PC Environment

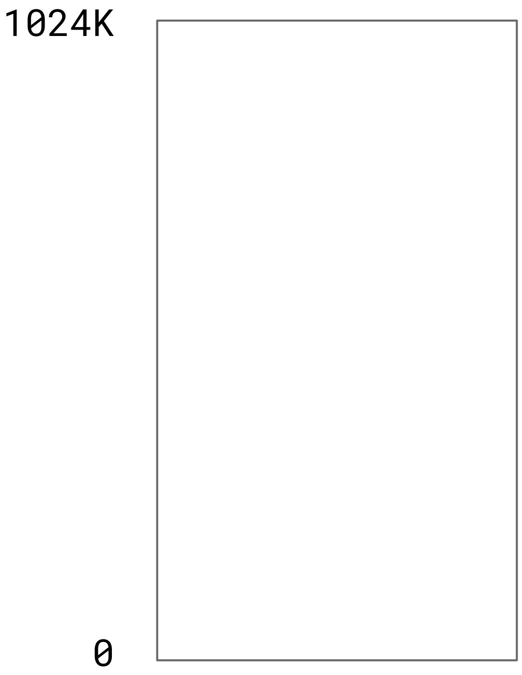
System view

- Hardware Architecture
    - Number systems
    - CPU & memory
    - Registers, ALU, buses
    - Fetch-execute cycle

System view

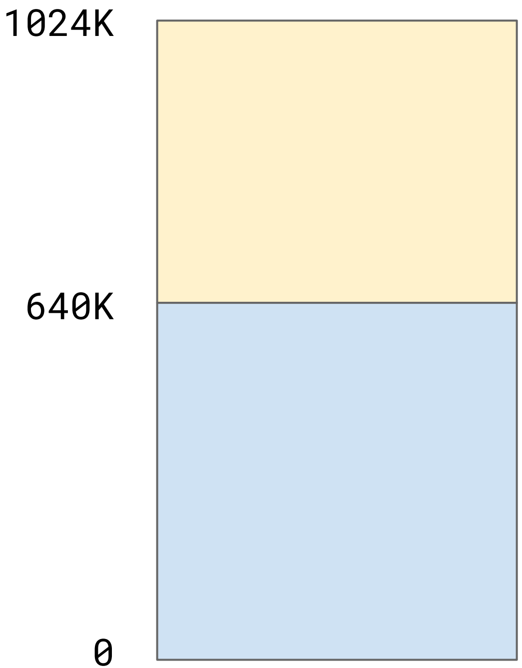
- Software
  - Applications
  - High-level language
  - OS
  - ASM
  - Machine code
- Hardware Architecture
  - Number systems
  - CPU & memory
  - Registers, ALU, buses
  - Fetch-execute cycle

8086/DOS environment

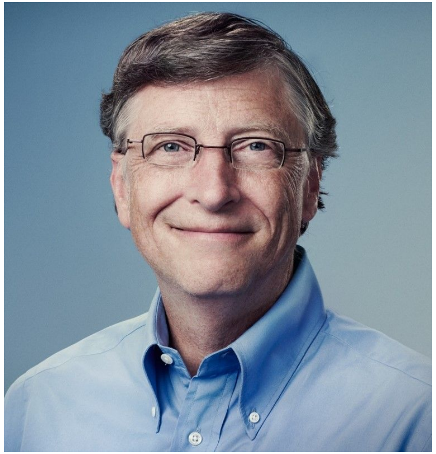


- **8086 can address 1MB of memory**
- **1MB = 2^{20} bytes**
- **5 hex digits are 20 bits**
- **Memory addresses range from**
  - `00000 to`
  - `FFFFF`
- $FFFFF_{16} + 1 = 2^{20} = 1M = 1024K$

8086/DOS environment

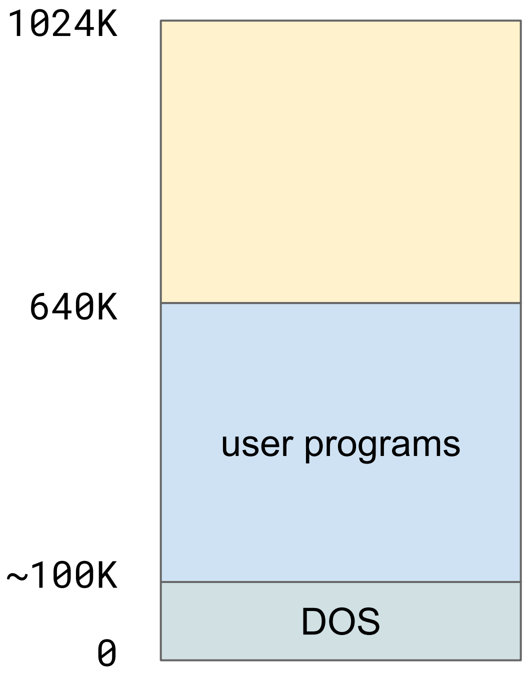


- DOS can only access
  - 640KB

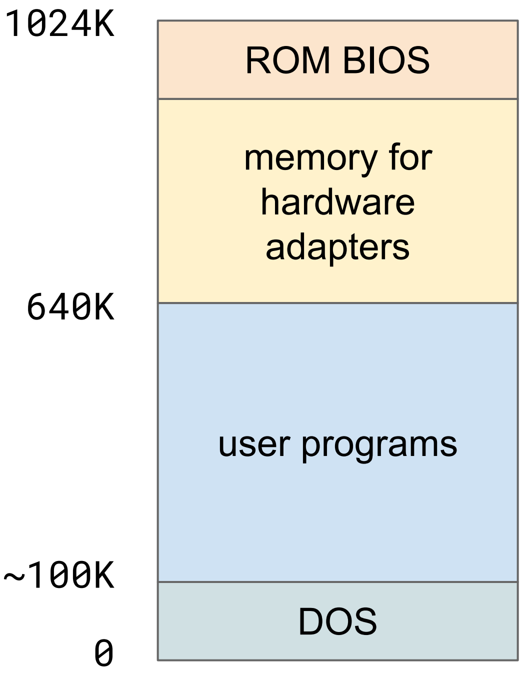


“640KB ought to be enough for anybody”

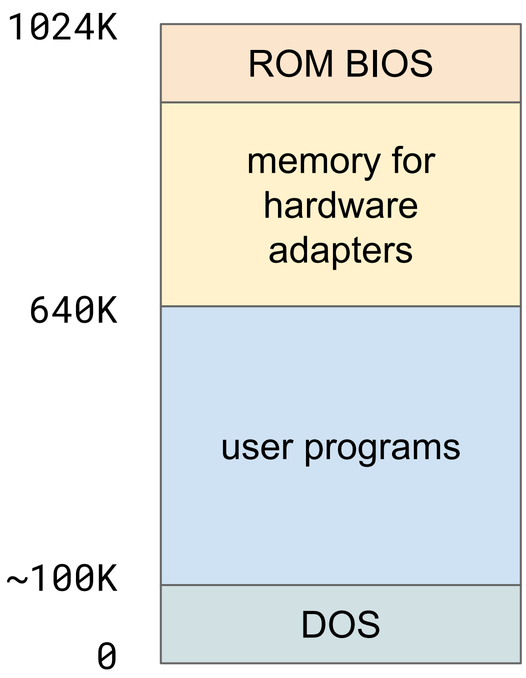
8086/DOS environment



8086/DOS environment

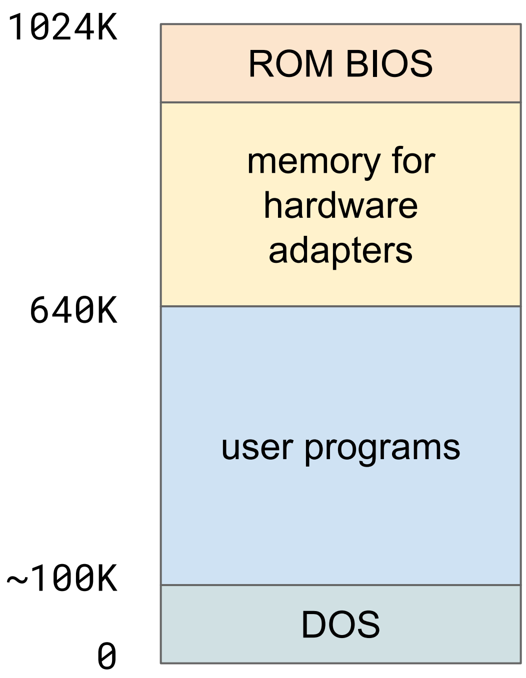


8086/DOS environment



- Memory mapped IO
- Display adapter
  - B0000
  - In DOS
- Write ‘A’ to memory address B0000
  - Write goes to adapter memory
  - Not to main memory
- Much faster than system call

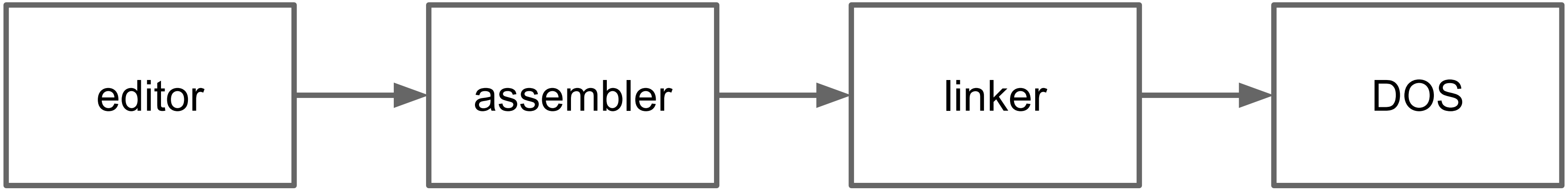
8086/DOS environment



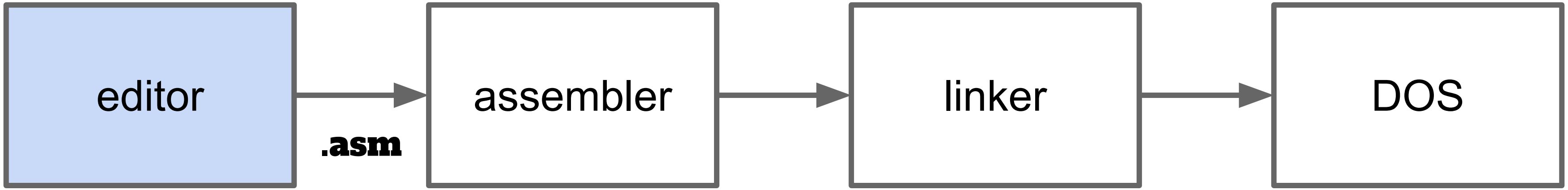
- Memory mapped I/O
- Faster
  - 10,000s of instructions for sys call
  - 1 instruction for write (MOV)
- To update display
  - Thousands of characters on screen
  - Times 10,000s of instructions
  - Way too slow to for anything like a 60Hz frame rate

Creating a program

Steps to create an executable program

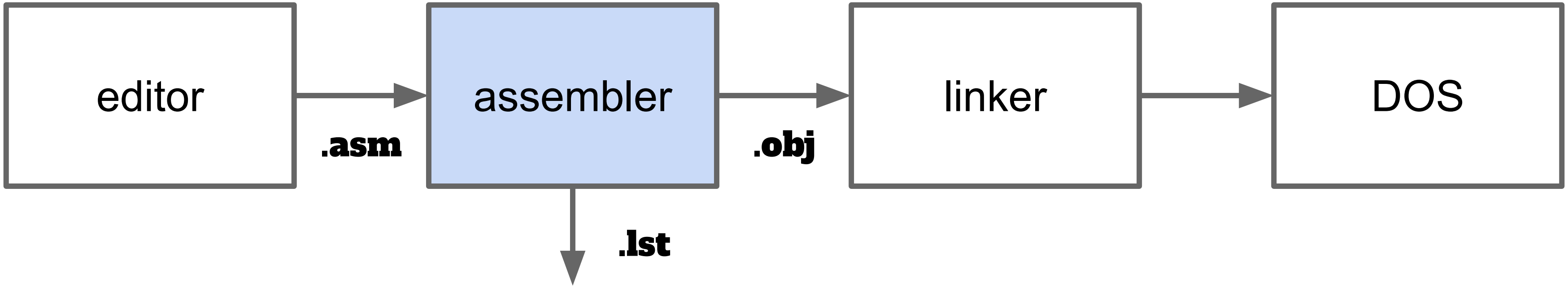


Steps to create an executable program



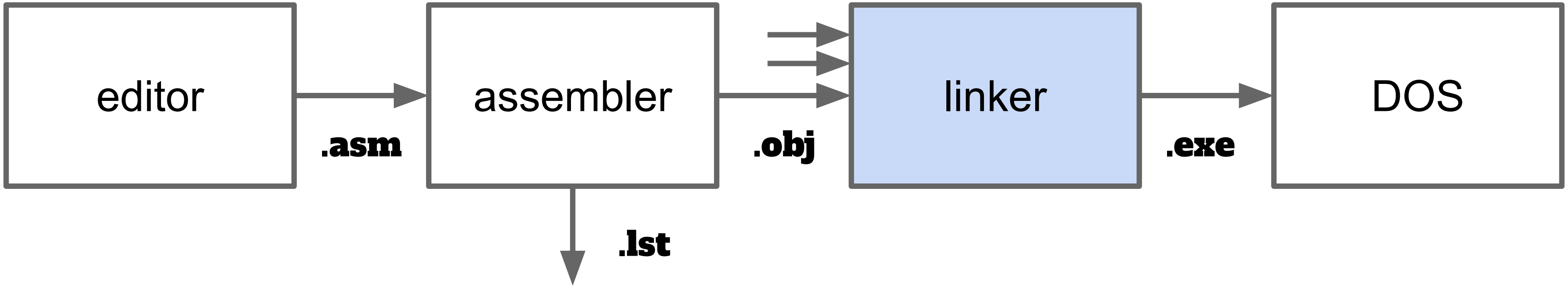
- Editor ⇒ prog.asm
  - Note: DOS is not case sensitive
  - foo.asm = FOO.ASM = fOo.AsM

Steps to create an executable program



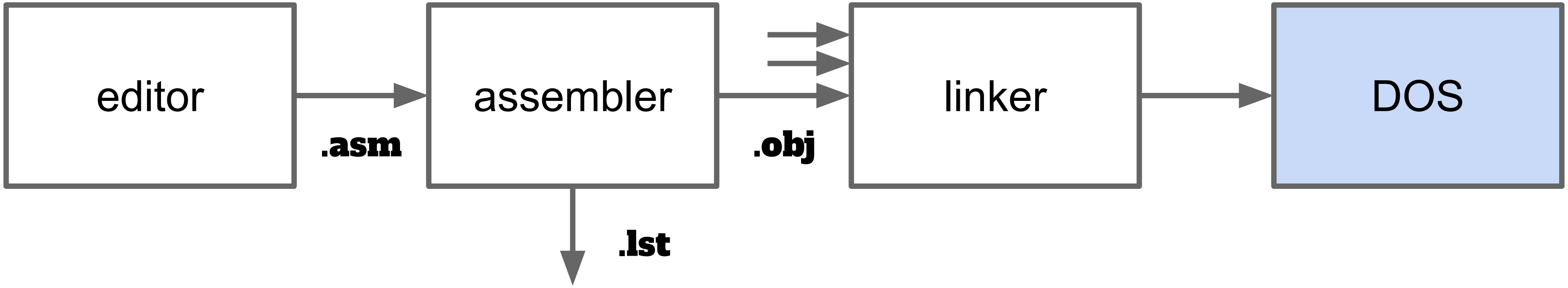
- Assembler
    - Input: foo.asm (assembly language source)
    - Output: foo.obj (relocatable object file)
    - Output: foo.lst (listing — human readable)

Steps to create an executable program



- Linker
  - Input: foo.obj, … (multiple object files)
  - Output: foo.exe (executable file)

Steps to create an executable program



- Executable file
  - Load: foo.exe
  - Execute at starting address
  - You got yourself a running application

Documentation

- What is purpose of documentation

To humor instructor or manager

Documentation

- What is purpose of documentation
- To help someone (including yourself) understand the code
  - To fix bugs
  - To add functionality
- CSC236
  - Documentation requirements
  - Part of grade
  - See web page

Staff will not read undocumented code.
If you want help with code, first document it

Assembly language

Line of code

label: opcode   dest,source   ;comment

- Each line may have these 4 components

Line of code

label: opcode dest,source ;comment

- label
  - 1 to 35 alphanumeric chars
  - First char is [a-zA-Z_@$]
  - Not case sensitive
  - Must end in colon (:)
  - No spaces
- Target of GOTO
- Some instructions
  - Can change control flow
  - Allows for looping and branching
- No high-level concepts in ASM
  - No while loops
  - No if-then-else statements

Line of code

label: opcode   dest,source   ;comment

- ;comment
  - Begins with semicolon (;)
  - Extends to end of line
  - Required in CSC236
  - Every line will have a comment
  - Say what you're trying to do
    - This may be non-obvious in assembly language.

Comments

```
;-------------------------------
; The block header describes what
; the code in the block does
;-------------------------------
calc_val:
          add       ax,bx                  ; comment
          sub       cx,[var]               ; comment

;-------------------------------
; The block header describes what
; the code in the block does
;-------------------------------
calc_two:
          shl       [grade],1              ; comment
          inc       dx                     ; comment
          div       [two]                  ; comment

0         10        20                     40
```

You'll have a block comment for each block of code (body of a loop, body of an if statement, start of the program, etc.)

You'll have a little comment for each line of code.

Line of code

label: opcode dest,source ;comment

- opcode
  - Specifies the instruction
  - For example: add, sub, mov, cmp
- dest,source
  - Supplies data (optional: 0, 1, or both)
  - Multiple forms
    - Register
    - Memory location
    - Immediate value

- **addax,1000**
- **dec bx**
- **cbw**

Line of code

label: opcode dest,source ;comment

- opcode
  - Specifies the instruction
  - For example: add, sub, mov, cmp
- dest,source
  - Supplies data (optional: 0, 1, or both)
  - Multiple forms
    - Register
    - Memory location
    - Immediate value

Restrictions

- **dest & source must be <u>same</u> size**
- **cannot both be memory references**

More (much more) later

Operand

- Destination & source
- Specified by address mode
  - How to locate the data
  - Register
  - Immediate value
  - Memory
  - Index -- for accessing list or structure
- Effective address calculation
  - Memory
  - Actual location of data
  - Depends on addressing mode

Numbers

- Decimal (default)
  - 100
  - 20
- Hexadecimal
  - Ends with ‘h’ or ‘H’
  - Begins with decimal digit [0-9]
  - 64h
  - 0A5h

The leading zero here would be easy to forget.

Coding

mov — “move”

- Copies the source to dest
- Source does <u>not</u> change
- Restrictions
  - Cannot move immediate data into a segment register
    - mov ds, 1000 ✘
    - mov ds, ax ✔
  - Cannot move data between two segment registers
    - mov ds, es ✘
- Does not set condition codes

Address modes

- Register direct
  - ax
  - bx
  - …
- Immediate
  - 7, 1000
  - 64h, 0A5h
  - ‘A’, ‘bcd’
- Memory direct
  - [var]

Nine combinations

- `mov ax, bx`
- `mov ax, 7`
- `mov ax, [vara]`
- `mov 7, bx`
- `mov 7, 23`
- `mov 7, [vara]`
- `mov [vara], bx`
- `mov [vara], 7`
- `mov [vara], [varb]`

Example: Copy B -> A

- That’s two memory references
  - Restricted to one per instruction
- Data must go through a register

```
mov   ax,[B]    ;load ax w/ B
mov   [A],ax    ;store B into A
```

- **What is the size of A?**
- **ax is a word**
  (if we used al or ah, it would be byte)
- **Therefore A & B are words**

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>   value    ;comment

num      db       1         ;
v1       db       100       ;
v2       db       64h       ;
neg1     db       -1        ;
neg2     db       0FFh      ;
big      db       255       ;
grand    dw       1000      ;
neg3     dw       -1        ;
```

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name    <db,dw>  value   ;comment

num     db       1       ;01
v1      db       100     ;
v2      db       64h     ;
neg1    db       -1      ;
neg2    db       0FFh    ;
big     db       255     ;
grand   dw       1000    ;
neg3    dw       -1      ;
```

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>  value    ;comment

num      db       1        ;01
v1       db       100      ;64
v2       db       64h      ;
neg1     db       -1       ;
neg2     db       0FFh     ;
big      db       255      ;
grand    dw       1000     ;
neg3     dw       -1       ;
```

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>  value    ;comment

num      db       1        ;01
v1       db       100      ;64
v2       db       64h      ;64
neg1     db       -1       ;
neg2     db       0FFh     ;
big      db       255      ;
grand    dw       1000     ;
neg3     dw       -1       ;
```

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name    <db,dw>  value   ;comment

num     db       1       ;01
v1      db       100     ;64
v2      db       64h     ;64
neg1    db       -1      ;FF
neg2    db       0FFh    ;
big     db       255     ;
grand   dw       1000    ;
neg3    dw       -1      ;
```

Declaring variables

- 4 components
  - Name (no ':')
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>  value   ;comment

num      db       1       ;01
v1       db       100     ;64
v2       db       64h     ;64
neg1     db       -1      ;FF
neg2     db       0FFh    ;FF
big      db       255     ;
grand    dw       1000    ;
neg3     dw       -1      ;
```

?

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>   value    ;comment

num      db        1        ;01
v1       db        100      ;64
v2       db        64h      ;64
neg1     db        -1       ;FF
neg2     db        0FFh     ;FF
big      db        255      ;FF
grand    dw        1000     ;
neg3     dw        -1       ;
```

FF,
255 or
-1?

I don’t care

Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name    <db,dw>  value   ;comment

num     db       1       ;01
v1      db       100     ;64
v2      db       64h     ;64
neg1    db       -1      ;FF
neg2    db       0FFh    ;FF
big     db       255     ;FF
grand   dw       1000    ;E8 03
neg3    dw       -1      ;
```

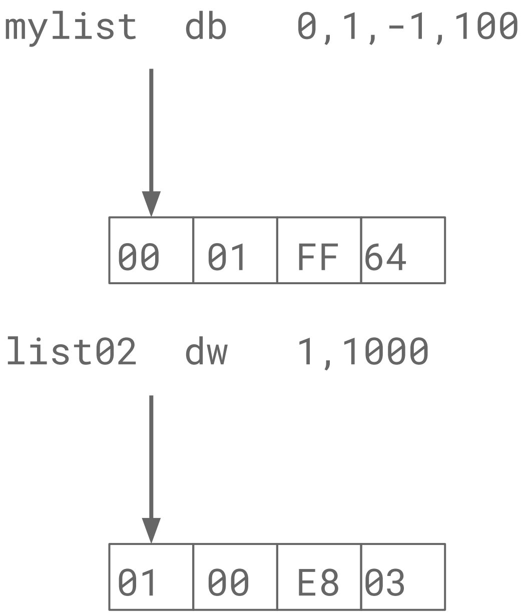
Declaring variables

- 4 components
  - Name (no ‘:’)
  - Size — byte or word
  - Value
  - Comment

```
name     <db,dw>  value   ;comment

num      db       1       ;01
v1       db       100     ;64
v2       db       64h     ;64
neg1     db       -1      ;FF
neg2     db       0FFh    ;FF
big      db       255     ;FF
grand    dw       1000    ;E8 03
neg3     dw       -1      ;FF FF
```

Lists

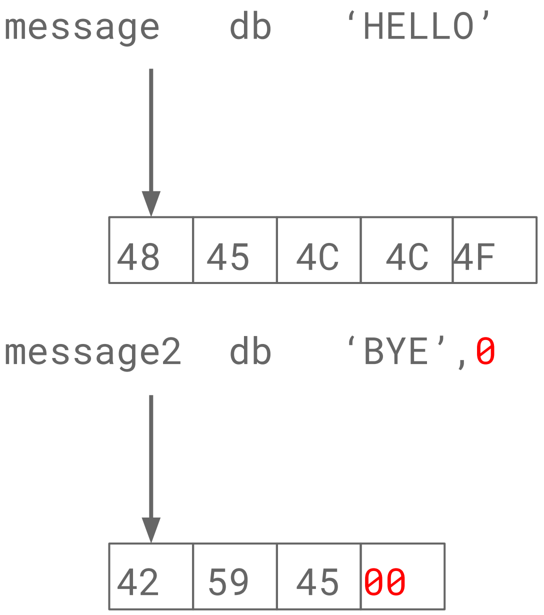
- Can create lists
- Sequences of values one after another in memory
- Essentially, this is array initialization.



Notice the byte swapping in memory.

ASCII data strings

- Characters are ASCII
  - Byte sized
- Strings are sequences of characters
- The assembler will not automatically add null termination.
- You can add it yourself
  - As another value in the sequence.
  - E.g., a null terminator (00)
  - Or, we'll sometimes use '$'.



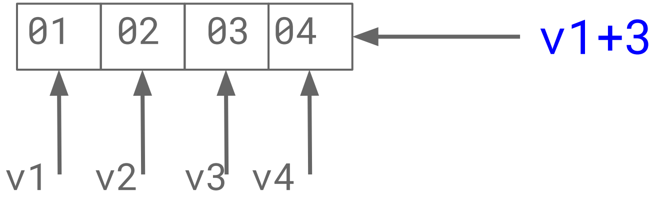
Notes about assembler data

Data is allocated in memory in the same order it is declared

```
v1    db    1
v2    db    2
v3    db    3
v4    db    4
```

Data is not marked as signed or unsigned

Programmer must know and use correctly



Add & Sub

- Syntax
  - <op> dest,source
- Same operand combos as mov
- Both operations set condition codes

```
add dst,src ;dst = dst+src
sub dst,src ;dst = dst-src

add ax,bx
add ax,7
sub ax,[var]
add [var],bx
sub [var],7
```

Calculate C = A+B

```
a    db   10    ;0A
b    db   55    ;37
c    db   00    ;00
```

```
                      AH   AL

mov   al,[a]           ?
add   al,[b]           ?
mov   [c],al           ?
```

Calculate C = A+B

```
a    db   10    ;0A
b    db   55    ;37
c    db   00    ;00
```

```
                      AH   AL

mov   al,[a]           ?   0A
add   al,[b]           ?
mov   [c],al           ?
```

Calculate C = A+B

```
a     db   10     ;0A
b     db   55     ;37
c     db   00     ;00
```

16 hex

进制

		AH	AL
mov	al,[a]	?	0A
add	al,[b]	?	41
mov	[c],al	?	

? =

Calculate C = A+B

```
a    db   10    ;0A
b    db   55    ;37
c    db   00    ;00
                   ;41
```

		AH	AL
mov	al,[a]	?	0A
add	al,[b]	?	41
mov	[c],al	?	41

Are there multiple solutions

Of course

Alternative 1

```
mov   al,[a]      ;al=A
mov   [c],al      ;C =A
mov   bl,[b]      ;bl=B
add   [c],bl      ;C =A+B
```

Alternative 2

```
mov   [c],[b]      ;C =B
add   [c],[a]      ;C =B+A
```

Carry flag and Overflow

- The carry flag should always indicate whether you have an unsigned overflow.
- On addition, it should be set if there's a carry out of the high-order bit.
- On subtraction, it should be set if there's a borrow into the high-order bit.

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
  0001
 -0010
```

```
SF =
ZF =
OF =
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
  0001
 -0010
     1

SF =
ZF =
OF =
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
   1
 0111
 ~~1000~~1
-0010
     1

SF =
ZF =
OF =
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

1
0111
~~1000~~1
-0010
1111

SF =
ZF =
OF =
CF =

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
   1
 0111
 10001
 -0010
  1111

SF = 1
ZF =
OF =
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
   1
 0111
 10001
 -0010
  1111

SF = 1
ZF = 0
OF =
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
    1
  0111
  10001
  -0010
   1111

SF = 1
ZF = 0
OF =
CF =
```

```
   +1        +1
-  +2    +   -2
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
   1
 0111
 10001
 -0010
  1111

SF = 1
ZF = 0
OF = 0
CF =
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
    1
  0111
  10001
  -0010
   1111

SF = 1
ZF = 0
OF = 0
CF =
```

- **CF — equals**
  - **Carry out of left-most bit on add**
  - **Borrow into left-most bit on sub**

How is CF set on subtraction

- Set to 1 if unsigned overflow

```
    1
 0111
 10001
-0010
  1111
```

```
 0001    A
 1101   !B
+   1
```

```
SF = 1
ZF = 0
OF = 0
CF = 1
```

How is CF set on subtraction

- Set to 1 if unsigned overflow

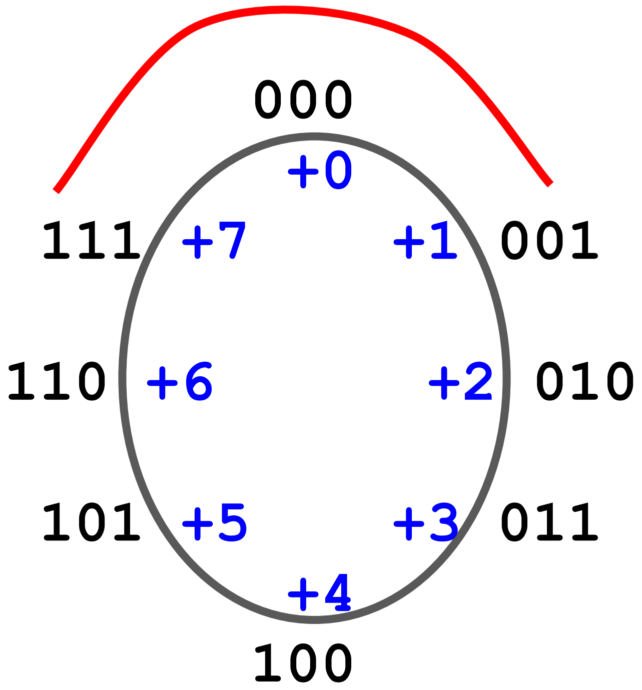
```
   1
 0111
 10001
 -0010
  1111

SF = 1
ZF = 0
OF = 0
CF = 1
```

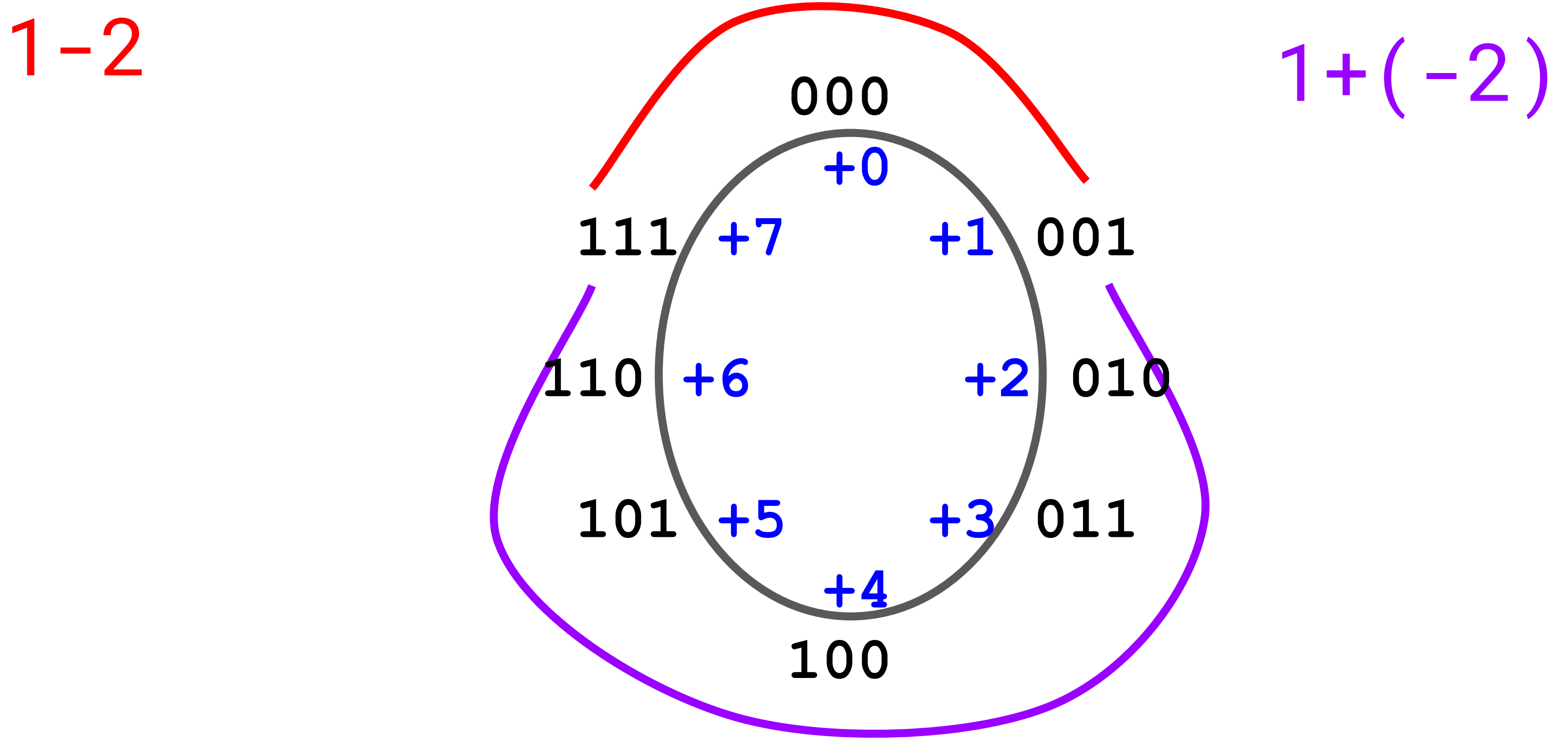
```
   1
 0001   A
 1101  !B
+   1
 1111

SF = 1
ZF = 0
OF = 0
CF =
```

CF should be be zero because there was no carry out of the left-most bit.

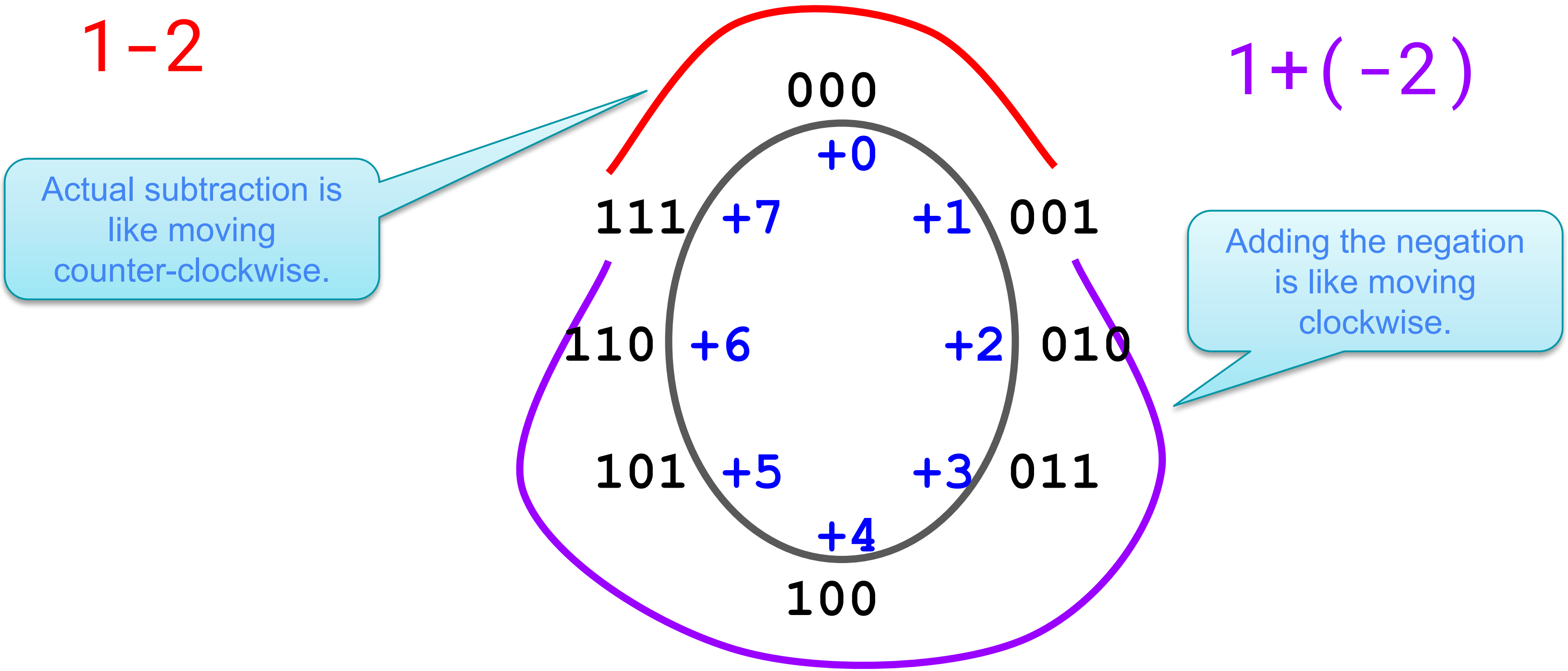
But it <u>was</u> an unsigned overflow.

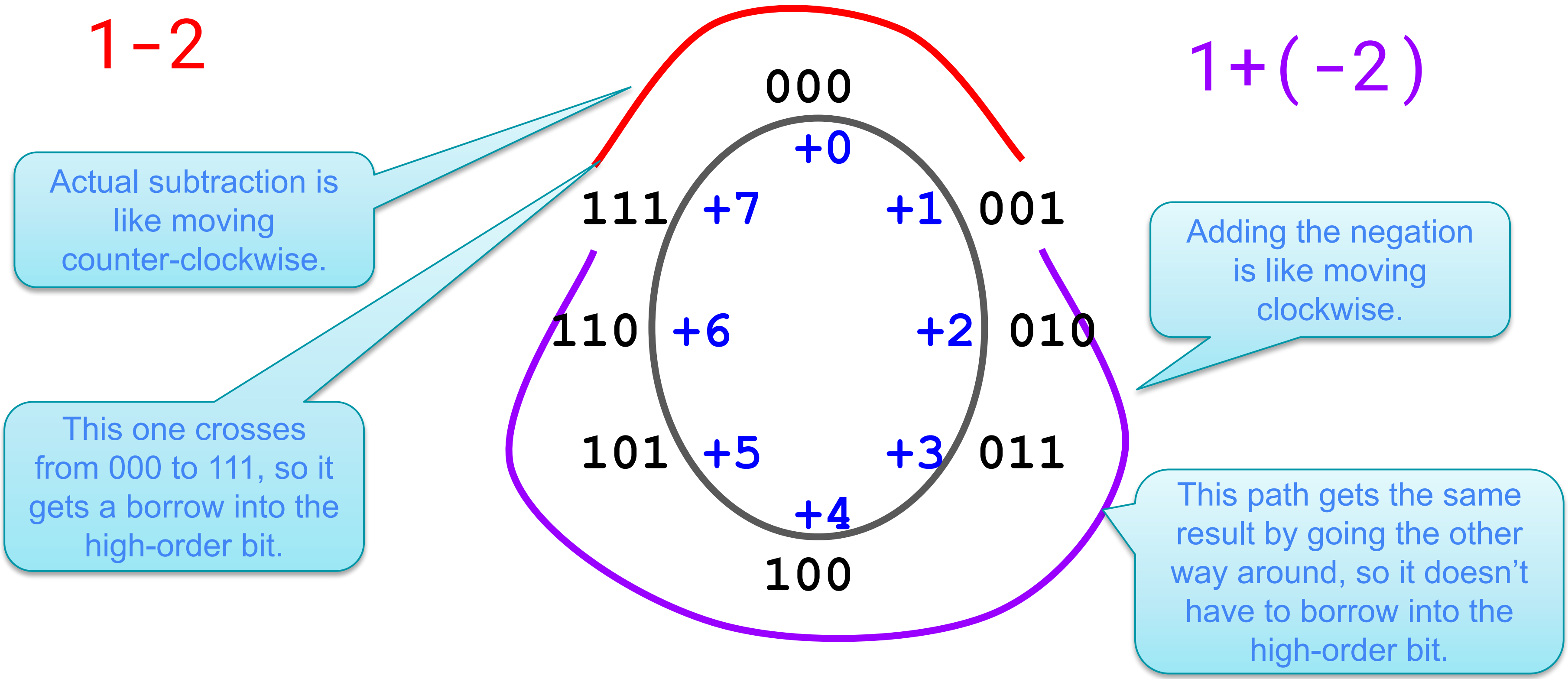
1-2





1-2
1+(-2)
000
+0
111
+7
+1
001
110
+6
+2
010
101
+5
+3
011
+4
100


1-2
1+(-2)
Actual subtraction is like moving counter-clockwise.
Adding the negation is like moving clockwise.
000
+0
111
+7
+1
001
110
+6
+2
010
101
+5
+3
011
+4
100

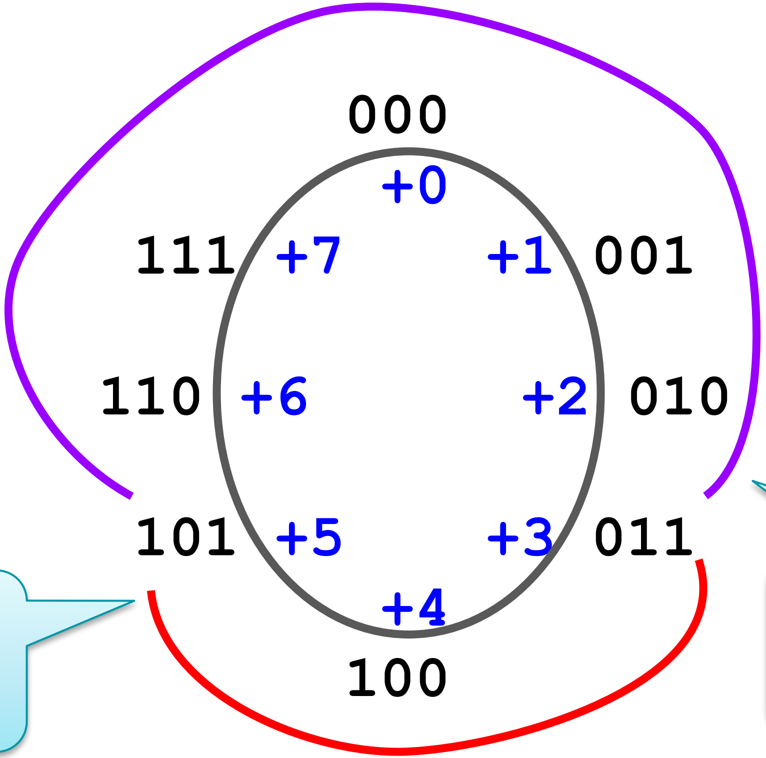


1-2
1+(-2)
000
001
010
011
100
101
110
111
+0
+1
+2
+3
+4
+5
+6
+7
Actual subtraction is like moving counter-clockwise.
This one crosses from 000 to 111, so it gets a borrow into the high-order bit.
Adding the negation is like moving clockwise.
This path gets the same result by going the other way around, so it doesn't have to borrow into the high-order bit.

Another Example

5-2

As an unsigned values, this should not be considered an overflow.



5+(-2)

But, adding -2 will cross from 111 to 000, so it will carry out of the high-order bit.

The red path shows the borrow behavior you should get.

The purple path will always get a carry that's the opposite.

Different result

- Unsigned subtraction directly via binary subtraction
    - Crosses from 000000 to 111111 when there's overflow.
    - Must borrow into the high-order bit.
    - Borrow into high-order bit indicates unsigned overflow
        - So, carry flag is set (to indicate a borrow)
- Unsigned subtraction via addition of the two's complement
    - The rule for setting OF (signed overflow) is fine.
    - But, the CF (unsigned overflow) is tricky.
    - Carry out of the high-order bit indicates no overflow.
    - No carry out of the high-order bit indicates overflow.
        - So, the CF will <u>always</u> be wrong.

8086 carry flag rule

8086 designers knew about this.

The 8086, on subtraction, always sets the CF to represent actual unsigned overflow (no matter how subtraction is implemented)

If you use two's complement addition to perform subtraction then the 8086 CF will be the opposite of the (computed) carry out of the high-order bit.

So, it will correctly indicate unsigned overflow no matter how subtraction is implemented.

(7-9 & 7-10 in Class Notes)