# Fundamentals of Computer Systems

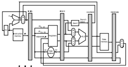
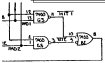
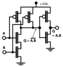
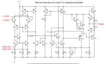
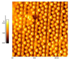
## The MIPS Instruction Set

Martha A. Kim

Columbia University

Spring 2016

# So, Where Are We?

| | | |
|---|---|---|
|  | Application Software | COMS 3157, 4156, et al. |
|  | Operating Systems | COMS W4118 |
| `;; voice 1 wave select` `ld a,(#CH1_W_NUM)` `and a` `ld a,(#CH1_W_SEL)` `jr nz,#00b4` `ld a,(#CH1_E_TABLE0)` | Architecture | Second Half of 3827 |
|  | Micro-Architecture | Second Half of 3827 |
|  | Logic | First Half of 3827 |
|  | Digital Circuits | First Half of 3827 |
|  | Analog Circuits | ELEN 3331 |
|  | Devices | ELEN 3106 |
|  | Physics | ELEN 3106 et al. |

# The Rest of the Semester: The Big Picture

Software

Hardware

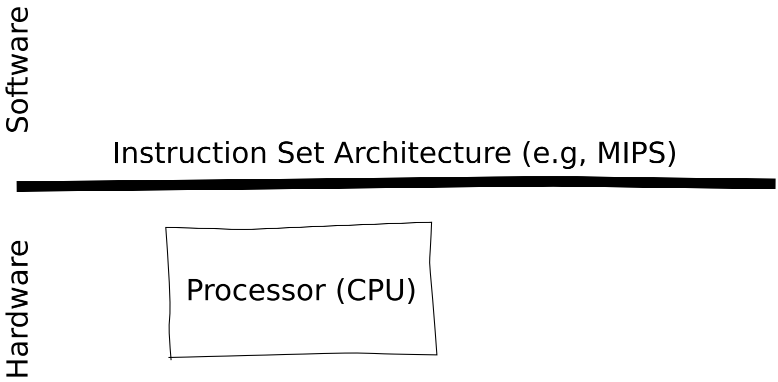
# The Rest of the Semester: The Big Picture

Software

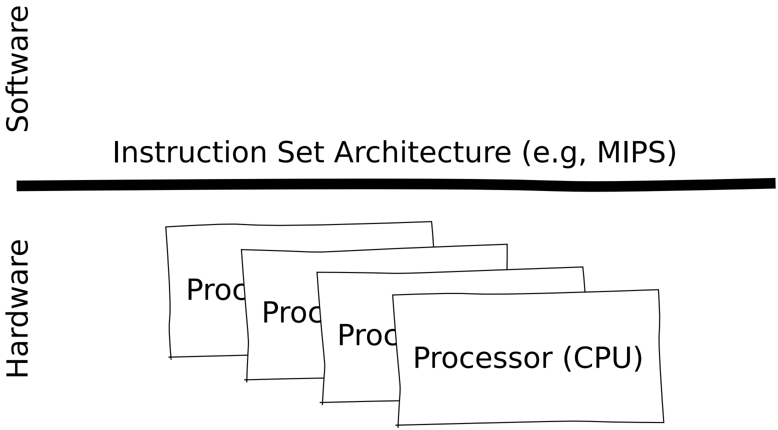
Instruction Set Architecture (e.g, MIPS)

Hardware

# The Rest of the Semester: The Big Picture



# The Rest of the Semester: The Big Picture



# C → Assembly → Machine Code

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

# C → Assembly → Machine Code

Compiler

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

```
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
  jr   $ra
```

# C → Assembly → Machine Code

Compiler

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

```
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
  jr   $ra
```

Assembler

```
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 → Assembly → Machine Code

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

Compiler

```
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
  jr   $ra
```

Assembler

```
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
```

```
00010000100001010000000000000111
00000000101001000001000000101010
00010100010000000000000000000011
00000000101001000010100000100011
00000100000000011111111111111100
00000000100001010010000000100011
00000100000000011111111111111010
00000000000001000001000000100001
00000011111000000000000000001000
```

# Algorithms

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

John von Neumann, *First Draft of a Report on the EDVAC*, 1945.

“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)

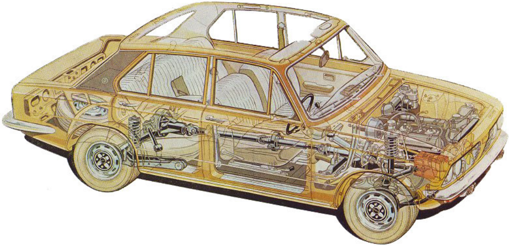
ISA: The interface or contact between the hardware and the software

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



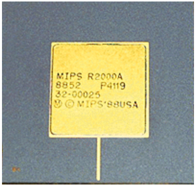
Microarchitecture:
The detailed implemention of the architecture

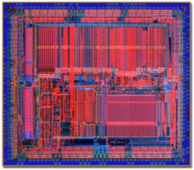
# MIPS

**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages

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.





# RISC vs. CISC Architectures

MIPS is a Reduced Instruction Set Computer (RISC). 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)
  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
  jr   $ra             # Return to caller
```

Opcodes

# 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
  jr   $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
  jr   $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
  jr   $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
  jr   $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
  jr   $ra             # Return to caller
```

Control-transfer instructions

# General-Purpose Registers

| Name | Number | Usage | Preserved? |
|---|---|---|---|
| \$zero | 0 | Constant zero | |
| \$at | 1 | Reserved (assembler) | |
| \$v0–\$v1 | 2–3 | Function result | |
| \$a0–\$a3 | 4–7 | Function arguments | |
| \$t0–\$t7 | 8–15 | Temporaries | |
| \$s0–\$s7 | 16–23 | Saved | yes |
| \$t8–\$t9 | 24–25 | Temporaries | |
| \$k0–\$k1 | 26-27 | Reserved (OS) | |
| \$gp | 28 | Global pointer | yes |
| \$sp | 29 | Stack pointer | yes |
| \$fp | 30 | Frame pointer | yes |
| \$ra | 31 | Return address | yes |

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 | |
|---|---|
| **add** | Add |
| **addu** | Add unsigned |
| **sub** | Subtract |
| **subu** | Subtract unsigned |
| **slt** | Set on less than |
| **sltu** | Set on less than unsigned |
| **and** | AND |
| **or** | OR |
| **xor** | Exclusive OR |
| **nor** | NOR |

| Arithmetic (immediate) | |
|---|---|
| **addi** | Add immediate |
| **addiu** | Add immediate unsigned |
| **slti** | Set on l. t. immediate |
| **sltiu** | Set on less than unsigned |
| **andi** | AND immediate |
| **ori** | OR immediate |
| **xori** | Exclusive OR immediate |
| **lui** | Load upper immediate |

| Shift Instructions | |
|---|---|
| **sll** | Shift left logical |
| **srl** | Shift right logical |
| **sra** | Shift right arithmetic |
| **sllv** | Shift left logical variable |
| **srlv** | Shift right logical variable |
| **srav** | Shift right arith. variable |

| Multiply/Divide | |
|---|---|
| **mult** | Multiply |
| **multu** | Multiply unsigned |
| **div** | Divide |
| **divu** | Divide unsigned |
| **mfhi** | Move from HI |
| **mthi** | Move to HI |
| **mflo** | Move from LO |
| **mtlo** | Move to LO |

# 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:

$$\textit{operation}\ R_D, R_S, R_T$$

"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

| a | b | c | f | g | h | i | j |
|---|---|---|---|---|---|---|---|
| \$s0 | \$s1 | \$s2 | \$s3 | \$s4 | \$s5 | \$s6 | \$s7 |

```
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 otherwise identical.

# Bitwise Logical Operator Example

```
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,

$$\textit{operation}\ R_D, R_S, I$$

"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.,

```
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:

```
lui $t0, 0xC0DE
ori $t0, $t0, 0xFACE
```

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

```
li $t1, 0xCAFE0B0E →  lui $t1, 0xCAFE
                      ori $t1, $t1, 0x0B0E
```

# Multiplication and Division

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

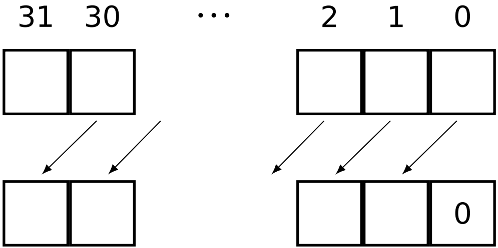
```
int multdiv(
  int a,      // $a0
  int b,      // $a1
  unsigned c, // $a2
  unsigned d) // $a3
{
  a = a * b + c;
  c = c * d + a;

  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:

**sll** $a0, $a0, 1



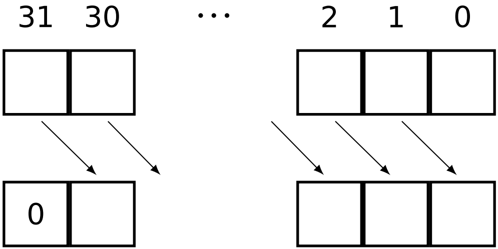

The variable form takes the number of bits to shift from a register (mod 32):

**sllv** $a1, $a0, $t0

# Shift Right Logical

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

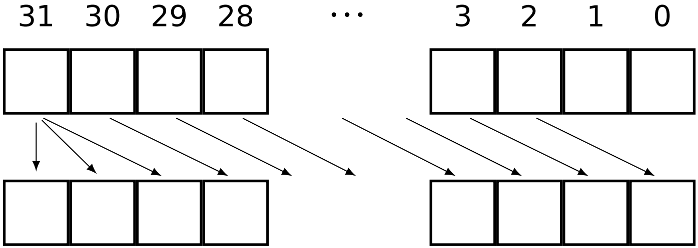
**srl** $a0, $a0, 1



# 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

**slt** $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;
}
```

```
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 and Store Instructions

| Load/Store Instructions | |
|---|---|
| **`lb`** | Load byte |
| **`lbu`** | Load byte unsigned |
| **`lh`** | Load halfword |
| **`lhu`** | Load halfword unsigned |
| **`lw`** | Load word |
| **`lwl`** | Load word left |
| **`lwr`** | Load word right |
| **`sb`** | Store byte |
| **`sh`** | Store halfword |
| **`sw`** | Store word |
| **`swl`** | Store word left |
| **`swr`** | Store word right |

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 |

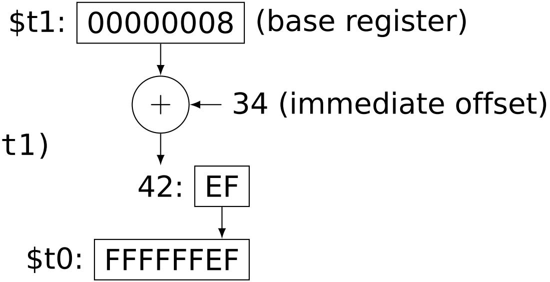
Bytes have non-negative integer addresses. Byte addresses on the 32-bit MIPS processor are 32 bits; 64-bit processors usually have 64-bit addresses.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |
| $\vdots$ | | | | | | | | |
| $2^{32}-1$: | 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.




**lb** `$t0, 34($t1)`

$-32768 < \text{offset} < 32767$

# Byte Load and Store

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: FFFFFFF0

# 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?

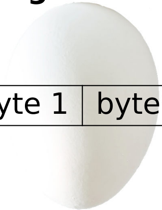
Little Endian: Intel, DEC, et al.

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

MIPS can do either

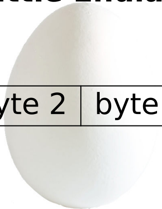
SPIM adopts its host's convention

**Big Endian**



| 31 | | | 0 |
|---|---|---|---|
| byte 0 | byte 1 | byte 2 | byte 3 |

**Little Endian**



| 31 | | | 0 |
|---|---|---|---|
| byte 3 | byte 2 | byte 1 | byte 0 |

# 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


| Jump and Branch Instructions | |
|---|---|
| **j** | Jump |
| **jal** | Jump and link |
| **jr** | Jump to register |
| **jalr** | Jump and link register |
| **beq** | Branch on equal |
| **bne** | Branch on not equal |
| **blez** | Branch on less than or equal to zero |
| **bgtz** | Branch on greater than zero |
| **bltz** | Branch on less than zero |
| **bgez** | Branch on greater than or equal to zero |
| **bltzal** | Branch on less than zero and link |
| **bgezal** | Branch on greter than or equal to zero and link |

# 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
  # ...

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.

# 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 greter 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().

```
  # 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** | |
| **tge tlt ...** | Conditional traps |
| **break** | Breakpoint trap, for debugging |
| **eret** | Return from exception |
| **Multiprocessor Instructions** | |
| **ll sc** | Load linked/store conditional for atomic operations |
| **sync** | Read/Write fence: wait for all memory loads/stores |
| **Coprocessor 0 Instructions (System Mgmt)** | |
| **lwr lwl ...** | Cache control |
| **tlbr tblwi ...** | TLB control (virtual memory) |
| **...** | Many others (data movement, branches) |
| **Floating-point Coprocessor Instructions** | |
| **add.d sub.d ...** | Arithmetic and other functions |
| **lwc1 swc1 ...** | Load/store to (32) floating-point registers |
| **bct1t ...** | Conditional branches |

# Instruction Encoding

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

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

Immediate-type: **`addi, ori, beq,`** ...

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

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

| op:6 | addr:26 |
|---|---|

# Register-type Encoding Example

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

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

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

| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | ADD<br>100000 |
|---|---|---|---|---|---|

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

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|---|---|---|---|---|---|

# Register-type Shift Instructions

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

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

**`sra`** encoding from the MIPS instruction set reference:

| SPECIAL<br>000000 | 0<br>00000 | rt | rd | sa | SRA<br>000011 |
|---|---|---|---|---|---|

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<br>001001 | rs | rt | immediate |
|---|---|---|---|

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

| 001001 | 10001 | 01000 | 1111 1111 1101 0110 |
|---|---|---|---|

# Jump-Type Encoding Example

| | |
|---|---|
| op:6 | addr:26 |

**`jal`** `0x5014`

**`jal`** encoding from the MIPS instruction set reference:

| | |
|---|---|
| JAL<br>000011 | instr_index |

Instruction index is a word address

| | |
|---|---|
| 000011 | 00 0000 0000 0001 0100 0000 0101 |

# Assembler Pseudoinstructions

| | | | |
|---|---|---|---|
| Branch always | **b** *label* | → | **beq** \$0, \$0, *label* |
| Branch if equal zero | **beqz** *s*, *label* | → | **beq** *s*, \$0, *label* |
| Branch greater or equal | **bge** *s*, *t*, *label* | → | **slt** \$1, *s*, *t*<br>**beq** \$1, \$0, *label* |
| Branch greater or equal unsigned | **bgeu** *s*, *t*, *label* | → | **sltu** \$1, *s*, *t*<br>**beq** \$1, \$0, *label* |
| Branch greater than | **bgt** *s*, *t*, *label* | → | **slt** \$1, *t*, *s*<br>**bne** \$1, \$0, *label* |
| Branch greater than unsigned | **bgtu** *s*, *t*, *label* | → | **sltu** \$1, *t*, *s*<br>**bne** \$1, \$0, *label* |
| Branch less than | **blt** *s*, *t*, *label* | → | **slt** \$1, *s*, *t*<br>**bne** \$1, \$0, *label* |
| Branch less than unsigned | **bltu** *s*, *t*, *label* | → | **sltu** \$1, *s*, *t*<br>**bne** \$1, \$0, *label* |

# Assembler Pseudoinstructions

| | | |
|---|---|---|
| Load immediate<br>$0 \le j \le 65535$ | **`li`** $d$, $j$ | $\rightarrow$ **`ori`** $d$, \$0, $j$ |
| Load immediate<br>$-32768 \le j < 0$ | **`li`** $d$, $j$ | $\rightarrow$ **`addiu`** $d$, \$0, $j$ |
| Load immediate | **`li`** $d$, $j$ | $\rightarrow$ **`liu`** $d$, hi16($j$)<br>**`ori`** $d$, $d$, lo16($j$) |
| Move | **`move`** $d$, $s$ | $\rightarrow$ **`or`** $d$, $s$, \$0 |
| Multiply | **`mul`** $d$, $s$, $t$ | $\rightarrow$ **`mult`** $s$, $t$<br>**`mflo`** $d$ |
| Negate unsigned | **`negu`** $d$, $s$ | $\rightarrow$ **`subu`** $d$, \$0, $s$ |
| Set if equal | **`seq`** $d$, $s$, $t$ | $\rightarrow$ **`xor`** $d$, $s$, $t$<br>**`sltiu`** $d$, $d$, 1 |
| Set if greater or equal | **`sge`** $d$, $s$, $t$ | $\rightarrow$ **`slt`** $d$, $s$, $t$<br>**`xori`** $d$, $d$, 1 |
| Set if greater or equal unsigned | **`sgeu`** $d$, $s$, $t$ | $\rightarrow$ **`sltu`** $d$, $s$, $t$<br>**`xori`** $d$, $d$, 1 |
| Set if greater than | **`sgt`** $d$, $s$, $t$ | $\rightarrow$ **`slt`** $d$, $t$, $s$ |

# Expressions

Initial expression:

```
x + y + z * (w + 3)
```

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

```
(((w + 3) * 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) * z)
```

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

# Conditionals

```
                          addu  $t0, $a0, $a1 # x + y
                          slti  $t0, $t0, 3   # (x+y)<3
if ((x + y) < 3)          beq   $t0, $0, ELSE
  x = x + 5;              addiu $a0, $a0, 5   # x += 5
else                      b     DONE
  y = y + 4;            ELSE:
                          addiu $a1, $a1, 4   # y += 4
                        DONE:
```

# Do-While Loops

Post-test loop: body always executes once

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

```
  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?
```

# While Loops

Pre-test loop: body may never execute

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

```asm
  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

```
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() {
  a[4] = a[3] = a[2] =
    a[1] = a[0] = 3;
  a[1] = a[2] * 4;
  a[3] = a[4] * 2;
}
```

| | |
|---|---|
| | ⋮ |
| 0x10010010: | a[4] |
| 0x1001000C: | a[3] |
| 0x10010008: | a[2] |
| 0x10010004: | a[1] |
| 0x10010000: | a[0] |
| | ⋮ |

```
  .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

```
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

```
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    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] 0401fff9  bgez  $0 -28 [wait]
[0040004c] 03e00008  jr    $31

[10010000] 6c6c6548 6f57206f H e l l o   W o
[10010008] 21646c72 0000000a r l d ! . . . .
```

# ASCII

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 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 by itself, then *resume whatever invoked it*.

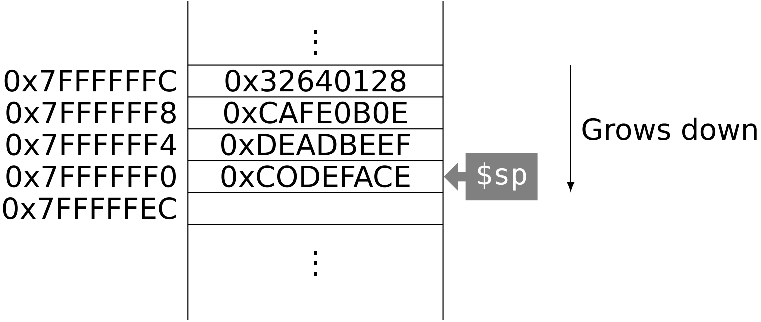
Exist for three reasons:

- 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

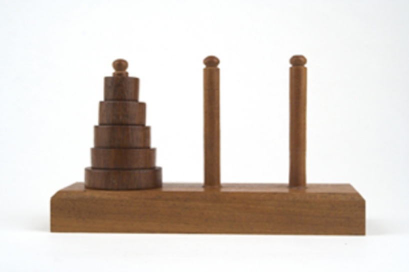
# Calling Conventions

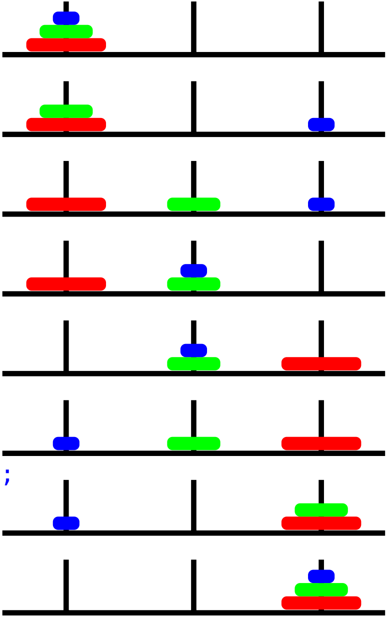
```
  # Call mysub: args in $a0,...,$a3
  jal mysub
  # Control returns here
  # Return value in $v0 & $v1
  # $s0,...,$s7, $gp, $sp, $fp, $ra unchanged
  # $a0,...,$a3, $t0,...,$t9 possibly clobbered

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


0x7FFFFFFC
0x32640128
0x7FFFFFF8
0xCAFE0B0E
0x7FFFFFF4
0xDEADBEEF
0x7FFFFFF0
0xCODEFACE
0x7FFFFFEC
$sp
Grows down

# Towers of Hanoi





```
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);
  }
}
```

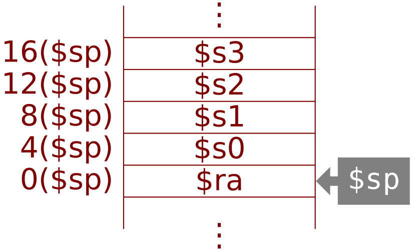
```
hmove:
  addiu $sp, $sp, -24
  beq   $a3, $0, L1
  sw    $ra, 0($sp)
  sw    $s0, 4($sp)
  sw    $s1, 8($sp)
  sw    $s2, 12($sp)
  sw    $s3, 16($sp)
```

| \$a0 | \$a1 | \$a2 | \$a3 |
|---|---|---|---|
| src | tmp | dst | n |

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:
  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, $s0
  syscall
  li    $v0, 4 # print_str
  la    $a0, arrow
  syscall
```

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

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

  li    $v0, 1 # print_int
  move  $a0, $s0
  syscall
  li    $v0, 4 # print_str
  la    $a0, arrow
  syscall
```

```
  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
```

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, $s0
  syscall
  li    $v0, 4 # print_str
  la    $a0, arrow
  syscall
```

```
  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

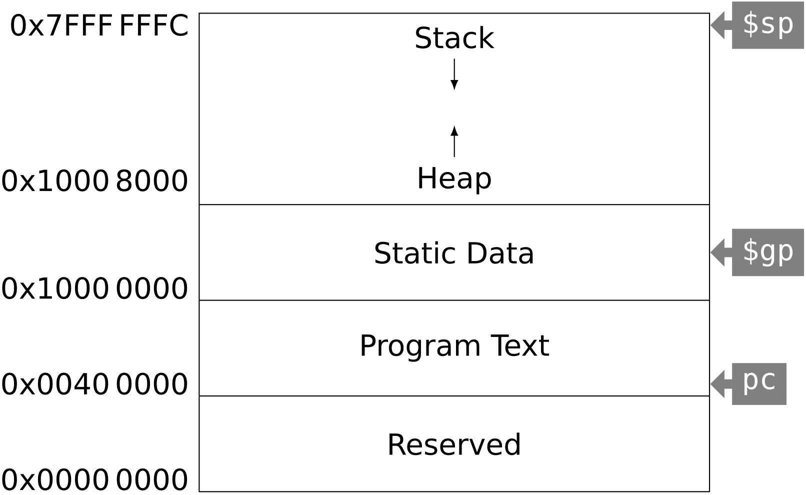
```
hmove:
  addiu $sp, $sp, -24
  beq   $a3, $0, L1
  sw    $ra, 0($sp)
  sw    $s0, 4($sp)
  sw    $s1, 8($sp)
  sw    $s2, 12($sp)
  sw    $s3, 16($sp)

  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, $s0
  syscall
  li    $v0, 4 # print_str
  la    $a0, arrow
  syscall

  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

```
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
    mul   $v0, $a0, $v0 # Compute n * fact(n-1)
    addiu $sp, $sp, 8   # Pop 2 words from stack
    jr    $ra           # return
```

# Memory Layout



# 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)