NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4255 COMPUTER DESIGN
EXERCISE REPORT

Exercise 1

Fredrik Berdon Haave
Michal Majercik
Salahuddin Asjad

October 16, 2015

Abstract

This exercise is about implementing a multi-cycle processor. It is utilizing a subset of the MIPS instruction set. The processor is designed with RTL diagrams and written in VHDL. The completed processor, as well as its individual parts, are verified with testbenches and tested on an FPGA.

Contents

1 Introduction

The task for this exercise is to design a multi-cycle processor that executes MIPS instructions. In this chapter, we will introduce some background and theory about how we approached this challenge.

1.1 A Multi-Cycle Processor

A processor operates in synchronization with what is known as a clock frequency. Every time the clock has a rising edge, the signals are advanced one step through the processor. A processor which only takes one such step per instruction is called a single-cycle processor. While this is quite a simple design, it has some drawbacks. One is that the longest critical path for any instruction determines the clock period for all instructions. This is inefficient, because not all instructions need such a long delay to propagate through their critical path.

A way of improving this inefficiency is to use registers to divide the datapath into segments, where each clock cycle only advances an instruction to the next segment. Although it now takes several clock cycles for an instruction to propagate through the processor, each instruction only needs to go through the segments that are necessary for it to be executed correctly. This shortens the datapath for those instructions that have a shorter critical path. Additionally, since for every clock cycle the instructions only need to be propagated through a single segment instead of the entire processor, the clock frequency can be increased considerably, thus yielding higher performance. The challenge of implementing a multi-cycle processor is that the different instructions does not use the same amount of clock cycles. This has to be handled by the control unit, for example by implementing it as a state machine.

A further improvement to the multi-cycle processor is pipelining. The difference here is that all segments are being utilized at once by different instructions at different places in their respective datapaths. This technique yields even higher performance, and is more commonly used in the real world. [3]

1.2 The MIPS Instruction Set

The MIPS instruction set is a reduced instruction set computer (RISC) instruction set architecture. It was developed by MIPS Technologies (then named MIPS Computer Systems, Inc.), and introduced in 1981. It is a popular instruction set for use in embedded devices and computer architecture courses in universities. [1]

There are three main types of MIPS instructions: the R-type, I-type and J-type. All instructions are 32 bits, and the first six bits of all instructions define the opcode. The opcode is used by the processor's control unit to determine what logic is required to execute the instruction. The R-type instructions use three registers, while the I-type instructions use two registers and an immediate value. The J-type instructions only have a jump address in addition to the opcode. An overview of the instruction types can be seen in Table 1.1.

Table 1.1 Overview of the different MIPS instruction types. (Taken from [2])

Type	Format (bits)					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Table 1.2 Overview of the MIPS instructions we are implementing support for in this exercise. Encoding information from [4].

Name	Type	Operation	Encoding
ADD	R	$d = $s + $t	0000 00ss ssst tttt dddd d000 0010 0000
SUB	R	$d = $s - $t	0000 00ss ssst tttt dddd d000 0010 0010
SLT	R	if $s < $t $d = 1; else $d = 0	0000 00ss ssst tttt dddd d000 0010 1010
AND	R	$d = $s & $t	0000 00ss ssst tttt dddd d000 0010 0100
OR	R	$d = $s | $t	0000 00ss ssst tttt dddd d000 0010 0101
BEQ	I	if $s == $t PC_adv (offset)); else PC_adv (1)	0001 00ss ssst tttt iiii iiii iiii iiii
LW	I	$t = MEM[$s + offset]	1000 11ss ssst tttt iiii iiii iiii iiii
SW	I	MEM[$s + offset] = $t	1010 11ss ssst tttt iiii iiii iiii iiii
LUI	I	$t = (imm << 16)	0011 11-- ---t tttt iiii iiii iiii iiii
J	J	PC = (PC & 0xfc000000) | target	0000 10ii iiii iiii iiii iiii iiii iiii
NOR	R	$d = !($s | $t)	0000 00ss ssst tttt dddd d000 0010 0111

In this exercise we are only implementing support for a subset of the MIPS instruction set. Those instructions we are supporting can be seen in Table 1.2. Please note that the NOR instruction is not required for this exercise, but we went beyond the requirements.

1.3 RTL Building Blocks and VHDL

The starting point for designing hardware is to sketch it with RTL building blocks. Such blocks include for instance logic gates and registers. Once a design has been laid out with RTL building blocks, one can start implementing it in a hardware description language. In this course we are using VHDL to write the modules of our design. We are working with Xilinx ISE v12.4.

We are also using ISE Simulator (ISim) which has the the ability to create testbenches for the circuits in order to check the correctness of the implementation. These testbenches work by applying a clock to the module and driving the input signals. The output signals can then be checked to see if they are as they should be.

1.4 The FPGA and Evaluation Kit

We are using a Xilinx Spartan6 FPGA on an Avnet Evaluation Kit, shown in Figure 1.1. It is connected to the computer via USB, where the Hostcomm utility[5] is used to upload a bitfile to the FPGA and test the implemented MIPS processor.

Figure 1.1 The Xilinx Spartan6 FPGA on the Avnet Evaluation Kit.

2 Solution

In this chapter, we will describe our design and implementation of the processor.

2.1 Datapath

The design of our datapath can be seen in Figure 2.1. It consists of few stages separated by registers.

2.1.1 Program Counter with Update Logic

This stage consists of the program counter (PC), a 32 bit register, that holds the address of the next instruction to be loaded from instruction memory. The register is updated at the beginning of every instruction cycle or after jump and branch instructions.

Regular Update at the Beginning of the Instruction Cycle

During a regular update, the PC value is incremented by one. This is done by an adder with a constant input value of one.

Update During Jump Instruction

The jump instruction takes the six highest bits of the address in the PC and concatenates it with the address field of the jump instruction. This value is then written to the PC. During this phase the 'jump' signal is written high so the concatenated value is fed to the PC input.

Update During Branch Instruction

The branch instruction updates the PC by the value of the immediate, but only in case the two registers in this instruction are equal. In this case the value of the PC is incremented first by one and then by the sign-extended value of the immediate and decremented by one. This decrementing is necessary because the value of the PC on the input of the adder is already incremented by one. During this phase the 'branch' control signal must be driven high to ensure that the right value is written back to the PC.

2.1.2 Instruction Register

This stage consists of the instruction register (IR), a 32 bit register, that holds an instruction during the entire instruction cycle.

2.1.3 Register File and Operands

After the instruction is fetched, the operands need to be acquired from the register file. This register file is an array of 32 registers with a 32 bit length. Three registers can be addressed at once. The first two are read only, and their value are outputted into two simple 32 bit registers. These registers hold this value until the next clock cycle. Data can be written to the third addressed register in case the write enable ('reg_we') control signal is high.

2.1.4 ALU and Execution of Instruction

In this stage the instruction is executed by the arithmetic logic unit (ALU). Which operation shall be executed is determined by the ALU operation unit. The output of this unit depends on the lowest 6 bits of the instruction and the control signal 'alu_op' from the control unit. The output of the ALU is fed to the data memory as an address in case of a load/store instruction, or stored in a simple register to be written to the register file in next clock cycle. In case of a store operation, data that should be stored are acquired from the register file. During the store operation the memory write enable ('mem_we') signal must be driven high. In case of a load or arithmetic operation, output from the ALU is led through a multiplexer to a simple register. The control signal ('mem_to_reg') must be driven low during arithmetic operation and high during load operation.

2.1.5 Simple Registers Between Stages

These registers are designed to hold their value only for one clock cycle when the processor is enabled and store their current value when the processor is disabled.

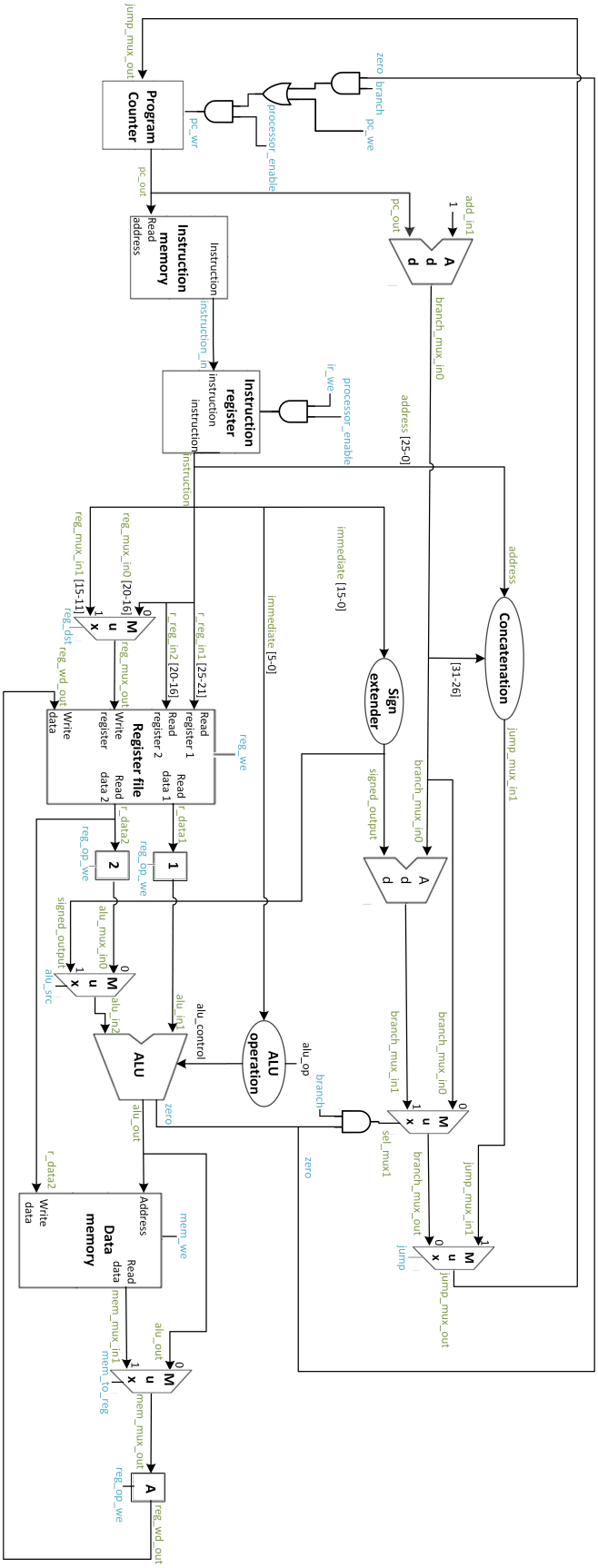

Figure 2.1 Block diagram of the processor datapath.

2.2 Control Unit

The design of the state machine in our control unit can be seen in Figure 2.2.

2.2.1 Fetching the Instruction

In order to fetch the instruction from the instruction memory, first the program counter needs to be updated. This is done in 'STATE_PC_UPDATE'. In the next clock cycle ('STATE_FETCH'), the instruction addressed by the PC is fetched from the instruction memory and written to the instruction register.

2.2.2 Decoding the Instruction

After the instruction is fetched it needs to be decoded in order to decide what operation should be executed. This is done in 'STATE_DECODE'. The branch is made according to the opcode of the instruction.

2.2.3 Executing Jump

Upon decoding a jump instruction, in 'STATE_JUMP' the control signal of the jump multiplexer is driven high in order to feed the new instruction address to the PC.

2.2.4 Executing Arithmetic Instructions

Execution of an arithmetic instruction is done in 3 stages. First ('STATE_RTYPE') the operands need to be fetched from the register file and written to operand registers. In the next stage the input to the ALU is chosen by the ALU source multiplexer by driving the 'reg_dst' signal high. The type of arithmetic operation is selected by the ALU operation unit, which is fed the 'alu_op' signal with value "01" and the lowest 6 bits of instruction. The result is calculated and fed through a multiplexer to register 'A' to be stored in the register file in the following clock cycle. In 'STATE_RTYPE_WB', the calculated value is stored to the register file on the write register address.

2.2.5 Executing Load and Store Instructions

These two instructions are the only two to work with data memory. The address in the data memory needs to be calculated. So in 'STATE_LW_SW', the value of the base register is fetched and fed to the ALU with an offset represented by the immediate part of the instruction. This address is passed to the data memory. In case of a store instruction, data from the register file that should be stored in memory are passed to the data input and memory write enable ('mem_we') is driven high. A load instruction needs to be executed during two cycles. After the address is fed to the data memory, the input of the writeback register ('A') has to be selected. This is done in 'STATE_LW_EX' by driving the 'mem_to_reg' signal of the multiplexer high. In STATE_LW_WB the obtained data are written to the register file by driving 'reg_we' high.

2.2.6 Executing Load Upper Immediate

This instruction loads the 16 lowest bits of the instruction to the upper bits of the target register. After decoding the instruction in 'STATE_LUI', the ALU is fed by the sign-extended immediate part of the instruction. This is shifted by 16 bits to the left. This value is stored to the register file in 'STATE_LUI_WB'.

2.2.7 Executing Branch Instruction

The branch instruction changes the PC by adding the immediate part of the instruction to the value of the PC. This is done only if the two registers mentioned in the instruction are equal. So after decoding the instruction these two registers are fetched and compared in 'STATE_COMPARE'. In case they are equal, the zero flag is set by the ALU and the result of adding the PC with the sign-extended immediate is written to the PC in 'STATE_BEQ'.

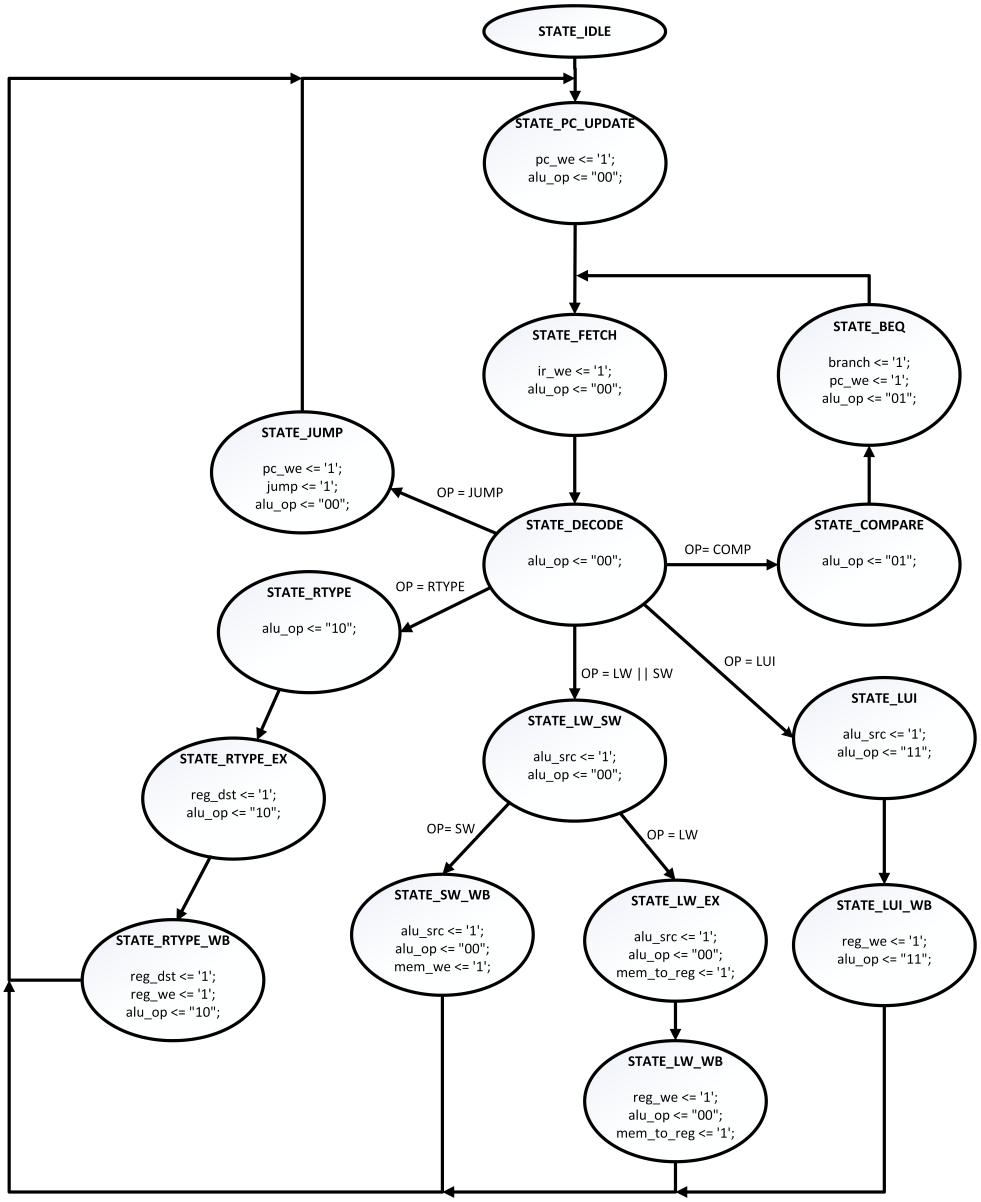

Figure 2.2 Block diagram of the state machine controlling the processor.

3 Results

In this chapter we present our results. The result for our processor is verified by the RTL implementation from Xilinx ISE, ISim techbenches and FPGA testing.

3.1 RTL Implementation

Below is how the RTL implementation of our processor turned out in Xilinx ISE. See Figure 3.1, 3.2 and 3.3.

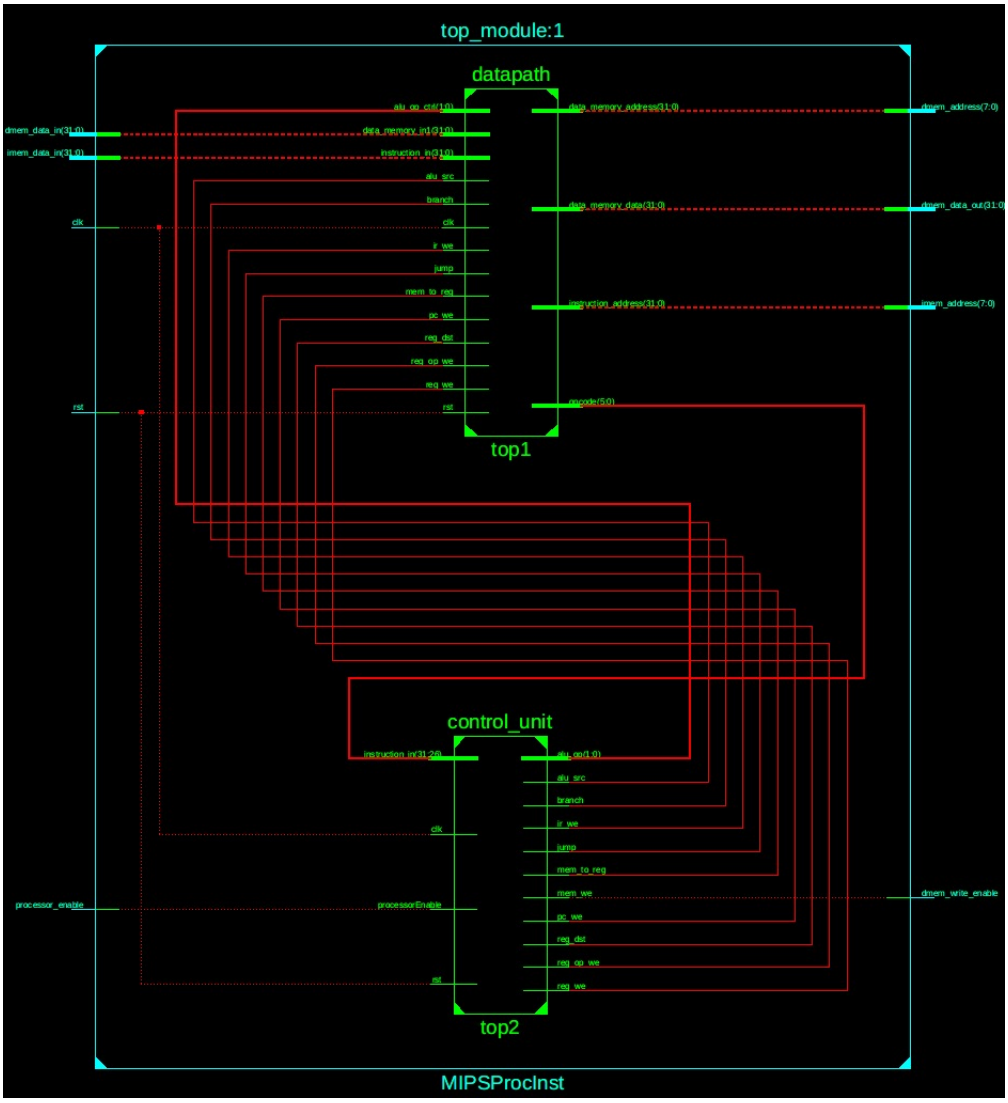

Figure 3.1 RTL schematic for the MIPS processor.

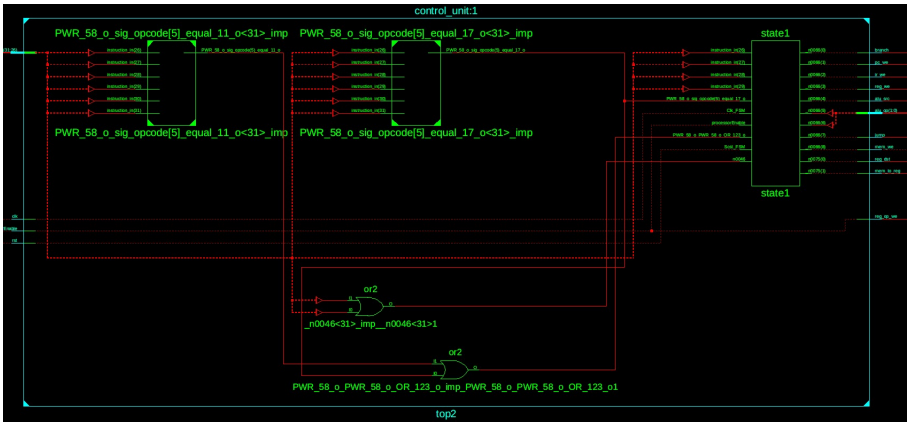

Figure 3.2 RTL schematic for the MIPS processor control unit.

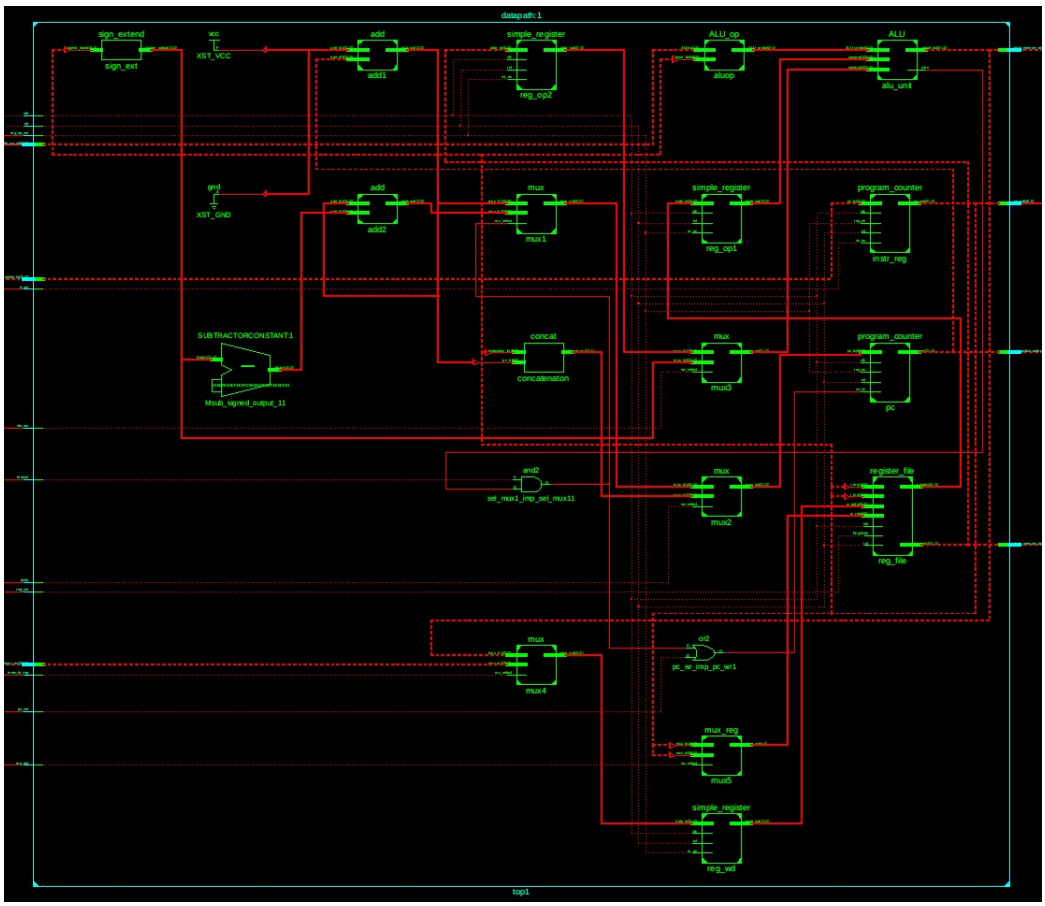

Figure 3.3 RTL schematic for the MIPS processor datapath.

3.2 Testbenches

We used ISim testbench to test every individual module of the processor. When those were confirmed to work correctly, we made a testbench for the entire processor, as shown in Figure 3.4. We also tested the processor with the provided testbench, as shown in Figure 3.5. Both top level testbenches confirmed that the processor was working right in the simulation.

Figure 3.4 Running our testbench.

Figure 3.5 Running the provided testbench.

3.3 FPGA Testing

The Hostcomm utility was used to upload the final bitfile to the FPGA. We then had to write the addresses into the instruction memory and data memory, run the processor and then read the expected data from the data memory. A screenshot of this process is shown in Figure 3.6.

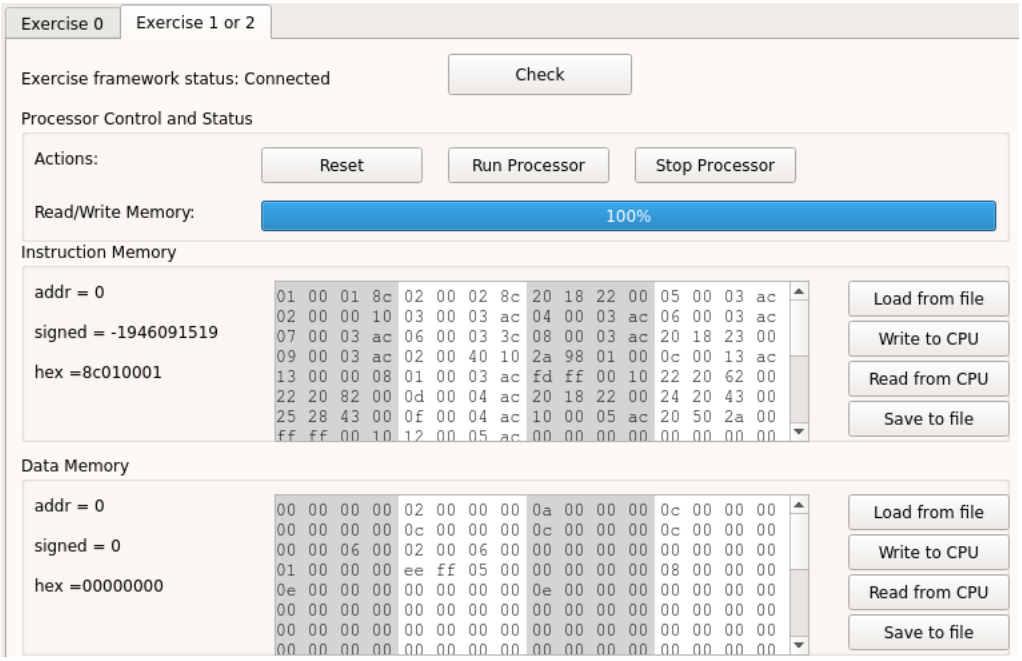

Figure 3.6 Processor running on FPGA.

However, not everything was working as we desired. We found out that our processor had a slight problem with the execution of the last branch instruction. In case of a branch instruction that should branch one step back, there was a slight error. This instruction is executed and the PC value is updated, but this is done one clock cycle too late, so the instruction after this particular branch instruction is executed too.

4 Discussion

In addition to the required minimum of instructions the processor had to support, we also implemented support for the NOR instruction. We also wanted to implement the SLL and SRL instructions, but we were not successful in doing so.

The state machine in the control unit ended up being too complicated. If we had been given the same task again, we could probably have made it simpler. However, we are happy with the way the datapath turned out.

As mentioned in the results, the processor has a problem with one branch instruction. We would have liked to be able to correct this error, but as mentioned above, the control unit is too complicated to fix in the limited time available to us.

4.1 Evaluation of Exercise 1

All of us are satisfied with the exercise lecture. We felt that it was good to get the learning from the exercise lecture before starting of with the assignment.

We are also happy with the exercise itself. It was an appropriate difficulty level for all of us.

5 Conclusion

The goal for this exercise was to create a multi-cycle processor which implements support for a subset of the MIPS instruction set. In addition to the required instructions, we chose to implement the NOR instruction.

We started of by sketching the block diagram for the datapath and the finite state machine for the control unit. Then after we went on to write the code for each component. The design of the processor was verified with testbenches for all the individual modules, as well as a testbench for the top level module. We also tested the processor on the FPGA. This uncovered a small error in the execution of one branch instruction.

Bibliography

[1] Computer Architecture and Design Group. Lab Exercises in TDT4255 Computer Design. Department of Computer and Information Science, 2015.

[2] Various contributors to Wikipedia. MIPS instruction set. `https://en.wikipedia.org/wiki/MIPS_instruction_set`, 2015.

[3] Donn Morrison. TDT4255 Computer Design Lecture 4: The Processor. Available on ItsLearning, 2015.

[4] University of Idaho. MIPS Instruction Reference. `http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html`, 1998.

[5] Yaman Umuroglu. TDT4255 Hostcomm Utility. `https://github.com/maltanar/tdt4255-hostcomm`, 2014.