Slide 1

Administrivia

- (None?)

Slide 2

"Shift" Instructions

- C << and >> (on unsigned numbers) are translated into `sll` ("shift left logical") and `srl` ("shift right logical").
- `sll` and `srl` do what the names imply(?): Bits "fall off" one side, and we add zeros at the other side.
- When shifting left, filling with zeros makes sense. But when shifting right, might want to extend the sign bit instead. `sra` ("shift right arithmetic") does that.
- All R-format instructions, and they use that "shift amount" field (others don't).
- These instructions very useful for multiplying and dividing by small powers of 2, important since multiplication and division likely to be slow (more later in the course).

Slide 3

Logical Operations

- Sometimes useful to be able to work with individual bits — e.g., to implement a compact array of boolean values.
- Thus, MIPS instruction set provides "logical operations". Hard to say whether these exist to support C bit-manipulation operations, or C bit-manipulation operations exist because most ISAs provide such instructions!

Slide 4

Bitwise And and Or

- C `&` is translated into `and` or `andi`. C `|` is translated into `or` or `ori`. Format/operands are analogous to `add` and `addi`. (Note however that while the immediate value in `add` is sign-extended, the one for `andi` is not.)

  (Note/recall that C has two sets of and/or operators — logical and bitwise. These are the bitwise ones.)
- (Example on next slide.)
- We could use these to set, clear, or test particular bits (`or` to set, `and` to clear, `and` with a 1 in the position to test and then a check of result).
- All R-format or I-format instructions.

Slide 5

Example of Bitwise And, Or

- Given the following values for `$s1`, `$s2`

```
0000 0001 0100 1010 0110 1001 0111 1010 1000
1010 0011 0101 0000 1001 1001 1111 0000 0101
```

  result of applying `and` is

```
0000 0001 0100 0000 0000 1001 0111 0000 0000
```

  and result of applying `or` is

```
0000 0001 0100 1010 0110 1001 0111 1010 1000
1010 0011 0101 1010 1111 1001 1111 1010 1101
```

Slide 6

Other Logical Operations

- “Exclusive or” implements . . . what the name suggests (see textbook).
- “Nor” likewise. Can be used to implement “not” (see textbook).

Slide 7

Flow of Control

- So far we know how to do (some) arithmetic, move data into and out of memory. What about if/then/else, loops? (See sidebar on p. 96 for early commentary on conditional execution.)
- Need instructions that allow us to "make a decision". Perhaps surprisingly, only two: `beq` ("branch if equal"), `bne` ("branch if not equal").
- Format:

  `beq reg1, reg2, label`

  where `label` is a "label" (text followed by `:` in source, either on the same line as the instruction to branch to or on a line by itself just before)

  and similarly for `bne`.
- Illustrate with an example . . .

Slide 8

Sidebar: `goto`

- Some very early HLLs implemented conditional execution using `goto`, also spelled `go to`.

  What it does: Immediately transfer control to some other point in the program, identified by a label (e.g., `here:`).
- Conditional execution and loops can all be expressed using `goto`. Makes some sense, since this is pretty much all the hardware can do.
- Very quickly became apparent that this made for code that was hard to reason about. So later languages have been "block structured".

Slide 9

Sidebar: goto in C, Continued

- goto still exists in C because every once in a while it makes for more-readable code (e.g., some error handling).
- Useful in this course as an intermediate step between block-structured (“normal”?) C and assembly language, which has no notion of block structuring.
- (Sometimes written goto. Same thing.)

Slide 10

Flow of Control Example

- Suppose we have this in C (and as usual all variables are 32-bit integers)

```
          if (i == j) goto L1:
          f = g + h;
L1:       f = f - i;
```

- What instructions should compiler produce? Assume we’re using $s0 through $s4 for f, g, h, i, j.
- (For now, punt on how to represent L1.)

Slide 11

Flow of Control Example, Continued

- Compiling

```
         if (i == j) goto L1:
         f = g + h;
L1:      f = f - i;
```

using `$s0` through `$s4` for `f`, `g`, `h`, `i`, `j`.

gives

```
         beq     $s3, $s4, L1
         add     $s0, $s1, $s2
L1:      sub     $s0, $s0, $s3
```

Slide 12

Another Flow of Control Example

- Of course, we don't usually have `goto` in C. More likely is this:

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

- What to do with this? Rewrite using `goto` ...

Slide 13

Another Flow of Control Example

- Rewriting

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

gives

```
         if (i != j) goto Else:
         f = g + h
         goto End:
Else:    f = g - h
End:     ....
```

and then we can continue as before. (How to do unconditional "goto"? `j` ("jump").)

Slide 14

Loops

- Do we have enough to do (some kinds of) loops? Yes — example:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop:
```

assuming we're using `$s1` through `$s4` for `g`, `h`, `i`, `j`, and `$s5` for the address of `A`.

(This time we'll use `sll` rather than two `add`s to multiply `i` by 4.)

Slide 15

Loops — Example Continued

- Result

```
Loop:   sll     $t1, $s3, 2      # $t1 <- 4*i
        add     $t1, $t1, $s5    # $t1 <- & of A[i]
        lw      $t0, 0($t1)      # $t0 <- A[i]
        add     $s1, $s1, $t0    # g = h + A[i]
        add     $s3, $s3, $s4    # i = i + j
        bne     $s3, $s2, Loop   # if (i!=h) goto Loop
```

Slide 16

Conditional Execution, Continued

- If hand-compiling from C, useful to first translate into code with only `goto` for out-of-sequence execution, and from there to MIPS.
- Example:

```
while (A[i] == k) {
          i = i + j;
}
```

Slide 17

Example Continued

- MIPS equivalent, with C-with-`goto` as comments (and assuming `$s0` has the address of `A` and registers `$s1` through `$s3` have `i`, `j`, and `k`):

```
Loop:
# if (A[i] != k) goto End:
        sll     $t0, $s1, 2     # i * 4
        add     $t0, $s0, $t1   # &A[i]
        lw      $t0, 0($t1)     # A[i]
        bne     $t0, $s3, End
#   i = i + j
        add     $s1, $s1, $s2
#   goto Loop:
        j       Loop
End:
```

Slide 18

More Flow of Control

- With what we have now we can do if/then/else and loops, but only if condition being tested is equals / not equals.
- So, we need instructions such as `blt`, `ble`, right?
- But those are apparently difficult to implement well; instead MIPS has "set on less than":

```
slt   r1, r2, r3
```

which compares the contents of registers `r2` and `r3` and sets `r1` — 1 if `r2` is smaller, else 0.

- Example — compile the following C:

```
        if (a < b) goto Less:
```

assuming we're using `$s0`, `$s1` for `a`, `b`.

Slide 19

Example Continued

- Equivalent MIPS:

```
slt     $t0, $s0, $s1
bne     $t0, $zero, Less
```

Slide 20

More Flow of Control, Continued

- Do we have enough now? for all six possible C comparisons of integers? Yes . . .
- One more C flow-of-control construct we could talk about — `switch` — but defer for now.
- Machine language for all of these instructions? Later.

Minute Essay

- Is the "shift amount" field big enough to represent all possible shifts? Is it bigger than it needs to be?

Slide 21

Minute Essay Answer

- It's just the right size — with 5 bits we can represent values of 0 through 31, and the range of possible meaningful shifts ranges from 0 through 31 as well. (Think for a minute about what happens when you shift a 32-bit value 32 bits left or right; is it useful?)

Slide 22