Multiplication, Multiplication Accumulator, and Division

Name: Jonathan J. Rodriguez

CSC343 – Computer Organization Lab, Spring 2020

Instructor: Izidor Gertner

Table of Contents

Objective

The objective of this laboratory exercise is to continue using the instruction register to apply changes to specific registers. We will be creating multiplication and division opcodes as an extra operation for register-to-register operation. We will also be creating a multiplication accumulator. The multiplication will be done using a comparison between bits of the second operand and "multiply" if there is a logical '1'. The division will be done using a comparison between the two operands and ensuring that division can be done.

Board Specifications & Software Used

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

Design Specifications & Functionality

Multiplication Unsigned

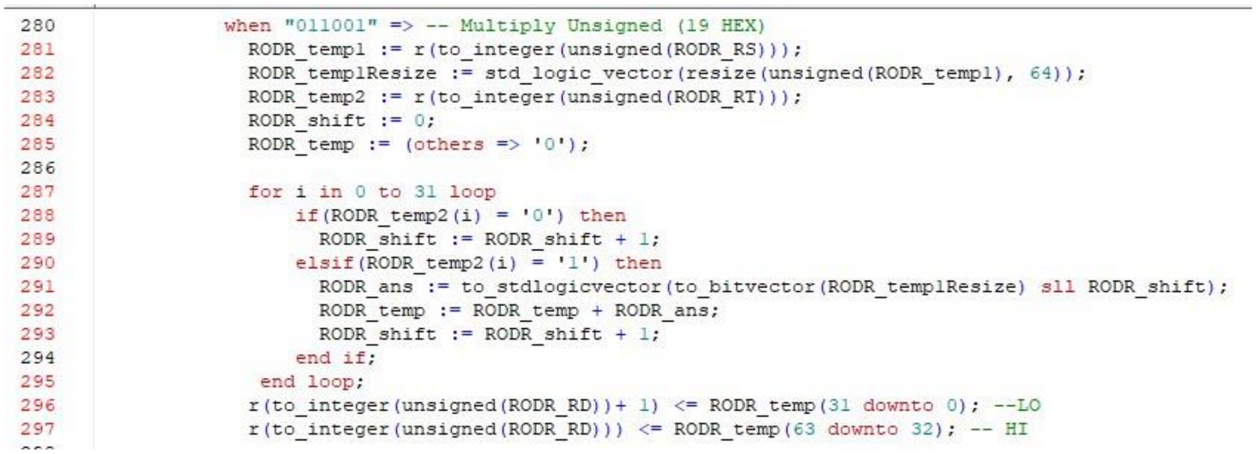
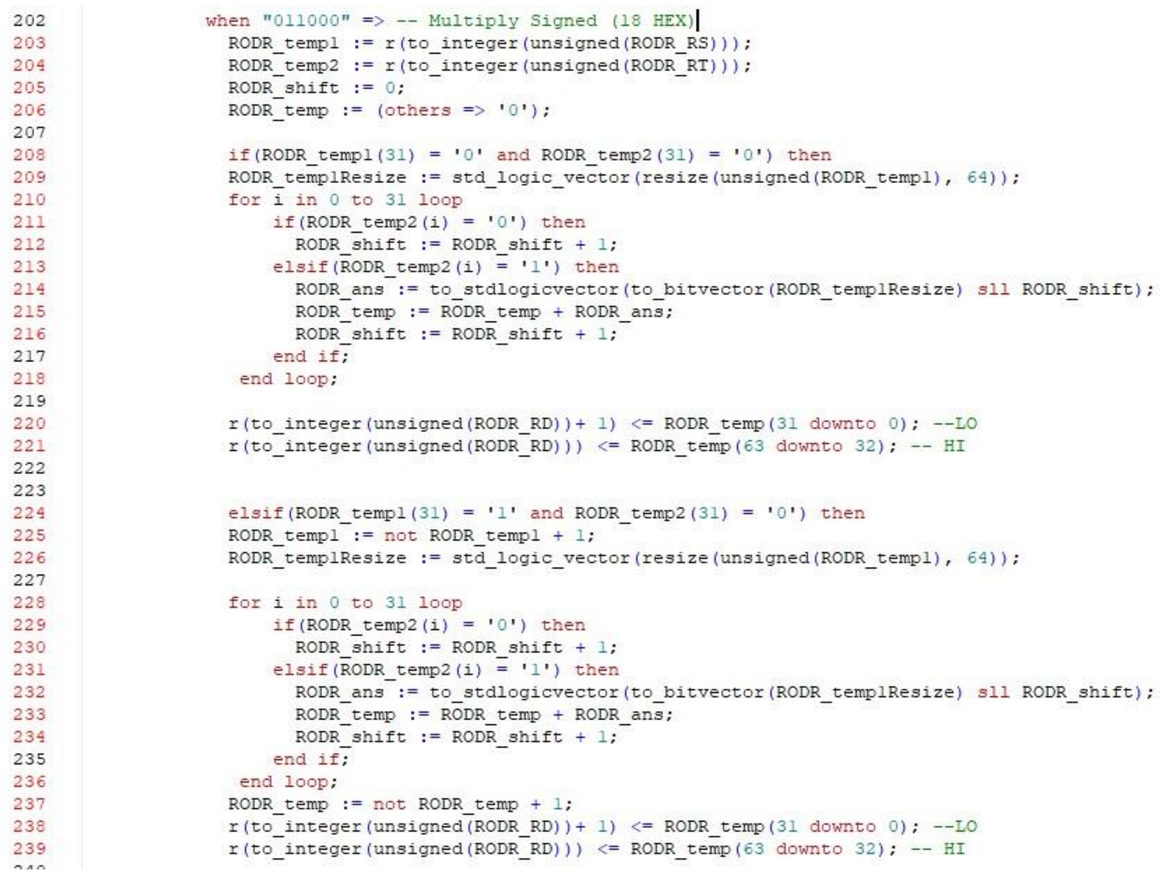

```
280            when "011001" => -- Multiply Unsigned (19 HEX)
281              RODR_temp1 := r(to_integer(unsigned(RODR_RS)));
282              RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));
283              RODR_temp2 := r(to_integer(unsigned(RODR_RT)));
284              RODR_shift := 0;
285              RODR_temp := (others => '0');
286
287              for i in 0 to 31 loop
288                  if(RODR_temp2(i) = '0') then
289                    RODR_shift := RODR_shift + 1;
290                  elsif(RODR_temp2(i) = '1') then
291                    RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
292                    RODR_temp := RODR_temp + RODR_ans;
293                    RODR_shift := RODR_shift + 1;
294                  end if;
295               end loop;
296              r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_temp(31 downto 0); --LO
297              r(to_integer(unsigned(RODR_RD))) <= RODR_temp(63 downto 32); -- HI
```

Figure 1: Code for Unsigned Multiplication

For this part, we have created an operation that will allow for unsigned multiplication. Using all components created from the previous lab regarding general operations with the use of the instruction register and register array, we can call this function. This portion is called using the 6-bit opcode for **FUNCT** using the opcode specified in line 280 of this figure. We then use **RODR_temp1** and **RODR_temp2** to store the contents of each register for us to compute with. **RODR_temp1Resize** is used to expand temp1 to 64 bits as our operation will be shifting it whenever any temp2 bit has a logical '1'. **RODR_shift** will be used as our way of shifting the bits that are being accumulated every time we do a calculation. **RODR_temp** will be used to store our output and parse it into two register for low and high bits.

We call a for loop from 0 to 31 as there are 32 bits from the second register's data. We use this as our way of saying whether based on this bit if we will be adding 0 or the entire bit sequence of the first operand into our result. If any bit from the second operand is '0', then we simply add to the shift counter. If any of them are '1', we allocate space into our result provided the current offset that is currently on. As we add from each operation, we shift to ensure that we add every result in their right base. The result will be placed into two registers based on the RD portion of the instruction register and its adjacent register (hi and low, respectively).

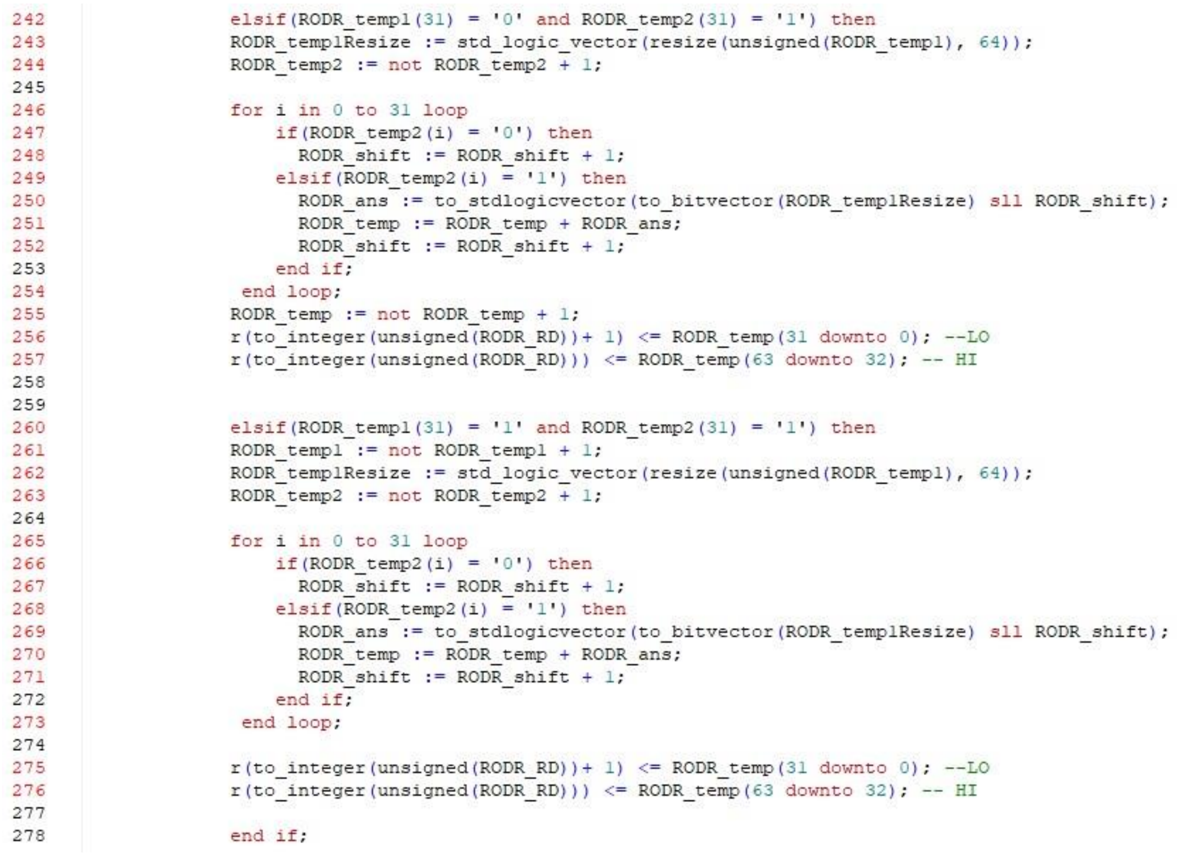
Multiplication Signed



```
when "011000" => -- Multiply Signed (18 HEX)
  RODR_temp1 := r(to_integer(unsigned(RODR_RS)));
  RODR_temp2 := r(to_integer(unsigned(RODR_RT)));
  RODR_shift := 0;
  RODR_temp := (others => '0');

  if(RODR_temp1(31) = '0' and RODR_temp2(31) = '0') then
  RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));
  for i in 0 to 31 loop
      if(RODR_temp2(i) = '0') then
        RODR_shift := RODR_shift + 1;
      elsif(RODR_temp2(i) = '1') then
        RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
        RODR_temp := RODR_temp + RODR_ans;
        RODR_shift := RODR_shift + 1;
      end if;
   end loop;

  r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_temp(31 downto 0); --LO
  r(to_integer(unsigned(RODR_RD))) <= RODR_temp(63 downto 32); -- HI


  elsif(RODR_temp1(31) = '1' and RODR_temp2(31) = '0') then
  RODR_temp1 := not RODR_temp1 + 1;
  RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));

  for i in 0 to 31 loop
      if(RODR_temp2(i) = '0') then
        RODR_shift := RODR_shift + 1;
      elsif(RODR_temp2(i) = '1') then
        RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
        RODR_temp := RODR_temp + RODR_ans;
        RODR_shift := RODR_shift + 1;
      end if;
   end loop;
  RODR_temp := not RODR_temp + 1;
  r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_temp(31 downto 0); --LO
  r(to_integer(unsigned(RODR_RD))) <= RODR_temp(63 downto 32); -- HI
```

Figure 2: First part of multiplication signed

The multiplication signed portion is very similar to the unsigned portion as it inherits the shifting of the first operand. However the end result will always be different depending on the first bit of the operands. If **RODR_temp1** or **RODR_temp2** have their MSB equal to '1', then we must perform two’s complement. This is to ensure that our current algorithm continues to work as it works only for magnitude operations. However, we also perform two’s complement on **RODR_temp** before storing them in registers as we need to ensure that the signed notation is preserved.

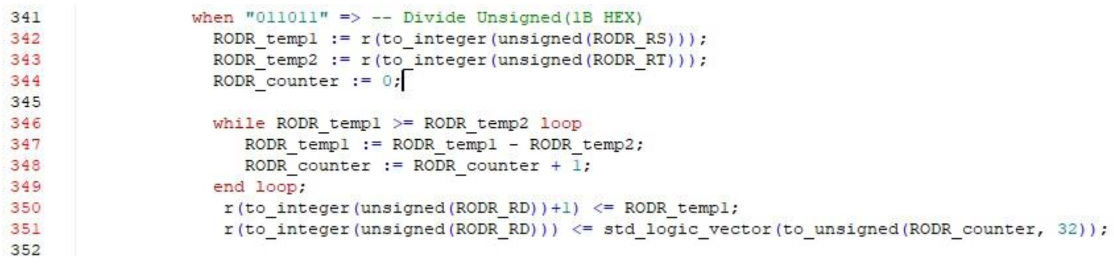


```
            elsif(RODR_temp1(31) = '0' and RODR_temp2(31) = '1') then
            RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));
            RODR_temp2 := not RODR_temp2 + 1;

            for i in 0 to 31 loop
                if(RODR_temp2(i) = '0') then
                  RODR_shift := RODR_shift + 1;
                elsif(RODR_temp2(i) = '1') then
                  RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
                  RODR_temp := RODR_temp + RODR_ans;
                  RODR_shift := RODR_shift + 1;
                end if;
             end loop;
            RODR_temp := not RODR_temp + 1;
            r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_temp(31 downto 0); --LO
            r(to_integer(unsigned(RODR_RD))) <= RODR_temp(63 downto 32); -- HI


            elsif(RODR_temp1(31) = '1' and RODR_temp2(31) = '1') then
            RODR_temp1 := not RODR_temp1 + 1;
            RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));
            RODR_temp2 := not RODR_temp2 + 1;

            for i in 0 to 31 loop
                if(RODR_temp2(i) = '0') then
                  RODR_shift := RODR_shift + 1;
                elsif(RODR_temp2(i) = '1') then
                  RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
                  RODR_temp := RODR_temp + RODR_ans;
                  RODR_shift := RODR_shift + 1;
                end if;
             end loop;

            r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_temp(31 downto 0); --LO
            r(to_integer(unsigned(RODR_RD))) <= RODR_temp(63 downto 32); -- HI

            end if;
```

Figure 3: Second part of VHDL code of Multiplication Signed

For example, if two negative numbers are multiplied, we forget about the signs and simply multiply the two operands. Only at the very end when completing the computation is when you think about what happens when multiplying two negative numbers. In this scenario, we don’t do two’s complement on the output as we want a positive answer at the end. However, if the two numbers are opposite in sign, the answer will be negative and this two’s complements needs to be done.

Division Unsigned



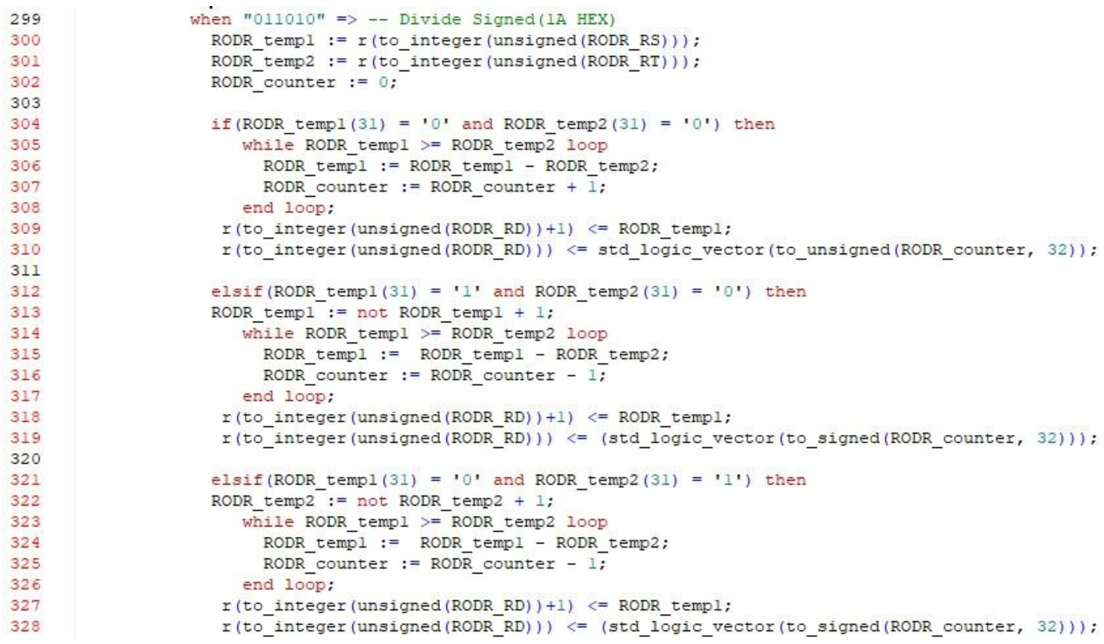
```
when "011011" => -- Divide Unsigned(1B HEX)
  RODR_temp1 := r(to_integer(unsigned(RODR_RS)));
  RODR_temp2 := r(to_integer(unsigned(RODR_RT)));
  RODR_counter := 0;

  while RODR_temp1 >= RODR_temp2 loop
     RODR_temp1 := RODR_temp1 - RODR_temp2;
     RODR_counter := RODR_counter + 1;
  end loop;
   r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
   r(to_integer(unsigned(RODR_RD))) <= std_logic_vector(to_unsigned(RODR_counter, 32));
```

Figure 4: VHDL code for Unsigned Division

In this section, we talk about unsigned division. There are many ways to do binary division, however we chose the simplest way to do so, which is a comparison between the two operands. Again, **RODR_temp1** and **RODR_temp2** will be used to store register RS and RT information, respectively. We are now also using a variable called **RODR_counter** that will be our quotient when running the comparison.

The comparison occurs if the first operand is bigger than the second. Subtraction and division are very similar as it confirms that there is a multiple of that number within the first operand, and that is counted via the quotient (**RODR_counter** variable). We also decrement every time temp1 is bigger than temp2 as it allows us to use temp1 to then be our remainder if the second operand can't be bigger than 0 (or be a multiple of that operand). The remainder than gets stored in the adjacent register of RD while the quotient gets stored on RD itself by converting from an integer to an std_logic_vector.

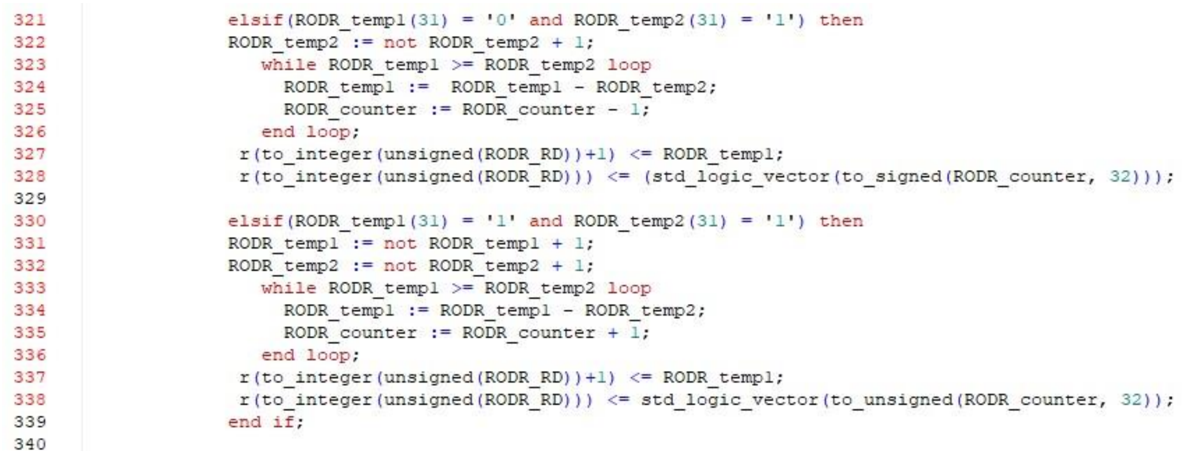
Division Signed



```
when "011010" => -- Divide Signed(1A HEX)
  RODR_temp1 := r(to_integer(unsigned(RODR_RS)));
  RODR_temp2 := r(to_integer(unsigned(RODR_RT)));
  RODR_counter := 0;

  if(RODR_temp1(31) = '0' and RODR_temp2(31) = '0') then
     while RODR_temp1 >= RODR_temp2 loop
       RODR_temp1 := RODR_temp1 - RODR_temp2;
       RODR_counter := RODR_counter + 1;
     end loop;
   r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
   r(to_integer(unsigned(RODR_RD))) <= std_logic_vector(to_unsigned(RODR_counter, 32));

  elsif(RODR_temp1(31) = '1' and RODR_temp2(31) = '0') then
  RODR_temp1 := not RODR_temp1 + 1;
     while RODR_temp1 >= RODR_temp2 loop
       RODR_temp1 :=  RODR_temp1 - RODR_temp2;
       RODR_counter := RODR_counter - 1;
     end loop;
   r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
   r(to_integer(unsigned(RODR_RD))) <= (std_logic_vector(to_signed(RODR_counter, 32)));

  elsif(RODR_temp1(31) = '0' and RODR_temp2(31) = '1') then
  RODR_temp2 := not RODR_temp2 + 1;
     while RODR_temp1 >= RODR_temp2 loop
       RODR_temp1 :=  RODR_temp1 - RODR_temp2;
       RODR_counter := RODR_counter - 1;
     end loop;
   r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
   r(to_integer(unsigned(RODR_RD))) <= (std_logic_vector(to_signed(RODR_counter, 32)));
```

Figure 5: First part of VHDL code for Division Signed

For this section, we have signed division. It is very similar to unsigned division except that we now account for when we want signed notation. The general operation of comparing both **RODR_temp1** and **RODR_temp2** is the same. However, only the inputs themselves and the resulting output gets changed. Nothing changes when the operands' most significant bits are both '0'. In all other cases, we either do two's complement on one or both operands and/or on the result as well to ensure that we preserve the sign.

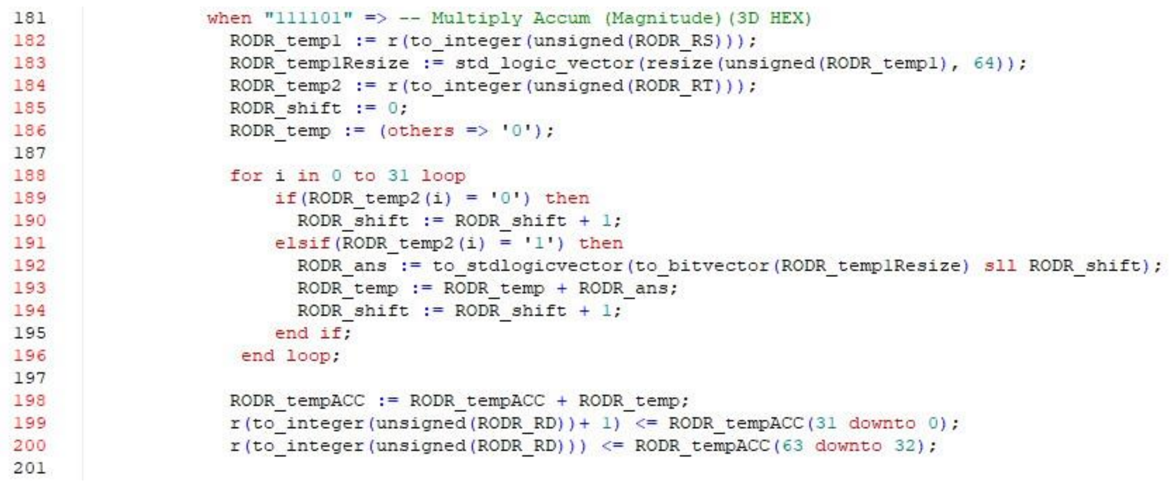


```
elsif(RODR_temp1(31) = '0' and RODR_temp2(31) = '1') then
RODR_temp2 := not RODR_temp2 + 1;
   while RODR_temp1 >= RODR_temp2 loop
     RODR_temp1 :=  RODR_temp1 - RODR_temp2;
     RODR_counter := RODR_counter - 1;
   end loop;
 r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
 r(to_integer(unsigned(RODR_RD))) <= (std_logic_vector(to_signed(RODR_counter, 32)));

elsif(RODR_temp1(31) = '1' and RODR_temp2(31) = '1') then
RODR_temp1 := not RODR_temp1 + 1;
RODR_temp2 := not RODR_temp2 + 1;
   while RODR_temp1 >= RODR_temp2 loop
     RODR_temp1 := RODR_temp1 - RODR_temp2;
     RODR_counter := RODR_counter + 1;
   end loop;
 r(to_integer(unsigned(RODR_RD))+1) <= RODR_temp1;
 r(to_integer(unsigned(RODR_RD))) <= std_logic_vector(to_unsigned(RODR_counter, 32));
end if;
```

Figure 6: Second Part of VHDL Code for Signed Division

If any of the most significant bits are 1 (either or), we MUST change the counter to be signed as we are decrementing to account for the negative result. The remainder stays unchanged as it only is used to refer to the portion that cannot be calculated due to the operands not being multiples of each other.

Multiplication Accumulator

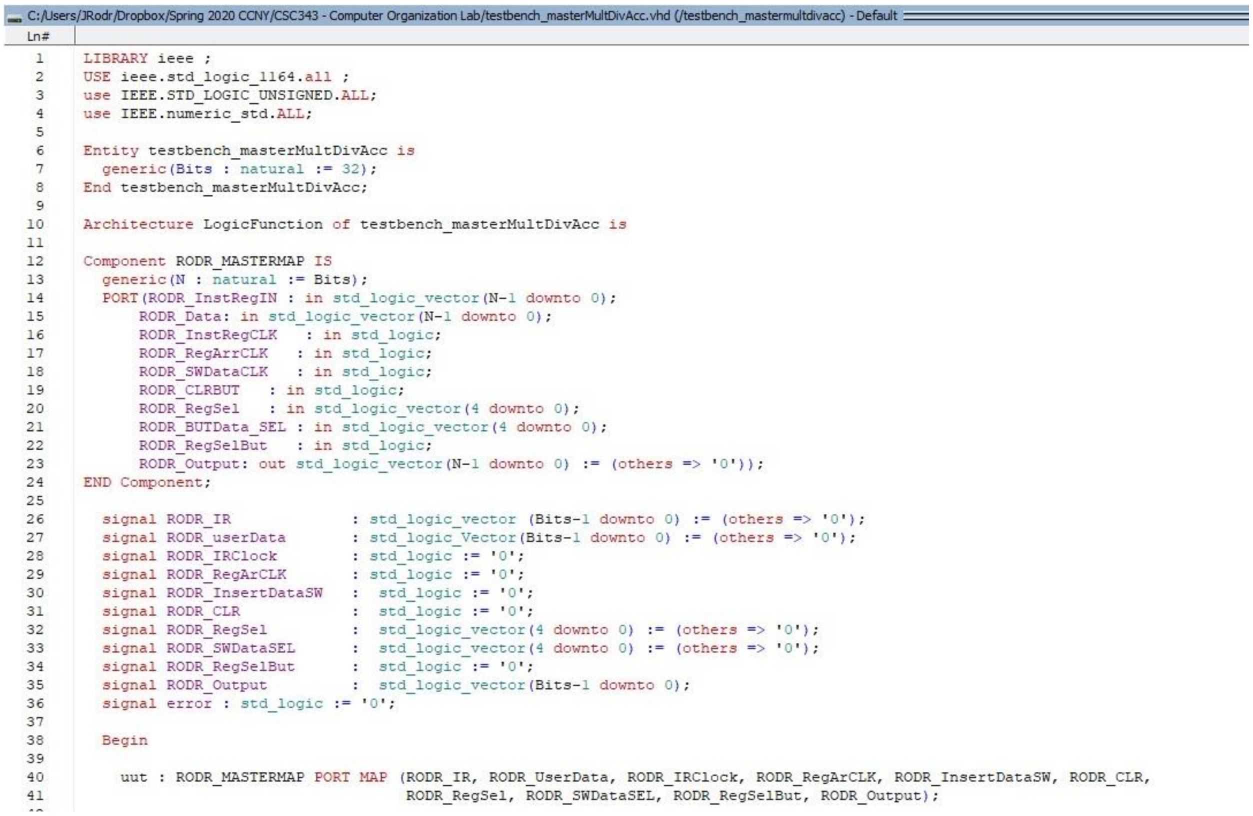


```
when "111101" => -- Multiply Accum (Magnitude)(3D HEX)
  RODR_temp1 := r(to_integer(unsigned(RODR_RS)));
  RODR_temp1Resize := std_logic_vector(resize(unsigned(RODR_temp1), 64));
  RODR_temp2 := r(to_integer(unsigned(RODR_RT)));
  RODR_shift := 0;
  RODR_temp := (others => '0');

  for i in 0 to 31 loop
      if(RODR_temp2(i) = '0') then
        RODR_shift := RODR_shift + 1;
      elsif(RODR_temp2(i) = '1') then
        RODR_ans := to_stdlogicvector(to_bitvector(RODR_temp1Resize) sll RODR_shift);
        RODR_temp := RODR_temp + RODR_ans;
        RODR_shift := RODR_shift + 1;
      end if;
   end loop;

  RODR_tempACC := RODR_tempACC + RODR_temp;
  r(to_integer(unsigned(RODR_RD))+ 1) <= RODR_tempACC(31 downto 0);
  r(to_integer(unsigned(RODR_RD))) <= RODR_tempACC(63 downto 32);
```

Figure 7: VHDL Code for Multiplication Accumulator

This code is for our multiplication accumulator. In inherits everything from the multiplication unsigned and signed sections except that we now use an extra variable to store previous outputs. **RODR_tempACC** is used to do so. All shift operations are the same on the first operand, except that we will keep track of the previous multiplication operations and continue doing so until the opcode is no longer called.

Testbench Code



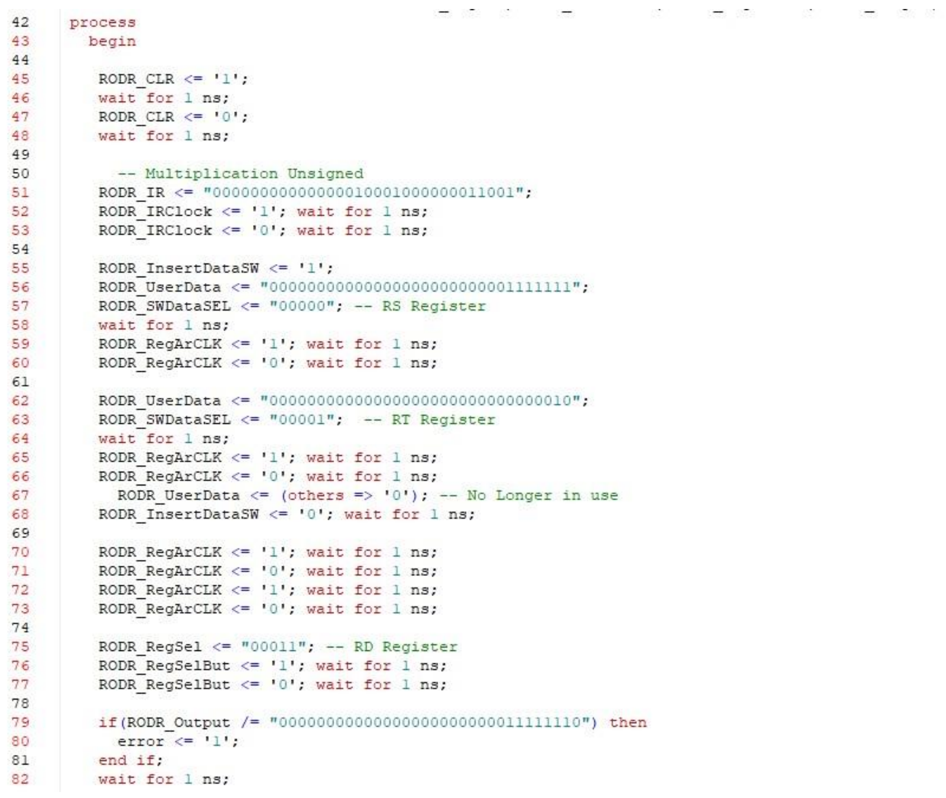
C:/Users/JRodr/Dropbox/Spring 2020 CCNY/CSC343 - Computer Organization Lab/testbench_masterMultDivAcc.vhd (/testbench_mastermultdivacc) - Default

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

Entity testbench_masterMultDivAcc is
  generic(Bits : natural := 32);
End testbench_masterMultDivAcc;

Architecture LogicFunction of testbench_masterMultDivAcc is

Component RODR_MASTERMAP IS
  generic(N : natural := Bits);
  PORT(RODR_InstRegIN : in std_logic_vector(N-1 downto 0);
      RODR_Data: in std_logic_vector(N-1 downto 0);
      RODR_InstRegCLK   : in std_logic;
      RODR_RegArrCLK   : in std_logic;
      RODR_SWDataCLK   : in std_logic;
      RODR_CLRBUT   : in std_logic;
      RODR_RegSel   : in std_logic_vector(4 downto 0);
      RODR_BUTData_SEL : in std_logic_vector(4 downto 0);
      RODR_RegSelBut   : in std_logic;
      RODR_Output: out std_logic_vector(N-1 downto 0) := (others => '0'));
END Component;

  signal RODR_IR              : std_logic_vector (Bits-1 downto 0) := (others => '0');
  signal RODR_userData        : std_logic_Vector(Bits-1 downto 0) := (others => '0');
  signal RODR_IRClock         : std_logic := '0';
  signal RODR_RegArCLK        : std_logic := '0';
  signal RODR_InsertDataSW    :  std_logic := '0';
  signal RODR_CLR             :  std_logic := '0';
  signal RODR_RegSel          :  std_logic_vector(4 downto 0) := (others => '0');
  signal RODR_SWDataSEL       :  std_logic_vector(4 downto 0) := (others => '0');
  signal RODR_RegSelBut       :  std_logic := '0';
  signal RODR_Output          :  std_logic_vector(Bits-1 downto 0);
  signal error : std_logic := '0';

  Begin

    uut : RODR_MASTERMAP PORT MAP (RODR_IR, RODR_UserData, RODR_IRClock, RODR_RegArCLK, RODR_InsertDataSW, RODR_CLR,
                                   RODR_RegSel, RODR_SWDataSEL, RODR_RegSelBut, RODR_Output);
```

Figure 8: Initialization of Testbench code

This is the start of our testbench code. It is the exact same as the previous lab's start as we use the same variables and components.

We will be instantiating the **RODR_MASTERMAP** as it contains all of the connections needed to supplement the correct output. In turn, this file will contain several input signals that can be controlled throughout the program. **RODR_IR** will control the inputs of the instruction register. **RODR_userData** will control the random inputs that go into each register provided by the user. **RODR_IRClock** is used to clock the instruction register. **RODR_RegArrCLK** is used to clock the register array. **RODR_InsertDataSW** is a switch that will determine when new user data can be stored. **RODR_CLR** will clear all data from all registers. **RODR_RegSel** is where the user decides which register they wish to look into. **RODR_SWDataSEL** will determine which register the user data will go into. **RODR_RegSelBut** will be used to output from the register array what the user wants to see based on RegSel. **RODR_Output** will be the output from the register selection.

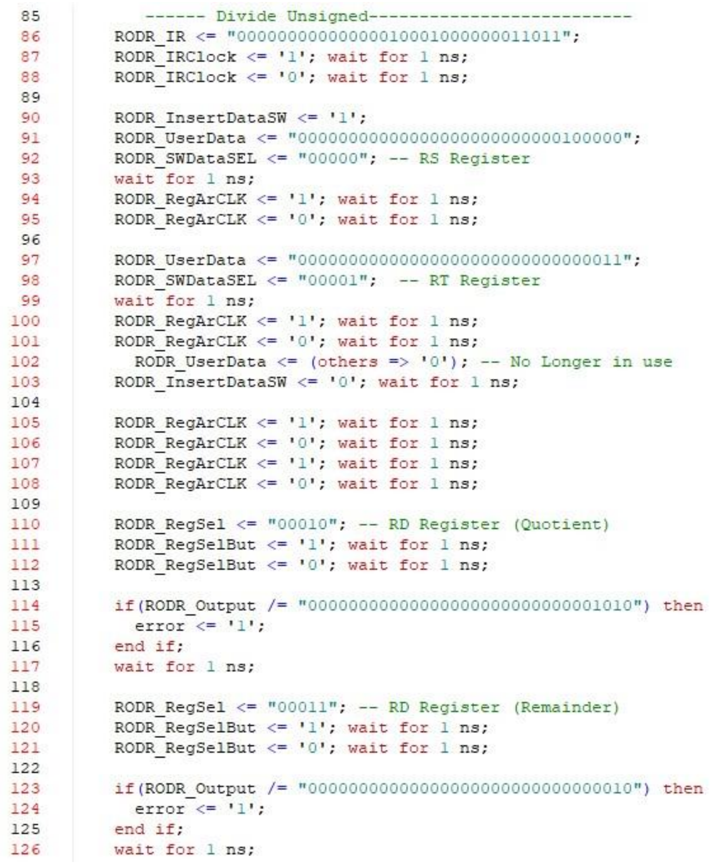


```vhdl
process
  begin

   RODR_CLR <= '1';
   wait for 1 ns;
   RODR_CLR <= '0';
   wait for 1 ns;

     -- Multiplication Unsigned
   RODR_IR <= "00000000000000010001000000011001";
   RODR_IRClock <= '1'; wait for 1 ns;
   RODR_IRClock <= '0'; wait for 1 ns;

   RODR_InsertDataSW <= '1';
   RODR_UserData <= "00000000000000000000000001111111";
   RODR_SWDataSEL <= "00000"; -- RS Register
   wait for 1 ns;
   RODR_RegArCLK <= '1'; wait for 1 ns;
   RODR_RegArCLK <= '0'; wait for 1 ns;

   RODR_UserData <= "00000000000000000000000000000010";
   RODR_SWDataSEL <= "00001";  -- RT Register
   wait for 1 ns;
   RODR_RegArCLK <= '1'; wait for 1 ns;
   RODR_RegArCLK <= '0'; wait for 1 ns;
     RODR_UserData <= (others => '0'); -- No Longer in use
   RODR_InsertDataSW <= '0'; wait for 1 ns;

   RODR_RegArCLK <= '1'; wait for 1 ns;
   RODR_RegArCLK <= '0'; wait for 1 ns;
   RODR_RegArCLK <= '1'; wait for 1 ns;
   RODR_RegArCLK <= '0'; wait for 1 ns;

   RODR_RegSel <= "00011"; -- RD Register
   RODR_RegSelBut <= '1'; wait for 1 ns;
   RODR_RegSelBut <= '0'; wait for 1 ns;

   if(RODR_Output /= "00000000000000000000000011111110") then
     error <= '1';
   end if;
   wait for 1 ns;
```

Figure 9: Multiplication Unsigned of VHDL Code

We start by creating a process that will take hold of all the data changes that has been made. Initially, we wish to clear all registers as they may be full of junk data on startup. We then proceed to go into the Register-to-Register MIPS instruction code, starting with the multiplication signed operation. We first initialize two registers, namely our RS and RT registers to two different user data based on **RODR_SWDataSEL** (for a total of 32 bits each). We then set **RODR_userData** to all logical 0s as it will no longer be in use for the time being. After that, we use **RODR_RegArCLK** to start the computation and then push into the registers we requested RS and RT to be in.

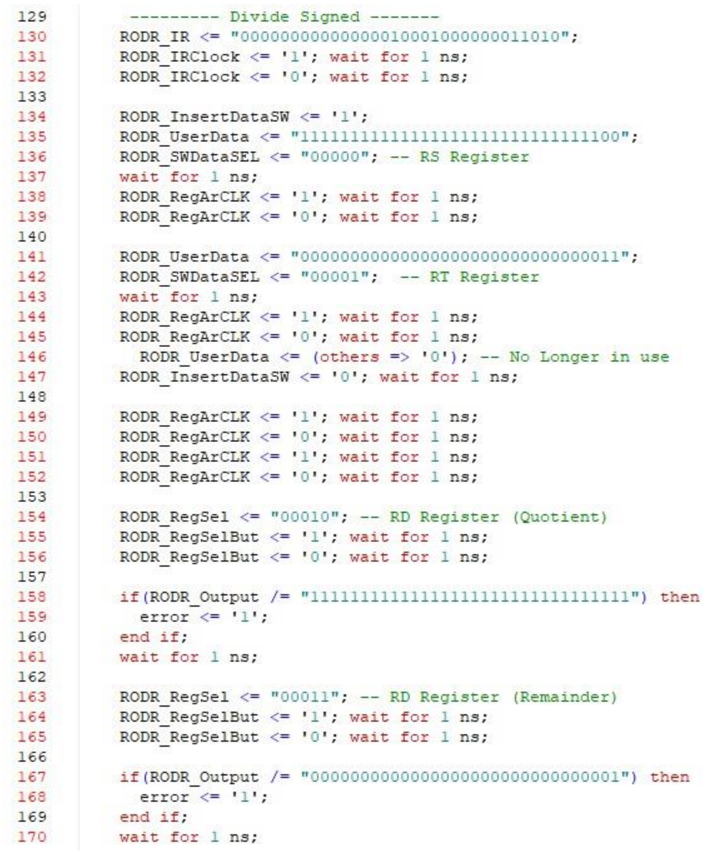
Lastly, we select our RD register as we wish to see the computational result (using **RODR_RegSel** and **RODR_RegSelBut)**. We also have error checking, saying that if the output isn't the one desired, then output an error and stop the simulation.



```
   ------ Divide Unsigned--------------------------
RODR_IR <= "00000000000000010001000000011011";
RODR_IRClock <= '1'; wait for 1 ns;
RODR_IRClock <= '0'; wait for 1 ns;

RODR_InsertDataSW <= '1';
RODR_UserData <= "00000000000000000000000000100000";
RODR_SWDataSEL <= "00000"; -- RS Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_UserData <= "00000000000000000000000000000011";
RODR_SWDataSEL <= "00001";  -- RT Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
  RODR_UserData <= (others => '0'); -- No Longer in use
RODR_InsertDataSW <= '0'; wait for 1 ns;

RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_RegSel <= "00010"; -- RD Register (Quotient)
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "00000000000000000000000000001010") then
  error <= '1';
end if;
wait for 1 ns;

RODR_RegSel <= "00011"; -- RD Register (Remainder)
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "00000000000000000000000000000010") then
  error <= '1';
end if;
wait for 1 ns;
```

Figure 10: Division Unsigned of Testbench Code

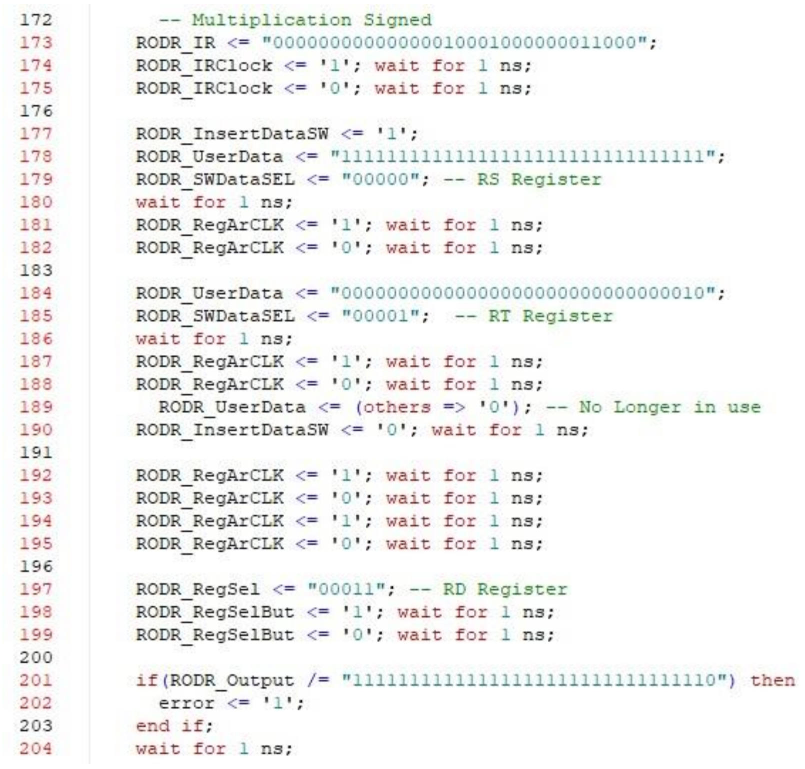
For unsigned division, it is almost identical to our divide unsigned testbench code, but with some changes. The two data that inserts into both RS and RT registers are different values (namely 32 and 3 in decimal, respectively). The way we insert them are the same with the use of **RODR_RegArCLK** being turned on and off. We then also want to output the contents of two registers. Based on the design from the Division Unsigned Section, we ensure that the quotient would go into the register RD specified in the instruction register, while the remainder will go to the adjacent register. This allows you to see the quotient and remainder together for better clarity. The answer should be: Q – 10 and Rem – 2.



```
  --------- Divide Signed -------
RODR_IR <= "00000000000000010001000000011010";
RODR_IRClock <= '1'; wait for 1 ns;
RODR_IRClock <= '0'; wait for 1 ns;

RODR_InsertDataSW <= '1';
RODR_UserData <= "11111111111111111111111111111100";
RODR_SWDataSEL <= "00000"; -- RS Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_UserData <= "00000000000000000000000000000011";
RODR_SWDataSEL <= "00001";  -- RT Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
  RODR_UserData <= (others => '0'); -- No Longer in use
RODR_InsertDataSW <= '0'; wait for 1 ns;

RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_RegSel <= "00010"; -- RD Register (Quotient)
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "11111111111111111111111111111111") then
  error <= '1';
end if;
wait for 1 ns;

RODR_RegSel <= "00011"; -- RD Register (Remainder)
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "00000000000000000000000000000001") then
  error <= '1';
end if;
wait for 1 ns;
```

Figure 11: Division Signed VHDL Testbench Code

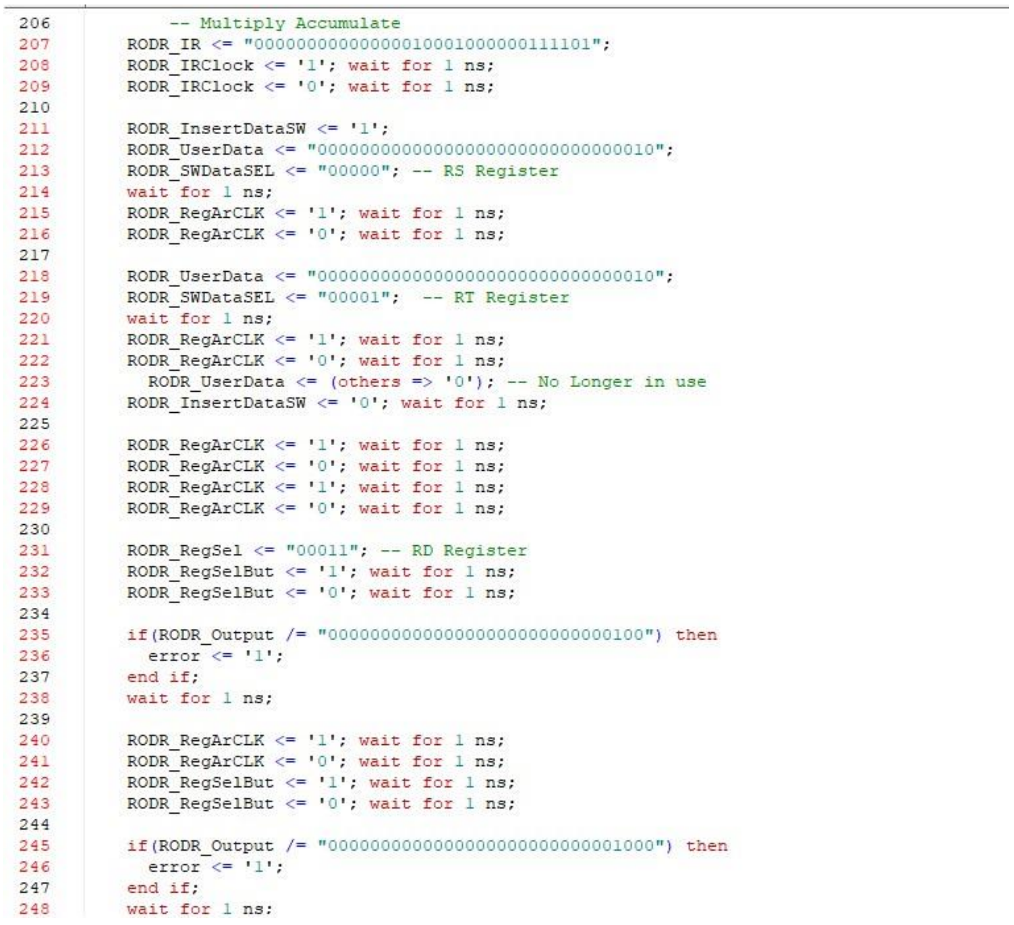
Division signed uses the same coding architecture as Division signed, albeit with only changes with its user data. For the RS register, we now have -4 and for the RT register we have 3 (both in decimal notation). If we don't get -1 as the quotient and 1 as the remainder, then the error flag will be lifted and a message will appear in the terminal window.



```
  -- Multiplication Signed
RODR_IR <= "00000000000000010001000000011000";
RODR_IRClock <= '1'; wait for 1 ns;
RODR_IRClock <= '0'; wait for 1 ns;

RODR_InsertDataSW <= '1';
RODR_UserData <= "11111111111111111111111111111111";
RODR_SWDataSEL <= "00000"; -- RS Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_UserData <= "00000000000000000000000000000010";
RODR_SWDataSEL <= "00001";  -- RT Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
  RODR_UserData <= (others => '0'); -- No Longer in use
RODR_InsertDataSW <= '0'; wait for 1 ns;

RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_RegSel <= "00011"; -- RD Register
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "11111111111111111111111111111110") then
  error <= '1';
end if;
wait for 1 ns;
```

Figure 12: Multiplication Signed Testbench Code

For multiplication signed, we go back to using the same format that multiplication unsigned has only with changes towards the register inputs. The RS register will have -1 in decimal while the RT register has 2. We then check on the lower half of the bits as the maximum answer for 32-bit multiplication is 64. Based on the coding architecture for the section “Multiplication Signed”, we designated the lower half to be the register adjacent to the one specified in the instruction register for clarity. If the answer is not -2, then an error flag will be set.



```
    -- Multiply Accumulate
RODR_IR <= "00000000000000010001000000111101";
RODR_IRClock <= '1'; wait for 1 ns;
RODR_IRClock <= '0'; wait for 1 ns;

RODR_InsertDataSW <= '1';
RODR_UserData <= "00000000000000000000000000000010";
RODR_SWDataSEL <= "00000"; -- RS Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_UserData <= "00000000000000000000000000000010";
RODR_SWDataSEL <= "00001";  -- RT Register
wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
  RODR_UserData <= (others => '0'); -- No Longer in use
RODR_InsertDataSW <= '0'; wait for 1 ns;

RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;

RODR_RegSel <= "00011"; -- RD Register
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "00000000000000000000000000000100") then
  error <= '1';
end if;
wait for 1 ns;

RODR_RegArCLK <= '1'; wait for 1 ns;
RODR_RegArCLK <= '0'; wait for 1 ns;
RODR_RegSelBut <= '1'; wait for 1 ns;
RODR_RegSelBut <= '0'; wait for 1 ns;

if(RODR_Output /= "00000000000000000000000000001000") then
  error <= '1';
end if;
wait for 1 ns;
```

Figure 13: Multiplication Accumulation Testbench Code

The multiplication accumulator will also retain the structure of its multiplication unsigned and signed counterparts. The main difference here is that the instruction has changed "111101" for the FUNCT portion along with the user data and the amount of times we run the simulation. We set both RS and RT registers to 2 in decimal for simplicity. We then use the lower half of the 64-bit answer as this is where everything up to 2^31-1 in decimal will be stored. To make the accumulator function, we must always clock the register array we created via **RODR_RegArCLK** and use **RODR_RegSelBut** to notice the changes that have been made inside the register.

Data Waveforms

Waveforms

- Note that for ALL waveforms, RODR_CLRBUT is initially turned on to clear all registers from having unnecessary data.
- Also note that to make operators occur, we must press the Register Clock:
  - Once for every data input (with the Switch for Data at '1')
  - Twice for the completion of arithmetic between registers
    - 1 Time for operation completion, the other to shift to register and output the result
- We will be storing data between Register 0 ('00000') and Register 1 ('00001') for Register-to-Register Arithmetic
  - Register 0: RS
  - Register 1: RT
  - Register 2: RD
  - Register 3: RD+1 (used to see Remainder, lower half of 32-bit multiplication)

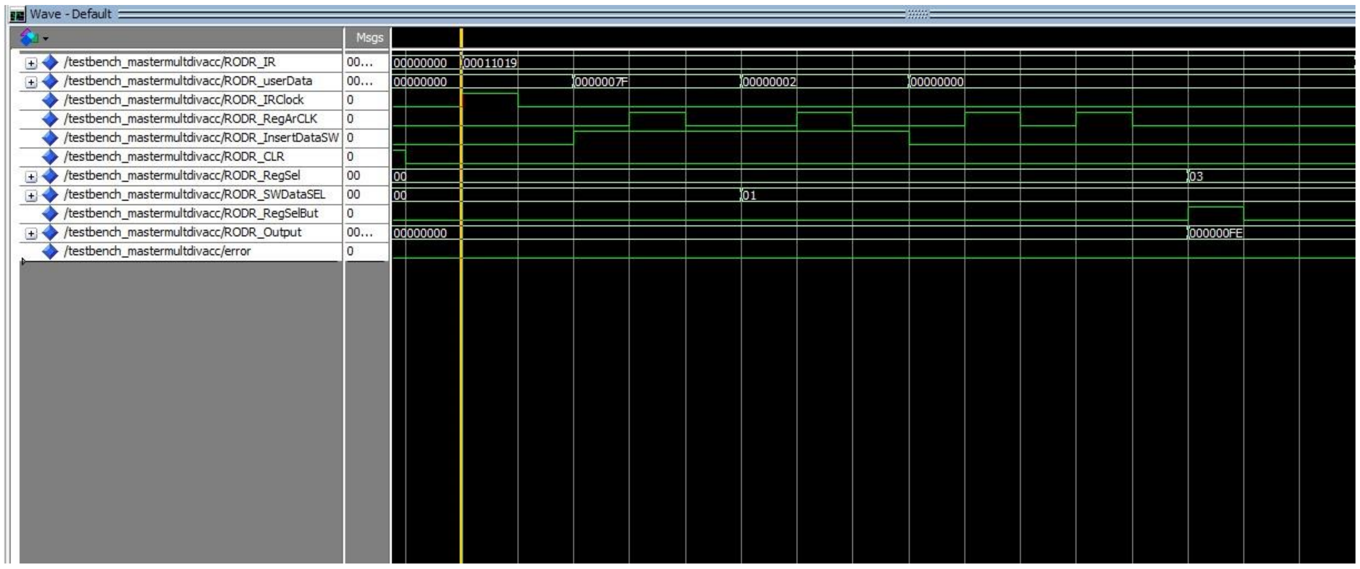

Wave - Default
Msgs
/testbench_mastermultdivacc/RODR_IR 00... 00000000 00011019
/testbench_mastermultdivacc/RODR_userData 00... 00000000 0000007F 00000002 00000000
/testbench_mastermultdivacc/RODR_IRClock 0
/testbench_mastermultdivacc/RODR_RegArCLK 0
/testbench_mastermultdivacc/RODR_InsertDataSW 0
/testbench_mastermultdivacc/RODR_CLR 0
/testbench_mastermultdivacc/RODR_RegSel 00 00 03
/testbench_mastermultdivacc/RODR_SWDataSEL 00 00 01
/testbench_mastermultdivacc/RODR_RegSelBut 0
/testbench_mastermultdivacc/RODR_Output 00... 00000000 000000FE
/testbench_mastermultdivacc/error 0

Figure 14: Multiplication Unsigned Waveform

Based on this waveform, we notice that the instruction register is inserted using **RODR_IRClock**. We then send in 0x0000007F as the first data that goes into register 0 and 0x00000002 into register 1. We then compute them using the **RODR_RegArCLK** and then use **RODR_RegSel** and **RODR_RegSelBut** to display the output of the lower 32-bits of the multiplication, in this case being 0x000000FE (or 254 in decimal notation). Based on the last line, no errors have been detected for this computation.

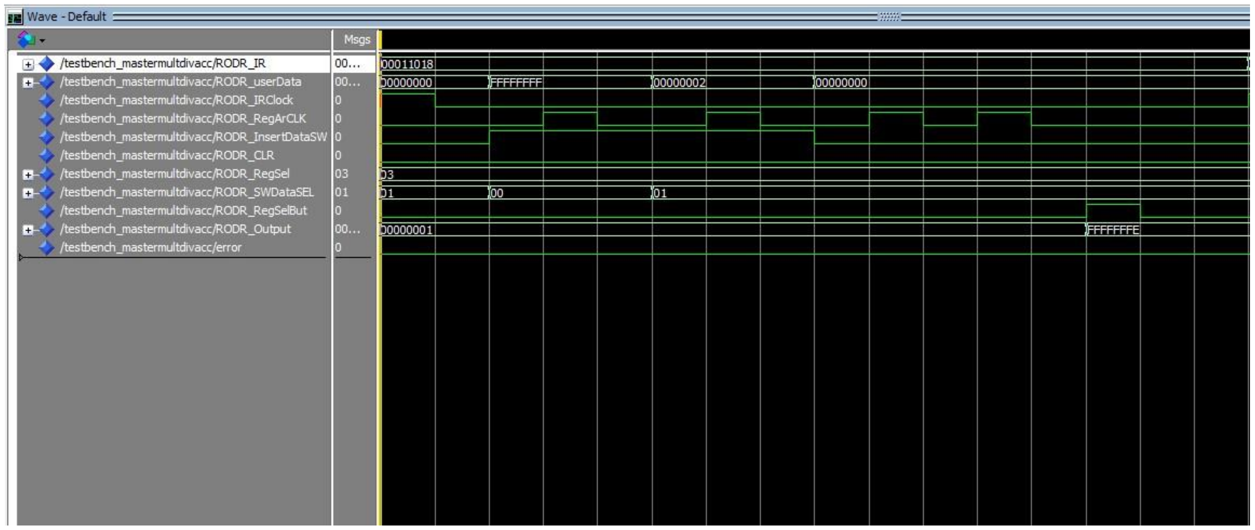

Wave - Default
Msgs
/testbench_mastermultdivacc/RODR_IR
/testbench_mastermultdivacc/RODR_userData
/testbench_mastermultdivacc/RODR_IRClock
/testbench_mastermultdivacc/RODR_RegArCLK
/testbench_mastermultdivacc/RODR_InsertDataSW
/testbench_mastermultdivacc/RODR_CLR
/testbench_mastermultdivacc/RODR_RegSel
/testbench_mastermultdivacc/RODR_SWDataSEL
/testbench_mastermultdivacc/RODR_RegSelBut
/testbench_mastermultdivacc/RODR_Output
/testbench_mastermultdivacc/error
00011018
00000000
FFFFFFFF
00000002
00000000
03
01
00
01
00000001
FFFFFFFE

Figure 15: Multiplication Signed Waveform

Based on this waveform, we notice that the instruction register is inserted using **RODR_IRClock**. We then send in 0xFFFFFFFF as the first data that goes into register 0 and 0x00000002 into register 1. We then compute them using the **RODR_RegArCLK** and then use **RODR_RegSel** and **RODR_RegSelBut** to display the output of the lower 32-bits of the multiplication, in this case being 0xFFFFFFFE (or -2 in decimal notation). Based on the last line, no errors have been detected for this computation.

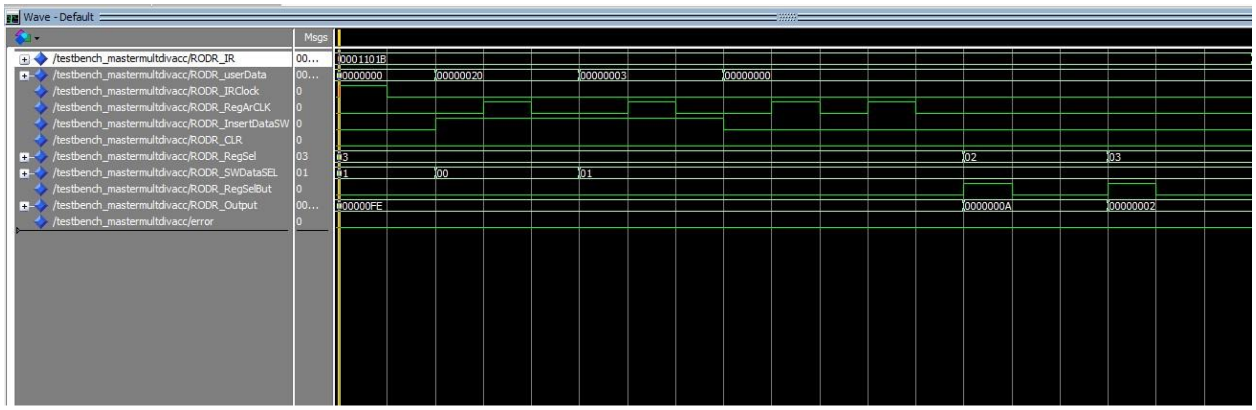

Wave - Default
Msgs
/testbench_mastermultdivacc/RODR_IR
/testbench_mastermultdivacc/RODR_userData
/testbench_mastermultdivacc/RODR_IRClock
/testbench_mastermultdivacc/RODR_RegArCLK
/testbench_mastermultdivacc/RODR_InsertDataSW
/testbench_mastermultdivacc/RODR_CLR
/testbench_mastermultdivacc/RODR_RegSel
/testbench_mastermultdivacc/RODR_SWDataSEL
/testbench_mastermultdivacc/RODR_RegSelBut
/testbench_mastermultdivacc/RODR_Output
/testbench_mastermultdivacc/error
0001101B
00000020
00000003
00000000
02
03
00
01
000000FE
0000000A
00000002

Figure 16: Division Unsigned Waveform

Based on this waveform, we notice that the instruction register is inserted using **RODR_IRClock**. We then send in 0x00000020 (or 32 in decimal notation) as the first data that goes into register 0 and 0x00000003 (or 3 in decimal notation) into register 1. We then compute them using the **RODR_RegArCLK** and then use **RODR_RegSel** and **RODR_RegSelBut** to display two outputs. 0x02 defines register 2 from the array that stores the quotient and 0x03 defines register 3 that stores the remainder. When doing division between register 0 and register

1, we get 0x0A (10 in decimal notation) as the quotient and 0x02 (2 in decimal notation) as the remainder, respectively. Based on the last line, no errors have been detected for this computation.

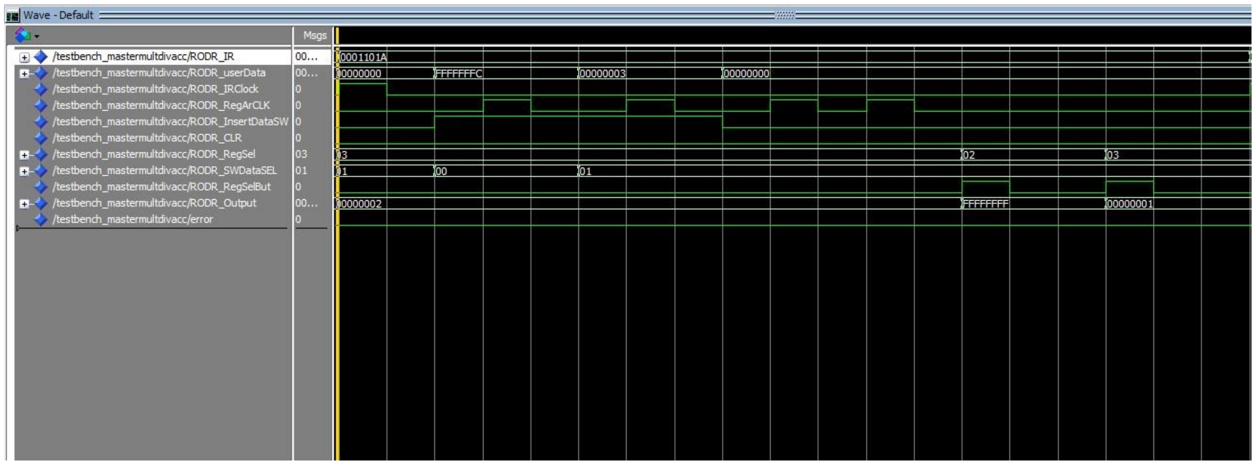

Wave - Default
Msgs
/testbench_mastermultdivacc/RODR_IR
/testbench_mastermultdivacc/RODR_userData
/testbench_mastermultdivacc/RODR_IRClock
/testbench_mastermultdivacc/RODR_RegArCLK
/testbench_mastermultdivacc/RODR_InsertDataSW
/testbench_mastermultdivacc/RODR_CLR
/testbench_mastermultdivacc/RODR_RegSel
/testbench_mastermultdivacc/RODR_SWDataSEL
/testbench_mastermultdivacc/RODR_RegSelBut
/testbench_mastermultdivacc/RODR_Output
/testbench_mastermultdivacc/error
0001101A
00000000
FFFFFFFC
00000003
00000000
03
02
03
01
00
01
00000002
FFFFFFFF
00000001

Figure 17: Division Signed Waveform

Based on this waveform, we notice that the instruction register is inserted using **RODR_IRClock**. We then send in 0xFFFFFFFC (-4 in signed decimal notation) as the first data that goes into register 0 and 0x00000003 (3 in signed decimal notation) into register 1. We then compute them using the **RODR_RegArCLK** and then use **RODR_RegSel** and **RODR_RegSelBut** to display two outputs. 0x02 defines register 2 from the array that stores the quotient and 0x03 defines register 3 that stores the remainder. When doing division between register 0 and register 1, we get 0xFFFFFFFF (or -1 in signed decimal notation) as the quotient and 0x00000001 (1 in signed decimal notation) as the remainder, respectively. Based on the last line, no errors have been detected for this computation.

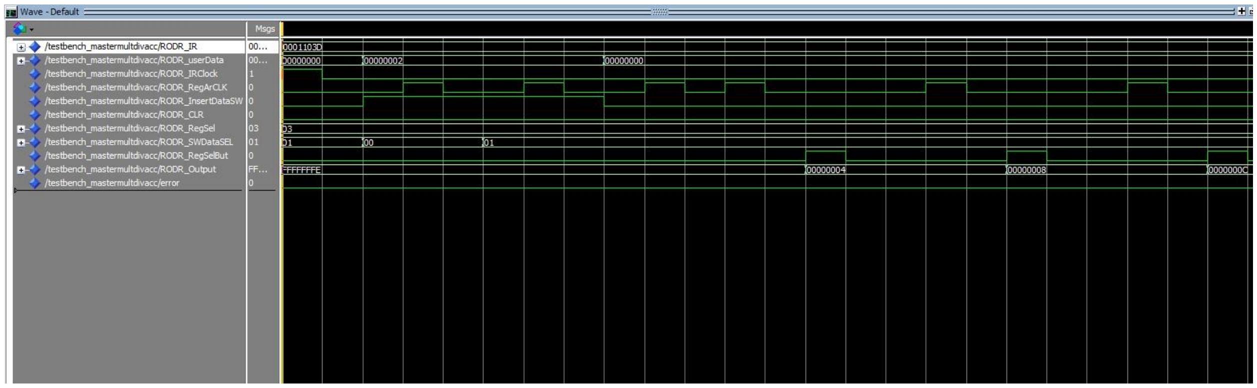

Wave - Default
Msgs
/testbench_mastermultdivacc/RODR_IR
/testbench_mastermultdivacc/RODR_userData
/testbench_mastermultdivacc/RODR_IRClock
/testbench_mastermultdivacc/RODR_RegArCLK
/testbench_mastermultdivacc/RODR_InsertDataSW
/testbench_mastermultdivacc/RODR_CLR
/testbench_mastermultdivacc/RODR_RegSel
/testbench_mastermultdivacc/RODR_SWDataSEL
/testbench_mastermultdivacc/RODR_RegSelBut
/testbench_mastermultdivacc/RODR_Output
/testbench_mastermultdivacc/error
0001103D
00000000
00000002
00000000
03
01
00
01
FFFFFFFE
00000004
00000008
0000000C

Figure 18: Multiplication Accumulation Waveform

Based on this waveform, we notice that the instruction register is inserted using **RODR_IRClock**. We then send in 0x00000002 as the first data that goes into register 0 and 0x00000002 into register 1. We then compute them using the **RODR_RegArCLK** and then use **RODR_RegSel** and **RODR_RegSelBut** to display several instances of the change within register 3. It houses the lower 32 bits of the multiplication between the two registers. As we continue to toggle **RODR_RegArCLK**, you will notice that the output will change according to their amount, increasing by a base of 4. Based on the last line, no errors have been detected for this computation.

Conclusion

In this laboratory exercise, I have learned how to create multiplication and division operations using adders and bit shifts to the left and right. We can accomplish division by checking for subtraction between the two operands. If we can do subtraction, then we can divide them and continue shifting until the operation is done. If we cannot we continue shifting. Multiplication provides for a similar approach when adding up all the results into one combined answer using the elementary way of multiplying integers. The multiplication accumulator provides for these two things, as they also multiply, but we add the multiplications together with the max capacity being 2^63 – 1. For division, it should only be a 32-bit answer since both RS and RT registers will never have more than 32-bits.