Instruction Encoding

Introduction

- An ISA defines a particular encoding (syntax) for each instruction it defines.
- It also defines the meaning (semantics) of that instruction.
- Typically, an ISA will define a number of different formats.
- Each format has different fields:
  - OPCODE says what the instruction does
  - OPERAND (fields) say where to find the inputs to the instruction

MIPS Encoding

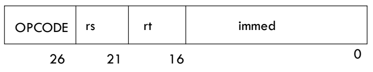
- The nice thing about MIPS (and other RISC ISAs) is that it has very few formats (basically just 3).
- All instructions are the same size (1 word = 32 bits)
- The 3 formats:
  - I-type (2 registers and an immediate value)
  - R-type (3 registers)
  - J-type (used only for jumps)

I-type Format

- An immediate instruction has the form:

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

- Recall that we have 32 registers in MIPS, 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).
- This leaves 16 bits for the immediate field:

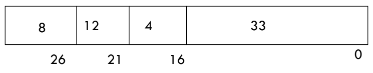


I-type Example

- Example:

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

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

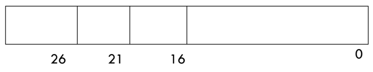


- What is this in binary? In hex?

Load/Store Instructions

- Recall that addresses are 32 bits. For this reason they can't be encoded directly in the instruction.
- Load/Store instructions take a register (containing an address) and an immediate offset.
- Example:

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

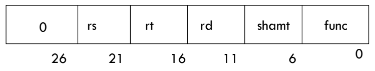


R-Type Instructions

- General form:

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

- All arithmetic/logical/comparison instructions require 3 regs.
- To keep the format regular, the OPCODE is always zero, and the real function is encoded in another 6 bit field.
- A 5-bit shift amount is also encoded in this format.

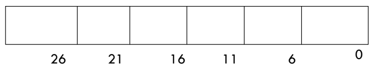


R-type Example

- Example:

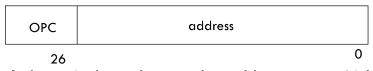
```
SUB        $7, $8, $9
```

- The opcode zero, the function code for SUB is 34.



J-type Format

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



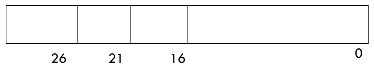
- We only have 26 bits of space, but addresses are 32 bits...
- The address must reference an instruction, so we drop the low two bits. We get the other 4 bits by combining with the PC.
- (We can jump pretty far...)

Branch Addressing

- BEQ/BNE are encoded (almost) like any old I-type instruction:

```
XXX      rs, rt, offset
BEQ     $14, $8, 1000
```

  - The opcode for BEQ is 4



Full Example

- Recall our loop example:

```
.data
array:     .space   400
.text
main:      add      $t0, $0, $0      # use t0 as a counter (i)
           addi     $t1, $gp, array  # t1 holds an address
           addi     $t2, $0, 100     # t2 holds constant 100
start:     slt      $t3, $t0, $t2
           beq      $t3, $0, done
           sw       $t0, 0($t1)      # a[i] = i
           addi     $t0, $t0, 1      # i = i + 1
           addi     $t1, $t1, 4      # why are we adding 4?
           j        start
done:      jr       $ra              # return to caller
```

Encoded:

- Here is the encoded version:

```
Address:       Machine code    Disassembly
00000000       0x00004020:     add     $t0, $0, $0
00000004       0x23890000:     addi    $t1, $gp, 0
00000008       0x200a0064:     addi    $t2, $0, 100
0000000c       0x010a582a:     slt     $t3, $t0, $t2
00000010       0x100b0004:     beq     $t3, $0, 4
00000014       0xad280000:     sw      $t0, $t1, 0
00000018       0x21080001:     addi    $t0, $t0, 1
0000001c       0x21290004:     addi    $t1, $t1, 4
00000020       0x08000003:     j       0x3
00000024       0x03e00008:     jr      $0, $ra, $0
```