Conditional Branching

Branching

Assembly language lacks traditional control flow statements.

- if/else
- while loops
- for loops

These are implemented using branches, jumps, and labels

MIPS Assembly Code

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target    # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1         # not executed
  sub   $s1, $s1, $s0       # not executed

target:
  add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Label

Label is target of branch instruction

If branch is taken, control moves to instruction immediately following label

Otherwise, label has no effect whatsoever

Label

Always keep in mind that labels do not provide scope

A label is not a function

You can branch *to* a label, but the existence of a label will not cause following section of code to be skipped if branch not taken

Assembly **always goes to next instruction unless branch is taken**

What is the value of $s0 at the end of execution for each assembly program?

```
1     addi $s0, $0, 10
2     addi $s1, $0, 10
3     beq $s0, $s1, equality
4     addi $s0, $s0, 100
5   equality:
6     addi $s0, $s0, 1000
7     addi $s1, $s1, 10
8
```

```
1     addi $s0, $0, 0
2     addi $s1, $0, 10
3     beq $s0, $s1, equality
4     addi $s0, $s0, 100
5   equality:
6     addi $s0, $s0, 1000
7     addi $s1, $s1, 10
8
```

Conditional Branching

MIPS has two conditional branches:

- beq
- bne

MIPS Assembly Code

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target    # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1         # not executed
  sub   $s1, $s1, $s0       # not executed

target:
  add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

MIPS Assembly Code

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  s11   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne   $s0, $s1, target    # $s0 == $s1, so branch is not taken
  addi  $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub   $s1, $s1, $s0       # $s1 = 5 – 4 = 1

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

Conditional Branching

Other branch instructions exist, mainly ones that compare register value to 0

MARS will also suggest **pseudoinstructions** such as ”branch if less than” that are assembled into other, actual instructions

For this course, use only beq and bne. You can combine them with slt to accomplish anything you will need

Instruction format

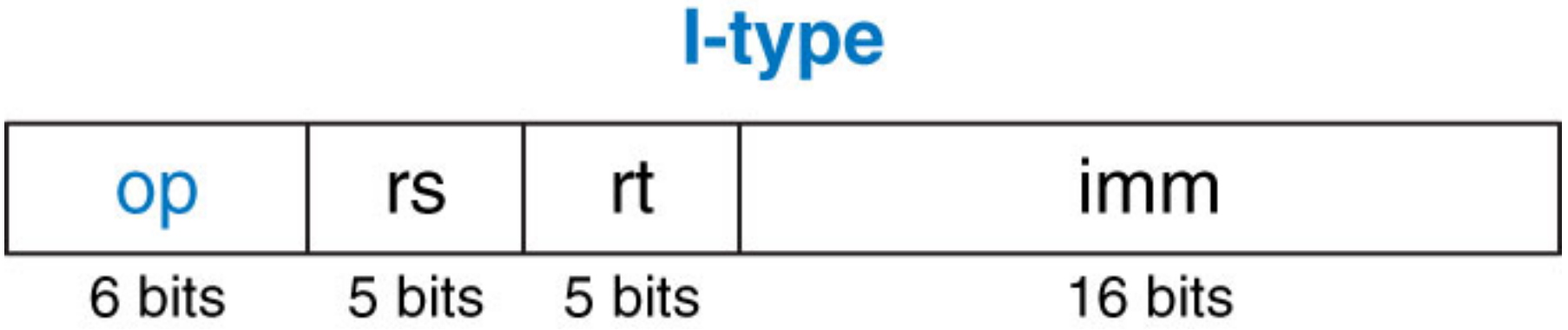
Remember that every instruction in MIPS is represented as a 32-bit number in some format

Opcode and registers are easy enough, but how do we represent label?

Recall: I-Type Instructions

Two register operands and an immediate

Branches are I-type instructions



Instruction format

Label represented as immediate

Recall that each instruction has address in memory

Label is human-readable way of determining which address to branch to, stored as number in computer

Because addresses are 32 bits and immediates are 16 bits, cannot store entire address in immediate field

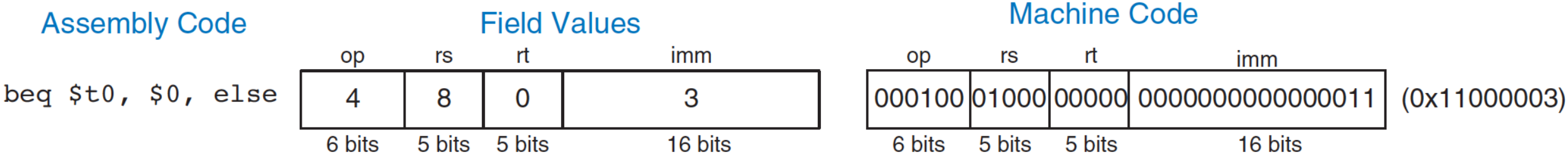
PC-Relative Addressing

MIPS Assembly Code

```
0xA4            beq  $t0, $0, else
0xA8            addi $v0, $0, 1
0xAC            addi $sp, $sp, 8
0xB0            jr   $ra
0xB4      else: addi $a0, $a0, -1
0xB8            jal  factorial
```

Conditional branch instructions use this strategy to specify the value of PC if a branch is taken

Immediate value is *how many instructions beyond the next we should go if branch is taken*



PC-Relative Addressing

Why choose number of instructions ”beyond next”?

Normal behavior is to move to next instruction, so we are recording how far to go beyond what we normally would

PC-Relative Addressing

Why relative to current program counter at all?

1. Often branching to nearby locations in code (think loops and if statements)
2. Need some way to get 32-bit number from 16-bit number

Immediate Value

Some I-type instructions treat immediate as signed, others do not

In case of branch, immediate is *signed*

Permitting negative branch target allows us to jump backwards in code – not allowing this would be very limiting

Immediate Value

Immediate value is 16-bit two's complement number

Value represents number of instructions (not bytes!) to skip if branch taken

Fixed size puts limit on how far branch can go. What is farthest number of instructions forward branch can send us?

Branches

Understanding how branch statements are stored helps us understand some limitations and will be critical to implementing in microarchitecture

When programming in assembly, no need to think about PC-relative addressing – just consider initial discussion of what branch does