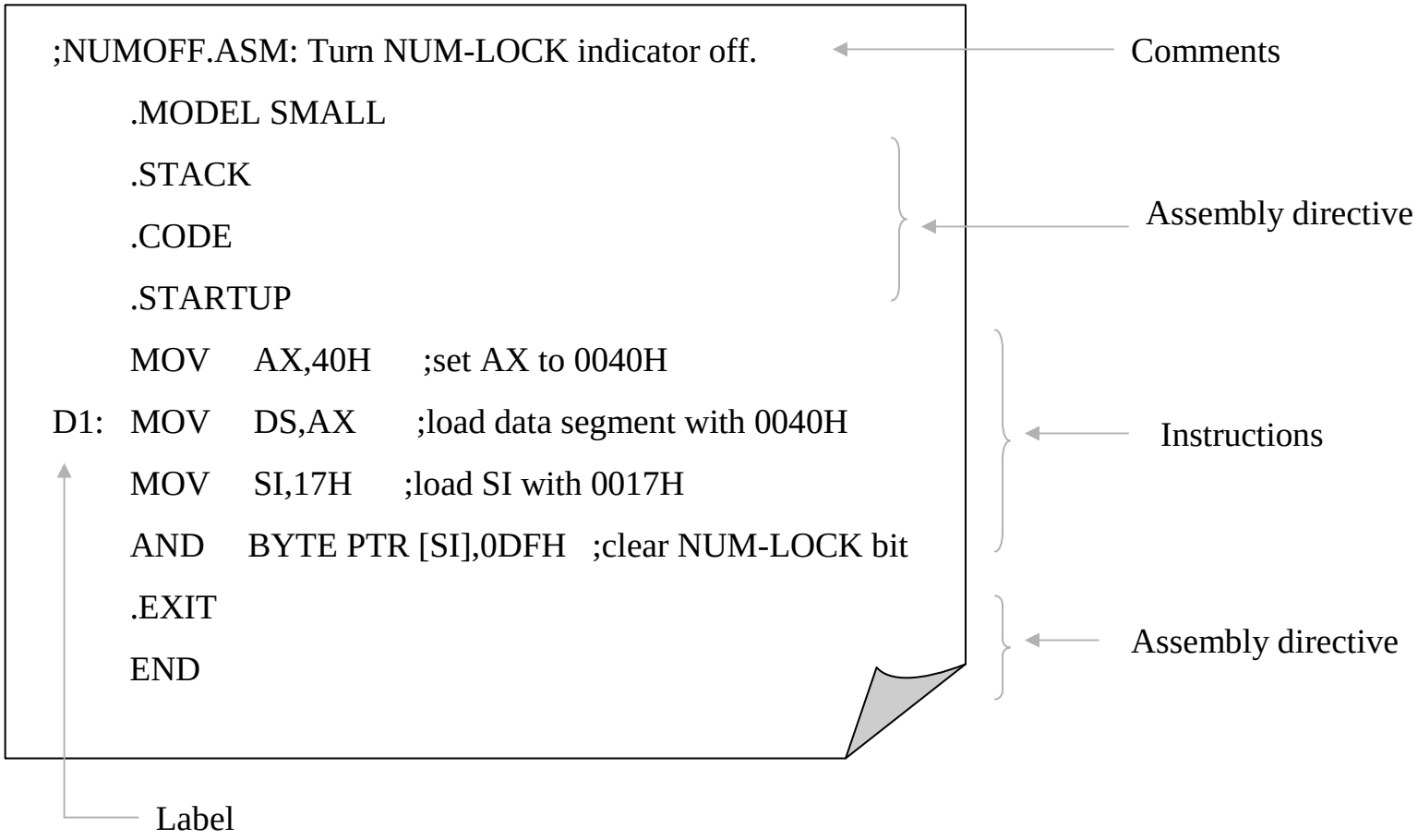
ENCS 238 Computer Organization & Assembly Language

Assembly Language

Computer Systems Eng. Department
Birzeit University

Example of Assembly Language Program



8086 Programming

- 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

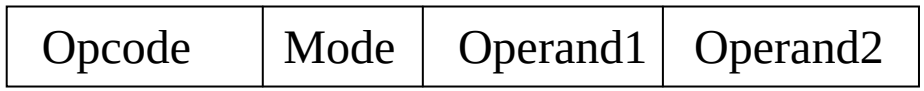
Instruction Format

- General Format of Instructions

Label: Opcode Operands ; Comment

- **Label**: It is optional. It provides a symbolic address that can be used in branch instructic
- **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



MOV AL, BL ⇨ 1 0 0 0 1 0 0 0 1 1 0 0 0 0 1 1

MOV — Register mode

8086 Programming (cont.)

- Instruction form
  - Op-code Destination Operand, Source Operand
  - MOV AX,100
- Variable declarations
  - Variable_Name Memory_Directive Value
  - Var1 DB 7

Multi-valued Variables

- The variables defined as db means each value is defined as bytes.
- However, there is no restriction on how many values we can define for each variable names.

multivar db 12h, 34h, 56h, 78h, 00h, 11h, 22h, 00h

Address	Value
100h	012h
101h	034h
102h	56h
103h	78h
104h	00h
105h	11h
106h	22h
107h	00h

multi valued variables

- So multi valued variables are stored contiguously.

  multivar2 dw 1234h, 5678h, 0011h, 2200h

Address	Value
100h	034h
101h	012h
102h	78h
103h	56h
104h	11h
105h	00h
106h	00h
107h	22h

Using dup

- Another way to declare a multi-valued variables are using dup command:

```
my_array db 5 dup (00h)
```

That example above is similar to:

```
my_array db 00h, 00h, 00h, 00h, 00h
```

dup is kind of shortcut to define variables with the same values.

- Of course you can define something like this:

```
bar_array db 10 dup (?)
```

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

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?

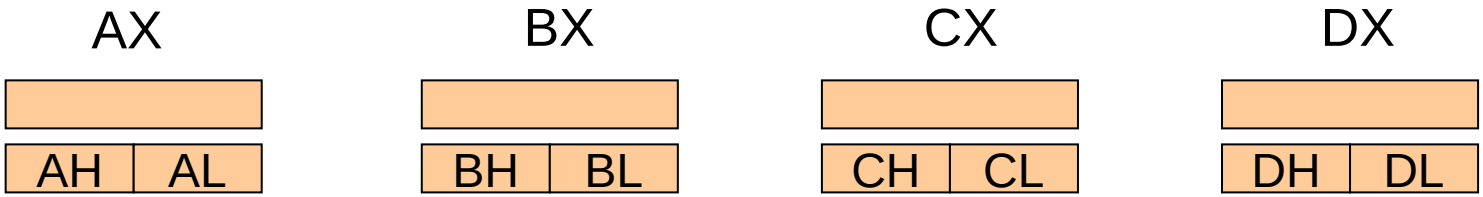
The 8086 Registers

- 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

General purpose registers

- AX, BX, CX, and DX: They can be assigned to any value you want.
  - **AX** (**accumulator register**). Most of **arithmetical** operations are done with AX.
  - **BX** (**base register**). Used to do **array** operations. BX is usually worked with other registers like **SP** to point to stacks.
  - **CX** (**counter register**). Used for **counter** purposes.
  - **DX** (**data register**). Used for storing **data** value.

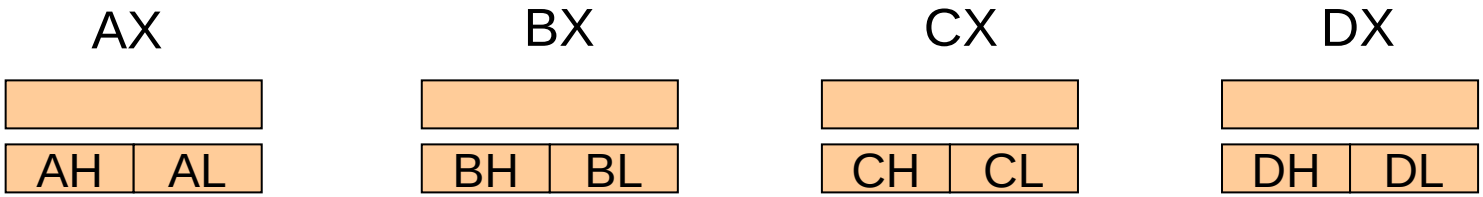
The 8086 Registers



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

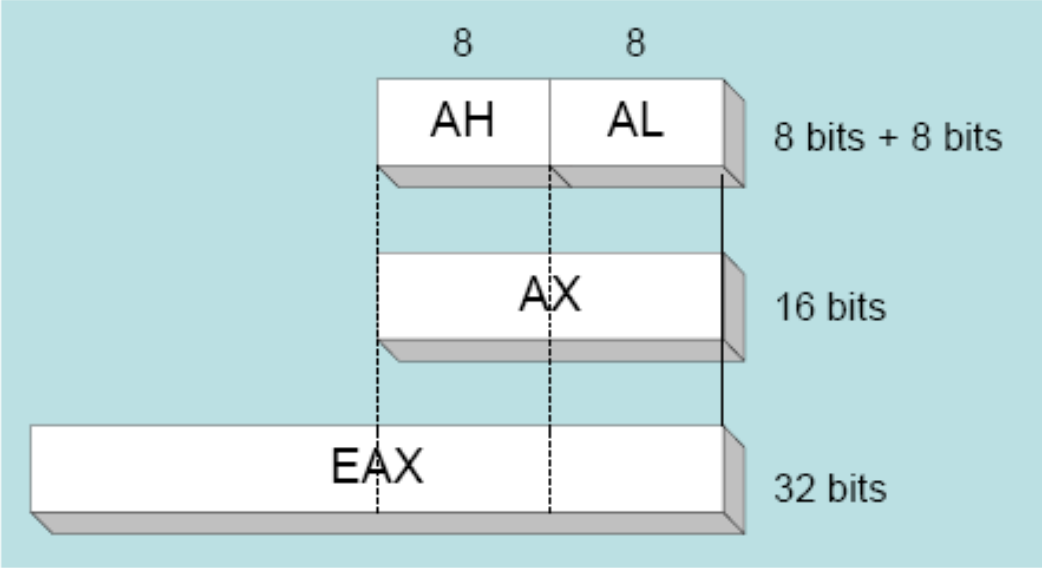
- The ISA makes it possible to refer to the low or high bits individually
  - AH, AL
  - BH, BL
  - CH, CL
  - DH, DL

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

Index registers

- SI and DI: Usually used to process arrays or strings:
  - **SI** (**source index**) is always pointed to the **source array**
  - **DI** (**destination index**) is always pointed to the **destination array**.

Segment registers

- CS, DS, ES, and SS:
  - **CS** (**code segment register**). Points to the segment of the running program. We may **NOT** modify CS directly.
  - **DS** (**data segment register**). Points to the segment of the data used by the running program. You can point this to anywhere you want as long as it contains the desired data.
  - **ES** (**extra segment register**). Usually used with **DI** and doing pointers things. The couple ***DS:SI*** and ***ES:DI*** are commonly used to do string operations.
  - **SS** (**stack segment register**). Points to stack segment.

Pointer registers

- BP, SP, and IP:
  - **BP** (**base pointer**) used for preserving space to use local variables.
  - **SP** (**stack pointer**) used to point the current stack.
  - **IP** (**instruction pointer**) denotes the current pointer of the running program. It is always coupled with **CS** and it is **NOT modifiable**. So, the couple of ***CS:IP*** is a pointer pointing to the current instruction of running program. You can **NOT** access ***CS*** nor ***IP*** directly.

Addresses in Memory

- We mentioned several registers that are used for holding addresses of memory locations
- Segments:
  - CS, DS, SS, ES
- Pointers:
  - SI, DI: indices (typically used for pointers)
  - SP: Stack pointer
  - BP: (Stack) Base pointer

- Let’s look at the structure of the address space

Address Space

- In the 8086 processor, a program is limited to referencing an address space of size 1MB, that is 2^{20} bytes
- Therefore, addresses are 20-bit long!
- A d-bit long address allows to reference 2^d different “things”
- Example:
  - 2-bit addresses
    - 00, 01, 10, 11
    - 4 “things”
  - 3-bit addresses
    - 000, 001, 010, 011, 100, 101, 110, 111
    - 8 “things”
- In our case, these things are “bytes”
  - One cannot address anything smaller than a byte
- Therefore, a 20-bit address makes it possible to address 2^{20} individual bytes, or 1MB

Address Space

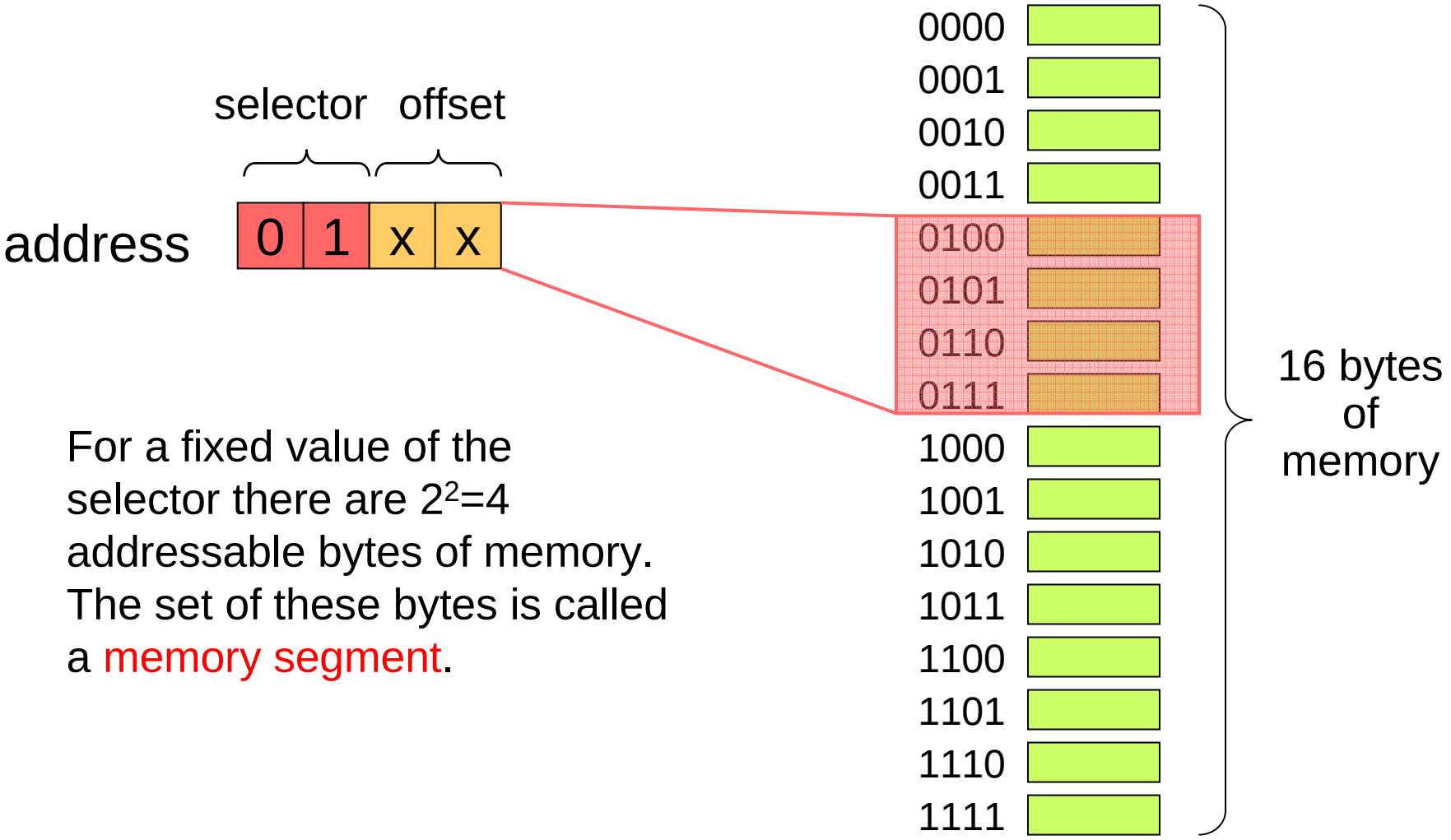
- One says that a running program has a 1MB address space
- And the program needs to use 20-bit addresses to reference memory content
  - Instructions, data, etc.
- Problem: registers are at 16-bit long! How can they hold a 20-bit address???
- The solution: split addresses in two pieces:
  - The selector
  - The offset

Simple Selector and Offset

- Let us assume that we have an address space of size 2^4=16 bytes
  - Yes, that would not be a useful computer
- Addresses are 4-bit long
- Let's assume we have a 2-bit selector and a 2-bit offset
  - As if our computer had only 2-bit registers

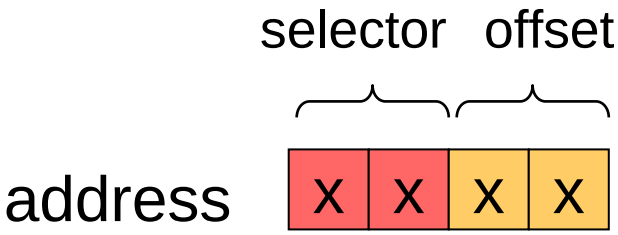
- We take such small numbers because it's difficult to draw pictures with 2^{20} bytes!

Selector and Offset Example

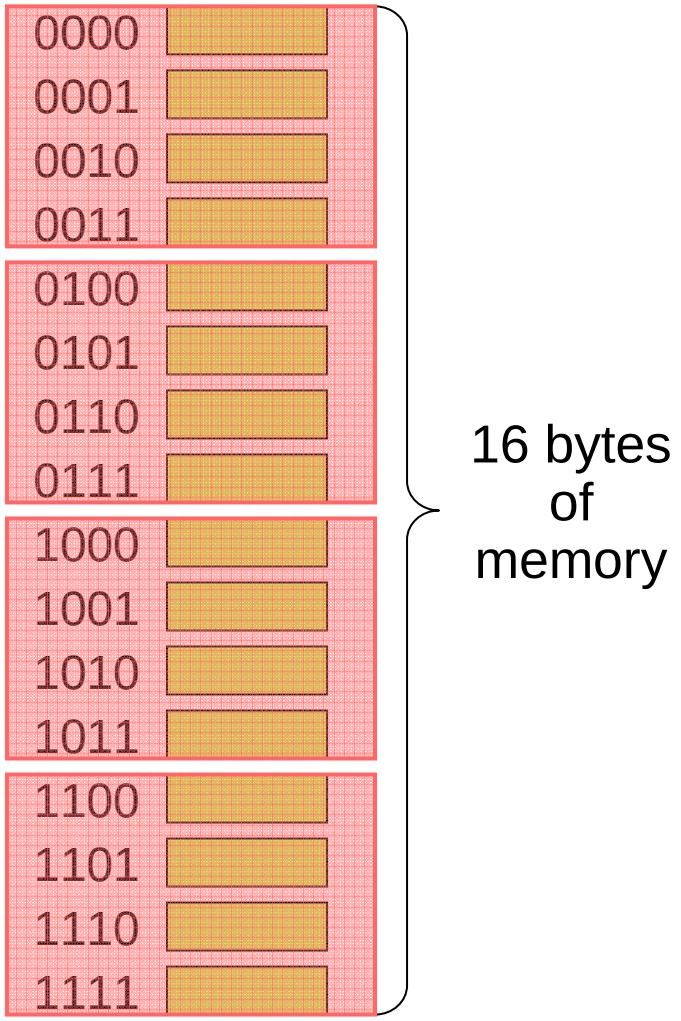


For a fixed value of the selector there are $2^2=4$ addressable bytes of memory. The set of these bytes is called a memory segment.

Selector and Offset Example



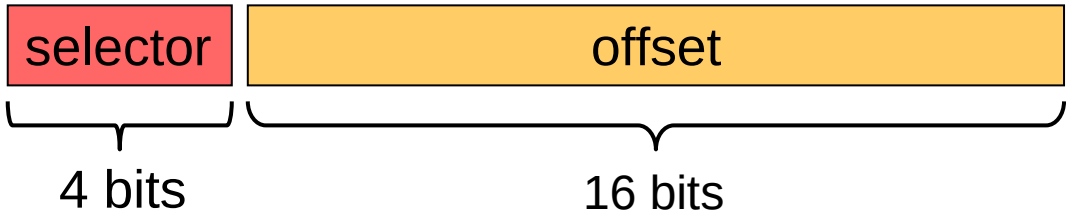
We have 16 bytes of memory
We have 4-byte segments
We have 4 segments



Selector and Offset

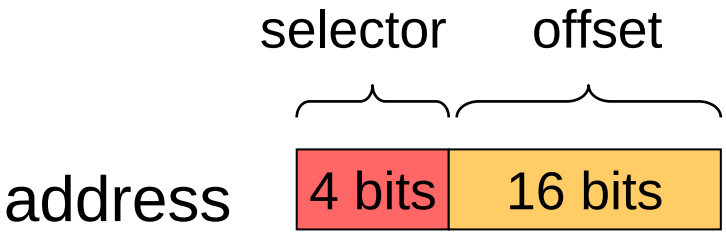
- The way in which one addresses the memory content is then pretty straightforward
- First, set the bits of the selector to “pick” a segment
- Second, set the bits of the offset to address a byte within the segment
- This all makes sense because a program typically addresses bytes that are next to each other, that is within the same segment
- So, the selector bits stay the same for a long time, while the offset bits change often
  - Of course, this isn’t true for tiny 4-byte segments as in our example…

For 20-bit Addresses

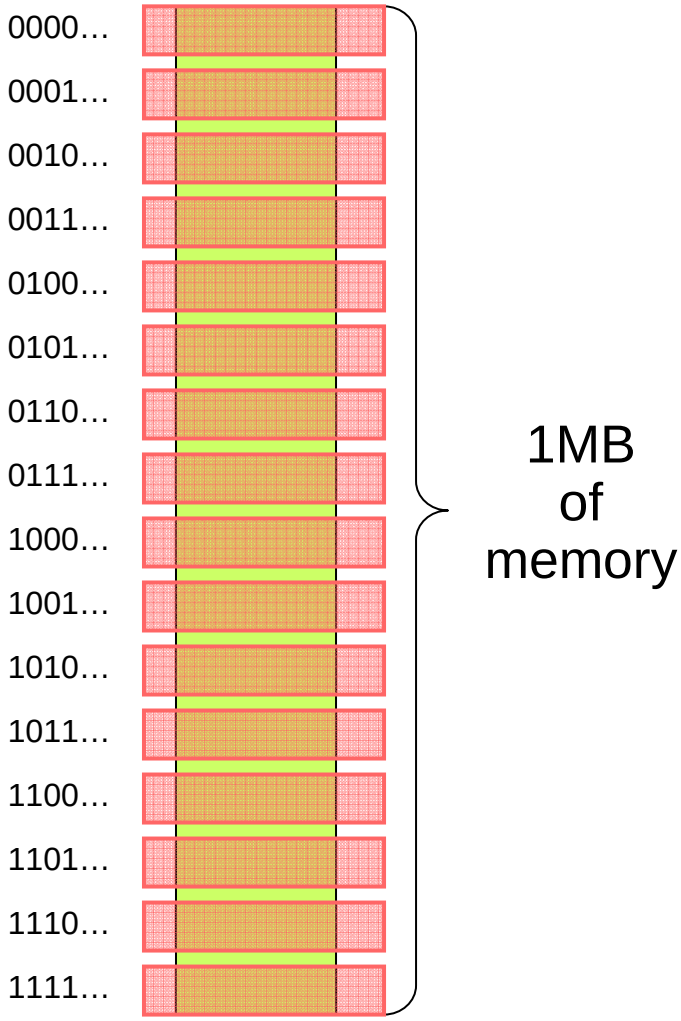


- On the 8086 the offset if 16-bit long
  - And therefore the selector is 4-bit
- We have $2^4 = 16$ different segments
- Each segment is 2^{16} byte = 64KB
- For a total of 1MB of memory, which is what the 8086 used

For 20-bit Addresses



We have 1MB of memory
We have 64K segments
We have 16 segments



The 8086 Selector Scheme

- So far we've talked about the selector as a 4-bit quantity, for simplicity
- This leads to 16 non-overlapping segments
- The designers of the 8086 wanted more flexibility
- E.g., if you know that you need only an 8K segment, why use 64K for it? Just have the "next" segment start 8K after the previous segment
  - We'll see why segments are needed in a little bit
- So, for the 8086, the selector is NOT a 4-bit field, but rather the address of the beginning of the segment
- But now we're back to our initial problem: Addresses are 20-bit, how are we to store an address in a 16-bit register???

The 8086 Selector Scheme

- What the designers of the 8086 did is pretty simple
- Enforce that the beginning address of a segment can only be a multiple of 16
- Therefore, its representation in binary always has its four lowest bits set to 0
- Or, in hexadecimal, its last digit is always 0
- So the address of a beginning of a segment is a 20-bit hex quantity that looks like: XXXX0
- Since we know the last digit is always 0, no need to store it
- Therefore, we need to store only 4 hex digits
- Which, lo and behold, fits in a 16-bit register!

The 8086 Selector Scheme

- So now we have two 16-bit quantities
  - The 16-bit selector
  - The 16-bit offset
- The selector must be stored in one of the “segment” registers
  - CS, DS, SS, ES
- The offset is typically stored in one of the “index” registers
  - SI, DI
  - But could be stored in a general purpose register
- Address computation is straightforward
- Given a 16-bit selector and a 16-bit offset, the 20-bit address is computed as follows
  - Multiply the selector by 16
    - This simply transforms XXXX into XXXX0, thanks to the beauty of hexadecimal
  - Add the offset
  - And voila

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 13DDE within a 64K segment defined by selector value 10DE. What is its offset?

- 13DDE = 10DE * 16_{10} + offset
- offset = 13DDE - 10DE0
- offset = 2FFE (a 16-bit quantity)

Extended register

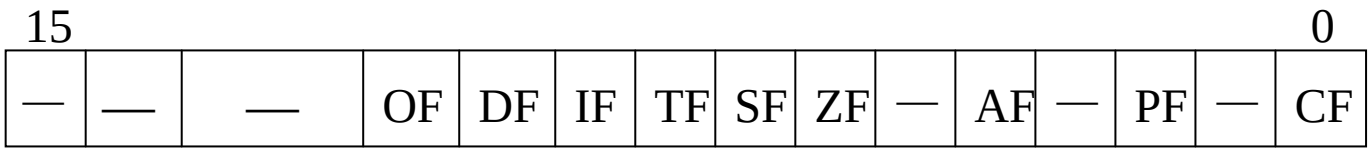
- 386 processors introduce extended register.
- Most of the registers, except segment registers are enhanced into 32-bit.
- So, we have extended registers *EAX*, *EBX*, *ECX*, and so on.
- *AX* is only the low 16-bit (bit 0 to 15) of *EAX*.
- There are NO special direct access to the upper 16-bit (bit 16 to 31) in extended register.

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

Flag register

- *Flag* is 16-bit register that contains processor status.
- It holds the value of which the programmers may need to access. This involves detecting whether the last arithmetic holds zero result or may be overflow.
- Intel doesn't provide a direct access to it; rather it is accessed via stack. (via POPF and PUSHF)
- You can access each flag attribute by using bitwise AND operation since each status is mostly represented by just 1 bit.

Flag register cont.

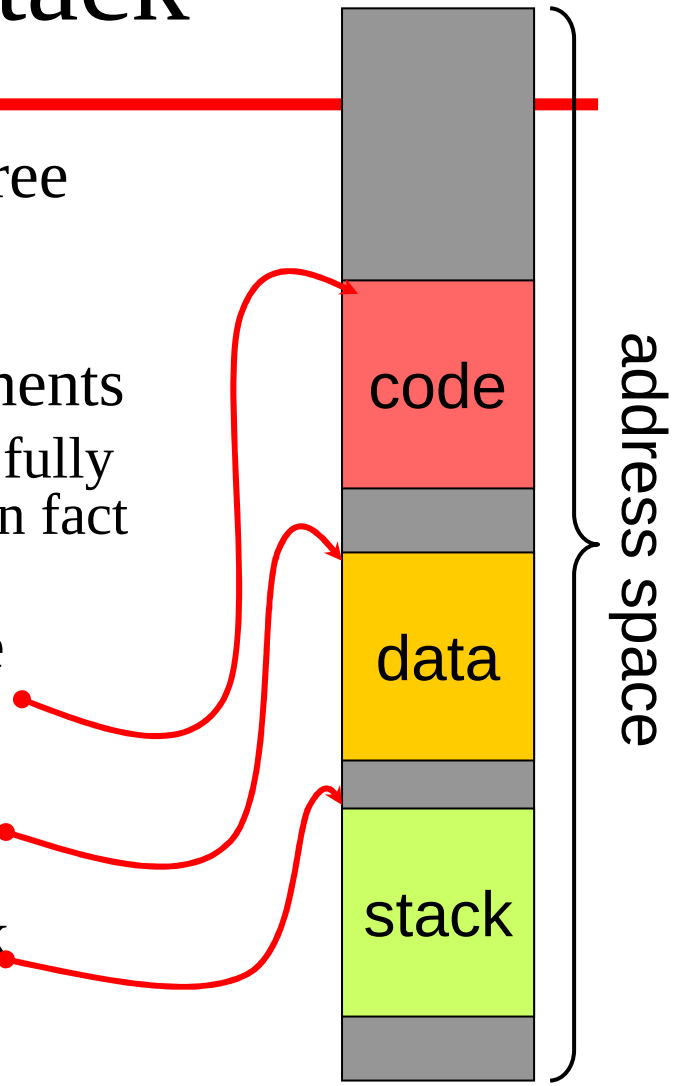
- C *carry flag* is turned to 1 whenever the last arithmetical operation, such as adding and subtracting, has *carry* or borrow otherwise 0.
- P *parity flag* It will set to 1 if the last operation (any operation) results even number of bit 1.
- A *auxilarry flag* It is set in Binary Coded Decimal (BCD) operations.
- Z *zero flag* used to detect whether the last operation (any operation) holds zero result.
- S *sign flag* used to detect whether the last operation holds negative result. It is set to 1 if the highest bit (bit 7 in bytes or bit 15 in words) of the last operation is 1.

Flag register cont.

- T *trap flag* used in debuggers to turn on the step-by-step feature.
- I *interrupt flag* used to toggle the interrupt enable or not. If the bit is set (= 1), then the interrupts are enabled, otherwise disabled. The default is on.
- D *direction flag* used for directions of string operations. If the bit is set, then all string operations are done backward. Otherwise, forward. The default is forward (= 0).
- O *the overflow flag* used to detect whether the last arithmetic operation result has overflowed or not. If the bit is set, then it has been an overflow.

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



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)

Example Using Shifts

- Say you want to count the number of bits that are equal to 1 in register EAX
- One easy way to do this is to use shifts
  - Shift 32 times
  - Each time the carry flag contains the last shifted bit
  - If the carry flag is 1, then increment a counter, otherwise do not increment a counter
  - When you're done the counter contains the number of 1's
- Let's write this in x86 assembly

Example Using Shifts

```
; Counting 1 bits in EAX
mov     bl, 0           ; bl is the number of 1 bits
mov     cl, 32          ; cl is the loop counter
loop_start:
    shl   eax, 1        ; left shift
    jnc   not_one       ; if carry != 1, jump to not_one
    inc bl              ; increment the number of 1 bits
not_one:
    dec cl              ; decrement the loop counter
    jnz loop_start      ; if more iterations goto loop_start
```

Find average of two numbers

```
.model small
.stack 100
.data
    No1             DB 63H          ; First number storage
    No2             DB 2EH          ; Second number storage
    Avg         DB ?    ; Average of two numbers
.code
START:
MOV AX,@data                    ; [ Initialises
    MOV DS,AX                   ;  data segment ]
    MOV AL,NO1                  ; Get first number in AL
    ADD AL,NO2                  ; Add second to it
    ADC AH,00H                  ; Put carry in AH
    SAR AX,1            ; Divide sum by 2
    MOV Avg,AL                  ; Copy result to memory
```

Find sum of numbers in the array

```
.model small
.data
        ARRAY   DB 12H,24H,26H,63H,25H,86H,2FH,33H,10H,35H
        SUM     DW 0
.code
START:
MOV AX,@data    ; [ Initialise
        MOV DS,AX       ;   data segment ]
        MOV CL,10       ; Initialise counter
        XOR DI,DI       ; Initialise pointer
        LEA BX,ARRAY    ; Initialise array base pointer
BACK:
         MOV AL,[BX+DI]  ; Get the number
        MOV AH,00H      ; Make higher byte 00h
        ADD SUM,AX      ; SUM = SUM + number
        INC DI          ; Increment pointer
        DEC CL          ; Decrement counter
        JNZ BACK        ; if not 0 go to back
       END STAR
```