CSE378 - Machine Organization and Assembly language

Ben Dugan
Winter 2001

Course Introduction
Questions We Hope to Answer

- **Hardware/Software interface:**
  - Relationship between compilers, assemblers, linkers, loaders: who does what in terms of getting my program to run?
  - What kind of instructions does the machine understand?

- **Organization:**
  - What are the basic pieces of the machine (registers, cache, ALU, busses)?
  - How are these pieces connected? How are they controlled?

- **Performance:**
  - What does it mean for one machine to be “faster” than another?
  - What are MFLOPS, MIPS, benchmark programs?

- **Implementation:**
  - What’s logic design?
  - What are the technologies (CMOS, VLSI, etc)?

Instruction Set Architecture

- ISA is an interface between the hardware and software.
- ISA is what is visible to the programmer (note that the OS and users might have different view)
- ISA consists of
  - instructions (operations, how are they encoded?)
  - information units (what is their size, how are they addressed)
  - registers (general or special purpose)
  - input-output control
- ISA is an abstract view of the machine: underlying details should be hidden from the programmer (although this is not always the case)
Computer Families

- Sequence of machines that have the same ISA (binary compatible). For example:
  1. IBM 360, 370, etc
  2. IBM PowerPC (601, 603, etc)
  3. DEC PDP-11, VAX
  4. Intel x86 (80286, 80386, 80486, Pentium)
  5. Motorola 680x0
  6. MIPS Rx000, SGI
  7. Sun SPARC
  8. DEC Alpha (21x64)

- With “portable” software, are “binary compatible” machines important?

Stored Program Computer History

- ENIAC: programmed by connecting wires and setting switches, data read from punched cards
- 1944-45: von Neumann joins ENIAC group (at U. Penn.), writes memo based on work with Eckert and Mauchly
- 1946: Burks, Goldstine and von Neumann (at IAS) write a paper based on above memo explaining the concept of the stored program computer (von Neumann machine)
- 1946 paper introduced the idea of treating the program as data, using binary representations, and defined the basic building blocks of the machine
- History neglects to credit many of the pioneers esp. Eckert and Mauchly, but also the early programmers of machines like ENIAC (usually women).
Computer Generations

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Tech-</td>
<td>Vacuum tubes</td>
<td>transis-</td>
<td>integrated</td>
<td>LSI</td>
<td>VLSI</td>
<td>Very VLSI</td>
</tr>
<tr>
<td>nology</td>
<td>tor</td>
<td>tors</td>
<td>circuits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Struc-</td>
<td>single proc-</td>
<td>multiple</td>
<td>micros</td>
<td>32-bit</td>
<td>64-bit +</td>
<td></td>
</tr>
<tr>
<td>ture</td>
<td>essor</td>
<td>functional</td>
<td>and minis</td>
<td>micro-</td>
<td>MP micros</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Vacuum</td>
<td>Magnetic</td>
<td>semi-</td>
<td>semi-</td>
<td>semi-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tubes</td>
<td>core</td>
<td>conductors</td>
<td>cond. 64KB</td>
<td>cond. 512KB</td>
<td></td>
</tr>
<tr>
<td>Example machine</td>
<td>UNIVAC</td>
<td>Burroughs</td>
<td>PDP-11</td>
<td>Apple II</td>
<td>Apple Mac,</td>
<td>Alpha,</td>
</tr>
<tr>
<td></td>
<td>1950s</td>
<td>5500</td>
<td>1969-77</td>
<td>1978-mid 80s</td>
<td>1980s</td>
<td>SPARC,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1960-68</td>
<td></td>
<td></td>
<td></td>
<td>1990s</td>
</tr>
</tbody>
</table>

- Instructions and data are binary strings
- 5 basic building blocks: arithmetic (datapath), control, memory, input, output:

Stored Program Computer

Control flow

Data/instruction flow
Registers

- Registers are visible both to hardware and programmer
  - High-speed storage of operands
  - Easy to name
  - Also used to address memory
- Most current computers have 32 or 64 registers
- Not all registers are "equal"
  - Some are special purpose (e.g., in MIPS $0$ is hardwired to 0).
  - Integer / Floating point
  - Conventions (stack pointers)
- Why no more than 32 or 64? (at least 3 good reasons)
The Memory System

- Memory is a hierarchy of devices/components which get increasingly faster (and more expensive) as they get nearer to the CPU:

<table>
<thead>
<tr>
<th>Memory level</th>
<th>Capacity (bytes)</th>
<th>Speed</th>
<th>Relative Speed</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>100s to 1000s</td>
<td>nanoseconds</td>
<td>1</td>
<td>??</td>
</tr>
<tr>
<td>Cache</td>
<td>16KB on-chip</td>
<td>nanoseconds</td>
<td>1-2</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>1MB off-chip</td>
<td>10s of ns</td>
<td>5-10</td>
<td>$100/MB</td>
</tr>
<tr>
<td>Primary memory</td>
<td>10-100MB</td>
<td>10s to 100s</td>
<td>10-100</td>
<td>$5/MB</td>
</tr>
<tr>
<td>Secondary mem.</td>
<td>1-10GB</td>
<td>10s of ms</td>
<td>1,000,000</td>
<td>$.1/MB</td>
</tr>
</tbody>
</table>

- Library metaphor of memory hierarchy

Review of Binary/Hex Representation

- Remember that computers represent all data (integers, floating point numbers, characters, instructions, etc.) in a binary representation. Interpretation depends on context.
- Representing integers: What characteristics does our scheme need?
  - Easy test for positive/negative.
  - Equal number of positive and negative numbers
  - Easy check for overflow
- Different schemes: sign and magnitude, 1’s complement, 2’s complement
- 2’s complement tricks (sign bit extension, converting from positive to negative, addition/subtraction)
- Hexidecimal notation
- Common powers of 2 (10:1024 (1K), 20:(1M), 30(1G), 8:256, 16: 64K, 32(4G))
**Why 2s Complement?**

- Easy sign test, (roughly) equal number of positive and negative numbers, easy to negate numbers, easy to add. (Note that with negation and add, we get subtraction for “free”.)

- How does the machine multiply numbers? One (very slow) way is repeated addition, here’s a faster way:
  
  ```plaintext
  // Regs A & B will hold the values to multiply
  // Reg product will hold the result.
  product <= 0.
  while (A != 0)
    if A is odd then
      product <= product + B.
    halve A.          // divide by 2; drop the remainder
    double B.
  end loop
  // at this point, product contains the answer
  ```

- Note we only need add, not-equal-to-zero, test-for-odd, halve, and double. How many times do we iterate?

---

**Information Units**

- Basic unit is the bit (stores a 0 or a 1)
- Bits are grouped together into larger units:
  - bytes = 8 bits
  - words = 4 bytes
  - double words = 2 words (8 bytes)
Memory

- Memory is an array of information units
  - Each unit has the same size
  - Each unit has a unique address
  - Address and contents are different

A memory of size N

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>122</td>
</tr>
<tr>
<td>1</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>n-1</td>
<td></td>
</tr>
</tbody>
</table>

- A C variable is an abstraction for a memory location

Addressing

- The address space is the set of all information units that a program can reference
- Most machines today are byte addressable
- Processor "size" impacts the size of the address space:
  - 16 bit processor: 64KB (too small nowadays)
  - 32 bit processor: 4GB (starting to be too small)
  - 64 bit processor: really big (should last for a while...)
- Rule of thumb: We’re using up address space at a rate of around 1 bit per year...
Addressing Words

- On a byte addressable machine, every word starts at an address divisible by 4:

A memory of size N bytes

- Big vs. Little Endian: within a data unit (e.g., word), how are the individual bytes laid out?
- Little/Big: address of data unit is address of low/high order byte (DEC MIPS is Little; SGI MIPS, SPARC are Big)

The CPU - Instruction Execution Cycle

- The CPU executes a program by following this cycle:
  1. Fetch the next instruction
  2. Decode it
  3. Execute it
  4. Compute the address of the next instruction
  5. Goto 1.
Instructions

- An instruction tells the CPU:
  - The operation to be performed (the opcode)
  - The operands (zero or more)
- For a given instruction, the ISA specifies
  - the meaning (semantics) of the opcode
  - how many operands are required (and their types)
- Operands can be of the following type
  - registers
  - memory address
  - constant (immediate data)
- In MIPS, the operands are typically registers or small constants