Ben Linam
Dylan Hall
EEL4712L
28 Sept. 2018
Lab 4

EEL4712L

Lab 4 – ALU Processing

Prelab

Preliminary Setup

Overview

*For this lab, we will be designing an **ALU** (**Arithmetic Logic Unit**). A device which takes in an input and performs a specified operation on that input based on a user defined protocol. We will be using **Quartus Prime** to design the **ALU** and implement its logic. Then, we will be using our **Altera DE1-SoC FPGA** board to present our design and confirm its accuracy.*

Materials

- An Altera DE1-SoC kit
- A computer with Quartus Prime

Procedure

*We will be using **ENTITY** designs in this laboratory. We will be designing a **ALU.** Once designed, we will be using the **Pin Assignment** utility provided by **Quartus Prime** to determine the **Software → Hardware** implementations.*

*Such that, we will be setting the logic inputs/outputs from our **VHDL** designs to physical pins on our **Altera DE1-SoC** boards to test and simulate the designs.*

***ALU** (Arithmetic Logic Unit)*

*For the purposes of this lab, we will be implementing a **4-bit ALU** which has access to a **3-bit op-code.** This means there will be **EIGHT** possible operations our **ALU** can perform.*

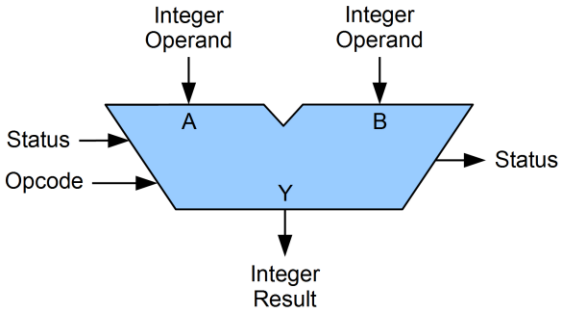
- Background:

Description:

*An **ALU** performs many operations, such as **logical** operations, containing, but not limited to: **AND, OR, NAND, NOR, XNOR, NOT** and **BIT-SHIFTS.** As well as **arithmetic** operations, such as: **ADDITION, SUBTRACTION, DIVISION** and **MUTLIPLICATION.***

*An **ALU** is a device used by a computer's **CPU** (**central processing unit**). The **ALU(s)** accessible by a **CPU** handle all its memory/task operations. When a command operation is tasked, the **ALU** will parse this data along with a **control bit(s)** to perform the necessary operation on the input data. A **control bit** is simply an input bit which holds encoded information for the **ALU** to decode into the appropriate operation. Because this **control bit** operates similarly to an **encoder**, this means that there is a relation to how many **control bit's** an **ALU** is designed to read, as well as how many operations the **ALU** can perform. Such that, if one implements a **n-bit** input **control bit**, one will only be able to implement 2^n operations.*

*Though, the expense, power consumption and heat dispersion of a **CPU** is directly proportional to the onboard **ALU's**. Therefore, as an **ALU** expands in complexity, as does the requirements and demands of the **CPU.***



*From this image, one can see that this **ALU** takes in two integers (**A** and **B**), in which it will perform a specified operation on, to produce an output **(Y)**. This operation is designated by the **Opcode** input bit. **Status** is just a general-purpose bit used to simplify scalability. i.e. cascading one **ALU** into another to perform larger operations on larger input data.*

Figure 1. Basic **ALU** block

- Logic/Truth Tables –

*Below you will see the logic truth-table for a **4-bit ALU** with a **3-bit Op-code**.*

The operations present in the truth table will be the ones being implemented in this laboratory.

Key (Table 1):

- Rows this color are designed as valid inputs.

- Rows this color are designed as invalid inputs.

- Cells this color represents valid outputs

4-Bit ALU w/ 3-Bit Op-Code											
Term	***Op-Code***				***Input***		***Status***				***Output***
	Code			*Description*	***A***	***B***	*Z*	*N*	C_{out}	*V*	*R*
0	*0*	*0*	*0*	***AND***	*0000*	*1111*	*1*	*0*	*0*	*0*	*0000*
1	*0*	*0*	*1*	***OR***	*0001*	*1110*	*0*	*1*	*0*	*0*	*1111*
2	*0*	*1*	*0*	***COMPLIMENT***	*0010*	*1101*	*0*	*1*	*0*	*0*	*1101*
3	*0*	*1*	*1*	***ADDITION***	*0011*	*1100*	*0*	*1*	*0*	*0*	*1111*
4	*1*	*0*	*0*	***INCREMENT***	*0100*	*1011*	*0*	*0*	*0*	*0*	*0101*
5	*1*	*0*	*1*	***DECREMENT***	*0101*	*1010*	*0*	*0*	*0*	*0*	*0100*
6	*1*	*1*	*0*	***SUBTRACTION***	*0110*	*1001*	*0*	*0*	*1*	*1*	*0011*
7	*1*	*1*	*1*	***NEGATATION***	*0111*	*1000*	*0*	*1*	*0*	*0*	*1000*
8	*0*	*0*	*0*	***AND***	*1000*	*0111*	*1*	*0*	*0*	*0*	*0000*
9	*0*	*0*	*1*	***OR***	*1001*	*0110*	*0*	*1*	*0*	*0*	*1111*
10	*0*	*1*	*0*	***COMPLIMENT***	*1010*	*0101*	*0*	*0*	*0*	*0*	*0101*
11	*0*	*1*	*1*	***ADDITION***	*1011*	*0100*	*0*	*1*	*0*	*0*	*1111*
12	*1*	*0*	*0*	***INCREMENT***	*1100*	*0011*	*0*	*1*	*0*	*0*	*1101*
13	*1*	*0*	*1*	***DECREMENT***	*1101*	*0010*	*0*	*1*	*0*	*0*	*1100*
14	*1*	*1*	*0*	***SUBTRACTION***	*1110*	*0001*	*0*	*1*	*1*	*1*	*1101*
15	*1*	*1*	*1*	***NEGATATION***	*1111*	*0000*	*1*	*0*	*0*	*0*	*0000*

Table 1.** Truth table for a **4-bit ALU

- Logic/Truth Tables Cont. -

*From above, the **FLAG** bits (**status),** function as follow;*

- **Z (zero flag)***: is only set to **'1'** whenever the output **R** is **"0000."***
- **N (negative flag)***: is only set to **'1'** whenever the output **R** is **NEGATIVE** (MSB is a zero).*
- $\mathbf{C_{out}}$ **(carry flag):** *is only set to **'1'** whenever an addition produces a **carry-out** or a subtraction produces a **borrow-out.***
- **V (overflow flag)**: *is only set to **'1'** whenever there is an overflow.*
    - *Two positive numbers add to make a negative.*
    - *Two negative numbers add to make a positive.*
    - *A positive number is subtracted from a negative and the result is positive.*
    - *A negative number is subtracted from a positive number and the result is negative.*

*The output is based on the **two 4-bit** input vectors **A** and **B,** as well as the **3-bit** input vector **op-code.** Each row has a specific operation which is set to perform on the two vectors **A** and **B.** The vectors **A** and **B** were not randomly allocated, as noted, **A** counts in binary from **zero** to **fifteen.** Whilst **B** counts down in binary from **fifteen** to **zero.** This functionality for the truth table may be changed in the future to provide a better range of test vectors.*

***Note:** these input vectors **WILL NOT** be the same ones tested in the simulation of our **ENTITY** design.*

- Realization/Simulation –

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5   ENTITY alu_4bit IS
6      GENERIC(
7      NMEMB : INTEGER := 4;
8      MSB   : INTEGER := 3);
9
10     PORT(
11     -- Inputs
12     clk,clr    : IN STD_LOGIC;
13     A,B        : IN STD_LOGIC_VECTOR(NMEMB-1 DOWNTO 0);
14     C          : IN STD_LOGIC_VECTOR(NMEMB-2 DOWNTO 0);
15
16     -- Outputs
17     Z,N,Cout,V : OUT STD_LOGIC;
18     R          : OUT STD_LOGIC_VECTOR(NMEMB-1 DOWNTO 0) );
19  END alu_4bit;
20
21  ARCHITECTURE logic OF alu_4bit IS
22     SIGNAL F: STD_LOGIC_VECTOR(NMEMB DOWNTO 0); -- temp variable used to manipulate the output
23  BEGIN
24     PROCESS(F,C,clr,clk)
25     BEGIN
26     IF(clr = '1') THEN
27        F <= (OTHERS => '0');
28     ELSIF(RISING_EDGE(clk)) THEN
29        CASE C IS
30           WHEN "000" =>
31              F <= '0' & A AND B;
32           WHEN "001" =>
33              F <= '0' & A OR B;
34           WHEN "010" =>
35              F <= '0' & NOT A;
36           WHEN "011" =>
37              F <= '0' & A + B;
38           WHEN "100" =>
39              F <= '0' & A + 1;
40           WHEN "101" =>
41              F <= '0' & A - 1;
42           WHEN "110" =>
43              F <= '0' & A - B;
44           WHEN "111" =>
45              F <= '0' & NOT A(MSB) & A(MSB-1 DOWNTO 0);
46           WHEN OTHERS =>
47              F <= (OTHERS => '0');
48        END CASE;
49     END IF;
50     END PROCESS;
51
52     R <= F(NMEMB-1 DOWNTO 0);                                  ---------- FLAGS ----------
53     Z <= '1' WHEN F = ("0000") ELSE '0';  -- Z = '1' if F is "0000"
54     N <= F(MSB);                          -- N = '1' if F's MSB = '1' (F is negative) Otherwise, N = '0'
55     Cout <= F(NMEMB);                     -- Cout = '1' when R(MSB) = '1', hence, there was a carry out
56     V <= ( (A(MSB) NOR B(MSB)) AND F(MSB) ) OR ( (A(MSB) AND B(MSB)) AND NOT F(MSB) ) WHEN C = ("011")
57     ELSE ( (A(MSB) AND NOT B(MSB)) AND NOT F(MSB) ) OR ( (NOT A(MSB) AND B(MSB)) AND F(MSB) ) WHEN C = ("110");
58
59  END logic;
```

Figure 2. Screen capture displaying the **Quartus Prime logic design** for the **4-Bit ALU** design

- Code Logic -

Following Figure 2.

*The **VHDL** code follows the basic format. We include the libraries, we declare an **ENTITY (alu_4bit)** then we declare that entities **PORTS** (in/out).*

*For the main functionality of the **ALU** we used separate case statements for each of the intended operations. Each case can be noted in **Table 1**. We used two **GENERICS**, **NMEMB** and **MSB. NMEMB** is used to represent the number of bits in each vector, where **MSB** is the most significant bit. This allows us to have quick and readable access to the most important bits of our input and output vectors. For our inputs, we used **clk** and **clr**, which are our asynchronous inputs for our process sensitivity list. Then we have input vectors **A, B** and **C.** The latter two are the actual **4-bit input** vectors used to perform the operation on, were the vector **C** is the **3-bit Op-Code** vector which designates the operation to perform on **A** and **B.** For determining the resulting output, we used a **N+1** vector, **F**. This was the easier way to determine if an operation creates a **carry-out.** Instead of creating some expression to determine if a **carry-out** is created, we can easily just check the **MSB** of the output vector, this is noted in $\boldsymbol{C_{out}}$.*

*When determine the **FLAG (status) bits** we followed the expected output from the lab documentation **(See Page 4.)**.*

- Simulated Waveform Diagram –

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

1. ***AND Operation (*** $C = \text{"}\mathbf{000}\text{"}$)

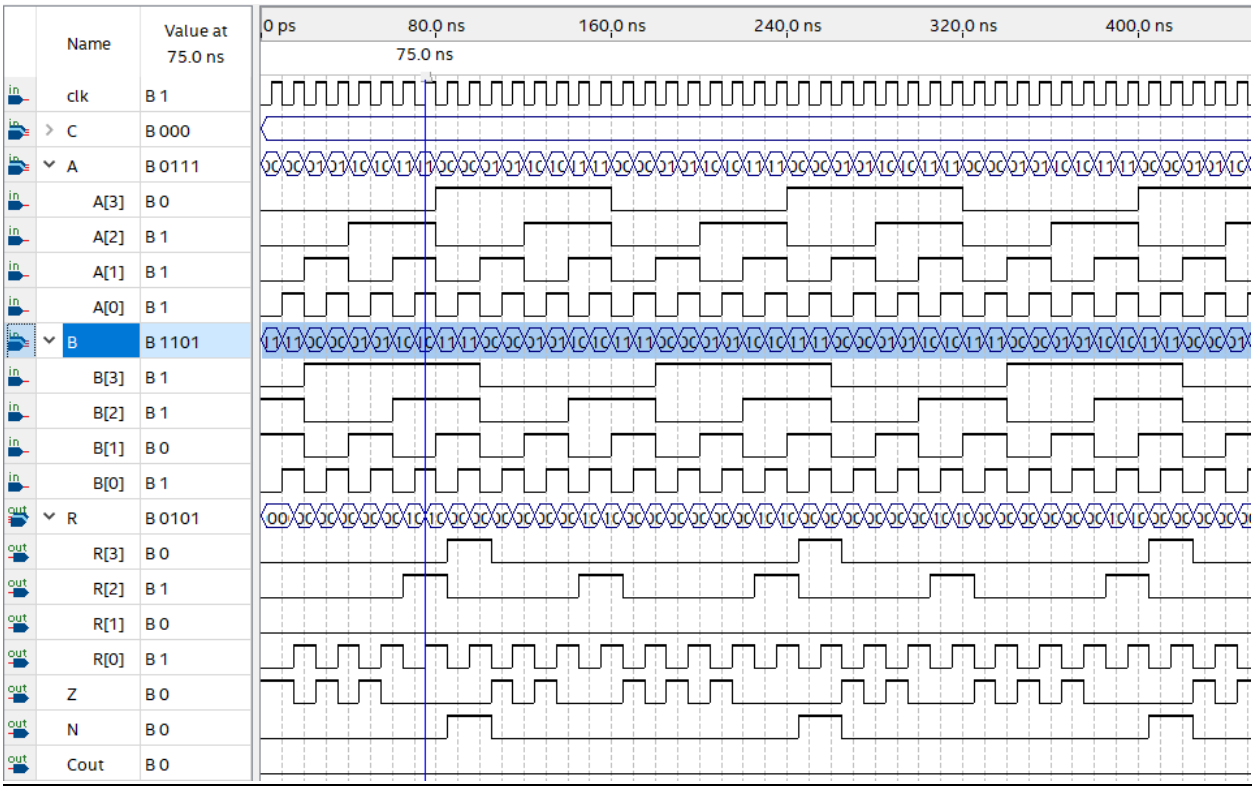


Figure 3. Screen capture displaying the simulation timing diagram of the logic design for **AND.**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

2. ***AND Operation (*** $C = \text{"}\mathbf{001}\text{"}$)

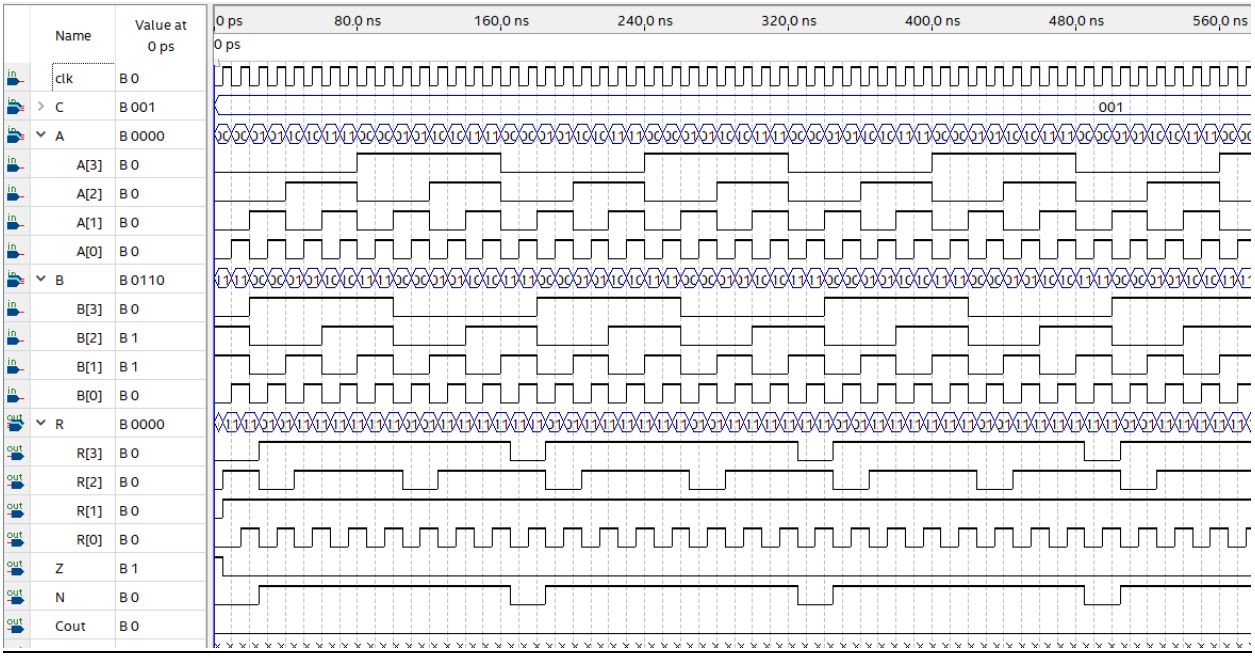


Figure 4. Screen capture displaying the simulation timing diagram of the logic design for **OR.**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

3. ***AND Operation (*** $C = \text{"}\mathbf{010}\text{"}$)

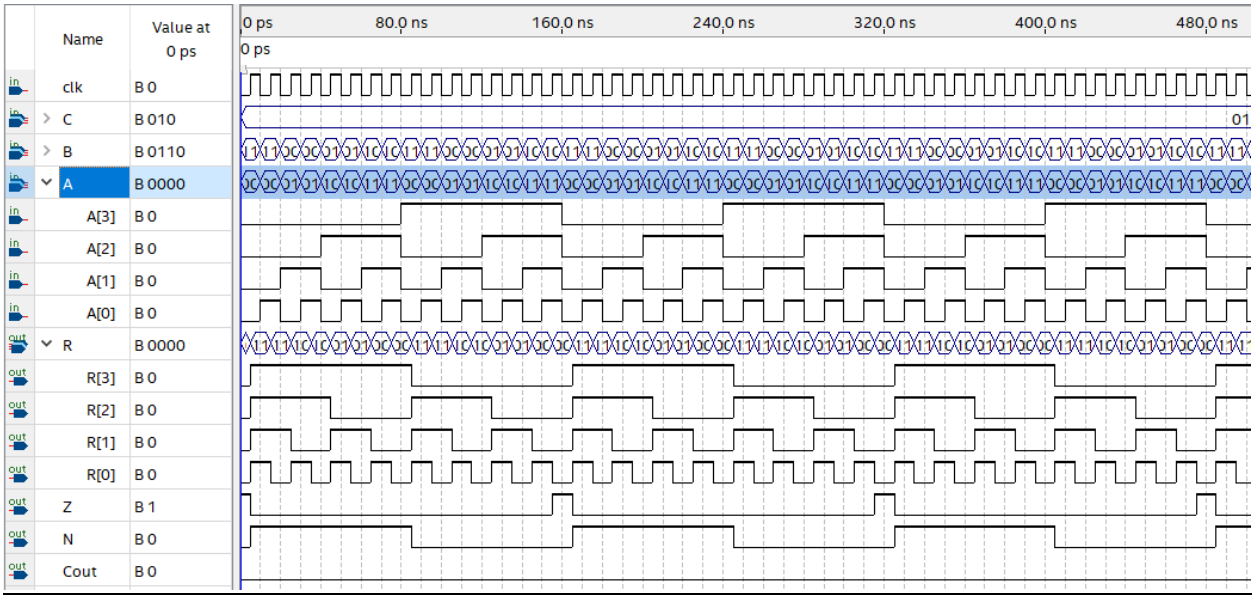


Figure 5. Screen capture displaying the simulation timing diagram of the logic design for **NOT.**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

4. ***AND Operation (*** $C = \text{"}\mathbf{011}\text{"}$)

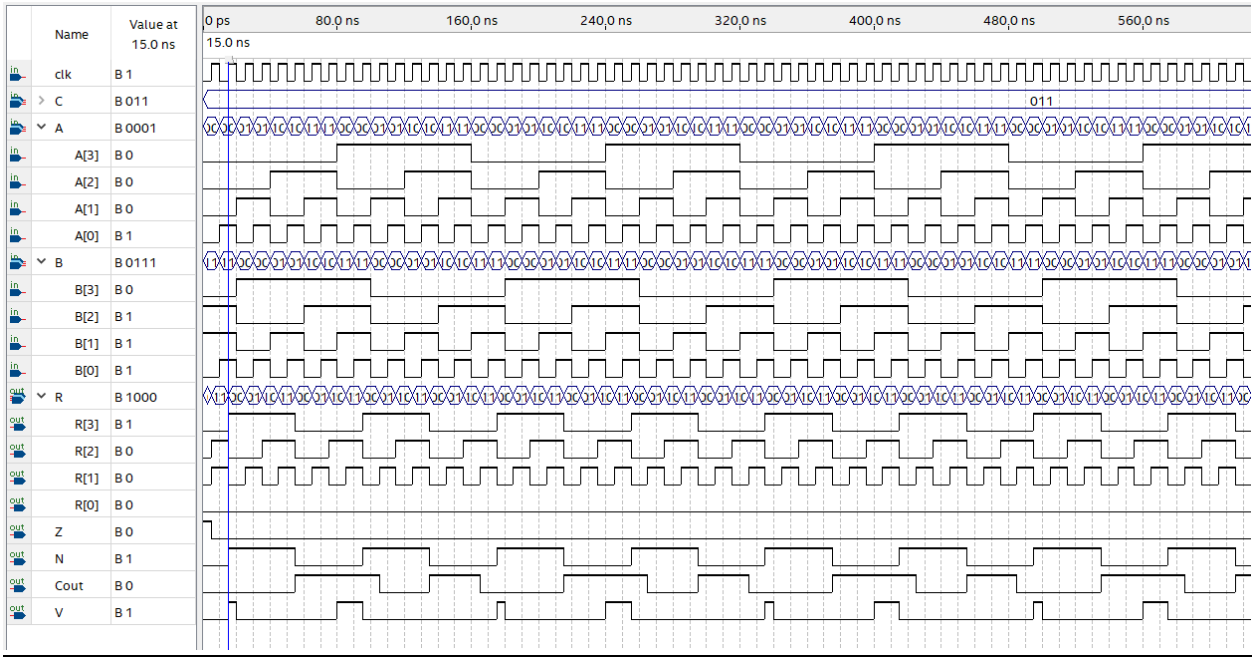


Figure 6. Screen capture displaying the simulation timing diagram of the logic design for **Addition.**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

5. ***AND Operation (*** $C = \text{"}\mathbf{100}\text{"}$)

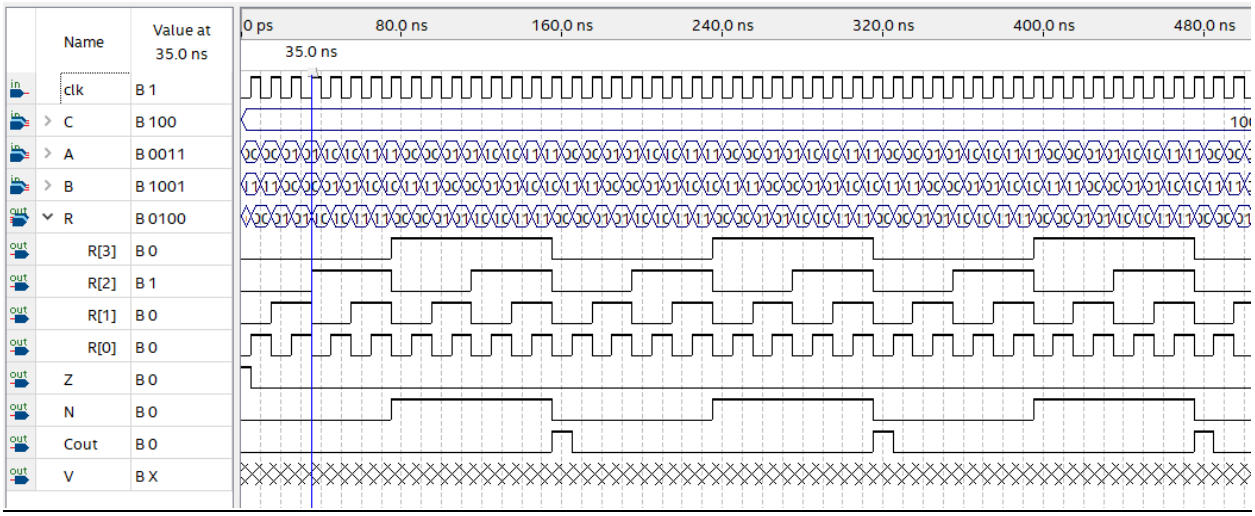


Figure 7. Screen capture displaying the simulation timing diagram of the logic design for **Increment**

6. ***AND Operation (*** $C = \text{"}\mathbf{101}\text{"}$)

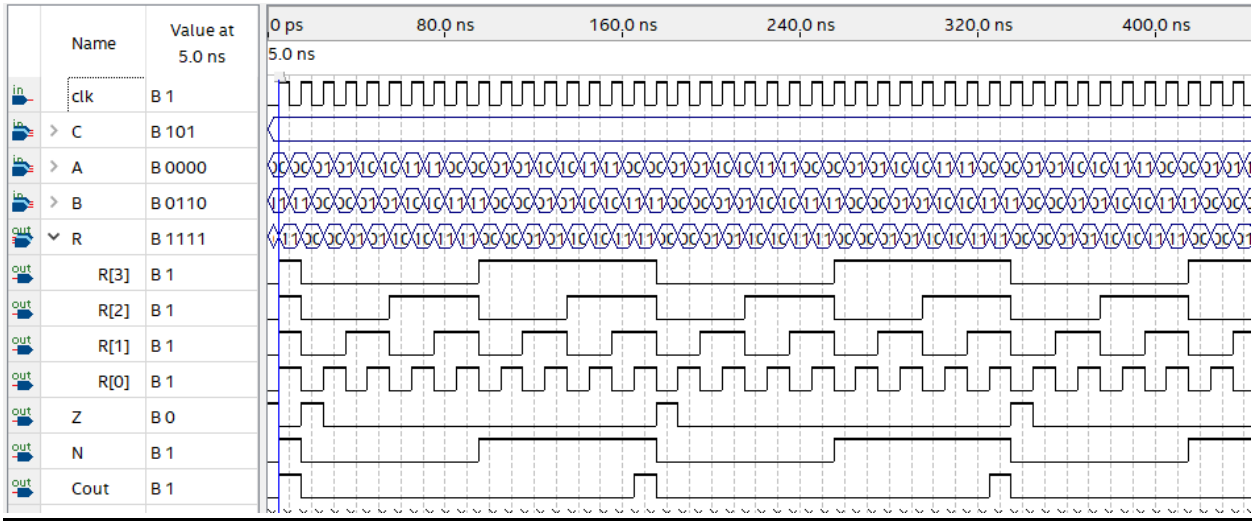


Figure 8. Screen capture displaying the simulation timing diagram of the logic design for **Decrement**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

7. ***AND Operation (*** $C = \text{"}\mathbf{110}\text{"}$)

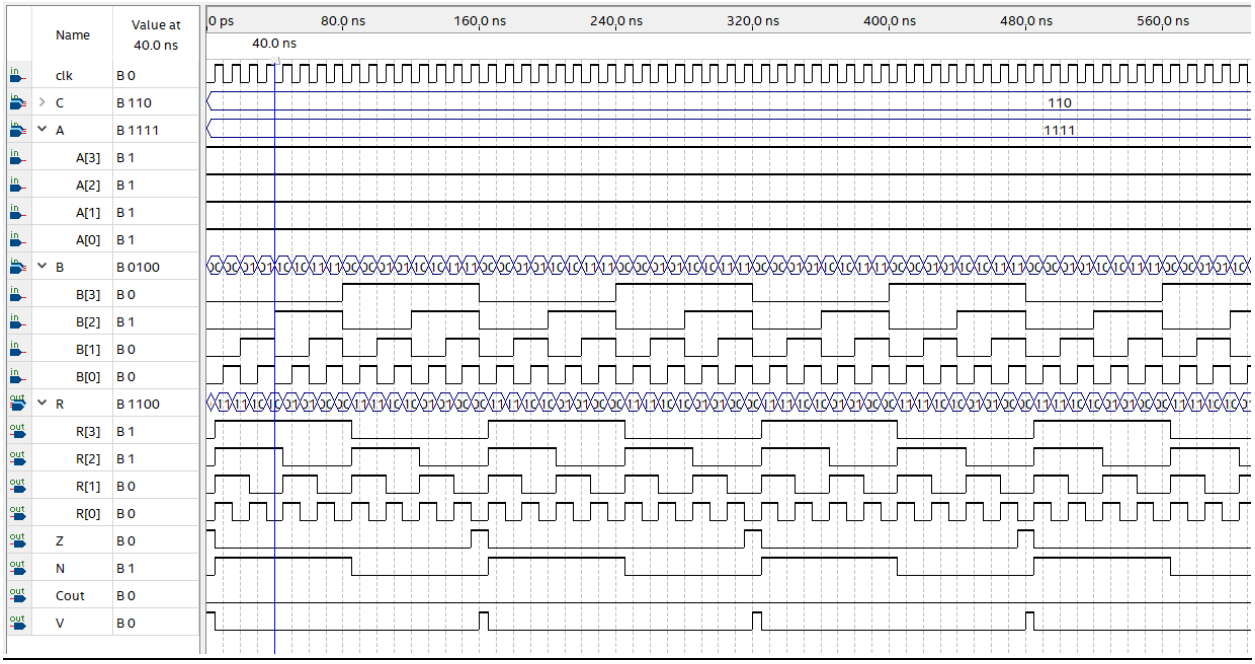


Figure 9. Screen capture displaying the simulation timing diagram of the logic design for **Subtraction.**

- Simulated Waveform Diagram Cont. -

*Below you will see the simulated timing/waveform diagrams of each **Op-Code** selection from the **VHDL** design above.*

8. ***AND Operation (*** $C = \text{"}\mathbf{111}\text{"}$)

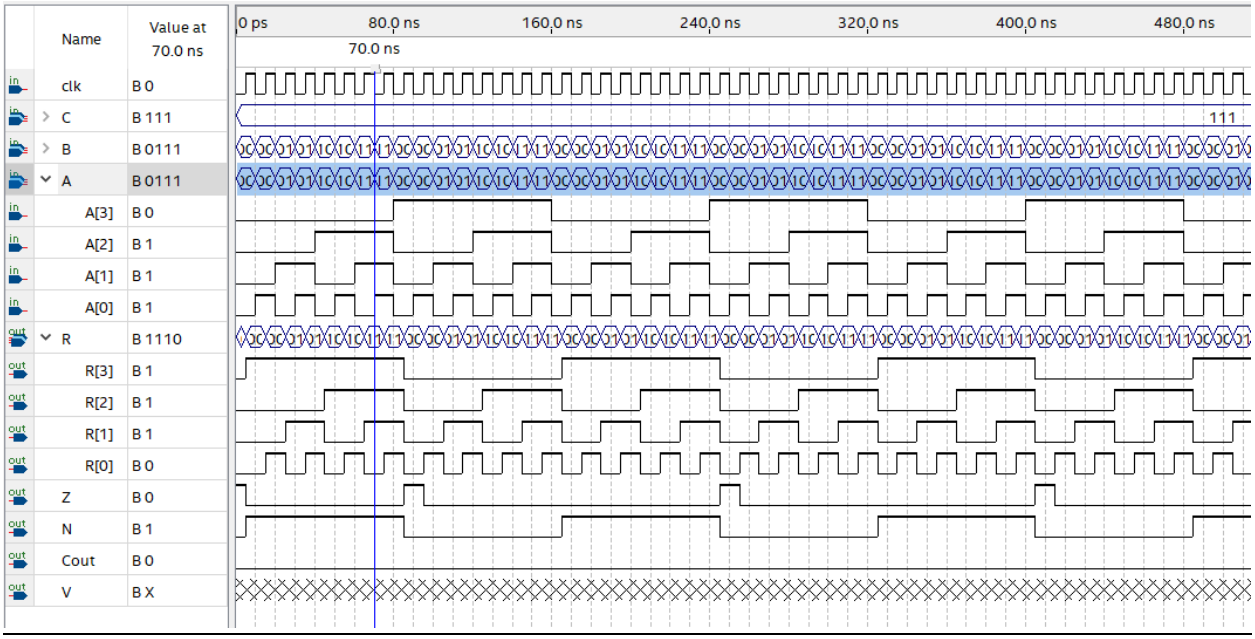


Figure 10. Screen capture displaying the simulation timing diagram of the logic design for **Negation.**

Conclusion:

*In summation, this lab was another good introduction to both **VHDL** and the **DE1-SoC** board. IT was also very enjoyable to implement the logic behind components in which we have learned previously. Being able to programmatically replicate the logic behind these components is something of beauty. It was a really good refresher on some basic combinational logic and a good opportunity to review previous concepts.*