EE446 Laboratory Work 2

Tunahan Aktaş

1. Preliminary Work

1.1. Arithmetic Logic Processor

1.1.1. Datapath Design

1.1.1.1. Datapath Design: Part 1

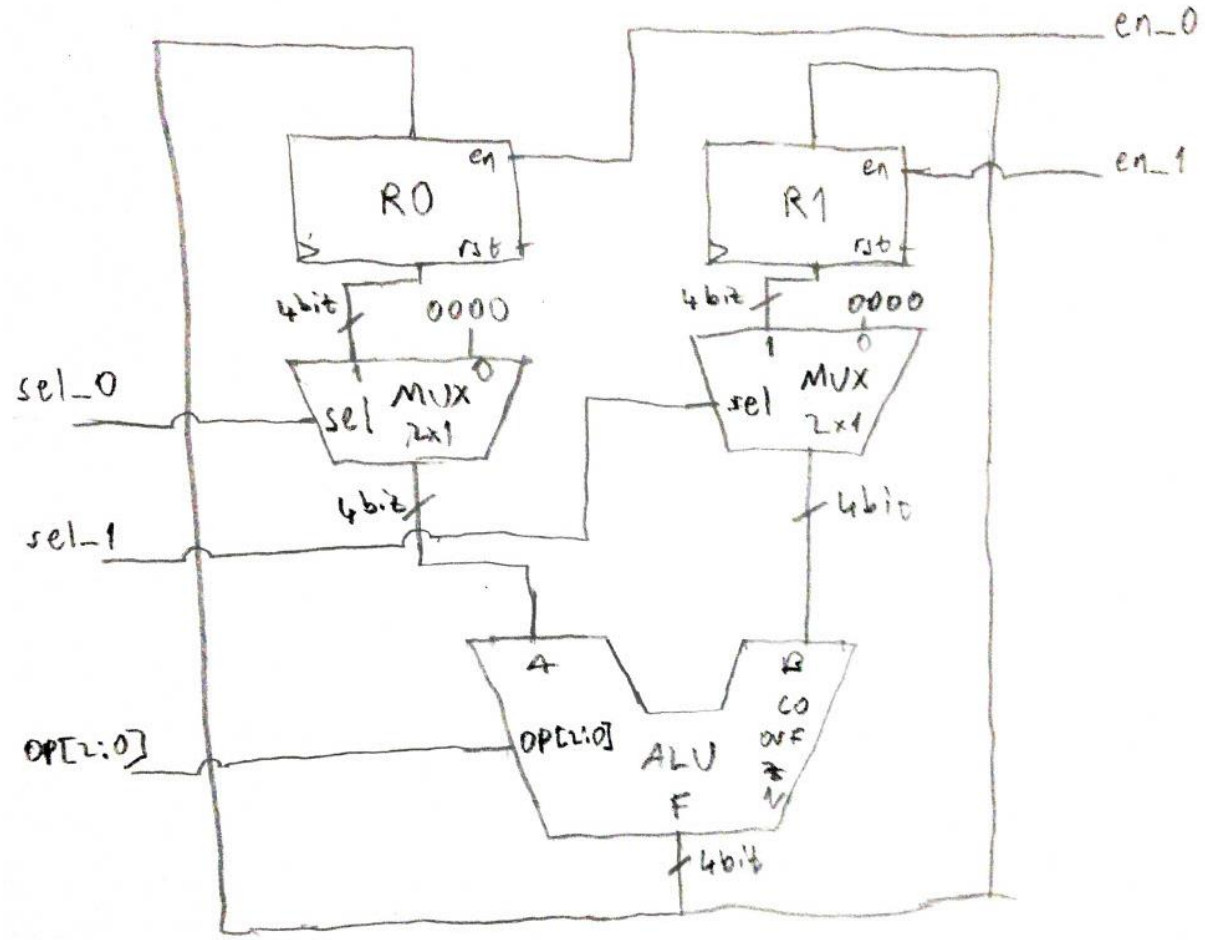


Figure 1: Shows the datapath, designed for part 1.

ALU is exported from the previous laboratory. It can make A-B or B-A operations.

- R0←-R0 operation: R0 is selected for A input port of ALU and 0 is selected for B input port. Write enable signal for R0 is 1, for R1 is 0. B-A operation is selected for ALU.
- R1←-R1 operation: 0 is selected for A input port of ALU and R1 is selected for B input port. Write enable signal for R0 is 0, for R1 is 1. A-B operation is selected for ALU.

Table 1: Control signals for datapath design part 1.

Control Signals	R0←-R0	R1←-R1
sel_0	1	0
sel_1	0	1
en_0	1	0
en_1	0	1
OP[2:0]	010 (B-A)	001 (A-B)

1.1.1.2. Datapath Design: Part 2

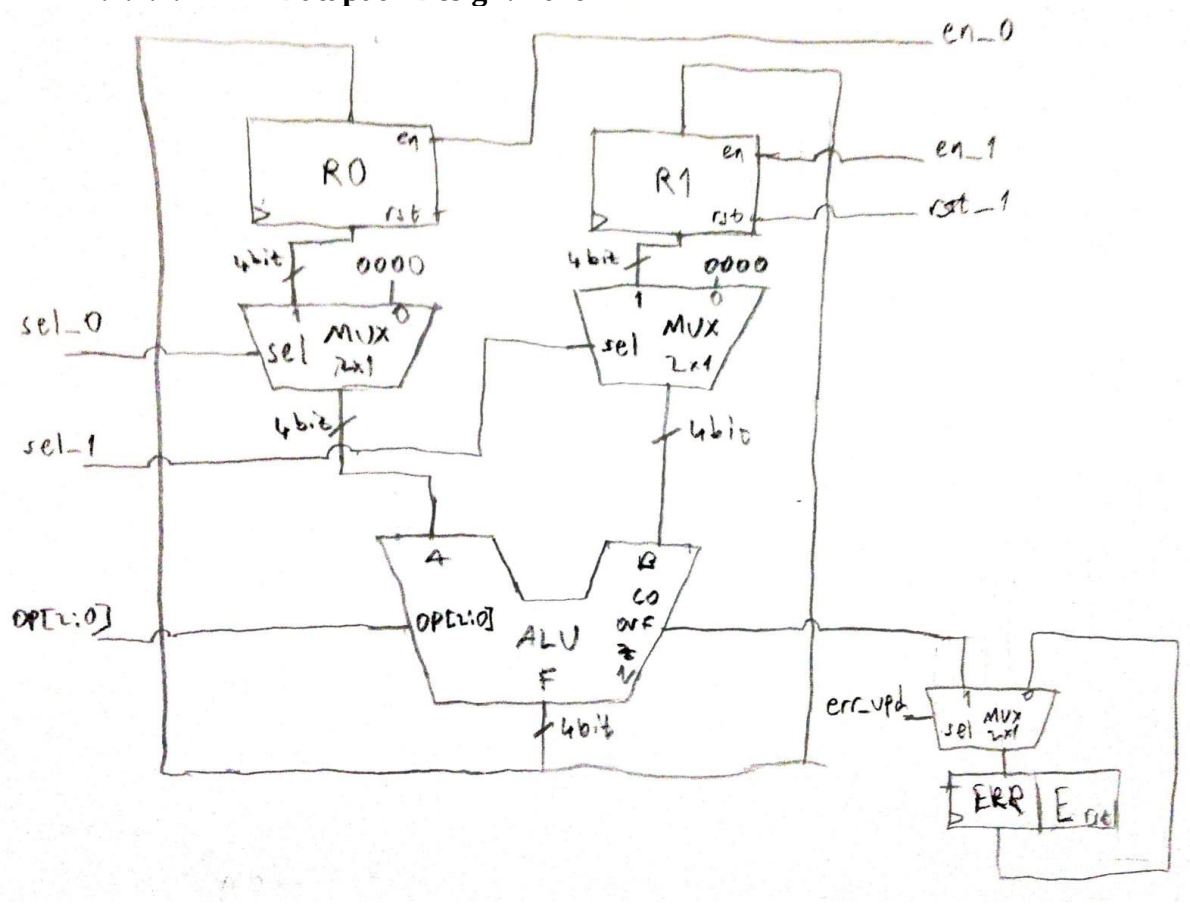


Figure 2: Shows the datapath, designed for part 2.

The ALU that we designed in the previous laboratory is already capable of doing ALP operations. We should send proper signals to our architecture and expand the datapath with the flag registers.

Table 2: Combined control signals for datapath design part 1 and 2.

Operations	Signals						
	sel_0	sel_1	en_0	en_1	rst_1	err_upd	OP[2:0]
Addition	1	1	1	0	1	1	000
Subtraction	1	1	1	0	1	1	001
AND	1	1	1	0	1	0	100
OR	1	1	1	0	1	0	101
EXOR	1	1	1	0	1	0	110
Bit Clear	1	1	1	0	1	0	111
Negative_R0	1	0	1	0	0	0	010

Negative_R1	0	1	0	1	0	0	001

1.1.1.3. Datapath Design: Multiplication Algorithm

In my design, I will use Booth's algorithm for signed 2's complement multiplication. Assume we have string of ones starting at m[th] bit and ending at n[th] bit:

$$Q = ..001..11 \ldots 00$$

We can express the magnitude of the string of ones as:

$$Q = 2^m + 2^{m+1} + .. + 2^n = \sum_{i=0}^{n} 2^i - \sum_{i=0}^{m-1} 2^i$$

Then, using geometric series equations:

$$Q = \frac{1 - 2^{n+1}}{1 - 2} + \frac{1 - 2^m}{1 - 2} = 2^{n+1} - 2^m$$

Booth's algorithm keeps track of string of ones in multiplier and uses the observation we made above.

If n_i^{th} bit is the starting bit of the i^{th} string of ones of Q and m_i^{th} bit is the ending bit of the i^{th} string of ones of Q, we can write:

$$A = Q.B = (2^{n_1+1} - 2^{m_1} + \cdots + 2^{n_i+1} - 2^{m_i}).B$$

Now, let us look at the flow chart to see how the algorithm keeps track of the string of ones.

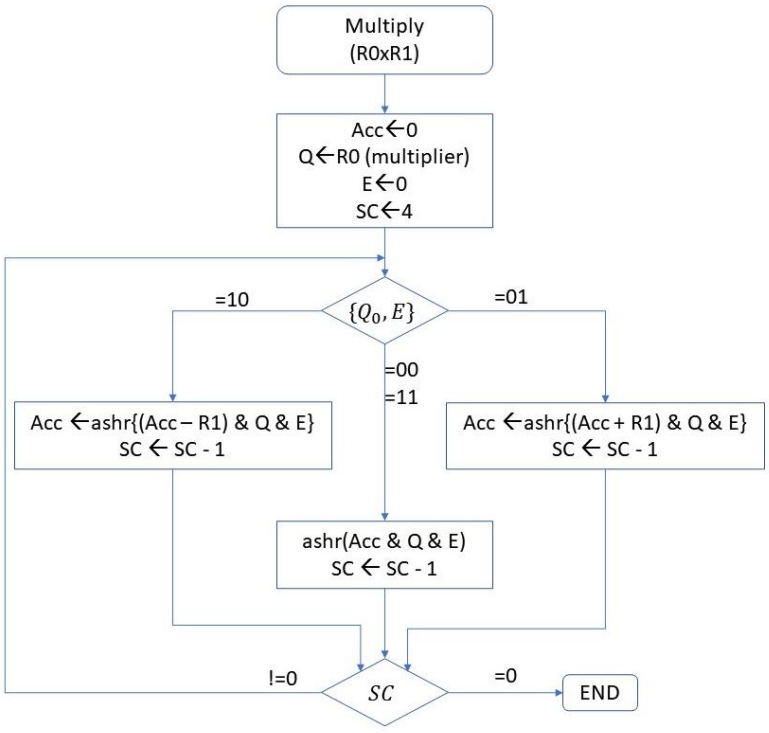


Figure 3: Shows flow chart of the multiplication algorithm.

Since we have 4-bit architecture, we will have 3-bit state counter to count 4 to 0. At first cycle, assuming we already have multiplicand in R0 and multiplayer in R1, we will load R0 to Q, Acc to 0, E to 0 and SC to 4. Then, starting from the least significant bit, we will check whether the least significant bit at that cycle is different than the previous cycle by checking $\{Q_n, E\}$ value. If:

$\{Q_0, E\} = 10$: Previous LSB was 0 and new LSB is 1. Start of string of 1's. We will subtract Acc from R1 and load the arithmetically shifted right version of the result to Acc. We will shift right Q and E also.

$\{Q_0, E\} = 01$: Previous LSB was 1 and new LSB is 0. Start of string of 0's. We will add Acc to R1 and load the arithmetically shifted right version of the result to Acc. We will shift right Q and E also.

$\{Q_0, E\} = 00\ or\ 11$: 00 or 11: Neither start nor end of string of 0′s or 1′s. We will shift right {Acc, Q, E}.

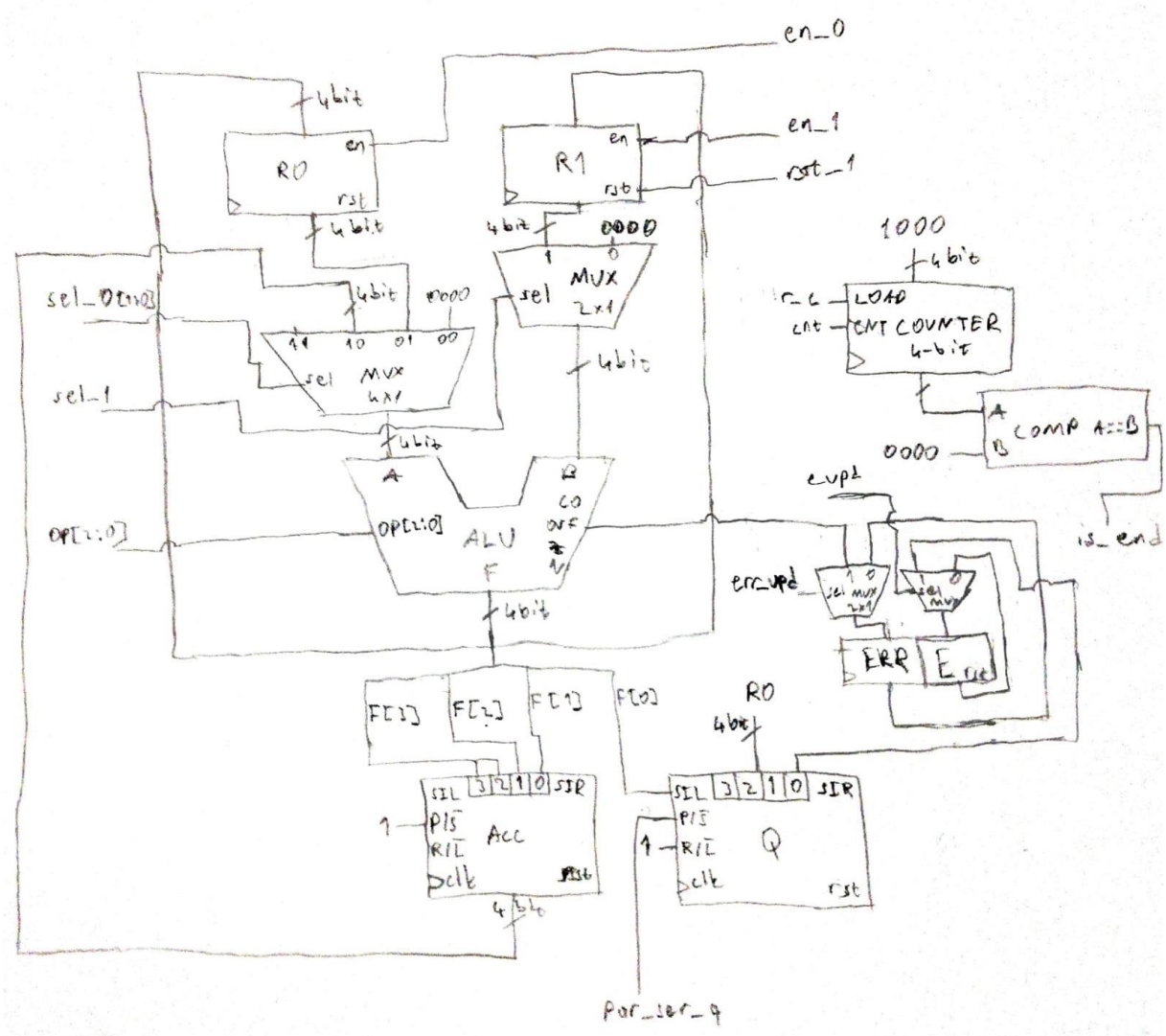


Figure 4:Shows updated datapath of the multiplication algorithm.

Updates:

- Acc register takes ashr(F) always,
- F[0] shifted left to Q always,
- Acc output is added to source A of the ALU and can be selected by sel_0[1:0],
- Q[0] can be shifted to E,
- 4 bit counter added.

- If we choose Acc as ALU source A, 0 as source B and ADD as ALU operation; we will have ashr(Acc & Q & E) at next clock edge.
- If we choose Acc as ALU source A, R0 as source B and ADD as ALU operation; we will have ashr((Acc+R0) & Q & E) at next clock edge.
- If we choose Acc as ALU source A, R0 as source B and SUB as ALU operation; we will have ashr((Acc-R0) & Q & E) at next clock edge.

We need to send Q_0 and E signals to the controller so that it can give the different signals to the datapath depending on the conditions discussed above.

1.1.1.4. Datapath Design: Division Algorithm

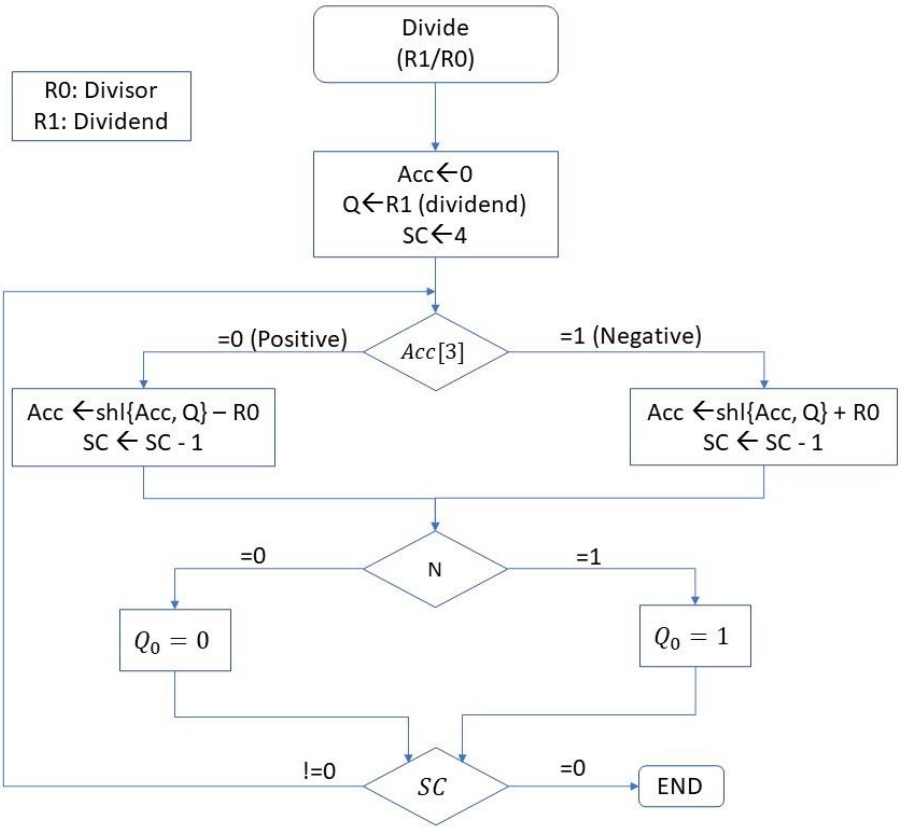


Figure 5: Flow chart of the non-restoring division algorithm.

I choose to design faster algorithm with relatively complex datapath. .

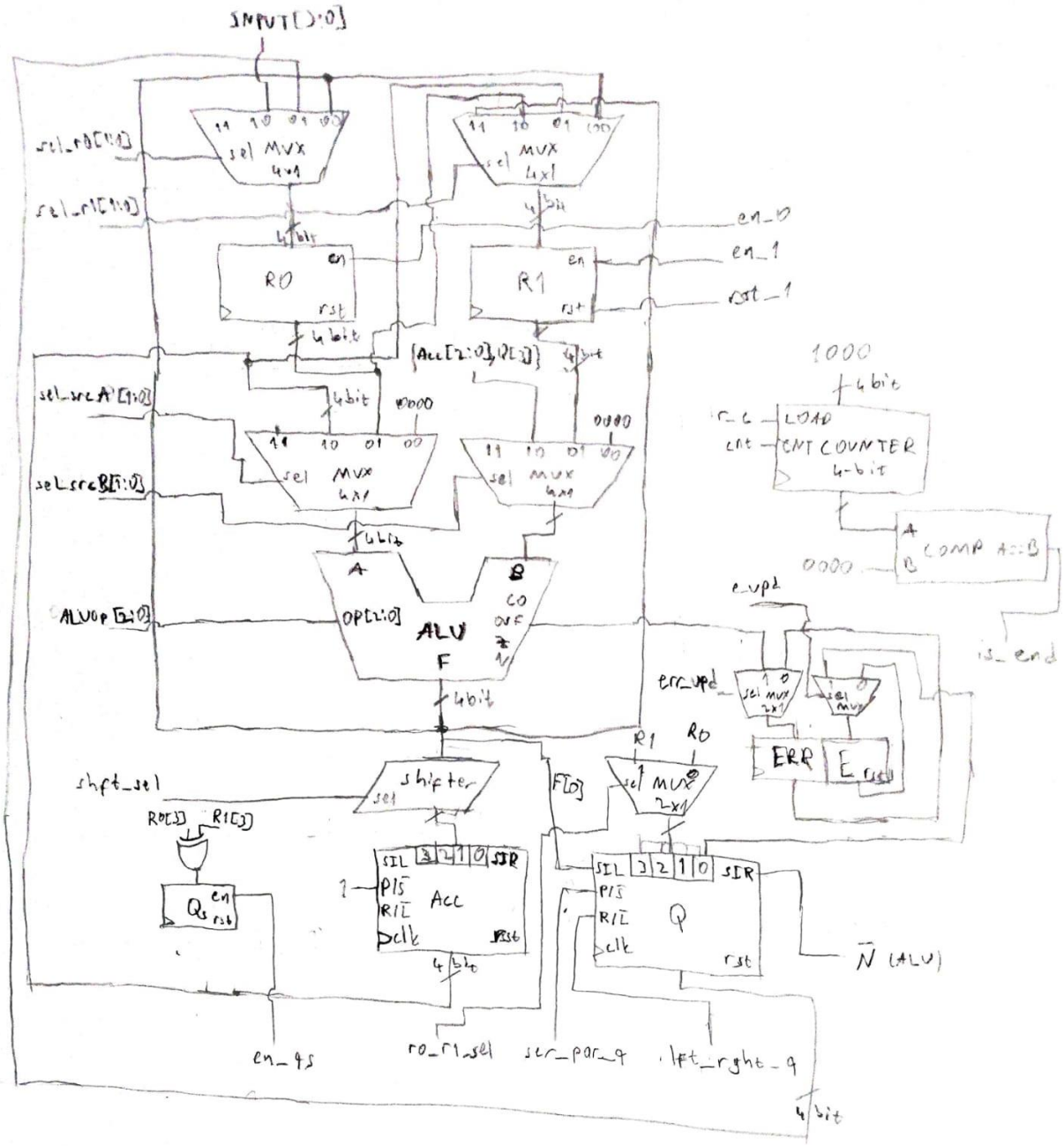


Figure 6: Shows the updated datapath for division algorithm.

Updates:

- Shifted Acc is connected to SrcB of the ALU throughout a multiplexer so that we can calculate Acc ←shl{Acc, Q} ± M.
- Shifter unit is added to the output of the ALU so that we can make arithmetic shift operation as we need in multiplication or we can take the ALU output directly as we need in division algorithm.
- A multiplexer is added to the parallel load of the Q register so that we can choose to load R0 (multiplication case) or R1 (division case).
- A multiplexer is added to the SIR of the Q register so that we can choose shift 1 or 0 depending on the negative flag of the ALU.

We need to output sign bits of the dividend (R1) and divisor (R0) to the controller so that we can convert them positive if they are negative initially. Also, we need to send the sign of Acc

register so that it can send necessary signals to the datapath depending on the sign as discussed above.

1.1.1.5. Datapath Design: Overall Design

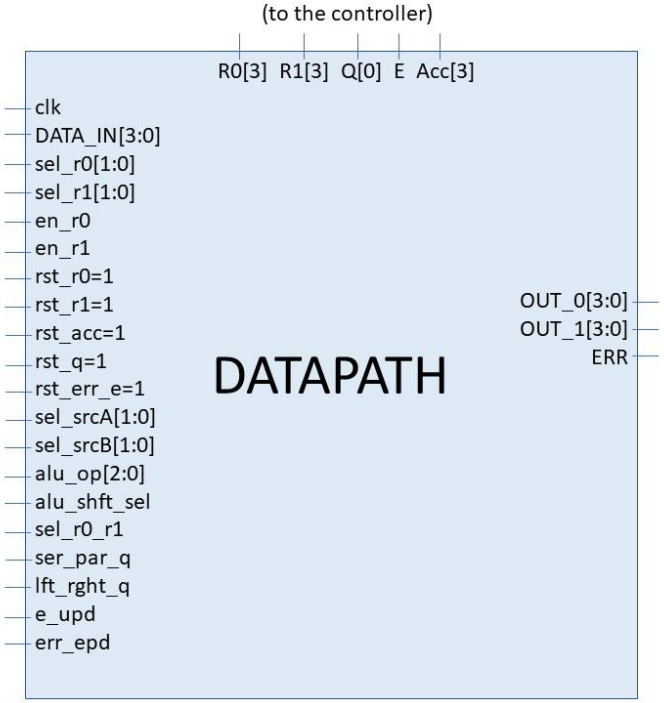


Figure 7: Blackbox diagram of the datapath architecture. The signals that are located at the left of the box are inputs to the datapath, others are outputs.

1.1.1.6. Datapath Design: Quartus II Implementation

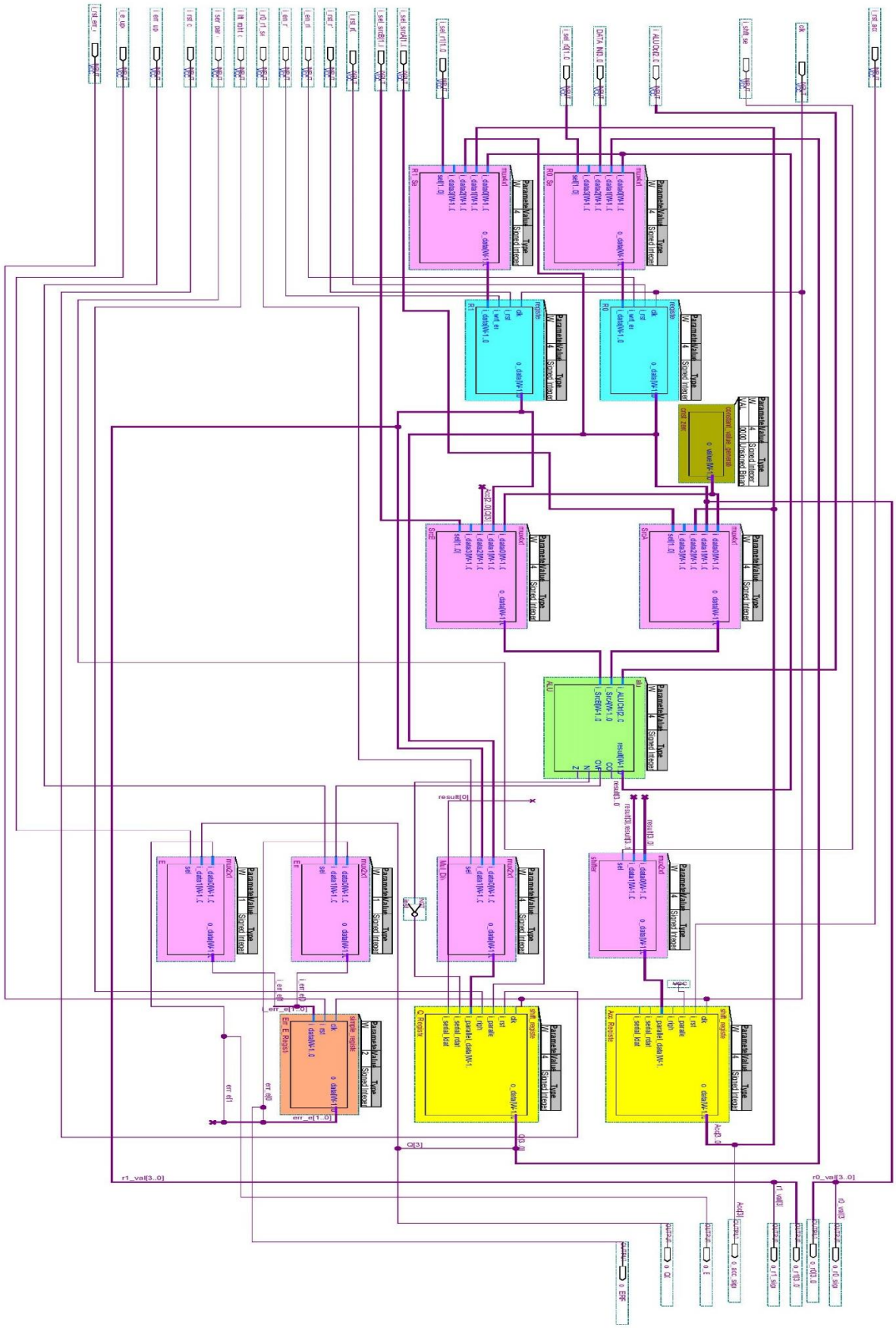


Figure 8: Shows the datapath design on Quartus II Schematic Editor.

1.1.2. Controller Design

1.1.2.1. Controller Design: Part 1

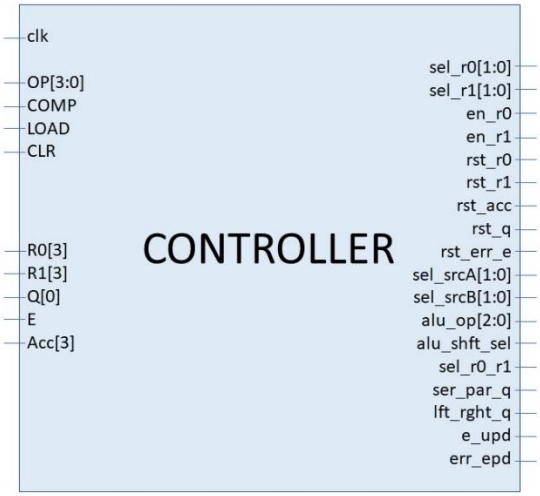


Figure 9: Black box diagram of the controller unit.

1.1.2.2. Controller Design: ASM of Box-1

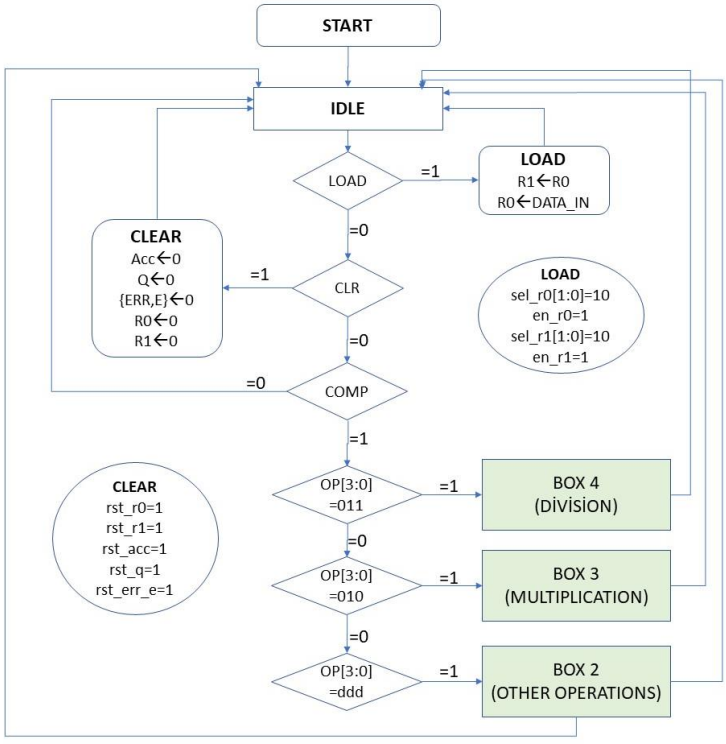


Figure 10: Shows the ASM chart of Box-1.

1.1.2.3. Controller Design: ASM of Box-2

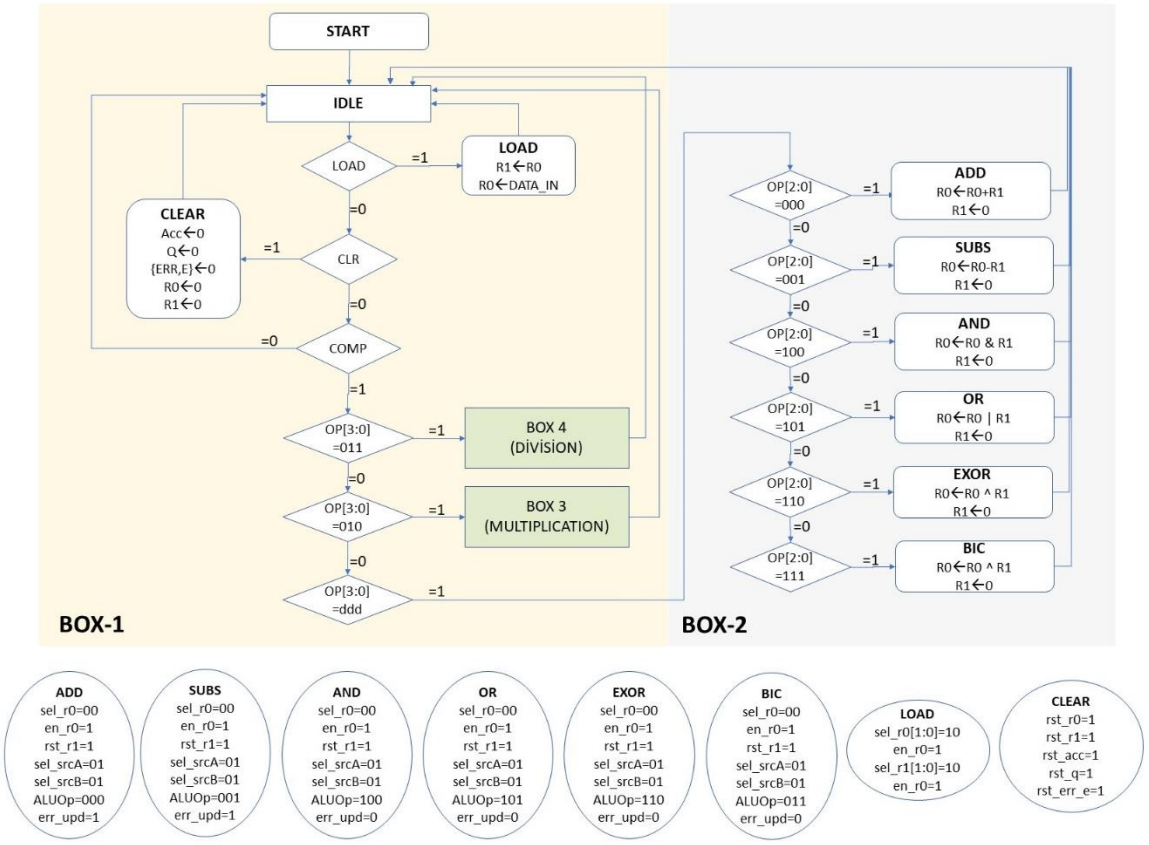


Figure 11: Shows the ASM chart of Box 1 and Box 2.

1.1.2.4. Controller Design: ASM of Box-3

As explained in *1.1.1.3*, I used Booth's multiplication algorithm, which does not make any assumptions neither on the signs of the multiplicand nor multiplier. The algorithm works well in negative or positive signed cases.

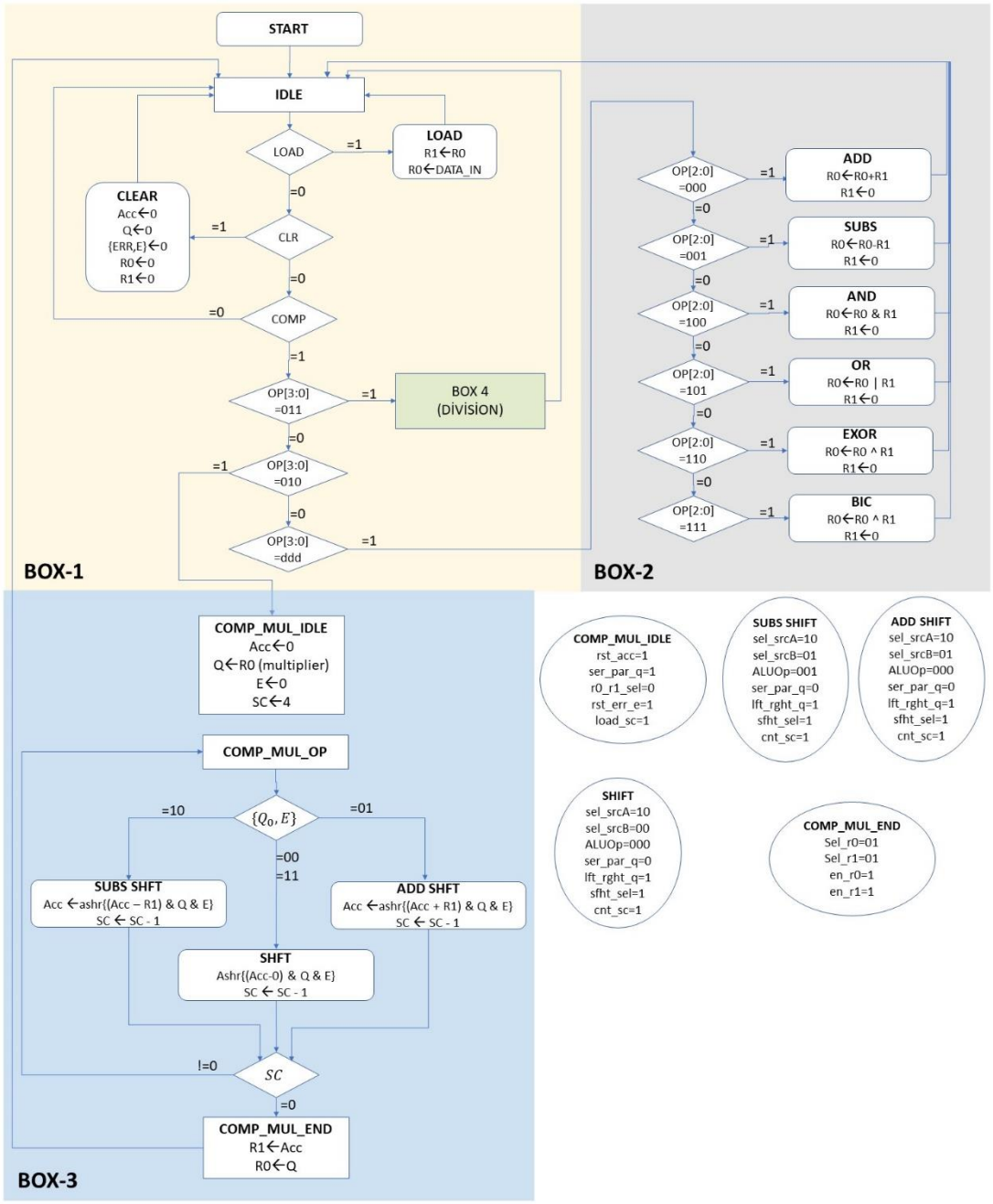


Figure 12: Shows the ASM chart of Box 1, Box 2 and Box 3.

1.1.2.5. Controller Design: ASM of Box-4

I used non-restoring division algorithm, which can handle only positive numbers. Therefore, my design determines and stores the sign of the result. Then, if R0 or R1 are negative, the algorithm inverts them positive before the division algorithm. After the division algorithm ends, depending on the sign of the result that we stored before the operation, sign changes done on the R0 and R1 registers.

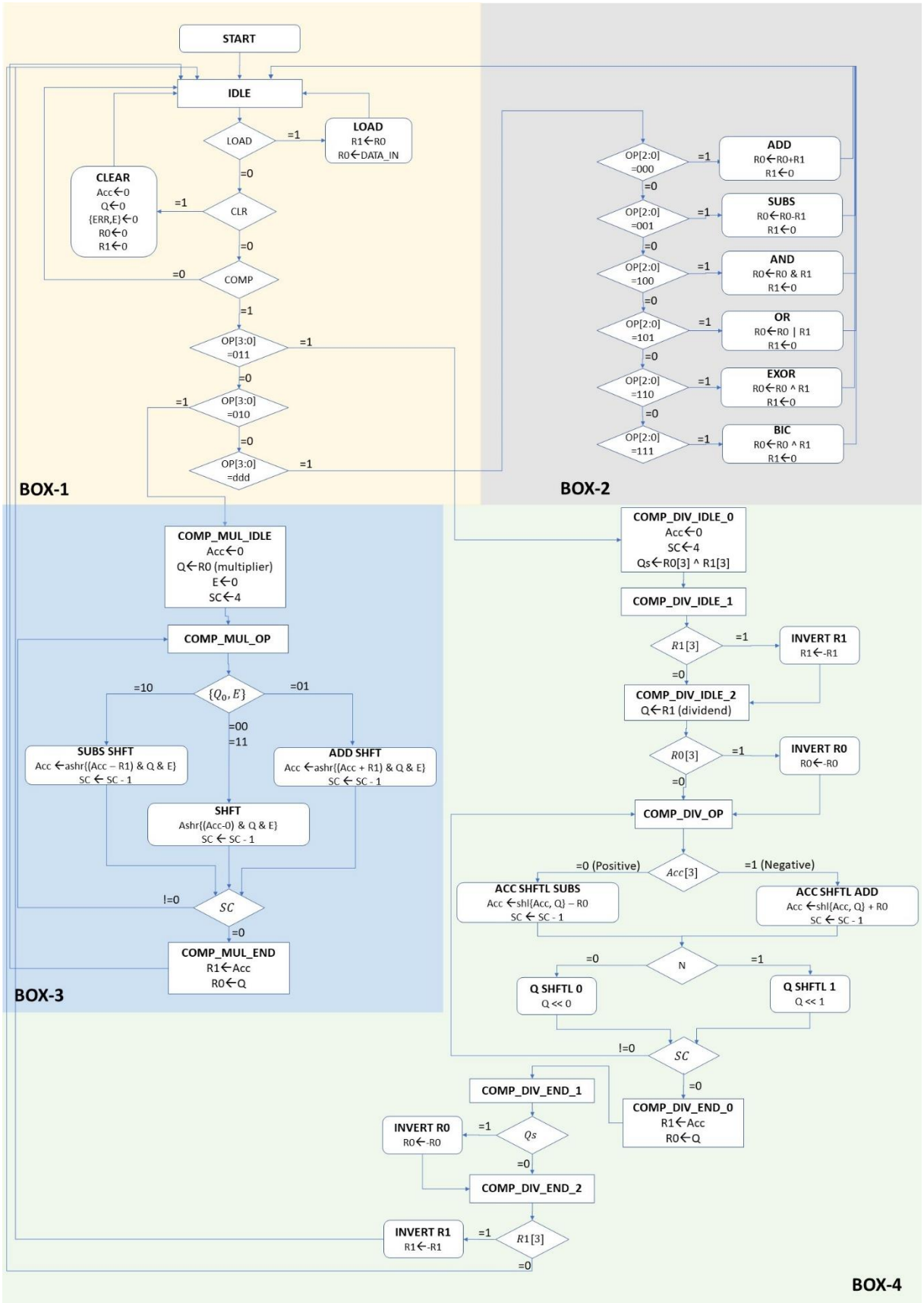


Figure 13: Shows the ASM chart of Box 1, Box 2 and Box 3.

1.1.2.6. Controller Design: Cycle Analysis

Table 3: Shows the clock cycle analysis of the ALP operations.

OP [2:0]	ALP	Clock Cycles
000	Addition	1
001	Subtraction	1
010	Multiplication	6
011	Division	10
100	AND	1
101	OR	1
110	EXOR	1
111	Bit Clear	1

1.1.2.7. Controller Design: State Analysis

Table 4: Shows the state analysis of the controller design.

Box Number	States	State Number
1	IDLE	1
2	-	0
3	COMP_MUL_IDLE, COMP_MUL_OP, COMP_MUL_END (Additionally, we have a state counter)	3
4	COMP_DIV_IDLE_0, COMP_DIV_IDLE_1, COMP_DIV_IDLE_2, COMP_DIV_OP, COMP_DIV_END_0, COMP_DIV_END_1, COMP_DIV_END_2, (Additionally, we have a state counter)	7

2. Experimental Work

2.1. Take and Store The 2's Complement Verification

Since we do not have an instruction to take and store 2's complement directly on ALP, I write a testbench for datapath and give necessary sequence of signals for this part.
Testbench file: datapath_tb.v.

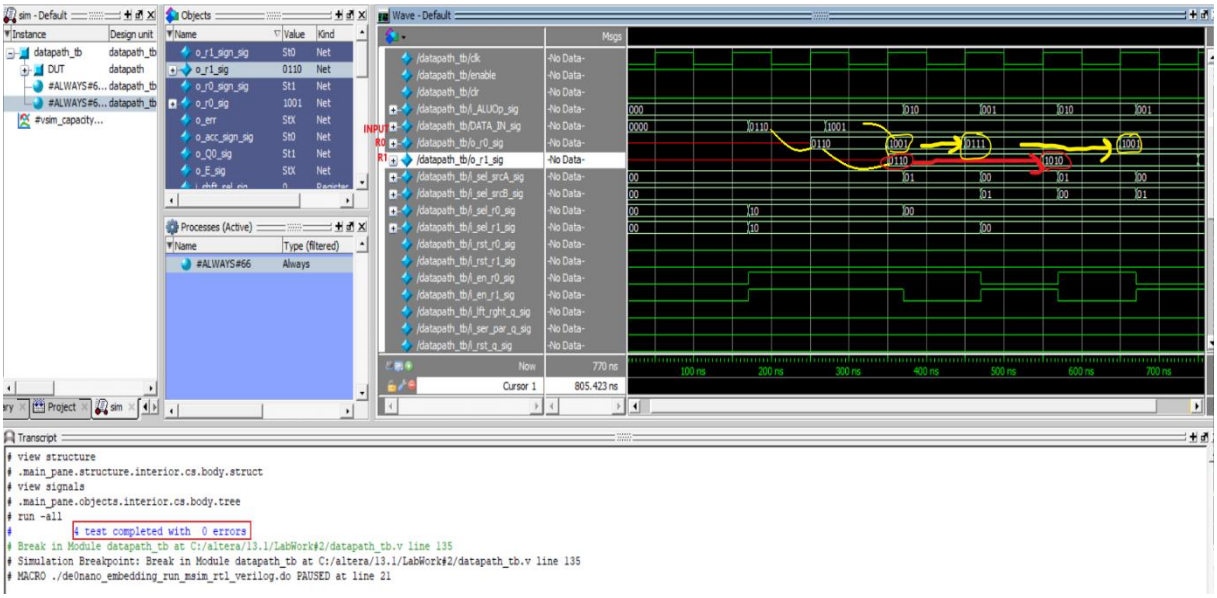
Table 5: Shows the main block of the datapath testbench. Taking and storing 2's comlement microoperation sequence is shown.

```
    #17
    // LOAD OP
    // R0 <- 4'b0110;
    DATA_IN_sig <= 4'b0110;
    i_sel_r0_sig[1:0] <= 2'b10;
    i_en_r0_sig <= 1;
    i_sel_r1_sig[1:0] <= 2'b10;
    i_en_r1_sig <= 1;
    #10
    // LOAD OP
```

```
    // R0 <- 4'b1001; R1 <- R0 (4'b0110)
    DATA_IN_sig <= 4'b1001;
    i_sel_r0_sig[1:0] <= 2'b10;
    i_en_r0_sig <= 1;
    i_sel_r1_sig[1:0] <= 2'b10;
    i_en_r1_sig <= 1;
    #10
    // R0 <- (-R0)
    i_sel_r0_sig[1:0]=00;
    i_en_r0_sig=1;
    i_en_r1_sig=0;
    i_sel_srcA_sig=01;
    i_sel_srcB_sig=00;
    i_ALUOp_sig=010;
    #10
    if(o_r0_sig != 4'b0111) begin
        $display("ERROR for R0. Expected: %b; Output: %b", 4'b0111,
o_r0_sig[3:0]);
        errors = errors + 1;
    end
    // R1 <- (-R1)
    i_sel_r1_sig=00;
    i_en_r0_sig=0;
    i_en_r1_sig=1;
    i_sel_srcA_sig=00;
    i_sel_srcB_sig=01;
    i_ALUOp_sig=001;
    #10;
    if(o_r1_sig != 4'b1010) begin
        $display("ERROR for R0. Expected: %b; Output: %b", 4'b1010,
o_r1_sig[3:0]);
        errors = errors + 1;
    end
    // R0 <- (-R0)
    i_sel_r0_sig[1:0]=00;
    i_en_r0_sig=1;
    i_en_r1_sig=0;
    i_sel_srcA_sig=01;
    i_sel_srcB_sig=00;
    i_ALUOp_sig=010;

    #10;
    if(o_r0_sig != 4'b1001) begin
        $display("ERROR for R0. Expected: %b; Output: %b", 4'b1001,
o_r0_sig[3:0]);
        errors = errors + 1;
    end
    // R1 <- (-R1)
    i_sel_r1_sig=00;
    i_en_r0_sig=0;
    i_en_r1_sig=1;
    i_sel_srcA_sig=00;
    i_sel_srcB_sig=01;
    i_ALUOp_sig=001;
    #10;
    if(o_r1_sig != 4'b0110) begin
        $display("ERROR for R0. Expected: %b; Output: %b", 4'b0110,
o_r0_sig[3:0]);
        errors = errors + 1;
    end
```

```
    $display("%d test completed with %d errors", 4, errors);
    $stop;
```



2.2. Multiplication and Division Excluded Verification

Table 6: Shows the instantination of ALP for experimental part.

```
ALP ALP_inst
(
    .clk(~KEY[0]) , // input  clk_sig
    .i_OP(SW[2:0]) ,    // input [2:0] i_OP_sig
    .i_DATA_IN(GPIO_0[3:0]) ,   // input [3:0] i_DATA_IN_sig
    .i_COMP(GPIO_0[5]) ,    // input  i_COMP_sig
    .i_LOAD(GPIO_0[4]) ,    // input  i_LOAD_sig
    .i_CLR(~KEY[1]) ,   // input  i_CLR_sig
    .ERR(GPIO_0[6]) ,   // output  ERR_sig
    .o_R0(LED[3:0]) ,   // output [3:0] o_R0_sig
    .o_R1(LED[7:4])     // output [3:0] o_R1_sig
);
```

2.2.1. Addition and Subtraction Verification

Simulation file: alu_add_sub.vwf

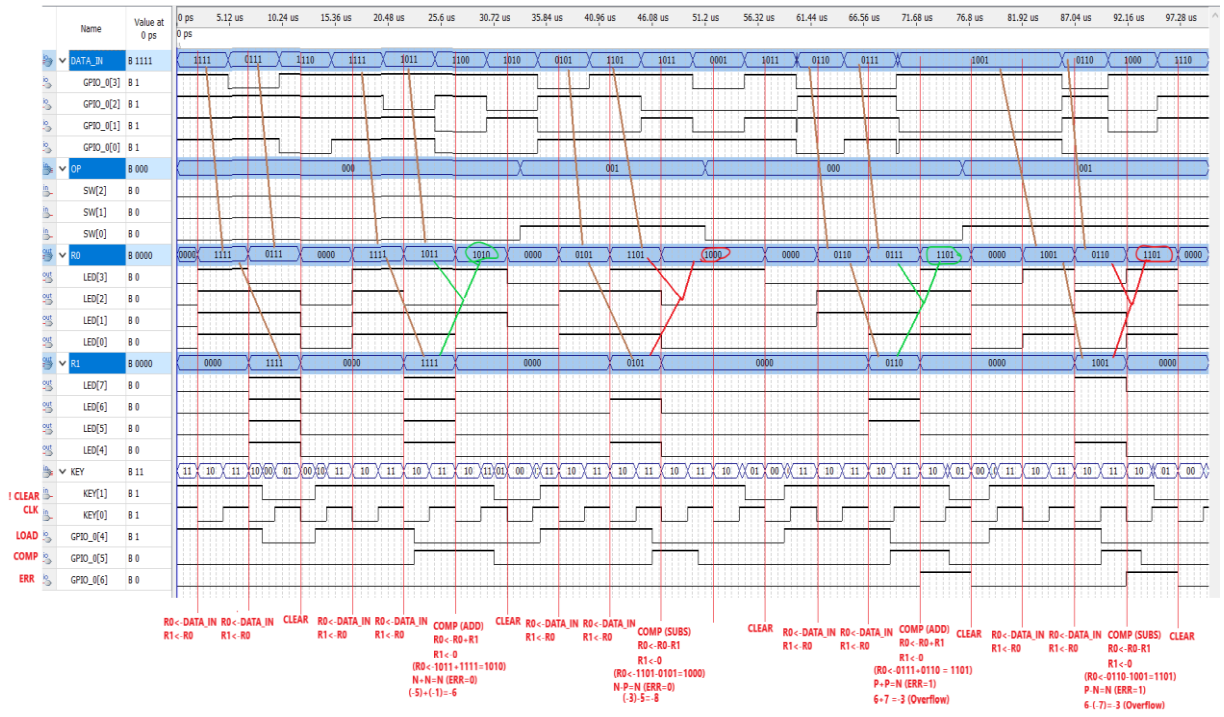


Figure 14: Shows the test result for addition and subtraction operations with and without overflow cases.

2.2.2. Logic Operations Verification

Simulation file: logic.vwf

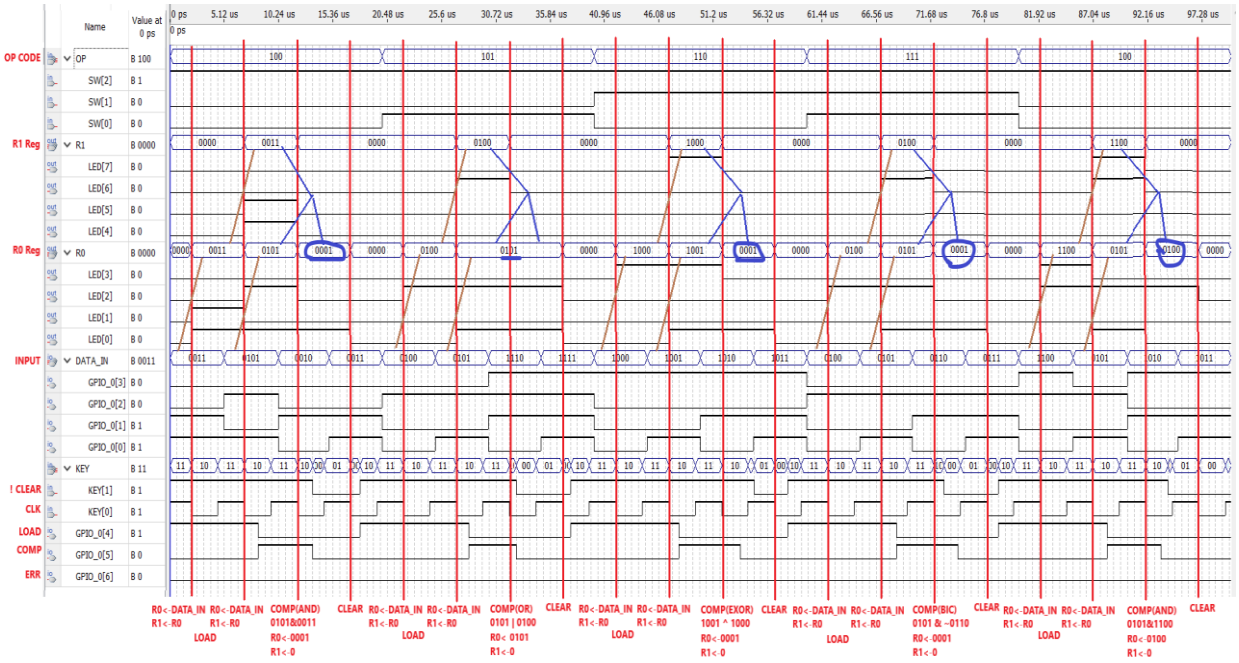


Figure 15: Shows the test result for logic operations.

2.3. Multiplication Verification

Simulation file: ALP_tb.v
Testvector file: alp_test.tv

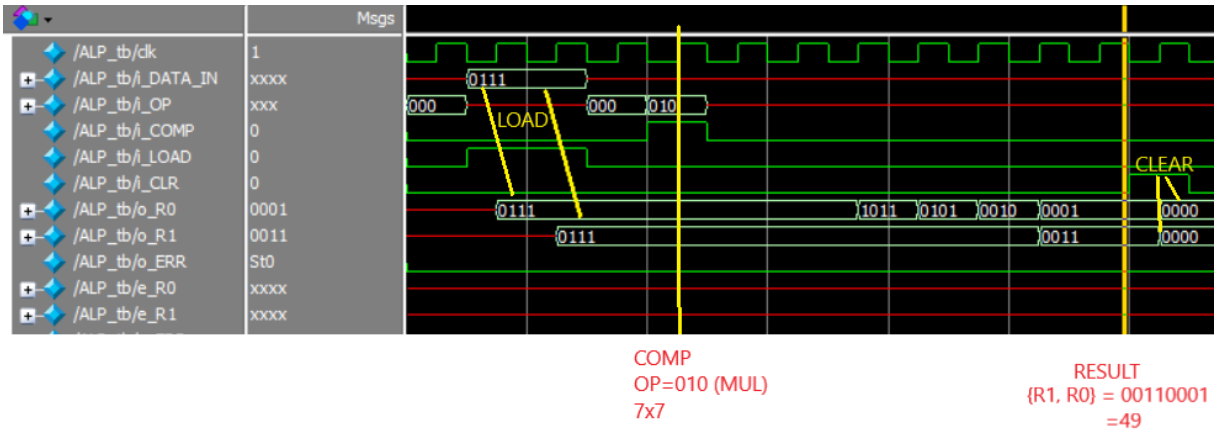


Figure 16: Shows multiplication test result for PxP case.

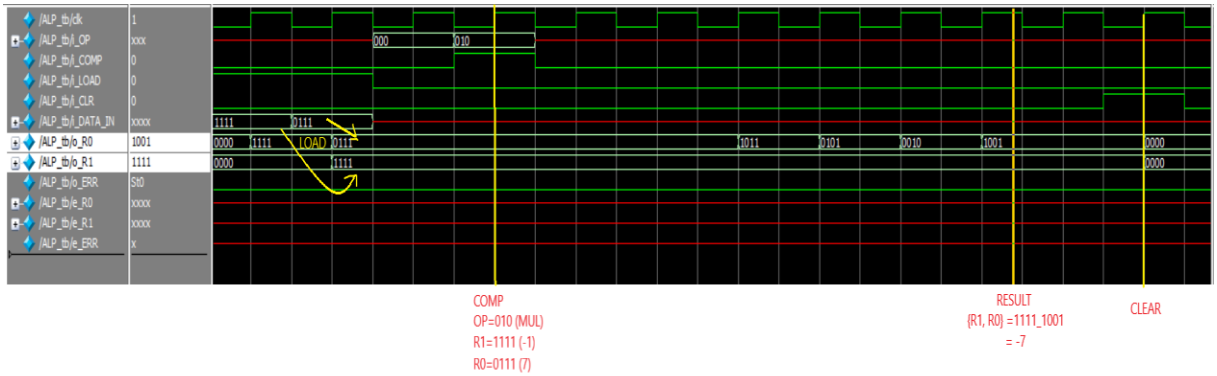


Figure 17:Shows multiplication test result for PxN case.

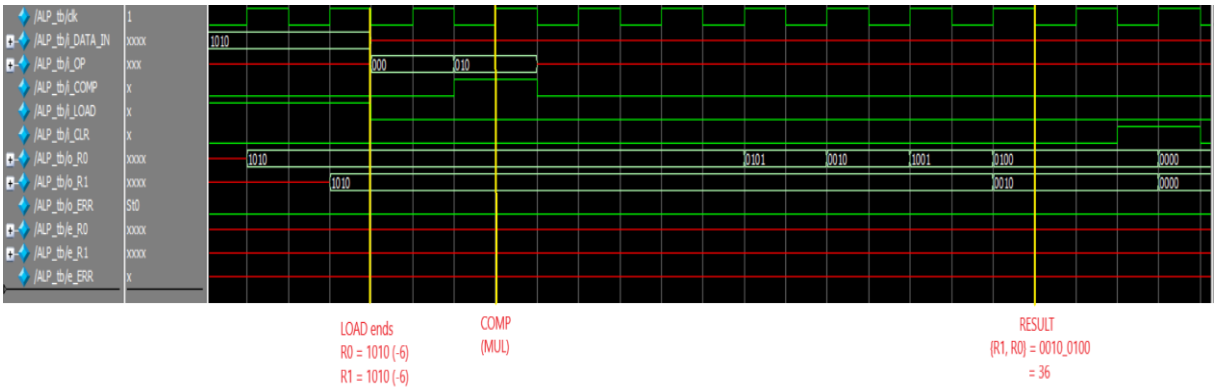


Figure 18: Shows multiplication test result for NxN case.