CSE 141 Discussion

Hacking the Simulator Infrastructure
Golden Release

- Final writeup details on the website, but basically it’s a combination of everything you have done thus far. (Again, pdf format please)
- Include your source files for your assembler and simulator, as well as your asm code for each program
- Zip it all up and send it to the TA by the deadline
General ISA Issues

• Missing components: The following are things you will NEED to include in your report for the Beta Release (and beyond)
  – Fibonacci EDP Analysis
  – Heavily Commented Code
  – Anything that was in the Design Concept (that wasn’t changed)
  – Detailed Function Call ABI
    • Lots of people gave the caller/callee responsibilities, fewer explained how to make a function call/return from one
General ISA Issues

• `la` vs `li`
  – So up to this point we have stated that `la` takes a label and your assembler translates that label into the appropriate value
  – Some have used `la` to both take in a raw value and a label
  – This is fine...if you let your assembler handle it
    • Otherwise, it will assume something like `0x3DEADBEEF` is a label...you don’t want that.
  – Potential solution, have another pseudo instruction `li` (load immediate) that does all the same things as `la`, but takes a value, not a label
  – Whatever you do, be consistent
General ISA Issues

• Immediate bounds
  – Remember the bounds of your immediates.
    • Don’t try to branch 100 instructions if you only have room for +/- 32
    • Don’t try to branch a value that will fit…but only if the immediate is unsigned, UNLESS this is by design.
      – Example: bne = 0001 000 {00000000} ← immediate (so +/- 64)
      – Bad: bne $r0 100
      – Good: bne $r0 56 or bne $r0 -10
General ISA Issues

• Address Space
  – Remember, you should be able to jump to any instruction across I-Mem and access any data across D-Mem
  – 95% of groups cover the D-Mem case, fewer cover the first.
  – You should have some form of a jr (jump register) instruction that jumps to any address in I-Mem
  – You don’t necessarily have to use it in SG or fib if you don’t need to, but you need to have that functionality
General ISA Issues

• SG Memory
  – Remember how it was said you can put memory wherever you want?
  – Well that’s not necessarily false, BUT if you want to make life easier, try to NOT have a .data section for SG

• Why?
  – Because you will load pre-made D-mem files for the Supergarbage programs, and it will be much easier to work with them if you leave them as is.
General ISA Issues

• Check your report to see if you run into any of these issues, even if the grader didn’t explicitly state you have this problem.
• These issues, if fixed now, will save you a lot of headaches come simulator debugging time.
• As always, come to office hours if you have questions
Simulator

- So you have assembled code, and you have an ISA you’ve been working on for almost a month, now what?
  - You aren’t yet ready to just plug it into hardware
  - Time to simulate!
- Once again, check the website’s simulator section and download the framework
- Think of your simulator as a gdb for your ISA
Simulator Commands

• *iload* $i\_file\_coe$ $start\_addr$
  – loads *.coe* files at $start\_addr$ of the instruction memory
• *dload* $d\_file\_coe$ $start\_addr$
  – loads *.coe* files at $start\_addr$ of the data memory
• *go* $number$
  – simulates next $number$ instructions
• *dump\_reg*
  – prints values in all registers
• *set\_reg* $reg\_num$ $value$
  – sets the register $reg\_num$ with the value $value$
• *dump\_imem* $addr$ $size$
  – disassembles instructions in the instruction memory from $addr$ to $addr + size$, which might not be exactly the same with the assembly instruction appeared in your assembly file
• *set\_imem* $addr$ $value$
  – sets the value at $addr$ of the instruction memory with the value $value$
Simulator Commands

- `dump_dmem $addr $size`
  - prints data memory values from $addr$ to $addr + $size$
- `set_dmem $addr $value`
  - sets the value at $addr$ of the data memory with the value $value$
- `dump_channel $channel`
  - prints values put on channel $channel$ by `out` instruction
- `clear_channel $channel`
  - discards all values written to channel $channel$ so that `out` instruction can proceed
- `put_channel $channel $value`
  - puts $value$ on channel $channel$ so that `in` instruction can get a value
- `set_buf_size $value`
  - changes the buffer size for each channel to $value$. the initial size is 32
- `instr_count`
  - shows the number of executed instructions
Simulator

• Good news! The simulator framework has all of that functionality built in for you.

• So what do you need to do?
  – First, look at the code and understand what those prebuilt functions do.
  – Most are self explanatory
    • initIMem(*), loadMem(*), etc.
Simulator

• Your TODO list:
  – Do a grep, CTRL+F, whatever, for any TODO comment
  – But basically, you need to tell the simulator what your opcode length and register file size are (in the class constructor)...
  – ...set the register that holds your stack pointer to be location 8191 (the memory is 8192 words long in the simulator, find this is the resetSimulator() method)...
  – ...and fill in the execution of your processor (in the execute(int) method).
    • This last one will obviously be the big one.
Simulator

- Dealing with in/out channels
  - Again, this is functionality that is prebuilt into the code.
  - That said, the simulator doesn’t know what values do give channels
  - Remember this?
    - `put_channel $channel $value`
      - puts $value on channel $channel so that in instruction can get a value
Simulator

• Dealing with in/out channels
  – What values do you put in?
  – How does it set up the channels anyway?
    • Linked Blocking Queue, so FIFO
  – In other words, as long as you have space in the queue, you can enter multiple values before executing the program.
  – To make sure the reference output matches your output, use the dump_channel command
Simulator

• Helpful Hints:
  – The lion’s share of the work is in the debugging of your ISA code and the simulator, don’t assume that it will be easy to just plug in the execution and be done.
  – Remember you can step through the program to find particular errors, the $go$ $number$ instruction is your friend.
  – Again, it would probably be good to check your output of your assembler before getting too deep in the simulator, just to make sure that isn’t the problem.
  – START EARLY!