Lab5: Additional Arithmetic for RISC240

Objective and Overview

The purpose of Lab 5 is to give you a chance to explore the microarchitectural features of a simple microprocessor. This lab will build on the knowledge gained in lecture and carry it further – to a complete synthesized implementation of RISC240 in SystemVerilog.

Schedule and Scoring

Lab 5 is a two-week lab effort. The last week of the lab falls in the last week of classes, so the write-up requirements for this lab will be skipped. As a result, no late demonstrations will be accepted. The final lab's check-off must be demonstrated to your TA during your lab period in week 2 (or earlier).

We would recommend that you consider Task 1 to be a pre-lab and have it finished at the beginning of Week 1, though we will not be requiring it. Be careful not to leave too much to be accomplished during the last week.

18 – 20 April	Complete Task 1. You should probably also complete Task 2.
25 – 27 April	Complete Task 3. Demo all tasks by end of lab.

Note: **A 3 point deduction will be made** each time that you unsuccessfully demonstrate your system for any of these tasks. Make sure to thoroughly test, and assure yourselves that you are prepared, before you approach a TA for a demo.

Lab 5 is worth 120 points:

- **25 points:** Demonstrate Task 1. Simulation results of your ADD32 benchmark using sim240, showing that the benchmark is correct and can correctly perform a series of test additions as requested by a TA.
- **5 points:** Documentation for Task 1. Complete questions 1 and 2. This will be checked during your Task 1 demo.
- **30 points:** Demonstrate Task 2. Show a TA that your modified RISC240, with datapath and FSM modifications to support the `ADD32` instruction, works properly. This includes error-free synthesis of the SystemVerilog, and a discussion with the TA of all of the changes you have made to the datapath and FSM.

- **10 points:** Documentation for Task 2. You should have a diagram of the processor datapath, clearly showing all modifications that you make to implement the **`ADD32`** instruction, and a diagram of the FSM states that you need to add to make use of the modified datapath components.

- **10 points:** Questions for Task 2. Complete questions 3-7. This will be checked during your Task 2 demo.

- **40 points:** Demonstrate Task 3. Demonstration to a TA of a complete synthesized system, including I/O. Both lab partners must be ready to answer questions on all aspects of the lab.

- **1 bonus point:** Show that you have submitted the TA feedback form.

No lab report is required for this lab. However, we will check documents as noted for each task during the various demos. The TAs will ~~probably~~ refuse to help you find bugs if your documents aren't up-to-date.

Late Penalties (the not-so-standard drill)

No late work will be accepted. Get something in on time, else you will receive a zero.

A Note about Teams and Collaboration

Lab 5 will be accomplished in teams. Continue to be a good teammate, communicate well, and work well. TAs have full authority to modify a lab partner's grade downward (yes, as far down as zero points) for ANY situation where they suspect an imbalance of effort among team members. In plain language: If you're coasting on the work of your partner, your grade will pay for it.

Once again, all work you turn in for this lab is expected to be that of your team. You may ask other students for general assistance, but you may not copy their work. Likewise, you may not copy work from other sources.

The RISC240 Processor

Tim Cook has hired you as a consultant to assist with the design of a new processor. Intended to be an ActualReality® processor, it should work well as a companion to the recently created M18 chip. The processor looks suspiciously like the RISC240 CPU, which you know so well. It is scheduled to go into production on 15 May 2023, but Mr. Cook has a few concerns. As a result, he has asked you to investigate the efficiency of the new microprocessor, in particular, the speed of execution of arithmetic operations.

When the RISC240 was designed, not enough thought was given to high-speed computing techniques, resulting in only simple instructions being implemented. More complex mathematical operations were expected to be constructed with those basic instructions on the existing architecture. How well would those work? Mr. Cook would like you to find out.

As you've learned in class, the RISC240 has a small number of instructions, which requires us to perform more complex operations using a series of these instructions. One example of a complex

operation is *32-bit addition*, which needs to be performed using a series of 16-bit ADD instructions. There are times when a complex operation is used often enough for it to eventually be included as a new instruction in the ISA. However, adding a new instruction should not be done lightly, as this requires several modifications (and requires updating the contract between the programmer and the hardware architects). As a result, a good designer should first evaluate the benefits that a new instruction would provide, to determine if the benefits outweigh the costs. Your job in this lab will be to determine how (in)efficient 32-bit addition currently is in the RISC240, and to make modifications to the RISC240 design in order to more efficiently perform multiplication.

We've provided a SystemVerilog description of the RISC240 architecture. It is a fairly straightforward implementation of the elements described in class. The processor has an FSM that controls the datapath. The datapath contains a variety of components, including an arithmetic logic unit, a register file, and multiplexers. You can find the source files in **`/afs/ece/class/ece240/handout/lab5`**. For a description of each file, see Section 10 in the RISC240 Reference Manual.

Task 1: Benchmarking 32-bit Addition in the Unmodified RISC240 Processor

You will need to write an assembly program that will serve as a benchmark to measure the current performance of addition in the unmodified RISC240 processor. Your benchmark will sum all of the 32-bit values in an array, storing the result to a 32-bit location in memory.

Here are the specifications for your benchmark. Please adhere strictly to them, as that will make things easier for a TA to assist you (also, to grade and evaluate).

- An array of 32-bit values is stored in memory at label **`ARRAY`**. You should ensure the array can be placed anywhere in memory.
- **`ARRAY`** consists of 32-bit values. Each value is stored such that bits [31:16] are in the lower addressed word (the word with the smaller address).
- The length of the array is specified in a 16-bit value stored in memory at label **`LENGTH`**. Your benchmark should properly handle any possible value of **`LENGTH`**, though we won't test with any value larger than **`$0100`**.
- After your benchmark has computed the sum of all the values in the array, store the result in memory at label **`SUM`**. Sum is also a 32-bit value, stored such that the higher-order bits are stored at the lower addressed word (the same way the values in the array are stored). Note that only 32-bits of the sum is stored. If the result is greater than 32-bits in size, those extra bits will be lost.
- Your code should begin at location **`$0010`**.
- Your code should use the instruction **`STOP`** to finalize computation.

You should use the software tools that we have already provided (i.e., **`as240`** and **`sim240`**) to ensure that your code successfully assembles and correctly performs 32-bit addition.

Now that you have a working benchmark, your next job is to synthesize the RISC240 processor in order to measure some of the characteristics of this benchmark program.

1. Get the source files from **`/afs/ece/class/ece240/handout/lab5`** (again, see the Reference Manual for more details.)
2. Make a project, in Quartus, to include all the files you retrieved. Also, include the **`memory.hex`** that you generated with asm240.
3. Synthesize, place and route the processor design. Record the size and speed of the design.

At this point, you are not actually executing the benchmark code on your FPGA board. Be patient, we will get there.

For Credit: Once you have **thoroughly** tested your code, demonstrate it to the TA. The TA will have several arrays for you to use as test cases, but it's best for you to think of different test cases and test them *before* your demo.

At the demo, submit the answers to the following questions to the TA, *in writing*.

1. Does the timing of your program depend on the particular values in the array? Why or why not?
2. Calculate the maximum and minimum clock periods your program could take, as a function of the length of the list. If your answer to the previous question is "No," these values should be the same.

*Note: You can do all of this task, except for the demo, before you come into lab. However, if you do, you **must** work with your partner on **all** of it.*

Task 2: Hardware Modifications for 32-bit Addition

Tim Cook was quite disappointed with your evaluation of the benchmark code (if you are reading this prior to completing the memo, don't be worried. Mr. Cook has enough CEO experience to allow him to anticipate technology trends). He would like you to make design modifications to the RISC240 to speed up the 32-bit addition task, as well as to correct the flag situation. If you get the modifications done before the end of your 2nd week lab period, he will be most pleased. [1]

Mr. Cook would like you to add a new instruction to the RISC240 to perform 32-bit addition at high speed (or as close as you can come). The original team of engineers has already generated an excellent design, so you shouldn't need to remove any existing functionality, nor change any of the existing datapath connections. However, you are authorized to move wires, add control outputs to the FSM, add states to the FSM and add control points and functionality to the ALU.

The New Instruction: Add the ADD32 instruction to the RISC240. **`ADD32 Rd, Rs1, Rs2`** will add Rs1+1, Rs1 to Rs2+1, Rs2 with the result left in Rd+1, Rd. Rd+1 means the register whose number is one more than Rd (in a mod 8 sense. If Rd is 7, then Rd+1 is R0). Ditto for Rs1+1 and Rs2+1.

It is possible that the destination registers are also source registers (such as **`ADD32 r3, r2, r6`**). In such cases, the results are allowed to be incorrect. The instruction's addressing mode is register, and the encoding is:

ADD32 rd, rs1, rs2

32-bit Addition

Semantics: `rd+1,rd ← rs1+1,rs1 + rs2+1,rs2`

CC Flags: `ZNCV - set normally`

Encoding:

15 9	8 6	5 3	2 0
0110 010	rd	rs1	rs2

Cycles: ?

Specifics: You will need to make a fair number of changes to the SystemVerilog description of the processor. This is not something that you can walk into lab during the second week and figure out. Come prepared with a SystemVerilog description that is ready to go. Simulate it before coming to lab. Check for synthesis errors before coming to lab. Work early and avoid Finals Fever! No late demos!

You will need to change the following parts:

- **FSM:** After all, you're adding new instructions here. You will have to specify the control states needed to do the new functions.
- **Datapath:** Some new register selection hardware will be needed for the register file — e.g. muxes to provide different inputs to the register file. Perhaps a different register specification decoder.
- **Oh my!** You'll need to modify the FSM again to control the new register selection hardware.

[1] If you don't get it done, Mr. Cook will not be pleased. Neither will you.

- **ALU:** You now need to be able to execute an addition operation using the carry flag in order to add the upper 16-bit values. Or do you? Don't fear the design choices.

Change as little as possible in the descriptions (be lazy!). Mr. Tim Cook will be very disappointed if you unnecessarily change other hardware or if you remove any functionality in the system. Of course, your revised description must remain synthesizable.

Synthesize your new design and note the speed and size of the new design. Ensure that there are no errors, especially any "inferred latches" in your design.

Mr. Cook, while not a microprocessor architect, does have an excellent sense for navigating tricky timing situations. He cautions you that you will need to be careful about when the outputs of the various modules are available.

Revise Your Benchmark: Now that you have made these changes, you need to write and assemble another program which takes advantage of your design modifications. Change the previous program to make use of the **`ADD32`** instruction.

Unfortunately, as240 doesn't support your new instructions. However, if you think carefully about the assembler pseudo-operations a bit, you'll probably come up with a way to include your instruction in the assembly file. Note that the TAs frown on editing the **`memory.mif`** file as an ugly hack. We are only looking for beautiful hacks here!

For Credit: Add answers to the following questions to your list from Task 1.

3. How many logic elements (i.e. ALMs) and registers does your design consume? (you learned how to find this information in Lab4.)
4. What is the maximum clock frequency (F_{max}) of your design? Again, you learned where to look up this number in Lab4, Task 4).
5. How many clock cycles does your ADD32 take to execute? Include the fetch and decode phases.
6. For an array of length L, what is the maximum and minimum number of clock cycles your revised benchmark will take to execute.
7. What percentage speedup did you achieve?
8. What sort of changes would be required if you had to ensure correct results when the source and destination registers overlapped? I'm not asking for an entire design, just an answer that shows you've thought through the problem.

Demonstrate to the TA your simulation results (sim240) and that your new design is synthesizable. Both teammates should be ready to answer questions about the changes you made to the datapath, FSM and to your benchmark.

Task 3: Running Your Modified Processor on the FPGA

The true test of your work will be shown when you get the RISC240 operating on the Altera FPGA board — along with your **ADD32** instruction. The SystemVerilog code you've been given will allow synthesis, with a few caveats:

- At the top of **constants.sv**, there is a comment (**\\`define synthesis**) **that you must uncomment before synthesizing**. Make sure to re-comment it for any simulation tasks.
- Your machine code needs to be included in the synthesis run. The as240 assembler outputs a file named **memory.mif**, which is a memory image of your program. Place it in your project directory so Quartus II can find it during synthesis.
- Your program should start at address **$0000**.
- You have no method for getting input to your program, nor receiving output from it. Fix this by creating two *memory-mapped* I/O ports.
  At memory address **$4000**, hook up a register, whose outputs will be displayed on **LEDR[15:0]**. That is, the register should only be enabled (for a load) when the address bus matches the memory address assigned (i.e. **$4000**). Your program can then write a value to **$4000** with the **SW** instruction to get it to display on the LEDs.
- Similarly, connect any read from memory address **$3000** to **SW[15:0]**, for input. Be careful here. Can you just connect the switches directly to the data bus?
- The memory should ignore any access to addresses **$3000** or **$4000**.

Revise Your Benchmark: Update your benchmark code to use the ADD32 instruction and the I/O capability that you've added. First, remove the several instructions that accomplished the addition and replace them with the **ADD32** instruction. Again, since the assembler does not support this instruction, you will have to use the same technique to add it to your code as you did for the simulation task.

Then, use the switch input (i.e. an LW from address **$3000**) as the **LENGTH** parameter. This change should be as easy as updating a line with a **.DW** to use **.EQU** instead (and changing the operand).

Next, output the result to the LEDs, instead of storing it in memory. This change should be as easy as the previous one.

Finally, put the entire benchmark into a loop. You should continually read the **LENGTH** input, calculate the sum of that number of 32-bit values in the array, output the **SUM** to the LEDs and then repeat. This will allow you to run multiple tests without having to re-synthesize and re-program the FPGA. Note that you will always be summing values from the same array, as we don't have enough I/O to input a whole array.

RISC240 User Interface: Unless you change it (which you can do in **`RISC240.sv`**, if you like), the user interface to the board is the following:

- **`KEY[0]`** is clock
- **`KEY[1]`** is **`reset_L`**
- **`SW[17:16]`** control what values show up on the seven-segment displays:
  - When 2'b00, **`HEX7,6,5,4`** will show PC and **`HEX3,2,1,0`** will show **`IR`**
  - When 2'b01, **`MAR`** and **`MDR`** will be shown.
  - When 2'b10, **`r4`** and **`r3`** will be shown.
  - When 2'b11, **`r2`** and **`r1`** will be shown.
- **`LEDR17`** shows **`~re_L`** (in other words, lights up when reading).
- **`LEDR16`** lights up when writing.

For Credit: Mr. Cook is a very busy executive. Because of the critical marketing campaign for zero-button mice and triangular phone devices that is underway, he won't need a report about your work. He has deputized the TAs to discuss your work with you and to watch your demo of a completely synthesized system with I/O capabilities.

You should be able to point out all the changes you have made to the RISC240 and describe why those changes occurred. Have a sketch of your changes on top of the datapath schematic (from Lecture 20) to illustrate. Your TA will also ask to see your synthesized demo. Be ready to answer the following questions:

- Compare how many clock ticks the benchmark code takes with both designs.
- How many logic elements were added to the hardware to create the **`ADD32`** instruction? Express this also as a percentage increase.
- Describe what you did to put the **`ADD32`** and instruction into your assembly language so that as240 would properly handle it.
- Which design is better? Why?