## Useful arithmetic and logical operations for the PIC32 processor

In these examples

$d, $s, and $t are PIC registers (see chart at end)
imm is a 16-bit literal
shamt is a 5-bit shift specifier

Some binary operations

| PIC instruction | C equivalent | Format |
|---|---|---|
| add $d,$s,$t | $d = $s + $t | R (100000) |
| addi $t,$s,imm | $t = $s + imm | I (001000) |
| and $d,$s,$t | $d = $s & $t | R (100100) |
| andi $t,$s,imm | $t = $s & imm | I (001100) |
| nor $d,$s,$t | $d = ~($s \| $t) | R (100111) |
| or $d,$s,$t | $d = $s \| $t | R (100101) |
| ori $t,$s,imm | $t = $s \| imm | I (001101) |
| sll $d,$t,shamt | $d = $t << shamt | R (000000) |
| sllv $d,$t,$s | $d = $t << $s | R (000100) |
| slt $d,$s,$t | $d = $s < $t | R (101010) |
| slti $t,$s,imm | $t = $s < imm | R (101001) |
| sra $d,$t,shamt | $d = $t >> shamt (signed) | R (000011) |
| srav $d,$t,$s | $d = $t >> $s (signed) | R (000111) |
| srl $d,$t,shamt | $d = $t >> shamt (unsigned) | R (000010) |
| srlv $d,$t,$s | $d = $t >> $s (unsigned) | R (000110) |
| sub $d,$s,$t | $d = $s - $t | R (100010) |
| xor $d,$s,$t | $d = $s ^ $t | R (100101) |
| xori $t,$s,imm | $t = $s ^ imm | I (001101) |

Some memory instructions

| PIC instruction | C equivalent | Format |
|---|---|---|
| lb $t,imm($s) | $t = MemByte[$s+imm] | I (100000) |
| lbu $t,imm($s) | $t = MemWord[$s+imm] | I (100100) |
| lw $t,imm($s) | $t = MemWord[$s+imm] | I (100011) |
| sb $t,imm($s) | MemByte[$s+imm] = $t | I (101000) |
| sw $t,imm($s) | MemWord[$s+imm] = $t | I (101011) |

Notes:
The immediate (imm) field is zero-extended for andi, ori, and xori In all other cases, immediate values are sign-extended.
The memory byte is zero-extended for lbu and sign-extended for lb.
The sra and srav do a signed shift. The srl and srlv do an unsigned shift.
The instructions slt and slti set the destination register to either 0 or 1.

Some control flow instructions

| PIC instruction | C equivalent | Format |
|---|---|---|
| `beq $s,$t,imm` | `Branch when $s==$t` | `I (000100)` |
| `bgtz $s,imm` | `Branch when $s>0` | `I (000111)` |
| `blez $s,imm` | `Branch when $s<=0` | `I (000110` |
| `bne $s,$t,imm` | `Branch when $s!=$t` | `I (000101)` |
| `j addr` | `Goto label` | `J (000010)` |
| `jal addr` | `Jump and link to label` | `J (000011)` |
| `jalr $s` | `Jump and link with register` | `R (001001)` |
| `jr $s` | `Goto with register` | `R (001000)` |

On the control flow branch instructions, the computation of the target varies. For most of these instructions, the target depends on the value of the PC, or program counter, which contains the address of the present instruction.

- For the R instructions, `jalr` and `jr`, the target is simply the value of the register `$s`. These instructions are generally used in calls to functions.
- For the J instructions, `j` and `jal`, the target is obtained by first taking the 26-bit `addr` field and shifting it over 2 places. This give 28 bits, of which the bottom two are always zero. Next, the top 4-bits of the PC are added to give the entire 32 bits.
  - `newPC = (addr<<2) | (PC & 0xF0000000)`
- For the I instructions, `beq`, `bgtz`, `blez` and `bne`, the 16-bit `imm` field is sign-extended and shifted two places and then added to PC+4, that is, the location of the next instruction.
  - `newPC = (SignExtend(imm) << 2) + PC + 4`

The branch instructions are much easier than you think. Count the number of instructions from the present instruction to the target instruction: If the target lies before the present instruction, that count should be negative. Subtract one from that count. The result becomes the `imm` field.

Instruction formats for encoding

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Register (R) encoding | `000000` | | | | | | `$s` | | | | | `$t` | | | | | `$d` | | | | | `shamt` | | | | | ***function*** | | | | | |
| Immediate (I) encoding | ***opcode*** | | | | | | `$s` | | | | | `$t` | | | | | `imm` | | | | | | | | | | | | | | | |
| Jump (J) encoding | ***opcode*** | | | | | | `addr` | | | | | | | | | | | | | | | | | | | | | | | | | |

Register names

| | | |
|---|---|---|
| `$zero` | 0 | the value 0 |
| `$at` | 1 | Assembly temporary – do not use |
| `$v0-$v1` | 2 - 3 | For function results |
| `$a0-$a3` | 4 - 7 | Arguments to functions – Caller saved |
| `$t0-$t7` | 8 - 15 | Temporaries – Caller saved |
| `$s0-$s7` | 16 - 23 | Saved – Called saved |
| `$t8-$t9` | 24 - 25 | Temporaries – Caller saved |
| `$k0-$k1` | 26 - 27 | Kernel register – do not use |
| `$gp` | 28 | Global pointer |
| `$sp` | 29 | Stack pointer |
| `$fp` | 30 | Frame pointer |
| `$ra` | 31 | Return address |