Lecture 5: MIPS Instruction Set

Jeremy R. Johnson
Anatole D. Ruslanov
William M. Mongan

Some or all figures from Computer Organization and Design: The Hardware/Software Approach, Third Edition, by David Patterson and John Hennessy, are copyrighted material (COPYRIGHT 2004 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED).
Introduction

• Objective: To introduce the MIPS instruction set and to show how MIPS instructions are represented in the computer.

• The stored-program concept:
  – Instructions are represented as numbers.
  – Programs can be stored in memory to be read or written just like data.
Stored Program Concept

- Instructions are just a sequence of 32 bits
- Programs are stored in memory — to be read or written just like data

- Fetch & Execute Cycle
  - Instruction is fetched and put into a special register
  - Bits in the instruction register determine the subsequent actions
  - When done, fetch the next instruction and continue execution
Instructions

• Language of the machine
• More primitive than higher level languages (e.g. C, C++, Java)
  – e.g. no sophisticated control flow, primitive data structures
• MIPS – ISA developed in the early 80’s (RISC)
  – Similar to other RISC architectures developed since the 1980's
  – Almost 100 million MIPS processors manufactured in 2002
  – Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...
  – Regular (32 bit instructions, small number of different formats)
  – Relatively small number of instructions
  – Register architecture (all instructions operate on registers)
  – Load/Store architecture (memory accessed only with load/store instructions, with few addressing modes)

• Design goals: maximize performance and minimize cost, reduce processor design time
The MIPS Processor is popular
Gadgets shipped with MIPS cores

• Cable Modems 94%
• DSL Modems 40%
• VDSL Modems 93%
• IDTV 40%
• Cable STBs 76%
• DVD Recorder 75%
• Game Consoles 76%
• Office Automation 48%
• Color Laser Printers 62%
• Commercial Color Copiers 73%

MIPS Arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

- Example

  C code: \[ A = B + C \]

  MIPS code: `add $s0, $s1, $s2`

  (associated with variables by compiler)

- Using the natural number of operands for an operation (e.g. addition) conforms to the design principle of keeping the hardware simple.
Temporary Variables

- Regularity of instruction format requires that expressions get mapped to a sequence of binary operations with temporary results being stored in temporary variables.

- Example

  C code: \[ f = (g + h) - (i + j); \]

  Assume \( f, g, h, i, j \) are in \$s0 through \$s4 respectively

  MIPS code:
  
  ```
  add $t0, $s1, $s2 # $t0 = g+h
  add $t1, $s3, $s4 # $t1 = i+j
  sub $s0, $t0, $t1 # f = $t0 - $t1
  ```
Registers vs. Memory

- Operands for arithmetic instructions must be registers, — only 32 integer registers are available

- Compiler associates variables with registers

- When too many variable are used to fit in 32 registers, the compiler must allocate temporary space in memory and then load and store temporary results to/from memory.

  - The compiler tries to put most frequently occurring variables in registers.
  - The extra temporary variables must be “spilled” to memory.
Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data

- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory

- Memory is byte addressed
  - Each address identifies an 8-bit byte

- Words are aligned in memory
  - Address must be a multiple of 4

- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - c.f. Little Endian: least-significant byte at least address
Memory Organization

• Viewed as a large, single-dimension array, where a memory address is an index into the array
• MIPS systems address memory in byte chunks
• The memory address (= index) points to a byte in memory.
• This is called "Byte addressing"
Memory Organization

- Most data items are grouped into words
- A MIPS word is 4 bytes or 32 bits

<table>
<thead>
<tr>
<th></th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<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>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Registers also 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 (alignment restriction)
- Bytes can be accessed from left to right (big endian) or right to left (little endian).
Load and Store

• All arithmetic instructions operate on registers
• Memory is accessed through load and store instructions

• An example C code: \( A[12] = h + A[8]; \)

Assume that \( s3 \) contains the base address of \( A \)

MIPS code:

\[
\begin{align*}
\text{lw} & \quad \text{\$t0, 32 (\$s3)} \\
\text{add} & \quad \text{\$t0, \$t0, \$s2} \\
\text{sw} & \quad \text{\$t0, 48 (\$s3)}
\end{align*}
\]

• Note: \text{sw} (store word instruction) has destination last.
• Note: remember arithmetic operands are registers, not memory!

This is invalid: \( \text{add 48 (\$s3), \$s2, 32 (\$s3)} \)
Our First Example

• Can we figure out the code?

```c
swap(int v[], int k)
{ int temp;
 temp = v[k]
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

```assembly
sw $15, 0($2)
sw $16, 4($2)
jr $31
```
The Constant Zero

- **MIPS register 0 ($zero) is the constant 0**
  - Cannot be overwritten
- **Useful for common operations**
  - E.g., move between registers

\[
\text{add } \$t2, \$s1, \$zero
\]
### Logical Operations

- **Instructions for bitwise manipulation**

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
Shift Operations

<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</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **shamt**: how many positions to shift
- **Shift left logical**
  - **Shift left** and fill with 0 bits
  - $\text{sll}$ by $i$ bits multiplies by $2^i$
- **Shift right logical**
  - **Shift right** and fill with 0 bits
  - $\text{srl}$ by $i$ bits divides by $2^i$ (unsigned only)
AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```plaintext
and $t0, $t1, $t2
```

```
$t2  0000 0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0000 1100 0000 0000
```
OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

or $t0, $t1, $t2

| $t2  | 0000 0000 0000 0000 0000 0000 1101 1100 0000 |
| $t1  | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0  | 0000 0000 0000 0000 0011 1101 1100 0000 |
NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0

- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \text{ NOT } ( a \text{ OR } b ) \)

```plaintext
nor $t0, $t1, $zero
```

\[
\begin{align*}
$t1 &= \begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \\
$t0 &= \begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\end{align*}
\]

Register 0: always read as zero
So far we’ve learned:

• **MIPS**
  — loading words but addressing bytes
  — arithmetic on registers only

• **Instruction**  
  **Meaning**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>
Machine Language

• Instructions, registers and data words are 32 bit long
  – Example: add $t1, $s1, $s2
  – Registers have numbers/indexes: $t1=9, $s1=17, $s2=18

• Instruction Format:

<table>
<thead>
<tr>
<th>000000</th>
<th>10001</th>
<th>10010</th>
<th>01001</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

• Guess what do the field names stand for?
Machine Language

• Consider the load-word and store-word instructions,
  − What would the regularity principle have us do?
  − New principle: Good design demands a compromise

• Introduce a new type of instruction format
  − I-format type for data transfer instructions
  − The other format was R-type for register (add and sub)

• Example: `lw $t0, 32($s2)`

<table>
<thead>
<tr>
<th>35</th>
<th>18</th>
<th>8</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit number</td>
</tr>
</tbody>
</table>

• Where's the compromise?
Addressing in Jumps

- **J format format (jump format – j, jal)**
  
<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>26-bits</td>
</tr>
</tbody>
</table>

- **Example: j 10000**

<table>
<thead>
<tr>
<th>2</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>26-bits</td>
</tr>
</tbody>
</table>
Target Addressing Example

- Loop code example
  - Assume Loop at location 80000

Loop:  
- `sll $t1, $s3, 2` at 80000
- `add $t1, $t1, $s6` at 80004
- `lw $t0, 0($t1)` at 80008
- `bne $t0, $s5, Exit` at 80012
- `addi $s3, $s3, 1` at 80016
- `j Loop` at 80020

Exit: ... at 80024
No-Op Instructions

• What would you expect a no-op instruction to be in binary?
• What is this in assembly?
Design Principles

- Simplicity favors regularity
  - All instructions 32 bits
  - All instructions have 3 operands
- Smaller is faster
  - Only 32 registers
- Good design demands good compromises
  - All instructions are the same length
  - Limited number of instruction formats: R, I, J
- Make common cases fast
  - 16-bit immediate constant
  - Only two branch instructions