CS2200 Recitation 1 Section A4

Welcome to CS2200!

Who are we?

Jacob Meyers

jmeyers35@gatech.edu (Please put [CS2200] in the subject)

Third-year CS Major (Sys/Arch & Intelligence)

Fourth semester as a CoC TA, first semester TAing CS2200

I enjoy listening to music, working out, and having my heart broken by my favorite sports teams

Sanjana Kadiveti

sanjanakadiveti@gatech.edu

Third-year CS (Info & Intel)

2nd semester TAing 2200

Conversation!
(I tried)

What is CS2200?

- The best class at Georgia Tech!
- An *integrated* course on computer hardware, operating systems, and networking
  - Common theme - handshake between hardware and software
- Highly conceptual - you need to understand the WHY behind concepts!
  - Buy the book, it's worth it (especially you Sys/Arch folks)

Some Logistics

- 5 projects
  - First two will be hardware-oriented and in CircuitSim
  - Last three will be OS oriented and in C
- ~10 Homeworks
  - Due at the beginning of recitation by 6:15pm
- Yes, recitation is mandatory :)

This Week's Announcements

- HW1 is out tonight!
  - Due next Wednesday (1/23) at the beginning of recitation or in office hours **before** recitation
  - **Bring your Buzzcard**
  - Print out double-sided (there is a scaling point deduction for failure to do so!)
- Project 1 (Datapath) will be released on Friday
  - Due 2/8
  - Start early!
  - No, seriously, start early. You will regret it if you don't.

Questions?

ISAs

What happens when we run a C program?

What we learned in 2110

- C Program -> Compiler -> Machine Code
- This isn't the whole story!

The Reality™

- C Program -> Compiler -> Assembly Code -> Assembler -> Machine Code
- You can use gcc's -S flag to output the assembly code

What is an ISA?

- Instruction Set Architecture
- Defines instructions supported by a particular processor as well as implementation details such as opcodes and instruction formats
- Example: In LC-2200, the ADD instruction is formatted as such:
    - Add $Rx, $Ry, $Rz
    - [opcode 31-28] [Rx 27-24] [Ry 23-20] [Reserved 19-4] [Rz 3-0]

So what? Why do we care?

The Big Picture

- The ISA defines a contract between hardware and software!
  - **“Meeting point between software and hardware”**
    - What does this mean?
- Compiler writers need it to know what assembly instructions can be used for a given processor
- Processor needs it to know how to interpret instructions written by programmers

RISC-y Business

- At some point in the 1980's, two schools of thought developed when it comes to ISA implementation
  - Some people thought ISAs should have a relatively small number of instructions, all of which can be executed in one clock cycle (RISC)
  - Some people thought ISAs should have a plethora of instructions, some of which perform multiple, complex operations over the course of several clock cycles (CISC)
    - CISC places a focus on decreasing number of instructions in assembly (compiler does less work, less RAM needed to store code)
- Who won? Well, it's complicated...
  - The most prevalent desktop/laptop CPU (x86) is a CISC architecture
  - ARM architecture, prevalent in mobile devices such as smartphones, are a RISC architecture
  - With memory becoming cheaper and cheaper, and compiler technology becoming more sophisticated, RISC architectures have largely overtaken CISC

Operating System

- Orchestrates synchronization
- Ensures everything is run when it needs to be run
- Interrupts
- Processor Availability

What goes into designing an ISA?

- We need to consider constructs of high-level languages
  - Expressions, assignment statements
  - Data abstractions
  - Conditionals and loops
  - Procedure/function calls
- What is a Register File?
  - What's the point?

What goes into designing an ISA?

- We should also consider addressing modes when selecting and implementing instructions
  - Register Addressing - Uses only registers.
    - Example: add
  - Base + Offset - Uses a base address contained in a register and an immediate value offset to compute a memory address to access (“effective address”)
    - Example: lw, sw
  - Base + Index - Uses a base address and an index address, both in registers to compute an effective address
    - Useful for array accesses in loops
  - PC-Relative - Some offset is added to the PC to compute an effective address
    - Used in branching instructions

Addressability

What do we *really* read from memory?

- Suppose we want to read an int from memory. How does this take place? Should we read the entire 4 bytes? What about 1 byte at a time?
- “Addressability” refers to the smallest unit that the ISA allows to be read from memory
    - Some common addressabilities are word-addressable and byte addressable
- LC-2200 only offers store word and load word instructions, so we say that LC-2200 is word addressable

Endianness

A Problem

- Suppose we read a word from memory. How can we tell which byte is the Most Significant Byte (MSB)?

A Solution (Kinda)

- Two Possibilities
  - The MSB is at the lowest memory address read (“Big Endian”)
  - The MSB is at the highest memory address read (“Little Endian”)

Example

0x15D8A172 in memory (Little Endian)

0x100	0x101	0x102	0x103
0x72	0xA1	0xD8	0x15

0x15D8A172 in memory (Big Endian)

0x100	0x101	0x102	0x103
0x15	0xD8	0xA1	0x72

Try It Yourself!

- Write the following as they would appear in memory in **both** Big Endian and Little Endian
  - int x = 0xC0D3D00D
  - char arr[] = {0xAA, 0xBB, 0xCC, 0xDD}

Aligned/Unaligned Access

Quick Example

- Imagine you are a compiler-writer, and want to place the int 0x45b7f094 into memory. Where would be the best place to store the first byte? In other words, where should the int begin? Assume Big Endian architecture with 32-bit words and word addressability.

+ 3	+ 2	+ 1	+ 0	
				0x100
				0x104

Some Possible Answers

+ 3	+ 2	+ 1	+ 0	
0x94	0xf0	0xb7	0x45	0x100
				0x104

+ 3	+ 2	+ 1	+ 0	
0xf0	0xb7	0x45		0x100
			0x94	0x104

+ 3	+ 2	+ 1	+ 0	
0xb7	0x45			0x100
		0x94	0xf0	0x104

+ 3	+ 2	+ 1	+ 0	
0x45				0x100
	0x94	0xf0	0xb7	0x104

The Correct Answer

+ 3	+ 2	+ 1	+ 0	
0x94	0xf0	0xb7	0x45	0x100
				0x104

Unaligned Accesses are slow!

- Most processors read word length data in one access - so what do we do about unaligned accesses?
- Disallow them entirely!
  - This is done by having various load instructions - load word, load byte, load short, etc
- This creates some extra work for compilers
  - Need to be aware of **Alignment Restrictions** and pad memory layouts accordingly

Padding Example

```
short a;    // a1a2
int b;      // b1b2b3b4
char c;
```

+ 3	+ 2	+ 1	+ 0	
		a2	a1	0x100
b4	b3	b2	b1	0x104
			c	0x108

LC-2200

Overview

- RISC Instruction Set
  - Only 8 Instructions!
- 32-bit instructions and data
- 16 programmer-visible registers (as seen to the right)
- Word Addressability

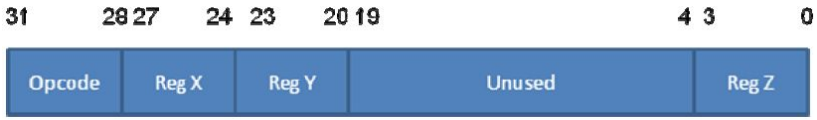
Reg #	Name	Use	callee-save?
0	$zero	always zero (by hardware)	n.a.
1	$at	reserved for assembler	n.a.
2	$v0	return value	No
3	$a0	argument	No
4	$a1	argument	No
5	$a2	argument	No
6	$t0	Temporary	No
7	$t1	Temporary	No
8	$t2	Temporary	No
9	$s0	Saved register	YES
10	$s1	Saved register	YES
11	$s2	Saved register	YES
12	$k0	reserved for OS/traps	n.a.
13	$sp	Stack pointer	No
14	$fp	Frame pointer	YES
15	$ra	return address	No

Table 2.2: Register convention

Instruction Types

R-type

`and, nand`

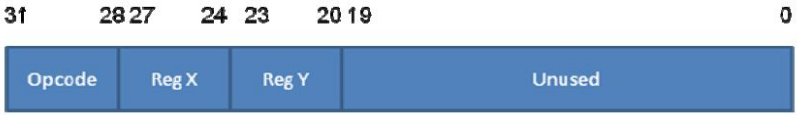


I-type

`addi, lw, sw, beq`

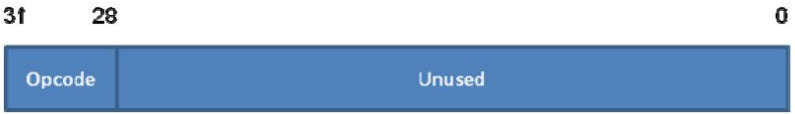
J-type

`jalr`



O-type

`halt`



LC-2200 Instructions

- ADD
  - add $dst, $src1, $src2
  - $dst <- $src1 + $src2
  - add $t2, $t0, $t1
- NAND
  - nand $dst, $src1, $src2
  - $dst <- ~($src1 & $src2)
  - nand $t2, $t0, $t1
- ADDI
  - addi $dst, $src, offset
  - $dst <- $src + offset
  - addi $t0, $t1, 15
- LW
  - lw $dst, offset($base)
  - $dst <- MEM[$base + offset]
  - lw $t0, 0x4($s0)
- SW
  - sw $src, offset($base)
  - MEM[$base + offset] <- $src
  - sw $t0, 0x4($s0)
- BEQ
  - beq $r1, $r2, offset
  - if r1 == r2: PC = PC + 1 + offset
  - beq $t0, $t1, done
- JALR
  - jalr $at, $ra
  - $ra <- PC + 1, PC <- $at
  -
- HALT
  - halt

Some Examples To Try

- Write an LC-2200 program to take an integer (assume it's stored in $s0) and return its negative (store it in $v0)
- Write an LC-2200 program to represent the basic conditional statement below. Assume the value of a is stored in $s0. Place return values in $v0.

```
if (a == 0) {
   return 0
} else {
   return 1
}
```

Calling Conventions

The Problem With Procedure Calls

```
...
addi $s0, $zero, 1        !$s0 contains important value for later
addi $a0, $zero, 2
jalr $at, $ra             !jumps to double routine which doubles the argument
add  $s1, $v0, $s0        !returns the returned value plus $s0
...
```

See the Issue?

- When we make procedure calls, we want to be able to resume where we left off before the procedure call
    - Need to track the state of the registers
- One problem - we don’t know what registers a procedure call will use!
- Enter the calling convention

Steps in Compiling Procedure Calls

- Preserve the state of the caller
- Pass the parameters to the callee
- Remember the return address
- Transfer control to callee
- Allocate space for callee's local variables
- Receive return value(s) from callee and pass them back to the caller
- Return to the point of the procedure call

Saving Registers

- Question - Can we just throw hardware at this problem?
- Question - Can we do better?

Peace Treaty

- Registers **`s0-s2`** are the caller's s registers
- Registers **`t0-t2`** are the temporary registers
- Registers **`a0-a2`** are the parameter passing registers
- Register **`v0`** is used for return value
- Register **`ra`** is used for return address
- Register **`at`** is used for target address
- Register **`sp`** is used as a stack pointer

Finalized Convention

- Caller saves temp registers
- Caller saves ADDITIONAL parameters
- Caller allocates space for additional return values
- Caller saves previous return address (from current $ra)
- Caller executes JAL $at, $ra
- Callee saves previous $fp, copies contents of $sp into $fp
- Callee saves any s registers it plans to use
- Callee's local variables allocated on the stack
- Prior to return, callee restores any saved s0 to s3 registers
- Upon return, caller restores previous $ra value
- Caller stores return values as desired
- Caller moves $sp to discard additional parameters
- Caller restores any saved temp (t0 - t3) registers from stack

Questions?