Communicating with Computers

2-3-2003

Opening Discussion

- What did we talk about last class?
- AMD PR ratings: would you rather compare MHz? Which is better?
- What do you know about assembly language programming? Have any of you ever done any assembly.

Speaking Their Language

- Your CS career has largely been about learning how to talk to a computer. However, everything you have said to them has had to go through a translator (or two) before getting to a point where the machine could understand it.
- Our objective in at this point is to get rid of one of those translators to get you a bit closer to the machine.

Assembly Language

- Machines speak machine language (duh!). We will be looking at that, but first we want to look at something that is just a step above it, assembly language.
- Assembly language is basically a way that we can encode machine language in something that we can read/write a bit more easily. Still, it is very near to the machine code it will be turned into.

Our First Assembly Instructions

- `add a,b,c`
  - a=b+c
- `sub a,b,c`
  - a=b-c
- All operations on MIPS take three arguments.
- The arguments to these instructions must be registers. There are 32 registers in MIPS machines.

Design Principle 1

- Your book occasionally puts in certain "design principles" as general rules that chip designers would want to follow.
- Your first one is: "Simplicity favors regularity." This makes sense, the fewer exceptions there are, the simpler it is to build something that follows your rules. We first see this in MIPS where instructions "always have three arguments".

Design Principle 2

- Smaller is faster (most of the time)
- This relates to the speed of light limits. It is why there are only 32 registers in MIPS and why registers are faster than cache which is faster than memory.
- This was part of the problem that the SPARC had. They had a "sliding" register window. You could only use 16 registers at a time, but they had something more like 128 total.

Naming Registers on MIPS

- Variable registers are called $s0, $s1, …, $s7.
- Registers used to store temporary values are called $t0, $t1, …, $t9.
- So the add operation might really look like this
  - `add $t0, $s1, $s3`
- All arithmetic operations are done on registers in the MIPS architecture.

Memory, Addresses, and Value

- Since the processor can only store 32 items at one time to actually do work on, it must go out to memory regularly to fill those registers, or store back calculated values.
- You can view memory as a huge array of bytes each single offset moves you one byte further.

The Load Instruction

- The primary instruction used to load from memory on MIPS is the "load word" command, lw.
- The three arguments to this are a register to load to, a constant offset, and a register base address.
  - `lw $t0, 0($s0) # load *$s0 to $t0`
- The offset should be a multiple of 4 so it is word aligned.

Layout of Bits

- Big Endian vs. Little Endian
- The order of bytes in a number is a free design parameter and is done differently on different machines.
- The chapter says MIPS is big endian while the appendix says it can be either. I think the appendix was talking about SPIM.
- x86 chips are little endian.

The Store Instruction

- The primary instruction used to store to memory on MIPS is the "store word" command, sw.
- `sw` takes three arguments just like `lw`.
  - `sw $t0, 0($s0) # store $t0 to *$s0`
- Again, the offset should be a multiple of 4 so it is word aligned.

Register Optimization

- Most of your program will use more than 32 variables (and far more than the 22 you really get to use on a MIPS chip). As a result, compilers have to put in load and store operations for some variables.
- Obviously you want to do as few load and store operations as possible. Figuring out how to achieve this can require complex algorithms like graph coloring algorithms.

Example

- Let's see how well you understand this now. Let's write a non-recursive C program to calculate Fibonacci numbers and then translate it to MIPS assembly.

Minute Essay

- Assume that you have the address of an array of ints in $s0. Write a segment of code to add the first five elements of that array so it ends up in $s1.
- Quiz #2 will be at the beginning of the next class.