NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4255 COMPUTER DESIGN
EXERCISE REPORT

Exercise 2

Fredrik Berdon Haave
Michal Majercik
Salahuddin Asjad

November 13, 2015

Abstract

This exercise is about implementing a pipelined 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 a FPGA. The testing uncovered no problems with our design.

Contents

1 Introduction

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

1.1 A Pipelined 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. When all segments are being utilized at once by different instructions at different places in their respective datapaths, this is called pipelining. This improves performance, because for every clock cycle the instructions only need to be propagated through a single segment instead of the entire processor. Since the critical path is shorter, the clock frequency can be increased considerably.

Ideally, a new instruction should be finished every clock cycle, as visualized in Figure 1.1. The challenge though, is that instructions may be dependent on the result of another instruction. When this situation occurs, there is a hazard. These hazards have to be handled, to ensure that the dataflow of the program stays correct. A hazard detection unit is needed to detect hazards. There are different types of hazards, for example data and control hazards.

A naive way of dealing with hazards is to stall the pipeline for as many clock cycles as required for the hazard to be resolved. The drawback of stalling is lower performance. Alternatively, the processor can implement correction techniques for handling hazards. One such technique for handling data hazards is forwarding, as shown in Figure 1.2. The idea here is for the resulting data from the ALU to bypass parts of the processor, and go directly to where it is needed by the dependent instruction. This may avoid or reduce stalling, but a forwarding unit is required to determine if and where to forward data.

Another type of hazard is the control hazard. This occurs when another instruction follows a branch. Because it is not yet known whether the branch is taken or not, the pipeline will have to stall until the result of the branch is known. Alternatively, the processor can try to predict the result of the branch. This is known as branch prediction, but it requires the pipeline to be flushed if the prediction is wrong.

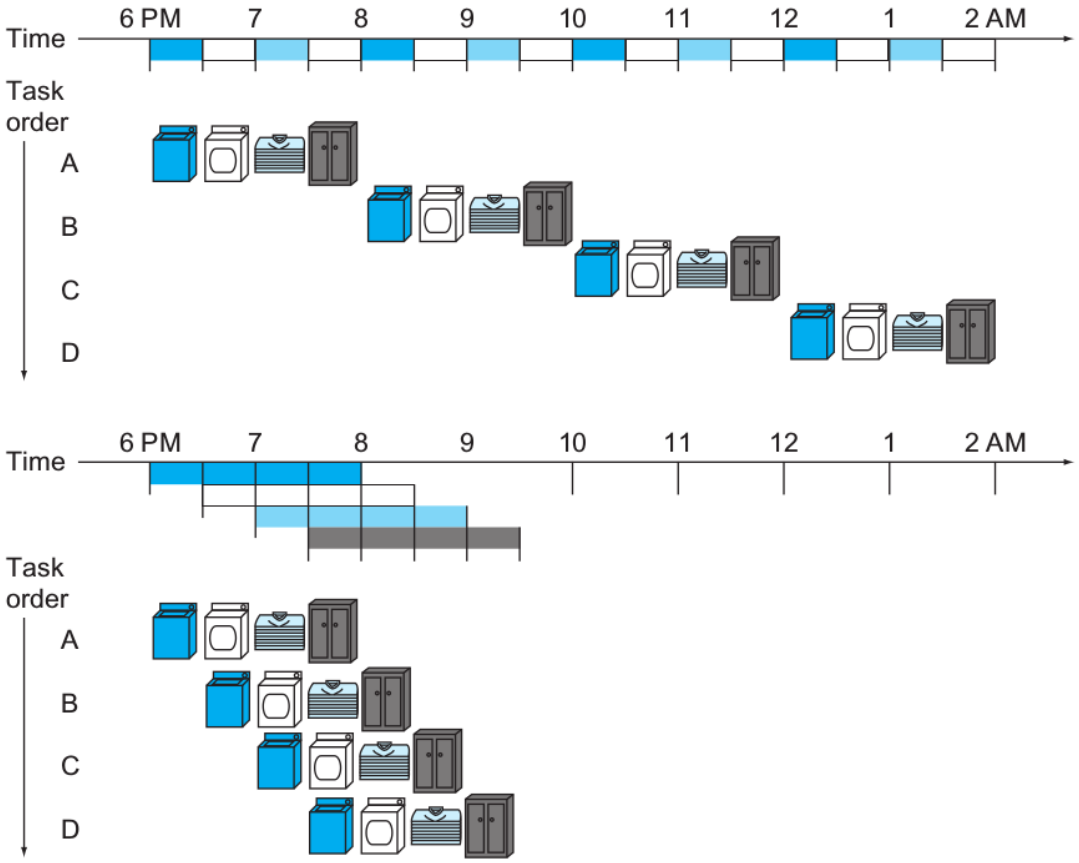

Figure 1.1 The laundry analogy for pipelining. Pipelining is illustrated in the bottom half. Taken from [4].

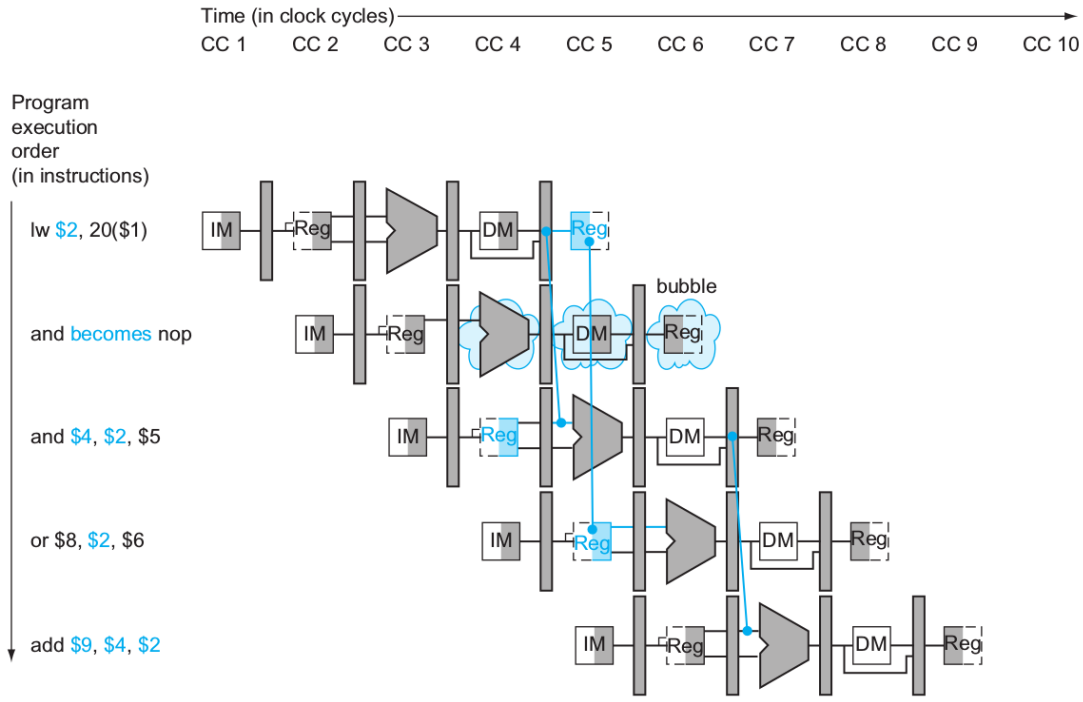

Figure 1.2 An illustration of forwarding. Stalling may still be required, as seen in this case when an AND follows a load. Taken from [4].

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 [3].

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

1.2 The MIPS Instruction Set

The MIPS instruction set is a reduced instruction set computer (RISC) instruction set architecture. 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.

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 The Solution for the Requirements

The processor is mainly constructed from a datapath with five pipeline stages, and a main control module. Additionally, a hazard detection unit, a forwarding unit and an ALU control unit is implemented, which can all be considered to be additional control modules.

1.3.1 Exercise Framework

The processor interacts with the provided data and instruction memories, which are single-ported and have sequential read and write semantics.

We are using VHDL with Xilinx ISE v.12.4 to write the modules of our design. 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 1.3. 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.

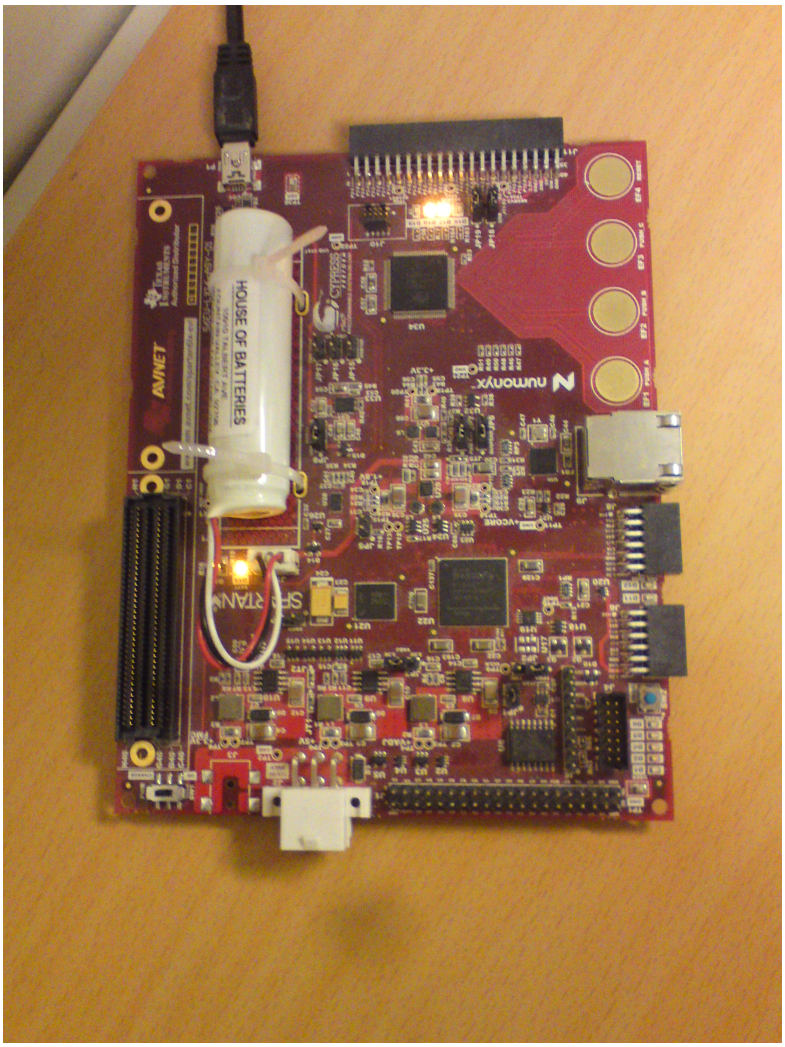

Figure 1.3 The Xilinx Spartan6 FPGA on the Avnet Evaluation Kit.

2 Solution

In this chapter, we will describe our design and implementation of the processor. The processor consists of a datapath and control unit. Data are handled by the datapath depending on control signals from control unit. The design of our processor can be seen in Figure 2.1.

2.1 Datapath

Datapath consists of five stages separated by registers. The stages are: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and write back (WB).

2.1.1 Instruction Fetch Stage (IF)

In this stage the instruction is fetched and passed to the register that separates the IF and ID stage.

Program Counter with Update Logic

The Program Counter (PC) is a 32 bit register that holds the address of the next instruction to be loaded from instruction memory. The register is usually updated at the beginning of every clock cycle, except during stalling, branching or jumping.

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.

Stalling the Pipeline

It is necessary to stall the pipeline when the instruction needs to use the value retrieved from data memory by the directly preceding load instruction. During the stall, writing to the PC is disabled and its value stays the same.

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 driven high so the concatenated value is fed to the PC input. After the jump the pipeline needs to be flushed, in order to prevent instructions loaded into the pipeline directly after jump from executing. This is why we moved the jump logic to the ID stage instead of handling it further in the pipeline. Thanks to this we only have to flush one instruction after jump.

Update During Branch Instruction

The branch instruction updates the PC by the value of the immediate, but only if 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. During this phase the 'branch' control signal must be driven high to ensure that the right value is written back to the PC. We moved the branch logic to the ID stage as well to prevent unnecessary flushing.

Register Between the IF and ID Stage

This register holds the value of PC in case it needs to be updated by a jump or branch instruction in the ID stage. It also holds value of the instruction that was read from instruction memory one cycle ago in case the pipeline needs to be stalled. This register can't hold the value of the instruction read from instruction memory in the current clock cycle because the provided instruction memory is slow and it takes one clock cycle to read from it.

2.1.2 Instruction Decode Stage (ID)

In this stage the instruction is decoded and operands are fetched from the register file. This stage also contains jump and branch logic for the reasons explained earlier.

Register File

The operands are 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 the register dividing the ID and EX phase and the compare unit. Data can be written to the third addressed register in case the write enable ('reg_we') control signal is high.

Branch Logic

Branch logic consists of a compare unit and an adder. The compare unit compares operands from the register file and outputs a signal ('equals') in case their values are equal. The adder adds PC to the sign extended value of the immediate part of the instruction. This sum is written back to the PC in case the branch should be taken.

Jump logic

This logic consists only of a concatenation unit, that concatenates the highest 6 bits of the PC with the lowest 26 bits of jump instruction.

Register Between the ID and EX Stage

This register holds the values of both operands fetched from the register file, the signed extended value of the immediate, the addresses of both read registers and write register and control signals from the control unit that will control the rest of the pipeline.

2.1.3 Execute Stage (EX)

In this stage operands are processed and the result is determined.

ALU

Instruction is executed by the Arithmetic Logic Unit (ALU). Which operands and what operation shall be done is determined by control signals. The output of the ALU is fed to the register between the EX and MEM stages.

Register Between the EX and MEM Stage

This register holds values of the ALU result, data to be written to data memory in next stage, destination register address in the register file and also control signals for the next stages of the pipeline.

2.1.4 Memory Stage (MEM)

This stage consists of data memory. The processor can read (LW) from it or write (SW) to it. If neither of these instructions are being executed, data from the ALU are only passed to the next stage.

Register Between the MEM and WB Stage

The last of the registers dividing our pipeline holds the ALU result, destination register address in the register file and control signals for the last stage of the pipeline. It cannot hold the data read from data memory because this memory is slow and reading from it takes one clock cycle.

2.1.5 Writeback Stage (WB)

In this stage it is decided whether and from which source the data will be written to the register file.

2.2 Control Unit

Unlike in the multi-cycle processor, the control unit in our pipelined processor is not sequential logic. It's combinational logic. The control of our processor can be divided into a few units.

2.2.1 Main Control Unit

The input of this logic is an instruction and the signal 'equals' from the compare unit. Based on these inputs the control unit outputs basic control signals. A table of these signals based on instructions can be seen in Table 2.1.

Table 2.1 Overview of the control signals. ('equals' is input signal)

Signal	**Instruction**						
	NOP	R-type	LW	SW	J	BEQ	LUI
reg_dest	0	1	0	0	0	0	0
ALU_op	00	10	00	00	00	00	01
ALU_src	0	0	1	1	0	0	1
mem_we	0	0	0	1	0	0	0
mem_re	0	0	1	0	0	0	0
reg_we	0	1	1	0	0	0	1
mem_to_reg	0	0	1	0	0	0	0
jump	0	0	0	0	1	0	0
branch	0	0	0	0	0	equals	0
flush	0	0	0	0	1	equals	0

2.2.2 Hazard Detection Unit

The aim of this unit is to stall the pipeline in case there is an instruction that uses data loaded by the instruction directly preceding this instruction. This is detected by comparing register $t from the current instruction with registers $s and $t from the preceding instruction. Also 'EX_mem_re' has to be high, meaning that the preceding instruction was load.

2.2.3 ALU Control Unit

This unit decides what operation should be executed by the ALU depending on the 'ALU_op' signal from the main control unit and the function field of the instruction.

2.2.4 Forwarding Unit

The aim of this unit is to bypass data from the MEM or WB stage into the ID or EX stage. The unit changes the input of the ID/EX register and the ALU by changing the signals controlling the forwarding multiplexers, depending on the addresses of operands and destinations and 'reg_we' signals.

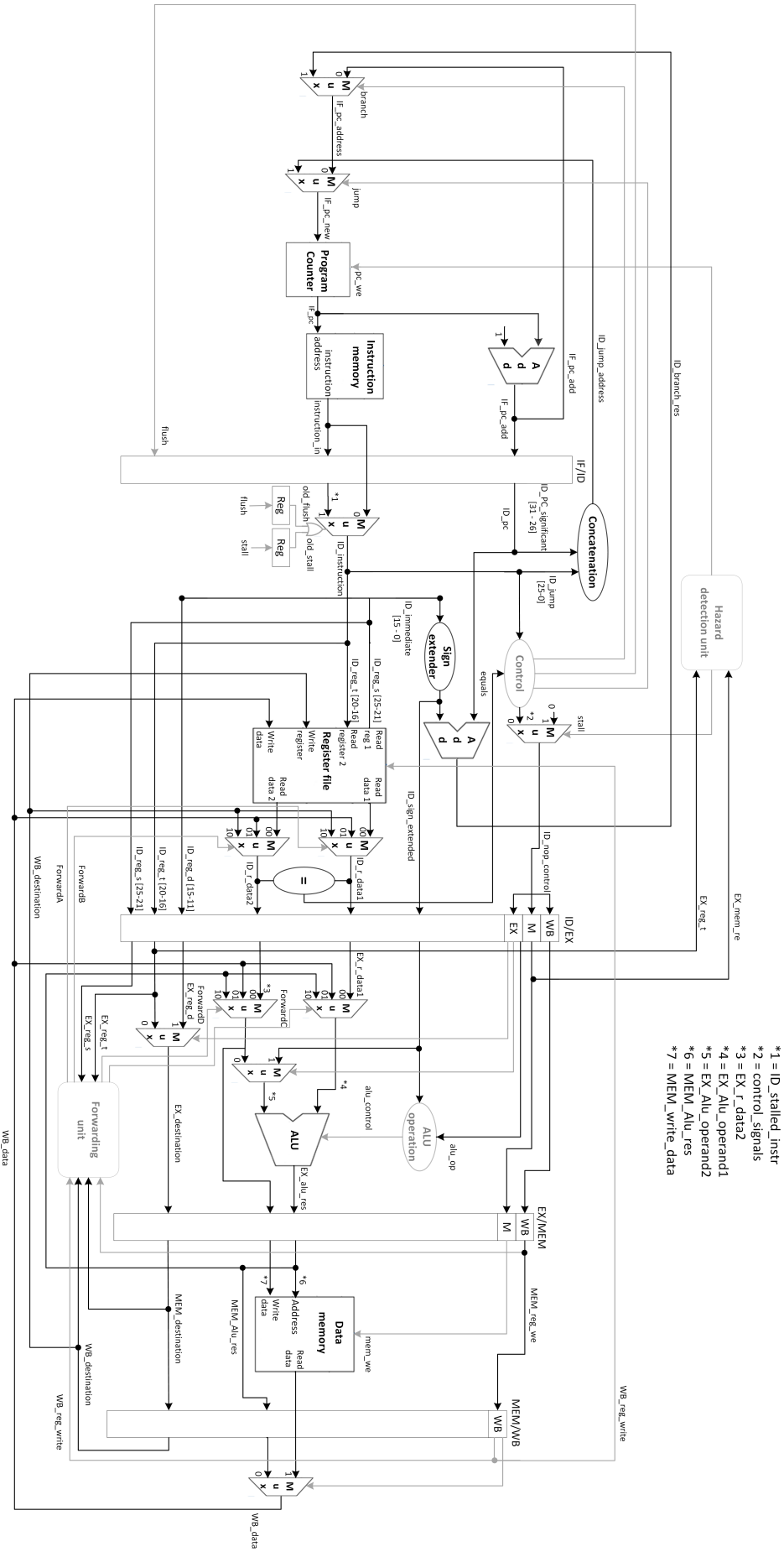

Figure 2.1 Block diagram of the processor.

3 Results

In this chapter we present our results. The result for our processor is verified by the ISim techbenches and FPGA testing.

3.1 Testbenches

We used ISim testbench to test every individual module of the processor. When those were confirmed to work correctly, we ran the provided testbench, as shown in Figure 3.3. This testbench confirmed that the processor was working correctly in the simulation.

The provided testbench we used tests the processor's handling of several hazard conditions, as shown in Figures 3.1 and 3.2. This requires stalling and forwarding to work as intended, as well as the rest of the processor of course.

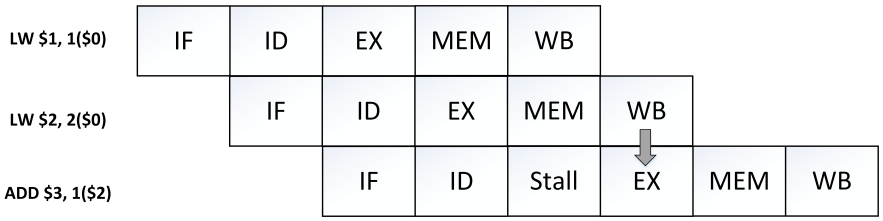

Figure 3.1 Example with stalling and forwarding

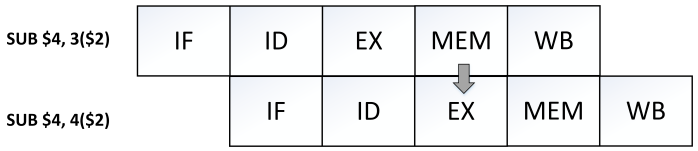

Figure 3.2 Example with forwarding

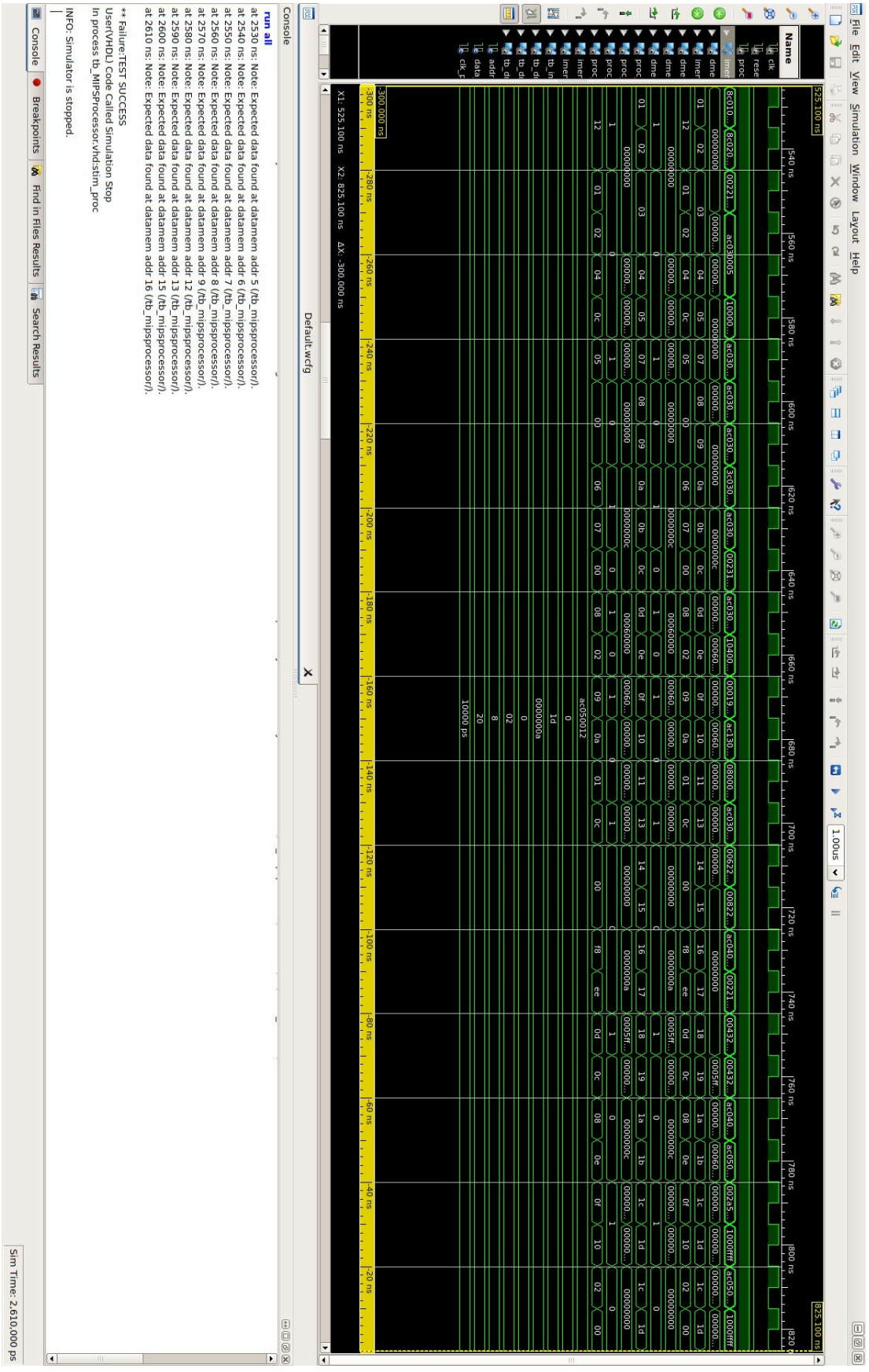

Figure 3.3 Running the provided testbench.

3.2 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 3.4. This shows the same provided testbench running on the FPGA, also here without errors.

For creating instructions to give Hostcomm, we made a Python script to do the conversion from assembly instructions to the hexadecimal little-endian format used by Hostcomm. This saved us some time compared to the last exercise, where we did the conversion by hand. The script is delivered along with the rest of our delivery.

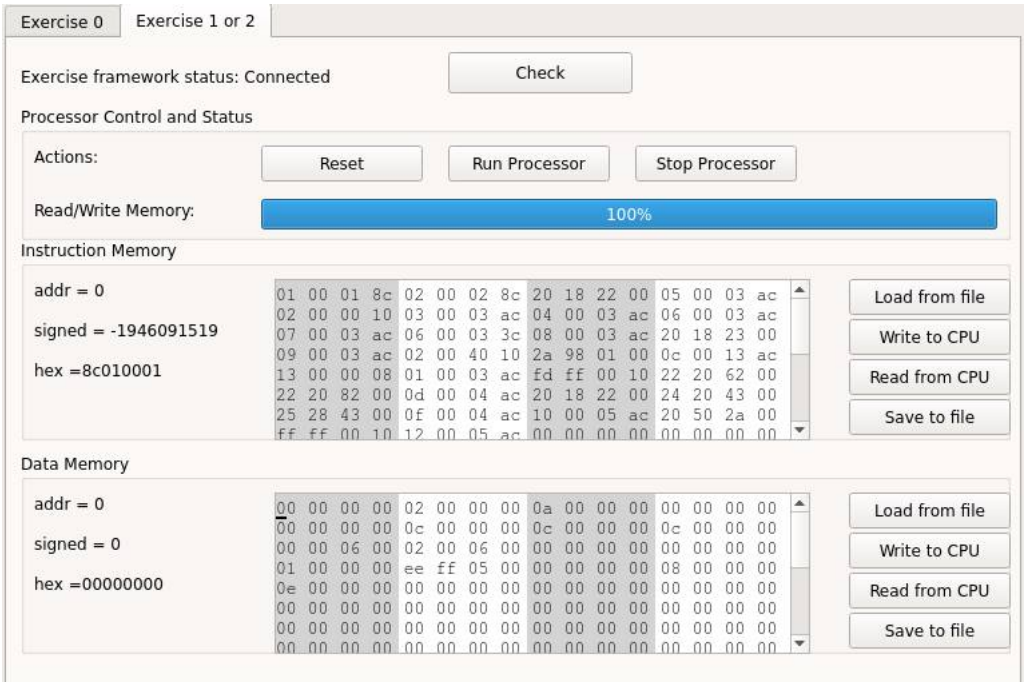

Figure 3.4 Processor running on FPGA.

4 Discussion

In addition to the required minimum of instructions the processor had to support, we also implemented support for the NOR instruction.

For testing the top level module, we only used the provided testbench. We would have liked to create a more substantial top level testbench, but we did not have enough time to do it. This was because getting the processor to a working state with the provided testbench took too long. This also reduced the usefulness of the Python script we had created for converting assembly to the hexadecimal format used by Hostcomm, but it was nice to have made it anyway.

The instruction and data memories take one clock cycle to output data. We discovered that this is too slow to for our pipeline. We therefore had to add bypasses for the data from both memories through the registers directly to the right in the datapath. Please refer to Figure 2.1 for an illustration of this.

In order to further improve the performance of our processor, more could have been done. This applies in particular to the implementation of branch prediction. Unfortunately we did not have time to explore further into this topic.

5 Conclusion

The goal for this exercise was to create a pipelined 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 processor. Then 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 the provided testbench for the top level module. We also tested the processor on the FPGA. This testing revealed no errors.

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

[4] David A. Patterson and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th Edition. Morgan Kaufmann, 2013.

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