The x86 Architecture

Lecture 24
Intel Manual, Vol. 1, Chapter 3

Robb T. Koether

Hampden-Sydney College

Fri, Mar 20, 2015

Outline

Overview

- See the reference “IA-32 Intel Architecture Software Developer’s Manual Volume 1: Basic Architecture”, Chapter 3.

Outline

Instructions

- Each instruction is of the form

  `[label:]` `mnemonic` `[operand`$_1$`][,` `operand`$_2$`][,` `operand`$_3$`]`

- 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 a 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

- Each operand is either a source operand or a 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.
- But only certain combinations are permitted.

Source and Destination Operands

- The standard interpretation of

  $$\texttt{mnemonic operand}_1\texttt{,operand}_2$$

  is that $\texttt{operand}_1$ is the destination and $\texttt{operand}_2$ is the source.

  $$\texttt{operand}_1 \leftarrow \texttt{operand}_1 \; \mathit{op} \; \texttt{operand}_2$$

- The Intel manuals are written according to the standard interpretation.

Source and Destination Operands

- However, the gnu interpretation of

  $$\texttt{mnemonic operand}_1\texttt{,operand}_2$$

  is that $\texttt{operand}_1$ is the source and $\texttt{operand}_2$ is the destination.

  $$\texttt{operand}_1 \; op \; \texttt{operand}_2 \rightarrow \texttt{operand}_2$$

- 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
    - “IA-32 Intel Architecture Software Developer’s Manual Volume 2A: Instruction Set Reference, A-M”
    - “IA-32 Intel Architecture Software Developer’s Manual Volume 2B: Instruction Set Reference, N-Z”

Instructions

- Different instructions require different numbers of operands.
- For example,
  - `hlt` - 0 operands
  - `inc` - 1 operand
  - `add` - 2 operands
  - `imul` - 1, 2, or 3 operands

Address Space

- 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.

Outline

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.

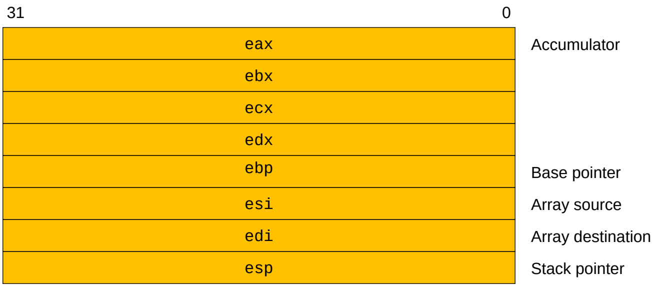
The General-Purpose Registers

- 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.

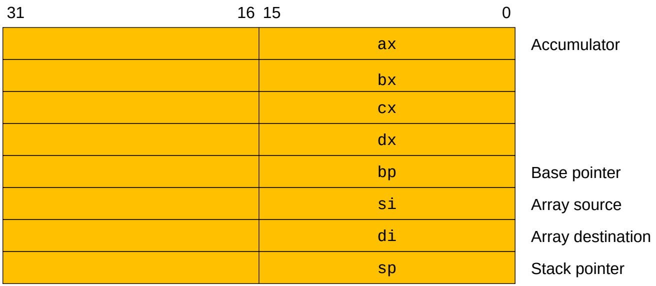
The General-Purpose Registers

- 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`.

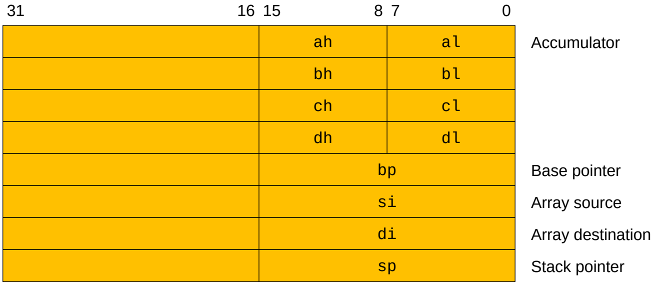
The General-Purpose 32-Bit Registers



The General-Purpose 16-Bit Registers



The General-Purpose 8-Bit Registers



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

Instruction Pointer

- Finally, there is the `eip` register, which is the instruction pointer.
- Register `eip` holds the address of the next instruction to be executed.
- We should never change the value of `eip` directly. It will be updated automatically as necessary.

Outline

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.

Outline

The Run-time Stack

- 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.

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.
- A good compiler would do that.

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.

$$\texttt{esp} \longrightarrow \texttt{ebp}$$

- 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.

$$\texttt{ebp} \longrightarrow \texttt{esp}$$

Outline

Assignment

Homework

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