What about negative nos.?

- Same binary representation
- Two’s complement
- 32-bit word
  - -2^{31} to $+2^{31} - 1$ (or -2,147,483,648 to + 2,147,483,647)

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: `if (i==j) h = i + j;`

```
    bne $s0, $s1, Label
    add $s3, $s0, $s1
Label:  ....
```

Control

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

```
if (i!=j)       beq $s4, $s5, Lab1
    h=i+j;      add $s3, $s4, $s5
else            j Lab2
    h=i-j;      Lab1:  sub $s3, $s4, $s5
                Lab2:  ...
```

- ***Can you build a simple* for *loop?***

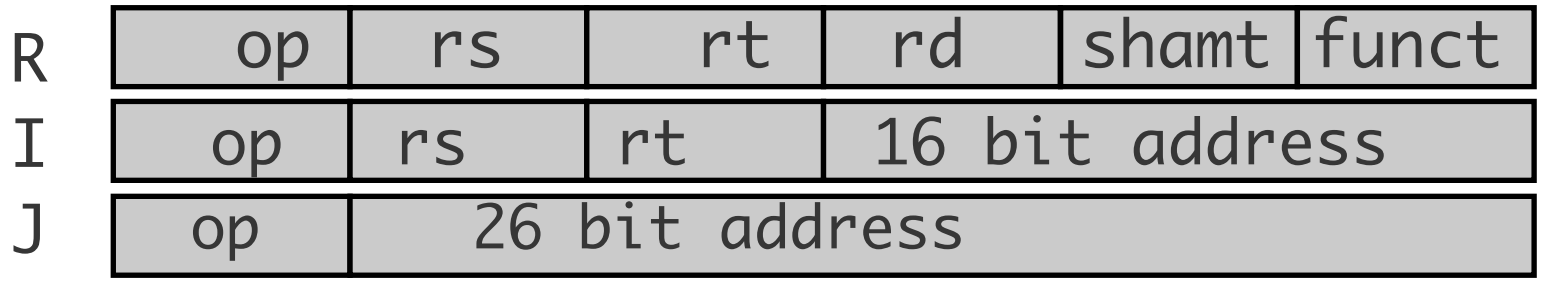
```
while (i != j)
     i +=1
```

So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3    $s1 = $s2 + $s3
sub $s1,$s2,$s3    $s1 = $s2 – $s3
lw $s1,100($s2)    $s1 = Memory[$s2+100]
sw $s1,100($s2)    Memory[$s2+100] = $s1
bne $s4,$s5,L      Jump to L if $s4 ≠ $s5
beq $s4,$s5,L      Jump to L if $s4 = $s5
j L                Next instr. is at L
```

- Formats:



Control Flow

- We have: `beq`, `bne`
  - what about `Branch-if-less-than`?
- New instruction:

```
slt $t0, $s1, $s2   →   if  $s1 < $s2 then
                            $t0 = 1
                        else
                            $t0 = 0
```

- Similarly, the constant version: `slti $t0, $s1, 10`
- Also, can compare with register `$z0`
- How to implement `blt` (branch–if–less–than)?

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., "`move $t0, $t1`" exists only in Assembly
  - would be implemented using "`add $t0,$t1,$zero`"
- When considering performance you should count real instructions

Supporting Functions (procedures)

What is needed?

- Functions: Analogy of a spy
  - secret plan, acquire resources, perform task, cover tracks, return with result
- Program has to
  - place params for function’s access
  - transfer control to procedure
  - acquire storage resources for the function
  - perform function’s instructions
  - place result for calling program’s access
  - return control to point of origin

Using registers

- Registers are fast!
  - `$a0 – $a3`: argument registers
  - `$v0-$v1`: value registers
  - `$ra`: return address register

Jump-Link and Program Counter

- Jump-and-link instruction
  - jumps to addr, store next instruction's addr in `$ra`: the *return address*
  - `jal ProcedureAddress`
- Program Counter (`PC`)
  - address of current instruction
  - ∴, `jal` stores `PC + 4` to setup procedure return
  - ∴, another instruction: `jr $ra`
    - jumps to address in `$ra`

Setup for Executing Functions

- Caller puts params in `$a0 – $a3`
- Uses `jal X` to jump to callee procedure `X`
- Callee performs its instructions
- Places results in `$v0 – $v1`
- Returns to caller by `$jr $ra`