ECE260: Fundamentals of Computer Engineering

MIPS Branching Instructions

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

Conditional Operations & Simple Jumps

- Not all programs involve a simple set of sequential instructions
  - MOST programs have conditionally executed code
    - If some condition is true, execute a specific block of code, otherwise execute some other code
    - If-statements / If-else statements / If-elseif-statements
- MIPS has ***branch*** and ***jump*** instructions to handle conditional code
  - Branch to a labeled instruction if a condition is true, otherwise, continue sequentially

Conditional Operations & Simple Jumps

- Branch-on-equal (beq)
  - I-type instruction with 16-bit immediate that stores distance (in words) to label
  - if (rs == rt) branch to label L1

```
beq rs, rt, L1
```

- Branch-on-not-equal (bne)
  - I-type instruction with 16-bit immediate that stores distance (in words) to label
  - if (rs != rt) branch to label L1

```
bne rs, rt, L1
```

- Jump (j)
  - J-type instruction with 26-bit immediate for address of label
  - unconditional jump to label L1

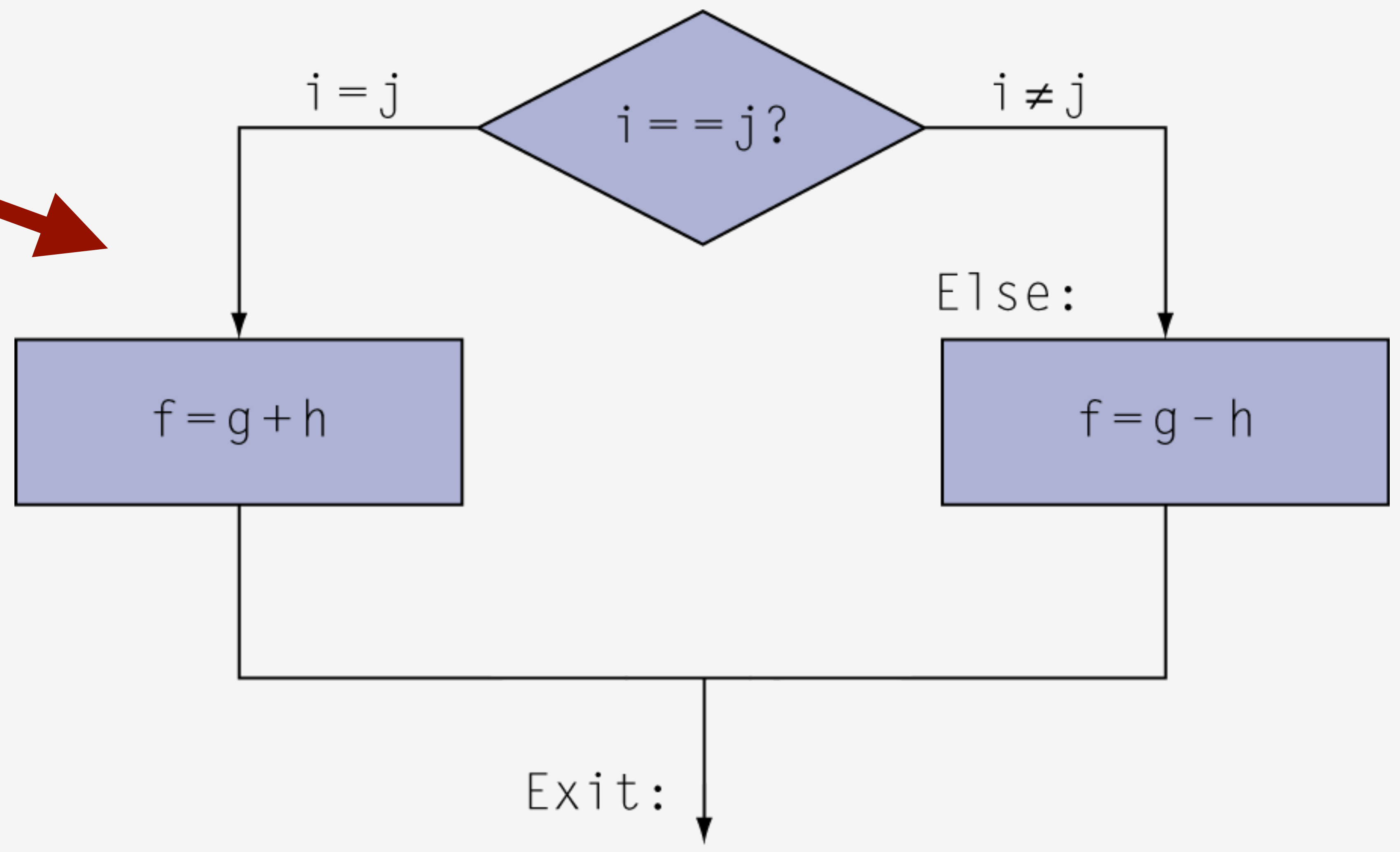
```
j L1
```

Compiling If Statements

- Example C code

```
if (i == j)
  f = g + h;
else
  f = g – h;
```

Flow Control of C code



- Oftentimes, the above code structure will result in a ***branch*** and a ***jump*** instruction

```
      bne $s3, $s4, Else  # check i==j, branch if they differ
      add $s0, $s1, $s2   # if i==j, do (g + h)
      j   Exit            # then jump to the end
Else: sub $s0, $s1, $s2   # if i!=j, do (g – h)
Exit:
```

Assume:
i in $s3
j in $s4
g in $s1
h in $s2

Compiling Loop Statements

- Example C code

```
while (save[i] == k) {
    i += 1;
}
```

Assume:
i in $s3
k in $s5
save baseAddr in $s6

- Looping structures continue to execute until some condition is met

```
Loop: sll  $t1, $s3, 2     # convert i to byte-addressing (i.e. i*4)
      add  $t1, $t1, $s6   # add offset of index i to baseAddr of save
      lw   $t0, 0($t1)     # load save[i]
      bne  $t0, $s5, Exit  # check save[i]==k, branch if they differ
      addi $s3, $s3, 1     # increment i
      j    Loop            # jump back to beginning of loop
Exit:
```

More Conditional Operations

- Set result to 1 if a condition is true, otherwise, set to 0
- Set-on-less-than (slt)
  - if (rs < rt) rd = 1; else rd = 0;

```
slt rd, rs, rt
```

- Set-on-less-than-immediate (slti)
  - if (rs < constant) rt = 1; else rt = 0;

```
slti rt, rs, constant
```

- Set-on-less-than-immediate-unsigned (sltiu)
  - if (rs < constant) rt = 1; else rt = 0;

```
sltiu rt, rs, constant
```

- Use slt instructions in combination with beq, and bne instructions

```
slt $t0, $s1, $s2    # if ($s1 < $s2)
bne $t0, $zero, L    #   branch to L
```

Branch Instruction Design

- Only beq and bne branch instructions
  - No native instructions for branch-on-less than (blt), branch-on-greater-or-equal (bge), etc.
- Hardware for <, >, ≤, ≥, is slower than hardware for =, ≠
  - Combining comparison with branch involves more work per instruction
    - At hardware level, additional levels of hardware logic result in slower hardware
    - Would require a slower clock
  - All instructions would be penalized and run at slower clock
- beq and bne are the common case, so they are implemented in hardware
- Pseudo-instructions exist for blt, bgt, ble, bge
  - Converted into some combination of slt, bne, beq

Signed vs. Unsigned Comparisons

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
  - Assume:

    $s0 = 1111 1111 1111 1111 1111 1111 1111 1111   (-1 or +4,294,967,295)

    $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
    - Signed comparison

```
slt  $t0, $s0, $s1  # –1 < +1 therefore, sets $t0=1
```

    - Unsigned comparison

```
sltu $t0, $s0, $s1  # +4,294,967,295 > +1 therefore, sets $t0=0
```