A Pipelined MIPS-Processor Excercise 2 in TDT4255

WASAZNIK, Aleksander Gustaw, SELVIK, Andreas Løve, and KULIA Geir

***Abstract*— The multi-cycle CPU described in our previous paper was extended to a single-cycle pipelined CPU. Advanced hazard-mitigation-techniques were introduced; including data-forwarding, hazard detection and static branch-prediction.**

This resulted in large gains in the speed of the CPU. The CPU is now able to process one instruction every cycle, at a maximum clock-speed of almost 90 MHz.

The CPU performs well in testing on a FPGA with a frequency of 25 MHz, and maintains full backwards compatibility for binaries made for our previous processor. The pipeline is transparent to the programmer as all hazards are detected and mitigated.

I. INTRODUCTION

The processor described is an evolution of our previous design of a multi-cycle MIPS processor as described in [1]. The processor was converted to use a pipelined design. Thereupon several innovations for ease-of-programming (in the form of hazard detection) and speed-optimization were introduced: Data-forwarding, pipeline-interlocking, static branch-prediction, and fast jumping. Each of these techniques and their implementation will be explained throughout this document.

It is recommended that the reader be familiar with the workings of a multi-cycle processor-architecture as described in [2, p. 242], as the primary focus of this paper will be the changes introduced to our previous design, which is very close to this design.

II. INNOVATIONS

Provided here is an overview for the spotlight features implemented in our design. The details of implementation for each of these is provided in Section III: Implementation.

A. Pipelining

Following the design outlined in [2, p. 272], a 5-stage pipeline was implemented. This allowed our processor to execute up to five instructions simultaneously, offering the processor between two and three times the throughput of our previous design, at the cost of some extra latency.

The processor was divided into the stages: *instruction fetch*, *decode*, *execute*, *memory* and *write-back*, the contents of which are described in detail in Section III: Implementation. The stages were separated by pipes consisting of flip-flops. Signals from external memory and the register file were not flip-flopped as they were synchronous, and provided the necessary delay on their own. The naive implementation of the pipeline was prone to several data hazards. The remedies implemented for them are described in the next few sections.

B. Data-forwarding

An instruction making use of data produced by any of the previous 3 instructions would have referenced data not yet committed to the register file.

To prevent use of stale data, a data-forwarding-system was implemented. The data-forwarding-system sidestepped the pipeline to access the most recent data for a specific register.

This innovation allowed the processor to run at optimal speed even when faced with register-dependencies between instructions.

C. Hazard detection with stalls

With the 5-stage pipeline design from [2, p. 272], a load followed by use of that value in the next instruction is not possible without a stall even with a data-forwarding unit, as the result of load-operations became ready first in the writeback stage as opposed to other instructions where it is ready in the memory stage.

A system was devised to insert a safe **noop**-instruction before an instruction dependent on a load not yet performed. This ensured the data loaded was not used before it was available.

D. Branch-prediction

Static branch-prediction in the form of assume-not-taken was implemented to allow the pipeline to continue uninterrupted if the branch was predicted correctly. If the branch was mispredicted, the system would invalidate two of the instructions in the pipeline, and continue on the correct path.

III. IMPLEMENTATION

A. Pipeline

The overview of the pipeline design is shown in Figure 1. The implementation in VHDL was proven to be very cumbersome simply because of the drastic proliferation of instances of signals crossing multiple pipes. Attempts at managing these to produce maintainable code include grouping signals traveling though one pipe as records in VHDL, with the intention of flip-flopping the record as a whole. This was however made difficult by the signals from the synchronous entities, which were not to be flip-flopped.

To make the RTL-drawings produced by the Xilinx toolchain readable, both the stages and the pipes were made entities. Xilinx did not draw records as a single signal.

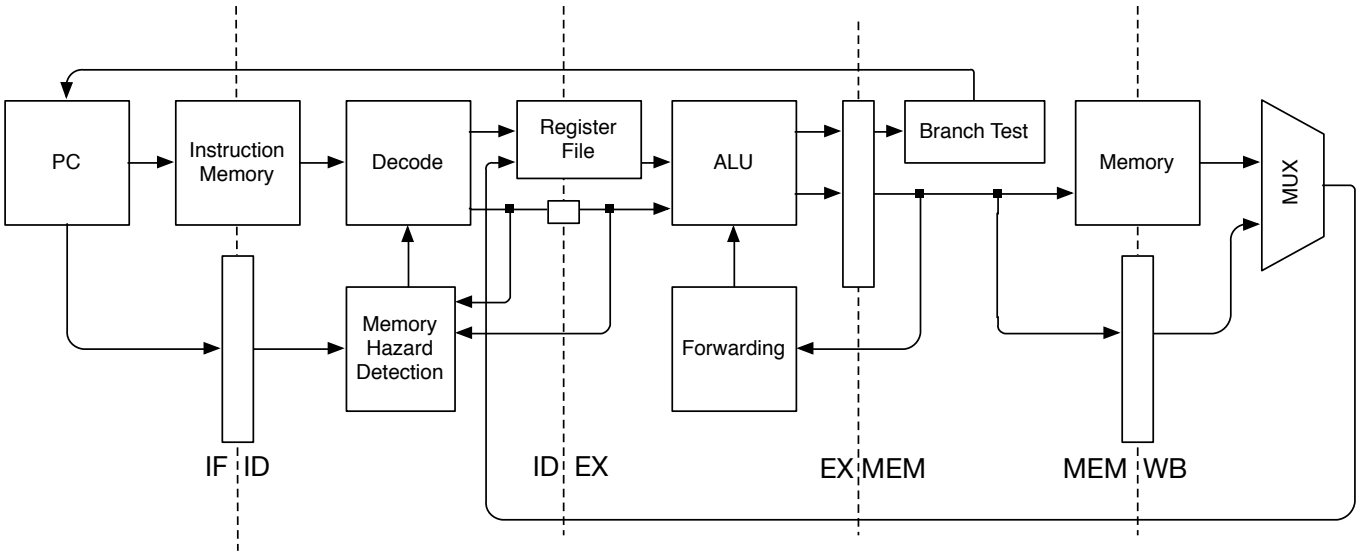

Fig. 1. Overview of the pipeline design and flow of data. The dotted lines denotes logical boundaries of the different stages. Entities that lie on the boundaries denote synchronous components, and as such act as part of a pipe. The stages represent a part of the pipeline in which a particular instruction is active.

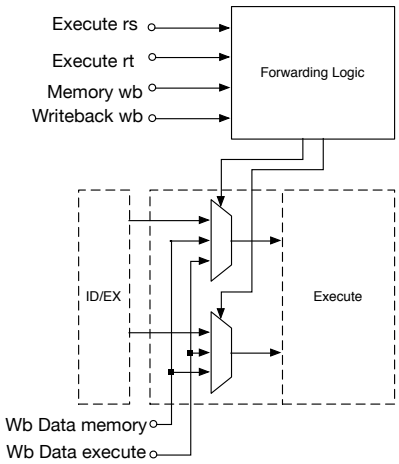

Fig. 2. Overview of the data forwarding unit.

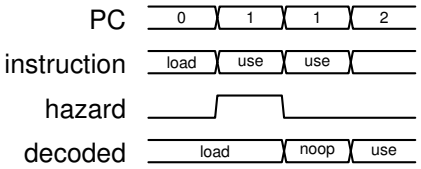

Fig. 3. A demonstraction of a hazard-detection.

B. Data-forwarding-unit

The data-forwarding unit consists of two parts: the logic and the forwarding muxers. The control of the two muxers is independent and duplicate of each other. The logic dictates that the muxers should draw from the freshest source of the specified registers, if they match the writeback registers of the forwarded data. The layout of the muxers is shown in Fig. 2.

C. Hazard-detection-unit

The hazard-detection-unit is responsible for detecting usage of registers with not yet satisfied memory request. If the instruction currently in execute-stage is a memory load, it should insert a noop if the next instruction is dependent on the result from the load. A RTL-drawing of the logic and the interaction with the surrounding pipeline is shown in Fig. 4.

When the hazard is detected, *insert_stall* is set high. This signal will tell the decode stage to force the write enable flags to 0, effectively converting the preceding instruction to a **noop**. In addition we have to make sure that the killed instruction is executed in the following cycle. This is achieved in the *if_to_id_pipe* by keeping a flip-flop with the last instruction and using this during a stall. Further, to avoid skipping an instruction the program counter in the instruction decode is frozen for a cycle.

D. Branch-prediction

As our branch prediction is ”assume not taken”, we could use the ”take branch” signal to flush the pipeline. This was implemented by simply OR-ing the ”takebranch” signal into the reset signal of the pipes between IF and ID, ID and EX, and EX and MEM. This flushes the correct part of the pipeline every time the branch is predicted wrong, ie. taken.

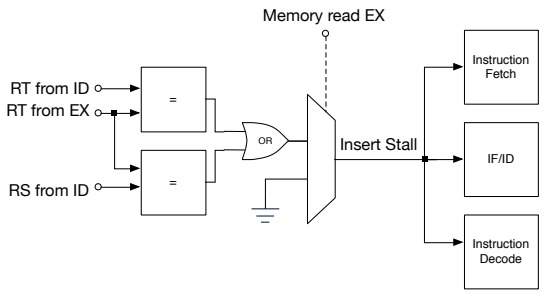

Fig. 4. Overview of the the hazard detection unit. The circuit determin wheter an hazard has occurred and stall Instruction Fetch, Instruction Deocde and the IF/ID pipe accordingly.

IV. Verification

A. Pipeline

Our pipeline was tested by running a modified version of the test we got handed out. The tests runs a MIPS program that loads values, calculates and stores. Then it checks whether memory contains the expected values. We modified it to insert 4 **noop** instruction between every instruction, as we did not handle hazards at this time. The test passes, and we get the same result as we did in the previous multicycle design.

B. Data-forwarding-unit

The forwarding unit relied on relatively simple logic. It has two independent instances of a function deciding whether to forward data, and from where. Of more interest was the question of whether it was properly integrated with the rest of the processor. To test this, a MIPS-program was written in assembly, designed to fully exercise the forwarding data-paths. The program aided in identifying a mismatched signal, that was fixed. The program can be found in Fig. 6.

C. Forwarding unit

To test our forwarding unit, we wrote a MIPS program with hazards that requires forwarding both from the memory stage and the write-back stage. And for each forwarding stage we have one case for each of the different forwarding paths. The test also exercises the case of accumulating values in a single register, to test the freshness-prioritizing. The MIPS assembly can be found in figure 6.

D. Hazard-detection

The hazard detection unit is tested in the same way as the forwarding unit, by a MIPS program that is crafted such that it will produce a different result if the hazard detection unit doesn't stall the pipeline properly. As can be seen from figure 7, it does two test; load add store and load store. The test passes.

E. Static branch prediction

The MIPS program found in figure 8 is executed and memory results are checked to see whether the correct instructions were executed.

The test passes.

F. Complete processor

Mulptiple test-programs was devised to exercise all the possible hazards, as described above. The programs are shown in Appendix A. In addition, we ran the original full system test that was run on the previous design. It contains multiple hazards. All test passes.

G. Implementation on FPGA

Using the provided Hostcomm framework, the same program as used in the simulation of the complete processor (in Subsection IV-F were performed on processor running on a Xilinx FPGA.

The processor was able to run on the FPGA and produced the correct memory values, as shown in appendix B-A.

V. Discussion

A. Branch-prediction

In our design a static assume-not-taken branch-predictor was implemented. A good quality of this design is that a correct prediction incurs no overhead. It is however particularly bad for loops (that are not unrolled), as it will mispredict on every but the last iteration.

Other branch-predictors were considered. But, as it was not part of the requirements of the exercises, they were not pursued in favour of other parts of the exercise.

1) Assume taken: Assume taken in a prediction rule that works very well for programs that loop. However even when successful, assuming taken gives the some penalty as a jump: two cycles.

2) Assume backwards taken: Assume-backwards branching-strategy is a good mixture of the two. Backwards branches are indicative of loops, so they are taken. Forward branches on the other hand are indicative of conditional execution, that may be uncertain or where the programmer can decide what is most likely. Choosing then the option with least overhead, the branch is assumed not taken.

3) Dynamic prediction: A more advanced form of branch-prediction keeps track of whether a branch in a particular location in the program tends to be taken.

4) Tournament prediction: Several branch-predictors can be implemented in a processor design. The processor then keeps track of which is the most efficient (most correct predictions) in the current stretch of program.

B. The necessity of the memory-stage

As our data-memory is synchronous, is should have been thought of as part of the pipe from one stage to the next. This left the *memory*-stage in out design almost empty, with only the branching-test remaining.

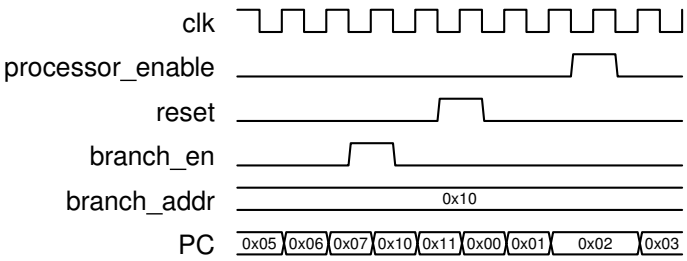

Fig. 5. Sample waveform of the test bench for the program counters. The processor_enable is added to delay the updating of the program counter while the processor is disabled.

C. PC start value

As the program counter is incremented on every clock up where processor enable is set, the easiest solution to make sure it starts at 0 is to start the PC at ”1111111”

D. Load-store-forwarding

A special case of pipe-interlocking could be avoided: The case of a store using a result loaded in the previous instruction. In the case of a store-operation the data needs not to pass through the ALU, but can instead be forwarded directly from the writeback stage to the memory stage.

This functionality doubled the speed at which memory-memory-copying could be performed.

It is doubtful that the extra complexity the memory-stage introduced was worth any gains it resulted in. If the branch-predictor was moved to either *execute* or *writeback*, the *memory*-stage could have been removed from the design, even through it virtually was still there.

VI. Test Bench

A. Test Bench

The test benches were mainly reused from the last exercise, and just extended with a few signals

B. Program Counter

Many test benches were reused from the last exercise[1], and extended with a few signals. Fig. 5 shows an example waveform where the previous test bench has been extended.

C. Data Hazard

The test bench to control whether the processor handled data hazards correctly consisted of four tests. It first tried to read after a write, then write after a read, write after a write and also to add a register with itself ten times.

VII. Conclusion

The processor was successfully implemented with a maximum clock frequency of 87 MHz reported by the timing analysis tool. Compared to our previous design, which used either 2 or 3 cycles per instruction at 87 MHz, the maximum throughput was improved by up to 200% as it now completes up to one instruction per cycle, in comparison with the old design. The pipelined processor has support for data forwarding, pipeline interlocking and static branch prediction. The processor ran successfully on the FPGA, and passed all test programs that were carefully constructed to detect errors.

Appendix A
Assambly testprogram

Figures 6, 7, and 8 show the testprograms used to verify the processor, both in simulation and on the FPGA.

Appendix B
Proofs of verification by screenshots

A. Testing on a FPGA

This screenshot shown in fig. 9 serves as proof of a successful test of the processor running on a FPGA. The code run is the canonical testbench, as handed out.

References

[1] A. L. Selvik, A. Wasaznik and G. Kulia, *Report, Lab Exercise 1*, 1st print, TDT4255 Computer Architecture, NTNU, Norway, 2014

[2] D. A. Patterson, J. L. Hennessy, *Computer Organization and Design*, fith edition, Morgan Kaufmann, 2014

Fig. 6. Forwarding test-program: This program was used to excercise all aspectes of the fowarding unit.

```
lw $1 0($0)     # Load 1 in to $1 from memory addr 0.
nop             # Empty pipeline.
nop
nop
nop

# Test simple forwarding rt from memory:
add $2 $1 $0
sw $2 4($0)     # Expect: 1 on addr 4

                # Test simple forwarding rt from writeback:
add $2 $1 $0
nop             # Let the data propagate to writeback.
sw $2 5($0)     # Expect: 1 on addr 5


                # Test simple forwarding rs from memory:
add $2 $0 $1
add $2 $2 $1
nop             # We are not testing rt - let it go
nop             # back to the register.
nop
nop
sw $2 6($0)     # Expect: 2 on addr 6

                # Test simple forwarding rs from writeback:
add $2 $0 $1
nop             # Let the data propagate to writeback.
add $2 $2 $1
nop             # We are not testing rt-forwarding -
nop             #let the data go back to the register.
nop
nop
sw $2 7($0)     # Expect: 2 on addr 7


                # Test accumulation rt:
add $2 $1 $0
add $2 $1 $2
add $2 $1 $2
add $2 $1 $2
add $2 $1 $2
sw $2 8($0)     # Expect: 5 on addr 8


                # Test accumulation rs:
add $2 $0 $1
add $2 $2 $1
add $2 $2 $1
add $2 $2 $1
add $2 $2 $1
nop             # We are not testing rt-forwarding -
nop             # let the data go back to the register.
nop
nop
sw $2 9($0)     # Expect: 5 on addr 9

                # Test load nop store:
lw $2 0($0)     # Load 1 to $2
nop
sw $2 10($0)
```

Fig. 7. Hazard-detection test-program: This program was used to excercise the hazard-detection-unit.

```
#Setup by loading into $4
lw $4 2($0) #$4 = 5
noop        # Empty pipeline, we are not testing this.
noop
noop
noop

#Start test 1
add $1 $4 $0   #$1 = 5
noop           #Not testing this part, nop
noop
lw $1 1($0)    #Load 10 into $1
add $2 $1 $0   #Add 0 to $1. This is the hazard we are testing.
noop               #Not tested here
noop
noop
sw $2 10($0)  #Store the result. 10 if it works, otherwise 5

# Test 2
add $1 $4 $0  # $1 = 5
noop
noop
lw $1 1($0)   # load 10 into $1
noop          # This is the task of the forwarding unit
sw $1 11($0)  # Store $1. This is the hazard.
```

Fig. 8. Branching test-program: This program was used to excercise branching.

```
lw $1 0($0)     # Load 1 in to $1 from memory addr 0.

nop             # Empty pipeline.
nop
nop
nop

# Test branching with distance 1:
beq $0 $0 1
sw $1 4($0)     # Skip. Expect: 0 on addr 4
sw $1 5($0)     # Expect: 1 on addr 5
sw $1 6($0)     # Expect: 1 on addr 6
sw $1 7($0)     # Expect: 1 on addr 7

nop             # Empty pipeline.
nop
nop
nop

# Test branching with distance 2:
beq $0 $0 2
sw $1 8($0)     # Skip. Expect: 0 on addr 8
sw $1 9($0)     # Skip. Expect: 0 on addr 9
sw $1 10($0)    # Expect: 1 on addr 10
sw $1 11($0)    # Expect: 1 on addr 11

nop             # Empty pipeline.
nop
nop
nop

# Test branching with distance 2:
beq $0 $0 2
sw $1 8($0)     # Skip. Expect: 0 on addr 8
sw $1 9($0)     # Skip. Expect: 0 on addr 9
sw $1 10($0)    # Expect: 1 on addr 10
sw $1 11($0)    # Expect: 1 on addr 11

nop             # Empty pipeline.
nop
nop
nop

# Test branching with distance 3:
beq $0 $0 3
sw $1 12($0)    # Skip. Expect: 0 on addr 12
sw $1 13($0)    # Skip. Expect: 0 on addr 13
sw $1 14($0)    # Skip. Expect: 0 on addr 14
sw $1 15($0)    # Expect: 1 on addr 15
```

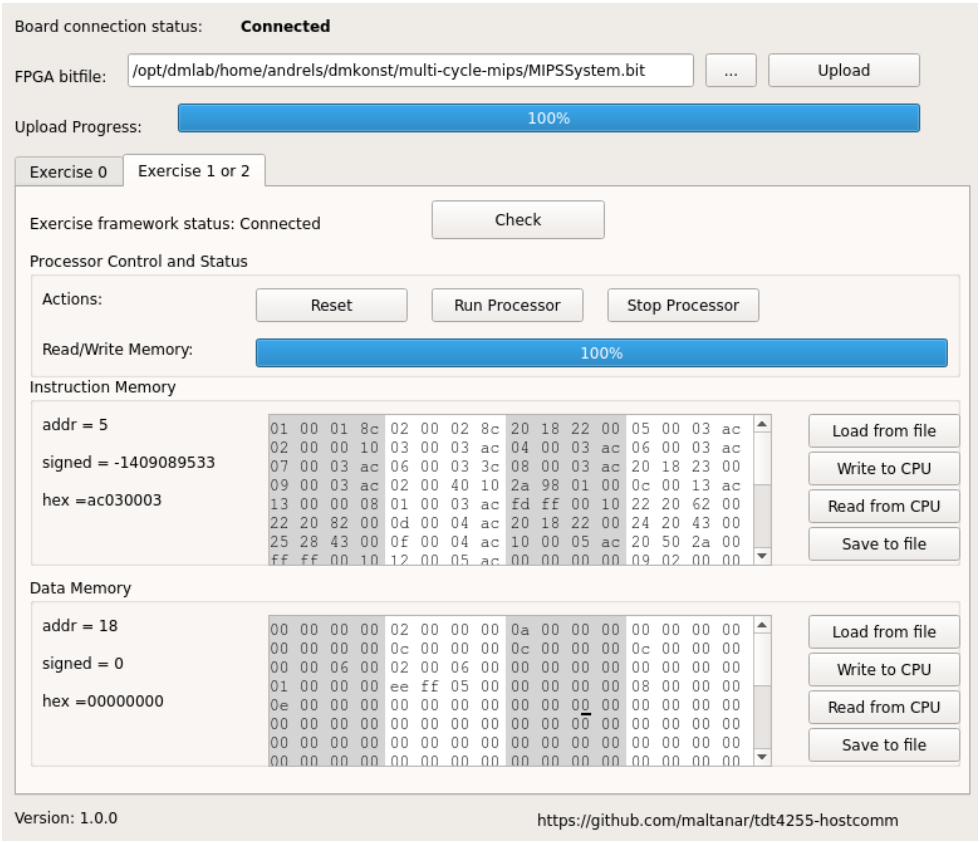

Fig. 9. A sucessful run of the canonical problem, testing all features of the processor.