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

```assembly
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;
}
```

Machine, Assembly, and C Code

```
0001000010000101000000000000000111  beq  $4, $5, 28
0000000010100100000010000000101010  slt  $2, $5, $4
0001010001000000000000000000000011  bne  $2, $0, 12
000000001010010000010100000100011  subu $5, $5, $4
000001000000000001111111111111100  bgez $0  -16
000000001000010100010000000100011  subu $4, $4, $5
000001000000000001111111111111100  bgez $0  -24
0000000000000100000010000000100001  addu $2, $0, $4
00000011111000000000000000000001000 jr  $31
```

```c
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

```
0001000010000101000000000000000111  beq  $4, $5, 28
00000000101001000001000001010100   slt  $2, $5, $4
0001010001000000000000000000000011  bne  $2, $0, 12
0000000010100100000000000000010011  subu $5, $5, $4
000001000000000001111111111111100  bgez $0 -16
00000000100001000000000001000111   subu $4, $4, $5
000001000000000001111111111111010  bgez $0 -24
0000000000000100000000010000010001  addu $2, $0, $4
0000000111111000000000000000000100  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

```assembly
00010000100001010000000000000111
000000001010000010000000101010
00101000100000000000000000000111
0000000010100000101000000100011
000001000000000000111111111111100
00000000100001010010000000100011
00000100000000011111111111111010
00000000000001000001000000100001
000001111100000000000000010000
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:

```assembly
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
```

```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 $a$
5. Go to step 2
6. Subtract $a$ from $b$
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     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>Arithmetic</th>
<th>Shift Instructions</th>
<th>Multiply/Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td><strong>sll</strong> Shift left logical</td>
<td><strong>mult</strong> Multiply</td>
</tr>
<tr>
<td>addu</td>
<td><strong>srl</strong> Shift right logical</td>
<td><strong>multu</strong> Multiply unsigned</td>
</tr>
<tr>
<td>sub</td>
<td><strong>sra</strong> Shift right arithmetic</td>
<td><strong>div</strong> Divide</td>
</tr>
<tr>
<td>subu</td>
<td><strong>sllv</strong> Shift left logical variable</td>
<td><strong>divu</strong> Divide unsigned</td>
</tr>
<tr>
<td>slt</td>
<td><strong>srlv</strong> Shift right logical variable</td>
<td></td>
</tr>
<tr>
<td>sltu</td>
<td><strong>srav</strong> Shift right arith. variable</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td><strong>mult</strong> Multiply</td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td><strong>multu</strong> Multiply unsigned</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td><strong>div</strong> Divide</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td><strong>divu</strong> Divide unsigned</td>
<td></td>
</tr>
<tr>
<td>andi</td>
<td><strong>mfhi</strong> Move from HI</td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td><strong>mthi</strong> Move to HI</td>
<td></td>
</tr>
<tr>
<td>xori</td>
<td><strong>mflo</strong> Move from LO</td>
<td></td>
</tr>
<tr>
<td>lui</td>
<td><strong>mtlo</strong> Move to LO</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:

```
operation 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></th>
<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></td>
<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>

\[
a = b - c;
\]

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

- `subu $s0, $s1, $s2`
- `addu $t0, $s4, $s5`
- `addu $t1, $s6, $s7`
- `subu $s3, $t0, $t1`

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

```assembly
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
```
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.,

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

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 \(0xC0DE\)FACE:

\[
\text{lui } \$t0, 0xC0DE \\
\text{ori } \$t0, \$t0, 0xFACE
\]

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

\[
\text{li } \$t1, \text{0xCAFE0BOE} \rightarrow \text{lui } \$t1, \text{0xCAFE} \\
\text{ori } \$t1, \$t1, \text{0x0BOE}
\]
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;
}
```

**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 $a0,$a2     # a / c
mflo $a0         # a = a/c
div $0,$a1,$a0   # b % a
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:

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

\[
\begin{array}{c}
31 & 30 & \cdots & 2 & 1 & 0 \\
\hline
\end{array}
\]

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

\[
\text{\texttt{sllv}} \ a1, \ a0, \ t0
\]
The logical form of right shift adds 0’s to the MSB.

\[
\text{sr1 } \$a0, \$a0, 1
\]

\[
\begin{array}{cccc}
31 & 30 & \ldots & 2 \\
\empty & \empty & & \empty \\
0 & \empty & \ldots & \empty \\
\end{array}
\]

\[
\begin{array}{cccc}
\empty & \empty & \empty & 0 \\
\empty & \empty & \empty & \empty \\
\end{array}
\]
Shift Right Arithmetic

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

\[ \text{sra } \$a0, \$a0, 2 \]

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & \cdots & 3 & 2 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Shifted Bits} & \text{Original Bits} \\
\end{array}
\]
Set on Less Than

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

Set $t0$ to 1 if the contents of $t1 < t2$; 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;
}
```

```asm
compare:
    move $v0, $zero
    slt  $t0, $a0, $a1
    beq $t0, $zero, .L1
    addi $v0, $v0, 42
.L1:
    sltu $t0, $a2, $a3
    beq $t0, $zero, .L2
    addi $v0, $v0, 99
.L2:
    j   $ra
```
### Load/Store Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lb</code></td>
<td>Load byte</td>
</tr>
<tr>
<td><code>lbu</code></td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td><code>lh</code></td>
<td>Load halfword</td>
</tr>
<tr>
<td><code>lhu</code></td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>Load word</td>
</tr>
<tr>
<td><code>lw</code></td>
<td>Load word left</td>
</tr>
<tr>
<td><code>lwr</code></td>
<td>Load word right</td>
</tr>
<tr>
<td><code>sb</code></td>
<td>Store byte</td>
</tr>
<tr>
<td><code>sh</code></td>
<td>Store halfword</td>
</tr>
<tr>
<td><code>sw</code></td>
<td>Store word</td>
</tr>
<tr>
<td><code>swl</code></td>
<td>Store word left</td>
</tr>
<tr>
<td><code>swr</code></td>
<td>Store word right</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:

```
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
Base Addressing in MIPS

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

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

\[
\begin{align*}
\text{\$t1: } & \quad \text{00000008 (base register)} \\
+ & \quad \text{34 (immediate offset)} \\
\text{42: } & \quad \text{EF} \\
\text{\$t0: } & \quad \text{FFFFFFFEF}
\end{align*}
\]

\(-32768 < \text{offset} < 32767\)
MIPS registers are 32 bits (4 bytes). Loading a byte into a register either clears the top three bytes or sign-extends them.

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

\[
\begin{align*}
42: & \quad \text{F0} \\
\text{lb} & \quad \text{$t0, 42($0$)} \\
\text{$t0: FFFFFFFF0$}
\end{align*}
\]
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                      # 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,

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

```mymsub:
    #  ...
    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.
Branches

Another family of branches tests a single register:

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

```
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.
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().

```
# 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>
</tr>
</thead>
<tbody>
<tr>
<td>tge tlt</td>
<td>Conditional traps</td>
</tr>
<tr>
<td>break</td>
<td>Breakpoint trap, for debugging</td>
</tr>
<tr>
<td>eret</td>
<td>Return from exception</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, ...`

<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 ...`

| op:6 | addr:26 |
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</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></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

```plaintext
op:6  rs:5  rt:5  rd:5  shamt:5  funct:6
```

```
sra $t0, $s1, 5
```

`sra` encoding from the MIPS instruction set reference:

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>SRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>000011</td>
</tr>
</tbody>
</table>

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

```plaintext
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>

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

```
ADDIU 001001 rs rt immediate
```

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

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

\begin{tabular}{|c|c|}
\hline
op:6 & addr:26 \\
\hline
\end{tabular}

jal 0x5014

jal encoding from the MIPS instruction set reference:

\begin{tabular}{|c|c|}
\hline
JAL & instr_index \\
000011 & \\
\hline
\end{tabular}

Instruction index is a word address

\begin{tabular}{|c|c|}
\hline
000011 & 00 0000 0000 0001 0100 0000 0101 \\
\hline
\end{tabular}
### Assembler Pseudoinstructions

<table>
<thead>
<tr>
<th>Branch always</th>
<th>b label</th>
<th>→ beq $0, $0, label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if equal zero</td>
<td>beqz s, label</td>
<td>→ beq s, $0, label</td>
</tr>
</tbody>
</table>
| Branch greater or equal| bge s, t, label | → slt $1, s, t  
                        |                     |  beq $1, $0, label |
| Branch greater or equal unsigned | bgeu s, t, label | → sltu $1, s, t 
                        |                     |  beq $1, $0, label |
| Branch greater than    | bgt s, t, label | → slt $1, t, s  
                        |                     |  bne $1, $0, label |
| Branch greater than unsigned | bgtu s, t, label | → sltu $1, t, s 
                        |                     |  bne $1, $0, label |
| Branch less than       | blt s, t, label | → slt $1, s, t  
                        |                     |  bne $1, $0, label |
| Branch less than unsigned | bltu s, t, label | → sltu $1, s, t 
                        |                     |  bne $1, $0, label |
### Assembler Pseudoinstructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pseudoinstruction</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load immediate, $0 \leq j \leq 65535$</td>
<td>\textit{li} \textbf{d, j}</td>
<td>\textit{ori} \textbf{d, $0, j}$</td>
</tr>
<tr>
<td>Load immediate, $-32768 \leq j &lt; 0$</td>
<td>\textit{li} \textbf{d, j}</td>
<td>\textit{addiu} \textbf{d, $0, j}$</td>
</tr>
<tr>
<td>Load immediate, negative</td>
<td>\textit{li} \textbf{d, j}</td>
<td>\textit{liu} \textbf{d, hi16(j)} \textit{ori} \textbf{d, d, lo16(j)}</td>
</tr>
<tr>
<td>Move</td>
<td>\textit{move} \textbf{d, s}</td>
<td>\textit{or} \textbf{d, s, $0$}</td>
</tr>
<tr>
<td>Multiply</td>
<td>\textit{mul} \textbf{d, s, t}</td>
<td>\textit{mult} \textbf{s, t} \textit{mflo} \textbf{d}</td>
</tr>
<tr>
<td>Negate unsigned</td>
<td>\textit{negu} \textbf{d, s}</td>
<td>\textit{subu} \textbf{d, $0, s$}</td>
</tr>
<tr>
<td>Set if equal</td>
<td>\textit{seq} \textbf{d, s, t}</td>
<td>\textit{xor} \textbf{d, s, t} \textit{sltiu} \textbf{d, d, 1}</td>
</tr>
<tr>
<td>Set if greater or equal</td>
<td>\textit{sge} \textbf{d, s, t}</td>
<td>\textit{slt} \textbf{d, s, t} \textit{xori} \textbf{d, d, 1}</td>
</tr>
<tr>
<td>Set if greater or equal unsigned</td>
<td>\textit{sgeu} \textbf{d, s, t}</td>
<td>\textit{sltu} \textbf{d, s, t} \textit{xori} \textbf{d, d, 1}</td>
</tr>
<tr>
<td>Set if greater than</td>
<td>\textit{sgt} \textbf{d, s, t}</td>
<td>\textit{slt} \textbf{d, t, s}</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.

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

\textbf{addiu} \quad \$t0, \$a0, 3 \quad \# w: \$a0
\textbf{mul} \quad \$t0, \$t0, \$a3 \quad \# x: \$a1
\textbf{addu} \quad \$t0, \$t0, \$a2 \quad \# y: \$a2
\textbf{addu} \quad \$t0, \$t0, \$a1 \quad \# z: \$a3

Consider an alternative:

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

\textbf{addu} \quad \$t0, \$a1, \$a2
\textbf{addiu} \quad \$t1, \$a0, 3 \quad \# Need a second temporary
\textbf{mul} \quad \$t1, \$t1, \$a3
\textbf{addu} \quad \$t0, \$t0, \$t1
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

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

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

Pre-test loop: body may never execute

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

```
move $a0, $0  # a = 0
move $a1, $0  # b = 0
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?
```
For Loops

“Syntactic sugar” for a while loop

```plaintext
for (a = b = 0 ; b != 10 ; b++)
    a += b;
```

is equivalent to

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

```plaintext
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() {
    a[1] = a[0] = 3;
}
```

```asm
.comm a, 20  # Allocate 20
.text    # Program next
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];

move $a1, $0  # i = 0
move $a0, $a1  # s = 0
li $t0, 10
la $t1, a  # base address of array
b TEST

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

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
```
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] 0401ffff bgez $0 -28 [wait]
[0040004c] 03e00008 jr $31

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

0x7FFFFFFFC | 0x32640128
0x7FFFFFF8 | 0xCAFE0B0E
0x7FFFFFF4 | 0xDEADBEEF
0x7FFFFFF0 | 0xCODEFACE
0x7FFFFFFEC | ...

$sp \quad \downarrow \quad \text{Grows down}
Towers of Hanoi

```c
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);
    }
}
```
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)

Allocate 24 stack bytes:
  multiple of 8 for alignment

Check whether $n == 0

Save $ra, $s0, ..., $s3 on the stack

\[
\begin{array}{cccc}
  \text{src} & \text{tmp} & \text{dst} & \text{n} \\
  \hline
  \vdots & & & \vdots \\
  16($sp) & & $s3 & \\
  12($sp) & & $s2 & \\
  8($sp) & & $s1 & \\
  4($sp) & & $s0 & \\
  0($sp) & & $ra & \\
\end{array}
\]
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:
  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)
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

Print src -> dst
void 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)
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
    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
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 $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
arrow: .asciiz "->"
newline: .asciiz "\n"
Factorial Example

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

Fact:  
```asm
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
mul $v0, $a0, $v0    # Compute n * fact(n-1)
addiu $sp, $sp, 8    # Pop 2 words from stack
jr $ra               # return
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFF FFFC</td>
<td>Stack</td>
</tr>
<tr>
<td>0x1000 8000</td>
<td>Heap</td>
</tr>
<tr>
<td>0x1000 0000</td>
<td>Static Data</td>
</tr>
<tr>
<td>0x1000 0000</td>
<td>Program Text</td>
</tr>
<tr>
<td>0x0040 0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

Symbols:
- $sp
- $gp
- pc
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)