Detecting and Avoiding Hazard Situations in Pipeline Architectures

Student: Leo Mihai Ioan

Structure of Computer Systems Projects

Technical University of Cluj-Napoca

2024

Contents:

Introduction

1.1 Context

The pipeline design of the MIPS comes with a lot of advantages such as instruction level parallelism which increases the throughput, but due to its architecture Structural, Data and Control hazards might appear.

The purpose of this unit is to detect and avoid them, resulting in a more reliable CPU unit.

1.2 Objectives

The main objective of this project is the VHDL implementation of the unit, as well as the creation of a testbench where the unit will be integrated with the already existing MIPS pipeline.

The hazards that need to be tackled are:

1 .Structural Hazards:

- resource dependency
- two instructions need to use the same resource at the same time
- solved by stealing or implementing asynchronous reads

1 .Data Hazards:

- data dependency
- an instruction needs data that is not yet available
- solved by forwarding

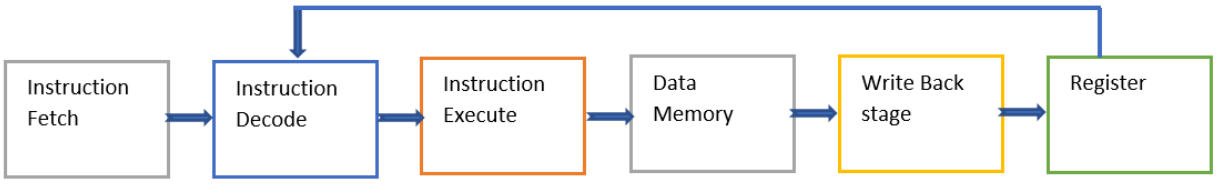
1 .Control Hazards:

- flow control
- The program control flow is decided before the new target address is calculated, mainly in jump and branch instructions
- solved by branch prediction

Bibliographic Research

2.1 MIPS pipeline

The mips pipeline is an architecture that uses a 5 stage pipeline to execute a set of instructions by overlapping their execution (instruction level parallelism ILP). It has a single-cycle datapath divided into the IF,ID,EX,MEM and WB stages each connected with the next by a register.



Pipelined MCU

(1)

2.2 Hazards

Hazards occur when an instruction cannot be executed until a previous one is executed, needing the resulting data or is using the same component. They can be solved by delaying, stalling the pipeline until the hazard is resolved, but this will increase the duration time.

2.3 Structural Hazards

When two instructions try to use the same component of the CPU the only solution is stall. The control unit should check if the components needed by the instruction would be free at the moment of the execution; if not then it will add a NoOpp instruction.

2.4 Data Hazards

They arise when the current instruction needs the result of previous instruction that has not been completed. The most common type is the RAW(read after write) hazard, there can also be WAR(write after read) and WAW(write after write) hazards, but due to the structure of the MIPS they rarely occur.

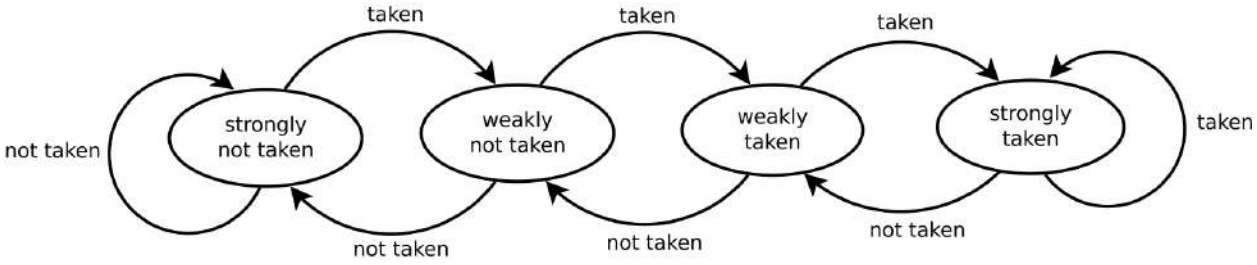
The RAW hazard can be solved by stalling, but a more efficient method is forwarding. If the result of an instruction would be needed by an immediate one (more than 3 clock cycles), its result would be transferred to the instruction that it needs before being written.

2.5 Control Hazards

When a jump or brunch instruction appears, the next instruction is known only when the result of the condition is obtained, if the assumption is made incorrect then the next instructions need to be flushed. The branch hazard can be solved by stolling until the result is obtained, but a most useful solution is predicting the next instruction.

Dynamic branch prediction keeps a record of the past behavior of the branches and makes an assumption based on that.

2 bit predictor



(2)

Analysis

3.1 Project Proposal

The final design of the MIPS pipeline with hazard detection and resolution aims to enhance the pipeline's efficiency while handling various types of hazards. The project involves implementing techniques like pipeline stalls, data forwarding, and dynamic branch prediction to maintain efficient instruction flow in the presence of hazards.

3.2 Project Analysis

3.2.1 Fixing structural hazards

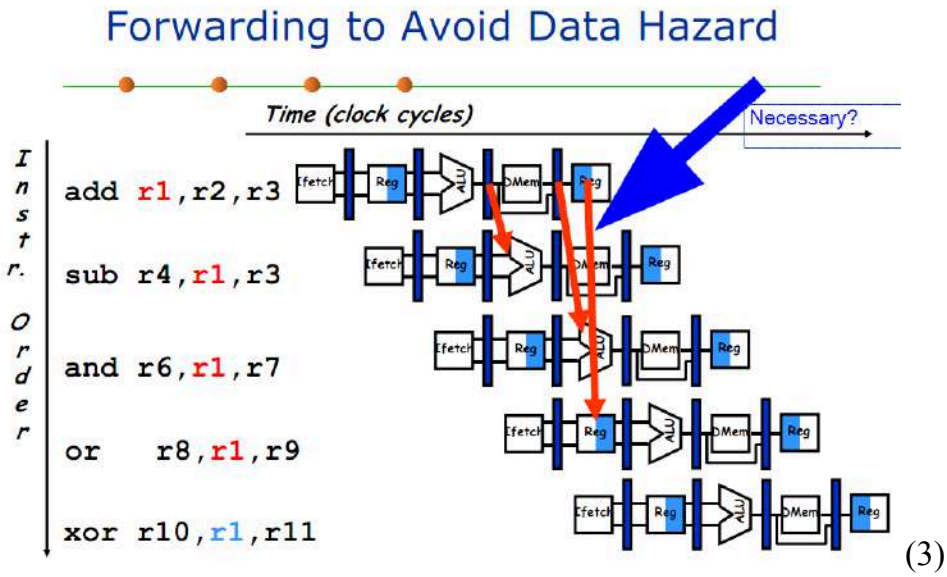
The only structural hazard present in the MIPS pipe architecture is the read and write from the register file in the same clock cycle:

- this can easily be fixed by implementing the write of the falling edge of the clock signal

3.2.2 Fixing data hazards

In case of multiple data sources, the forwarding mechanism prioritizes the most recent data written to the register. This ensures that instructions receive the correct values and that the pipeline's functionality aligns with the program's sequential requirements.

Data hazards, specifically Read-After-Write (RAW) and Load Data Hazards (LDH), arise due to dependencies between instructions. Here, a hardware solution using a forwarding unit and hazard detection unit is crucial.



(3)

1. **Read-After-Write (RAW) Hazard and Forwarding**
   RAW hazards occur when an instruction requires data from a previous instruction that hasn't completed its execution. The forwarding unit in this pipeline architecture detects the availability of the required data in the EX or MEM stages and bypasses it directly to the next instruction needing it. Scenarios involving RAW hazards include:
   - When the result of an instruction in the EX/MEM pipeline register is needed by a subsequent instruction in the EX stage.
   - When the data in the MEM/WB pipeline register is required for an instruction in the EX stage.
   - When a store instruction needs the value being written to a register in the MEM stage.

2. **Load Data Hazard (LDH) and Pipeline Stall**
   LDH occurs when a load instruction is immediately followed by an instruction that uses the loaded data. Since forwarding cannot provide the data from the MEM stage in time, a one-cycle stall (bubble) is inserted to delay the dependent instruction until the data is available. This delay prevents erroneous data reads and ensures data consistency across the pipeline.

3.2.3 Fixing control hazards

Branch prediction is a technique in MIPS (and other) pipelined processors that helps improve the efficiency of executing instructions by guessing the outcome of conditional branches before they are resolved.

How 2-Bit Dynamic Branch Prediction

The 2-bit predictor uses a 2-bit saturating counter for each branch instruction in a table, which records the recent behavior of that branch. The idea is to store a state for each branch that can be updated based on the most recent outcomes and used for prediction in the future.

Each counter has four possible states:

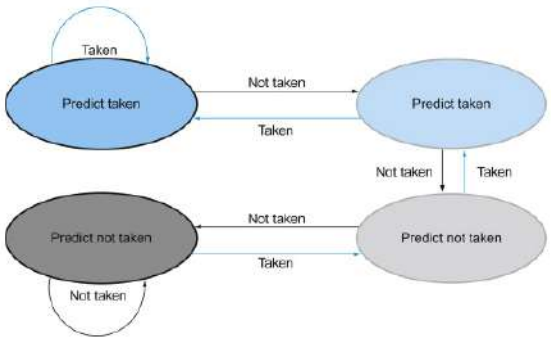
- Strongly Not Taken (00) Weakly Not Taken (01) Weakly Taken (10) Strongly Taken (11)

These states represent the predictor's confidence about whether the branch will be taken or not.

State Transitions and Prediction:

1. Predict Taken: If the counter is in either of the "Taken" states (10 or 11), the branch is predicted to be taken.
2. Predict Not Taken: If the counter is in either of the "Not Taken" states (00 or 01), the branch is predicted to be not taken.

When the actual outcome of the branch instruction is known (after it completes in the pipeline), the 2-bit counter is updated as follows:



(4)

This 2-bit counter design allows the predictor to tolerate occasional mispredictions without drastically changing its prediction, which is why it’s called a saturating counter.

Integration into the MIPS Pipeline:

In a MIPS pipeline with branch prediction:

1. The prediction is made early in the pipeline (usually in the Instruction Fetch stage) so that the pipeline can speculatively fetch the next instruction.
2. If the prediction is correct, the pipeline continues smoothly.
3. If the prediction is incorrect, a pipeline flush is required to discard the speculatively fetched instructions, and the correct instructions are fetched instead. This introduces a penalty, but the dynamic prediction mechanism aims to reduce these penalties by predicting branches accurately most of the time.

Design

4.1 MIPS pipe design

The MIPS is a designed on a 16 bit architecture there 3 types of instructions :

function			OPcode		
000	0	xor	000	0	R-type
001	1	and	001	1	xori
010	2	or	010	2	andi
011	3	add	011	3	ori
100	4	sub	100	4	lw
101	5	sll	101	5	sw
110	6	srl	110	6	branc
111	7	sra	111	7	jump

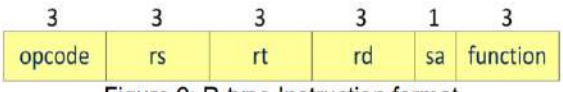


Figure 2: R-type Instruction format

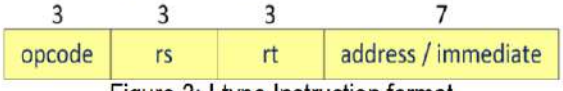


Figure 3: I-type Instruction format

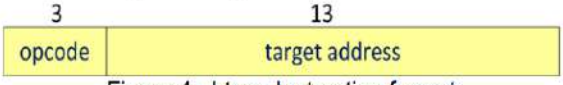


Figure 4: J-type Instruction format

(5)

Fields Explanation:

- rs: Source register 1. It usually holds one operand for the operation.
- rt: Source register 2 (or target register in I-type instructions). It can hold a second operand or serve as the destination in I-type instructions.

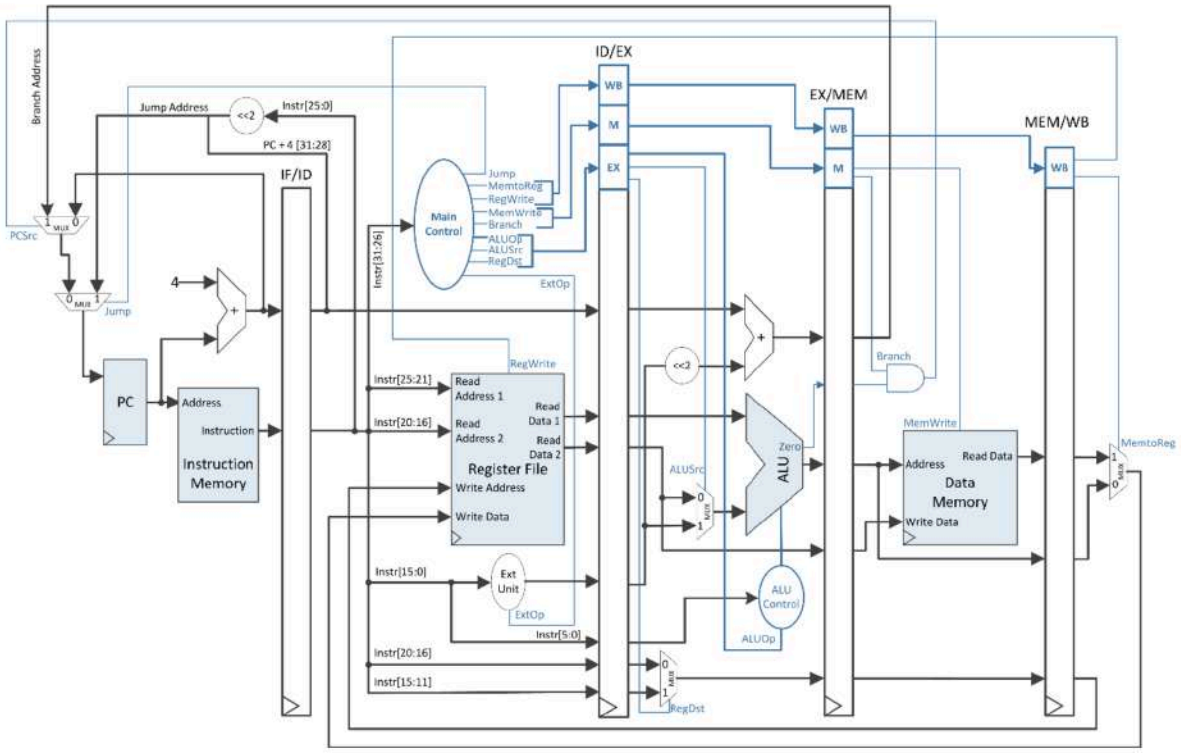
- rd: Destination register. For R-type instructions, the result of the operation is stored here.
- sa: Shift amount for shift instructions
- function: Function to determine the operation for T-type instruction
- immediate: Immediate value. A constant value used in I-type instructions, often providing offset values or immediate data for operations.
- target address: Address for jump instruction

Function	Operation	Type	Explanation
000	xor	R-type	rd = rs XOR rt. This performs a bitwise XOR between rs and rt, storing the result in rd.
001	and	R-type	rd = rs AND rt. This performs a bitwise AND between rs and rt, storing the result in rd.
010	or	R-type	rd = rs OR rt. This performs a bitwise OR between rs and rt, storing the result in rd.
011	add	R-type	rd = rs + rt. This adds the values in rs and rt, storing the result in rd.
100	sub	R-type	rd = rs - rt. This subtracts the value in rt from rs, storing the result in rd.
101	sll	R-type	rd = rt << sat. This shifts the bits in rt to the left by a certain amount, storing the result in rd.
110	srl	R-type	rd = rt >> sa. This shifts the bits in rt to the right, storing the result in rd.
111	sra	R-type	rd = rt >> sa (arithmetically). This performs an arithmetic right shift on rt and stores the result in rd.

Opcode	Function	Type	Explanation
000	R	R-type	Opcode for R-type instruction
001	xori	I-type	rt = rs XOR imm. This performs a bitwise XOR between rs and an immediate value, storing the result in rt.
010	andi	I-type	rt = rs AND imm. This performs a bitwise AND between rs and an immediate value, storing the result

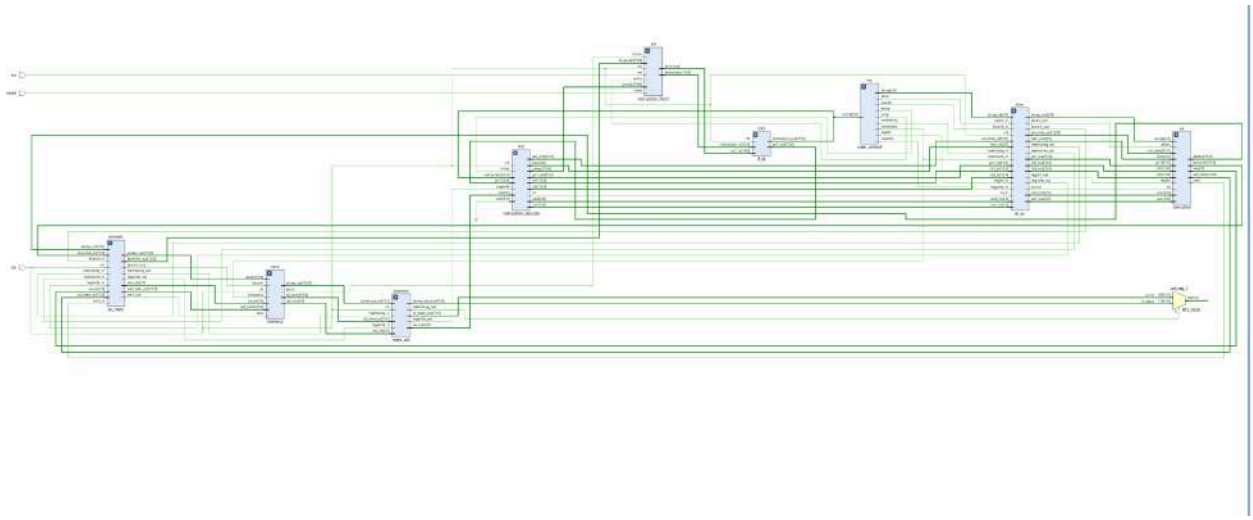
			in rt.
011	ori	I-type	rt = rs OR imm. This performs a bitwise OR between rs and an immediate value, storing the result in rt.
100	lw	I-type	rt = Mem[rs + imm]. This loads a word from memory at the address rs + imm into rt.
101	sw	I-type	Mem[rs + imm] = rt. This stores the value in rt into memory at the address rs + imm.
110	branc	I-type	if rs == rt then PC = PC + imm. This performs a branch if rs is equal to rt.
111	jump	I-type	PC = imm. This sets the program counter to imm, causing a jump to a new location.

The mips pipeline design

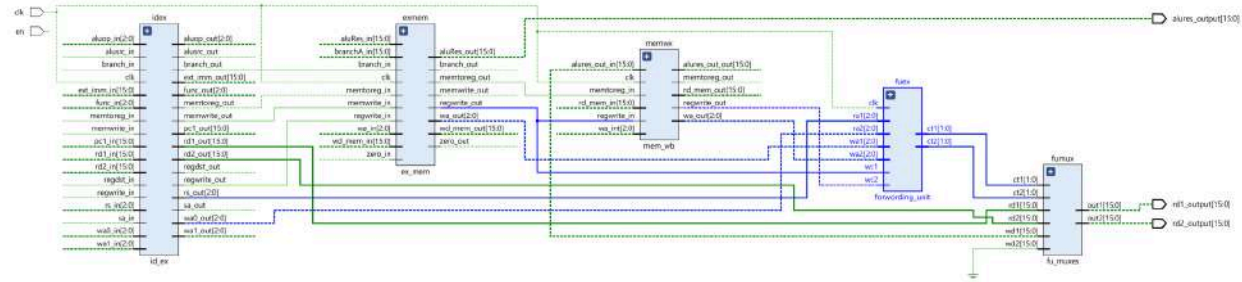


(6)

Vivado generated schematic



4.2 Forwarding unit design



When the addresses of the sources in the execution stage match the address of the destination from memory or write back stages a signal will be generated

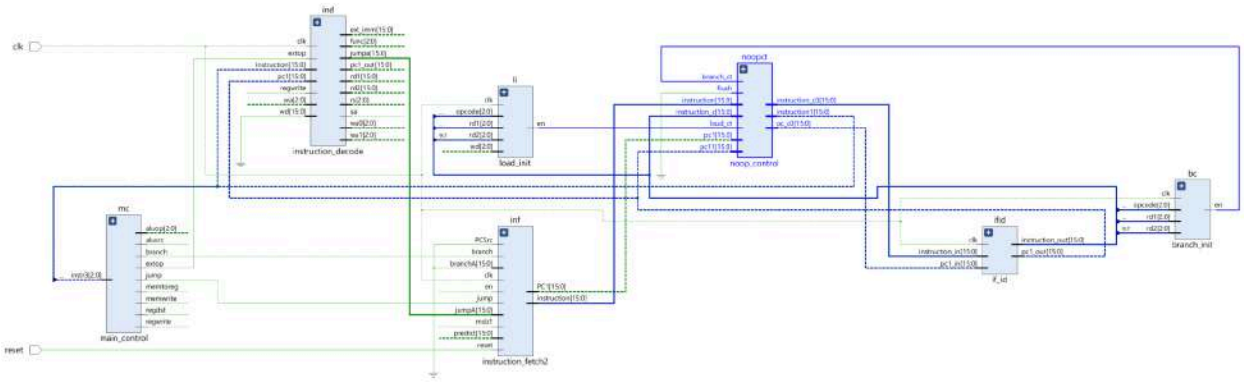
A signal can be:

00 or 11- if no forwarding is not indeed

01- if forwarding from wb stage

10- if forwarding from mem stage

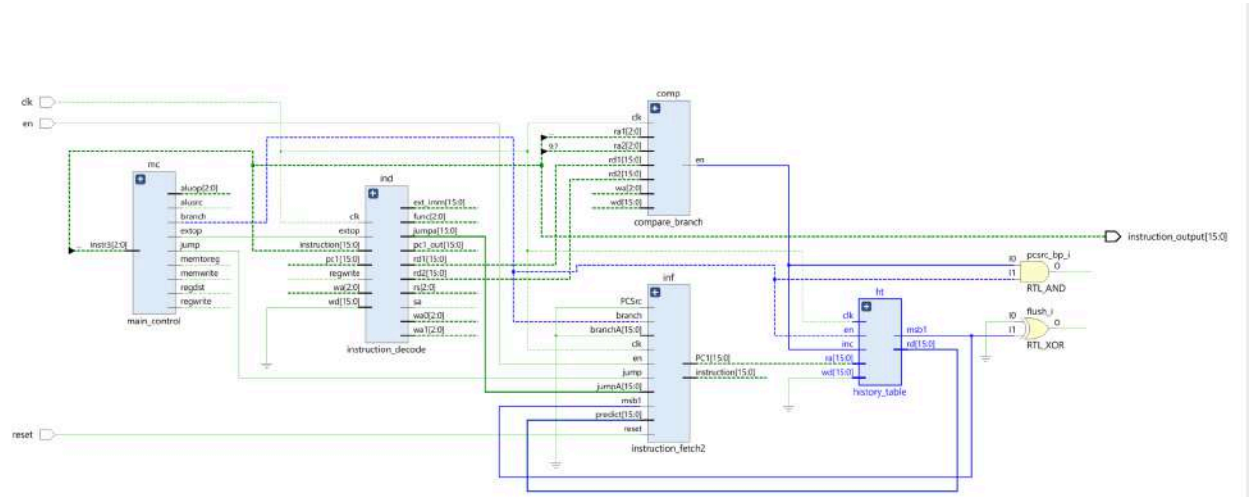
4.3 Load and Branch hazard control design:



When a load or branch instructions, that cause hazards, are detected in the pipeline a No operation instruction will be added.

This will be done with the help of jump instruction, that will cause no change to the data, and return the program counter back;

4.4 Branch prediction design:



Implementation

5.1Forwarding unit

Will generate signals to see which date needs to be forward it to the execution stage:

Ports:

- *clk:in std_logic;*
- *ra1: in std_logic_vector(2 downto 0);*
- *ra2: in std_logic_vector(2 downto 0);*
- *wa1: in std_logic_vector(2 downto 0);*
- *wa2: in std_logic_vector(2 downto 0);*
- *ct1: out std_logic_vector(1 downto 0);*
- *ct2: out std_logic_vector (1 downto 0)*

Design:

ct1<="01" when ra1=wa2 else "10" when ra1=wa1 else "00";
ct2<="01" when ra2=wa2 else "10" when ra2=wa1 else "00";

Signals explained:

ra1->rs
ra2->rt
wa1-> destination address in the memory stage
wa21> destination address in write back stage

The muxes for the the feed the date for the execution unit:
Ports:

- *rd1: in std_logic_vector(15 downto 0);*
- *rd2: in std_logic_vector(15 downto 0);*
- *wd1: in std_logic_vector(15 downto 0);*
- *wd2: in std_logic_vector(15 downto 0);*

- *ct1: in std_logic_vector(1 downto 0);*
- *ct2: in std_logic_vector(1 downto 0);*
- *out1: out std_logic_vector(15 downto 0);*
- *out2: out std_logic_vector(15 downto 0)*

Design:

out1<=wd2 when ct1="01" else wd1 when ct1="10" else rd1;
out2<=wd2 when ct2="01" else wd1 when ct2="10" else rd2;

Signals explained:

rd1->data stored in rs
rd2->data stored in rt
wd1-> write data in the memory stage
wd2-> write data in write back stage
out1-> first input to the execution unit
out2-> second input to the execution unit

5.2 Load and Branch hazard control

No operation control:

Ports:

- *loud_ct : in std_logic;*
- *branch_ct : in std_logic;*
- *flush: in std_logic;*
- *pc1: in std_logic_vector(15 downto 0);*
- *pc11: in std_logic_vector(15 downto 0);*
- *instruction: in std_logic_vector(15 downto 0);*
- *instruction_c: in std_logic_vector(15 downto 0);*
- *instruction1: out std_logic_vector(15 downto 0);*
- *instruction_c0: out std_logic_vector(15 downto 0);*
- *pc_c0: out std_logic_vector(15 downto 0);*

Design:

```
noop_ct<=loud_ct or branch_ct ;
noop2_ct<= loud_ct and branch_ct;
instruction1 <= "111" & pc11(12 downto 0)-x"0001" when noop2_ct='1'
        else "111" & pc11(12 downto 0) when noop_ct = '1' else
        instruction_c;
instruction_c0 <=x"0001" when instruction_c(15 downto 13)="111"
         or noop2_ct='1' or flush='1' else instruction_c when noop_ct = '1'
        else instruction;
pc_c0 <= pc11 when noop_ct = '1' else pc1;
```

Signals explained:

load_ct-> load hazard
branch_ct-> branch hazard
pc1,pc11-> the instruction addresses in if and id stages
instruction_c-> the instruction in the first flip flop
instruction_c-> the instruction given to the first flip flop
pc_c0-> the instruction address given to the first flip flop
instruction1-> instruction use by the instruction decode component

Load Hazard detect:

Ports:

- *clk: in std_logic;*
- *memtoreg: in std_logic;*
- *opcode: in std_logic_vector(2 downto 0);*
- *rd1: in std_logic_vector(2 downto 0);*
- *rd2: in std_logic_vector(2 downto 0);*
- *wd: in std_logic_vector(2 downto 0);*
- *en: out std_logic ;*

Design:

```
en<='0' when memtoreg ='0' else '1' when ((rd1=wd) and opcode/="000")
        else '1' when ((rd1=wd) or (rd2=wd)) else '0';
```

Branch Hazard detect:

Ports:

- *clk: in std_logic;*
- *opcode: in std_logic_vector(2 downto 0);*
- *rd1: in std_logic_vector(2 downto 0);*
- *rd2: in std_logic_vector(2 downto 0);*
- *wd: in std_logic_vector(2 downto 0);*
- *en: out std_logic*

Design:

```
aux<='0' when opcode /="110" or p='1' else '1' when ((rd1=wd) or
        (rd2=wd)) else '0';
process (clk)
begin
         if rising_edge (clk ) then
            p<=aux;
        end if;
end process;
en<=aux;
```

5.3 Branch prediction

History table:

Ports:

- *clk: in std_logic;*
- *en: in std_logic;*
- *inc: in std_logic;*
- *ra: in std_logic_vector(15 downto 0);*
- *wd: in std_logic_vector(15 downto 0);*
- *msb: out std_logic;*
- *msb1: out std_logic;*

- *rd: out std_logic_vector(15 downto 0);*

Design:

```
ra1<=ra-"0001";
process(clk)
begin
   if rising_edge( clk) then
      ls_ra<=ra-x"0001";
   end if;
end process;
 process(clk)
  begin
   if rising_edge(clk) and en='1' then
      if predict(conv_integer(ls_ra))=wd then
         if inc = '1' then
            case state(conv_integer(ls_ra)) is
               when "00" => state(conv_integer(ls_ra))<="01";
               when "01" => state(conv_integer(ls_ra))<="11";
               when "10" => state(conv_integer(ls_ra))<="11";
               when "11" => state(conv_integer(ls_ra))<="11";
               when others => state(conv_integer(ls_ra))<="10";
             end case;
         elsif inc='0' then
            case state(conv_integer(ls_ra)) is
               when "00" => state(conv_integer(ls_ra))<="00";
               when "01" => state(conv_integer(ls_ra))<="00";
               when "10" => state(conv_integer(ls_ra))<="00";
               when "11" => state(conv_integer(ls_ra))<="10";
               when others => state(conv_integer(ls_ra))<="10";
            end case;
         end if;
      else
         predict(conv_integer(ls_ra))<=wd;
         state(conv_integer(ls_ra))<="01";
      end if;
```

```
   end if;
  end process;
rd<=predict(conv_integer(ra1))when state(conv_integer(ra1))(1)='1' else
ra;

msb<=state(conv_integer(ra1))(1);
msb1<=state(conv_integer(ls_ra))(1);
```

Signals explained:

load_ct-> load hazard
branch_ct-> branch hazard
pc1,pc11-> the instruction addresses in if and id stages
instruction_c-> the instruction in the first flip flop
instruction_c-> the instruction given to the first flip flop
pc_c0-> the instruction address given to the first flip flop
instruction1-> instruction use by the instruction decode component

Testing and Validation

The testing part involves running a set of instruction into the mips pipeline that could trigger the hazard

Register file:

x"0000",--0
x"0002",--1
x"0004",--2
x"0006",--3
x"0008",--4
x"000A",--5
x"000C",--6
x"00FF",--7

Data memory:

signal ram: rammem:=(

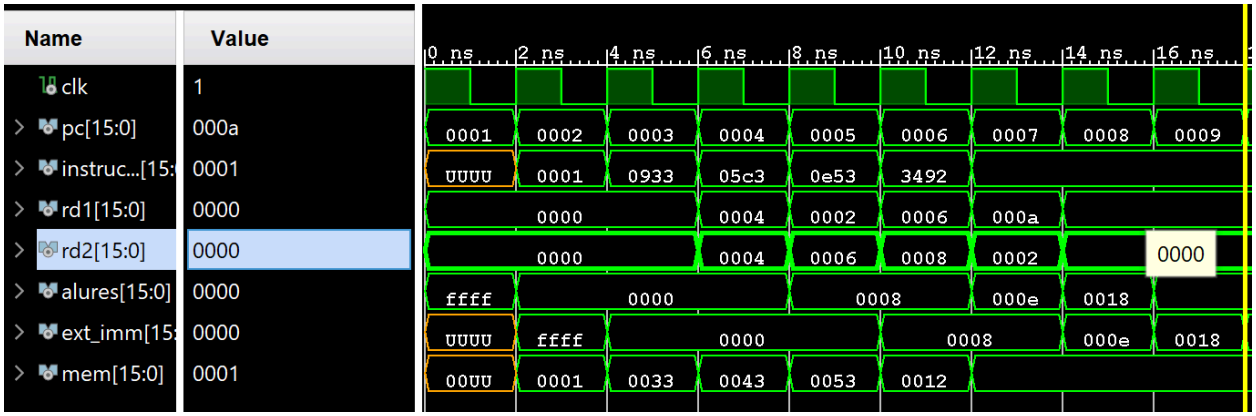
```
x"0001",
x"0002",
x"0003",
x"0004",
x"0005",
x"0006",
x"0007",
x"0008",
x"0009",
x"000A",
x"000B",
x"000C",
x"000D",
x"000E",
x"000F",
others =>x"0000"
);
```

7.1 Testing data hazard

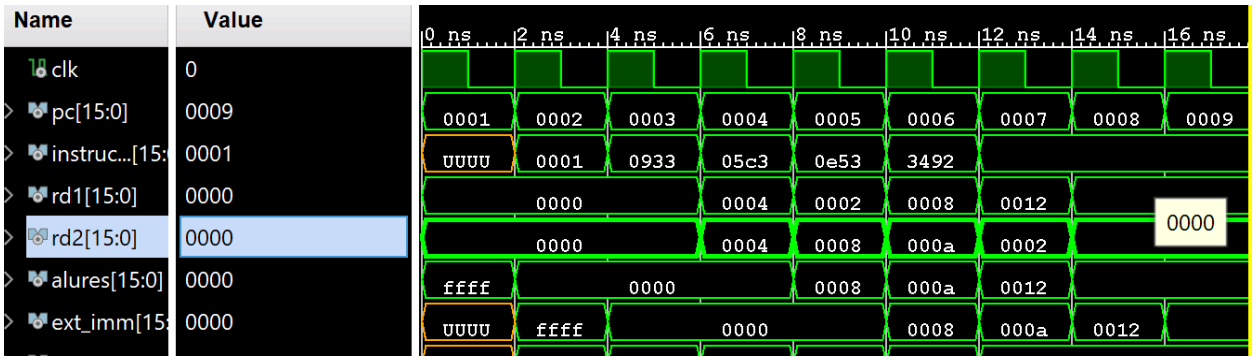
Testing forwarding unit:

OPP	DATA	BINARY	HEXA
add $3= $2+$2	4+4	000 010 010 011 0 011	0933
add $4= $1+$3	2+8	"000 001 011 100 0 011"	05c3
add $5=$3+$4	8+10	"000 011 100 101 0 011"	0e53
xor $1= $5&18	x12Xorx0a	"001 101 001 0010010"	3492

Without forwarding:



With forwarding:

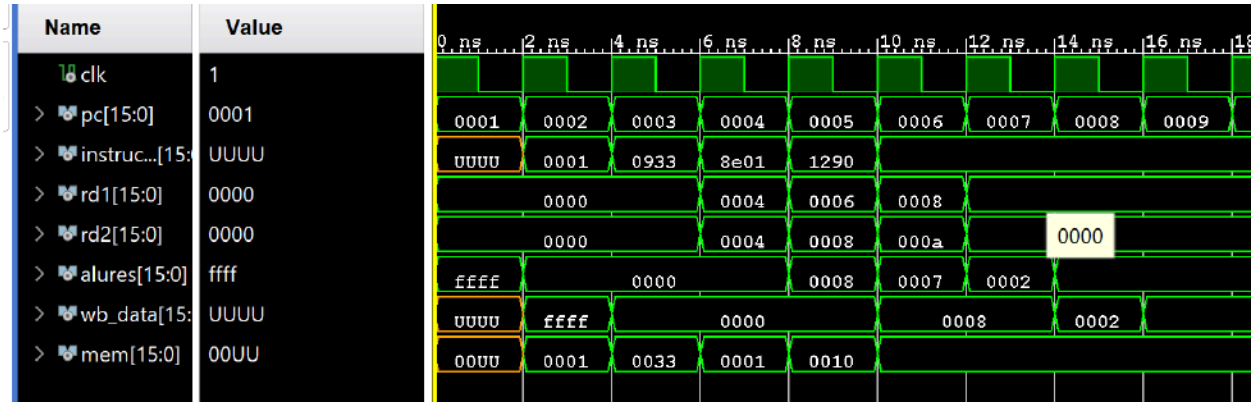


- we can observe that without the forwarding the additions uses the previous values of the register

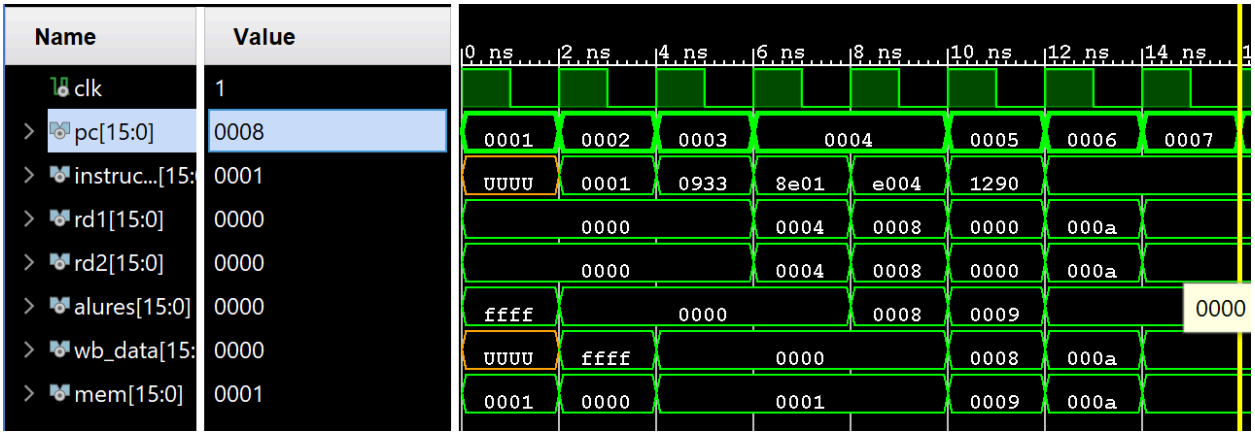
Testing LW pipeline stall:

OPP	DATA	BINARY	HEXA
add $3= $2+$2	4+4	"000 010 010 011 0 011"	0933
lw $4 = MEM[$3+1]	MEM[8+1]	"100 011 100 000 0 001"	8e01
xor &1= $4 xor $5	AxorA	"000 100 101 001 0 000"	1290

Without:



With

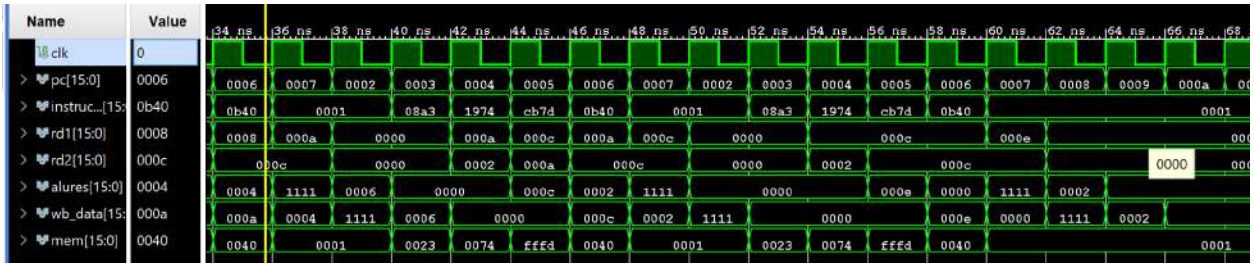


-e004: jump to position 4(the position of the instruction) to make one stall until load instruction is maded

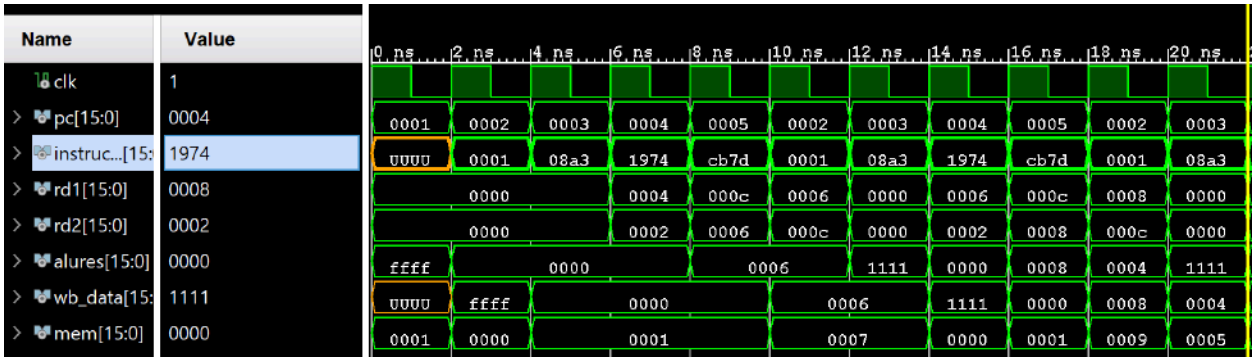
Testing Branch Prediction:

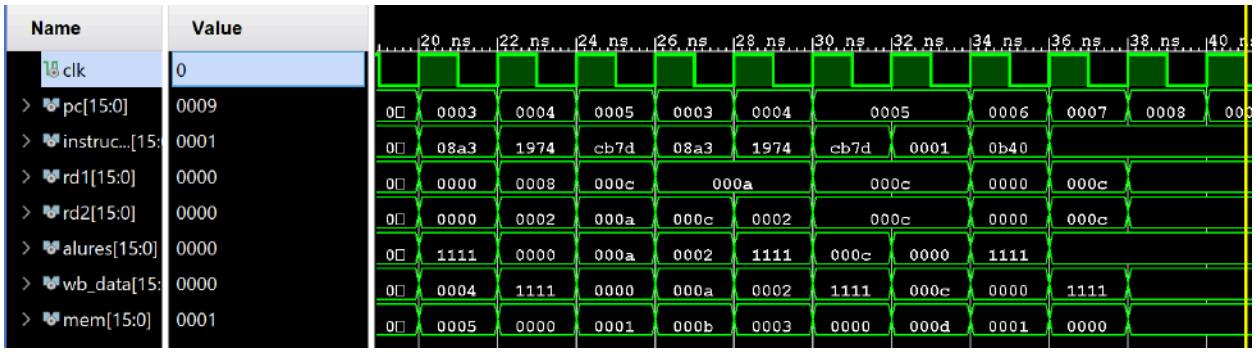
OPP	DATA	BINARY	HEXA
add $2=$2+$1	4+2..	"000 010 001 010 0 011"	08a3
sub $7=$6-$2	C-6..	"000 110 010 111 0 100"	1974
bneq $2 $6 -2	until C=C	"110 010 110 111 1 101"	cb7d
xor $4=$2xor$6	CxoriC	"000 010 110 100 0 000"	0b40

Without



With:

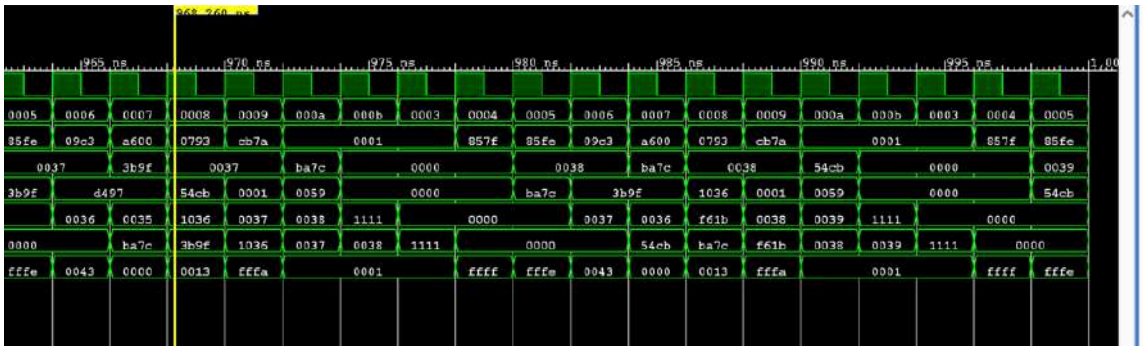




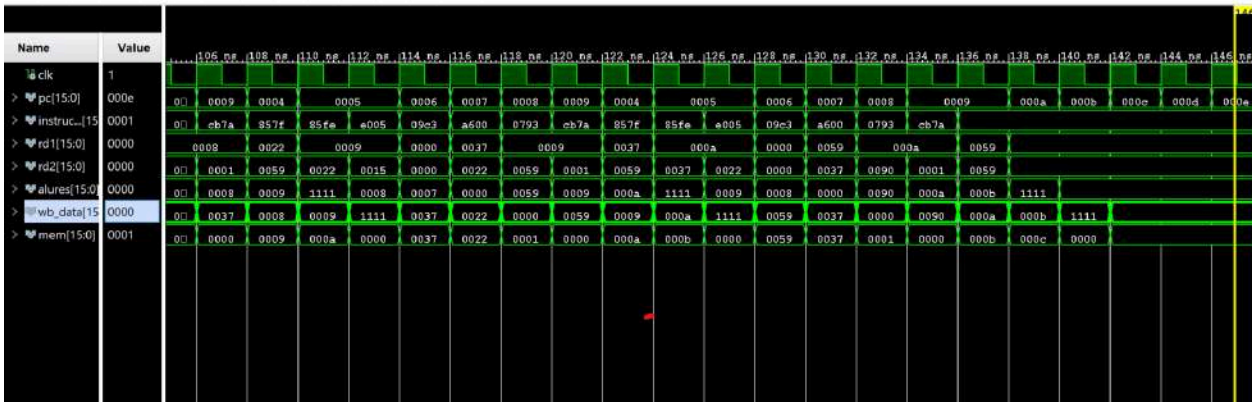
Generate Fibonacci:

OPP	DATA	BINARY	HEXA
lw $2 = MEM[$1-1]	Initial 1	1000010101111111	857f
lw $3=MEM[$1-2]	Initial 2	1000010111111110	85fe
add $4=$2+$3	Add the previous	0000100111000011	09c3
sw $4=MEM[$1]	Store the result	1010011000000000	a600
add $1=$1+$7	Increment position	000011110010011	0793
benq $2 $6 -5	Brench until =x59	1100101101111010	cb7a

without:



with:



- We can see that without the pipeline we would not even reach the x59 value

Bibliography

https://projectfpga.com/mips/ (1)
https://mihai.utcluj.ro/
https://www.cs.umd.edu/~meesh/411/CA-online/chapter/pipeline-hazards/index.html
Pictures:

https://en.m.wikipedia.org/wiki/File:Branch_prediction_2bit_saturating_counter-dia.svg (2)

https://stackoverflow.com/questions/5867374/mips-pipelining-question (3)

https://www.chegg.com/homework-help/questions-and-answers/fill-following-branch-prediction-information-table-various-types-branch-predictors-predict-q72428777(4)

https://mihai.utcluj.ro/wp-content/uploads/ca/labs/Lab07.pdf (5)

https://mihai.utcluj.ro/wp-content/uploads/ca/labs/Lab09.pdf (6)

https://stackoverflow.com/questions/40371727/mips-datapath-confusion (7)

https://courses.cs.washington.edu/courses/cse378/07au/lectures/L12-Forwarding.pdf