Introduction to SMIPS

Li-Shiuan Peh
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

February 22, 2012

Instructions are bits (i.e., as numbers)
Programs are stored in memory
- to be read or written just like data

Stored Program Concept

- Instructions are fetched and put into a special register
- Bits in the register "control" the subsequent actions
- Fetch the next instruction and continue

memory for data, programs, compilers, editors, etc.
### Multiple Levels of Representation

<table>
<thead>
<tr>
<th>High Level Language Program</th>
<th>Assembly Language Program</th>
<th>Machine Language Program</th>
<th>Machine Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Assembler</td>
<td></td>
<td>Control Signal Specification</td>
</tr>
</tbody>
</table>

- `temp = v[k];`
- `v[k] = v[k+1];`
- `v[k+1] = temp;`
- `lw $15, 0($2)`
- `lw $16, 4($2)`
- `sw$16, 0($2)`
- `sw$15, 4($2)`

### Instruction Set Architecture (ISA)

- **Programmer’s view of the computer**
  - Instructions, operands

---

February 22, 2012

http://csg.csail.mit.edu/6.5078

L5-4
Example ISAs

- Intel 80x86
- ARM
- IBM/Motorola PowerPC
- HP PA-RISC
- Oracle/Sun Sparc
- 6.004’s Beta

Why MIPS? It’s simple! Most taught ISA

World’s First Android 4.0 (Ice Cream Sandwich) Tablet available in the market—Yes it’s MIPS!

We’re all around you.

You just don’t know it.

The #1 processor in digital TVs
MIPS I Instruction Set Architecture

Instruction Categories
- Load/Store
- Computational
- Jump and Branch
- Floating Point
- Memory Management
- Special

Register:
- r0 - r31
- PC
- HI
- LO

3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td>jump target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SMIPS: a subset of the full MIPS32 ISA

SMIPS Registers: Fast Locations for Data

- 32 32-bit registers: $0, $1, ... , $31
  - operands for integer arithmetic
  - address calculations
  - temporary locations
  - special-purpose functions defined by convention
- 1 32-bit Program Counter (PC)
- 2 32-bit registers HI & LO:
  - used for multiply & divide
MIPS arithmetic

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

Example:

C code:        MIPS code:
A = B + C     add $s0, $s1, $s2

C code:        MIPS code:
A = B + C + D;
E = F - A;
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0

MIPS Load-Store Architecture

- Every operand must be in a register (a few exceptions)
- Variables have to be loaded in registers.
- Results have to be stored in memory.

\[ a = b + c \]
load \( b \) in register \( Rx \)
load \( c \) in register \( Ry \)

\[ d = a + b \]
\[ Rz = Rx + Ry \]
store \( Rz \) in \( a \)

\[ Rt = Rz + Rx \]
store \( Rt \) in \( d \)

more variables than registers, so need explicit load and stores.
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

Q: How to specify a memory location?

Load & Store Instructions

- Base+Offset addressing mode: offset (base register)
  - e.g., 32($s3)
- Example:
  - \( A \): an array of 100 words
  - the base address of the array \( A \) is in \( $s3 \)
  - base register: \( $s3 \)
  - offset: 32

- MIPS code:
  - \( \text{lw } $t0, 32($s3) \)
  - \( \text{add } $t0, $s2, $t0 \)
  - \( \text{sw } $t0, 32($s3) \)

- Store word has destination last
Example: Compiling using a Variable Array Index

- **C code:**
  
  \[ g = h + A[i] \]

  $s3$: base register for A

  g, h, i: $s1, $s2, $s4

- **MIPS code:**

  ```mips
  add $t1, $s4, $s4           # $t1 = 2 * i
  add $t1, $t1, $t1             # $t1 = 4 * i
  add $t1, $t1, $s3             # $t1 = address of A[i]
  lw $t0, 0($t1)                    # $t0 = A[i]
  add $s1, $s2, $t0             # g = h + A[i]
  ```

LUI instruction

- Load upper immediate
- LUI rt, zero-ext-imm
- Shifts 16-bit immediate into high-order 16 bits, with 16 zeros in low order bits -> rt

- How is LUI useful?
Control: bne & beq

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:
  
  \[
  \begin{align*}
  \text{bne } &\ t0, \ t1, \text{Label} \\
  \text{beq } &\ t0, \ t1, \text{Label}
  \end{align*}
  \]

- Example: if (i==j) h = i + j;
  
  \[
  \begin{align*}
  \text{bne } &\ s0, \ s1, \text{Label} \\
  \text{add } &\ s3, \ s0, \ s1 \\
  \text{Label:} &\ \text{....}
  \end{align*}
  \]

SLT instructions

- Set if less than
- \textit{E.g.} \texttt{SLTI rt, rs, signed-imm}
  
  \text{If } (rs < \text{imm}), \text{ rt}=1, \text{ else rt}=0

What is SLT used for?
MIPS unconditional branch instructions:
\[ j \ label \]

- Example:

```plaintext
if (i!=j)  
  beq $s4, $s5, Lab1
  h=i+j;   
  add $s3, $s4, $s5
else      
  j Lab2
  h=i-j;   
  Lab1:    
  sub $s3, $s4, $s5
  Lab2:    ...
```

- \( PC <- PC + 4 + 4\times SE\text{EXT}(\text{literal}) \)

---

Jumps

JAL target (jump and link)

- PC+8 -> R31
- Why PC+8?
- How is JAL useful?

February 22, 2012
http://csg.csail.mit.edu/6.5078 L5-18
Jumps

- **JR rs**
  - Jumps to address in register

- **PC <- Reg[rs]**

**JR vs. J or JAL?**

---

Jumps

- **JALR – Jump and link register**

- **JALR rd, rs**
  - Jumps to rs
  - Writes link address into rd

**Why JALR vs. JAL?**
MIPS: Stack detective!

- Call procedure: jump and link (jal)
- Return from procedure: jump register (jr)
- Argument values: $a0 - $a3
- Return value: $v0

Template:
- Call setup
- Prologue
- Epilogue
- Return cleanup
Procedure call setup

1. Place current parameters into stack (space already allocated by caller of this procedure)
2. Save any TEMPORARY registers that need to be preserved across the procedure call
3. Place first 4 parameters to procedure into $a0-$a3
4. Place remainder of parameters to procedure into allocated space within the stack frame
**Procedure call setup**

```
# call setup for call to procB
# save current (live) parameters into the space specifically
# allocated for this purpose within caller's stack frame
sw $a0, 64($sp)       # only needed if values are 'live'
sw $a1, 68($sp)       # only need if values are 'live'
# save any registers that need to be preserved across the call
sw $t0, 36($sp)       # only need if values are 'live'
sw $t1, 40($sp)       # only need if values are 'live'
# put parameters into proper location
lw $a0, 44($sp)       # load R into $a0
lw $a1, 48($sp)       # load S into $a1
lw $a2, 52($sp)       # load T into $a2
lw $a3, 56($sp)       # load U into $a3
lw $t0, 60($sp)       # load V into a temp register
sw $t0, 20($sp)       # outgoing arg# must go on the stack
# end call setup
```
Time to actually call function! 😊

```assembly
# procedure call
jal procB
```

---

Return cleanup

1. copy needed return values and parameters from $v0-v1, $a0-a3, or stack frame to correct places
2. restore any temporary registers from stack frame (saved in call setup)

```assembly
# return cleanup for call to procB
# restore saved registers
lw $a0, 64($sp)
lw $a1, 68($sp)
lw $t0, 36($sp)
lw $t1, 40($sp)
# return values are in $v0 and $v1
```
Epilogue

1. restore (copy) return address from stack frame into $ra
2. restore from stack frame any saved registers (saved in prologue)
3. de-allocate stack frame (move $sp so the space for the procedure's frame is gone)

```plaintext
# procedure epilogue
# restore return address
lw $ra, 32($sp)
# restore ss registers saved in prologue
lw $s0, 24($sp)
lw $s1, 28($sp)
# put return values in $v0 and $v1
mov $v0, $t0
# deallocate stack frame
add $sp, $sp, 60
# return
jr $ra
```

MIPS coprocessor 0 instructions: mfc0, mtc0

- interrupts, exceptions, resets
- Beta vs MIPS
Exception Registers

- Not part of the register file.
- **Cause**
  - Records the cause of the exception
- **EPC (Exception PC)**
  - Records the PC where the exception occurred
- **EPC and Cause**: part of Coprocessor 0
- Move from Coprocessor 0
  - `mfc0 $t0, EPC`
  - Moves the contents of EPC into $t0

Exceptions

- Save cause and exception PC
- Jump to exception handler (0x0000_1100)
- Exception handler:
  - Saves registers on stack
  - Reads the Cause register
    - `mfc0 Cause, $t0`
  - Handles the exception
  - Restores registers
  - Returns to program
    - `mfc0 EPC, $k0`
    - `jr $k0`
Exception Causes

<table>
<thead>
<tr>
<th>ExcCode</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Hint</td>
<td>External interrupt.</td>
</tr>
<tr>
<td>2</td>
<td>Tint</td>
<td>Timer interrupt.</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address or misalignment error on load.</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>Address or misalignment error on store.</td>
</tr>
<tr>
<td>6</td>
<td>AdEF</td>
<td>Address or misalignment error on fetch.</td>
</tr>
<tr>
<td>8</td>
<td>Sys</td>
<td>Syscall exception.</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>Breakpoint exception.</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>Reserved instruction exception.</td>
</tr>
<tr>
<td>11</td>
<td>CpU</td>
<td>Coprocessor Unusable.</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>Arithmetic Overflow.</td>
</tr>
</tbody>
</table>

Cause register

```
 31 30 29 28 27 16 15 8 7 6 2 10
  BD 0  CE 0  IP 0  ExcCode 0
     1 1  2 12 8 1  5 2
```
Reset

- mtc0 zero, $9 # init counter
- mtc0 zero, $11 # timer interrupt
- : :
- J kernel_init