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

Cozette Dyer, A20188563
Coleman Curtsinger, A20189885

1. INTRODUCTION:

In this lab we are tasked with using a low level assembly language in order to implement a set list of instructions into the ALU. In the previous lab we looked at all of these instructions in a higher level language, breaking them down into assembly during lab 3 really allowed us to better understand how the decoder processes instructions into specific values such as Rn, Rd, Sh, etc. which is utilized in the ALU. One of the most important purposes for this lab was to understand how each instruction performed by the ALU, whether it be a data processing, branch, or memory instruction is logically constructed.

Our group felt it best to start out by refreshing ourselves on what each instruction given should produce as an output. We wanted to make sure we had a solid base of understanding before we started to code in the provided *arm_single.sv* file so that we could avoid having to go back and fix mistakes caused by a lack of knowledge. After a quick refresher, we started implementing each instruction logic into the module alu of the .sv file. Appendix B provided in the Lab 2 files was a really great resource we utilized in order to obtain the different opcodes and conditions for each instruction. Another resource we had was the given instructions for *ADD* and *SUB*, this helped for formatting our subsequent instructions.

To summarize, in this lab we modified a single cycle arm architecture file to allow for the taking in of many more instructions using system verilog. We utilized the given *arm3hex* file to convert *.s* example programs into *.dat* files. Lastly, we modified the *.do* file in order to run multiple different example programs through our alu in order to ensure the correctness of our waveforms and inherently, our code. A more in depth explanation of the testing strategy we went through and how our group worked to design the alu instructions is below.

2. BASELINE DESIGN:

The baseline design for the ALU control logic, in the *arm_single.sv* file, is a great example and easily modified. However, this design has its limitations. For instance, this baseline design of the ALU is 2-bit, defined by the `ALUControl[1:0]` in the code shown below. Unfortunately, this 2-bit ALU can only handle 4 different operations, as 2-bit binary (ie 00, 01, 10, 11) only allows 4 unique code sequences. Thus, under this limited baseline design, only the ADD, SUB, AND, and ORR instructions can be implemented. However, this limitation is easily fixed by changing the size of the ALUControl array, as discussed in the following section. The baseline code for the case statement is given below:

```
always_comb
  casex (ALUControl[1:0])
    2'b0?:  Result = sum;
    2'b10:  Result = a & b;
    2'b11:  Result = a | b;
    default: Result = 32'bx;
  Endcase
```

In the above case statement, the first case is `2'b0?` which then sets the result to sum. At first glance, this `?` may seem like a mistype; however, in this case, this `?` acts as a don't care. The logic doesn't care if this last bit is a 1 or a 0, because the result is the same. Thus, the ADD and SUB instructions are implemented with one line of code, instead of one. This approach of using a *don't care* bit only works when the results are the same for two (or more, with more bits) instructions. If the op-code is 00, then the given instruction is ADD. Thus, b is left alone and summed together for the result. However, if the op-code is 01, then the instruction is SUB, and thus *b* must be

inverted (i.e. the negative of *b*). Both operations are addition (*a + b* and *a + -b)*, which allows the *don't care* bit to work. The code for this operation is given below:

```
assign condinvb = ALUControl[0] ? ~b : b;
assign sum = a + condinvb + ALUControl[0];
```

The sum is the addition of variable *a*, *b* (either *b* or *-b*), and the ALUContol bit. When this control bit is 1 (the SUB instruction), sum is converted to 2C, as the way to convert a number to 2C is to invert the bits and add 1. However, when the instruction is ADD (control bit is 0), the sum is increased by 0, which has no effect. Additionally, the .do file, which tells what files to compile and allocates memory, was given to us with little modification, as discussed in the next section.

3. DESIGN:

The biggest modification to the baseline design given in the *arm_single.sv* file is to increase the size of the ALU. Before, the ALU could only handle 4 unique instructions. However, we modified the ALU to 5-bits, and thus the new ALU can perform 25 different operations, as 5-bits has 25 unique combinations in binary. We had originally planned to double the ALUControl size to 4 bits but ran out of bits to implement the shifts (ASR, LSL, LSR, MOV, and ROR). With only 4-bits, we had two instructions remaining to implement, and thus had to increase the size to 5-bits.

To implement the CMN and CMP instructions, the result is the same as ADD and SUB instructions respectively because the CMN and CMP instructions set flags based on the results of the operation of ADD and SUB. Simularily, the results from the TEQ and TST instructions aren't stored in a register, as TEQ and TST only set the flags for an operation and the result is disregarded. Thus, we set the variable *RegWrite* to zero, so that the result will not be stored in the register. The code shown below is for the CMP instruction.

```
5'b00110:  begin          // CMP
  RegWrite = 0;
  Result = a - b;
end
```

One of the most difficult instructions for us to implement was the ROR, as we didn't understand how to rotate numbers. However, to solve this problem of rotating without creating numerous temporary variables and concatenating those temporary variables together for the result, we right shifted the input *a* by *shamt* and then left shifted *a* by (32 - shamt) and finally ORed the two shifts together. This process is shown in the following code:

```
5'b01111:  Result = a >> Instr[11:7] | a << (32-Inst[11:7]);  // ROR
```

To prove the accuracy of this process, take this example of the process of a rotate:

A = 1010 → left shift by 2 → 1000
B = 1110 → right shift by (4 - 2) → 0011
A OR B = 1000 | 0011 = 1011

This result of 1011 is the correct rotation as the last two digits of A are the first two digits of the result, as they have been rotated to the beginning. Similarly, the last two digits of the result are the first two digits of B. Thus, by ORing the two shifts, the result is correctly rotated.

The only modification to the *.do* file we implemented is to type the following command: *MEMORY_FILE ./memfile.dat* .This command tells the simulation that the memory file being used is *memfile.dat*. This memory file contains the compiled program. Thus, to run the other inputs, we changed the MEMORY_FILE to other .dat files for each input. For instance, to run the *arithtest.s* test input file we changed the .do file by modifying it like so: *MEMORY_FILE ./arithtest.dat.* Once taking in the input file, the simulator places the compiled program into address 0 of the *imem* file, once the *mem load -startaddress 0 -i ${MEMORY_FILE} -format hex /testbench/dut/imem/RAM* line is run.

4. TESTING STRATEGY:

The test inputs for this program were given to us in the inputs folder. However, to use these inputs, they first had to be converted to *.dat* (output) files. The *.dat* files are the machine code of the *.s* input files; an example of a given input file is shown below. The python script *arm3hex* converts a *.s* input file to machine code *.dat* outfile. Once the *.dat* files for each given input file were created, we modified the MEMORY_FILE variable in the *.do* file, as discussed in the previous section. The very beginning of the input file *memfile.s* and *memfile.dat* is shown below:

memfile.s	**arm3hex.py**	**memfile.dat**
MAIN: SUB R0, R15, R15 @ R0 = 0 ADD R2, R0, #5 @ R2 = 5 ADD R3, R0, #12 @ R3 = 12	Convert to machine code	E0 4F 00 0F E2

The table above illustrates the process of converting the input file into something usable. Please note that the table above doesn't show the entirety of each file. Instead, the table is shown above to illustrate the general format of the inputs within each file. The @'s in *memfile.s* are comments that are used for quick reference. For instance, the first comment states that register 0 should have the value 0; additionally, register 2 should have the value 5, and register 3 the value of 12. By directly stating what the results should be, verification of results is incredibly easy and quick for both the students and TA. The waveform results of the code above are shown in Figure 1. This method of direct testing allows the TA and grader to easily and quickly determine if the simulation results are correct. By directly setting the inputs, instead of randomizing them, we are certain of the correct answers. This is especially important given that the whole lab is fairly large and complicated. In addition to the given input files to test the accuracy of our microarchitecture, we also periodically compiled our code to ensure that we caught errors quickly. This testing was especially important as the *arm_single.sv* file is over 500 lines long.

5. EVALUATIONS:

Figures 1 and 2 showcase the register files which are being updated according to the provided *memfile.s* file. The figures are inline with the text (rather than in the appendix) for easy lookup between the table and its related figure. The tables and figures are on the next two pages. Register 1 remains undefined, as denoted by the solid red line in Figure 1, as none of the instructions write to register 1. This also applies to registers 9 through 14. The table below shows the first 6 instructions and the color of the box highlighting the result of Figure 1. Additionally, small comments are also included for clarification. A second table is also included for figure 2. As stated in the comments of *memfile.s,* if the program executes correctly, the value 7 should be written to address 100. At time 180ns, the value of 7 is written to register 2 as highlighted by the purple circle in Figure 2. Thus, the tables outline the process of each instruction and its corresponding waveform as well as the correct result at the end, we can conclude that our single-cycle ARMv4 mircoactuture implementation is accurate.

memfile.s	Figure 1
`SUB R0, R15, R15  @ R0 = 0`	Red. R0 is not modified again and remains zero.
`ADD R2, R0, #5  @ R2 = 5`	Orange. R2 is 5 and remains so.
`ADD R3, R0, #12  @ R3 = 12`	Yellow. R3 is c, which is 12 in Hex.
`SUB R7, R3, #12  @ R7 = 3`	Green. R7 is 3 and remains unchanged.
`ORR R4, R7, R2  @ R4 = ( 3 OR 5 ) = 7`	Cyan. R4 is 7 and is unchanged.
`ADD R5, R2, R4  @ R5 = ( 12 AND 7 ) = 4`	Cornflower blue. R5 is 4 for 1 clock cycle before being changed to *b* by the next instruction.

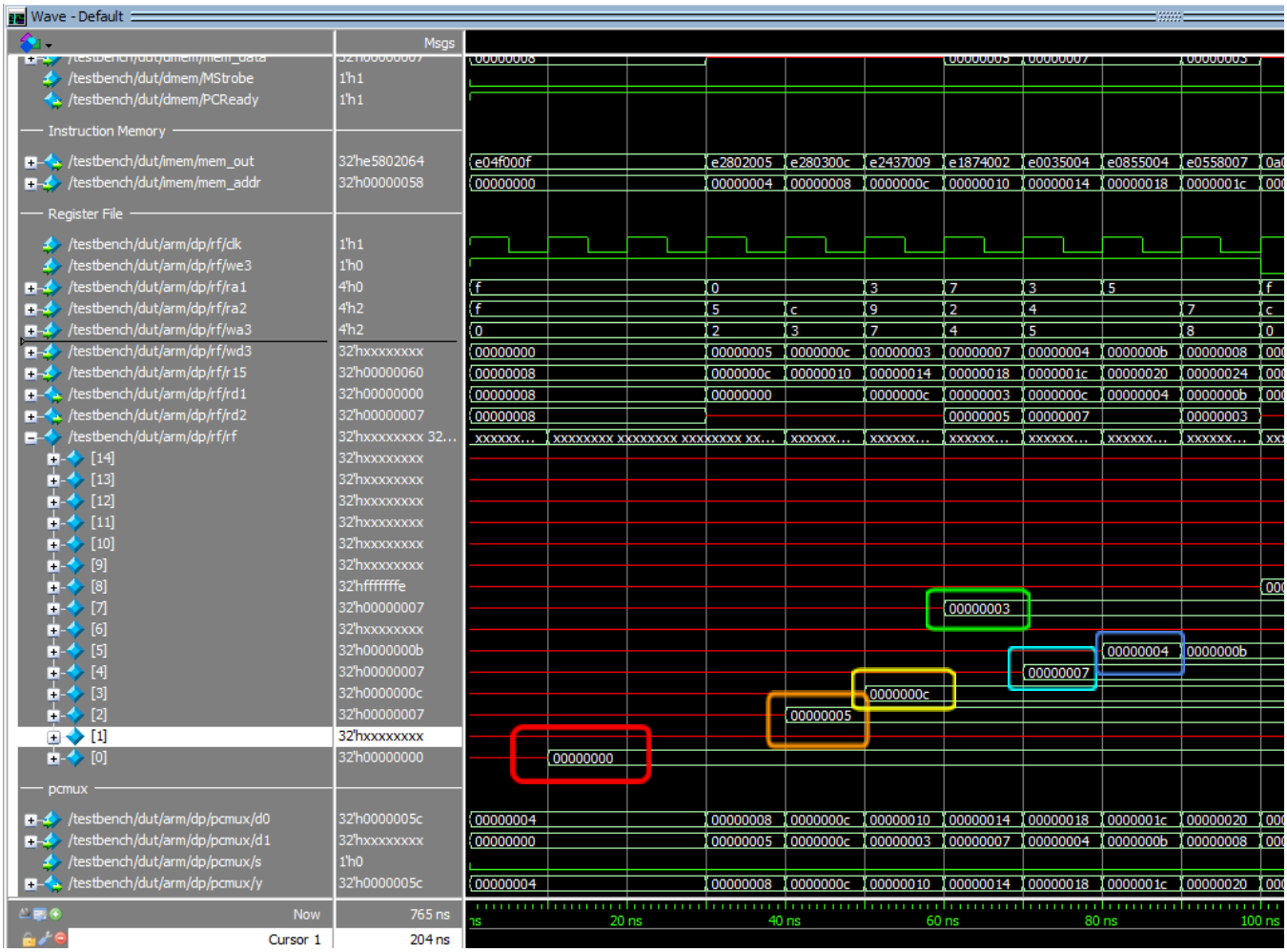

Figure 1: Waveform from 0 to 100 ns showing the registers for input file memfile.s.

memfile.s	Figure 2
`SUBS R8, R5, R7 @ R8 <= 11 - 3 = 8, set Flags`	Red. R8 is 8 but quickly changed. Flags are set, but not shown in Figure 2.
`BEQ END @ shouldn't be taken`	Not taken. If the branch was, the END loop would have executed and made R2 =7. R2 is currently 5.
`SUBS R8, R3, R4 @ R8 = 12 - 7 = 5`	Orange. R8 is now 5 as branch END was not taken.
`BGE AROUND @ should be taken`	Branch taken. Code begins in AROUND loop.
`ADD R5, R0, #0 @ should be skipped`	Skipped, as the branch was taken.
`SUBS R8, R7, R2 @ R8 = 3 - 5 = -2, set Flags`	Yellow. R8 is FFe, which is -2 in Hex. Flags set, but not shown.
`ADDLT R7, R5, #1 @ R7 = 11 + 1 = 12`	Green. R7 is c (hex for 12).
`SUB R7, R7, R2 @ R7 = 12 - 5 = 7`	Cyan. R4 is 7 and is unchanged.
`STR R2, [R0, #100]@ mem[100] = 7`	Purple circle. R2 holds value at address 100 which was 7. End of code.

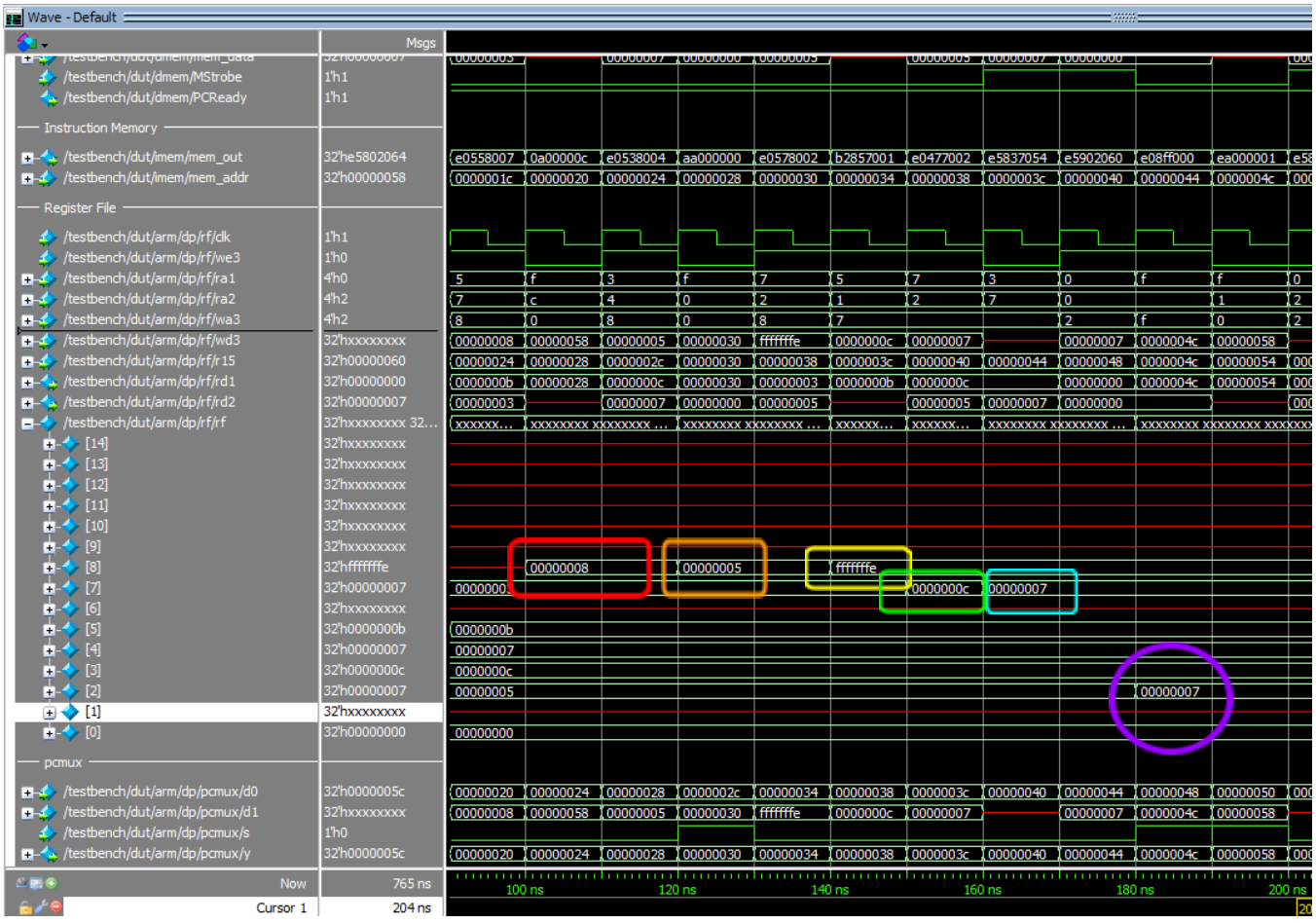

Figure 2: Waveform from 100 to 200 ns showing the registers for input file memfile.s