C335
Computer Structures

MIPS Instructions (Part #6)

Dr. David R. Surma
Department of Computer and Information Sciences
Adapted from Morgan Kaufmann, Dr. L. Zhang and others
Programming Styles

- Procedures (subroutines, functions) allow the programmer to structure programs making them:
  - easier to understand and debug and
  - allowing code to be reused

- Procedures allow the programmer to concentrate on one portion of the code at a time:
  - parameters act as barriers between the procedure and the rest of the program and data, allowing the procedure to accept passed values (arguments) and to return values (results)
Six Steps in Execution of a Procedure

- Main routine (caller) places parameters in a place where the procedure (callee) can access them
  - $a0 - a3$: four argument registers
- Caller transfers control to the callee
- Callee acquires the storage resources needed
- Callee performs the desired task
- Callee places the result value in a place where the caller can access it
  - $v0 - v1$: two value registers for result values
- Callee returns control to the caller
  - $ra$: one return address register to return to the point of origin
Registers play a major role in keeping track of information for function calls.

- **Register conventions:**
  - Return address: $ra$
  - Arguments: $a0$, $a1$, $a2$, $a3$
  - Return value: $v0$, $v1$
  - Local variables: $s0$, $s1$, ..., $s7$

- The **stack** is also used; more later.
... sum(a,b);... /* a,b:$s0,$s1 */

C/C++

```c
int sum(int x, int y) {
    return x+y;
}
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
... sum(a,b); ... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}

address
1000 add $a0,$s0,$zero  # x = a; x in $a0
1004 add $a1,$s1,$zero  # y = b; y in $a1
1008 addi $ra,$zero,1016 # $ra=1016
1012 j sum  # jump to sum
1016 ...

2000 sum: add $v0,$a0,$a1
2004 jr $ra  # return to the caller
... sum(a,b); ... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}

Question: Why use \texttt{jr} here? Why not simply use \texttt{j}?

Answer: \texttt{sum} might be called by many places, so we can’t return to a fixed place. The calling proc to \texttt{sum} must be able to say “return here” somehow.

\texttt{2000 sum: add \$v0,\$a0,\$a1}
\texttt{2004 \textbf{jr} \$ra}  \# return to the caller
Instruction Support for Functions (4/5)

- Single instruction to jump and save return address: “jump and link” (jal)

- Before:

  1008  addi $ra,$zero,1016  # $ra = 1016
  1012  j sum                # goto sum

- After (using jal):

  1008  jal sum   # $ra = 1012, goto sum

- Why have a jal?
  - Make the common case fast: function calls are very common. (Also, you don’t have to know where the code is loaded into memory with jal.)
Instruction Support for Functions (5/5)

- Syntax for `jal` (jump and link) is same as for `j` (jump):
  
  `jal label`

- `jal should really be called laj` for “link and jump”:
  
  - Step 1 (link): Save address of next instruction into $ra (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label
Spilling Registers

What if the callee needs to use more registers than allocated to argument and return values?

- it uses a stack — a “last-in-first-out” data structure
  - One of the general registers, $sp ($29), is used to address the stack (which “grows” from high address to low address)
  - add data onto the stack — **push**
    \[ sp = sp - 4 \]
    store the data on stack at new $sp (a pre-decrement operation)
  - remove data from the stack — **pop**
    \[ sp = sp + 4 \]
    load the data from stack at $sp (a post-increment operation)
Compiling a C Leaf Procedure

- **Leaf** procedures are ones that do not call other procedures. Give the MIPS assembler code for

```c
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

where g, h, i, and j are in $a0, a1, a2, a3

```mips
label leaf_ex:    addi  $sp,$sp,-8  
                 sw     $s0,4($sp)  
                 sw     $s1,0($sp)  
                 add    $s0,$a0,$a1  
                 add    $s1,$a2,$a3  
                 sub    $s1,$s0,$s1  
                 add    $v0, $s1,$zero  
                 lw     $s1,0($sp)  
                 lw     $s0,4($sp)  
                 addi   $sp,$sp,8  
                 jr      $ra
```
Compiling a C Leaf Procedure

- **Key items:**
  
  ```
  leaf_ex:  
  addi $sp,$sp,-8  #make stack room
  sw $s0,4($sp)  #save $s0 on stack
  sw $s1,0($sp)  #save $s1 on stack
  add $s0,$a0,$a1
  add $s1,$a2,$a3
  sub $s1,$s0,$s1
  add $v0, $s1,$zero  #$v0=$s1
  lw $s1,0($sp)  #restore $s1
  lw $s0,4($sp)  #restore $s0
  addi $sp,$sp,8  #adjust stack ptr
  jr $ra
  ```

- Important to “leave things as we found them”
- $s0 and $s1 are being used as local variables. Original contents must be restored - why?
- $sp must be restored to original contents
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}

- Something (main, proc) called `sumSquare`, now `sumSquare` is calling procedure `mult`.

- So there’s a value in $ra that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.

- Need to save `sumSquare` return address before call to `mult`.
Nested Procedures (2/2)

- In general, may need to save some other info in addition to $ra$.

- When a C program is run, there are 3 important memory areas allocated:
  - **Static**: Variables declared once per program, cease to exist only after execution completes. (C global variables).
  - **Heap**: Variables declared dynamically
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values
MIPS Memory Allocation for Program and Data

Memory

Stack

Dynamic data (heap)

Static data

Text (Your code)

Reserved

$sp

$gp

PC

0x 0040 0000

0x 1000 8000

0x 1000 0000

0x 7ff fc

0x 0000 0000

0x 0000 0000

0x 0040 0000

0x 1000 0000

0x 7ff fc

0x 0000 0000

0x 0000 0000

0x 0000 0000
Procedure Call and Stack

Stacking of Subroutine Calls & Returns and Environments:

A: CALL B
    B: CALL C
        C: RET
    B: RET
        A

A

A

A

A

A

A
So we have a register $sp$ which always points to the last used space in the stack. {top of stack}

To use stack, we decrement this pointer by the amount of space we need and then fill it with info.

So, how do we compile this?

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```
Using the Stack (2/2)

**Hand-compile**

```markdown
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

sumSquare:

- `addi $sp,$sp,-8`  # space on stack
- `sw $ra, 4($sp)`  # save ret addr
- `sw $a1, 0($sp)`  # save y

```
add $a1,$a0,$zero  # mult(x,x)
jal mult  # call mult
```

```
lw $a1, 0($sp)  # restore y
add $v0,$v0,$a1  # mult()+y
```

```
lw $ra, 4($sp)  # get ret addr
```

```
addi $sp,$sp,8  # restore stack
```

```
jr $ra
```

mult: ...

"push"  

```
sw $ra, 4($sp)  # save ret addr
sw $a1, 0($sp)  # save y
```

```
add $a1,$a0,$zero  # mult(x,x)
jal mult  # call mult
```

```
lw $a1, 0($sp)  # restore y
add $v0,$v0,$a1  # mult()+y
```

```
lw $ra, 4($sp)  # get ret addr
addi $sp,$sp,8  # restore stack
jr $ra
```

"pop"
Steps for Making a Procedure Call

1) Save necessary values onto stack.
2) Assign argument(s), if any.
3) jal call
4) Restore values from stack.
Rules for Procedures

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions
  
  So what are they?
## MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no*</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes*</td>
</tr>
</tbody>
</table>
MIPS Register Convention

- **$at**: may be used by the assembler at any time; unsafe to use
- **$k0–$k1**: may be used by the OS at any time; unsafe to use
- **$gp, $fp**: don’t worry about them
- **Note**: Feel free to read up on $gp and $fp in Appendix A, but you can write perfectly good MIPS code without them.
MIPS Addressing Modes

- **Register addressing** – operand is in a register
- **Base (displacement) addressing** – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction
- **Immediate addressing** – operand is a 16-bit constant contained within the instruction
- **PC-relative addressing** – instruction address is the sum of the PC and a 16-bit constant contained within the instruction
- **Pseudo-direct addressing** – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC
Addressing Modes Illustrated

1. Register addressing

| op | rs | rt | rd | funct |

| Register |
| word operand |

2. Base addressing

| op | rs | rt | offset |

| Memory |
| word or byte operand |

3. Immediate addressing

| op | rs | rt | operand |

4. PC-relative addressing

| op | rs | rt | offset |

| Memory |
| branch destination instruction |

5. Pseudo-direct addressing

| op | jump address |

| Memory |
| jump destination instruction |
## Review: MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OpC</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic (R &amp; I format)</strong></td>
<td>add</td>
<td>0 &amp; 20</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>0 &amp; 22</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>8</td>
<td>addi $s1, $s2, 4</td>
<td>$s1 = $s2 + 4</td>
</tr>
<tr>
<td></td>
<td>shift left logical</td>
<td>0 &amp; 00</td>
<td>sll $s1, $s2, 4</td>
<td>$s1 = $s2 &lt;&lt; 4 (fill with zeros)</td>
</tr>
<tr>
<td></td>
<td>shift right logical</td>
<td>0 &amp; 02</td>
<td>srl $s1, $s2, 4</td>
<td>$s1 = $s2 &gt;&gt; 4 (fill with zeros)</td>
</tr>
<tr>
<td></td>
<td>shift right arithmetic</td>
<td>0 &amp; 03</td>
<td>sra $s1, $s2, 4</td>
<td>$s1 = $s2 &gt;&gt; 4 (fill with sign bit)</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>0 &amp; 24</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>0 &amp; 25</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>nor</td>
<td>0 &amp; 27</td>
<td>nor $s1, $s2, $s3</td>
<td>$s1 = not ($s2</td>
</tr>
<tr>
<td></td>
<td>and immediate</td>
<td>c</td>
<td>and $s1, $s2, ff00</td>
<td>$s1 = $s2 &amp; 0xff00</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>d</td>
<td>or $s1, $s2, ff00</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>f</td>
<td>lui $s1, 0xffff</td>
<td>$s1 = 0xffffffff0000</td>
</tr>
</tbody>
</table>
## Review: MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OpC</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data transfer</strong></td>
<td>load word</td>
<td>23</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory($s2+100)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>2b</td>
<td>sw $s1, 100($s2)</td>
<td>Memory($s2+100) = $s1</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>20</td>
<td>lb $s1, 101($s2)</td>
<td>$s1 = Memory($s2+101)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>28</td>
<td>sb $s1, 101($s2)</td>
<td>Memory($s2+101) = $s1</td>
</tr>
<tr>
<td></td>
<td>load half</td>
<td>21</td>
<td>lh $s1, 101($s2)</td>
<td>$s1 = Memory($s2+102)</td>
</tr>
<tr>
<td></td>
<td>store half</td>
<td>29</td>
<td>sh $s1, 101($s2)</td>
<td>Memory($s2+102) = $s1</td>
</tr>
<tr>
<td><strong>Cond. branch</strong></td>
<td>br on equal</td>
<td>4</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1===$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>br on not equal</td>
<td>5</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 !=$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>set on less than immediate</td>
<td>a</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2&lt;100) $s1=1; else $s1=0</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>0 &amp; 2a</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2&lt;$s3) $s1=1; else $s1=0</td>
</tr>
<tr>
<td><strong>Uncond. jump</strong></td>
<td>jump</td>
<td>2</td>
<td>j 2500</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>0 &amp; 08</td>
<td>jr $t1</td>
<td>go to $t1</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>3</td>
<td>jal 2500</td>
<td>go to 10000; $ra=PC+4</td>
</tr>
</tbody>
</table>
Review: MIPS R3000 ISA

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

- 3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<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>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>R format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>I format</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>26 bit jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>J format</td>
<td></td>
</tr>
</tbody>
</table>