17. Instruction Sets: Characteristics and Functions

Chapter 12
Introduction

- Section 12.1, 12.2, and 12.3 pp. 406-418

- Computer Designer:
  - Machine instruction set provides the functional requirements for the processor

- Assembly Programmer:
  - Machine instruction set provides the types of supported data, registers, and the capabilities of the ALU
ELEMENTS OF MACHINE INSTRUCTIONS
Machine Instructions

- We have already learned that an instruction is composed of a series of bytes where a portion of the instruction is used for the **opcode** and the other portion is used for one or more **operands**.

- Simple opcodes include: ADD, SUB, MUL, DIV, MOV, ...
Recall the IAS Instruction and Instruction Set

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Opcode</th>
<th>Symbolic Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data transfer</strong></td>
<td>00001010</td>
<td>LOAD MQ</td>
<td>Transfer contents of register MQ to accumulator AC</td>
</tr>
<tr>
<td></td>
<td>00001001</td>
<td>LOAD MQ,M(X)</td>
<td>Transfer contents of memory location X to MQ</td>
</tr>
<tr>
<td></td>
<td>00100001</td>
<td>STOR M(X)</td>
<td>Transfer contents of accumulator to memory location X</td>
</tr>
<tr>
<td></td>
<td>00000001</td>
<td>LOAD M(X)</td>
<td>Transfer M(X) to the accumulator</td>
</tr>
<tr>
<td></td>
<td>00000010</td>
<td>LOAD –M(X)</td>
<td>Transfer –M(X) to the accumulator</td>
</tr>
<tr>
<td></td>
<td>00000011</td>
<td>LOAD [M(X)]</td>
<td>Transfer absolute value of M(X) to the accumulator</td>
</tr>
<tr>
<td></td>
<td>00000100</td>
<td>LOAD –[M(X)]</td>
<td>Transfer –[M(X)] to the accumulator</td>
</tr>
<tr>
<td><strong>Unconditional branch</strong></td>
<td>00001101</td>
<td>JUMP M(X,0:19)</td>
<td>Take next instruction from left half of M(X)</td>
</tr>
<tr>
<td></td>
<td>00001110</td>
<td>JUMP M(X,20:39)</td>
<td>Take next instruction from right half of M(X)</td>
</tr>
<tr>
<td><strong>Conditional branch</strong></td>
<td>00001111</td>
<td>JUMP+ M(X,0:19)</td>
<td>If number in the accumulator is nonnegative, take next instruction from left half of M(X)</td>
</tr>
<tr>
<td></td>
<td>00010000</td>
<td>JUMP+ M(X,20:39)</td>
<td>If number in the accumulator is nonnegative, take next instruction from right half of M(X)</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>00000101</td>
<td>ADD M(X)</td>
<td>Add M(X) to AC; put the result in AC</td>
</tr>
<tr>
<td></td>
<td>00000111</td>
<td>ADD [M(X)]</td>
<td>Add [M(X)] to AC; put the result in AC</td>
</tr>
<tr>
<td></td>
<td>00000110</td>
<td>SUB M(X)</td>
<td>Subtract M(X) from AC; put the result in AC</td>
</tr>
<tr>
<td></td>
<td>00001000</td>
<td>SUB [M(X)]</td>
<td>Subtract [M(X)] from AC; put the remainder in AC</td>
</tr>
<tr>
<td></td>
<td>00001011</td>
<td>MUL M(X)</td>
<td>Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ</td>
</tr>
<tr>
<td></td>
<td>00011100</td>
<td>DIV M(X)</td>
<td>Divide AC by M(X); put the quotient in MQ and the remainder in AC</td>
</tr>
<tr>
<td><strong>Address modify</strong></td>
<td>00010010</td>
<td>STOR M(X,8:19)</td>
<td>Replace left address field at M(X) by 12 most significant bits of AC</td>
</tr>
<tr>
<td></td>
<td>00010011</td>
<td>STOR M(X,28:39)</td>
<td>Replace right address field at M(X) by 12 most significant bits of AC</td>
</tr>
</tbody>
</table>

(a) Number word

(b) Instruction word
Instruction Format

• Another simple instruction format might be:

![Instruction Format Diagram]

• What are the main differences between the above instruction format and the IAS instruction format?
Elements of an Instruction

- Operation code (opcode)
  - Do this

- Source Operand reference
  - To this

- Result Operand reference
  - Put the answer here

- Next Instruction Reference
  - When you have done that, do this...
Instruction Types

- We have spent most of our time in the high-level programming world. A high-level language must eventually be translated into some kind of machine language usually through some assembly language.

- Machine language instructions typically fall into one of four categories:
  - Data Processing: Arithmetic and logical instructions
  - Data Storage: Memory instructions
  - Data Movement: I/O instructions
  - Program Flow Control: Test and branch instructions
NUMBER OF ADDRESSES
Processor Architectures

• Historically, processor architectures have been defined in terms of the number of addresses contained within the instruction
Three Addresses

- Operand1, Operand2, Result OR Result, Operand1, Operand2

- \( a = b + c; \)

- Maybe a fourth - next instruction (usually implicit)
  - Not common

- Needs very long words to hold everything
Two Addresses

• One address doubles as operand and result

• \( a = a + b \)

• Reduces length of instruction

• Requires some extra work

• Temporary storage to hold some results
One Address

- Implicit second address
- Usually a register (accumulator)
- Common on early machines
Zero Address

- Zero addresses can be used for some instructions
- Uses a stack
Example: 3-Address Instruction

• What 3-address instructions could be used to compute the following:

\[ Y = \frac{A - B}{C + (D \cdot E)} \]
Example: 2-Address Instruction

- What 2-address instructions could be used to compute the following:

\[ Y = \frac{A - B}{C + (D \cdot E)} \]
Example: 1-Address Instruction

• What 2-address instructions could be used to compute the following:

\[ Y = \frac{A - B}{C + (D \cdot E)} \]
Example: 0-Address Instruction

<table>
<thead>
<tr>
<th>Number of Addresses</th>
<th>Symbolic Representation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>OP A, B, C</td>
<td>A ← B OP C</td>
</tr>
<tr>
<td>2</td>
<td>OP A, B</td>
<td>A ← A OP B</td>
</tr>
<tr>
<td>1</td>
<td>OP A</td>
<td>AC ← AC OP A</td>
</tr>
<tr>
<td>0</td>
<td>OP</td>
<td>T ← (T − 1) OP T</td>
</tr>
</tbody>
</table>

AC = accumulator  
T = top of stack  
(T − 1) = second element of stack  
A, B, C = memory or register locations

- What would the assembly language look like for the equation using a stack architecture?

\[ Y = \frac{A - B}{C + (D \cdot E)} \]
Instruction Set Design

• When designing an instruction set, consider
  • Operation repertoire
    • How many ops?
    • What can they do?
    • How complex are they?
  • Data types
  • Instruction formats
    • Length of opcode field
  • Registers
    • Number of CPU registers available
    • Which operations can be performed on which registers?
  • Addressing modes
  • RISC v CISC
TYPES OF OPERANDS
Types of Operands

• We know that the processor operates on data. General categories of data are:
  • Addresses

• Numbers
  • Binary integer (or binary fixed point)
  • Binary floating point
  • Decimal (packed decimal)

• Characters
  • ASCII etc.

• Logical Data
  • Bits or flags
INTEL X86 DATA TYPES
x86 Data Types

- 8 bit Byte
- 16 bit word
- 32 bit double word
- 64 bit quad word
- 128 bit double quadword
- Addressing is by 8 bit unit
- Words do not need to align at even-numbered address
- Data accessed across 32 bit bus in units of double word read at addresses divisible by 4
- Little endian
## x86 Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.</td>
</tr>
<tr>
<td>Integer</td>
<td>A signed binary value contained in a byte, word, or doubleword, using two's complement representation.</td>
</tr>
<tr>
<td>Ordinal</td>
<td>An unsigned integer contained in a byte, word, or doubleword.</td>
</tr>
<tr>
<td>Unpacked binary coded decimal (BCD)</td>
<td>A representation of a BCD digit in the range 0 through 9, with one digit in each byte.</td>
</tr>
<tr>
<td>Packed BCD</td>
<td>Packed byte representation of two BCD digits; value in the range 0 to 99.</td>
</tr>
<tr>
<td>Near pointer</td>
<td>A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.</td>
</tr>
</tbody>
</table>
## x86 Data Types

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Far pointer</td>
<td>A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.</td>
</tr>
<tr>
<td>Bit field</td>
<td>A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.</td>
</tr>
<tr>
<td>Bit string</td>
<td>A contiguous sequence of bits, containing from zero to $2^{32} - 1$ bits.</td>
</tr>
<tr>
<td>Byte string</td>
<td>A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{32} - 1$ bytes.</td>
</tr>
<tr>
<td>Floating point</td>
<td>See Figure 12.4.</td>
</tr>
<tr>
<td>Packed SIMD (single instruction, multiple data)</td>
<td>Packed 64-bit and 128-bit data types</td>
</tr>
</tbody>
</table>
x86 Data Types

- Byte unsigned integer
- Word unsigned integer
- Doubleword unsigned integer
- Quadword unsigned integer
- Byte signed integer
- Word signed integer
- Doubleword signed integer
- Quadword signed integer
x86 Data Types

Single precision floating point

Double precision floating point

Double extended precision floating point
The Pentium floating-point numbers conform to the IEEE 754 standard.

Pentium data is stored using little-endian style which means that the least significant byte is stored in the lowest address.

For the C declaration:

```c
int intVal = -10;
```

Show what memory would look like if the variable `intVal` is located at memory location 1000. Use HEX notation.
ARM DATA TYPES
ARM Data Types

- 8 (byte), 16 (halfword), 32 (word) bits
- Halfword and word accesses should be word aligned
- Nonaligned access alternatives
- Supports Big-Endian and Little-Endian
- Unsigned integer interpretation supported for all types
- Twos-complement signed integer interpretation supported for all types
- Majority of implementations do not provide floating-point hardware
  - Saves power and area
  - Floating-point arithmetic implemented in software
  - Optional floating-point coprocessor
  - Single- and double-precision IEEE 754 floating point data types