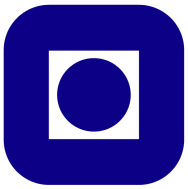

NTNU

TDT4255 - Computer Design

Exercise 1

Multi-Cycle MIPS Processor

Mathias Ose, Jonas Halland, Jonatan Lund

October 16, 2015

Abstract

In this exercise the group created a VHDL implementation of a multicycle processor capable of executing a subset of the MIPS instruction set. The processor was written such that it could integrate with other provided modules, which facilitated testing on an FPGA. The same program was tested first in a testbench simulation, then on the FPGA. After successfully implementing and testing the required subset of intructions, some additional instructions were also added to the processor capabilitites.

Contents

1 Introduction **1**
- 1.1 MIPS Instruction Set . . . 1
- 1.2 Approach . . . 1

2 Solution **2**
- 2.1 Development . . . 2
- 2.2 Top Level Architecture . . . 2
- 2.3 Control Submodule . . . 2
- 2.4 Registers Submodule . . . 3
- 2.5 Immediate Value Transform Submodule . . . 3
- 2.6 Arithmetic Logic Unit . . . 3
- 2.7 Program Counter Submodule . . . 4
  - 2.7.1 Limiting Output . . . 4
- 2.8 Adding More Instructions . . . 5
  - 2.8.1 Arithmetic and Logical I-Type Instructions . . . 5
  - 2.8.2 Shift Instructions . . . 5

3 Results **9**
- 3.1 Testing . . . 9
  - 3.1.1 Unit Testing . . . 9
  - 3.1.2 System Testing . . . 9
- 3.2 Instruction Set Architecture . . . 9

4 Discussion **12**
- 4.1 Unit Testing . . . 12
- 4.2 System Testing . . . 12
- 4.3 "Load Immediate" . . . 12
- 4.4 Performance . . . 12
- 4.5 Progress . . . 13
- 4.6 Further Work . . . 13

5 Conclusion **14**

A Suggested Architecture **15**

Acronyms

ALU Arithmetic Logic Unit.

FPGA Field Programmable Gate Array.

MIPS Microprocessor without Interlocked Pipeline Stages.

PC Program Counter.

RTL Register Transfer Language.

VHDL Very-Large-Scale Integration (VLSI) Hardware Description Language.

1 Introduction

This report presents a solution to exercise 1 of the TDT4255 course at NTNU. The objective of the exercise is to implement a multi cycle processor capable of executing a subset of the MIPS instruction set, in VHDL. After the design simulation is verified with testbenches in *XILINX ISim* it is to be synthesized and tested on a *XILINX Spartan-6 FPGA*.

1.1 MIPS Instruction Set

The instruction set for this exercise is MIPS instruction set with 32 bit wide instructions. MIPS is a RISC instruction set, and there are three different instruction formats: R(egister)-type, I(mmediate)-type and J(ump)-type. The MIPS processor has a few fast registers, and uses load and store instructions to move data to and from a bigger but slower data memory. Instructions are fetched from a memory separate from the data memory.

1.2 Approach

The top level architecture of the processor is divided into separate modules. These modules are created by following test-driven development methods and iterative changes as the system comes together as a whole. Once these submodules are finished they are connected and the system is tested as a whole.

2 | Solution

2.1 Development

For these exercise the primary tools used were:

- *XILINX ISE* for writing and syntax checking VHDL
- *XILINX ISim* for testing simulated modules in testbenches
- *hostcomm*[3] for testing the synthesized processor on an FPGA
- *git* for collaboration and version control
- *draw.io* for sketching components for the report

2.2 Top Level Architecture

A VHDL project was provided for the exercise, containing instruction and data memory implementations, a *MIPSProcessor* VHDL module without a functioning implementation, and a *MIPSSystem* VHDL module connecting the processor module to the memories.

The scope of the exercise was to create the *MIPSProcessor* implementation. An RTL sketch of a suggested architecture was provided (see appendix A). The implementation described in this report is closely based on this suggested architecture.

Most of the functionality of the processor was implemented in appropriate submodules. On the top level the processor only consists of signals connecting ports of different submodules, and a few multiplexers for directing signals based on other signals.

An overview of the processor architecture can be seen in figure 2.1.

2.3 Control Submodule

The control submodule combines internal state (`FETCH`, `EXECUTE`, `STALL`) and the input from the current instruction to decide various control signals which the other submodule use as inputs.

During the fetch stage the control module does nothing except prepare to go to the next state at the next clock cycle. It is during the execute stage that most of the output happens. In this stage the state machine can be viewed a *Mealy machine*, with the `opcode` and `funct` bits from the input instruction deciding the outputs. The outputs that are set non-default are shown in figure 2.2.

The LW and SW instructions require I/O to the data memory module, which is a comparatively slow operation. Therefore the stall state is required as a continuation of the execute state for these instructions.

The pc_write output controls the program counter submodule and ensures that the next instruction is not fetched before the current instruction has had enough time to execute.

2.4 Registers Submodule

The register module contains an array of 32 general purpose registers. Each register is 32 bits. Both the R-type and I-type instructions read and write these registers. Register 0 is reserved, and should always contain the value 0.

Data can be written to a register by enabling register_write. The input value of write_data will then be written to the register addressed by write_register.

The two outputs of the module do not require enabling, and will constantly output the value addressed by the corresponding address input.

The labeled inputs and outputs of the module can be seen in figure 2.1.

2.5 Immediate Value Transform Submodule

In the suggested architecture there was indicated a module for sign extending a 16-bit vector to 32 bits. This 16-bit vector comes from the immediate value part of I-type instructions. At first we implemented this without a submodule, since it only required a one-line function call in VHDL to extend the number, as shown in listing 2.1.

Listing 2.1: Extending a vector in VHDL

```
extended <= operand_t(resize(unsigned(imm_val), operand_t'length));
```

As we made progress on the exercise, we realised that the LUI instruction required a feature that could extend a 16-bit vector to 32 bits by left-shifting, placed in the same location in the datapath as the sign-extending feature. The natural solution to this was to create a new submodule which could take a 16 bit input and a 1 bit control signal, then output the appropriate extended vector. Figure 2.3 shows a sketch of this module.

2.6 Arithmetic Logic Unit

The implemented ALU can perform arithmetic, logical and shifting operations. In each execute cycle the control module decodes the instruction to an appropriate opcode and the ALU receives this opcode as an input. Table 2.1 shows the operations implemented in the ALU.

The arithmetic and logical operations take two 32 bit operand inputs. The first is always a register value, the second may be either from a register or an immediate value from the instruction that has been extended to 32 bits. These operations are implemented by delegating to VHDL built in operations.

Shift operations use the `shift_amount` part of the instruction as another input. Shifting is implemented by delegating to the standard shift functions in VHDL. These instrutions were added later than the others, as described in section 2.8.

The ALU module outputs a signal to the PC module called `alu_zero` whenever the current operation yields a 0 result.

ALU operation	Description
ADD	Addition
SUB	Subtraction
SLT	Set if less than
AND	And
NOR	Not or
XOR	Exclusive or
SLL	Shift left logical
SRL	Shift right logical
SRA	Shift right arithmetic

Table 2.1: Implemented ALU operations

2.7 Program Counter Submodule

The implementation requires a special purpose register for the program counter. The PC register has been placed in a submodule which is also called PC, together with the logic for changing the value of the register based on inputs.

When the conrol unit reaches the `FETCH` state, this register must contain the address of the next instruction to be executed. The PC register can only change value when the control module enables the `pc_write` signal, which it does in the `EXECUTE` or `STALL` states. This way the register will have a new value by the time the control module goes to the `FETCH` state, and the instruction memory can use the cycle to transfer the next instruction.

By default the program counter increases by one every cycle it is write enabled. The processor implementation also supports *jump* and *branch* instructions. Both of these may overwrite the current PC value with a new value. In the case of the `beq` instruction both the `branch` signal from the control module and the `alu_zero` signal from the ALU must be high or else the default incrementing behavior is used. Figure 2.4 shows the multiplexing for this.

In this implementation the program counter is initialized to its maximum value on reset. When the processor is enabled, the first thing to happen is that the program counter increments, overflowing to the 0 address. The control module goes to `FETCH`, and program execution may begin.

2.7.1 Limiting Output

When the submodule was first implemented, the register was set to be 32 bits wide and the operations related to the register were implemented accordingly. This was based on the suggested architecture sketch and the MIPS reference[1, Appendix D]. When integrating the submodule with the rest of the systems we realized that the system expected a generic

address width, and that for the testbench this value was set to 8. The solution to this was to add a "chopping" feature before the output of the module that discards higher order bits so that only the desired number of bits is outputted. All internal arithmetic in the module still operates on 32 bit values, and only the output is chopped.

2.8 Adding More Instructions

After having sucessfully tested the minimum required instruction set on the FPGA, we took another look at the full MIPS instruction set.

We looked up the binary values of instructions and functions online[2], then implemented some of them.

2.8.1 Arithmetic and Logical I-Type Instructions

With an already functioning implementation containing various ALU operations and I-type insructions, it was very simple to also implement the I-type versions of the existing arithmetic-logical operations, `ADDI`, `ANDI`, `ORI` and `SLTI` by setting the correct signals in the control module. With `NOR` and `XOR` existing as operators in VHDL, these could be implemented as ALU operations just as simply as the existing ones, which enabled the instructions `NOR`, `XOR` and `XORI`

2.8.2 Shift Instructions

In order to implement the shift instructions `SLL`, `SRL`, the ALU needed access to the part of the instruction that contains the number of positions to shift. This signal was added and can be seen in figure 2.1. With that connection it was again a very simple task to add the shift functionality to the ALU, since VHDL provides functions for this. Listing 2.2 shows the syntax for shift operations in VHDL.

Listing 2.2: Shifting a vector in VHDL

```
r <= operand_t(shift_left(unsigned(vec), to_integer(unsigned(shamt))));
r <= operand_t(shift_right(unsigned(vec), to_integer(unsigned(shamt))));
r <= operand_t(shift_right(signed(vec), to_integer(unsigned(shamt))));
```

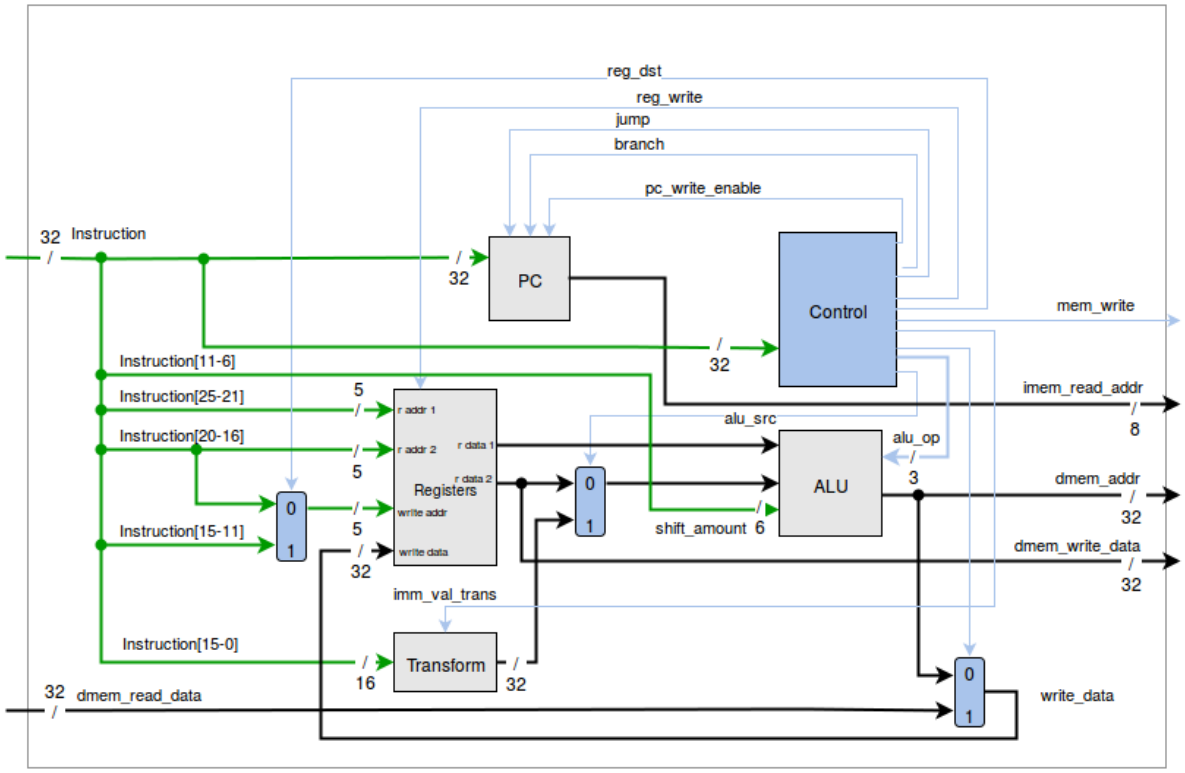

Figure 2.1: Top level architecture of processor

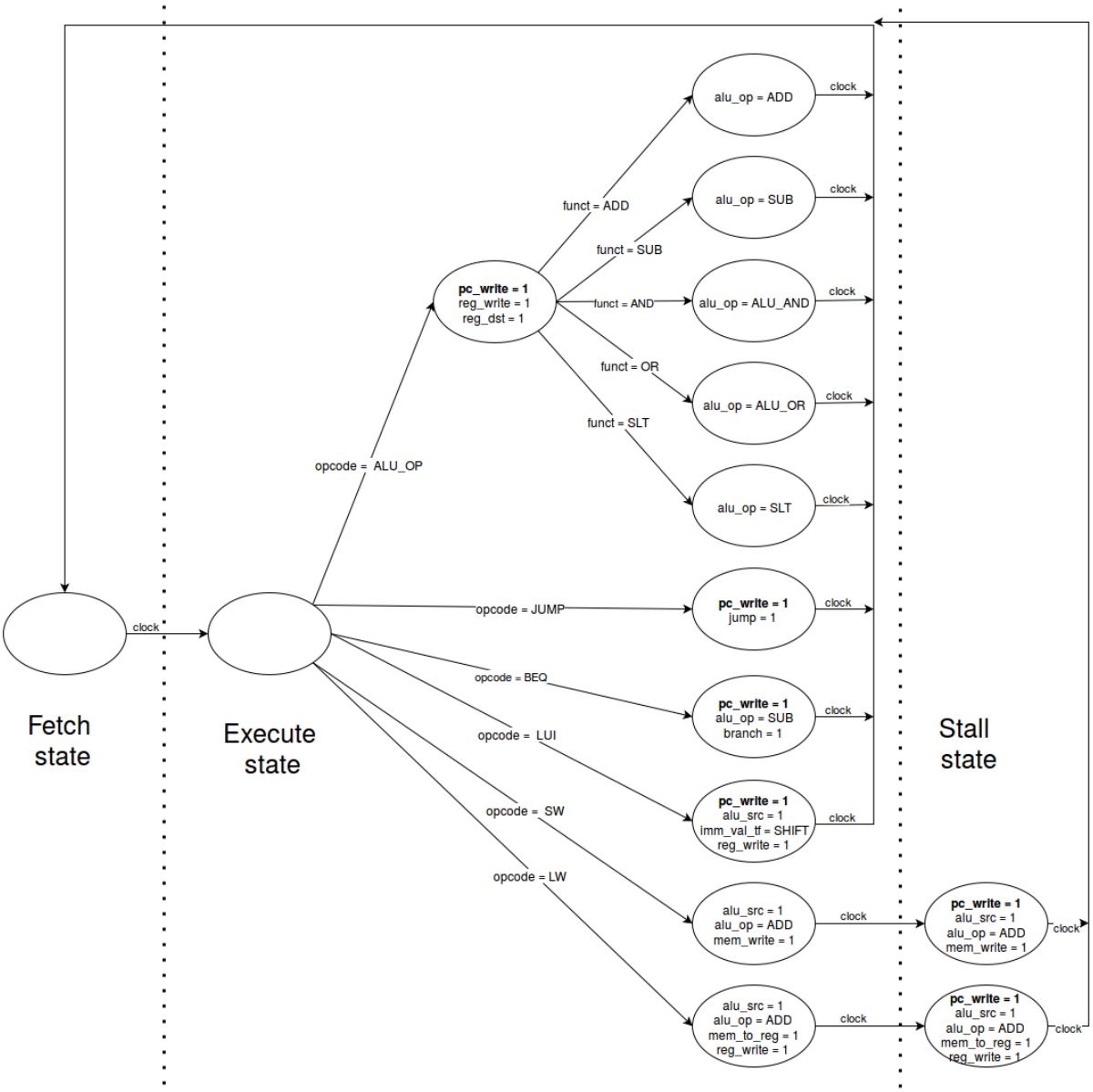

Figure 2.2: State machine of control submodule.
Binary signals default to 0, `ALU_OP` defaults to `NO_OP` and `imm_val_tf` defaults to `SIGN_EXTEND`.

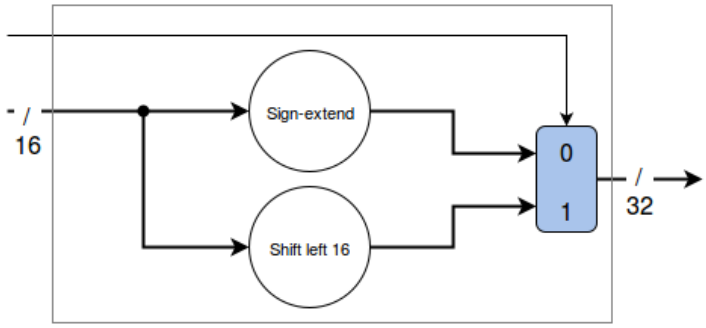

Figure 2.3: Sketch of immediate value transform submodule

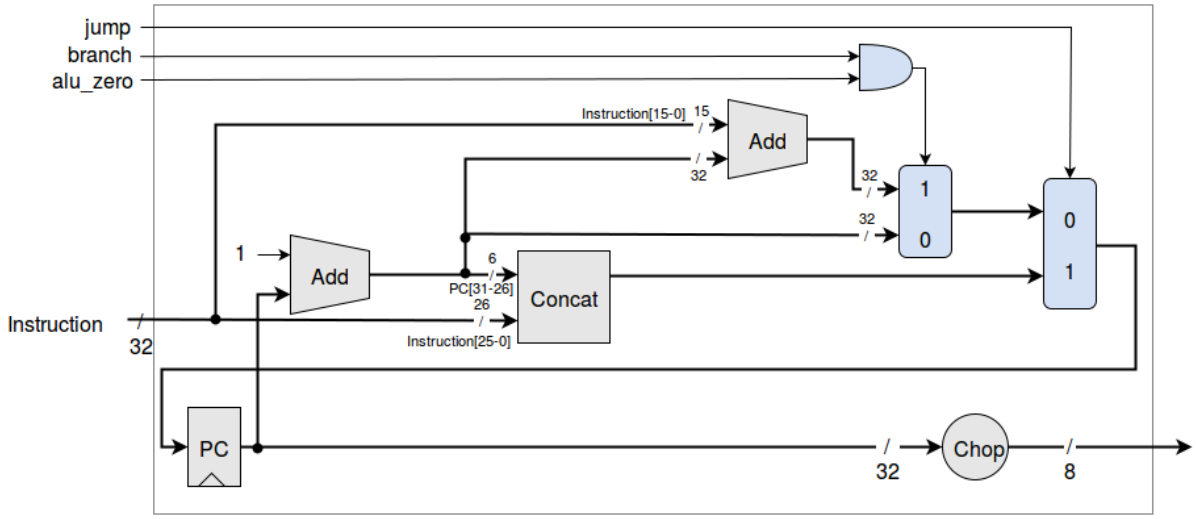

Figure 2.4: Program counter submodule sketch

3 Results

3.1 Testing

The implementation was tested at different levels. Each submodule was written with its own unit test. When put together they were verified as a system. Finally the implementation was also tested on an FPGA.

3.1.1 Unit Testing

Each VHDL submodule implemented has its own testbench. We tried following test-driven development practices by writing failing tests first, then implementing the feature(s) to make the tests pass. Tests were written based on our assumptions and since we lack experience, sometimes needed to be changed as we learned more and corrected our implementation.

3.1.2 System Testing

tb_MIPSProcessor

A testbench for the processor was provided. It loads data and instruction to memory, lets the processor run for a while, then checks that the data memory contains what it expects. Our processor implementation passes all the tests of this testbench. Figure 3.1 shows a screenshot from this simulation.

On the FPGA

After passing all the tests of `tb_MIPSProcessor`, the same program was adapted to make it compatible with the *hostcomm* tool. The *MIPSSystem* was synthesized and a bitfile was created. After uploading the bitfile, program and data to the FPGA, the processor was started then stopped after a short while. The data memory was read back and inspected, and we could see that it matched the expected output from the tesbench. Figure 3.2 shows a screenshot of this.

3.2 Instruction Set Architecture

The implemented instruction set is a subset of a full MIPS instruction set. All the implemented instructions can be seen in table 3.1.

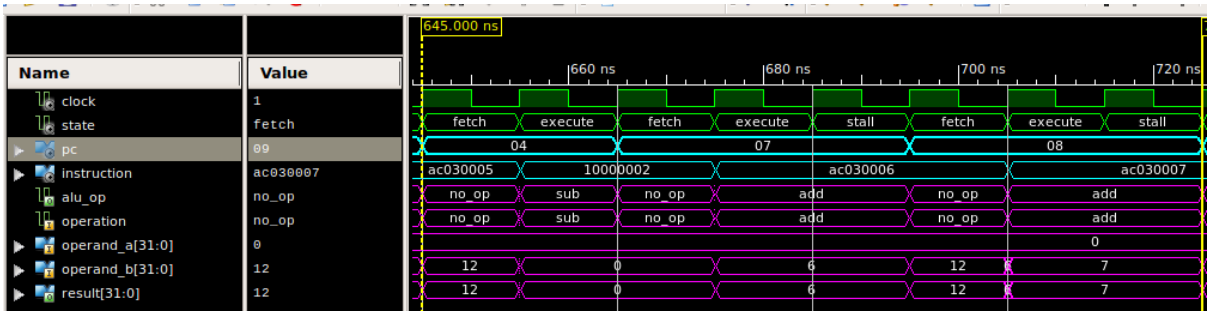

Figure 3.1: Showing ALU-related signals while *ISim* simulates the instructions

```
beq $0, $0, 2
sw $3, 6($0)
sw $3, 7($0)
```

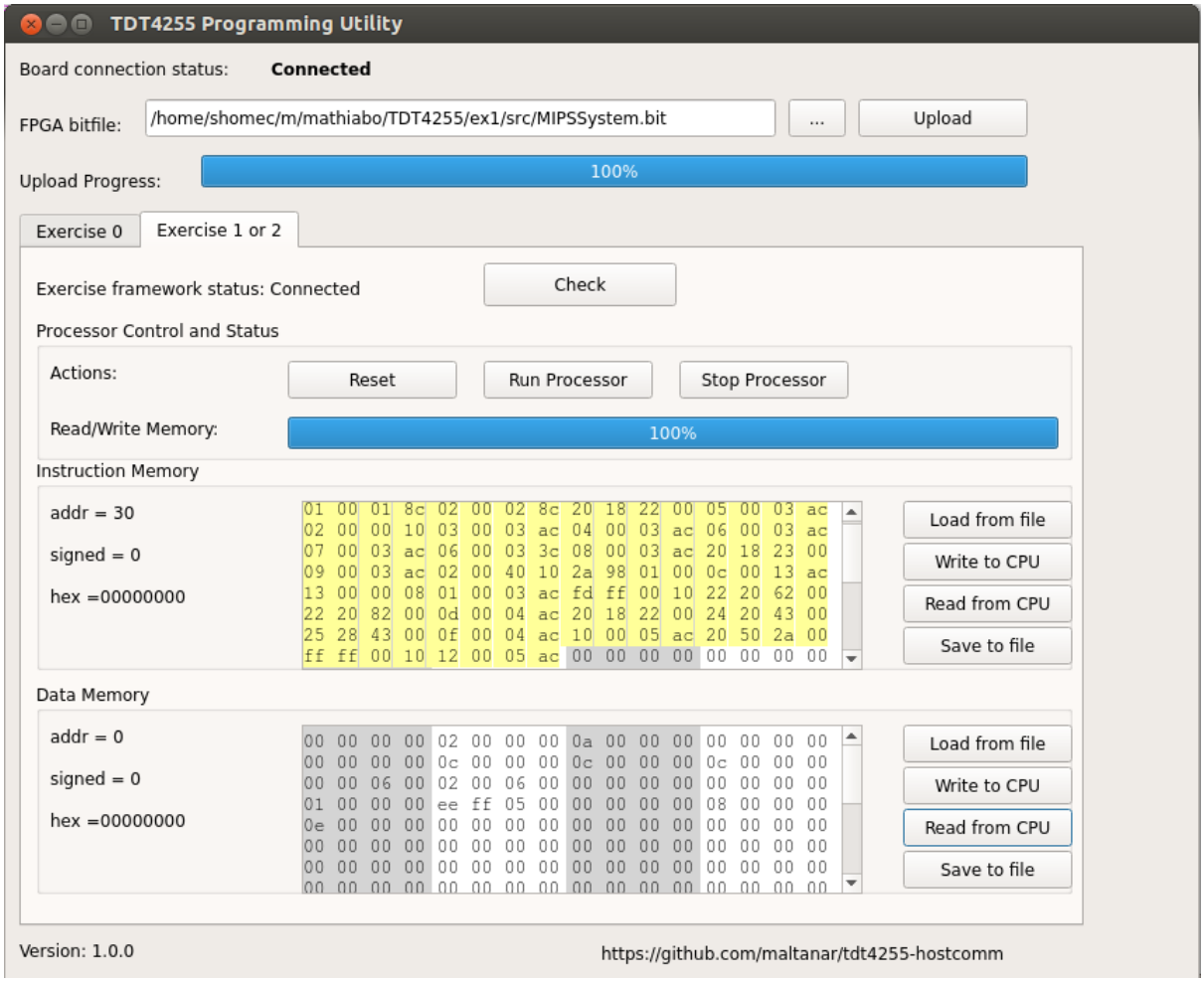

Figure 3.2: *hostcomm* after running the processor on the FPGA. The values in the data memory has been read from CPU.

Instruction	Type	Required
ADD	R	X
SUB	R	X
ADDI	I	-
AND	R	X
OR	R	X
NOR	R	-
XOR	R	-
ANDI	I	-
ORI	I	-
XORI	I	-
SLL	R	-
SRL	R	-
SRA	R	-
SLT	R	X
SLTI	I	-
J	J	X
BEQ	I	X
LUI	I	X
LW	I	X
SW	I	X

Table 3.1: Implented instruction set

4 Discussion

4.1 Unit Testing

New features were mostly developed by first writing failing tests, then implementing the features required to pass the tests. Sometimes the test were later found to be flawed and had to be adapted. Despite this, having test was very useful, to be able to run at any time and check that new functionality did not break old functionality.

4.2 System Testing

After implementing each component separately, the system was tested as a whole. During this stage the system testbench showed unexpected behaviour. Initially it was really hard to find where the error originated from, but as we gained more experience with the simulator, errors became easier to trace. Most of the errors were logical errors: wrong status signals, wrong MUX order and missing signals in sensitivity list.

4.3 "Load Immediate"

The task description required the "LDI" instruction to be implemented. There is a pseudo-instruction called "li" in the MIPS reference card [1], but since it is a pseudo-instruction it is not within the scope of the processor implementation. After asking for a clarification we decided that implementing the `LUI` instruction would suffice as the minimum requirement, and by later adding other instructions like `ADDI` and `SLL` we made sure that a compiler could in fact translate the pseudo-instruction to some combination of supported instructions.

4.4 Performance

The final implementation may achieve a maximum clock frequency of $46.430MHz$ according to the synthesis report. The processor is not pipelined and only handles one instruction at any given time, for multiple cycles, meaning it finished one instruction every second or third clock cycle, depending on whether the instruction accesses the data memory or not. The implemented processor may execute 23.215 million instructions per second in best case scenario, and 15.477 million in the worst case.

4.5 Progress

There were no big technical innovations required, the main obstacle that had to be overcome was to learn to use VHDL and the *XILINX* tools.

The submodules were created first, with testbenches based on our own assumptions. This was relatively simple and seemingly worked well, but as we were expecting, when the modules were connected the main testbench revealed a lot of problems. Many hours were spent inspecting the waveform simulations in *ISim* and different changes were tested. One by one the errors were identified and corrected, and a working implementation eventually emerged.

At this point we had learned a lot and had an intimate understanding of our own implementation, so adding more instructions at this point was a breeze.

4.6 Further Work

If we had more time we could take a closer look at optimizing. Possible optimizations include hardware utilization, clock frequency and energy efficiency.

Test coverage could definently be improved. The "extra" instructions that were added last are not covered by the main testbench program. We have added some tests, for example for the new ALU operations, but had to prioritize report writing rather than extensive testing.

In the next exercise the implementation will be modified to use a pipelined architecture, which should improve the clock frequency significantly.

5 Conclusion

The requirements for implemented instructions were fulfilled. All tests in the provided testbench were successful, and the MIPS program from the testbench was tested on the FPGA with the *hostcomm* tool, where it produced the same output as on the testbench.

Additional instructions beyond the minimum requirement were also implemented and verified with testbenches to some extent.

The exercise was a very good learning experience. The group members went from a very limited understanding of VHDL and processor design, to being able to find and reason about errors and easily implement more features in the existing design. *ISim* was especially useful as soon as we understood how to utilize it during debugging.

A | Suggested Architecture

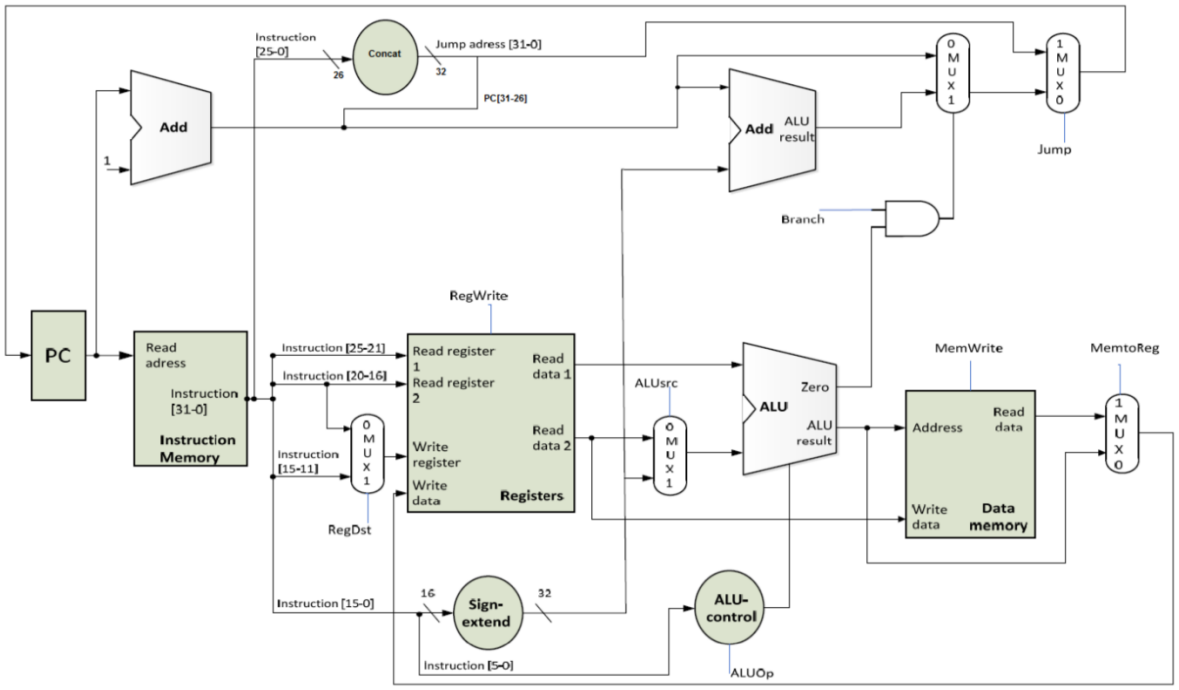

Figure A.1: The suggested architecture that was provided

References

[1] Computer Architecture and Design Group. Lab exercises in tdt4255 computer design, 2015-08-26.

[2] OpenCores.org. MIPS opcodes reference. `http://opencores.com/project,plasma,opcodes`, 2015.

[3] Yaman Umuroglu. tdt4255-hostcomm. `https://github.com/maltanar/tdt4255-hostcomm`, 2014-08-19.