Introduction to Assembly
A User’s View of Computer Systems

Level 5
Application program level
(Spreadsheet, Word Processor)

Level 4
High-level language level
(C, Java)

Level 3
Assembly language level

Level 2
Machine language level

Level 1
Operating system calls

Level 0
Hardware level

Increased level of abstraction

System dependent

Courtesy: Guide to Assembly Language Programming in Linux, Sivarama P. Dandamudi, 2005
What is Assembly Language?

• Low-level programming language
• Influenced by:
  • The architecture of the processor
  • The instruction set

• Two basic types of processors
  • CISC (Complex Instruction Set Computers)
  • RISC (Reduced Instruction Set Computers)

• Pentium is an example of a CISC processor
• Assembler translates assembly to machine code
• NASM is a popular assembler for x86 architecture
Advantage of High-Level Languages

• Program development is faster
• Programs are easier to maintain
• Programs are portable

so, why to program in the Assembly language?
Why to program in Assembly?

- Efficiency
  - Time efficiency
  - Space efficiency
- Direct hardware control
When is it Useful?

• Practical purposes (embedded systems)
• Cracking

Personal Satisfaction
IA – 32 Architecture (IA - Intel Architecture, i386)

- 32 bit Registers
- Logical and Arithmetic instructions (instruction set)
- Fetch-decode-execute cycle
- Addressing Modes
  - Registers
  - Immediate
  - Direct
  - Indirect
Fetch-decode-execute

Fetch

Execute

Decode
Registers

- Ten 32-bit and six 16-bit registers:
  - General registers
  - Control registers
  - Segment registers (not in use in Linux)
General registers

- Data registers
- Pointer registers
- Index registers
General Registers - Data

- Data registers
  - Four 32-bit registers (EAX, EBX, ECX, EDX)
  - Four 16-bit registers (AX, BX, CX, DX)
  - Eight 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL)

- Data registers can be used in arithmetic and logical instructions

- Special functions for specific instructions
  - EAX – Accumulator (mul)
  - ECX – Counter (loop)
General Registers – Pointer and Index

- String Registers (could be used as general-purpose)
  - ESI
  - EDI

- Pointer Registers (could be used as general-purpose)
  - Mainly used to maintain the stack
    - ESP
    - EBP
Control Registers

• Instruction Pointer (EIP)
  • Tracks the next instr.
  • Updated once an instr. is executed, jump, etc.

• Flag register
  • Affected by logical and arithmetic inst.
  • Affects conditional jump operations
Flags

- Each flag bit is independent from the rest of the bits.
  - **OF**— The *Overflow flag* is set when the result of an operation becomes too large to fit in the operand it originally occupied.
  - **SF**— The *Sign flag* is set when the result of an operation forces the operand to become negative.
  - **ZF**— The *Zero flag* is set when the results of an operation become zero
  - **CF**— The *Carry Flag* is set when an arithmetic or shift operation "carries out" a bit from the operand
Some instructions and the flags they affect

MOV  NOT  JMP*
does not affect flags

NEG
The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

AND  OR
The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined

DEC  INC
The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result

ADD  ADC  SUB  SBB
The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

CMP
Modify status flags in the same manner as the SUB instruction

The full set of instructions can be found here.
Overview of Assembly Language

• Statements can have one of the following classes:
  • Executable
  • Directive/pseudo-instructions - constants and more
• All three classes use the same format
  • [label] instruction [operands] ;[comment]
• Fields in [ ] are optional
• Examples

<table>
<thead>
<tr>
<th>repeat:</th>
<th>inc</th>
<th>result</th>
<th>;executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR:</td>
<td>EQU</td>
<td>0DH</td>
<td>; directive</td>
</tr>
</tbody>
</table>
Data Initialization

• Storage allocation statement

variable-name define-directive initial-value [,initial-value], …

• Define directive takes one of these basic forms

  • DB Define Byte ; allocates 1 byte
  • DW Define Word ; allocates 2 bytes
  • DD Define Doubleword ; allocates 4 bytes
  • DQ Define Quadword ; allocates 8 bytes
  • DT Define Ten Bytes ; allocates 10 bytes
Examples

- sorted DB ‘y’
- sorted DB 79H
- sorted DB 1111001B

- Allocate two bytes of contiguous storage and initialize it to 25159
  - value DW 25159
  - automatically converted to its 16-bit hex. equivalent (6247H)
  - address: x x+1
    - 47 62

Pentium uses little-endian: stores the least significant byte in the smallest address.
More Examples:

- A string
- `message` DB ‘H’
  DB ‘E’
  DB ‘L’
  DB ‘L’
  DB ‘O’
- more compactly:
  
  message DB ‘HELLO’

- Array with 8 elements
- `marks` DW 0
  DW 0
  DW 0
  DW 0
  DW 0

  can be abbreviated:
  
  marks TIMES 8 DW 0
Uninitialized Data

• Reserve space for uninitialized data/variables

```
variable-name  reserve-directive  number_of_elements
```

• Reserve directives
  • RESB  Reserve a byte
  • RESW  Reserve a Word
  • RESD  Reserve a Doubleword
  • RESQ  Reserve a Quadword
  • REST  Reserve Ten bytes

<table>
<thead>
<tr>
<th>response</th>
<th>RESB</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>RESW</td>
<td>100</td>
</tr>
<tr>
<td>total</td>
<td>RESD</td>
<td>1</td>
</tr>
</tbody>
</table>
Addressing Modes

• Assembly language instructions require operands
• Operands can be (addressing mode):
  • in a register internal to the processor
  • in the instruction itself
  • in the main memory (usually in the data segment)
Register Addressing Mode

- Using processor’s internal registers
- Very efficient.
- Copies the content of the source register into the dest. register
- Works with 32-bit, 16-bit and 8-bit registers

mov EAX, EBX
mov BX, CX
mov AL, CL
Immediate Addressing Mode

• Data is part of the instruction
• Data is located in the **code segment** not in the data segment
• The immediate data is **always a constant**

```
mov AL, 75
```
Direct Addressing Mode

<table>
<thead>
<tr>
<th>response</th>
<th>DB ‘Y’ ;allocates a byte, initializes to Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>table1</td>
<td>TIMES 20 DD 0 ;allocates 80 bytes, initializes to 0</td>
</tr>
<tr>
<td>name1</td>
<td>DB ‘Jim Ray’ ; 7 bytes</td>
</tr>
</tbody>
</table>

mov AL, [response] ; copies Y into AL register
mov [response], ‘N’ ; N is written into response
mov [name1], ‘K’ ; writes K as the first character of name1
move [table1], 56 ; 56 is written in the first element – table1[0] = 56

- mov EBX, table1 VS. mov EBX, [table1]

memory-to-memory transfer is not supported for instructions that require two operands!
Indirect Addressing Mode

• How to access second element of table1?
• This mode is required for variable with several elements.

```
mov EBX, table1
mov [EBX], 100 ; table[0] = 100
add EBX, 4 ; EBX = EBX + 4
mov [EBX], 99; table[1] = 99
lea EBX, [table1 + ESI] ; computes address at runtime, lea returns the address where as mov returns the content
```
Ambiguous Moves

• Moving immediate value into memory sometimes causes ambiguity

\[
\begin{align*}
\text{mov} & \ \text{EBX}, \ \text{table1} \ \ ; \ \text{table1 TIMES 20 DD 0} \\
\text{mov} & \ \text{ESI}, \ \text{name1} \ ; \ \text{name1 DB ‘Jim Ray’} \\
\text{mov} & \ [\text{EBX}], \ 100 \\
\text{mov} & \ [\text{ESI}], \ 100
\end{align*}
\]

- Word equivalent of 100? maybe a byte equivalent of 100?
- Clarify this by using a type specifier

\[
\begin{align*}
\text{mov} & \ \text{DWORD} \ [\text{EBX}], \ 100 \ \leftrightarrow \ \text{mov} \ [\text{EBX}], \ \text{DWORD 100} \\
\text{mov} & \ \text{BYTE} \ [\text{EBX}], \ 100 \ \leftrightarrow \ \text{mov} \ [\text{EBX}], \ \text{BYTE 100}
\end{align*}
\]
## Type Specifiers

<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Bytes addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>WORD</td>
<td>2</td>
</tr>
<tr>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>QWORD</td>
<td>8</td>
</tr>
<tr>
<td>TBYTE</td>
<td>10</td>
</tr>
</tbody>
</table>
Arithmetic Instructions – INC and DEC

• increment/decrement the operand by 1
• The operand can be either in register or in memory

```
inc destination ; might be 8-, 16-, 32-bit
dec destination ; might be 8-, 16-, 32-bit
```
ADD

- Used to add 8-, 16-, 32-bit operands

```c
add destination, source ; dest = dest + source
```

- `inc EAX` is preferred to add `EAX, 1`
- Less memory, same speed
SUB and CMP

- **SUB** used to subtract 8-, 16-, 32-bit operands

```
sub destination, source ; dest = dest - source
```

- **CMP** compares two operands (equal, not equal, ..)
- CMP behavior is similar to SUB except that the result is not saved
  - So, how can we retrieve the comparison result?
- CMP is typically used with a conditional JUMP inst.

Examples:

- `mov al, 11111100_b`
- `mov bl, 00000010_b`
- `cmp al, bl ; (ZF (zero flag) gets a value 0)`

- `mov al, 11111100_b`
- `mov bl, 11111100_b`
- `cmp al, bl ; (ZF (zero flag) gets a value 1)`
Conditional Jump

- Jump if the specified condition is satisfied

```
  j<cond>  label  ;identifies the condition
```

- The condition being tested is the result of the last arithmetic or logic operation

```
read_char:
  mov  DL,0
  ...
  (code for reading a character into AL)
  ...
  cmp  AL, 0DH ;compares the character to CR (asci:0DH)
  je   CR_received ; if equal, jump to CR_received
  inc  CL ;otherwise, increment CL and
  jmp  read_char ; go back to read another char.
```

```
CR_received:
  mov  DL, AL
```

but, the CMP doesn’t save the result, so what really happens ?!!
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF = 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above or equal</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal</td>
<td>CF = 1 or ZF = 1</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above</td>
<td>CF = 0 or ZF = 0</td>
</tr>
<tr>
<td>JA</td>
<td>Jump if above</td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal</td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if not greater</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JG</td>
<td>Jump if greater</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if not less or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if parity even</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if not parity</td>
<td>PF = 0</td>
</tr>
<tr>
<td>JPO</td>
<td>Jump if parity odd</td>
<td>PF = 0</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX register is 0</td>
<td>CX = 0</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if ECX register is 0</td>
<td>ECX = 0</td>
</tr>
</tbody>
</table>
Unconditional Jump

```asm
mov EAX, 1
inc_again:
    inc EAX
    jmp inc_again
mov EBX, EAX
...
```
Multiplication

- Two multiplication instr.: \textit{mul} (\textit{unsigned}) and \textit{imul} (\textit{signed})

- The source operand can be in a register (8/16/32 bit) or in memory

- Immediate operands are not allowed

- Second operand is one of the following:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>r/m8</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>r/m16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>EAX</td>
<td>r/m32</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>

\textbf{MUL r/m8}

\begin{align*}
\text{mov bl,5} & ; \text{multiplier} \\
\text{mov al,9} & ; \text{multiplicand} \\
\text{mul bl}   & ; \text{ax} \leftarrow 2D_h
\end{align*}
Division

- Two Division instr.: \textit{div} and \textit{idiv}

\texttt{div source}

- The source operand is the divisor

![Division diagram]

16 bit dividend
\[
\begin{array}{c}
AX \\
\hline
8 \text{ bit Divisor}
\end{array}
\]

= \quad \text{Quotient} \quad \text{Remainder}
\[
\begin{array}{c}
\text{AL} \\
\text{AH}
\end{array}
\]
Iteration Instruction

```
mov CL, 50
repeat1:
    <loop body>
    dec CL
    jnz repeat1

...  
...  
```
Logical Instructions

Example:

```
mov al, 11111100_b
mov bl, 00000010_b
or AL, BL ;(AL gets a value 11111110_b)
```

```
...  
and AL, 01H
je    bit_is_zero

<code to be executed when the bit is one>

jmp   skip1

bit_is_zero:    
    <code to be executed when bit is zero>

skip1:          
    <rest of the code>
```
Logical Instructions

• Shift
  • SHL – Shift left (put zeros in the empty bits)
  • SHR – Shift right (put zeros in the empty bits)
    SHL ax, cl
    SHR ax, <number>

• Rotate
  • ROL - Rotate left – (shift left and insert the left most bit to the right most bit)
  • ROR – Rotate right (shift right and insert the right most bit to the left most bit)
Example 1 – Max. among 3 numbers

; Data section begins
section .data
var1 dd 40
var2 dd 20
var3 dd 30

section .text
    global _start

_start:
    mov ecx, [var1]
    cmp ecx, [var2]
    jg check_third_var
    mov ecx, [var2]

check_third_var:
    cmp ecx, [var3]
    jg _exit
    mov ecx, [var3]

_exit:
    mov ebx, ecx
    mov eax, 1
    int 80h
Example 2 – Hello World

section .text
  global _start ; must be declared for linker (ld)
section .data
  msg db 'Hello world!', 0xa ; our dear string
  len equ $ - msg ; length of our dear string

_start: ; tell linker entry point

  mov edx, len ; message length
  mov ecx, msg ; message to write
  mov ebx, 1 ; file descriptor (stdout)
  mov eax, 4 ; system call number (sys_write)
  int 0x80 ; call kernel

  mov eax, 1 ; system call number (sys_exit)
  int 0x80 ; call kernel
NSAM – assembling your code

• To assemble a file, you issue a command of the form

  • > nasm -f <format> <filename> [-o <output>] [-l listing]

• Example:

  • > nasm -f elf myasm.s -o myelf.o

• It would create myelf.o file that has elf format (executable and linkable format).
Gdb-GNU Debugger

- Run Gdb from the console by typing \texttt{gdb executableFileName}
- Adding breaking points by typing: \texttt{break label}
- Start debugging by typing: \texttt{run parameters (argv)}

- \texttt{si} – one step forward
- \texttt{c} – continue to run the code until the next break point.
- \texttt{q} – quit gdb
- \texttt{p $eax} – prints the value in eax
- \texttt{x $esp+4} – prints the address in esp + 4 hexadecimal and the value (dword) that stores in this address. It is possible to use label instead of esp.
  Type \texttt{x} again will print the next dword in memory.