# TDT 4255 Computer Design: Exercise 2

Guillaume Felley, Marija Mladenovic, Kevin Abraham

November 2015

## Contents

## Abreviation

**IF :** Intruction Fetch

**ID :** Instruction Decode

**EX :** Execution

**MEM :** Memory

**WB :** Write Back

**PC :** Program Counter

**NOP :** No Operation

## Abstract

This report is the result of our work for the Exercise 2 in the context of the course TDT4255 Computer design. The goal was to implement a pipelined microprocessor with different features. A pipelined CPU is the next step in computer design after implementing the multi-cycle CPU which we did in Exercise 1. The complexity will be push a bit further in order to achieve better performances on our processors. Here we will explain the idea of a pipelined processor, analyse it's operation mode, why it has been design like this and discuss about performances and improvements.

# 1 Introduction

In this second exercise we will extend the simple processor from the first exercise by changing the datapath to a pipeline.
The main idea of a pipelined processor is to reorganize the different parts of the CPU to optimize the use of our resources. The main difference with the multi-cycle CPU of Exercice 1 is that one instruction will start executing each clock cycle. And more important the instructions overlaps to maximize the utilisation of resources. Each part of our processor : instruction and data memory, register file, ALU should be busy the most of time. To understand this concept we can draw an analogy with the laudry machine. We have several batch of clothing to processed, for each batch we have to wash in the washing machine, dry in the dryer, fold and store them. The non pipeline approach is to do one thing at a time. This means first wash and then dry, fold and store and only start the second batch once we finish to proceed the first one. If each stage take 30 minutes it took us 2 hours to do one batch and so 8 hours to do 4 batches. Each resources is use one quarter of the time.

The pipeline method is to begin wasing the second batch once we've put the first one in the drier. We processed several batch at a time, this concept is well

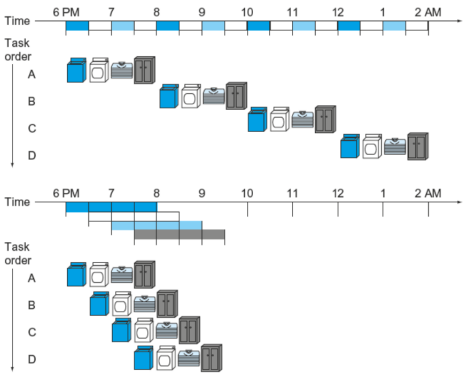


Figure 1: Laundry analogy

illustrated in figure1. At first we can see a sequential approach which take 8 hours and then the pipeline approach which take 3.5 hours. The key is that each resource is being used most of the time.

The major requirement of this second exercise is to implement a pipelined design with 5 stages. This means we will have to cut the datapath of our single-cycle CPU in 5 parts and insert a register between each part. These registers will allow the processing of several instruction at a time. Further, we will have to solve several hazards which can occur during the execution. For example a data dependencies between two consecutive instruction or taken branch. In this last case we will have to flush the datapath because we started to execute the wrongs instructions. The solving of hazards is one of the biggest challenges in this design. Then we are free to implement any other performance improving techniques. Finally we had to follow the MIPS convention[2] for the Operation codes and instruction encoding.

We start by implementing the recommended design figure 2 from the textbook[1] without any hazards solving. Once every instruction were working individually we find a solution for the hazards. We unfortunately didn't have the time to design additional performance improvement technique.

# 2 Solution

## 2.1 Top level

Our top level solution is showed figure 3. As we can see we add 4 big registers which separate our 5 stages :

**Instruction Fetch :** This stage include the program counter and the instruction memory which take one clock cycle to fetch the instruction. The instruction memory is only read and the input address is directly the PC

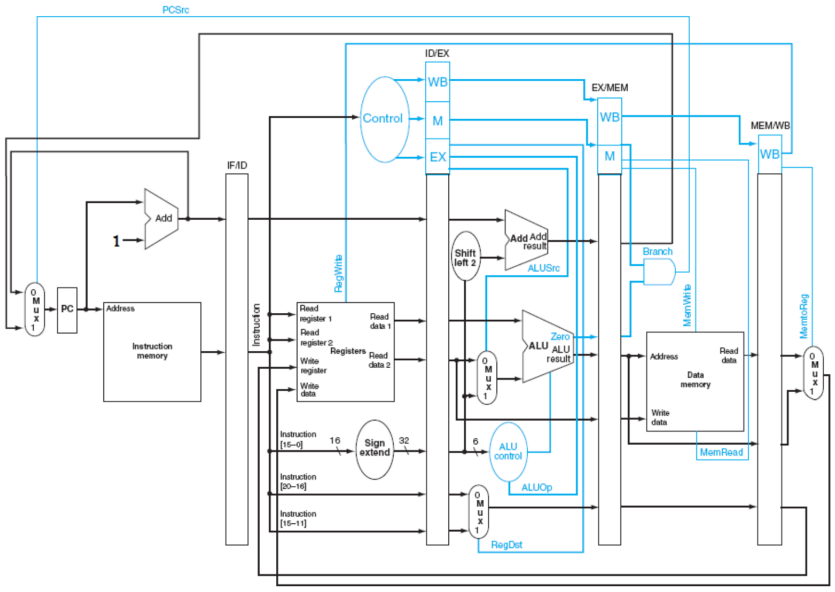


Figure 2: Suggested Architecture

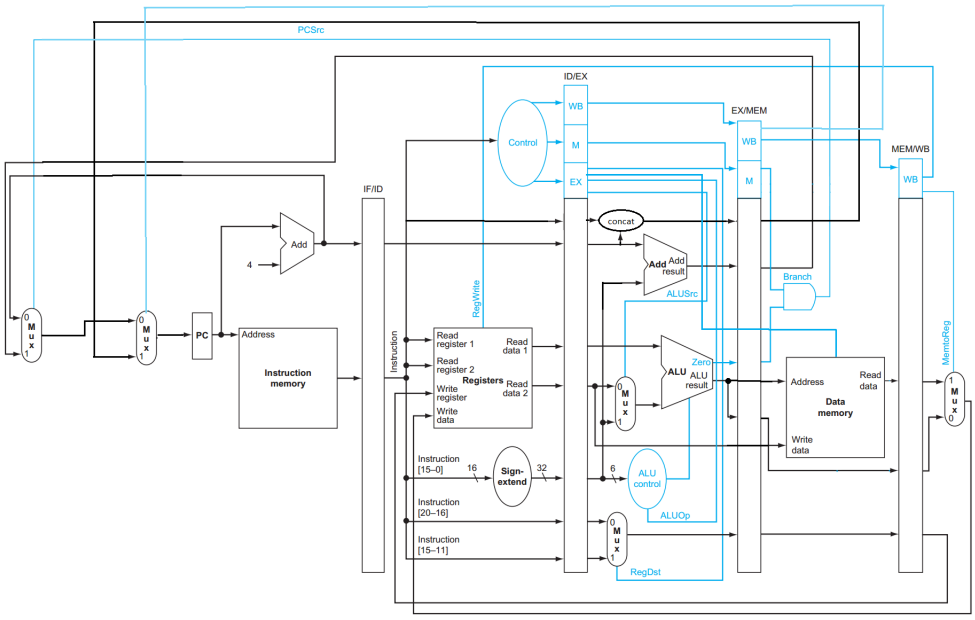


Figure 3: RTL sketch

register. This stage is separated from the next stage by the IF/ID register

**Instruction decode :** Here we have the instruction decoder which separates the different fields of the instruction. The register file, a memory where we don't need to wait a clock cycle to read or write. In our code we can see that the data is write in the register file on a falling edge of the clock, it also works with the code in comment which does the same but this generates a latch warring. So in order to test on the FPGA we prefered the solution without a latch. The control unit which releases most of the control signals required in the next stages. The control signals depend on the Opcode of the instruction. It's a part of the code define by the MIPS convention [**MIPS** ] that we use to know which operation we have to do. The ID is separate from the EX by the register call ID/EX

**Execution :** The ALU is the principal element of this stage. Also, we need a controller for the ALU and an adder to compute branches and other stuff for specific instruction. It is separated from the next stage by the EX/MEM register.

**Memory :** Here the data menmory is included. This stage concerns only the `LOAD` and `STORE` instruction. The separating register with the following is the MEM/WB

**Write Back :** This stage redirects the correct data to the register file if it's needed.

We implemented the same set of instruction as in the previous exercise : `LW, SW, LUI, BEQ, J` and the R-type instruction : `ADD, SUB, AND, OR, SLT` Please refer to the MIPS intruction reference[2] for more information.

The majors differences between our design and figure 2 are. The hardware required to implement the `J` instruction (Jump). We need a control signal, something to concatenate the address extract from the instruction with the 6 first bits of the PC. And a multi-plexer on the program counter to affect the jump address. The result of the ALU and the data which needs to be written in the data memory are directly connected to the memory and do not go trough the EX/MEM register. The reason for it is that we need a rising edge on the clock to write the data in the memory. With this solution the data is written on the clock cycle following the computation of the address.

## 2.2 Hazard solving

We have treated here 2 kinds of hazards : Data hazards and Control hazard. The first one is due to a data dependence in a consecutive set of instructions. A value is needed but it still some where in the data, the write back didn't happen yet. The solution here is to forward the required data at the good place or in some special cases stall the pipeline until the needed data is produce. Control hazard are due to branches and jumps. When our PC has to jump somewhere else in the instruction memory. In those cases the wrong set of instructions

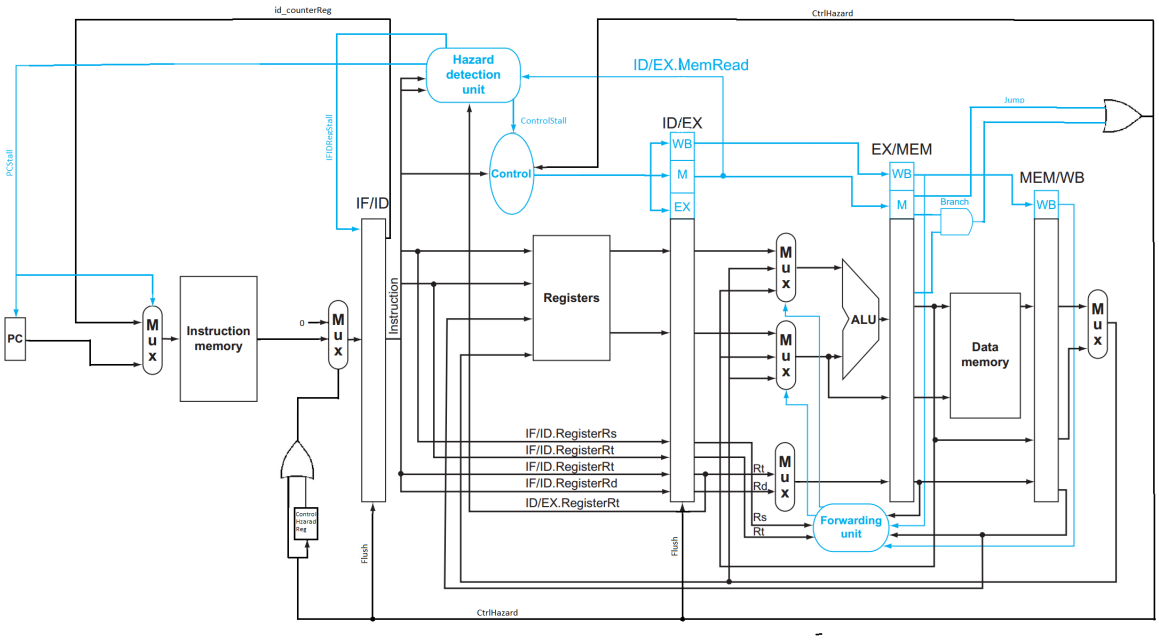


Figure 4: RTL sketch of hazard solving

has started execution. The only solution is to flush the pipeline, this means to empty it. We can see our RTL sketch of the hazard solving system on figure 4.

To solve data hazard we need first to detect the data dependencies. It's done in a module call hazard forwarding unit during the EX stage. It take as inputs the Rt and Rs field (See mips reference[2] for more information) of the current executed instruction. And it will compare it to the destination register (Rd field) of a previous instruction further in the pipeline. Note that for some instruction as `LOAD` and `LUI` the destination register is the Rt field, we need then a multi-plexer to get the right destination register. So if one of the inputs of the ALU it's the one that should go in the Rd register. We will have to redirect the value from the MEM or WB stage back to the ALU. We can see the conditions for the detection of a data hazard in the following code as we implemented it in VHDL. As we can see we need several other information to detect a real data dependencies. First to be sure that the needed value is produce by an instruction

which write in the register file, so we check the control signal RegWrite. And that the used register is not $0. In this case no forwarding should happen, the value associated to $0 is always 0 and should stay 0.

```
if mem_regWrite = '1' and --MEM Hazard on A forward data from MEM stage
   mem_RD /= "00000" and
   mem_RD = ex_RS then
            forwardA <= "10";
elsif wb_regWrite = '1' and -- WB Hazard on A forward data from WB stage
      wb_RD /= "00000" and
      not(mem_regWrite ='1' and mem_RD /= "00000" and mem_RD = ex_RS)
      and wb_RD = ex_RS then
           forwardA <= "01";
else                       -- No hazard on A
           forwardA <= "00";
end if;

if mem_regWrite ='1' and --MEM Hazard one B forward data from MEM stage
   mem_RD /= "00000" and
   mem_RD = ex_RT then
         forwardB <= "10";
elsif wb_regWrite = '1' and -- WB Hazard on B forward data from WB stage
      wb_RD /= "00000" and
      not(mem_regWrite ='1' and mem_RD /= "00000" and mem_RD = ex_RT) and
      wb_RD = ex_RT then
         forwardB <= "01";
else                       -- No Hazard on B data from register file
         forwardB <= "00";
end if;
```

Once we have detected a hazard we will redirect from the MEM or WB the right value to 2 Multi-plexer with 3 entrance control by the forwardA et forwardB signal. See comments in the previous code for more information.

In a case of a load follow by a data depend instruction there is no forwarding possible. Because the value is needed in the execute stage but doesn't exist yet, it will be available only at the end of the MEM stage. So the only solution is to stall the pipeline and then forward the value from the WB stage. This is the condition to stall the pipeline :

```
if ex_memRead = '1' and (ex_RT = RS or ex_RT = RT) then
...        -- Stall the pipeline
else
...        -- No need to stall the pipeline
end if;
```

This is implemented in the Hazard detection unit. If we need to stall we will need to stop the incrementation of the program counter trough a control signal. Force the input of the instruction memory to the previous instruction

trough the same control signal and a multi-plexer. And inject a NOP in the pipeline with the IF/ID register, thus send to the decode unit the `0x00000000` instruction.

We now have to deal with control hazard. The simple way to detect this is get the signals jump or branch in the MEM stage, this is the earliest moment when we can know if we have to jump and where we have to jump. So we have our control signal for control hazard : `CtrlHazard = Jump or Branch` With this signal we now have to flush the EX and ID. To flush these stage we will simply turn all the outputs of IF/ID and ID/EX registers to 0 when the signal CtrlHazard is "1". And finally we have to be sure that nothing will go in the pipeline until the right instruction from jump in instruction memory is availiable. To do so we will force the output of the instruction memory to `0x00000000` with a multi-plexer during teh 2nd clock cycle. After that the right instruction will come.

# 3 Result

## 3.1 Test

We used 4 test bench during our implementation the first one `tb_MIPSProcessor_0.tb` Check if all the instruction work independently and without overlapping instruction by adding a lot of NOP in the testbench provided in support files archive. Once every instruction where working we tested to overlap some instruction in the testbench `tb_MIPSProcessor_1.tb` but still avoiding hazards. Then we made a testbench to test data hazards: `tb_MIPSProcessor_2.tb` in order to test forwarding. And finally use the most complete testbench `tb_MIPSProcessor_3.tb` is simply the testbench provided for exercise one which include control and data hazard. We needed the testbenches 0,1,2 during the different stages of our development but the most advanced one is the number 3. All these test are now working on the Xilinix simulator.

To show that our CPU work we can have look to figure 5. It's from the testbench number 3. We can see the clock, the PC, the fetched instruction and different intermediary result. Each stage a different color. As an exemple we will follow an instruction trough the data path. This instruction is `add $3, $1, $2` which simply add the content of the registers $1 (contains 2) and $2 (contains 12) and place the result in register $3. This instruction is encoded `0x00221820` and can be found at the address 22 of our instruction memory. By following the red line figure 5 from the left we can see : The PC is at the address 22, on next cycle the instruction has been fetch. After it goes through the decode stage and we can see our registers addresses in the fields Rs, Rt and Rd. Next clock cycle we see the 2 inputs of the ALU and the result, so we have $10 + 2 = 12$. On the a stair lower it's the MEM stage noting interesting happen for this instruction here. And then finally it's the WB stage where we can see the destination register, the value which has to be written and the control signal equal to 1 which allow the writting in the register file.

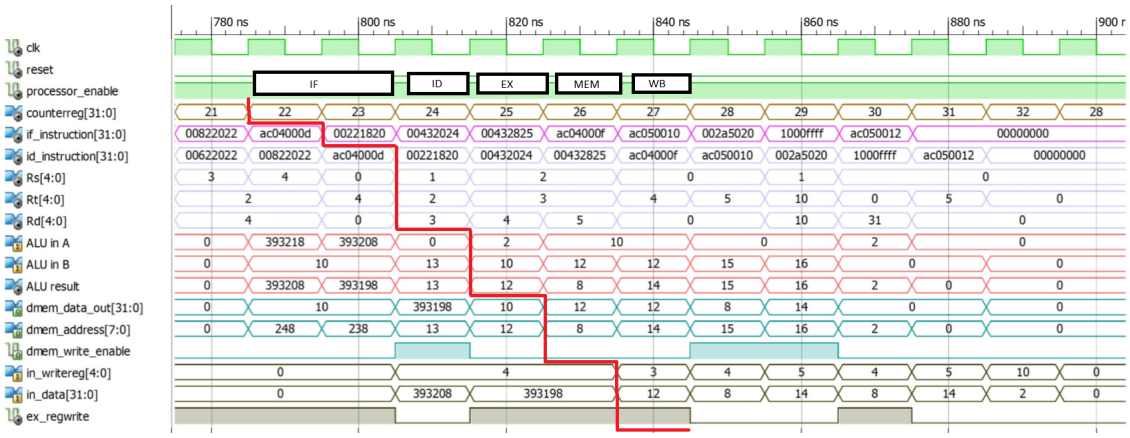


Figure 5: Simulator result

We will now show the different hazards trough the datapath. On figure 6 we can see the execution of the following set of instruction :

```
1. 0x8C020002    lw $2, 2($0)    $2 = 10 Red line figure 6
2. 0x00221820    add $3, $1, $2  $3 = 12 Green line figure 6
3. 0xAC030005    sw $3, 5($0)    Saving value 12 at address 5
4. 0x10000002    beq $0, $0, 2   Jmp address +2 = 8 Orange line figure 6
5. 0xAC030003    sw $3, 3($0)    SKIPPED (Saving value12address 3)
6. 0xAC030004    sw $3, 4($0)    SKIPPED (Saving value 12 at address 4)
7. 0xAC030006    sw $3, 6($0)    Saving value 12 at address 6
```

The `LOAD` following by the `ADD` causes a stall of the pipeline at `PC = 4` once the value is loaded from the data memory the pipline is realased and the value is forwarded from the WB stage (Blue arrow). We can also see a control hazard during the execution of the `BEQ` (Orange line). At `PC = 7` the result of the ALU is 0 so the branch is taken. So the next clock cycle the pipeline is flushed and every value in the EX, ID and IF stage are zero. Instruction at address 5 and 6 are skipped and we start excuting directly the number 7 which is the right instruction.

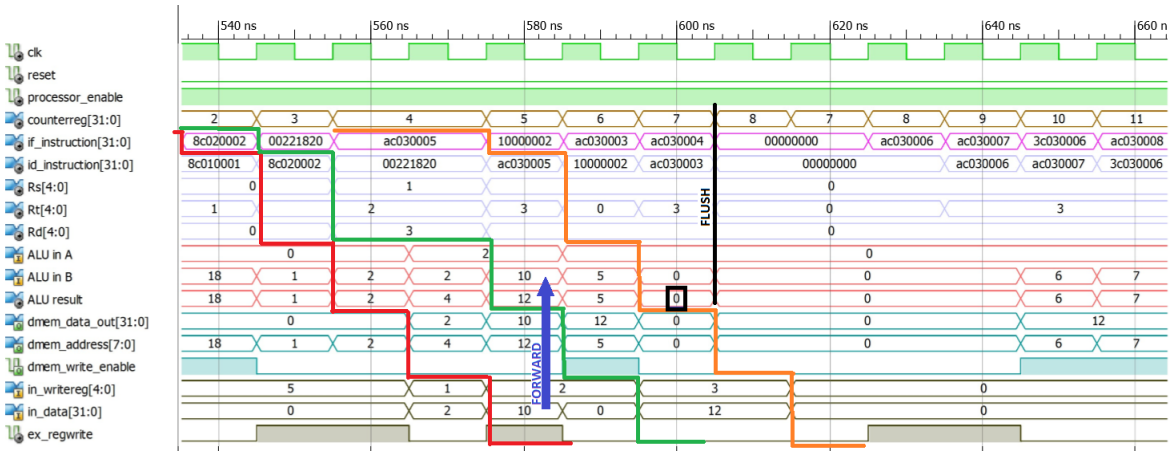


Figure 6: Simulation of Hazard

## 3.2 FPGA

We successfully tested our design on the FPGA this time. We avoided a majority of the warnings. There is no latch warning, the only remaining ones are from the non affectation of the upper bits of our PC and other input address of memory. We can see figure 7 the hostcomm system conected to the running FPGA. We load the same set of instruction as in testbench 3. After execution and reading the data memory we find the right value. For example adresse 9 we find `0x00060002` which is the expected result. The loaded testbench and results will be available in the archive.

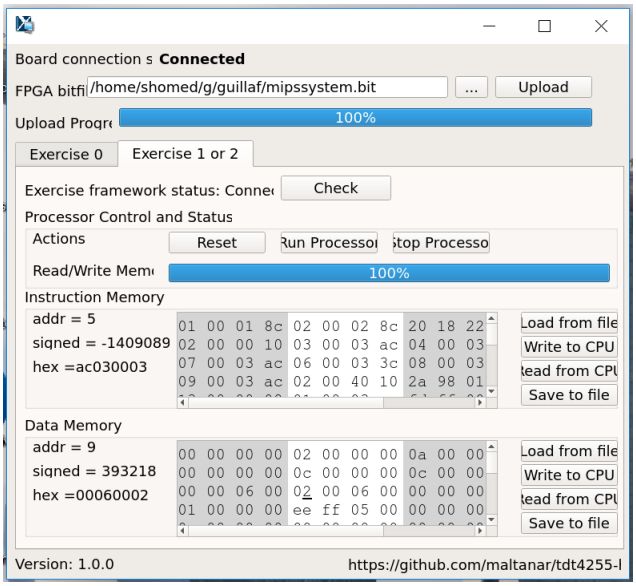


Figure 7: FPGA test

# 4 Discussion

Our processor met the requirements and it achieve better performances than the multi-cycle implemented in exercise 1. As we can see on the simulation the most resources are used the most of the time. If we compare the number of clock cycle it needed 60 clock cycle to reach the last instruction of the testbench and only 34 for the pipeline one. Our major weakness in this project are :

**Branches performances :** We where suppose to implement performances improvements techniques in our design. And branches is the big problem because we choose the hypothesis that branches are always not taken. But most of the time we use branches because of a loop and they are more likely to be taken. Furthermore an average programme contain more or less 25% of branches so it reduce drastically our performances. Solutions should have been to implement branches always taken or a dynamic branch prediction method. It's a technique where you save in a table the result of each branch if it was taken or not taken. Then you execute your branch like his last execution.

**A lack of various testbenches :** We have 4 testbenches but only one test every features of our CPU. We should do more, they are maybe other problem to correct that we didn't see.

**The falling edge :** In the implementation of the register file, it's probably not a proper way to do. But the problem with a rising edge was that we had the old values at the outputs of the register file. It mean it cause a data hazard when we need a value which where produce 3 clock cycle before. We find solutions without the falling edge but they where producing latches warning.

**The report :** The workload for this exercise wasn't well shared. I did this whole exercise alone because my teammates both had "healthcare" problem or they maybe give up the course, i don't know. They will normally justify it. One them (Kevin) finally helped me for a proofreading of the report. So keep in mind that everything has been done by one person. My biggest weaknesses are English syntax and grammar. I tried to do my best in this report but there is still a lot of improvement do to.

# 5 Conclusion

All the technical requirements have been met except the implementation of performance improvement techniques. We designed a pipeline CPU based one the suggested architecture we complete it with few intruction like `LUI` and `JUMP`. We solve data hazard problem by forwarding data or by staling the pipeline if it's really necessary. We also set control hazard by considering branches always not taken and flushing the pipeline in case of jumps or taken branches. Finally it has been tested with success on the FPGA.

# References


[1] John L. Hennessy David A. Patterson. *Computer Organization And Design fifth Edition*. Morgan Kaufman, 2014.

[2] University of Idaho. *MIPS Instruction Reference*. Sept. 1998. URL: `http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html`.