ELEC 204 – DIGITAL SYSTEM DESIGN

EXPERIMENT #5 – PRELIMINARY WORK

Ahmet Hamdi Ünal (60167)

KOÇ UNIVERSITY – SPRING 2020

| REPORT DATE: *MAY 2, 2020*

Table of Contents

Question 5.1

Build knowledge on implementing sequential circuits with VHDL: Watch the video lecture 20 on finite state machine design. Listing 1 includes a sample implementation of the unsigned division FSM example in the tutorial video. Study, implement and modify it to perform 4-bit signed division in 2's complement system.

4-bit signed division ↔ Range = $\{-8,-7,-6,\dots-1,0,1,\dots\ 7\}$
The procedure to be followed is as follows:

1. Take 2 numbers in signed 2's complement system, dividend and divisor.
2. Now, consider the following 4 cases:
    i. Both positive (both MSBs are 0)
    ii. Both negative (both MSBs are 1) or
    iii. Dividend is negative (MSB=1) and divisor is positive (MSB=0).
    iv. Dividend is positive (MSB=1) and divisor is negative (MSB=0).
3. If both are positive, the operation is as follows:
    - First, introduce a counter (Q), which will be the quotient at the end.
    - Take 2's complement of divisor (A) and sum it with dividend (B) (in other words, subtract divisor from dividend). Assign this result to B. Increase Q by 1.
    - Check whether B is smaller than A. If $B \geq A$, then repeat the operation above. Else, division is finished.
    - When division is finished, Q is the quotient, and B is the remainder.
4. If both are negative, take 2's complement of both numbers and perform the operation for the case both positive. At the end, take 2's complement of the remainder to fix the sign issue.
5. If dividend is negative, and divisor is positive, take 2's complement of dividend one and perform the operation for the case both positive. At the end, take 2's complement of the quotient and remainder to fix the sign issue.
6. If dividend is positive, and divisor is negative, take 2's complement of dividend one and perform the operation for the case both positive. At the end, take 2's complement of the quotient to fix the sign issue.

Before presenting my code, let me explain the code given in the lab manual.

Sample Finite State Machine (FSM) VHDL code:

First, with the following declaration in the beginning of the code, the program implements the functionality of signed and unsigned numbers. We will use it in the upcoming parts of this code.

```
use IEEE.NUMERIC_STD.ALL;
```

Next, inputs and outputs are defined.

```
entity Lab5Tutorial is
      Generic (N : INTEGER:=50*10**6); --50*10ˆ6 Hz Clock
      Port ( MCLK : in STD_LOGIC;
             A : in STD_LOGIC_VECTOR(3 downto 0);
             B : in STD_LOGIC_VECTOR(3 downto 0);
             Start : in STD_LOGIC;
             RES : out STD_LOGIC_VECTOR(3 downto 0));
end Lab5Tutorial;
```

Here, MCLK refers to the main clock, whose frequency is N in the second line. A, B are two 4-bit input numbers. Start is the initiator of the design, and RES is the output to be shown. The interesting part is remainder is not shown, but calculated inside the code.

We then set intermediate signals and some constants in the architecture part.

```
architecture Behavioral of Lab5Tutorial is
signal CLK_DIV : STD_LOGIC;

signal X: unsigned(3 downto 0) := ”0000”; --Dividend variable
signal Y: unsigned(3 downto 0) := ”0001”; --Divisor variable
signal Q: unsigned(3 downto 0) := ”0000”; --Quotient
signal R: unsigned(3 downto 0) := ”0000”; --Remainder

--FSM with 3 states
constant init: STD_LOGIC_VECTOR(2 downto 0) := ”001”;
constant compute: STD_LOGIC_VECTOR(2 downto 0) := ”010”;
constant done: STD_LOGIC_VECTOR(2 downto 0) := ”100”;
--State variable with 3 flip-flops
signal State: STD_LOGIC_VECTOR(2 downto 0) := ”001”;
```

CLK_DIV is the Clock Divider that we have learnt and used in the previous lab. It will be implemented in the upcoming *process* part. X, Y, Q, R are declared, and assigned to some values which are not so important at this point. In the next sections, we X and Y will be used to represent A and B inputs as unsigned integers, whereas Q and R will be quotient and remainder at the end.

There are 3 states: INIT, COMPUTE and DONE. One Hot state assignments of them are made here. Also, a State intermediate signal is created, which is and will always be assigned to the current state of the system.

Next, the operations begin. RES is assigned to the quotient, which is the result of the division. At this point, it is set to 0.

```
begin
--Result of A/B division
RES <= STD_LOGIC_VECTOR(Q);
```

Next, we divide the main clock.

```
--Clock divider
process(MCLK)
variable Counter : INTEGER range 0 to N;
begin
        if rising_edge(MCLK) then
            Counter := Counter + 1;
             --Clock frequency 1000/2 = 500Hz
             if (Counter = N/1000-1) then
                    Counter := 0;
                    CLK_DIV <= not CLK_DIV;
             end if;
        end if;
end process;
```

This is a process sensitive to MCLK. So, at every change of MCLK, this process runs. Since MCLK has a high frequency, it runs many times. A variable Counter is declared in this code. This Counter is increased by 1 at every rising edge of the clock cycle. In short, with the help of the *if* statement inside, main clock has a rising edge 50*10^6 times in a second, and every 0.001 seconds, CLK_DIV is rising or falling depending on its current state. This way, we have set up a new clock to be used. Main clock was changing every 0.000000002 seconds which is too fast. This new clock changes every 0.001 seconds.

Now we proceed to the division process.

```
--X/Y Division
process(CLK_DIV)
begin
        if rising_edge(CLK_DIV) then
            case State is
                when init =>
                    --Transfer inputs to local signals
                    X <= unsigned(A);
                    Y <= unsigned(B);
                    --Start computing X/Y with Start push-button and
                    --non-zero denominator
                    if (Start = '1') and (Y > 0) then
                        State <= compute;
                        Q <= "0000"; --Reset quotient to zero
                    else
                        State <= init;
                    end if;
                    when compute =>
                            if X >= Y then
                                    X <= X -Y;
                                    Q <= Q + 1;
                            end if;
                            if X >= Y then
                                    State <= compute;
                            else
                                    State <= done;
                            end if;
                    when done =>
                            R <= X;
                            State <= init;
                    when others =>
                            State <= init;

            end case;
        end if;
end process;
end Behavioral;
```

First, we note the CLK_DIV part. We have noted that CLK_DIV is changed every 0.001 second. therefore, its rising edge occurs every 0.0005 seconds. At these rising edges, we check state and perform operations based on the current state.

At the INIT state, inputs A and B are introduced as *unsigned integers* and assigned to X and Y. After this process, a START signal is expected. As long as this START signal is not received, state is not changed and stays at the INIT state.

However, if START signal is received, state is set to compute. At this state, the introduced algorithm is used. This algorithm is a simple but a working one, and we will implement the 4-bit signed division algorithm mainly using this. In its core, division is finding out how many *divisors* inside the *dividend*. We subtract *divisor* one at a time and continue to do so as long as the result of this division does not give a negative number. Therefore, if we count how many times this subtraction is repeated, we find out the *quotient* and the remained subtraction becomes *remainder*. Depending on the possible result of this subtraction, next state is determined. When there is no possible subtractions without getting negative outputs, we set state to DONE.

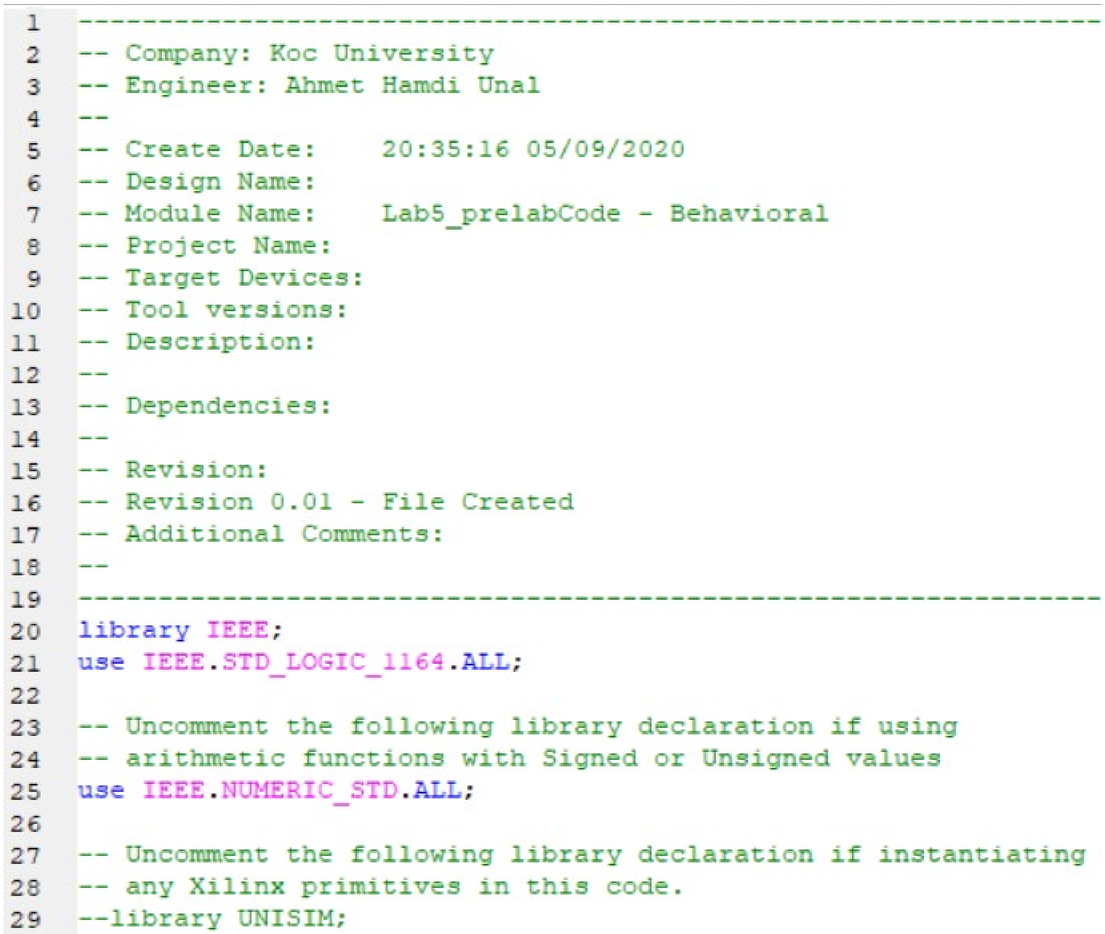
At this DONE state, remainder is assigned to the intermediate signal R, though it is not assigned to any LEDs or any other output ports. Also, state is set to INIT so that new divisions can be done.

When others part is crucial. If a glitch occurs, or any other unexpected stuff, current state may turn out to be something which is not assigned by us. In such case, we force the system to go back to INIT state so that we can keep the program running.

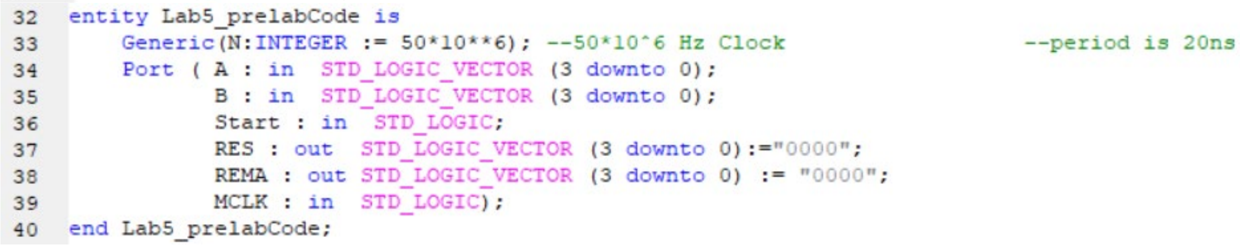
Now, I move onto my design.

My 4-bit signed division in 2’s complement system VHDL Code:

The code I am going to provide below is written by me to perform division operation with 4-bit signed integers. Anything that is necessary is already explained at the beginning of this section. Nonetheless, after displaying the code, I will make some entries on code implementations of the algorithms I have already provided. After explaining the code, I will provide simulation results for necessary test cases. At last, I will put some pictures taken during the run on the FPGA board. Before starting, let me note that I have modified the code pretty much. I will go through necessary details. Let me start by noting that I am also showing the remainder result by using LEDs. So, I reserve 4 rightmost LEDs for the result RES and 4 leftmost LEDs for the remainder REMA.

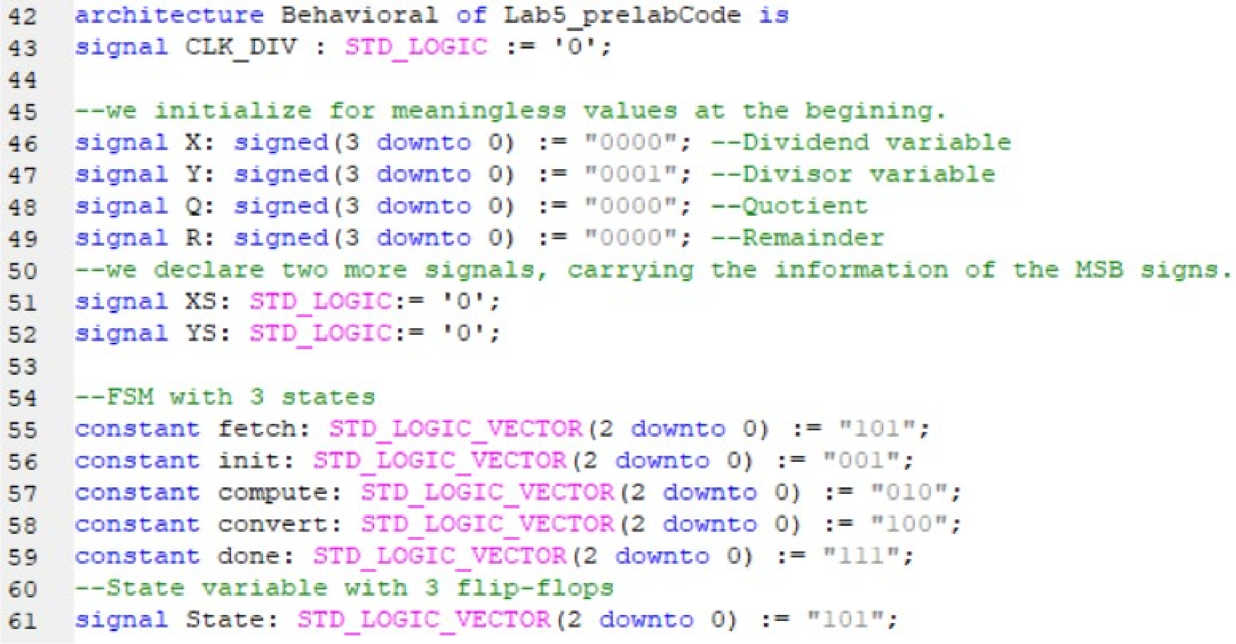


```vhdl
----------------------------------------------------------------
-- Company: Koc University
-- Engineer: Ahmet Hamdi Unal
--
-- Create Date:    20:35:16 05/09/2020
-- Design Name:
-- Module Name:    Lab5_prelabCode - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
```



```vhdl
entity Lab5_prelabCode is
    Generic(N:INTEGER := 50*10**6); --50*10^6 Hz Clock                  --period is 20ns
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Start : in  STD_LOGIC;
           RES : out  STD_LOGIC_VECTOR (3 downto 0):="0000";
           REMA : out STD_LOGIC_VECTOR (3 downto 0) := "0000";
           MCLK : in  STD_LOGIC);
end Lab5_prelabCode;
```

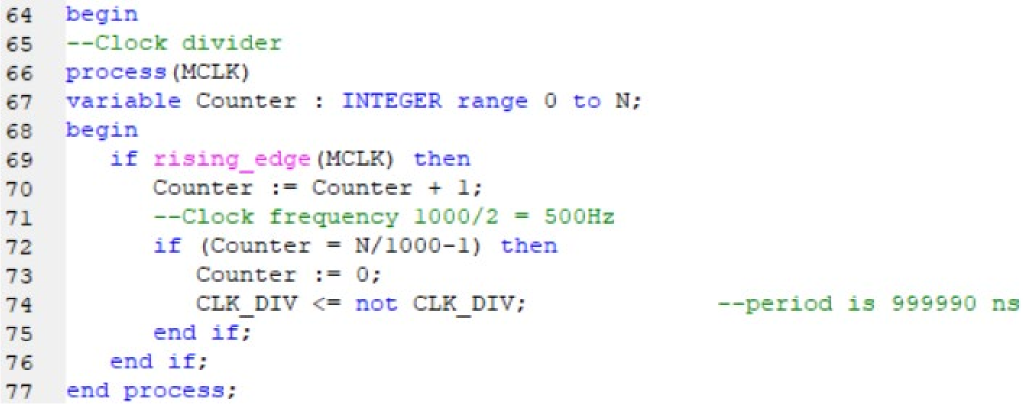
I have added an extra output variable REMA, which will be output in the end. I have also set REMA's and RES's initial value to 0000. This was missing in the example code in the manual, which I believe is an undesired situation (due to uninitialized values). To calculate the period using the frequency, I have used the relation $f = 1/T$. Therefore, MCLK has a $T_c = 20\ ns$.



```
architecture Behavioral of Lab5_prelabCode is
signal CLK_DIV : STD_LOGIC := '0';

--we initialize for meaningless values at the begining.
signal X: signed(3 downto 0) := "0000"; --Dividend variable
signal Y: signed(3 downto 0) := "0001"; --Divisor variable
signal Q: signed(3 downto 0) := "0000"; --Quotient
signal R: signed(3 downto 0) := "0000"; --Remainder
--we declare two more signals, carrying the information of the MSB signs.
signal XS: STD_LOGIC:= '0';
signal YS: STD_LOGIC:= '0';

--FSM with 3 states
constant fetch: STD_LOGIC_VECTOR(2 downto 0) := "101";
constant init: STD_LOGIC_VECTOR(2 downto 0) := "001";
constant compute: STD_LOGIC_VECTOR(2 downto 0) := "010";
constant convert: STD_LOGIC_VECTOR(2 downto 0) := "100";
constant done: STD_LOGIC_VECTOR(2 downto 0) := "111";
--State variable with 3 flip-flops
signal State: STD_LOGIC_VECTOR(2 downto 0) := "101";
```

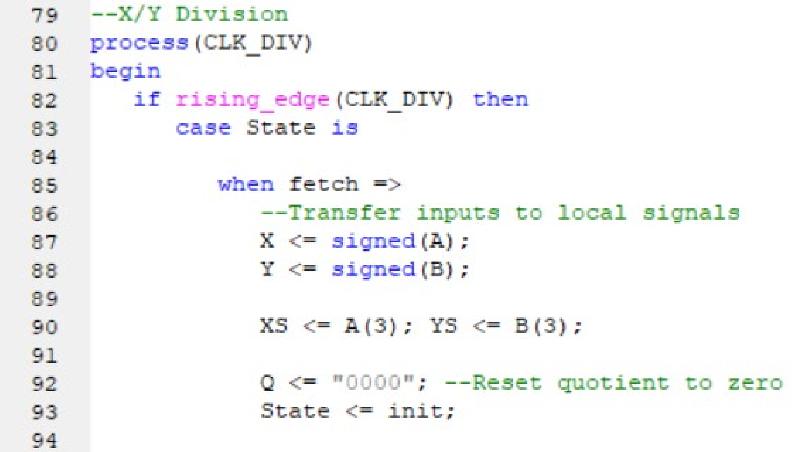
I start by assigning a value to CLK_DIV. This was missing in the example code provided in the lab manual, which makes CLK_DIV uninitialized. This is not something desired, so I set it to 0. I have also defined two new intermediate signals XS and YS. They will be storing the **S**igns of X and Y, correspondingly.

Furthermore, I define two more states, and modify some of the states. In total, I have 5 states with the state codes provided above. In *fetch* state, I assign A and B to X and Y and store the sign information of A and B. In *init* state, I make the negative to positive conversion. *compute* state is the same. In *convert* state, I make the necessary conversions, depending on the sign combinations of the input numbers, and assign X to R. In the state *done*, RES and REMA are updated with Q and R.



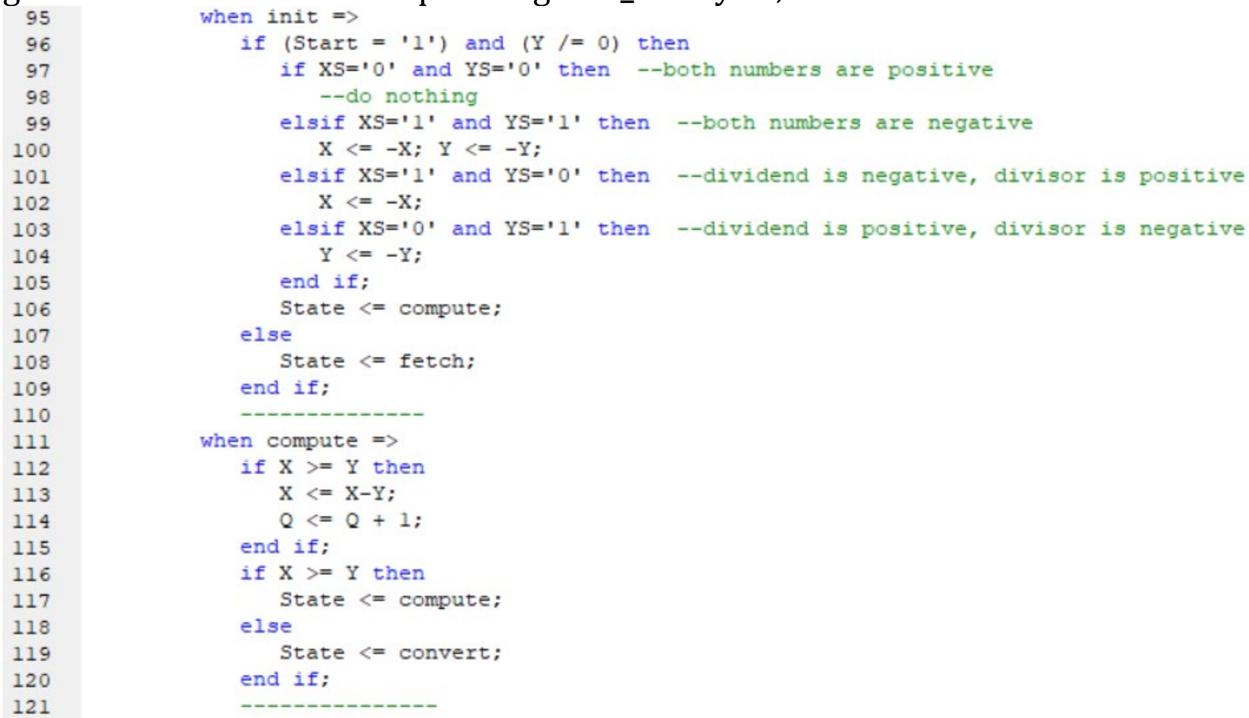
```
begin
--Clock divider
process(MCLK)
variable Counter : INTEGER range 0 to N;
begin
   if rising_edge(MCLK) then
      Counter := Counter + 1;
      --Clock frequency 1000/2 = 500Hz
      if (Counter = N/1000-1) then
         Counter := 0;
         CLK_DIV <= not CLK_DIV;             --period is 999990 ns
      end if;
   end if;
end process;
```

This part is also unchanged: CLK_DIV assignment is the same. The important part here may be the period of CLK_DIV. MCLK starts from 0 and has a periodicity of 10 ns. Therefore, counter becomes 1 at 10ns, 2 at 30 ns, 3 at 50ns, etc. When counter becomes N/1000-1=49999, CLK_DIV negates. From the sequence, we can deduce that $t = (2\ Counter - 1) \times 10$ ns. Therefore, we find $T/2 = 999{,}970$ ns. So, period of CLK_DIV is $1.999{,}940 \approx 2\ ms$. You can verify this result by observing the simulation results in the upcoming sections.



```
--X/Y Division
process(CLK_DIV)
begin
   if rising_edge(CLK_DIV) then
      case State is

         when fetch =>
            --Transfer inputs to local signals
            X <= signed(A);
            Y <= signed(B);

            XS <= A(3); YS <= B(3);

            Q <= "0000"; --Reset quotient to zero
            State <= init;

```

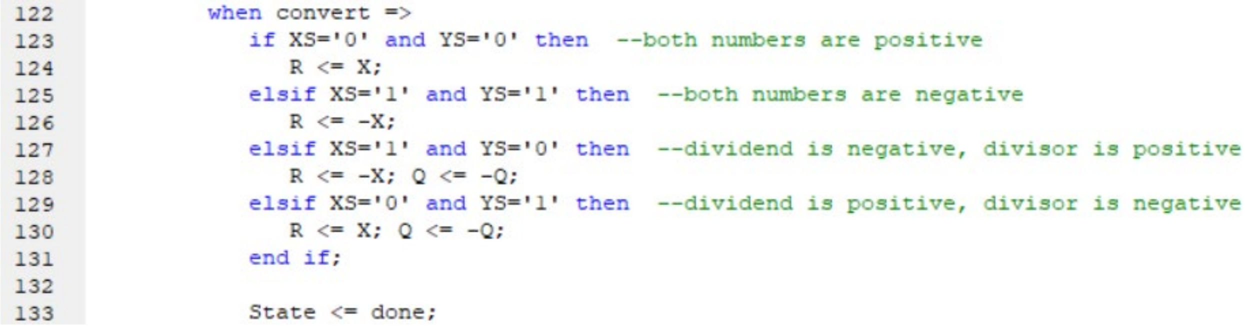
In this fetch state, X and Y are assigned. Note that here, the assignments are *signed*. In the manual example, since division was unsigned, they were unsigned. Also, XS and YS are assigned with the MSB of the numbers entered. Q is set to 0000 and to go to the next state in the upcoming CLK_DIV cycle, State is set to *init*.



```
         when init =>
            if (Start = '1') and (Y /= 0) then
               if XS='0' and YS='0' then  --both numbers are positive
                  --do nothing
               elsif XS='1' and YS='1' then  --both numbers are negative
                  X <= -X; Y <= -Y;
               elsif XS='1' and YS='0' then  --dividend is negative, divisor is positive
                  X <= -X;
               elsif XS='0' and YS='1' then  --dividend is positive, divisor is negative
                  Y <= -Y;
               end if;
               State <= compute;
            else
               State <= fetch;
            end if;
            --------------
         when compute =>
            if X >= Y then
               X <= X-Y;
               Q <= Q + 1;
            end if;
            if X >= Y then
               State <= compute;
            else
               State <= convert;
            end if;
            ---------------
```

If the Start button is pressed and Y is not 0, division starts. Else, it goes back to the *fetch* state, giving the user another chance to enter a nonzero divisor.

To be able to use the same division algorithm for the unsigned binary numbers, I convert negative numbers to positive numbers in the *init* state after being stored their signs in the *fetch* state. Now, the program can move onto the *compute* state, where exactly the same computing algorithm is used. After *compute* state is completed (it may repeat for a few times), program continues to the *convert* state.



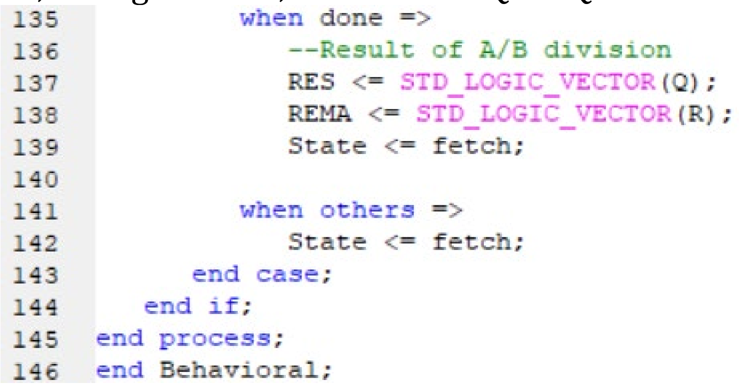
```
      when convert =>
        if XS='0' and YS='0' then  --both numbers are positive
          R <= X;
        elsif XS='1' and YS='1' then  --both numbers are negative
          R <= -X;
        elsif XS='1' and YS='0' then  --dividend is negative, divisor is positive
          R <= -X; Q <= -Q;
        elsif XS='0' and YS='1' then  --dividend is positive, divisor is negative
          R <= X; Q <= -Q;
        end if;

        State <= done;
```

This state may be the most important one in terms of experimenting and deducing a general pattern. By trying for the numbers in the range {−8,7}, I have deduced the following results for the division:

Table 1: Signed division rules

Operation	Remainder	Quotient
+/+	+	+
+/−	−	+
−/+	+	−
−/−	−	−

At the *init* state, I had converted all negative numbers to positive to be able to use the unsigned division algorithm. Therefore, in this *convert* state, I do these conversions to fix the sign issues. As expected, there is no change in the +/+ case since the algorithm is mainly constructed for this case. I directly set the value of X to R. In the -/- case, both remainder and quotient must be negative. Since I have converted negative to positive in the *init* state, the results I get are always positive. So, I assign -X to R, and convert Q to -Q. The same logic goes for +/- and -/+ cases.



```
      when done =>
        --Result of A/B division
        RES <= STD_LOGIC_VECTOR(Q);
        REMA <= STD_LOGIC_VECTOR(R);
        State <= fetch;

      when others =>
        State <= fetch;
    end case;
  end if;
end process;
end Behavioral;
```

At last, *done* state is performed. In this state, Q is assigned to the result output variable RES after being converted to STD_LOGIC_VECTOR and R is assigned to the remainder output variable REMA after being converted in the same manner. The reason for the conversion is that RES and REMA are defined as STD_LOGIC_VECTORs, whereas Q and R are signed numbers. After all is done, program goes back to *fetch* state to get new numbers for division in the next clock cycles. There is also an extra case which may occur due to glitches or something else. In this case, program goes back to the *fetch* state, which is a safe option.

Below, I provide the pin assignments. It is mostly unchaged. The only addition is that I have added the 4 leftmost LEDs to show the result of the remainder.

PIN ASSIGNMENTS (.UCF)

```
NET "MCLK" LOC = "P40"; # CLOCK

NET "Start" LOC = "P32"; # BTN0

NET "A<0>" LOC = "P94"; # SW4
NET "A<1>" LOC = "P90"; # SW5
NET "A<2>" LOC = "P88"; # SW6
NET "A<3>" LOC = "P85"; # SW7

NET "B<0>" LOC = "P15"; # SW0
NET "B<1>" LOC = "P12"; # SW1
NET "B<2>" LOC = "P5"; # SW2
NET "B<3>" LOC = "P4"; # SW3

NET "RES<0>" LOC = "P16"; # LED0
NET "RES<1>" LOC = "P13"; # LED1
NET "RES<2>" LOC = "P6"; # LED2
NET "RES<3>" LOC = "P3"; # LED3

NET "REMA<0>" LOC = "P86"; # LED6
NET "REMA<1>" LOC = "P84"; # LED7
NET "REMA<2>" LOC = "P83"; # LED8
NET "REMA<3>" LOC = "P77"; # LED9
```

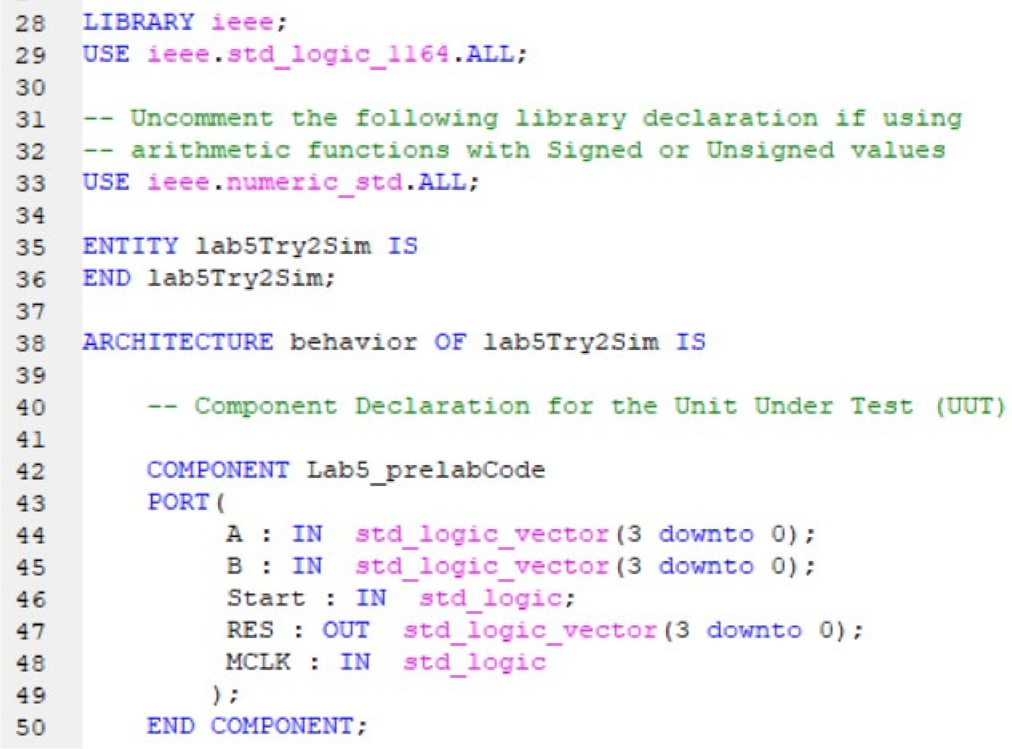
Simulation Results:

I have tested the functionality of the Start button and tested some of the cases that are indicated in the .sim file. Since used many times, below I provide the 2’s complement system representation of the selected numbers:

Figure 1: 2's Complement Representation of Some 4-bit Numbers

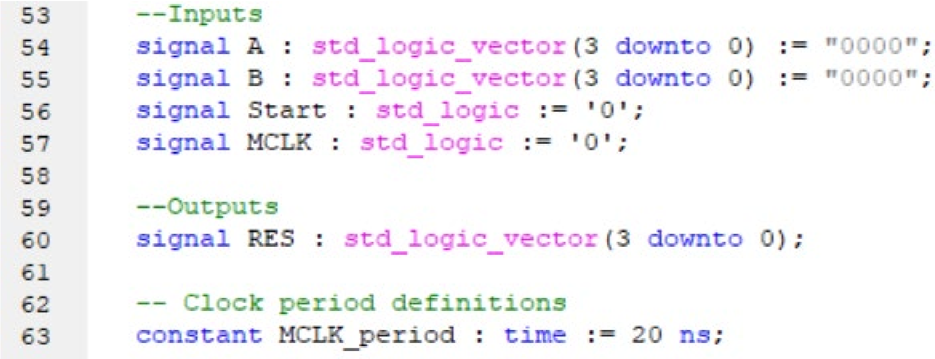
7	0111	-7	1001
6	0110	-6	1010
1	0001	-1	1111
3	0011	-3	1101

Below is the simulation code, and then, the simulation results.



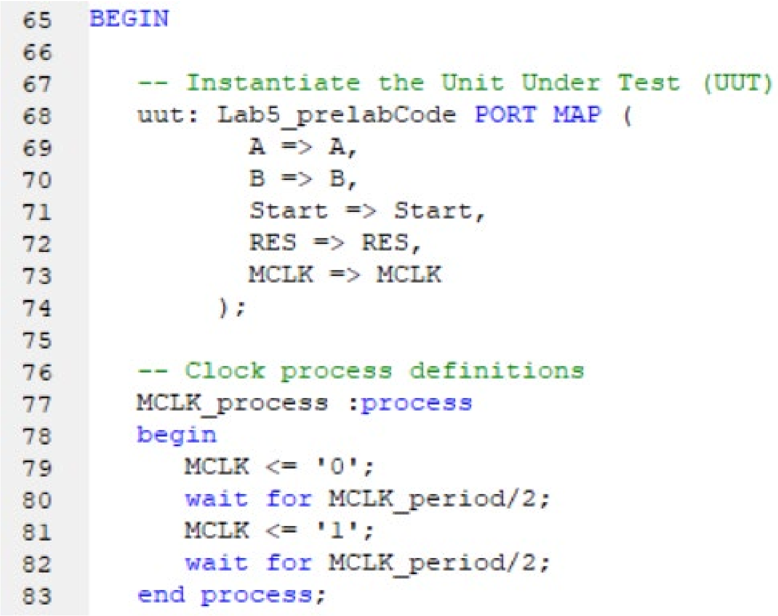
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
USE ieee.numeric_std.ALL;

ENTITY lab5Try2Sim IS
END lab5Try2Sim;

ARCHITECTURE behavior OF lab5Try2Sim IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Lab5_prelabCode
    PORT(
         A : IN  std_logic_vector(3 downto 0);
         B : IN  std_logic_vector(3 downto 0);
         Start : IN  std_logic;
         RES : OUT  std_logic_vector(3 downto 0);
         MCLK : IN  std_logic
        );
    END COMPONENT;
```

Up to this point, code is generic which I have made no modifications.



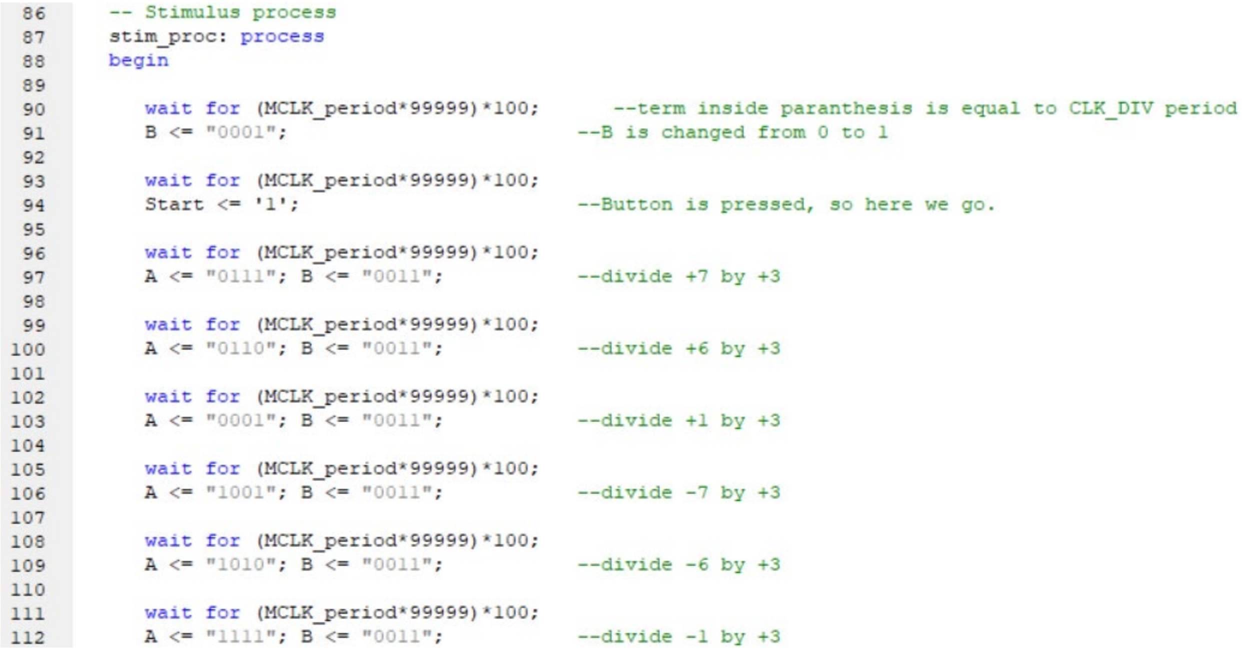
```
   --Inputs
   signal A : std_logic_vector(3 downto 0) := "0000";
   signal B : std_logic_vector(3 downto 0) := "0000";
   signal Start : std_logic := '0';
   signal MCLK : std_logic := '0';

   --Outputs
   signal RES : std_logic_vector(3 downto 0);

   -- Clock period definitions
   constant MCLK_period : time := 20 ns;
```

I have set the initial values of A and B to 0000 here. So, in my simulation, I have assumed that the user starts using the FPGA board with the A=B=0000 switch combination (Start button un-pressed). Moreover, MCLK_period is set to 20 ns, which is the MCLK period defined in the .VHD file.

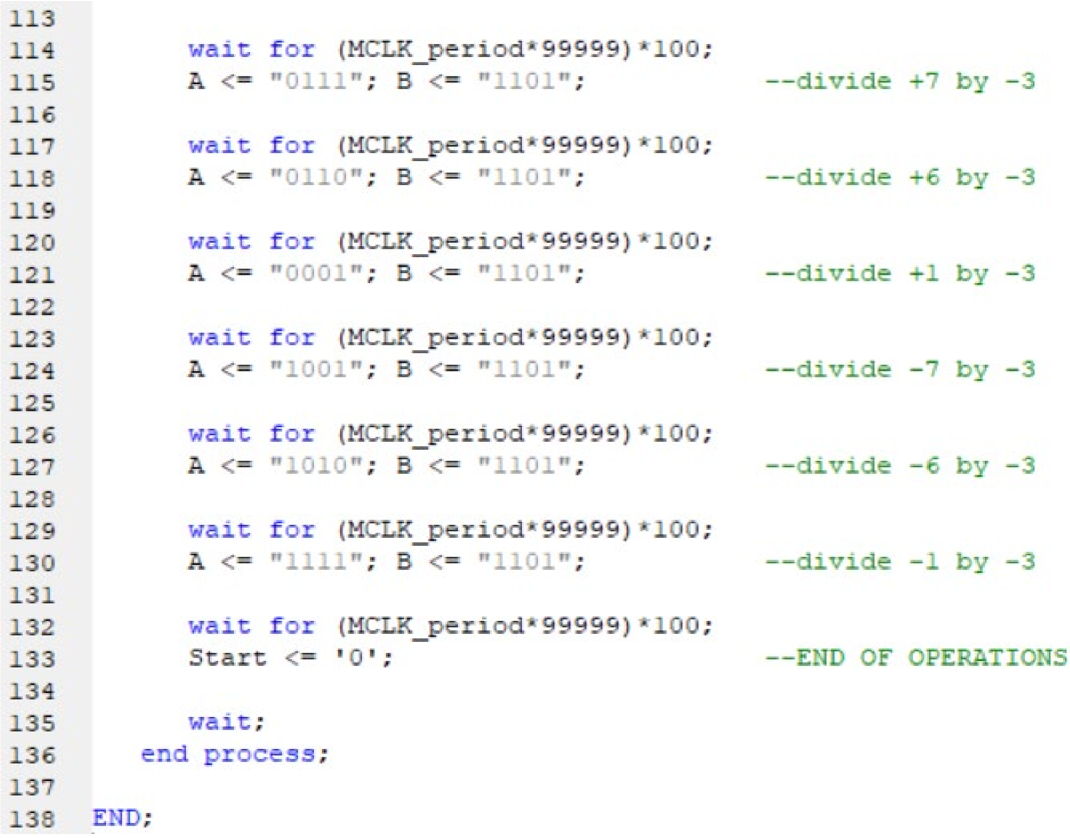


```
BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: Lab5_prelabCode PORT MAP (
          A => A,
          B => B,
          Start => Start,
          RES => RES,
          MCLK => MCLK
        );

   -- Clock process definitions
   MCLK_process :process
   begin
      MCLK <= '0';
      wait for MCLK_period/2;
      MCLK <= '1';
      wait for MCLK_period/2;
   end process;
```

Again, this part is generic, no modifications I have made. Below, my main contribution comes to the simulation.



```
   -- Stimulus process
   stim_proc: process
   begin

      wait for (MCLK_period*99999)*100;      --term inside paranthesis is equal to CLK_DIV period
      B <= "0001";                       --B is changed from 0 to 1

      wait for (MCLK_period*99999)*100;
      Start <= '1';                      --Button is pressed, so here we go.

      wait for (MCLK_period*99999)*100;
      A <= "0111"; B <= "0011";          --divide +7 by +3

      wait for (MCLK_period*99999)*100;
      A <= "0110"; B <= "0011";          --divide +6 by +3

      wait for (MCLK_period*99999)*100;
      A <= "0001"; B <= "0011";          --divide +1 by +3

      wait for (MCLK_period*99999)*100;
      A <= "1001"; B <= "0011";          --divide -7 by +3

      wait for (MCLK_period*99999)*100;
      A <= "1010"; B <= "0011";          --divide -6 by +3

      wait for (MCLK_period*99999)*100;
      A <= "1111"; B <= "0011";          --divide -1 by +3
```



```

      wait for (MCLK_period*99999)*100;
      A <= "0111"; B <= "1101";           --divide +7 by -3

      wait for (MCLK_period*99999)*100;
      A <= "0110"; B <= "1101";           --divide +6 by -3

      wait for (MCLK_period*99999)*100;
      A <= "0001"; B <= "1101";           --divide +1 by -3

      wait for (MCLK_period*99999)*100;
      A <= "1001"; B <= "1101";           --divide -7 by -3

      wait for (MCLK_period*99999)*100;
      A <= "1010"; B <= "1101";           --divide -6 by -3

      wait for (MCLK_period*99999)*100;
      A <= "1111"; B <= "1101";           --divide -1 by -3

      wait for (MCLK_period*99999)*100;
      Start <= '0';                       --END OF OPERATIONS

      wait;
   end process;

END;
```

Here, I have provided the test cases. I have made the simulation to wait for (MCLK_period*99999)*100 seconds to move onto the next stimulus. MCLK_period*99999 corresponds to the half of the CLK_DIV period. I have multiplied it with 100 so that any operation can be finished in that time range. Actually, multiplying with, say, 10 would possibly be enough. I have multiplied it with 100 to separate results of operations from each other and get better visuals in the simulation part.

First, I change B to 1, but left Start=0. In this time range, I expect no state transitions to *compute*. Next, I press Start and kept it pressed until the last operation in the code, so that test cases can be tested.

See the screenshots I have taken from the simulation below. I will provide only some of the signals in the simulations for comprehensibility. The ones I will show will be necessary for the verification of the function of the code. If interested, you can run the .WCFG file I have provided to observe how the other signals change through the run.

Also, please note that I will show the simulation variable values in decimal so that verification can be done easily, but in the code, everything is binary.

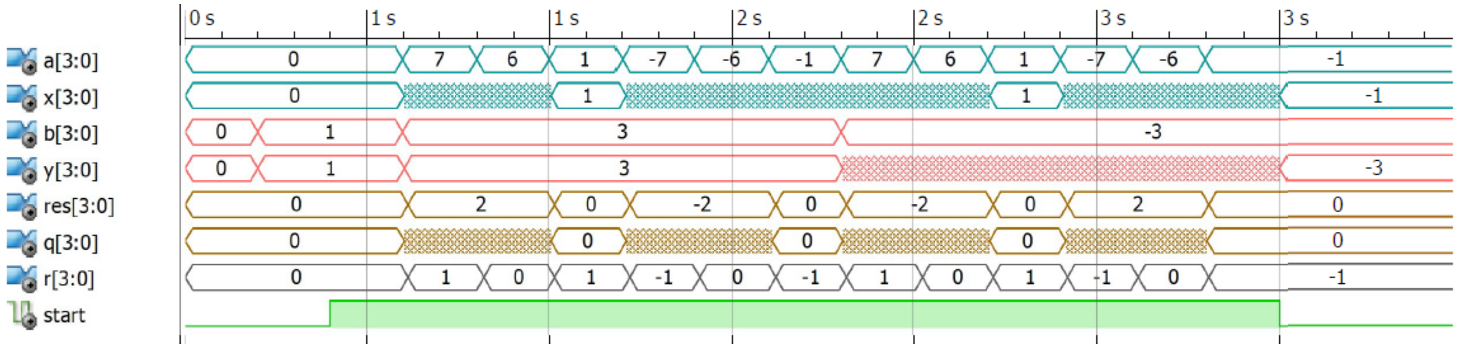


Figure 2: Simulation Results

When Figure 2 is observed detailly, it is actually a very good tool to understand the code we have just written. At first, everything is 0. Then, divisor (B) becomes 1, but nothing changes. Then, Start becomes 1 (button is pressed). Still nothing changes, but actually the quotient and remainder is calculated and found as 0. Then, the test cases I have explained in the simulation code is done in order. Note that in the time being A is constant, X is subject to change. Same goes for B and Y. This is normal, the input (switch combination) does not change, while we change the corresponding *number* variables inside the code for calculations.

To understand the simulation result, simply divide $\boldsymbol{a}$ by $\boldsymbol{b}$, and then check $\boldsymbol{res}$ and $\boldsymbol{r}$. For example, look at $t = 2s$. -6 is divided by 3, and the quotient is -2 and remainder is 0, as it should be.

Observe that when Start is signal is set back to 0 at the end (that is, Start button is not pressed anymore), everything stays as it is. Latest remainder (-1) and result (0) is being shown on the FPGA board by LEDs, and they will be continued to be shown as long as the user does not press the Start button and activates the circuit.

Experimental Results on the FPGA Board:

Since I do not have FPGA board due to circumstances, I have requested to a friend of mine to upload my code to her FPGA board and try the following cases: $5/2$, $-5/2$, $5/-2$ and $-5/-2$. The results are all correct and as follows (please note that 4 rightmost bits are RES and 4 leftmost bits are REMA):

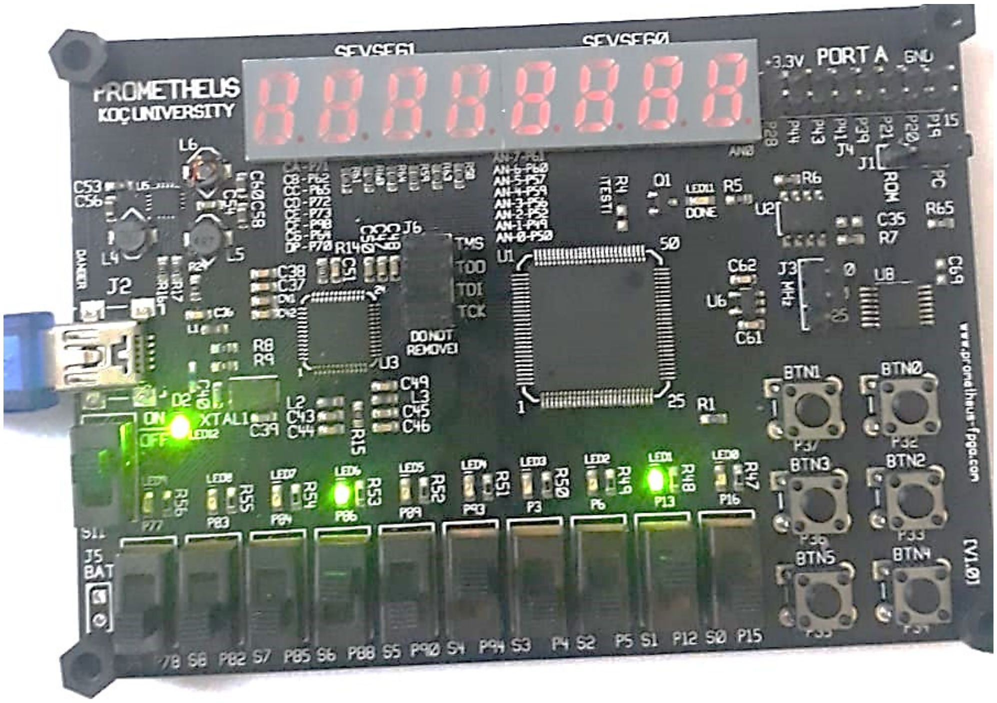


Figure 3: FPGA Board: 5/2

Above is 5/2. The remainder is 0001=1 and the result is 0010=2, as expected.

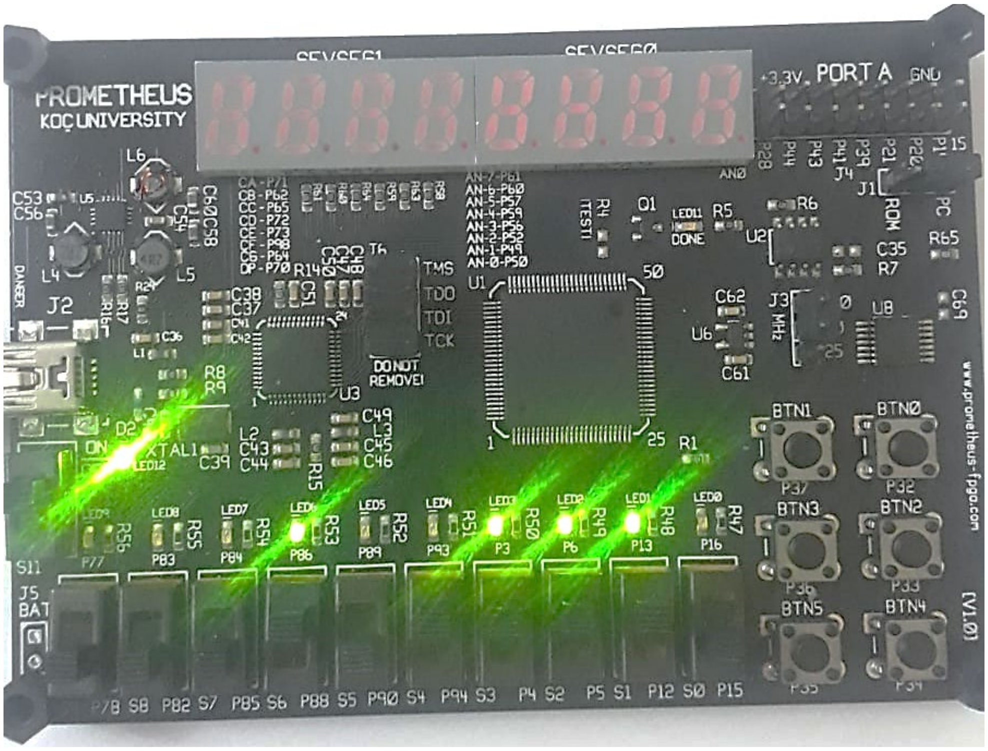


Figure 4: FPGA Board: 5/-2

Above is 5/-2. The remainder is 0001=1 and the result is 1110=-2, as expected.

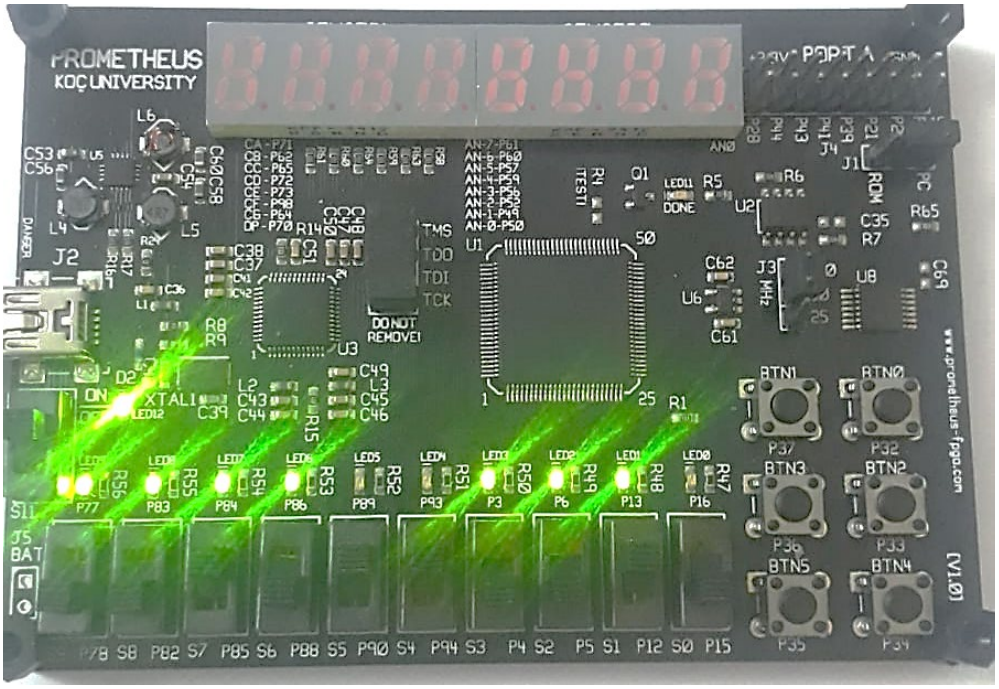

PROMETHEUS
KOÇ UNIVERSITY
PORT A
LED7 LED6 LED5 LED4 LED3 LED2 LED1 LED0
P77 P83 P84 P86 P89 P93 P3 P6 P13 P16
BTN1 BTN0 BTN3 BTN2 BTN5 BTN4
P37 P32 P36 P33 P35 P34
P78 S8 P82 S7 P85 S6 P88 S5 P90 S4 P94 S3 P4 S2 P5 S1 P12 S0 P15
DO NOT REMOVE!
www.prometheus-fpga.com
[V1.0]

Figure 5: FPGA Board: -5/2

Above is -5/2. The remainder is 1111=-1 and the result is 1110=-2, as expected.

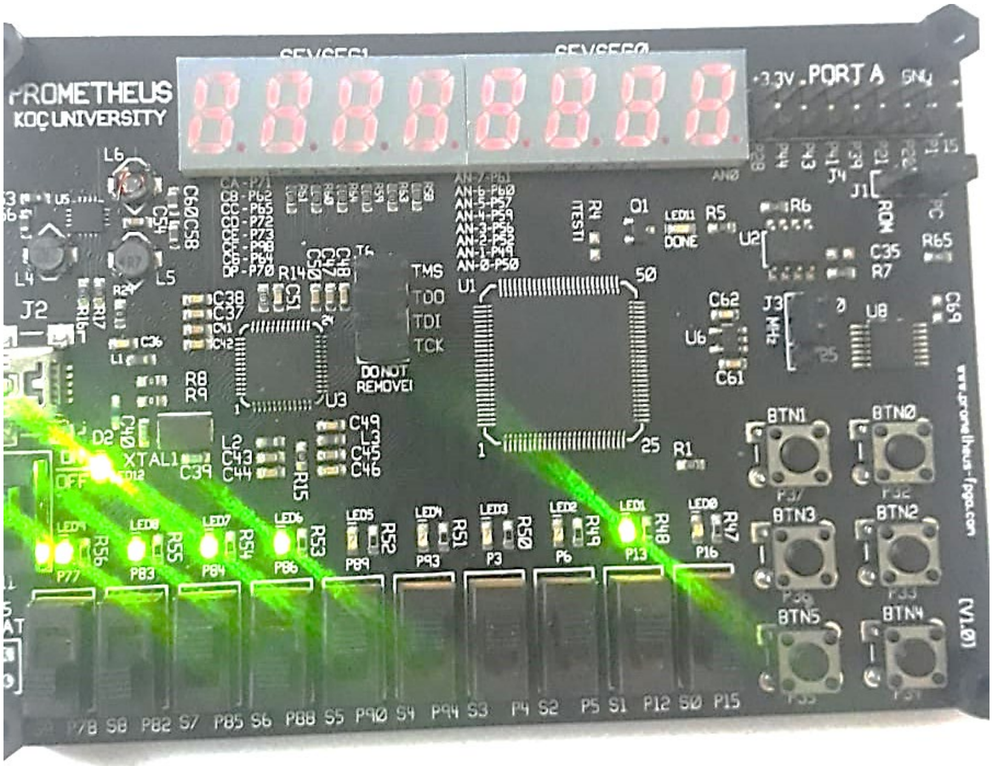

SEVSEG1
SEVSEG0
PROMETHEUS
KOÇ UNIVERSITY
PORT A
LED7 LED6 LED5 LED4 LED3 LED2 LED1 LED0
P77 P83 P84 P86 P89 P93 P3 P6 P13 P16
BTN1 BTN0 BTN3 BTN2 BTN5 BTN4
P78 S8 P82 S7 P85 S6 P88 S5 P90 S4 P94 S3 P4 S2 P5 S1 P12 S0 P15
DO NOT REMOVE!
www.prometheus-fpga.com
[V1.0]

Figure 6: FPGA Board: -5/-2

Above is -5/-2. The remainder is 1111=-1 and the result is 0010=2, as expected.

Question 5.2

FSM Design of the DAC: Design a finite state machine for the DAC addressing specifications given in the problem statement. Construct the minimum data path for this FSM. Specify functional blocks and all control inputs. Design the control unit and find the control input functions.

Before presenting my design, let me introduce some of the variables.

Table 2: Variables and their explanations

VARIABLE	EXPLANATION
D	Vector representing input switches
B	Vector representing input buttons
X	Latest number (result) computed
Y	Latest input number
L	Led output
S	Seven-Segment output
P	Button assignments

There are 6 buttons. We require on of them to be pressed at a time. I will notate these cases with the assignments below:

Table 3: Button Assignments

P (3 downto 1)	Operation	Button on Board
000	Addition	B0
001	Subtraction	B1
011	Negation	B2
010	XOR	B3
110	AND	B4
100	OR	B5
1x1	*Undesired*	*Other*

It may seem unnecessary at this point, but I will define a new variable V (for Valid) that checks whether there is only 1 button pressed at a time, and that button is one of the 6 buttons we have defined. Since there are only 6 buttons on the FPGA Board, it may seem unnecessary, but it is a safe option. Also, as I said, it prevents multiple pushes.

			P3	
	P2 P1 \ P3	0	1	
	00	1	1	
	01	1	0	P1
P2	11	1	0	
	10	1	1	

K-Map 1: Table for V

Therefore, $V = P(3)' + P(1)'$. So, except both P3 and P1 are 1, all other cases are valid. We can even make this V better by enforcing the condition that no two or more buttons are allowed to be pressed at the same time. The idea is simple, but the equation will look confusing. Below, I have written it. Note that inside the second parenthesis, while a button is 1, the rest is forced to be 0. So, desired condition is applied. This implementation of V will be useful in the upcoming parts.

V Equation

$$V = (P(3)' + P(1)')\,(B0'B1'B2'B3'B4'B5 + B0'B1'B2'B3'B4B5' + B0'B1'B2'B3B4'B5' + B0'B1'B2B3'B4'B5' + B0'B1B2'B3'B4'B5' + B0B1'B2'B3'B4'B5')$$

See the circuit design for V in the next parts for a better understanding.

ASM Chart:

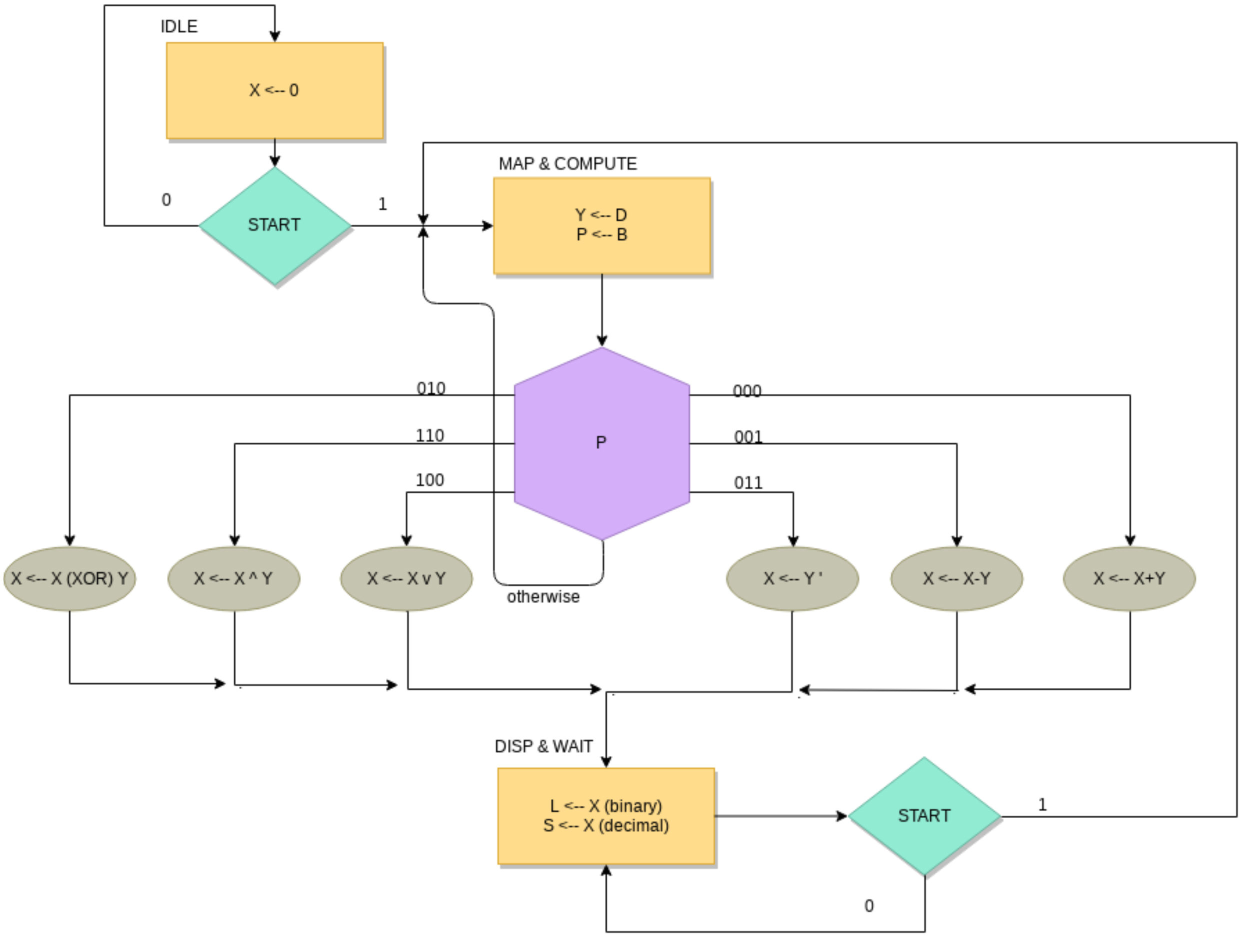


Figure 7: ASM Chart (main)

IDLE is the first state. For example, the device is turned on. Then, it starts in the state IDLE. I set X to 0 at this point, which is a free choice. As long as START is not received, it stays in IDLE state.

When a button is pressed, START is 1, and IDLE goes to the next state MAP & COMPUTE. In this state, switches are read and depending on whether they are turned on or off, the resulting 2's complement binary numbers is assigned to Y. similarly, buttons are read and depending on which ones are pressed or not, button assignment are done. At this point, these assignments are done in accordance with the Table 2 I have provided.

If P is an invalid state of P, then, system stays in MAP & COMPUTE state. Otherwise, the corresponding operation is operated. After the operation is done, DISP & WAIT becomes the next state.

In DISP & WAIT state, the result of the operation is transferred to LEDs and Seven-Segment Display via corresponding variable updates. After the result is shown on the FPGA board, another START command is waited. At this point, the user sets the next input number that she will be using, and then presses one of the buttons. Therefore, START becomes 1, and system goes back to MAP & COMPUTE state.

Since it is not stated, this design does not stop functioning. In other words, it does not have an EXIT mode. After it is run once, it keeps going between these 3 states until is turned off externally by the user.

Also, this design does not have a RESET mode either. So, the operations must be done over the latest result we have obtained. Note that both RESET and EXIT states are not necessary nor hard to implement operations. One can easily turn the FPGA off from its switch or RESET it by performing multiplication when all the switches are turned off. Since it is not required, I will not add such states: the lab manual states that the most minimal design is desired. However, for example, the two leftmost switches which we do not use can be introduced for these purposes.

State Diagram:

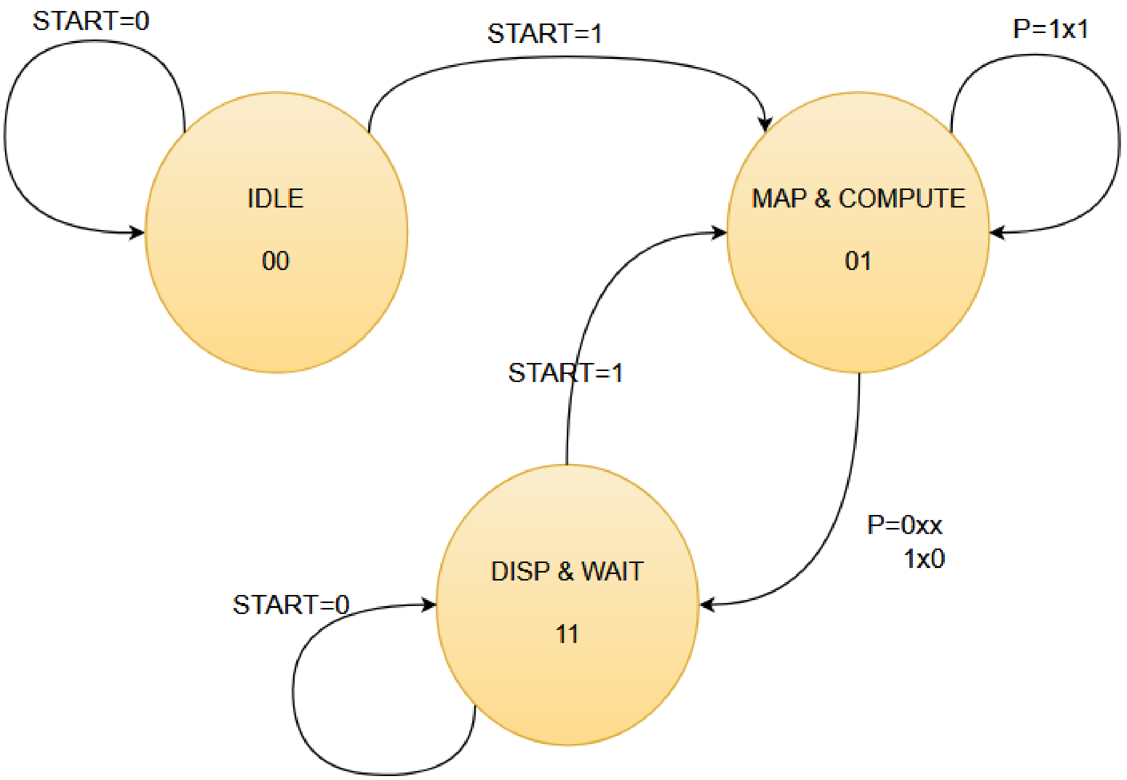


Figure 8: State Diagram (main)

From ASM Chart, there were 3 states. For the minimal design, I have used $ceil\{\log_2 n\}$ -where n is the number of states- to find the minimal state assignment. Therefore, each state is of 2 bits. Same inputs in ASM Chart is used here as inputs. Since state transitions do not give out outputs in this design, there is no output.

Before going on to the state table, I must point out a problem. P input is a 3-digit input. As a result, there are 4 inputs, and 2 states. Therefore, when writing next state equations, I have to deal with a 6-variable K-map. I do not want to do this, so I will introduce a new variable V, which I have mentioned under K-Map 1. V is the shorthand for valid. Since as long as the pressed button is valid, system moves on to the next state. Single variable V can be used here instead of the 3 digits of P. Please note that we need P to be a vector of 3 digits, it is the most minimal assignment for P. We exploit the values of P when we decide on *which* operation to be performed, so we do not actually use it for state transitions. Therefore, though we add an extra step to the design, this eases the circuit analysis and comprehensibility of the design. When V is introduced, ASM Chart is changed to:

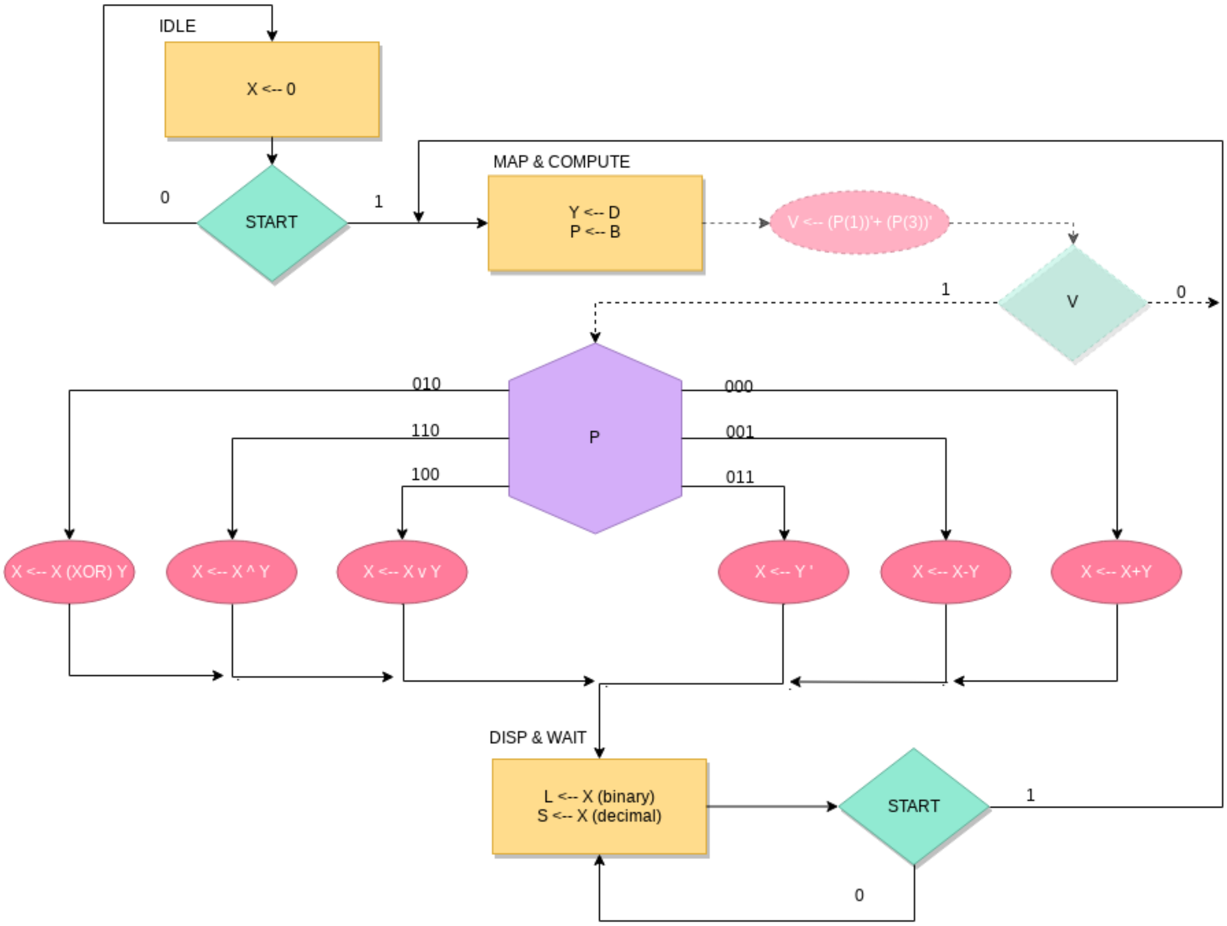


Figure 9: ASM Chart (alternative)

Since we check for validity before reaching the vector decision box (purple), vector decision box does not need an *otherwise* output in this design. I have used dashed lines to indicate the change in the chart. It should be interpreted as solid lines by the reader.

With this move, the state diagram is also simplified. The new state diagram and table is as follows:

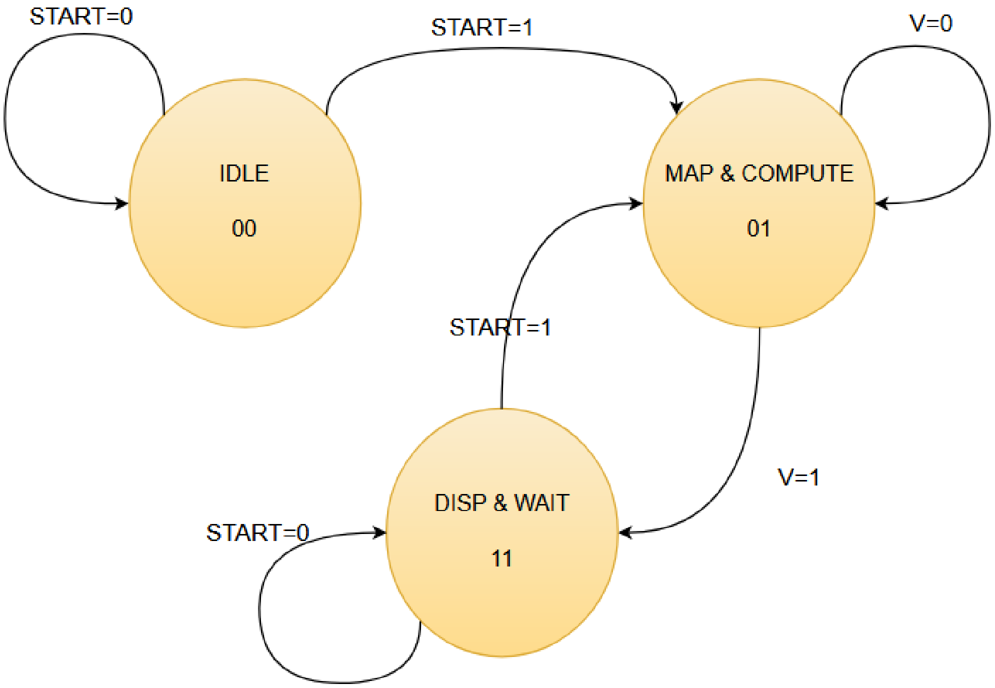


Figure 10: State Diagram (alternative)

State Table:

Table 4: State Table

Present State(Q_1Q_0)	Input(Start)	Input (V)	Next State (Q_1^+, Q_0^+)
00	0	x	00
	1	x	01
01	x	0	01
	x	1	11
11	0	x	11
	1	x	01
10	x	x	00

Now we are to construct 4-variable K-maps, thanks to this new variable V. Please note that for all inputs, next state of the state 10 (undesired) is set to 00 to be safe.

K-Maps:

First, let us construct the K-map for Q_1:

K-Map 2: Q1

				Start		
	Q1 Q0 \ Start V	00	01	11	10	
	00	0	0	0	0	
	01	0	1	1	0	Q0
Q1	11	1	1	0	0	
	10	0	0	0	0	
			V			

$Q1$ Equation

Therefore, it is concluded that $Q_1^+ = Q_1Q_0\overline{Start} + \overline{Q_1}Q_0V$

Next, construct K-map for Q_0:

K-Map 3: Q0

				Start		
	Q1 Q0 \ Start V	00	01	11	10	
	00	0	0	1	1	
	01	1	1	1	1	Q0
Q1	11	1	1	1	1	
	10	0	0	0	0	
			V			

$Q0$ Equation

Therefore, it is concluded that $Q_0^+ = Q_0 + \overline{Q_1}\,Start$

Circuit Designs:

The figure below is for Control Unit.

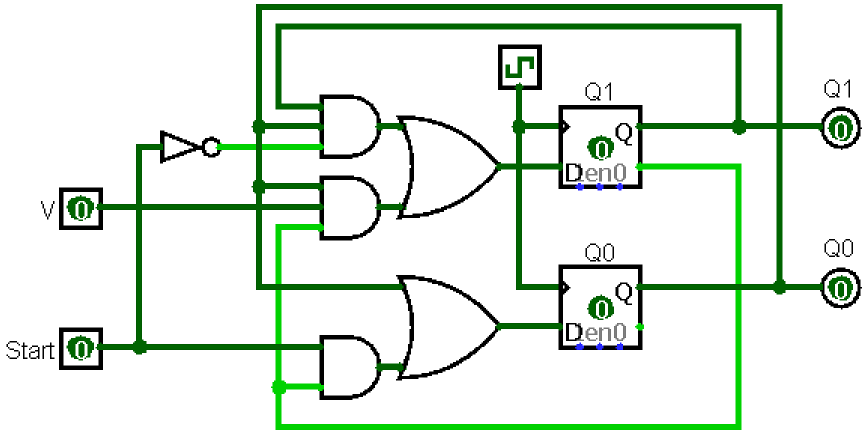


Figure 11: State Circuit

In the upcoming figures, I will represent some circuits that operate inside the DAC. In the end, I will provide Control Unit and Data Path properly.

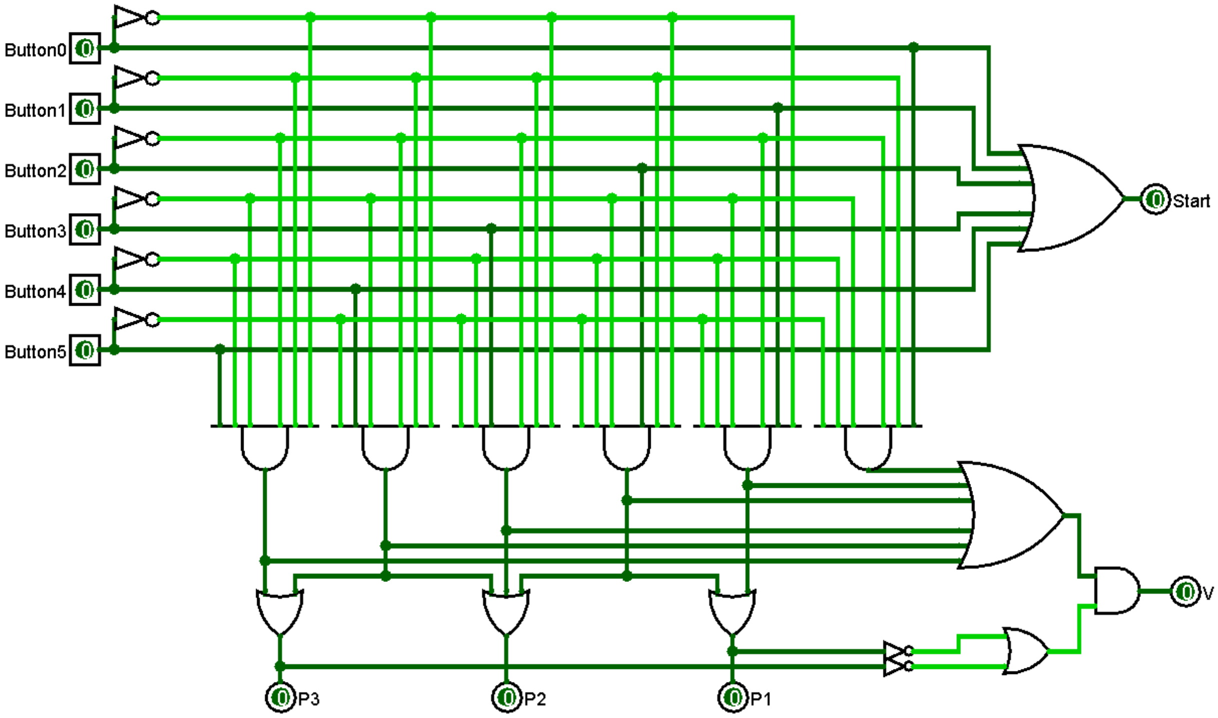


Figure 12: Start, V and P1, P2, P3 Constructor Circuit

Observe that both Start, and V is 0. Let us press one of the buttons:

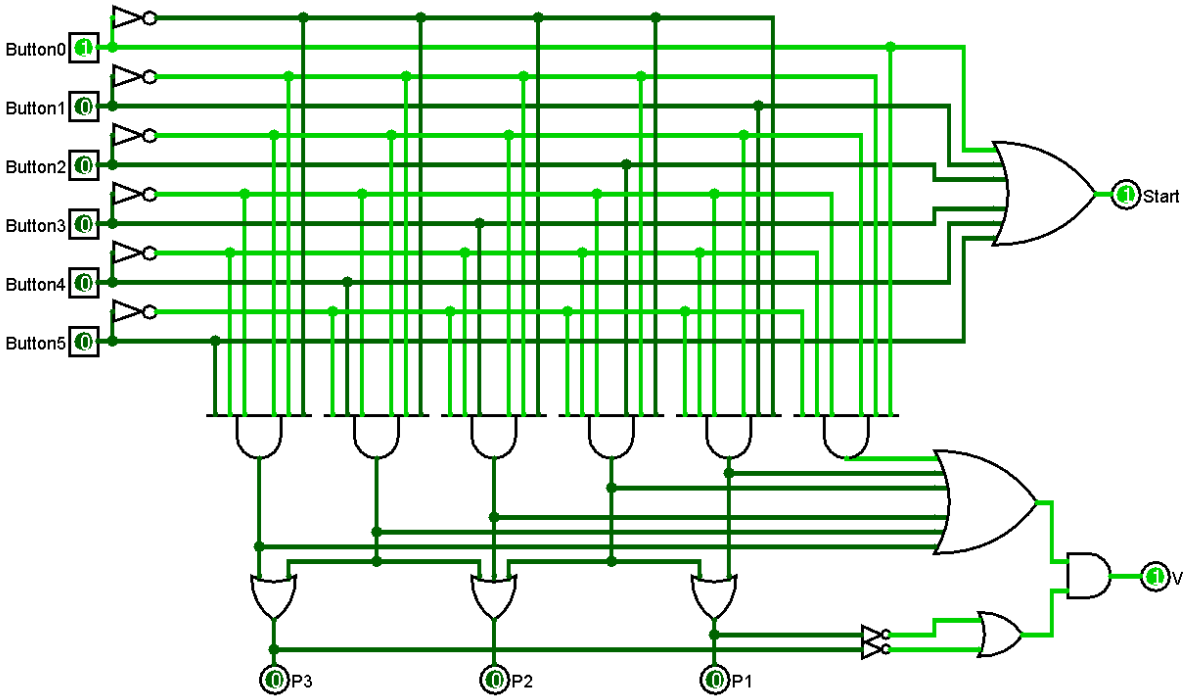


Figure 13: Circuit in Fig. 12 with Button0 pressed

Start was controlling whether any button is pressed. When B0 is pressed, it turned on. Also, since there is only one button pressed and P=000 is a valid input, V=1.

Observe the below case when two buttons are pressed:

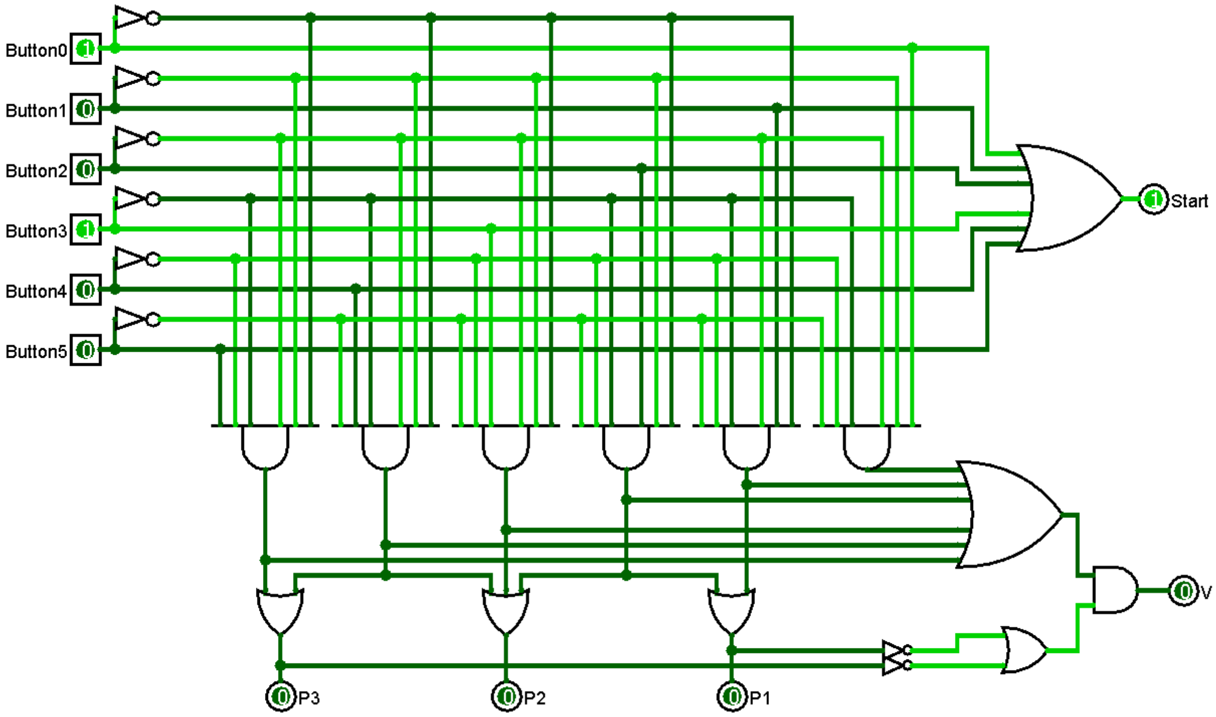


Figure 14: : Circuit in Fig. 12 with Button0 and Button3 pressed

Because there is at least one pressed button, Start signal is turned on. However, this is not a valid input (Pressing two buttons at the same time is not acceptable), so V=0. Therefore, the system is still in MAP & COMPUTE state, waiting for an acceptable button input.

Roughly, Datapath is as follows:

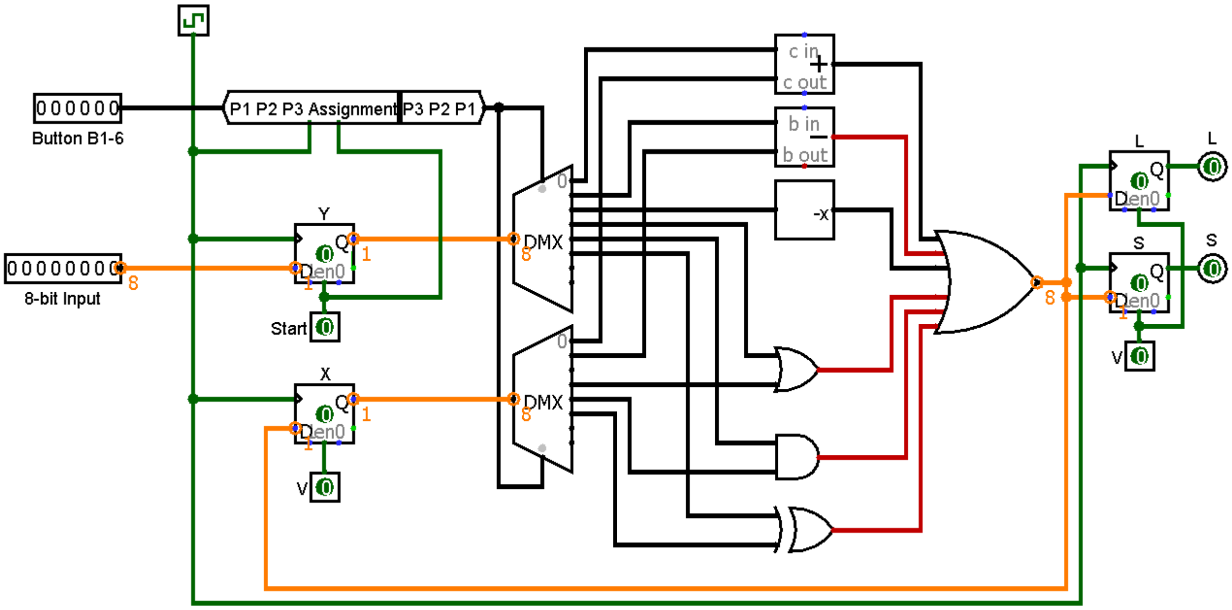


Figure 15: Data Path (main)

Button inputs go into P1 P2 P3 Assignment block, which I have just provided above in the previous pages. As extra, I have added clock input and Load input (Start), so that it can be controlled. This P3 P2 P1 the becomes the inputs of the Demultiplexers that are used to choose the operation.

8-bit input is transferred is transferred to Y via D flip-flop. It is driven by clock and Start input, just as the case of P1, P2, P3 Assigner. Then, this Y value goes into the first Demodulator.

X is also clock driven, but different from the other two, it is controlled by V input. So, if the new button input is not valid then X will not be updated, meaning that it keeps displaying the last result. X value then goes into the second Demodulator.

For both X and Y's D flip-flops, please note that this is a schematic. I know that D flip-flop does not work with an input of 8-digits. In an expanded version, each input bit D_i should go into separate D flip-flops, thus generating Y_i as a result. Similarly, each result bit X_i should go into separate D-flip flops, thus generating X_i.

Demultiplexers lead X and Y to the correct operator. Demultiplexers perform this decision by considering $P_3P_2P_1$ select bits they take into. I have clearly defined the operations and the operations that correspond to them in Table 2. I have considered the output select bits of demultiplexers in this order (from up to down): $\{000, 001, 011, 010, 110, 100, 101, 111\}$. When one operation is selected, the rest demultiplexer outputs become 0. $0 \vee 0 = 0 \wedge 0 = 0 \oplus 0 = 0 - 0 = 0 + 0 = \overline{0} = 0$. Therefore, they do not have any effect in the output of the large OR at the end of the circuit.

At last in the end of the circuit, there are D flip-flops for L and S, controlling Led Outputs and Seven Segment Display. They are updated at each clock cycle and as long as the button input is valid (V=1).

I have already provided the circuit design for $Start, V$ and P_1, P_2, P_3. I will not provide them again. Let me just write the equations representing them. V is already calculated in terms of P_i.

$Start$ signal does not have such safe conditions. When any of the buttons (one or more) is pressed, Start becomes 1. Therefore, its equation is simpler when compared to V and as follows:

$Start$ Equation

$$Start = B1 + B2 + B3 + B4 + B5 + B6$$

Finally, we write the equation of P_1, P_2 and P_3:

$P1, P2, P3$ Equations

$$P_3 = B0'B1'B2'B3'B4'B5 + B0'B1'B2'B3'B4B5'$$

$$P_2 = B0'B1'B2'B3'B4B5' + B0'B1'B2'B3B4'B5' + B0'B1'B2B3'B4'B5'$$

$$P_1 = B0'B1'B2B3'B4'B5' + B0'B1B2'B3'B4'B5'$$

Now, I think that I do not have to show the circuit inside D flip-flops since we have never done them in the examples too. Instead, I will draw the circuits inside Demultiplexers, Addition, Subtraction and Negation blocks. By providing their circuit diagrams, I believe that there will not be any black box left to explain.

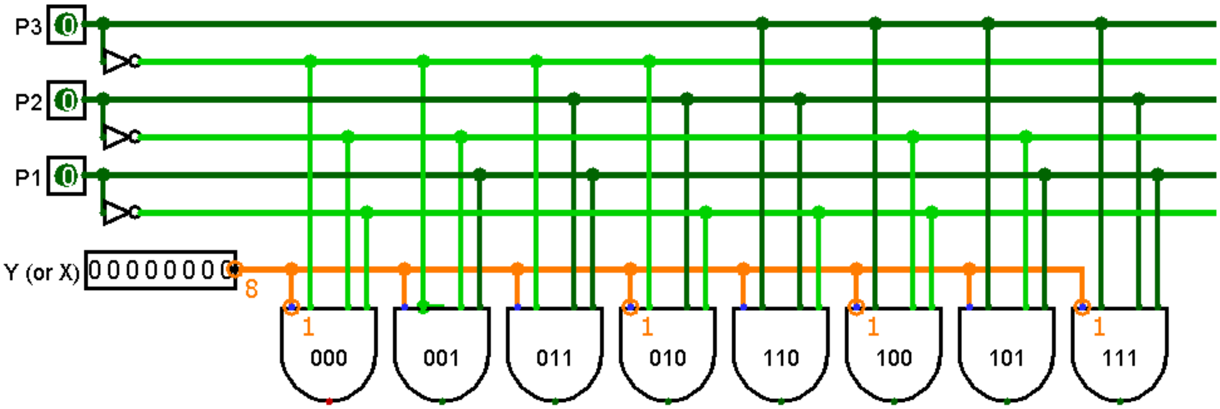


Figure 16: Demultiplexer

Above, I have provided the circuit inside the demultiplexer. Again, this is mostly a representation. In fact, AND gates' inputs cannot be 1 bit and 8-bit at the same time. Either we convert P_i to 8-bit (can be easily done by repetition of the bits) before putting into AND gates, or for each bit, we create copies of the system above.

Now, let us focus on the subtractor. Here, our aim is to construct an 8-bit subtractor.

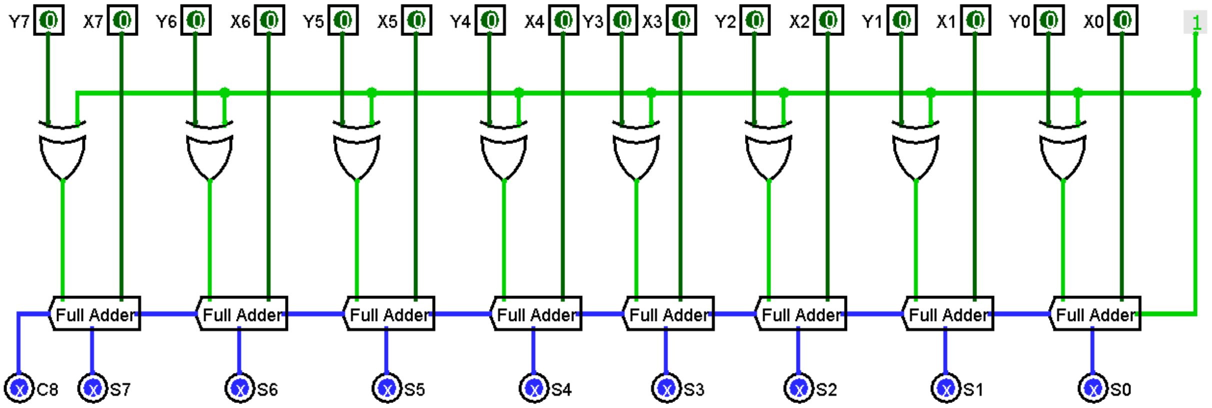


Figure 17: 8-bit Subtractor

where Full Adder component is as follows (from lecture notes):

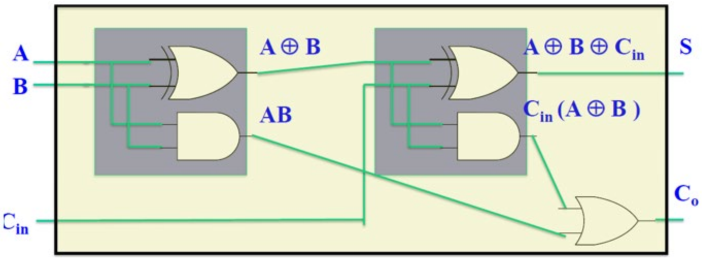


Figure 18: Full Adder

8-bit adder is also similar to the subtractor. This time, $c_0 = 0$, so we can eliminate XOR gates in the subtractor design.

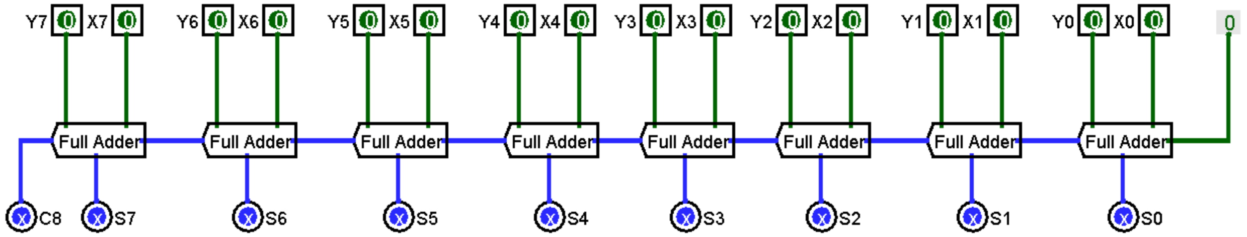


Figure 19: : 8-bit Adder

At last, we are to construct Negation block.

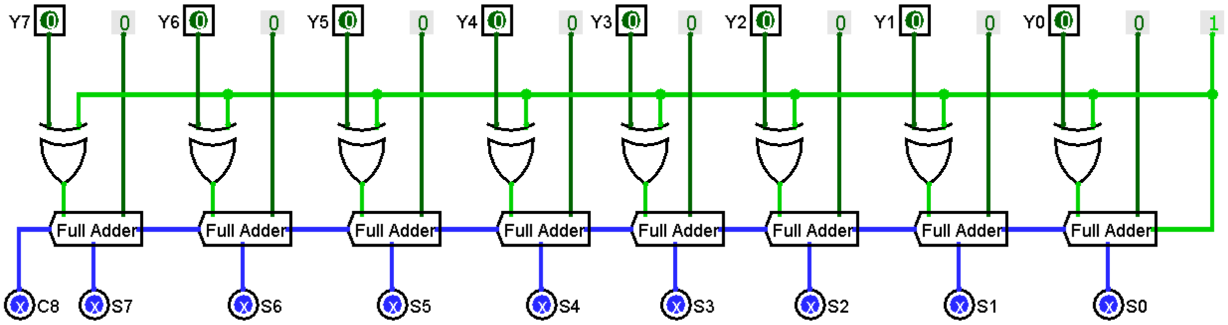


Figure 20: 8-bit subtractor

With this 8-bit negator, I have completed the explanation of the block used in the Datapath. Having explained all details of the data path, I will add here an alternative data path, which reduces cost by eliminating one of the demultiplexers and OR gate. The operation choice is done after all the operations are calculated in this design, so I am not sure whether it is optimal or not. However, it should reduce cost in terms of being decreasing the number of gates used. See the circuit below:

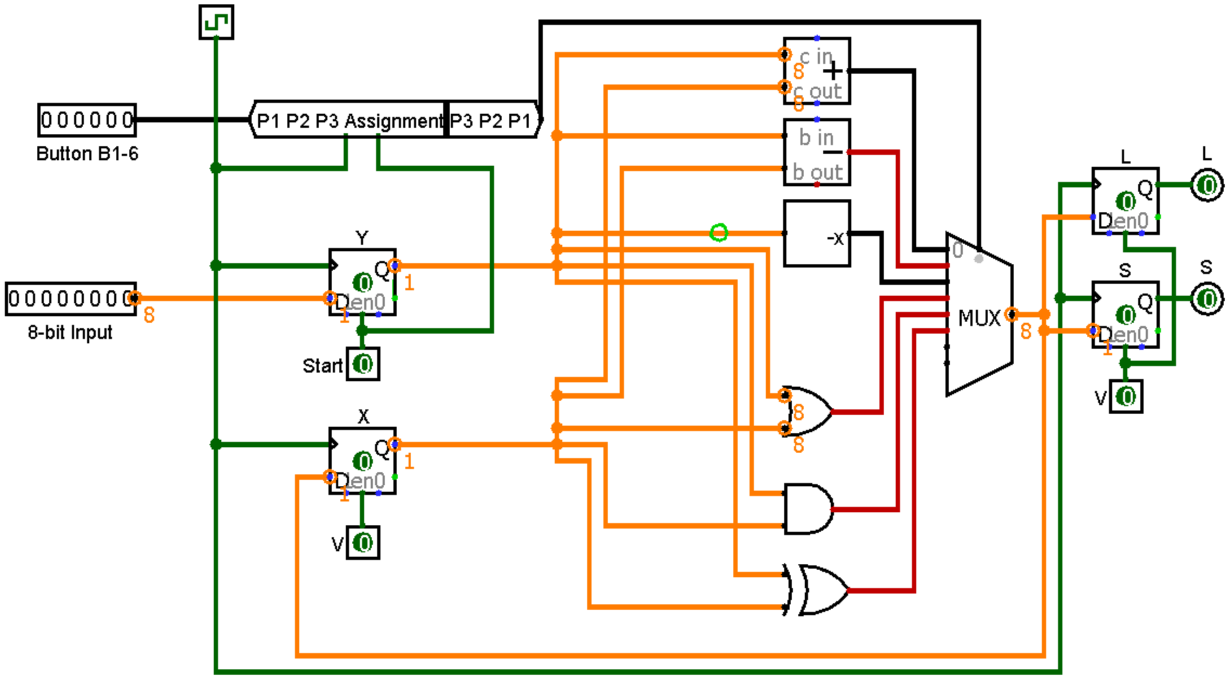


Figure 21: Data Path (alternative)

No further explanation is necessary since the blocks in this design are mostly unchanged. DMX is replaced by a MUX because with this design, circuit chooses the desired operation at end.

As one can see in Figure 15 and Figure 21, I have used D flip-flops at the end of the circuit to update L and S. We can relax this condition and let the result be displayed on the LEDs and Seven-Segment display directly. This can further reduce cost. Please note that. For that case, I will provide the circuits for these assignments. One can construct LED and Seven-Segment assignment circuits. The first one is straightforward. Using the notation *M* for the 8-digit output of MUX in Figure 21:

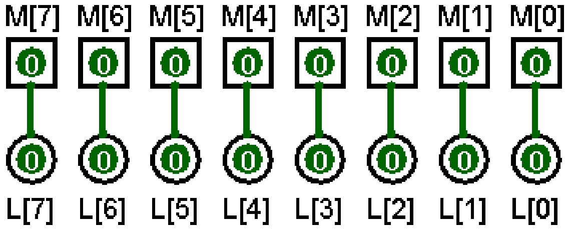


Figure 22: LED Assignments

7-Segment Display assignment is also straightforward, but a bit more detailed. Again, M is assigned to 7 LEDs of the display. But for example, if the result is 00000000, we light up a, b, c, d, e, f, but not g. So, the result is shown on the 7-

Segment display in decimal. This increases readability but requires the assignment of each segment individually for each digit. Furthermore, below, I will provide this for the 2-bit case only (range is {0,1,2,3}). For the 8-bit case, this manual assignment by using simple gates requires a great amount of time and not necessary for the purpose at the moment. So, suppose M has 2 digits and understand the circuit below. Then, for higher digit cases, same logic applies with more gates and assignments.

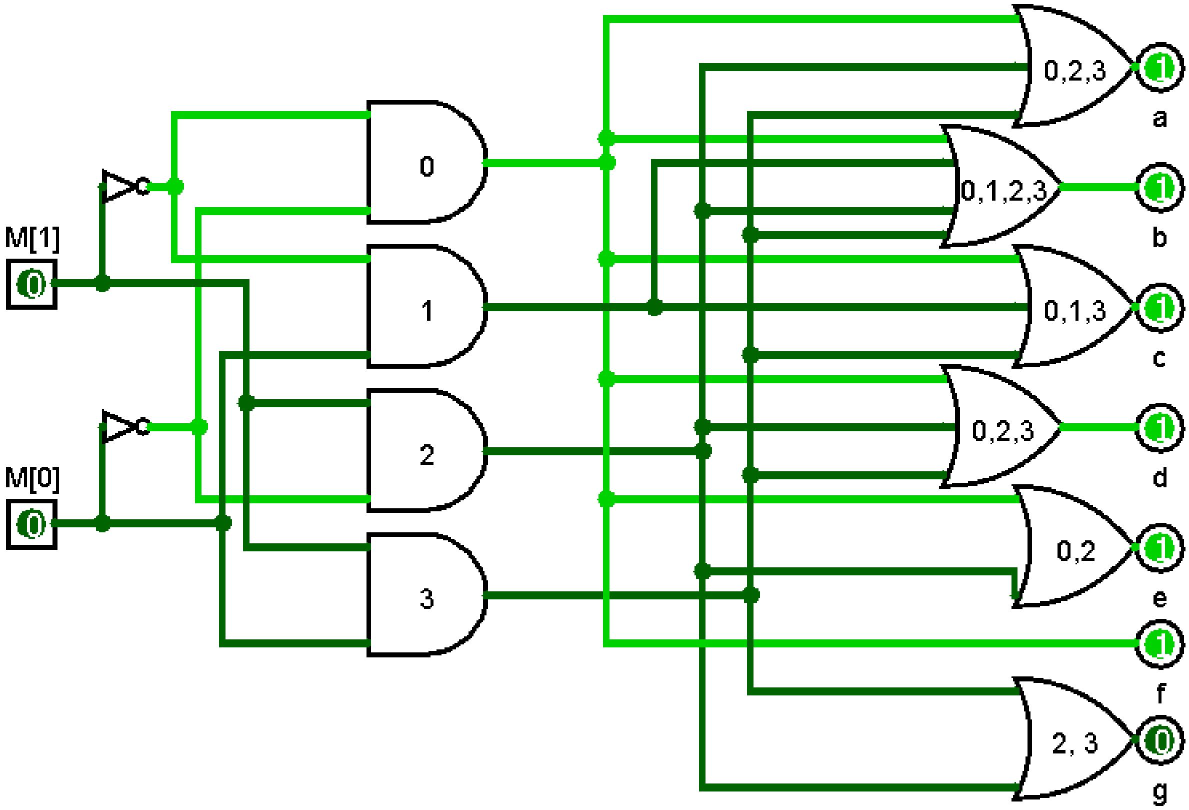


Figure 23: Seven-Segment Display LED Assignments for 2 bits

From this figure, one can verify that 0 lights up a, b, c, d, e, f but not g.

Codewise, SSD it is easier to implement using if-else statements. Moreover, seven segment display for 8-bit numbers code is already available to us (ELEC 204 takers) in the GitHub webpage of Mr. Arash. So, as long as the way to construct a Seven-Segment display is understood, the rest is just straightforward.

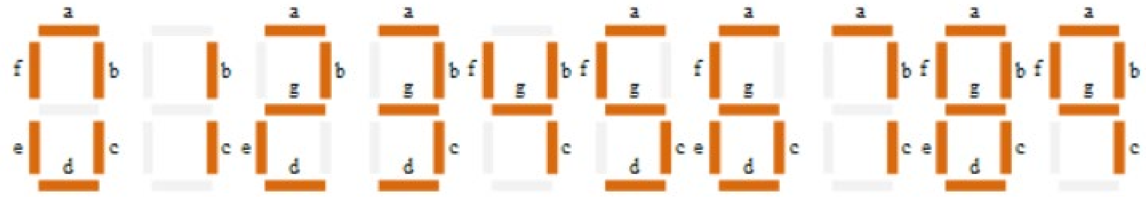


Figure 24: 7 - Segment Display LED assignments[1]

Considering these assignments, circuits with for number with any digits can be constructed by following the logic path described above. So, I will end the discussion about 7-Segment Display here.

Up to this point, I have provided all the details about Data Path, and provided its circuit design too. Now, at last, I will express the control unit as we did in the class:

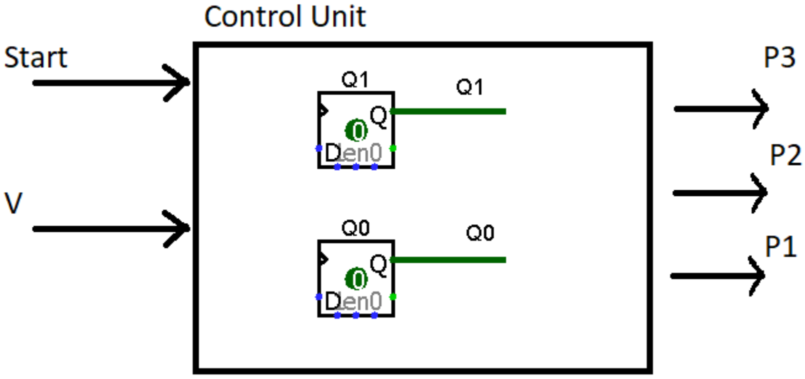


Figure 25: Control Unit

I have already provided the equations describing P_is in the previous parts. Since all P_is occur in MAP & COMPUTE state we can also express them in terms of Q_1 and Q_0. Considering MAP&COMPUTE state is assigned to 01, we can write the condition $\bar{Q}_1 Q_0$ for each P_i.

[1] Image is taken from https://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html

Extra Functionalities (LAB Project)

See Figure 21. MUX at the end still has 2 inputs empty. So, we can add 2 extra functionalities to this design, without changing it. In this section, I will provide 2 new extra functionalities. One of these functionalities will *multiplication* since it is an essential operation missing. Also, we have implemented division in the Question 1, so why would not we also construct multiplication here for a complete picture? The second will be *Lazy Caterer's Sequence*. It is an amusing formula that can be used to find the maximal number of pieces you can slice a pizza with n cuts.

First, let me first put these new operations into the data path I have already provided above in Figure 21.

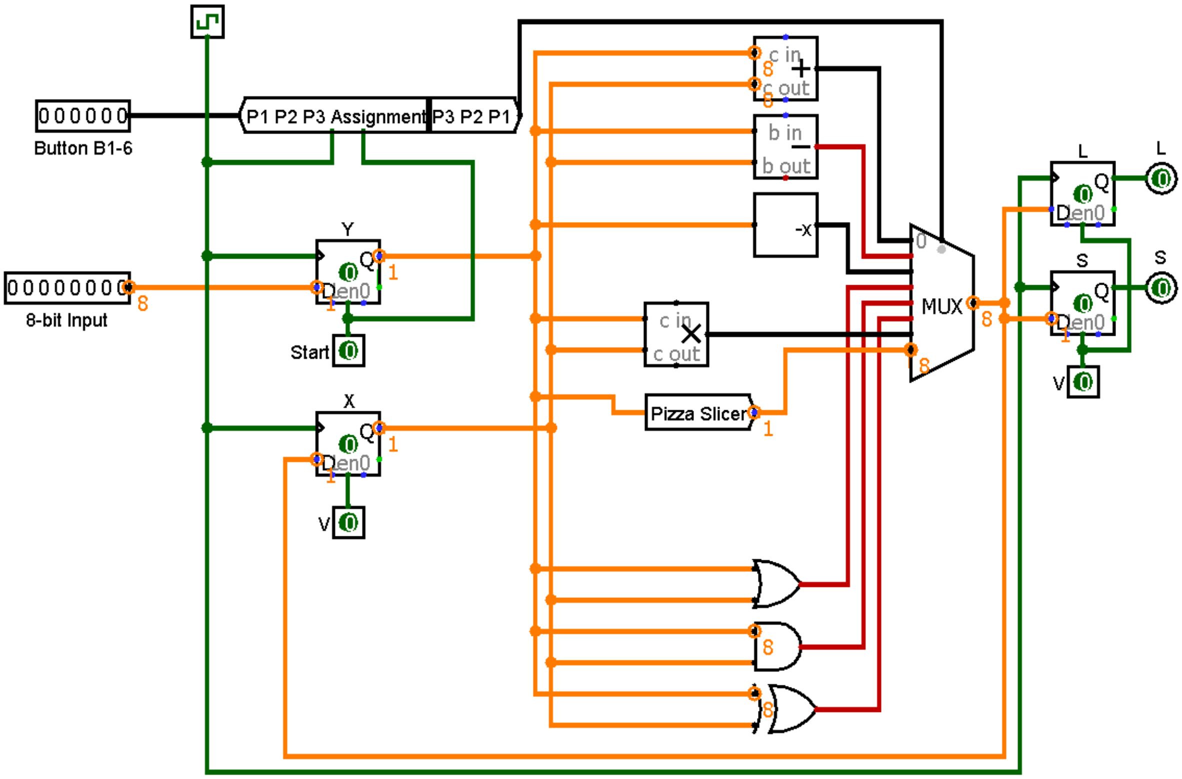


Figure 26: Updated data path

It is easy to put the operations boxes as in Figure 26, but now we have to introduce how the circuit performs these operations with the use of simple gates. From Table 3, we read that states 101 and 111 are unassigned. So, I assign 101 to the multiplication operation, and 111 to the Pizza Slicer. With these new assignments, Validness check variable V is now restricted only to single button push check. Thus, its equation reduces to:

$V = B0'B1'B2'B3'B4'B5 + B0'B1'B2'B3'B4B5' + B0'B1'B2'B3B4'B5' + B0'B1'B2B3'B4'B5' + B0'B1B2'B3'B4'B5' + B0B1'B2'B3'B4'B5'$

This does not change the state diagram in Figure 10 and the state table in Table 4. Therefore, the rest does not change. Same K-maps are to be constructed, along with the same state equations. In result, same Control Unit and the updated Data Path I have provided above are valid. I only must update Figure 12: Start, V and P1, P2, P3 Constructor Circuit, just the part with V though. The extra check condition for V is removed now, as I have explained just above and provided the new equation for V. When Figure 12 is updated, it becomes as follows:

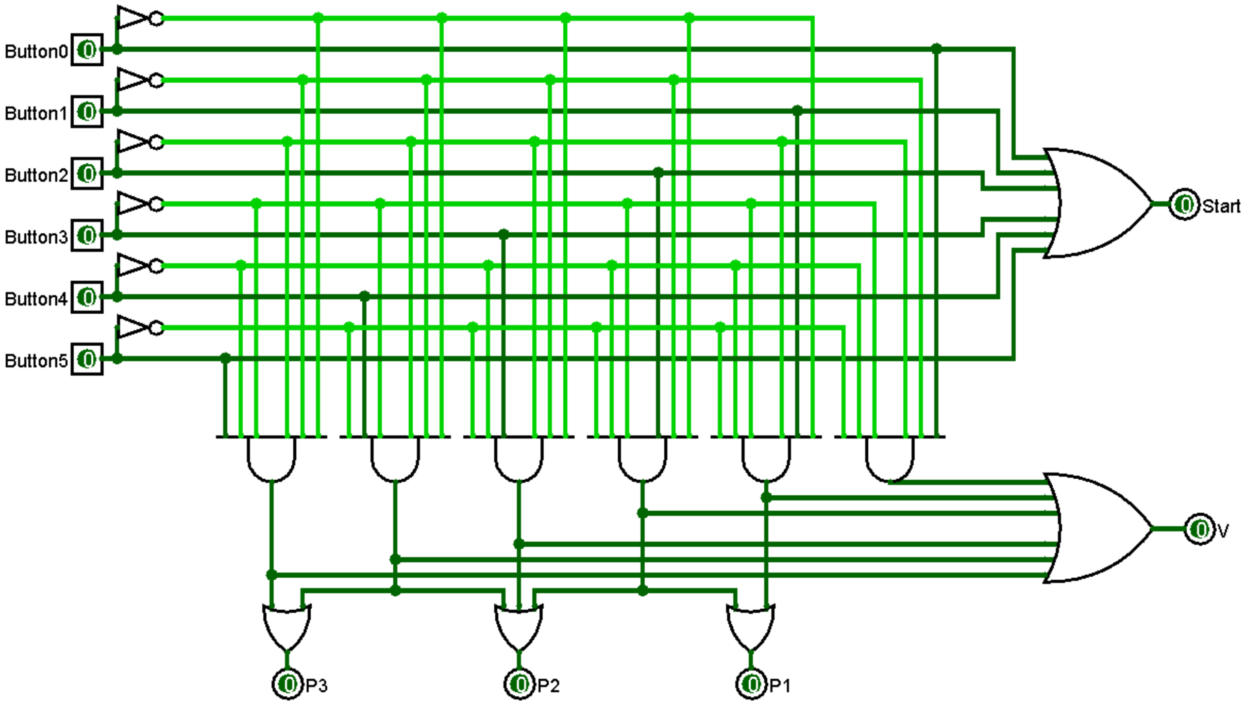


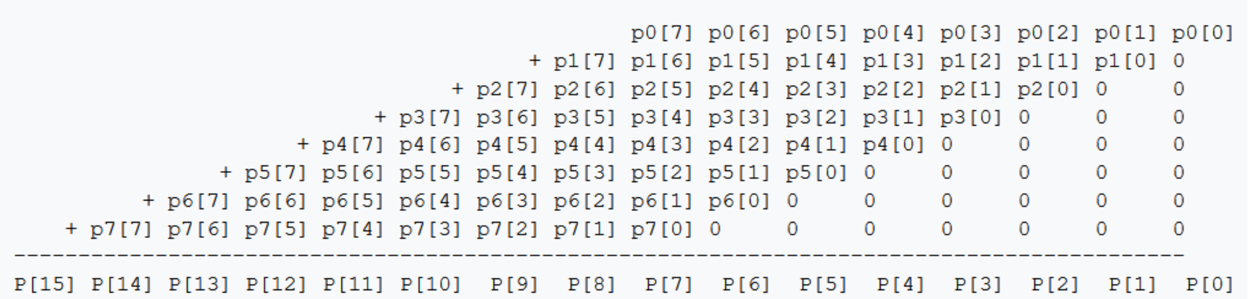
Figure 27: Updated Start, V and P1, P2, P3 Constructor Circuit

After adding these necessary modifications to the circuit, let me start with the multiplication operation.

1. Multiplication

8-bit multiplication is a bit complicated process when it is compared to the other operations in 2’s compliment signed arithmetic operations such addition / subtraction. Unsigned version is a bit simpler and easier to understand and implement. However, since we work with 2’s complement signed 8-bit numbers, I will not cut corner and implement this version.

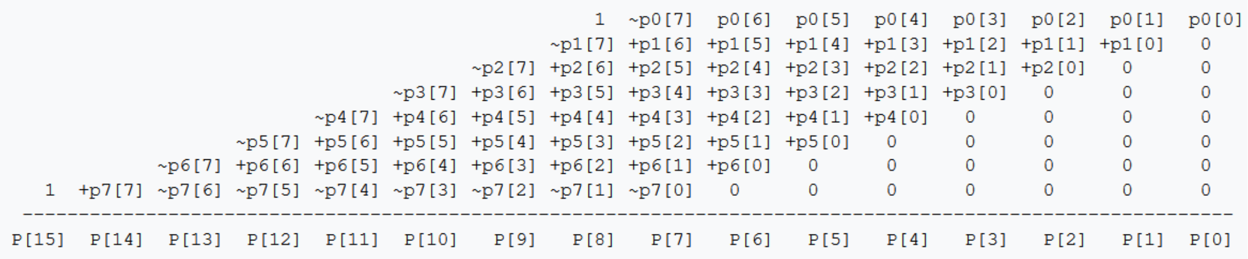
Consider we have two 8-bit numbers A and B. For regular multiplication, B_0 is multiplied with the all bits of A first. Then, B_1 is multiplied with A, and written below the preceding operation with 1 shift to the left. This is performed for all bits of B and then summed. At least this is what happens in the unsigned version.



```
                                                        p0[7]  p0[6]  p0[5]  p0[4]  p0[3]  p0[2]  p0[1]  p0[0]
                                               + p1[7]  p1[6]  p1[5]  p1[4]  p1[3]  p1[2]  p1[1]  p1[0]  0
                                        + p2[7]  p2[6]  p2[5]  p2[4]  p2[3]  p2[2]  p2[1]  p2[0]  0      0
                                 + p3[7]  p3[6]  p3[5]  p3[4]  p3[3]  p3[2]  p3[1]  p3[0]  0      0      0
                          + p4[7]  p4[6]  p4[5]  p4[4]  p4[3]  p4[2]  p4[1]  p4[0]  0      0      0      0
                   + p5[7]  p5[6]  p5[5]  p5[4]  p5[3]  p5[2]  p5[1]  p5[0]  0      0      0      0      0
            + p6[7]  p6[6]  p6[5]  p6[4]  p6[3]  p6[2]  p6[1]  p6[0]  0      0      0      0      0      0
     + p7[7]  p7[6]  p7[5]  p7[4]  p7[3]  p7[2]  p7[1]  p7[0]  0      0      0      0      0      0      0
 -------------------------------------------------------------------------------------------------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]  P[9]   P[8]   P[7]   P[6]   P[5]   P[4]   P[3]   P[2]   P[1]   P[0]
```

Figure 28: 8-bit binary multiplication[2]

This operation is simple and easy to understand as I have noted. I have provided this since it is easier to understand the 2's complement signed version after reviewing this and understanding the notation.



```
                                                        1       ~p0[7]  p0[6]   p0[5]   p0[4]   p0[3]   p0[2]   p0[1]   p0[0]
                                                        ~p1[7]  +p1[6]  +p1[5]  +p1[4]  +p1[3]  +p1[2]  +p1[1]  +p1[0]  0
                                                ~p2[7]  +p2[6]  +p2[5]  +p2[4]  +p2[3]  +p2[2]  +p2[1]  +p2[0]  0       0
                                        ~p3[7]  +p3[6]  +p3[5]  +p3[4]  +p3[3]  +p3[2]  +p3[1]  +p3[0]  0       0       0
                                ~p4[7]  +p4[6]  +p4[5]  +p4[4]  +p4[3]  +p4[2]  +p4[1]  +p4[0]  0       0       0       0
                        ~p5[7]  +p5[6]  +p5[5]  +p5[4]  +p5[3]  +p5[2]  +p5[1]  +p5[0]  0       0       0       0       0
                ~p6[7]  +p6[6]  +p6[5]  +p6[4]  +p6[3]  +p6[2]  +p6[1]  +p6[0]  0       0       0       0       0       0
1       +p7[7]  ~p7[6]  ~p7[5]  ~p7[4]  ~p7[3]  ~p7[2]  ~p7[1]  ~p7[0]  0       0       0       0       0       0       0
 ----------------------------------------------------------------------------------------------------------------------------
P[15]   P[14]   P[13]   P[12]   P[11]   P[10]   P[9]    P[8]    P[7]    P[6]    P[5]    P[4]    P[3]    P[2]    P[1]    P[0]
```

Figure 29: 8-bit signed 2's complement system multiplication[3]

So, the procedure to be followed is as follows:

1) Calculate 1st row as usual, negate the MSB and add an extra bit of 1 as the new MSB.
2) In the intermediate rows, calculate and write rows as usual, then negate MSB.
3) Calculate the last row as usual. Then, negate all bits, except the MSB. As in the first row, add an extra bit of 1 as the new MSB.
4) Sum all.

Since we work with 8-bit numbers, it will be a long and kind of complex circuit, but the logic of the circuit is explained above and is not actually much a complicated one.

As a last remark before the circuit design, note that output is 16-bit long. FPGA board has only 8 LEDs. Therefore, the result on the LEDs should not be of interest and only the Seven-Segment Display should be considered. I can implement this

[2] https://en.wikipedia.org/wiki/Binary_multiplier#Unsigned_numbers

[3] https://en.wikipedia.org/wiki/Binary_multiplier#Signed_integers

extra condition on the Data Path design, but it is really not so necessary and complicates the design for such an obvious thing. So, I only suggest the user to consider the Seven-Segment output for this operation and ignore the LED outputs.

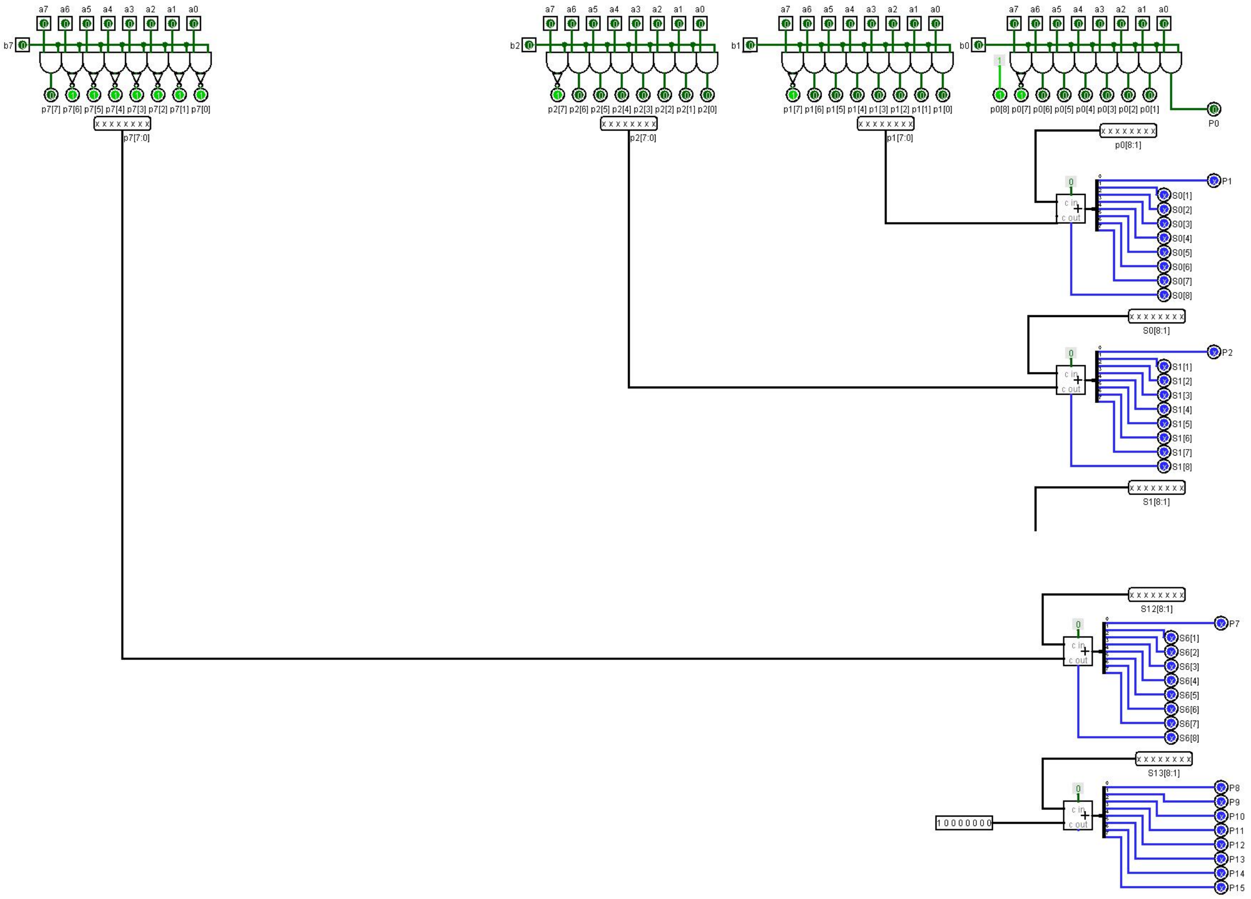

a7 a6 a5 a4 a3 a2 a1 a0
b7
p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0]
xxxxxxxx
p7[7:0]
b2
p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0]
p2[7:0]
b1
p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0]
p1[7:0]
b0
1
p0[8] p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1]
P0
p0[8:1]
0
c in
c out
P1
S0[1] S0[2] S0[3] S0[4] S0[5] S0[6] S0[7] S0[8]
S0[8:1]
P2
S1[1] S1[2] S1[3] S1[4] S1[5] S1[6] S1[7] S1[8]
S1[8:1]
S12[8:1]
P7
S6[1] S6[2] S6[3] S6[4] S6[5] S6[6] S6[7] S6[8]
S13[8:1]
10000000
P8 P9 P10 P11 P12 P13 P14 P15

Figure 30: 8-bit signed 2's complement multiplication (cropped)

The figure may not be clearly visible, so I suggest the reader to zoom in. I have skipped some of the intermediate multiplication operations since the are the same as I have provided representative circuits above. I did not want to complicate the figure more since it is difficult to read even in this simplified view. In the above circuit, first B_0 is multiplied with A at the up-right corner. The LSB of this operation is assigned to $P0$. Then, MSB is negated and 1 is added to the left of the MSB and a new 8-bit integer is formed. Then, this 8-bit integer is summed with the result of the multiplication of $B_1 \times A$ (after negating its MSB). The LSB gives out $P1$. With C_{out}, this operation gives us an 8-bit number when we exclude $P1$. Then, this 8-bit integer is summed with $B_2 \times A$ (after negating its MSB). Same operations as above applied. Up to the last line in Figure 29, same intermediate operations are applied. In the last line, all digits are negated except the MSB, so it is kind of the opposite of what is done up to this point. As usual, this is summed with the result of the latest summation. The LSB of this summation gives out $P7$. Now, there is only 1 operation is left. We again construct the 8-bit integer by using C_{out} of the summation and excluding $P7$. Then, at last, we sum this with 10000000. The result of this summation gives us all the left bits of the multiplication, ranging from $P8$ to $P15$. Therefore, the multiplication operation ends. Note that in the figure, I have put the bits that form the result of the multiplication to the very right of the circuit. This way, I have wanted to make the result more understandable and easier to detect. This 16-bit result cannot be shown on the LEDs since we do not have 16 LEDs on the FPGA Board as I have mentioned. Therefore, the 7-Segment Display result should be considered for the result.

Blow, I provide the full (complete) version of the multiplier in Fig. 30. It will be difficult to read, so Fig. 30 is satisfactory enough, you can skip this image below.

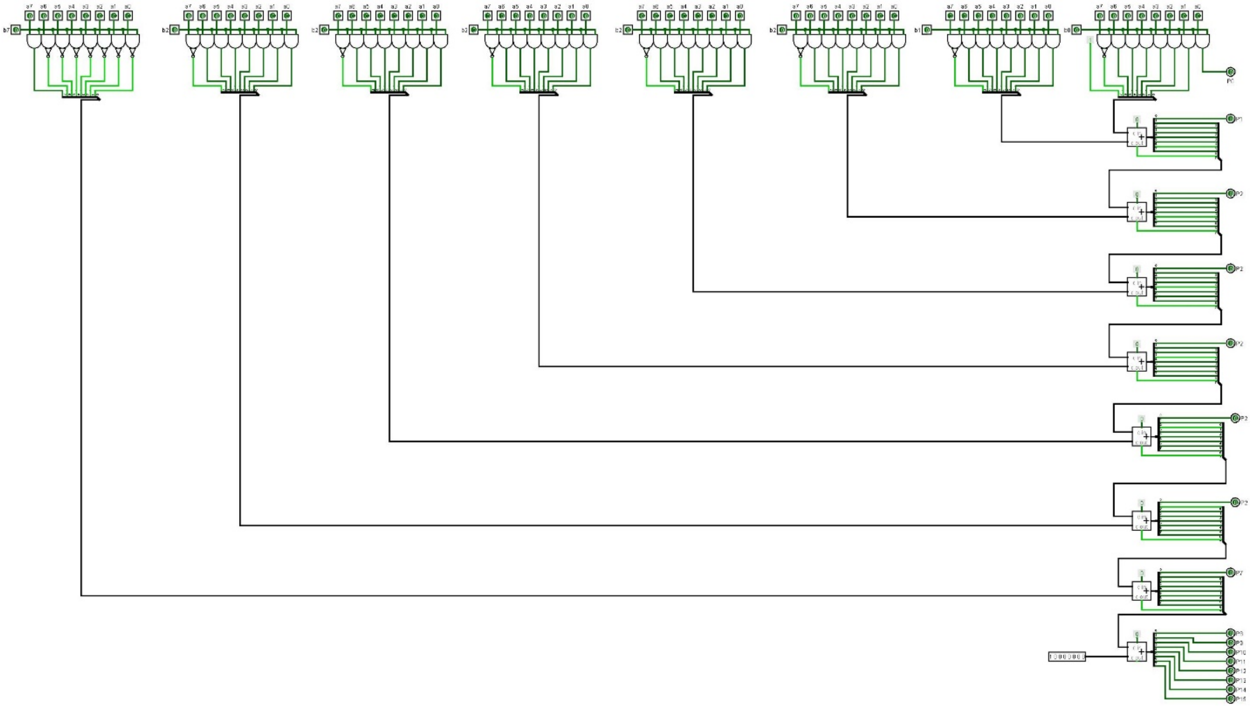


Figure 31: : 8-bit signed 2's complement multiplication (complete)

2. Lazy Caterer’s Sequence (Pizza Slicer)

Lazy Caterer’s Sequence (formally, known as central polygonal numbers) is basically gives us a formula that describes how many pieces we can split a circle with a fixed number of *straight* cuts at most. It is entertaining to consider the circle as a pizza (or a pancake, bread, etc.) and the cuts as knife cuts. It does not guarantee that the pieces are going to be equal. Therefore, it may not be a fair way of slicing a pizza, but at least it gives at most how many people can eat a pizza (supposing we can only slice the pizza with a limited number of straight cuts, because, why not right?). It can be a challenging activity, and in real life, a person who has the FPGA Board that performs this operation can challenge anyone in a pizza slicing activity.

The formula is as follows:

$$p = \frac{n^2 + n + 2}{2}$$

where p is the number of pieces and n is the number of cuts. As I have noticed, division is a very much complicated operation to perform with simple circuit elements. Therefore, I will provide a circuit that can calculate $2p$, that is, twice of the maximum number of pieces we can get. You can think it as we have 2 pizzas, so we multiply the result with 2. Thus, the circuit I am about to provide is going to calculate the maximum number of slices that we can extract out of **2** pizzas with a fixed number of cuts, n.

Since I have provided the multiplication operation in the previous part, I will use a simple block for the sub-circuit that performs the multiplication operation. This circuit has a one input, n. I suppose this is the 8-bit 2’s complement number provided by the user with the use of switches. So, n here corresponds to Y in the data path in Figure 26. The operations are to be done are multiplication, summation, and another summation in order. Before moving on to the circuit, it may be beneficial to remind that 2 in the above equation is represented with **00000010** in 2’s complement signed 8-bit representation.

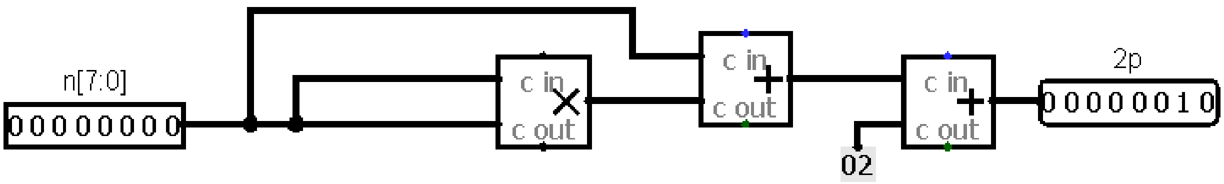


Figure 32: Pizza Slicer Circuit Design

In the design above, $n[7:0]$ is the input 8-bit signed integer. 8-bit signed multiplication operation in 2’s complement system is explained, and its circuit design is in Figure 30. 8-bit signed addition operation in 2’s complement system is explained, and its circuit design is in Figure 19. The integer 2 is 00000010 in 8-bit signed 2’s complement system. Therefore, there is nothing left out, and this operation can work properly. See some of its outputs below:

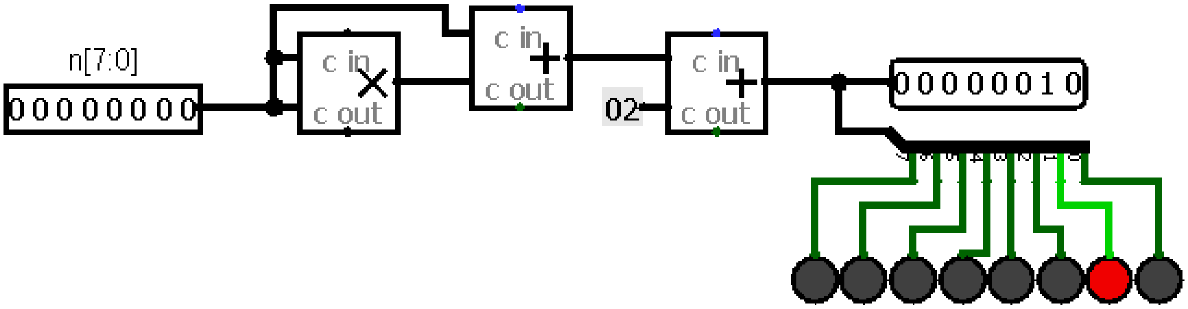


Figure 33: Pizza slicer for n=0

Zero cuts split 2 pizza into two slices, which is obvious.

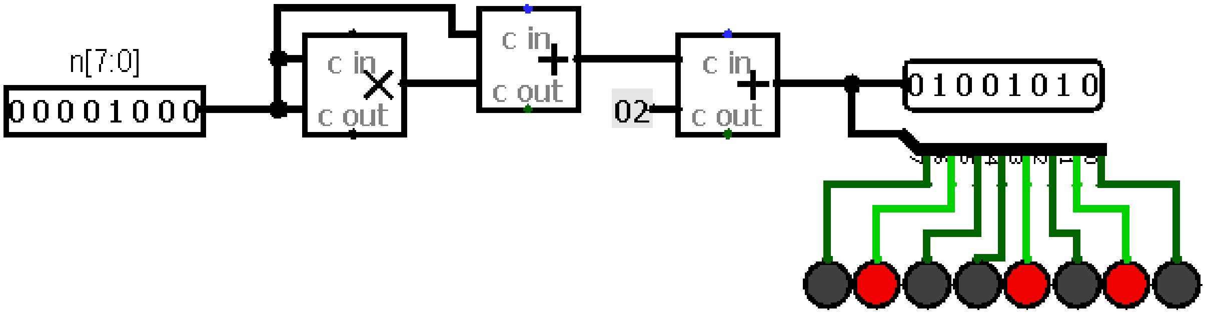


Figure 34: Pizza slicer for n = 16

With 8 cuts, one can have at most 74 slices out of 2 pizzas.