CSE141L Lab 3, Part B: Execution Unit Control and Test

CSE 141L Homepage | Lab Overview | Lab Description

Deadline:	1. Initial Drop (Steps 1-3): See schedule 2. Final Report (All Steps): See schedule
Changes:	1. **May 9 2012** -- Updated for 2012. 2. **May 18 2011 @ 5PM -** Added more details about using COE files to generate new memory modules. Also, a new COE to HEX converter tool is available. It has the same functionality as the python version used in CSE 141 but is written in Java to make it easier to run on the lab computers. You may also want to use the code to automatically generate HEX files in your assembler. For more details, check out Step 3 (Basic Instruction Set Test). 3. **May 19 2011 @ 9:30AM -** Added link to a screenshot showing how to specify memory initialization HEX file in MegaWizard. 4. **May 21 2011 @ 4:00PM -** Fixed typo in top.v involving reset_i signal. Also, added clarification about using io_devices.v file only for simulation.

Lab Overview:

Finally, after many weeks of work, it is time to put the remaining pieces together and finish the implementation of your processor. In the first part of this lab, you created the datapath for your execution. In this part, you will need to implement the control for your processor and test everything to shake out any bugs you might have. This part will require a significant amount of work so you should expect to put in some overtime to make sure it is done on time. The AwesomeCore investors are salivating over the prospects of a great new processor so lets make sure we deliver. Good luck!

Lab Deliverables:

For this part of the lab, you will need to finish up the last major task for your processor: implementing the control for your backend. After finishing up the control, you will be able to perform functional testing of your processor to make sure that it is doing what you expect. For the Final Lab (4) you will have a chance to optimize your processor so do not worry about doing that right now. Just make sure everything works correctly for this lab. We will hold off on interviewing you until you have finished optimizing so there will be no demonstration required for this part.

In order to make the work more manageable, there will be two due dates for this part of the lab. A preliminary progress report with the work from steps 1-3 will be due in one week. A full progress report, including *all* of the steps will be due in two weeks.

- **Demonstration:** None

- **Progress Report:** Once again, there are questions scattered throughout this lab that you should answer for your progress report. For questions that required a screen capture, make sure you fully describe the screenshot. You will also need to include all of the Verilog files from your project (*.v, *.tbw, *.ngc, *.mif, *.coe, *.hex, etc.). Make sure your code is written in a clear and consistant style with liberal use of comments.

You will need to turn in an electronic copy by on the deadline via moodle (note there are 2 deadlines). The answers to the questions should be in a PDF file. The Verilog files (as well as your PDF) should be included in a ZIP file.

Lab Description:

Jump to: Control Logic | Finalize Fetch Unit | Instruction Set Test | IO Test | Write a Simple Loader | VirtualMachine Test | Evaulation

Step 1: Control Logic

Complete your backend by implementing its control logic. In your report write a state transition diagram for your processor showing the states, their transitions and the control signals causing the transitions. Include the source files in an appendix at the end of your progress report.

Step 2: Finalize Front End

If you have not done so already, finalize the implementation of your front end. In order to do this, you will probably want to generate a new instruction memory module with M = 4K or 8K entries of width 17 bits. You will probably need to make sure the front end instantiates a FIFO module of the appropriate width as well (e.g. lg M + 17). 7 or 15 entries in the FIFO should be fine. (As the architect, you should use your discretion.) Make sure you include all the files from these modules in your ZIP. Include your fetch.v in the appendix of your progress report.

Step 2b: Testing Infrastructure

To help in your testing, you should create a testbench code for your design that logs instructions in your final pipeline stage. It should print out the instr #, PC, the instruction word, the register # written, value that was written, and any I/O events that occur. (Often a good idea to record reads and writes to memories; but you want to print the information out in the same cycle that the other info is printed out.) You should be able to modify your Java simulator to print out the same information; you may have to fiddle a little bit it so that things line up (e.g. instr #). You may need to add some registers so that all of this information is available at the same time. We recommend you do the printing all in the last stage of your pipeline so that the order of writes matches that in your simulator (which does not model pipelining). If you print the output the same way, then you can textually difference (e.g. unix diff, or emacs ediff) the files to determine where your C simulator and your RTL diverge.

You will still need to look at the waveforms to debug; but this will help you identify issues more quickly. We recommend you take a look at the .chk files in the C++ fetch unit testbench distrubition (i.e. vector_simple2.tfw) to see the code for opening a file, code for generating a clock, and the code for sampling the inputs to registers just before the negedge of the clk.

Note that you should use the (non-synthesizable) hierarchy traversal shortcuts for accessing state within your (behavioral) verilog; but remember you will have to `ifdef them out to get things to work in timing

simulation;
e.g. for the fetch unit, at the top level, you could have written something like:

```
if (UUT.fifo_enqueue)
 begin
    $fwrite(filedes,"Enque %9.9x %5.5x\n", UUT.pc_prev_r, UUT.ram_data);
 end
```

Step 3: The Rise of the Fibonacci Machine

Overview of Testing Methodology

To run your program, you will need to use your assembler to generate COE files for your instruction and data memories. Because Altera uses a different (HEX) format than you generate with your assembler, you will need to convert these files to the appropriate format. We have provided you with a Java class, hex_converter.java, which automatically converts COE files to HEX format. (Check out this script using SVN: **svn co http://parallel.ucsd.edu/asm-sim-framework/trunk/coe-to-hex**) Run the program with the "-h" option for instructions on how to use it. You may also want to incorporate this code directly into your assembler so that your assembly output is in HEX instead of COE format.

After you have generated the HEX files, use them to create new instruction and data memory modules. You can follow the same steps to create memory modules with *MegaWizard* as you did in Lab 3A but you will additionally need to specify the HEX file that should be used to initialize memory, as shown in this image. Having generated the new memory modules, you can now perform behavioral and timing simulations to verify that your instructions are working correctly. Include the assembly file for your test program in your progress report.

Before you try **timing simulation** you should verify that **behavioral simulation** works.

Synthesis for timing simulation, you will only be synthesizing your core.v with Quartus; top.v and io_device.v will be added to your modelsim project afterwards and will be unsynthesized. Timing simulation can be challenging to debug; both for synthesis and for simulation. For synthesis, the safest thing is to stick to the synthesizable Verilog constructs that we have used in class. Behavioral verilog, using negedge of clock, initial statements, assignments to registers, etc, are all red flags for synthesis problems.

Debugging timing simulation In timing simulation, much of the internals of your core are obfuscated by the optimization process. This makes things very challenging. In many cases, register names are preserved; as long as they were not optimized away. One approach to guarantee you can see certain values is to create a system verilog struct that you route all the way to the top-level of your design. Then you can add and remove wires for debugging without writing code to route them through your module hierarchy. See this file for some tips on System Verilog structs. If wires are part of the interface of your top-level module in synthesis, then they will not be optimized away since they are legitimate outputs. (Keep in mind you may only be able to route a small number of signals to top level because of pin limitations.)
It often pays to identify "risky" elements of your design from a timing simulation point of view. Then write smaller unit tests that just test those particular components (e.g. 12 ported register file) or constructs (e.g. structs; unfamiliar syntax.) Testing smaller portions of your design that present risk of not synthesizing correctly, is often a time-saver even if you don't find a bug, because now you know with much more certainty where *the bug is not located.*

Step 3a: Basic Instruction Test

To verify that you have basic functionality enabled, you should show us that all of the instructions in your ISA are operating correctly. To do this, you should be able to reuse the test file you created in the 141 Project; however now that you have more insight, you may have ideas about how to modify it. This should be an assembly file that uses *all* of the instructions in your ISA *except* the *in* and *out* instructions (these instructions will be tested next).

Q1	What was your strategy for testing all your instructions? Is there a way to quickly verify that all of the instructions executed correctly?
Q2	Include screen captures from at least 3 points in your behavioral simulation. One of these must be for the end of your program but the rest of the points are of your choosing. Be sure to explain what each of the screen captures is showing. Also, make sure the captures are zoomed in enough to see actual values and that the labels for the signals are visible in the capture.
Q3	Include screen captures for at least 3 points in your timing simulation. They can be from the same points where you captured for behavioral simulation in Q2. For output, you may want to wire the register file write inputs to the top level via a struct.

Step 3b: Preventative Testing

Any instructions which you are not confident about working, you should write small tests for, and compare between your ISA simulator and your RTL. It's easier to debug small programs than large ones, so before you make the jump to Fib, you should make sure you have no known issues. Keep these and other tests around in a structured directory hierarchy (called a regression suite) so you can rerun them if necessary.

Step 3c: Fibonacci

Now is the time to test and debug your machine on Fibonacci. Use your handy reference traces from your simulator, and debug behavior and then timing simulation. As you "fix" your design, you may want to rerun your prior instruction tests to verify that you have not regressed. *For testing, I recommend you start with test cases that target the simpler specific cases in fib, and then move to the more advanced cases. I.e. -1,1,2,29,30,48,49,3,4,... Also, for timing simulation, keep the recursion depth below 5, because it runs very slowly.*

Q4	Include screen captures from at least 3 different runs of fibonacci with different inputs in your behavioral simulation. Typically these must be for the end of your program, so you can see the output. Also, make sure the captures are zoomed in enough to see actual values and that the labels for the signals are visible in the capture.
Q5	Include screen captures for the corresponding timing simulations. For output, you may want to wire the register file write inputs to the top level via a struct.

Important Note:

Everything up to this point should be part of the preliminary report. The preliminary report report will typically not be graded directly; but will be used by us to determine if you are actively working on your project and making progress towards the final goal. If a few of your instructions still have issues, this is ok. If you have

time, continue on SuperGarbage and turn that in as part of the preliminary submission. The following steps are only necessary for the final report.

Step 4: IO Test

In your 141 ISA project, you were provided the code for a VirtualMachine function. While the data for your implementation of the VirtualMachine function resides in your processor, the programs that your VM will run (those written in the SuperGarbage ISA) are not on chip. To access them, you will need to use the processor's IO interface to talk to an external device. We have provided you with io_devices.v, a Verilog module for an IO device that supplied SuperGarbage programs to your processor. Also include in this module is a clock counter that will allow us to measure your performance when executing a SuperGarbage program. Please be aware that the provided IO devices module is only meant for behavioral simulation. It contains Verilog that is not synthesizable so if you try to compile it using Quartus, you will receive errors.

You also have access to top.v which is a top level module that includes your processor core and the IO devices. Note that you will need to modify top.v if you did not use the exact core interface given in part 1 of this lab.

We recommend that you extend your simulator to support this interface, so that you can generate reference traces to aid in the debugging of your RTL.

The interface of the *io_devices* module is given below:

```
module io_devices#(parameter D_WIDTH = 34, PA_WIDTH = 4)
(
        input   reset_i,
        input   clk,

        input   read_req_i,
        input   write_req_i,
        input   [PA_WIDTH-1 : 0] read_addr_i,
        input   [PA_WIDTH-1 : 0] write_addr_i,
        input   [D_WIDTH-1 : 0] din_i,
        output  [D_WIDTH-1 : 0] dout_o,
        output  read_ack_o,
        output  write_ack_o
);
```

The *io_devices* module provides the following services via I/O channels:

Input Channels:

- #1 : Request the next word of SuperGarbage binary file
- #2 : Request the current clock count
- #3 : Request an input data from "input.txt" file
- others : Reserved

Output Channels

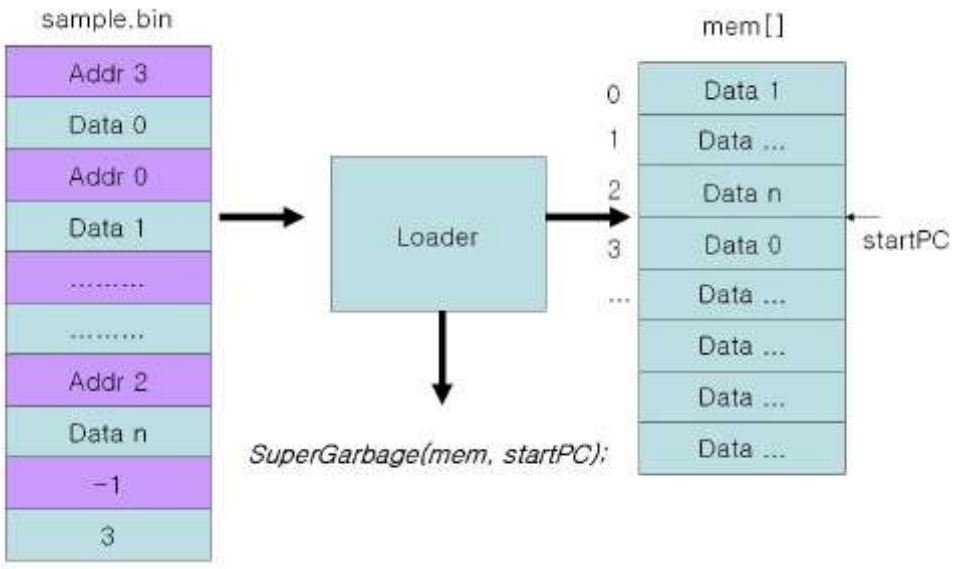
- #1 : Select a SuperGarbage binary file
- #2 : Set clock count
- #3 : Debug output
- others : Reserved

Now that you have an IO device to interact with, you should write a small assembly program to test your *in* and *out* instructions. Include this assembly program as part of the appendices of your progress report.

Q6	Explain how the setup of your IO test. Get a screen capture of the behavioral simulation of this test and explain the results.

Step 5: Write a Simple Loader

To run a SuperGarbage program using your VM, your processor first needs to properly load the application into the data memory. The following figure shows the format of the SuperGarbage binary file and how a loader works. After selecting the application to load, the loader requests the selected SuperGarbage binary file via input channel #1, word-by- word. The odd-numbered words contain the address where the following word will be stored. For example, in the figure below, 'Data 0' will be loaded into address 3 in your data memory. The loader continues loading data until it encounters the address *-1* (0x3FFFFFFF), which means the end of the file. The word following address *-1* is the starting PC for the SuperGarbage program. After the loader is finished, it should call the VirtualMachine function you have written.



The following Java-esque pseudo-code shows how to load a program from the *io_devices* module and run the VirtualMachine. We have provided several SuperGarbage applications for you to test on. Changing the *num_app* variable in the pseudo-code will change which of these applications you load. You should write the loader in your assembly language, build the COE files, convert them to HEX, and generate a memory module. Be sure to include the assembly code for your loader in an appendix of your progress report. Note that get_instr_count is a macro that translates into whatever assembly instructions are needed to access a counter that counts the number of instructions executed in your processor since reset (you will add this functionality to your processor; see below).

```
// SuperGarbage Loader
// num _app
// 0: app0.bin
// 1: app1.bin
// 2: app2.bin

final int num_app = 0;

word mem[4096];
word startPC;

// basic code stub
set $SP;
```

```
set $GP;

// select an app
out( num_app, 1);

// load data from an external device while(addr != -1)
do {
    word addr, data;
    addr = in(0x1);
    data = in(0x1);

    if (addr == 0x3FFFFFFFL) {
        startPC = data;
        break;

    } else {
        mem[addr] = data;
    }
} while(true);

// Finally, call the virtual machine
start = get_instr_count();
VirtualMachine(startPC, mem);
end = get_instr_count();
out (0x3,end - start);
```

Step 6: VirtualMachine Test

In this lab, you will use the following three SuperGarbage applications to test your design. You will need to put them in your project directory. Applications get inputs from input channel #3, which is fed by the input file. If you want to change the input data, modify the input file for the corresponding application.

Benchmark	Assembly File	Binary File	Input File	Counter Input File
Compare	app0.s	app0.bin	app0.in	app0.cnt
Fibonacci	app1.s	app1.bin	app1.in	app1.cnt
Bubble Sort	app2.s	app2.bin	app2.in	app2.cnt

You should now run your loaded with the app0 (Compare) to make sure your loader is working correct.

Q7	Perform a behavioral simulation of your loader using the app0. Get a screencapture of the integer that is returned from the VirtualMachine function. What was this integer?

SuperGarbageSim

Although the VM is very simple, it is not easy to understand SuperGarbage code. To help you understand the behavior of SuperGarbage applications, you are provided with a SuperGarbageSim, a reference simulator for SuperGarbage applications. SuperGarbageSim can load and execute any SuperGarbage application binary file. You can also monitor or modify contents of memory and the program counter. The SuperGarbageSim supports the following commands:

- *load $file.bin*
  loads *.bin file and *.in file, and set the $PC to the entry point of the application

- *go $number*
  executes next *$number* instructions. if $number is not specified, continues execution until a 'halt' instruction is executed.

- *step*
  execute one instruction at the current $PC, and increase $PC.

- *get_pc*
  prints the current $PC value

- *set_pc $value*
  sets the $PC to *$value*

- *disasm $addr $size*
  disassembles instructions from *$addr* to *$addr* + *$size*

- *get_mem $addr $size*
  prints data memory values from *$addr* to *$addr* + *$size*

- *set_mem $addr $value*
  sets the value at *$addr* of the memory with the value *$value*

- *count*
  shows the number of executed SG instructions

- *help*
  shows a help message

The following files comprise the SuperGarbage simulator. After downloading and compiling them, you can start the simulator by running the command 'java SuperGarbageSim'.

1. Instruction.java
2. SuperGarbageSim.java

Here are a few examples of SuperGarbageSim commands:

```
prompt> load test1.bin      // load 'test1.bin' file
prompt> disasm              // disassembles 10 instructions from PC
prompt> disasm 0x10 15      // disassembles 15 instructions from 0x10
prompt> set_mem 14 0x0      // set the memory location 14(0xe) to 0x0
```

Q8	With the aid of SuperGarbageSim, figure out what the final PC of app0 should be. Explain how you got this number. How does it compare to the integer that was returned in your screen capture as part of Q4?

VirtualMachine Performance

All three provided SuperGarbage applications measure their execution times by using the cycle counter service implemented in *io_devices.v*. They calculate the execution time by requesting the cycle counter value at the beginning and end of the execution and getting the difference. They then send the number of cycles for the execution to the debug output channel (output #3) so that you can easily see it.

In order to count the number of instructions executed on your processor, you should add an instruction counter to your processor (this was briefly referred to in step 5). It should be initialized to zero when your processor is reset and incremented after every successful execution of an instruction. Make sure to only count instructions that have committed; i.e. since some instructions may be squashed after fetch.

Note: dmem.v occassionally refuses a request to the memory (for debugging purposes). However, for testing of the applications, we do not want this to happen. We have therefore provided dmem_no_refuse.v, which does not refuse requests. When doing performance analysis, use dmem_no_refuse.v rather than the old dmem.v.

Q9	Run all the benchmark applications and gather the following information for each benchmark: 1. Number of execution cycles for the application (using io_device.c cycle counter. 2. Dynamic instruction count for your processor, excluding the time for the loader (using your processor's instruction counter and your get_instr_count() macro as detailed in Step 5; or less ideally, your simulator). 3. The Cycles Per Instruction (CPI). Present this information using a simple table.
Q10	Why does CPI vary across applications?

Step 7: Evaluation

Now that you have passed two important tests, let's evaluate the performance of your processor. Perform a post-route simulation with the testbench you wrote in Step 3 (Instruction Set Test).

Q11	What is the maximum achievable frequency of your processor? Include a screen capture on the post-route simulation result at that frequency in your hardcopy report. Using this frequency, calculate the execution time for all three benchmarks.
Q12	What is the critical path of your processor? Draw the critical path on the datapath schematic you made in part 1 of this lab.
Q13	Propose one idea for increasing the performance of your processor. Why do you think it is an effective way to boost the performance?

This lab created by Donghwan Jeon, 2007. Modified by Sat Garcia, 2008.