CSEE 3827: Fundamentals of Computer Systems, Spring 2011

7. MIPS Instruction Set Architecture

Prof. Martha Kim (martha@cs.columbia.edu)
Outline (H&H 6.1-6.7.1)

- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Compiling, Assembling, and Loading
- Odds and Ends
Assembly Language

• To command a computer, you must understand its language.
  
  • **Instructions**: words in a computer’s language
  
  • **Instruction set**: the vocabulary of a computer’s language
  
  Instructions indicate the operation to perform and the operands to use.
  
  • **Assembly language**: human-readable format of instructions
  
  • **Machine language**: computer-readable format (1’s and 0’s)
Machine v. Assembly Code

High-level language program (in C)

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

Assembly language program (for MIPS)

```assembly
swap:
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

Binary machine language program (for MIPS)

```
00000000010100001000000000011000
00000000001100000011000000100001
10011001100100100000000000000000
10011101111101000000000000000100
10101101111101000000000000000000
10101100110010000000000000000100
0000001111100000000000000001000
```

(source code)

(assembly code)

(machine code)
What is an ISA?

• An Instruction Set Architecture, or ISA, is an interface between the hardware and the software.

• An ISA consists of:
  
  • a set of operations (instructions)
  
  • data units (sizes, addressing modes, etc.)
  
  • processor state (registers)
  
  • input and output control (memory operations)
  
  • execution model (program counter)
Why have an ISA?

• An ISA provides binary compatibility across machines that share the ISA

• Any machine that implements the ISA X can execute a program encoded using ISA X.

• You typically see families of machines, all with the same ISA, but with different power, performance and cost characteristics.

  • e.g., the MIPS family: MIPS 2000, 3000, 4400, 10000
MIPS Architecture

• MIPS = Microprocessor without Interlocked Pipeline Stages

• MIPS architecture developed at Stanford in 1984, spun out into MIPS Computer Systems

• As of 2004, over 300 million MIPS microprocessors had been sold

• Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

• Once you’ve learned one architecture, it’s easy to learn others.
MIPS is a RISC Architecture

• RISC = Reduced Instruction Set Computer

• RISC is an alternative to CISC (Complex Instruction Set Computer) where operations are significantly more complex.

• Underlying design principles, as articulated by Hennessy and Patterson:
  • Simplicity favors regularity
  • Make the common case fast
  • Smaller is faster
  • Good design demands good compromises

• MIPS (and other RISC architectures) are “load-store” architectures, meaning all operations performed only on operands in registers. (The only instructions that access memory are loads and stores)
What is an ISA?

• An Instruction Set Architecture, or ISA, is an interface between the hardware and the software.

• An ISA consists of:
  
  • a set of operations (instructions)  
  • data units (sized, addressing modes, etc.)  
  • processor state (registers)  
  • input and output control (memory operations)  
  • execution model (program counter)  

(for MIPS)

  - arithmetic, logical, conditional, branch, etc.
  - 32-bit data word
  - 32, 32-bit registers
  - load and store
  - 32-bit program counter
int fact(int n) {
    if (n < 1) return 1;
    else return (n * fact(n - 1));
}

An example Program in MIPS: Factorial(n)

```mips
fact:
  addi $sp, $sp, -8        # adjust stack for 2 items
  sw   $ra, 4($sp)        # save return address
  sw   $a0, 0($sp)        # save argument
  slti $t0, $a0, 1        # test for n < 1
  beq  $t0, $zero, L1
  addi $v0, $zero, 1      # if so, result is 1
  addi $sp, $sp, 8        # pop 2 items from stack
  jr   $ra                # and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      # and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```
An Program in MIPS

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

Instructions
(description of operation to be performed during a cycle)

C code

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># if so, result is 1</td>
</tr>
<tr>
<td>addi $v0, $zero, 1</td>
<td># pop 2 items from stack</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
<td># and return</td>
</tr>
<tr>
<td>jr $ra</td>
<td></td>
</tr>
<tr>
<td>L1:</td>
<td></td>
</tr>
<tr>
<td>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>
An Program in MIPS

C code

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

MIPS code

```
addi $sp, $sp, -8     # adjust stack for 2 items
sw   $ra, 4($sp)      # save return address
sw   $a0, 0($sp)      # save argument
slti $t0, $a0, 1      # test for n < 1
beq  $t0, $zero, L1
addi $v0, $zero, 1    # if so, result is 1
addi $sp, $sp, 8      # pop 2 items from stack
jr   $ra              # and return
L1: addi $a0, $a0, -1     # else decrement n
jal  fact             # recursive call
lw   $a0, 0($sp)      # restore original n
lw   $ra, 4($sp)      # and return address
addi $sp, $sp, 8      # pop 2 items from stack
mul  $v0, $a0, $v0    # multiply to get result
jr   $ra              # and return
```
int fact(int n) {
    if (n < 1) return 1;
    else return (n * fact(n - 1));
}

C code

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

MIPS code

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1   # if so, result is 1
    addi $t0, $zero, 1    # test for n < 1
    jr   $ra              # and return
    addi $a0, $a0, -1     # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      # and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    jr   $ra              # and return
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

Constants
An Program in MIPS

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

C code

MIPS code

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        # pop 2 items from stack
    jr   $ra                # and return
L1:  addi $a0, $a0, -1     # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        # and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```
An Program in MIPS

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

C code

Control Labels and “Jump” Instructions

### MIPS code

```mips
# Adjust stack for 2 items
addi $sp, $sp, -8
# Save return address
sw $ra, 4($sp)
# Save argument
sw $a0, 0($sp)
# Test for n < 1
slti $t0, $a0, 1
# If so, result is 1
beq $t0, $zero, L1
# Pop 2 items from stack
addi $v0, $zero, 1
# And return
addi $sp, $sp, 8
jr $ra

L1:  # Else decrement n
    addi $a0, $a0, -1
    # Recursive call
    jal fact
    # Restore original n
    lw $a0, 0($sp)
    # And return address
    lw $ra, 4($sp)
    # Pop 2 items from stack
    addi $sp, $sp, 8
    # Multiply to get result
    mul $v0, $a0, $v0
    # And return
    jr $ra
```
Instruction Classes

- **Memory Access**: Move data to/from memory from/to registers

- **Arithmetic/Logic**: Perform (via functional unit) computation on data in registers (store result in a register)

- **Jump/Jump Subroutine**: direct control to a different part of the program (not next word in memory)

- **Conditional branch**: test values in registers. If test returns true, move control to different part of program. Otherwise, proceed to next word

**NB**: These are functional classes. Later we will classify the instructions according to their formats (R-type, I-type, etc.)
Arithmetic Instructions

- Addition and subtraction

- Three operands: two source, one destination

- \texttt{add a, b, c} \ # a gets b + c

- All arithmetic operations (and many others) have this form

---

**Design principle:**

Regularity makes implementation simpler

Simplicity enables higher performance at lower cost
Arithmetic Example 1

C code

\[ f = (g + h) - (i + j) \]

MIPS assembly

```
add t0, g, h  # temp t0=g+h
add t1, i, j  # temp t1=i+j
sub f, t0, t1 # f = t0-t1
```

Arithmetic Example 1 w. Registers

MIPS assembly w.o registers

```
add t0, g, h  # temp t0=g+h
add t1, i, j  # temp t1=i+j
sub f, t0, t1 # f = t0-t1
```

store: f in $s0, g in $s1, h in $s2, i in $s3, and j in $s4

MIPS assembly w. registers

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```
Memory Operands

- Main memory used for composite data (e.g., arrays, structures, dynamic data)

- To apply arithmetic operations
  - Load values from memory into registers (load instruction = mem read)
  - Store result from registers to memory (store instruction = mem write)

- Memory is byte-addressed (each address identifies an 8-bit byte)

- Words (32-bits) are aligned in memory (meaning each address must be a multiple of 4)

- MIPS is big-endian (i.e., most significant byte stored at least address of the word)
Memory Operands

- Main memory used for composite data (e.g., arrays, structures, dynamic data)

- To apply arithmetic operations
  - Load values from memory into registers (load instruction = mem read)
  - Store result from registers to memory (store instruction = mem write)

- Memory is byte-addressed (each address identifies an 8-bit byte)

- Words (32-bits) are aligned in memory (meaning each address must be a multiple of 4)

- MIPS is big-endian (i.e., most significant byte stored at least address of the word)
Memory Operand Example 1

\[ g = h + A[8] \]

**C code**

\[ g \text{ in } $s1, \ h \text{ in } $s2, \ \text{base address of } A \text{ in } $s3 \]

\[ \text{index} = 8 \text{ requires offset of 32 (8 items x 4 bytes per word)} \]

**MIPS assembly**

\[ \text{lw } $t0, 32($s3) \ # \text{ load word} \]
\[ \text{add } $s1, $s2, $t0 \]
Memory Operand Example 2


**C code**

\[ h \text{ in } \$s2, \text{ base address of } A \text{ in } \$s3 \]

- index = 8 requires offset of 32 (8 items x 4 bytes per word)
- index = 12 requires offset of 48 (12 items x 4 bytes per word)

**MIPS assembly**

```
lw $t0, 32($s3)   # load word
add $t0, $s2, $t0
sw $t0, 48($s3)   # store word
```
Registers v. Memory

- Registers are faster to access than memory

- Operating on data in memory requires loads and stores
  - (More instructions to be executed)

- Compiler should use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important for performance
Immediate Operands

- Constant data encoded in an instruction
  
  \[ \text{addi } \$s3, \$s3, 4 \]

- No subtract immediate instruction, just use the negative constant
  
  \[ \text{addi } \$s2, \$s1, -1 \]

**Design principle:** make the common case fast

*Small constants are common*

*Immediate operands avoid a load instruction*
The Constant Zero

- MIPS register 0 ($zero) is the constant 0
- $zero cannot be overwritten
- Useful for many operations, for example, a move between two registers

\[
\text{add } \$t2, \$s1, \$zero
\]
# Register Numbers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
<td>Values for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$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–$s7</td>
<td>16–23</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24–25</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$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>

**Note:** Register 1 ($at) is reserved for the assembler, and 26-27 ($k0–$k1) are reserved for the OS.
### MIPS instructions to date

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>32\text{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>sub (subtract)</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>34\text{ten}</td>
<td>n.a.</td>
</tr>
<tr>
<td>add immediate</td>
<td>I</td>
<td>8\text{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>constant</td>
</tr>
<tr>
<td>lw (load word)</td>
<td>I</td>
<td>35\text{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>sw (store word)</td>
<td>I</td>
<td>43\text{ten}</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
</tbody>
</table>

**NB:** \( \text{reg} = \text{register number between 0 and 31}; \\
\text{address} = 16\text{-bit address} \)
MIPS R-format Instructions

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: register destination number
  - sham: shift amount (00000 for now)
  - funct: function code (extends opcode)
R-format Example

<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>

add $t0, $s1, $s2

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>
MIPS I-format Instructions

- Includes immediate arithmetic and load/store operations
  - op: operation code (opcode)
  - rs: first source register number
  - rt: destination register number
  - constant: offset added to base address in rs, or immediate operand
MIPS Logical Operations

• Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>s1l</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

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

- Shift left logical (op = sll)
  - Shift left and fill with 0s
  - \( sll \) by \( i \) bits multiplies by \( 2^i \)
- Shift right logical (op = srl)
  - Shift right and fill with 0s
  - \( srl \) by \( i \) bits divides by \( 2^i \) (for unsigned values only)
- \texttt{shamt} indicates how many positions to shift
- example: \texttt{sll $t2, $s0, 4 \# $t2 = $s0 \ll 4 \text{ bits}}
- R-format

```
0 0 16 10 4 0
```
Full Complement of Shift Instructions

• sll: shift left logical \( (\text{sll } t0, t1, 5 \ # t0 \leq t1 \ll 5) \)
• srl: shift right logical \( (\text{srl } t0, t1, 5 \ # t0 \leq t1 \gg 5) \)
• sra: shift right arithmetic \( (\text{sra } t0, t1, 5 \ # t0 \leq t1 \ggg 5) \)

• Variable shift instructions:
  • sllv: shift left logical variable
    \( (\text{sllv } t0, t1, t2 \ # t0 \leq t1 \ll t2) \)
  • srlv: shift right logical variable
    \( (\text{srlv } t0, t1, t2 \ # t0 \leq t1 \gg t2) \)
  • srav: shift right arithmetic variable
    \( (\text{srav } t0, t1, t2 \ # t0 \leq t1 \ggg t2) \)
Generating Constants

- 16-bit constants using `addi`:

```
// int is a 32-bit signed word
int a = 0x4f3c
```

C code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

MIPS assembly

- 32-bit constants using `lui` and `ori`:

```
int a = 0xFEDC8765;
```

C code

```
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

MIPS assembly

*lui* loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.
AND Operations

• example: \texttt{and $t0, t1, t2} \quad \# \quad \texttt{$t0 = t1 \& t2}$

• Useful for masking bits in a word (selecting some bits, clearing others to 0)

\begin{center}
\begin{tabular}{c}
$\texttt{t1}$: 0000 0000 0000 0000 0000 1101 1100 0000 \\
$\texttt{t2}$: 0000 0000 0000 0000 0011 1100 0000 0000 \\
$\texttt{t0}$: 0000 0000 0000 0000 0000 1100 0000 0000
\end{tabular}
\end{center}
OR Operations

• example:  `or $t0, $t1, $t2`  # $t0 = $t1 | $t2

• Useful to include bits in a word (set some bits to 1, leaving others unchanged)

| $t1: | 0000 0000 0000 0000 0000 0000 1101 1100 0000 |
| $t2: | 0000 0000 0000 0000 0000 0000 11 1100 0000 0000 |
| $t0: | 0000 0000 0000 0000 0000 0000 11 1101 1100 0000 |
NOT Operations

- Useful to invert bits in a word

- MIPS has 3 operand NOR instruction, used to compute NOT

- example:  `nor $t0, $t1, $zero  # $t0 = ~$t1`

<table>
<thead>
<tr>
<th>$t1:</th>
<th>0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0:</td>
<td>1111 1111 1111 1111 1111 0010 0011 1111</td>
</tr>
</tbody>
</table>
Conditional Operations

• Branch to a labeled instruction if a condition is true

• Otherwise, continue sequentially

• Instruction labeled with colon e.g. L1: add $t0, $t1, $t2

• beq rs, rt, L1 # if (rs == rt) branch to instr labeled L1

• bne rs, rt, L1 # if (rs != rt) branch to instr labeled L1

• j L1 # unconditional jump to instr labeled L1
Compiling an If Statement

```c
if (i == j)
    f = g+h
else
    f = g-h
```

```
bne $s3, $s4, Else
    add $s0, $s1, $s2
    j Exit
Else:
    sub $s0, $s1, $s2
Exit:
```

- Where, f is in $s0, g is in $s1, and h is in $s2
- The assembler calculates the addresses corresponding to the labels
Compiling a Loop Statement

C code

```c
while (save[i] == k) 
  i += 1
```

MIPS assembly

```
Loop:
  sll $t1, $s3, 2
  add $t1, $t1, $s5
  lw $t0, 0($t1)
  bne $t0, $s4, Exit
  addi $s3, $s3, 1
  j Loop
Exit:
```

- Where, i is in $s3, k is in $s4, address of save in $s5
Basic Blocks

• A basic block is a sequence of instructions with
  • No embedded branches except at the end
  • No branch targets except at the beginning

• A compiler identifies basic blocks for optimization

• Advanced processors can accelerate execution of basic blocks
More Conditional Operations

• Set result to 1 if a condition is true

  • \texttt{slt rd, rs, rt} \quad \# (rs < rt) ? rd=1 : rd=0

  • \texttt{slti rd, rs, constant} \quad \# (rs < constant) ? rd=1 : rd=0

• Use in combination with \texttt{beq} or \texttt{bne}

  \texttt{slt \$t0, \$s1, \$s2} \quad \# if ($s1 < $s2)
  \texttt{bne \$t0, $zero, L} \quad \# \text{branch to L}
Branch Instruction Design

• Why not blt, bge, etc.?

• Hardware for <, >= etc. is slower than for = and !=
  • Combining with a branch involves more work per instruction, requiring a slower clock
  • All instructions penalized because of this

• As beq and bne are the common case, this is a good compromise
Signed v. Unsigned

• Signed comparison: slt, slti

• Unsigned comparison: sltu, sltui

• Example:

```
$s0: 1111 1111 1111 1111 1111 1111 1111 1111
$s1: 0000 0000 0000 0000 0000 0000 0000 0001
```

```bash
slt $t0, $s0, $s1  # signed: -1 < 1 thus $t0=1
sltu $t0, $s0, $s1 # unsigned: 4,294,967,295 > 1 thus $t0=0
```
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
  6. Return to place of call
Register Usage

- $a0-$a3: arguments
- $v0, $v1: result values
- $t0-$t9: temporaries, can be overwritten by callee
- $s0-$s7: contents saved

*** must be restored by callee

- $gp: global pointer for static data
- $sp: stack pointer
- $fp: frame pointer
- $ra: return address

Note: There is nothing special about these registers’ design, only their implied use!!!

E.g., could store return value in $sp if calling and callee program both agreed to do this - just beware of messing up the stack for all other programs if not properly restored
Memory Layout

• **Text:** program code

• **Static data:** global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp$ initialized to an address allowing +/- offsets in this segment

• **Dynamic data:** heap
  - e.g., malloc in C, new in Java

• **Stack:** automatic storage

\[\text{Memory Layout} \]

\[\begin{align*}
\text{Stack} \quad & \quad \text{Dynamic data} \\
\text{Static data} \quad & \quad \text{Text} \\
\text{Reserved} \quad & \quad \text{Reserved}
\end{align*}\]

\[\begin{align*}
\text{sp} & \rightarrow 7\text{fff} \ f\text{ff}c_{\text{hex}} \\
\text{gp} & \rightarrow 1000 \ 8000_{\text{hex}} \\
& \quad 1000 \ 0000_{\text{hex}} \\
\text{pc} & \rightarrow 0040 \ 0000_{\text{hex}} \\
& \quad 0
\end{align*}\]
Local Data on the Stack

- Local data allocated by the callee

- Procedure frame (activation record) used by compiler to manage stack storage

- Cross-call register preservation

<table>
<thead>
<tr>
<th>Preserved</th>
<th>Not preserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved registers: $s0−$s7</td>
<td>Temporary registers: $t0−t9</td>
</tr>
<tr>
<td>Stack pointer register: $sp</td>
<td>Argument registers: $a0−$a3</td>
</tr>
<tr>
<td>Return address register: $ra</td>
<td>Return value registers: $v0−v1</td>
</tr>
<tr>
<td>Stack above the stack pointer</td>
<td>Stack below the stack pointer</td>
</tr>
</tbody>
</table>
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)
int leaf_example(int g, h, i, j) {
    int f;
    f = (g+h) - (i+j);
    return f;
}

C code

- Arguments g, h, i, j in $a0 - $a3
- f will go in $s0 (so will have to save existing contents of $s0 to stack)
- result in $v0
Leaf Procedure Example 2

C code

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

MIPS assembly

```
leaf_example:
  addi $sp, $sp, -4
  sw $s0, 0($sp)
  add $t0, $a0, $a1
  add $t1, $a2, $a2
  sub $s0, $t0, $t1
  add $v0, $s0, $zero
  lw $s0, 0($sp)
  addi $sp, $sp, 4
  jr $ra
```

- save $s0 on stack
- procedure body
- result
- restore $s0
- return
Non-Leaf Procedures

• A non-leaf procedure is a procedure that calls another procedure

• For a nested call, the caller needs to save to the stack
  • Its return address
  • Any arguments and temporaries needed after the call

• After the call, the caller must restore these values from the stack
Non-Leaf Procedure Example

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

**C code**
Non-Leaf Procedure Example 2

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

### C code

```mips
<table>
<thead>
<tr>
<th>fact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -8</td>
</tr>
<tr>
<td>sw $ra, 4($sp)</td>
</tr>
<tr>
<td>sw $a0, 0($sp)</td>
</tr>
<tr>
<td>slti $t0, $a0, 1</td>
</tr>
<tr>
<td>beq $t0, $zero, L1</td>
</tr>
<tr>
<td>addi $v0, $zero, 1</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
</tr>
<tr>
<td>jr $ra</td>
</tr>
<tr>
<td>L1:</td>
</tr>
<tr>
<td>addi $a0, $a0, -1</td>
</tr>
<tr>
<td>jal fact</td>
</tr>
<tr>
<td>lw $a0, 0($sp)</td>
</tr>
<tr>
<td>lw $ra, 4($sp)</td>
</tr>
<tr>
<td>addi $sp, $sp, 8</td>
</tr>
<tr>
<td>mul $v0, $a0, $v0</td>
</tr>
<tr>
<td>jr $ra</td>
</tr>
</tbody>
</table>
```
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 are variable-length encodings
Byte/Halfword Operations

• Could use bitwise operations

• MIPS has byte/halfword load/store

  • `lb rt, offset(rs)` # sign extend byte to 32 bits in rt
  • `lh rt, offset(rs)` # sign extend halfword to 32 bits in rt
  • `lbu rt, offset(rs)` # zero extend byte to 32 bits in rt
  • `lhu rt, offset(rs)` # zero extend halfword to 32 bits in rt
  • `sb rt, offset(rs)` # store rightmost byte
  • `sh rt, offset(rs)` # store rightmost halfword
String Copy Example

```c
void strcpy (char x[], char y[]) {
    int i;
    i = 0;
    while ((x[i]=y[i]) != '\0')
        i += 1;
}
```

C code (naive)

- Null-terminated string
- Addresses of x and y in $a0 and $a1 respectively
- i in $s0
void strcpy (char x[], char y[]) {
    int i;
    i = 0;
    while ((x[i]=y[i]) != ‘\0’)
        i += 1;
}

C code (naive)

<table>
<thead>
<tr>
<th>strcpy</th>
<th>MIPS assembly</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: 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: 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 bits usually sufficient

- For occasional, 32-bit constant:

  \[
  \text{lui \ rt, constant}
  \]

  \[
  \begin{align*}
  \text{• copies 16-bit constant to the left (upper) bits of rt} \\
  \text{• clears right (lower) 16 bits of rt to 0}
  \end{align*}
  \]

- example usage:

  \[
  \begin{align*}
  \text{lui \ $s0, 61} & \quad \text{$s0: \hspace{1cm}} & \hspace{1cm} \text{0000 0000 0111 1101 0000 0000 0000 0000} \\
  \text{ori \ $s0, $s0, 2304} & \quad \text{$s0: \hspace{1cm}} & \hspace{1cm} \text{0000 0000 0111 1101 0000 1001 0000 0000}
  \end{align*}
  \]
## Branch Addressing

- Branch instructions specify: opcode, two registers, branch target

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

- Most branch targets are near branch (either forwards or backwards)

- PC-relative addressing
  - target address = PC + (offset * 4)
  - PC already incremented by four when the target address is calculated
Jump Addressing

• Jump (j and jal) targets could be anywhere in a text segment, so, encode the full address in the instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

• target address = PC[31:28] : (address * 4)
Target Addressing Example

• Loop code from earlier example

• Assume loop at location 80000

Loop: sll $t1, $s3, 2
     add $t1, $t1, $s5
     lw $t0, 0($t1)
     bne $t0, $s4, Exit
     addi $s3, $s3, 1
     j Loop

Exit:
Addressing Mode Summary

1. Immediate addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Immediate} & & \\
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{rd} \ldots \text{func} \\
   \text{Registers} & \text{Register} & & \\
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Address} & \text{Register} & + \\
   \text{Memory} & \text{Byte} & \text{Halfword} & \text{Word} \\
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Address} & \text{PC} & + \\
   \text{Memory} & \text{Word} & \\
   \end{array}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c}
   \text{op} \\
   \text{Address} \\
   \text{Memory} & \text{Word} \\
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{PC} \\
   \text{Memory} & \text{Word} \\
   \end{array}
   \]
Branching Far Away

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

• Example:

\[
\text{beq} \; $s0,$s1, \; L1 \quad \text{becomes} \quad \text{bne} \; $s0,$s1, \; L2
\]

\[
\text{j} \; L1
\]

\[
L2: \; \ldots
\]
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions, one to one.

- Pseudoinstructions are shorthand. They are recognized by the assembler but translated into small bundles of machine instructions.

\[
\begin{align*}
\text{move } & \quad \text{becomes} \quad \text{add} \\
move \, $t0,$t1 & \quad \rightarrow \quad add \, $t0,$zero,$t1 \\
blt \, $t0,$t1,L & \quad \rightarrow \quad slt \, $at,$t0,$t1 \quad \text{bne} \, $at,$zero,L
\end{align*}
\]

- $at$ (register 1) is an “assembler temporary”
Programming Pitfalls

• Sequential words are not at sequential addresses -- increment by 4 not by 1!

• Keeping a pointer to an automatic variable (on the stack) after procedure returns
Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise, opcode tells what instruction it is

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Field Values</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x2237FFF1)</td>
<td>001000 10001 10111 1111 1111 1111 0001</td>
<td>8 17 23 -15</td>
</tr>
<tr>
<td>(0x02F34022)</td>
<td>000000 10111 10011 01000 00000 100010</td>
<td>0 23 19 8 0 34</td>
</tr>
</tbody>
</table>
In conclusion: Fallacies

1. Powerful (complex) instructions lead to higher performance
   - Fewer instructions are required
   - **But** complex instructions are hard to implement. As a result implementation may slow down all instructions including simple ones.
   - Compilers are good at making fast code from simple instructions.

2. Use assembly code for high performance
   - Modern compilers are better than predecessors at generating good assembly
   - More lines of code (in assembly) means more errors and lower productivity
In conclusion: More Fallacies

3. Backwards compatibility means instruction set doesn’t change

![Graph showing growth of x86 instruction set over time.](image)

**FIGURE 2.43** *Growth of x86 instruction set over time.* While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors. Copyright © 2009 Elsevier, Inc. All rights reserved.