Addressing and Arrays vs. Pointers

2-12-2003

Opening Discussion

- What did we talk about last class?
- Why make the assembly language explicitly handle the job of dealing with the stack?

Recursive Fibonacci

- Let's take a few minutes to write a recursive implementation of a function that calculates Fibonacci numbers.

Immediate Instructions

- Technically, when we use constants in arithmetic or comparison operations we use slightly different instructions that use the I-type machine language format.
- If we need more than 16 bits to express a value the lui instruction lets us put an immediate value in the upper 16 bits of a register.

Design Principle 4

- Make the common case fast.
- This seems like a very common sense principle but it isn't that straight forward. The real question is trying to figure out what the common case is. This can be especially hard for a new instruction set.
- A corollary of this is that the exceptional cases can be slow. Should they add complexity?

Addressing for Jumps

- Because programs can be bigger than 64k, we can't use straight addressing for all jumps, even for I-type instructions.
- PC-referenced addressing is used for conditional branches because they typically don't go that far. It also uses a word count instead of a byte count.
- J-type instructions with 26-bit addresses are used for the j and jal instructions. They also hit only word addresses.

Arrays vs. Pointers

- In C/C++, arrays basically are pointers. However, you can write code that handles it with array syntax or pointer syntax. These two translate into somewhat different assembly language. Your book goes through an example, but they fail to optimize the pointer approach. Let's look at this.

Code

- Let's go through and do an example of initializing the elements of an array to a certain value using both syntax forms and then convert them to assembly and see if we can optimize it.
- Are you familiar with the "pointer style". Can you read it that way? Could you write it that way?

Minute Essay

- A classic problem in theory of CS is the 3n+1 problem. Given the following routine, the question is, does it terminate for all possible inputs. Translate this to assembly. You can use the rem function like a modulo (rem rdest, rnum, rdenom).

```
void TN1(int n) {
     while(n>1) {
          if(n%2==1) n=3*n+1;
          else n=n/2;
     }
}
```