|  |  |  |
| --- | --- | --- |
| **Credit Hours System**  **GENN332(Spring 2020)** |  | **Cairo University**  **Faculty of Engineering** |

**Computer Architecture**

**Final Assessment**

**Report**

**Submission Date:** 02/06/2020

**Submitted to:** Dr. Mayada Hadhoud

Table of Contents

[**Abstract** 4](#_Toc41357484)

[**Introduction** 5](#_Toc41357485)

[**ISA Specifications** 6](#_Toc41357486)

[**Specifications** 6](#_Toc41357487)

[**Registers** 6](#_Toc41357488)

[**Input-Output** 6](#_Toc41357489)

[**Instructions** 7](#_Toc41357490)

[**Design** 9](#_Toc41357491)

[**Full Design** 9](#_Toc41357492)

[**Fetching Stage** 10](#_Toc41357493)

[3- **Adder B** 11](#_Toc41357494)

[4- **Mux A** 11](#_Toc41357495)

[5- **Mux B** 11](#_Toc41357496)

[6- **Mux C** 11](#_Toc41357497)

[7- **INT UNIT** 11](#_Toc41357498)

[8- **Instruction Memory:** 11](#_Toc41357499)

[9- **Mux D:** 11](#_Toc41357500)

[10- **IF/ID Buffer:** 11](#_Toc41357501)

[**Decode Stage** 12](#_Toc41357502)

[1- **Control Unit:** 12](#_Toc41357503)

[2- **Mux A** 17](#_Toc41357504)

[3- **Mux B** 17](#_Toc41357505)

[4- **Register File** 17](#_Toc41357506)

[5- **HDU** 17](#_Toc41357507)

[6- **ID/EX Buffer** 17](#_Toc41357508)

[**Execute Stage** 18](#_Toc41357509)

[**Zero Extender** 18](#_Toc41357510)

[**ALU** 18](#_Toc41357511)

[**Flags Register:** 19](#_Toc41357512)

[**Outport** 19](#_Toc41357513)

[**Inport**: 19](#_Toc41357514)

[**Forwarding Unit**: 20](#_Toc41357515)

[**General Flow:** 20](#_Toc41357516)

# **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.

(N.B.  word = 2 bytes)**.** **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)

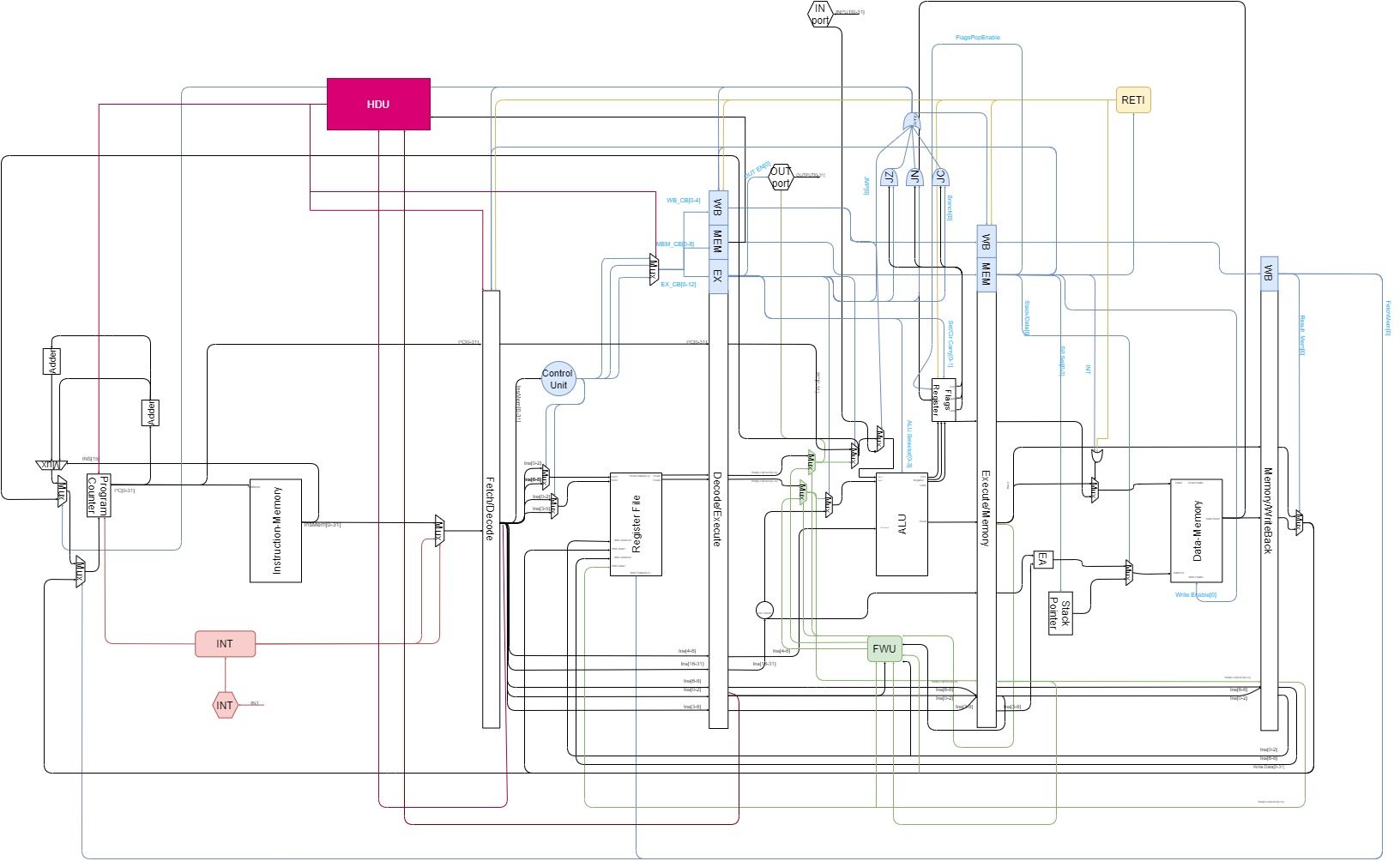
### **Instructions**

|  |  |  |
| --- | --- | --- |
| **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 |  |

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

1. **Mux A**: 2x1 mux which selects between “Rdst” & “Rsrc1” based on the selector coming from control unit.
2. **Mux B**: 2x1 mux which selects between “Rdst” & “Rsrc2” based on the selector coming from control unit.
3. **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:
   1. “Address 1 Data”, 32 bits
   2. “Address 2 Data” 32 bits
   3. “Address 1” 32 bits
   4. “Address 2” 32 bits
   5. “Write Selector”, 32 bits
4. **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.
5. **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**



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

**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

**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.

**Outport:** takes a one bit enable, and 32 bits value to output.

**Inport**:takes a 32 bit value.

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

**3 AND gates and 1 OR gate.**

**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.

**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.

### **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**

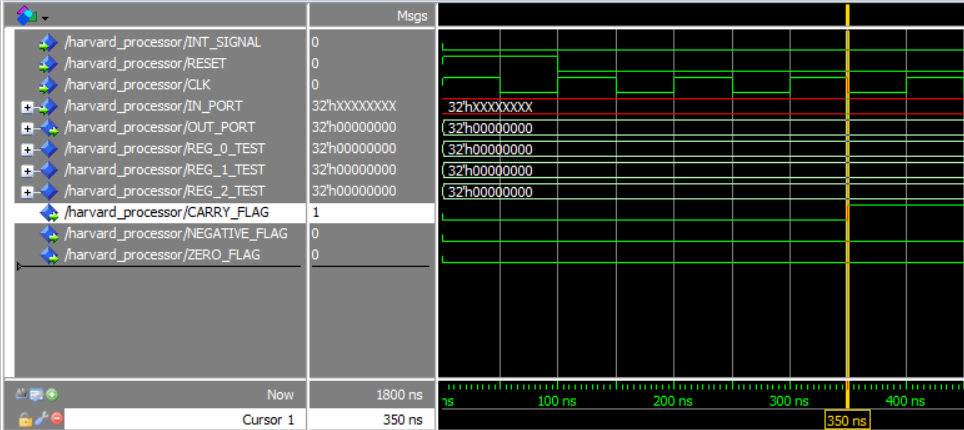
**Inputs:**

* 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.

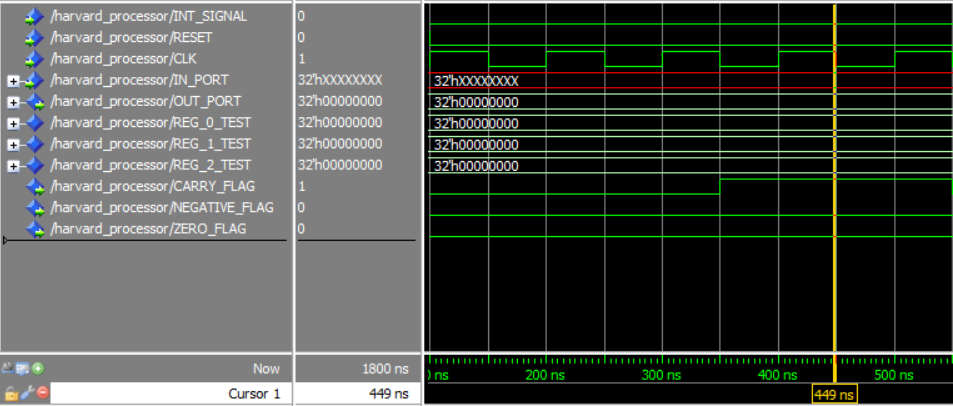
# **One Operand Test Cases**



The cursor above is at 350 ns which means:

Fetch => CLRC

DECODE => NOP

EXECUTE => SETC CF = 0 , NF = S , ZF = S \*S MEANS NO CHANGE\*

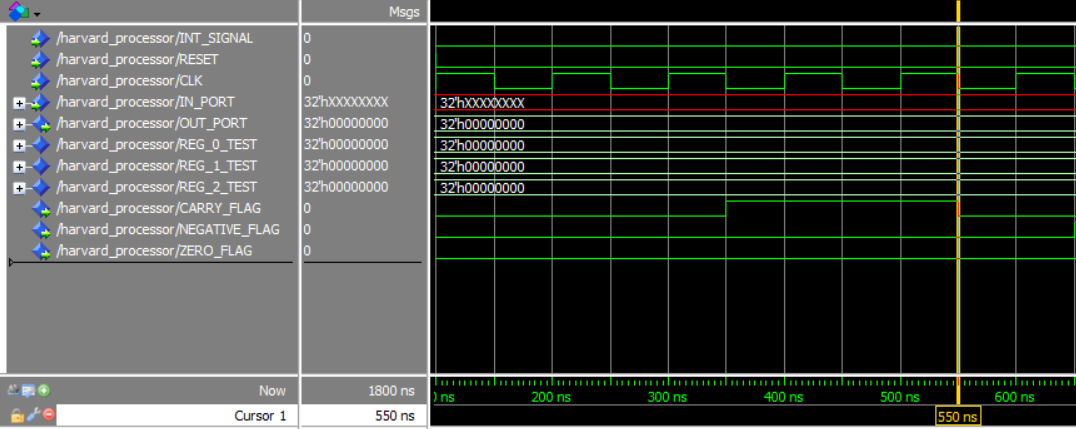
The cursor above is at 450 ns which means:

Fetch => NOT R1

DECODE => CLRC

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

MEMORY => SETC



The cursor above is at 550 ns which means:

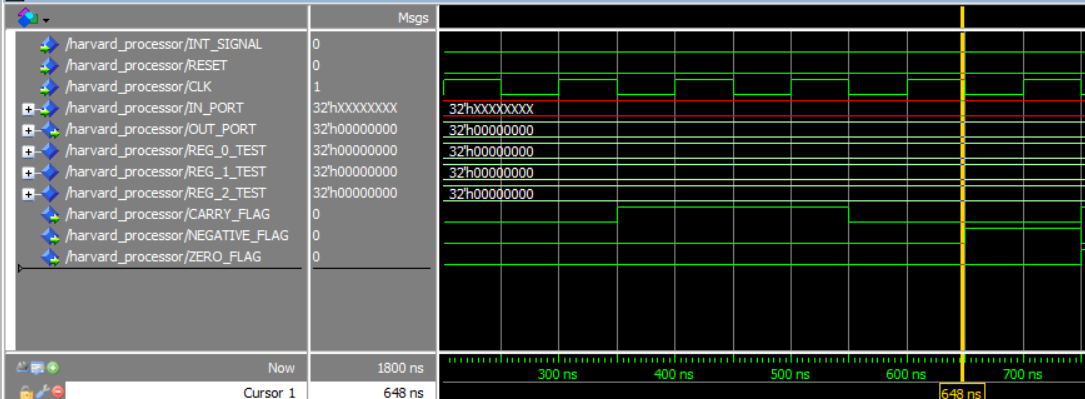
Fetch => INC R1

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:

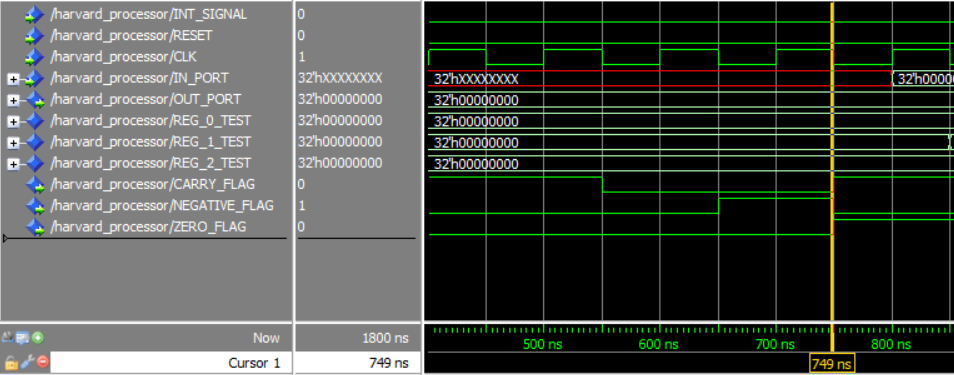
Fetch => IN R1

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:

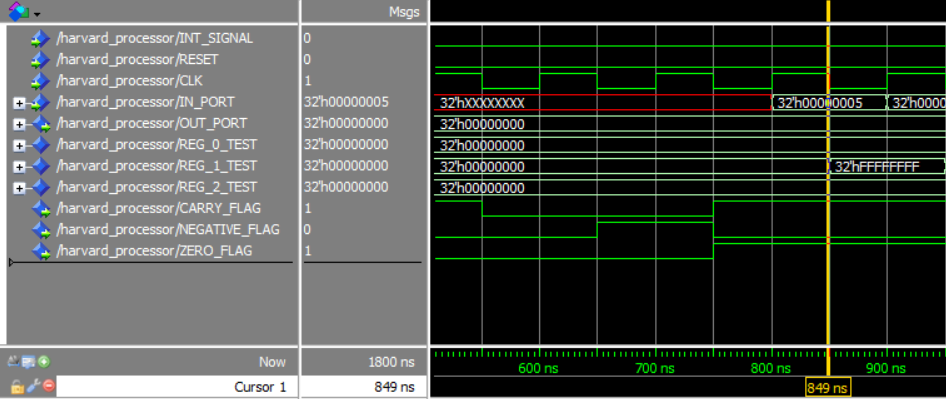
Fetch => IN R2

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:

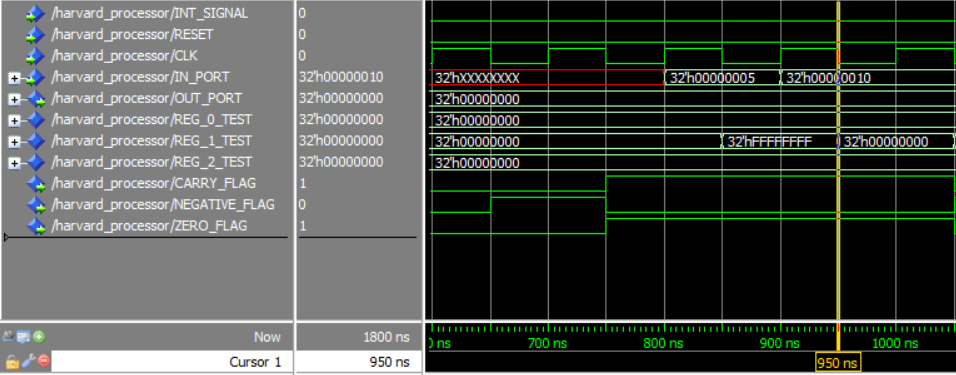
Fetch => NOT R2

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:

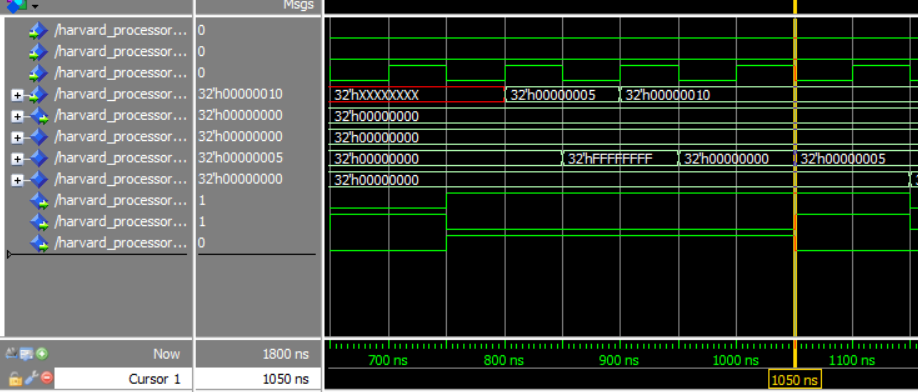
Fetch => INC R1

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:

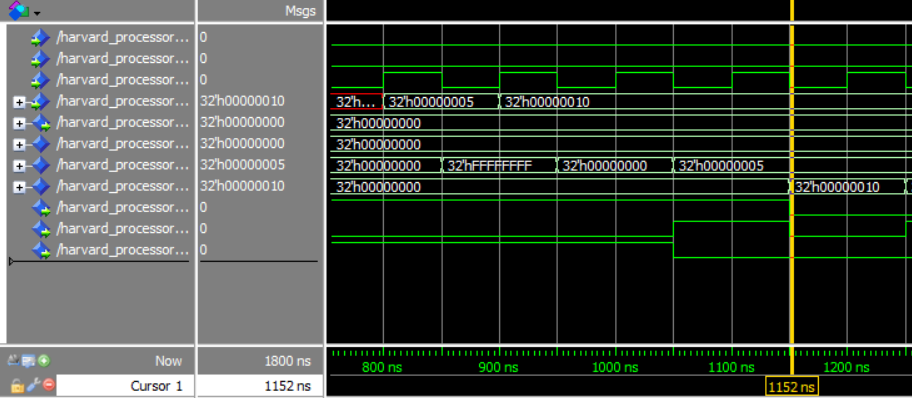
Fetch => DEC R2

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:

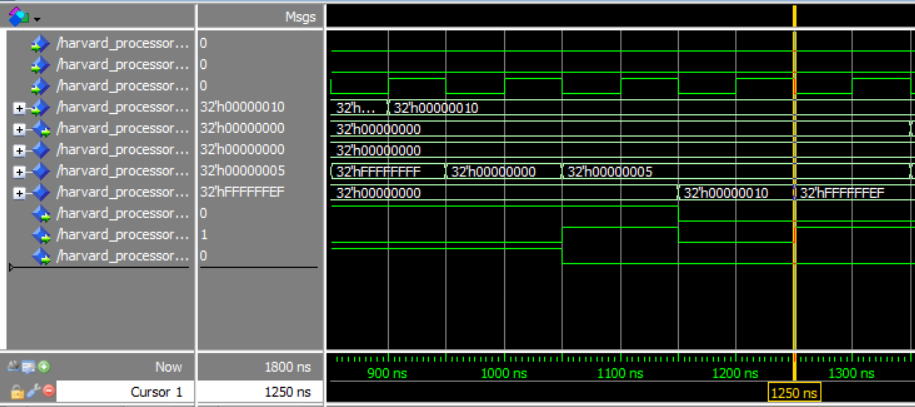
Fetch => OUT R1

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:

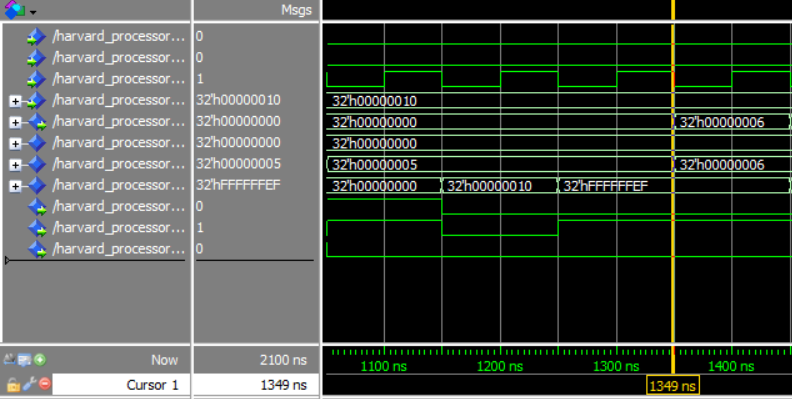
Fetch => OUT R2

DECODE => OUT R1

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

MEMORY => INC R1

WRITE-BACK => NOT R2 R2= FFFFFFEF



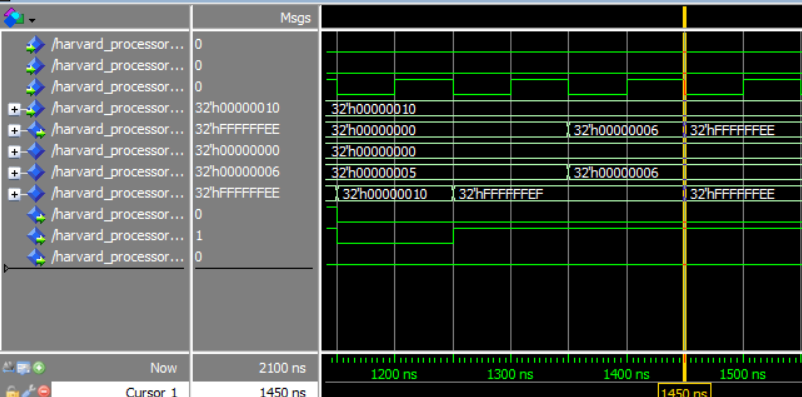
The cursor above is at 1350 ns which means:

DECODE => OUT R2

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

MEMORY => DEC R2

WRITE-BACK => INC R1 R1= 6



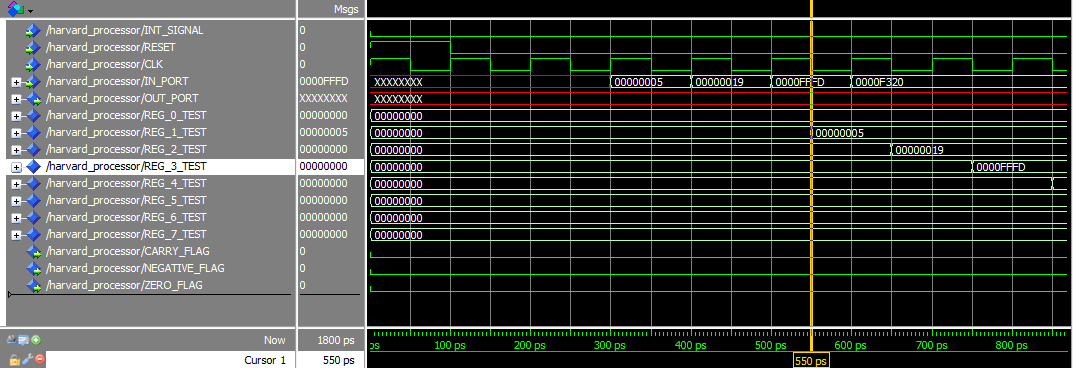
The cursor above is at 1450 ns which means:

EXECUTE => OUT R2 CF = S , NF = S , ZF = S OUT PORT = FFFFFFEF

MEMORY => OUT R1

WRITE-BACK => DEC R2

# **Two Operand Test Cases**



The Cursor above is at 550 ps which means:

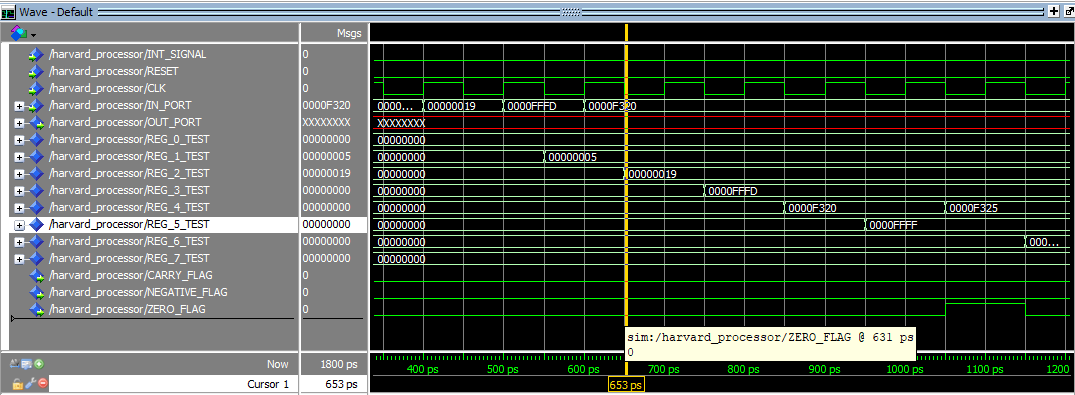
Fetch => IADD R3,R5,2

Decode => IN R4

Execute => IN R3

Memory =>IN R2

WriteBack =>IN R1   R1 = 00000005



The Cursor above is at 653 ps which means:

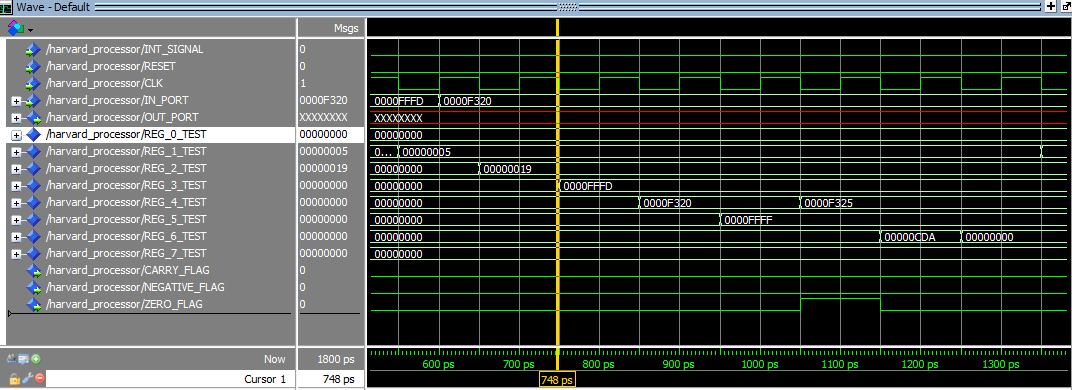
Fetch => ADD  R1,R4,R4

Decode => IADD R3,R5,2

Execute =>IN R4

Memory =>IN R3

WriteBack =>IN R2   R2= 00000019



The Cursor above is at 748 ps which means:

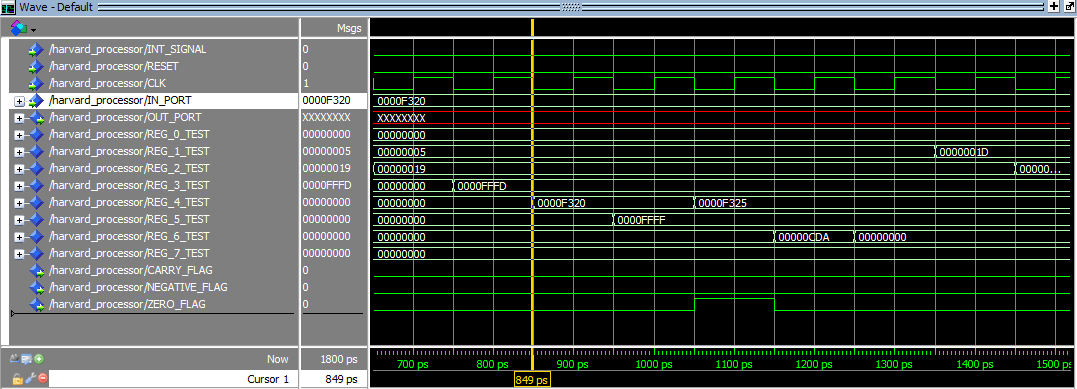
Fetch => SUB  R5,R4,R6

Decode => ADD  R1,R4,R4

Execute =>IADD R3,R5,2

Memory =>IN R4

WriteBack =>IN R3   R3 = 0000FFFD



The Cursor above is at 849 ps which means:

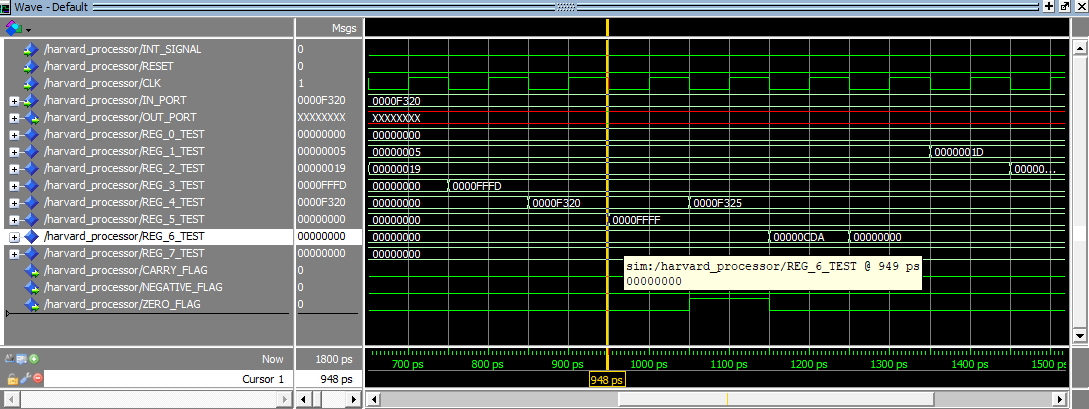
Fetch => AND  R7,R6,R6

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 849 ps which means:

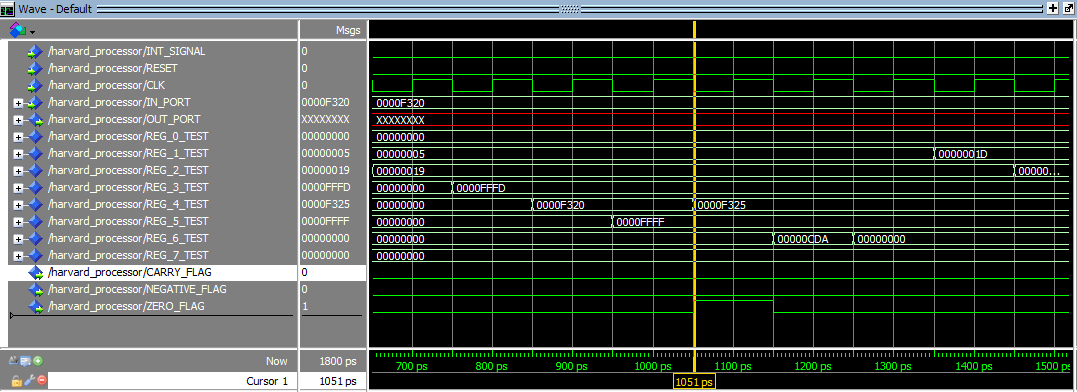
Fetch =>OR   R2,R1,R1

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 1051 ps which means:

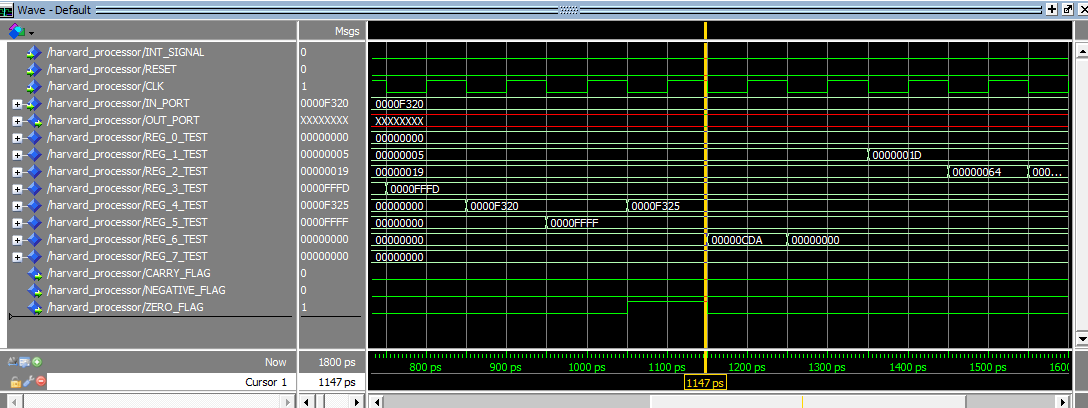
Fetch =>SHL  R2,2

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 1147 ps which means:

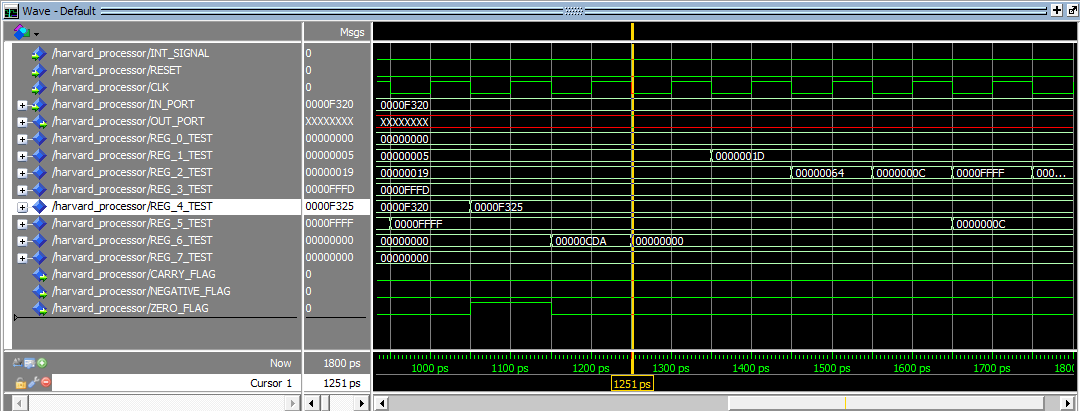
Fetch =>SHR  R2,3

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 1251 ps which means:

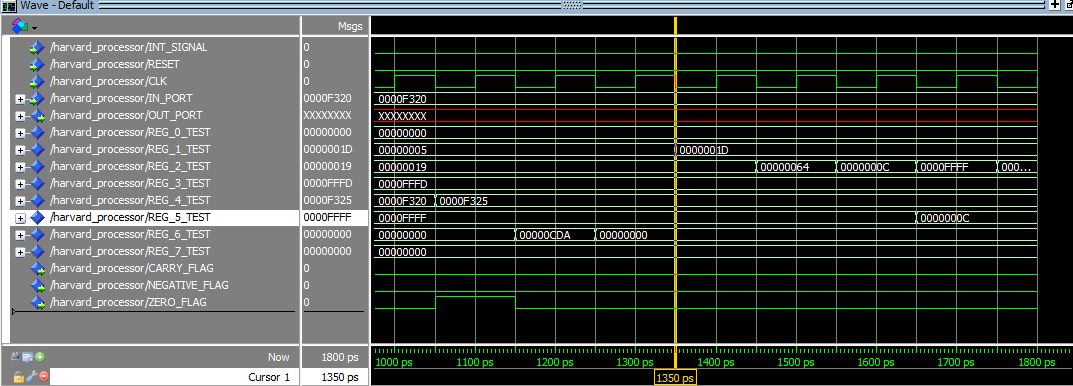
Fetch =>SWAP R2,R5

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:

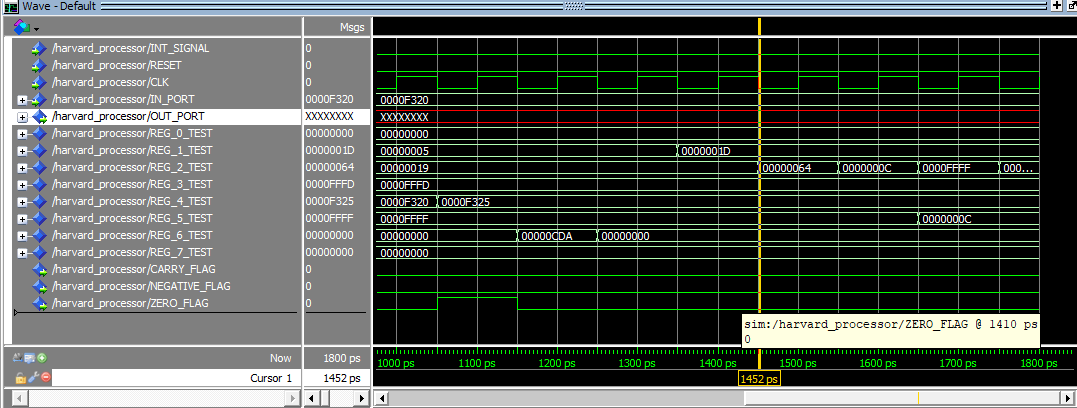
Fetch =>ADD  R5,R2,R2

Decode => SWAP R2,R5

Execute =>SHR  R2,3

Memory =>SHL  R2,2

WriteBack =>OR   R2,R1,R1    #R1 = 0000001D



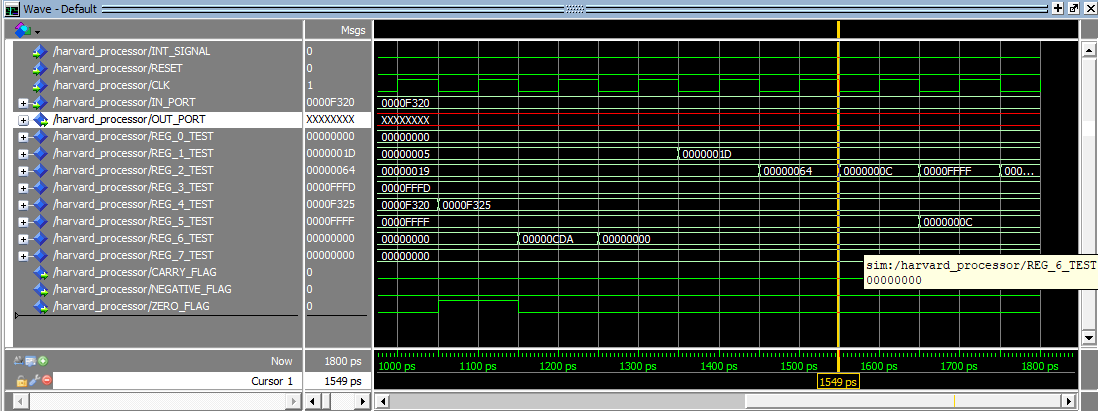
The Cursor above is at 1452 ps which means:

Decode => ADD  R5,R2,R2

Execute =>SWAP R2,R5

Memory =>SHR  R2,3

WriteBack =>SHL  R2,2   #R2 = 00000064

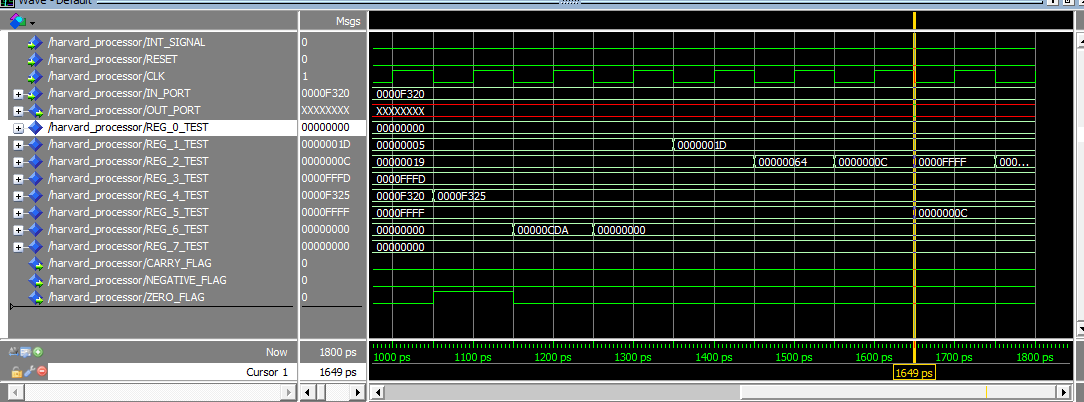


The Cursor above is at 1549 ps which means:

Execute =>ADD  R5,R2,R2

Memory =>SWAP R2,R5

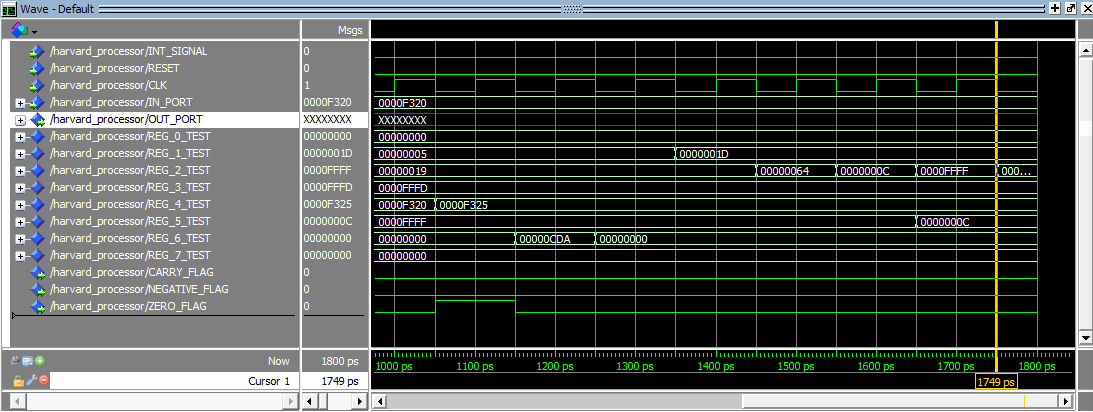
WriteBack =>SHR  R2,3   #R2 = 0000000C



The Cursor above is at 1649 ps which means:

Memory =>ADD  R5,R2,R2

WriteBack =>SWAP R2,R5       #R2 = 0000FFFF       #R5 = 0000000C



The Cursor above is at 1749 ps which means:

WriteBack =>ADD  R5,R2,R2  #R2 = 0001000B

