Slide 1

Administrivia

- Reminder: Homework 1 due today. Hardcopy now, or later in my mailbox in the ASO (Administrative Support Office in the CSI).
- Next homework to be on the Web soon, due in a week.
- Quiz 1 a week from today. Open book/notes. Likely topics those covered by homework.
- We should discuss whether to reschedule the midterm for after spring break.
- Wikipedia article on “MIPS architecture” is (mildly) interesting reading. Still in use!

Slide 2

Recap/Review

- We’ve talked about a small suite of MIPS instructions for basic arithmetic and conditional execution.
- We’ve also talked about how to encode them in binary — except we haven’t yet talked about how to address addressing.
- So we almost know everything we need to know to translate programs in an HLL into assembler — except we don’t know how to do functions.

Slide 3

Procedure Calls

- How do we call procedures (a.k.a. functions, methods)? Consider an example:

```
a = a + a;
x = foo(a);
b = b + b;
y = foo(b);
```

- If we've compiled this code (and function `foo`), what do we have in memory when it's running? What's supposed to happen when we get to a call to `foo`?

Slide 4

Procedure Calls, Continued

- So, what we have to do to call a procedure is:

1. Put parameters where procedure can find them.
2. Transfer control to procedure.
3. Acquire storage resources for procedure (recall that every time you call a C function you get a "new copy" of all its local variables).
4. Run procedure.
5. Put results where caller can find them.
6. Return control to caller.

- How to do all this?

Slide 5

Register Conventions

- From hardware point of view, all registers are in some sense the same (except 0).
- From software point of view, it's useful to agree about how to use them — for parameters, return values, etc. Idea is that compilers automatically enforce conventions, human-written assembly code should follow them too.
- So far — `$s0` through `$s7` used for variables, `$t0` through `$t9` used as "scratch pads". (See reference card for numeric equivalents.)
- Add two more groups — `$a0` through `$a3` for parameters (punt for now on what to do if more than four), `$v0` and `$v1` for return values.

Slide 6

Jumping To/From Procedures

- When we jump to a procedure, must remember where we came from so we can return. Do this with "jump and link"

  ```
  jal    label
  ```

  which puts address of next instruction in register `$ra` and jumps to `label`. (How do we know address of next instruction? "Program counter" (special register) has address of current instruction.)

- We can then get back with "jump to register"

  ```
  jr   r1
  ```

  which jumps to address in register `r1`.

Slide 7

Register Saving and Local Variables

- Actually running the called procedure is straightforward, right?
- Yes, except we need some way to save/restore registers — so we don't mess up caller (by convention, "temporary" registers might change, but most others don't).
- We also need a way to make space for local variables.

Slide 8

Register Saving and Local Variables, Continued

- Common solution — use part of memory as a stack (familiar ADT, right?), for saving registers and other local storage. Makes recursive procedures easier.
- By convention, stack starts at high address and "grows" to lower addresses, and register `$sp` ("stack pointer") points to top.
- How to push / pop?
- Since `$sp` can change during computation, can use register `$fp` ("frame pointer") to point to start of area ("procedure frame") for saved registers, local variables.

Slide 9

Other Variables

- Last but not least, we (may?) need someplace to store variables that can be preallocated (static/global) and variables that are dynamically allocated (e.g., with `malloc` in C).
- By convention, we put them right after the program code and use register `$gp` (“global pointer”) to point to them. Typically call the memory used for dynamically-allocated variables “the heap”.

Slide 10

Procedure Calls, Revisited

- Calling procedure must:
  - Put parameters in `$a0` through `$a3` (if more than four, on stack).
  - Determine address of called procedure and jump there, saving address of next instruction.
  - Get return value from `$v0` (and `$v1`, if used).
- Called procedure must:
  - Save registers as needed, including return address.
  - Retrieve parameters and do calculation.
  - Put results in `$v0` and `$v1`.
  - Restore saved registers.
  - Return to caller.

Slide 11

Example

- How to compile the following?

```
int main() {
        a = 5; b = 6; c = 7;
        x = addproc(a, b, c);
        return 0;
}
int addproc(int a, int b, int c) {
        int x;
        x = a + b + c;
        return x;
}
```

(Sample program `call-addproc.s`.)

Slide 12

Data Formats, Revisited

- Recall, inside the computer “it’s all ones and zeros” — so we must encode anything we want to represent.
- Integers — binary numbers, often 32 bits for MIPS, but could be other sizes too. Several choices for signed numbers; two’s complement is the most common.
- Real numbers — floating-point format, later.
- Text — ASCII (8 bits per character) or Unicode (16 bits). Strings represented with explicit length field (Java) or terminating character (C).
- Many, many more complex formats (`.doc`, `MP3`, `GIF`, etc.).

Slide 13

More Load/Store Instructions

- MIPS architecture defines `lw` and `sw` for loading/storing data in 32-bit chunks; also defines `lb` (“load byte”) and `sb` (“store byte”) for loading/storing data in 8-bit chunks, plus instructions to load/store data in 16-bit chunks.

  All must align on appropriate boundaries.

Slide 14

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.

Slide 15

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).

Slide 16

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.
- 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 17

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.)
- 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 18

Pseudo-Direct Addressing

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

  (Actually, address is address in instruction times 4, or'd with upper bits of program counter.)
- Example 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 19

Minute Essay

- Would you object to rescheduling the midterm for after spring break? say the Thursday after (3/20)?
- What does the following code do? i.e., what is in registers $s0 and $s1 after it executes?

```
        add     $s0, $zero, $zero
        addi    $s1, $zero, 1
        addi    $s2, $zero, 4
l1:
        addi    $s0, $s0, 1
        add     $s1, $s1, $s1
        bne     $s0, $s2, l1
```

Slide 20

Minute Essay Answer

- We could trace through the code, which sets values in three registers and then executes a loop:

  $s0 is initially set to 0 and then takes on values 1, 2, 3, and 4

  $s1 is initially set to 1 and then takes on values 2, 4, 8, and 16

  $s2 is initially set to 4 and doesn't change