9 | R & I Format Instructions

R Format Instructions

R (register format) Example: add $v0, $ra, $zero

- Instruction fields: 32 bits for an instruction
- opcode 6 bits
- src1 = rs 5 bits # can specify any of the 32 general registers
- src2 = rt 5 bits
- dest = rd 5 bits # destination register
- shamt 5 bits # usually shift instructions; if not, ignored
  # stands for shift amount
- function 6 bits # part of opcode, 6 up and 6 down

opcode	rs	rt	rd	shamt	function
31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0

In assembly language, it's add $rd, $rs, $rt # add regs
Notice how the order is different, as shown in the above table.

Let's do the same as the example above.
In the Appendix (.5 Core Instructions), we can see the add function. This will produce:

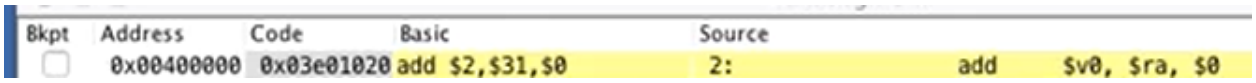
opcode	rs	rt	rd	shamt	function
00 0000	1 1111	0 0000	0 0010	? ????	10 0000

- Note: The value stated in the function in the appendix is 20. However, note that this is **20 in *hex***, not decimal! Convert 20 in hex to binary.

- $rt is by register 0 ($zero), which is register 0; thus 0's are filled in that box.
- $rs is by register $ra, which is register 31; thus, 31 in decimal → binary
- $rd is by register $v0, which is register 2; thus, 2 in decimal → binary
- shamt can be anything since this isn't a shift instruction, but we're going to put 0's when we code it

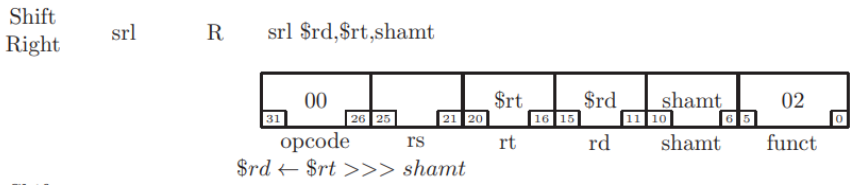
- Here's how we code it: **0000 0011 1110 0000 0001 0010 0000**
- This is with the proper spacing if we were to put all the binaries together
- To write this in hex, it is **$03e0\ 1020_{16}$**

How do we know that we did everything correctly? *Let's ask MARS and put the instruction right there in a .text section. Then, assemble the code. You can always check like this (in code).*



Let's try srl $t0, $a3, 19!
Note that the register numbers are translated from decimal to binary!

opcode	rs	rt	rd	shamt	function
00 0000	? ????	0 0111	0 1000	1 0011	00 0010



The code for this is: **0000 0000 0000 0111 0100 0100 1100 0010**. The hex is **$0007\ 44c2_{16}$**.

As for logical instructions (and, or, etc.), there is a pseudo-op: not ($rd ← ~$rt):

```
not   $v1, $t9        # $v1 = ~ $t9 is really
nor   $v1, $t9, $0    # ~ $t9 = $t9 nor 0, basically inverting all bits here
                      # it performs the or operation, then complements bits
                      # NOR Instruction (udel.edu)
```

Also, for the jr (jump register) instruction,

I Format (Immediate) Instructions

I (immediate format) Example: addi $a0, $a0, 1, including two bullets

- Memory reference Example: lw $a0, x

- Conditional branch Example: beq zero, done

- opcode 6 bits
- src = rs 5 bits # can specify any of the 32 general registers
- dest = rt 5 bits
- immediate 16 bits # usually just a constant

opcode	rs	rt	immediate
31 … 26	25 … 21	20 … 16	15 … 0

In assembly language, it's addi $rs, $rt, imm # add immediate
Notice how the order is the same, as shown in the above table.
This is why the largeness of immediate is limited to 16 bits in this instruction.

Let's do an example: addi $t3, $v0, -3
In the same Appendix, we can see the addi function. This will produce:

opcode	rs	rt	immediate
00 1000	0 0010	0 1011	1111 1111 1111 1101

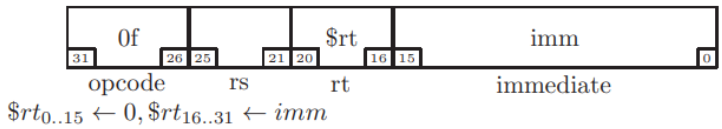
- The opcode is 08, thus we convert that to binary to get the number
- imm is the immediate put in; thus, convert -3 to binary (~~the signed 2 complement~~, or the 16 bit representation of -3; if you don't believe it, add 3 to it - you'll get 0)
- $rs is by register $v0, which is register 2; thus, 2 in decimal → binary
- $rt is by register $t3, which is register 11; thus 11 in decimal → binary

❖ The 32-bit instruction laid out: **0010 0000 0100 1011 1111 1111 1111 1101**
❖ The instruction in hexadecimal is **$204b\ fffd_{16}$**
❖ You can check this with MARS as well in the same way

Let's try lui $t8, 33! (This instruction loads the immediate in the upper 16 bits and clears the lower 16 bits of the register.)

opcode	rs	rt	immediate
00 1111	? ????	1 1000	0000 0000 0010 0001

lui I lui $rt, imm



The code for this is: **0011 1100 0001 1000 0000 0000 0010 0001**. The hex is **3c18 0021**$_{16}$.

It's important to note that memory reference instructions are I format.

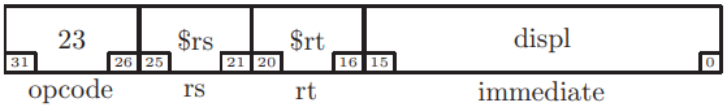
Explicit address

- The $rs field is the register containing a memory address.
- The immediate field is the displacement.
- Actual address = ($rs) + imm

Example: lw $v0, 16($a0)

opcode	rs	rt	immediate
10 0011	0 0100	0 0010	0000 0000 0001 0000

lw I lw $rt,displ($rs)



The code is: **1000 1100 1000 0010 0000 0000 0001 0000**. The hex is **8c82 0010**$_{16}$.

Also note, *if $a0 contains 0040 0004, the effective address would be 0040 0014*. The 16 would be added to the address, in which 16 (decimal) in hexadecimal is just 10.

Symbolic address

The assembler knows:

- The address of the .data area of memory (the #Address)
- For each symbol, its displacement from the beginning of .data area of memory

Example: .data # Address Displacement

```
three:  .word   3   # 0x1001000   0
five:   .word   5   # 0x1001004   4
nine:   .word   9   # 0x1001008   8
```

The assembler generates two instructions for this statement: lw $rt, label
disp is the displacement of the data label from the start of data memory.

```
lui   $at, 0x1001      # start of data area (the start of the addresses)
lw    $rt, disp($at)   # knows disp, $at contains beginning of memory, loads
                       # from that location into the single location
```

Example (using previous .data): lw $a1, nine

Two instructions generated:

```
lui   $at, 0x1001      # $at = 0x1001 0000
lw    $a1, 8($at)      # disp is 8 since three (0), five (4), nine (8)
```

Translate to machine language: *First, lui, then lw.*

opcode	rs	rt	immediate
00 1111	? ????	0 0001	0001 0000 0000 0001

The code is: **0011 1100 0000 0001 0001 0000 0000 0001**. The hex is **$3c01\ 1001_{16}$**.

opcode	rs	rt	immediate
10 0011	0 0001	0 0101	0000 0000 0000 1000

The code is: **1000 1100 0010 0101 0000 0000 0000 1000**. The hex is **$8c25\ 0008_{16}$**.

Conditional branch instructions

This includes branch if equal and branch if not equal. The other conditional branch instructions are pseudo-ops (no machine language instructions).

Conditional branching is done by branching to a *relative* address, which is relative to the *next* instruction. For example,

- Immediate field of 3 means branch to the instruction which is 3 instructions *after* the instruction *following* the branch.

- Immediate field of -4 means branch to the instruction which is 4 instructions *before* the instruction *following* the branch.

```
Example:    up:    li      $t0, 3
                   li      $t1, 4
                   beq     $t0, $t1, up        # branch -3 instructions
```

Here, we are below the beq instruction. To get to "up", we have to go up 3 instructions prior to the current place that we're at.

opcode	rs	rt	immediate
00 0100	0 1000	0 1001	1111 1111 1111 1101

The code is: **0001 0001 0000 1001 1111 1111 1111 1101**. The hex code is **1109 fffd**$_{16}$.

```
Example:           bne     $t0, $t1, down      # branch +1 instruction
                   li      $t0, $0
           down:
                   li      $t1, 0
```

Here, we are below the bne instruction. To get to "down", we have to move 1 instruction down.

opcode	rs	rt	immediate
00 0101	0 1000	0 1001	0000 0000 0000 0001

The code is: **0001 0101 0000 1001 1111 1111 1111 1101**. The hex code is **1509 0001**$_{16}$.

Other conditional branch instructions

This includes blt, ble, bgt, and bge. These are all pseudo-ops using set if less than: **slt** (R format).
Here's the format: **slt** **$rd, $rs, $rt** # $rd = 1 if $rs < $rt, $rd = 0 if $rs >= $rt

```
Example:    lp:    addi    $a0, $a0, -1
                   blt     $a0, $v0, lp
```

Assembler generates two instructions for blt (imagine these positionally in place of blt):

```
slt     $at, $a0, $v0       # sets $at register to 1; otherwise 0
bne     $at, $0, lp         # if $at is not 0, then $a0 < $v0, go to lp
                            # lp is relative -3!
```

Translate to machine language: *First, slt, then bne.*

opcode	rs	rt	rd	shamt	function
00 0000	0 0100	0 0010	0 0001	? ????	10 1010

The code is: **0000 0000 1000 0010 0000 1000 0010 1010**. The hex code is **0082 082a$_{16}$**.

opcode	rs	rt	immediate
00 0101	0 0001	0 0000	1111 1111 1111 1101

The code is: **0001 0100 0010 0000 1111 1111 1111 1101**. The hex code is **1420 fffd$_{16}$**.

The rest of the pseudo-ops:

Instruction	slt	branch
`blt rs, rt, dest`	`slt $at, rs, rt`	`bne $at, $0, dest`
`ble rs, rt, dest`	`slt $at, rt, rs`	`beq $at, $0, dest`
`bgt rs, rt, dest`	`slt $at, rt, rs`	`bne $at, $0, dest`
`bge rs, rt, dest`	`slt $at, rs, rt`	`beq $at, $0, dest`

Branch if less than or equal (ble) to have their $rt and $rs registers reversed.

- <= is the complement of greater than (use of beq)
- We're actually asking if $rs is strictly greater than $rt?
    - If it is, we'll get a 1 in the $at register (don't want to branch)
    - If we get a 0 in the $at register, we'll branch.

Branch if greater than (bgt) have $rt and $rs registers reversed. This is the same as before, except this time it uses the bne.

Branch if greater than or equal to (bge) are the same, but we use the beq here.

So basically, the steps are:

Show the fields in binary.

- Function in format appendix always

Group the bits into groups of 4.
Convert to hexadecimal to get a more concise version.