# Accumulator Design

Name: Jonathan J. Rodriguez

CSC343 – Computer Organization Lab, Spring 2020

Instructor: Izidor Gertner

The City College of New York

# Table of Contents

## Objective

The objective of this laboratory exercise is to use both the DE2 and the newer DE1-SOC boards to display binary numbers onto seven segment displays. However, we will also be adding two binary inputs from the board onto N-Bit Registers and create an accumulator to continuously add up inputs. The use of Quartus 13.0 (for DE2) and Quartus 16.1 (for DE1-SOC) software will allow us to design electronic circuits that utilizes the switches on the boards to create the output that is desired.

## Board Specifications & Software Used

1. Altera DE2 Board (Quartus II 13.0 sp1)
2. Altera DE1-SOC Board (Quartus Prime 16.0)

## Design Specifications & Functionality

### *N-Bit Shift Register*

RODR_NBit_Shift_Register.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RODR_NBit_Shift_Register is
   generic(nbits : natural);
   Port (RODR_CLK : in STD_LOGIC;
         RODR_SET : in STD_LOGIC;
         RODR_CLR : in STD_LOGIC;
          RODR_INS : in STD_LOGIC_VECTOR (7 downto 0) := (Others => '0');
          RODR_INP: in std_LOGIC_VECTOR (nbits-1 downto 0) := (Others => '0');
          Load    : in std_logic := '0';
          RODR_OUT : out STD_LOGIC_VECTOR (nbits-1 downto 0));
end RODR_NBit_Shift_Register ;
```

*Figure 1: First part of VHDL code for N-Bit Shift Register*

The first thing that both the DE2 and DE1-SOC boards will encounter is the N-bit shift register. We will need two of them because of how we will be taking their outputs into an adder/subtractor for computation. We will also need one more after the adder for use in the 2-to1 multiplexer. However, they will use two different modes of inputs, parallel or serial shift.

To create a register, we will first need **RODR_CLK**, which will determine when the bits stored within the register will be pushed out to the adder for computation. We also have both an asynchronous **RODR_SET** and **RODR_CLR** inputs that will set the output of the registers to be either all 1's or 0's, respectively. **RODR_INS** will only be used when using serial outputs. This

means that we will be shifting in 8 bits from the board from MSB to LSB. **RODR_INP** will only be used when using parallel outputs. This means that it will automatically intake N bits straight from the previous electronic device into the register itself and output it. **Load** will be used to either use serial or parallel outputs (‘0’ for Parallel, ‘1’ for Serial). **RODR_OUT** will be the output from the register itself into the adder/subtractor.

```
architecture Behavioral of RODR_NBit_Shift_Register is
signal tempOut : std_LOGIC_VECTOR (nbits-1 downto 0);    -- Bit storage for Shifting and Set/Clear

shared variable index: integer := nbits-8;
begin
   process(RODR_CLK, RODR_CLR, RODR_SET, RODR_INP, RODR_INS)
   begin
      if(RODR_CLR ='1' AND RODR_SET = '0') then
         tempOut <= (others => '0');
         index := nbits-8;
      elsif(RODR_SET ='1' AND RODR_CLR = '0') then
         tempOut <= (others => '1');
         index := nbits-8;
      elsif (rising_edge(RODR_CLK)) then
         if(load = '1') then
            tempOut(index+7 downto index) <= RODR_INS;
            index := index-8;
         elsif(load = '0') then
            tempOut <= RODR_INP;
         end if;
      end if;
      RODR_OUT <= TempOut;
   end process;
end Behavioral;
```

*Figure 2: Processes declared for N-Bit Shift Register*

The second process depends on **RODR_CLK**, **RODR_CLR**, and **RODR_SET**. Due to the nature of the clear and set signals being asynchronous, they will set **tempOut** to either all ‘0’ or ‘1’ and have that as its output. It will also reset the index if it has ever changed from the rising clock edge This will allow for any new inputs into the registers to start with the most significant bits. However, if the rising edge of the clock appears, we will first check **Load** to see if it is in serial or parallel mode. If it is in serial, then we shift in the bits from the switches on the board into the register, decrement the index to the 8 least significant bits, and then output it.  If it is in parallel mode, it then just outputs the result entirely.

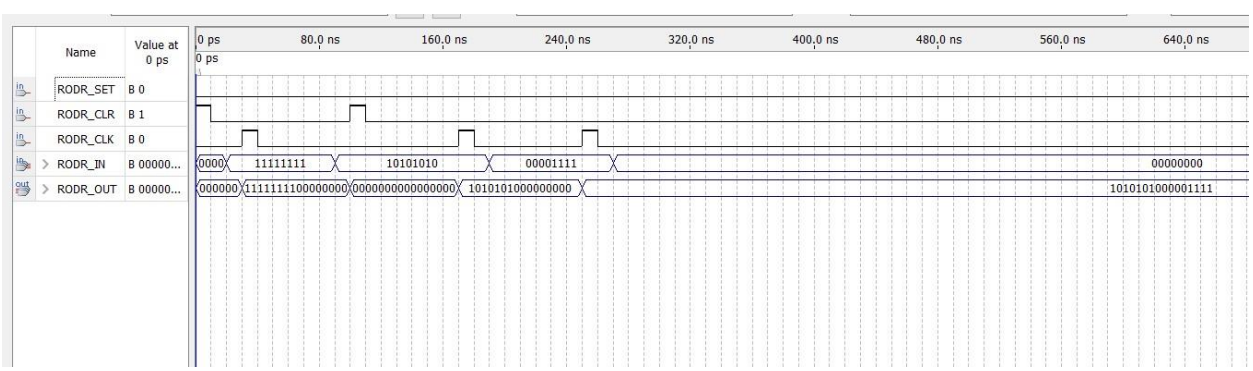


*Figure 3: Waveform for N-Bit Shift Register*

Based on this waveform, we see that there is a need for a clear signal for the register itself. This is to ensure that any random data that initially enters it will automatically be reset to

all 0's. We then proceed to use the clock to take in inputs starting with the combination of '11111111' to the first 8 most significant bits. However, as another clear signal has been set, the output will then turn back to all 0s and reset the index back to the most significant bits. Once the clock signal goes back up on the input of '10101010', it will then be placed in the MSB and another time with '00001111' as the LSB to give the result of '1010101000001111' (or 43535 in decimal).

## *N-Bit Adder/Subtractor*

RODR_Adder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------

entity RODR_Adder is

   generic(nbits: natural);
port( RODR_A:  in std_logic_vector(nbits-1 downto 0);
      RODR_B:  in std_logic_vector(nbits-1 downto 0);
      RODR_overflow: out std_logic;
      RODR_CIN : in std_logic;
      RODR_sum:   out std_logic_vector(nbits-1 downto 0));

end RODR_Adder;

-------------------------------------------------------
```

*Figure 3: VHDL Code for an N-Bit Adder/Subtractor*

In this electronic device, we will be focusing on the adder. An adder simply adds two number (or in this case two bit sequences) and produces a final result. We use a generic variable called **nbits** to be used in the master file to appoint the number of bits the entire system will have. We then have **RODR_A** and **RODR_B** be the two outputs from two n-bit shift registers. **RODR_overflow** will be used designate if the addition of both A and B binary numbers result in an overflow. **RODR_CIN** will be used to designate whether or not the device will act as either an adder or subtractor. **RODR_Sum** will be the output of the addition of both A and B.

```
architecture LogicFunction of RODR_Adder is
   Component RODR_UnsignedSigned
      generic (nbits : natural := nbits);
      PORT(RODR_SIGBIT : IN STD_LOGIC;
         RODR_IN   : IN STD_LOGIC_VECTOR (nbits-1 downto 0);
         RODR_OUT  : OUT STD_LOGIC_VECTOR (nbits-1 downto 0));
   END Component;

-- define a temparary signal to store the result

signal result: std_logic_vector(nbits downto 0);      -- Extra bit to accomodate for overflow
signal flip : std_logic_vector(nbits-1 downto 0);



begin
   Bflip : RODR_UnsignedSigned      PORT MAP(RODR_CIN, RODR_B, flip);

   process(RODR_CIN, result, RODR_A, RODR_B, flip)
   begin
      if(RODR_CIN = '0') then
         result <= ('0' & RODR_A)+('0' & RODR_B);
         RODR_sum <= result(nbits-1 downto 0);
         RODR_overflow <= result(nbits);

      elsif(RODR_CIN = '1') then
         result <= ('0' & RODR_A)+('0' & flip);
         RODR_sum <= result(nbits-1 downto 0);
         RODR_overflow <= result(nbits);
      end if;
   end process;
end LogicFunction;
```

*Figure 4: Part 2 of VHDL code for N-Bit Adder/Subtractor*

To make this work, we used the Unsigned/Signed circuit (to be explained later) that will take care of when **RODR_CIN** is ‘1’ and a subtractor is then created. We also use **result** to store the result of adding A and B temporarily before outputting it. The signal **flip** will be done to B in order to make it conduct two’s complement in a subtractor scenario. We use a “Port Map” to use the Unsigned/Signed circuit.

Inside the process, we either have an adder or subtractor functionality. If the carry in is ‘0’, then act like an adder and display the result and overflow as usual. However, If the carry in is ‘1’, we used the two’s complemented version of B and add it with A and get its sum that way. Later on we will then confirm if the output is either a negative or positive number using the MSB from the **result** along with if **RODR_Sigbit** was flipped.

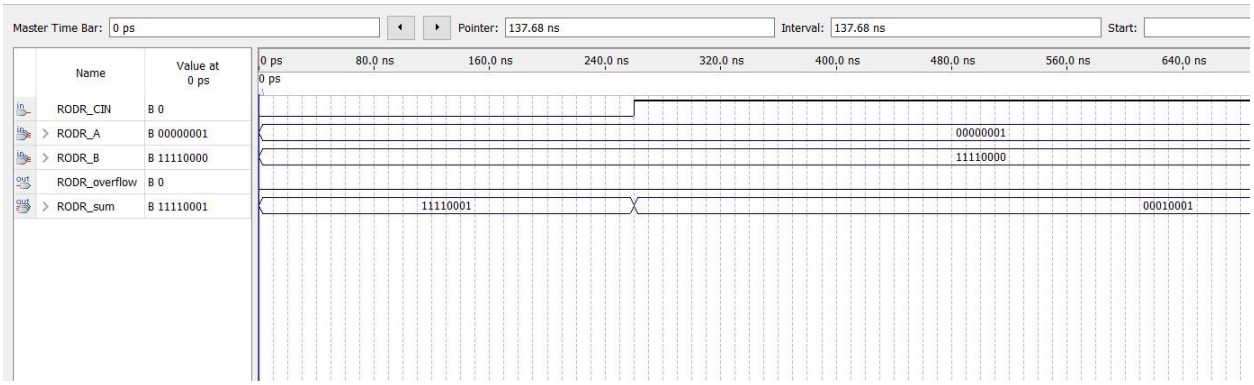


*Figure 5: Waveform for N-Bit Adder/Subtractor*

Based on this waveform, we see how an adder/subtractor works. If the carry in is ‘0’, thn we add both A and B together and simply put it into the output **RODR_sum**. However, if the carry in is ‘1’, we then must do two’s complement on B and add both A and B together again. The output is then ‘00010001’ (or 241 in binary). Because the carry in is 1, and when we do two’s complement on it again we get ‘11101111’ with the MSB being ‘1’, we get a negative sign on the seven segment display.

## *Two-to-One Multiplexer*

RODR_2TO1MUX.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------

entity RODR_2TO1MUX is

   generic(nbits: natural);
port( RODR_A:  in std_logic_vector(nbits-1 downto 0);
      RODR_B:  in std_logic_vector(nbits-1 downto 0);
      RODR_SEL : in std_logic;
      RODR_OUT : Out std_logic_vector(nbits-1 downto 0));

end RODR_2TO1MUX;

-------------------------------------------------------
```

*Figure 5: VHDL code for 2-to-1 Multiplexer*

This electronic device will allow us to create an accumulator using the outputs from the first N-bit register as **RODR_A** and the output from the adder/subtractor as **RODR_B**. **RODR_SEL** will be used to determine whether the user wants the whole circuit to either work as an accumulator or simple adder/subtractor. **RODR_OUT** will be used to output to the adder/subtractor either A or B.

```

architecture LogicFunction of RODR_2TO1MUX is
begin
   process(RODR_SEL)
   begin

      case RODR_SEL is

      when '0' =>
         RODR_OUT <= RODR_A;
      when '1' =>
         RODR_OUT <= RODR_B;
      when others =>
         RODR_OUT <= (others => '1');
      end case;
   end process;

end LogicFunction;
```

*Figure 6: Part 2 of VHDL code for 2-to-1 Multiplexer*

Inside the architecture, we have a simple process taking in the selector. If **RODR_SEL** is ‘0’, the circuit will act like an adder/subtractor and use the two initial registers for computation.

If **RODR_SEL** is ‘1’, then we will use the previous accumulation of both A and B and use the second register to add or subtract from it.

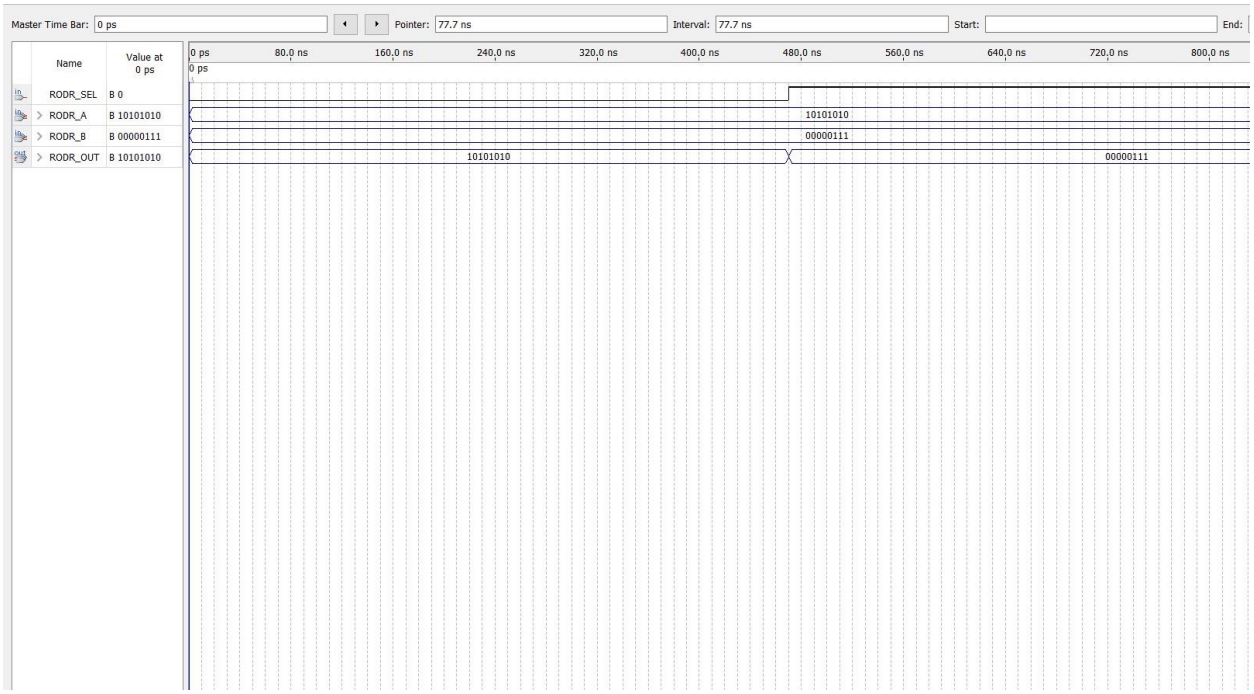


*Figure 7: Waveform for 2-to-1 Multiplexer*

From this waveform, we see that when **RODR_SEL** is ‘0’, it will simply output the combination of A, which is ‘10101010’. Once the selection turns to ‘1’, the output will automatically switch to B, which is ‘00000111’.

---

## *Unsigned/Signed Circuit (Two’s Complement)*

---

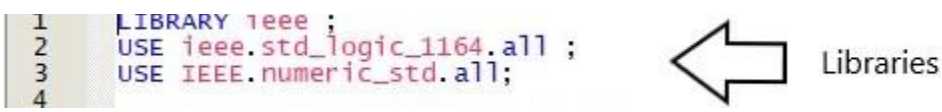


*Figure 7: Libraries used in RODR_UnsignedSigned VHDL file*

The first thing that the board is faced with is the Unsigned/Signed circuit. It takes in the inputs from the switches on either the DE2 or DE1-SOC boards and determines if two’s complement is needed in order to be displayed correctly on the seven segment displays.

To determine if we need to do such a conversion, we first need to set up our libraries so that we can take in logical inputs from the switches, as ***Figure 7*** shows. We are using *IEEE.std_logic_1164.all* to define our logical inputs. We are also using *IEEE.numeric_std.all* to provide for arithmetic in logical vectors.

```
ENTITY RODR_UnsignedSigned IS
PORT(RODR_SIGBIT : IN STD_LOGIC;                          -- Switch for Unsigned/Signed
     RODR_IN   : IN STD_LOGIC_VECTOR (7 downto 0) ;       -- Switch Inputs
     RODR_OUT  : OUT STD_LOGIC_VECTOR (7 downto 0);       -- Signed or Unsigned Output
     RODR_LEDS : OUT STD_LOGIC_VECTOR (8 downto 0));      -- LED On/Off
END RODR_UnsignedSigned ;
```

Inputs & Outputs

*Figure 8: Inputs and Outputs defined*

Since the DE2 has 18 switches and the DE1-SOC has 10, we will only be using 8 switches to accommodate for the limitation. **RODR_SIGBIT** is set as a button that the user will define as to whether they want the input to be signed or unsigned. **RODR_IN** is set as the first 8 switches on the board (from the right) and will be used to input binary numbers that will eventually be displayed on the seven segment displays. **RODR_OUT** will be the output from the Unsigned/Signed circuit which will be either signed or unsigned. **RODR_LEDS** is the output of the LED's that will be displayed on the board (recently has been removed as the LEDs are now independent of all electronic devices). There has also been the addition of a generic variable called **nbits** that allows us to do two's complement on any number of bits input into this circuit.

```
ARCHITECTURE LogicFunction OF RODR_UnsignedSigned IS

    signal IN_FLIP : STD_LOGIC_VECTOR (7 downto 0);                  --Functionality to Flip all Bits from switch
    signal ADD_ONE : STD_LOGIC_VECTOR (7 downto 0) := "00000001";    -- Add one to Complete "Two's Complement"
    signal SIGNED_OUT : STD_LOGIC_VECTOR (7 downto 0);               -- Temporary signal to store end result

    BEGIN
        IN_FLIP <= NOT RODR_IN;
```

Temp. Variables

*Figure 9: Defining temporary variables to store values needed for two's complement and outputs*

There are also some assigned variables that will also be used in the case that we need two's complement. Signal **IN_FLIP** is used for automatically inverting **RODR_IN** to start the two's complement process. Signal **ADD_ONE** is used to "add one" to the inversion of the bits taken in from the board to complete two's complement. Signal **SIGNED_OUT** is a temporary variable that will store the resulting output.

```
SIGNED_OR_UNSIGNED: process(RODR_IN, IN_FLIP, ADD_ONE, SIGNED_OUT, RODR_SIGBIT)  -- Create a process for sequential logic reading, not concurrent

variable carry : STD_LOGIC;                    -- Initialized to one to add 1 bit to Bit flipped RODR_IN
BEGIN
   for LEDNUM in 0 to 7 loop                   -- Assignment for switches based on
      RODR_LEDS(LEDNUM) <= RODR_IN(LEDNUM);
   end loop;
   RODR_LEDS(8) <= RODR_SIGBIT;                -- Assigned RODR_SIGBIT to the 8th LED (from the right)

   case RODR_SIGBIT is                         -- Checks if Unsigned/Signed Switch is On/Off

   when '0' =>                                 --CASE 1: UNSIGNED
      RODR_OUT <= RODR_IN;

   when '1' =>                              --CASE 2: SIGNED
      if((RODR_IN(7) = '0')) then           -- 0 to 127
         RODR_OUT <= RODR_IN;
      else
         if (IN_FLIP(0) = '0') then            -- Checks to see if IN_FLIP's first bit 0 or 1 and add accordingly
            SIGNED_OUT <= IN_FLIP XOR ADD_ONE;
            carry := '0';
         elsif (IN_FLIP(0) = '1') Then
            SIGNED_OUT <= IN_FLIP XOR ADD_ONE;
            carry := '1';
         end if;
         for i in 1 to 7 loop              --CHECK THROUGH REST OF IN_FLIP ARRAY checking each bit using the carry
            if (IN_FLIP(i) = '0') then
               SIGNED_OUT(i) <= IN_FLIP(i) XOR carry;
               carry := '0';
            elsif ((carry = '1') AND (IN_FLIP(i) xor carry) = '0') Then
               SIGNED_OUT(i) <= IN_FLIP(i) XOR carry;
               carry := '1';
            end if ;
```

LED Logic

*Figure 10: For loop that turns on or off LED's on board*

There is a designation of **LEDNUM** that will loop through the LED's and automatically set it to the inputs from the switches (including the signed/unsigned switch). NOTE: This part has been removed as the LEDs have now been removed outside the circuit.

```
SIGNED_OR_UNSIGNED: process(RODR_IN, IN_FLIP, ADD_ONE, SIGNED_OUT, RODR_SIGBIT)  -- Create a process for sequential logic reading, not concurrent

variable carry : STD_LOGIC;                    -- Initialized to one to add 1 bit to Bit flipped RODR_IN
BEGIN
   for LEDNUM in 0 to 7 loop                   -- Assignment for switches based on
      RODR_LEDS(LEDNUM) <= RODR_IN(LEDNUM);
   end loop;
   RODR_LEDS(8) <= RODR_SIGBIT;                -- Assigned RODR_SIGBIT to the 8th LED (from the right)

   case RODR_SIGBIT is                         -- Checks if Unsigned/Signed Switch is On/Off

   when '0' =>                                 --CASE 1: UNSIGNED
      RODR_OUT <= RODR_IN;

   when '1' =>                              --CASE 2: SIGNED
      if((RODR_IN(7) = '0')) then           -- 0 to 127
         RODR_OUT <= RODR_IN;
      else
         if (IN_FLIP(0) = '0') then            -- Checks to see if IN_FLIP's first bit 0 or 1 and add accordingly
            SIGNED_OUT <= IN_FLIP XOR ADD_ONE;
            carry := '0';
         elsif (IN_FLIP(0) = '1') then
            SIGNED_OUT <= IN_FLIP XOR ADD_ONE;
            carry := '1';
         end if;
         for i in 1 to 7 loop               --CHECK THROUGH REST OF IN_FLIP ARRAY checking each bit using the carry
            if (IN_FLIP(i) = '0') then
               SIGNED_OUT(i) <= IN_FLIP(i) XOR carry;
               carry := '0';
            elsif ((carry = '1') AND (IN_FLIP(i) xor carry) = '0') Then
               SIGNED_OUT(i) <= IN_FLIP(i) XOR carry;
               carry := '1';
            end if ;
```

Carry from each binary number in IN_FLIP

Unsigned Switch ON/OFF Logic

*Figure 11: Creating two's complement using an XOR and variable "Carry" system*

In order to create two's complement, we have created a variable called **carry**, which will determine whether or not the addition of the first bit with two's complement will be either lead to a logical '0' carry or a logical '1' carry. The use of **RODR_SIGBIT** will also be carried into a process that will provide for the use of switch statements on whether we need to check for two's complement. If logical '0' is passed in from the button (off on the board), then simply pass **RODR_IN** through to **RODR_OUT** for the Binary to BCD Decoder. If logical '1' is passed in from the switch (flipped upward on the board), then we must check if the binary input needs to use two's complement to show a negative number. If the most significant bit in **RODR_IN** is '0', then we also just simply pass through **RODR_IN** to **RODR_OUT**. If the most significant bit is '1', then we must do two's complement.

We used an initial condition for the first bit of the binary number from **IN_FLIP** and check whether if it is a logical '0' or '1'. If it is a logical '0', then we use an XOR gate to add both **IN_FLIP** and **ADD_ONE** to **SIGNED_OUT** and start without any carry. If **IN_FLIP** starts with a logical '1', then we will also add **IN_FLIP** and **ADD_ONE** using an XOR gate, but provide for a carry of '1'. The for loop right after will allow for the use of the variable **carry** to go through each bit in **IN_FLIP** to complete two's complement.

```
            end if ;
         end loop ;
      RODR_OUT <= SIGNED_OUT;          -- Put temporary array of bits into RODR_OUT
      end if;

   end case;

END process Signed_OR_Unsigned ;

END LogicFunction ;
```

*Figure 12: End of code for RODR_UnsignedSigned*

At the very end we pass in the two's complement version of the input **RODR_IN** and pass it to the output of the circuit.

This circuit has also been used within the adder/subtractor to complete two's complement on the second output in case that **RODR_CIN** is '1' to subtract the first input from the second input.

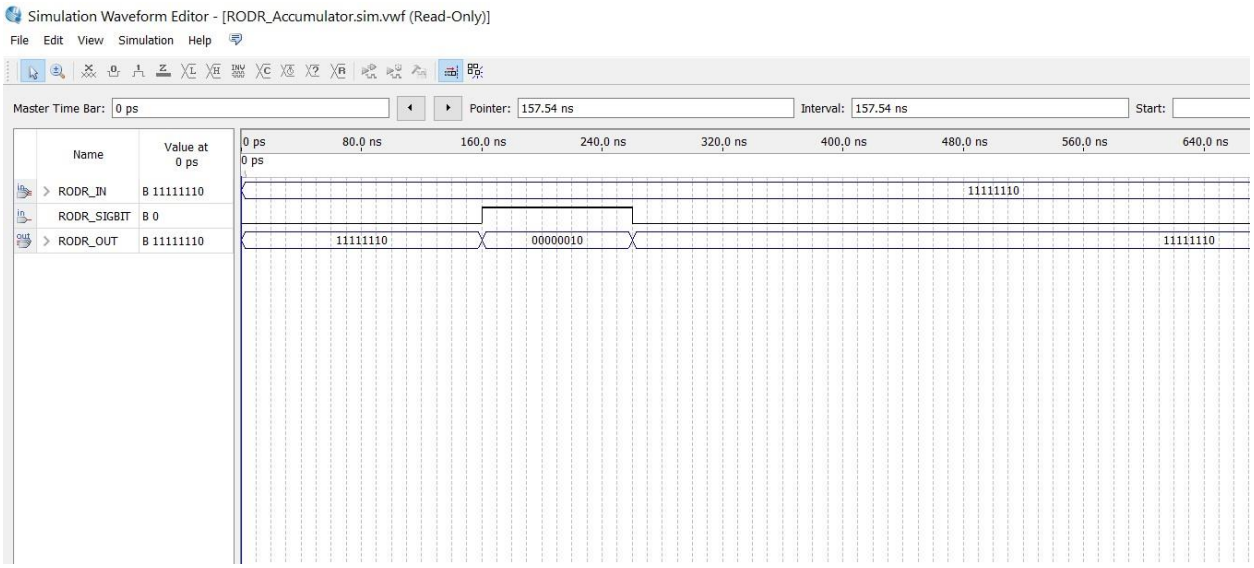


*Figure 13: Waveform for Unsigned/Signed Circuit*

For the waveform, we will be using 8 bits as our test for the unsigned/signed circuit. For the input, we place in the number 254 (or '11111110' in binary) into **RODR_IN** initially with **RODR_SIGBIT** at logical '0'. This will then simply carry the input straight to the output. However, once **RODR_SIGBIT** jumps to a logical '1', the output then changes to accommodate for two's complement, outputting -2 (or '00000010' in binary) with the help of the Plus/Minus decoder to show that onto the seven segment displays. Once **RODR_SIGBIT** goes back down to a logic '0', it then again outputs what the input was directly.

## *Binary to BCD Decoder*

RODR_BINARYBCD.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity RODR_BINARYBCD is
   generic (nbits : natural;
            BCDBITS : natural); -- nbits(8) -> BCDBITS(12), nbits(16) -> BCDBits (20), nbits(32) -> BCDBITS (40)

    Port ( RODR_IN : in  STD_LOGIC_VECTOR (nbits-1 downto 0);
           RODR_BCDVector : out std_logic_vector (BCDBITS-1 downto 0));
end RODR_BINARYBCD;
```

*Figure 13: First part of VHDL code for Binary to BCD Decoder*

In this section we will be discussing the functionality of the Binary to BCD Decoder. Due to how we will be having n-bit shift registers and an n-bit adder, we will also need an n-bit Binary to BCD decoder. In order to do so, we have declared a generic variable called **nbits** that would allow for any number of bits to be imported into this decoder. We have also called another generic variable called **BCDBITS** that will be used to create an expanded array that will be cable of using the double-dabble algorithm.

This decoder will be taking in the input from the two's complement circuit under **RODR_IN** and output a BCD vector (**RODR_BCDVector**) that will then go to a decoder for the seven segment displays.

```
architecture Behavioral of RODR_BINARYBCD is
begin
   process(RODR_IN)
   variable temp : STD_LOGIC_VECTOR (nbits-1 downto 0);
   variable bcd : UNSIGNED (BCDBITS-1 downto 0) := (others => '0');  -- Unsigned to do number inputs

   begin
      bcd := (OTHERS => '0');
      temp(nbits-1 downto 0) := RODR_IN;     -- read input into temp variable

      for i in 0 to nbits-1 loop
         for j in 0 to (BCDBITS/4)-1 loop
            if bcd((j*4)+3 DOWNTO j*4) > 4 then
               bcd((j*4)+3 DOWNTO j*4) := bcd((j*4)+3 DOWNTO j*4) + 3;
            end if;
         end loop;
      -- don't need to do anything to upper 4 bits of bcd

      -- shift bcd left by 1 bit, copy MSB of temp into LSB of bcd
         bcd := bcd(BCDBITS-2 downto 0) & temp(nbits-1);

      -- shift temp left by 1 bit
         temp := temp(nbits-2 downto 0) & '0';
         end loop;
      -- set outputs
      RODR_BCDVector <= STD_LOGIC_VECTOR(bcd);
   end process bcd1;
end Behavioral;
```

*Figure 12: Second part of VHDL code for Binary to BCD Decoder*

In the architecture, the double-dabble algorithm will take place. We declared a variable **temp** that will store the initial input into the decoder itself. Variable **bcd** will be the longer version of **temp** that will hold the converted Binary to Binary Coded Decimal to be output in the end. We then iterate through each consecutive 4 bits and check if they are greater than 4. We do this to avoid any coded decimal to be over '1001' (or 9 in decimal) and thus truncate it and add 3

to move the excess bits into the next unit. We continuously do this until every sequence abides by the rule and then output it. Within the process, we also have to shift the bits in order to correct any 4 bits that may be more than ‘1001’.

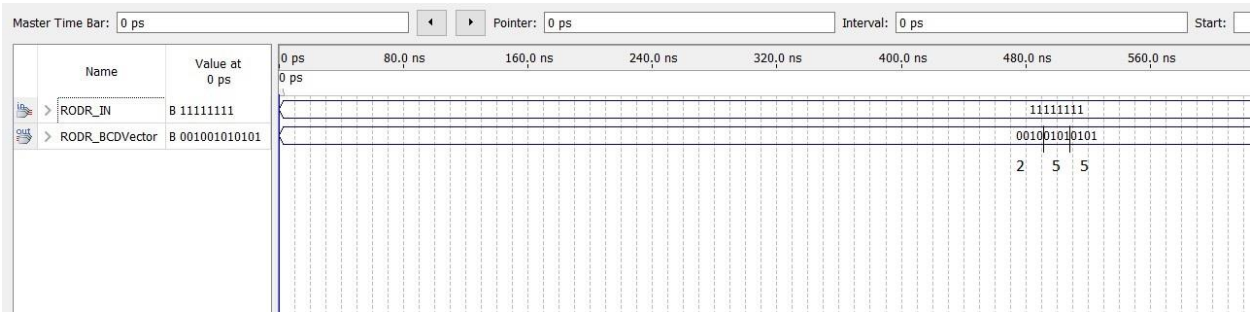


*Figure 13: Waveform for Binary to BCD Decoder*

Based on this waveform, we can see the functionality of the Binary to BCD Decoder. We input bit sequence ‘11111111’ into the decoder (or 255 in binary). The output will then parse the units in both the ones, tens, and hundreds place (from right to left) and output the number 255 with 4 bits representing each unit. This is done in **RODR_BCDVector**.

## *BCD to Seven Segment Display Decoder*

RODR_BCDDecoder.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all

   ENTITY RODR_BCDDecoder IS
   PORT(RODR_IN   : IN STD_LOGIC_VECTOR (3 downto 0) ;   -- Input from Binary to BCD Decoder (For Ones, Tens, and Hundreds)
        RODR_OUT  : OUT STD_LOGIC_VECTOR (6 downto 0);   -- Output as segments from 7 Segment Display
END RODR_BCDDecoder ;

ARCHITECTURE LogicFunction OF RODR_BCDDecoder IS
BEGIN

   LogicO_To_9: process(RODR_IN)    -- Create a process for sequential logic reading, not concurrent
   BEGIN
      case RODR_IN is               -- Switch statement checking 4-bit RODR_IN

      when "0000" =>                -- Number 0
         RODR_OUT <= "0000001";
      when "0001" =>                -- Number 1
         RODR_OUT <= "1001111";
      when "0010" =>                -- Number 2
         RODR_OUT <= "0010010";
      when "0011" =>                -- Number 3
         RODR_OUT <= "0000110";
      when "0100" =>                -- Number 4
         RODR_OUT <= "1001100";
      when "0101" =>                -- Number 5
         RODR_OUT <= "0100100";
      when "0110" =>                -- Number 6
         RODR_OUT <= "1100000";
      when "0111" =>                -- Number 7
         RODR_OUT <= "0001111";
      when "1000" =>                -- Number 8
         RODR_OUT <= "0000000";
      when "1001" =>                -- Number 9
         RODR_OUT <= "0001100";

      when others =>
         RODR_OUT <= "1111111";     -- Segment Display is off (Error Checking)

      END case;

   END process LOGICO_TO_9 ;

END LogicFunction ;
```

*Figure 15: VHDL code for BCD to Seven Segment Decoder*

The BCD to Seven Segment Decoder will be used to take the units sections place sections from the Binary to BCD decoder under separate decoders to then be displayed on the boards.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all

   ENTITY RODR_BCDDecoder IS
   PORT(RODR_IN   : IN STD_LOGIC_VECTOR (3 downto 0) ;   -- Input from Binary to BCD Decoder (For Ones, Tens, and Hundreds)
        RODR_OUT  : OUT STD_LOGIC_VECTOR (6 downto 0);   -- Output as segments from 7 Segment Display
END RODR_BCDDecoder ;
```

*Figure 16: Libraries and Inputs/Outputs defined in RODR_BCDDecoder*

In this decoder, we are only using the *ieee.std_logic_1164.all* library to define our logical variables and vectors. **RODR_IN** will be used to take in the 4 bits provided by a selection from the Binary to BCD Decoder. **RODR_OUT** will be the code that enable and disables parts of the seven segment displays to display numbers 0 to 9.

```
ARCHITECTURE LogicFunction OF RODR_BCDDecoder IS
BEGIN

   Logic0_To_9: process(RODR_IN)     -- Create a process for sequential logic reading, not concurrent
   BEGIN
      case RODR_IN is                -- Switch statement checking 4-bit RODR_IN

      when "0000" =>                 -- Number 0
         RODR_OUT <= "0000001";
      when "0001" =>                 -- Number 1
         RODR_OUT <= "1001111";
      when "0010" =>                 -- Number 2
         RODR_OUT <= "0010010";
      when "0011" =>                 -- Number 3
         RODR_OUT <= "0000110";
      when "0100" =>                 -- Number 4
         RODR_OUT <= "1001100";
      when "0101" =>                 -- Number 5
         RODR_OUT <= "0100100";
      when "0110" =>                 -- Number 6
         RODR_OUT <= "1100000";
      when "0111" =>                 -- Number 7
         RODR_OUT <= "0001111";
      when "1000" =>                 -- Number 8
         RODR_OUT <= "0000000";
      when "1001" =>                 -- Number 9
         RODR_OUT <= "0001100";

      when others =>
         RODR_OUT <= "1111111";      -- Segment Display is off (Error Checking)

      END case;

   END process LOGIC0_TO_9 ;

END LogicFunction ;
```

Lookup table for 4-bit binary input

*Figure 17: Switch case that will be used to decode number 0 to 9*

In this architecture, we will be using a switch case to determine the number sequence being taken in to each decoder. The input that will be accepted will be the binary numbers “0000” to “1001” signifying their decimal counterparts 0 to 9. Based on both the DE2 & DE1-SOC manuals, **RODR_OUT** will be determined based on the bit sequences from those manuals to display numbers 0 through 9. If there are other inputs that are taken in through **RODR_IN**, it will then display nothing. This is used to do some error checking to signify that there is something wrong with the mapping of all these decoders and circuits together.

From the Binary to BCD Decoder, we will need a certain amount of seven segment decoders depending on the number of bits we're using (3 for 8 bits, 5 for 16 bits, and 7 for 32 bits). NOTE: We would need 10 decoders to display a full 32-bit decimal number but because of both DE2 and DE1-SOC limitations, we are only allowed to use 7 or 6 maximum to also include the plus/minus sign.

Note that there will be no waveform for this as it is best to be displayed using the seven segment displays on the DE2 and DE1-SOC boards.

## *Plus/Minus Decoder*

RODR_PlusMinus.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY RODR_PlusMinus IS
   PORT(RODR_SIGBIT : IN STD_LOGIC;                        -- Input from Signed/Unsigned Switch
        RODR_IN: IN STD_LOGIC_VECTOR (7 downto 0);         -- Input from Bit Switches
        RODR_OUT : OUT STD_LOGIC_VECTOR (6 downto 0));     -- Output for 7 Segment Display
END RODR_PlusMinus ;

ARCHITECTURE LogicFunction OF RODR_PlusMinus IS
BEGIN
   plus_or_minus : process(RODR_SIGBIT, RODR_IN)
   BEGIN

      if   ((RODR_SIGBIT = '0') AND (RODR_IN(7) = '0')) then      -- Display Nothing (Unsigned)
         RODR_OUT <= "1111111";
      elsif((RODR_SIGBIT = '0') AND (RODR_IN(7) = '1')) then      -- Display Nothing (Unsigned)
         RODR_OUT <= "1111111";
      elsif((RODR_SIGBIT = '1') AND (RODR_IN(7) = '1')) then      -- Display Minus (Signed)
         RODR_OUT <= "1111110";
      elsif((RODR_SIGBIT = '1') AND (RODR_IN(7) = '0')) then      -- Display Plus (Signed)
         RODR_OUT <= "1001110";
      end if;
   end process plus_or_minus;

END LogicFunction ;
```

*Figure 18: VHDL Code showcasing the Plus/Minus decoder*

In this decoder, we are using the 6th seven segment display to display either plus, minus or nothing. This is decided based on whether the unsigned/signed switch is turned off or on, respectively.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY RODR_PlusMinus IS
   PORT(RODR_SIGBIT : IN STD_LOGIC;                        -- Input from Signed/Unsigned Switch
        RODR_IN: IN STD_LOGIC_VECTOR (7 downto 0);         -- Input from Bit Switches
        RODR_OUT : OUT STD_LOGIC_VECTOR (6 downto 0));     -- Output for 7 Segment Display
END RODR_PlusMinus ;
```

*Figure 19: Libraries and Inputs/Outputs defined in RODR_PlusMinus*

We are using the *ieee.std_logic_1164.all* library to help define our logical variables and vectors. In this decoder, we are taking in **RODR_SIGBIT**, which allows us to use the unsigned/signed switch to know if we need a positive or negative sign displayed. **RODR_IN** will take the entire input vector from the switches in order to show a plus or minus. **RODR_OUT** will be the decoded version for the display to show either a "+", "-", or turned off.

```
ARCHITECTURE LogicFunction OF RODR_PlusMinus IS
BEGIN
   plus_or_minus : process(RODR_SIGBIT, RODR_IN)
   BEGIN

      if   ((RODR_SIGBIT = '0') AND (RODR_IN(7) = '0')) then      -- Display Nothing (Unsigned)
         RODR_OUT <= "1111111";
      elsif((RODR_SIGBIT = '0') AND (RODR_IN(7) = '1')) then      -- Display Nothing (Unsigned)
         RODR_OUT <= "1111111";
      elsif((RODR_SIGBIT = '1') AND (RODR_IN(7) = '1')) then      -- Display Minus (Signed)
         RODR_OUT <= "1111110";
      elsif((RODR_SIGBIT = '1') AND (RODR_IN(7) = '0')) then      -- Display Plus (Signed)
         RODR_OUT <= "1001110";
      end if;
   end process plus_or_minus;

 END LogicFunction ;
```

*Figure 20: Logic behind Plus/Minus Decoder*

In order to use this decoder properly, we have to check **RODR_SIGBIT** and the most significant bit in **RODR_IN** to know when to display either a positive, negative, or nothing is displayed at all. These if statements will use an AND gate to determine whether we need any of those as an output.

To ensure that nothing is displayed, **RODR_SIGBIT** bit MUST be a logical '0'. This means that the bits being displayed to the output is unsigned and thus does not need an indication of a plus or minus as it is already implied that the decimal number displayed is positive. However, if **RODR_SIGBIT** is a logical '1' (switch flipped), then we need to display positive or negative. To display a negative, the most significant bit from **RODR_IN** must be a logical '1'. This means that two's complement was done in the Unsigned/Signed Circuit to account for it being a negative number between [-1 to -128]. Otherwise, display a plus as the binary number is between decimal converted 0 to 127.

No waveform will be displayed for this as it is better with a demonstration with the binary bits on the board itself.

## *Master File*

RODR_MASTERMAP.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Entity RODR_MASTERMAP IS
   generic(N : natural := 16;
            BCDB: natural := 20);

   PORT(RODR_SIGBIT      : IN STD_LOGIC;
        RODR_Clock1      : IN STD_LOGIC;
        RODR_Clock2      : IN STD_LOGIC;
        RODR_Preset      : IN STD_LOGIC;
        RODR_Clear       : IN STD_LOGIC;
        RODR_Select      : IN STD_LOGIC;      -- Enable
           RODR_binIN    : IN STD_LOGIC_VECTOR (7 downto 0) ;
           RODR_LEDONOFF : OUT STD_LOGIC_VECTOR (15 downto 0);
               RODR_OF   : out std_logic; -- Overflow
               ZerFlag   : out std_logic;
               NegFlag   : out std_logic;

      RODR_SIGN          : OUT STD_LOGIC_VECTOR (6 downto 0);     -- For Plus/Minus Seven Segment Display (Last One)
    RODR_SegmentDisplays : out STD_LOGIC_VECTOR (48 downto 0));      -- For 7 Displays

END RODR_MASTERMAP;
```

*Figure 21: Ports for Pin Assignment on Board using Master File*

In this file we will be mainly connecting all the electronics we created together here. This will allow us to create an accumulator and continuously accrue numbers until overflow has been created.

Based on this figure, we have created two generic variables called **N** and **BCDB**. These two variables allow us to allocate the number of bits for the entire system and the vector allocation needed for the double-dabble algorithm, respectively. In the ports for pin assignments, we use **SIGBIT** to determine whether we are doing signed/unsigned arithmetic. Both **Clock1** and **Clock2** are used to shift in bits into the first two registers for addition. **Preset** and **Clear** allow for all the registers to either be 1s or 0s, respectively. **Select** tells us whether we will be using the mux for accumulator or not. **binIn** is used for the inputs from the board itself. We will be using 8 switches to set this under. **LEDONOFF** will be used to display red LEDs on the board to show the registers' current binary output. **OF** is the overflow bit and will be displayed on a green LED if overflow has occurred. **ZerFlag** is used to show whether or not the arithmetic from the two N bit registers is 0. If so, it will be on under a green LED. **NegFlag** is used to show if the arithmetic outputs a negative answer. If so, it will be on under a green LED. **SIGN** is used to show the sign of the output itself. It will be displayed on the last LED. **SegmentDisplays** will be used to set up the pin assignments to all seven segment displays to be used.

```
ARCHITECTURE CONNECTIONS of RODR_MASTERMAP IS
   Component RODR_NBit_Shift_Register
      generic(nbits : natural := N);
      Port (RODR_CLK : in STD_LOGIC;
            RODR_SET : in STD_LOGIC;
            RODR_CLR : in STD_LOGIC;
            RODR_INS : in STD_LOGIC_VECTOR (7 downto 0) := (Others => '0');
            RODR_INP: in std_LOGIC_VECTOR (nbits-1 downto 0) := (Others => '0');
            Load     : in std_logic := '0';
            RODR_OUT : out STD_LOGIC_VECTOR (nbits-1 downto 0));
   END Component;

   Component RODR_Adder
      generic(nbits : natural := N);
      port(     RODR_A :   in std_logic_vector(nbits-1 downto 0);
                RODR_B :   in std_logic_vector(nbits-1 downto 0);
         RODR_overflow :   out std_logic;
         RODR_CIN      :   in std_logic;
         RODR_sum      :   out std_logic_vector(nbits-1 downto 0));
   END Component;

   Component RODR_UnsignedSigned
      generic (nbits : natural := N);
      PORT(RODR_SIGBIT : IN STD_LOGIC;
            RODR_IN   : IN STD_LOGIC_VECTOR (nbits-1 downto 0);
            RODR_OUT  : OUT STD_LOGIC_VECTOR (nbits-1 downto 0));
   END Component;

   Component RODR_BINARYBCD
      generic(  nbits : natural := N;
             BCDBITS : natural := BCDB);
      PORT (        RODR_IN : in  STD_LOGIC_VECTOR (nbits-1 downto 0);
            RODR_BCDVector : out std_logic_vector (BCDBITS-1 downto 0));
   END Component ;

   Component RODR_BCDDecoder
      PORT(RODR_IN : IN STD_LOGIC_VECTOR (3 downto 0) ;
          RODR_OUT : OUT STD_LOGIC_VECTOR (6 downto 0));
   END Component ;
```

```
   Component RODR_PlusMinus
      PORT(RODR_SIGBIT : IN STD_LOGIC;
               RODR_IN : IN STD_LOGIC;
              RODR_OUT : OUT STD_LOGIC_VECTOR (6 downto 0));
   END Component;

   Component RODR_2TO1MUX
      generic(nbits: natural := N);
      port(RODR_A:   in std_logic_vector(nbits-1 downto 0);
      RODR_B:  in std_logic_vector(nbits-1 downto 0);
      RODR_SEL : in std_logic;
      RODR_OUT : Out std_logic_vector(nbits-1 downto 0));
   END Component;
```

*Figure 22: Components used to connect accumulator*

In this figure, we see all the components created to make the accumulator. All these components have been explained in the previous subsections in detail. The reason why we needed to have components into this file is because of the need to connect them together with all of its functionality under one file.

```
signal RODR_NBitShiftOutput1 : std_LOGIC_VECTOR (N-1 downto 0);
signal RODR_MUXOUT           : std_LOGIC_VECTOR (N-1 downto 0);
signal RODR_NBitShiftOutput2 : std_LOGIC_VECTOR (N-1 downto 0);

signal RODR_ADDOUT : Std_LOGIC_VECTOR (N-1 downto 0);

signal RODR_TCO : STD_LOGIC_VECTOR (N-1 downto 0);    --TCO = TWo's Complement Output

signal RODR_BCD_Vector : STD_LOGIC_VECTOR(BCDB-1 downto 0);       -- Output from Binary to BCD Decoder

shared variable RODR_SevenSegment: STD_LOGIC_VECTOR (48 downto 0) := (Others => '1');           -- Seven Segment outputs

signal RODR_Clock3 : std_logic := RODR_Select;
signal RODR_AdderTOMUX : std_LOGIC_VECTOR (N-1 downto 0);

signal RODR_out   : std_LOGIC_VECTOR (N-1 downto 0);

```

*Figure 23: Temporary outputs to store data from PORT MAPS*

In order to properly store data and ensure safe data flow between components, we will need temporary signals and variables. It is also needed because all of the connections between components will not be user interrupted, so we will need the components to communicate with each other. **NBitShiftOutput1, NBitShiftOutput2**, and **MUXOUT** are signals that will be outputs from their respective registers, and multiplexer. **ADDOUT** will be used for the output of the adder/subtractor. **TCO** is the two's complement output from the unsigned/signed circuit. **BCD_Vector** is used to take the output from the Binary to BCD Decoder and safely placing it into **SegmentDisplays**. **Clock3** is a variable that will control when the third register with parallel input will receive data for accumulation. It is tied to **Select** as it will only be used when the mux wants data from the third register. **AdderTOMUX** is used to take in the selected output from the mux into the adder. **RODR_Out** will be not be used as it was used for an earlier version for TCO conversion.

```
BEGIN

   RODR_NBitShift1   : RODR_NBit_Shift_Register    PORT MAP(not RODR_Clock1, not RODR_Preset, not RODR_Clear, RODR_binIN, open ,'1',
                                                            RODR_NBitShiftOutput1);

   RODR_MUX          : RODR_2TO1MUX                PORT MAP (RODR_NBitShiftOutput1, RODR_AdderTOMUX, RODR_Select, RODR_MUXOUT);

   RODR_NBitShift2   : RODR_NBit_Shift_Register    PORT MAP(not RODR_Clock2, not RODR_Preset, not RODR_Clear, RODR_binIN,open, '1',
                                                            RODR_NBitShiftOutput2);

   LEDS: process(RODR_NBitShiftOutput1, RODR_NBitShiftOutput2)
   begin
      if(RODR_NBitShiftOutput2 = (RODR_NBitShiftOutput2'range => '0') ) then
         if(N-1 = 31) then
            RODR_LEDONOFF <= RODR_NBitShiftOutput1(15 downto 0);
         else
            RODR_LEDONOFF <= RODR_NBitShiftOutput1(N-1 downto 0);
         end if;

      elsif(RODR_NBitShiftOutput1 = (RODR_NBitShiftOutput1'range => '0')) then
         if(N-1 = 31) then
            RODR_LEDONOFF <= RODR_NBitShiftOutput2(15 downto 0);
         else
            RODR_LEDONOFF <= RODR_NBitShiftOutput2(N-1 downto 0);
         end if;
      else
         if(N-1 = 31) then
            RODR_LEDONOFF <= RODR_NBitShiftOutput2(15 downto 0);
         else
            RODR_LEDONOFF <= RODR_NBitShiftOutput2(N-1 downto 0);
         end if;
      end if;
   end process LEDS;
```

*Figure 24: Connections to first two shift registers, 2-to-1 Multiplexer, and LED functionality*

We then begin by setting up the clock signals, preset, clears, inputs, load variable, and the outputs. The reason why we "not" the clocks, presets, and clears is because both the DE2 and

DE1-SOC boards already have the clocks automatically at a logic high, so we turn it low for proper use. We then have a process that controls the LED functionality. It states that whenever we are using 31 bits, we simply truncate the signal down to using 16 LEDs as 31 bits would require 31 LEDs but we don't have that many on any of those boards. We also check if any of the registers are empty and if so, we just output the register that is currently in use. However if they're both in use, simply use the last register to output onto the LEDs.

```
    RODR_Add          : RODR_Adder                  PORT MAP (RODR_MUXOUT, RODR_NBitShiftOutput2, RODR_OF, RODR_SIGBIT, RODR_ADDOUT);

    RODR_TWOSCOMPLEMENT   : RODR_UnsignedSigned  PORT MAP(RODR_SIGBIT, RODR_ADDOUT, RODR_TCO);

    FLAGS: process(RODR_ADDOUT)
    begin
       if(RODR_ADDOUT = 0) then
          ZerFlag <= '1';
       elsif(RODR_ADDOUT > 0) then
          ZerFlag <= '0';
       end if;
       if((RODR_SIGBIT AND RODR_ADDOUT(N-1)) = '1') then
          NegFlag <= '1';
       else
          NegFlag <= '0';
       end if;
    end process FLAGS;

    RODR_BINARYTOBCD      : RODR_BINARYBCD       PORT MAP(RODR_TCO, RODR_BCD_Vector);

    decoders: for i in 0 to 4 generate        -- Only Work on 7 Seven Segment Displays (2,4,6 for 8, 16, 32 bit)
       DECODERX : RODR_BCDDecoder    PORT MAP(RODR_BCD_Vector((i*4)+3 downto i*4), RODR_SevenSegment((i*7)+6  downto i*7));
    END generate decoders;

    RODR_SegmentDisplays <= RODR_SevenSegment;

    RODR_POSNEG           : RODR_PlusMinus       PORT MAP (RODR_SIGBIT, RODR_ADDOUT(N-1), RODR_SIGN);

    RODR_OUTTOMUX         : RODR_NBit_Shift_Register PORT MAP(RODR_Clock3, not RODR_Preset, not RODR_Clear, open, RODR_ADDOUT, '0',
                                                              RODR_AdderTOMUX);
 END CONNECTIONS;
```

*Figure 25: All other connections between electronics, Flags, and BCDDecoder auto generation*

The adder will then add the first input from the mux and the second register output. It also outputs any overflow that comes from the computation, takes input from the **SIGBIT** as a carry in signal to determine if it will be used as an adder or subtractor. It will also output using the temporary signal **ADDOUT**. Two's complement will then take place, taking **SIGBIT** and **ADDOUT** for conversion to the seven segment displays and outputting to signal **TCO**.

For both negative flags and zero flag, we placed them in a process. This process allows for the user to see if the result of the adder led to either a zero answer or negative computation.

After that, we create a Binary to BCD Decoder and convert it to binary coded decimal using **TCO** and outputting it as a **BCD_Vector**. The vector then gets placed to the Seven segment displays. The decoders auto-generation occurs depending on the amount of bits being put. It strictly allocates the maximum number of units needed to create the decimal number. For 8 bits we only need 3 units, 16 we need 5 units, and 32 we need 10. However, because we have a limitation of seven segment displays, we had to truncate the amount of bits being shown for 32 bits to only 6/7 displays (depending on which board is in use). **PlusMinus** is then created to display whether the number is positive or negative using the user input of **SIGBIT**, the most significant bit of **ADDOUT**, and output to a **SIGN** decoder.

The last thing will be the third register that is using parallel in, parallel out. It will take in the input from the adder in parallel and store it in the register. It will then be used ONLY if the clock is a logical ‘1’ (which is when the mux is being used). Otherwise, the register will be bypassed and the computation between the two registers will be used.

## Waveforms

<u>Master File</u>

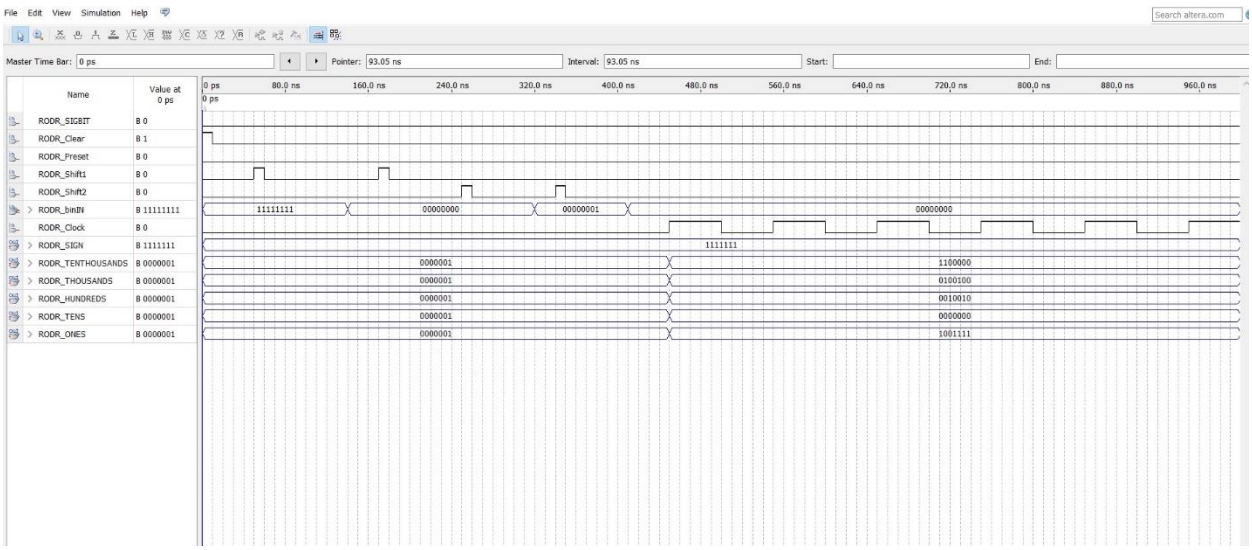


*Figure 25: Output waveforms for all inputs and outputs that have been given pin assignments to both DE2 and DE1-SOC*

From this file, we see how the entire code should work. There is an initialization of **RODR_Clear** to ensure that the registers are in ready mode. We then proceed to use the first n-bit register to shift in the first 8 MSB into it using **RODR_Shift1**. We then push it again after flipping the switches from the board (**RODR_binIN**) to all binary 0’s. After that, the second register is the next one to be set with all 0s as the MSB using **RODR_binIN** and **RODR_Shift2** to shift it to the left. We then press the shift button once more to input ‘00000001’ from the board into the second register. Once all the inputs are stored, we then start **RODR_Clock** and the output displayed on the seven segment displays are shown. The decimal converted binary for that number is 65281. **RODR_Sign** will remain off with all 1s as **RODR_SigBit** is off (implied unsigned). If it were signed, the output would be -255.

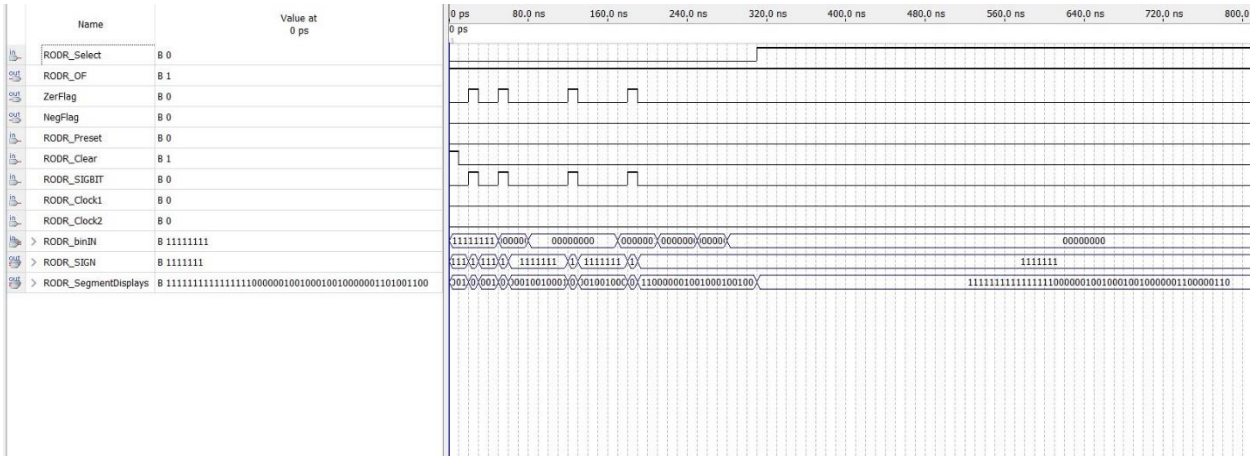


*Figure 26: Updated Waveform of entire accumulator operation*

## Board Operation

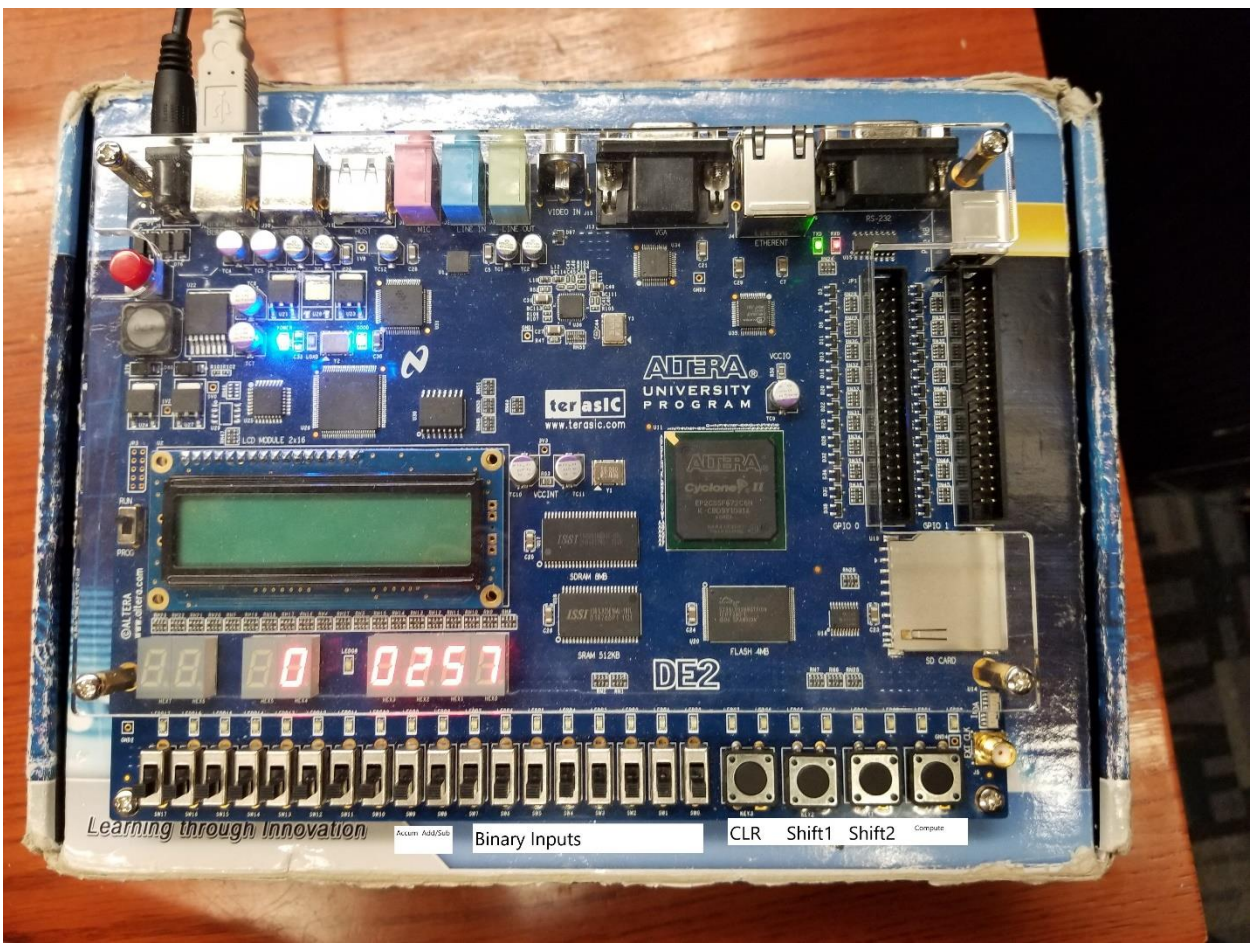


*Figure 27: Using DE2 board for accumulation*

In this picture we will showcase how the operations are created on the board shown. Initially, we would press the **CLR** button in order to remove any garbage data that may be lying around (can be seen on the displays). Once clear, we can then compute two numbers. Now, we have two shift buttons, one for each register that will import a binary number into an adder for computation. **Shift1** and **Shift2** will be pressed to enter an input from the switches one at a time. Depending on the number of bits requested, it will determine the amount of presses needed to import the correct binary number desired. For example, if 8 bits were required, we would only add 8 bits to each register. However, if 16 bits were required, we would need two shifts per register (one to insert the first sequence of bits and the other for the other sequence to make 16 bits as we are only using 8 bits from the board). Once that has been completed, we press the **Compute** button and it should display your result on the seven segment displays.

If you were to turn on the **Add/Sub** switch, you would turn the adder into a subtractor. In order to do so, we then switch it on and upon pressing the **Compute** button, you then get subtraction between the first two registers.

If you wanted to accumulate numbers, we would then have to switch on the **Accum** switch. Now, whenever you insert a number into the second register, it will accumulate in another register and display it onto the screen with every **Compute** button press. You can also subtract from that register by using the **Add/Sub** button and subtract from the accumulation itself.

Also, the three green LED's on top of the buttons (from the right) are used to signify if a negative number has been shown (negative flag), the result of a computation is zero (zero flag), or overflow from a computation. Every time the **Compute** button is pressed, we check if the result satisfied any of those conditions. If not, no green LED will be displayed.

In terms of the red LEDs, they will always show the inputs inserted into the two registers used for computation. It will only show a maximum of 16 bits. So despite us using 32 bit registers, we only display the lower 16 bits on the board.

## *LPM Modules*

When using the LPM modules, there was noticeably different way that Intel has produced their results. They also use a multiplexer and register taking the output from the Adder in order to compute their result. However, I haven't used any D Flip flops to store the result that would change the state of the mux nor the Carry In to the adder. All those changes would occur within the adder itself. In my design, the adder would store the inputs from both registers and check if the carry In is on or off. If so, the second register would then be used as a subtractor and subtract from the first register. Then at the next clock cycle, it would output that result. I also accumulate by using the clock signal and checking if the **Accum** switch on the board was turned on or off to feed the result back into the mux. With Intel's design, it uses only one clock signal for the three

registers and manage their inputs and outputs via the D flip flops for the accumulation and adding/subtracting operations.

When setting up the LPM module, we could only use up to 16 bits on the DE2 and 8 bits on the DE1, I tested it with only by using the maximum number of switches per board minus two since we need switches for the **add/sub** switch and the **select** switch. The clocks will be set to the first button (can use *figure 27* as reference), and the two switches in the same place on the DE1-SOC board and the two leftmost switches on the DE2 board. Based on my observation, I noticed that the output would automatically generate whenever I press the clock. In my design, I had to create them separately and use another clock to output onto the seven segment display. This saves button space and allows for more efficient computation. Also, I noticed that at times I would have to press the clock twice in order to get an output. This is most likely due to how the D flip flops store its input and then once the clock cycle is received, it may need to be pressed again to do its computation once the information from each register is gathered.

## Conclusion

In this laboratory exercise, I have learned how to create an accumulator using a 2-to-1 Multiplexer and a reconfiguration of my adder to also include subtraction functionality. The subtraction was made possible by implementing two's complement on the second input and then adding them together to get the proper result. At times, it would create an overflow of a bit but it is not included in the overall scheme as we automatically truncate it down for the proper binary conversion. The 2-to-1 Mux taught me the reason why we needed another register after the adder. We couldn't just simply take the output of the adder/subtractor to the mux as it would create an infinite loop. So we add a register right after to ensure that the data is saved and safely transfer the output to both the mux and output once a rising edge clock has been reached.