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

00010000100001010000000000000111
00000000101010000010000010100110
00010100010000000000000000000011
00000000010100000101000001000111
0000100000000011111111111111100
00000000010000101001000000010011
0000000000000011111111111110101
00000000000000100000100000010001
00000000000000000000111111111100
00000000000000000000000000100001
00000011111000000000000000100000
Machine, Assembly, and C Code

```
0001000010000101000000000000000111 beq $4, $5, 28
00000000101001000001000001010100 slt $2, $5, $4
000101000100000000000000000000011 bne $2, $0, 12
00000000101001000010100001000111 subu $5, $5, $4
00000100000000011111111111111100 bgez $0 -16
000000001000010100100000000100011 subu $4, $4, $5
0000010000000000011111111111111101 bgez $0 -24
00000000000010000100000001000010 addu $2, $0, $4
000000111110000000000000000001000 jr $31
```

"Humans"
Machine, Assembly, and C Code

```
000100001000010100000000000001111
00000000101001000001000010101010
00010100100000000000000001111111
00000001010010000101000001000111
000001000000000001111111111111100
00000000100001010010000000100011
00000100000000011111111111111010
00000000000001000001000001000001
00000011111000000000000000001000

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
```
### Machine, Assembly, and C Code

```
000100001000010100000000000000111
000000001010010000000000000000011
00101000100000000000000000000011
000000001010010000000000000000011
000001000000000000011111111111110
00000000100001010010000000100011
00000000100001010010000000100001
00000111110000000000001000
00000000000001000001000000100001
00000011111000000000000000000001000
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
```

```c
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

“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*

MIPS developed at Stanford by Hennessey et al. MIPS Computer Systems founded 1984. SGI acquired MIPS in 1992; spun it out in 1998 as MIPS Technologies. Now, mostly an embedded core competing with ARM. In many wireless WiFi routers.
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 $a$ \[ a > b \text{ here} \]
5. Go to step 2
6. Subtract $a$ from $b$ \[ a < b \text{ here} \]
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)
  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
```

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

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 العامة gcd # and repeat

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

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

Comments

- Introduce a comment
- Add comments to explain the code
- Use labels for easier navigation

```
# if a = b, go to exit
# Is b > a?
# Yes, goto .L1
# Subtract b from a (b < a)
# and repeat
# Subtract a from b (a < b)
# and repeat
# return a
# Return to caller
```
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:

```assembly
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
```

.L1:

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

.L2:

```assembly
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>$zero</td>
<td>0</td>
<td>Constant zero</td>
<td>= 0</td>
</tr>
<tr>
<td>$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

## Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add $1$, $2$, $3$</td>
</tr>
<tr>
<td>addu</td>
<td>Add unsigned</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract</td>
</tr>
<tr>
<td>subu</td>
<td>Subtract unsigned</td>
</tr>
<tr>
<td>slt</td>
<td>Set on less than $a &lt; b$</td>
</tr>
<tr>
<td>sltu</td>
<td>Set on less than unsigned</td>
</tr>
<tr>
<td>and</td>
<td>AND</td>
</tr>
<tr>
<td>or</td>
<td>OR</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>nor</td>
<td>NOR</td>
</tr>
</tbody>
</table>

## Arithmetic (immediate)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>Add immediate $1$, $2$, $4$, $2$</td>
</tr>
<tr>
<td>addiu</td>
<td>Add immediate unsigned</td>
</tr>
<tr>
<td>slti</td>
<td>Set on l. t. immediate</td>
</tr>
<tr>
<td>sltiu</td>
<td>Set on less than unsigned</td>
</tr>
<tr>
<td>andi</td>
<td>AND immediate</td>
</tr>
<tr>
<td>ori</td>
<td>OR immediate</td>
</tr>
<tr>
<td>xori</td>
<td>Exclusive OR immediate</td>
</tr>
<tr>
<td>lui</td>
<td>Load upper immediate</td>
</tr>
</tbody>
</table>

## Shift Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>Shift left logical</td>
</tr>
<tr>
<td>srl</td>
<td>Shift right logical</td>
</tr>
<tr>
<td>sra</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>sllv</td>
<td>Shift left logical variable</td>
</tr>
<tr>
<td>srlv</td>
<td>Shift right logical variable</td>
</tr>
<tr>
<td>srav</td>
<td>Shift right arith. variable</td>
</tr>
</tbody>
</table>

## Multiply/Divide

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>Multiply</td>
</tr>
<tr>
<td>multu</td>
<td>Multiply unsigned</td>
</tr>
<tr>
<td>div</td>
<td>Divide</td>
</tr>
<tr>
<td>divu</td>
<td>Divide unsigned</td>
</tr>
<tr>
<td>mfhi</td>
<td>Move from HI</td>
</tr>
<tr>
<td>mthi</td>
<td>Move to HI</td>
</tr>
<tr>
<td>mflo</td>
<td>Move from LO</td>
</tr>
<tr>
<td>mtlo</td>
<td>Move to LO</td>
</tr>
</tbody>
</table>

\[ \underbrace{\text{32}}_{\text{32}} \times \underbrace{\text{32}}_{\text{32}} = \underbrace{\text{32}}_{\text{32}} \]
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:

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

“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

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

\[
\begin{array}{cccccccc}
  a & b & c & f & g & h & i & j \\
  $s0 & $s1 & $s2 & $s3 & $s4 & $s5 & $s6 & $s7 \\
\end{array}
\]

“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 identical.

Use these
Bitwise Logical Operator Example

\[
\begin{array}{c}
\text{li }$t0, 0xFF00FF00 \# "Load immediate"\\
\text{li }$t1, 0xF0F0F0F0 \# "Load immediate"\\
\text{nor }$t2, $t0, $t1 \# Puts 0x000F000F in $t2 \\[=\text{print_int}\]
\text{li }$v0, 1 \# print_int \\[=\text{print_int}\]
\text{move }$a0, $t2 \# print contents of $t2
\text{syscall}
\end{array}
\]
Immediate Computational Instructions

Example:

\[
\text{addiu } \$t0, \$t1, 42
\]

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

In general,

\[
\text{operation } R_D, R_S, I
\]

“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.,

\[
\text{ori } \$t0, \$0, 42
\]

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 \(0xC0DEFACE\):

\[
\begin{align*}
\text{lui } \$t0, 0xC0DE \\
\text{ori } \$t0, \$t0, 0xFACE
\end{align*}
\]

The assembler automatically expands the **li** pseudo-instruction into such an instruction sequence

\[
\begin{align*}
\text{li } \$t1, 0xCAFE0B0E &\rightarrow \text{lui } \$t1, 0xCAFE \\
\text{ori } \$t1, \$t1, 0x0B0E
\end{align*}
\]
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; // 1st
    c = c * d + a; // 1st
    a = a / c; // 2nd
    b = b % a; // 2nd
    c = c / d; // 2nd
    d = d % c;

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

```
multdiv:
mult $a0,$a1 # a * b (H1,H0) = $20x46
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 $a0,$a2 # a / c
mfhi $a1 # b = b%a
divu $a2,$a3 # c / d
mflo $a2 # c = c/d
addu $t2,$a0,$a1 # a + b
addu $t2,$t2,$a2 # (a+b) + c
divu $a3,$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:

\[
sll \; a0, \; a0, \; 1
\]

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

\[
sllv \; a1, \; a0, \; t0
\]
Shift Right Logical

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

```
srl $a0, $a0, 1
```

```
31 30  ...  2 1 0
```

```
0
```

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

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

Set $t0$ to 1 if the contents of $t1 \lt t2$; 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/Store Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lb</code></td>
<td>Load byte</td>
<td>Byte</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>Halfword</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>Word</td>
</tr>
<tr>
<td><code>lwl</code></td>
<td>Load word left</td>
<td></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></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>

The MIPS is a load/store architecture: data must be moved into registers for computation.

Other architectures e.g., (x86) allow arithmetic directly on data in memory.
Memory on the MIPS

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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

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.

\[
\begin{array}{cccccccc}
0: & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1: & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
2: & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\vdots & & & & & & & & \\
2^{32} - 1: & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{array}
\]

4 Gb total
Base Addressing in MIPS

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

\[
\text{lb } 
\begin{array}{c}
\text{offset} \\
\end{array} 
\begin{array}{c}
t1: 00000008 \\
\end{array} 
\begin{array}{c}
\text{(base register)} \\
\end{array} 
\begin{array}{c}
+ \\
\end{array} 
\begin{array}{c}
34 \text{ (immediate offset)} \\
\end{array} 
\begin{array}{c}
42: \\
\end{array} 
\begin{array}{c}
\text{EF} \\
\end{array} 
\begin{array}{c}
\text{F} \\
\text{FFFFFFFEF} \\
\end{array} 
\begin{array}{c}
\text{	extit{negative}} \\
\end{array} 
\begin{array}{c}
\text{sign-extended} \\
\end{array} 
\begin{array}{c}
\text{$t0: FFFFFFFFEEF}$ \\
\end{array} 
\begin{array}{c}
-32768 < \text{offset} < 32767 \\
\end{array}
\]
MIPS registers are 32 bits (4 bytes). Loading a byte into a register either clears the top three bytes or sign-extends them.

\[
\begin{align*}
42: & \quad \text{F0} \\
\text{lbu } & \quad \text{t0, 42($0)} \\
\text{t0:} & \quad 000000\text{F0} \quad \text{unsigned} = 0 \text{ extend}
\end{align*}
\]

\[
\begin{align*}
42: & \quad \text{F0} \\
\text{lb } & \quad \text{t0, 42($0)} \\
\text{t0:} & \quad \text{FFFFFF0} \quad \text{signed} = \text{sign-extended}
\end{align*}
\]
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
Testing Endianness

.data
myword:

.word 0

.text
main:
la $t1, myword
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.
## Jump and Branch Instructions

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

The simplest form, `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:

```
jr $t3
```

Instructions must be four-byte aligned; the contents of the register must be a multiple of 4.
Jump and Link

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
    # ...
```

`bne` 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.
Branches

Another family of branches tests a single register:

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

myelse:

```assembly
# ...
```

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

#### Exception Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>tge tlt ...</td>
<td>Conditional traps</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>Breakpoint trap, for debugging</td>
<td></td>
</tr>
<tr>
<td>eret</td>
<td>Return from exception</td>
<td></td>
</tr>
</tbody>
</table>

#### Multiprocessor Instructions

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

#### Coprocessor 0 Instructions (System Mgmt)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwr lw1</td>
<td>Cache control</td>
</tr>
<tr>
<td>tlbr tblwi</td>
<td>TLB control (virtual memory)</td>
</tr>
<tr>
<td>...</td>
<td>Many others (data movement, branches)</td>
</tr>
</tbody>
</table>

#### Floating-point Coprocessor Instructions

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

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

| op:6 | rs:5 | rt:5 | rd:5 | shamt:5 | funct:6 |

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

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

Jump-type: j, jal...

| op:6 | addr:26 |

32-bit

32

add $d, $s, $t

addi $t, $s, 42

Immediate value

12-bits

addi or bne
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
```

3 reg

Add encoding from the MIPS instruction set reference:

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>100000</td>
</tr>
</tbody>
</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>000000</th>
<th>000000</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>SRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

$\text{addiu } $t0, $s1, -42$

$\text{addiu}$ encoding from the MIPS instruction set reference:

<table>
<thead>
<tr>
<th>ADDIU</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001001</td>
<td>10001</td>
<td>01000</td>
<td>1111 1111 1101 0110</td>
</tr>
</tbody>
</table>

Since $t0$ is register 8 and $s1$ is 17,
Jump-Type Encoding Example

jal 0x5014

jal encoding from the MIPS instruction set reference:

<table>
<thead>
<tr>
<th>JAL</th>
<th>instr_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

Instruction index is a word address:

| 000011 | 00 0000 0000 0001 0100 0000 0101 |

PC is word aligned
## Assembler Pseudoinstructions

<table>
<thead>
<tr>
<th>Branch condition</th>
<th>Pseudoinstruction</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>b label</td>
<td>beq $0, $0, label</td>
</tr>
<tr>
<td>If equal zero</td>
<td>beqz s, label</td>
<td>beq s, $0, label</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>bge s, t, label</td>
<td>slt $1, s, t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>beq $1, $0, label</td>
</tr>
<tr>
<td>Greater or equal unsigned</td>
<td>bgeu s, t, label</td>
<td>sltu $1, s, t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>beq $1, $0, label</td>
</tr>
<tr>
<td>Greater than</td>
<td>bgt s, t, label</td>
<td>slt $1, t, s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne $1, $0, label</td>
</tr>
<tr>
<td>Greater than unsigned</td>
<td>bgtu s, t, label</td>
<td>sltu $1, t, s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne $1, $0, label</td>
</tr>
<tr>
<td>Less than</td>
<td>blt s, t, label</td>
<td>slt $1, s, t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne $1, $0, label</td>
</tr>
<tr>
<td>Less than unsigned</td>
<td>bltu s, t, label</td>
<td>sltu $1, s, t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne $1, $0, label</td>
</tr>
<tr>
<td>Assembler Pseudoinstructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0 \leq j \leq 65535$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{li} \ d, \ j \quad \rightarrow \ \textit{ori} \ d, \ 0, \ j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-32768 \leq j &lt; 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{li} \ d, \ j \quad \rightarrow \ \textit{addiu} \ d, \ 0, \ j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{li} \ d, \ j \quad \rightarrow \ \textit{liu} \ d, \ \text{hi16}(j) \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ \textit{ori} \ d, \ d, \ \text{lo16}(j) \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{move} \ d, \ s \quad \rightarrow \ \textit{or} \ d, \ s, \ 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the result will fit in 32 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{mul} \ d, \ s, \ t \quad \rightarrow \ \textit{mult} \ s, \ t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ \textit{mflo} \ d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{negu} \ d, \ s \quad \rightarrow \ \textit{subu} \ d, \ 0, \ s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set if equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{seq} \ d, \ s, \ t \quad \rightarrow \ \textit{xor} \ d, \ s, \ t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ \textit{sltiu} \ d, \ d, \ 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set if greater or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{sge} \ d, \ s, \ t \quad \rightarrow \ \textit{slt} \ d, \ s, \ t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ \textit{xori} \ d, \ d, \ 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set if greater or equal unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{sgeu} \ d, \ s, \ t \quad \rightarrow \ \textit{sltu} \ d, \ s, \ t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ \textit{xori} \ d, \ d, \ 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set if greater than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{sgt} \ d, \ s, \ t \quad \rightarrow \ \textit{slt} \ d, \ t, \ s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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
mul $t0, $t0, $a3
addu $t0, $t0, $a2
addu $t0, $t0, $a1
```

Consider an alternative:

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

```
addu $t0, $a1, $a2
addiu $t1, $a0, 3
mul $t1, $t1, $a3
addu $t0, $t0, $t1
```

Assumption

\(w + 3\)

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

\(x + y\)

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

\(w + 3\)

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

\(x + y\)

\((x + y) + ((w + 3) \times z)\)
Conditionals

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

addu \$t0, \$a0, \$a1 \# \(x + y\)
slti \$t0, \$t0, 3 \# \((x+y) < 3\)
beq \$t0, \$0, ELSE
addiu \$a0, \$a0, 5 \# \(x += 5\)
b DONE

ELSE:
addiu \$a1, \$a1, 4 \# \(y += 4\)
DONE:
Do-While Loops

Post-test loop: body always executes once

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

```assembly
move $a0, $0  # a = 0
move $a1, $0  # b = 0
li $t0, 10    # load constant

TOP:
addu $a0, $a0, $a1  # a = a + b
addiu $a1, $a1, 1  # b = b + 1
bne $a1, $t0, TOP  # b != 10?
```

(add can throw an exception)
While Loops

Pre-test loop: body may never execute

```plaintext
a = 0;
move $a0, $0  # a = 0
b = 0;
move $a1, $0  # b = 0
while (b != 10) {
    a = a + b;
    addu $a0, $a0, $a1  # a = a + b
    b = b + 1;
    addiu $a1, $a1, 1  # b = b + 1
}
```

The loop body is executed if `b` is less than 10. The loop ends when `b` becomes 10, at which point `a` is the result of the loop's computation.
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;
}
```

```
move $a1, $0 # b = 0
move $a0, $a1 # a = b
li $t0, 10
b    TEST    # test first
BODY:
addu $a0, $a0, $a1 # a = a + b
addiu $a1, $a1, 1 # b = b + 1
TEST:
bne $a1, $t0, BODY # b != 10?
```
Arrays

```c
int a[5];

void main() {
}
```

```
.text
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  # constant
la $t1, a  # base address of array
```

BODY:
```
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  # i = i + 1
```

TEST:
```
sltu $t2, $a1, $t0  # i < 10?
bne $t2, $0, BODY
```
Summing the contents of an array

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

```assembly
move $a0, $0  # s = 0
la $t0, a    # &a[0]
addiu $t1, $t0, 36  # i = a + 9
b TEST

BODY:
lw $t2, 0($t1)  # *i
addu $a0, $a0, $t2  # s += *i
addiu $t1, $t1, -4  # i--

TEST:
sltu $t2, $t1, $t0  # i < a
beq $t2, $0, BODY
```

- **s = 0** is initialized to 0 using `move $a0, $0`.
- **la $t0, a** loads the address of the array `a[0]` into `$t0`.
- **addiu $t1, $t0, 36** calculates the address of `i` as `a + 9`.
- The **for loop** iterates from `s=0` to `i >= a`, updating `s` by adding the value of `*i`.
- **lw $t2, 0($t1)** loads the value pointed by `i` into `$t2`.
- **addu $a0, $a0, $t2** updates `s` by adding the value loaded into `$t2`.
- **addiu $t1, $t1, -4** decrements `i`.
- **sltu $t2, $t1, $t0** checks if `i` is less than `a`.
- **beq $t2, $0, BODY** jumps to the `BODY` section if `i < a`.

**Pointer Update**:
- `$t1` is used to store the value of `i`.
- Pointer increment occurs during the decrement operation using `addiu $t1, $t1, -4`.

**Array Access**:
- Access to array elements is done through `$t2` which points to the current element.

**Assembly Code**:
- `TEST` and `BODY` sections are used to structure the assembly code.
- The loop is designed to sum the elements of the array from `a+9` to `a`.

**Notes**:
- The assembly code is annotated with comments highlighting the key operations.
- The loop and pointer update are critical for reading and understanding the code.
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, 0xffff0000  # 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] 1140fffe  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] 0401fff9  bgez  $0, -28 [wait]
[0040004c] 03e00008  jr   $31

[10010000] 6c6c6548 6f57206f  Hello  Wo
[10010008] 21646c72 0000000a  rld .....
```
| 0: | NUL \0 | DLE | 0 | @ | P | ‘ | p |
| 1: | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2: | STX | DC2 | " | 2 | B | R | b | r |
| 3: | ETX | DC3 | # | 3 | C | S | c | s |
| 4: | EOT | DC4 | $ | 4 | D | T | d | t |
| 5: | ENQ | NAK | % | 5 | E | U | e | u |
| 6: | ACK | SYN | & | 6 | F | V | f | v |
| 7: | BEL \a | ETB | ’ | 7 | G | W | g | w |
| 8: | BS \b | CAN | ( | 8 | H | X | h | x |
| 9: | HT \t | EM | ) | 9 | I | Y | i | y |
| A: | LF \n | SUB | * | : | J | Z | j | z |
| B: | VT \v | ESC | + | ; | K | [ | k | { |
| C: | FF \f | FS | , | < | L | \ | l | |
| D: | CR \r | GS | – | = | M | ] | m | } |
| E: | SO | RS | . | > | N | ^ | n | ~ |
| F: | SI | US | / | ? | O | _ | o | DEL |
Subroutines

a.k.a. procedures, functions, methods, et al.

Code that can run 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 restored to value on entry
# $a0,...,$a3, $t0,...,$t9 possibly clobbered
jr $ra
# Return to the caller

mysub: # Entry point: $ra holds return address
# First four args in $a0, $a1, ..., $a3
# Entry point: $ra holds return address
body of the subroutine ...
# $v0, and possibly $v1, hold the result
# $gp, $sp, $fp, and $ra also restored
...
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>0xCAFE0B0E</td>
</tr>
<tr>
<td>0x7FFFFFF4</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>0x7FFFFFF0</td>
<td>0xCODEFACE</td>
</tr>
<tr>
<td>0x7FFFFFFE</td>
<td></td>
</tr>
</tbody>
</table>

- $sp$ grows down
- $0(sp)$
- $s(sp)$
- $v(sp)$

for storing registers

Top of the stack in memory

Lowest address in memory

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);
    }
}
Allocating 24 stack bytes:
multiple of 8 for alignment

Check whether \( n = 0 \)

Save \( \$ra, \$s0, \ldots, \$s3 \) on the stack
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 src
move $s1, $a1 tmp
move $s2, $a2 dst
addiu $s3, $a3, -1 n - 1
```

Save src in $s0
Save tmp in $s1
Save dst in $s2
Save n – 1 in $s3
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(src, dst, tmp, n−1)

$n-1$ saved all the $s$ registers

$\dddot{a}0$ is src
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  # "dst"
syscall
li $v0, 4  # print_str
la $a0, newline
syscall

Print src -> dst
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, $s2
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)
mov $s0, $a0
move $s1, $a1
move $s2, $a2
addiu $s3, $a3, -1
move $a1, $s2
move $a2, $s1
move $a3, $s3
jal hmove

Defensive
might not
be necessary.

Recursive call

Call
hmove(tmp, src, dst, n−1)
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)
Restore variables
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, ELSE
    li $v0, 1           # Yes, return 1
    addiu $sp, $sp, 8   # Pop 2 words from stack
    jr $ra               # return
ELSE:
    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
    lw $v0, 0($sp)      # Compute n * fact(n-1)
    mul $v0, $a0, $v0   # Compute n * fact(n-1)
    addiu $sp, $sp, 8   # Pop 2 words from stack
    jr $ra               # return
```
Memory Layout

- Stack
- Heap
- Static Data
- Program Text
- Reserved
Differences in Other ISAs

More or fewer general-purpose registers (Itanium: 128; 6502: 3)

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

Registers that are more specialized (x86)

More addressing modes (x86: 6; VAX: 20)

Arithmetic instructions that also access memory (x86; VAX)

Arithmetic instructions on other data types (bytes and halfwords)

Variable-length instructions (x86; ARM)

Predicated instructions (ARM, VLIW)

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