Instruction Encoding

Introduction

- Remember that in a stored program computer, instructions are stored in memory (just like data)
- Each instruction is fetched (according to the address specified in the PC), decoded, and exectuted by the CPU
- The ISA defines the format of an instruction (syntax) and its meaning (semantics)
- An ISA will define a number of different instruction formats.
- Each format has different fields
- The OPCODE field says what the instruction does (e.g. ADD)
- The OPERAND field(s) say where to find inputs and outputs of the instruction.

MIPS Encoding

- The nice thing about MIPS (and other RISC machines) is that it has very few instruction formats (basically just 3)
- All instructions are the same size (32 bits = 1 word)
- The formats are consistent with each other (i.e. the OPCODE field is always in the same place, etc.)
- The three formats:
  - I-type (immediate)
  - R-type (register)
  - J-type (jump)

I-type (immediate) Format

- An immediate instruction has the form:

```
XXXI   rt, rs, immed
```

- Recall that we have 32 registers, so we need ??? bits each to specify the rt and rs registers
- We allow 6 bits for the opcode (this implies a maximum of ??? opcodes, but there are actually more, see later)
- This leaves 16 bits for the immediate field.

31	25	20	15
OPC	rs	rt	immed
26	21	16	0

I-type Example

- Example:

```
ADDI   $a0, $12, 33      # a0 <- r12 + 33
```

- The ADDI opcode is 8, register a0 is register # 4.

31	25	20	15
8	12	4	33
26	21	16	0

- What would this be in binary? In hex?

Load-Store Formats

- A memory address is 32 bits, so it cannot be directly encoded in an instruction.
- Recall the use of a base register + offset (16-bits) in the load-store instructions.
- Thus, we need an OPCODE, a destination/source register (destination for load, source for store), a base register, and an offset.
- This sounds very similar to the I-type format... example:

```
LW  $14, 8($sp)    # r14 is loaded from stack+8
```

- The LW opcode is 35 (0x23)

31	25	20	15
35	29	14	8
26	21	16	0

R-type (register) format

- General form:

```
XXX    rd, rt, rs
```

- Arithmetic-logical and comparison instructions require the encoding of 3 registers, the rest can be used to specify the OPCODE.
- To keep the format as regular as possible, the OPCODE has a primary "opcode" and a "function" field.
- We also need 5 bits for the shift-amount, in case of SHIFT instructions.
- The 16 bits used for the immediate field in the I-type instruction are therefore split into 5 bits for rd, 5 bits for shift-amount, and 6 bits for function (the other fields are the same).

31	25	20	15	10	5
OPC	rs	rt	rd	sht	funct
26	21	16	11	6	0

R-type Example

```
SUB     $7, $8, $9      # r7 <- r8 - r9
```

- The opcode for all R-type instructions is zero, the function code for SUB is 34, the shift amount is zero.

31	25	20	15	10	5
0	8	9	7	0	34
26	21	16	11	6	0

- What is this in binary/hex?

J-type (Jump) Format

- For a jump, we only need to specify the opcode, and we can use the other bits for an address:

31	25
OPC	address
26	0

- We only have 26 bits for the address, but MIPS addresses are 32 bits long...
- Because the address must reference an instruction, which is a word address, we can shift the address left by 2 bits (giving us 28 bits). We get the other 4 bits by combining with the 4 high-order bits of the PC.

Branch Addressing

- There are 2 kinds of branches:
  1. EQ/NEQ family (compares 2 regs for (in)equality), example:
     `BEQ     $14, $8, 1000`
  2. Compare-to-zero family (compares 1 reg to zero), example:
     `BGEZ    $14, 1000`
- Both "families" require OPCODE, rs register, and offset
- (1.) requires an additional register (rt)
- (2.) requires some encoding for (>=, <=, <, >)

31	25	20	15
OPC	rs	rt	offset/4
26	21	16	0

or code (for >, <, etc)

- Note that we divide the offset by 4. Why?

Branch example

```
BEQ  $14, $8, 1000 # PC := PC+1000 if r14==r8

BGEZ $14, 20       # PC := PC+20 if r14 >= 0
```

- The opcode for BEQ is 4; for BGEZ is 1, the code for >= is 1

31	25	20	15
4	14	8	250
26	21	16	0

31	25	20	15
1	14	1	5
26	21	16	0

Assembly Language Version

- Recall our running example:

```
        .data               # begin data segment
array:  .space 400          # allocate 400 bytes

        .text               # begin code segment
        .globl main         # entry point must be global

main:   move $t0, $0        # $t0 is used as counter
        la   $t1, array     # $t1 is pointer into array
start:  bge  $t0, 100, exit# more than 99 iterations?
        sw   $t0, 0($t1)    # store zero into array
        addi $t0, $t0, 1    # increment counter
        addi $t1, $t1, 4    # increment pointer into array
        j    start          # goto top of loop
exit:   j    $ra            # return to caller of main...
```


Machine Language Version

```
Encoded:    Machine Ins:        Source Ins:
----------  ---------------     ----------------------
0x00004021  addu $8, $0, $0   ; 9: move$t0, $0
0x3c091001  lui $9, 4097      ; 10: la$t1, array
0x29010064  slti $1, $8, 100  ; 11: bge$t0, 100, exit
0x10200005  beq $1, $0, 20
0xad280000  sw $8, 0($9)      ; 12: sw$t0, 0($t1)
0x21080001  addi $8, $8, 1    ; 13: addi$t0, $t0, 1
0x21290004  addi $9, $9, 4    ; 14: addi$t1, $t1, 4
0x0810000b  j 0x0040002c      ; 15: jstart
0x03e00008  jr $31            ; 16: j$ra
```