Department of Electrical and Computer Engineering,
Gina Cody School of Engineering and Computer Science,
Concordia University, Montreal, Canada

COEN 6501: Digital Design and Synthesis

Design of an Arithmetic Unit Performing Z=1/4 [A*B] +1

Abishek Arumugam Thiruselvi - 40218896

Manish Pejathaya - 40194909

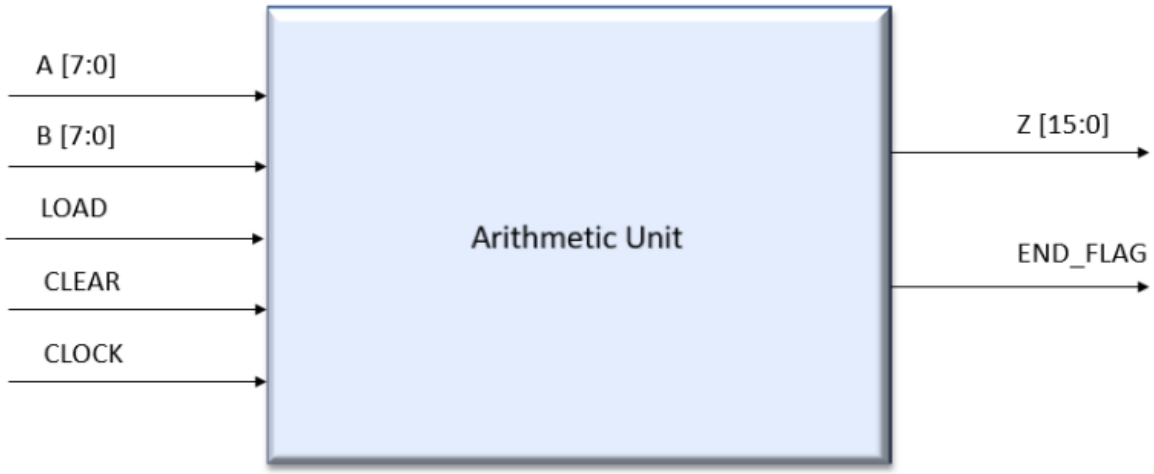
Mahsa Vahed - 40183415

November 2022

Abstract

The objective of this project is to create an arithmetic device that can compute Z=1/4 [A*B] + 1. Two 16-bit unsigned operands, A and B, will be provided to the unit. When LOAD = 1, the data is latched by two clock-synchronous internal registers (RA and RB). The device produces results in a 32-bit format. END FLAG is set high when the output is produced. In response to an asynchronous CLEAR signal, all registers are set to 0. Array multiplier is selected for the multiplication operation.

Expansion of the method for 16 bits operands and Pipelining of the design are the Extra features for our project.



Contents

Table of Figures

Introduction

In arithmetic logic, multiplication is an important operation. The arithmetic-logic unit, a section of a central processing unit, carries logic and arithmetic functions on the operands of computer instruction.

The purpose of this project is to design an arithmetic unit capable of calculating Z=1/4 [A×B] +1. This unit receives two unsigned 16-bit operands A and B. Two clock synchronous internal registers RA and RB latch the data when LOAD = 1. The unit outputs the results in a 16-bit register RZ output port, and END_F rises. An asynchronous CLEAR signal will clear all the registers to '0'. Expansion of the method for 32 bits operands and Pipelining of the design are added to the project as extra features. The product of (AxB) is computed and stored using flip flop if the control flag is high in stage 1, The division by 4 of the stage 1 result is computed and stored using flip flop if the control flag is high in stage 2, and the result of stage 2 is incremented by one and sent to mux to get the result Z in stage 3.

Signal Specifications

- A: Unsigned 16-bit Operand A
- B: Unsigned 16-bit Operand B
- Z: Signed Output
- CLEAR: Clears selected registers
- LOAD: Loads Operand into internal registers
- CLOCK: Input Clock
- END_F: Indicates end of operation

Proposed Algorithm in Pipeline

Data Path Stages are described as follows:

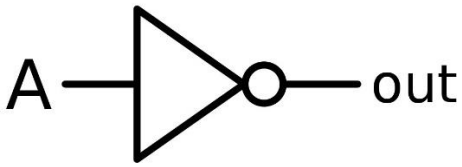
- Stage 1: The product of AxB is computed and stored using flip flop if the control flag is high
- Stage 2: The division by 4 of the stage 1 result is computed and stored using a flip flop if the control flag is high.
- Stage 3: The result of stage 2 is incremented by one and sent to mux to get the final result Z.

DESIGN AND ANALYSIS OF COMPONENTS

We use the basic logic gates. the following illustrations and tables show the circuit symbol and logic combinations of logic gates.

NOT

The NOT gate, also known as an inverter, is a logic gate used in digital logic to implement logical negation. In mathematical logic, it is comparable to the logical negation operator. [1]



Truth Table

A	OUT
0	1
1	0

Figure 1: NOT Gate

AND

An AND gate is a logic gate. And it has two or more inputs and a single output. An AND gate operates on logical multiplications. In this gate, if one of the inputs is low or 0, then the output is also 0. If all the inputs are high or 1, then the output will be 1 or high. [1]

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

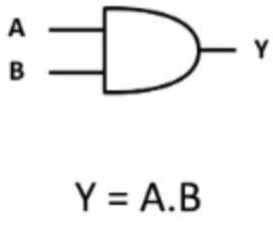
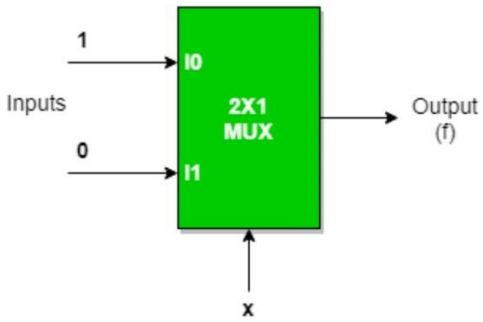


Figure 2:AND Gate

Mux2:1

Multiplexer is a combinational circuit that have many inputs and single output. Output is based on the control or select inputs. [1]



Truth Table

x	f
0	1
1	0

Figure 3: MUX2:1 Gate

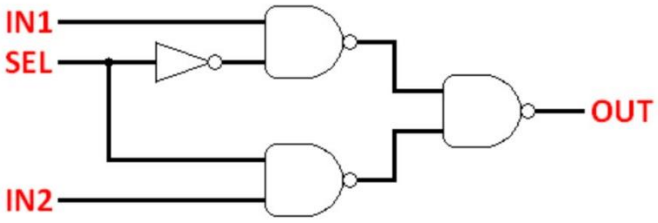


Figure 4: Mux Detailed Gate

D-Flip Flop

A latch, often known as a flip-flop, is a circuit with two stable states that can be used to store state data. Signals which is applied to one or more inputs and one or two outputs allow the circuit to altering state. It guarantees that both inputs, S and R, are never equal to one at the same time. One of the key uses of a D flip flop is to add delay to a timing circuit so that data can be sampled at predetermined intervals.[1]

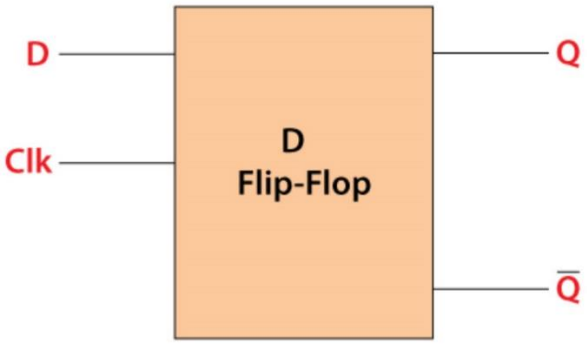


Figure 5: DFF Gate

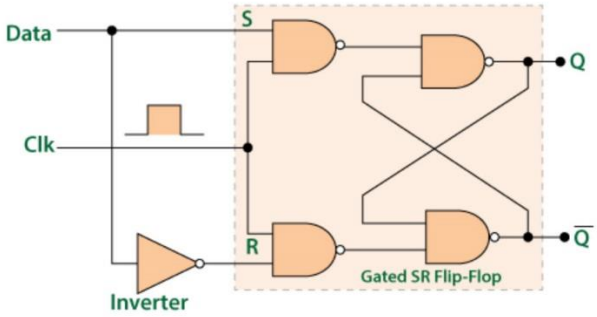


Figure 6: DFF Detailed Gate

Full Adder

Full Adder is an adder that adds three inputs and generates two outputs. The inputs are A and B and carry as C-in. The output carry is C-out and the normal S, which is SUM. [2]

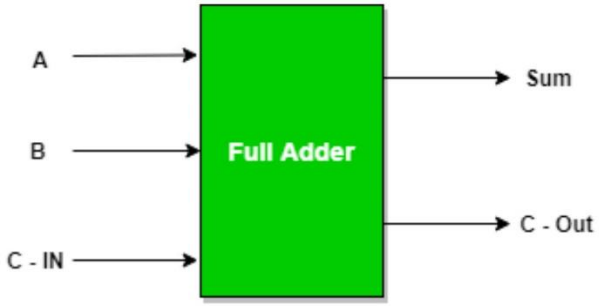


Figure 7: Full Adder

Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 8: Full Adder Truth Table

RTL synthesis and Testing of various built combinational components

All RTL syntheses are compiled and synthesized by Mentor Graphics Precision RTL Synthesis. we described all gates we have used in our project with their functionality in above parts. In this part we will display all synthesis files.

DFF

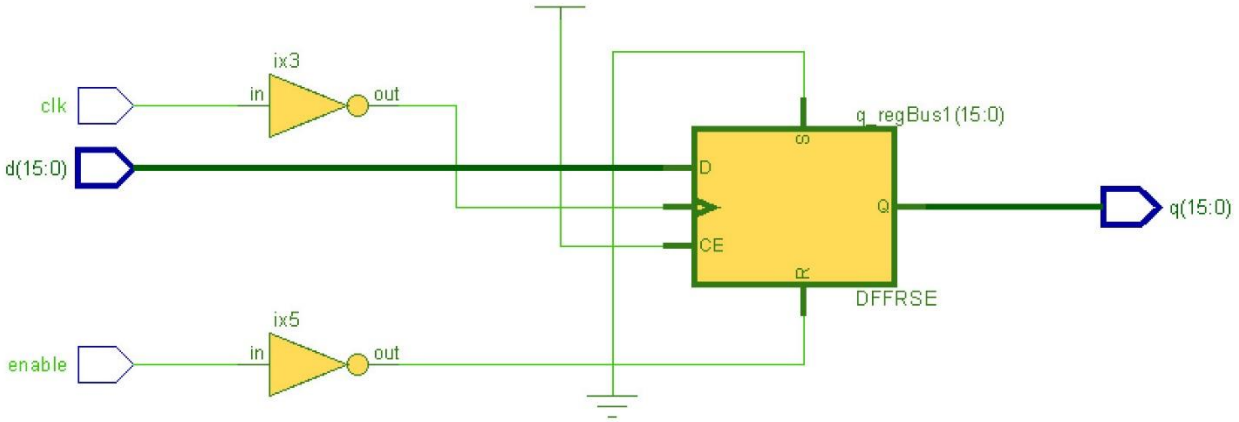


Figure 9: DFF16 Synthesis

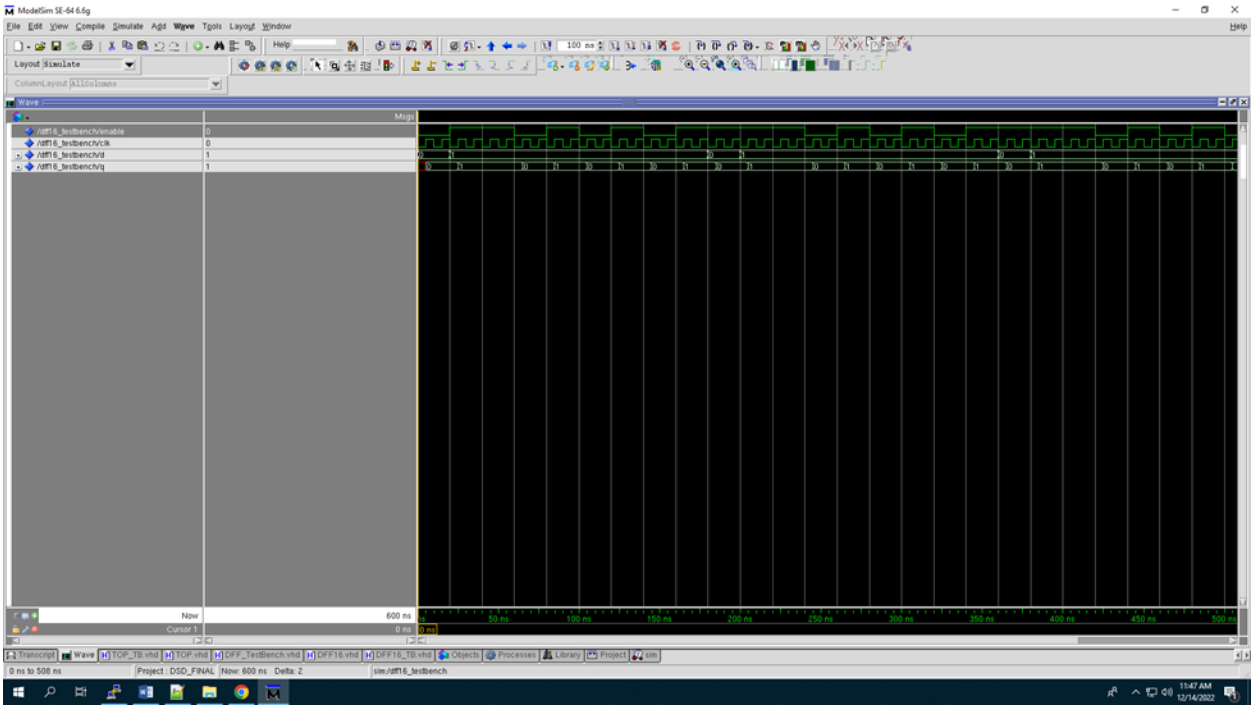


Figure 10: Simulation Result for DFF16

MUX

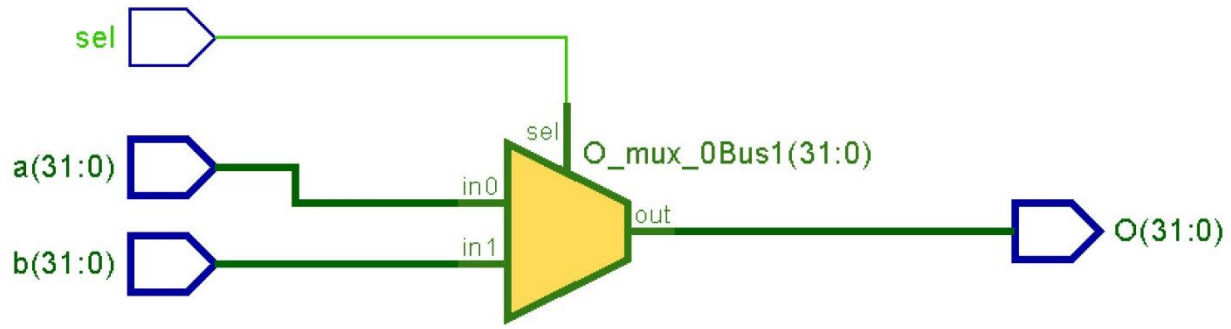


Figure 11: MUX Synthesis

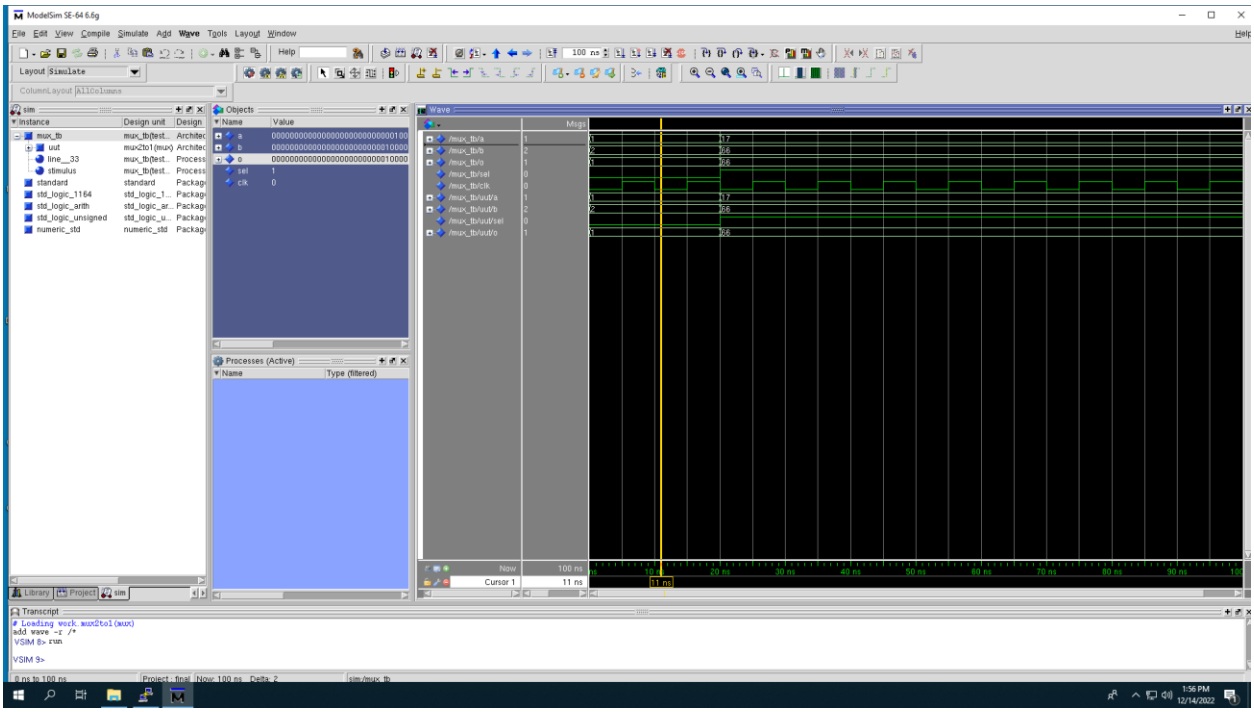


Figure 12: Simulation Result for MUX

FULL ADDER

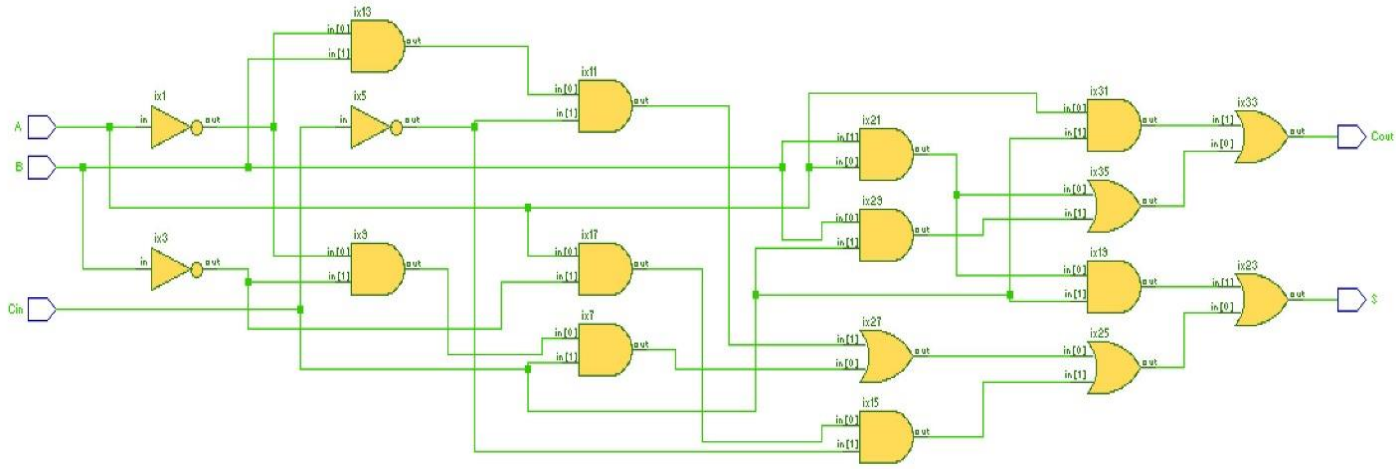


Figure 13: Full Adder Synthesis

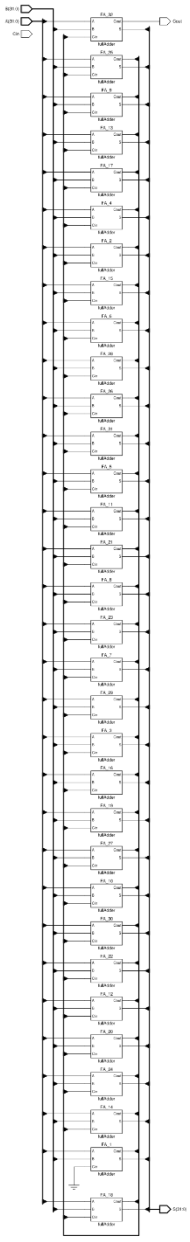


Figure 14: Full Adder32 Synthesis

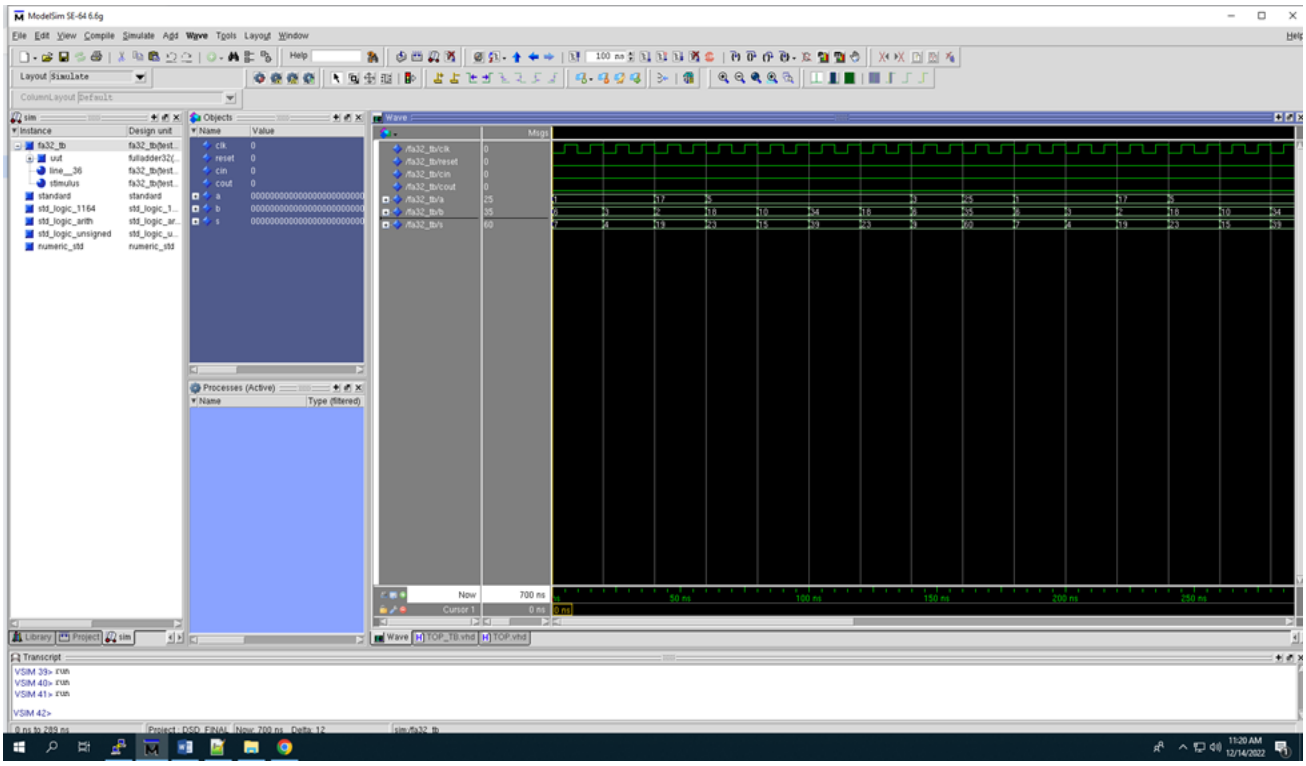


Figure 15: Simulation Result for Full Adder32

Multiplier

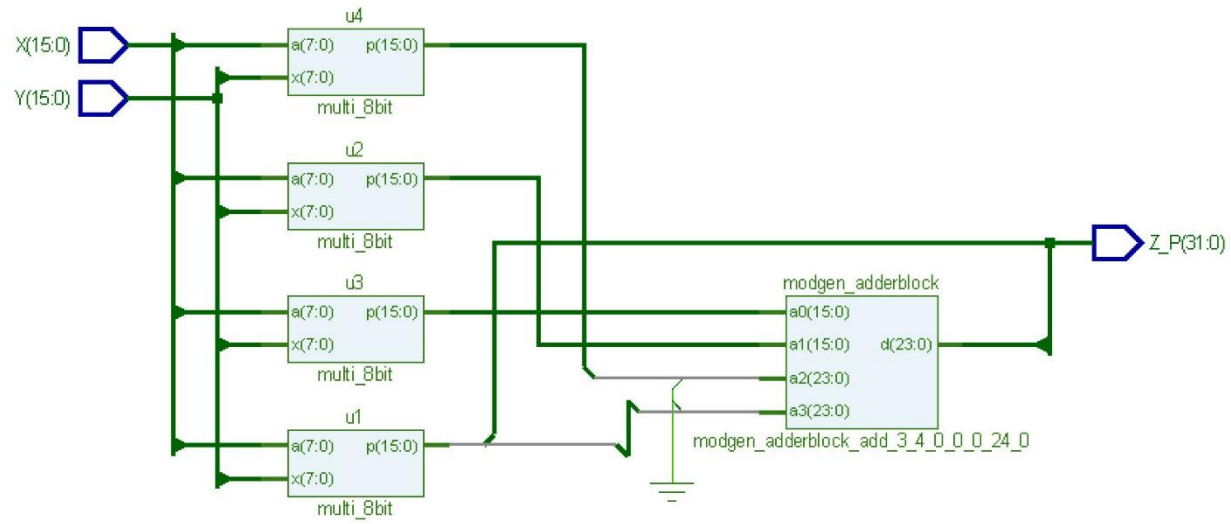


Figure 16: MULTIPLIER 16X16 Synthesis

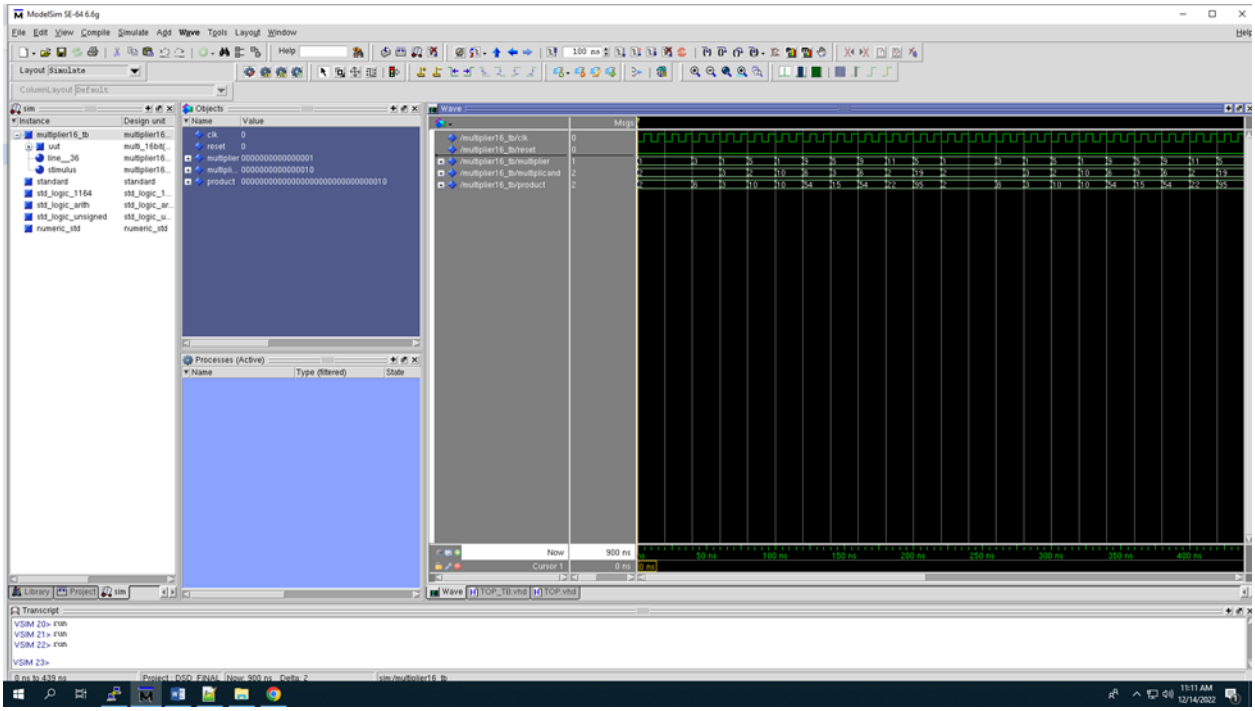


Figure 17: Simulation Result for MULTIPLIER 16X16

Multiplier Design

Design of an 8-bit Multiplier

The Multiplication algorithm used to multiply an 8-bit multiplier and an 8-bit multiplicand is as follows:

Consider we have to multiply the following two values: A(7 downto 0): A7, A6, A5, A4, A3 A2, A1, A0 and B(7 downto 0): A7,A6, A5, A4, A3, A2, A1, A0.

We make use of the AND gate to generate partial products (PP). Since we have an 8-bit multiplicand and multiplier, we will be generating, 8*8, i.e., 64 Partial products.

```
                                        A7    A6    A5    A4    A3    A2    A1    A0
                                    x   B7    B6    B5    B4    B3    B2    B1    B0
                                    ------------------------------------------------
                                        A7.B0 A6.B0 A5.B0 A4.B0 A3.B0 A2.B0 A1.B0 A0.B0
                                  +   A7.B1 A6.B1 A5.B1 A4.B1 A3.B1 A2.B1 A1.B1 A0.B1
                              +   A7.B2 A6.B2 A5.B2 A4.B2 A3.B2 A2.B2 A1.B2 A0.B2
                          +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B3 A2.B3 A1.B3 A0.B3
                      +   A7.B4 A6.B4 A5.B4 A4.B4 A3.B4 A2.B4 A1.B4 A0.B4
                  +   A7.B5 A6.B5 A5.B5 A4.B5 A3.B5 A2.B5 A1.B5 A0.B5
              +  A7.B6 A6.B6 A5.B6 A4.B6 A3.B6 A2.B6 A1.B6 A0.B6
          +A7.B7 A6.B7 A5.B7 A4.B7 A3.B7 A2.B7 A1.B7 A0.B7
          ------------------------------------------------------------------------------
          P15   P14   P13   P12   P11   P10   P9    P8    P7    P6    P5    P4    P3    P2    P1    P0
```

Patial Products to be added

The equation for the addition is: $P(m+n) = A(m)B(n) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j}$.

Figure 18: Partial Product Design

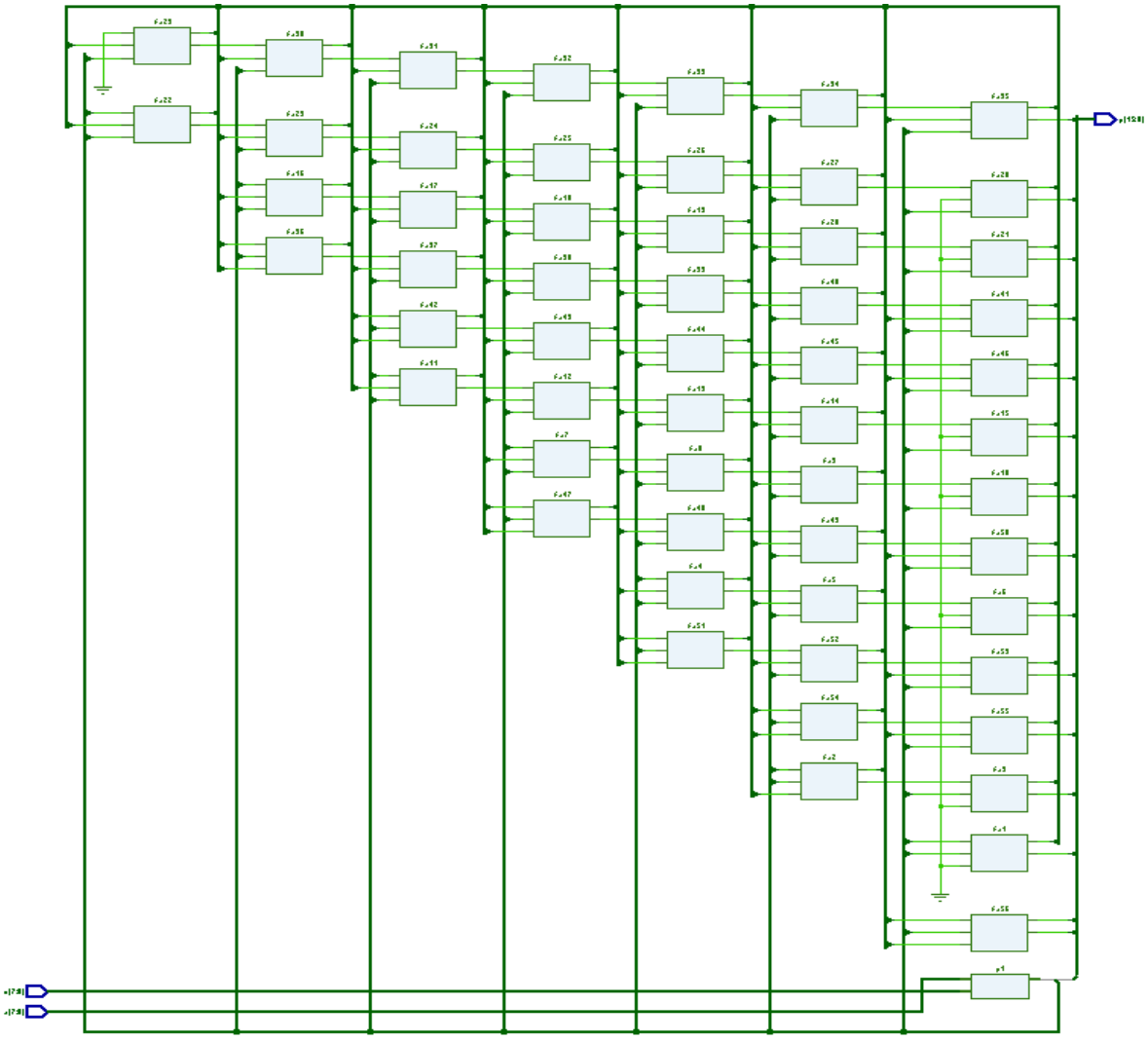


Figure 19: 8-bit Array Multiplier Synthesis

The requirement for designing an 8-bit multiplier is as follows:

REQ1: 64 AND gates.

REQ2: 56 Full adders.

Design of a 16-bit multiplier using 8-bit multiplier

The Following is achieved by dividing the two 16-bit numbers, say m and n, into 8-bit numbers, I.e., mHigh, mLow and nHigh, nLow. The algorithm and the code are shown in the Figure below.

```
                                    [mHigh mLow]    x
                                    [nHigh nLow]
                                    - - - - - - - -
                        [mHigh * nLow] [mLow * nLow]
          [mHigh * nHigh] [mLow * nHigh]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[mHigh * nHigh] + [mLow * nHigh + mHigh * nLow] + [mLow * nLow]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
-- 8x8 to 16x16

u1: multi_8bit PORT MAP ( mLow, nLow, mLow_nLow);
u2: multi_8bit PORT MAP ( mHigh, nLow, mHigh_nLow);
u3: multi_8bit PORT MAP ( mLow, nHigh, mLow_nHigh);
u4: multi_8bit PORT MAP ( mHigh, nHigh, mHigh_nHigh);

mHigh_nHigh_32(15 DOWNTO 0) <= mHigh_nHigh(15 DOWNTO 0);

first <= to_integer(unsigned(mLow_nLow));
second <= to_integer(unsigned(mHigh_nLow));
third <= to_integer(unsigned(mLow_nHigh));
fourth <= to_integer(unsigned(mHigh_nHigh_32));

sum <= fourth + (third + second)*256 + first;
Z_P <= STD_LOGIC_VECTOR(to_signed(sum, 32));

END MULTIPLIER_16;
```

Figure 20: Design of a 16-bit multiplier using 8-bit multiplier

Technical implementation

Implementation of CLEAR operation

When the clear flag is high, the inputs are multiplied by 0.

```
BEGIN
  A_IP <= ( 15 DOWNTO 0 => (NOT CLR)) AND A;
  B_IP <= ( 15 DOWNTO 0 => (NOT CLR)) AND B;
```

Implementation of LOAD operation

First, all the input flags are set. Then the inputs are loaded into the register when the load is high.

```
 -- LOAD AT POS
 LD_POS <= LOAD;
 CLR_POS <= CLR;
 LOAD_IN <= LD_POS;

 -- LOAD INPUTS IN FLIP FLOP
 FF0_A_IP : DFF16 PORT MAP ( d => A_IP, enable => LOAD_IN, clk => CLK, q => A_REG);
 FF0_B_IP : DFF16 PORT MAP ( d => B_IP, enable => LOAD_IN, clk => CLK, q => B_REG);
```

Implementation of the END FLAG

The End flag is set and passed into a flip-flop as a control signal for the next stage.

```
-- END FLAG SETTING
END_FLAG <= CLK OR LD_POS;
--FF0_LD_EF: DFF PORT MAP  (d => LD_POS, enable => FF0_LD_EN, clk => CLK, q => END_FLAG);
```

Implementation of the Arithmetic Unit

STEP 1. The clear flag is set and pushed to the register

STEP 2. 16-bit multiplication with output as PRODUCT_AB.

```
FF0_CLR_CF: DFF PORT MAP  (d => CLR, enable => FF0_CLR_EN, clk => CLK, q => CLR_FLAG);

-- EQUATION STAGES 1/4[A x B]+1

-- A x B USING 16BIT MULTIPLIER

AB_MULTI_16 : multi_16bit PORT MAP (X => A_REG, Y => B_REG, Z_P => PRODUCT_AB);
```

STEP 3. PRODUCT_AB is pushed into FF in stage 1 to propagate to the next stage.

STEP 4. Flags are passed to next stage

```
S1_AxB_FF : DFF32 PORT MAP (d => PRODUCT_AB, enable => S0_EN_FF, clk => CLK, q => S1_AxB);
S1_E_FLAG : DFF PORT MAP (d => END_FLAG, enable => END_FLAG, clk => CLK, q => S1_END_FLAG);
S1_C_FLAG : DFF PORT MAP (d => CLR_FLAG, enable => CLR_FLAG, clk => CLK, q => S1_CLR_FLAG);

S1_EN_FF <= S1_END_FLAG;
```

STEP 5. The division operation is performed by concatenation with output as S2_SR_DIV_VAL, stage2.

STEP 6. S2_SR_DIV_VAL pushed into FF in stage 2 to propagate to the next stage.

STEP 7. Flags are passed to the next stage.

```
S2_SR_FF: DFF32 PORT MAP(d => S2_SR_DIV_VAL, enable => S1_EN_FF, clk => CLK, q => S2_SR_OP);
S2_E_FLAG : DFF PORT MAP (d => S1_END_FLAG, enable => S1_END_FLAG, clk => CLK, q => S2_END_FLAG);
S2_C_FLAG : DFF PORT MAP (d => S1_CLR_FLAG, enable => S1_CLR_FLAG, clk => CLK, q => S2_CLR_FLAG);

S2_EN_FF <= S2_END_FLAG;
```

STEP 8. Contains decimal ‘1’ in the binary format of 32-bit.

STEP 9. The Addition operation of 1 and FF output from stage 2 is performed here:

```
-- STAGE 3 ADDITION
S3_ADD_OPERAND <= (0 => '1', OTHERS => '0');

S3_ADD_FA : FullAdder32 PORT MAP (A => S2_SR_OP, B => S3_ADD_OPERAND, Cin => '0', S => Z_OUT, Cout => FA_32_COUT);

MUX_OUT : MUX2TO1 PORT MAP (a => Z_OUT, b => (OTHERS => '0'), sel => S2_CLR_FLAG, O => Z_REG);

Z <= Z_REG;
```

TOP-Level Result

The figure below shows the final simulation result for the TOP level implementation.

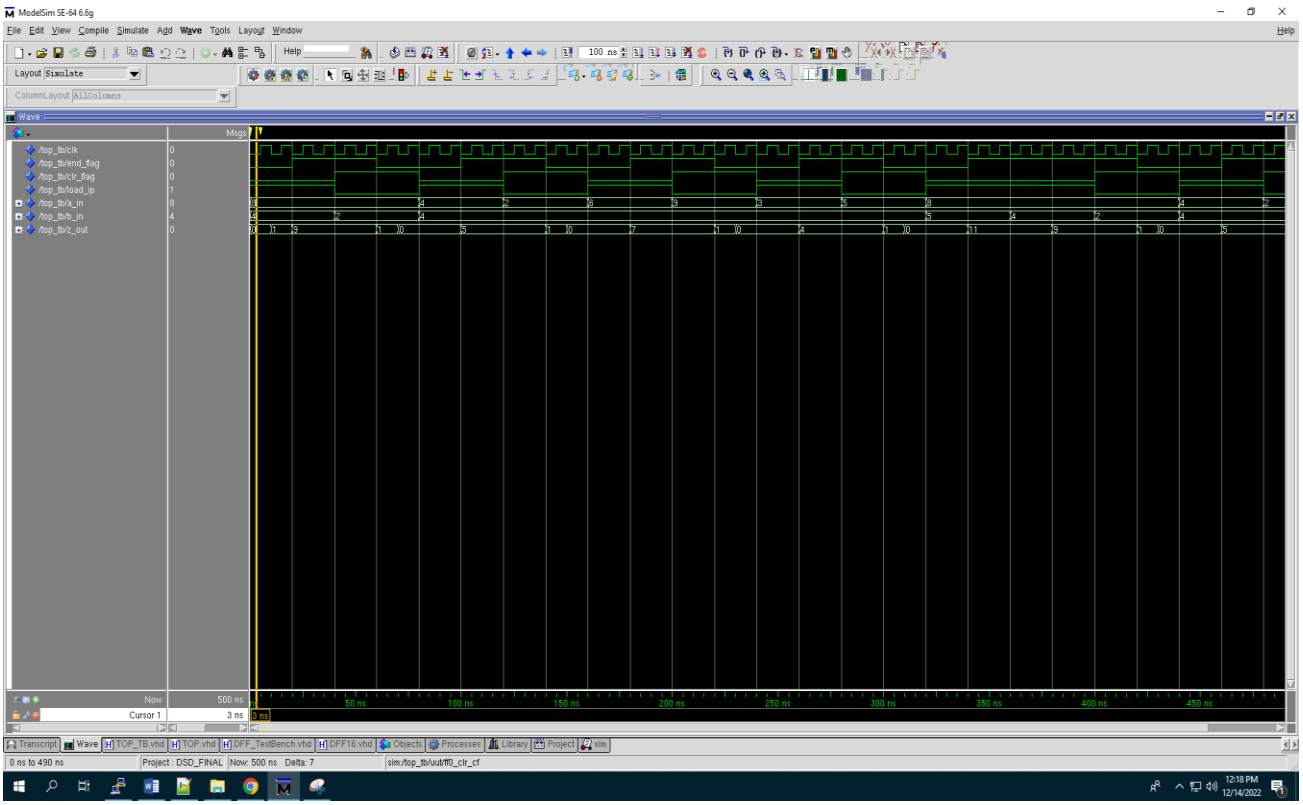


Figure 21: Simulation Result of TOP-Level

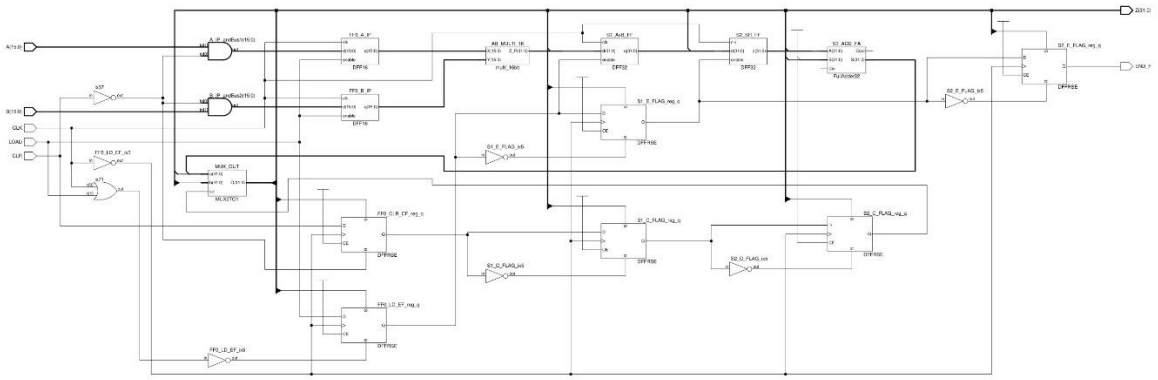


Figure 22: TOP-Level Synthesis

Precision RTL Area Report

```
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//              Portions copyright 1991-2008 Compuware Corporation
//                    UNPUBLISHED, LICENSED SOFTWARE.
//           CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux m_vah@poise.encs.concordia.ca #1 SMP Wed Nov 2 16:40:42 CDT 2022 3.10.0-1160.80.1.el7.x86_64 x86_64
//
//  Start time Wed Dec 14 17:39:19 2022

************************************************************
Device Utilization for 2VP100ff1696
************************************************************
Resource                        Used    Avail   Utilization
------------------------------------------------------------
IOs                             68      1164      5.84%
Global Buffers                  1       16        6.25%
LUTs                            812     88192     0.92%
CLB Slices                      406     44096     0.92%
Dffs or Latches                 98      91684     0.11%
Block RAMs                      0       444       0.00%
Block Multipliers               0       444       0.00%
Block Multiplier Dffs           0       15984     0.00%
GT_CUSTOM                       0       20        0.00%
------------------------------------------------------------

*************************************************

Library: work    Cell: TOP    View: TOP_LEVEL

*************************************************

  Cell                    Library  References     Total Area

 BUFGP                    xcv2p     1 x
 FDR_1                    xcv2p    98 x       1     98 Dffs or Latches
 FullAdder32              work      1 x      59     59 gates
                                             49     49 LUTs
 GND                      xcv2p     3 x
 IBUF                     xcv2p    34 x
 LUT1                     xcv2p     6 x       1      6 LUTs
 LUT2                     xcv2p    62 x       1     62 LUTs
 LUT3                     xcv2p    40 x       1     40 LUTs
 LUT4                     xcv2p    24 x       1     24 LUTs
 MUXCY_L                  xcv2p    46 x       1     46 MUX CARRYs
 OBUF                     xcv2p    33 x
 VCC                      xcv2p     1 x
 XORCY                    xcv2p    48 x
 multi_8bit               work      1 x     158    158 LUTs
                                            183    183 gates
 multi_8bit_unfolded0     work      1 x     159    159 LUTs
                                            185    185 gates
 multi_8bit_unfolded1     work      1 x     160    160 LUTs
                                            185    185 gates
 multi_8bit_unfolded2     work      1 x     160    160 LUTs
                                            185    185 gates

 Number of ports :                           68
 Number of nets :                           519
 Number of instances :                      401
 Number of references to this view :          0

Total accumulated area :
 Number of Dffs or Latches :                 98
 Number of LUTs :                           812
 Number of MUX CARRYs :                      46
 Number of gates :                          930
 Number of accumulated instances :         1082
```

Figure 23: Device Utilization

Resources

1. GitHub: https://github.com/abishekat/DSD

Challenges

The red lines in the simulation output for the END_FLAG is due to flip-flops because the flip-flop functions at the clock event. The issue was later resolved.

Conclusion

We successfully implemented an Arithmetic unit to perform the operation **Z = ¼ [A*B] + 1**. The additional requirement of a 16-bit multiplier and multiplicand was also implemented. The pipelining of the system is done through flip-flops in three stages. The control path of the system is controlled using the following flags LD_POS, CLR_POS, LOAD_IN, FF0_LD_EN, FF0_CLR_EN, S0_EN_FF, S1_EN_FF and S2_EN_FF.

References

1. M. Mahapatro et al., "Design of Arithmetic Circuits Using Reversible Logic Gates and Power Dissipation Calculation," 2010 International Symposium on Electronic System Design, 2010, pp. 85-90, doi: 10.1109/ISED.2010.25.
2. Y. Duanmu, J. Yang, J. Li, X. Xue, M. Jing and X. Zeng, "A 16-bit Arithmetic Logic Unit Design by Using Gate Diffusion Input," *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, 2020, pp. 1-3, doi: 10.1109/ICSICT49897.2020.9278245.
3. COEN-6521-2221-CC *https://moodle.concordia.ca/*. [Online]
4. https://github.com/abishekat/DFT/blob/main/GROUP%2013%20REPORT.pdf