The x86 Architecture
Lecture 15
Intel Manual, Vol. 1, Chapter 3

Robb T. Koether
Hampden-Sydney College
Fri, Mar 6, 2009
Outline

1. Overview of the x86 Architecture
   - Instruction Format
   - Registers
   - Data Types
   - The Run-time Stack

2. Assignment
Each instruction is of the form

```
[<label>:] mnemonic [operand₁][, operand₂][, operand₃]
```

The number of operands is 0, 1, 2, or 3, depending on the mnemonic.

Each operand is either

- An immediate value,
- A register, or
- A memory address.
Source and Destination Operands

- Each operand is either a source operand or an destination operand.
- A source operand, in general, may be
  - An immediate value,
  - A register, or
  - A memory address.
- A destination operand, in general, may be
  - A register, or
  - A memory address.
Source and Destination Operands

- The standard interpretation of
  \[ \text{mnemonic } \text{operand}_1, \text{operand}_2 \]
  is that \( \text{operand}_1 \) is the destination and \( \text{operand}_2 \) is the source.

  \[ \text{operand}_1 \leftarrow \text{operand}_1 \ op \ \text{operand}_2 \]

- The Intel manuals are written according to the standard interpretation.
However, the gnu interpretation of
\[ \text{mnemonic operand}_1,\text{operand}_2 \]
is that \text{operand}_1 \text{ is the source and operand}_2 \text{ is the destination.}

\[ \text{operand}_1 \ \text{op} \ \text{operand}_2 \rightarrow \text{operand}_2 \]

Therefore, we will have to interpret the information in the Intel manuals accordingly.
Instructions

- Not every logical combination of operands is permitted in every instruction.
- See the references
Different instructions require different numbers of operands.

For example,

- `hlt` - 0 operands
- `inc` - 1 operand
- `add` - 2 operands
- `imul` - 1, 2, or 3 operands
The memory addresses are 32 bits, so they can access up to 4 GB of memory.

A global variable or function is referenced by its name, which is a label representing its address.

Local variables are referenced by an offset from the base pointer, which holds the base address of the activation record on the stack.
Basic Registers

- There are
  - Eight 32-bit “general-purpose” registers,
  - One 32-bit EFLAGS register,
  - One 32-bit instruction pointer register (eip), and
  - Other special-purpose registers.
The General-Purpose Registers

- The eight 32-bit general-purpose registers are eax, ebx, ecx, edx, esi, edi, ebp, and esp.
- For calculations, we will use eax, ebx, ecx, and edx.
- Register esp is the stack pointer.
- Register ebp is the base pointer.
- Registers esi and edi are source and destination index registers for array and string operations.
The General-Purpose Registers

- The registers $eax$, $ebx$, $ecx$, and $edx$ may be accessed as 32-bit, 16-bit, or 8-bit registers.
- The other four registers can be accessed as 32-bit or 16-bit.
- For example,
  - Register $eax$ represents a 32-bit quantity.
  - The low-order two bytes of $eax$ may be accessed through the name $ax$.
  - The high-order byte of $ax$ is named $ah$.
  - The low-order byte of $ax$ is named $al$. 
# The General-Purpose 32-Bit Registers

<table>
<thead>
<tr>
<th></th>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>ebp</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Accumulator**
- **Base pointer**
- **Array source**
- **Array destination**
- **Stack pointer**
### The General-Purpose 16-Bit Registers

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ax</td>
</tr>
<tr>
<td>16-15</td>
<td>bx</td>
</tr>
<tr>
<td>16-15</td>
<td>cx</td>
</tr>
<tr>
<td>16-15</td>
<td>dx</td>
</tr>
<tr>
<td>16-15</td>
<td>bp</td>
</tr>
<tr>
<td>16-15</td>
<td>si</td>
</tr>
<tr>
<td>16-15</td>
<td>di</td>
</tr>
<tr>
<td>0</td>
<td>sp</td>
</tr>
</tbody>
</table>

- **ax**: Accumulator
- **bx**: Base pointer
- **cx**: Array source
- **dx**: Array destination
- **bp**: Stack pointer
### The General-Purpose 8-Bit Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>31 0</td>
</tr>
<tr>
<td>ah</td>
<td>16 15</td>
</tr>
<tr>
<td>bh</td>
<td>8 7</td>
</tr>
<tr>
<td>ch</td>
<td></td>
</tr>
<tr>
<td>dh</td>
<td></td>
</tr>
<tr>
<td>bp</td>
<td></td>
</tr>
<tr>
<td>si</td>
<td></td>
</tr>
<tr>
<td>di</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td></td>
</tr>
</tbody>
</table>

- **Accumulator**: al, ah, bh, ch, dh
- **Base pointer**: bp
- **Array source**: si
- **Array destination**: di
- **Stack pointer**: sp
EFLAGS Register

- The various bits of the 32-bit EFLAGS register are set (1) or reset (0) according to the results of certain operations.
- We will be interested in the bits:
  - CF - carry flag
  - PF - parity flag
  - ZF - zero flag
  - SF - sign flag
Finally, there is the \texttt{eip} register, which is the instruction pointer.

Register \texttt{eip} holds the address of the next instruction to be executed.

We should never change the value of \texttt{eip} directly. It will be updated automatically as necessary.
Data Types

- There are 5 integer data types.
  - Byte - 8 bits.
  - Word - 16 bits.
  - Doubleword - 32 bits.
  - Quadword - 64 bits.
  - Double quadword - 128 bits.

- We will use doublewords (for ints) unless we have a specific need for one of the other types.
The run-time stack supports procedure calls and the passing of parameters between procedures.

The stack is located in memory.

The stack grows towards low memory.

- When we push a value, $esp$ is decremented.
- When we pop a value, $esp$ is incremented.
Using the Run-time Stack

- Typically, if an operation produces a result, we will push that result onto the stack.
- The next operation, if it expects a previous result, will pop it off the stack.
- The alternative is to use the registers to pass results, but that is much more complicated since we would have to keep track of which registers were free.
Function Calls and the Base Pointer

- When we make a function call, we use the base pointer \( ebp \) to store the location of the top of the stack \( esp \) before the function call.

\[
\text{esp} \rightarrow \text{ebp}
\]

- Then we push the parameters and local variables of the function onto the stack.

- When we return from the function, we use the base pointer to restore the top of the stack to its previous location.

\[
\text{ebp} \rightarrow \text{esp}
\]
Assignment

Homework

Download the Intel Manual, Vol. 1, and read Chapter 3.