**Computer Architecture Final Assessment**

**Team :14**

**Name Sec BN**

**Ahmed Mohamed Zakaria 1 4**

**Abdullah Ezzat 1 34**

**Omar Salah Aly 2 1**

**Mohamed Ahmed Mohamed Ahmed 2 10**

Table of Contents

[Not implemented modules: 4](#_Toc41825579)

[**1.**Full design: 5](#_Toc41825580)

[Instruction Set Architecture 6](#_Toc41825581)

[Two operands: 6](#_Toc41825582)

[one operand: 7](#_Toc41825583)

[Memory operations: 8](#_Toc41825584)

[Branch and Change of Control Operations: 9](#_Toc41825585)

[Control Signals 10](#_Toc41825586)

[Data Forwarding Unit 13](#_Toc41825587)

[Hazard Detection Unit 17](#_Toc41825588)

[Branch-Hazard unit: 19](#_Toc41825589)

[Long-Fetch-Hazard unit: 20](#_Toc41825590)

[Load-use-Hazard unit: 20](#_Toc41825591)

[Wrong-prediction unit: 21](#_Toc41825592)

[RET-RTI-Reset-INT unit: 21](#_Toc41825593)

[Swap-Hazard unit: 21](#_Toc41825594)

[Swap-use-Hazard unit: 21](#_Toc41825595)

[PC Predictor 22](#_Toc41825596)

[2.Analysis: 24](#_Toc41825597)

[1. One-op testcase: 24](#_Toc41825598)

[2. Two-op testcase: 27](#_Toc41825599)

[3. Branch testcase: 29](#_Toc41825600)

[4. Branch prediction testcase: 34](#_Toc41825601)

[5. Memory testcase: 39](#_Toc41825601)

[Project Testing: 41](#_Toc41825602)

# Not implemented modules:

**The processor has no Memory cache system and all instructions are working**

# **1.**Full design:

## Instruction Set Architecture

Rsrc1 ; 1st operand register

Rsrc2 ; 2nd operand register

Rdst : result register

EA ; Effective address (20 bit)

Imm ; Immediate Value (16 bit)

### Two operands:

IR0 & IR1 = ’00’.

Possible formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SHL, SHR, | 5-bits opcode | 3-bits src1 | 8-bits don’t care | 16-bits Imm |
| SWAP, | 5-bits opcode | 3-bits src1 | 3-bits src2 | 2-bits don’t care | |
| ADD, SUB, AND, OR, | 5-bits opcode | 3-bits src1 | 3-bits src2 | 3-bits dst | 2-bits don’t care | |
| IADD, | 5-bits opcode | 3-bits src1 | 3-bits don’t care | 3-bits dst | 2-bits don’t care | | 16-bits Imm |

From IR0 -> IR4 “opcode”:

|  |  |
| --- | --- |
| Instruction | opcode |
| ADD | 00000 |
| SUB | 00001 |
| IADD | 00010 |
| AND | 00011 |
| OR | 00100 |
| SHL | 00101 |
| SHR | 00110 |
| SWAP | 00111 |

Format of src-reg code dst-reg code:

|  |  |
| --- | --- |
| Reg | Code |
| R0 | 000 |
| R1 | 001 |
| R2 | 010 |
| R3 | 011 |
| R4 | 100 |
| R5 | 101 |
| R6 | 110 |
| R7 | 111 |

### one operand:

IR0 & IR1 = ’01’

possible formats:

|  |  |  |
| --- | --- | --- |
| NOP, | 5-bits opcode | 11-bits don’t care |
| NOT, INC, DEC, OUT, IN, | 5-bits opcode | 3-bits src1 | 8-bits don’t care |

From IR0 -> IR4 “opcode”:

|  |  |
| --- | --- |
| Instruction | opcode |
| NOP | 01000 |
| NOT | 01001 |
| INC | 01010 |
| DEC | 01011 |
| OUT | 01100 |
| IN | 01101 |
| Free slot | 01110 |
| Free slot | 01111 |

Format of src-reg code dst-reg code:

|  |  |
| --- | --- |
| Reg | Code |
| R0 | 000 |
| R1 | 001 |
| R2 | 010 |
| R3 | 011 |
| R4 | 100 |
| R5 | 101 |
| R6 | 110 |
| R7 | 111 |

### Memory operations:

IR0 & IR1 = ’10’

possible formats:

|  |  |  |  |
| --- | --- | --- | --- |
| PUSH, POP, | 5-bits opcode | 3-bits dst | 8-bits don’t care |
| LDM, | 5-bits opcode | 3-bits dst | 8-bits don’t care | 16-bits Imm |
| LDD, | 5-bits opcode | 3-bits dst | 4-bits don’t care | 20-bits EA |
| STD, | 5-bits opcode | 3-bits src1 | 4-bits don’t care | 20-bits EA |

From IR0 -> IR4 “opcode”:

|  |  |
| --- | --- |
| Instruction | opcode |
| PUSH | 10000 |
| POP | 10001 |
| LDM | 10010 |
| LDD | 10011 |
| STD | 10100 |
| Free slot | 10101 |
| Free slot | 10110 |
| Free slot | 10111 |

Format of src-reg code dst-reg code:

|  |  |
| --- | --- |
| Reg | Code |
| R0 | 000 |
| R1 | 001 |
| R2 | 010 |
| R3 | 011 |
| R4 | 100 |
| R5 | 101 |
| R6 | 110 |
| R7 | 111 |

### Branch and Change of Control Operations:

IR0 & IR1 = ’11’.

possible formats:

|  |  |  |  |
| --- | --- | --- | --- |
| RET, RTI, | 5-bits opcode | 11-bits don’t care | |
| CALL, JMP, JZ | 5-bits opcode | 3-bits dst | 8-bits don’t care |

From IR0 -> IR4 “opcode”:

|  |  |
| --- | --- |
| Instruction | opcode |
| JZ | 11000 |
| JMP | 11001 |
| CALL | 11010 |
| RET | 11011 |
| RTI | 11100 |
| Free slot | 11101 |
| Free slot | 11110 |
| Free slot | 11111 |

Format of src-reg code dst-reg code:

|  |  |
| --- | --- |
| Reg | Code |
| R0 | 000 |
| R1 | 001 |
| R2 | 010 |
| R3 | 011 |
| R4 | 100 |
| R5 | 101 |
| R6 | 110 |
| R7 | 111 |

## Control Signals:

**Execute Signals: (3bits)**

**Alu Selector:**

ALU\_SEL = OPCODE(3 downto 0);

**IO/ALU: (1 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 0 - alu | Rest |
| 1 – io | 01101 |

**Out Selector: (1 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 0 – ‘Z’ | Rest |
| 1 – Rdst | 01100 |

**ALU operand 2 Selector: (1 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 0 – Rsrc2 | Rest |
| 1 – IMM | 00010  00101  00110  10010 |

**MEMORY SIGNALS: (7bits)**

**Read/Write Select: (1 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 0 – Read | 10001  10011  11011  11100 |
| 1 – Write | 10000  10100  11010 |

**Value Selector: (2 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 00 – “Z” | rest |
| 01 – [Rsrc1] | 10000  10100 |
| 10 – PC | 11010 |
| 11 – FLAGS&PC | INTR |

**Address Selector: (2 bits)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 00 – 1,0 | RST |
| 01 – 3,2 | INTR |
| 10 – EA | 10011  10100 |
| 11 – SP/SP+2 | 10000  10001  11010  11011  11100 |

**(SP ALU) + (SP/SP+2) Selector: (1 bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 0 – ‘+’ and ‘SP+2’ | 10001  11011  11100 |
| 1 – ‘-‘ and ‘SP’ | 10000  11010 |

**SP load: (1 bit)**

|  |  |
| --- | --- |
| **LOAD** | **OPCODE** |
| 0 | Rest of them |
| 1 | 10001  11011  11100  10000  11010 |

**WB SIGNALS: (4 bits)**

**Write Value Select: (2 bits)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 00 – ‘Z’ | rest |
| 01 – MEM | 10001  10011 |
| 10 – EXE | 01001  01010  01011  01101  00111  00000  00001  00010  00011  00100  00101  00110  10010 |
| 11 – [Rsrc1] | Second swap |

**Write Address Select: (2-bit)**

|  |  |
| --- | --- |
| **SEL** | **OPCODE** |
| 00 – Rsrc1 | 01001  01010  01011  01101  00111  00101  00110  10001  1001د0  10011 |
| 01 – Rdst | 00000  00001  00010  00011  00100 |
| 10 – Rsrc2 | Second swap |

## Data Forwarding Unit

Forwarding unit

Rs(1st src from ID/EX)

Rt(2nd src from ID/EX)

Forward A

Rd3(dst from EX/MEM)

Rd2(dst from ID/EX)

Forward B

Rd4(dst from MEM/WB)

EX. RegWrite

MEM.RegWrite

WB.RegWrite

|  |  |
| --- | --- |
| ALU operand A | ALU operand B |
| Else if ( (Rs == Rd3) and (MEM.RegWrite))  ForwardA = 01  Else if ((Rs == Rd4) and (WB.RegWrite))  Forward A =10  Else ForwardA=00 | Else if ( (Rt == Rd3) and (MEM.RegWrite))  Forward B= 01  Else if ((Rt == Rd4) and (WB.RegWrite))  Forward B=10  Else Forward B =00 |

Special unit "inside DF unit ":

R\_Dst(to PC\_Predictor)

Dst from Reg\_file

Dst from IF

Dst from EX/MEM

Dst from MEM/WB

If(Dst(from IF)== Dst(from EX/MEM))

{

R\_Dst(to PC\_Predictor)= Dst(from EX/MEM)

}

Else If(Dst(from IF== Dst(from MEM/WB))

{

R\_Dst(to PC\_Predictor)= Dst(from MEM/WB)

}

Else R\_Dst(to PC\_Predictor) = Dst(from reg\_file)

**INTR/RESET HANDLER**

If INTR\_MEM == 1 and INTR\_WB == 0:

Rd/Wr = 1 -- Wr

val\_sel = 00

add\_sel = 11

SP\_load = 1

SP\_Alu = 1

else If INT\_WB == 1:

Rd/Wr = 0 -- Rd

val\_sel = xx

add\_sel = 01

SP\_load = 0

SP\_Alu = x

else If Reset == 1:

Rd/Wr = 0 -- Rd

val\_sel = xx

add\_sel = 00

SP\_load = 0

SP\_Alu = x

else: Rd/Wr = Rd/Wr\_in -- Rd

val\_sel = val\_sel\_in

add\_sel = add\_sel\_in

SP\_load = SP\_load\_in

SP\_Alu = SP\_Alu\_in

**SWAP HANDLER**

If Opcode == 00111 and Val\_sel == 0:

val\_sel = 100

add\_sel = 1

else if opcode = 00111 and val\_sel != 0:

val\_sel = 101

add\_sel = 0

else:

val\_sel = val\_sel\_in

add\_sel = add\_sel\_in

## Hazard Detection Unit

Stall\_bit\_1

1-bit Reg

Opcode\_F

Rdst\_D\_code

Rdst\_F\_code

Branch-Hazard

Prediction\_bit

Rsrc1\_D\_code

Opcode\_D

Rsrc1\_E\_code

Opcode\_E

Rsrc1\_M\_code

Opcode\_M

Rdst\_M\_code

Rdst\_E\_code

Long-Fetch-Hazard

clr

Stall\_bit\_2

1-bit Reg

Opcode\_F

Fetch-Hazard

Previous\_Stall

1-bit Reg

Swap-Hazard

Opcode\_D

Load-use-Hazard

Stall\_bit\_3

Opcode\_D

Rsrc1\_D\_code

Opcode\_E

Rsrc2\_D\_code

Stall\_bit\_6

Rdst\_E\_code

Swap-use-Hazard

Opcode\_D

Opcode\_E

Opcode\_M

Stall\_bit\_7

Wrong-prediction

Rsrc2\_D\_code

Rsrc1\_D\_code

Rdst\_E\_code

Stall\_bit\_4

Opcode\_E

Rdst\_M\_code

Wrong-prediction-bit

Prediction-bit

ZF

Stall\_bit\_5

1-bit Reg

RET-RTI-Reset-INT

Opcode\_M

Opcode\_F

INT\_F

Falling-edge checker

INT\_W

Reset\_M

Reset\_F

Load\_ret\_PC

Hazard-Detection

Unit

Stall\_bit\_3

Stall\_bit\_1

PC\_Write

Stall\_bit\_6

Stall\_bit\_5

Stall\_bit\_7

Stall\_bit\_7

Stall\_bit\_6

Stall\_bit\_5

Stall\_bit\_4

Stall\_bit\_3

Control\_Unit\_Mux

Stall\_bit\_2

Stall\_bit\_1

## Branch-Hazard unit:

As we predict in fetch stage so if we have a jz instruction with Prediction\_bit = 1 “predict taken”, a jmp instruction or a call instruction so we need to pass Rdst to the PC.

This will cause hazard in some cases if Rdst is not ready in the register file.

Case 1:

* Add R3,R1,R2
* Jz R3
* Instr. “if taken”

When “jz” is in fetch stage, “add” will be in decode stage so Rdst will not be calculated yet.

We need to stall once then forward Rdst from execute stage to the PC.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Add R3,R1,R2 | F | D | E | M | W |  |  |  |
| Jz R3 |  | F | stall | stall | stall | stall |  |  |
| Jz R3 |  |  | F | D | E | M | W |  |
| Instr. |  |  |  | F | D | E | M | W |

Case 2:

* LDD R3,Imm
* Instr.
* Jz R3
* Instr. “if taken”

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LDD R3,Imm | F | D | E | M | W |  |  |  |  |
| Instr. |  | F | D | E | M | W |  |  |  |
| Jz R3 |  |  | F | stall | stall | stall | stall |  |  |
| Jz R3 |  |  |  | F | D | E | M | W |  |
| Instr. |  |  |  |  | F | D | E | M | W |

Case 3:

* LDD R3,Imm
* Jz R3
* Instr. “if taken”

Same as case 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LDD R3,Imm | F | D | E | M | W |  |  |  |  |
| Jz R3 |  | F | stall | stall | stall | stall |  |  |  |
| Jz R3 |  |  | F | stall | stall | stall | stall |  |  |
| Jz R3 |  |  |  | F | D | E | M | W |  |
| Instr. |  |  |  |  | F | D | E | M | W |

Case 4:

* Swap R1,R2
* Jmp R1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Swap R1,R2 | F | stall | stall | stall | stall |  |  |  |  |  |  |
| Swap R1,R2 |  | F | D | E | M | W |  |  |  |  |  |
| Jmp R1 |  |  | F | stall | stall | stall | stall |  |  |  |  |
| Jmp R1 |  |  |  | F | stall | stall | stall | stall |  |  |  |
| Jmp R1 |  |  |  |  | F | stall | stall | stall | stall |  |  |
| Jmp R1 |  |  |  |  |  | F | D | E | M | W |  |
| Instr. |  |  |  |  |  |  | F | D | E | M | W |

In general: if I have jz with taken pred. , jmp, call, I will stall once in these cases

1. The previous instr. is one-op, two-op, LDM, LDD or POP that has the Rdst as me.
2. The instr. before previous one is LDM, LDD or POP that has the Rdst as me.
3. One of the previous 3 instructions is swap with the Rsrc or Rdst as my Rdst

**Stall\_bit\_1**: stall the whole pipe

## Long-Fetch-Hazard unit:

There’re some instructions that are 32-bit in size so it can’t be fetched once from the memory.

So what we need is to fetch the first half and to stall this half in the decode till we fetch the second part and start decoding and to make sure that the next half will not cause this stalling even if it seems to be a 32-bit instr. cause in fact it’s not, it’s just the rest of the last instruction.

**Stall\_bit\_2**: stall decode of the next cycle only.

## Load-use-Hazard unit:

Example:

* LDD R3,Imm
* Add R1,R2,R3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LDD R3,Imm | F | D | E | M | W |  |  |  |  |
| Add R1,R2,R3 |  | F | D | stall | stall | stall |  |  |  |
| Add R1,R2,R3 |  |  | F | D | E | M | W |  |  |

Note: no register because I want to stop fetching next instruction once add get into decode stage to fetch the same instr. again.

**Stall\_bit\_3**: stall the whole pipe

## Wrong-prediction unit:

It outputs 1-bit “Wrong-prediction-bit” (prediction-bit XNOR ZF)

If there’s a “jz” instruction in execute stage, there are two cases:

* If prediction-bit = zero-flag : Wrong-prediction-bit = 0
* If prediction-bit != zero-flag : Wrong-prediction-bit = 1

**Stall\_bit\_4**: stall decode only.

## RET-RTI-Reset-INT unit:

If you found RET, RTI instruction or interrupt or reset in fetch stall upcoming instructions till this instruction or this interrupt/reset finishes memory stage so that the new PC is ready.

**Stall\_bit\_5**: stall the whole pipe

## Swap-Hazard unit:

If there’s a swap in decode it stalls it and re-fetches it again.

To make sure that the re-fetched swap will not cause another stall, we keep track of the last stall due to swap, if it was ‘1’ so do not stall.

**Stall\_bit\_6**: stall the whole pipe

## Swap-use-Hazard unit:

Example:

* Swap R1,R2
* Add R1,R2,R3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Swap R1,R2 | F | stall | Stall | stall | stall |  |  |  |  |
| Swap R1,R2 |  | F | D | E | M | W |  |  |  |
| Add R1,R2,R3 |  |  | F | D | stall | stall | stall |  |  |
| Add R1,R2,R3 |  |  |  | F | D | stall | stall | stall |  |
| Add R1,R2,R3 |  |  |  |  | F | D | E | M | W |

Note: no register because I want to stop fetching next instruction once add get into decode stage to fetch the same instr. again.

**Stall\_bit\_7**: stall the whole pipe

## PC Predictor

If (Wrong\_Prediction\_bit == 0 and Load\_ret\_PC== 0):

if ( (opcode\_F == jz) and (Prediction\_bit == 1) ) or ( (opcode\_F == jmp) or ( (opcode\_F == call):

PC\_predicted = Rdst\_val

PC\_UnPredicted = PC+1

If (opcode\_F == jz) and (Prediction\_bit == 0):

PC\_predicted = PC+1

PC\_UnPredicted = Rdst\_val

Else If (Wrong\_Prediction\_bit == 1):

PC\_predicted = Unpredictted\_PC\_E

PC\_UnPredicted = PC+1

Else If (Load\_ret\_PC== 1):

PC\_predicted = PC\_Mem

PC\_UnPredicted = PC+1

PC\_Write

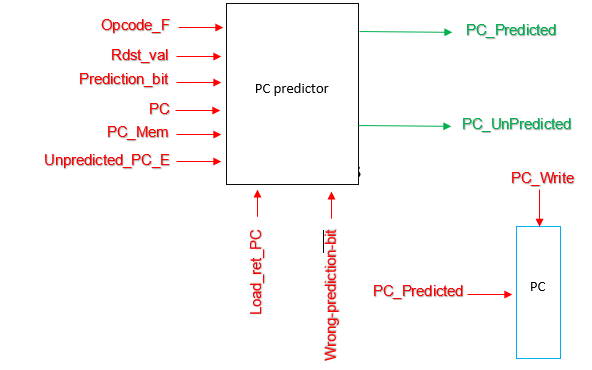
PC\_Predicted

PC

Load\_ret\_PC

Wrong-prediction-bit

Unpredicted\_PC\_E



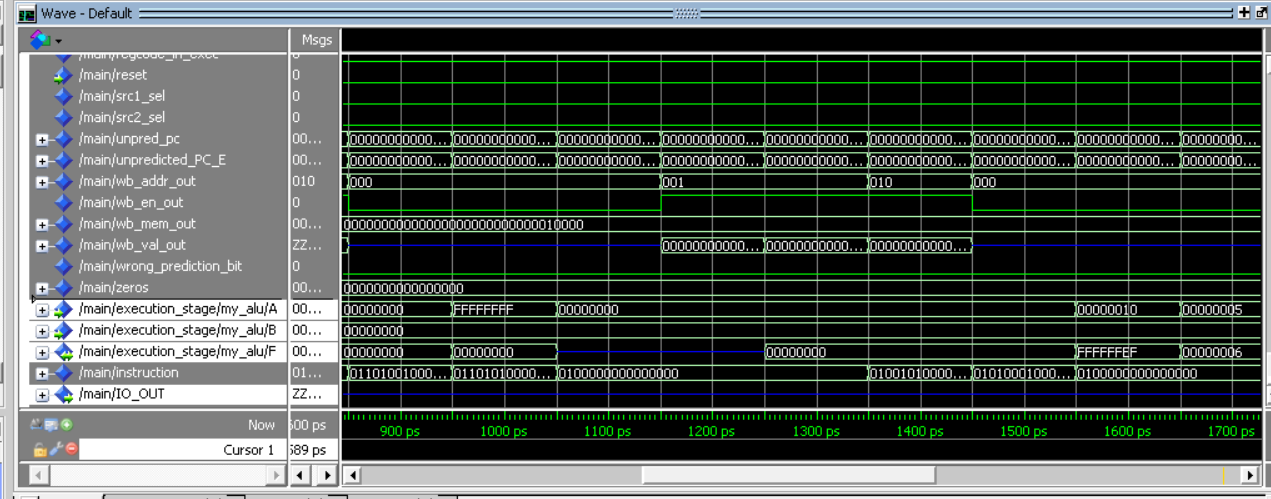
# 2.Analysis:

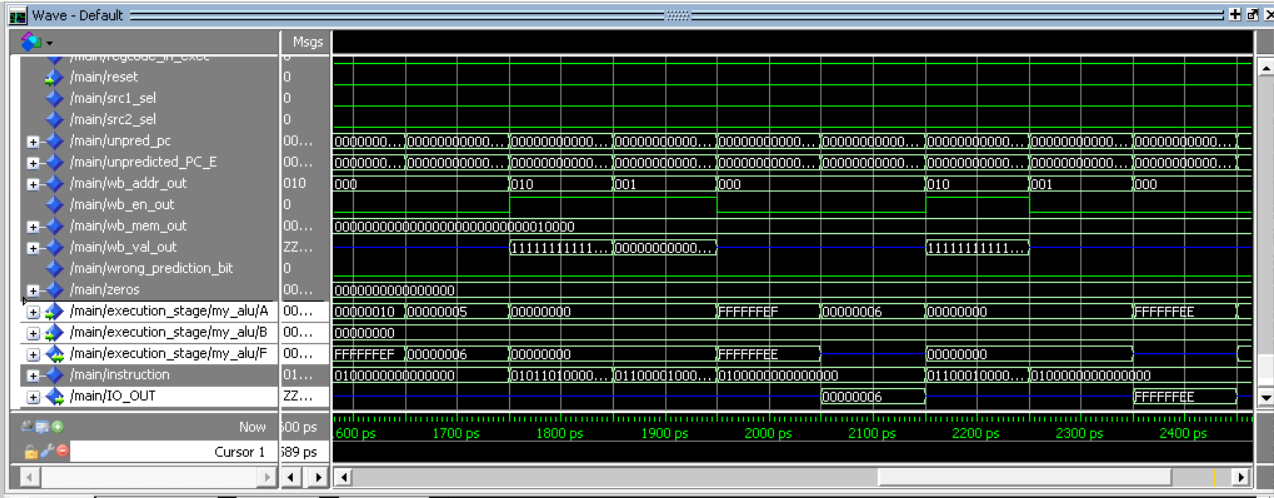
## 1. One-op testcase

Without control hazard unit and forwarding unit:

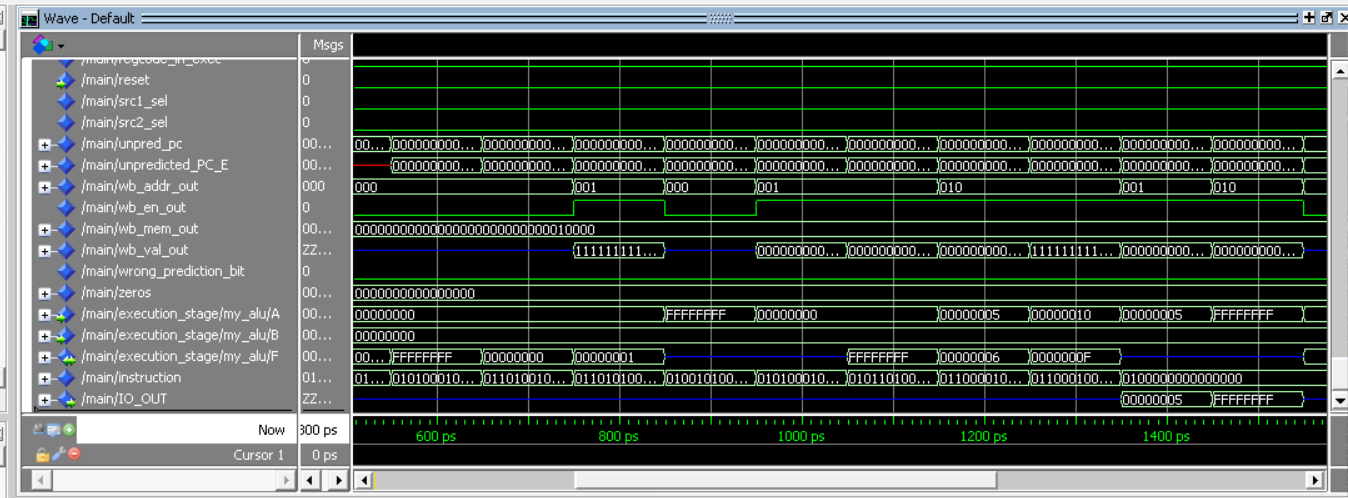
Number of clock cycle = 26

Hazards:





Without control hazard unit and forwarding after add NOP to solve hazards



Without control hazard unit and forwarding before add NOP to solve hazards

Note: there is 4 data hazards

* Red: When “inc R1” enter execution stage, R1 value has not yet been updated, so it increases “00000000” instead of “FFFFFFFF”
* Blue: When “NOT R2” enter execution stage, R2 value has not yet been updated, so it inverses “00000000” instead of “00000010”
* White: When “Dec R2” enter execution stage, R2 value has not yet been updated, so it decreases “00000010” instead of “FFFFFFEF”
* Yellow: When “out R2” enter execution stage, R2 value has not yet been updated, so it out “FFFFFFFF” instead of “FFFFFFEE”

Without forwarding unit:

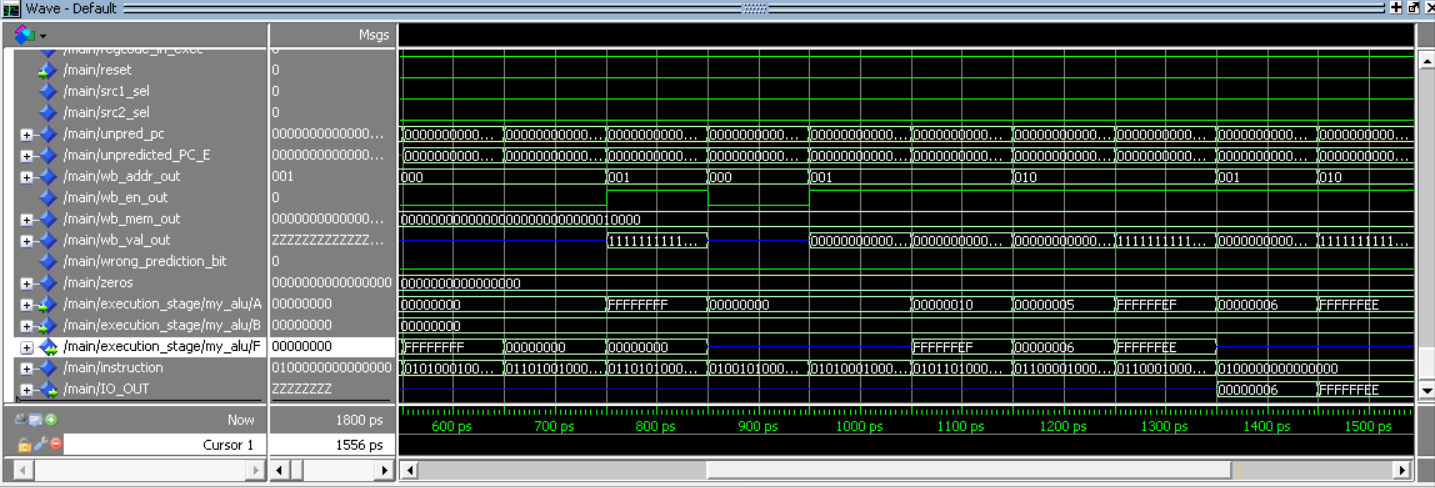
Same as without control hazard unit and forwarding unit

Without control hazard unit:

Number of clock cycle = 17

Hazards: non

With control hazard unit and forwarding unit:



With control hazard unit and forwarding

Number of clock cycle = 17

Hazards: non

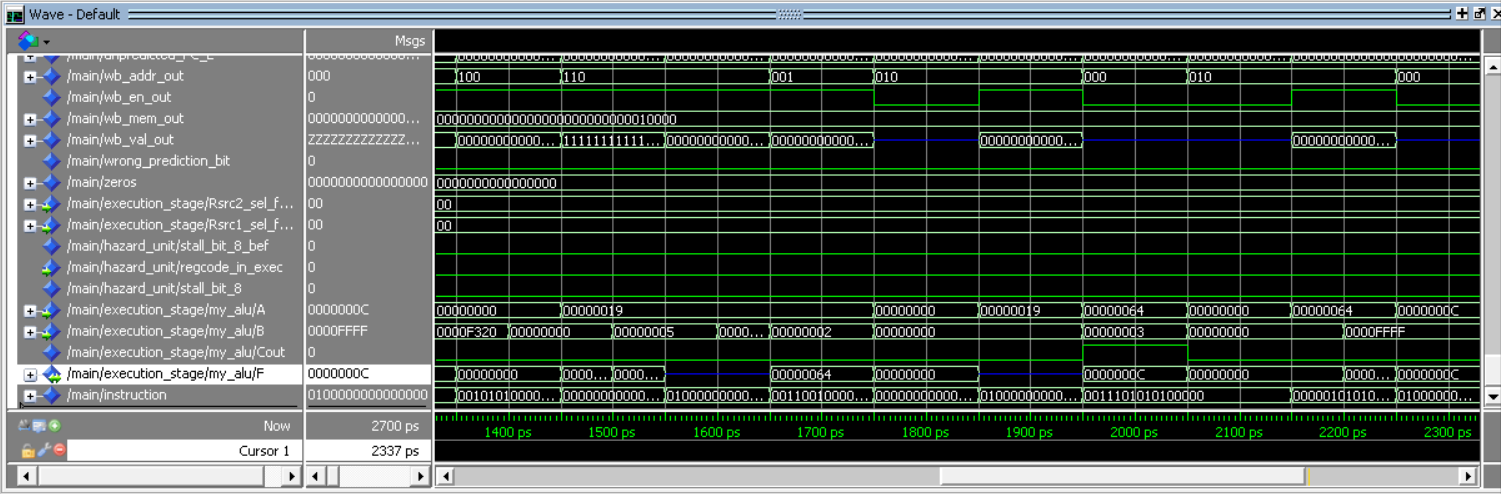
## 2. Two-op testcase

Two operand test case

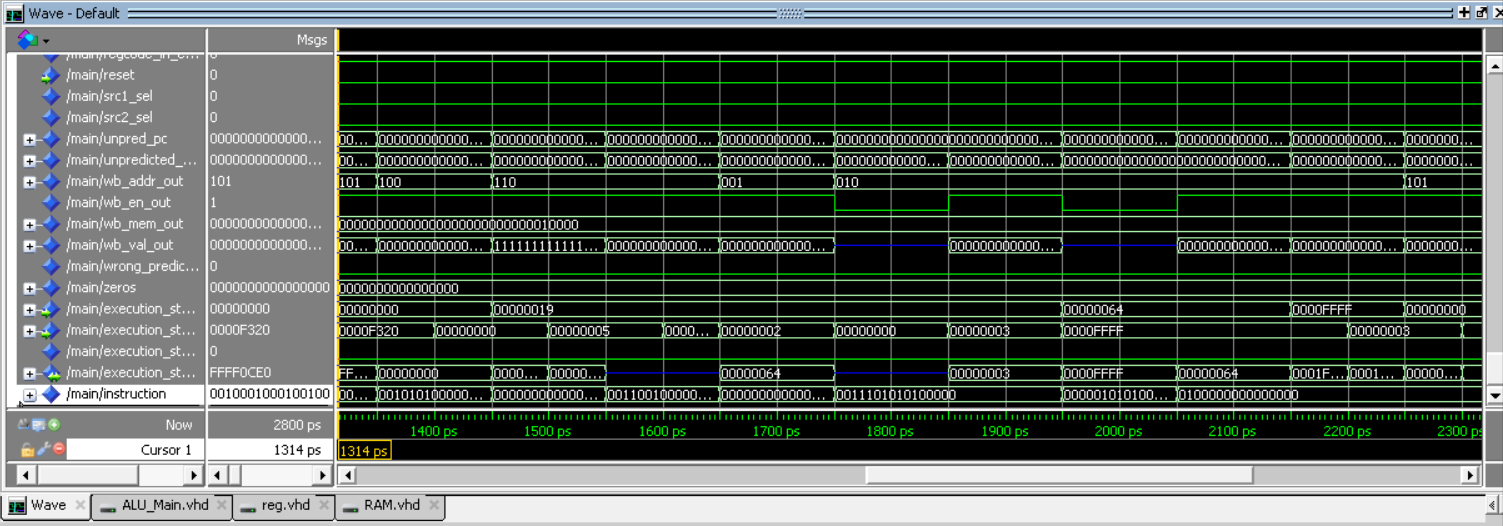
Without control hazard unit and forwarding unit:

Number of clock cycle = 27

Hazards:



Without control hazard unit and forwarding after add NOP to solve hazards



Without control hazard unit and forwarding before add NOP to solve hazards

Note: there is 2 data hazards

* Red: When “SHR R2,3” enter execution stage, R2 value has not yet been updated, so it shifts “00000019” instead of “00000064”
* Yellow: When “SWAP R2,R5” enter execution stage, R2 value has not yet been updated, so it swaps “0000FFFF” with “00000064” instead of “0000FFFF” with “0000000C”

Without forwarding unit:

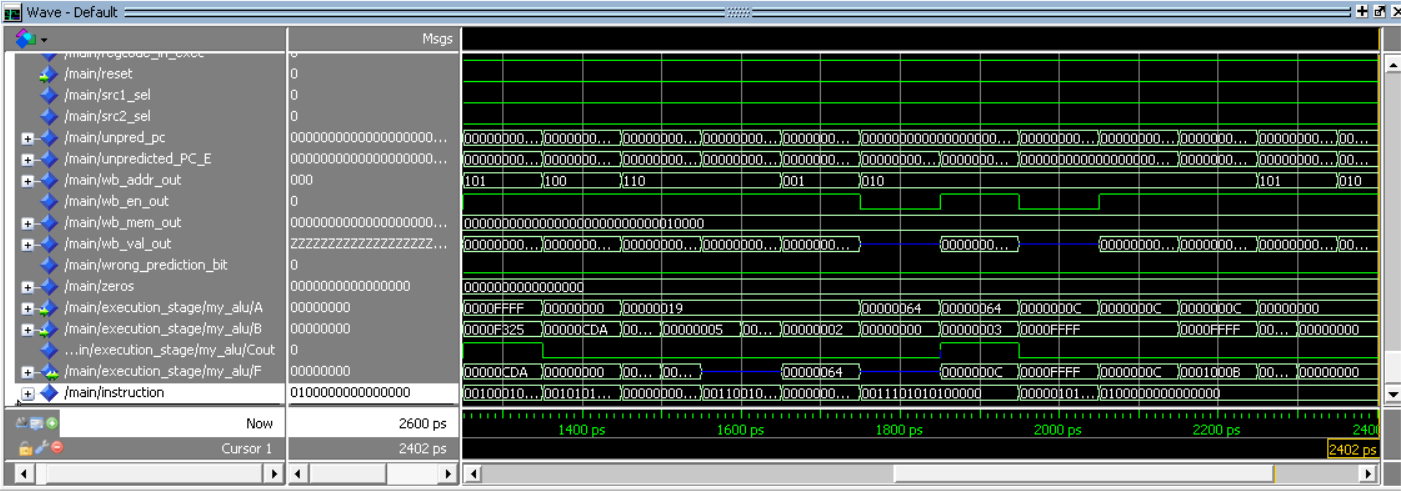
Same as without control hazard unit and forwarding unit

Without control hazard unit:

Number of clock cycle = 25

Hazards: non

With control hazard unit and forwarding unit:



With control hazard unit and forwarding

Number of clock cycle = 25

Hazards: non

## 3. Branch testcase

* (Fixed Assembly file by adding nop’s of each incremental step is attached)
* Hazards in the testcase:
  1. JMP R1 - Control Hazard
  2. AND R1, R5, R5 - Control Hazard Because of the Interrupt
  3. JZ R2 - Control Hazard
  4. JZ R3 - Control Hazard
  5. NOT R5

INC R5 - Data Hazard

* 1. In R6

JZ R6 - Data and Control Hazard

* 1. RTI - Control Hazard
  2. Call R6 - Control Hazard because of the interrupt
  3. RET - Control Hazard

1. Without Forwarding and Hazard Detection Unit: (71 cycles using nops)
   * All Hazards were present:

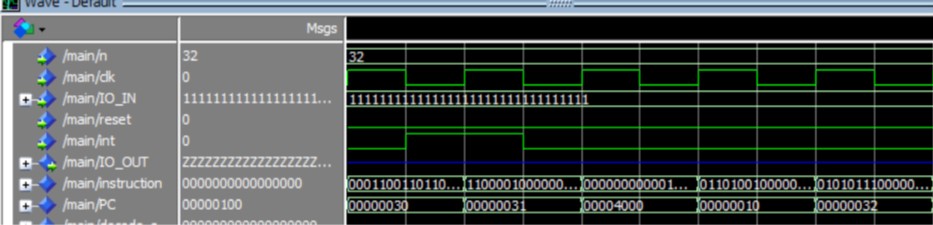


Figure 1: undefined behaviour on interrupt at 30 and instruction at 32 was fetched and excuted

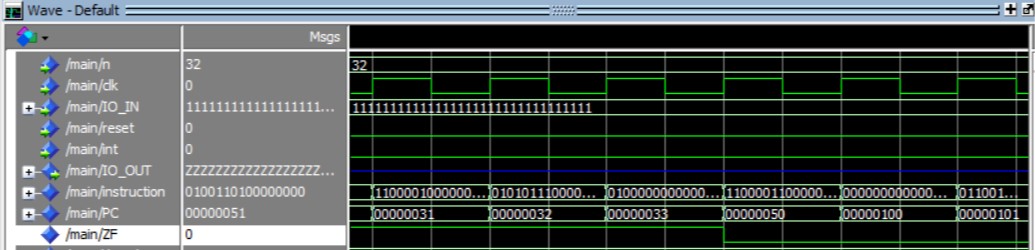


Figure 2: JZ at 31 was non-taken and the JZ at 50 was taken. and wrongly fetched instructions wasn't flushed

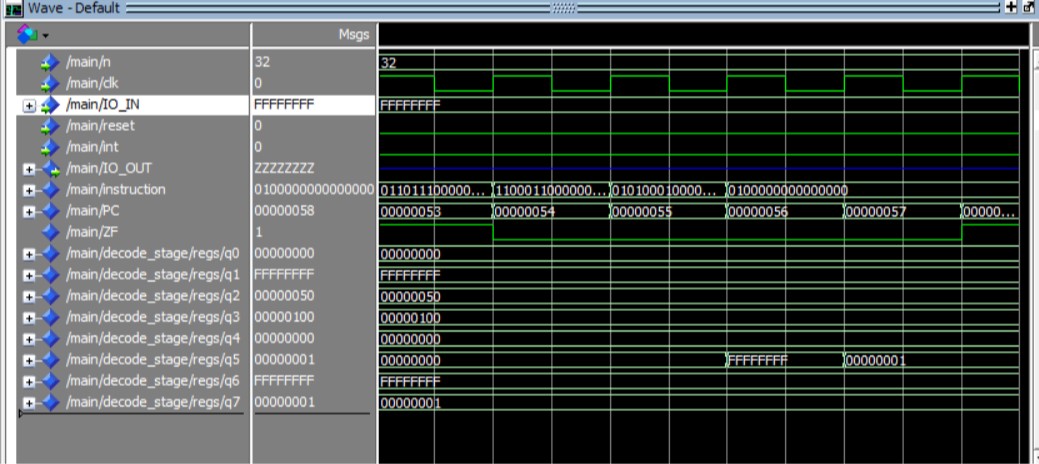


Figure 3: Data Hazard at R5 where an old value was used

* + To solve them using no operation:
    1. The JMP at 1, JZ at 3 and 4: add two nop after them to allow the new address calculation to take place without fetching wrong instructions
    2. Interrupt at 2 and 8: add 4 nop after them because the interrupts happen on 2 cycles and the second cycle is like a load-use case.
    3. The RTI at 7 and ret at 9: add 3 nop after them because it’s a load-use case where the PC is loaded from the memory
    4. The data Hazards at 5 and 6: add two nop between the two instructions
    5. The control hazard at 6: add two nop after the jump to allow the new address calculation to take place without fetching wrong instructions

1. With Forwarding Unit but no Hazard Detection: (69 cycles using nops)
   * All the hazards are still present except: Data Hazard at the INC R5 at 5 is solved using forwarding:

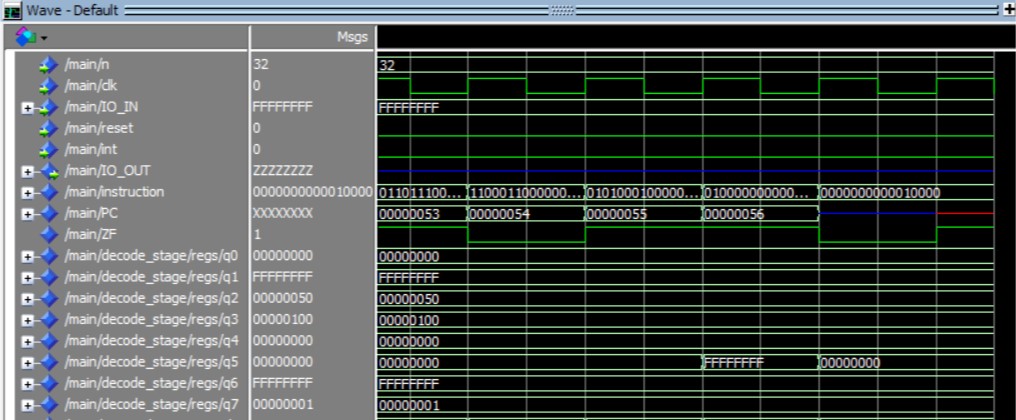


Figure 4: Data Hazard Solved using Forwarding unit

* + Data Hazard at 6 is not solved because branch unit is the one using the R6 value not the alu.

1. With Hazard Detection Unit but without the PC prediction: (67 cycles using nops)
   * The Data Hazard at 6 is handled by the stalling of the hazard unit.
   * The rest of the Hazards are still present.
2. With everything on: (59 cycles using nops)
   * All the Control Hazards are solved using branch prediction and flushing:

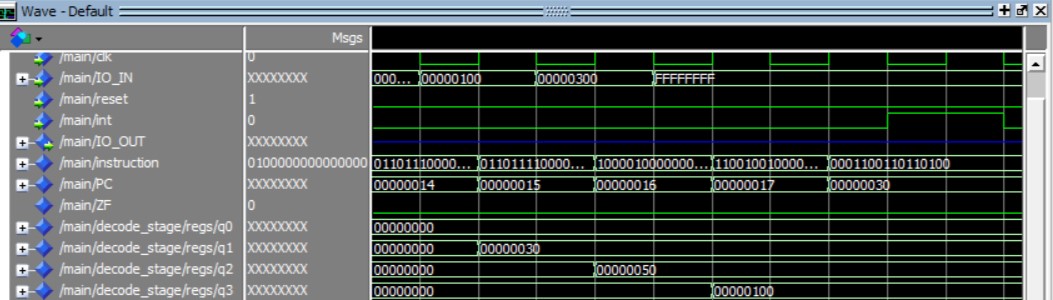


Figure 5: the control hazard of JMP at 17

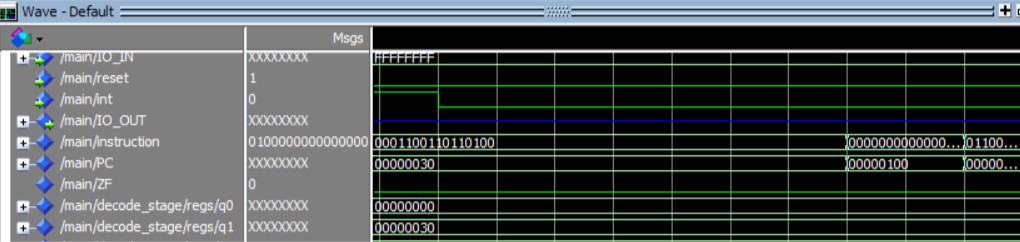


Figure 6: the control hazard of the interrupt handled by hardware stalling

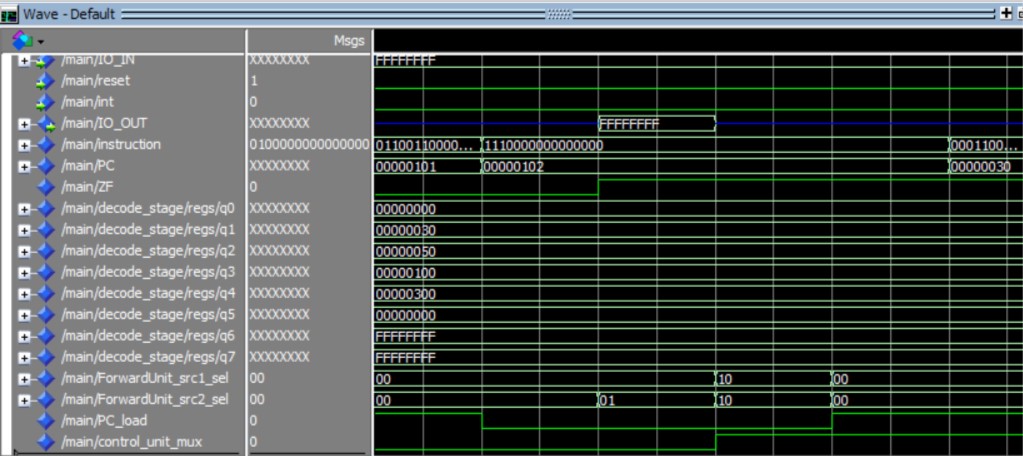


Figure 7: the control hazard at 102

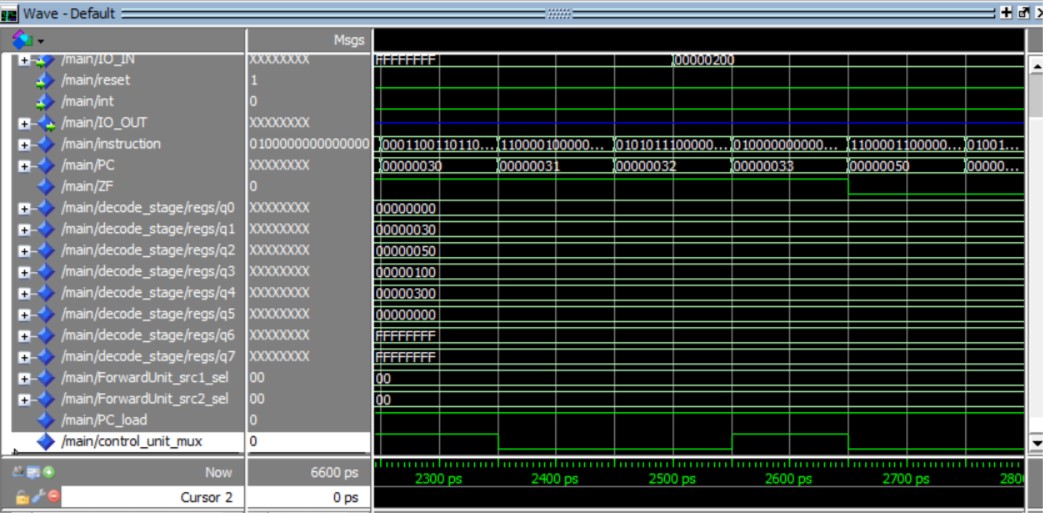


Figure 8: the instruction 32 flushed by "control\_unit\_mux" signal

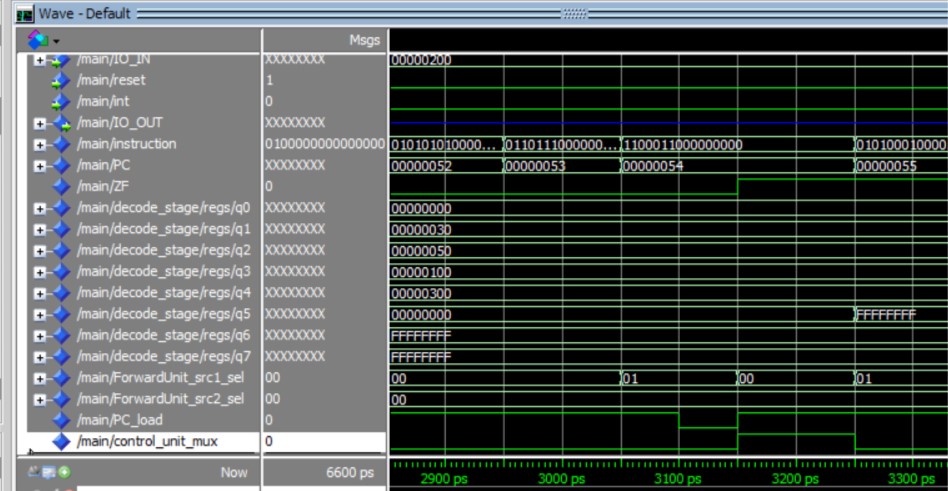


Figure 9: the control hazard at 54 JZ R6 by hardware stall

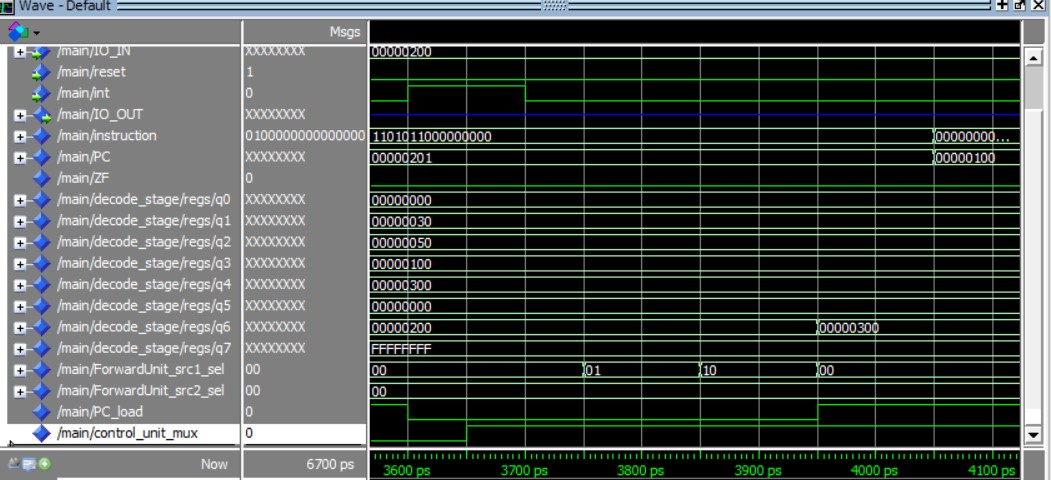


Figure 10: the interrupt signal at 201 handled using hardware stall

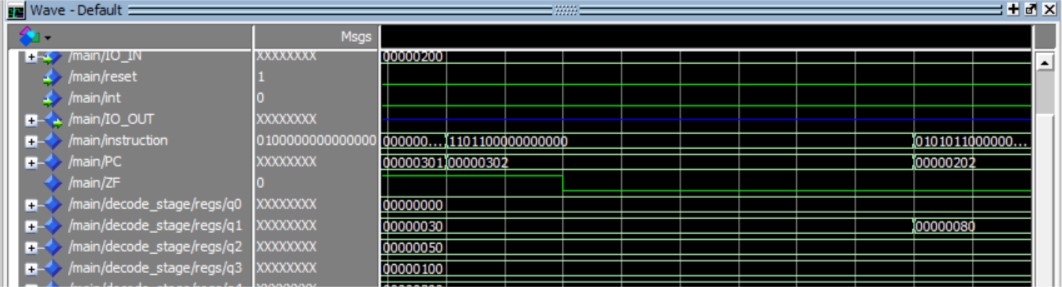


Figure 11: the control hazard at 302 handled using hardware stalls making sure the INC R7 is not executed.

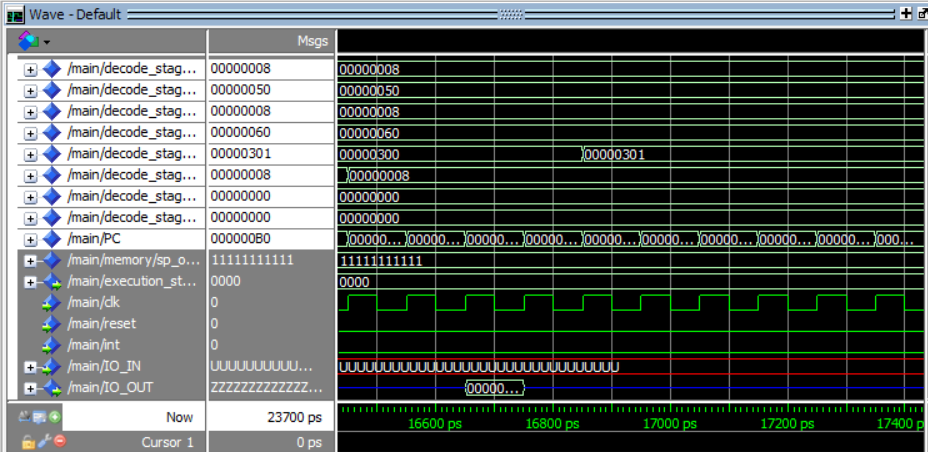
## 4. Branch prediction testcase

Times of end cycles

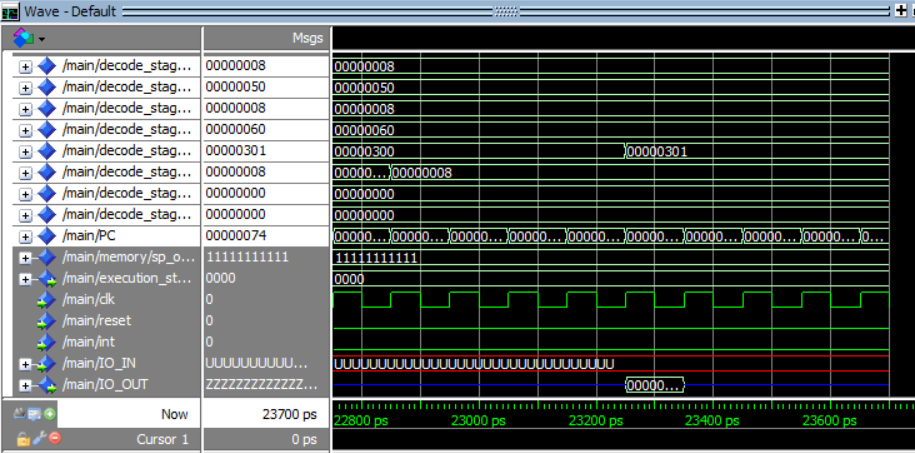
* with forwarding only: 16900
* without forwarding & hazard detection & flushing: 23300
* with all units working: 13700
* with forwarding and hazard detection: 16900

Waveforms:

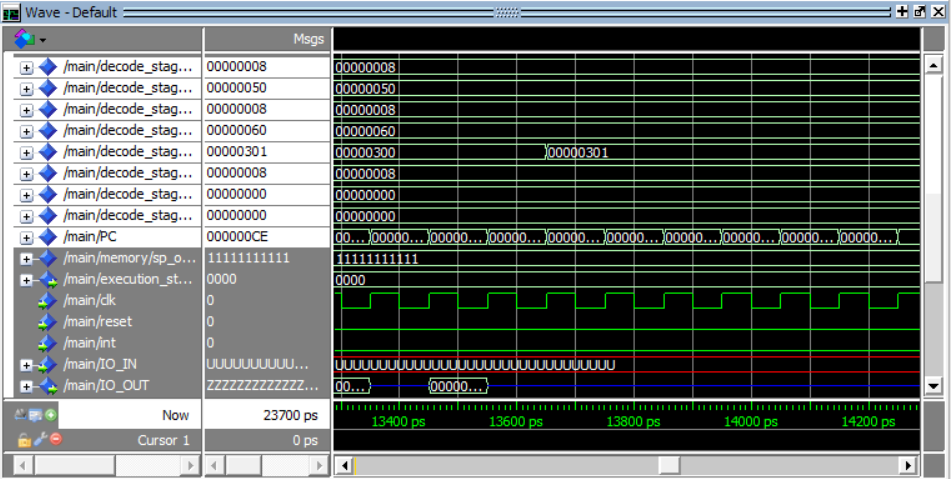
1. with forwarding only



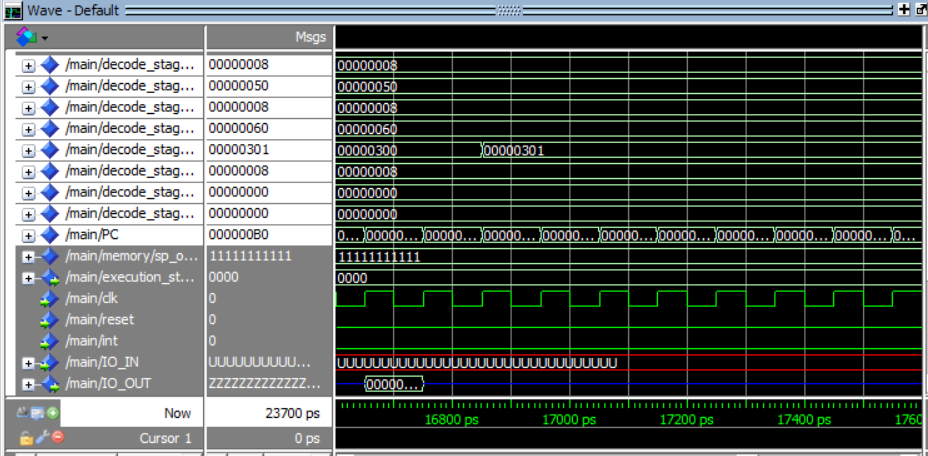
1. without forwarding & hazard detection & flushing



1. with all units working



1. with forwarding and hazard detection



Comments:

1. No data hazard with forwarding because ldm loads imm values not from memory so the forwarding unit will forward it.
2. Both with forwarding only and with forwarding and hazard detection got the same number of cycles needed to finish because in this file forwarding unit made that there’s no data hazards at all.

Hazards happened:

* Disabling flushing and always predict not taken caused control hazard after each jmp/jz/call instructions and to solve it we added 2 NOPs after each of them

Instructions got the hazard:

JMP R3

JZ R1

JMP R3

JMP R3

JZ R3

Check the testcase analysis code

* Disabling hazard detection unit caused no hazards
* Disabling both hazard detection unit and forwarding caused data hazards and to solve it we added NOPs before each instruction causing hazard to make sure that while it’s in decode stage the data are ready in write back stage

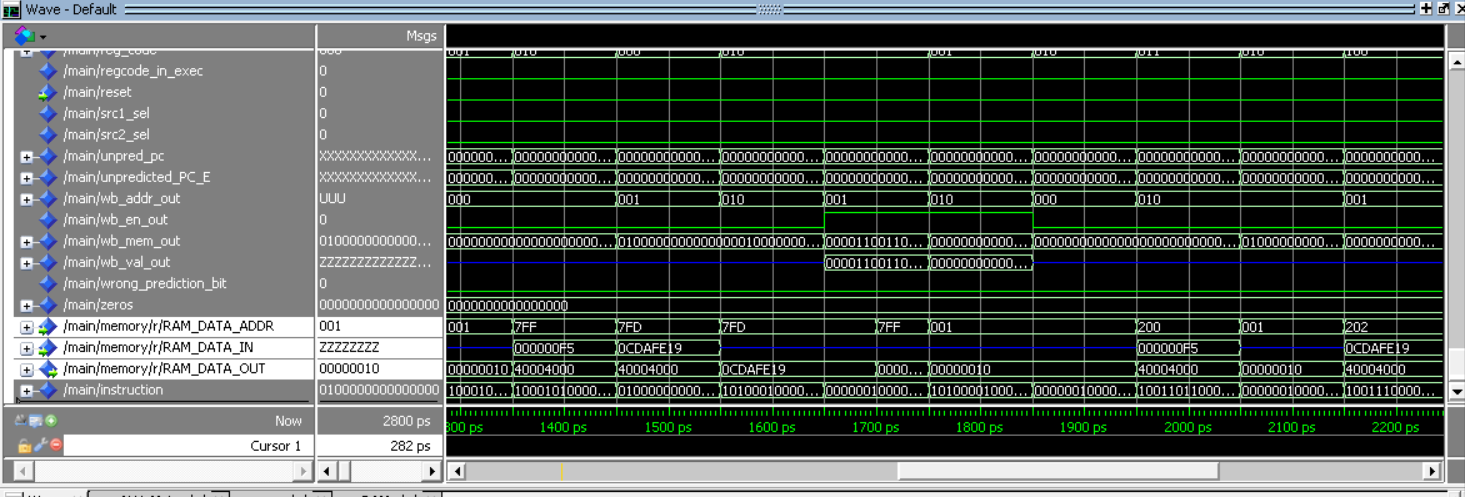
|  |  |
| --- | --- |
| Instructions got the hazards | Number of NOPs needed before it |
| JMP R3 | 1 |
| OUT R4 | 2 |
| JMP R3 | 1 |
| OUT R4 | 2 |
| AND R0,R2,R5 | 2 |
| OUT R4 | 2 |

## 5. Memory testcase

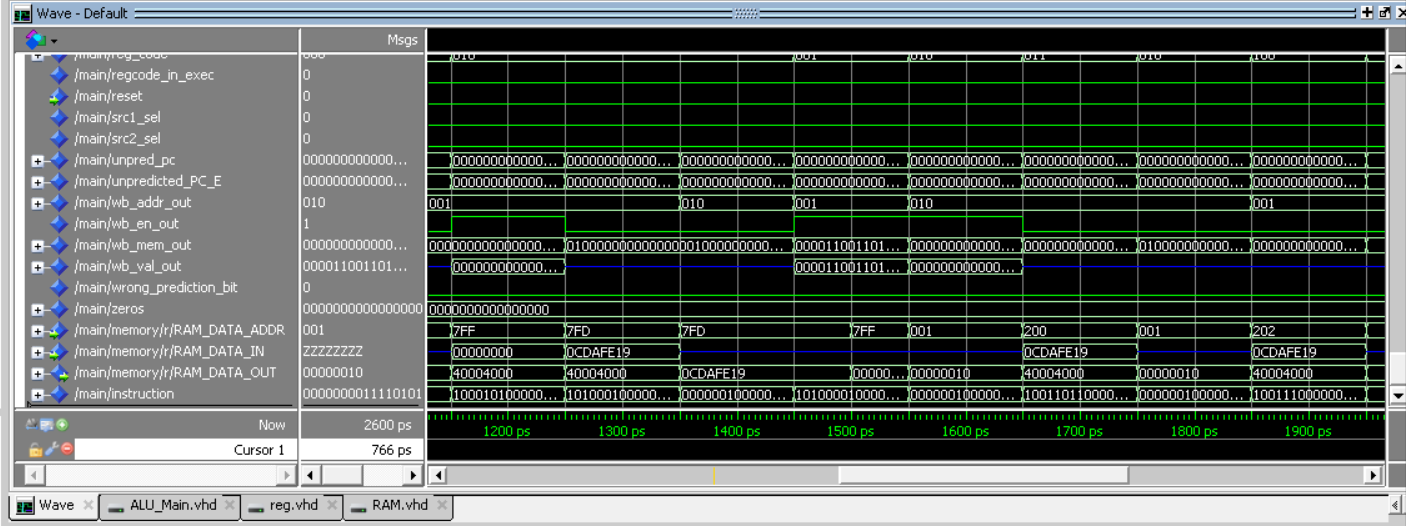
Without control hazard unit and forwarding unit:

Number of clock cycle = 28

Hazards:



Without control hazard unit and forwarding after add NOP to solve hazards



Without control hazard unit and forwarding before add NOP to solve hazards

Note: there is 2 data hazards

* Red: When “PUSH R1” enter execution stage, R1 value has not yet been updated, so it pushes “00000000” instead of “000000F5”
* White: When “STD R2,200” enter execution stage, R2 value has not yet been updated, so it stores “0CDAFE19” instead of “000000F5”

Without forwarding unit:

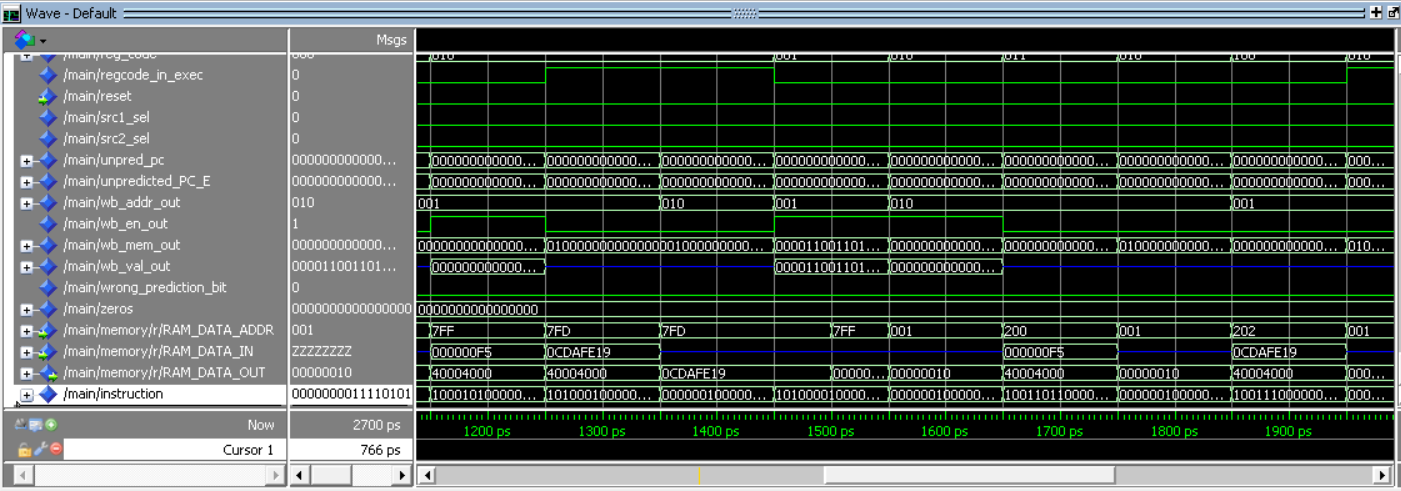
Same as without control hazard unit and forwarding unit

Without control hazard unit:

Number of clock cycle = 25

Hazards: non

With control hazard unit and forwarding unit:



With control hazard unit and forwarding

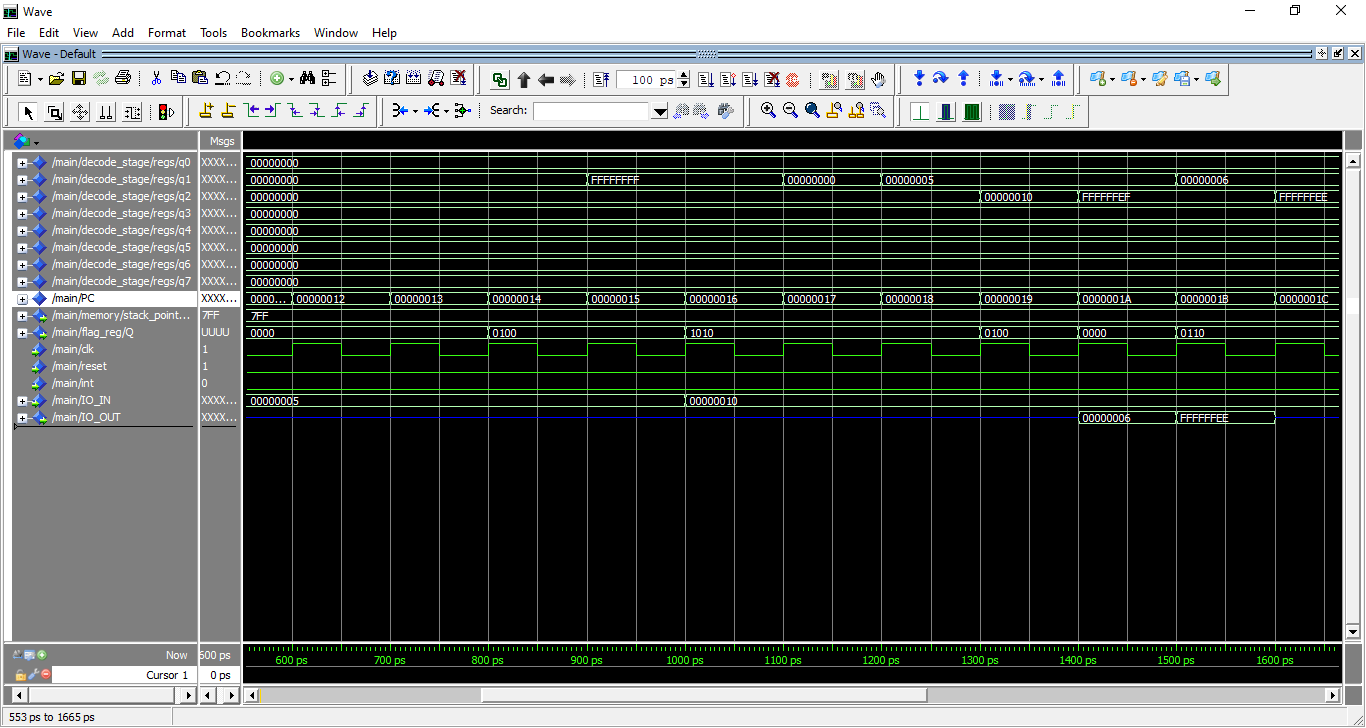
Number of clock cycle = 25

Hazards: non

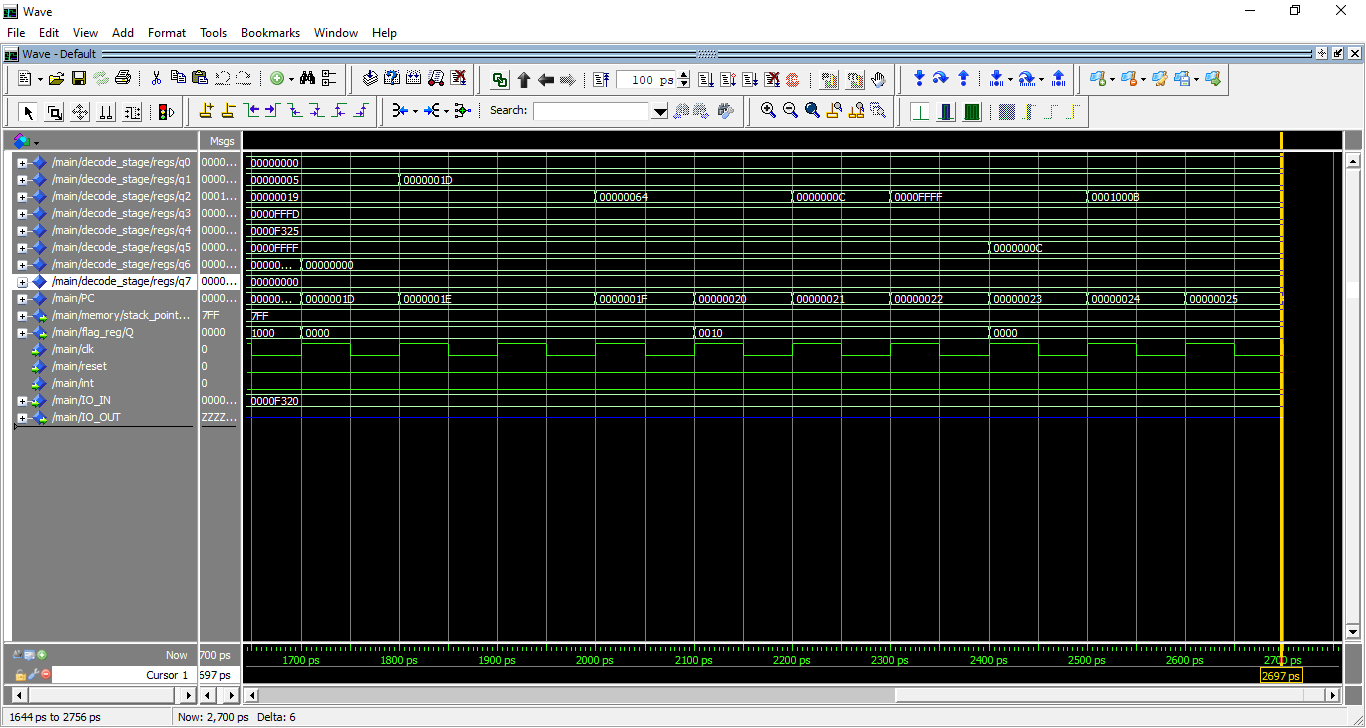
# Project Testing

**All do files included in folder “do-files”**

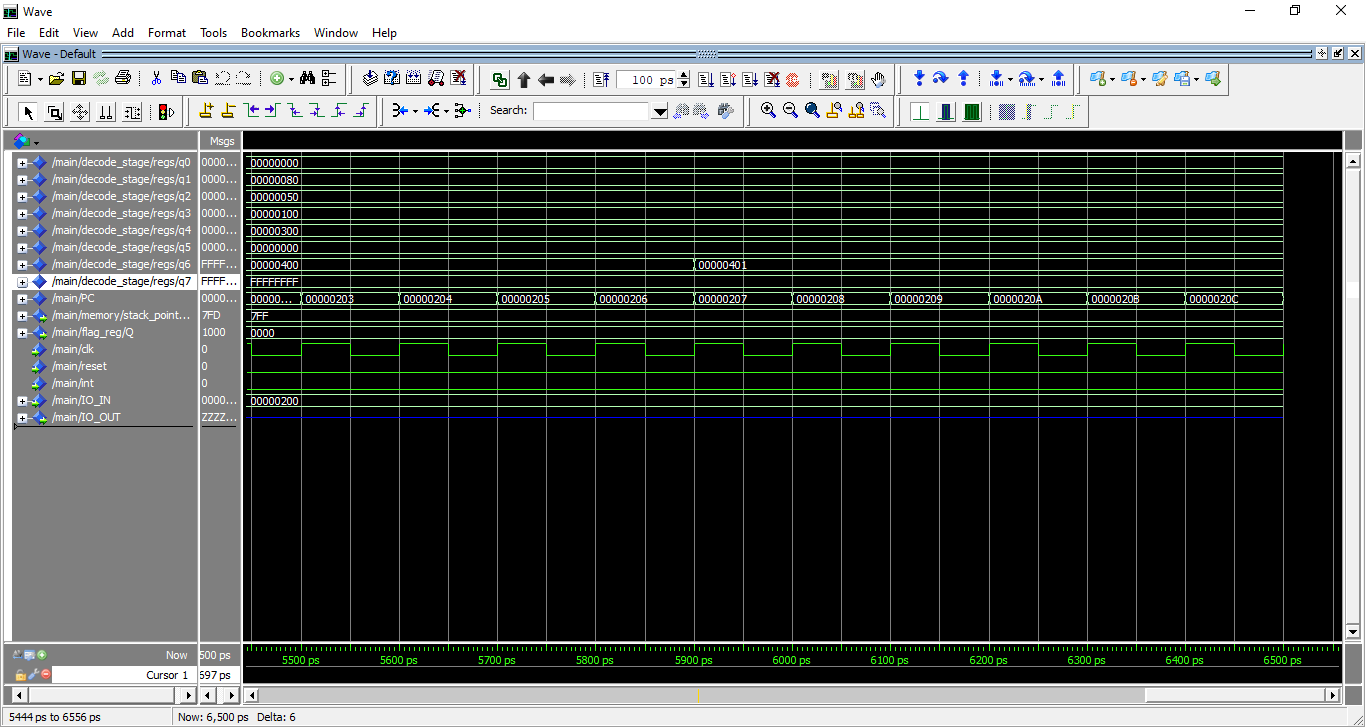
Test case 1: one operand: (do file name: oneOperand\_dofile.txt, included in Project Testing folder)



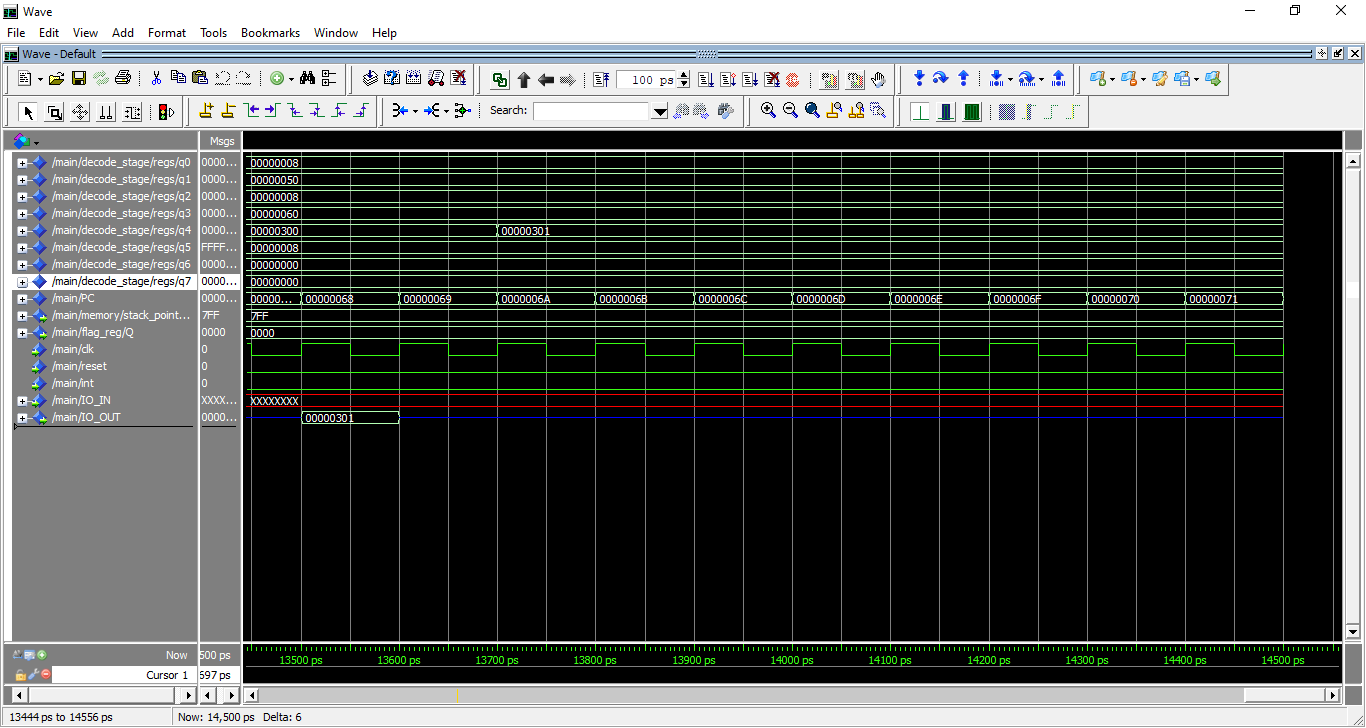
Test case 2: two operands: (do file name: twoOperands\_dofile.txt, included in Project Testing folder)



Test case 3: branch:



Test case 4: branch prediction:



Test case 5: memory:

