18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

Lab5: A Multiplication Coprocessor 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).

There are 5 tasks to Lab 5, but we will not provide a specific schedule of when they must be accomplished — you are responsible for your own schedule. Here's a suggestion:

28 – 30 November	Complete tasks 1-4.
5 – 7 December	Task 5. 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:

- **15 points:** Demonstrate Task 1. Simulation results of your multiplication benchmark using sim240, showing that the benchmark is correct and can correctly perform a series of test multiplications as requested by a TA. Written answers to the questions posed in the lab handout for this task.

- **15 points:** Demonstrate Task 2. Show a TA that your multiplier coprocessor can be successfully simulated. Demonstrate operation with some very well-chosen test cases using a testbench.

- **30 points:** Demonstrate Task 3. Show a TA that your assembly code from Task 1 executes properly on the RISC240 hardware (in simulation on VCS). Also, show that you have properly synthesized using Quartus.

- **30 points:** Demonstrate Task 4. Show a TA that your modified RISC240, with datapath and FSM modifications to support the `MUL` and `MULI` instructions, works properly. This includes error-free simulation of the SystemVerilog, and a discussion with the TA of all of the changes you have made to the datapath and FSM. Be able to run your Task 1 assembly code, with modifications for the new instructions.

- **10 points:** Documentation for Task 4. You should have a diagram of the processor datapath, clearly showing all modifications that you make to implement the `MUL` and `MULI` instructions, and a diagram of the FSM states that you need to add to make use of the modified datapath components.

- **20 points:** Demonstrate Task 5. 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.

No lab report is required for this lab. Neither will we collect your code. 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

For this lab, you will investigate the RISC240 microarchitecture, and modify the microarchitecture to support two new instructions that perform an 8-bit multiply operation.

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 such a complex operation is *multiplication*, which needs to be performed using a series of shift and 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 multiplication currently is in the RISC240, and to design new hardware that can 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.

Your Design Problem

Task 1: Benchmarking Multiplication 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 multiplication in the RISC240 processor. In order to make things easier to grade and evaluate, we will insist on the following register conventions for your program:

- **`r1`** holds the first 8-bit factor to be multiplied, sign-extended to 16-bits.
- **`r2`** holds the second 8-bit factor to be multiplied, also sign-extended.
- **`r3`** holds the 16-bit product of the two 8-bit factors.

You cannot use these registers for anything else at any point in your program. You are free to use registers **`r4`**–**`r7`** however you please.

Using this convention, your program should perform the operation **`r3`** = **`r1`** $\times$ **`r2`**. More specifically, your program should do the following:

- Keep a 16-bit word at memory address **`$3000`** that holds factor 1, and another 16-bit word at memory address **`$3002`** that holds factor 2.
- The instructions should start at memory address **`$200`**.
- Load the two factors from memory into **`r1`** and **`r2`**.
- Perform the multiplication. We recommend that you think about how to use loops, as this will help you think about how to design the multiplier hardware in Task 2.
- Make sure the final result for the multiplication is in **`r3`**, and is written to memory address **`$4000`**, when your program stops.

One thing to note: while your factors can be held in 8 bits, everything in the RISC240 processor is performed at a 16-bit granularity. Your program must perform **signed** multiplication, so you need to make sure of a couple of things:

- The factors must fall with the range [-128, 127] (inclusive). Your program does not need to (and should not) check the validity of the factors.
- The product in `r3` must be correctly stored as a 16-bit two's complement number.

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 multiplication.

When you demo your work to the TA for this task, you should have answered the following questions in written form for the TA to review:

1. What is the biggest possible magnitude (i.e. absolute value) of the product?
2. How many bits are necessary to guarantee that the resulting product will be represented properly?
3. Does this fit within the RISC240's data formats?
4. Does the timing of your multiplication benchmark depend on the particular values being multiplied? Why or why not?
5. If the answer to the previous question was "Yes", determine the maximum and minimum number of clock ticks the algorithm could possibly take. If "No", determine the number of clock ticks it will always take.
6. Do you think a multiplication coprocessor will reduce the number of clock ticks needed for multiplication in the RISC240 system? Why or why not? Please be very specific.

Multiplication Algorithm: For this lab, we are going to have you implement a simple version of the multiplication algorithm.[1] Let's look at calculating $C = A \times B$, where A and B are both 8 bits. You can perform the multiplication using the following algorithm:

1. Clear the product (i.e. the result register).
2. Determine the absolute value of B (we'll call this B_{abs}).
3. Check the LSB of B_{abs}. If it's a 1, add A to the product.
4. Shift B_{abs} right by one bit, and A left by one bit.
5. Repeat Steps 3 and 4 until you have checked all 8 bits of B_{abs}.
6. If B was originally negative, negate the product.

[1] There are many algorithms for multiplication that are more efficient than what we describe here. If you are interested, you can check out the Wikipedia article on Booth's multiplication algorithm. However, while Booth's is a cool algorithm, we caution you against implementing it for this lab, given that it is more complex, and you cannot submit your work late.

One thing to keep in mind: if $B = -128$, you will have to make sure that $B_{abs} = 128$. Along the same lines ensure that zero is handled properly. Also, in Step 5, you can end early if $B_{abs} = 0$, but it's up to you if you want to implement this check or not.

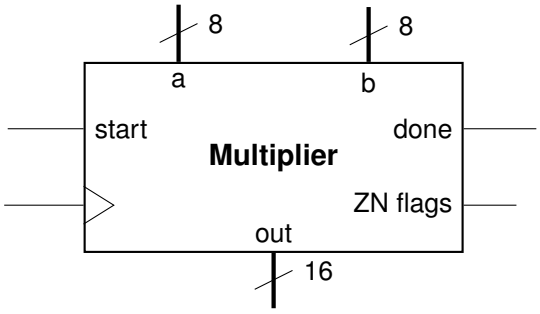
For Credit: Once you have **thoroughly** tested your code, demonstrate it to the TA. The TA will ask you to perform a number of different multiplication operations, so it's best for you to think of different test cases and test them *before* your demo.

*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: Build a Multiplication Coprocessor

It seems that performing multiplication using the existing RISC240 ISA and microarchitecture eats up a lot of processing time. Let's make some design modifications to reduce this overhead. To start, we will build a new multiplication *coprocessor*. A coprocessor is a hardware unit, separate from the ALU, that can quickly perform certain operations. Many popular processors over the years have included different types of coprocessors (e.g., floating-point units, GPUs, vector processors, neural network accelerators). One advantage of a coprocessor is that it can be made optional: some processors can include a coprocessor and provide additional functionality, while other processors can omit the coprocessor to reduce the price (and in some cases the energy consumption) of the overall processor.

For this task, you will build the coprocessor. You don't need to touch the RISC240 SystemVerilog yet. Your job in this task is to build the coprocessor using the datapath components that you have available in your `library.sv` file. The coprocessor should take in two 8-bit inputs (the factors), perform signed multiplication, and output a 16-bit product.



Since the coprocessor will be added to the processor datapath, it needs to have control points and status points, so that the processor's FSM can control the coprocessor. Your coprocessor should include the following control/status points:

- **`start`**: a one-bit active-high control point that tells the coprocessor to start performing the multiplication.
- **done**: a one-bit active-high status point that the coprocessor uses to tell the FSM that it is done performing the multiplication. Whenever **done** is asserted, the correct 16-bit product must be on the output lines of the coprocessor.

- **`ZN flags`**: two bits that tell you if the result was zero or negative. They are asserted only when **`done`** is asserted.

Multiplication Algorithm: Use the same algorithm that we describe in Task 1, but this time make sure that your coprocessor can perform the entire algorithm in hardware. This means that if you have to keep data in the middle of the calculation, you likely need some datapath component(s) to hold this data.

For Credit: You should write a testbench for the multiplication coprocessor, and ensure that it correctly performs signed multiplication for a number of test cases (you can use the same test cases as you did for Task 1). When you are confident that your coprocessor is working correctly, demonstrate it to a TA.

Task 3: Run Your Benchmark on the Unmodified RISC240 Processor

Your next task is to synthesize, place, and route the RISC240 processor:

- Get the SystemVerilog files from **`/afs/ece/class/ece240/handout/lab5`**
- Simulate the processor executing your benchmark code (with VCS) and record the number of cycles that the processor takes.
- Make a project, to include all of the SystemVerilog files. Assign pins to outputs. Include the memory image of your assembly code benchmark (i.e., a **`memory.mif`** file).
- Synthesize, place and route the processor design. Record the size of the design, measured in logic elements and memory bits. At the top of **`constants.sv`**, there is a comment (**`//`define synthesis`**) **that you must uncomment before synthesizing**. (TAs will deduct one point from your score if they have to remind you of this fact.)

For Credit: Once you have completed the simulation successfully, show the simulation results to a TA.

Task 4: Add the Multiplication Coprocessor to the RISC240 Processor

In order to make use of the new coprocessor that you designed in Task 3, you must introduce two new instructions into the ISA: **`MUL`** and **`MULI`**. The instructions have the following requirements:

MUL rd, rs1, rs2
Multiplication

Semantics: `rd ← rs1[7:0] × rs2[7:0]`
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Encoding:

15 9	8 6	5 3	2 0
0110 000	rd	rs1	rs2

Cycles: ?

MULI rd, rs1, imm
Multiply Immediate

Semantics: `rd ← rs1[7:0] × imm[7:0]`
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Encoding:

15 9	8 6	5 3	2 0
0110 001	rd	rs1	000
imm			

Cycles: ?

Modifying the RISC240 SystemVerilog: You will need to change the following parts of the SystemVerilog code to add your coprocessor:

- **FSM:** After all, you're adding new instructions here. You will have to specify the control states needed to do the new instructions.
- **Datapath:** You'll need to add your multiplication coprocessor, along with hardware to select what inputs the coprocessor reads and whether you read the output of the ALU or the coprocessor.
- **FSM (again):** Oh my! You'll need to modify the FSM again to control the new selection hardware.

Change as little as possible in the descriptions (be lazy!). The TAs 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.

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 both of the previous programs to make use of the **`MUL`** and **`MULI`** instructions.

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.hex`** file as an ugly hack. We are only looking for beautiful hacks here!

For Credit: Once again, we've decided to minimize the written requirements for this lab. We have deputized the TAs to discuss your work with you and to watch your demo. 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 demo. Both lab partners must be ready to discuss the entire design and to answer the following questions:

- Compare how many clock ticks the benchmark code (both versions) takes with both designs.
- Make an estimate of how much extra hardware would be required, were we able to synthesize your new design onto the FPGA. Express this estimate as a percentage increase and be ready to justify this value.
- Which design is better? Why?
- Describe what you did to put the **`MUL`** and **`MULI`** instructions into your assembly language so that as240 would properly handle them.

Task 5: 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 **`MUL`** and **`MULI`** instructions. 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.
- You have no method for getting input to your program, nor receiving output from it. Fix this by creating three *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`** to get it to display on the LEDs. Similarly, connect any read from memory address **`$3000`** to **`SW[15:8]`**, and any read from memory address **`$3002`** to **`SW[7:0]`**, for input. The memory should ignore writes to addresses **`$3000`** and **`$3002`**. Note that you are inputting 8 bits into a 16-bit memory location, so you need to *sign-extend* the input from the switches from 8 bits to 16 bits.
- 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 can find it during synthesis.
- On the FPGA, your RISC240 processor always starts with **`$0000`** in the **`PC`** after reset. Since your program starts at address **`$200`**, you will need to add an instruction at address **`$0000`** to branch to your program.

Revise Your Benchmark: If you wrote your program exactly according to the specifications that we asked for in Tasks 1 and 4, then you only need to modify two things. First, remove the parts of the assembly that store the initial data words into memory addresses **`$3000`** and **`$3002`**. At this point, you should synthesize, place, and route your SystemVerilog, and load it onto the FPGA to test your **`MUL`** instruction's functionality. Second, once you are confident that **`MUL`** is working, add the following two instructions *after your* **`MUL`** *instruction*:

- **`MULI r4, r1, $0005`**
- **`MULI r2, r2, $FFFC`**

Again, since the assembler does not support these instructions, you will have to use the same technique to add these to your code as you did for the **`MUL`** instruction in Task 4.

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: Once again, we've decided to minimize the written requirements for this lab. We have deputized the TAs to discuss your work with you and to watch your demo. 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 include the multiplication instructions? Express this also as a percentage increase.
- Which design is better? Why?