Flow of control – Conditional branch instructions

You can compare on

- equality or inequality of two registers
- comparison of a register with 0 ($>, <, \geq, \leq$)

and branch to a target that is:

- a signed displacement (expressed in *number of instructions*, not in number of bytes) from the instruction *following* the branch.
- In assembly language just put your target address because computing the number of instructions is difficult (some pseudoinstructions translate into 2 real instructions)

Opcode	Operands	Comments
BEQ	rs,rt,target	#branch to target if rs $=$ rt
BNE	rs,rt,target	#branch to target if rs $\neq$ rt
BGTZ	rs,target	#branch to target if rs > 0
BLTZ	rs,target	#branch to target if rs < 0
BLEZ	rs,target	#branch to target if rs ≤ 0
BGEZ	rs,target	#branch to target if rs ≥ 0

Comparisons betwen two registers

The branch instructions are restrictive wrt comparisons of two registers. If you want to compare for larger, smaller etc. use the SLT (Set on Less Than) and then the corresponding branch instruction.

Opcode	Operands	Comments
SLT	rd,rs,rt	# rd = 1 if rs < rt otherwise rd = 0
SLTU	rd,rs,rt	#idem but rs and rt are unsigned

Example: Branch to L1 if \$5 < \$6

```
SLT   $10,$5,$6    #$10 = 1 if r5 <r6
BNE   $10,$0,L1    # if so branch
```

Note: You can use pseudoinstructions such as:

```
BLT   $5,$6, L1    #Branch to L1 if if r5 <r6
```

This translates into 2 instructions:

```
SLT   $1,$5,$6    #$1 = 1 if r5 <r6
BNE   $1,$0,L1    # if so branch
```

(Note the use of register 1 by assembler)

Jump Instructions

Unconditional transfer of control

Opcode	Operands	Comments
JUMP	target	#go to specified target address
JR	rs	#jump to the address stored in rs

Jump and Link is used for procedure calls.

Opcode	Operands	Comments
JAL	target	#jump to target address; save PC in r31
JALR	rs,rd	#jump to the address stored in rs, rd = PC

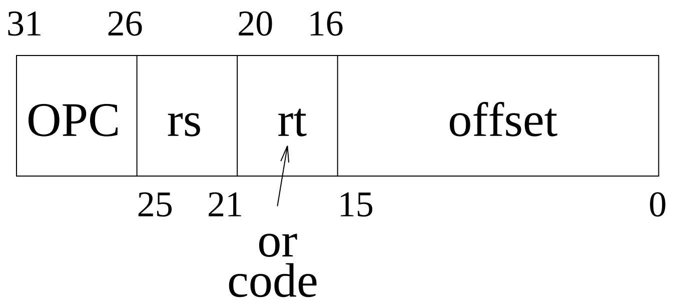
When calling a procedure use JAL; to return use JR $31
(or equivalently JR $ra)

Branch addressing

For branch addressing, you need the OPCODE, either two registers or one register and some encoding for $\leq, \geq$ ... that can easily fit into the field for one register (why?), and an offset.

You don't need a base register for the offset. The base register is the PC.

Opcode	Operands	Comments
XXX1	rs,rt,offset	#beq rs,rt, target
XXX2	rs,cond,offset	#bgez rs, offset



Example of Branch addressing

Opcode	Operands	Comments
BEQ	\$14,\$8,1000	#Branch to PC + 1000 (*4) if r14 = r8
BGEZ	\$14,1000	#Branch to PC + 1000 (*4) if r14 $\geq$ 0

31 26		20 16	
4	14	8	1000
	25 21		15 0

31 26		20 16	
1	14	1	1000
	25 21		15 0

Summary of Addressing modes

Each ISA specifies a number of addressing modes.

The MIPS has very few of them (RISC again), namely:

1. *register* mode. Operand is a register
2. *base* or *displacement* or *index* mode
   Operand is at address "register + 16-bit signed offset"
3. *immediate* mode. Operand is a constant in the instruction
4. *PC-relative* used for branching. As *base* but the register is PC and offset is in words.

Other interesting arithmetic and logical instructions
Multiply and Divide instructions

Multiplying two 32-bit numbers can yield a 64-bit number. Hence the use of HI and LO registers.

Dividing two numbers yields a quotient and a remainder. Hence the need also for two result registers.

Opcode	Operands	Comments
MULT	rs,rt	#HI/LO = rs * rt
MULTU	rs,rt	#HI/LO = rs * rt

HI gets upper 32 result bits, LO get lower 32 bits

Opcode	Operands	Comments
DIV	rs,rt	#divide rs by rt
DIVU	rs,rt	#divide rs by rt

LO gets quotient, HI gets remainder.
If an operand is negative, the remainder is not specified by the MIPS architecture.

Multiply and Divide instructions (Ct'd)

There are instructions to Move between HI/LO and registers.

Opcode	Operands	Comments
MFHI	rd	#Move from HI to rd
MFTI	rd	#Move to HI from rd

Note in SPIM:

Opcode	Operands	Comments
MUL	rd,rs,rt	#is a short cut for
MULT	rs,rt	
MFLO	rd	

Logic Instructions

Used to manipulate bits within words; set-up masks etc.

Opcode	Operands	Comments
AND	rd,rs,rt	#rd = AND(rs, rt)
ANDI	rt,rs,immed	#rt = AND(rs, immed)
OR	rd,rs,rt	#rd = OR(rs, rt)
ORI	rt,rs,immed	#rt = OR(rs, immed)
XOR	rd,rs,rt	#rd = XOR(rs, rt)
XORI	rt,rs,immed	#rt = XOR(rs, immed)
NOR	rd,rs,rt	#rd = NOR(rs, rt)

The immediate constant is limited to 16-bits. If we want to use a 32-bit constant, we need to use LUI.
There is also a pseudoinstruction

Opcode	Operands	Comments
NOT	rt,rs	#Bit-wise complement of rs in rt

Logic Instructions Examples

1. Create a *mask* of all 1's for the low-order byte of Register 6. Don't care about the other bits.

Opcode	Operands	Comments
ORI	\$6,\$6,0x00ff	#set r6<7:0> to 1's

2. Clear high-order byte of Register 6 but leave the 3 other bytes the same.

Opcode	Operands	Comments
LUI	\$5,0x00ff	#r5<31,24> all 0's,r5<23,16> are 1's, #rest are 0's
ORI	\$5,\$5,0xffff	#r5<15,0> all 1's
AND	\$6,\$6,\$5	#clear high-order byte

Shift instructions

Logical shifts: Zeros are inserted

Opcode	Operands	Comments
SLL	rd,rt,sh_amt	#rd = rt shifted left sh_amt bits
SRL	rd,rt,sh_amt	#rd = rt shifted right sh_amt bits

The maximum for sh_amt is 32.

Arithmetic shifts: The sign bit is extended from the other end. Useful only to the right.

Opcode	Operands	Comments
SRA	rd,rt,sh_amt	#rd = rt shifted right sh_amt bits
		#with high-order (sign) bit extended

Example: $5=1111 1111 0000 0000 0000 0000 0000 0000

Opcode	Operands	Comments
SRL	$5,$5,6	#shift right logical 6 bits
		# 0000 0011 1111 1100 0000
SRA	$5,$5,6	#shift right arithmetic 6 bits
		# 1111 1111 1111 1100 0000

An example

High-Level language version

```
int a[100];
int i;

for (i=0;i<100;i++){
   a[i] = 5;
}
```

Assembly-language version

Assume: start address of array a in r15.

We use r8 to store the value of i and r9 for the value 5

```
        add     $8,$0,$0        #initialize i
        li      $9,5            #r9 has the constant 5
Loop:   mul     $10,$8,4        #r10 has i in bytes
                                #could use a shift left by 2
        addu    $14,$10,$15     #address of a[i]
        sw      $9,0($14)       #store 5 in a[i]
        addiu   $8,$8,1         #increment i
        blt     $8,100,Loop     #branch if loop not finished
                                #taking lots of liberty here!
```

Machine-language version generated by a compiler

```
[0x00400020]    0x00004020  add $8, $0, $0        ; 1: add       $8,$0,$0
[0x00400024]    0x34090005  ori $9, $0, 5         ; 2: li        $9,5
[0x00400028]    0x34010004  ori $1, $0, 4         ; 3: mul       $10,$8,4
[0x0040002c]    0x01010018  mult $8, $1
[0x00400030]    0x00005012  mflo $10
[0x00400034]    0x014f7021  addu $14, $10, $15    ; 5: addu      $14,$10,$15
[0x00400038]    0xadc90000  sw $9, 0($14)         ; 6: sw        $9,0($14)
[0x0040003c]    0x25080001  addiu $8, $8, 1       ; 7: addiu     $8,$8,1
[0x00400040]    0x29010064  slti $1, $8, 100      ; 8: blt       $8,100,Loop
[0x00400044]    0x1420fff9  bne $1, $0, -28 [Loop-0x00400044]
```