Today’s Lecture

Today we’ll cover:

- What do machine instructions look like?
  - Instructions & condition codes
  - Branching
  - Addressing.
- How do we store data in the machine?
  - Text
  - Floating point
  - Data structures
- Fetch-Execute cycle:
  - “Tie it all up”

Arithmetic & Logical Instructions

- Some common ALU instructions are:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>C/Java Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>d = a &amp; b;</td>
</tr>
<tr>
<td>xor</td>
<td>d = a ^ b;</td>
</tr>
<tr>
<td>bis</td>
<td>d = a</td>
</tr>
<tr>
<td>bic</td>
<td>d = a &amp; (~b);</td>
</tr>
<tr>
<td>add</td>
<td>d = a + b;</td>
</tr>
<tr>
<td>sub</td>
<td>d = a - b;</td>
</tr>
<tr>
<td>rsb</td>
<td>d = b - a;</td>
</tr>
<tr>
<td>shl</td>
<td>d = a &lt;&lt; b;</td>
</tr>
<tr>
<td>shr</td>
<td>d = a &gt;&gt; b;</td>
</tr>
</tbody>
</table>

Both d and a must be registers; b can be a register or a (small) constant.

- Typically also have addc and subc, which handle carry or borrow (for multi-precision arithmetic), e.g.

```
add d0, a0, b0 // compute "low" part.
addc d1, a1, b1 // compute "high" part.
```

- May also get:
  - Arithmetic shifts: asr and asl(?)
  - Rotates: ror and rol.

Conditional Execution

- Seen flags C, N, V; add Z (zero), logical NOR of all bits in output.

- Can predicate execution based on (some combination) of flags, e.g.

```
sub d, a, b    // compute d = a - b
beq proc1    // if equal, goto proc1
br proc2     // otherwise goto proc2
```

Java equivalent approximately:

```
if (a-b) proc1() else proc2();
```

- On ARM everything conditional, e.g.

```
sub d, a, b    # compute d = a - b
moveq d, #5   # if equal, d = 5;
move d, #7     # otherwise d = 7;
```

Java equiv: d = (a-b) ? 5 : 7;

- “Silent” versions useful when don’t really want result, e.g. tst, teq, cmp.

- Alt (MIPS): beq reg1 reg2 L1
**Condition Codes**

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Meaning</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ, Z</td>
<td>Equal, zero</td>
<td>Z == 1</td>
</tr>
<tr>
<td>NE, NZ</td>
<td>Not equal, non-zero</td>
<td>Z == 0</td>
</tr>
<tr>
<td>MI</td>
<td>Negative</td>
<td>N == 1</td>
</tr>
<tr>
<td>PL</td>
<td>Positive (incl. zero)</td>
<td>N == 0</td>
</tr>
<tr>
<td>CS, HS</td>
<td>Carry, higher or same</td>
<td>C == 1</td>
</tr>
<tr>
<td>CC, LO</td>
<td>No carry, lower</td>
<td>C == 0</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>V == 1</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>V == 0</td>
</tr>
<tr>
<td>HI</td>
<td>Higher</td>
<td>C == 1 &amp; Z == 0</td>
</tr>
<tr>
<td>LS</td>
<td>Lower or same</td>
<td>C == 0</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal</td>
<td>N == V &amp; Z == 0</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
<td>N == V</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>N != V</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
<td>N != V</td>
</tr>
</tbody>
</table>

- HS, LO, etc. used for unsigned comparisons (recall that $C$ means “borrow”).
- GE, LT, etc. used for signed comparisons: check both $N$ and $V$ so always works.

**Loads & Stores**

- Have variable sized values, e.g. bytes (8-bits), words (16-bits), longwords (32-bits) and quadwords (64-bits).
- Load or store instructions usually have a suffix to determine the size, e.g. ‘b’ for byte, ‘w’ for word, ‘l’ for longword.
- When storing > 1 byte, have two main options: big endian and little endian; e.g. storing longword 0xDEADBEEF into memory at address 0x4.

**Big Endian**

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
<td>EF</td>
<td>BE</td>
<td>AD</td>
</tr>
</tbody>
</table>

**Little Endian**

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>BE</td>
<td>AD</td>
<td>DE</td>
<td>DE</td>
<td>AD</td>
<td>BE</td>
</tr>
</tbody>
</table>

If read back a byte from address 0x4, get 0xDE if big-endian, or 0xEF if little-endian.
- Today have x86 & Alpha little endian; Sparc & 68K, big endian; MIPS & ARM either.

**Addressing Modes**

- An addressing mode tells the computer where the data for an instruction is to come from.
- Get a wide variety, e.g.

  - **Register:**
    - Add: $r1$, $r2$, $r3$
    - Immediate: $r1$, $r2$, $#25$
  - **PC Relative:** $bq 0x20$
  - **Register Indirect:** $ldr r1, [r2]$
  - " + Displacement:" $str r1, [r2, #8]$
  - **Indexed:** $movl r1, (r2, r3)$
  - **Absolute/Direct:** $movl r1, 0x0F1EA0130$
  - **Memory Indirect:** $addl r1, 0x0F1EA0130$

- Most modern machines are load/store ⇒ only support first five:
  - allow at most one memory ref per instruction (there are very good reasons for this)
- Note that CPU generally doesn’t care *what* is being held within the memory.
- i.e. up to *programmer* to interpret whether data is an integer, a pixel or a few characters in a novel.

**Representing Text**

- Two main standards:
  1. **ASCII:** 7-bit code holding (English) letters, numbers, punctuation and a few other characters.
  2. **Unicode:** 16-bit code supporting practically all international alphabets and symbols.
- ASCII default on many operating systems, and on the early Internet (e.g. e-mail).
- Unicode becoming more popular (esp UTF-8!).
- In both cases, represent in memory as either *strings* or *arrays*: e.g. “Pub Time!”

<table>
<thead>
<tr>
<th>String</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 62 75 50</td>
<td>Ox351A, 25E4</td>
</tr>
<tr>
<td>65 60 69 54</td>
<td>Ox351A, 25E8</td>
</tr>
<tr>
<td>xx xx 00 21</td>
<td>Ox351A, 25EC</td>
</tr>
</tbody>
</table>

- 0x492077697368206974207761732032a2d28
Floating Point

- In many cases want to deal with very large or very small numbers.
- Use idea of “scientific notation”, e.g. \( n = m \times 10^e \)
  - \( m \) is called the mantissa
  - \( e \) is called the exponent.
  
  e.g. \( C = 3.01 \times 10^8 \) m/s.
- For computers, use binary i.e. \( n = m \times 2^e \), where \( m \) includes a “binary point”.
- Both \( m \) and \( e \) can be positive or negative; typically
  - sign of mantissa given by an additional sign bit.
  - exponent is stored in a biased (excess) format.
  
  \[ n = (-1)^m \times 2^{e-b} \] where \( 0 \leq m < 2 \) and \( b \) is the bias.
  
  e.g. 4-bit mantissa & 3-bit bias-3 exponent allows positive range \([0.001_2 \times 2^{-3}, 1.111_2 \times 2^8]\)
  
  \[ \left\lfloor \frac{1}{2^5}, \frac{15}{30} \right\rfloor, \left\lfloor \frac{1}{60}, 30 \right\rfloor \]

Floating Point cont.

- In practice use IEEE floating point with normalised mantissa \( m = 1.x \ldots x_2 \)
  \[ n = (-1)^m \times 2^{e-b} \]
- Both single (float) and double (double) precision:

  
  - IEEE fp reserves \( e = 0 \) and \( e = \max\):  
    - \( \pm 0 \): both \( e \) and \( m \) zero.
    - \( \pm \infty \): \( e = \max \), \( m \) zero.
    - NaNs: \( e = \max \), \( m \) non-zero.
    - Denorms: \( e = 0 \), \( m \) non-zero
  
  - Normal positive range \([2^{-126}, \sim 2^{128}]\) for single, or \([2^{-1022}, \sim 2^{1024}]\) for double.
  
  - NB: still only \( 2^{32}/2^{64} \) values — just spread out.

Data Structures

- **Records / structures**: each field stored as an offset from a base address.
- **Variable size structures**: explicitly store addresses (pointers) inside structure, e.g.
  
  datatype rec = node of int * int * rec  
  
  leaf of int;

  val example = node(4, 5, node(6, 7, leaf(8)));

  Imagine example is stored at address 0x1000:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0F30</td>
<td>0xFFF</td>
<td>Constructor tag for a leaf Integer 8</td>
</tr>
<tr>
<td>0x0F34</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0x0F3C</td>
<td>0xFFF</td>
<td>Constructor tag for a node Integer 6</td>
</tr>
<tr>
<td>0x0F40</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0x0F44</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0x0F48</td>
<td>0x0F30</td>
<td>Address of inner node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>0xFFF</td>
<td>Constructor tag for a node Integer 4</td>
</tr>
<tr>
<td>0x1004</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0x100C</td>
<td>0x0F30</td>
<td>Address of inner node</td>
</tr>
</tbody>
</table>

Instruction Encoding

- An instruction comprises:
  
  - **a. an opcode**: specify what to do.
  - **b. zero or more operands**: where to get values
    
  e.g. add r1, r2, r3 = 1010111 001 00 001 01

  - Old machines (and x86) use variable length encoding motivated by low code density.

  - Most modern machines use fixed length encoding for simplicity. e.g. ARM ALU operations.

  and \( r13, r13, #3 = 0xe20dd01f = \)

  
  \[
  \begin{array}{cccccccc}
  1110 & 00 & 1 & 0000 & 0 & 1101 & 1101 & 000000111111
  \end{array}
  \]

  bic \( r3, r3, r2 = 0xe3c33002 = \)

  
  \[
  \begin{array}{cccccccc}
  1110 & 00 & 1100 & 0 & 0011 & 0011 & 000000000000
  \end{array}
  \]

  cmp \( r1, r2 = 0xe1510002 = \)

  
  \[
  \begin{array}{cccccccc}
  1110 & 00 & 1400 & 1 & 0001 & 0000 & 000000000000
  \end{array}
  \]
Fetch-Execute Cycle Revisited

1. CU fetches & decodes instruction and generates (a) control signals and (b) operand information.
2. Inside EU, control signals select functional unit (“instruction class”) and operation.
3. If ALU, then read one or two registers, perform operation, and (probably) write back result.
4. If BU, test condition and (maybe) add value to PC.
5. If MAU, generate address (“addressing mode”) and use bus to read/write value.
6. Repeat ad infinitum.

Summary

You should now understand:

- Different forms of machine instructions,
- Different forms of addressing,
- Representing text and data structures,
- Floating point representation.

Next lecture: Buses and I/O devices

Background Reading:

- Hennessy/Patterson:
  - Chapter 3 - Machine Instructions (MIPS)
  - Section 4.8 - Floating Point