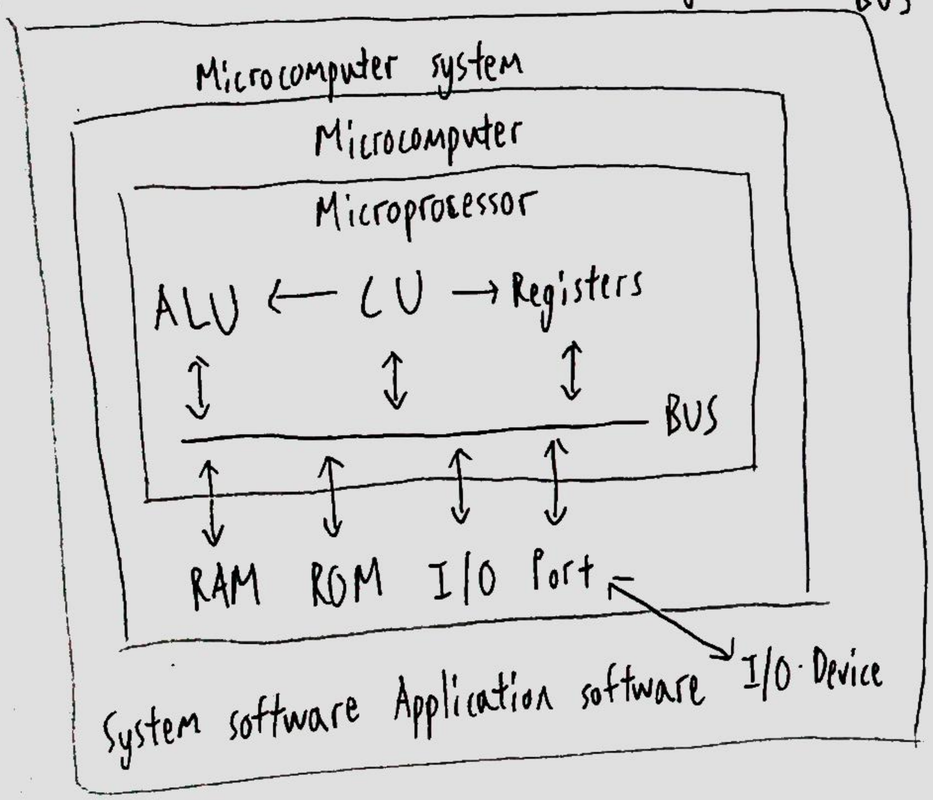


ALU
I/O ports
memory
Application softwares
CU ← microprocessor ← Microcomputer
softwares → system softwares
Registers
BUS
Microcomputer System → I/O devices

Computer Organization

Von Neumann Architecture

Central Processing Unit (CPU)
Input → - Control Unit (CU) → Output
- Arithmetic / Logic Unit (ALU)

executes

Key concepts

Memory Unit ← read/write stored ← Instruction

in sequential order

sequential execution

stored

Data ← Abstraction

contents of memory are addressable

Instructions and data are stored in same read-write memory

sequential execution model

Components Overview

Memory

- control each module and data/address buses
- Control Bus → bi~~uni~~directional
  - command signal (CPU → memory)
  - state signal memory → CPU
- Types
  - Address Bus
    - Width
    - total no. memory locations addressable $2^{\wedge}$
    - Unidirectional (read)
  - Data Bus → Bidirectional (read/write)
    - width → same as registers
    - word
- communication pathway (CPU ↔ memory I/O BUS)
- Structures
  - single-bus → poor throughput
    - simple
  - CPU-Central Dual-Bus → poor CPU performance
    - efficient data transfer
    - same address space for memory and I/O
  - Memory-Central Dual-Bus → High CPU performance
    - high throughput
- Concepts
  - Bit (b) - binary digit (0 or 1)
  - Byte (B)
    - 8 bits
    - smallest addressable unit
  - Nibble 4 bits
  - Word
    - no. of bits a CPU can process one time (depends on data bus)
- Memory Module
  - a set of locations defined by sequentially numbered addresses

I/O

- memory-mapped I/O
  - use same instructions to memory and I/O
  - CPU unaware which type of data
- Isolated I/O
  - separate address space
  - different sets of accessing instructions

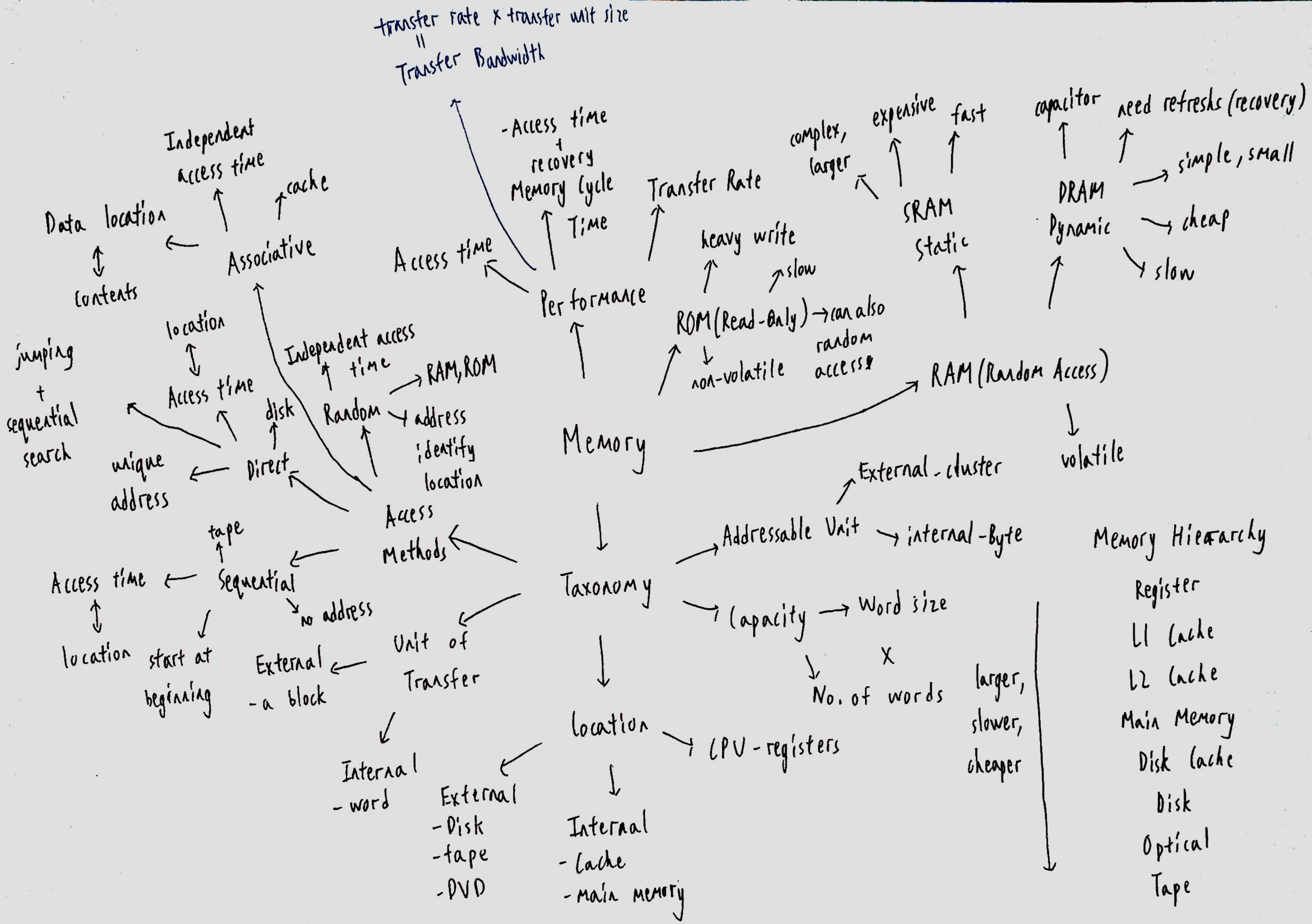
CPU

- Core
  - ALU
    - logic functions
    - 2 inputs
    - multifunctional calculators
    - arithmetic functions
  - Registers
    - results stored in
  - CU
    - works under instructions (pre-defined code which defines a specific operation)
    - instructor decoder
      - decode instructions
      - produce control signals
    - program counter
      - point to address of next instruction
- Clock

Instruction Architecture set

ISA

- Reduced Instruction Set Computers
  - simple hardware
  - fast
  - easy to pipeline
  - Fixed instruction time
  - Fixed size (1 word)
- Complex Instruction Set Computers CISC
  - backward compatibles
  - variable formats
  - variable length


transfer rate x transfer unit size
=
Transfer Bandwidth
Memory
Performance
Access time
-Access time
+
recovery
Memory Cycle
Time
Transfer Rate
heavy write
slow
ROM(Read-Only)
non-volatile
can also
random
access
complex,
larger
expensive
fast
SRAM
Static
capacitor
need refreshs (recovery)
DRAM
Dynamic
simple, small
cheap
slow
RAM(Random Access)
volatile
Addressable Unit
internal-Byte
External-cluster
Taxonomy
Capacity
Word size
X
No. of words
location
CPU-registers
Internal
External
-Cache
-main memory
Unit of
Transfer
Internal
-word
External
-a block
Access
Methods
Sequential
Access time
location
start at
beginning
no address
tape
Direct
unique
address
Random
address
identify
location
RAM, ROM
Independent access
time
Access time
location
disk
jumping
+
sequential
search
Associative
Data location
contents
Independent
access time
cache
Memory Hierarchy
Register
L1 Cache
L2 Cache
Main Memory
Disk Cache
Disk
Optical
Tape
larger,
slower,
cheaper

Input/Output Module

- interface between CPU/memory and I/O devices

Problems

- External Devices
  - machine readable
  - Human readable
  - slower than CPU
  - different operation logic
- Wide variety of peripherals
  - Hide/reveal device properties
  - single/multiple devices
  - Decisions → control device directly / CPU control
- communication
  - CPU
  - Device

Function

- Control/Timing
- Communication
- Data buffering
- error detection

Techniques

Programmed

- CPU executes program
  - Def to directly control → Transfer data
  - → read/write commands to I/O module
  - checking device status
- commands
  - check status
  - control
    - tell command what to do
  - CPU issues address
    - identify module
  - performance and cost
    - simple,
    - waste of CPU time
- Cycle
  1) CPU → I/O module (requests)
  2) I/O module → status → CPU
  3) I/O module → data → CPU
  4) CPU → data → memory

Interrupt Driven

- both need CPU read/write
- Cycle
  1) CPU → command → I/O module
  2) I/O module do works
  3) I/O module → interrupts → CPU (when data is ready)
  4) CPU deal with event
     a) save context
     b) do event
     c) recover to previous work

Designs Issues

- Identify Interrupting module
  - Daisy chain / hardware poll
    - 1 interrupt line
    - bus master (module must claim bus to send)
  - software poll + a line for a module
    - share one interrupt request line
    - CPU ask module in turn
    - CPU clears status
- Locating Handler Programs
  - multiple interrupts → set priorities
    - nesting of interrupts
  - use general handler program

Direct memory access DMA

- DMA controller
  - read/write
  - device address
  - starting address of memory
  - amount of data
- Cycle
  1) CPU
  2) CPU do other work
  3) DMA controller → interrupts (work done) → CPU
- DMA Configurations
  - simple, performance ↓
    - detached DMA
      - single bus
      - integrated DMA
      - 1 transfer, bus once
    - use general handler program
  - complex ↓, performance ↑
    - detached DMA
      - separate I/O bus
      - 1 transfer, bus once
      - pointer to address interrupt vector
      - * performance ↑

- DMA Transfer Cycle Stealing
  - DMA may takes over bus for a cycle
  - each transfer, twice bus

Memory Organization

*when n_{CPU} > CPU addressable unit(1B)

↓ throw away ←

$\log_2\left(\frac{\text{data bus width (Byte)}}{\text{addresable unit (Byte)}}\right)$

↓ lower bits

$\log_2$(total capacity of $N_{CPU} \times n_{CPU}$)

=

address line no.

↑

Word extension

$N_{chip} = N_{CPU}$

$n_{chip} < n_{CPU}$

use $\log_2\left(\frac{n_{CPU}}{n_{chip}}\right)$ no. of address line → $n \to 2^n$ decoder

↑ high bits

connect to $\frac{n_{CPU}}{n_{chip}}$ memory array

use $\frac{n_{CPU}}{n_{chip}}$ no. of parallel memory array

one memory array use data bus at a time

→ Bit Extension

$N_{chip} \times n_{chip}$

Vs

$N_{CPU} \times n_{CPU}$

$n_{chip} = n_{CPU}$

$N_{chip} < N_{CPU}$

use $\frac{N_{CPU}}{N_{chip}}$ no. of chip in parallel, one chip gives

n_{chip} bit → $\frac{N_{CPU}}{N_{chip}} \times n_{chip}$ bit

=

data bus width

↓ memory array

bit line

col decoder $n \to 2^n$

row decoder $n \to 2^n$

word line

one bit per cell

↓ ↓ ↓ ↓ data bus

$\overline{E_x}$ block enable

$\overline{WE}$ write enable

↓ Chip

Notation

$N_{chip} \times n_{chip}$

↓ no. of n-bit words

↓ chip word width

=

data bus width

form a memory array ←

permanent storage

↓ Characteristics

↓ nonvolatile

Read only Memory

ROM

↓ Types

Programable

written during manufacture

-expensive

Read mostly

↓ Erasable Programable EPROM

-erased by UV light

↑ Flash memory

Electrically Erasable EEPROM

- $T_{write} > T_{read}$

↑ Applications

↓ function tables

↑ System programs

BIOS

↑ Library subroutines

IP, DS, SS, ES, instruction queue cleared

RESET → CS = FFFFH
- reset CPU

NMI
- non-maskable interrupt

INTA
- interrupt acknowledgement

INTR
- interrupt request

HLDA
- hold request acknowledgement

HOLD
- input, hold bus request

DT/~R
- direction for Data Transceivers

~DEN
- enable data transceivers

~BHE
- 0, AD8-AD15 used

ALE
- latch enable signal

READY
- memory/I/O ready

M/~IO
- memory/I/O

~WR
- write

~RD
- read

Control Signals of chip interface

80X86 microprocessor

minimum mode (single CPU)

MN/~MX

maximum mode
- multiple CPUs

ADi — address latch

Address and Data bus

ALE 1 input → output (address)

ALE 0 output held
*used as data bus

address latch ↑used as

E/L	D	Q	$\bar{Q}$
0	x	Qprev	$\bar{Q}$prev
1	0	0	1
1	1	1	0

D-Latch

~DEN DT/~R

Data Transceivers

Data bus

Bus Cycle
*at least 4 cycles

1) Address Output

2) start output
- write enable
- read enable

3) Unknown cycle : output

4) $\overline{RD}$ and $\overline{WR}$ switch

Write
Read
wait for I/O/memory ready

Trap Flag
- set to execute instruction at a time

Direction Flag DF
- direction of string operations

3 Control Flags

IF Interrupt Flag
- set to enable external maskable interrupt

*BCD-related arithmetic
- carry from d3 to d4
AF Auxiliary Carry Flag

*signed
- sign bit error
OF Overflow Flag

ZF zero flag

PF Parity Flag low-8bits
- 1 when no. of 1's is even

SF sign flag
- sign bit

6 conditional flags

CF Carry Flag
- carry out from d7 (8-bit) or d15 (16-bit)
*for unsigned
1) 2 unsigned added, 8bit→9bit
2) 2 unsigned - small - big

加法進位
減法借位

1×16-bit flag register (9/11 bits used)

Execution Unit EU
- executes instructions

ALU

AH High Byte
AL low Byte
AX Accumulator

Components

4×16-bit general registers
- Data DX
- Base BX
- Count CX

2×16-bit pointer registers
- Stack Pointer SP
- Base Pointer BP

2×16-bit index registers
- source index SI
- Destination Index DI

4×16-bit segment registers
- code CS
- data DS
- extra ES
- SS stack

Bus interface unit BIU
- access memory and peripherals

address relocation

bus control

Data transfer between CPU and memory/I/O devices

operand fetch and storage

instruction queuing

instruction fetch

Components

1×6B instruction queue

1×16-bit instruction pointer IP

1×20-bit address adder

CS<<4+IP

Internal Structure

8086 Memory Management

- how CPU access data
  - Addressing mode
    - Register → data in registers, all except CS, IP
      - * Destination and source should have same size
    - Immediate - constant
      - can't move to segment registers → if want to move constant → segment register
        1) constant → register
        2) register → segment register
      - embedded in instructions
      - no need to access memory
    - Register Indirect
      - Data in memory
      - address in register
      - need access memory
      - registers → SI, DI, BX
    - Direct
      - data in memory, address given
      - effective / offset address in DS (default)
      - need access memory
    - Based Indexed Relative
      - combine based + indexed, one base DS:BX + one index SS:BP
      - need access memory
      - segment overrides → MOV AX, CS:[BP] = CS:BP (default: SS:BP)
      - offset registers are used with default segment registers
      - MOV CL, [BX][DI]+8 = DS:BX+DI+8
    - Indexed Relative
      - data in memory
      - address calculated with DI, SI (index)
      - default: DS
      - MOV DX, [SI]+5 = DS:SI+5
    - Based Relative
      - Data in memory
      - Address calculated with BX, BP (base)
      - default segment DS:BX; SS:BP
      - MOV CX, [BX]+10 = DS:BX+10
  - Address
    - Physical
      - actual address in physical location or address bus
      - conversion
        1) segment value * 16 (<<4)
        2) + offset address
        * one logical → one physical
    - Logical
      - segment value + offset address
        - determines beginning of segment
        - relative location within segment
      - segment overlapping
- program consists of 3 segments
  - code segment
    - instructions
    - logical address = CS:IP
      - if instructions physically outside of range, change CS
    - BIOS
      - Basic input-output system
      - inside ROM
        - PC → control → DOS
        - Load DOS from disk → RAM
        - test all devices connected
  - data segment
    - information
    - logical address = DS:offset
    - Endian
      - Little: low byte low memory
      - Big: High Byte low memory
  - extra segment
    - for string operation
    - ES:offset
  - stack segment
    - temporary information
    - logical address = SS:SP
      - SP -= 2(x) push
      - constant for push/pop
      - 2500
      - most registers in CPU can stored in stack
      - upper addresses → downward → lower addresses
      - push / pop
        - 16-bit operation
    - max 64KB
    - CS + IP
      - 0000 ~ FFFF
      - wrap-around
        - when CS + IP > FFFFFH
        - modulo

Editor Program
↓ .asm
Assembler Program
↳ .lst ← ↓ .obj → .crf
↓
Linker Program ← other .obj
↓ ↳ .map
.exe

- no stack segment
stack in end area ↑
* data first, use jump to code
data segment inside code segment ↑

MOV AX, @DATA
MOV DS, AX

only have code segment ↑
data/code start at 0100H ↑

- DS/ES must be initialized
- give CS, SS proper values

- compact code ≤64KB
.COM file ← * 1 segment

DOS ↑

1) edit program ↑
2) assemble ← assemble steps ← Assembly language → control transfer instructions
↓
3) link

PROC
...
RET → return control, restore CS:IP from stack
ENDP ↑

- used to call subroutine
CALL ↑

Jumps → unconditional JMP (* default NEAR)
- according to flag register)
conditional (* default SHORT)

control transfer instructions → Range → SHORT (intrasegment) - change IP (1B range)
→ NEAR (intrasegment) - change IP (2B range)
↓ FAR (intersegment) - change CS, IP

label SEGMENT
...
label ENDS ↑
full segment definition
- multiple segments ← segments ← Structure → A series of statements

* only 3 segments
simplified segment definition
↓ .STACK ← - stack segment
↓ .DATA - data segment
↓ .CODE → ENDP
↑ PROC (FAR/NEAR)
- code segment
* entry must be FAR

instructions
- do the real work
- translated into machine code

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

- data ↔ constant
EQU (equate)

DW (define word) - allocate word size memory
DB (define byte) - allocate byte size memory
DUP (duplicate)

directives (pseudo-instructions)
- give instructions to assembler (how to translate)
→ ORG (origin) - beginning of offset address
↓ Model - size of memory model
→ PTR - temporarily change type / range of variable / label

Label
Type - range of jumps ←
↑
Data Types
↓ variable
↓ Type - DB/DW...
↓ logical address → segment value - use SEG to get segment value
↓ offset address - use OFFSET/LEA to get offset address

Assembly Language

Operations

Logic Instructions

- NOT operand → don't change flags
- XOR dest, src
- OR dest, src
- AND dest, src
  - register/memory
  - immediate/r/m
- → - AF undetermined
  - clear CF, OF

Logical SHIFT

- SHL dest, times
  - CF ← MSB ← ... ← LSB ← 0
- SHR dest, times
  - 0 → MSB → ... → LSB → CF
  - Times > 1
    - MOV CL, times
    - SHR xx, CL
  - Times = 1
    - SHR xx, 1

ROTATE

- ROL dest, times
  - CF ↑
  - MSB ← ... ← LSB
- ROR dest, times
  - MSB → ... → LSB
  - CF
- RCL
  - reverse of RCR
  - CF ↑
  - MSB → ... → LSB
- RCR dest, times

Arithmetic Operations

- unsigned Subtraction
  - SUB dest, src; dest = dest - src
  - SBB dest, sr; dest = dest - src - CF
    - for multi-byte
    - CF = 1 → negative result
    - CF = 0 → positive result
  - DEC dest
    - x immediate
    - x change CF
  - INC dest
- Unsigned Addition
  - ADD dest, src; dest = dest + src
    - * no memory to memory
    - register/memory ↑
    - register/memory/immediate
  - ADC dest, src
    - dest = dest + src + CF
    - for multi-byte
- unsigned multiplication
  - MUL operand
    - word x word
      - implicit AX
      - result AX, DX
    - word x byte
      - AL hold B,
      - AH = 0
      - result AX, DX
    - B x B
      - implicit AL
      - result AX
- unsigned division
  - DIV denominator
  - word / word
    - xume DX, AX
    - q AX, r DX
  - double word / word
    - xume AX, clear DX
    - q AX, r DX
  - word / Byte
    - xume AX,
    - q AL, r AH
  - B / B
    - numerator AL
    - clear AH
    - q AL, r AH

COMPARE

- CF = 1, ZF = 0
  - dest < src
- ZF = 1
  - dest = src
- dest > src CF = 0, ZF = 0
- flag change as (dest - src)
- CMP dest, src
- unsigned
- signed
  - OF = SF, ZF = 0
    - dest > src
  - ZF = 1
    - dest = src
  - dest < src
    - OF != SF

Binary Coded Decimal BCD

- unpacked
  - 1B = 1 no.
- packed
  - 4bits = 1 no.

ASCII → unpacked BCD

- remove higher 4 bits

```
MOV AH, asc
AND AH, 0FH
MOV unpack, AH
```

ASCII → packed BCD

```
MOV AH, asc
MOV AL, asc+1
AND AX, 0F0FH
MOV CL, 4
SHL AH, CL
OR AH, AL
MOV unpack, AH
```

1) ASCII → unpacked BCD
2) 2 unpacked —combine→ one packed

input port
- use tri-state buffer to connect to system data bus

output port
- latch data from CPU

one bus cycle: odd/even banks at the same time

aligned (even address)

word-memory operations

Misaligned (odd address)

2 bus cycles

second cycle (even bank)
$D_7 - D_0$ → high byte

first cycle (odd bank)
$D_{15} \sim D_8$ → low byte

* even and odd banks memory organization

odd bank — $D_{15} - D_8$

even bank — $D_7 - D_0$

$A_{19} - A_1$

$\overline{BHE}$

A_0

BHE	A0	
0	0	even word
0	1	odd byte
1	0	even byte

port range 0000h~ffffh
65536 port

Indirect
* use register to store addr

I/O instructions
* no segment concept for port addresses

* immediate
Direct
port range 00h ~ DFH (256)

Memory and I/O in 8086

Data Integrity

parity bit (for DRAM)
- odd parity (reverse of even parity)
- even parity
  - 0 if no. of 1's is even
  - 1 if no. of 1's is odd

Checksum Byte (for ROM)
- store the 2's complement of sum of other bytes

to check (add sum of other bytes with checksum Byte)
- =0 → data safe
- ≠0 → data corrupted

memory address decoding

- decode memory address to locate specific memory chip

linear select decoding
- only selected lines are decoded
  - with aliases (same memory unit with multiple addresses) — cheap
- absolute address decoding
  - all address line are decoded

$\log_2(N_{mem})$ directly connected to chips

address line = $\log_2(N_{CPU})$

$N_{mem} \times n_{mem}$
$N_{CPU} \times n_{CPU}$

additional lines → used to distinguish chips

when $N_{mem} > N_{CPU}$
- several line from decoder point to the same chip

when $N_{mem} < N_{CPU}$
use $\frac{N_{CPU}}{N_{mem}}$ chips
use higher address line combine with decoder line to locate chips

8255 PPI Chip

Programmable Parallel Interface

Mode 0 (simple I/O) → CPU directly read/write using IN/OUT
→ any port can be I/O independently
→ no handshaking

Mode 1 (strobe I/O)

Timing in mode 1 input:
1) device —data→ $PA_0 \sim PA_7$, activates $\sim STB_A$ —latch data→ Port A
2) 8255 activates IBF_A
3) IBF_A, $\sim STB_A$, $INTE_A$ all high, 8255 activates $INTR_A$ → CPU
4) CPU —read→ PA, clear $INTR_A$, read finish, clear IBF_A

PC_6, PC_7 can used as separate I/O
$PC_0 \sim PC_2$ ↑use PB
$PC_3 \sim PC_5$ ↑use PA

input handshaking:
- request interrupt to CPU
- $INTR_A$ (output) ↑interrupt
- 1: allow input signals
- INTE -internal bit ← $PC_4(A)$, $PC_2(B)$
- signal device input latch full
- IBF (output) Input Buffer Full
- $\sim STB$ (input)
- load data into port latch

input data not latched
output data latched

output handshaking:
- $\sim OBF$ (output) output signals
- $PC_6(A)$ → INTE ← $PC_2(B)$
- output buffer full (data latched)
- INTR (output) - interrupt CPU
- $\sim ACK$ (input) - acknowledge device took data
- PC_4, PC_5 separate I/O

Timing in mode 1 output:
1) $INTR_A$ active —CPU write→ PA
2) $\sim OBF_A$ activate ↓inform device
3) device get data, $\sim ACK_A$ PB
4) $\sim OBF_A$ off, $\sim ACK_A$, $INTE_A$ all high
8255 —$INTR_A$→ CPU

mode 1 output handshaking → PA → $PC_3 \sim PC_5$

Internal Structure and PINs:
- Read/Write control logic
  - RESET ← -clear CR, set all port as input
  - $\sim CS$ chip select
  - A_1, A_0 -port selection
    - 11: CR
    - 10: PC
    - 01: PB
    - 00: PA
- 8-bit Control Register CR -used to setup the chip
- 3 data ports
  - Port A $PA_0 \sim PA_7$
  - Port B $PB_0 \sim PB_7$ — can be programmed all as input/output
  - Port C $PC_0 \sim PC_7$
    - PCL -any bit can be programmed individually
    - PCU

Data bus buffer - interface between CPU and 8255
- Bidirectional
- tri-state
- 8-bit

unrelated | line select of PC | → 0: Reset, → 1: set

| 0 | * | x | * | D_3 | D_2 | D_1 | D_0 |

BSR mode
- output port: only PC
- can set/reset each line of PC individually

mode selection word

| D_7 | D_6 | D_5 | D_4 | D_3 | D_2 | D_1 | D_0 |

Group A control: $D_6 \sim D_5$ mode selection: 00 mode 0, 01 mode 1, 10/11 mode 2
- D_4: 1: input, 0: output
- D_3: PCU → 1: Input, 0: Output

Group B control:
- D_2 mode selection: 1 mode 1, 0 mode 0
- D_1: 1: input, 0: output
- D_0: PCL: 0: output, 1: input

D_7: 1: I/O mode, 0: BSR mode

interrupt when ready

↑

status stored in Port C

↑

directly check I/O device status →

Interruptions vs polling

↕

mode 1, vs mode 0

↑

8255 PPI Chip 2

↓

Mode 2 (Bidirectional Bus) → bidirectional handshake I/O

↘ both I/O data latched

↘ only PA can ^both I/O

↘ use handshake lines

$P_{C_3} \sim P_{C_7}$

↓

* $PC_0 \sim PC_2$ separate I/O line

~ STB_A

↑

$INTR_A$ ← signals → $\sim OBF_A$

IBF_A ↓ ↓

~ ACK_A

8253/4 Timer

Internal Structure

- Data bus buffer
  - D7~D0 Data bus
  - bi-directional 8-bit
  - tri-state
- Read/Write control logic
  - A1, A0
    - select counter/ control word register
  - ~CS
    - tied to a decoded address
- Control word register
  - D7 D6 D5 D4 D3 D2 D1 D0
  - SC1 | SC0 | RW1 | RW0 | M2 | M1 | M0 | BCD
  - BCD → 0: Binary, 1: BCD
  - M2 M1 M0 → mode
  - RW1 RW0 → R/W
    - 11 R/W LSB first, MSB next
    - 10 R/W MSB
    - 01 R/W LSB
    - 00 counter latch
      - 1) write control word to latch count
      - 2) read the latched count
  - SC1 SC0 → select counter
    - 11 → illegal (8253)
    - 10 → read-back (8254)
    - 01 → 1
    - 00 → 0
- 3 Independent Counters
  - input CLK → 1~65536 (Binary), 1~10000 (BCD)
  - input GATE (use to enable/disable counter)
  - output OUT (output frequency)
  - counting process → 16-bit count
    - decrement
    - Counter register CR
      - 0 → generate pulse to interrupt CPU
      - one CLK pulse
    - Counting Element CE (start count when GATE=1)
      - check GATE - rising edge of CLK
      - count down - falling edge of CLK
      - output latch

Mode 0 (Interrupt on Terminal count)
- doesn't repeat automatically
- when GATE=0, pause, resume when GATE=1
- count start automatically

Mode 1 (Hardware Retriggerable One Shot)
- auto start counting new input
- when GATE=1 in middle of counting, repeat
- count start after GATE=1
- need GATE=1 to repeat
- new input need to wait for GATE=1

Mode 2 (Rate Generator)
- count start when GATE=1
- GATE=0, stop counting
- GATE=1, resume
- auto repeat
- new count start after current count finish

Mode 3 (Square Wave Rate Generator)
- auto repeat
- auto start
- initially output=high
- even count
  - -2 till 0
  - output=low, repeat
- odd count
  - first -1, then -2 till 0, output=0, reload
  - high (n+1)/2
  - low (n-1)/2

Mode 4 (software triggered strobe)
- count start automatically if GATE=1
- automatically repeat
- GATE=1, counting
- GATE=0, disable
- new count - count start next CLK
- if 2B, need 2 CLK
- initial high
- output go low after 1 CLK after 0

Mode 5 (hardware triggered strobe (retriggerable))
- auto repeat
- count start when GATE trigger
- not triggered, current half not affected
- triggered, start from new count on next CLK
- new count
- GATE=1, count
- GATE=0, output=1, if GATE=1 again, restart
- GATE trigger, repeat
- new count - need trigger
- output → initial high
- go low after 1 CLK after 0

Interrupts in 8086

Interrupt Vector Table IVT first 1KB to store
- 256 types INT 00~0FFH
- *4B → PA of Interrupt vector = CS, IP
  - points to entrance address of ISR Interrupt Service Routine

Hardware (external) ← Interrupts in 8086 → Software (internal)
- INT xx (0~255) instruction → 256 Interrupts

Hardware (external)
- non-maskable (from NMI)
- Maskable (from INTR)

non-maskable (from NMI)
- Properties
  - Type: INT 02
  - trigger: NMI pin
    - input
    - rising edge
    - 2 cycle high activate
- Reasons
  - interrupt from co-CPU
  - RAM parity check error

Procedure
1) CPU checks NMI
2) INT 02 interrupt generates
3) ISR executes
4) CPU returns

Maskable (from INTR)
- Properties
  - trigger: INTR pin
    - input
    - high active
    - no predefined type
  - IF
    - 1: enable
    - 0: disable
  - STI: set
  - CLI: clear
- Reasons
  - interrupt requests of external I/O

Procedures
1) I/O interrupt → CPU
2) CPU check INTR on last cycle of instruction if INTR: high + IF=1, responds
3) CPU
   - 2 negative pulses
   - ~INTA pin
   - I/O device
4) I/O
   - Type N → Data bus
   - *on second ~INTA
5) CPU reads N push FR, CS, IP
6) Clear IF, TF load ISR
7) end of ISR IRET pop IP, CS, FR
8) CPU returns

8259 Programmable Interrupt Controller PIC
- IRR Interrupt request register 8-bit
- IMR Interrupt mask register 8-bit
  - ↑clears ~INTA
- interrupt requests → priority resolver → various priority schemes
- IS In Service register ← clears ← EOI
- Properties
  - 64 interrupt inputs
  - can mask interrupt

Determining Priority
- Hardware Checking
  - daisy chain logic
- Software Polling
  - first check has higher priority
- Vectored interrupt controller

Software (internal)
- INT xx (0~255) instruction → 256 Interrupts
  - 0~4: dedicated interrupts → 04 signed number overflow
    - 00 divide error
    - 01 single step
      - if TF=1, for debugging
    - 03 breakpoint - debugging
  - 5~31: system use
    - 10H~1FH BIOS
  - 32~255: users
    - 20H~3FH DOS

CPU ← Data Bus ← buffer ← interrupts
- buffer ← IOR

Handle of interrupt
1) push CS: IP and FR
2) load CS: IP with N
   - CS = mem[4*N+2]
   - IP = mem[4*N]
3) execute ISR, at the end IRET
4) pop CS: IP and FR

Priority

INT > NMI > INTR

Interrupt Nesting
- high priority interrupts can interrupt lower priority interrupts