EE204 Notes

Organization

Von Neumann Architecture

1. Five components partitioning: input, output, memory, ALU, control unit
2. Three key concepts:
    1. Both instructions and data are stored in a **single** read-write memory
    2. The contents of memory are **addressable** by location, without regard to the type of data
    3. Execution occurs in a **sequential** fashion

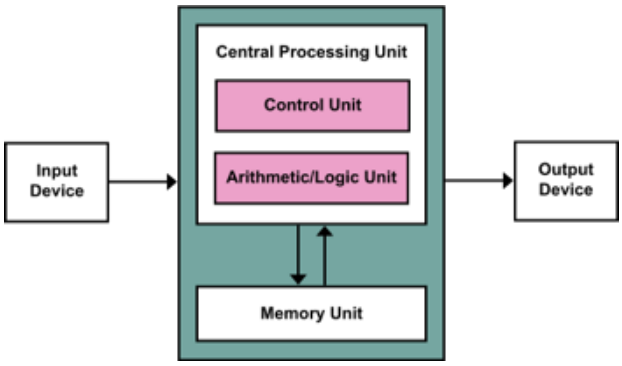

Figure 1:

Compared to Harvard architecture:

Figure 2:

Microprocessor

- the CPU circuitry can been reduced to *IC* (Integrated Circuit) *s*cale, consisting of **ALU**, **CU** and **registers**
- contains **no** RAM, ROM, or I/O ports on the chip itself

Microcomputer

CPU + Memory + IO ports + Bus

Bus: address bus / data bus / control bus

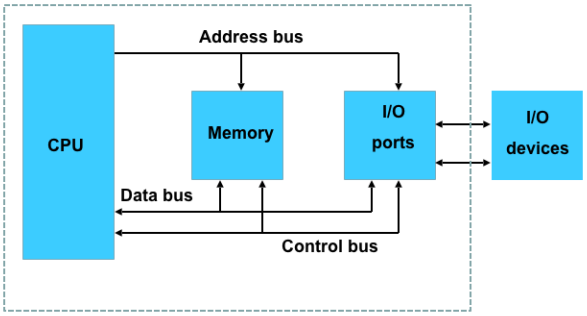

Figure 3:

Microcomputer system = microcomputer + peripheral IO + software

CPU

ALU + CU (instruction decoder + program counter)

CISC:

- Variable execution time of different format instructions
- Variable instruction length (1 word to n words)

RISC:

- Fixed size (1 word)
- Fixed time for all instructions

Memory

- Bit (b)
- Byte (B)
- Nibble: half a byte

- Word: the number of bits that a CPU can process at one time. Depends on the width of the CPU's registers and that of the data bus
- Double word

Characteristics

- Location: CPU / Internal / External
- Capacity: Word size / Number of words
- Unit of transfer: interal (usually a word) / external (usually a block)
- Addressable unit: byte / cluster
- Performance: access time / memory cycle time / transfer rate
- RAM (DRAM / SRAM), ROM
- Hierarchy list: Registers, L1, L2, Main, Disk cache, Disk, optical, tape

Chip Organization

Physical arrangement of bits into words (chip word, NOT system word). e.g., 16M can be 1M of 16-bit words, or $2048 \times 2048 \times 4$ bit array (multiplex row and column to reduce pins).

Organize a memory module:

- If the module needs bigger unit of transfer than that of given memory chips, *bit extension*.
- If the module needs larger number of words than that of given memory chips, *word extension*.

Input/Output

I/O Modules

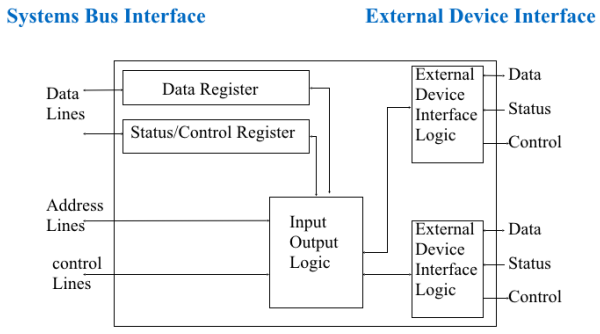

Figure 4:

- Why: to fit wide variety of peripherals:
  - hide or reveal device properties to CPU
  - support multiple or single device
  - control device functions or leave for CPU
- Functions: Control/Timing, CPU Communication, Device Communication, Data buffering, error detection
- I/O steps (data from an external to processor):
  - I/O module returns the device status
  - If the device is ready, CPU requests data transfer by means of a command to the I/O module
  - I/O module gets a unit of data from device
  - I/O module transfers the data to CPU
  - Variations for output, DMA, etc.

Input output techniques

Programmed I/O

- CPU executes a program that gives it direct control of the I/O operation: Sensing device status, Read/write commands to the I/O module, Transferring data
- I/O Commands:
  - CPU issues address: Identifies module (& device if >1 per module)
  - CPU issues commands: control, test, read, write
  - CPU commands contain the identifier (address) of the corresponding module (and device)
- Simple, but if CPU is faster, it's a huge waste of CPU time.
- Details
  - CPU requests I/O operation
  - I/O module performs operation
  - I/O module sets status bits
  - CPU checks status bits periodically
  - I/O module does not inform CPU directly
  - I/O module does not interrupt CPU

– CPU may wait or come back later (for example, with the help of time-sharing OS)

Interrupt Driven I/O

1. CPU requests I/O operation
2. I/O module performs operation whilst CPU does other work
3. I/O module informs CPU when something comes up by interrupting CPU
4. CPU deals with this event

Handling an interrupt: from a protocol perspective

Identifying Interrupt Module:

- Connect a dedicated line for each module
- Software poll
- Daisy chain or hardware poll: Int ACK signal is sent down a chain
- But master: module must claim the bus before raising interrupt. e.g., PCI & SCSI
- Interrupt controller. e.g, 8259

Localizing handler programs:

- Using a general handler program. Con: handler is a program; location is fixed; less flexible.
- Interrupt vectors: store a pointer (int vector) in a fixed location.

Dealing with multiple interrupts:

- Set priorities for interrupts
- Nesting of interrupts

Problem with Programmed and Interrupt-driven I/O? They both need the involvement of CPU.

Direct Memory Access

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

DMA controller needs to know: read/write, device address, starting address of memory block for data, amount of data to be transferred. DMA controller sends interrupt when finished.

DMA Transfer Cycle Stealing: In an instruction cycle, the processor may be suspended due to DMA operation. CPU suspended just before it accesses bus; DMA controller takes over bus for a cycle; Transfer of one word of data; Not an interrupt: CPU does not switch

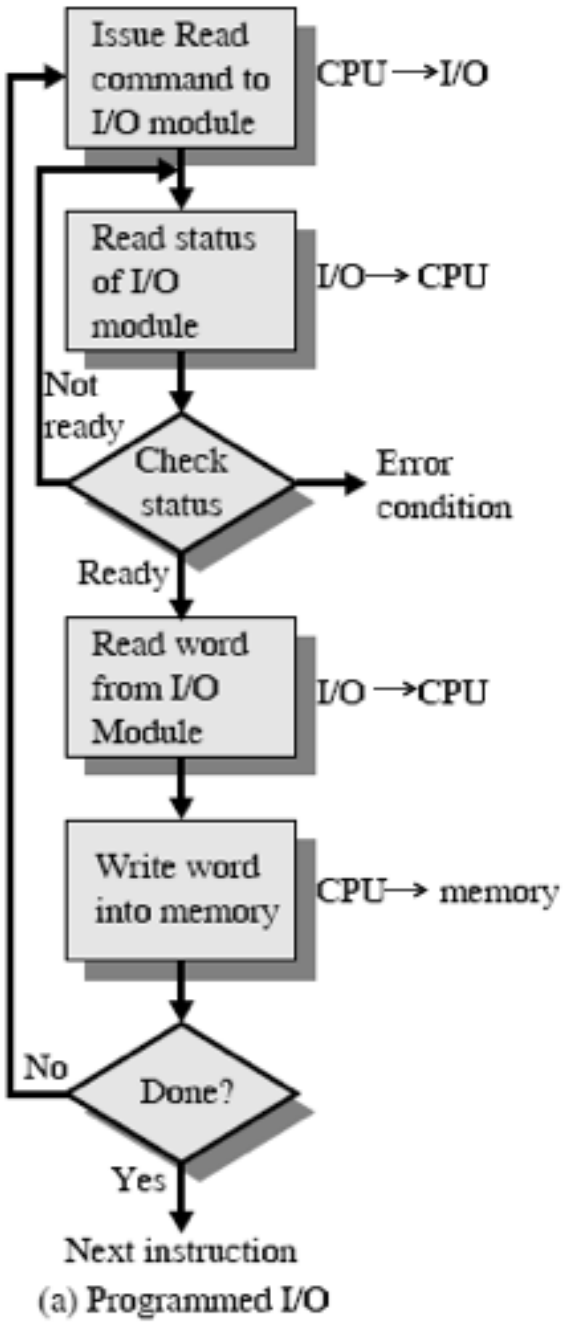

Figure 5:

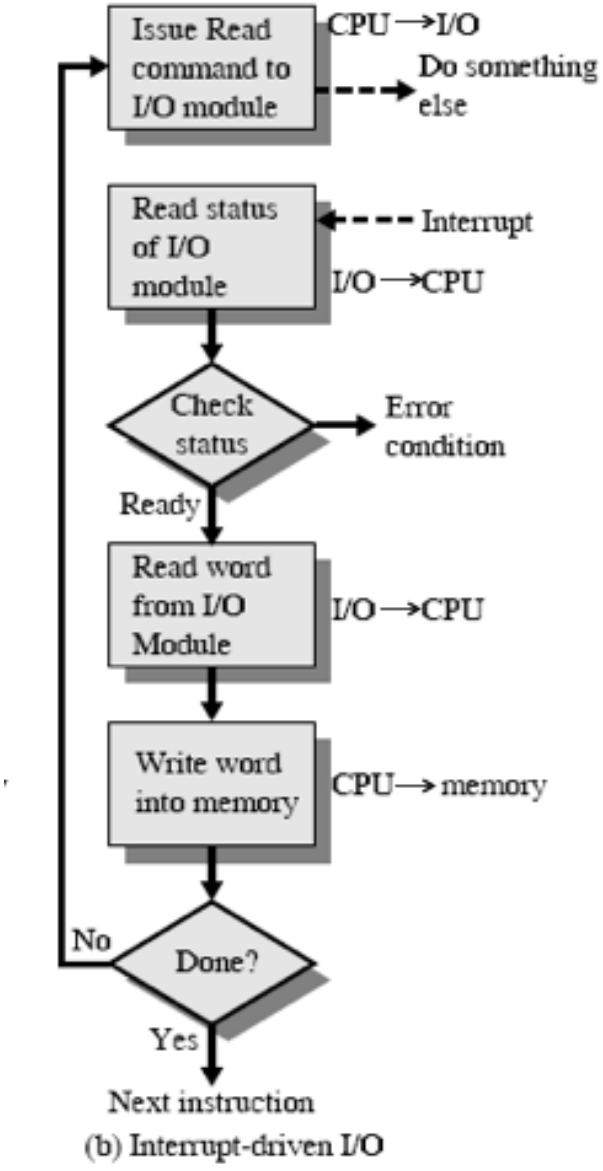

Figure 6:

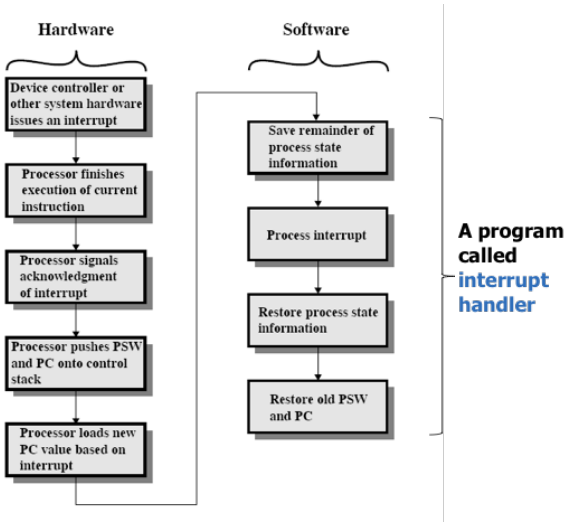

Figure 7:

context; Slows down CPU but not as much as CPU doing transfer.

Suspend twice:

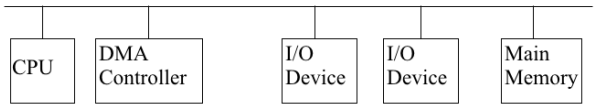

Figure 8:

Suspend once:

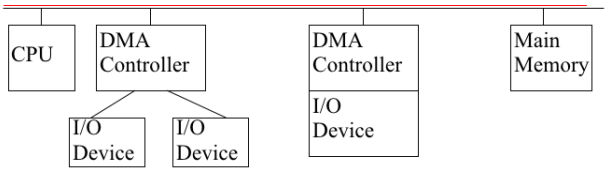

Figure 9:

Bus

A **bus** is a communication pathway connecting two or more devices. Bus is a **shared** transmission medium, so one device at a time.

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

Arbitration: distributed protocols, e.g., CSMA/CD in Ethernet; centralized scheme, e.g., Master/Slave.

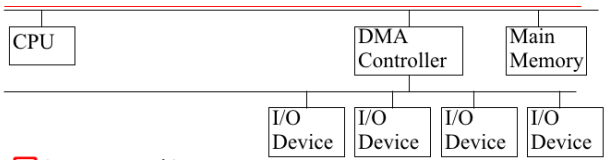

Figure 10:

Type: dedicated / multiplexed. Timing: synchronous / asynchronous.

Single-bus structure

A bus connects all modules. Pro: simple; Con: poor performance in terms of throughput.

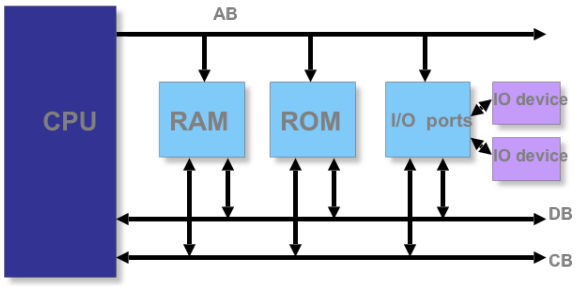

Figure 11:

CPU-Central Dual-Bus Structure

A dedicated bus between CPU and memory, and a dedicated bus between CPU and I/O devices. Pro: efficient in terms of data transfer. Con: information between memory and I/O devices has to go through CPU. Therefore, poor CPU performance.

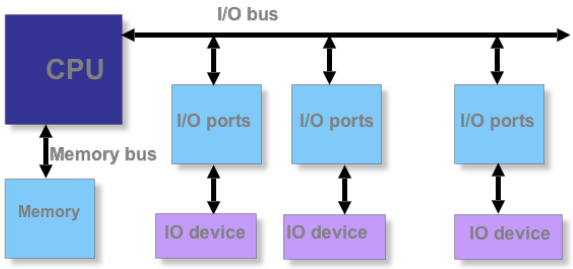

Figure 12:

Memory-Central Dual-Bus Structure

Gain both High CPU performance and data transfer throughput.

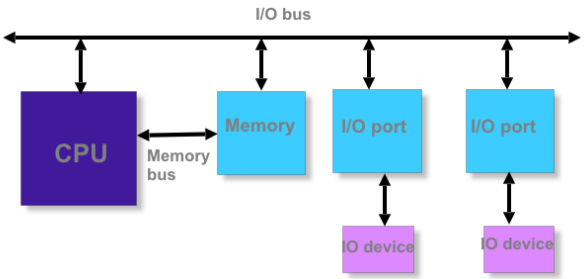

Figure 13:

Data bus

- Used to provide a path for moving data between system modules
- Bidirectional
  - CPU read: Memory (I/O device) → CPU
  - CPU write: CPU → Memory (I/O device)
- The width of data bus is as wide as the registers of a CPU (i.e. the width of a *word*). Determines how much data the processor can read or write in one memory or I/O cycle, which also defines a word of this computer.

Address bus

- Used to designate the source or destination of the data on the data bus that the processor intends to communicate with
- Unidirectional: CPU → memory or I/O devices
- The width of address bus, n → addressable memory 2^n.

Control bus

- Used to control each module and the use of data and address buses: Command and timing information between modules
- Consists of two sets of unidirectional control signals
  - Command signal: CPU → Memory (I/O device)
  - State signal: Memory (I/O device) → CPU

Note: Input/Output is defined from the processor's point of view

Addressing scheme

Memory mapped I/O

One 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.

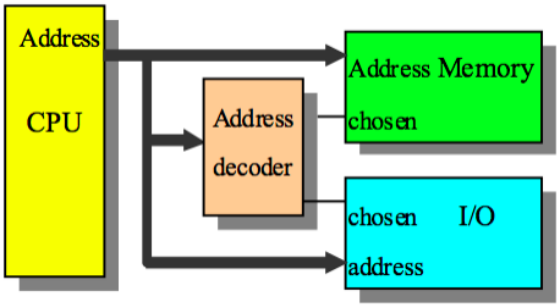

Figure 14:

Isolated I/O

Two separated address spaces for memory and I/O modules. Using different sets of accessing instructions. Special (limited) commands for I/O.

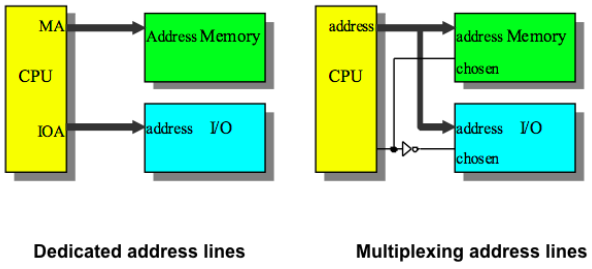

Figure 15:

Essential difference: CPU does not send control signal in Isolated I/O.

Microcontrollers (MCS)

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

Embedded Systems

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

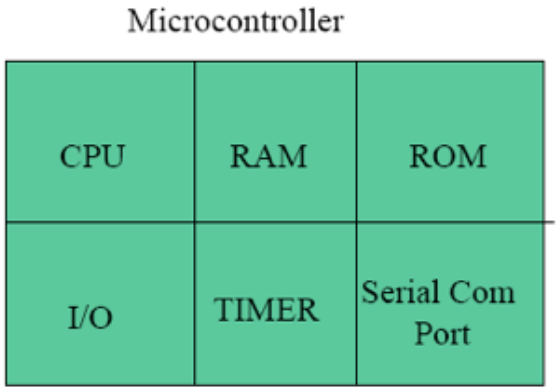

Figure 16:

Using microcontrollers is cheap but sometimes inadequate for the task.

- Microcontrollers differ in terms of their RAM,ROM, I/O sizes and type. ROM often as program memory (like BIOS), RAM often as program memory and data memory.

80x86 Microprocessor

Structure

8086, 16-bit, 20-bit address/data bus. 8088, 16-bit internal, 8-bit external

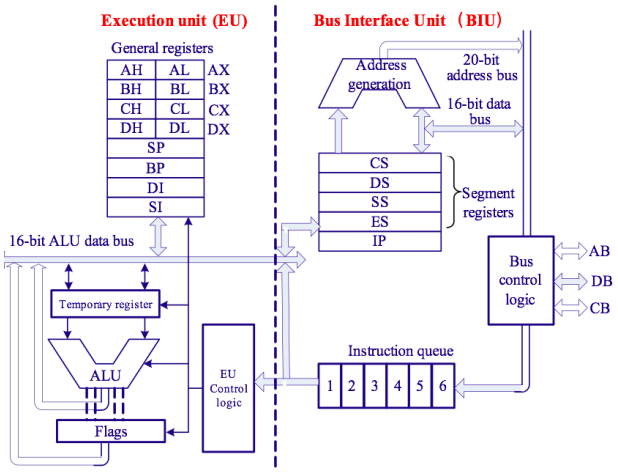

Figure 17:

Two sections: Bus interface unit (BIU, takes in charge of data transfer between CPU and memory as well as I/O devices) and Execution unit (EU, takes in charge of instruction execution). Work simultaneously (pipeline).

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).

Figure 18:

9 of 16 bits are used in FR.

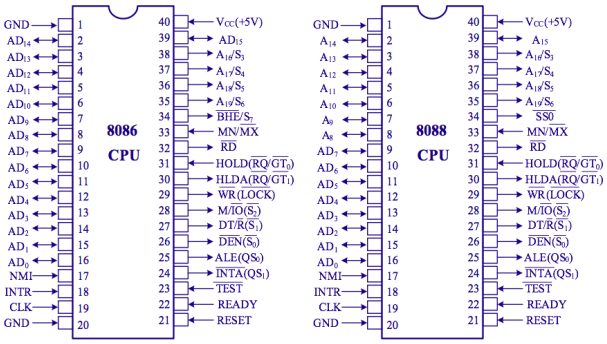

Figure 19:

8086/88's two work modes:

- Minimum mode: $MN/\overline{MX} = 1$. Single CPU. Control signal from CPU.
- Maximum mode: $MN/\overline{MX} = 0$. Multiple CPU. 8288 chip supports.

Control Signals

- **MN/~MX:** Minimum mode (high level), Maximum mode (low level)
- **~RD:** output, CPU is reading from memory or IO
- **~WR:** output, CPU is writing to memory or IO
- **M/~IO:** output, CPU is accessing memory (high level) or IO (low level)
- **READY:** input, memory/IO is ready for data transfer
- **~DEN:** output, used to enable the data transceivers

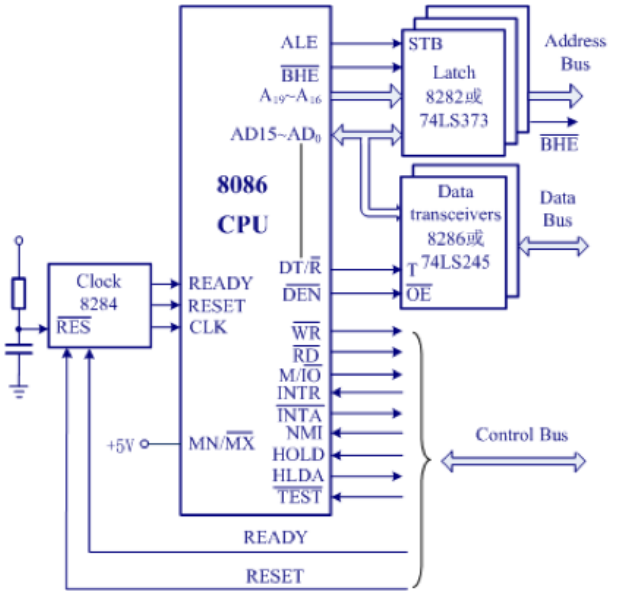

Figure 20:

- **DT/~R:** output, used to inform the data transceivers the direction of data transfer, i.e., sending data (high level) or receiving data (low level)
- **~BHE:** output, ~BHE=0, AD8-AD15 are used, ~BHE=1, AD8-AD15 are not in use
- **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 the instruction queue are cleared
  - CS = FFFFH
  - What is the address of the first instruction that the CPU will execute after reset?

8086/88 Bus Cycle

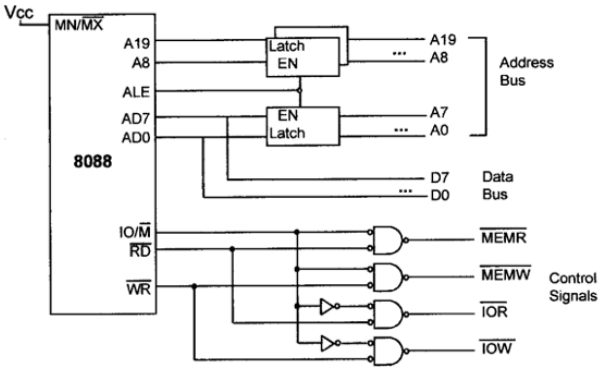

Figure 21:

At least 4 bus cycles.

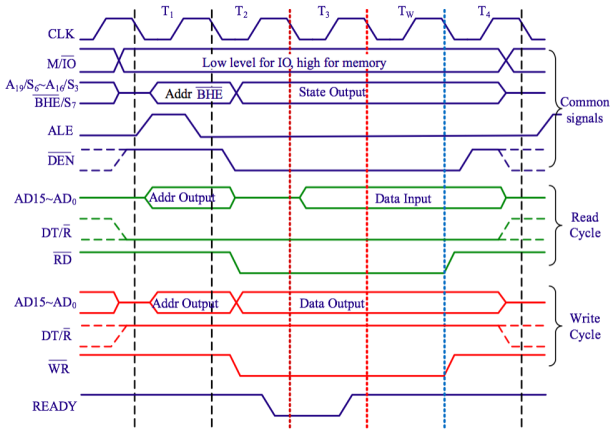

Figure 22:

8086 Programming

- A typical program on 8086 consists of at least three segments
- – code segment: contains instructions that accomplish certain tasks
  - data segment: stores information to be processed
  - stack segment: store information temporarily
- What is a segment: A memory block includes up to 64KB (for IP). Begins on an address evenly divisible by 16, i.e., an address looks like in XXXX0H (for CS).
- 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., an instruction in the code segment has a logical address in the form of `CS` (code segment register)`:IP` (instruction pointer). Map to `(CS << 4) + IP`.
- Physical Address Wrap-around, when adding the offset to the shifted segment value results in an address beyond the maximum value FFFFFH.
- Segment overlapping: Dynamic behaviour of the segment and offset concept map be desirable in some circumstances.

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

Figure 23:

Code Segment

- What if desired instructions are physically located beyond the current code segment? *Change the CS value so that those instructions can be located using new logical addresses.*

Data Segment

- Information to be processed is stored in the data segment.
- Logical address of a piece of data: `DS:offset`
  - Offset value: e.g., 0000H, 23FFH
  - Offset registers for data segment: **BX, SI and DI**.
- 8086 is **little endian**: the low byte of the data goes to the low memory location.

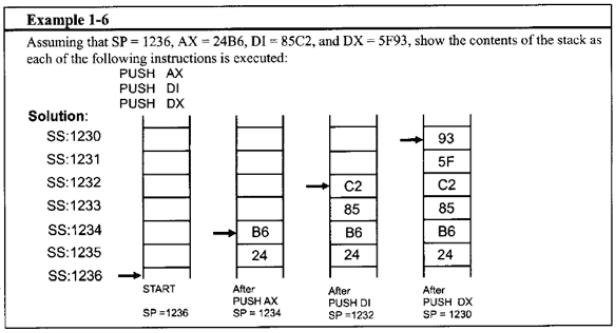

Figure 24:

Stack Segment

A section of RAM memory used by the CPU to store information temporarily

- 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 the CPU from the stack using `push` and `pop`, respectively
- Grows **downward** from upper addresses to lower addresses in the memory allocated for a program. (to protect other programs from destruction; ensure that the code section and stack section would not write over each other)

Extra Segment

An extra data segment, essential for string operations

BIOS function

Basic input-output system (BIOS)

- 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

Flag Register

- 16-bit, status register, processor status word (PSW)
- 6 conditional flags CF, PF, AF, ZF, SF, and OF
  - **CF (Carry Flag):** set whenever there is a carry out, from d7 after a 8-bit op, from d15 after a 16-bit op, *used to detect errors in unsigned arithmetic operations.*
  - **PF (Parity Flag):** the parity of the op result's low-order byte, set when the byte has an even number of 1s
  - **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 op

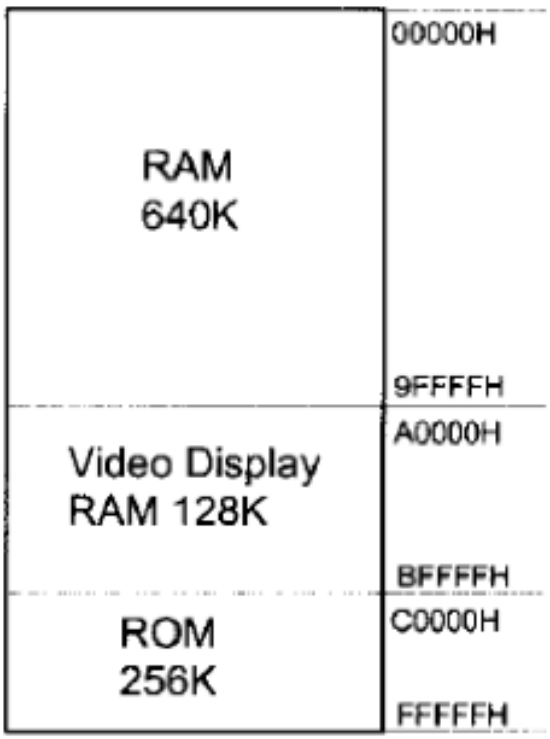

Figure 25:

  - **OF (Overflow Flag):** set when the result of a signed number operation is too large, causing the sign bit error, *used to detect errors in signed arithmetic operations.*
- 3 control flags: DF, IF, TF
  - **IF (Interrupt Flag):** set or cleared to enable or disable only the external maskable interrupt requests. After reset, all flags are cleared which means you (as a programmer) have to set IF in your program if allow INTR.
  - **DF (Direction Flag):** indicates the direction of string operations
  - **TF (Trap Flag):** when set it allows the program to single-step, meaning to execute one instruction at a time for debugging purposes

Addressing Modes

`MOV dest, source`. `dest` and `source` should have the same size.

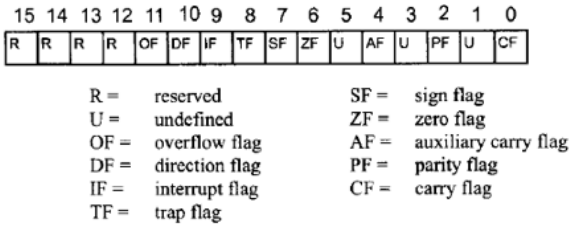

Figure 26:

Register addressing mode

```
MOV BX, DX
MOV ES, AX
```

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

Immediate addressing mode

The source operand is a constant. Cannot be segment registers

```
MOV AX, 2550H
MOV CX, 625 ;decimal
MOV BL, 40H
```

Direct addressing mode

Offset address in the data segment (DS) by default.

```
MOV DL, [2400] ; move contents of DS:2400H
MOV [3518], AL
```

Register indirect addressing mode

Offset address in the data segment (DS) by default. Registers for this purpose are SI, DI, and BX.

```
MOV AL, [BX]   ; move contents of DS:BX
```

Based relative addressing mode

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 and DS:BX+11 into CX
MOV AL, [BP]+10  ; PA = (SS << 4) + BP + 5
```

Indexed relative addressing mode

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

```
MOV DX, [SI]+5
MOV [DI-8], BL
```

Based indexed relative addressing mode

```
MOV CL, [BX][DI]+8 ; PA = (DS<<4) + BX + DI + 8
MOV AH, [SI+29+BP] ; order does not matter
MOV AH, [SI][BP]+29
```

Segment overrides

```
MOV AX, CS:[BP]         ; override default SS
MOV SS:[BX][DI]+32, AX ; override default DS
```

Assembly

Full segment definition

```
DaSeg1 segment
    str1 db 'Hello World! $'
DaSeg1 ends

StSeg segment
    dw   128  dup(0)
StSeg ends

CoSeg segment
    start   proc far
        assume cs:CoSeg, ss:StSeg

        mov ax, DaSeg1     ; set segment registers:
        mov ds, ax
        mov es, ax

        call subr       ;call subroutine

        mov ah, 1        ; wait for any key....
        int 21h

        mov ah, 4ch      ; exit to operating system.
        int 21h
   start endp

    subr proc

        mov dx, offset str1
        mov ah, 9
        int 21h          ; output string at ds:dx

        ret
    subr endp

CoSeg ends

     end start   ; set entry point and stop the assembler.
```

Figure 27:

Simplified segment definition

```
;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

Figure 28:

- The **MODEL** directive: Selects the size of the memory model
- DOS determines the CS and SS segment registers automatically. DS (and ES) has to be manually specified.
- Procedures definition, Entrance proc should be FAR.

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

- Program starts from the entrance. Ends whenever calls 21H interruption with AH = 4CH.
- Conditional Jumps

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

Figure 29:

- `JMP [SHORT|NEAR|FAR PTR] label`. Near by default.
- Data types: Directives
  - **ORG:** indicates the beginning of the offset address

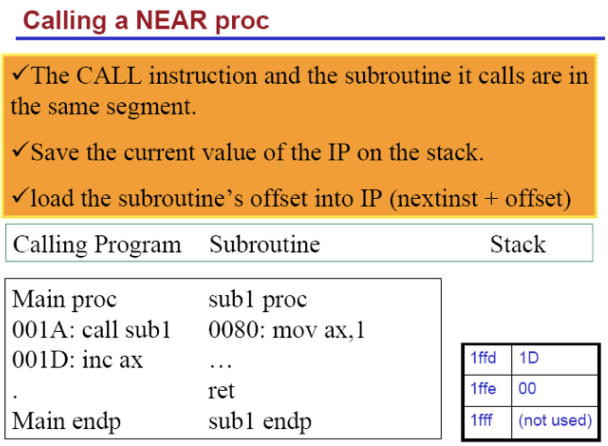

Figure 30:

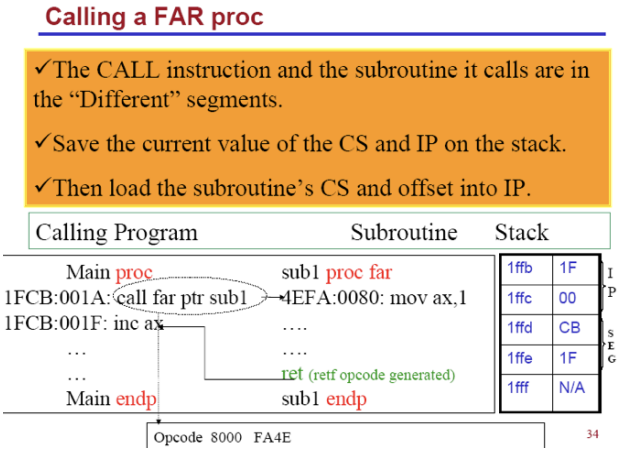

Figure 31:

    - E.g., `ORG 10H`
  - Define variables:
    - **DB:** allocate byte-size chunks
      - E.g., `x DB 12 | y DB 23H,48H | Z DB 'Good Morning!' | str DB "I'm good!"`
      - **DW, DD, DQ**
    - **EQU:** define a constant
      - E.g., `NUM EQU 234`
    - **DUP:** duplicate a given number of characters
    - E.g., `x DB 6 DUP(23H) | y DW 3 DUP(0FF10H)`
- Variables
  - For variables, they may have names
  - Variable names have three attributes: **Segment value, Offset address, Type**: how a variable can be accessed (e.g., DB is byte-wise, DW is word-wise)
  - Get the segment value of a variable: Use **SEG** directive (E.g., `MOV AX, SEG luchyNum`)
  - Get the offset address of a variable Use **OFFSET** directive, or **LEA** instruction E.g., `MOV AX, OFFSET time`, or `LEA AX, time`
- Labels
  - Implicit: `AGAIN: ADD AX, 03423H`
  - Use **LABEL**: `AGAIN LABEL FAR \\ ADD AX 03423H`
  - Also have segment, offset, type (NEAR / FAR).
- PTR directive

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

JMP FAR PTR aLabel
```

- ADD dest, src. dest += src.
  - dest can be a register or in memory.
  - src can be a register, in memory or an immediate.
  - **NO MEM-TO-MEM**. Change ZF, SF, AF, CF, OF, PF.
- ADC dest, src. dest += src + CF

```
TITLE    PROG3-1A (EXE)  ADDING 5 BYTES
PAGE     60,132
.MODEL SMALL
.STACK 64
;-------------------------------
         .DATA
COUNT    EQU    05
DATA     DB     125,235,197,91,48
         ORG    0008H
SUM      DW     ?
;-------------------------------
         .CODE
MAIN     PROC   FAR
         MOV    AX,@DATA
         MOV    DS,AX
         MOV    CX,COUNT        ;CX is the loop counter
         MOV    SI,OFFSET DATA  ;SI is the data pointer
         MOV    AX,00           ;AX will hold the sum
BACK:    ADD    AL,[SI]         ;add the next byte to AL
         JNC    OVER            ;If no carry, continue
         INC    AH              ;else accumulate carry in AH
OVER:    INC    SI              ;increment data pointer
         DEC    CX              ;decrement loop counter
         JNZ    BACK            ;if not finished, go add next byte
         MOV    SUM,AX          ;store sum
         MOV    AH,4CH
         INT    21H             ;go back to DOS
MAIN     ENDP
         END    MAIN
```

Figure 32:

Addition of multi-byte numbers:

- `LOOP xxxx = DEC CX \\ JNZ xxxx`
- SUB dest, src. SBB dest, src (dest -= src + CF)
- MUL
  - byte vs byte: One implicit operand is AL, the other is the operand, result is stored in AX
  - word vs word: One implicit operand is AX, the other is the operand, result is stored in DX & AX
  - word vs byte: AL hold the byte, AH = 0, the word is the operand, result is stored in DX & AX
- DIV
  - byte / byte: Numerator in AL, clear AH; quotient is in AL, remainder in AH
  - word / word: Numerator in AX, clear DX; ; quotient is in AX, remainder in DX
  - word / byte: Numerator in AX; quotient is in AL (max 0FFH), remainder in AH

```
TITLE     PROG3-2 (EXE)  MULTIWORD ADDITION
PAGE      60,132
.MODEL SMALL
.STACK 64
;-----------------------------------------
          .DATA
DATA1     DQ      548FB9963CE7H
          ORG     0010H
DATA2     DQ      3FCD4FA23B8DH
          ORG     0020H
DATA3     DQ      ?
;-----------------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          CLC                             ;clear carry before first addition
          MOV     SI,OFFSET DATA1         ;SI is pointer for operand1
          MOV     DI,OFFSET DATA2         ;DI is pointer for operand2
          MOV     BX,OFFSET DATA3         ;BX is pointer for the sum
          MOV     CX,04                   ;CX is the loop counter
BACK:     MOV     AX,[SI]                 ;move the first operand to AX
          ADC     AX,[DI]                 ;add the second operand to AX
          MOV     [BX],AX                 ;store the sum
          INC     SI                      ;point to next word of operand1
          INC     SI
          INC     DI                      ;point to next word of operand2
          INC     DI
          INC     BX                      ;point to next word of sum
          INC     BX
          LOOP    BACK                    ;if not finished, continue adding
          MOV     AH,4CH
          INT     21H                     ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Figure 33:

```
DATA_A    DD  62562FAH
DATA_B    DD  412963BH
RESULT    DD  ?
...       ... ...
          MOV   AX,WORD PTR DATA_A
          SUB   AX,WORD PTR DATA_B
          MOV   WORD PTR RESULT,AX
          MOV   AX,WORD PTR DATA_A +2
          SBB   AX,WORD PTR DATA_B +2
          MOV   WORD PTR RESULT+2,AX
```

Figure 34:

```
MOV   AL,DATA1
MOV   BL,DATA2
MUL   BL
MOV   RESULT,AX
```

```
MOV   AL,DATA1
MOV   SI,OFFSET DATA2
MUL   BYTE PTR [SI]
MOV   RESULT,AX
```

```
DATA3     DW    2378H
DATA4     DW    2F79H
RESULT1   DW    2 DUP(?)

MOV   AX,DATA3
MUL   DATA4
MOV   RESULT1,AX
MOV   RESULT1+2,DX
```

```
DATA5     DB    6BH
DATA6     DW    12C3H
RESULT3   DW    2 DUP(?)

MOV   AL,DATA5
SUB   AH,AH
MUL   DATA6
MOV   BX,OFFSET RESULT3
MOV   [BX],AX
MOV   [BX]+2,DX
```

Figure 35:

  - double-word / word: Numerator in DX, AX; quotient is in AX (max 0FFFFH), remainder in DX. Denominator can be in a register or in memory

```
MOV     AL,DATA7
SUB     AH,AH
DIV     10
```

```
MOV     AX,10050
SUB     DX,DX
MOV     BX,100
DIV     BX
MOV     QOUT2,AX
MOV     REMAIND2,DX
```

```
MOV     AX,2055
MOV     CL,100
DIV     CL
MOV     QUO,AL
MOV     REMI,AH
```

```
DATA1   DD      105432
DATA2   DW      10000
QUOT    DW      ?
REMAIN  DW      ?

MOV     AX,WORD PTR DATA1
MOV     DX,WORD PTR DATA1+2
DIV     DATA2
MOV     QUOT,AX
MOV     REMAIN,DX
```

Figure 36:

- XLAT

```
SQUR_TABLE DB 0,1,4,9,16,25,36,49
MOV BX, OFFSET SQUR_TABLE
MOV AL, 05 ; retrieve 6th element
XLAT       ; put in AL
```

- Logic: AND/OR/XOR/NOT
  - SHR dest, times. copy to CL when times > 1.
  - SHL dest, times
  - ROR dest, times, RCR dest, times (MSB -> LSB -> CF -> ...)
  - ROL dest, times, RCL dest, times (CF <- MSB <- LSB <- ...)

Example: BCD conversion

```
asc  DB  '3'
unpack DB  ?
-------------------------
MOV AH, asc
AND AH, 0Fh
MOV unpack, AH

; combine two unpacked into one packed
asc DB  '23'
unpack DB  ?
-------------------------
MOV AH, asc
MOV AL, asc+1
AND AX, 0F0Fh
MOV CL, 4
SHL AH, CL
OR AH, AL
MOV unpack, AH
```

- CMP dest, src

Table 3-3: Flag Settings for Compare Instruction

Compare operands	CF	ZF
destination > source	0	0
destination = source	0	1
destination < source	1	0

Figure 37:

Jump Based on Unsigned Comparison

These flags are based on unsigned comparison

Mnemonic	Description	Flags/Registers
JA	Jump if above op1>op2	CF = 0 and ZF = 0
JNBE	Jump if not below or equal op1 not <= op2	CF = 0 and ZF = 0
JAE	Jump if above or equal op1>=op2	CF = 0
JNB	Jump if not below op1 not <opp2	CF = 0
JB	Jump if below op1<op2	CF = 1
JNAE	Jump if not above nor equal op1< op2	CF = 1
JBE	Jump if below or equal op1 <= op2	CF = 1 or ZF = 1
JNA	Jump if not above op1 <= op2	CF = 1 or ZF = 1

Figure 38:

Jump Based on Signed Comparison

These flags are based on signed comparison

Mnemonic	Description	Flags/Registers
JG	Jump if GREATER op1>op2	SF = OF AND ZF = 0
JNLE	Jump if not LESS THAN or equal op1>op2	SF = OF AND ZF = 0
JGE	Jump if GREATER THAN or equal op1>=op2	SF = OF
JNL	Jump if not LESS THAN op1>=op2	SF = OF
JL	Jump if LESS THAN op1<op2	SF <> OF
JNGE	Jump if not GREATER THAN nor equal op1<op2	SF <> OF
JLE	Jump if LESS THAN or equal op1 <= op2	ZF = 1 OR SF <> OF
JNG	Jump if NOT GREATER THAN op1 <= op2	ZF = 1 OR SF <> OF

Figure 39:

Example: Given the ASCII table, write an algorithm to convert lowercase letters in a string into uppercase letters and implement your algorithm using 86 assembly language.

```
.MODEL SMALL
.STACK 64
;-------------------------------------
        .DATA
DATA1   DB      'mY NAME is jOe'
        ORG     0020H
DATA2   DB      14 DUP(?)
;-------------------------------------
        .CODE
MAIN    PROC    FAR
        MOV     AX,@DATA
        MOV     DS,AX
        MOV     SI,OFFSET DATA1         ;SI points to original data
        MOV     BX,OFFSET DATA2         ;BX points to uppercase data
        MOV     CX,14                   ;CX is loop counter
BACK:   MOV     AL,[SI]                 ;get next character
        CMP     AL,61H                  ;if less than 'a'
        JB      OVER                    ;then no need to convert
        CMP     AL,7AH                  ;if greater than 'z'
        JA      OVER                    ;then no need to convert
        AND     AL,11011111B            ;mask d5 to convert to uppercase
OVER:   MOV     [BX],AL                 ;store uppercase character
        INC     SI                      ;increment pointer to original
        INC     BX                      ;increment pointer to uppercase data
        LOOP    BACK                    ;continue looping if CX > 0
        MOV     AH,4CH
        INT     21H                     ;go back to DOS
MAIN    ENDP
        END     MAIN
```

Figure 40:

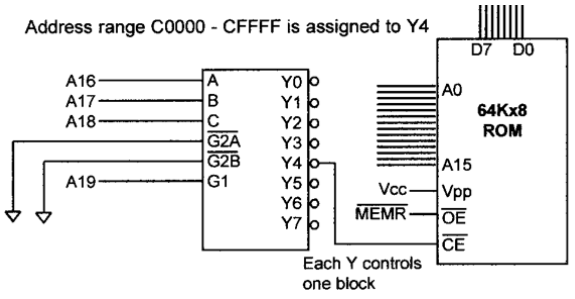

Figure 41:

8086 Memory and IO

Memory Address decoding

Examine address decoding using logic gates and 74LS138 decoder chips

- Absolute address decoding: all address lines are decoded
- Linear select decoding: only selected lines are decoded. Cheap. But with aliases.

An example diagram for an example problem (for reference).

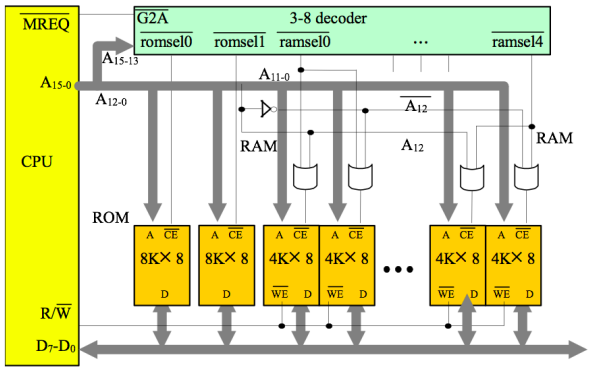

Figure 42:

Example 2. Assume one computer system needs 512 byte RAM and 512 byte ROM. If RAM is built with 128×8 memory chips and ROM is built with 512×8 memory chips, please specify the address range of each memory chip. Given that RAM chips need CS~ and WE~ control signals, ROM chips need only CS~ control signal, and the CPU has 16 address pins (A15~A0), 8 data pins (D7~D0) and R/W~ and MREQ~ control signals, draw the connection between the CPU and the memory.

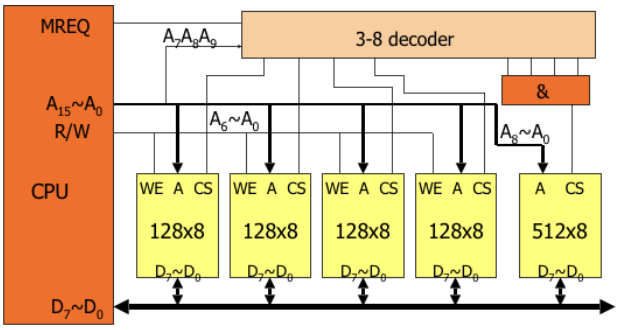

Figure 43:

Data Integrity

Checksum byte

Add all bytes together and drop all carries. Take the 2's complement of the sum. Store the checksum byte together with data. *Check the integrity by adding data and the checksum together.*

Parity bit

- even parity: if the number of 1s in the series of bits is odd, then the parity bit is set to 1; otherwise, set to 0, making the total number of 1s even (Data + the parity bit)
- odd parity.
- PF in 8086 use odd parity.

Memory organization in 8086

Even and odd banks

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

Figure 44:

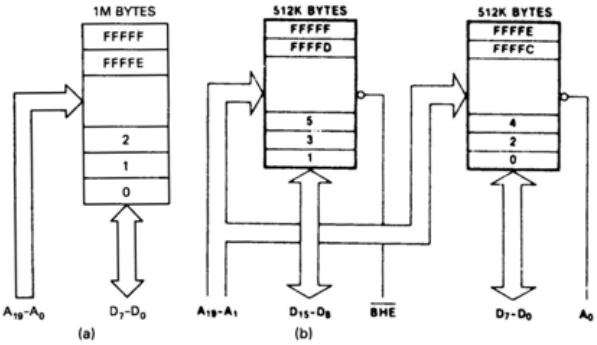

Figure 45:

- Aligned Word-memory operations: Both the high and low banks are accessed at the same time. Both A_0 and $\overline{BHE}$ are set to 0. This 16-bit word is transferred over the complete data bus D_0 through D_{15} in just one bus cycle.
- Misaligned Word-memory operations: Two bus cycles are needed. During the first bus cycle, the byte of the word located at address $X + 1$ in the high bank is accessed over D_8 through D_{15}. Even though the data transfer uses data lines D_8

through D_{15}, to the processor it is the low byte of the addressed data word. In the second memory bus cycle, the even byte located at $X + 2$ in the low bank is accessed over bus lines D_0 through D_7.

I/O in x86 family

- X86 microprocessors have an I/O space in addition to memory space
- 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
- Also referred to as peripheral I/O or isolated I/O
- Direct I/O instructions: `IN AL, port#;` `OUT port#, AL`. `port#` ranges from 0h to 0ffh, 256 ports in total.
- Indirect I/O instructions:

```
MOV DX, port#;   65536 ports in total
IN AL, DX
```

No segment concept for port addresses.

- 16-bit: `IN AX, port#`.

Output port design

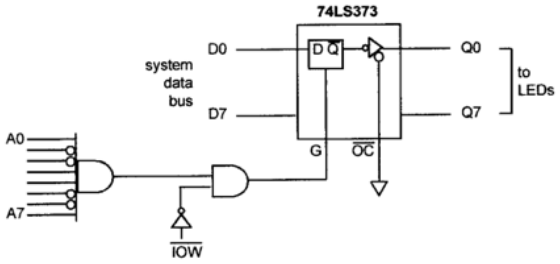

Figure 46:

Input port design

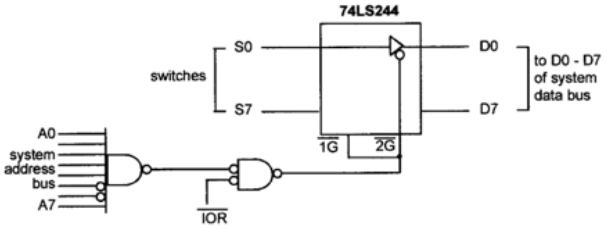

Figure 47: