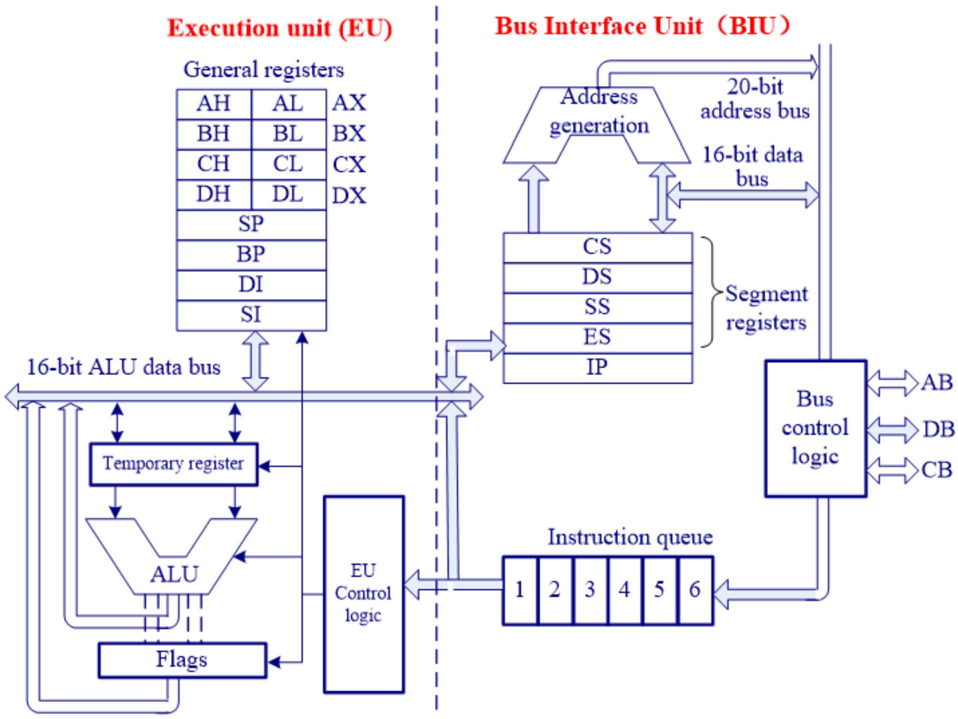
3 80X86 Microprocessor

3.1 Internal Structure of 80x86

Evolution of x86 Family

- 8086 (1978):
  - first *16-bit* microprocessor.
  - *20-bit address data bus*, i.e. $2^{20} = 1\ \mathrm{MB}$ memory.
  - segment-base memory management（基于段的内存管理）.
  - First *pipelined* microprocessor.
- 8088
  - 16-bit internal, 8-bit external data bus
  - Fit in the 8-bit world.
- 80286, 80386, 80486, ...

Two sections work simultaneously.



- **Bus interface unit (BIU)**: accesses memory and peripheral (I/O devices).
  - Two types of data: instruction, operand（操作数）.
  - need to fetch instructions and store instructions in queue.
  - Consists of:
    - 16-bit segment registers（段寄存器）: CS, DS, ES, SS;
    - 16-bit instruction pointer (program counter): IP (PC);
    - 20-bit address adder;
    - 6-byte instruction queue.
  - While EU is executing an instruction, the BIU will fetch the next several instruction from the memory and store them in the instruction queue.
- **Execution unit (EU)**: executes instructions previously fetched.

- Take in charge of instruction execution.
- Consists of:
    - 16-bit general registers: AX, BX, CX, DX;
    - 16-bit pointer registers: SP (stack pointer), BP (base pointer);
    - 16-bit index registers: SI (source index), DI (destination index);
    - 16-bit flag register（状态寄存器）: 9 of 16 bits are used;
    - ALU.

Registers

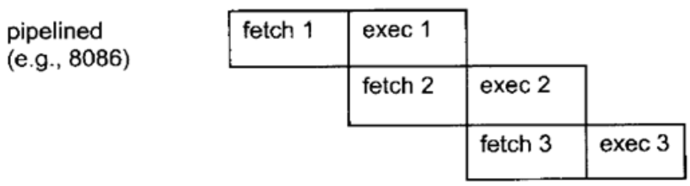
- On-chip storage: fast & expensive.
- Store information temporarily.
- Some registers can be decomposed into two smaller 8-bit registers. (e.g. AX can be decomposed into AH (A-High) and AL (A-Low)).
- Six groups of registers.

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).

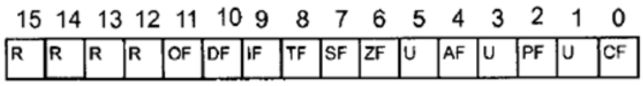
Pipelining

- Pipelined vs non-pipelined:
    - 8085: non-pipelined. n instructions need $2n$ cycles.
    - 8086: pipelined. n instructions need $(n+1)$ cycles if *perfectly-pipelined*.



- In 8086: two-stage pipelining: fetch & exec.
- When it works?
    - Sequential instruction execution
    - **Branch penalty**: when *jump instruction* executed, all pre-fetched instructions are discarded.

Flag Register



R =	reserved	SF =	sign flag
U =	undefined	ZF =	zero flag
OF =	overflow flag	AF =	auxiliary carry flag
DF =	direction flag	PF =	parity flag
IF =	interrupt flag	CF =	carry flag
TF =	trap flag		

- 16-bit (only 9 bits are used), *status register*, processor status word (PSW).
- **Control Flags**: DF (Direction Flag), IF (Interrupt Flag), TF (Trap Flag).
- **Conditional Flags**
    - CF (carry flag): set whenever there is a carry out, from d7 after a 8-bit op, from d15 after a 16-bit op
    - PF (parity flag): the parity of the operation result's low-order byte, set when the byte has an even number of 1s. **Ensure that** the combination of result and parity flag always have odd 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 operation.
    - OF (overflow flag): set when the result of a signed number operation is too large, causing the sign bit error.

Signed Number

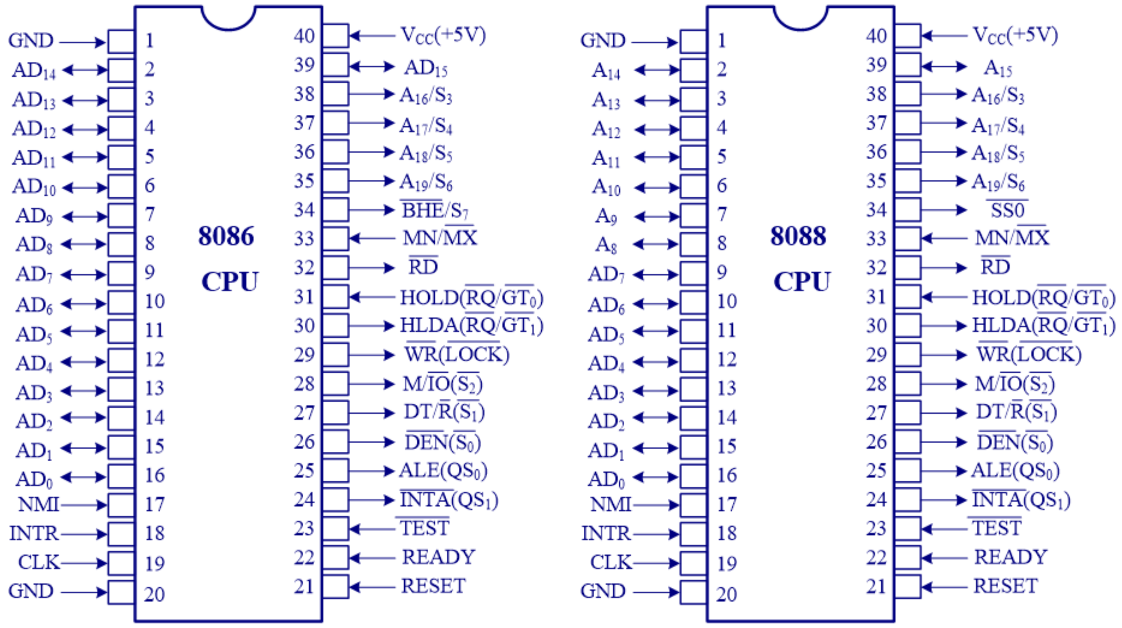
- Original value （原码）；
- One's complement （反码）；
- Two's complement （补码）: CPU uses two's complement in computer.
- **MSB** (most significant bit, sign bit).
- **CF** (carry flag) is used to detect errors in *unsigned* arithmetic operations;
- **OF** (overflow flag) is used to detect errors in *signed* arithmetic operations.

> [*Example*] Conditional Flags
>
> | | | | |
> |---|---|---|---|
> | | 38 | 0011 | 1000 |
> | + | 2F | 0010 | 1111 |
> | | 67 | 0110 | 0111 |
>
> CF = 0 since there is no carry beyond d7
> PF = 0 since there is an odd number of 1s in the result
> AF = 1 since there is a carry from d3 to d4
> ZF = 0 since the result is not zero
> SF = 0 since d7 of the result is zero
> OF = 0 since there is no carry from d6 to d7 and no carry beyond d7
>
> CPU does not know whether an operation is unsigned or signed. CPU treats them as both signed and unsigned, and simply reports the outcome of all possibilities (OF and CF). A high-level programmer must choose which flag to use (OF - signed, CF - unsigned).

Control Flags

- **IF (Interrupt Flag)**: set of cleared to enable or disable only the external maskable interrupt requests.
- **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.

3.2 Chip Interface of 8086



AD means the pin is used for both transferring address and data, and A means the pin is only used for transferring address.

Differences between 8086 CPU and 8088 CPU:

- 8086: AD_0 to AD_{15}, and A_{16} to A_{19}.
- 8088: AD_0 to AD_7, and A_8 to A_{19}.

Work Mode: there are two work modes of 8086/88.

- Minimum code: $MN/\overline{MX} = 1$.
  - Single CPU;
  - Control signals from CPU.
- Maximum code: $MN/\overline{MX} = 0$
  - Multiple CPUs (8086 + 8087)
  - 8288 control chips supports.

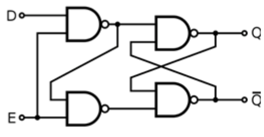
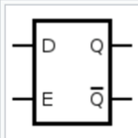
Control Signals:

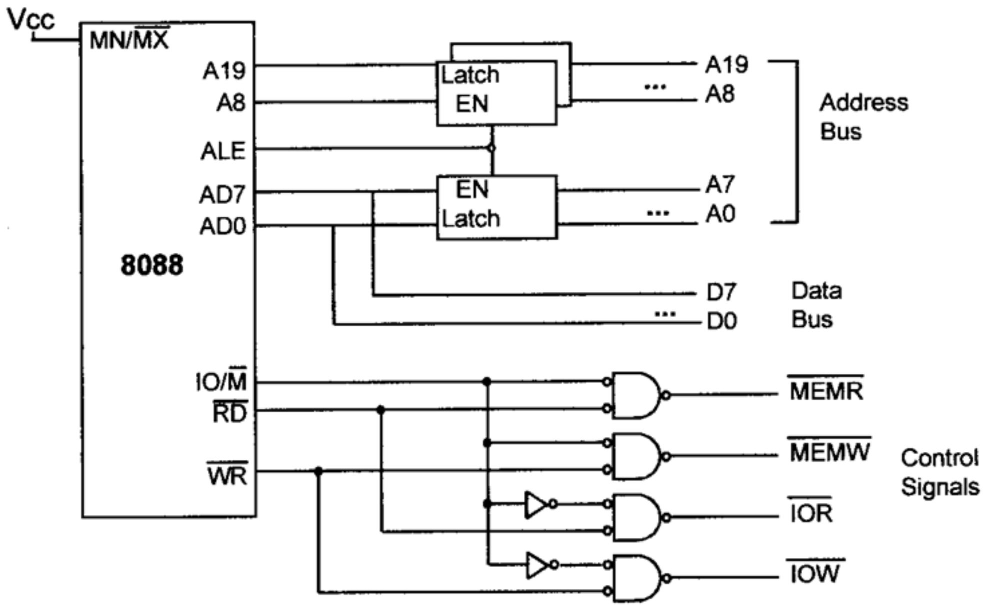
Pins with hat means it is activated when signal is low; and pins without hat means it is activated when signal is high.

- $MN/\overline{MX}$: 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 use 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. (i.e. sending - high, receiving - low).
- $\overline{BHE}$: output. When $\overline{BHE} = 0$, AD_8 to AD_{15} are used; otherwise, AD_8 to AD_{15} are not used;

- ALE: output, used as the latch enable signal of the address latch（地址锁存器）.
- $HOLD$: 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.

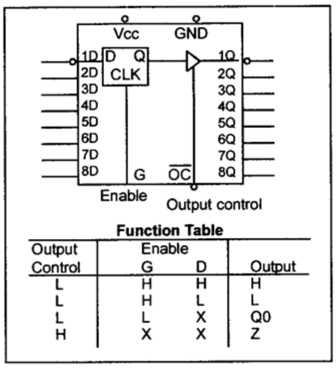
Address/Data Bus Demultiplexing & Address Latching

> [*Example*] **Sequential Logic**（时序逻辑电路）: D Latch （D 锁存器）
>
> Gated D latch truth table
>
> | E/C | D | Q | $\overline{Q}$ | Comment |
> |---|---|---|---|---|
> | 0 | X | Q_{prev} | $\overline{Q}_{prev}$ | No change |
> | 1 | 0 | 0 | 1 | Reset |
> | 1 | 1 | 1 | 0 | Set |
>
> 
>
> A gated D latch based on an $\overline{SR}$ NAND latch
>
> 
>
> Symbol for a gated D latch
>
> Logically, D-latch is the same as a SRAM cell.
>
> Only a high voltage signal in E/C can change the signal in D latch.

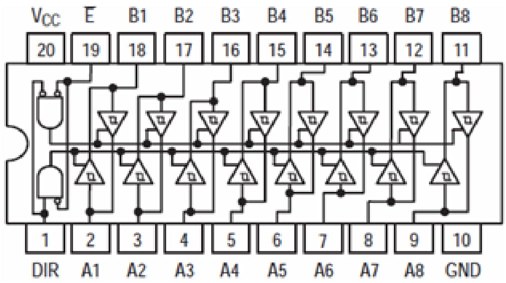


CPU first activate ALE (Address Latch Enable) (which links to the E pins in D latch), then transfer the address signals, then the signal will be stored in D latch (aka address latches). After that the CPU can deactivate ALE and transfer the data signals to the data bus, and address bus can read the address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

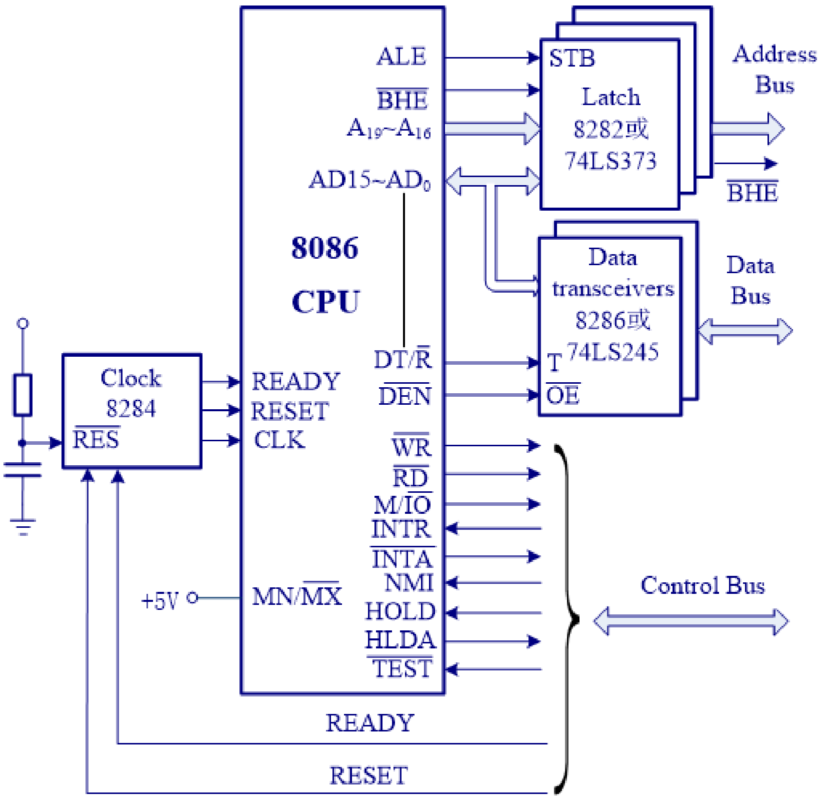
Do bit extensions on the latch, then we get Latch 74LS373.



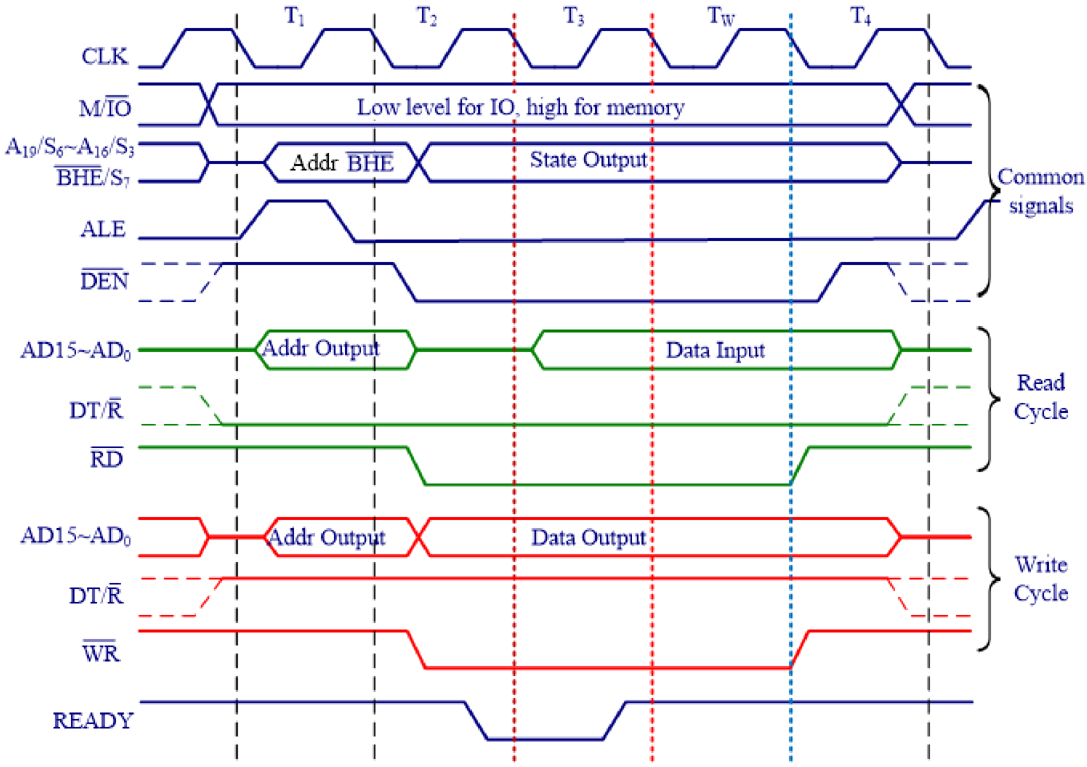
We also need transceivers (74LS245), which is bit-extension result of some *tri-gates*（三态门）. The function of tri-gates enables the signal can only flow at one directions. Connect $DT/\overline{R}$ and $\overline{DEN}$ to the corresponding pins in transceiver and then it works.



Connect the latch and transceivers to the CPU, and we get the 8086 CPU models.



> [*Example*] 8086 Bus cycle for data transfers



3.3 Memory Management in 8086

A typical program on 8086 consists of at least 3 *segments*（段）.

- 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 $64\ \mathrm{KB}$.
- 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;

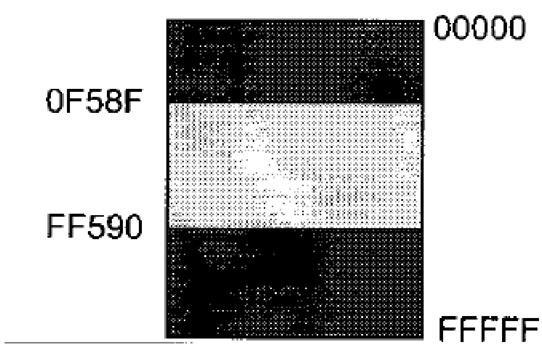
Logical Address: Consists of a *segment value* (determines the beginning of a segment) and an *offset address* (a relative location within a 64KB segment).

- *physical address* = 16 * *segment value* + *offset address*

  > [*Example*] an instruction in the code segment has a logical address in the form of CS (code segment register): IP (instruction pointer).

- **Wrap-around**: when adding the offset to the shifted segment value results in an address beyond the maximum value $FFFFFH$, then we actually wraps around from $00000H$ again (modulo $100000H$).

  > [*Example*] If $CS = FF59H$, then the low range is $FF590H$, and the ranges goes to $FFFFFH$, and wraps around from $00000H$ to $0F58FH$ $(FF590 + FFFF)$.



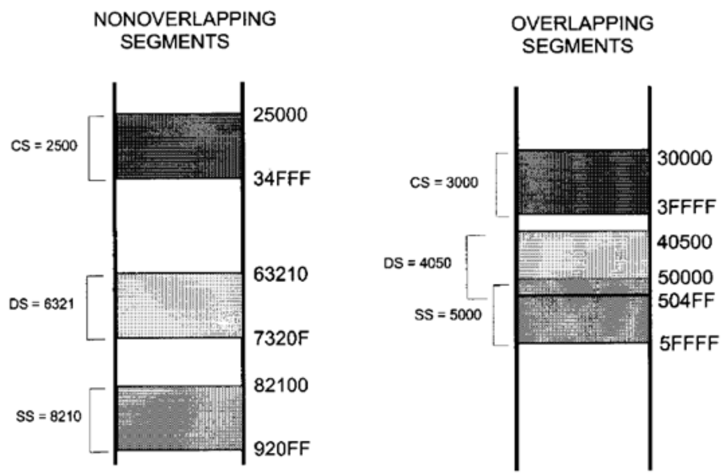
Logical Address to Physical Address

- Shift the segment value left one hex digit (or 4 bits)
- Then adding the above value to the offset address
- One logical -> only one physical.

Physical Address to Logical Address

- **NOT** one-to-one mapping!!!
- One physical address can be derived from several different logical addresses.

Segment Overlapping



- Dynamic behavior of the segment and offset concept;
- May be desirable in some circumstances.

Code Segment

- Logical address of an instruction $CS : IP$;
- Physical address is generated to retrieve this instruction from memory;
- 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.

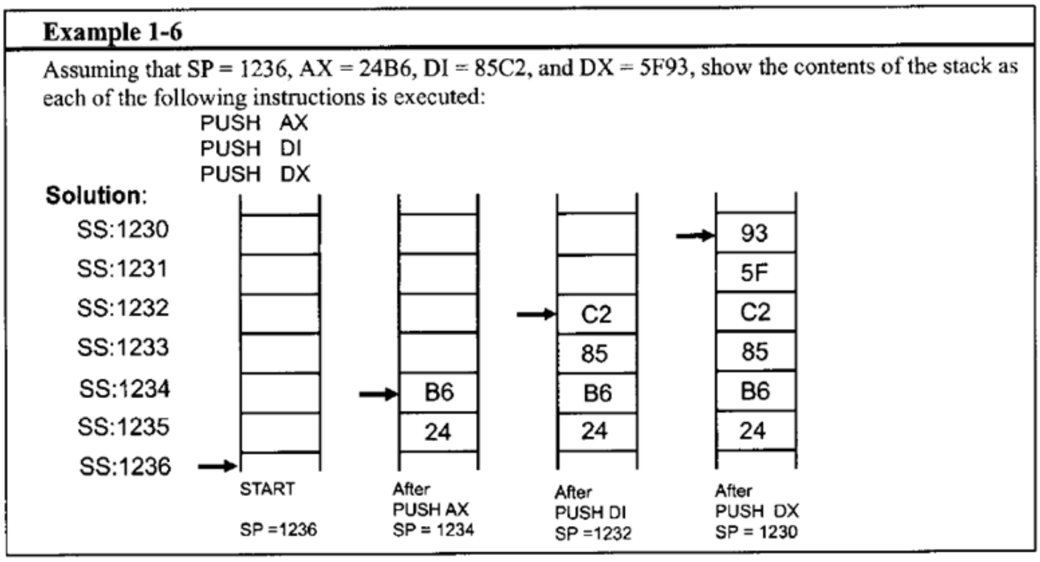
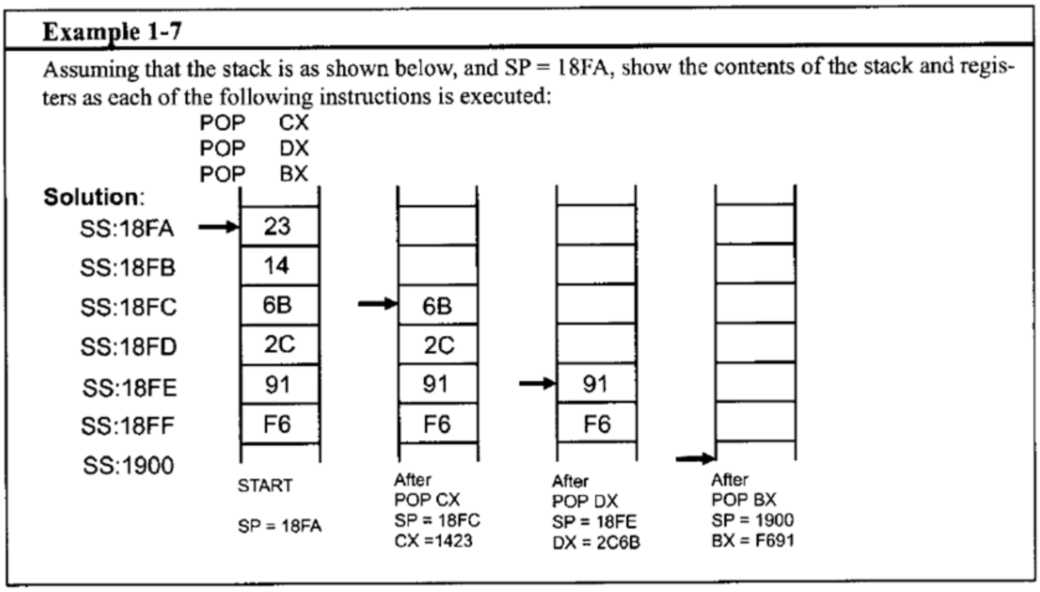
Data Segment

- Logical address of a piece of data: $DS : offset$;
  - $offset$ registers: BX, SI, DI.
- Physical address is generated to retrieve data (8-bit of 16-bit) from memory;
- 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**: a multi-byte value will be stored in the consecutive address in the memory.

- Little endian: 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.

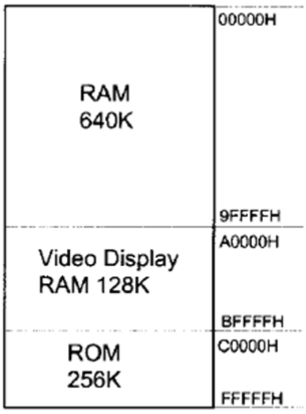
Stack Segment

- 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 in the memory allocated for a program.
    - To protect other programs from destruction;
    - Also, we need to ensure that the code section and stack section would not write over each other!
        - Hackers can use these properties to modify your code by stack overflow!

> [*Example*] Stack push & Stack pop. (Use the little endian)
>
> 
> **Example 1-6**
> Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed:
>
> ```
> PUSH  AX
> PUSH  DI
> PUSH  DX
> ```
>
>
>
> 
> **Example 1-7**
> Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:
>
> ```
> POP   CX
> POP   DX
> POP   BX
> ```
>
>
>
> **NOTE**: Remember to update SP after every operation.

Extra segments

- Logical address of a piece of data: $ES : offset$
    - Offset registers for data segment: BX, SI, DI.

> [*Example*] **Memory Map of the IBM PC**
>
> 
>
>
> - 1MB logical address space;
> - 640K max RAM;
> - Video display RAM;
> - ROM:
>   - 64KB BIOS;
>   - Various adapter cards.

3.4 Addressing Modes in 8086

> 8086 has seven distinct addressing modes:
>
> - register;
> - immediate;
> - direct;
> - register indirect;
> - based relative;
> - indexed relative;
> - based indexed relative.

MOV instruction

$$MOV \text{ destination, source}$$

Destination and source should have the same result.

Register Addressing Mode: Data are held within registers

- no need to access memory;
- Data can be moved among all registers except CS (can not be set) and IP (cannot be accessed by MOV).

> [*Example*]
>
> ```
> MOV AX, BX
> ```

Immediate Addressing Mode: The source operand is a constant.

- Embedded in instructions;
- no need to access memory.
- Immediate numbers CANNOT be moved to segment registers.

> [*Example*]

> ```
> MOV AX, 2550H
> MOV BL, 40H
> ```

Direct Addressing Mode: Data is stored in memory and the address is given in instructions.

- Segment address in data segment (DS) by default;
- Need to access memory to gain data.

> [*Example*]
>
> ```
> MOV DL, [2400]
> ```
>
> move contents of `DS: 2400H` into `DL`.

Register Indirect Addressing Mode: 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;
- Need to access memory to gain the data.

> [*Example*]
>
> ```
> MOV AL, [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;
- Need to access memory to gain data;

> [*Example*]
>
> $MOV\ CX, [BX] + 10$: move $DS : BX + 10$ and $DS : BX + 10 + 1$ into CX;
>
> $MOV\ AL, [BP] + 5$

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.

- The default segment is data segment DS;
- Need to access memory to gain the data;

> [*Example*]
>
> ```
> MOV DX, [SI + 5]
> ```
>
> move `DS: SI+5` and `DS: SI+5+1` into `DX`.

Based Indexed Relative Addressing Mode: 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;
- Need to access memory to gain data.

> [*Example*]
>
> ```
> MOV CL, [BX][DI] + 8
> ```

> move `DS: BX+DI+8` into `CL`.

Summarize

$$PA = \{CS, SS, DS, ES\} : \{BX, BP\} + \{SI, DI\} + \{8/16\text{−bit displacement}\}$$

where, PA is the actual address.

We can also specify the segment register in the code

> [*Example*]
>
> ```
> MOV AX, CS:[BP]
> MOV DX, SS:[SI]
> ```