Instruction Set Architectures

MIPS

The GCD Algorithm

MIPS Registers

Types of Instructions
  - Computational
  - Load and Store
  - Jump and Branch
  - Other

Instruction Encoding
  - Register-type
  - Immediate-type
  - Jump-type

Assembler

Pseudoinstructions

Higher-Level Constructs
  - Expressions
  - Conditionals
  - Loops
  - Arrays
  - Strings & Hello World
  - ASCII

Subroutines
  - Towers of Hanoi Example
  - Factorial Example

Memory Layout

Differences in Other ISAs
Machine, Assembly, and C Code

00010000100001010000000000000000111
0000000010100100000100000101010
00010100010000000000000000000011
00000000101001000100001000100011
00000100000000011111111111111100
000001000001010010001010000101011
00000100000000011111111111111100
00000000100001010010000000100011
00001000000000111111111111111010
00000000000010000100001000010001
000000111111000000000000100000001
0000001111100000000000000000001000
Machine, Assembly, and C Code

```
beq $4, $5, 28
slt $2, $5, $4
bne $2, $0, 12
subu $5, $5, $4
bgez $0 -16
subu $4, $4, $5
bgez $0 -24
addu $2, $0, $4
jr $31
```

gcd:
```
beq $a0, $a1, .L2
slt $v0, $a1, $a0
bne $v0, $zero, .L1
subu $a1, $a1, $a0
gcd .L1:
subu $a0, $a0, $a1
gcd .L2:
move $v0, $a0
j $ra
```

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```
Machine, Assembly, and C Code

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

```assembly
gcd:
    beq $a0, $a1, .L2
    slt $v0, $a1, $a0
    bne $v0, $zero, .L1
    subu $a1, $a1, $a0
    b gcd

.L1:
    subu $a0, $a0, $a1
    b gcd

.L2:
    move $v0, $a0
    j $ra
```
Machine, Assembly, and C Code

```
0001000010000101000000000000000111
00000000101001000000100001010110
000101000100000000000000000000011
0000000010100100000010000100011
00000100000000011111111111111100
00000000100001010000001000010011
0000000000000010100001000000100001
00000100000000011111111111111110
000000000000010000001000000100001
0000000011111100000000000000000100
```

```
beq $4, $5, 28
slt $2, $5, $4
bne $2, $0, 12
subu $5, $5, $4
bgez $0 -16
subu $4, $4, $5
bgez $0 -24
addu $2, $0, $4
jr $31
```

gcd:
```
beq $a0, $a1, .L2
slt $v0, $a1, $a0
bne $v0, $zero, .L1
subu $a1, $a1, $a0
.b gcd
.L1:
    subu $a0, $a0, $a1
    b  gcd
.L2:
    move $v0, $a0
    j  $ra
```

```
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```
al·go·rithm

a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end especially by a computer

Merriam-Webster
The Stored-Program Computer


“Since the device is primarily a computer, it will have to perform the elementary operations of arithmetics most frequently. [...] It is therefore reasonable that it should contain *specialized organs for just these operations*.

“If the device is to be [...] as nearly as possible all purpose, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control organs which see to it that these instructions [...] are carried out. The former must be *stored in some way* [...] the latter are represented by definite operating parts of the device.

“Any device which is to carry out long and complicated sequences of operations (specifically of calculations) *must have a considerable memory*. 
Instruction Set Architecture (ISA)

ISA: The interface or contact between the hardware and the software

Rules about how to code and interpret machine instructions:

- Execution model (program counter)
- Operations (instructions)
- Data formats (sizes, addressing modes)
- Processor state (registers)
- Input and Output (memory, etc.)
Architecture vs. Microarchitecture

Architecture: The interface the hardware presents to the software

Microarchitecture: The detailed implementation of the architecture
MIPS

Microprocessor without Interlocked Pipeline Stages

RISC vs. CISC Architectures

MIPS is a Reduced Instruction Set Computer. Others include ARM, PowerPC, SPARC, HP-PA, and Alpha.

A Complex Instruction Set Computer (CISC) is one alternative. Intel’s x86 is the most prominent example; also Motorola 68000 and DEC VAX.

RISC’s underlying principles, due to Hennessy and Patterson:

- Simplicity favors regularity
- Make the common case fast
- Smaller is faster
- Good design demands good compromises
The GCD Algorithm

Euclid, *Elements*, 300 BC.

The greatest common divisor of two numbers does not change if the smaller is subtracted from the larger.

1. Call the two numbers $a$ and $b$
2. If $a$ and $b$ are equal, stop: $a$ is the greatest common divisor
3. Subtract the smaller from the larger
4. Repeat steps 2–4
The GCD Algorithm

Let’s be a little more explicit:

1. Call the two numbers \( a \) and \( b \)
2. If \( a \) equals \( b \), go to step 8
3. if \( a \) is less than \( b \), go to step 6
4. Subtract \( b \) from \( a \) \( a > b \) here
5. Go to step 2
6. Subtract \( a \) from \( b \) \( a < b \) here
7. Go to step 2
8. Declare \( a \) the greatest common divisor
9. Go back to doing whatever you were doing before
Euclid’s Algorithm in MIPS Assembly

gcd:

```
beq $a0, $a1, .L2  # if a = b, go to exit
sgt $v0, $a1, $a0  # Is b > a?
bne $v0, $zero, .L1  # Yes, goto .L1

subu $a0, $a0, $a1  # Subtract b from a (b < a)
gcd  # and repeat
```

```
.L1:
subu $a1, $a1, $a0  # Subtract a from b (a < b)
gcd  # and repeat
```

```
.L2:
move $v0, $a0  # return a
j $ra  # Return to caller
```

Instructions
Euclid’s Algorithm in MIPS Assembly

```assembly
gcd:
    beq $a0, $a1, .L2  # if a = b, go to exit
    sgt $v0, $a1, $a0  # Is b > a?
    bne $v0, $zero, .L1 # Yes, goto .L1

    subu $a0, $a0, $a1 # Subtract b from a (b < a)
    b gcd # and repeat

.L1:
    subu $a1, $a1, $a0 # Subtract a from b (a < b)
    b gcd # and repeat

.L2:
    move $v0, $a0 # return a
    j $ra # Return to caller
```

Operands: Registers, etc.
Euclid’s Algorithm in MIPS Assembly

```
gcd:    beq   $a0, $a1, .L2    # if a = b, go to exit
       sgt   $v0, $a1, $a0    # Is b > a?
       bne   $v0, $zero, .L1  # Yes, goto .L1

       subu  $a0, $a0, $a1    # Subtract b from a (b < a)
       b    gcd               # and repeat

.L1:

       subu  $a1, $a1, $a0    # Subtract a from b (a < b)
       b    gcd               # and repeat

.L2:

       move  $v0, $a0         # return a
       j     $ra              # Return to caller

Labels
```
Euclid’s Algorithm in MIPS Assembly

gcd:
beq $a0, $a1, .L2  # if a = b, go to exit
sgt $v0, $a1, $a0  # Is b > a?
bne $v0, $zero, .L1  # Yes, goto .L1

subu $a0, $a0, $a1  # Subtract b from a (b < a)
b  gcd  # and repeat

.L1:
subu $a1, $a1, $a0  # Subtract a from b (a < b)
b  gcd  # and repeat

.L2:
move $v0, $a0  # return a
j  $ra  # Return to caller

Comments
Euclid’s Algorithm in MIPS Assembly

gcd:

```
beq $a0, $a1, .L2  # if a = b, go to exit
sgt $v0, $a1, $a0  # Is b > a?
bne $v0, $zero, .L1 # Yes, goto .L1

subu $a0, $a0, $a1  # Subtract b from a (b < a)

b     gcd          # and repeat

.L1:

subu $a1, $a1, $a0  # Subtract a from b (a < b)

b     gcd          # and repeat

.L2:

move $v0, $a0      # return a

j     $ra          # Return to caller
```

Arithmetic Instructions
Euclid’s Algorithm in MIPS Assembly

gcd:
  beq $a0, $a1, .L2  # if a = b, go to exit
  sgt $v0, $a1, $a0  # Is b > a?
  bne $v0, $zero, .L1  # Yes, goto .L1

  subu $a0, $a0, $a1  # Subtract b from a (b < a)
  b gcd
  # and repeat

.L1:
  subu $a1, $a1, $a0  # Subtract a from b (a < b)
  b gcd
  # and repeat

.L2:
  move $v0, $a0  # return a
  j $ra  # Return to caller

Control-transfer instructions
General-Purpose Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
<th>Preserved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Constant zero</td>
<td></td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Reserved (assembler)</td>
<td></td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
<td>Function result</td>
<td></td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4–7</td>
<td>Function arguments</td>
<td></td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8–15</td>
<td>Temporaries</td>
<td></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>Temporaries</td>
<td></td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>26-27</td>
<td>Reserved (OS)</td>
<td></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>

Each 32 bits wide
Only 0 truly behaves differently; usage is convention
Types of Instructions

- **Computational**: Arithmetic and logical operations
- **Load and Store**: Writing and reading data to/from memory
- **Jump and branch**: Control transfer, often conditional
- **Miscellaneous**: Everything else
## Computational Instructions

<table>
<thead>
<tr>
<th><strong>Arithmetic</strong></th>
<th><strong>Shift Instructions</strong></th>
<th><strong>Multiply/Divide</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>multiply</td>
</tr>
<tr>
<td>addu</td>
<td>addu</td>
<td>multiply unsigned</td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td>divide</td>
</tr>
<tr>
<td>subu</td>
<td>subu</td>
<td>divide unsigned</td>
</tr>
<tr>
<td>slt</td>
<td>slt</td>
<td>mfhi</td>
</tr>
<tr>
<td>sltu</td>
<td>sltu</td>
<td>mthi</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>mflo</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>mtlo</td>
</tr>
<tr>
<td>xor</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>nor</td>
<td></td>
</tr>
</tbody>
</table>

### Arithmetic (immediate)

<table>
<thead>
<tr>
<th><strong>Arithmetic (immediate)</strong></th>
<th><strong>Shift Instructions</strong></th>
<th><strong>Multiply/Divide</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>addiu</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>slti</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td>sltiu</td>
<td></td>
</tr>
<tr>
<td>andi</td>
<td>andi</td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td>ori</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>nor</td>
<td></td>
</tr>
<tr>
<td>lui</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computational Instructions

Arithmetic, logical, and other computations. Example:

```
add $t0, $t1, $t3
```

“Add the contents of registers $t1 and $t3; store the result in $t0”

Register form:

```
op\ation R_D, R_S, R_T
```

“Perform operation on the contents of registers $R_S$ and $R_T$; store the result in $R_D”

Passes control to the next instruction in memory after running.
Arithmetic Instruction Example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0$</td>
<td>$s1$</td>
<td>$s2$</td>
<td>$s3$</td>
<td>$s4$</td>
<td>$s5$</td>
<td>$s6$</td>
<td>$s7$</td>
</tr>
</tbody>
</table>

\[
a = b - c;
\]

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

\[
\begin{align*}
\text{subu} & \quad $s0$, $s1$, $s2 \\
\text{addu} & \quad $t0$, $s4$, $s5 \\
\text{addu} & \quad $t1$, $s6$, $s7 \\
\text{subu} & \quad $s3$, $t0$, $t1
\end{align*}
\]

“Signed” addition/subtraction (\texttt{add/sub}) throw an exception on a two’s-complement overflow; “Unsigned” variants (\texttt{addu/subu}) do not. Resulting bit patterns otherwise identical.
Bitwise Logical Operator Example

main:
move $s0, $ra # Save the return address

li $t0, 0xFF00FF00 # "Load immediate"
li $t1, 0xF0F0F0F0 # "Load immediate"

nor $t2, $t0, $t1 # Puts 0x000F000F in $t2

li $v0, 1 # print_int
move $a0, $t2 # print contents of $t2
syscall

j $s0 # return from main()

This assembles and runs under the SPIM simulator.
Immediate Computational Instructions

Example:

```
addiu $t0, $t1, 42
```

“Add the contents of register $t1 and 42; store the result in register $t0”

In general,

```
operation R_D, R_S, I
```

“Perform operation on the contents of register $R_S$ and the signed 16-bit immediate $I$; store the result in $R_D$”

Thus, $I$ can range from $-32768$ to $32767$. 
32-Bit Constants and lui

It is easy to load a register with a constant from −32768 to 32767, e.g.,

\begin{verbatim}
ori $t0, $0, 42
\end{verbatim}

Larger numbers use “load upper immediate,” which fills a register with a 16-bit immediate value followed by 16 zeros; an OR handily fills in the rest. E.g., Load $t0 with 0xC0DEFACE:

\begin{verbatim}
lui $t0, 0xC0DE
ori $t0, $t0, 0xFACE
\end{verbatim}

The assembler automatically expands the \texttt{li} pseudo-instruction into such an instruction sequence

\begin{verbatim}
li $t1, 0xCAFE0BOE \rightarrow lui $t1, 0xCAFE
ori $t1, $t1, 0x0BOE
\end{verbatim}
Multiplication and Division

Multiplication gives 64-bit result in two 32-bit registers: HI and LO. Division: LO has quotient; HI has remainder.

```c
int multdiv(
    int a,     // $a0
    int b,     // $a1
    unsigned c, // $a2
    unsigned d) // $a3
{
    a = a * b + c;
    c = c * d + a;

    a = a / c;
    b = b % a;
    c = c / d;
    d = d % c;

    return a + b + c + d;
}
```

```assembly
multdiv:
mult $a0,$a1     # a * b
mflo $t0
addu $a0,$t0,$a2 # a = a*b + c
mult $a2,$a3    # c * d
mflo $t1
addu $a2,$t1,$a0 # c = c*d + a
divu $0,$a0,$a2  # a / c
mflo $a0         # a = a/c
div $0,$a1,$a0   # b % a
mfhi $a1         # b = b%a
divu $0,$a2,$a3  # c / d
mflo $a2         # c = c/d
addu $t2,$a0,$a1 # a + b
addu $t2,$t2,$a2 # (a+b) + c
divu $0,$a3,$a2  # d % c
mfhi $a3         # d = d%c
addu $v0,$t2,$a3 # ((a+b)+c) + d
j     $ra
```
**Shift Left**

Shifting left amounts to multiplying by a power of two. Zeros are added to the least significant bits. The constant form explicitly specifies the number of bits to shift:

\[
\text{sll}\ \text{a0}, \text{a0}, 1
\]

The variable form takes the number of bits to shift from a register (mod 32):

\[
\text{sllv}\ \text{a1}, \text{a0}, \text{t0}
\]
Shift Right Logical

The logical form of right shift adds 0’s to the MSB.

\[ \text{sr}l \text{ } \$a0, \$a0, 1 \]

31 30  \[ \cdots \]  2 1 0

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Shift Right Arithmetic

The “arithmetic” form of right shift sign-extends the word by copying the MSB.

`sra $a0, $a0, 2`
Set on Less Than

\[
\text{slt} \; t0, \; t1, \; t2
\]

Set $t0$ to 1 if the contents of $t1 < t0$; 0 otherwise. $t1$ and $t2$ are treated as 32-bit signed two's complement numbers.

```c
int compare(int a,  // $a0
    int b,  // $a1
    unsigned c, // $a2
    unsigned d) // $a3
{
    int r = 0;  // $v0
    if (a < b) r += 42;
    if (c < d) r += 99;
    return r;
}
```
### Load and Store Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lb</code></td>
<td>Load byte</td>
<td></td>
</tr>
<tr>
<td><code>lbu</code></td>
<td>Load byte unsigned</td>
<td></td>
</tr>
<tr>
<td><code>lh</code></td>
<td>Load halfword</td>
<td></td>
</tr>
<tr>
<td><code>lhu</code></td>
<td>Load halfword unsigned</td>
<td></td>
</tr>
<tr>
<td><code>lw</code></td>
<td>Load word</td>
<td></td>
</tr>
<tr>
<td><code>lwl</code></td>
<td>Load word left</td>
<td>The MIPS is a load/store architecture: memory can only be transferred to/from registers.</td>
</tr>
<tr>
<td><code>lwr</code></td>
<td>Load word right</td>
<td></td>
</tr>
<tr>
<td><code>sb</code></td>
<td>Store byte</td>
<td>Other processors e.g., (x86) can also do arithmetic on memory contents.</td>
</tr>
<tr>
<td><code>sh</code></td>
<td>Store halfword</td>
<td></td>
</tr>
<tr>
<td><code>sw</code></td>
<td>Store word</td>
<td></td>
</tr>
<tr>
<td><code>swl</code></td>
<td>Store word left</td>
<td></td>
</tr>
<tr>
<td><code>swr</code></td>
<td>Store word right</td>
<td></td>
</tr>
</tbody>
</table>
Memory on the MIPS

Memory is byte-addressed. Each byte consists of eight bits:

```
 7 6 5 4 3 2 1 0
```

Bytes have non-negative integer addresses. Byte addresses on the 32-bit MIPS processor are 32 bits; 64-bit processors usually have 64-bit addresses.

```
 0:   7 6 5 4 3 2 1 0
 1:   7 6 5 4 3 2 1 0
 2:   7 6 5 4 3 2 1 0
  
2^{32} - 1: 7 6 5 4 3 2 1 0
```

4 Gb total
MIPS registers are 32 bits (4 bytes). Loading a byte into a register either clears the top three bytes or sign-extends them.

```
42: F0

lbu $t0, 42($0)

$t0: 000000F0
```

```
42: F0

lb $t0, 42($0)

$t0: FFFFFFFF0
```
Base Addressing in MIPS

There is only one way to refer to what address to load/store in MIPS: base + offset.

\[
\text{lb } \$t0, 34(\$t1)
\]

\[
\begin{align*}
\$t1: & \ 00000008 \quad \text{(base register)} \\
& \uparrow \\
& + \ 34 \quad \text{(immediate offset)} \\
& \downarrow \\
\text{lb } \$t0, 34(\$t1) \\
& \uparrow \\
42: & \ EF \\
& \downarrow \\
\$t0: & \ FFFFFFFFEEF \\
-32768 < \text{offset} < 32767
\end{align*}
\]
The Endian Question

MIPS can also load and store 4-byte words and 2-byte halfwords.

The * endian question: when you read a word, in what order do the bytes appear?

Little Endian: Intel, DEC, et al.

Big Endian: Motorola, IBM, Sun, et al.

MIPS can do either

SPIM adopts its host’s convention

### Big Endian

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing Endianness

.data # Directive: ‘‘this is data’’
myword:
  .word 0 # Define a word of data (=0)
.text # Directive: ‘‘this is program’’
main:
  la $t1, myword # pseudoinstruction: load address
  li $t0, 0x11
  sb $t0, 0($t1) # Store 0x11 at byte 0
  li $t0, 0x22
  sb $t0, 1($t1) # Store 0x22 at byte 1
  li $t0, 0x33
  sb $t0, 2($t1) # Store 0x33 at byte 2
  li $t0, 0x44
  sb $t0, 3($t1) # Store 0x44 at byte 3
  lw $t2, 0($t1) # 0x11223344 or 0x44332211?
  j $ra
Alignment

Word and half-word loads and stores must be aligned:
words must start at a multiple of 4 bytes;
halfwords on a multiple of 2.

Byte load/store has no such constraint.

```
lw $t0, 4($0)  # OK
lw $t0, 5($0)  # BAD: 5 mod 4 = 1
lw $t0, 8($0)  # OK
lw $t0, 12($0) # OK

lh $t0, 2($0)  # OK
lh $t0, 3($0)  # BAD: 3 mod 2 = 1
lh $t0, 4($0)  # OK
```
Jump and Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>j</code></td>
<td>Jump</td>
</tr>
<tr>
<td><code>jal</code></td>
<td>Jump and link</td>
</tr>
<tr>
<td><code>jr</code></td>
<td>Jump to register</td>
</tr>
<tr>
<td><code>jalr</code></td>
<td>Jump and link register</td>
</tr>
<tr>
<td><code>beq</code></td>
<td>Branch on equal</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>Branch on not equal</td>
</tr>
<tr>
<td><code>blez</code></td>
<td>Branch on less than or equal to zero</td>
</tr>
<tr>
<td><code>bgtz</code></td>
<td>Branch on greater than zero</td>
</tr>
<tr>
<td><code>bltz</code></td>
<td>Branch on less than zero</td>
</tr>
<tr>
<td><code>bgez</code></td>
<td>Branch on greater than or equal to zero</td>
</tr>
<tr>
<td><code>bltzal</code></td>
<td>Branch on less than zero and link</td>
</tr>
<tr>
<td><code>bgezal</code></td>
<td>Branch on greater than or equal to zero and link</td>
</tr>
</tbody>
</table>
Jumps

The simplest form,

```plaintext
j  mylabel
# ...
```

sends control to the instruction at `mylabel`. Instruction holds a 26-bit constant multiplied by four; top four bits come from current PC. Uncommon.

Jump to register sends control to a 32-bit absolute address in a register:

```plaintext
jr  $t3
```

Instructions must be four-byte aligned; the contents of the register must be a multiple of 4.
Jump and Link stores a return address in $ra for implementing subroutines:

```
jal  mysub
    #  Control resumes here after the jr
    #  ...

mysub:
    #  ...
    jr  $ra  #  Jump back to caller
```

`jalr` is similar; target address supplied in a register.
Branches

Used for conditionals or loops. E.g., “send control to myloop if the contents of $t0 is not equal to the contents of $t1.”

myloop:
    # ...

    bne $t0, $t1, myloop
    # ...

beq is similar “branch if equal”

A “jump” supplies an absolute address; a “branch” supplies an offset to the program counter.

On the MIPS, a 16-bit signed offset is multiplied by four and added to the address of the next instruction.
Another family of branches tests a single register:

```assembly
bgez $t0, myelse  # Branch if $t0 positive
# ...
```

```assembly
myelse:
  # ...
```

Others in this family:

- `blez` Branch on less than or equal to zero
- `bgtz` Branch on greater than zero
- `bltz` Branch on less than zero
- `bltzal` Branch on less than zero and link
- `bgez` Branch on greater than or equal to zero
- `bgezal` Branch on greater than or equal to zero and link

“and link” variants also (always) put the address of the next instruction into $ra, just like `jal`.
Other Instructions

**syscall** causes a system call exception, which the OS catches, interprets, and usually returns from.

SPIM provides simple services: printing and reading integers, strings, and floating-point numbers, sbrk() (memory request), and exit().

```assembly
# prints "the answer = 5"
.data
str:
  .asciiz "the answer = 
.text
li $v0, 4  # system call code for print_str
la $a0, str # address of string to print
syscall  # print the string

li $v0, 1  # system call code for print_int
li $a0, 5  # integer to print
syscall    # print it
```
## Other Instructions

<table>
<thead>
<tr>
<th><strong>Exception Instructions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tge</code> <code>tlt</code> ... Conditional traps</td>
</tr>
<tr>
<td><code>break</code>                    Breakpoint trap, for debugging</td>
</tr>
<tr>
<td><code>eret</code>                     Return from exception</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Multiprocessor Instructions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ll</code> <code>sc</code> <code>sync</code> Load linked/store conditional for atomic operations</td>
</tr>
<tr>
<td>Read/Write fence: wait for all memory loads/stores</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Coprocessor 0 Instructions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lwr</code> <code>lw1</code> ... Cache control</td>
</tr>
<tr>
<td><code>tlbr</code> <code>tblwi</code> ... TLB control (virtual memory)</td>
</tr>
<tr>
<td>...                          Many others (data movement, branches)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Floating-point Coprocessor Instructions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add.d</code> <code>sub.d</code> ... Arithmetic and other functions</td>
</tr>
<tr>
<td><code>lwc1</code> <code>swc1</code> ... Load/store to (32) floating-point registers</td>
</tr>
<tr>
<td><code>bct1t</code> ... Conditional branches</td>
</tr>
</tbody>
</table>
## Instruction Encoding

### Register-type: `add, sub, xor, ...`

<table>
<thead>
<tr>
<th>op:6</th>
<th>rs:5</th>
<th>rt:5</th>
<th>rd:5</th>
<th>shamt:5</th>
<th>funct:6</th>
</tr>
</thead>
</table>

### Immediate-type: `addi, subi, beq, ...`

<table>
<thead>
<tr>
<th>op:6</th>
<th>rs:5</th>
<th>rt:5</th>
<th>imm:16</th>
</tr>
</thead>
</table>

### Jump-type: `j, jal ...`

<table>
<thead>
<tr>
<th>op:6</th>
<th>addr:26</th>
</tr>
</thead>
</table>
Register-type Encoding Example

<table>
<thead>
<tr>
<th>op:6</th>
<th>rs:5</th>
<th>rt:5</th>
<th>rd:5</th>
<th>shamt:5</th>
<th>funct:6</th>
</tr>
</thead>
</table>

```
add  $t0, $s1, $s2
```

```
add encoding from the MIPS instruction set reference:
```

<table>
<thead>
<tr>
<th>SPECIAL 000000</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0 00000</th>
<th>ADD 100000</th>
</tr>
</thead>
</table>

Since $t0 is register 8; $s1 is 17; and $s2 is 18,

```
000000 10001 10010 01000 00000 100000
```
Register-type Shift Instructions

<table>
<thead>
<tr>
<th>op:6</th>
<th>rs:5</th>
<th>rt:5</th>
<th>rd:5</th>
<th>shamt:5</th>
<th>funct:6</th>
</tr>
</thead>
</table>

`sra $t0, $s1, 5`

`sra` encoding from the MIPS instruction set reference:

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>0</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>SRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
<td>0000</td>
<td>10010</td>
<td>01000</td>
<td>00101</td>
</tr>
</tbody>
</table>

Since $t0$ is register 8 and $s1$ is 17,

| 000000 | 00000 | 10010 | 01000 | 00101 | 000011 |
Immediate-type Encoding Example

```
| op:6 | rs:5 | rt:5 | imm:16 |
```

```
addiu $t0, $s1, -42
```

```
addiu encoding from the MIPS instruction set reference:
```
```
ADDIU 001001 10001 01000 1111 1111 1101 0110
```

Since $t0 is register 8 and $s1 is 17,

```
001001 10001 01000 1111 1111 1101 0110
```
Jump-Type Encoding Example

\[
\begin{array}{|c|c|}
\hline
\text{op:6} & \text{addr:26} \\
\hline
\end{array}
\]

\textit{jal} 0x5014

\textit{jal} encoding from the MIPS instruction set reference:

\[
\begin{array}{|c|c|}
\hline
\text{JAL} & \text{instr\_index} \\
\hline
000011 & \\
\hline
\end{array}
\]

Instruction index is a word address

\[
\begin{array}{|c|c|}
\hline
000011 & 00 0000 0000 0001 0100 0000 0101 \\
\hline
\end{array}
\]
<table>
<thead>
<tr>
<th>Assembler Pseudoinstructions</th>
<th>b label → beq $0, $0, label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch always</td>
<td>beqz s, label → beq s, $0, label</td>
</tr>
<tr>
<td>Branch if equal zero</td>
<td>bge s, t, label → slt $1, s, t</td>
</tr>
<tr>
<td>Branch greater or equal</td>
<td>bgeu s, t, label → sltu $1, s, t</td>
</tr>
<tr>
<td>Branch greater than</td>
<td>bgt s, t, label → slt $1, t, s</td>
</tr>
<tr>
<td>Branch greater than</td>
<td>bgtu s, t, label → sltu $1, t, s</td>
</tr>
<tr>
<td>Branch less than</td>
<td>blt s, t, label → slt $1, s, t</td>
</tr>
<tr>
<td>Branch less than unsigned</td>
<td>bltu s, t, label → sltu $1, s, t</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assembler Pseudoinstructions

Load immediate \( \text{li} \ d, j \rightarrow \text{ori} \ d, \$0, j \)
0 \( \leq j \leq 65535 \)

Load immediate \( \text{li} \ d, j \rightarrow \text{addiu} \ d, \$0, j \)
\(-32768 \leq j < 0 \)

Load immediate \( \text{li} \ d, j \rightarrow \text{liu} \ d, \text{hi16}(j) \)

Move \( \text{move} \ d, s \rightarrow \text{or} \ d, s, \$0 \)

Multiply \( \text{mul} \ d, s, t \rightarrow \text{mult} \ s, t \)
\( \text{mflo} \ d \)

Negate unsigned \( \text{negu} \ d, s \rightarrow \text{subu} \ d, \$0, s \)

Set if equal \( \text{seq} \ d, s, t \rightarrow \text{xor} \ d, s, t \)

Set if greater or equal \( \text{sgge} \ d, s, t \rightarrow \text{sltiu} \ d, d, 1 \)

Set if greater or equal unsigned \( \text{sggeu} \ d, s, t \rightarrow \text{sltu} \ d, s, t \)

Set if greater than \( \text{sgt} \ d, s, t \rightarrow \text{slt} \ d, t, s \)
Expressions

Initial expression:

\[ x + y + z \times (w + 3) \]

Reordered to minimize intermediate results; fully parenthesized to make order of operation clear.

\[ (((w + 3) \times z) + y) + x \]

```
addiu $t0, $a0, 3       # w: $a0
mul $t0, $t0, $a3       # x: $a1
addu $t0, $t0, $a2     # y: $a2
addu $t0, $t0, $a1     # z: $a3
```

Consider an alternative:

\[ (x + y) + ((w + 3) \times z) \]

```
addu $t0, $a1, $a2
addiu $t1, $a0, 3       # Need a second temporary
mul $t1, $t1, $a3
addu $t0, $t0, $t1
```
Conditionals

if \((x + y) < 3\)
  \(x = x + 5\);
else
  \(y = y + 4\);

\[
\begin{align*}
\text{addu} & \quad \$t0, \quad \$a0, \quad \$a1 \quad \# \quad x + y \\
\text{slti} & \quad \$t0, \quad \$t0, \quad 3 \quad \# \quad (x+y)<3 \\
\text{beq} & \quad \$t0, \quad \$0, \quad L0 \quad \# \quad \text{if false}
\end{align*}
\]

\[
\begin{align*}
\text{addiu} & \quad \$a0, \quad \$a0, \quad 5 \quad \# \quad x += 5 \\
\text{beq} & \quad L1 \quad \# \quad \text{skip else}
\end{align*}
\]

\[
\begin{align*}
L0: & \quad \text{addiu} \quad \$a1, \quad \$a1, \quad 4 \quad \# \quad y += 4
\end{align*}
\]

L1:
Do-While Loops

Post-test loop: body always executes once

\[
\begin{align*}
a &= 0; \\
b &= 0; \\
do \{ & \\
    a &= a + b; \quad \text{L1:} \\
    b &= b + 1; \\
\} & \text{ while } (b \neq 10); \\
\end{align*}
\]

\[
\begin{align*}
\text{move } &\quad \text{ $a0$, $0$ } \# a = 0 \\
\text{move } &\quad \text{ $a1$, $0$ } \# b = 0 \\
\text{li } &\quad \text{ $t0$, 10 } \# \text{ load constant} \\
\text{addu } &\quad \text{ $a0$, $a0$, $a1$ } \# a = a + b \\
\text{addiu } &\quad \text{ $a1$, $a1$, 1 } \# b = b + 1 \\
\text{bne } &\quad \text{ $a1$, $t0$, L1 } \# b \neq 10? \\
\end{align*}
\]
While Loops

Pre-test loop: body may never execute

```plaintext
a = 0;
b = 0;
while (b != 10) {
    a = a + b;
b = b + 1;
}
```

```plaintext
move $a0, $0  # a = 0
move $a1, $0  # b = 0
li $t0, 10
b L3  # test first
L2:
addu $a0, $a0, $a1  # a = a + b
addiu $a1, $a1, 1  # b = b + 1
L3:
bne $a1, $t0, L2  # b != 10?
```
For Loops

“Syntactic sugar” for a while loop

```
for (a = b = 0 ; b != 10 ; b++)
    a += b;
```
is equivalent to

```
a = b = 0;
while (b != 10) {
    a = a + b;
    b = b + 1;
}
```
moved
```
move $a1, $0  # b = 0
move $a0, $a1  # a = b
li $t0, 10
b L3         # test first
L2:
    addu $a0, $a0, $a1  # a = a + b
    addiu $a1, $a1, 1  # b = b + 1
L3:
    bne $a1, $t0, L2  # b != 10?
```
Arrays

```c
int a[5];

void main() {
}
```

```assembly
.comm a, 20 # Allocate 20 .text # Program next main:
main:
    la $t0, a # Address of a
    li $t1, 3
    sw $t1, 0($t0) # a[0]
    sw $t1, 4($t0) # a[1]
    sw $t1, 8($t0) # a[2]
    sw $t1, 12($t0) # a[3]
    sw $t1, 16($t0) # a[4]
    lw $t1, 8($t0) # a[2]
    sll $t1, $t1, 2 # * 4
    sw $t1, 4($t0) # a[1]
    lw $t1, 16($t0) # a[4]
    sll $t1, $t1, 1 # * 2
    sw $t1, 12($t0) # a[3]
    jr $ra
```
Summing the contents of an array

```c
int i, s, a[10];
for (s = i = 0 ; i < 10 ; i++)
    s = s + a[i];
```

```assembly
move $a1, $0  # i = 0
move $a0, $a1  # s = 0
li   $t0, 10
la   $t1, a   # base address of array
b     L2
L1:
    sll  $t3, $a1, 2  # i * 4
    addu $t3, $t1, $t3 # &a[i]
    lw   $t3, 0($t3)  # fetch a[i]
    addu $a0, $a0, $t3 # s += a[i]
    addiu $a1, $a1, 1
L2:
    sltu $t2, $a1, $t0 # i < 10?
    bne  $t2, $0, L1
```
### Summing the contents of an array

```c
int s, *i, a[10];
for (s=0, i = a+9 ; i >= a ; i--)
  s += *i;

move $a0, $0  # s = 0
la   $t0, a   # &a[0]
addiu $t1, $t0, 36  # i = a + 9
b    L2
L1:
  lw   $t2, 0($t1)  # *i
  addu $a0, $a0, $t2  # s += *i
  addiu $t1, $t1, -4  # i++
L2:
  sltu $t2, $t1, $t0  # i < a
  beq $t2, $0, L1
```
Strings: Hello World in SPIM

# For SPIM: "Enable Mapped I/O" must be set
# under Simulator/Settings/MIPS

.data
hello:
    .asciiz "Hello World!\n"

.text
main:
    la    $t1, 0xffffff0000 # I/O base address
    la    $t0, hello

wait:
    lw    $t2, 8($t1)      # Read Transmitter control
    andi $t2, $t2, 0x1    # Test ready bit
    beq  $t2, $0, wait

    lbu   $t2, 0($t0)      # Read the byte
    beq  $t2, $0, done    # Check for terminating 0

    sw    $t2, 12($t1)     # Write transmit data

    addiu $t0, $t0, 1     # Advance to next character

    b      wait

done:
    jr    $ra
Hello World in SPIM: Memory contents

```
[00400024] 3c09ffff  lui  $9, -1
[00400028] 3c081001  lui  $8, 4097 [hello]
[0040002c] 8d2a0008  lw   $10, 8($9)
[00400030] 314a0001  andi $10, $10, 1
[00400034] 1140ffe6  beq  $10, $0, -8 [wait]
[00400038] 910a0000  lbu  $10, 0($8)
[0040003c] 11400004  beq  $10, $0, 16 [done]
[00400040] ad2a000c  sw   $10, 12($9)
[00400044] 25080001  addiu $8, $8, 1
[00400048] 0401fft9  bgez  $0 -28 [wait]
[0040004c] 03e00008  jr   $31

[10010000] 6c6c6548 6f57206f  Hello  Wo
[10010008] 21646c72 0000000a  rld ! . . . .
```
## ASCII

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td></td>
<td>NUL</td>
<td>\0</td>
<td>DLE</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>‘</td>
<td>p</td>
</tr>
<tr>
<td>1:</td>
<td></td>
<td>SOH</td>
<td>!</td>
<td>DC1</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td>STX</td>
<td>&quot;</td>
<td>DC2</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td>ETX</td>
<td>#</td>
<td>DC3</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td>EOT</td>
<td>$</td>
<td>DC4</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td>ENQ</td>
<td>%</td>
<td>NAK</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>6:</td>
<td></td>
<td>ACK</td>
<td>&amp;</td>
<td>SYN</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>7:</td>
<td></td>
<td>BEL</td>
<td>\a</td>
<td>ETB</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>8:</td>
<td></td>
<td>BS</td>
<td>\b</td>
<td>CAN</td>
<td>(</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>9:</td>
<td></td>
<td>HT</td>
<td>\t</td>
<td>EM</td>
<td>)</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>A:</td>
<td></td>
<td>LF</td>
<td>\n</td>
<td>SUB</td>
<td>*</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>B:</td>
<td></td>
<td>VT</td>
<td>\v</td>
<td>ESC</td>
<td>+</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td>C:</td>
<td></td>
<td>FF</td>
<td>\f</td>
<td>FS</td>
<td>,</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td></td>
<td>CR</td>
<td>\r</td>
<td>GS</td>
<td>-</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>E:</td>
<td></td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>F:</td>
<td></td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>
Subroutines

a.k.a. procedures, functions, methods, et al.
Code that can run by itself, then *resume whatever invoked it*.

Exist for three reasons:

- **Code reuse**
  Recurring computations aside from loops
  Function libraries

- **Isolation/Abstraction**
  Think Vegas:
  What happens in a function stays in the function.

- **Enabling Recursion**
  Fundamental to divide-and-conquer algorithms
Calling Conventions

# Call mysub: args in $a0,...,$a3
jal mysub
# Control returns here
# Return value in $v0 & $v1
# $s0,...,$s7, $gp, $sp, $fp, $ra unchanged
# $a0,...,$a3, $t0,....,$t9 possibly clobbered

mysub:   # Entry point: $ra holds return address
    # First four args in $a0, $a1, ..., $a3

    # ... body of the subroutine ...

    # $v0, and possibly $v1, hold the result
    # $s0,....,$s7 restored to value on entry
    # $gp, $sp, $fp, and $ra also restored
jr $ra     # Return to the caller
The Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFFC</td>
<td>0x32640128</td>
</tr>
<tr>
<td>0x7FFFFFF8</td>
<td>0xCAFEB0E</td>
</tr>
<tr>
<td>0x7FFFFFF4</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>0x7FFFFFF0</td>
<td>0xCODEFACE</td>
</tr>
<tr>
<td>0x7FFFFFFEC</td>
<td></td>
</tr>
</tbody>
</table>

$sp$ grows down
void move(int src, int tmp, int dst, int n)
{
    if (n) {
        move(src, dst, tmp, n-1);
        printf("%d->%d\n", src, dst);
        move(tmp, src, dst, n-1);
    }
}
Allocate 24 stack bytes: multiple of 8 for alignment
Check whether n == 0
Save $ra, $s0, ..., $s3 on the stack
hmove:
  addiu $sp, $sp, -24
  beq $a3, $0, L1
  sw $ra, 0($sp)
  sw $s0, 4($sp)
  sw $s1, 8($sp)
  sw $s2, 12($sp)
  sw $s3, 16($sp)
  move $s0, $a0
  move $s1, $a1
  move $s2, $a2
  addiu $s3, $a3, -1

  Save src in $s0
  Save tmp in $s1
  Save dst in $s2
  Save n – 1 in $s3
hmove:

```assembly
addiu $sp, $sp, -24
beq $a3, $0, L1
sw $ra, 0($sp)
sw $s0, 4($sp)
sw $s1, 8($sp)
sw $s2, 12($sp)
sw $s3, 16($sp)
move $s0, $a0
move $s1, $a1
move $s2, $a2
addiu $s3, $a3, -1
move $a1, $s2
move $a2, $s1
move $a3, $s3
jal hmove
```

Call

hmove(src, dst, tmp, n−1)
Print src -> dst
hmove:
    addiu $sp, $sp, -24
    beq $a3, $0, L1
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)
    move $s0, $a0
    move $s1, $a1
    move $s2, $a2
    addiu $s3, $a3, -1
    move $a1, $s2
    move $a2, $s1
    move $a3, $s3
    jal hmove

Call
    hmove(tmp, src, dst, n−1)

li $v0, 1 # print_int
move $a0, $s2
syscall
li $v0,4 # print_str
la $a0, newline
syscall
move $a0, $s1
move $a1, $s0
move $a2, $s2
move $a3, $s3
jal hmove

li $v0, 1 # print_int
move $a0, $s0
syscall
li $v0, 4 # print_str
la $a0, arrow
syscall
hmove:
    addiu $sp, $sp, -24
    beq $a3, $0, L1
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)
    move $s0, $a0
    move $s1, $a1
    move $s2, $a2
    addiu $s3, $a3, -1
    move $s0, $a0
    move $s1, $a1
    move $s2, $a2
    addiu $s3, $a3, -1
    move $a1, $s2
    move $a2, $s1
    move $a3, $s3
    jal hmove
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)

    li $v0, 1 # print_int
    move $a0, $s2
    syscall
    li $v0,4 # print_str
    la $a0, newline
    syscall
    move $a0, $s1
    move $a1, $s0
    move $a2, $s2
    move $a3, $s3
    jal hmove
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)

    li $v0, 1 # print_int
    move $a0, $s2
    syscall
    li $v0,4 # print_str
    la $a0, newline
    syscall

    Restore variables
hmove:
  addiu $sp, $sp, -24
  beq $a3, $0, L1
  sw $ra, 0($sp)
  sw $s0, 4($sp)
  sw $s1, 8($sp)
  sw $s2, 12($sp)
  sw $s3, 16($sp)
  move $s0, $a0
  move $s1, $a1
  move $s2, $a2
  addiu $s3, $a3, -1
  move $a1, $s2
  move $a2, $s1
  move $a3, $s3
  jal hmove

li   $v0, 1 # print_int
move $a0, $s2
syscall
li   $v0,4 # print_str
la   $a0, newline
syscall
move $a0, $s1
move $a1, $s0
move $a2, $s2
move $a3, $s3
jal hmove
lw   $ra, 0($sp)
lw   $s0, 4($sp)
lw   $s1, 8($sp)
lw   $s2, 12($sp)
lw   $s3, 16($sp)

L1:
  addiu $sp, $sp, 24 # free
  jr $ra              # return

.data
ra    .asciiz "->"
arrow: .asciiz "\n"
newline: .asciiz "\n"
Factorial Example

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

fact:
```
addiu $sp, $sp, -8   # allocate 2 words on stack
sw $ra, 4($sp)   # save return address
sw $a0, 0($sp)   # and n
slti $t0, $a0, 1   # n < 1?
beq $t0, $0, L1   # Yes, return 1
li $v0, 1
addiu $sp, $sp, 8   # Pop 2 words from stack
jr $ra   # return

L1:
addiu $a0, $a0, -1   # No: compute n-1
jal fact   # recurse (result in $v0)
lw $a0, 0($sp)   # Restore n and
lw $ra, 4($sp)   # return address
mul $v0, $a0, $v0   # Compute n * fact(n-1)
addiu $sp, $sp, 8   # Pop 2 words from stack
jr $ra   # return
```
Differences in Other ISAs

More or fewer general-purpose registers (E.g., Itanium: 128; 6502: 3)

Arithmetic instructions affect condition codes (e.g., zero, carry); conditional branches test these flags

Registers that are more specialized (E.g., x86)

More addressing modes (E.g., x86: 6; VAX: 20)

Arithmetic instructions that also access memory (E.g., x86; VAX)

Arithmetic instructions on other data types (E.g., bytes and halfwords)

Variable-length instructions (E.g., x86; ARM)

Predicated instructions (E.g., ARM, VLIW)

Single instructions that do much more (E.g., x86 string move, procedure entry/exit)