UNIVERSITY OF MORATUWA

Faculty of Engineering

Registered Module No: CS 4362

Hardware Description Language

Pipelined MIPS Processor

Semester project

Date of Submission:

24/10/2021

170387F - M. W. G. V. Melaka

170524B - R. M. T. S. Ratnayake

Department of Computer Science and Engineering

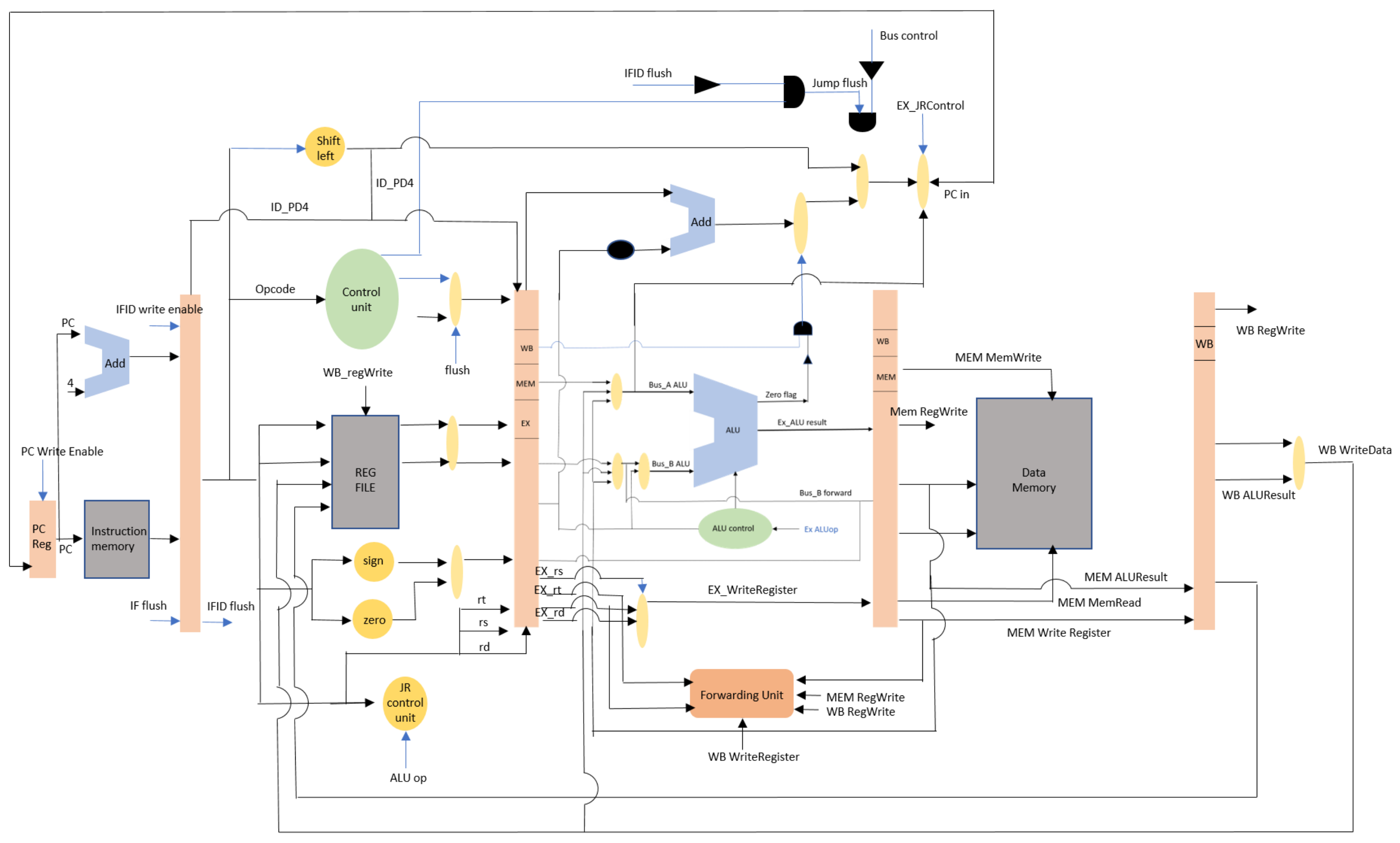
Table of Content

Github Link -
https://github.com/tharaka27/MIPS_32bit_Processor
https://github.com/vihan125/Mips-Processor

Video Link -
https://youtu.be/GdhMIt1_C6Q

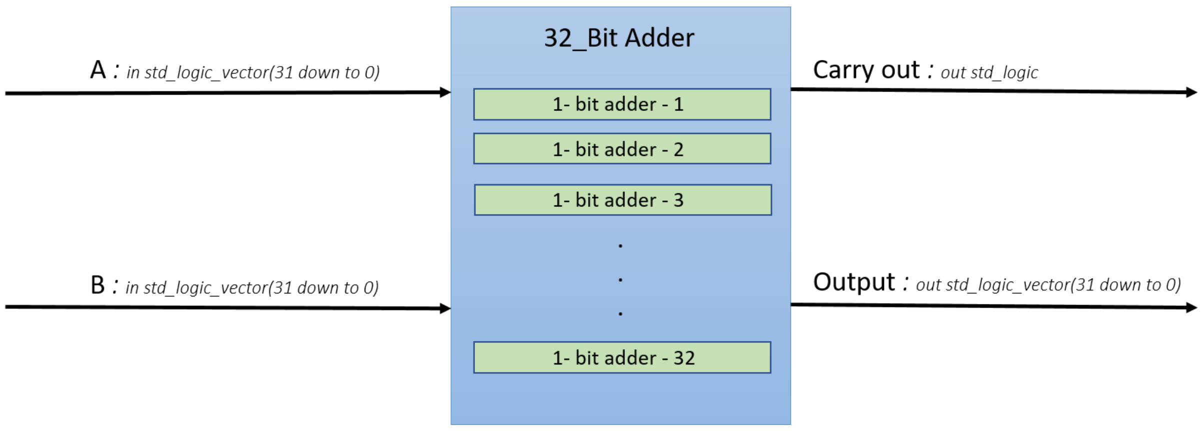
Or
https://drive.google.com/drive/folders/1y1DM5LUwLrQi048qd_ftFo-1Mnc4WK4g?usp=sharing

Architecture of the pipelined processor



32 - bit adder

This module is responsible for adding two 32 bit numbers. This module is made up of 32, 1-bit full adders as shown in the following figure and acts as a ripple carry adder.


32_Bit Adder
A : in std_logic_vector(31 down to 0)
B : in std_logic_vector(31 down to 0)
1- bit adder - 1
1- bit adder - 2
1- bit adder - 3
1- bit adder - 32
Carry out : out std_logic
Output : out std_logic_vector(31 down to 0)

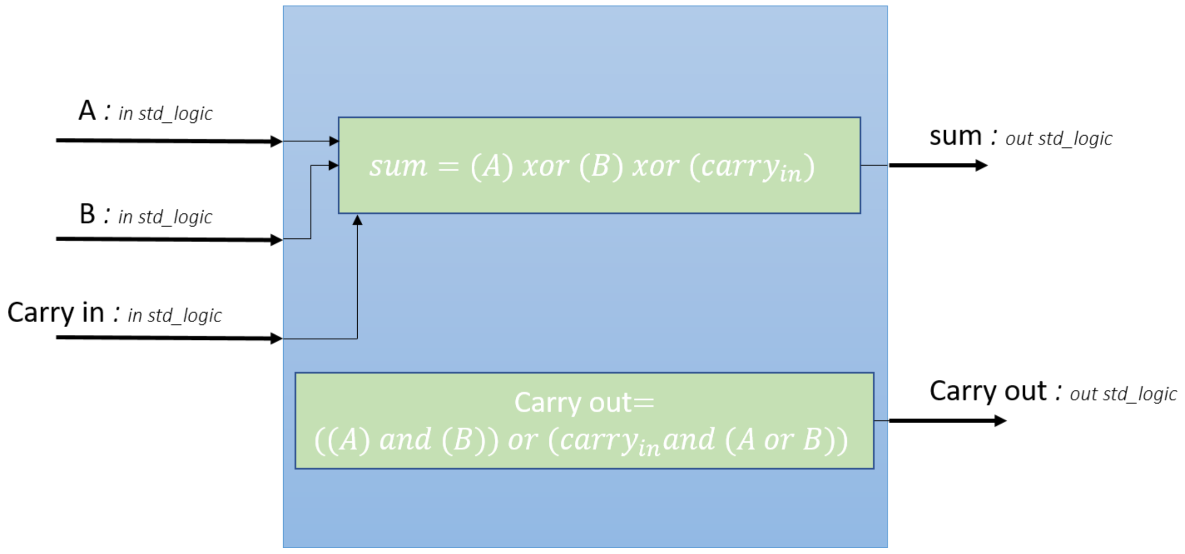
Let's have a closer look at the 1-bit adders which the 32-bit adder is made up of.

1 - bit adder

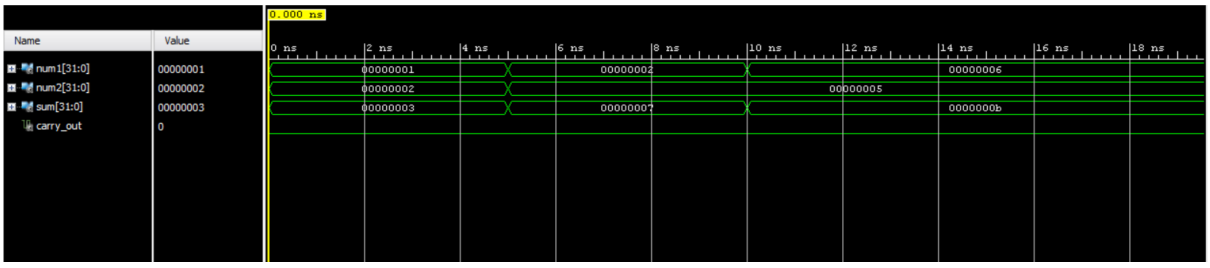
a 1-bit adder is responsible for adding two individual bits and output the result as the sum and the "carry" of the summation using the following equations.

$$\begin{aligned} sum &= (Bit\ 1\ )\ xor\ (Bit\ 2)\ xor\ (carry\ in) \\ carry\ out &= ((Bit\ 1)\ or\ (Bit\ 2))\ or\ ((carry\ in\ )and\ (Bit\ 1\ or\ Bit\ 2)) \end{aligned}$$

Following diagram shows a rough sketch of the 1-bit adder component,


A : in std_logic
B : in std_logic
Carry in : in std_logic
$sum = (A)\ xor\ (B)\ xor\ (carry_{in})$
Carry out=
$((A)\ and\ (B))\ or\ (carry_{in} and\ (A\ or\ B))$
sum : out std_logic
Carry out : out std_logic

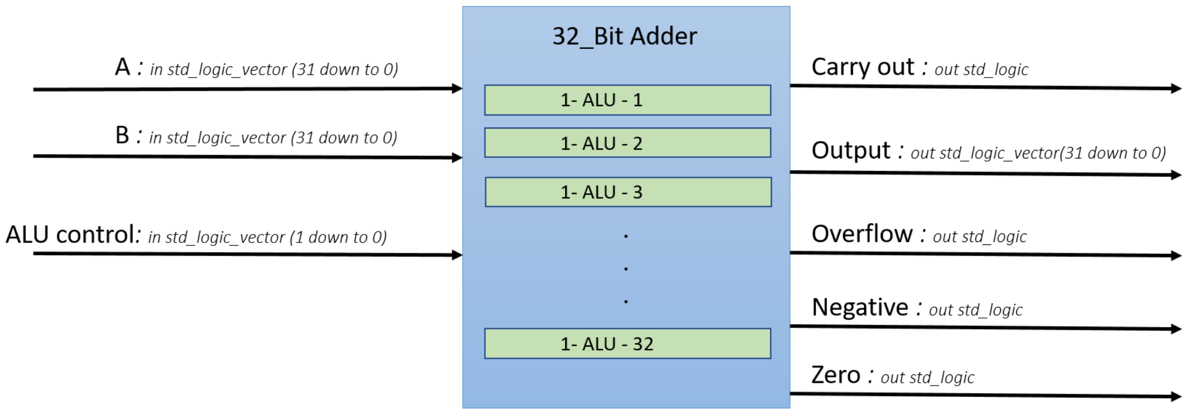
Timing diagram of 32-bit adder



Name	Value
num1[31:0]	00000001
num2[31:0]	00000002
sum[31:0]	00000003
carry_out	0

32-bit Arithmetic and Logic Unit (ALU)

As in the 32-bit adder, the 32-bit Arithmetic and logic unit (ALU) is made up of a collection of 32, 1-bit ALU sub components. Following diagram gives a rough overview of the inputs and outputs of the 32-bit ALU component.


32_Bit Adder
A : in std_logic_vector (31 down to 0)
B : in std_logic_vector (31 down to 0)
ALU control: in std_logic_vector (1 down to 0)
1- ALU - 1
1- ALU - 2
1- ALU - 3
1- ALU - 32
Carry out : out std_logic
Output : out std_logic_vector(31 down to 0)
Overflow : out std_logic
Negative : out std_logic
Zero : out std_logic

ALU performs a single operation out of possible 4 operations that is defined as the "ALU control" input of the module. The operations related to the corresponding bit combinations of the "ALU control bus" is as follows,

ALU control bus	Operation
00	Add / Add-R (values from registers)
10	Sub / Sub-R (values from registers)
11	SLT (set on less than)
01	XOR / XORi (xor with immediate value)

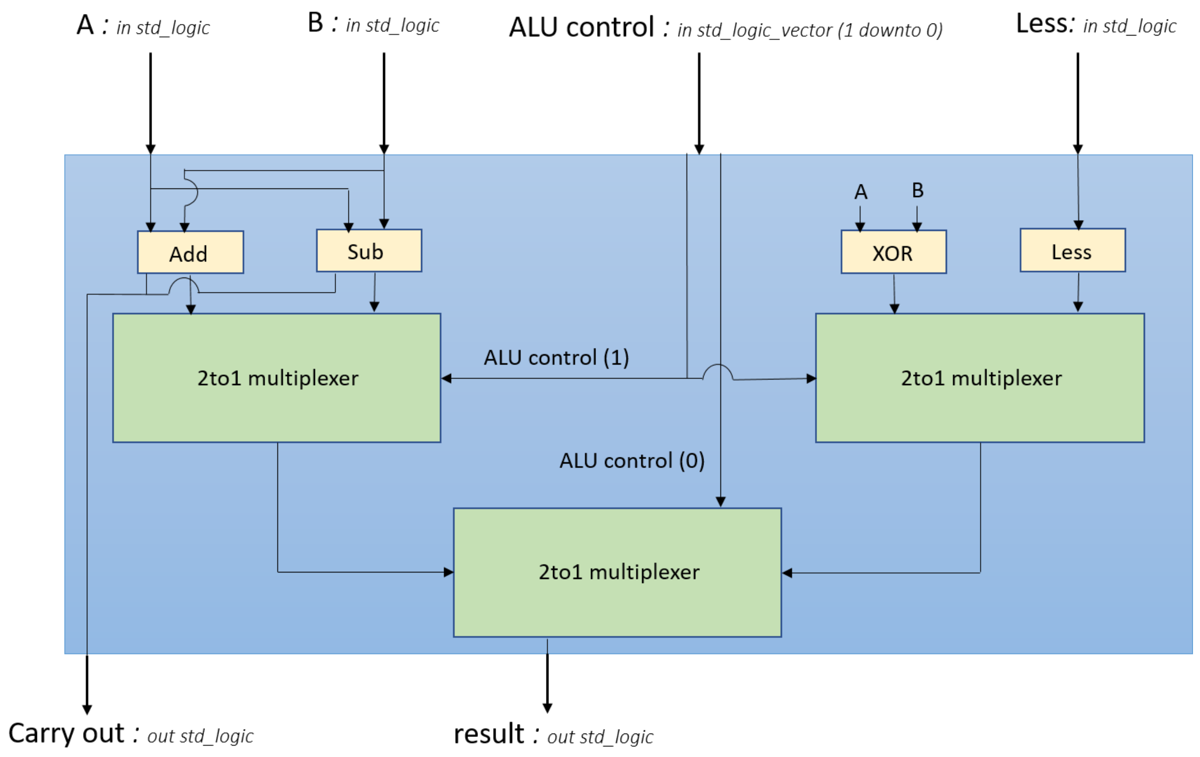
Let's have a closer look at the 1-bit ALU component and the logic that drives inside it.

1-bit Arithmetic and logic unit

Inside the 32-bit ALU unit, the operation that needs to be carried out performs in individual bits inside a 1 bit ALU component. The 1-bit ALU component consists of three 2X1 multiplexers to choose the correct output that is relevant to the operation given by the ALU control bus.

- Multiplexer to choose between "Addition" and "Subtraction" operations.
- Multiplexer to choose between "XOR" and "Less than" operations.
- Multiplexer to choose between results of the first two multiplexers.

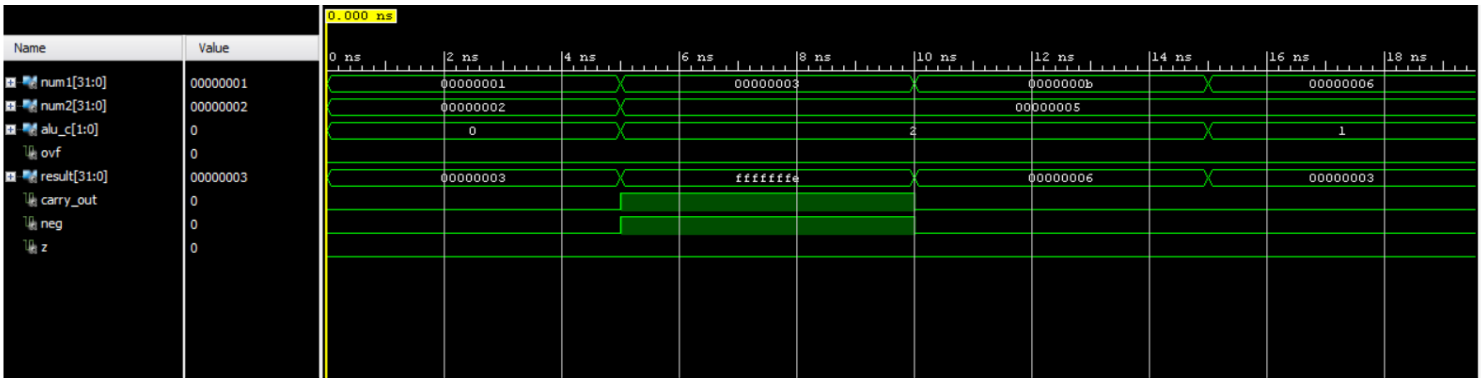
Logic inside the 1-bit ALU on how three multiplexers work is explained in a more elaborated manner from the following diagram,



As shown in the above figure, the bit at position '1' in the ALU control bus decides two operations from the available four operations of the ALU and the final operation is decided by the bit at position '0' in the ALU control bus from the previously selected two operations.

For example if the bit combination at ALU control bus is "01", according to the above diagram first two multiplexers choose operation at position '0' which is "ADD" and "XOR" and then from the third multiplexer "XOR" operation is selected which is in the '1' position of the third multiplexer.

Timing Diagram of 32-bit ALU



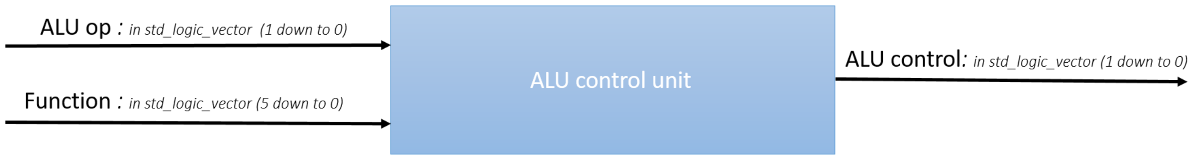
In the above timing diagram, the first clock cycle shows an additional operation which is given as the operation in ALU control bus as “00” (refer table in the above section). And the result of the operation is shown in the result bus as expected (1+2 =3).

Then a subtraction operation is carried out which is indicated by the ALU control bus as “10” (refer table in the above section). In this case the answer of the subtraction is a negative value (3-5 = -2) and it is indicated in the result bus in two’s complement format.

In the final clock cycle “XOR” operation is carried out between 6 and 5 as indicated by the ALU control bus as “01” (refer table in the above section). And the answer 3 is indicated in the results as expected. (“00..00101 XOR “00..00110 ” = “00..00011” == 3).

ALU control unit

ALU control unit is responsible for deciding what the value of the ALU control bus should be depending on the decoded instruction values. Following diagram gives an abstract overview of the inputs and outputs of the ALU control unit component.

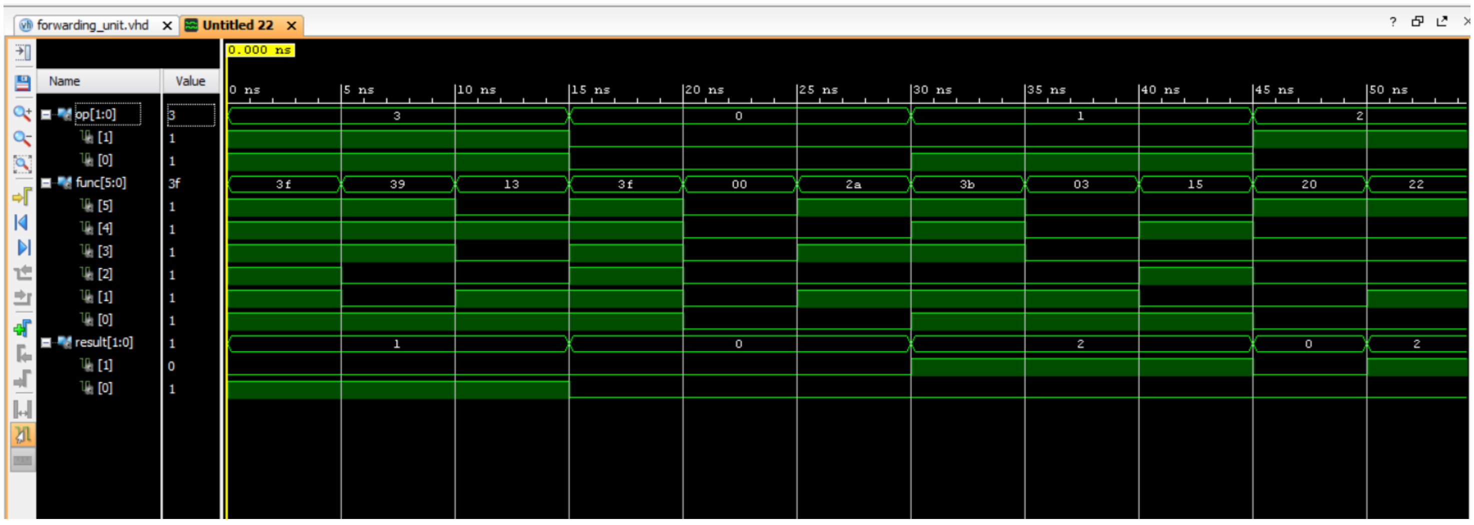

ALU op : in std_logic_vector (1 down to 0)
Function : in std_logic_vector (5 down to 0)
ALU control unit
ALU control: in std_logic_vector (1 down to 0)

After receiving two inputs from decoded instruction data from two buses, one with a length of 2 bits (ALU op) and the other one with a length of 6 bits (Function) ALU control unit decides what the ALU control bus value should be according to the following table.

ALU op	Function	ALU control	ALU operation	Instruction
11	xxxxxx	01	XOR	XORi (XOR with immediate value)
00	xxxxxx	00	ADD	LW/SW (Load or Store word)
01	xxxxxx	10	SUB	BNE (Branch if not equal)
10	100000	00	ADD	ADD-R (Add register values)
10	100010	10	SUB	SUB-R (Sub register values)
10	101010	11	SLT	SLT-R (Set on less than)

In every other combination of bits which is not indicated in the table ALU operation bus is set to “00” value.

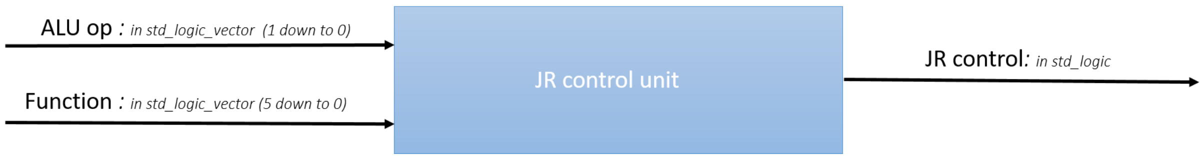
Timing diagram of ALU control unit



In the shown timing diagram we can see that to the corresponding combinations of “ALU op” bus and “Function” bus ALU control module has given the expected results as indicated in the table of above section.

JR (Jump Register) control unit

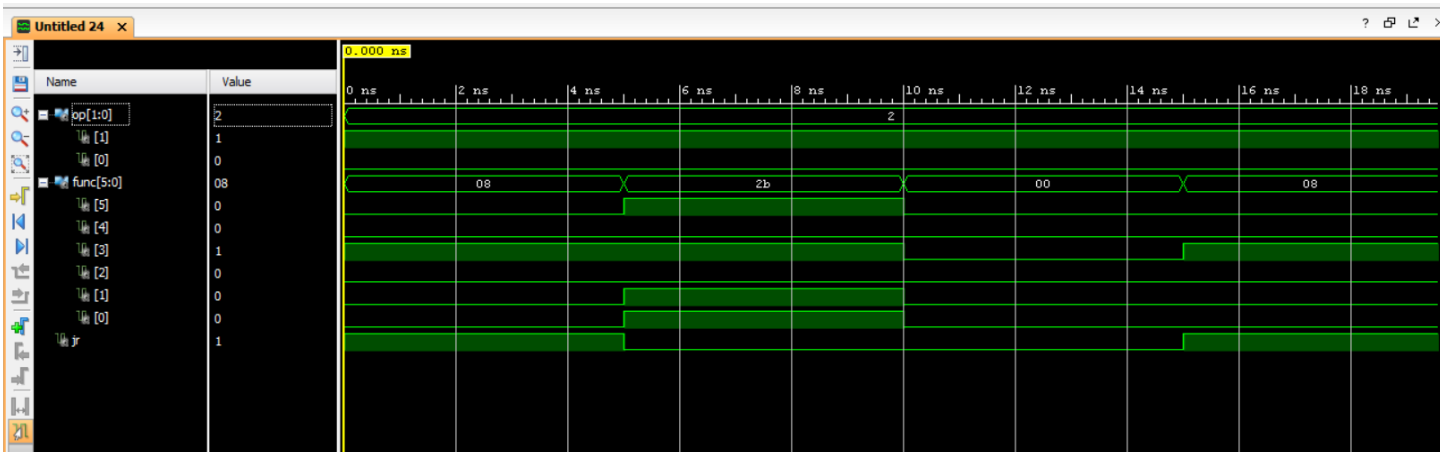
The Jr control unit is responsible for emitting a jump register signal which will be caused by a jump register instruction in the program. Following diagram shows an overview of the inputs and outputs of the JR control unit.


ALU op : in std_logic_vector (1 down to 0)
Function : in std_logic_vector (5 down to 0)
JR control unit
JR control: in std_logic

Same as in the ALU control unit, the JR control unit also receives decoded instruction data from “ALUop” bus and the “Function” bus. The combination bits of “ALUop” and “Function” buses that indicates a “JUMP $reg” instruction and triggers the jump signal is,

- ALUop = “10”
- Function = “001000”

Which is a combination which is not handled in the ALU control unit. In every other combination of ALUop and Function bus bits the jump register signal is ‘0’.

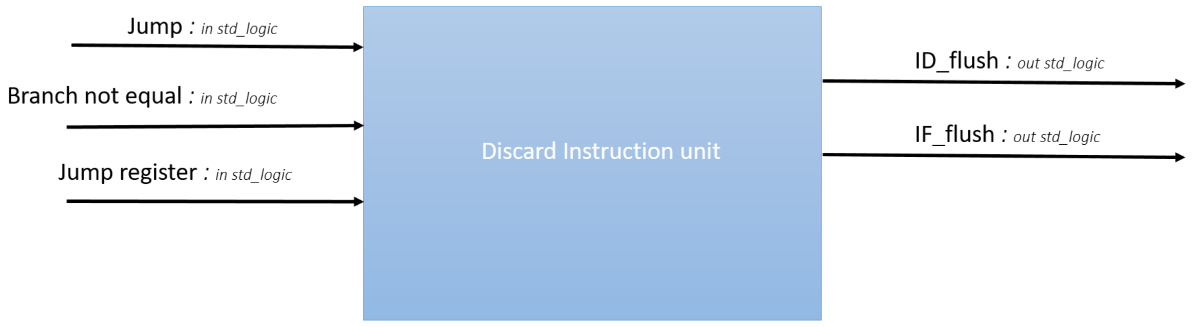


Timing diagram of JR control unit

It is shown in the above timing diagram that only in the combination of ALUop bus = “10” and Function bus = “001000” , the jump register signal is made high which is the expected behaviour of the JR control unit.

Discard Instruction Unit

This unit is responsible for handling control hazards that occur by “Jump”, “BNE” or “Jr” instructions. The Discard Instruction unit keeps track of “Jump” , “Branch if not equal” and “Jump register” signals and flush instructions in instruction fetch or instruction decode phase. Following diagram gives an overview of the inputs and outputs of the discard instruction unit.

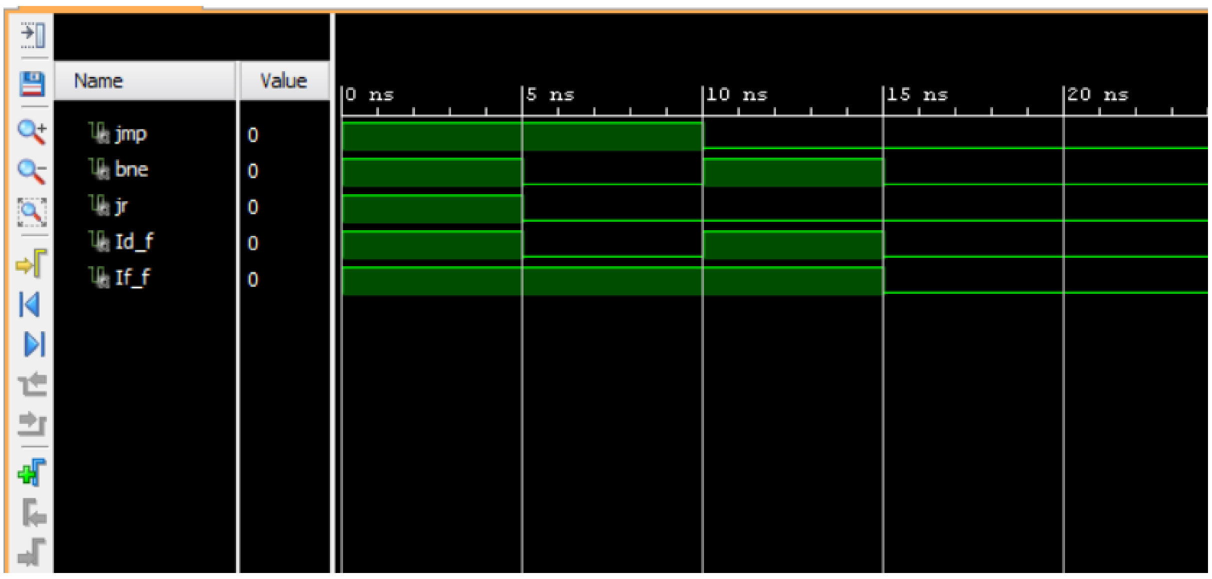


Logic that runs Inside the Instruction discard unit is as follows,

- $IF\ Flush\ =\ Jump\ or\ BNE\ or\ JR$
- $ID\ Flush\ =\ BNE\ or\ JR$

Instructions in the Instruction fetch stage are flushed whenever any of the three signals occur and Instructions in the instruction decode stage are flushed only when any of the “BNE” or “JR” signal is present.

Timing diagram for Instruction discard unit

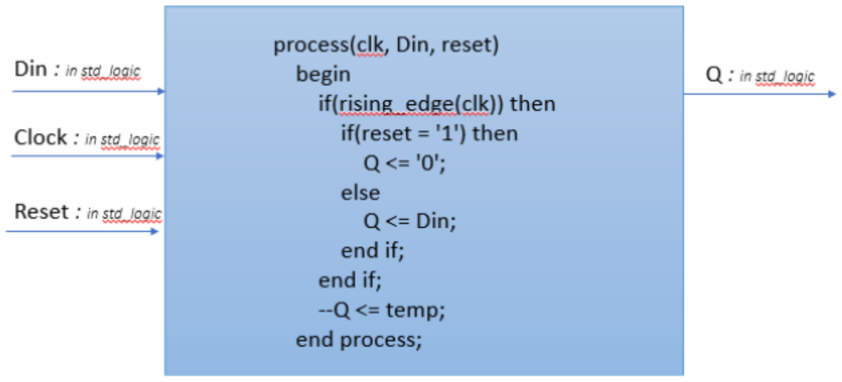


As shown in the timing diagram instructions unit follows the logic that is stated in the above section.

Register 32 bit

Flip flop

A D (or Delay) Flip Flop is a digital electronic circuit used to delay the change of state of its output signal (Q) until the next rising edge of a clock timing input signal occurs.



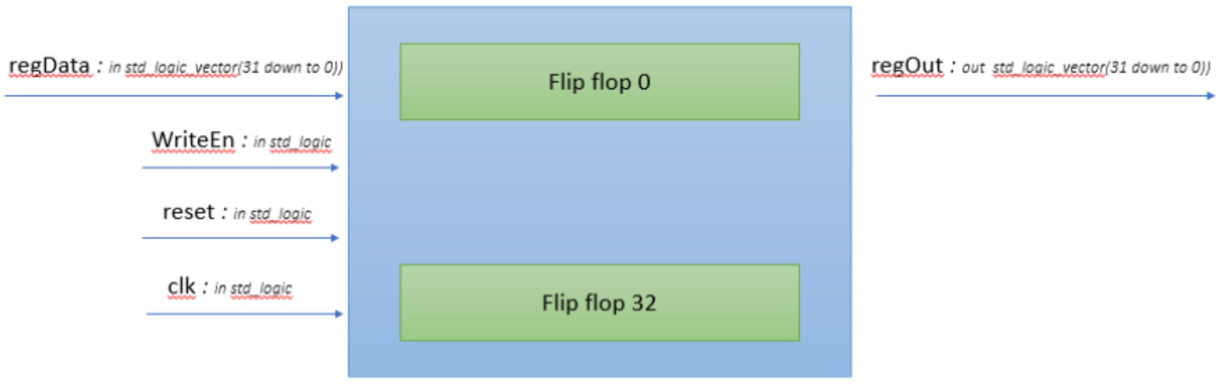
```
process(clk, Din, reset)
  begin
    if(rising_edge(clk)) then
      if(reset = '1') then
        Q <= '0';
      else
        Q <= Din;
      end if;
    end if;
    --Q <= temp;
  end process;
```

The truth table for the D Flip Flop

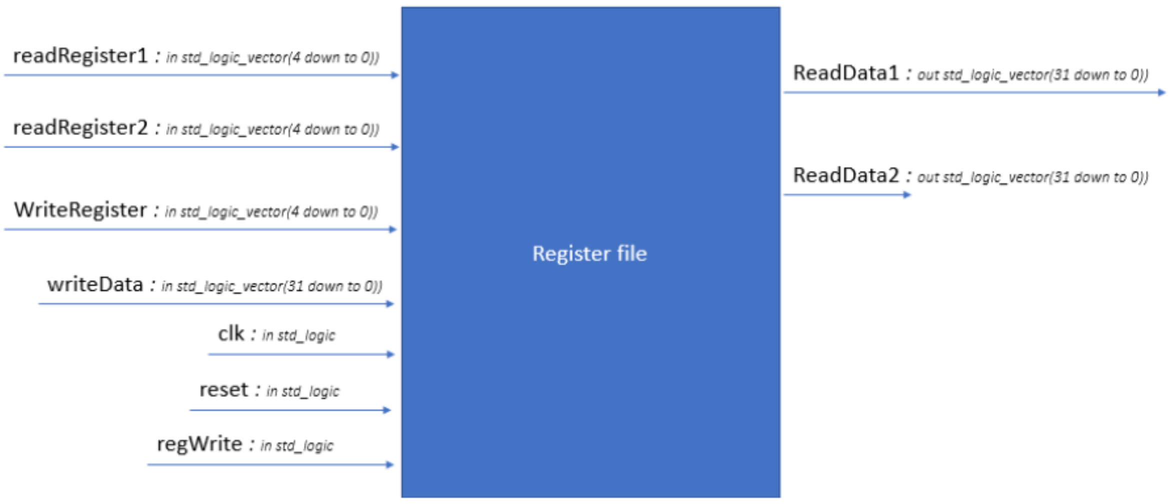
D	Clk	Q
0	Rising edge	0
1	Rising edge	1

When the clock input is set to 1, the "set" and "reset" inputs of the flip-flop are both set to 1. So it will not change the state and store the data present on its output before the clock transition occurs. In simple words, the output is "latched" at either 0 or 1.

Using flip flops we have created the register 32 bit module which is capable of storing a data word at a time


regData : in std_logic_vector(31 down to 0))
WriteEn : in std_logic
reset : in std_logic
clk : in std_logic
Flip flop 0
Flip flop 32
regOut : out std_logic_vector(31 down to 0))

Register File

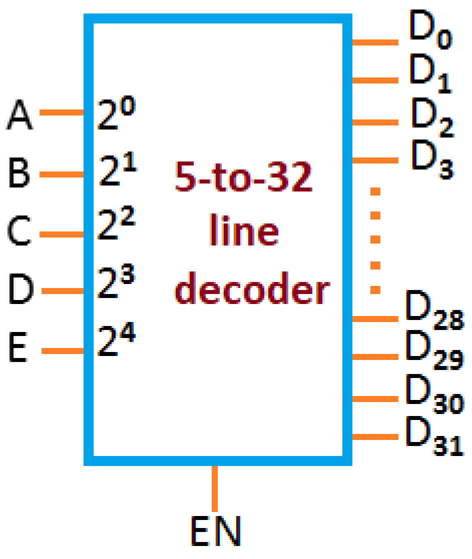

readRegister1 : in std_logic_vector(4 down to 0))
readRegister2 : in std_logic_vector(4 down to 0))
WriteRegister : in std_logic_vector(4 down to 0))
writeData : in std_logic_vector(31 down to 0))
clk : in std_logic
reset : in std_logic
regWrite : in std_logic
Register file
ReadData1 : out std_logic_vector(31 down to 0))
ReadData2 : out std_logic_vector(31 down to 0))

The register file keeps track of the registers in the MIPS processor. The register file is created using three different components.

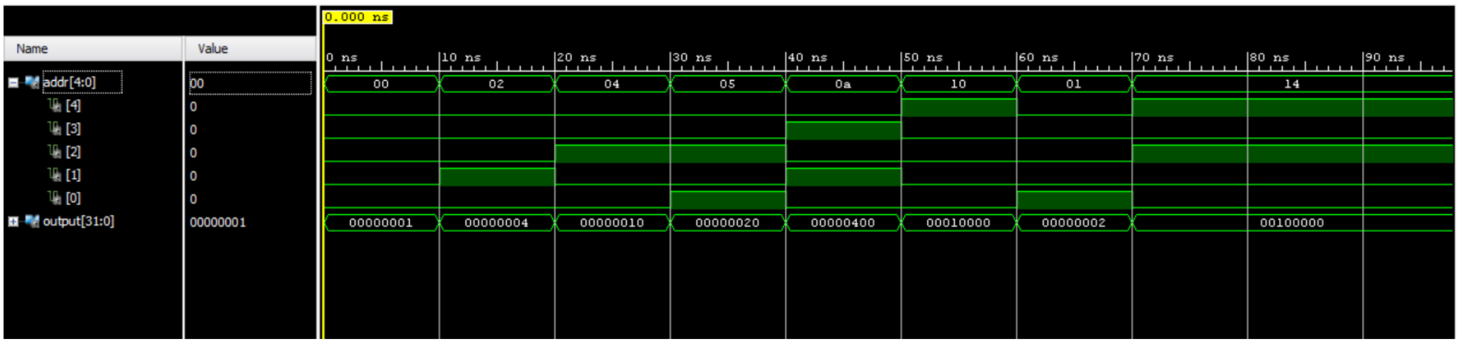
1. 5 to 32 Decoder
2. RAM array
3. 32x32 to 32 Multiplexer

5 to 32 Decoder

Decoder is a combinational circuit that has ‘n’ input lines and maximum of 2n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means the decoder detects a particular code. The outputs of the decoder are nothing but the minterms of ‘n’ input variables lines, when it is enabled.



Timing diagram of 5 to 32 decoder

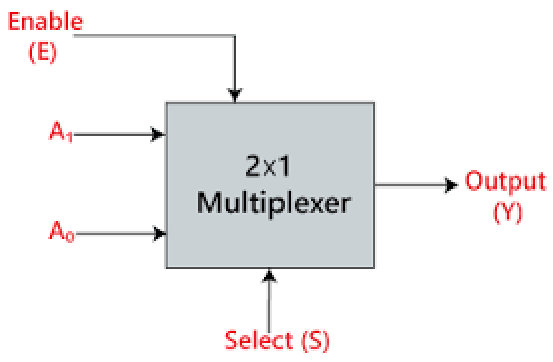


As shown in the above timing diagram the decoder has 5 input pins which are mapped to 32 bit output. For input 00000 the system outputs 0x00000001.

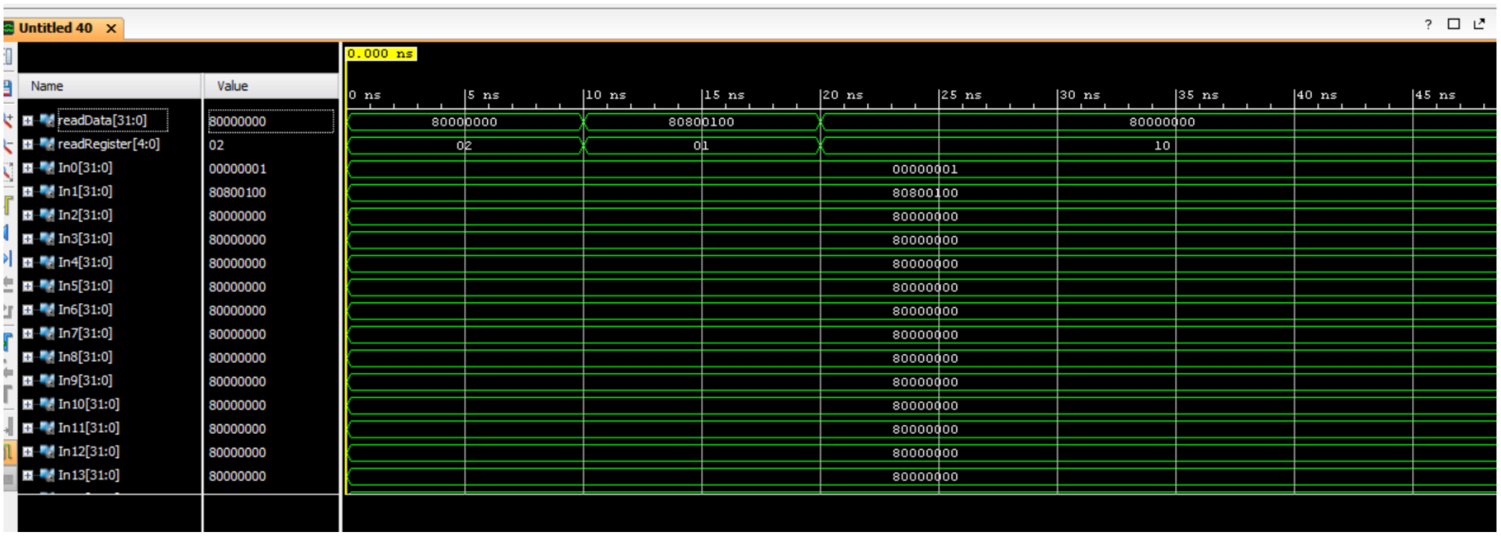
32x32 to 32 Multiplexer

Multiplexer is a combinational circuit that has a maximum of 2n data inputs, ‘n’ selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are ‘n’ selection lines, there will be 2n possible combinations of zeros and ones. So, each combination will select only one data input. In our project we have created a 32x32 to 32 multiplexer using 2 to 1 multiplexers.



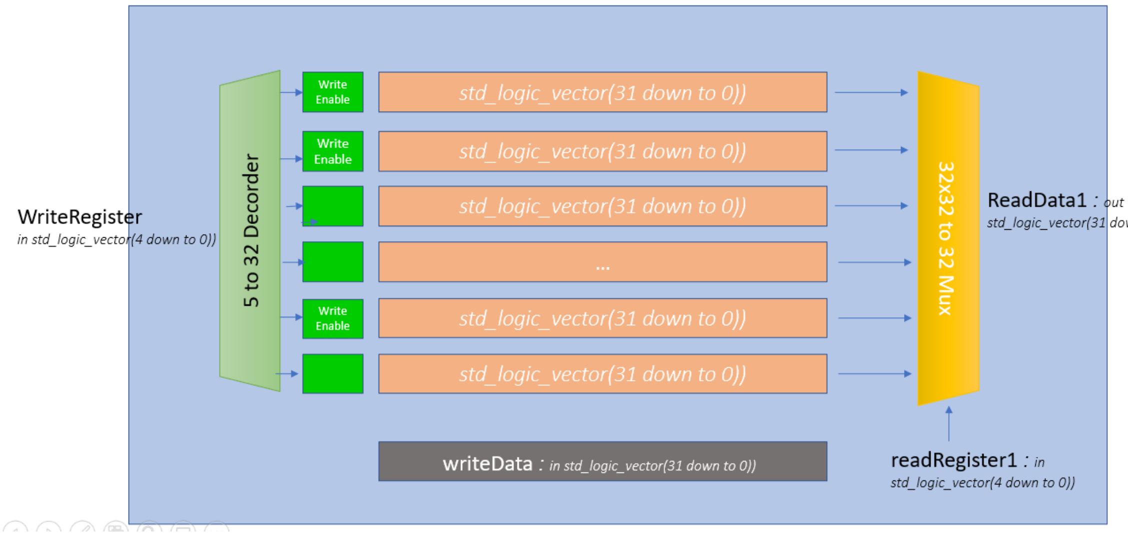
Timing diagram of 32x32 to 32 Multiplexer



```
Untitled 40
0.000 ns
Name                Value       0 ns   5 ns   10 ns   15 ns   20 ns   25 ns   30 ns   35 ns   40 ns   45 ns
readData[31:0]      80000000    80000000      80800100        80000000
readRegister[4:0]   02          02            01              10
In0[31:0]           00000001    00000001
In1[31:0]           80800100    80800100
In2[31:0]           80000000    80000000
In3[31:0]           80000000    80000000
In4[31:0]           80000000    80000000
In5[31:0]           80000000    80000000
In6[31:0]           80000000    80000000
In7[31:0]           80000000    80000000
In8[31:0]           80000000    80000000
In9[31:0]           80000000    80000000
In10[31:0]          80000000    80000000
In11[31:0]          80000000    80000000
In12[31:0]          80000000    80000000
In13[31:0]          80000000    80000000
```

The above timing diagram shows the input and output values of the 32x32 to 32 multiplexer.

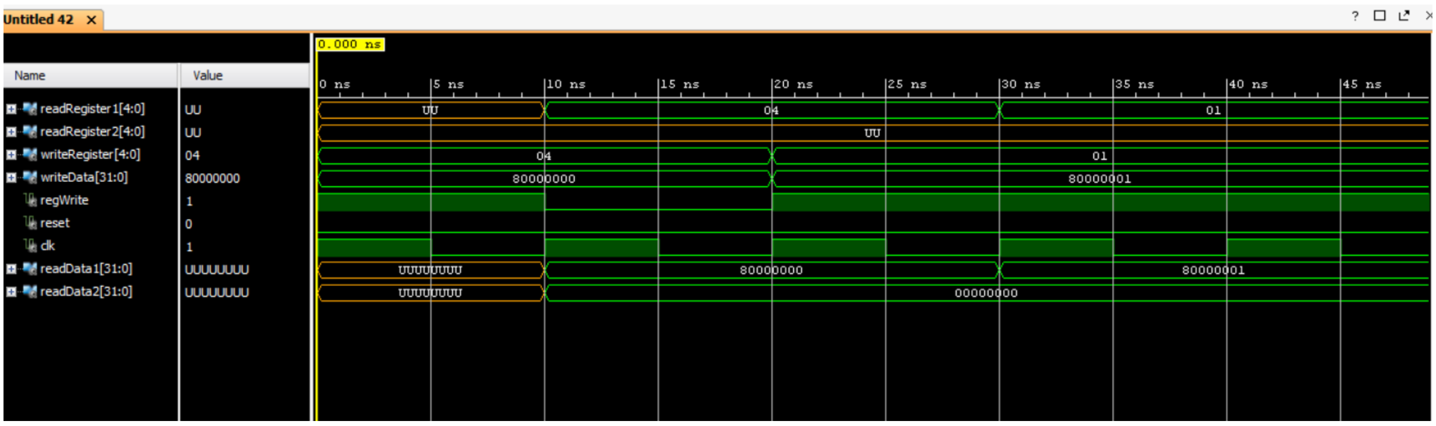
Using above mentioned modules we have created the register file for our project. The decoder is being used to select the write Element from the RAM array. The multiplexer is used to read the values of a vector given the address.


WriteRegister
in std_logic_vector(4 down to 0))
5 to 32 Decorder
Write Enable
Write Enable
Write Enable
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
...
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
writeData : in std_logic_vector(31 down to 0))
32x32 to 32 Mux
ReadData1 : out std_logic_vector(31 down to 0))
readRegister1 : in std_logic_vector(4 down to 0))

The above diagram shows the inside of the register file having 5 to 32 decoder. The RAM consists of 32, 32 bit std_logic_vectors. For writing a value we will input it through the writeData bus and make the writeEnable bit corresponding to the std_logic_vector high using the decoder. Then the value will be written to the respective registers.

For reading we have used the 32x32 to 32 multiplexer which can output a 32bit vector given a readRegister address.

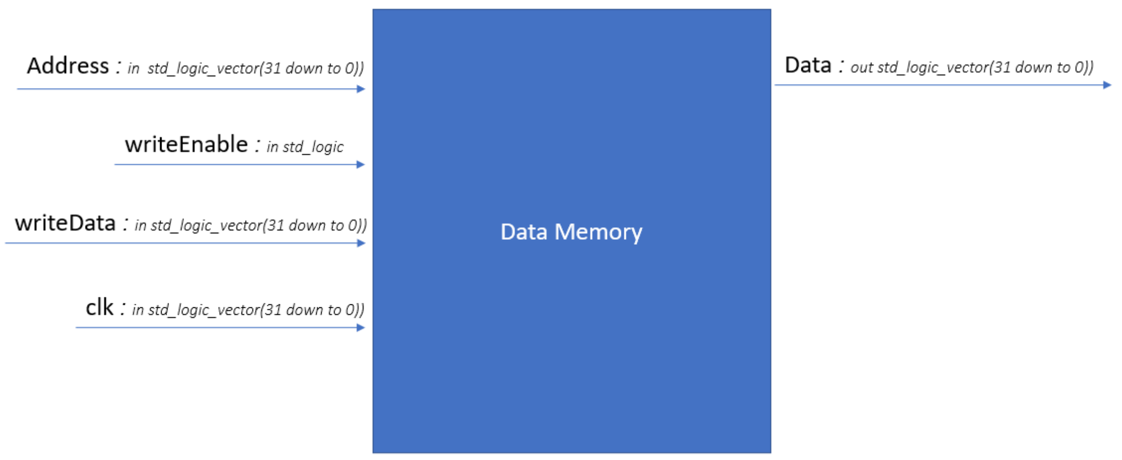
Timing diagram of register file



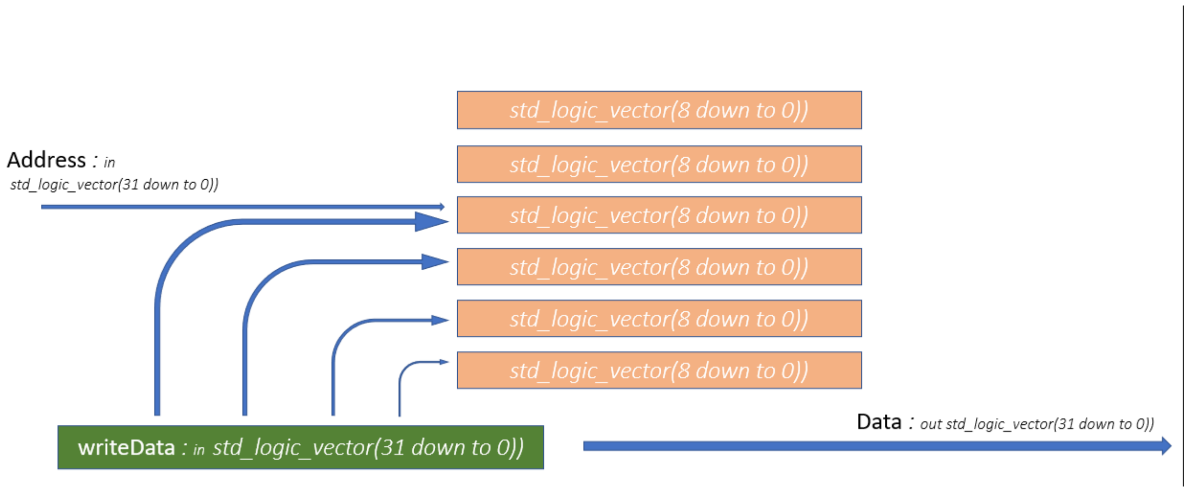
The above timing diagram shows the working of the register file. First we write the value “10000000000000000000000000000000” to the address “00100”. Then we will read the value which is the successful output in the 2nd clock cycle. Then we write an “10000000000000000000000000000001” again to “00001” address.

```
    writeRegister <= "00100";
    writeData     <= "10000000000000000000000000000000";
    regWrite      <= '1';
    reset         <= '0';
    wait for 10ns;

    readRegister1 <= "00100";
    regWrite      <= '0';
    wait for 10ns;


    writeRegister <= "00001";
    writeData     <= "10000000000000000000000000000001";
    regWrite      <= '1';
    wait for 10ns;


    readRegister1 <= "00001";
    wait for 10ns;
```

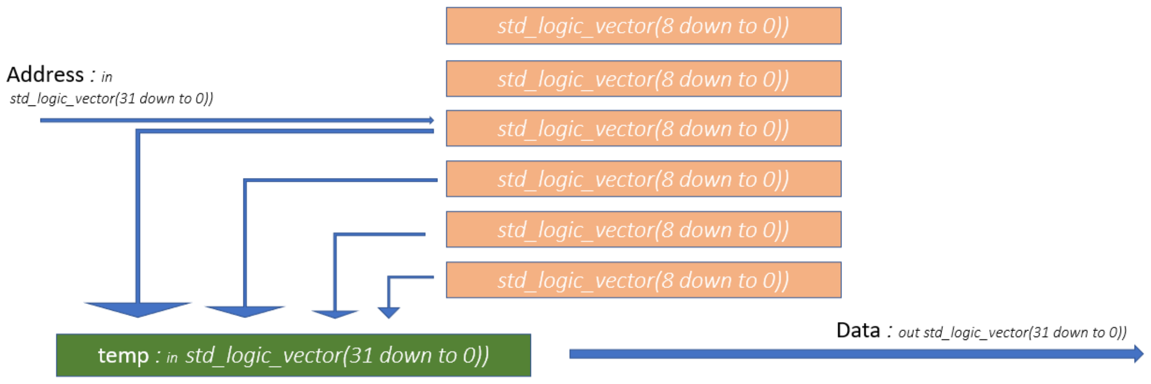
Data Memory


Address : in std_logic_vector(31 down to 0))
writeEnable : in std_logic
writeData : in std_logic_vector(31 down to 0))
clk : in std_logic_vector(31 down to 0))
Data Memory
Data : out std_logic_vector(31 down to 0))

The data memory module holds the data of the executing instructions inside the processor. The data memory module consists of an array(0 to 1023) of std_logic_vector(7 downto 0) array.

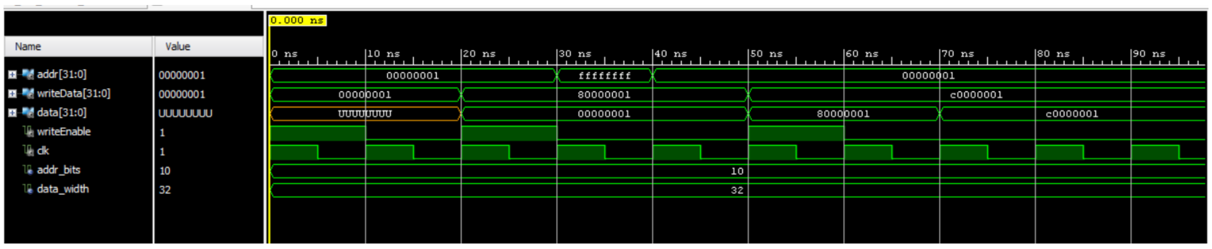

std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
Address : in std_logic_vector(31 down to 0))
writeData : in std_logic_vector(31 down to 0))
Data : out std_logic_vector(31 down to 0))

To use the constraint block ram in Basys three boards successfully we split a 32 bit single std_logic_vector data element to 4 different 8 bit elements. So when writeData and address is given we first split the data into 4 block and write them in the memory starting from the address mentioned in the Address std_logic_vector.


Address : in
std_logic_vector(31 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
std_logic_vector(8 down to 0))
temp : in std_logic_vector(31 down to 0))
Data : out std_logic_vector(31 down to 0))

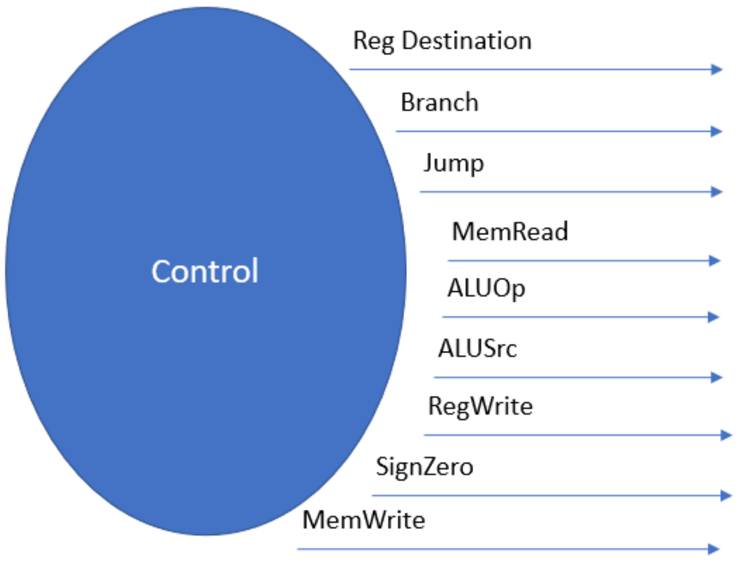
The above diagram shows the typical reading data. When the address is given the data memory will concatenate four near addresses and create the full data value. The concatenated values are then returned from the module.

Timing diagram of data memory



Control Unit

The control unit is responsible for the control of the processor. It issues respective flags to the modules based on the opcode.

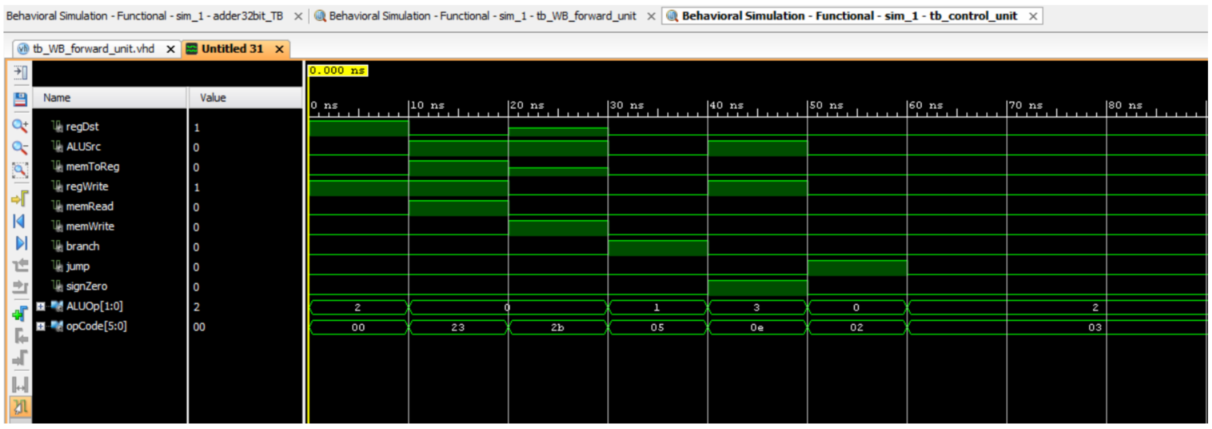


The control unit implemented in the project is capable of handling

1. R type operations
2. Load word operation - lw
3. Store word operation - sw
4. Branch if not equal - bne
5. XOR immediate - XORI
6. Jump operation - j

Output	R	lw	sw	bne	XORI	j	default
regDst	1	0	-	0	0	0	0
ALUSrc	0	1	1	0	1	0	0
memToReg	0	1	-	0	0	0	0
regWrite	1	1	0	0	1	0	0
memRead	0	1	0	0	0	0	0
memWrite	0	0	1	0	0	0	0
branch	0	0	0	1	0	0	0
jump	0	0	0	0	0	0	0
signZero	0	0	0	0	1	1	0
ALUOp	10	00	00	01	11	00	10

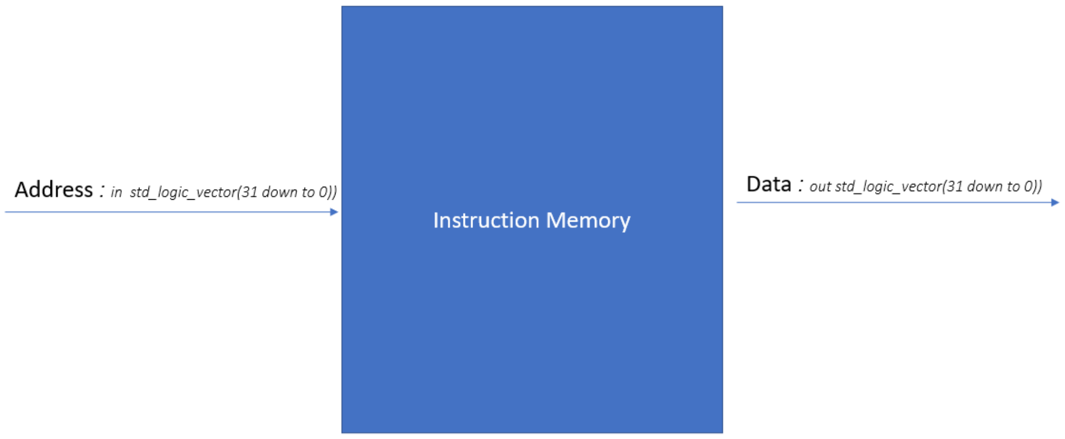
Timing diagram of control unit



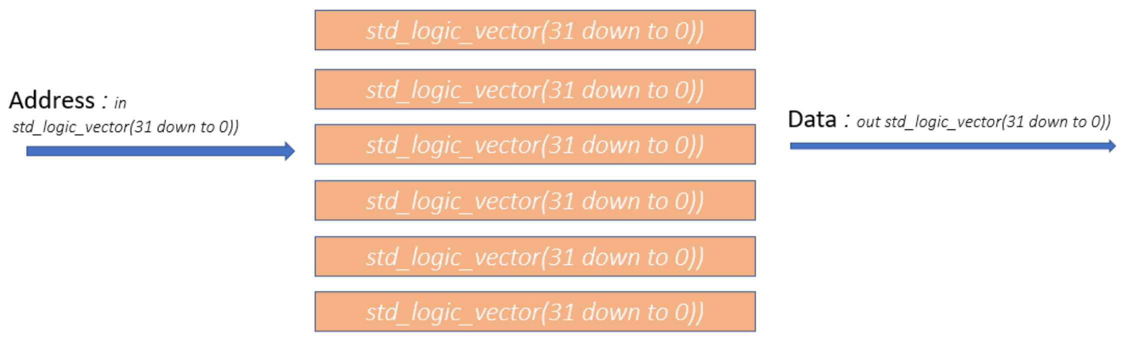
The above timing diagram shows the output for respective opcode values.

Instruction Memory

The instruction memory is a Read Only Memory(ROM) which stores the instructions to be executed.


Address : in std_logic_vector(31 down to 0))
Instruction Memory
Data : out std_logic_vector(31 down to 0))

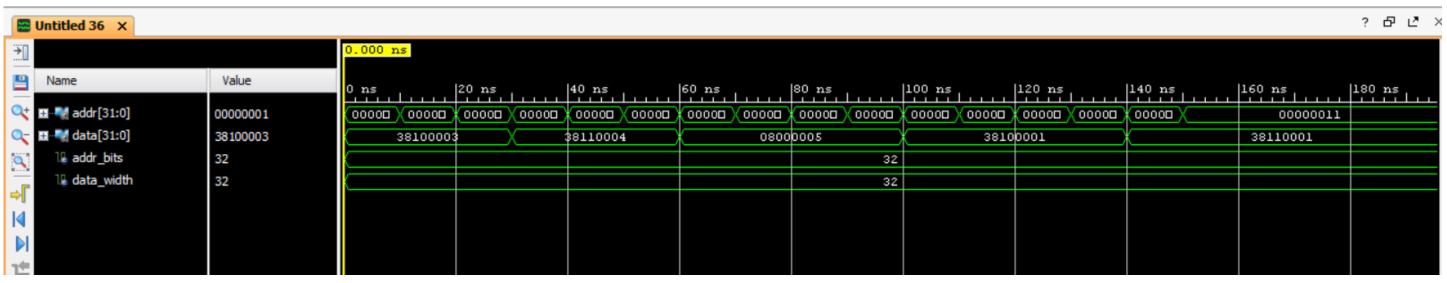
The instruction memory has been created using an array(0 to 14) of std_logic_vector(31 downto 0) array.


Address : in
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
std_logic_vector(31 down to 0))
Data : out std_logic_vector(31 down to 0))

```vhdl
signal instructionROM : rom_type := (
        "00111000000100000000000000000011", -- 940572675
        "00111000000100010000000000000100", -- 940638212
        "00001000000000000000000000000101", -- 134217733
        "00111000000100000000000000000001", -- 940572673
        "00111000000100010000000000000001", -- 940638209
        "00000010001100001001000000100010", -- 36737058
        "00010110000100011111111111111100",  -- 370278396
        "00000010000100011001100000100000", -- 34707488
        "10101110010100110000000000010000", -- 2924675088
        "10001110010101000000000000010000", --  2387869712
        "00000010000101001010100000101010", -- 34908202
        "10001110010100110000000000010000", -- 2387804176
        "00111010010100110000000000000001", -- 978518017
        "00111010101101010000000000000001", -- 984940545
        "00000010101000000000000000001000" -- 44040200
    );
```

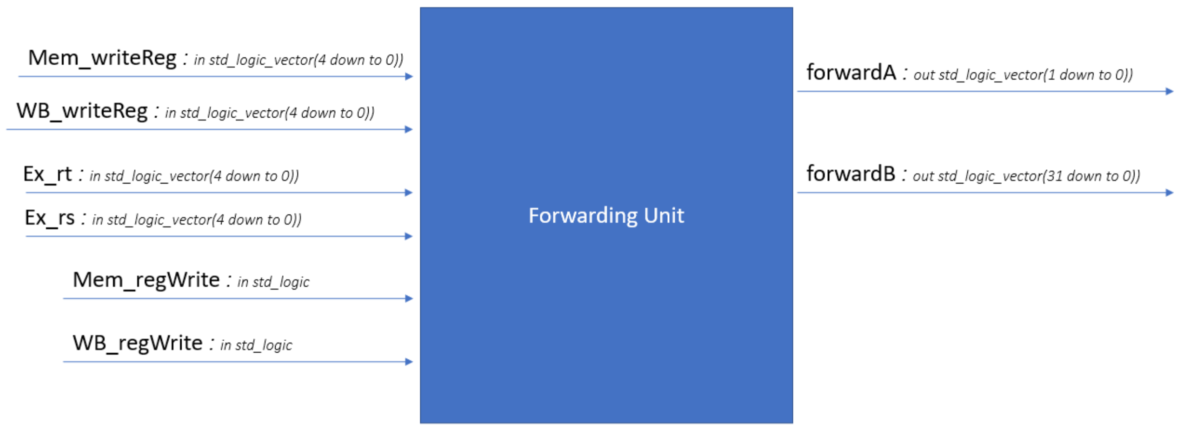
The timing diagram for the instruction memory is as follows. For each input address it return the corresponding instruction from the memory.

Timing diagram of Instruction memory



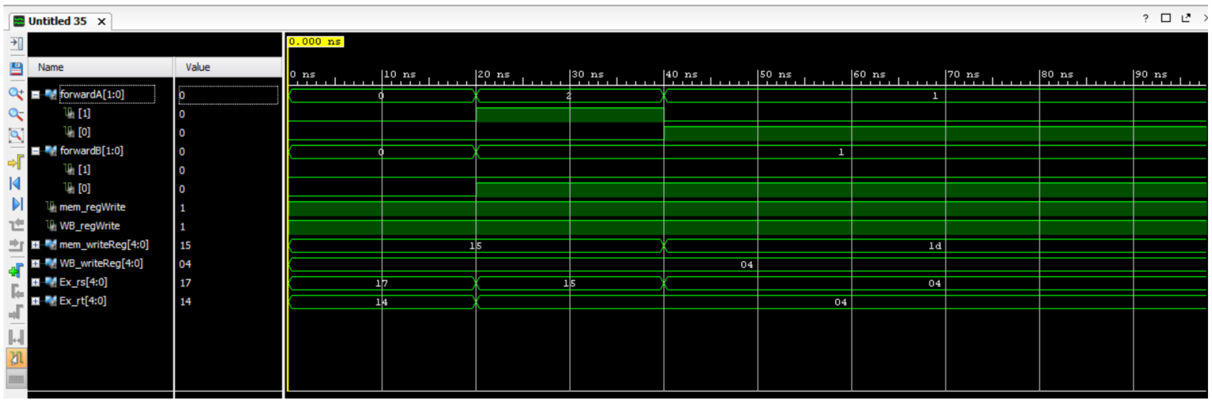
Forwarding Unit

The Forwarding Unit is designed to solve the data hazards in pipelined MIPS Processors. The correct data at the output of the ALU is forwarded to the input of the ALU when data hazards are detected. Data hazards are detected when the source register (EX_rs or EX_rt) of the current instruction is the same as the destination register (MEM_WriteRegister or EX_WriteRegister) of the previous instruction.



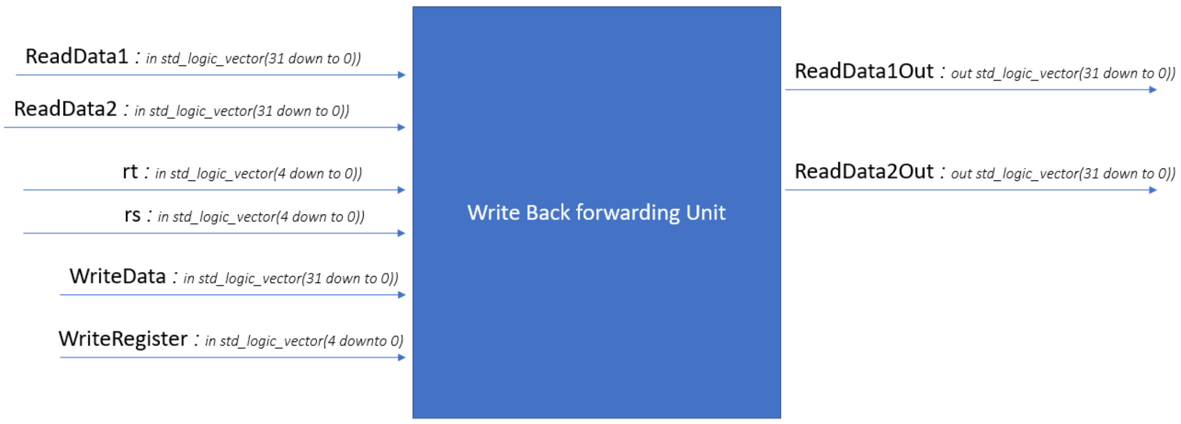
Mem_writeReg : in std_logic_vector(4 down to 0))
WB_writeReg : in std_logic_vector(4 down to 0))
Ex_rt : in std_logic_vector(4 down to 0))
Ex_rs : in std_logic_vector(4 down to 0))
Mem_regWrite : in std_logic
WB_regWrite : in std_logic
Forwarding Unit
forwardA : out std_logic_vector(1 down to 0))
forwardB : out std_logic_vector(31 down to 0))

Timing diagram of Forwarding unit

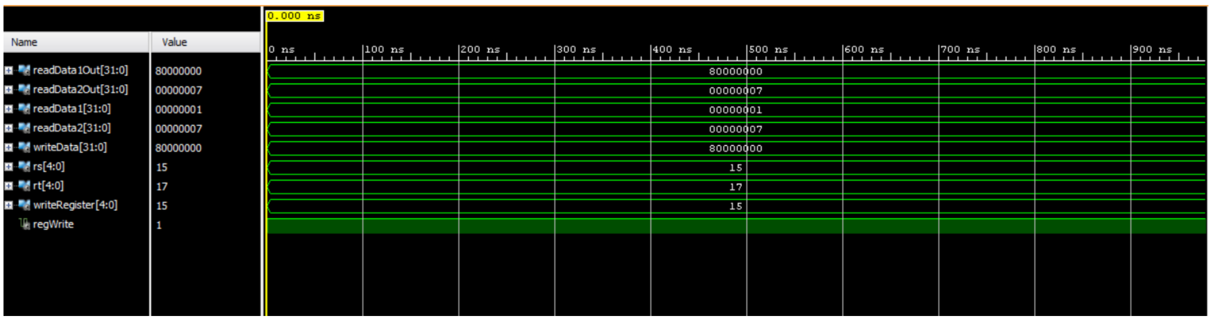


Write Back Forwarding Unit

Another hazard could happen at the Write Back Stage when writing and reading at the same address. The readout data may not be the correct writing data. To resolve this problem, a WB_Forward unit is designed to forward directly the correct writing data to the output data.


ReadData1 : in std_logic_vector(31 down to 0))
ReadData2 : in std_logic_vector(31 down to 0))
rt : in std_logic_vector(4 down to 0))
rs : in std_logic_vector(4 down to 0))
WriteData : in std_logic_vector(31 down to 0))
WriteRegister : in std_logic_vector(4 downto 0)
Write Back forwarding Unit
ReadData1Out : out std_logic_vector(31 down to 0))
ReadData2Out : out std_logic_vector(31 down to 0))

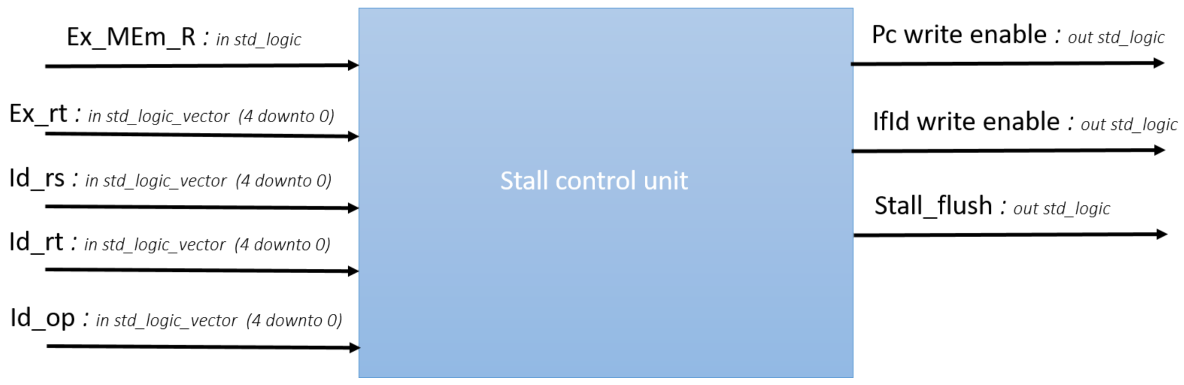
Timing diagram of write back forwarding unit



```
0.000 ns

Name                  Value       0 ns   100 ns   200 ns   300 ns   400 ns   500 ns   600 ns   700 ns   800 ns   900 ns
readData1Out[31:0]    80000000                                               80000000
readData2Out[31:0]    00000007                                               00000007
readData1[31:0]       00000001                                               00000001
readData2[31:0]       00000007                                               00000007
writeData[31:0]       80000000                                               80000000
rs[4:0]               15                                                     15
rt[4:0]               17                                                     17
writeRegister[4:0]    15                                                     15
regWrite              1
```

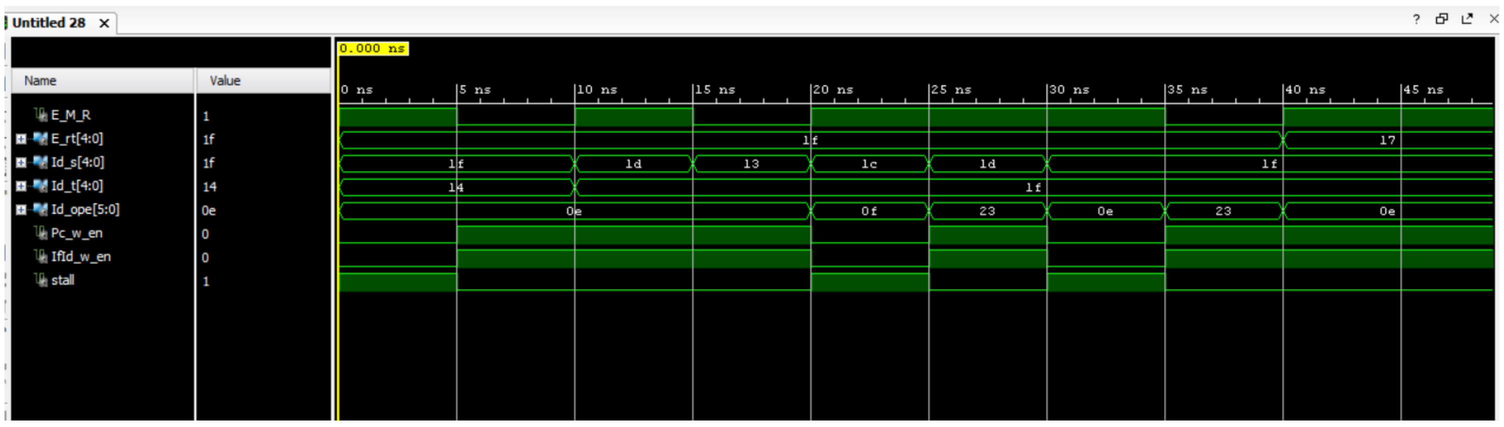
Stall control unit

Stall Control Unit is important when there are data hazards in the pipeline, in such cases the pipeline should be delayed by one clock cycle. Data hazards occur when the destination register of the current reading memory instruction is the same as the source register of the next instruction in the ID stage except for ID_rt of XORI and LW instructions (where Rt is the destination register not source register with XORI and LW). In the stall control unit two addresses are compared using “XOR” gates (if two addresses are the same results in ‘0’) and the output was set accordingly. Following image shows an overview of inputs and outputs of the component.


Ex_MEm_R : in std_logic
Ex_rt : in std_logic_vector (4 downto 0)
Id_rs : in std_logic_vector (4 downto 0)
Id_rt : in std_logic_vector (4 downto 0)
Id_op : in std_logic_vector (4 downto 0)
Stall control unit
Pc write enable : out std_logic
IfId write enable : out std_logic
Stall_flush : out std_logic

Here Ex_rt is the destination address of the current instruction which is in the execution stage and Id_rs and Id_rt are the source and destination register address in the decode stage.

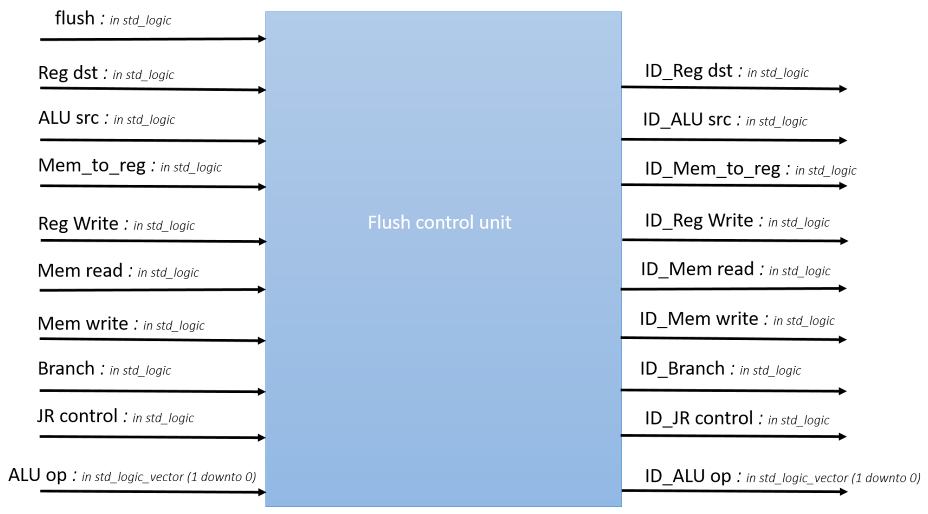
Timing Diagram of Stall control unit



As shown in the timing diagram whenever the destination register of the current instruction is equal to source or destination of the instructions in the decode stage, Flush bit is set to ‘1’.

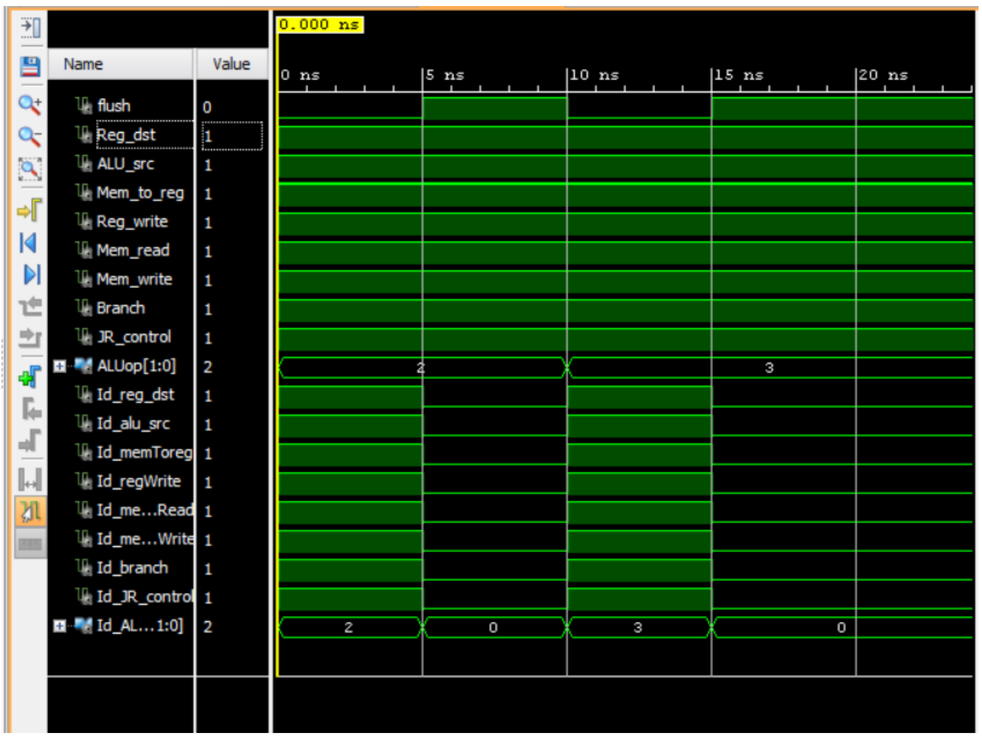
Flush control unit and Discard Instruction unit

In pipelined instruction execution, when a branch , jump sort of instructions are executed according to the result of the instruction some instructions that are already in the pipeline might have to be flush out of the pipeline, these are known as control hazards of a pipelined architecture. A Flush control unit and Discard instruction unit is there to handle these control hazards in the pipeline. Following diagram shows an overview of the inputs and outputs of the component.



flush : in std_logic
Reg dst : in std_logic
ALU src : in std_logic
Mem_to_reg : in std_logic
Reg Write : in std_logic
Mem read : in std_logic
Mem write : in std_logic
Branch : in std_logic
JR control : in std_logic
ALU op : in std_logic_vector (1 downto 0)
Flush control unit
ID_Reg dst : in std_logic
ID_ALU src : in std_logic
ID_Mem_to_reg : in std_logic
ID_Reg Write : in std_logic
ID_Mem read : in std_logic
ID_Mem write : in std_logic
ID_Branch : in std_logic
ID_JR control : in std_logic
ID_ALU op : in std_logic_vector (1 downto 0)

Timing diagram of Flush control unit



As shown in the timing diagram whenever the flush bit is high all the data that was available in the input buses is cleared.

Pipelined MIPS processor

The MIPS(Microprocessor without Interlocked Pipeline Stages) processor is a RISC (Reduced Instruction Set Computer) processor. One significance of the RISC processors is that as the name suggests it contains much more simple instructions when compared with their CISC (Complex Instruction Set Computer).

Pipeline processing refers to overlapping operations by moving data or instructions into a conceptual pipe with all stages of the pipe performing simultaneously. Pipeline is mainly used to improve throughput of the processor rather than the latency. Therefore pipelines do not make a certain operation faster instead it allows to perform multiple tasks which are having less dependency between them simultaneously. For example, while one instruction is being executed, the computer is decoding the next and while decoding other instructions can write to the registers.

But in pipelined processors due to small dependencies that exist between them pipeline hazards can occur. Namely there are three main types of hazards as,

- Data hazards
- Structural hazards
- Control hazards

Therefore handling these hazards in a pipelined processor is important to get efficient and correct results. In this project we are implementing a pipelined MIPS with pipelined architecture.

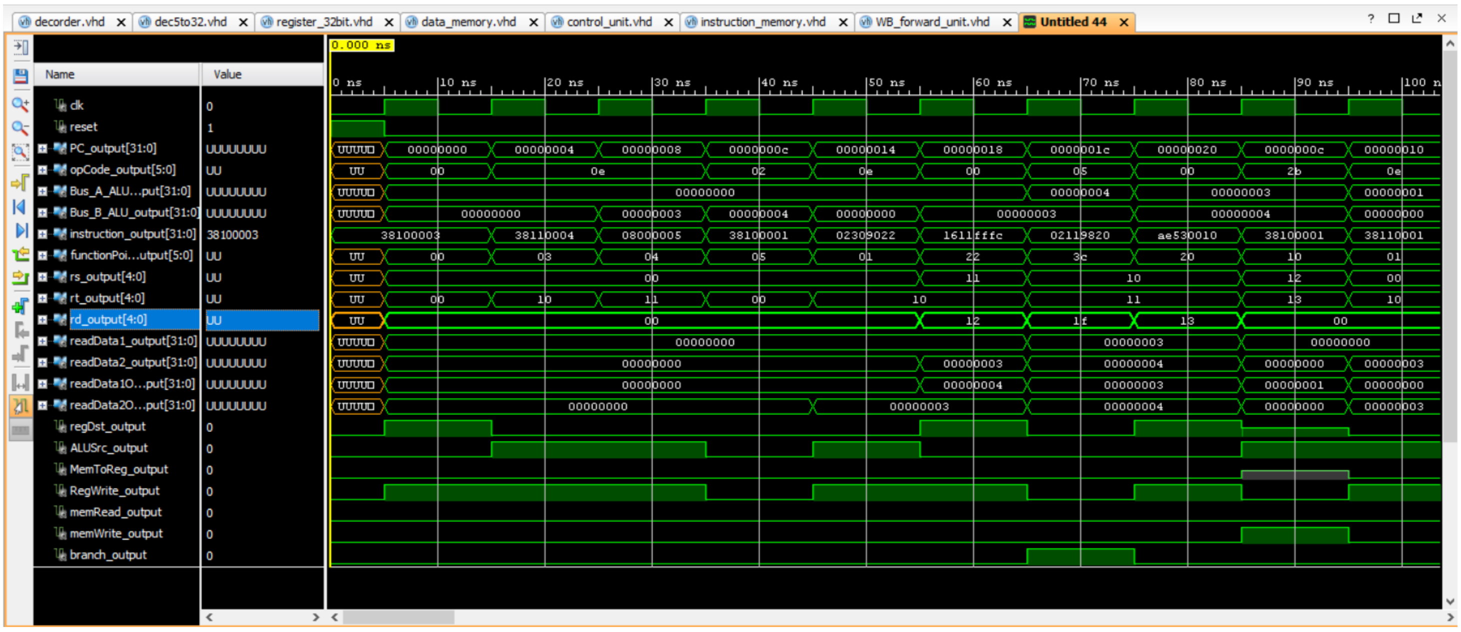
The instruction set that we are going to perform in the processor is as follows. It has chosen to cover all the XORI, SW, LW, SLT, J, JR, SUB, ADD operations implemented in the processor.

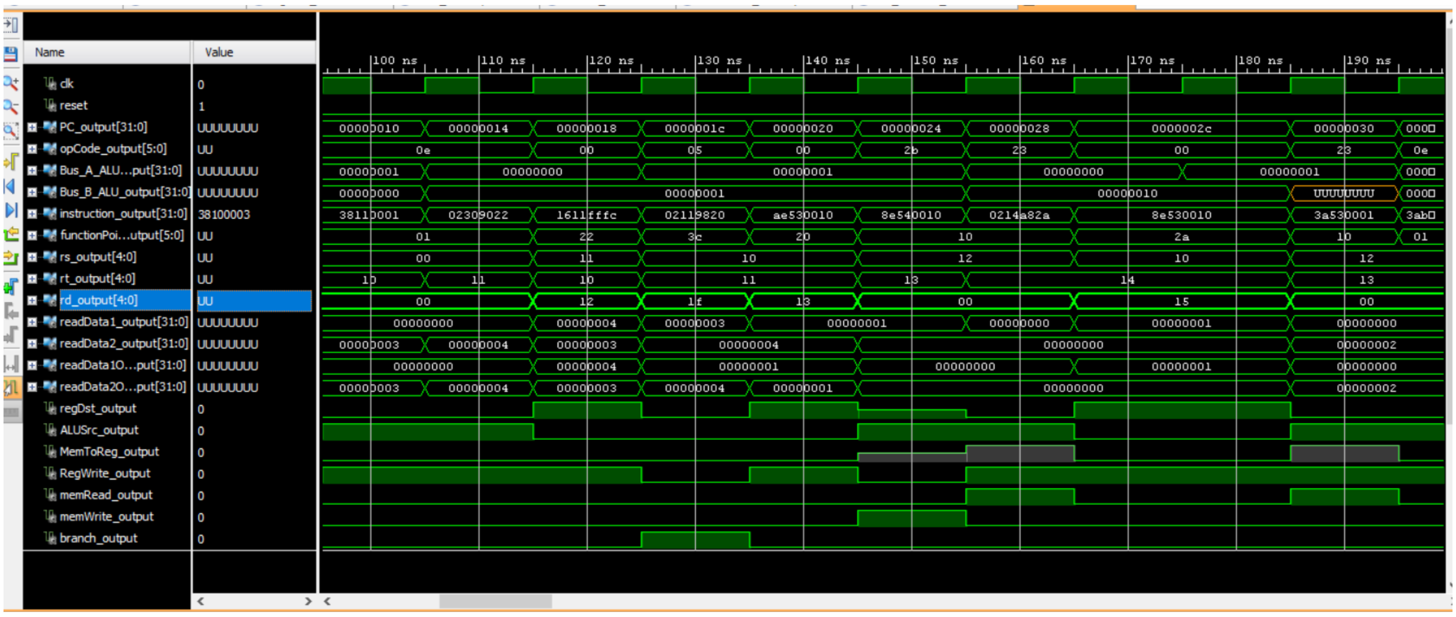
```
Start :     xori $s0, $zero, 0x0003
            xori $s1, $zero, 0x0004
             j next1
next 2:     xori $s0, $zero, 0x0001
            xori $s0, $zero, 0x0003
next 3:     sub $s2, $s1, $s0
            bne $s0, $s1, next2
            add $s3, $s0, $s1
            sw  $s3, 16($s2)
            lw   $s3, 16($s2)
            slt   $s5, $s0 , $s4
            lw   $s3, 16($s2)
            xori $s3, $s2, 0x0001
            xori $s5, $s5, 0x0001
            jr $s5
```

So let’s first hand solve the above problem.

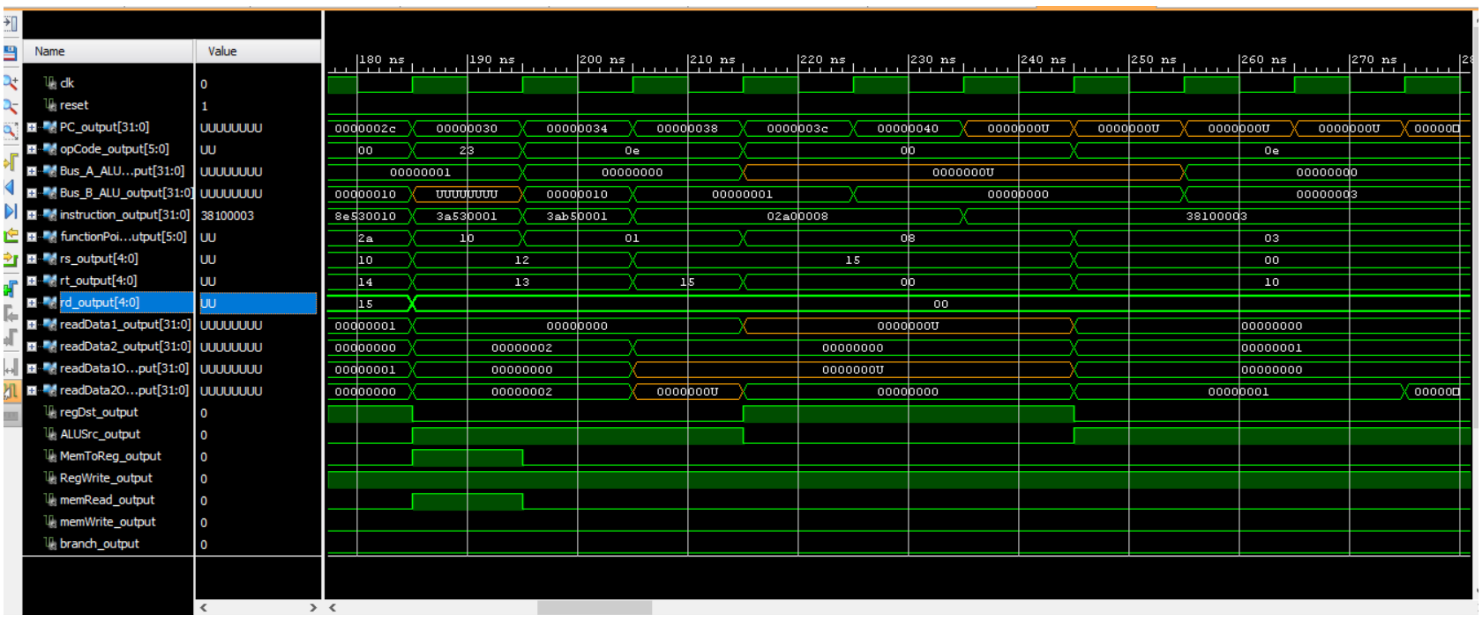
xori $s0, $zero, 0x0003
$0 = 3
xori $s1, $zero, 0x0004
$0 = 3 and $1 = 4
j next1
Jump to next1
sub $s2, $s1, $s0
$0 = 3 and $1 = 4 $2 = 1
bne $s0, $s1, next2
success go to next2
xori $s0, $zero, 0x0001
$0 = 1 and $1 = 4 $2 = 1
xori $s1, $zero, 0x0001
$0 = 1 and $1 = 1 $2 = 1
sub $s2, $s1, $s0
$0 = 1 and $1 = 1 $2 = 0
bne $s0, $s1, next2
branch not taken
add $s3, $s0, $s1
$0 = 1 and $1 = 1 $2 = 0 $3 = 2
sw $s3, 16($s2)
16 <- 2
lw $s3, 16($s2)
$0 = 1 and $1 = 1 $2 = 0 $3 = 2 $4 = 2
slt $s5, $s0 , $s4
$0 = 1 and $1 = 1 $2 = 0 $3 = 2 $4 = 2 $5 = 1
lw $s3, 16($s2)

\$0 = 1 and \$1 = 1 \$2 = 0 \$3 = 2   \$4 = 2 \$5 = 1
xori \$s3, \$s2, 0x0001
\$0 = 1 and \$1 = 1 \$2 = 0 \$3 = 1   \$4 = 2 \$5 = 1
xori \$s5, \$s5, 0x0001
\$0 = 1 and \$1 = 1 \$2 = 0 \$3 = 1   \$4 = 2 \$5 = 0
jr \$s5
jump to 0





```
Name                      Value
clk                       0
reset                     1
PC_output[31:0]           UUUUUUUU
opCode_output[5:0]        UU
Bus_A_ALU...put[31:0]     UUUUUUUU
Bus_B_ALU_output[31:0]    UUUUUUUU
instruction_output[31:0]  38100003
functionPoi...utput[5:0]  UU
rs_output[4:0]            UU
rt_output[4:0]            UU
rd_output[4:0]            UU
readData1_output[31:0]    UUUUUUUU
readData2_output[31:0]    UUUUUUUU
readData1O...put[31:0]    UUUUUUUU
readData2O...put[31:0]    UUUUUUUU
regDst_output             0
ALUSrc_output             0
MemToReg_output           0
RegWrite_output           0
memRead_output            0
memWrite_output           0
branch_output             0
```



```
Name                      Value
clk                       0
reset                     1
PC_output[31:0]           UUUUUUUU
opCode_output[5:0]        UU
Bus_A_ALU...put[31:0]     UUUUUUUU
Bus_B_ALU_output[31:0]    UUUUUUUU
instruction_output[31:0]  38100003
functionPoi...utput[5:0]  UU
rs_output[4:0]            UU
rt_output[4:0]            UU
rd_output[4:0]            UU
readData1_output[31:0]    UUUUUUUU
readData2_output[31:0]    UUUUUUUU
readData1O...put[31:0]    UUUUUUUU
readData2O...put[31:0]    UUUUUUUU
regDst_output             0
ALUSrc_output             0
MemToReg_output           0
RegWrite_output           0
memRead_output            0
memWrite_output           0
branch_output             0
```