ASM Language

CSC 236

Precision vs. accuracy

Precision

- Number of digits in the calculation

Accuracy

- Is the result correct?

```
  1.174567940325672
+ 2.348920178544532
 27.010203040599999
```

Precise but not accurate

Safeguard — anti-ballistic missile system

- Determines position of incoming missile
  - Emits a signal
  - Bounces off missile
  - Detect reflection
  - Measure time (t_{rt})
- Distance to missile
  - Distance = speed x time
  - Speed 186,000 miles/sec

Safeguard — anti-ballistic missile system

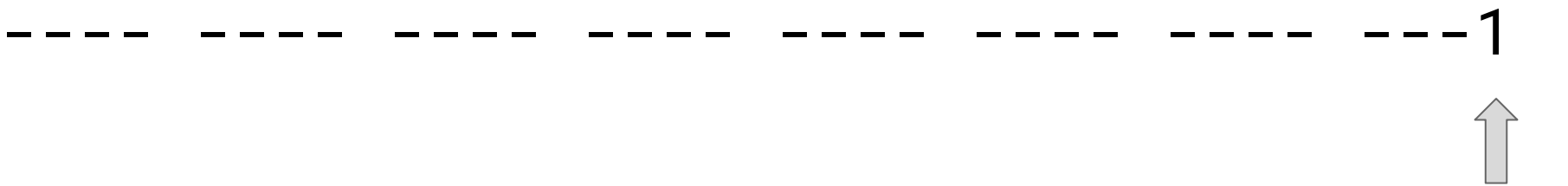
- Precision timer
  - 13.02 us
- Distance of 1 tick
  - 186,000 mile/s x 13.02 us
  - 1.86×10^{5} mile/s x 1.302×10^{-5} s
  - ~ 2.42 mile
- Round trip
  - Smallest distance is ~1.21 miles
  - Eg, 10 ticks ⇒ missile is approx 12.1 miles away

Safeguard — anti-ballistic missile system

- Preliminary testing
  - 1,000+ mile radius
  - 3 orders of magnitude worse than expected
- Examined code
  - Software is usually the problem
  - 3 months … no code errors
  - ?

Safeguard — anti-ballistic missile system

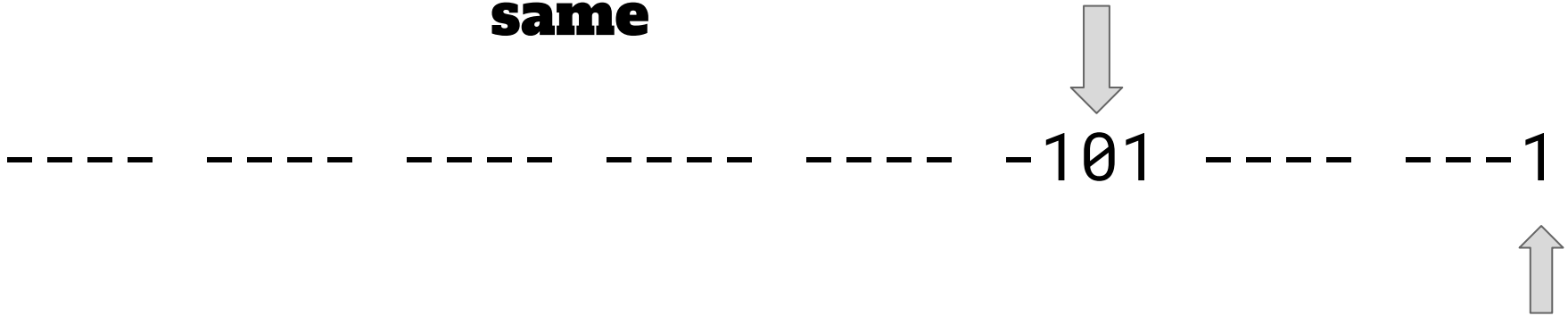
- Timer
  - 13.02 us precision
  - 32-bit counter



Safeguard — anti-ballistic missile system

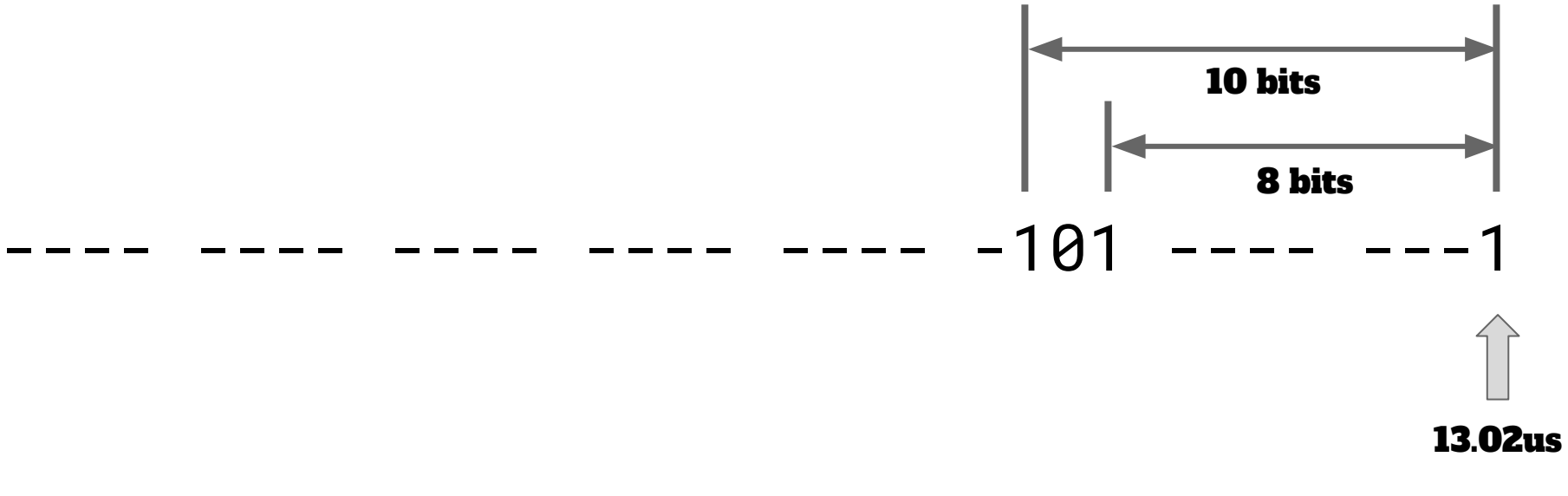
- Timer
  - 13.02 us precision
  - 32-bit counter

Discovered these bits always the same



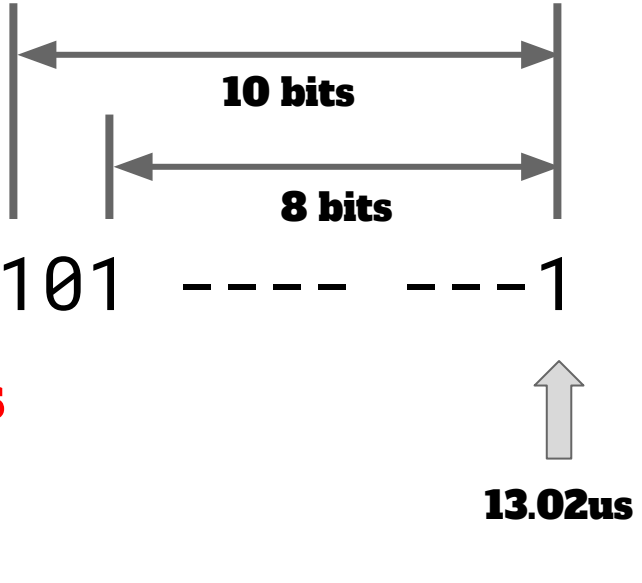
Safeguard — anti-ballistic missile system

- Timer
  - 13.02 us precision
  - 32-bit counter



Safeguard — anti-ballistic missile system

- Timer
  - 13.02 us precision
  - 32-bit counter
- Fixed bits
  - 13.02us x 2^8 = 3.3ms
  - 13.02us x 2^{10} = 13.33ms
  - Sum = 16.66ms



- **1/16.66ms = 60 cycles/sec = 60Hz**
- **Timer was updated every 16.66ms**
- **Precision 13.02us**
- **Accuracy 16.66ms**

Calculate C = A+B

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

```
mov  al,[a]      ; ax = --0A
add  al,[b]      ; ax = --41
mov  [c],al      ;  c = 41
```

How does this become a *program*?

```
;-------------------------------------------------------------
;   Program:  Hello (MASM version)
;
;   Function: Writes "Hello World" to the standard output device.
;             It is equivalent to the famous C++ program, hello world.
;
;             It uses 3 constant declarations to generate the message,
;             to show that dos writes data until it finds a '$'.
;
;             A single constant will also work, such as
;             msg  db  'Hello World',13,10,'$'
;
;             In assembler you have as much flexibility as you want.
;
;   Owner:    DAL
;
;   Date:     Changes
;   01/26/01  original version
;
;---------------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;---------------------------------------


;---------------------------------------
        .data                         ;start the data segment
;---------------------------------------
msg     db         'Hello World'      ;the hello world message
eol     db         13,10              ;cr/lf = \n in c++
term    db         '$'                ;dos end of string
                                      ;
msglen  dw         term-msg+1         ;total length of the message
;---------------------------------------


;---------------------------------------
        .code                         ;start the code segment
;---------------------------------------
hello:                                ;
        mov        ax,@data           ;establish addressability to the
        mov        ds,ax              ;data segment for this program
;---------------------------------------
; write out the message
;---------------------------------------
        mov        dx,offset msg      ;point to the message
        mov        ah,9               ;set the dos code to write a string
        int        21h                ;write the string
;---------------------------------------
; terminate program execution
;---------------------------------------
exit:                                 ;
        mov        ax,4c00h           ;set dos code to terminate program
        int        21h                ;return to dos
        end        hello              ;end marks the end of the source code
                                      ;....and specifies where you want the
                                      ;....program to start execution
;---------------------------------------
```

- **Program has *sections***
  - **Directives**
  - **Data**
  - **Code**

```
;-----------------------------------------------------------
;   Program:  Hello (MASM version)
;
;   Function: Writes "Hello World" to the standard output device.
;             It is equivalent to the famous C++ program, hello world.
;
;             It uses 3 constant declarations to generate the message,
;             to show that dos writes data until it finds a '$'.
;
;             A single constant will also work, such as
;             msg  db  'Hello World',13,10,'$'
;
;             In assembler you have as much flexibility as you want.
;
;   Owner:    DAL
;
;   Date:     Changes
;   01/26/01  original version
;-----------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-----------------------------------

;-----------------------------------
        .data                         ;start the data segment
;-----------------------------------
msg     db         'Hello World'      ;the hello world message
eol     db         13,10              ;cr/lf = \n in c++
term    db         '$'                ;dos end of string
                                      ;
msglen  dw         term-msg+1         ;total length of the message
;-----------------------------------


;-----------------------------------
        .code                         ;start the code segment
;-----------------------------------
hello:                                ;
        mov        ax,@data           ;establish addressability to the
```

- **Program has *sections***
  - **Directives**
  - **Data**
  - **Code**

```
;-----------------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-----------------------------------------
```

```
;-----------------------------------------------------------
;   Program:  Hello (MASM version)
;
;   Function: Writes "Hello World" to the standard output device.
;             It is equivalent to the famous C++ program, hello world.
;
;             It uses 3 constant declarations to generate the message,
;             to show that dos writes data until it finds a '$'.
;
;             A single constant will also work, such as
;             msg  db  'Hello World',13,10,'$'
;
;             In assembler you have as much flexibility as you want.
;
;   Owner:    DAL
;
;   Date:     Changes
;   01/26/01  original version
;-----------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-----------------------------------

;-----------------------------------
        .data                         ;start the data segment
;-----------------------------------
msg     db         'Hello World'      ;the hello world message
eol     db         13,10              ;cr/lf = \n in c++
term    db         '$'                ;dos end of string
                                      ;
msglen  dw         term-msg+1         ;total length of the message
;-----------------------------------


;-----------------------------------
        .code                         ;start the code segment
;-----------------------------------
hello:                                ;
        mov        ax,@data           ;establish addressability to the
```

- **Directives**
  - **Flags for the assembler**
- **.model small**
  - **Limits segments to 64KB (8086)**
- **.8086**
  - **Only allow 8086 instructions**
- **.stack 256**
  - **Declare a stack size**
  - **256 B is big enough for this class**

```
;-----------------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-----------------------------------------
```

```
;-------------------------------------------------------------
;   Program:  Hello (MASM version)
;
;   Function: Writes "Hello World" to the standard output device.
;             It is equivalent to the famous C++ program, hello world.
;
;             It uses 3 constant declarations to generate the message,
;             to show that dos writes data until it finds a '$'.
;
;             A single constant will also work, such as
;             msg  db  'Hello World',13,10,'$'
;
;             In assembler you have as much flexibility as you want.
;
;   Owner:    DAL
;
;   Date:     Changes
;   01/26/01  original version
;
;-------------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-------------------------------------
;-------------------------------------
        .data                         ;start the data segment
;-------------------------------------
msg     db
eol     db
term    db

msglen  dw
;-----------------
;-----------------
        .code
;-----------------
hello:
        mov
        mov
;-----------------
; write out the mes
;-----------------
        mov
        mov
        int
;-----------------
; terminate program
;-----------------
exit:
        mov
        int
        end


;-----------------
```

```
;-----------------------------------------
          .data                          ;start the data segment
;-----------------------------------------
msg       db        'Hello World'        ;the hello world message
eol       db        13,10                ;cr/lf (DOS line termination)
term      db        '$'                  ;DOS end of string
                                         ;
msglen    dw        term-msg+1           ;total length of the message
;-----------------------------------------
```

- **Program has *sections***
  - **Directives**
  - **Data**
  - **Code**

```
;-----------------------------------------------------------------
;   Program:  Hello (MASM version)
;
;   Function: Writes "Hello World" to the standard output device.
;             It is equivalent to the famous C++ program, hello world.
;
;             It uses 3 constant declarations to generate the message,
;             to show that dos writes data until it finds a '$'.
;
;             A single constant will also work, such as
;             msg  db  'Hello World',13,10,'$'
;
;             In assembler you have as much flexibility as you want.
;
;   Owner:    DAL
;
;   Date:     Changes
;   01/26/01  original version
;
;-----------------------------------------
        .model     small              ;64k code and 64k data
        .8086                         ;only allow 8086 instructions
        .stack     256                ;reserve 256 bytes for the stack
;-----------------------------------------


;-----------------------------------------
        .data                         ;start the data segment
;-----------------------------------------
msg     db         'Hello World'      ;the hello world message
eol     db         13,10              ;cr/lf = \n in c++
term    db         '$'                ;dos end of string
                                      ;
msglen  dw         term-msg+1         ;total length of the message
;-----------------------------------------
```

```
;-----------------------------------------
        .code                         ;start the code segment
;-----------------------------------------
hello:                                ;
        mov        ax,@data           ;establish addressability to the
        mov        ds,ax              ;data segment for this program
;-----------------------------------------
; write out the message
;-----------------------------------------
        mov        dx,offset msg      ;point to the message
        mov        ah,9               ;set the dos code to write a string
        int        21h                ;write the string
;-----------------------------------------
; terminate program execution
;-----------------------------------------
exit:                                 ;
        mov        ax,4c00h           ;set dos code to terminate program
        int        21h                ;return to dos
        end        hello              ;end marks the end of the source code
                                      ;....and specifies where you want the
                                      ;....program to start execution
;-----------------------------------------
```

- **Code segment**
  - **Begins with** .code

```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset msg
        mov     ah,9
        int     21h
;---------------------------------------
; terminate program execution
;---------------------------------------
exit:
        mov     ax,4c00h                ;
        int     21h                     ;
        end     hello                   ;
                                        ;
                                        ;
;---------------------------------------
```

Three parts

1. **initialization**

Remember:

- **DS & ES set by programmer**
- **Cannot move immediate into segment register**

Data segment register

- Not allowed
  - `mov ds, @data`
  - Cannot move immediate to segment register
- What is @data?
  - It is the location of the data segment
  - In memory — it is a memory label
- Consider
  - `val    db    20h`
  - val is the name for a memory location
  - 20h is the value you want to store there.

@data

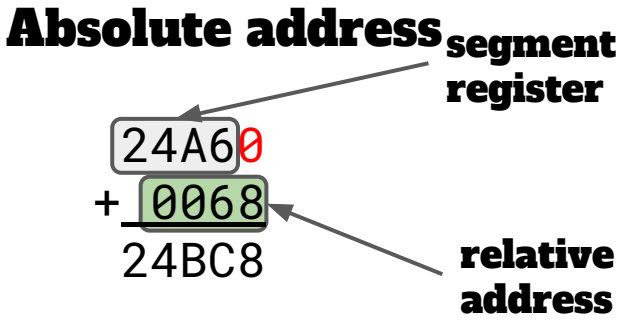
- What is at “@data”
  - A built-in, implicit assembler variable
  - It denotes the beginning of the data segment
  - It is a memory reference, location
  - The assembler interprets this as an *immediate* value
- Why a name
  - Loader decides (not programmer)
  - Decision is made when DOS loads program into memory (not static)

mov ds, @data

- Why doesn't this work?
  - @data is an immediate value
- What about?
  - `val  dw  @data  ; val = @data`
  - `mov  ds,[val]`
- Why doesn't that work?

mov ds, @data

- Why doesn't this work?
  - @data is an immediate value
- What about?
  - `val dw @data ; val = @data`
  - `mov ds,[val]`
- Why doesn't that work?
  - How to reference val?
  - val is a relative offset
  - Must initialize the ds first



```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset m
        mov     ah,9
        int     21h
;-----------------------------
; terminate program execution
;-----------------------------
exit:
        mov     ax,4c00h
        int     21h
        end     hello


;-----------------------------
```

Three parts

1. **initialization**

Remember:

- **DS & ES set by programmer**
- **Cannot move immediate into segment register**

If program does not have data, do you need to initialize ds?

```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset m
        mov     ah,9
        int     21h
;-----------------------------
; terminate program execution
;-----------------------------
exit:
        mov     ax,4c00h
        int     21h
        end     hello


;-----------------------------
```

Three parts

1. **initialization**

Remember:

- **DS & ES set by programmer**
- **Cannot move immediate into segment register**

If program does not have data, do you need to initialize ds?

No.

```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset msg           ;
        mov     ah,9                    ;
        int     21h                     ;
;---------------------------------------
; terminate program execution
;---------------------------------------
exit:                                   ;
        mov     ax,4c00h                ;
        int     21h                     ;
        end     hello                   ;
                                        ;
                                        ;
;---------------------------------------
```

Three parts

1. **Initialization**
2. **Body**

```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset msg
        mov     ah,9
        int     21h
;---------------------------------------
; terminate program execution
;---------------------------------------
exit:
        mov     ax,4c00h
        int     21h
        end     hello

;---------------------------------------
```

Three parts

1. **Initialization**
2. **Body**
3. **Termination**

- **int — “interrupt”**
  - **Generates a software interrupt**
  - **21h is caught by DOS**
- **int 21h — is a service call to DOS**
  - **ah — desired service**
- **4C**
  - **Terminate (al is the exit status)**
  - **Eg, exit(0)**

```
;---------------------------------------
        .code                           ;
;---------------------------------------
hello:                                  ;
        mov     ax,@data                ;
        mov     ds,ax                   ;
;---------------------------------------
; write out the message
;---------------------------------------
        mov     dx,offset msg
        mov     ah,9
        int     21h
;---------------------------------------
; terminate program execution
;---------------------------------------
exit:                                   ;
        mov     ax,4c00h                ;
        int     21h                     ;
        end     hello                   ;
                                        ;
                                        ;
;---------------------------------------
```

Three parts

1. **Initialization**
2. **Body**
3. **Termination**

- **End**
  - **Last statement**
  - **Mandatory**
  - **Has to match start label**

Assembler keywords

- Reserved words
  - Cannot be used as labels or variable names
- Includes
  - Opcodes — add, mov, sub
  - Directives — end
- Full list on web page
  - See: “Reserved Words in MASM”

Data sizes

- Source and destination must be same size
- Suppose you want to add AX and DL
  - `add  ax,dl`

Data sizes

- Source and destination must be same size
- Suppose you want to add AX and DL
  - `add  ax,dl`

- In C:
  - `char a = 9;`
  - `short b = 10;`
  - `short c;`
  - `c = a + b;`

a is *implicitly* cast from char to short

Data sizes

- Source and destination must be same size
- Suppose you want to add AX and DL
  - `add  ax,dl`

- In C:
  - Can do *explicit* cast
  - e.g., `long x = (long)c;`

Asm casting

- 4-bit to 8-bit
- Unsigned
  - 1_{10} = 0001_2 = $\underline{0000}0001_2$
  - 10_{10} = 1010_2 = $\underline{0000}1010_2$
- Extend with zeros
- Byte to word
  - `mov al,[var]` `;ax = ?? 0A`$_{16}$
  - `mov ah,0` `;ax = 00 0A`$_{16}$

It's convenient to be able to access ax as two bytes.

Asm casting

- 4-bit to 8-bit
- Signed
  - $+1_{10} = 0001_2 = \underline{0000}0001_2$
  - $-1_{10} = 1111_2 = \underline{????}1111_2$

Asm casting

- 4-bit to 8-bit
- Signed
  - $+1_{10} = 0001_2 = \underline{0000}0001_2$
  - $-1_{10} = 1111_2 = \underline{1111}1111_2$

Asm casting

- 4-bit to 8-bit
- Signed
  - $+1_{10} = 0001_2 = \underline{0000}0001_2$
  - $-1_{10} = 1111_2 = \underline{????}1111_2$
- For positive
  - Add zeros (same as unsigned)
- For negative
  - Add ones
- How do you know which?

Asm casting

- Byte to word
- Signed
  - $+1_{10} = 01_{16} = 0001_{16}$
  - $-1_{10} = FF_{16} = FFFF_{16}$
- For positive
  - Add zeroes (same as unsigned)
- For negative
  - Add ones
- How do you know which?

Asm casting

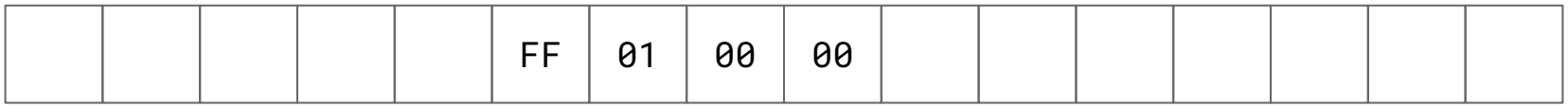
- Byte to word
- Signed
  - $+1_{10} = 01_{16} = 0001_{16}$
  - $-1_{10} = FF_{16} = FFFF_{16}$
- For positive
  - Add zeroes (same as unsigned)
- For negative
  - Add ones
- How do you know which?
  - cbw — convert byte to word

Asm casting

- Unsigned
  - Move 00_{16} into high-order byte
  - Works for any general-purpose register (ax, bx, cx, dx)
- Signed
  - Use cbw
  - ax & al are implicit source and destination
  - First: move byte into al
  - Then: call cbw
  - ax holds the sign-extended value of al

Unsigned conversion

```
a db  255  ; FF16
b db  1    ; 0116
c dw  0    ; 00 0016
```



Cannot convert a or b in memory

Unsigned conversion

```
a db  255  ; FF16
b db  1    ; 0116
c dw  0    ; 00 0016
```

```
mov   al,[a]   ;ax = __ FF
mov   ah,0     ;ax = 00 FF
mov   bl,[b]   ;bx = __ 01
mov   bh,0     ;bx = 00 01
add   ax,bx    ;ax = 01 00
mov   [c],ax
```

Unsigned conversion

```
a db  255  ; FF16
b db  1    ; 0116
c dw  256  ; 00 0116
```

```
mov   al,[a]   ;ax = __ FF
mov   ah,0     ;ax = 00 FF
mov   bl,[b]   ;bx = __ 01
mov   bh,0     ;bx = 00 01
add   ax,bx    ;ax = 01 00
mov   [c],ax
```

Signed conversion

```
a db  -1    ; FF16
b db  2     ; 0216
c dw  0     ; 00 0016
```

Signed conversion

```
a db  -1    ; FF16
b db  2     ; 0216
c dw  0     ; 00 0016
```

```
mov   al,[a]    ;ax = __ FF
cbw             ;ax = FF FF
```

sign-bit in al is 1
⇒ cbw extends with 1s

Signed conversion

```
a db  -1    ; FF16
b db  2     ; 0216
c dw  0     ; 00 0016
```

```
mov   al,[a]    ;ax = __ FF
cbw             ;ax = FF FF
mov   [c],ax    ;c  = FF FF
mov   al,[b]    ;ax = FF 02
cbw             ;ax = 00 02
```

sign-bit in al is 0
⇒ cbw extends with 0s

Signed conversion

```
a db  -1    ; FF16
b db  2     ; 0216
c dw  0     ; 00 0016
```

```
mov   al,[a]    ;ax = __ FF
cbw             ;ax = FF FF
mov   [c],ax    ;c  = FF FF
mov   al,[b]    ;ax = FF 02
cbw             ;ax = 00 02
add   [c],ax    ;c =  00 01
```

Signed conversion

```
a db  -1    ; FF16
b db  2     ; 0216
c dw  1     ; 01 0016
```

```
mov   al,[a]    ;ax = __ FF
cbw             ;ax = FF FF
mov   [c],ax    ;c  = FF FF
mov   al,[b]    ;ax = FF 02
cbw             ;ax = 00 02
add   [c],ax    ;c =  00 01
```

Some basic opcodes

- `inc dest  ;dest = dest + 1`
- `dec dest  ;dest = dest - 1`
- `neg dest  ;dest = 0 - dest = -dest`
- `cwd       ;converts 16-bit signed value in ax`
  `          ;to a 32-bit signed value in dx:ax`

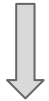
Increment

- `ADD SI,1 ;SI = SI + 1`
  - ASM LANG is **not** case sensitive
  - Increments the si register
  - Sets the condition codes: SF, ZF, OF, CF
- `inc si ;SI = SI + 1`
  - Increments the si register
  - Sets the condition codes: SF, ZF, OF
  - But not CF

Why not?

Suppose need to add column of numbers

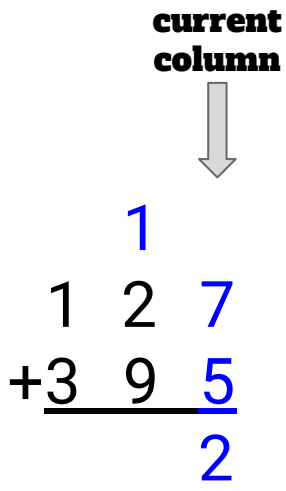
current column



```
 1  2  7
+3  9  5
```

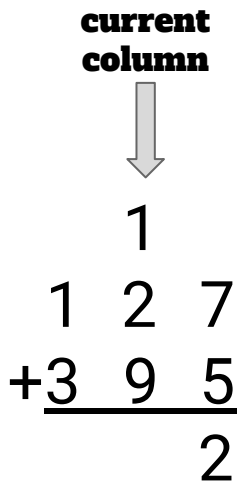
```
i = 0
TOP[i] + BOT[i]
```

Suppose need to add column of numbers




```
i = 0
TOP[i] + BOT[i]
```

Suppose need to add column of numbers




```
i++
TOP[i] + BOT[i] + CARRY
```

Must be able to move index pointer without losing carry information

Moving the pointer with an inc or dec maintains the carry

Dest = source

This is legal:

```
    add ax,ax    ;doubles ax
```

Immediate values

How you specify immediate data affects the readability of code

- `mov al,65`
- `mov al,41h`
- `mov al,'A'`

Which is better?

Immediate values

How you specify immediate data affects the readability of code

- `mov al,65`
- `mov al,41h`
- `mov al,'A'`

Which is better?

- Depends on how al is being used
- If as a char, last one is much more readable than the others

Compare

- cmp dest, src — Compare instruction
  - Compares two values
  - Signed or unsigned
  - No special hardware
  - Works by subtracting src from dest

Compare

`sub ax,bx`

- Calc ax - bx
- Save result in ax
- Sets CC

`cmp ax,bx`

- Calc ax - bx
- Does not save result in ax
- Sets CC

```
ax < bx    result < 0
ax = bx    result = 0
ax > bx    result > 0
```

Compare

`sub ax,bx`

- Calc ax - bx
- Save result in ax
- Sets CC

`cmp ax,bx`

- Calc ax - bx
- Does not save result in ax
- Sets CC

```
ax < bx     result < 0
ax = bx     result = 0
ax > bx     result > 0
```

Is this a signed or an unsigned comparison?

Depends on which condition codes you check.