**General Purpose Registers**

| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | Constant value 0 |
| $v0-$v1 | 2-3 | Values for results and expression evaluation |
| $a0-$a3 | 4-7 | Arguments |
| $t0-$t7 | 8-15 | Temporaries |
| $s0-$s7 | 16-23 | Program variables |
| $t8-$t9 | 24-25 | More temporaries |
| $gp | 28 | Global pointer |
| $sp | 29 | Stack pointer |
| $fp | 30 | Frame pointer |
| $ra | 31 | Return address |

| Operation | Opcode in MIPS | Meaning |
|---|---|---|
| Addition | `add $rd, $rs, $rt` | `$rd = $rs + $rt` |
| | `addi $rt, $rs, C16`$_{2s}$ | `$rt = $rs + C16`$_{2s}$ |
| Subtraction | `sub $rd, $rs, $rt` | `$rd = $rs - $rt` |
| Shift left logical | `sll $rd, $rt, C5` | `$rd = $rt << C5` |
| Shift right logical | `srl $rd, $rt, C5` | `$rd = $rt >> C5` |
| AND bitwise | `and $rd, $rs, $rt` | `$rd = $rs & $rt` |
| | `andi $rt, $rs, C16` | `$rt = $rs & C16` |
| OR bitwise | `or $rd, $rs, $rt` | `$rd = $rs \| $rt` |
| | `ori $rt, $rs, C16` | `$rt = $rs \| C16` |
| NOR bitwise | `nor $rd, $rs, $rt` | `$rd = $rs ↓ $rt` |
| XOR bitwise | `xor $rd, $rs, $rt` | `$rd = $rs ^ $rt` |
| | `xori $rt, $rs, C16` | `$rt = $rs ^ C16` |

**R-format**

- arith $rd, $rs, $rt
- **shift** $rd, $rt, shamt. **($rs=0)!!!!**

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

| Fields | Meaning |
|---|---|
| **opcode** | Partially specifies the instruction<br>Equal to **0** for all **R-Format** instructions |
| **rs** | Specify register containing first operand |
| **rt** | Specify register containing second operand |
| **rd** | Specify register which will receive result of computation |
| **shamt**<br>C5<br>[0, $2^5$-1] | Amount a shift instruction will shift by 5 bits (i.e. 0 to 31)<br>Set to **0** in all non-shift instructions |
| **funct** | Combined with opcode exactly specifies the instruction |

Arithmetic operands (for **add**) are **registers**, not memory!

**I-format**

- arith $rt, $rs, C16$_{2s}$
- ld/st $rt, C16$_{2s}$($rs)
- logic $rt, $rs, C16
- branch $rs, $rt, label

**Remark:**

- C16$_{2s}$ is in 2s complement [sign-extended]
- C16 is raw binary (*no negative*) [NOT sign-extended]
- label is converted to number first (*PC-relative addressing*)

| opcode | rs | rt | Immediate |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

**ATTENTION: branch operation is $rs, $rt, label**

| Fields | Meaning |
|---|---|
| **rt** | specifies register to receive result different from R-format instructions |
| **Immediate**<br>C16$_{2s}$<br>[-$2^{15}$, $2^{15}$-1] | a ***signed 2s' complement integer***, ***Except*** for bitwise operations (**andi, ori, xori**)<br>16 bits → represent up to $2^{16}$ values<br>**Large enough to handle**:<br>The offset in a typical **lw** or **sw**<br>Most of the values used in the **addi, slti** instructions<br>- But **not enough** to specify the entire target address! (16 bits vs 32 bits address)<br>- **addi** will not work properly as the processor can only work with 32-bits |

Can branch to ± $2^{15}$ bytes from the PC:

**branch calculation:**

- If the branch is not taken: PC = PC + 4
- If the branch is taken: PC = (PC + 4) + (immediate × 4)

**Remark:**

- branches use **PC-relative addressing**
- load/store use **base addressing**

**J-format**

- **j L1** (Technically equivalent to **beq $s0, $s0, L1)**

| opcode | target address |
|---|---|
| 6 | 26 |

**Optimization:**

1) jumps will only jump to word-aligned addresses, so last 2 bits are always 00
2) Assume the address ends with '00' and leave them out
3) Now we can specify **28 bits** of 32-bit address

**28 bits→32 bits (32 bits = 4 bytes):**

4) MIPS choose to take the 4 MSB from **PC+4.**

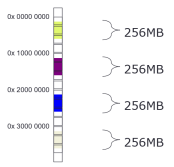
This means that we cannot jump to anywhere in memory, but it should be sufficient *most of the time*

**Address Format:**

| 4 bits MSB | Target Address (26bits) | **00** |
|---|---|---|

**256MB Boundary:** Due to the use of the first 4-bits from the PC, we can only jump within our block.

- 16 blocks in total, each block contains $2^{26}$ instructions.
- If you are at the top of the boundary, cannot jump up.
- If you are at the bottom of the boundary, cannot jump down.



**Pseudo-Instruction** "Fake" instruction that gets translated to corresponding MIPS instruction(s). Provided for convenience in coding only. i.e. move $s0, $s1

**Inequality:**

slt $t0, $s1, $s2 ⟺ $t0 = ($s1 < $s2) ? 1 : 0;

**Condition**

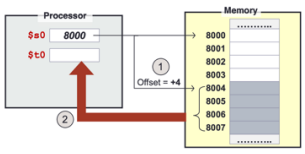
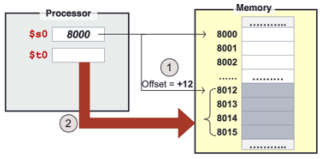
**Common technique:** Invert the condition for shorter code

| | |
|---|---|
| slt $t0, $s1, $s2<br>bne $t0, $zero, L | if ($s1 < $s2) goto L; |
| slt $t0, $s2, $s1<br>**beq** $t0, $zero, L | if ($s1 <= $s2) goto L; |
| slt $t0, $s2, $s1<br>beq $t0, $zero, L | if ($s1 > $s2) goto L; |
| slt $t0, $s1, $s2<br>**beq** $t0, $zero, L | if ($s1 >= $s2) goto L; |

**Memory Address in MIPS**

- Given a ***k*-bit** address, the address space is of size $\mathbf{2^k}$.
- Memory address are 32 bits long. $2^{30}$ memory **words(4-bytes)** in total [0, $2^{30}$], consecutive words **differ by 4**.
- Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

**Word Alignment:** for N-bits word, use **Address % N** to check

| load word (lw) | Store word (sw) |
|---|---|
|  |  |
| 1. **$s0** + 4 = **8004**<br>2. load **Mem[8004]** to **$t0** | 1. **$s0** + 12 = **8012**<br>2. Content of **$t0** is stored into word at **Mem[8012]** |

- Only **load** and **store** can access data in memory.
- MIPS disallows load/store unaligned word using **lw/sw**

**Address Mode**

(1) **Register addressing**: operand is a register

MIPS: add, sub, and, or, xor, nor, slt, sll, srl


(2) **Immediate addressing:** operand is a constant within the instruction itself

MIPS: addi, andi, ori, xori, slti


(3) **Base addressing** (displacement addressing): operand is at the memory location whose address is sum of a register and a constant in the instruction.

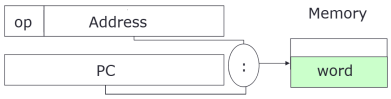
MIPS: lw, sw


(4) **PC-relative addressing**: address is sum of PC and constant in the instruction.

MIPS: beq, bne (branch)


(5) **Pseudo-direct addressing**: 26-bit of instruction concatenated with upper 4-bits of PC

MIPS: j




**Bit-wise Operation**

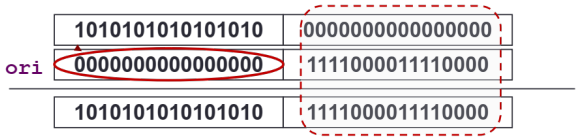
**Large Constant:** load a 32-bit constant into a register

1. Use "load upper immediate" (lui) to set the upper 16-bit:

```
lui   $t0, 0xAAAA      #1010101010101010
```


2. Use "or immediate" (ori) to set the lower-order bits:

```
ori   $t0, $t0, 0xF0F0 #1111000011110000
```




**Shifting:** $N \in [0,5]$

**[T]** we can use shift operation as multiple/divide operation

| MIPS | C | Math |
|---|---|---|
| sll $s0, $s0, N | a <<= N | a = a * $2^N$ |
| srl $s0, $s0, N | a >>= N | a = a / $2^N$ |

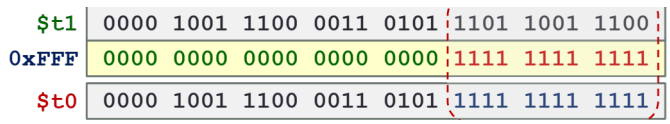
**Logic:**

| a | b | AND | OR | NOR | XOR |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **1** | 0 |
| 0 | 1 | 0 | 1 | 0 | **1** |
| 1 | 0 | 0 | 1 | **0** | 1 |
| 1 | 1 | 1 | 1 | 0 | **0** |

(1) **AND:** masking operation

- Place 0s into positions to be ignored→bits turn into 0s
- Place 1s for interested positions→bits will remain same

(2) **OR:** places a 1 in the result if either operand bit is 1




(3) **NOR**: nor $t0, $t0, $zero⟺not $t0, $t0

**Remark:** can only inverse, as **NO** NORI in MIPS

(4) **XOR:** xor $t0, $t0, $t2 ($t2 = 11111….11)⟺not $t0, $t0

**Application:** XORI, inverse target place(use 1), others remain same(use 0)

**Remark:**

(a) There is no NORI, but there is XORI in MIPS, not much need for NORI.

(b) There is no NOT instruction in MIPS

**[T]** Copy over bits 1, 3 and 7 of $s1 into $s0, without changing any other bits of $s0.

**Step 1:** Use the property that x AND 1 = x to copy out the values of bits 7, 3 and 1 of b into $t0. Note that we **zero all other bits** so that they don't change anything in $s0 when we OR later on.

andi $t0, $s1, 0b00000000**1**000**1**0**1**0

**Step 2:** We use the property of x OR 0 = x to **copy** in the bits into a, so we prepare a by zero-ing bits 7, 3 and 1. To do this we need the mask 1111111111111111 1111111101110101

lui $t1, 0b1111111111111111
ori $t1, $t1, 0b11111111**0**111**0**1**0**1
and $s0, $s0, $t1

**Step 3:** Now OR together a and $t0 to copy over the bits
or $s0, $s0, $t0