Computer Architecture Project MS4 Report

Due:11th May 2020

Name: Omar Mahdy

ID: 900160493

Name: Mohamed Ashraf Taha

ID: 900172754

- **Project Goals & Design:**

In this submission we try and implement a pipeline RISCV processor supporting All RV32I Base Instruction with CSR, Fence, and FenceI implemented as nops and Ecall implemented as a termination instruction ,Set as well as eight instructions from the RV32M Standard Extension (Multiplication & Division). We also, move the branch outcome and target address computation to the ID stage and handle the resulting data hazards.Our main goal is for our implementation to work correctly on all our implemented instructions and for it to work on all combinations of inputs. We achieve this by testing for all possible combinations that we can think of for every instruction. Also, we aim to make our code readable as much as possible. We achieve this by simply trying to label all wires and reg appropriately especially in the top module titled Full_Datapath. You can see that most of the wires and registers and modules are named exactly the same like that one in the lectures slides and our block diagram or with similar naming making it easier for you to know which wire corresponds to that in the diagram and all Multiplexers are labeled as MUX's with a name to signify what they are for. Also, you will find that all outputs in the top module of any module will have the word "out" in them to show that they are outputs for a certain module, except if they are used for a selection line for something then they will have the word "sel" in them. Moreover, for human readability we try and used the defined bits given in the define file. We do this to make our code less prone to human errors, as well as make it as much readable as possible. Lastly, we try and use more case statements than if and else statements to make the code easier to read and compile. Another thing we try to avoid doing is using the shorthand if's instead of MUXS to make the code clearer. In addition, we try and implement we as little hardware cost as possible in our capabilities. For instance, you will find that if a single bit is not needed for a case or if statement, we will not take, this can be seen clearly in the ALU,the Data memory and ALU control modules. In this pipelined implementation we aim to make all instructions work and avoid data, control and structural hazards from the single ported memory and do through testing.

• Implementation

In this pipelined implementation we try and carry out thorough testing for all instructions as well as implement 2 bonus features, we support RV32M Standard Extension (Multiplication & Division) and we also move the branch outcome to the Decode stage instead of the memory stage and handle the resulting data hazards. Firstly, we support the RV32M instructions by adding their opcode to the control unit and then choose the appropriate control signals accordingly we then move to the ALU control unit and create a custom ALU selection signals ,one for multiplication and 2 others one for division and one for the modulus operation. In the ALU module we then compare the function 3 of the instruction to determine what kind of RV32M instruction is and do the appropriate operation to it. Moreover, we moved the branch outcome to the decode stage by creating a comparator module in the decode stage that takes the branch and function 3 of the instruction to know what kind of branch instruction it is. The comparator then outputs a branch signal that is 1 when branch is taken and 0 when it is not. This signal is now the selection line for us to jump to a branched address or not. We then handle data hazards by creating a new custom forwarding unit in the decode stage that takes the branch signal and regwrite and rd from the mem stage we then take RS1 and RS2 from the decode stage and if either RS1 or RS2 is equal to rd and regwrite equal 1 and rd not equal 0 we set forwarding signal to 1 and we use a mux with selection line the forwarding signal and its chooses to the output from the alu for the mem stage to either RD1 or RD2 in the decode stage.

• Testing Procedure and Results

We tested by adding the machine code, the 32 bit instruction in the instruction memory and commented the equivalent assembly code next to it. We got the machine code from the RARS emulator we wrote the instruction we needed and got the equivalent 32-bit instruction, as well as the expected output for a certain instruction from it. We tested and checked all 47 instructions as well as all instructions from the RV32M

Standard Extension and did thorough testing on them testing all possible combinations that we thought of on all instructions we support, we tested extensively data hazards and load use hazards as well, we always end our testing with ecall instruction for termination . Below is our Results.

- **Result**

Arithmetic Test:

```
// addi x0,x0,x0        // testcases
//addi x1,x0,2      //x1=x0+2=2                       x1=2
//addi x2,x0,-20    //x2=x0+(-20)= -20                x2=-20
//addi x3,x0,7      //x3=x0+7=7                       x3=7
//addi x4,x0,-8     //x4=x0+(-8)= -8                  x4=-8
//sub x5, x3,x1     //x5=7-2=5                        x5=5
//sub x6, x1,x3     //x6=2-7 = -5                     x6=-5
//sub x7, x2,x4     //x7=(-20)-(-8)= -12              x7=-12
//sub x8, x7,x3     //x8=(-12)-7= -19                 x8=-19
                    //FORWARDING case 1
//sub x9, x1 ,x8    //x9=2-(-19)= 21                  x9=21
               //FORWARDING case 2
//SLLI x10,x9,1     //x10=21<<1= 42                   x10=42
                    //FORWARDING case 3
//lw x11, 0(x0)     //x11=writes 10                   x11=10
//SRLI x12,x11,1//x12=10>>1= 5                        x12=5
       x12=5        //Load Use hazard CASE 1
//SRAI x13,x4, 1    //x13=(-8)<<<1=-4                 x13=-4
//andi x14,x5, 1    //x14=5&1=1                       x14=1
//andi x15,x5,-1    //x15=5&-1=5                      x15=5
//ORI x16,x14,-25   //x16=1|-25=-25                   x16=-25
//ORI x17,x14,25    //x17=1|24=25                     x17=25
//ORI x18,x2,-8     //x18=(-20)|(-8)=-4               x18=-4
//Xor x19,x17,x1    //x19=25^2=27                     x19=27
//Xor x20,x2,x14    //x20=(-20)^1=-19                 x20=-19
//Xor x21,x2,x7     //x21=(-20)^(-12)=24              x21=24
```

//SLT x22,x1,x3 //x22=1 2<7

x22=1

//SLT x23,x3,x1 //x23=0 7>2

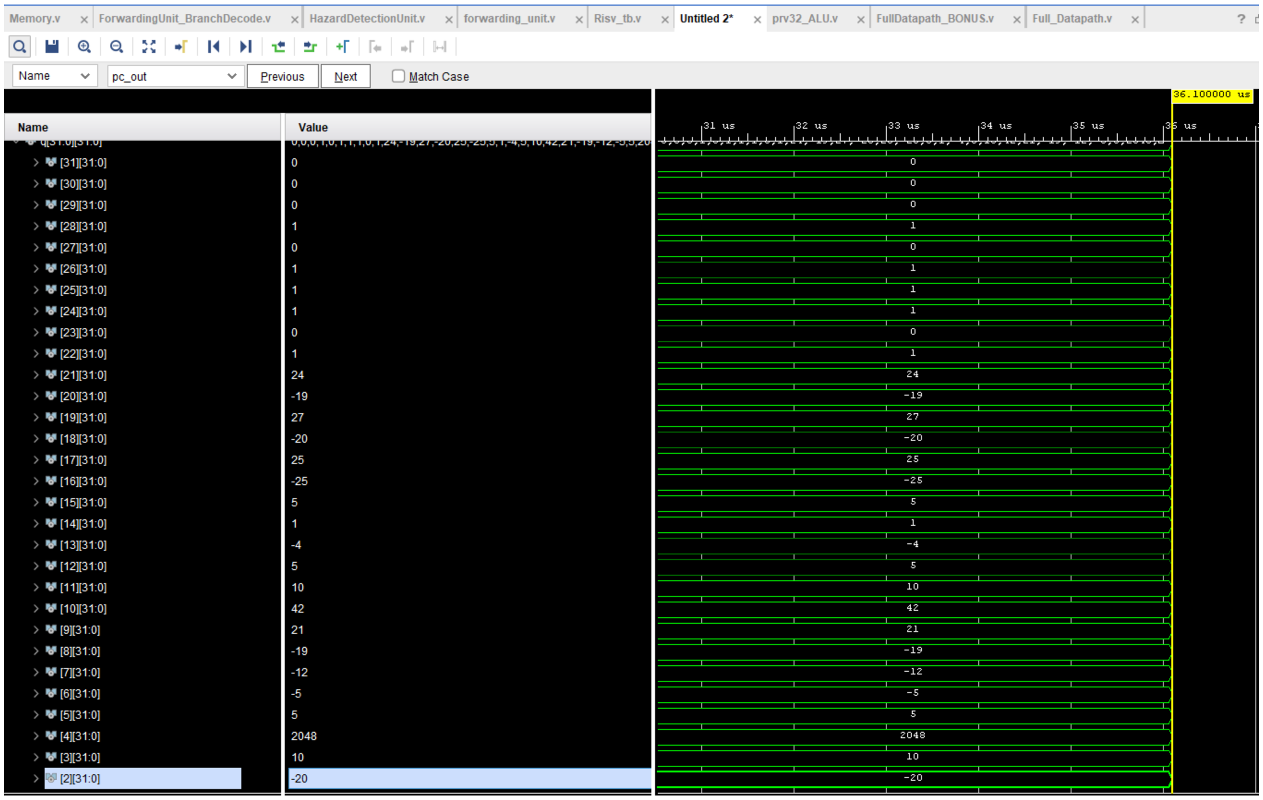
x23=0

//SLT x24,x4,x1 //x24=1 -8<2

x24=1

//SLT x25,x4,x6 //x25=1 -8<5	**x25=1**
//SLT x26,x6,x5 //x26=0 -8<5	**x26=1**
//SLT x27,x1,x4 //x27=0 2>-8	**x27=0**
//SLTu x28,x1,x3 //x28=1 2<7	**x28=1**
//SLTu x29,x3,x1 //x29=0 7>2	**x29=0**
//SLTu x30,x4,x1 //x30=0 8>2	**x30=0**
//SLTu x31,x4,x6 //x31=0 8>5	**x31=0**
//Xori x19,x17,2 //x19=25^2=27	**x19=27**
//Xori x20,x2,1 //x20=(-20)^1=-19	**x20=-19**
//Xori x21,x2,-17 //x21=(-20)^(-17)=3	**x21=3**
//SLTu x1,x6,x5 //x1=1 5<8	**x1=1**

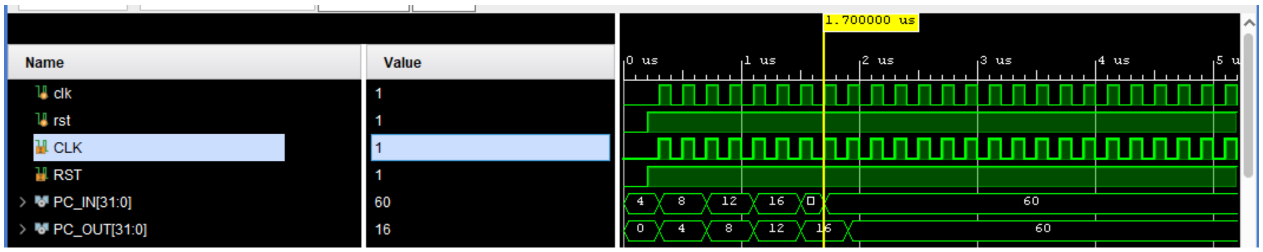
Register File Outputs



```
Name            Value
q[31:0][31:0]   0,0,0,1,0,1,1,1,0,1,24,-19,27,-20,25,-25,5,1,-4,5,10,42,21,-19,-12,-5,5,20
  [31][31:0]    0
  [30][31:0]    0
  [29][31:0]    0
  [28][31:0]    1
  [27][31:0]    0
  [26][31:0]    1
  [25][31:0]    1
  [24][31:0]    1
  [23][31:0]    0
  [22][31:0]    1
  [21][31:0]    24
  [20][31:0]    -19
  [19][31:0]    27
  [18][31:0]    -20
  [17][31:0]    25
  [16][31:0]    -25
  [15][31:0]    5
  [14][31:0]    1
  [13][31:0]    -4
  [12][31:0]    5
  [11][31:0]    10
  [10][31:0]    42
  [9][31:0]     21
  [8][31:0]     -19
  [7][31:0]     -12
  [6][31:0]     -5
  [5][31:0]     5
  [4][31:0]     2048
  [3][31:0]     10
  [2][31:0]     -20
```

Branch Test:
Example

Bge x3,x1,44
We jump from decode stage and fetching starts at PC_Out= 16



Name	Value
clk	1
rst	1
CLK	1
RST	1
PC_IN[31:0]	60
PC_OUT[31:0]	16

<u>All Test Cases will be shown in the demo this is just a sample to show we did through testing.</u>