**What is Assembly Language?**

- In a high level language (HLL), one line of code usually translates to 2, 3 or more machine instructions.
- Some statements may translate to hundreds or thousands of machine instructions.
- In Assembly Language (AL), one line of code translates to one machine instruction.
- AL is a "human readable" form of machine language.
- HLLs are designed to be "machine-independent". But machine dependencies are almost impossible to eliminate.
- ALs are NOT machine-independent. Each different machine (processor) has a different machine language. Any particular machine can have more than one assembly language (NASM and MASM).

**NASM and MASM**

- We will use NASM (Netwide Assembler) in this course.
- NASM is operating system independent.
  - One of the two widely used Linux assemblers.
  - The other is GAS (GNU assembler).
- The syntax differs significantly in many ways from MASM (Microsoft Assembler).
  - MASM syntax is a standard that is almost always understood by other x86 assemblers (TASM, CHASM, A386, etc.).
  - MASM syntax has some significant defects that makes coding prone to error. Many of these are rectified in NASM.
- We will not cover NASM syntax in full depth.
  - We are interested in a basic machine interface, NOT a production assembler language.
  - NASM has many syntactic constructs similar to C.
  - NASM has an extensive preprocessor similar to C's.

**Basic Elements of NASM Assembler**

- **Character Set**
  - Letters a...z A...Z ()
  - Digits 0..9
  - Special Chars ? @ $ . ~
- NASM (unlike most assemblers) is case-sensitive with respect to labels and variables.
- It is not case-sensitive with respect to keywords, mnemonics, register names, directives, etc.
- Special Characters

**Basic Elements: Integers**

- Integers: numeric digits (incl A-F) with no decimal point.
- May include radix specifier at end:
  - b or y binary
  - d decimal
  - h hexadecimal
  - q octal
- Examples
  - 200 decimal (default)
  - 200d decimal
  - 200h hex
  - 200q octal
  - 1011011b binary
- Note: Hexadecimal literals start with A-F must have a leading 0.
  - Consider the constant AH = 10d. This is the name of a register. Consider the constant 205d = FH. This could be the name of a variable.
- NASM also supports other syntactic forms, e.g., 0x08 or $c8

**Basic Elements: Statements**

- Syntactic Form: 
  - [label:] [mnemonic] [operands] [;comment]
- (Note: [label] can also be [name])
  - Variable names are used in data definitions.
  - Labels are used to identify locations in code.
- Note that ALL parts are optional => blank lines are legal.
- Statements are free form; they need not be formed into columns.
- Statement must be on a single line, max 128 chars.
- Example:
  - L100: add eax, edx ; add subtotal to running total
  - Labels often appear on a separate line for code clarity:
    - L100:
      - add eax, edx ; add subtotal to running total

**Litersals**

- Literals are values are known or calculated at assembly time.
- Examples:
  - 'This is a string constant'
  - "So is this"
  - Backquoted strings can use escape chars `\`
  - 123
  - 1.2
  - 0FAAh
  - $1A01
  - 0x1A01
Basic Elements: Labels and Names

- Names identify labels, variables, symbols, or keywords
- May contain: A...Z, a...z
- 0...9
- ? _ @ $ . ~
- NASM is case-sensitive (unlike most x86 assemblers)
- First character must be a letter, _ or "" (which has a special meaning in NASM as a "local label" which can be redefined)
- Names cannot match a reserved word (and there are many!)

Types of Assembler "Statements"

1. Directives
   limit EQU 100 ; defines a symbol limit
   %define limit 100 ; like C #define
   CPU P4 ; Use Pentium 4 instruction set
2. Data Definitions
   msg db 'Welcome to Assembler!'
   db 0Dh, 0Ah
   count dd 0
   mydat dd 1,2,3,4,5
   read 100 ; reserves 400 bytes of
   ; uninitialized data
3. Instructions
   mov eax, ebx
   add ecx, 10

Variables, Labels and Constants

- count1 db 100 ; variable called count1
- count2 times 100 db (0); variable called count2 (100 bytes)
- count3 EQU 100 ; constant called count3
- strl DB 'This is a string' ; variable strl 16 bytes
- strlen EQU $-strl ; const strlen = 16
- label1: mov eax, 0 ; label1 is the address of instruction
  ;....
  jmp label1
- Notes: count1 is the address of a single byte, count2 is the address of the first byte of 100 bytes of storage
- Count3 does not allocate storage; it is a textual EQUate (symbolic substitution; similar to C #define)
- The $ has a special meaning: the location counter

Names, Labels and Colons

- count1 db 100 ; variable called count1
- label: mov eax, 0 ; label is the address of instruction
- jmp label
- Notes: count1 is the address of a single byte, count2 is the address of the first byte of 100 bytes of storage
- Count3 does not allocate storage; it is a textual EQUate (symbolic substitution; similar to C #define)
- The $ has a special meaning: the location counter

What is a Variable Anyway?

- Variable names such as count1 and count2 represent addresses
- The only information that the assembler associates with a variable is the location counter (current address) of the item you declared at that address
- Arrays are a fiction imposed by HLLs. They do not exist at the machine level. The assembler sees no difference between count1 and count2 except for the address

The Location Counter

- The symbol $ refers to the location counter
- As the assembler processes source code, it emits either code or data into the object code.
- The location counter is incremented for each byte emitted
- In the example above, count1 is numerically the same value as strl (which is an address)
- With $-strl the assembler performs the arithmetic to compute the length of strl
- Note the use strl in this expression as a numeric value (the address of the first byte)
Keywords

- Keywords can be register names, instruction mnemonics, assembler pseudo-ops and directives.
- Examples:
  - eax mov proc db
  - ah add sub .IF
  - ASSUME END .WHILE SIGNED
- If you accidentally use a keyword as a variable name, you may get a misleading error message.

Hello World (MASM MS-DOS)!

```asm
.dosseg
.model small
.stack 100H
.data
hello_message db 'Hello, World!',0dh,0ah,'$'
.code
main proc
    mov ax, @data
    mov ds,ax
    mov ah,09
    mov dx,offset hello_message
    int 21H
    mov ax,4c00h
    int 21h
main endp
.end main
```

Hello World (NASM Linux)!

```asm
SECTION .data ; data section
msg: db "Hello World",10 ; the string to print, 10=cr
len: equ $-msg ; len is a value, not an address
SECTION .text ; code section
main:
    mov edx,len ; arg3, length of string to print
    mov ecx,msg ; arg2, pointer to string
    mov ebx,1 ; arg1, where to write, screen
    mov eax,4 ; write sysout command to int 80 hex
    int 0x80 ; interrupt 80 hex, call kernel
    mov ebx,0 ; exit code, 0=normal
    mov eax,1 ; exit command to kernel
    int 0x80 ; interrupt 80 hex, call kernel
```

Hello World (NASM Windows)!

```asm
section .text
_main:
    push message
    call _printf
    add esp,4
    ret
message:
    db 'Hello, World', 10, 0
```

Hello World (NASM Linked with C - Windows)

```c
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

Hello World (NASM Linked with C - Linux)

```c
#include <stdio.h>
int main(void)
{
    printf("Hello, World!");
    return 0;
}```
Object Code, Load Images, Linkage

- In a high-level language, the process of producing an executable program looks like this:
  - Source code → object code → executable program
  - Compile → Link
- Object code modules are language-independent linker inputs
- An executable program may not contain "pure machine language": it may have unresolved memory references, function calls, etc.

Executable Program → Load Image → Run
  - OS program loader → Load IP

- The operating system supplies a "program loader" that resolves these references and creates a "load image"
- The load image is pure machine language

Overview and Grouping of x86 Instructions

- Assembler Instructions can be categorized by function:
  1. Data Movement
  2. Arithmetic and Comparison
  3. Bit and Logical Operations
  4. Character Operations
  5. Control Flow
  6. Miscellaneous
  7. Semaphore and Process Sync (486+)
  8. System Programming Instructions

- In the list below, 16/32 bit instructions are in all uppercase while 32-bit only instructions are lowercase
- Assembler mnemonics are NOT case-sensitive; either can be used

Data Movement Instructions

- Moving Data:
  - MOV, PUSH, POP, XCHG, PUSHA, POPA
  - movsx, movzx, pushd, popd, pushad, popad

- I/O: IN, OUT, ins, outs

- Address Loading: LDS, LEA, LES, lfs, lss, lgs

- Flag Transfer:
  - LAHF, SAHF, PUSHF, POPF, pushf, popfd

Arithmetic and Comparison Instructions

- Addition:
  - ADC, ADD, INC

- Subtraction and Comparison
  - CMP, DEC, NEG, SBB, SUB, setcc

- Multiplication:
  - MUL, IMUL

- Division:
  - DIV, IDIV

- Sign Extension:
  - CBW, CWD, cwde, cdq

- BCD Arithmetic:
  - DAA, DAS, AAA, AAS, AAM, AAD

Bit and Logical Operations

- Logical:
  - AND, TEST, OR, XOR, NOT
  - bt, btr, bts, btc, bsf, bar

- Shifts:
  - SAL, SHL, SAR, SHR
  - shld, shrd

- Rotations:
  - RCL, RCR, ROL, ROR

Character Operations

- Table Lookup
  - XLAT

- Block (String) Moves:
  - MOVSB, LODS, STOS

- Block (String) Comparison:
  - CMPS, SCAS

  The block instructions can be suffixed with an operand size, for example MOVSB, MOVSW, moved

- Repetition Prefixes:
  - REP, REPFE, REPZ, REPNE, REPNZ
Control Flow

- Unconditional Jump: 
  JMP
- Subroutine Call: 
  CALL, RET
- Conditional Jumps:
  JA, JNE, JB, JBE, JC, JCXZ, JE, JG, JGE, JL, JLE, JNA, JNBE, JNC, JNE, JNG, JNLE, JNPE, JNS, JNZ, JPO, JS, JL

The above conditionals are not all distinct
- Loops:
  LOOP, LOOPNE, LOOPD, LOOPDE
- Interrupts:
  INT, INTO, IRET
- Flag Control:
  CLI, CLD, CMC, STC, STD, STI

System Instructions

- 80286+
  lgdt, lidt, lidt, egdh, stdh, stdt, aspl, iar, lal, verr, verw, clts, smaw, lsw
- 80386+
  invd, lmovpg, whlnvd, mov (cr, dr, tr)
- The miscellaneous and systems instructions will not be covered in detail in this course
- NOP is worth mentioning however it does nothing at but occupy space and slow down programs (it does take time to execute)
- A NOP instruction is a one-byte instruction that takes a small amount of time (1-3 clock ticks) to execute
- Used for:
  leaving room to patch machine code
  aligning instructions on address boundaries
  very short delays in loops
  Implemented as xchg eax, eax (exchange eax with itself)

Data Definition

- A variable is a symbolic name for a location in memory where some data are stored
- Variables are identified by names
  The offset address associated with the name is the distance from the beginning of the segment to the beginning of the variable
- A variable name does not indicate how much storage is allocated
- Assemblers associate some minimal type information with names to help prevent errors
  NASM allows you to shoot yourself in the foot a bit more than other assemblers

Initialized Data Definitions

- Allocates storage for one or more values
- The name is optional
- May be followed by a list of values
- Only the first value is "named"
- A variable name is just a synonym for a numeric address
- Other values can be accessed via an offset from the name:
  foo
  dw 3, 4, 99, 1202, -3
  ...
  mov ax, foo+2; load second element of array
  ; same as mov ax, [foo+2]

Initialized Data

- NASM initialized data directives go into the data segment
  segment .data
- Data definition directives cause the assembler to allocate storage and associate some type (size) information with the name
  Description | Size | Type
  ------ | ---- | ----
  db | Define Byte | 1 Byte
  dw | Define Word | 2 Word
  dd | Define Doubleword | 4 Doubleword
  df, dp | Define FarWord | 6 far pointer
  dq | Define Quadword | 8 quad word
  dt | Define Tenbyte | 10 tenbyte
- Programmers conventionally refer to "dword", "qword" and even "fword" instead of the full names
Data Definition Syntax

- NASM has different syntax for initialized and uninitialized values
- INITIALIZED VALUES:
  - \[ \text{name} \ Dx \ \text{initialvalue} \ [,\text{initialvalue}] \]
    where \(Dx\) is
    - db byte
    - dw word
    - dd double word
    - df far pointer (6-byte)
    - dq quad word
    - dt ten-byte
- For this course, we will generally use db, dw, and dd, dq only

Uninitialized Data

- NASM uses different syntax for uninitialized data, which goes into the "bss" segment
- \text{segment .bss}
- Just like \(Dxx\) but use resXX with the number of objects needed
  - buffer resb 64 ; reserve 64 bytes
  - wordvar resw 1 ; reserve a word
  - reallarray resd 10 ; array of ten words
  - qvar req 1 ; 1 qword

Historical Relic

- BSS came from "Block Started by Symbol", an assembler for IBM 704 in the 1950s
- Almost universally used by C and C++ compilers today

Examples

- This example also shows how little-endian memory works
  - \text{mag}
    - db "ASM is fun!",0dh,0ah
    - db "(for masochists)"
  - \text{byt}
    - db 12h, 34h, 56h, 78h
  - \text{wrd}
    - dw 1234h, 5678h
  - \text{dwd}
    - dd 12345678h
  - \text{qwd}
    - dq 12345678h
  - \text{one dd}
    - 1.0
- Displayed by debug:
  - 0EB4:0100  41 53 4D 20 69 73 20 66
  - 0EB4:0108  75 6E 21 0D 0A 28 66 6F
  - 0EB4:0110  72 20 6D 61 73 6F 63 68
  - 0EB4:0118  69 73 74 73 29 12 34 56
  - 0EB4:0120  78 34 12 78 56 78 56 34
  - 0EB4:0128  12 78 56 34 12 00 00 00
  - 0EB4:0130  00 00 00 80 3F 72 6F 6D

IEEE Floats

- Note that assemblers will assemble IEEE format numbers.
  - INT100DD 100 ; 32 bit integer with value 100
  - FLT100DD 100.0 ; IEEE float
  - DBL100DD 100.0 ; IEEE double (64-bit)
  - EXT100DD 100.0 ; IEEE extended (80-bit)

- The decimal point cues the assembler to construct and emit a floating point number
- Numbers with a decimal point can only appear in data definitions; they are never used as literals in code

Pointers in Data Definitions

- The offset of a variable or instruction can be stored in a pointer
  - list dw 1,2,3,4
  - lptr dd list
  - str dd 'This is a string',0dh,0ah
  - spt dd str
  - p2 dd lptr
  - cls dd clear_screen

  - clear_screen:
    - \text{asm code to clear the screen}
    - ret
- Lptr is initialized to the address represented by list; likewise for spt and p2
- p2 is a "pointer to a pointer"
- cls is a pointer to a function
The TIMES prefix
- The TIMES prefix causes the data definition or instruction to be assembled multiple times. Ex:
  ```
  zerobuf TIMES 2048 db 0
  ```
- The argument to TIMES is an expression
  ```
  buffer: db 'hello, world'
  times 64-#buffer db '
  ```
- TIMES can also be applied to instructions to provide trivially unrolled loops
  ```
  TIMES 32 add eax, edx
  ```
- In `.bss` (uninitialized data) these are equivalent:
  ```
  TIMES 1000 read 1
  read 1000
  ```
  But the latter assembles 1000 times faster

Basic Instructions
- We will start taking a tour through the basic parts of the x86 instruction set
- For each instruction, we need to look at the syntax, semantics (effects on operands and flags) and what legal operands may be used
- Operand abbreviations:
  ```
  reg 8,16, or 32-bit register except segreg
  regs segment register
  imm immediate operand
  mem 8,16, or 32-bit memory cell
  mem16 16-bit word in memory
  reg8, mem8 8-bit register/memory
  reg32, mem32 32-bit register/memory
  ```

MOV (Move)
- Used to copy (not move) values from one place to another.
- Some other architectures use two or more different instructions: LOAD (mem->reg) and STORE (reg->mem)
- Syntax:
  ```
  MOV destination, source
  ```
- NOTE THE ORDER OF THE OPERANDS!
- Semantics:
  ```
  destination <- source; source unaltered
  Flags: ODITSZAPC unchanged
  ```
- Operands:
  ```
  reg, reg
  mem, reg
  reg, mem
  reg16, segreg
  mem16, segreg
  segreg, reg16
  segreg, mem16
  ```

Assembler vs. HLL
- The illegal instruction `mov eax, ebx/16` looks a lot like the assignment statement `a = b / 16;`
- But in assembler we have to “operate” the machine.
  ```
  mov eax, ebx ; copy eax to ebx
  push edx ; save edx
  mov dx, 16 ; prep for 32-bit division
  div dx ; eax = eax/16
  pop edx ; restore register
  ```
- Note: there is an easier way
  ```
  mov eax, ebx ; copy ax to bx
  sar eax, 4 ; sar = Shift Arithmetic Right
  ```

Operands
- Instructions with register operands are the most efficient
  ```
  2-3 times faster than L1 cache access
  ```
- Immediate operands are constants
- Immediate operands are coded into the instruction itself
  ```
  mov eax, 5 assembles the “5” into an 8 or 32-bit word within the instruction
  ```
- Note that immediate operands can written in assembly language as expressions that are evaluated at time of assembly
  ```
  mov eax, 65536/4
  ```
- Expression used for immediate operands must be resolvable by the assembler

Direct Operands
- A direct operand refers to the contents of memory at the offset of a variable
  ```
  count db 0
  ```
- ```
  mov al, [count]
  ```
- This is called a “direct” operand because it is a direct memory access
- The address of the operand is encoded into the instruction
- This instruction loads from memory at the address given by count
- In debug, if count is at address 4010013Eh, the above instruction would appear as
  ```
  mov al, [4010013E]
  ```
In NASM a variable name is an address.

- Note that while value arithmetic in a statement is NOT legal, address arithmetic often IS legal (remember the function of the bus interface unit)

\[
\text{mov ebx, \{count\} ; OK, load addr of count}
\]

- If the operand is a register, square brackets indicate indirect addressing

\[
\text{mov eax, [ebx] ; eax \leftarrow contents of dword at address ebx}
\]

- If the operand is a number, square brackets indicate direct addressing

\[
\text{mov [10A8h], eax \leftarrow contents of word at address DS:10A8h}
\]

- If the operand is a variable name, square brackets indicate direct addressing

\[
\text{mov [count], eax \leftarrow contents of word at addr DS:count}
\]

- Remember that variable names are symbols representing addresses.

\[
\text{mov ebx, \{count\} + 2 ; OK, Load 32-bit located at address of count + 2}
\]

\[
\text{mov ebx, \{count\} + 2 ; ILLEGAL}
\]

\[
\text{mov eax, ds:ebx+esi ; Load at address ds:ebx+esi}
\]

\[
\text{mov ds, ax \leftarrow content of ax}
\]

\[
\text{mov [eax], 0100h \leftarrow Word at address ds:edx \leftarrow 100h}
\]

\[
\text{mov [0122h], es \leftarrow Word at address DS:0122H \leftarrow content of es}
\]

\[
\text{mov es, ax \leftarrow Contents of ax}
\]

\[
\text{MOV [SaveEAX], eax \leftarrow Stores contents of eax into SaveEAX}
\]

\[
\text{MOV SaveEAX, eax \leftarrow Not legal in NASM}
\]

\[
\text{mov eax, ebx+4} \quad \text{Not legal. Requires two instructions: mov eax, ebx
add eax, 4}
\]

- The bus interface unit has adder circuitry used in calculating effective addresses

The literal value 4 in [ebx+4] is called a displacement

Displacements can also be used variable names are involved:

\[
\text{mov eax, [data+4] \leftarrow eax \leftarrow contents of dword at address data+4}
\]

\[
\text{mov eax, [array] \leftarrow contents of word at addr array}
\]

\[
\text{mov eax, [array+4] \leftarrow 2nd element of array}
\]

\[
\text{mov ebx, [array+8] \leftarrow store to 3rd element}
\]

\[
\text{mov ebx, [array+12] \leftarrow 4th element of array}
\]

- A memory operand is really a numeric constant at run-time—the address of the variable

- Can use offset from an address to access additional memory

Example

```
data
array dd 11,12,13,14,15,16 ; array of 6 words
code
mov eax, array ; load first element
mov ebx, [array+4] ; 2nd element of array
add ebx, eax ; add the two elements
mov [array+8], ebx ; store to 3rd element
```

- Note that the MOV mnemonic represents seven different machine language instructions. More on this later
Assembler is not C or Java

- In C or Java indexes in an array are compiled to correct element size:
  
  - C/Java
  - Asm equivalent
  
    ```
    int x, y;
    int a[5];
    ...
    x = 5;
    y = a[1];
    a[2] = x + y;
    ```

- In assembler we have offsets not indices (in example above an int is 4 bytes)

A First Program

- This program prompts the user to enter two numbers and then displays the sum

```
%include "asm_io.inc"
;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output

prompt1 db "Enter a number: ", 0
dont

prompt2 db "Enter another number: ", 0
outmsg1 db "You entered ", 0
outmsg2 db " and ", 0
outmsg3 db " the sum of these is ", 0
```

A First Program-2

```
; ; uninitialized data is put in the .bss segment
;
segment .bss
;
; These labels refer to double words used to store the inputs

input1 resd 1
input2 resd 1
;
; code is put in the .text segment
```
PUSH and POP

- Used to store and retrieve data from the stack.
- PUSH and POP work with words or doublewords only (not bytes).
- To PUSH a byte it has to be extended to a word.
- PUSH Syntax:
  ```
  PUSH source
  PUSHD source
  ```
- Semantics: (16|32)
  1. \( SP \leftarrow SP - 2 \) | \( ESP \leftarrow ESP - 4 \)
  2. \([SS:SP] \leftarrow source\) | \([SS:ESP] \leftarrow source\)
- Flags: ODITSZAPC unchanged
- Operands:
  - reg16 mem16 segreg reg32 mem32
  - reg16 mem16 segreg reg32 mem32

POPD Syntax:

- POPD Syntax:
  ```
  POPD dest
  ```
- Semantics:
  1. \( dest \leftarrow [SS:SP] \) | \( dest \leftarrow [SS:ESP] \)
  2. \( SP \leftarrow SP + 2 \) | \( ESP \leftarrow ESP + 4 \)
- Flags: ODITSZAPC unchanged
- Operands:
  - reg16 mem16 segreg reg32 mem32

Notes:
- esp is decremented as stack grows.
- esp always points to last item on stack, not to the next available word (except when stack is empty).
- POPD syntax only necessary when dest is not a register.
- Other instructions that affect the stack (CALL and RET) will be discussed later.

Variations on a Theme: PUSHA, PUSHF, POPA, POPF

- PUSHF (PUSH Flags)
  - Purpose: Push a copy of the flags register onto the stack.
  - Syntax:
    ```
    PUSHF
    ```
  - Semantics:
    1. \( SP \leftarrow SP - 2 \) | \( ESP \leftarrow ESP - 4 \)
    2. \([SS:SP] \leftarrow Flags\) register | \([SS:ESP] \leftarrow EFLAGS\)
- Flags: ODITSZAPC unchanged
- Operands:
  - reg16 mem16 segreg reg32 mem32
  - imm

POPF (POP Flags)

- Purpose: Word at top of stack copied to Flags register.
- Syntax:
  ```
  POPF
  ```
- Semantics:
  1. \( Flags \leftarrow [SS:SP] \) | \( Eflags \leftarrow [SS:ESP] \)
  2. \( SP \leftarrow SP + 2 \) | \( ESP \leftarrow ESP + 4 \)
- Notes:
  1. PUSHF and POPF can be used to save the state of the flag register before calling a subroutine.
  2. POPF is the only instruction that allows modification of the Trap Flag, except that it is cleared when an INT is executed.
  3. \( O \) = bit 8; \( D \) = bit 9; \( I \) = bit 9; \( T \) = bit 8; \( S \) = bit 7; \( Z \) = bit 6; \( A \) = bit 4; \( P \) = bit 2; \( C \) = bit 0
  4. This instruction and SAHF are the only ways that the x86 can directly modify the SZAP bits in the flags register. Note that the ODIT bits cannot be accessed with POPF, although there are instructions that work directly with D and I.
  5. POPF/POPF have more complex semantics in 386+ processors as OS may not allow direct control of IF.

PUSHA, PUSHAD, POPA, POPAD

- PUSHA and POPA became available on the 80186.
  - PUSH pushes eax, ebx, ecx, edx, original esp, ebp, esi, edi in that order.
  - POPA pops them back in reverse order.
- PUSHAD and POPAD are used to distinguish 32 variants where there might be some question about whether 32 or 16 bit registers are intended.
- We’ll just keep it simple and use PUSHA and POPA.
- Generally PUSHA it is more efficient than pushing individual registers if more than 4 registers must be saved.

XCHG

- How do you swap the contents of two registers? You need a third “place” to put the data.
  ```
  mov [temp],eax
  mov eax,ebx
  mov ebx,[temp]
  ```
- XCHG does this in one instruction:
  ```
  xchg eax,ebx
  ```
- Syntax:
  ```
  XCHG dest1, dest2
  ```
- Semantics:
  ```
  dest1 \leftarrow dest2
  dest2 \leftarrow dest1
  ```
- Flags: ODITSZAPC unchanged
- Operands (order is irrelevant):
  ```
  reg,seg mem,reg mem,mem
  ```
- 1. At least one operand must be in a general register.
- 2. Cannot use XCHG with seg regs or mem/mem.
- 3. Note that the assembler NOP (No Operation) instruction is actually assembled as XCHG eax, eax.
Operand Size Specification

- With many instructions size of operand can be inferred from instruction itself. Examples:
  - PUSH AX
  - POP EAX
  - MOV AX,[EBX]
- With other instructions size of one operand can be inferred from size of the other operand.
  - Examples: (EBX = 007Ah)
    - MOV AX,[EBX] refers to a byte at address [EBX]
    - ADD [EBX], 0FFFFH adds to dword at [EBX]
    - AND [EBX], 0FFH
- What if memory at [EBX] = FF 00 00 00? ADD [EBX], 1 results in 00 10 00 00 for a 16 or 32-bit add
  ADD [EBX], 1 results in 00 10 for a 16-bit add
  ADD [EBX], 1 results in 00 00 for a 16-bit add
  (Remember little-endian ordering: FF 00 is 00 FF)

Operand Size Specifiers

- Register references can be resolved by size of register
- Some memory references can be resolved by known size of other operand
- Memory-immediate references are usually ambiguous
- Operand size specifiers byte, word, dword etc are used to resolve ambiguous references
  - add dword [esi], 1
  - add byte [esi], 1
  - add word [esi], 1
- Because NASM does not associate declared size with variables, you must specify operand size when adding imm to memory
  - myvar dd 0
  - add dword [myvar], 1

Add and Sub

- For both signed and unsigned addition and subtraction
- Remember when using them that addition is commutative but subtraction is not
- Syntax:
  - ADD dest, source
  - SUB dest, source
- Semantics:
  - dest <- dest + source
  - dest <- dest - source
  - Flags: O...SZAPC modified for sum/difference
  - DIT.... unchanged
- Note that addition and subtraction modify ALL status flags
- Operands:
  - reg,reg mem,reg reg,mem
  - reg,immed mem,immed
- Note immediate add to memory: you can do arithmetic without using registers at all.

Clearing a Register

- Often we need to clear a register (set it to 0)
- Preferred technique is to use SUB or XOR (discussed later)
  - sub eax, eax is a small, fast 2 byte instruction
  - mov eax, 0 is a slightly slower 3 (or 6) byte instruction
- Although sub eax, eax and mov eax, 0 have the same effect on EAX, the semantics are different. How?
  - When would you prefer to use mov eax, 0?
- Note that to clear memory, a MOV is required because you cannot do a mem-mem subtract

Using the Flags

- Because ADD and SUB affect the flags, you can test for conditions without using a CMP (compare) instruction:
  - Use this:
    - add ebx, ecx
    - cmp ebx, 0
    - js done
  - Not this:
    - add ebx, ecx
    - cmp ebx, 0
    - je done

  - More about this when we look at conditional jumps

ADC and SBB

- Add with carry and subtract with borrow
- Used to implement arithmetic with words larger than register size
- Syntax:
  - ADC dest source
  - SBB dest source
- Semantics:
  - dest <- dest + source + CF
  - dest <- dest - source - CF
  - Flags: O...SZAPC modified
  - DIT.... unchanged
- Operands:
  - reg,reg mem,reg reg,mem
  - reg,immed mem,immed
  - reg,immed mem,immed
ADC Example

Example: Unsigned 32-bit addition in 16-bit machine

vars dd 2 dup (?) ; two 32-bit vars
mov ax, vars
mov ax, [si]; low-order word
mov dx, [si+2]; high-order word
add [si+4], ax ; add ax into low order word
add [si+6], dx ; add dx into h.o. word with CF

move ax, vars
mov ax, [si] ; AX = FFFE
mov dx, [si+2]; DX = 0000
add [si+4], ax ; FFFE + 0003 = 0001 CF Set
adc [si+6], dx ; 0000 + 0000 + 1(cf) = 1
result: 00010001h = 65536 + 1 = 65537

Compare (CMP)

• Compare (CMP) is subtraction without storage of the result. Only the flags are affected
• Syntax:
  CMP dest, source
• Semantics:
  compute dest - source and modify flags
Flags: O...SZAPC modified
.DIT...... unchanged
• Operands:
  reg, reg mem, reg reg, mem
  reg, immed mem, immed
• Notes:
  Operands same as SUB; can compare immed to mem
  We will discuss in more detail with conditional jumps

INC and DEC

• How many times have you written something like this:
  for (i = 0; i < 100; i++) {...
• We increment (add 1) or decrement (subtract 1) so often that special instructions are provided for these operations
• INC and DEC are one-operand instructions
• Syntax:
  INC dest
  DEC dest
• Semantics:
  dest <- dest + 1
  dest <- dest - 1
  Flags: O...SZAP. Modified
  .DIT.....C unchanged
• Operands:
  reg mem
• CAUTION: inc eax is NOT ALWAYS a substitute for add eax, 1
• Why?

Register-Indirect Addressing

• A register can be used to access memory
  compare
  MOV eax, ebx ; copy ebx into eax
  with
  MOV eax, [ebx]; load eax with dword
  whose address is in ebx
• The use of a register to hold an address is called indirect addressing
  eax, ebx, ecx, edx, esi, edi, esp and ebp can all be used for indirect addressing
• We will cover this in more detail later

CF

• The answer is that inc eax leaves CF unmodified
• If eax = 0FFFFFFFFh, then CF will be set by add eax, 1
• This seems a rather strange and arbitrary warp in the semantics of the instruction
• There is however a good reason: INC and DEC are often used for address arithmetic (incrementing and decrementing index registers)
  Multi-word arithmetic is usually done in a loop with index registers pointing to operands
  CF is especially significant in multi-word arithmetic
  We’ll see an example later

Multiplication

• Multiplication is available in 8, 16 and 32 bits
• 8-bit multiplication multiplies 2 8-bit operands and produces a 16 bit result. One operand is in AL, result is in AX
• 16-bit multiplication multiplies 2 16-bit operands and produces a 32 bit result. One operand is in AX, result is in DX:AX
• 32-bit multiplication multiplies 2 32-bit operands and produces a 64 bit result. One operand is in eax, result is in eax:edx
• Even if the product of 2 k-bit operands is only k bits long, the upper k bits of the 2k-bit result are cleared (MUL) or set to the top-order bit of the product (IMUL)
Multiplication & Division

• x86 multiplication and division instructions are much more restricted than addition or subtraction

• Both require the use of the accumulator register (al, ax, or eax)

• Unlike addition and subtraction, multiplication and division have separate instructions for unsigned and signed operands

  • Unsigned: MUL, DIV
  • Signed: IMUL, IDIV

MUL (unsigned MULtiply)

• Syntax: MUL source

• Semantics:
  AX <-- AL * source8
  DX:AX <-- AX * source16
  edx:eax <-- eax * source32

  Flags: O.......C modified
         .SEAE. undefined
         .DIT...... unchanged

• Operands:
  reg, mem

  NOTE: No Immediate

  Overflow and Carry flags are identical after a multiplication
  0 = entire product is available in low-order part (AL, AX or eax)
  1 = product occupies both parts (AX, DX:AX or eax:edx)

IMUL (Integer MULtiply)

• Syntax: IMUL source

• Semantics:
  AX <-- AL * source8
  DX:AX <-- AX * source16
  edx:eax <-- eax * source32

  Flags: O.......C modified
         .SEAE. undefined
         .DIT...... unchanged

• Operands:
  reg, mem

  NOTE: No Immediate!

  Overflow and Carry flags are identical after a multiplication
  0 = entire product is available in low-order part (AL, AX or eax)
  1 = product occupies both parts (AX, DX:AX or eax:edx)

3-Address IMUL

• In the 32-bit instruction set, IMUL is extended to a 3 operand instruction and an (apparent) immediate 2 operand instruction

• imul dest, source1, source2

  IMUL reg, imm
  IMUL cx, 100 cx= cx*100
  IMUL reg, r/m,imm IMUL ecx,edx,3 ecx= ecx* edx
  IMUL reg, reg IMUL ecx, dx

  Note: 2 operand IMUL cx, 100 == IMUL cx, cx, 100

• Operands:
  reg16, imm
  reg32, imm
  reg16, reg/mem16, imm
  reg32, reg/mem32 imm
  reg16, reg16
  reg32, reg32

Limitations of IMUL Immediate

• Note that the MUL instruction was never extended in this fashion

• When using IMUL in this form the upper half of the product is discarded; result is same as unsigned.

• It can only be used for "small" products that do not exceed n bits for n-bit source operands

Overflow / Carry with MUL and IMUL

• Overflow and carry indicate if LOW-ORDER part of product is valid alone

• Each sequence below starts with MOV AL,42D

  1) MOV BL,3
     IMUL BL
     O = 0 C = 0; AX = 126 (AL = 126)

  2) MOV BL,5
     IMUL BL
     O = 1 C = 1; AX = 210 (AL = -46)

  3) MOV BL,5
     IMUL BL
     O = 0 C = 0; AX = 210 (AL = -46)

  4) MOV BL,7
     IMUL BL
     O = 1 C = 1; AX = 294 (AL = -38)
Division: DIV and IDIV

- DIV for (unsigned) numbers and IDIV (signed)
- 8-bit division requires a 16-bit dividend in AX and an 8-bit divisor in an 8-bit location.
  - If the quotient requires more than 8 bits, then a divide overflow error is generated.
  - This causes an INT 0 exception; the default INT 0 handler terminates the program, displays a "Divide Overflow" message and returns to the OS.
- In 8-bit division the quotient goes into AL and the remainder into AH.
- 16-bit division requires a 32-bit dividend in DX:AX and a 16-bit divisor in a 16-bit location.
- 32-bit division requires a 64-bit dividend in edx:eax and a 32-bit divisor in a 32-bit location.

NOT

- Used to logically negate or complement an operand
- Same effect can be achieved with XOR but if the entire operand has to be complemented then NOT is faster

NEG

- NEG (NEGate) is 2's complement negation
- Like NOT it is a single operand instruction
- Subtracts operand from 0 and replaces it with result

Arithmetic Example: math.asm

```
#include "asm_io.inc"
segment .data
  ; Output strings
  prompt       db    "Enter a number: ", 0
  square_msg   db    "Square of input is ", 0
  cube_msg     db    "Cube of input is ", 0
  cube25_msg   db    "Cube of input times 25 is ", 0
  quot_msg     db    "Quotient of cube/100 is ", 0
  rem_msg      db    "Remainder of cube/100 is ", 0
  neg_msg      db    "The negation of the remainder is ", 0
segment .bss
  input       resd 1
segment .text
        global  _asm_main
        _asm_main:
        enter   0,0               ; setup routine
        pusha
```

Flags:
- O...SZAPC undefined
- .DIT.... unchanged

Operands:
- reg mem

Notes:
- 1. NOT is a unary operation
- 2. NOT is logical negation (1's complement); every bit is inverted; NEG is the two's complement negation
- 3. Unlike other boolean operations NOT has no effect on the flags
math.asm:2

mov eax, prompt
call print_string

call read_int
mov [input], eax

imul eax ; edx:eax = eax * eax
mov ebx, eax ; save answer in ebx
mov eax, square_msg
call print_string
mov eax, ebx
call print_int
call print_nl

math.asm:3

mov eax, [input]
imul ebx, [input] ; ebx *= [input]
mov eax, cube_msg
call print_string
mov eax, ebx
call print_int
call print_nl

imul ecx, ebx, 25 ; ecx = ebx*25
mov eax, cube25_msg
call print_string
mov eax, ecx
call print_int
call print_nl

math.asm:4

mov eax, ebx
; initialize edx by sign extension
cdq
mov ecx, 100 ; can’t divide by immediate value
idiv ecx ; edx:eax / ecx
mov eax, edx ; save quotient into ecx
mov eax, quot_msg
call print_string
mov eax, ecx
call print_int
call print_nl

mov eax, rem_msg
call print_string
mov eax, edx
call print_int
call print_nl

math.asm:5

neg edx ; negate the remainder
mov eax, neg_msg
call print_string
mov eax, edx
call print_int
call print_nl

pops
mov eax, 0 ; return back to C
leave
ret

Transfer of Control Instructions

- The basic transfer of control instructions are:
  - Unconditional Jumps (JMP)
  - Conditional Jumps (many)
  - Call and Return (CALL and RET)
  - Interrupt and Interrupt Return (INT and IRET)

- Jumps are like GOTO statements in a high level language
- High level selection and iteration structures (if, while, for, etc.)
  are implemented at the machine level with jumps or branches -
  the term varies by processor
- Conditional jumps operate by inspecting one or more of the
  flags
- Intel processors are CISC machines and provide other control
  structures also - mainly LOOP instructions and repeated string
  instructions
- These are complex instructions and we will study them later
Short, Near, Far

- Before we examine jumps we need to look at distances in memory
- The segmented architecture of 8086 allows two types of addresses:
  - NEAR specified by 16 bit offset relative to CS
  - FAR specified by 32 bit segment:offset; anywhere within 1MB physical memory
- For 32 machines we still have NEAR or FAR
  - NEAR specified by 32 bit offset relative to CS
  - FAR specified by 48 bit segment:offset; anywhere within 4GB physical memory
- Machine encoding of jumps adds another type of address:
  - SHORT specified by 8-bit signed address; range is -128 to +127 bytes
- On 386+ processors, conditional jumps (Jxx) can be:
  - SHORT +127 to -128 bytes
  - NEAR +32,767 to -32,768 (if not flat model)
  - FAR +2G to -2G (flat model)
- Note that SHORT jumps are require only one byte for the jump offset

The JMP Instruction

- Syntax:
  - JMP dest
    - JMP NEAR dest
    - JMP FAR dest
  - JMP SHORT dest
- Semantics
  - EIP <- dest
  - Flags: ODITSZAPC unchanged
- Operands:
  - index reg mem label
- Note: under operands we have added two new types: index register and label
  - A "label" is a symbolic name for an address

JMP Targets

- The target of a JMP instruction can be an address stored in memory or an index register
  - This allows run-time computation of addresses
  - This and the similar operation of the CALL instruction provide a hardware foundation for late-bound function calls in dynamically-typed and/or object-oriented languages
- For this course we will only use the simplest form: JMP dest
- In assembly we can write SHORT JMP, NEAR JMP or FAR JMP, but most assemblers accept just JMP and figure out from context which type of jump is appropriate
  - SHORT JMP is used as an optimization: reduces the length of the instruction to 2 bytes

Conditional Jumps and Sequences

- Conditional jumps (abbreviated Jxx or Jcc) inspect the flags
  - If designated condition is met, jump is taken otherwise control falls through to next instruction
  - A typical conditional sequence looks like:
    - CMP OP1, OP2
    - JE label
  - JE jumps if ZF = 1 (result is zero) and JNE jumps if ZF = 0 (result non-zero)
  - Recall that the compare (CMP) instruction is exactly like SUB but modifies flags only
  - Don't use unnecessary instructions:
    - sub eax, ebx
    - cmp eax, 0 ; not necessary- SUB already
      - jle negative ; set the flags
    - All you need is:
      - sub eax, ebx
      - jle negative

Grouping of x86 Conditionals

- The x86 conditionals are a large and at first confusing set of mnemonics
- The key to understanding them is to regard them in three groups:
  1. Unsigned conditionals. These correspond to standard relational operators for arithmetic
  2. Signed conditionals. These also correspond to standard relational operators for arithmetic
  3. Single-flag tests (including the test for equality)
    - 4. And the oddball JCXZ (Jump if CX = 0)
- Remember that the processor neither knows nor cares if you intend the operands of an ADD instruction to be signed or unsigned
- The distinction is made when testing the results with conditional jumps

Synonyms

- Many conditionals have two mnemonics (and some have three) that correspond to a single machine language instruction
- You can use whichever fits the sense of the program to make it more readable
- Examples: JZ (Jump if Zero) and JE (Jump if Equal)
  - JNZ (Jump if Non-Zero) and JNE (Jump if not Equal)
- Compare:
  - SUB AK, BK
  - JE label ; jump if zero to label
  - With
    - CMP AK, BK
    - JE label ; jump if equal to label
  - "Jump if equal" would sound odd following a subtraction. The natural question is "equal to what?"
  - Likewise, "Jump if zero" would sound odd after a compare instruction
Overview of Conditional Mnemonics

• Unsigned arithmetic: JB, JNAE, JBE, JNA, JA, JNBE, JAE, JNB
• Signed Arithmetic: JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL
• Single Flag: Zero: JE, JL, JNE, JNZ
Overflow: JO, JNO
Sign: JC, JNC
Parity: JP, JPE, JNP, JPO
• Test for CX = 0: JCXZ, jecxz

Syntax & Semantics of Conditional Jumps

• All of the conditionals are the same
• Syntax:
  Jcc dest
• Semantics:
  EIP < dest if condition is true
  Flags: ODTISZAPC unchanged
• operands:
  short or near label
• The fact that the flags are unaffected means that the flags can be tested repeatedly if necessary. Ex:
  cmp ax, amt
  jbe isless ; jump if ax < amt (unsigned)
  jae ismore ; jump if ax > amt (unsigned)
  jmp equal

The Unsigned Conditionals

• JB, JNAE, JBE, JNA, JA, JNBE, JAE, JNB
  JB (Jump if Below)
  JNAE (Jump if not Above or Equal)
  JC (Jump if Carry set)
  Semantics: IP < dest if CF = 1
  JBE (Jump if Below or Equal)
  JNA (Jump if not Above)
  Semantics: IP < dest if CF = 1 orZF = 1
  JA (Jump if Above)
  JNBE (Jump if not Below or Equal)
  Semantics: IP < dest if CF = 0 andZF = 0
  JAE (Jump if Above or Equal)
  JNB (Jump if not Below)
  Semantics: IP < dest if CF = 0
  JNC (Jump if No Carry)
  Semantics: IP < dest if CF = 0
• Note that semantics are precisely expressed in terms of flags but English mnemonics make a lot more sense

The Signed Conditionals

• JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL
• Signed: JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL
• The key to distinguishing (and remembering) signed and unsigned conditional jumps is
  UNSIGNED: B (below) and A (above)
  SIGNED: L (less than) and G (greater than)
• Unsigned conditionals are used with unsigned data; for example when comparing addresses or characters
• Signed conditionals are only used with 2's complement data.
• Signed are by far the most common

CMP Revisited

• Remember that CMP performs a subtraction and sets the flags
• Consider what happens with CMP AL,2

<table>
<thead>
<tr>
<th>AL</th>
<th>Signed</th>
<th>Unsigned</th>
<th>O S Z C Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 00-01</td>
<td>0 - 1</td>
<td>0 - 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>2 12</td>
<td>2</td>
<td>12</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3 03-7F</td>
<td>3 - 127</td>
<td>3 - 127</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>4 00-81</td>
<td>&lt; -128</td>
<td>128 - 129</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>5 80-FF</td>
<td>&gt; -129</td>
<td>&gt; -129</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

• If AL < 2 in a signed interpretation look at rows 1, 4, and 5. Here SF <> OF, otherwise if AL = 2 SF=OF
• If AL < 2 in an unsigned interpretation look at row 1. Here CF is set, otherwise it is clear
• Semantics can be precisely expressed in terms of flags but we will see that English mnemonics make a lot more sense

Signed and Unsigned Conditionals

• Unsigned: JB, JNAE, JBE, JNA, JA, JNBE, JAE, JNB
• Signed: JL, JNGE, JLE, JNG, JG, JNLE, JGE, JNL

• The key to distinguishing (and remembering) signed and unsigned conditional jumps is
  UNSIGNED: B (below) and A (above)
  SIGNED: L (less than) and G (greater than)
• Unsigned conditionals are used with unsigned data; for example when comparing addresses or characters
• Signed conditionals are only used with 2’s complement data.
• Signed are by far the most common
Zero Flag / Equality

• These are the most common conditionals used

JE (Jump if Equal)
JZ (Jump if Zero)
Semantics: IP ← dest if ZF = 1

JNE (Jump if Not Equal)
JNZ (Jump if not Zero)
Semantics: IP ← dest if ZF = 0

CF and OF

Carry: JC (Jump if Carry)
JNC (Jump if no Carry)
Semantics:
JC: IP ← dest if CF = 1
JNC: IP ← dest if CF = 0

Overflow: JO (Jump if Overflow)
JNO (Jump if no Overflow)
Semantics:
JO: IP ← dest if OF = 1
JNO: IP ← dest if OF = 0

SF and PF

Sign: JS (Jump if Sign)
JNS (Jump if no Sign)
Semantics:
JS: IP ← dest if SF = 1
JNS: IP ← dest if SF = 0
Remember that SF = 1 if result is negative

• Parity JP (Jump if Parity)
JPE (Jump if Parity Even)
Semantics: IP ← dest if PF = 1
JNP (Jump if Parity)
JPO (Jump if Parity Odd)
Semantics: IP ← dest if PF = 0
Remember that PF = 1 on even parity

And the Oddball Jump: JECXZ

• This jump instruction inspects the CX register, not the flags

• JECX (Jump if Cx = 0)
JECXZ (Jump if ecx = 0)
Semantics:
wi ← dest if ECX = 0
Flags: ODITSZAPC unchanged

• This is mostly of interest as a test before LOOP instructions but can be used to inspect CX (or ECX) without setting flags.
• Note that no “inverse” instruction (JECXNZ) exists

Conditional Jump Examples

• Finally some complete functions to look at!
Check for alphabetic characters
Find min-max in array of integers

isalpha:
; accepts a char in AL and returns ZF set if A..Z or a..z
; returns ZF clear otherwise
push eax ; save char in al
and al, 11011111b ; convert al to uppercase
cmp al, 'A'
jb li ; if below, ZF clear
cmp al, 'Z'
ja li ; if above, ZF clear
sub eax, eax ; sets zf
li:
pop eax ; no effect on flags
ret

• Example call
mov al, char
call isAlpha
jnz notAlpha

isAlpha:
; accepts a char in AL and returns ZF set if A..Z or a..z
; returns ZF clear otherwise
push eax ; save char in al
and al, 11011111b ; convert al to uppercase
cmp al, 'A'
jb li ; if below, ZF clear
cmp al, 'Z'
ja li ; if above, ZF clear
sub eax, eax ; sets zf
li:
pop eax ; no effect on flags
ret

• Example call
mov al, char
call isAlpha
jnz notAlpha

18
Notes on isalpha function

- This function returns a boolean in the zero flag
  ZF = 1 function => True, ZF = 0 => false
- Note the AND Mask so that we have to check upper-case range only
- eax is pushed so that we can restore AL
- Note use of unsigned conditional jumps
- If either cmp is true, we know that zf is clear
- If neither is true, sub ax, ax will set zf

Min-Max Values in Array

```
ARRAYCOUNT equ 500
array resd ARRAYCOUNT
largest resd 1
smallest resd 1

MinMax:
    mov edi, array            ; base address of array
    mov eax, [edi]            ; get first element
    mov [largest], eax        ; initialize largest
    mov [smallest], eax       ; initialize smallest
    mov eax, ARRAYCOUNT       ; number of elements in array
    L1:
        mov eax, [edi]            ; get an element
        cmp eax, [smallest]       ; is eax < smallest?
          jge L2                    ; no, skip
        mov [smallest], eax       ; yes, save it
        L2:
          cmp eax, [largest]         ; is eax > largest?
          jle L3                     ; no, skip
          mov [largest], eax       ; yes, save it
        L3:
          add edi, 4               ; advance pointer
          loop L1
```

Notes on Min-Max

- Algorithm: first element in array is both largest so far, and smallest so far. Use this to initialize variables largest and smallest
- Note that we cannot use this code:
  ```
  mov [largest], [edi]         ; initialize largest
  ```
  because that is a memory-to-memory MOV
- Note use of signed comparison instructions. Code has to be modified for unsigned integers
- Because array was defined with dd, elements are 32 bits so we have to add 4 to edi to advance pointer

The LOOP Instructions

- LOOP is a hardware looping instruction
  ```
  mov ecx, 5
  loop SHORT_LABEL:
      ...
  LOOP
  ```
- LOOP decrements ecx and jumps to SHORT_LABEL if ecx is non-zero
- Syntax:
  Loop dest
- Semantics:
  (1) ecx <-- ecx - 1 (flags unchanged)
  (2) if ecx != 0 then eip <-- dest
- Flags: ODITSZAPC unchanged
- Operands:
  Short label only (-128 to +127 bytes)

Which explains the oddball...

- Testing before the loop: the jecxz instruction tests for ecx = 0 before the loop is executed
- Commonly, when the loop variable initial value is unknown, you would see the above sequence as
  ```
  Initialize ecx
  ...
  jecxz short_label:
  ...
  LOOP short_label
  quit:
  ```

LOOP

- Note that ecx is decremented before being tested
- How many times will the loop be performed above, assuming that ecx is not altered by the code?
- What happens if `mov ecx, 5` is replaced by `mov ecx, 0`?
  How many times is the loop performed?

  \[
  0 - 1 = \text{FFFFF} = 4,294,967,295
  \]
  ecx is interpreted as an unsigned value by the loop instruction
Sum an Array of Integers

```assembly
size EQU 100
array TIMES size dd 0
...
sub eax, eax ; initialize sum
mov ebx, array ; pointer to array
mov ecx, size ; number of elements
L1:
  add eax, [ebx]
add ebx, 4
loop L1
```

Alternate code with displacement addressing

```assembly
size EQU 100
array TIMES size dd 0
...
sub eax, eax ; initialize sum
sub ebx, ebx ; initial offset = 0
mov ecx, size ; number of elements
L1:
  add eax, [ebx+array]
add ebx, 4
loop L1
```

Checking for overflow

```assembly
• Code varies with signed/unsigned ints:
  size EQU 100
  array TIMES size dd 0
  ...
  sub eax, eax ; initialize sum
  sub ebx, ebx ; initial offset = 0
  mov ecx, size ; number of elements
  L1:
    add eax, [ebx+array]
  jo overflow ; signed OR
  jc overflow ; unsigned
  add ebx, 4
  loop L1
• Note that conditional jump has to be immediately after
  the add into eax, because add bx, Z affects flags
```

64-bit sum of 32-bit array

```assembly
• Note that ADC DX, 0 seems to add 0 to DX (a bit silly)
  but it is Add with Carry so CF is added as well
• This only works for unsigned integers
  size EQU 100
  array TIMES size dd 0
  ...
  sub eax, eax ; initialize l.o. sum
  sub edx, edx ; initialize h.o. sum
  sub ebx, ebx ; initial offset = 0
  mov ecx, size ; number of elements
  L1:
    add eax, [ebx+array]
    adc edx, 0
    lea ebx, [ebx+4]
   _loop L1
```

Signed 64-bit sum of 32-bit array

```assembly
• In order to add an array of signed 32-bit integers in
  an 32-bit array, we need to use the following
  algorithm:
    Initialize sum to 0
    Initialize pointer to array
  L1:
    load from [ebx]
    sign-extend value to 64 bits
    add into 64-bit sum
    check OF
  Loop L1
• We’ll get back to this later
```

Conditional LOOPS

```assembly
• Loops come in several flavors
  Conditional loops are LOOPE (LOOPZ) and LOOPNE (LOOPNZ)
• These loops have two tests:
  test for ecx = 0 AND for Z = 1 or Z = 0
  LOOPE (LOOP if Equal) and LOOPZ (LOOP if Zero)
• Syntax:
  LOOPE dest
  LOOPZ dest
• Semantics:
  (1) ecx -- ecx - 1 (flags unchanged)
  (2) if ecx != 0 AND ZF = 1 then eip -- dest
  Flags: OHSIAPC unchanged
• Operands:
  Short label only (-128 to +127 bytes)
```
LOOPE/LOOPZ

- Notes:
  1. Because the test for loop continuation is an AND, loop will terminate when either ZF = 0 OR ecx = 0.
  2. This loop is typically used when "searching for a mismatch"
  3. Just as JZ/JE are synonyms, LOOPE and LOOPZ are the same machine instruction
  4. It may be more convenient to read the mnemonics as "loop while equal" and "loop while zero" because they are more similar to a WHILE loop than a count-controlled FOR loop

Example: Find First Non-Zero Element

```
hits TIMES 256 dd 0
....
mov ecx, 256 ; number of values
mov ebx, hits-4 ; one element before first val
L1:
  add ebx,4
  cmp dword [ebx], 0 ; compare immediate to mem
  loopz L1
L2:
  How do we know at L2 why the loop terminated?
  We may have found a non-zero value, or we may have failed to find one after inspecting all 256 elements
```

After Conditional Loop Termination

```
hits TIMES 256 dd 0
....
mov ecx, 256 ; number of values
mov ebx, hits-4 ; one element before first val
L1:
  add ebx,4
  cmp dword [ebx], 0 ; compare immediate to mem
  loope L1
L2:
  The obvious answer is to examine the word that ebx points to with cmp dword [ebx], 0.
  But an easier solution lies in the semantics of loope:
  Flags: ODITSZAPC unchanged
  So the flags remain set by the cmp instruction, and at L2 we can have:
  L2: jz not_found
```

LOOPNE and LOOPNZ

- LOOPNE (LOOP if/while Not Equal) and LOOPNZ (LOOP if/while Not Zero)
- Syntax:
  LOOPNE dest
  LOOPNZ dest
- Semantics:
  (1) ecx <= ecx - 1 (flags unchanged)
  (2) if ecx != 0 AND ZF = 0 then eip <= dest
  Flags: ODITSZAPC unchanged
- Operands:
  Short label only (~128 to +127 bytes)
- Notes:
  1. Since test is an AND, loop will terminate when either ZF = 1 OR ecx = 0
  2. This loop is typically used when "searching for a match"

Example: Find First Space

```
LOOPNE Example: Find First Space
```

Be careful about order of instructions

```
• This doesn’t work
```

```
LOOPNE Example: Find First Space
```

```
• Find the first space in a string. Handy for parsing strings such as command-line arguments:
  aString resb strsize
  ....
  FindBlank:
    mov esi, aString - 1
    mov ecx, strsize
    mov al, 20h ; 20h = ASCII space
    L1:
      inc esi
      cmp [esi], al
      loopne L1
    ; loop terminates because we either found a space
    ; OR searched the entire string. Inspect the flags:
    jz FoundBlank
    jmp NoBlanks
```

Notes:

- 1. Because the test for loop continuation is an AND, loop will terminate when either ZF = 0 OR ecx = 0.
- 2. This loop is typically used when "searching for a mismatch"
- Just as JZ/JE are synonyms, LOOPE and LOOPZ are the same machine instruction
- It may be more convenient to read the mnemonics as "loop while equal" and "loop while zero" because they are more similar to a WHILE loop than a count-controlled FOR loop
Example: First Negative Integer
- This example finds the first negative int in an array, using a TEST instruction (AND operands and modify flags) to inspect top bit

```assembly
nums read vectorsize

FindNeg:
    mov esi, nums - 4
    mov cx, vectorsize
L1:
    add esi, 4
    test byte [esi+3], 80h; check top bit
    loops L1
    jnz FoundNeg
    jmp NoNeg
```

The TEST Instruction
- TEST is AND without storing results (affects flags only) For now consider:

```text
80FF = 1011 0000 1111 1111
AND 8000 = 1000 0000 0000 0000
Result: 8000 = 1000 0000 0000 0000
```

```text
70FF = 0111 0000 1111 1111
AND 8000 = 1000 0000 0000 0000
Result: 0000 = 0000 0000 0000 0000
```

LOOP on the Pentium
- LOOP and its variations are expensive instructions on a Pentium - typically 6 clocks.

- Optimization of asm code can involve replacement of LOOPS:

```assembly
aString resb strsize

FindBlank:
    mov esi, aString - 1
    mov ecx, strsize
    mov al, 20h; 20h = ASCII space
L1:
    inc esi; 1 clock
    cmp [esi], al; 1 clock
    loopne L1; 6 clocks
    jz FoundBlank
    jmp NoBlank
```

Replacing LOOP
- LOOP can be replaced with conditional jumps

```assembly
FindBlank2:
    mov esi, aString - 1
    mov ecx, strsize
    mov al, 20h; 20h = ASCII space
L1:
    inc esi; 1 clock
    cmp [esi], al; 1 clock
    jne testCX; not a space, check cx(1 clock)
jmp FoundBlank; was a space, process it
```

TestCX:
- any string left?
    dec ecx; adjust counter (1 clock)
    jnz L1; 1 clock
    jmp NoBlanks

- We had an 8-clock loop before; now it is 5 clocks

Arithmetic Example: Prime Numbers
```
#include <stdio.h>

int main()
{
    unsigned guess; // current guess for prime /
    unsigned factor; // possible factor of guess /
    unsigned limit; // find primes up to this value /

    printf("Find primes up to: ");
    scanf("%u", &limit);
    printf("\n"); /* treat first two primes as special case */
    printf("2\n");
    printf("3\n");
guess = 5; // initial guess /
while ( guess <= limit ) { /
    factor = 3;
    while ( factor*factor < guess && guess % factor != 0 )
        factor += 2;
    if ( guess % factor != 0 )
        printf("%d\n", guess);
    guess += 2; // only look at odd numbers */
}
return 0;
```
prime.asm:2
mov eax, Message
call print_string
mov [limit], eax
mov eax, 2     ; printf("2
");
call print_int
call print_nl
mov eax, 3     ; printf("3
");
call print_int
call print_nl
mov dword [Guess], 5     ; Guess = 5;

prime.asm:3

while_limit:                         ; while ( Guess <= Limit )
    mov eax, [Guess]
    cmp eax, [limit]
    jnb end_while_limit ; use jnb instead of jnz if data is unsigned
silver_factor:
    mov eax, [guess]     ; if answer won't fit in eax
    cmp eax, [Guess]
    jze end_while_factor ; if !(factor*factor < guess)
    mov edx, 0
    div eax
    cmp edx, 0
    je end_while_factor ; if !(guess % factor != 0)
    add edx, 2
    jmp while_factor
end_while_factor:

end_if:
add eax, 2
mov [Guess], eax ; guess += 2
jmp while_limit
endwhile_limit:

Translating Standard Control Structures
* IF Statement
if ( condition )
    then block ;
else
    else block ;
endif

* IF Statement without ELSE
; code to set FLAGS
; select Jxx to jump if condition false
jxx endif
; code for then block
endif

Translating Standard Control Structures
* WHILE loop (test at top)
while( condition ) {
    body of loop;
}

* WHILE loop (test at top) with back to C
while ( condition )
    body of loop;
    jmp while
endwhile:
While and Do Loops

- **DO loop** (test at bottom)
  ```
  do {
    body of loop;
  } while( condition )
  ```

- **Assembler**
  ```
  do:
    ; body of loop
    ; code to set FLAGS based on condition
    ; select Jxx to jump if condition is TRUE
    jxx do
  ```

Quiz Mar 22

- Which of the following instructions do **not** affect the flags. (Circle your answers):
  - MUL
  - JMP
  - MOV
  - DEC
  - PUSH
  - JNAE

- Which instruction is the same as JNAE?
  - JNLE
  - JLE
  - JB
  - JBE

- In a NASM program, what goes into:
  ```
  segment .text
  segment .data
  segment .bss
  ```