Unconditional Branching: Jump

Branching

Already saw conditional branch statements

Sometimes, we want to move to another section of code regardless of any condition

Can you think of an example?

Branching

Already saw conditional branch statements

Sometimes, we want to move to another section of code regardless of any condition

Can you think of an example?

- Function calls
- Also helpful for loops

Branching

What about the following?

```
beq $0, $0, label
```

Branching

What about the following?

```
beq $0, $0, label
```

Works correctly, but has limited range – branch is I-type instruction with 16-bit immediate

By leaving out registers, we can save more room for address

Jumps

Unconditional branch is called *jump*

MIPS Assembly Code

```
  addi  $s0, $0, 4      # $s0 = 4
  addi  $s1, $0, 1      # $s1 = 1
  j     target          # jump to target
  addi  $s1, $s1, 1     # not executed
  sub   $s1, $s1, $s0   # not executed

target:
  add   $s1, $s1, $s0   # $s1 = 1 + 4 = 5
```

Jumps

That’s about it – jump is very simple instruction

Note that jump instruction uses labels, just like branch instructions

However, way jump represents labels is different

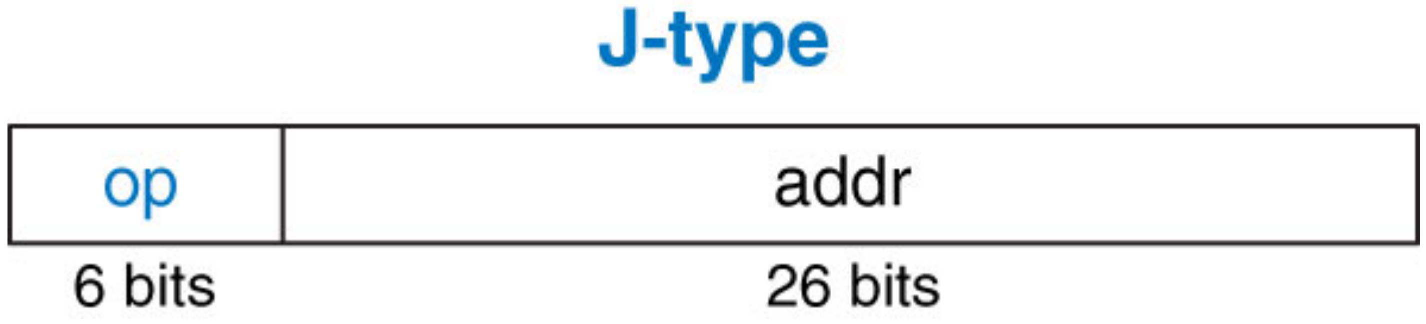
In fact, jump gets its own special instruction type

J-Type Instructions

Third (and final!) MIPS instruction format

Includes:

- 6-bit opcode (same as all instructions)
- Single 26-bit address operand



J-Type Instructions

Quick note – we will see two J-type instructions:

- `j` (jump)
- `jal` (jump and link)

Don’t worry about difference for now. `jal` will be helpful for writing functions in assembly. Until you know you need it, just use `j`

Pseudo-Direct Addressing

Once again, we cannot store full 32-bit address in instruction

Assume top 4 bits come from PC+4

Append 26 bits we have

Shift result by 2 bits (multiply by 4)

Result is the address of next instruction

MIPS Assembly Code

```
0x0040005C        jal    sum
...
0x004000A0  sum:  add    $v0, $a0, $a1
```

Assembly Code

`jal sum`

Field Values

op	addr
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

JTA	0000 0000 0100 0000 0000 0000 1010 0000	(0x004000A0)
26-bit addr	0000 0000 0100 0000 0000 0000 1010 0000	(0x0100028)
	0 1 0 0 0 2 8	

Pseudo-Direct Addressing

Seems odd at first, but quite straightforward once we understand the two "tricks":

- Multiplying by 4
- Taking first 4 bits from `PC + 4`

Remember: reason for all this trouble is that we cannot store all 32 bits of address

Pseudo-Direct Addressing

Multiplying by 4

- Instructions are word-aligned – every instruction starts at multiple of 4

Taking first 4 bits from `PC + 4`

- Need 4 bits from somewhere – assume we are not jumping extremely far away

Pseudo-Direct Addressing

"Pseudo"-direct because we cannot fit full 32-bit address into 28 bits

Turns out not to matter – see later that all MIPS code fits within subset of 32-bit address space

Remainder of 32-bit address space reserved for data (variables, arrays, etc.)

Pseudo-Direct Addressing

MIPS Assembly Code

```
0x0040005C         jal      sum
. . .
0x004000A0   sum:  add      $v0, $a0, $a1
```

Essentially, we are just storing all bits of address that matter

Assembly Code

`jal sum`

Field Values

op	addr
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

One more jump

There is third jump instruction in addition to `j` (jump) and `jal` (jump and link)

Final jump instruction is: `jr` (jump register)

Register holds 32-bit value, so no fancy addressing needed – just jump to location specified in register

Jump register

MIPS Assembly Code

```
0x00002000  addi  $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004  jr    $s0              # jump to 0x00002010
0x00002008  addi  $s1, $0, 1       # not executed
0x0000200c  sra   $s1, $s1, 2      # not executed
0x00002010  lw    $s3,  44($s1)    # executed after jr instruction
```

Jump register

You should have noticed something odd – J-type instructions do not have register as argument

Even though it is functionally a jump, `jr` is an R-type instruction

Remember to separate what an instruction does from how it is stored

Jump register

Like `jal`, `jr` is useful when writing functions

In fact, the two work together to ensure we can move to and from functions correctly

For now, just remember to end all of your functions with

```
jr $ra
```