MIPS Instruction Set Architecture

(Second Edition: Chapter 3
Fourth Edition: Chapter 2)
from Dr. Andrea Di Blas’ notes
Instruction Set Architectures

• What is the ISA

• Types of ISA
  - Accumulator Architecture
  - General Purpose Architecture
    • Memory Operands
    • Register Operands (Load/Store)

• MIPS ISA
Instruction Set Architecture

• Definition 1: The interface between a computer’s software and its hardware.
• Definition 2: A computer’s Assembly Language
• Advantage: Allows different computer types (with the same ISA) to run identical software.
ISA - Specifics

ISA is all of the *programmer-visible* components and operations of the computer.

- memory organization
  - address space - how many locations can be addressed?
  - addressability - how many bits per location?
- register set
  - how many? what size? how are they used?
- instruction set
  - opcodes
  - data types
  - addressing modes

The ISA provides all the information needed for someone to write a program in machine language (or translate from a high-level language to machine language).
Basic ISA Types - based on Operands

• A, B, C are operands in A+B = C
• Stack Architecture - no explicit operands
• Accumulator Architecture - one explicit operand
• General Purpose Register Architectures: 3 explicit operands
  - Memory-Memory
  - Register-Memory
  - Register-Register
Stack Architecture

0 explicit operands. All operands on Stack.

A = B * (C + D * B)

1. Push B
2. Push D
3. *
4. Push C
5. +
6. Push B
7. *
8. Pop A

<table>
<thead>
<tr>
<th></th>
<th>0.</th>
<th>2.</th>
<th>4.</th>
<th>8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>V3</td>
<td>V3</td>
<td>V3</td>
<td>V3</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Accumulator Architecture

One explicit operand per instruction. Other two are (1 source, 1 destination) are in the Accumulator.

\[ A = B \times (C + D \times B) \]

1. Load B
2. Mult D
3. Add C
4. Mult B
5. Store A
GPR Architecture: Register-Memory

Two Operands: a Register (for one source and destination) and a memory location.

\[ A = B \times (C + D \times B) \]

1. Load R0, B
2. Mult R0, D
3. Add R0, C
4. Mult R0, B
5. Store R0, A
GPR Architecture: Memory-Memory

Three Operands, each can be from a register or from memory.

\[ A = B \times (C + D \times B) \]

1. \texttt{Mult} RO, D, B
2. \texttt{Add} RO, RO, C
3. \texttt{Mult} A, B, RO
GPR Architecture: Register - Register (Load/Store)

Three Operands, each must be from a register.
Load and Store to move data memory↔registers

\[
A = B \times (C + D \times B)
\]

1. Load R0, B
2. Load R1, D
3. Mult R2, R0, R1
4. Load R3, C
5. Add R3, R3, R2
6. Mult R4, R0, R0
7. Store R4, A
Comparison of GPR Arch.

• Memory-Memory (DEC VAX)
  - Fewest instructions
  - Most memory accesses (up to 3 data accesses per instruction)

• Register-Memory (PDP-11, IBM 360, i80x86)
  - 1-2 data memory accesses on average
  - Smaller code size
  - Variable instruction length

• Load/Store Register-Register
  - Fewest data memory accesses
  - Simple code generation
  - Simpler to Pipeline!
Advantages for each ISA type

- **Stack Arch.**
  - No need to have explicit operands. (ALU instructions simpler)

- **Memory-Memory GPU Arch.**
  - Very flexible, fewer instructions to execute program.

- **Accumulator and Reg-Mem GPU Arch.**
  - Only one explicit operand. Fewer data memory accesses

- **Load/Store (Reg-Reg GPU) Arch.**
  - Flexible once in registers, fewer data memory accesses, designed for pipelining.
RISC type of Register-Register GPU

Reduced Instruction Set Computer (RISC) research
• Many complex instructions were seldom, if ever, used in compiled code
• In some cases more, simpler instructions were faster than fewer, complicated instructions.
• Microinstruction pipelining techniques could be applied to processors if instructions were changed.
RISC design principles

• Make the common case fast.
• Smaller is faster
  - Design easier to optimize
  - Distance increases propagation delay.
• If it is expensive to implement in hardware, maybe it should be implemented in software instead.
• Simplicity favors regularity
RISC architecture characteristics

- Fixed size instructions (for easy pipelining and decoding)
- Memory transactions allowed only in separate Load and Store instructions (for easier pipelining)
- Few and simple addressing modes
- Large orthogonal register sets (no/few special registers)
- No/few complicated instructions
Microprocessor without Interlocked Pipeline Stages (MIPS)

- Company founded by Hennessey and others in 1984 to build a commercial RISC processor.
- Bought by Silicon Graphics and is now MIPS Technologies
- Processors used by Sony, Nintendo, etc.
Units of Data

- **BIT**: one binary digit (0 or 1)
- **BYTE**: an 8-bit binary string
- **NIBBLE or NYBBLE**: a 4-bit binary string
- **WORD**: a string of n bits, where n is ISA dependent
- **DOUBLEWORD**: an ordered set of 2 words
- **QUADWORD**: an ordered set of 4 words
Our MIPS

• 32 bit architecture: word length, address length.
• 32 General Purpose Registers: very generous in 1984.
• Instruction set function:
  - Computational (uses ALU): and, or, add, etc.
  - Memory Access: lw, lb, sw, sb
  - Program flow
• Instruction set format:
  - R-type
  - I-type
  - J-type
R-type (register) Format

- **op** (opcode): basic operation of instruction - also determines format - op = 0 for all R-type instructions
- **rs**: first source operand
- **rt**: second source operand
- **rd**: destination
- **shamt**: shift amount
- **funct**: function variant (e.g. add and sub same op, but add has funct=32 and sub has funct=34

<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>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

- 6 bits
- 5 bits
- 6 bits
### R-type Instructions

**Examples:**

- **add** \( Rd, Rs, Rt \) \( \text{; } Rd = Rs + Rt \)
- **sub** \( Rd, Rs, Rt \)
- **or** \( Rd, Rs, Rt \)

#### add $3, $17, $3

<table>
<thead>
<tr>
<th>MEANING</th>
<th>comp</th>
<th>$17</th>
<th>$18</th>
<th>$3</th>
<th>-</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>0</td>
<td>17</td>
<td>18</td>
<td>3</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Binary</td>
<td>00 0000</td>
<td>1 0001</td>
<td>1 0010</td>
<td>0 0011</td>
<td>0 0000</td>
<td>10 0000</td>
</tr>
<tr>
<td>Hex</td>
<td>00</td>
<td>11</td>
<td>12</td>
<td>03</td>
<td>00</td>
<td>20</td>
</tr>
</tbody>
</table>
I-type (immediate) Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset/immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21 20 16 15</td>
</tr>
</tbody>
</table>

| 6 bits | 5 bits | 5 bits | 16 bits |

- **op (opcode):** basic operation of instruction (e.g. `lw` opcode = 35, `addi` opcode = 8, `beq` opcode = 4)
- **rs/base:** register containing source operand or base
- **rt:** destination register for `addi` or loads, source register for stores, second operand for `beq`
- **offset/immediate:** immediate field in computation instructions, byte address offset (wrt rs) in load/store instructions, word address offset (wrt PC) in branch instructions - always sign extended to a 32-bit value.
I-type Instructions

examples:

- **addi** Rd,Rs,N ; Rd = Rs + SignExt(N)
- **ori** Rd,Rs,N ; Rd = Rs | SignExt(N)
- **beq** Rs,Rt,Label ; If(Rs == Rt) goto Label
- **lw** Rt,N(Rs) ; Rt = Mem[Rs + SignExt(N)]
- **sw** Rt,N(Rs) ; Mem[Rs + SignExt(N)] = Rt

**addi** $8,$22,-16

<table>
<thead>
<tr>
<th>MEANING</th>
<th>addi</th>
<th>$22</th>
<th>$8</th>
<th>-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>9</td>
<td>22</td>
<td>8</td>
<td>-16</td>
</tr>
<tr>
<td>Binary</td>
<td>001001</td>
<td>10110</td>
<td>01000</td>
<td>1111 1111 1111 0000</td>
</tr>
<tr>
<td>Hex</td>
<td>09</td>
<td>16</td>
<td>08</td>
<td>FFF0</td>
</tr>
</tbody>
</table>
J-type (jump) Format

- **op (opcode)**: basic operation of instruction (e.g. j opcode = 2)
- **target**: target word address of the instruction to jump to.
J-type Instructions

examples:

\texttt{j Label} ; \text{goto Label}
\texttt{Jal Label} ; \$31 = PC+4, goto Label

\begin{tabular}{|l|l|l|}
\hline
\textbf{MEANING} & \textbf{j} & \textbf{Label} \\
\hline
Decimal & 2 & \\
\hline
Binary & 00 0010 & (part of) Label’s address \\
\hline
Hex & 2 & \\
\hline
\end{tabular}
R-, I, and J-type format comparison

R-type:

<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>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
</tbody>
</table>

I-type:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset/immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
</tbody>
</table>

J-type:

<table>
<thead>
<tr>
<th>op</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
</tr>
</tbody>
</table>