8 | Floating Points, Instruction Format, Binary Fields, and Pseudo Operations

Floating Point Instructions

The **floating point** data-type allows for non-integers (including very large/small integers, numbers that aren't integers, numbers that are very close to 0), corresponding to Java. We have (and their floating point instruction suffixes):

- float: single precision (32 bits) .s (1 word)
- double: double precision (64 bits) .d (2 words)

Examples: 2.06, 6.02e23 ($6.02 * 10^{23}$), 1e-50 ($1.0 * 10^{-50}$), -45.0

However, note that they aren't as precise as you might expect. When testing it in BlueJ, the errors can be seen with very long zeroes and numbers behind the decimal.

In MIPS architecture, there are **32 floating point registers** (not associated with the ones we've seen already); they are in coprocessor 1 (could be a separate area in the same chip), where each is 32 bit, named from $f0….$f31.

> For double precision, use even-odd register pairs, like $f0, $f1 is a register pair, where you refer to the register as its even number. (*Also, note that $f0 can store any value. It doesn't have to be 0.*)

Examples of single precision instructions:

- add.s $fd, $fs, $ft # $fd ← $fs + $ft
- sub.s $fd, $fs, $ft # $fd ← $fs - $ft
- mul.s $fd, $fs, $ft # $fd ← $fs * $ft
- div.s $fd, $fs, $ft # $fd ← $fs / $ft (get one result)
- mov.s $fd, $fs # $fd ← $fs

Examples of floating point data in memory:

```
        .data
avogadro:   .float      6.02e23
```

```
pi:         .double     3.14159
small:      .float      1.0e-33
```

Some essential instructions (to load from, and storing to, memory):

```
lwc1        $fd, loc    # symbolic or explicit address; $fd ← memory_loc
swc1        $fd, loc    # symbolic or explicit address; memory_loc ← $fd
```

To view the values of the floating points, you have to look at "Coproc 1" tab above the Registers in MARS.

How does conditional branching work with floating point numbers?

It's a two-step process:

1. **Compare** (floating point registers); also must precede the branch

```
c.eq.s      # compare for equality
c.lt.s      # compare for less than
c.le.s      # compare for less than or equal to
```

2. **Branch**, conditional on result of the compare (assuming it was executed just prior)

```
bc1t        # branch if the comparison was true
bc1f        # branch if the comparison was false
```

Example of conditional branching: Branch to done if $f0 == $f2, branch to lp if $f0 != $f2, branch to done if $f0 > $f2, branch to done if $f0 >= $f2 (< is the complement of >=)

```
c.eq.s      $f0, $f2    # does $f0 == $f2? compare for equality
bc1t        done

c.eq.s      $f0, $f2
bc1f        lp          # tests if it's false this time so it will move to lp

c.lt.s      $f2, $f0    # is $f2 < $f0? (note this equals $f0 > $f2)
bc1t        done

c.lt.s      $f0, $f2
bc1t        done
```

Machine Language

Machine language is the binary 'language' of coded instructions where the processor is capable of executing it directly. It can be produced by:

- An assembler (software that translates from assembly language to machine language)
- A compiler (software which translates from high level language to machine language)
- A person (not normally done)

Instruction formats are:

- R (register format) Example: add $v0, $a0, $a1
  - 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 above table

  - R Format Example: add $v0, $ra, $zero

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

    - opcode = 0_{16}
    - function = 20_{16} (distinguishes add function from for example, subtract)
    - rs = 31_{16} (31st register, aka $ra register)
    - rt = 0_{16} (0th register, for $0)

        - rd = 2_{16} (2nd register, for $v0)
        - shamt = it doesn't matter what's in here, ignores since it's not a shift instruction, so the assembler will always put 0s here
        - Thus, this instruction is: 0000 0011 1110 0000 0001 0000 0010 0000

- 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 above table
    - This is why the largeness of immediate is limited to 16 bits in this instruction

- J (jump format) Example: j lp

    - opcode 6 bits
    - jump (word) address 26 bits

opcode	**jump (word) address**
31 … 26	25 … 0

    - A bit harder to understand since a word address is 32 bits
    - In assembly language, it's j label # (unconditional) jump
    - We'll see how assembler takes the label and translates it into a 26-bit address

To find out how many different instructions (different opcodes) can be found, count how many bits there are in the opcode section and use that as n; then solve 2^n.

Similarly can be done for registers. If register fields rs, rt, and rd are always *n* bits in length, you can apply 2^n to get *x* many different values, meaning these fields can specify any one of the *x* registers.

Binary Fields

Let's represent binary fields in hexadecimal, for convenience. Example of an 8-bit field is 1011 0111 = $b7_{16}$ = 0xb7. (Note how we separate them in groups of 4.) But what if the length of the field is not a multiple of 4?

Example of a 5-bit field: 1 0101 = 15_{16} = 0x15.

> Here, we separate the rightmost (low-order) four first, leaving the other on its own. Notice how 5_{16} represents 4 bits, 1_{16} represents 1 bit (it can only be 0 or 1).

Example of a 10-bit field: 10 1111 0110 = $2f6_{16}$ = 0x2f6

> Possible values for 2_{16} in the hexadecimal answer are 0, 1, 2, or 3. Notice how the remaining groups are 4 bits.

Example of a 7-bit field, given 62_{16} = 0x62: 110 0010

> Show the low-order first, which is the 2_{16}, which is the 16. AKA, we'll always group from the low-order end.

Pseudo Operations

MIPS assembler allows for operations which are not MIPS instructions, for the programmer's convenience and clarity. For example,

Load immediate: Assembler substitutes an addi instruction

```
li      $t0, 3          # put 3 into $t0
addi    $t0, $0, 3      # $t0 = 0 + 3
```

Move: Assembler substitutes an add instruction

```
move    $t0, $a0        # $t0 = $a0
add     $t0, $0, $a0    # $t0 = 0 + $a0
```

Not: Assembler substitutes a nor (not or) instruction

```
not     $t0, $t1        # $t0 = ~ $t1 (not $t1)
nor     $t0, $t1, $0    # $t0 = ~ ($t1 ∨ 0)
                        # ($t1 ∨ 0) gives you what you gave, $t1
```