Packaging of Instructions

2-5-2003

Opening Discussion

- Do you have any questions about the quiz?
- What did we talk about last class?

Code for Summing an Array

- In the last minute essay I asked you to write code that would sum the elements of an array. A fair number of you wrote something that was roughly correct. Most who didn't forgot to load in values.
- The array is in memory (I stated its address was stored in \$s0). You have to load the array values before you can add them. What is the optimal way to do this?

Binary Numbers

- Because they matter for this section we should do a quick review of binary numbers, we'll do more later for chapter 4. For now all that matters in that binary works just like decimal, but there are only two possibilities for each digit (0 and 1) and each new digit to the left is a higher power of 2 instead of 10.
- Ex. 101101=32+8+4+1=45

Encoding Machine Language

- The machine only stores bits (0 or 1) so all instructions have to be encoded in numbers this way.
- This includes the operator and its operands. (No "add" in memory)
- Registers are encoded as follows:
  - $s0-$s7 -> 16-23
  - $t0-$t7 -> 8-15, $t8-$t9 -> 24-25
- All MIPS instructions are 32 bits long.

Fields of a MIPS Instruction

- We can break an instruction into separate fields, the value of each tells us what we need to know.

op	rs	rt	rd	shamt	funct
6 bits	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**

- op=opcode, rs=first source, rt=second source, rd=destination, shamt=shift amount, funct=function code
- This can't work for everything. It has problems with loads.

Design Principle 3

- Good design demands good compromises.
- In MIPS, they left all instruction 32-bits long, but changed the layout for some. The idea was that this reduced complexity more than variable sized instructions.
- The type of instruction just discussed is called an R-type instruction. There are also I-type instructions. They are used for data transfer.

Fields of I-type

- Data transfer instructions use an alternate arrangement for their fields.

op	rs	rt	address
6 bits	**5 bits**	**5 bits**	**16 bits**

- op=opcode, rs=base register, rt=value register, address=offset from base
- This allows any offset within $\pm 2^{15}$ or -32,768 to 32,767.

Opcodes and Function Codes

- Every instruction that you can perform on a MIPS processor is specified by some combination of opcode and function code. I-type instructions have to be specified by the opcode alone.

	op	**funct**
add	**0**	**32**
sub	**0**	**34**
lw	**35**	**n.a.**
sw	**43**	**n.a.**

Stored-Program Computers

- The idea of encoding instructions and programs as numbers is the foundation of modern computing. We build one type of memory and mix both in it.
- The semantics of this encoding of the instructions is what defines the "instruction set architecture". Any program that follows it can run on any chip that uses it.

Self-Modifying Code?

- It doesn't take a huge leap of the imagination to see that if programs and data are stored together you might be able to modify code the same way you can data.
- This type of trick used to be used more when people worked more closely with assembly/machine language. It's inherently unsafe though.

Branching Instructions

- You need these to do anything fun. Conditional branches only branch sometimes and give up the power to do things like if and loops. They really work like an if with a goto.

```
          beq $s0, $s1, Else
          add $s2, $s0, $s0
     Else:sub $s2, $s2, 1
```

	Type	op	rs	rt	address
beq	I	4	reg	reg	Label addr
bne	I	5	reg	reg	Label addr

Inequalities?

- If you want to check if one number is less than another you can't use beq or bne by themselves. Instead MIPS gives you a helper instruction *set on less than*, slt. This sets the first argument to 1 if the second is less than the third.

	Type	op	rs	rt	rd	shamt	funct
slt	R	0	reg	reg	reg	0	42

Time to Code?

- If we have time let's go ahead and try to start that routine for calculating Fibonacci numbers.

Minute Essay

- Write a little piece of assembly code that will double the value in $s1 until the value in $t0 is zero. Note that this is a loop. Feel free to make it a post check loop.
- For next class make sure you have read through 3.6 and have looked at appendix A.