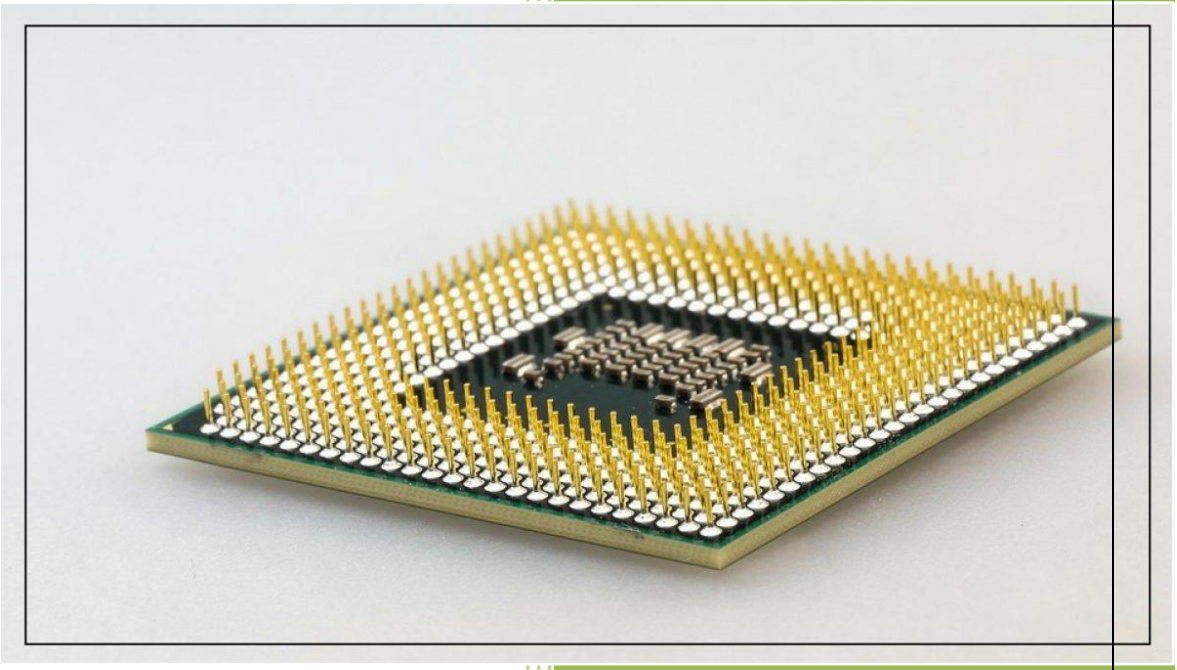
Spring 2020

Computer Architecture Project



The A-Processor 101

Team 5

Names:

Ahmed Abdou Allam (2180001)

Hazem Nagi Morsi Ahmed (1165024)

Mahmoud Aref Mahmoud(1165022)

Mahmoud Ahmed Atwa (1162014)

Processor characteristics and Design assumptions:

This processor is fully working correctly with direct mapped cache system with Harvard architecture, and with all specifications and the ISA mentioned in the project's document.

- **Design Assumptions:**
  - Assuming that any location addressed in the data cache/ memory will start with an even address (as mentioned in document to avoid splits).
  - The instruction can split among different blocks and it is handled perfectly by our processor.
  - The processor can handle Nested interrupts and the interrupt is **Level Triggered**.
  - The processor has full forwarding and static branch prediction not taken.
  - Out Port, in Port and flag content changes are always done in execute stage.
  - In cache the miss that arrives first gets to be served first except if data and instruction miss happens at the same time the data miss will be handled first.
  - The data miss causes the processor to be stalled(except write back stage) but the instruction miss keeps working and forwarding NOP's (will give us higher CPI along with the next assumption)
  - Main memory process is assumed to have the ability to be interrupted i.e. if there is a jump and an instruction miss the PC value will be replaced and the cache memory system will handle the new address and this will decrease the number of cycles needed.
  - Note in perfect cache you have to load both memories as RST and Interrupt address are in the Data Memory
- **Design Schematics and Op-Codes:**
  - You can find the full design in the following link: A-Processor101
  - You can find the Cache/Memory System design in the following link: Memory_Cache
  - You can find the Op-Codes used in the following Link: Op-Codes
- **Attachments:**
  - **AProcessor_Final:** this folder contains the code (full processor with memory cache system) along with Memory files (**.HEX**) and does files (**.DO**) in folders memory_files and do_files respectively.
  - **PerfectCache:** this folder contains the code (assuming perfect cache i.e. the processor with no cache) along with Memory files (**.HEX**) and does files (**.DO**) in folders memory_files and do_files respectively.
  - **AssemblerEXE:** this folder contains our assembler EXE File.
  - **AssemblerCode:** this folder contains our assembler's code.
  - **Assembler_Manual.pdf:** This shows how the assembler is working and restrictions on how to use it.
- **Important Notes:**
  - **Our assembler works different than required as it deals with every numbers in (Decimal) and doesn't ignore comments (everything is mentioned in the assembler manual doc file).**

The main units of design description:

1. Control Unit.
   Inputs: instruction [31…25]
   Has 27 output signals and one output bus of Length 3
   The control unit follows the following truth table attached in Op-Codes document attached.

2. Hazard detection unit "HDU":
   Inputs (6):
   - is_Load/Pop

- Has_SRC2
- Has_SRC1
- Src1_ADD
- SRC2_ADD
- LDD_Address

Output (1)

- Hazard_detected

The truth table is attached in Op-Codes file

And it follows the following Algorithm:

```
function isThereAHazard() {
  if (!is_Load) return false;
  if (Has_Src1 && Src1_ADD === LDD_Address) {
    return true;
  } else if (Has_Src2 && Src2_ADD === LDD_Address) {
    return true;
  }
  return false;
}
```

3. Forwarding_Unit "FU":
   Inputs (4 signals, 6 busses each of Length 3):
   - Reg_Dest_EX 3 bits
   - Reg_Dest_MEM 3 bits
   - SRC_SWAP_MEM 3 bits
   - SRC_SWAP_EX 3 bits
   - REG_SRC_ADD1 3 bits
   - REG_SRC_ADD2 3 bits
   - Is_Swap_MEM
   - Is_Swap_Ex
   - WB_MEM
   - WB_EX

   Outputs (2 busses each of Length 3)

   - FWD_A 3 bits
   - FWD_B 3 bits

And it follows the following Algorithm:

```
//FWDA

if (WB_Ex == 1)
{
```

```
        if (is_Swap_EX)
        {
                if (Reg_SRC_ADD1 == SRC_SWAP_EX)
                        FWD_A = 100;
                else if (REG_SRC_ADD1 == REG_DEST_EX)
                        FWD_A =011 ;
        }
        else
        {
                if (Reg_SRC_ADD1 == REG_DEST_EX)
                        FWD_A =001;
        }
}
else if (WB_MEM == 1)
{
        if (is_Swap_MEM)
        {
                if (Reg_SRC_ADD1 == SRC_SWAP_MEM)
                        FWD_A = 110;
                else if (REG_SRC_ADD1 == REG_DEST_MEM)
                        FWD_A = 101;
        }
        else
        {
                if (Reg_SRC_ADD1 == REG_DEST_MEM)
                        FWD_A = 010;
        }
}
else
        FWD_A = 000;

//FWDB

if (WB_Ex == 1)
{
        if (is_Swap_EX)
        {
                if (Reg_SRC_ADD2 == SRC_SWAP_EX)
                        FWD_B = 100;
                else if (REG_SRC_ADD2 == REG_DEST_EX)
                        FWD_B =011 ;
        }
        else
        {
                if (Reg_SRC_ADD2 == REG_DEST_EX)
                        FWD_B =001;
        }
}
else if (WB_MEM == 1)
{
        if (is_Swap_MEM)
        {
                if (Reg_SRC_ADD2 == SRC_SWAP_MEM)
                        FWD_B = 110;
                else if (REG_SRC_ADD2 == REG_DEST_MEM)
                        FWD_B = 101;
        }
        else
        {
                if (Reg_SRC_ADD2 == REG_DEST_MEM)
                        FWD_B = 010;
        }
}
else
```

```
FWD_B = 000;
```

4. Interupt_Handler:
   Inputs: (3 signal, Bus of length 32 and bus of length 3)
   - INT
   - RST
   - RTI
   - PC_IN[31…0]
   - Flags_IN[2…0]

   Outputs(6 signals, Bus of length 3 and bus of length 32):
   - Ret_flags
   - Save_in_ST_INT
   - Save_Flags_INT
   - Is_OUT_PC_RST
   - Is_OUT_PC_INT
   - Flags_OUT_INT[2..0]
   - PC_OUT_INT[31..0]
   - Is_OUT_PC_RTI

Interrupt handler description:

Interrupt handler is a circular register with two pointers (one for the beginning of data one is the first empty location) of size (30) I.e. may handle up to 15 Nested Interrupts.

If the starting pointer equal the ending pointer it means it has nothing to do

Every code in schedule begins with 3 definition bits and ends with the value saved in stack (32 bit) of total of 35 bit

If it gets an interrupt it adds the following codes to the schedule. (And add 3 to the end pointer taking into consideration it can’t pass start pointer).

001 0’s

010 +0’s +Flags

011 + PC_to_be_saved

If it gets an RTI it adds the following codes to the schedule. (And add 2 to the end pointer taking into consideration it can’t pass start pointer).

100 + 0’s

101+0’s

If it gets a Reset it forces Is_OUT_PC_RST

Note that it takes interrupt signal from the ID/EX buffer not EX/MEM buffer.

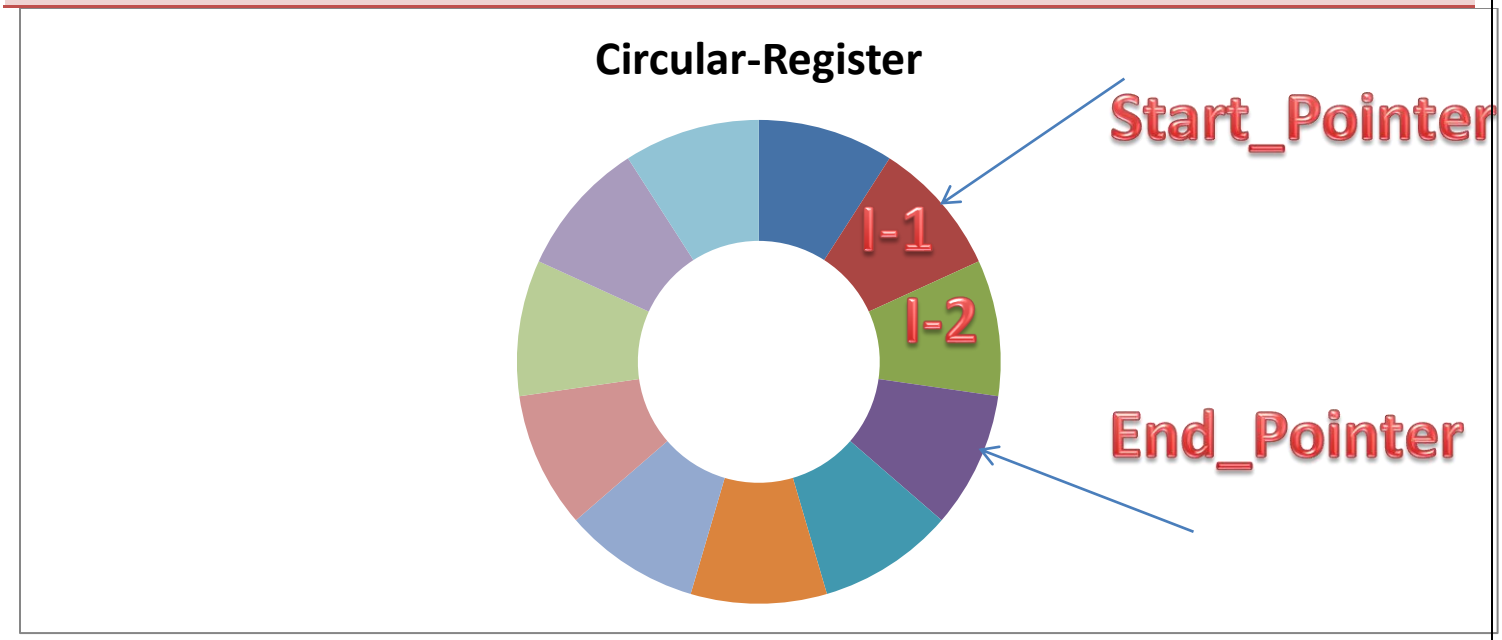
It reads interrupts, Resets and return, and handles the above on rising edge.

If it get nothing it change nothing.

And gives the signals of scheduled events on falling edge and advances start pointer (if the two pointers are equal nothing is done).

Output signals According to the schedule.

Code	Is_OUT_PC_INT	Save_Flags_INT	Save_in_ST_INT	Is_OUT_PC_RTI	Ret_flags	Flags_OUT_INT[2.0]	PC_OUT_INT[31.0]
001+0's	1	0	0	0	0	0	0
010+0's+Flags	0	1	1	0	0	Flags	0
011+PC	0	0	1	0	0	0	PC
100+0's	0	0	0	1	0	0	0
101+0's	0	0	0	0	0	1	0



Test Cases:

All test cases will be done in 3 steps (Unless there is no change):

- **With no Hazard Detection Unit, Flushing and Forwarding Unit. And then discuss how to solve them with NOP's**
- **With Forwarding Unit and then discuss how to solve other hazards using NOP's**
- **With Forwarding Unit and Hazard Detection Unit and then discuss how to solve other Hazards with NOP's**

All test cases also except (Memory Cache test cases) are done assuming perfect cache (we have tested all of them with full design but as far as we were told we don't have to document all of that).

In this report we didn't use reordering as a solution as we have referred to the hardware solutions that can be done in this processor not the software ones.

❖ **One Operand test cases:**

1.setC #C --> 1
2.NOP #No change
3.ClRC #C --> 0
4.NOT R1 #R1 =FFFFFFFF , C--> no change, N --> 1, Z --> 0
5.inc R1 #R1 =00000000 , C --> 1 , N --> 0 , Z --> 1
6.in R1 #R1= 5,add 5 on the in port, flags no change
7.in R2 #R2= 10,add 10 on the in port, flags no change
8.NOT R2 #R2= FFFFFFEF, C--> no change, N -->1,Z-->0
9.inc R1 #R1= 6, C --> 0, N -->0, Z-->0
10.Dec R2 #R2= FFFFFFEE,C-->1 , N-->1, Z-->0
11.out R1
12.out R2

N.B: Flag Updates is done after the Instruction enters the pipe by 4 clock cycles(**MEM** Stage).

N.B: Value is written on the output port after the Instruction enters the pipe by 4 clock cycles(**MEM** Stage).

- Without ForwardingUnit , HazardDetection or any Flushing

N.B: Flushing is disabled after Reset Signal as if there is no Flushing the Reset Signal won't be working correct and go to wrong location before going to the right one.

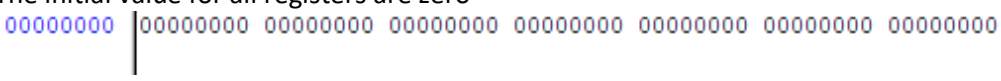
The processor in this run isn't working properly

Identifying Hazards

1.Data Hazard: RAW dependency Between Instr4 & Instr5 on R1
2.Data Hazard: RAW dependency Between Instr7 & Instr8 on R2
3.Data Hazard: RAW dependency Between Instr8 & Instr10 on R2
4.Data Hazard: RAW dependency Between Instr9 & Instr11 on R1
5.Data Hazard: RAW dependency Between Instr10 & Instr12 on R2

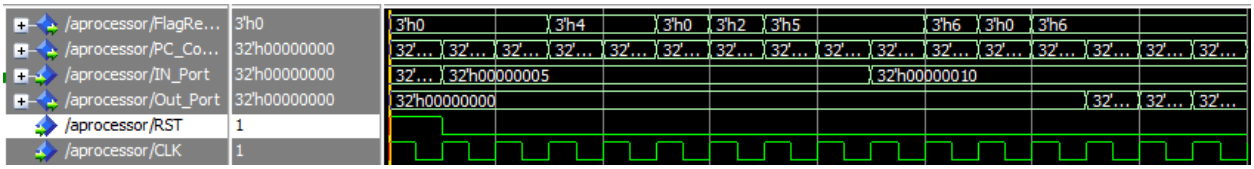
Registers' Values

The initial value for all registers are zero



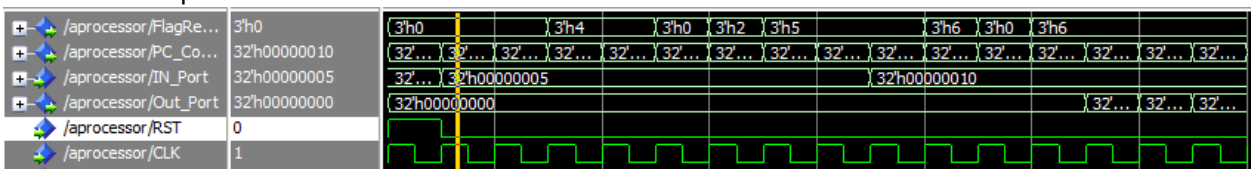
Clock cycle 1

Reset Signal is set to 1.



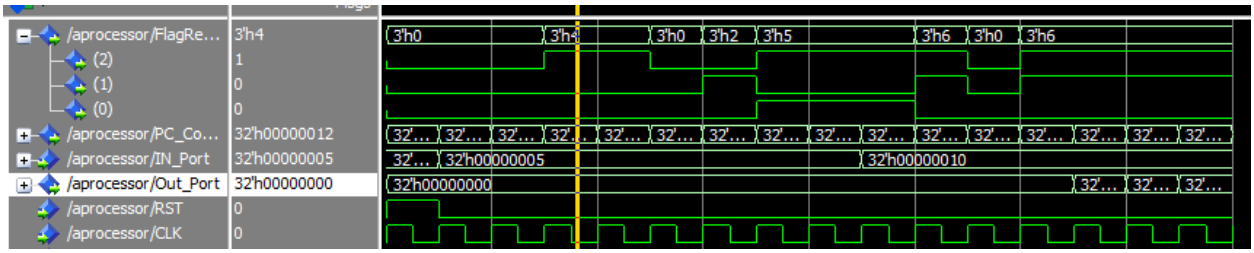
Clock cycle 2

PC counter is updated with the Reset Address.



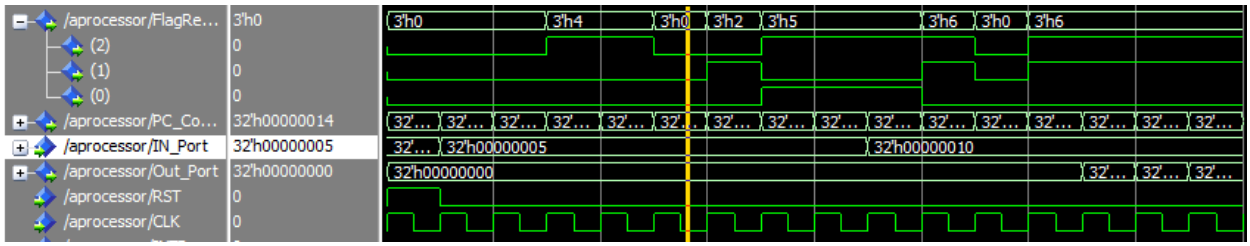
Clock cycle 4

The carry is set to 1.



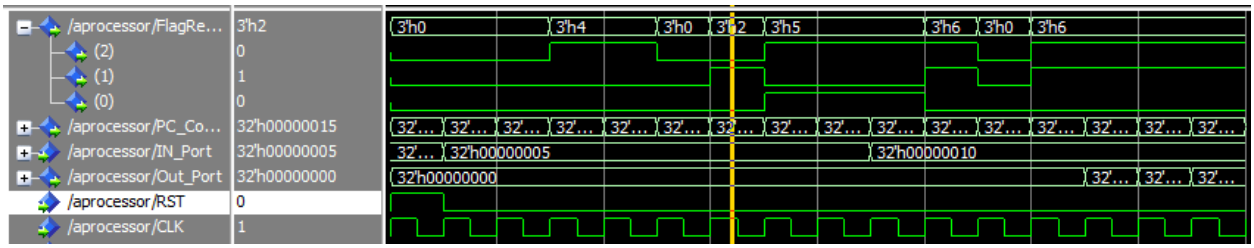
Clock cycle 6

The carry is set to 0.



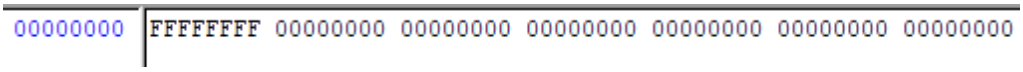
Clock cycle 7

Instr4 has updated the flags.

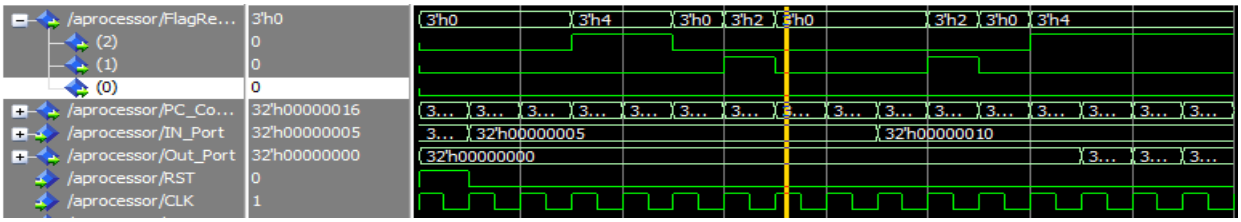


Clock cycle 8

Instr4 has written back the value to R1.



Instr5 has updated the flags wrong.

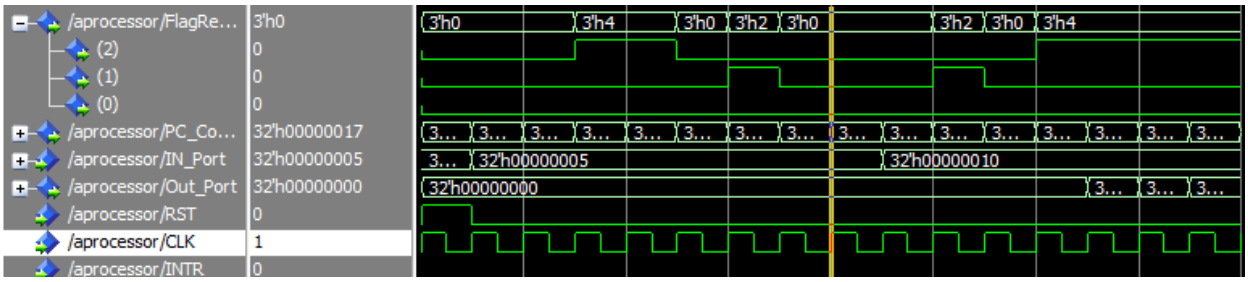


Clock cycle 9

Instr5 has written back the value to R1 but wrong as it have read x00000000 not xFFFFFFFF.

00000000 00000001 00000000 00000000 00000000 00000000 00000000 00000000

Instr6 has no flag updates.

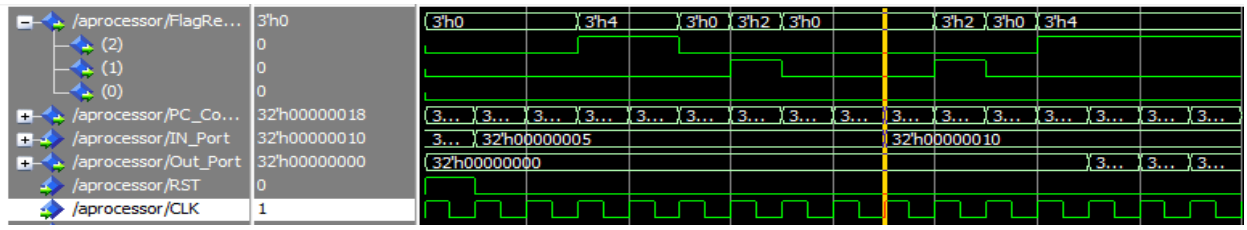


Clock cycle 10

Instr6 has written back the value to R1.

00000000 00000005 00000000 00000000 00000000 00000000 00000000 00000000

Instr7 has no flag updates.

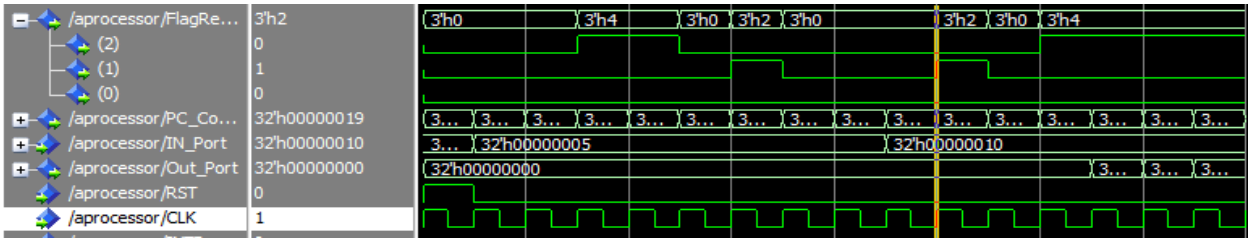


Clock cycle 11

Instr7 has written back the value to R2.

00000000 00000005 00000010 00000000 00000000 00000000 00000000 00000000

Instr8 has updated the flags wrong.

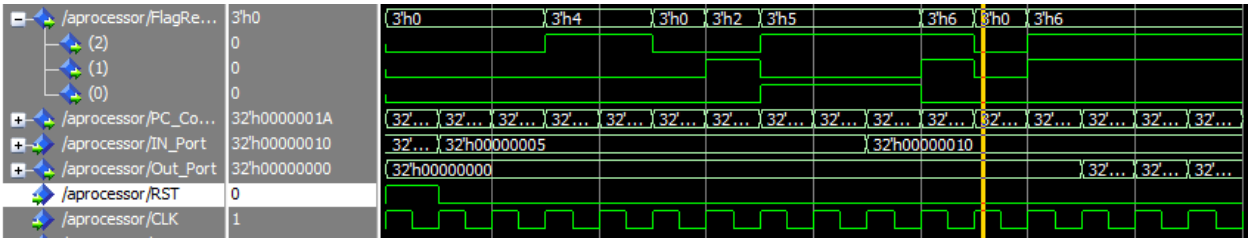


Clock cycle 12

Instr8 has written back the value to R2 but wrong as it have read x00000000 not x00000010 .

00000000 00000005 FFFFFFFF 00000000 00000000 00000000 00000000 00000000

Instr9 has updated the flags right this time.

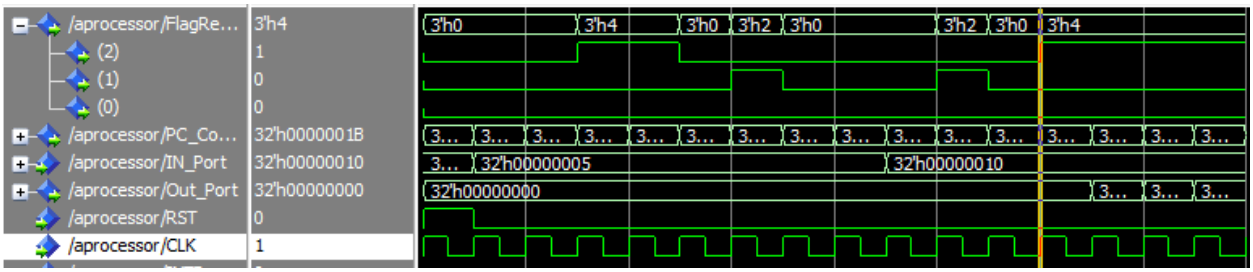


Clock cycle 13

Instr9 has written back the value to R1.

00000000 00000006 FFFFFFFF 00000000 00000000 00000000 00000000 00000000

Instr10 has updated the flags wrong.

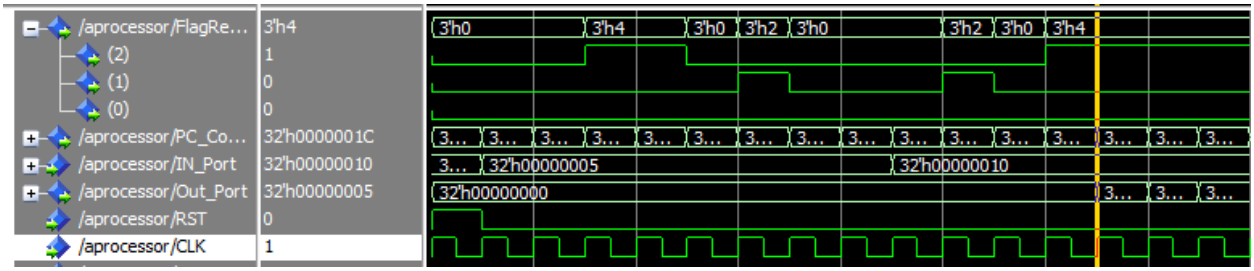


Clock cycle 14

Instr10 has written back the value to R2 but wrong as it has mistakenly read x00000010.

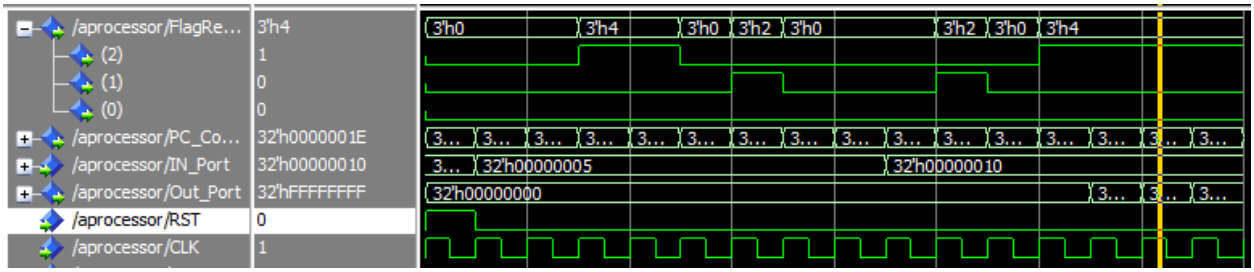
```
00000000 |00000006 0000000F 00000000 00000000 00000000 00000000 00000000
```

Instr11 updated out port by value of R1 but wrong as it have mistakenly read x00000005.



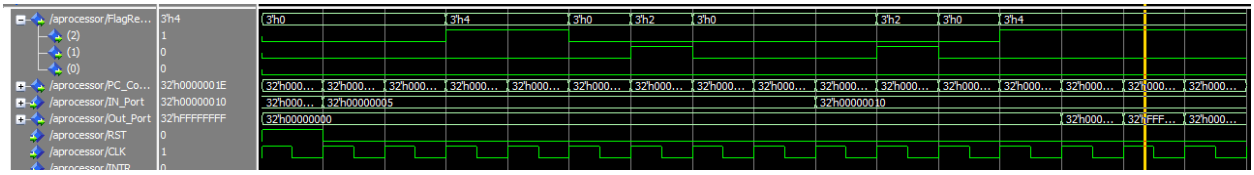
Clock cycle 15

Instr12 updated out port by value of R2 but wrong as it have mistakenly read xFFFFFFFF.



Clock cycle 16

The processor finished working but not correct.

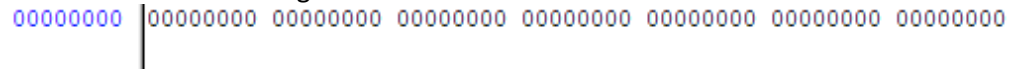


Solving Hazards by NOP and the processor will work properly but in 22 clock cycles

1.setC #C --> 1
2.NOP #No change
3.ClRC #C --> 0
4.NOT R1 #R1 =FFFFFFFF , C--> no change, N --> 1, Z --> 0
NOP
NOP
5.inc R1 #R1 =00000000 , C --> 1 , N --> 0 , Z --> 1
6.in R1 #R1= 5,add 5 on the in port, flags no change
7.in R2 #R2= 10,add 10 on the in port, flags no change
NOP
NOP
8.NOT R2 #R2= FFFFFFEF, C--> no change, N -->1,Z-->0
9.inc R1 #R1= 6, C --> 0, N -->0, Z-->0
NOP
10.Dec R2 #R2= FFFFFFEE,C-->1 , N-->1, Z-->0
11.out R1
NOP
12.out R2

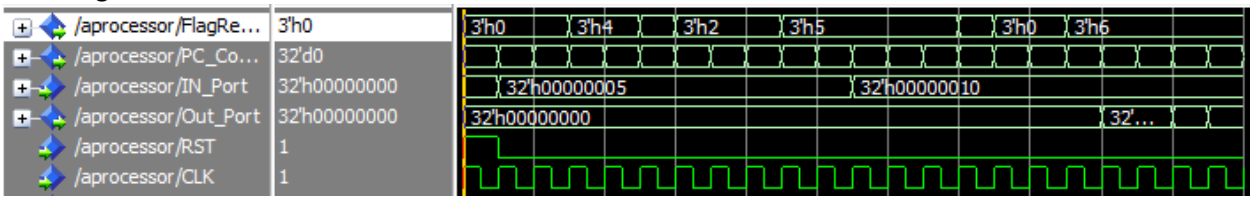
Registers' Values

The initial value for all registers are zero



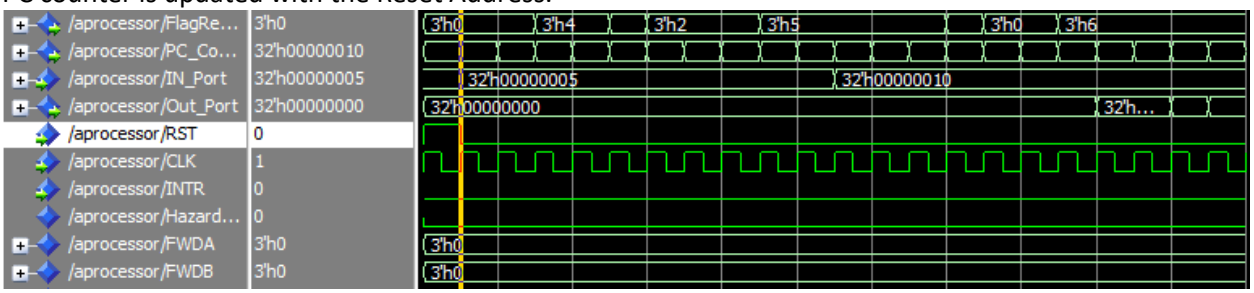
Clock cycle 1

Reset Signal is set to 1.



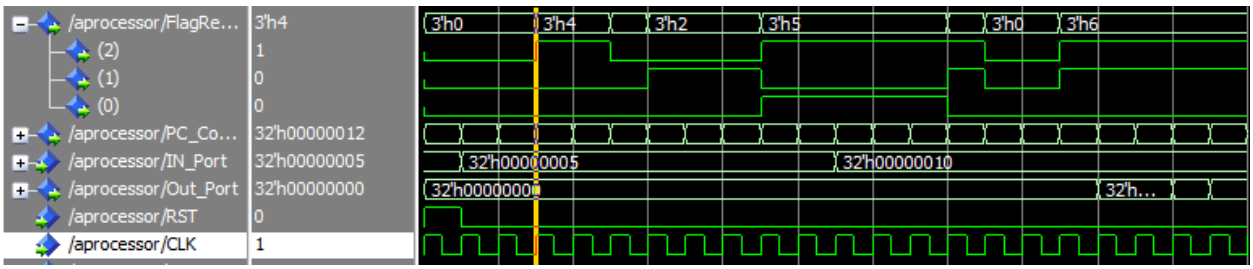
Clock cycle 2

PC counter is updated with the Reset Address.



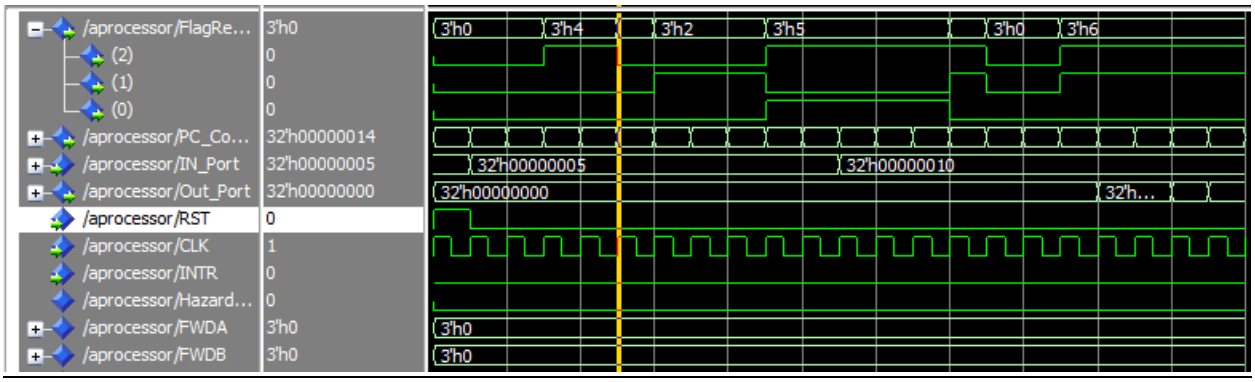
Clock cycle 4

The carry is set to 1.



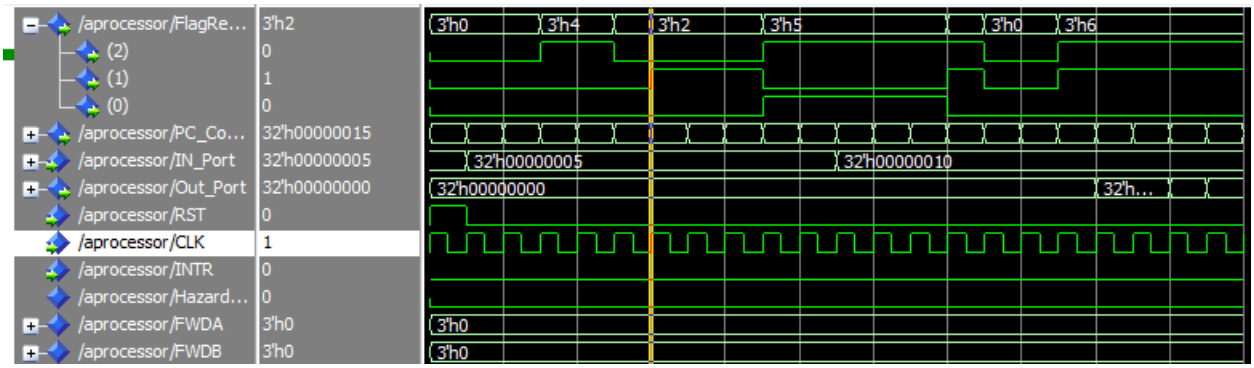
Clock cycle 6

The carry is set to 0.



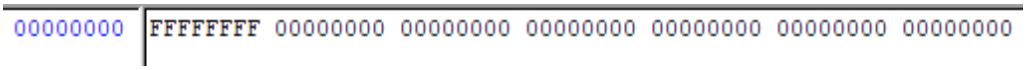
Clock cycle 7

Instr4 has updated the flags.



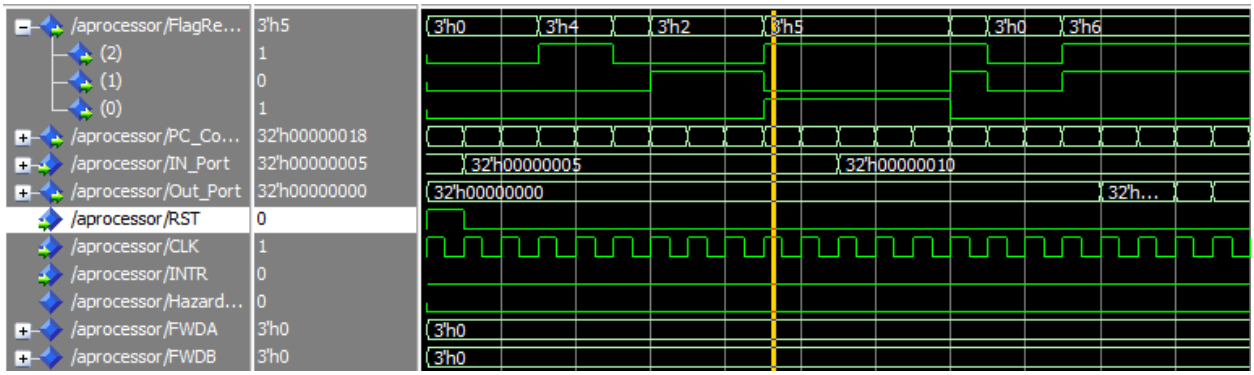
Clock cycle 8

Instr4 has written back the value to R1.



Clock cycle 10

Instr5 has updated the flags.



Clock cycle 11

Instr5 has written back the value to R1.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Clock cycle 12

Instr6 has written back the value to R1.

00000000	00000005	00000000	00000000	00000000	00000000	00000000	00000000

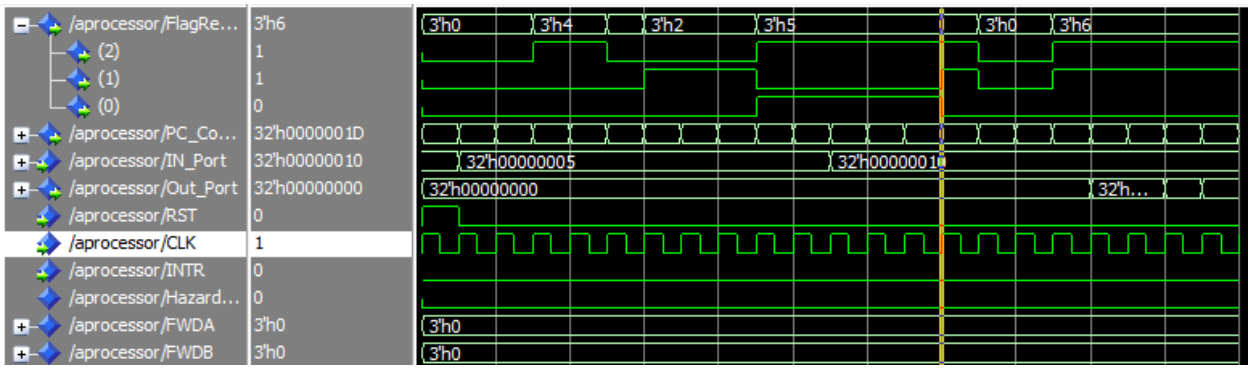
Clock cycle 13

Instr7 has written back the value to R2.

00000000	00000005	00000010	00000000	00000000	00000000	00000000	00000000

Clock cycle 15

Instr8 has updated the flags.

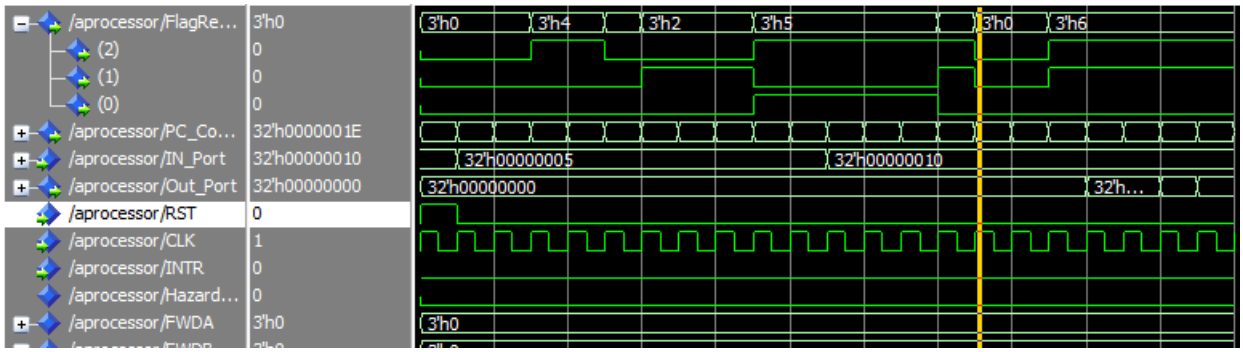


Clock cycle 16

Instr8 has written back the value to R2.

00000000	00000005	FFFFFFEF	00000000	00000000	00000000	00000000	00000000

Instr9 has updated the flags.



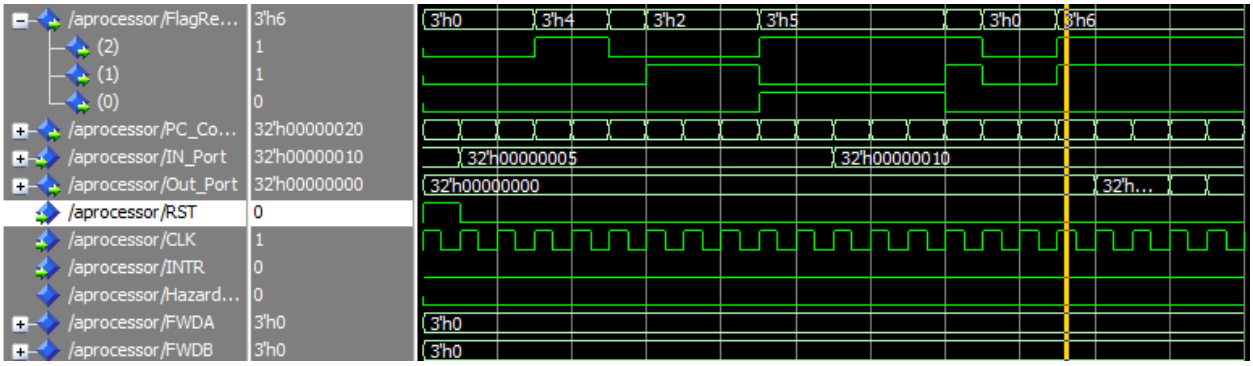
Clock cycle 17

Instr9 has written back the value to R1.

00000000	00000006	FFFFFFEE	00000000	00000000	00000000	00000000	00000000

Clock cycle 18

Instr10 has updated the flags.

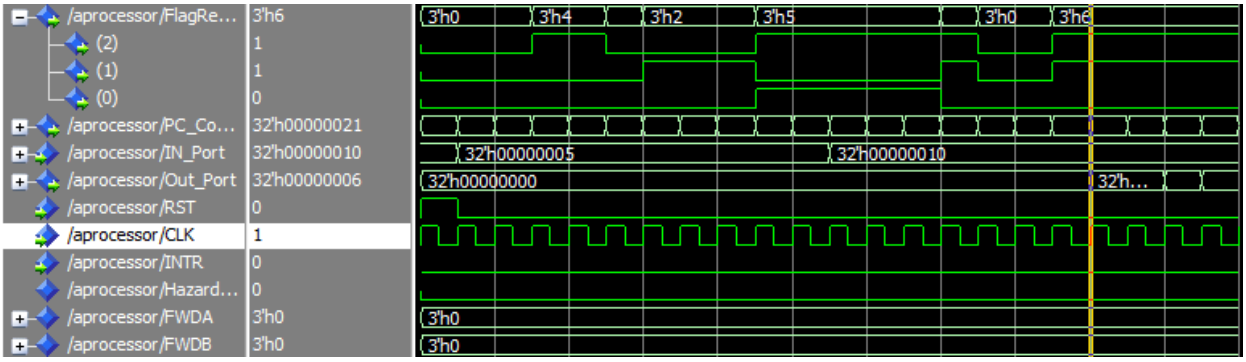


Clock cycle 19

Instr10 has written back the value to R2.

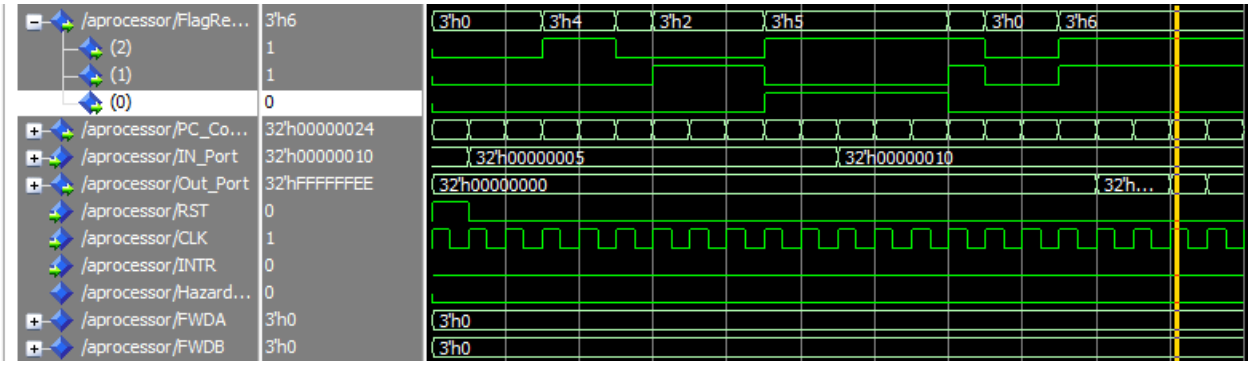
```
00000000 00000006 FFFFFFEE 00000000 00000000 00000000 00000000 00000000
```

Instr11 updated out port by value of R1.



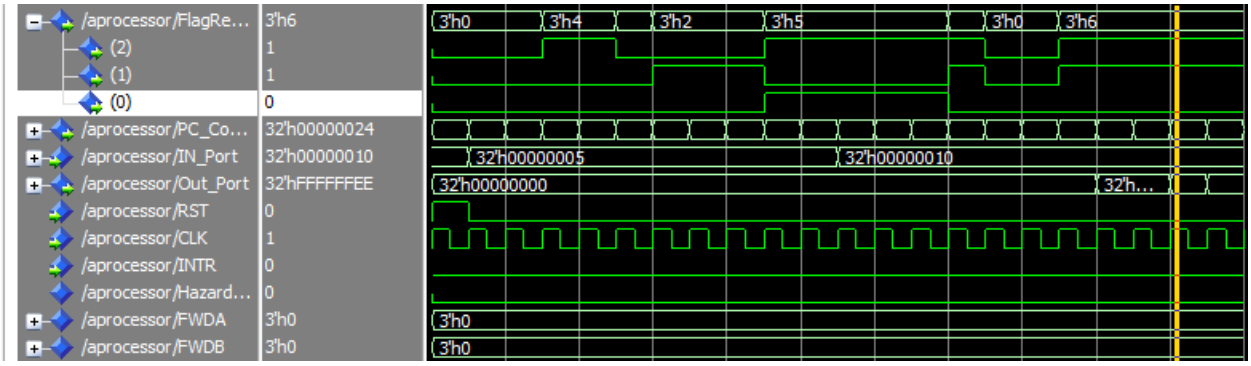
Clock cycle 21

Instr12 updated out port by value of R2.



Clock cycle 22

The processor finished working correctly.



The Incremental Run for the following cases gives the same output which is correct and reflects that the processor is working correctly and properly as the above hazards are solved by using ForwardingUnit so no need for adding NOP after adding HazardDetection or Flushing as there are no hazards to solve
N.B: The Snapshots are the same for the following 3 cases

The processor will finish working in 16 cycles (K+n-1=5+12-1=16) and as there is no load use case or branch prediction in the above code.

- Without HazardDetection or any Flushing

  ForwardingUnit solved all Hazards

- Without any Flushing

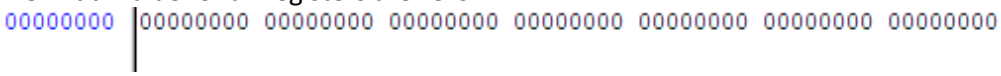
  ForwardingUnit solved all Hazards

- Full Working Processor

  ForwardingUnit solved all Hazards

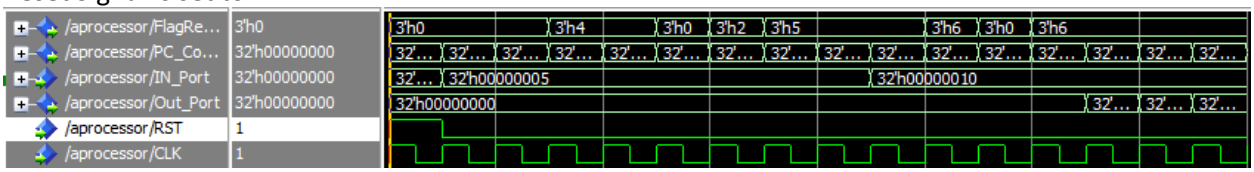
Registers' Values

The initial value for all registers are zero



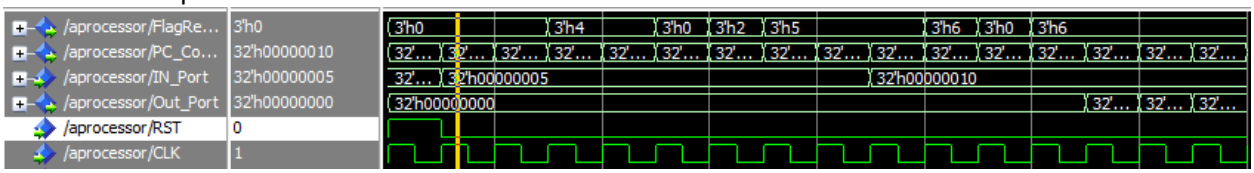
Clock cycle 1

Reset Signal is set to 1.



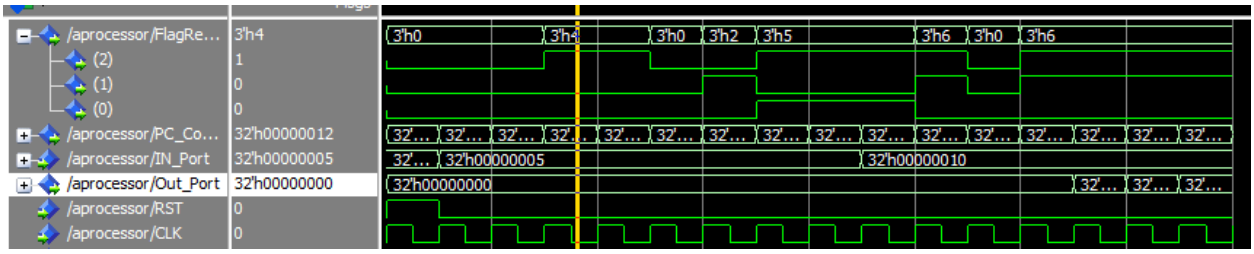
Clock cycle 2

PC counter is updated with the Reset Address.



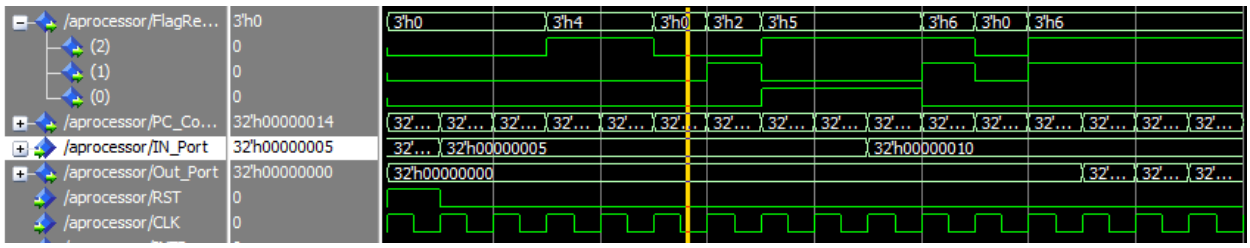
Clock cycle 4

The carry is set to 1.



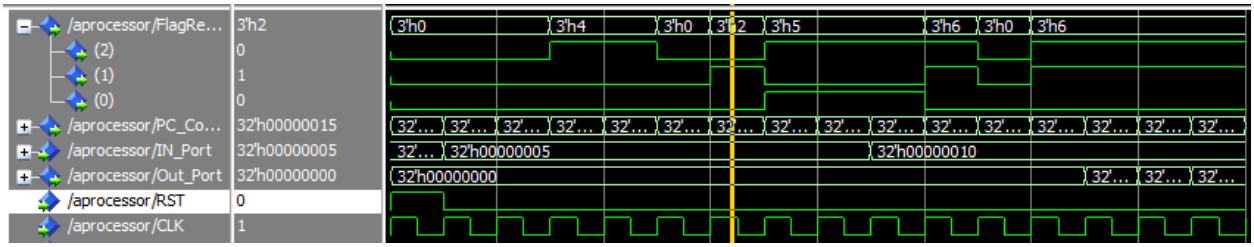
Clock cycle 6

The carry is set to 0.



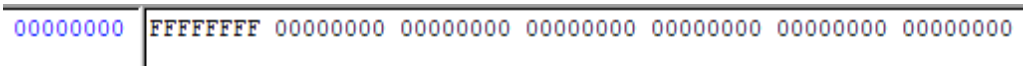
Clock cycle 7

Instr4 has updated the flags.

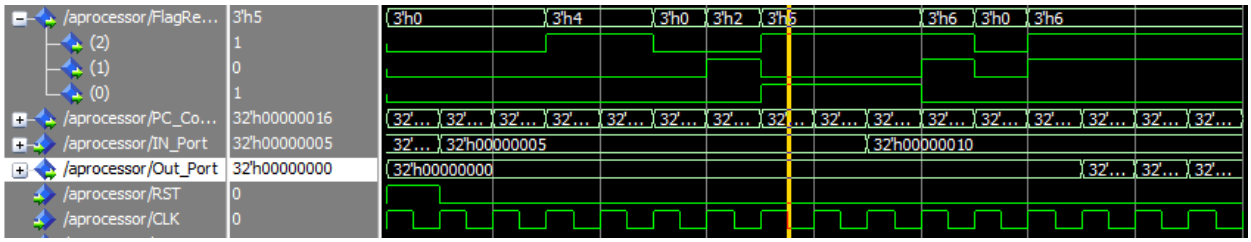


Clock cycle 8

Instr4 has written back the value to R1.

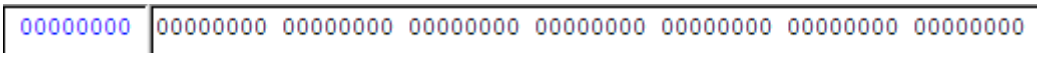


Instr5 has updated the flags.

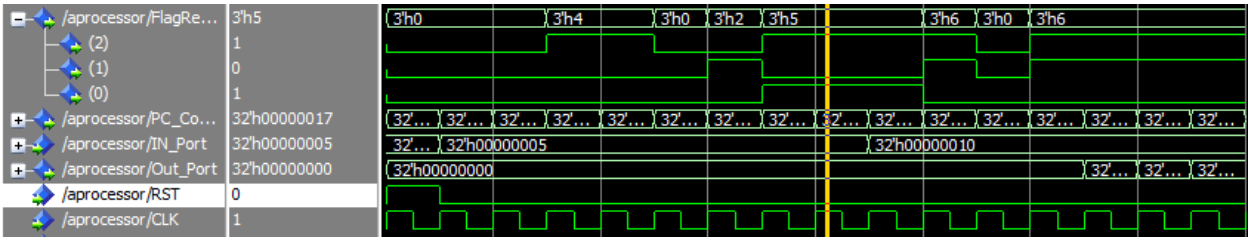


Clock cycle 9

Instr5 has written back the value to R1.

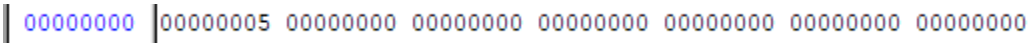


Instr6 has no flag updates.

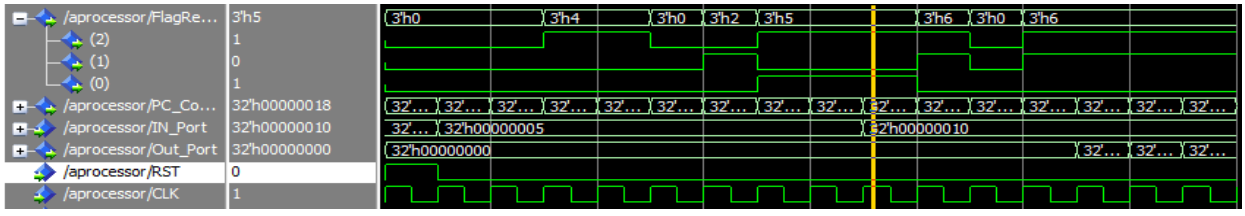


Clock cycle 10

Instr6 has written back the value to R1.



Instr7 has no flag updates.

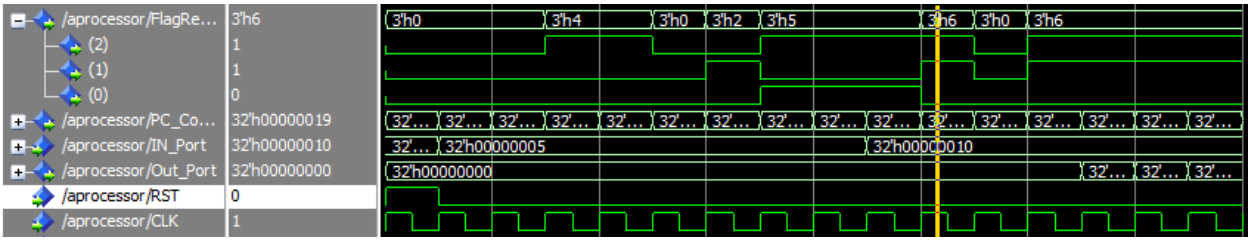


Clock cycle 11

Instr7 has written back the value to R2.

```
00000000 |00000005 00000010 00000000 00000000 00000000 00000000 00000000
```

Instr8 has updated the flags.

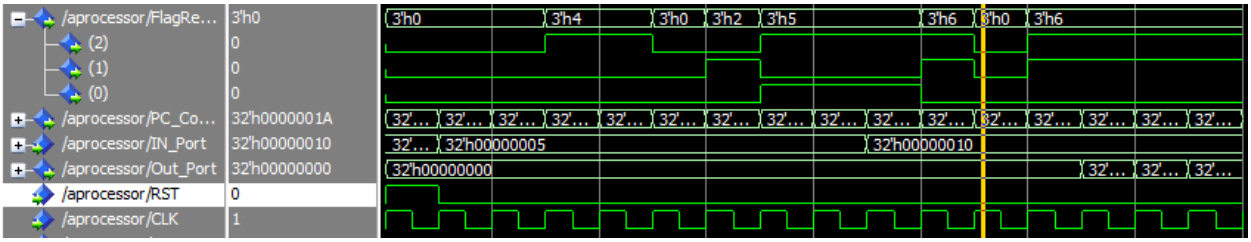


Clock cycle 12

Instr8 has written back the value to R2.

```
00000000 |00000005 FFFFFFEF 00000000 00000000 00000000 00000000 00000000
```

Instr9 has updated the flags.

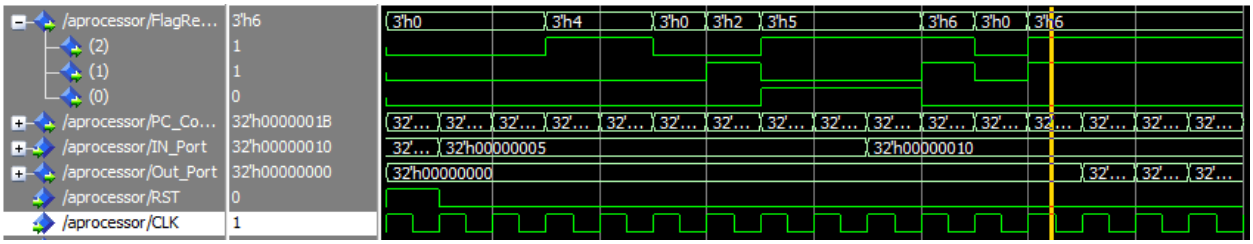


Clock cycle 13

Instr9 has written back the value to R1.

```
00000000 |00000006 FFFFFFEE 00000000 00000000 00000000 00000000 00000000
```

Instr10 has updated the flags.

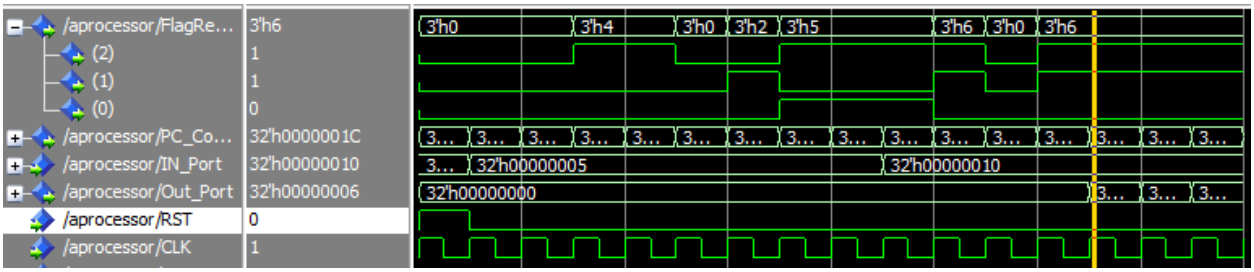


Clock cycle 14

Instr10 has written back the value to R2.

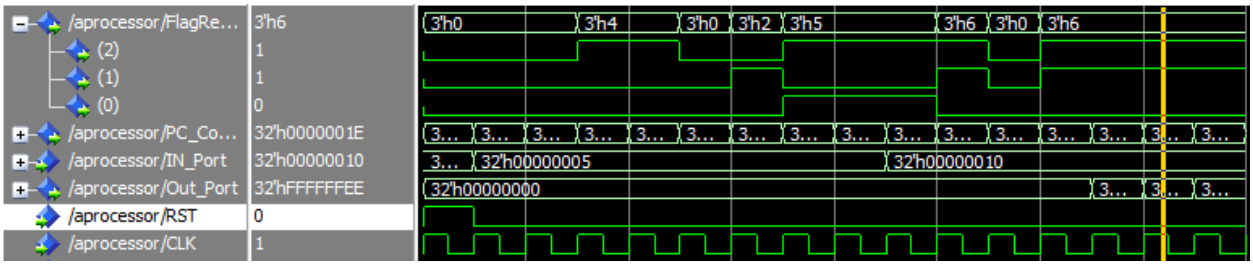
```
00000000 |00000006 FFFFFFEE 00000000 00000000 00000000 00000000 00000000
```

Instr11 updated out port by value of R1.



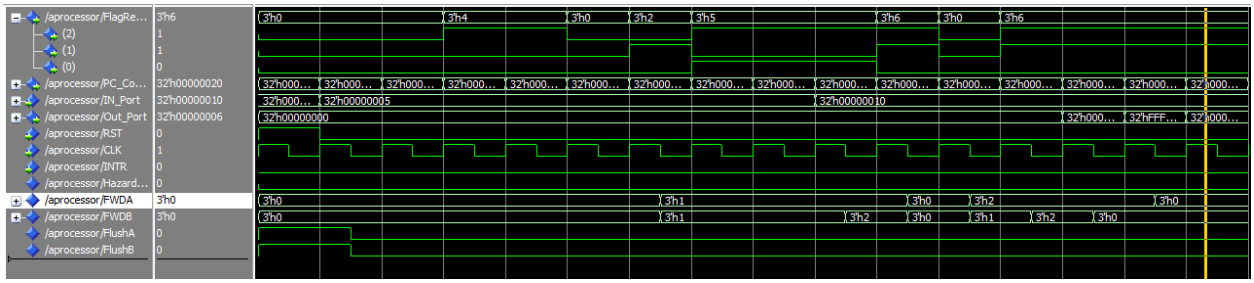
Clock cycle 15

Instr12 updated out port by value of R2.



Clock cycle 16

The processor finished working correctly.



❖ **Two Operand test cases:**

1.in R1 #add 5 in R1

2.in R2 #add 19 in R2

3.in R3 #FFFD

4.in R4 #F320

5.IADD R3,R5,2 #R5 = FFFF , flags no change

6.ADD R1,R4,R4 #R4= F325 , C-->0, N-->0, Z-->0

7.SUB R5,R4,R6 #R6= 0CDA , C-->1, N-->0,Z-->0

8.AND R7,R6,R6 #R6= 00000000 , C-->no change, N-->0, Z-->1

9.OR R2,R1,R1 #R1=1D , C--> no change, N-->0, Z--> 0

10.SHL R2,2 #R2=64 , C--> 0, N -->0 , Z -->0

11.SHR R2,3 #R2=0C , C -->1, N-->0 , Z-->0

12.SWAP R2,R5 #R5=0C ,R2=FFFF ,no change for flags

13.ADD R5,R2,R2 #R2= 1000B (C,N,Z= 0)

N.B: Flag Updates is done after the Instruction enters the pipe by 4 clock cycles(**MEM** Stage).

- Without ForwardingUnit , HazardDetection or any Flushing

N.B: Flushing is disabled after Reset Signal as if there is no Flushing the Reset Signal won't be working correct and go to wrong location before going to the right one.

The processor in this run isn't working properly

Identifying Hazards

1.Data Hazard: RAW dependency Between Instr3 & Instr5 on R3

2.Data Hazard: RAW dependency Between Instr4 & Instr6 on R4

3.Data Hazard: RAW dependency Between Instr6 & Instr7 on R4

4.Data Hazard: RAW dependency Between Instr7 & Instr8 on R6

5.Data Hazard: RAW dependency Between Instr10 & Instr11 on R2

6.Data Hazard: RAW dependency Between Instr11 & Instr12 on R2

7.Data Hazard: RAW dependency Between Instr12 & Instr13 on R2 & R5

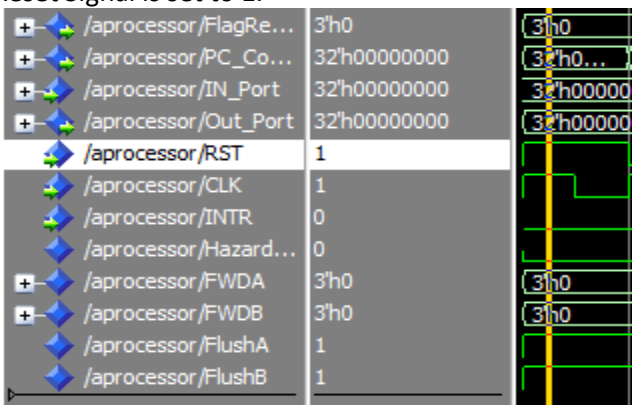
Registers' Values

The initial value for all registers are zero

```
00000000 |00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

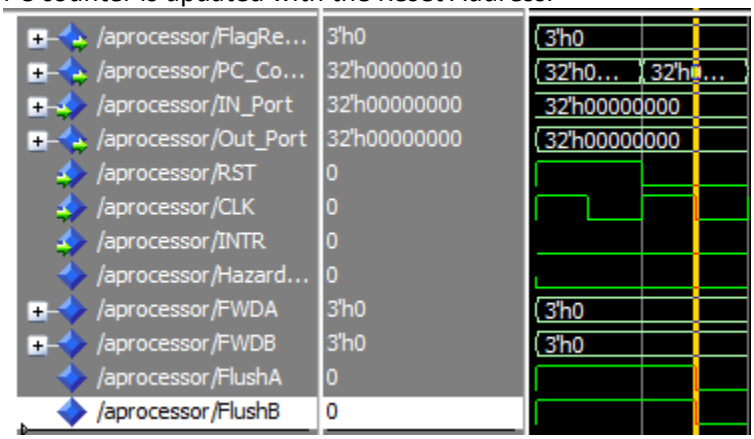
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

Instr1 has written back the value to R1.

```
00000000 |00000005 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 6

Instr2 has written back the value to R2.

```
00000000 |00000005 00000019 00000000 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 7

Instr3 has written back the value to R3.

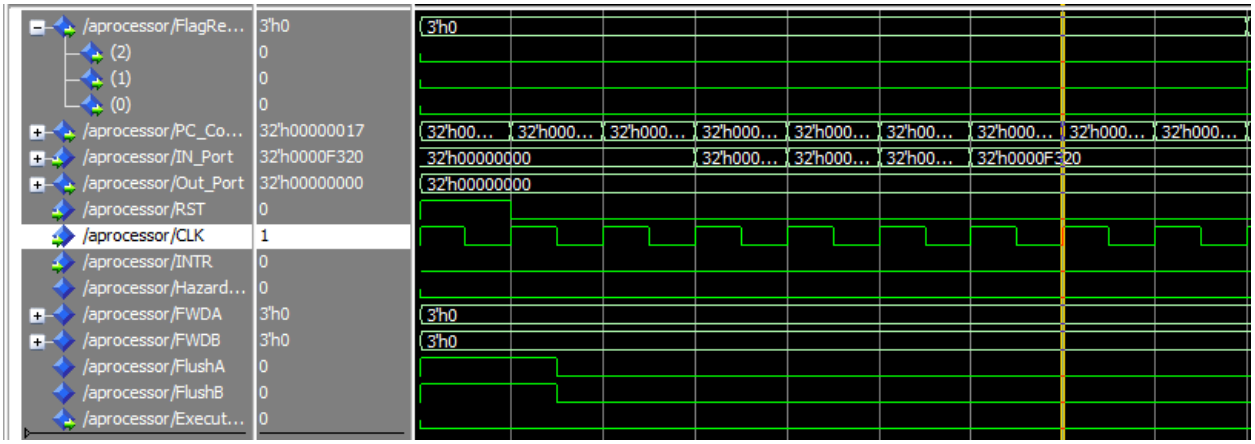
```
00000000 |00000005 00000019 0000FFFD 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 8

Instr4 has written back the value to R4.

```
00000000 |00000005 00000019 0000FFFD 0000F320 00000000 00000000 00000000
```

Instr5 has no flag updates .

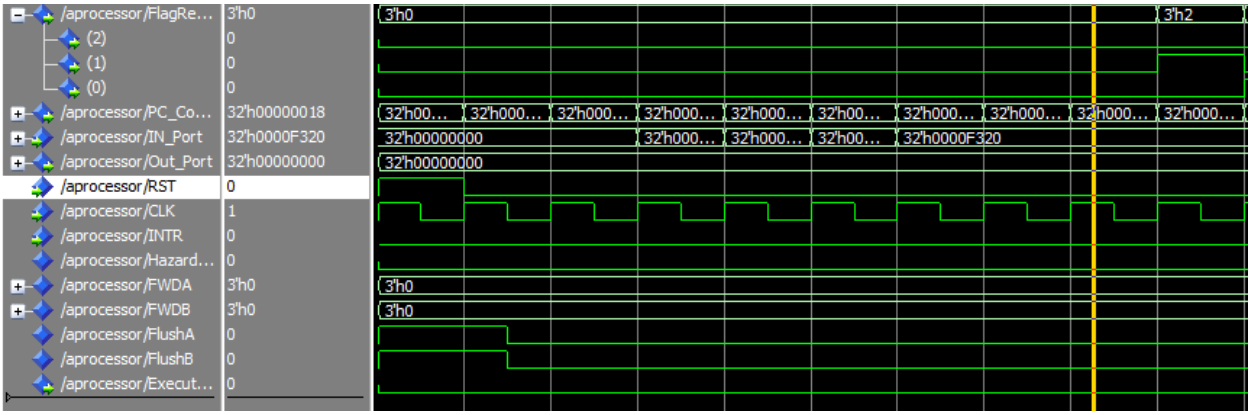


Clock cycle 9

Instr5 has written back the value to R5 but wrong as it has mistakenly read x000000000.

```
00000000 |00000005 00000019 0000FFFD 0000F320 00000002 00000000 00000000
```

Instr6 has updated the flags but correct by luck.

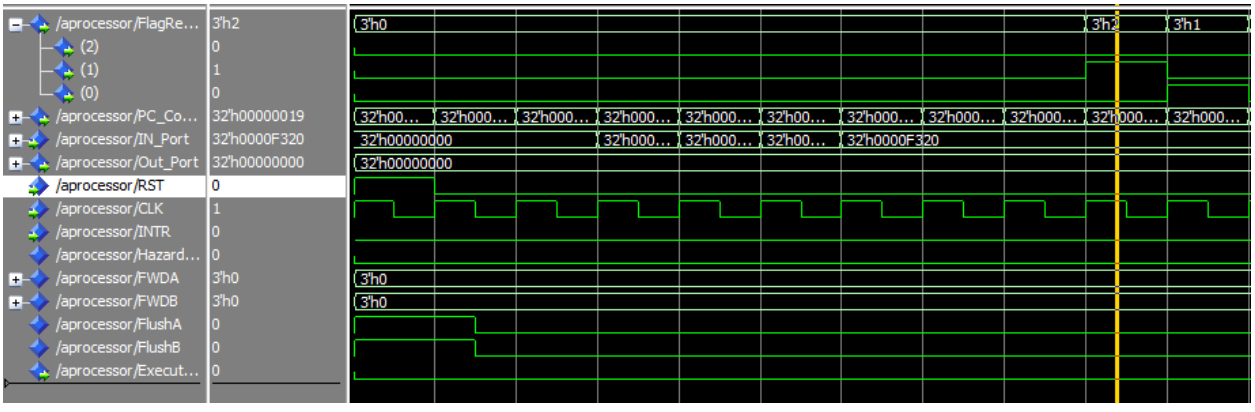


Clock cycle 10

Instr6 has written back the value to R4 but wrong as it has mistakenly read x000000000 .

```
00000000 |00000005 00000019 0000FFFD 00000005 00000002 00000000 00000000
```

Instr7 has updated the flags wrong.

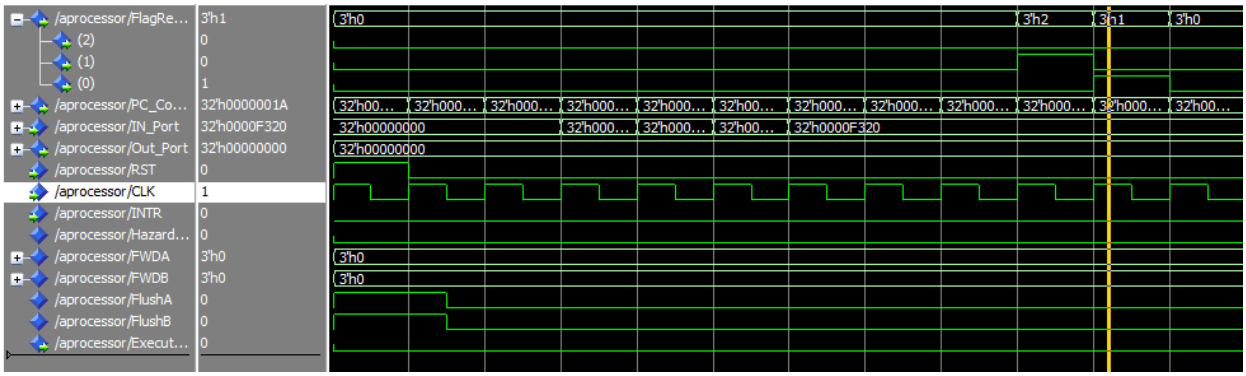


Clock cycle 11

Instr7 has written back the value to R6 but wrong.

```
00000000 00000005 00000019 0000FFFD 00000005 00000002 FFFF0CE0 00000000
```

Instr8 has updated the flags wrong.

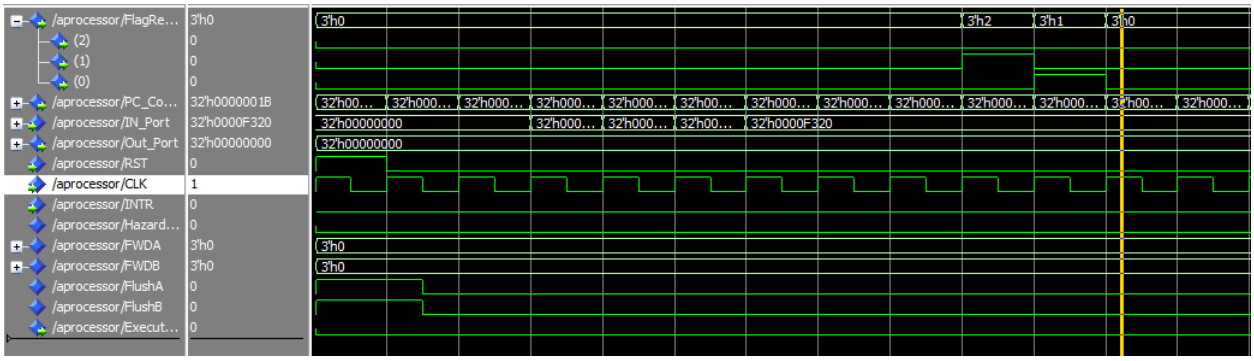


Clock cycle 12

Instr8 has written back the value to R6 but correct as it ANDS with zero.

```
00000000 00000005 00000019 0000FFFD 00000005 00000002 00000000 00000000
```

Instr9 has updated the flags wrong.

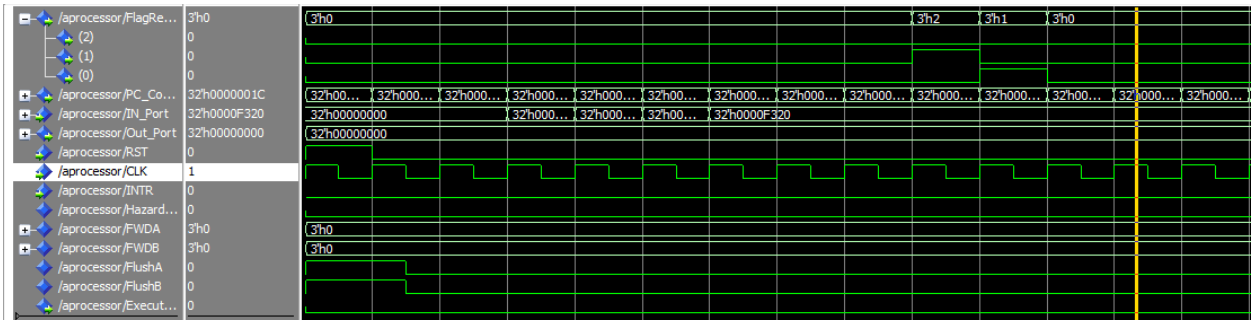


Clock cycle 13

Instr9 has written back the value to R1 as it read the correct values.

```
00000000 0000001D 00000019 0000FFFD 00000005 00000002 00000000 00000000
```

Instr10 has updated the flags correct as it read the right value.

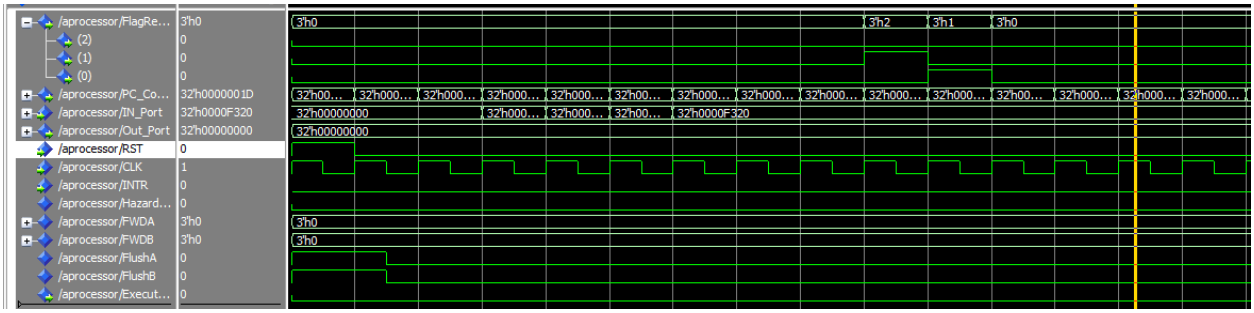


Clock cycle 14

Instr10 has written back the value to R2 as it read the correct value.

```
00000000 |0000001D 00000064 0000FFFD 00000005 00000002 00000000 00000000
```

Instr11 has updated the flags wrong.

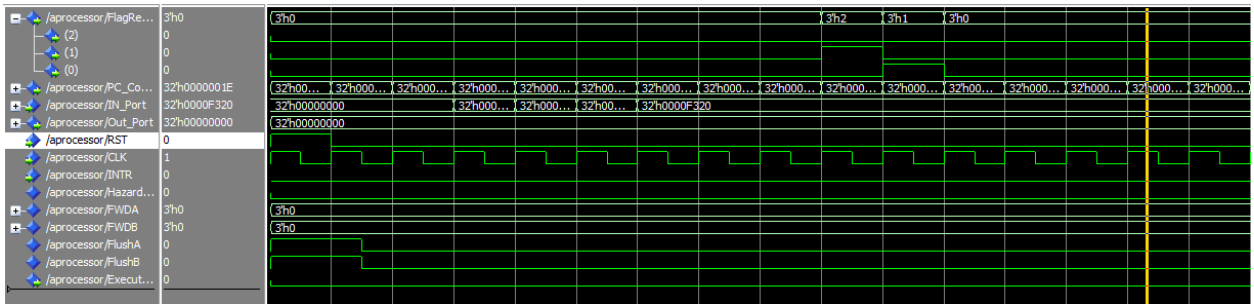


Clock cycle 15

Instr11 has written back the value to R2 but wrong as it read the old value x19.

```
00000000 |0000001D 00000003 0000FFFD 00000005 00000002 00000000 00000000
```

Instr12 has no flag updates.

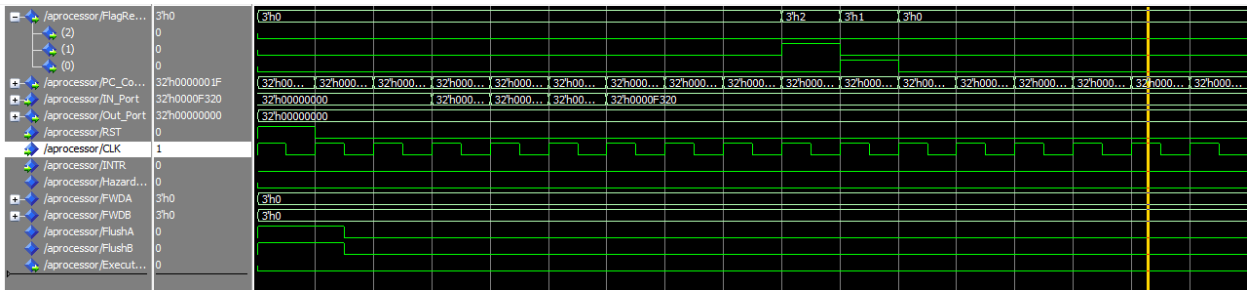


Clock cycle 16

Instr12 has written back the value to R2 & R5 but the old wrong values.

```
00000000 |0000001D 00000002 0000FFFD 00000005 00000019 00000000 00000000
```

Instr13 has updated the flags but correct by luck.



Clock cycle 17

Instr13 has written back the value to R2 but wrong value.

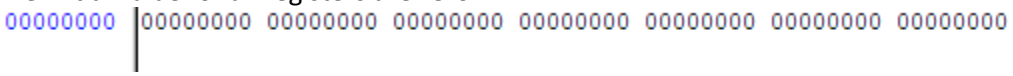
```
00000000 |0000001D 00000066 0000FFFD 00000005 00000019 00000000 00000000
```

Solving Hazards by NOP and the processor will work properly but in 29 clock cycles

1.in R1 #add 5 in R1
2.in R2 #add 19 in R2
3.in R3 #FFFD
4.in R4 #F320
NOP
5.IADD R3,R5,2 #R5 = FFFF , flags no change
NOP
6.ADD R1,R4,R4 #R4= F325 , C-->0, N-->0, Z-->0
NOP
NOP
7.SUB R5,R4,R6 #R6= 0CDA , C-->1, N-->0,Z-->0
NOP
NOP
8.AND R7,R6,R6 #R6= 00000000 , C-->no change, N-->0, Z-->1
9.OR R2,R1,R1 #R1=1D , C--> no change, N-->0, Z--> 0
10.SHL R2,2 #R2=64 , C--> 0, N -->0 , Z -->0
NOP
NOP
11.SHR R2,3 #R2=0C , C -->1, N-->0 , Z-->0
NOP
NOP
12.SWAP R2,R5 #R5=0C ,R2=FFFF ,no change for flags
NOP
NOP
13.ADD R5,R2,R2 #R2= 1000B (C,N,Z= 0)

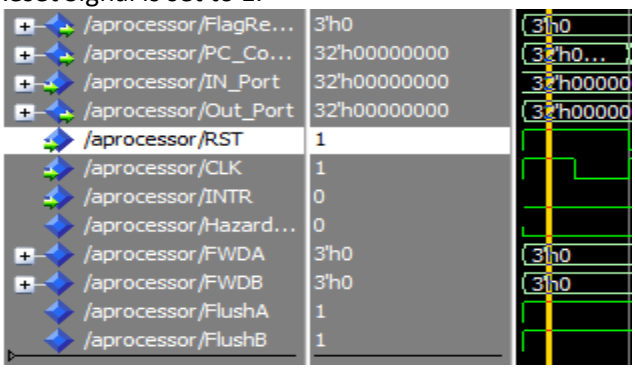
Registers' Values

The initial value for all registers are zero



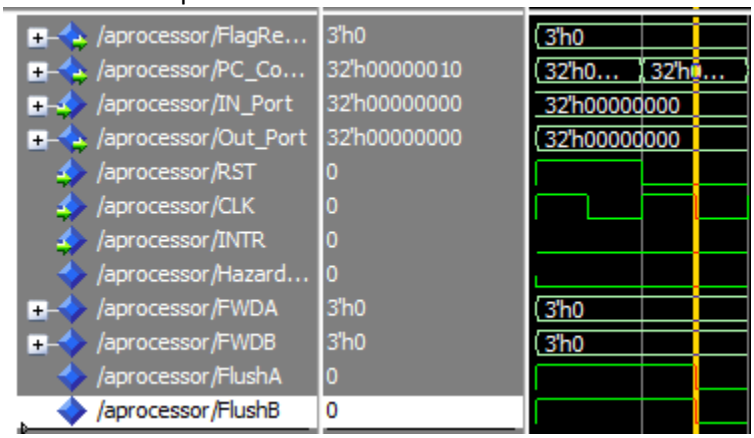
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

Instr1 has written back the value to R1.

```
00000000 |00000005 00000000 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 6

Instr2 has written back the value to R2.

```
00000000 |00000005 00000019 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 7

Instr3 has written back the value to R3.

```
00000000 |00000005 00000019 0000FFFD 00000000 00000000 00000000 00000000
```

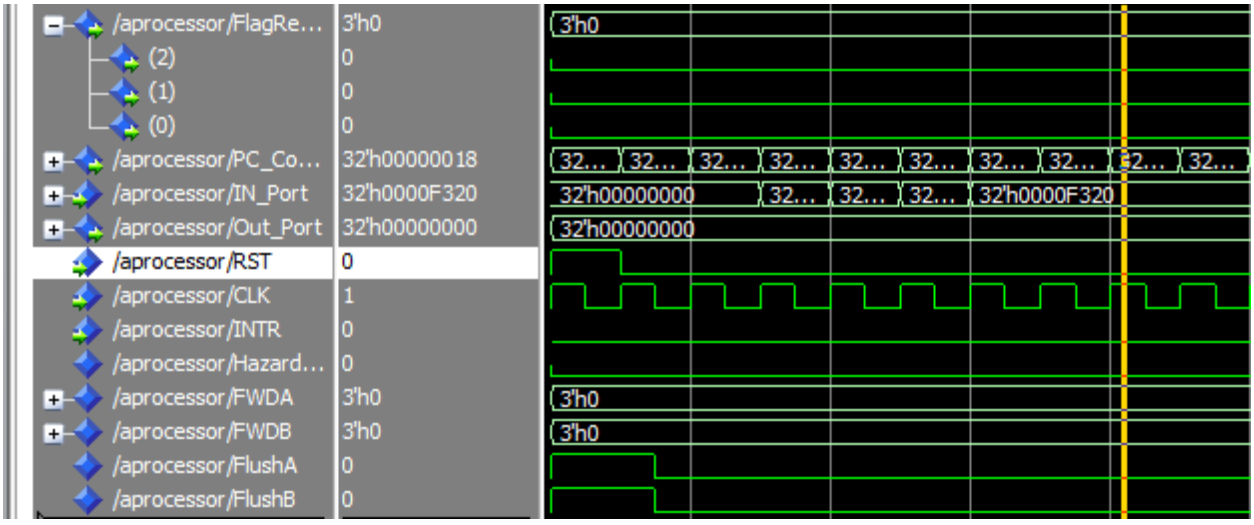
Clock cycle 8

Instr4 has written back the value to R4.

```
00000000 |00000005 00000019 0000FFFD 0000F320 00000000 00000000 00000000
```

Clock cycle 9

Instr5 has no flag updates.



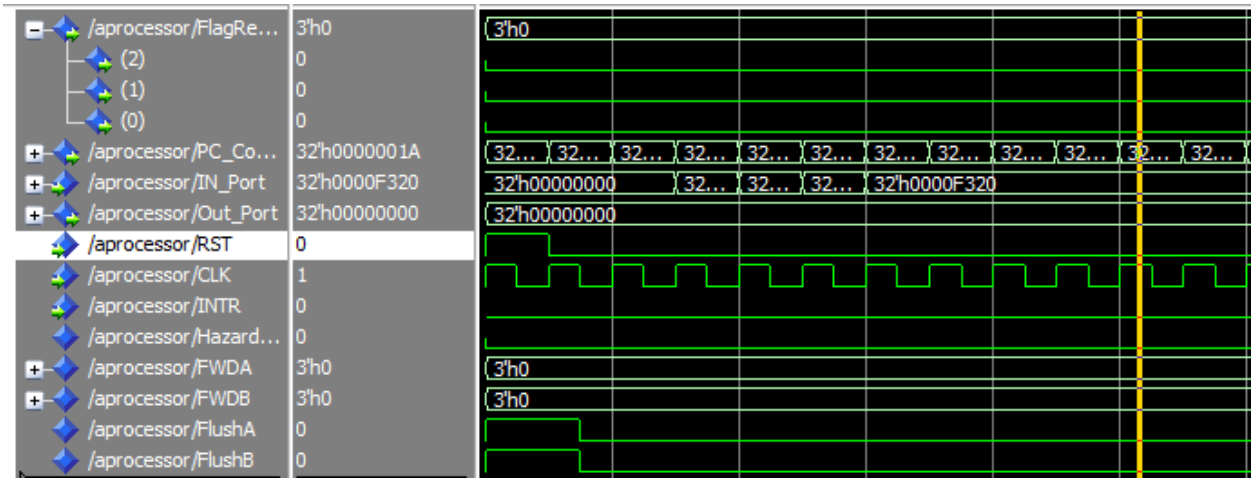
Clock cycle 10

Instr5 has written back the value to R5.

```
00000000 |00000005 00000019 0000FFFD 0000F320 0000FFFF 00000000 00000000
```

Clock cycle 11

Instr6 has no flag updates.



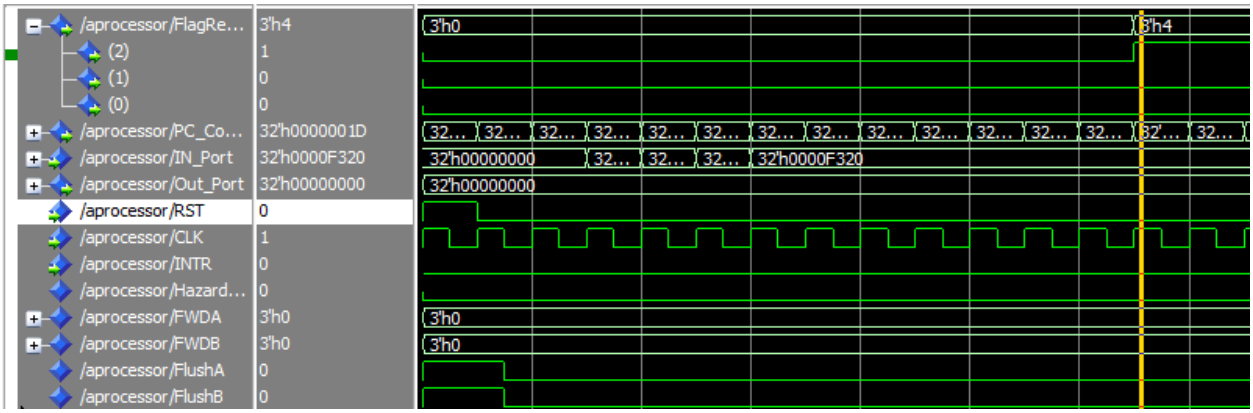
Clock cycle 12

Instr6 has written back the value to R4.

```
00000000 |00000005 00000019 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Clock cycle 14

Instr7 has updated the flags.



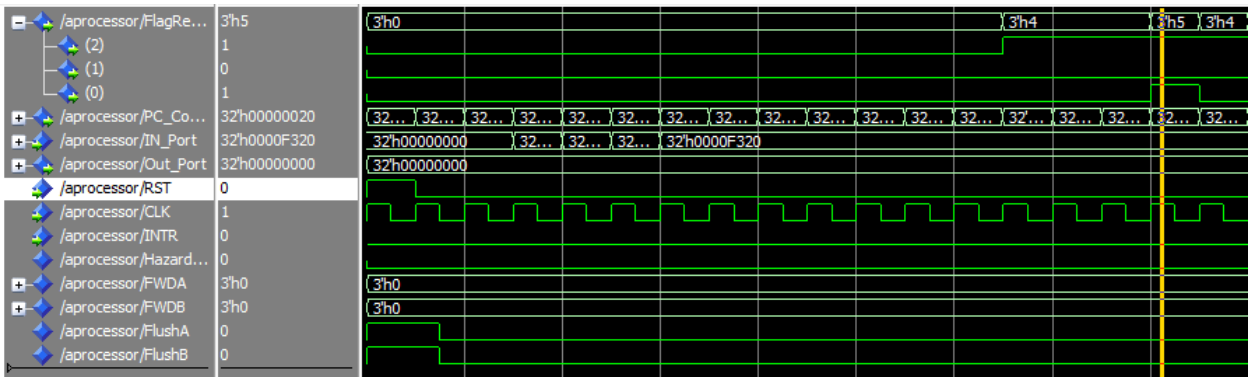
Clock cycle 15

Instr7 has written back the value to R6.

```
00000000 |00000005 00000019 0000FFFD 0000F325 0000FFFF 00000CDA 00000000
```

Clock cycle 17

Instr8 has updated the flags.

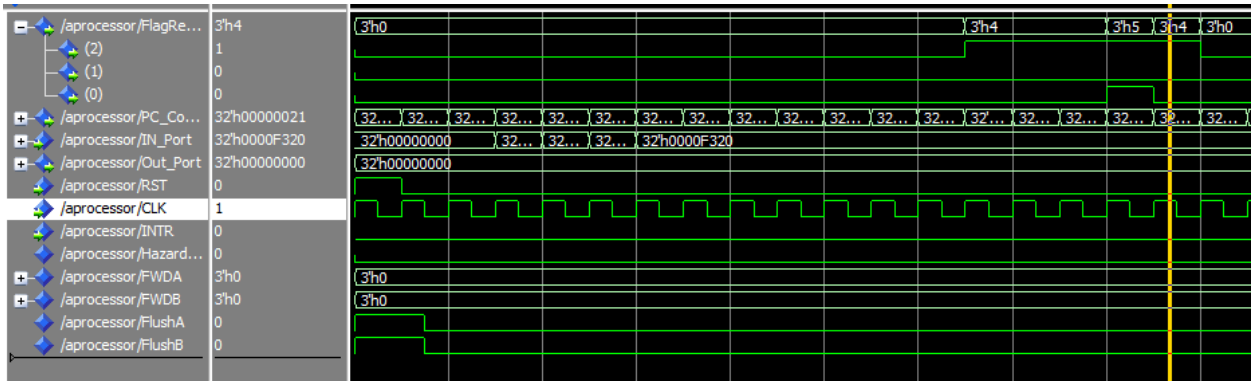


Clock cycle 18

Instr8 has written back the value to R6.

```
00000000 |00000005 00000019 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Instr9 has updated the flags.

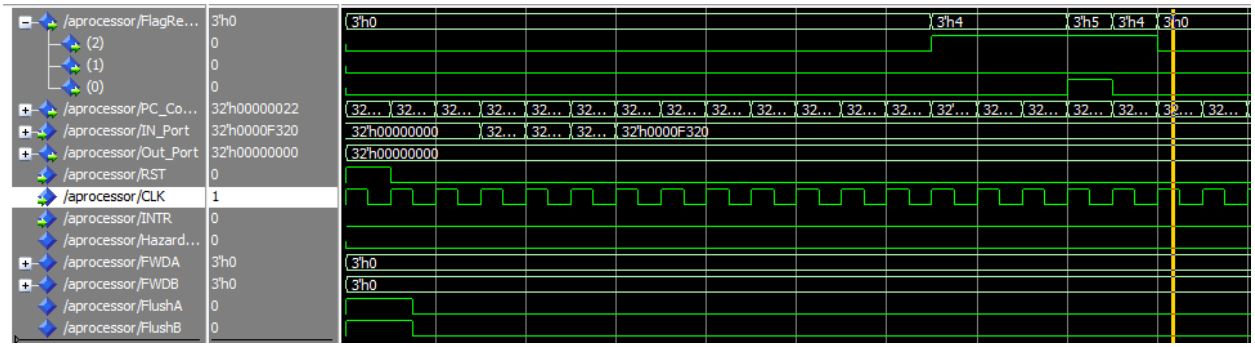


Clock cycle 19

Instr9 has written back the value to R1.

```
00000000 |0000001D 00000019 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Instr10 has updated the flags.



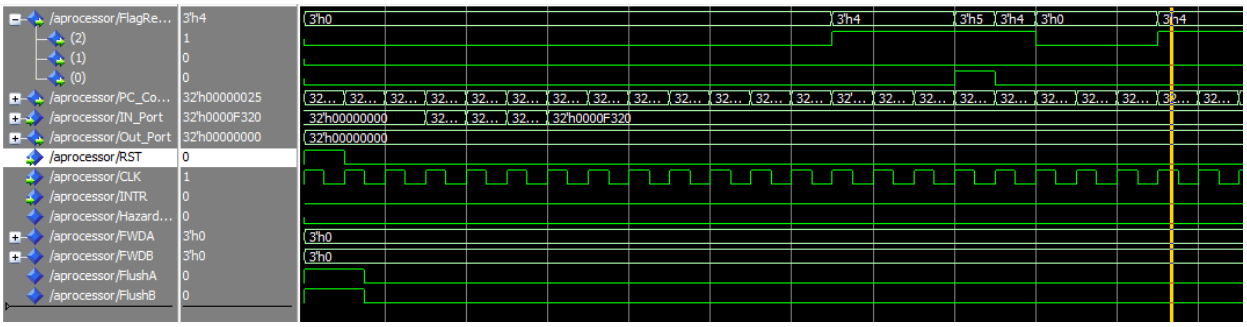
Clock cycle 20

Instr10 has written back the value to R2.

```
00000000 |0000001D 00000064 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Clock cycle 22

Instr11 has updated the flags.



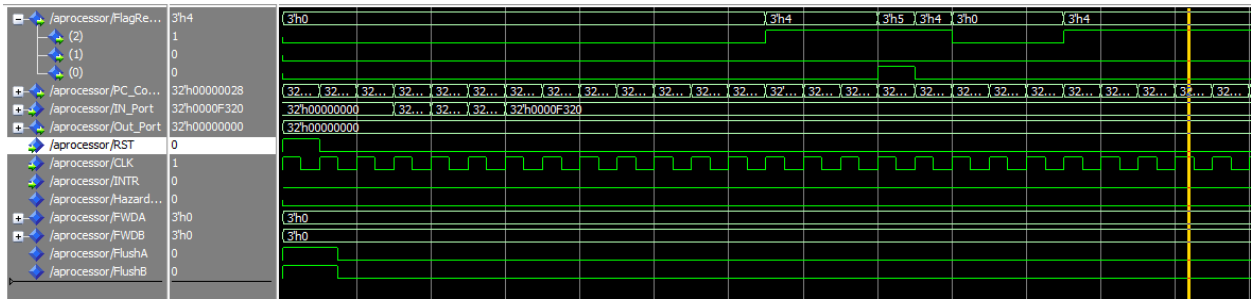
Clock cycle 23

Instr11 has written back the value to R2.

```
00000000 |0000001D 0000000C 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Clock cycle 25

Instr12 has no flag updates.



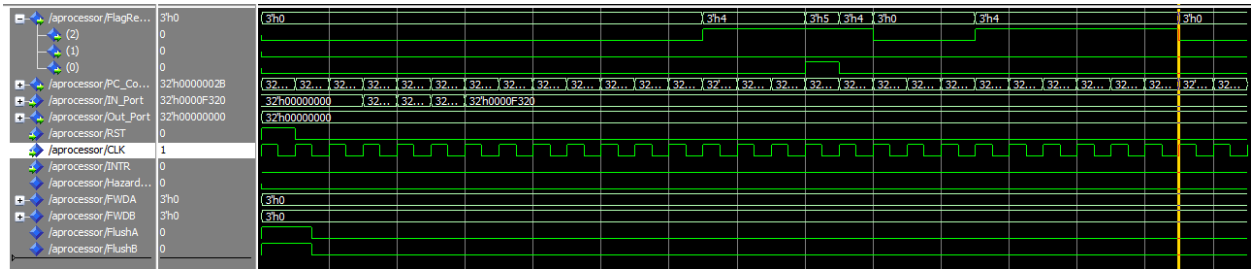
Clock cycle 26

Instr12 has written back the value to R2 & R5.

```
00000000 0000001D 0000FFFF 0000FFFD 0000F325 0000000C 00000000 00000000
```

Clock cycle 28

Instr13 has updated the flags.



Clock cycle 29

Instr13 has written back the value to R2.

```
00000000 0000001D 0001000B 0000FFFD 0000F325 0000000C 00000000 00000000
```

The Incremental Run for the following cases gives the same output which is correct and reflects that the processor is working correctly and properly as the above hazards are solved by using ForwardingUnit so no need for adding NOP after adding HazardDetection or Flushing as there are no hazards to solve

N.B: The Snapshots are the same for the following 3 cases

The processor will finish working in 17 cycles (K+n-1=5+13-1=17) and as there is no load use case or branch prediction in the above code.

- Without HazardDetection or any Flushing

  ForwardingUnit solved all Hazards

- Without any Flushing

  ForwardingUnit solved all Hazards

- Full Working Processor

  ForwardingUnit solved all Hazards

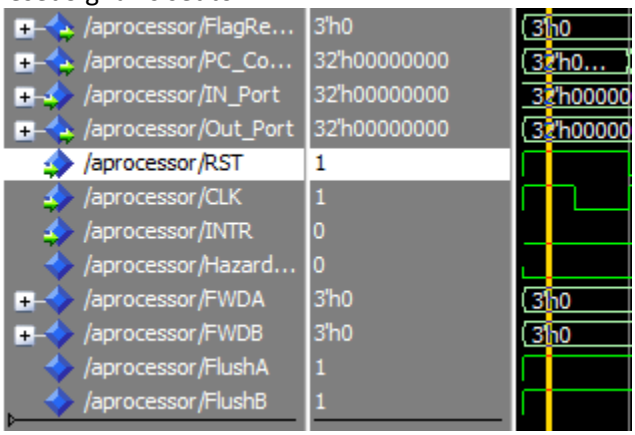
Registers' Values

The initial value for all registers are zero

```
00000000 |00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

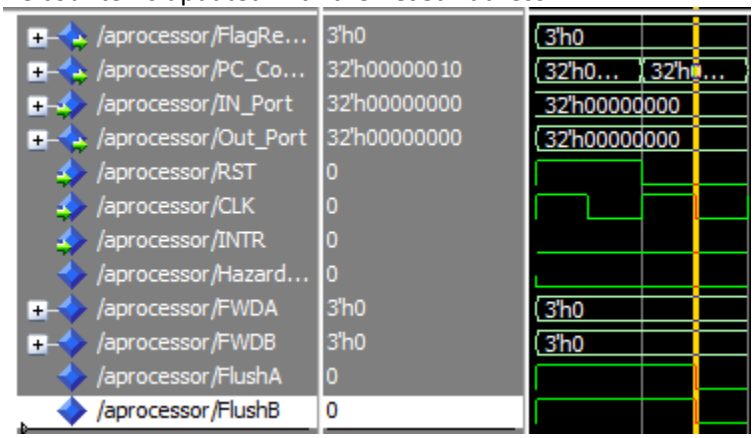
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

Instr1 has written back the value to R1.

```
00000000 |00000005 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 6

Instr2 has written back the value to R2.

```
00000000 |00000005 00000019 00000000 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 7

Instr3 has written back the value to R3.

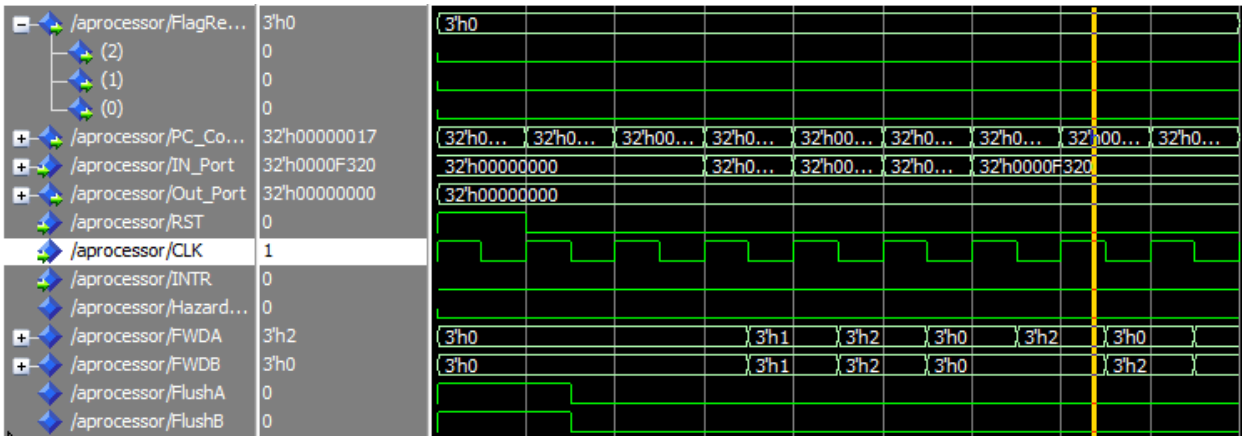
```
00000000 |00000005 00000019 0000FFFD 00000000 00000000 00000000 00000000 00000000
```

Clock cycle 8

Instr4 has written back the value to R4.

```
00000000 |00000005 00000019 0000FFFD 0000F320 00000000 00000000 00000000 00000000
```

Instr5 has no flag updates.

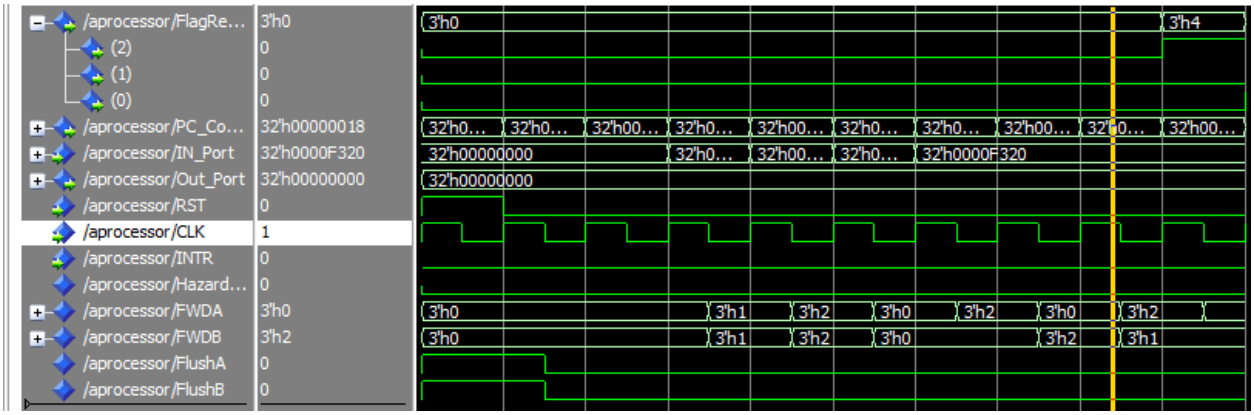


Clock cycle 9

Instr5 has written back the value to R5.

```
00000000 00000005 00000019 0000FFFD 0000F320 0000FFFF 00000000 00000000
```

Instr6 has updated the flags.

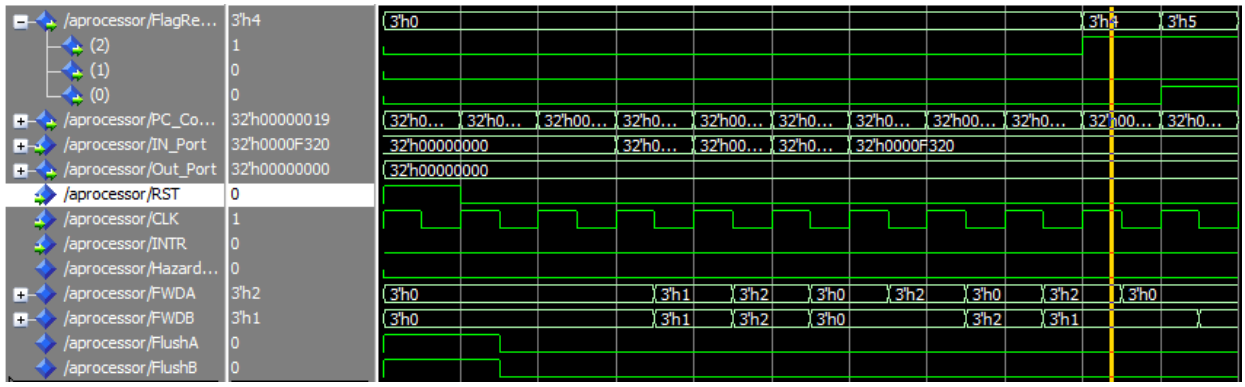


Clock cycle 10

Instr6 has written back the value to R4.

```
00000000 00000005 00000019 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Instr7 has updated the flags.

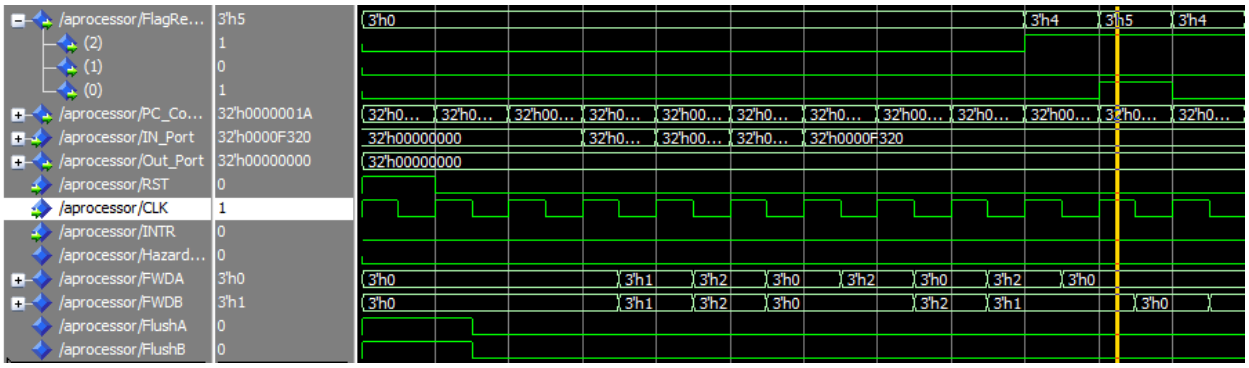


Clock cycle 11

Instr7 has written back the value to R6.

```
00000000 00000005 00000019 0000FFFD 0000F325 0000FFFF 00000CDA 00000000
```

Instr8 has updated the flags.

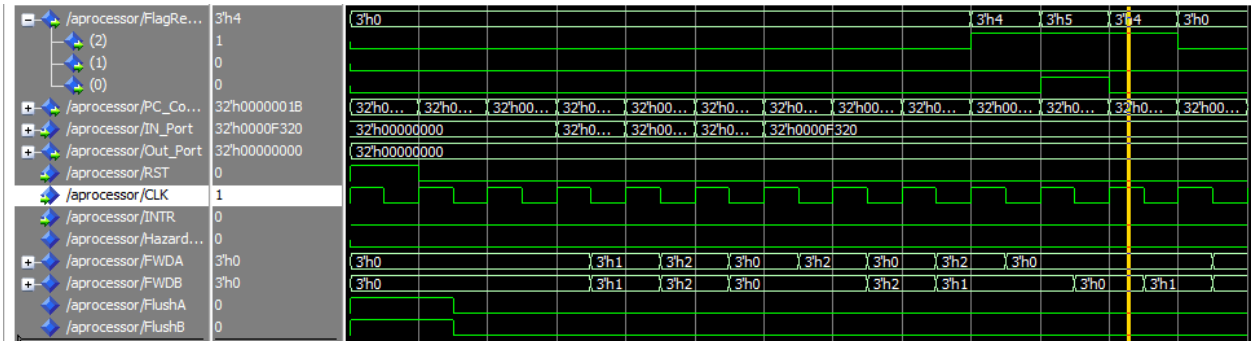


Clock cycle 12

Instr8 has written back the value to R6.

00000000	00000005	00000019	0000FFFD	0000F325	0000FFFF	00000000	00000000

Instr9 has updated the flags.

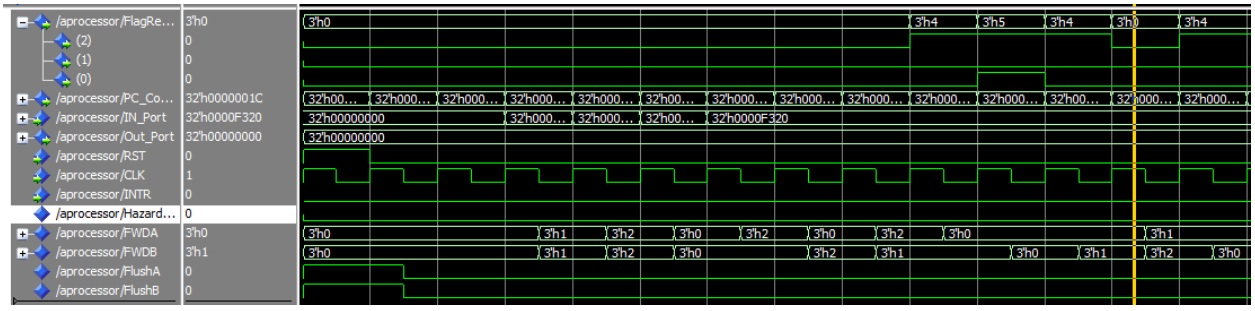


Clock cycle 13

Instr9 has written back the value to R1.

00000000	0000001D	00000019	0000FFFD	0000F325	0000FFFF	00000000	00000000

Instr10 has updated the flags.

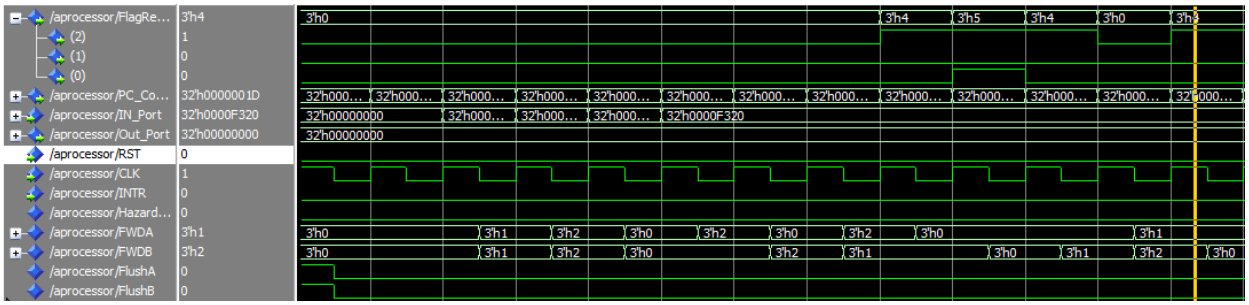


Clock cycle 14

Instr10 has written back the value to R2.

00000000	0000001D	00000064	0000FFFD	0000F325	0000FFFF	00000000	00000000

Instr11 has updated the flags.

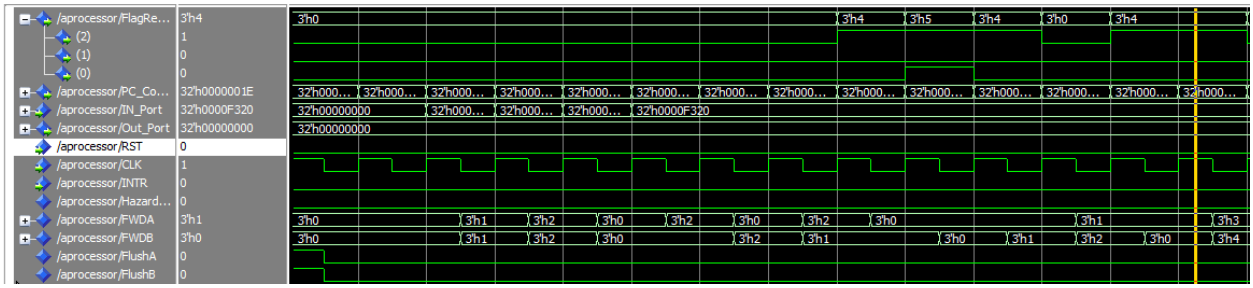


Clock cycle 15

Instr11 has written back the value to R2.

```
00000000 0000001D 0000000C 0000FFFD 0000F325 0000FFFF 00000000 00000000
```

Instr12 has no flag updates.

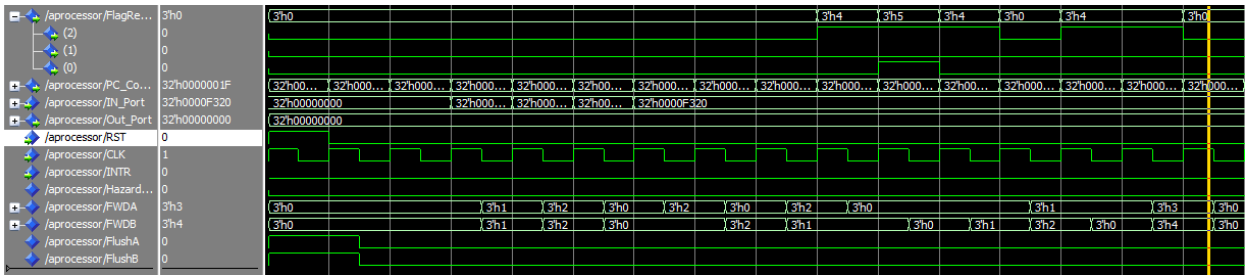


Clock cycle 16

Instr12 has written back the value to R2 & R5.

```
00000000 0000001D 0000FFFF 0000FFFD 0000F325 0000000C 00000000 00000000
```

Instr13 has updated the flags.



Clock cycle 17

Instr13 has written back the value to R2.

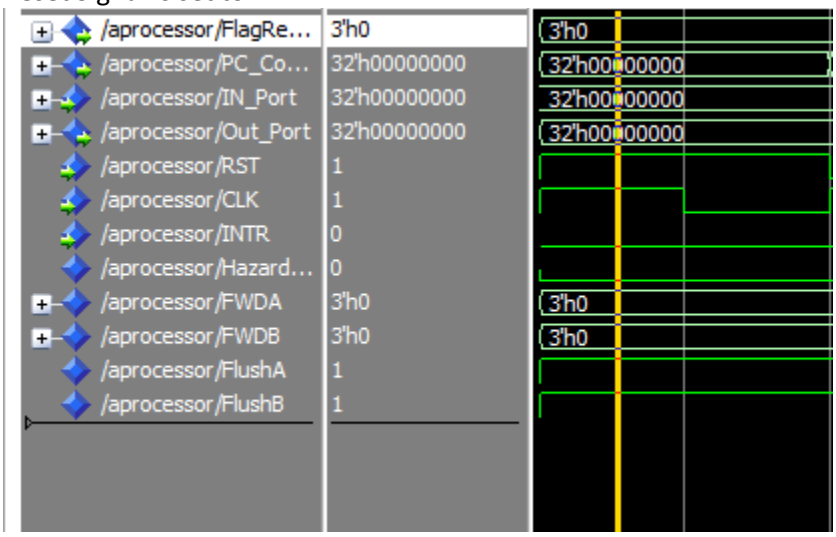
```
00000000 0000001D 0001000B 0000FFFD 0000F325 0000000C 00000000 00000000
```

Branch test cases

Without FU, HDU AND Flushing

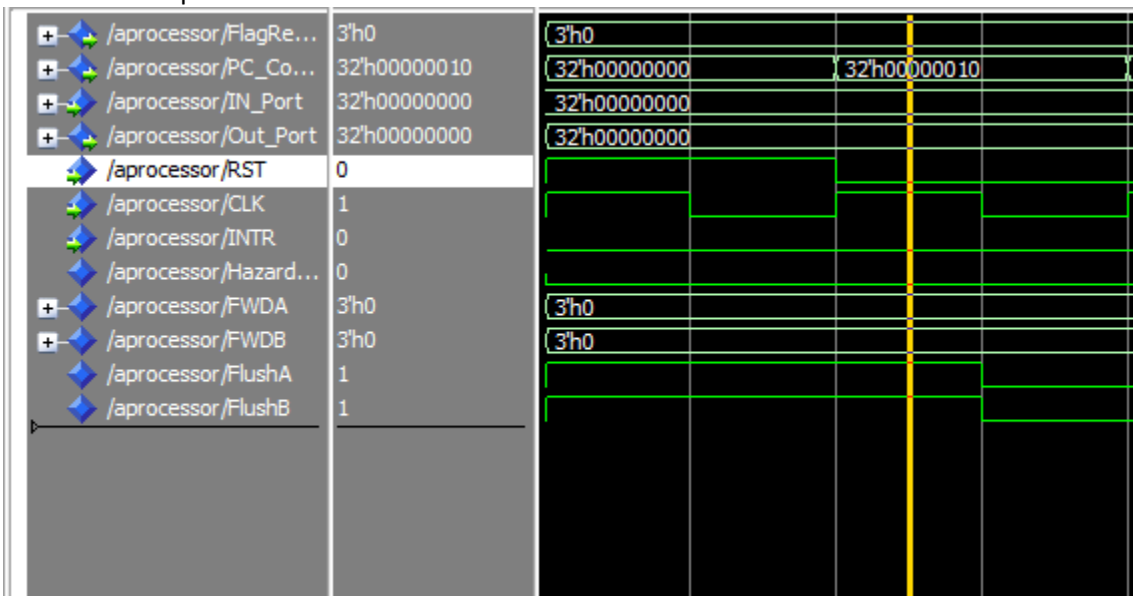
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

R1 is updated with the value x30

Clock cycle 6

R2 is updated with the value x50

Clock cycle 7

R3 is updated with the value x100

Clock cycle 8

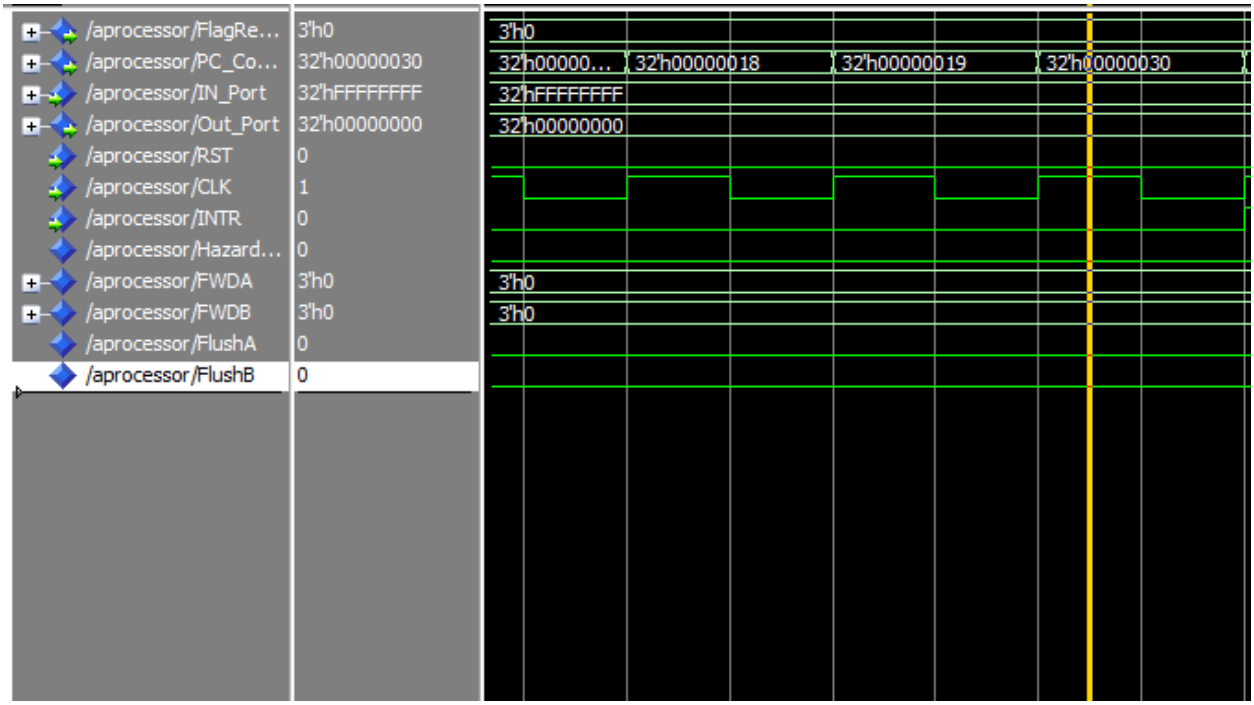
R4 is updated with the value x300

Clock cycle 9

R7 is updated with the value xFFFFFFFF

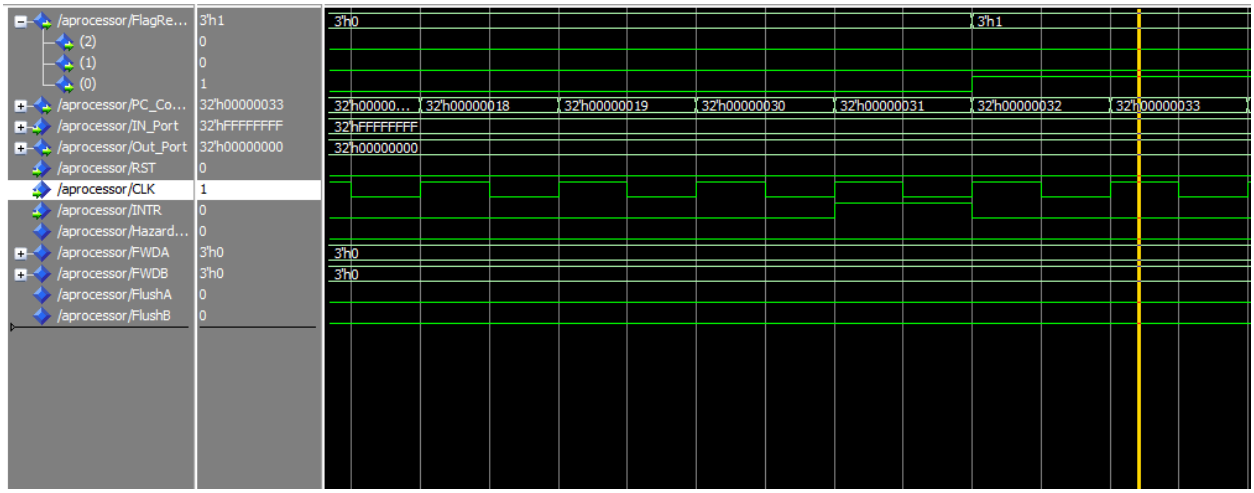
Clock cycle 12

Make the JMP and to x30



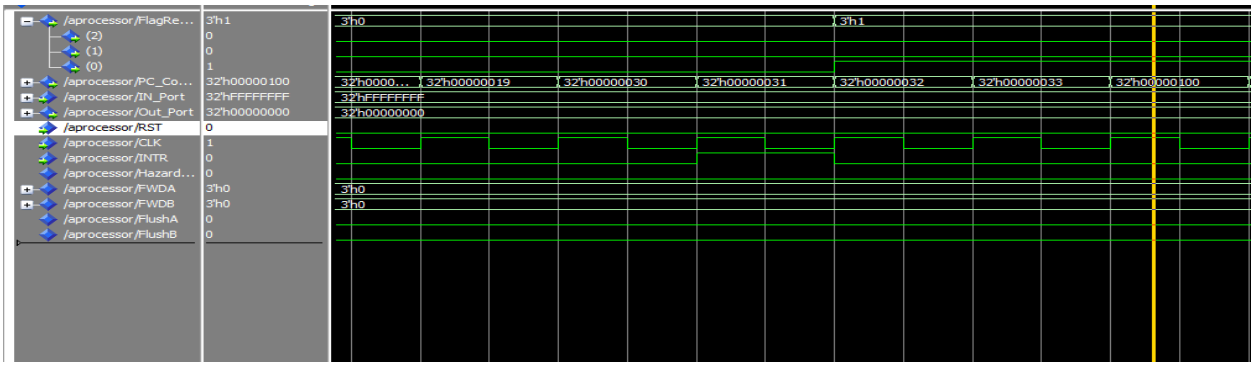
Clock cycle 15

Updated the Flags



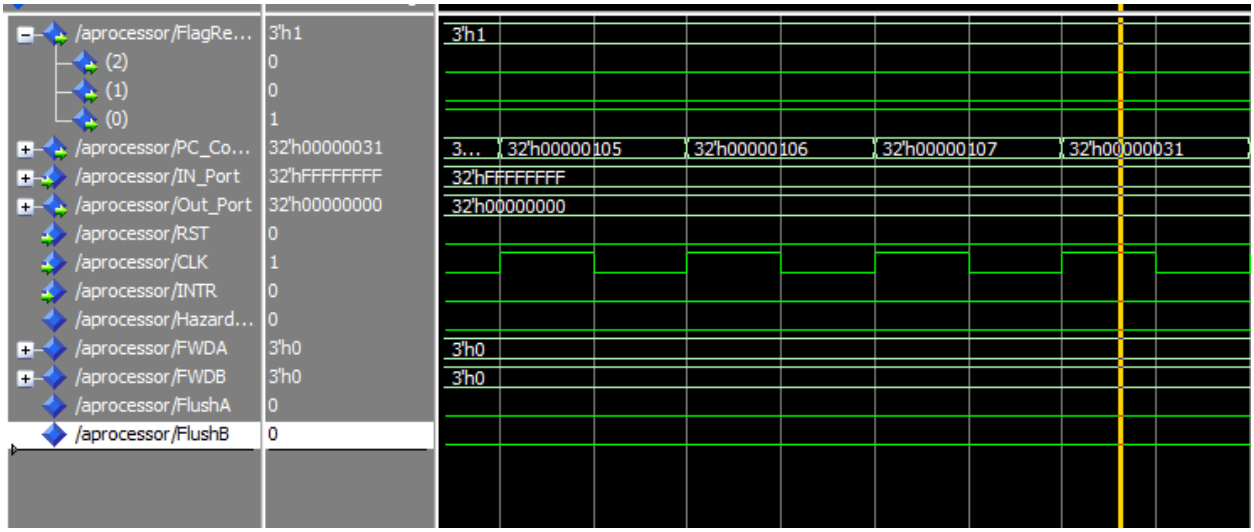
Clock cycle 16

Go to the INT x100



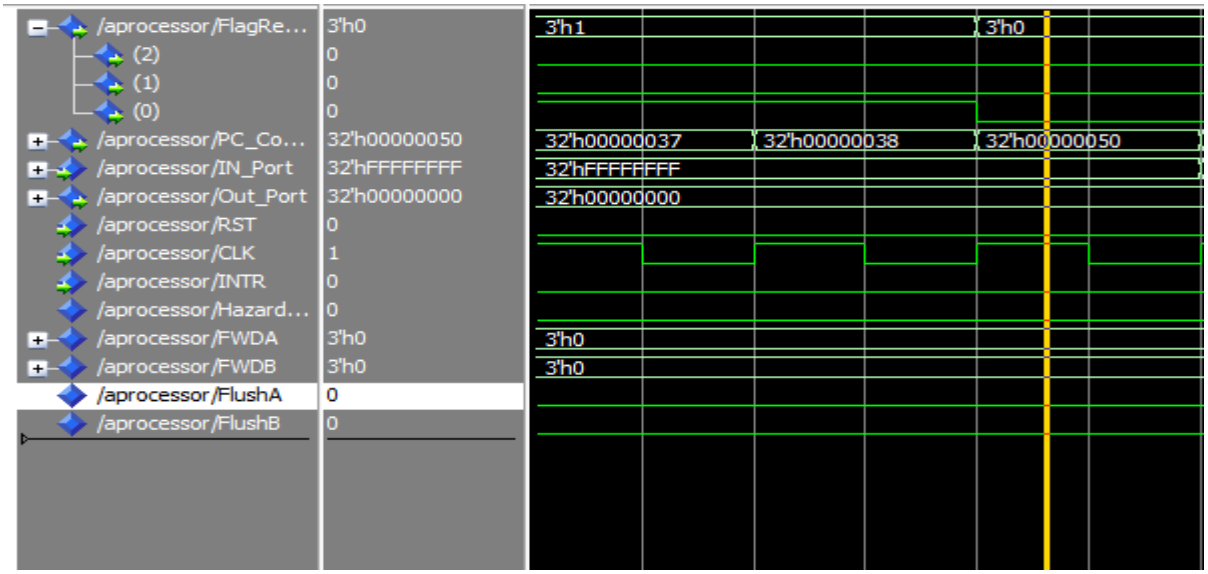
Clock cycle 23

Returned from the INT



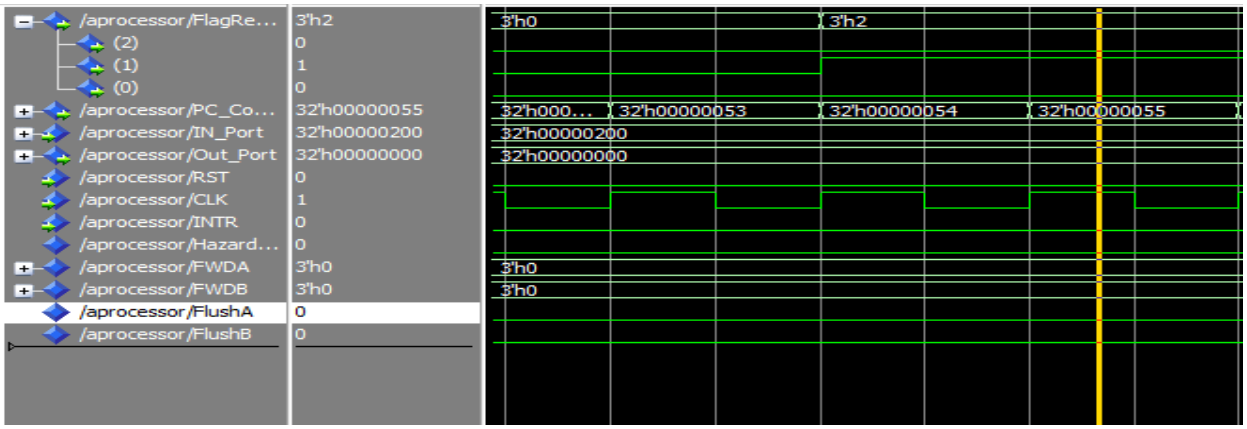
Clock cycle 31

Make the JZ and Set the Zero Flag to zero



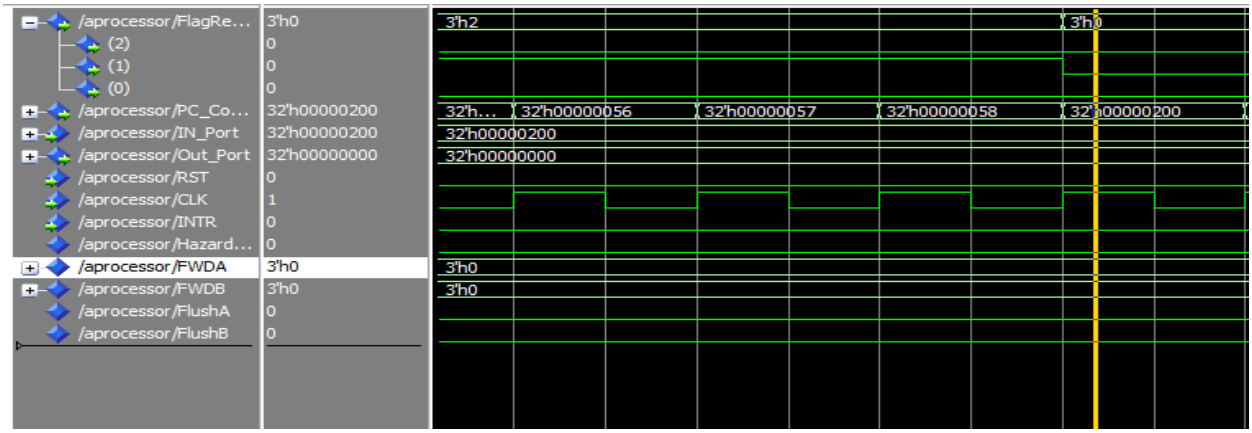
Clock cycle 36

R5 is updated with xFFFFFFFF and Neg Flag is Set to 1



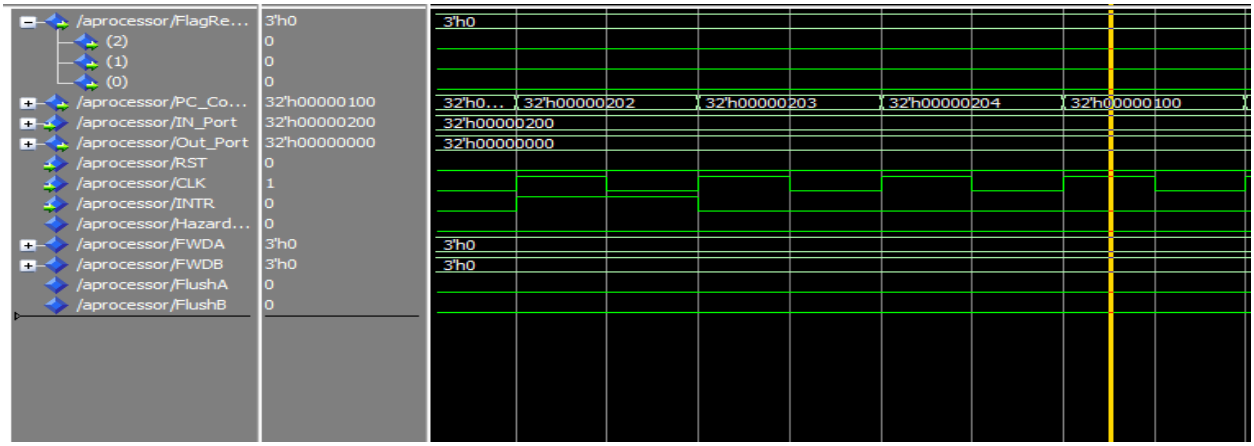
Clock cycle 40

Make the JN and Neg Flag is set to 0.



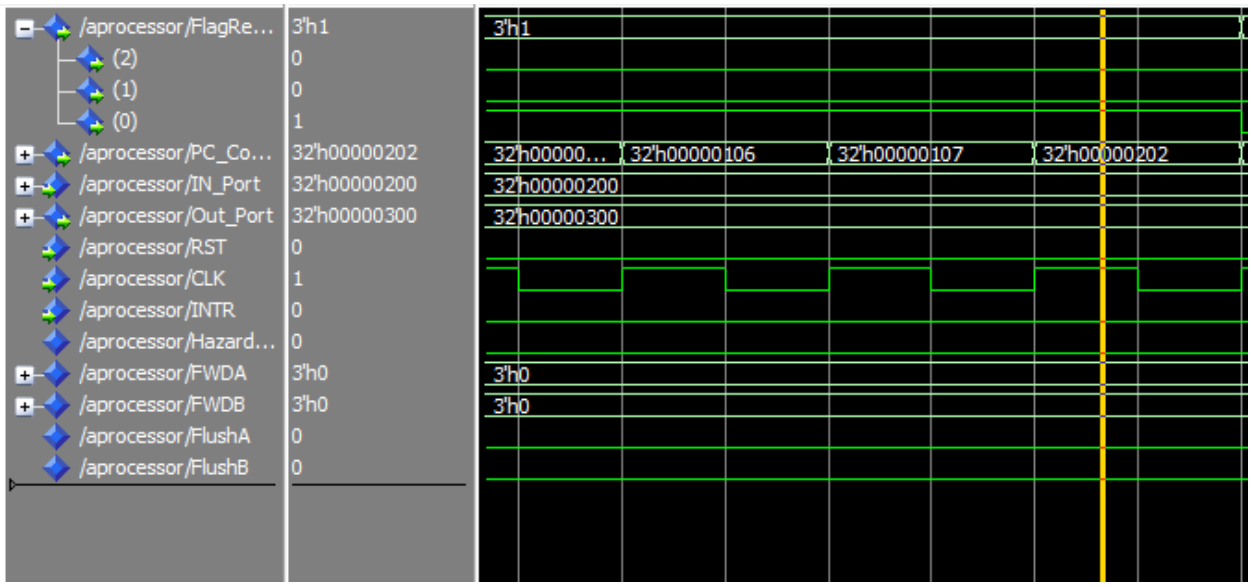
Clock cycle 45

Go to the INT x100



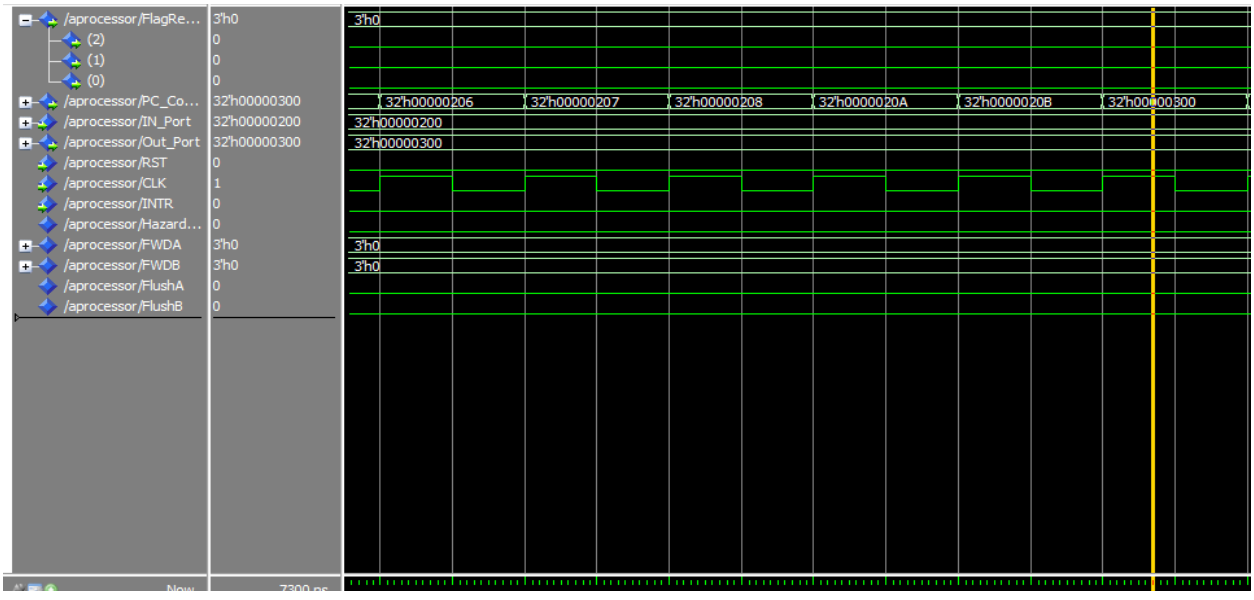
Clock cycle 52

Returned from the INT



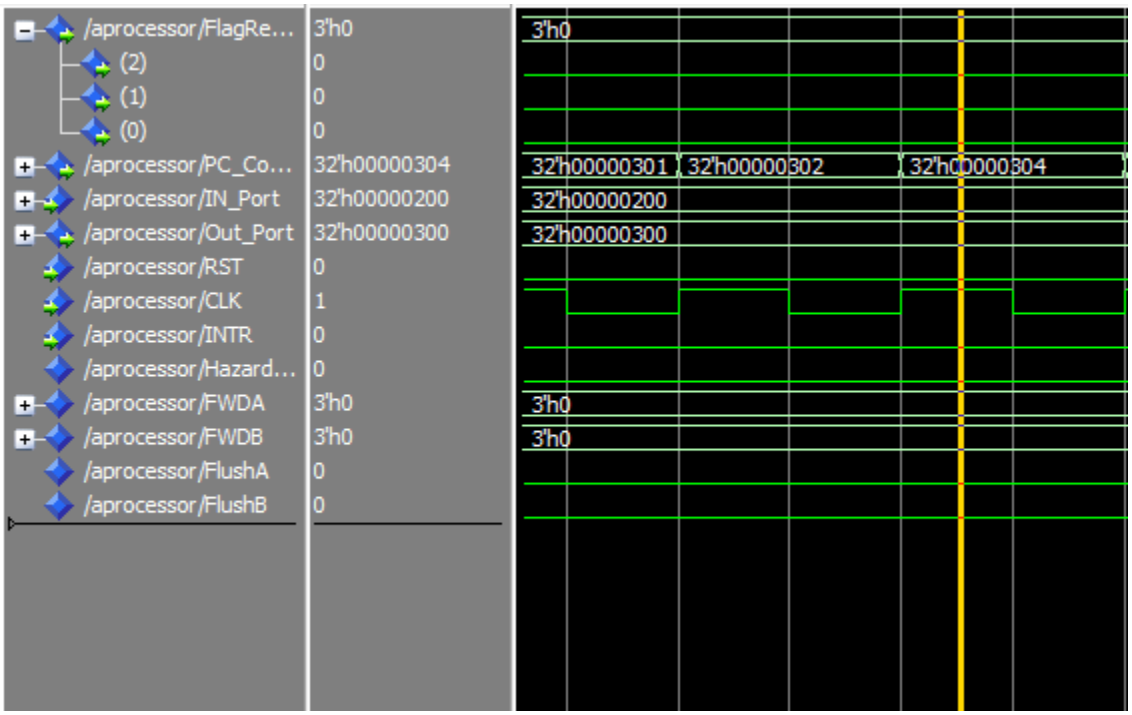
Clock cycle 61

Go to Call x300



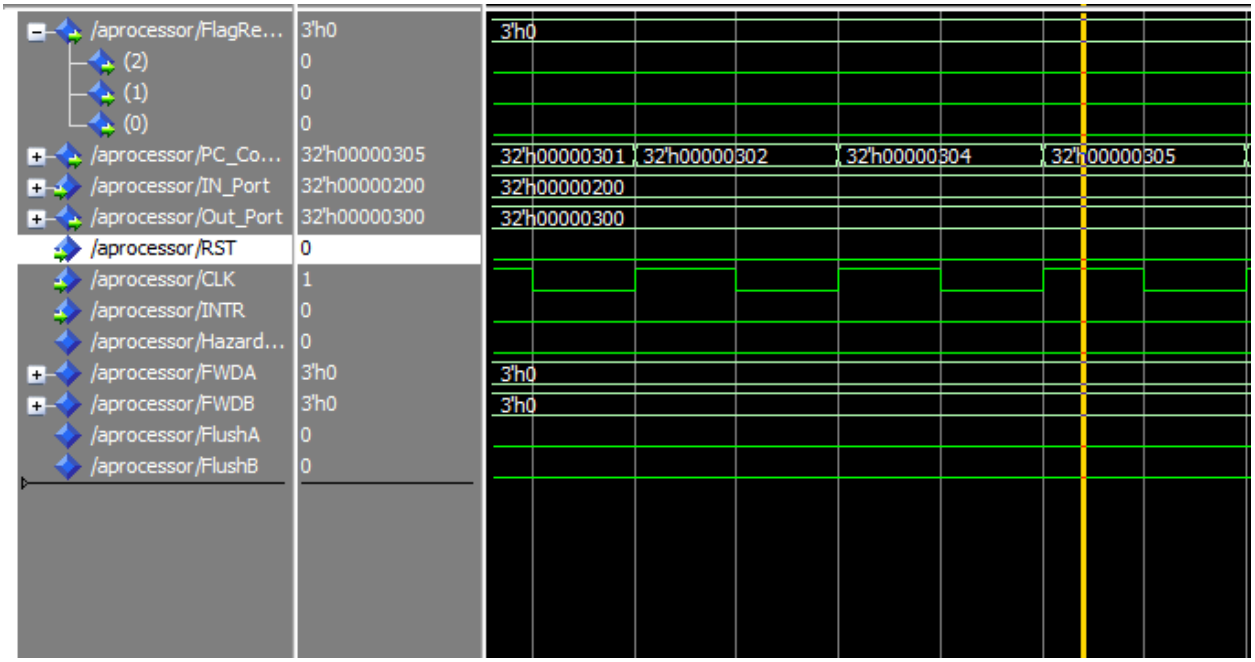
Clock cycle 64

R6 is updated with x400 and Flags are updated right



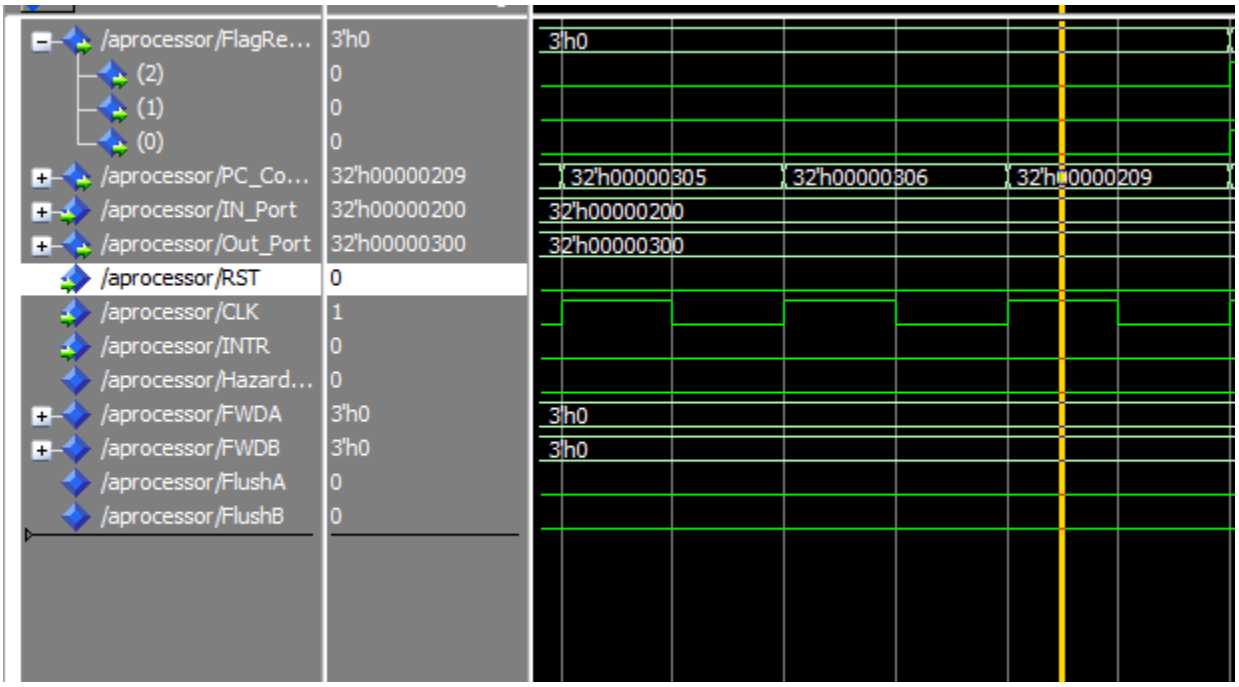
Clock cycle 65

R1 is updated with x80 and Flags are updated right



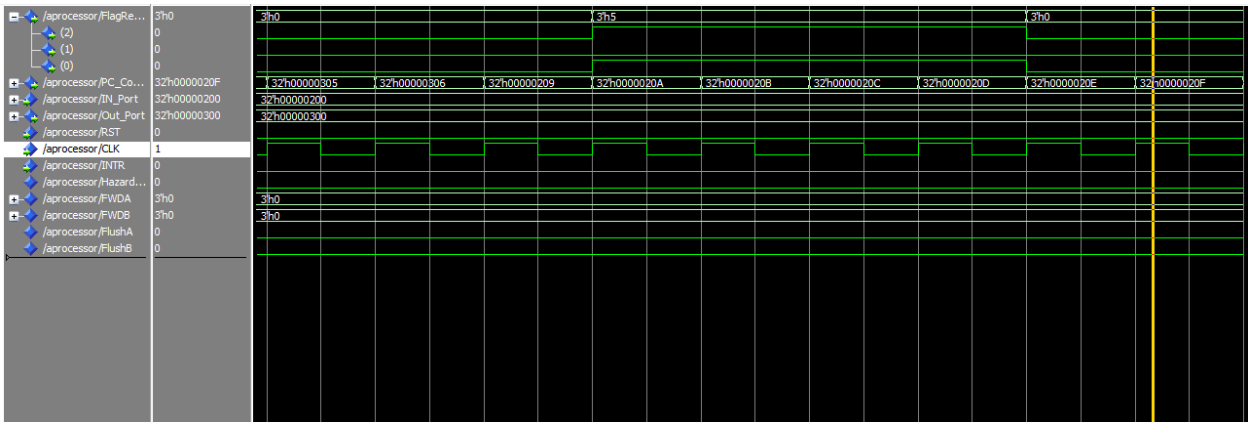
Clock cycle 67

Returned from the call



Clock cycle 73

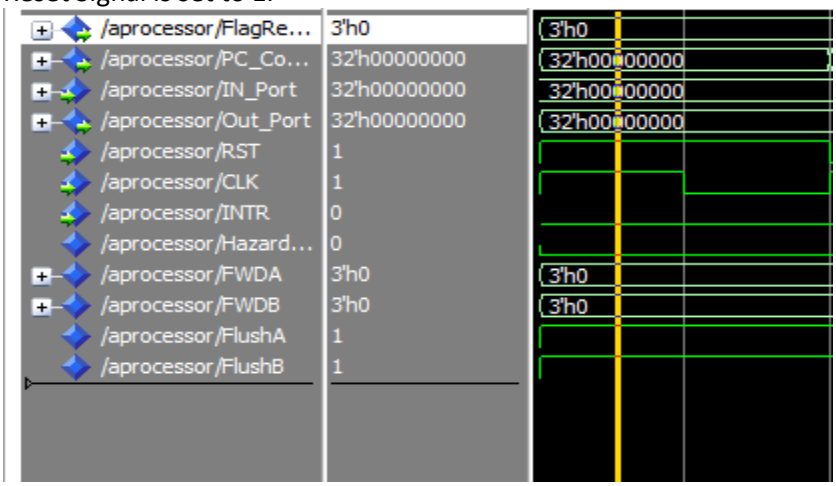
R6 is updated with x401 , Flags are updated right and the processor finished working



Without HDU AND Flushing

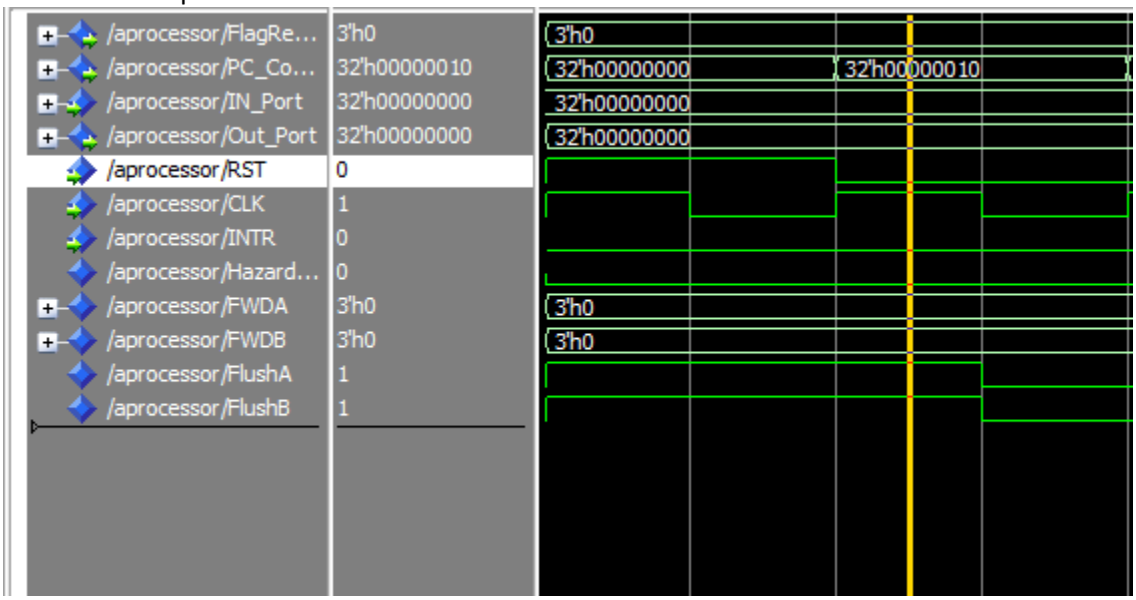
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

R1 is updated with the value x30

Clock cycle 6

R2 is updated with the value x50

Clock cycle 7

R3 is updated with the value x100

Clock cycle 8

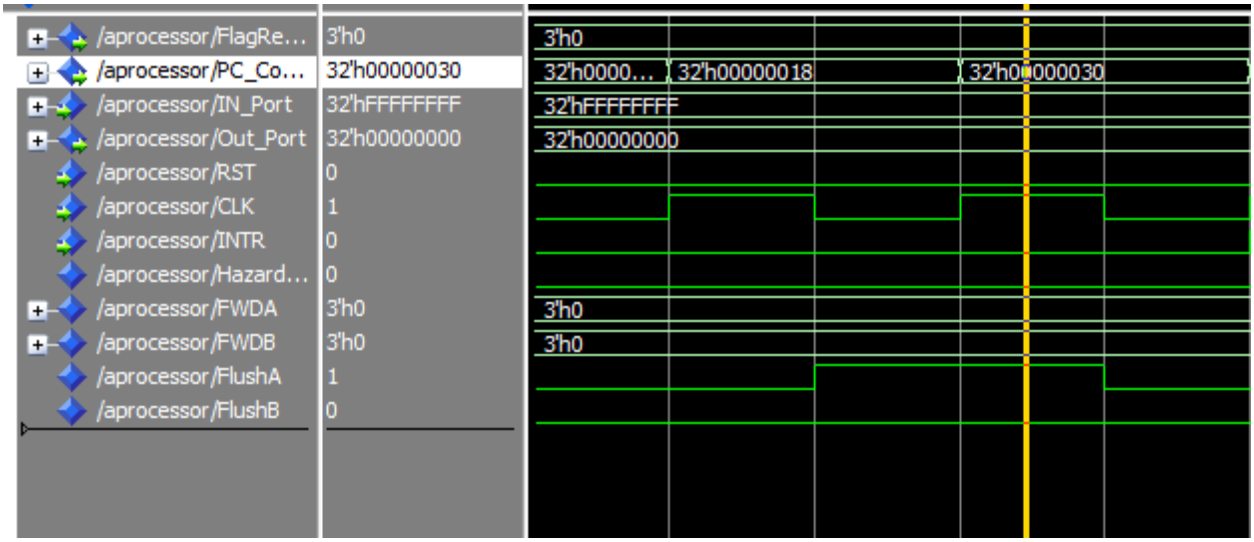
R4 is updated with the value x300

Clock cycle 9

R7 is updated with the value xFFFFFFFF

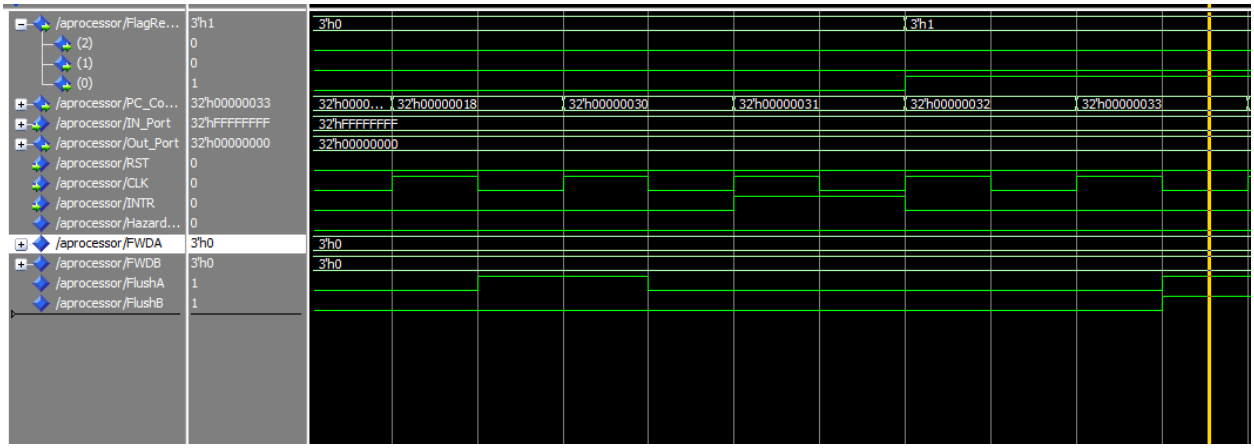
Clock cycle 11

Make the JMP and to x30



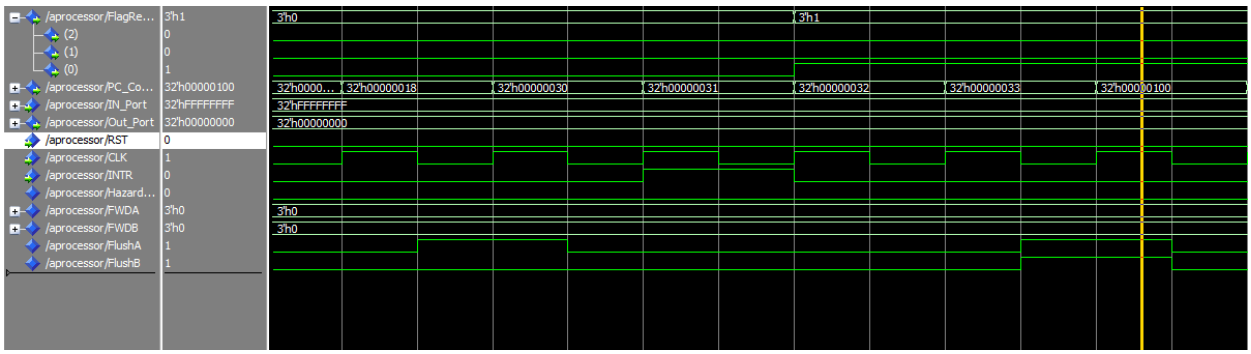
Clock cycle 14

Updated the Flags



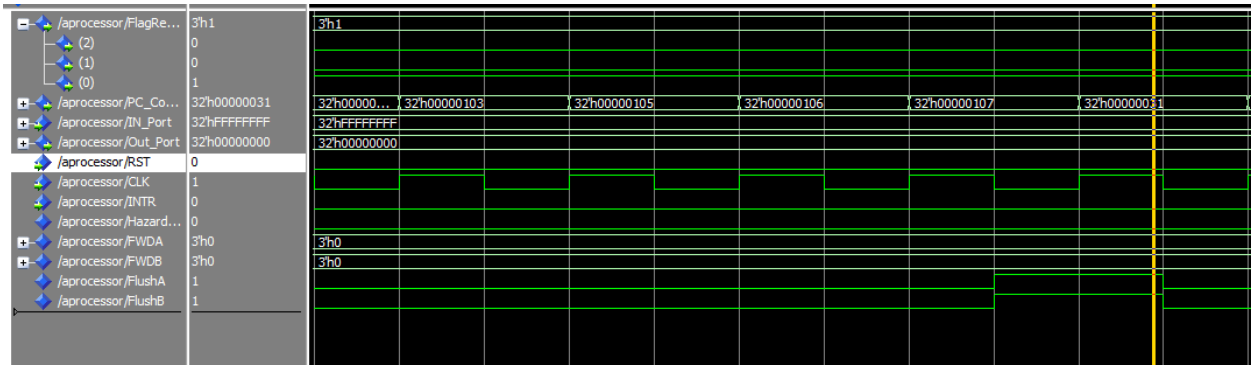
Clock cycle 15

Go to the INT x100



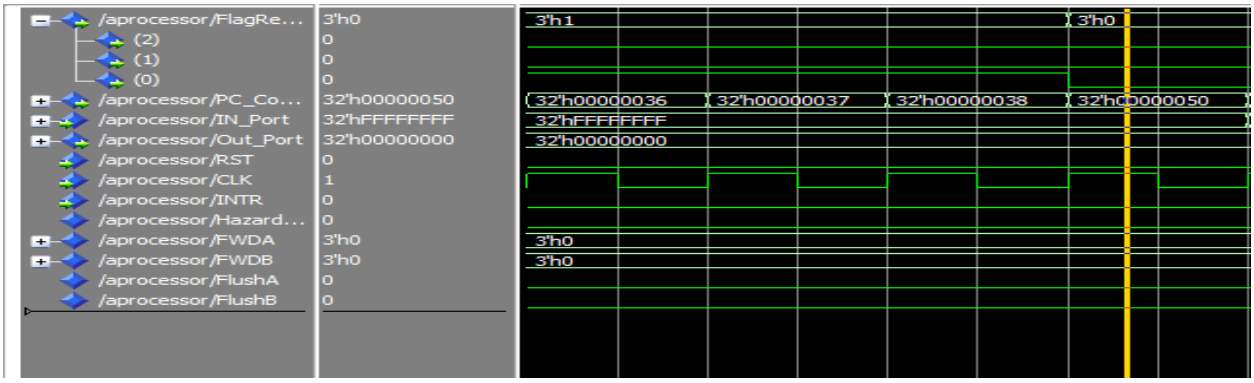
Clock cycle 22

Returned from the INT



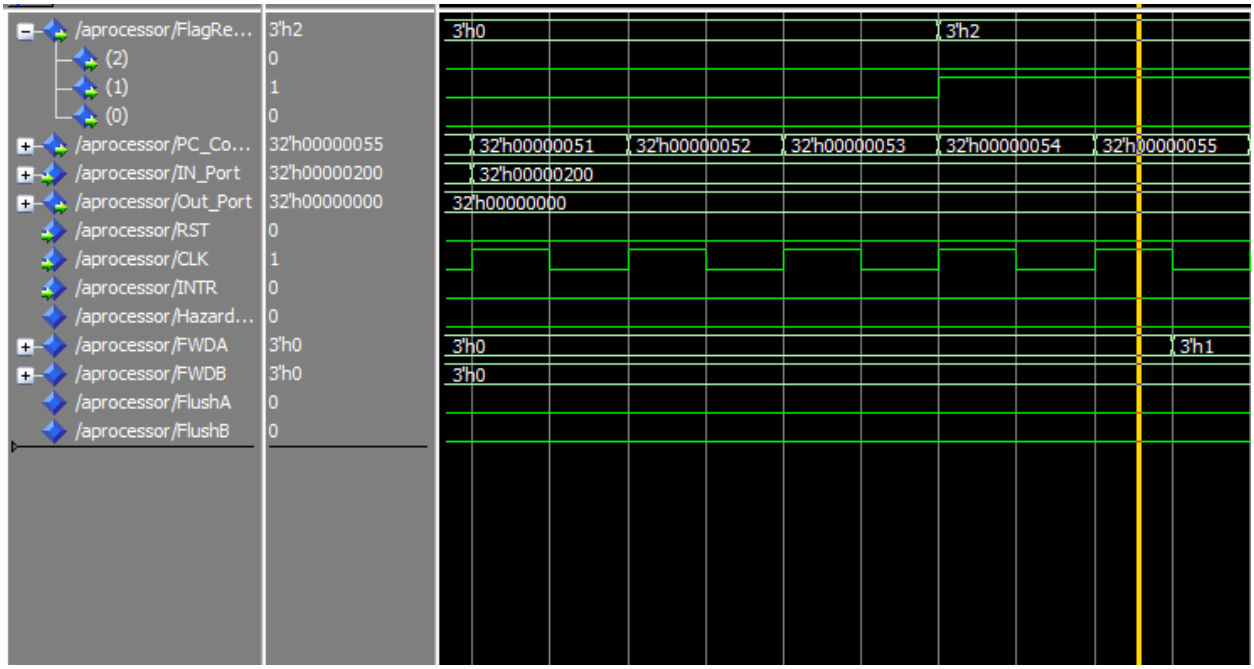
Clock cycle 30

Make the JZ and Set the Zero Flag to zero



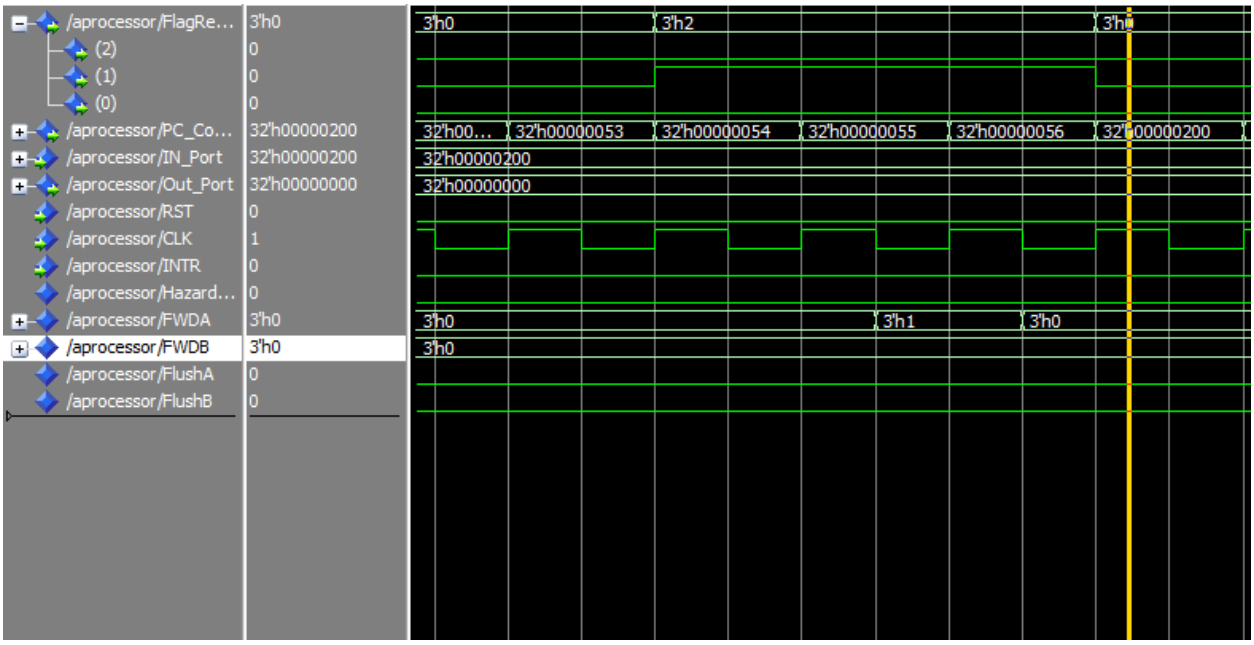
Clock cycle 35

R5 is updated with xFFFFFFFF and Neg Flag is Set to 1



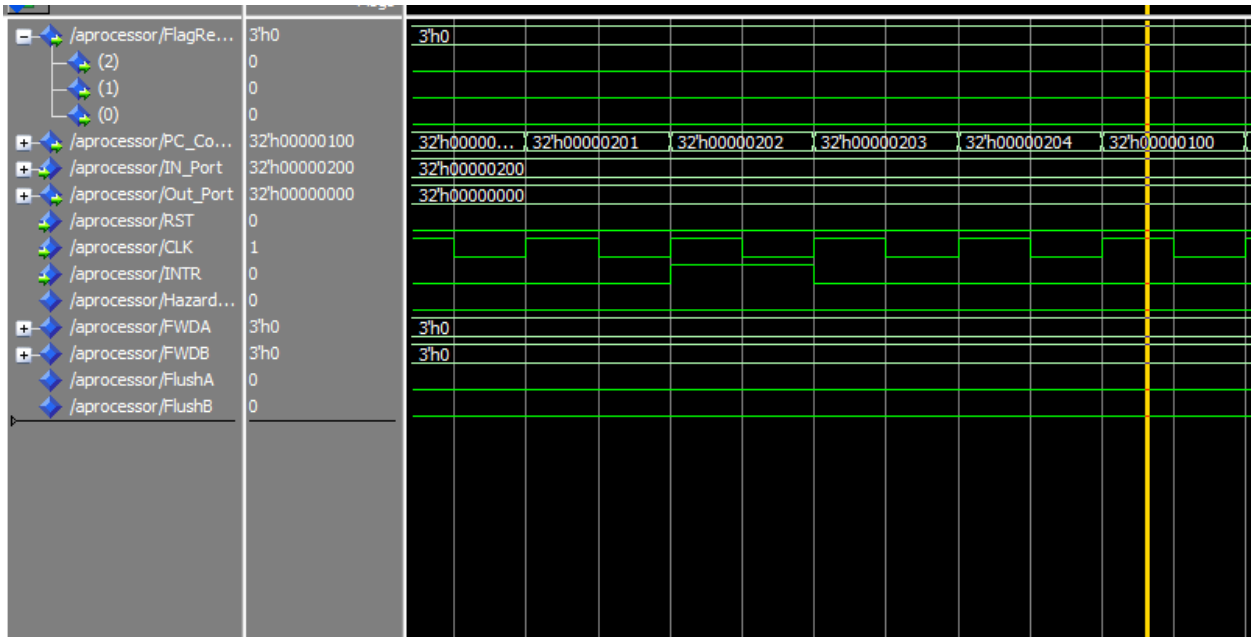
Clock cycle 37

Make the JN and Neg Flag is set to 0.



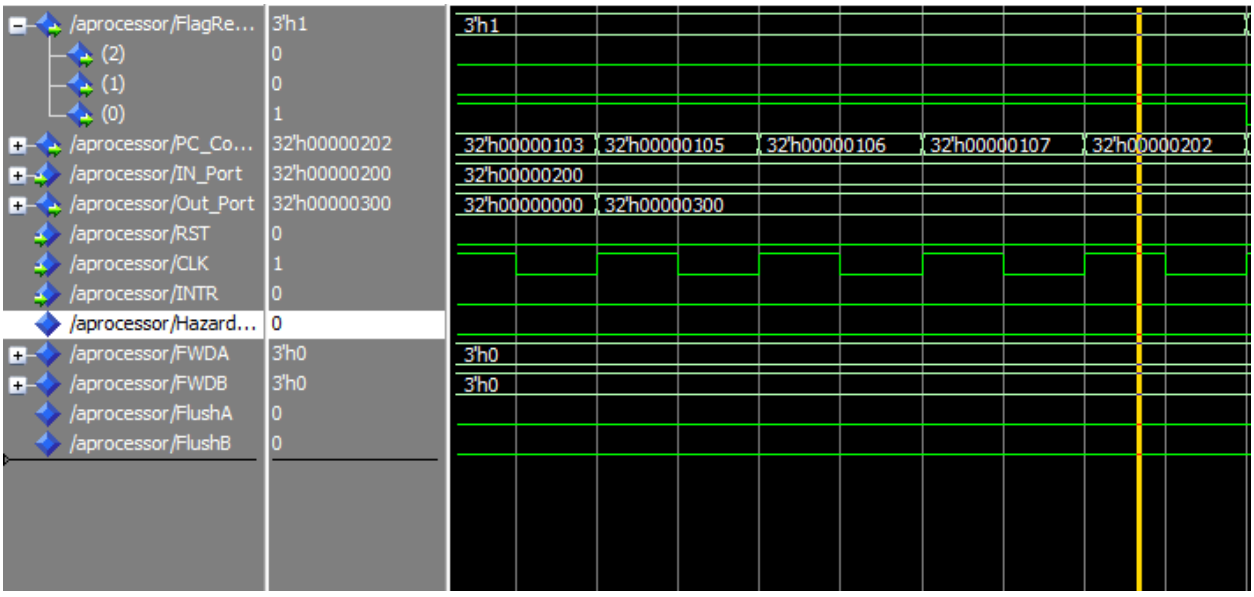
Clock cycle 42

Go to the INT x100



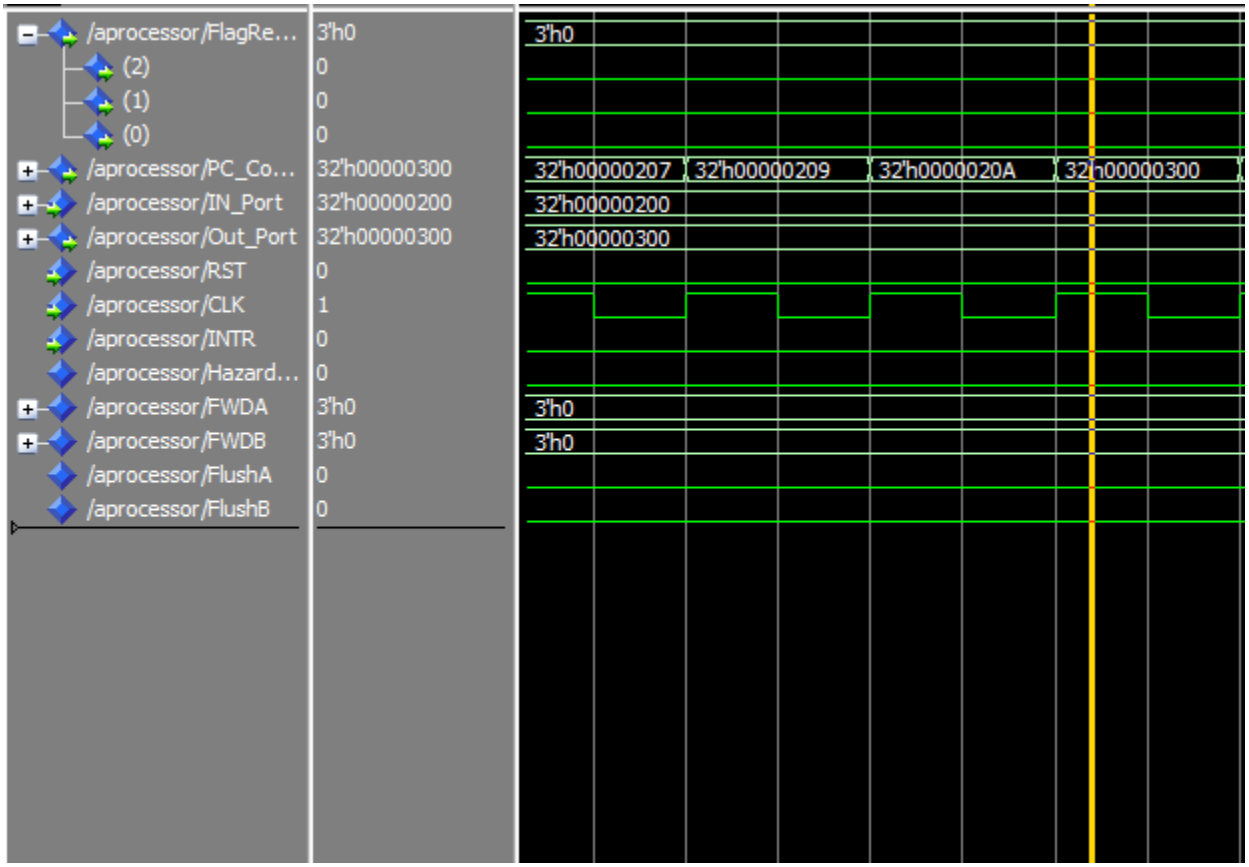
Clock cycle 49

Returned from the INT



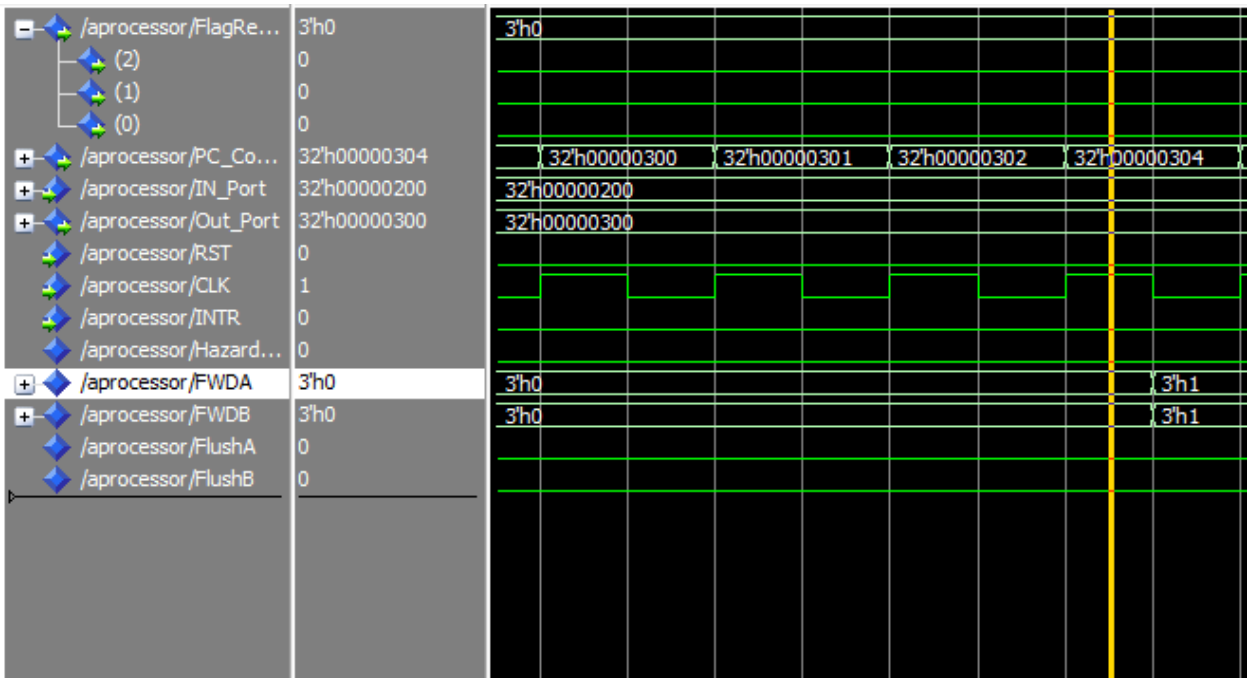
Clock cycle 57

Go to Call x300



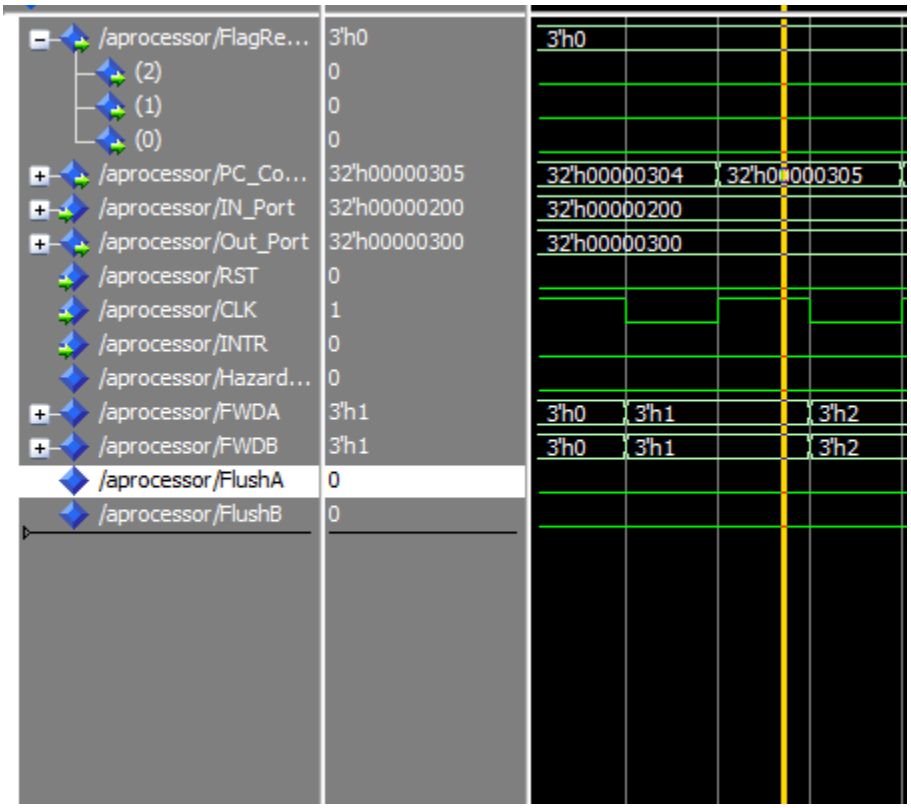
Clock cycle 60

R6 is updated with x400 and Flags are updated right



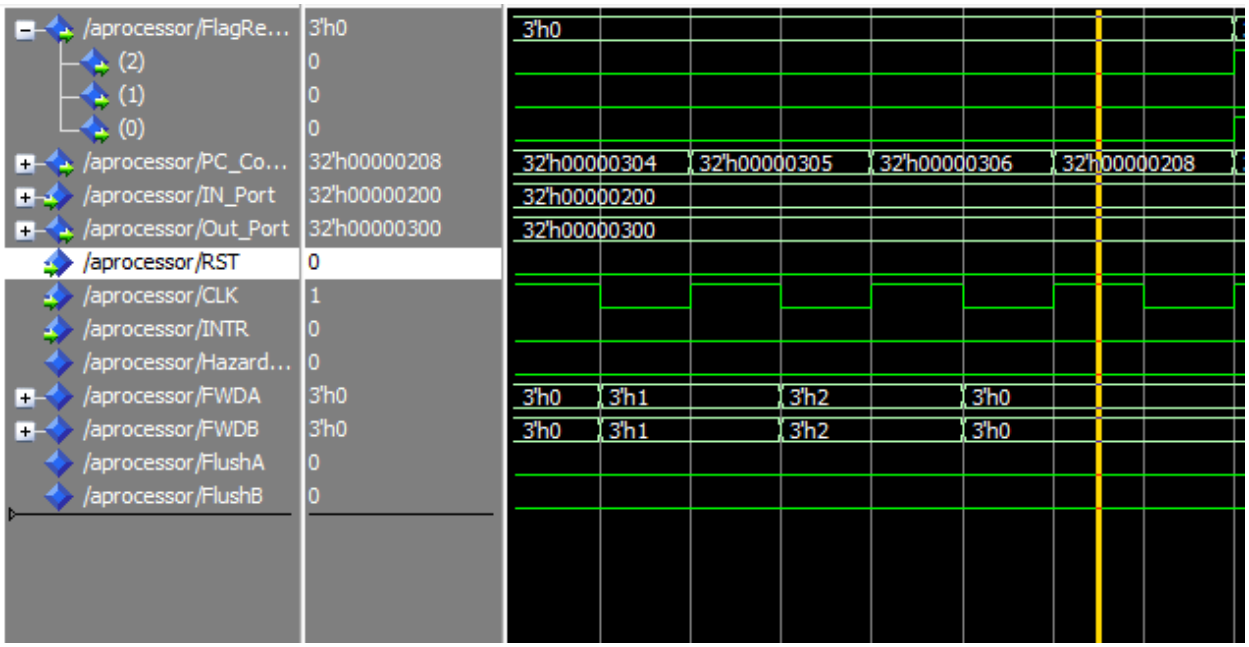
Clock cycle 61

R1 is updated with x80 and Flags are updated right



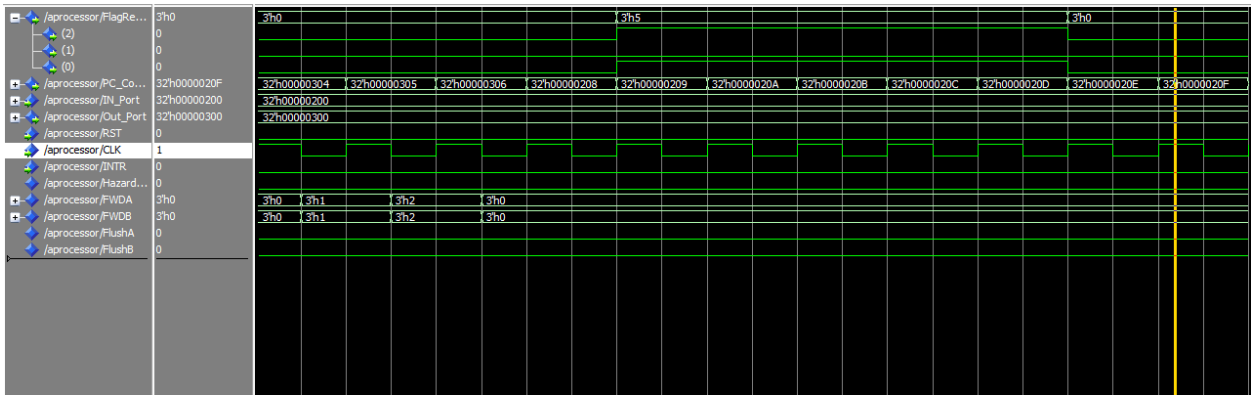
Clock cycle 63

Returned from the call



Clock cycle 70

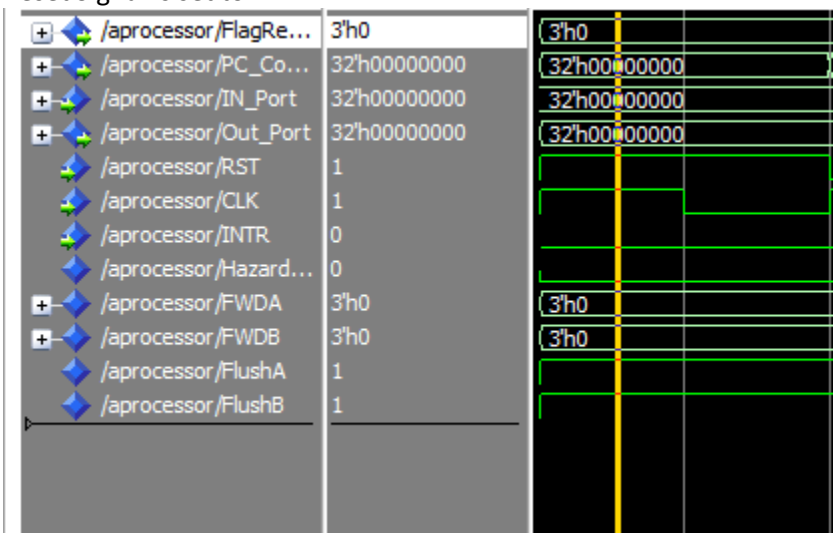
R6 is updated with x401 , Flags are updated right and the processor finished working



Without Flushing Only

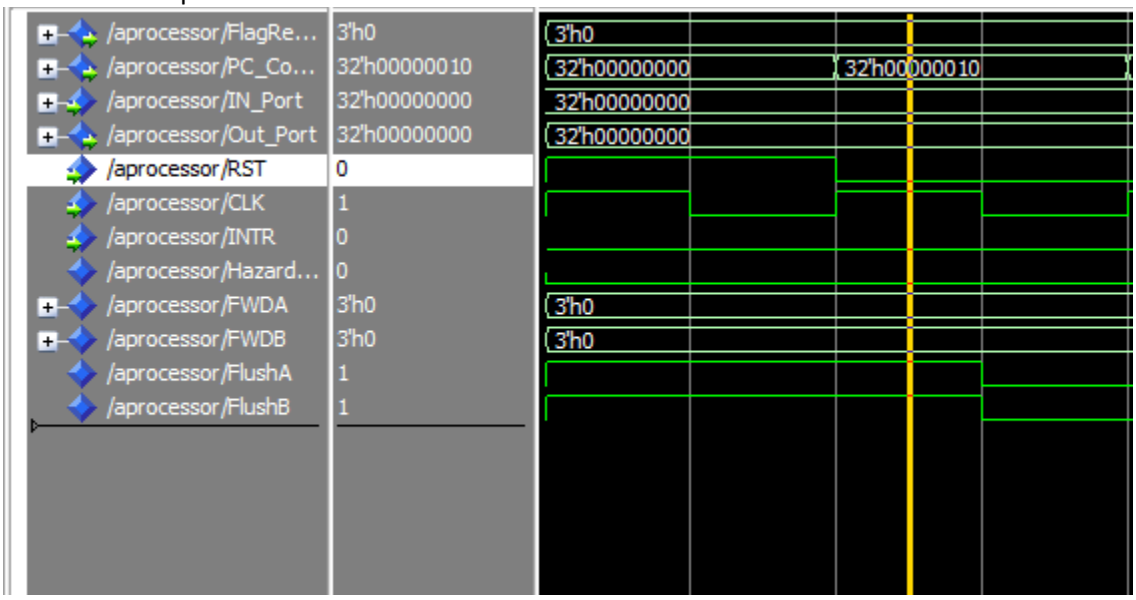
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

R1 is updated with the value x30

Clock cycle 6

R2 is updated with the value x50

Clock cycle 7

R3 is updated with the value x100

Clock cycle 8

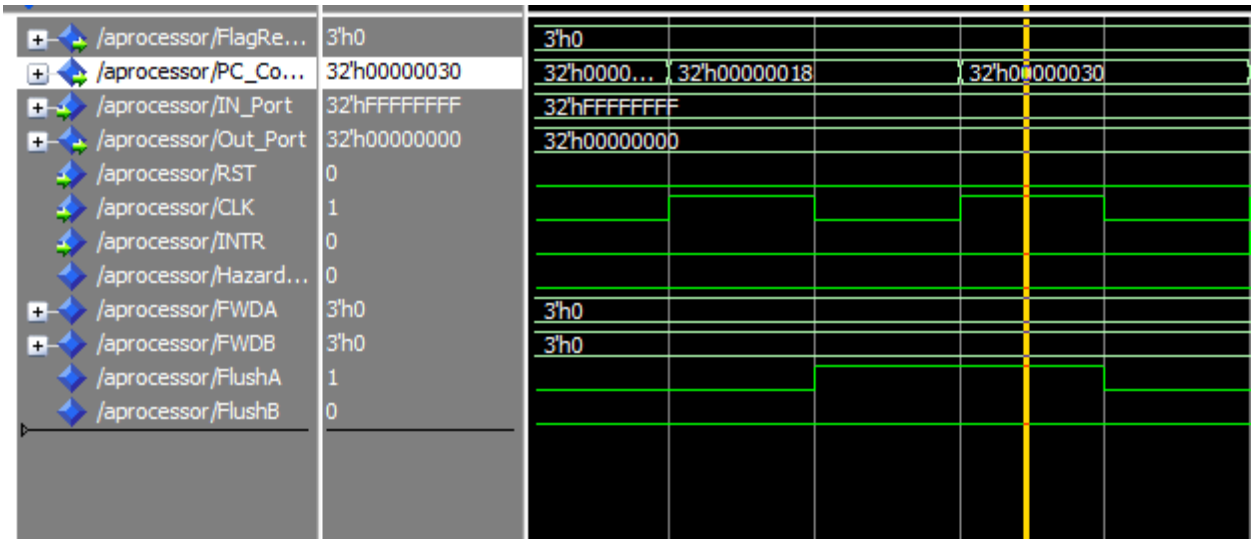
R4 is updated with the value x300

Clock cycle 9

R7 is updated with the value xFFFFFFFF

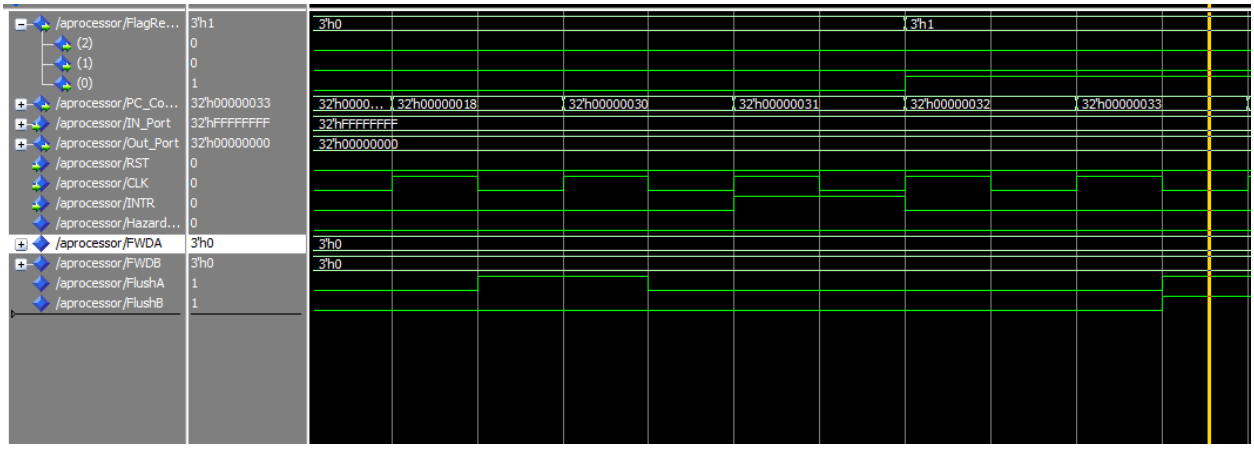
Clock cycle 11

Make the JMP and to x30



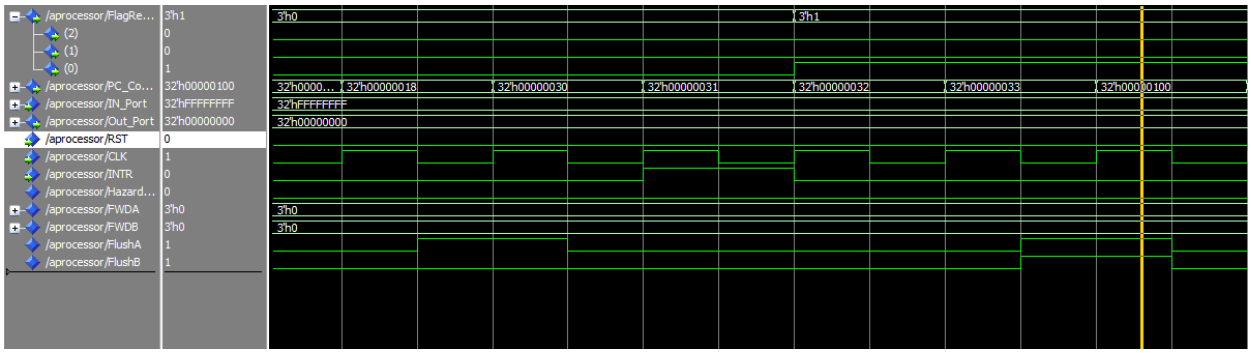
Clock cycle 14

Updated the Flags



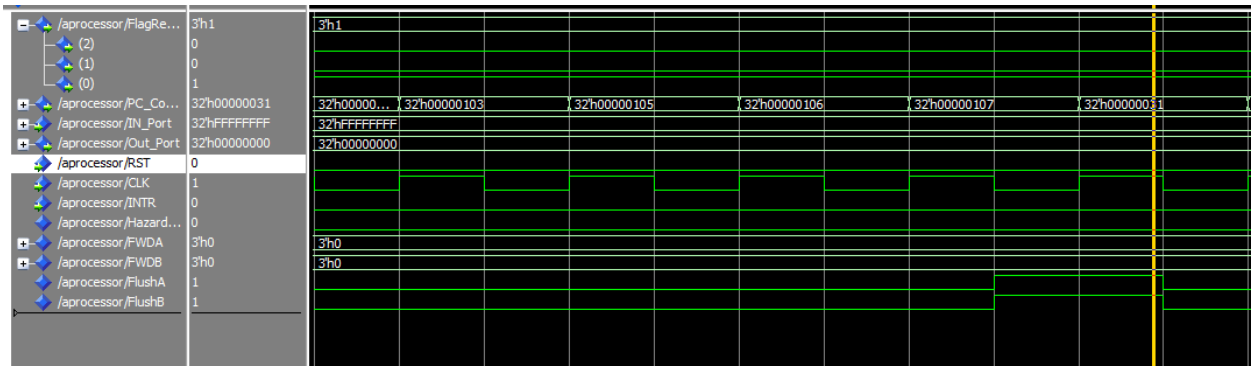
Clock cycle 15

Go to the INT x100



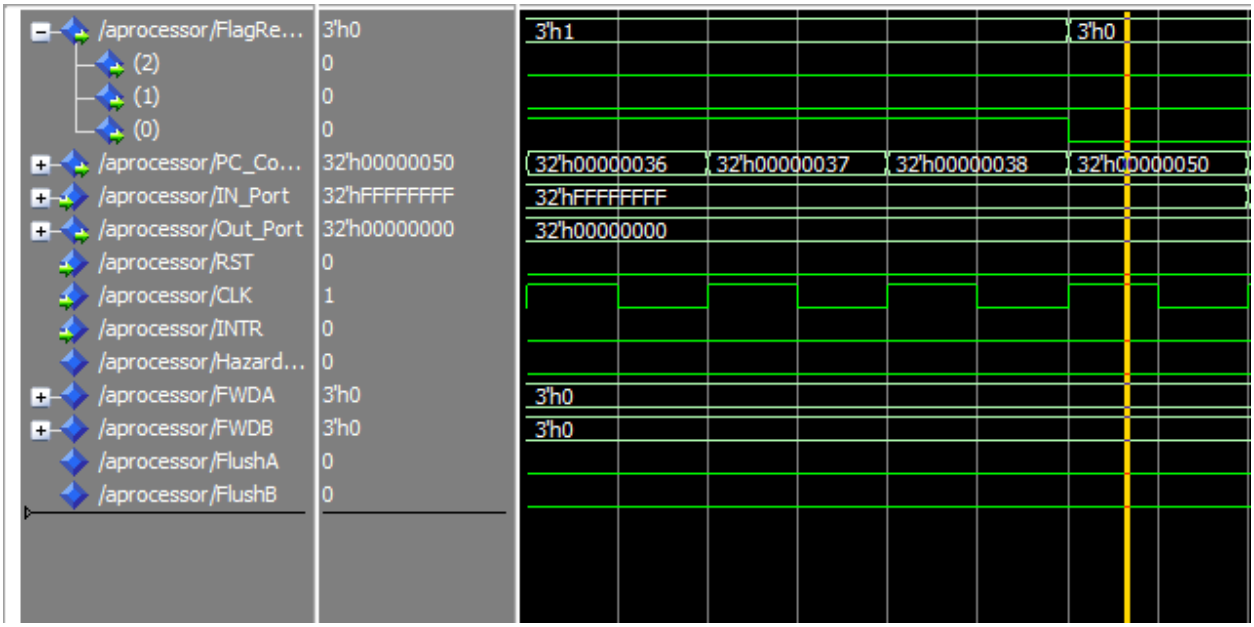
Clock cycle 22

Returned from the INT



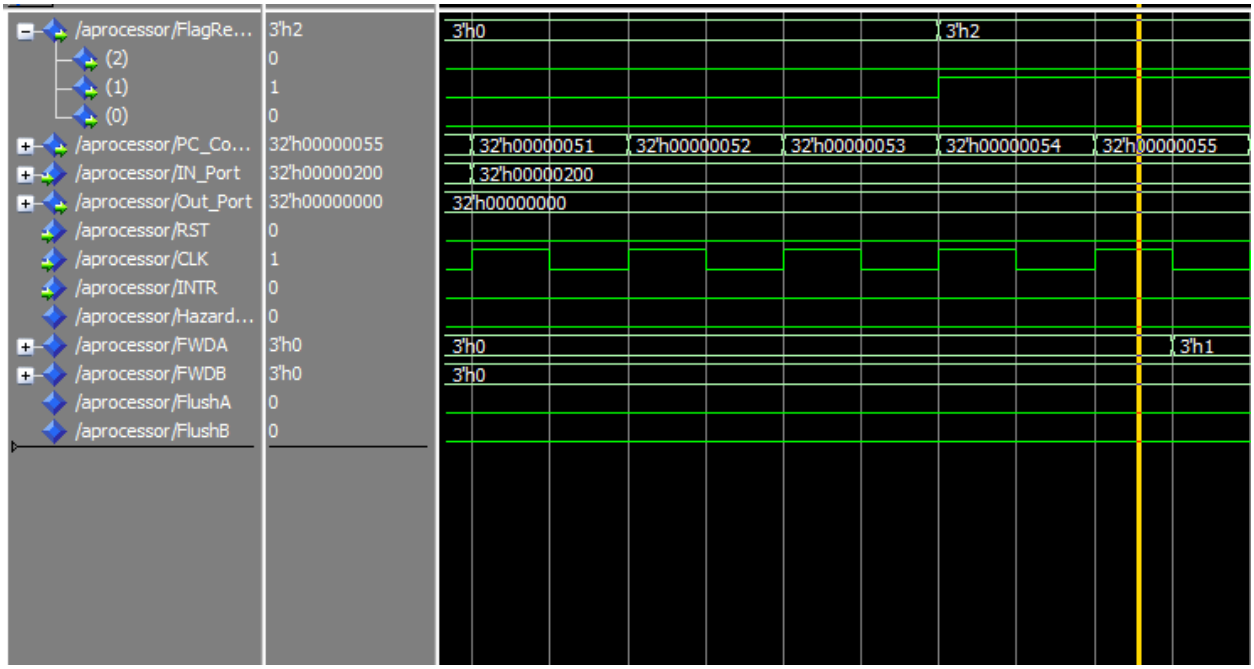
Clock cycle 30

Make the JZ and Set the Zero Flag to zero



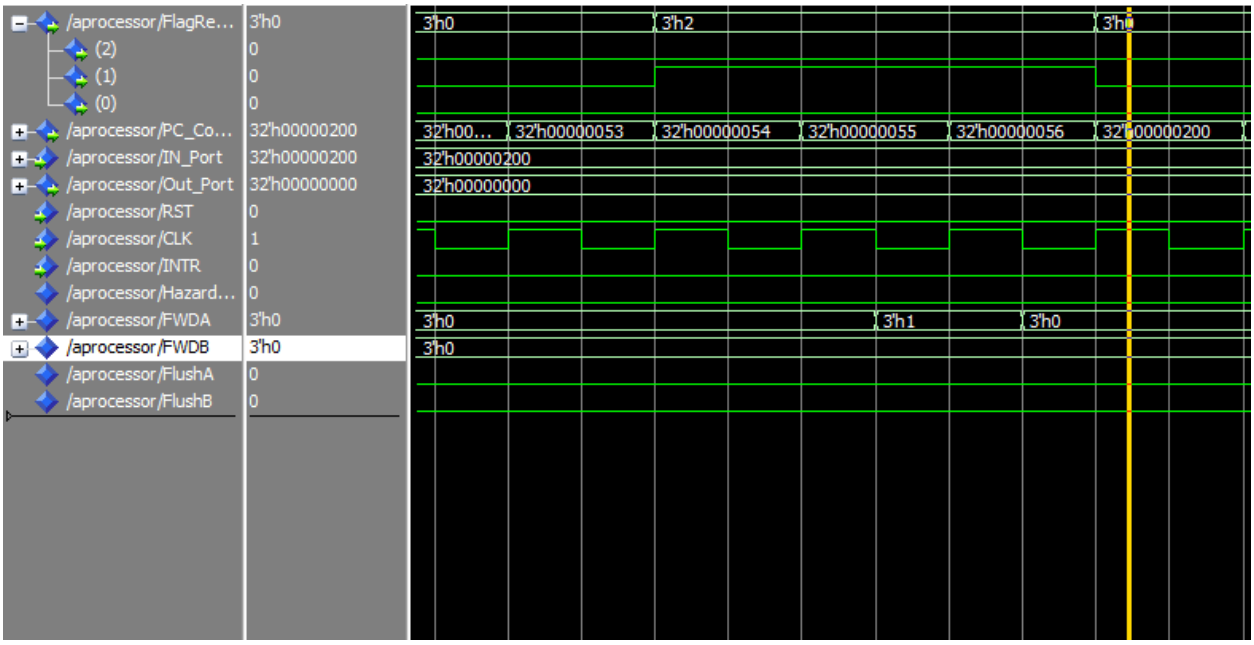
Clock cycle 35

R5 is updated with xFFFFFFFF and Neg Flag is Set to 1



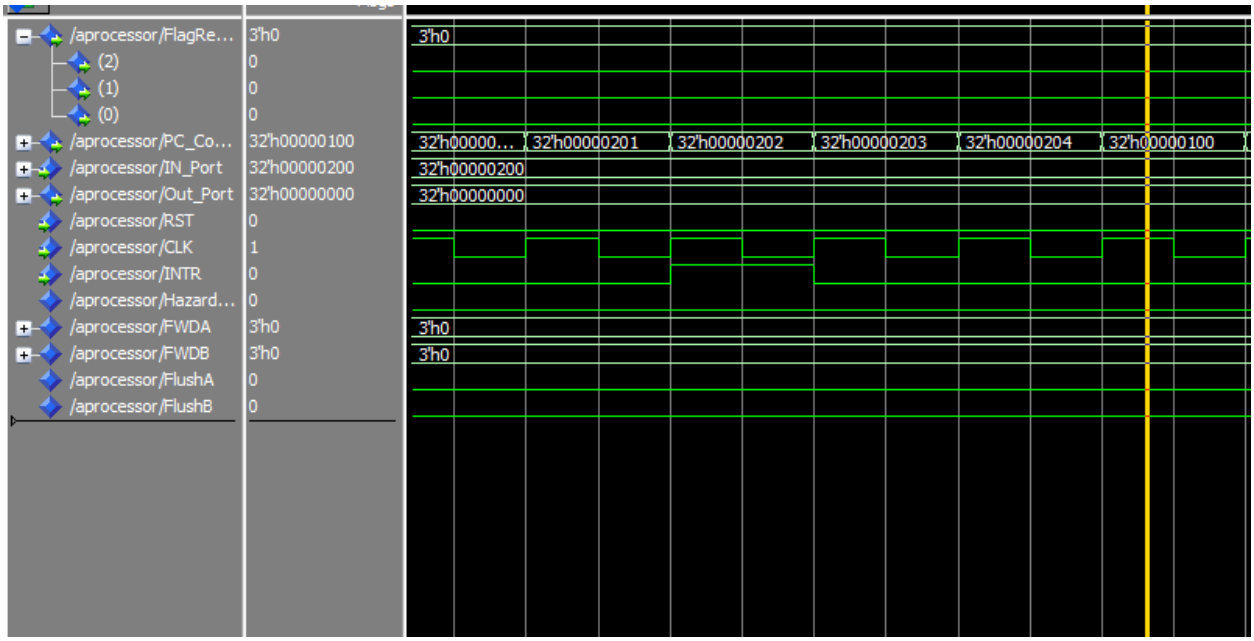
Clock cycle 37

Make the JN and Neg Flag is set to 0.



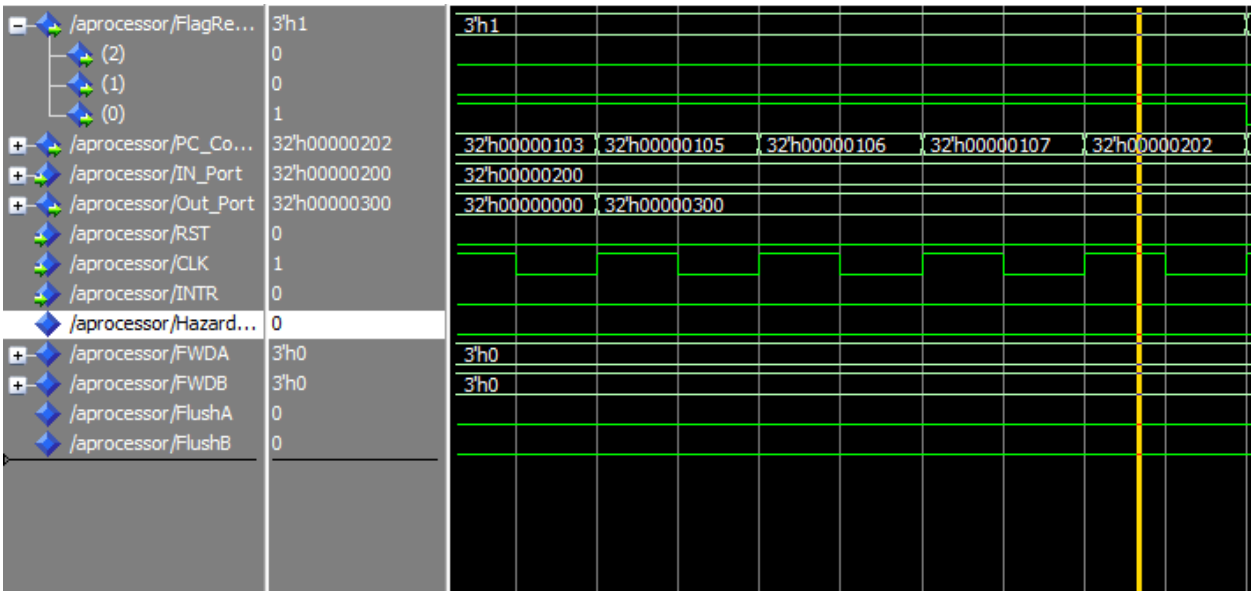
Clock cycle 42

Go to the INT x100



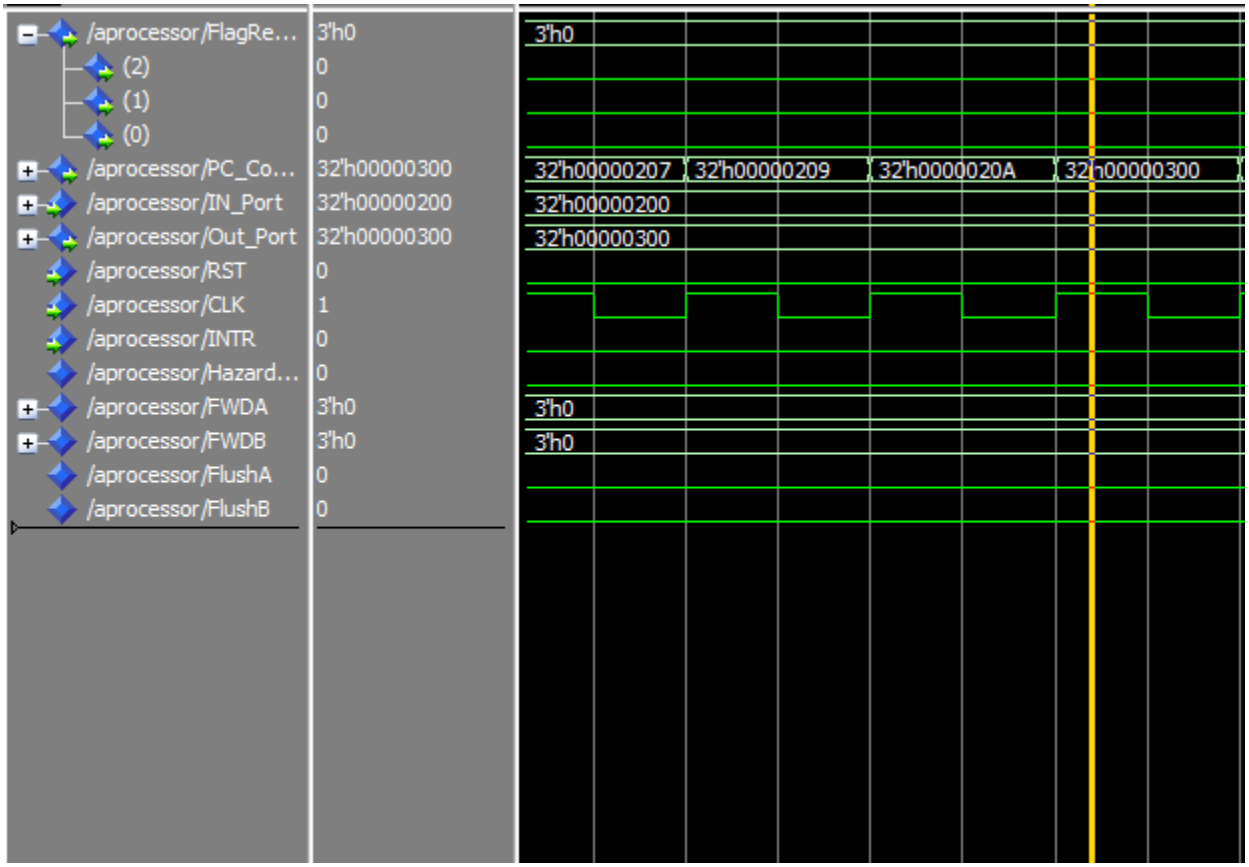
Clock cycle 49

Returned from the INT



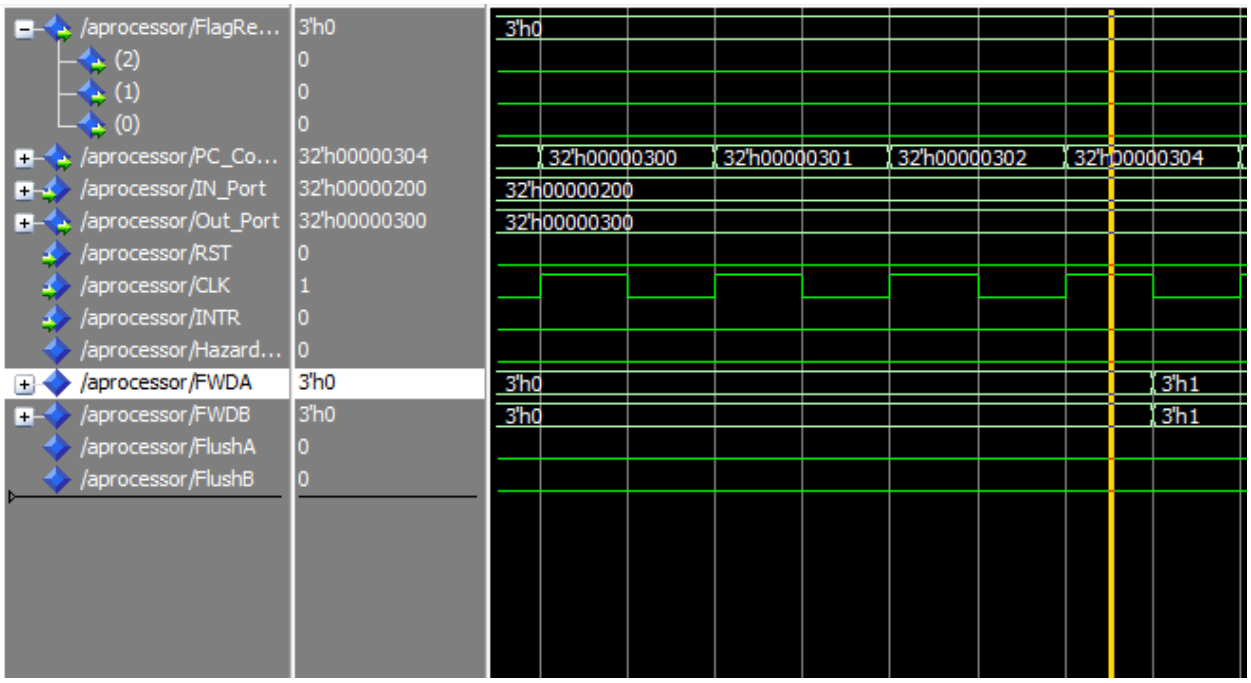
Clock cycle 56

Go to Call x300



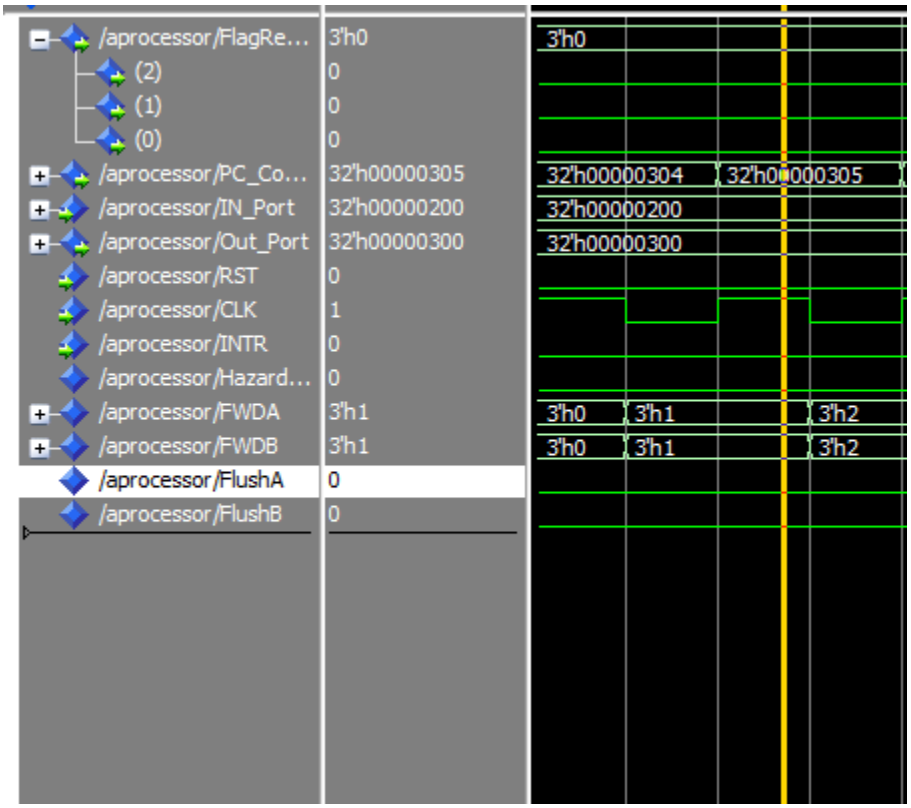
Clock cycle 59

R6 is updated with x400 and Flags are updated right



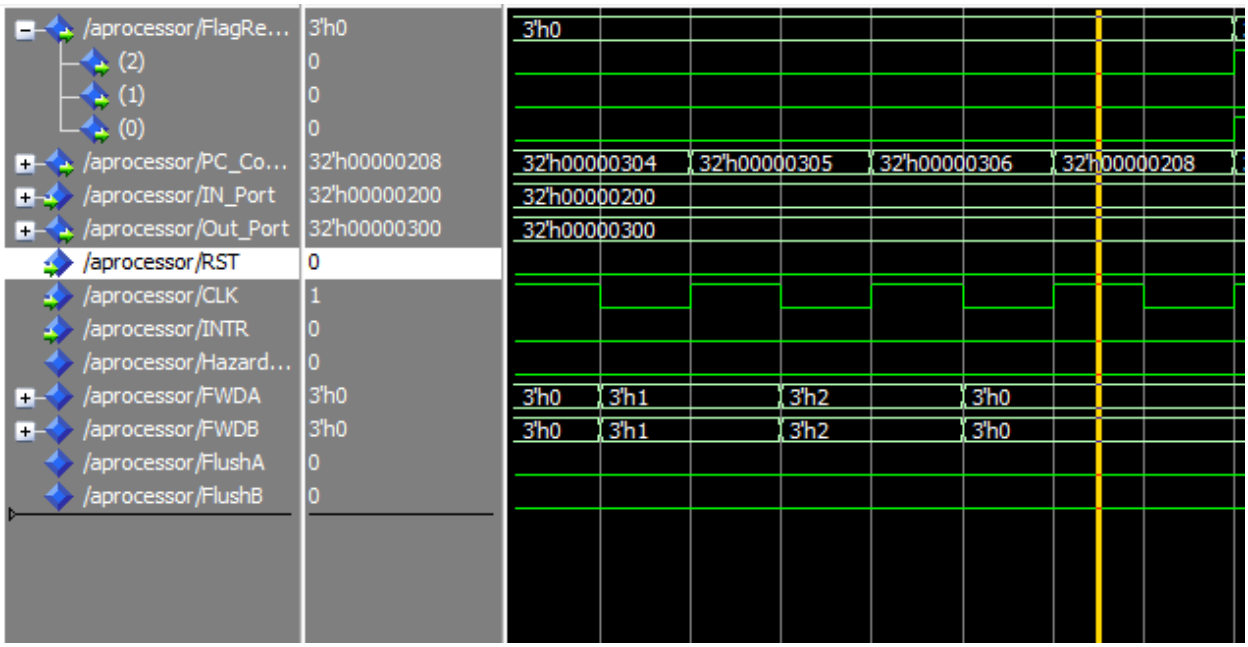
Clock cycle 60

R1 is updated with x80 and Flags are updated right



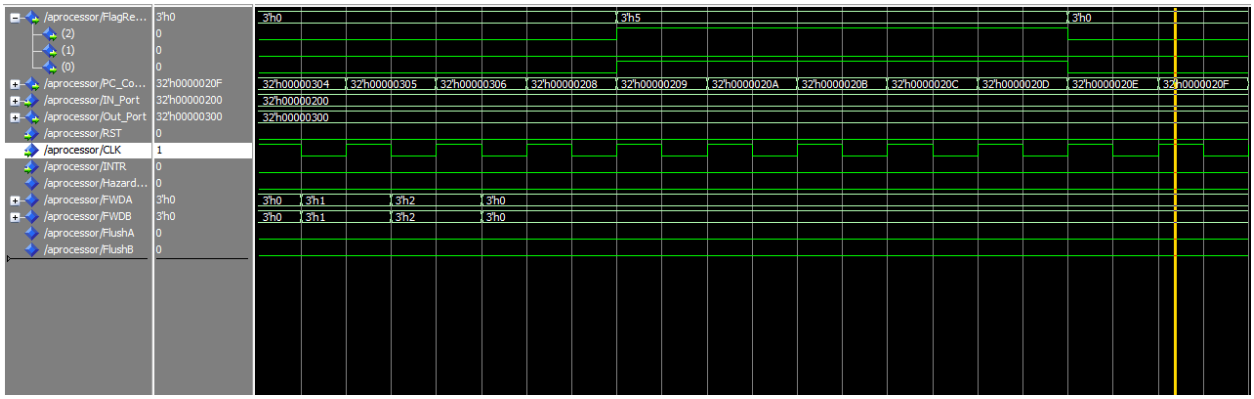
Clock cycle 62

Returned from the call



Clock cycle 69

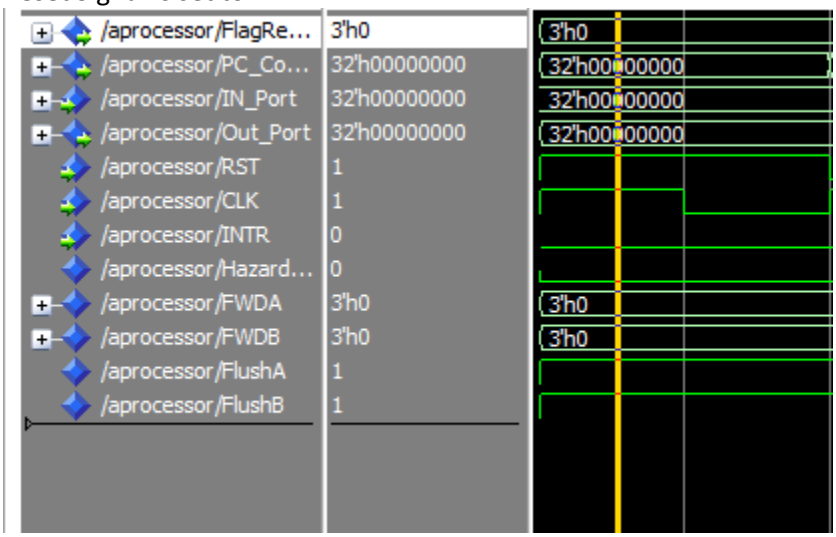
R6 is updated with x401 , Flags are updated right and the processor finished working



Full Processor

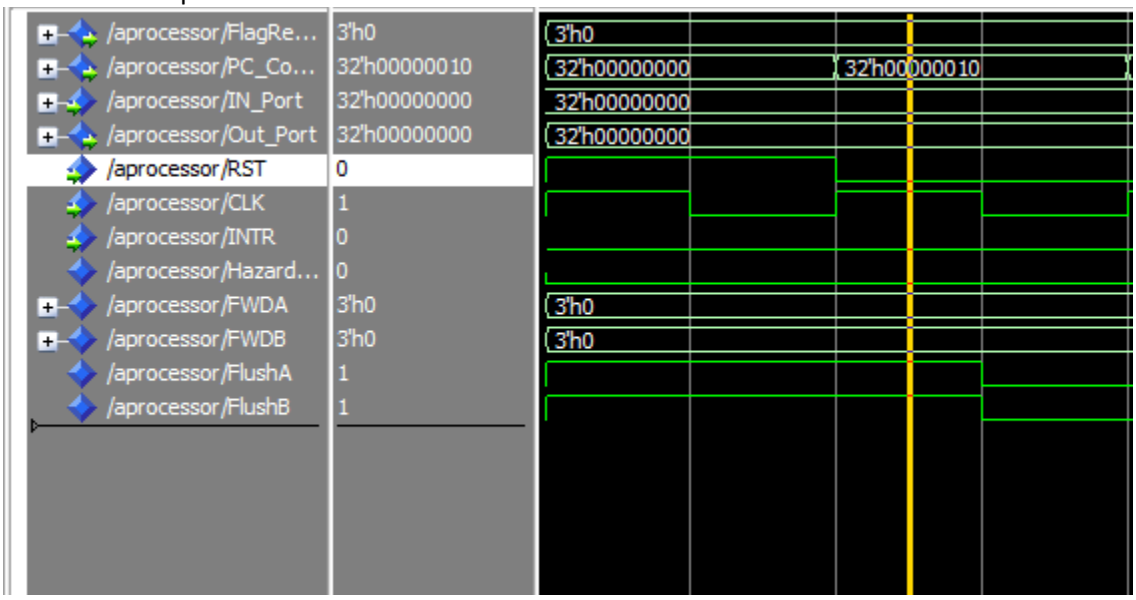
Clock cycle 1

Reset Signal is set to 1.



Clock cycle 2

PC counter is updated with the Reset Address.



Clock cycle 5

R1 is updated with the value x30

Clock cycle 6

R2 is updated with the value x50

Clock cycle 7

R3 is updated with the value x100

Clock cycle 8

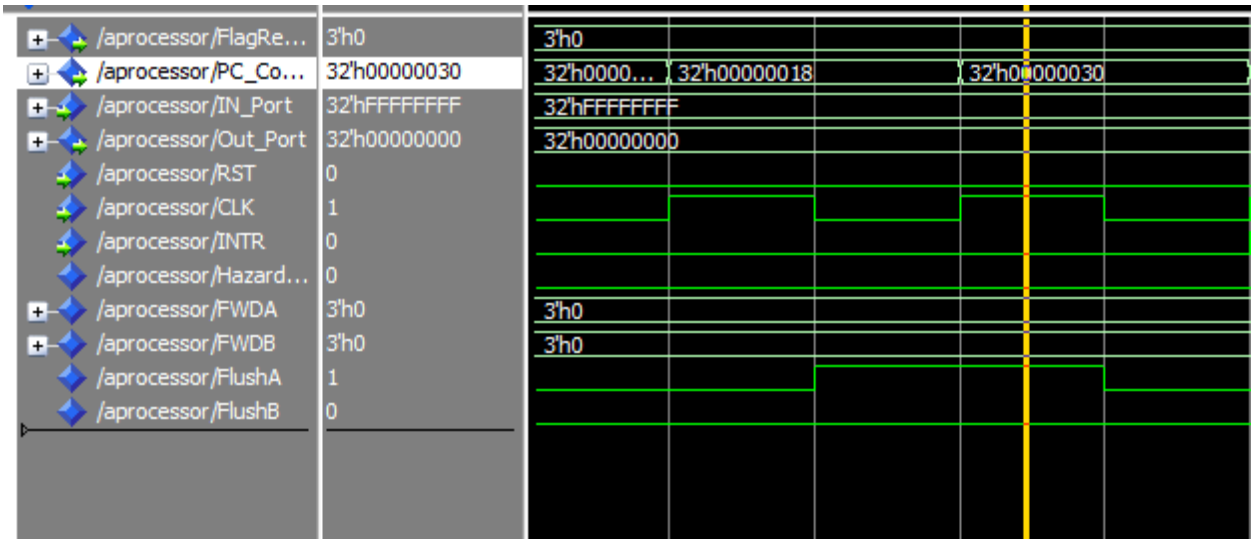
R4 is updated with the value x300

Clock cycle 9

R7 is updated with the value xFFFFFFFF

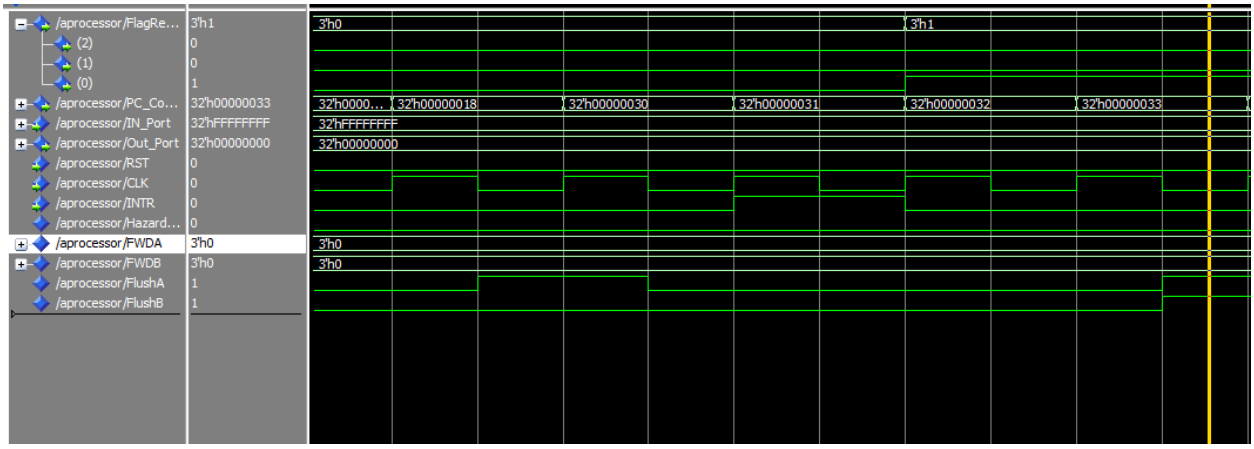
Clock cycle 11

Make the JMP and to x30



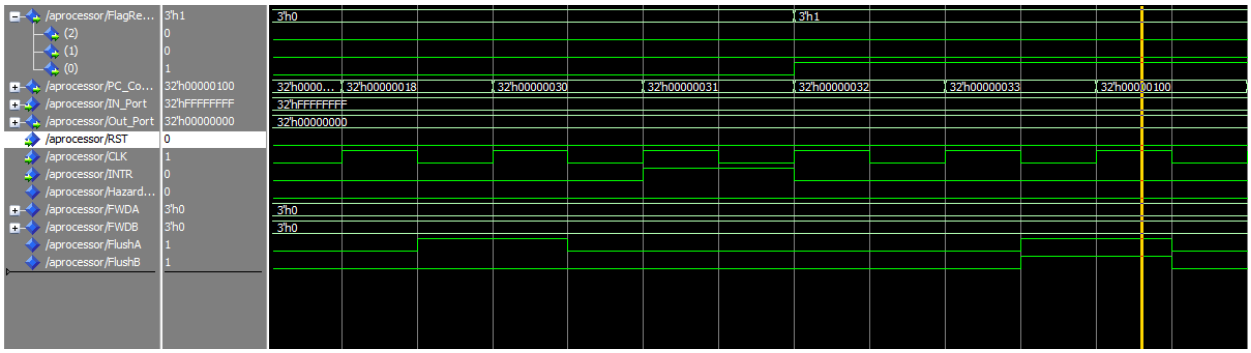
Clock cycle 14

Updated the Flags



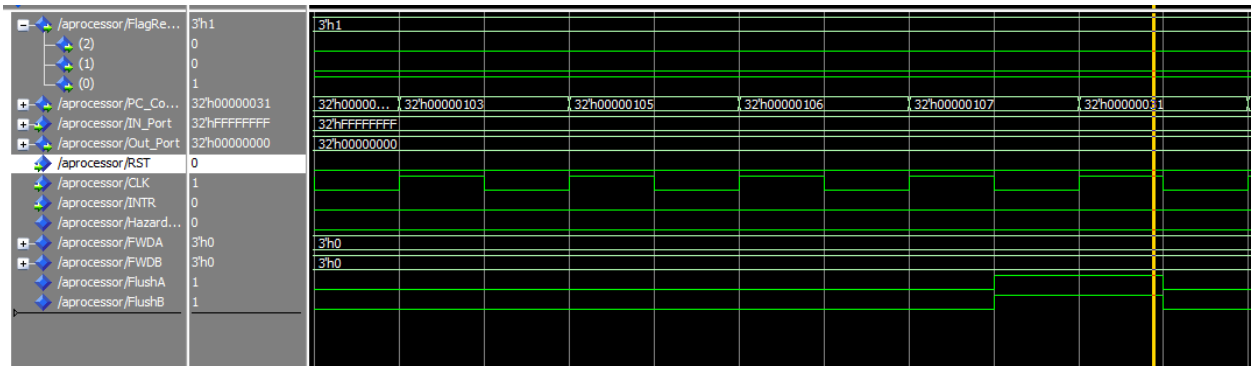
Clock cycle 15

Go to the INT x100



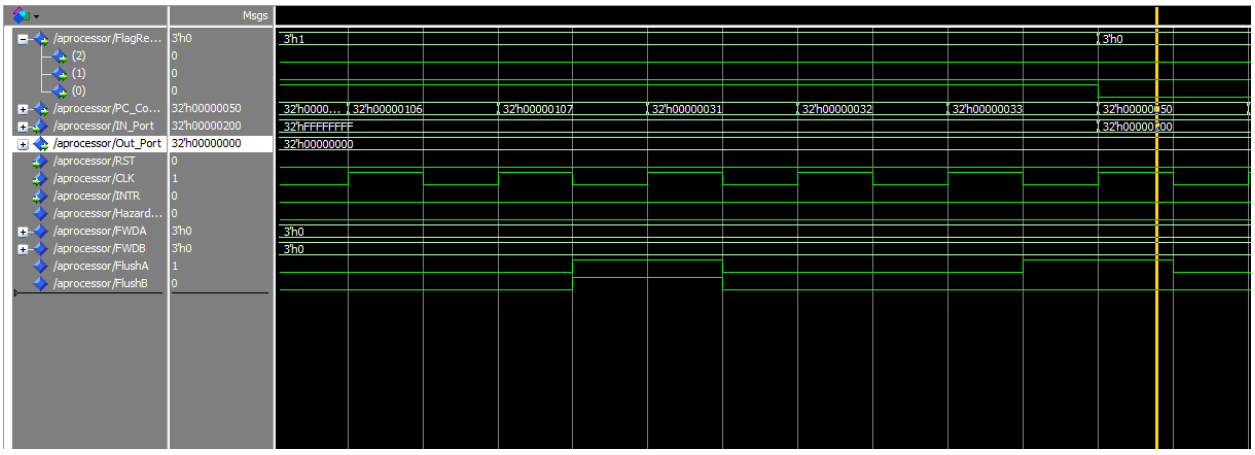
Clock cycle 22

Returned from the INT



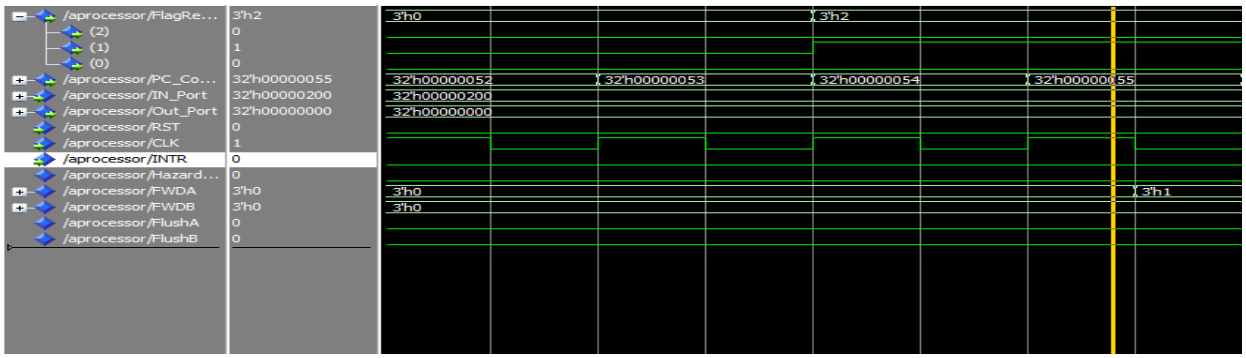
Clock cycle 25

Make the JZ and Set the Zero Flag to zero



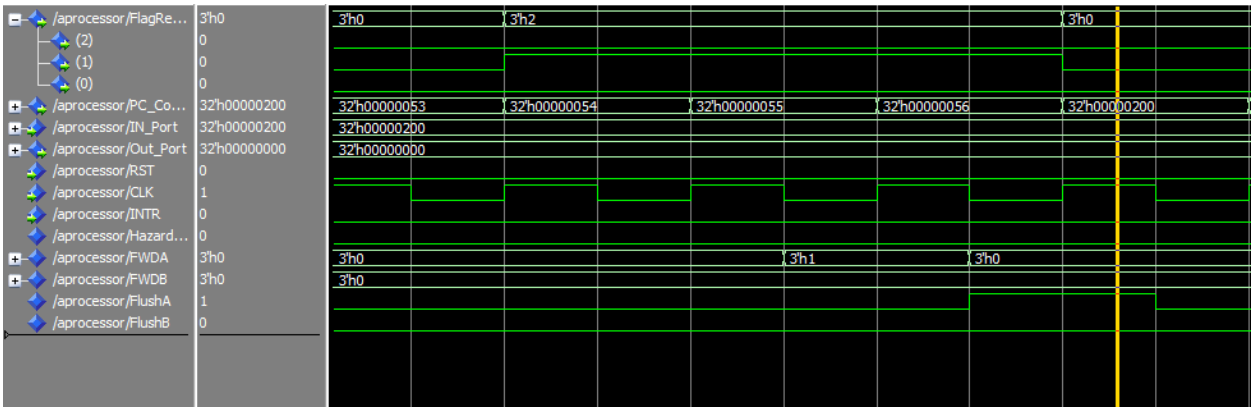
Clock cycle 30

R5 is updated with xFFFFFFFF and Neg Flag is Set to 1



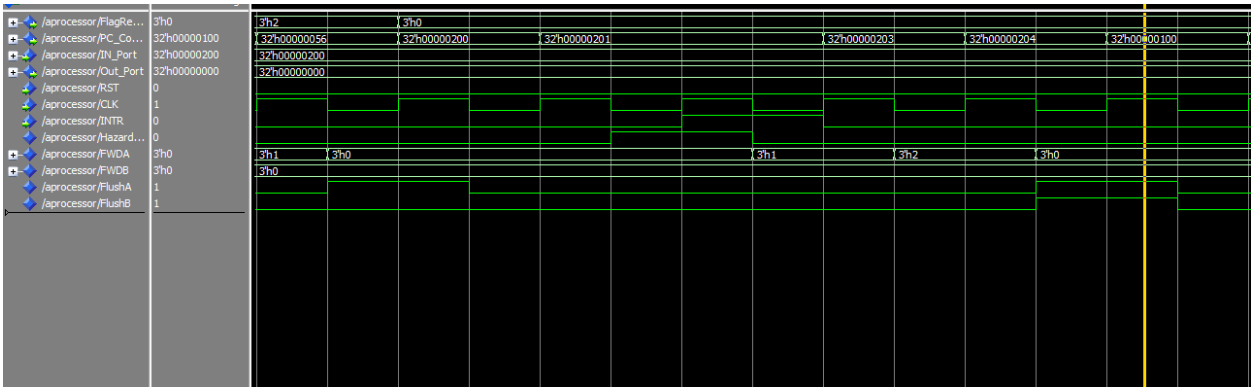
Clock cycle 32

Make the JN and Neg Flag is set to 0.



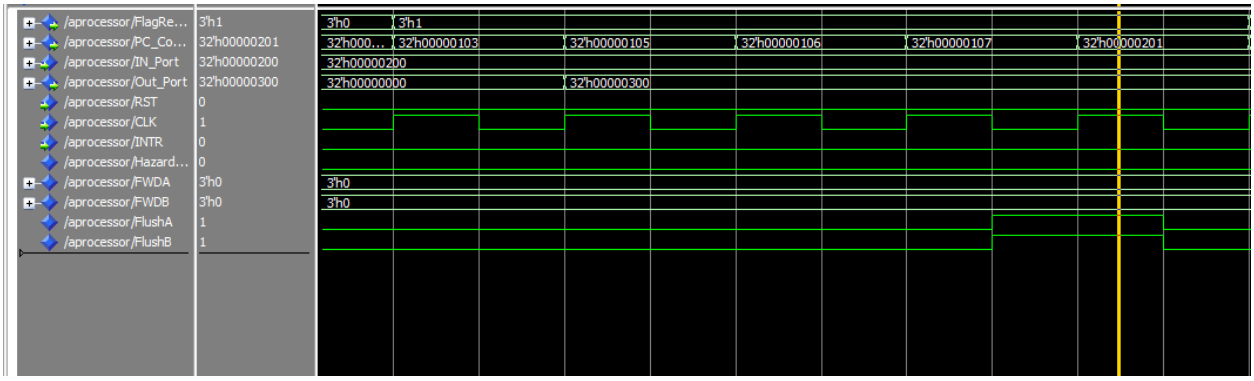
Clock cycle 37

Go to the INT x100



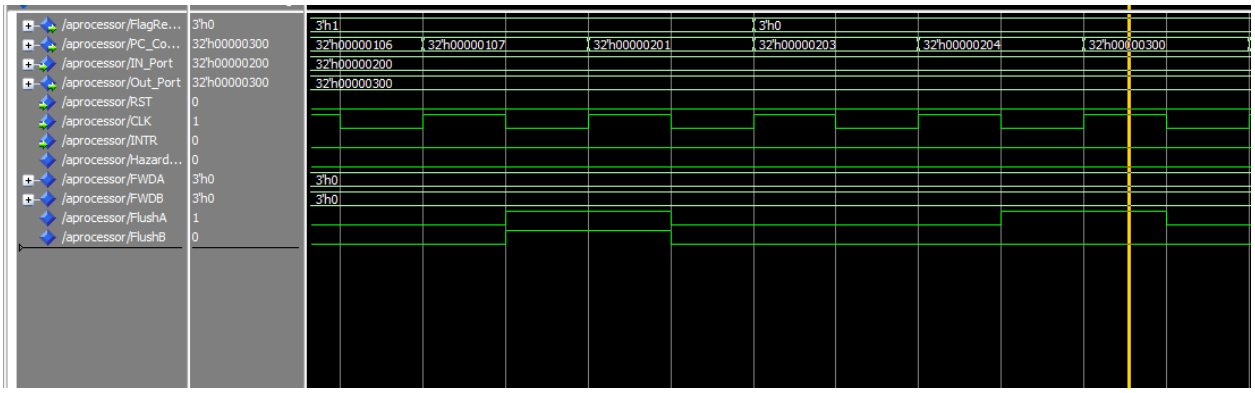
Clock cycle 44

Returned from the INT



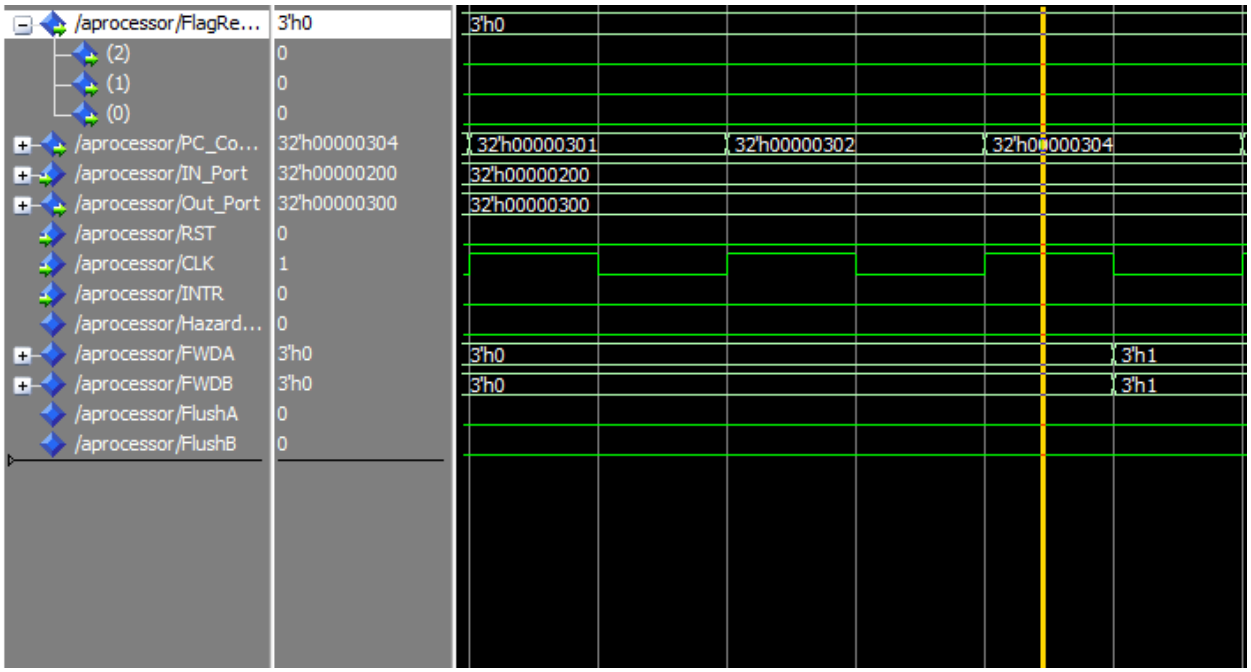
Clock cycle 47

Go to Call x300



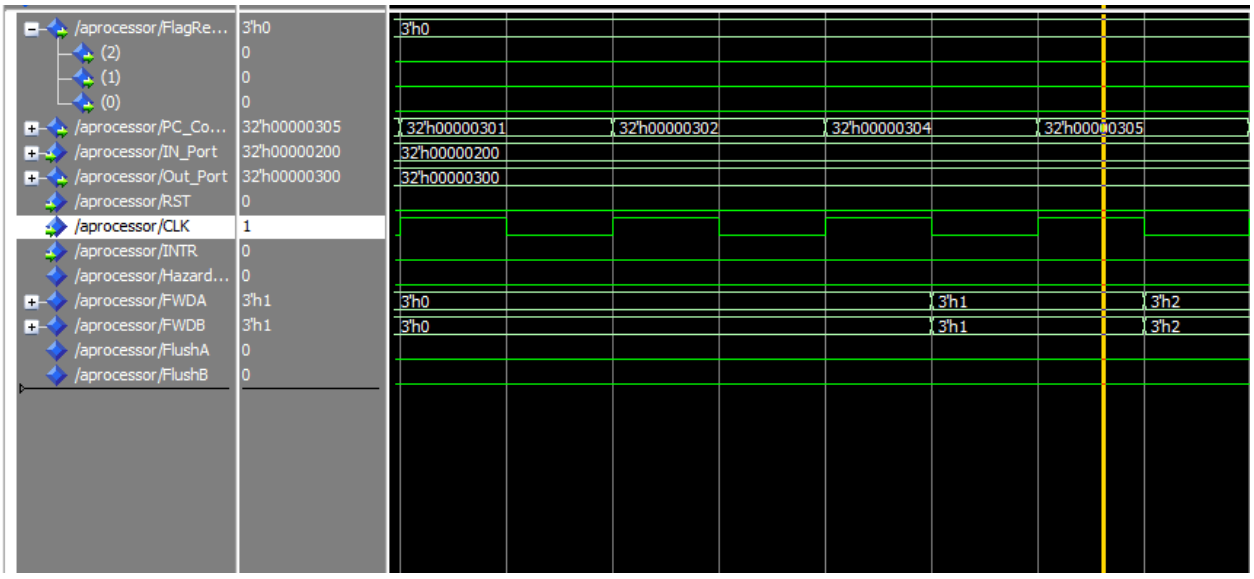
Clock cycle 50

R6 is updated with x400 and Flags are updated right



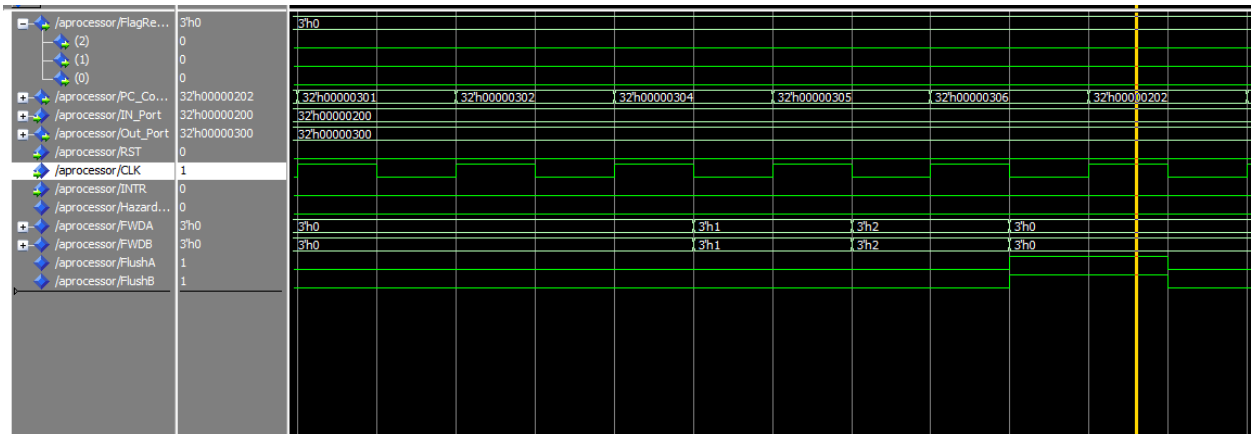
Clock cycle 51

R1 is updated with x80 and Flags are updated right



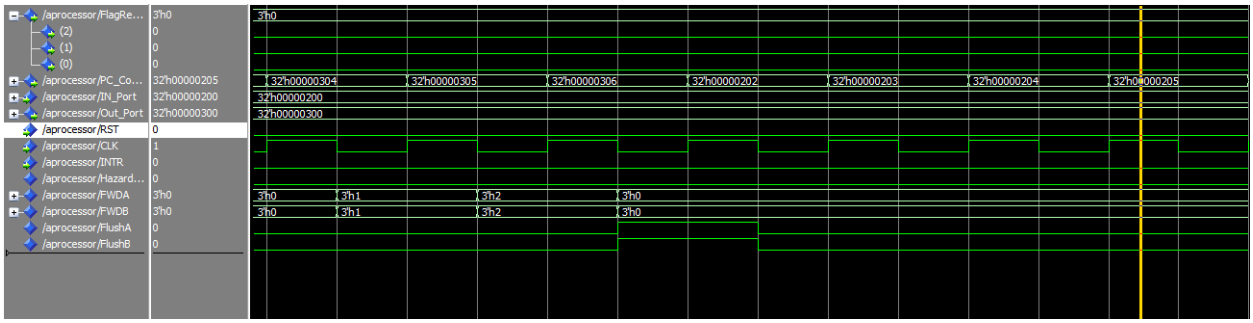
Clock cycle 53

Returned from the call



Clock cycle 56

R6 is updated with x401 , Flags are updated right and the processor finished working



❖ **Branch Prediction test cases:**

This part is documented using perfect cache but also you can find its do file and HEX file in the full processor with cache

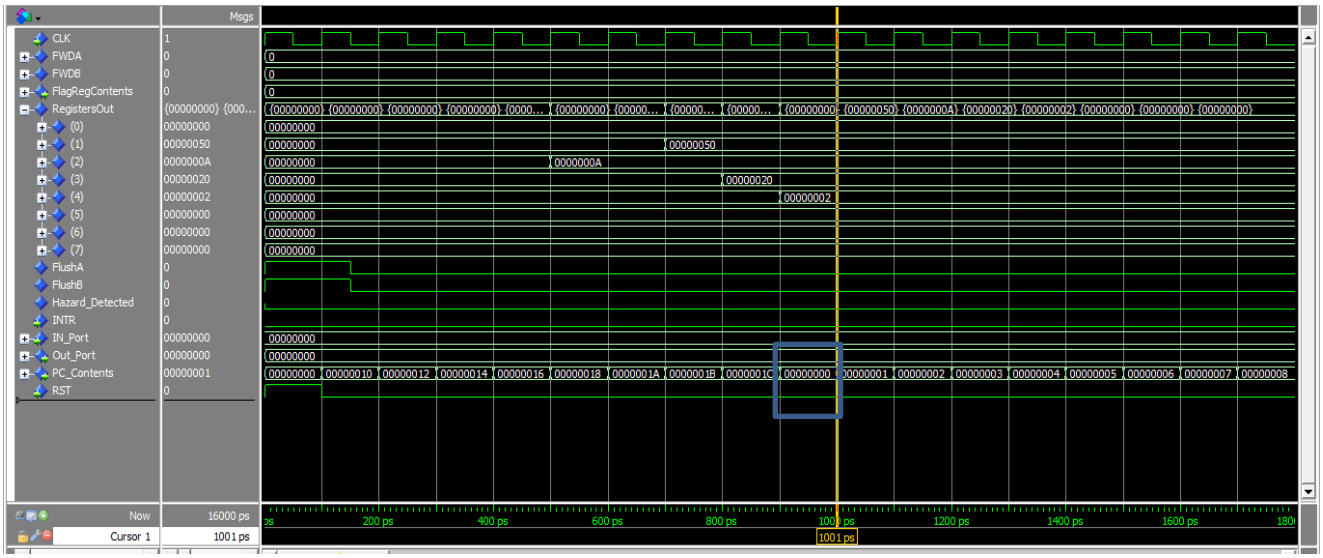
a) Processor with no hazard detection unit or forwarding unit or flushing

The processor will have data hazards due to the absence of forwarding unit as we can see in:

```
.ORG 10
LDM R2,0A #R2=0A
LDM R0,0  #R0=0
LDM R1,50 #R1=50
LDM R3,20 #R3=20
LDM R4,2  #R4=2
JMP R3    #Jump to 20—will jump to wrong location.
.ORG 20
SUB R0,R2,R5 #check if R0 = R2
JZ R1 #jump if R0=R2 to 50
ADD R4,R4,R4 #R4 = R4*2
OUT R4
INC R0
JMP R3 #jump to 20
.ORG 50
LDM R0,0 #R0=0
LDM R2,8 #R2=8
LDM R3,60 #R3=60
LDM R4,3  #R4=3
JMP R3 #jump to 60 –Will jump to a wrong location.
.ORG 60
ADD R4,R4,R4 #R4 = R4*2
OUT R4
INC R0
AND R0,R2,R5 #when R0 < R2(8) answer will be zero
JZ R3 #jump if R0 < R2 to 60
INC R4
OUT R4
```

And some instructions are not meant to be executed will be executed because there is no flushing

And the following are screen shots to show you what happened:



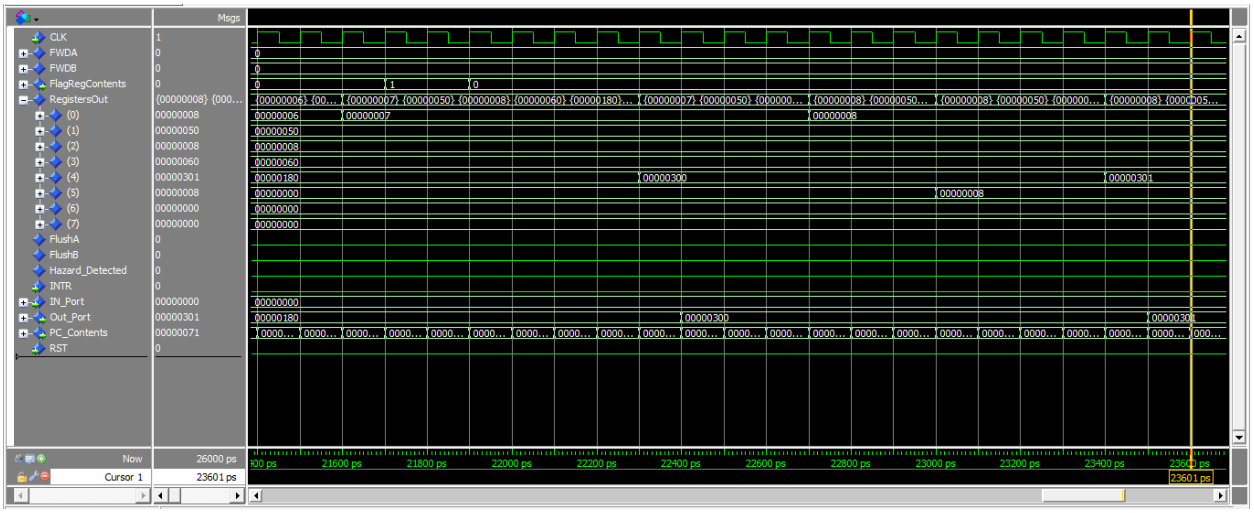
As we can see it has jumped to a wrong location

And this can be solved by adding the following lines:

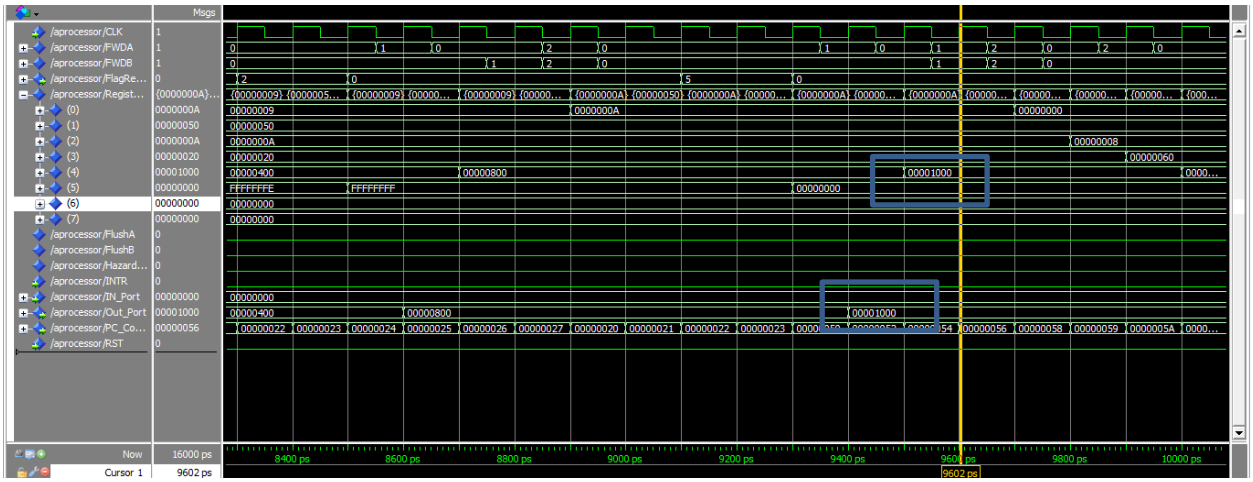
```
.ORG 10
LDM R2,0A #R2=0A
LDM R0,0  #R0=0
LDM R1,50 #R1=50
LDM R3,20 #R3=20
LDM R4,2  #R4=2
NOP
JMP R3    #Jump to 20
.ORG 20
SUB R0,R2,R5 #check if R0 = R2
JZ R1 #jump if R0=R2 to 50
NOP
NOP
ADD R4,R4,R4 #R4 = R4*2
NOP
NOP
OUT R4
INC R0
JMP R3 #jump to 20
NOP
NOP
.ORG 50
LDM R0,0 #R0=0
LDM R2,8 #R2=8
LDM R3,60 #R3=60
LDM R4,3  #R4=3
NOP
```

```
JMP R3 #jump to 60
NOP
NOP
.ORG 60
ADD R4,R4,R4 #R4 = R4*2
NOP
NOP
OUT R4
INC R0
NOP
NOP
AND R0,R2,R5 #when R0 < R2(8) answer will be zero
JZ R3 #jump if R0 < R2 to 60
NOP
NOP
INC R4
NOP
NOP
OUT R4
```

And by executing it we have found the output correct but it has taken 236 cycle.



b) By adding forwarding Unit the data hazards will be solved but the instructions that shouldn't be executed will still happen as shown.



Here Add R4, R4, R4 and Out R4 shouldn't be executed

And this can be solved by adding the following Lines

```
.ORG 10
LDM R2,0A #R2=0A
LDM R0,0  #R0=0
LDM R1,50 #R1=50
LDM R3,20 #R3=20
LDM R4,2  #R4=2
JMP R3    #Jump to 20
.ORG 20
SUB R0,R2,R5 #check if R0 = R2
JZ R1 #jump if R0=R2 to 50
NOP
NOP
ADD R4,R4,R4 #R4 = R4*2
OUT R4
INC R0
JMP R3 #jump to 20
NOP
NOP
.ORG 50
LDM R0,0 #R0=0
LDM R2,8 #R2=8
LDM R3,60 #R3=60
LDM R4,3  #R4=3
JMP R3 #jump to 60
NOP
NOP
.ORG 60
ADD R4,R4,R4 #R4 = R4*2
OUT R4
INC R0
AND R0,R2,R5 #when R0 < R2(8) answer will be zero
JZ R3 #jump if R0 < R2 to 60
```

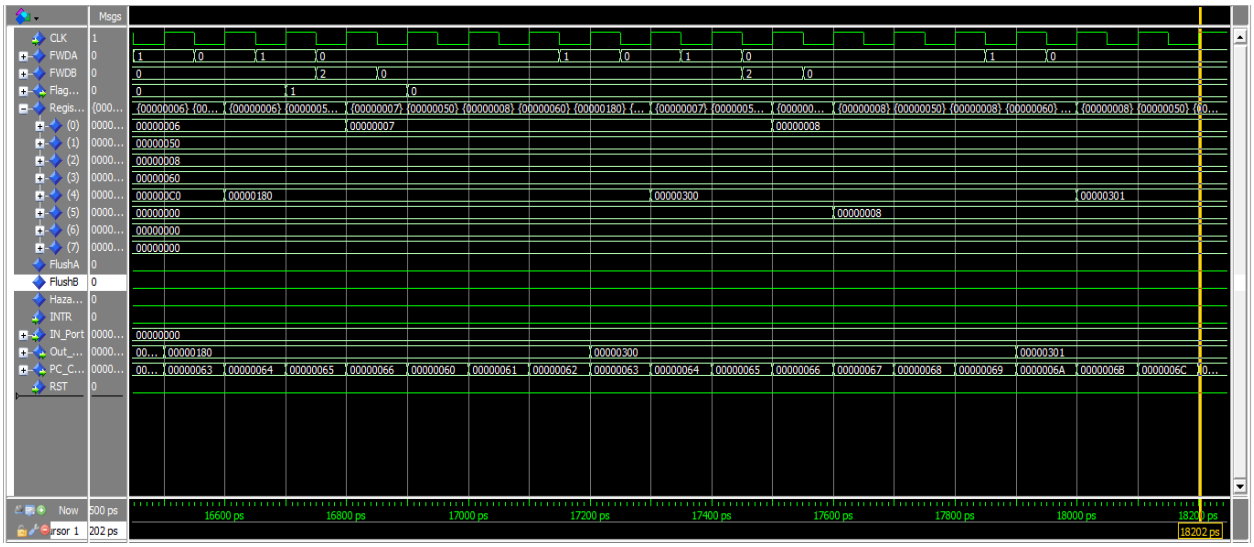
NOP

NOP

INC R4

OUT R4

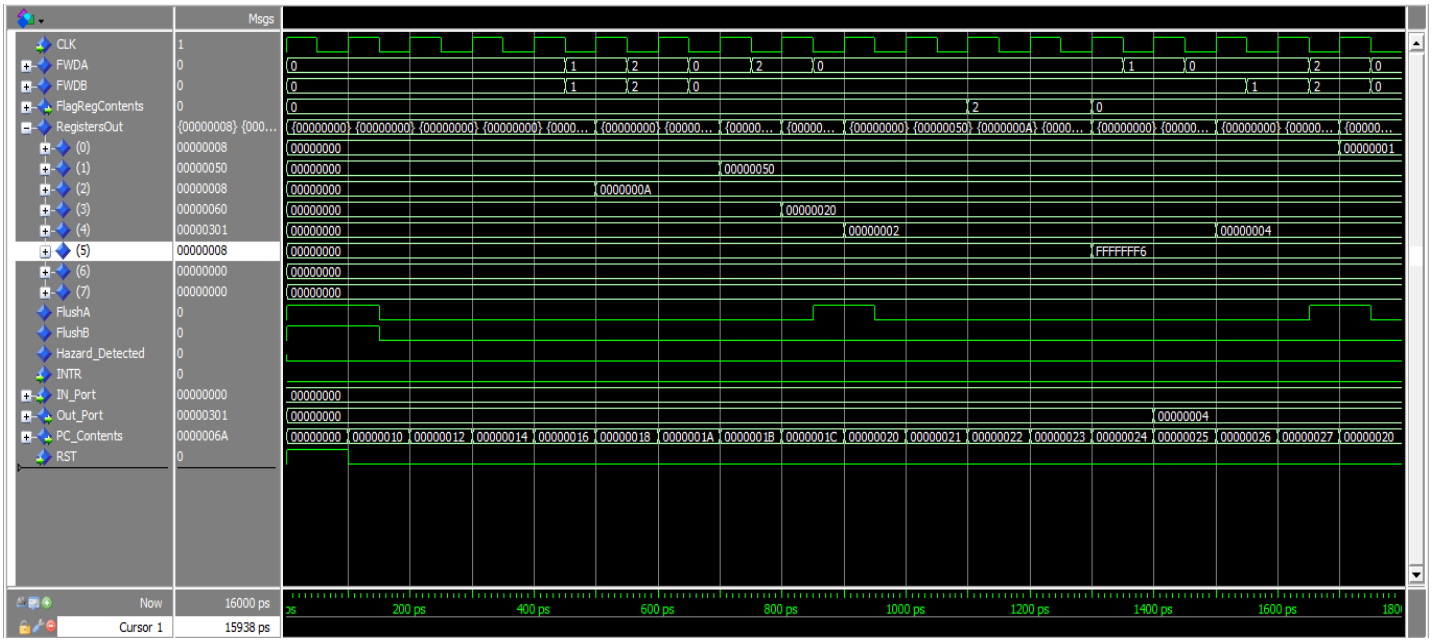
And by executing it we have found the output correct but it has taken 182 cycle.



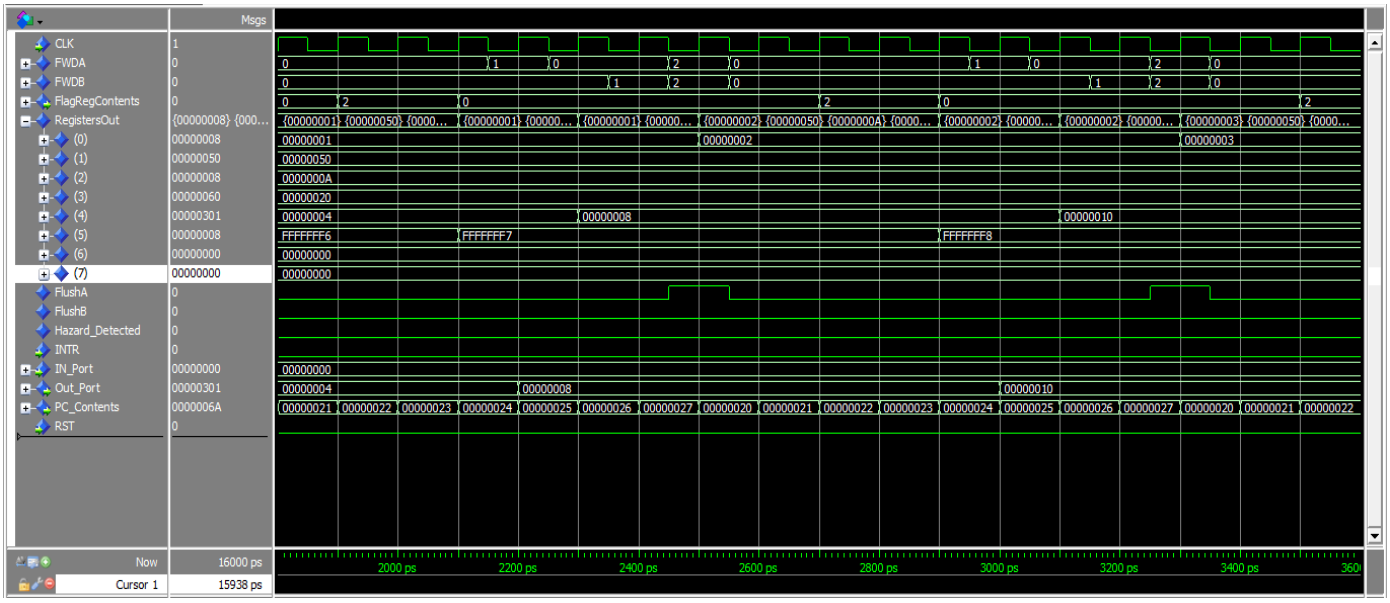
c) By adding hazard detection unit nothing will change as there is no load-use cases

d) Perfectly working processor screen shots:

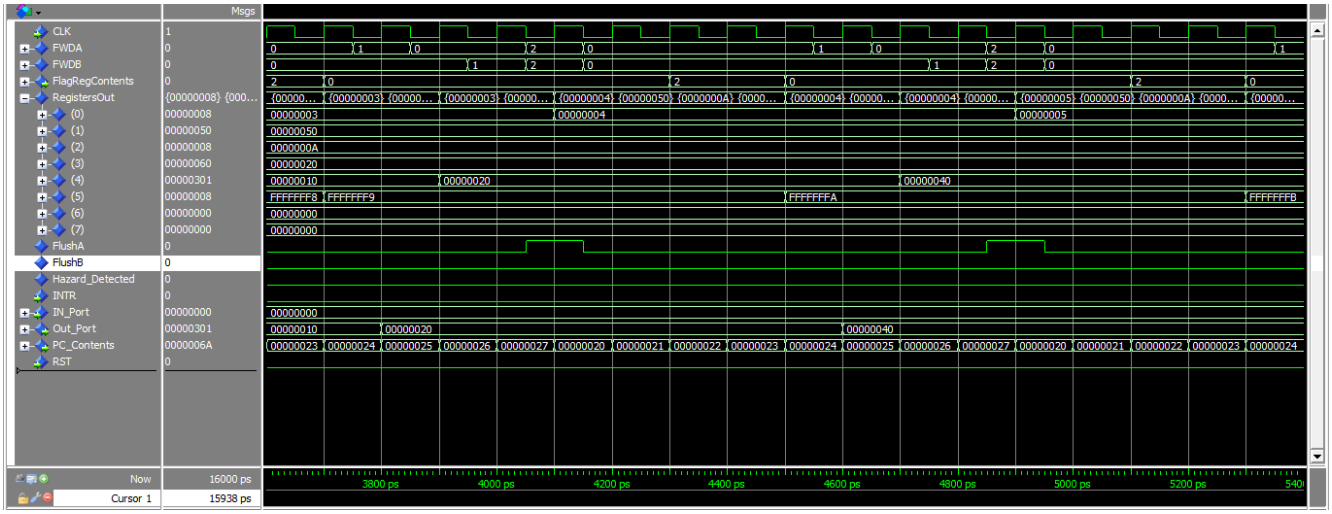
First 18 cycles:



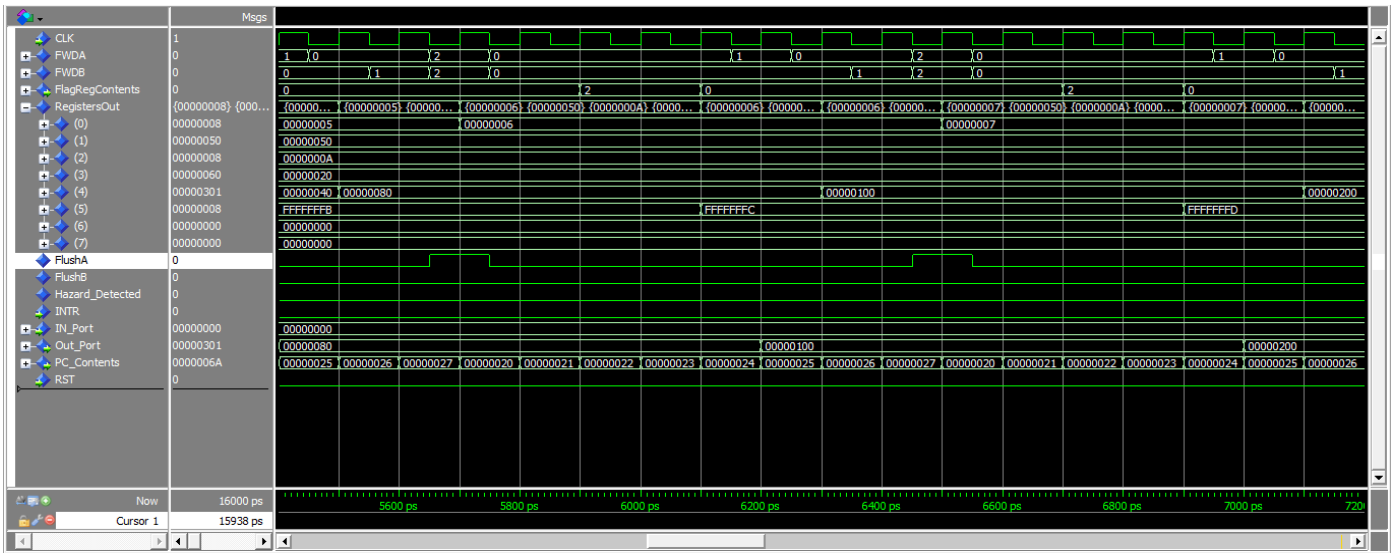
From cycle 18 to 36:



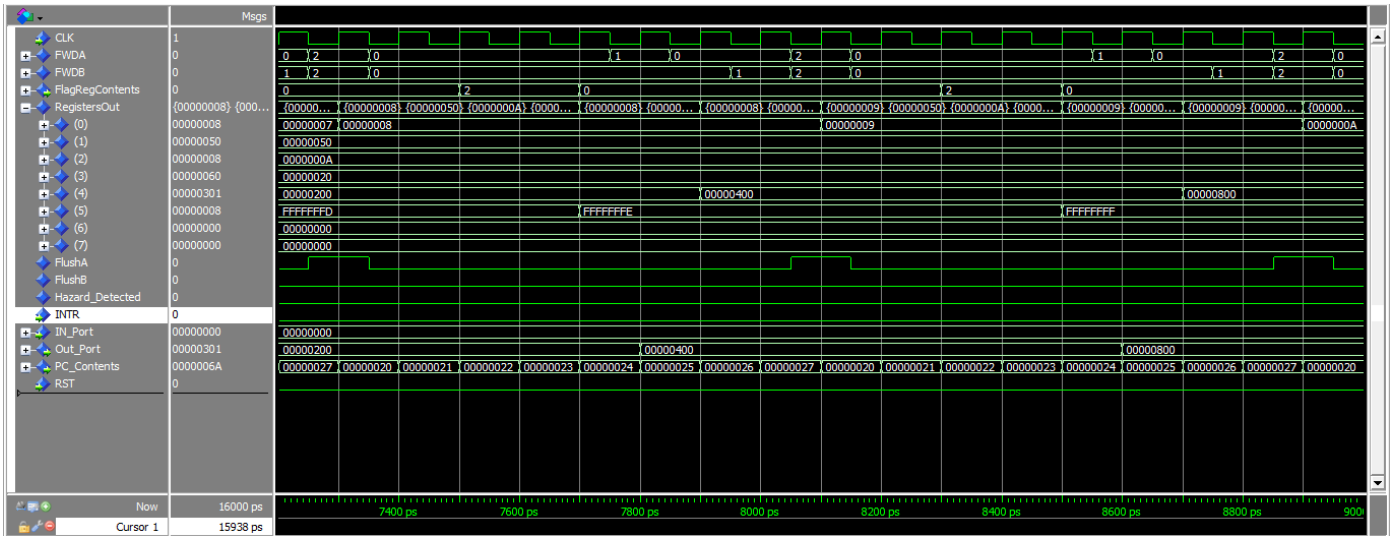
From cycle 36 to 54:



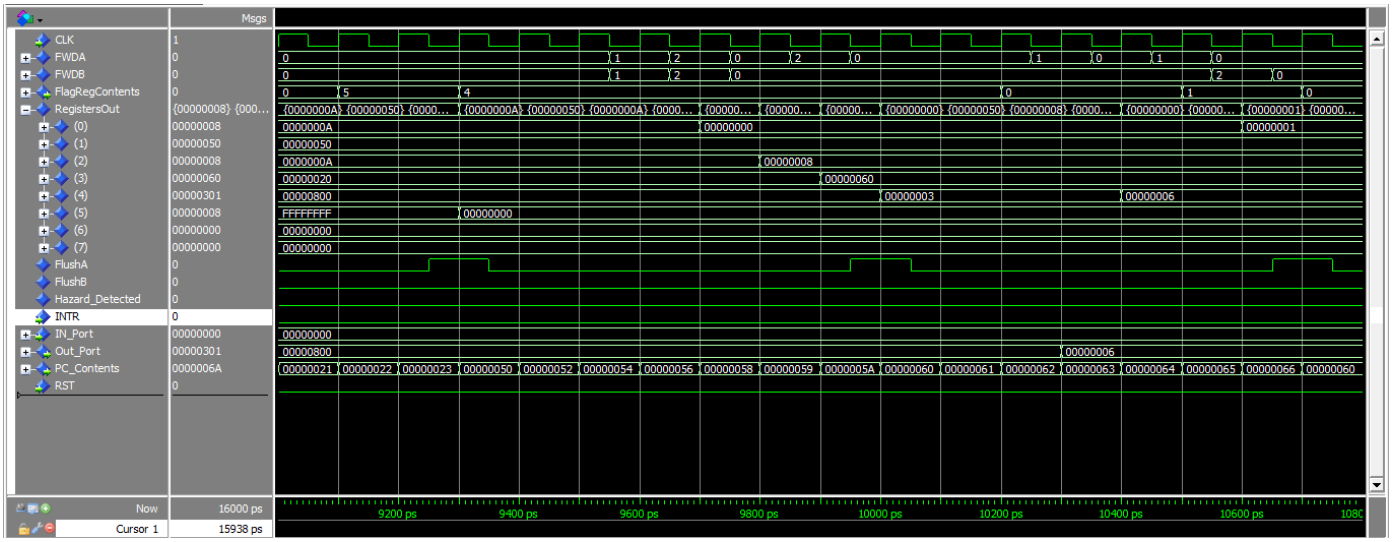
From cycle 54 to 72



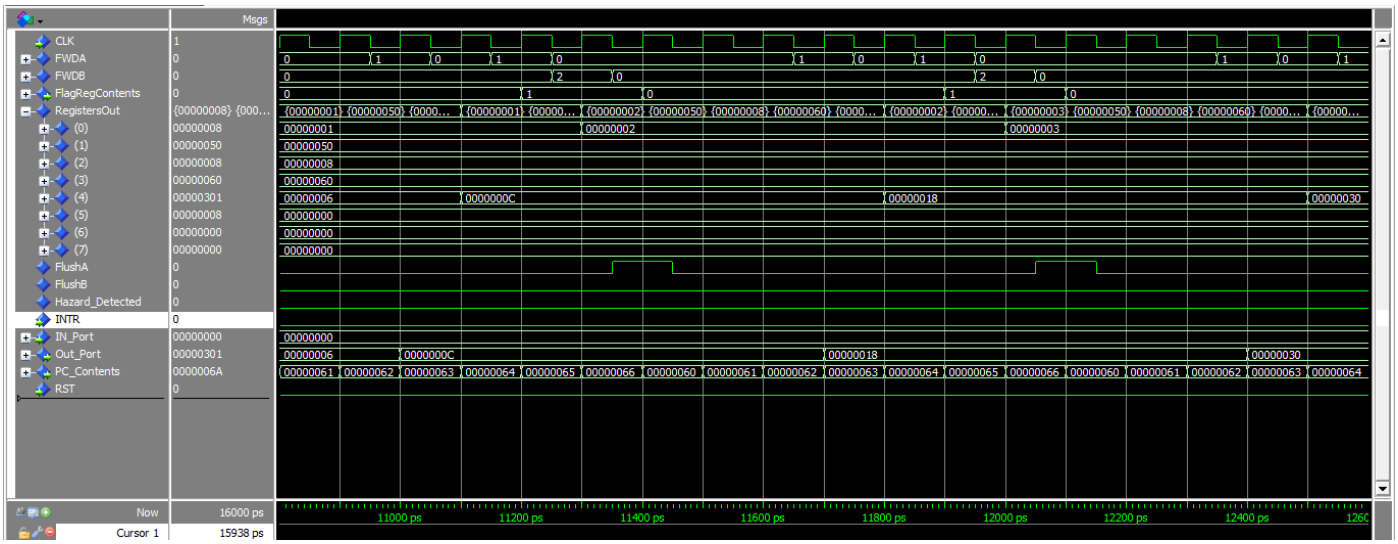
From cycle 72 to 90:



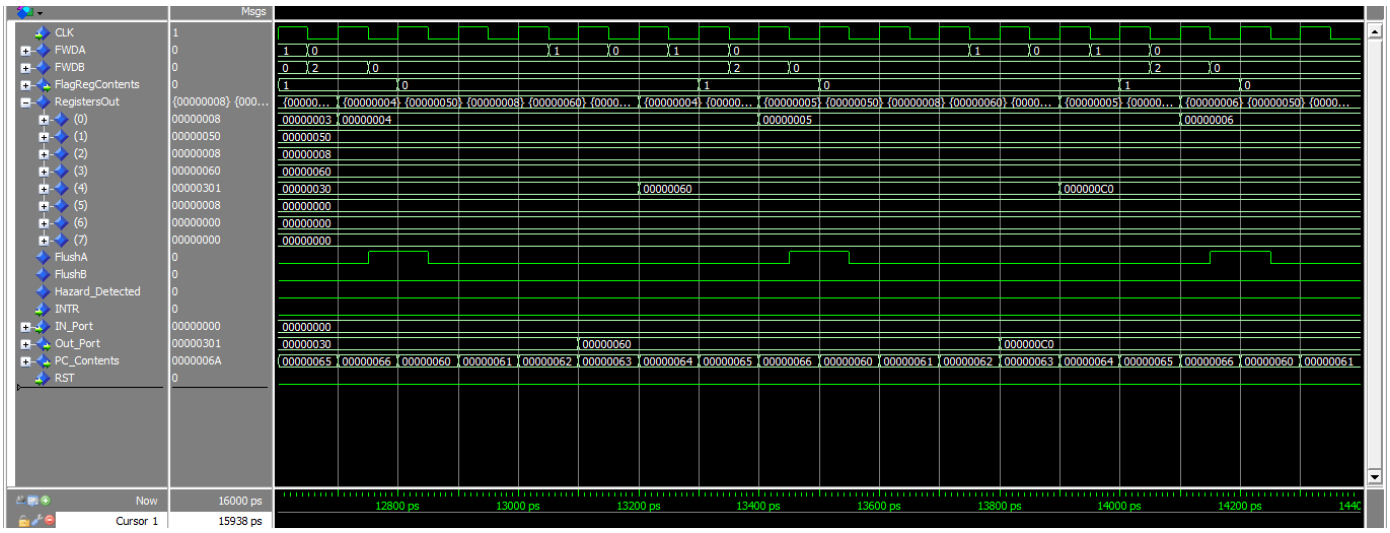
From Cycle 90 to 108:



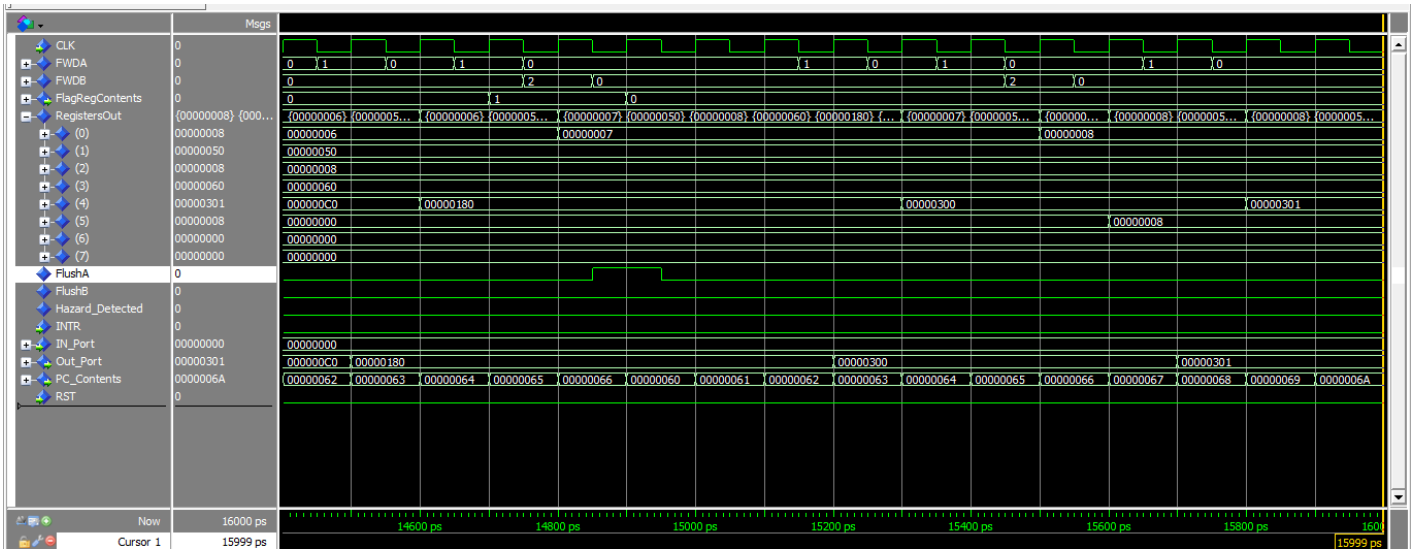
From cycle 108 to 126:



From cycle 126 to 144:



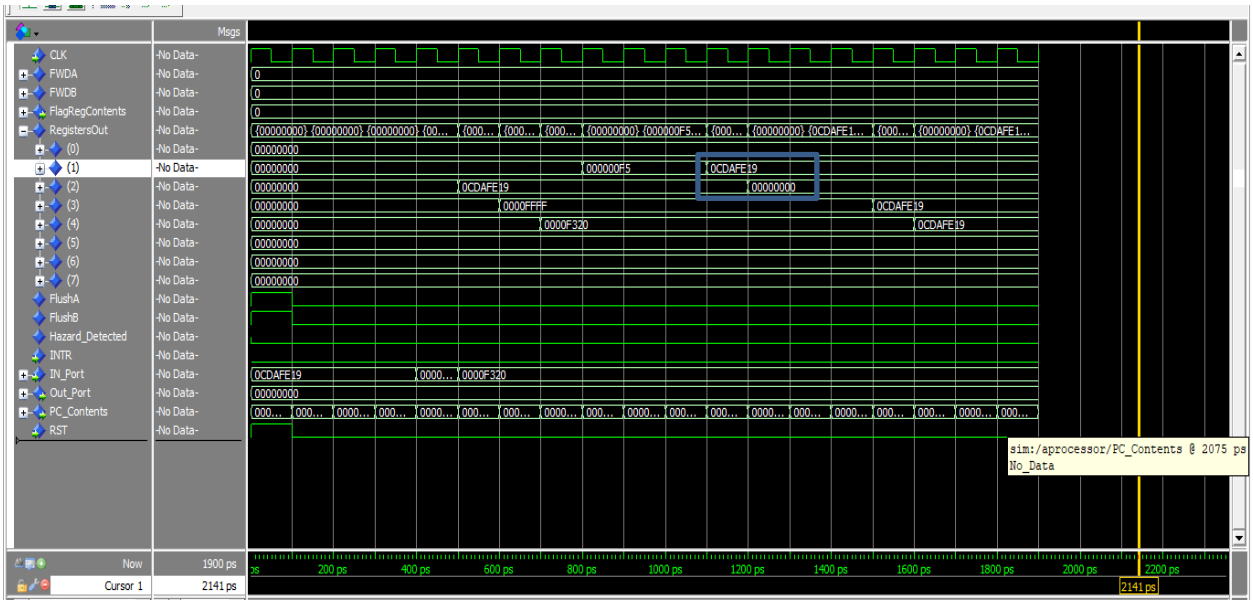
From 144 to 160:

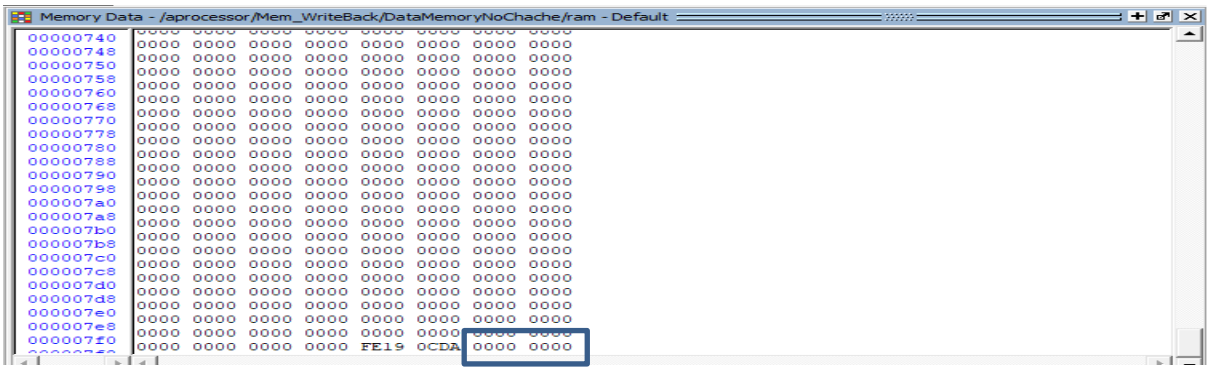


❖ **Memory test cases:**

This part is documented using perfect cache but also you can find its do file and HEX file in the full processor with cache

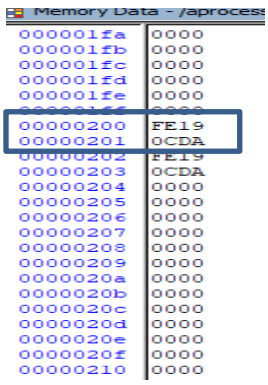
2) With no hazard detection unit ,Forwarding or Flushing:





As we can see the old value of R1 was pushed.

So the values popped were wrong also.



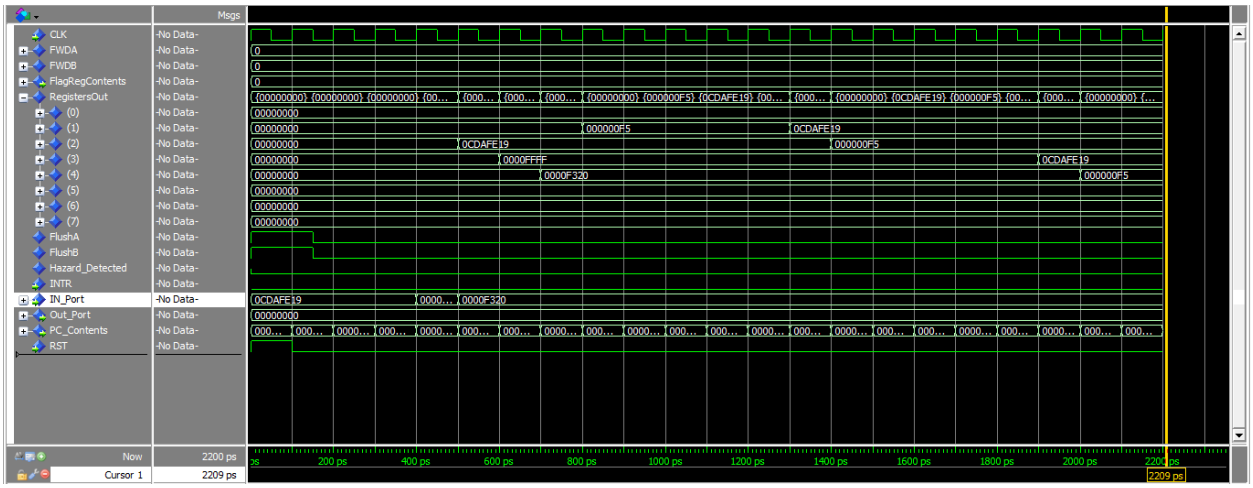
And as we can see it has stored the old value of R2 (before the pop)

We can solve this by adding NOP's as follow:

```
in R2      #R2=0CDAFE19 add 0CDAFE19 in R2
in R3      #R3=FFFF
```

```
in R4       #R4=F320
LDM R1,F5   #R1=F5
NOP
NOP
PUSH R1     #SP=7FC, M[7FE, 7FF] = F5
PUSH R2     #SP=7FA,M[7FC, 7FD]=0CDAFE19
POP R1      #SP=7FC,R1=0CDAFE19
POP R2      #SP=7FE,R2=F5
NOP
NOP
STD R2,200  #M[200, 201]=F5
STD R1,202  #M[202, 203]=0CDAFE19
LDD R3,202  #R3=0CDAFE19
LDD R4,200  #R4=5
```

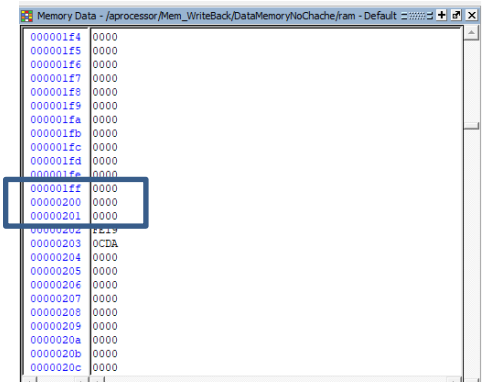
And then the waveform will be:

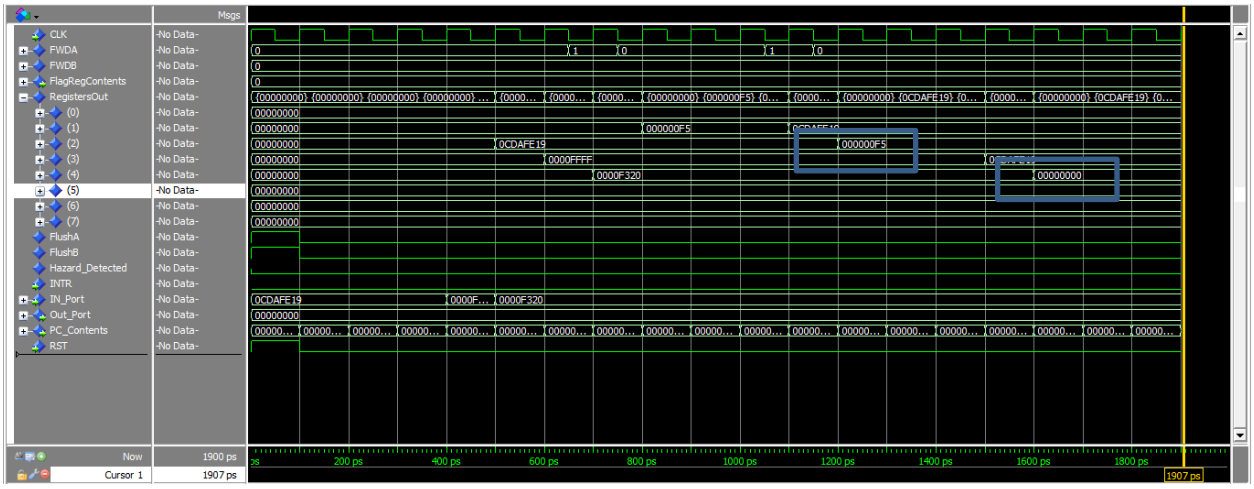


This shows us that the processor is working correctly but with 22 cycles not 19 cycles as with the full processor.

3) With forwarding unit:

Load-Use case problem will arise as it will take the value before correctly loading it from memory for example:





As we can see the old value of R2 was stored not the popped one.

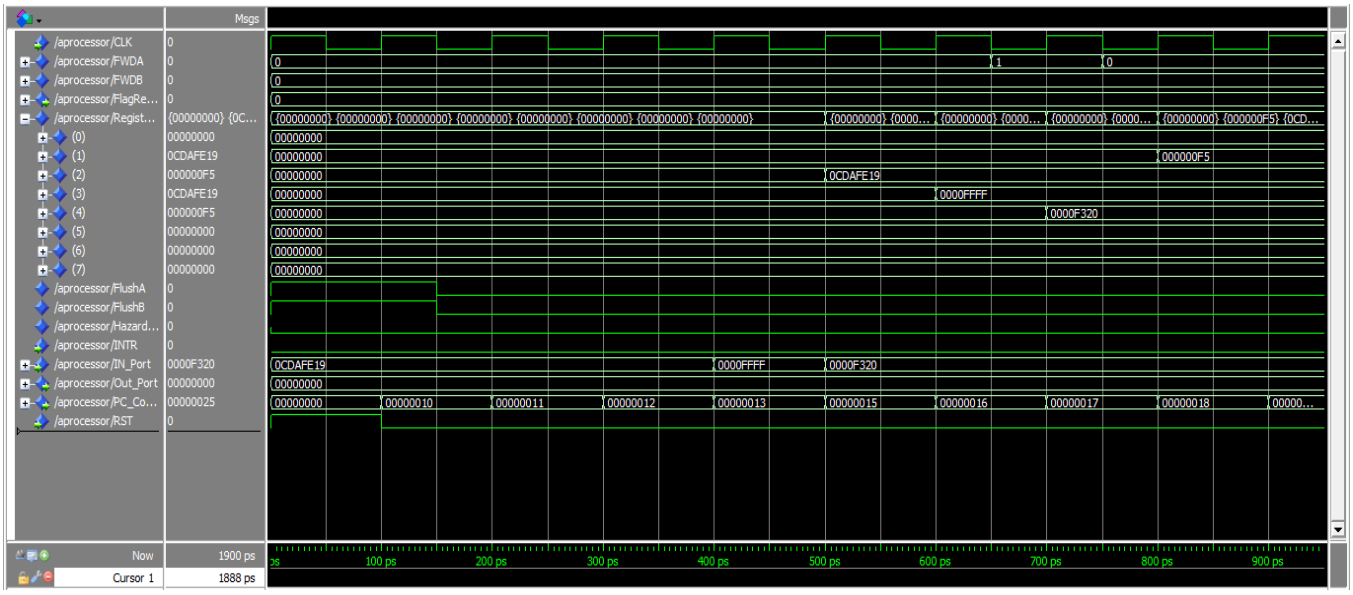
So in order to solve it we have used the following code which have gave us the same result as with full processor and with the same number of cycles.

```
.ORG 10
in R2       #R2=0CDAFE19 add 0CDAFE19 in R2
in R3       #R3=FFFF
in R4       #R4=F320
LDM R1,F5   #R1=F5
PUSH R1     #SP=7FC, M[7FE, 7FF] = F5
PUSH R2     #SP=7FA,M[7FC, 7FD]=0CDAFE19
POP R1      #SP=7FC,R1=0CDAFE19
POP R2      #SP=7FE,R2=F5
NOP
STD R2,200  #M[200, 201]=F5
STD R1,202  #M[202, 203]=0CDAFE19
LDD R3,202  #R3=0CDAFE19
LDD R4,200  #R4=5
```

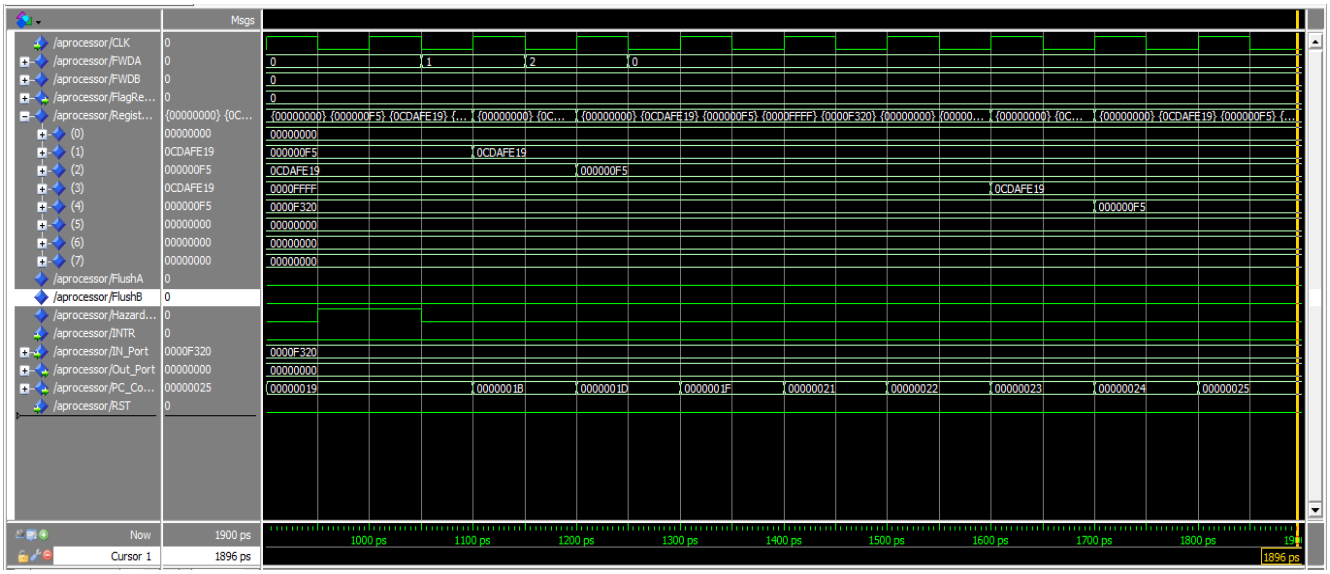
Reordering solution here will give us more performance but as we have mentioned that we only discuss the solutions with hardware not software.

4) Perfectly working processor screen shots (same as with no flushing as there are no branch instructions):

First 9 cycles:

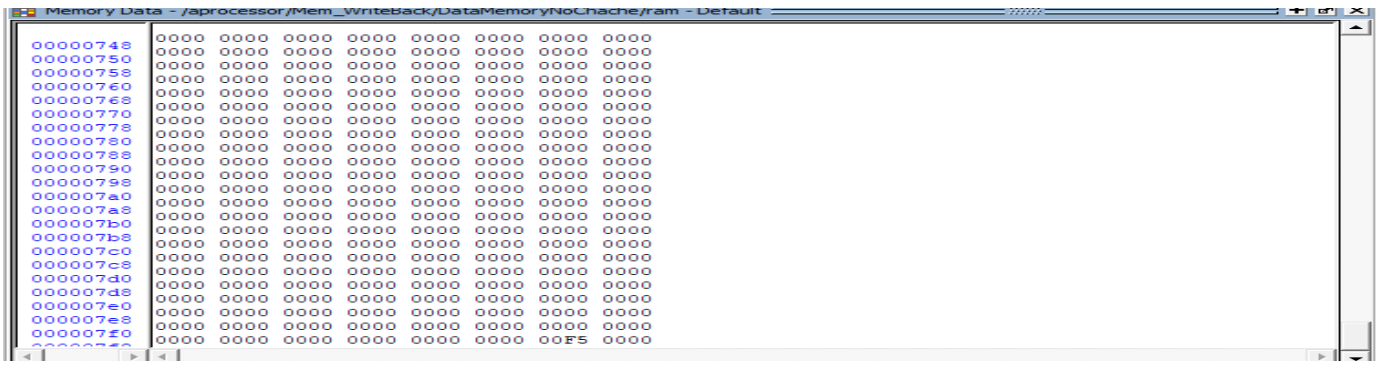


From cycle 9 to cycle 19:

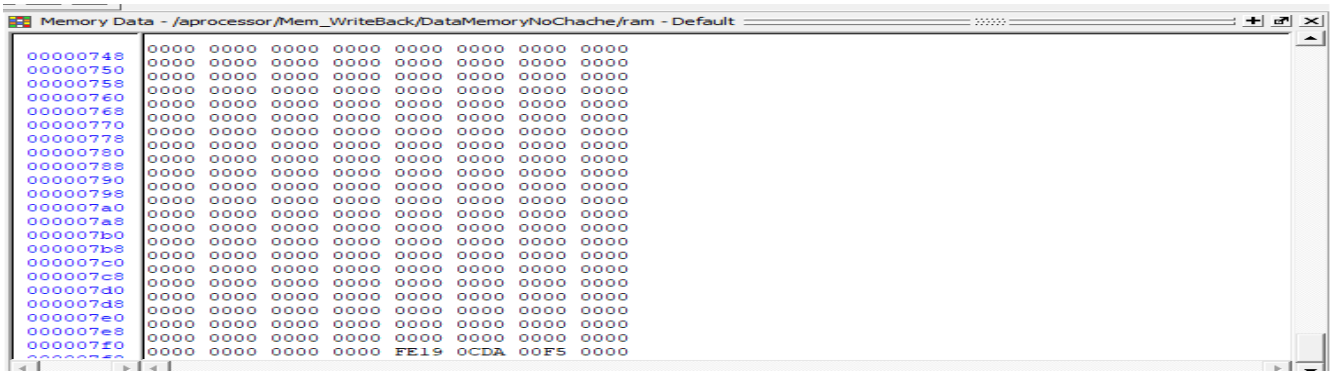


Memory Contents at different cycles:

At cycle 9



At cycle 10



- **Memory Cache test cases:**

Note that this is the only file done with the full processor.

a) Processor with no hazard detection unit or forwarding unit or flushing

The processor will have data hazards due to the absence of forwarding unit as we can see in:

.ORG 118

ADD R4,R4,R6 # R6=R4 * 2

STD R6,312 #M[312, 313]=R6, Data cache write miss (once in loop A, with each iteration in loop b), replaced block is not dirty (first time in B, block will be dirty as a result of STD R1,210

LDM R7,0FF #R7=0FF

AND R6,R6,R7

LDM R5,1 #R5=1

OR R5,R5,R6

LDD R6,312 #R6=M[312, 313]

Ret

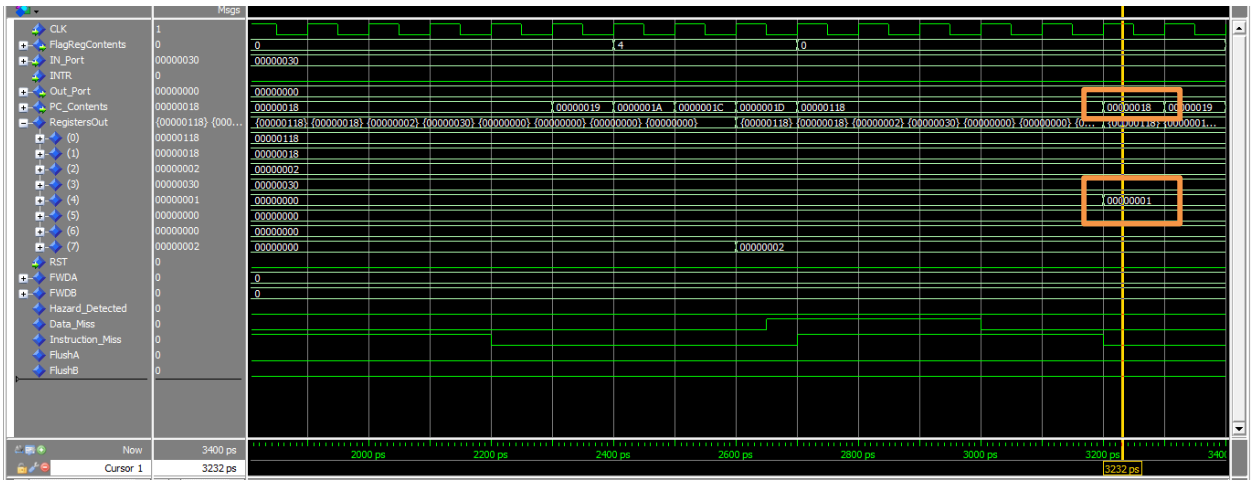
As we can see there is a data dependency "RAW" between Add R4,R4,R6 and STD R6,312 and between LDM R5,1 and OR R5,R5,R6

And flushing also will rise problems as there is instructions will be executed and doesn't meant to be executed

Here are some screen shots to show the errors:

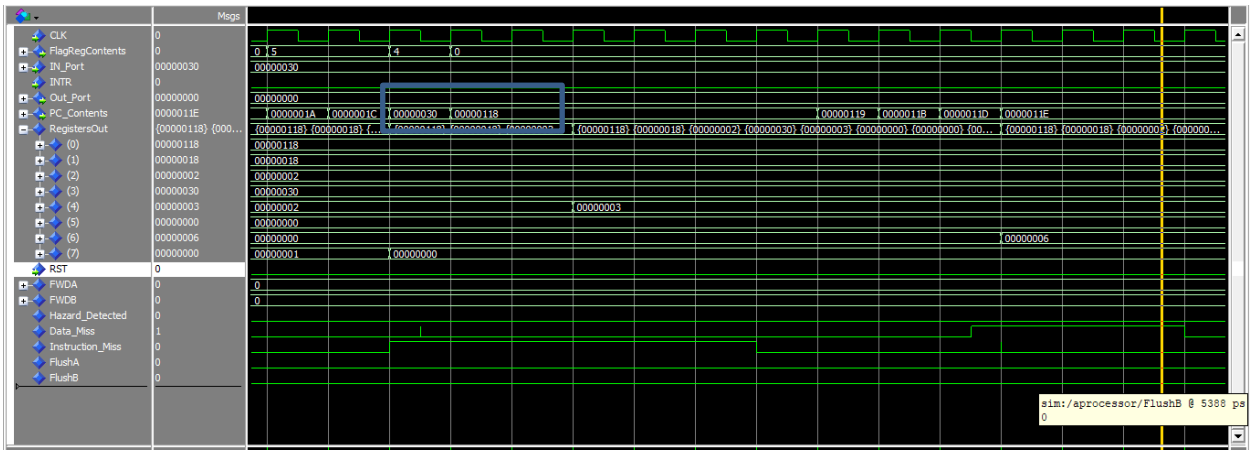
As we can see after executing CALL R0 instruction INC R4 is executed which is wrong and it should be flushed

Then JMP R1 is executed causing it to go out of the function called

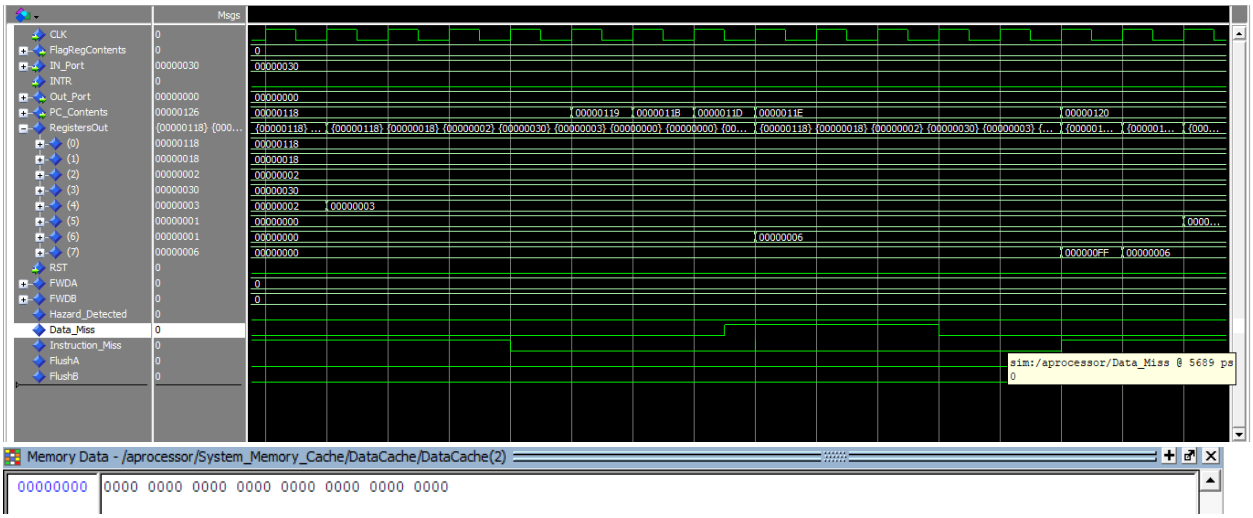


After doing this for 2 times JZ R3 will be executed

And because there is no flushing CALL R0 is also executed.



And as we can see the stored value of R6 is its old value which is 0 not 6 due to absence of forwarding



And these cases will continue to happen as we go.

And this can be solved by adding NOP's as follow:

.ORG 10

```
in R0 #R0 = 118, Instr cache read miss
in R1 #R1 = 18
in R2 #R2 = 2
in R3 #R3 = 30
JMP R1 #jump to 18
Nop
Nop
.ORG 18 #Loop A
SUB R2,R4,R7 #check if R4 = R2, Instr cache read miss will occur with each loop iteration
JZ R3 #jump to 30 if R4 = R2
Nop
Nop
CALL R0 #Instr cache read miss for block starting with 118 each time we call. Data cache write miss for the first iteration in loop for block starting with 7F8.
Nop
Nop
Nop
OUT R5
INC R4
JMP R1 #jump to 18
.ORG 30
in R1 #R1 = 38
in R3 #R3 = 50
in R4 #R4 = 0
STD R1,210 #M[210, 211]=38, Data cache write miss, replaced block is dirty
JMP R1 #jump to 38
.ORG 38 #Loop B
SUB R2,R4,R7 #check if R4 = R2, Instr cache read miss will occur for first loop iteration
JZ R3 #jump to 50 if R4 = R2
Nop
Nop
CALL R0#Instr cache read miss for block starting with 118 for first loop iteration.
Nop
Nop
Nop
LDD R1,210 #R1 = 38, Data cache read miss with each iteration, replaced block is dirty
OUT R5
INC R4
JMP R1 #jump to 38
Nop
Nop
.ORG 50
STD R6,212 #M[212, 213]=R6
.ORG 118
ADD R4,R4,R6 # R6=R4 * 2
Nop
```

Nop

STD R6,312 #M[312, 313]=R6, Data cache write miss (once in loop A, with each iteration in loop b), replaced block is not dirty (first time in B, block will be dirty as a result of STD R1,210

LDM R7,0FF #R7=0FF

AND R6,R6,R7

LDM R5,1 #R5=1

Nop

Nop

OR R5,R5,R6

LDD R6,312 #R6=M[312, 313]

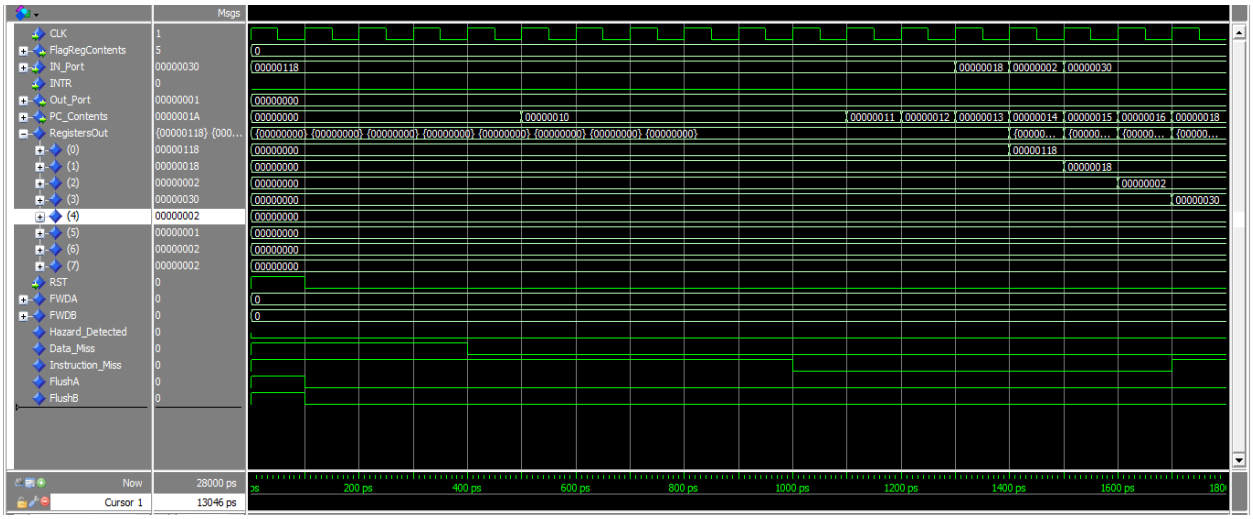
Ret

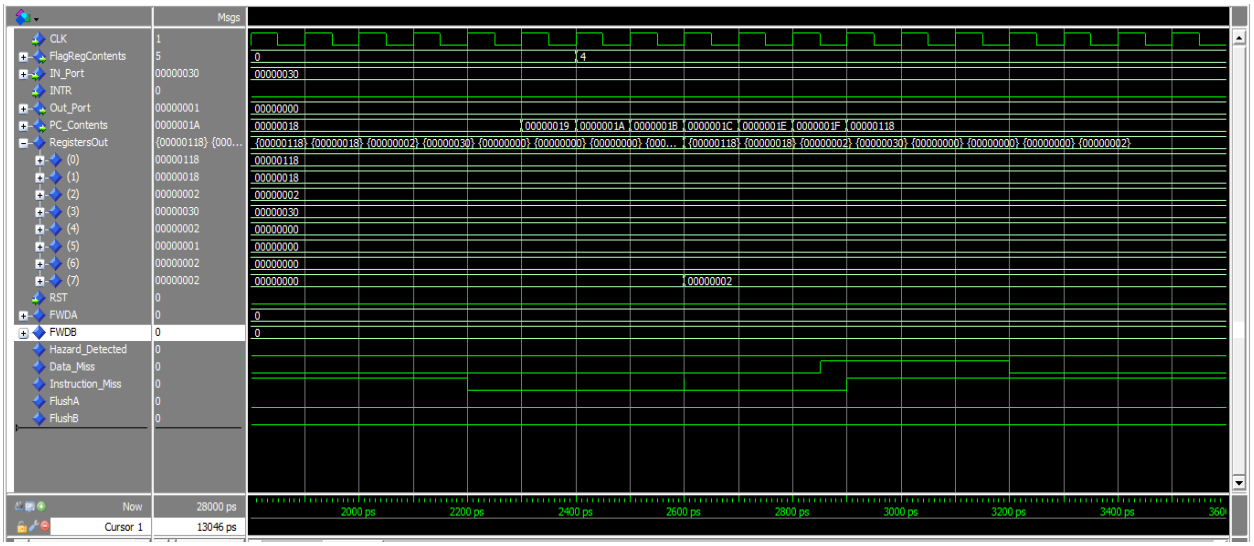
Nop

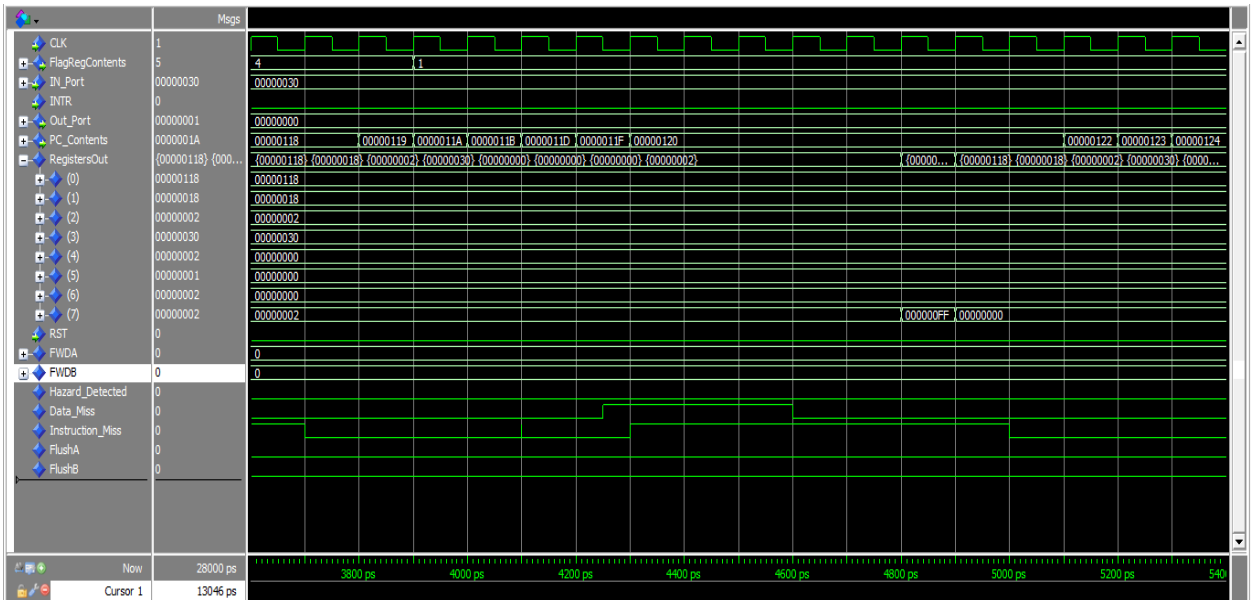
Nop

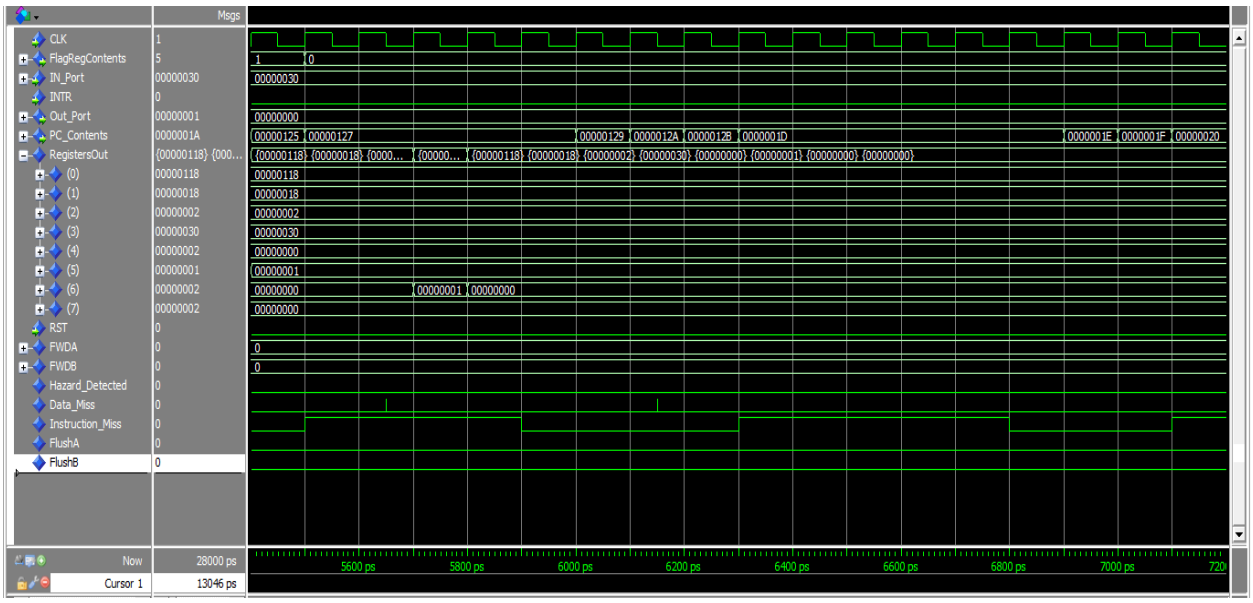
Nop

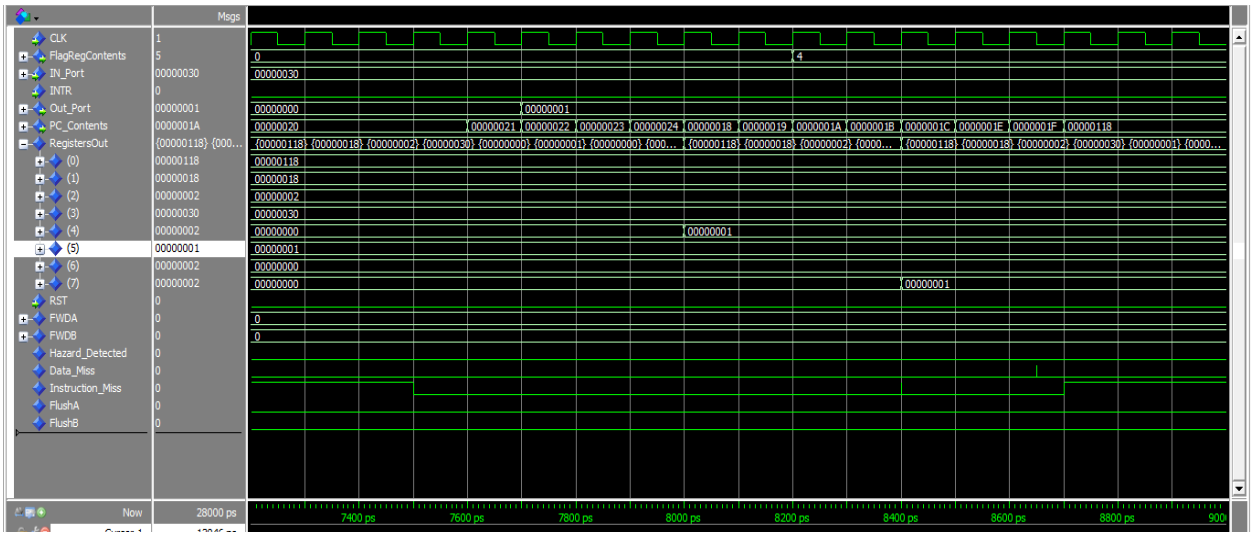
And by running that; the following are all screen shots in order: (Note that they are the same as the full working processor except with more cycles any notes on them are done once in full processor run and there is more details)

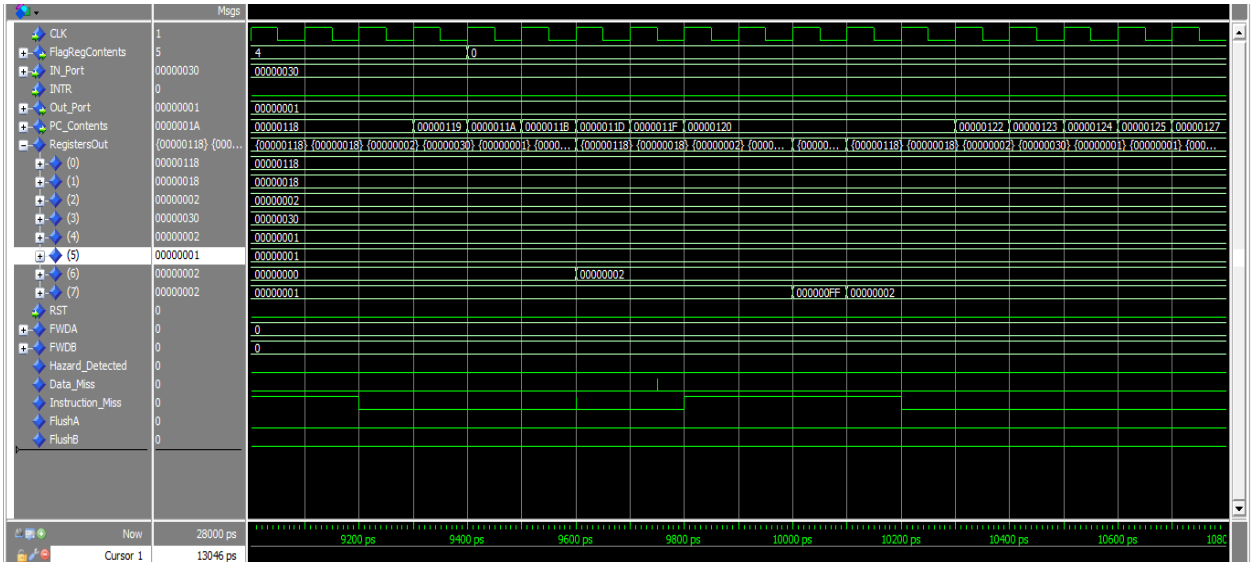


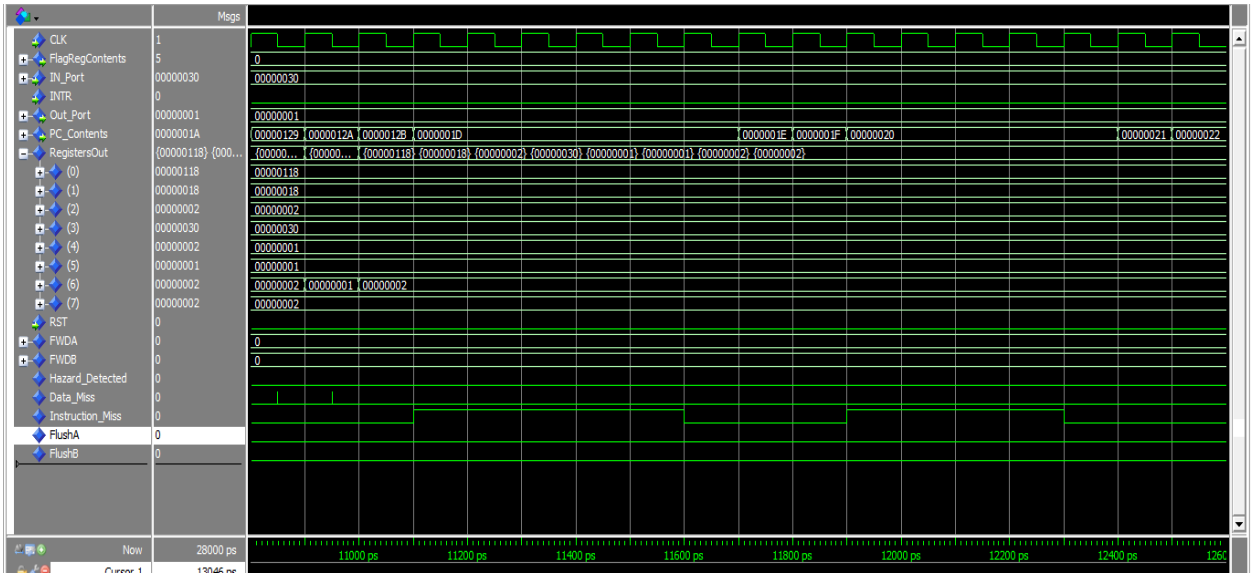


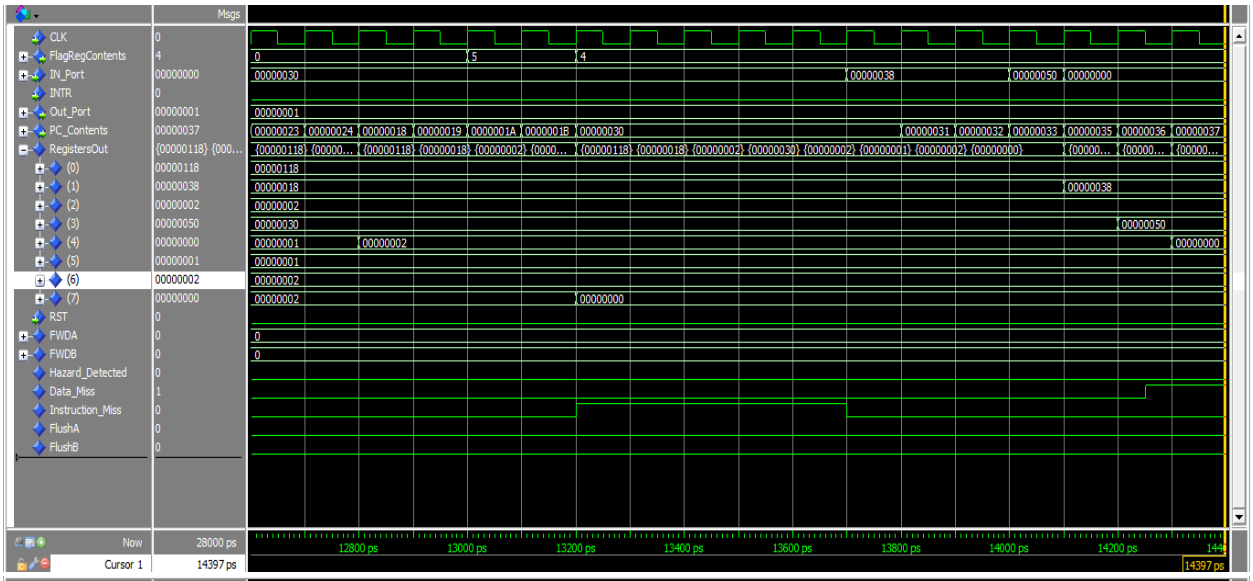

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
13046 ps
3800 ps
4000 ps
4200 ps
4400 ps
4600 ps
4800 ps
5000 ps
5200 ps

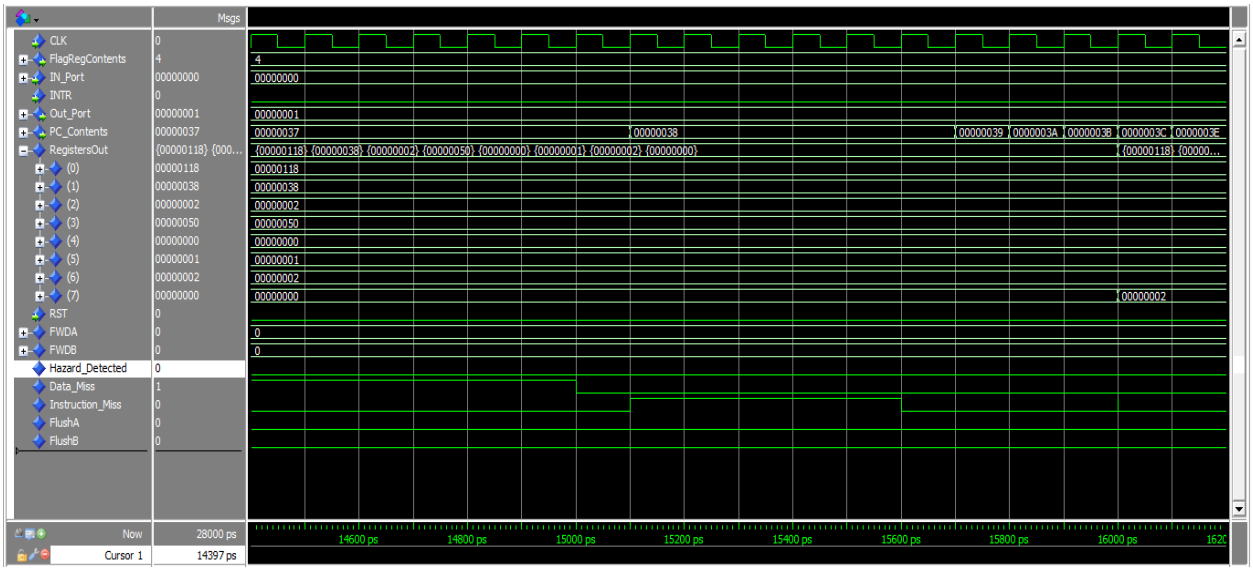

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
13046 ps
5600 ps
5800 ps
6000 ps
6200 ps
6400 ps
6600 ps
6800 ps
7000 ps

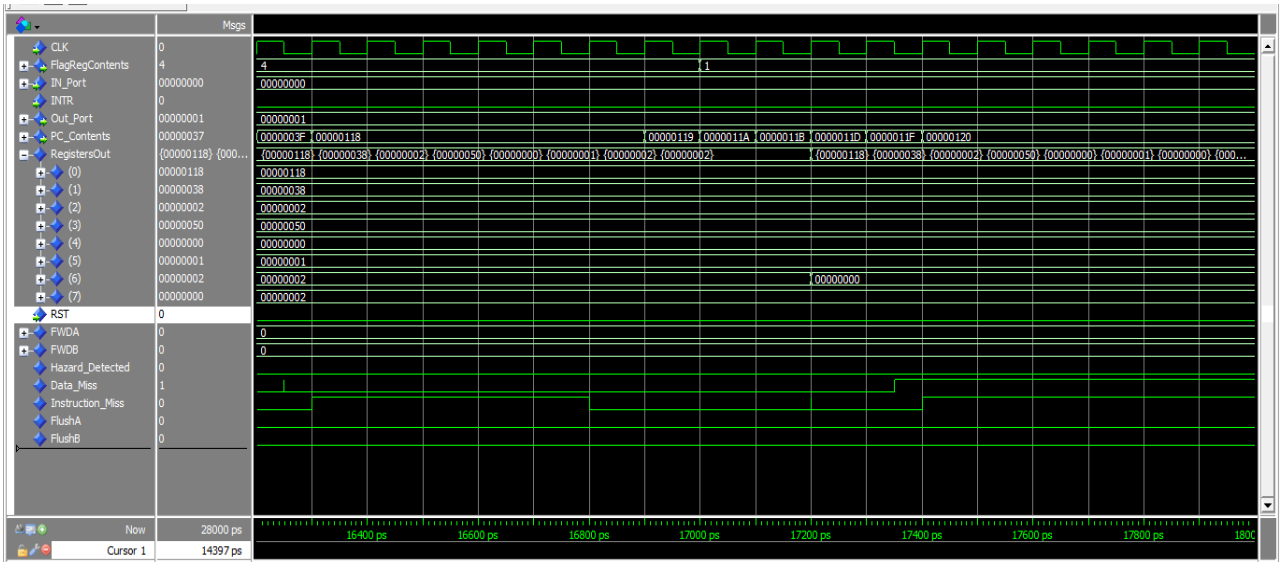

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
7400 ps
7600 ps
7800 ps
8000 ps
8200 ps
8400 ps
8600 ps
8800 ps

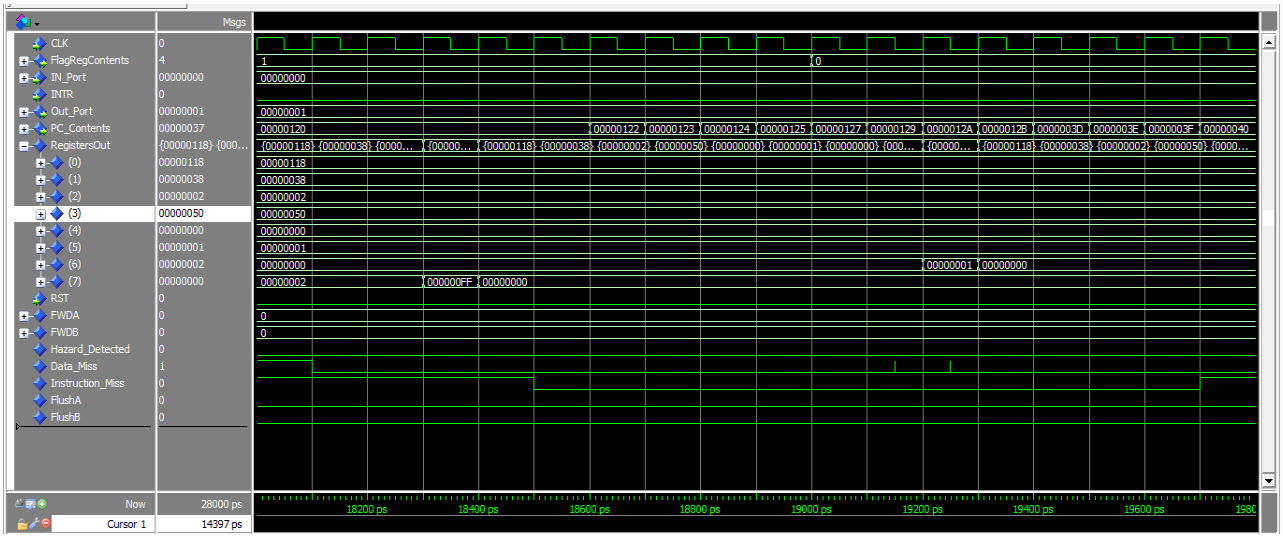

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
13046 ps
9200 ps
9400 ps
9600 ps
9800 ps
10000 ps
10200 ps
10400 ps
10600 ps


Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
11000 ps
11200 ps
11400 ps
11600 ps
11800 ps
12000 ps
12200 ps
12400 ps

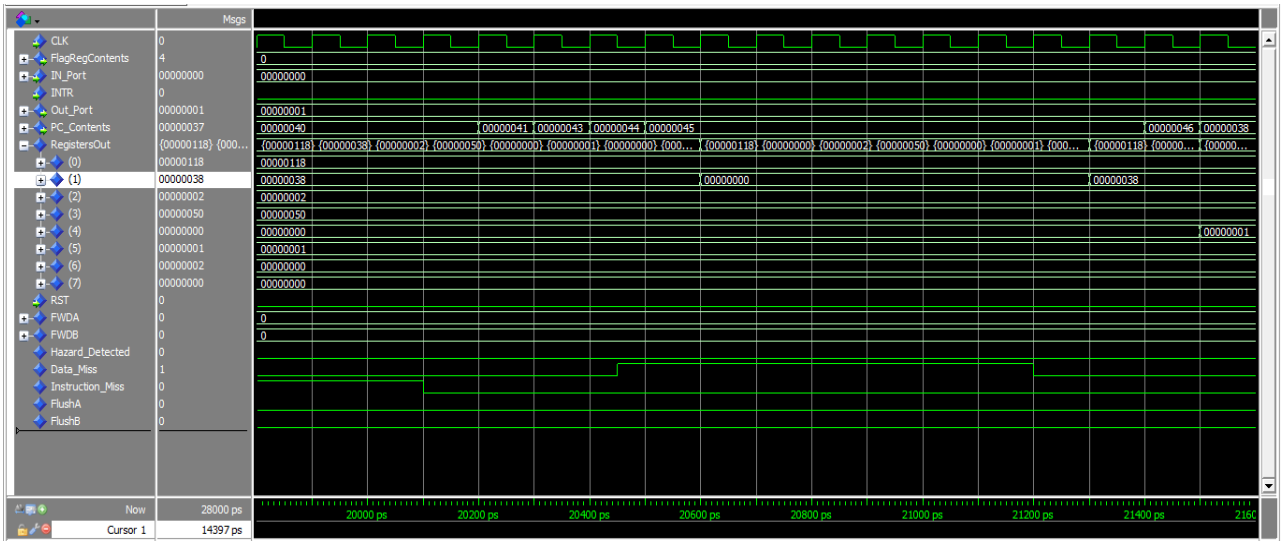

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ps
12800 ps
13000 ps
13200 ps
13400 ps
13600 ps
13800 ps
14000 ps
14200 ps

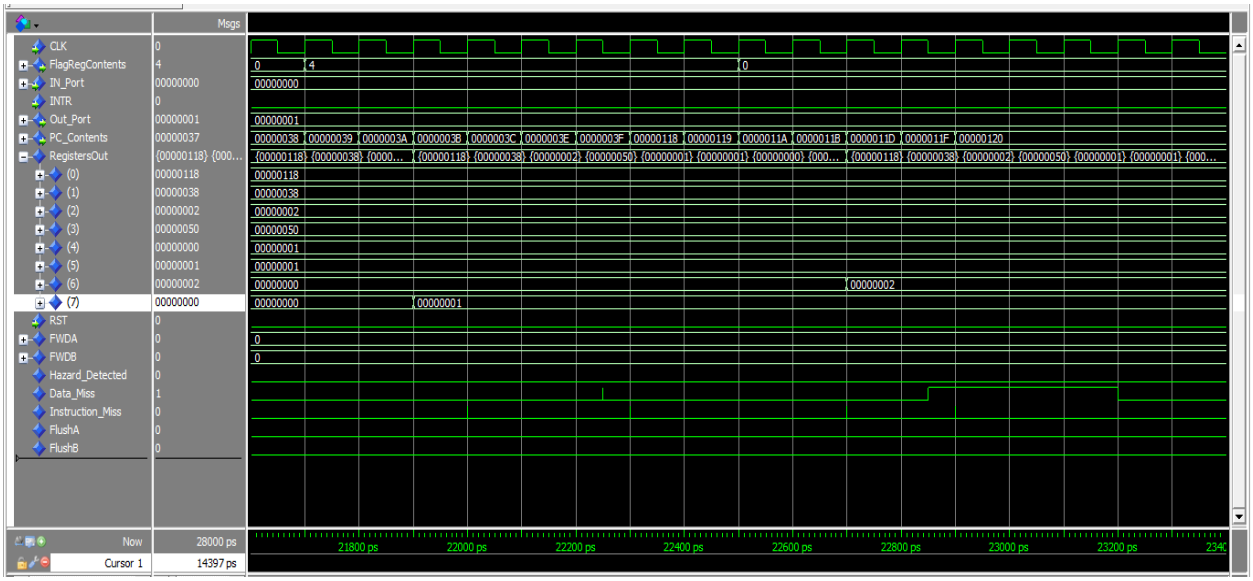

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ps
14600 ps
14800 ps
15000 ps
15200 ps
15400 ps
15600 ps
15800 ps
16000 ps

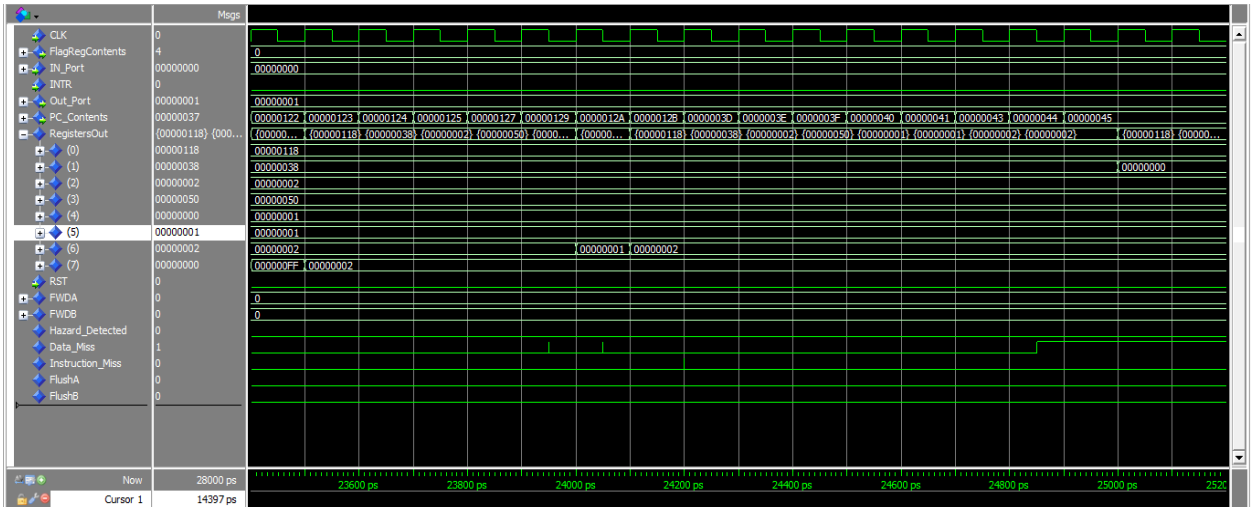

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ps
16400 ps
16600 ps
16800 ps
17000 ps
17200 ps
17400 ps
17600 ps
17800 ps

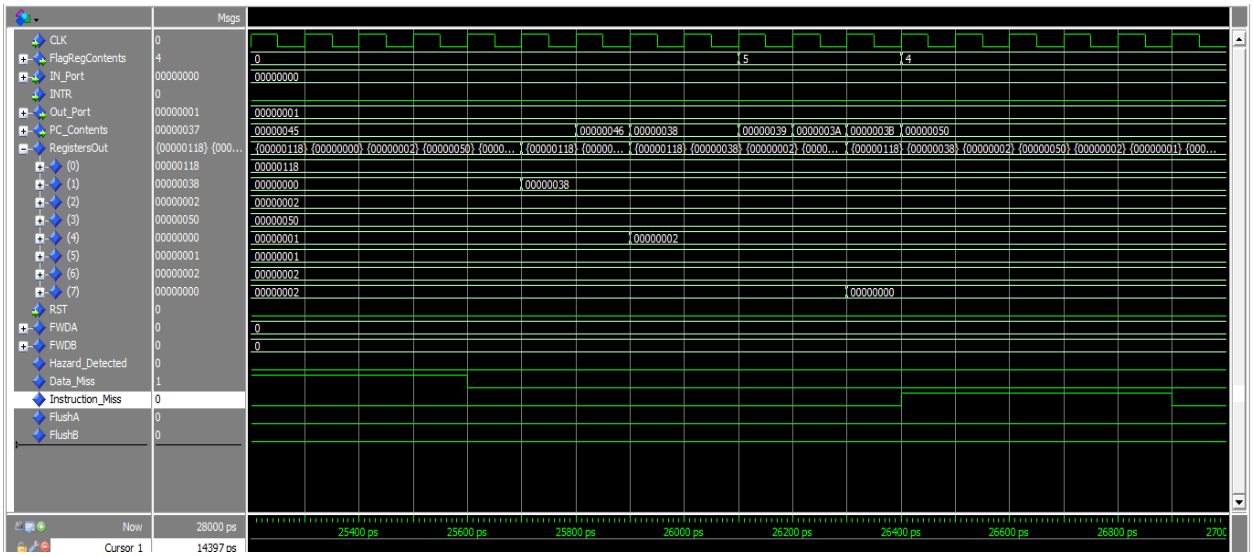


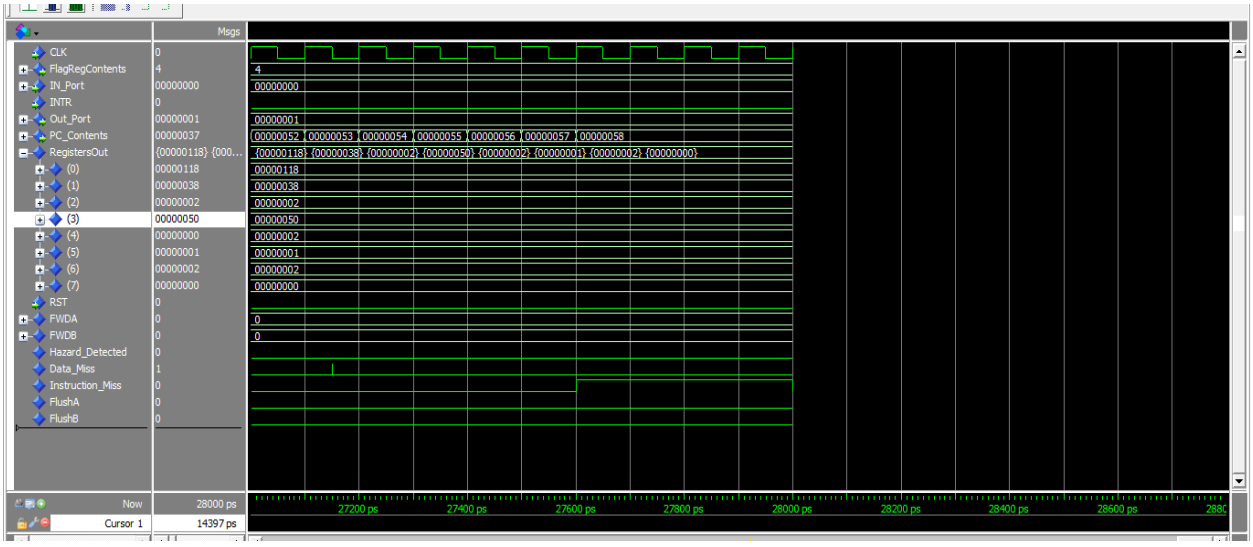
(Writing back zeros here is meaningless so there is no error here but we don't stall write back stage in order to handle interrupts correctly so it will keep writing garbage till it writes the true value).






Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ps
23600 ps
23800 ps
24000 ps
24200 ps
24400 ps
24600 ps
24800 ps
25000 ps

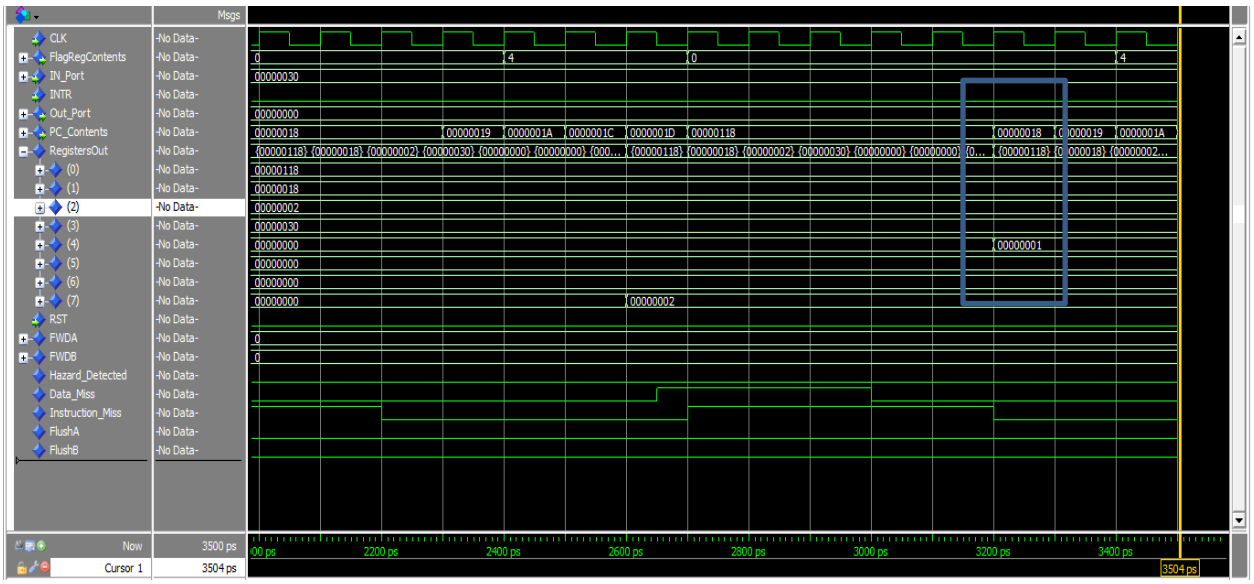

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ns
25400 ps
25600 ps
25800 ps
26000 ps
26200 ps
26400 ps
26600 ps
26800 ps


Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
28000 ps
Cursor 1
14397 ps
27200 ps
27400 ps
27600 ps
27800 ps
28000 ps
28200 ps
28400 ps
28600 ps

b) Now by adding forwarding unit some hazards are solved but the problems of non-flushed instructions will continue for example:

As we have seen before: after executing CALL R0 instruction INC R4 is executed which is wrong and it should be flushed

Then JMP R1 is executed causing it to go out of the function called

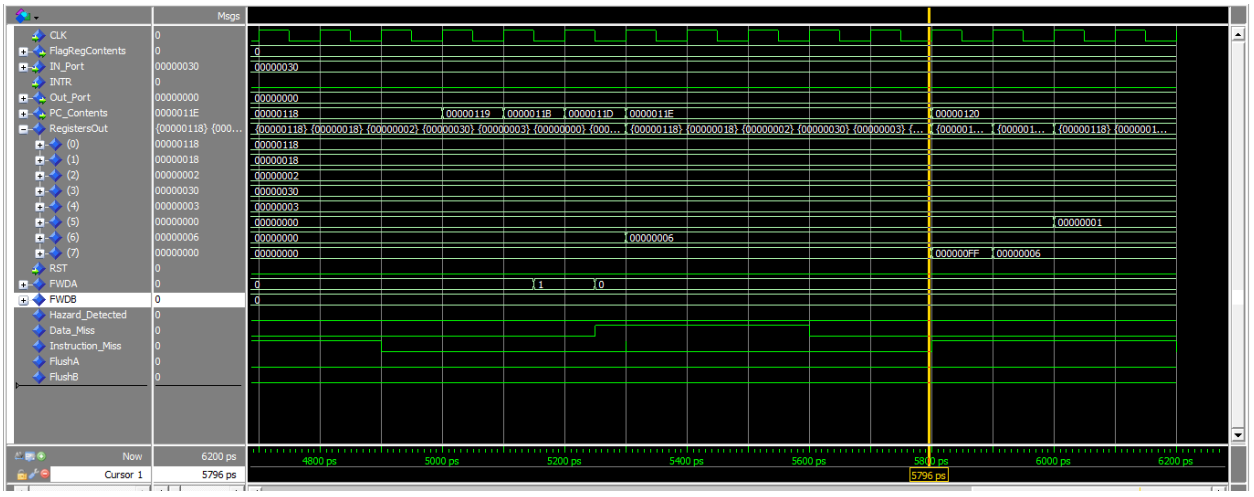


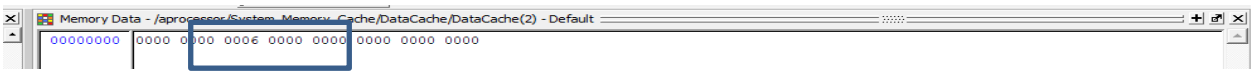
And same as before but the forwarded values are correct now as we can see the stored value is correct (which implies that it is forwarded correctly).

In the sequence:

ADD R4,R4,R6 # R6=R4 * 2

STD R6,312 #M[312, 313]=R6





And this can be solved by adding NOP's as follow:

.ORG 10

in R0 #R0 = 118, Instr cache read miss

in R1 #R1 = 18

in R2 #R2 = 2

in R3 #R3 = 30

```
JMP R1 #jump to 18
Nop
Nop
.ORG 18 #Loop A
SUB R2,R4,R7 #check if R4 = R2, Instr cache read miss will occur with each loop iteration
JZ R3 #jump to 30 if R4 = R2
Nop
Nop
CALL R0 #Instr cache read miss for block starting with 118 each time we call. Data cache write miss for the first iteration in loop for block starting with 7F8.
Nop
Nop
Nop
OUT R5
INC R4
JMP R1 #jump to 18
.ORG 30
in R1 #R1 = 38
in R3 #R3 = 50
in R4 #R4 = 0
STD R1,210 #M[210, 211]=38, Data cache write miss, replaced block is dirty
JMP R1 #jump to 38
.ORG 38 #Loop B
SUB R2,R4,R7 #check if R4 = R2, Instr cache read miss will occur for first loop iteration
JZ R3 #jump to 50 if R4 = R2
Nop
Nop
CALL R0#Instr cache read miss for block starting with 118 for first loop iteration.
Nop
Nop
Nop
LDD R1,210 #R1 = 38, Data cache read miss with each iteration, replaced block is dirty
OUT R5
INC R4
JMP R1 #jump to 38
Nop
Nop
.ORG 50
STD R6,212 #M[212, 213]=R6
.ORG 118
ADD R4,R4,R6 # R6=R4 * 2
STD R6,312 #M[312, 313]=R6, Data cache write miss (once in loop A, with each iteration in loop b), replaced block is not dirty (first time in B, block will be dirty as a result of STD R1,210
LDM R7,0FF #R7=0FF
AND R6,R6,R7
LDM R5,1 #R5=1
```

OR R5,R5,R6

LDD R6,312 #R6=M[312, 313]

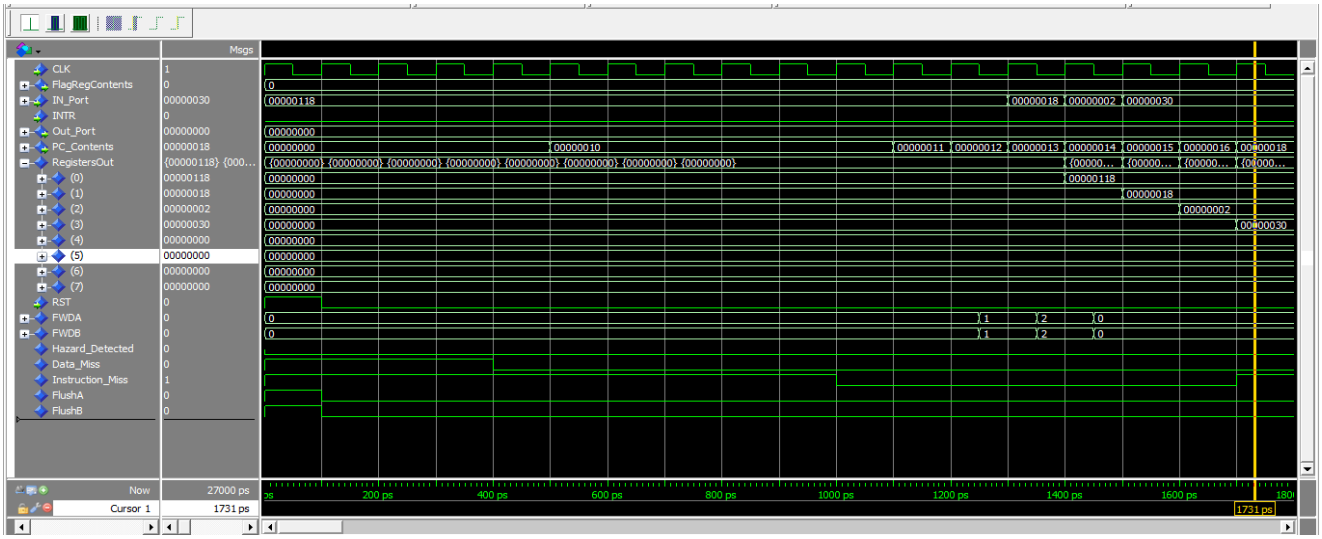
Ret

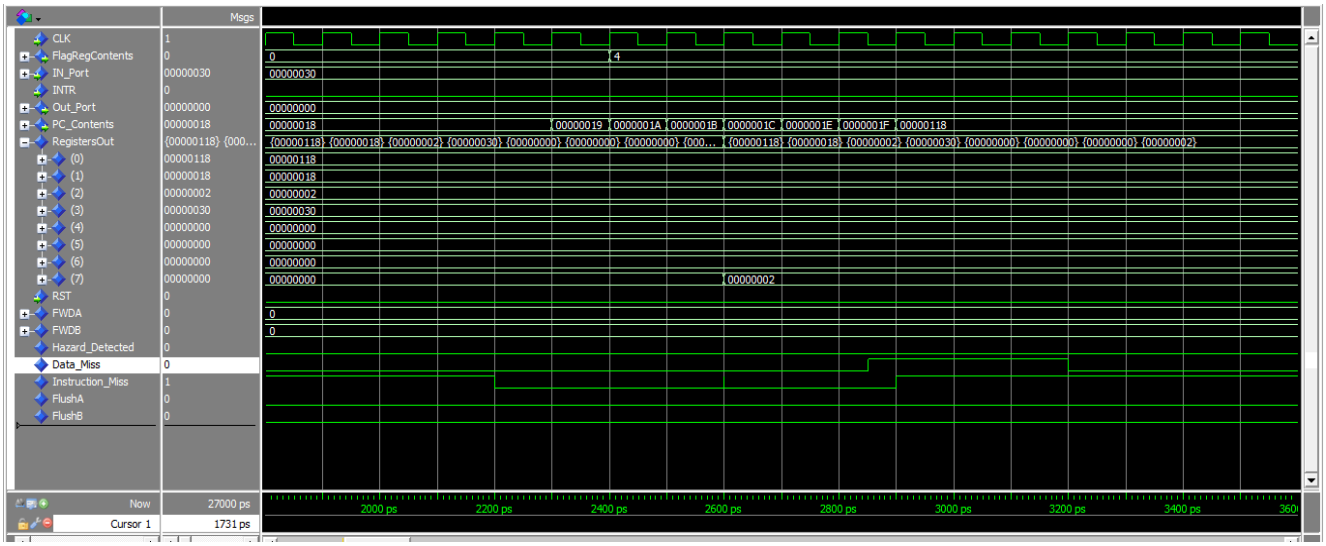
Nop

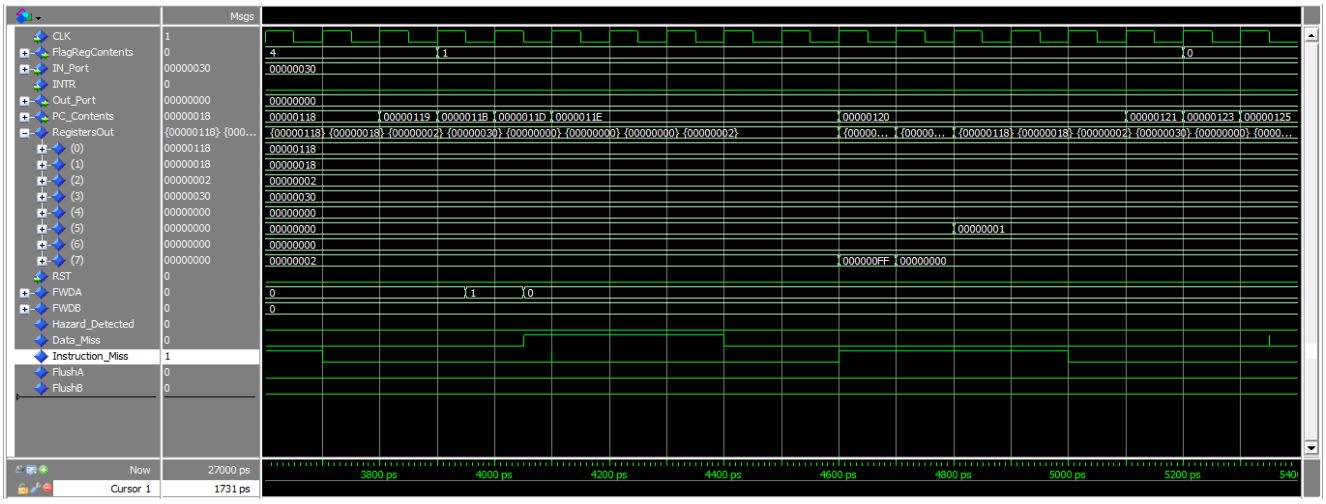
Nop

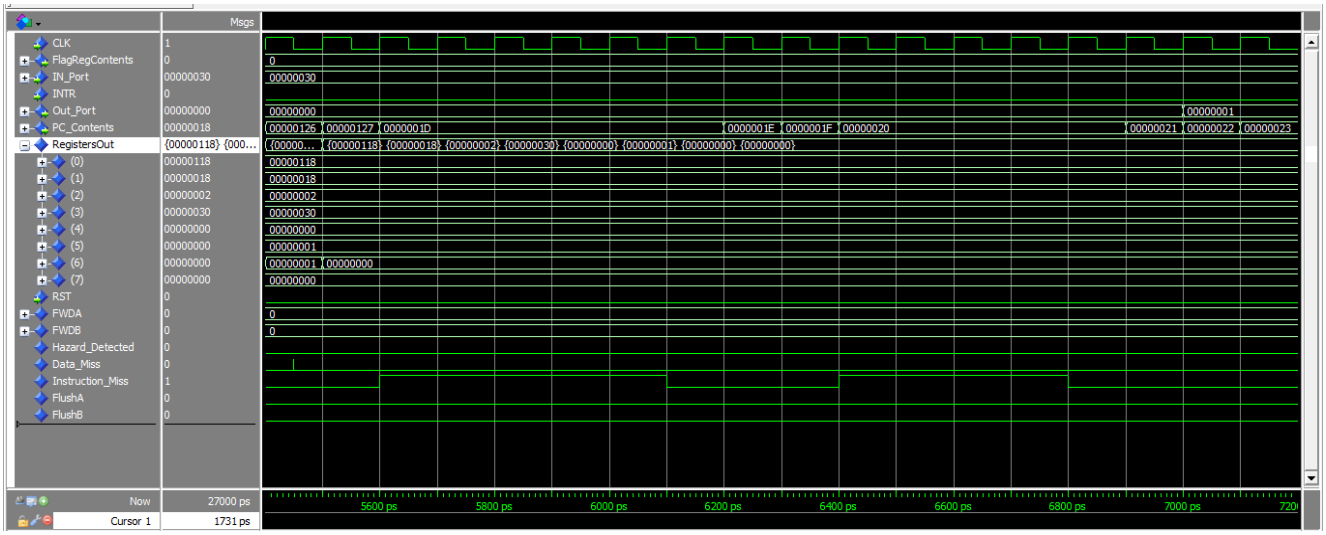
Nop

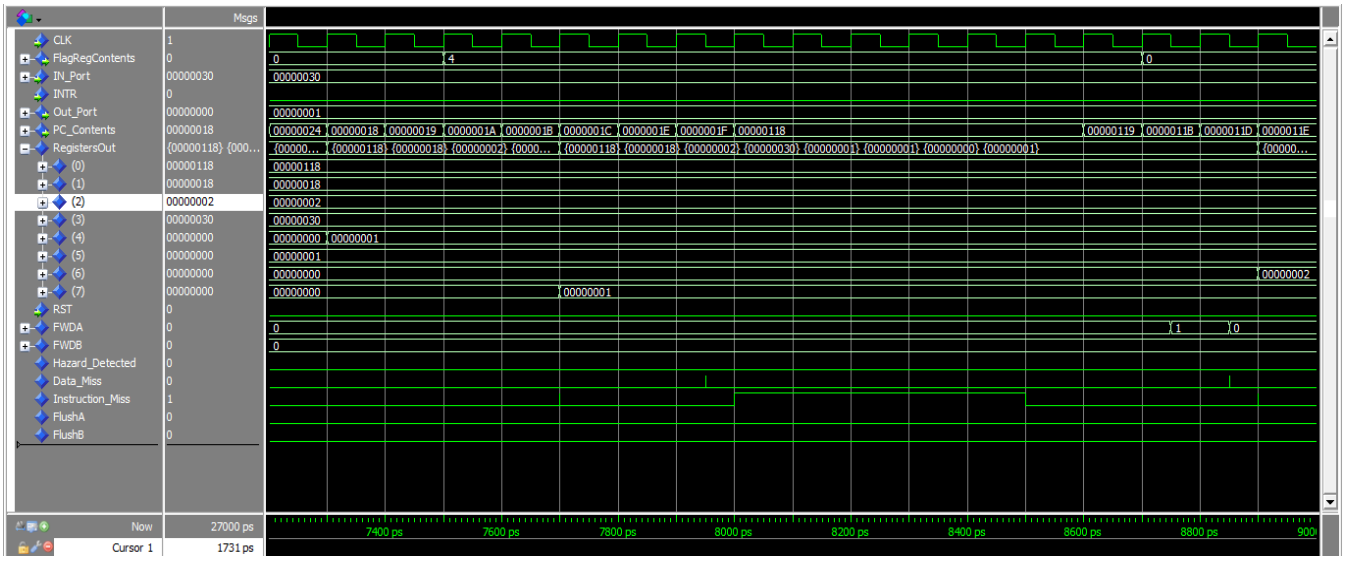
And by running that; the following are all screen shots in order: (Note that they are the same as the full working processor except with more cycles any notes on them are done once in full processor run and there is more details):

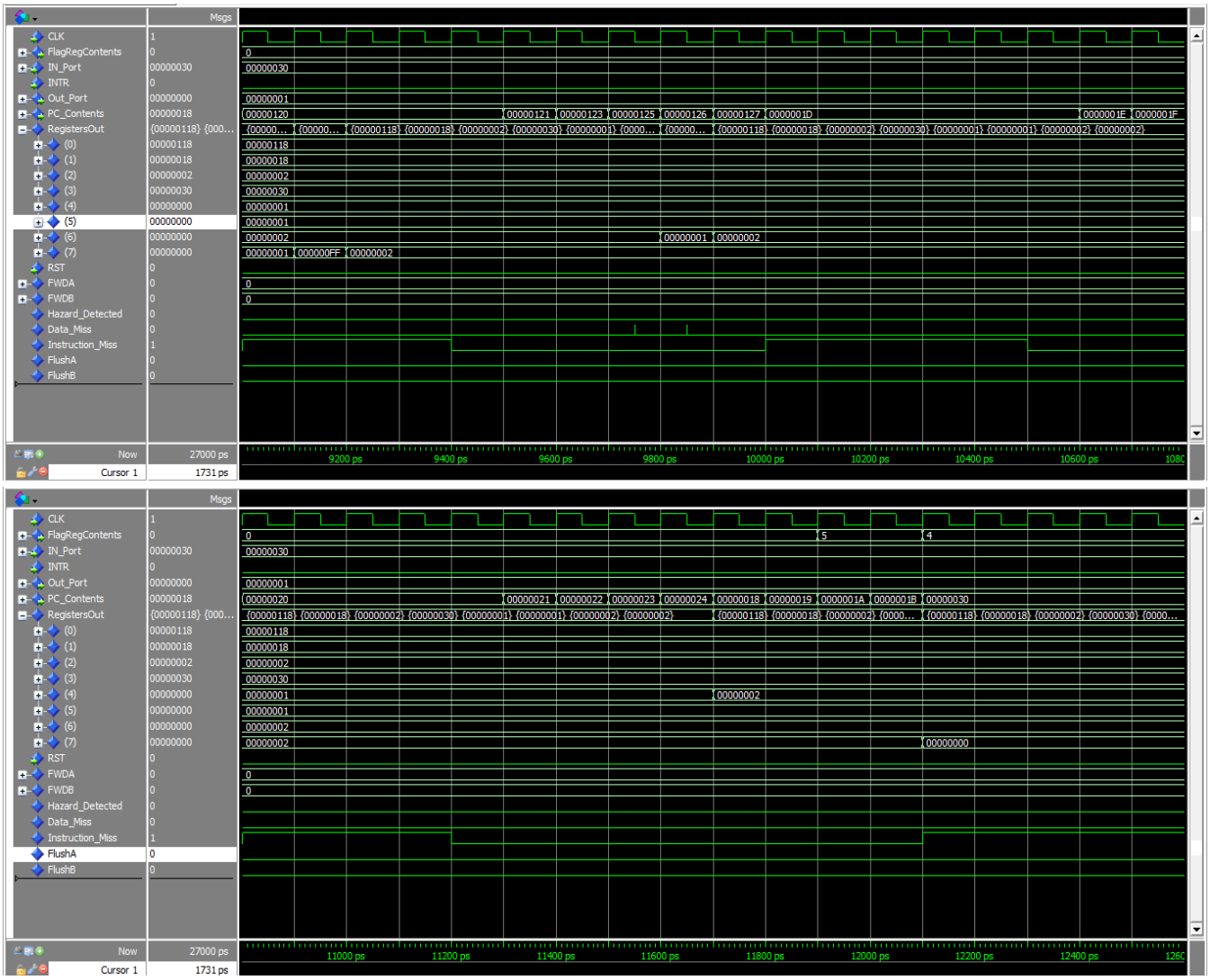


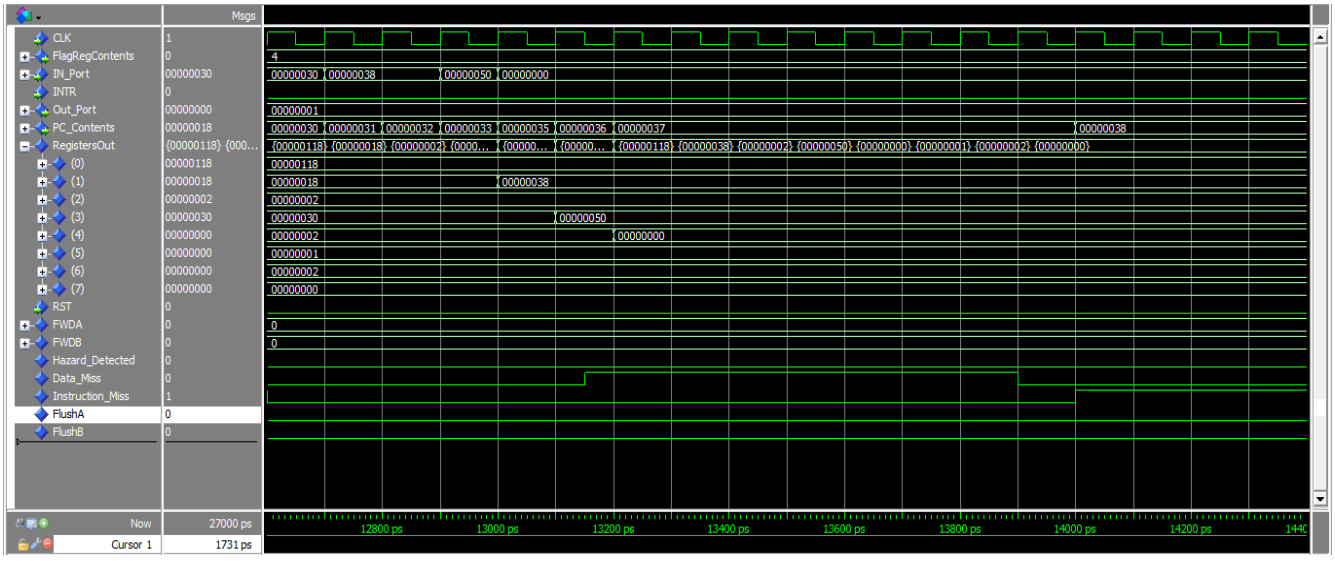


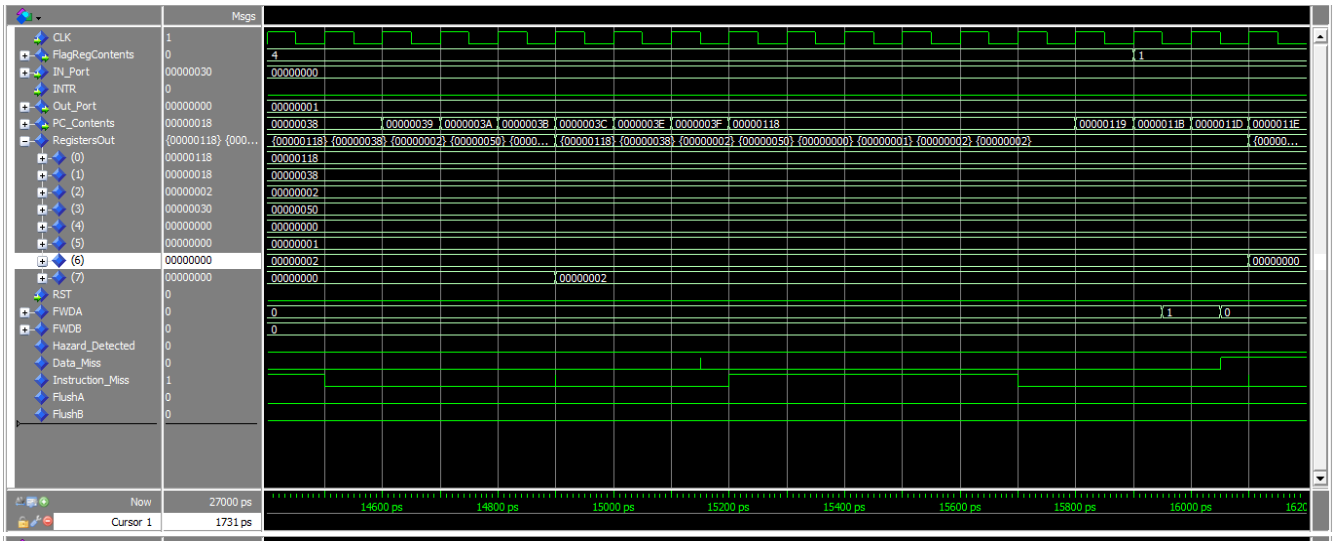

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
3800 ps
4000 ps
4200 ps
4400 ps
4600 ps
4800 ps
5000 ps
5200 ps

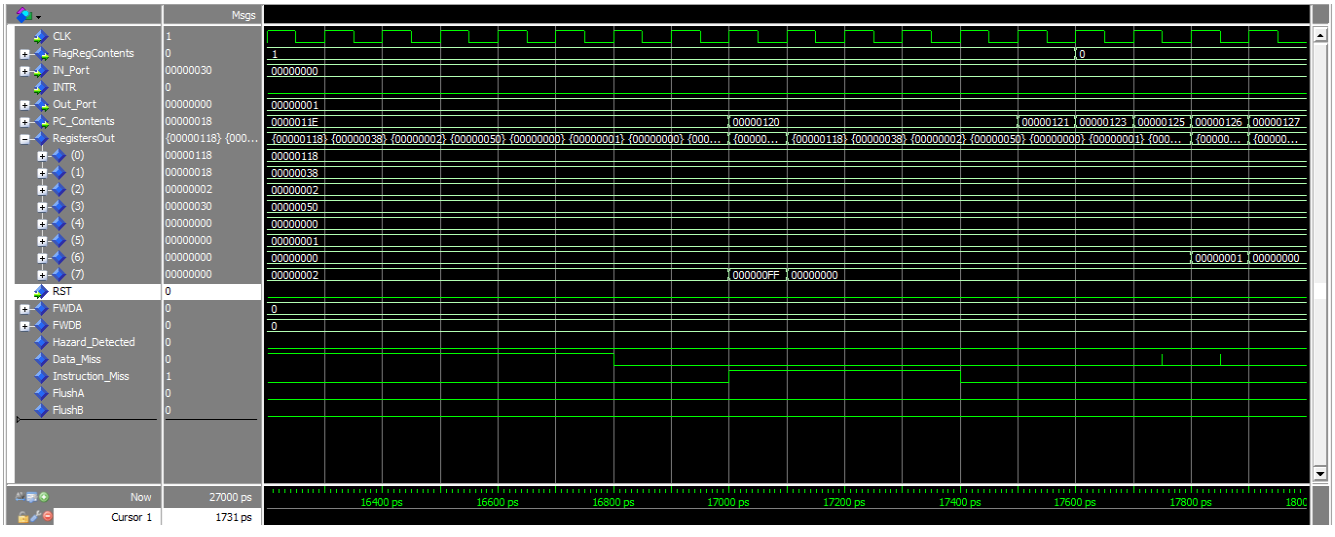

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
5600 ps
5800 ps
6000 ps
6200 ps
6400 ps
6600 ps
6800 ps
7000 ps

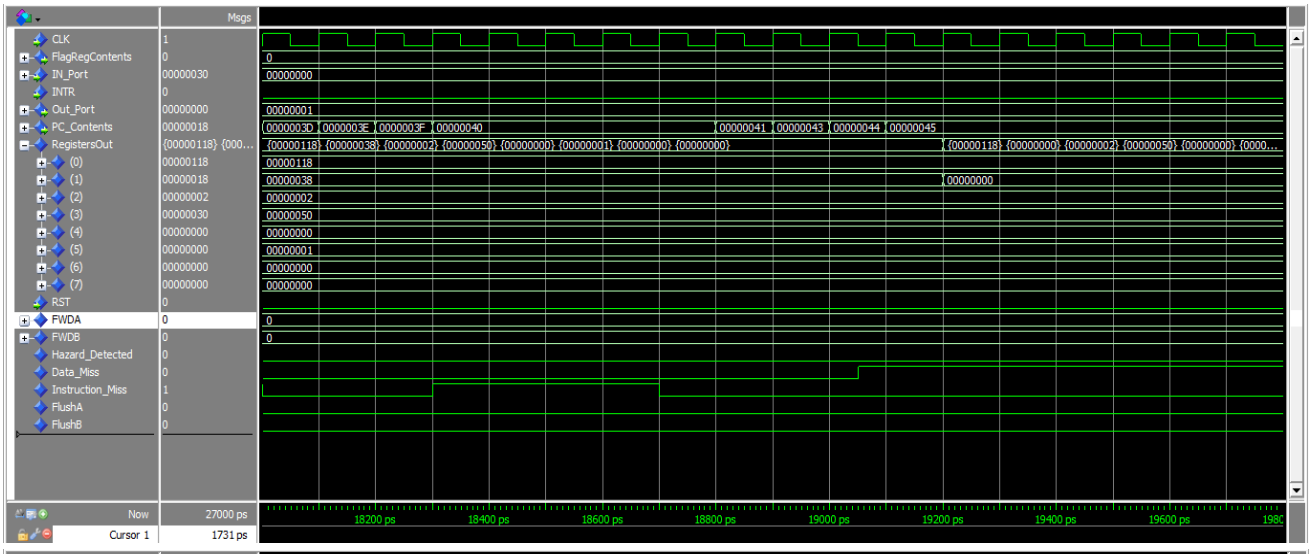

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
7400 ps
7600 ps
7800 ps
8000 ps
8200 ps
8400 ps
8600 ps
8800 ps

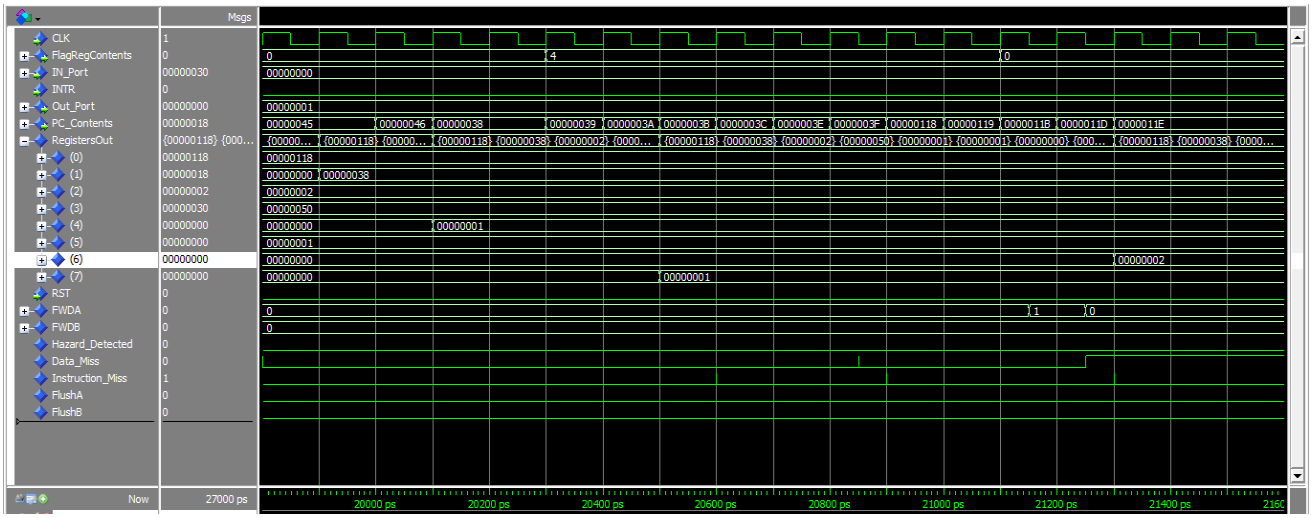

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
9200 ps
9400 ps
9600 ps
9800 ps
10000 ps
10200 ps
10400 ps
10600 ps
Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
11000 ps
11200 ps
11400 ps
11600 ps
11800 ps
12000 ps
12200 ps
12400 ps


Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
12800 ps
13000 ps
13200 ps
13400 ps
13600 ps
13800 ps
14000 ps
14200 ps

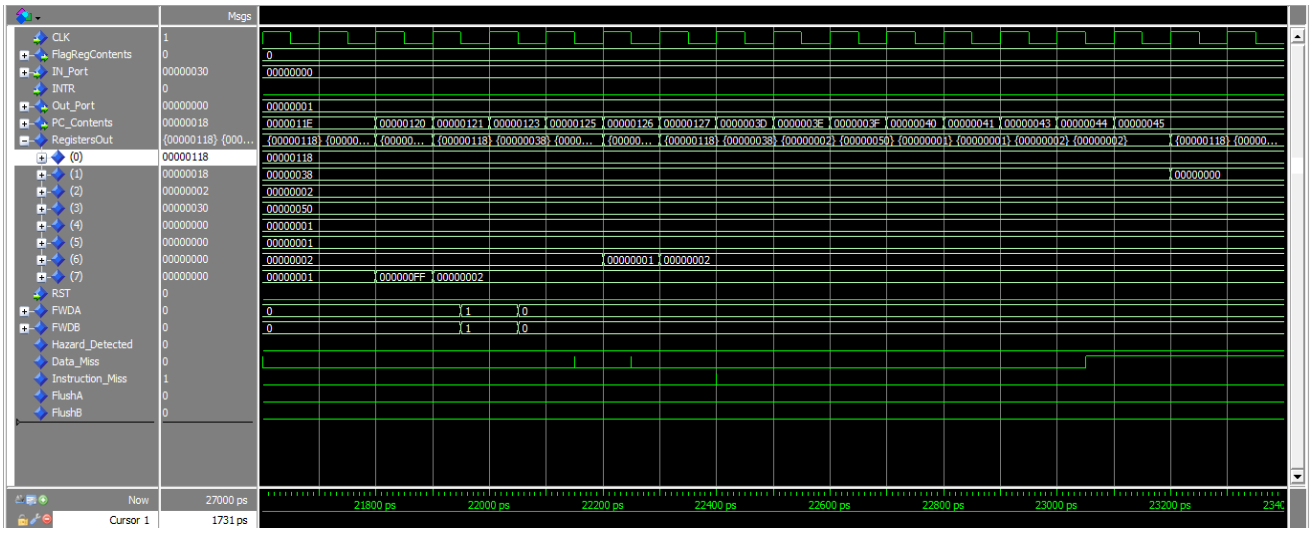

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
14600 ps
14800 ps
15000 ps
15200 ps
15400 ps
15600 ps
15800 ps
16000 ps

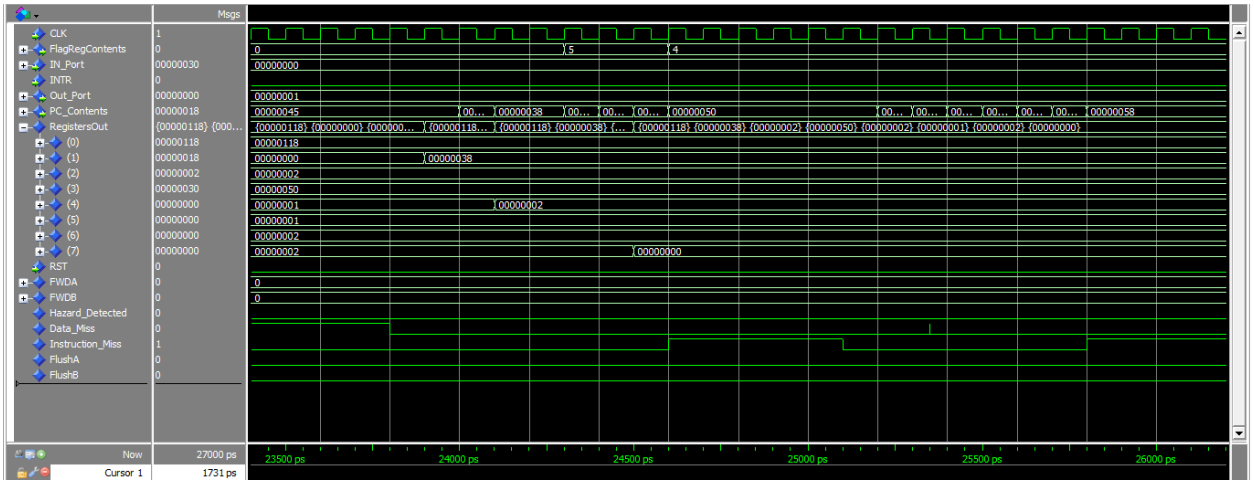

Msgs
CLK
FlagRegContents
IN_Port
INTR
Out_Port
PC_Contents
RegistersOut
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
RST
FWDA
FWDB
Hazard_Detected
Data_Miss
Instruction_Miss
FlushA
FlushB
Now
27000 ps
Cursor 1
1731 ps
16400 ps
16600 ps
16800 ps
17000 ps
17200 ps
17400 ps
17600 ps
17800 ps





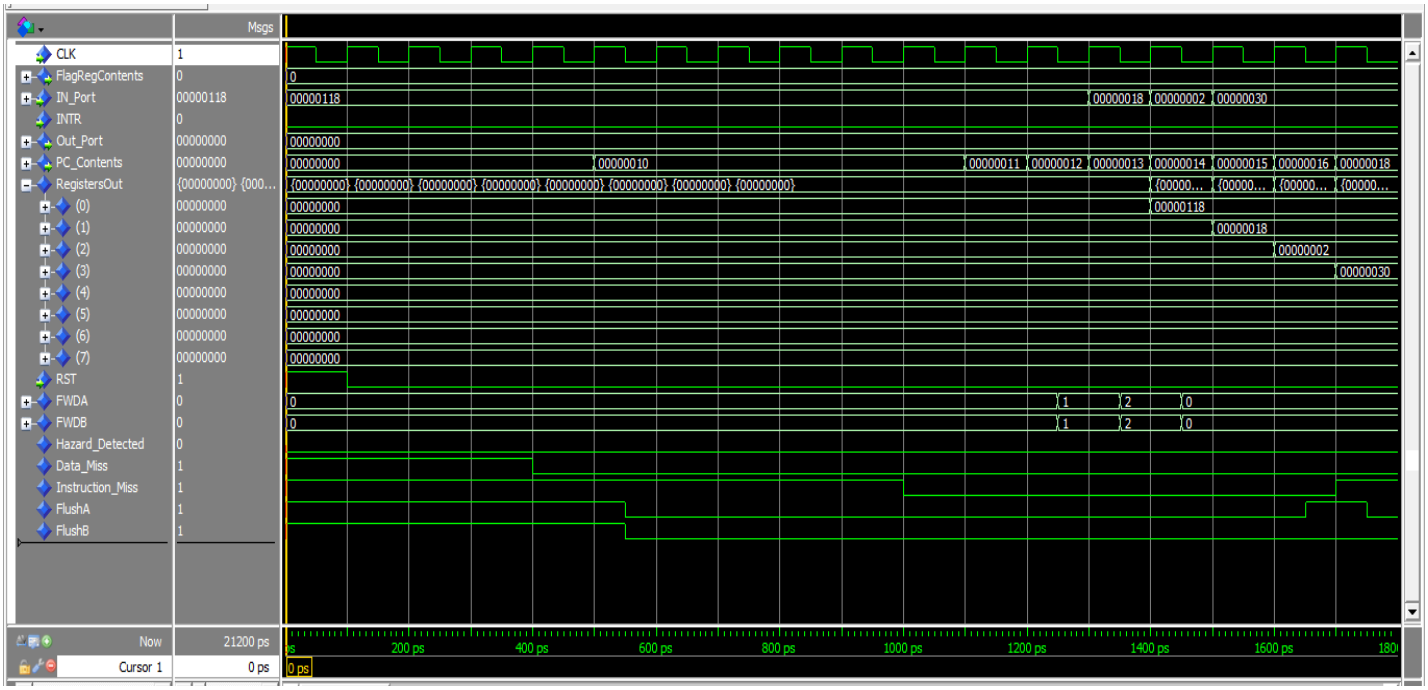
(Writing back zeros here is meaningless so there is no error here but we don't stall write back stage in order to handle interrupts correctly so it will keep writing garbage till it writes the true value).





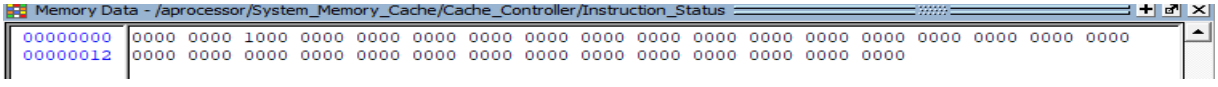
After adding hazard detection unit it will be the same as the last point as there is no load use cases in this code.

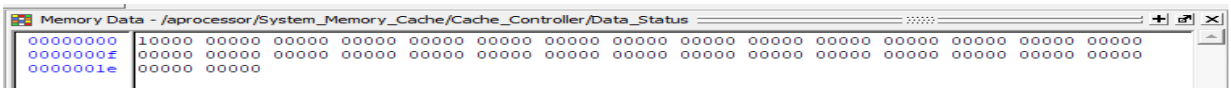
c) Perfectly working processor screen shots.



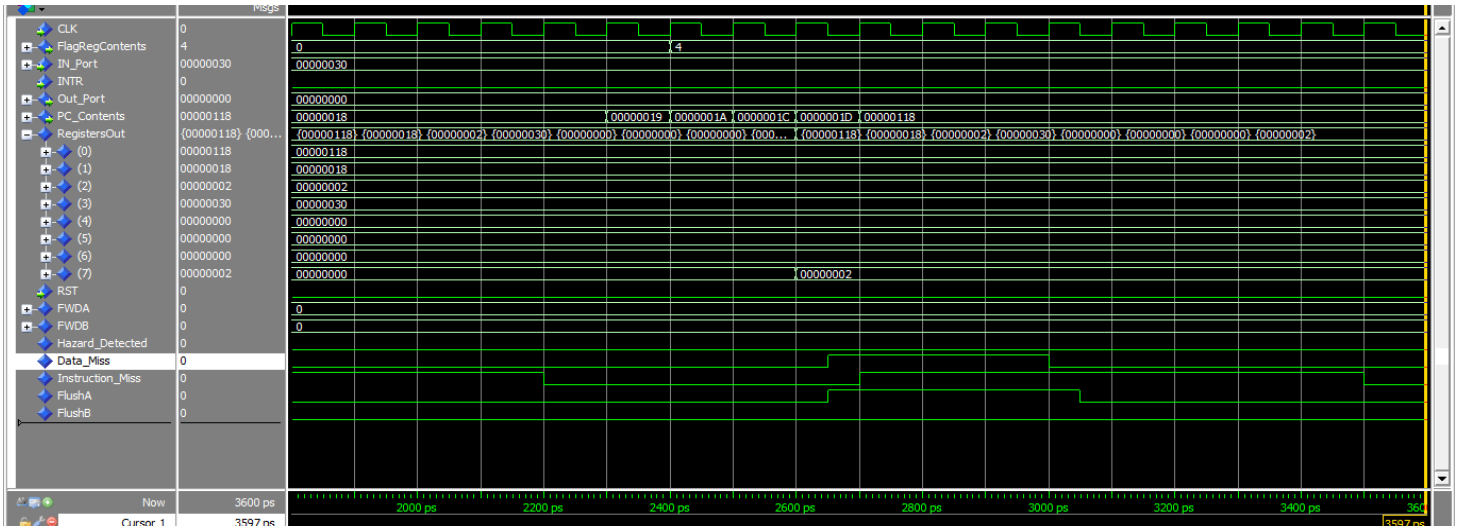
First 18 cycles:

After the first 18 cycle shown the instruction status and data status in the cache controller are as follow:

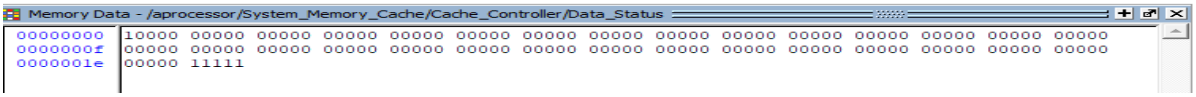


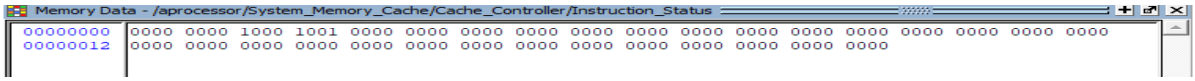


Cycles from 18 to 36:

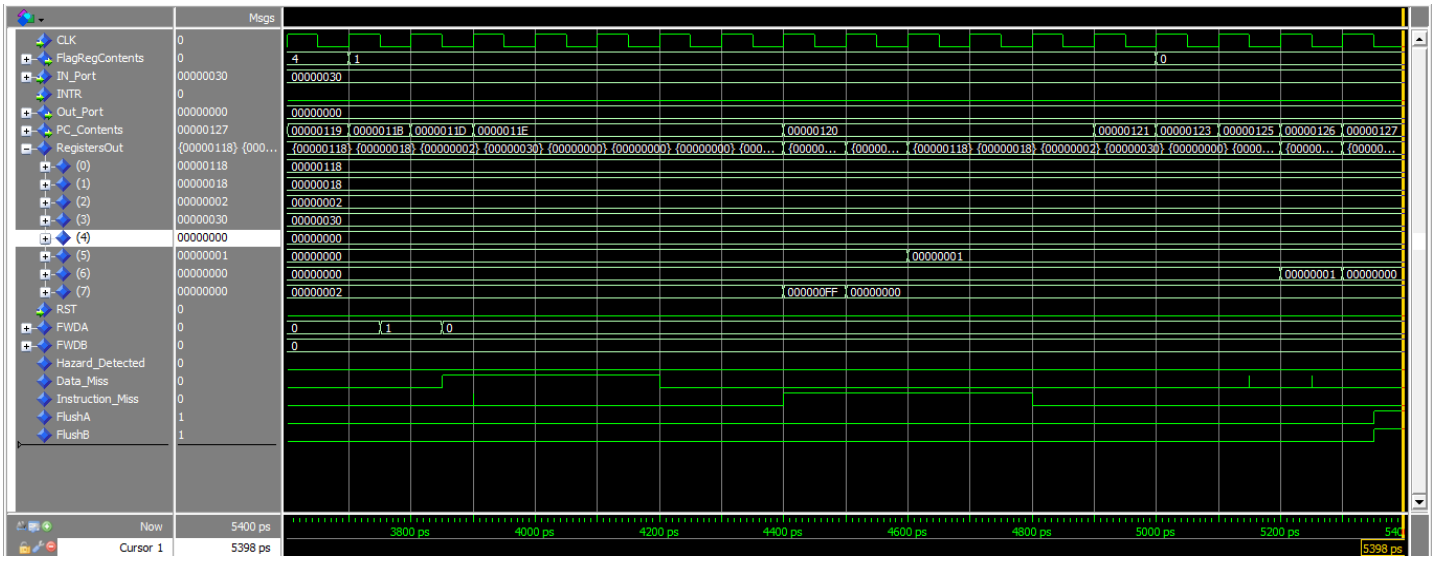


shown the instruction status and data status in the cache controller are as follow after those instructions:

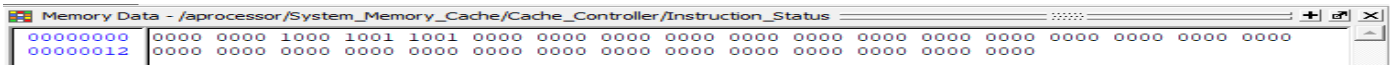


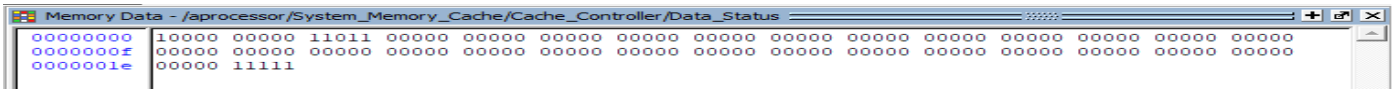


Cycles from 36 to 54:

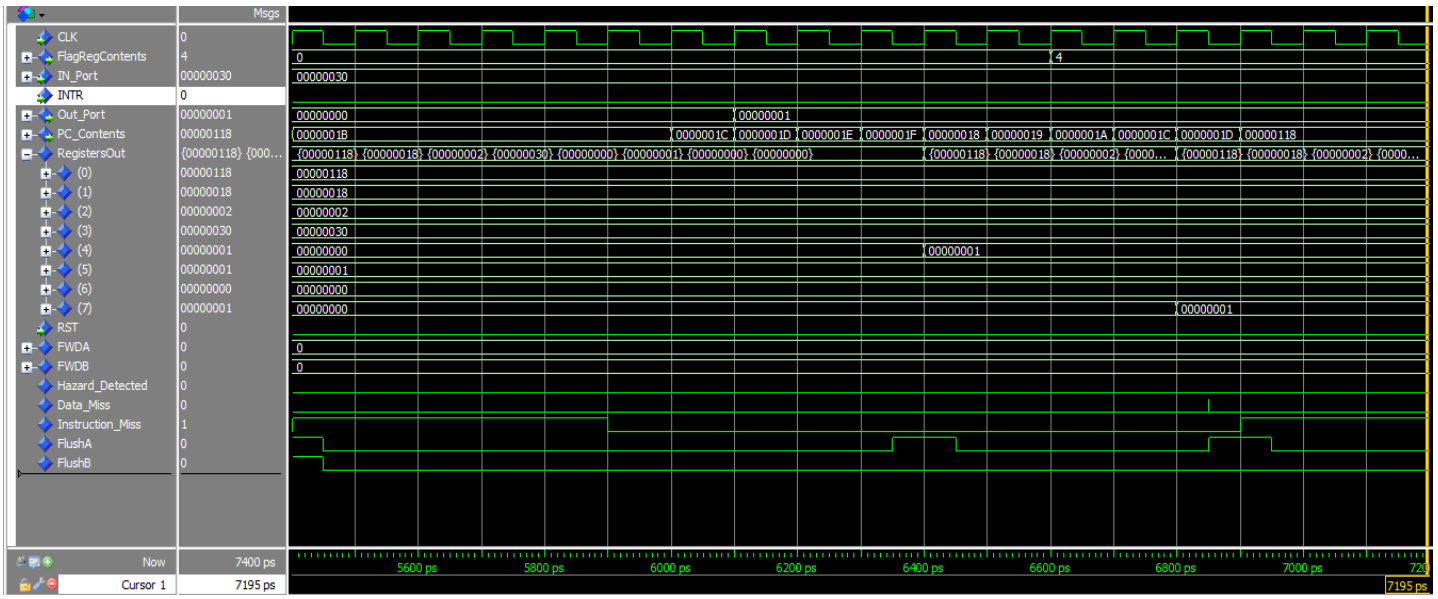


shown the instruction status and data status in the cache controller are as follow after those instructions:

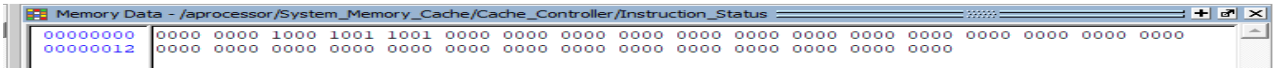


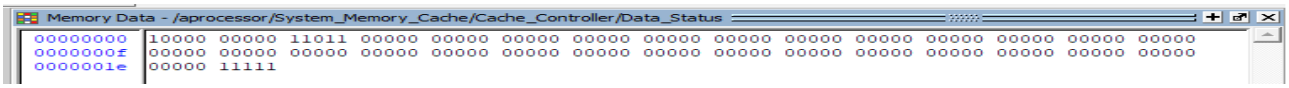


Cycles from 54 to 72:

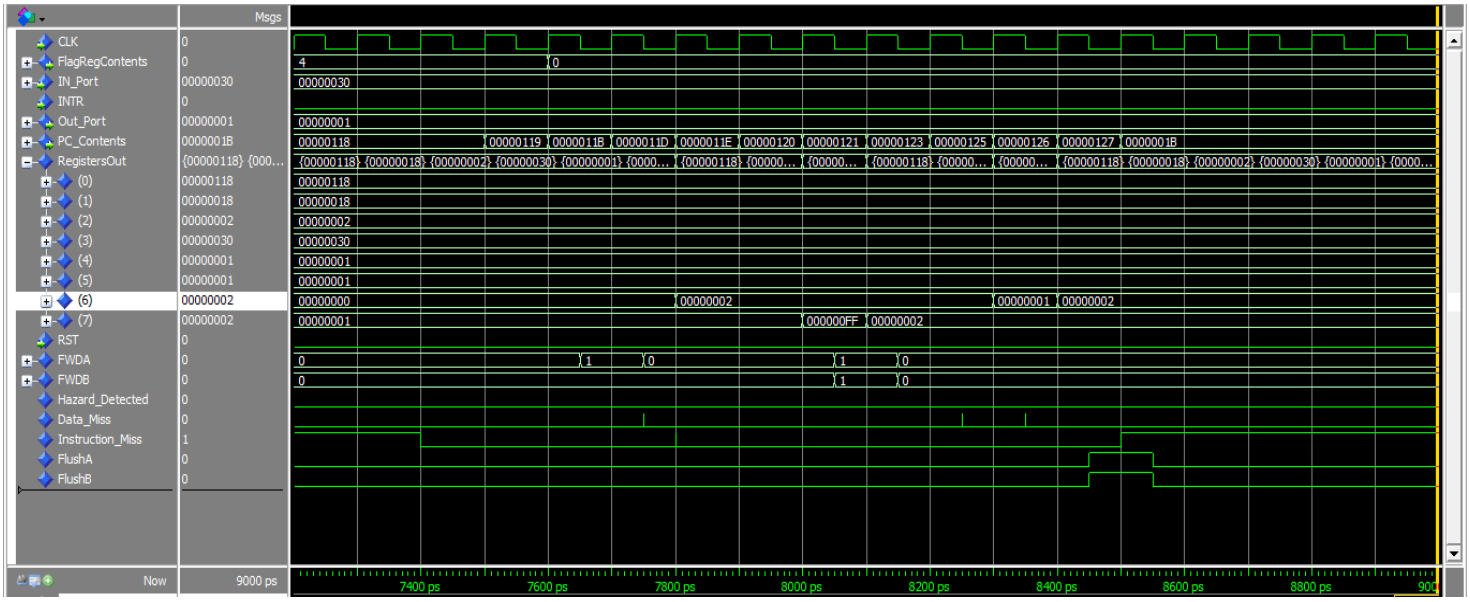


shown the instruction status and data status in the cache controller are as follow after those instructions:

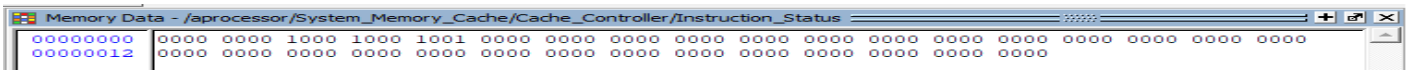


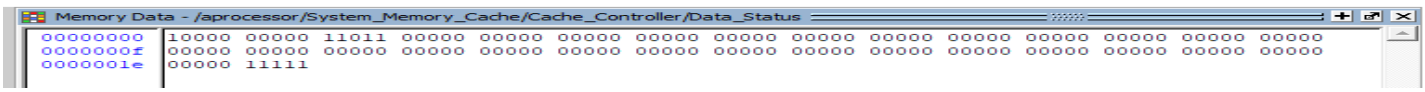


Cycles from 72 to 90:

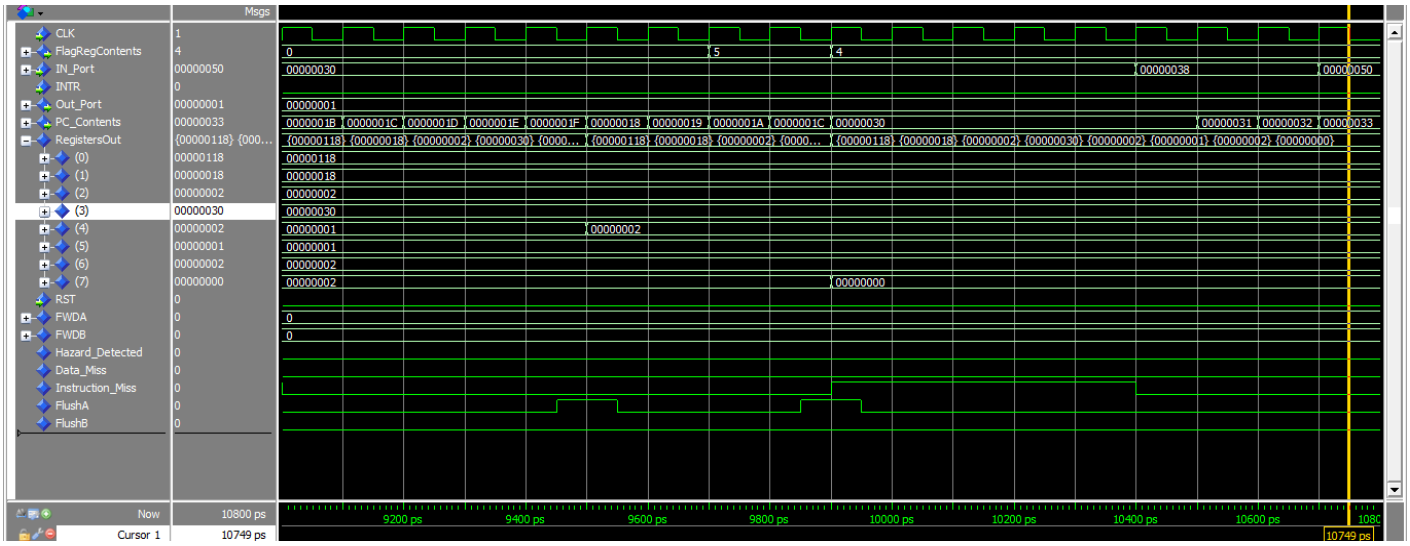


shown the instruction status and data status in the cache controller are as follow after those instructions:

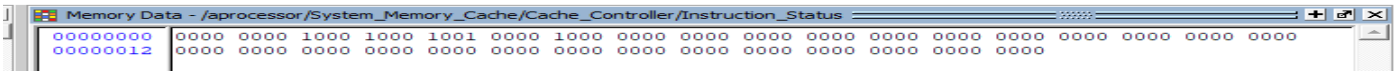


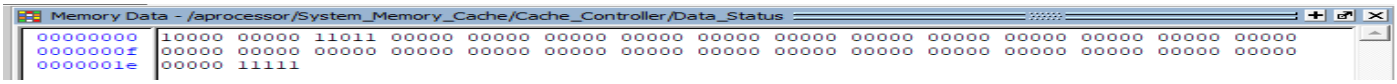


Cycles from 90 to 108:

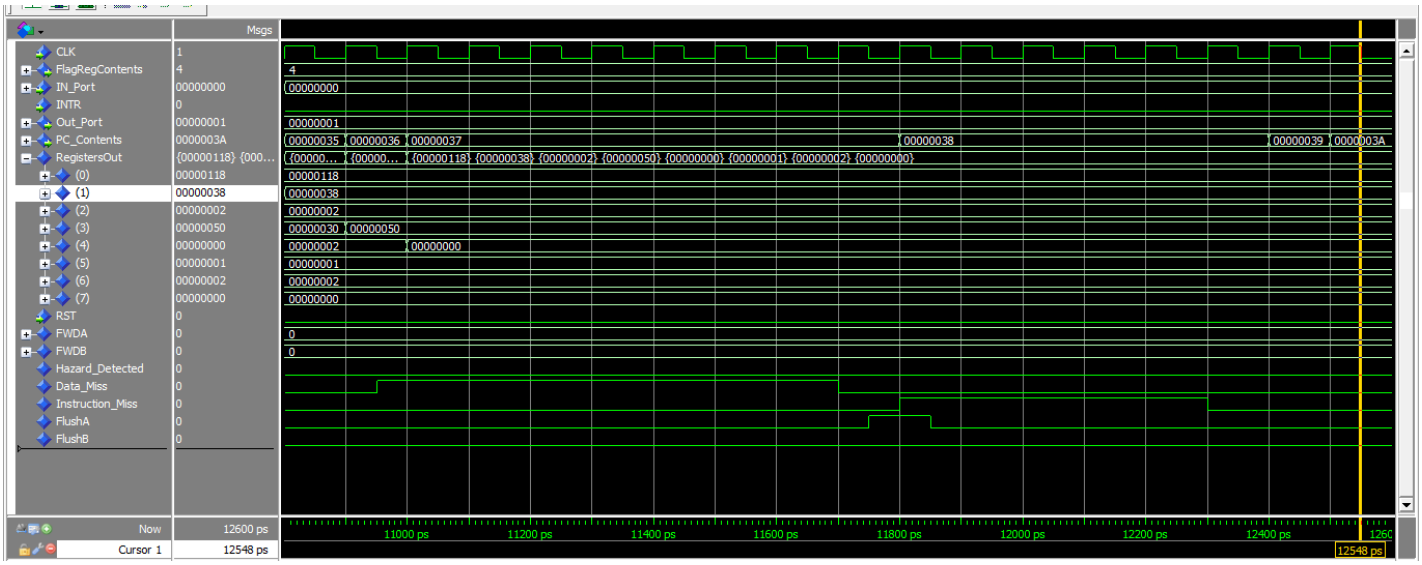


shown the instruction status and data status in the cache controller are as follow after those instructions:

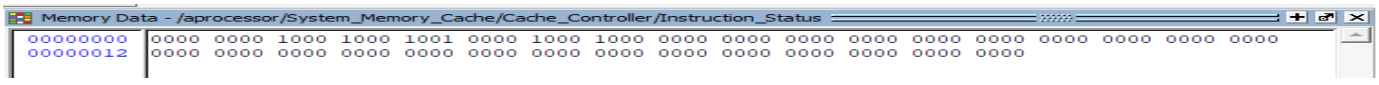


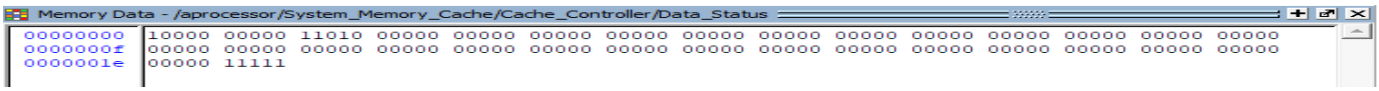


Cycles from 108 to 126:

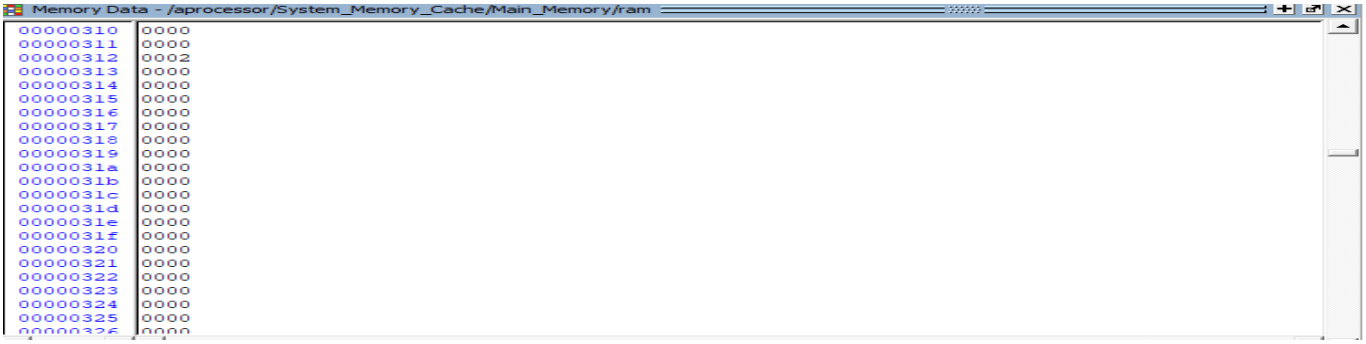


shown the instruction status and data status in the cache controller are as follow after those instructions:

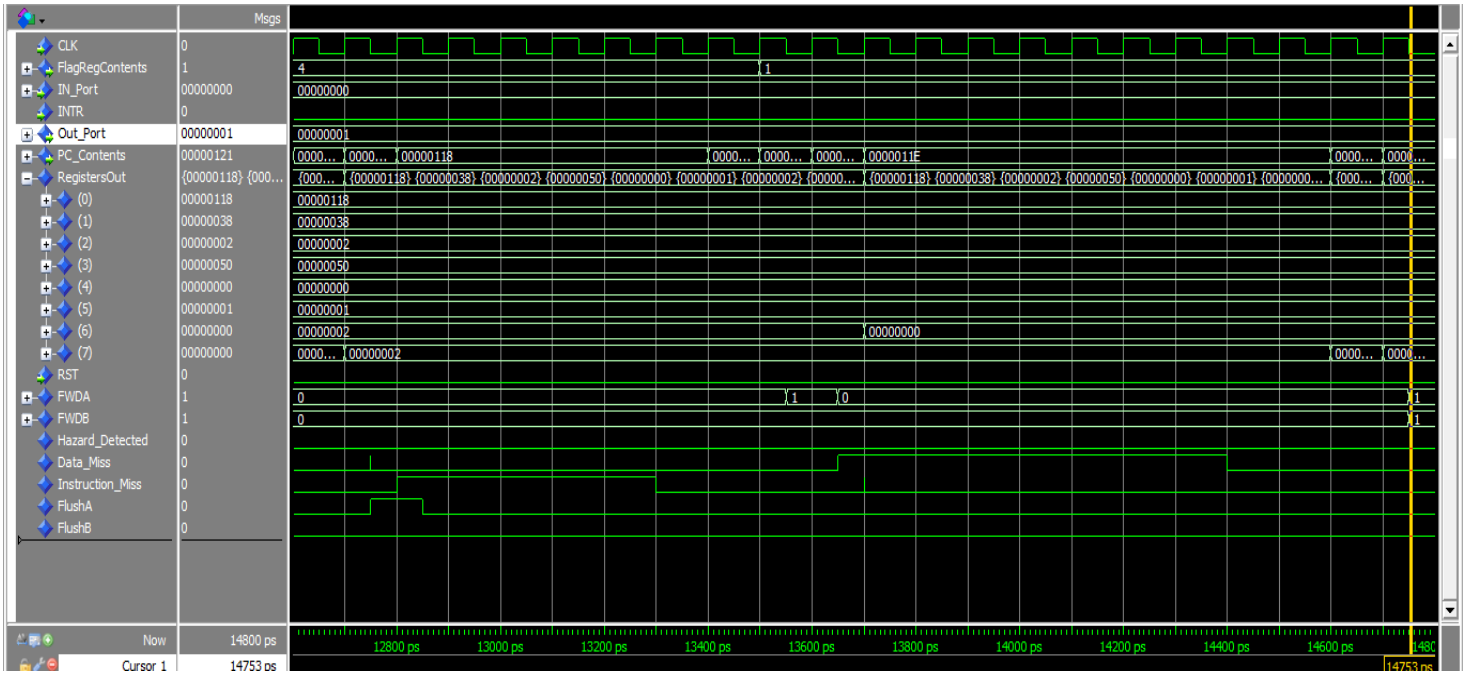




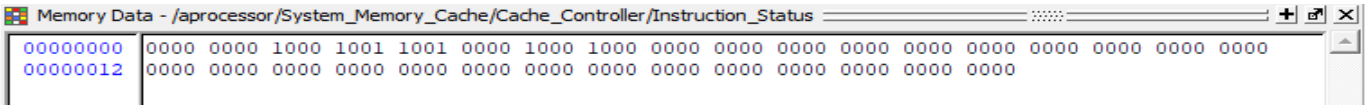
Dirty block written back to the main memory:

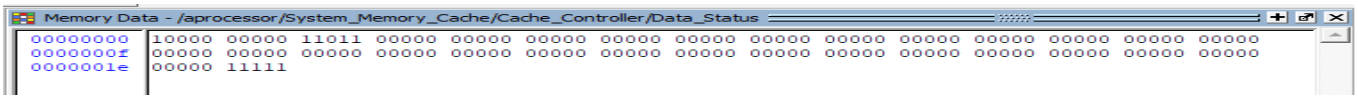


Cycles from 126 to 148 (to show the write back in main memory):



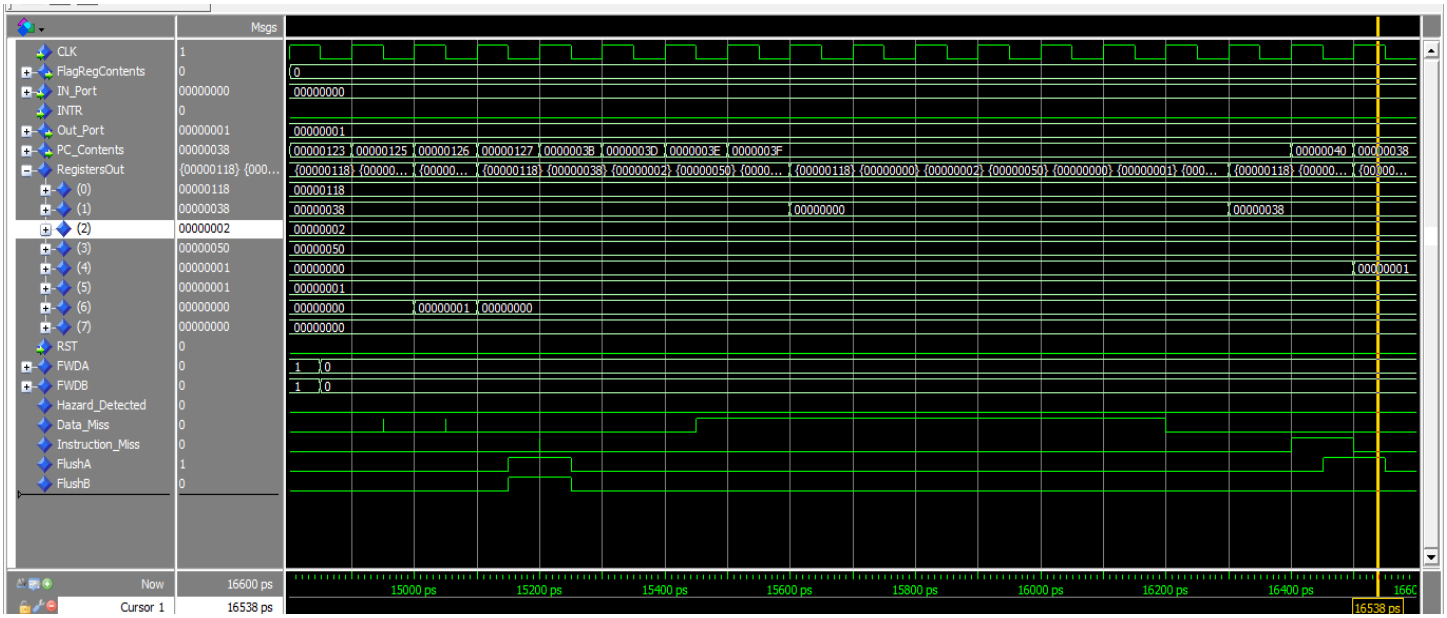
shown the instruction status and data status in the cache controller are as follow after those instructions:



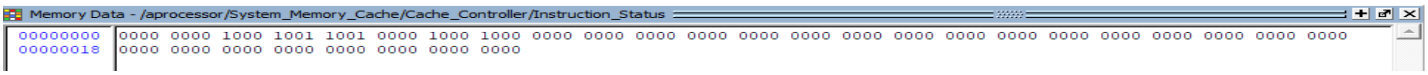


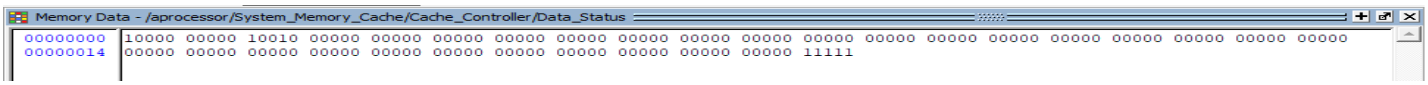
Dirty block written back to the main memory:

Cycles from 148 to 166: (Writing back zeros here is meaningless so there is no error here but we don't stall write back stage in order to handle interrupts correctly so it will keep writing garbage till it write the true value).

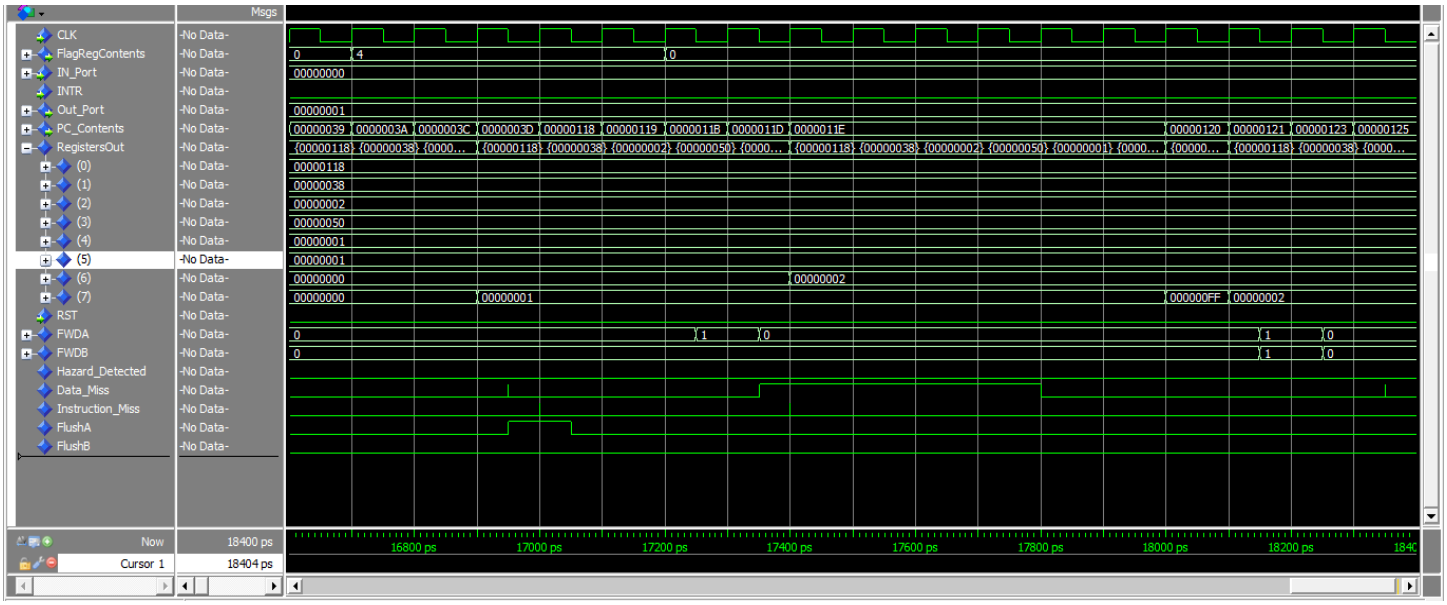


shown the instruction status and data status in the cache controller are as follow after those instructions:

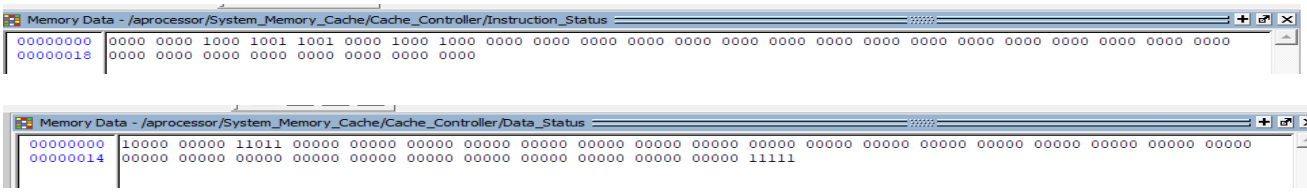




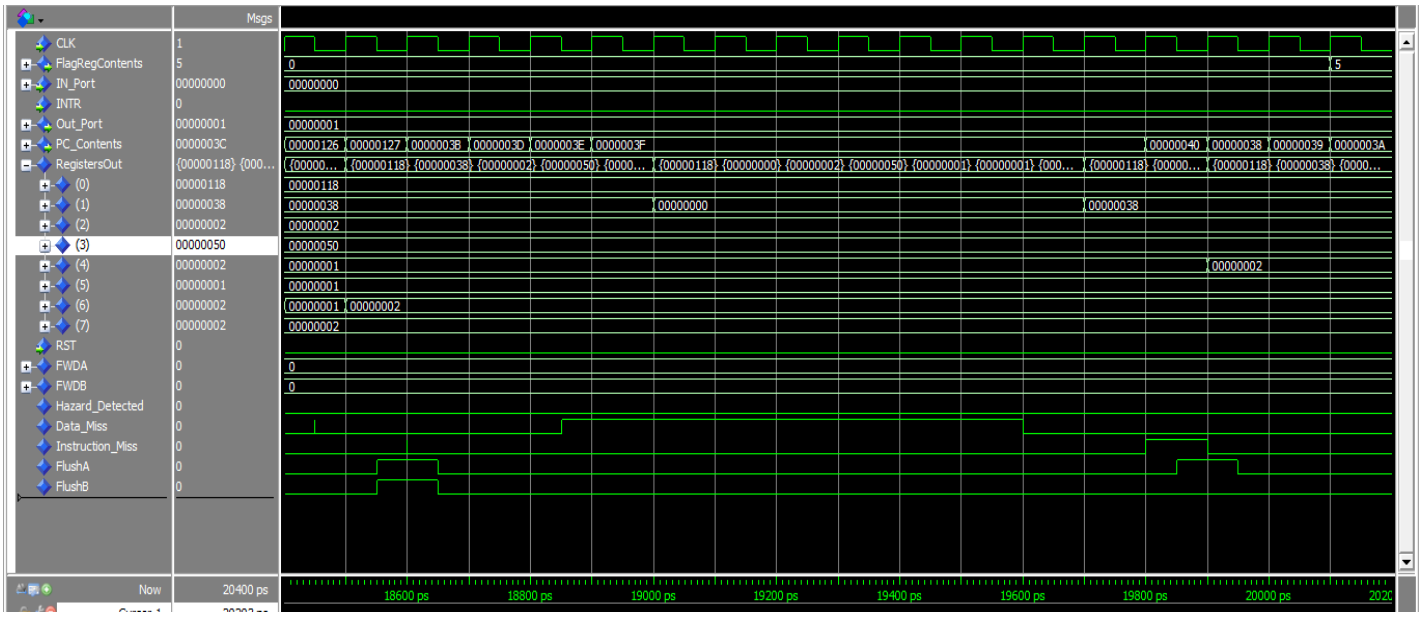
Cycles from 166 to 184:



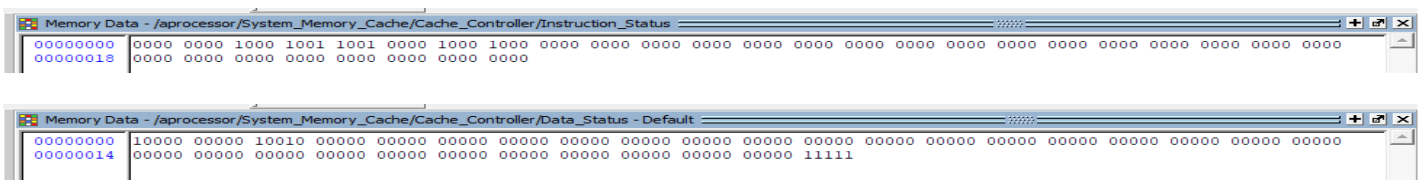
shown the instruction status and data status in the cache controller are as follow after those instructions:



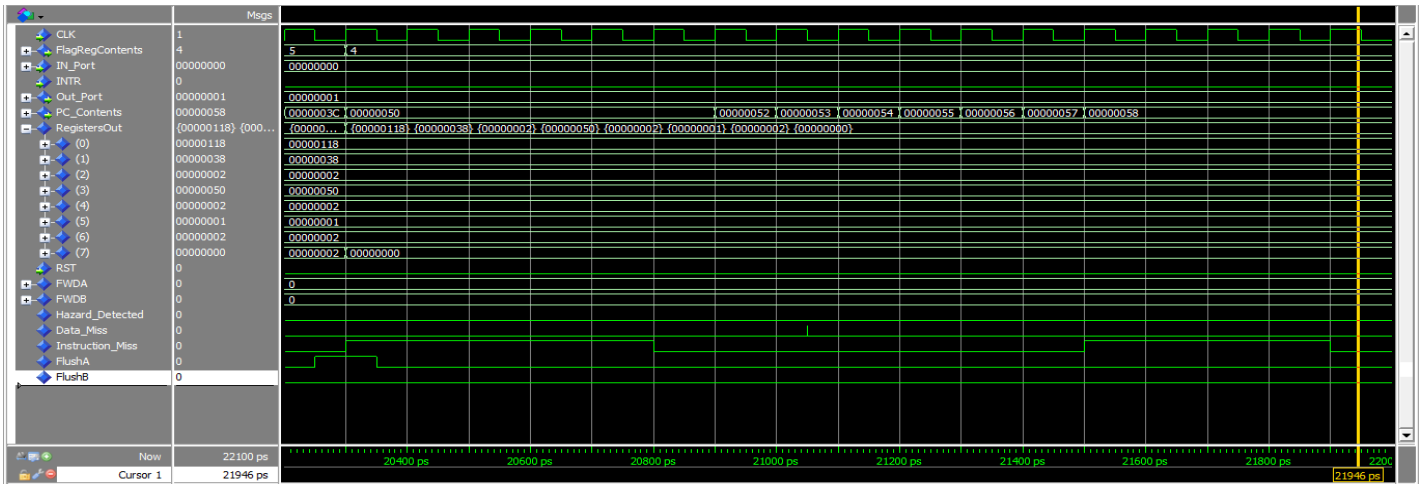
Cycles from 184 to 202:



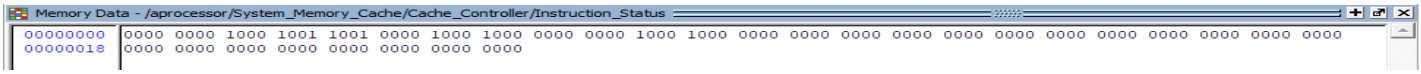
shown the instruction status and data status in the cache controller are as follow after those instructions:

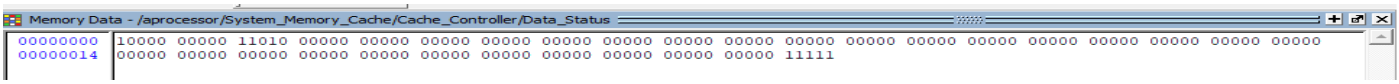


Cycles from 202 to 220:



shown the instruction status and data status in the cache controller are as follow after those instructions:





Data cache at index 2 final values:

Conclusion:

As we can see the effectiveness of hazard detection unit ,forwarding unit and branch prediction as they have increased the CPI but also if we have used software solutions it could have increased the CPI more and more and memory cache system also has reduced accessing the memory evrey time in 4 cycles.