Lab 3: Single-Cycle ARMv4 Microarchitecture Implementation in Verilog HDL

Due **Friday 3/19** (midnight)

Instructor: Bingzhe Li
TA: Ricardo Hernandez

1. Introduction

In this lab, you will implement an ARM processor using a hardware description language (SystemVerilog). The microarchitecture of the ARMv4 machine is single-cycle, a simple microarchitecture that we covered in class.

This lab's objective is to take Lab 2 and see the issues related to its implementation in hardware. For this laboratory, we will use the Verilog Hardware Descriptive Language (HDL) modeling language. Hopefully, this laboratory will also allow you to see the choices for the ARM Instruction Set Architecture and how they make hardware better or worse. For full credit, you should achieve the following in this lab.

- Implement a single-cycle ARM machine in Verilog.
- 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 boards and the onboard SDRAM using the DDR3 built-in controller.

2. Specifications of the ARM ARMv4 Machine

2.1 Instruction Set Architecture

The machine should support all of the following ARMv4 instructions (the instructions striked through mean that these instructions have been implemented as discussed in Sec. 2.3).

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

The Data Processing instructions should work with both the Immediate and Register formats (you do not need to implement the Register Shifted Register format). The Memory instructions should work with offset based addressing (preindexing and postindexing are not necessary). The Branch instructions should work with the standard branch format. All instructions should support the list of condition codes we've discussed in class during lectures.

2.2 PC Prefetch

Similar to Lab 2, the PC register does not directly reflect the PC + 8 offset specified by ARM. Therefore, if the last instruction executed is at address 0x400020, the PC dumped will have the value 0x400024 (next instruction to execute) instead of 0x40002C (next instruction to execute + 8). However, to ensure correctness when using PC, every instruction that uses PC should include the +8 offset (e.g., B or BL). Therefore, the instruction $R0 \leftarrow R15 + R1$ is equivalent to $R0 \leftarrow R15 + R1 + 8$. The only exception to this rule is when storing PC into the link register R14. In this case, you should always store the address of the next instruction.

2.3 Microarchitecture

The machine has a single-cycle microarchitecture: every instruction takes exactly one cycle to execute. Aside from correctness (as defined by the architectural specifications), this is the only constraint that we are placing on the machine's microarchitecture. As long as these two constraints are satisfied (i.e., correctness and single-cycle operation), you are free to implement the microarchitecture in anyway you want. To guide you along the way, we provide an description of the single-cycle microarchitecture that has some of the instructions implemented (7 instructions[1]). A simple testbench is also given along with a few test programs to help with the process of checking, but further testing will be necessary. Some things to keep in mind related to this implementation:

- The architectural state of the machine (excluding memory) is stored in registers: general-purpose registers and Current Program Status Register (CPSR).
- There is a global wire called the "clock (clk)" that is connected to all the registers. The clock synchronizes all events for update
- When a register sees a rising edge on the clock, the register captures the instantaneous "snapshot" of the values on its input. From then on, the register holds the captured values and feeds them to its output. (i.e., the clock edge basically activates a write).
- The output from the register(s) are fed into a combinational circuit consisting of logic gates (e.g., ADD) to instigate the control. In turn, the output from the logic gates are fed back as input to the register(s) that only change on the postive edge of the clock.
- At the next rising edge on the clock, the register again captures the values on its input.
- At each rising edge, the execution of an instruction is initiated. At the next rising edge, the values stored in all the registers (i.e., the architectural state) should be updated in such a way that the instruction can be considered to have correctly executed.
- The Verilog model given to you utilizes a Harvard architecture. The code is loaded through the DO file at the beginning of simulation. Please check the example code to understand how this works.

The odd thing about this type of design, as mentioned in class, is that current instructions really do not update the state of the architecture until the next clock cycle.

3. Behavioral Memory for Simulation

As mentioned in class and in the textbook, the memory is stored as bytes. For this lab, 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. Also, the system stores data in big-endian mode and the Verilog DO file loads the program. To get this to be in the right format, some modifications were made to the script `arm2hex`, resulting in a new version called `arm3hex`. It has the same format as `arm2hex`, however, when you run it, it stores the output in big-endian byte-addressable form.

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

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

Just as we learned in class with PC+8 being a historical item, many companies use a specific endianess because of historical reasons. The true ARMv4 architecture has the ability to switch endianess with a switch at boot.

[1] striked through in Section 2.1

Two new test programs have been included in your inputs/ directory: one named fib.s which runs a Fibonacci calculation F(n) where n is 32, and one named memfile.s which exercises a variety of arithmetic functions and memory operations and is considered 'successful' when memory address #100 holds the value 7 after the program finishes. You can check the result of these programs and the others by using your simulator from Lab 2 and the arm2hex script. We will start using .dat as the file extension for programs in the format for the Verilog processor and continue to use .x as the extension for simulator-format programs.

You can modify the line in the DO file (arm_single.do) to run the simulation.

```
set MEMORY_FILE ./memfile.dat
```

The MEMORY_FILE is then loaded into the system with this command:

```
mem load -startaddress 4000 -i ${MEMORY_FILE} -format hex /testbench/dut/ram/RAM
```

For this memory item, it pulls the MEMORY_FILE into the simulation and puts the program at location 4000 or 0xFA0. The MEMORY_FILE is your compiled program that you want stored in memory. You can choose any value you want, but you must modify the Verilog to reset to this location for the PC (see the flopr module within arm_single.sv for more information). This **start location** for your program (i.e., .text in the ELF format) was chosen differently than what you compiled through lab 1. 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.

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

For this lab, 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 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.

The DSDBs are kept in the supplies closet in the endeavor lab so you'll need to talk to the Instructor/TAs to get one if you need access to one during lab time. Part of the reason for this is so we can ask you if your simulation in ModelSim is working. If it is not, you really should not start on the FPGA portion of this lab - it will only make things vastly more complicated and we will not be able to help you as much.

The board looks like the PCB in Figure 1. The FPGA is labeled 18 and has a heat sink on it, whereas, our SDRAM memory is labeled 19 in Figure 1. The board plugs into the ELVIS III board through the connector labeled 11. There are other pins and items that we will utilize throughout the labs to help us with debugging.

To work with the FPGA we will be using the Vivado Design Suite, which will handle synthesis, implementation, and also give us access to some debugging tools. However, it should be mentioned that, even with debugging tools, it is *substantially harder* to identify problems in your Verilog on an FPGA, let alone fix them, if you do not already have things working in behavioral simulations like ModelSim. Therefore, it is important you make sure your design is functioning correctly in ModelSim simulations before you move on to the FPGA.

You will be provided with a Vivado project to get you started that already has most of the configuration details set. In fact, the only file missing from the project is the arm_single.sv file that holds the majority of the ARM processor itself. This will be made available on Canvas for you to download, and once you're comfortable with it you can commit it to your repository. We are intentionally not adding this to your repositories so you can easily grab a fresh copy if you feel you messed something up and want to start clean. Inside the zip file on Canvas, you will find the project folder and a script. More information will be given on how to use this later, but the goal is to run the script and it will automatically set up the remaining components of the Vivado project for you. **Ideally**, after running the script, you can just connect to the FPGA and your project will run.

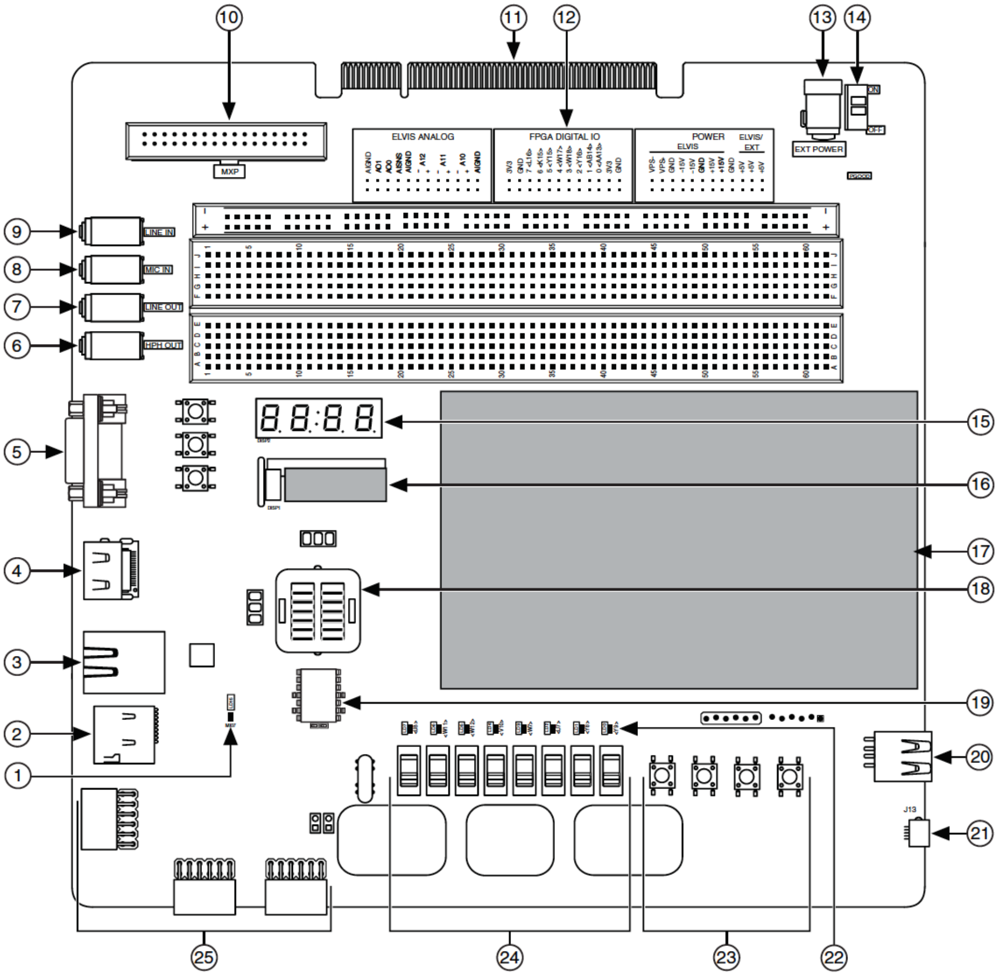


Figure 1: National Instruments Digital System Develpment Board (DSDB).

The top-level design in Vivado utilizes a fabric or interconnection matrix to connect the programmable logic (PL) to parts of the chip we will use to communicate with the memory. This interace in Xilinx is called AXI or Advanced eXtensible Interface. The AXI is based on ARM's AMBA 3.0 open standard and is a specific Intellectual Property (IP) block on the FPGA. The AMBA standard was originally developed by ARM for use in microcontrollers, with the first version being released in 1996 and is now used in many devices that deal with external I/O [1]. Basically, the AXI interface provides a methodology for arbiting connections on its bus structure.

Unlike during the simulation where you changed the DO file to load in a new test program, the FPGA project uses the `$readmemh` command to load in a program of your choice. You can do this from within Vivado by opening the `imem.v` file in the sources panel in the top left and modifying the `$readmemh` command. We have included the `memfile.dat` and `fib.dat` test programs in the project so you can easily switch between them, but if you would like to test your own programs you may do so by adding them with the add sources button and then changing the `$readmemh` command appropriately. By default, the project comes ready to run `memfile.dat`. Once you change `$readmemh` and save the file in Vivado, a notification should pop up to tell you that some modules are out-of-date and give you the option to update them. You should click the button to do so otherwise your changed won't propagate through the project! Anytime you make changes to the verilog files inside Vivado, you will need to update the modules by clicking the button in the notification that appears.

When the `imem.v` file is synthesized, the data in the `.dat` file you selected will be loaded into Block Ram or BRAM. BRAM is a flexible and fast way to store larger amounts of data in an FPGA. The synthesizer for Xilinx FPGAs is smart enough to detect when a block of data in your HDL would be better stored in BRAM instead of individual Flip-Flops throughout the FPGA, so you don't have to manually create and work with the BRAM yourself. Although BRAM is larger than individual Flip-Flops in the FPGA, it is significantly smaller than the external DDR DRAM memory. However, it will be close to the 'processor' and therefore act like registers at the speed the FPGA will be running at. **NOTE:** If you forget to including your `.dat` file in the project but you modify the `$readmemh`, Vivado may lock up during synthesis trying to find a file that hasn't been added to the project. If this happens, you will have to cancel the synthesis and then add the file, else synthesis will never end.

4.1 Memory Interfacing

One of the most challenging elements in computer architecture is utilizing memory. Although memory is typically a different chip or integrate circuit on most designs, its crucial to the operation of any software. Good performance always comes from good memory design as well as getting the data to the processor quickly. To accomplish this, most designers utilize DRAM as its relatively fast, but most importatly can be integrated with large memory spaces (i.e., in the GB). Although DRAM is useful, it involves complicated memory interfaces to make sure the data can be stored accurately and quickly on the DRAM device. This is a tradeoff engineers made to make DRAM big - that is, they sacrificed the simplicity of the circuit for the size of DRAM that can be created.

To make the data get to the processor correctly, a memory controller is usually used. Most designs today use a fast memory controlled called a Double Data Rate (DDR) synchronous interface. This memory controller is crucial to making the Synchronous DRAM works correctly. It speeds up the transfer by utilizing both edges of the clock. Fortunately, there is a DDR3 controller on the FPGA, so we will use this IP that is already there through the AXI interface via a memory mapped address. The Double Data Rate (DDR) memory controller consists of three major modules: a core memory controller and scheduler (DDRC), an AXI memory port interface (DDRI) and a digital PHY and controller (DDRP) [1]. Although the DDR controller allows the transfer of data to/from the processor easily, it is an asynchronous protocol in that it has to wait for the memory to respond that it is complete.

This means you will have to utilize a specific protocol on your ARM single-cycle architecture to make sure it works. The protocol works as follows: First, a memory strobe is initiated (`MStrobe`) that is asserted to start the memory transfer process. The `MStrobe` signal remains asserted as long as a memory transfer is initiated (i.e., a LDR/STR instruction). This means you will have to modify your ARM microarchitecture, specifically in the control, to output this signal. Once the memory transfer is complete, it will respond with a signal (`PReady`) to indicate it is complete. After a `PReady` is received, the processor can deassert the `MStrobe` signal. This is called a handshake or asynchronous protocol. Unfortunately, your processor has its own clock and it cannot wait for the memory to complete, therefore, you have to stop or **stall** the processor to make sure it does not go forward. The easiest way to do this to use an **Enable** on your PC. Fortunately, you should have a Enabled flip-flop in your `arm_single.sv` Verilog file to accomplish this task. A Finite State Machine controller is provided in your `elvis` directory of your repository to help with interfacing the strobe and enable signals.

4.2 Synthesis

With the Verilog files modified and the `.dat` imported and set up, you can use `generate bitstream` to begin processing the project. `generate bitstream` starts the toolchain that takes the project through the following steps:

Block Generation → Synthesis → Implementation → Generate Bitstream

As indicated in class, it is important that if you implement any hardware, it should be as close as possible to the given implementation using digital gates as possible. Your Verilog should use a RTL or structural coding for any design. If you are unsure whether your design is RTL or structural, please ask through Canvas or the instructor/TA. Using behavioral implementations or code you find through Google searches, many result in synthesis churning for hours to find a good implementation. Do not be tempted to model your hardware incorrectly by thinking you can find a similar design on the Internet. Once implemented, make sure you test your design through thorough simulation and do not synthesis until you are sure your design will work.

If any errors occur during the toolchain execution, the process will halt and a log of the errors will be printed. More often than not, the errors explain exactly where the issue is and you can easily go to their location, solve the problem, and restart the toolchain with `generate bitstream`. The final product, the bitstream, is what you will upload to the FPGA. To do this, you will need to use the hardware manager tool. Make sure the DSDB board is connected to the computer with the microUSB cable and powered on before trying to connect to the board. If you have used the ELVIS III before in another class, you will know they are already connected to the computers in the lab. However, the DSDB needs to be connected with a separate cable - it cannot communicate through the ELVSI III. Once connected, you can program the FPGA with your bitstream.

Once you have this running and are able to view the results of the FPGA run, you should include data about how many cycles your memory reads and writes take in your report. This is the main way in the report to show that you have successfully got your project working on the FPGA, though we will also be testing things ourselves while grading to make sure things worked as you described.

5. Lab Resources

The simulation source code for the lab is provided in Canvas. The source code is written in Verilog, a very popular hardware description language. Specifically, the Verilog is in a form of System Verilog that should not differ from Verilog except for the way inputs and outputs are defined. If you need a refresher on Verilog, please refer to the Harris/Harris text as well as the Verilog Tips document on Canvas [2]. As mentioned on the documents within Canvas, making your Verilog resemble the digital logic you want to implement will make things easier. Do not be tempted to Google code and simply copy-paste as this can easily lead to synthesis problems in Vivado.

6. Getting Started & Tips

1. In the top-level skeleton (`arm_single.sv`), the following 8 instructions are already implemented:

   ```
   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. Please note that LDR/STR only implements offset-based load and store instructions. Preindex and postindex types of instructions can be implemented, but might require a considerable hardware investment and/or additional cycles.

2. At this point, you should be able to simulate the test input memfile.dat by executing the following command in your shell: `vsim -do arm_single.do`

3. You should check for correctness by comparing the register values of the simulation (inside ModelSim) against a reference register dump (e.g, through your simulator from Lab2). You can also look at the source assembly code and find out what the final values or results should be in the program if you prefer to do it manually.

4. It is far easier to add instructions one at one time and check to make sure they work before going on to the next instruction. This is where writing your own assembly files comes into play as you can write one that tests exactly what you want while only additionally using the instructions you know work. The files we've provided for you generally test a large number of instructions at once and may not be good for individual instruction testing. Also, make sure you use the methodology heavily discussed in class to make adding new hardware easier.

5. It is easier to implement datapath functionality first within the datapath module and then make sure it works by completing the control module afterwards. Guessing which signals you need to add to the control and then trying to figure out how to use them in the datapath is a recipe for disaster.

6. You should also analyze the performance of the `memfile.s` and programs in the `benchmarks` directory in your repository with the ELVIS III board. Address whether memory impacted or did not impact your design. Explain your reasoning within your lab report.

6.1 Tips

- You are pretty much not going to finish this assignment if you start late - guaranteed. Do not believe me; ask others who have had this class. 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.

- Just as a reiteration, we made the memory significantly smaller than what this ARMv4 architecture can support.

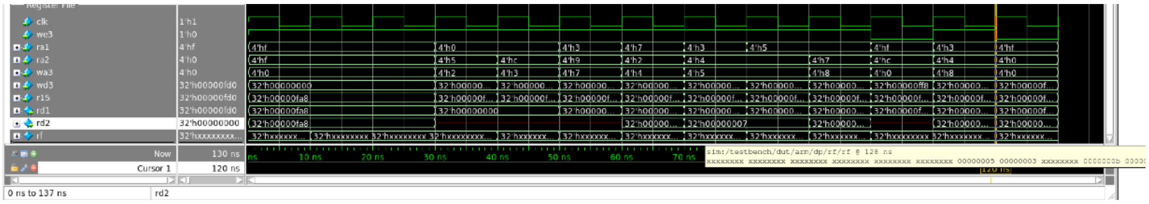
- When you encounter a technical problem, please sift through your log files and look for any error specifically. Many times the error tells you the problem although not always in detail.

- Be cautious about how Verilog handles signed/unsigned numbers.

- The `DO` file is modified to segment the waveform to order the screen so it is easier to read. Try to understand what the file is changed within the `DO` file - good clean output on a waveform can tremendously aid in your debugging experience. For example, you should see all the datapath signals in one section, whereas, the control signals are in another section. This will help you keep track of what signals are accomplishing which task. Also, the memory and register file are added as two-dimensional arrays on the waveform within the DO file:

  ```
  add wave -hex /testbench/dut/arm/dp/rf/rf
  ```

  If you move your mouse over the specific signal on the waveform, you should see all values for the register file as a yellow box, as shown in Figure 2. This can help in debugging values – you can also display all the memory values by going to the Memory List window through the View Menu in MGC ModelSim. The resulting window should appear and you can double click any memory item (i.e., two-dimensional array) to view it properly, as shown in Figure 3.



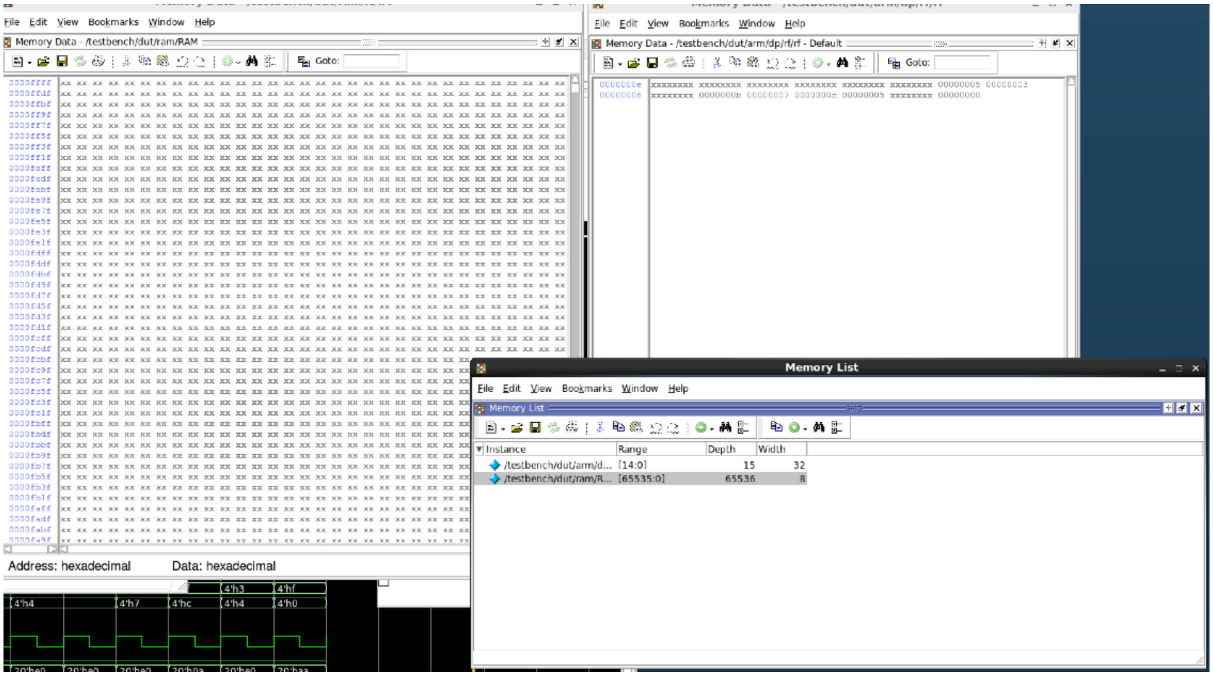
```
sim:/testbench/dut/arm/dp/rf/rf @ 128 ns
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000005 00000003 xxxxxxxx 0000000b 0000
```

Figure 2: Two Dimensional Vector Display in MGC ModelSim Waveform.

- Modify your `DO` file for any signals you want to identify. Sometimes, ModelSim will not log deeply-nested hierarchies unless they are identified in the `DO` file.

- Memory accesses <u>must</u> 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.

- 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 lab1 and lab2 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


File Edit View Bookmarks Window Help
Memory Data - /testbench/dut/ram/RAM
Goto:
File Edit View Bookmarks Window Help
Memory Data - /testbench/dut/arm/dp/rf/rf - Default
Goto:
Memory List
File Edit View Bookmarks Window Help
Memory List
4'h3
4'hf
4'h4
4'h7
4'hc
4'h4
4'h0
20'hea0
20'hea0
20'hea0
20'hea0
20'hea0
20'haa

```
0000ffff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ffdf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ffbf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ff9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ff7f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ff5f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ff3f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000ff1f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000feff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fedf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000febf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fe9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fe7f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fe5f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fe3f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fe1f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fdff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fddf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fdbf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fd9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fd7f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fd5f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fd3f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fd1f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fcff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fcdf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fcbf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fc9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fc7f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fc5f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fc3f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fc1f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fbff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fbdf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fbbf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fb9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fb7f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fb5f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fb3f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fb1f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000faff xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fadf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fabf xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000fa9f xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
```

Address: hexadecimal Data: hexadecimal

```
0000000e xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000005 00000003
00000006 xxxxxxxx 0000000b 00000007 0000000c 00000005 xxxxxxxx 00000000
```

Instance	Range	Depth	Width
/testbench/dut/arm/d...	[14:0]	15	32
/testbench/dut/ram/R...	[65535:0]	65536	8

Figure 3: Memory List Display in MGC ModelSim Waveform.

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.

7. Handin

You should commit all your code to your GitHub repository and 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, 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.

8. 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 2 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] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* UK: Strathclyde Academic Media, 2014.

[2] S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2015.