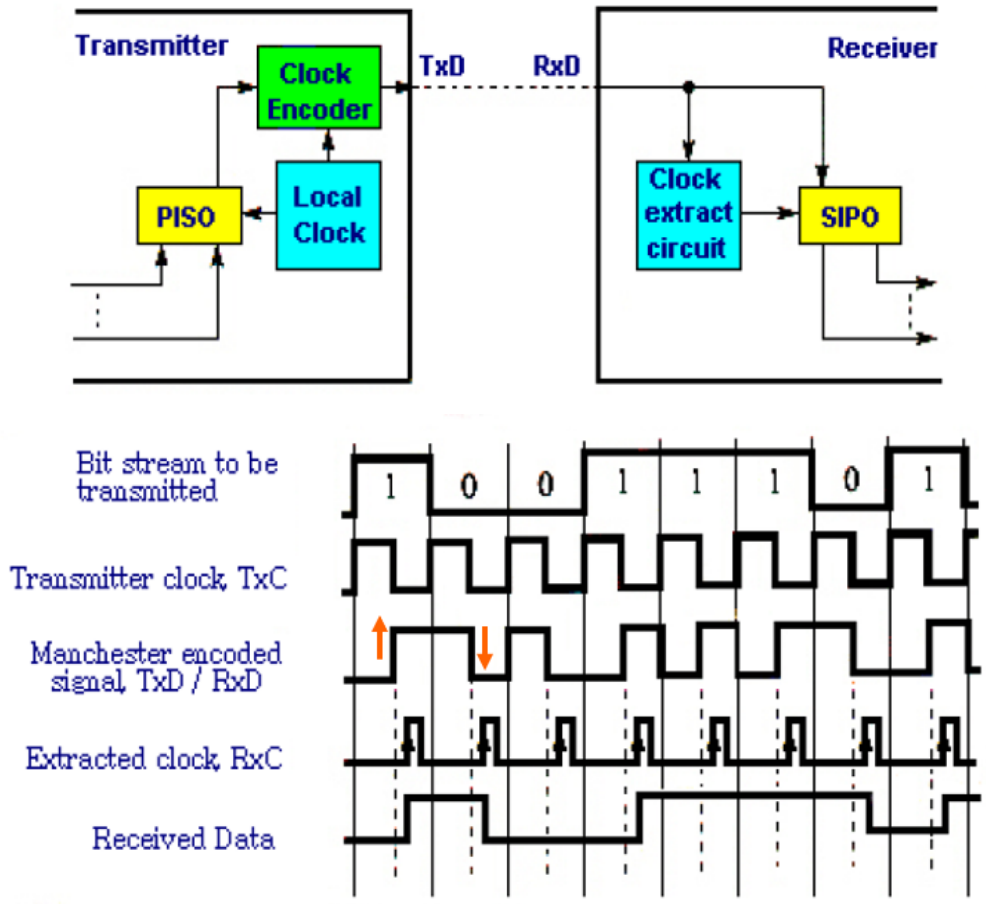


Synchronous Transmission

The complete block of frame is transmitted as a contiguous stream with no delay between each 8-bit.



Clock encoding and extraction



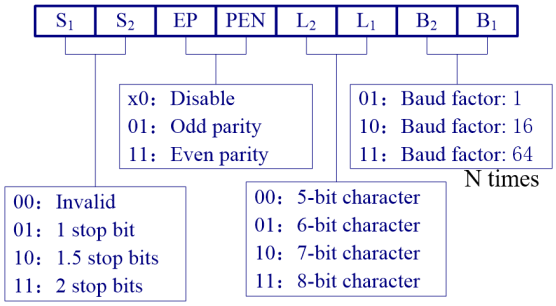
Use characters as start&end VS Use bits

8251 USART Chip

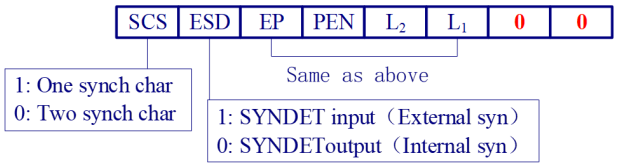
- Capable of doing both asynchronous and synchronous data communication
  - synchronous: baud rate 0-64K, characters can be 5, 6, 7, or 8 bits, automatically detect or insert sync characters
  - Asynchronous: baud rate 0-19.2K, characters can be 5, 6, 7, or 8 bits, automatically insert start, stop and parity bits, TxC and RxC clocks can be 1, 16, or 64 times of the baud rate
- Full duplex, double-buffered

- Error checking circuit

8251 Mode Word



Asynchronous



Synchronous