Lab 4: Pipelined ARMv4 Microarchitecture Implementation in Verilog HDL

Due **Sunday 4/4** (midnight)

Instructor: Bingzhe Li
TA: Ricardo Hernandez

1. Introduction

In Lab 3, you implemented a single-cycle ARM machine. In this lab, you will implement a pipelined ARM machine. The pipeline must consist of five stages as discussed in class and covered in our textbook [1].

Now that multiple instructions can be "in-flight" at the same time, you must detect dependencies between instructions and handle them correctly. This lab's objective is to take Lab 3 and see the issues related to its pipelined implementation in hardware. For this laboratory, we will use the SystemVerilog Hardware Descriptive Language (HDL) modeling language. Hopefully, this laboratory will also allow you to see that executing things in parallel within hardware is not an easy choice despite being something the human body can do effortlessly. For full credit, you should achieve the following in this lab.

- Implement a pipelined ARM machine in SystemVerilog.
- Ensure that your implementation is correct by simulating it and verifying the register dumps of the test inputs.
- Implement your design on the National Instruments Elvis III and DSDB and the onboard SDRAM using the built-in DDR3 controller. Collect performance data and compare it to the data you obtained from the single-cycle microarchitecture.

2. Specifications of the ARM ARMv4 Machine

2.1 Instruction Set Architecture

The machine should support all of the following ARMv4 instructions as with Lab 3.

ADC	~~ADD~~	~~AND~~	ASR	~~B~~	~~BL~~
BIC	CMN	CMP	EOR	~~LDR~~	LSL
LSR	MOV	MVN	~~ORR~~	ROR	SBC
~~STR~~	~~SUB~~	TEQ	TST		

2.2 Microarchitecture

Unlike a single-cycle microarchitecture, a pipelined microarchitecture divides the "work" required to execute an instruction across multiple cycles. Each cycle corresponds to a stage within a pipeline. The major advantage of a pipelined microarchitecture is that it can execute multiple instructions in parallel: multiple instructions can be in the pipeline at the same time, albeit at different stages.

However, the major difference is the control cannot be centralized like the single-cycle or multi-cycle architectures. They have to be passed down the pipelined in order to simplify the complexity. In order to help alleviate problems between stages, a Hazard Detection Unit (HDU) (through the `hazard` Verilog module), as mentioned in class, is utilized to keep things working without conflicts.

2.2.1 Pipeline Stages

For this lab, you must implement the following five-stage pipeline. Please ensure that there are exactly five stages. Please ensure that each stage does exactly what it is supposed to do (no more, no less). For example, as long as the memory is accessed only during the MEM stage, you are allowed to generate control signals (or perform other bookkeeping activities) for other stages. Later on, for extra credit, you will be allowed to design your own custom pipeline if you want.

Stage	Name	Specification
1	IF	Instruction Fetch
2	ID	Instruction decode and register file read
3	EX	Execution or memory address calculation
4	MEM	Memory access
5	WB	Writeback to the register file

2.2.2 Handling Data Dependence

For this lab, you must implement a five-stage pipelines that handles data dependences. It should handle the following methods for data dependence as discussed in your textbook and class [1]

1. Stalling. When a data dependence is detected, simply prevent later instructions from entering/progressing through the pipeline. This leads to idle pipeline stages referred to as "bubbles". For this lab, your implementation must stall only when necessary.

2. Forwarding. When a data dependence is detected, allow an earlier instruction to send data directly to a later instruction even before the data has been written back into the register file. For this lab, you must forward data into the end of the decode stage. Your implementation is still allowed to stall, but only when stalling cannot be prevented by forwarding data.

2.3 Memory

As mentioned in class, the previous lab and in the textbook, the memory is stored as bytes. Just like in lab 3, the memory module handles the memory as a Harvard architecture. That is, there is one memory where the instructions are stored and the other where data is stored. You can compile your programs to run on your processor the same way as you did in lab3: using the `arm3hex` script. It will output the program in big-endian machine code, which your processor's memory system is set up to handle.

```
Linux/Mac:
./arm3hex arm-none-eabi-as memfile.s memfile.dat

Windows:
python arm3hex arm-none-eabi-as memfile.s memfile.dat
```

You can check the result of other programs by using your simulator from Lab 2 or other publicly available ARM simulators, such as VisUAL (`https://salmanarif.bitbucket.io/visual/index.html`).

You can modify the `set MEMORY_FILE` line in the DO file change which program your simulator will run.

```
set MEMORY_FILE ./memfile.dat
mem load -startaddress 0 -i ${MEMORY_FILE} -format hex /testbench/dut/ram/RAM
```

This will pull the `MEMORY_FILE` into the simulation and put the program at location 0. This **start location** for your program (i.e., .text in the ELF format) was chosen differently than what you compiled through lab 2. We did this to model the way the FPGA handles memory internally when you get to that portion of the lab. You should not see a difference as the `flopenr` module within `arm_single.sv` loads the starting memory location into the PC upon a reset.

3. Synthesis and Implementation on the National Instruments ELVIS III board

For this lab as with the previous laboratory, we will also implement our architecture into a real device with real memory. The board we plan on using is the National Instruments Virtual Implementation Suite or NI ELVIS III board. In addition to the ELVIS III board, we will attach an add-on device to give us use of a Xilinx Zynq-7Z020 FPGA. For this, we will be using the National Instruments Digital System Development Board (DSDB). The DSDB board is also populated with two Micron DDR3 memory components to give a single rank, 32-bit wide interface with a total of 512 MB of data memory. More information on DRAM and the DDR interface can be found in [2].

Just like lab 3, you should use the provided Vivado project and script to automatically set up the remaining parts of the project, import your pipelined ARM file, and start the toolchain. You can follow the same guide as you used in lab 3, just replacing `arm_single.sv` with `arm_pipelined.sv`.

4. Lab Resources

Any necessary source code for this lab will be distributed through Canvas. However, because you're working on a design that expands on what you did in lab 3, you can copy your lab 3 files into your lab 4 repository to make things easier.

Additional items will be posted to Canvas such as the Vivado project and additional documentation.

5. Getting Started & Tips

1. In the top-level skeleton (`arm_pipelined.sv`), it implements 8 instructions:

   ```
   ADD, AND, B, BL, LDR, ORR, STR, SUB
   ```

   You should use these to understand how the processor works and modify the datapath/control to add the remaining instructions. Instead of repeating part of the same lab again, you should copy your progress from your lab 3 repository to your lab 4 repository and use that as you base to create a pipelined processor.

2. Since it takes additional cycles to commit an operation than the single-cycle ARM microrarchitecture, I highly advise starting this lab early. Good use of your DO file to highlight areas of the pipeline may also be helpful.

3. Review the material on pipelined machines from the lecture notes and the textbook [1]. The whole key of pipelining is making sure all signals you want to pass to the next stage go through a storage device. If you forget to store a signal, you will have an error sometime in your processor's lifetime. Some signals bypass the registers, but have to be maintained by the control logic which complicates operation for concurrency.

4. The DO file is setup to automatically load in `memfile.dat`, which is the same as the program you used for testing in lab 3. Run the following command to run the simulation: `vsim -do arm_pipelined.do`

5. You should check for correctness by comparing the register dump of the simulation against a reference register dump (e.g., through your simulator from Lab1). Just remember that it takes time for your design to complete the operation as opposed to a single-cycle version.

6. It is far easier to add instructions one at a time and check to make sure they work before going on to the next instruction.

7. As with lab 3, it is easier to implement datapath functionality first within the datapath module and then make sure it works by completing the control module afterwards.

8. You should also analyze the performance of the `memfile.s`, `fib.s`, and programs you wrote with the ELVIS III board. Address whether memory impacted or did not impact your design. Also, compare your results with the single-cycle microrarchitecture and determine which one is faster and what benefits each design has over the other. Explain your reasoning within your lab report.

5.1 Tips

- You are pretty much not going to finish this assignment if you start late - guaranteed. So, start early and apportion your time appropriately.
- Read this handout in detail. Ask questions to the TAs or Instructor using Canvas (especially early on). The sooner you ask, the sooner we can respond.
- When you encounter a technical problem, please sift through your logs file and look any error specifically. Many times the error tells you the problem.
- Be cautious about how Verilog handles signed/unsigned numbers.
- Memory accesses must be aligned. If you are accessing a byte, any address is allowed. But, if you are accessing a 4-byte word, the two lowest bits of the address must be zero.
- The cycle time of your implementation is determined by the critical path of the slowest stage. While we do not really check the cycle time, mainly due to not synthesizing your design, you should still try to keep your cycle time as short as possible. Keep in mind that cycle time is not the whole story. Forwarding paths may actually increase your cycle time, but also improve overall performance (by reducing stalls and decreasing CPI).
- We (the Instructor and TAs) cannot debug your project for you. Part of the point of these labs are to learn to debug hardware designs yourself. We can discuss design methodologies with you, talk about proper ways to approach specific issues, and help explain why an error might be occurring, but we cannot sift through your code and find the problem for you (even if we occasionally did that in lab0 and lab1 previously). Even though all of you will start with the same base template, you all will approach the problem differently and we would have to spend time looking through your design to figure out what you did relative to what everyone else did before we can even start diagnosis. This makes it impossible for even all the TAs combined to help everyone with detailed debugging as it would simply take too long - leaving that task to you. The tools will output helpful information about where something is going wrong if you're having trouble compiling something. If you're having an issue with an instruction that isn't running correctly, you can look at all the signals in the system as each step of the program runs to see where things go wrong. Trace the instruction through the processor and make sure each signal is getting set appropriately. Eventually, you can find the culprit misbehaving signal and then correct it in the HDL.
- Finally, its important to make your processor work first, and then make it work fast. Premature optimization is the root of all evil.

6. Hand in

You should upload **a copy of your code** (compressed Zip file including all code) and **your lab report (PDF file)** to Canvas. Your code should be readable, well-documented, **compilable** and **executable**. In addition, please turn in additional test cases that you used in the `inputs/` directory. If you feel the need to describe any additional aspects of your design in detail, please include these in a separate README. Make sure that you test your Verilog extensively with a suite of test cases, just like in Lab 2 and 3, so that you are confident that you have implemented all instructions correctly. This is for your benefit in later labs.

Again, in order to grade the lab we will use some unknown assembly code that you will not see. To get the maximum grade for this lab, your Verilog must be able to simulate all the instructions asked in Section 2. Your job is to really try to make sure your simulator covers all possible instructions that this unknown code may utilize.

7. Extra Credit

Again, there are many possibilities for extra credit, but you should only attempt this if you get the baseline project done and you have extra time. Some of the instructions you did for Lab 3 are not given to help you get started, however, they can easily be added to the list to increase the complexity of your computer.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2016.

[2] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.