Slide 1

Administrivia

- Reminder: Homework 4 due Friday. (If you get it to me by 5:30pm, there's at least a chance I can get it graded by Monday.)
- Exam 1 next Wednesday in class. Review sheet posted. We can spend at least part of next class reviewing.
- Homework 2 graded. (At last!)

Slide 2

Minute Essay From Last Lecture

- Pretty much unanimously people liked doing a problem in class! I'll try to do more of this, at least from time to time.

Slide 3

Homework 2 Essays

- Several people said things were starting to make sense (sometimes they also said initially things were confusing and difficult). That's the goal of homework! One said "looks harder than it is". I tend to agree?
- One problems didn't seem hard, just tedious and not of much practical value. Maybe so, but my feeling is that the understanding I hope you get helps round out your education as a computer scientist.
- One said it was interesting how we were starting to de-mystify things that had been mysterious.
- One said the last problem made him think about different kinds of loops in C. Indeed!

Slide 4

Homework 2 Essays, Continued

- One said he was thankful for HLLs and compilers. Indeed!
- Perhaps the most interesting comment: Examples of compiling C seem long and tedious; hard to imagine writing a whole compiler!

  Indeed! but the way I understand it, people don't: If starting from scratch (unlikely these days), typically start by writing a sort of starter compiler, enough to translate at least some of the language, no optimization. Then use it to develop a slightly more complicated compiler . . . "Lather, rinse, repeat"?

Slide 5

Practice Problem Recap/Review

- When writing programs in MIPS assembly language, I like to start by writing C and then translate. Allows me to debug basic logic in a language with better support for that.
- (Look at my answer for problem?)

Slide 6

Sidebar(?): Parallel Execution and Synchronization

- A lot of commodity hardware these days features multiple processing units ("cores") sharing access to memory. One reason for this is that in theory we can make individual applications faster by splitting computation up among processing elements.
- Having processing elements share memory makes parallel programming easier in some ways but has risks ("race conditions"). Avoiding the risks requires some way to control access to shared variables (e.g., to implement notion of "lock").

Slide 7

Parallel Execution and Synchronization, Continued

- Most texts on operating systems discuss synchronization issues and present several solutions ("synchronization mechanisms"), some rather high-level and others not.

  (Why is this in O/S textbooks? because O/Ss typically have to manage "processes" executing concurrently, either truly at the same time or interleaved.)

- The most primitive can (with some simplifying assumptions) be implemented with no hardware support. But hardware support is very useful.

Slide 8

Sidebar: Why is Implementing a Lock Hard?

- It might seem like it would be straightforward to implement a lock — just have an integer variable, with value 0 meaning "unlocked" and anything else meaning "locked". And then you "lock" by looping until the value is 0, then setting to nonzero:

```
while (lock != 0) {}
lock = 1;
```

  and "unlock" by setting back to 0.

- But this doesn't work! (Why not?)

Slide 9

Instructions for Synchronization

- Key goal in designing hardware support for synchronization is to provide "atomic" (indivisible) load-and-store. This allows writing a low-level implementation of "lock" idea.
- Many architectures do this with a single instruction (e.g., "test and set" or "compare and swap"). Requires two accesses to memory so may be difficult to implement efficiently.
- MIPS approach: Same idea, but using a pair of instructions, `ll` ("load linked") and `sc` ("store conditional").

Slide 10

MIPS Instructions for Synchronization

- `ll` loads a value from memory and somehow remembers the location and value. Syntax:

  ```
  ll reg1, displacement(reg2)
  ```

  Operands used as for `lw`.
- `sc` stores a value into memory — *IF* the location has not changed since a previous `ll` from that address.

  ```
  sc reg1, displacement(reg2)
  ```

  Operands used almost as for `lw`, except that `reg1` is set to indicate whether the store "succeeded" (i.e., value had not changed since `ll`). So one can regard a (`ll`, `sc`) pair as forming a single atomic load/store.
- (How to make this work? Hardware designers' problem! glib answer but maybe all we can do in this course.)

Slide 11

MIPS Instructions for Synchronization — Example

- Example of use (from textbook p. 122):

```
again:
    addi    $t0, $zero, 1       # $t0 <- value to place in lock
    ll  $t1, 0($s1)             # $t1 <- old lock value
    sc  $t0, 0($s1)             # try to set new lock value
    beq $t0, $zero, again       # $t0 == 0 means store failed
                                #   (so try again)
    add $s4, $zero, $t1         # $s4 <- old lock value
```

Slide 12

Multiplication and Division

- In the factorial example I use what appears to be an instruction `mul`. Really a pseudoinstruction (though SPIM doesn't seem to think so).
- "Real" multiply has only two operands

  `mult src_reg1, src_reg2`

  and it puts a 64-bit result in special-purpose registers `lo`, `hi`. Can access them with `mflo`, `mfhi`, e.g.,

  `mflo dest_reg`
- Divide (`div`) similar; quotient goes in `lo` and remainder in `hi`.

Slide 13

More Examples

- Try adding code to the factorial example to check for overflow. (I'll write this, maybe with your help.)
- Try writing a procedure like the simple one I show for CSCI 1120 to divide. (Practice problem.)

Slide 14

Minute Essay

- In the programming problem for Homework 4, I say you won't get full credit if you don't follow conventions for calling procedures. Why does this matter? Couldn't you pass arguments to a procedure in whatever registers you want, as long as caller and called agree?
- And why save/restore registers?

Slide 15

Minute Essay Answer

- If you write both the calling program and its called procedures, it might seem like it hardly matters how you communicate between caller and called. But think about how well (or not well) this would work for a larger project! and even for small projects, isn't it easier to always follow convention rather than inventing one for each procedure?
- Saving/restoring registers . . . You can skip this if the procedure doesn't modify any of the registers normally saved/restored. I say probably good style to do it anyway; better to just copy boilerplate than try to think through exactly what each case needs?