# MIPS Assembly

The MIPS Assembly Language

## MIPS Overview

### Iterations of MIPS

#### MIPS before MIPS 36/64

##### MIPS I @ ~ 1985 R2000 ⇔ (80386 Gen 3)

Basic stuff, integer / floating point registers

##### MIPS II @ ~ 1989 R6000 ⇔ Intel 80486 (Gen 4)

MIPS II had introduced some additional instructions:

- Synchronize Shared Memory,
- Load Linked Word,
- Store Conditional Word,
- Trap-on-Condition
- Branch-Likely Instructions etc.

, **with** some extra arithmetic instructions:

- Some floating-point square root instruction
- Some instructions that convert single/doubles to 32-bit integers

| Name | Mnemonic |
|---|---|
| Synchronize Shared Memory | SYNC |
| Trap if Greater Than or Equal | TGE |
| Branch on Less Than or Equal to Zero Likely | BLEZL |
| Trap if Greater Than or Equal Immediate | TGEI |
| Branch on Less Than Zero and Link Likely | BLTZALL |
| Branch on Greater Than or Equal to Zero and Link Likely | BGEZAL |
| Floating-Point Square Root | SQRT.S |
| Floating-Point Square Root | SQRT.D |
| Floating-Point Round to Word Fixed-Point | ROUND.S |
| Floating-Point Truncate to Word Fixed-Point | TRUNC.S |
| Floating-Point Ceiling to Word Fixed-Point | CEIL.S |
| Floating-Point Ceiling to Word Fixed-Point | FLOOR.S |
| Branch on Greater Than Zero Likely | BGTZL |
| Load Linked | LL |
| Load Doubleword to Coprocessor 1 | LDC1 |
| Store Conditional | SC |

| Store Doubleword to Coprocessor 1 | SDC1 |
|---|---|

### MIPS III @ ~ 1991 (R4000) ⇔ Gen 4 - Gen 5

MIPS III added support for 64-bit memory addressing and integer operations.

(While intel also added 64-bit databus)

General-purpose `$registers`/`hi`/`lo` and `$pc` are extended to 64bit to support introduction of doubleword.

Coprocessor 3 (CP3) support instructions are removed, their opcodes used for the new doubleword instructions.

The floating-point control registers were not extended for compatibility. The only new floating-point instructions added were those to copy doublewords between the CPU and FPU convert single- and double-precision floating-point numbers into doubleword integers and vice versa.

### MIPS IV @ 1994 R8000 ⇔ 1 year after Pentium (Gen 5)

Indexed addressing mode for FP loads /stores added

Prefetch & Several FP instructions added

### MIPS V @ 1996 ⇔ 1 year after Pentium Pro (Gen 6)

Introduced alongside Digital Media Extension added

## Cheat Sheet

### Instruction Formats

#### R-Type

| | Instruction Format | | | | | |
|---|---|---|---|---|---|---|
| Bit # | 31 - 26 | 25 -21 | 20 - 16 | 15 - 11 | 10 - 6 | 5 - 0 |
| R-type | op | rs | rt | rd | smt | func |
| add | 0 | rs | rt | rd | 0 | 100000 |
| addu | 0 | rs | rt | rd | 0 | 100001 |
| sub | 0 | rs | rt | rd | 0 | 100010 |
| subu | 0 | rs | rt | rd | 0 | 100011 |
| and | 0 | rs | rt | rd | 0 | 100100 |
| or | 0 | rs | rt | rd | 0 | 100101 |
| xor | 0 | rs | rt | rd | 0 | 100110 |
| nor | 0 | rs | rt | rd | 0 | 100111 |
| sll | 0 | 0 | rt | rd | s | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| srl | 0 | 0 | rt | rd | s | 10 |
| sra | 0 | 0 | rt | rd | s | 11 |
| sllv | 0 | rs | rt | rd | 0 | 100 |
| srlv | 0 | rs | rt | rd | 0 | 110 |
| srav | 0 | rs | rt | rd | 0 | 111 |
| jr | 0 | rs | 0 | 0 | 0 | 1000 |
| slt | 0 | rs | rt | rd | 0 | 101010 |
| sltu | 0 | rs | rt | rd | 0 | 101011 |

**I-Type**

| I-type | op | rs | rt | Immediate |
|---|---|---|---|---|
| addi | 1000 | rs | rt | immediate |
| addiu | 1001 | rs | rt | immediate |
| andi | 1100 | rs | rt | immediate |
| lui | 1111 | 0 | rt | immediate |
| ori | 1101 | rs | rt | immediate |
| xori | 1110 | rs | rt | immediate |
| beq | 100 | rs | rt | immediate |
| bne | 101 | rs | rt | immediate |
| slti | 1010 | rs | rt | immediate |
| sltiu | 1011 | rs | rt | immediate |
| lw | 100011 | rs | rt | immediate |
| sw | 101011 | rs | rt | immediate |

**J-Type**

| J-type | op | | address | |
|---|---|---|---|---|
| j | 000010 | | address | For example: `j 10000` |
| jal | 000011 | | address | For example: `jal 10000` |

## Instructions Quick Reference

### Arithmetic and Logical Instructions

| Instruction | Operation |
|---|---|
| add $d, $s, $t | $d = $s + $t |
| addu $d, $s, $t | $d = $s + $t |
| addi $t, $s, i | $t = $s + SE(i) |
| addiu $t, $s, i | $t = $s + SE(i) |
| and $d, $s, $t | $d = $s & $t |
| andi $t, $s, i | $t = $s & ZE(i) |
| div $s, $t | lo = $s / $t; hi = $s % $t |
| divu $s, $t | lo = $s / $t; hi = $s % $t |
| mult $s, $t | hi:lo = $s * $t |
| multu $s, $t | hi:lo = $s * $t |

| | |
|---|---|
| nor $d, $s, $t | $d = ~($s \| $t) |
| or $d, $s, $t | $d = $s \| $t |
| ori $t, $s, i | $t = $s \| ZE(i) |
| sll $d, $t, a | $d = $t << a |
| sllv $d, $t, $s | $d = $t << $s |
| sra $d, $t, a | $d = $t >> a |
| srav $d, $t, $s | $d = $t >> $s |
| srl $d, $t, a | $d = $t >>> a |
| srlv $d, $t, $s | $d = $t >>> $s |
| sub $d, $s, $t | $d = $s - $t |
| subu $d, $s, $t | $d = $s - $t |
| xor $d, $s, $t | $d = $s ^ $t |
| xori $d, $s, i | $d = $s ^ ZE(i) |

### Constant-Manipulating Instturcions

| Instruction | Operation |
|---|---|
| lhi $t, i | HH($t) = i |
| llo $t, i | LH($t) = i |

### Comparison Instructions

| Instruction | Operation |
|---|---|
| slt $d, $s, $t | $d = ($s < $t) |
| sltu $d, $s, $t | $d = ($s < $t) |
| slti $t, $s, i | $t = ($s < SE(i)) |
| sltiu $t, $s, i | $t = ($s < SE(i)) |

### Branch Instructions

| Instruction | Operation |
|---|---|
| beq $s, $t, label | if ($s == $t) pc += i << 2 |
| bgtz $s, label | if ($s > 0) pc += i << 2 |
| blez $s, label | if ($s <= 0) pc += i << 2 |
| bne $s, $t, label | if ($s != $t) pc += i << 2 |

### Jump Instructions

| Instruction | Operation |
|---|---|
| j label | pc += i << 2 |
| jal label | $31 = pc; pc += i << 2 |
| jalr $s | $31 = pc; pc = $s |
| jr $s | pc = $s |

### Load Instructions

| Instruction | Operation |
|---|---|
| lb $t, i($s) | $t = SE (MEM [$s + i]:1) |
| lbu $t, i($s) | $t = ZE (MEM [$s + i]:1) |

| | |
|---|---|
| `lh $t, i($s)` | `$t = SE (MEM [$s + i]:2)` |
| `lhu $t, i($s)` | `$t = ZE (MEM [$s + i]:2)` |
| `lw $t, i($s)` | `$t = MEM [$s + i]:4` |

### Store Instructions

| Instruction | Operation |
|---|---|
| `sb $t, i($s)` | `MEM [$s + i]:1 = LB ($t)` |
| `sh $t, i($s)` | `MEM [$s + i]:2 = LH ($t)` |
| `sw $t, i($s)` | `MEM [$s + i]:4 = $t` |

### Data Movement Instructions

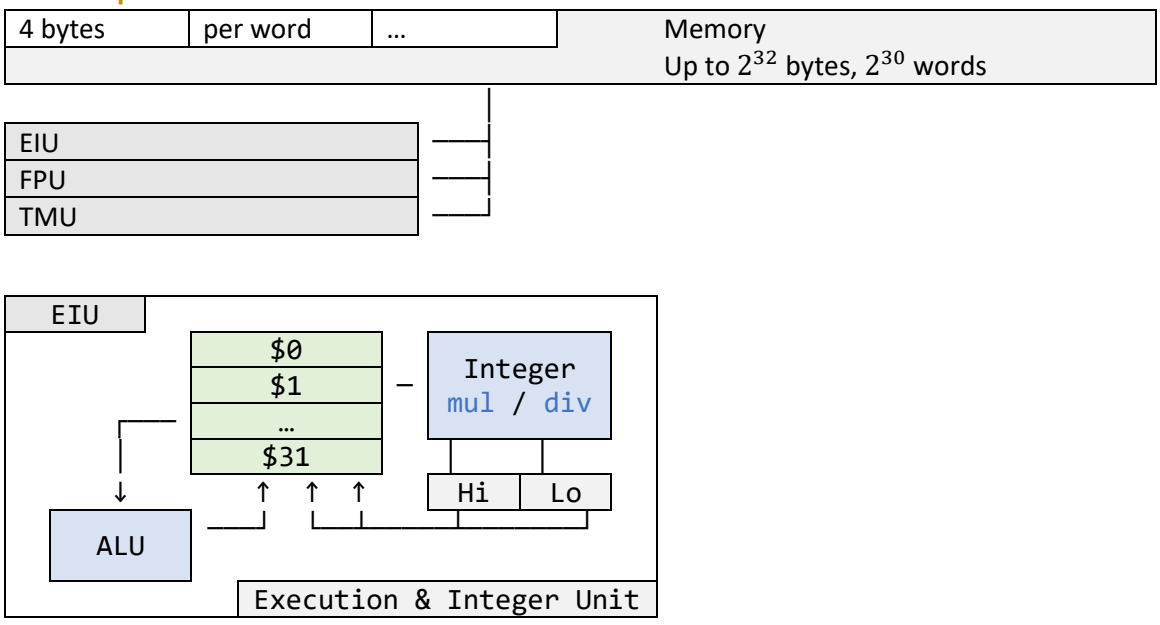
| Instruction | Operation |
|---|---|
| `mfhi $d` | `$d = hi` |
| `mflo $d` | `$d = lo` |
| `mthi $s` | `hi = $s` |
| `mtlo $s` | `lo = $s` |

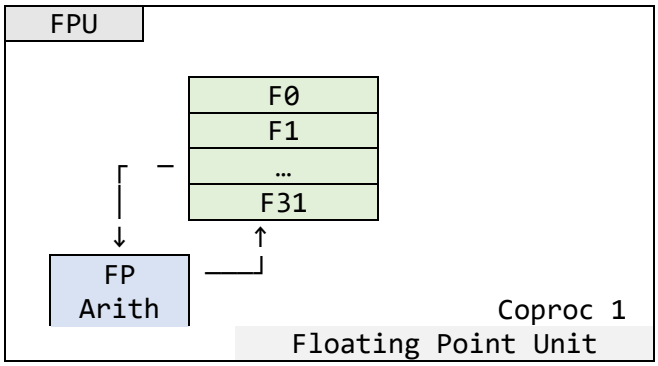
### Exception and Interrupt Instructions

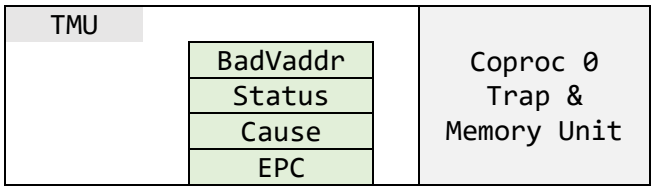
| Instruction | Operation |
|---|---|
| `trap 1` | Print integer value in $4 |
| `trap 5` | Read integer value into $2 |
| `trap 10` | Terminate program execution |
| `trap 101` | Print ASCII character in $4 |
| `trap 102` | Read ASCII character into $2 |

## MISP Architecture Visualized

### MIPS Graph






## Registers

### Registers Overview

A CPU **register** is a temporary storage built into the CPU itself (separate from the memory)

MIPS has:

- 32 of 32-bit integer registers (`$0` ~ `$31`), and
- 32 32-bit floating-point registers ($f0 ~ $f31).

### General-purpose Registers

| Sequence | Register Name | Register Description |
|---|---|---|
| 0 | **$zero($0)** | Always **0** |
| 1 | **$at** | (**assembler temporary**)<br>reserved by the assembler |
| 2-3 | **$v0 - $v1** | (return **values**)<br>from expression evaluation and function results |
| 4-7 | **$a0 - $a3**<br>(a4???) | (procedure **arguments**)<br>First four parameters for subroutine.<br>Not preserved across procedure calls |
| 8-15 | **$t0 - $t7** | (caller save, **temporaries**)<br>Caller saved if needed. Subroutines can use w/out saving.<br>Not preserved across procedure calls |
| 16-23 | **$s0 - $s7** | (**saved values**) |

| | | |
|---|---|---|
| | | Callee save.<br>A subroutine using one of these must save original and restore it before exiting.<br>Preserved across procedure calls |
| 24-25 | **$t8 - $t9** | (caller save, **temporaries**)<br>Caller saved if needed. Subroutines can use w/out saving.<br>These are in addition to $t0 - $t7 above.<br>Not preserved across procedure calls. |
| 26-27 | **$k0 - $k1** | reserved for use by the interrupt/trap handler (of OS) |
| 28 | **$gp** | **global pointer**<br>Points to the middle of the 64K block of memory in the static data segment. |
| 29 | **$sp** | **stack pointer** (callee save) |
| 30 | **$s8/$fp** | **saved value / frame pointer** (callee save)<br>Preserved across procedure calls |
| 31 | **$ra** | **return** address (callee save) |

Violet shading: Preserved between procedure calls

The global pointer is for static data within the program.

### Reserved Registers

Some general-purpose registers are in fact reserved.

The following reserved registers should not be used in user programs:

| Sequence | Register Name | Register Usage |
|---|---|---|
| $1 | $at | Assembler Temporary |
| $26-27 | $k0 - $k1 | Reserved for **OS** |
| $28 | $gp | Global Pointer |
| @TMU | $epc | Exception Program Counter |

### Miscellaneous Registers

In addition to previously listed registers, there are also some miscellaneous registers. They normally reside in a **coprocessor**:

| Register Name | Register Usage |
|---|---|
| **$status** @TMU | Status Register |
| **$cause** @TMU | Exception cause register |
| $pc | Program Counter |
| $hi | Used for multiplication/division |
| $lo | |

, **where**:

- `$pc` Points to the next instruction to be executed, not traditionally accessed by user programs
- `$status` is the process status register, updated by CPU after each instruction
- `$cause`, used in the event of an exception/interruption, essentially stores error code

## MIPS Instructions - Data Alteration

### MIPS Field Types

#### MIPS Field Types An Introduction

MIPS-32 instructions each occupy `32` bits of space.

Instructions have different formats, while their lengths remain the same.

MIPS instructions fall into those 3 categories:

| | |
|---|---|
| `R-type` | For **Arithmetic Operations and such** |
| `I-type` | For **immediate** and **data transfer** instructions |
| `J-type` | For **Jump** |

In memory, they are generally expressed such as raw bits:

| `R-type, 32 bits` | | | | | |
|---|---|---|---|---|---|
| `000000` | `10001` | `10010` | `01000` | `00000` | `100000` |
| (6 bits) | (5 bits) | (5 bits) | (5 bits) | (5 bits) | (6 bits) |

#### MIPS Fields-R

These instructions are identified by an opcode of 0, and differentiated by funct values.

**R-Type** instructions include `add` and `sub`:

| op | rs | rt | rd | `shamt` | funct |
|---|---|---|---|---|---|
| (6 bits) | (5 bits) | (5 bits) | (5 bits) | (5 bits) | (6 bits) |

, **where**:

- `op`: Basic operation of the instruction, also called the opcode
- `rs`: The first register source operand
- `rt`: The second register source operand
- `rd`: The register destination operand
- `shamt`: Shift amount
- `funct`: Function, specific variant of op

#### MIPS Fields-I

**I-type** instructions include `addi`, `lw` and `sw`:

In this case, the Constant or Address section serves as either the constant as an operand, or the int address offset.

| op | rs | rt | Constant or Address |
|---|---|---|---|
| (6 bits) | (5 bits) | (5 bits) | (16 bits) |

**for example**:

| `lw $t0,32($s3) # Temporary reg $t0 gets A[8]` | | | |
|---|---|---|---|
| op | rs | rt | Constant or Address |
| … | `$s3` | `$t0` | `bin(32)` |
| … | `19` | `8` | `bin(32)` |

**MIPS Fields-J**

| op | address |
|---|---|
| (6 bits) | (26 bits) |

## Instructions: Addition and Subtraction

**R-Arithmetic Operands, addition and subtraction**

| | |
|---|---|
| `add $d, $s, $t` | `$d = $s + $t` |
| `addu $d, $s, $t` | `$d = $s + $t` |

| | |
|---|---|
| `addi $t, $s, i` | `$t = $s + SE(i)` |
| `addiu $t, $s, i` | `$t = $s + SE(i)` |

| | |
|---|---|
| `sub $d, $s, $t` | `$d = $s - $t` |
| `subu $d, $s, $t` | `$d = $s - $t` |

More complicated performances can be executed such as:

```
# f = (g + h) – (i + j);
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 – $t1, which is (g + h)–(i + j)
```

| Overflow with add |
|---|
| Only `add` and `addi` could generate overflow |

## Instructions: Multiplication and Division: `mult, div, etc.`

### Multiplication: mult and multu

In multiplication, the register stores the result into two separate 32-bit registers: the higher part in `$hi`, and the lower part in `$lo`.

| | |
|---|---|
| `mult $s, $t` | `hi-lo = $s * $t` |
| `multu $s, $t` | `hi-lo = $s * $t` |

### Division: Accessing hi and lo

| | |
|---|---|
| `div $s, $t` | `lo = $s / $t; hi = $s % $t` |
| `divu $s, $t` | `lo = $s / $t; hi = $s % $t` |

### Accessing $hi and $lo registers

| | |
|---|---|
| `mfhi $d` | $d = hi |
| `mflo $d` | $d = lo |

| | |
|---|---|
| `lhi $t, i` | Load upper half of register with immediate `i` |
| `llo $t, i` | Load lower half of register with immediate `i` |

### Application of Data Movements

Result of a multiplication (of two 32-bit numbers) is a 64-bit number, stored in two 32-bit registers, `$hi` and `$lo`:

```
mult $t1, $t2 # hi,lo = $t1 * $t2
mflo $t0 # $t0 = lo
mfhi $t3 # $t3 = hi
```

, **or**, using the *shorthand*,

```
mul $t0, $t1, $t2 # hi,lo = $t1 * $t2; $t0 = lo
```

| `mul` ≠ `mult` |
|---|
| `mul` and `mult` are neither pseudo-instructions.<br>Actually, `mul` can only store data in the lower 32-bit information: as the tradeoff for being able to fit 3 registers in one single instruction. |

## Instructions: Bit-Wise Operations

### Logical Operations

It would be useful to operate on fields of bits within a word or even on individual bits.

| Logical operations | C operators | Java operators | MIPS Instructions |
|---|---|---|---|
| Shift left | << | << | `sll` |
| Shift right | >> | >>> | `srl` |
| Bit-by-bit AND | & | & | `and, andi` |

| | | | |
|---|---|---|---|
| Bit-by-bit OR | \| | \| | or, ori |
| Bit-by-bit NOT | ~ | ~ | nor |

There are two classes of logical operations:

- shifts: move all the bits in a word to the left or right, filling emptied bits with 0
    - Additionally, effectively

**Shift instructions**

| | |
|---|---|
| sll $d, $t, a | $d = $t << a |
| sllv $d, $t, $s | $d = $t << $s |

| | |
|---|---|
| sra $d, $t, a | $d = $t >> a |
| srav $d, $t, $s | $d = $t >> $s |

| | |
|---|---|
| srl $d, $t, a | $d = $t >>> a |
| srlv $d, $t, $s | $d = $t >>> $s |

In an **shift** instruction, the `shamt` section is no longer dormant:

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| (6 bits) | (5 bits) | (5 bits) | (5 bits) | <THIS^^> | (6 bits) |

**Bitwise AND/OR**

| | |
|---|---|
| and $d, $s, $t | $d = $s & $t |
| andi $t, $s, i | $t = $s & ZE(i) |
| or $d, $s, $t | $d = $s \| $t |
| ori $t, $s, i | $t = $s \| ZE(i) |
| nor $d, $s, $t | $d = ~($s \| $t) |
| xor $d, $s, $t | $d = $s ^ $t |
| xori $d, $s, i | $d = $s ^ ZE(i) |

The value of each single digit of the resulted register would depend on the values of the two arguments, such as:

- and: 1 if arguments are both 0

| Example of AND |
|---|
| 0000 0000 0000 0000 0000 1101 0000 0000 |
| 0000 0000 0000 0000 0011 1100 0000 0000 |
| 0000 0000 0000 0000 0000 1100 0000 0000 |

## Instructions: Logical Operations

### Branch Instructions

| | |
|---|---|
| `beq $s, $t, label` | if ($s == $t) pc += i << 2 |
| `bgtz $s, label` | if ($s > 0) pc += i << 2 |
| `blez $s, label` | if ($s <= 0) pc += i << 2 |
| `bne $s, $t, label` | if ($s != $t) pc += i << 2 |

They imply branching when:

| | |
|---|---|
| `beq $s, $t, label` | Equal |
| `bgtz $s, label` | Not equal |
| `blez $s, label` | Greater than 0 |
| `bne $s, $t, label` | Less or equal to 0 |

These instructions jump to a specific `label` if certain conditions are met.

, **such as**:

```
# With the following correlation:
```

| f | g | h | i | j |
|---|---|---|---|---|
| $s0 | $s1 | $s2 | $s3 | $s4 |

```
# The equivalent of if (i == j) f = g + h; else f = g – h;:

bne $s3,$s4,Else # go to Else if i ≠ j
add $s0,$s1,$s2 # f = g + h (skipped if i ≠ j)
j Exit # go (jump) to Exit
Else:sub $s0,$s1,$s2 # f = g – h (skipped if i = j)
Exit:
```

### Jump Instructions

| | |
|---|---|
| **`j`** `label` | pc += i << 2 |
| **`jr`** `$rs` | pc = $s |
| **`jal`** `label` | $31 = pc; pc += i << 2 |
| **`jalr`** `$rs` | $31 = pc; pc = $s |

| | |
|---|---|
| `j label` | Jump to `label` |
| `jr $rs` | Jump to `$rs` |
| `jal label` | Jump to `label`, |
| `jalr $rs` | Jump to `$rs`, store next line in $rd |

Among them, `jal` and `jalr` perform a super-set of the operations of `j`/`jr`

`jal`/`jalr` and `j`/`hr` can be implemented one in terms of the other:

| Emulate JAL/JALR with J/JR | |
|---|---|
| Original | Emulated |
| jal foo | la $ra, ret_label<br>j foo<br>ret_label: |

| Emulate J/JR with JAL/JALR | |
|---|---|
| Original | Emulated |
| j foo | prolog:<br>addi $sp, $sp, -4<br>sw $ra, ($sp)<br><br>jal foo<br><br>epilog:<br>lw $ra, ($sp)<br>addi $sp, $sp, 4 |

### Set If Instructions

| | |
|---|---|
| slt $d, $s, $t | $d = ($s < $t) |
| sltu $d, $s, $t | $d = ($s < $t) |
| slti $t, $s, i | $t = ($s < SE(i)) |
| sltiu $t, $s, i | $t = ($s < SE(i)) |

| | |
|---|---|
| seq | Set on Equak |
| slt | Set on Less Than |
| sle | Set no Less than or Equal |

For instructions such as seq $t1, $t2, $t3:

Register in place of $t1 would be set to 1 if the following conditions were met, otherwise 0.

Requirements such as:

| | |
|---|---|
| seq | $t2 == $t3 |
| slt | $t2 < $t3 |
| sle | $t2 <= $t3 |

`slt` and `slti` can be used in combination with `beq` and `bne`:

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L # branch to L
```

, because:

- Hardware for < and >=, … ar slower than = and ≠
- Combining with branch involves more work per instruction, requiring a slower clock
- All instructions penalized
- `beq` and `bne` are common case

### Set If Instructions: Signed / Unsigned Comparison

Unsigned comparisons compare registers as if they were unsigned integers:

| | |
|---|---|
| `slt, slti` | Signed comparison |
| `sltu` | Set on Less Than, unsigned |
| `sltui` | Set on Less Than Immediate, unsigned |

- For example, with

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

, **thus**:

| | |
|---|---|
| `slt $t0, $s0, $s1 # signed` | `-1 < +1 → $t0 = 1` |
| `sltu $t0, $s0, $s1 # unsigned` | `+4,294,967,295 > +1 → $t0 = 0` |

Operands can also be compared with a constant:

```
slti $t0,$s2,10 # $t0 = 1 if $s2 < 10
```

The MIPS architecture doesn't include branch on less than because it is too complicated.

## MIPS Components - Memory and Data

### Memory Overview

#### A Program - the Memory Layout:

| | |
|---|---|
| High Memory | **Stack** |
| | **Heap** |
| | Uninitialized Data |
| | Data |
| | Text (Code) |
| Lowest Memory | Reserved |

, **where**:

- `Stack` starts in high memory and grows downward
- `Heap` is where dynamically allocated data will be stored
- `Uninitialized data` contains declared variables whose values are yet to be assigned
- `Data` is where the initialized data is stored
- `Text` (code) is where the machine language is stored
- `Reserved` is not available to uer programs

### Memory Range

To store a word, four bytes are required (which uses four memory addresses)

| Name | Example | Comments |
|---|---|---|
| $2^{30}$ memory words | `Memory[0], Memory[4], …, Memory[4294967292]` | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

### Alignment Restriction, Big and Little Endian

MIPS address individual bytes, thus

| Address of a word matches that of one of the 4 bytes within the word |
|---|

(Hence, address of sequential words differ by 4)

Alignment Restriction: In MIPS, words must start at addresses that are multiples of 4.

Furthermore, addressing schemes are divided into big endian and little endian:

- **Big Endian**: address (word) == address (most significant address of the word)
  - (MIPS is also in this league)
- **Little Endian**: address (word) == address (least significant of the word)

| **Big Endian** | | | |
|---|---|---|---|
| word | | | |
| byte | byte | byte | byte |

| **Little Engian** | | | |
|---|---|---|---|
| word | | | |
| byte | byte | byte | byte |

Byte address also affect the array index. To get the proper byte address in the code above, the offset has to be 4 × 8 = 32, so that the `lw` instruction would select the param/4th entry, rather than the paramth entry:

| | | | |
|---|---|---|---|
| | | 20 | 100 |
| | | 16 | 10 |
| | | 12 | 101 |
| | | 8 | 1 |
| | | 4 | 110 |
| | | 0 | |
| | | Address | Data |
| Processor | | Memory | |

```
# A[12] = h + A[8]
# h=$s2, A=$s3
lw    $t0,32($3)        # Temporary reg $t0 gets A[8] (not 32!)
add   $t0,$s2,$t0       # Temporary reg $t0 gets h + A[8]
lw    $t0,32($3)        # Stores h + A[8] back into A[12], using 48
as offset and register $s3 as the base register
```

### Indianness affecting Stack Growth Direction ?

❝ Stack growth direction is orthogonal to integer endianness❞

❝ There is zero connection between the order of bytes within a wider integer (word), and whether a stack push adds or subtracts from the stack pointer. As far as a push is concerned, storing the data is a single operation. ❞

### Processors and Direction of Stack Growth

- Intel 8051: up
- x86: down.
- MIPS: Typically down (via load & store)
- SPARC: selectable. The standard ABI uses down.
- PPC: down, I think.
- System z: in a linked list, I kid you not (but still down, at least for zLinux).
- ARM: selectable, but Thumb2 has compact encodings only for down (LDMIA = increment after, STMDB = decrement before).
- 6502: down (but only 256 bytes).
- RCA 1802A: any way you want, subject to SCRT implementation.
- PDP11: down.

## Instructions: Memory IO

### Load Instructions

Load byte, load half, load word

| | |
|---|---|
| `lb $t, i($s)` | $t = SE (MEM [$s + i]:1) |
| `lbu $t, i($s)` | $t = ZE (MEM [$s + i]:1) |
| `lh $t, i($s)` | $t = SE (MEM [$s + i]:2) |
| `lhu $t, i($s)` | $t = ZE (MEM [$s + i]:2) |
| `lw $t, i($s)` | $t = MEM [$s + i]:4 |

### Store Instructions

| | |
|---|---|
| `sb $t, i($s)` | MEM [$s + i]:1 = LB ($t) |
| `sh $t, i($s)` | MEM [$s + i]:2 = LH ($t) |
| `sw $t, i($s)` | MEM [$s + i]:4 = $t |

### Applying Loading & Storing Instructions

Taking `lw` and `sw` as an example:

| | |
|---|---|
| `lw $t, i($s)` | Load word, load stuff from memory into register |
| `sw $t, i($s)` | Save word, put stuff from register into memory |

The size of **register** (<< *where arithmetic operations occur*) is tiny compared to memory. Instructions that transfer data between memory and register are data transfer instructions.

To access a word in memory, the instruction must supply the memory address

Compiling an Assignment When an Operand is in Memory

```
# g = h + A[8]

# g at $s1, h at $s2, A at $s3
# The address of this array element is the sum of the base of the
array A, found in register $s3, plus the number to get element 8

lw $t0,8($s3) # Temporary reg $t0 gets A[8]

add $s1,$s2,$t0 # g = h + A[8]
# The constant in a data transfer instruction is called the offset,
and the register added to form the address is called the base
register.
```

### Load hi / Load lo

| | |
|---|---|
| `lhi $t, i` | Load high immediate, load upper half of register with immediate `i` |

| | |
|---|---|
| `llo $t, i` | Load low immediate, load lower half of register with immediate `i` |

## Instructions: Memory Move

### Accessing hi and lo

| | |
|---|---|
| `move $d $s` | $d = $s |

| | |
|---|---|
| `mfhi $d` | $d = hi |
| `mflo $d` | $d = lo |

| | |
|---|---|
| `mthi $s` | hi = $s |
| `mtlo $s` | lo = $s |

## Implementation: Stack Access

### Spilling

**Spilling** is the process of putting less commonly used variables into memory.

Although:

| |
|---|
| Data is more useful in the register |
| Register is smaller, thus faster. |

, spilling is required to put aside data not used right now, in order to save space for registers.

### Stack In

Sometimes, we might need to store their old values in a stack:

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw $t1, 8($sp) # save register $t1 for use afterwards
sw $t0, 4($sp) # save register $t0 for use afterwards
sw $s0, 0($sp) # save register $s0 for use afterwards
```

The reason why the stack point gets deduced from is because, in this memory layout, stack is supposed to be growing downwards:

| |
|---|
| **$sp** |
| ↓ |
| **$sp** (new) |
| |
| **Heap** |
| … |

Also keep in mind that stack operations should be made before data is actually loaded.

### Example, what to Do after Stack In

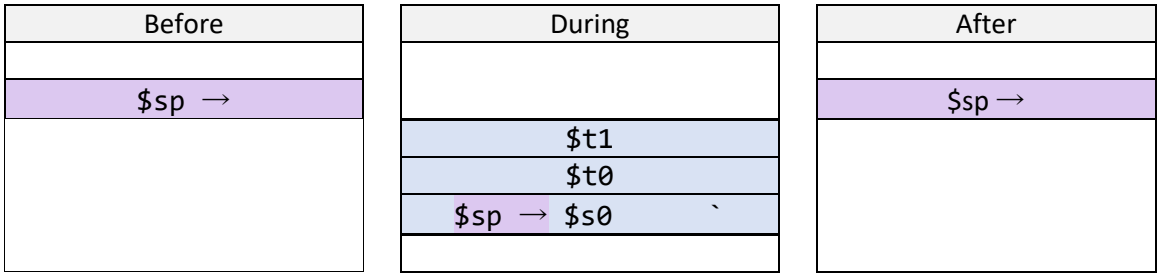
```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 – $t1, which is (g + h)–(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

### Stack Out

```
lw $s0, 0($sp) # restore register $s0 for caller
lw $t0, 4($sp) # restore register $t0 for caller
lw $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

### Stack Access Visualized

In these steps, stack can be visualized as:

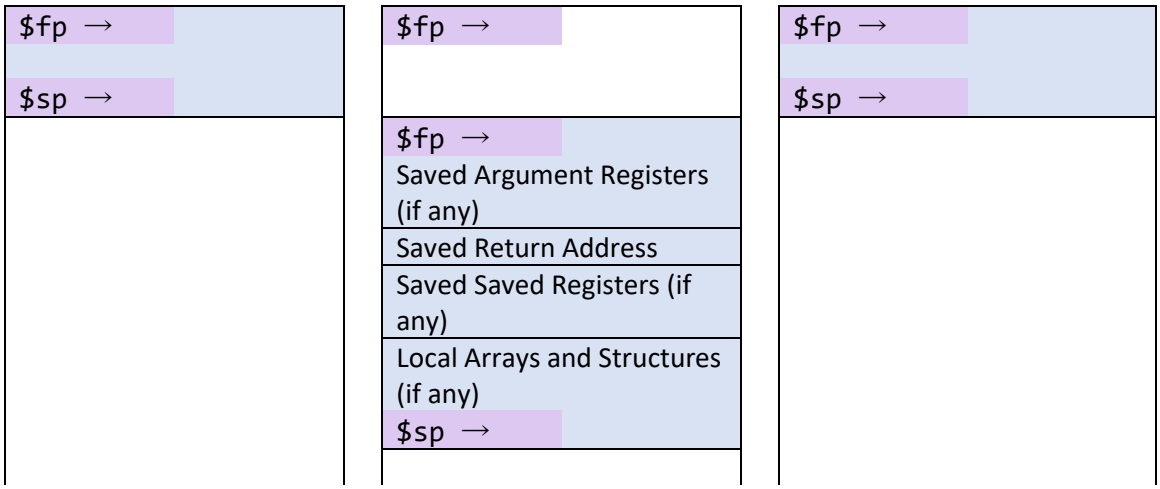



, **where**:

- **This** shade denotes `$sp`, the stack pointer
- **This** denotes stored data

### Local Data on the Stack

| | | |
|---|---|---|
| `$fp` | Frame Pointer | Points to beginning of the stack frame |
| `$sp` | Stack Pointer | Points to end of the stack frame |



| | | |
|---|---|---|
| | | |

| Relation between `$fp` and `$sp` |
|---|
| When a frame pointer is used, it is initialized using the address in $sp on a call, and $sp is restored using $fp. |

`$fp` is needed to access arguments and local variables in case it is not statically known how much the stack will grow (variable length parameters and dynamic local arrays)

The stack is also used to store variables that are local to the procedure that do not fit in the registers (local arrays and structures). Procedure frame / activation record: The segment of the stack containing a procedure's saved registers and local variables

Some MIPS software use a frame pointer (`$fp`) to point to the first word of the frame of a procedure: Stack pointer might change during the procedure, so references to a local variable in memory might have different offsets depending on where they are in the procedure, making them hard to understand.

Alternatively, a `$fp` offers a stable base register within a procedure for local memory references.

An activation record appears on the stack whether or not an explicit frame pointer is used.

## Procedure Call

### Procedure Call Components - Registers

#### Registers necessary for Procedure Calls

MIPS allocates those registers for procedure calling:

| | |
|---|---|
| `$a0 - $a3` | Four argument registers in which to pass parameters |
| `$v0 - $v1` | Two value registers in which to return values |
| `$ra` | Return address register |

#### Temporary Registers

To avoid saving & restoring a register whose value is never used, MIPS separates 18 of the registers into two groups:

`t` registers exist as a convention so that some registers would not need to be stored (know that there is no way some program knows whether or not to save a register otherwise: it cannot determine if a register contains valid data.)

| | |
|---|---|
| `$t0 - $t9` | 10 temporary registers that are not preserved by the callee |

| | |
|---|---|
| `$s0 - $s7` | 8 registers that must be preserved on a procedure call<br>(If used, the callee must save and restore them) |

## Procedure Call Components - Instructions

### Concepts

In the execution of a procedure, such as

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

, **the program** must follow these six steps:

1. Place parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin.

### jal: j for Procedure call, Jump and Return

| | |
|---|---|
| `jal Label` | |

| Procedure Call: **Jump and Link** |
|---|
| `jal <Label>`<br>`Address of following instruction puts in $ra`<br>`Jumps to Label: sets program counter to <Label>` |

Here, "link" means that an address/link is formed to the calling site (stored in $ra) that llows the procedure to return to the proper address

`jal` saves PC+4 in register $ra to link to the following instruction to set up the procedure return.

### jr: Procedure Return

| | |
|---|---|
| `jr $ra` | |

| Procedure Return: **Jump register** |
|---|
| `jr $ra`<br>`Copies $ra to program counter $pc`<br>`Can also be used for computer jumps, for case/switch statements` |

### What Happens in a Procedure call:

- Caller puts parameters in `$a0 ~ $a3`
- ↓ `jal <Label of the Callee>` to the callee
- ↓ Callee performs the calculations,
- ↓ Callee stores results in `$v0 ~ $v1`
- ✓ `jr $ra` back to the caller

## Workarounds for Procedure Call

### The Extra Parameter Problem

What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer.

The procedure then expects the first four parameters to be in registers $a0 through $a3 and the rest in memory, addressable via the frame pointer.

## Procedure Examples

### Leaf Procedure Example

```
int leaf (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
# Arguments g, …, j in $a0, …, $a3
# f in $s0 (hence, need to save $s0 on stack)
# Result in $v0
```

| | |
|---|---|
| `leaf:` | |
| `addi $sp, $sp, -4` | Save $s0 on stack |
| `sw $s0, 0($sp)` | |
| `add $t0, $a0, $a1` | Procedure body |
| `add $t1, $a2, $a3` | |
| `sub $s0, $t0, $t1` | |
| `add $v0, $s0, $zero` | Result |
| `lw $s0, 0($sp)` | Restore $s0 |
| `addi $sp, $sp, 4` | |
| `jr $ra` | Return |

### Non-Leaf Procedures

Procedures that call other procedures: caller needs to save on the stack:

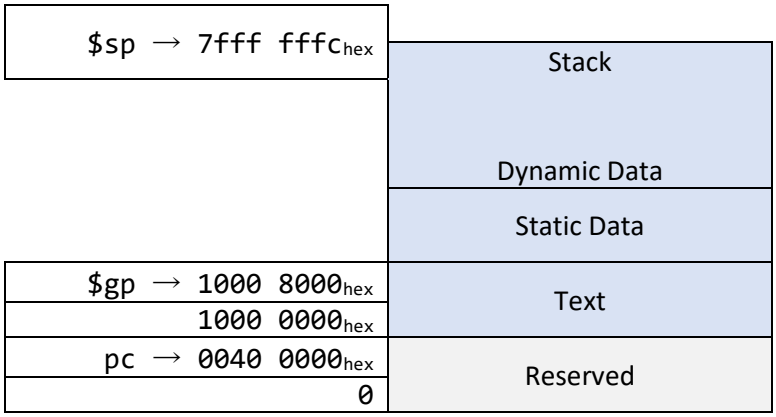
- Its return address
- Any arguments and temporaries needed after the call

**Such as** this:

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

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

**Memory Layout**




, **where**:

- Text: Program code
- Static data: glocal variables e.g. static variables in C
- Dynamic data: heap, e.g. malloc in C, new in Java, object creation in Python…
- Stack: automatic storage

## Size Workarounds

### The 32-bit Constant Problem

#### The Problem Ahead

| As a matter of fact… |
| --- |
| One immediate operand can be no more than 16 bytes (a halfword) long<br><br>Thus, to load an 32-bit immediate value (constant) into a register,<br>it needs to be loaded 16 bits at a time |

| `lui $s Imm` | `Load Upper Immedaite` |
| --- | --- |

`lui` (load upper immediate) specifically sets the ***upper 16*** bits of a constant in a register, allowing a subsequent instruction to specify the ***lower 16*** bits in the constant.

After command `lui $t0, 255:`
Register `$t0` becomes

| 0000 0000 0000 0000 | 0000 0000 0000 0000 |
| --- | --- |
| ↓ | |
| 0000 0000 1111 1111 | 0000 0000 0000 0000 |

#### Example: Loading a 32-bit constant

For example, when loading a 32-bit constant into register `$s0`:

0000 0000 0011 1101 0000 1001 0000 0000

| `lui rt, constant` | | |
| --- | --- | --- |
| `lui $s0, 61` | 0000 0000 0011 1101 | 0000 0000 0000 0000 |
| `ori $s0, $s0, 2304` | 0000 0000 0011 1101 | 0000 1001 0000 0000 |

`lui $s0, 61 # 61 decimal =` 0000 0000 0011 1101 binary

| 0000 0000 0011 1101 | 0000 0000 0000 0000 |
| --- | --- |

`ori $s0, $s0, 2304 # 2304 decimal =` 0000 1001 0000 0000

| 0000 0000 0011 1101 | 0000 1001 0000 0000 |
| --- | --- |

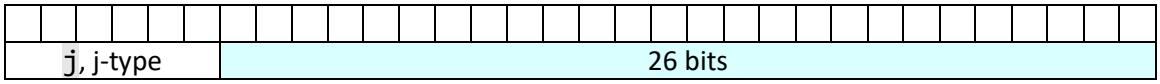
| Elaborations |
| --- |
| Creating 32-bit constants needs care. The instruction `addi` copies the |

leftmost bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. Logical or immediate from Section 2.5 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

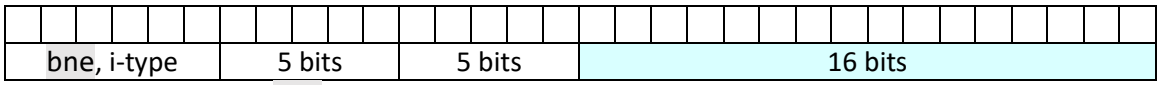
## Addressing in Branches and Jumps

### The Addressing Size Problem

Although jump instructions (with j format) has 26 bits for a space:




, **however**,




in instructions such as `bne`, there are only 16 bits for the branch address

### PC Relative Addressing (< $2^{16}$ addresses away)

`Program counter = Register + Branch address`

Since most branching actions happen less than $2^{16}$ addresses away, PC-relative addressing can be used.

Since it's convenient for the hardware to increment the PC early to point to the next instruction, MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to PC itself.

For example, in an while **loop**:

```
Loop:sll $t1,$s3,2 # Temp reg $t1 = 4 * i
    add $t1,$t1,$s6 # $t1 = address of save[i]
    lw $t0,0($t1) # Temp reg $t0 = save[i]
    bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
    addi $s3,$s3,1 # i = i + 1
    j Loop # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, the assembled instruction would look like:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Loop: sll | 80000 | 0 | 0 | 19 | 9 | 4 | 0 |
| add | 80004 | 0 | 9 | 22 | 9 | 0 | 32 |
| lw | 80008 | 35 | 9 | 8 | 0 | | |
| bne | 80012 | 5 | 8 | 21 | 2 (relative addressing) | | |
| addi | 80016 | 8 | 19 | 19 | 1 | | |
| j Loop | 80020 | 20000 | | | | | |
| | 80024 | … | | | | | |

PC-relative addressing: An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

### *Really* Far Branching (> $2^{16}$ addresses away)

Nearly every conditional branch is to a nearby location, but occasionally it branches far away, farther than can be represented in the 16 bits of the conditional branch instruction.

In this case the assembler would :

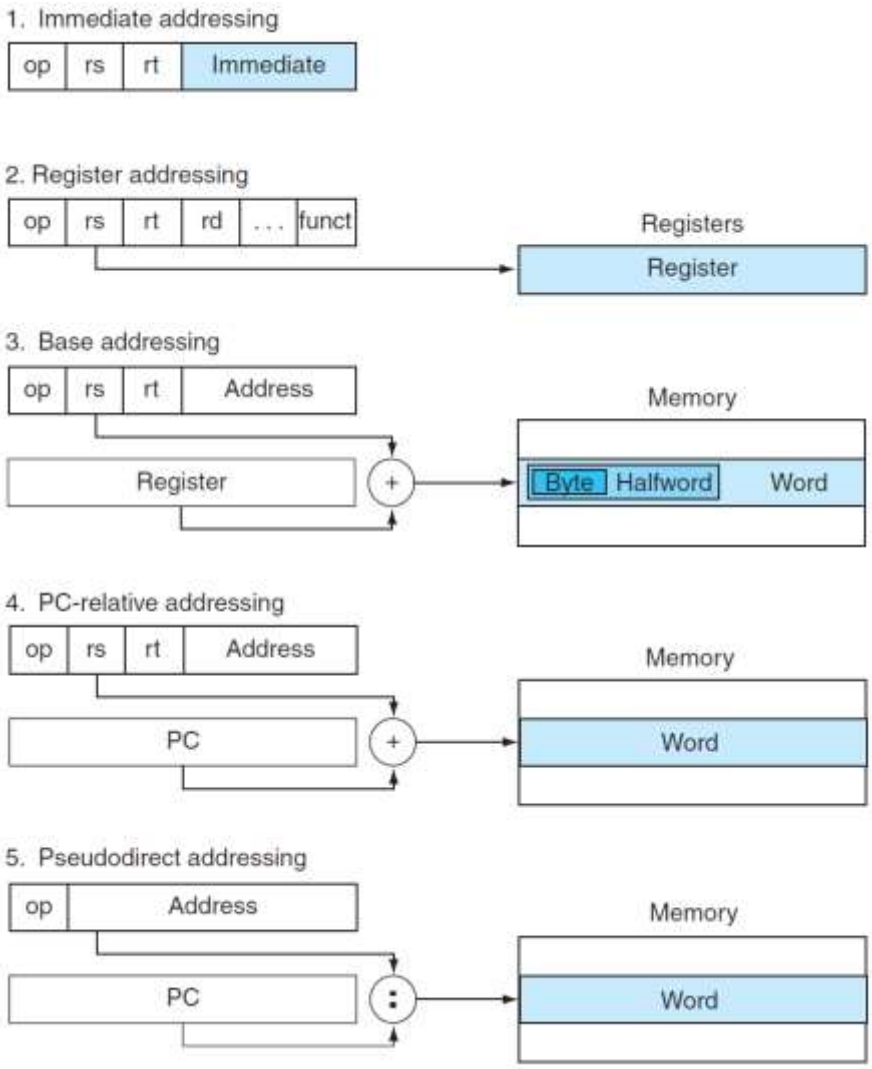
- insert an unconditional jump (`j`) to the branch target, and
- invert the condition so that the branch decides whether to skip the jump.

```
    beq     $s0, $s1, L1
    bne     $s0, $s1, L2
    j       L1
L2:
```

## MIPS Addressing Summary

### Addressing Modes

| | |
|---|---|
| 1. Register / direct | Where the operand is a register |
| 2. Base / displacement | Operand is at the memory location whose addressing is the sum of a register and constant in the instruction |
| 3. Immediate | Operand is a constant within the instruction itself |
| 4. PC-relative | Address is the sum of PC and constant in the instruction |
| 5. Pseudo-direct | Jump address is the 26 bits of the instruction concatenated with the upper bits of the PC |




Note that a single operation can use more than one addressing mode. `Add`, for example, uses both immediate (addi) and register (add) addressing.

### Register / Direct Addressing

Direct addressing mode is when the register or memory location contains the actual values.

| Example |
| --- |
| `lw $t0, var1`<br>`lw $t2, var2` |

, **where**:

- Registers and variables `$0`, `$1`, `var1` and `var2` are all accessed in direct mode addressing.

### Immediate Mode

With actual value of the operands, rather than registers that are supposed to contain them.

| Example |
| --- |
| `li $t0, 57`<br>`add $t0, $t0, 57` |

### Indirection

`()` are used to denote an indirect memory access.

An indirect memory access means the CPU will read the provided address and then go to that address to access the value located there

```
la    $t0, lst
add   $t0, $t0, 4
lw    $s1, ($t0)
```

The address, in `$t0`, is a word size (32-bits). Memory is byte addressable. As such, if the data items in "lst" (from above) are words, then four must be added to get the next element.

With `add` it will get the next word value in array (named lst in this example).

| Displacement Addressing + Indirection |
| --- |
| A form of displacement addressing is allowed. For example, to get the second item from a list of long sized values: |
| `la $t0, lst`<br>`lw $s1, 4($t0)` |

### PC-relative Addressing

PC-relative addressing is used for conditional branches.

| op | rs | rt | address |
| --- | --- | --- | --- |
| | | | + pc |
| | | | Actual Address |

### Addressing Example

Example

```
##
##    Program Name:     min-max.s
##
##          - will print out the minimum value min
##          - and the maximum value max of an array.
##
##          - Assume the array has at least two elements (a[0] and
a[1]).
##          - It initializes both min and max to a[0] and then
##          - goes through the loop count - 1 times.
##          - This program will use pointers.
```

```
##
##
##          t0 - point to array elements in turn
##          t1 - contains count of elements
##          t2 - contains min
##          t3 - contains max
##          t4 - each word from array in turn
##

#################################################
#                                               #
#               text segment                    #
#                                               #
#################################################

        .text
        .globl __start
__start:                        # execution starts here

      la $t0,array              # $t0 will point to the elements
      lw $t1,count              # exit loop when $t1 == 0
      lw $t2,($t0)              # initialize min = a[0]
      lw $t3,($t0)              # initialize max = a[0]
      add $t0,$t0,4             # pointer to start at a[1]
      add $t1,$t1,-1            # and go round count - 1 times


loop: lw $t4,($t0)              # load next word from array
      bge $t4,$t2,notMin        # skip if a[i] >= min
      move $t2,$t4              # copy a[i] to min

notMin:     ble $t4,$t3,notMax      # skip if a[i] <= max
      move $t3,$t4              # copy a[i] to max


notMax:     add $t1,$t1,-1          # decrement counter
      add $t0,$t0,4             # increment pointer by word
      bnez $t1, loop            # continue if counter > 0


      la $a0,ans1               # print prompt on terminal
      li $v0,4          # system call to print
      syscall                   # out "min = "


      move $a0,$t2              # print result min
      li $v0,1
      syscall
```

```
      la $a0,ans2       # print out "max = "
      li $v0,4
      syscall

      move $a0,$t3            # print result max
      li $v0,1
      syscall

      la $a0,endl       # syscal to print out
      li $v0,4          # a new line
      syscall

      li $v0,10         # Exit
      syscall                 # Bye!


##################################################
#                                                #
#               data segment                     #
#                                                #
##################################################

        .data
      array:      .word 3,4,2,6,12,7,18,26,2,14,19,7,8,12,13
      count:      .word 15
      ans1: .asciiz "min = "
      ans2: .asciiz "\nmax = "
      endl: .asciiz "\n"

##
##    end of file min-max.s
```

## Implementation of Advanced Utilities

### ASCII String Operation (Such as, C)

#### ASCII String Overview

In this example, we would use the ASCII encoding scheme where each and every character is resented on a byte basis (0 - 7f(DEL))

there are three choices for re[resenting a sring:

(1) The first position of the string is reserved to give the length of a string
(2) (As in a Structure), accompanying variable has length of the string
(3) The last position of a string is indicated by a character used to mark the end of a string

| C's approach to strings |
| --- |
| C uses option (3) to represent a string, namely puts an 0 (`nul`) at the end of every string |

### Loading Strings

| | |
|---|---|
| lb | Load byte (halfword) |
| sb | Store byte |
| lbu | Load byte, unsigned |

```
lb rt, <offset>(rs) #sign extend to 32 bits
lbu rt, <offset>(rs) #zero extend to 32 bits
sb rt, <offset>(rs) #store rightmost byte
```

### Example: A String Copy Procedure

To implement the following algorithm (?):

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

Written with the following correspondance:

| x | y | i |
|---|---|---|
| $a0 | $a1 | $s0 |

```
strcpy:

    # (Adjust the stack pointer, save $s0 first)
    addi $sp, $sp, -4 # adjust stack for 1 item
    sw $s0, 0($sp) # save $s0

    # (Initialize i==0)
    add $s0, $zero, $zero # i = 0

    # Beginning of the Loop, form y[i] y adding $s0(i) to $a1(y)
    # (No need to *4, since it's based on byte not word)
L1: add $t1, $s0, $a1 # addr of y[i] in $t1

    # Then, load $t1(the offset)
    lb $t2, 0($t1) # $t2 = y[i]

    # Similarly, acquire address of x[i]
    add $t3, $s0, $a0 # addr of x[i] in $t3

    # Save y[i] in x[i]
    sb $t2, 0($t3) # x[i] = y[i]
```

```
    # Exit the loop if it's last character of the string (str == nul)
    beq $t2, $zero, L2 # exit loop if y[i] == 0

    # Else, increment it and repeat the loop
    addi $s0, $s0, 1 # i = i + 1

    j L1 # Start next iteration of loop
    # Exit condition
L2: lw $s0, 0($sp) # restore saved $s0
    addi $sp, $sp, 4 # pop 1 item from stack
    jr $ra # and return to $ra
```

## Unicode String Operations (Such as in Java)

### Unicode String overview

In Unicode, a character is represented by 16 bits:

`0x????`

(Two bytes), also called halfwords

### Loading Strings

| | |
|---|---|
| `lh` | Load halfword |
| `sh` | Store halfword |
| `lhu` | Load halfword, unsigned |

In Java includes a word that gives the length of the string, similar to Java arrays.

## Addressing Schemes Actually

### Branch Addressing

Branch instructions specify opcode two registers target address

`beq rs, rt, L1`

| PC-relative addressing (PC already incremented by 4): |
|---|
| `target address = PC + offset × 4` |

### Jump Addressing

`j` and `jal` targets could be anywhere in the code

| (Pseudo) direct addressing |
|---|
| target address = $PC_{31..28}$ : address × 4 |

### Target Addressing Example

To implement

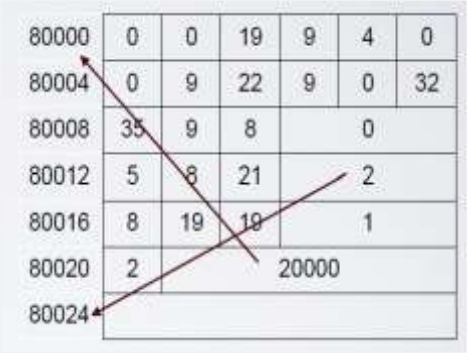
```
while (save[i] == k)
    i += 1;
```

,

With the following correspondence:

| i | k | save |
|---|---|---|
| $s3 | $s5 | $s6 |

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: …
```




### Branching Far Away

If branch target is too far to encode with 16-bit offet, assembler rewrites the code:

```
      beq $s0,$s1, L1
       ↓
      bne $s0,$s1, L2
      j L1
L2: …
```

### Address Mode Summary

```
addi $s3, $s3, 1
sub $s0, $s1, $s2
lw $t0, 32($s3)
bne $t0, $zero, L
j Exit
```

## Arrays

### Concepts

Arrays are stored in the Data Segment of a MIPS program.

The barebone implementation of an aray should support those three functionalities:

| Operation | C Equivalent |
|---|---|
| Getting data from an array cell | x = list[i]; |
| Storing data to an array cell | list[i] = x; |
| Determine length of an array | list.length |

### Array with .space, Storage Allocation

To reserve sufficient space for an array:

```
      .data
list: .space 1000 # reserves a block of 1000 bytes
```

This 1000 bytes has arbitrary type. For example, it could be used to contain:

- array of 1000 char values (ASCII codes)
- array of 250 int values
- array of 125 double values

### Array Declaration with Initialization

Array can be declared then defined (initialized) with a list of initualizers:

```
        .data
vowels: .byte 'a', 'e', 'i', 'o', 'u'
pow2:   .word 1, 2, 4, 8, 16, 32, 64, 128
```

, **where**:

- `pow2`, for example, declares a block of 32 bytes set to store 8 words (each with a size of 4 bytes)

### Array Initialization by Traversal

```
.data
    list: .space 1000
    listsz: .word 250 # using as array of integers
.text
    main: lw $s0, listsz # $s0 = array dimension
    la $s1, list # $s1 = array address
    li $t0, 0 # $t0 = # elems init'd
initlp: beq $t0, $s0, initdn
    sw $s1, ($s1) # list[i] = addr of list[i]
    addi $s1, $s1, 4 # step to next array cell
    addi $t0, $t0, 1 # count elem just init'd
    b initlp
initdn:
    li $v0, 10
    syscall
```

### Array Access

load word (lw) and store word (sw) are required to access data in the array.

Assuming that we are in If we have a declaration such as:

```
list: .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7
```
, the address

The following snippet of code will place the value of list[6] into the $t4:

```
    la $t3, list         # put address of list into $t3
    li $t2, 6            # put the index into $t2
    add $t2, $t2, $t2    # double the index
    add $t2, $t2, $t2    # double the index again (now 4x)
    add $t1, $t2, $t3    # combine the two components of the address
    lw $t4, 0($t1)       # get the value from the array cell
```

If we wish to assign to the contents of $t4 to list[6] instead, the last line would simply be:

```
    sw $t4, 0($t1)      # store the value into the array cell
```

## Breaking out

### Syscalls

#### Concepts

If a program was not terminated at the end, it may:

will blunder on through memory, interpreting whatever it finds as instructions to execute

… **which** is definitely what we'd want to happen.

The `syscall` instruction instructs the program o give control back to the OS., the OS would then look at register $v0 to see what the program asks it to do.

… for example,

```
      li $v0, 10 # syscall code 10 is for exit.
      syscall # make the syscall.
```

#### Syscall Codes

http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

| Service | Code | Arguments | Result |
|---|---|---|---|
| print integer | 1 | $a0 = value | (none) |
| print float | 2 | $f12 = float value | (none) |
| print double | 3 | $f12 = double value | (none) |
| print string | 4 | $a0 = address of string | (none) |
| read integer | 5 | (none) | $v0 = value read |
| read float | 6 | (none) | $f0 = value read |
| read double | 7 | (none) | $f0 = value read |
| read string | 8 | $a0 = address where string to | (none) |

| | | | |
|---|---|---|---|
| | | be stored<br>$a1 = number of characters to read + 1 | |
| memory allocation | 9 | $a0 = number of bytes of storage desired | $v0 = address of block |
| exit (end of program) | 10 | (none) | (none) |
| print character | 11 | $a0 = integer | (none) |
| read character | 12 | (none) | char in $v0 |
| open file | 13 | $a0 = address of null-terminated string containing filename<br>$a1 = flags<br>$a2 = mode | $v0 contains file descriptor (negative if error). *See note below table* |
| read from file | 14 | $a0 = file descriptor<br>$a1 = address of input buffer<br>$a2 = maximum number of characters to read | $v0 contains number of characters read (0 if end-of-file, negative if error). *See note below table* |
| write to file | 15 | $a0 = file descriptor<br>$a1 = address of output buffer<br>$a2 = number of characters to write | $v0 contains number of characters written (negative if error). *See note below table* |
| close file | 16 | $a0 = file descriptor | |
| exit2 (terminate with value) | 17 | $a0 = termination result | *See note below table* |
| time (system time) | 30 | | $a0 = low order 32 bits of system time<br>$a1 = high order 32 bits of system time. *See note below table* |

| | | | |
|---|---|---|---|
| MIDI out | 31 | $a0 = pitch (0-127)<br>$a1 = duration in milliseconds<br>$a2 = instrument (0-127)<br>$a3 = volume (0-127) | Generate tone and return immediately. *See note below table* |
| sleep | 32 | $a0 = the length of time to sleep in milliseconds. | Causes the MARS Java thread to sleep for (at least) the specified number of milliseconds. This timing will not be precise, as the Java implementation will add some overhead. |
| MIDI out synchronous | 33 | $a0 = pitch (0-127)<br>$a1 = duration in milliseconds<br>$a2 = instrument (0-127)<br>$a3 = volume (0-127) | Generate tone and return upon tone completion. *See note below table* |
| print integer in hexadecimal | 34 | $a0 = integer to print | Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary. |
| print integer in binary | 35 | $a0 = integer to print | Displayed value is 32 bits, left-padding with zeroes if necessary. |
| print integer as unsigned | 36 | $a0 = integer to print | Displayed as unsigned decimal value. |
| set seed | 40 | $a0 = i.d. of pseudorandom number generator (any int).<br>$a1 = seed for corresponding pseudorandom number generator. | No values are returned. Sets the seed of the corresponding underlying Java pseudorandom number generator (java.util.Random). *See note below table* |
| random int | 41 | $a0 = i.d. of pseudorandom number generator (any int). | $a0 contains the next pseudorandom, uniformly distributed int value from this random number generator's sequence. *See note below table* |

| | | | |
|---|---|---|---|
| random int range | 42 | $a0 = i.d. of pseudorandom number generator (any int).<br>$a1 = upper bound of range of returned values. | $a0 contains pseudorandom, uniformly distributed int value in the range 0 <= [int] < [upper bound], drawn from this random number generator's sequence. *See note below table* |

### Exception and Interrupt Instruction

| | |
|---|---|
| `trap 1` | Print integer value in $4 |
| `trap 5` | Read integer value into $2 |
| `trap 10` | Terminate program execution |
| `trap 101` | Print ASCII character in $4 |
| `trap 102` | Read ASCII character into $2 |

## Appendix: Pseudo-instructions

### Overview

#### Concepts

In MIPS, some operations can be performed with the help of other instructions. They can be coded in assembly language, and assembler will expand them to rel instructions.

#### Arithmetic

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| `abs dest src` | absolute value | addu $1, $2, $0<br>bgez $2, 8<br>(offset=8 → skip 'sub' instruction)<br>sub $1, $2, $0 | |
| `clear $t` | clear | or $t, $zero, $zero | t = 0 |
| `li $t, C` | load 16-bit immediate (when C can be in 16 bits) | ori $t, $zero, C_lo /<br>addiu $dst, 0, imm, etc. | t = C |
| `li $t, C` | load 32-bit immediate | lui $t, C_hi<br>ori $t, $t, C_lo | t = C |
| `la $t, A` | load label address | lui $t, A_hi<br>ori $t, $t, A_lo | t = A |

## Data Moves

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| `move $t, $s` | move | or $t, $s, $zero | t = s |
| `clear $t` | clear | or $t, $zero, $zero | t = 0 |
| `li $t, C` | load 16-bit immediate (when C can be in 16 bits) | ori $t, $zero, C_lo / addiu $dst, 0, imm, etc. | t = C |
| `li $t, C` | load 32-bit immediate | lui $t, C_hi<br>ori $t, $t, C_lo | t = C |
| `la $t, A` | load label address | lui $t, A_hi<br>ori $t, $t, A_lo | t = A |

## Branches

| Assembly syntax | Name | Expansion |
|---|---|---|
| `b C` | branch unconditionally | beq $zero, $zero, C |
| `bal C` | branch unconditionally and link | bgezal $zero, C |
| `bgt $s, $t, C` | branch if greater than | slt $at, $t, $s<br>bne $at, $zero, C |
| `blt $s, $t, C` | branch if less than | slt $at, $s, $t<br>bne $at, $zero, C |
| `bge $s, $t, C` | branch if greater than or equal | slt $at, $s, $t<br>beq $at, $zero, C |
| `ble $s, $t, C` | branch if less than or equal | slt $at, $t, $s<br>beq $at, $zero, C |
| `bgtu $s, $t, C` | branch if greater than unsigned | sltu $at, $t, $s<br>bne $at, $zero, C |
| `beqz $s, C` | branch if zero | beq $s, $zero, C |
| `beq $t, V, C` | branch if equal to immediate | ori $at, $zero, V<br>beq $t, $at, C |
| `bne $t, V, C` | branch if not equal to immediate | ori $at, $zero, V<br>bne $t, $at, C |
| `sge` | Set on greater or equals to | |
| `sgt` | Set on greater than | |

## Multiplication / Division

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| mul $d, $s, $t | multiplicate and return 32 bits | mult $s, $t<br>mflo $d | d = (s * t) & 0xFFFFFFFF |
| div $d, $s, $t | quotient | div $s, $t<br>mflo $d | d = s / t |

| | | | |
|---|---|---|---|
| rem $d, $s, $t | remainder | div $s, $t<br>mfhi $d | d = s % t |

### Jumps

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| jalr $s | jump register and link to ra | jalr $s, $ra | ra = PC + 4; goto s; |

### Logical Operations

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| not $t, $s | not | nor $t, $s, $zero | t = ~s |

### No-operations

| Assembly syntax | Name | Expansion | Operation in C |
|---|---|---|---|
| nop | nop | sll $zero, $zero, 0 | {} |

# Functioning MIPS

## Logic Structures in MIPS

### Overview

#### DeMorgan: Condition and Negated Condition

| Condition | Negated Condition |
|---|---|
| `x > y` | `x <= y` |
| `x >= y` | `x < y` |
| `x < y` | `x >= y` |
| `x <= y` | `x > y` |
| `<cond1> && <cond1>` | `! <cond1> \|\| ! <cond2>` |
| `<cond1> \|\| <cond1>` | `! <cond1> && ! <cond2>` |

### Conditional Structures

#### if-then-else

| Syntax in Higher-level Languages |
|---|
| `if (x < j) {`<br>`  k = k + i;`<br>`} else {`<br>`  k = k + j`<br>`}` |

Implementation in MIPS

```
    slt $5, $2, $2 # test i < j
    beq $5, $0, Else # If false goto Else
    addu $4, $4, $2 # k = k + i
    j Endif # goto Endif
Else:
    addu $4, $4, $3 #k = k + j
Endif:
```

## Loop

### Overview

There are two ways in which a conditional loop can be implemented

| The "break" method | The "continue" method |
|---|---|
| task:<br>// Tasks<br>If condition met, **break** (j exit)<br>// **Will not** be executed after **break**<br>j task<br><br>exit:<br>// **Will** be executed after **break** | task:<br>// Tasks<br>if condition is met, **continue** (j task)<br>// **Will** be executed after **break**<br>// i.e. if ! **continue**<br>exit: // Arbitraty |
| Legends | |
| | Loop |
| | May be executed |
| | Will not be touched |

### for

Syntax as in Higher-level Languages

```
for(<condition>){

    set of statements

    iteration++

}
```

MIPS Implementation

```
.data

msg1: .asciiz"A program in MIPS to test for loop. "
    # String to be printed, syntactically arbitrary
.text

    li $t0,10      # Initializes counter with value 10
```

```
    la $a0,msg1    # Initializes message with .asciiz
for_loop:
    blt $t0,1,Exit # When counter < 1, branck to Exit

    li $v0,4
    syscall
    sub $t0,$t0,1 # Loop body, task to be done

    j for_loop

Exit:
li $v0,10
syscall
```

## while

Syntax as in Higher-level Languages

```
while (<condition>) {
    set of statements
}
```

MIPS Implementation

User is asked to input an integer value continuously
Exits if value entered is less than 10

```
.data
msg1: .asciiz "Enter a number: "

.text


    li $v0,4
    la $a0,msg1

while:

    syscall // Prints
    li $v0,5
    syscall
    bgt $v0,10,print
    li $v0,10
    syscall

    j while

print:
    li $v0,1
    move $a0,$v0
    syscall
```

```
    j while
```

### do-while

Syntax as in Higher-level Languages

```
do{
    set of statements
}

while(<condition>){
    set of statements
}
```

MIPS Implementation

do is executed no matter what
Condition check is stored in while,
  If certain condition is met, jump to do

```
.data
msg1: .asciiz "This is the example of a do-while loop in MIPS
assembly. "
msg2: .asciiz "Enter a number: "
.text

li $v0,4
la $a0,msg2
syscall

li $v0,5
do:
syscall
li $v0,4
la $a0,msg1
syscall

while:
bgt $v0,1,exit
j do

exit:
li $v0,10
syscall
```

### Switch

Syntax as in Higher-level Languages

```
switch( i ) {
case 1: i++ ;  // falls through
case 2: i += 2 ;
```

```
        break;
case 3: i += 3 ;
}
```

MIPS Implementation

```
         addi $r4, $r0, 1      # set temp to 1
         bne $r1, $r4, C2_COND # case 1 false: branch to case 2 cond
         j C1_BODY             # case 1 true: branch to case 1
C2_COND: addi $r4, $r0, 2      # set temp to 2
         bne $r1, $r4, C3_COND # case 2 false: branch to case 2 cond
         j C2_BODY             # case 2 true: branch to case 2 body
C3_COND: addi $r4, $r0, 3      # set temp to 3
         bne $r1, $r4, EXIT    # case 3 false: branch to exit
         j C3_BODY             # case 3 true: branch to case 3 body
C1_BODY: addi $r1, $r1, 1      # case 1 body: i++
C2_BODY: addi $r1, $r1, 2      # case 2 body: i += 2
         j EXIT                # break
C3_BODY: addi $r1, $r1, 3      # case 3 body: i += 3
EXIT:
```

## Composite Conditionals via Short-circuiting

### A & B

Syntax as in Higher-level Languages

```
if ( i == j <cond1> && i == k <cond2> )
  i++ ;  // if-body
else
  j-- ;  // else-body
j = i + k ;
```

Behavior

| As <cond1> == false | As <cond1> == true | |
|---|---|---|
| Short-circuiting occurs<br>j else | Evaluate <cond2> | |
| | <cond2>==false | <cond2>==true |
| | j else | j if |

MIPS Implementation

```
      bne  $r1, $r2, ELSE   # cond1: branch if ! ( i == j )
      bne  $r1, $r3, ELSE   # cond2: branch if ! ( i == k )
      addi $r1, $r1, 1      # if-body: i++
      j L1                  # jump over else
ELSE: addi $r2, $r2, -1     # else-body: j--
L1:   add $r2, $r1, $r3     # j = i + k
```

### A | B

Syntax as in Higher-level Languages

```
if ( i == j <cond1> || i == k<cond2> )
  i++ ;   // if-body
else
  j-- ;   // else-body
j = i + k ;
```

Behavior

| As <cond1> == true | As <cond1> == false | |
|---|---|---|
| Short-circuiting occurs<br>j if | Evaluate <cond2> | |
| | <cond2>==false | <cond2>==true |
| | j else | j if |

MIPS Implementation

```
      beq  $r1, $r2, IF      # cond1: branch if ( i == j )
      bne  $r1, $r3, ELSE    # cond2: branch if ! ( i == k )
IF:   addi $r1, $r1, 1       # if-body: i++
      j L1                   # jump over else
ELSE: addi $r2, $r2, -1      # else-body: j--
L1:   add $r2, $r1, $r3      # j = i + k
```

## Advanced Utilities in MIPS

### Integer IO

#### Input

```
li      $v0,5      # code 5 == read integer
syscall            # Invoke the operating system.
                   # Read in one line of ascii characters.
                   # Convert them into a 32-bit integer.
                   # $v0 <-- two's comp. int.
```

#### Output

```
li      $v0,1      # code 1 == print integer
lw      $a0,int    # $a0 == the integer
syscall            # Invoke the operating system.
                   # Convert the 32-bit integer into characters.
                   # Print the character to the monitor.
```

#### Example Program

```
# ounces.asm
#
# Convert ounces to pounds and ounces.

        .text
        .globl  main
```

```
main:   li      $v0,4       # print prompt
        la      $a0,prompt  #
        syscall
        li      $v0,5       # read in ounces
        syscall

        li      $t1,16      # 16 oz. per pound
        divu    $v0,$t1     # lo = pound; hi = oz.

        mflo    $a0
        li      $v0,1       # print
        syscall             # pounds
        li      $v0,4       # print "pounds"
        la      $a0,pout
        syscall

        mfhi    $a0         # print
        li      $v0,1       # ounces
        syscall             #
        li      $v0,4       # print
        la      $a0,ozout   # "ounces"
        syscall

        li      $v0,10      # exit
        syscall

        .data
prompt: .asciiz "Enter ounces: "
pout:   .asciiz " Pounds\n"
ozout:  .asciiz " Ounces\n"
```

## String IO

### String Read

The exception handler can also read in a string from the keyboard.

```
li      $v0,8       # code 8 == read string
la      $a0,buffer  # $a0 == address of buffer
li      $a1,16      # $a1 == buffer length
syscall             # Invoke the operating system.

 . . . .

        .data
buffer: .space 16   # reserve 16 bytes
```

Details: Register $a1 contains the length (in bytes) of the input buffer. Up to ($a1)-1 characters are read from the keyboard and placed in buffer as a null terminated string.

The user ends the string by hitting "enter". The "enter" character appears in the buffer as the newline character '\n', 0x0a. This byte is followed by the null byte 0x00. If the user enters a string that is exactly ($a1)-1 characters long the newline character is omitted from the buffer. No matter what, there is a null at the end of data in the buffer.

**Example Program**

```
# overdue.asm

        .text
        .globl  main

main:
        # get patron name
        li      $v0,4           # print prompt
        la      $a0,prompt      #
        syscall
        li      $v0,8           # code 8 == read string
        la      $a0,name        # $a0 == address of buffer
        li      $a1,24          # $a1 == buffer length
        syscall                 # Invoke the operating system.

        # print the letter
        li      $v0,4           # print greeting
        la      $a0,letter      #
        syscall
        li      $v0,4           # print body
        la      $a0,body        #
        syscall

        li      $v0,10      # exit
        syscall

        .data
prompt: .asciiz "enter name, followed by comma-enter: "
letter: .ascii  "\n\nDear "
name:   .space  24

body:   .ascii  "\nYour library books are way\n"
        .ascii  "overdue.  Please return them\n"
        .ascii  "before we give your name\n"
        .asciiz "to the enforcement squad.\n\n"
```

# QtSpim: The Environment

## MIPS @ QtSpim

### Overview

#### Qtspim: The Bare Minimum

`[FP Regs]` `[Int Regs [16]]`: Floating point / Integer Registers

`[Data]` `[Text]`: Text / Data Segment

`<Bottom Section>`: Qtspim Execution Log

#### Execution

Load File:

- File ⟶ Simulator

Reinitialize and Load File

- For debugging purposes

#### Simulator Settings

Simulator - Settings

- Qtspim: Font, Background color, # of recent files
- MIPS:
  - Bare machine: qtspim simulation
  - Accept pseudo instructions
  - Enable delayed branches:
    - Whether or not to delay branching actions such as `bne` and `ben`
  - Enable delayed loads
  - Enable mapped I/O

Simple Machine (for normal users) vs. Bare Machine (Real MIPS processor)

### Program Template

#### Basic Template

```
# Name and general description of program
# ---------------------------------------
# Data declarations go in this section.
.data
# program specific data declarations
# ---------------------------------------
# Program code goes in this section.
```

```
.text
.globl main
.ent main
main:
# -----
# your program code goes here.
# -----
# Done, terminate program.
li $v0, 10
syscall # all done!
.end main
```

## Assembler Directives

### Concepts

An assembler directive is  message to the assembler that tells something it needs to know in order to carry out the assembly process.

They are prefixed with a `.`

This includes:

- Noting where data is declared
- Noting where the code is defined
- Defining the start and end of procedures/functions,  such as .data and .text
- Used to declare data

## Application of Data Directives: Data Declarations

### Data Declarations Keywords

All variables and constants should be stored in the `.data` section.

| Declaration | |
|---|---|
| `.byte` | 8-bit variable |
| `.half` | 16-bit variable |
| `.word` | 32-bit variable |
| `.ascii` | ASCII string |
| `.asciiz` | NULL terminated ASCII string |
| `.float` | 32-bit IEEE floating-point number |
| `.double` | 64-bit IEEE floating-point number |
| `.space <n>` | <n> bytes of uninitialized memory |

### Integer Data Declaration

Integer values are defined with .word, .half or .byte, two's complement form represents negative numbers

The following examples declare registers

```
wVar1: .word 500000
wVar2: .word -100000

hVar1: .half 5000
hVar2: .half -3000

bVar1: .byte 5
bVar2: .byte -3
```

### Floating-point Data Declarations

```
pi:   .float      3.14159
tao:  .double     6.28318
```

### String Data Declaration

To define a series of concatenated strings, the NULL-terminated one must be placed last:

```
message:    .ascii "Line 1: Goodbye World\n"
            .ascii "Line 2: So, long and thanks "
            .ascii "for all the fish.\n"
            .asciiz "Line 3: Game Over.\n"
```

### Constants

Constants are created with the `=` sign, for example:

```
TRUE = 1
FALSE = 0
```

Constants are conventionally (but not necessarily) all UPPER CASE.

## Application of Data Directives: Segment Declaration

### Program Code

The code must be preceded by the `.text` directive.

| | |
|---|---|
| `.text` | Text (a.k.a. code) segment begins |
| `.globl <name>` | Defines name of initial/main procedure |
| `.ent <name>` | Similar to .globl, [optional] |

A procedure (such as main) should be ended with the `.end <name> directive`.

### Code Example

```
# Name and general description of program
# ---------------------------------------
# Data declarations go in this section.
.data
# program specific data declarations
# ---------------------------------------
```

```
# Program code goes in this section.
.text
.globl main
.ent main
main:
# -----
# your program code goes here.
# -----
# Done, terminate program.
li $v0, 10
syscall # all done!
.end main
```

### Calculation Example

```
# Assuming the following data declarations:
num: .word 0
wnum: .word 42
hnum: .half 73
bnum: .byte 7
wans: .word 0
hans: .half 0
bans: .byte 0
# To perform, the basic operations of:
num = 27
wans = wnum
hans = hnum
bans = bnum
# The following instructions could be used:
li $t0, 27
sw $t0, num # num = 27
lw $t0, wnum
sw $t0, wans # wans = wnum
lh $t1, hnum
sh $t1, hans # hans = hnum
```

## Labels

### Concepts

Labels are cod locations, typically used as:

- To indicate a function/procedure name, or
- as target of a jump

| {WARNING} |
| --- |
| Note that Labels are case-sensitive, meaning Loop and loop can be two separate things. |

When a label appears along on a line, it refers to the following memory location.

**main:**

The label main: should point to the first instruction of the progrm, such as:

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
#     leaving the result in register $t0.
# Registers used:
#     t0 - used to hold the result.
#     t1 - used to hold the constant 1.
main: li $t1, 1 # load 1 into $t1.
      add $t0, $t1, 2 # $t0 = $t1 + 2.
# end of add.asm
```