Slide 1

Administrivia

- I do send a lot of e-mail-to-all. Consider setting up a TMail filter so it's easier to keep track of? Filter on subject containing "csci 2321" or "csci 1120 / 2321".
- I still intend to share with each of you a "grade summary" similar to what I did at early-alert-grades time. More by e-mail soon?
- I had thought I'd make more-extensive use of TLearn but probably will not. Instead I'm aiming to have information accessible from my home page and/or the course Web site.

Slide 2

Administrivia

- Virtual office hours via Zoom. Hours on my home page (`www.cs.trinity.edu/˜bmassing`) plus link to Trinity-users-only document with links etc.
- Several updates to course Web site, under "Useful links":
  - Information about class meetings and recordings, via links to Trinity-users-only documents.
  - What the University is saying to students about remote learning. Section on "ABCs of remote learning" worth a look, especially "Set boundaries".
- Schedule page updated with next two weeks' topics and readings. Sorry about the delay — I forgot!
- Next quiz a week from today (tentative).

Slide 3

Exam 1 Follow-Up

- I thought trying to reproduce the conditions of an in-class exam remotely would work and be the best option for the result I wanted. I didn't want to just make the exam take-home because I wanted the "timed" aspect.
- This seems to have worked okay for many students, aside from some difficulties figuring out how to "mark up" PDF.
- For others, not so much. Not all of you apparently have access to a quiet work environment. That that didn't occur to me — I really apologize.
- My impression is that everyone made a real effort to make it work. Appreciated!
- More in minute essay.

Slide 4

Numbers and Arithmetic — Overview

- Most current architectures represent integers as fixed-length two's complement binary quantities.

  (But note there are/were architectures that support variable-length "packed decimal", with each byte storing representations of two base-10 digits.)
- Most current architectures these days represent real numbers using one or more of the formats laid out by IEEE 754 standard. Based on a base-2 version of scientific notation, plus special values for zero, plus/minus "infinity", and "not a number" (NaN).

  (But historically there have been architectures that could represent fractional quantities using base-10 "fixed-point" notation, and this may be coming back.)

Slide 5

Numbers and Arithmetic — Overview, Continued

- Arithmetic can (in principle anyway) be done using same techniques taught to grade-school children.

  (Well, I hope still taught? Fans of classic science fiction may know Asimov short story “The Feeling of Power” (1958?), which posits a world in which no humans can do simple arithmetic without a computer. But he didn’t predict how pervasive and affordable computers would become!)

Slide 6

Binary Versus Decimal (Review)

- In decimal (base 10) notation, each digit is multiplied by a power of 10. Same idea for binary (base 2), but using powers of 2.
- So, converting from binary to decimal is easy (if tedious), working from definition.

  Brief example:

  $1011_2 = (8+2+1)_{10} = 11_{10}$

Slide 7

Binary Versus Decimal (Review), Continued

- Converting from decimal to binary? Repeatedly divide by 2 and record remainders . . .
- Why does this work? Could describe this as a recursive algorithm for computing $bits(n)$:
  - Base case is $n{<}2$; trivial.
  - For recursive step, divide n by 2 to get quotient q and remainder r. Then $n{=}2q{+}r$, and:

    Last bit of $bits(n)$ should be r.

    Remaining bits are $bits(q)$, left-shifted by 1.

Slide 8

Other Number Bases (Review)

- Binary useful for showing real internal state but not very compact. Decimal compact but not so easy to convert to/from binary.
- Easy to convert binary to/from power-of-2 base. Hence use of “octal” (base 8) and “hexadecimal” (base 16). For base 16, need more than 10 “digits” to make idea of positional notation work (tangent — very powerful idea! compare to Roman numerals), use letters A etc. (uppercase or lowercase).

  Conversion is based on some simple if tedious algebra: Group bits, right to left, in groups of 3 (for octal) or 4 (for hexadecimal), and factor out a power of 8 or 16 from each group.
- Note — can also convert directly to/from decimal, much as for binary.

Slide 9

Binary Versus Decimal (Review?), Continued

- Terminology: “Least significant” and “most significant” bits.
- Seems like there would be one obvious way to store the multiple bytes of one of these in memory, but no: “big endian” versus “little endian” (names from *Gulliver’s Travels*).

Slide 10

Representing Integers (Review)

- Representing non-negative integers straightforward: Convert to binary and pad on the left with zeros.
- What about negative integers?
- Could try using one bit for sign, but then you have +0 and -0, and there are other complications.
- Or . . . consider analogy of a car odometer: Representable numbers form a circle, and adding 1 to largest number yields 0.

Slide 11

Representing Integers (Review), Continued

- Could implement the car-odometer idea in binary, and then choose where to “cut the circle” (between smallest and largest):
  - Between 0 and all ones — unsigned integers.
  - Between largest number with 0 as the MSB and smallest number with 1 as MSB — “two’s complement” signed integers.
- Note: With this scheme +1/-1 moves “around the circle” — nothing special needed for negative numbers.

Slide 12

Representing Integers (Review), Continued

- Note: If we have n bits, adding 2^n to x gives us x again. Leads to an easy way to compute $-x$: Compute $2^n - x$, and note that

  $2^n - x = (2^n - 1) - x + 1$

  which is very easy to compute . . .
- (This is the familiar(?) method of “flipping the bits” and adding 1. Not magic!)

Slide 13

Signed Versus Unsigned

- If we have n bits, can use them to represent signed values. (What range?)

  Or can use them to represent non-negative values only (“unsigned values”). (What range?)
- Many MIPS instructions have “unsigned” counterparts — `addu`, `addiu`, `sltu`, etc.
- Example: Suppose we have

  `0x00000000` in `$t0`

  `0xfffffff2` in `$t1`

  What happens if we execute `slt $t2, $t0, $t1`?

  What happens if we execute `sltu $t2, $t0, $t1`?

  (Same bits, different interpretations!)

Slide 14

Sign Extension (Review?)

- If we have a number in 16-bit two’s complement notation (e.g., the constant in an I-format instruction), do we know how to “extend” it into a 32-bit number?

  For non-negative numbers, easy.

  For negative numbers, also not too hard — consider taking absolute value, extending it, then taking negative again.
- In effect — “extend” by duplicating sign bit.
- (Note that not all instructions that include a 16-bit constant do this.)

Slide 15

Two’s Complement and Addition/Subtraction (Review)

- Addition in binary works much like addition in decimal (taking into account the different bases). Note what happens if one number is negative.
- Subtraction could also be done the way we do in decimal. But could also compute $a-b$ as $a+(-b)$, which makes for simpler hardware (more about this soon).

Slide 16

Integer Addition/Subtraction and Overflow

- If adding two n-bit numbers, result can be too big to fit in n bits — “overflow”.
- For unsigned numbers, how could we tell this had happened?
- How about for signed numbers?

Slide 17

Addition/Subtraction and Overflow, Continued

- Note 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?
- If we add two negative numbers and get overflow, how can we tell this has happened?
- (Figure 3.2 in textbook summarizes.)

Slide 18

Addition/Subtraction and Overflow, Continued

- When we detect overflow, what do we do about it?
- From a HLL standpoint: 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 fixed address ("exception handler"), usually containing operating-system code to take appropriate action.

Slide 19

Addition/Subtraction and Overflow, Continued

- C can ignore overflow (may depend on implementation — “undefined behavior”?). So a real C compiler for MIPS might use unsigned arithmetic.
- Examples in the textbook don’t do this, perhaps to keep things simpler. SPIM also apparently ignores overflow.

Slide 20

Implementing Arithmetic — Preview

- In next chapter, start talking about hardware design (though still at a somewhat abstract level).
- For now, may be useful to know that the low-level building blocks are entities that can evaluate Boolean expressions(!).
- So for example, 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. (Figures B.5.2, B.5.7.)
- Multiplication and division, however, may need to be more complex, involving multiple steps and control-flow logic.

Slide 21

Multiplication

- (First discuss simple “humans can understand this” / proof of concept approach.)
- As with addition, first think through how we do this “by hand” in base 10. (Example, briefly.)
- Can do the same thing in base 2, but it’s simpler, no? computing the partial results is easier. (More next time.)

Slide 22

Multiplication, Continued

- In MIPS architecture, 64-bit product / work area kept in 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
  ```

- Note, however, that a “smart” compiler might turn some multiplications into shifts. (Which ones?)

Slide 23

Division

- (Again, first discuss simple “humans can understand this” / proof of concept approach.)
- As with other arithmetic, first think through how we do this “by hand” in base 10. (Example, briefly.)
- Can do the same thing in base 2. More next time.

Slide 24

Division, Continued

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

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

  Assembler provides a “pseudoinstruction”:

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

- Note, however, that a “smart” compiler might turn some divisions into shifts. (Which ones?)

Slide 25

Minute Essay

- As noted, I had what I thought were reasonable reasons for asking you to do the exam in real time at a fixed time. How did that work for you? Were you able to actually focus on the exam during one of those two times?
- If you turned in a PDF, how did you produce it? exam on paper and then scan, edit PDF in place (with what tool?), . . . ?
- Any comments right now about content? I'll probably ask again when I return them (probably by putting something in your "graded work" folders).