Hazard Detection and Avoidance Unit

Student: Radu-Augustin Vele

Structure of Computer Systems Project

Technical University of Cluj-Napoca

January 15, 2023

Contents

Introduction

1.1 Context

The aim of this project is to provide a Hazard Detection and Avoidance unit for the MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) Pipeline architecture. Taking advantage of pipelining in the design of a CPU brings multiple benefits, but comes with the cost of possibly occurring hazard conditions. This decreases the reliability of the computer system that depends on that CPU.

A hazard detection and avoidance unit improves the design by identifying and solving some more frequent hazard conditions. The resulting CPU design can be used for building hazard-proof, personalized computer systems for specific use cases.

1.2 Objectives

The unit will be designed in VHDL and included in a Xilinx Vivado Project containing an already-designed MIPS Pipeline. A testbench is also provided for simulation purposes. With some additional components such as a debouncer and a seven-segment display unit, the design could easily be programmed on an FPGA board.

Besides the general execution of instructions, the final MIPS will detect and resolve three types of hazard conditions:

List 1.2.1: The types of hazards detected and solved

- Read-After-Write (data hazard): dependencies of instructions on data that is not yet available.
- Load data hazard (data hazard): a special case where stalls (bubbles) need to be inserted.
- Control hazards (branch hazards): situations where the flow of the program is not the intended one as several instructions that would not be executed after a branch are already in the pipeline

Bibliographic Research

2.1 What is a hazard?

The MIPS architecture has been designed to include 5 pipeline stages: Instruction Fetch (**IF**), Instruction Decode (**ID**), Execution (**EX**), Memory (**MEM**) and Write Back (**WB**). The hazards occur when the instruction that is next in line can not be executed properly. They be divided into three categories [1]: **structural** hazards, **data** hazards, and **control** hazards.

In this project, the focus is on the last two types of hazards as the structural ones are a concern only when more instructions need to access the same resource simultaneously. It is not the case for the MIPS processor as the only such resource would be the memory. As Data and Instruction Memories are separated, structural hazards can not occur.

Moreover, only hardware solutions are explored as there is also the option to use software approaches for solving them (usually involving the compiler, e.g. reordering instructions).

2.2 Solving data hazards

Data hazards occur when the instruction that is about to be executed needs some data that is not ready yet. This could be easily but inefficiently solved by inserting stalls (instructions with no effect) until all data is ready. A better solution is designing a **forwarding unit** [1] that fast-forwards the data to where is needed immediately after it is ready. This adds the need for more control signals that allow data to be forwarded only when necessary.

In the case of the chosen architecture, there is a special case that involves an instruction that needs data loaded from the memory in the previous instruction. In this case, one stall **needs** to be inserted. Thus, a **hazard detection unit** [1] is needed. It has to be able to detect the special condition and insert a NOP instruction.

2.3 Solutions for control hazards

Control hazards generally involve branch instruction. The outcome of the branch is only known in a later stage in the pipeline (Execution or Memory stage). Therefore, at each clock cycle between the start of the branch and the result of its test a new instruction is introduced in the pipeline. If the branch condition is true those instructions must not have an effect to have a correct program. Therefore they need to be **flushed** [1].

The chosen solution involves **Dynamic Branch Prediction** [1], meaning that we decide to execute or not the instruction following a branch during the program execution. The decision is based on whether the previous branches were taken or not, this information being saved in a table-like data structure.

Analysis

3.1 Project Proposal

The final MIPS Pipeline with Hazard Detection and Resolution will encompass the features described below.

List 3.1.1: Features of the final MIPS

1. Instruction Set Architecture including 14 Instructions 4 (feature inherited from the simple pipeline implementation)
2. Data Hazard - Read after Write - detection and solving
3. Data Hazard - Load - detection and solving
4. Control Hazard detection and solving using Dynamic Branch Prediction
5. One or more test programs that challenge the hazard management

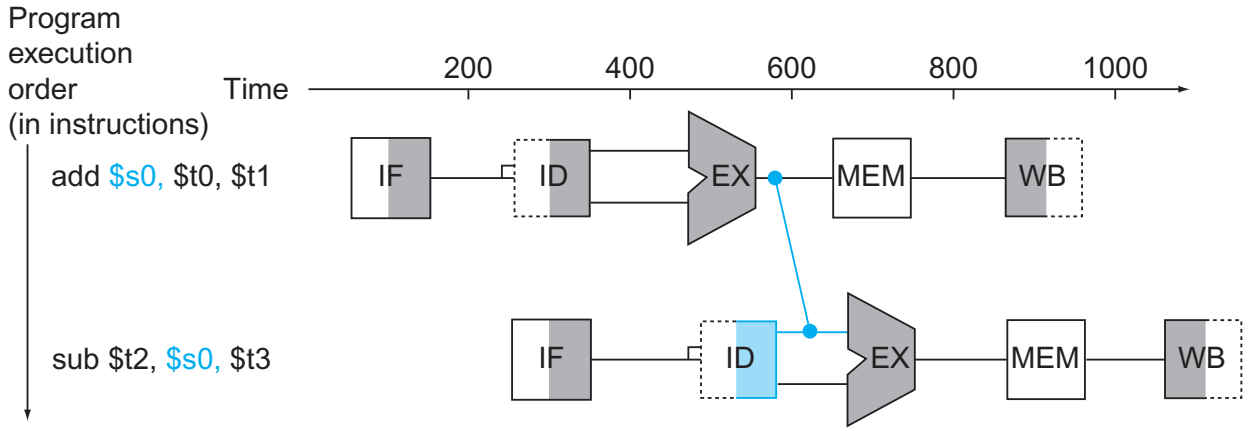
3.2 Project Analysis

3.2.1 Hardware solutions for data hazards

First, the RAW (Read-After-Write) hazard is solved using the **forwarding unit**. The desired behavior is similar to 3.2.1. It can be seen that the result of the addition is immediately available in the EX stage of the add instruction. Consequently, the forwarding unit should transport it to the execution unit of the following instruction.

There are multiple scenarios regarding where the data is available and where it is needed. Those are described below.

Figure 3.2.1: Data forwarding for RAW [1]



List 3.2.1: Data hazard scenarios

1. The first instruction (I1) (I-type or R-type) writes to a register (R_{dest}) and the second instruction (I2) attempts to perform an operation using R_{dest} as an operand.

   This way, the value that is about to be written to R_{dest} is available in EX/MEM pipeline register when I2 is in the Execution stage.

2. The first instruction (I1) (I-type or R-type) writes to a register (R_{dest}), another instruction is issued, and finally, the third instruction (I3) attempts to perform an operation using R_{dest}.

   In this case, the needed value is available in the MEM/WB pipeline register when I3 is in the execution stage.

3. The first instruction writes to R_{dest} and the second one (I2) is a **store instruction** that inserts the value of R_{dest} into the memory.

   In this case, the needed value can be forwarded from the MEM/WB pipeline register to the MEM stage of the store instruction.

4. The first instruction writes to R_{dest}, the second one is a "safe" instruction, and the third one is a store instruction that attempts to store the value at R_{dest} in the Register file.

5. The first instruction is a load and the next one is a store that inserts the previously loaded value at a different address.

   In this case, the data needed to be stored is available in the MEM/WB stage when it is needed (in MEM stage of the store instruction). However, this situation will be solved using the Hazard Detection unit that is designed specifically for the times the first instruction is a load.

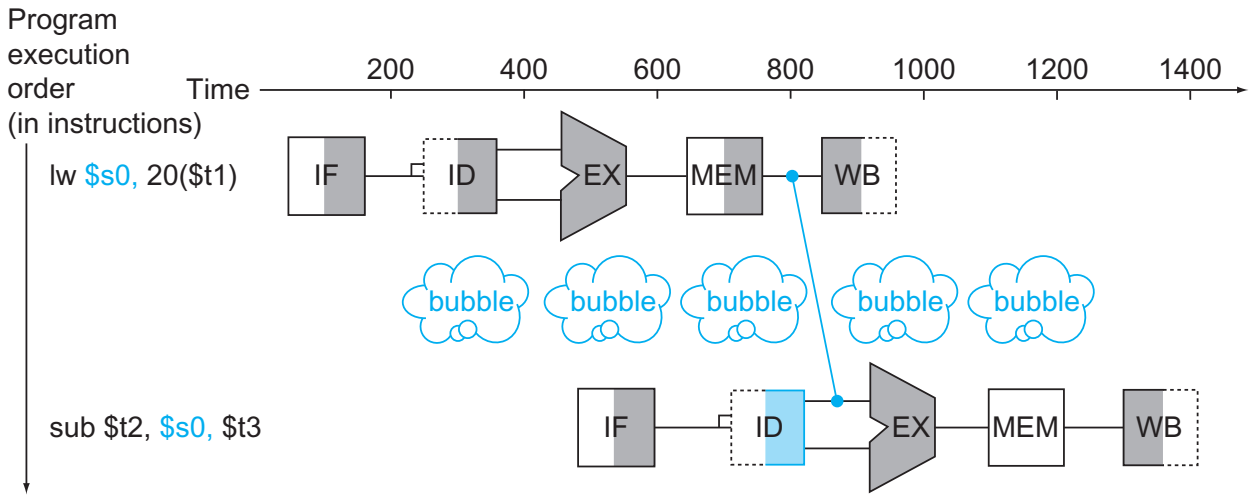
Remark 3.2.1: Forwarding priority

An instruction that uses a register previously written needs to get the value most recently written to that register.

The mechanisms for detecting these situations and generating the control signals are described in the Implementation section.

The Load Data hazard (LDH) appears in the situation depicted in 3.2.1. The bubble or stall represents that the instruction depending on the load (and all the following ones) are delayed for one clock cycle. The state of the register file and the memory must not change after the stall.

Figure 3.2.2: Inserting a stall and forwarding for Load Data Hazards [1]



3.2.2 Control Hazards and Dynamic Branch Prediction

For the **control hazards**, we need a hardware mechanism for flushing the instructions after a branch whose correct behavior was not predicted accordingly.

To minimize the number of instructions that need to be flushed, the branch detection logic is moved to the Instruction Decode stage. Therefore, only one instruction is started before the branch decision is known. So, the flushing mechanism can focus on deleting the data transmitted from the Instruction Fetch Phase to the Instruction Decode Phase through the pipeline register (IF/ID).

Moving the branch to the ID stage adds new Data Hazards. (contents of two registers need to be available for checking for equality and inequality)

3.2.2.1 Data hazards for Branch Prediction in ID Stage

The dependency between the branch (ID) and the previous instruction (EX) can not be resolved without a stall, so the Hazard detection unit needs to be extended to detect this case.

One more case is when the **previous instruction is a load** and the destination of the load is a register used in branch detection. In this case, the load data hazard is detected and a **first stall** is inserted. However, the forwarding still can not be performed, **one more stall needs to be inserted**.

After the two stalls, the second forwarding condition described in the next section will be met and loaded data will be forwarded.

A dynamic branch prediction method will be implemented to determine the next instruction to be executed.

Dynamic Branch Prediction uses data obtained during the program execution to predict which instruction to execute next after the current one.

3.2.2.2 How it works

The chosen approach features a **branch history table**, a combination between a branch target buffer and a 2-bit predictor 3.2.2.2 table. The table accumulates information to predict the next instruction of the program counter. On one hand it keeps a counter that is incremented when a branch is taken and decremented when it is not taken, and on the other hand, it keeps the address that was input into the program counter the last time when the branch was taken.

There are two stages in the prediction mechanism: **Read** the BHT and **generate** a prediction, and **Check** the prediction and **update** the BHT. They are described below.

A. Read BHT and Predict

In this step, the next PC address is chosen using the branch history table. If a branch is predicted, the target address at the same memory address as the predictor is sent to the memory, otherwise, the next PC address is `PC + 1` (usual behavior for non-branch instructions). For the next step (ID) we will send the PC bits that were used to index the BHT to the IF/ID pipeline register, alongside the prediction.

B. Check prediction and update BHT

This stage is reached one clock cycle after the prediction has been made. There are two possible outcomes of the prediction checking.

If it is found that the prediction is different from the actual result of the branch, a **flush** is generated and the next fetched instruction is the one computed according to the branch. Otherwise, the flow of the program is not changed (the predicted instruction goes on.

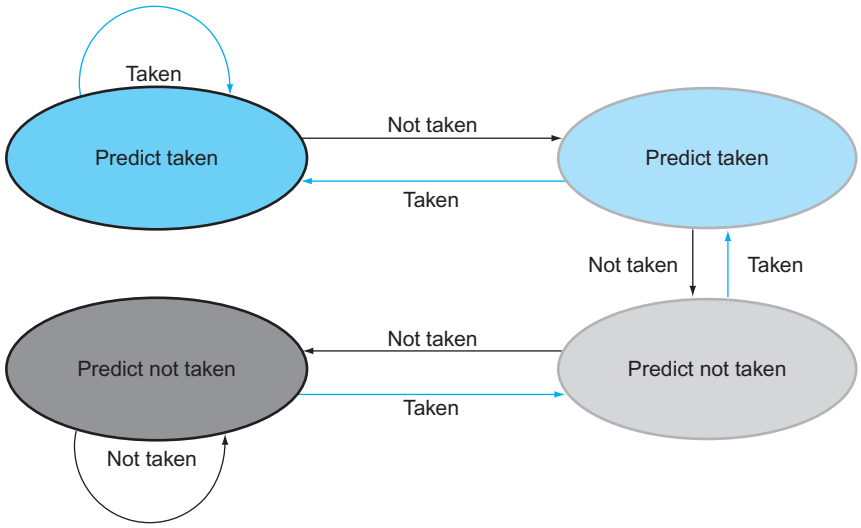
Now that when the branch outcome is known, the BHT needs to be enriched with this information.

1. if the branch was taken the BHT predictor at the address received through the IF/ID register is **incremented** and the computed branch target address is **updated to the one newly computed**.

2. if the branch was **not** taken, the BHT predictor at the address received through the IF/ID register is **decremented** (if 0 is reached, it doesn't go below) and the target address is **not** overwritten.

As branch decisions tend to repeat (see loops), this proves a lot better than static prediction (assume branch not taken). Based on the history, the next address inserted in the program counter can either be the branch target, or the next instruction.

Figure 3.2.3: Two-bit predictor [1]



Design

The design of the MIPS pipeline on 16 bits is not the object of this project. It is considered to be a fully functional processor (as shown in 4) supporting a fixed set of instructions (see table 4).

Remark 4.0.1: Notation

Note: the notations of the instructions are:

- `r_ins $dest, $op1, $op2` - for r-type instructions
- `i_ins $dest, $op1, imm` - for i-type instructions
- `branch $op1, $op2, imm` - for branches
- `jmp imm` - for jumps

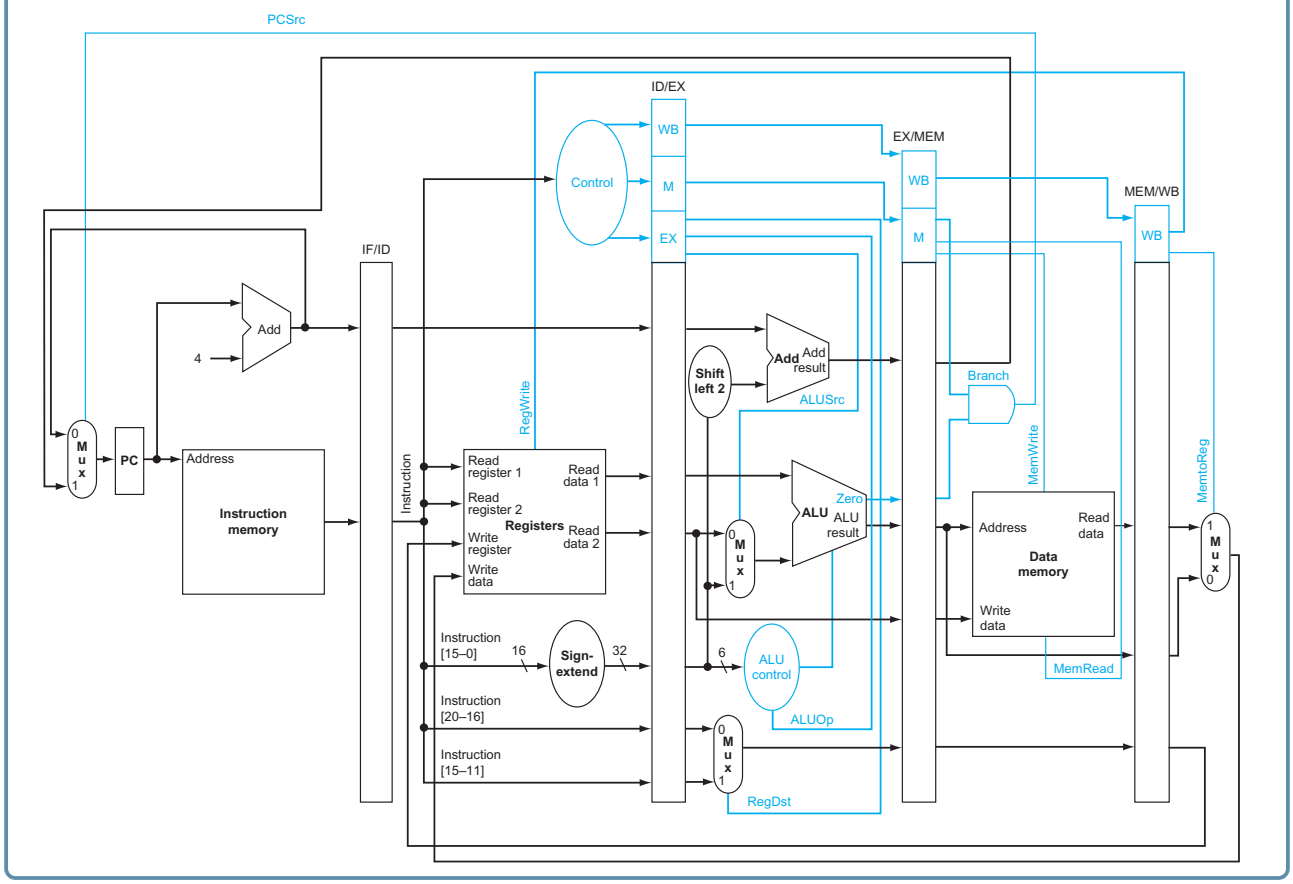
Table 4.0.1: Instructions supported by the processor

Instruction	Type	RTL Abstract
`add $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] + RF[rt]
`sub $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] - RF[rt]
`sll $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] $\ll$ RF[rt]
`srl $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] $\gg$ RF[rt]
`and $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] & RF[rt]
`or $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] || RF[rt]
`xor $rd, $rs, $rt`	R	RF[rd] $\leftarrow$ RF[rs] $\oplus$ RF[rt]
`addi $rt, $rs, imm`	I	RF[rd] $\leftarrow$ RF[rs] + S_{Ext}(imm)
`subi $rt, $rs, imm`	I	RF[rd] $\leftarrow$ RF[rs] - S_{Ext}(imm)
`lw $rt, $rs, imm`	I	RF[rt] $\leftarrow$ Mem[RF[rs] + S_{Ext}(imm)]
`sw $rt, $rs, imm`	I	Mem[RF[rs] + S_{Ext}(imm)] $\leftarrow$ RF[rt]

`beq $rt, $rs, imm`	I	**if** RF[rs] == RF[rt] **then**
		PC $\leftarrow$ PC + 1
		else
		PC $\leftarrow$ PC + 1 + S_{Ext}(imm)

`bneq $rt, $rs, imm`	I	**if** RF[rs] $\neq$ $RF[rt]$**then** PC $\leftarrow$ PC + 1 **else** PC $\leftarrow$ PC + 1 + S_{Ext}(imm)
`jmp imm`	J	PC $\leftarrow$ Ext(imm)

Figure 4.0.1: MIPS Pipeline with no hazard detection and resolution [1]



4.1 Forwarding Unit Design

The main goal of this unit is to generate control signals that determine data forwarding, based on some tested conditions. The decision of whether to select the forwarded data in the needed context is taken by the forwarding unit.
The different cases that were presented in the Project Analysis section are solved as it is covered next.

Remark 4.1.1: Notations for signals

- Pipeline_Stage/Next_Pipeline_Stage refers to the pipeline registers between the two stages
- Register Rs refers to the address of the first operand of the R and I type instructions
- Register Rt refers to the address of the second operand of the R-type instruction and the address of the destination of the I-type instruction.
- RegWrite signal is the control signal generated by the Control Unit that determines writing in the register file.

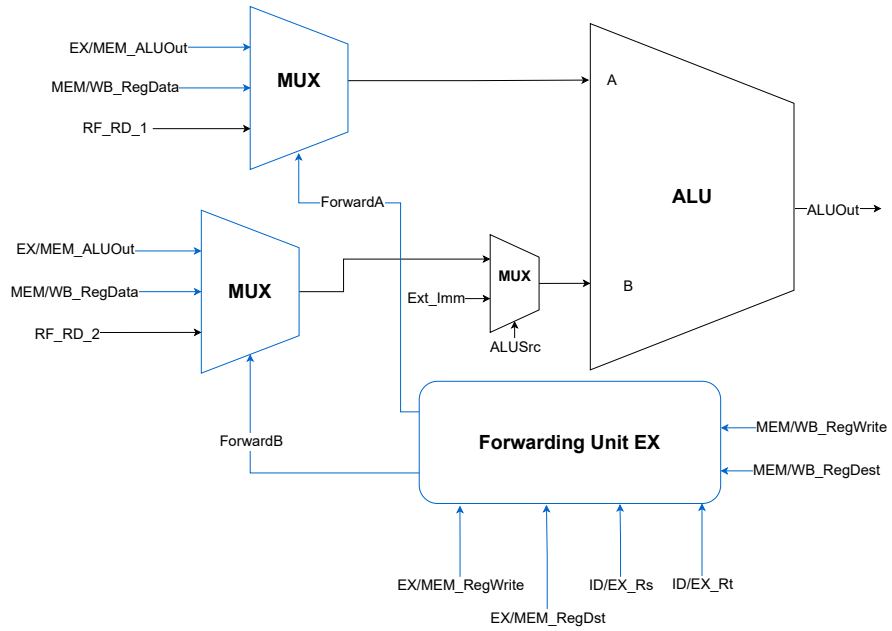
4.1.1 Forwarding to the EX stage

To solve the first two hazards described in 3.2.1 a forwarding unit needs to be added in the Executions stage.

The forwarded data will be taken from EX/MEM or MEM/WB registers and sent to the ALU input corresponding to Rs (Input A) or Rd (Input A). Two additional Multiplexers are added to ensure the correctness of the forwarding, and the forwarding unit is placed in the EX stage.
The selections of these multiplexers are the output signals of the Forwarding unit.

To put it all together, the additional hardware changes so far (in blue) are shown in 4.1.1.

Figure 4.1.1: Hardware Modifications for forwarding in the Execution Stage



4.1.2 Forwarding to the MEM stage

This situation imposes forwarding data to the data entry of the Data Memory, therefore, another forwarding unit (Forwarding Unit MEM) placed in the MEM stage is necessary. Data is needed from both the previous instruction and the one before it.

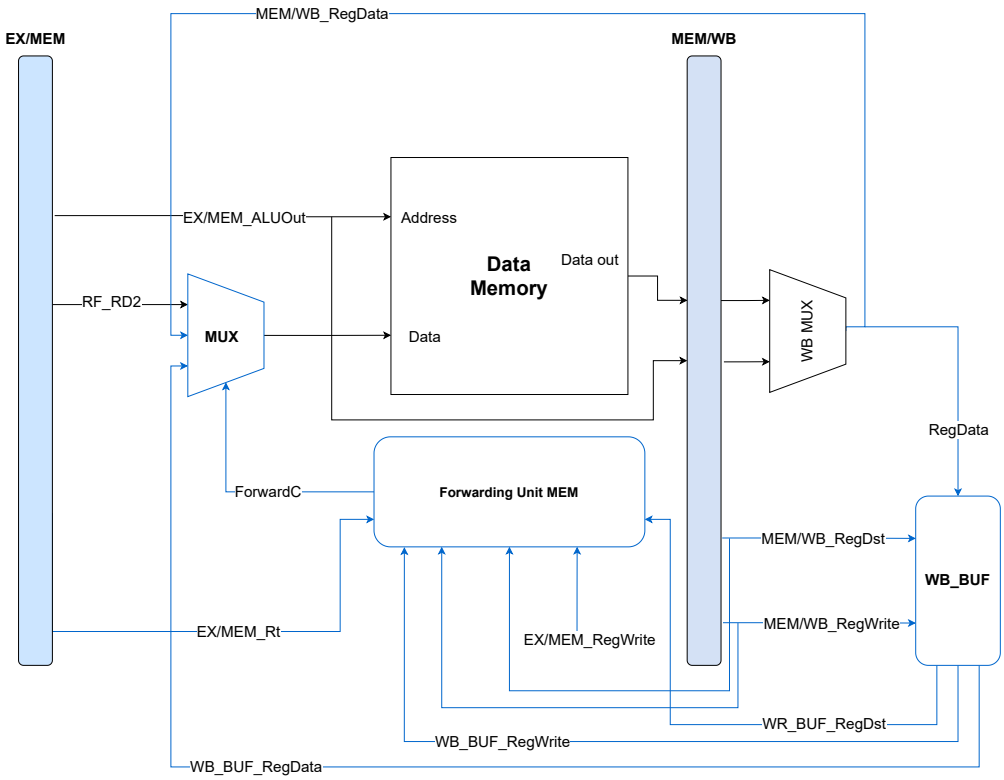
The second case is a special one as the information regarding the address of the register file where the instruction before the previous one wrote and the control signal that enabled the writing have already passed through the whole pipeline and are no longer available.

The solution is to buffer this information for one more clock cycle to be able to use the forwarding unit in the memory stage. This buffer register is called **WB_BUF**.

In addition, the WB_BUF_RegData (data that was written to the Register File at the WB_BUF_RegDst address) needs to be buffered so that it can be forwarded to the data input of the Data Memory.

The hardware modifications and additions (in blue) are shown in 4.1.2

Figure 4.1.2: Hardware Modifications for forwarding in the Memory Stage



4.2 Hazard detection Unit Design

This component detects **load data hazards**. It needs to check the dependency between the currently executing instruction (that is already in the ID stage) and the previous one that in this case needs to be a load for the bubble to be inserted.

The mechanism of stalling the pipeline is achieved by making sure that the register file and the data memory are not modified. Therefore, the control signals generated for instruction currently in the ID stage will be pulled to 0 by the Hazard Detection Unit (with the aid of a multiplexer).

Moreover, the state of the instruction in the ID stage needs to be frozen, so the IF/ID pipeline register needs to maintain its state. This is achieved by adding an enable signal that would prevent the instruction in the IF stage to overwrite the IF/ID register.

Finally, the instruction currently in the IF stage needs to be issued again after a clock cycle so the program counter will be also stopped by using an enable signal.

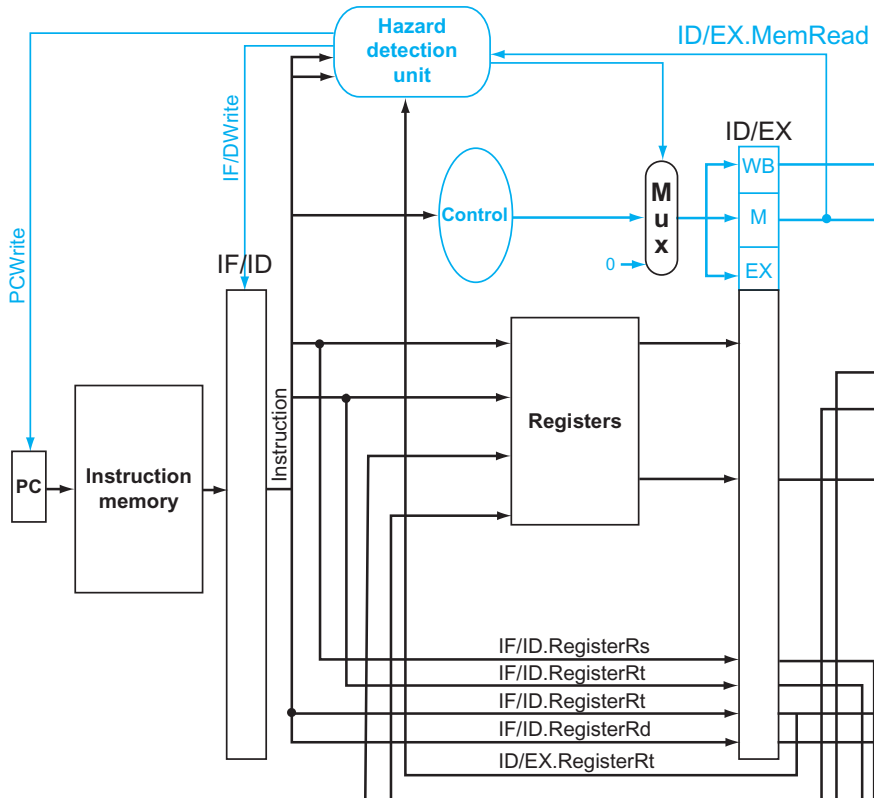
As mentioned in the Project analysis, the load data hazard resolution solves the situation where a store attempts to insert into memory what has been previously loaded from another memory location. In this situation, a stall is inserted after the load. Afterward, the situation is solved by forwarding the unit (the case where data is forwarded from WB_BUF to the MEM stage).

The hardware modifications are shown in 4.2.

4.3 Branch Prediction Unit Design

This unit involves three design steps: moving the branch detection logic and address computation to the ID stage to minimize the number of instructions executed before knowing the

Figure 4.2.1: The hazard detection unit and the added signals [1].



branch decision is known, flushing the instructions that were not supposed to be executed, and predicting the branch dynamically according to a history table.

4.3.1 Minimize the number of instructions executed after the branch

The minimization of the number of instructions executed after the branch is achieved by generating the branch signal in the ID stage and performing the comparison between the data read from the register file (as in this architecture we will only support branch on equal and branch on not equal).

As mentioned in the project analysis, new data dependencies are added and need to be solved in the ID stage. That is achieved by **extending the hazard detection unit** and adding a new **forwarding unit**.

The detection unit checks for the hazards condition and the forwarding unit provides the correct data (either from EX/MEM of MEM/WB pipeline registers) to the Equality Detector with the aid of two multiplexers.

4.3.2 Mechanism for flushing

For this case, the Flush signal will be generated and it will empty up the IF/ID register.

All of the hardware modifications necessary for moving the branch decision to the ID stage and flushing are shown in 4.3.2. For readability, the components and signals that don't influence the branch instruction processing have been omitted.

Figure 4.3.1: Modifications to the ID stage after moving the branch decision here

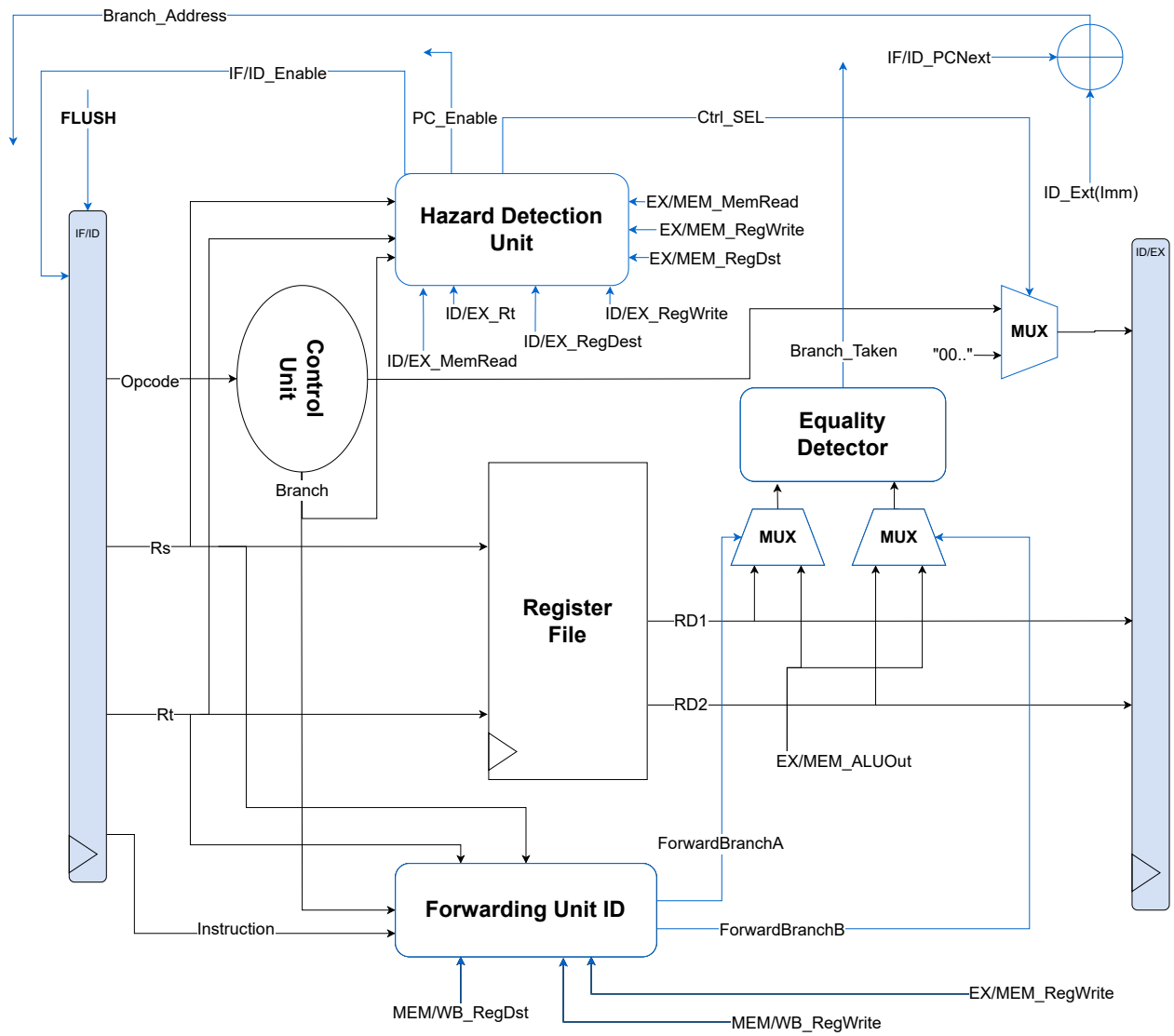
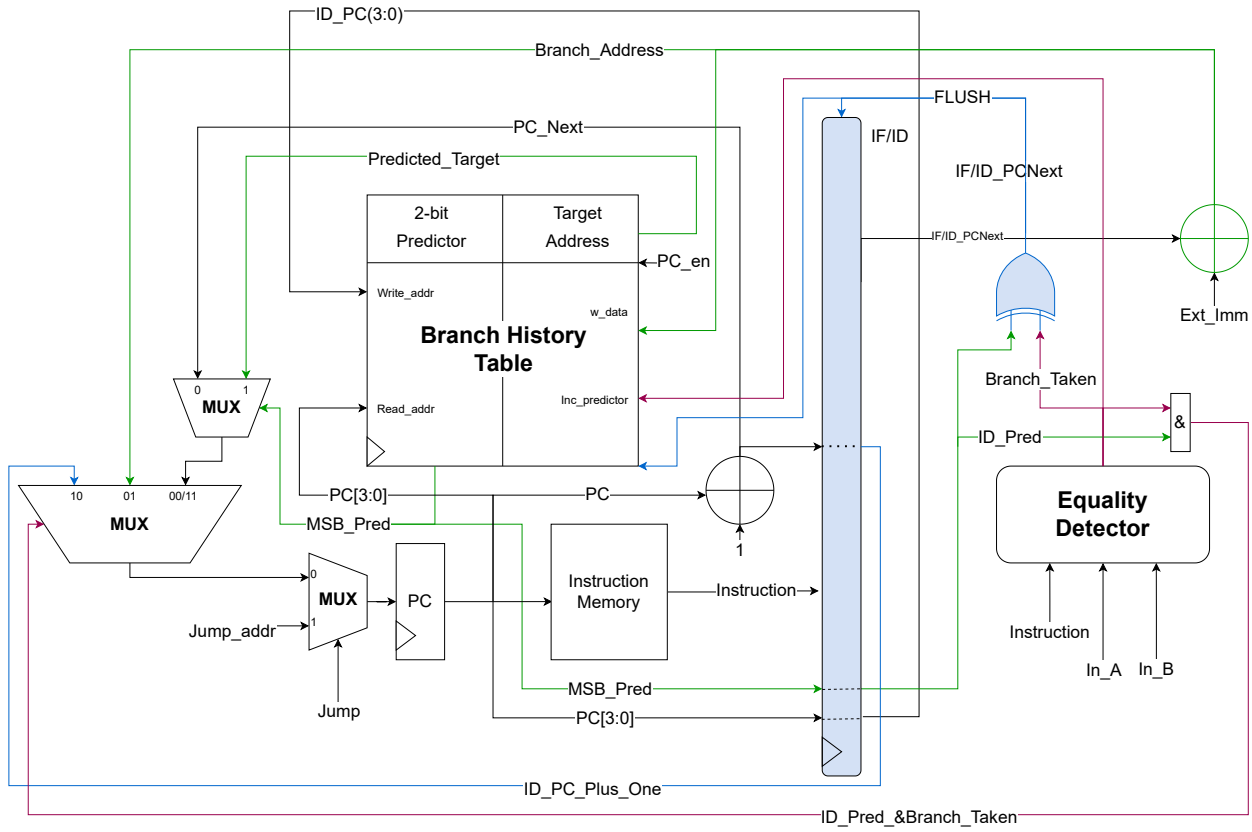


Figure 4.3.2: Hardware additions for the dynamic branch prediction



4.3.3 Dynamic branch prediction

Designing the circuitry that accomplishes dynamic branch prediction involves two stages: read from BHT (Branch history table) and predict (happening in the IF stage), and check prediction and update BHT (that happens in the ID stage after the branch is checked and the branch address is computed).

The structure of the table, shown in 4.3.3 as well is similar to the one of a Branch Target Buffer. It is a memory addressed by the lower 4 bits of the program counter address. In a memory entry, there is a 2-bit predictor and the target address.

The design of the dynamic branch prediction (excluding the MIPS components that do not concern this stage) is depicted in 4.3.3

Implementation

The implementation stage starts with adjusting and testing the MIPS pipeline implementation to make sure that it works as expected. Then, the individual components (forwarding units, hazard detection unit, etc.) are designed through simple processes that generally check conditions and assign selections for multiplexers.

5.1 Changes to the MIPS pipeline implementation

The MIPS source code written in **VHDL** is divided into 5 main modules, each one corresponding to a pipeline stage. In addition, the top-level module creates instances of these other modules alongside pipeline registers.
However, a few changes need to be made before adding the hazard-handling logic.

5.1.1 Reading after writing in the register file

This modification allows data to be available when, for instance, instruction I1 in the ID stage depends on instruction I2 in the WB stage. The dependency is detected on the falling edge and data is assigned to the outputs right from the write_data signal.

Listing 5.1.1: Read after Write in reg_file.vhd

```
if RISING_EDGE(clk100MHz) then -- writing
   if reg_write = '1' then
       curr_content(to_integer(UNSIGNED(write_address))) <= write_data; --
          synchronous write
   end if;
end if;

if FALLING_EDGE(clk100MHz) then -- reading
   read_data1 <= curr_content(to_integer(UNSIGNED(read_address1)));
   read_data2 <= curr_content(to_integer(UNSIGNED(read_address2)));

   if read_address1 = write_address and reg_write = '1' then
       read_data1 <= write_data;
   end if;

   if read_address2 = write_address and reg_write = '1' then
       read_data2 <= write_data;
   end if;
end if;
```

5.1.2 Adding WB_BUF register

For the case shown in 4.1.2 we find the need to buffer some of the output signals of the WB stage in an additional pipeline register.

5.1.3 Jump Handling in IF

The jump address and signal computation is moved to the IF stage to avoid introducing an extra instruction in the pipeline (and eventually flushing it).

Listing 5.1.2: Jump handling in IF_unit.vhd

```
jump_detection: jump_if <= '1' when (tmp_instruction(15 downto 13) = "111")
    else '0'; -- jump
zero_ext: jump_addr_if <= "000" & tmp_instruction(12 downto 0);
mux2: nxt_pc <= jump_addr_if when (jump_if = '1') else mux1_out;
```

5.2 Forwarding Unit

There are three forwarding units (placed in the IF_unit, ID_unit, and MEM_unit). Their implementation, inputs, and the meaning of the outputs is shown below.

5.2.1 ID Forwarding Unit

List 5.2.1: Inputs of the forwarding unit

- EX/MEM_RegWrite
- EX/MEM_RegDst
- ID_Branch
- ID_Rs
- ID_Rt

Listing 5.2.1: Conditions for forwarding to ID stage

```
if (ID_Branch = '1') and (EX_MEM_RegWrite = '1') and EX_MEM_RegDst = ID_Rs
   then
    ForwardA <= '1';
end if;

if (ID_Branch = '1') and (EX_MEM_RegWrite = '1') and EX_MEM_RegDst = ID_Rt
   then
    ForwardB <= '1';
end if;
```

The outputs **ForwardA** and **ForwardB** are the selections of the multiplexers placed before the Equality Detection Unit.

5.2.2 EX Forwarding Unit

List 5.2.2: EX Forwarding unit inputs

- ID_EX_Rs
- ID_EX_Rt
- EX_MEM_RegDest
- EX_MEM_RegWrite
- MEM_WB_RegWrite
- MEM_WB_RegDest

The conditions for determining data forwarding are described in the following listing.

Listing 5.2.2: Conditions for forwarding to EX

```
-- FWD from before the previous => less priority
if MEM_WB_RegWrite = '1' then
    if MEM_WB_RegDst = ID_EX_Rs then
        ForwardA <= "10";
    elsif MEM_WB_RegDst = ID_EX_Rt then
        ForwardB <= "10";
    end if;
end if;

-- FWD from previous => more priority
if EX_MEM_RegWrite = '1' then
    if EX_MEM_RegDst = ID_EX_Rs then
        ForwardA <= "01";
    elsif EX_MEM_RegDst = ID_EX_Rt then
        ForwardB <= "01";
    end if;
end if;
```

Table 5.2.1: The meaning of the bits assigned to ForwardA or ForwardB

Assigned value	Consequence
00	ALU input is **not** represented by forwarded data
01	ALU input is forwarded from **EX/MEM** pipeline register
10	ALU input is forwarded from **MEM/WB** pipeline register
11	-

5.2.3 MEM Forwarding Unit

List 5.2.3: MEM Forwarding unit inputs

- EX_MEM_Rt
- EX_MEM_MemWrite
- MEM_WB_RegWrite
- MEM_WB_RegDst
- WB_BUF_RegDst
- WB_BUF_RegWrite

Listing 5.2.3: Conditions for forwarding to MEM

```
-- FWD from WB_BUF less priority
if WB_BUF_RegWrite = '1' then
    if EX_MEM_Rt = WB_BUF_RegDst and EX_MEM_MemWrite = '1' then
        ForwardC <= "10";
    end if;
end if;

-- FWD from MEM_WB more priority
if MEM_WB_RegWrite = '1' then
    if EX_MEM_Rt = MEM_WB_RegDst and EX_MEM_MemWrite = '1' then
        ForwardC <= "01";
    end if;
end if;
```

Table 5.2.2: The meaning of the bits assigned to ForwardC

Assigned value	Consequence
00	Data Memory data input is **not** represented by forwarded data
01	Data Memory data input is forwarded from **MEM/WB** pipeline register
10	Data Memory data input is forwarded the from **WB_BUF** register
11	-

5.3 Hazard Detection Unit

The hazard detection unit is placed in the ID stage having as its main purpose the generation of stalls (for both load hazards and branch data dependencies).

The information necessary for the hazards to be detected is:

List 5.3.1: HDU Inputs

- ID_Rt
- ID_Rs
- ID_EX_Rt
- ID_EX_MemRead - HIGH only when the instruction is a load.
- ID_Branch - the signal generated by the control unit in ID.
- ID_EX_RegWrite
- ID_EX_RegDst
- ID_Rs - address of the first register to be read from RF.
- ID_Rt - address of the second register to be read from RF.
- EX/MEM_RegDst
- EX/MEM_MemRead

Listing 5.3.1: Hazard detection logic

```
-- load data hazard
if ID_EX_MemRead = '1' and (ID_Rs = ID_EX_Rt or ID_Rt = ID_EX_Rt) then
    IF_ID_WriteEn <= '0';
    Ctrl_Sel <= '0';
    PC_Enable <= '0';
end if;

-- branch dependencies
if(EX_MEM_RegWrite = '1' and Branch_instruction = '1') and (ID_Rs =
   EX_MEM_WrAddrChosen or ID_Rt = EX_MEM_WrAddrChosen) then
    IF_ID_WriteEn <= '0';
    Ctrl_Sel <= '0';
    PC_Enable <= '0';
end if;

if (ID_EX_RegWrite = '1' and Branch_instruction = '1') and (ID_Rs =
   EX_WrAddrChosen or ID_Rt = EX_WrAddrChosen) then
    IF_ID_WriteEn <= '0';
    Ctrl_Sel <= '0';
    PC_Enable <= '0';
end if;
```

Table 5.3.1: The outputs of the Hazard Detection Unit and their meaning

Output	on HIGH	on LOW
PC_Enable	No hazard, normal counting	LDH detected, freeze the counting
Ctrl_SEL	No hazard, send control signals	LDH detected, send zeros
IF/ID_Enable	No hazard, send IF data	LDH detected, preserve reg. state

5.4 Branch Prediction

5.4.1 Branch History Table

The branch history table is implemented as a custom RAM with a special meaning for the bits. The reading is done asynchronously, while writing is controlled by multiple enables and the master clock.

Listing 5.4.1: The structure of the BHT

```
type bht is array (0 to 15) of STD_LOGIC_VECTOR(17 downto 0);

--  ----------------------------
-- | Predictor | Target Address |
--  ----------------------------
-- 17          15               0

signal curr_bht: bht := (
    B"00_0000000000000000",
    B"00_0000000000000000",
    B"00_0000000000000000",
    others => B"00_0000000000000000"
);
```

Listing 5.4.2: Updating the BHT content

```
if flush = '1' and inc_predictor = '1' then
-- the previous prediction was different from the outcome (that is branch
   taken)
    curr_bht(to_integer(UNSIGNED(write_address)))(15 downto 0) <=
       update_data;
end if;

curr_bht(to_integer(UNSIGNED(write_address)))(17 downto 16) <=
   curr_predictor;
```

Testing & Validation

The Testing and Validation stage involves running simulations of the top-level mips for all hazards mentioned in the Project Proposal. The overall structure of a test includes the input program, the pipeline diagram with the dependencies, the expected results (with and without resolution), and the actual results.

The content of the register file and the data memory is presented in the listing below.

Listing 6.0.1: Content of the Register File and Data Memory

```
-- Register File Content
signal curr_content: reg_content := (
    x"0000", -- 0
    x"0000", -- 1
    x"0001", -- 2
    x"0000", -- 3
    x"0003",
    x"0002",
    x"0002",
    x"ABCD",
    others => x"1111");

-- Memory Content
signal curr_content: ram_content := (
    x"0002",
    x"0005",
    x"0007",
    x"1111",
    x"ABC8",
    x"0004",
    x"0006",
    x"0008",
    others => x"0000");
```

6.1 Testing general data hazards

6.1.1 Forwarding to EX stage

Listing 6.1.1: Test program for data hazards

```
B"000_001_010_011_0_001", -- 0: add $3 = $1 + $2
B"001_011_100_0001000",   -- 1: addi $4 = $3 + 8
B"000_100_011_101_0_001", -- 2: add $5 = $4 + $3
B"000_011_010_111_0_001", -- 2: add $7 = $3 + $2
others => x"0000"
```

Next, the pipeline diagram shows the dependencies between instructions.

Table 6.1.1: Pipeline Diagram with dependencies

Instruction	CC_0	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
add $3, $1, $2	IF	ID	EX	MEM	**WB**			
addi $4, $3, 8		IF	ID	EX	MEM	**WB**		
add $5, $4, $3			IF	ID	EX	MEM	WB	
add $7, $3, $2				IF	ID	EX	MEM	WB

List 6.1.1: Expected Results

Expected results without forwarding

- CC_5: \$3 = 0x1
- CC_6: \$4 = 0x8
- CC_7: \$5 = 0x3
- CC_8: \$7 = 0x2 (assuming the reading from RF is done on the falling edge)

Expected results with forwarding

- CC_3 : ForwardA = 0b01
- CC_4: ForwardA = 0b01 & ForwardB = 0b10
- CC_5: \$3 = 0x1
- CC_6: \$4 = 0x9
- CC_7: \$5 = 0xA
- CC_8: \$5 = 0x2

Figure 6.1.1: Actual Results

6.1.2 Forwarding to MEM stage (store instructions)

Listing 6.1.2: Test program for data hazards and forwarding to MEM stage

```
B"000_001_100_011_0_001", -- 0: add $3 = $1 + $4
B"011_011_011_0000001",   -- 1: sw $3 $1 1
B"011_011_011_0000001",   -- 1: sw $3 $1 2
others => x"0000"
```

Table 6.1.2: Pipeline Diagram with dependencies

Instruction	CC_0	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
add $3, $1, $4	IF	ID	EX	MEM	**WB**		
sw $3, $1, 1		IF	ID	EX	MEM	WB	
sw $3, $1, 2			IF	ID	EX	MEM	WB

List 6.1.2: Expected Results

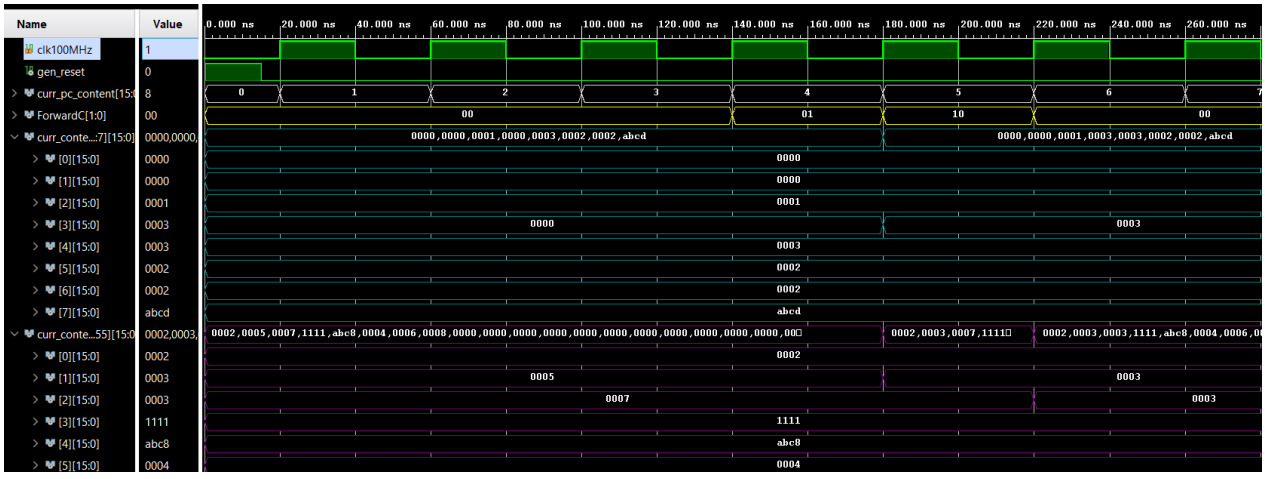
Expected results without forwarding

- CC_5: \$3 = 0x3
- CC_5: MEM[1] = 0x0
- CC_6: MEM[2] = 0x0

Expected results with forwarding

- CC_4: ForwardC = 0b01
- CC_5: ForwardC = 0b10
- CC_5: \$3 = 0x3
- CC_5: MEM[1] = 0x3
- CC_6: MEM[2] = 0x3

Figure 6.1.2: Actual Results



6.2 Testing the load data hazards

Listing 6.2.1: Test program to copy + paste in memory

```
B"010_001_010_0000010", -- 0: lw $2, $1, 2
B"011_001_010_0000011", -- 1: sw 2, $1, 3
others => x"0000"
```

Table 6.2.1: Pipeline Diagram with dependencies

Instruction	CC_0	CC_1	CC_2	CC_3	CC_4	CC_5
lw $2, $1, 2	IF	ID	EX	MEM	**WB**	
sw $2, $1, 3		IF	ID	EX	MEM	WB

List 6.2.1: Expected Results

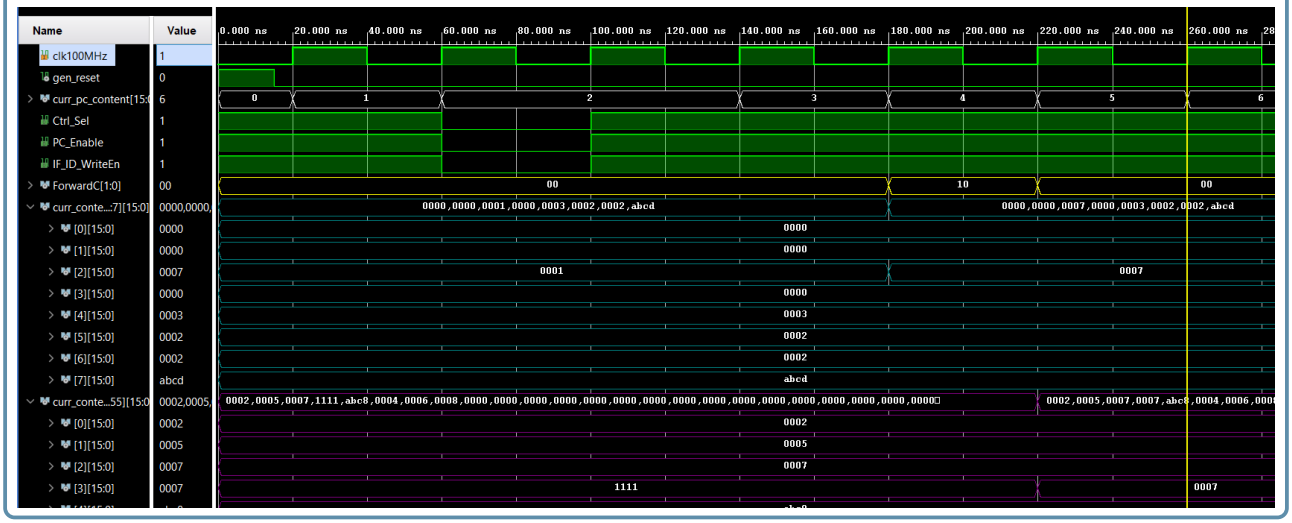
Expected results without forwarding

- CC_5: \$2 = 0x7
- CC_5: MEM[3] = 0x01

Expected results with forwarding

- CC_2: Stall ⇒ IF/ID_Write = 0 & Ctrl_Sel = 1 & PC_Enable = 0
- CC_4: ForwardC = 0b01
- CC_5: \$2 = 0x7
- CC_6: MEM[3] = 0x07 (delayed with one clock cycle)

Figure 6.2.1: Actual Results



6.3 Testing the control hazards

6.3.1 Taking advantage of the BHT in a loop

Listing 6.3.1: Test program for a simple loop - no dependencies

```
B"010_001_001_0000010",   -- 0: lw $1, $1, 2
B"001_000_000_0000001",   -- 1: addi $0, $0, 1
B"000_100_100_100_0_001", -- 2: add $4, $4, $4
B"000_101_101_101_0_001", -- 3: add $5, $5, $5
B"110_000_001_1111100",   -- 4: bneq $0, $1, -4
B"011_010_000_0000011",   -- 3: sw $0, $2
others => x"0000"
```

List 6.3.1: Expected Results

Expected results without forwarding

- the following PC content sequence: $0 \rightarrow 1 \ldots 4 \rightarrow 5 \rightarrow 1$
- the memory is affected every time the store enters the pipeline (as it is not flushed)
- when the loop ends there is no delay

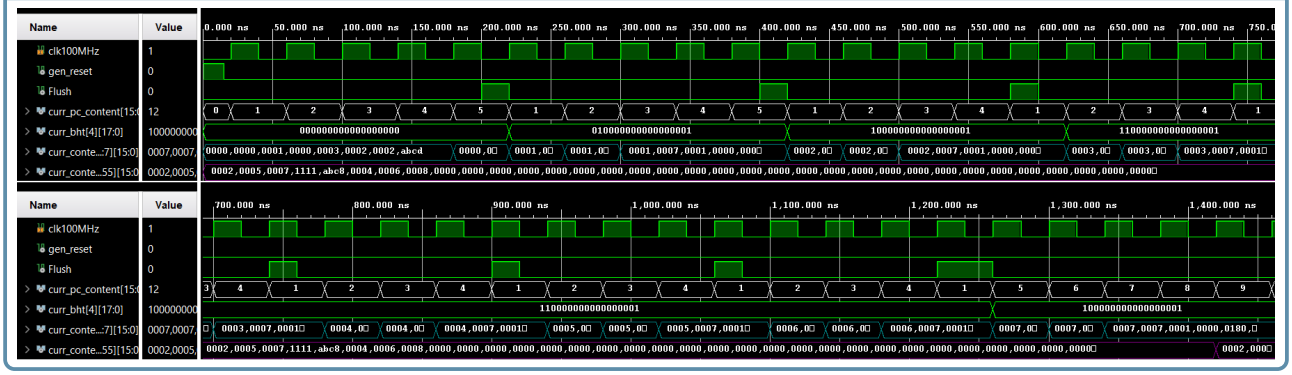
Expected results with forwarding

- the first two iterations the PC content sequence is the same, but the memory is not affected
- starting with the third iteration, the sequence immediately continues with 1 after 4
- the branch history table changes the counter everytime a branch is taken/not taken
- when the loop ends, the branch is taken by default and the instruction at the beginning of the loop is flushed.

Remark 6.3.1: Signals of interest

Our signals of interest are: curr_pc_content, curr_bht, and Flush

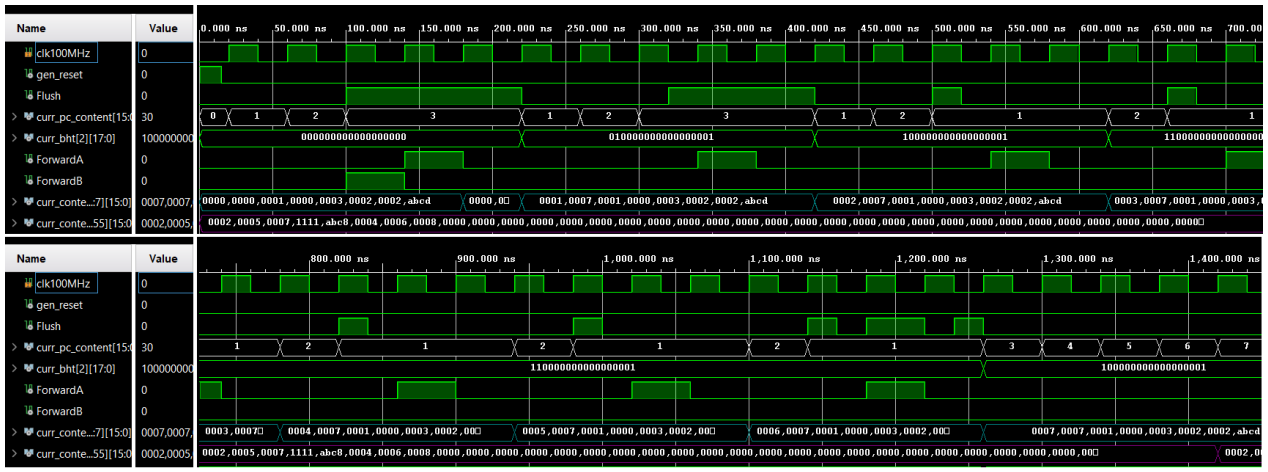
Figure 6.3.1: Results



6.3.2 Resolving data dependencies for branches

The removal of the two buffer instructions (2 and 3) in the previous program (and, of course changing the immediate value in the branch) will determine the same behavior, but some stalls (and forwarding) are introduced as the branch instruction depends on the one immediately previous.

Figure 6.3.2: Results



6.4 Fibonnaci numbers computer program - combining hazards

The following program computes the first 5 Fibonacci numbers and stores the fifth one in the memory it is executed multiple times to show that the persistence of the BHT data leads to better performance over time.

Listing 6.4.1: Fibonnaci counter program

```
B"011_000_000_0000000",-- 0: store MEM[0] <= $0 - constant 0
B"010_000_100_0000000",-- 1: load $4 MEM[0] = 0
B"010_000_101_0000001",-- 2: load $5 MEM[1] = 5

    -- Loop start
    B"000_001_010_011_0_001", -- 3: add $3 = $1 + $2
    B"000_010_000_001_0_001", -- 4: add $1 = $2 + $0
    B"000_011_000_010_0_001", -- 5: add $2  = $3 + $0
    B"001_100_100_0000001", -- 6: addi $4 = $4 + 1 -- counter
    B"110_100_101_1111011", -- 7: bneq $4 $5 -5

B"011_000_010_0000110", -- 8: store MEM[6] <- $2
B"111_0000000000000",-- 9: Jump 0
others => x"0000"
```

List 6.4.1: Expected Results

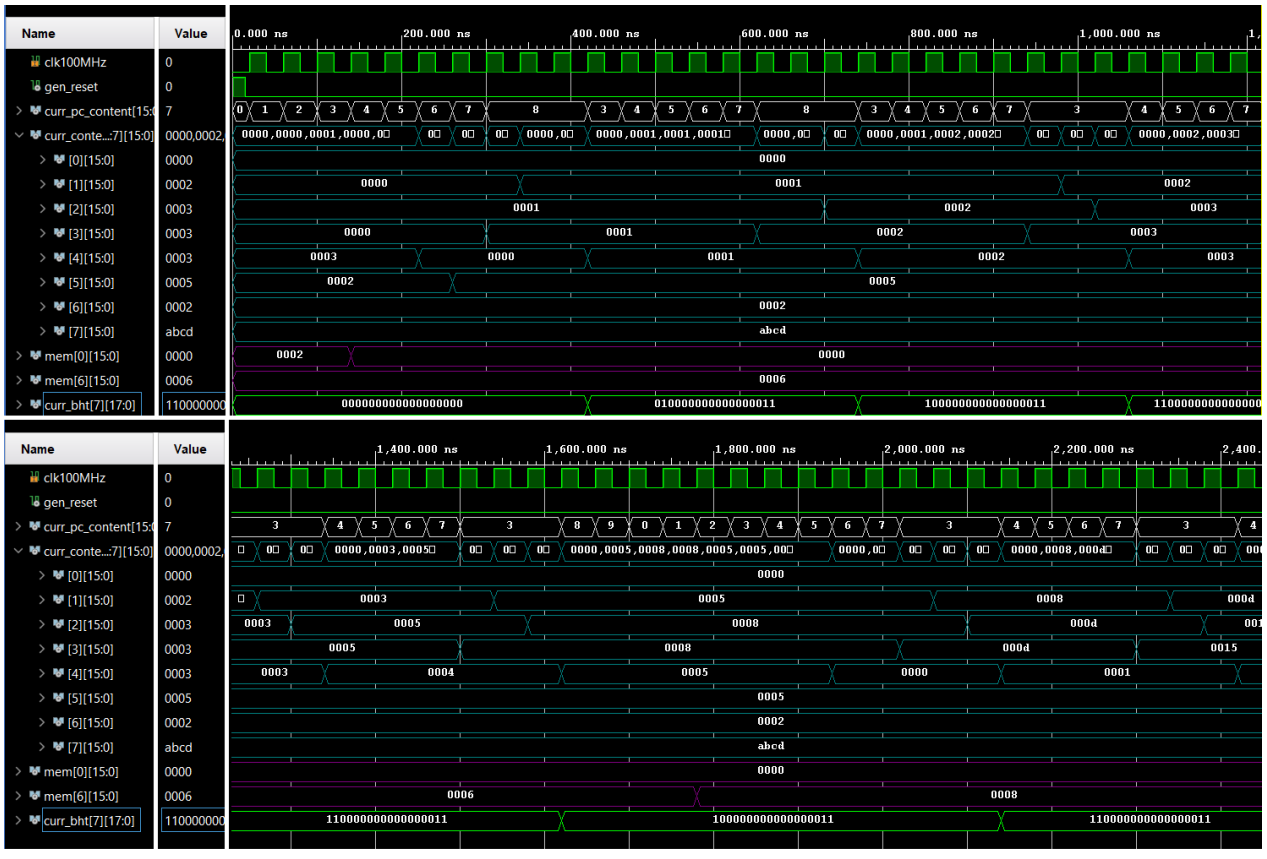
The program contains the following hazards

- data hazard between 1 and 0
- data hazards between all instructions in the loop
- data hazard between 7 and 6 (needs forwarding and stalling)
- control hazard determined by the branch instruction

Therefore, the execution will certainly not yield the expected result, and it will modify the memory contents every time the loop is executed.
Moreover, the loop execution will not improve over time.

With hazard handling, the result in the memory location MEM[6], after the execution of the program, should be 8. Moreover, after the program is executed once, the branch should be taken after the first iteration.

Figure 6.4.1: Results



Conclusions

At first sight, solving pipelining hazards using hardware mechanisms may not seem to be such a complicated problem. However, while going through the analysis, design, and implementation it can be noticed that the possible scenarios tend to add up. Some solutions to a hazard may bring other hazardous conditions that need to be treated as well. Moreover, the complexity of such a project grows when going in-depth for each scenario.

Nevertheless, the goal of obtaining a hazard-proof pipeline MIPS implementation has been achieved. Both data and control hazards are now resolved. This enables the designer to use the pipeline MIPS without having to worry about treating hazards in their programs.

With a simple FPGA board (such as Digilent's Basys3) this MIPS implementation is placed at the center of a prototyped computer system. If needed the instruction set architecture could easily be extended and peripherals can be added. The sky is the limit.

Bibliography

[1] D. A. Patterson and J. L. Hennessy, “Computer organization and design: the hardware/-software interface 5th ed.” 2014.