x86 Assembler

Computer Systems: Sections 3.1-3.5

Disclaimer

I am not an x86 assembler expert.

I have never written an x86 assembler program.

(I am proficient in IBM S/360 Assembler and LC3 Assembler.)

You will NOT be expected to WRITE x86 assembler.

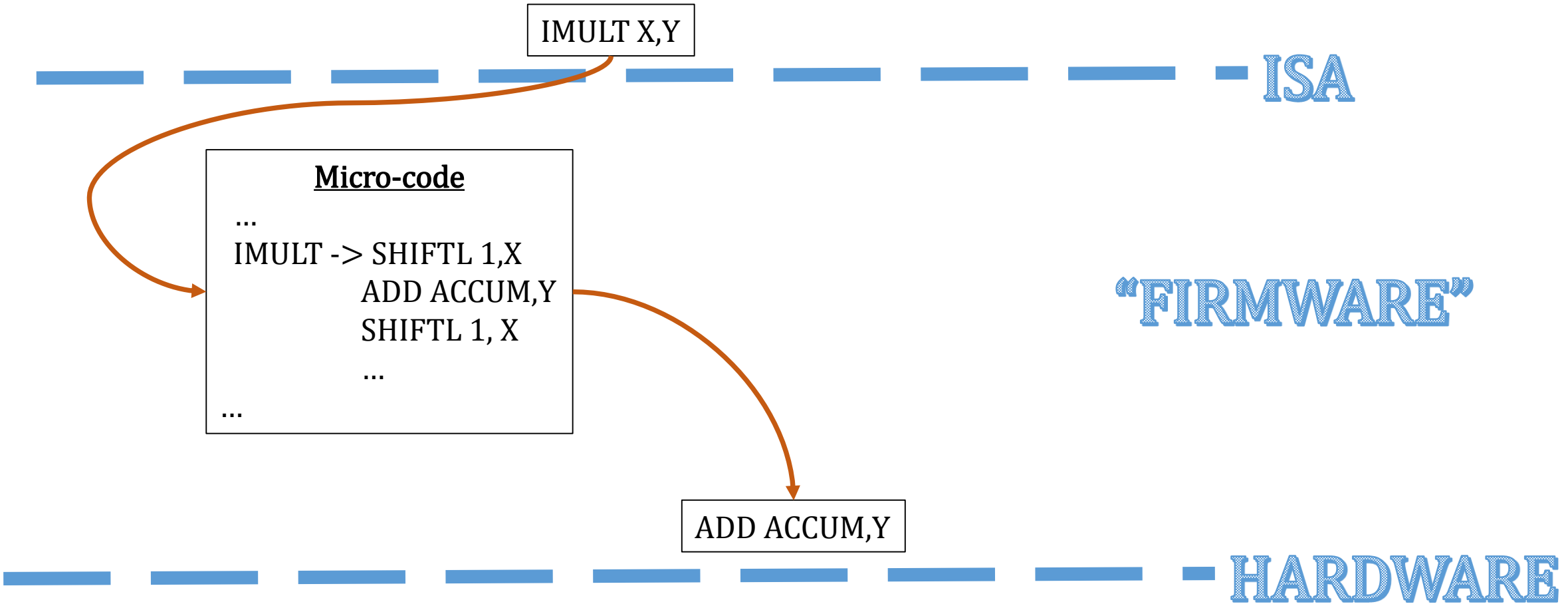
You WILL be expected to be able to READ x86 assembler!

x86 Generalizations

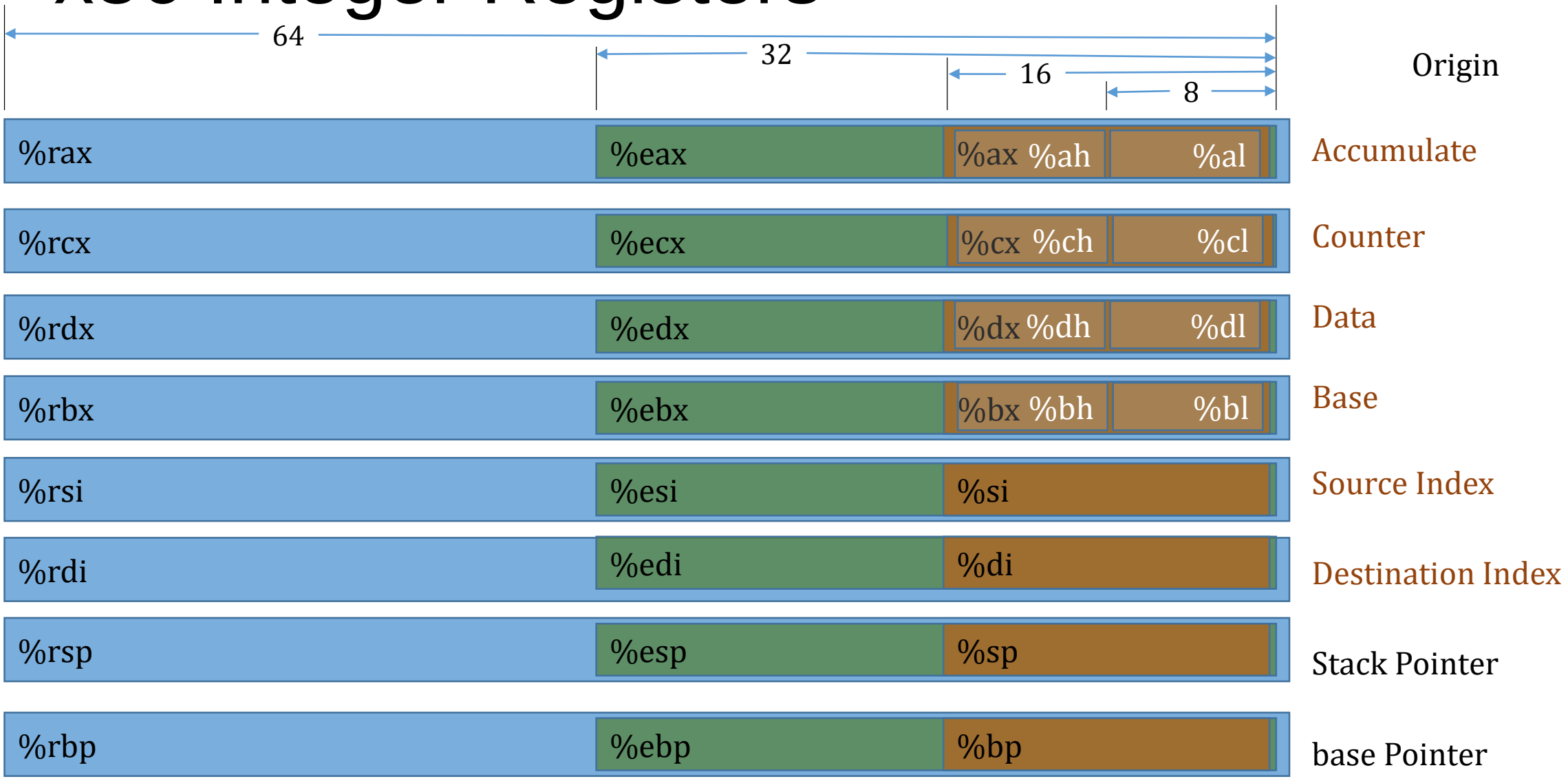
- Designed to be Compiler Friendly
  - Stack manipulation instructions
  - ALU uses data from memory as well as registers!
- Downward Compatible
  - Newer versions are supersets of older versions
- CISC (Complex Instruction Set Computing)

RISC Concept

CISC Concept



x86 Integer Registers



x86 Data "Types"

No type checking - Instruction and/or context implies data type

- Arithmetic instructions treat operands as numbers
  - Either signed or unsigned!
- Instruction suffix used to identify precision of arguments
  - b – 1 byte (8 bits)
  - w – word (2 bytes, 16 bits)
  - l – long word (4 bytes, 32 bits)
  - q – quad word (8 bytes, 64 bits)
- With no suffix, register type implies precision of arguments
  - ah/al – b – 8 bits
  - ax – w – 16 bits
  - eax – l – 32 bits
  - rax – q – 64 bits
- Floating point – 4, 8, or 10 bytes

x86 Assembler Syntax

[<label>:] <mnemonic> arg1[,arg2…] [; comment]

- Label optional – identifies start of this line
- mnemonic – See http://ref.x86asm.net/ for a complete list
- Up to 4 arguments
- Comment ends at the end of this line

Assembler Argument Generalities

- Reads LEFT to RIGHT ⇒ (AT&T syntax… INTEL syntax is RtoL)
  - mov 5,eax ; move the constant 5 => eax
  - add bx, ax ; add bx + ax => ax (ax = bx+ax;)
- Arguments may be: register, constant value, memory reference
- Only ONE argument may be a memory reference!
  - at least one argument must be a register or constant
- Optional argument prefixes
  - % - register e.g. “movl 5,%eax”
  - $ - constant value e.g. “movl $5,%eax”

Constant (literal) values

Similar to C Conventions….

- Numbers are decimal by default, octal if preceded by 0, hex if preceded by 0x
- Single characters are enclosed in single quotes, including special characters such as ‘\n’, ‘\t’
- Strings are arrays of characters enclosed in double quotes
- Labels may be used in place of addresses

Basic x86 addressing modes

- Normal: Register contains address of target in memory
  - Parenthesis indicate “Use Value at this address”
  - movl (%ecx),%eax ; Put the value at memory[ecx] into eax
- Displacement: Register is near address of target in memory
  - Offset from register specified before parenthesis
  - movl 8(%ebp),%edx ; Put the value at 8 past the base pointer into edx

- See http://en.wikipedia.org/wiki/X86#Addressing_modes if you need the entire story – but it’s complicated

Table Addressing Mode

- Generally: <OFFSET>(<BASEREG>,<ROWREG>,<WIDTH>)
  - e.g. 4(%ebx,%ecx,12)
- Address: = (<BASEREG>)+(ROWREG)*<WIDTH> + <OFFSET>
  - e.g. (%ebx)+(%ecx)*12+4

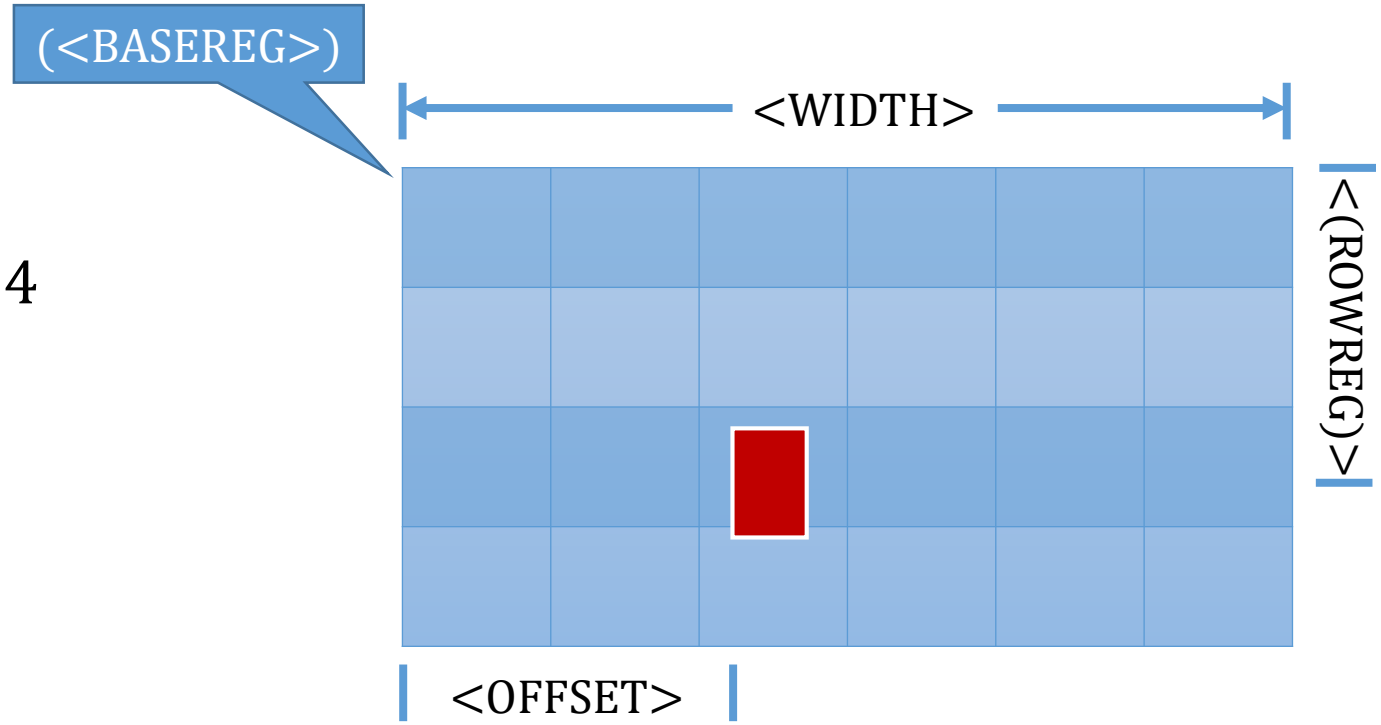
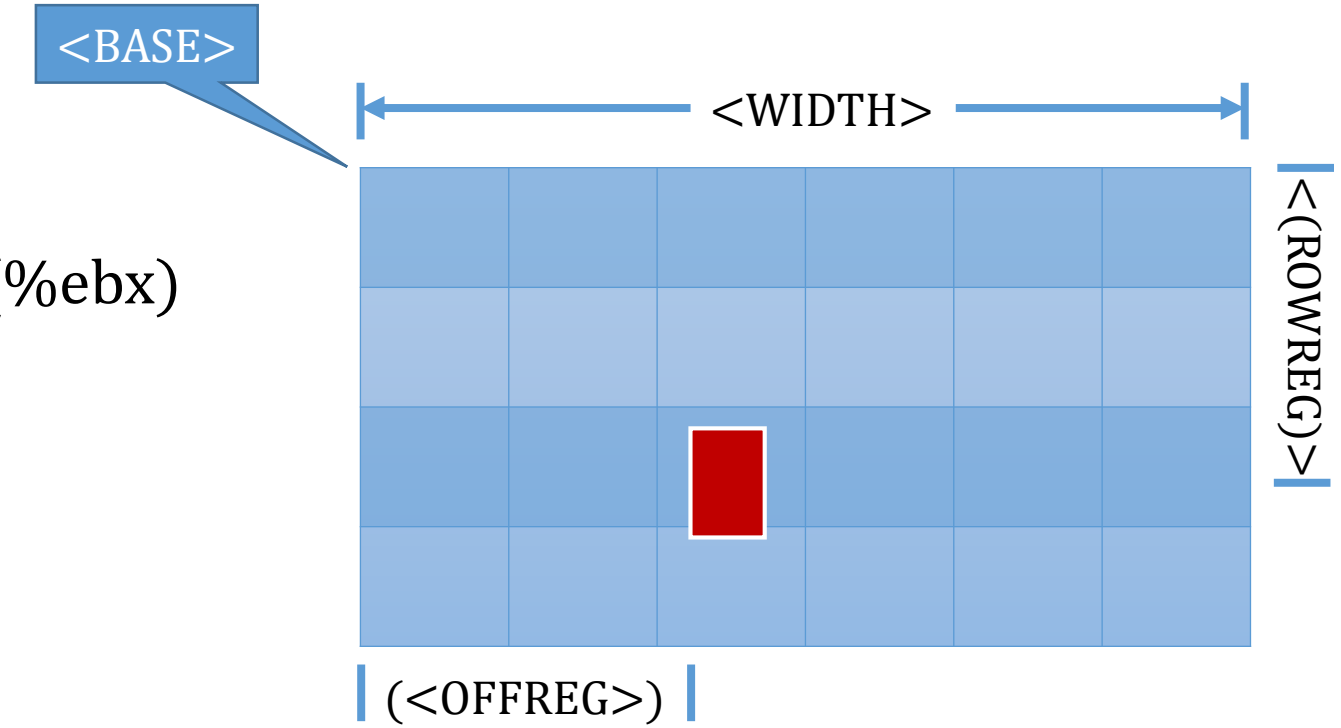


Table Addressing Mode

- Also: <BASE>(<COLREG>,<ROWREG>,<WIDTH>)
  - e.g. mytable(%ebx,%ecx,12)
- Address: = <BASE> + (<COLREG>)+(ROWREG)*<WIDTH>
  - e.g. mytable+(%ecx)*12+(%ebx)



The MOV instruction

- Most often used instruction!
- More “copy” than “move”
- Copies 1,2,4, or 8 bytes from ARG1 to ARG2

```
mov $-12,%eax ; put -12 into 4 byte eax register
mov  $0xffff,(%esp) ; put -1 at top of stack
mov 12(%ebp),%eax ; copy data at 12 past base pointer into eax
movb chartab(%eax,%ebx,14),%al
                    ; copy byte at row %ebx, col %eax of chartab into %al
```

Logic Instructions

- Standard 2 arg: and or xor shl shr

```
and %ebx,%eax ; eax=eax & ebx
shr $4,%eax; eax = (unsigned)eax>>4
```

- Single argument: not neg

```
not %eax ; flip bits in eax
neg %ebx; take two's complement (flip bits and add 1) of ebx
```

Arithmetic Instructions

- Standard integer arithmetic: add sub
  add $10,(%eax); (*eax)=(*eax)+10
  sub $4,%esp ; esp=esp-4 (move stack pointer down)
- "Special" integer arithmetic: imul idiv
  - imul cannot write to memory
  - idiv divides register pair (EDX:EAX) and puts quotient/remainder back
- Single argument: inc dec
  inc %eax; eax=eax+1 – same as add eax,1
  dec (%esp) ; decrement the value at the top of the stack by 1
- Floating Point Instructions

Dealing with Pointers

- Load effective address: lea
  - Used for implicit arrays/structures, etc.
  - Calculates address from first argument, and writes that address to second

```
    lea $8(%esp),%eax ; %eax->array[0]
    move $0,%edx ; sum=0
loop: add (%eax),%edx ; sum=sum+(%eax)
    add $4,%eax ; increment pointer
    cmp $0,(%eax) ; is (%eax) >=0?
    jge loop ; Yes… continue
```

Not done yet…

- Next – x86 control instructions…