Slide 1

Administrivia

- (None.)

Slide 2

Numbers and Arithmetic — Review/Recap

- As in many other architectures, in MIPS integers are represented as 32-bit binary quantities. They can be signed or unsigned. Signed numbers are represented using two's complement notation.
- Addition in binary works much like addition in decimal (taking into account the different bases). Two's complement means we don't have to do anything special if one value is negative, and we can do subtraction by taking the negative of the second operand (easy) and adding.
- But there is one catch, namely that the value can overflow . . .

Slide 3

Addition/Subtraction and Overflow, Continued

- Notice that we can't get overflow unless input operands have the same sign.
- If we add two positive numbers and get overflow, how can we tell this has happened? Does this always work?
- If we add two negative numbers and get overflow, how can we tell this has happened? Does this always work?

Slide 4

Addition/Subtraction and Overflow, Continued

- When we detect overflow, what do we do about it?
- From a HLL standpoint, we could ignore it, crash the program, set a flag, etc.
- To support various HLL choices, MIPS architecture includes two kinds of addition instructions:
  - Unsigned addition just ignores overflow.
  - Signed addition detects overflow and "generates an exception" (interrupt) — hardware branches to a fixed address ("exception handler"), usually containing operating system code to take appropriate action.

  This is why, if you look at MIPS assembler for C programs, the arithmetic is unsigned — C ignores overflow, so why bother to look for it.

Slide 5

Implementing Arithmetic — Preview

- In the next chapter we start talking about hardware design (though still at a somewhat abstract level).
- For now it may be useful to know that the low-level building blocks are entities that can evaluate Boolean expressions — very simple ones at the lowest level, and slightly more complex ones one level up.
- So for example we can implement addition by first making a "one-bit adder" that maps three inputs (two operands and carry-in) to two outputs (result and carry-out), and then chaining together 32 of them.
- (Multiplication and division may need to be more complex, involving multiple steps and control-flow logic.)

Slide 6

More Arithmetic — Multiplication

- As with addition, first think through how we do this "by hand" in base 10. (Review terminology: In $a \times b$, call a the "multiplicand" and b the "multiplier".) Example?
- We can do the same thing in base 2, but it's simpler, no? computing the partial results is easier. This gives the textbook's first algorithm, figure 3.5. (Work through example if time permits.)

  Notice also that overflow could be a lot worse here — so normally we'll compute a result twice as big as the inputs.

  (We can do better — later.)
- What about signs? Algorithm works, if we extend the sign bit when we shift right.

Slide 7

Multiplication, Continued

- In MIPS architecture, 64-bit product / work area is kept two special-purpose registers (`lo` and `hi`). Two instructions needed to do a multiplication and get the result:

  ```
  mult rs1, rs2
  mflo rdest
  ```

  Assembler provides a "pseudoinstruction":

  ```
  mul rdest, rs1, rs2
  ```

- Notice, however, that a "smart" compiler might turn some multiplications into shifts. (Which ones?)

Slide 8

Division

- As with other arithmetic, first think through how we do this "by hand" in base 10. (Review terminology: We divide "dividend" a by "divisor" b to produce quotient q and remainder r, where $a = bq + r$ and $0 \leq |r| < b$.) Example?

  We can do the same thing in base 2; this gives the algorithm in figure 3.10. (Work through example if time permits.)

  (Here too we can do better — later).

- What about signs? Simplest solution is (they say!) to perform division on non-negative numbers and then fix up signs of the result if need be.

Slide 9

Division, Continued

- In MIPS architecture, 64-bit work area for quotient and remainder is kept in same two special-purpose registers used for multiplication (`lo` and `hi`). After division, quotient is in `lo` and remainder is in `hi`. Two (or more) instructions needed to do a division and get the result:

```
div rs1, rs2
mflo rq
mfhi rr
```

  Assembler provides a "pseudoinstruction":

```
div rdest, rs1, rs2
```

- Notice, however, that a "smart" compiler might turn some divisions into shifts. (Which ones?)

Slide 10

Minute Essay

- Have you had any exposure to the lower-level things mentioned today (AND and OR gates).