CS430 Computer Architecture

Spring 2015
Chapter 12
Instruction Sets: Characteristics and Functions

- Section 12.1 pp. 406-412
- Machine Instruction Characteristics
- We have already learned that an instruction is composed of a series of bytes where a portion of the instruction is for the opcode and the other portion is for one or more operands.
- Simple opcodes include: ADD, SUB, MUL, DIV, MOV, ...
Instruction Format

- An instruction format might be:
Elements of an Instruction

- Operation code (Op code)  
  — Do this
- Source Operand reference  
  — To this
- Result Operand reference  
  — Put the answer here
- Next Instruction Reference  
  — When you have done that, do this...
Elements of an Instruction

- 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
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
Processor Architectures

- 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
Processor Architectures

- One Address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
Processor Architectures

- Zero Address
  - zero addresses can be used for some instructions
  - uses a stack
Program to Execute

- \( Y = \frac{A-B}{C+(D \cdot E)} \) (3 address)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB Y, A, B</td>
<td>Y ← A – B</td>
</tr>
<tr>
<td>MPY T, D, E</td>
<td>T ← D × E</td>
</tr>
<tr>
<td>ADD T, T, C</td>
<td>T ← T + C</td>
</tr>
<tr>
<td>DIV Y, Y, T</td>
<td>Y ← Y ÷ T</td>
</tr>
</tbody>
</table>
Program to Execute

- \( Y = \frac{A - B}{C + (D \cdot E)} \) (2 address)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE Y, A</td>
<td>Y ← A</td>
</tr>
<tr>
<td>SUB Y, B</td>
<td>Y ← Y − B</td>
</tr>
<tr>
<td>MOVE T, D</td>
<td>T ← D</td>
</tr>
<tr>
<td>MPY T, E</td>
<td>T ← T × E</td>
</tr>
<tr>
<td>ADD T, C</td>
<td>T ← T + C</td>
</tr>
<tr>
<td>DIV Y, T</td>
<td>Y ← Y ÷ T</td>
</tr>
</tbody>
</table>
Program to Execute

- \( Y = \frac{A - B}{C + (D \times E)} \) (1 address)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD D</td>
<td>AC \leftarrow D</td>
</tr>
<tr>
<td>MPY E</td>
<td>AC \leftarrow AC \times E</td>
</tr>
<tr>
<td>ADD C</td>
<td>AC \leftarrow AC + C</td>
</tr>
<tr>
<td>STOR Y</td>
<td>Y \leftarrow AC</td>
</tr>
<tr>
<td>LOAD A</td>
<td>AC \leftarrow A</td>
</tr>
<tr>
<td>SUB B</td>
<td>AC \leftarrow AC - B</td>
</tr>
<tr>
<td>DIV Y</td>
<td>AC \leftarrow AC \div Y</td>
</tr>
<tr>
<td>STOR Y</td>
<td>Y \leftarrow AC</td>
</tr>
</tbody>
</table>
Program to Execute

- $Y = \frac{A-B}{C+(D\cdot E)}$

<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>

- What would the assembly language look like for the above equation using a stack architecture?
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 op code field
- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes
- RISC v CISC
Types of Operands

- Reading pp. 413-418
- 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
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 twos 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**: 7 bits, values 0 to 255
- **Word unsigned integer**: 16 bits, values 0 to 65535
- **Doubleword unsigned integer**: 32 bits, values 0 to 4294967295
- **Quadword unsigned integer**: 64 bits, values 0 to 18446744073709551615
- **Byte signed integer**: 8 bits, values -128 to 127
- **Word unsigned integer**: 16 bits, values -32768 to 32767
x86 Data Types

Doubleword unsigned integer

Quadword unsigned integer

Single precision floating point

Double precision floating point

Double extended precision floating point
x86 Data Types

- The Pentium does not require that word, double-words, or quad-words be aligned on any particular boundary; however, if data is accessed across a 32-bit bus, data transfers take place in 32-bit quantities beginning with an address divisible by 4. If data is not aligned on such a boundary, then multiple transfers are needed to get the data.

- 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 `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

- 8 (byte), 16 (halfword), 32 (word) bits
- Halfword and word accesses should be word aligned
- Nonaligned access alternatives
- 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
ARM Supports Big-Endian
ARM Supports Little-Endian