Faculty Of Engineering

The Hashemite University, Zarqa - Jordan April

2024

Computer Architecture

Bus Based Architecture

Student Name	*ID#*
Issa Qandah	*2036177*
Hassan TaqiEddin	*2036057*

Table of content :

1. Introduction

In this phase we made the data path including all previous modules (RegisterFile,ALU) and made new modules (DataMemory,ImmGen,other registers, and muxes).
In bus-based architecture based on RISC-V ISA, which is an open-source, streamlined design for computer processors, emphasizing simplicity and adaptability. It uses a reduced instruction set computing (RISC) approach with a small set of basic instructions, making it efficient and flexible for a wide range of applications.
The bus is shared between all components, and it is defined as wire (with multiplexers to determine which read and which write) and only one of the shared components can write per cycle.
Microprogramming is applied here, which is a technique used in computer architecture where complex instructions of a CPU are broken down into simpler microinstructions stored in a control memory.
These microinstructions control the operations of the CPU's control unit, allowing for efficient execution of instructions.
By using microprogramming, CPUs can be designed more flexibly, enabling easier modification and optimization of instruction sets without altering the hardware design.

here which has these characteristics:

- *High CPI relatively: as it involves additional overhead in fetching and executing microinstructions.*
- *Cycle duration becomes small (Higher frequency): allowing for more efficient instruction execution through finer control over hardware operations.*
- *Easy to apply more instructions: offering flexibility in expanding the range of supported operations without necessitating significant hardware modifications.*

Until now there is no control unit implementation, so there is need to manually write the input signal to the datapath to make sure that the design is working properly, however control unit is going to be implemented at the next phase.

One modification we made on the reference design is removing the (ALU Control) module replacing it with 4-bits ALUOp signal coming directly from the control unit.

The design has 32-general purpose registers and another 4-special purpose registers (A,B,PC,IR), however PC register is placed inside the register file.

There is only one memory, that contains the instruction segment and the data segment and it has Busy signal which will be discussed in its section.

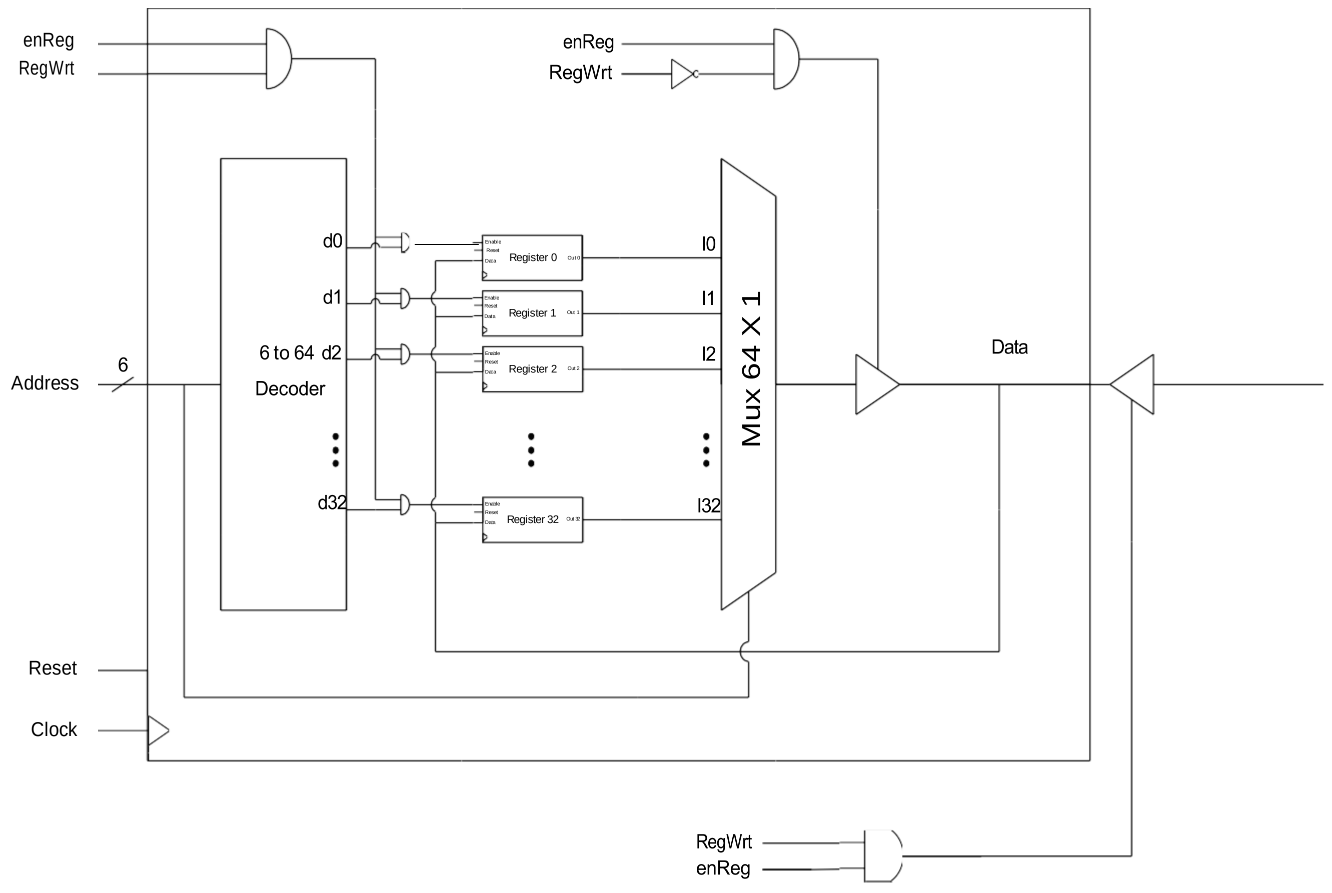
Immediate generation module has four types of immediate extension (I,S,B,J) each one has its use in term of type of extension (Zero or signed) and how many bits is extended.

2. Compunents

2.1 Register File:

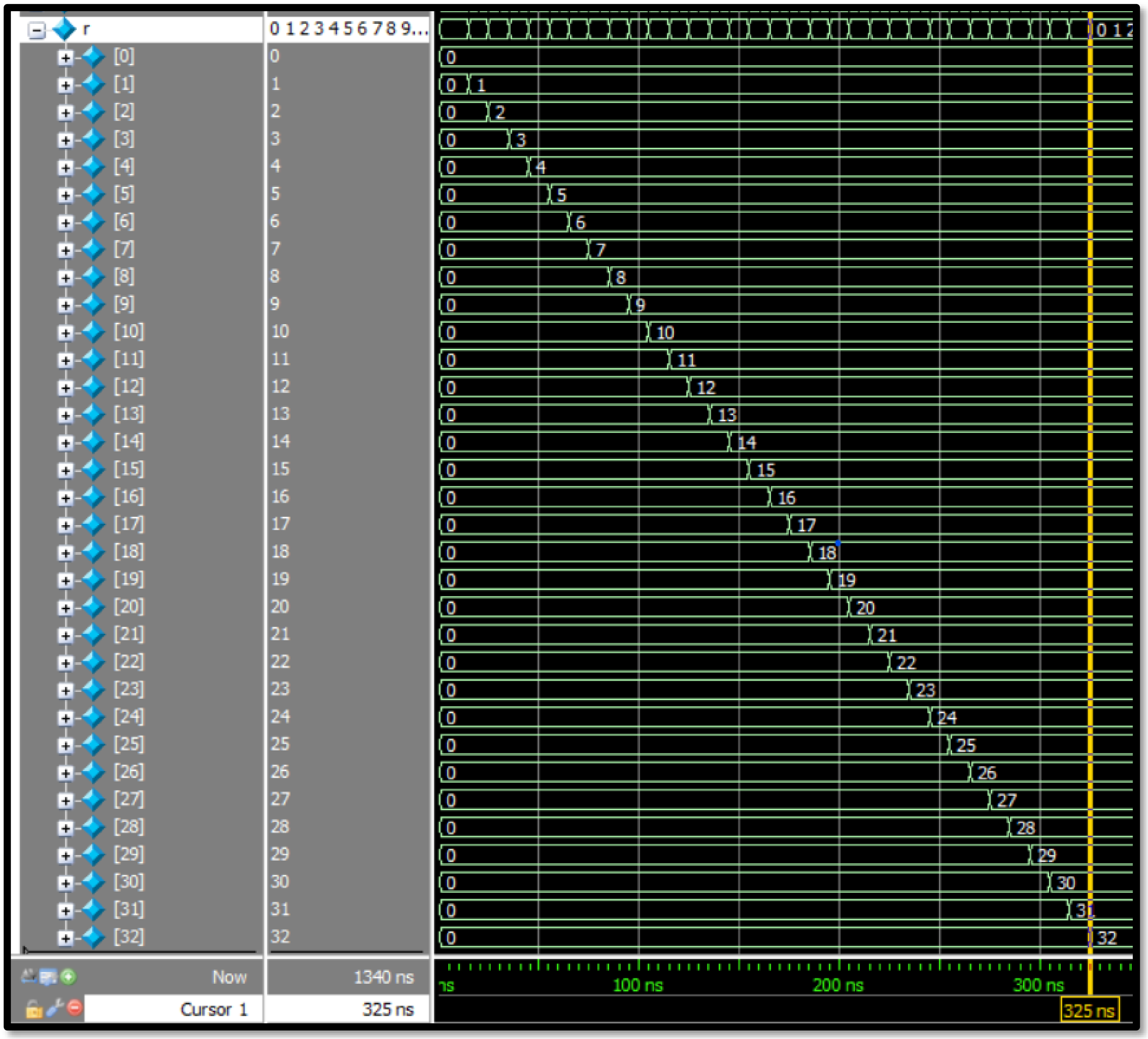
Component	***Description***
Registers	*32 General Purpose Registers (GPRs) + PC register*
Address Input	*6-bit address (addr), determines which register is to be read or written to*
Data Port	*Bidirectional, 32-bit*
Control Signals	*RegWr: Determines read/write operation (1 for write, 0 for read)* *enReg: General enable control, If enReg is 1, then the register reads or writes depending on RegWr. If enReg is 0, then nothing is done*
Read Operation	*Asynchronous*
Write Operation	*On positive edge of clock*
64x1 Decoder	*Decodes the 6-bit address input to select a specific register to write on*
64 by 6 Mux	*Selects register output data to read from based on address input*
Tri-state Buffer	*2 Tri-state Buffers, in case of reading from register file the write buffer puts the port on z state to allow the reading buffer putting data on the port,* *in case of writing on register file, the reading buffer puts the port on z state allowing* *the writing buffer to put data on the port*

2.1.1 Schematic design:



2.1.2 Register File Testing

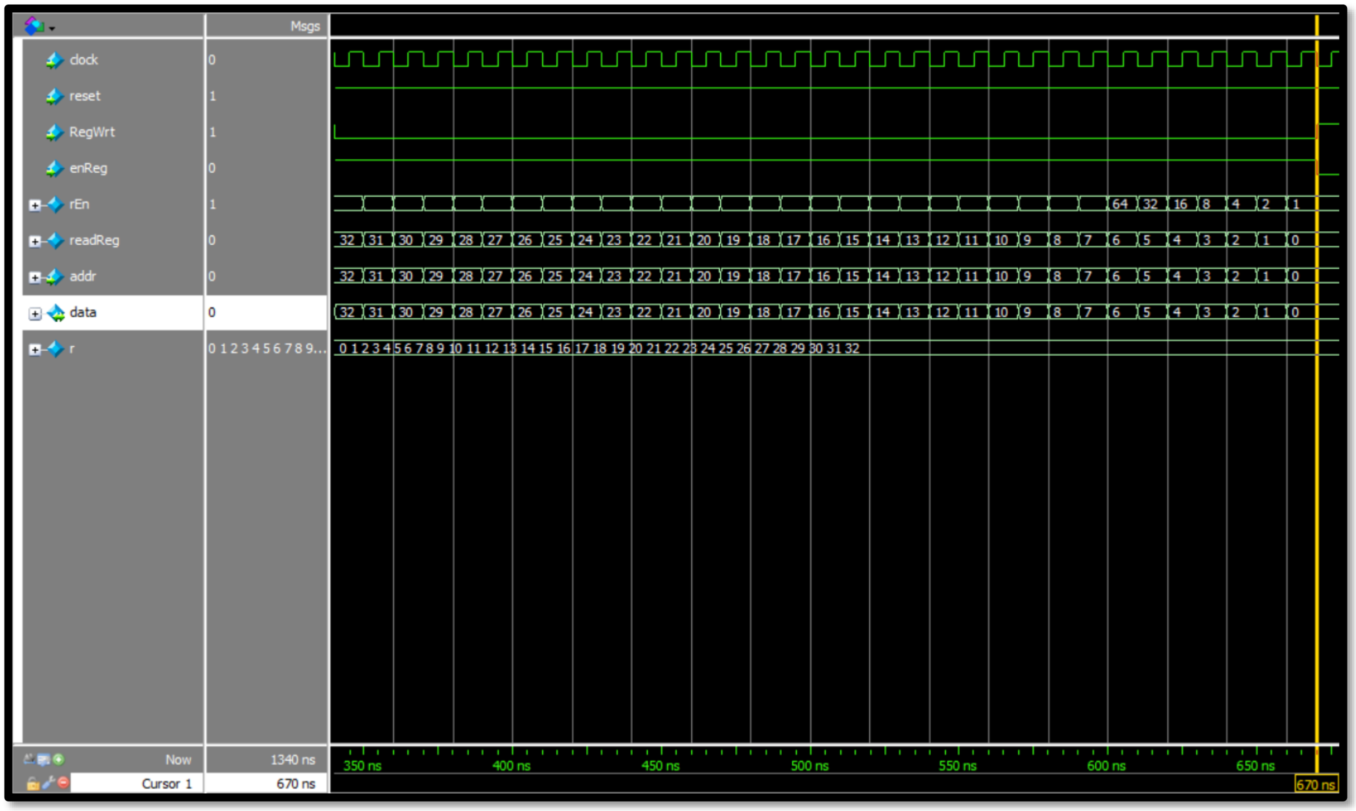
***Test case 1.1**: Writing on all registers their addresses as values while "RegWrt, enReg" signals are enabled.*



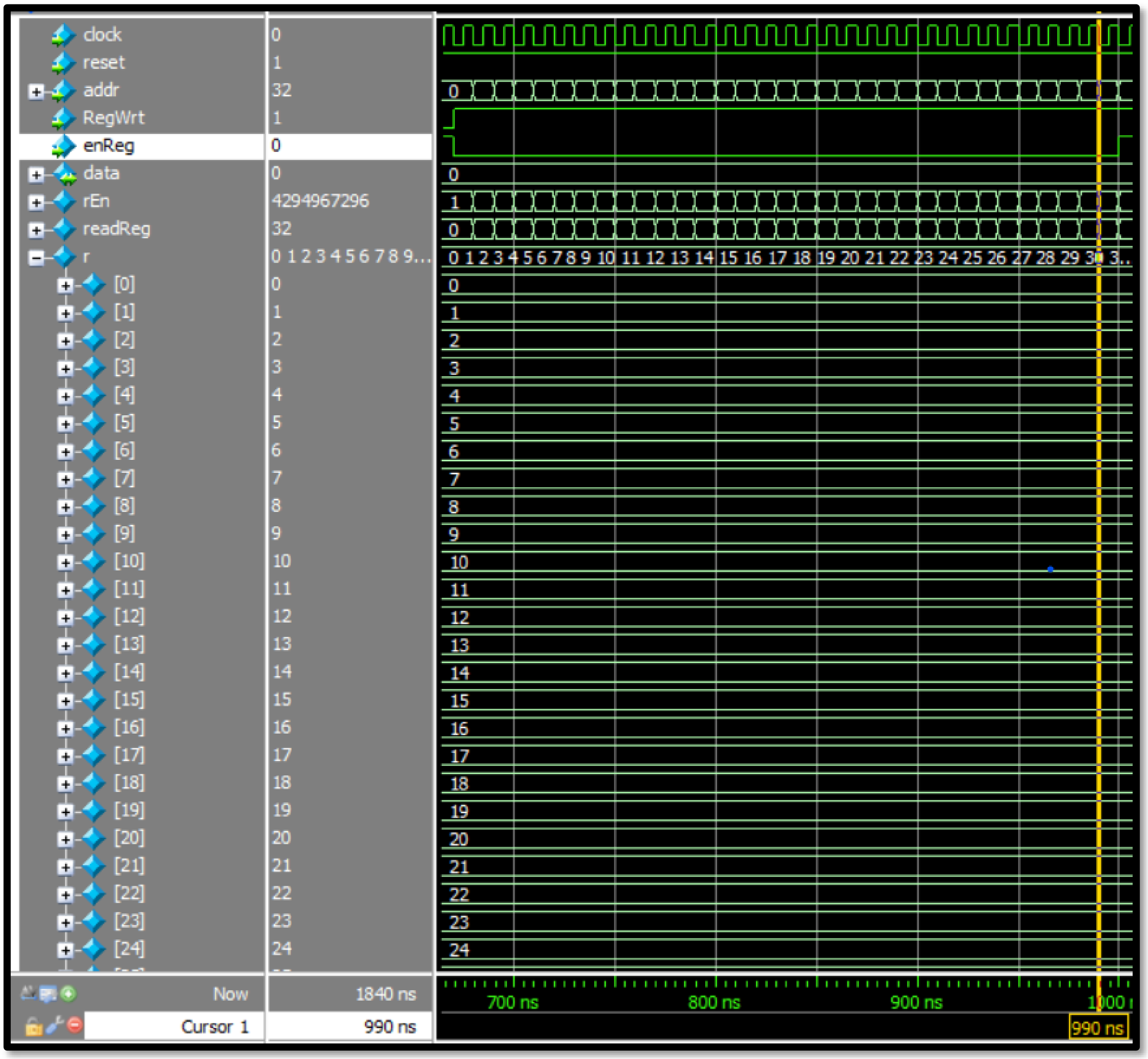
r	0 1 2 3 4 5 6 7 8 9...
[0]	0
[1]	1
[2]	2
[3]	3
[4]	4
[5]	5
[6]	6
[7]	7
[8]	8
[9]	9
[10]	10
[11]	11
[12]	12
[13]	13
[14]	14
[15]	15
[16]	16
[17]	17
[18]	18
[19]	19
[20]	20
[21]	21
[22]	22
[23]	23
[24]	24
[25]	25
[26]	26
[27]	27
[28]	28
[29]	29
[30]	30
[31]	31
[32]	32
Now	1340 ns
Cursor 1	325 ns

***Test case 1.2**: Read all register values in descending order after the write operation.*

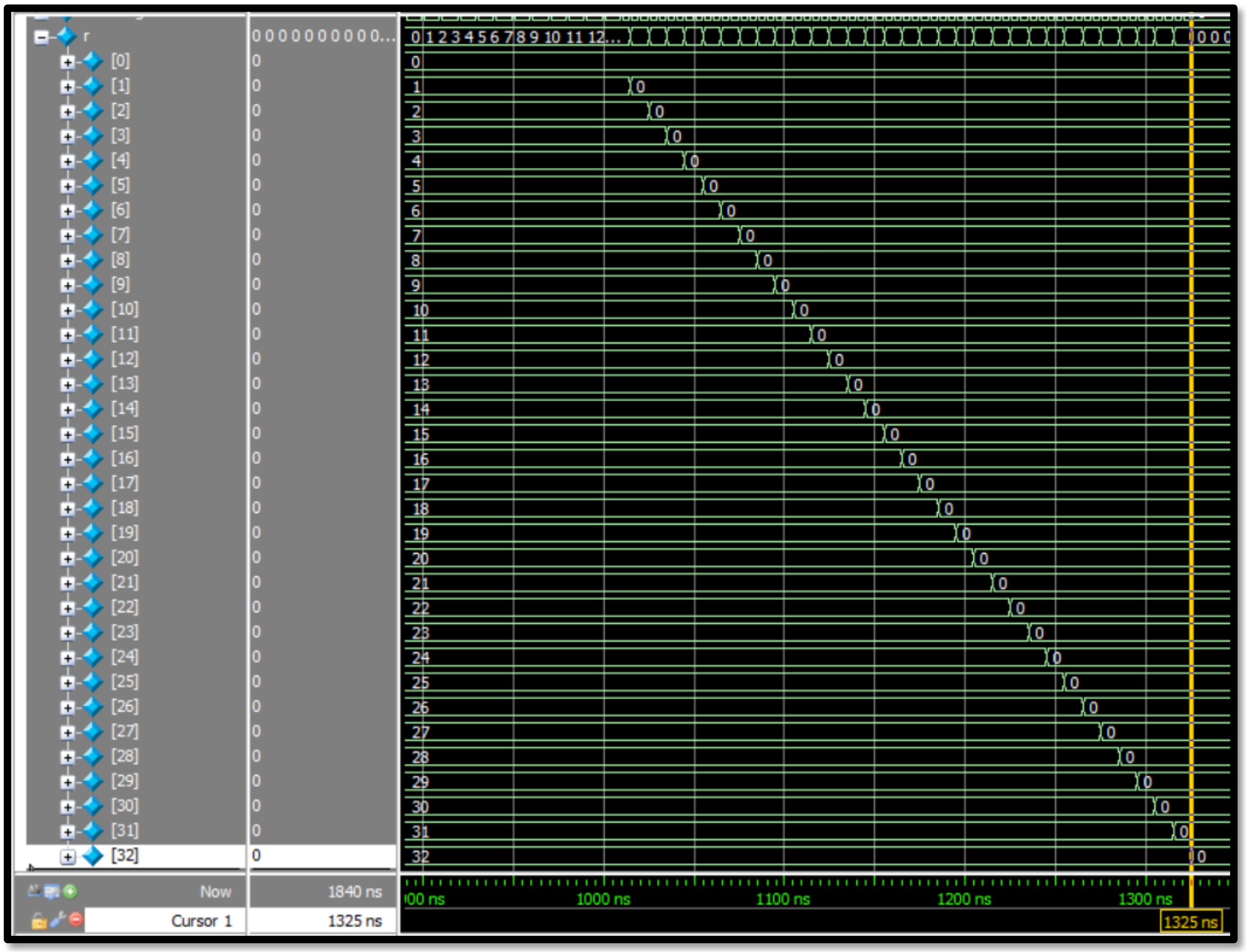
Note the signals at first, then see addresses and data are the same values



Test case 2.1: *Attempt to write zero value on all registers with the "enReg" signal deactivated.*
No write operation occurs, and register values remain unchanged.



```
clock      0
reset      1
addr       32
RegWrt     1
enReg      0
data       0
rEn        4294967296
readReg    32
r          0 1 2 3 4 5 6 7 8 9...
[0]        0
[1]        1
[2]        2
[3]        3
[4]        4
[5]        5
[6]        6
[7]        7
[8]        8
[9]        9
[10]       10
[11]       11
[12]       12
[13]       13
[14]       14
[15]       15
[16]       16
[17]       17
[18]       18
[19]       19
[20]       20
[21]       21
[22]       22
[23]       23
[24]       24
Now        1840 ns
Cursor 1   990 ns
```



Now 1840 ns

Cursor 1 1325 ns

2.2 ALU

This ALU (Arithmetic and Logical Unit) performs 10-operations which is enough to implement the majority of RISC-V instructions, these operands are special purpose register (A, B), to make the ALU operation.

Here is our 10 operations and their needed operations (This table might be changed later)

Operation	***Alu Op***	***Mirco Operation***
COPY_A	*4'b0000*	*A-> Bus*
COPY_B	*4'b0001*	*B -> Bus*
INC_A_1	*4'b0010*	*A+1 ->Bus*
DEC_A_1	*4'b0011*	*A-1 ->Bus*
INC_A_4	*4'b0100*	*A+4 -> Bus*
DEC_A_4	*4'b0101*	*A-4 -> Bus*
ADD	*4'b0110*	*A+B -> Bus*
SUB	*4'b0111*	*A-B -> Bus*
AND	*4'b1000*	*A&B ->Bus*
OR	*4'b1001*	*A | B ->Bus*

2.2.1 ALU Testing

Testing here is based on the opcode and each operation has been tested twice, first time to compute random values based on the inputs and opcode and the second time checking the zero flag along with testing its functionality.

***Test case**:*

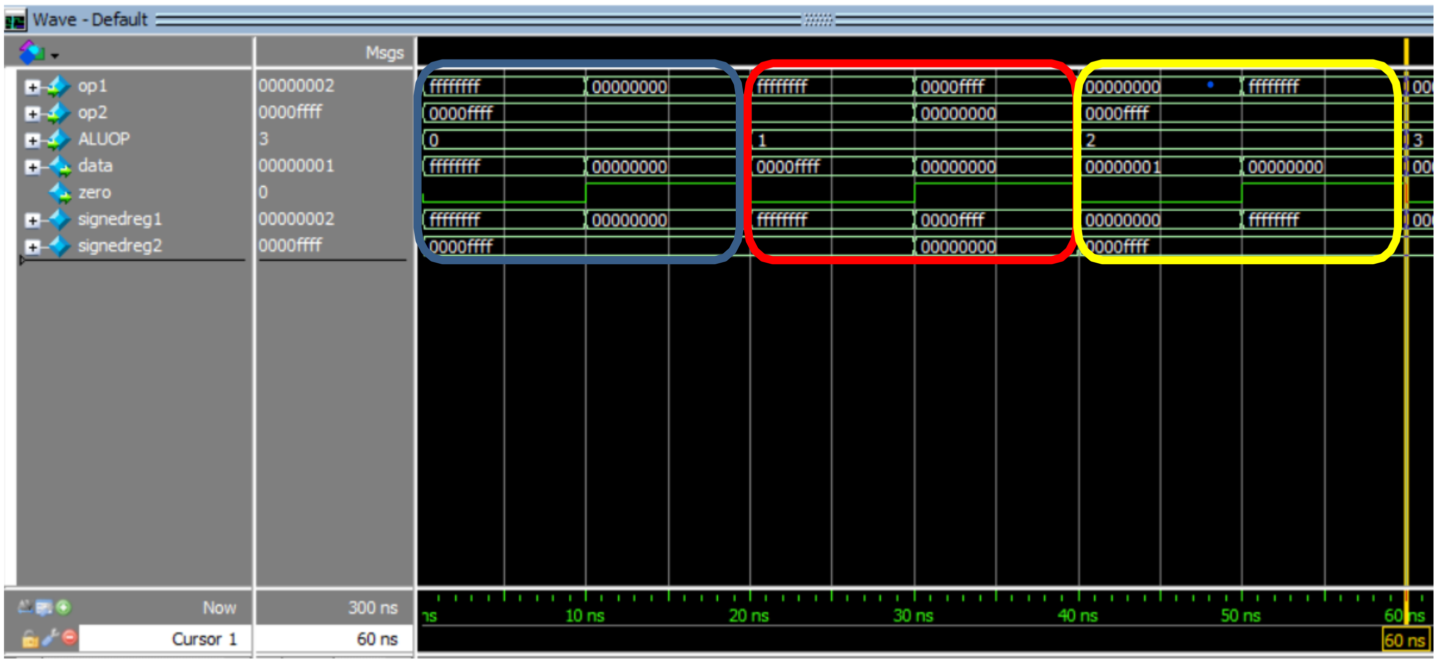
There are 3 test cases here:

<u>Note: All zero-flag testing has the same expected output.</u>

1: Testing COPY_A operation
Expected output: data =0xffffffff, zero = 0

2: Testing COPY_B operation
Expected output: data =0x0000ffff, zero = 0

3: Testing INC_A_1 operation
Expected output: data =0x00000001 , zero = 0

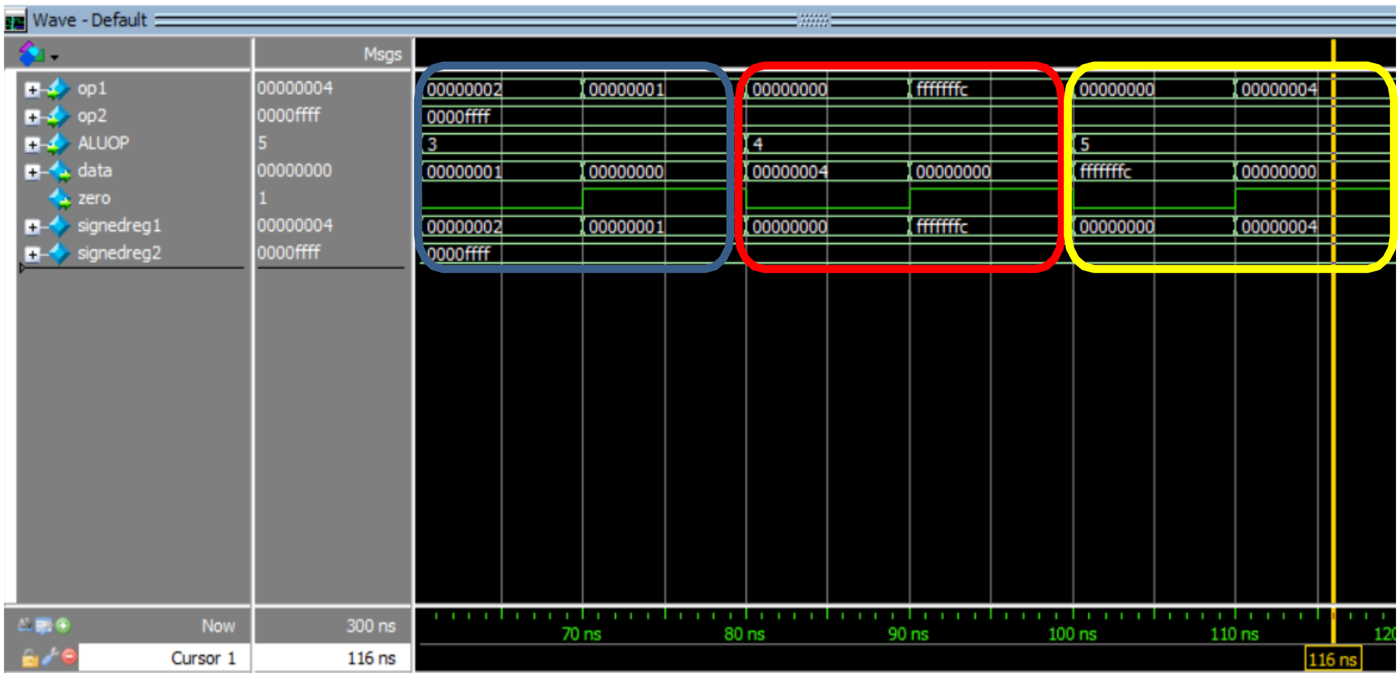


```
op1         00000002
op2         0000ffff
ALUOP       3
data        00000001
zero        0
signedreg1  00000002
signedreg2  0000ffff
```

4: Testing DEC_A_1 operation
Expected output: data =0x00000001, zero = 0

5: Testing INC_A_4 operation
Expected output: data =0x00000004, zero = 0

6: Testing DEC_A_4 operation
Expected output: data =0xfffffffc, zero = 0



```
op1         00000004   00000002 | 00000001 | 00000000 | fffffffc | 00000000 | 00000004
op2         0000ffff   0000ffff
ALUOP       5          3 | 4 | 5
data        00000000   00000001 | 00000000 | 00000004 | 00000000 | fffffffc | 00000000
zero        1
signedreg1  00000004   00000002 | 00000001 | 00000000 | fffffffc | 00000000 | 00000004
signedreg2  0000ffff   0000ffff
```

7: Testing ADD operation

Expected output: data =0xffffffff, zero = 0

8: Testing SUB operation

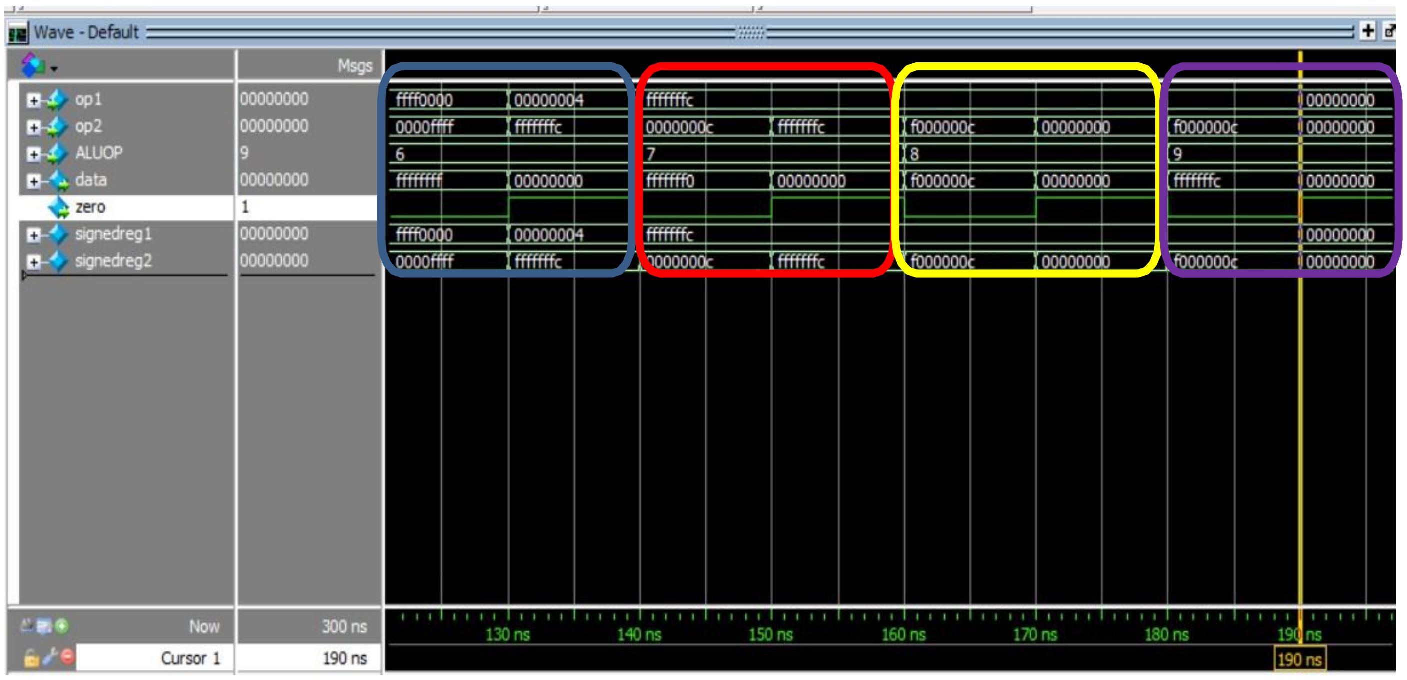
Expected output: data =0xfffffff0, zero = 0

9: Testing AND operation

Expected output: data =0xf000000c , zero = 0

10: Testing OR operation

Expected output: data =0xfffffffc, zero = 0

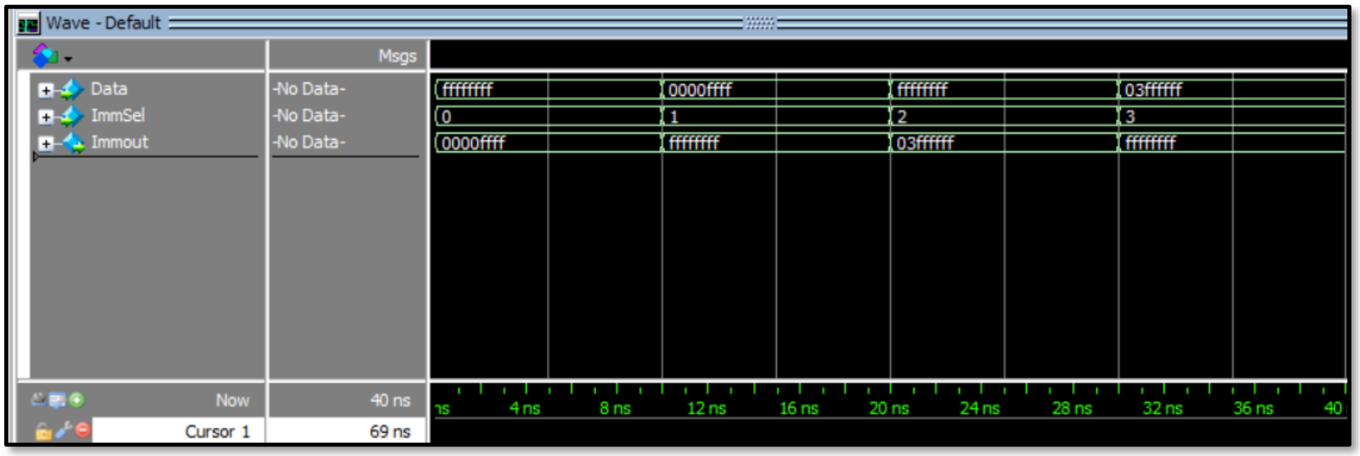

Wave - Default
Msgs
op1 00000000
op2 00000000
ALUOP 9
data 00000000
zero 1
signedreg1 00000000
signedreg2 00000000
ffff0000 00000004 fffffffc 00000000
0000ffff fffffffc 0000000c fffffffc f000000c 00000000 f000000c 00000000
6 7 8 9
ffffffff 00000000 fffffff0 00000000 f000000c 00000000 fffffffc 00000000
ffff0000 00000004 fffffffc 00000000
0000ffff fffffffc 0000000c fffffffc f000000c 00000000 f000000c 00000000
Now 300 ns
Cursor 1 190 ns
130 ns
140 ns
150 ns
160 ns
170 ns
180 ns
190 ns
190 ns

2.3 ImmGen

This module has four types of immediate extension (I,S,B,J) each one has its use in term of type of extension (Zero or signed) and how many bits is extended.
Immediate for I, S and B types (each 12 bits) extended , immediate f for J type (20 bits).
Also in case of J and B , the immediates shifted to left by 3 bits (one bit is the discarded lsb in the machine code , other 2 bits to multiply the immediate by 4 because our memory is byte addresable)
It has two inputs (Data,ImmSel), ImmSel is 2-bit selector to select the suitable extended immediate for each type.

2.3.1 ImmGen Testing

Test case 1: Data<= 32'hffffffff. ImmSel <= 2'b00. Expected output is 32'h0000ffff	Test case 2 Data <= 32'h0000ffff. ImmSel <= 2'b01. Expected output is 32'hffffffff	Test case 3 Data <= 32'hffffffff. ImmSel <= 2'b10. Expected output is 32'h03ffffff	Test case 4 Data <= 32'h03ffffff. ImmSel <= 2'b11. Expected output is 32'hffffffff



2.4 Memory

This memory is byte-addressable, big Indian scheme which contain 64-location (256 byte), it writes on posedge of the clock, while the enable is activated if MemWrt is 1, then it is a write, otherwise it's a read.
The first 32-word (128 byte) are reserved for the instructions and the other 32-words are for data .
For more realistic memory it has busy signal which indicates weather mamory still operating or not , to hold the processor from doing anything during access memory time.
The bus is here bidirectional wire (inout) controlled by tri-state buffer using the enable signal enMem and write signal MemWrt, if both are enabled the data is reading from the memory, else it will be Z to be overwritten.

Componen t	***Description***
Memory	*256-byte location for ROM and RAM (Can be extended)*
Address Input	*32-bit address (addr), determines which location is to be read or written to*
Data Port	*Bidirectional, 32-bit*
Control Signals	*MemWrt: Determines read/write operation (1 for write, 0 for read)* *enMem: General enable control, If enMem is 1, then the register reads or writes depending on MemWrt . If enMem is 0, then nothing is done*
Read Operation	*Asynchronous*
Write Operation	*On positive edge of clock*
Tri-state Buffer	*2 Tri-state Buffers, in case of reading from memory the write buffer puts the port on z state to allow the reading buffer putting data on the port,* *in case of writing on memory, the reading buffer puts the port on z state allowing* *the writing buffer to put data on the port*

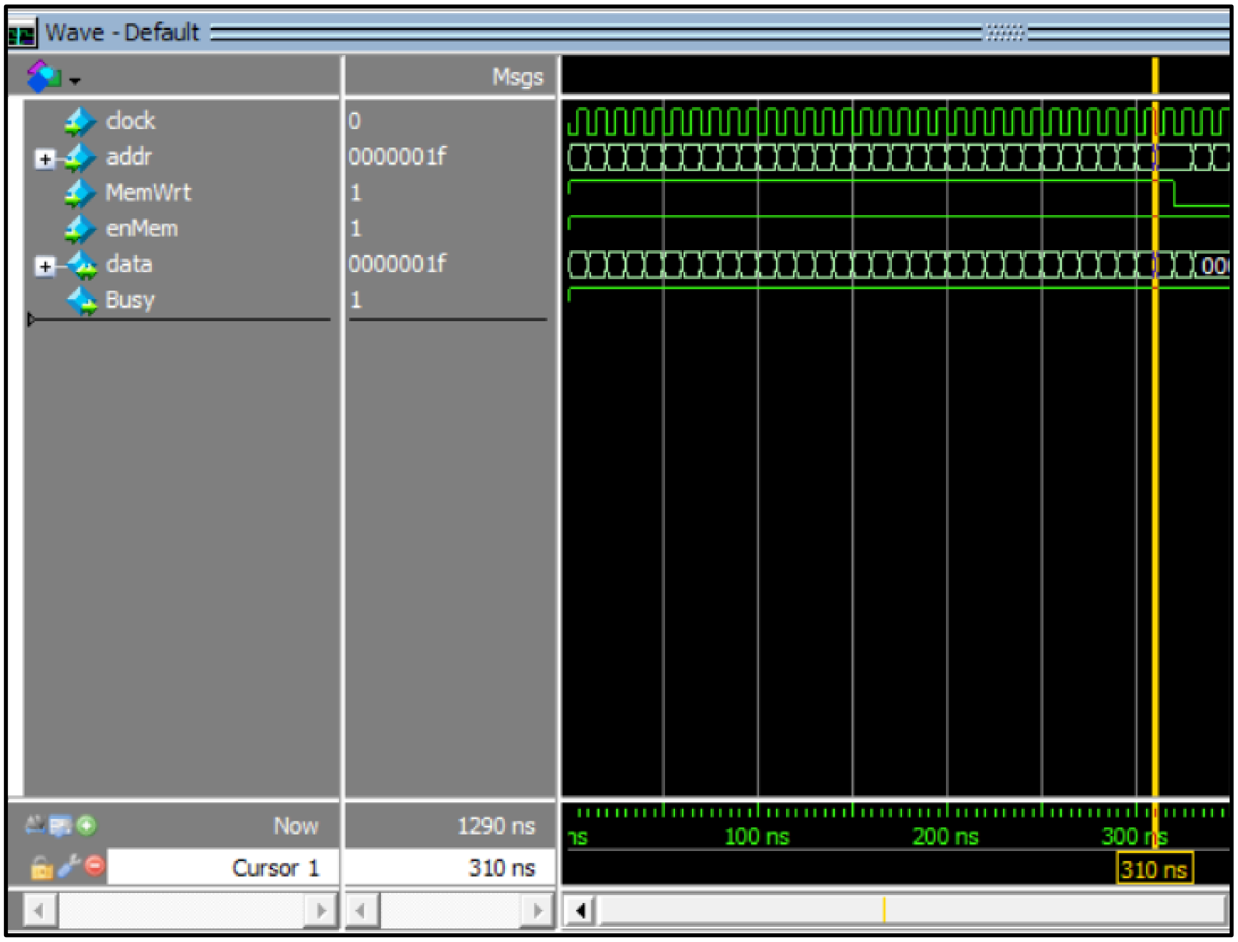
2.4.1 Memory Testing

For memory testing we made the memory 32-byte location, and this is the testing criteria:

1. *Writing on all locations with location address.*
2. *Reading from all locations (backward).*
3. *Try to write zero on all locations while enable is deactivated.*
4. *Write zero on all registers.*

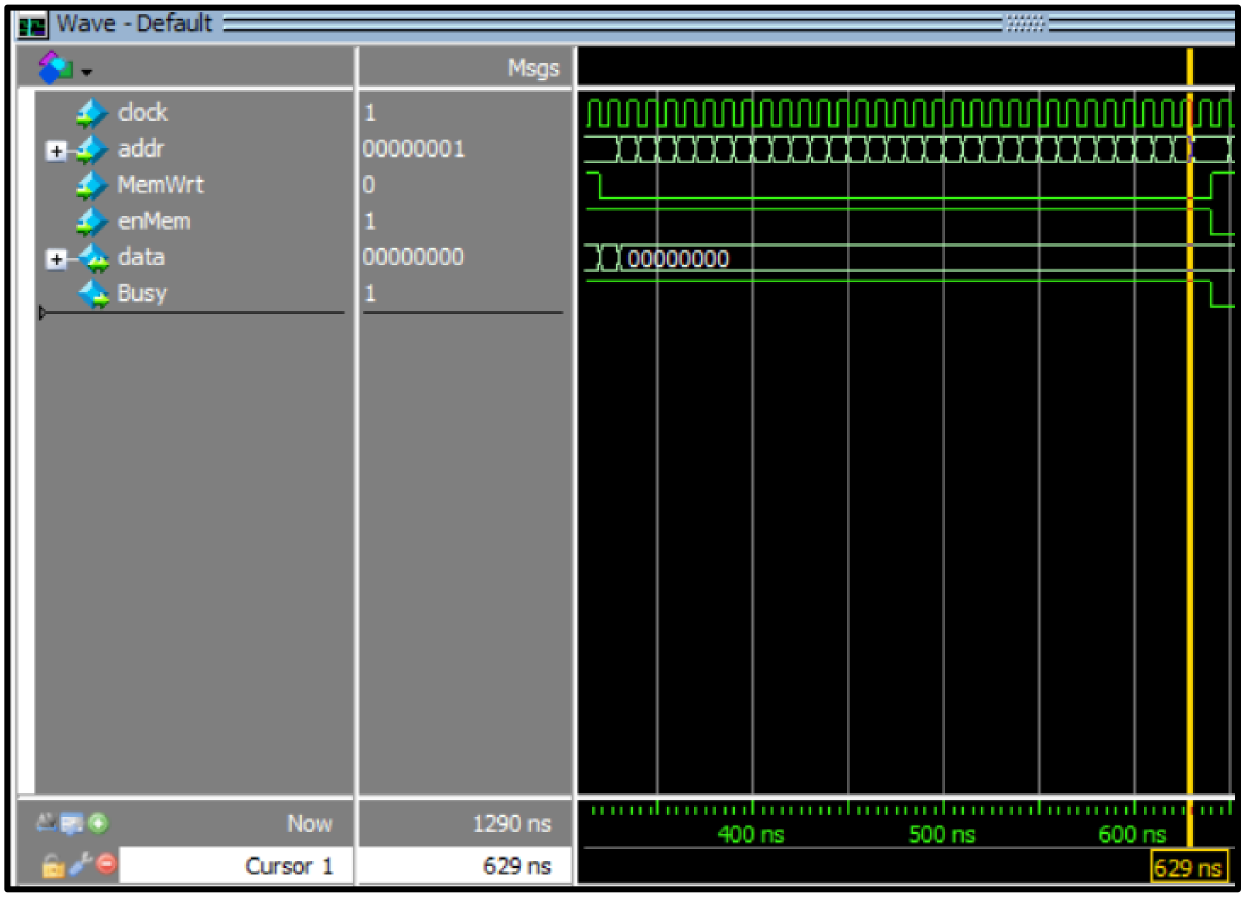
And by checking each location value after writing or checking the output after reading and the busy signal if the enable is 1 then it is activated, for each testcase notice the input signal which indicates write or reading.

1. *Writing on all locations with location address, MemWrt ,enMem =(1,1).*



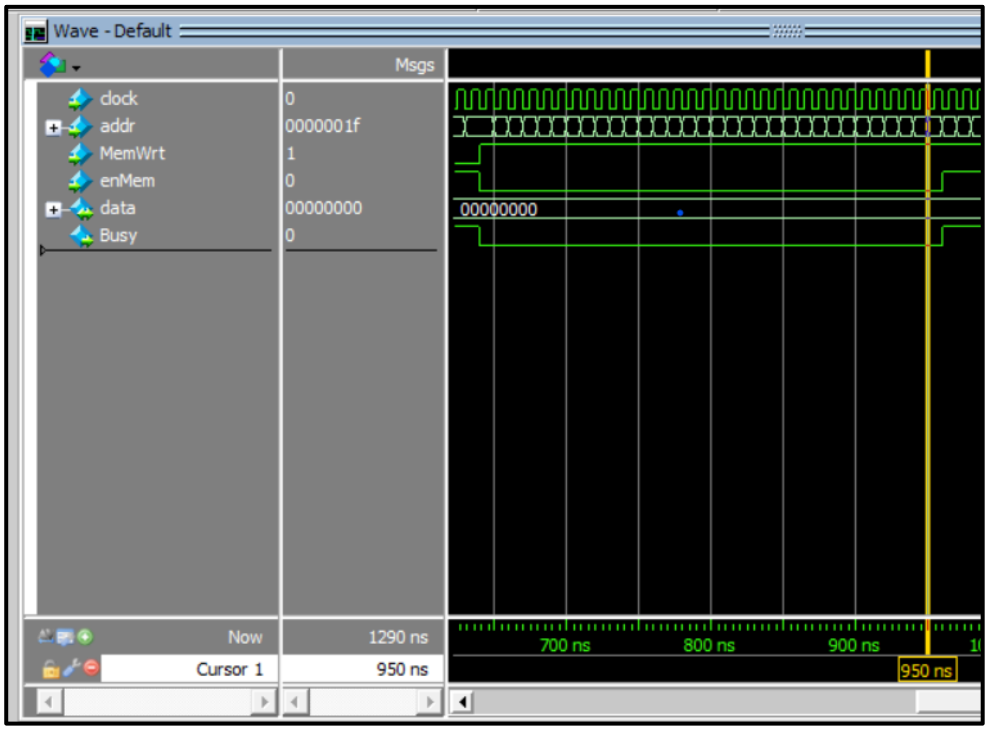
clock	0
addr	0000001f
MemWrt	1
enMem	1
data	0000001f
Busy	1

2. *Reading from all locations (backward), MemWrt ,enMem =(0,1).*

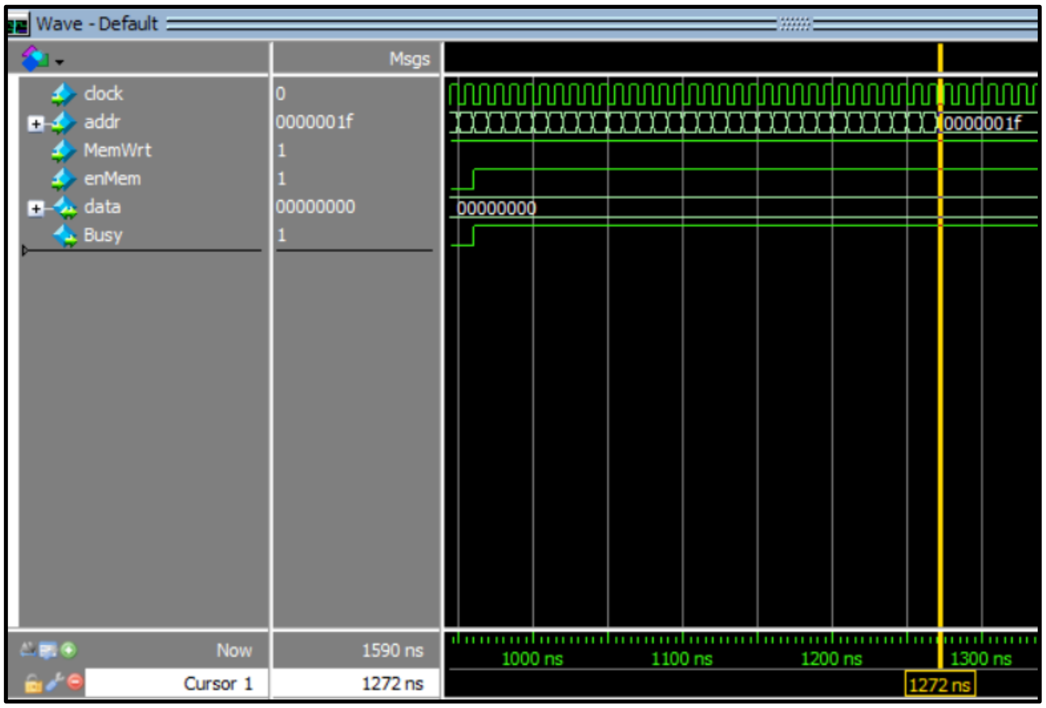


```
clock    1
addr     00000001
MemWrt   0
enMem    1
data     00000000
Busy     1
Now      1290 ns
Cursor 1 629 ns
```

3. *Try to write zero on all locations while enable is deactivated* *MemWrt ,enMem =(1,0), note the data is forced at zero*



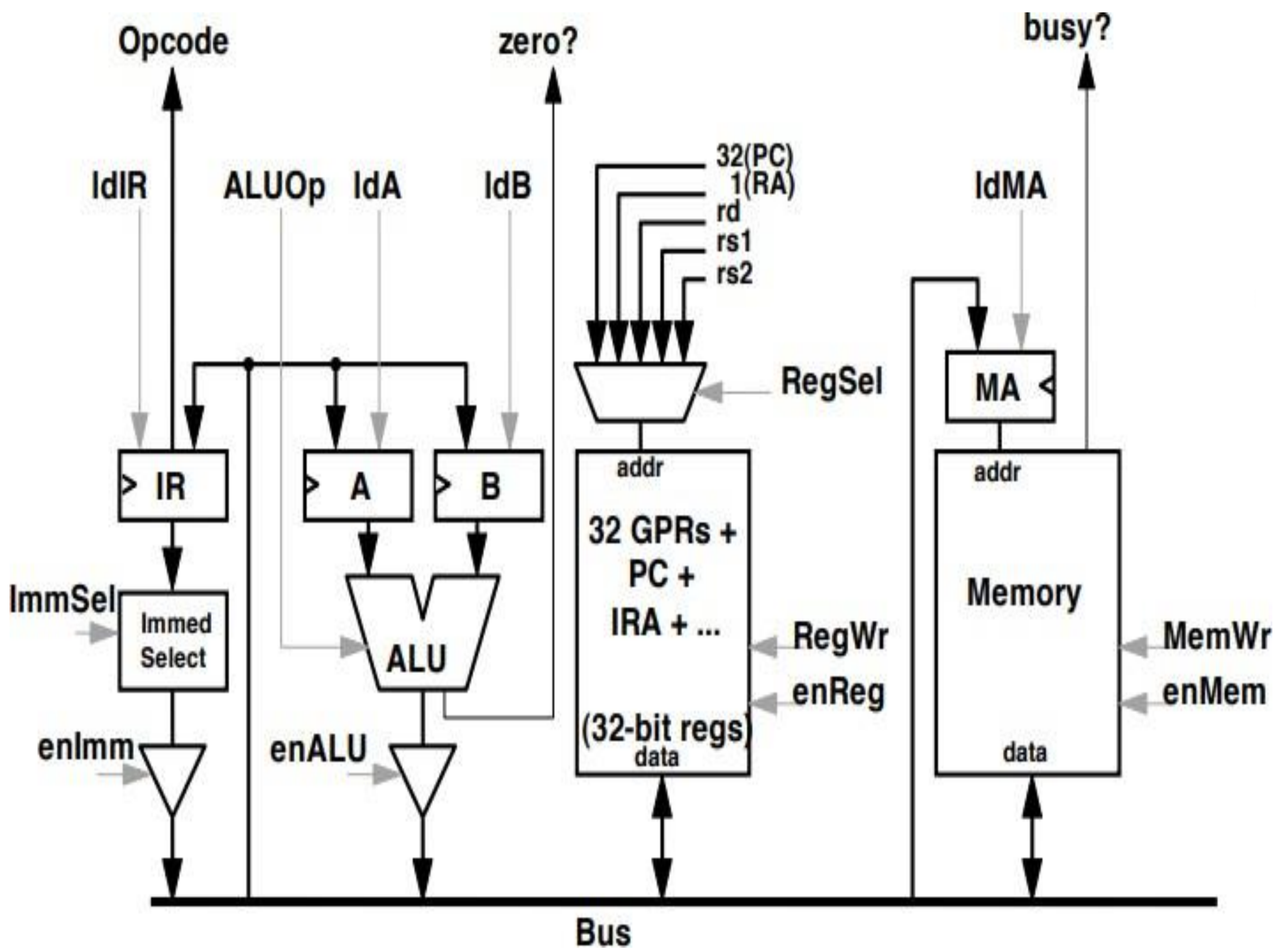
4. *Write zero on all registers, deactivated MemWrt ,enMem =(1,1).*



3. Data path

a bus-based implementation of the RISC-V architecture. In this architecture, the different components of the machine share a common 32-bit bus through which they communicate. Control signals determine how each of these components is used and which components get to use the bus during a particular clock cycle.

Schematic Design:



Supported instructions and their microcodes:

1) ADD instructions:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
ADD0	A -> REG[RS]	0	RS	0	1	1	X	X	0	X	X	0	X	0	N	X
ADD1	B -> REG[RT}	0	RT	0	1	0	1	X	0	X	X	0	X	0	N	X
ADD2	REG[RD] -> A+B	0	RD	1	1	0	0	ADD	1	X	X	0	X	0	J	FETCH

2) SUB instruction:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
SUB0	A -> REG[RS]	0	RS	0	1	1	X	X	0	X	X	0	X	0	N	X
SUB1	B -> REG[RT}	0	RT	0	1	0	1	X	0	X	X	0	X	0	N	X
SUB2	REG[RD] -> A+B	0	RD	1	1	0	0	SUB	1	X	X	0	X	0	J	FETCH

3) ADDI instruction:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
ADDI3	A -> REG[RS]	0	rs	0	1	1	x	x	0	x	x	0	x	0	N	X
ADDI4	B -> imm	0	x	x	0	0	1	x	0	x	x	0	imm_I	1	N	X
ADDI5	REG[RD] -> A+B	0	rd	1	1	0	0	ADD	1	0	x	0	x	0	N	X
ADDI6		0	X	X	0	X	X	X	0	X	X	0	X	0	J	FETCH0

4) LW instruction:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
LW0	A -> OFFSET	0	X	X	0	1	X	X	0	X	X	0	SE16	1	N	X
LW1	B -> REG[RS]	0	RS	0	1	0	1	X	0	X	X	0	X	0	N	X
LW2	MA -> A+B	0	X	X	0	0	0	ADD	1	1	X	0	X	0	N	X
LW3	REG[RT] -> MEM	0	RT	1	1	X	X	X	0	0	0	1	X	0	S	
LW4		0	X	X	0	X	X	X	0	X	X	0	X	0	J	FETCH0

5) SW instruction

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
SW0	A -> OFFSET	0	X	X	0	1	X	X	0	X	X	0	SE16	1	N	X
SW1	B -> REG[RS]	0	RS	0	1	0	1	X	0	X	X	0	X	0	N	X
SW2	MA -> A+B	0	X	X	0	0	0	ADD	1	1	X	0	X	0	N	X
SW3	MEM-> REG[RT]	0	RT	0	1	X	X	X	0	0	1	1	X	0	S	
SW4		0	X	X	0	X	X	X	0	X	X	0	X	0	J	FETCH0

6) Jump instruction:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
	A -> PC															
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
J0	B -> immJ	0	X	X	0	1	1	X	0	X	X	0	imm_j	1	N	X
J1	PC -> A + B	0	pc	1	1	0	0	ADD	1	X	X	0	X	0	N	X
J3		0	X	X	0	X	X	X	0	X	X	0	X	0	J	FETCH0

7) BEQ instruction:

State	PseudoCode	ld IR	RegSel	RegW	enReg	ld A	ld B	ALUOp	enALU	ldMA	MemW	enMem	EXSel	enImm	MBr	Nest State
FETCH0	MA -> PC A -> PC	0	PC	0	1	1	X	X	0	1	X	0	X	0	N	X
FETCH1	IR -> MEM[MA]	1	X	X	0	0	X	X	0	0	0	1	X	0	S	X
FETCH2	PC -> PC(A) + 4	0	PC	1	1	0	X	INC_A_4	1	X	X	0	X	0	D	X
DISPATCH																
BEQ0	A -> REG[RS]	0	RS	0	1	1	X	X	0	X	X	0	X	0	N	X
BEQ1	B -> REG[RT]	0	RT	0	1	0	1	X	0	X	X	0	X	0	N	X
BEQ2	A -> A-B	0	X	X	0	1	0	SUB	1	X	X	0	X	0	FNEZ	FETCH0
BEQ3	A -> REG[PC]	0	PC	0	1	1	0	X	0	X	X	0	X	0	N	X
BEQ4	B -> imm	0	X	X	0	0	1	X	0	X	X	0	imm_B	1	N	X
BEQ5	REG[PC] -> A + B	0	pc	1	1	0	0	ADD	1	x	x	0	x	0	N	x
		0	X	X	0	X	X	X	0	X	X	0	X	0	J	FETCH0

Datapath testing

In order to test each instruction without control unit , we have to pass inputs control signals manually to the data path in the test bench .

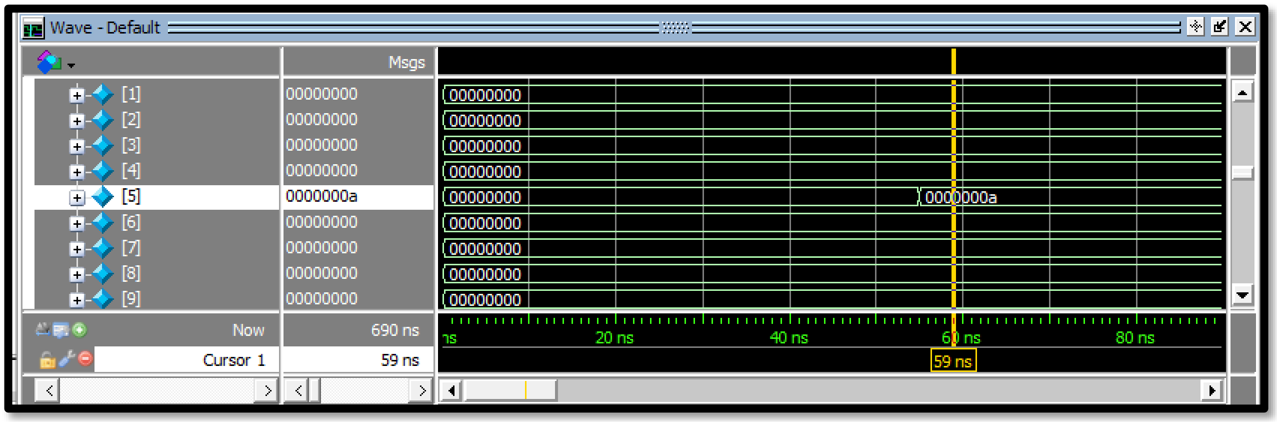
First initialize data memory with instructions and data

Machine codes found in "Memory_init .txt" also

```
Word0:      00 A0 02 93   //ADDI X5,X0,10
Word1:      00 50 03 B3   //ADD X7,X5,X0
Word2:      40 72 82 B3   //SUB X5,X5,X7
Word3:      08 00 24 03   //LW X8,128(X0)
Word4:      08 80 22 23   // SW X8, 132(X0)
Word5:      00 40 00 6F   // J 4
Word6:      00 00 00 00
Word7:      00 00 00 00
Word8:      00 00 00 00
Word9:      00 A0 02 93   // ADDI X5,X0,10
Word10  :   00 72 82 63   // BEQ x5,x7,5
Word11:     01 40 02 93   //ADDI X5,X0,20
Word12:     00 00 00 00
Word13:     00 00 00 00
Word14:     00 00 00 00
Word15:     40 72 82 B3   //SUB X5,X5,X7
. . . . . . . . .
Word32:     11 00 11 FF //Data
```

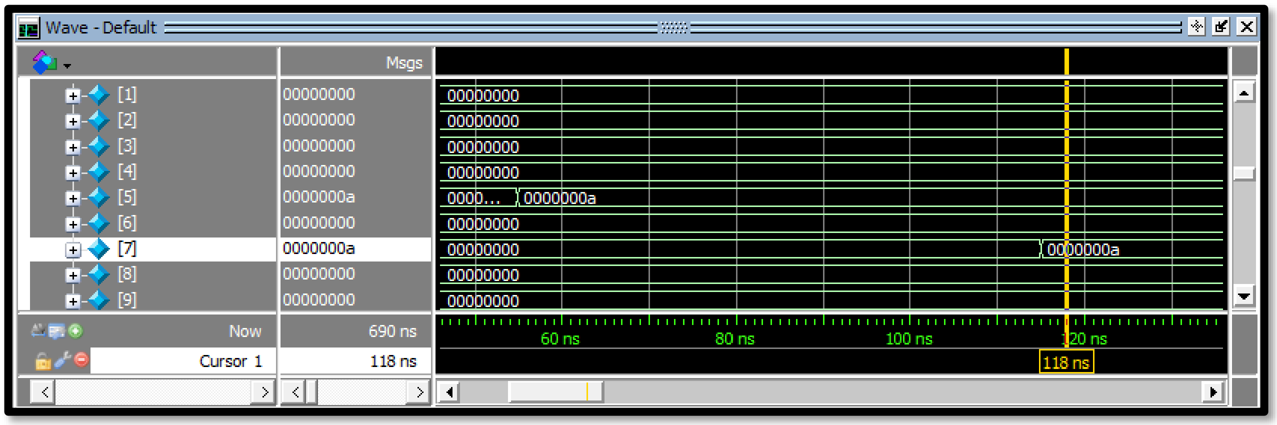
1) Final Results after executing first instruction:

ADDI X5,X0,10
Expected to write value of 10 on register x5


```
Wave - Default
                Msgs
[1]   00000000   00000000
[2]   00000000   00000000
[3]   00000000   00000000
[4]   00000000   00000000
[5]   0000000a   00000000  0000000a
[6]   00000000   00000000
[7]   00000000   00000000
[8]   00000000   00000000
[9]   00000000   00000000
Now       690 ns   ns   20 ns   40 ns   60 ns   80 ns
Cursor 1   59 ns   59 ns
```

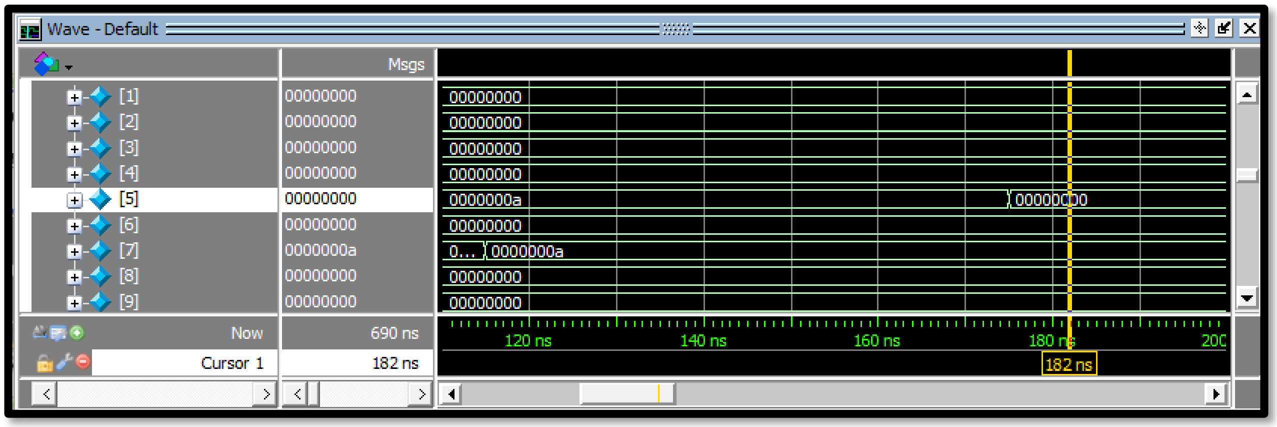
2) Executing second instruction :

ADD X7,X5,X0
Expected to copy value of register x5 to register x7 which is 10


```
Wave - Default
                Msgs
[1]   00000000   00000000
[2]   00000000   00000000
[3]   00000000   00000000
[4]   00000000   00000000
[5]   0000000a   0000...  0000000a
[6]   00000000   00000000
[7]   0000000a   00000000  0000000a
[8]   00000000   00000000
[9]   00000000   00000000
Now       690 ns   60 ns   80 ns   100 ns   120 ns
Cursor 1  118 ns   118 ns
```

3) Executing third instruction:

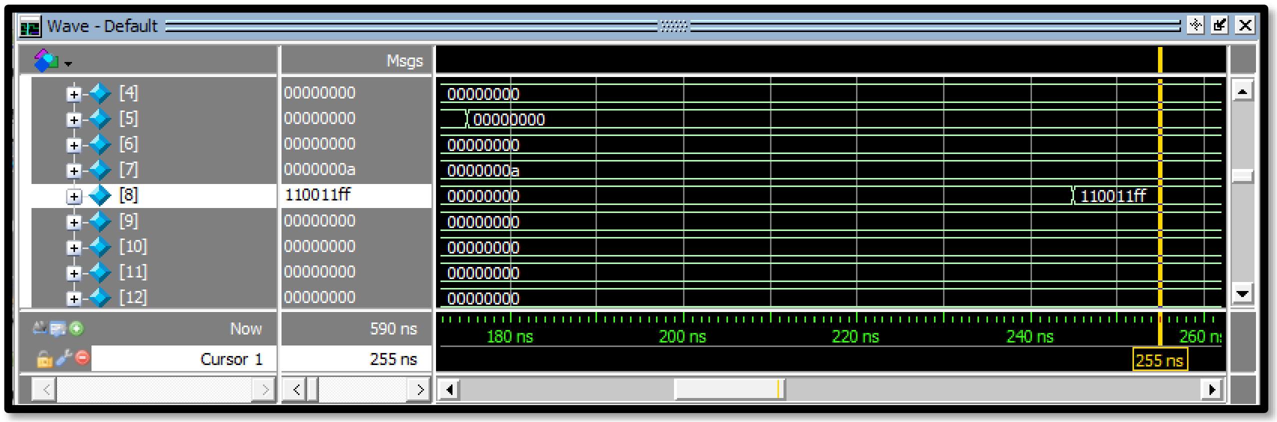
SUB X5,X5,X7
Expected to clear register x5



```
[1]  00000000
[2]  00000000
[3]  00000000
[4]  00000000
[5]  00000000
[6]  00000000
[7]  0000000a
[8]  00000000
[9]  00000000
Now       690 ns
Cursor 1  182 ns
```

4) Executing fourth instruction:

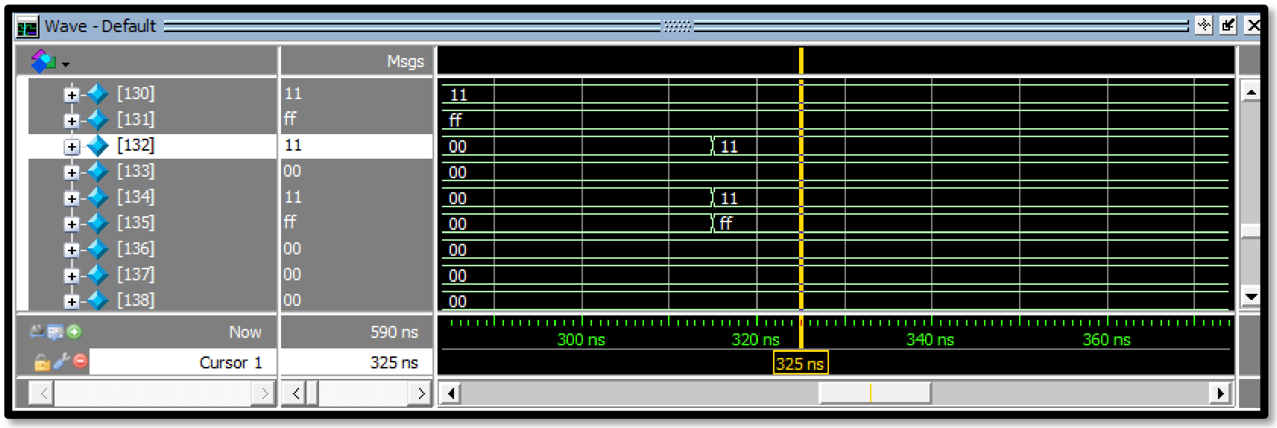
LW X8,128(X0)
Expected to load value (0x 11 00 11 ff) from memory into register x8 (0x 11 00 11 ff stored at word 32 in memory)

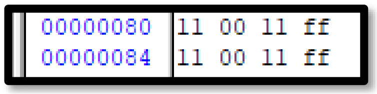


```
[4]   00000000
[5]   00000000
[6]   00000000
[7]   0000000a
[8]   110011ff
[9]   00000000
[10]  00000000
[11]  00000000
[12]  00000000
Now       590 ns
Cursor 1  255 ns
```

5) *Executing Fifth instructions:*

SW X8, 132(X0)
Expected to store value of register X8(0x 11 00 11 ff) into word 33 of memory

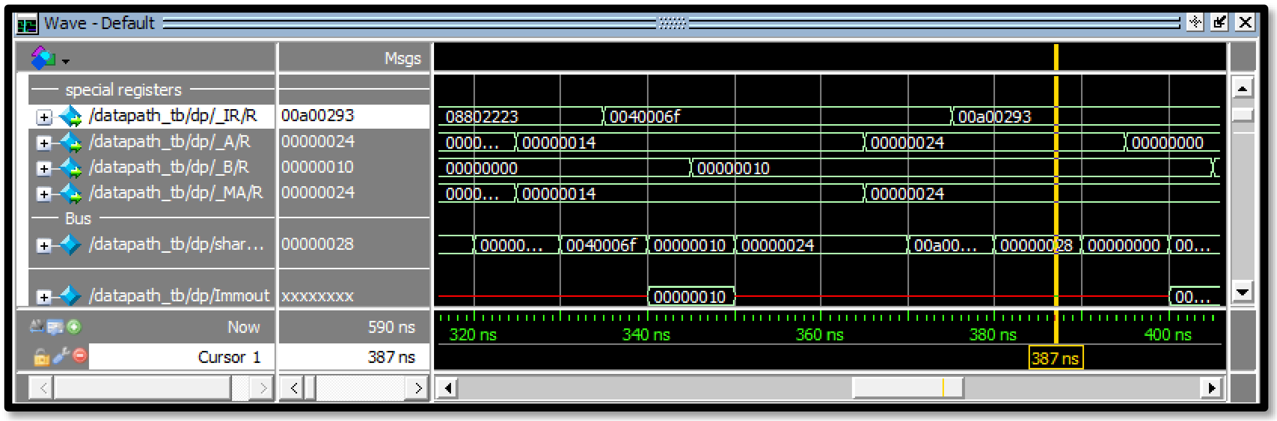




```
00000080  11 00 11 ff
00000084  11 00 11 ff
```

6) *Executing sixth instruction:*

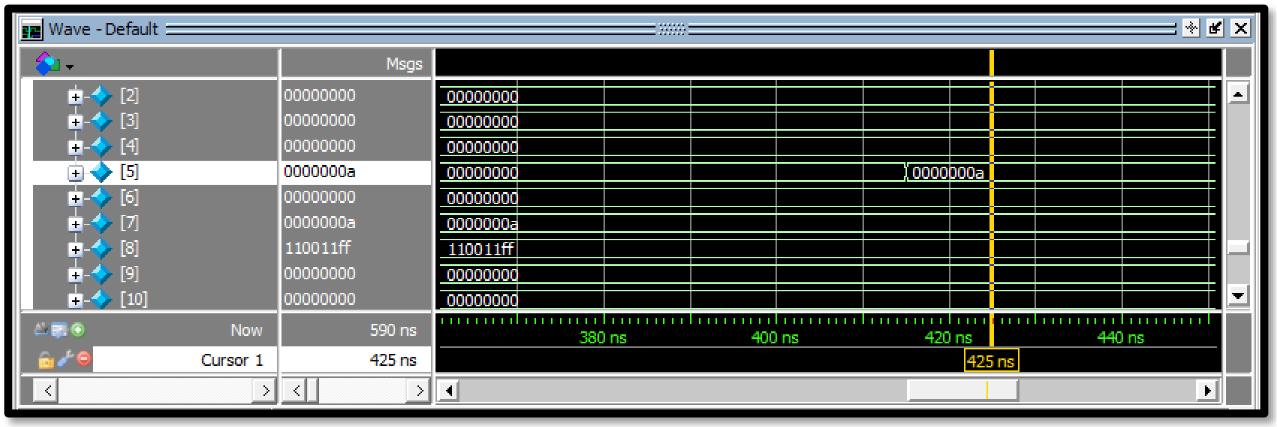
J 0x4
Expected to jump to instruction at word 9 of memory
And fetching (ADDI X5,X0,10) instruction



7) Executing seventh instruction:

ADDI X5,X0,10

Expected to write value of 10 on register x5 again

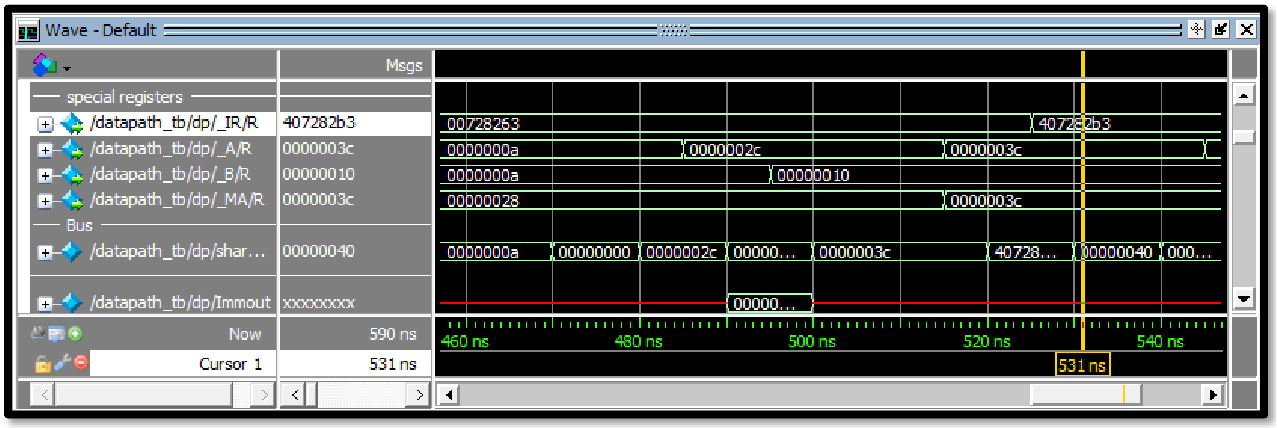


```
Wave - Default
                Msgs
[2]             00000000    00000000
[3]             00000000    00000000
[4]             00000000    00000000
[5]             0000000a    00000000            0000000a
[6]             00000000    00000000
[7]             0000000a    0000000a
[8]             110011ff    110011ff
[9]             00000000    00000000
[10]            00000000    00000000
Now             590 ns      380 ns   400 ns   420 ns   440 ns
Cursor 1        425 ns                        425 ns
```

8) Executing Eighth instruction

BEQ x5,x7,5

While X5 == X7 = > 10, Expected to jump to instruction in word 16 (SUB X5,X5,X7) instead of fetching next instruction (ADDI X5,X0,20)

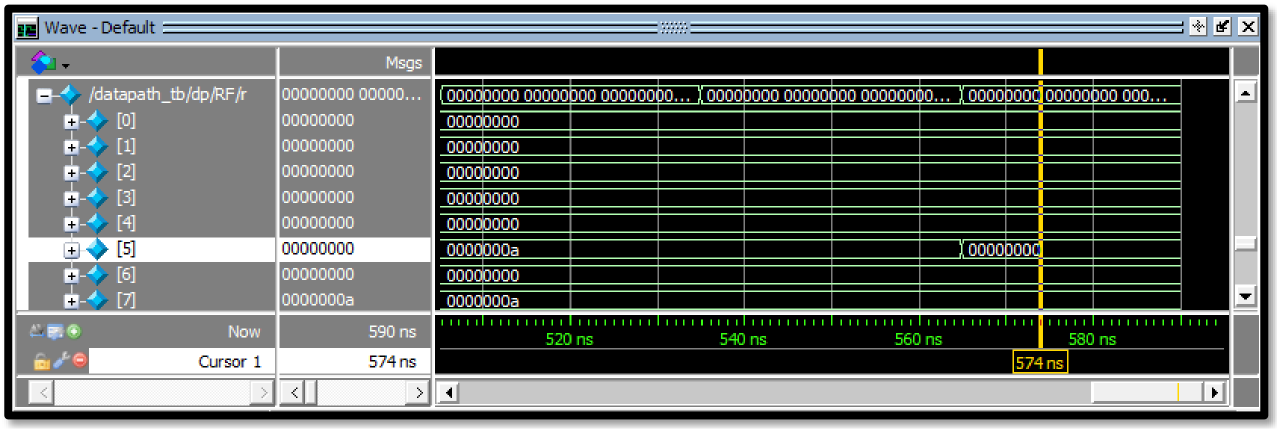


```
Wave - Default
                              Msgs
special registers
/datapath_tb/dp/_IR/R         407282b3    00728263                                407282b3
/datapath_tb/dp/_A/R          0000003c    0000000a       0000002c        0000003c
/datapath_tb/dp/_B/R          00000010    0000000a            00000010
/datapath_tb/dp/_MA/R         0000003c    00000028                       0000003c
Bus
/datapath_tb/dp/shar...       00000040    0000000a  00000000  0000002c  00000...  0000003c  40728...  00000040  000...
/datapath_tb/dp/Immout        xxxxxxxx                          00000...
Now                           590 ns      460 ns   480 ns   500 ns   520 ns   540 ns
Cursor 1                      531 ns                                 531 ns
```

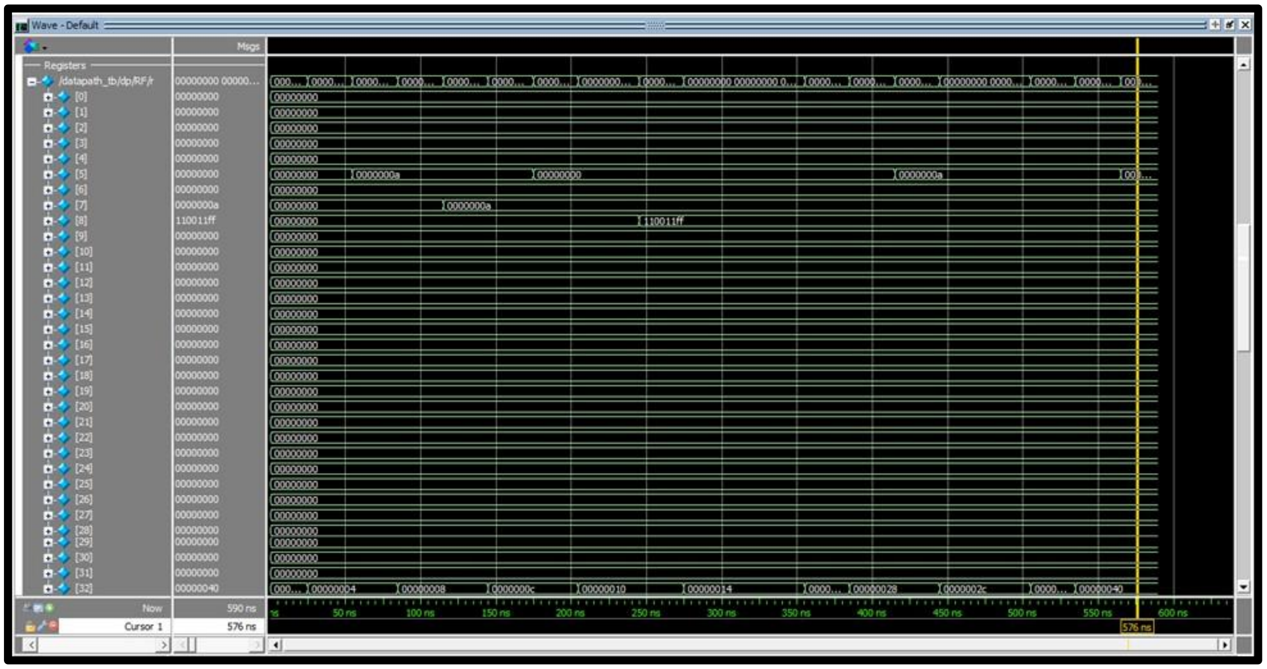
9) Executing ninth and last instruction

SUB X5,X5,X7

Expected to clear register X5 again

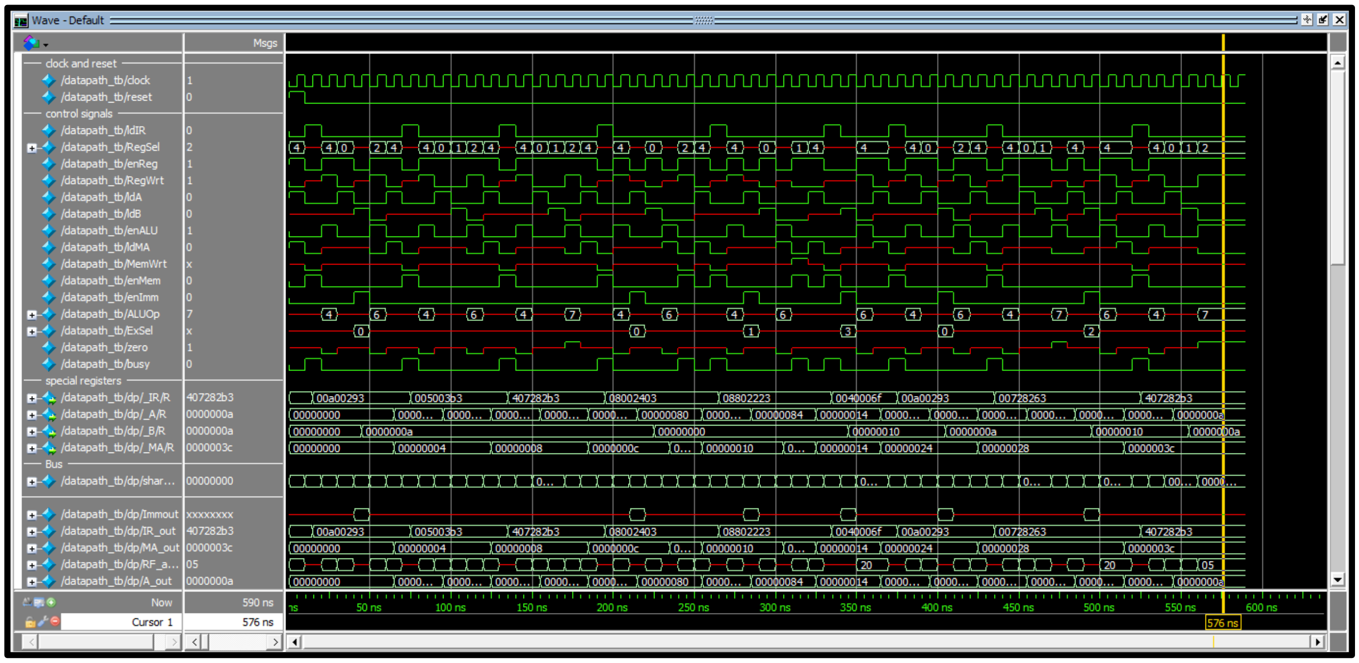


- Final view of registers values in register file



General view of the waveforms:

Note : you can find .do file for the waveform signal (wave.do)



Micro coded Control Unit

ROM-based microcoded control unit
manages the CPU by storing micro-operations in a ROM, each representing a control signal sequence for executing specific instructions. The first three bits of each micro-operation indicate the next state, enabling the control unit to fetch subsequent micro-operations.

Micro-Operation Storage

The micro-operations are stored in a ROM, with each location containing a single microcode. The organization of the ROM is as follows:

- **Address 0**: Reset operation
- **Addresses 1 to 3**: Fetch microcodes
- **Address 4**: Dispatch operation
- **Addresses 5 to 7**: ADD instruction microcodes
- **Addresses 8 to 10**: SUB instruction microcodes
- **Addresses 11 to 13**: ADDI instruction microcodes
- **Addresses 14 to 17**: LW instruction microcodes
- **Addresses 18 to 21**: SW instruction microcodes
- **Addresses 22 to 27**: BEQ instruction microcodes
- **Addresses 28 to 30**: JAL instruction microcodes

Micro Program Counter (μPC)

The **μPC** determines the address of the micro-operation to fetch from the ROM. The **μPC** is updated based on the **nextState** value, which is derived from the first three bits of the current micro-operation.

State Definitions

- **Fetch**: Initiates the fetch operation by setting the μPC to 1.
- **Next**: Increments the μPC to point to the next microcode.
- **Dispatch**: Uses the OP_GROUPS value to set the μPC, determining the starting address of the microcodes for the current instruction.
- **FNEZ**: Checks if the zero flag is not set. If true, resets the μPC; otherwise, increments it by 1.
- **FEQZ**: Checks if the zero flag is set. If true, increments the μPC by 1; otherwise, resets it.

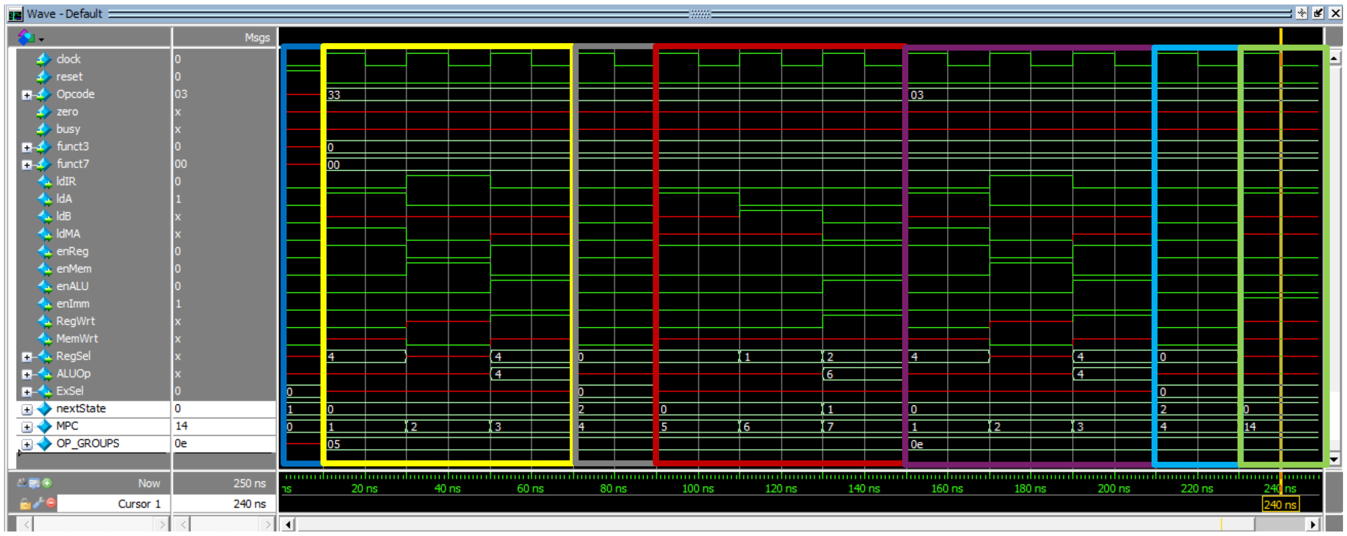
Dispatch Operation

The dispatch operation depends on the opcode and function fields of the current instruction. The value of OP_GROUPS is determined based on these fields and holds the starting address of the microcodes for the instruction to be executed.

Testing control unit :

Observe the value of MPC how it move from micro code to another

Reset	Fetch	Dispatch	Execute ADD Instruction	Fetching next instruction (LW)	Dispatch	Execute (LW) Instruction



	Msgs
clock	0
reset	0
Opcode	03
zero	x
busy	x
funct3	0
funct7	00
ldIR	0
ldA	1
ldB	x
ldMA	x
enReg	0
enMem	0
enALU	0
enImm	1
RegWrt	x
MemWrt	x
RegSel	x
ALUOp	x
ExSel	0
nextState	0
MPC	14
OP_GROUPS	0e
Now	250 ns
Cursor 1	240 ns

Rest of instructions will be tested on the data path with control unit

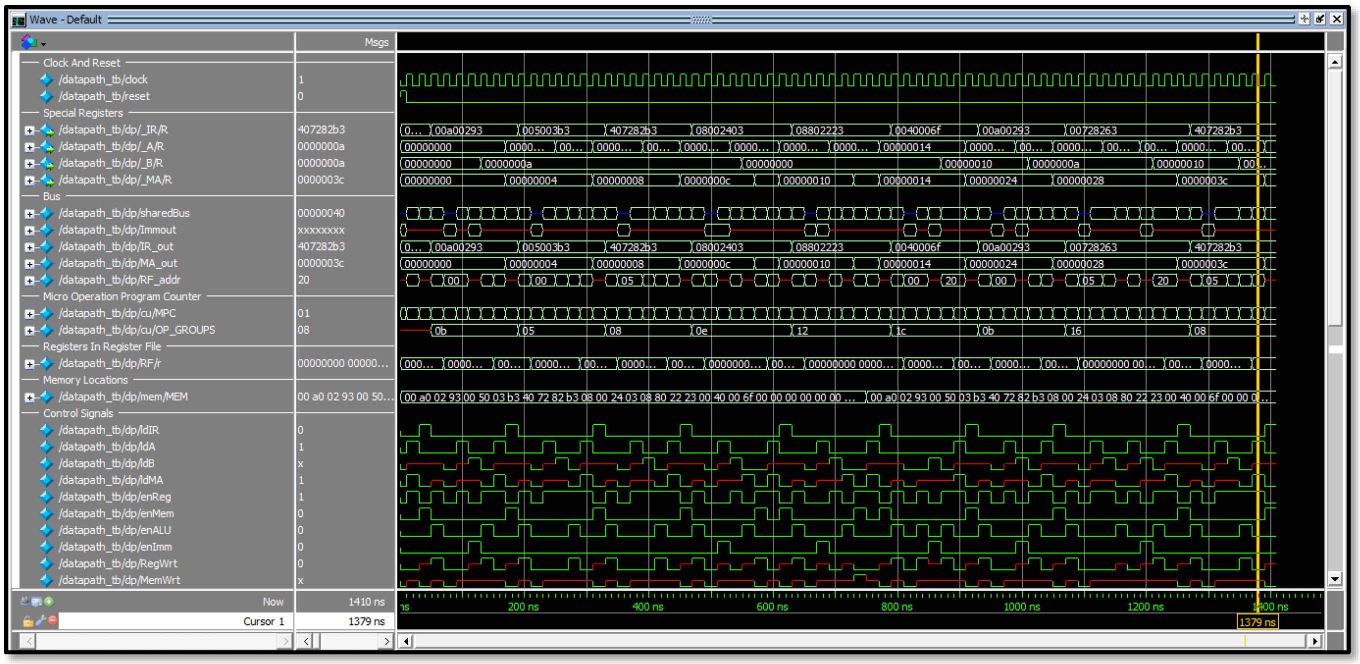
Testing DataPath Again After implementing control unit:

testing same benchmark we tested earlier :
Machine codes found in "Memory_init .txt" also

```
Word0:      00 A0 02 93   //ADDI X5,X0,10
Word1:      00 50 03 B3   //ADD X7,X5,X0
Word2:      40 72 82 B3   //SUB X5,X5,X7
Word3:      08 00 24 03   //LW X8,128(X0)
Word4:      08 80 22 23   // SW X8, 132(X0)
Word5:      00 40 00 6F   // J 4
Word6:      00 00 00 00
Word7:      00 00 00 00
Word8:      00 00 00 00
Word9:      00 A0 02 93   // ADDI X5,X0,10
Word10  :   00 72 82 63   // BEQ x5,x7,5
Word11:     01 40 02 93   //ADDI X5,X0,20
Word12:     00 00 00 00
Word13:     00 00 00 00
Word14:     00 00 00 00
Word15:     40 72 82 B3   //SUB X5,X5,X7
. . . . . . . . .
Word32:     11 00 11 FF //Data
```

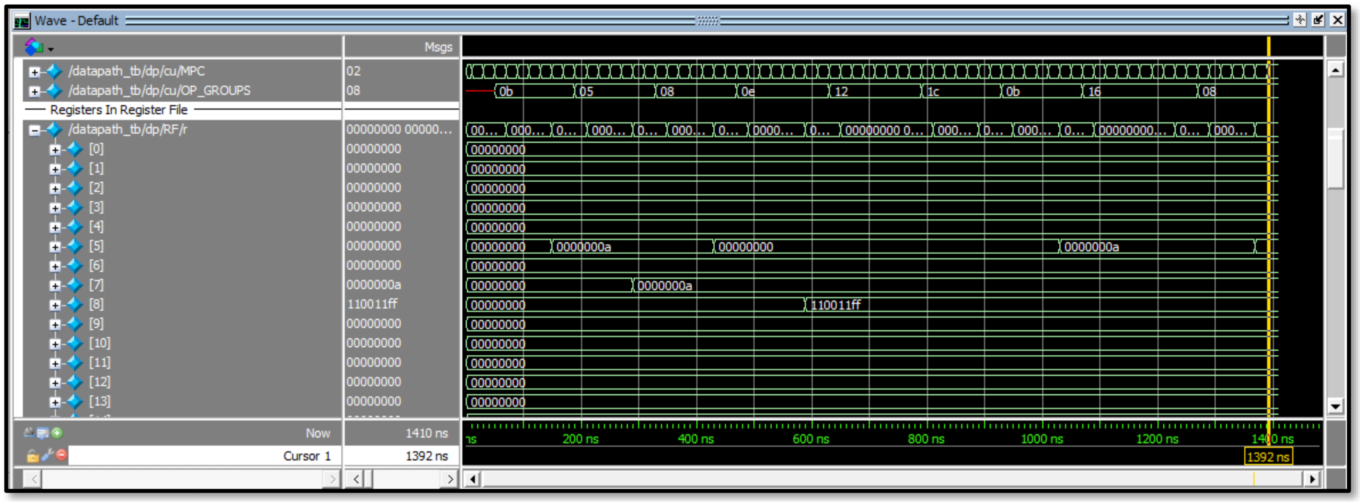
General view of the waveforms:

Note : you can find .do file for the waveform signal (wave.do)



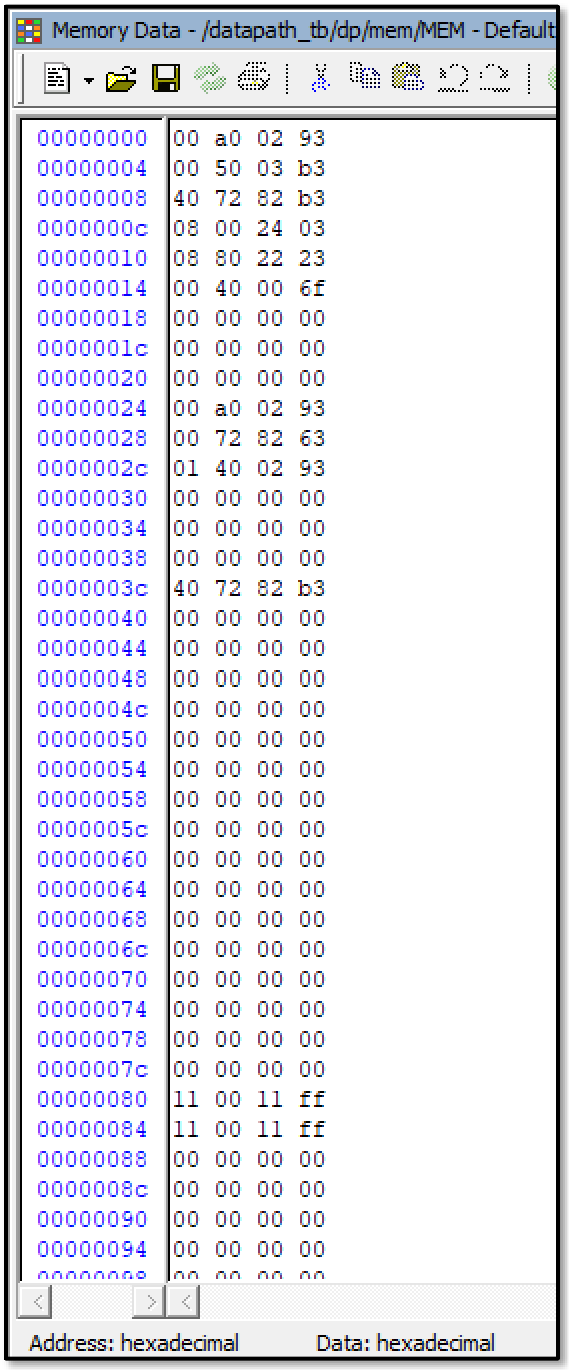
```
Wave - Default
                                        Msgs
— Clock And Reset —
  /datapath_tb/clock                    1
  /datapath_tb/reset                    0
— Special Registers —
+ /datapath_tb/dp/_IR/R                 407282b3
+ /datapath_tb/dp/_A/R                  0000000a
+ /datapath_tb/dp/_B/R                  0000000a
+ /datapath_tb/dp/_MA/R                 0000003c
— Bus —
+ /datapath_tb/dp/sharedBus             00000040
+ /datapath_tb/dp/Immout                xxxxxxxx
+ /datapath_tb/dp/IR_out                407282b3
+ /datapath_tb/dp/MA_out                0000003c
+ /datapath_tb/dp/RF_addr               20
— Micro Operation Program Counter —
+ /datapath_tb/dp/cu/MPC                01
+ /datapath_tb/dp/cu/OP_GROUPS          08
— Registers In Register File —
+ /datapath_tb/dp/RF/r                  00000000 00000...
— Memory Locations —
+ /datapath_tb/dp/mem/MEM               00 a0 02 93 00 50...
— Control Signals —
  /datapath_tb/dp/ldIR                  0
  /datapath_tb/dp/ldA                   1
  /datapath_tb/dp/ldB                   x
  /datapath_tb/dp/ldMA                  1
  /datapath_tb/dp/enReg                 1
  /datapath_tb/dp/enMem                 0
  /datapath_tb/dp/enALU                 0
  /datapath_tb/dp/enImm                 0
  /datapath_tb/dp/RegWrt                0
  /datapath_tb/dp/MemWrt                x
                              Now       1410 ns
                         Cursor 1       1379 ns
```

- Final view of registers values in register file



```
Wave - Default
                                        Msgs
+ /datapath_tb/dp/cu/MPC                02
+ /datapath_tb/dp/cu/OP_GROUPS          08
— Registers In Register File —
- /datapath_tb/dp/RF/r                  00000000 00000...
    + [0]                               00000000
    + [1]                               00000000
    + [2]                               00000000
    + [3]                               00000000
    + [4]                               00000000
    + [5]                               00000000
    + [6]                               00000000
    + [7]                               0000000a
    + [8]                               110011ff
    + [9]                               00000000
    + [10]                              00000000
    + [11]                              00000000
    + [12]                              00000000
    + [13]                              00000000
                              Now       1410 ns
                         Cursor 1       1392 ns
```

- Final view of Memory values after executing the program



```
00000000  00 a0 02 93
00000004  00 50 03 b3
00000008  40 72 82 b3
0000000c  08 00 24 03
00000010  08 80 22 23
00000014  00 40 00 6f
00000018  00 00 00 00
0000001c  00 00 00 00
00000020  00 00 00 00
00000024  00 a0 02 93
00000028  00 72 82 63
0000002c  01 40 02 93
00000030  00 00 00 00
00000034  00 00 00 00
00000038  00 00 00 00
0000003c  40 72 82 b3
00000040  00 00 00 00
00000044  00 00 00 00
00000048  00 00 00 00
0000004c  00 00 00 00
00000050  00 00 00 00
00000054  00 00 00 00
00000058  00 00 00 00
0000005c  00 00 00 00
00000060  00 00 00 00
00000064  00 00 00 00
00000068  00 00 00 00
0000006c  00 00 00 00
00000070  00 00 00 00
00000074  00 00 00 00
00000078  00 00 00 00
0000007c  00 00 00 00
00000080  11 00 11 ff
00000084  11 00 11 ff
00000088  00 00 00 00
0000008c  00 00 00 00
00000090  00 00 00 00
00000094  00 00 00 00
```

Address: hexadecimal   Data: hexadecimal