Credit Hours System
CMPN301(Spring 2020)

Cairo University
Faculty of Engineering

Computer Architecture

Final Assessment

Report

Submitted to: Dr. Mayada Hadhoud

Member Name Team 4	Member ID
Kamel Mohsen Kamel	1162325
Ahmad Nader Adel	1162296
Eman Gamal ElBadawy	1170298
Eman Mohamed Saleh	1170297

Table Of Contents

Abstract

This report is prepared to discuss the steps and results of implementing a Harvard-Architecture processor in VHDL, in addition to the processor VHDL implementation there is a python based Assembler that assembles the set of assembly instructions to its equivalent hex – code to be inserted to the memory.

The processor includes Full Forwarding, Hazard Detection Unit, and Static Branch Prediction (assuming not taken always). The processor is made on a limited set of instructions that are mentioned in the Instruction part of the report, and the assembler generates two ".mem": "assemblyfilenameData.mem" and "assemblyfilenameInstruction.mem".

Introduction

The Harvard-Architecture processor is based upon having two separate memories: one for Memory & Stack Data and the other is for Instructions, thus eliminating the Structural Hazards resulting from having a shared memory for both Data and Instructions. The processor mentioned in this report is a Five Stage pipelined processor which increases the CPU instruction throughput - the number of instructions completed per unit of time. But it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the pipeline control. The types of hazards that are faced in this type of processors is control hazards, data-hazards, and data-hazards causes a load-case hazard when a memory data is needed to be written to a register before directly using that register, This report will briefly explain how this hazards where handled and what would be the results if they were not handled using the ModelSim application to generate a wave-form representing the results of the processor and initiating its inputs.

PS: The Processor was supposed to contain and additional part to support memory cache, but MEMORY CACHE WAS NOT IMPLEMENTED due to limited time restrictions

ISA Specifications

The processor in this project has a RISC-like instruction set architecture. There are eight 4-byte general purpose registers; R0, till R7. Another two general purpose registers, one works as program counter (PC). And the other, works as a stack pointer (SP); and; hence, points to the top of the stack. The initial value of SP is (**2^11-1**). The memory address space is **4 KB of 16-bit** width and is word addressable. **The bus between memory and the processor is (32-bit)**

When an interrupt occurs, the processor finishes the currently fetched instructions (instructions that have already entered the pipeline), then the address of the next instruction (in PC) is saved on top of the stack, and PC is loaded from address [2-3] of the memory (the address takes two words). To return from an interrupt, an RTI instruction loads PC from the top of stack, and the flow of the program resumes from the instruction after the interrupted instruction.

Specifications

Registers

R[0:7]<31:0>	;Eight 32-bit general purpose registers
PC<31:0>	; 32-bit program counter
SP<31:0>	; 32-bit stack pointer
CCR<3:0>	; condition code register
Z<0>:=CCR<0>	; zero flag, change after arithmetic, logical, or shift operations
N<0>:=CCR<1>	; negative flag, change after arithmetic, logical, or shift operations
C<0>:=CCR<2>	; carry flag, change after arithmetic or shift operations.

Input-Output

IN.PORT<31:0>	; 32-bit data input port
OUT.PORT<31:0>	; 32-bit data output port
INTR.IN<0>	; a single, non-mask able interrupt
RESET.IN<0>	; reset signal
Rsrc1	; 1st operand register
Rsrc2	; 2nd operand register
Rdst	; result register

EA ; Effective address (7 bit)
Imm ; Immediate Value (16 bit)
SHFT ; Immediate Value (5 bit)

Instructions

Mnemonic	Function
One Operand	
NOP	PC ← PC + 1
SETC	C ←1
CLRC	C ←0

NOT Rdst	NOT value stored in register Rdst
	R[Rdst] ← 1's Complement(R[Rdst]);
	If (1's Complement(R[Rdst]) = 0): Z ←1; else: Z ←0;
	If (1's Complement(R[Rdst]) < 0): N ←1; else: N ←0

INC Rdst	Increment value stored in Rdst
	R[Rdst] ←R[Rdst] + 1;
	If ((R[Rdst] + 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rdst] + 1) < 0): N ←1; else: N ←0

DEC Rdst	Decrement value stored in Rdst
	R[Rdst] ←R[Rdst] – 1;
	If ((R[Rdst] – 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rdst] – 1) < 0): N ←1; else: N ←0

OUT Rdst	OUT.PORT ← R[Rdst]
IN Rdst	R[Rdst] ←IN.PORT
Two Operands	
SWAP Rsrc, Rdst	Store the value of Rsrc 1 in Rdst and the value of Rdst in Rsc1 flag shouldn't change
ADD Rsrc1, Rsrc2, Rdst	Add the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0
IADD Rsrc1,Rdst,Imm	Add the values stored in registers Rsrc1 to Immediate Value and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0

SUB Rsrc1, Rsrc2, Rdst	Subtract the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0
AND Rsrc1, Rsrc2, Rdst	AND the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0
OR Rsrc1, Rsrc2, Rdst	OR the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0
SHL Rdst, SHFT	Shift left Rsrc by SHFT value and store result in same register
SHR Rdst, SHFT	Shift right Rsrc by SHFT value and store result in same register
Memory Operations	
PUSH Rdst	M[SP--] ← R[Rdst];
POP Rdst	R[Rdst] ← M[++SP];
LDM Rdst, Imm	Load immediate value (16 bit) to register Rdst R[Rdst] ← {0,Imm<15:0>}
LDD Rdst, EA	Load value from memory address EA to register Rdst R[Rdst] ← M[EA];
STD Rdst, EA	Store value in register Rsrc to memory location EA M[EA] ←R[Rsrc];
Branch and Change of Control Operations	
JZ Rdst	Jump if zero If (Z=1): PC ←R[Rdst]; (Z=0)
JN Rdst	Jump if negative If (N=1): PC ←R[Rdst]; (N=0)
JC Rdst	Jump if negative If (C=1): PC ←R[Rdst]; (C=0)
JMP Rdst	Jump PC ←R[Rdst]
CALL Rdst	(M[SP] ← PC + 1; sp-2; PC ← R[Rdst])
RET	sp+2, PC ←M[SP]
RTI	sp+2; PC ← M[SP]; Flags restored

Input Signals	Function
Reset	PC ←{M[1], M[0]}
Interrupt	M[Sp]←PC; sp-2;PC ← {M[3],M[2]}; Flags preserved

Project Content

Assembler Directory:

1- Assembler.exe : This is an executable file that runs the assembler code and generates the output to same directory of the input with the same names i.e : if file Test.asm is selected then the output will be a TestData.mem and TestInstruction.mem files generated in the same directory of Test.asm
2- Assembler.py : This is the source code of the assembler in python language

Bank_of_Buffers Directory: Contains the .vhd files for the intermediates bank of buffers

Decoder Directory: Contains the .vhd files for the deocde stage

Design _Opcodes Directory: Contains

1- CMPN301 _Final_Assessment_Spring2020.pdf : A pdf version of the latest given project description
2- DesignRemade.drawio : The draw.io website generated file for the design
3- DesignRemade.pdf : The draw.io website generated pdf file for the design
4- OPCODES.xlsx : An Excel file containing the whole opcodes used in every unit in the project (minor and major units)

Executing Directory: Contains the .vhd files for the executing stage

Fetcher Directory: Contains the .vhd files for the fetching stage

Integrated Directory: Contains the .vhd files for the integrated Processor

Memory Directory: Contains the .vhd files for the Memory and Write back Stage

Report Directory: Contains a

1- Pdf version of the final report
2- Word version of the final report

Test_Cases Directory: Contains all the test cases in subdirectories with their .mem files already compiled and their .do files

Running Instructions

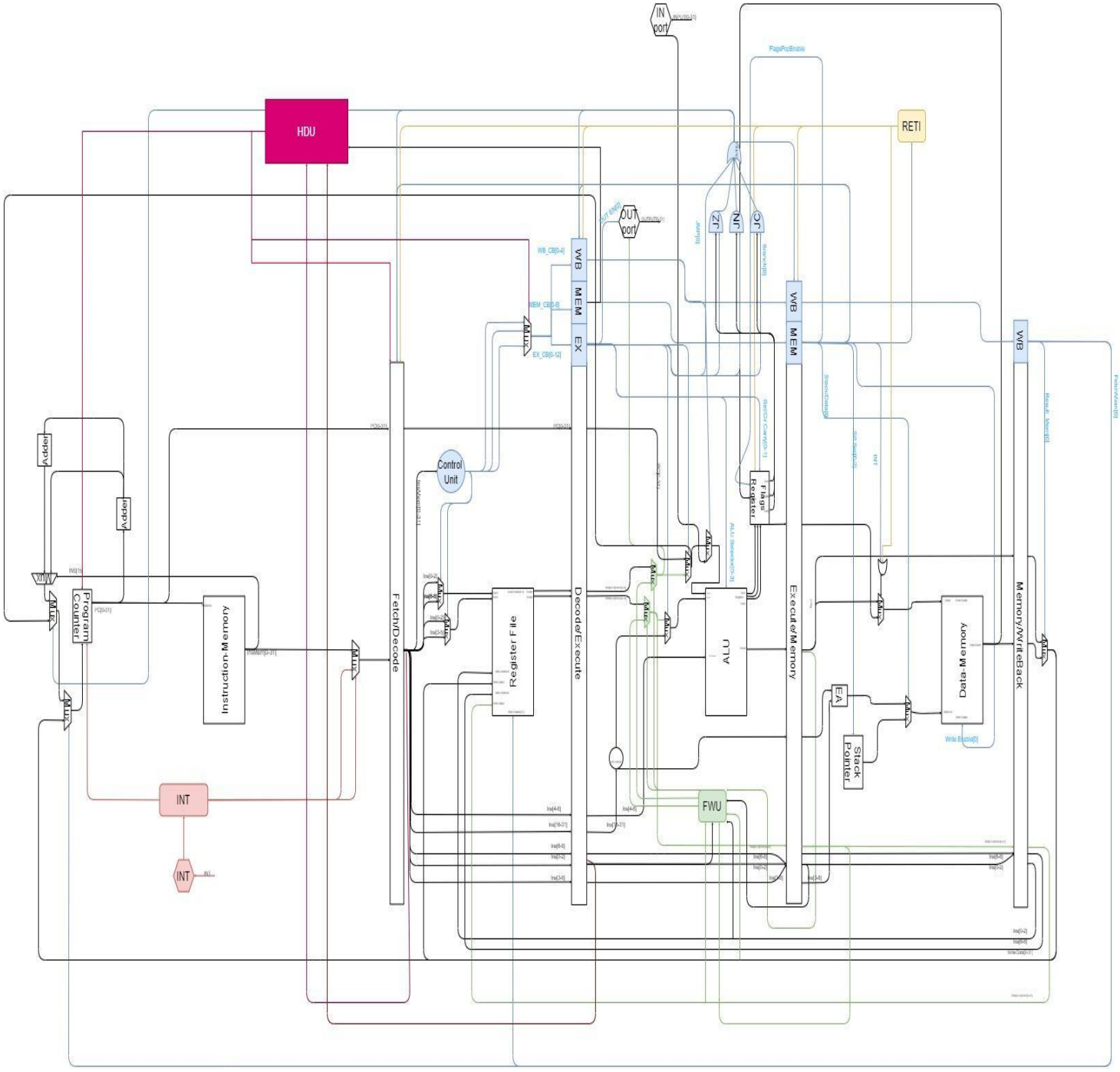
To run the processor follow this steps:

1- Write your assembly code using the above set of instruction
2- Compile them using Assembler.exe
3- The write their .do files (you can use the following as a template)**Don't forget to change the <> as mentioned below

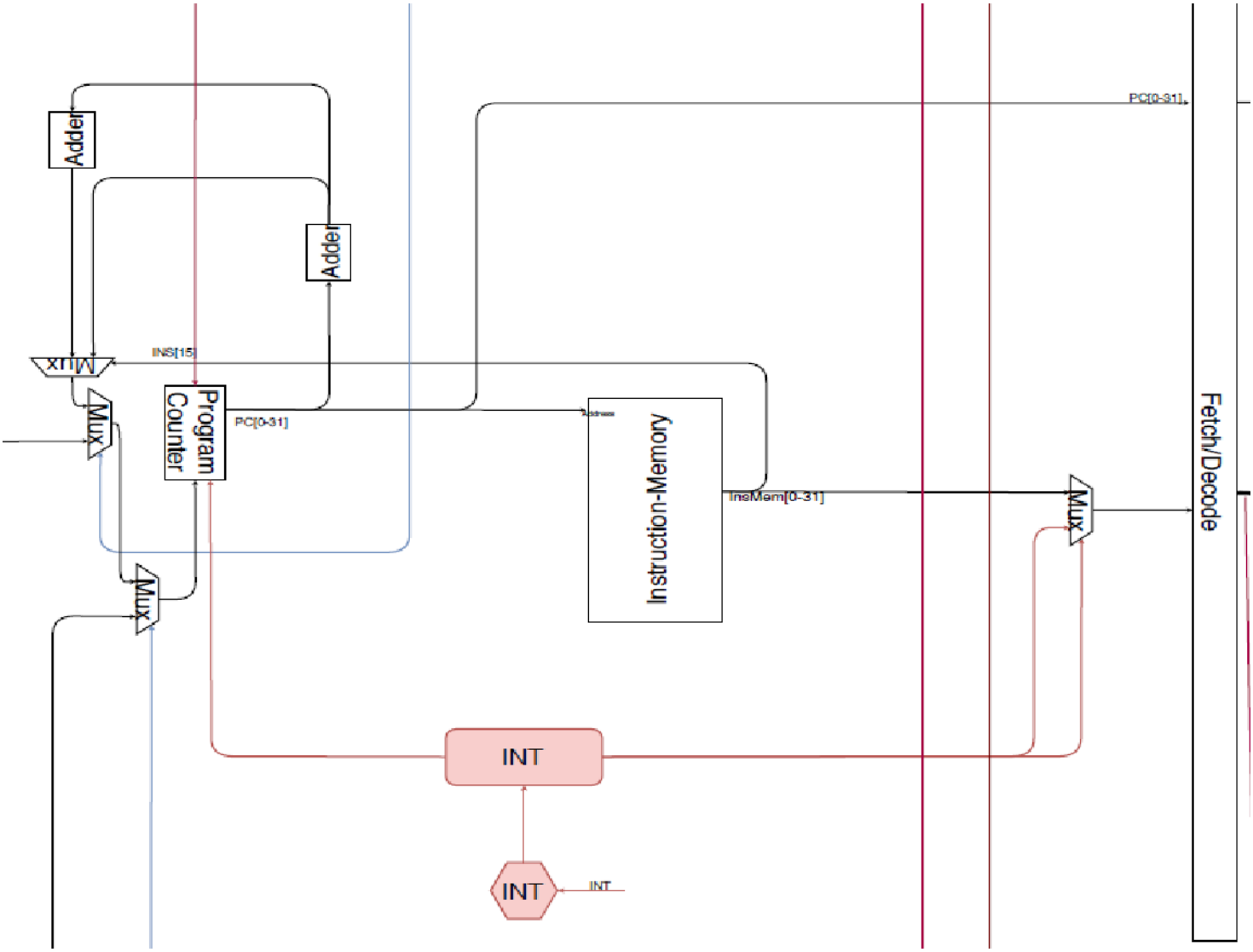
```
vsim -gui work.harvard_processor
add wave -position insertpoint  \
sim:/harvard_processor/INT_SIGNAL \
sim:/harvard_processor/RESET \
sim:/harvard_processor/CLK \
sim:/harvard_processor/IN_PORT \
sim:/harvard_processor/OUT_PORT \
sim:/harvard_processor/REG_0_TEST \
sim:/harvard_processor/REG_1_TEST \
sim:/harvard_processor/REG_2_TEST \
sim:/harvard_processor/REG_3_TEST \
sim:/harvard_processor/CARRY_FLAG \
sim:/harvard_processor/NEGATIVE_FLAG \
sim:/harvard_processor/ZERO_FLAG
force -freeze sim:/harvard_processor/RESET 1 0
force -freeze sim:/harvard_processor/INT_SIGNAL 0 0
force -freeze sim:/harvard_processor/CLK 1 0, 0 {50 ns} -r 100
mem load -
i {<Instruction.mem_Directory_Path>/***Data.mem} /harvard_processor/MEMORY_UNITT/Memory/ram
mem load -
i {<Instruction.mem_Directory_Path>/***Instruction.mem} /harvard_processor/FETCHING_UNIT/INST_MEM/ra
m
run
force -freeze sim:/harvard_processor/RESET 0 0
run
```

Design

Full Design



Fetching Stage



The above picture is in depth detailed design of the fetching unit implemented in the processor.

1- **Program Counter**: a register that hold the current address of the next instruction to be fetched, "Mux C" is the Input that this unit is supposed to store, and it is directly fetched out to the out wire.

2- **Adder A**: adder that is directly after the PC that takes the current PC value and adds 1 to it to get the next value.

3- **Adder B**: adder that adds another 1 to the PC+1 that results from "Adder A" this is to handle the different instruction sizes since some instructions are 16 bit long and other instructions are 32 bit long.

4- **Mux A**: 2x1 mux which selects between "Adder A" & "Adder B" based on the selector which is "INSTRUCTION[15]".

5- **Mux B**: 2x1 mux which selects between "Mux A" & "Jump Location" coming from Execute stage based on the selector "Jump Bit" which also comes from Execution stage in case of a JMP case or successful Branching.

6- **Mux C**: 2x1 mux which selects between "Mux B" & "Memory Location" coming from Write Back stage based on the selector "Memory Bit" which also comes from Write Back stage in case of a JMP case or successful Branching.

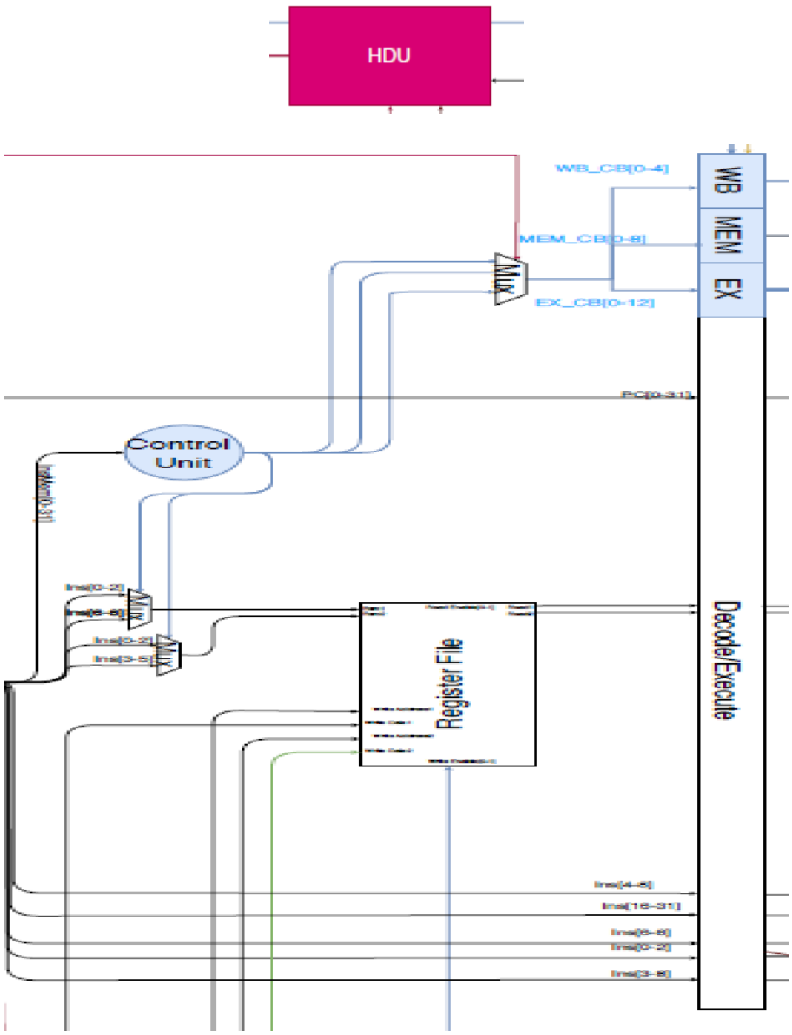
7- **INT UNIT**: a unit that takes control of the fetching unit when "INT Signal" is received and initiate a set of instructions that is related to the INT procedure, when done the control is gone back the normal PC and Instruction Memory.

8- **Instruction Memory:** a 16 bit width memory that holds the instruction for the whole program that will be executed, it has 2048 memory location with the "Program Counter" value as an input to select which instruction to fetch.

9- **Mux D:** 2x1 mux which selects between "Memory Instruction" & "INT Instruction" based on the selector which is "INT Controller" which come from "INT UNIT"

10- **IF/ID Buffer:** this a bank of registers that is used to hold the values of the "Fetched Instruction" & "Current PC" and thus causing the pipeline to initiate.

Decode Stage



The above picture is in depth detailed design of the Decoding unit implemented in the processor.

1- **Control Unit:** a unit that generate the control signals that decide the function of each component in the processor.

In the next pages are the outputs of the control unit for each stage:

Decode Control Signals

OPERATION	Rdst_Rsrc1/Rdst_Rsrc2
NOP	0x0/0x0
SETC	0x0/0x0
CLRC	0x0/0x0
INC	0x0/0x0
DEC	0x0/0x0
OUT	0x0/0x0
IN	0x0/0x0
SWAP	0x1/0x0
ADD	0x1/0x1
SUB	0x1/0x1
AND	0x1/0x1
OR	0x1/0x1
NOT	0x0/0x0
SHL	0x0/0x0
SHR	0x0/0x0
PUSH	0x0/0x0
POP	0x0/0x0
CALL	0x0/0x0
RET	0x0/0x0
RTI	0x0/0x0
JZ	0x0/0x0
JN	0x0/0x0
JC	0x0/0x0
JMP	0x0/0x0
IADD	0x1/0x0
LDM	0x0/0x0
LDD	0x0/0x0
STD	0x1/0x0
push flags	0x0/0x0
pop m[2]+m[3]	0x0/0x0
push pc	0x0/0x0

Execute Control Signals

OPERATION	Jmp/OUT/JC/JN/JC/Set_Clr_Carry/Reg_IMM/PC_Reg/ALU_Selc

NOP	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
SETC	0x0/0x0/0x0/0x0/0x0/0x0/0x01/0x0/0x0/0x0000
CLRC	0x0/0x0/0x0/0x0/0x0/0x0/0x10/0x0/0x0/0x0000
INC	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0001
DEC	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0010
OUT	0x0/0x1/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
IN	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0011
SWAP	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0011
ADD	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0100
SUB	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0101
AND	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0110
OR	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0111
NOT	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x1000
SHL	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x1001
SHR	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x1010
PUSH	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0011
POP	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
CALL	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0001
RET	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
RTI	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
JZ	0x0/0x0/0x1/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
JN	0x0/0x0/0x0/0x1/0x0/0x0/0x00/0x0/0x0/0x0000
JC	0x0/0x0/0x0/0x0/0x1/0x0/0x00/0x0/0x0/0x0000
JMP	0x1/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
IADD	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x1/0x1/0x0100
LDM	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x1/0x1/0x1011
LDD	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
STD	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x1/0x0011
push flags	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
pop m[2]+m[3]	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0000
push pc	0x0/0x0/0x0/0x0/0x0/0x0/0x00/0x0/0x0/0x0011

Memory Control Signals

OPERATION	Data_Stack/SPSel/WriteEnable/Call/RTI/INT
NOP	0x0/0x000/0x0/0x0/0x0/0x0
SETC	0x0/0x000/0x0/0x0/0x0/0x0
CLRC	0x0/0x000/0x0/0x0/0x0/0x0
INC	0x0/0x000/0x0/0x0/0x0/0x0
DEC	0x0/0x000/0x0/0x0/0x0/0x0
OUT	0x0/0x000/0x0/0x0/0x0/0x0
IN	0x0/0x000/0x0/0x0/0x0/0x0
SWAP	0x0/0x000/0x0/0x0/0x0/0x0
ADD	0x0/0x000/0x0/0x0/0x0/0x0
SUB	0x0/0x000/0x0/0x0/0x0/0x0
AND	0x0/0x000/0x0/0x0/0x0/0x0
OR	0x0/0x000/0x0/0x0/0x0/0x0
NOT	0x0/0x000/0x0/0x0/0x0/0x0
SHL	0x0/0x000/0x0/0x0/0x0/0x0
SHR	0x0/0x000/0x0/0x0/0x0/0x0
PUSH	0x1/0x001/0x1/0x0/0x0/0x0
POP	0x1/0x010/0x0/0x0/0x0/0x0
CALL	0x1/0x011/0x1/0x1/0x0/0x0
RET	0x1/0x100/0x0/0x0/0x0/0x0
RTI	0x1/0x100/0x0/0x0/0x1/0x0
JZ	0x0/0x000/0x0/0x0/0x0/0x0
JN	0x0/0x000/0x0/0x0/0x0/0x0
JC	0x0/0x000/0x0/0x0/0x0/0x0
JMP	0x0/0x000/0x0/0x0/0x0/0x0
IADD	0x0/0x000/0x0/0x0/0x0/0x0
LDM	0x0/0x000/0x0/0x0/0x0/0x0
LDD	0x0/0x000/0x0/0x0/0x0/0x0
STD	0x0/0x000/0x1/0x0/0x0/0x0
push flags	0x1/0x001/0x1/0x0/0x0/0x1
pop m[2]+m[3]	0x0/0x010/0x0/0x0/0x0/0x0
push pc	0x1/0x001/0x1/0x0/0x0/0x0

Memory Control Signals

OPERATION	Result_Mem/WriteEnable/IN/RegPC_MemPC
NOP	0x0/0x00/0x0/0x0
SETC	0x0/0x00/0x0/0x0
CLRC	0x0/0x00/0x0/0x0
INC	0x0/0x01/0x0/0x0
DEC	0x0/0x01/0x0/0x0
OUT	0x0/0x00/0x0/0x0
IN	0x0/0x01/0x1/0x0
SWAP	0x0/0x10/0x0/0x0
ADD	0x0/0x01/0x0/0x0
SUB	0x0/0x01/0x0/0x0
AND	0x0/0x01/0x0/0x0
OR	0x0/0x01/0x0/0x0
NOT	0x0/0x01/0x0/0x0
SHL	0x0/0x01/0x0/0x0
SHR	0x0/0x01/0x0/0x0
PUSH	0x0/0x00/0x0/0x0
POP	0x1/0x01/0x0/0x0
CALL	0x1/0x00/0x0/0x1
RET	0x1/0x00/0x0/0x1
RTI	0x0/0x00/0x0/0x1
JZ	0x0/0x00/0x0/0x0
JN	0x0/0x00/0x0/0x0
JC	0x0/0x00/0x0/0x0
JMP	0x0/0x00/0x0/0x0
IADD	0x0/0x01/0x0/0x0
LDM	0x0/0x01/0x0/0x0
LDD	0x1/0x01/0x0/0x0
STD	0x0/0x00/0x0/0x0
push flags	0x0/0x00/0x0/0x0
pop m[2]+m[3]	0x1/0x00/0x0/0x1
push pc	0x0/0x00/0x0/0x0

2- **Mux A**: 2x1 mux which selects between "Rdst" & "Rsrc1" based on the selector coming from control unit.

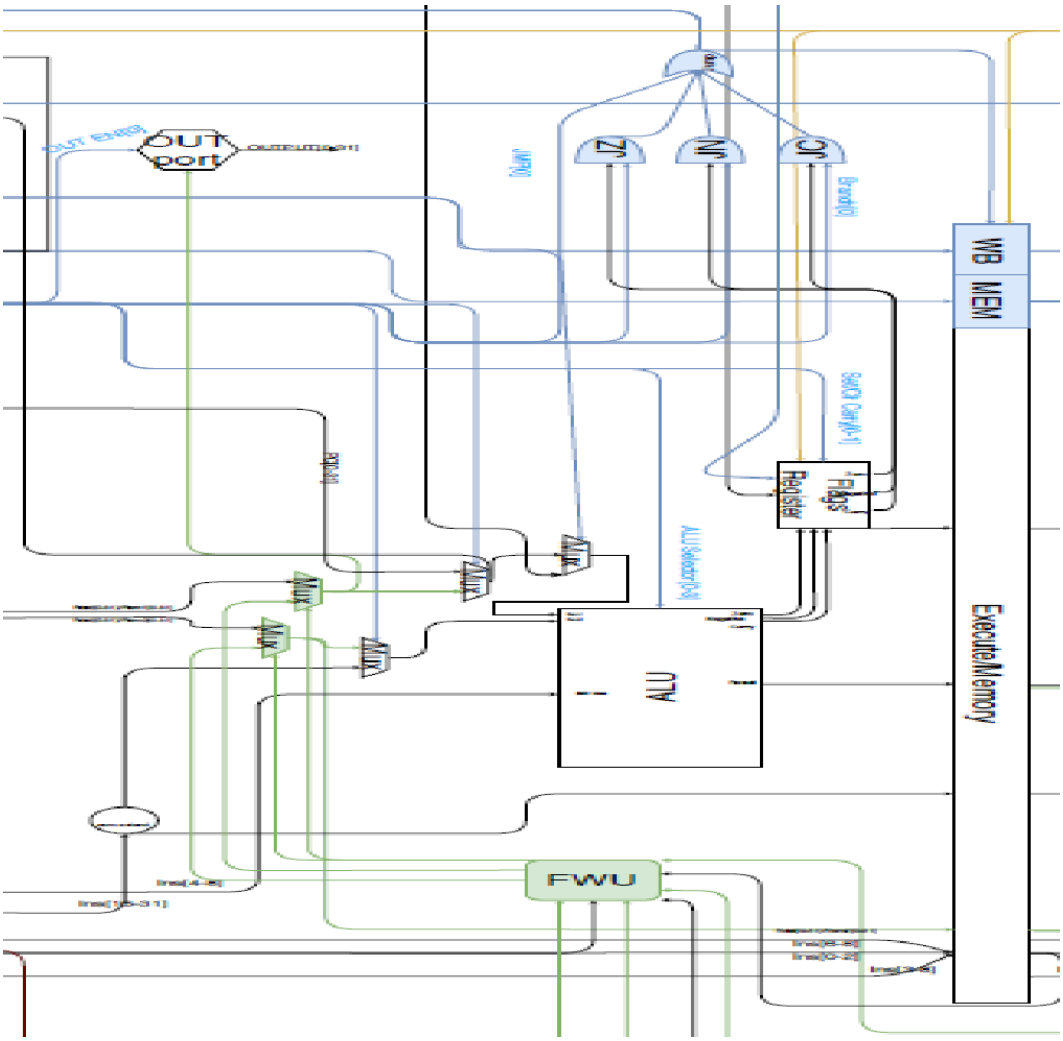
3- **Mux B**: 2x1 mux which selects between "Rdst" & "Rsrc2" based on the selector coming from control unit.

4- **Register File**: a unit containing 8 registers of 32 bit size that takes 5 inputs from the Write back stage and fetches 2 outputs. The inputs are:

a. “Address 1 Data”, 32 bits
b. “Address 2 Data” 32 bits
c. “Address 1” 32 bits
d. “Address 2” 32 bits
e. “Write Selector”, 32 bits

5- **HDU**: a unit that decides if there is a load case use hazard and initiates signals to cause a 1 cycle stall by giving 0’s to the next in line signal buffers and stopping PC and IF/ID buffer.
6- **ID/EX Buffer**: this a bank of registers that is used to hold the values of the “Fetched Instruction” & “Current PC” & “Control Signals” for each stage.

Execute Stage



1. **Zero Extender**: takes a 16 bit value (the immediate value) and extends to 32 bits value by adding zeros to the left.

2. **ALU**: executes the following operations:

ALU selector	Operation
0001	Increment Rsrc1
0010	Decrement Rsrc2
0011	Pass Rsrc1

0100	Add Rsrc1 and Rsrc2
0101	Subtract Rsrc2 from Rsrc1
0110	And Rsrc1 and Rsrc2
0111	Or Rsrc1 and Rsrc2
1000	Not Rsrc1
1001	Shift left
1010	Shift right
1011	Pass Rsrc2

The ALU takes 2 32-bit operands, 5 bits as a shift amount, 4-bit selector as inputs.
It outputs 32-bit result and 1 bit zero, carry and negative flags

3. **Flags Register:** is a register file that keeps the values of the zero, negative and carry flags.

Its inputs are:

- Zero flag from ALU
- Negative flag from ALU
- Zero flag from ALU
- SETC: 2 bits that if set to 01 will set the carry flag and if set to 10 will clear the carry flag, else it is disabled.
- RETI bit

Its outputs are:

- Zero flag going into the jump unit
- Carry flag going in to the jump unit
- Negative flag going in to the jump unit
- 4 bits flags going in to Execute/Memory buffers.

4. **Out port:** takes a one bit enable, and 32 bits value to output.

5. **In port**: takes a 32 bit value.

6. **5 Multiplexers** explained in the following section.

3 AND gates and 1 OR gate.

7. **Forwarding Unit**: the unit responsible for forwarding Rsrc1 and Rsrc2 from Memory and Write Back stages. It is also responsible for the selectors of the multiplexers that pass either the register values or the forwarded values.

8. **Execute/Memory Buffer:**

Inputs and Outputs:

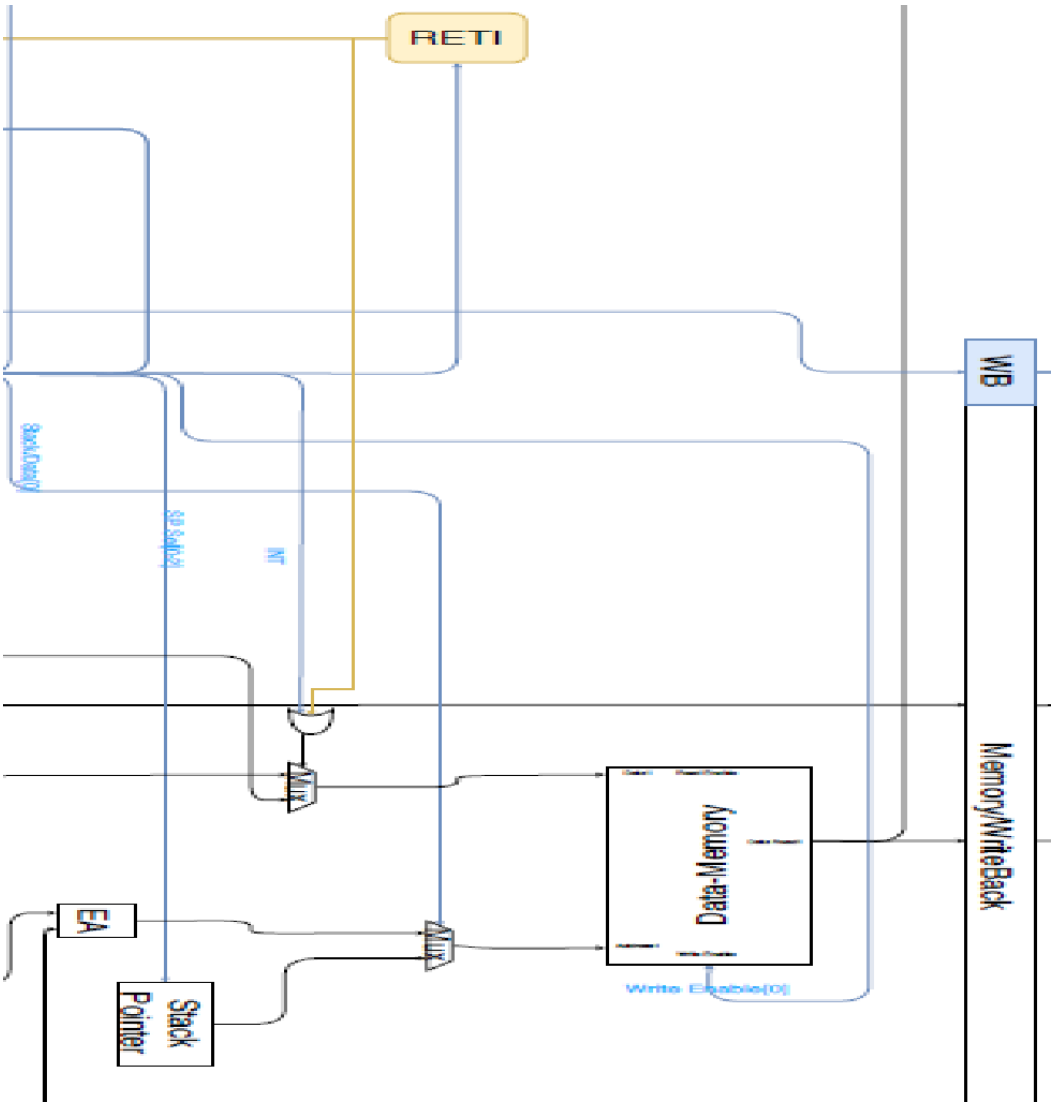
- Rdst/Rsrc2 32 bit value
- Bits [0-8]: representing the addresses of the Rsrc1, Rsrc2 and Rdst.
- The instruction bits [16-31]
- The ALU result.
- The values of the Flags Register.
- Control signals: stack/Data selector for the memory stage, Flags Register pop enable...etc.

9. **General Flow:**

1. Rsrc1 and Rsrc2 enter the execution stage.
2. If the instruction is 16 bit, the zero extender extends zeros, if it is 32 it extends the immediate value.
3. It is determined whether Rsrc1, or the forwarded value will pass according to MUX1, based on a selector from the Forwarding Unit.
4. It is determined whether Rsrc2, or the forwarded value will pass according to MUX2, based on a selector from the Forwarding Unit.
5. MUX3 then determines between the passed value from MUX1 and the Program Counter, based on a selector from the control unit.
6. MUX4 determines between the passed value from MUX2 and the immediate value passed from the zero extender based on a selector from the control unit.
7. MUX5 determines between the MUX3's result or the in port based on the In port enable.
8. The ALU then executes its operations and outputs the result to the buffer, and the flags result to the flag register.

9. Each AND gate receives a flag and a control unit signal, so the first AND has zero flag and control unit signal indicating if the command being executed is “JZ” and so on.
10. OR gate then takes the output from the AND gates and an additional signal indicating if the command is JMP. So if the output of the OR gate is “1”, then a jump has to be carried out.

Memory Stage



Data Memory: This memory stores data.
Inputs:

- Data to be entered in the memory which comes from the result of MUX1(flags or data)
- Address1: comes from the result of MUX2 (stack address or effective address)
- Control Signal: Write Enable.

Outputs the value read from the Memory.
Stack Pointer: This pointer points at the first available location in the stack.

Effective Address: calculates the effective address, from the 16 bit immediate value and the Rsrc1 and Rsrc2 addresses.

MUX1: determines between the flags and the result of the ALU, based on the “OR” of the INT and RETI values.

MUX2: determines between the stack pointer and the effective address based on a control signal coming from the control buffers.

Write Back Stage

Inputs:

- Result of ALU that was passed through the memory stage and is coming from the buffer.
- Value that is coming from the Data memory and going through the buffer.
- A control signal deciding between the 2 above inputs.
- A control signal deciding if the program counter is to be updated by the write back value, that originally came from the Data-Memory.
- The first 3 bits of the instruction [0-2] from the previous buffer, representing the Rdst address.
- The bits [6-8] bits of the instruction from the previous buffer, representing the Rsrc1 address.
- Rdst or Rsrc

Outputs:

- The first 3 bits of the instruction [0-2] from the previous buffer, representing the Rdst address.
- The bits [6-8] bits of the instruction from the previous buffer, representing the Rsrc1 address.
- Rdst or Rsrc.
- The result of the multiplexer that goes to the register file and the program counter.

Advanced Analysis of the design

INT Unit: this unit is activated when it receives an int signal and takes control from the pc and instruction memory, it function as follows.

1- It generates 3 NOP instructions to allow the pipeline to clear its content
2- Then it generates the following commands in oreder
   - Push PC
   - Push Flags
   - Pop M[2&3] to Pc
3- Then it generates 3 more NOP instructions to avoid making the pipeline from taking any wrong instructions

Hazard Detection Unit: This unit finds if there is a load case use hazard that is going to happen and prevents that by stalling the pipe one clock cycle until the write back is done. This stall is achieved by:

1- Stalling the pc
2- Stalling the if/id buffer
3- Sending a NOP instruction to the following buffers

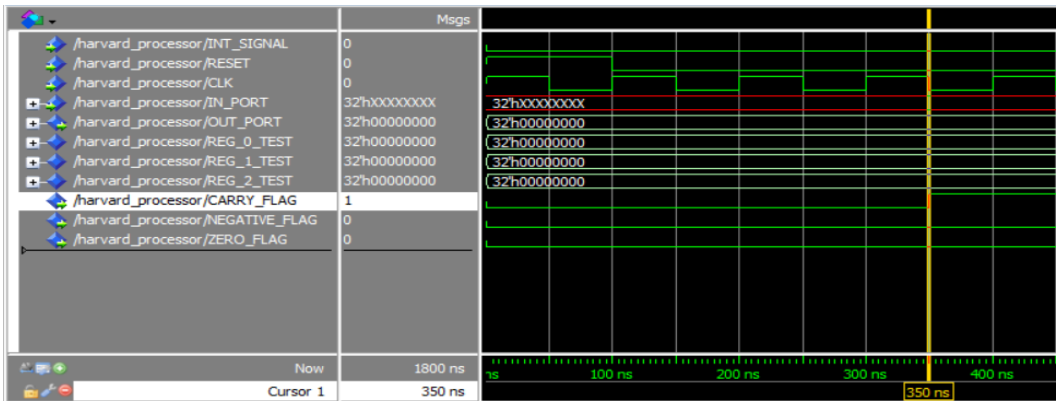
Out/In Ports: These are the ports used to communicate between the processor and outer I/O devices, the Out port is mainly a register that is used to the required register the device requesting it, the in port is a wire that the input devices sends a signal to where the processor takes it and write it to the required register.

Forwarding Unit: This unit is used to prevent data hazards that occur when using the same register in two consecutive commands, if this is to be removed the commands will have to wait for the previous one to finish writing back or to take the old register value which will result in a wrong value.

RTI Unit: This unit is used to help the RTI command stall the pipe one cycle so that it can pop the flags and the pc on two cycles.

Case I: HDU, FWU and flushing are active

One Operand Test Case

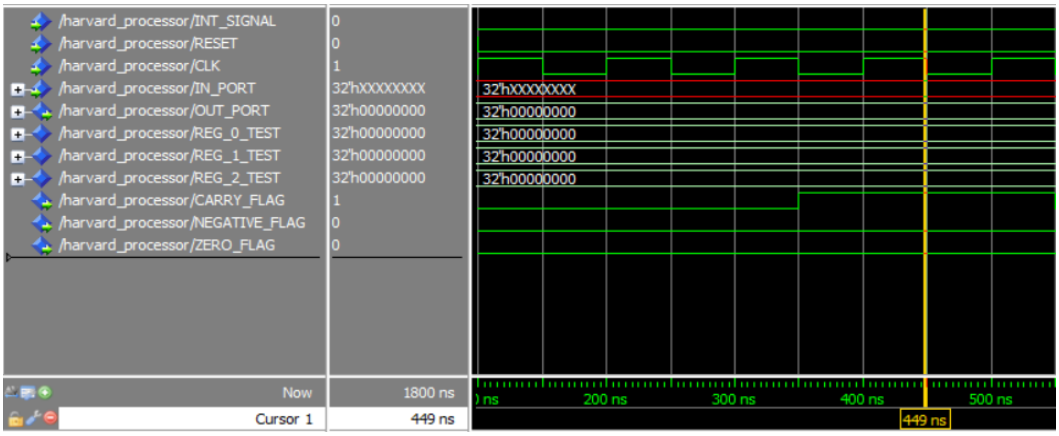


The cursor above is at 350 ns which means:

Fetch => CLRC

DECODE => NOP

EXECUTE => SETC ∴ CF = 1 , NF = S , ZF = S *S MEANS NO CHANGE*



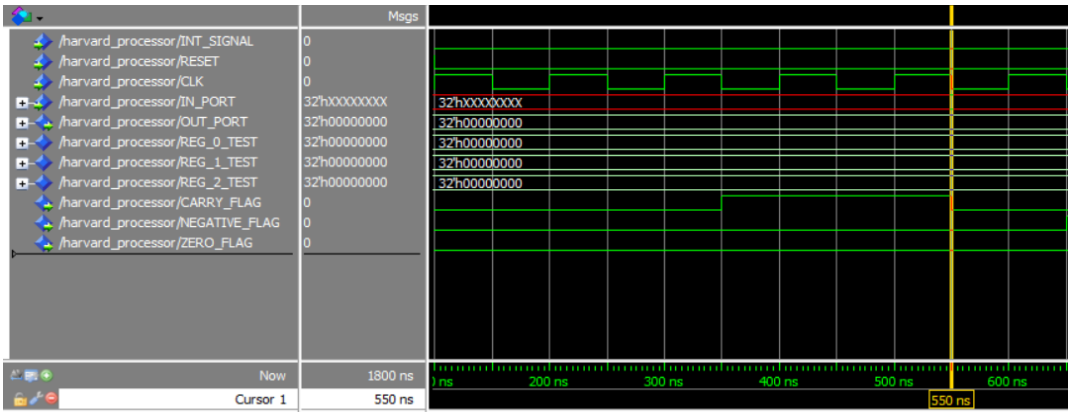
The cursor above is at 450 ns which means:

Fetch => NOT R1

DECODE => CLRC

EXECUTE => NOP∴ CF = S , NF = S , ZF = S

MEMORY => SETC



The cursor above is at 550 ns which means:

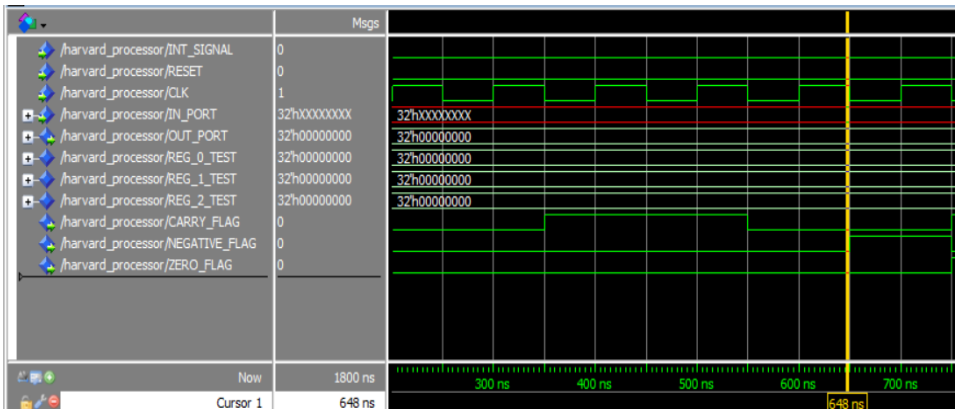
Fetch => INC R1

DECODE => NOT R1

EXECUTE => CLRC ∴ CF = 0 , NF = X , ZF = X

MEMORY => NOP

WRITE-BACK => SETC



The cursor above is at 750 ns which means:

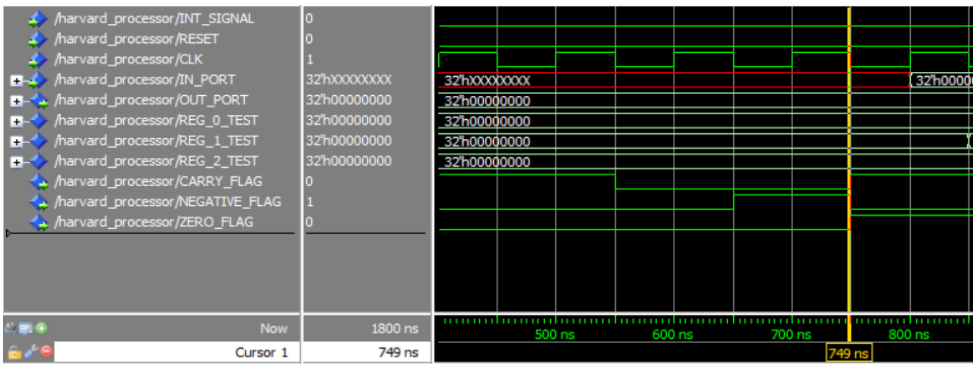
Fetch => IN R1

DECODE => INC R1

EXECUTE => NOT R1 ∴ CF = S , NF = 1 , ZF = 0

MEMORY => CLRC

WRITE-BACK => NOP



The cursor above is at 650 ns which means:

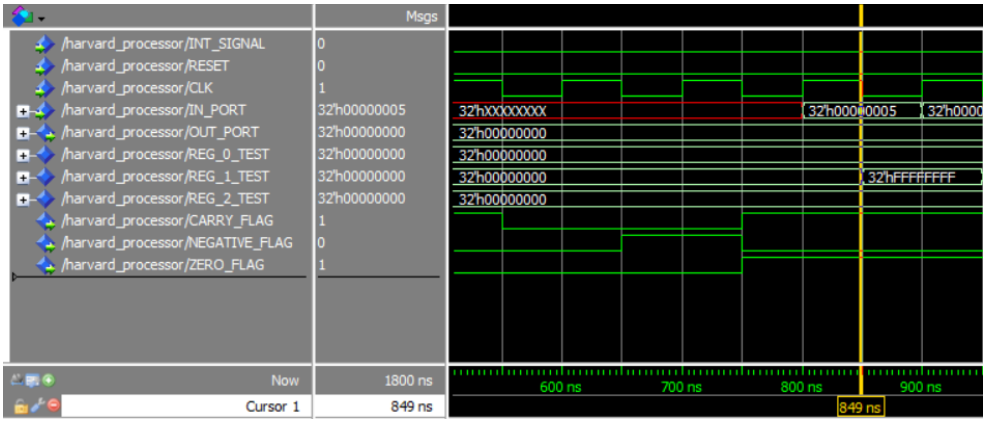
Fetch => IN R2

DECODE => IN R1

EXECUTE => INC R1 ∴ CF = 1 , NF = 0 , ZF = 1

MEMORY => NOT R1

WRITE-BACK => CLRC



The cursor above is at 850 ns which means:

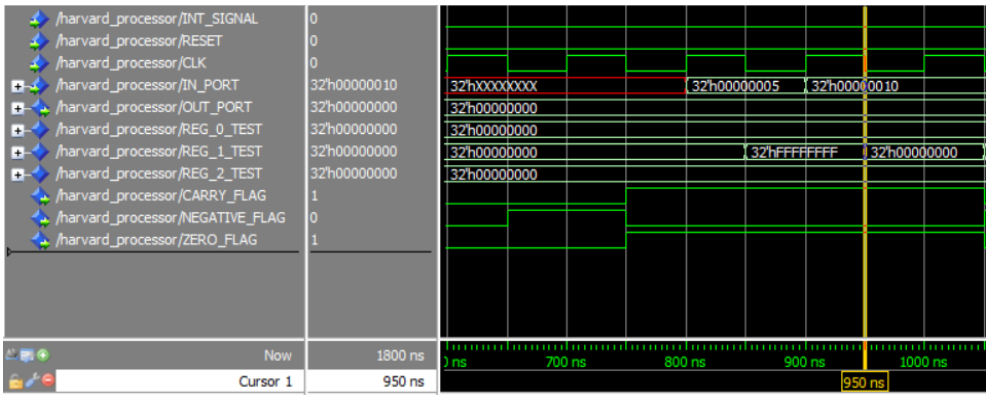
Fetch => NOT R2

DECODE => IN R2

EXECUTE => IN R1 ∴ CF = S , NF = S , ZF = S

MEMORY => INC R1

WRITE-BACK => NOT R1 R1 = FFFFFFFF



The cursor above is at 950 ns which means:

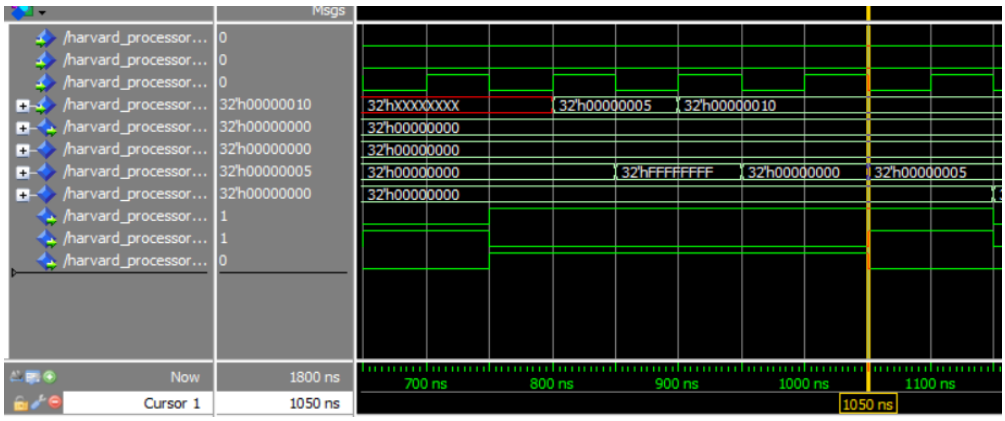
Fetch => INC R1

DECODE => NOT R2

EXECUTE => IN R2 ∴ CF = S , NF = S , ZF = S

MEMORY => IN R1

WRITE-BACK => INC R1 R1 = 0



The cursor above is at 1050 ns which means:

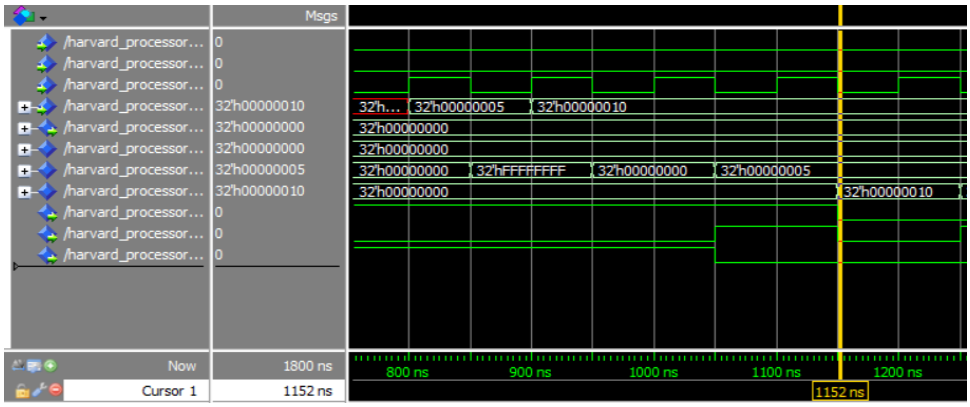
Fetch => DEC R2

DECODE => INC R1

EXECUTE => NOT R2 ∴ CF = S , NF = 1 , ZF = 0 , R1 =5

MEMORY => IN R2

WRITE-BACK => IN R1 R1 = 5



The cursor above is at 1150 ns which means:

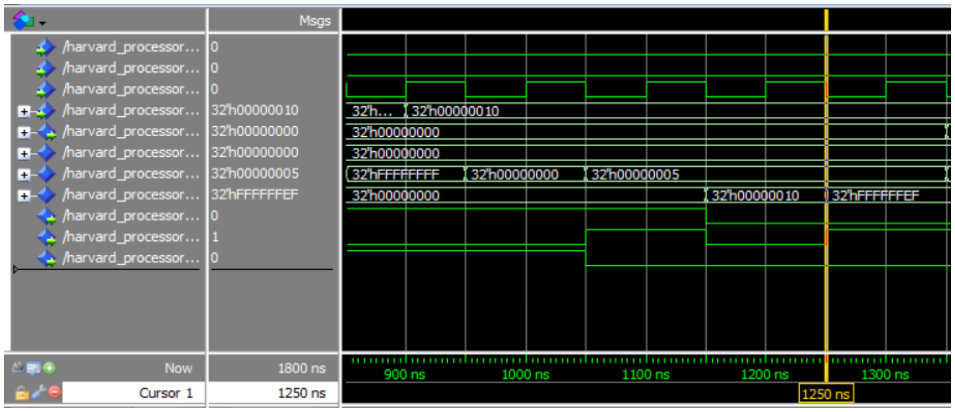
Fetch => OUT R1

DECODE => DEC R2

EXECUTE => INC R1 ∴ CF = 0, NF = 0 , ZF = 0 R2 = 10

MEMORY => NOT R2

WRITE-BACK => IN R2 R2 = 10



The cursor above is at 1250 ns which means:

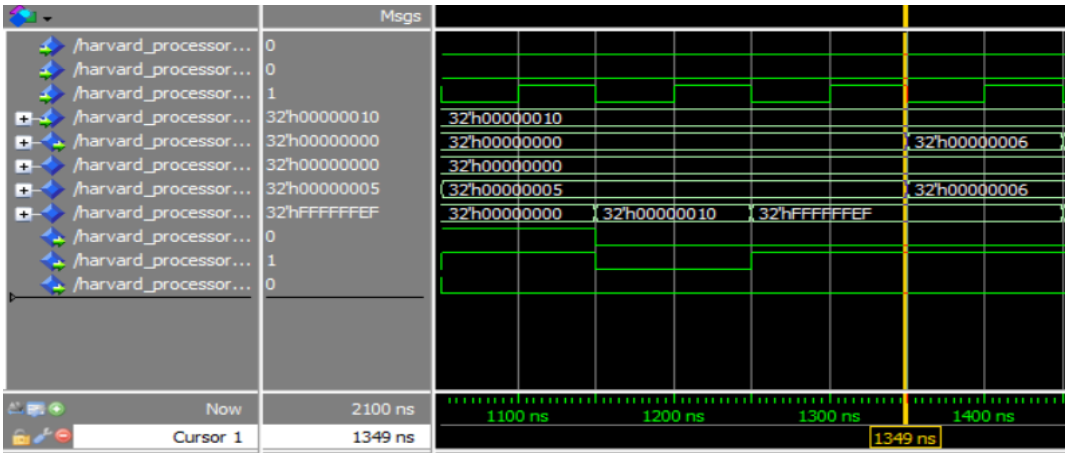
Fetch => OUT R2

DECODE => OUT R1

EXECUTE => DEC R2 ∴ CF = 0, NF = 1 , ZF = 0

MEMORY => INC R1

WRITE-BACK => NOT R2 R2= FFFFFFEF



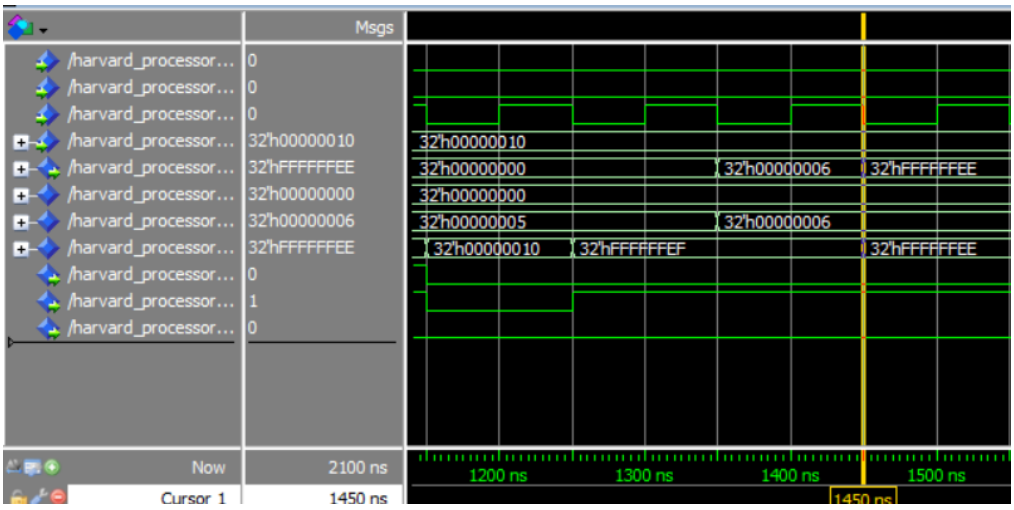
The cursor above is at 1350 ns which means:

DECODE => OUT R2

EXECUTE => OUT R1 ∴ CF = S, NF = S , ZF = S OUT PORT = 6

MEMORY => DEC R2

WRITE-BACK => INC R1 R1= 6



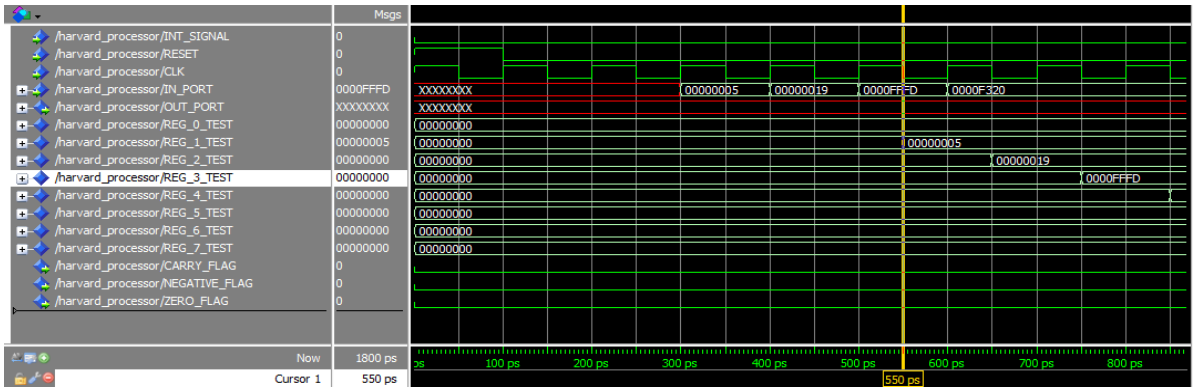
The cursor above is at 1450 ns which means:

EXECUTE => OUT R2 ∴ CF = S , NF = S , ZF = S OUT PORT = FFFFFFEE

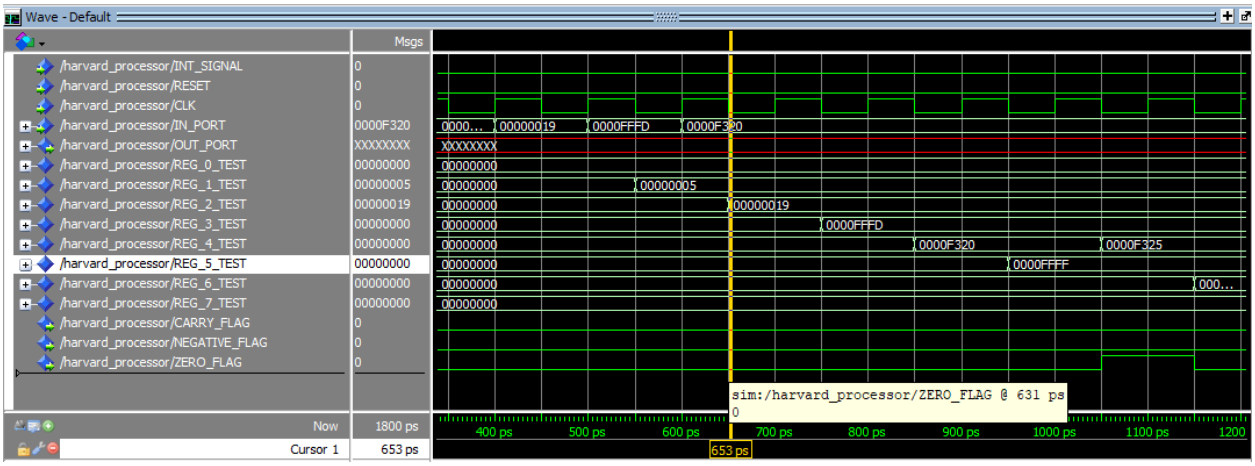
MEMORY => OUT R1

WRITE-BACK => DEC R2

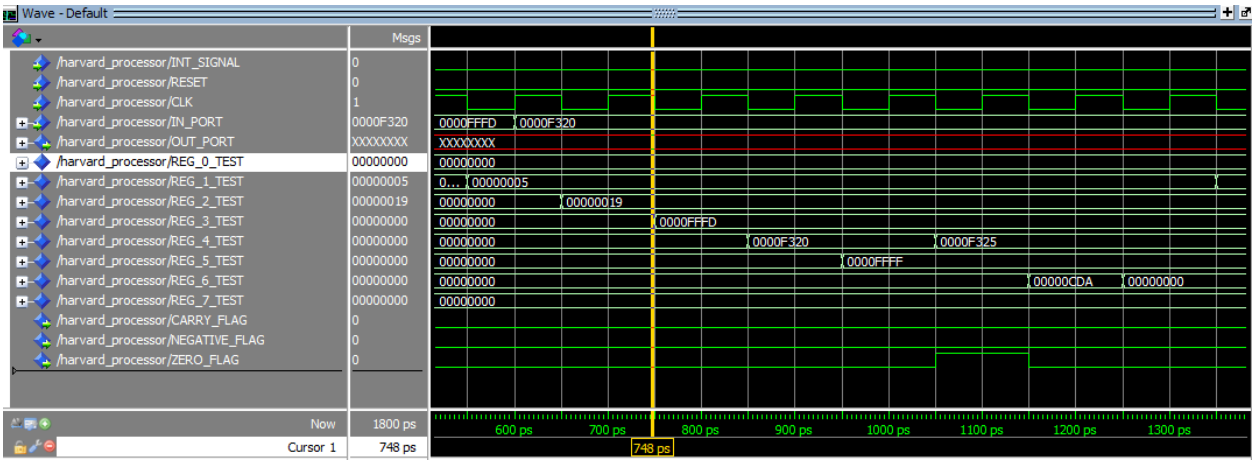
Two Operand Test Case



The Cursor above is at 550 ps which means:
Fetch => IADD R3,R5,2
Decode => IN R4
Execute => IN R3
Memory =>IN R2
WriteBack =>IN R1 R1 = 00000005



The Cursor above is at 650 ps which means:
Fetch => ADD R1,R4,R4
Decode => IADD R3,R5,2
Execute =>IN R4
Memory =>IN R3
WriteBack =>IN R2 R2= 00000019



The Cursor above is at 750 ps which means:

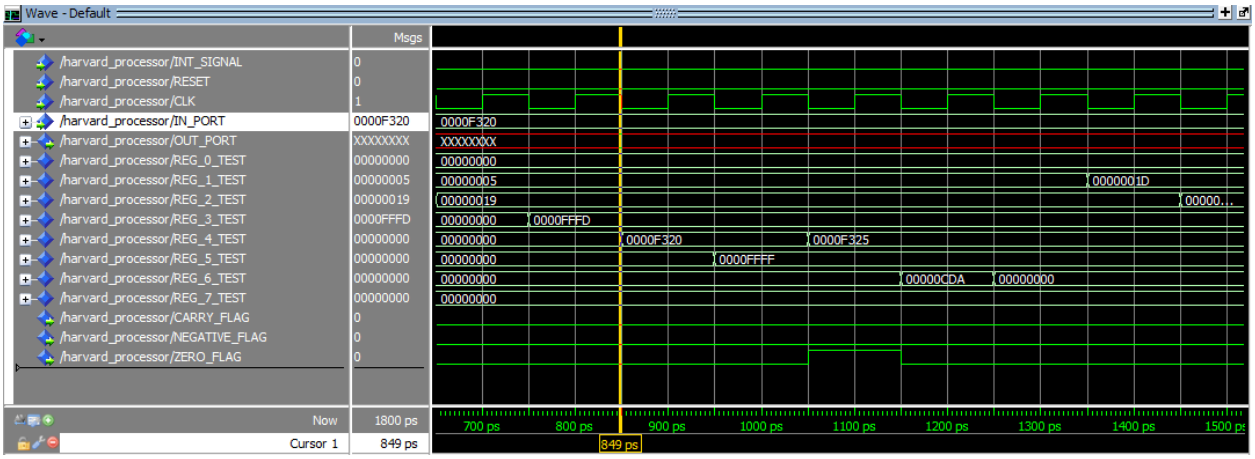
Fetch => SUB R5,R4,R6

Decode => ADD R1,R4,R4

Execute =>IADD R3,R5,2

Memory =>IN R4

WriteBack =>IN R3 R3 = 0000FFFD



The Cursor above is at 850 ps which means:

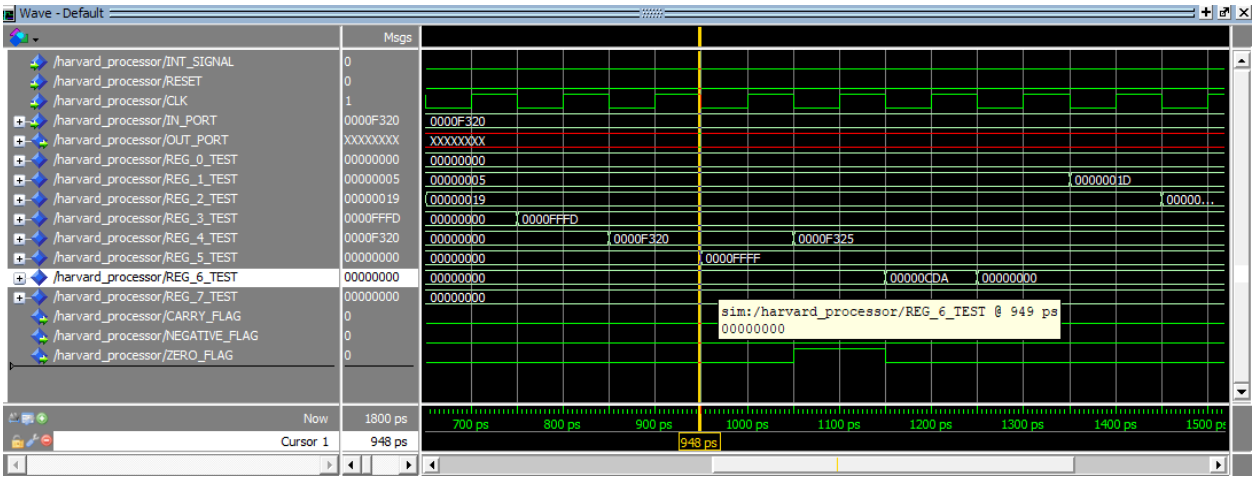
Fetch => AND R7,R6,R6

Decode => SUB R5,R4,R6

Execute =>ADD R1,R4,R4

Memory =>IADD R3,R5,2:

WriteBack =>IN R4 #R4 = 0000F320



The Cursor above is at 950 ps which means:

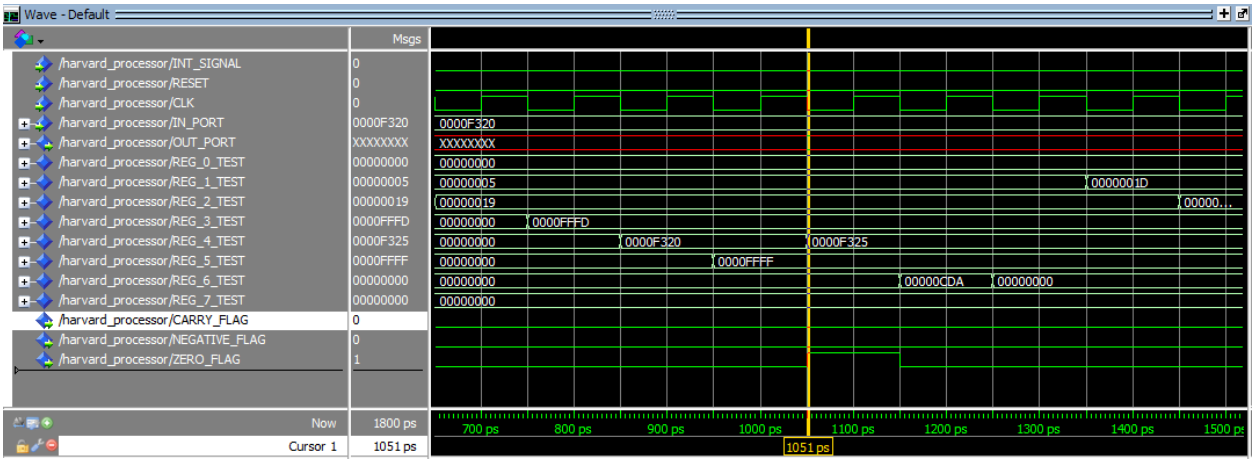
Fetch =>OR R2,R1,R1

Decode => AND R7,R6,R6

Execute =>SUB R5,R4,R6

Memory =>ADD R1,R4,R4

WriteBack =>IADD R3,R5,2 #R5 = 0000FFFFF



The Cursor above is at 1050 ps which means:

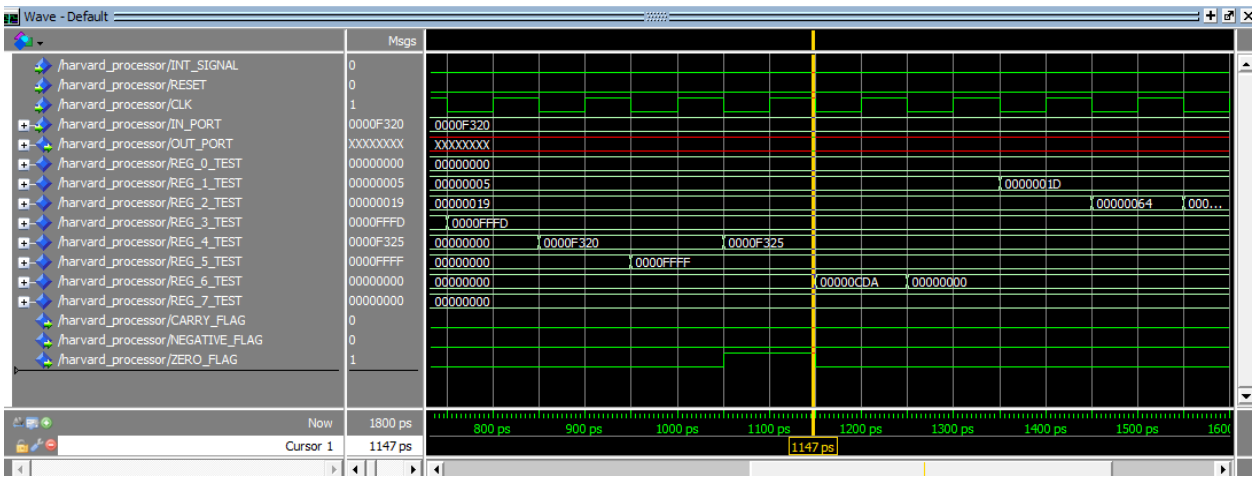
Fetch =>SHL R2,2

Decode => OR R2,R1,R1

Execute =>AND R7,R6,R6 ZF = 0

Memory =>SUB R5,R4,R6

WriteBack =>ADD R1,R4,R4 #R4 = 0000F325



The Cursor above is at 1150 ps which means:

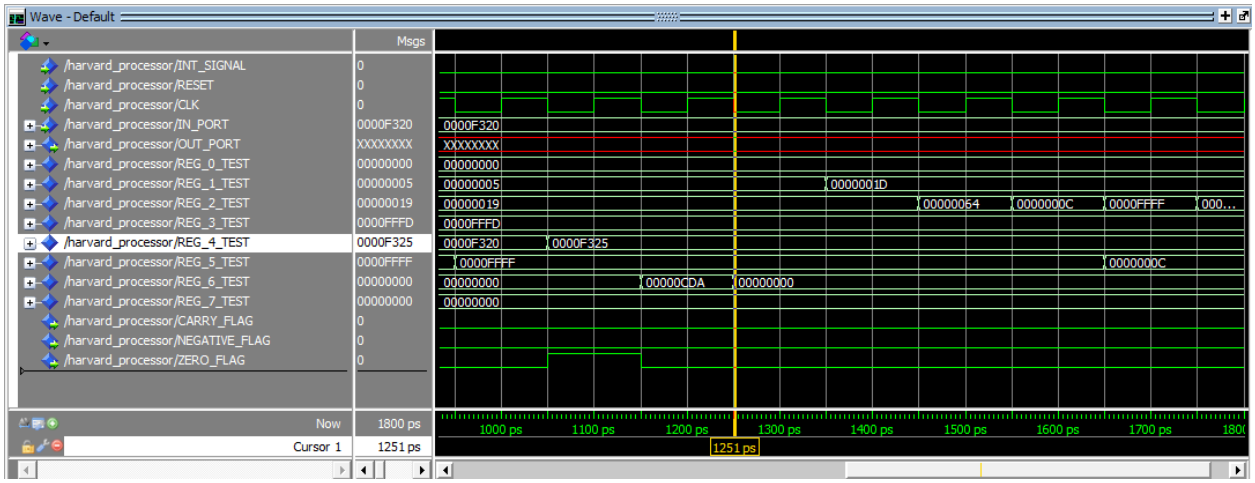
Fetch =>SHR R2,3

Decode => SHL R2,2

Execute =>OR R2,R1,R1 ZF = 0

Memory =>AND R7,R6,R6

WriteBack =>SUB R5,R4,R6 R6 = 00000CDA



The Cursor above is at 1250 ps which means:

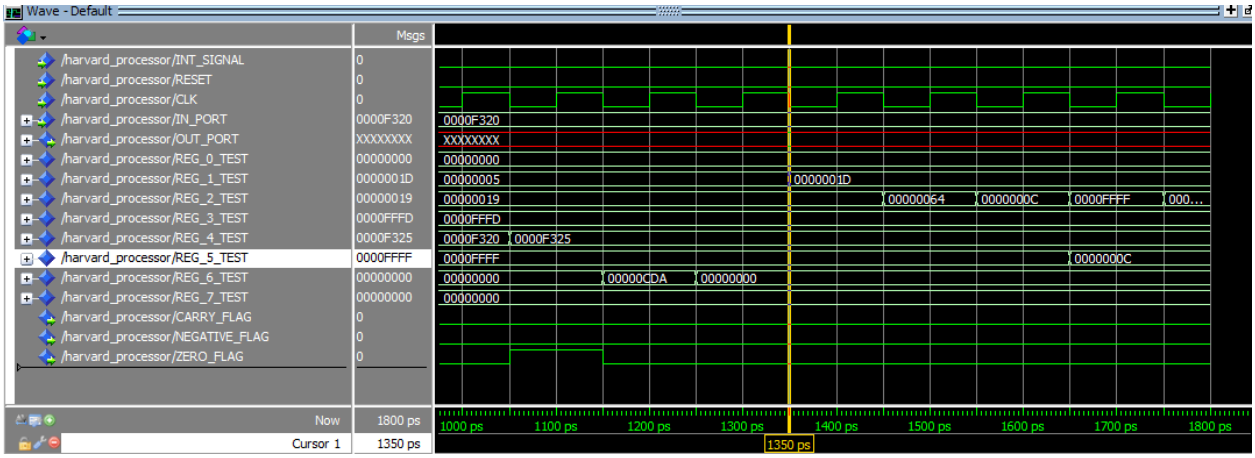
Fetch =>SWAP R2,R5

Decode => SHR R2,3

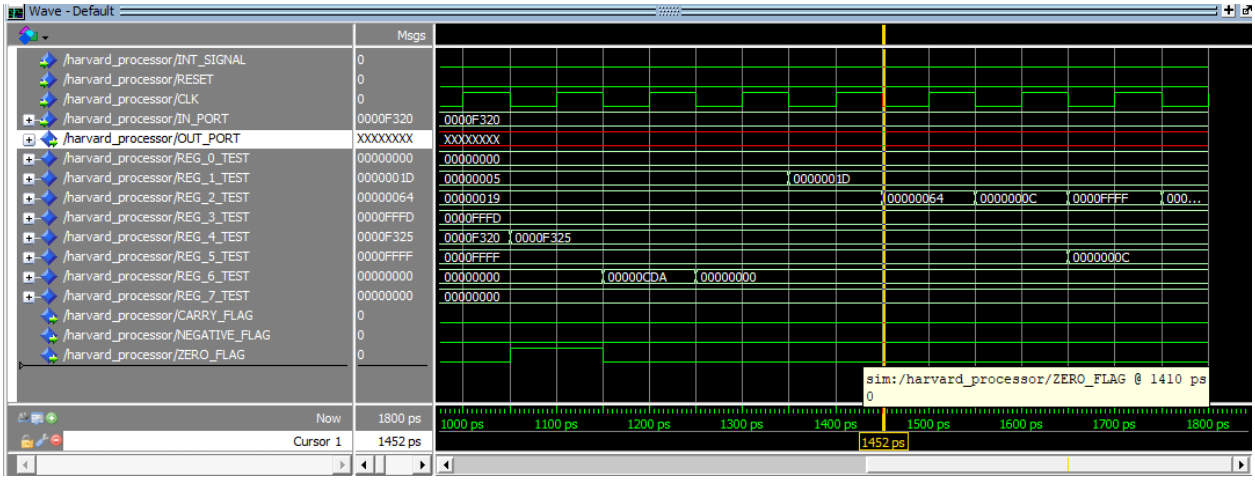
Execute =>SHL R2,2

Memory =>OR R2,R1,R1

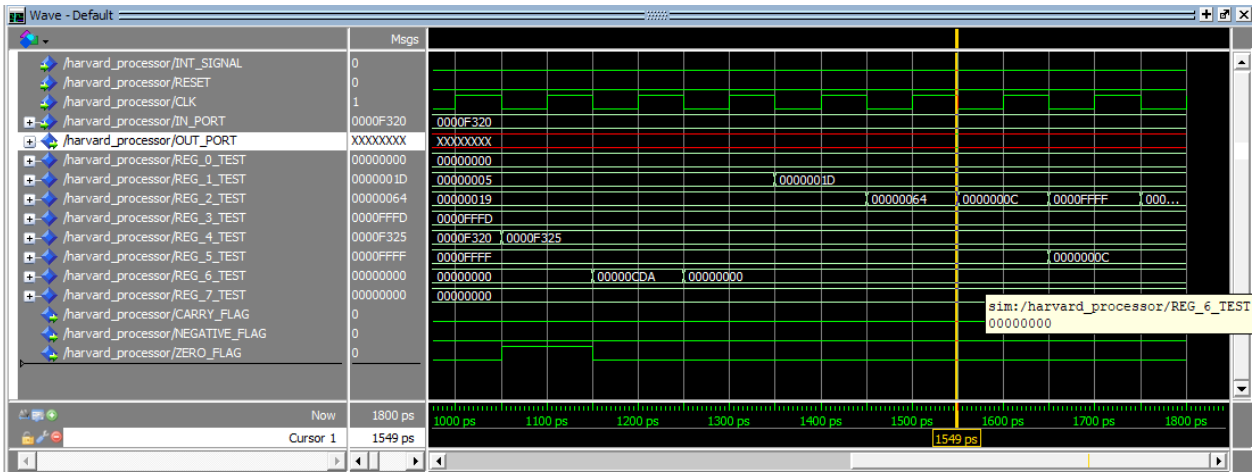
WriteBack =>AND R7,R6,R6 # R6 = 00000000



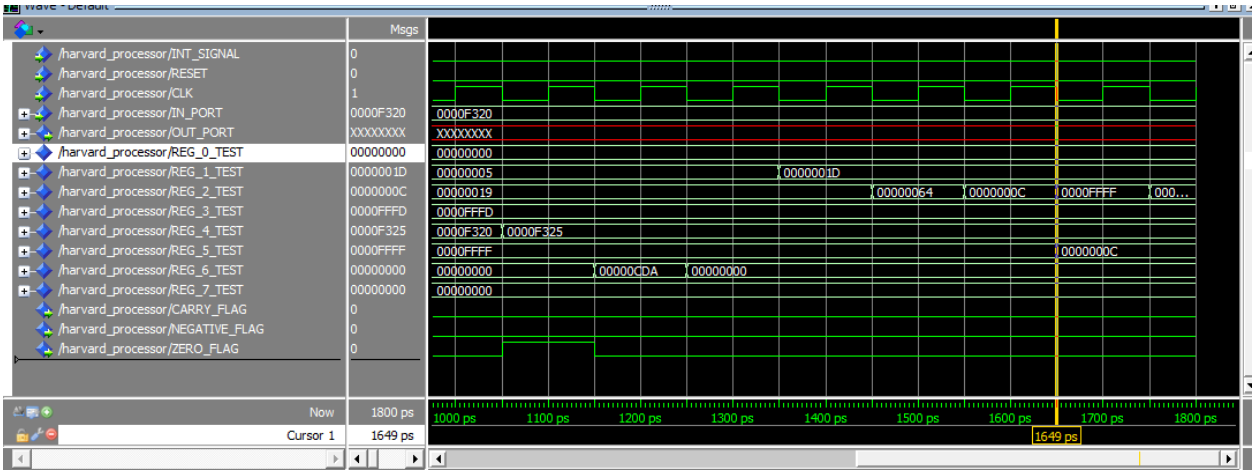
The Cursor above is at 1350 ps which means:
Fetch =>ADD R5,R2,R2
Decode => SWAP R2,R5
Execute =>SHR R2,3
Memory =>SHL R2,2
WriteBack =>OR R2,R1,R1 #R1 = 0000001D



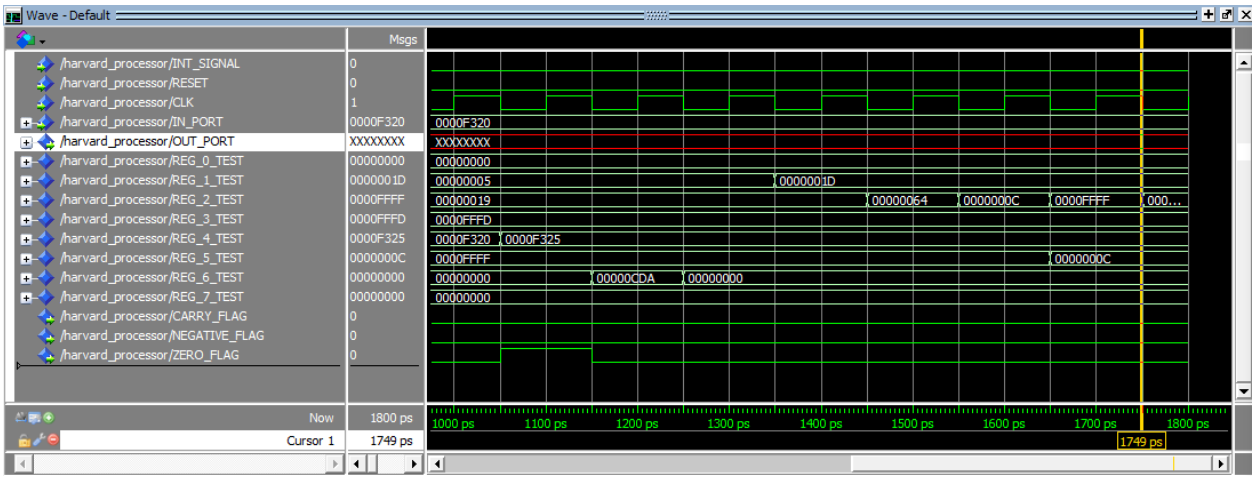
The Cursor above is at 1450 ps which means:
Decode => ADD R5,R2,R2
Execute =>SWAP R2,R5
Memory =>SHR R2,3
WriteBack =>SHL R2,2 #R2 = 00000064



The Cursor above is at 1550 ps which means:
Execute =>ADD R5,R2,R2
Memory =>SWAP R2,R5
WriteBack =>SHR R2,3   #R2 = 0000000C

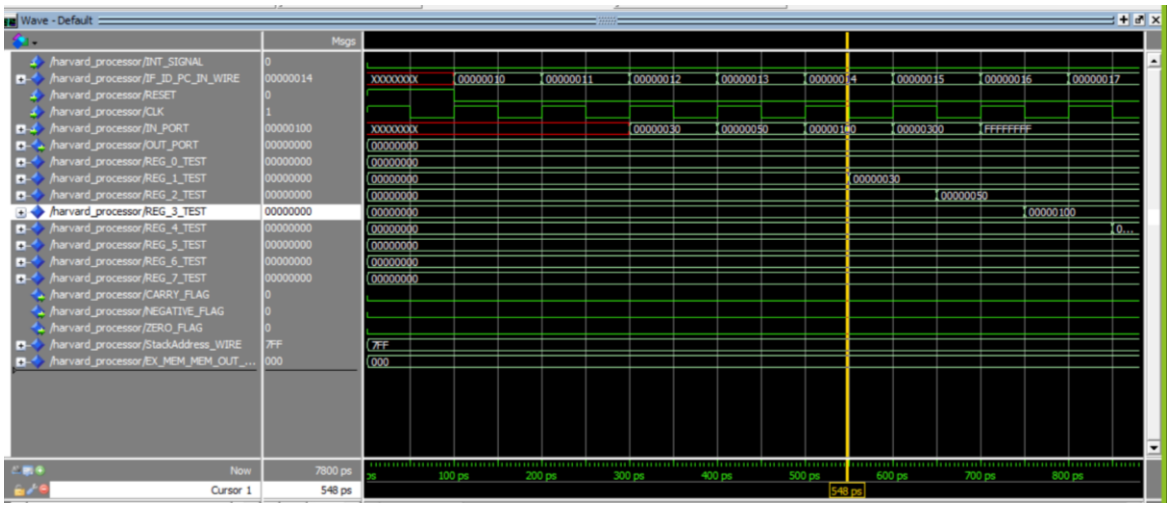


The Cursor above is at 1650 ps which means:
Memory =>ADD R5,R2,R2
WriteBack =>SWAP R2,R5 #R2 = 0000FFFF #R5 = 0000000C

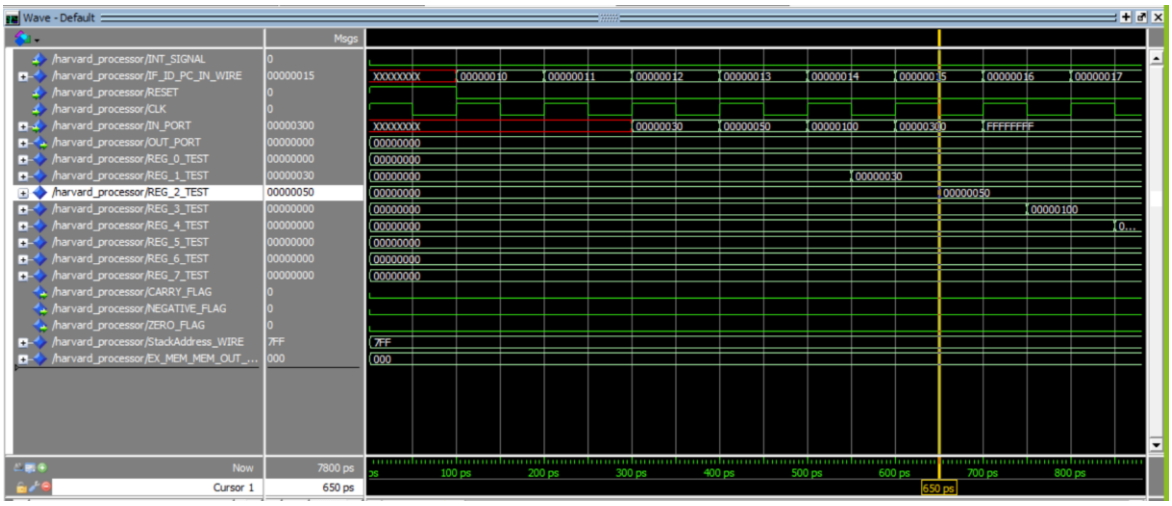


The Cursor above is at 1750 ps which means:
WriteBack =>ADD R5,R2,R2 #R2 = 0001000B

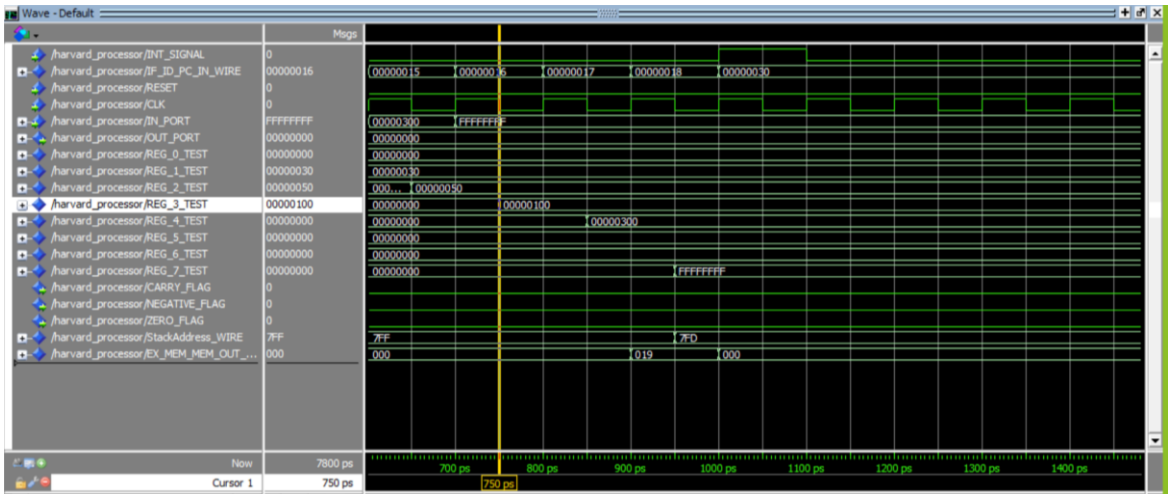
Branch Test Case



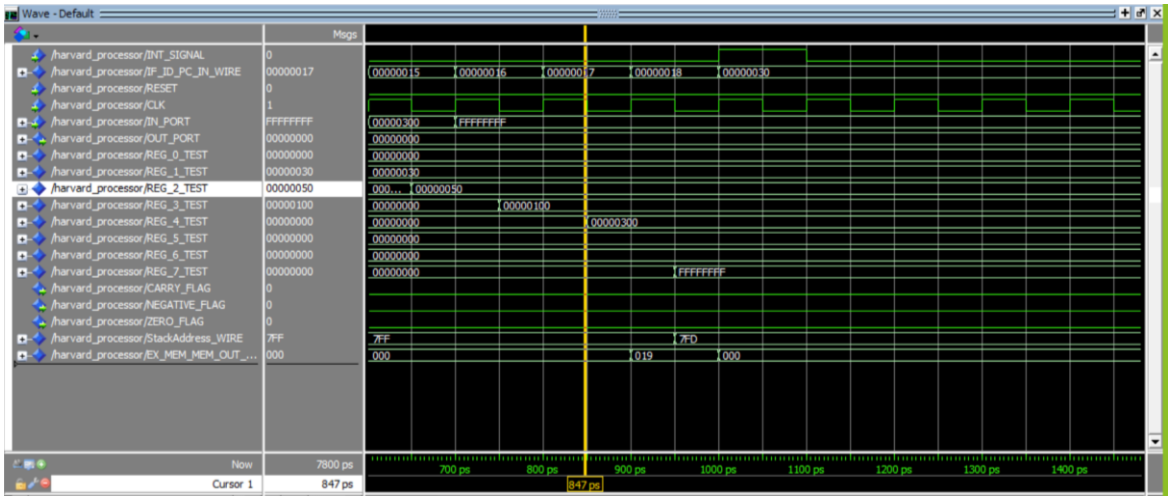
The Cursor above is at 548 ps which means:
Fetch => IN R7
Decode => IN R4
Execute => IN R3
Memory =>IN R2
WriteBack => IN R1 R1 = 00000030



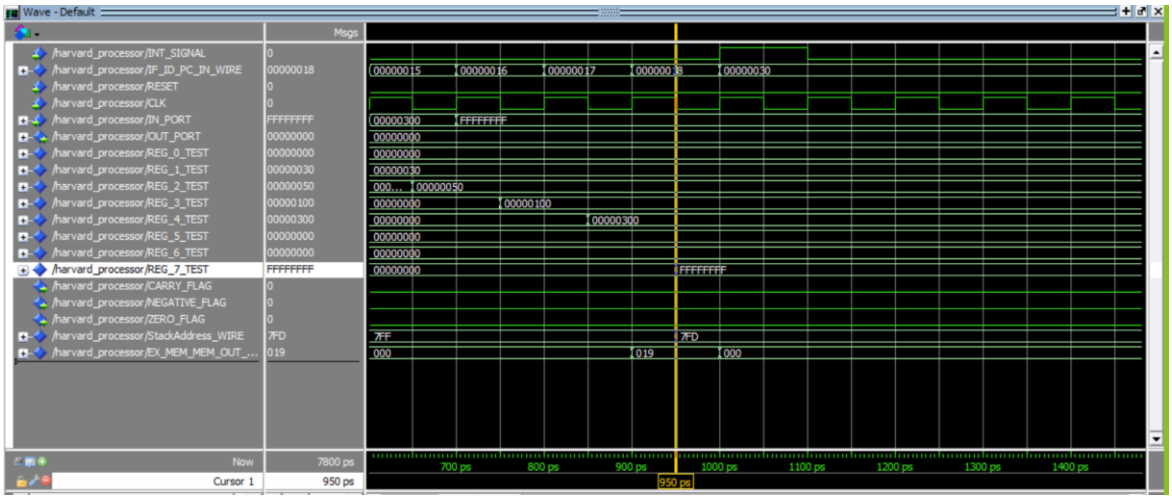
The Cursor above is at 650 ps which means:
Fetch => PUSH R4
Decode => IN R7
Execute => IN R4
Memory =>IN R3
WriteBack => IN R2 R2 = 00000050



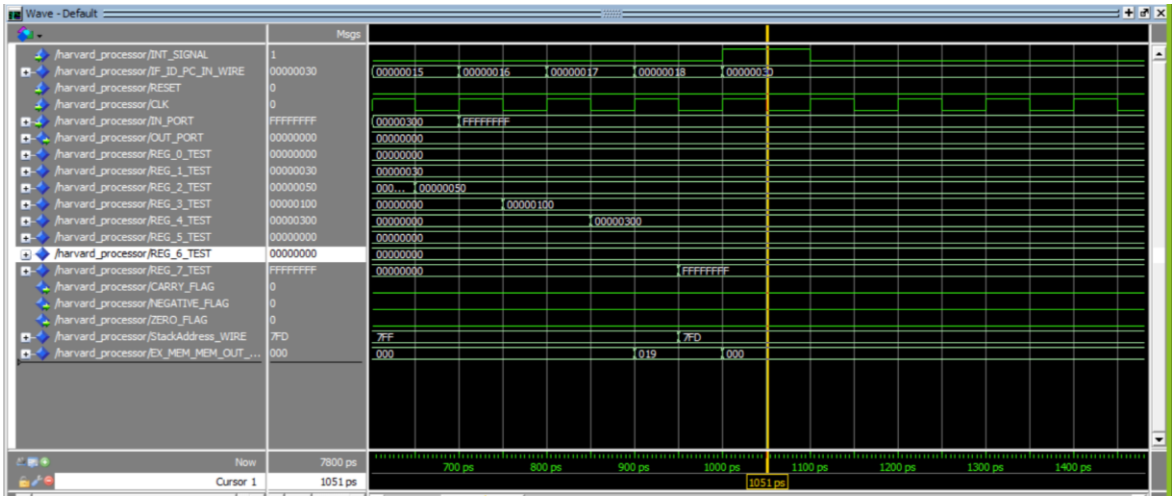
The Cursor above is at 750 ps which means:
Fetch => JMP R1
Decode => PUSH R4
Execute => IN R7
Memory =>IN R4
WriteBack => IN R3 R3 = 00000100



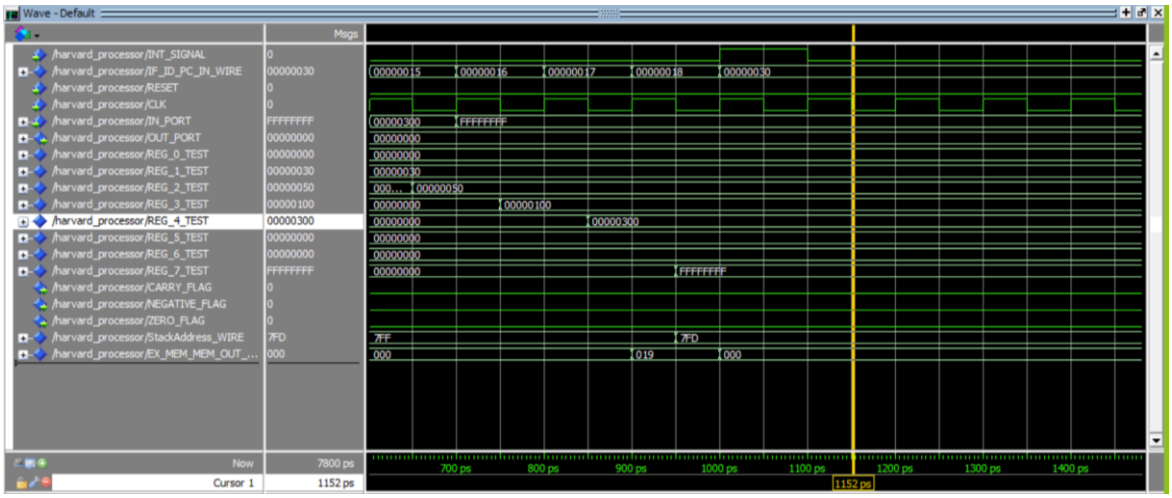
The Cursor above is at 850 ps which means:
Fetch => INC R7
Decode => JMP R1
Execute => PUSH R4
Memory =>IN R7
WriteBack => IN R4 R4 = 00000300



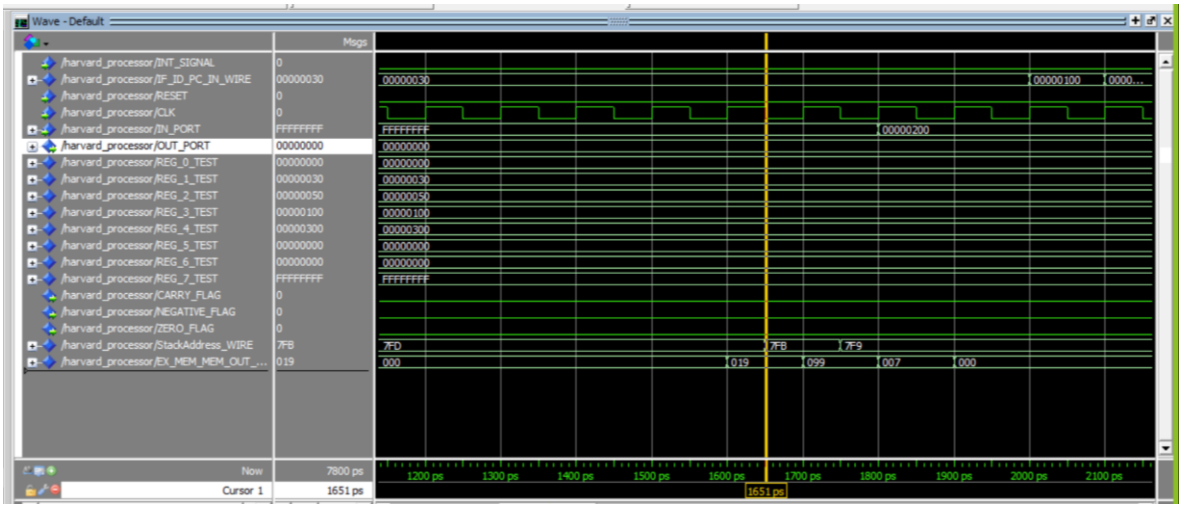
The Cursor above is at 950 ps which means:
Fetch => NOP
Decode => INC R7
Execute => JMP R1
Memory => PUSH R4 SP=7FD (FIRST EMPTY PLACE IN STACK)
WriteBack => IN R7 R7 = FFFFFFFF



The Cursor above is at 1051 ps which means:INT = 1
Fetch => BUBBLE BEC OF INTERRUPT
Decode => BUBLE
Execute => BUBLE
Memory => JMP R1
WriteBack => PUSH R4

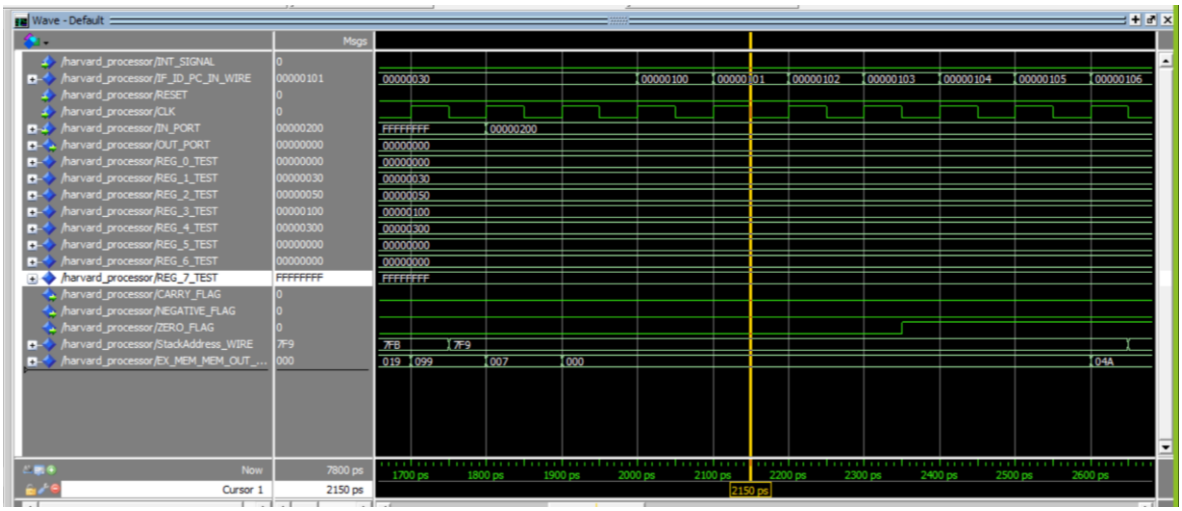


The Cursor above is at 1152 ps which means:
Fetch => BUBBLE
Decode => BUBLE
Execute => BUBLE
Memory =>BUBLE
WriteBack => JMP R1

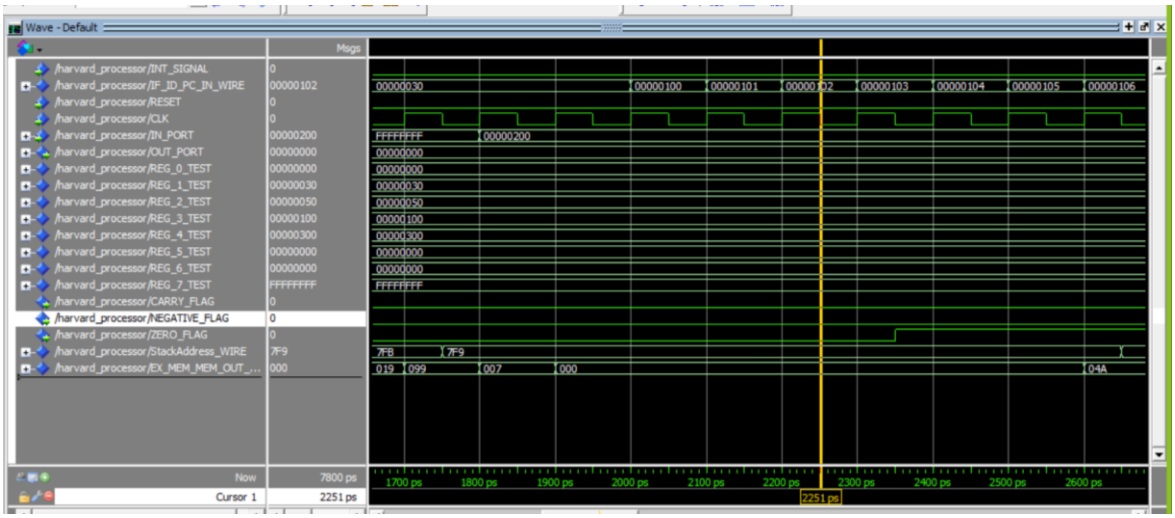


START INTERRUPT PROTOCOL => PUSH FLAGS AND PC (PC No Change)

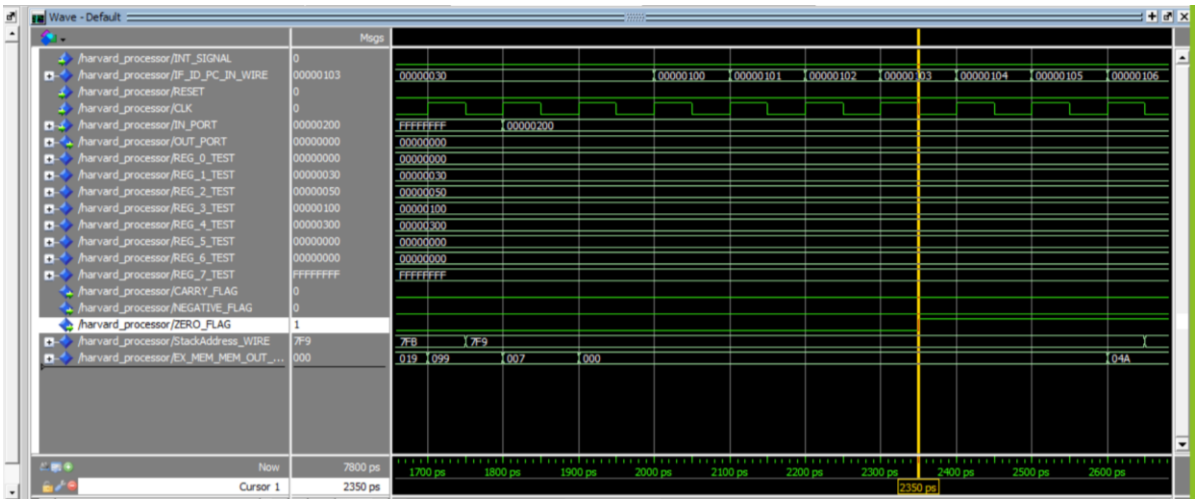
(INT routine)



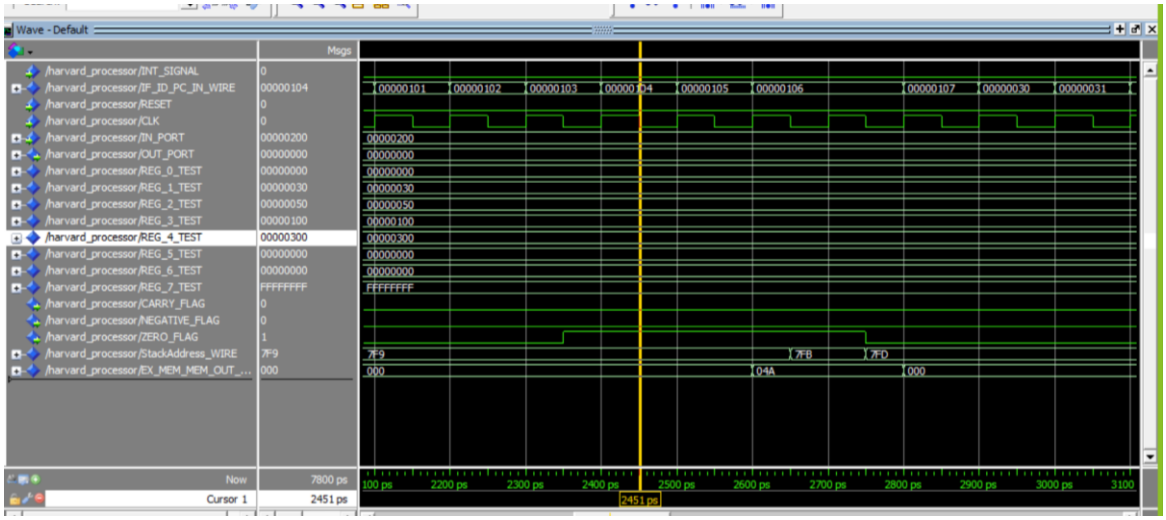
The Cursor above is at 2150 ps which means:
Fetch => AND R0,R0,R0
Decode => CLRC
Execute => BUBLE
Memory => BUBLE
WriteBack => BUBBLE



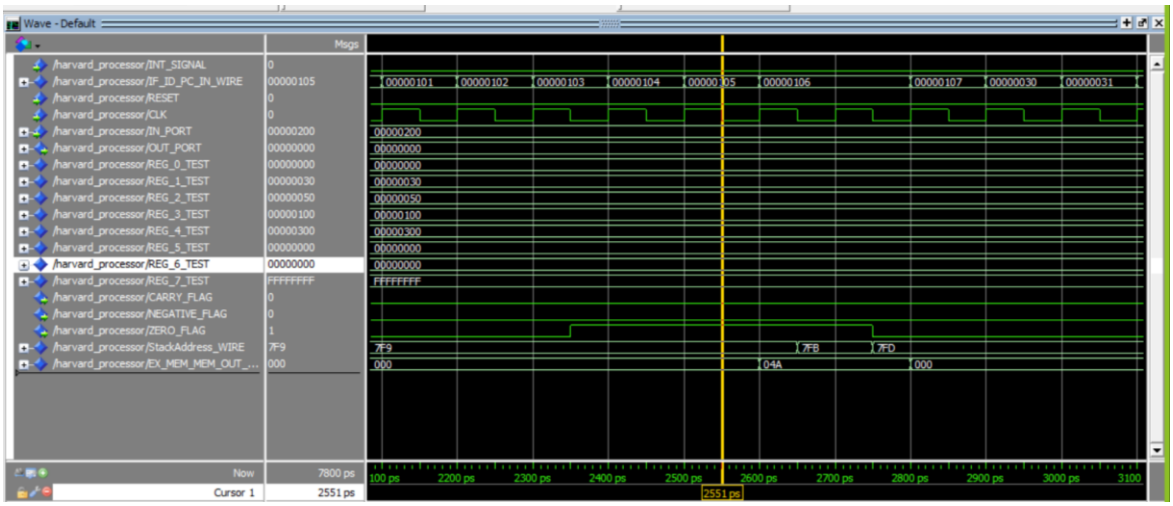
The Cursor above is at 2251 ps which means:
Fetch => out R6
Decode => AND R0,R0,R0
Execute => CLRC
Memory => BUBLE
WriteBack => BUBBLE



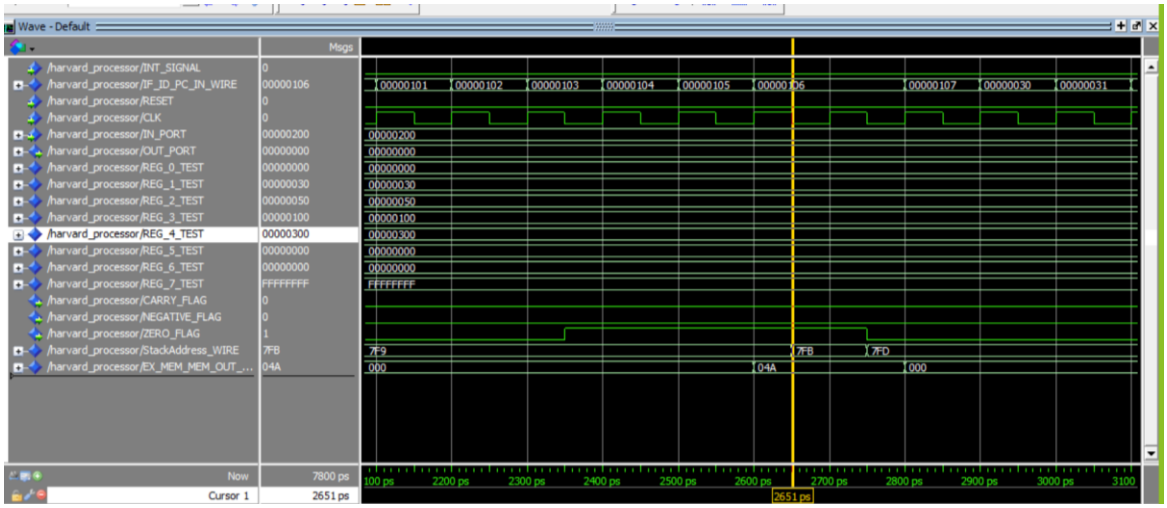
The Cursor above is at 2350 ps which means:
Fetch => rti
Decode => out R6
Execute => AND R0,R0,R0
Memory => CLRC
WriteBack => BUBBLE



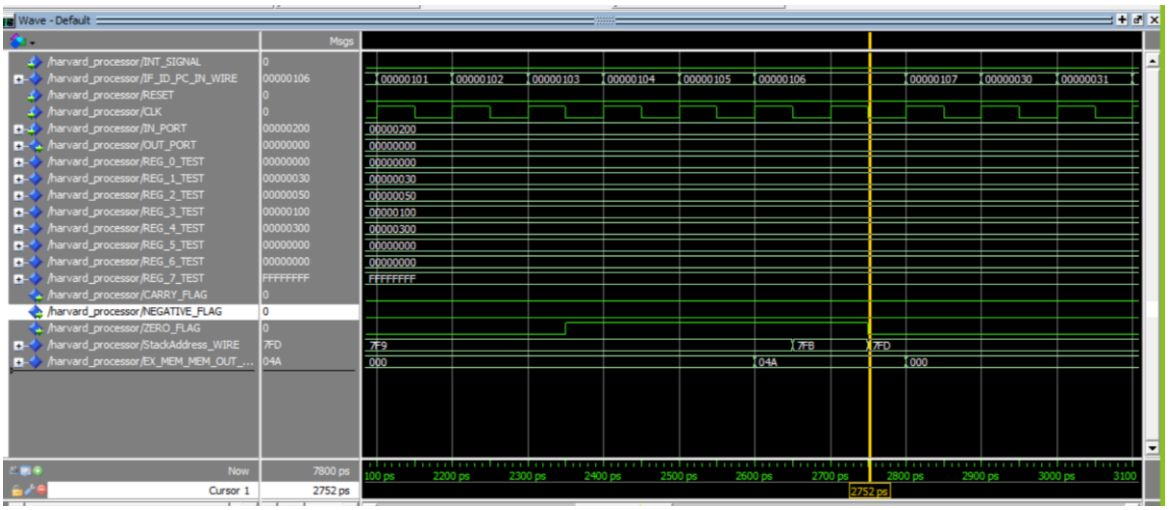
The Cursor above is at 2451 ps which means:
Fetch => NOP
Decode => rti
Execute => out R6
Memory => AND R0,R0,R0
WriteBack => CLRC



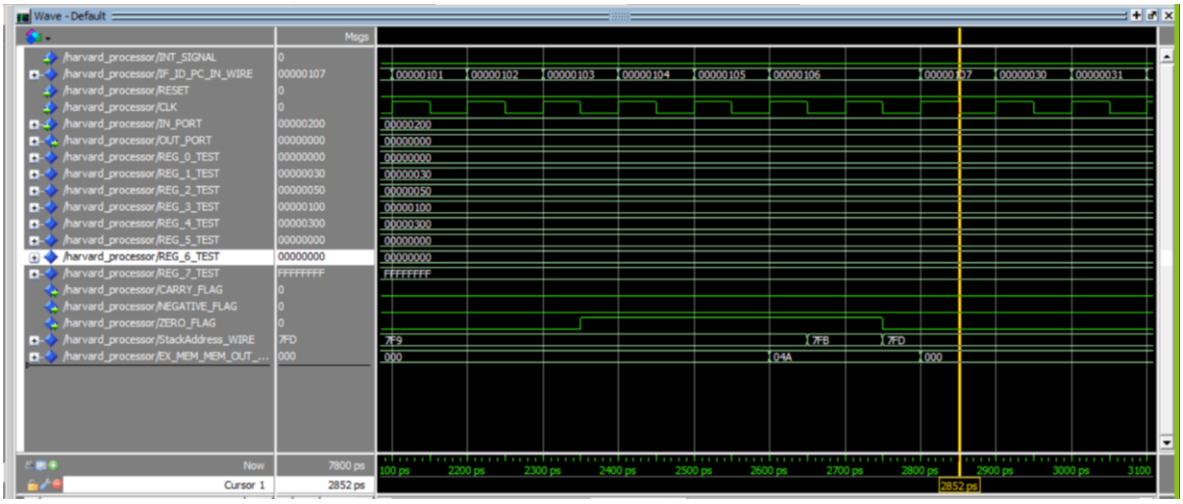
The Cursor above is at 2551 ps which means:
Fetch => NOP
Decode => NOP
Execute => rti
Memory => out R6
WriteBack => AND R0,R0,R0



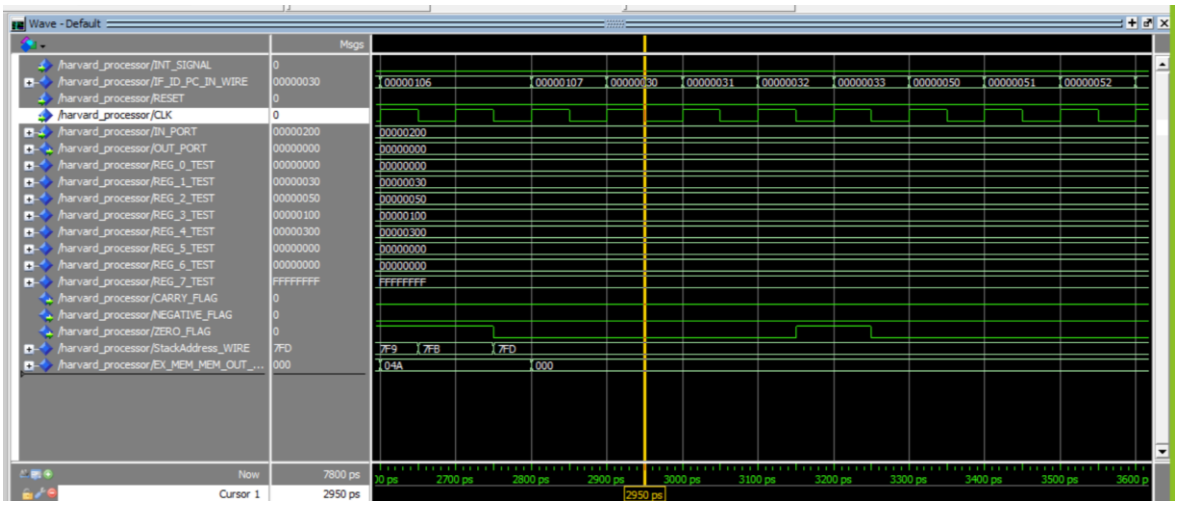
The Cursor above is at 2651 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => rti
WriteBack => out R6



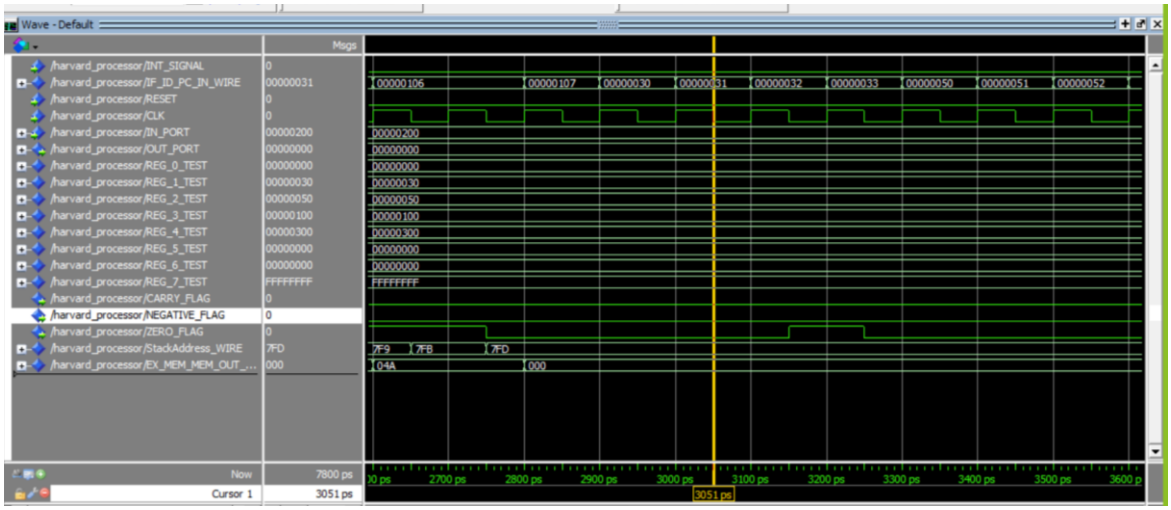
The Cursor above is at 2752 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => rti
WriteBack => rti



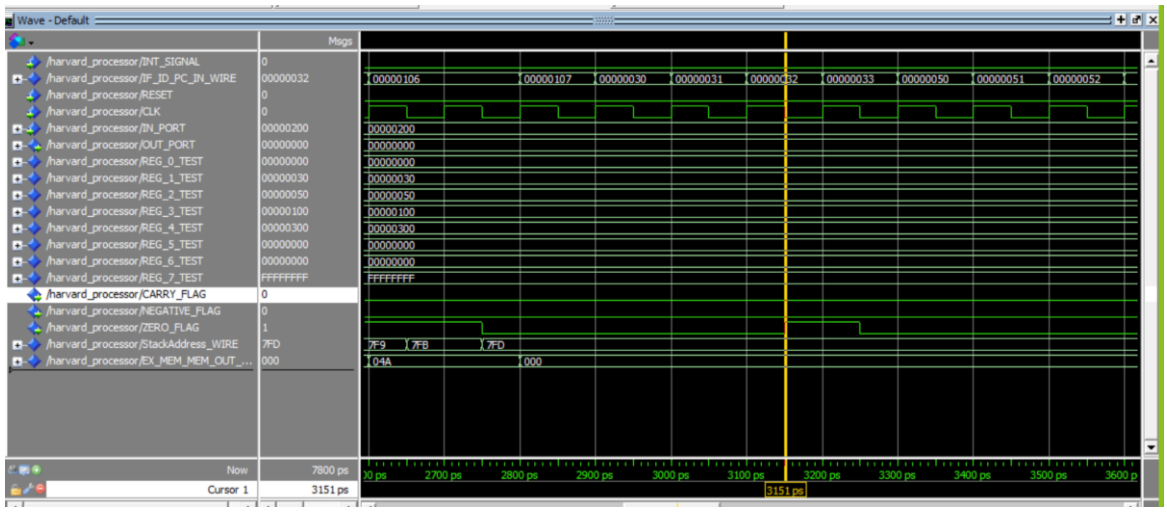
The Cursor above is at 2852 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => NOP
WriteBack => rti



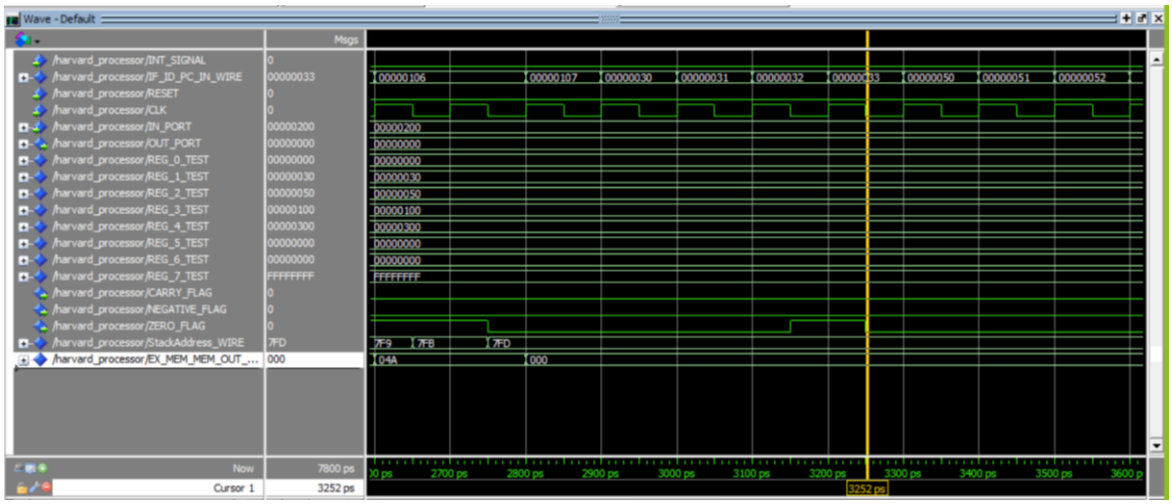
The Cursor above is at 2950 ps which means:
Fetch => AND R1,R5,R5 PC=30
Decode => NOP
Execute => NOP
Memory => NOP
WriteBack => NOP



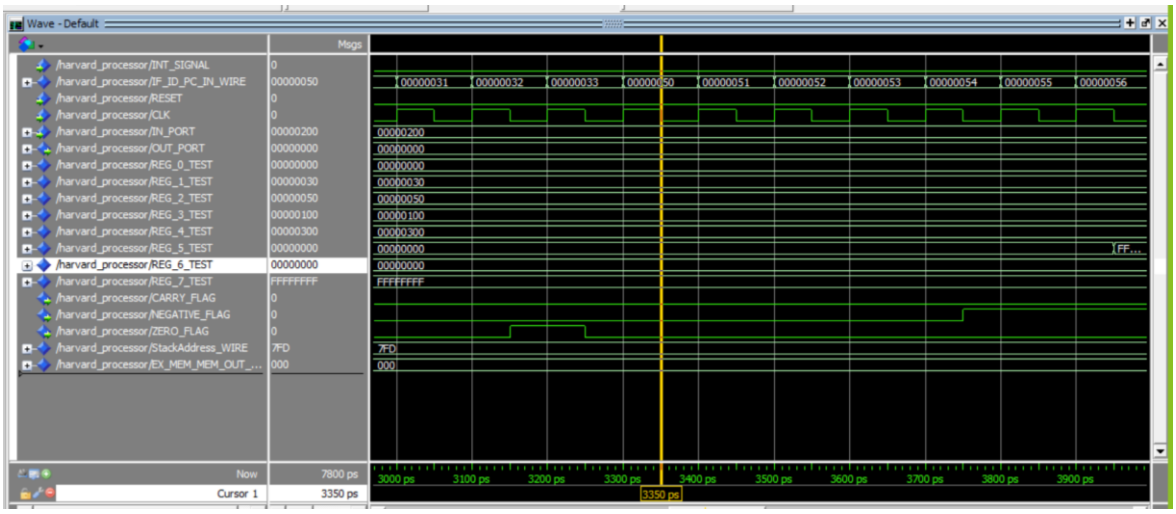
The Cursor above is at 3051 ps which means:
Fetch => JZ R2
Decode => AND R1,R5,R5
Execute => NOP
Memory => NOP
WriteBack => NOP



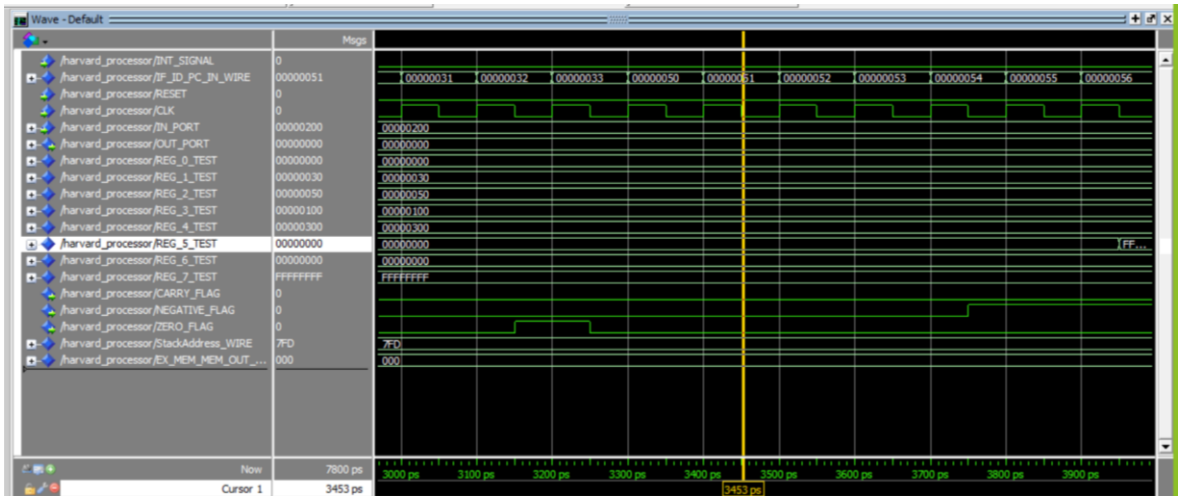
The Cursor above is at 3151 ps which means:
Fetch => INC R7
Decode => JZ R2
Execute => AND R1,R5,R5 R5=00000000 AND ZERO FLAG=1
Memory => NOP
WriteBack => NOP



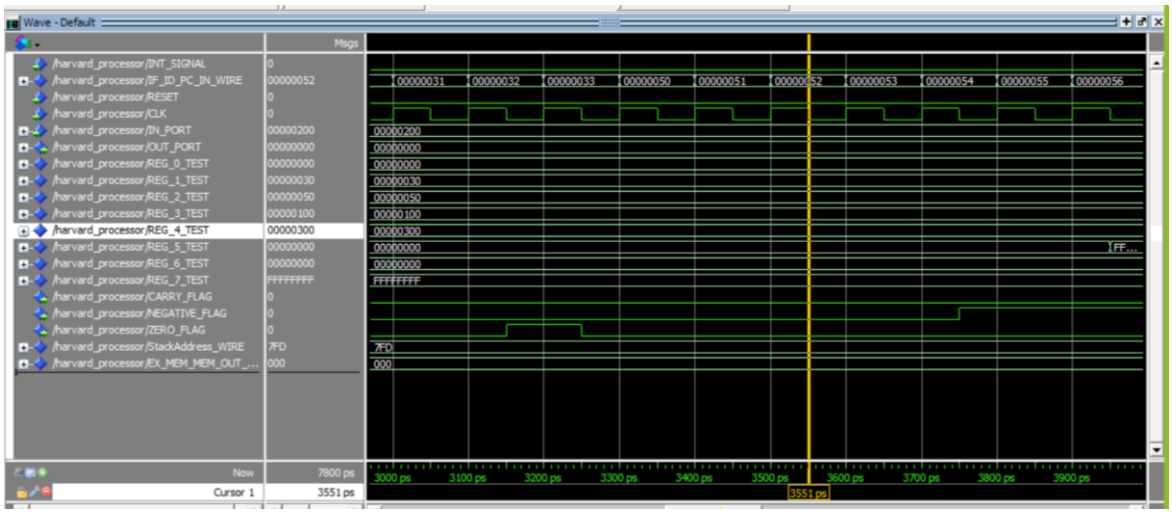
The Cursor above is at 3252 ps which means:
Fetch => NOP
Decode => INC R7
Execute => JZ R2
Memory => AND R1,R5,R5
WriteBack => NOP



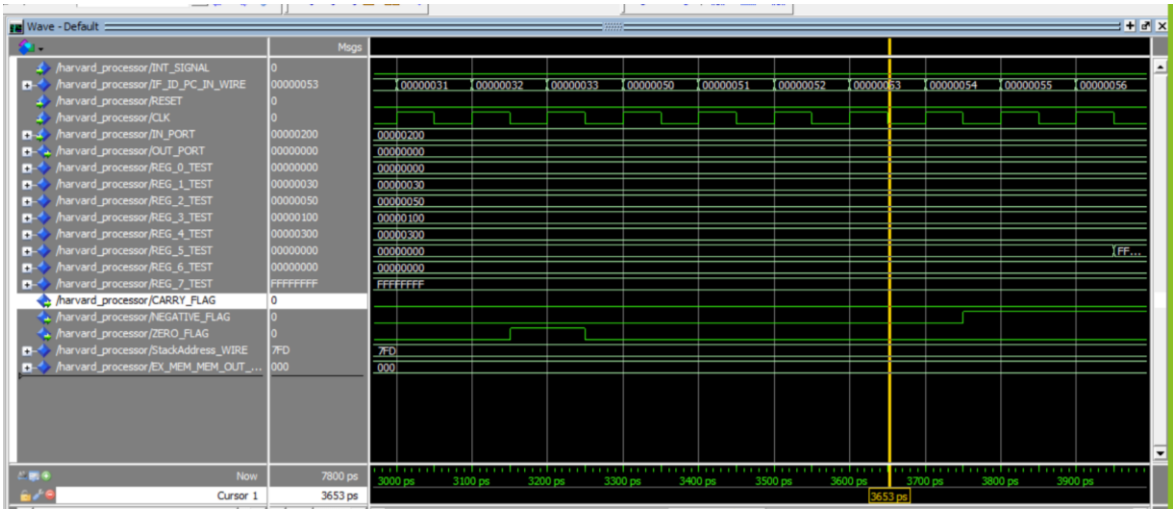
The Cursor above is at 3350 ps which means:
Fetch => JZ R1 PC=50
Decode => BUBLE
Execute => BUBLE
Memory => JZ R2
WriteBack => AND R1,R5,R5



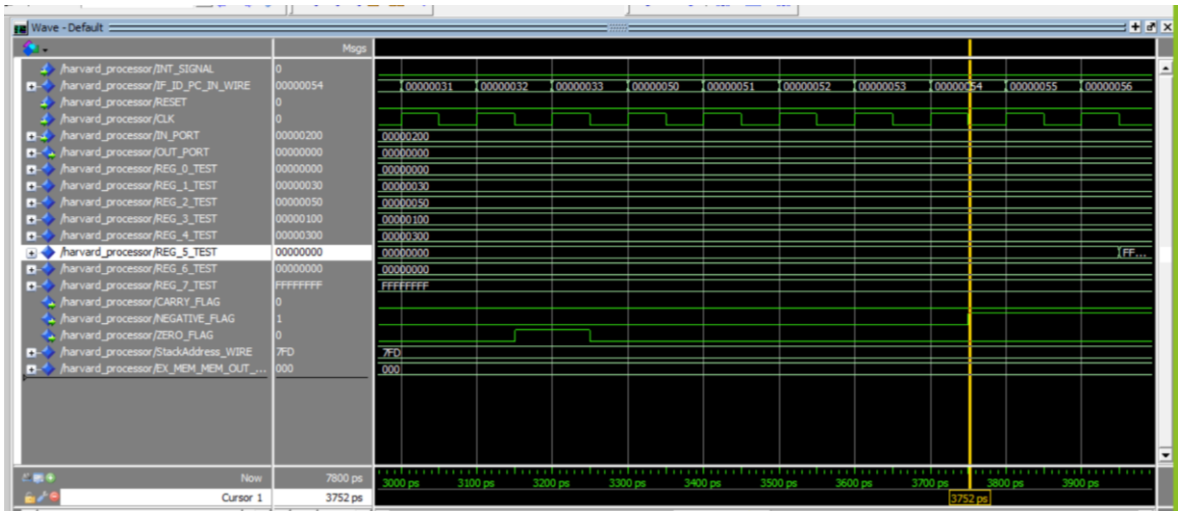
The Cursor above is at 3453 ps which means:
Fetch => JC R3
Decode => JZ R1
Execute => BUBLE
Memory => BUBLE
WriteBack => JZ R2



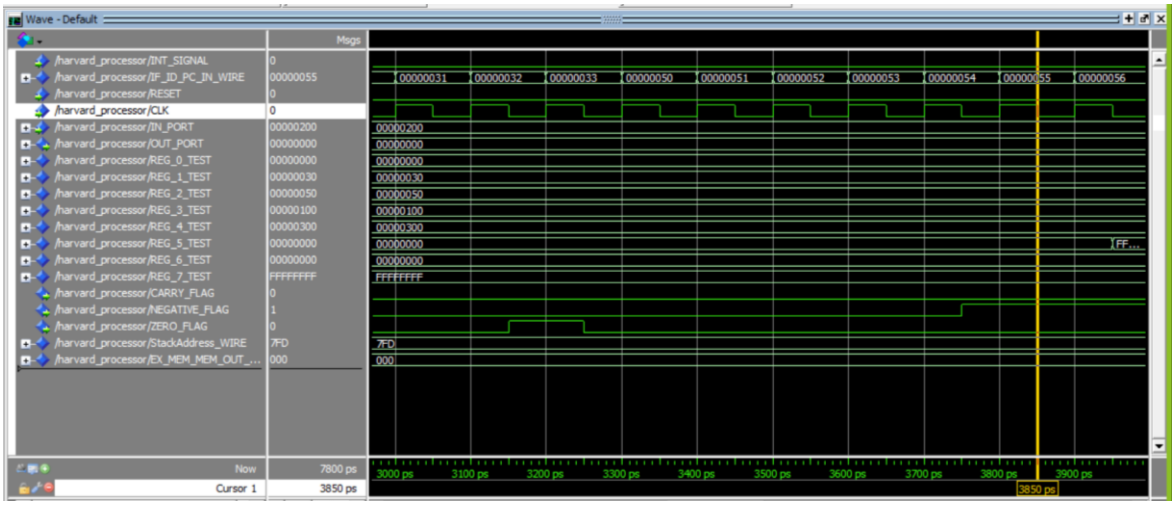
The Cursor above is at 3551 ps which means:
Fetch => NOT R5
Decode => JC R3
Execute => JZ R1
Memory => BUBLE
WriteBack => BUBLE



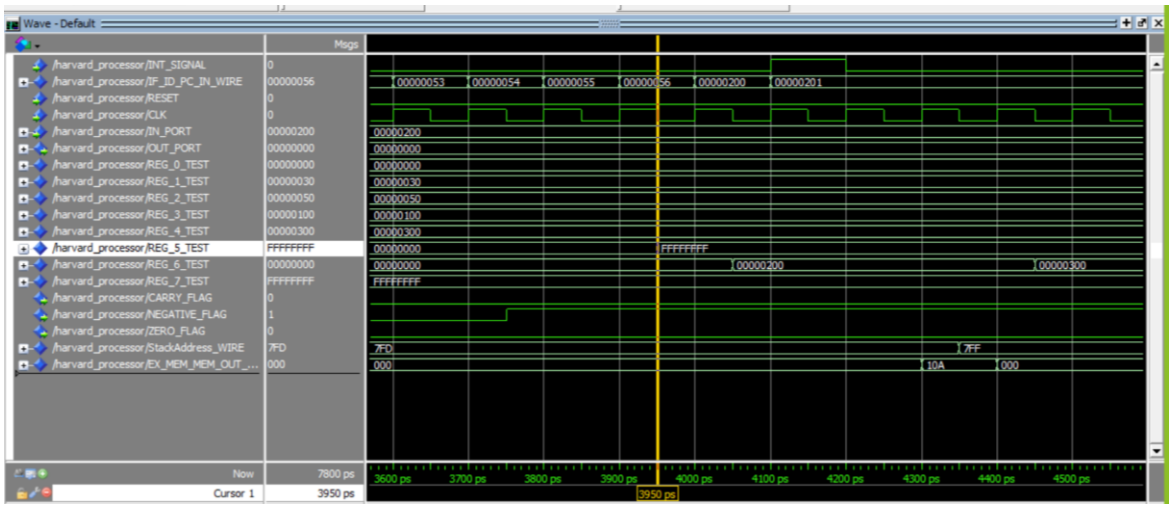
The Cursor above is at 3653 ps which means:
Fetch => IN R6
Decode => NOT R5
Execute => JC R3
Memory => JZ R1
WriteBack => BUBLE



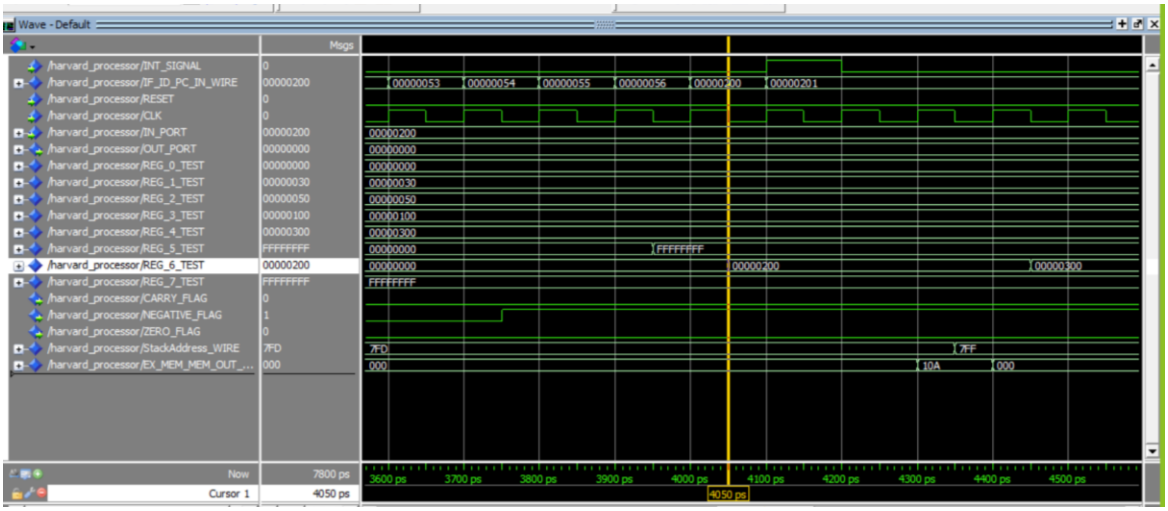
The Cursor above is at 3752 ps which means:
Fetch => JN R6
Decode => IN R6
Execute => NOT R5 ZERO FLAG=1
Memory => JC R3
WriteBack => JZ R1



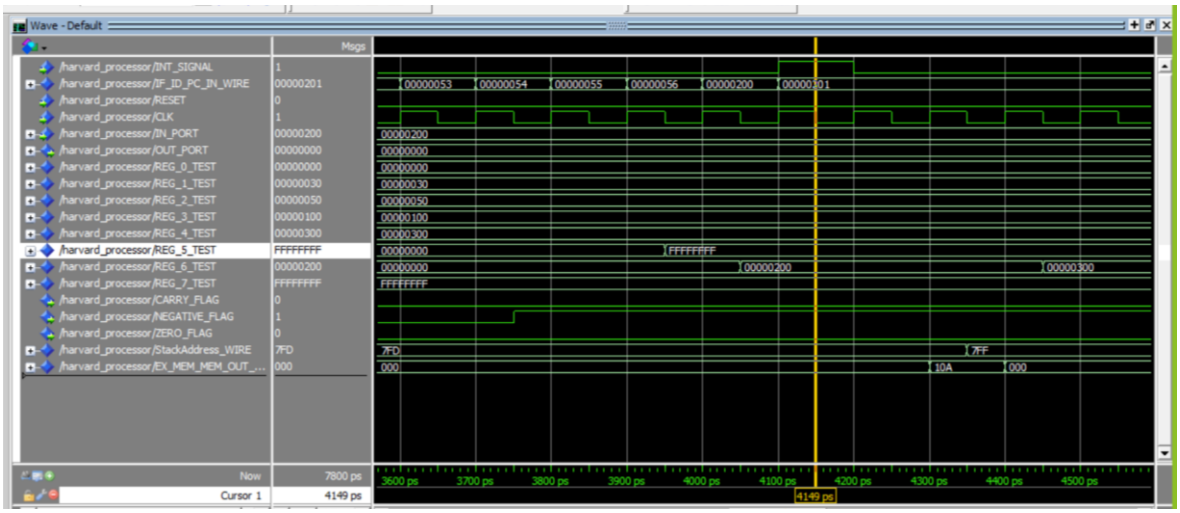
The Cursor above is at 3850 ps which means:
Fetch => INC R1
Decode => JN R6
Execute => IN R6
Memory => NOT R5
WriteBack => JC R3



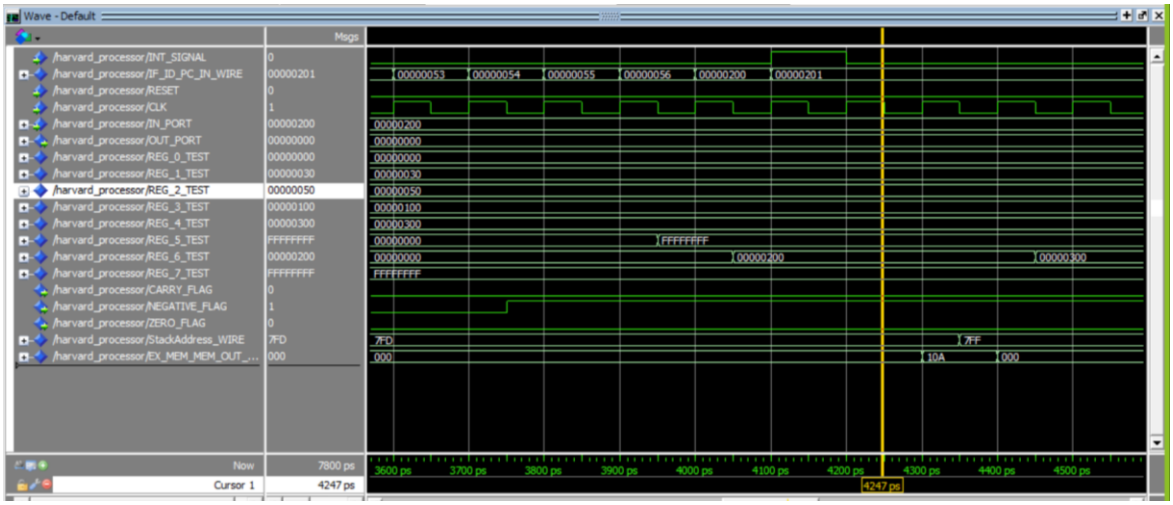
The Cursor above is at 3950 ps which means:
Fetch => NOP
Decode => INC R1
Execute => JN R6
Memory => IN R6
WriteBack => NOT R5 R5=FFFFFFFF



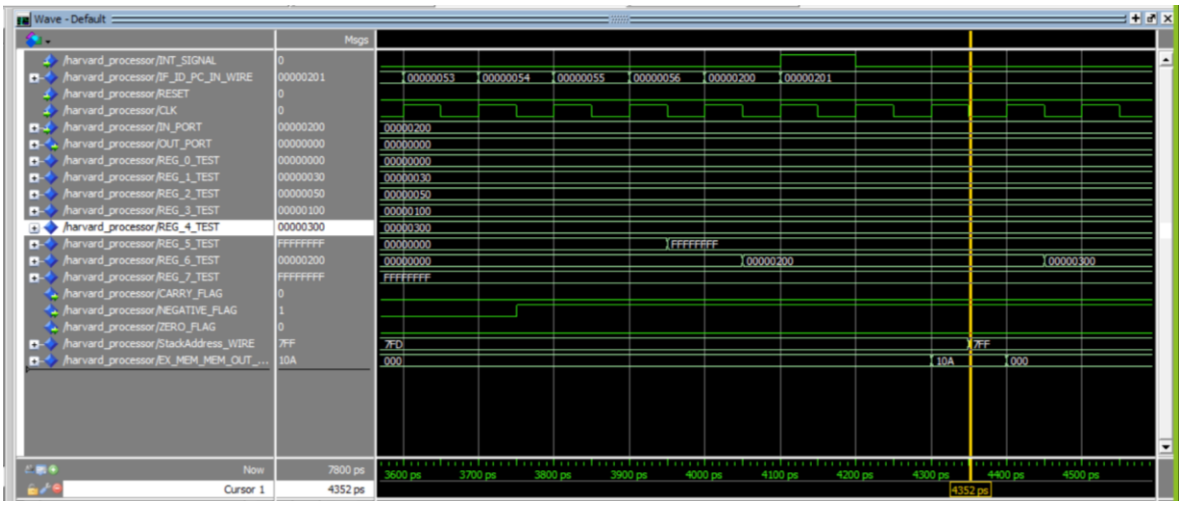
The Cursor above is at 4050 ps which means:
Fetch => POP R6 PC=200
Decode => BUBLE stall pipe bec of jmp
Execute => BUBLE stall pipe bec of jmp
Memory => JN R6
WriteBack => IN R6 R6=00000200



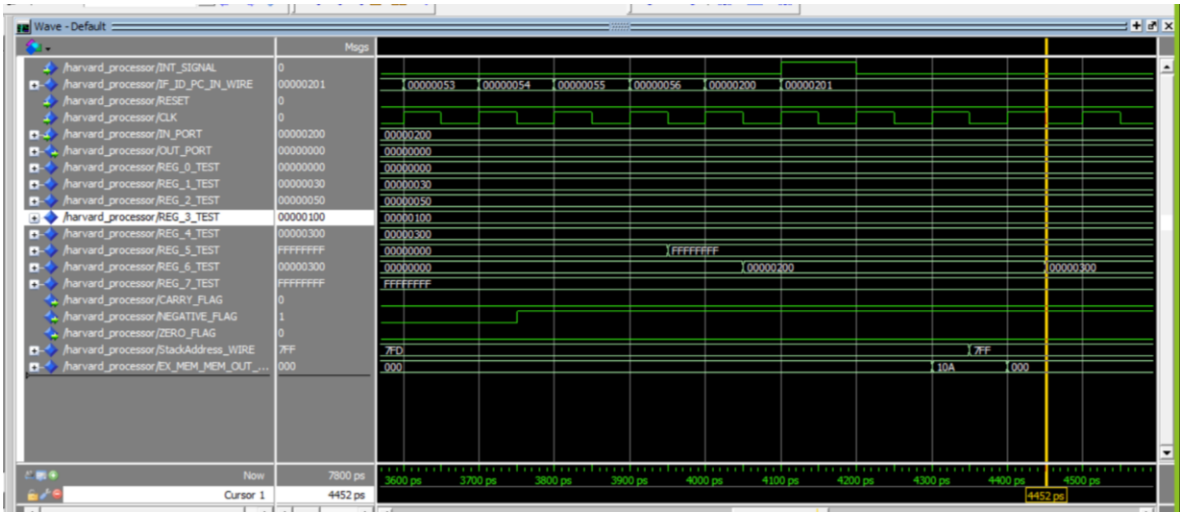
The Cursor above is at 4149 ps which means: INT =1
Fetch => BUBLE BECAUSE OF INTERRUPT PROTOCOL
Decode => POP R6
Execute => BUBLE
Memory => BUBLE
WriteBack => JN R6



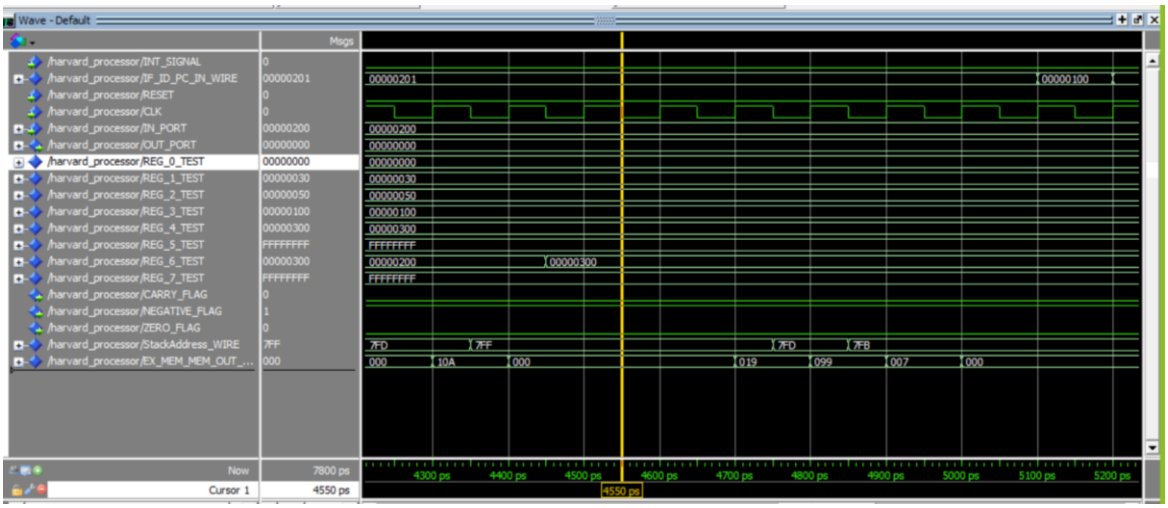
The Cursor above is at 4247 ps which means:
Fetch => BUBLE
Decode => Call R6
Execute => POP R6
Memory => BUBLE
WriteBack => BUBLE



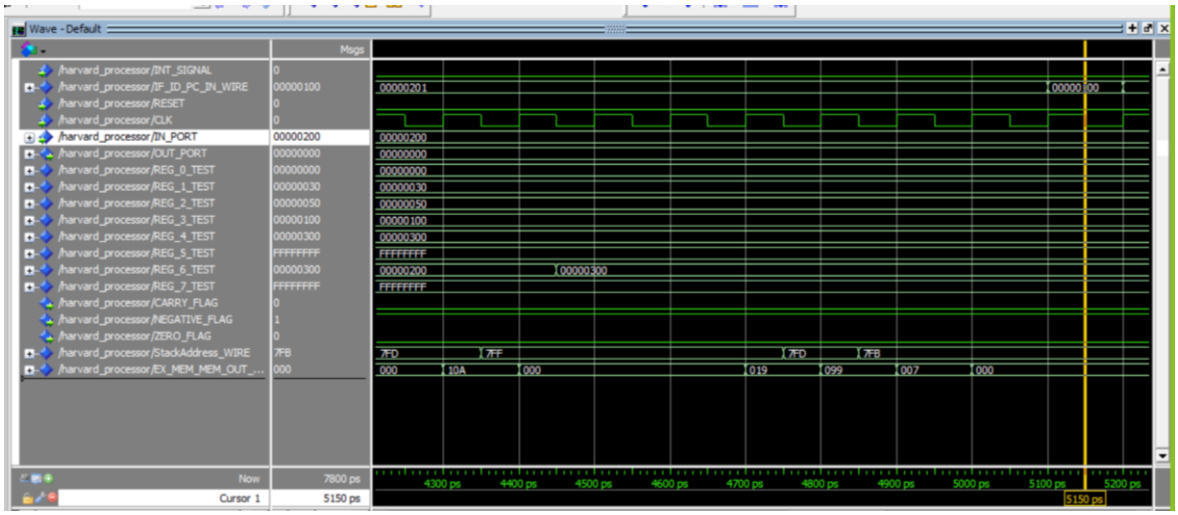
The Cursor above is at 4352 ps which means:
Fetch => BUBBLE
Decode => BUBBLE
Execute => BUBLE
Memory => POP R6 SP=7FF
WriteBack => BUBLE



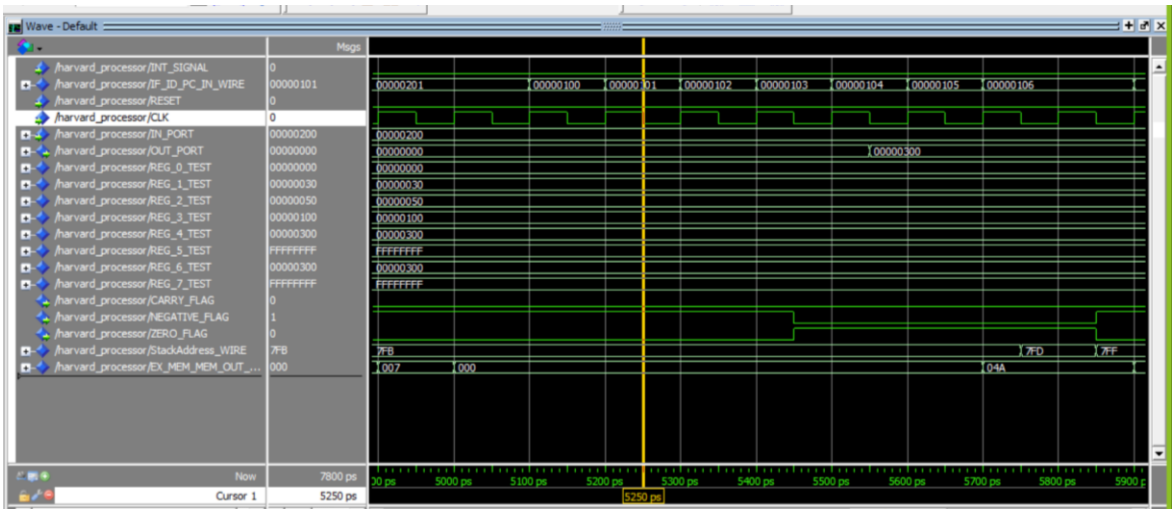
The Cursor above is at 4452 ps which means:
Fetch => BUBBLE
Decode => BUBBLE
Execute => BUBBLE
Memory => BUBBLE
WriteBack => POP R6 R6=00000300



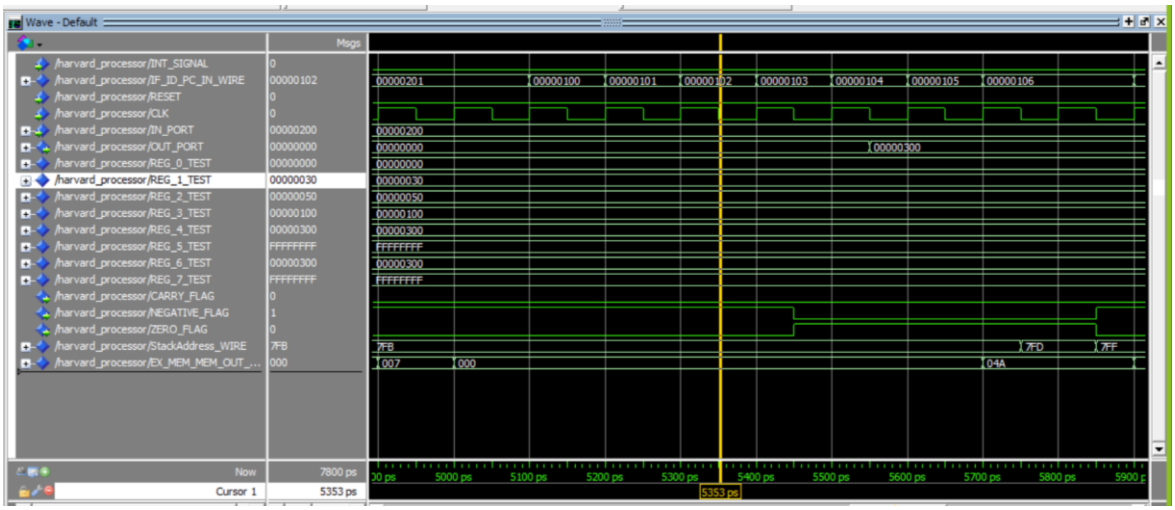
START INTERUPT PROTCOL => PUSH FLAGS AND PC



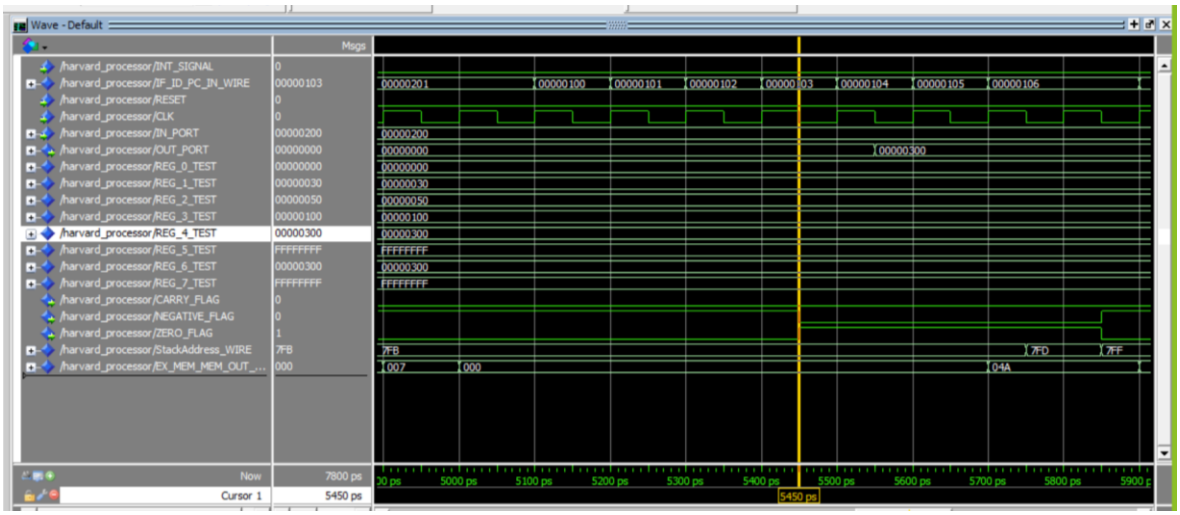
The Cursor above is at 2150 ps which means:
Fetch => CLRC
Decode => BUBBLE
Execute => BUBBLE
Memory => BUBBLE
WriteBack => BUBBLE



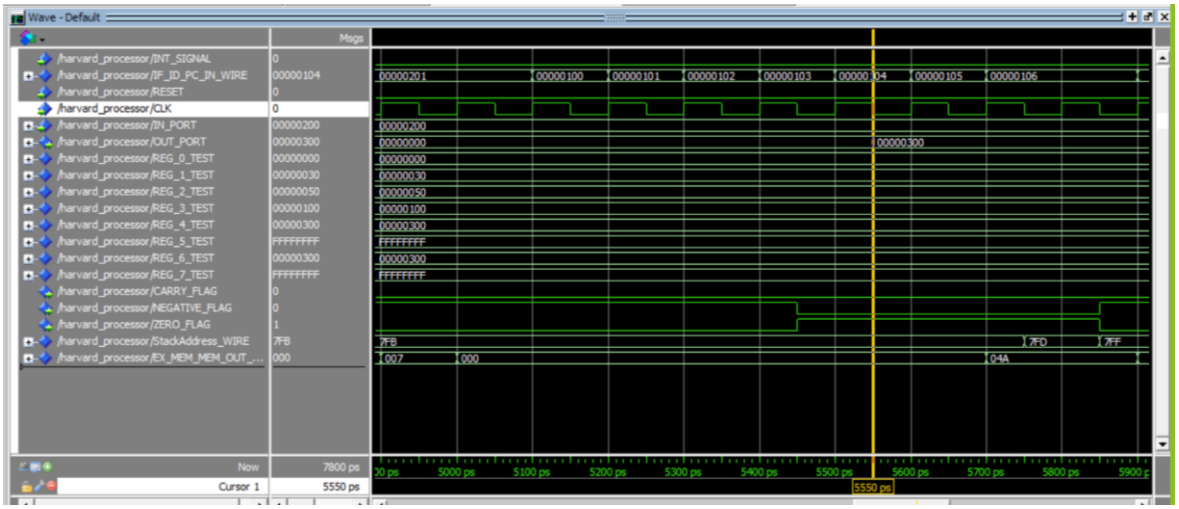
The Cursor above is at 5250 ps which means:
Fetch => AND R0,R0,R0
Decode => CLRC
Execute => BUBLE
Memory => BUBLE
WriteBack => BUBBLE



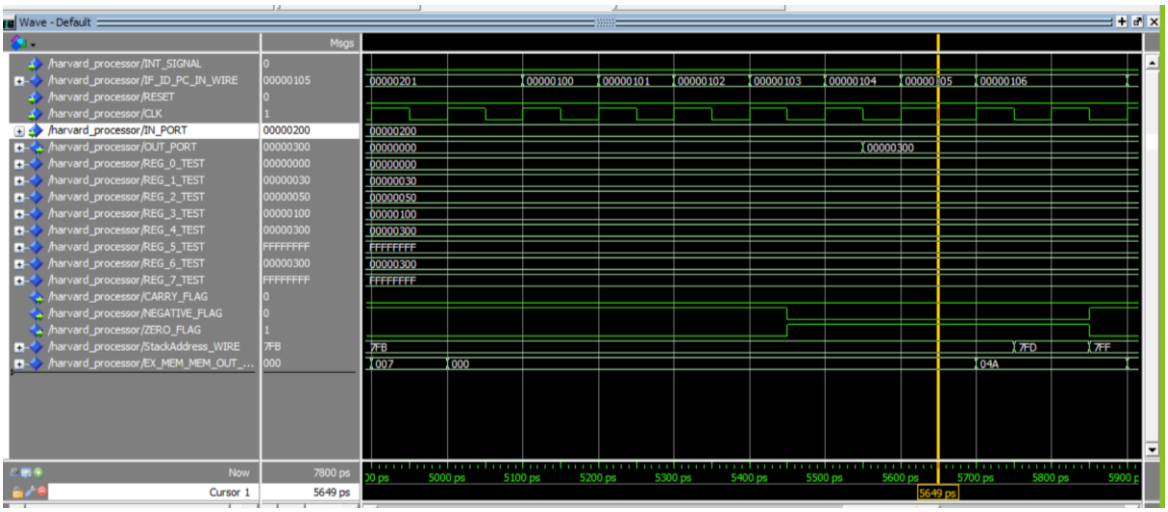
The Cursor above is at 5353 ps which means:
Fetch => out R6
Decode => AND R0,R0,R0
Execute => CLRC
Memory => BUBLE
WriteBack => BUBBLE



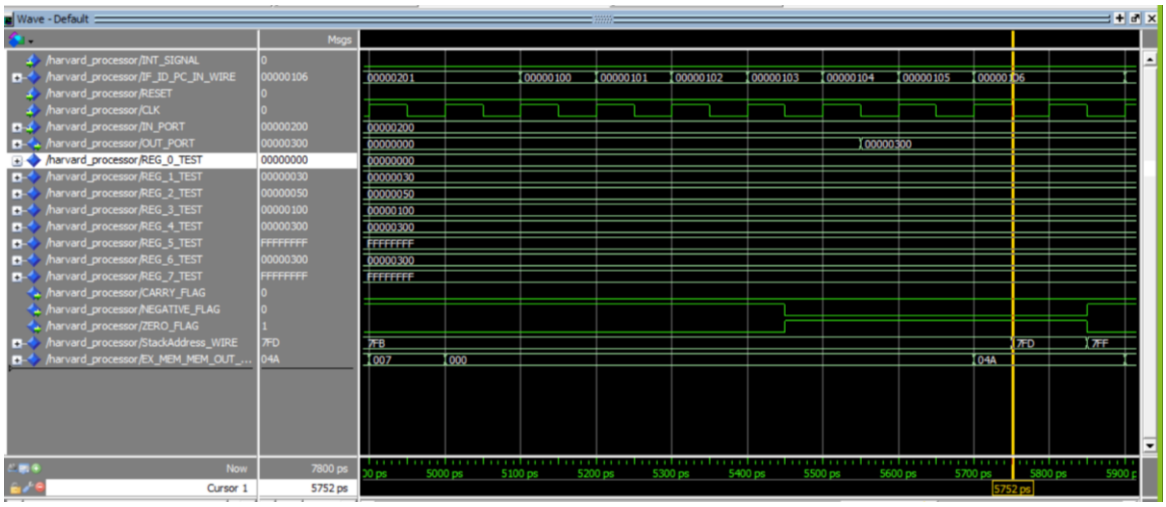
The Cursor above is at 5450 ps which means:
Fetch => RTI
Decode => OUT R6
Execute => AND R0,R0,R0
Memory => CLRC
WriteBack => BUBBLE



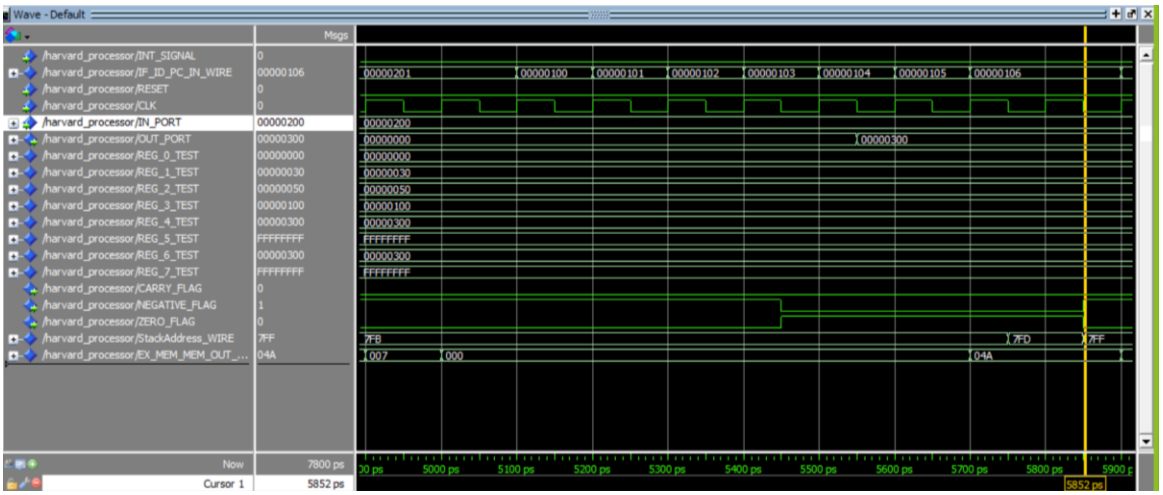
The Cursor above is at 5550 ps which means:
Fetch => NOP
Decode => RTI
Execute => OUT R6
Memory => AND R0,R0,R0
WriteBack => CLRC



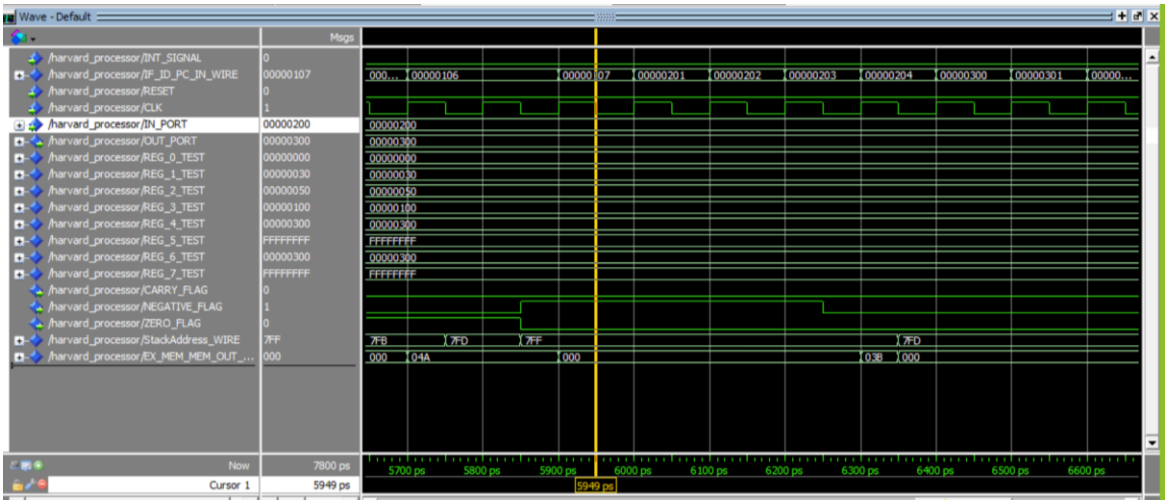
The Cursor above is at 5649 ps which means:
Fetch => NOP
Decode => NOP
Execute => RTI
Memory => OUT R6
WriteBack => AND R0,R0,R0



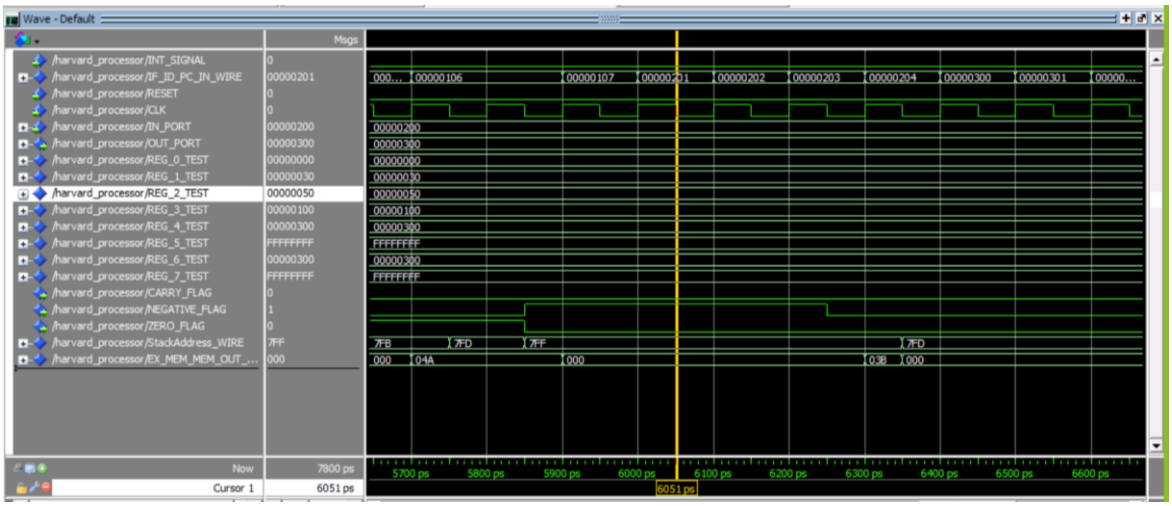
The Cursor above is at 5752 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => RTI
WriteBack => OUT R6



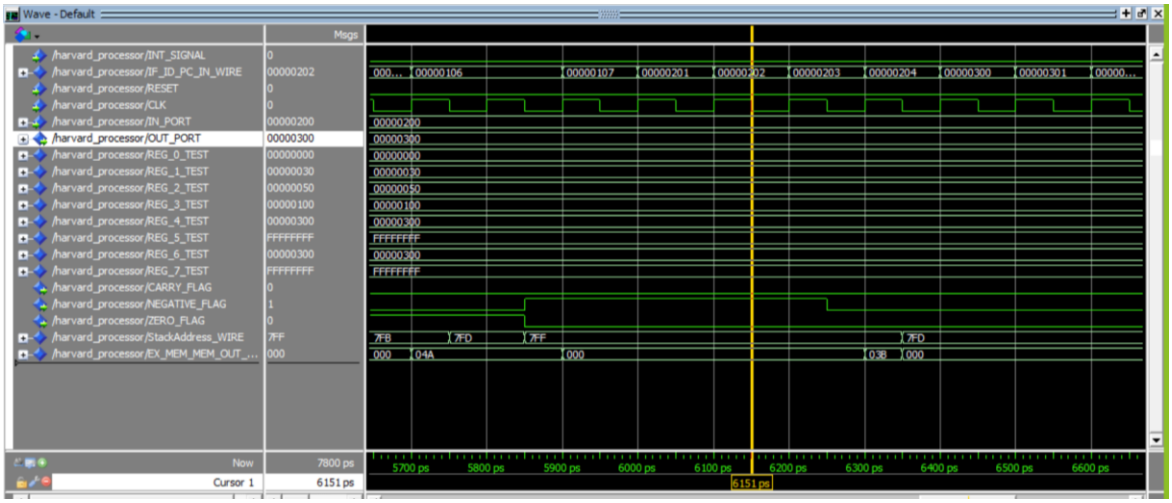
The Cursor above is at 5852 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => RTI
WriteBack => RTI



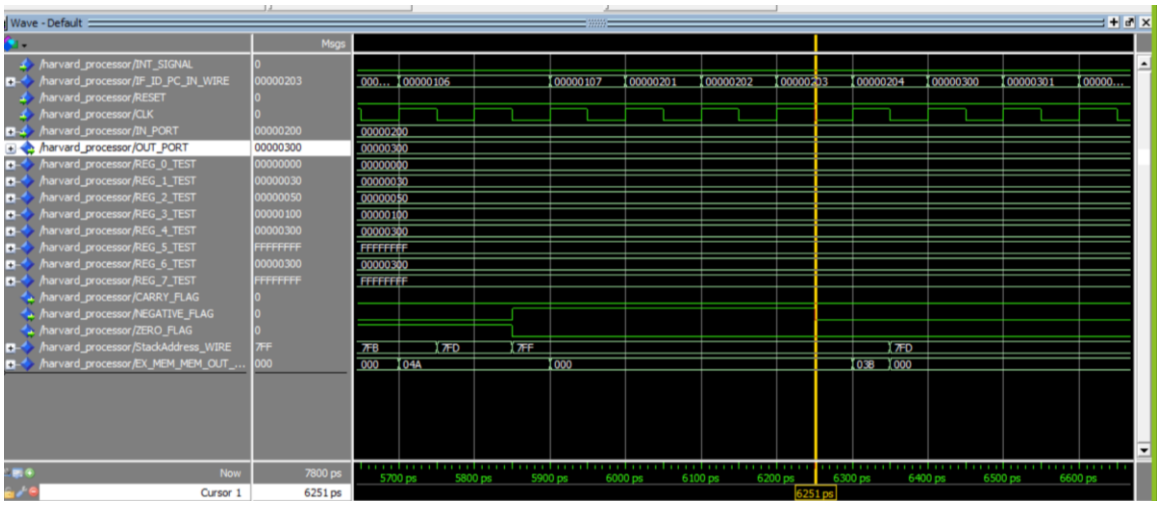
The Cursor above is at 5949 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => NOP
WriteBack => RTI



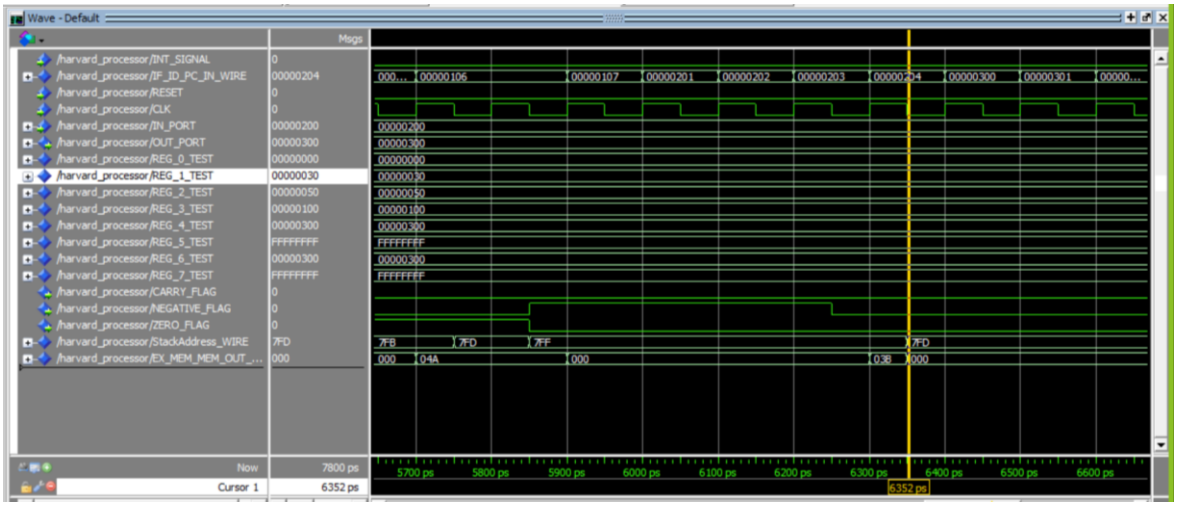
The Cursor above is at 6051 ps which means:
Fetch => Call R6
Decode => NOP
Execute => NOP
Memory => NOP
WriteBack => NOP



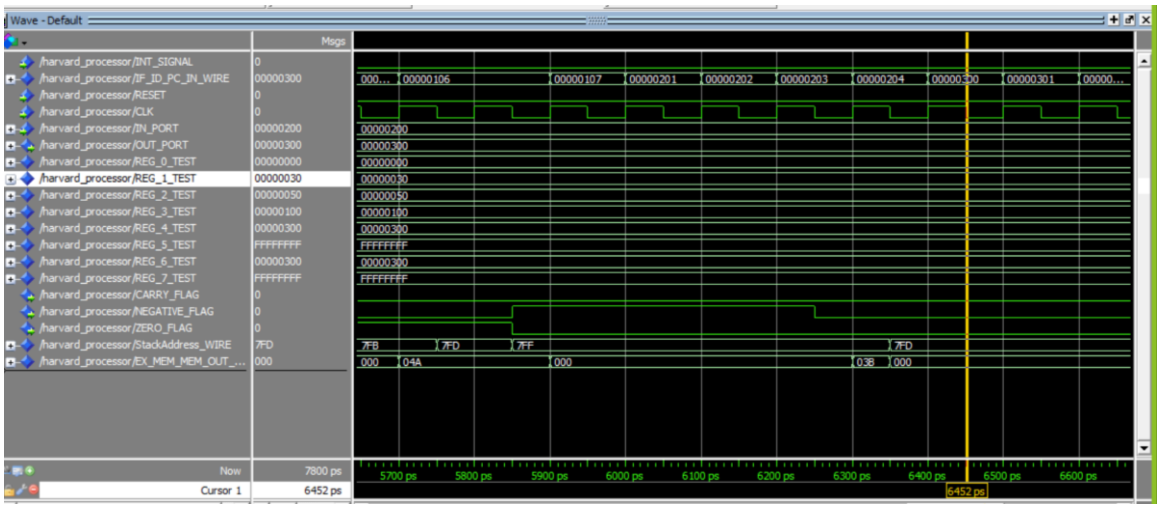
The Cursor above is at 6151 ps which means:
Fetch => INC R6
Decode => Call R6
Execute => NOP
Memory => NOP
WriteBack => NOP



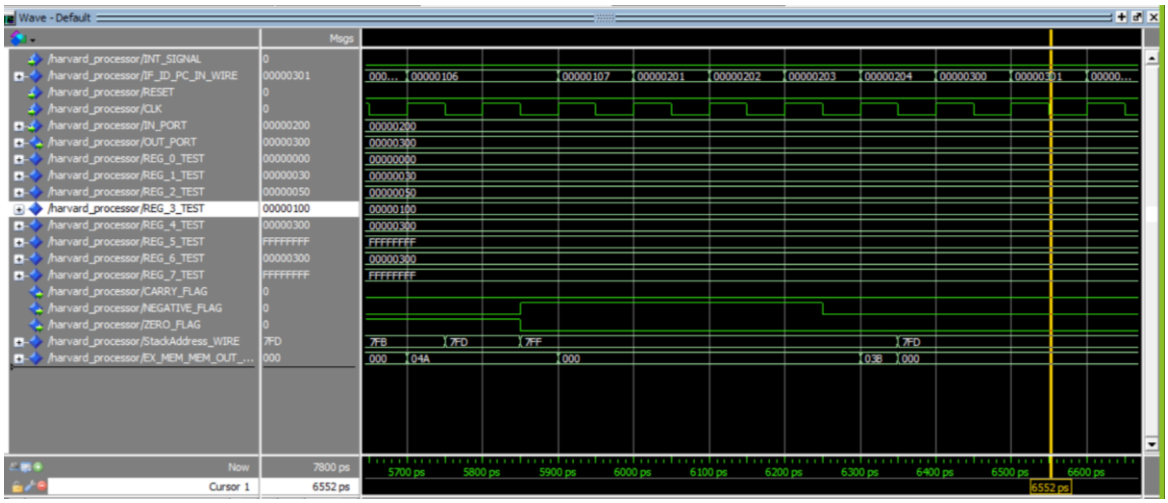
The Cursor above is at 6251 ps which means:
Fetch => NOP
Decode => INC R6
Execute => Call R6
Memory => NOP
WriteBack => NOP



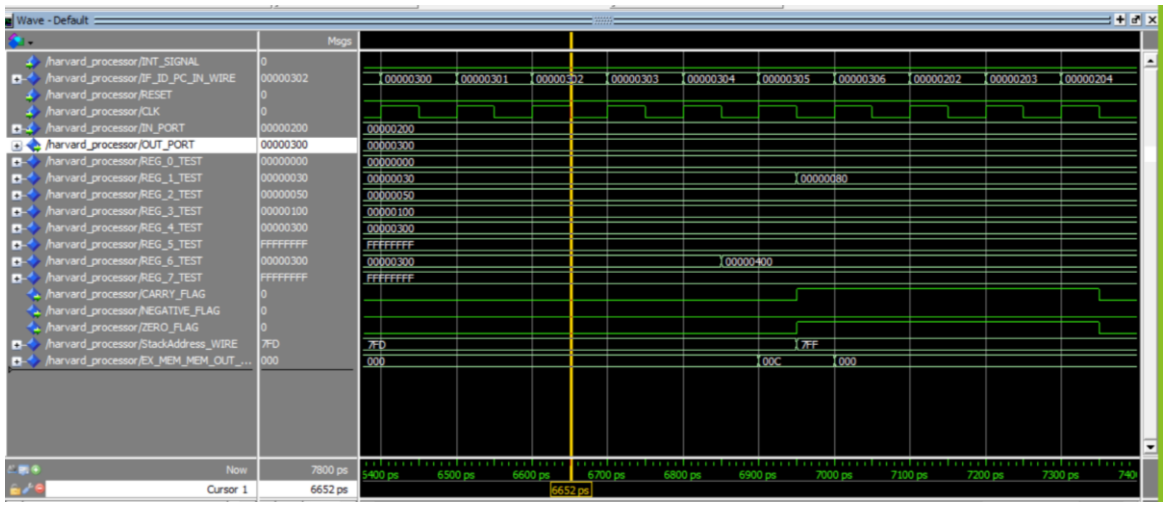
The Cursor above is at 6251 ps which means:
Fetch => NOP
Decode => NOP
Execute => INC R6
Memory => Call R6 SP=7FD
WriteBack => NOP



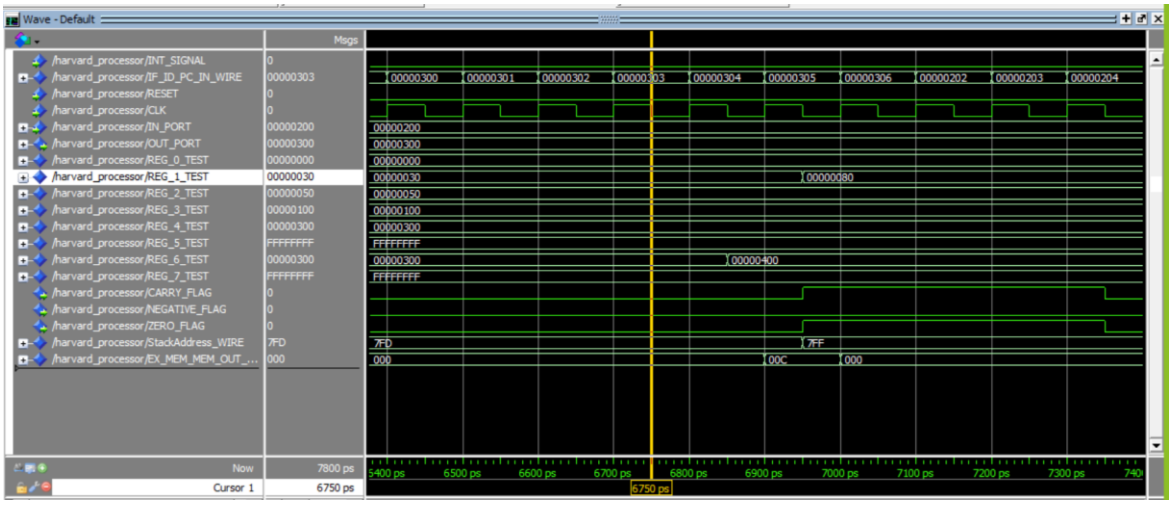
The Cursor above is at 6452 ps which means:
Fetch => Add R3,R6,R6
Decode => BUBLE FLUSH PIPE BEC OF CALL
Execute => BUBLE
Memory => BUBLE
WriteBack => Call R6



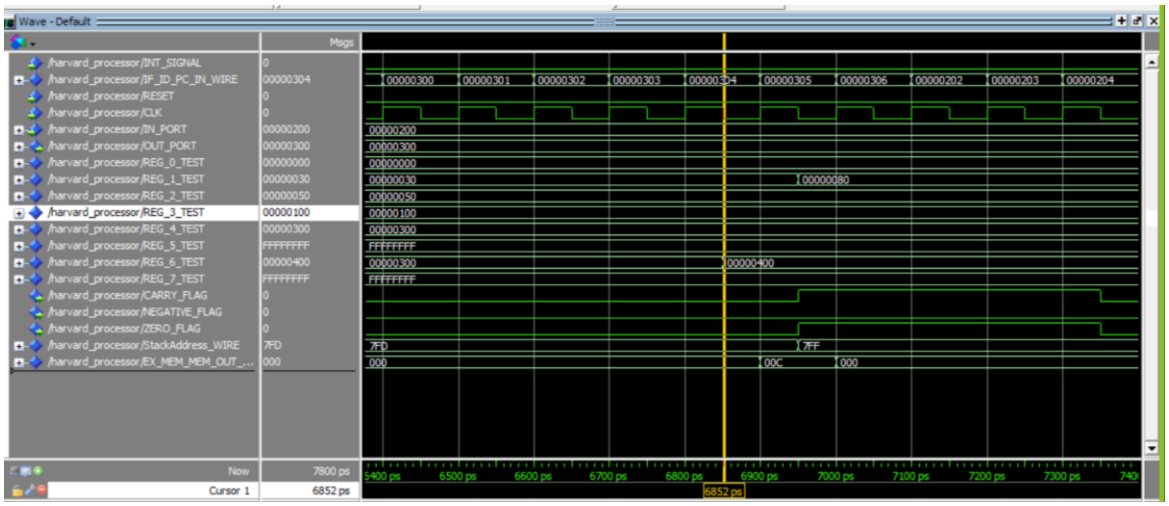
The Cursor above is at 6552 ps which means:
Fetch => Add R1,R2,R1
Decode => Add R3,R6,R6
Execute => BUBLE
Memory => BUBLE
WriteBack => BUBLE



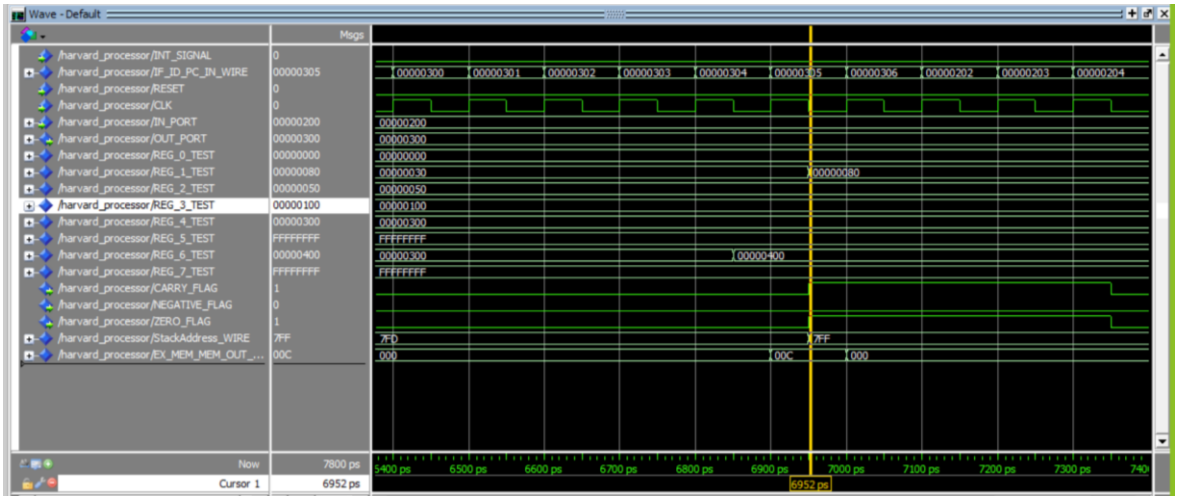
The Cursor above is at 6652 ps which means:
Fetch => RET
Decode => Add R1,R2,R1
Execute => Add R3,R6,R6
Memory => BUBLE
WriteBack => BUBLE



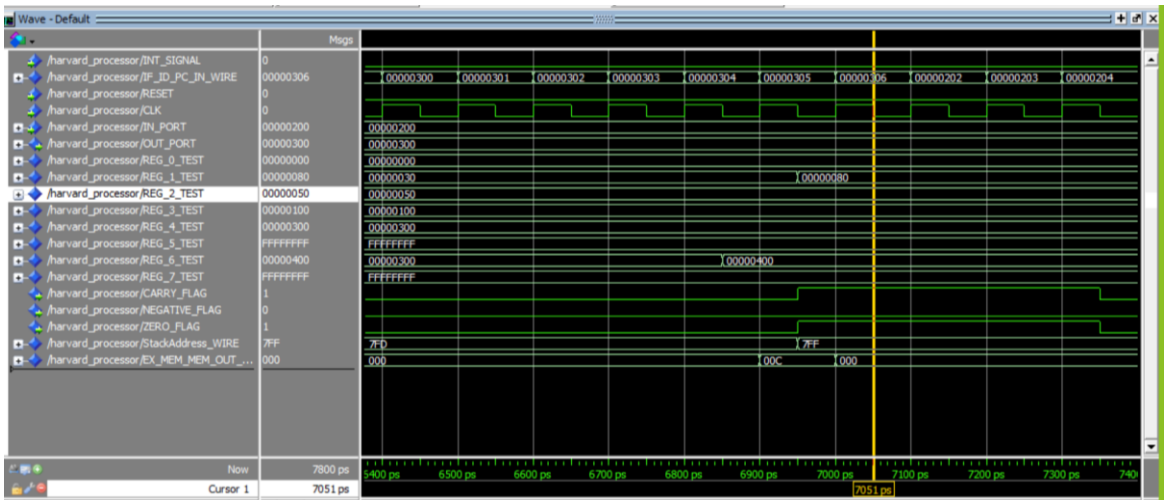
The Cursor above is at 6750 ps which means:
Fetch => INC R7
Decode => RET
Execute => Add R1,R2,R1 CARRY => 0 ZERO => 0 NEGATIVE =>0
Memory => Add R3,R6,R6
WriteBack => BUBLE



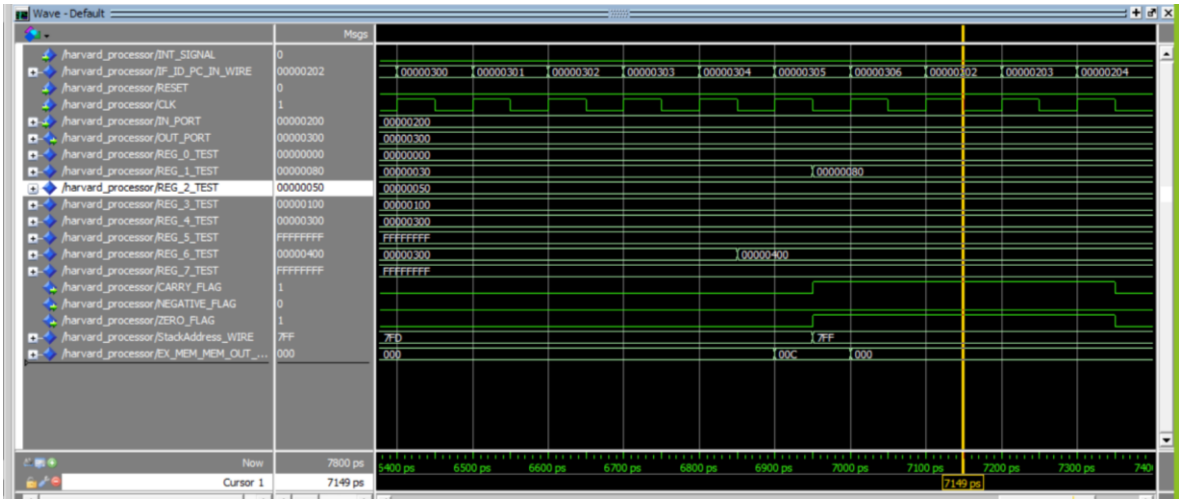
The Cursor above is at 6852 ps which means:
Fetch => NOP
Decode => INC R7
Execute => RET
Memory => Add R1,R2,R1
WriteBack => Add R3,R6,R6 R6 = 00000400



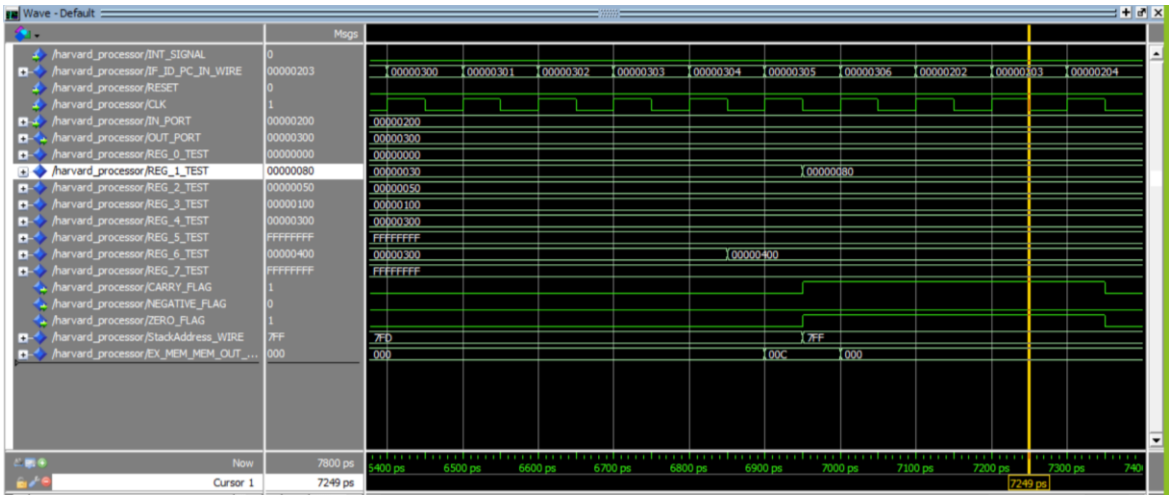
The Cursor above is at 6952 ps which means:
Fetch => NOP
Decode => NOP
Execute => INC R7
Memory => RET
WriteBack => Add R1,R2,R1 R1 = 00000080



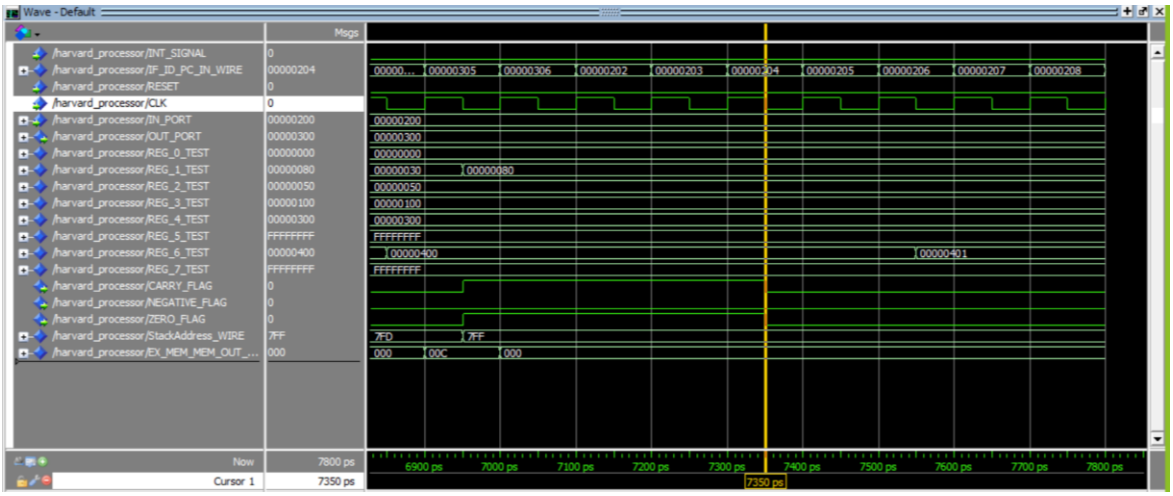
The Cursor above is at 7051 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => INC R7
WriteBack => RET



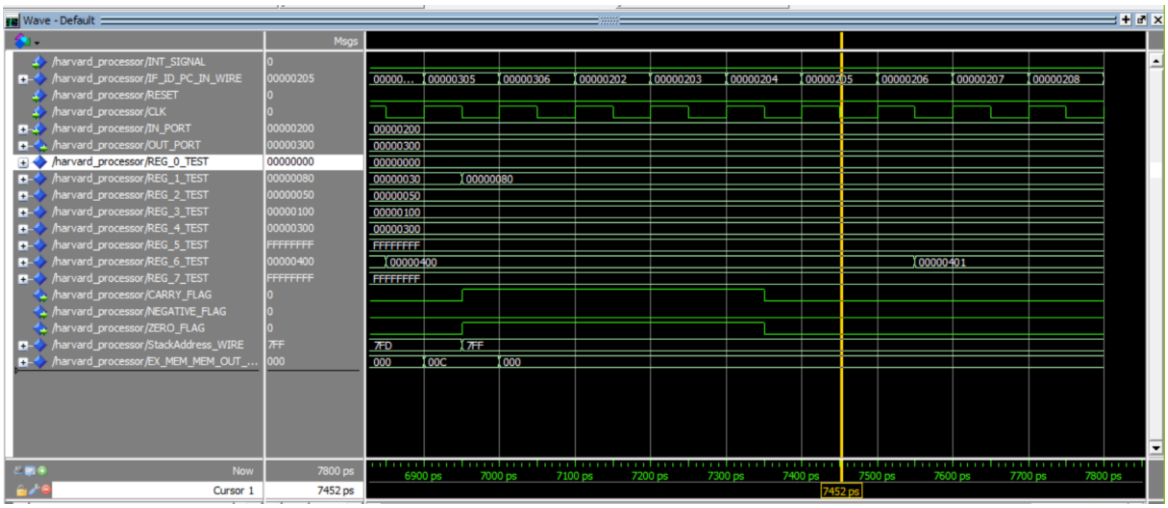
The Cursor above is at 7149 ps which means:
Fetch => INC R6 PC=202 BECAUSE OF RETURN
Decode => BUBBLE
Execute => BUBBLE
Memory => BUBBLE
WriteBack => BUBBLE



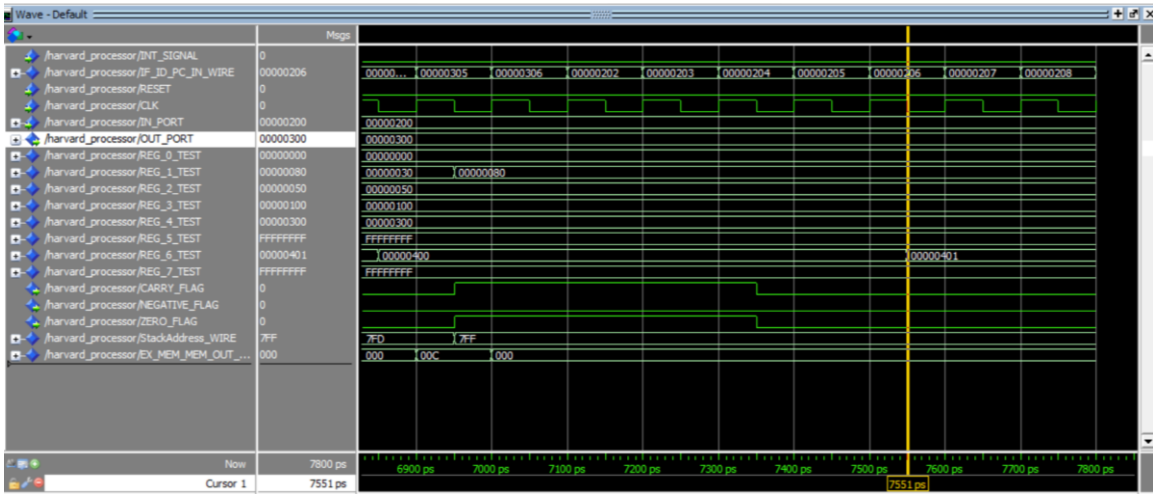
The Cursor above is at 7249 ps which means:
Fetch => NOP
Decode => INC R6
Execute => BUBBLE
Memory => BUBBLE
WriteBack => BUBBLE



The Cursor above is at 7350 ps which means:
Fetch => NOP
Decode => NOP
Execute => INC R6
Memory => BUBBLE
WriteBack => BUBBLE

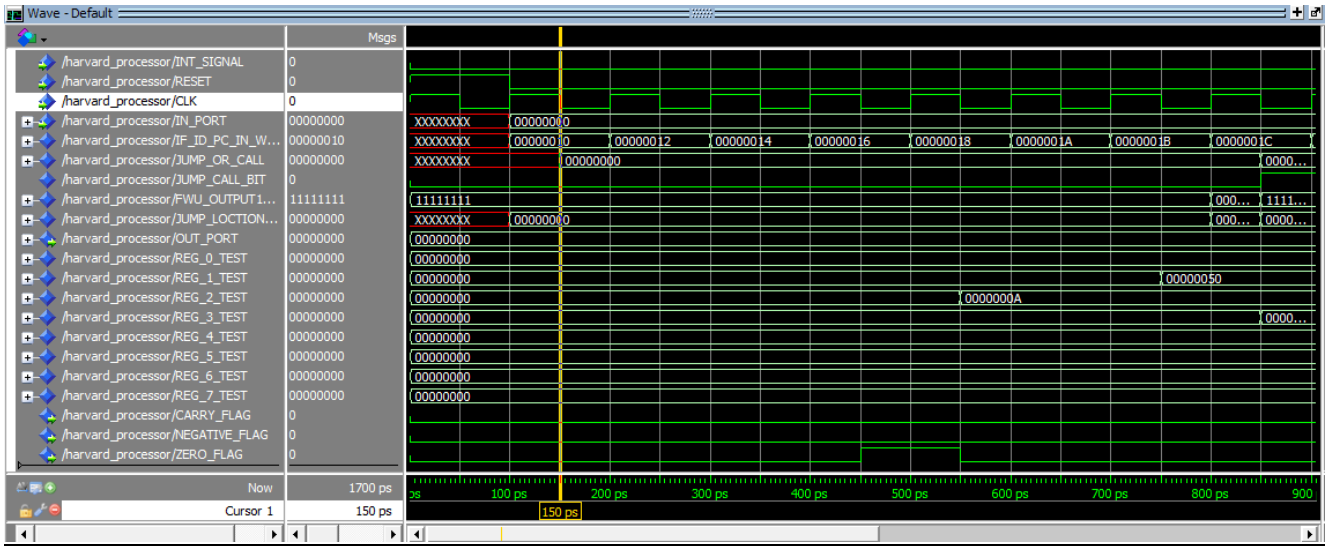


The Cursor above is at 7452 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => INC R6
WriteBack => BUBBLE

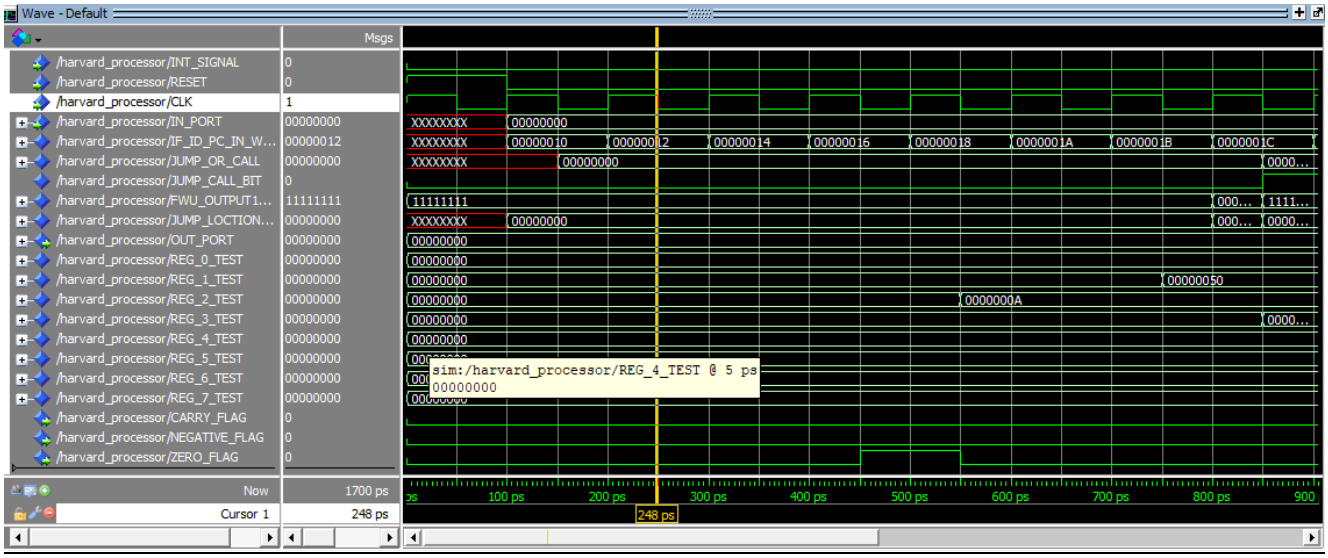


The Cursor above is at 7551 ps which means:
Fetch => NOP
Decode => NOP
Execute => NOP
Memory => NOP
WriteBack => INC R6   R6=00000401

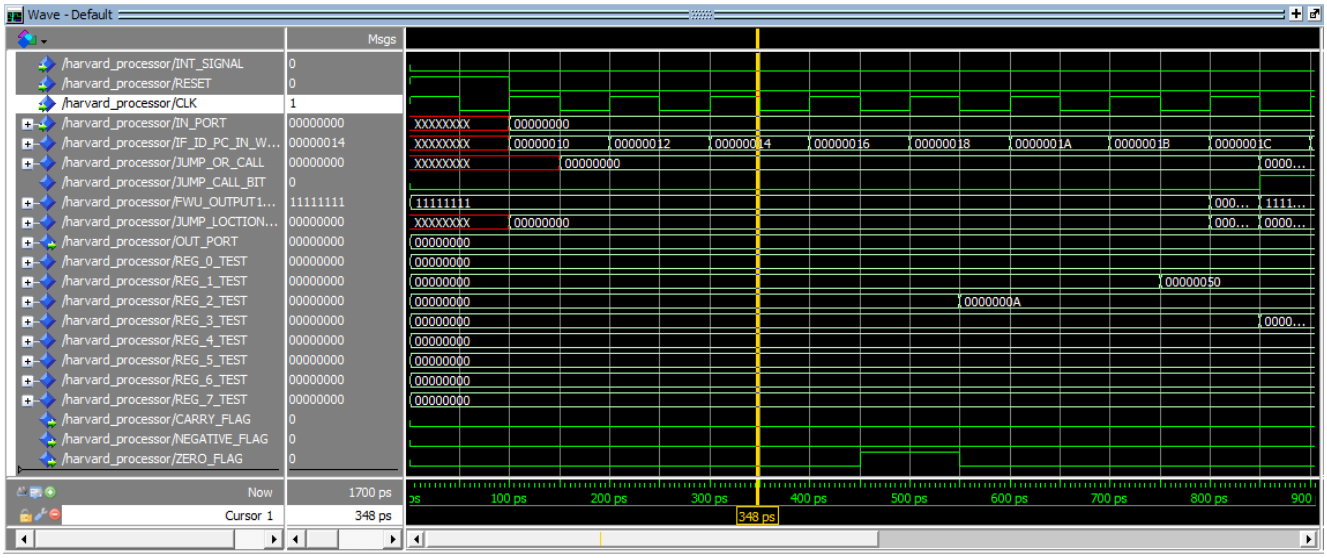
Branch Prediction Test Case



The Cursor above is at 150 ps which means:
Fetch => LDM R2,0A



The Cursor above is at 248 ps which means:
Fetch => LDM R0,0
Decode = >LDM R2,0A

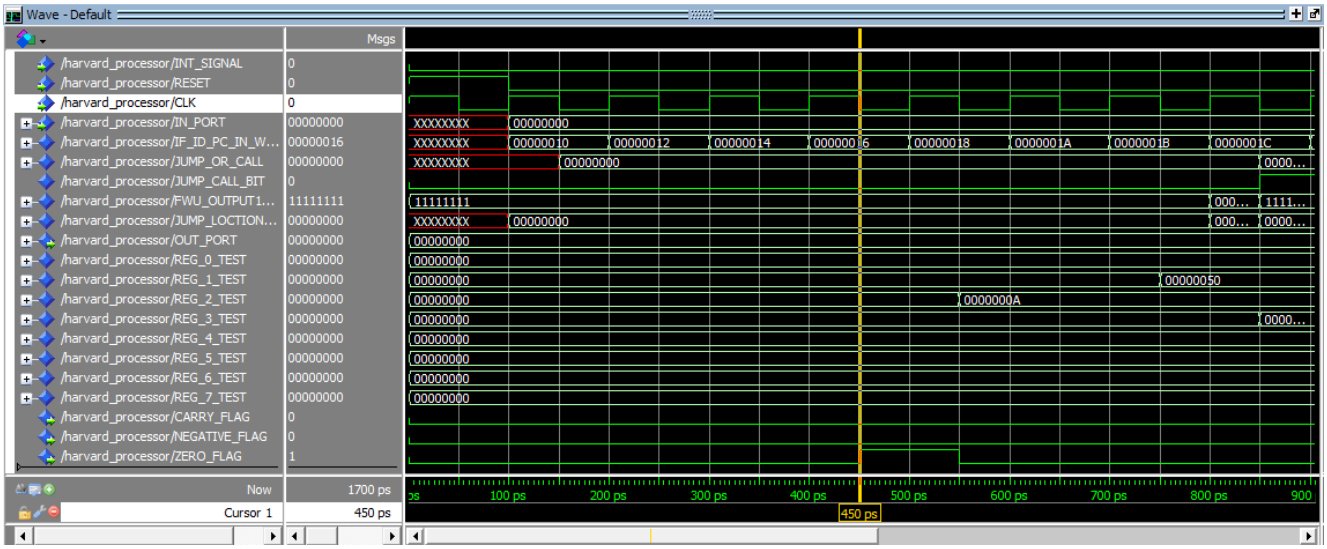


The Cursor above is at 348ps which means:

Fetch => LDM R1,50

Decode = >LDM R0,0

Execute =>LDM R2,0A



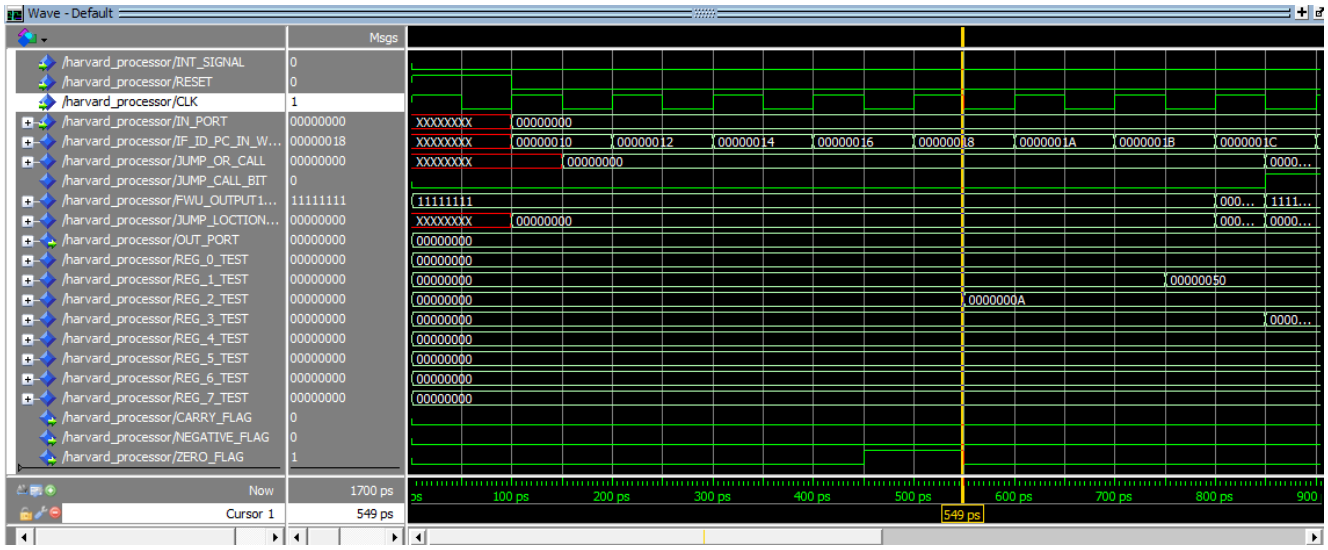
The Cursor above is at 450ps which means:

Fetch => LDM R3,20

Decode = > LDM R1,50

Execute =>LDM R0,0

Memory=>LDM R2,0A



The Cursor above is at 549ps which means:

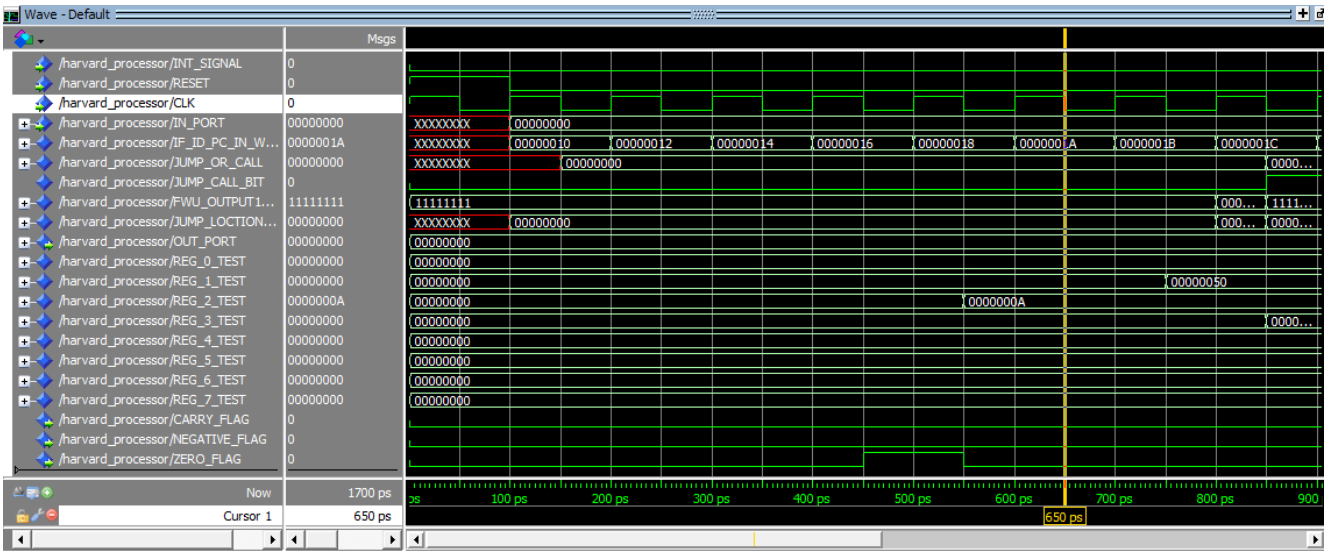
Fetch => LDM R4,2

Decode = >LDM R3,20

Execute =>LDM R1,50

Memory=>LDM R0,0

WriteBack = >LDM R2,0A #R2=0A



The Cursor above is at 650ps which means:

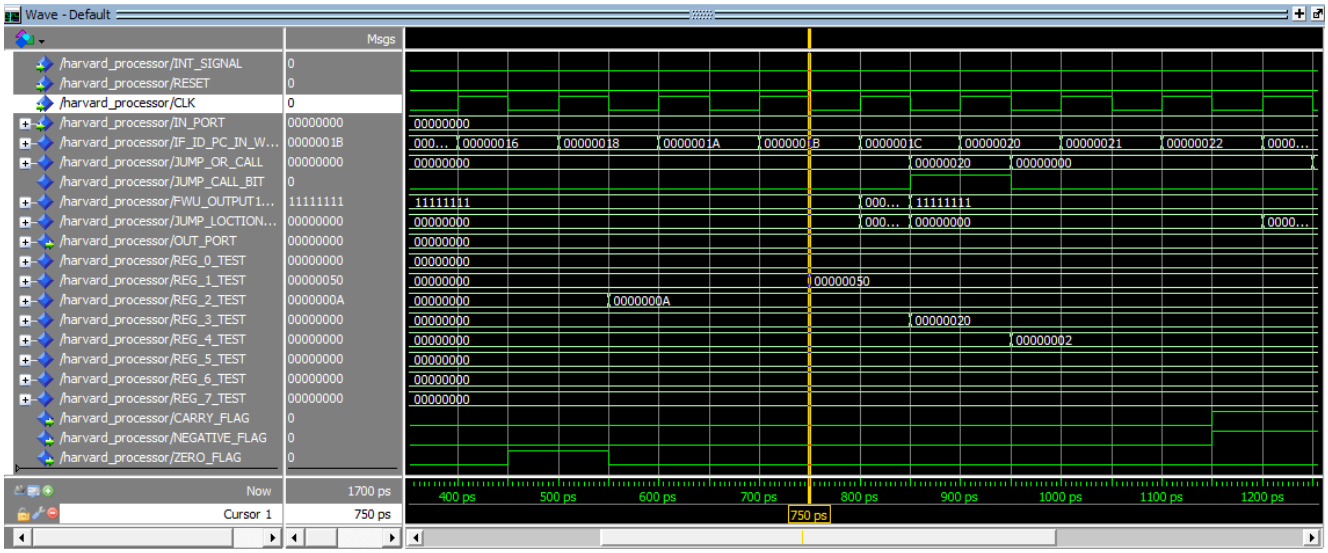
Fetch => JMP R3

Decode = >LDM R4,2

Execute =>LDM R3,20

Memory=>LDM R1,50

WriteBack = >LDM R0,0 #R0=0



The Cursor above is at 750ps which means:

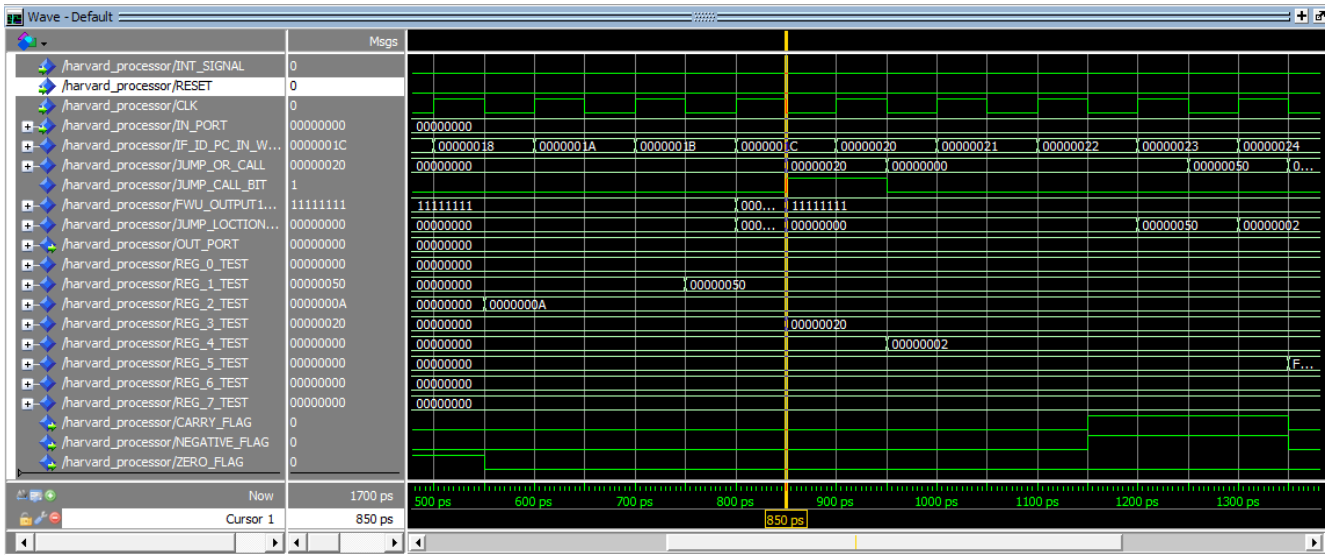
Fetch => NOP

Decode = >JMP R3

Execute =>LDM R4,2

Memory=>LDM R3,20

WriteBack = >LDM R1,50 #R1=50



The Cursor above is at 850ps which means:

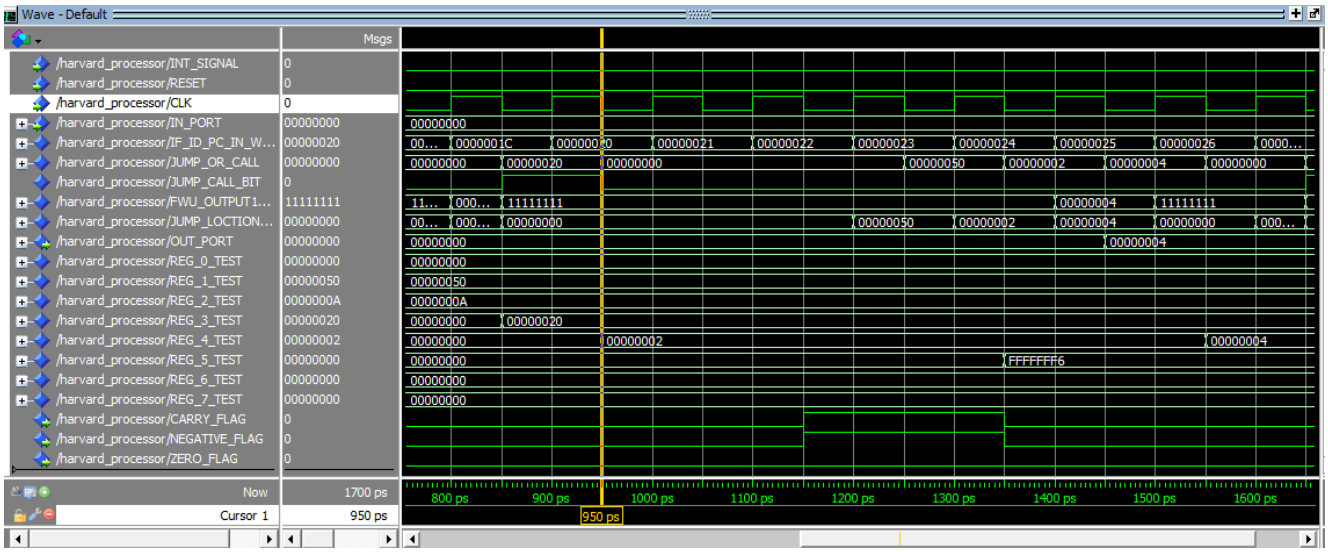
Fetch => NOP

Decode = > NOP

Execute =>JMP R3

Memory=>LDM R4,2

WriteBack = >LDM R3,20 #R3=20



The Cursor above is at 950ps which means:

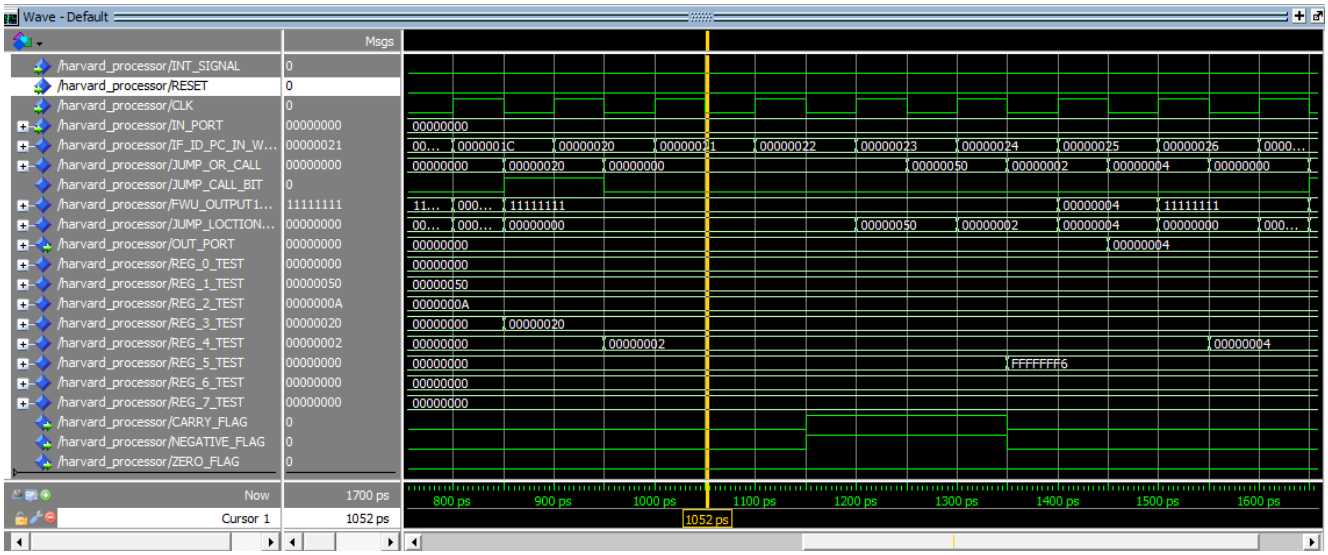
Fetch =>SUB R0,R2,R5

Decode = > NOP

Execute => NOP

Memory=> JMP R3

WriteBack = >LDM R4,2 #R4=2



The Cursor above is at 1052ps which means:

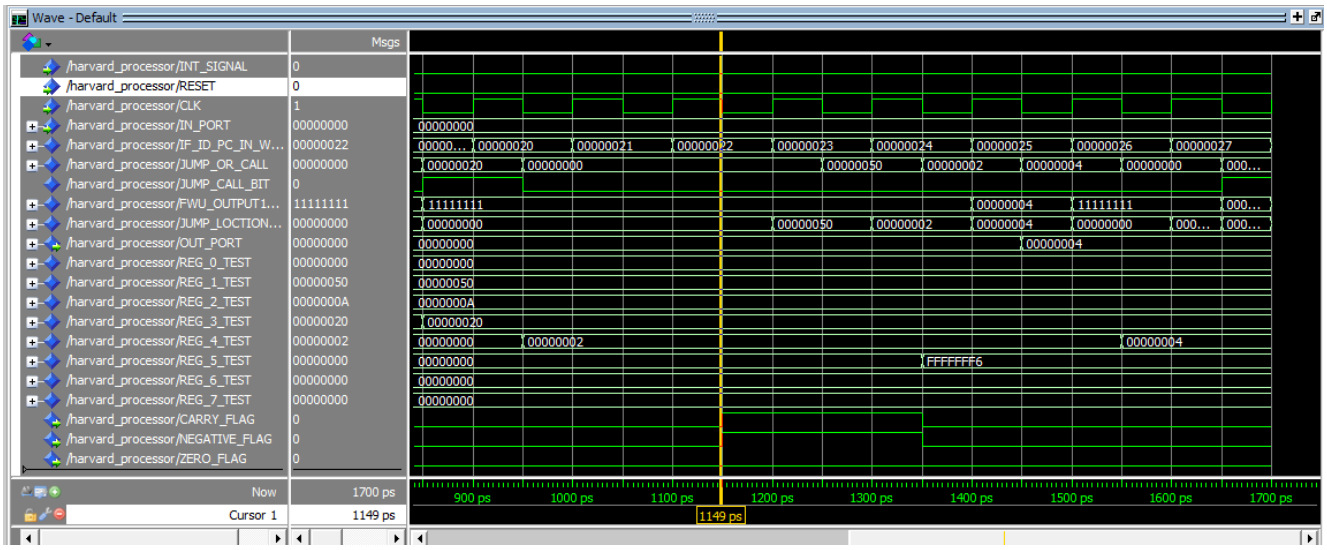
Fetch =>JZ R1

Decode = >SUB R0,R2,R5

Execute => NOP

Memory=> NOP

WriteBack = >JMP R3



The Cursor above is at 1149ps which means:

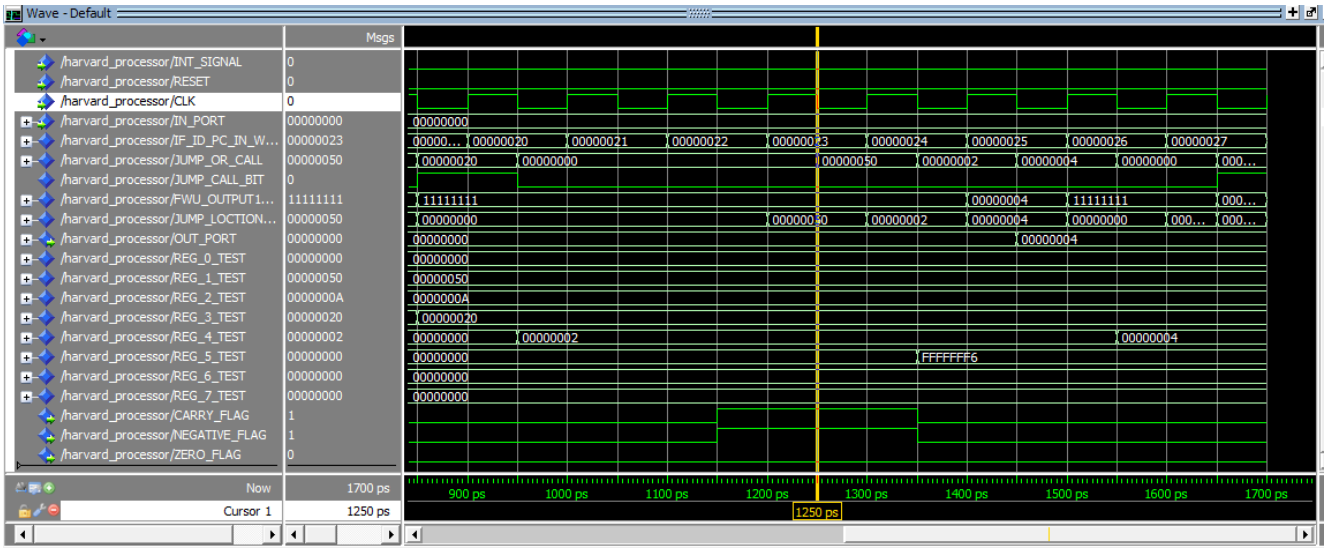
Fetch => ADD R4,R4,R4

Decode = >JZ R1

Execute => SUB R0,R2,R5

Memory=> NOP

WriteBack = > NOP



The Cursor above is at 1250ps which means:

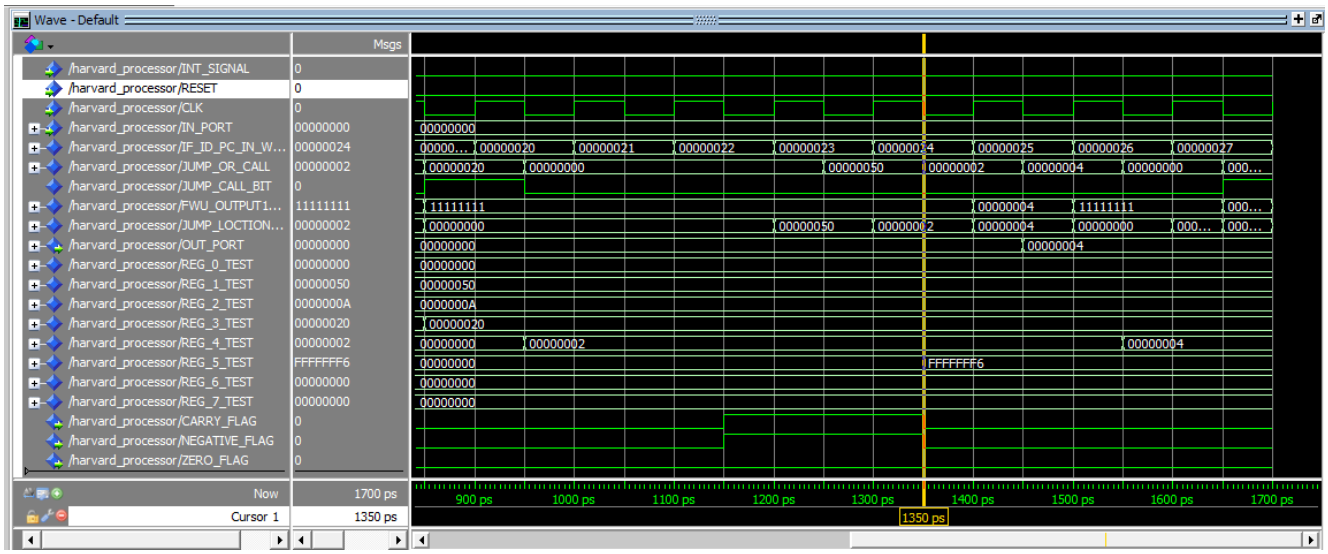
Fetch => OUT R4

Decode = >ADD R4,R4,R4

Execute => JZ R1

Memory=> SUB R0,R2,R5

WriteBack = > NOP



The Cursor above is at 1350ps which means:

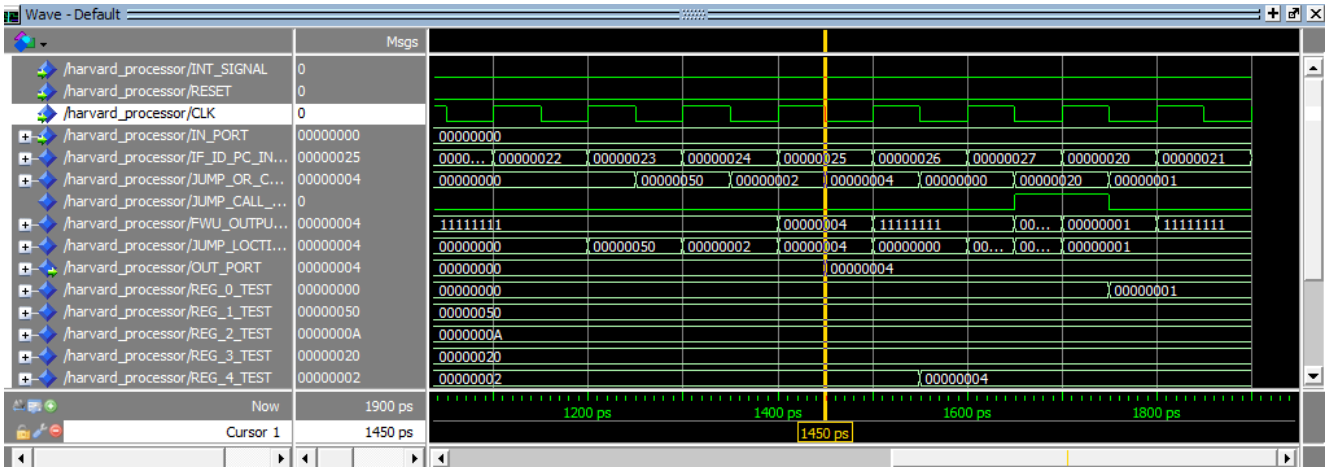
Fetch => INC R0

Decode = >OUT R4

Execute => ADD R4,R4,R4

Memory=> JZ R1

WriteBack = > SUB R0,R2,R5 #R5 = FFFFFFF6



The Cursor above is at 1450ps which means:

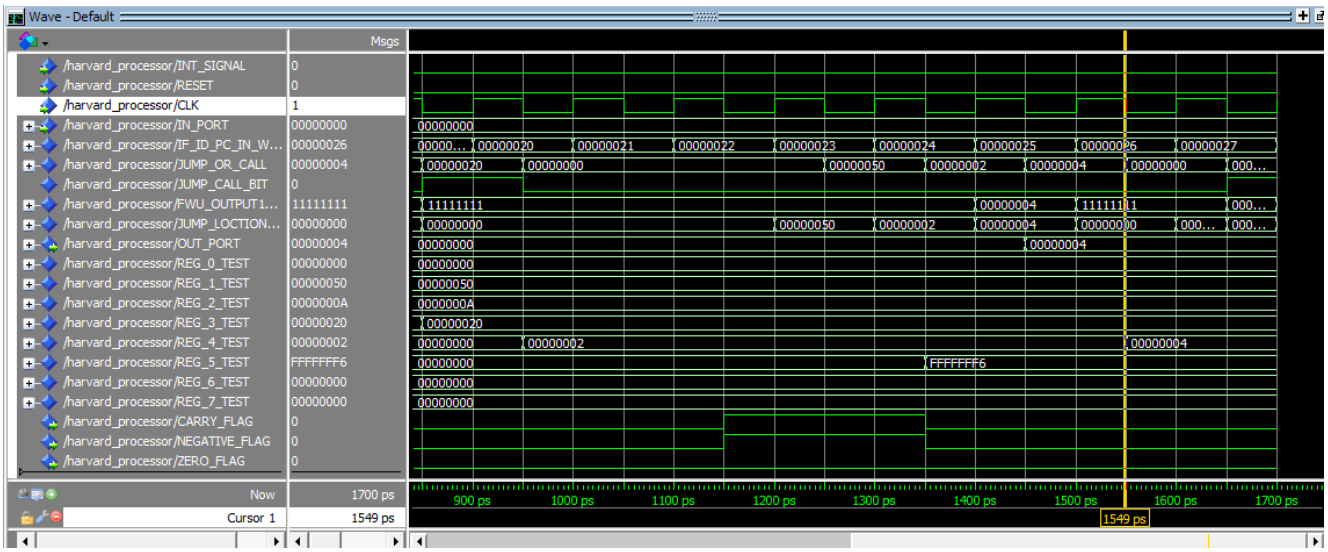
Fetch => JMP R3

Decode = >INC R0

Execute => OUT R4 #OUTPORT = 00000004

Memory=>ADD R4,R4,R4

WriteBack = > JZ R1



The Cursor above is at 1549ps which means:

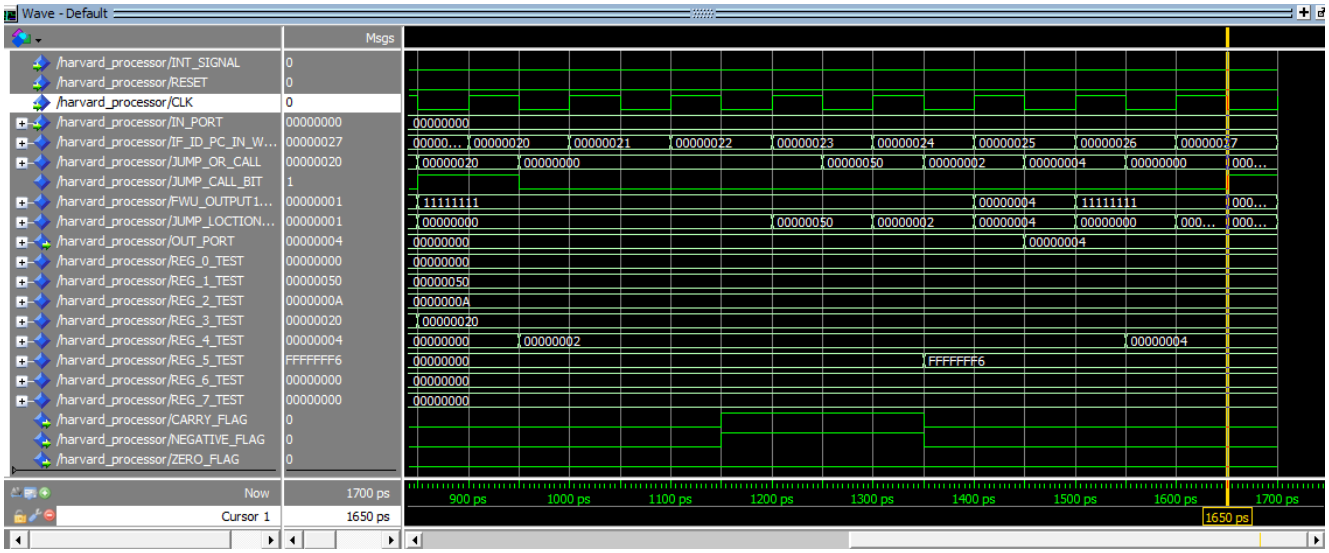
Fetch => NOP

Decode = >JMP R3

Execute => INC R0

Memory=>OUT R4

WriteBack = > ADD R4,R4,R4 #R4 = 00000004



The Cursor above is at 1650ps which means:

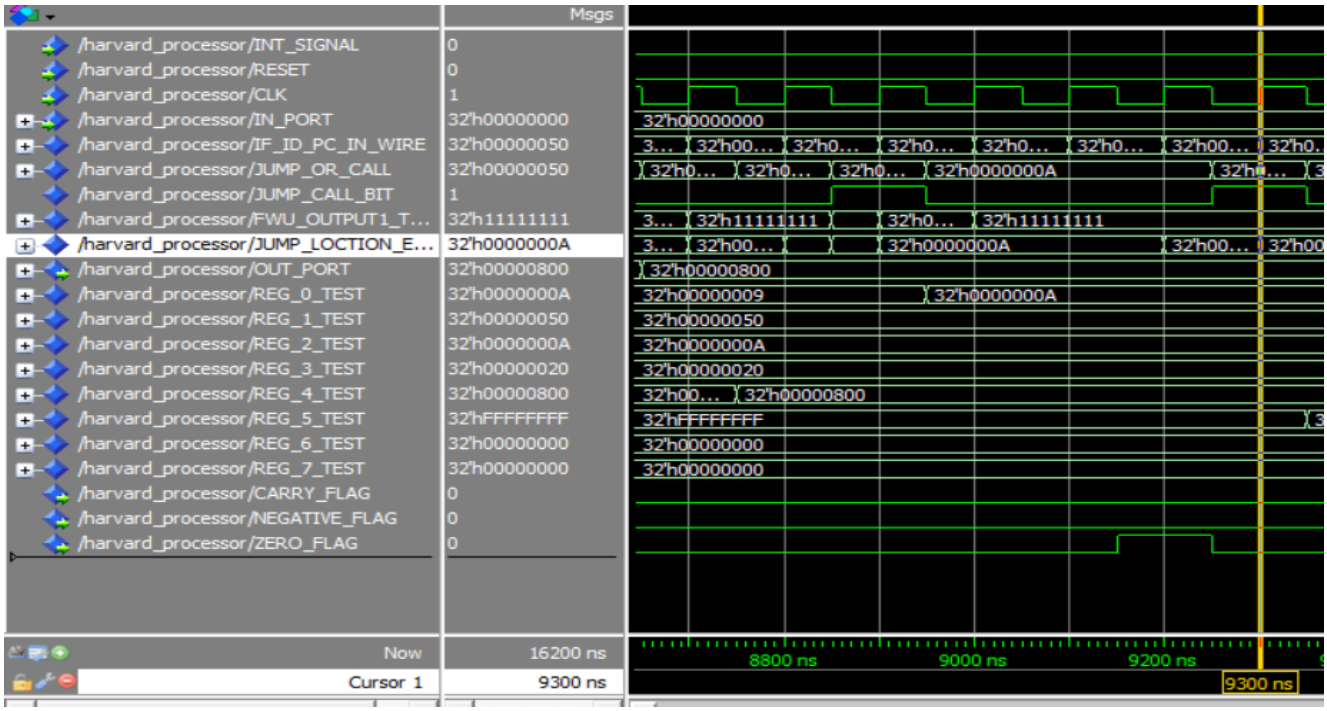
Fetch => NOP

Decode = >NOP

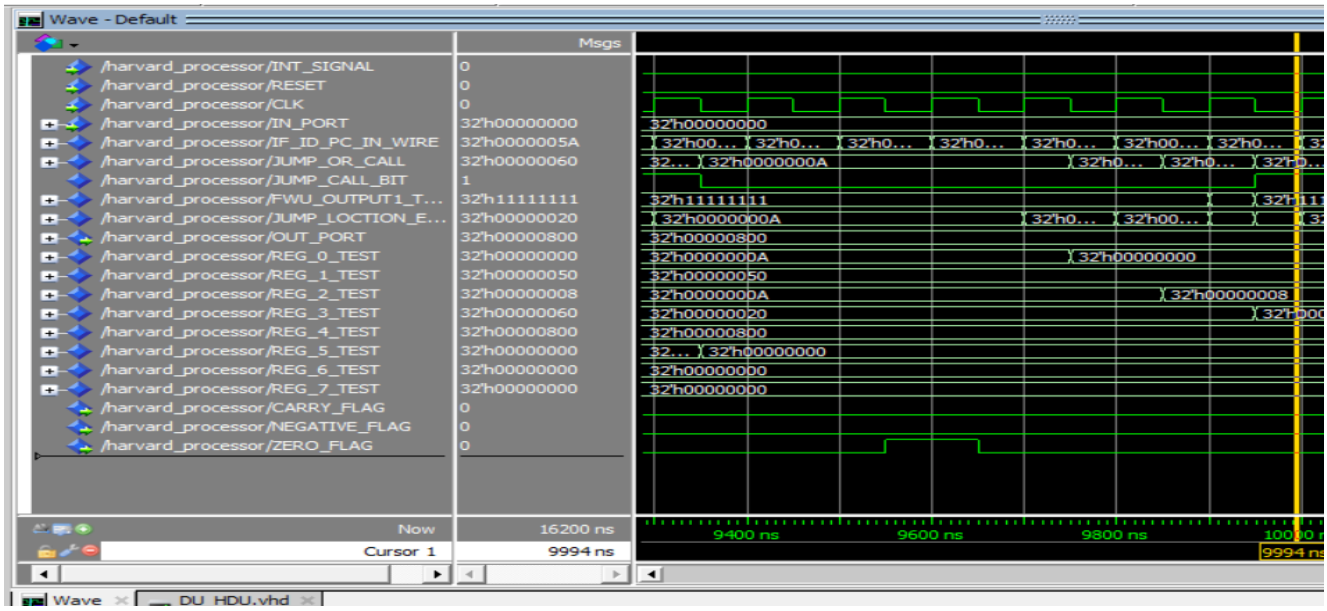
Execute => JMP R3

Memory=>INC R0

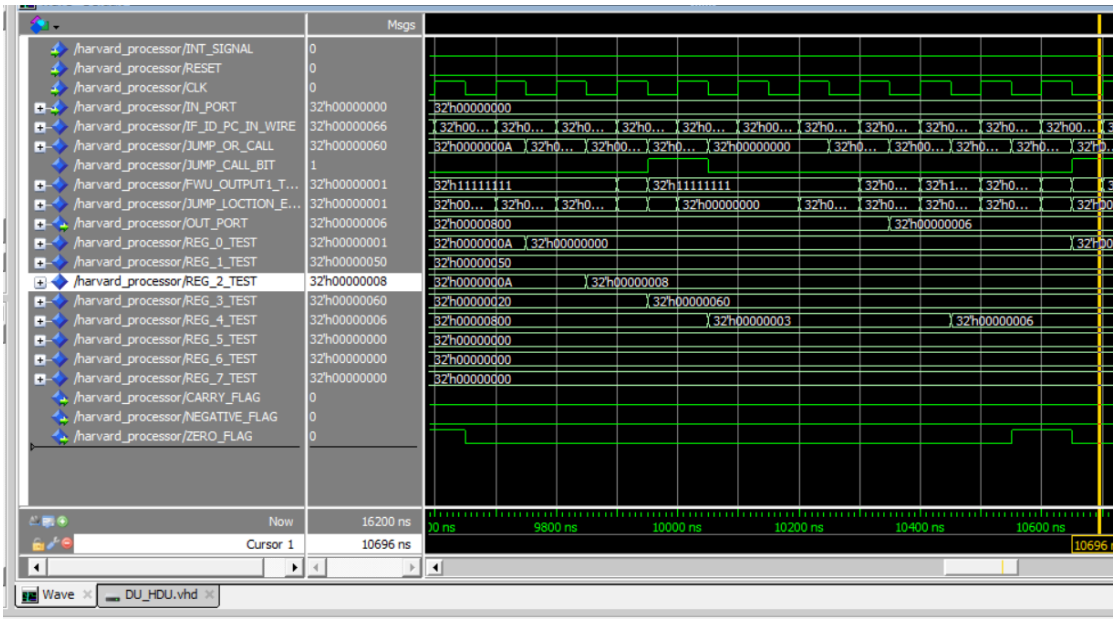
WriteBack = > OUT R4



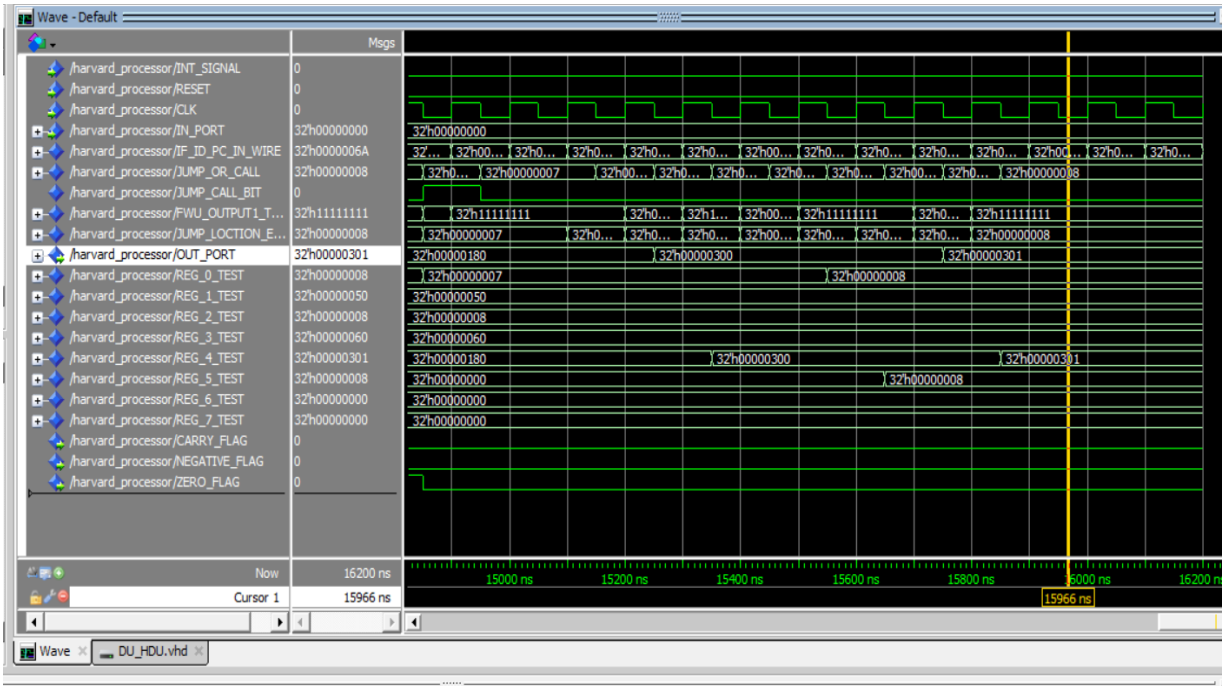
The loop in 900ps till 1700ps will keep happening 11 times until R1 becomes and then the JZ occurs this will happen at 9300 which is the execute stage of JZ of the 11th loop



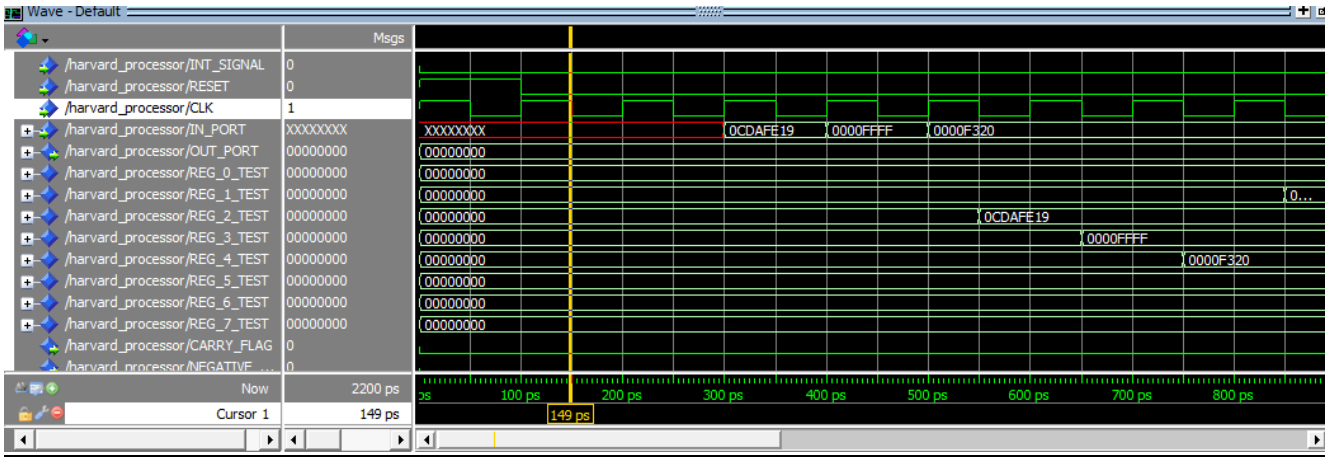
from 50 till 5A the LDM instructions will occur normally then a jmp instruction will go to place 60 and then start the following loop



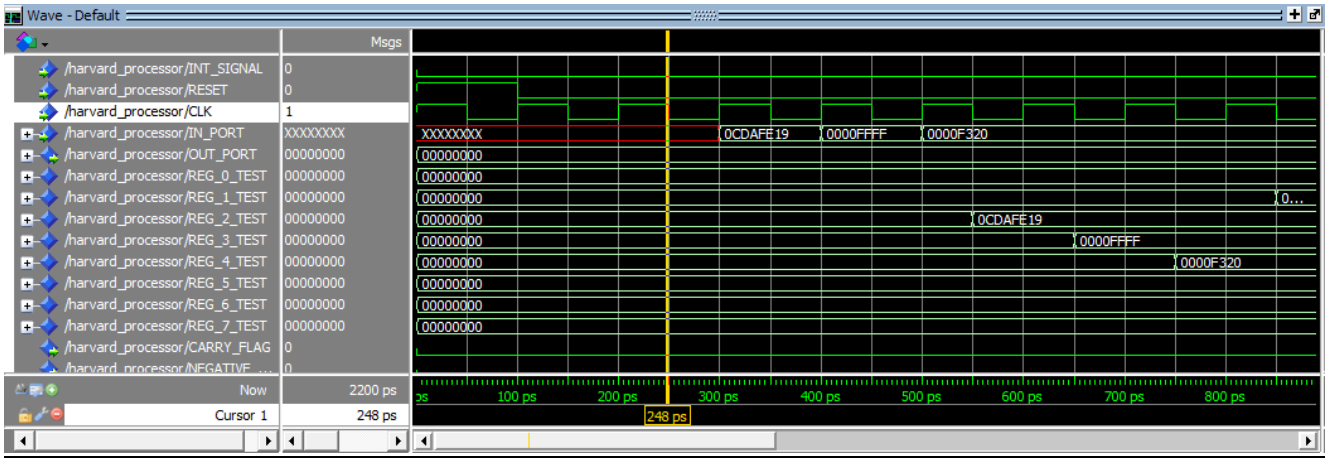
This loop goes on from 10000 to 10700 and then restarts for 8 times and on the 8th time the condition is achieved and the R3 is not a zero anymore then the loop is exited



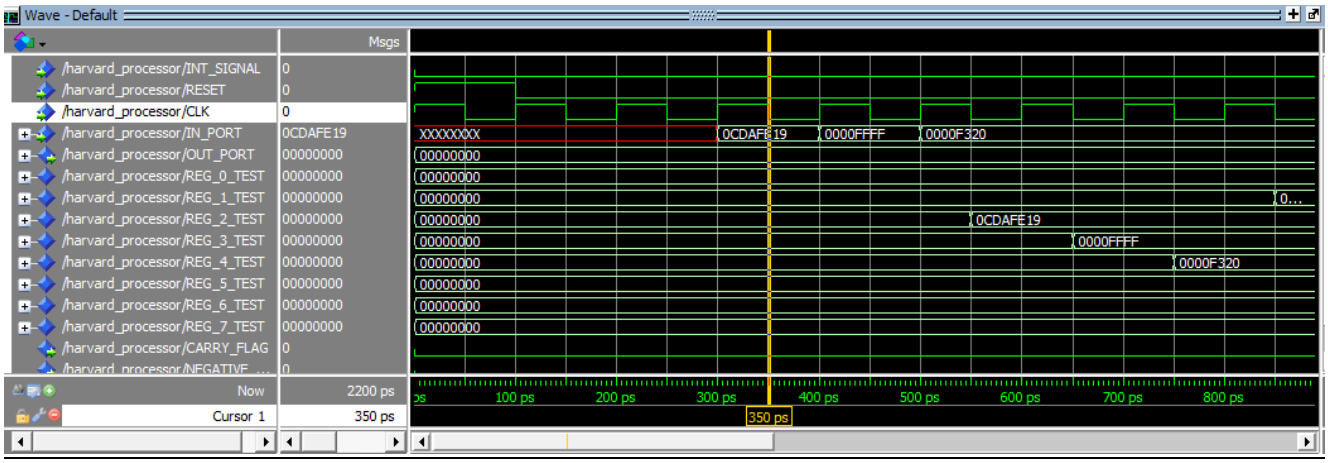
Memory Test Case



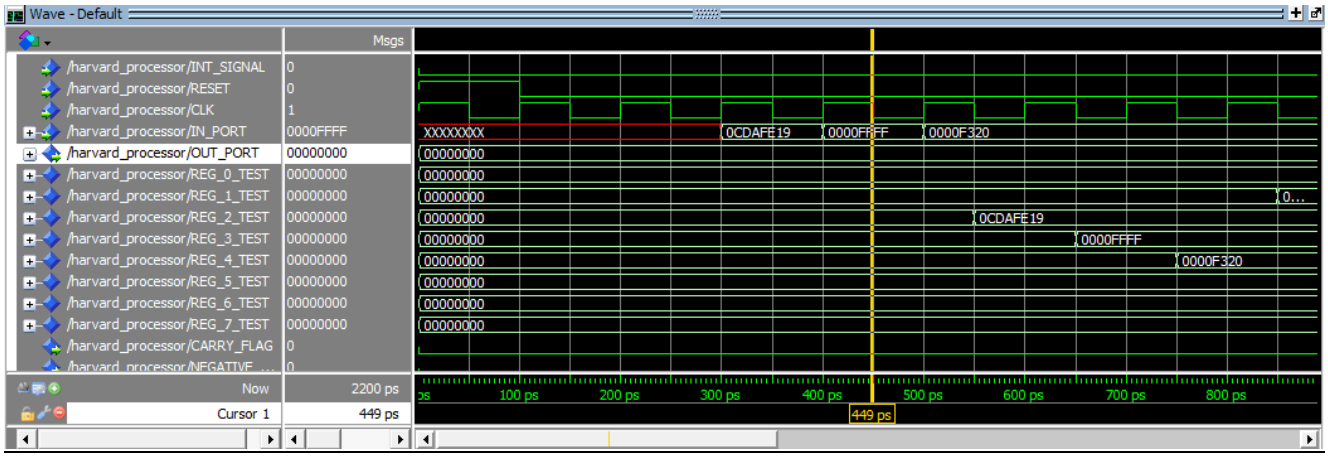
Fetch =>in R2



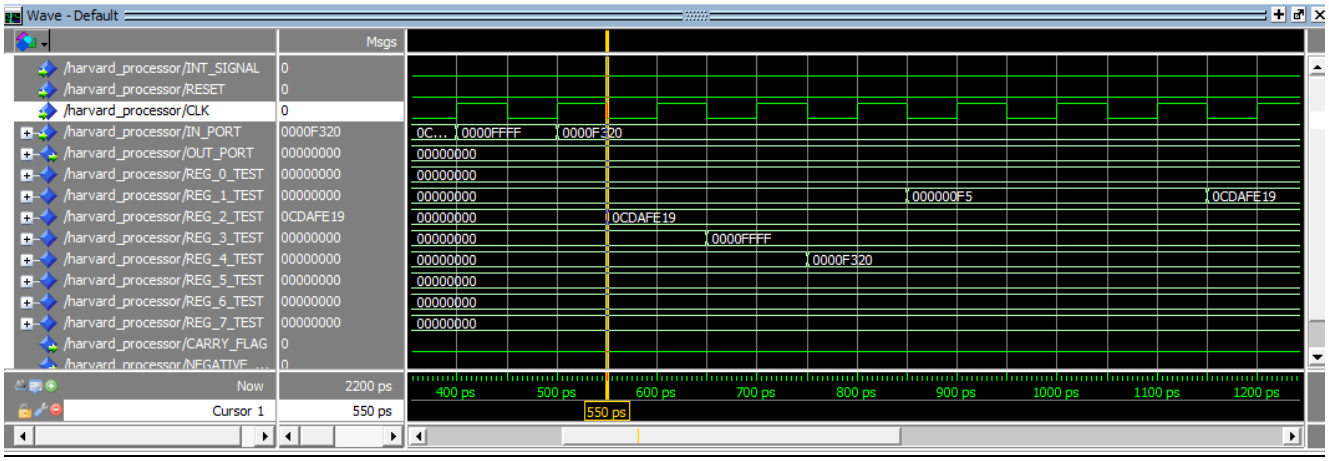
Fetch =>in R3
Decode => in R2



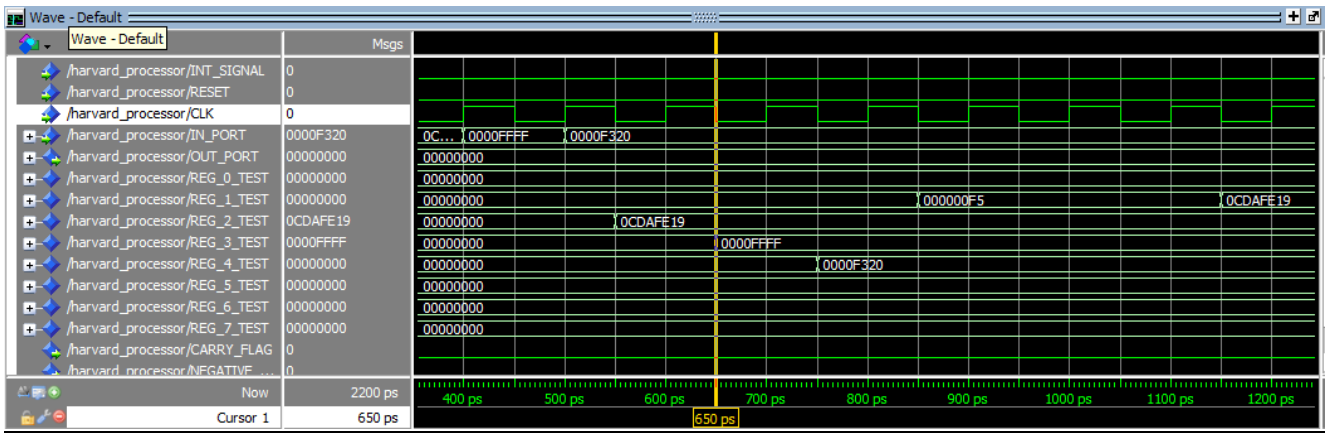
Fetch =>in R4
Decode => in R3
Execute => in R2



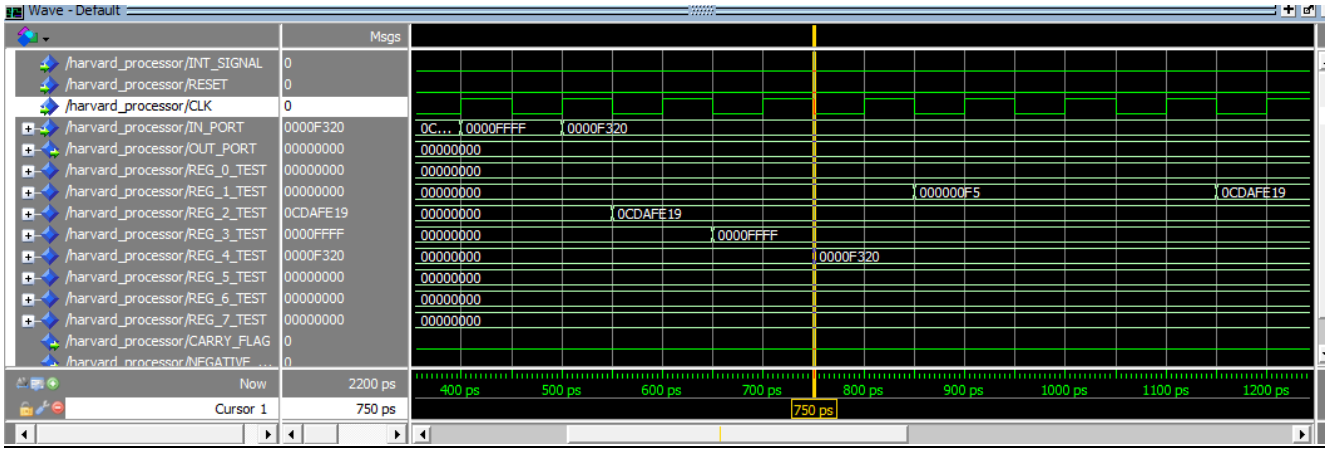
Fetch =>LDM R1,F5
Decode => in R4
Execute => in R3
Memory =>in R2



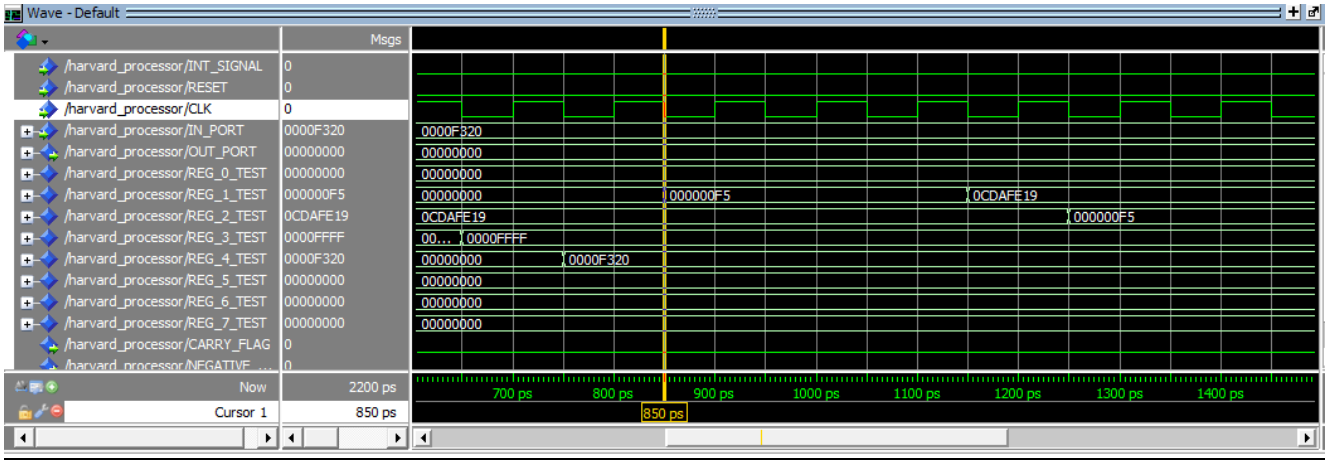
Fetch =>PUSH R1
Decode => LDM R1,F5
Execute => in R4
Memory =>in R3
WriteBack =>in R2 #R2 = 0CDAFE19



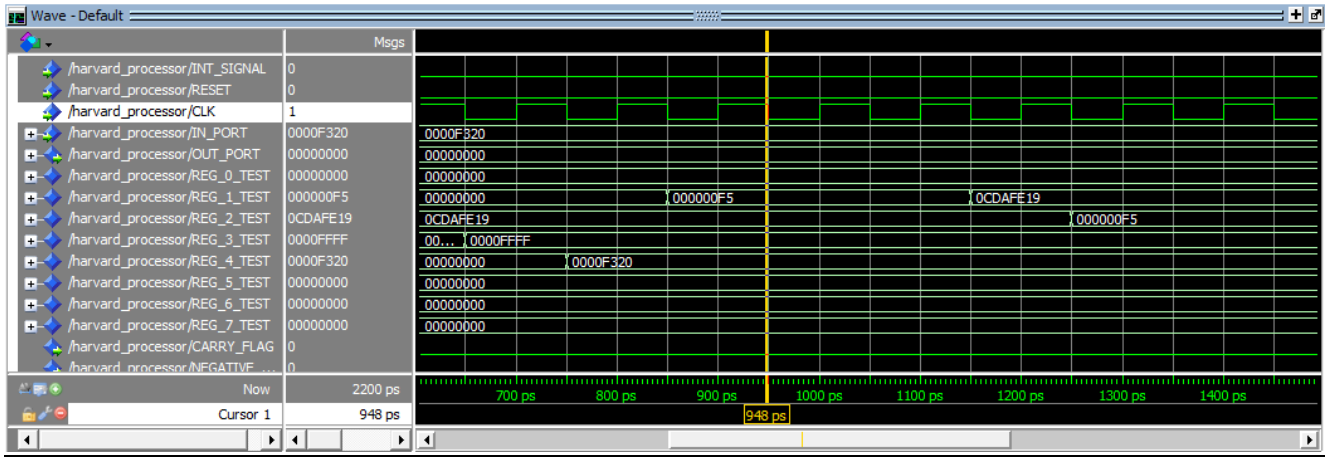
Fetch =>PUSH R2
Decode => PUSH R1
Execute => LDM R1,F5
Memory => in R4
WriteBack =>in R3 #R3=FFFF



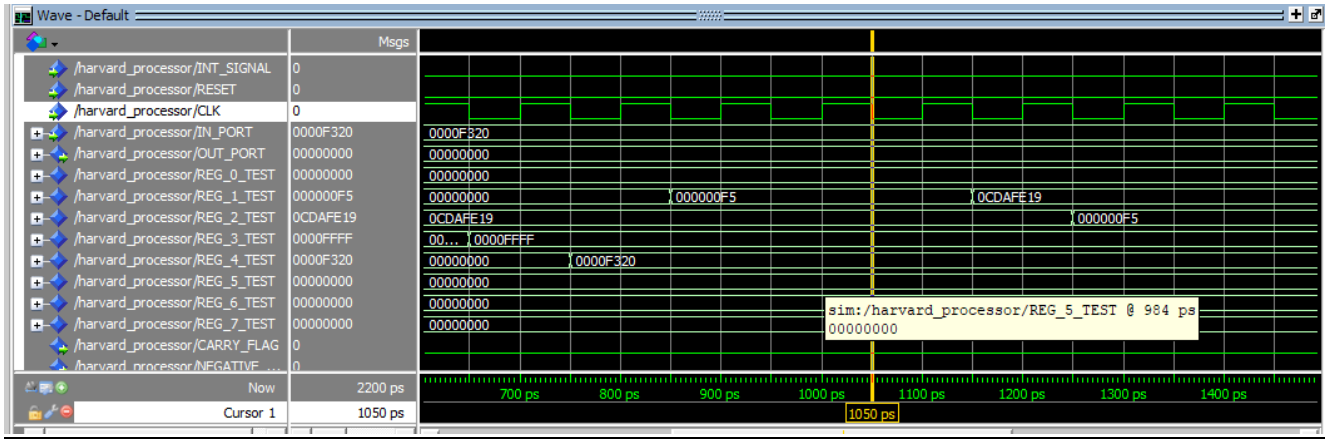
Fetch =>POP R1
Decode => PUSH R2
Execute => PUSH R1
Memory =>LDM R1,F5
WriteBack => in R4 #R4=F320



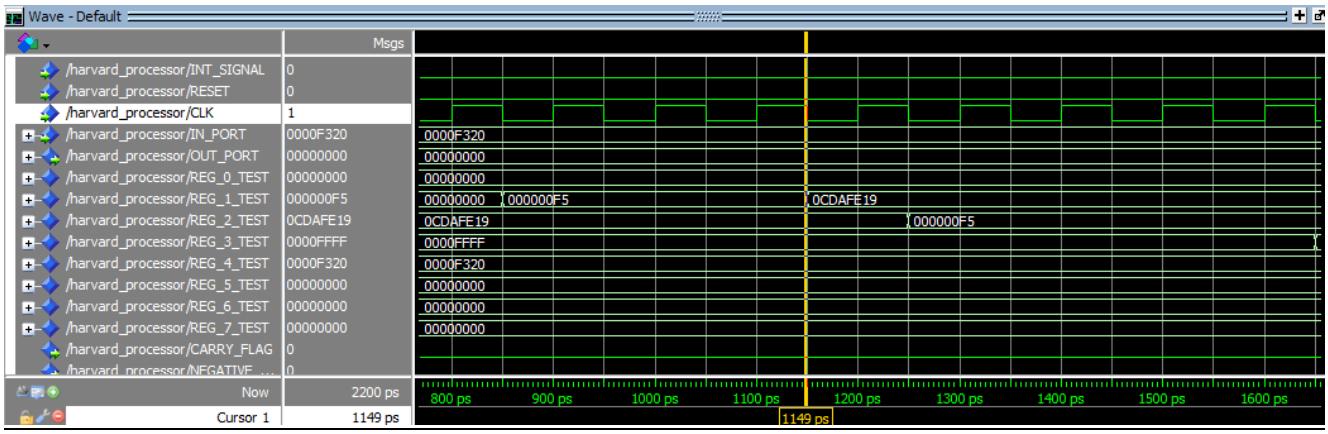
Fetch =>POP R2
Decode => POP R1
Execute =>PUSH R2
Memory => PUSH R1 #M[7FE, 7FF] = F5
WriteBack => LDM R1,F5 #R1=F5



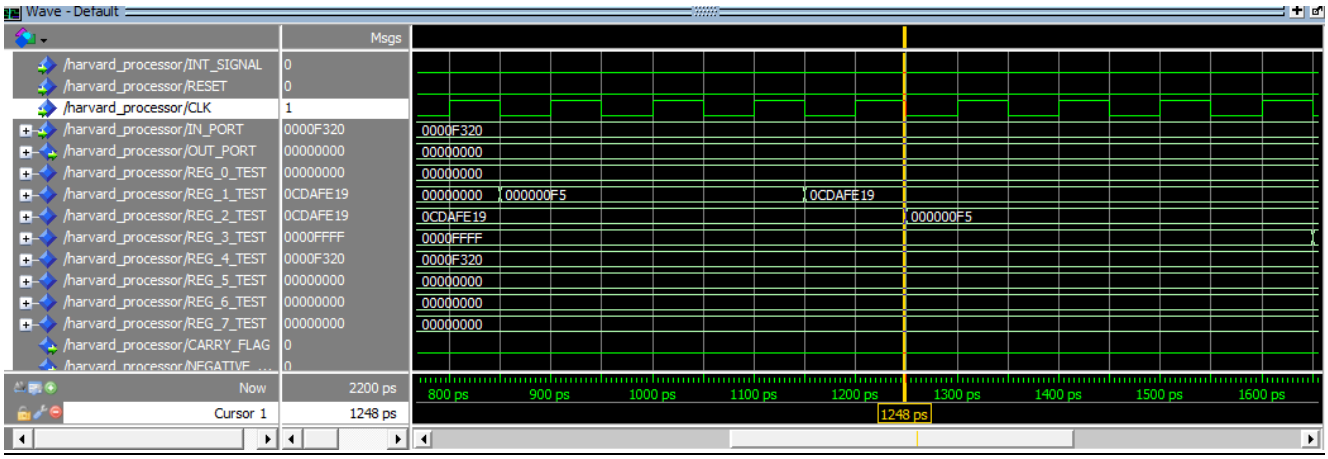
Fetch =>STD R2,200
Decode =>POP R2
Execute => POP R1
Memory => PUSH R2 #M[7FC, 7FD]=0CDAFE19
WriteBack => PUSH R1



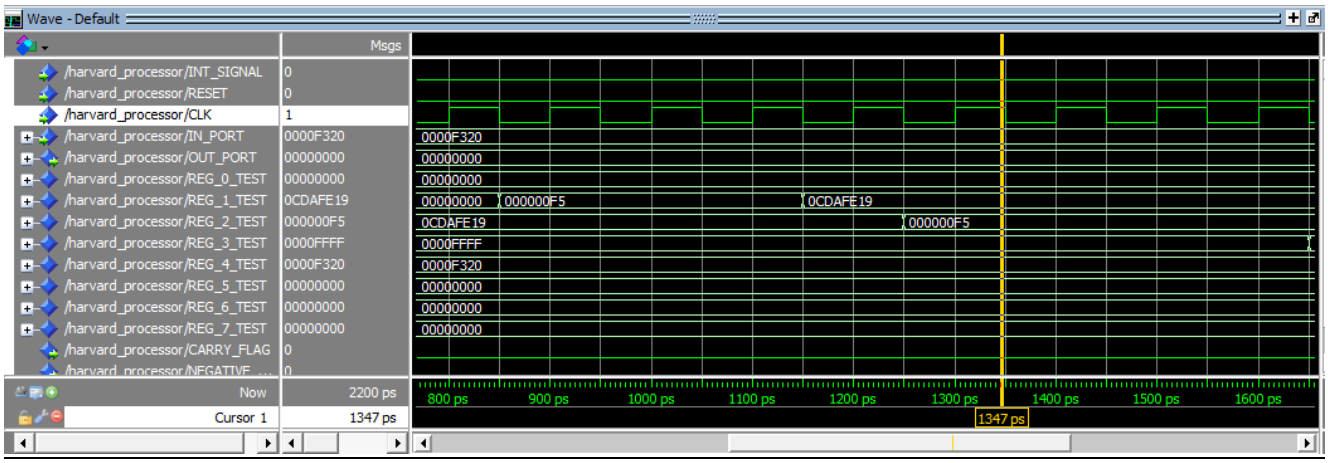
Fetch =>STD R1,202
Decode =>STD R2,200
Execute =>POP R2
Memory => POP R1
WriteBack => PUSH R2



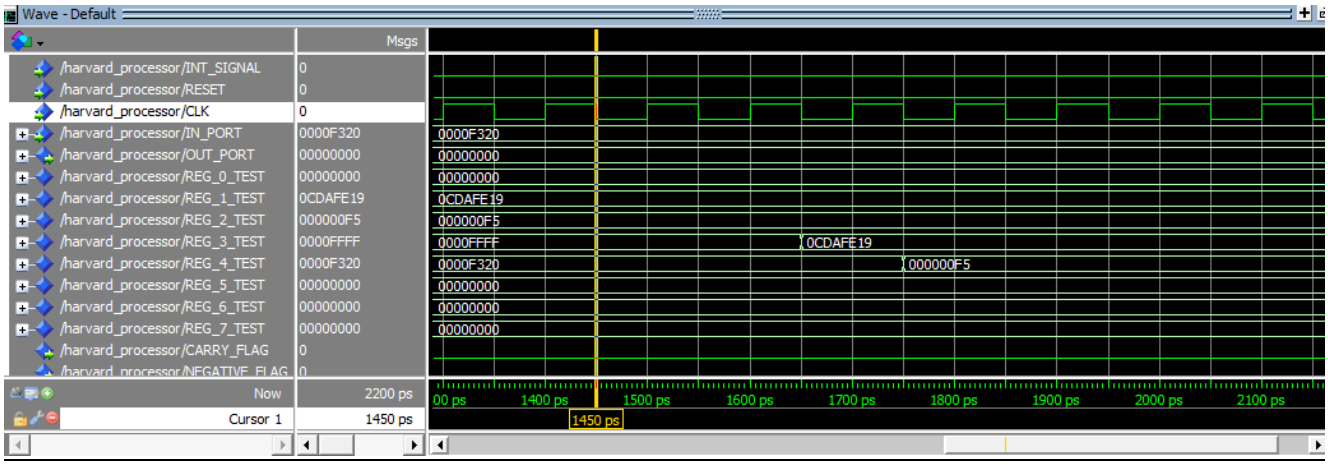
Fetch =>STD R1,202
Decode =>STD R2,200
Execute => BUBBLE # HAZARD LOAD CASE
Memory =>POP R2
WriteBack => POP R1 R1=0CDAFE19 POP M[7FC,7FD] TO R1



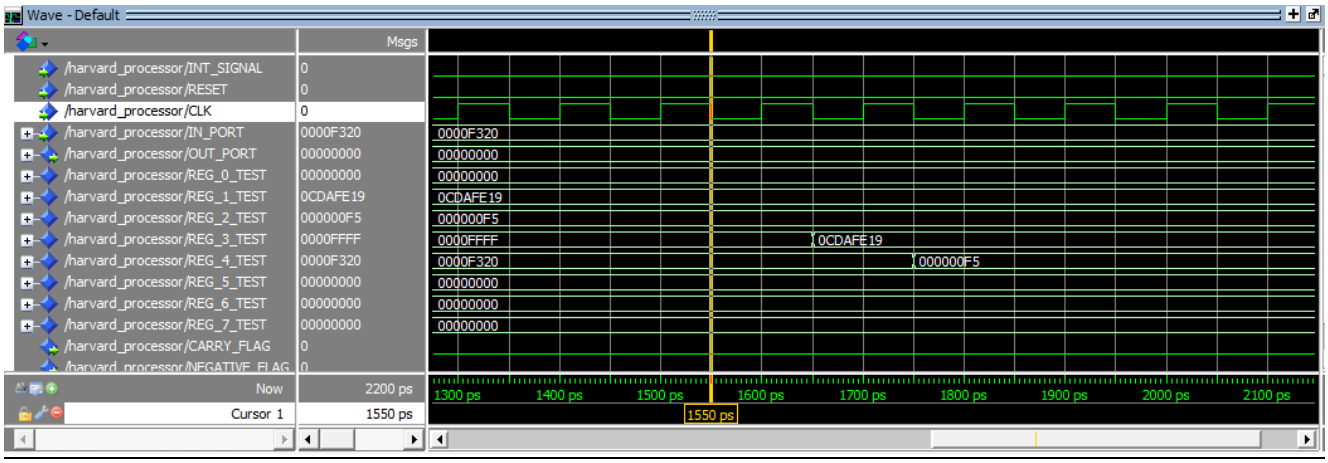
Fetch =>LDD R3,202
Decode =>STD R1,202
Execute=>STD R2,200
Memory => BUBBLE # HAZARD LOAD CASE
WriteBack => POP R2 #R2=F5 POPING M[7FC,7FD] TO R2



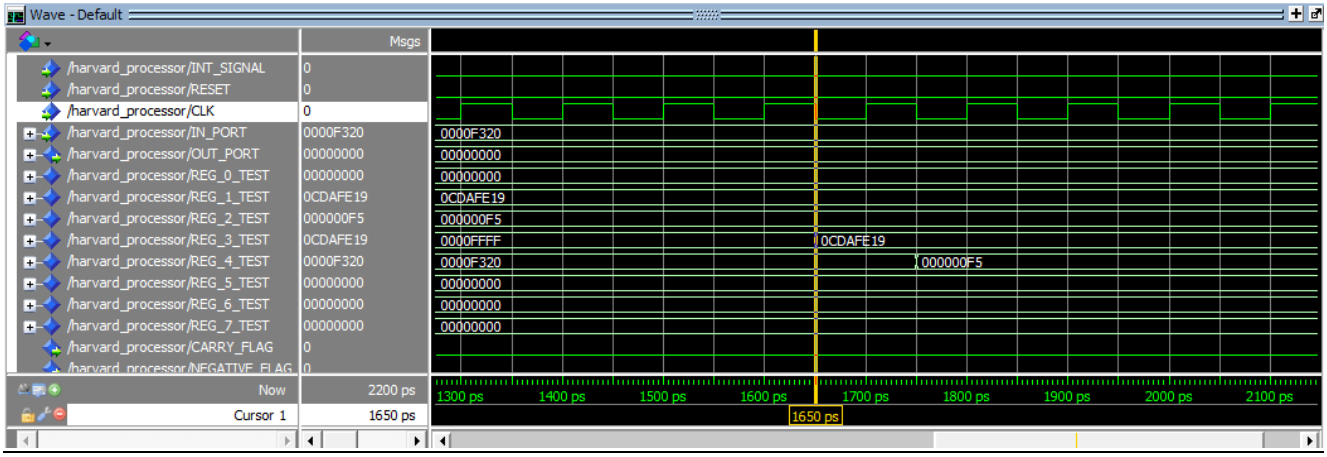
Fetch =>LDD R4,200
Decode =>LDD R3,200
Execute =>STD R1,202
Memory =>STD R2,200 #M[200, 201]=F5
WriteBack => BUBBLE



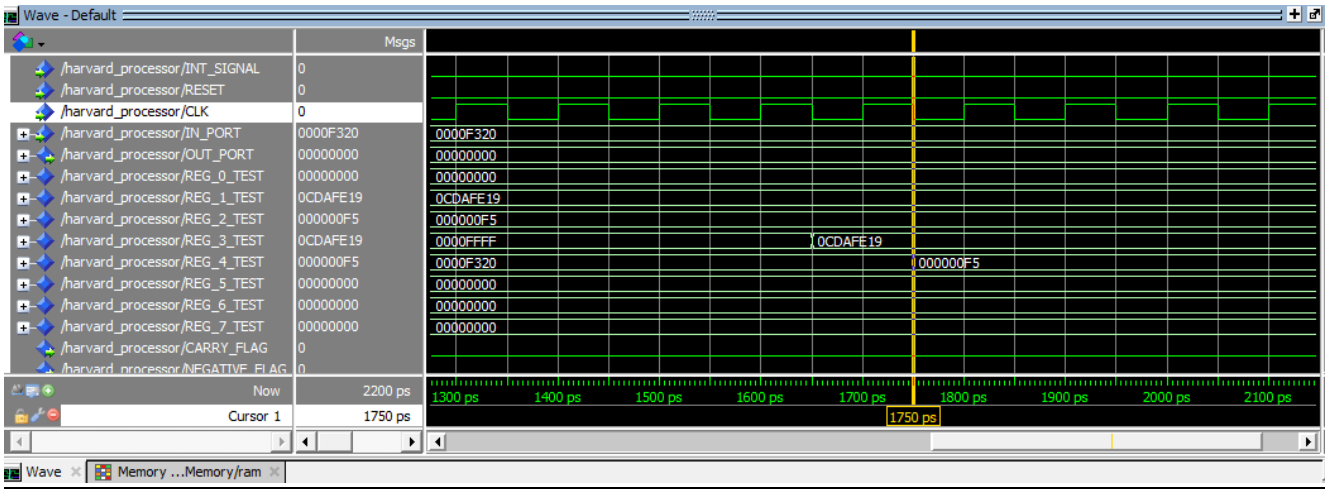
Decode =>LDD R4,200
Execute =>LDD R3,202
Memory => STD R1,202 M[202, 203]=0CDAFE19
Write Back => STD R2,200



Execute =>LDD R4,200
Memory => LDD R3,202
WriteBack =>STD R1,202



Memory =>LDD R4,200
WriteBack => LDD R3,202 #R3=0CDAFE19



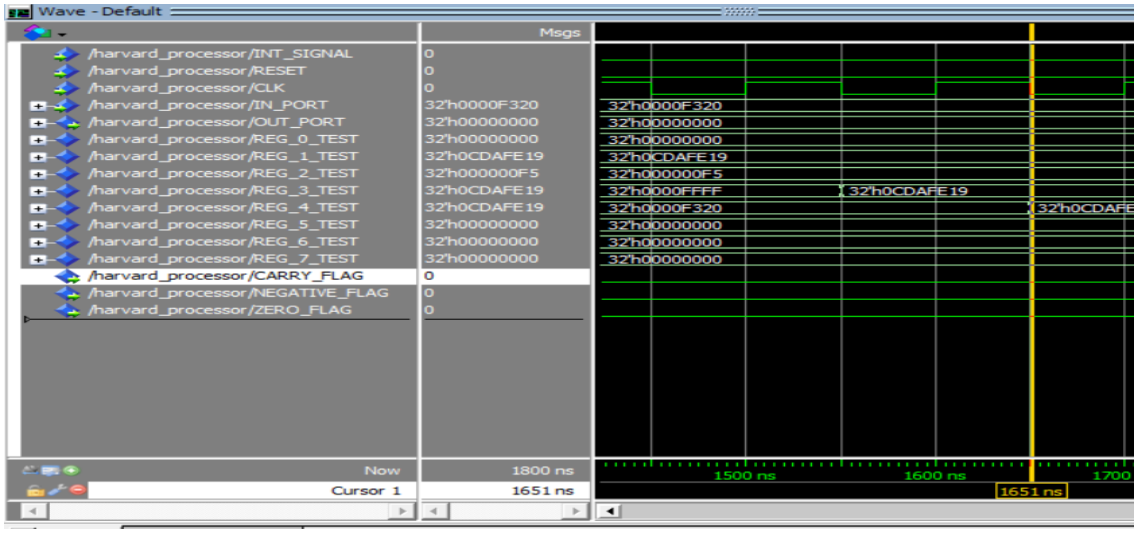
WriteBack => LDD R4,200 #R4=F5

Case II: HDU and Flushing Off

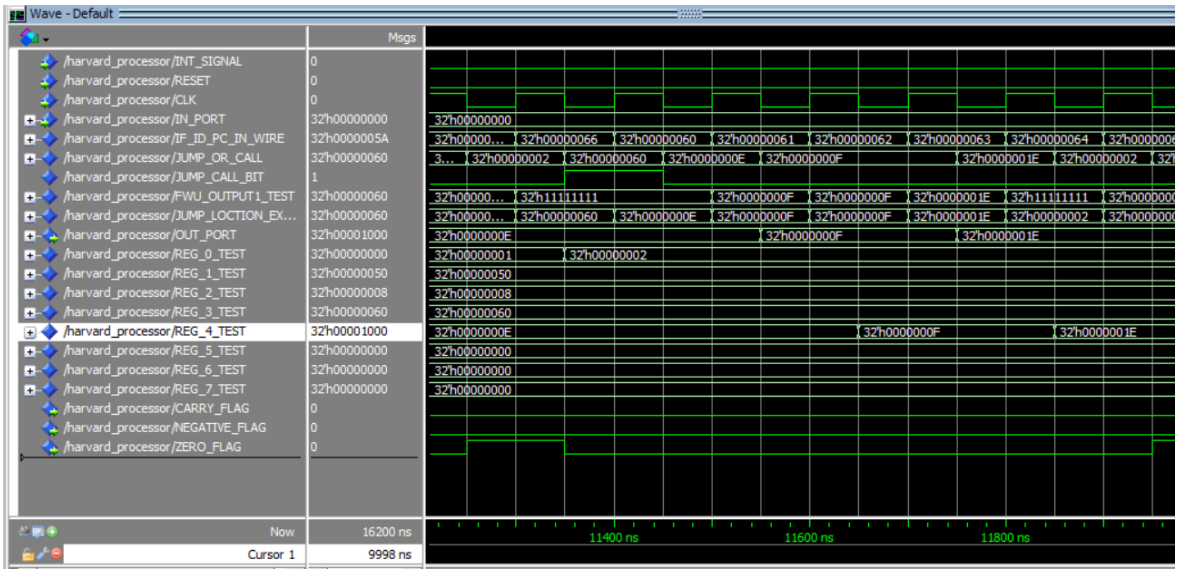
One Operand did not suffer any change and that is expected because this test case does not contain any load cause that may need the HDU or need for flush

Two Operand did not suffer any change and that is expected because this test case does not contain any load cause that may need the HDUor need for flush

Memory did Suffer a change in two instructions because of the load case use (pop R2) that was not handled which took the old value of R2 0CDAFE19 and stored it in M[200,201] instead of saving the previously loaded F5 and later on when this value was popped to R3 it was already the wrong value.




```
000001b8 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
000001cc 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
000001e0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
000001f4 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 FE19 0CDA FE19 0CDA 0000 0
00000208 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0
```

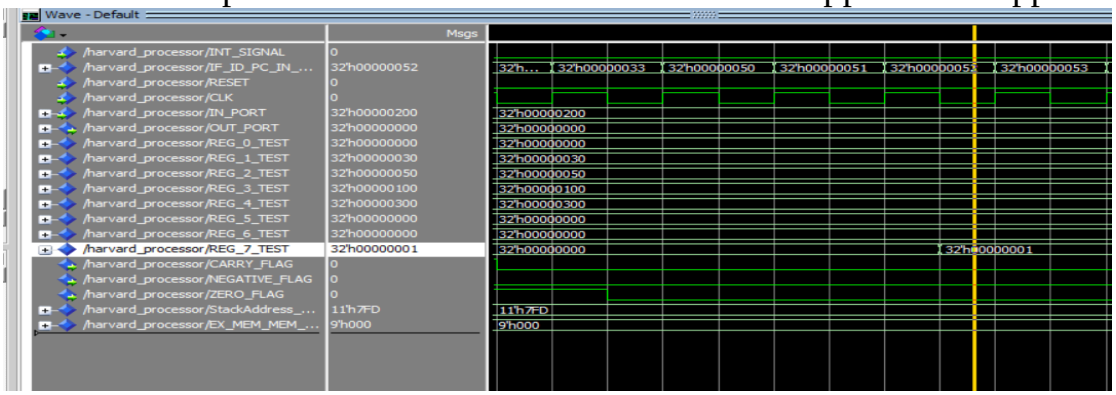


Branch prediction did Suffer a change in 2 places but the first time it was not noticed, the instructions that entered the pipe were nop since I initialize the unfilled instructions with nop instructions and there is a 10 locations gap after the jmp of the first loop and in the last iteration the R4 is already zero so when the commands that are in the pipe not flushed they do not show a change on the value as they also remake it a 0 but in the second loop the loop suffers a lot because of the commands that enter the pipe directly affect the R4 register which increments the R4 and out before starting the next operation as shown above the out is done twice and there is an increment to its value which is not supposed to happen.

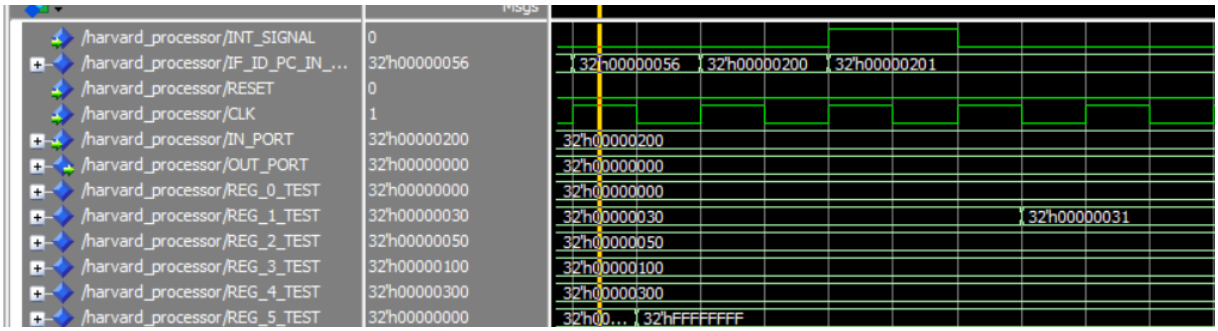
Branch did Suffer a change 4 times because of wrong instructions not cleared from the pipe

another problem that was going to happen was due to load case is that at location 200 the pop r6 is going to pop the wrong value to the r6 but since that value is 200 the call recalled again with the correct value

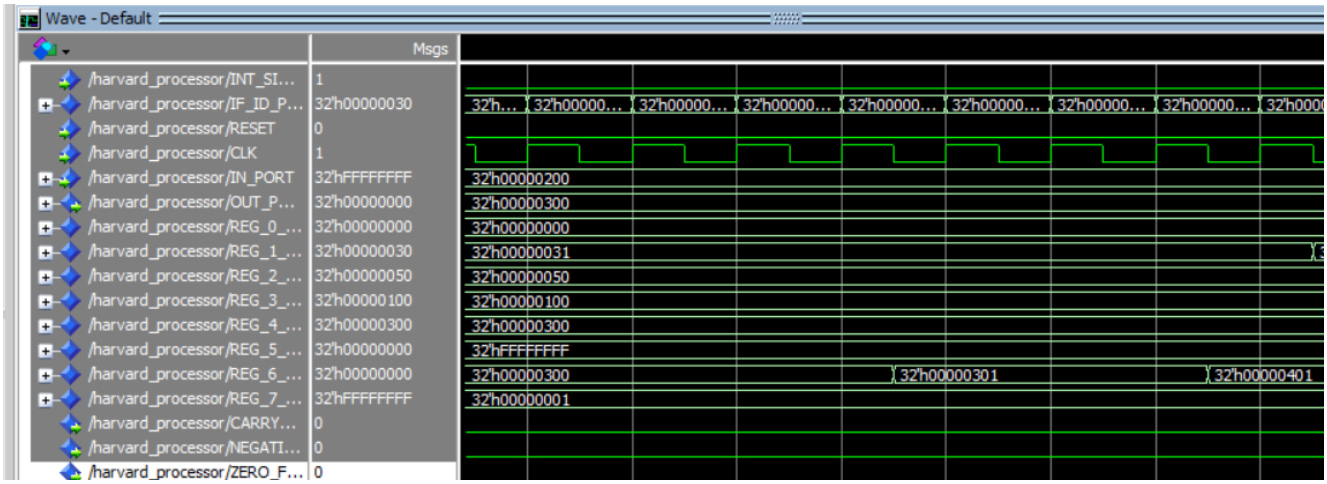
in the below picture the instruction inc R7 was not supposed to happen



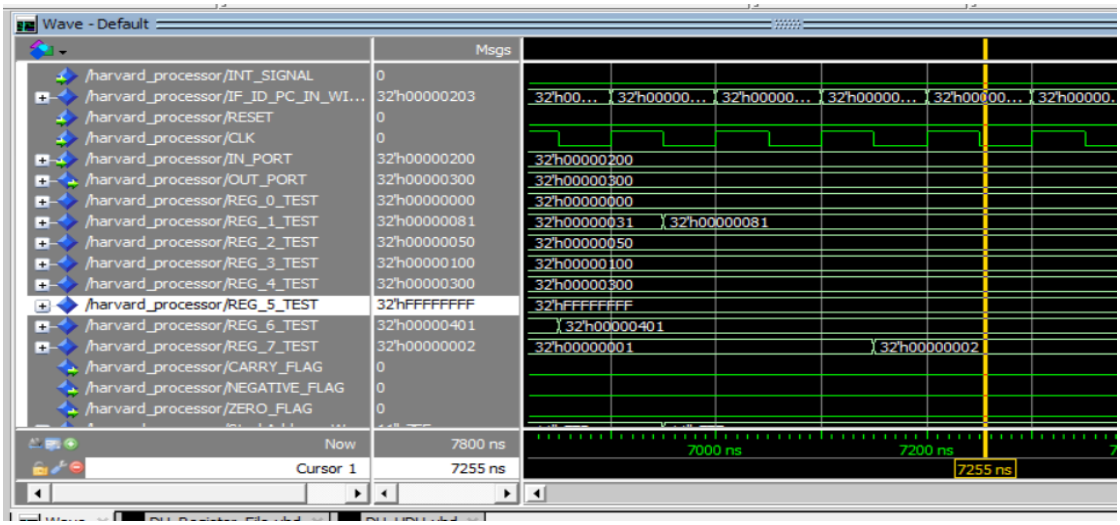
in the below picture the instruction inc R1 was not supposed to happen



in the below picture the instruction inc R6 was not supposed to happen



in the below picture the instruction inc R7 was not supposed to happen



Case III: Flushing Off

One Operand did not suffer any change and that is expected because this test case does not contain any need for flushing

Two Operand did not suffer any change and that is expected because this test case does not contain any need for flushing

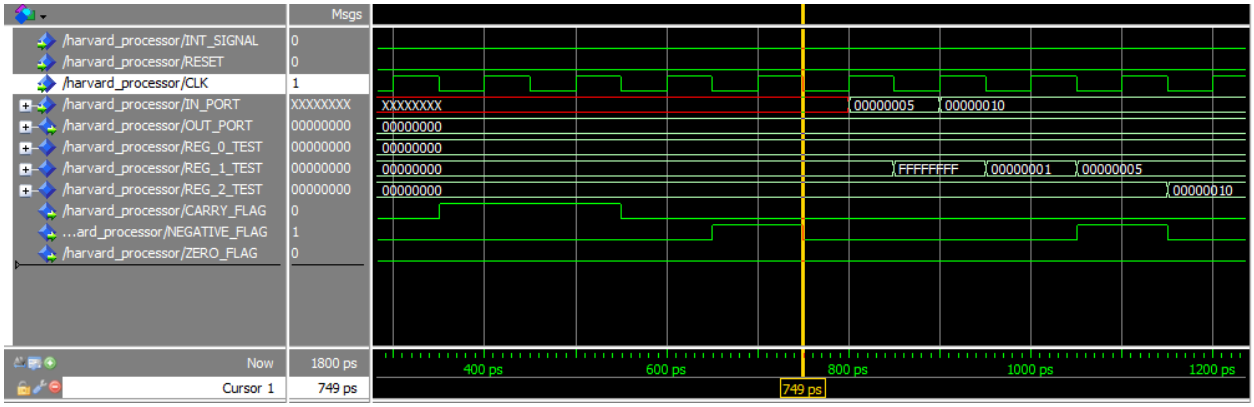
Memory did not suffer any change and that is expected because this test case does not contain any need for flushing

Branch suffered from the exact same problems as the one mentioned above since the above problems were all due to the flushing and none due to the load case use

Branch Prediction also suffered from the same problems as above since they were all due to flushing and none to memory or load case use

Case IV:All Turned Off

One Operand



The cursor is now at 750 ps:

Fetch =>in R2

Decode => in R1

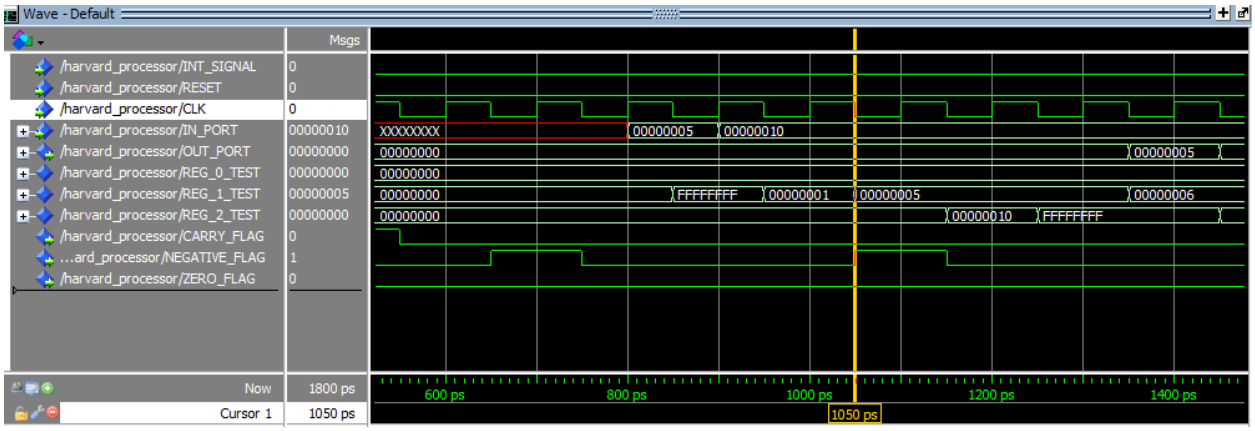
Execute =>inc R1

Memory =>NOT R1

WriteBack =>CLRC

Hazard

A data hazard occurs here as the execute stage of inc R1 uses the old value of R1 (00000000) instead of the value that's the result (FFFFFFFF) of the previous instruction (NOT R1).



The cursor is now at 1050 ps:

Fetch =>Dec R2

Decode => inc R1

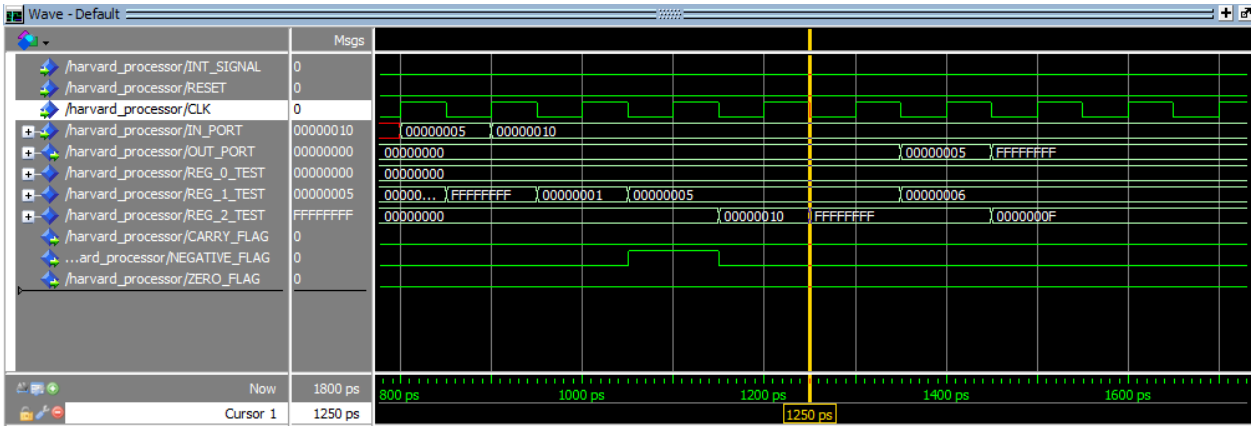
Execute => NOT R2

Memory =>in R2

WriteBack =>in R1 #R1 = 00000005

Hazard

Another Data Hazard occurs here as in the execute stage of NOT R2. Normally, the instruction should use the value (00000010) that is the result of the previous instruction (IN R2). Instead, it uses the old value of R2 which is (000000000).



The cursor is now at 1250 ps:

Fetch => out R2

Decode => out R1

Execute =>Dec R2

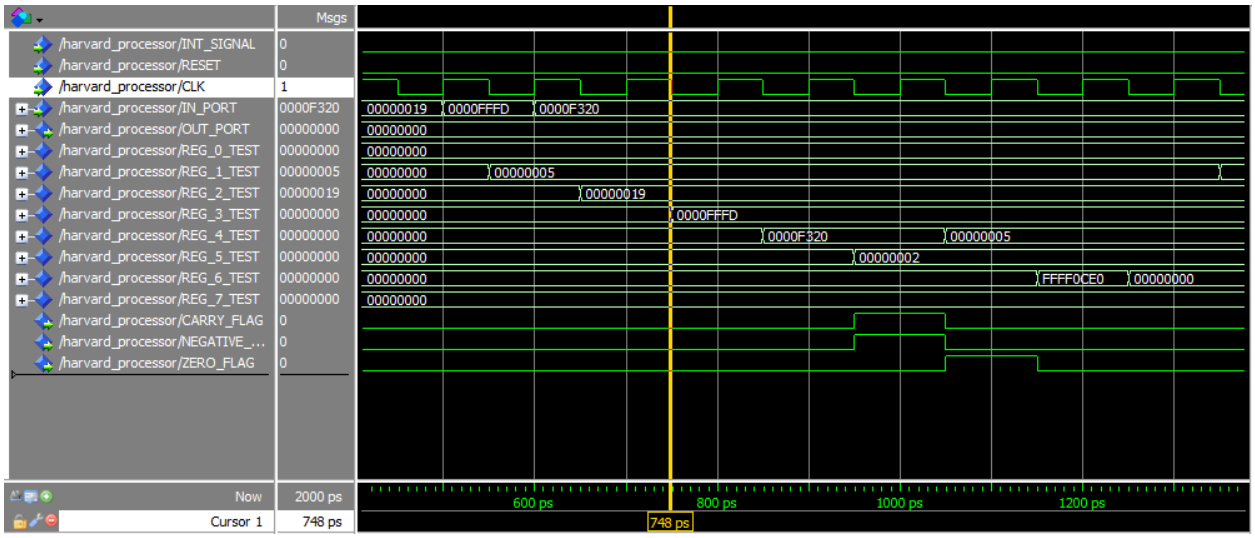
Memory =>inc R1

WriteBack =>NOT R2 #R2 =FFFFFFFF -WRONG

Hazard

The final data hazard starts here where the instruction DEC R2 should be using the value of R2 that's the result of (NOT R2), but instead it uses the old value of R2 which is (00000010) .

TWO OPERAND



The cursor is now at 750 ps:

Fetch => SUB R5,R4,R6

Decode => ADD R1,R4,R4

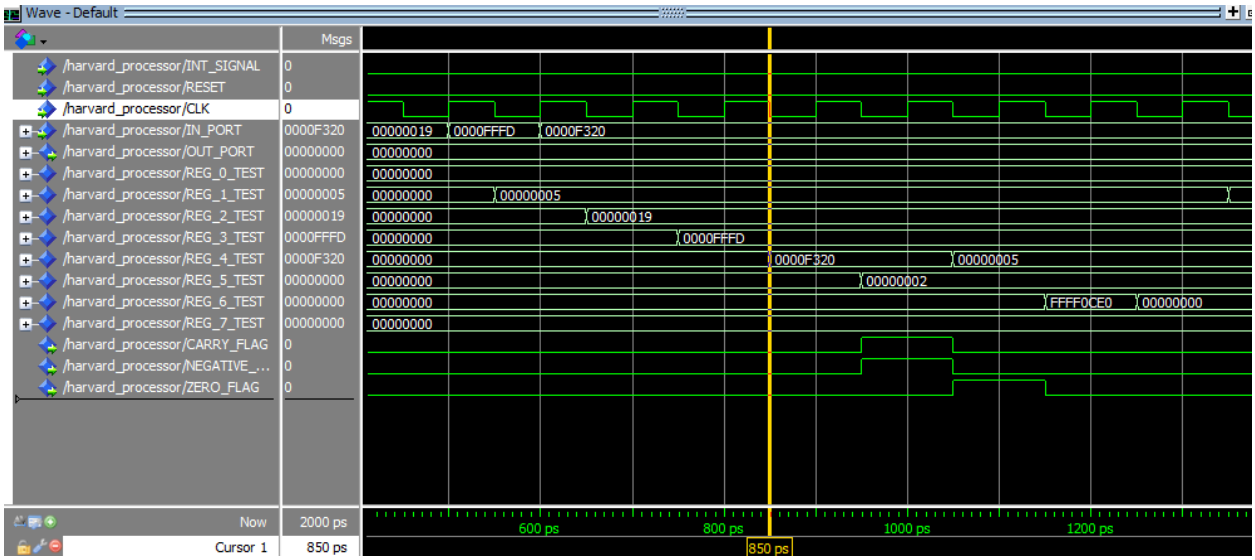
Execute => IADD R3,R5,2

Memory =>IN R4

WriteBack => IN R3 #R3 = FFFD

Hazard

A data hazard occurs here as the (IADD R3,R5,2) enters the execution stage. It uses the old value of R3 (00000000) instead of the value (0000FFFD) that results from a previous instruction(IN R3).



The cursor is now at 850 ps:

Fetch => AND R7,R6,R6

Decode => SUB R5,R4,R6

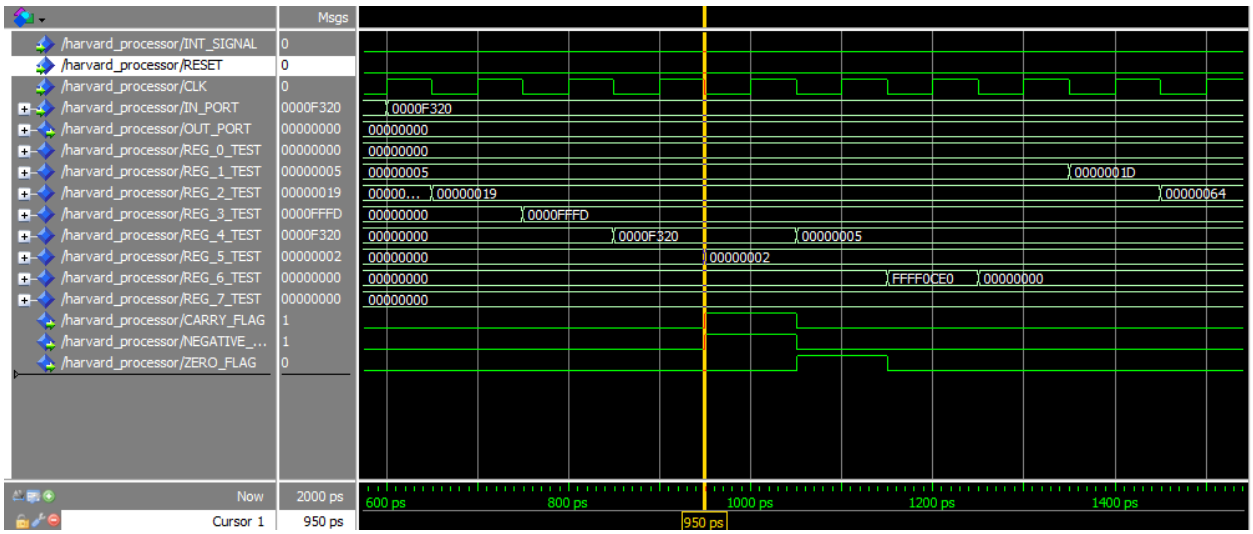
Execute => ADD R1,R4,R4

Memory =>IADD R3,R5,2

WriteBack =>IN R4 #R4 = 320

Hazard

A data hazard occurs here as the execute stage of (ADD R1,R4,R4) uses the value of R4 as (00000000) instead of the new value (0F320) coming from the instruction (IN R4).



The cursor is now at 950 ps:

Fetch => OR R2,R1,R1

Decode => AND R7,R6,R6

Execute => SUB R5,R4,R6

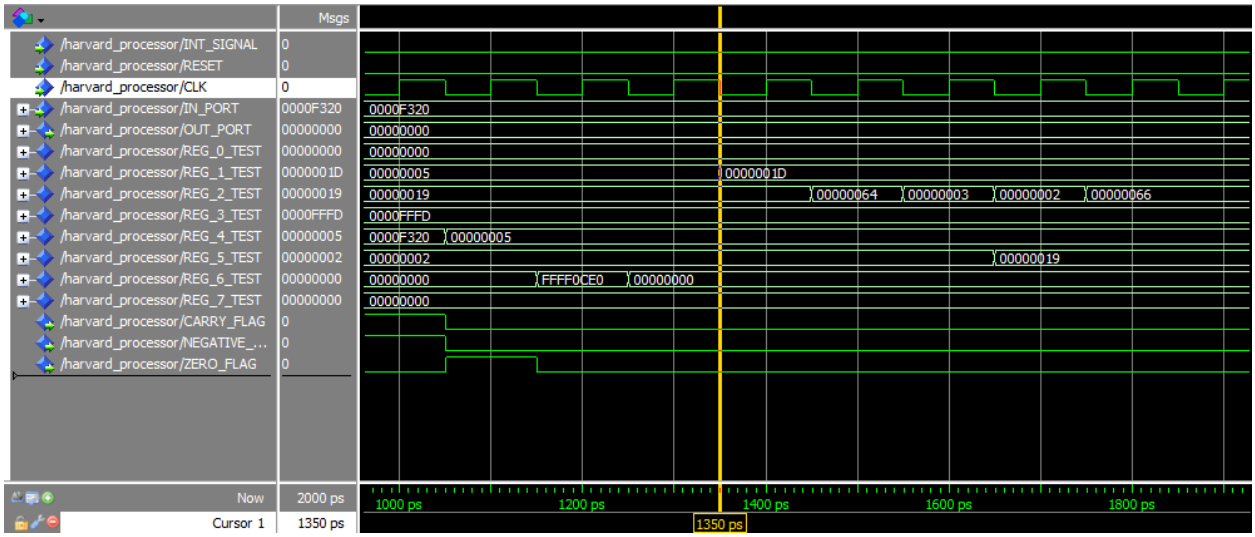
Memory =>ADD R1,R4,R4

WriteBack =>IADD R3,R5,2 #R5 = 02 -WRONG R2 VALUE

Hazard

A data hazard occurs here as the execute stage of (SUB R5,R4,R6) uses the value of R4 as (F320) instead of the new value (05, which is incorrect from an earlier data hazard) coming from the instruction (ADD R1,R4,R4).

The value for R5 is also incorrectly taken as (00000000) instead of (00000002) coming from IADD R3,R5,2..



The cursor is now at 1350 ps:

Fetch =>ADD R5,R2,R2

Decode => SWAP R2,R5

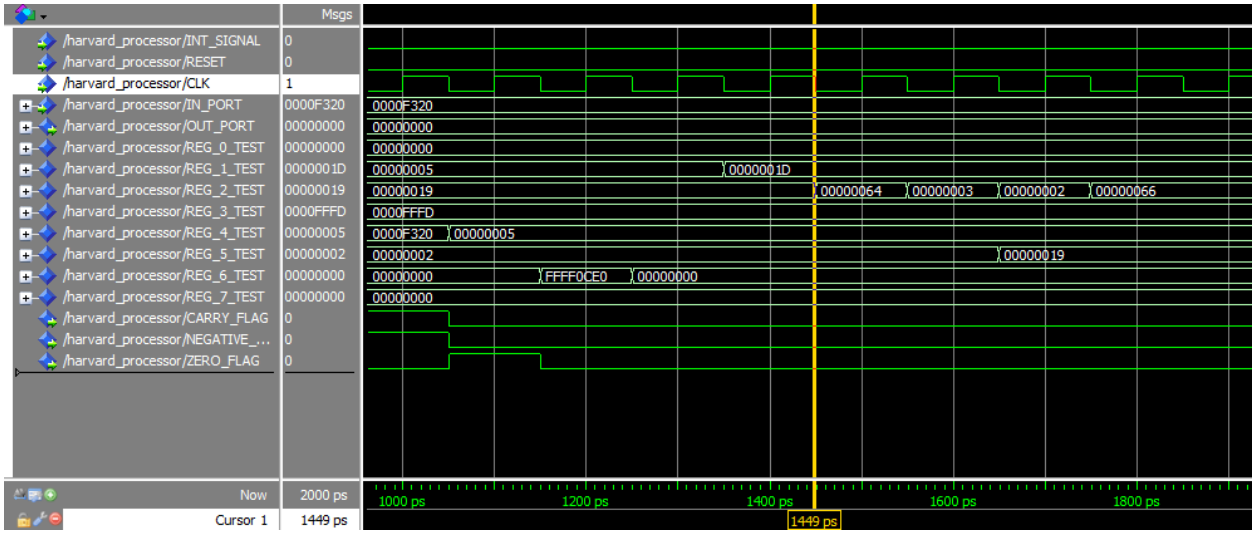
Execute => SHR R2,3

Memory =>SHL R2,2

WriteBack =>OR R2,R1,R1 #R1 = 1D

Hazard

Another Data Hazard occurs here as SHR R2,3 in the execution stage takes the old value of R2 (019) instead of the value that's output from SHL R2, 2 (064).



The cursor is now at 1450 ps:

Decode => ADD R5,R2,R2

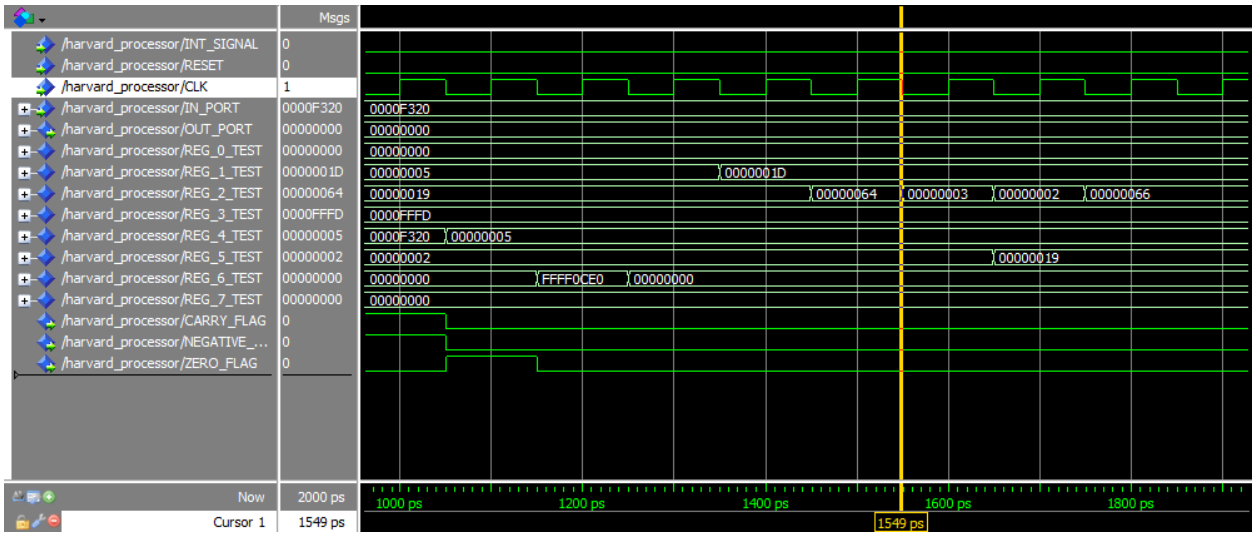
Execute => SWAP R2,R5

Memory =>SHR R2,3

WriteBack =>SHL R2,2 #R2 = 64

Hazard

Another Data Hazard occurs here as SWAP R2,R5 in the execution stage takes the old value of R2 (019) instead of the value that's output from SHR R2, 3 (03, which is also wrong due to an earlier hazard).



The cursor is now at 1550 ps:

Execute =>ADD R5,R2,R2

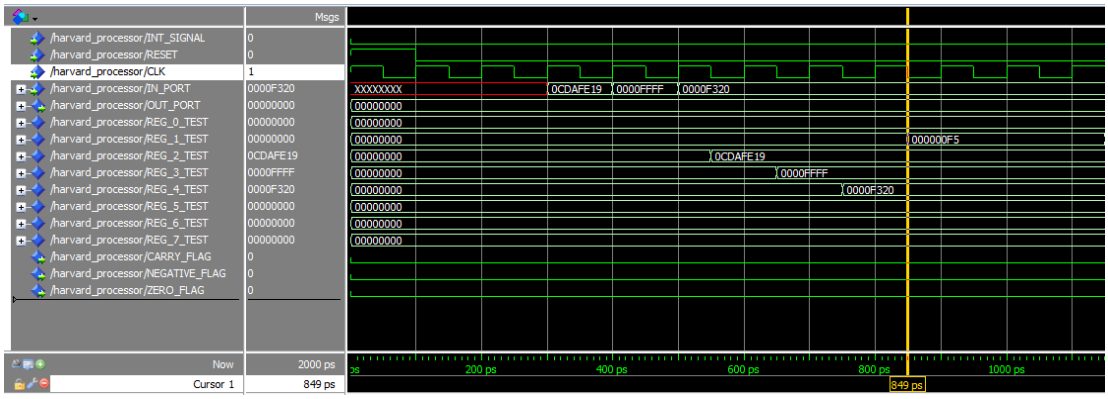
Memory =>SWAP R2,R5

WriteBack =>SHR R2,3 #R2 = 3

Hazard

The final hazard that occurs is the execution stage of ADD R5,R2,R2 takes R2 as (64) instead of (02) and R5 as (02) instead of (019).

Memory



The cursor is now at 850ps which means:

Fetch =>POP R2

Decode =>POP R1

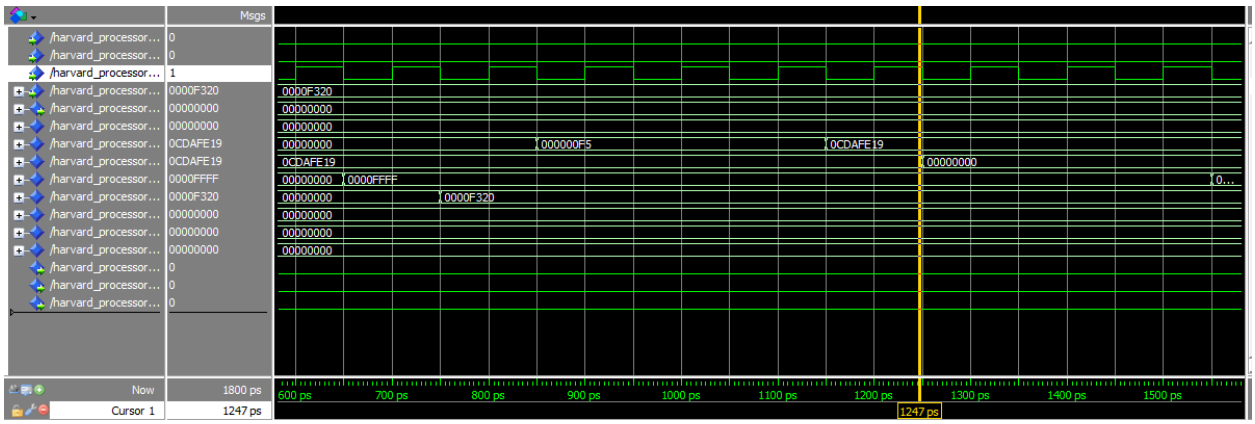
Execute => PUSH R2

Memory =>PUSH R1

WriteBack => LDM R1,F5

Hazard

A data hazard occurs here in the memory stage of Push R1 as R1 is seen as (00000000) instead of F5 which is the result of LDM R1, F5



The cursor is now at 1250ps which means:

Fetch =>LDD R4,200

Decode =>LDD R3,202

Execute =>STD R1,202

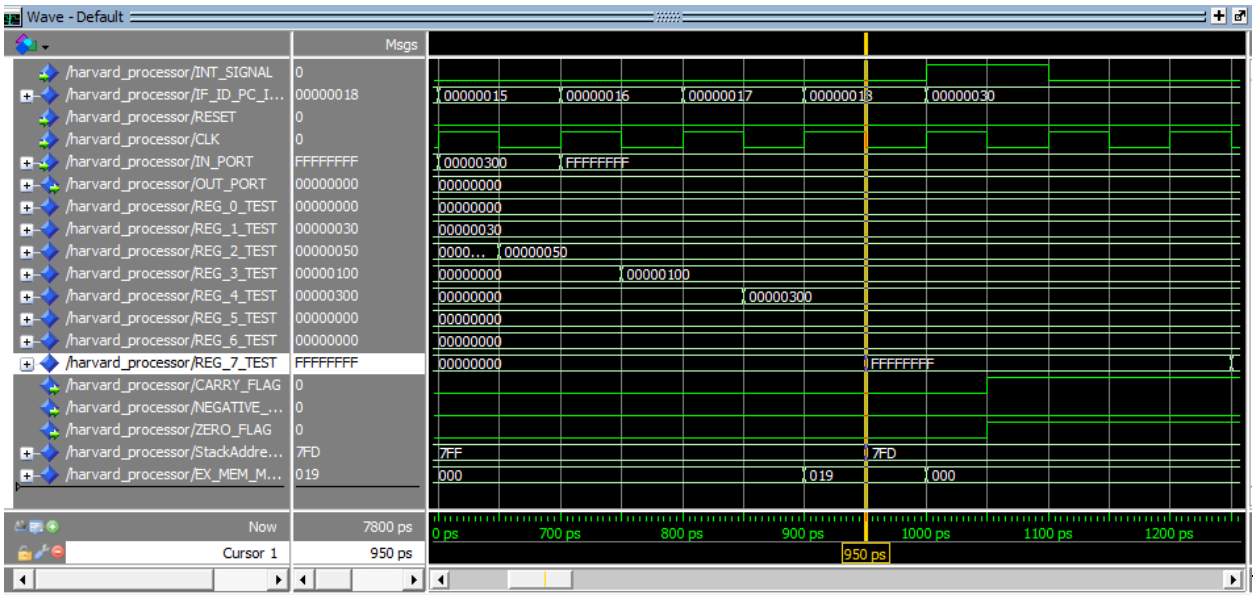
Memory => STD R2,200

WriteBack =>POP R2

Hazard

A hazard occurs here in the memory stage of (STD R2, 200) as the instruction uses the value of R2 as (0CDAFE19) instead of (00000000, which is also wrong due to an earlier hazard).

Branch



The cursor is at 950 ps :

Fetch =>NOP

Decode =>INC R7

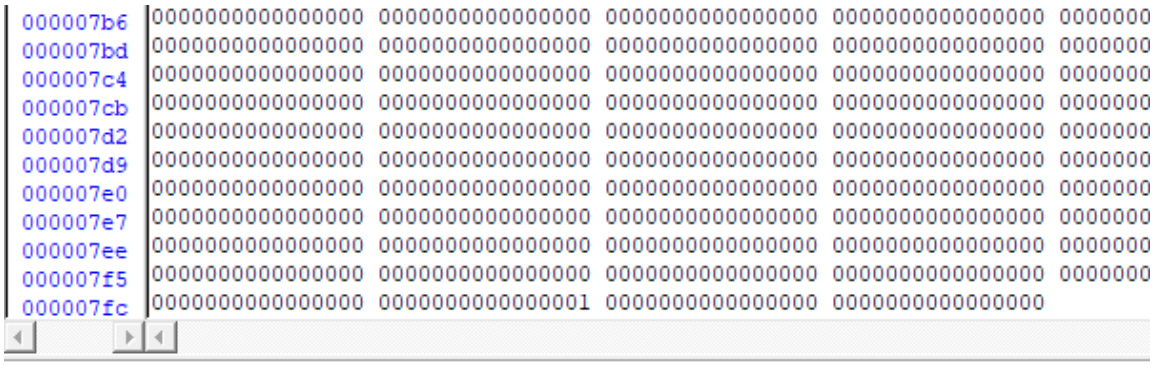
Execute => Jmp R1

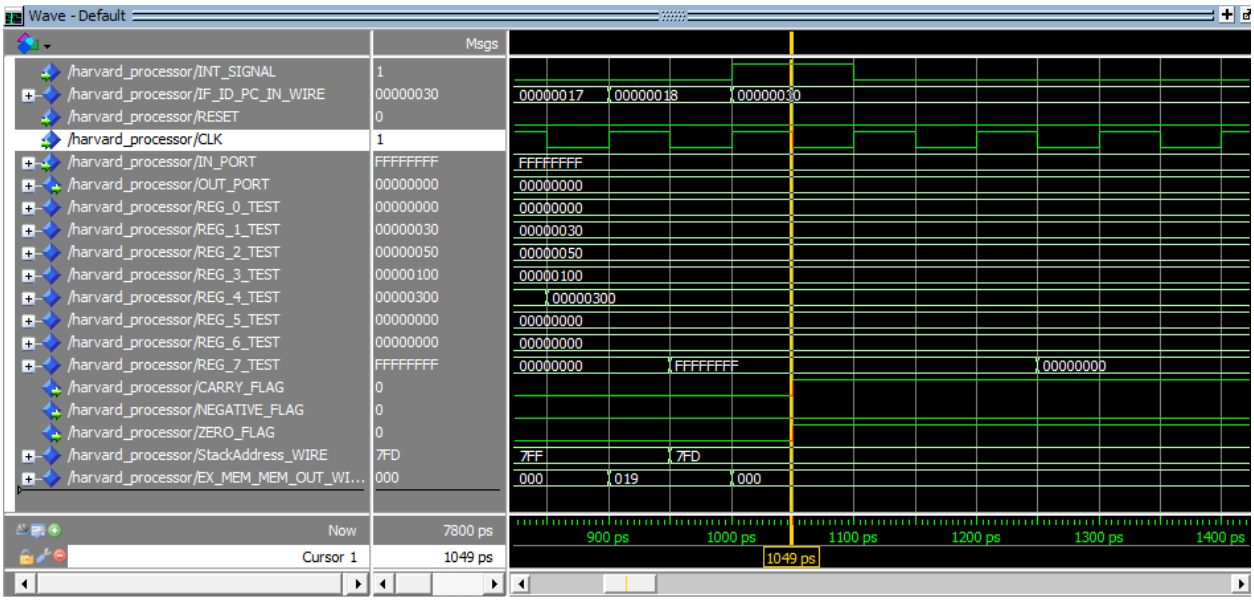
Memory => PUSH R4

WriteBack => IN R7

Hazard

Data Hazard occurs here as the value of R4 stored in the memory is 00 not 300 as visible in the next screenshot.





The cursor is at 1049 ps:

Fetch =>NOP

Decode =>NOP

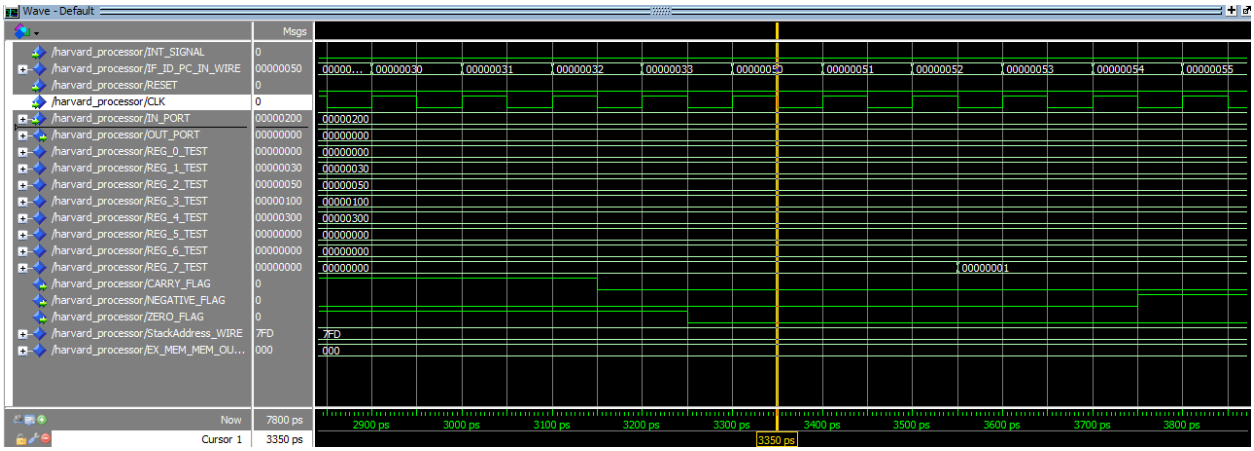
Execute => INC R7

Memory =>JMP R1

Writeback =>PUSH R4

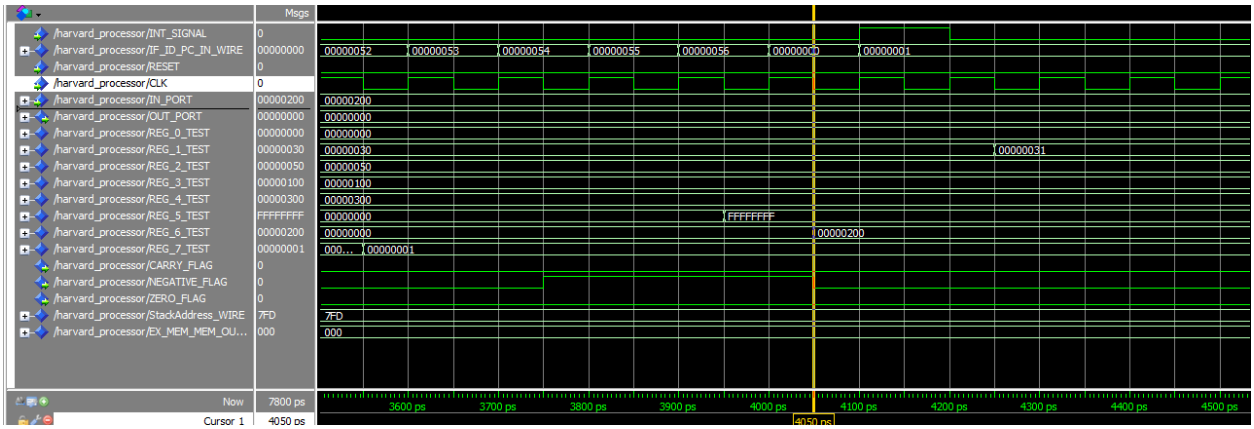
Hazard

This is a control hazard. Due to the flushing being turned off, when the JMP was executed at the last cycle, the previous buffers weren't flushed. So, the instruction (INC R7) which follows the JMP R1 (and shouldn't be executed) was executed.



Control Hazard:

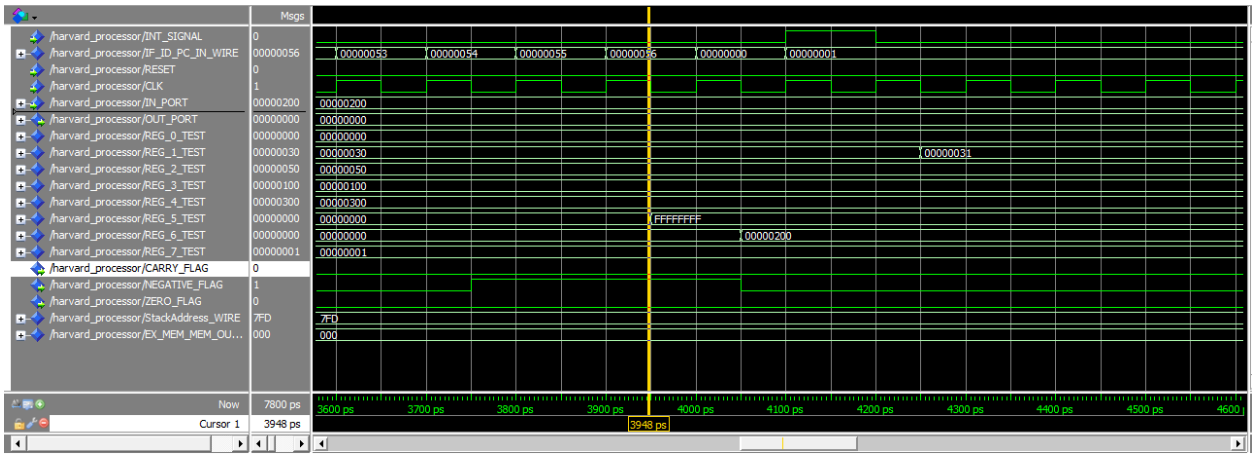
Execute stage of the instruction INC R7, the instruction shouldn't be executed as it's preceded by JZ R2 which should be taken, however due to turning off flushing, it executes normally.



Cursor at 4050:

Control Hazard:

Execute stage of INC R1 is incorrectly executed due to turning off the flushing, as this instruction is preceded by JN R6.

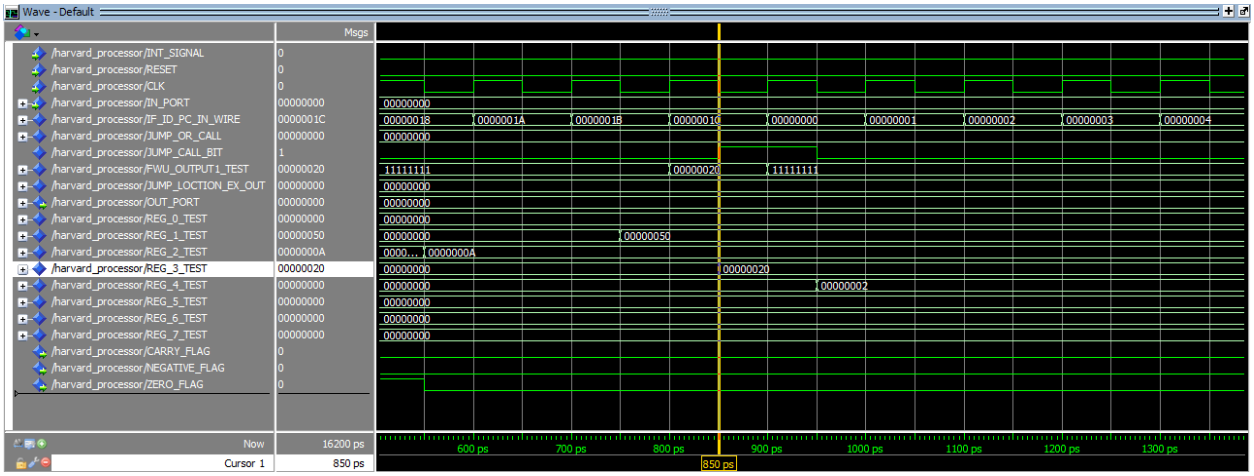


Cursor at 3948

Data Hazard

JN R6 uses the old value of R6(00000000) instead of the new value (00000200), due to turning off the forwarding unit.

Branch Prediction



A data hazard occurs here as the instruction JMP R3 executes. The correct value that the instruction should use is 20, but instead it uses (00000000). This of course can be solved by using a forwarding unit.

This resets the instructions to zero, which spoils the whole program.