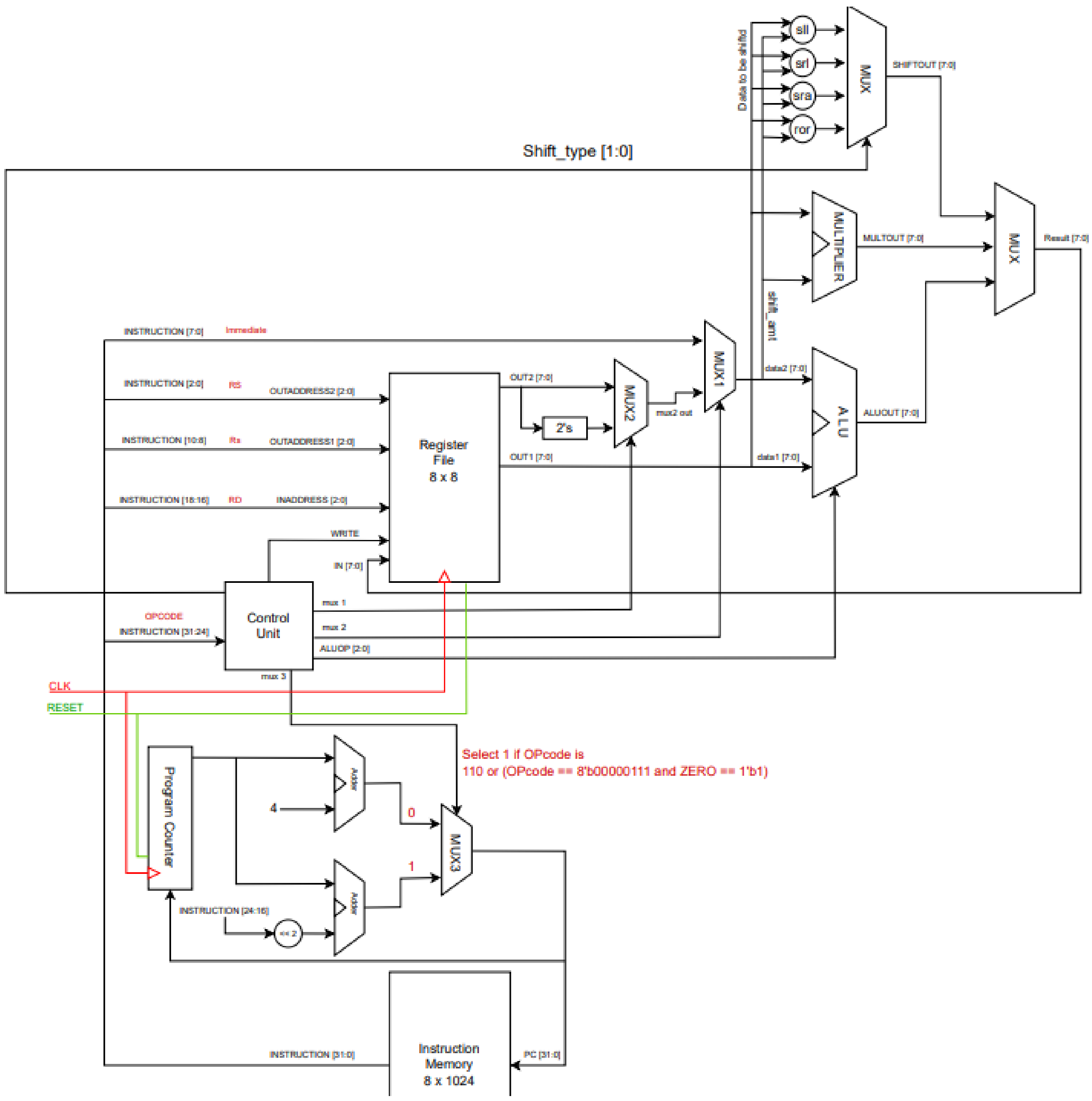
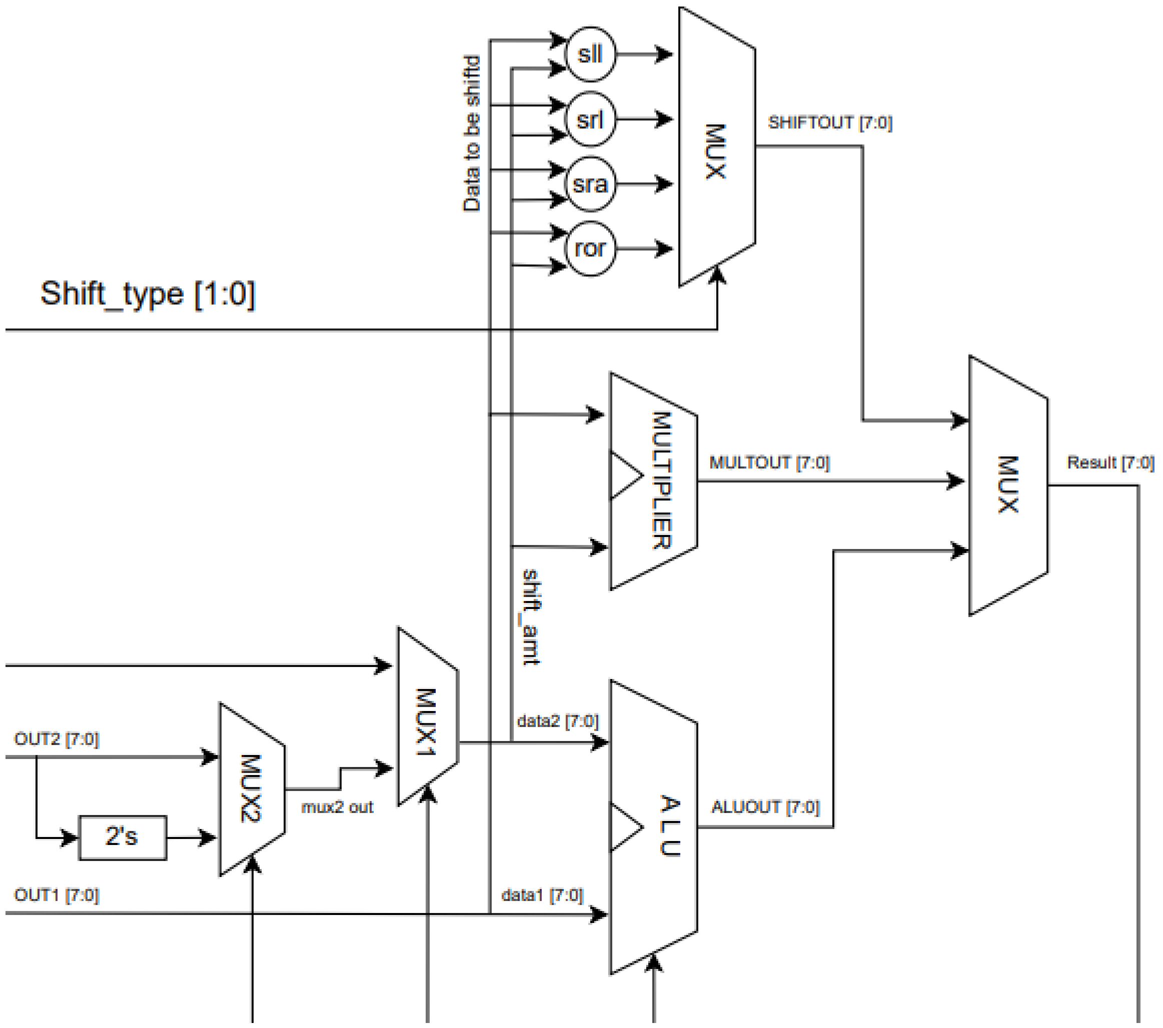
The Big Picture of the Simple 8-Bit Computer

Here is the complete block diagram of a simple 8-bit computer which performs the **add, sub, and, or, beq, j, mult, sll, srl, sra, ror, bne** instructions. So this can be used to perform different algorithms to find factorial of a number, nth fibonacci number, et cetra. So lets dive into the implementation of mult, sll, srl, sra, ror, bne instructions…



Implementation of mult, ssl, srl, sra, ror, bne instructions…

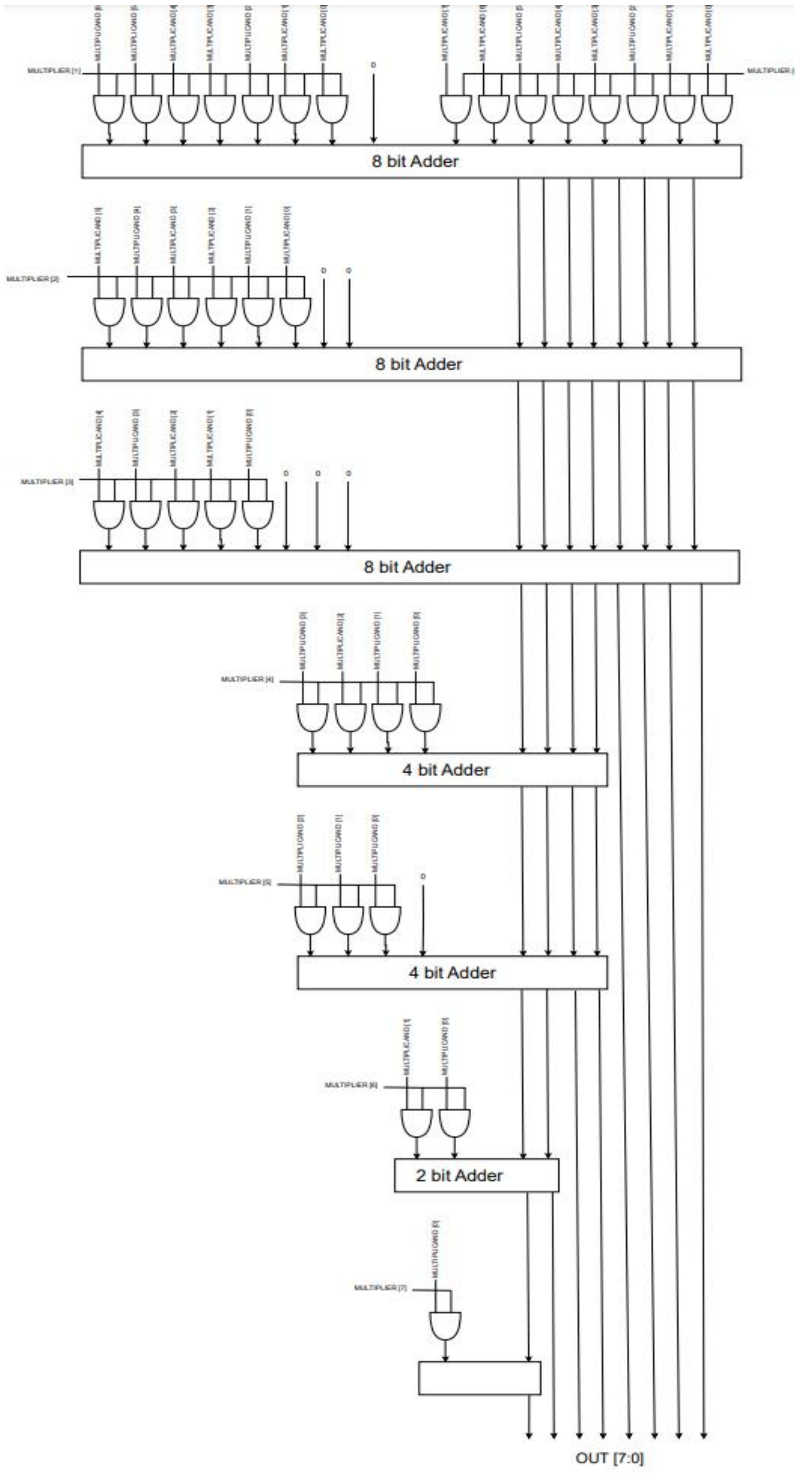
Additianal two units have been introduced to perform shift and multiplication operations. Shift module has 4 sub units and a multiplexer to select shift_type. And there is another multipexer to select the result from shift, ALU or Multiplier based on the instruction. Selection signals come from the control unit.



Multiplier

A delay of #2 is added for the multiply operation.

Circuit Diagram

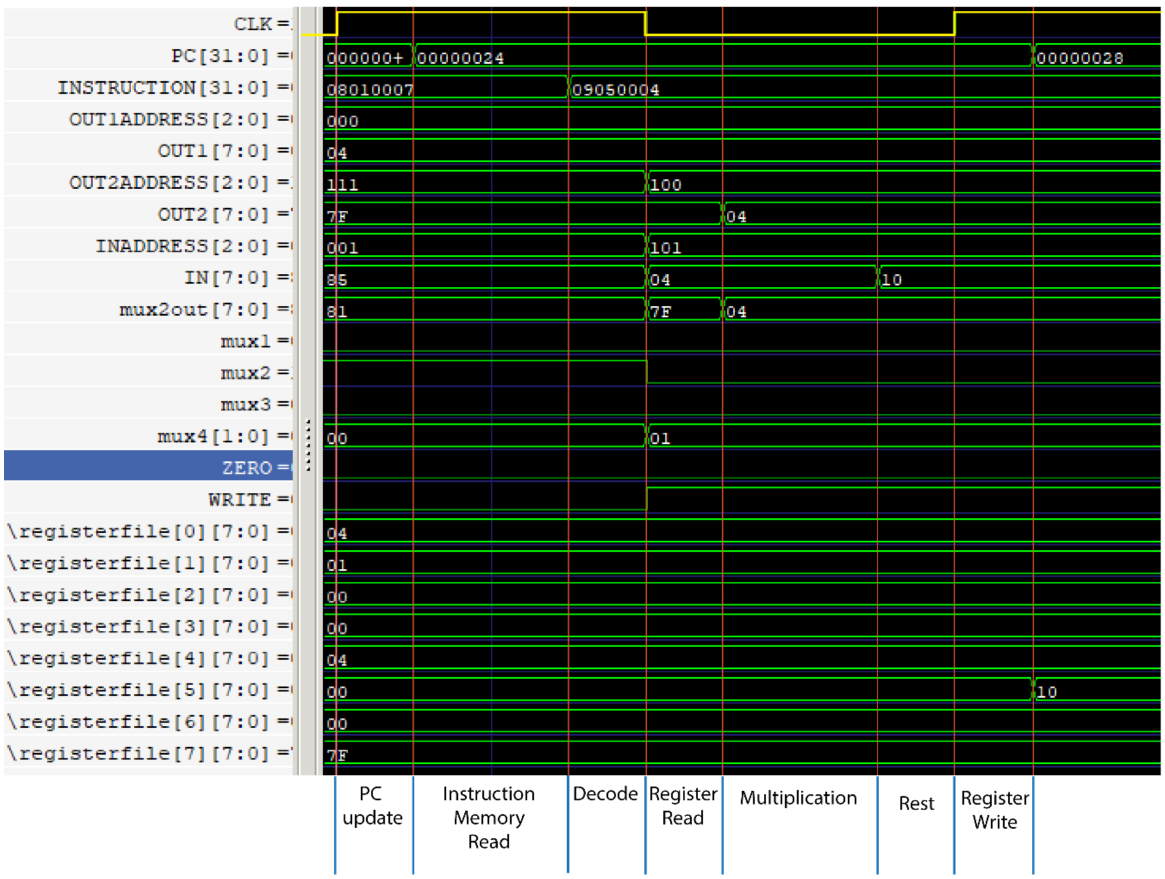


Verilog Code

```
module multiplier #(
    parameter W = 8
) (
    input  [W-1:0] MULTIPLICAND,
    input  [W-1:0] MULTIPLIER,
    output [W-1:0] OUT
);
    // normal multiplication as we do on decimal numbers
    assign #2 OUT = {MULTIPLICAND*MULTIPLIER[0]    } +
                    {MULTIPLICAND*MULTIPLIER[1],1'b0} +
                    {MULTIPLICAND*MULTIPLIER[2],2'b00} +
                    {MULTIPLICAND*MULTIPLIER[3],3'b000} +
                    {MULTIPLICAND*MULTIPLIER[4],4'b0000} +
                    {MULTIPLICAND*MULTIPLIER[5],5'b00000} +
                    {MULTIPLICAND*MULTIPLIER[6],6'b000000} +
                    {MULTIPLICAND*MULTIPLIER[7],7'b0000000};

Endmodule
```

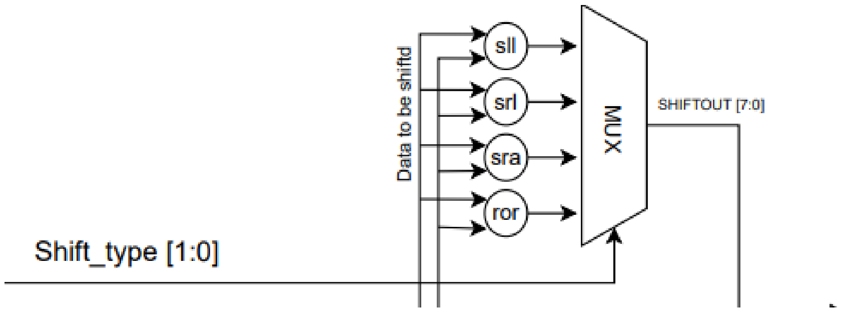
Waveforms



Shift

As shown below I have added new functional units for shifting and to select what type of shift is needed to perform base on the instruction. Shift_type [1:0] signal comes from the control unit.

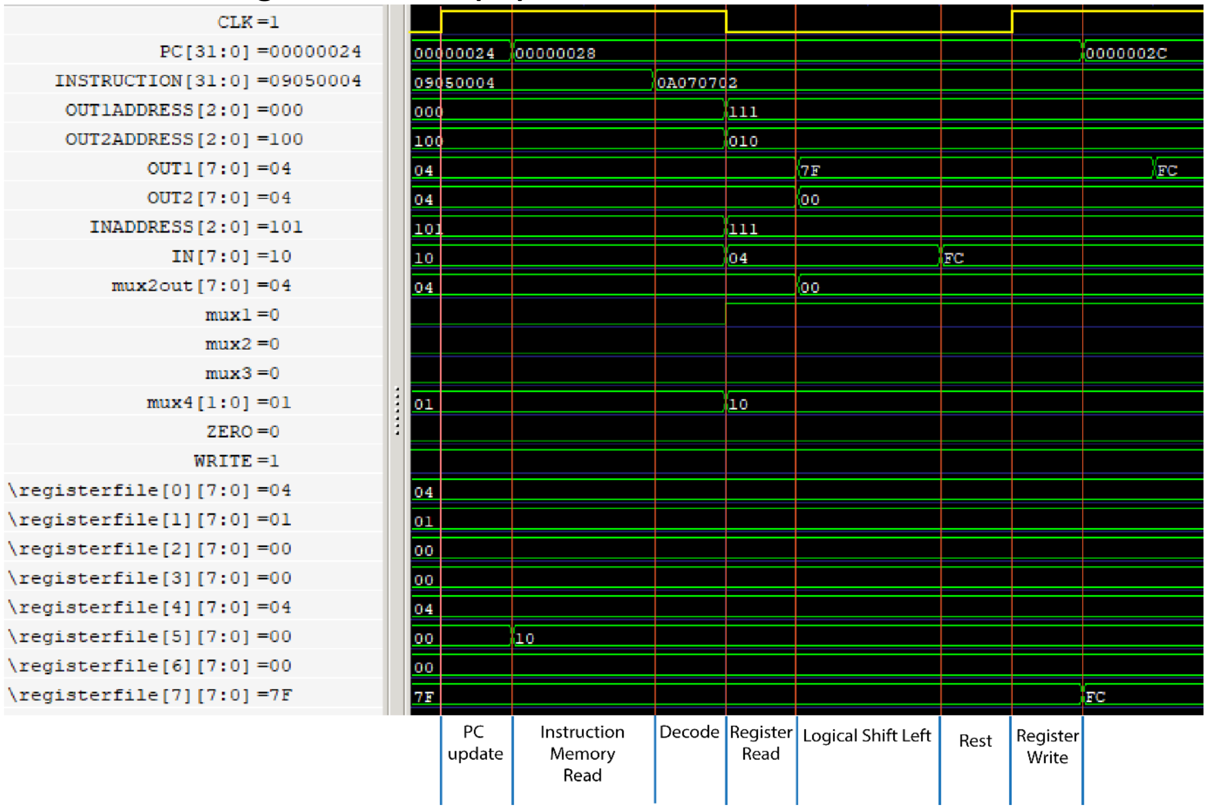
A delay of #3 is added for the shift operation.



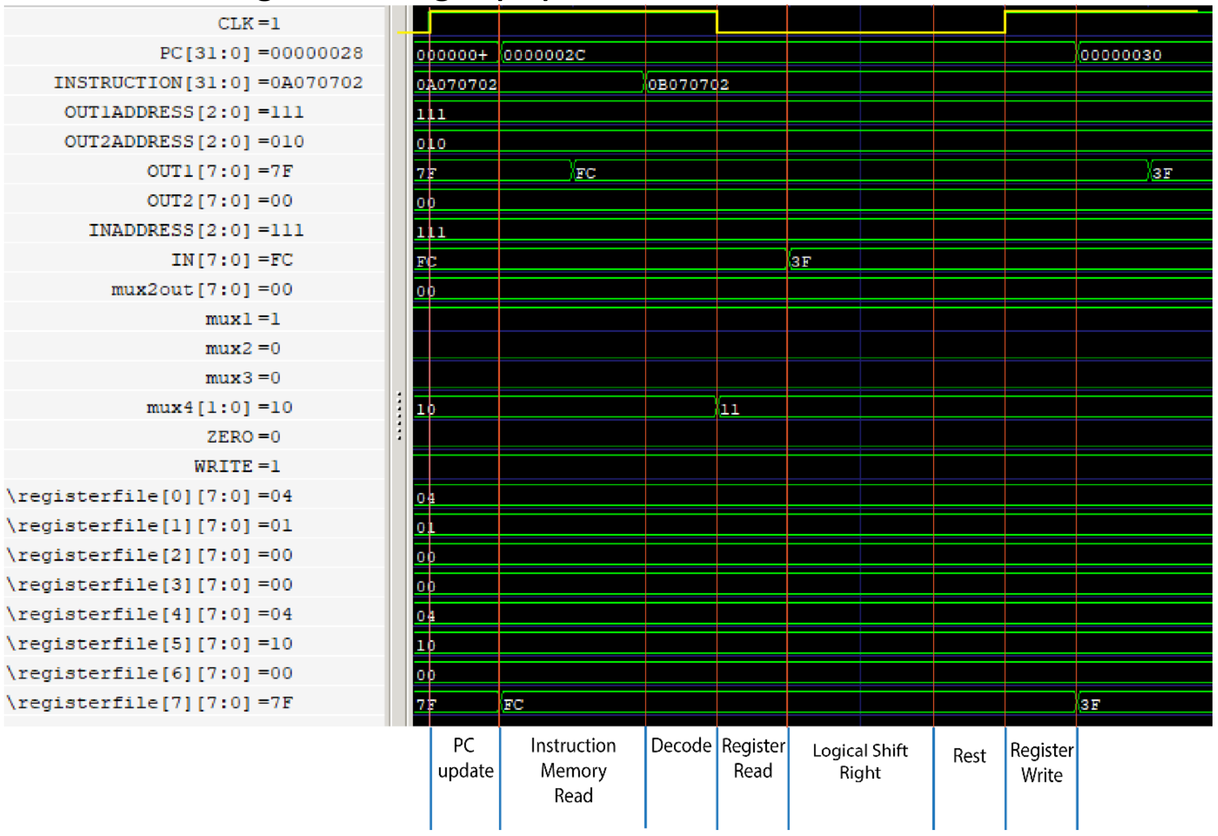
Verilog Code

```
module shift #(
    parameter W = 8
) (
    input       [W-1:0] data,
    input       [W-1:0] shift_amt,
    input       [1:0] shift_type,
    output reg  [W-1:0] result
);
    integer i;
    always @(*) begin
        #3
        result = data;
        case(shift_type)
            2'b00:  begin // LSL
                    for (i=0;i<shift_amt;i = i+1) result = {result[W-2:0], 1'b0};
                    end
            2'b01:  begin // LSR
                    for (i=0;i<shift_amt;i =i+1) result = {1'b0, result[W-1:1]};
                    end
            2'b10:  begin // SRA
                    for (i=0;i<shift_amt;i=i+1) result = {data[W-1], result[W-
1:1]};
                    end
            2'b11:  begin // ROR
                    for (i=0;i<shift_amt;i=i+1) result = {data[0], result[W-
1:1]};
                    end
        endcase
    end
endmodule
```

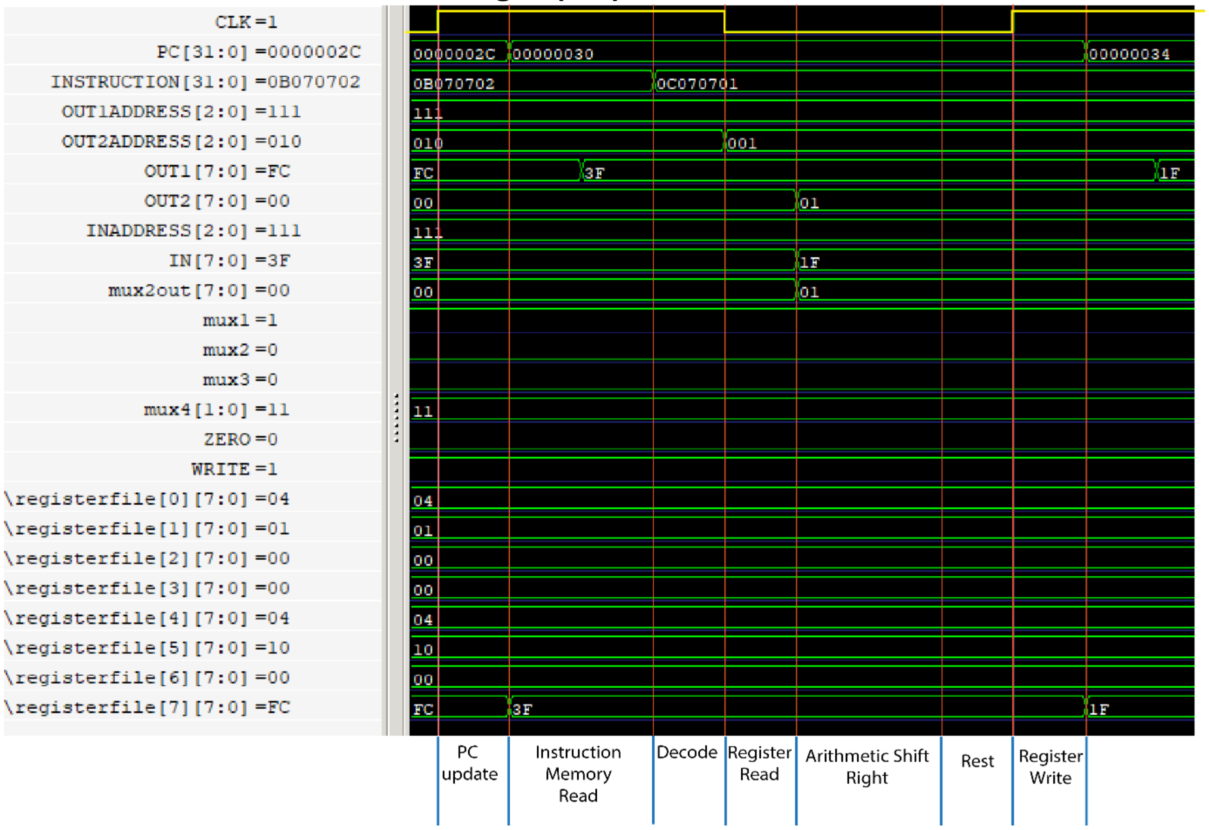
Waveform for logical shift left (sll)



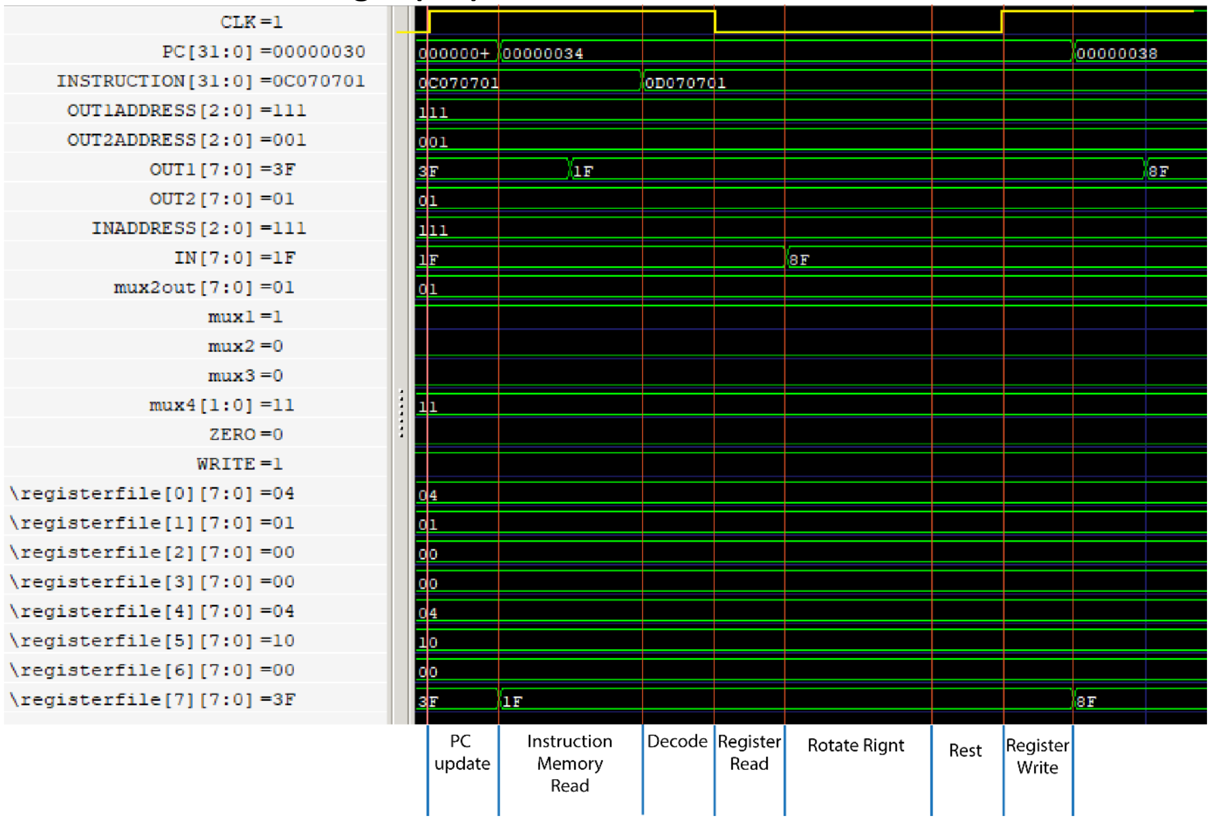
Waveform for logical shift right (srl)



Waveform for arithmetic shift right (sra)



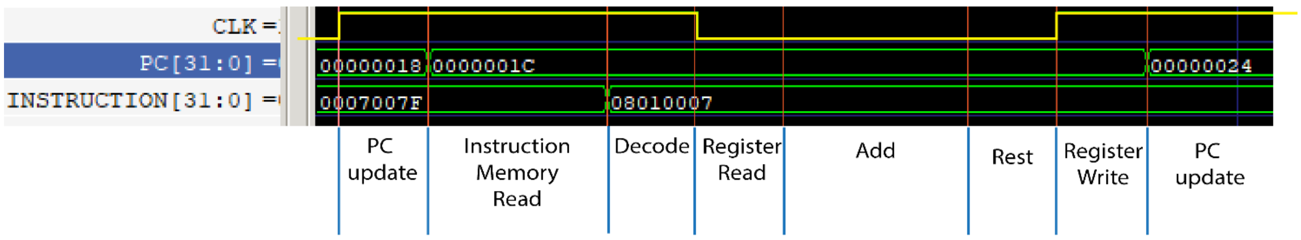
Waveform for rotate right (ror)



Bne instruction

For bne instruction we did a small modification for beq instruction. When we call beq control unit checks whether the given two registers equal or not and it indicates by a signal called ZERO. If ZERO = 1, given two registers are equal otherwise not equal. So if ZERO = 0 and instruction = 8'b00001000 then bne should execute.

Waveform for branch not equal (bne)

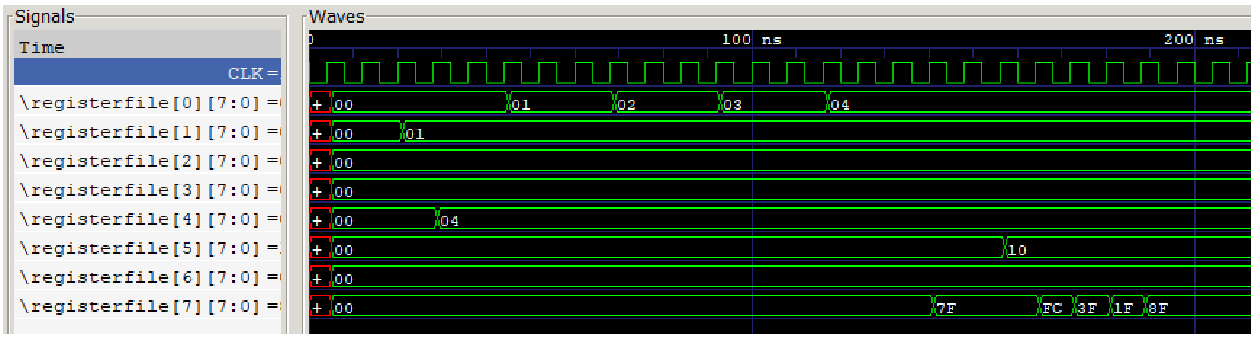


Initial Address : 28 (in decimal)
After bne instruction : 36 (in decimal)

Here is the sample assembly code that we used for testing purpose,

// Assembly program	// binary instructions
loadi 0 0x00	00000000 00000000 00000000 00000000
loadi 1 0x01	00000001 00000000 00000001 00000000
loadi 4 0x04	00000100 00000000 00000100 00000000
beq 0x2 0 4	00000100 00000000 00000010 00000111
add 0 0 1	00000001 00000000 00000000 00000001
j 0xFD	00000000 00000000 11111101 00000110
loadi 7 0xFF	01111111 00000000 00000111 00000000
bne 0x01 0 7	00000111 00000000 00000001 00001000
loadi 6 0xFF	11111111 00000000 00000110 00000000
mult 5 0 4	00000100 00000000 00000101 00001001
sll 7 7 0x02	00000010 00000111 00000111 00001010
srl 7 7 0x02	00000010 00000111 00000111 00001011
sra 7 7 0x01	00000001 00000111 00000111 00001100
ror 7 7 0x01	00000001 00000111 00000111 00001101

Variation of Register Values for the above program,



Instruction Formats

add (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_0000

add (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_0001

sub (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_0100

and (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_0010

orr (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_0011

beq (bits 31-24)	Addr>>2 (bits 23-16)	R1 (bits 15-8)	R2 (bits 7-0)

OPCODE:0000_0111

j (bits 31-24)	Addr>>2 (bits 23-16)	- (bits 15-8)	- (bits 7-0)

OPCODE:0000_0110

mult (bits 31-24)	Rd (bits 23-16)	Rt (bits 15-8)	Rs (bits 7-0)

OPCODE:0000_1001

sll (bits 31-24)	R1 (bits 23-16)	R2 (bits 15-8)	Shift_amt (bits 7-0)

OPCODE:0000_1010

srl (bits 31-24)	R1 (bits 23-16)	R2 (bits 15-8)	Shift_amt (bits 7-0)

OPCODE:0000_1011

sra (bits 31-24)	R1 (bits 23-16)	R2 (bits 15-8)	Shift_amt (bits 7-0)

OPCODE:0000_1100

ror (bits 31-24)	R1 (bits 23-16)	R2 (bits 15-8)	Shift_amt (bits 7-0)

OPCODE:0000_1101

bne (bits 31-24)	Addr>>2 (bits 23-16)	R1 (bits 15-8)	R2 (bits 7-0)

OPCODE:0000_1000

****

END OF THE DOCUMENT