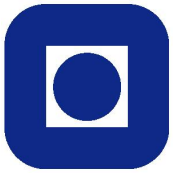

Norwegian University
of Technology and Science

Report
Lab Exercise 1

Wasaznik, Aleksander
Selvik, Andreas Løve
Kulia ,Geir

TDT4255 Computer Architecture
Computer Architecture and Design Group
Department of Computer and Information Science
October 20, 2014

Preface

We wish to give a special thanks to Abdullah Al Hasib for postponing the deadline for the project due to illness. We also want to give thanks to Ilse Visser, Liang Zhu and the organizations PVV and Omega Verksted for providing much needed sustenance trough all hours of the day *and* night. A last thank goes to China and its culture that provided the amazing "Chinese Gun Powder". Without its high levels of caffeine it would be impossible to complete this task.

Abstract

A simple multi-cycle CPU was implemented on a Spartan-6 FPGA. It implements a subset of the MIPS instruction set. The core focus has been performance through keeping the hardware as simple as possible. By writing VHDL with hardware in mind at every step a simple and understandable RTL was achieved. After some additional optimization a max clock frequency of 85MHz was reached. With instructions taking 2 to 3 cycles this results in a performance which is significantly faster than the earlier implementations.

Contents

List of Figures

List of Tables

–

Chapter 1

Introduction

This report explains the implementation of a multi-cycle processor in VHDL, including the design choices made, as per requirement in Excercise 1 of the course TDT4255 Computer Architecture at NTNU[1, p. 44].

1.1 Requirements

The main requirement for the design of the processor was that it should use a simplified version of MIPS instruction. The encoding format having been described in the compendium appendix [1, p. 64]. The processor should have instructions for each of the types of instructions described in Table 1.1 with the operation code (opcode) specified by figure 1.2.

Table 1.1: Requirements

Arithmetic logic unit (ALU):	ADD, SUB, SLT, AND, OR instructions
Branch:	Conditional branch instruction
Memory:	LOAD and STORE instructions
Load Immediate (LUI):	load the upper 16 bits of a register with the given value
Jump instruction:	J-jump to the specified address

Table 1.2: Opcode

Opcode	Operation
000 000	ALU operations (and, or, add, slt, sll)
000 100	Branch if Equal (Beq)
000 010	Jump
100 011	Load word (lw)
101 011	Store word (sw)
001 111	Load upper immediate

The design was to be accompanied with testbenches for all significant parts of the processor. The result of these and handed out tests must be reported. The design should also be demonstrated to run on the development boards at the lab.

The requirements are described in full details in the compendium [1, p. 44].

1.2 Goals

The primary goal, after successfully implementing the requirements, was to optimize the processor for performance.

As a strategy to achieve this, we aimed to write VHDL code that synthesizes into clear and understandable RTL. This was going to aid in both checking for optimality and correctness.

Another overarching goal was to produce readable and maintanable VHDL code. Something that could help achieving the aforementioned goals through easily understandable and modifiable code.

1.3 Workflow

It was decided that the RTL sketch provided in the compenduim was a good design. The sketch was refined and split into entities, and RTL for the individual entites was devised. Testbenches were then written to match the expected behaviour of the entities. Then the actual implementation was written.

1.4 Tools

Manufacturer	Equipment Name	Model
Xilinx	Integrated Software Environment (ISE)	Version 12.4
Xilinx	ISE Simulator (ISim)	Version 12.4
NTNU	Hostcom	TDT Programming Utility
Texas Instruments	Avnet Spartan-6 LX16 Evaluation Kit	S6EV-LX16-PCB-B

Chapter 2

Solution Description

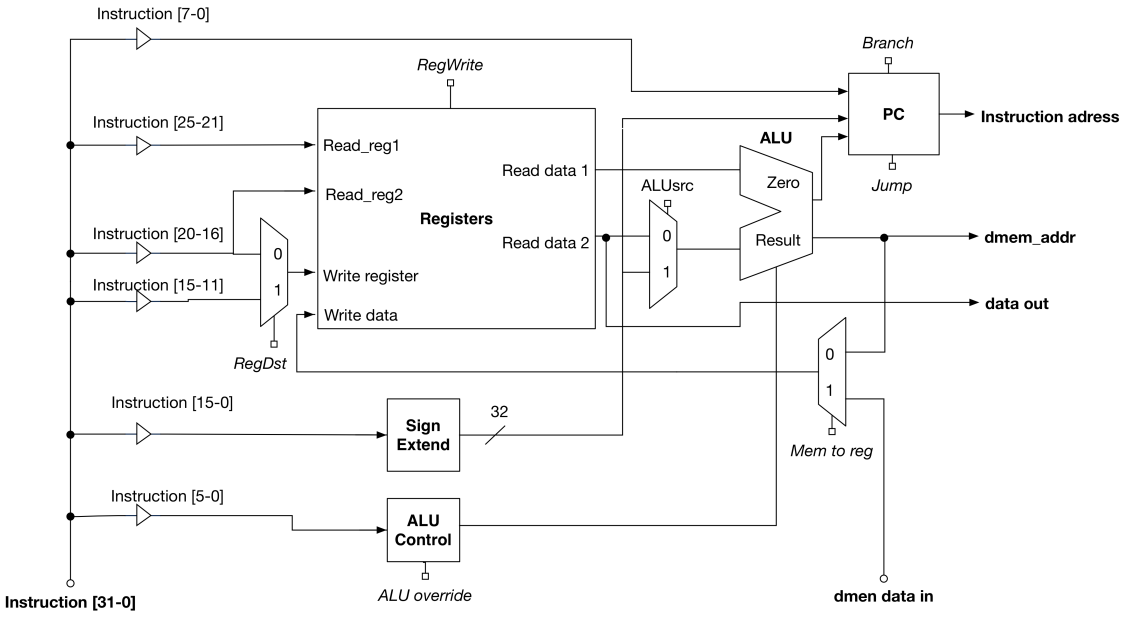

Figure 2.1: The top level RTL sketch. Signals in italic with a square at the end are connected to the *decode* module.

2.1 Top level design

See figure 2.1 for the top level RTL. The Program Counter (PC from now on) contains the current instruction memory address, which is sent to the instruction memory. After a clock cycle, the instruction signal contains the instruction at the specified memory location. Different parts of the instruction is then routed to different parts of the processor. A central role is played by the decode unit, connected to all the open squares in figure 2.1, it uses the op code of the instruction to make sure the processor completes the correct operation.

2.2 Program Counter

Inputs and Output

Input: Clock, next_PC, update_PC
Output: current_PC

Function

The Program Counter (PC) is set to *next_PC* on a rising clock edge if *update_PC*

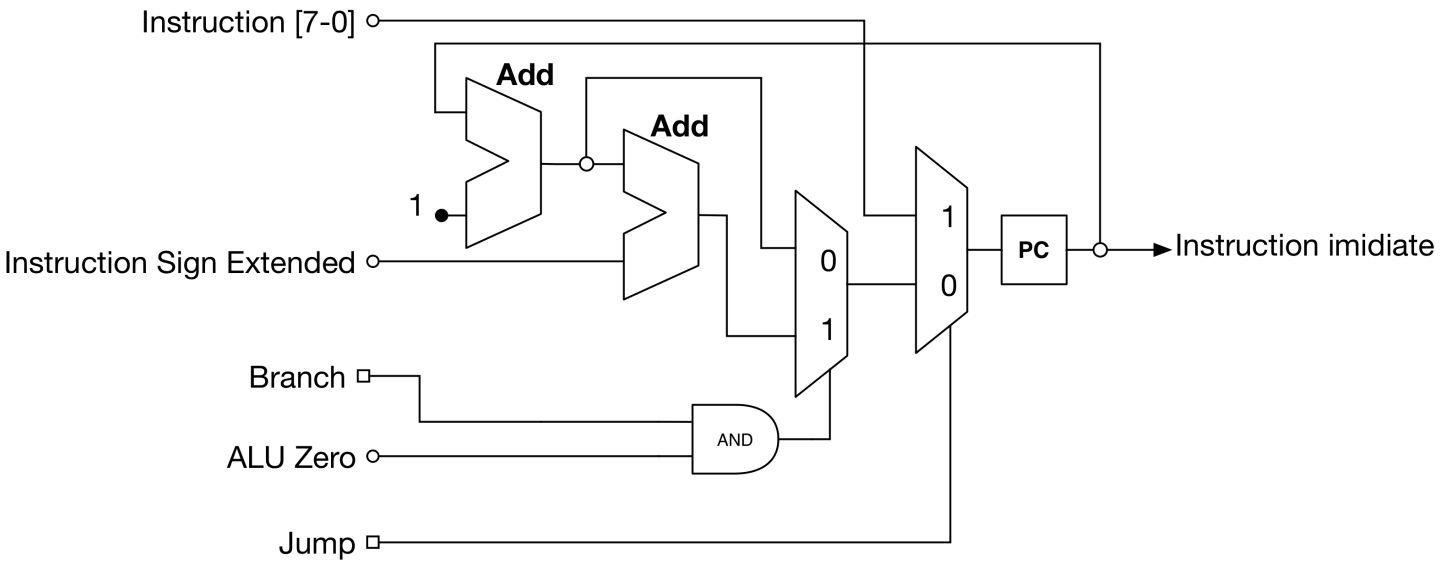

Figure 2.2: RTL sketch for the Program Counter

is set. The $next\_PC$ is controlled by the *decode* module through two muxes; *branch* and *jump*.

If none of them are set, $next\_PC$ is simply $current\_PC + 1$.

If *jump* is set, $next\_PC$ is set to immediate-value part of the instruction.

If *branch* is set and the result of from the ALU is 0, the $next\_PC$ is set to $current\_PC$ + the immidiate value from the instruction.

2.3 Registers

Inputs and Output

Inputs: Clock, read_select_A, read_select_B, write_select, write_data, write_enable

Outputs: register_data_A, register_data_B

Function

This entity represents the register-file of the processor. There are 32 registers. As per specification, register *0* is considered special, and always contains the value *0*.

The two *read_select*-signals control which registers are read out on the two outputs. Additionally, on clock-rise, if *write_enable* is active, write_data is latched onto the register chosen by *write_select*.

The register-file does not react to resets of the system, and retains any values between resets. Also, the value of registers (with the exception of register 0) is undefined until written to once.

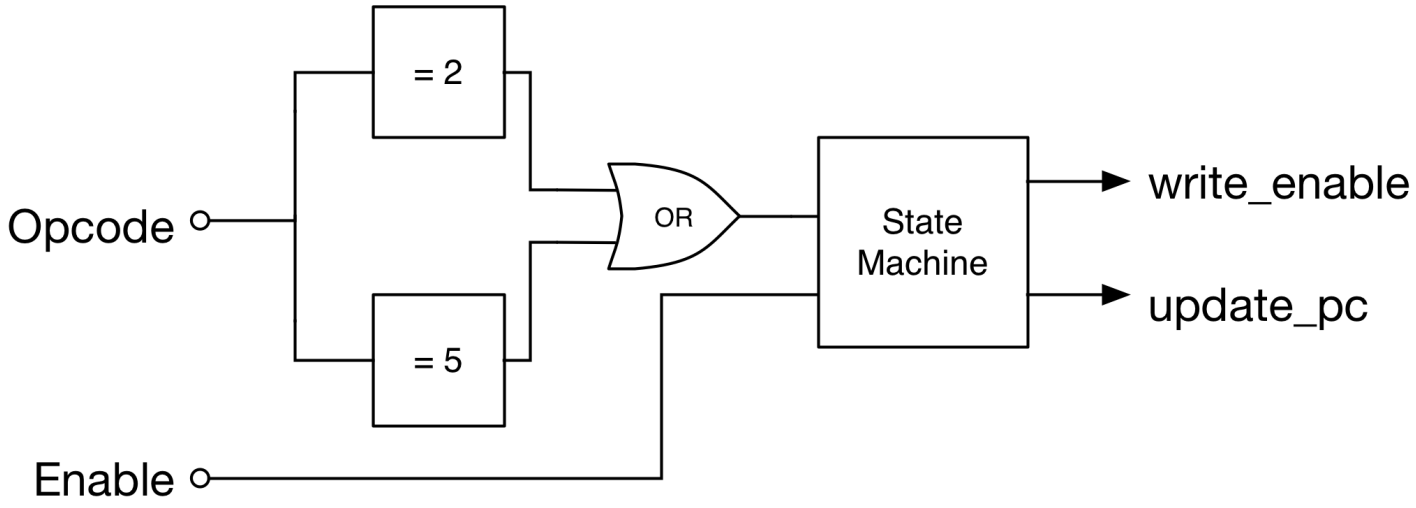

Figure 2.3: RTL sketch for the *control* entity

2.4 Control

Inputs and Output

Inputs:	Clock, reset, processor_enabled, opcode
Outputs:	update_pc, global_write_enable

Function

This entity keeps track of the state the processor is in. There are five states in the processor: *disabled*, *fetch*, *execute* and *memory_wait*. Any transisions happen on clock rise. Both a reset and *processor_enable* inactive forces the processor into the disabled-state. See figure 2.4 for all the transitions.

Disabled The processor sits in this state as long as *processor_enable* is low or *reset* is high. When that is no longer the case, the state is changed to *first_fetch* on the next clock rise. The purpose of the state is to ensure that at least a whole cycle is spent in *fetch* after a reset.

Fetch The purpose of this state is to allow the external memories one cycle to ready data. The signal *global_write_enable* is disabled. This disallows any commits to registers or memory, as the instruction is not yet valid.

Execute This state enables *global_ write_ enable*. Thus at the end of the state, effects of the instruction are committed to registers and memory. If the current instruction is a load or store the next state is memory_wait, otherwise it is fetch. If fetch is the next state, set *update_ pc* high so that it is updated at the end of this state.

Stall As the data memory needs a cycle to access, the load and store instructions need an extra clock cycle delay before fetching the next instruction. That delay is implemented with this state; jumping to fetch after one cycle. It also set *update_ pc* high.

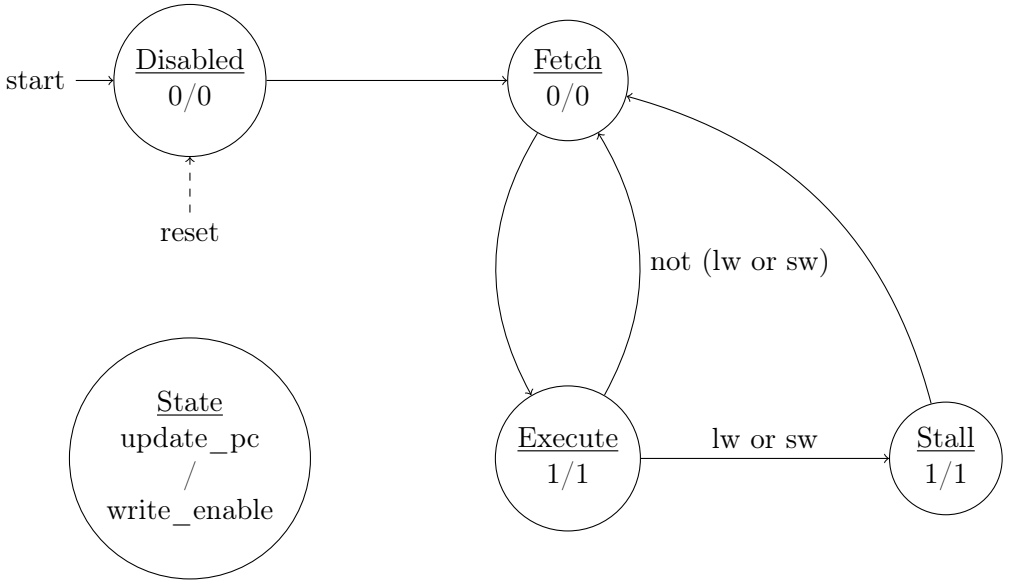

Figure 2.4: The state machine at the heart of the control entity

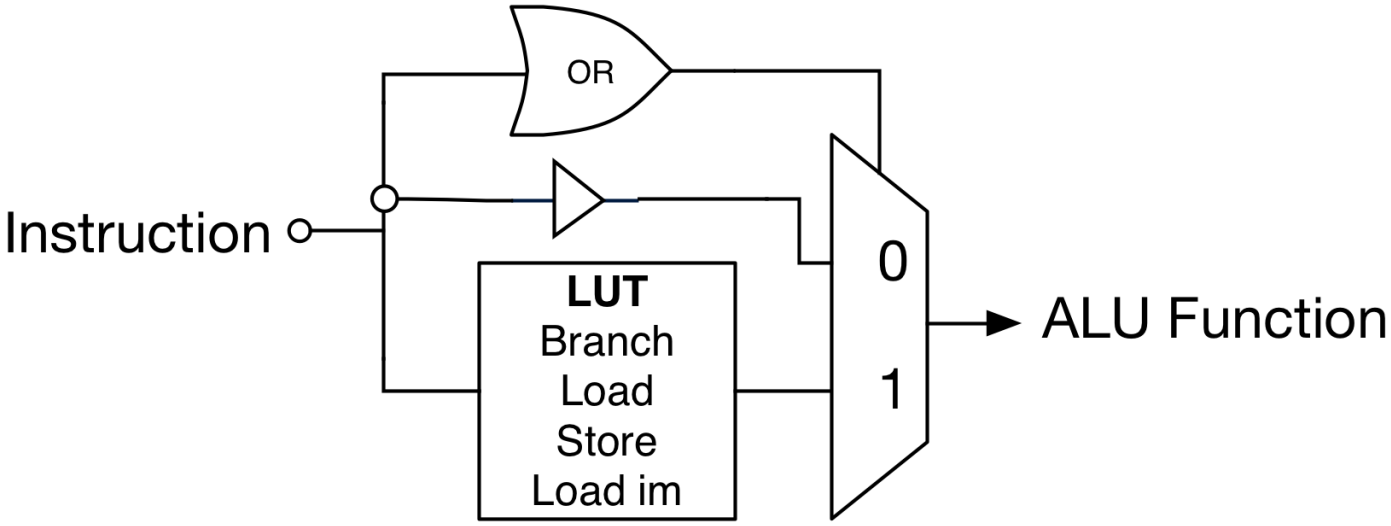

Figure 2.5: RTL sketch for the *ALU Control* entity

2.5 ALU Control

Inputs and Output

Inputs: Function (instruction[5-0]), ALU override
Output: ALU function

Function

As long as the ALU override is set to disabled, ALU Control will simply convert the function field from the instruction to the cor rect function of our *alu_funct_t* type, be it *alu_add*, *alu_sub* or any of the other implemented functions. The ALU Override signal however allows the decode to override the ALU operation, this is necesarry for functions such as load where the ALU has to compute an addition, but the instruction does not end with the add function value.

2.6 Decode

Inputs and Output

Inputs:	opcode, write_enable
Outputs:	reg_dest
	pc_control, mem_to_reg
	alu_override, mem_write_enable
	alu_src, reg_write_enable

Function

The *decode* entity is an asynchronous entity responsible for setting up the control-signals marked with a square in figure 2.1.

The functionality is equivalent to a lookup table for each of the signals. The signals are functions of *opcode* and *write_ enable*.

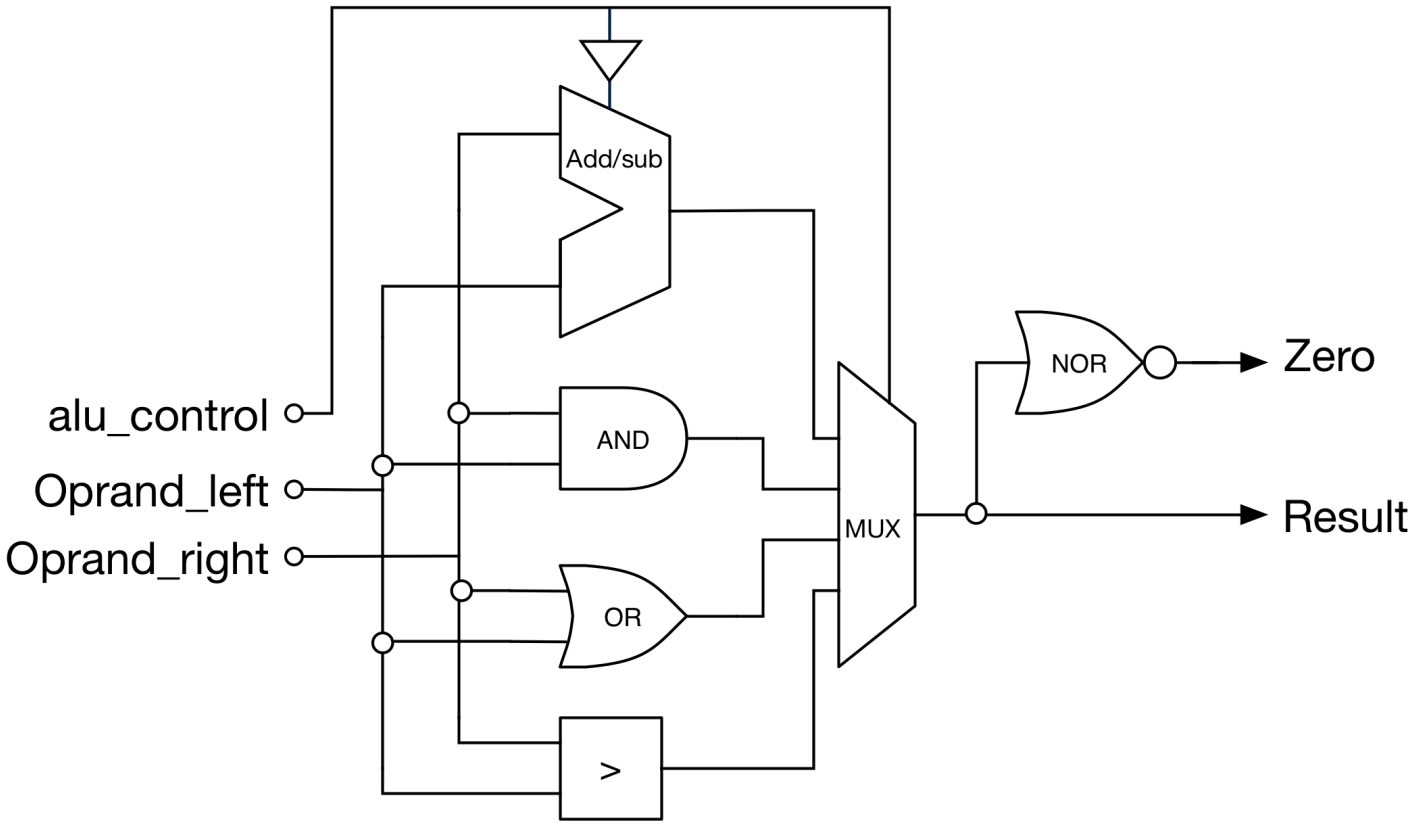

Figure 2.6: RTL sketch for the *ALU Control* entity

2.7 Arithmetic logic unit

Inputs and Outputs

Inputs: operand_left, operand_right, operator
Output: result, result_is_zero

Function

The arithmetic logic unit, ALU, is the core of the processors computing and is what preform the integer arithmetic and logical operations.

An ALU implemented were able to do addition and subtraction as well as shift left and logical “and” and “or”. An early RTL sketch of the planned implementation is shown in

figure 2.6.

The input contains a vector of which the operations will be preformed on. The outgoing resulting vector will be decided by the *ALU operation select* via a multiplexer. Then the output will hold the result chosen by the multiplexer. If all bits in the outgoing vector are zero the *Zero*-flag will be set high.

Chapter 3

Tests and results

There was made several test benches for all the essential and complex units in the design. In this chapter the tests will be explained.

3.1 Control

This test bench checked the expected behaviour of the output of the control unit both when the processor was enabled and disabled. Black box testing was used for both cases. It was assumed only valid values of the *opcode*.

During the testing of control unit when the processor was enabled, all valid values of *opcode* was tested with two or three clock cycles on each value corresponding to the specifications. Then the corresponding output was examined for correct output on right clock cycle. All tests were successful when the processor was finished. See figure A for a screenshot.

3.2 ALU

The ALU testbench tests all the functions supported by the ALU with multiple values. For each test, it verifies both the *result* and the *result_is_zero* flag. Care was taken when choosing numbers to tests. Interesting values that might lead to edge cases, such as adding 1 to the biggest number, using 0 and negative numbers was chosen.

We experienced no problems with the ALU and this test, and as the VHDL implementation is simple and uses functions from the *std_numeric* library, there was never anything wrong

with our operations, so the main purpose of this test became to make sure the correct operations are executed.

See screenshot at figure A

3.3 ALU Control

As the ALU control entity simply translates a std$_{l}ogicvectortoanenumandallowsabypass, thetestbenchsimplyte$

3.4 Decode

The decode test is an exhaustive table based test. It starts by checking that no opcodes enables any write signal if *write_enable* is not set. After which it goes through all our supported opcodes and verifies that the signals that matter for that opcode are being set, leaving room for don't care values. See screenshot A.

3.5 Top Level Test

The top level test that came with the exercise was used. It writes a 31 instruction long program to instruction memory and executes it, checking the data memory for the expected values.

One of the instructions, namely *add $10 $1 $10* reads from a register never assigned to. This is undefined in the design presented, see the discussion at 4.3.1. The test does not actually make any checks regarding this. See screenshot A.

3.6 Program Counter

The PC-test excercises the step, branch and jump functionality of the PC. See screenshot: A

3.7 Testing on the FPGA

Tested with the provided DummyArch implementation of the MIPS processor provided, the hostcomm utility was able to make a connection.

A rudimentary implementation was devised and uploaded. The hostcomm utility was unable to make a connection to the framework on the FPGA this time. The solution was to delete the xilinx project, create a new one, and re-add all files.

A custom program for processor was quickly written to test the design on the FPGA. The program consisted of repeatedly adding one (read from a memory location) to a register and writing the result to memory. The result of the test seemed to only affect the lowest byte. The problem was wrong endianness of the one-value in memory.

It was decided that the program in the handed out file tb_MIPSProcessor.vhd should be used for testing. The program was converted into binary and had its endianness reversed. It was uploded to the FPGA, and so was appropriate data. The run was succsessful as documented by Figure 5. The data fectched back is correct.

Chapter 4

Discussion

4.1 Entities

4.1.1 Arithmetic logic unit

The Arithmetic logic unit, ALU, though it is one of the most complicated parts of the processor, was made easy by the ieee.numeric_std library. The test for the ALU also make use of the same library. The synthesizer even removed the shift used for the load-upper-immediate instruction. Since it shifted a constant 16bit, it was reduced to just connection the bits to higher output bits, with ground on the lower.

4.1.2 Buffer vs. Out in Port Specifications

It was often useful to use the output signal of an entity in the entity itself. The alternatives considered were to either use an intermediate signal, or have the output to be a buffer. Though discommended by some, notably Xilinx, after much research, no good argument against using it in normal cases was to be found. However, in tri-state logic it might give inaccurate simulation results [2]. This is not a situation that is affecting the cases were buffers were implemented. No buffer related errors occurred while testing the processor.

4.1.3 The program counter

The program counter, though in theory simple, was a mayor source of problems to be considered.

On the first attempt, the *PC_latch* was clocked on the logical value *pc_update* emitted by the control entity. This gave warnings from the compiler, as this is bad practice. Instead it was necessary to use the global clock.

It is imperative that the PC updates at the beginning of the *fetch* state. This means that *pc_update* must be set in the previous cycle. An obvious solution would be to add an extra state. However, as this increases number of cycles used to process an instruction, this solution was not favored. Instead, with a slight increase in complexity, *pc_update* is set on either *execute* or *memory_wait*, whichever precedes the *fetch* state.

An alternative design that was considered was to set the *update_pc* signal in the $fetch$ state. While this means that the pc is actually changed at the end of the $fetch$ state, a bypass of the PC by the means of a mux when in fetch state would be introduced. This approach needed some consideration, as the signal would quite likely have time to make more than one round-trip propagating trough the add-one path (or maybe even branch, depending on the instruction). The proposed solution to this problem was to add a latch for the next PC signal, so that it would change at most once per cycle.

The choice was whether to introduce complexity in the PC or Control. Both solutions can be tested quite easily. Setting update-pc on fetch is more "aesthetically pleasing", but a latch and a mux is undoubtedly both more hardware-heavy and have slower propagation time than a few logical gates. Thus the former alternative was chosen on the merit of being more optimal in hardware.

4.2 Generics and use-directives

4.2.1 Jump

The handed-out code gives generics to the MIPSProcessor: ADDR_WIDTH and DATA_WIDTH. The task does not specify what these are used for, but it is easy to deduce that they are the address-size for both the instruction- and the data-memory and the size of the signed integers used as data.

In the RTL-sketch in the compendium, there is an entity that concatenates the top of PC with the 16-bit immediate value. If we assume that ADDR_WIDTH does not go above 16, we can leave this entity out. A reason for wanting to get rid of the generic, is that this facilitates writing nicer VHDL by allowing importing of a type defined in a central location. When using generics, it becomes necessary to write type-constraints many times. It is the difference between using *std_logic_vector(data_width)* and *data_t*.

4.3 The register file

The synthetized RTL showed that we got two ram-entities, each of the size of the whole register file. This is because one dual-port ram can at most service two simultaneous read or write requests. Our design needs to be able to simultaneously read two registers *and* write to another one. (The registers are free to all be the same one - that does not need special considerations.)

The number of rams could be reduced by introducing another state in addition to the execute, the commit state, in which we don't read, but only write to memory. In the execute state, we would not write to the ram. If we wanted to optimize for size on the FPGA, this alternative design might be viable, but we choose to go for a design with fewer cycles. This means more of the processor is in use simultaneously, giving us better speed *and* better power efficiency.

Studying the timing analysis, it was noted that the longest signal path goes trough our register-file. Adding this extra state would increase the max frequency of the processor. However, due to lack of time on the project, it is not known whether this increase would be enough to outweigh the extra state. It should also be noted that there are no clocks on the development boards that go faster that the current max frequency. One could be synthetized with a PLL, though.

The reason an ip-core was not used, is again because there is no standard RAM that can service our asynchronous requests.

4.3.1 The reset functionality in register-file

At first it was intended to reinitialize the register-file to all zeros upon reset. However, including the reset functionality caused the design to be rendered as a lot of flip-flops and big muxers. It also triggered warnings from the HDL Advisor in the compiler. See appendix B for more information. The compiler could not reduce the design to a dedicated RAM on a SLICEM on the FPGA. Instead it was rendered as using flip-flops. Not including the reset functionality rendered the register-file as using block RAM in the RTL.

Using a reset signal the entity was rendered with the utilization used in Table 4.3. Compared to not using a reset signal with an without reset (Table 4.1), it was using a very large amount of Slices. Using dedicated block RAM, the design was using a lot less of the FPGA and gave a mayor boost in the maximum frequency the design could be run at.

The decision was made that the values in the registers will be treated as undetermined until written to after a reset. This allows the design of the register to not include a reset. The

cost of this is of course that we introduce non-determinism to pathologic code that uses uninitialized registers.

Table 4.1: Fabric utilization for async, without reset RAM

Logic Utilization	Used	Available	Utilization
Number of Slice LUTs	112	9 112	1%
Number of fully used LUT-FF pairs	0	112	0%
Number of bonded IOBs	113	232	48%
Number of BUFG/BUFGCTRLs	1	16	6%

Table 4.2: Fabric utilization for sync, without reset RAM

Logic Utilization	Used	Available	Utilization
Number of Slice LUTs	2	9112	0%
Number of fully used LUT-FF pairs	0	2	0%
Number of bonded IOBs	113	232	48%
Number of BUFG/BUFGCTRLs	1	16	6%
Number of Block RAM/FIFO	1	32	3%

Table 4.3: Fabric utilization for async, with reset RAM

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	1024	18224	5%
Number of Slice LUTs	736	9112	8%
Number of fully used LUT-FF pairs	560	1200	46%
Number of bonded IOBs	114	232	49%
Number of BUFG/BUFGCTRLs	1	16	6%

4.3.2 Working, but suboptimal code

In the first stage of the implementation both test were run and messages and RTL from the synthetizer were inspected. Several instances of working, but suboptimal code were found:

The initial implementation of control had a extranous state with a empty implementation. The caused the design to render as using a RAM. There was once again observed a lack of don't-care notation in vhdl.

The RAM was initially not rendered as block RAM. This is discussed in a section 4.3.1.

4.3.3 Asynchronous components

At first, it was belived that not adding dependencies on the clock if not nessesary would give us simpler hardware. The the register-file was implemted with asynchronus reads. The RTL showed this as block RAM, however the HDL Advisor warned that the asynchrounus design causes the ram to be rendered as LUTs (See appendix C). Our design could not be reduced to a block-ram resource. This was resolved by clocking the reads. We now got a message confirming the reduction (See appendix D).

However it was now nessessary to add another state to the state-machine. This was again not done, because there was not enough time to create and test this system as well as sythesize a faster clock using a PLL. We would likely need to change the clock source to the 66MHz on-board clock. This again would have changed the timings of the uart in the hostcomm framework. This was deemed unfesable at this stage of the project.

Tables 4.1 and 4.2 show the difference in used hardware on the FPGA. As evident, there is not much difference, and the asynchronous version even has a lower propagation delay because it's missing a flip-flop. The way the rest of our system is designed, the asynchronous leads to best peperformance.

Chapter 5

Conclusion

A simple MIPS based processor has been implemented that successfully passes both the tests written specifically for this processor and the test handed out with the exercise and runs on the FPGA.

One of the main advantages of the processor is its minimalistic hardware. This was achieved by always thinking about the RTL implementation while writing the code.

The goal in mind was to implement a fast processor with a neat RTL and therefore hopefully generate relatively power efficient design.

Having strict specifications for all the modules defined in early RTL sketches, and having good test benches before implementing the design was important to make it possible to achieve these goals. We have learned a lot about the peculiarities of VHDL, the value of test benches and RTL sketches, as well as how to read and understand parts of the Xilinx reports.

The processor uses between two and three cycles per instruction, and with a final clock speed at 85 MHz, we think the performace is quite solid.

The actual complexity of the processor is 361 slice LUTS where 316 is used to implement logic. This means that less than 5 % of the FPGA's circuitry is in use.

In conclusion, it has been an intensive assignment that has taught us a lot that will be of tremendous help during the Computer Project as well as exercise 2. It was also the first processor we implemented, something which is insanely cool.

Bibliography

[1] *Lab Exercises in TDT4255 Computer Architecture*, Computer ArchitFeackturlteeat nfodrDinefsoigrnmGasrjounpsteknologi, Department of Computer and Information Science, October 7, 2014

[2] *Post on discussion on Out and Buffer*, ralblas@forum.xilinx, 19 October 2014, `http://forums.xilinx.com/t5/Synthesis/quot-buffer-quot-and-quot-out-quot-ports-in-XST-for-Spartan6/td-p/289517#U475618`

Appendices

Appendix A

Screenshots of Testresults

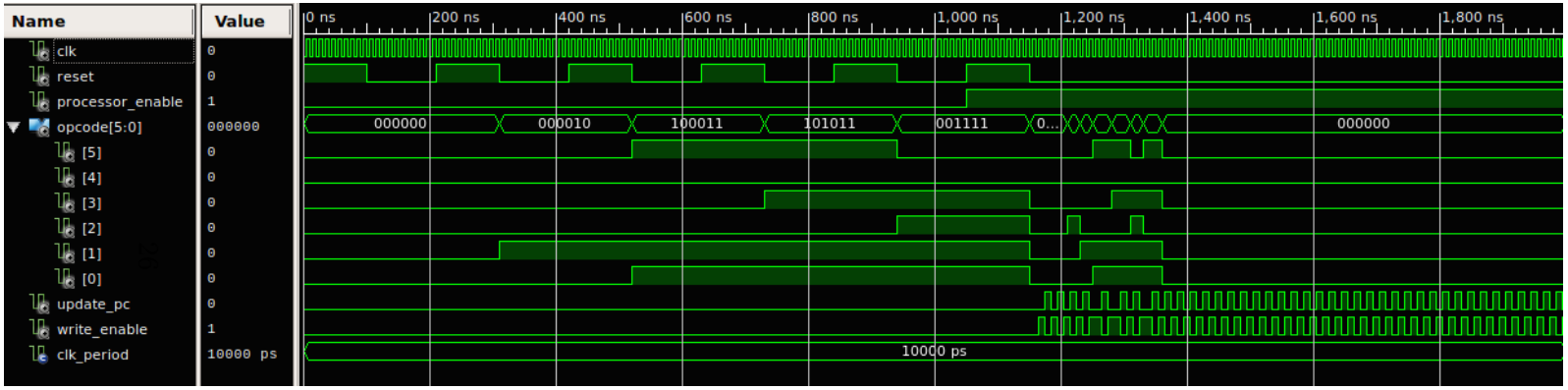

Figure A.1: Screenshot of waveform from the test bench for the control unit.

```
--------------------------------------------------------------------------
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
at 0 ps: Note: == Checking disabled processor == (/tb_control/).
Finished circuit initialization process.
at 110 ns: Note: opcode: 000000: add, sub, and, or, slt, sll (/tb_control/).
at 320 ns: Note: opcode: 000010: jump (/tb_control/).
at 530 ns: Note: opcode: 100011: load (/tb_control/).
at 740 ns: Note: opcode: 101011: store (/tb_control/).
at 950 ns: Note: opcode: 001111: load immediate (/tb_control/).

# run 1.00us
at 1050 ns: Note: === Testbench one passed successfully! === (/tb_control/).
at 1050 ns: Note: == Checking changes in the state machine. == (/tb_control/).
at 1210 ns: Note: Checking opcode 000100: beq branch if equal.   (/tb_control/).
at 1230 ns: Note: Checking opcode 000010: jump.   (/tb_control/).
at 1250 ns: Note: Checking opcode 100011: lw load word.   (/tb_control/).
at 1280 ns: Note: Checking opcode 101011: sw store word.   (/tb_control/).
at 1310 ns: Note: Checking opcode 001111: lui load upper imm.   (/tb_control/).
at 1330 ns: Note: Checking opcode 101011: sw store word.   (/tb_control/).
at 1360 ns: Note: Checking opcode 000000: ALU operation (and, or, add, sub, slt, sll).   (/tb_control/).
at 1380 ns: Note: === Testbench two passed successfully! === (/tb_control/).
at 1380 ns: Note: === Tests Sucess! === (/tb_control/).
```

Figure A.2: Screenshot of output from the test bench for the control unit.

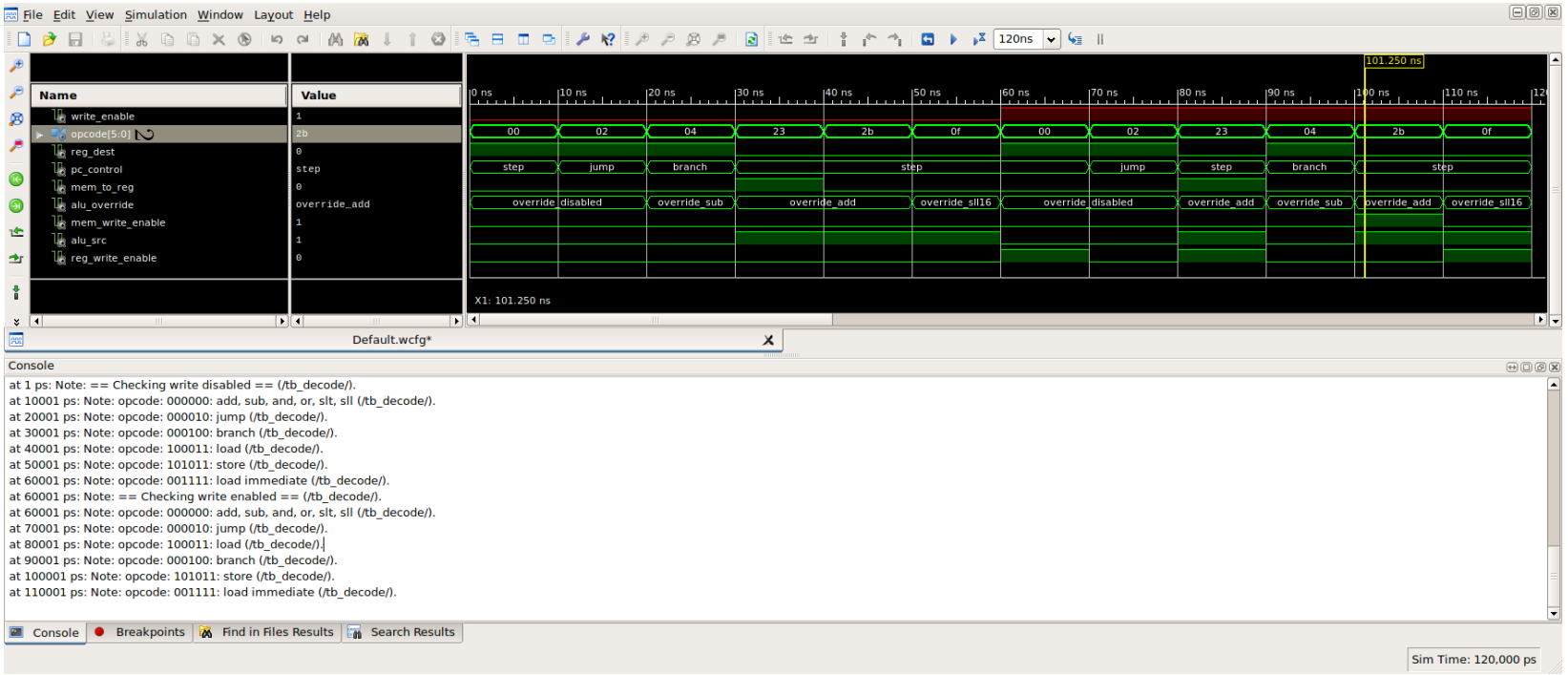

Figure A.3: Screenshot of the waveform and output of the *decode* test bench

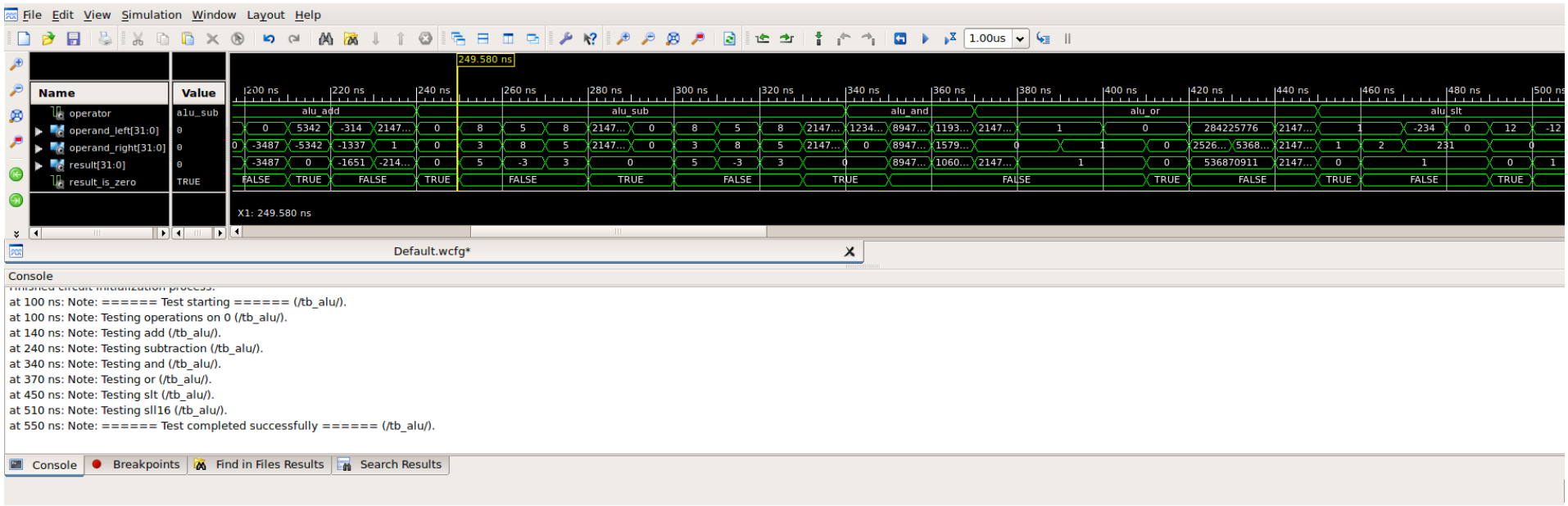

Figure A.4: Screenshot of the waveform and output of the ALU test bench

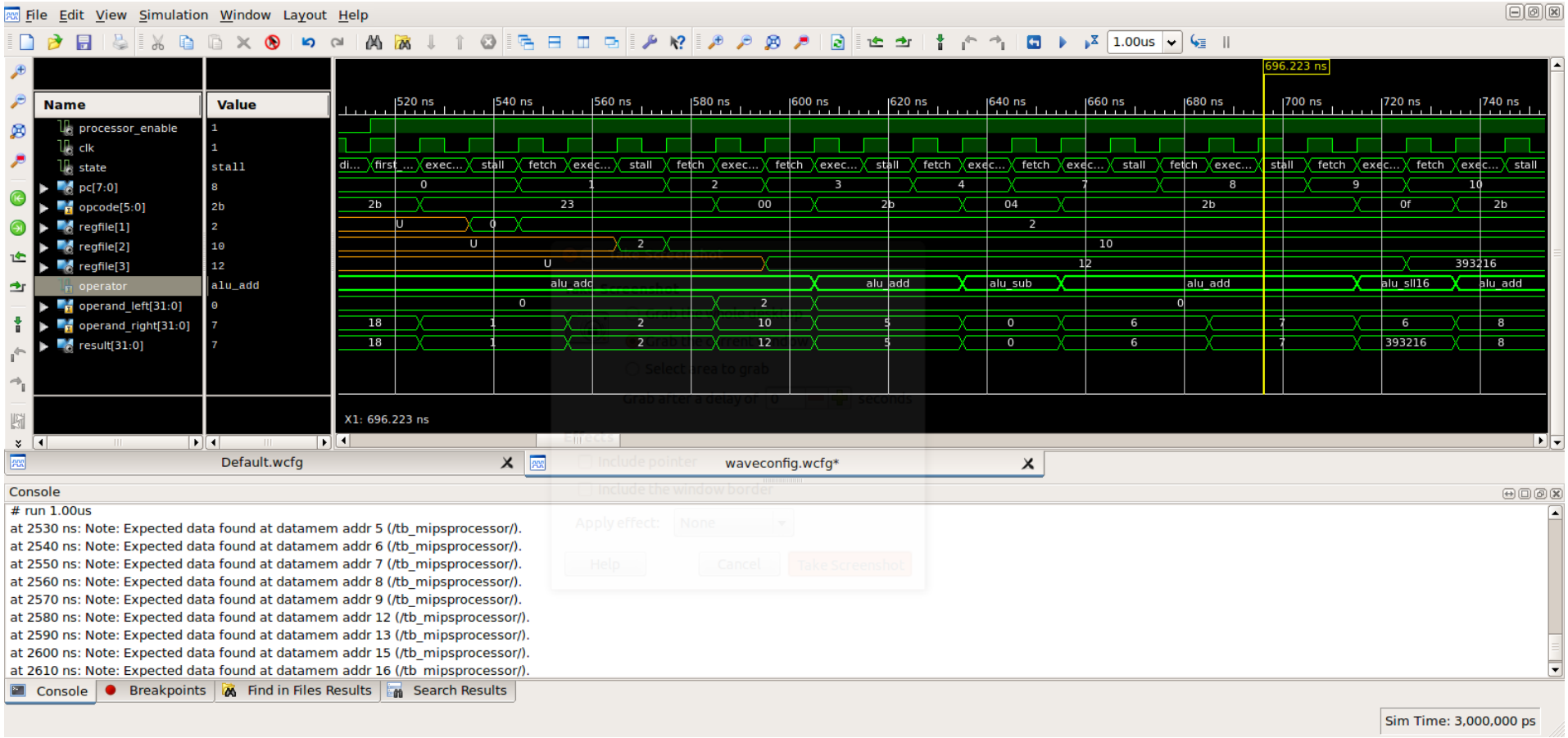

Figure A.5: Screenshot of the waveform and output of the top level test bench

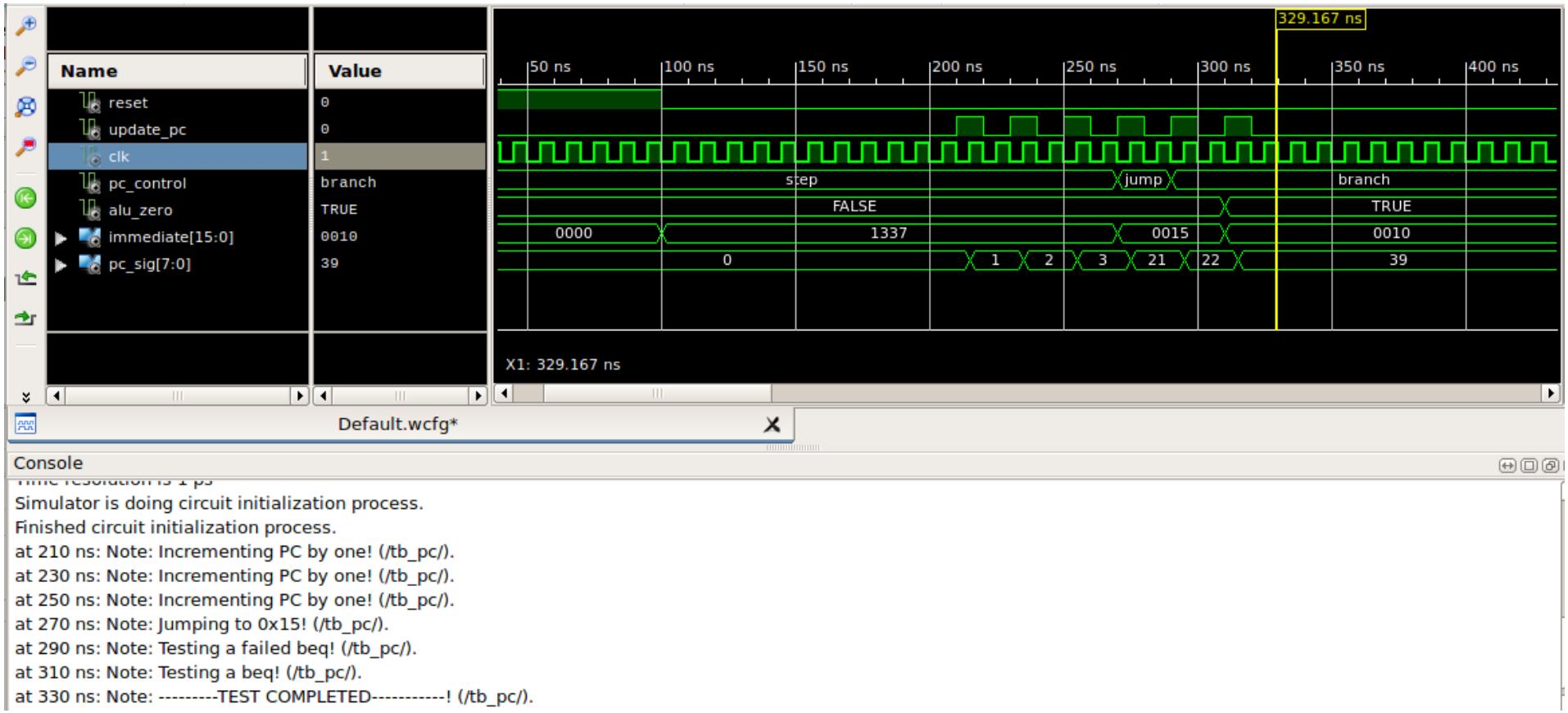

Figure A.6: Screenshot of successful testbench of the program-counter entity testing step, branch and jump

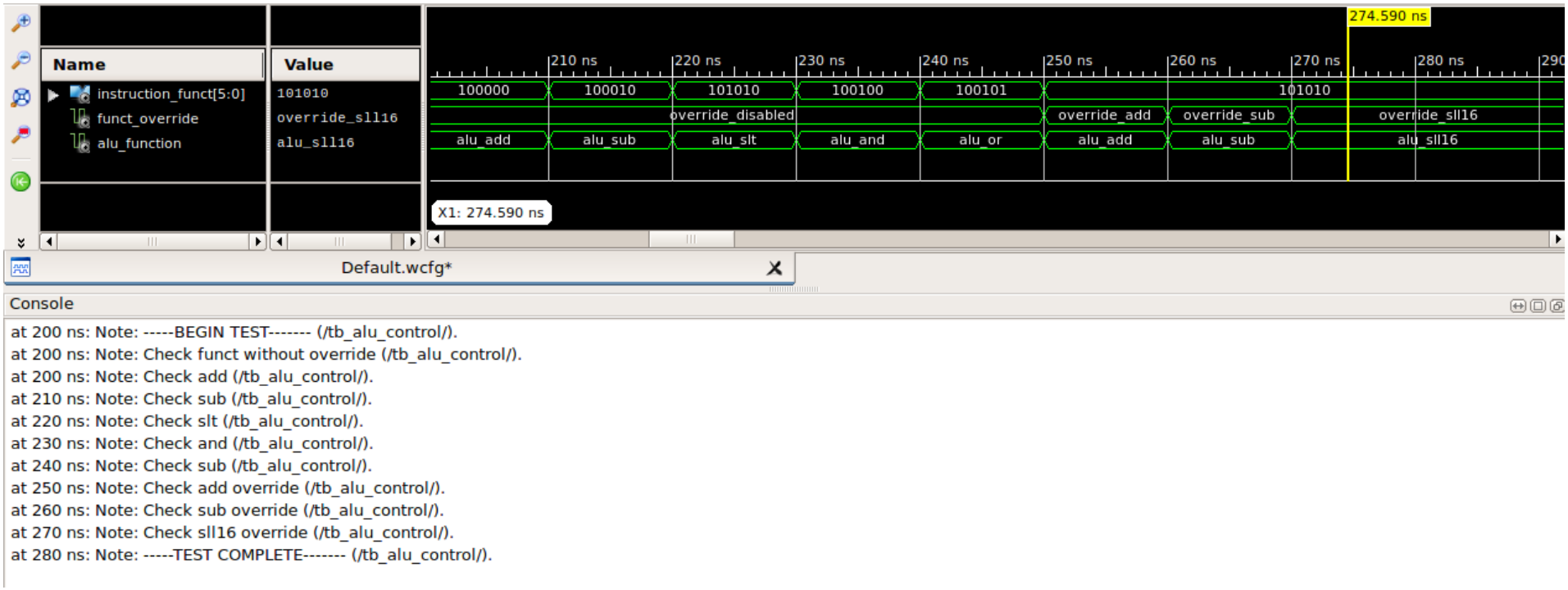

Figure A.7: Screenshot of successful testbench of the alu control entity, checking translation to function enum and override functionality

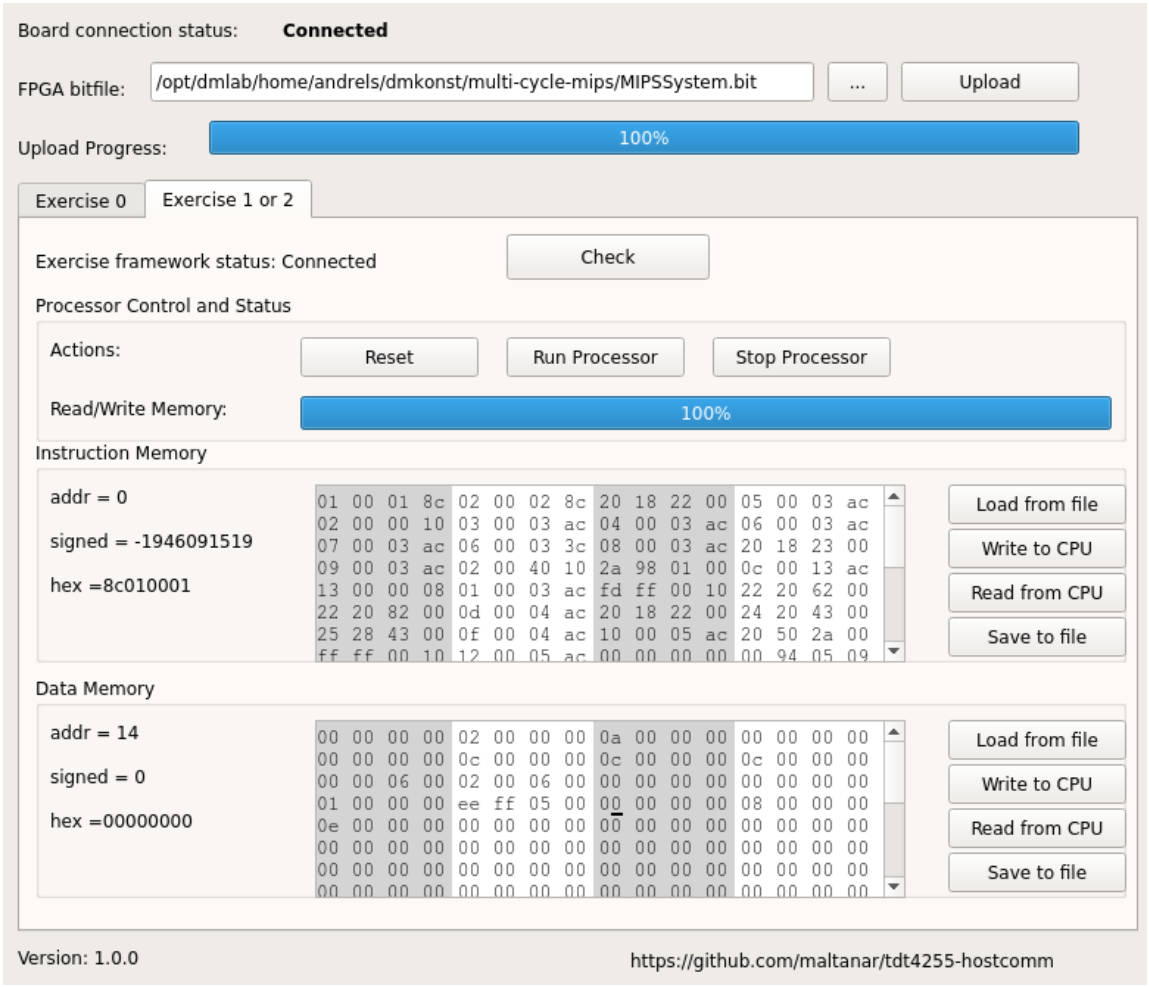

Figure A.8: Screenshot of the the hostcomm utitlity after successful run of the program in tb_MIPSProcessor.vhd

Appendix B

Xst:3019 - HDL ADVISOR

Xst:3019 - HDL ADVISOR - 1024 flip-flops were inferred for signal <regfile>. You may be trying to describe a RAM in a way that is incompatible with block and distributed RAM resources available on Xilinx devices, or with a specific template that is not supported. Please review the Xilinx resources documentation and the XST user manual for coding guidelines. Taking advantage of RAM resources will lead to improved device usage and reduced synthesis time.

Appendix C

Xst:3031 - HDL ADVISOR

Xst:3031 - HDL ADVISOR - The RAM <Mram_regfile> will be implemented on LUTs either because you have described an asynchronous read or because of currently unsupported block RAM features. If you have described an asynchronous read, making it synchronous would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding guidelines.

Appendix D

Xst:3040 - HDL ADVISOR

Xst:3040 - The RAM <Mram_regfile> will be implemented as a BLOCK RAM, absorbing the following register(s):

Post-PAR Static Timing Report:

Timing summary: ————————

Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)

Constraints cover 424283 paths, 0 nets, and 3150 connections

Design statistics: Minimum period: 11.764ns1 (Maximum frequency: 85.005MHz)