Assembly Language

Processors Prior to 8086

- (1971) 4004 – First processor made by the Intel Corporation. Allowed computer intelligence to be put into small devices like cell phones, key chains, calculators, etc.

- (1972) 8008 – Twice as powerful as the 4004, but was used in the Mark-8. Mark-8 was one of first personal computers.

Processors Prior to 8086 (cont.)

- (1974) 8080 – Slight improvement on the 8008 with a more complex instruction set. Started to mass produce for personal computers. Last processor update before 8086.

Intel's 8086 and 8088

- (1978) 8086/8088 – Biggest improvement of the 8-bit processors. Laid the groundwork for the X86 architecture in processors. X86 is still used in the newer Pentium models today. The 8088 processor was selected by IBM to be placed in the "IBM PC" which was their most popular product. Skyrocketed Intel's stature as a company and was honored by being named a Fortune 500 company.

Processors After 8086/8088

- (1982-89) 286/386/486 – Started being able to run multiple programs at one time and point and click operating systems.

- (1993-2001) Pentium's 1-4 – Much faster speeds allowed multimedia elements like voice, sounds, and graphics to run much clearer and faster.

8086 Programming

- 8086 is low level assembly language:
  - Uses fetch/execute cycle.
  - Registers hold addresses for data, instructions and program flow counter.
  - Flows from beginning to end, manipulated by changing counter.
- Compatible in design with future 16bit or greater microprocessors.
- 6 basic sets of instructions, not including string manipulation.

8086 Programming (cont.)

- Arithmetic
  - Addition, subtraction etc. ADD, SUB
- Logic
  - Logical operations. AND, OR, XOR
- Shift
  - Shifting bits, rotate, logic and arithmetic. SAR, SHL
- Data Transfer
  - Moving data, copying. MOV, OUT, POP
- Control Transfer
  - Flow control, jumps, and subroutines. JMP, RET
- Processor Control
  - Processor instructions. NOP, CLI

8086 Programming (cont.)

- Instruction form
  - Op-code Destination Operand, Source Operand
  - MOV AX,100
- Variable declarations
  - Variable Name Memory Directive Value
  - Var1 DB 7

Memory Interfacing

- The 8086 has a 20-bit address bus.
- The subsystem is organized into 8-bit bytes instead of 16-bit words.
- The memory is split into two 8-bit banks.
- Words can be stored at even or odd addresses.

Input / Output

- The 8086 uses three main methods of I/O: Programmed, Interrupt-Driven and Direct Memory Access.
- Programmed I/O relies on the processor to poll the devices.
- Interrupt-Driven I/O is the opposite of programmed because the devices signal when they need service.

Input / Output

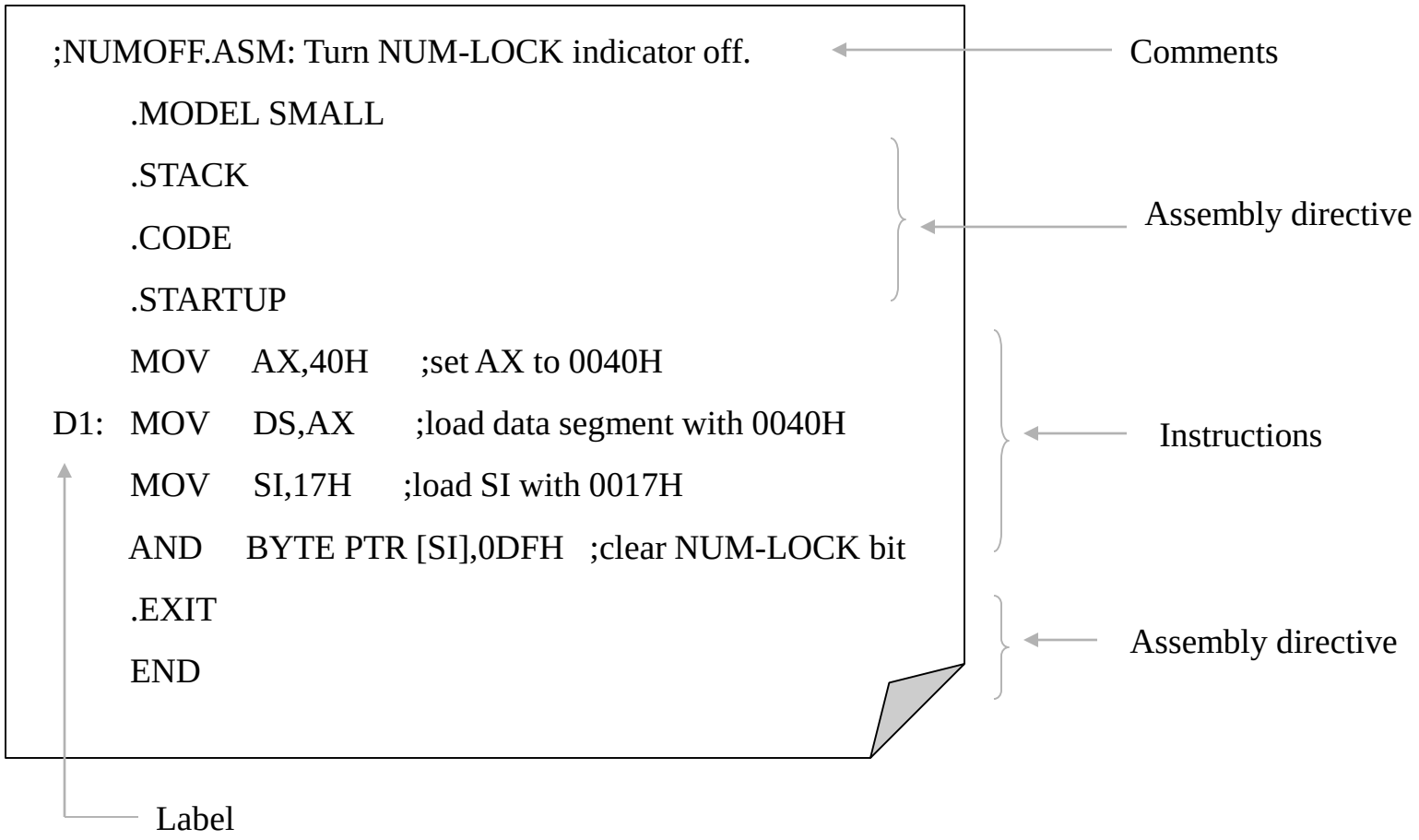
- The 8086 has two pins for hardware interrupts.
- The second pin can be expanded by the Programmable Interrupt Controller (PIC).
- Direct memory access removes the processor from the memory and I/O allowing a direct transfer.
- The 8086 uses the Direct Memory Access controller (DMAC) to remove the processor from the loop and control the system.

Overview of Assembly Language

- Advantages:
  - ✓ Faster as compared to programs written using high-level languages
  - ✓ Efficient memory usage
  - ✓ Control down to bit level
- Disadvantages:
  - × Need to know detail hardware implementation
  - × Not portable
  - × Slow to development and difficult to debug
- Basic components in assembly Language:

  Instruction, Directive, Label, and Comment

Example of Assembly Language Program



The Big (Simplified) Picture

High-level code

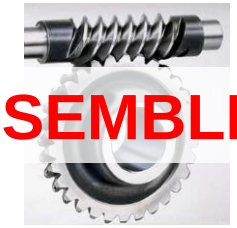
```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

COMPILER

Assembly code

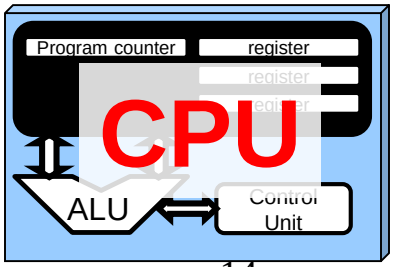
```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```



ASSEMBLER

Machine code

```
010000101010110110
101010101111010101
101001010101010001
101010101010100101
111100001010101001
000101010111101011
010000000010000100
000010001000100011
101001010010101011
000101010010010101
010101010101010101
101010101111010101
101010101010100101
111100001010101001
```



The Big (Simplified) Picture

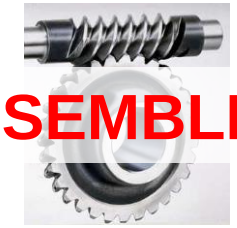
High-level code

```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

Hand-written Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

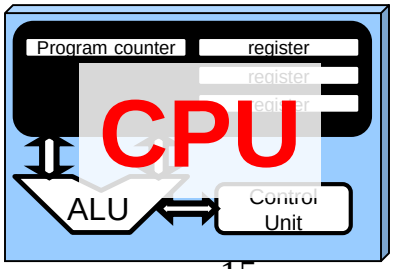


Machine code

```
010000101010110110
101010101111010101
101001010101010001
101010101010100101
111100001010101001
000101010111101011
010000000010000100
000010001000100011
101001010010101011
000101010010010101
010101010101010101
101010101111010101
101010101010100101
111100001010101001
```

Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```



What we do in this class:

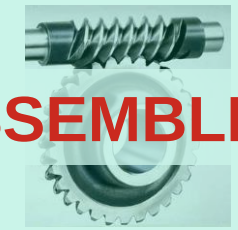
High-level code

```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

Hand-written Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

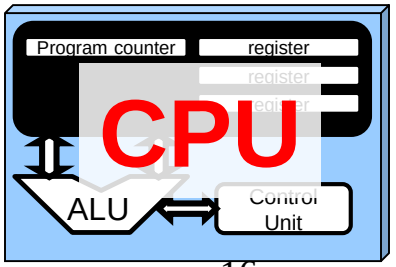


Machine code

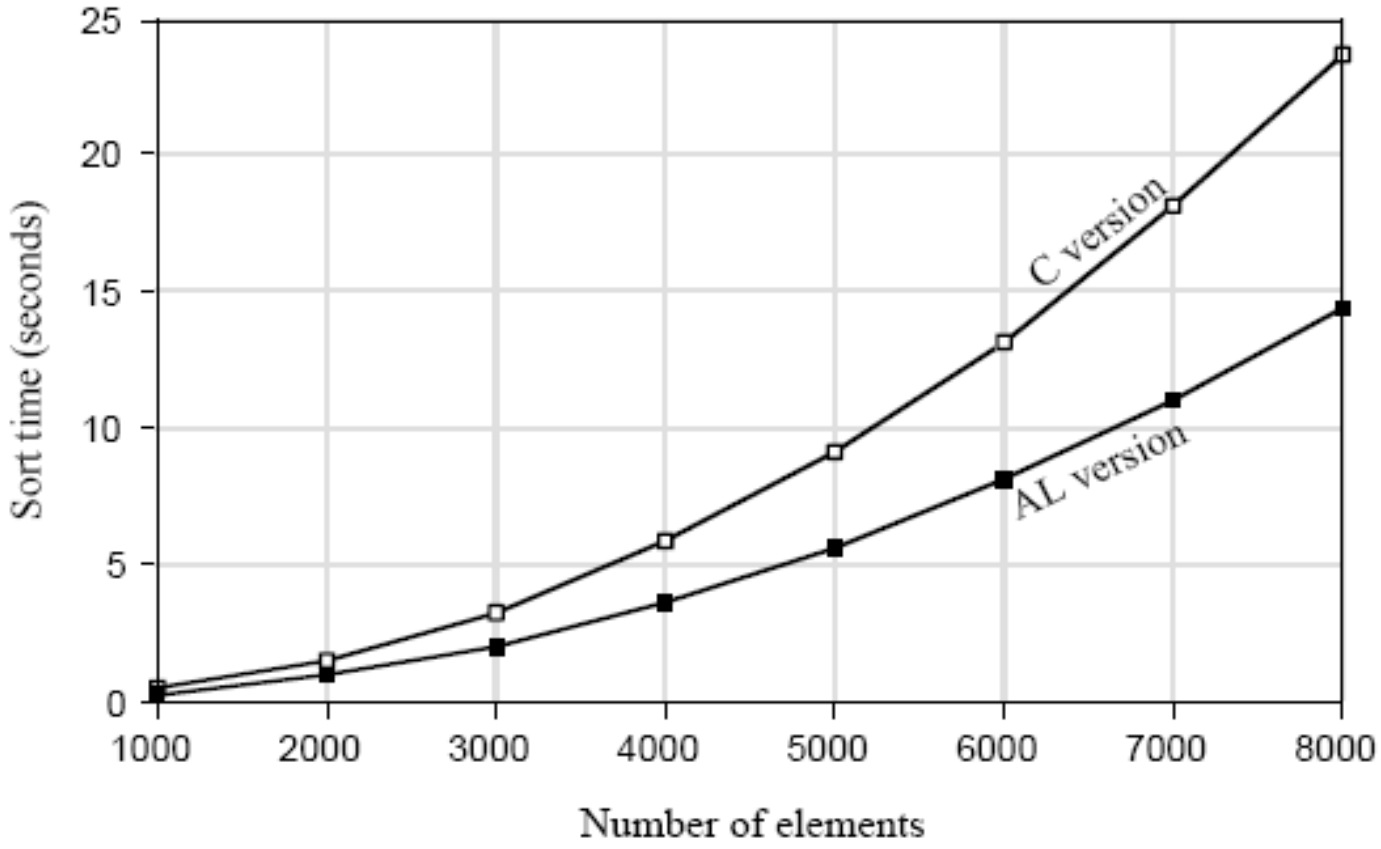
```
010000101010110110
101010101111010101
101001010101010001
101010101010100101
111100001010101001
000101010111101011
010000000010000100
000010001000100011
101001010010101011
000101010010010101
010101010101010101
101010101111010101
101010101010100101
111100001010101001
```

Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```



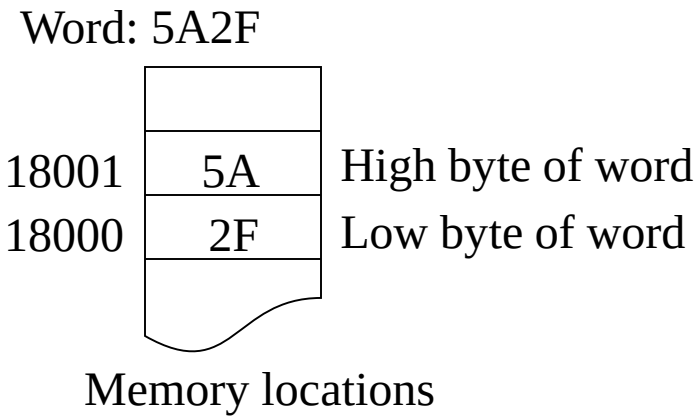
Performance : Bubble Sort Example

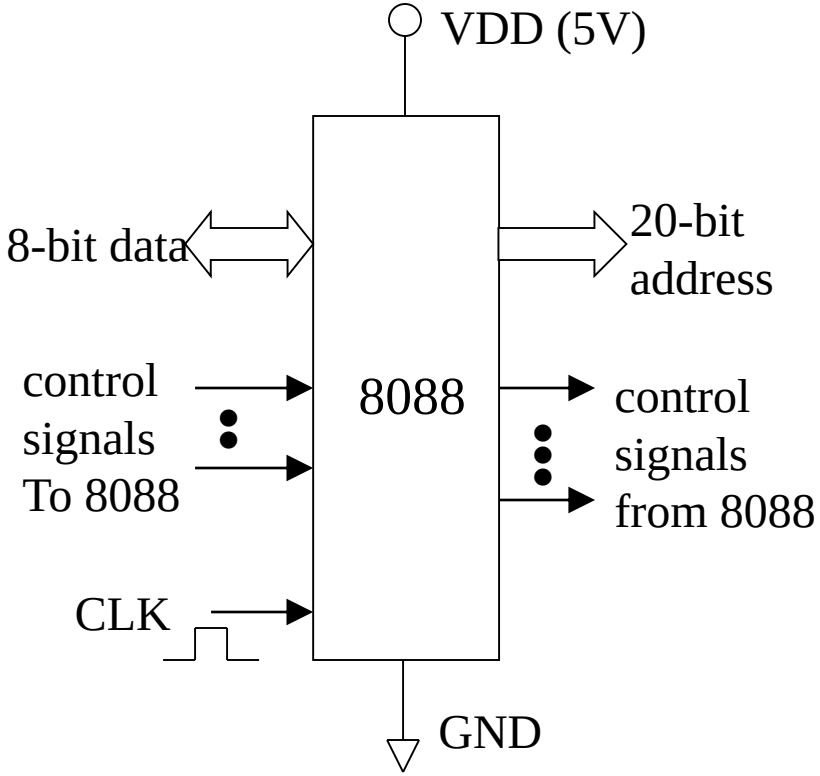


Software Model for the 8086

Overview

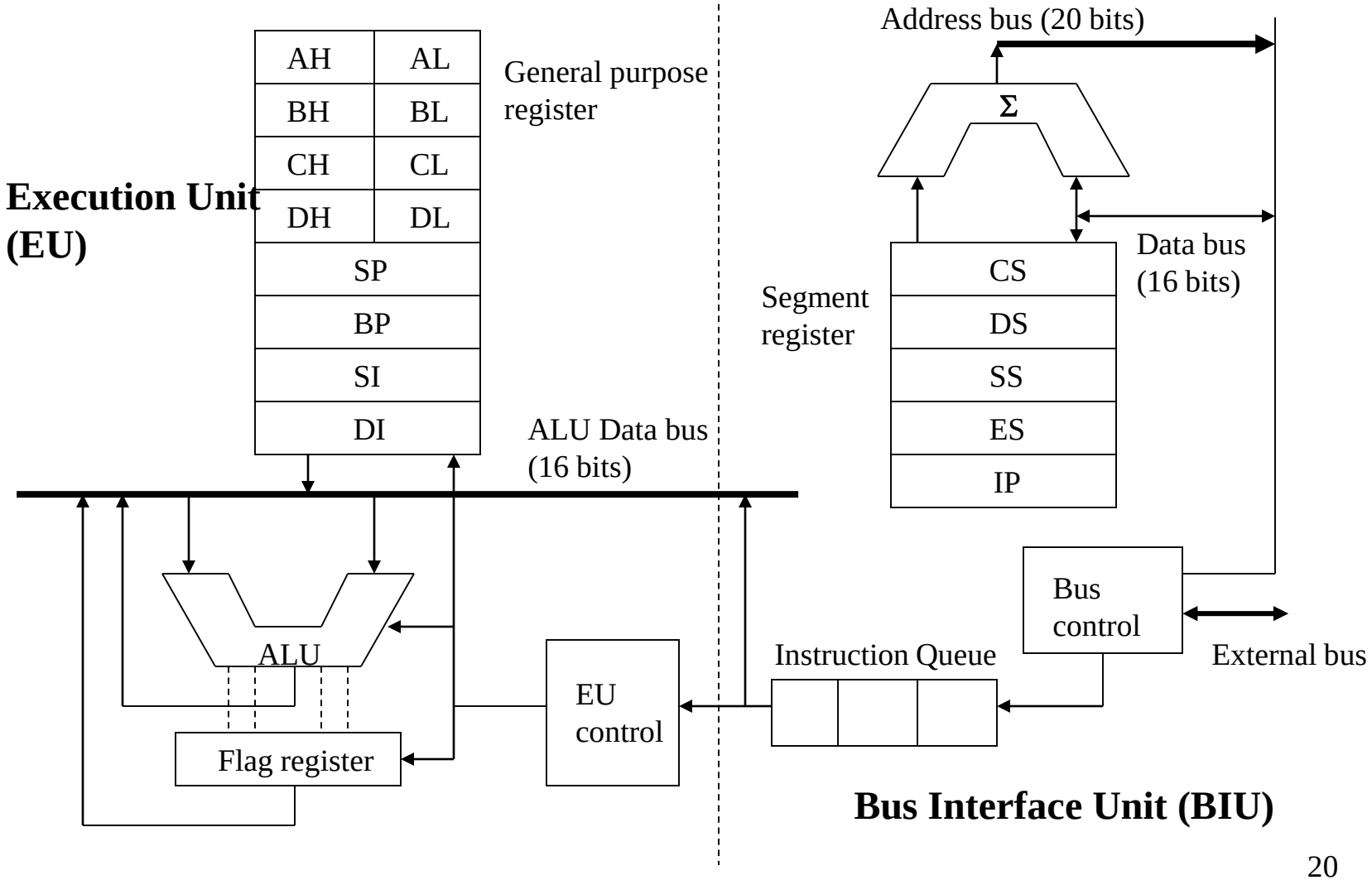
- Intel 8088 facts
  - 20 bit address bus allow accessing 1 M memory locations
  - 16-bit internal data bus and 8-bit external data bus. Thus, it need two read (or write) operations to read (or write) a 16-bit datum
  - Byte addressable and byte-swapping





8088 signal classification

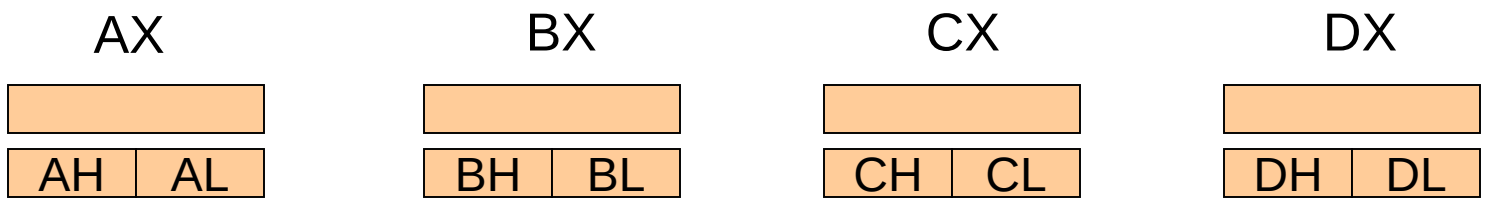
Organization of 8088/8086



The 8086 Registers

- To write assembly code for an ISA (Instruction Set Architecture) you must know the name of registers
  - Because registers are places in which you put data to perform computation and in which you find the result of the computation (think of them as variables)
  - The registers are really numbered, but assembly languages give them "easy-to-remember" names
- The 8086 offered 16-bit registers
- Four general purpose 16-bit registers
  - AX
  - BX
  - CX
  - DX

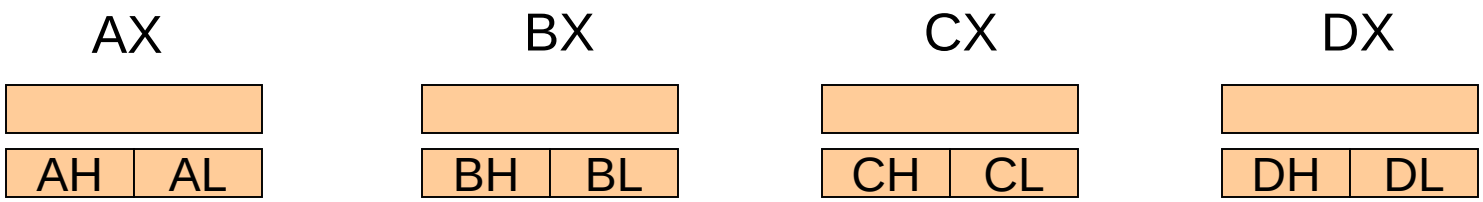
The 8086 Registers



- Each of the 16-bit registers consists of 8 “low bits” and 8 “high bits”
  - Low: least significant
  - High: most significant

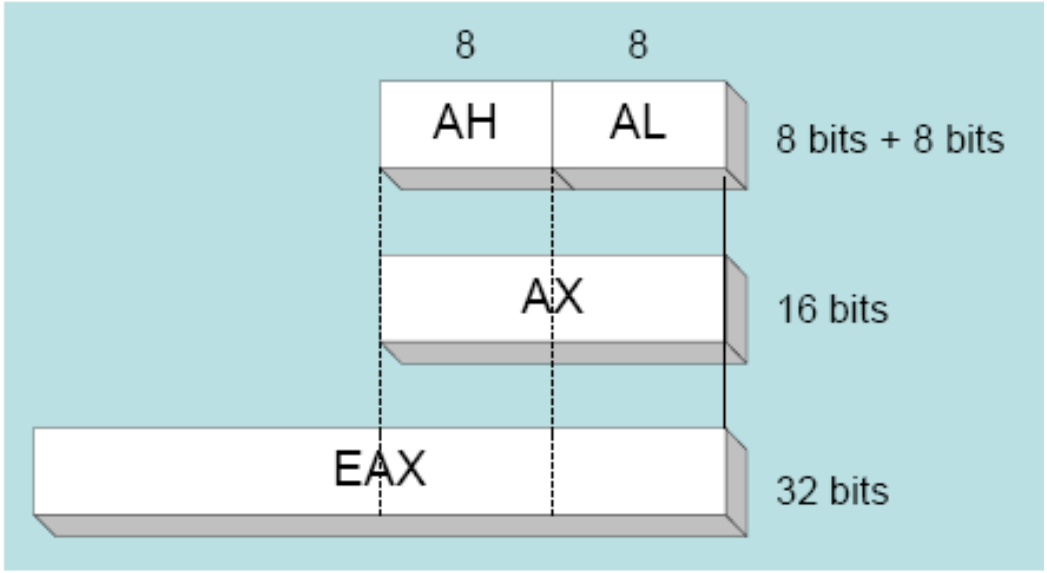
- The ISA makes it possible to refer to the low or high bits individually
  - AH, AL
  - BH, BL
  - CH, CL
  - DH, DL

The 8086 Registers



- The xH and xL registers can be used as 1-byte register to store 1-byte quantities
- Important: both are “tied” to the 16-bit register
  - Changing the value of AX will change the values of AH and AL
  - Changing the value of AH or AL will change the value of AX

The 80x86 Registers



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The 8086 Registers

- Two 16-bit index registers:
  - SI
  - DI
- These are basically general-purpose registers
- But by convention they are often used as "pointers", i.e., they contain addresses
- And they cannot be decomposed into High and Low 1-byte registers

The 8086 Registers

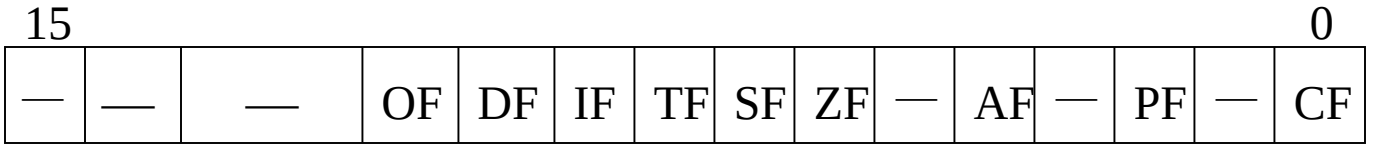
- Two 16-bit special registers:
  - BP: Base Pointer
  - SP: Stack Pointer
  - We'll discuss these Later
- Four 16-bit segment registers:
  - CS: Code Segment
  - DS: Data Segment
  - SS: Stack Segment
  - ES: Extra Segment
  - We'll discuss them later as well

The 8086 Registers

- The 16-bit Instruction Pointer (IP) register:
  - Points to the next instruction to execute
- The 16-bit FLAGS registers
  - Information is stored in individual bits of the FLAGS register
  - Whenever an instruction is executed and produces a result, it may modify some bit(s) of the FLAGS register
  - Example: Z (or ZF) denotes one bit of the FLAGS register, which is set to 1 if the previously executed instruction produced 0, or 0 otherwise

Flag Register

- Flag register contains information reflecting the current status of a microprocessor. It also contains information which controls the operation of the microprocessor.



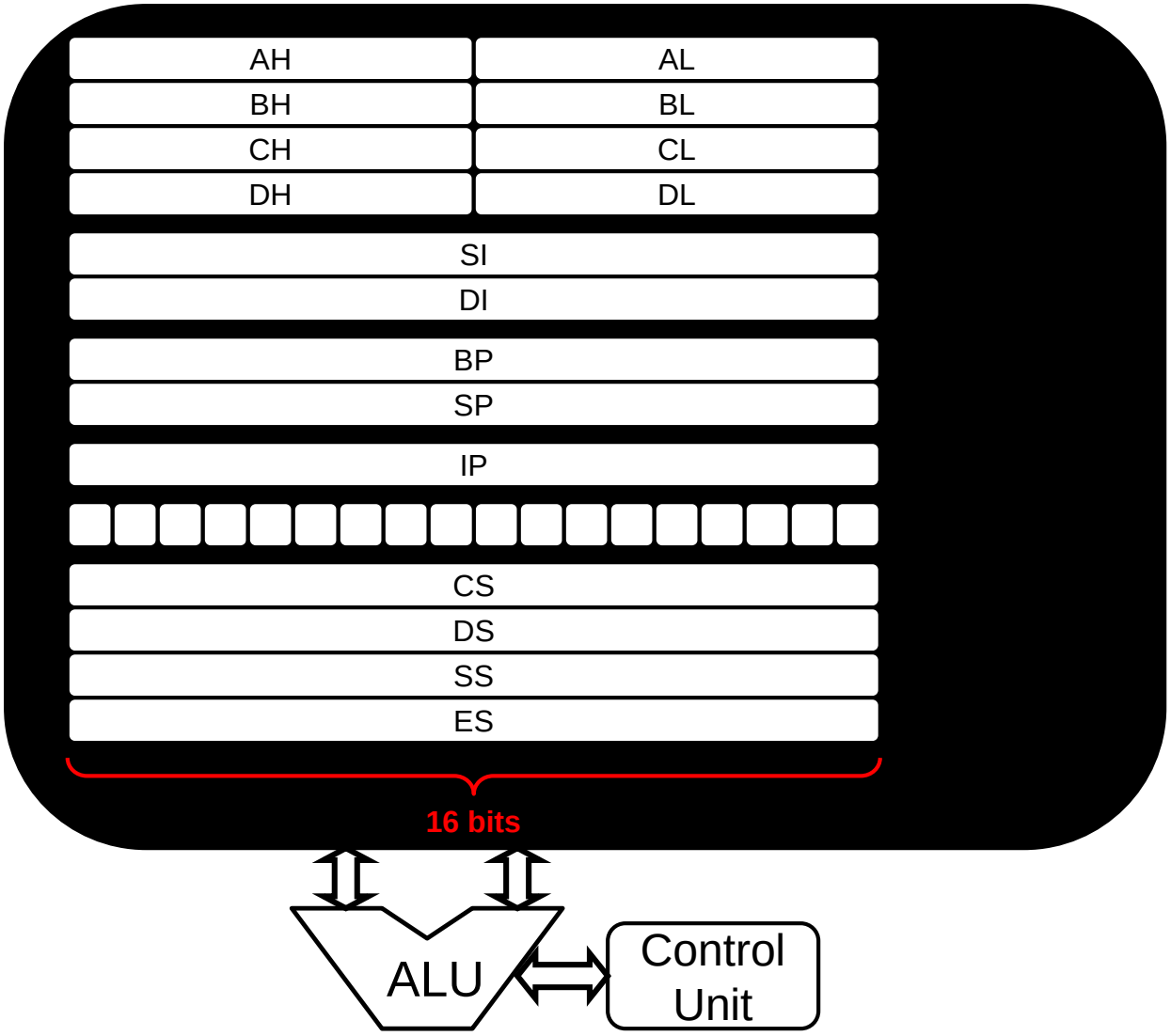
- Control Flags

IF:	Interrupt enable flag
DF:	Direction flag
TF:	Trap flag

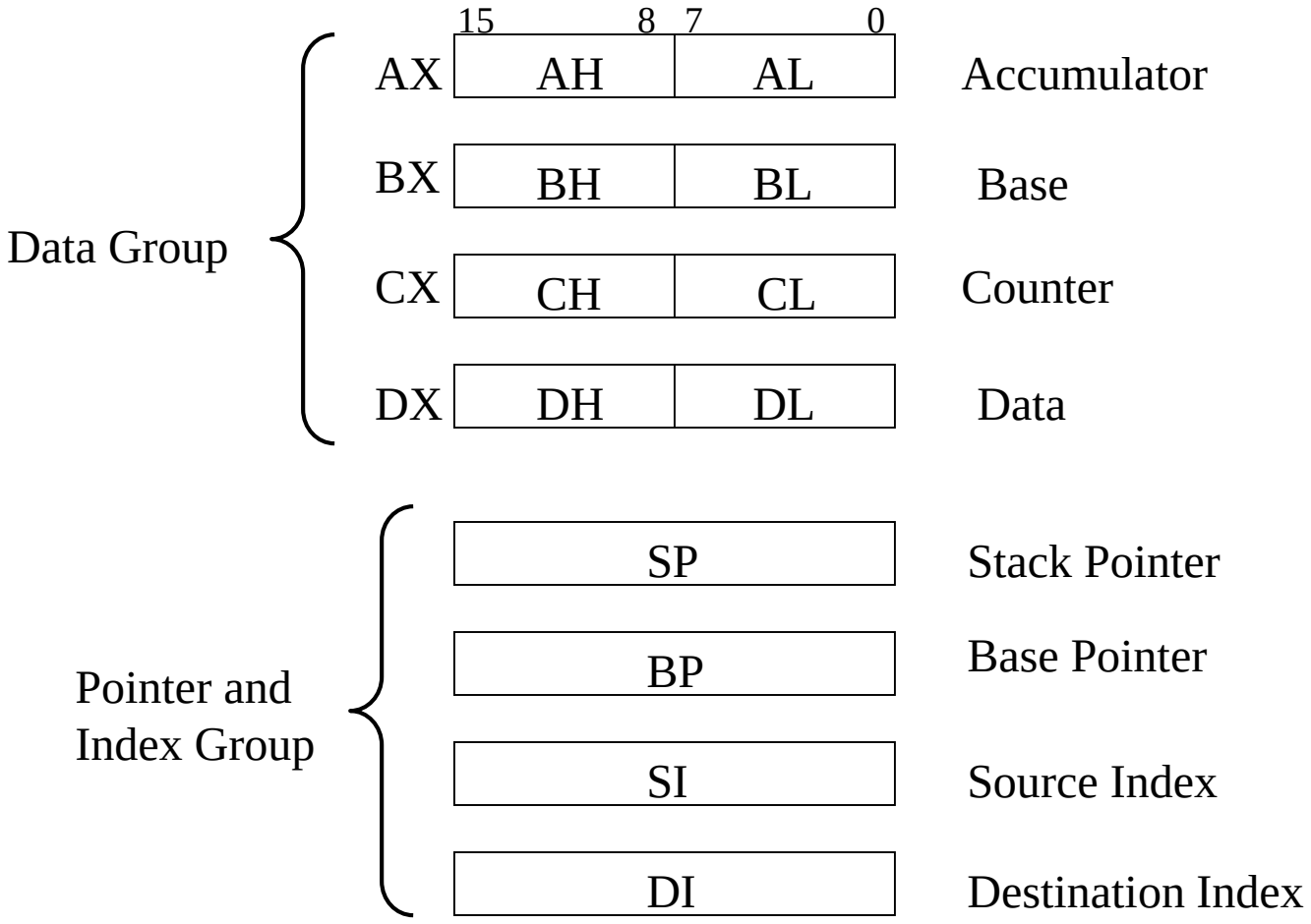
- Status Flags

CF:	Carry flag
PF:	Parity flag
AF:	Auxiliary carry flag
ZF:	Zero flag
SF:	Sign flag
OF:	Overflow flag

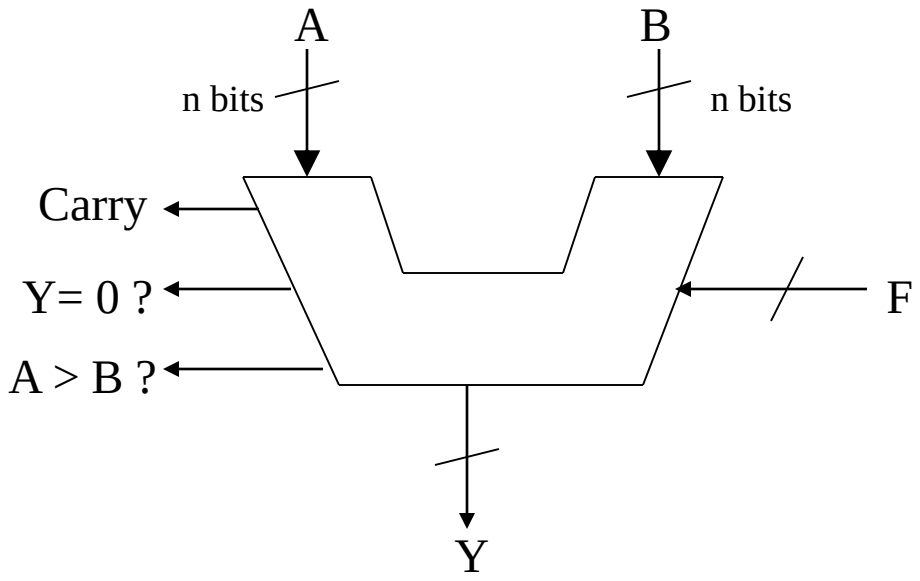
The 8086 Registers



General Purpose Registers



Arithmetic Logic Unit (ALU)

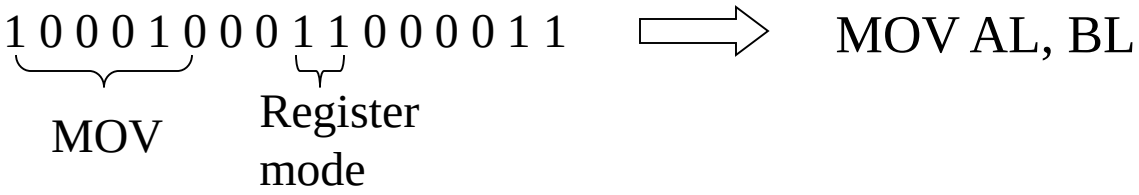


F	Y
0 0 0	A + B
0 0 1	A - B
0 1 0	A - 1
0 1 1	A ***and*** B
1 0 0	A ***or*** B
1 0 1	***not*** A
• • •	• • •

- Signal F control which function will be conducted by ALU.
- Signal F is generated according to the current instruction.
- Basic arithmetic operations: *addition*, *subtraction*, •••••
- Basic logic operations: *and*, *or*, *xor*, *shifting*,•••••

Instruction Machine Codes

- Instruction machine codes are binary numbers
  - ***For Example:***



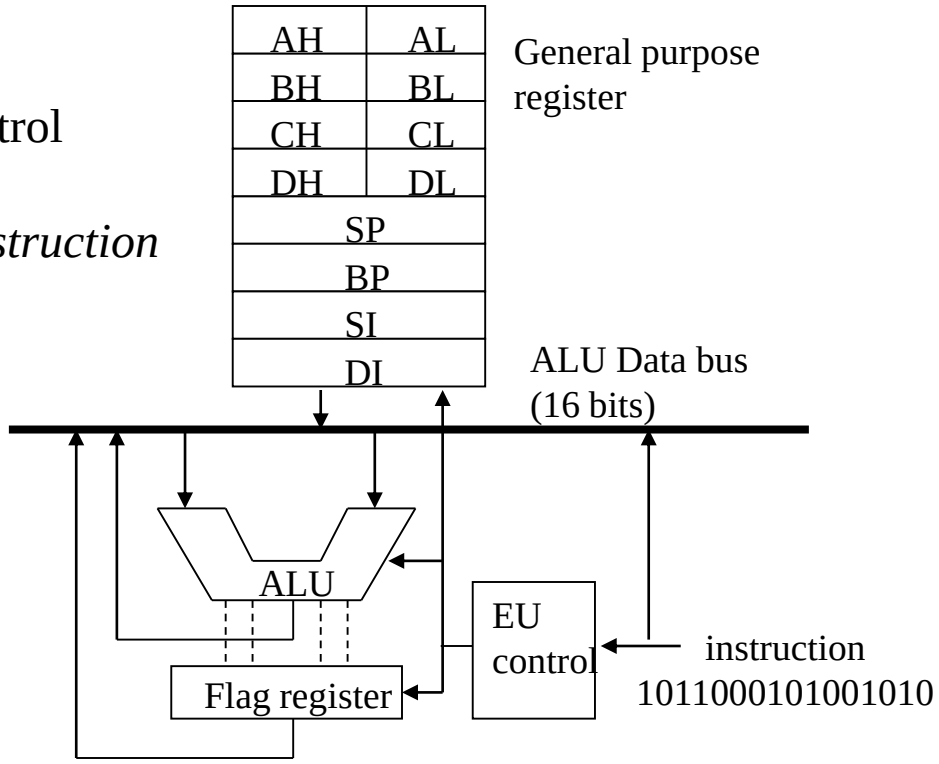
- Machine code structure

Opcode	Mode	Operand1	Operand2

  - Some instructions do not have operands, or have only one operand
  - Opcode tells what operation is to be performed.
    (EU control logic generates ALU control signals according to Opcode)
  - Mode indicates the type of a instruction: *Register type, or Memory type*
  - Operands tell what data should be used in the operation. Operands can be addresses telling where to get data (or where to store results)

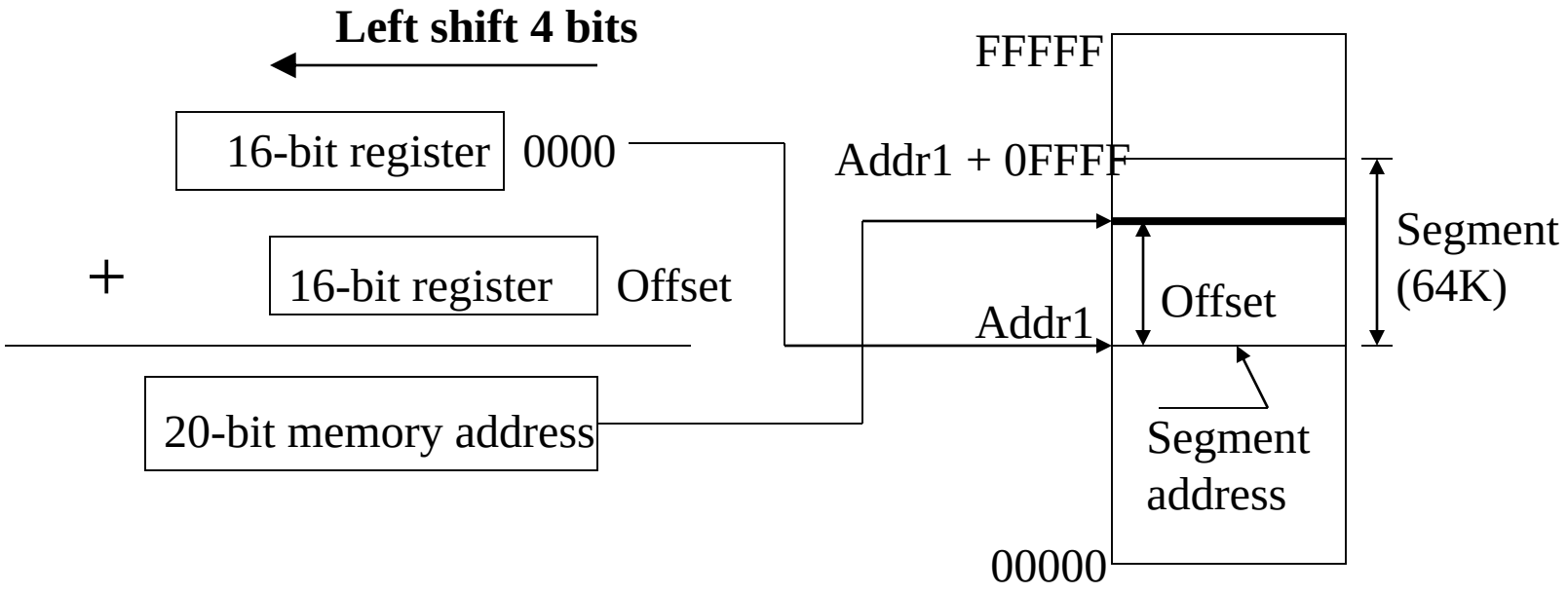
EU Operation

1. Fetch an instruction from instruction queue
2. According to the instruction, EU control logic generates control signals.
   (*This process is also referred to as instruction decoding)*
3. Depending on the control signal, EU performs one of the following operations:
   - An arithmetic operation
   - A logic operation
   - Storing a datum into a register
   - Moving a datum from a register
   - Changing flag register



Generating Memory Addresses

- How can a 16-bit microprocessor generate 20-bit memory addresses?

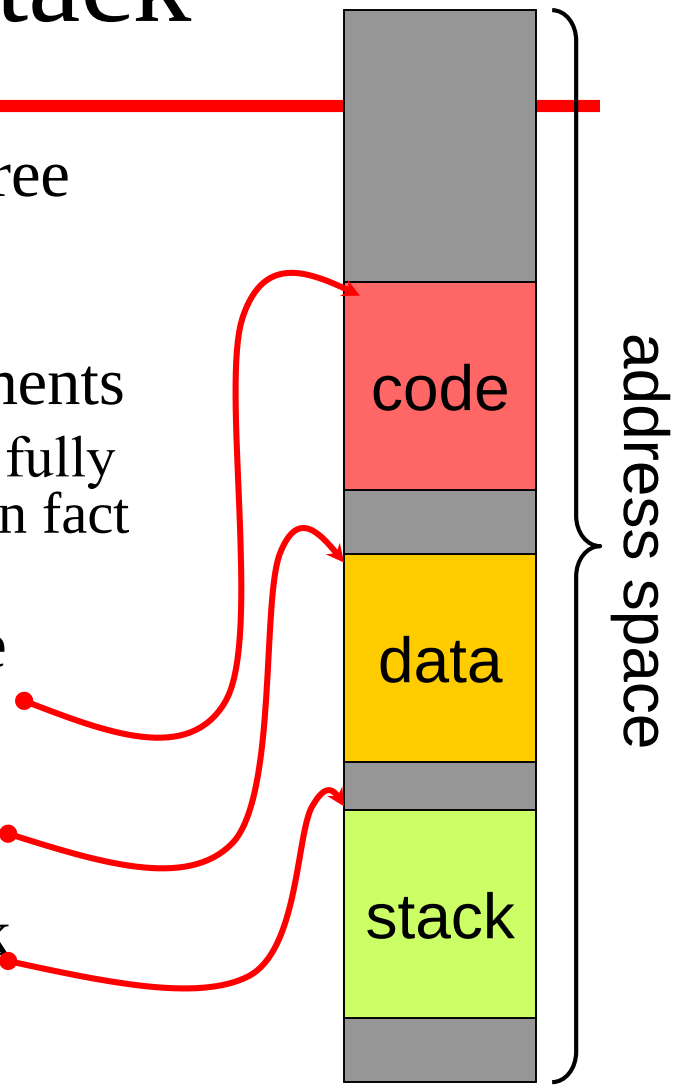


Intel 80x86 memory address generation

1M memory space

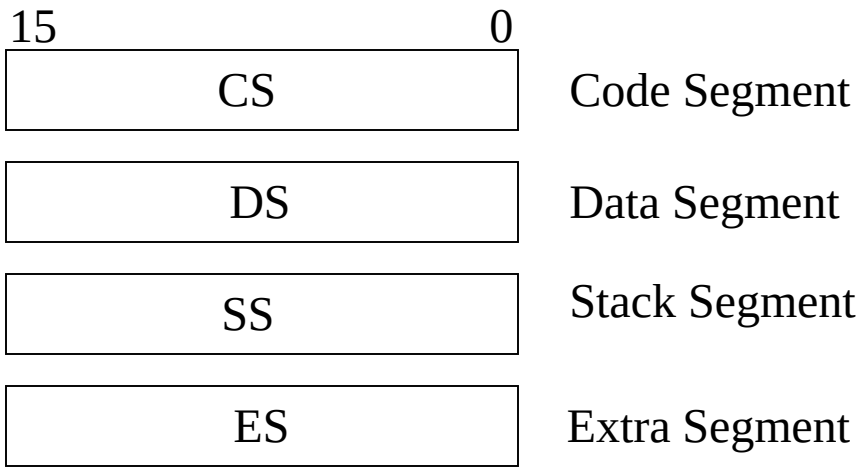
Code, Data, Stack

- A program constantly references all three regions
- Therefore, the program constantly references bytes in three different segments
  - For now let's assume that each region is fully contained in a single segment, which is in fact not always the case
- CS: points to the beginning of the code segment
- DS: points to the beginning of the data segment
- SS: points to the beginning of the stack segment



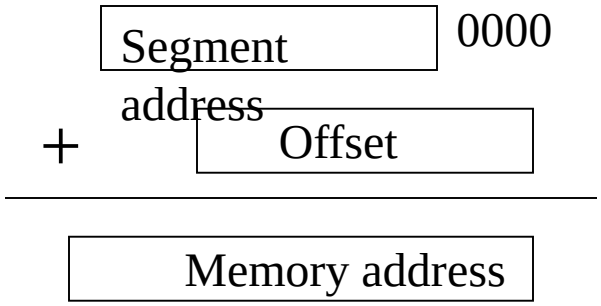
Memory Segmentation

- A segment is a 64KB block of memory starting from any 16-byte boundary
  - For example: 00000, 00010, 00020, 20000, 8CE90, and E0840 are all valid segment addresses
  - The requirement of starting from 16-byte boundary is due to the 4-bit left shifting
- Segment registers in BIU



Memory Address Calculation

- Segment addresses must be stored in segment registers
- Offset is derived from the combination of pointer registers, the Instruction Pointer (IP), and immediate values



- Examples

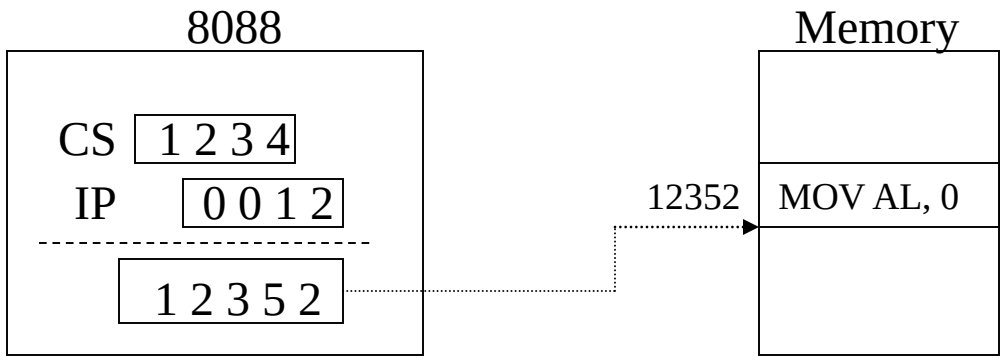
CS	3	4	8	A	0
IP +		4	2	1	4
Instruction address	3	8	A	B	4

SS	5	0	0	0	0
SP +		F	F	E	0
Stack address	5	F	F	E	0

DS	1	2	3	4	0
DI +		0	0	2	2
Data address	1	2	3	6	2

Fetching Instructions

- Where to fetch the next instruction?



- Update IP
  - — After an instruction is fetched, Register IP is updated as follows:

IP = IP + Length of the fetched instruction

  - — For Example: the length of **MOV AL, 0** is 2 bytes. After fetching this instruction, the IP is updated to 0014

Accessing Data Memory

- There is a number of methods to generate the memory address when accessing data memory. These methods are referred to as **Addressing Modes**
- Examples:
  - *Direct addressing:* **MOV AL, [0300H]**

DS	1	2	3	4	0
		0	3	0	0
Memory address	1	2	6	4	0

(assume DS=1234H)

  - *Register indirect addressing:* **MOV AL, [SI]**

DS	1	2	3	4	0
		0	3	1	0
Memory address	1	2	6	5	0

(assume DS=1234H)

(assume SI=0310H)

In-class Exercise

- Consider the byte at address 13DDE within a 64K segment defined by selector value 10DE. What is its offset?

In-class Exercise

- Consider the byte at address 13DFE within 64K segment defined by selector value 10DE. What is its offset?

- 13DDE = 10DE * 16_{10} + offset
- offset = 13DFE - 10DE0
- offset = 301E (a 16-bit quantity)

Address Computation Example

- Consider the whole 1MB address space
- Say that we want a 64K segment whose end is 8K from the end of the address space
- The address at the end of the address space is FFFFF
- 8K in binary is 10-0000-0000-0000, that is 02000 in hex
- So the address right after the end of the segment is
  FFFFF - 02000 + 1 = FEFFF + 1 = FE000
- The length of the segment is 64K
- 64K in binary is 1-0000-0000-0000-0000, that is 10000
- So the address at the beginning of the segment is
  FF000 - 10000 = EF000
- So the value to store in a segment register is EF00
- To reference the 43th byte in the segment, one must store 002A (= 42_{10}) in an index register
- The address of that byte is: EF000 + 002A = EF02A
- The address of the last byte in the segment is: EF000 + 07FFF = F6FFF
  - Which is right before FF000, the beginning of the last 8K of the address space

Instructions Format and Compilation

Developing software for the personal computer .ASM file

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
;
.MODEL SMALL
.STACK
.CODE
.STARTUP
MOV     AX,40H                  ;set AX to 0040H
MOV     DS,AX                   ;load data segment with 0040H
MOV     SI,17H                  ;load SI with 0017H
AND     BYTE PTR [SI],0DFH      ;clear NUM-LOCK bit
.EXIT
END
```

All characters following a ";" till the line end are "comments", ignored by the assembler

Assembler reserved words

Assembly language instructions

Developing software for the personal computer .ASM file

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
;
        .MODEL SMALL
        .STACK
        .CODE
        .STARTUP
        MOV     AX,40H          ;set AX to 0040H
        MOV     DS,AX           ;load data segment with 0040H
        MOV     SI,17H          ;load SI with 0017H
        AND     BYTE PTR [SI],0DFH  ;clear NUM-LOCK bit
        .EXIT
        END
```

Register pair (16 bit) (destination of "MOV")

Hexadecimal value to be loaded (source for "MOV")

Data Segment register pair

Prepare the Data Segment

Source Index

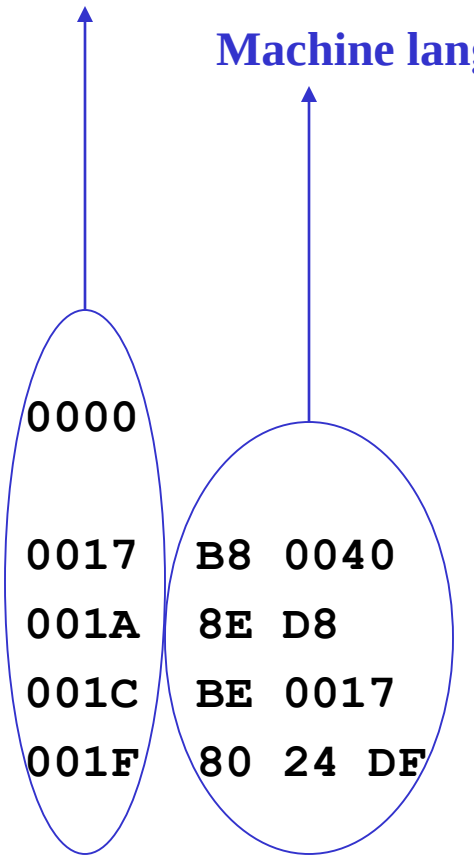
The complete address of the byte containing NumLock bit is specified.

Second operand for logical "AND" (immediate hexadecimal value)

First operand and destination for logical "AND" Memory address specified by DS and SI together.

ANDing with DFH=1101.1111B, only b5 (bit 5) of specified memory location is affected (reset to 0)

1.7 Developing software for the personal computer .LST file




```
                        ;NUMOFF.ASM: Turn NUM-LOCK indicator off.
                        ;
                        .MODEL SMALL
                        .STACK
0000                    .CODE
                        .STARTUP
0017  B8 0040           MOV  AX,40H        ;set AX to 0040H
001A  8E D8             MOV  DS,AX         ;load data segment with 0040H
001C  BE 0017           MOV  SI,17H        ;load SI with 0017H
001F  80 24 DF          AND  BYTE PTR [SI],0DFH   ;clear NUM-LOCK bit
                        .EXIT
                        END
```

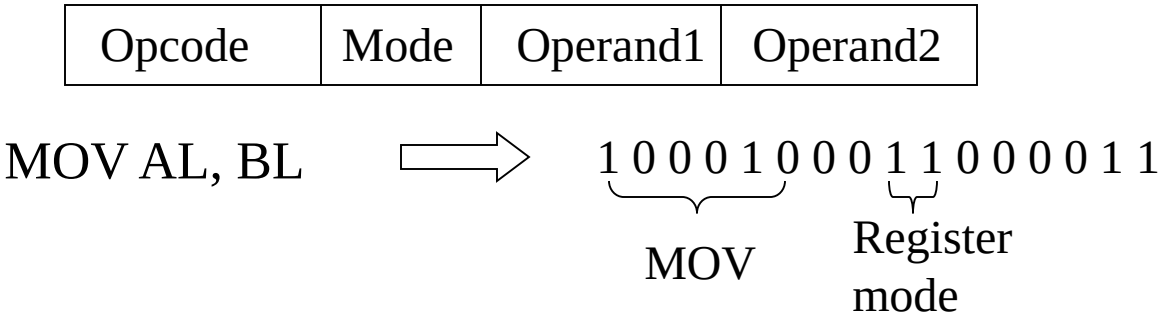
Instruction Format

- General Format of Instructions

Label: Opcode Operands ; Comment

- **Label**: It is optional. It provides a symbolic address that can be used in branch instructio
- **Opcode:** It specifies the type of instructions
- **Operands**: Instructions of 80x86 family can have one, two, or zero operand
- **Comments**: Only for programmers' reference

- Machine Code Format



Assembler Directives

❑ Source File

```
DATA    SEGMENT PARA 'DATA‘
                ORG     7000H
                POINTS  DB      16 DUP(?)
                SUM     DB      ?
DATA    ENDS

CODE    SEGMENT PARA 'CODE‘
                ASSUME CS:CODE, DS:DATA
                ORG     8000H
TOTAL:          MOV     AX,7000H
                MOV     DS,AX
                MOV     AL,0
              •••••••••
CODE    ENDS
END TOTAL
```

❑ List File

```
0000                    DATA    SEGMENT PARA 'DATA’
                        ORG     7000H
7000  0010 [00]         POINTS  DB      16 DUP(?)
7010          00        SUM     DB      ?
7011                    DATA    ENDS

0000                    CODE    SEGMENT PARA 'CODE'
                        ASSUME CS:CODE, DS:DATA
                        ORG     8000H
8000  B8 7000  TOTAL:   MOV     AX,7000H
8003  8E D8    MOV     DS,AX
8005  B0 00    MOV     AL,0
                •••••••••
```

Assembler Directives

- SEGMENT directive
- ENDS directive
- END directive
- ORG directive
- DB: Define Byte; DW, ….
- ASSUME directive

— Specifies the segment register (segment Register) that will be used to calculate the effective addresses for all labels and variables defined under a given segment or group name (segment Name).

If CS = 1230H and DS = 5678H, what are the physical memory addresses of label TOTAL and variable SUM?

Assembler Directives

❑ Simplified Segment Directives

```
 .MODEL SMALL
            .DATA
            ORG     7000H
POINTS  DB      16 DUP(?)
SUM     DB      ?

.CODE
            ORG     8000H
TOTAL:      MOV     AX,7000H
            MOV     DS,AX
            MOV     AL,0
            ••••••••
            RET
 END     TOTAL
```

❑ Predefined .MODEL Types

	DATA SEGMENT	CODE SEGMENT
TINY	one	one
SMALL	one	one
MEDIUM	one	multiple
COMPACT	multiple	one
LARGE	multiple	multiple
HUGE	multiple	multiple
FLAT*	one	one

* Flat is used for 32-bit addressing