COMP 362 Project Description

July 30, 2020

1 Intro

In this class, the final project will be to build a functional microprocessor that can execute RISC-V instructions. We will design the processor in Verilog and simulate it in ModelSim. If we have time, we will synthesize the design for the DE10-Lite board. The project will be completed in six phases:

1. Build a single-cycle non-pipelined processor with an idealized memory that never stalls.
2. Add pipeline latches to separate the design into five stages, still using an ideal memory.
3. Transition the memory into a stalling banked memory module.
4. Replace the stalling memory with a custom-designed cache.
5. Synthesize on an FPGA.

2 Milestones

2.1 Design Review

Each team will draw a complete schematic of the unpipelined CPU using a CAD tool like OpenOffice Draw of Adobe Illustrator. Each module, bus, and signal should be uniquely named. The schematic should be hierarchical so the top-level design contains only large blocks (like register file, ALU, control logic, etc). There will be a one-to-one mapping of modules in your schematic to modules that you implement in Verilog.

During the design review, the team will need to describe how the datapath can implement any legal RISC-V instruction using your schematic as a reference. Teams should also be able to justify their design decisions. The team should think through the control path and decode logic.

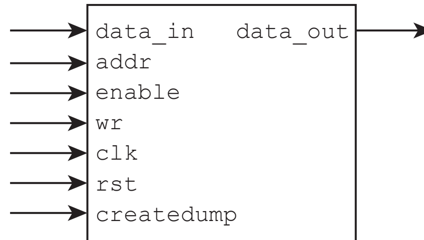

Figure 1: Block diagram of the single-cycle memory.

enable	wr	Function	data_out
0	x	No Operation	0
1	0	Read	mem[addr]
1	1	Write data_in	0

Table 1: Operation of the single-cycle memory.

2.2 Phase I: Single-Cycle, Unpipelined CPU

In this phase of the design, you will implement a single cycle unpipelined CPU similar to Figure 4.17 on page 257 of the Hennesey and Patterson textbook. You will use a single-cycle memory for the instruction memory and data memory. A verilog module for the memory will be available on the course website. Both memories will be initialized with identical copies of your program binary, including the program's instructions and its initialized data.

2.2.1 Single-Cycle Memory Specification

During each cycle, the `enable` and `wr` inputs determina what function the memory will perform according to Table 1.

During a read cycle, the data output will immediately reflect the contents of the `addr` input and will change in a flow through fashion if the address changes. For writes, the `wr`, `addr`, and `data_in` signals must be stable at the rising edge of the `clk`.

At the beginning of the simulation, the memory is initialized from a file called `loadfile_all.img`. You can change the name of the load file in the Verilog source of the memory module if you want. The file is loaded at the first rising edge of the clock during reset. The file format is:

```
@0
12
12
12
```

where `@0` specifies the starting address of zero and `12` represents any 2-digit hex number. Any number of lines may be specified up to the size of the memory.

You can produce a hex file output in this format with `gcc` and binutils using the following syntax:

```
gcc -c testfile.s
objcopy -O verilog testfile.o loadfile_all.img
```

At the end of the simulation, the memory can produce a dumpfile so you can determine what has been written to memory. When the `createdump` signal is asserted at the rising edge of a clock, the memory will create a file called `dumpfile` in the Mentor directory. You may want to use the decode of the `halt` instruction to assert `createdump` on the data memory. The dumpfile will contain memory values from zero to the highest address modified by a write cycle (not the highest address loaded from the loadfile). The format is:

```
0000 1234
0001 1234
0002 1234
```

2.2.2 Testing Your Design

You will test your design by writing simple programs in assembly language and running those programs to ensure that they produce the expected output. On demo day, the instructional team will verify that your CPU can correctly execute some sample programs that we have prepared. To be sure that your processor will pass, you should run your own rigorous tests before demo day. There are a few different types of tests that people usually run:

1. **Instruction-specific tests** to verify that the CPU can correctly execute every variant of a particular instruction.
2. **Random instruction tests** are just a bunch of random instructions that don't necessarily do anything useful.
3. **Targeted tests** to verify that particular paths in the design are working. This will become much more important in the pipelined design when we introduce bypass logic between pipeline stages. We will need to test that all of the bypass muxes are working properly by writing programs that contain dependence chains among instructions.

2.3 Phase II: Pipelined Design with Perfect Memory

In Phase II, the pipelined version of your design needs to be running correctly with no optimizations. Correclty means that it must detect and do the right thing on pipeline hazards (eg. stall). You will still use the single-cycle memory model.

2.4 Phase 2.5: Pipelined Design with Stalling Memory

In this phase, you will replace your perfect memory with a stalling memory, which is similar except it has `stall` and `done` output signals. Your pipeline will need to stall to handle these conditions. Verify your design. You should use the following procedure:

- **Replace IMEM** with the stalling memory, keeping the DMEM the same (i.e. aligned perfect memory). Verify your design.
- **Replace DMEM** with the stalling memory, keeping the IMEM aligned perfect memory. Verify again.
- **Replace both IMEM and DMEM** and verify.

2.5 Phase III: Pipelined Multi-cycle Memory with Optimizations

In this phase you will synthesize your design on the MAX 10 FPGA board. You will replace your memory modules with memories provided by the Quartus II software.