**Slide 1**

## Administrivia

- (By e-mail.)

**Slide 2**

## Working With Characters

- Character data can be ASCII (1 byte per character) or Unicode (2 bytes per character). Character strings can be represented various ways (`struct` or similar, or null-terminated). (Aside: Unicode is — remarkable in how many alphabets have encodings. Wikipedia article is interesting!)
- How to work with characters? `lb`/`sb`, `lh`/`sh`.
- Nothing very deep here?

**Slide 3**

## Working with Constants, Revisited

- Recall `addi` instruction. Exists because often we need to use a small constant in a program.
- Uses same format ("I format") as `lw` and `sw`, which allows 16 bits for constant.
- What if we need more than 16 bits? "Load upper immediate" instruction:

  ```
  lui register, constant
  ```

  Puts (16-bit) constant in "upper" 16 bits of register. Follow with `addi` (or, better, `ori`) to load a full 32-bit constant.
- Example: two instructions assembler generates for `la` pseudoinstruction (example in simulator).

**Slide 4**

## Addressing Modes

- We've been unspecific about how to specify addresses of a lot of things.
- So, now look at various "addressing modes" — ways to specify where to find an operand.
- Which is used? Defined by instruction format (R, I, J). (J? yes, format for jump instructions that include a label — `jal` and `j`.)

**Slide 5**

## Addressing Modes, Continued

- Register addressing: Value is in one of the general-purpose registers. Assembler defines symbolic names for them (e.g., `$t0`).
- Immediate addressing: Value is in instruction itself (as in, e.g., `addi`).
- Base-displacement addressing: Value is in memory, with address calculated by adding a displacement to what's in a register. Example is memory-address operand of `lw`, `sw`.
- PC-relative addressing (more shortly).
- Pseudo-direct addressing (more shortly).

**Slide 6**

## PC-Relative Addressing

- Address is formed by adding offset in instruction (16 bits) and contents of the program counter (special register).

  (Actually, address is offset times 4, plus the *updated* program counter. Simulator doesn't quite simulate this, unless run with the flag `-delayed_branches`.
- Example is conditional branches (`beq`, `bne`).
- Does this limit what we can do with `beq` and `bne`? If so, how often will it matter? What could we do to work around it?

Slide 7

## PC-Relative Addressing, Continued

- 16-bit offset obviously limits how far we can "jump". But probably fine for most uses (conditional execution, loops).
- If not, can rework code to use `j` or `jr`. (Apparently assemblers are supposed to do this if the offset is too big.)

Slide 8

## PC-Relative Addressing — Example

- As an example, try working out machine code for the `bne` in this line. (May be helpful to annotate with relative locations so we easily compute offset we need.)

```
        bne     $t0, $t1, There
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero
        add     $t4, $zero, $zero
There:
        sub     $t5, $zero, $zero
```

**Slide 9**

## PC-Relative Addressing — Example, Continued

- Look up opcode — `0x5`.
- Look up register numbers — 8, 9.
- Compute needed offset by . . . Strictly speaking, should be offset from relative location of instruction *after* the `bne` to "branch target" (`There`), *divided by 4*. (Why divided by 4? always a multiple of 4! so last two digits always 0 . . . ) But just counting instructions gives the same effect (and here's it 3).
- Rearranging bits and converting to hexadecimal, we get `0x15090003`.

  Does this agree with what SPIM shows? Not quite . . .

**Slide 10**

## PC-Relative Addressing — Example, Continued

- In real implementations, PC has already been incremented when branch executes. This means that the instruction right after the branch is executed whether the branch succeeds or not — "branch delay slot". (May depend on version of MIPS.)
- Ignoring this behavior keeps examples manageable, so that's what SPIM does — and it calculates offsets from current instruction. If I ask you to translate a branch into machine code I want you to do the right thing rather than what SPIM does.

**Slide 11**

## Pseudo-Direct Addressing

- Address is formed by combining address in instruction (26 bits) and upper bits of program counter:

  As with PC-relative addressing, no real need to store last 2 digits, since always 0.

  Actual address is address field in instruction, times 4, OR'd with upper bits of program counter to give 32 bits in all.
- Example of use is unconditional branch (`j`).
- Does this limit what we can do with `j`? If so, will that be a problem? Can we work around it?

**Slide 12**

## Pseudo-Direct Addressing, Continued

- 26-bit address does limit what we can do, but probably fine for most uses (conditional execution and loops, procedure calls).
- If not enough, can rework code to use `jr`. (And in fact assemblers may do this.)

Slide 13

## Pseudo-Direct Addressing — Example

- As an example, trying working out machine code for the previous example with `j There` replacing the `bne`:

```
        j       There
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero
        add     $t4, $zero, $zero
There:
        sub     $t5, $zero, $zero
```

Slide 14

## Pseudo-Direct Addressing — Example, Continued

- Look up opcode — `0x2`.
- To get 26-bit value for the address, need not a relative location (as for `bne`) but an absolute one.

  To do that, need to know where in memory the (machine) code resides. Suppose we paste this code into the starter example, right after the "opening linkage" code, and use as starting address of whole progrram location where SPIM puts `main:`. That's `0x0040 0024`. Counting up, get an address of `0x0040 003c` for `There`. Remove top four bits of that and divide by 4 to get

  `0000 0100 0000 0000 0000 0011 11`
- Putting the two fields together and converting to hexadecimal gives `0810000f`, which agrees with SPIM.

**Slide 15**

## 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 16**

## 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 17**

## 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 18**

## 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 19

## 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 20

## Variables — Review/Clarification

- Declaring a variable in a high-level language (e.g., `int x;` in C) reserves space for it in memory (in principle anyway — more shortly) and assigns it a name (for the purposes of compilation).

  Space can be in "data" segment of memory, for static/global variables, or "on stack" for local variables.

- Referencing the variable implies accessing the associated memory location. (Figuring out the instructions to do that is part of the compiler's job. Presumably it has some sort of map from names to locations.)

  In MIPS, that means a load (for read) or store (for write). A very simple compiler would do this for every access. But . . .

Slide 21

## Variables, Continued

- Memory access is slow compared to processor speed, so good compilers will streamline things by sometimes keeping values of frequently-used variables in registers, only loading or storing when necessary to preserve semantics.

  This is why the textbook examples talk about associating registers with variables. (Clearer?)

- I said "in principle" because a good compiler might even figure out that it might be possible to just use a register to hold a variable's value and never assign it a memory location. Simple contrived example:

```
int foobar(int x) {
    int y = x+1;
    return y;
}
```

  No need to have `y` in memory at all, right?

Slide 22

## Minute Essay

- How much of the discussion of parallelism was review for you?:
- Questions?