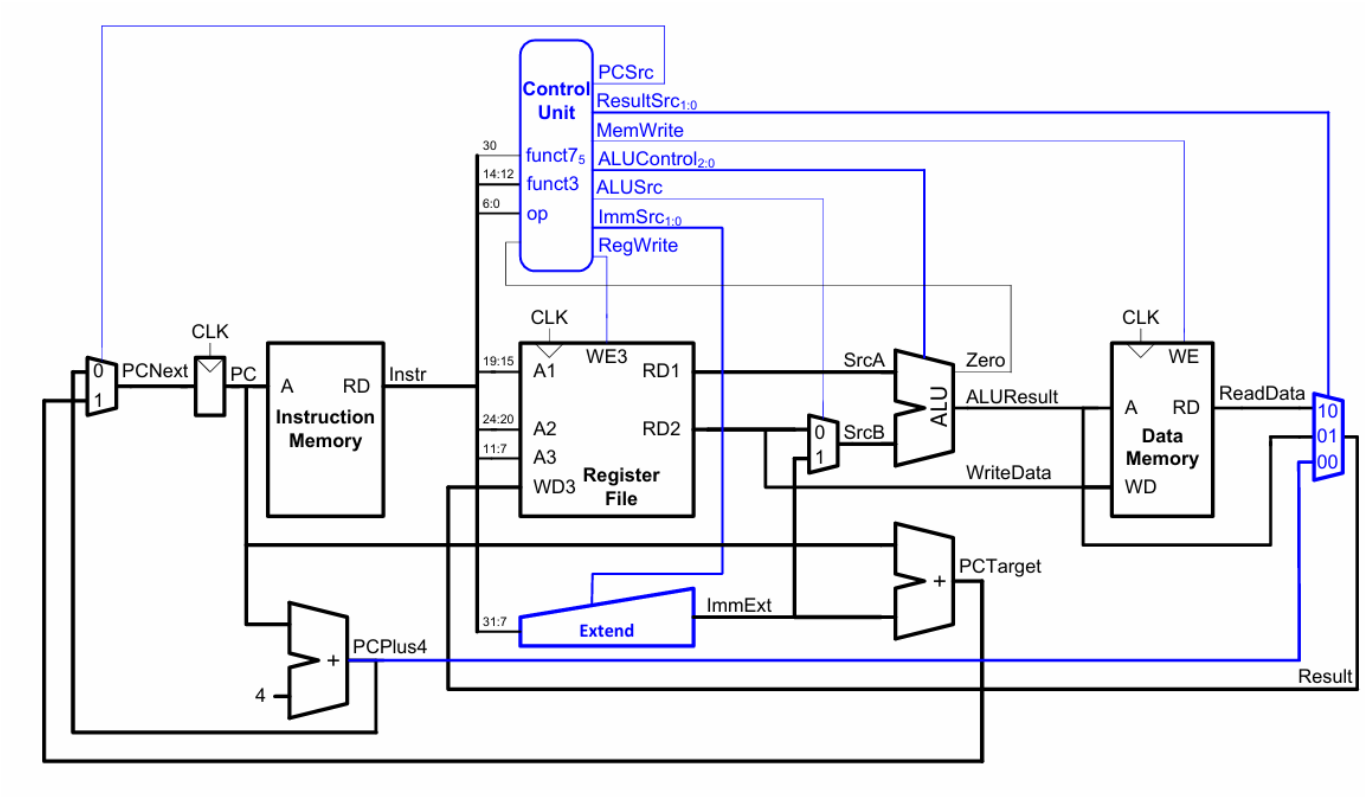
RISC-V®

Design RISC-V Process



- We divide our microarchitectures into two interacting parts: the data path and the control unit.

- The datapath operates on words of data. It contains structures such as memories registers, ALUs, and multiplexers.

- The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction.

- The control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

- This RISC-V processor supports the following instruction types:

R-type → {add, sub, sll, xor, srl, slt, or, and}

I-type → {lw, addi, slli, xori, srli, slti, ori, andi}

S-type → {sw}

SB-type → {beq, bne}

UJ-type → {jal}

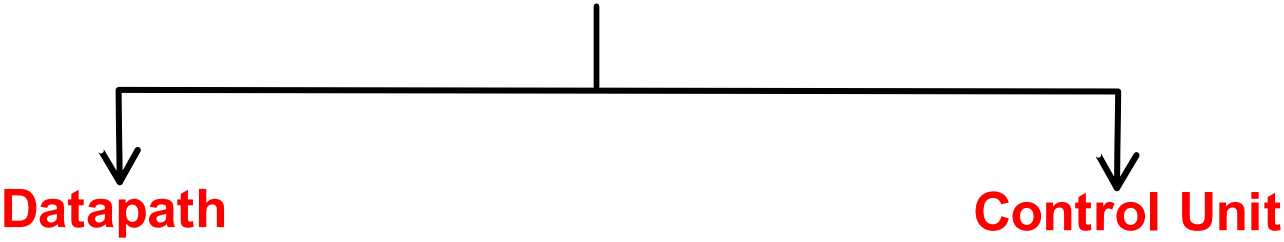
Desgin Note:-

- A good way to design a complex system is to start with hardware containing the state elements.

- These elements include the memories and the architectural state (the program counter and registers).

- Then, add blocks of combinational logic between the state elements to compute the new state based on the current state.

Microarchitecture Division



Datapath

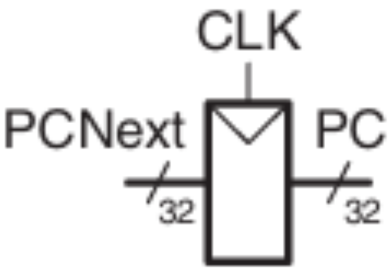
Control Unit

Datapath:- Handles data flow through components to execute instructions.

Control Unit:- Generates control signals to guide the datapath.

Datapath Blocks

(1) Program Counter (PC):-



Input Data:

clk --> Clock
rst_n --> asynchronous negative reset
in_pc --> next instruction address

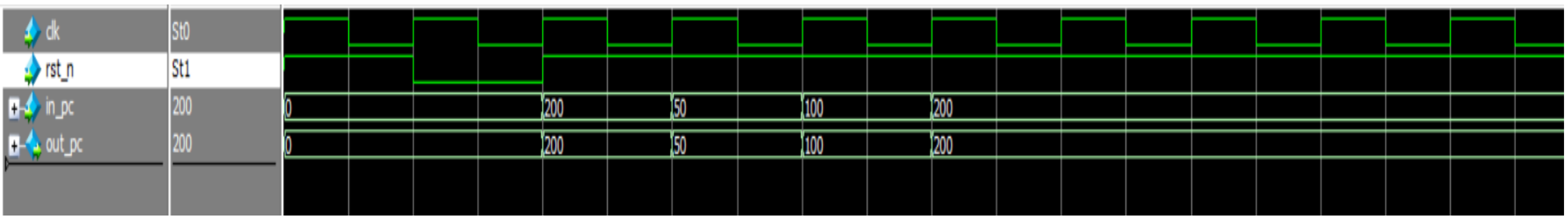
Output Data:

out_pc --> Current instruction address

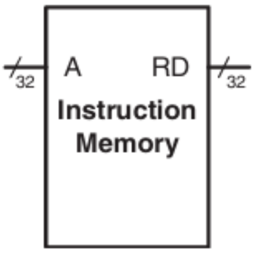
Function:

Holds address of the current instruction.

Verification:



(2) Instruction Memory:-



Input Data:

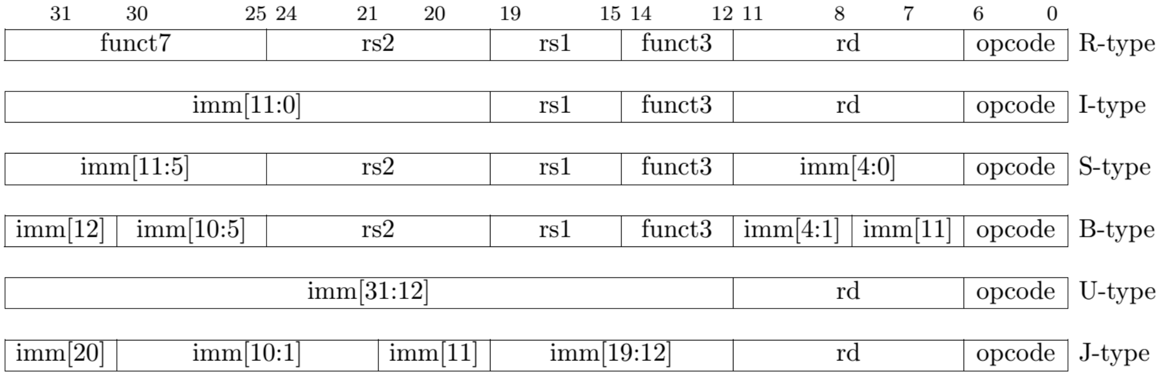
instr_address --> instruction address coming from Program counter.

Output Data:

instruction --> instruction covered {Address Register Files of sourec & Destination & Opcode & immediate}.

Function:

Fetch instruction from memory using PC.



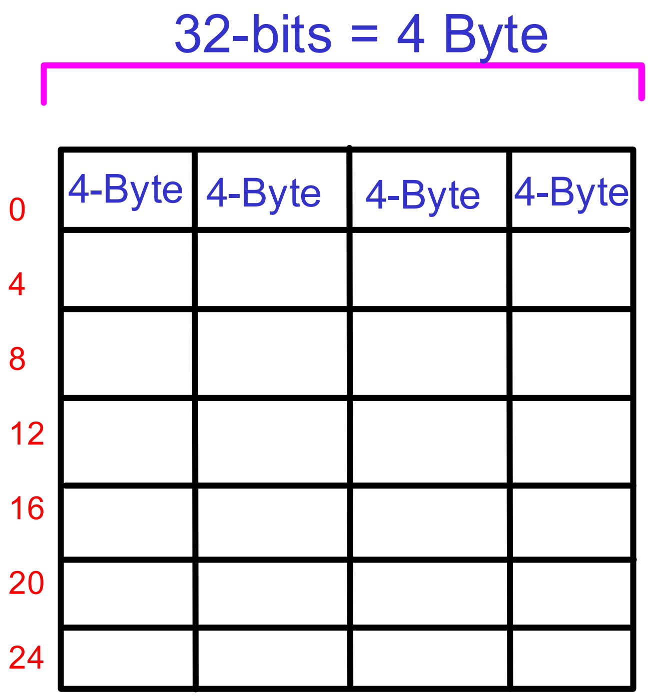
31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

- Why do shift by 2 in instruction address?

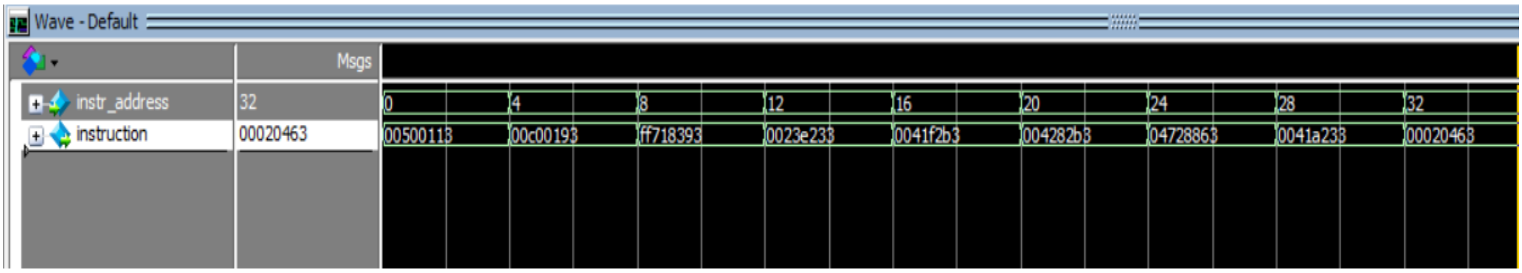
Because:

Each instruction = 4 bytes

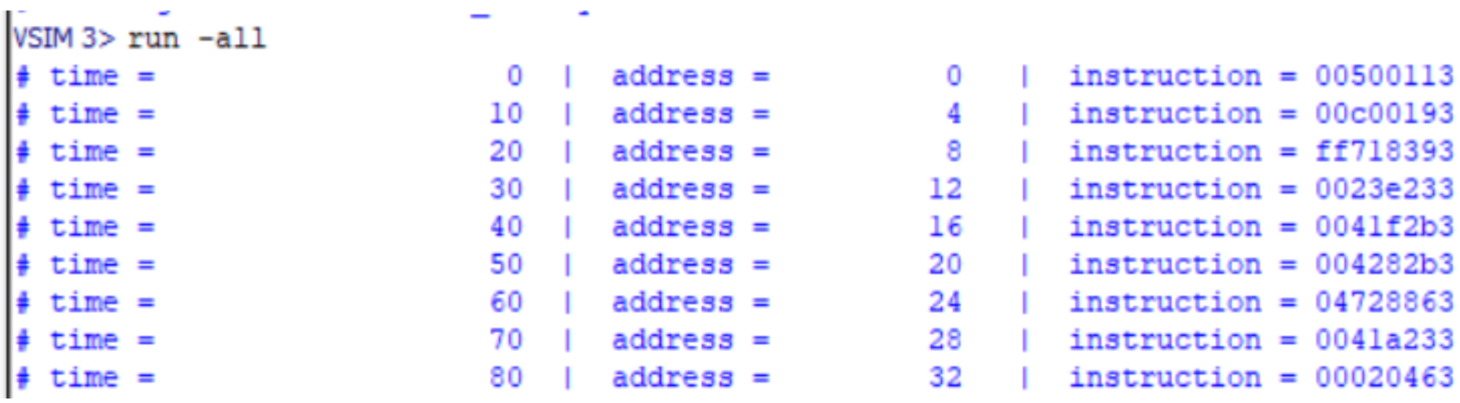
Shift by 2 bits = dividing the address by 4



Verification:

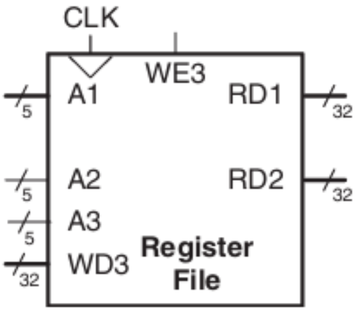


```
instr_address   32        0        4        8        12       16       20       24       28       32
instruction     00020463  00500113 00c00193 ff718393 0023e233 0041f2b3 004282b3 04728863 0041a233 00020463
```



```
VSIM 3> run -all
# time =                   0  |  address =           0   |  instruction = 00500113
# time =                  10  |  address =           4   |  instruction = 00c00193
# time =                  20  |  address =           8   |  instruction = ff718393
# time =                  30  |  address =          12   |  instruction = 0023e233
# time =                  40  |  address =          16   |  instruction = 0041f2b3
# time =                  50  |  address =          20   |  instruction = 004282b3
# time =                  60  |  address =          24   |  instruction = 04728863
# time =                  70  |  address =          28   |  instruction = 0041a233
# time =                  80  |  address =          32   |  instruction = 00020463
```

(3) Register File:-



Input Data:

clk --> Clock
A1,A2 (5 bits) --> read register address
A3 (5 bits) --> write register address
WD3 (32 bits) --> write data input
WE --> write enable

Output Data:

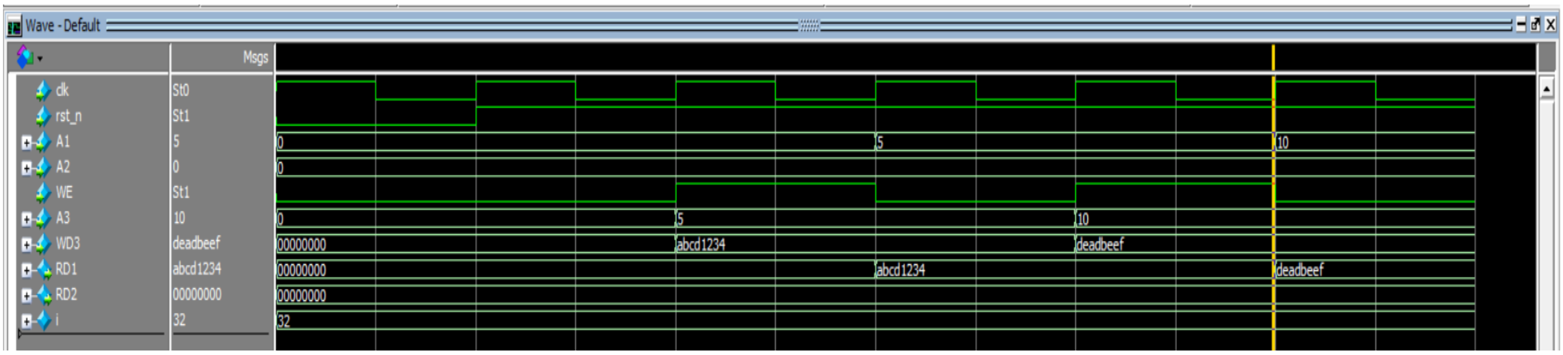
RD1 (32 bits) --> read data outputs
RD2 (32 bits) --> read data outputs

Function:

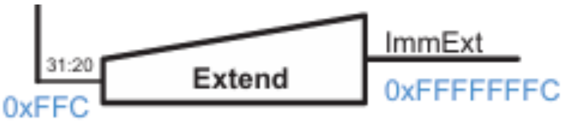
Reads/writes general-purpose registers.

Register	Description
x0	Hard-wired to 0
x1	Return address
x2	Stack pointer
x3	Global Pointer
x4	Thread Pointer
x5 to x7	Temporary Registers
x8	Frame Pointer/Saved Register
x9	Saved Register
x10 to x11	Function Arguments/Return Value
x12 to x17	Function arguments
x18 to x27	Saved Registers
x28 to x31	Temporary registers

Verification:



(4) Immediate Generator:-



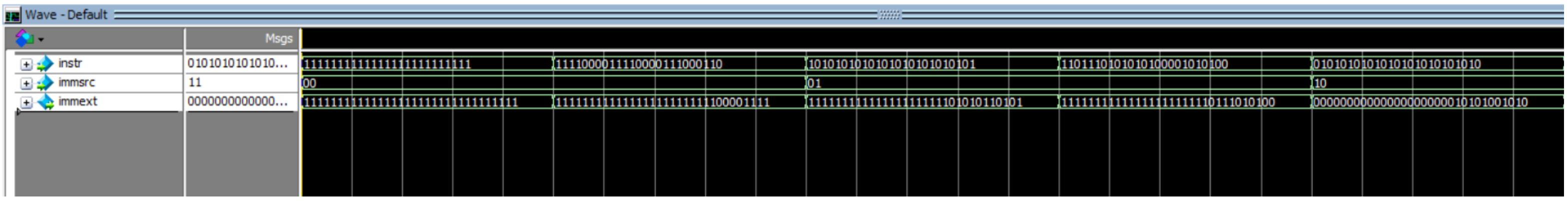
Input Data:

instruction (32-bit)
immediate source selector

Output Data: extended immediate output

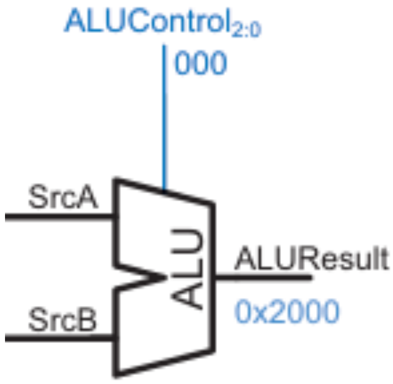
Function: Extracts and sign-extends immediate values from instruction.

Verification:



```
instr    0101010101010...  11111111111111111111111111  1111000011110000111000110  1010101010101010101010101  1101110101010100001010100  0101010101010101010101010
immsrc   11                00                                                     01                                                    10
immext   0000000000000...  11111111111111111111111111111111  1111111111111111111111100001111  11111111111111111111101010110101  11111111111111111111101110101100  00000000000000000000001010100101 0
```

(5) ALU:-



Input Data:

Input_1 ---> (from register)

Input_2 ---> (register or immediate)

alu_control ---> (from ALU Control unit)

Output Data:

alu_result ---> (32-bit)

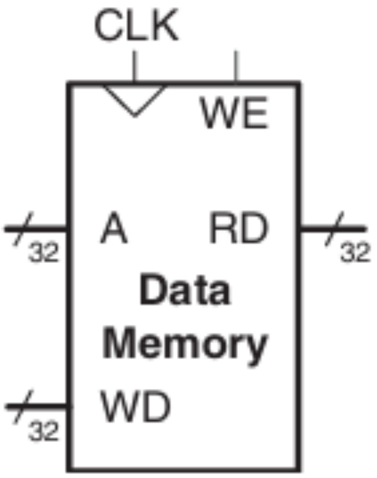
zero flag ---> (for branching)

Function:

Executes arithmetic/logical operations.

Verification:

(6) Data Memory:-



Input Data:

A -----> address (from ALU)

WD ---> write_data (from reg file)

WE --->MemRead, MemWrite (control signals)

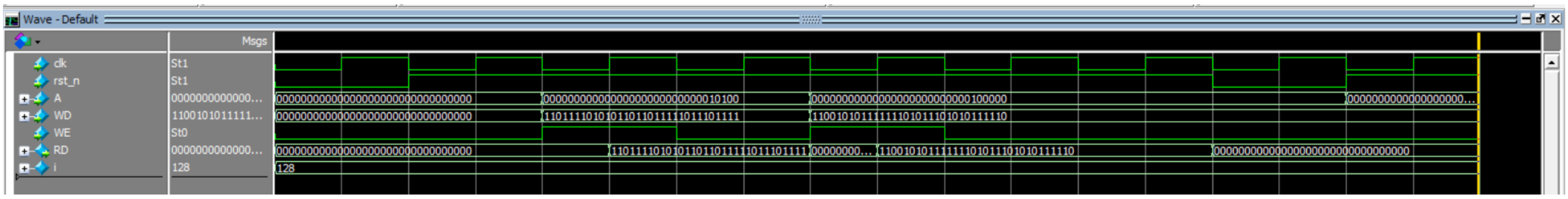
Output Data:

RD ----> read_data

Function:

Performs load/store.

Verification:



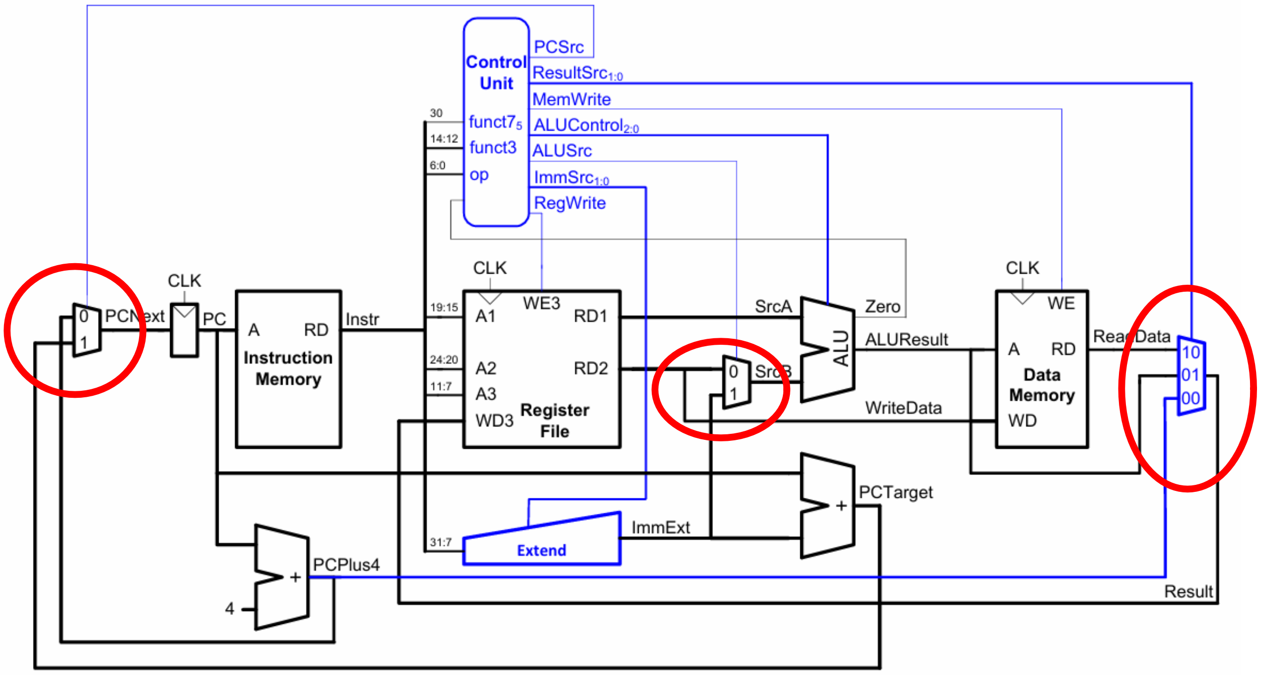
```
clk     St1
rst_n   St1
A       0000000000000...   00000000000000000000000000000000   00000000000000000000000000010100   00000000000000000000000000100000   0000000000000000000...
WD      1100101011111...   00000000000000000000000000000000   11011110101011011011111011101111   11001010111111101011101010111110
WE      St0
RD      0000000000000...   00000000000000000000000000000000   11011110101011011011111011101111   00000000...   11001010111111101011101010111110   00000000000000000000000000000000
i       128                128
```

(7) MUXes:-

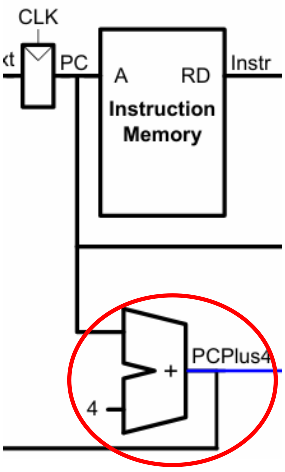
1- ALUSrc MUX.

2- MemToReg MUX

3- PCSrc MUX



(8) PC Adder:-

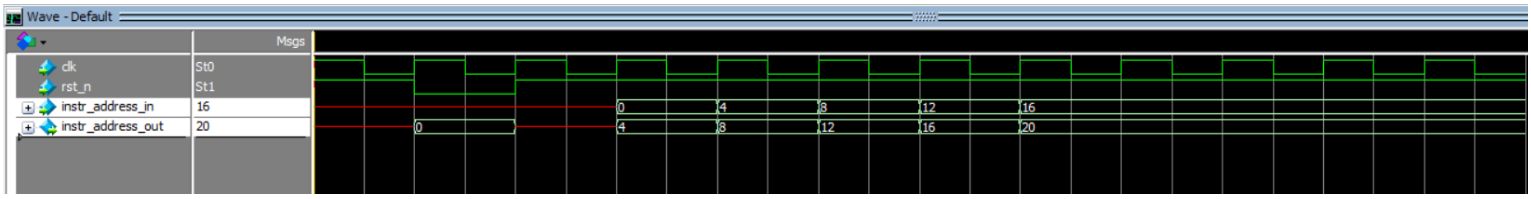


Input Data: PC (32-bit): Current value of the Program Counter.

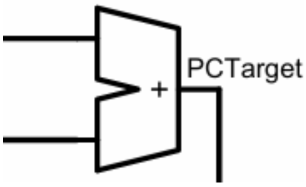
Output Data: PC_plus_4 (32-bit): Result of PC + 4

Function: Adds 4 to the current PC.

Verification:



(9) Branch Target Adder:-



Input Data:

PC_plus_4 (32-bit) ---> The address of the next instruction (sequential path).

imm16 (16-bit) ---> Immediate field from the instruction (offset for branch).

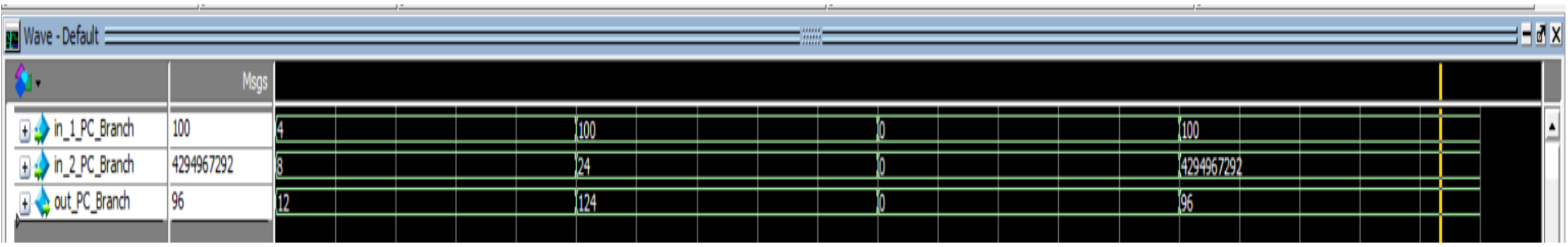
Output Data:

branch_target (32-bit) ---> The computed address to jump to if the branch is taken.

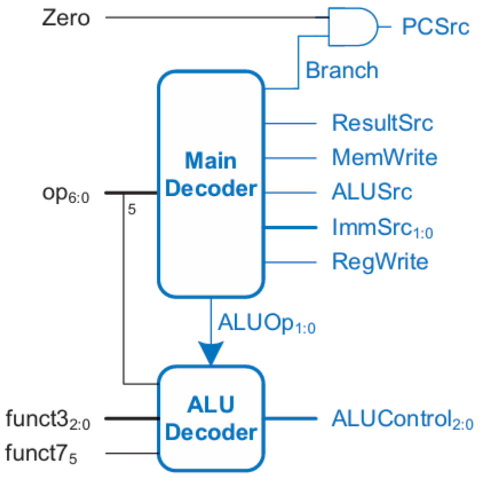
Function:

Computes the target address for a branch instruction by adding the sign-extended, shifted immediate to PC + 4.

Verification:



Control Unit Blocks



(1) Main Control Unit:-

Input Data: opcode ---> (from instruction)

Output Data:

Branch ---> Enables branching instructions (e.g., beq, bne)

ResultSrc ---> Selects between ALU result or memory output for write-back to register

MemWrite ---> Enables memory write (e.g., sw)

ALUSrc ---> Selects between register value or immediate as ALU second operand

Jump ---> Enables jump instructions (e.g., jal, jalr)

ImmSrc ---> Selects the immediate format (I, S, B, U, J) based on instruction type

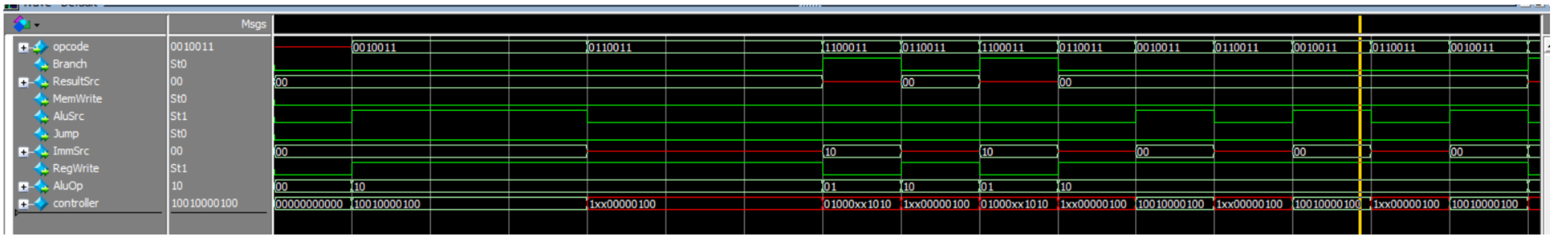
RegWrite ---> Enables register file write-back

ALUOp ---> Determines the ALU operation type (used by ALU control)

Function:

Decodes opcode and generates control signals.

Verification:



	Msgs
opcode	0010011
Branch	St0
ResultSrc	00
MemWrite	St0
AluSrc	St1
Jump	St0
ImmSrc	00
RegWrite	St1
AluOp	10
controller	10010000100

(2) ALU Control:-

Input Data:

ALUOp (from main control)

funct3, funct7 (from instruction)

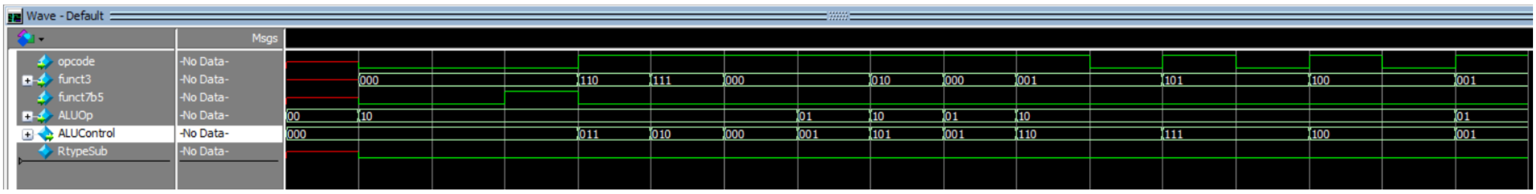
Output Data:

alu_control signal

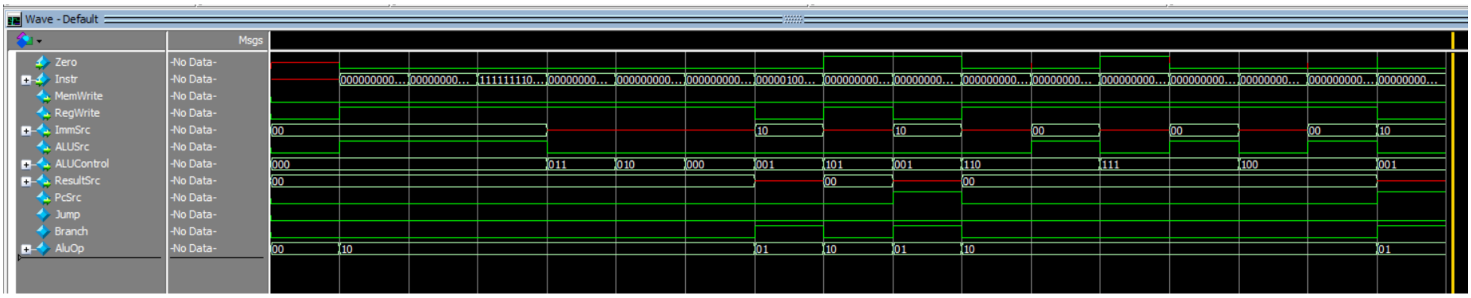
Function:

Determines ALU operation using funct3, funct7, ALUOp.

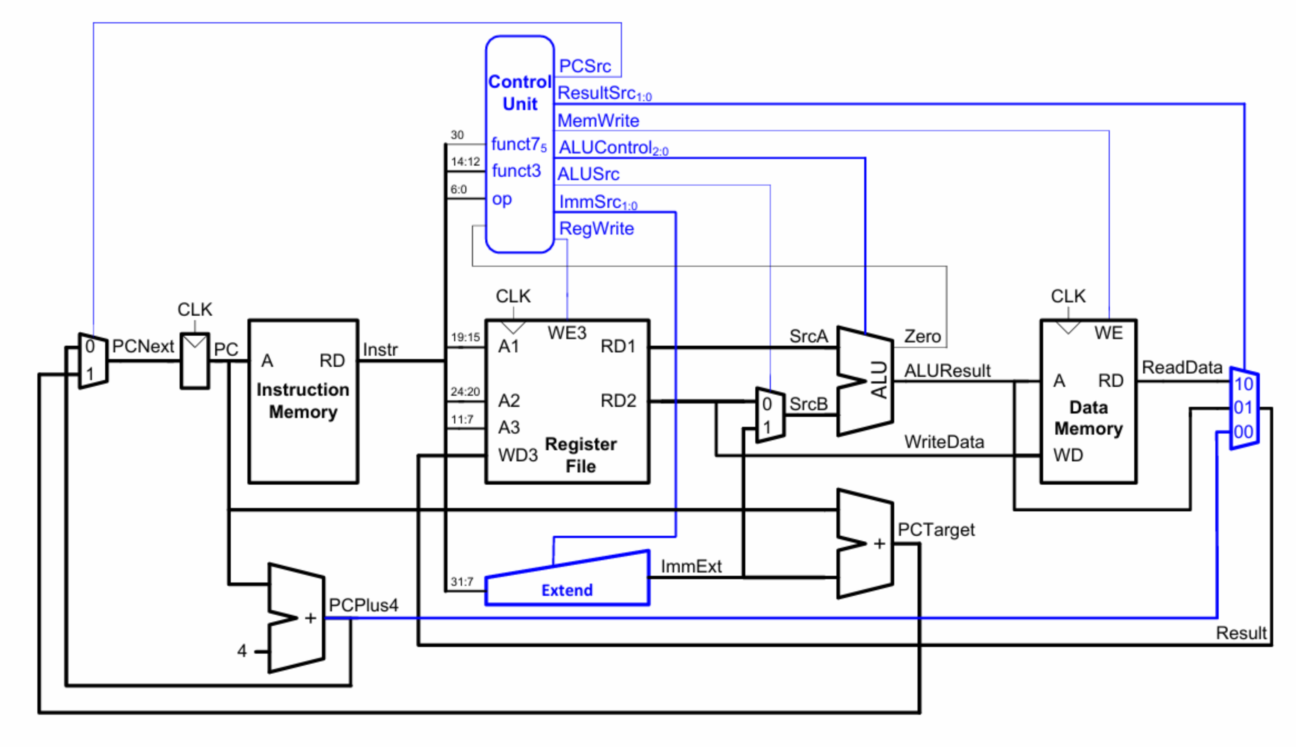
Verification:



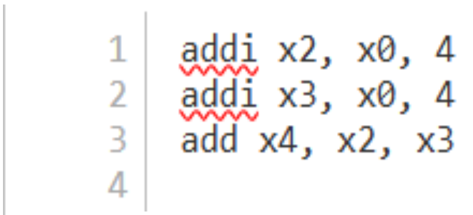
Control unit Verification :



Testing Processor



(1) add instruction:



```
addi x2, x0, 4
addi x3, x0, 4
add x4, x2, x3
```

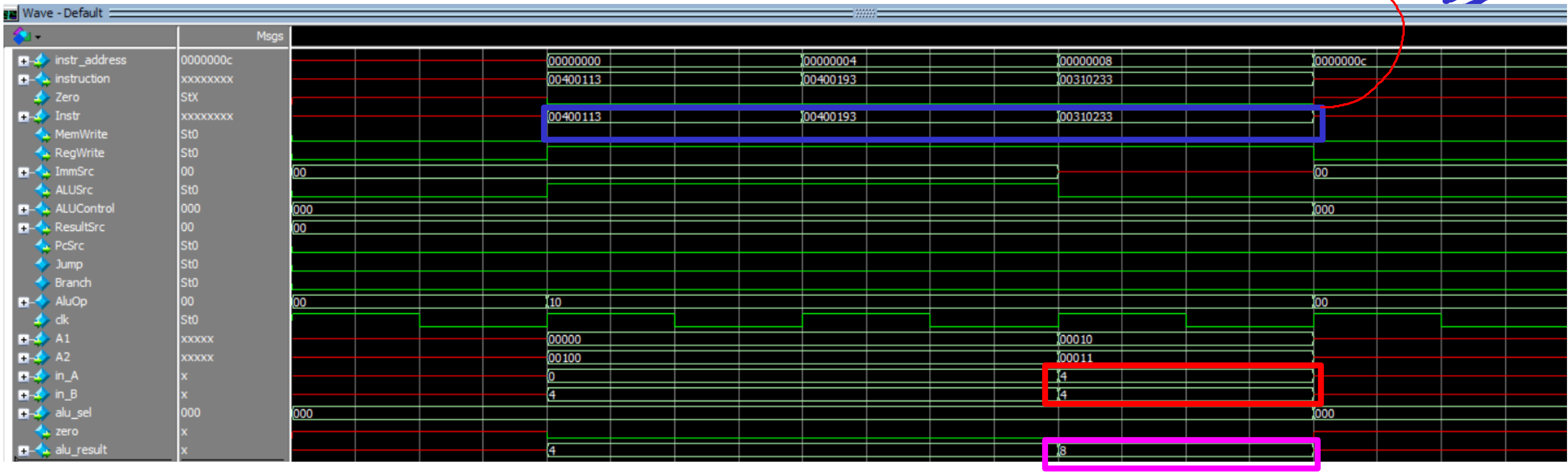
00400113
00400193
00310233

add

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type

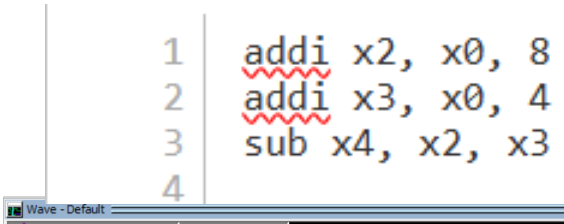
addi

instructions



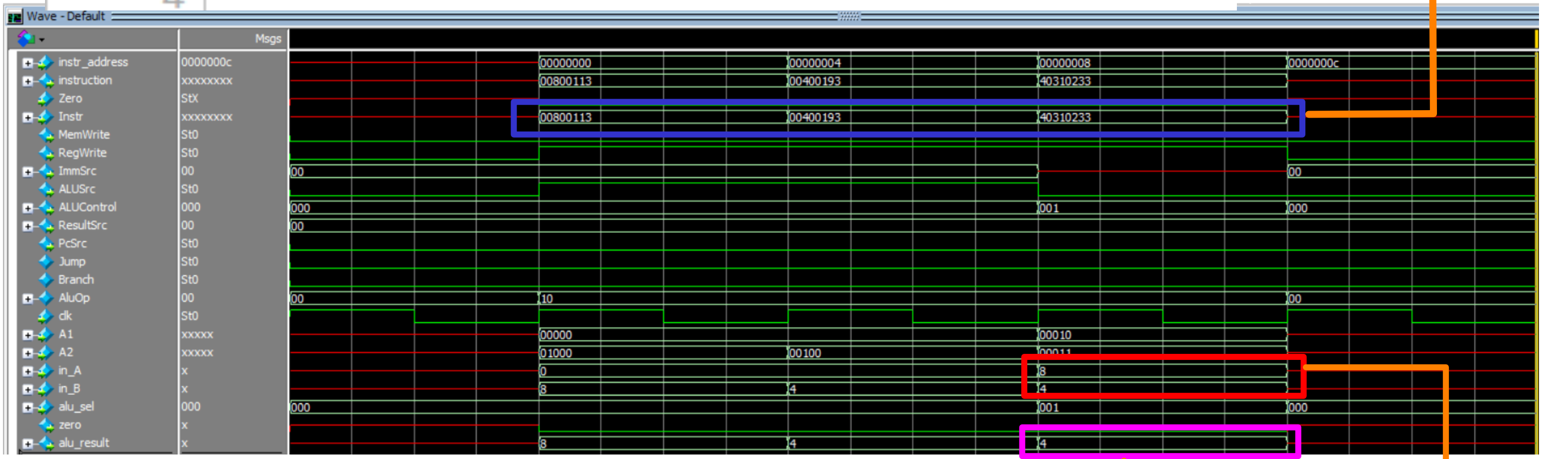
Signal	Msgs
instr_address	0000000c
instruction	xxxxxxxx
Zero	StX
Instr	xxxxxxxx
MemWrite	St0
RegWrite	St0
ImmSrc	00
ALUSrc	St0
ALUControl	000
ResultSrc	00
PcSrc	St0
Jump	St0
Branch	St0
AluOp	00
clk	St0
A1	xxxxx
A2	xxxxx
in_A	x
in_B	x
alu_sel	000
zero	x
alu_result	x

(2) sub instruction:



```
addi x2, x0, 8
addi x3, x0, 4
sub x4, x2, x3
```

00800113
00400193
40310233



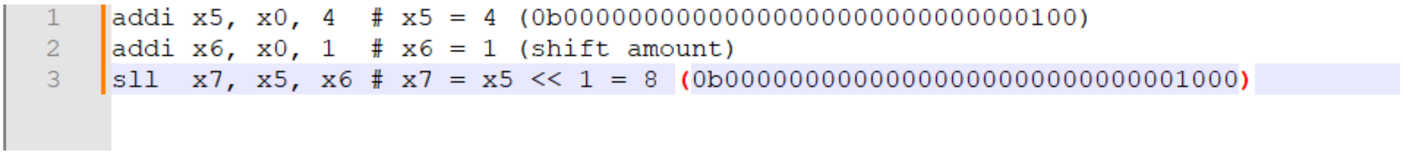
instructions

two inputs ALU

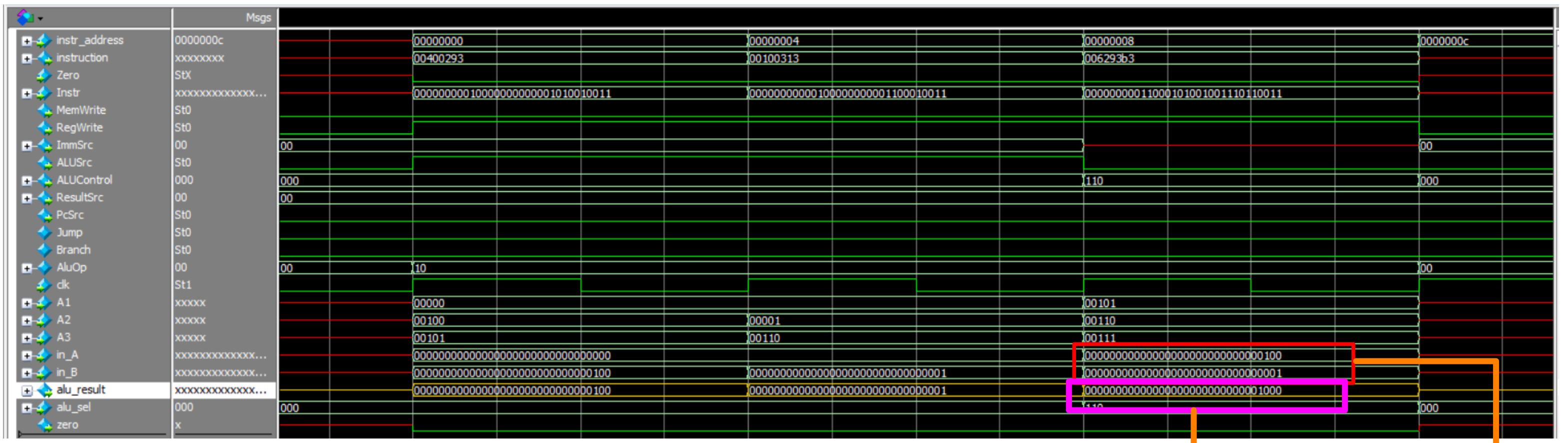
Result

(3) sll instruction:

sll = "Shift Left Logical"



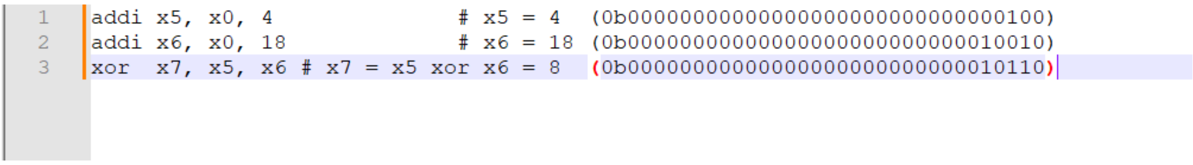
```
addi x5, x0, 4  # x5 = 4 (0b00000000000000000000000000000100)
addi x6, x0, 1  # x6 = 1 (shift amount)
sll  x7, x5, x6 # x7 = x5 << 1 = 8 (0b00000000000000000000000000001000)
```



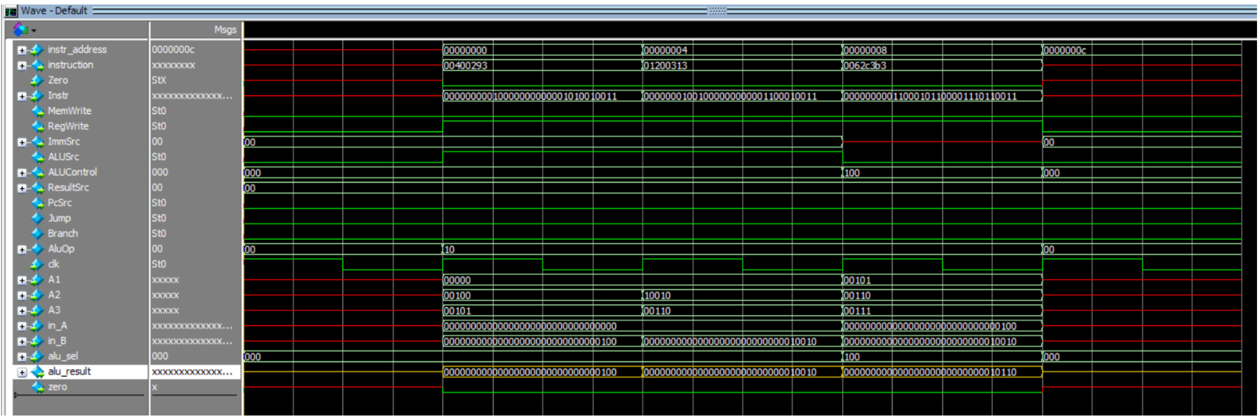
Result

two inputs ALU

(4) xor instruction:



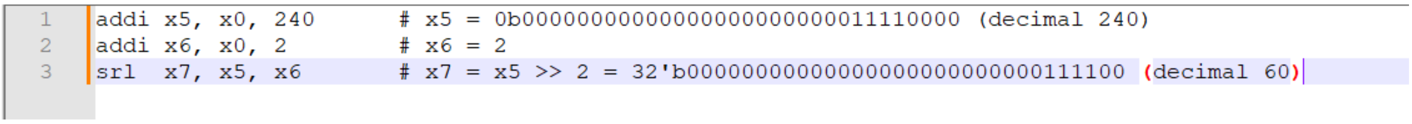
```
addi x5, x0, 4              # x5 = 4   (0b00000000000000000000000000000100)
addi x6, x0, 18             # x6 = 18  (0b00000000000000000000000000010010)
xor  x7, x5, x6 # x7 = x5 xor x6 = 8   (0b00000000000000000000000000010110)
```



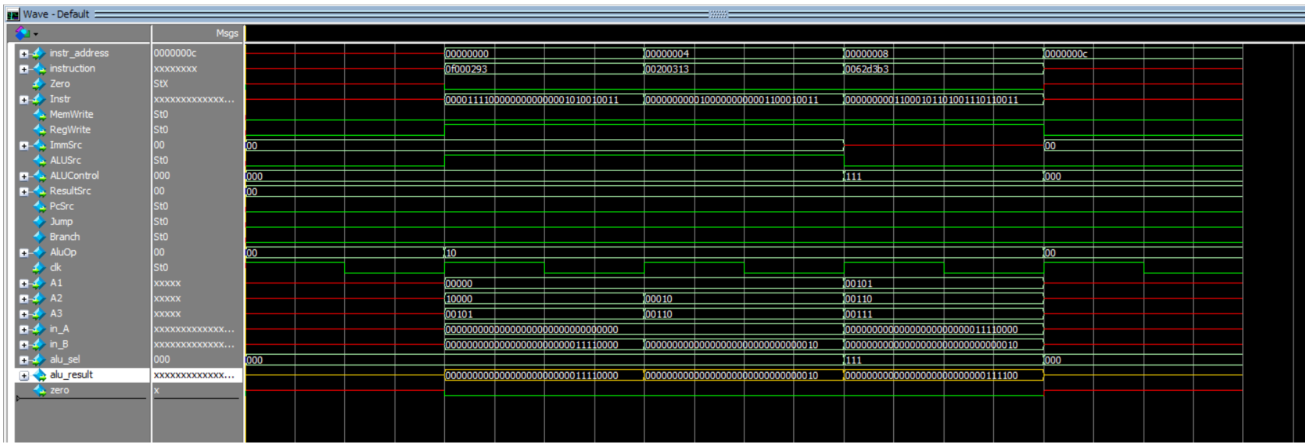
```
instr_address  0000000c          00000000 | 00000004 | 00000008 | 0000000c
instruction    xxxxxxxx          00400293 | 01200313 | 0062c3b3
Zero           StX
Instr          xxxxxxxxxxxxx...  00000000010000000000001010010011 | 00000001001000000000001100010011 | 00000000011000101100001110110011
MemWrite       St0
RegWrite       St0
ImmSrc         00                00 | 00
ALUSrc         St0
ALUControl     000               000 | 100 | 000
ResultSrc      00                00
PcSrc          St0
Jump           St0
Branch         St0
AluOp          00                00 | 10 | 00
clk            St0
A1             xxxxx             00000 | 00101
A2             xxxxx             00100 | 10010 | 00110
A3             xxxxx             00101 | 00110 | 00111
in_A           xxxxxxxxxxxxx...  00000000000000000000000000000000 | 00000000000000000000000000000100
in_B           xxxxxxxxxxxxx...  00000000000000000000000000000100 | 00000000000000000000000000010010 | 00000000000000000000000000010010
alu_sel        000               000 | 100 | 000
alu_result     xxxxxxxxxxxxx...  00000000000000000000000000000100 | 00000000000000000000000000010010 | 00000000000000000000000000010110
zero           x
```

(5) srl instruction:

srl = Shift Right Logical

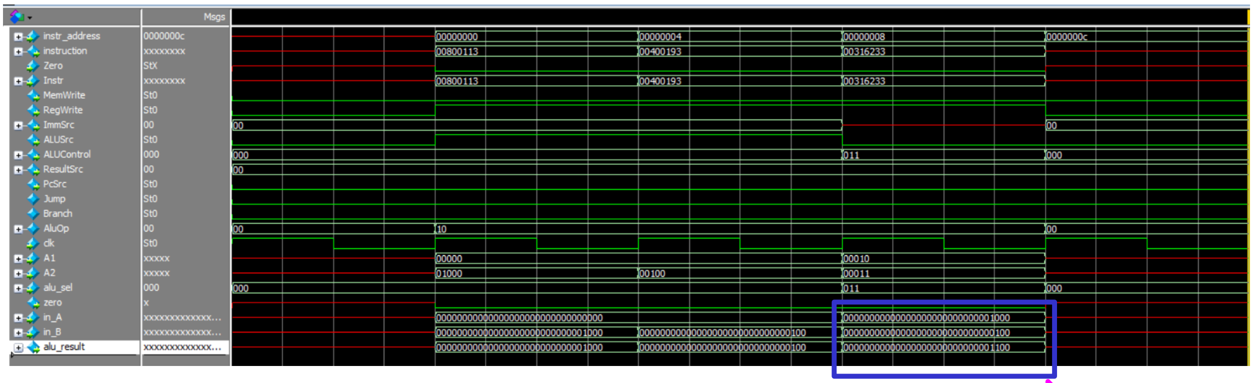


```
addi x5, x0, 240      # x5 = 0b00000000000000000000000011110000 (decimal 240)
addi x6, x0, 2        # x6 = 2
srl  x7, x5, x6       # x7 = x5 >> 2 = 32'b00000000000000000000000000111100 (decimal 60)
```



```
instr_address  0000000c          00000000 | 00000004 | 00000008 | 0000000c
instruction    xxxxxxxx          0f000293 | 00200313 | 0062d3b3
Zero           StX
Instr          xxxxxxxxxxxxx...  00001111000000000000001010010011 | 00000000001000000000001100010011 | 00000000011000101101001110110011
MemWrite       St0
RegWrite       St0
ImmSrc         00                00 | 00
ALUSrc         St0
ALUControl     000               000 | 111 | 000
ResultSrc      00                00
PcSrc          St0
Jump           St0
Branch         St0
AluOp          00                00 | 10 | 00
clk            St0
A1             xxxxx             00000 | 00101
A2             xxxxx             10000 | 00010 | 00110
A3             xxxxx             00101 | 00110 | 00111
in_A           xxxxxxxxxxxxx...  00000000000000000000000000000000 | 00000000000000000000000011110000
in_B           xxxxxxxxxxxxx...  00000000000000000000000011110000 | 00000000000000000000000000000010 | 00000000000000000000000000000010
alu_sel        000               000 | 111 | 000
alu_result     xxxxxxxxxxxxx...  00000000000000000000000011110000 | 00000000000000000000000000000010 | 00000000000000000000000000111100
zero           x
```

(6) or instruction:



```
instr_address  0000000c          00000000 | 00000004 | 00000008 | 0000000c
instruction    xxxxxxxx          00800113 | 00400193 | 00316233
Zero           StX
Instr          xxxxxxxx          00800113 | 00400193 | 00316233
MemWrite       St0
RegWrite       St0
ImmSrc         00                00 | 00
ALUSrc         St0
ALUControl     000               000 | 011 | 000
ResultSrc      00                00
PcSrc          St0
Jump           St0
Branch         St0
AluOp          00                00 | 10 | 00
clk            St0
A1             xxxxx             00000 | 00010
A2             xxxxx             01000 | 00100 | 00011
alu_sel        000               000 | 011 | 000
zero           x
in_A           xxxxxxxxxxxxx...  00000000000000000000000000000000 | 00000000000000000000000000001000
in_B           xxxxxxxxxxxxx...  00000000000000000000000000001000 | 00000000000000000000000000000100 | 00000000000000000000000000000100
alu_result     xxxxxxxxxxxxx...  00000000000000000000000000001000 | 00000000000000000000000000000100 | 00000000000000000000000000001100
```

```
    00000000000000000000000000001000
or  00000000000000000000000000000100
    --------------------------------
    00000000000000000000000000001100
```

OR instruction