### Aside: 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>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</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 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$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 addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Procedure Calling

Steps required
1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure’s operations
5. Place result in register for caller, remove storage
6. Return to place of call
Register Usage

- \$a0 – \$a3: arguments (reg’s 4 – 7)
- \$v0, \$v1: result values (reg’s 2 and 3)
- \$t0 – \$t9: temporaries
  - Can be overwritten by callee
- \$s0 – \$s7: saved
  - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)
Procedure Call Instructions

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Address of following instruction put in $ra
  - Jumps to target address

- Procedure return: jump register
  `jr $ra`
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements
Leaf Procedure Example

- C code:

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

- Arguments g, ..., j in $a0, ..., $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0
### Leaf Procedure Example

**MIPS code:**

```mips
leaf_example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Register</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$sp, $sp, -4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$s0, 0($sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$t0, $a0, $a1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$t1, $a2, $a3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>$s0, $t0, $t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$v0, $s0, $zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$s0, 0($sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>$sp, $sp, 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Arguments g, ..., j in $a0, ..., $a3**

- **Save $s0 on stack**
- **Procedure body**
- **Result**
- **Restore $s0**
- **Return**

---

\[ f = (g + h) - (i + j); \]  
return f;
Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack (in a stack-frame or activation record):
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
Non-Leaf Procedure Example

- C code:

```c
int fact (int n) {
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0
## Non-Leaf Procedure Example

**MIPS code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -8</td>
<td># adjust stack for 2 items</td>
</tr>
<tr>
<td>sw $ra, 4($sp)</td>
<td># save return address</td>
</tr>
<tr>
<td>sw $a0, 0($sp)</td>
<td># save argument</td>
</tr>
<tr>
<td>slti $t0, $a0, 1</td>
<td># test for n &lt; 1</td>
</tr>
<tr>
<td>beq $t0, $zero, L1</td>
<td></td>
</tr>
<tr>
<td>addi $v0, $zero, 1</td>
<td># if so, result is 1</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
<td># pop 2 items from stack</td>
</tr>
<tr>
<td>jr $ra</td>
<td># and return</td>
</tr>
<tr>
<td>L1: addi $a0, $a0, -1</td>
<td># else decrement n</td>
</tr>
<tr>
<td>jal fact</td>
<td># recursive call</td>
</tr>
<tr>
<td>lw $a0, 0($sp)</td>
<td># restore original n</td>
</tr>
<tr>
<td>lw $ra, 4($sp)</td>
<td># and return address</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
<td># pop 2 items from stack</td>
</tr>
<tr>
<td>mul $v0, $a0, $v0</td>
<td># multiply to get result</td>
</tr>
<tr>
<td>jr $ra</td>
<td># and return</td>
</tr>
</tbody>
</table>

If \( n < 1 \) return 1; else return \( n \times \text{fact}(n - 1) \); Argument \( n \) in $a0, Result in $v0
- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp$ initialized to address allowing $\pm$ offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage
Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings
Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case
    - `lb rt, offset(rs)`   `lh rt, offset(rs)`
  - Sign extend to 32 bits in `rt`
    - `lbu rt, offset(rs)`   `lhu rt, offset(rs)`
  - Zero extend to 32 bits in `rt`
    - `sb rt, offset(rs)`   `sh rt, offset(rs)`
  - Store just rightmost byte/halfword
String Copy Example

- C code (naïve):
  - Null-terminated string
  ```c
  void strcpy (char x[], char y[])
  {
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
  ```
  - Addresses of x, y in $a0, $a1
  - i in $s0
String Copy Example

MIPS code:

```mips
i = 0;
while ((x[i]=y[i])!='\0')
  i += 1;
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -4</td>
<td># adjust stack for 1 item</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td># save $s0</td>
</tr>
<tr>
<td>add $s0, $zero, $zero</td>
<td># i = 0</td>
</tr>
<tr>
<td>L1:</td>
<td></td>
</tr>
<tr>
<td>add $t1, $s0, $a1</td>
<td># addr of y[i] in $t1</td>
</tr>
<tr>
<td>lbu $t2, 0($t1)</td>
<td># $t2 = y[i]</td>
</tr>
<tr>
<td>add $t3, $s0, $a0</td>
<td># addr of x[i] in $t3</td>
</tr>
<tr>
<td>sb $t2, 0($t3)</td>
<td># x[i] = y[i]</td>
</tr>
<tr>
<td>beq $t2, $zero, L2</td>
<td># exit loop if y[i] == 0</td>
</tr>
<tr>
<td>addi $s0, $s0, 1</td>
<td># i = i + 1</td>
</tr>
<tr>
<td>j L1</td>
<td># next iteration of loop</td>
</tr>
<tr>
<td>L2:</td>
<td></td>
</tr>
<tr>
<td>lw $s0, 0($sp)</td>
<td># restore saved $s0</td>
</tr>
<tr>
<td>addi $sp, $sp, 4</td>
<td># pop 1 item from stack</td>
</tr>
<tr>
<td>jr $ra</td>
<td># and return</td>
</tr>
</tbody>
</table>
```
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  \[ \text{lui } rt, \text{ constant} \]
  - Copies 16-bit constant to left 16 bits of \( rt \)
  - Clears right 16 bits of \( rt \) to 0

\[
\begin{align*}
lui \; &s0, \; 61 \quad \text{0000 0000 0111 1101 0000 0000 0000 0000} \\
\text{ori } &s0, \; s0, \; 2304 \quad \text{0000 0000 0111 1101 0000 1001 0000 0000}
\end{align*}
\]
Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

(Pseudo)Direct jump addressing
- Target address = PC_{31...28} : (address × 4)
Target Addressing Example

- Loop code from earlier example
- Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Prev.</th>
<th>PC</th>
<th>Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td>20000</td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Target Addressing Example

- **Note:** 80000 dec is 0x00013880
- 20000 dec is 0x4E20 (0100 1110 0010 0000), with two extra zeros to the right, and 4 zeros to the left it is 0x00013880.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: sll $t1, $s3, 2</td>
<td>80000</td>
<td>00013880</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0100111000100000</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>350000100000</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>582100000001</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>81919001</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td>20000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code.

Example

```assembly
beq $s0,$s1, L1
↓
bne $s0,$s1, L2
j L1
L2: ...
```
Addressing Mode Summary

1. Immediate addressing
   
   \[
   \text{op \hspace{0.5cm} rs \hspace{0.5cm} rt \hspace{0.5cm} \text{Immediate}}
   \]

2. Register addressing
   
   \[
   \text{op \hspace{0.5cm} rs \hspace{0.5cm} rt \hspace{0.5cm} rd \ldots \text{func}}
   \]

3. Base addressing
   
   \[
   \text{op \hspace{0.5cm} rs \hspace{0.5cm} rt \hspace{0.5cm} \text{Address}}
   \]

4. PC-relative addressing
   
   \[
   \text{op \hspace{0.5cm} rs \hspace{0.5cm} rt \hspace{0.5cm} \text{Address}}
   \]

5. Pseudodirect addressing
   
   \[
   \text{op \hspace{0.5cm} \text{Address}}
   \]

Chapter 2 — Instructions: Language of the Computer — 23
Concluding Remarks

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- **Layers of software/hardware**
  - Compiler, assembler, hardware

- **MIPS: typical of RISC ISAs**
  - c.f. x86
Instruction Complexity: Program Size, ExTime/Ins

- Program size in machine instructions ($P$)
- Av. execution time per instruction ($T$)

$P \times T$
Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
MIPS Organization So Far

Processor

Register File

- src1 addr
- src2 addr
- dst addr
- write data

32 registers
($zero - $ra)

32 bits

src1 data

src2 data

32 bits

Memory

- read/write addr
- read data
- write data

32 bits

1…1100

2^30 words

32 bits

byte address (big Endian)

Fetch

PC = PC + 4

Exec

Decode

ALU

Add

Add

32

0...1100

0...1000

0...0100

0...0000