Lab Report #4

Name: Ruochen Duan
Student ID: 1405106

① Introduction

In the lab 4, we need to implement the pipelined ARM CPU as shown in figure1.

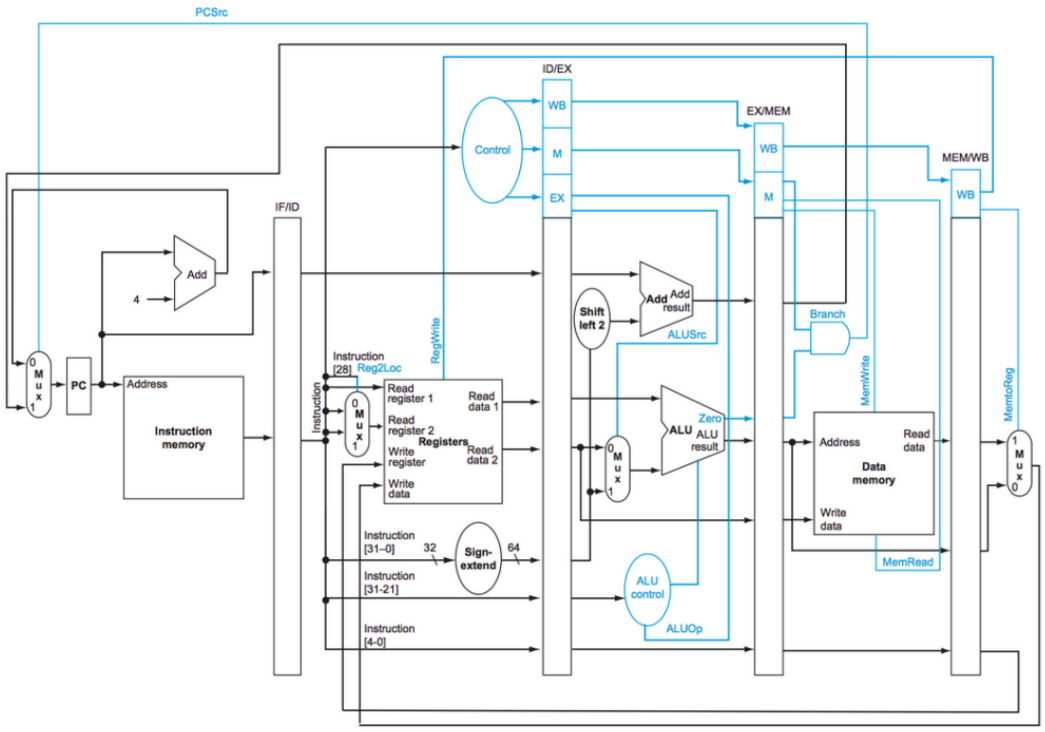


Figure 1: **Schematic of Pipelined ARM (LEGv8) processor for Lab 3.** This is Figure 4.50 in the textbook

The difference between pipelined CPU and the single-cycle CPU in the Lab3 is that in the Lab4 we need to build 4 additional registers, which are IF/ID, ID/EX, EX/MEM and MEM/WB, to store the data produced in each stage and send data to devices or the next register in each cycle. After creating these registers, we are able to wire these registers to each devices like figure 1.

② Registers

This part shows the code of each registers mentioned above.

1. IF/ID

```vhdl
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity IF_ID is

port(     clk : in STD_LOGIC;
          rst : in STD_LOGIC;
          IF_PC_in:in STD_LOGIC_VECTOR(63 downto 0);
          IF_Instruction_in:in STD_LOGIC_VECTOR(31 downto 0);
          ID_PC_out:out STD_LOGIC_VECTOR(63 downto 0);
          ID_Instruction_out:out STD_LOGIC_VECTOR(31 downto 0)
    );
end IF_ID;

architecture behav1 of IF_ID is
begin
     process(clk,rst)
     begin
           if clk = '1' and clk'event then
              ID_PC_out<=IF_PC_in;
              ID_Instruction_out <=IF_Instruction_in;
           end if;
     end process;
end behav1;

```

2. ID/EX

E:/Modelism/ee126/lab4/ID_EX.vhd - Default

```vhdl
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  ID_EX is
port (
clk : in STD_LOGIC;
rst : in STD_LOGIC;
ID_WB_RegWrite_in:in STD_LOGIC;
ID_WB_MemtoReg:in STD_LOGIC;
ID_MEM_Ubranch:in STD_LOGIC;
ID_MEM_Cbranch:in STD_LOGIC;
ID_MEM_MemRead:in STD_LOGIC;
ID_MEM_MemWrite:in STD_LOGIC;
ID_EX_ALUop:in STD_LOGIC_VECTOR(1 downto 0);
ID_EX_ALUsrc:in STD_LOGIC;
ID_PC_in: in STD_LOGIC_VECTOR(63 downto 0);
ID_RD1: in STD_LOGIC_VECTOR(63 downto 0);
ID_RD2: in STD_LOGIC_VECTOR(63 downto 0);
ID_SignExtend: in STD_LOGIC_VECTOR(63 downto 0);
ID_Instruction31_21: in STD_LOGIC_VECTOR(10 downto 0);
ID_Instruction4_0: in STD_LOGIC_VECTOR(4 downto 0);
```

Ln#

```vhdl
EX_WB_RegWrite_in:out STD_LOGIC;
EX_WB_MemtoReg:out STD_LOGIC;
EX_MEM_Ubranch:out STD_LOGIC;
EX_MEM_Cbranch:out STD_LOGIC;
EX_MEM_MemRead:out STD_LOGIC;
EX_MEM_MemWrite:out STD_LOGIC;
EX_EX_ALUop:out STD_LOGIC_VECTOR(1 downto 0);
EX_EX_ALUsrc:out STD_LOGIC;
EX_PC_in: out STD_LOGIC_VECTOR(63 downto 0);
EX_RD1: out STD_LOGIC_VECTOR(63 downto 0);
EX_RD2: out STD_LOGIC_VECTOR(63 downto 0);
EX_SignExtend: out STD_LOGIC_VECTOR(63 downto 0);
EX_Instruction31_21: out STD_LOGIC_VECTOR(10 downto 0);
EX_Instruction4_0: out STD_LOGIC_VECTOR(4 downto 0)
 );
end ID_EX;

```

E:/Modelism/ee126/lab4/ID_EX.vhd - Default

```
Ln#
41  architecture behav1 of ID_EX is
42  begin
43        process(clk,rst)
44           begin
45                  if rst ='1' then
46  EX_WB_RegWrite_in  <='0';
47  EX_WB_MemtoReg     <='0';
48  EX_MEM_Ubranch     <='0';
49  EX_MEM_Cbranch     <='0';
50  EX_MEM_MemRead     <='0';
51  EX_MEM_MemWrite    <='0';
52  EX_EX_ALUop        <="00";
53  EX_EX_ALUsrc       <='0';
54  EX_PC_in           <=x"0000000000000000";
55  EX_RD1             <=x"0000000000000000";
56  EX_RD2             <=x"0000000000000000";
57  EX_SignExtend      <=x"0000000000000000";
58  EX_Instruction31_21<="00000000000";
59  EX_Instruction4_0  <="00000";
60                elsif clk = '1' and clk'event then
61                    EX_WB_RegWrite_in<=ID_WB_RegWrite_in;
62                    EX_WB_MemtoReg   <=ID_WB_MemtoReg;
63                    EX_MEM_Ubranch   <=ID_MEM_Ubranch;
64                    EX_MEM_Cbranch   <=ID_MEM_Cbranch;
65                    EX_MEM_MemRead   <=ID_MEM_MemRead;
66                    EX_MEM_MemWrite  <=ID_MEM_MemWrite;
67                    EX_EX_ALUop      <=ID_EX_ALUop;
68                    EX_EX_ALUsrc     <=ID_EX_ALUsrc;
69                    EX_PC_in         <=ID_PC_in;
70                    EX_RD1           <=ID_RD1;
71                    EX_RD2           <=ID_RD2;
72                   EX_SignExtend    <=ID_SignExtend;
73                   EX_Instruction31_21<=ID_Instruction31_21;
74                   EX_Instruction4_0  <=ID_Instruction4_0;
75                end if;
76     end process;
77  end behav1;
78
```

3. EX/MEM

```
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   entity EX_MEM is
5   port(
6   clk : in STD_LOGIC;
7   rst : in STD_LOGIC;
8   EX_WB_RegWrite_in:in STD_LOGIC;
9   EX_WB_MemtoReg:in STD_LOGIC;
10  EX_MEM_Ubranch:in STD_LOGIC;
11  EX_MEM_Cbranch:in STD_LOGIC;
12  EX_MEM_MemRead:in STD_LOGIC;
13  EX_MEM_MemWrite:in STD_LOGIC;
14  EX_AdderResult:in STD_LOGIC_VECTOR(63 downto 0);
15  EX_Zero:in STD_LOGIC;
16  EX_ALUResult:in STD_LOGIC_VECTOR(63 downto 0);
17  EX_RD2: in STD_LOGIC_VECTOR(63 downto 0);
18  EX_Instruction4_0: in STD_LOGIC_VECTOR(4 downto 0);
19
20  MEM_WB_RegWrite_in:out STD_LOGIC;
21  MEM_WB_MemtoReg:out STD_LOGIC;
22  MEM_MEM_Ubranch:out STD_LOGIC;
23  MEM_MEM_Cbranch:out STD_LOGIC;
24  MEM_MEM_MemRead:out STD_LOGIC;
25  MEM_MEM_MemWrite:out STD_LOGIC;
26  MEM_AdderResult:out STD_LOGIC_VECTOR(63 downto 0);
27  MEM_Zero:out STD_LOGIC;
28  MEM_ALUResult:out STD_LOGIC_VECTOR(63 downto 0);
29  MEM_RD2:out STD_LOGIC_VECTOR(63 downto 0);
30  MEM_Instruction4_0:out STD_LOGIC_VECTOR(4 downto 0)
31     );
32  end EX_MEM;
```

```
architecture behavl of EX_MEM is
begin
      process(clk,rst)
      begin  if rst ='1' then
MEM_WB_RegWrite_in      <='0';
MEM_WB_MemtoReg         <='0';
MEM_MEM_Ubranch         <='0';
MEM_MEM_Cbranch         <='0';
MEM_MEM_MemRead         <='0';
MEM_MEM_MemWrite        <='0';
MEM_AdderResult         <=x"0000000000000000";
MEM_Zero                <='0';
MEM_ALUResult           <=x"0000000000000000";
MEM_RD2                 <=x"0000000000000000";
MEM_Instruction4_0      <="00000";
           elsif rising_edge(clk) then
MEM_WB_RegWrite_in      <=EX_WB_RegWrite_in;
MEM_WB_MemtoReg         <=EX_WB_MemtoReg;
MEM_MEM_Ubranch         <=EX_MEM_Ubranch;
MEM_MEM_Cbranch         <=EX_MEM_Cbranch;
MEM_MEM_MemRead         <=EX_MEM_MemRead;
MEM_MEM_MemWrite        <=EX_MEM_MemWrite;
MEM_AdderResult         <=EX_AdderResult;
MEM_Zero                <=EX_Zero;
MEM_ALUResult           <=EX_ALUResult;
MEM_RD2                 <=EX_RD2;
MEM_Instruction4_0      <=EX_Instruction4_0;
         end if;
  end process;
end behavl;
```

4. MEM/WB

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity MEM_WB is
port(
clk : in STD_LOGIC;
rst : in STD_LOGIC;
MEM_WB_RegWrite_in:in STD_LOGIC;
MEM_WB_MemtoReg:in STD_LOGIC;
MEM_RD:in STD_LOGIC_VECTOR(63 downto 0);
MEM_ALUResult:in STD_LOGIC_VECTOR(63 downto 0);
MEM_Instruction4_0: in STD_LOGIC_VECTOR(4 downto 0);

WB_WB_RegWrite_in:out STD_LOGIC;
WB_WB_MemtoReg:out STD_LOGIC;
WB_RD:out STD_LOGIC_VECTOR(63 downto 0);
WB_ALUResult:out STD_LOGIC_VECTOR(63 downto 0);
WR: out STD_LOGIC_VECTOR(4 downto 0)
);
end MEM_WB;
architecture behavl of MEM_WB is
begin
     process(clk,rst)
        begin
               if rst='1' then
WB_WB_RegWrite_in<='0';
WB_WB_MemtoReg   <='0';
WB_RD            <=x"0000000000000000";
WB_ALUResult     <=x"0000000000000000";
WR               <="00000";
           elsif rising_edge(clk) then

WB_WB_RegWrite_in<=MEM_WB_RegWrite_in;
WB_WB_MemtoReg   <=MEM_WB_MemtoReg;
WB_RD            <=MEM_RD ;
WB_ALUResult     <=MEM_ALUResult;
WR               <=MEM_Instruction4_0;
          end if;
       end process;
end behavl;
```

③ Test Result

We can use the same testbench used in the Lab3. The simulation we do is:

```
ADD X11, X9, X10        10001011000010100000000100101011
STUR X11, XZR,0         11111000000000000000001111101011
SUB X12, X9, X10        11001011000010100000000100101100
STUR  X11, [XZR,0]      11111000000000000000001111101011
STUR  X12, [X12,8]      11111000000000001000000110001100
STUR  X12, [X12,8]      11111000000000001000000110001100
ORR X21, X19, X20       10101010000010100000001001110101
NOP
NOP
STUR X21, [XZR,0]       11111000000000000000001111110101
NOP
NOP
NOP
NOP
LSR X21, X19, X20       11010011010101000000001001110101
```

And the initialized values in DMEM and registers are:

Registers

```
$X9  = 0x0000000000000010
$X10 = 0x0000000000000008
$X11 = 0x0000000000000002
$X12 = 0x000000000000000A
$X19 = 0x00000000CEA4126C
$X20 = 0x000000001009AC83
$X21 = 0x0000000000000000
$X22 = 0x0000000000000000
```

DMEM

```
DMEM(0x0)  = 1
DMEM(0x8)  = 2
DMEM(0x16) = 3
DMEM(0x24) = 4
```

DMEM values are in Hex.

Test Result(I set each cycle equals to 100 ns.):

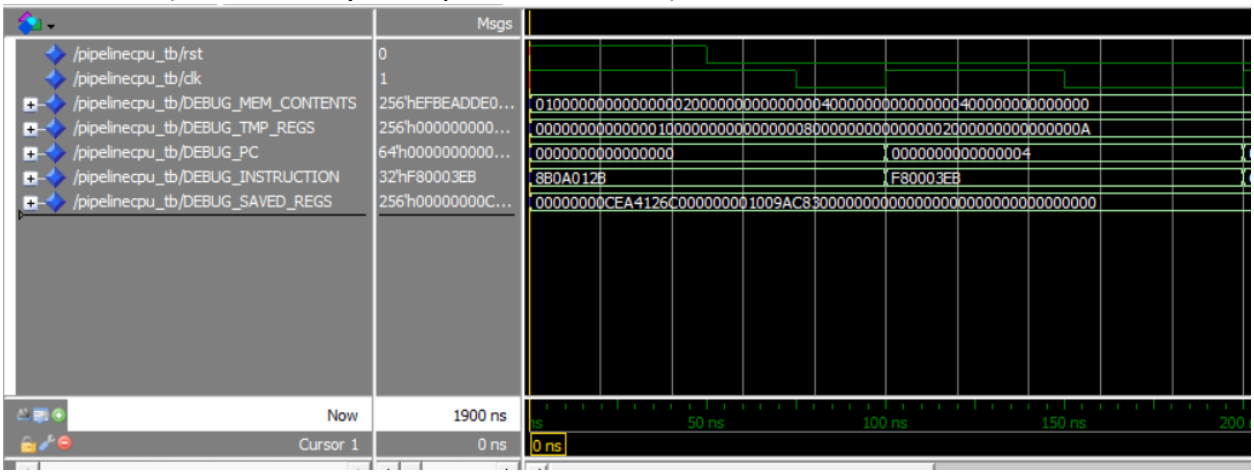


Figure 1(0-200ns)

In the cycle 1, during t=0 ns to t =75 ns, rst = 1 and clk = 1. So the values of CPU do not change. And at t = 75s I set r clk = 0, and keep its value to the end of the

cycle 1. At t = 50 ns I set rst = 0 and keep its value to the end of the cycle 1. And PC gets the first address of instruction from IMEM which is

10001011000010100000000100101011 = 8B0A 012B

(ADD, X11, X9, X10)

In the cycle 2, PC gets the second address of instruction from IMEM which is

11111000000000000000001111101011 = F800 03EB

(STUR X11, XZR,0)

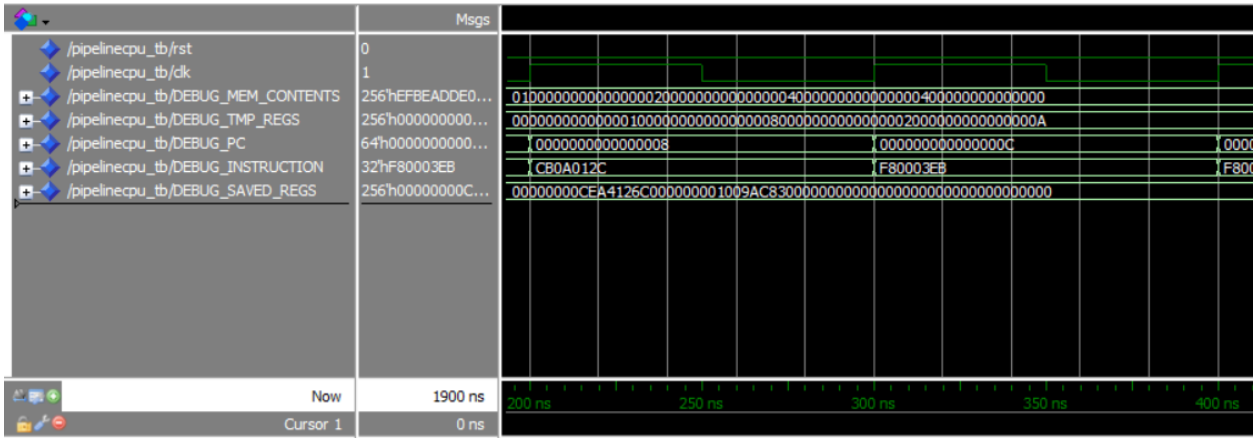


Figure 2(200-400ns)

In the figure 2, during t=200 ns to 400ns, we can see that nothing changed except PC and Instruction.

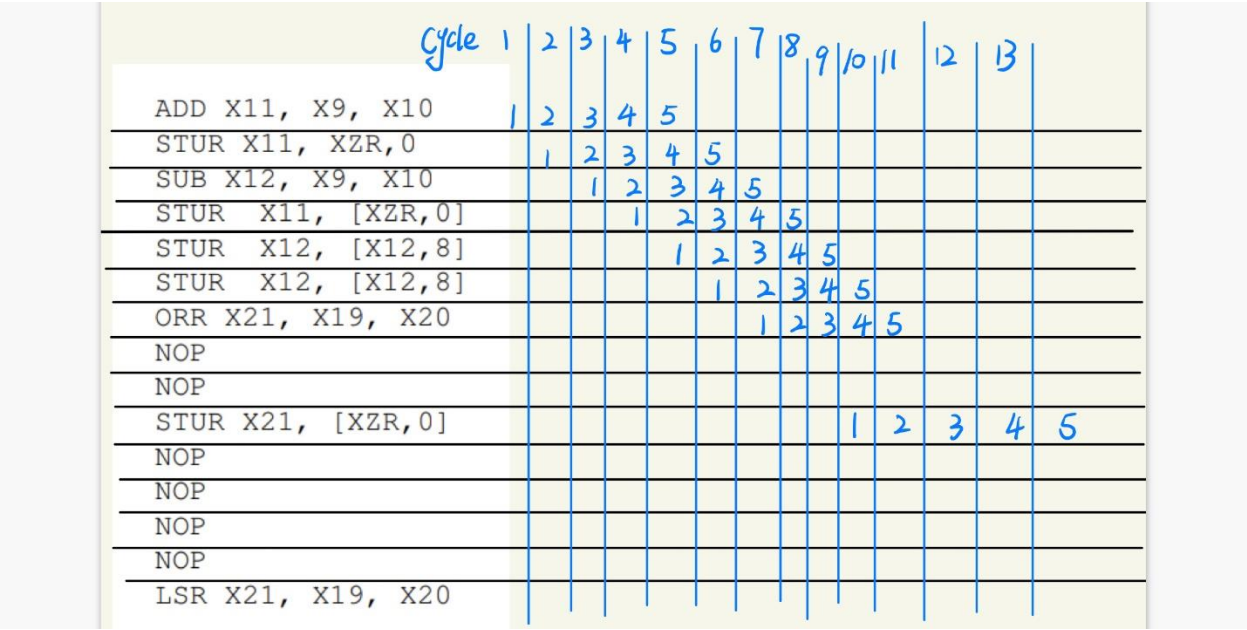


Figure 3(Timing diagram)

Figure3 is the timing diagram for this simulation, With this figure we can more easily interpret the test results.

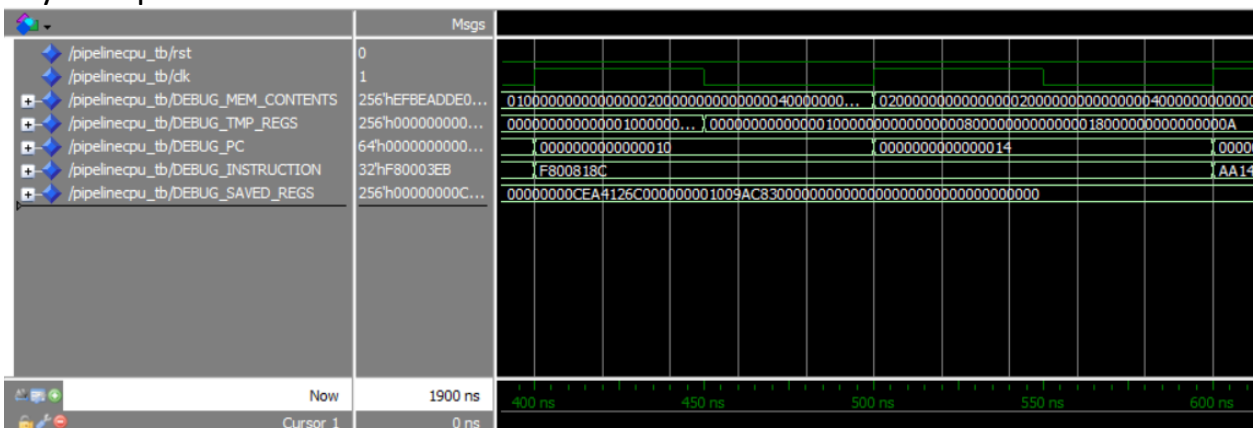


Figure 4(400-600ns)

In the cycle 5(400 ns -500ns), we note that at t =450 ns, the value of temp registers changed from

```
sim:/pipelinecpu_tb/DEBUG_TMP_REGS @ 368 ns
256'h0000000000000010000000000000000800000000000000020000000000000000A
```

To

```
sim:/pipelinecpu_tb/DEBUG_TMP_REGS @ 466 ns
256'h0000000000000010000000000000000800000000000000018000000000000000A
```

Because we have finished the instruction ADD, X11, X9, X10, the value of X11 is changed to the result X9+X10.

In the cycle 6 we can see that at t=500 ns, the value of DMEM changed from

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 475 ns
256'h0100000000000000020000000000000040000000000000004000000000000000
```

to

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 525 ns
256'h0200000000000000020000000000000040000000000000004000000000000000
```

Because we store value of X11 to address 0 of the DMEM. However we do not store the result of (ADD, X11, X9, X10), but the initialized value of X11, because according to figure 3,we get data of X11 in cycle2, but at that time the first instruction had not finished, the result of it was not written back to register. So, we get data hazard at this point.

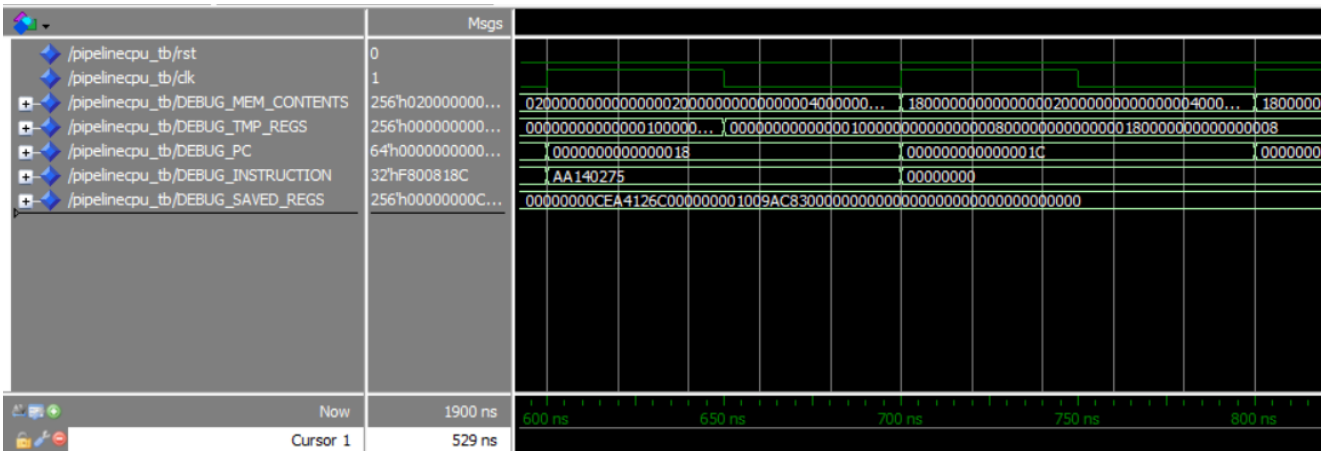


Figure 5(600-800ns)

In the cycle 7(600 ns-700 ns), we note that at t =650 ns, the value of temp registers changed from

```
sim:/pipelinecpu_tb/DEBUG_TMP_REGS @ 636 ns
256'h0000000000000010000000000000008000000000000001800000000000000A
```

to

```
sim:/pipelinecpu_tb/DEBUG_TMP_REGS @ 659 ns
256'h0000000000000010000000000000008000000000000001800000000000000008
```

Because we have finished the instruction SUB, X12, X9, X10, the value of X11 is changed to the result X9-X10.

In the cycle 8(700 ns -800ns) we can see that at t=700 ns, the value of DMEM changed from

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 680 ns
256'h020000000000000002000000000000000400000000000000040000000000000
```

to

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 722 ns
256'h180000000000000002000000000000000400000000000000040000000000000
```

Because we store value of X11 to address 0 of the DMEM. According to figure 3,we get data of X11 in cycle5, and in the cycle 5 the X11 had been already written , So we store a different data (result of first instruction) in DMEM. At this point, we do not have data hazard.

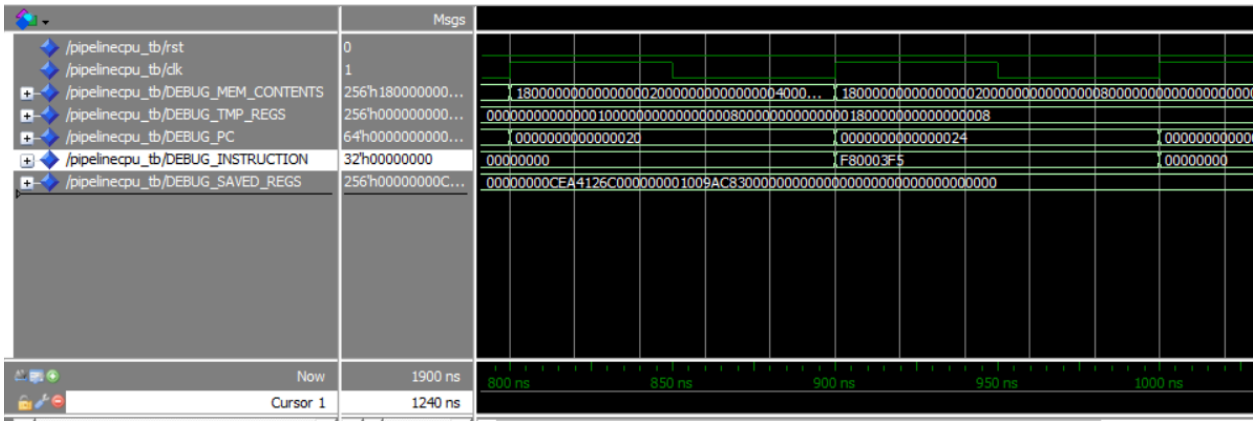


Figure 6(800-1000ns)

In the cycle 9(800 ns -900ns) we can see that at t=800 ns, the value of DMEM changed from

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 773 ns
256'h1800000000000000020000000000000004000000000000004000000000000000
```

to

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 828 ns
256'h1800000000000000020000000000000004000A000000000000000000000000000
```

Because we store value of X12 from dmemBytes(16) to dmemBytes(23) of the DMEM. However we do not store the result of (SUB, X12, X9, X10), but the initialized value of X12, because according to figure 3,we get data of X12 in cycle6, but at that time the third instruction had not finished, the result of it was not written back to register. So, we get data hazard at this point.

In the cycle 10(900 ns -1000ns) we can see that at t=900 ns, the value of DMEM changed from

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 874 ns
256'h1800000000000000020000000000000004000A000000000000000000000000000
```

to

```
sim:/pipelinecpu_tb/DEBUG_MEM_CONTENTS @ 919 ns
256'h1800000000000000020000000000000008000000000000000000000000000000000
```

Because we store value of X12 from dmemBytes(16) to dmemBytes(23) of the DMEM. According to figure 3, we get data of X12 in cycle7, and in the cycle 7 the X12 had been already written, So we store a different data (result of first instruction) in DMEM. At this point, we do not have data hazard.

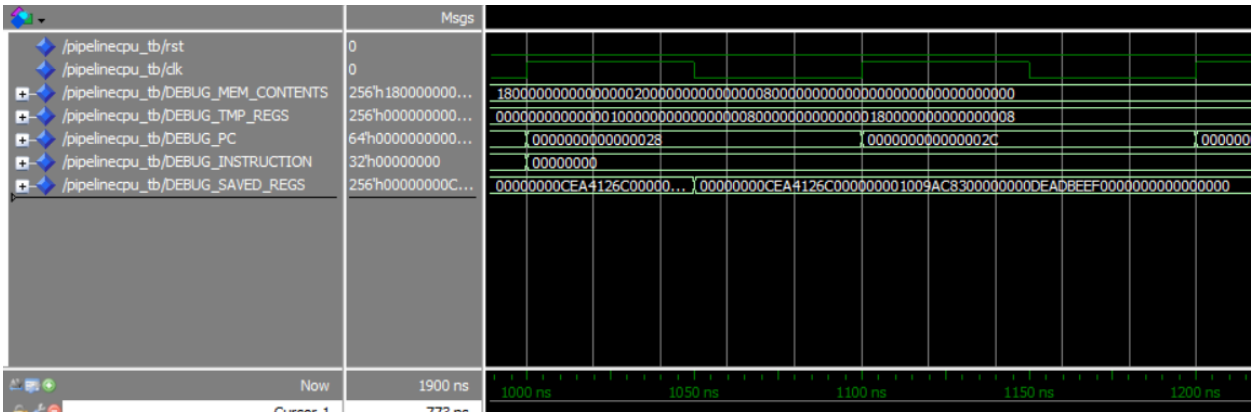
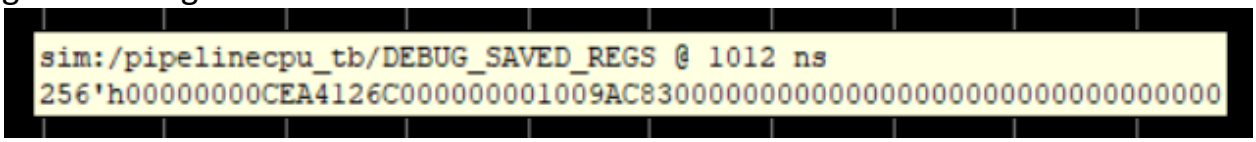


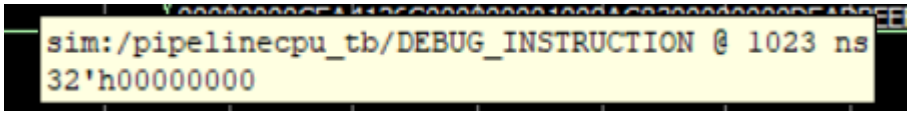
Figure 7(1000-1200ns)

In the cycle 11(1000-1100 ns), we can see that at t=1050 ns, the value of saved registers changed from



to

Because at cycle 11 we finished the instruction ORR, X21, X19,20. And we execute the NOP instruction.



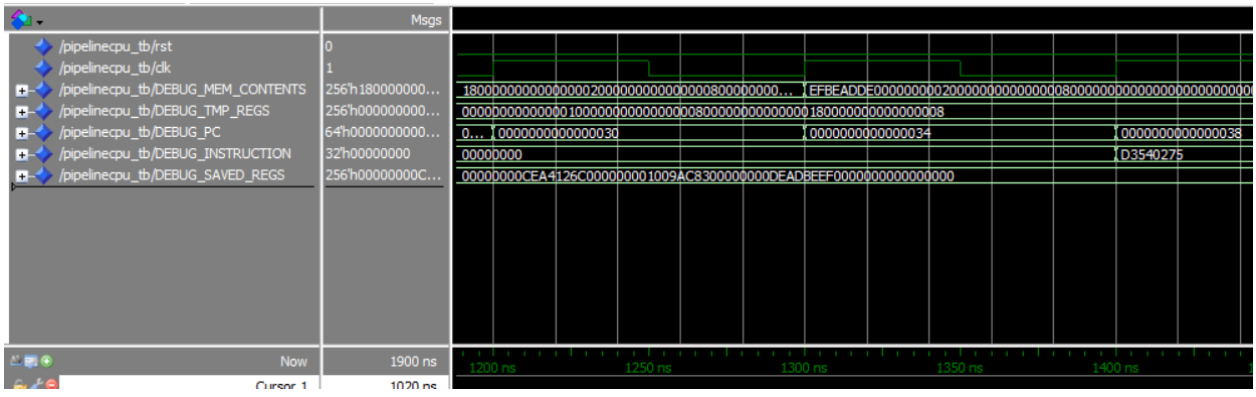
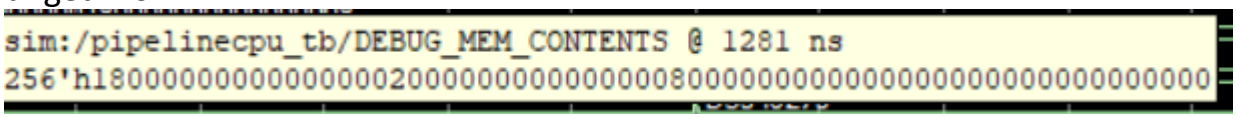


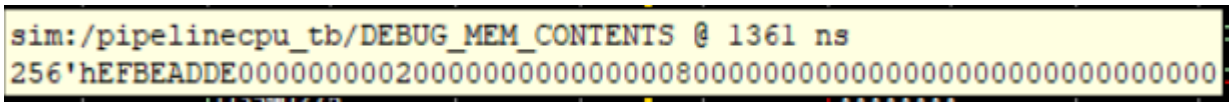
Figure 7(1200-1400ns)

In the cycle 14 (1300-1400ns), we can see that at t=1300 ns, the value of DMEM

changed from



to



Even though at cycle 14, we had not finished all stage of the instruction ORR, X21, X19,20, the value of X21 had been changed, because we did not need to write back from DMEM (the fifth stage). So, we store right data to DMEM.

Figure 8(1000-1400ns)

Figure 8 shows value of the PC and instruction. We can see that in the cycle11 to cycle 14, we execute the NOP instruction.