Hazard Detection & Branch Prediction in a MIPS Pipeline Architecture

Student: Iobaj Andrei Sebastian

Group: 30433

Technical University of Cluj-Napoca

Project Laboratory Supervisor: A. Hangan

Contents

1. Introduction

The goal of this project is to implement and test, in VHDL, a unit that can detect and avoid different hazard situations. The list of hazards that will be addressed by this unit includes Data and Control hazards. The circuit will also be able to predict branch instructions via Dynamic prediction.

This unit will be designed with the intent to be easily included in any MIPS pipeline architecture, both on paper and in a VHDL project. It can be used as both a guide to help other people implement their own Hazard Detection and Prevention Unit, but also by people who want to improve the already existing design in their own work.

The circuit shall be implemented and simulated in IDE provided by Vivado. Internally, the MIPS will have multiple programs given to it as tests, that could or not include different types of hazards (there will be a program for each of the hazard types). Having the behavior known beforehand, as to what should the output be, the constant evolution of the program shall be closely monitored and see how the circuit behaves when certain hazard situations should be encountered. If the result is satisfactory, the test is considered as passed.

A MIPS with a pipeline architecture will need to be implemented and tested with a basic small program, making sure that the design works properly on the Basys3 board. Then, another set of programs that will require the MIPS to handle hazards. See how the behavior of the circuit is now, compared to a program that could be handled. Implement the Hazard Detection and Prevention Unit, both as a hardware design and in VHDL. Connect the previously mentioned circuit to the already implemented MIPS and test the created programs. See if the results are satisfactory and conclude the project.

<u>Project Plan:</u>

- [Week 4-14] Documentation: It will be constantly made, in parallel with the development of the project.
- [Week 4-5] Implement in VHDL a working MIPS with pipeline architecture.
- [Week 5-6] Implement a program to test the MIPS and test the design.
- [Week 6] Make sure the basic MIPS implementation works smoothly for the next design step.
- [Week 6-7] Research, identify and decide what approach is best to design the Hazard Detection and Prevention Unit, what is needed to implement it and how to best implement it.
- [Week 7-11] Implement the researched and chosen design, both in a schematic and in VHDL.
- [Week 11-12] Make several short but concrete test programs to show if the MIPS behaves as it should.
- [Week 12] Test the implementation, debug anything if needed and conclude the results.
- [To be added if things go according to plan]

2. Bibliographic Study

MIPS stands for "Microprocessor without Interlocked Pipeline Stages." It's a type of microprocessor architecture that was initially developed by MIPS Computer Systems, Inc. in the 1980s. The MIPS architecture has been widely used in various applications, including personal computers, workstations, embedded systems, and more.

A MIPS pipeline architecture is a type of microprocessor design that enhances instruction throughput by breaking down the instruction execution process into a sequence of stages, such as instruction fetch, decode, execute, memory access, and write-back. This pipelining increases the efficiency of instruction processing but introduces the potential for problems, called hazards.

In the MIPS pipeline architecture, several hazards can occur, potentially impacting the smooth execution of instructions. One common hazard is the "data hazard," arising from dependencies between instructions that require the same data. There are three types of data hazards:

1) Read-after-write (RAW) hazard: Occurs when an instruction needs to read a register before a prior instruction writing to the same register completes. This can lead to incorrect results if not managed properly.
2) Write-after-read (WAR) hazard: Arises when an instruction writes to a register before a prior instruction reads from the same register. While it might not impact the correctness of the computation, it can lead to unexpected behavior if instructions are not sequenced correctly.
3) Write-after-write (WAW) hazard: Happens when two instructions attempt to write to the same register, and the order of their execution affects the final value in the register.

These hazards need to be managed to ensure proper instruction execution and maintain program correctness. Techniques like forwarding (also known as bypassing) and stalling (inserting no-operation cycles) are used to mitigate these hazards in MIPS pipelines.

Another type of hazard is the "control hazard," which occurs due to changes in the program flow, like branches or jumps. Predicting branches and ensuring the correct flow of instructions without stalling the pipeline unnecessarily is crucial to prevent delays caused by control hazards. Techniques like branch prediction and instruction reordering help mitigate these hazards by speculatively executing instructions based on predictions.

Structural hazards are another concern in the MIPS pipeline architecture, arising from resource conflicts when multiple instructions need the same hardware component simultaneously. These hazards occur when the pipeline stages require access to the same resource concurrently, such as the memory unit or the arithmetic logic unit (ALU). For instance, if two instructions need access to the memory unit simultaneously, a structural hazard arises. To address this, MIPS processors often use techniques like resource duplication, where critical components are replicated, or scheduling techniques to manage the allocation of resources efficiently across multiple instructions, minimizing the impact of structural hazards on the pipeline's performance.

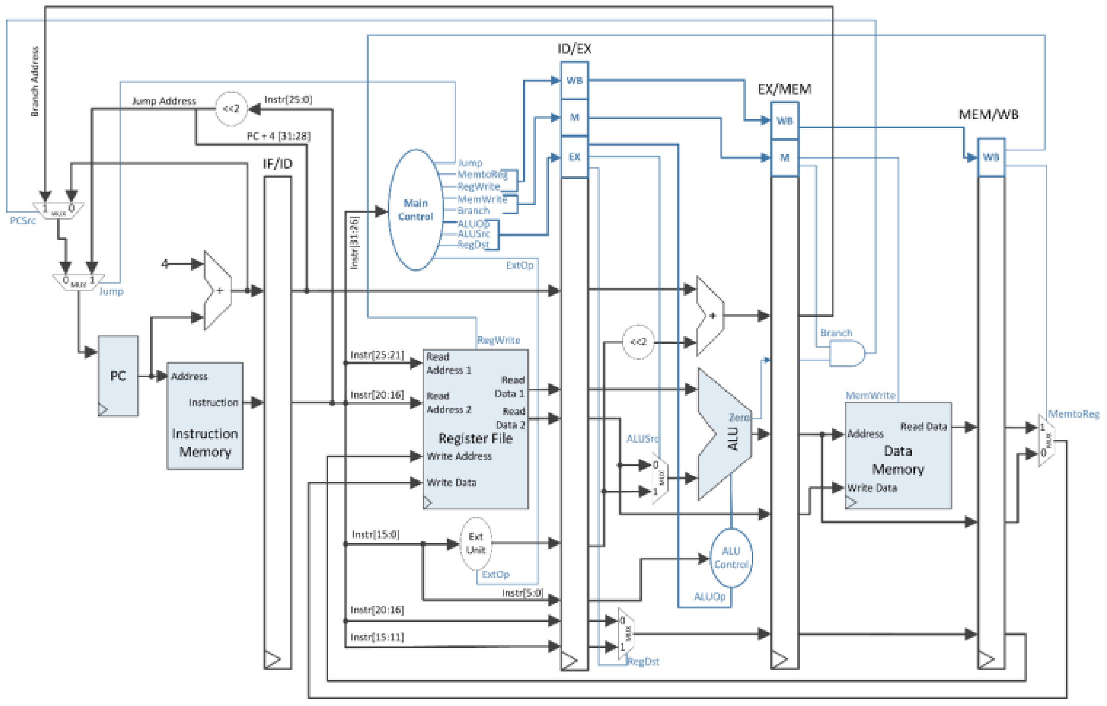


Fig. 1: MIPS with a pipeline architecture

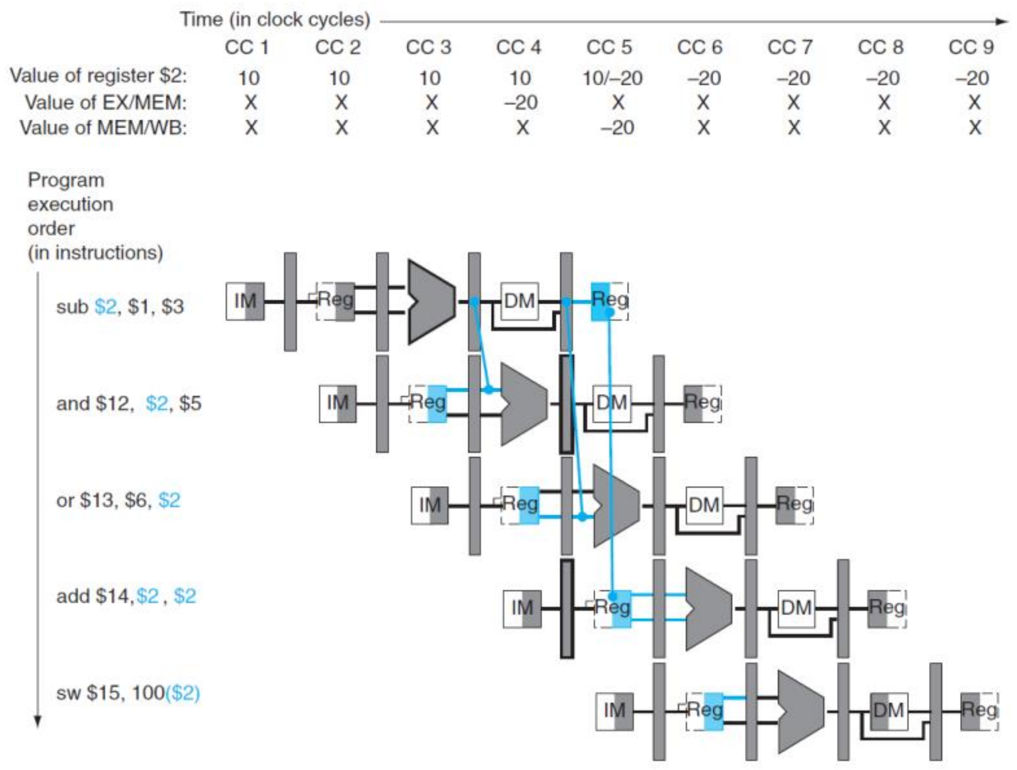
3. Analysis

3.1 Hazard Detection

As shown above, hazards do pose a threat to the correct execution of different programs in a MIPS processor. That is why there exist several methods of predicting and solving these issues. That is why some of the following features are much needed in the final product:

- A fully working MIPS without the hazard detection unit, with different implemented instructions and a testing program.
- Data Hazard detection and solving.
- Control Hazard detection and solving.
- A fully working MIPS with the hazard detection units and with different implemented instructions and a testing program.

Now we need to analyze the types of hazards that could be solved in our design choice. One of the most elegant ways to implement a hazard detection and solving unit, is by also having a forwarding unit. Doing it like this provides flexibility in terms of data access and availability, making it much easier to pass data to different components at different. Like this, we could easily solve the Read After Write (RAW) hazards.



Time (in clock cycles)
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Value of EX/MEM: X X X –20 X X X X X
Value of MEM/WB: X X X X –20 X X X X
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14,$2 , $2
sw $15, 100($2)
IM
Reg
DM

Fig. 2: Forwarding example

In Fig. 2, we have presented an example of how forwarding works to avoid RAW hazards. Let us analyze this and see how it is done.

Firstly, we notice that the first instruction is executed immediately. The second and third instructions depend on the new value that will be stored in the $2 register. We cannot use the value that currently resides there, because we shall have wrong results in the next operations. We need to wait for the MIPS to write back the result in memory. Or do we?

That is where the data forwarding comes in. The result is actually available much early. For the second instruction, we can take the result from the EX/MEM register file and for the third instruction, we can take the result from the MEM/WB register file. In case of the last two instructions, the process is trivial.

Secondly, this raises a new question. If, indeed the previous example shows 2 cases from where the data could be taken, just how many are there? Two cases are, obviously, the above ones, where we take the data from the EX/MEM register file and another where we take the data from the MEM/WB register file. Another case where there are problems are the SW and LW instructions.

In case of a store instruction immediately after an instruction that writes to the destination register, we can forward the data only from the MEM/WB register file. In case there is an instruction with no hazard possibility in between them, then the process is, again, trivial.

In the case of the load instruction, then the problem is very much the same, theoretically. The data is available in the MEM/WB register file and can be forwarded to the next instruction. But there is a small catch. It doesn't work!

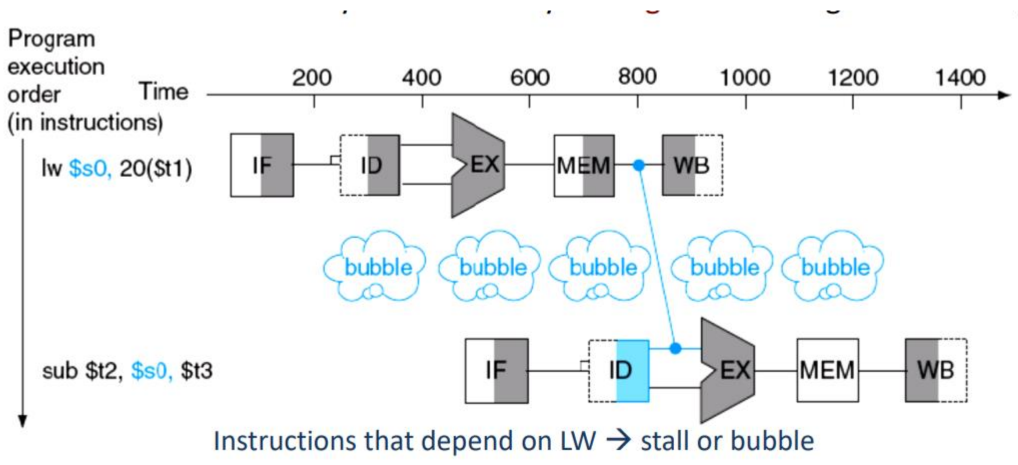


Fig. 3: Load Instruction problem

In case of the LW instruction, forwarding doesn't work. The instruction that follows a LW reads the register written by LW. As such LW can cause hazards by itself. The solution is to stall the program, in order to provide the processor with enough time to write the data into the needed register. We shall need a special hazard detection unit to specifically target this case that may occur.

Another two data hazards that will be treated indirectly, just by implementing the Hazard Unit and the Forwarding Unit are WAR and WAW hazards.

WAW hazards, known as name dependencies or output dependencies, introduce challenges related to maintaining the proper execution order of instructions that aim to write to the same destination register. In our pipeline design, the forwarding unit emerges as a central component in addressing these WAW hazards. This unit can efficiently detect scenarios where multiple instructions seek to write to the same register and in forwarding the most recent data to the instruction ready to execute.

WAR hazards, also referred to as anti-dependencies, introduce a distinct set of challenges associated with the sequencing of read and write operations on a register. In our pipeline architecture, which incorporates a hazard detection and solving unit bolstered by the forwarding unit, we tackle WAR hazards effectively. To address these challenges, our design integrates register file synchronization. This synchronization ensures that data written by a later instruction is withheld until the data has been successfully read by the dependent instruction, thereby resolving WAR hazards.

3.2 Branch Prediction

In MIPS pipeline architecture, branch prediction anticipates whether conditional branches will be taken or not, crucial for uninterrupted instruction flow. It uses historical data to forecast branch outcomes, allowing processors to preemptively fetch and execute instructions. This strategy minimizes stalls in the pipeline, optimizing overall performance by making educated guesses about branch directions. Branch prediction is directly correlated to control hazards. They occur whenever the pipeline makes an incorrect branch prediction decision, resulting in instructions entering the pipeline to be discarded. This is called a pipeline flush. As such, we shall need a unit that will flush the pipeline if a branch was not predicted correctly.

There are multiple possible solutions to a problem such as this, however, the optimal approach is Dynamic Branch Prediction. This choice is based on the fact that, a MIPS program is not always predictable. Instead of relying solely on static patterns, dynamic prediction adapts to the behavior of branches as the program runs. In a MIPS pipeline architecture, dynamic branch prediction is advantageous because it adjusts to changing program conditions, enhancing accuracy compared to static approaches. This adaptability minimizes pipeline stalls by more accurately predicting branch outcomes, allowing for efficient instruction fetching and execution.

The Dynamic Branch Prediction works somehow as follows: We need a Branch History table that is a combination of Branch Target Buffer (BTB) and a two-bit predictor.

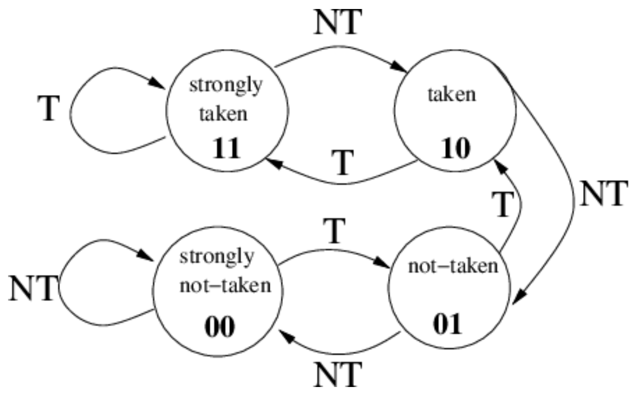


Fig. 4: Principle of working for a 2-bit predictor table

The dynamic branch prediction mechanism involves two stages: Read BHT and Predict, and Check Prediction and Update BHT.

- In the "Read BHT and Predict" stage, the mechanism selects the next program counter (PC) address using the branch history table. If a branch is predicted, the target address associated with the predictor is sent to memory. Otherwise, the next PC address is set to PC + 1, following the usual behavior for non-branch instructions.
- In the "Check Prediction and Update BHT" stage, which occurs one clock cycle after the prediction, the system checks the accuracy of the prediction. If the prediction differs from the actual branch outcome, a flush is generated, and the next fetched instruction aligns with the correct branch outcome. If the prediction matches the actual outcome, the program execution continues without interruption.

The branch history table is then updated based on the known branch outcome. If the branch was taken, the predictor at the corresponding address is incremented, and the branch target address is updated. If the branch was not taken, the predictor is decremented (unless it reaches zero), and the target address remains unchanged.

The first step to set up the ground for a branch prediction unit is to move most of the branch computation logic to the Instruction Decode (ID) stage of the pipeline. This way, it is possible to have an instruction be executed before the computation of the branch address is made and known. As such, the flush will only focus on removing the data that has passed in the IF/ID register file.

By moving the branch computation logic to the ID, we observe the rise of two possible hazardous scenarios. One targets the data dependencies between the branch instruction and the preceding instruction. In this case, a stall needs to be inserted. The other case refers to the situation of the Load instruction. It is possible that, during the execution of the program, we want to load some data, and the destination of the load is a register used for determining the branch address. The solution, as before, is to insert a stall here as well.

4. Design

We shall start our thinking process from the design presented in Fig. 1. In this schematic, we have presented a MIPS with a pipeline architecture, with no hazard detection. This kind of MIPS processor works with different instructions. In our case, the processor supports the instructions below, with the following syntax:

ADDI – Add immediate

Description: Adds a register and a signed immediate value and stores the result in a register

Operation: $t ⇓ $s + imm; advance_pc (4);

Syntax: addi $t, $s, imm

Encoding: 0010 00ss ssst tttt iiii iiii iiii iiii

LW – Load word

Description: A word is loaded into a register from the specified address.

Operation: $t ⇓ MEM[$s + offset]; advance_pc (4);

Syntax: lw $t, offset($s)

Encoding: 1000 11ss ssst tttt iiii iiii iiii iiii

SW – Store word

Description: The contents of $t is stored at the specified address.

Operation: MEM[$s + offset] ⇓ $t; advance_pc (4);

Syntax: sw $t, offset($s)

Encoding: 1010 11ss ssst tttt iiii iiii iiii iiii

BNE – Branch on not equal

Description: Branches if the two registers are not equal

Operation: if $s != $t advance_pc (offset << 2)); else advance_pc (4);

Syntax: bne $s, $t, offset

Encoding: 0001 01ss ssst tttt iiii iiii iiii iiii

ANDI – Bitwise and immediate

Description: Bitwise ands a register and an immediate value and stores the result in a register

Operation: $t ⇓ $s & imm; advance_pc (4);

Syntax: andi $t, $s, imm

Encoding: 0011 00ss ssst tttt iiii iiii iiii iiii

ORI – Bitwise or immediate

Description: Bitwise ors a register and an immediate value and stores the result in a register

Operation: $t ⇓ $s | imm; advance_pc (4);

Syntax: ori $t, $s, imm

Encoding: 0011 01ss ssst tttt iiii iiii iiii iiii

J – Jump

Description: Jumps to the calculated address

Operation: PC ⇓ nPC; nPC = (PC & 0xf0000000) | (target << 2);

Syntax: j target

Encoding: 0000 10ii iiii iiii iiii iiii iiii iiii

These instructions can be used to develop simple, yet effective programs to test the performance of the MIPS. We shall first put into perspective one of the most important ideas in such a project, that being a Forwarding Unit.

Forwarding to the Execute (EX) stage of the pipeline involves the detection of dependencies between the instruction currently residing in the EX stage and preceding instructions in earlier pipeline stages, such as the Memory (MEM) or Write Back (WB) stages. The forwarding process ensures that the EX stage can access the most recent and relevant data required for execution. By employing multiplexers and intelligent routing, the forwarding unit selects the appropriate data source, which may be registers or data from various pipeline registers.

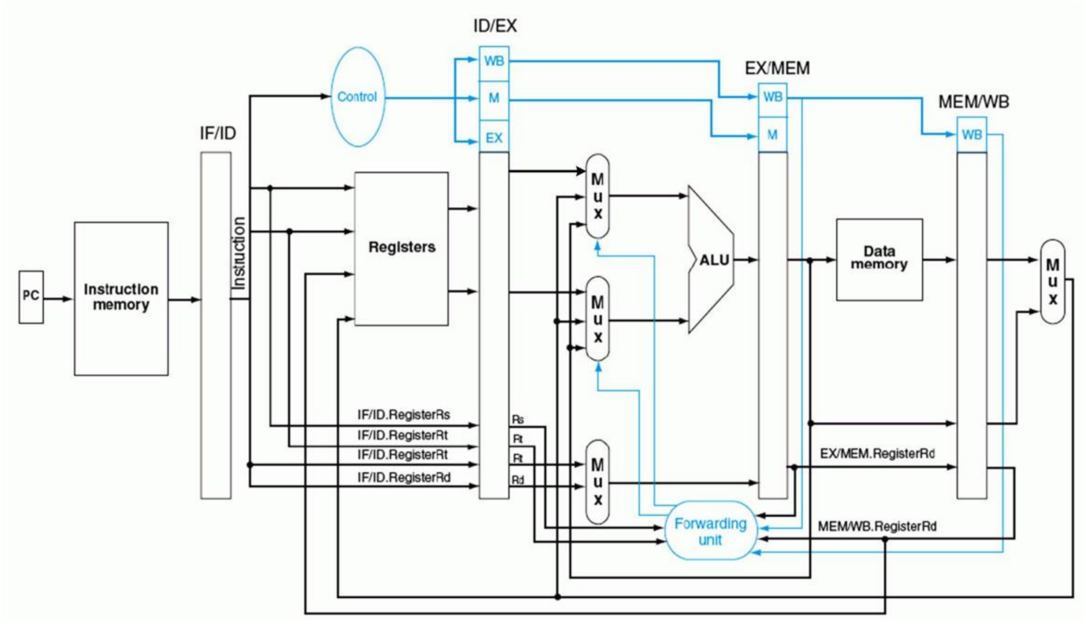


Fig. 5: MIPS with an implemented Forwarding Unit in EX

Forwarding to the Memory (MEM) stage of the pipeline is responsible for ensuring that data reaches the memory, taking into account dependencies the last two instruction. As we do not have access, in the MEM stage of the second to last instruction, we shall need to insert a buffer after the WB stage to have access to the data written at the address of the destination register.

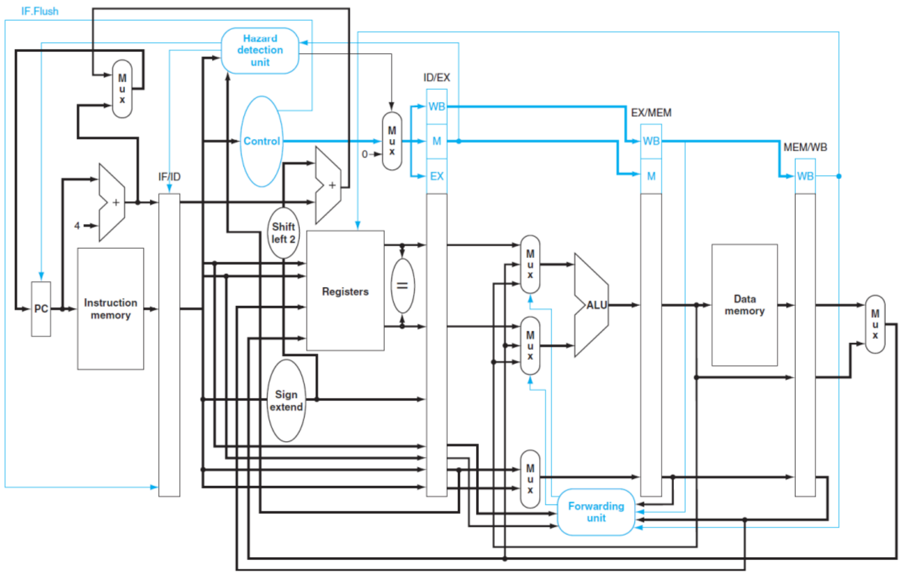


Fig. 6: MIPS with an implemented Forwarding + Hazard Detection unit

The Hazard Detection unit has as its main purpose to detect and mitigate three key types of hazards: data hazards and branch hazards involving the EX/MEM and ID/EX pipeline stages, and data hazards between the ID/EX and IF/ID stages.

The hazard detection unit compares register addresses and control signals in different pipeline stages to determine if such hazards exist. If detected, it asserts a stall signal, pausing the pipeline to resolve the hazard and ensure correct instruction execution, thereby maintaining program correctness and preventing errors arising from dependencies between instructions.

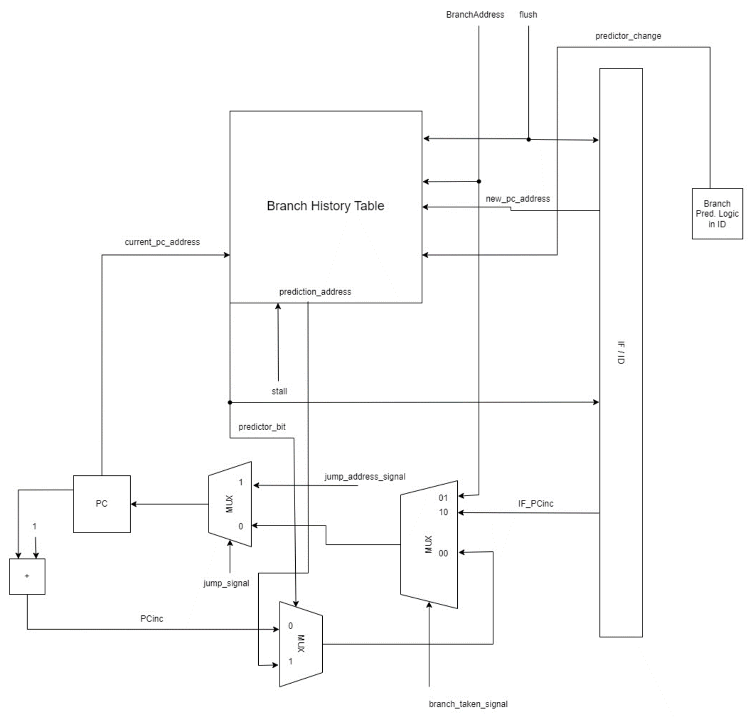


Fig. 7: Basic schematic for a Dynamic Branch Prediction

The last added component will be the Branch Prediction Unit. To achieve a correct design of one such unit, we must think about two steps:

- A mechanism for flushing
- Branch prediction according to the proposed idea of using a Branch History Table.

The mechanism for flushing shall be triggered by a signal. The value is obtained by comparing two values, if the branch actually happened and if the branch was predicted to happen. If both values are the same, no flush is needed. The purpose of this signal is to clear the IF/ID pipeline register file.

The branch prediction is also presented above. As previously stated, the structure of a branch history table, composed of 2-bit predictors and target addresses. The prediction happens in the IF stage and then feeds it to the branch history table for comparison.

Another aspect of the design is to move the branch detection logic and address computation to the ID stage of the pipeline, as stated above. The purpose of this move has been already explained above, in the Analysis step of the documentation.

5. Implementation

The first steps in the implementation stage are to implement a MIPS with a pipeline architecture and the thorough testing of it to ensure good functionality. After that, the following components required, those being the forwarding units, hazard detection unit and branch prediction unit.

1) MIPS Pipeline Architecture

The MIPS is divided into 5 modules. The Instruction Fetch, Instruction Decode, Execution Unit, Memory and Write Back. These 5 modules are made, separately, in 5 different files to ensure good coding style and clarity. There is also a main module which takes all of them, combining them into the final MIPS implementation. The pipeline register files are also situated there, along with the control unit for generating the control signals needed for the MIPS.

As the purpose of this project is not to focus on the simple MIPS with pipeline architecture, I only wanted to mention it as to give an idea of the starting point of this project work.

2) Forwarding Unit

The Forwarding Unit is designed in mind with just a few processes to achieve the desired goals. As it was presented both in Analysis and Design, we have to forward the data to two multiplexers before the ALU and another forwarding is to the MEM stage.

The forwarding to the ALU involves the source and target register. We shall denote the source as A and target as B.

The A forwarding shall involve a process with the below input data:

- ID_EX_RegRs,
- EX_MEM_RegWrite,
- EX_MEM_RegDst,
- MEM_WB_RegWrite,
- MEM_WB_RegDst

Then, we shall need to check the following conditions:

- If the EX/MEM stage is writing to the register AND the Source register in the ID/EX is the same as the Destination register in the EX/MEM stage AND we are not trying to write in the 0 register, then we are forwarding from the EX/MEM pipeline register file.
- If the MEM/WB stage is writing to the register AND the Source register in the ID/EX is the same as the Destination register in the MEM/WB stage AND we are not trying to write in the 0 register, then we are forwarding from the MEM/WB pipeline register file.
- In all the other cases, no forwarding is required.

The B forwarding involves the same logic, just that, we do not check the source register, but the target register. As such, these are the inputs of the process:

- ID_EX_RegRt,
- EX_MEM_RegWrite,
- EX_MEM_RegDst,
- MEM_WB_RegWrite,
- MEM_WB_RegDst

Then, we shall need to check the following conditions:

- If the EX/MEM stage is writing to the register AND the Target register in the ID/EX is the same as the Destination register in the EX/MEM stage AND we are not trying to write in the 0 register, then we are forwarding from the EX/MEM pipeline register file.
- If the MEM/WB stage is writing to the register AND the Target register in the ID/EX is the same as the Destination register in the MEM/WB stage AND we are not trying to write in the 0 register, then we are forwarding from the MEM/WB pipeline register file.
- In all the other cases, no forwarding is required.

The last case we need to consider is forwarding to the MEM stage. This is done via the same structure as above, a process with the following inputs:

- EX_MEM_RegRt,
- EX_MEM_MemWrite,
- MEM_WB_RegDst,
- MEM_WB_RegWrite,
- BUF_WB_RegDst,
- BUF_WB_RegWrite

Then we shall check the following:

- If the MEM/WB stage instruction is trying to write in the register. If that is the case, then we shall check if the Target register in the EX/MEM stage is equal to the Destination register in the MEM_WB stage AND the instruction in the EX/MEM register file is writing to memory. If that is the case, we are forwarding from the MEM/WB pipeline stage. Otherwise, no forwarding is required.
- If the BUF_WB stage instruction wrote in the register, then we check if the Target register in the EX/MEM stage is equal to the BUF_WB Destination register AND if the instruction in the EX/MEM register file is trying to write in memory. If these conditions are true, then we are forwarding from the buffered WB register. If not, we are not forwarding.

3) Hazard Detection Unit

The implementation principle of the hazard detection unit is similar to the one for the forwarding unit. We use processes to achieve the desired result. The logic was presented, just as before during the Analysis and Design steps of this documentation. We create ourselves a process with the following inputs:

- ID_EX_RegWrite,

- ID_EX_MemRead,
- ID_EX_RegRt,
- IF_ID_RegRs,
- IF_ID_RegRt,
- branch_input

The process involves checking the one load data hazard condition we identified before, as well as the two branch hazards that might occur.

First, let us implement the data hazard detection. For this, we simply need to check if the Source register file in the IF/EX stage is the same as the Target register file in the ID/EX stage OR if the Target register file in the IF/EX stage is the same as the Target register file in the ID/EX stage. If this is true and the instruction in the ID/EX stage is reading from memory, then we must insert a stall.

In the other two hazard cases, we need to check, for one, the branch hazard regarding the EX/MEM stage and the IF/ID stage and for the other the branch hazard regarding the ID/EX MEM stage and the IF/ID stage.

For the first one, we check if the instruction is indeed a branch instruction and if the instruction in the EX/MEM stage is trying to write in a register. If this is true and the Target register from the IF/ID stage is the same as the address that comes from the EX/MEM pipeline register OR the Source register is the same as the address that comes from the EX/MEM pipeline register then we need to insert a stall.

For the second one, we check if the instruction is indeed a branch instruction and if the instruction in the ID/EX stage is trying to write in a register. If this is true and the Target register from the IF/ID stage is the same as the address that comes from the EX stage OR the Source register is the same as the address that comes from the EX stage then we need to insert a stall.

4) Branch Prediction

The Branch Prediction Unit, just as stated in the Analysis and Design part of the project documentation, will consist of a branch history table. This table shall be represented as a record a 2-bit predictor and a predicted address. The table shall contain 32 of such structures.

After that, we shall make a process including these inputs:

- clk,
- pc_enable,
- current_pc_address,
- new_pc_address,
- new_target_address,
- predictor_change,
- branch

Then we shall assign the current predictor with the predictor of the next pc address and check if it needs incrementing or decrementing based on the clock, pc_enable, branch and the value of the predictor_change. If it is 1, we increment the predictor and if it is 0 we decrement the predictor. This was all presented in figure 4.

If we do not have a branch in the previous instruction and the flush signal is instead on logical 1, then we need to update the branch history table with the new_target_address. At the end we update the predictor in case it suffered modifications.

6. Testing and validation:

1) MIPS Pipeline Architecture

The general MIPS Pipeline Architecture was tested on the FPGA board with an infinite Fibonacci sequence. The program is presented below:

```
B"001_000_001_0000000", --addi $1,$0,0
B"001_000_010_0000001",   --addi $2,$0,1
B"001_000_011_0000000", --addi $3,$0,0
B"001_000_100_0000001", --addi $4,$0,1

B"011_011_001_0000000", --sw $1,0
B"011_100_010_0000000", --sw $2,0

--NOOP
B"000_000_000_000_0_111",

B"010_011_001_0000000", --lw $1,0
B"010_100_010_0000000", --lw $2,0

--NOOP
--NOOP
B"000_000_000_000_0_111",
B"000_000_000_000_0_111",
B"000_001_010_101_0_000", --add $5,$1,$2
B"000_000_010_001_0_000", --add $1,$0,$2

--NOOP
B"000_000_000_000_0_111",

B"000_000_101_010_0_000", --add $2,$0,$5

B"111_0000000001001", --j 9

--NOOP
B"000_000_000_000_0_111",
```

The outcome of this program was satisfactory with the MIPS working correctly. So, we continued the testing with testbenches for the other components we implemented.

2) Forwarding Unit

For the Forwarding Unit, we have implemented 5 different testing scenarios. Each test scenario will be presented below with the duration it is represented. Also, an image of the waveform will be inserted.

Test case 1: Forwarding scenario 1 (from 0ns to 10ns)

We want to forward A from the EX/MEM and NOT forward B.

Test case 2: Forwarding scenario 2 (from 10ns to 20ns)

We want to forward A from the EX/MEM and B from MEM/WB.

Test case 3: Forwarding scenario 3 (from 20ns to 30 ns)

We want to NOT forward A and forward B from EX/MEM

Test case 4: No forwarding scenario (from 30ns to 40ns)

We provide random values to the control signals random addresses for the registers (that do not match)

Test case 5: Forwarding scenario 1 + Forwarding to Memory Stage (40ns – onward)

We want to forward A from the EX/MEM and NOT forward B

We want to forward from MEM/WB for Memory Stage

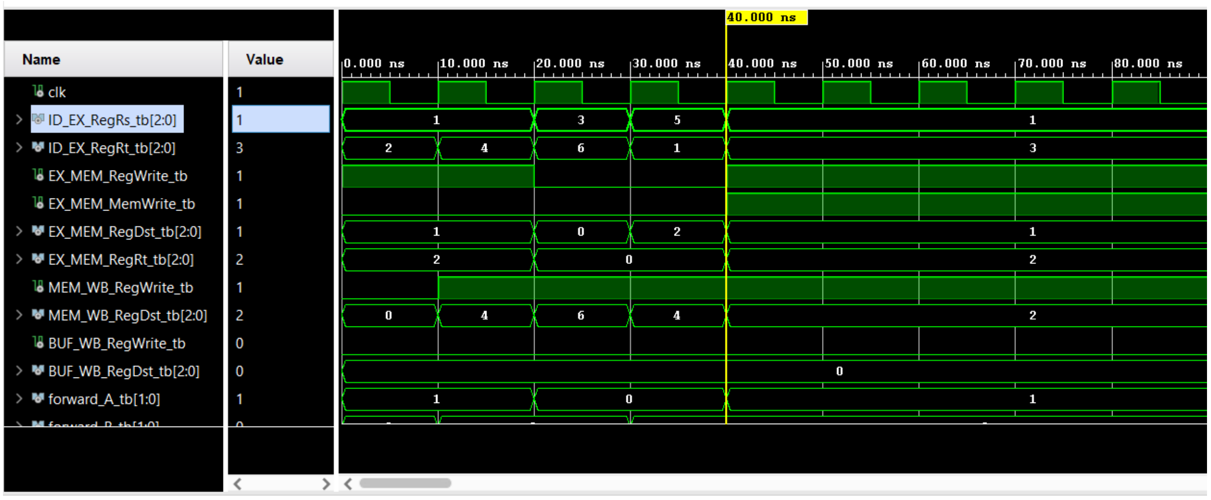


Fig. 8: Waveform for the Forwarding Unit Testbench

3) Hazard Detection Unit

For the Hazard Detection Unit, we have implemented two test, one in which the inputs should determine a data hazard and one in which a data hazard should not be identified.

Test case 1: No hazard detected scenario (from 0ns to 10ns)

Test case 2: Hazard detected scenario (from 10ns - onwards)

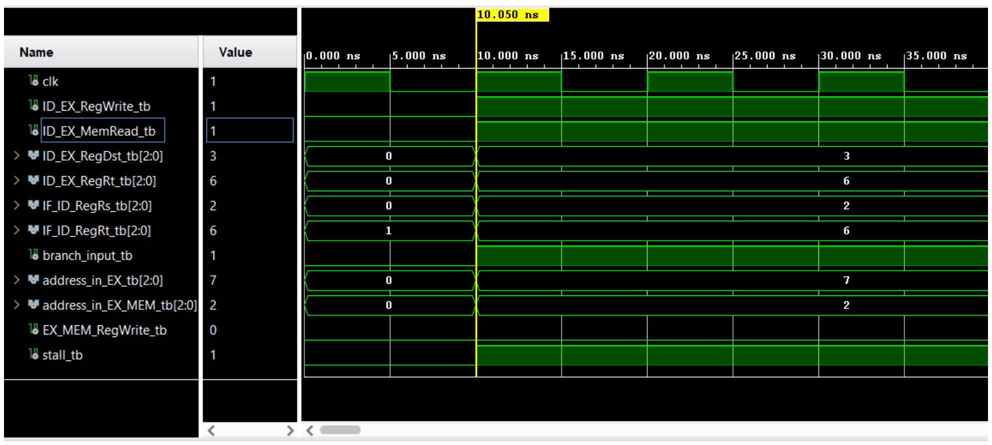


Fig. 9: Waveform for the Hazard Detection Unit Testbench

4) Branch Prediction

For the branch prediction, we have implemented 4 tests to test the normal behavior of the module.

Test case 1: Testing normal behavior without changes (from 0ns to 10ns)

Test case 2: Testing branch taken scenario (from 10ns to 20ns)

Test case 3: Testing branch not taken scenario (from 20ns to 30ns)

Test case 4: Test to see if we change the addres in bht (from 30ns - onwards)

At test4, the prediction_address is the signal we want to observe. Due to the data codification, it was pretty complicated to hard-code data in the record and output the data in the testbench. However, we see that, on the rising edge of the clock, we value gets changed, but on the falling edge of the clock, the value changes back to the hardcoded value. That is the case of strange hardcodation, but it was the best it could be done at that time. We can see though that the module works.

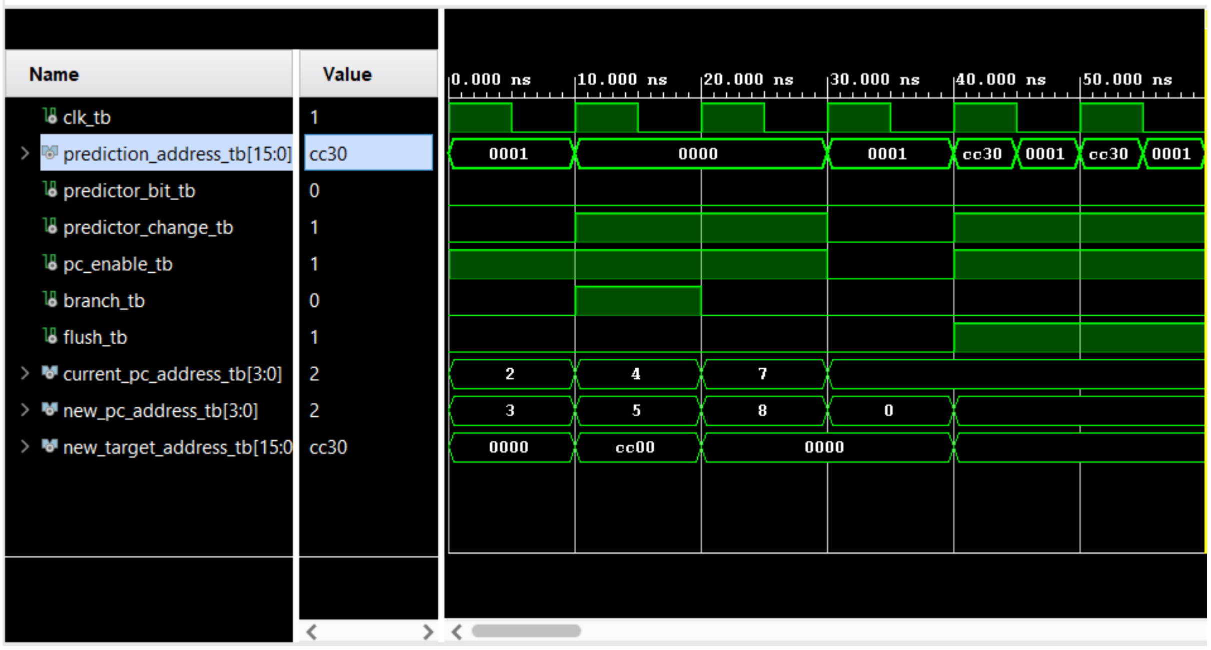


Fig. 10: Waveform for the Branch Prediction Testbench

5) Complete Design Testing

In order to test the correct functionality of the MIPS with every module implemented, the first tested program with the Fibonacci sequence was ran, except this time, all the NOOPs were removed. The goal is to achieve the same functionality.

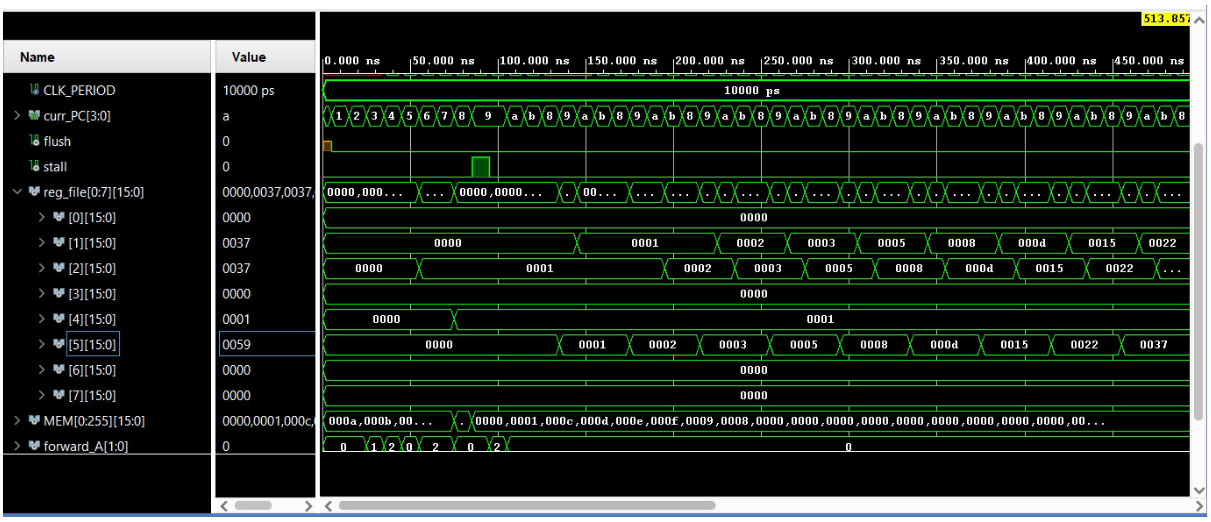


Fig. 11: Waveform for the complete design with the original program (no NOOPs)

Now that we know the MIPS works correctly, based on the fact we achieve the same output result, we need to test the unit under certain conditions. We shall have to develop several short programs that will test the MIPS in different possible hazardous situations and see if the programs executes as it should.

The first test will be to test the forwarding capability of the MIPS. We shall do this by first testing the forwarding functionality to the Execution Unit. The simplest way to achieve this is to use a register as the destination of an addition or subtraction and then use it in several other instructions as the source or target of the same operation. The used program is inserted below:

B"000_000_000_000_0_111", -- NOOP

B"000_100_011_010_0_001", --SUB $2 = $4 - $3

B"001_010_101_0000001", --ADDI $5 = $2 + 1

B"000_101_010_110_0_000", -- ADD $6 = $5 + $2

B"000_010_001_111_0_000", -- ADD $7 = $2 + $1

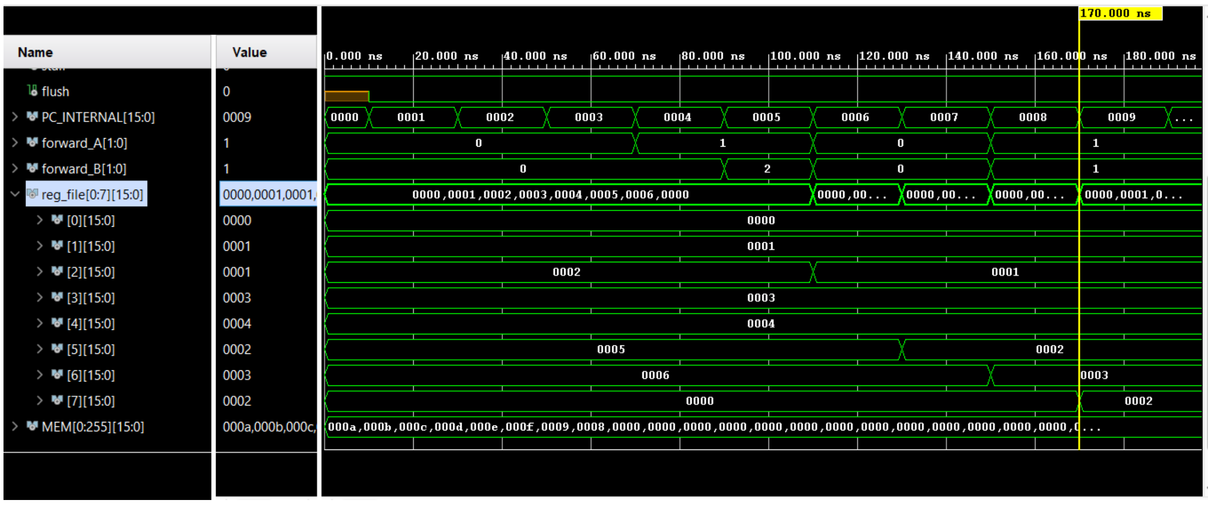


Fig. 12: Waveform for the testing of EX forwarding

As we can see, every instruction is executed correctly, each result being correctly placed in the desired register and the correct, updated, values of registers, $2 and $5 are used as parameters for the last two additions.

Without forwarding, the results would be very different. For example, the first 3 instructions would have as parameters, the initial values of the registers and, besides the first one, all shall have incorrect values. The last instruction on the other hand, shall benefit from the correct and updated value of the register $2 and the result shall be correct, however that is only because the instruction had time to finish executing.

The next test will be to test the forwarding capability to the MEM stage of the MIPS pipeline. This is easily achieved by changing the contents of a certain register and then inserting two SW instructions that have as desired register, the previously mentioned one. The test program is inserted below:

B"000_000_000_000_0_111", -- NOOP

B"000_001_100_011_0_000", -- ADD $3 = $1 + $4

B"011_000_011_0000000", -- SW $3 $0 1

B"011_000_011_0000001", -- SW $3 $0 2

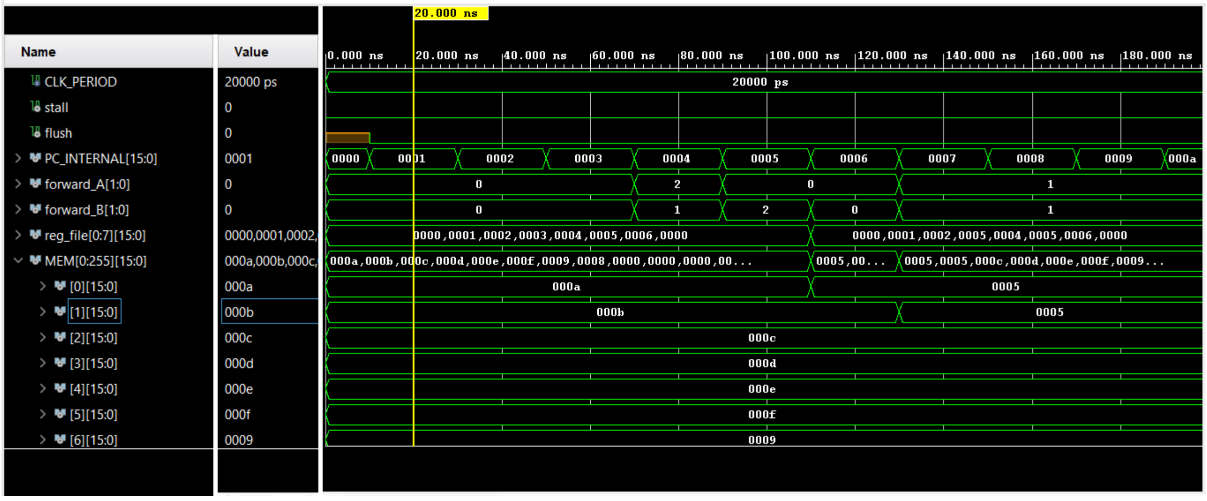


Fig. 13: Waveform for the testing of MEM forwarding

Without forwarding, the results would not be the same. In case forwarding would not be present, the first instruction would not have time to complete for the updated result to be written in memory, as such the original value of the register $3 (that would be 3) should be written. Instead, we can see that the result of the operation is instead written.

The last Data Hazard that must be tested is the Load Data Hazard. To do this, we shall make a program load some data from memory and proceed to use that data to do some operations. Below is the designed program and waveform:

B"000_000_000_000_0_111", -- NOOP

B"010_000_001_0000010", -- LW $1 $0 2

B"000_001_010_011_0_000", -- ADD $3 = $1 + $2

B"000_011_100_101_0_001", -- SUB $5 = $3 - $4

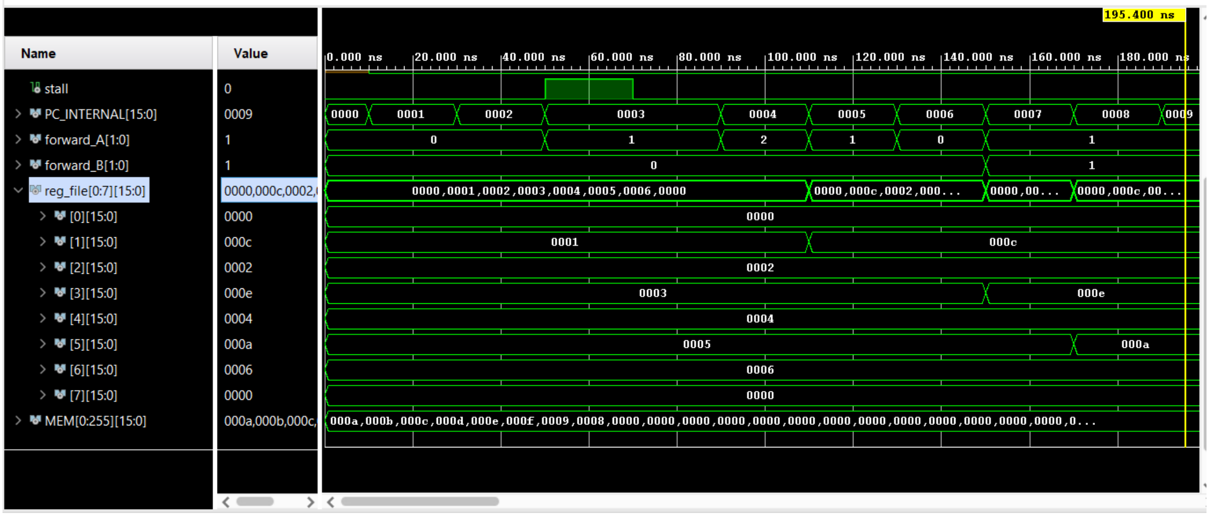


Fig. 14: Waveform for the testing of Load Data Hazards

In this case, if no hazard would be detected by the Hazard Detection Unit and no stall would be inserted, then the loading into register $1 would not happen in time for the updated value to be used for the addition and subtraction operations that follow. As such, the original value will be used, and the resulting operation will have the wrong result.

For testing the Control Hazards and using the designed Branch History Table, we must design a short program that uses the branch instructions. One such short program is to continuously add 1 to a register until it reaches a certain value. The chosen value is F (15), and it is retrieved from memory at the start of the program. When it finishes, we will save it to memory. The program is presented below along with the corresponding waveform:

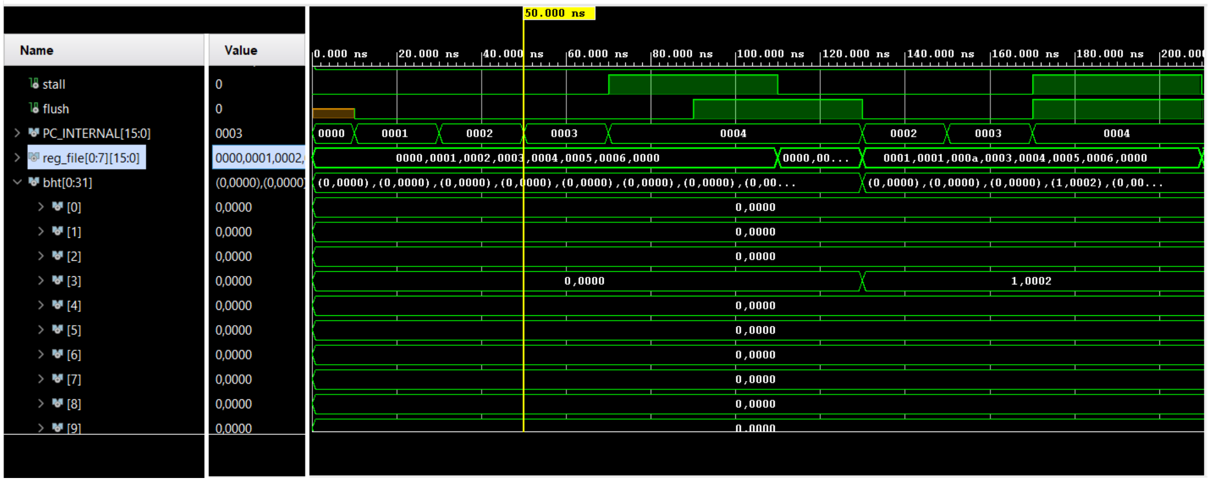
B"000_000_000_000_0_111", -- NOOP

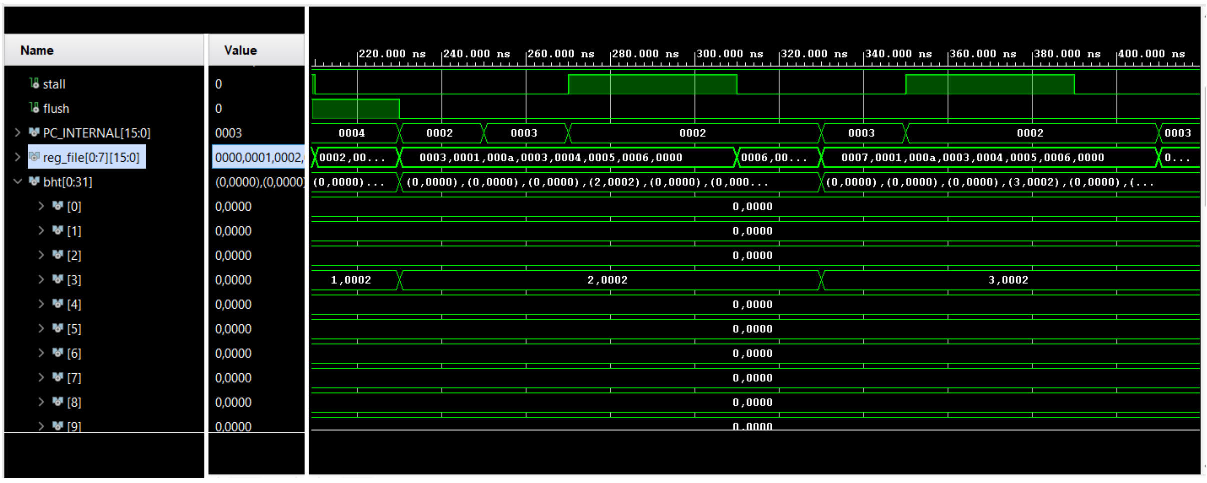
B"010_000_010_0000000", -- LW $2 $0 0

B"000_000_001_000_0_000", -- ADD $0 = $0 + $1

B"100_000_010_1111110", -- BNE $0 $1 -1

B"011_010_000_0000000", -- SW $2 $0 0





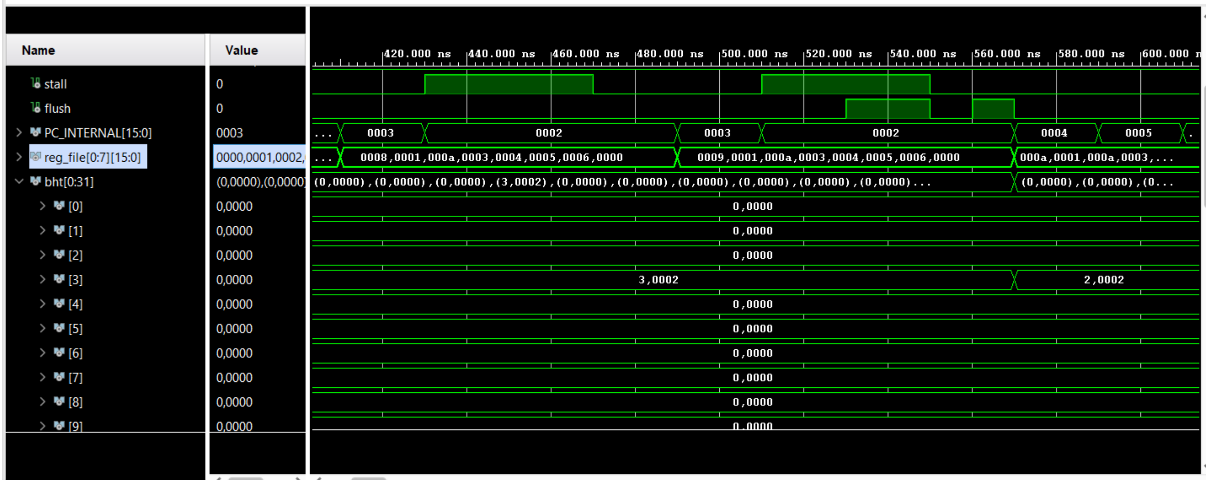


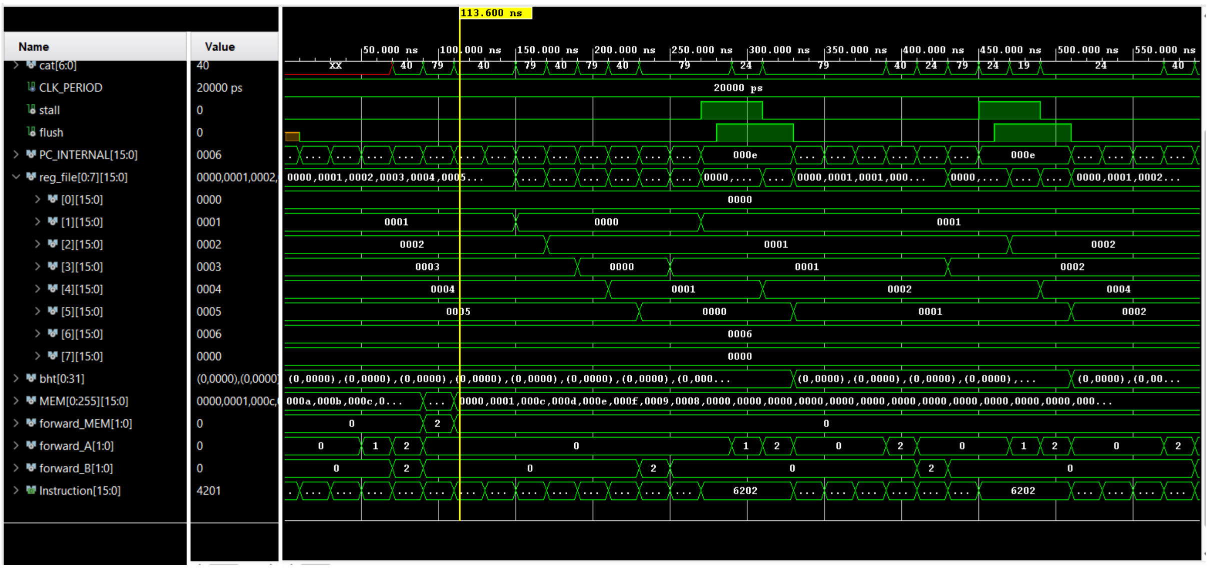
Fig. 15, 16, 17: Waveform for the testing of Control Hazards

With no forwarding, the result we would expect would be different. The major difference is the memory, that would get changed every time the loop was executed, due to the store instruction. In addition, the loop would always execute as it should, from start to finish (including the SW instruction)

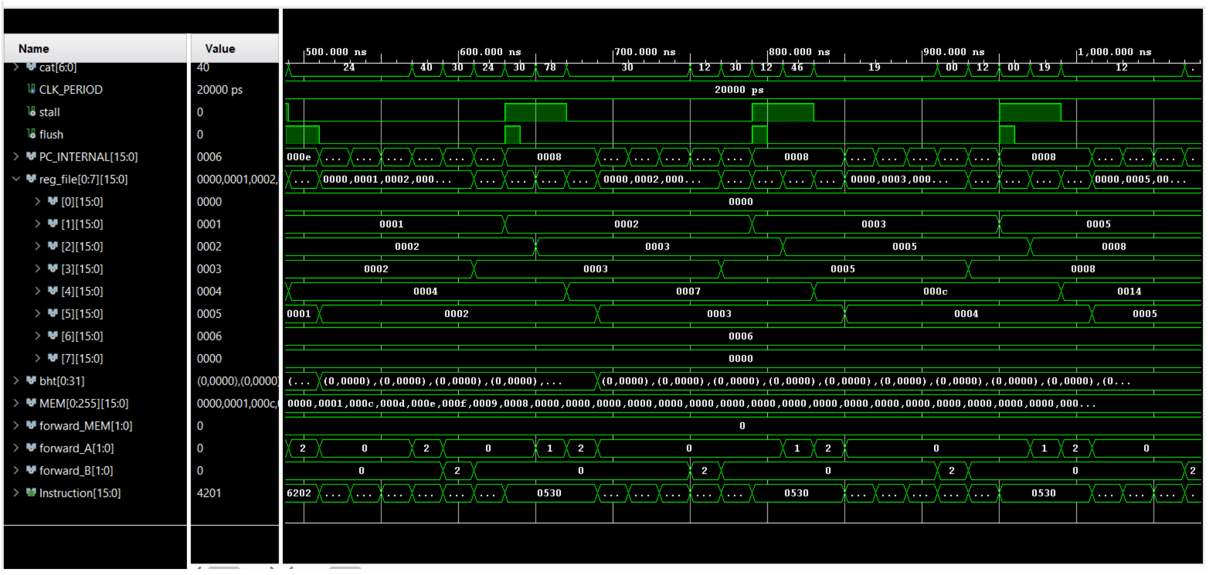
With forwarding, however, we notice a big difference from what would be expected without forwarding. The difference is the way the memory remains unchanged, due to the flushing of the pipe. Moreover, the sequence of execution is different. Noticeable is that, from the 3rd iteration of the loop, the last instruction does not execute anymore and at the end, it gets executed automatically. This is because of the successful implementation and use of the Branch History Table with Dynamic Branch Prediction, being able to adapt during the execution of the program.

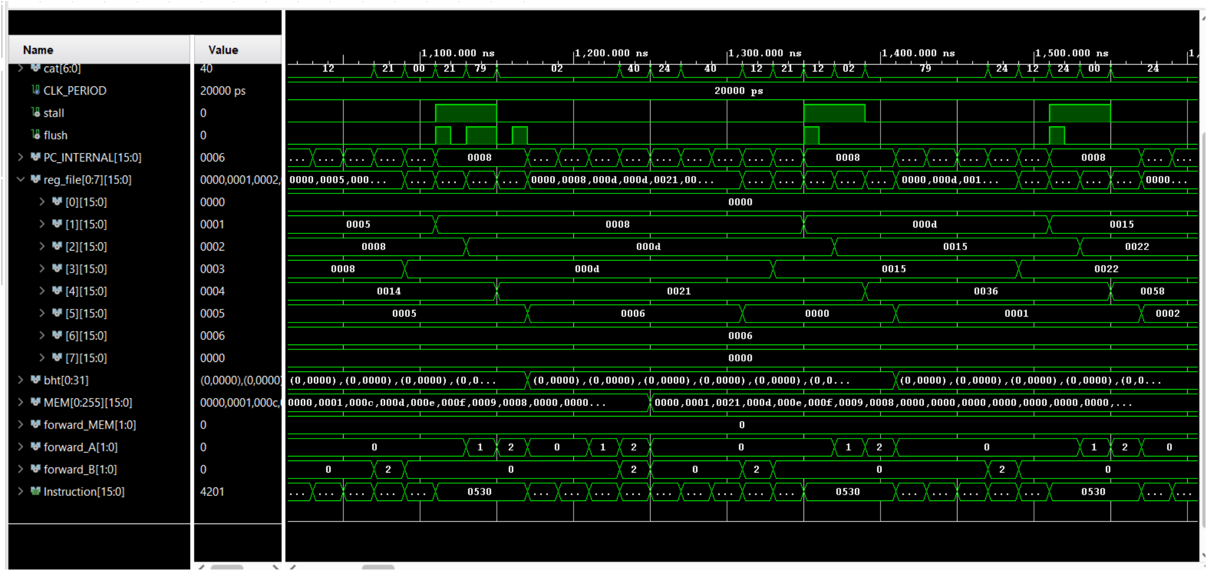
As a final test, the original program was adapted to encounter multiple hazards and still achieve the correct functionality. As such, besides the infinite computation of the Fibonacci Sequence, a register to hold the sum of the numbers was assigned and, in addition, an iterator too. Its purpose is to make the program flow leave the computation loop every 6 computed numbers and then store in memory the value of the sum up until that moment. The program along with the waveform are inserted below:

```
B"000_000_000_000_0_111", -- NOOP (0)
B"011_000_001_0000001", -- SW $1 $0 1 -- store 1 in MEM (1)
B"011_000_000_0000000", -- SW $0 $0 0 -- store 0 in MEM (2)
B"010_000_001_0000000", -- LW $1 $0 0 -- 1st Fibonacci number (3)
B"010_000_010_0000001", -- LW $2 $0 1 -- 2nd Fibonacci number (4)
B"010_000_011_0000000", -- LW $3 $0 0 -- 3rd Fibonacci number (5)
B"010_000_100_0000001", -- LW $4 $0 1 -- sum (starting from 1) (6)
B"010_000_101_0000000", -- LW $5 $0 0 -- iterator - jump here after loop (7)
-- loop start
B"000_001_010_011_0_000", -- ADD $3 = $1 + $2 (8)
B"000_000_010_001_0_000", -- ADD $1 = $0 + $2 (9)
B"000_000_011_010_0_000", -- ADD $2 = $0 + $3 (10)
B"000_100_011_100_0_000", -- ADD $4 = $4 + $3 (11)
B"001_101_101_0000001", -- ADDI $5 = $5 + 1 (12)
B"100_101_110_1111010", -- BNE $5 $6(=6) -6 (13)
-- end loop
B"011_000_100_0000010", -- SW $4 $0 2 - store the sum here after loop (14)
B"111_0000000000111", -- J 7 - repeat the loop
```



Name	Value
cat[6:0]	40
CLK_PERIOD	20000 ps
stall	0
flush	0
PC_INTERNAL[15:0]	0006
reg_file[0:7][15:0]	0000,0001,0002,
[0][15:0]	0000
[1][15:0]	0001
[2][15:0]	0002
[3][15:0]	0003
[4][15:0]	0004
[5][15:0]	0005
[6][15:0]	0006
[7][15:0]	0000
bht[0:31]	(0,0000),(0,0000)
MEM[0:255][15:0]	0000,0001,000c,
forward_MEM[1:0]	0
forward_A[1:0]	0
forward_B[1:0]	0
Instruction[15:0]	4201





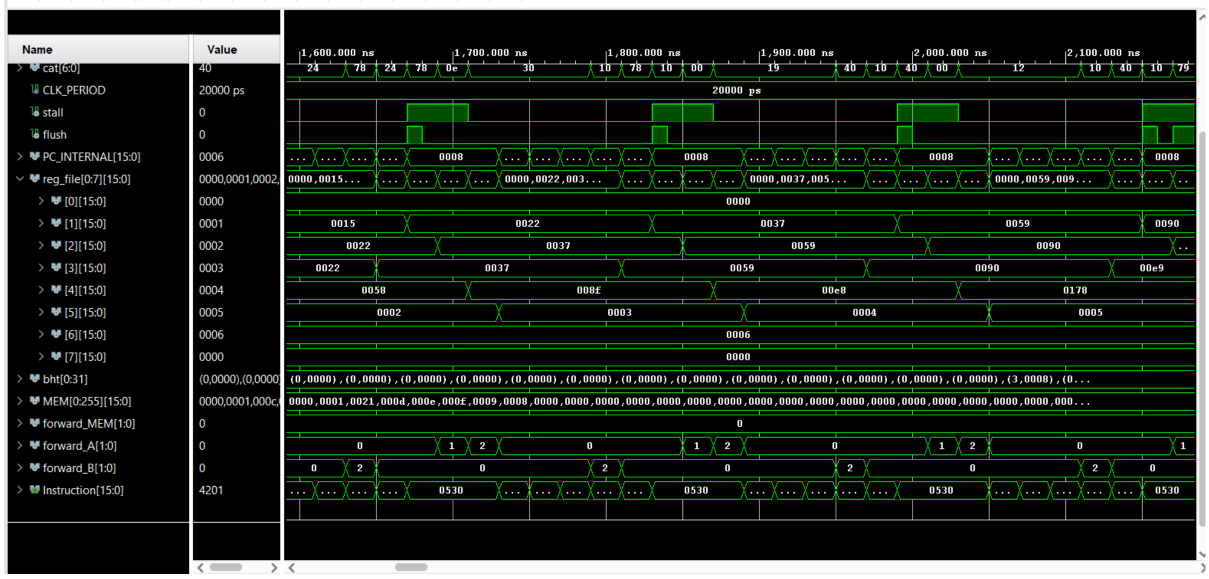


Fig. 18, 19, 20, 21: Waveform for the testing the complete Final Design

With no forwarding, this program would be a complete mess. There are several hazards located here: a data hazard at the initial second store and first load instructions, data hazard in the internal loop of the program and a control hazard determined by the branch instruction. In addition, the memory would be written into every traversal of the loop.

With hazard detection and branch prediction this is not the case, as the program is executed smoothly and correctly every time. On top of that, with the help of dynamic branch predictions, rather than static, the loop's execution performance is increased after the first traversal, again proving the superiority of dynamic branch prediction in a MIPS.

7. Conclusions

The proposed Hazard Detection and Branch Prediction project was quite a difficult one. The needed time to analyze every hazardous scenario and how a branch could be predicted, the latter being something I did not encounter before, was by far a lot more than the predicted amount. The difficulty of this project kept rising when the scenarios and needed considerations kept piling up and bugs started appearing in the implementation.

However, this did not determine me to give up! Taking everything slowly and methodically, asking questions and finding the right sources of information proved to be the greatest assets that brought this project to a final, successfully functioning state.

This was a great opportunity to test my VHDL programming skills and knowledge about the general functionalities of the MIPS along with the discovery of new ones.

8. References:

[1] Chat GPT [Online] - https://chat.openai.com/

[2] Wikipedia [Online] - https://en.wikipedia.org/wiki/Hazard_(computer_architecture)

[3] Mihai Negru, Computer Architecture https://users.utcluj.ro/~negrum/index.php/computer-architecture/

[4] Radu-Augustin Vele, "Hazard Detection and Avoidance Unit" [Online] - https://moodle.cs.utcluj.ro/pluginfile.php/191082/mod_folder/content/0/Radu-Augustin_Vele_MIPS_Hazards_Documentation%20%281%29.pdf

[5] Saffa Omran, Hadeel Shakir Mahmood, "VHDL prototyping of a 5-stages pipelined RISC processor for educational purposes" [Online] - https://www.researchgate.net/publication/283778560_VHDL_prototyping_of_a_5-stages_pipelined_RISC_processor_for_educational_purposes

[6] Analyze and Improve srv32 [Online] - https://hackmd.io/@peishan/HkJrtKpoY

[7] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 3 rd edition, ed. Morgan–Kaufmann, 2005. - https://ia600201.us.archive.org/24/items/ComputerOrganizationAndDesign3rdEdition/-computer%20organization%20and%20design%203rd%20edition.pdf