NTNU

TDT4255 - Computer Design

Exercise 2

Pipelined MIPS Processor

Mathias Ose, Jonatan Lund

November 13, 2015

Abstract

In this exercise the group builds on the VHDL MIPS processor from exercise 1, removing the multi-cycle aspect and instead implementing pipelining. In order to mitigate hazards, data-forwarding, stalling and pipeline flushing is implemented. As before, the design is validated both with simulated testbenches and physically on an FPGA.

Contents

1 Introduction **1**

1.1 Five-stage MIPS Pipeline 1
1.2 Pipeline Hazards 2
1.3 Exercise Requirements 2

2 Solution **3**

2.1 Development 3
2.2 Pipelined Design 3
2.2.1 Pipeline Stages 3
2.2.2 Information-passing Between Stages 4
2.3 Pipeline Hazards 5
2.3.1 Structural Hazards 5
2.3.2 Data Hazards 5
2.3.3 Control Hazards 7

3 Results **8**

3.1 Testing 8
3.1.1 Simulated Testbenches 8
3.1.2 Testing on an FPGA 10
3.2 Measured Performance 10

4 Discussion **11**

4.1 Test Coverage 11
4.2 Progress 11
4.3 Performance 11
4.4 Further Work 12

5 Conclusion **13**

A Architecture Sketches **14**

B Instruction Set Architecture **16**

1 Introduction

This report presents a solution to exercise 2 of the TDT4255 course at NTNU. The exercise builds on the solution from exercise 1 [3], a VHDL implementation of a multi-cycle processor capable of executing a subset of the MIPS instruction set. In this exercise, the multi-cycle aspect is removed, and instead a pipelined design is implemented.

In addition to our own code from exercise 1, exercise 2 also builds on what was provided with exercise 1. This includes a *MIPSSystem* VHDL module that the processor is a submodule in, which connects the processor to data and instruction memories and facilitates system testing in a provided testbench and on an FPGA.

1.1 Five-stage MIPS Pipeline

The MIPS instruction set is designed to enable pipelining. The pipeline that is described in this report is a typical MIPS pipeline, with the following five stages:

Instruction Fetch (IF)
In this stage the instruction memory receives an address and outputs the corresponding instruction.

Instruction Decode (ID)
In this stage the instruction that was read in the IF stage is decoded. Control signal values are determined here that will be used in the following pipeline stages.

It is also in this stage that registers are read, and the read values are passed to the next stage.

Execute Instruction (EX)
In this stage the ALU receives its input values and operation and outputs a result.

Memory Access (MEM)
In this stage a value can be either read from or written to data memory.

If the instruction is a jump or a branch that is taken, this is the stage where the new program counter will be set.

Write to Registers (WB)
In the final stage of the pipeline, a new value may be written to its destination register.

1.2 Pipeline Hazards

When implementing pipelining in a processor, there are complications that might arise that must be handled to ensure correct results. Instructions often depend on preceding instructions, and if two instructions with some dependency between each other are in the pipeline at the same time, there is a *hazard* which must be handled in a some way. *Patterson & Hennessy* lists the following pipeline hazards [2, Chapter 4.5]:

Structural Hazards
Instructions that access the same hardware component cannot execute in parallel. E.g. with a shared instruction and data memory, instructions cannot be fetched at the same time as data is being written.

Data Hazards
Instructions that use some value (i.e. registers) depend on the preceding instructions that modify the same value. Executing an instruction without first receiving the new value for the dependency will yield incorrect results.

Control Hazards
Also called **branch hazards**. When a branch instruction is executed, the next instruction to be executed may be one of two possibilities. The pipeline must be able to react to a branch and flush instructions that are not to be executed from the pipeline before the results of those instructions are committed.

1.3 Exercise Requirements

The exercise description outlines the requirements[1, Section 5.3]:

- Support the same instruction set as exercise 1 required[1, Section 4.3]. The implemented instructions are listed in appendix B.
- Implement a pipelined design
- Detection and correction of hazards
- Verification with testbenches
- Verification on an FPGA

All of the requirements are fulfilled in this implementation.

2 | Solution

2.1 Development

As with exercise 1, the primary tools used for development were:

- *XILINX ISE* for writing and syntax checking VHDL
- *XILINX ISim* for testing simulated modules in testbenches
- *hostcomm*[4] for testing the synthesized processor on an FPGA
- *git* for collaboration and version control
- *draw.io* for sketching components for the report

2.2 Pipelined Design

A pipelined processor requires synchronous separation between the different stages. When a clock cycle begins, each stage must get some input, process it, then output the result by the end of the cycle. The next stage gets its next cycle input from the previous stage output.

This is implemented by adding registers between the executing units of the stages. Each stage reads its input register at the beginning of the cycle and writes its result to an output register which is the input register for the next stage. Thus the clock frequency of the entire processor is dependent upon the time it takes for the slowest stage to read input, process it, then write output.

2.2.1 Pipeline Stages

Figure 2.1 illustrates which components are used by each pipeline stage. The forwarding unit and the hazard detection unit is left out to simplify the figure. These modules are described in section 2.3.2.
During development, the group used two design sketches for reference. These are attached to the report in appendix A.

Resources Shared by Stages

One thing to note in figure 2.1 is that the PC module and the registers module take inputs from multiple stages. The PC module will select the input from MEM if a control signal is asserted, or else it defaults to the input from the IF stage. For the registers module, the input from the ID stage is asking to read registers, while the input from the

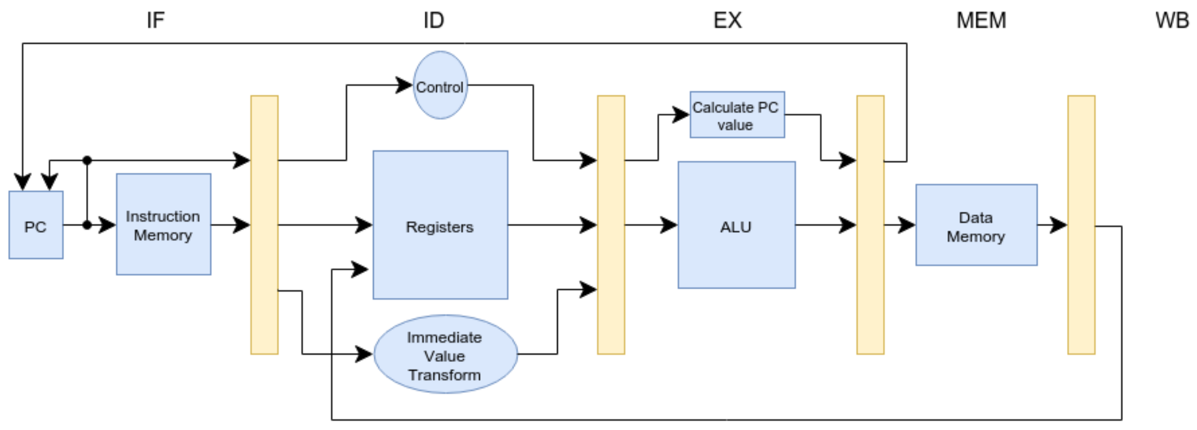


Figure 2.1: Simplified overview of the architecture.

WB stage is writing to a register. The registers module is designed to allow these actions concurrently. This way structural hazards are avoided.

2.2.2 Information-passing Between Stages

The registers simplified as yellow blocks in figure 2.1 are implemented as separate entities of the same VHDL sources.

The outputs from the instruction and data memories are the slowest part of the pipeline. To avoid having to wait too long between cycles, these outputs bypass registers and will arrive directly from the output of the memory to the next stage.

VHDL Register Modules

The values that pass through the registers are of different sizes, from 32-bit instructions and operands to single bit signals. The registers for these transfers are implemented in VHDL as generic components that can be re-used.

With the exception of the type of data they hold, all the registers are the same. They have an asynchronous reset that can be used both for a system reset and a pipeline flush, they take in an input on each rising clock edge. They also have a write enable input that defaults to enable, but can be deasserted to hold the current value instead of replacing it with input.

`bit_register`
Holds a single bit value.

`generic_register`
Has a generic `WIDTH` property that is specified when an entity is created. Holds a bit vector of the specified width.

`EX_register`, `MEM_register`, `WB_register`
These special registers hold the record control signal bundles that the control module outputs. These registers can be seen in figure A.1 labeled EX, M and WB.

2.3 Pipeline Hazards

The following sections descriptibes how the processor detects and corrects various hazards.

2.3.1 Structural Hazards

Structural hazards are avoided entirely in the implemented design. The separation of instruction memory from data memory means that the IF stage may read the instruction memory concurrently with the WB stage reading or writing the data memory.

2.3.2 Data Hazards

Forwarding

When a result is calculated in the execute stage, it will be several cycles until it is ready to be read from a register by a dependent instruction. One solution to this problem is to stall the pipeline until the result is ready in the register, but this is not the most efficient solution.

A better approach is to implement a forwarding scheme, making results in the MEM or WB stages available in earlier stages.

A forwarding unit forwards data from MEM or WB stage, to the EX and the ID stage when needed. Figure 2.2 and listing 2.1 illustrate the forwarding unit. This generic component is instantiated twice in the ID stage and twice in the EX stage. A register value and its associated register address comes from a previous stage. If the selecter unit detects that the selected register has an updated value in the MEM or WB stage, it will reroute this value to replace the value from the earlier stage.

Figure 2.3 shows the placement of the forwarders in the pipeline.

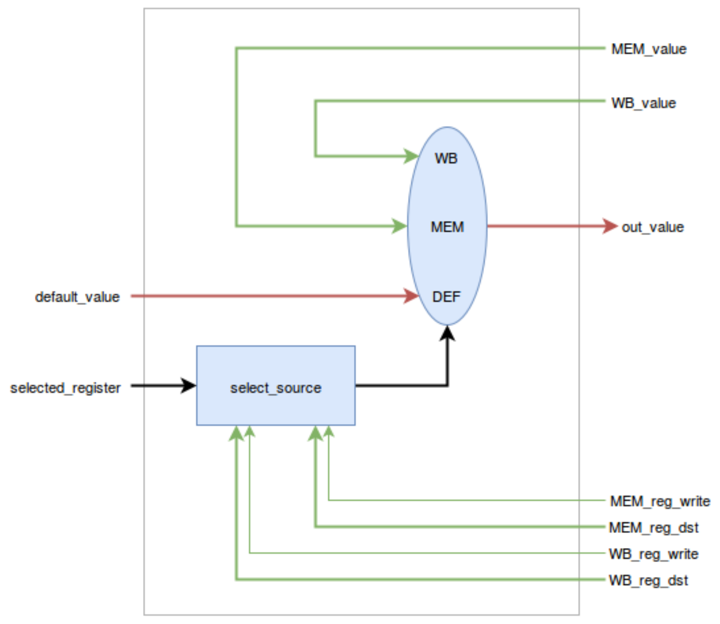


Figure 2.2: Forwarder multiplexer unit. Inputs on the right come from a later stage.

Listing 2.1: Source select logic

```
if MEM_reg_write = '1' and selected_register = MEM_reg_dst then
    selected_source <= MEM;
elsif WB_reg_write = '1' and selected_register = WB_reg_dst then
    selected_source <= WB;
else
    selected_source <= DEFAULT;
end if;
```

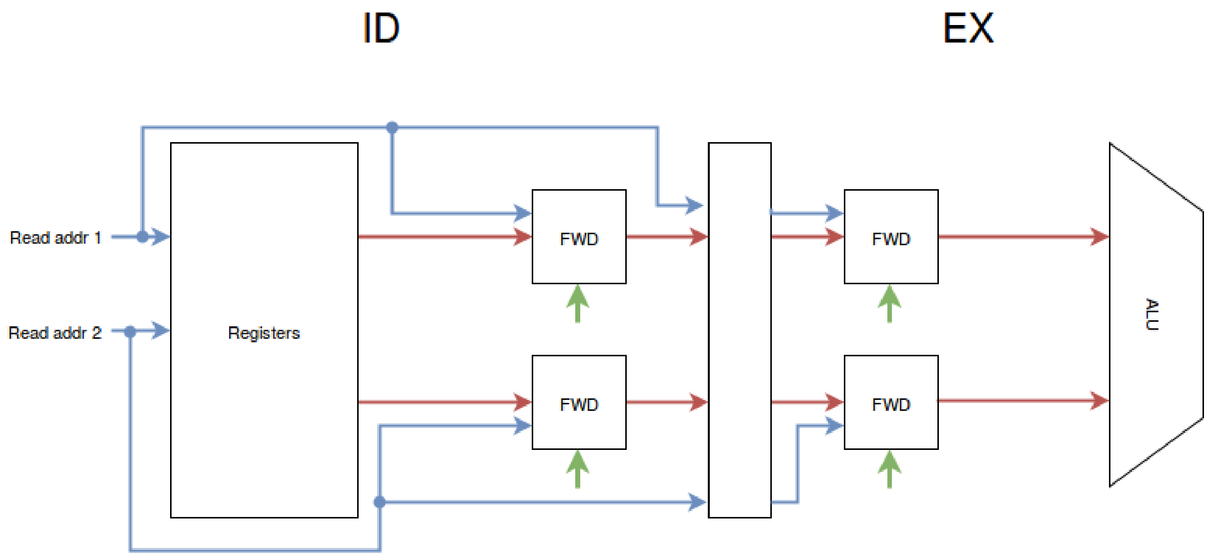


Figure 2.3: Simplified overview of the parts of the pipeline involving forwarding. The green arrows represent inputs from later stages, and are the same for all forwarders.

Stalling

If there is a *load word* instruction with an immediately following dependent instruction, forwarding can not solve the hazard entirely, and a stall is required.

To enable this a hazard detection unit is implemented in the processor. The unit detects data dependency between a `lw` instruction in the EX stage and a dependent instruction is in the ID stage. The expression for this is shown in listing 2.2. If this is the case is will hold the program counter value and the instruction in the IF to ID registers for one more cycle. The instruction passing from ID to EX will continue execution, but is replaced in the register afterwards with a no-op to avoid executing it twice. Once the `lw` instruction has passed the EX stage the stall condition is no longer valid, and the pipeline continues as normal.

Figure 2.4 shows the hazard detection unit inputs and what its output signal controls.

Listing 2.2: Expression used to set stall control signal

```
stall <= '1' when
  mem_read = '1' and (ex_rt = id_rs or ex_rt = id_rt) else '0';
```

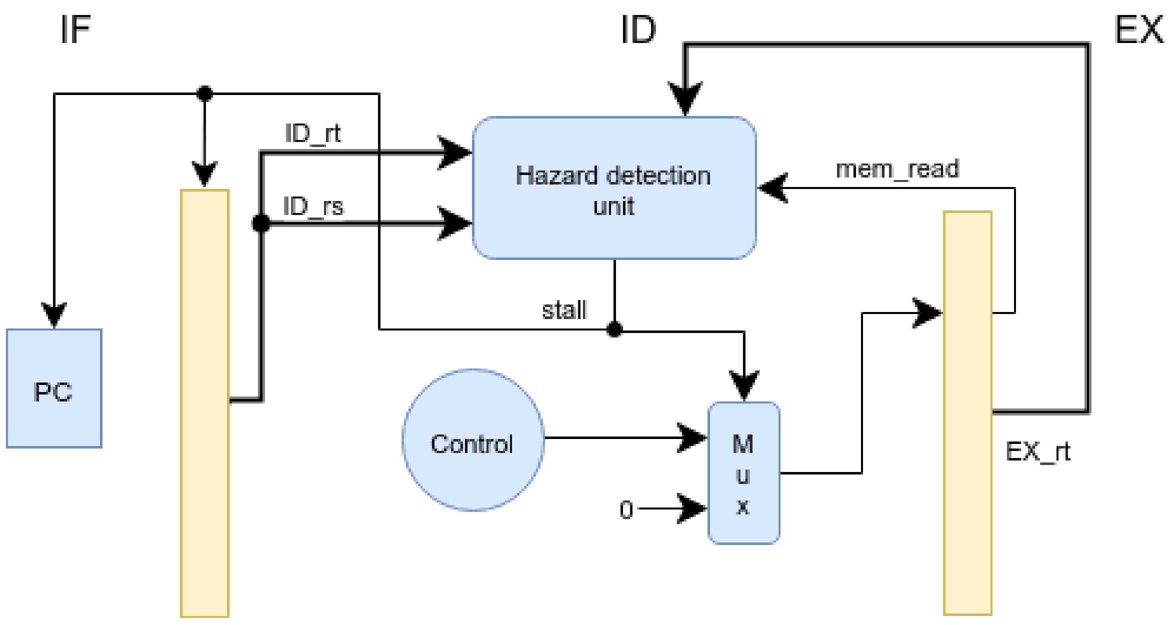


Figure 2.4: Overview of the hazard detection unit.

2.3.3 Control Hazards

The implementation doesn't do any branch prediction, it just assumes the branch is not taken. When a branch is taken and changes the execution order the pipeline before the EX stage has to be flushed, replacing instructions with `no_op` instructions.

In the implementation there is a `flush` signal which is indicates when the pipline has to be flushed. Listing 2.3 shows the condition for asserting the `flush` signal.

Listing 2.3: Control hazard detection

```
if jump = '1' or (branch = '1' and alu_zero = '1') then
    flush_pipeline <= '1';
```

The `flush` signal is connected to the reset on the registers that are to be flushed, and the result of resetting these registers is a `no_op`.

3 Results

3.1 Testing

3.1.1 Simulated Testbenches

For the modules that were reused from exercise 1, the associated testbenches were also reused. Changes to the modules were followed by changes to the associated testbench.

In addition to the provided processor testbench, there are three separate testbenches created that each test the pipeline's ability to handle a specific hazard. By making each of these tests pass, the provided testbench also started passing.

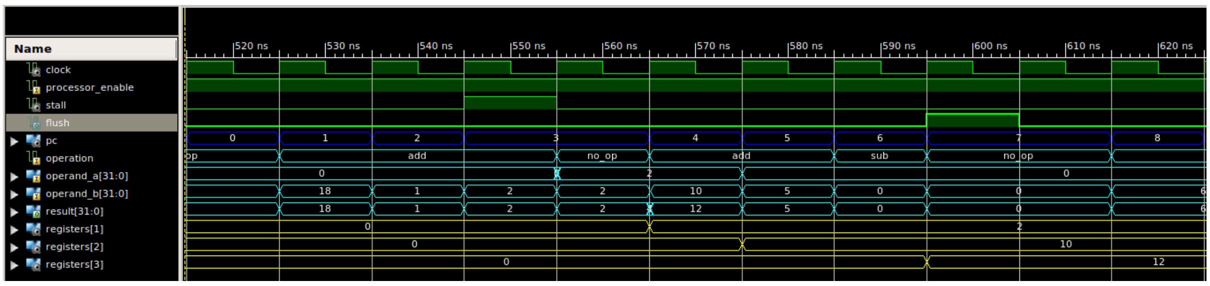


Figure 3.1: Showing some assorted signals while *ISim* simulates the provided testbench. The instruction `lw $2, 2($0)` is fetched at PC=2 and is followed by `add $3, $1, $2`, requiring a stall.
The add instruction can be seen executed at PC=4, and the resulting value 12 is written to register 3 by PC=7.

Forwarding Test

Table 3.1 shows the pipeline executing the test program for the forwarding testbench. RAW hazard are highlighted, both where the EX stage needs forwarding to the ALU, and where the ID stage needs register values before they are written.

Stalling Test

Table 3.2 shows the pipeline from the stalling testbench. A value is being read to register 1, and is needed for the next instruction (addition). Since the value won't be available until after the load instruction has passed the MEM stage, the add instruction needs to be stalled. Because forwarding has also been implemented, only one cycle of stalling is required.

	IF	ID	EX	MEM	WB
0	lw $1, 1($0)				
1	lw $2, 2($0)	lw $1, 1($0)			
2	add $3, $1, $1	lw $2, 2($0)	lw $1, 1($0)		
3	add $4, $3, $3	add $3, $1, $1	lw $2, 2($0)	lw $1, 1($0)	
4	sub $5, $4, $3	add $4, $3, $3	add $3, $1, $1	lw $2, 2($0)	lw $1, 1($0)
5	sw $3, 3($0)	sub $5, $4, $3	add $4, $3, $3	add $3, $1, $1	lw $2, 2($0)
6	sw $4, 4($0)	sw $3, 3($0)	sub $5, $4, $3	add $4, $3, $3	add $3, $1, $1
7	sw $5, 5($0)	sw $4, 4($0)	sw $3, 3($0)	sub $5, $4, $3	add $4, $3, $3
8		sw $5, 5($0)	sw $4, 4($0)	sw $3, 3($0)	sub $5, $4, $3
9			sw $5, 5($0)	sw $4, 4($0)	sw $3, 3($0)
10				sw $5, 5($0)	sw $4, 4($0)
11					sw $5, 5($0)

Table 3.1: Pipeline containing RAW hazards requiring forwarding to correct.

PC	IF	ID	EX	MEM	WB
0	lw $1, 2($0)				
1	add $1, $1, $1	lw $1, 2($0)			
2	sw $1, 3($0)	add $1, $1, $1	lw $1, 2($0)		
2	sw $1, 3($0)	add $1, $1, $1		lw $1, 2($0)	
3		sw $1, 3($0)	add $1, $1, $1		lw $1, 2($0)
4			sw $1, 3($0)	add $1, $1, $1	
5				sw $1, 3($0)	add $1, $1, $1
6					sw $1, 3($0)

Table 3.2: Pipeline containing a RAW hazard that requires stalling to correct.

Branching Test

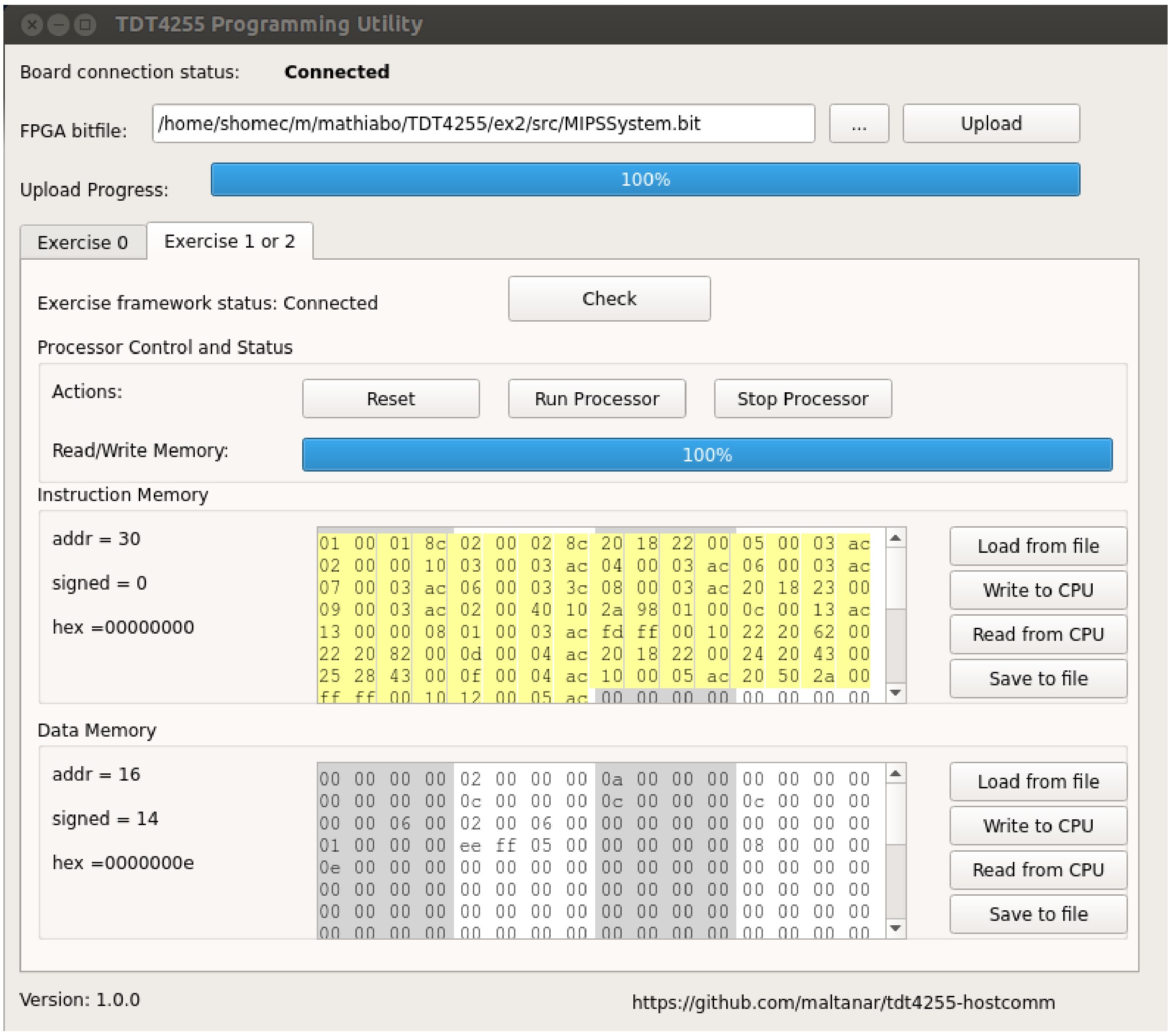
Table 3.3 shows the pipeline from the branching testbench. In the test program a branch is taken which means the add instruction is not to be committed. When the branch instruction reaches the MEM stage and updates the program counter, the instructions from the wrong branch are flushed.

PC	IF	ID	EX	MEM	WB
0	lw $1, 1($0)				
1	beq $0, $0, 1	lw $1, 1($0)			
2	add $2, $1, $1	beq $0, $0, 1	lw $1, 1($0)		
3	no op (1)	add $2, $1, $1	beq $0, $0, 1	lw $1, 1($0)	
4	no op (2)	no op (1)	add $2, $1, $1	beq $0, $0, 1	lw $1, 1($0)
3	no op (1)	(flushed)	(flushed)	(flushed)	beq $0, $0, 1

Table 3.3: Pipeline containing a branch that is taken

3.1.2 Testing on an FPGA

The test program from exercise 1 was reused when implementing the new processor design on an FPGA. After ensuring all the testbenches described above passed, the FPGA version also produced correct results.



TDT4255 Programming Utility

Board connection status: **Connected**

FPGA bitfile: /home/shomec/m/mathiabo/TDT4255/ex2/src/MIPSSystem.bit

Upload Progress: 100%

Exercise framework status: Connected

Processor Control and Status

Read/Write Memory: 100%

Instruction Memory

addr = 30

signed = 0

hex =00000000

```
01 00 01 8c 02 00 02 8c 20 18 22 00 05 00 03 ac
02 00 00 10 03 00 03 ac 04 00 03 ac 06 00 03 ac
07 00 03 ac 06 00 03 3c 08 00 03 ac 20 18 23 00
09 00 03 ac 02 00 40 10 2a 98 01 00 0c 00 13 ac
13 00 00 08 01 00 03 ac fd ff 00 10 22 20 62 00
22 20 82 00 0d 00 04 ac 20 18 22 00 24 20 43 00
25 28 43 00 0f 00 04 ac 10 00 05 ac 20 50 2a 00
ff ff 00 10 12 00 05 ac 00 00 00 00 00 00 00 00
```

Data Memory

addr = 16

signed = 14

hex =0000000e

```
00 00 00 00 02 00 00 00 0a 00 00 00 00 00 00 00
00 00 00 00 0c 00 00 00 0c 00 00 00 0c 00 00 00
00 00 06 00 02 00 06 00 00 00 00 00 00 00 00 00
01 00 00 00 ee ff 05 00 00 00 00 00 08 00 00 00
0e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Version: 1.0.0

https://github.com/maltanar/tdt4255-hostcomm

Figure 3.2: Resulting date memory loaded from the FPGA after running the pipelined processor implementation with the test program.

3.2 Measured Performance

The final implementation may acheive a clock frequency of 48.6 MHz according to the synthesis report from Xilinx ISE. The clock frequency is not measured on an actual FPGA. In exercise 1 the multi-cycle processor achieved a clock frequency of 46.4 MHz.

4 | Discussion

4.1 Test Coverage

During development testbenches for each hazard correction method were created to make sure they were properly handled. Each testbench tests a specific method to handle a specific hazard.

The provided testbench contains multiple hazards and tests that the different hazard correction methods work together.

For better test coverage we could have created more programs for system testing, to test the interaction between more instructions, hazards and hazard correction methods.

4.2 Progress

The group gained valuable experience about VHDL and the *XILINX* toolchain from exercise 1. In this exercise the coding and testing went along smoothly. We started using a few new tools, such as records and generics.

The main obstacle for the exercise was learning the theory of a pipelined processor. The textbook and lecture slides were used to gain a better understanding of this, to a much larger degree than in exercise 1.

4.3 Performance

With a pipelined design, the processor achieves a slightly higher max frequency compared to the multi-cycle design. Even though the max frequency has not increased considerably, the number of instructions per cycle has increased greatly with the new design. These improvements together means a noticeable performance improvement over the multi-cycle processor design.

The group does not have enough experience with the synthesis report from *Xilinx ISE* to determine were the design could have been optimized. Given more time, increasing clock frequency would have been prioritized.

The processor is able to execute one instruction per clock cycle most of the time. The exceptions to the one instruction per clock cycle rule are:

- *Load word* instructions with an immediately following data dependent instruction, which requires the system to stall for one cycle.
- *Jump* or taken *branch* instructions, where the processor has to flush three stages.

4.4 Further Work

There is a lot to be gained from improving loop performance. By implementing branch prediction the number of pipeline flushes could be reduced.

In order to improve on the load word stalls, a dynamic scheduling scheme could be implemented, rearranging instructions to cover up the required stalling cycle.

It would also be possible to have more than one instruction executed each cycle by implementing multiple functional units. This requires extra hardware and increases complexity.

Perhaps most crucially the current implementation lacks exception mechanisms. Implementing this would be very much required if the processor were to be used in any kind of real-life scenario.

5 Conclusion

The requirements for the implementation were fulfilled. The new implementation has a pipelined design, improving the processor throughput, but avoids the pitfalls that pipeline hazards create. The test program from exercise 1 outputs correct results both in the testbench simulation and on an FPGA.

The unit tests from the previous exercise still pass, some of them with some modifications, and new tests have been created and passed for the specific pipeline hazards.

The instructions from exercise 1 that were not part of the requirements are still supported in the new implementation.

The exercise had a good learning outcome. The group members now have a much better understanding of the workings of a pipelined processor. The group also began using more VHDL features such as records, functions and generics that were not utilized in exercise 1.

A | Architecture Sketches

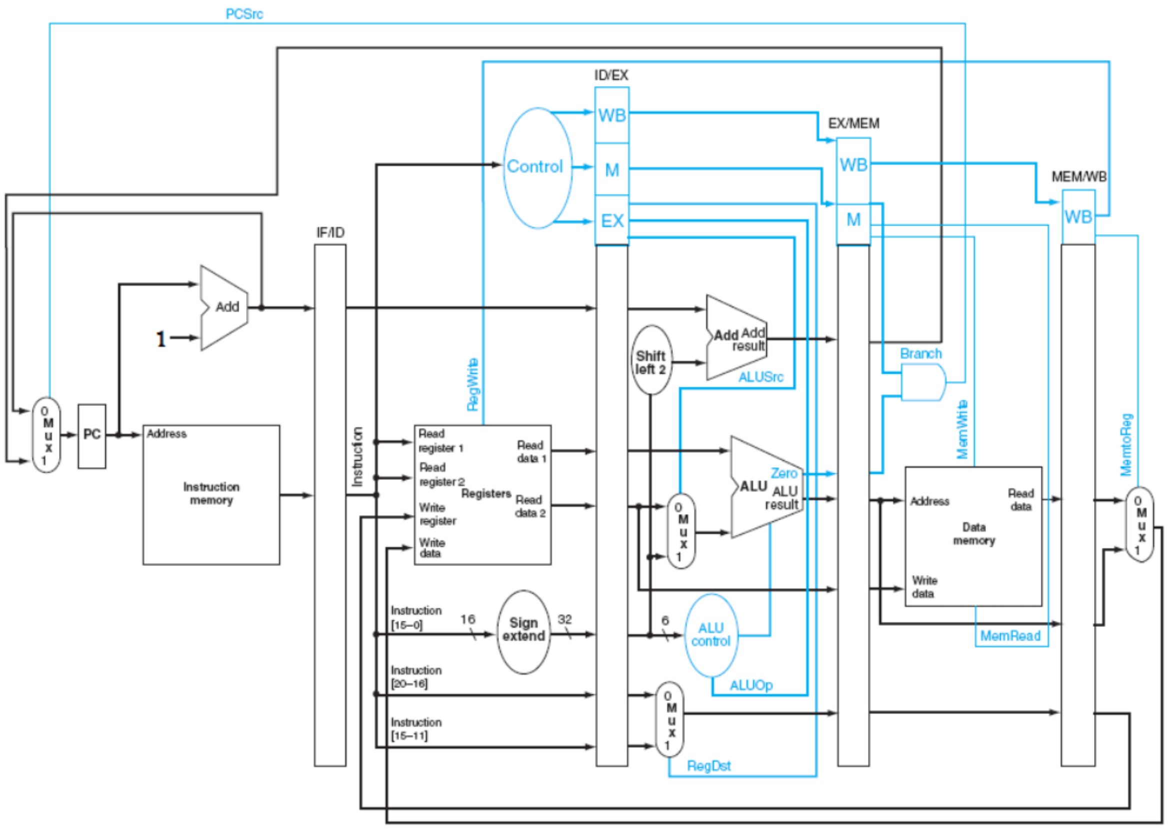


Figure A.1: Suggested architecture from the exercise description[1, Section 5.2]

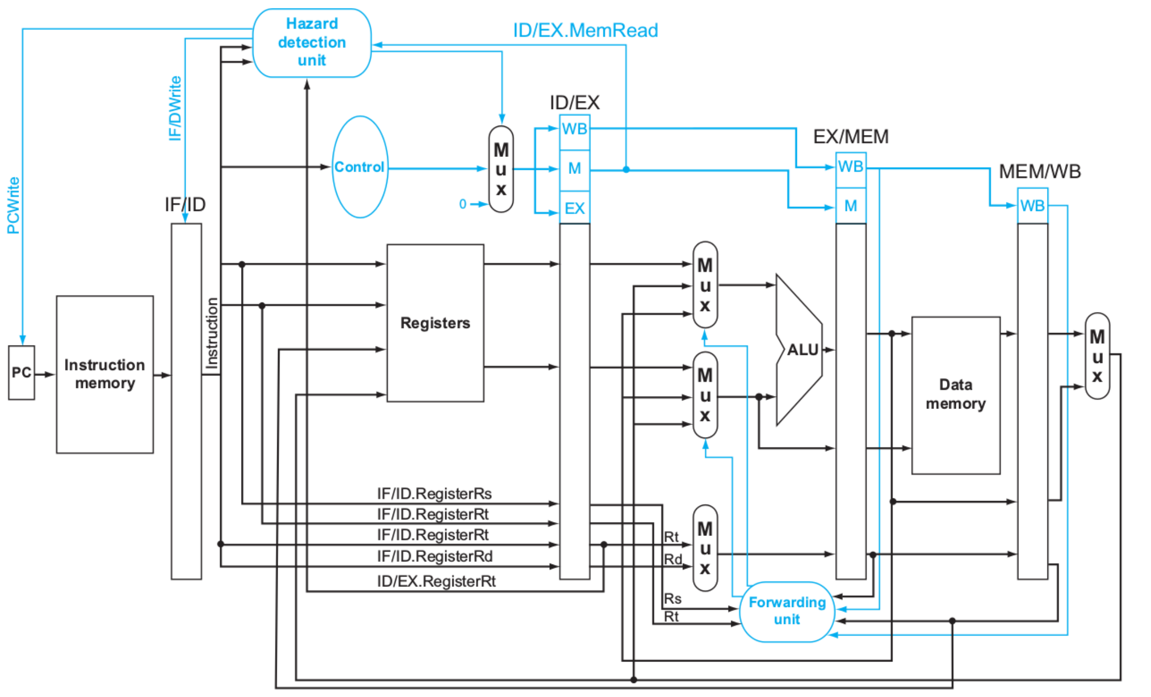


Figure A.2: Architecture from *Patterson & Hennessy*[2, Chapter 4.7].

B | Instruction Set Architecture

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 B.1: Implented instruction set

References

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

[2] John L. Hennessy David A Patterson. Computer Organization and Design, Fifth Edition. Morgan Kaufmann, 2014.

[3] Jonatan Lund Mathias Ose, Jonas Halland. Exercise 1: Multi-cycle mips processor. `https://github.com/mathiasose/TDT4255/raw/master/ex1/report/report.pdf`, 2015.

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