Spring 2017  
EE 3613: Computer Organization  
Chapter 2: Instruction Set Architecture  
– 1/4

Avinash Kodi  
Department of Electrical Engineering & Computer Science  
Ohio University, Athens, Ohio 45701  
E-mail: kodi@ohio.edu  
Website:  
http://oucsace.cs.ohiou.edu/~avinashk/classes/ee461a/ee461a.htm

Course Administration

- Lecture Notes 1 (Introduction), 2 (Performance) and 3 (Instruction Set Architecture) posted

- Homework #1 posted, due next Friday 1/27
(Von Neumann) Processor Organization

- **Control** needs to
  - Input instructions from memory
  - Issue signals to control the information flow between datapath components and to control what operations they perform
  - Control instruction sequencing

- **Datapath** needs to have the
  - Components – the functional units and storage needed to execute instructions
  - Interconnects – components connected so that instructions can be routed, and data loaded from and stored into memory

Where do ISA fit in a computing system?

- **Application Software**
- **Compiler**
- **Architecture** – (Instruction Set Architecture)
  - Platform Specific
  - a limited number of assembly language commands understood by the hardware (ADD, LOAD, etc)
- **Microarchitecture** (Hardware Implementation of ISA)
  - Pentium IV implements x86 ISA
  - Motorola G4 implements the Power PC ISA

- **Software**
- **Hardware**
- **Circuits**
- **Devices**
Instruction Set Design (1/2)

- What instructions should be included?
  - Add, Multiply, Divide, Sqrt [functions]
  - Branch [flow control]
  - Load/store [storage management]

- What storage locations?
  - How many registers?
  - How much memory?

- How should instructions be formatted?
  - 0, 1, 2 or more operands
  - Immediate operands

Instruction Set Design (2/2)

- How to encode instructions?
  - **RISC** (Reduced Instruction Set Computer)
    - All instructions are the same length (Eg: MIPS, PowerPC, Sun UltraSparc, XAP Processor, ARM processor)

  - **CISC** (Complex Instruction Set Computer)
    - Instructions can vary in size (Eg: VAX, Intel x86)

- What instructions can access memory?
  - For MIPS, only load/store can access memory (load-store architecture)
  - We will be working with MIPS architecture set
Software Program to Machine Code

C Program

```
main()
{
    int a, *b, c;
    c = a + b;
}
```

Assembly code

```
.text
.global main
lw $s1, 100($0)
add $s1, $s2, $s3
```

Machine code

```
0x7456
0xA16B
```

Unsigned Binary Representation

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0...0000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000001</td>
<td>0...0001</td>
<td>1</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0...0010</td>
<td>2</td>
</tr>
<tr>
<td>0x00000003</td>
<td>0...0011</td>
<td>3</td>
</tr>
<tr>
<td>0x00000004</td>
<td>0...0100</td>
<td>4</td>
</tr>
<tr>
<td>0x00000005</td>
<td>0...0101</td>
<td>5</td>
</tr>
<tr>
<td>0x00000006</td>
<td>0...0110</td>
<td>6</td>
</tr>
<tr>
<td>0x00000007</td>
<td>0...0111</td>
<td>7</td>
</tr>
<tr>
<td>0x00000008</td>
<td>0...1000</td>
<td>8</td>
</tr>
<tr>
<td>0x00000009</td>
<td>0...1001</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFFFFFFFC</td>
<td>1...1100</td>
<td>$2^{32} - 4$</td>
</tr>
<tr>
<td>0xFFFFFFFFD</td>
<td>1...1101</td>
<td>$2^{32} - 3$</td>
</tr>
<tr>
<td>0xFFFFFFFFE</td>
<td>1...1110</td>
<td>$2^{32} - 2$</td>
</tr>
<tr>
<td>0xFFFFFFFFF</td>
<td>1...1111</td>
<td>$2^{32} - 1$</td>
</tr>
</tbody>
</table>

2^31 2^30 2^29 2^28 2^27 2^26 2^25 2^24 bit weight
31 30 29 28 27 26 25 24 bit position

0...0...0...0...0...0...0...0...1 bit

- $2^{32} - 1$
**ASCII: Beyond Numbers**

- **American Std Code for Info Interchange (ASCII):** 8-bit bytes representing characters

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Null</td>
<td>32</td>
<td>space</td>
<td>48</td>
<td>0</td>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>33</td>
<td>49</td>
<td>!</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>50</td>
<td>*</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>35</td>
<td>51</td>
<td>#</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>37</td>
<td>53</td>
<td>%</td>
<td>5</td>
<td>69</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>39</td>
<td>55</td>
<td>'</td>
<td>7</td>
<td>71</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>bksp</td>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>tab</td>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>42</td>
<td>'</td>
<td>58</td>
<td>:</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>+</td>
<td>59</td>
<td>;</td>
<td>75</td>
<td>K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>44</td>
<td>.</td>
<td>60</td>
<td>&lt;</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>111</td>
<td>o</td>
<td>127</td>
<td>DEL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Architecture Specification**

- **Data Types**
  - Bit, byte, signed/unsigned, logical, floating point, character
- **Operations**
  - Data movement, arithmetic, shift/rotate, conversion, input/output, control, system calls
- **# of operands**
  - 3, 2, 1, or 0 operands
- ** Registers**
  - Integer, floating point, control
- **Storage for Operands**
  - Registers, memory locations, stack locations, fixed registers, fixed location
Assembly Code

- a.k.a. Register-transfer-language (RTL)
- Fields
  - **Opcode** – what instruction to perform
  - **Source** – input operand specifiers
  - **Destination** – output operand specifiers
    - What data to perform operation on

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>$t0</td>
<td>$s1</td>
<td>$s2</td>
</tr>
</tbody>
</table>

- Translation – value of $s1 added to $s2 put in $t0

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement
  - `add $t0, $s1, $s2`
  - `sub $t0, $s1, $s2`
- Each arithmetic statement performs only **one** operation
- Each arithmetic instruction fits in **32 bits** and specifies exactly **three** operands
- Those operands are all contained in the datapath’s register file ($t0, $s1, $s2) – indicated by $`
- Operand order is fixed (**destination first**)

**Design Principle 1: Simplicity favors regularity**
Machine Language – Add Instruction

- Instructions like registers and words of data are all 32 bits long

- Arithmetic instruction Format (R Format): 
  \[ \text{add } \$t0, \$s1, \$s2 \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits – opcode that specifies the instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 bits – register file address of the first source operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 bits – register file address of the second operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 bits – register file address of the result's destination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 bits – shift amount (for shift instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits – function code augmenting the opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly Code Example

- What are the contents of the registers after executing the given assembly code?

Program:
\[
\begin{align*}
\text{add } \$s1, \$s2, \$s3 \\
\text{multi } \$s3, \$s3, 3 \\
\text{sub } \$s2, \$s3, \$s2
\end{align*}
\]

Initial Register File:

<table>
<thead>
<tr>
<th>$s1</th>
<th>$s2</th>
<th>$s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>-4</td>
<td>57</td>
</tr>
</tbody>
</table>

After executing:

add \$s1, \$s2, \$s3  
multi \$s3, \$s3, 3  
sub \$s2, \$s3, \$s2
Assembly Instruction Encoding

- Since EDSAC (1949), almost all computers stored program instructions the same way as they store data
- Each instruction is encoded as a number

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$s1$</th>
<th>$s2$</th>
<th>$t0$</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00000</td>
<td>10001</td>
<td>1010</td>
<td>0100</td>
<td>0000</td>
</tr>
</tbody>
</table>

- $m$ bits can encode $2^m$ different values
- $n$ values can be encoded in $\lceil \log_2(n) \rceil$ bits
- For above example, we can have __ opcodes and __ registers
- And for the above example, the code is ________ in hexadecimals

Storage Architecture

- Registers
  - Fast and small (and useful)

- Immediate values
  - Specifying constants in instructions

- Memory
  - Big and complex (and useful)
MIPS Register Storage

- Holds **thirty-two** 32-bit registers
  - 2 read ports, 1 write port

- Registers are
  - **Faster** than main memory
    - But register files with more locations are slower
    - Read/write port increase impacts speed quadratically

  **Design Principle 2: Smaller is faster**
  - Easier for a compiler to use
    - Eg: \((A - B) - (C \times D) - (E \times F)\) can do multiplies in any order
  - Can hold variables so that
    - Code density improves (since registers are named with fewer bits than a memory location

Immediate Values

- Small constant values placed in instructions

- They are stored in memory only because all instructions are in memory (traditionally, not in MIPS)

- In MIPS, constants are built into the instruction having a single operand
  - Example: `ptr++;` → `addi $s1, $s1, #4`
  - Useful for branch instructions
    - target address is often immediate in the instruction

  **Design Principle 3: Make the common case fast**

- Size of the immediate is usually determined by how many bits are left in the instruction format
Memory Storage

- Large array of storage accessed using memory addresses

  - A machine with a 32 bit addresses can reference memory locations starting from 0 to $2^{32} - 1$ (or 4,294,967,295)
  - A machine with a 64 bit address can reference memory locations starting from 0 to $2^{64} - 1$ (or 18,446,744,073,709,551,615)

- Lots of different ways to calculate the addresses

Memory Architecture: The MIPS Memory Image

- Stack
  - Activation Records: local variables, parameters, etc
- Heap Data
  - Dynamically allocated data (new or malloc() )
- Static Data
  - Global data and static local data
- Text
  - Machine code instructions (and some constants)
- Reserved
  - Reserved for operating system
MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions to access memory:

  lw  $t0, 4($s3)  # load word from memory
  sw  $t0, 8($s3)  # store word to memory

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5-bit address.

- The memory address – a 32-bit address – is formed by adding the contents of the base address register to the offset value:
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of address in the base register.
  - Note that offset can be positive or negative.

Machine Language – Load Instruction

- Load/Store Instruction Format (I format): lw  $t0, 24($s2)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  24_{10} + $s2 =

  ...0001 1000
  + ...1001 0100
  ...1010 1100 =
  0x120040ac

  $s2  0x12004094

  $t0  0x120040ac

  Memory

  0xffffffff
  0x120040ac
  0x12004094
  0x0000000c
  0x00000008
  0x00000004
  0x00000000

  data  word address (hex)