String toUpper

Convert a string to all upper case

```
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= ‘a’ && *index <= ‘z’)
        *index = *index +(‘A’ - ‘a’);
    index++;
}
```

```
# $t0=index, $t2=‘a’, $t3=‘z’, $t4=‘A’-’a’, Memory[100]=string
 lw      $t0, 100($zero)        # index = string
 LOOP:
        lbu     $t5, 0($t0)             # t5 = *index
        beq     $t5, $zero, END         # exit if *index == 0
        slt     $t6, $t5, $t2           # is *index < ‘a’?
        bne     $t6, $zero, NEXT        # don’t change if < ‘a’
        slt     $t6, $t3, $t5           # is ‘z’ < *index?
        bne     $t6, $zero, NEXT        # don’t change if ‘z’ < *index
        add     $t5, $t5, $t4           # t5 = *index + (‘A’ - ‘a’)
        sb      $t5, 0($t0)             # *index = new value;
 NEXT:
        addi    $t0, $t0, 1             # index++;
        j       LOOP                    # continue the loop
 END:
```

Machine Language vs. Assembly Language

Assembly Language

- mnemonics for easy reading
- labels instead of fixed addresses
- Easier for programmers
- Almost 1-to-1 with machine language

Machine language

- Completely numeric representation
- format CPU actually uses

```
SWAP:
    sll   $2, $5, 2
    add   $2, $4, $2      // Compute address of v[k]
    lw    $15, 0($2)      // get v[k]
    lw    $16, 4($2)      // get v[k+1]
    sw    $16, 0($2)      // save new value to v[k]
    sw    $15, 4($2)      // save new value to v[k+1]
    jr    $31             // return from subroutine
```

```
000000 00000 00101 00010 00010 000000
000000 00100 00010 00010 00000 100000
100011 00010 01111 00000 00000 000000
100011 00010 10000 00000 00000 000100
101011 00010 10000 00000 00000 000000
101011 00010 01111 00000 00000 000100
000000 11111 00000 00000 00000 001000
```

Labels

Labels specify the address of the corresponding instruction

- Programmer doesn't have to count line numbers
- Insertion of instructions doesn't require changing entire code

```
# $t0 = N, $t1 = sum, $t2 = I
   add   $t1, $zero, $zero   # sum = 0
   add   $t2, $zero, $zero   # I = 0
TOP:
   bne   $t0, $t2, END       # I!=N
   add   $t1, $t1, $t2       # sum += I
   addi  $t2, $t2, 1         # I++
   j     TOP                 # next iteration
END:
```

implicit

Notes:

[31:26]

- Jumps are pseudo-absolute:

  PC = { PC[~~31:28~~], 26-bit unsigned-Address, "00" }

- Branches are PC-relative:

  PC = PC + 4 + 4*(16-bit signed Address)

Instruction Types

Can group instructions by # of operands

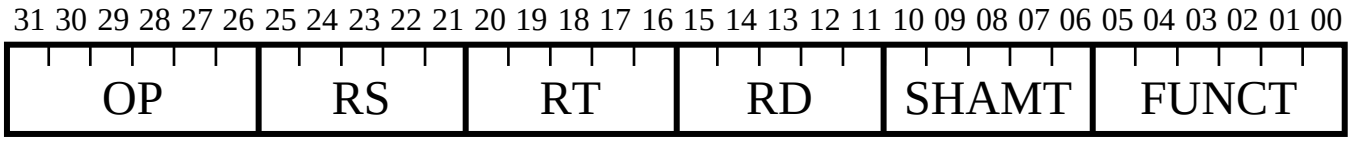
- 3-register
- 2-register
- 1-register
- 0-register

```
add   $t0, $t1, $t2      # t0 = t1+t2
addi  $t0, $t1, 100      # t0 = t1+100
and   $t0, $t1, $t2      # t0 = t1&t2
andi  $t0, $t1, 7        # t0 = t1&b0111
sllv  $t0, $t1, $t2      # t0 = t1<<t2
sll   $t0, $t1, 6        # t0 = t1<<6
lw    $t0, 12($t1)       # $t0 = Memory[$t1+10]
sw    $t2, 8($t3)        # Memory[$t3+10] = $t2
j     25                 # go to 100 - PC = 25*4 (instr are 32-bit)
jr    $ra                # go to address in $ra - PC = value of $ra
beq   $t0, $t1, FOO      # if $t0 == $t1 GOTO FOO - PC = FOO
bgez  $t0, FOO           # if $t0 >= 0 GOTO FOO - PC = FOO
slt   $t0, $t1, $t2      # if ($t1 < $t2) $t0 = 1 else $t0 = 0;
```

Instruction Formats

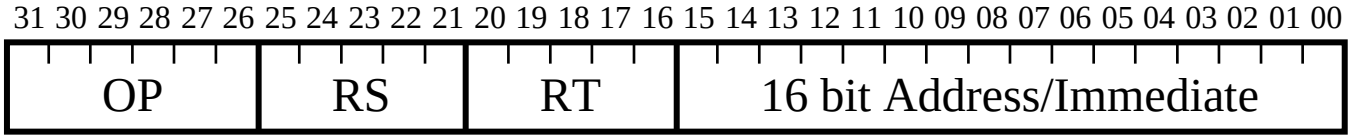
All instructions encoded in 32 bits (operation + operands/immediates)

Register (R-type) instructions



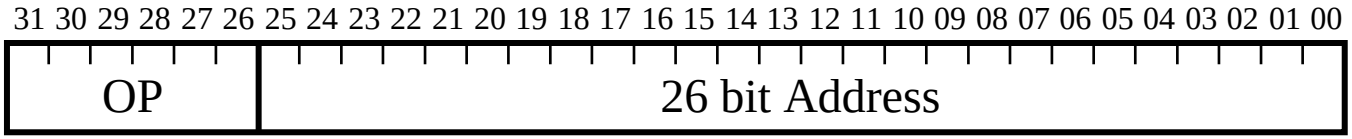
(OP = 0,16-20)

Immediate (I-type) instructions



(OP = any but 0,2,3,16-20)

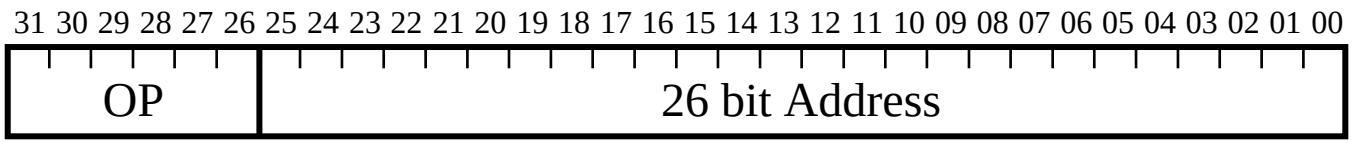
Jump (J-type) instructions



(OP = 2,3)

J-Type

Used for unconditional jumps



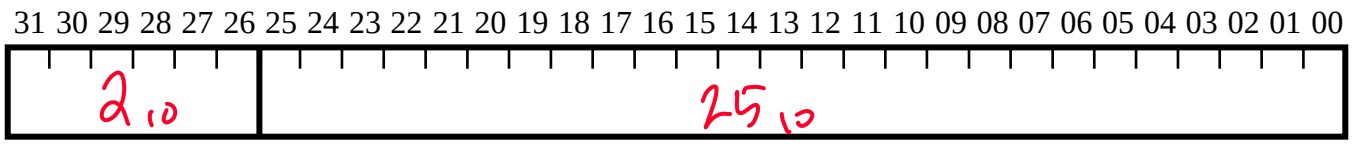
2: j (jump)

3: jal (jump and link)

Note: top ~~4~~ 6 bits of jumped-to address come from current PC

Example:

```
j     25       # go to 100, PC = 25*4 (instr are 32-bit)
```



I-Type

Used for operations with immediate (constant) operand

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
OP	RS	RT	16 bit Address/Immediate
	Op1, L/S addr	Op2, Dest, L/S targ	

04: beq
05: bne
06: blez
07: bgtz
08: addi
09: addiu
10: slti
11: sltiu
12: andi
13: ori
14: xori
32: lb
35: lw
40: sb
43: sw

```
addi    $8, $9, 100        # $8 = $9+100
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
08_{10}	9_{10}	8_{10}	100_{10}

```
beq     $8, $9, -11        # if $8 == $9 GOTO (PC+4+FOO*4)
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
04_{10}	8_{10}	9_{10}	-11_{10}

```
lw      $8, 12($9)         # $8 = Memory[$9+12]
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
35_{10}	9_{10}	8_{10}	12_{10}

R-Type

Used for 3 register ALU operations

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
OP	RS	RT	RD	SHAMT	FUNCT
00 (16-20 for FP)	Op1	Op2	Dest	Shift amount (0 for non-shift)	

```
add     $8, $9, $10      # $8 = $9+$10
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
00	9	10	8	0	32

```
sll     $8, $9, 6        # $8 = $9<<6
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
00	X	9	8	6	00

```
sllv    $8, $9, $10      # $8 = $9<<$10
```

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
00	10	9	8	0	4

00:	sll
02:	srl
03:	sra
04:	sllv
06:	srlv
07:	srav
08:	jr
24:	mult
26:	div
32:	add
33:	addu
34:	sub
35:	subu
36:	and
37:	or
38:	xor
39:	nor
42:	slt