Assembly - 8.28.11

CSCI 2160

* Hollerith Code
- 1880 census took 10 years to calculate
- Hollerith came up tabulating machine
- the card had 80 columns
- 12-1 was A, 12-2 was B
- I was 12-9
- only did uppercase
- J was 11-1, R was 11-9
- S was 0-2, Z was 0-9

* Number Systems
- $476_9 + 583_9$ $\quad 467_8 + 557_8$ $\quad 1011_2 + 1101_2$

$$\begin{array}{r} 476_9 \\ +\,583_9 \\ \hline 1170_9 \end{array} \qquad \begin{array}{r} 467_8 \\ +\,557_8 \\ \hline 1246_8 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\,1\,1} \\ 1011_2 \\ +\,1101_2 \\ \hline 11000_2 \end{array}$$

- $\begin{array}{r} {\scriptstyle 3\,12\,15} \\ 4\,8\,8_7 \\ -\,146_7 \\ \hline 256_7 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\,\;10\,11} \\ 2\,1\,4_4 \\ -\,132_4 \\ \hline 13_4 \end{array}$ $\quad 1\emptyset\#_{11} = 256_{10} + 0 + 11 = 257_{10}$
- needed 8 bits in a byte so that all characters on a keyboard could be represented
- addresses are binary also
- $1100_2 - 12$

* EBCDIC
- 12-1 - $1100_{12}|0001$ - A
- 12-2 - $1100_{12}|0010$ - B
-

- when reading memory, you need to know context

- 1101 0001 - J in EBCDIC
- 1110 0001 - S in EBCDIC
- EBCDIC has zone and digit, derived from Hollerith

* ASCII
- starting with capital A, 0100 0001 (65 in decimal) + 41_h
- letters after are incremented
- for ASCII, turn on 3rd bit
- lowercase a = 0110 0001 - 97 + 61_h
- for lowercase, add 32 in decimal or 20_h

* Numbers in EBCDIC
- 0 - F0 in EBCDIC
- 9 - F9 in EBCDIC
- EBCDIC, characters precede numbers when sorting
- ASCII - numbers precede characters

* Numbers in ASCII
- 0 is 30_h in ASCII
- 9 is 39_h in ASCII
-

9.1.11

Assembly

CSCI 2160

* Bytes
- byte = 8 bits 2^8
- word = 2 bytes 2^{16}
- double word = 2 words = 4 bytes 2^{32}
- quad word = 4 words = 8 bytes 2^{64}

* Signed integers
- MSB is the sign bit
- use two's compliment - first represent positive value, flip the bits & add 1
- -37 = 0DBh
- -97 → 01100001 → 10011110 + 1 = 10011111 = 9Fh
- -1 → 00000001 → 11111110 +1 = 1111 1111 = 0FFh
- in Assembly, you must write a 0 before a character in hex
- D7h as unsigned = 215
- D7h as character = Ö
- D7h as signed = 28h is flipped = 28h+1h = 29h = -41
- 83h as unsigned = 128 + 3 = 131
- 83h as signed = 7Ch + 1 = 70h = -125
- 73h as unsigned = 112 + 3 = 115
- 73h as signed = 115
- 80h as unsigned = 128
- 80h as signed = -128
- you can always represent one more negative value than positive in a signed binary number
- 8000h as signed = -32768
- 8000h as unsigned = 32768
- -125 as byte = 01111101 + 1 = 01111110 → 11111110 → 0FDh ✗ 83

-125 as byte = 01111101 → 10000010 + 1 = 10000011 → 83h ✓
bits flip bits add 1

* Symbols
- byte - db
- word - dw
- double word - dd
- quad word - dq
- answer byte
- byte answer; Java
- int value = 35; Java
- value dd 35 Assembly
or value dword 35 Assembly
- Java → Assembly
- short tax = 13; → tax dw 13h
- long value = 19; → value qw 19h
- int number = -20 → number dd -20
- byte answer = 'B'; → answer db 'B'
(write lined up)

```
.data
tax     word   13
value   qword  19
number  dword  -20
answer  byte   'B'      (you can use ' or ")
```

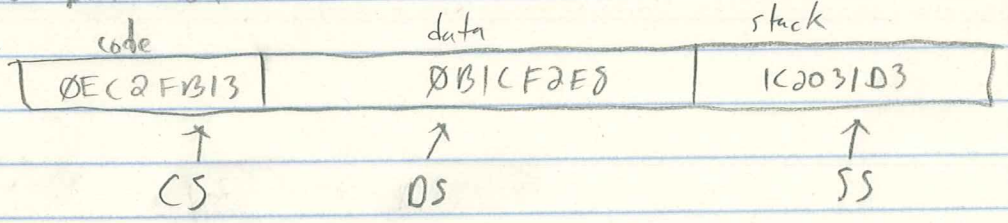
- store values in memory in reverse byte order
- Assembly assumes decimal if there is no trailing letter; use d to specify decimal, use h to specify hex
- 13d → (hex) 000Dh → (reverse byte) 0D00h
- -20d as dword → (hex) FFFFFFECh → (flip bits +1) 00000014h → (reverse byte) 14000000h
'B' as byte → (hex) 42h → (reverse) 42h

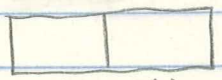
Statement	hex	reverse byte
- A word 22 →	0016h →	1600h
- B word 19 →	0013h →	1300h
- C dword 37 →	00000025h →	25000000h

- data values are stored sequentially
- strings are not stored in reverse byte order

* Registers

- RAX, RBX, RCX, RDX (workhorse registers) (32-bits)
- CS, DS, SS (32 bit)
- flat segment model - can use all 32 bits to represent an address
- 32 bits can store 4 GB of addresses
- 2^{40} - terabyte
- 2^{50} - petabyte
- 2^{60} - exabyte
- 2^{70} - zettabyte
- 2^{80} - yottabyte
- lower RAX is EAX, lower RBX is EBX, so on
- descriptor table





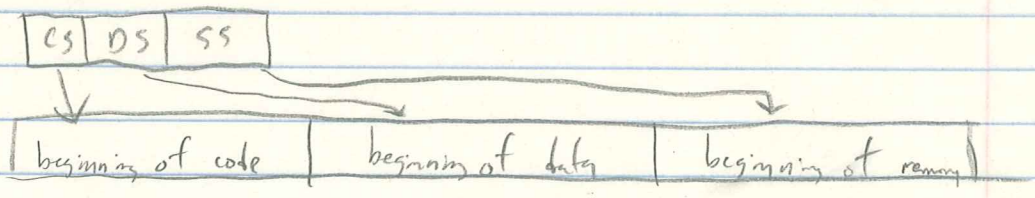
AX : 16 bits
AL+AH : 8 bits
EAX : 32 bits

Assembly

CSCI 2160

* Quiz 1 & 2 Review
- A in ASCII is 41h, 65d
- a in ASCII is 61h, 97d
- A in Hollerith is 12-1, EBCDIC is C1h
- J in Hollerith is 11-1, EBCDIC is D1h (not direct translation)
- C1-C9, D1-D9, E2-E9 - EBCDIC alphabet
- 1d in EBCDIC is F1h, ASCII is 31h
- 0d in ASCII is 30h
- Remember that a digit in a base can not be larger or equal to the base
- Long quiz on Thursday! 9.8.11

* Notes
- register - high speed memory that perform arithmetic, holds instructions
- workhorse registers - A, B, C, & D
- 8 bit - AH & AL
- 16 bit - AX, BX, CX, & DX
- 32 bit - EAX, EBX, ECX, EDX
- 64 bit - RAX, RBX, RCX, RDX
- CS, DS, SS - 16 bit
- CS - code, DS - data, S - segment (stacks)
- CS gives us a description
-



- index registers - SI & DI
- SI - source index, DI - destination index
- used to hold starting address of something

Assembly

9.6.11

CSCI 2160

- index registers are 16 bit
- ESI, EDI - extended index registers, 32 bit
- RSI, RDI - 64 bit index registers
- pointer registers (no 8 bit)
- BP - base pointer, keeps track of where stack was at time a piece of code is entered
- SP - stack pointer, point to top of stack
- IP - instruction pointer
- IP is not to be messed with, but hackers uses IP to inject code
- IP is updated automatically by operating system
- all of these have extended registers - EBP, ESP, EIP
- .data, .stack, .code - directives
- .stack 100h - allocates stack of 256 bytes
- 1 byte byte db
  2 word dw
  4 dword dd
  8 qword dq
- .data

				location counter
bTax	byte	?	00	0000
wVal	word	27	001B	0001
dVal	dword	-7	FFFFFFF9	0003
qNum	qword	19	00000000 00000013	0007
				000F

- assembler creates symbol table
  symbol - location counter - size

Assembly

CSCI 2160

- if you define an array of values, the location counter reflects that 1 to 1
- the size table reflects the size of what is declared (byte, word, dword, etc.)

loc ctr	.data	equivalent hex	size
0000	byte 9,7	09, 07	1
0002	word 17, -2, 3	0011, FFFE, 0003	2
0008	dword 29, 32	0000001D, 00000020	4
0010	qword 5	00000000 00000005	8
0018	byte '123A'	31323341	1
001C			

- in memory ...

```
.data
09 07
11 00 FE FF 03 00
1D 00 00 00 20 00 00 00
05 00 00 00 00 00 00 00
31 32 33 41
```

- dup directive (used with # in front
- 2 dup (?) - 00 00 (byte)
- 3 dup (7) - 0007 0007 0007 (word)
-

Assembly

CSCI 2160

- wX word 33, 19
  wY word 26, 39, 55
  wZ word ?
  wA word 2 dup (5)
  dVal dword 3 dup (-19)
  dStr byte 2 dup ('ABC')

loc ctr	actual hex	reverse byte order
0000	0021, 0013	21 00 13 00
0004	001A, 0027, 0037	1A 00 27 00 37 00
000A	0000	00 00
000C	0005, 0005	05 00 05 00
0010	FFFFFFED, FFFFFFED, FFFFFFED	ED FF FF FF ED FF FF FF ED FF FF FF
001C	41 42 43 41 42 43	41 42 43 41 42 43
0022		

- Quiz on Thursday will cover everything up to 9.6.11

Assembly

- ESP contains address of stack first byte beyond the top
- On Quiz 3, 12b is 0018ff8C
  12c is 0018FE8C
- push - 1.) subtracts from ESP 2.) copies bytes to stack
  pop - 1) copies data from stack 2.) pointer moved
- Quiz on Thursday, 9.15

- Integer constants
  decimal - add ax 24; add ax 24d
  binary - add ax 100b,
  hex - add ax 64h,
  character - add ax 'A'
  string - add ax 'John went'; add ax "John went"
- Integer expressions
  () - parenthesis
  * ÷ / - multiply & divide
  mod - modulus
  + - - add, subtract
- Reserved words
  instruction mnemonics - add, sub, mul, mov
  registers - eax, ebx...
  directives - tells MASM what to do
  attributes - messages to assembly - byte, dword
  operators - +, -, ()
  pre-defined symbols - @
- Identifiers
  must begin with A-Z, a-z, _, $, @, ?
  can be up to 247 characters
  normally not case sensitive, unless switch is turned on
  must start A-Z, a-z, other characters can be numerical

Assembly

- Directives
  not executable
  define variable or procedures
  can assign names to memory segments or code
  case insensitive
  Example - .486, .model
  first three lines of program should be
  .486
  .model flat
  .stack 100h

- .asm → assembler → .lst
  .obj → linker
  .obj
  .lib

- Instructions - 4 parts
  [label] operation [operands] [comment]
  label - assigned address of the instruction that follows, label appears in the symbol table
- to debug, _start:
  first line under .code
- last thing is PUBLIC start
- last line is END
  operation - mov
- b for byte, w for word, d for dword, q for qword, str for string (use these for variable names)
- CRLF - 10 13, 0
- strPrompt1 byte "Enter first number", 0
- to use Bailey's macro, follow string with a 0
- receiving operand is always first
-

Assembly

- each instruction must have a comment!
- add ebx, 4 ; ebx points to next integer value

```
; Program
; Programer
; Course
```

Write why? for comments

- Assembly cannot do memory to memory operations, must use registers!
- add wX, wY - WRONG!
- mov ax, wX ; wZ =
  add ax, wY ; wX +
  mov wZ, ax ; wY
- sub op1, op2 ; op1 gets result of op1 - op2
- Nop
  mov eax, dVal ; dVal : : 27d
  loader always makes sure it is on a double word boundary
  nop is used to align, takes up one byte of storage,
  speeds up processing

- Quiz 4 will be over this review

```
; Program        filename.asm          line up!
; Programer      Name
; Course         CSCI 2160-201
; Assignment
; Date
; Purpose
```

```
.486
.model flat
.stack 100h
```

(review handout on D2L)

- display prompt 1
  get value 1
  display prompt 2
  get value 2
  add values
  convert answer
  display answer

- .code

Quiz 4 BWB.asm

```
_start:
    puts    strPrompt1          ; display first input prompt
    gets    strInput, 10        ; get first value from keyboard
    stringToNum strInput, dXval ; convert to a true binary value
    puts    strPrompt2          ; display 2nd input prompt
    gets    strInput, 10        ; get second value fm keyboard
    stringToNum, dYval          ; convert to a true binary value
    mov     eax, dXval          ;   dXVal =
    add     eax, dYval          ;     dYval +
    mov     dXVal, eax          ;       dXval
    numToString strOut, dXval   ; convert the binary value to a numeric string
    puts    strOut              ; display string

    INVOKE ExitProcess, 0
    PUBLIC _start
    END
```

Assembly

- from memory to registers, remember that it is stored in memory in reverse byte order
- to reference memory, use brackets

```
mov ebx, offset wT      ; address of wT
mov eax, [ebx]          ; load address of ebx into eax
```

-

24	37	AE	57

```
mov ax, wT
mov ax, wT+1        AE37
mov ax, wT[1]   - 1 reference byte, ast position (offset)
```

- wT word 5,4,3,8,17,22

```
short wT = {5,4,3,8,17,22}
mov ax, wT          ax: 0005
mov ax, wT[1]   ; is the same as mov ax, wT+1      ax: 0400
mov ax, wT[4]       ax: 0003
```

- mov eax, [ebx + esi] - add two registers to determine address
  must use ebx + esi - class standard

- ml /c /coff /Zi /Fl test.asm
  macro.bat " " %1.asm
- link /debug /subsystem:console /entry:start /out:%1.exe %1.obj convutil20109.obj kernel32.lib utility20109.obj
-

Assembly

- equal-sign directive
  COUNT = 10
  COUNT is replaced, works as a constant, compiler substitutes that value at assemble time
- $ - current value of location counter
  wVal word 10, 20, 18, -7, 55 — 00AC
  wValSize = ($ - wVal) / # of bytes (2) — 00B6

  mov ecx, wValSize
  (use ecx for looping)
- equ directive - equates
  xVal equ [ebp + 4]
  equ is used more than = or $
- movsx - copy & extend sign — mov eax, wVal — FFFFFFFF
  movzx - copy & extend zero — mov eax, wVal — 00 00 FF FF

  wVal word -1

- cbw - convert byte to word — AL → AX
  cwd - convert word to double word — AX → DX:AX pair
  cwde - convert word to double word extended — AX → EAX
  cdq - convert double word to quad word — EAX → EDX:EAX pair

  works only for ax/eax registers
- EFLAGS - 32 bits
  status - contained in lower 16 bit

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Assembly

0 - carry flag
2 - parity flag
6 - zero flag
7 - sign flag
11 - overflow flag

carry - set when there is a carry out on an operation
parity - set to keep number of 1 bits odd for odd parity machine
sign - set to 1 when result is negative
zero - set to 1 when result is 0
overflow - result to 2 positives becoming negative

```
add ax, wVal
jo  - jump overflow
jc  - jump carry
```

- inc [operand] → value is increased by 1 (doesn't touch flags)
  inc [ebx] → invalid! must know size
  inc word ptr [ebx] - valid

- neg [operand] → replaces operand with 2's compliment (w flags set)
  neg ax

- add bVal, bVal bVal = 127 bVal = 1
  CF=0, OF=1

  add -128 + -128
  CF=1, OF=1
  handle overflow by jo

Assembly

- EFL represents the EFLAGS
  EFL will turn red when it is changed
- ESP - stack pointer

- neg does not just flip bits, it performs a 2's compliment

9.22.11

Assembly

- Exam 1 is next Thursday
- cbw - works with A register - convert byte to word
  eax: CE24E5C6
  cbw = CE24FFC6
- cwd - convert word to double word dx:ax
  eax: ce24e5c6 edx: 2416ABCD
  cwd = edx = 2416FFFF
- cwde - eax: FFFFe5c6
- cdq - edx:eax
  FFFFffff:ce24e5c6

movsx eax, cl — cl goes into al & sign is filled
movzx eax, cl — cl goes into al & zero is filled

Assembly

- type - # of bytes - the type that is assigned
  lengthof - # of different values
  sizeof - total # of bytes used

bVal Byte 20 dup(0) type=1, length=20, sizeof=type x lengthof

mov ax, type bVal ax: 00 01
mov ax, lengthof bVal ax: 00 14
mov ax, sizeof bVal ax = 00 14

- label - doesn't have location etc., give a location a different type

dT dword ? loc: 00 4C

- jump [label] - unconditional jump - "go to"

```
mov eax
add eax, dVal
jump end_of_program
```

- loop - works with ecx
  1) ecx is decremented by 1
  2) if (ecx != 0), then jump to the label

- wVal word 15, 20, 32, -1, 6

```
mov ecx, lengthof wVal;      ecx = 5
mov eax, 0;                  set sum to 0;
mov ebx, offset wVal;        ebx = beginning of array;
```

```
stloop:
    add  eax, word ptr [ebx];      sum += element at ebx
    add  ebx, 2;                   ebx points to next element
    loop stloop;
```

-
```
strOne byte "Hello, there!", 0
strTwo byte 40 dup(?)

mov ecx, sizeof strOne;
mov eax, 0;
mov ebx, offset strOne;        mov edx, offset strTwo;
stloop:
    mov al, byte ptr [ebx];
    mov strTwo, al;
    add ebx, type strOne;
    add edx, type strTwo;
loop stloop
```

- use ebx, esi, & edi for pointer registers
- inc & dec is a little faster than add

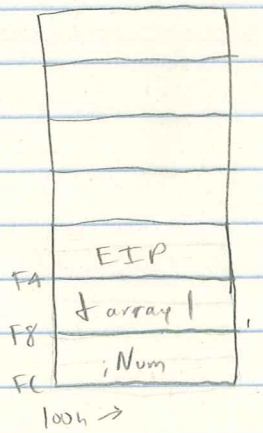
- [label] proc
    push ebp
    move ebp, esp

    pop ebp
    ret
    end proc
; this is the shortest proc, in Java, this is { }

; short function (short, short)
  wVal1 word ?
  wVal2 word ?
  wVal3 word ?
; wVal3 = function (wVal1, wVal2)
  push wVal2
  push wVal1
  call function
  mov wVal3, ax ; or mov wVal3, word ptr eax
  add esp, 4
; sVal = f1(iVal, sVal1, sVal2)
  push sVal2
  push sVal1
  push iVal
  call f1
  mov sVal, ax
  add esp, 8

- ;Total= SumArray(&array1, ;Num) ; C syntax
  push ;Num
  push offset array1
  call SumArray
  mov ;Total, eax
  add esp, 8

1) EIP is pushed onto stack
2) EIP is loaded with address of procedure
   when the function is entered, the old EIP value is on the top of the stack
3) when the function is entered, ebp is pushed onto stack
   - the programmer must push & pop the stack equal times by the end of the method
4) ret copies what is on top of stack to EIP



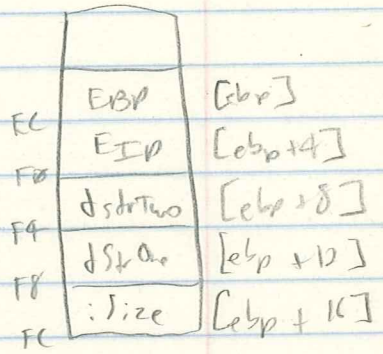
- sZ = addshorts(sX, sY)

```
wX  word  27
wY  word  16
wZ  word  ?
; wZ = addshorts(wX, wY)
    push wY
    push wX
    call addshorts
    mov wZ, eax
    add esp, 4
addshorts proc
    push ebp
    mov ebp,esp
    mov ax, [ebp+10]
    add ax, [ebp+8]
    pop ebp
    ret
```

- strCopy (dstOne, dstTwo, iSize)

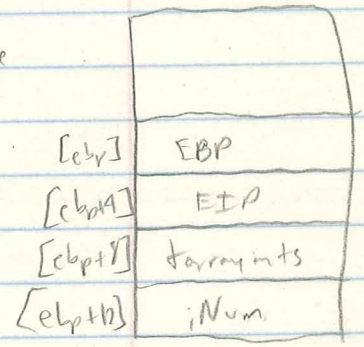
```
    push iSize
    push offset strOne
    push offset strTwo
    call strCopy

    add esp, 8;          ; restore stack pointer
```

```
strCopy proc
    push ebp            ; preserve base pointer
    mov ebp, esp        ; new stack frame
    push eax; push ecx; push edi; push esi;   preserve registers
    mov ecx, [ebp+16]   ; ecx = # of bytes copied
    mov esi, [ebp+12]   ; esi → source str
    mov edi, [ebp+8]    ; edi → dest str
stLoop:
    mov al, [esi]
    mov [edi], al
    inc esi
    inc edi
    loop stLoop;
```



- int AddsUp(&arrayints, iNum) ; write procedure

```
AddsUp proc
✓   push ebp
✓   mov ebp, esp
✓   mov ecx, [ebp+12]
    mov esi, [ebp+8]   ; ✓mov eax, 0;
stLoop:
    mov ebx, [esi]      ; move source into ebx
    add eax, ebx        ; add eax & ebx
    add esi, type arrayints   ; increment esi by # of bytes in type
    loop stLoop         ; restart loop if ecx ≠ 0
```



would also want to preserve ebx & ecx after AddsUp proc

- Compare

cmp op1, op2

what happens is op1 - op2 so that flags get set

still cannot compare memory to memory

```
strName byte "Hello, there!", 0
  mov ebx, offset strName   ; ebx -> array

  cmp byte ptr [ebx], 0
  je finished
```

(je = jump equal; jne = jump not equal; jl = jump less than)

je tests the zero flag

-

```
sub count, 1
jz done
```

jz tests the zero flag

je = jz but use which one makes sense

Assembly

- msgHello db "_____", 0
  puts _____

loopStr:
  mov ecx, 100 ; counter
  ✗ CRLF db 10,13,0 ✓ puts CRLF
  ✗ msgHello db "_____", 0 ✓ puts msgHello
  ~~puts msgHello~~
  loop loopStr

- f1(int, short, int) : int UML
  iVal = f1(iVal1, sVal2, iVal3) Java

```
push iVal3
push sVal2
push iVal1
call f1
mov iVal, eax
add esp, 10
```

- sVal = f1(sVal1, sVal2, bVal1) ; must cast

```
mov al, bVal1
cbw                ; ax = bVal1
cwde               ; eax = bVal1
push eax
push sVal2
mov ax, sVal1
cwde
push eax
```

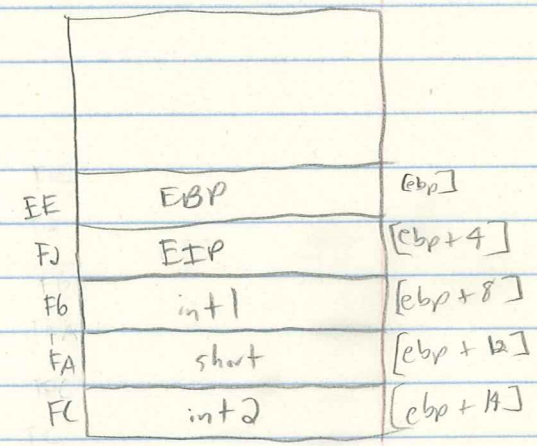
```
call f1
mov sVal, ax
add esp, 10
```

- f1(int, short, int) : int
  iVal = f1(&array1, sVal, &array2)

UML

Java

```
push offset array2
push sVal
push offset array1
call f1
mov iVal, eax
add esp, 10
```



- f2(int, short) : void
  f2(iVal, sVal);

```
push sVal
push iVal
call f2
add esp, 6
```

- f2(int, short) : void
  f2(sVal, bVal)

```
mov al, bVal
cbw                 ; ax = bVal
push ax
mov ax, sVal
cwde                ; eax = sVal
push eax
call f2
add esp, 6
```

- mul - multiply two positive (even if one is negative)
  imul - sign is considered

imul op
can only multiply 1x1, 2x2, or 4x4
;Val = bVal x sVal ; cannot do!
what kind of multiply is determined by operand
op1 is multiplier
;Mul bVal 1 byte multiplier → al, result is in ax
imul sVal 2 byte ... → ax, result is dx:ax
imul dVal 4 byte → eax, result is edx:eax

bVal byte 5
sVal word 10
;Val dword 15

sVal2 = bVal1 x bVal2 ; legal
;Val = sVal1 x sVal2 ; legal
LVal = ;Val1 x ;Val ; legal

- 4 x 5
  mov al, 4
  mov bl, 5
  imul bl ;AX = 20

- 300 x 10
  mov ax, 300
  mov bx, 10
  imul bx ; dx:ax

- iVal = sVal1 x sVal2

```
mov ax, sVal1
imul sVal2          ; dx:ax
mov word ptr iVal, ax
mov word ptr iVal+2, dx
```

or

```
mov ebx, offset iVal
mov [ebx], ax
mov [ebx+2], dx
```

- sVal = sVal1 x sVal2

```
mov ax, sVal
imul sVal2          ; dx:ax
mov sVal, ax        ; dx is lost
```

- LVal = lVal1 x lVal2

```
mov eax, lVal
imul lVal2          ; edx:eax
mov dword ptr LVal, eax
mov dword ptr LVal+4, edx
```

— or —

```
mov ebx, offset LVal
mov ebx, eax
mov ebx+4, edx
```

- byte x byte → anything but sign overflows into AH, CF & OF is set
  word x word → " " dx, CF & OF is set
  dword x dword → " " edx, CF & OF is set

- iVal = sVal x bVal ; convert byte to short

```
mov al, bVal
cbw             ; ax = bVal
imul sVal       ; dx:ax
mov ebx, offset iVal
mov [ebx], ax
mov [ebx+2], dx
```

- iVal = iVal1 x bVal

```
mov al, bVal
cbw             ; ax = bVal
cwde            ; eax = bVal
imul iVal1      ; edx:eax
mov iVal, eax   ; edx is lost
```

- ascint (&string) : int
  intasc (&string, in) : void
  gets
  puts

```
extern ascint:Near32, intasc:Near32
```

for ascint - can only accept numbers
sets carry flag if it's an invalid string
sets overflow flag if it's too big

```
call
jc                      ; use jump carry
jo                      ; use jump overflow
mov :Val, eax
add esp, 4
jmp
```

Assembly

- look at questions in Chapter 3, 4, & 5 will be on Exam
- no division on the Exam

- ;Val = sVal1 x sVal2

```
mov ax, sVal1
imul sVal2          ; dx:ax
mov iVal1, eax      mov word ptr iVal, ax      or
                    mov word ptr iVal+2, dx
```

- qVal = bVal1 x lVal2

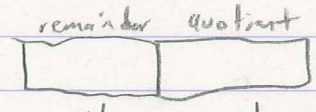
```
mov al, bVal1
cbw                 ; ax = bVal1
cwde                ; eax = bVal1
mov ebx, dword ptr lVal2    ; cannot go down
imul ebx            ; edx:eax
mov ebx, offset qVal
mov [ebx], eax
mov [ebx+4], edx
```

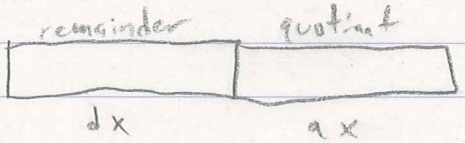
```
mov ebx, offset iVal
mov [ebx], ax
mov [ebx+2], dx
```

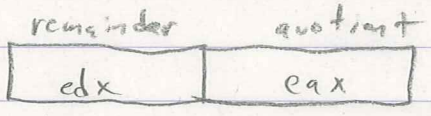
- Division - idiv (keeps sign)

```
idiv op1
```

if size of op1 = 1, then quotient is in ax register
if size of op1 = 2, then quotient should be in dx:ax
if size = 4, then should be in edx:eax







- program crashes if dividing by 0, check for this
- when multiplying, you get overflow if anything other than sign bit is there
- bT byte 7
  sX word 29

```
mov ax, sX
idiv bT          ; quotient is in al, remainder is in ah
```

remainder	quotient
01	04

- dividend must be twice as big as quotient
  2 bytes / 1 byte = 1 byte

```
sY = sX / bT
mov ax, sX
idiv bT
cbw
mov sY, ax
```

- sY = bT / sX

```
mov al, bT
cbw              ; ax = bT
cwd              ; dx:ax = bT
idiv sX          ; dx = remainder, ax = quotient
mov sY, ax
```

- ; X = sY / sX

```
mov ax, sY
cwd                 ; dx:ax
idiv sX             ; dx = remainder, ax = quotient
```

x `mov iX, dword ptr ax`

```
cwde
mov iX, eax
```

- ; X = sY / bT

```
mov ax, sY
idiv bT             ; ah = remainder, al = quotient
cbw                 ; ax = quotient
cwde                ; eax = quotient
mov iX, eax         ; iX = eax
```

- ; Y = sY % bVal

```
mov ax, sY          ; ax = sY
idiv bT             ; ah = remainder, al = quotient
mov al, ah          ; move remainder into al
cbw                 ; ax = remainder
cwde                ; eax = remainder
mov iY, eax
```

- sY = iVal / sVal;

x

```
mov eax, iVal       ; eax = iVal
idiv sVal           ; dx = remainder, ax = quotient
mov sY, ax          ; sY = ax
```

```
mov dx, word ptr iVal + 2
mov ax, word ptr iVal
```

OR

```
movsx ebx, sVal                 mov sY, ax
mov eax, iVal
cdq                 ; edx:eax = iVal
idiv ebx
```

- ; X = sVal % iVal

```
mov ax, sVal
cwde          ; eax = sVal
cdq           ; edx:eax = sVal
idiv iVal     ; eax = quotient
mov iX, edx   ; iX = remainder
```

- iVal = sVal % bVal;

```
mov  ax, sVal
idiv bVal     ; ah = remainder, al = quotient
mov al, ah    ; al = ah
cbw           ; ax
cwde          ; eax
mov iVal, eax ; iVal = eax
```

- Exam Review
- assembling, linking, loading
assembler gives object file - program in machine code
linker - adds external libraries
loaders - uses offsets to resolve actual addresses

Hollerith code for A = 12-1