Lecture 2: The Instruction Set Architecture

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August
Quiz 0

Combinational

Sequential

MUX

\[ x' \cdot y + x \cdot y' = \text{XOR} \]

XOR

\[
\begin{array}{c|c|c|c|c}
\hline
x & y & \text{XOR} \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{array}
\]
Quiz 0

- Bit: Binary Digit / two distinct states of matter or energy

\[
\begin{array}{ccc}
XOR: & x & y & out \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

- XOR: Combinational Circuit

- MUX: Sequential State

out depends on in

Lives in the Now

out depends on in and state

History
CD

3 Miles of Music
Pits and Lands

Transition represents a bit state (1/on/red/female/heads)
No change represents other state (0/off/white/male/tails)
As Music:

\[01110101 \times 2 = 117/256 \text{ position of speaker}\]

As Number:

\[01110101 \times 2 = 1 + 4 + 16 + 32 + 64 = 117_{10} = 75_{16}\]

(Get comfortable with base 2, 8, 10, and 16.)

As Text:

\[01110101 \times 2 = 117^{th} \text{ character in the ASCII codes} = “u”\]
# Interpretation – ASCII

<table>
<thead>
<tr>
<th>ASCII value</th>
<th>Character</th>
<th>Control character</th>
<th>ASCII value</th>
<th>Character</th>
<th>ASCII value</th>
<th>Character</th>
<th>ASCII value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>(null)</td>
<td>NUL</td>
<td>032</td>
<td>(space)</td>
<td>064</td>
<td>@</td>
<td>096</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>☐</td>
<td>SOH</td>
<td>033</td>
<td>!</td>
<td>065</td>
<td>A</td>
<td>097</td>
<td>a</td>
</tr>
<tr>
<td>002</td>
<td>☠</td>
<td>STX</td>
<td>034</td>
<td>&quot;</td>
<td>066</td>
<td>B</td>
<td>098</td>
<td>b</td>
</tr>
<tr>
<td>003</td>
<td>♥</td>
<td>ETX</td>
<td>035</td>
<td>#</td>
<td>067</td>
<td>C</td>
<td>099</td>
<td>c</td>
</tr>
<tr>
<td>004</td>
<td>♤</td>
<td>EOT</td>
<td>036</td>
<td>$</td>
<td>068</td>
<td>D</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>005</td>
<td>♣</td>
<td>ENQ</td>
<td>037</td>
<td>%</td>
<td>069</td>
<td>E</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>006</td>
<td>♦</td>
<td>ACK</td>
<td>038</td>
<td>&amp;</td>
<td>070</td>
<td>F</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>007</td>
<td>(beep)</td>
<td>BEL</td>
<td>039</td>
<td></td>
<td>071</td>
<td>G</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>008</td>
<td>☘</td>
<td>BS</td>
<td>040</td>
<td>(</td>
<td>072</td>
<td>H</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>009</td>
<td>(tab)</td>
<td>HT</td>
<td>041</td>
<td>)</td>
<td>073</td>
<td>I</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>010</td>
<td>(line feed)</td>
<td>LF</td>
<td>042</td>
<td>'</td>
<td>074</td>
<td>J</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>011</td>
<td>(home)</td>
<td>VT</td>
<td>043</td>
<td>+</td>
<td>075</td>
<td>K</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>012</td>
<td>(form feed)</td>
<td>FF</td>
<td>044</td>
<td></td>
<td>076</td>
<td>L</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>013</td>
<td>(carriage return)</td>
<td>CR</td>
<td>045</td>
<td>-</td>
<td>077</td>
<td>M</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>014</td>
<td>↳</td>
<td>SO</td>
<td>046</td>
<td>.</td>
<td>078</td>
<td>N</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>015</td>
<td>☥</td>
<td>SI</td>
<td>047</td>
<td>/</td>
<td>079</td>
<td>O</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>016</td>
<td>⏯</td>
<td>DLE</td>
<td>048</td>
<td>0</td>
<td>080</td>
<td>P</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>017</td>
<td>←</td>
<td>DC1</td>
<td>049</td>
<td>1</td>
<td>081</td>
<td>Q</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>018</td>
<td>↑</td>
<td>DC2</td>
<td>050</td>
<td>2</td>
<td>082</td>
<td>R</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>019</td>
<td>!!</td>
<td>DC3</td>
<td>051</td>
<td>3</td>
<td>083</td>
<td>S</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>020</td>
<td>=</td>
<td>DC4</td>
<td>052</td>
<td>4</td>
<td>084</td>
<td>T</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>021</td>
<td>$</td>
<td>NAK</td>
<td>053</td>
<td>5</td>
<td>085</td>
<td>U</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>022</td>
<td>—</td>
<td>SYN</td>
<td>054</td>
<td>6</td>
<td>086</td>
<td>V</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>023</td>
<td>↑↑</td>
<td>ETB</td>
<td>055</td>
<td>7</td>
<td>087</td>
<td>W</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>024</td>
<td>↓↓</td>
<td>CAN</td>
<td>056</td>
<td>8</td>
<td>088</td>
<td>X</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>025</td>
<td>→</td>
<td>EM</td>
<td>057</td>
<td>9</td>
<td>089</td>
<td>Y</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>026</td>
<td>←</td>
<td>SUB</td>
<td>058</td>
<td>:</td>
<td>090</td>
<td>Z</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>027</td>
<td>←</td>
<td>ESC</td>
<td>059</td>
<td>;</td>
<td>091</td>
<td>[</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>028</td>
<td>(cursor right)</td>
<td>FS</td>
<td>060</td>
<td>&lt;</td>
<td>092</td>
<td>\</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>029</td>
<td>(cursor left)</td>
<td>GS</td>
<td>061</td>
<td>=</td>
<td>093</td>
<td>]</td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>030</td>
<td>(cursor up)</td>
<td>RS</td>
<td>062</td>
<td>&gt;</td>
<td>094</td>
<td>^</td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>031</td>
<td>(cursor down)</td>
<td>US</td>
<td>063</td>
<td>?</td>
<td>095</td>
<td>–</td>
<td>127</td>
<td>☐</td>
</tr>
</tbody>
</table>
Princeton Computer Science Building West Wall
Interpretation

As Music:
\[01110101_2 = 117/256\] position of speaker

As Number:
\[01110101_2 = 1 + 4 + 16 + 32 + 64 = 117_{10} = 75_{16}\]

As Text:
\[01110101_2 = 117^{th} \text{ character in the ASCII codes} = \text{“u”}\]

CAN ALSO BE INTERPRETED AS MACHINE INSTRUCTION!
Binary Code and Data (Hello World!)

- Programs consist of Code and Data
- Code and Data are Encoded in Bits

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000 0000 .ELF.........
...
00000260: 5002 0000 0000 0000 006c 6962 632e 736f P...........libc.so
00000270: 2e36 2e31 0070 7269 6e74 6600 5f5f 6c69 .6.1.printf.__li
00000280: 6263 5f73 7461 7274 5f6d 6e20 6e00 474c bc_start_main.GL
00000290: 4942 4335f 322e 3200 0000 0200 0200 0000 IBC_2.2........
...
00000860: 4865 6c6c 6f20 576f 626c 6420 0000 0000 Hello world!....
...
4000000000000690 <main>:
4000000000000690: 00 10 15 08 80 05 [MII] alloc r34=ar.pfs,5,4,0
4000000000000696: 04 00 c4 00 mov r35=r12
40000000000006a0: 0a 20 81 03 00 24 mov r33=00
40000000000006a6: 40 02 90 30 20 00 [MMI] add r35=96,r1,;,
40000000000006ac: 04 08 00 84 ld8 r36=[r36]
40000000000006b0: 00 00 00 01 00 mov r32=r1
40000000000006b6: 00 00 00 02 00 [MFB] nop.m 0x0
40000000000006bc: 00 00 00 00 00 [MFB] br.call.sptk.many b0=40000000000000460,;;
40000000000006c0: 00 08 00 40 00 21 [MII] mov r1=r32
40000000000006c6: 80 00 00 00 42 00 mov r8=r0
40000000000006cc: 20 02 aa 00 mov.i ar.pfs=r34
40000000000006d0: 00 00 00 00 01 00 [MII] nop.m 0x0
40000000000006d6: 00 08 05 80 03 80 mov b0=r33
40000000000006dc: 01 18 01 84 mov r12=r35
40000000000006e0: 1d 00 00 00 01 00 [MFB] nop.m 0x0
40000000000006e6: 00 00 00 02 00 80 nop.f 0x0
40000000000006ec: 08 00 84 00 br.ret.sptk.many b0,;;
```
Software: Produce Bits Instructing Machine to Manipulate State or Produce I/O

Hardware: Read and Obey Instruction Bits
Instructions

Computers process information

- Input/Output (I/O)
- State (memory)
- Computation (processor)

- Instructions instruct processor to manipulate state
- Instructions instruct processor to produce I/O in the same way
Typical modern machine has this **architectural** state:

1. Main Memory
2. Registers
3. Program Counter

**Architectural** – Part of the assembly programmer’s interface
(Implementation has additional microarchitectural state)
Main Memory (AKA: RAM – Random Access Memory)
- Data can be accessed by address (like a big array)
- Large but relatively slow
- Decent desktop machine: 1 Gigabyte, 800MHz

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01011001₂</td>
</tr>
<tr>
<td>0001</td>
<td>F5₁₆</td>
</tr>
<tr>
<td>0002</td>
<td>78₁₆</td>
</tr>
<tr>
<td>0003</td>
<td>3A₁₆</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00000000₂</td>
</tr>
</tbody>
</table>

Byte Addressable
State – Main Memory

Read:
1. Indicate READ
2. Give Address
3. Get Data

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01011001\textsubscript{2}</td>
</tr>
<tr>
<td>0001</td>
<td>F5\textsubscript{16}</td>
</tr>
<tr>
<td>0002</td>
<td>78\textsubscript{16}</td>
</tr>
<tr>
<td>0003</td>
<td>3A\textsubscript{16}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00000000\textsubscript{2}</td>
</tr>
</tbody>
</table>

**Diagram:**
- **READ**
- **Address: 0002**
- **Data: 78\textsubscript{16}**
State – Main Memory

Write:
1. Indicate WRITE
2. Give Address and Data

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01011001&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>0001</td>
<td>F5&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>0002</td>
<td>78&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>0003</td>
<td>12&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00000000&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
State – Registers

Registers (AKA: Register File)

- Data can be accessed by register number (address)
- Small but relatively fast (typically on processor chip)
- Decent desktop machine: 8 32-bit registers, 3 GHz

<table>
<thead>
<tr>
<th>Register</th>
<th>Data in Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000_{16}</td>
</tr>
<tr>
<td>1</td>
<td>F629D9B5_{16}</td>
</tr>
<tr>
<td>2</td>
<td>7B2D9D08_{16}</td>
</tr>
<tr>
<td>3</td>
<td>00000001_{16}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>DEADBEEF_{16}</td>
</tr>
</tbody>
</table>
State – Program Counter

Program Counter (AKA: PC, Instruction Pointer, IP)
- Instructions change state, but which instruction now?
- PC holds memory address of currently executing instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>Data in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01011001_2</td>
</tr>
<tr>
<td>0001</td>
<td>F5_{16}</td>
</tr>
<tr>
<td>0002</td>
<td>ADD_{inst}</td>
</tr>
<tr>
<td>0003</td>
<td>SUBTRACT_inst</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00000000_2</td>
</tr>
</tbody>
</table>
Program Counter (AKA: PC, Instruction Pointer, IP)

- Instructions change state, but which instruction now?
- PC holds address of currently executing instruction
- PC is updated after each instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>Data in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01011001₂</td>
</tr>
<tr>
<td>0001</td>
<td>F5₁₆</td>
</tr>
<tr>
<td>0002</td>
<td>ADDₜᵢₙₜ</td>
</tr>
<tr>
<td>0003</td>
<td>SUBTRACTₜᵢₙₜ</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00000000₀₂</td>
</tr>
</tbody>
</table>
State – Summary

Typical modern machine has this architectural state:
1. Main Memory – Big, Slow
2. Registers – Small, Fast (always on processor chip)
3. Program Counter – Address of executing instruction

Architectural – Part of the assembly programmer’s interface
(implementation has additional microarchitectural state)
An Aside: State and The Core Dump

- Core Dump: the state of the machine at a given time
- Typically at program failure
- Core dump contains:
  - Register Contents
  - Memory Contents
  - PC Value

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
</tr>
<tr>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08: 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>10: 9222 9120 1121 A120 1121 A121 7211 0000</td>
</tr>
<tr>
<td>18: 0000 0001 0002 0003 0004 0005 0006 0007</td>
</tr>
<tr>
<td>20: 0008 0009 000A 000B 000C 000D 000E 000F</td>
</tr>
<tr>
<td>28: 0000 0000 0000 FE10 FACE CAFE ACED CEDE</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E8: 1234 5678 9ABC DEF0 0000 0000 F00D 0000</td>
</tr>
<tr>
<td>F0: 0000 0000 EEEE 1111 EEEE 1111 0000 0000</td>
</tr>
<tr>
<td>F8: B1B2 F1F5 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
Software: Produce Bits Instructing Machine to Manipulate State or Produce I/O

Hardware: Read and Obey Instruction Bits
Instructions

An ADD Instruction:

```
add r1 = r2 + r3    (assembly)
```

Parts of the Instruction:

- **Opcode (verb)** – what operation to perform
- **Operands (noun)** – what to operate upon
- **Source Operands** – where values come from
- **Destination Operand** – where to deposit data values
Instructions

“The vocabulary of commands”
Specify how to operate on state

Example:
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>
Instructions:

“The vocabulary of commands”
Specify how to operate on state

Example:

40: add $3 = r2 + r3$
44: sub $r3 = r1 - r0$
48: store $M[r3] = r1$
52: load $r2 = M[2]$

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>0</td>
</tr>
</tbody>
</table>

Program Counter

40
Instructions

Example:
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>0</td>
</tr>
</tbody>
</table>

Program Counter

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
</tr>
</tbody>
</table>
Instructions:

“The vocabulary of commands”
Specify how to operate on state

Example:

40: add r1 = r2 + r3
44: sub 3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
Instructions:

“The vocabulary of commands”
Specify how to operate on state

Example:
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = 3
52: load r2 = M[ 2 ]
Instructions:
“The vocabulary of commands”
Specify how to operate on state

Example:
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load 5 = M[ 2 ]
Instructions

"The vocabulary of commands"
Specify how to operate on state

Example:
40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
## Instructions

**Note:**

1. **Insts Executed in Order**
2. **Addressing Modes**

**Example:**

40: `add r1 = r2 + r3`
44: `sub r3 = r1 - r0`
48: `store M[r3] = r1`
52: `load r2 = M[2]`

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>0</td>
</tr>
</tbody>
</table>
main() {
    int a = 15, b = 1, c = 2;

    add r1 = r2 + r3       a = b + c;  /* a gets 3 */

    sub r3 = r1 - r0       c = a;  /* c gets 3 */

    store M[ r3 ] = r1    *(int *)c = a;
                          /* M[c] = a */

    load r2 = M[ 2 ]      b = *(int *)(2);
                         /* b gets M[2] */
}
Branching

Suppose we could only execute instructions in sequence.

Recall from our example:

40: add r1 = r2 + r3 \( PC = PC + 4 \)
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]

• In a decent desktop machine, how long would the longest program stored in main memory take?
• Assume:
  • 1 instruction per cycle
  • An instruction is encoded in 4 bytes (32 bits)
Therefore…

- Some instructions must execute more than once
- PC must be updated

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
56: PC = 40
Unconditional Branches

- Unconditional branches always update the PC
- AKA: Jump instructions

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[r3] = r1
52: load r2 = M[2]
56: jump 40

- How long will the program take?
Conditional Branch

- Conditional Branch sometimes updates PC
- AKA: Branch, Conditional Jump

Example

40: r1 = 10
44: r1 = r1 - 1
48: branch r1 > 0, 44  \[\text{if r1 is greater than 0, PC = 44}\]
52: halt

- How long will this program take?
Conditional Branch

- What does this look like in C?

- Example

  10: “Hello\n” ; data in memory
  36: arg1 = 10 ; argument memory address is 10
  40: r1 = 10
  44: r1 = r1 - 1
  48: call printf ; printf(arg1)
  52: branch r1 > 0, 44
  56: halt

Details about red instructions/data next time...
Indirect Branches

- Branch address may also come from a register
- AKA: Indirect Jump

Example:

40: add r1 = r2 + r3
44: sub r3 = r1 - r0
48: store M[ r3 ] = r1
52: load r2 = M[ 2 ]
56: jump r4
60: halt
Branch Summary

- Reduce, Reuse, Recycle (instructions)
- Branch instructions update state

### Registers

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0788</td>
<td>B700</td>
<td>0010</td>
<td>0401</td>
<td>0002</td>
<td>0003</td>
<td>00A0</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>00</th>
<th>08</th>
<th>10</th>
<th>18</th>
<th>20</th>
<th>28</th>
<th>E8</th>
<th>F0</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>9222 9120 1121 A120 1121 A121 7211 0000</td>
<td>0000 0001 0002 0003 0004 0005 0006 0007</td>
<td>0008 0009 000A 000B 000C 000D 000E 000F</td>
<td>0000 0000 0000 FF0D 0000</td>
<td>1234 5678 9ABC DEF0 0000 0000 0000 F00D 0000</td>
<td>0000 0000 EEEE 1111 EEEE 1111 0000 0000</td>
<td>B1B2 F1F5 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

A Note on Notation…

• Assembly syntax is somewhat arbitrary

• Equivalent “Add” Instructions
  • add r1, r2, r3
  • add r1 = r2, r3
  • r1 = r2 + r3
  • add r1 = r2 + r3
  • add $1, $2, $3
  • ...

• Equivalent “Store Word” Instructions
  • sw $1, 10($2)
  • M[r2 + 10] = r1
  • st.w M[r2 + 10] = r1
  • ...

Specific Instance: MIPS Instruction Set

- MIPS – SGI Workstations, Nintendo, Sony...

State:
- 32-bit addresses to memory (32-bit PC)
- 32 32-bit Registers
- A “word” is 32-bits on MIPS
- Register $0 ($zero) always has the value 0
- By convention, certain registers are used for certain things – more next time...
MIPS Usage in Course

- Used throughout book
- We will use it on homework and exams
- For clarity of lecture, MIPS not always used
- Refer to book for all instructions discussed
Some Arithmetic Instructions:

- **Add:**
  - Assembly Format: `add <dest>, <src1>, <src2>`
  - Example: `add $1, $2, $3`
  - Example Meaning: `r1 = r2 + r3`

- **Subtract:**
  - Same as add, except “sub” instead of “add”
Specific Instance: MIPS Instruction Set

Some Memory Instructions:

- **Load Word:**
  - Assembly Format: `lw <dest>, <offset immediate> (<src1>)`
  - Example: `lw $1, 100 ($2)`
  - Example Meaning: `r1 = M[r2 + 100]`

- **Store Word:**
  - Assembly Format: `sw <src1>, <offset immediate> (<src2>)`
  - Example: `sw $1, 100 ($2)`
  - Example Meaning: `M[r2 + 100] = r1`
Some Branch Instructions:

- **Branch Equal:**
  - Assembly Format: `beq <src1>, <src2>, <target immediate>`
  - Example: `beq $1, $2, 100`
  - Example Meaning: branch `r1 == r2, 100`  
    If r1 is equal to r2, PC = 100

- **Branch Not Equal:** Same except `beq` -> `bne`

- **Jump:**
  - Assembly Format: `j <target immediate>`
  - Example: `j 100`
  - Example Meaning: jump 100  
    PC = 100
How are MIPS Instructions Encoded?
### MIPS Encodings

**32-bits/Instruction**

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>I:</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>address / immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J:</td>
<td>op</td>
<td></td>
<td></td>
<td>target address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **op**: basic operation of the instruction (opcode)
- **rs**: first source operand register
- **rt**: second source operand register
- **rd**: destination operand register
- **shamt**: shift amount
- **funct**: selects the specific variant of the opcode (function code)
- **address**: offset for load/store instructions ($\pm 2^{15}$)
- **immediate**: constants for immediate instructions
MIPS Add Instruction Encoding

Add is an R inst

```
add $17, $18, $19
```

<table>
<thead>
<tr>
<th>R:</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

Diagram showing the encoding of the add instruction.
MIPS Add Instruction Encoding

Subtract instruction: `sub $17, $18, $19`

<table>
<thead>
<tr>
<th>R:</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>0</td>
<td>34</td>
</tr>
</tbody>
</table>

`sub` is an R instruction.
Add and Subtract

A little foreshadowing…

add

sub
Memory Addressing

View memory as a single-dimensional array

Since 1980: Elements of array are 8-bits

We say “byte addressable”

Assuming 32-bit words:
1. How are bytes laid out in word read?

2. Can a word start at any address?
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

... Registers hold 32 bits of data

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
  
  i.e., what are the least 2 significant bits of a word address?
# Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>R2 ← R2−d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2+R3*]</td>
</tr>
</tbody>
</table>
Hello World

The Hello World Algorithm:
1. Emit “Hello World”
2. Terminate

```c
/*
 * Good programs have meaningful comments
 */
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```
Hello World

/*
 * Good programs have meaningful comments
 */
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}

C Program

GNU C Compiler

IA-64 Assembly Language

/file "hello.c"
/pred.safe_across_calls p1-p5,p16-p63
/.section .rodata.str1.8,"ams",@progbits,1
/align 8
/LC0:
/stringz "Hello world!\n"
/text
/align 16
/global main#
/proc main#
/main:
/prologue 12, 33
/save ar.pfs, r34
/alloc r34 = ar.pfs, 0, 3, 1, 0
/addl r35 = @ltoff(.LC0), gp
/save rp, r33
/mov r33 = b0

::
/body
/ld8 r35 = [r35]
/br.call.sptk.many b0 = printf#

::
/mov r8 = r0
/mov ar.pfs = r34
/mov b0 = r33
/br.ret.sptk.many b0
/endp main#
/ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.2 2.96-112.7.2)"
Hello World

IA-64 Assembly Language

```assembly
.file  "hello.c"
.pred.safe_across_calls p1-p5,p16-p63
.section .rodata.str1.8,"ams",@progbits,1
.align 8
.LC0:
.stringz "Hello World!\n"
.text
.align 16
.global main#
.proc main#
main:
  .prologue 12, 33
  .save ar.pfs, r34
  alloc r34 = ar.pfs, 0, 3, 1, 0
  addl r35 = @ltoff(.LC0), gp
  .save rp, r33
  mov r33 = b0
  ;
  .body
  ld8 r35 = [r35]
  br.call.sptk.many b0 = printf#

  mov r8 = r0
  mov ar.pfs = r34
  mov b0 = r33
  br.ret.sptk.many b0
.endp main#
.ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.2 2.96-112.7.2)"
```
Interfaces in Computer Systems

Software

- Applications
- Operating System
- Compiler
- Firmware

Hardware

- Instruction Set Processor
- I/O System
- Datapath & Control
- Digital Design
- Circuit Design
- Layout
Control
(from the back of a napkin)
The Hardware/Software Interface

Software

Applications

Operating System

Compiler

Firmware

Instruction Set Architecture

Instruction Set Processor

I/O System

Datapath & Control

Digital Design

Circuit Design

Layout

Hardware
The Instruction Set Architecture

“The vocabulary of commands”

- Defined by the Architecture (x86)
- Implemented by the Machine (Pentium 4, 3.06 GHz)
- An Abstraction Layer: The Hardware/Software Interface
- Architecture has longevity over implementation
- Example:

  add r1 = r2 + r3    (assembly)

  001 001 010 011    (binary)

  Opcode (verb)    Operands (nouns)
Some Figure and Text Acknowledgements

- Dan Connors
- David Patterson
- The COS126 Team (Wayne, Sedgewick)
- A. Mason
- Intel Corporation
- Amazon.com
- www.uiuc.edu