ECEN 4243-5080
Lab 2: Single-Cycle ARMv4 Microarchitecture Implementation in Verilog HDL
Christan Moser A20173104
Isabell Cook A20208072

Section 1 - Introduction

The purpose of this lab is to explore and expand upon our knowledge gained in Lab 2. We are taking a set of the same ARMv4 instructions but instead of implementing them in C code we are using the Verilog Hardware Description Language (HDL) in SystemVerilog, which is a hardware description language. In implementing the instructions in this different way we get to see the hardware version of these instructions' implementations and the issues that come with it.

In lecture we have gone over the different instructions and how they communicate to the machine through different languages. In lecture we have taken hexadecimal instructions and converted them into binary then into ARMv4 instructions, which we really dived into through Lab 2. The big difference between Lab 2 and 3 is that in Lab 3 we are able to experiment with how the language speaks to the hardware through the use of HDL. Also, this lab uses Big Endian instead of the typical Little Endian that most languages, like C, use.

Our design functions as the given baseline design does. The program takes in ARMv4 instructions that are either register or immediate values, then changes a set of 16 registers, a process counter, and flags according to the indicated instruction. The implemented instructions consist of ADC, ADD, AND, ASR, B, BIC, CMN, CMP, EOR, LDR, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, SUB, TEQ, TST, and BL. Our program was not implemented on the National Instruments Elvis III and DSDB boards due to time constraints.

We tested our design against the given test files, memfile.dat and fib.dat, by going into the do file and changing the file that is going to be executed. For each test our program functioned just as intended, with little delay at run time. Our program took the input from the test files and showed their results through the waveforms in verilog.

Section 2 - Baseline Design

The baseline design was to implement a single-cule ARM machine in Verilog that ran a set of instructions: ADC, ADD, AND, ASR, B, BIC, CMN, CMP, EOR, LDR, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, SUB, TEQ, TST, and BL. The Data Processing instructions should work with both Immediate and Register formats, not Register Shifted Register, the memory instructions should only use offset base addressing, and the Branch instructions should work using the standard branch format. To ensure that the design is working we were instructed to simulate it using a .do file and verify the register outputs were correct using the created waveforms. Then there are the options according to the baseline design to implement the design on the National Instruments Elvis III and DSDB boards.

The baseline process itself would take in a formatted .dat file through the .do file. This .dat file would be read for ARM instructions. These instructions would be broken down by the program into the different parts of the instructions, including the cond, funct, Rn, Rd, and Src2. These different parts of the instructions would then be sent out to different parts of the program, like the control, datapath, and alu, which then processed the data to decide how the program would handle the instruction. There are a total of 16 registers (R0 through R15) that the instructions can modify as well as a process counter that is modified according to the different instructions.

We are given a program that accounts for a couple instructions: AND, ADD, B, LDR, ORR, STR, SUB, and BL. We are then to use those as an example of how to implement the other fourteen instructions. We are provided with a .do file that contains the run instructions of the program that we change according to what test file we want to run.

Our implementation of this design included all that listed in the baseline design minus the optional board implementations. Due to the limited amount of time we focused our efforts in ensuring that the design itself was performing properly then to worried about the board implementation. Had we had more time then we would have done the board implementation.

This design exhibits many design principles: modularity, encapsulation, and extensibility. The design is modular because it is broken up into the different parts of the machine/process. The ALU process is separated from the MUX processes, and even the MUX processes are separated from other MUX processes. We also have the controller separated from the datapath, and both of those are separate from the main arm process. The design has encapsulation because each of the processes in the design are used over and over again for more then one function. The alu controls all the data processing instruction actions, not just the add instruction. The design also has extensibility due to how easy it would be to add to it. Since the sections are broken apart so nicely you can easily see where you could add to the design. For example, you could easily add more instructions just by adding to a few spots in the program, which includes the alu.

Section 3 - Design

This design creates a single -cycle ARM machine in Verilog that has a total of 16 registers (R0 through R15) and a process counter that can be modified. The accounts for a total of twenty two different ARM instructions that are either a data processing instruction, a memory instruction, or a branch instruction. The data processing instructions account for Register or Immediate formats, not Register Shifted Register. The Memory instructions account for only the offset based addressing instructions. The branch instructions only account for the standard branch format instructions. The instructions included are ADC, ADD, AND, ASR, B, BIC, CMN, CMP, EOR, LDR, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, SUB, TEQ, TST, and BL.

A test file is read through the .do file, which is the verilog version of a makefile. The .do file reads in the specified test file and deciphers the text into a set of ARM instructions. These ARM instructions are then taken and broken down into their respective pieces: cond, Func, Src2, Rn, Rd, Op code, and etc. These pieces of the instruction are then fed to different processes within the design, such as the alu, controller, datapath, decoder, conditional logic, and etc, to decide what the instruction should do to either the process counter or the registers. Once the pieces are sent to all of their respective processes then the design will take the information created at each process, combine them, and change either the process counter or register accordingly.

For a simple example, the design breaks down an instruction from the test file and converts it to binary, 1110 00 1 0100 0 0001 000 000 0001010. The design would then go to the decoder and find which type of instruction this is based on the OP code that is created from the binary instruction. In this case the OP code is 00, which is a data processing instruction. The design then looks at the 5th index of the Funct piece of the instruction, which is 101000, and since Verilog uses Big Endian the 5th index is a 1 which indicates to the design that this is an immediate function. The design then sets the RegSrc, ImmSrc, ALUSrc, and etc based off of this information. Then using the 4th through 1st indexes of the Func piece of instruction, which corresponds to the cmd, the design figures out which data processing instruction is being used. Here we have 0100 which is an ADD instruction. This information finds

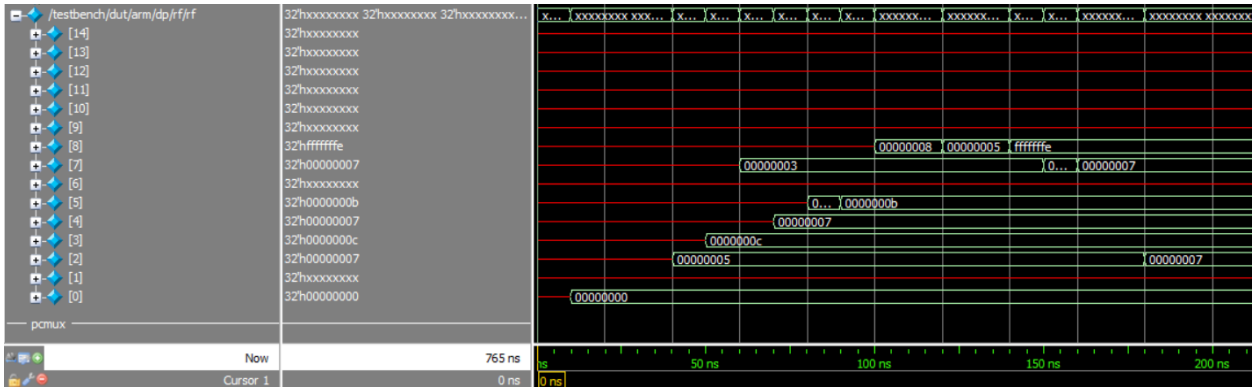
its way through the datapath to the ALU where the ADD instruction is implemented with the given registers from the instruction.

We chose to implement this design in this way to show how the hardware actually functions and to show which piece of hardware accounts for which step in the processes are executing ARM instructions. In doing this we get to use a number of design principles, as well as see the hardships that come with hardware implementation of these instructions. We were able to break up the design into the different hardware pieces which is part of the design principle modularity. We were able to expand and are able to continue expanding the design to include different instructions rather easily due to the design principle extensibility. We are able to reuse the same process to handle multiple different commands, thanks to the design principle encapsulation. Furthermore we see the hardships of handing ARM instructions through hardware by manipulating the design in terms of hardware. We see that it is a little bit of a process to get parts of information from the top at the arm all the way down to the ALU. Then sometime you will need information from the ALU back up at the top under the controller. Information is handled very differently using hardware implementation then when handled by just basic coding logic, as we say in Lab 2.

Section 4 - Testing Strategy

To test the efficacy of our design, we implemented the given ARM instructions with multiple instructions and compared the instructions to the values in the register file. The two main files here are memfile.s and fib.s which we used the arm3hex python script to convert this into byte-addressable hex instructions that can be fed into the single-cycle ARM processor Verilog program. This was then analyzed in MODELSIM and the waveform output inside the simulator was compared to the correct values of the registers based off of the commands.

The MODELSIM waveform output also allowed us to check the correctness of flags and memory addresses to understand the operation of the simulator and how the ARM processor works on a more detailed level. The unit testing checked the accuracy of data-processing, memory, and branch instructions required for implementing into the processor. The fib.s file was used generally to test the branching functionality of the processor and the flag operations while the memfile.s file was used generally for more arithmetic operations and checking. Both types of operations were used in both files but fib.s implemented a recursive function and was thus more rigorous in this branch testing. Memfile.s used both ADD, SUB, and ADDS, SUBS to give us a better understanding of how our design handled flags and set flags.



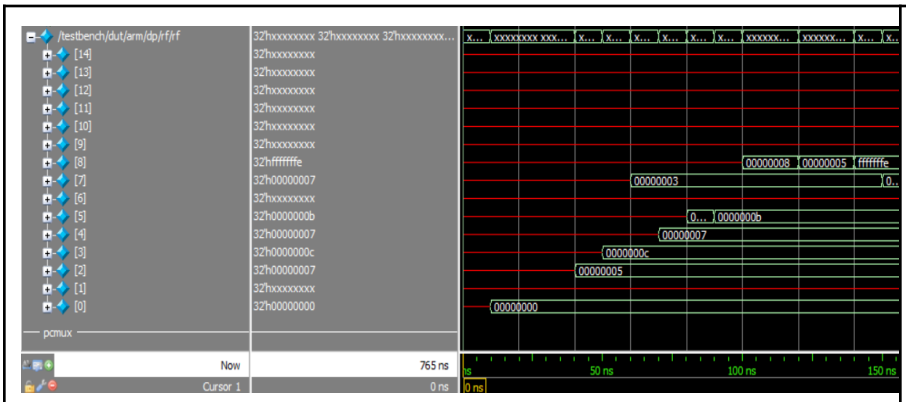
As an example, here is an image of our MODELSIM waveform output. It allowed us to view with each clock cycle how the register values would change to reflect the operations of each instruction. This design is single-cycle and thus will only complete a single instruction in each clock cycle. This made analysis

simple as only one instruction could occur at a time. MODELSIM graphed the waveforms for all of the instructions but only the Register File was required in the analysis for us to properly evaluate the design completeness.

Section 5 - Evaluation

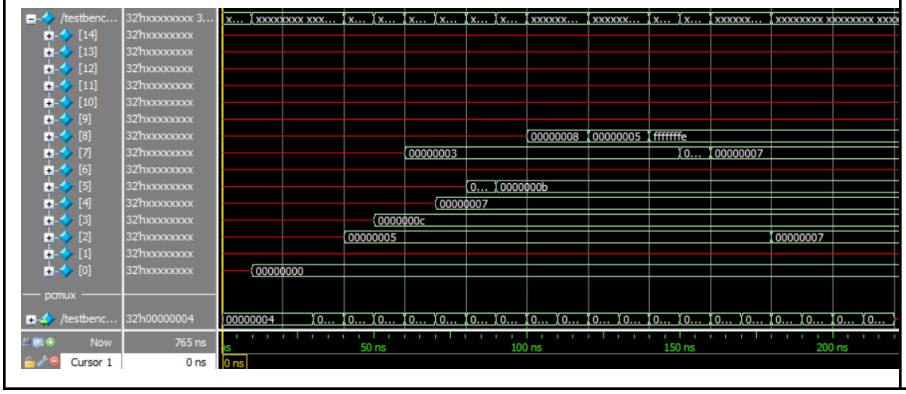
Our design was tested against the test files given, including memfile and fib. For each of the test files we ensured that the values of the registers given by the waveforms in verilog match up with their expected values from the test files.

Memfile analysis:

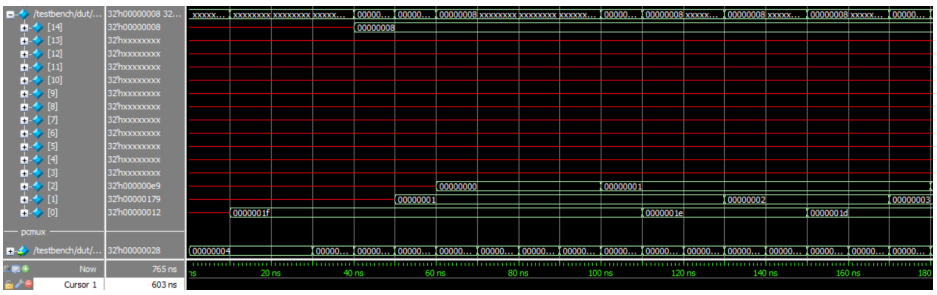
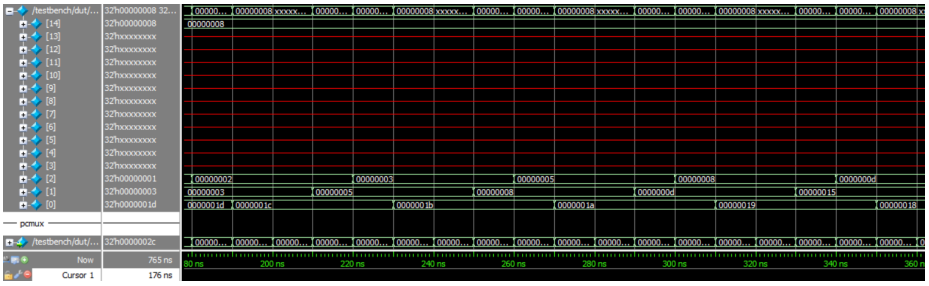
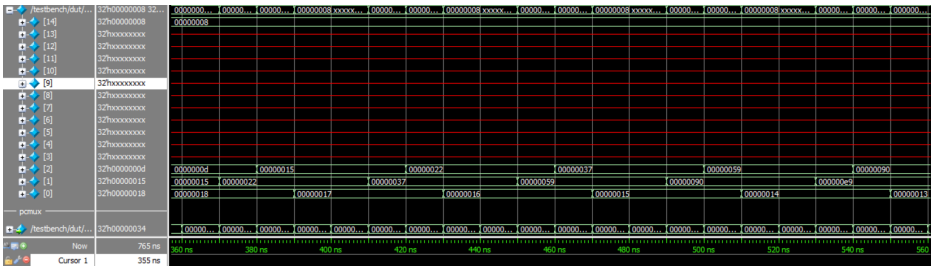
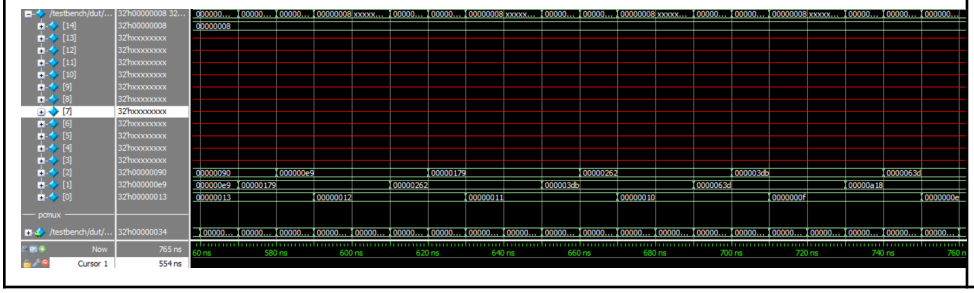
<table>
<tr>
<td> 
</td>
<td>In this waveform we can see the initial operation is SUB R0, R15, R15, which fills register 0 with 0. This indicates SUB is working correctly.

ADD then goes to 5, which is the correct result of ADD R2, R0, #5. Thus, ADD works correctly.

We see that R4 = 7, which is the result of 3 OR 5 thus indicating the ORR function works correctly.

We can also see that R5 goes to 4 and then C which indicates the AND function is working correctly.</td>
</tr>
<tr>
<td> 
</td>
<td>This image is an expansion of the results of the previous waveform. WE can see that the memory addressing works correctly because R2 is filled with 7, which is the value of mem[100].</td>
</tr>
</table>

Fib analysis:

	The first instruction of fib is to MOV 0x1F to R0, which is numerically 31 in decimal. Obviously, we can see MOV works correctly because of this. We then branch to fib and fill R1 with 1, R2 with 0, and compare R0 to 0. This is not the case, so the function goes to loop and completes a fibonacci sequence recursively for N times, where N was initialized to 31. (32 rounds).
	This is further elaboration of the loop function which goes ADD r1, r1, r2; SUB r2, r1, r2; SUBS r0, r0, #1; bpl loop. This continues throughout the program.
	Continuous looping which showcases the Fibonacci sequence.
	The clock was not continuing long enough to complete all 32 iterations, but worked correctly in all cases.

Given the results of the use case test suite, our single-cycle ARM processor operated within the given design and operated correctly. If we had more time, we would have implemented the design on the Vivado FPGA hardware, but given the restraints of the time, we were unable to complete this portion. Additionally, we would have ran more testing suites for more rigorous case testing.