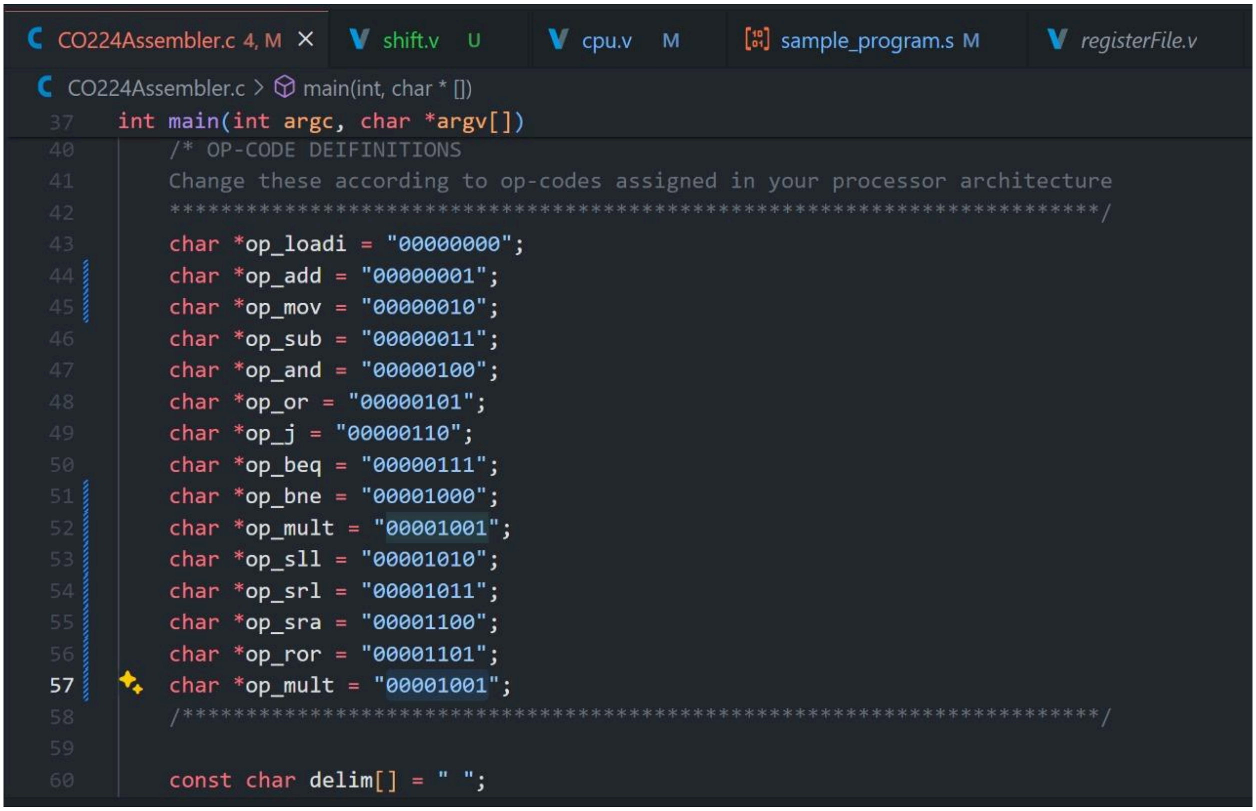
CO224 - Computer Architecture 2024
Department of Computer Engineering
Lab 5 – Building a Simple Processor
Part 5 – Extended ISA – Report

Group 02 – Dilshan D.M.T. (E/20/069) Karunarathne K.N.P. (E/20/189)

In Part 5, the CPU's instruction set was expanded by adding five new instructions. This required modifying the original CPU hardware and implementing several changes and restrictions in the CPU's assembler.

CPU assembler with the new instructions displayed below:



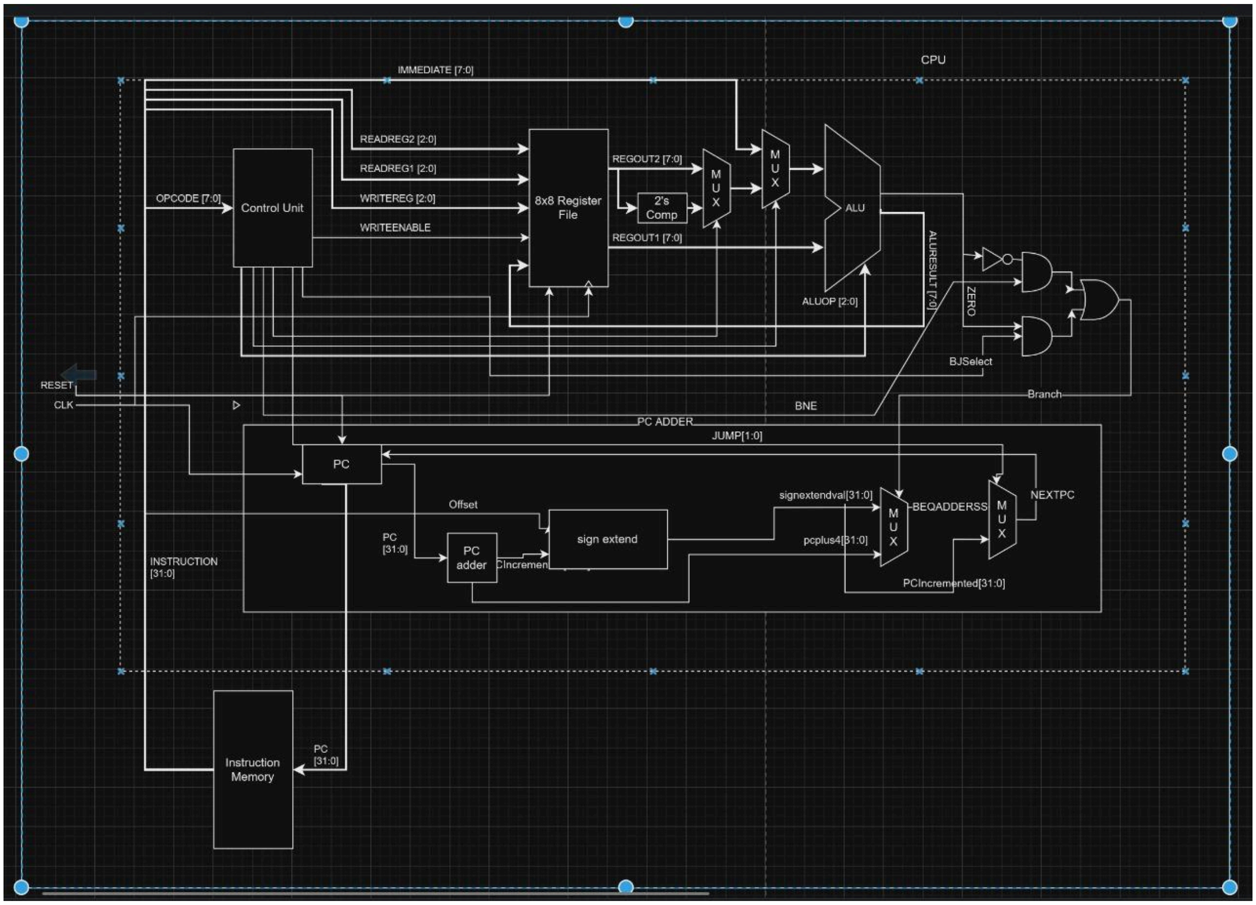
```
int main(int argc, char *argv[])
    /* OP-CODE DEIFINITIONS
    Change these according to op-codes assigned in your processor architecture
    ************************************************************************/
    char *op_loadi = "00000000";
    char *op_add = "00000001";
    char *op_mov = "00000010";
    char *op_sub = "00000011";
    char *op_and = "00000100";
    char *op_or = "00000101";
    char *op_j = "00000110";
    char *op_beq = "00000111";
    char *op_bne = "00001000";
    char *op_mult = "00001001";
    char *op_sll = "00001010";
    char *op_srl = "00001011";
    char *op_sra = "00001100";
    char *op_ror = "00001101";
    char *op_mult = "00001001";
    /***********************************************************************/

    const char delim[] = " ";
```

The new instructions and their intended usage are detailed below:

mult <r4> <r2> <r1>   : Multiplies the values in r1 and r2 and stores result in r4
sll <r3> <r2> <offset>: Logically left shifts r2 by offset value and stores result in r4
srl <r5> <r2> <offset>: Logically right shifts r2 by offset value and stores result in r5
sra <r1> <r7> <offset>: Arithmetically right shifts r7 by offset value and stores result in r1
ror <r4> <r1> <offset>: Rotates r1 right by offset value and stores result in r4
bne <offset> <r1> <r2>: Branches based on offset if values in r1 and r2 are not equal

To support these new instructions, we made a slight adjustment to the Branch/Jump hardware and added several functional units to the CPU's ALU. These modifications are depicted in the updated CPU block diagram and the ALU functions table provided below.

CPU block diagram:



ALU functions:

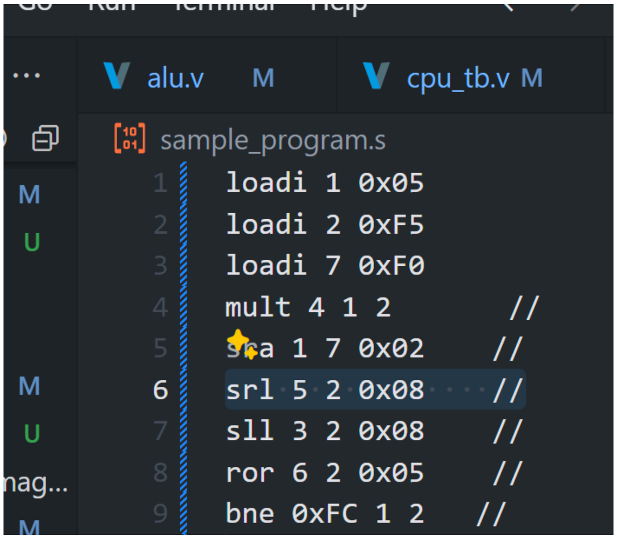
Select	Function	Description	Supported Instruction	Delay
000	FORWARD	DATA2→RESULT	loadi,mov	#2
001	ADD	DATA1+ DATA2→RESULT	add,sub,beq,bne	#2
010	AND	DATA1& DATA2→RESULT	and	#1
011	OR	DATA1 |DATA2→RESULT	or	#1
100	LSHIFT	DATA1<<DATA2→RESULT	sll	#2
101	RSHIFT	(Logical Shift) DATA1>>DATA2→RESULT Or (Arithmetic Shift) DATA1>>>DATA2→RESULT Or Rotate Right DATA1 by DATA2 times (Operation chosen depends on first two MSB bits of DATA2)	sra,srl,ror	#2
110	MULT	DATA1*DATA2→RESULT	mult	#2

The instruction set used in this implementation is shown as follows:

loadi 1 0x05 (Load the immediate value 5 to register 1)
loadi 2 0xF5 (Load the immediate value F5 to register 2)
loadi 7 0xF0 (Load the immediate value F0 to register 7)
—-----------New Instructions—---------------------
mult 4 1 2 (multiply value in register 1 by value in register 2, and place the result in register 4)

sra 1 7 0x02 (apply arithmetic shift right 2 times on value in register 7, and place the result in register 1)
srl 5 2 0x08 (apply logical shift right 8 times on value in register 2, and place the result in register 5)
sll 3 2 0x08 (apply logical shift left 8 times on value in register 2, and place the result in register 3)
ror 4 1 0x02 (apply rotate right 2 times on value in register 1, and place the result in register 4)
bne 0x02 1 2 (if values in registers 1 and 2 are not equal, branch 2 instructions forward)

Instruction set taken in the implementation:

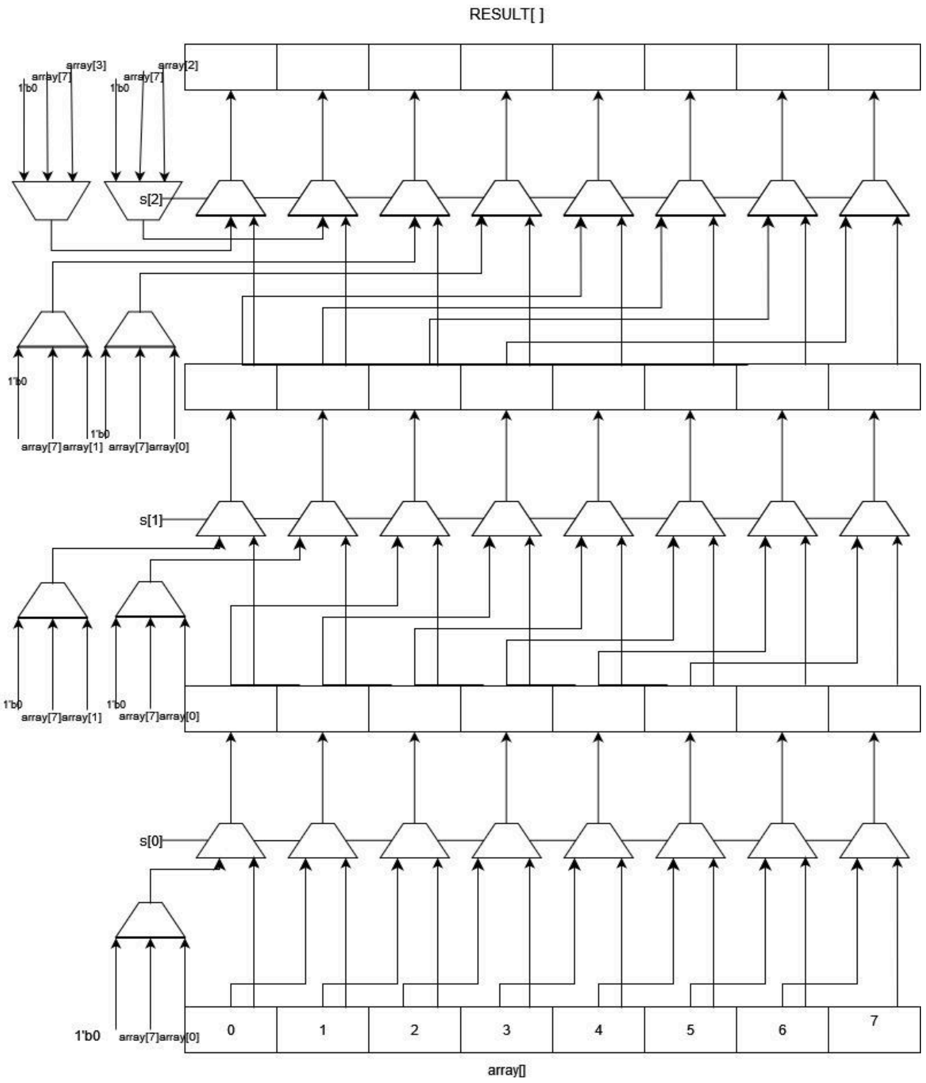


```
loadi 1 0x05
loadi 2 0xF5
loadi 7 0xF0
mult 4 1 2        //
sra 1 7 0x02     //
srl 5 2 0x08     //
sll 3 2 0x08     //
ror 6 2 0x05     //
bne 0xFC 1 2    //
```

RSHIFT Unit

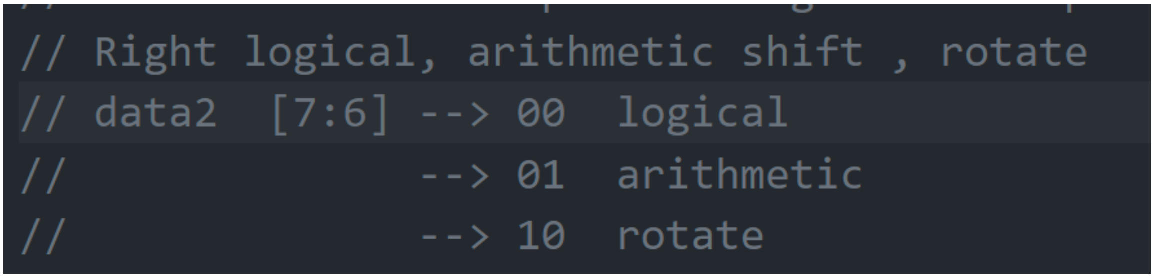
The three right shift instructions `srl/sra/ror` were implemented using a single functional unit called RSHIFT, which manages all right shift and rotate operations.

Here is the hardware implementation of right shift unit:



This unit determines the correct operation to perform based on the first two most significant bits (MSBs) of the shift value provided in DATA2.

These different operations identified as shown below by this method:



```
// Right logical, arithmetic shift , rotate
// data2  [7:6] --> 00  logical
//              --> 01  arithmetic
//              --> 10  rotate
```

As with the RSHIFT unit, only the first three least significant bits (LSBs) of the shift value are considered, Since shifts greater than 8 logically shift generate 0 value, arithmetic shift generates -1 and rotate generates the same value. As the above diagram we shift binary values using 2x1 Mux. The empty blocks were right shifted using 3x1 Mux. Input of the 3x1 Mux will decide whether it's a rotate arithmetically shifted or logically .

`srl/sra/ror:`

PC Update	Instruction Memory Read		Register Read		ALU	
#1	#2		#2		#1	
	PC+4 Adder		Decode			
	#1		#1			
Register Write						
#1						

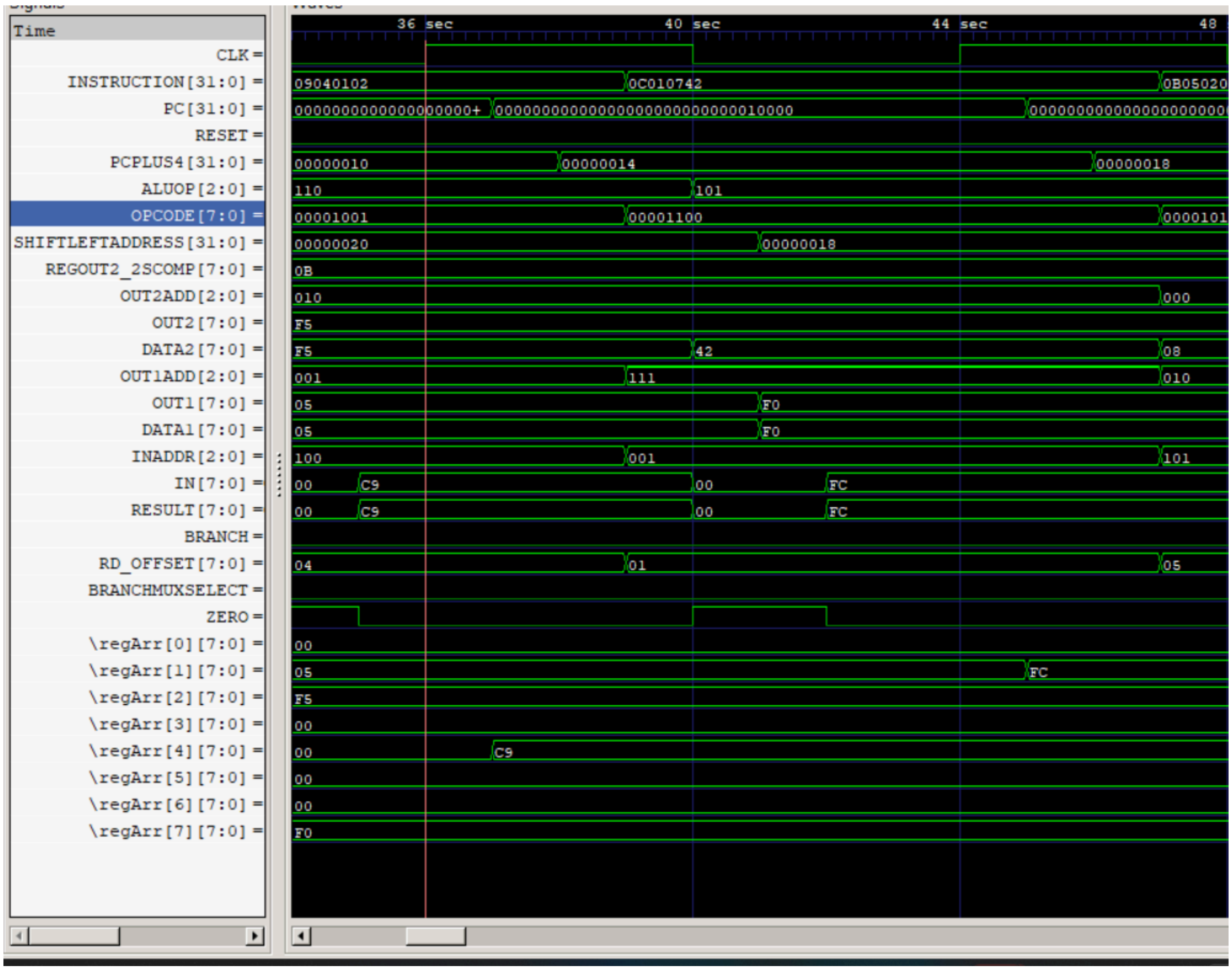


Figure : Arithmatical right shift(sra)

Even Though We have added 2 unit delay to the alu the total delay we get after the register read is 1 unit.

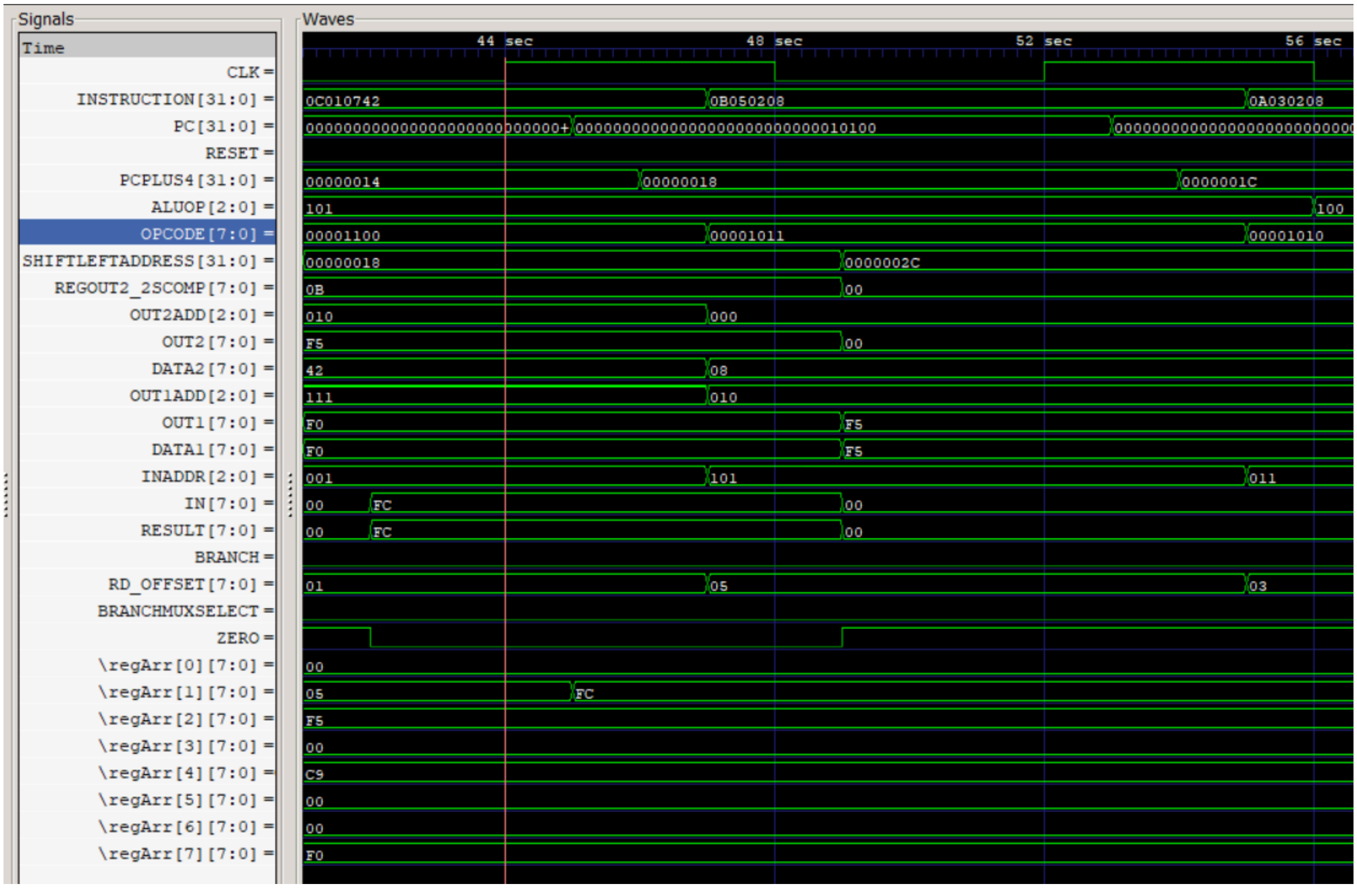


Figure : Logical shift right (srl)

srl/sra/ror:

PC Update	Instruction Memory Read		Register Read		ALU	
#1	#2		#2		#1	
	PC+4 Adder		Decode			
	#1		#1			
Register Write						
#1						

srl/sra/ror:

PC Update	Instruction Memory Read		Register Read		ALU	
#1	#2		#2		#1	
	PC+4 Adder		Decode			

	#1		#1	
Register Write				
#1				

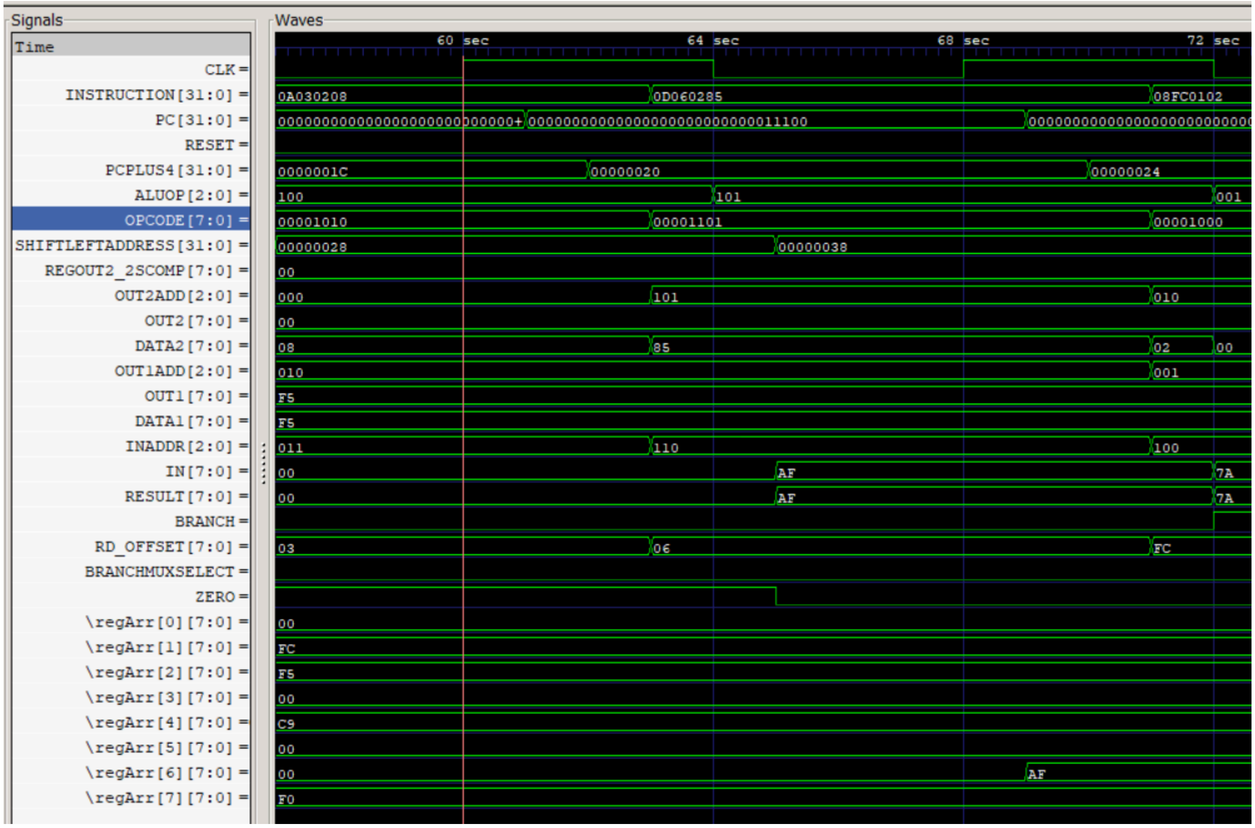


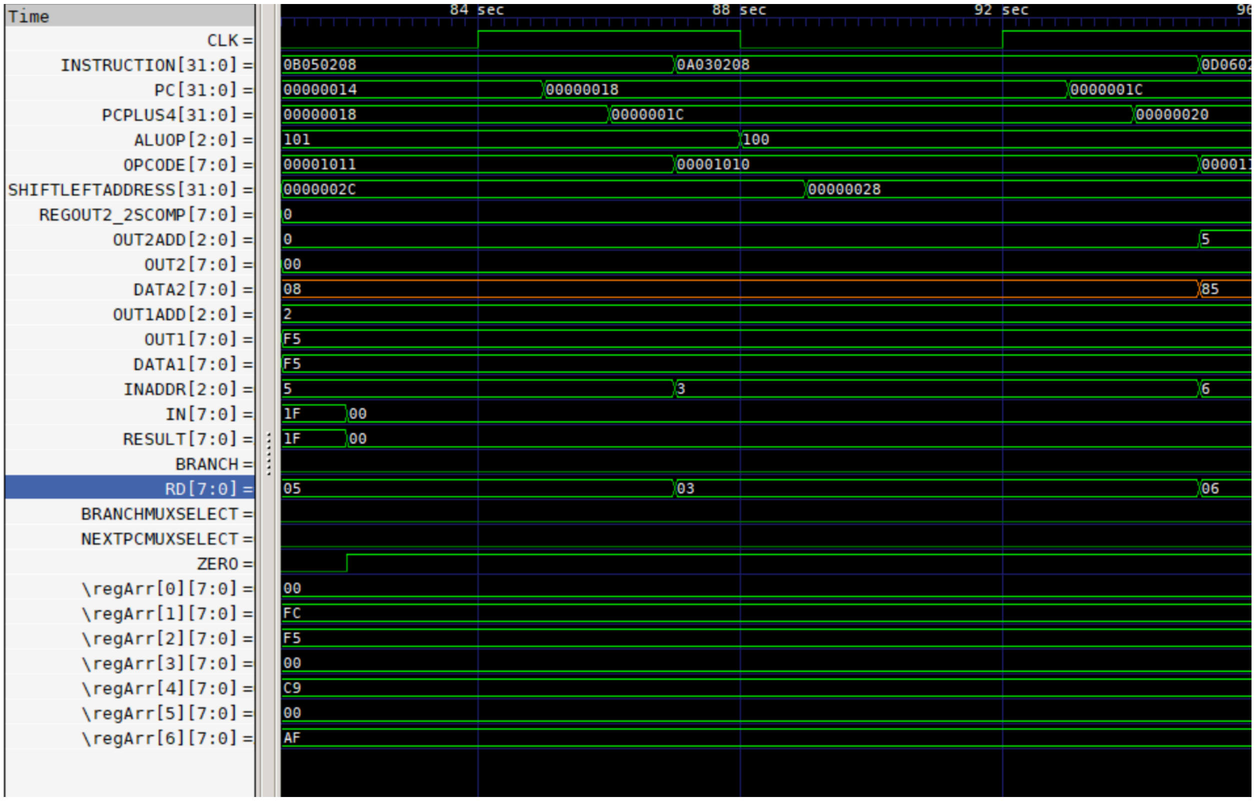
Figure : rotation (ror)

LSHIFT Unit

Another functional unit was added to the ALU to enable bitwise left shift operations, using multiple layers of multiplexers (MUXes) similar to the logical shift right to produce the shifted output.

Given the 8-bit size of the data word, shifting by a value greater than 8 gives the same result zero. To reduce hardware complexity, the shifter only supports shift operations up to 8 shifts.

The timing for the left shift instruction is as follows

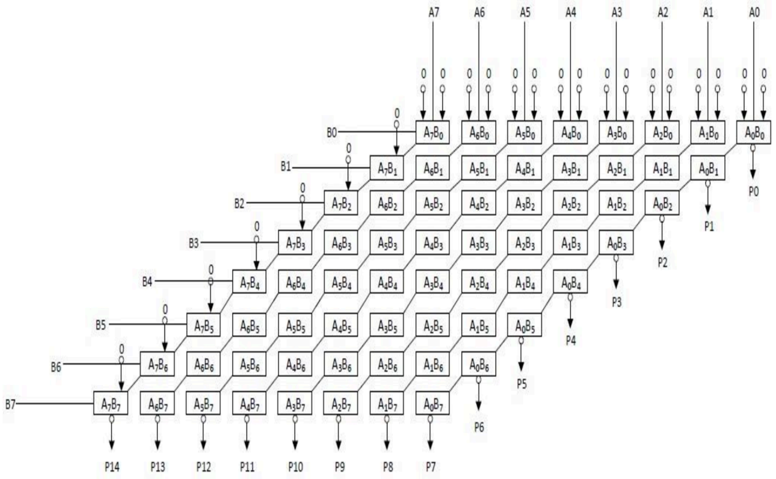


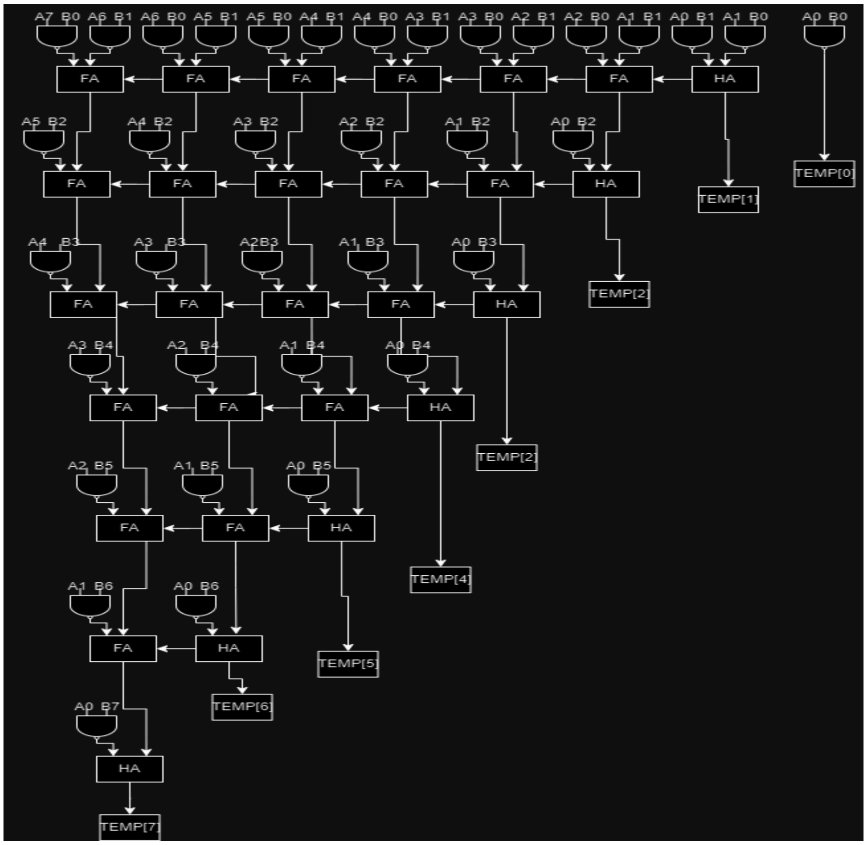
```
Time                        84 sec       88 sec       92 sec       96
CLK =
INSTRUCTION[31:0] =         0B050208     0A030208     0D060
PC[31:0] =                  00000014     00000018     0000001C
PCPLUS4[31:0] =             00000018     0000001C     00000020
ALUOP[2:0] =                101          100
OPCODE[7:0] =               00001011     00001010     00001
SHIFTLEFTADDRESS[31:0] =    0000002C     00000028
REGOUT2_2SCOMP[7:0] =       0
OUT2ADD[2:0] =              0            5
OUT2[7:0] =                 00
DATA2[7:0] =                08           85
OUT1ADD[2:0] =              2
OUT1[7:0] =                 F5
DATA1[7:0] =                F5
INADDR[2:0] =               5            3            6
IN[7:0] =                   1F           00
RESULT[7:0] =               1F           00
BRANCH =
RD[7:0] =                   05           03           06
BRANCHMUXSELECT =
NEXTPCMUXSELECT =
ZERO =
\regArr[0][7:0] =           00
\regArr[1][7:0] =           FC
\regArr[2][7:0] =           F5
\regArr[3][7:0] =           00
\regArr[4][7:0] =           C9
\regArr[5][7:0] =           00
\regArr[6][7:0] =           AF
```

PC Update	Instruction Memory Read	Register Read	ALU	
#1	#2	#2	#2	
	PC+4 Adder	Decode		
	#1	#1		
Register Write				
#1				

Mult Unit

For the MULT instruction, a separate functional unit was added to the ALU.This unit uses an array of Full Adders and Half Adders arranged in multiple layers to compute each bit of the output. The Full Adder was also implemented as a module using half adders and the half adder was made using basic combinational logic.

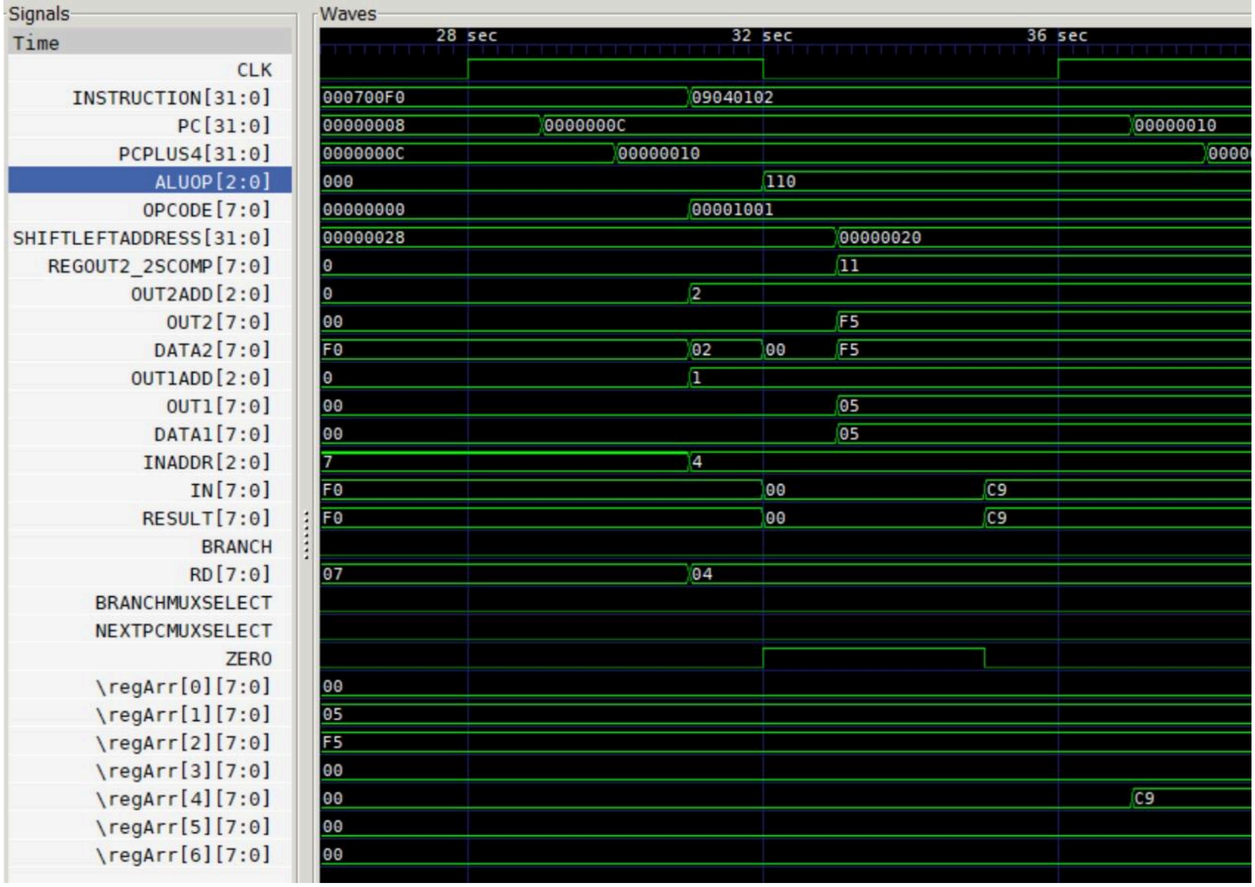

A7 A6 A5 A4 A3 A2 A1 A0
0
B0 B1 B2 B3 B4 B5 B6 B7
A_7B_0 A_6B_0 A_5B_0 A_4B_0 A_3B_0 A_2B_0 A_1B_0 A_0B_0
A_7B_1 A_6B_1 A_5B_1 A_4B_1 A_3B_1 A_2B_1 A_1B_1 A_0B_1
A_7B_2 A_6B_2 A_5B_2 A_4B_2 A_3B_2 A_2B_2 A_1B_2 A_0B_2
A_7B_3 A_6B_3 A_5B_3 A_4B_3 A_3B_3 A_2B_3 A_1B_3 A_0B_3
A_7B_4 A_6B_4 A_5B_4 A_4B_4 A_3B_4 A_2B_4 A_1B_4 A_0B_4
A_7B_5 A_6B_5 A_5B_5 A_4B_5 A_3B_5 A_2B_5 A_1B_5 A_0B_5
A_7B_6 A_6B_6 A_5B_6 A_4B_6 A_3B_6 A_2B_6 A_1B_6 A_0B_6
A_7B_7 A_6B_7 A_5B_7 A_4B_7 A_3B_7 A_2B_7 A_1B_7 A_0B_7
P0 P1 P2 P3 P4 P5 P6
P14 P13 P12 P11 P10 P9 P8 P7

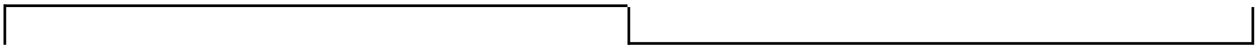


However, the multiplier is limited by the need to fit the result within 8 bits.Therefore, any multiplication operation which needs a result greater than 255 will result in an inaccurate output.

Since the MULT unit uses a large array of Adders, it can be assumed that a significant amount of time is taken for the calculation of the result when compared with the other functional units of the ALU. Therefore, a simulation delay of #2 time units was introduced for this unit.

The timing for the MULT instruction is as follows
.

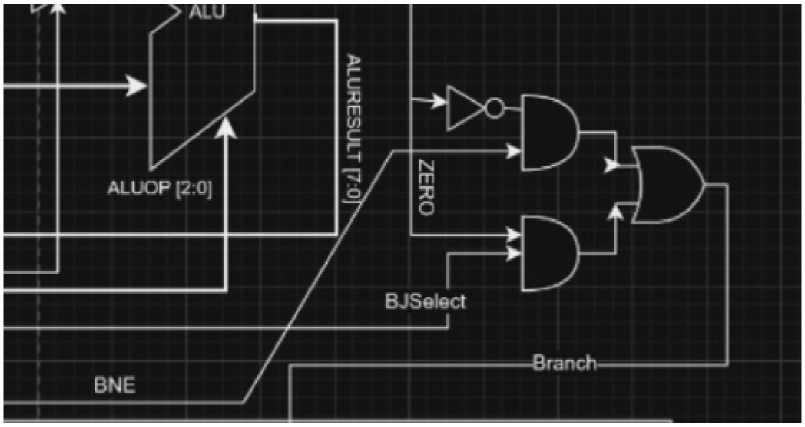




PC Update	Instruction Memory Read		Register Read		ALU	
#1	#2		#2		#2	
	PC+4 Adder		Decode			
	#1		#1			
Register Write						
#1						

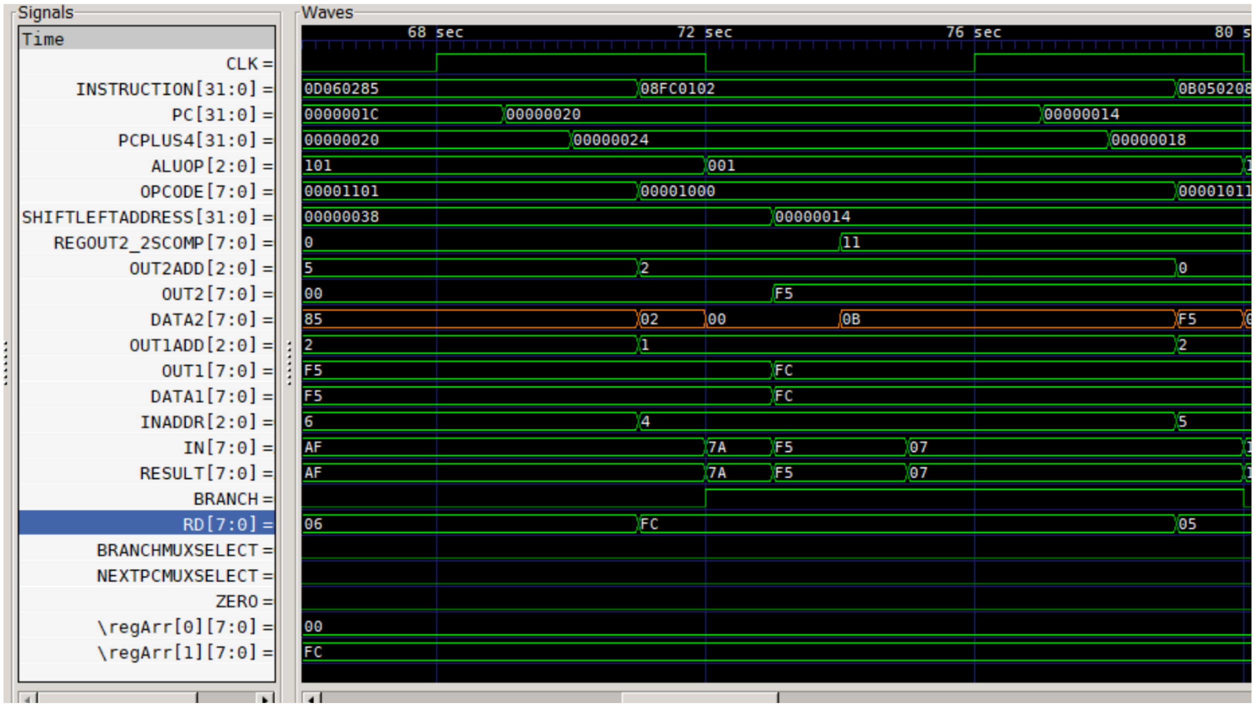
BNE instruction

Here is the hardware implementation of BNE instruction.



As shown in the CPU block diagram, the Flow Control Unit has been modified using combinational logic gates.
If BNE is high and ZERO is LOW then its identified as BNE True or If BjSelect is high and ZERO is HIGH then it is identified as BEQ is True and it branch to the relevant location.

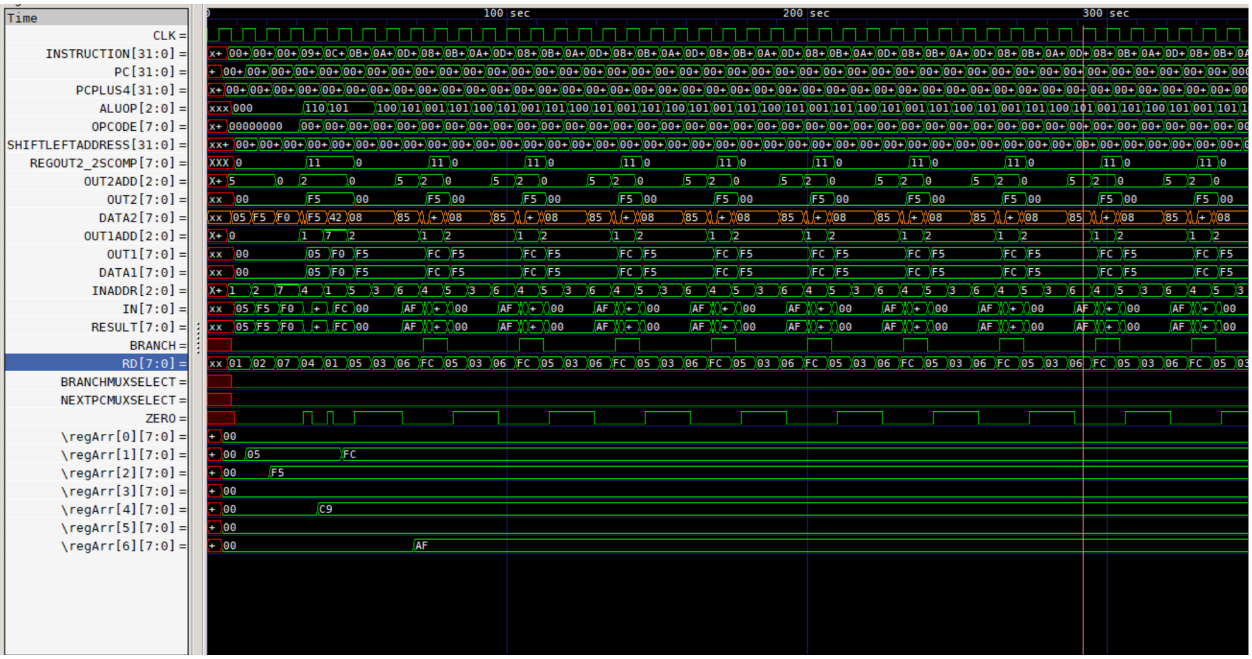
The timing for the BNE instruction is as follows



PC Update	Instruction Memory Read	Register Read		

<table>
<tr><td>#1</td><td colspan="2">#2</td><td colspan="2">#2</td><td></td><td></td></tr>
<tr><td rowspan="2"></td><td>PC+4 Adder</td><td rowspan="2"></td><td colspan="2">Branch/Jump Target Adder</td><td colspan="2"></td></tr>
<tr><td>#1</td><td colspan="2">#2</td><td colspan="2"></td></tr>
<tr><td colspan="3" rowspan="2"></td><td>Decode</td><td colspan="3" rowspan="2"></td></tr>
<tr><td>#1</td></tr>
</table>

Since register 1 and 2 values are not equal after coming to the end it loops again after branching to the given memory location.We can see that they remain being not equal throughout the programme therefore the loops run infinitely as shown in the figure below



Time
100 sec
200 sec
300 sec
CLK
INSTRUCTION[31:0]
PC[31:0]
PCPLUS4[31:0]
ALUOP[2:0]
OPCODE[7:0]
SHIFTLEFTADDRESS[31:0]
REGOUT2_2SCOMP[7:0]
OUT2ADD[2:0]
OUT2[7:0]
DATA2[7:0]
OUT1ADD[2:0]
OUT1[7:0]
DATA1[7:0]
INADDR[2:0]
IN[7:0]
RESULT[7:0]
BRANCH
RD[7:0]
BRANCHMUXSELECT
NEXTPCMUXSELECT
ZERO
\regArr[0][7:0]
\regArr[1][7:0]
\regArr[2][7:0]
\regArr[3][7:0]
\regArr[4][7:0]
\regArr[5][7:0]
\regArr[6][7:0]