Bruno E. Gracia Villalobos

EE 4513 – Introduction to VLSI

Design and Implementation of Variable Width Array Multiplier and Synchronous, Up-Down Counter Integrated Circuits in Verilog and Cadence Encounter

October 26, 2019

The path for the author's designs in the Linux server is: /home/UTSARR/ohf614/lab8hw

INTRODUCTION

This report describes the design and implementation of two distinct integrated circuits: a 64x64 Array Multiplier and a 16-bit Synchronous Up-Down Counter. To better understand the underlying structures of both designs, they are developed at a structural level of abstraction—with basic logic gates as building blocks (AND, OR, INV, XOR, etc.). Further, to make the designs reusable in the future, they are both modular in the way that they can be instantiated by a wrapper with variable data widths: both Array Multiplier and Synchronous counter can be implemented from 4 bits up to 2^N bits with a simple change of parameter.

The report is divided into three sections: the 64x64 Array Multiplier Design, the 16-bit Synchronous Counter Design, and the Conclusion. Each section will describe the process for developing the final design at the integrated circuit level with Cadence Encounter.

64x64 ARRAY MULTIPLIER

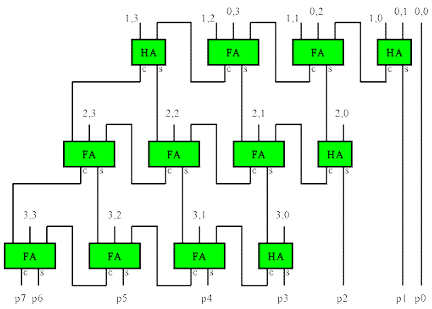


Figure 1: 4x4 Array Multiplier from massmind.org

The 64x64 Array Multiplier diagram is based off a 4x4 Array Multiplier architecture. The 4x4 Array Multiplier diagram referenced from an online forum is shown in Figure 1. Studying the design for patterns, an algorithm is developed to build a variable width solution using the Verilog HDL.

In a high level, an Array Multiplier is a structured datapath for the partial sums of an elementary school method for multiplication. For example, consider the following 4x4 binary multiplication.

				1	0	1	1
			X	1	1	0	1
				1	0	1	1
		+	0	1	0	0	
–	–							
		1	0	1	0	1	1
	+	1	0	1	1		
–	–							
		1	1	0	1	1	1
+	1	0	1	1			
1	0	0	0	1	1	1	1

In the table, the upper green row is the Multiplicand, and the lower green row is the Multiplier. Assuming little endianness, bits 0 and 1 are multiplied with the multiplicand to produce two 4-bit results, of which bit 1's multiplication is shifted left 1 bit due to its place value. Next, the two 4-bit results are added, and a partial sum is formed (shown in blue). Because bit 0 of the final 8-bit result is now "freed" from further arithmetic, this bit represents the final answer for that place value and can be "dropped down" as shown in the p0 result in Figure 1.

Now, bit 2 is multiplied with the multiplicand to produce a 4-bit result, and this is added with the first partial sum. The

result of this addition is the second partial sum, and now p1 is “freed” and can be dropped down. The 3rd bit is multiplied with the multiplicand, and the third and final partial sum is created. It is important to mention the final partial sum will have (N*2)-(N-1) bits that “drop” down from the adders. The intuition for the array multiplier should be clear by now, and it is evident that an NxN bit multiplication will require (N-1) partial sums to realize the product. Next, the Verilog HDL design is presented.

Structural Design in Verilog HDL

Using a structural approach allowed the author to keep in mind performance and resource efficiency, and although the target delivery format is an integrated circuit, the Verilog HDL is modularity of the design is kept in mind for an Artix-7 FPGA. From previous assignments, the half-adder (HA) and full-adder (FA) modules are instantiated depending on the number of *bits* specified by the parameter, and the Array Multiplier module is presented next.

```
1  `timescale 1ns / 1ps
2  module ARRAY_MULTIPLIER #(parameter bits = 128) (
3      input [bits-1:0] a, b,
4      output [bits*2-1:0] p
5  );
6
7  //rows means the partial sums
8  //for 4-bit adder:
9  //carries[1][0] is the carry out from the rightmost halfadder in the 1st partial sum
10
11 //DECLARING: [index] carries [rows]
12 //ACCESSING: carries[rows][index]
13
14 wire [bits-1:0]     carries [bits-1:1];
15 wire [bits*2-1:0]   sums [bits-1:1];
16
17 assign p[0] = a[0] & b[0]; //always true
```

Figure 2: Portlist and internal wires for ARRAY_MULTIPLIER

Figure 2 displays the portlist and internal wires for the ARRAY_MULTIPLIER module. Similar to the example in the table presented earlier, the inputs for this module are a multiplicand and a multiplier with N bits in width as specified by the parameter passed in. The output is a product of N*2 bits in width. Now, given the diagram in Figure 1, the wire busses presented in lines 14 and 15 of the HDL code represent the carry propagations between adders, as well as their outputs. Essentially, these busses are used to route the outputs from each adder in the Array Multiplier. The assign statement of line 17 is always true as long as *bits* parameter is set to > 2.

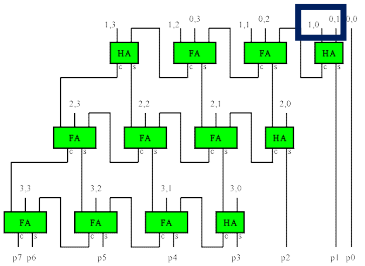


Figure 3: Revisited 4x4 Array Multiplier

Before explaining the algorithm for building an Array Multiplier with variable width, it is important to revisit the architecture displayed in Figure 1. The comma separated values (CSV) as the inputs of various adders denote a reduction AND (&) operation between the bits from each operand, with the bit to the right of the comma representing the bit number of the multiplier, and the bit to the left representing the bit number of the multiplicand; let’s call the multiplicand “A”, and the multiplier “B”. Consider again the multiplication in the table from the introduction section.

A	1	0	1	1
Placevalue	3	2	1	0
B	1	1	0	1
Placevalue	3	2	1	0

With the table legend in mind, the CSV values should make sense now. For example, shown highlighted, one of the inputs to the first half adder on the top right of Figure 1 denotes the operation A[0] & B[1]. The other input is the operation A[1] & B[0]. Now, a final statement about the architecture is given to better understand the Verilog logic

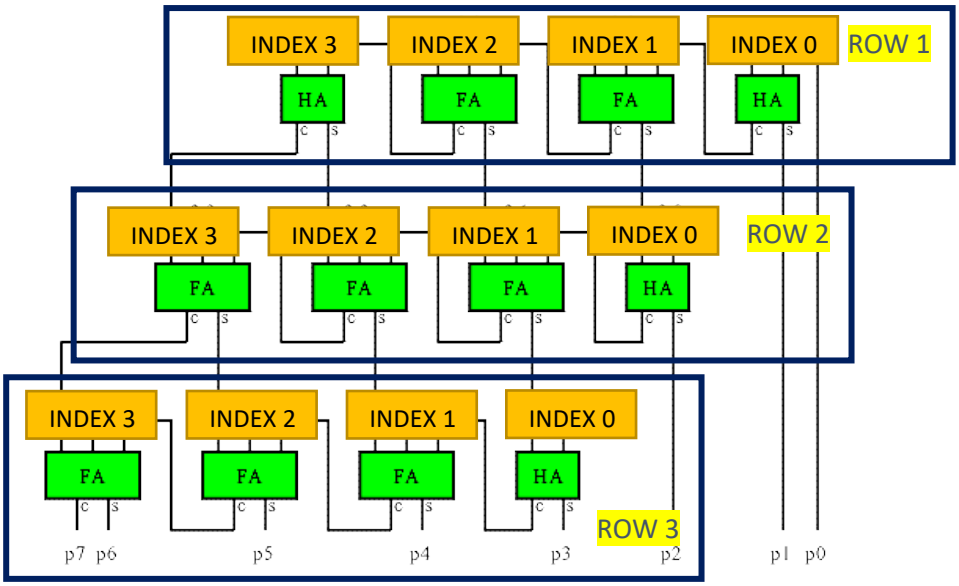


Figure 4: Rows and columns in the architecture

```
genvar i,j;
generate
    for(i=1; i < bits; i=i+1) begin : GEN_PARTIALS
        for(j=0; j < bits; j=j+1) begin: GEN_ADDERS
```

Figure 5: Generate logic for creating variable width Array Multiplier

The outer loop called GEN_PARTIALS describes the rows as shown in Figure 4 starting with row 1. The inner loop called GEN_ADDERS describes the columns of each row as shown in Figure 5, starting with index 0. This approach allows the algorithm to cycle through each partial sum, instantiating the necessary HA or FA to build the Array Multiplier.

However, a problem still arises on the type of adder needed. For example, the first row has two half adders and a full adder, and the subsequent rows have only one half adder and three full adders. For this reason, the algorithm is divided into three logic cases: the first row, the last row, and all rows in between. For an NxN Array Multiplier, (N-3) partial sum rows are present in between the first and last rows. Although the Array Multiplier is realized as a combinational

design, the generate statement is inherently algorithmic, and a basic flow chart in Figure 6 is presented for a single iteration of the outer loop

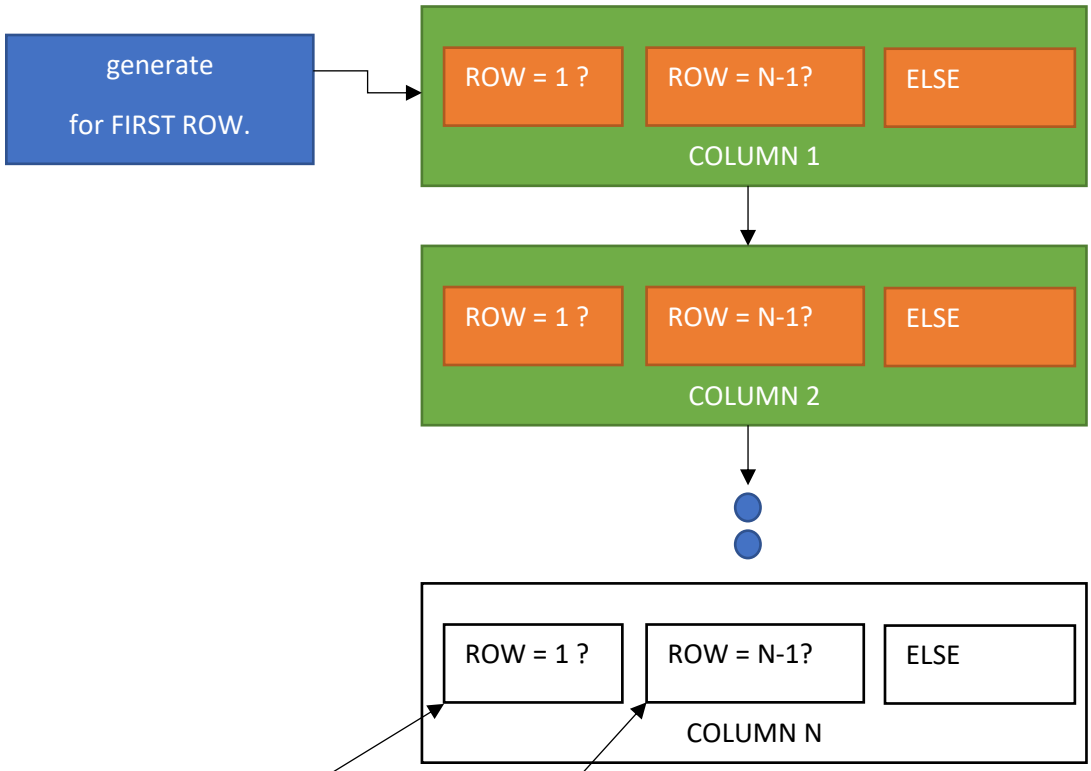


Figure 6: Basic flow chart for generate logic

```
24              //FIRST LEVEL OF PARTIAL SUMS
25              if(i==1) begin
26
27                  //FIRST ADDER
28                  if(j==0) begin
29                      ADD_HALF INSTF_HA_FIRST (
30                          .a(a[i]&b[j]),
31                          .b(a[i-1]&b[i]),
32
33                          .sum(p[i]),
34                          .cout(carries[i][j])
35                      );
62              //LAST LEVEL OF PARTIAL SUMS
63              else if(i==(bits-1)) begin
64
65                  //FIRST ADDER
66                  if(j==0) begin
67                      ADD_HALF INSTL_HA_FIRST (
68                          .a(sums[i-1][j+1]),
69                          .b(a[i]&b[j]),
70
71                          .sum(p[i+j]),
72                          .cout(carries[i][j])
73                      );
74                  end
75
```

Figure 7: First and second levels of partial sum generation

The row boxes are iterations of the inner for loop, and the boxes inside the columns are the case statements. An example for the first row case statement is now presented next. As shown in Figure 7, the arrows signal to the logic block present in Figure 6. Essentially, for each column of each partial row, three conditions are checked to generate the appropriate adder.

Simulation and Results

Now the results for the Array Multiplier are presented. For this assignment, a 64x64 is instantiated as the Unit-Under-Test (UUT) in the testbench using a parameter *bits* = 64. Figure 8 displays the simulation results for an arbitrary multiplication.

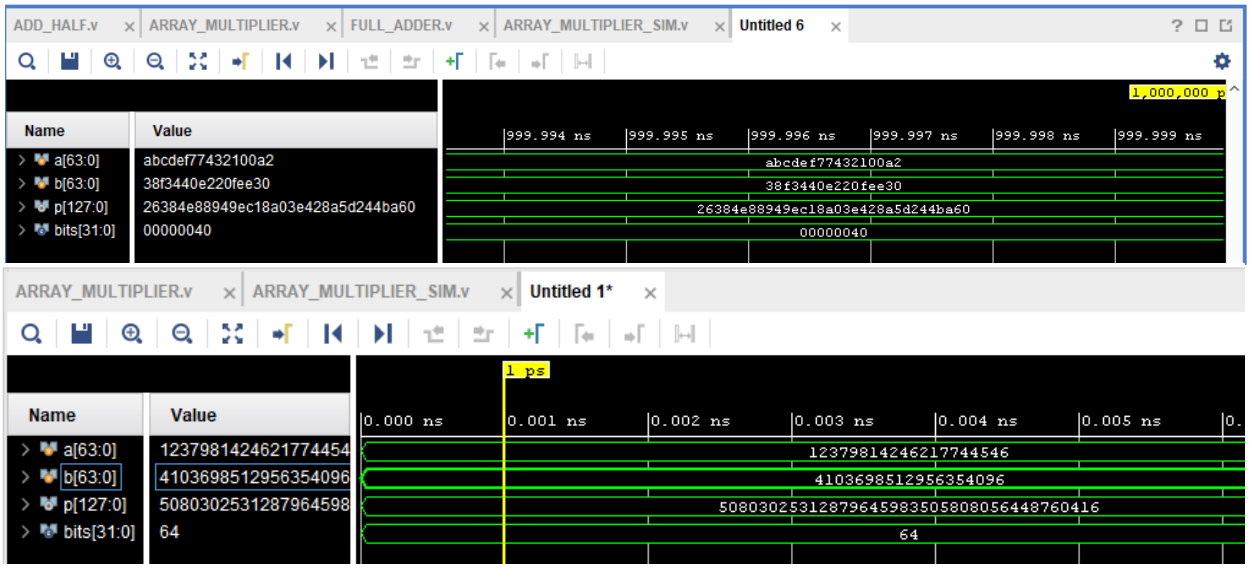


Figure 9: 64x64 multiplication using the ARRAY_MULTIPLIER Design (TOP: HEX). (BOTTOM: DECIMAL)

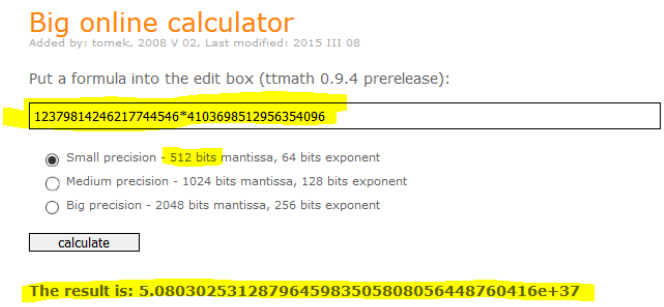


Figure 8: Verified 64x64 bit multiplication

Given 64x64 bit multiplication is too large to be carried out by the author's 64-bit system, the product is verified online as shown in Figure 9. Next, another example is given. The multiplicand is shifted left by 1 bit with a multiplier of 2 as shown in Figure 10.

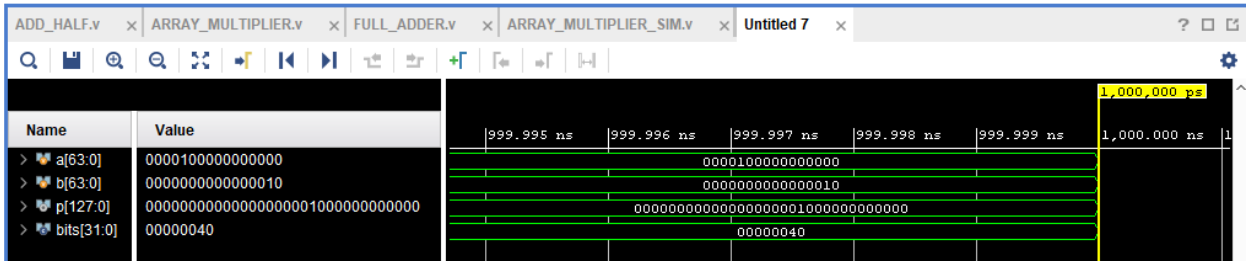


Figure 10: Left shift 1 bit simulation

Now that the design is verified for correct operation, the elaborated design as generated by the Xilinx Vivado tool is presented in the next page. The schematic is surely complex, and the

reader should be aware this is due to the structural approach taken for the design given an integrated circuit will be built next.

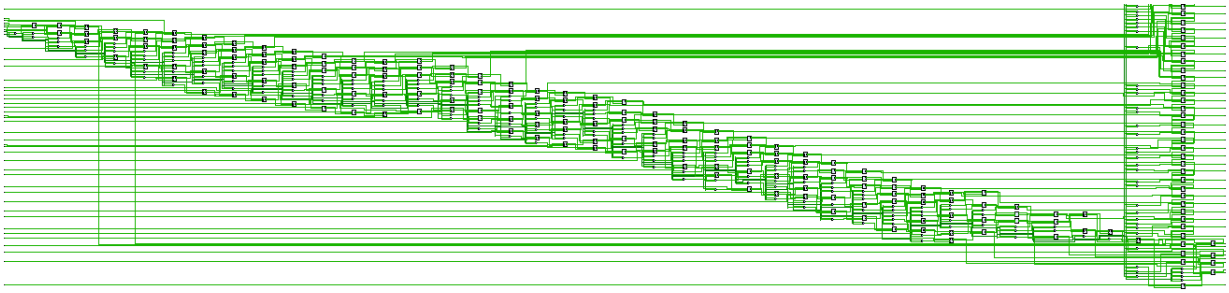
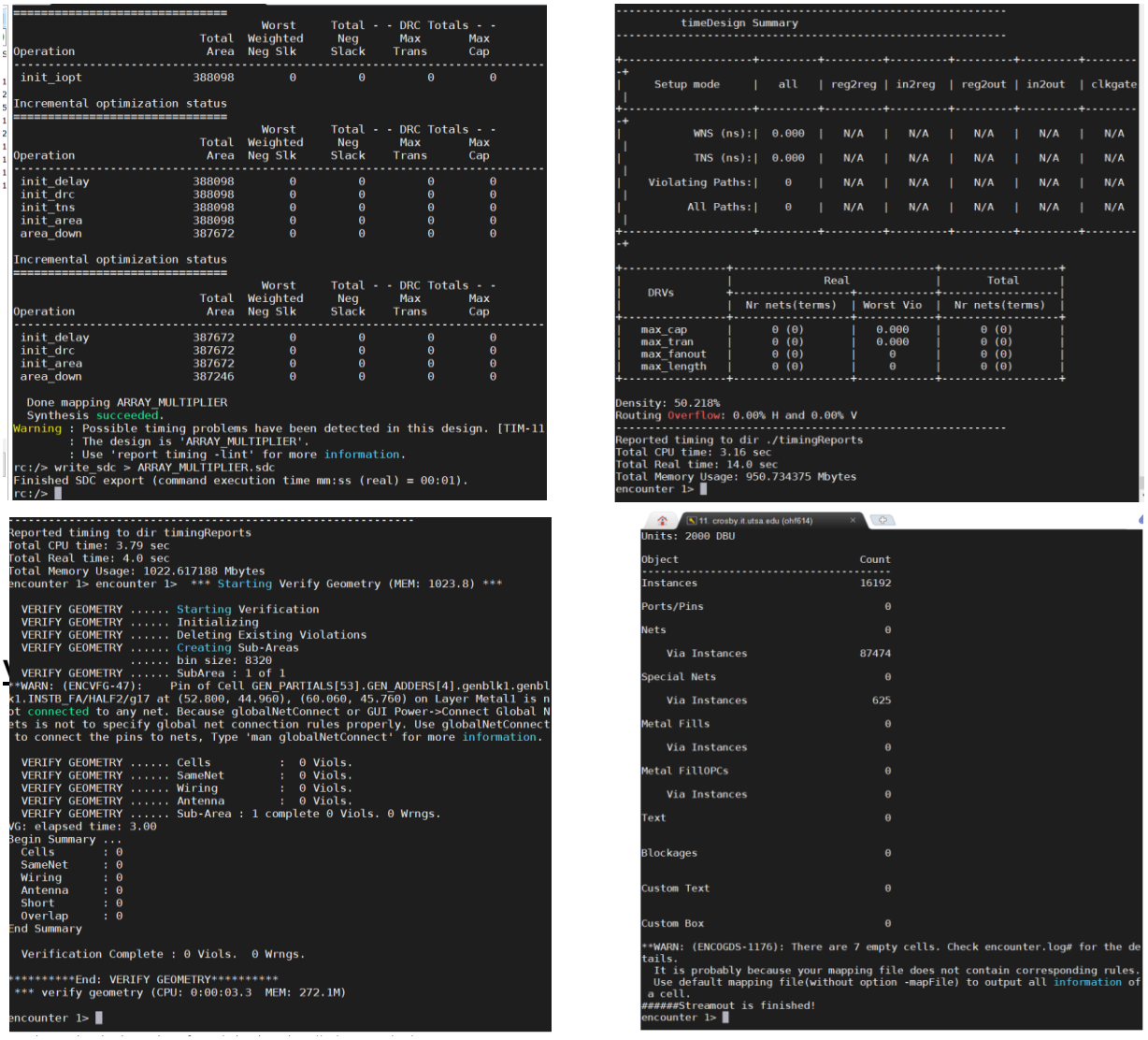


Figure 11: 64x64 bit multiplier elaborated design

Integrated Circuit Design with Cadence Encounter

The next step for the 64x64 Array Multiplier is the design of the integrated circuit in Cadence Encounter. The following figures are better left without written observations and are presented as such next.



```
==============================
                                Worst     Total - - DRC Totals - -
                          Total Weighted    Neg      Max       Max
Operation                  Area Neg Slk    Slack    Trans      Cap
--------------------------------------------------------------------------------
 init_iopt               388098        0        0        0        0

Incremental optimization status
==============================
                                Worst     Total - - DRC Totals - -
                          Total Weighted    Neg      Max       Max
Operation                  Area Neg Slk    Slack    Trans      Cap
--------------------------------------------------------------------------------
 init_delay              388098        0        0        0        0
 init_drc                388098        0        0        0        0
 init_tns                388098        0        0        0        0
 init_area               388098        0        0        0        0
 area_down               387672        0        0        0        0

Incremental optimization status
==============================
                                Worst     Total - - DRC Totals - -
                          Total Weighted    Neg      Max       Max
Operation                  Area Neg Slk    Slack    Trans      Cap
--------------------------------------------------------------------------------
 init_delay              387672        0        0        0        0
 init_drc                387672        0        0        0        0
 init_area               387672        0        0        0        0
 area_down               387246        0        0        0        0

  Done mapping ARRAY_MULTIPLIER
  Synthesis succeeded.
Warning : Possible timing problems have been detected in this design. [TIM-11
        : The design is 'ARRAY_MULTIPLIER'.
        : Use 'report timing -lint' for more information.
rc:/> write_sdc > ARRAY_MULTIPLIER.sdc
Finished SDC export (command execution time mm:ss (real) = 00:01).
rc:/>
```

```
------------------------------------------------------------
         timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+---------+--------
-+
|    Setup mode      |   all   | reg2reg | in2reg  | reg2out | in2out  | clkgate
|
+--------------------+---------+---------+---------+---------+---------+--------
-+
|           WNS (ns):|  0.000  |   N/A   |   N/A   |   N/A   |   N/A   |   N/A
|
|           TNS (ns):|  0.000  |   N/A   |   N/A   |   N/A   |   N/A   |   N/A
|
|    Violating Paths:|    0    |   N/A   |   N/A   |   N/A   |   N/A   |   N/A
|
|          All Paths:|    0    |   N/A   |   N/A   |   N/A   |   N/A   |   N/A
|
+--------------------+---------+---------+---------+---------+---------+--------
-+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|     DRVs       +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 50.218%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------
Reported timing to dir ./timingReports
Total CPU time: 3.16 sec
Total Real time: 14.0 sec
Total Memory Usage: 950.734375 Mbytes
encounter 1>
```

```
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 3.79 sec
Total Real time: 4.0 sec
Total Memory Usage: 1022.617188 Mbytes
encounter 1> encounter 1>  *** Starting Verify Geometry (MEM: 1023.8) ***

  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
  VERIFY GEOMETRY ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (ENCVFG-47):    Pin of Cell GEN_PARTIALS[53].GEN_ADDERS[4].genblk1.genbl
k1.INSTB_FA/HALF2/g17 at (52.800, 44.960), (60.060, 45.760) on Layer Metal1 is n
ot connected to any net. Because globalNetConnect or GUI Power->Connect Global N
ets is not to specify global net connection rules properly. Use globalNetConnect
 to connect the pins to nets, Type 'man globalNetConnect' for more information.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
  VERIFY GEOMETRY ..... Sub-Area : 1 complete 0 Viols. 0 Wrngs.
VG: elapsed time: 3.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:03.3  MEM: 272.1M)

encounter 1>
```

```
11: crosby.it.utsa.edu (ohf614)
Units: 2000 DBU

Object                          Count
----------------------------------------
Instances                       16192

Ports/Pins                          0

Nets                                0

    Via Instances               87474

Special Nets                        0

    Via Instances                 625

Metal Fills                         0

    Via Instances                   0

Metal FillOPCs                      0

    Via Instances                   0

Text                                0


Blockages                           0


Custom Text                         0


Custom Box                          0

**WARN: (ENCOGDS-1176): There are 7 empty cells. Check encounter.log# for the de
tails.
  It is probably because your mapping file does not contain corresponding rules.
  Use default mapping file(without option -mapFile) to output all information of
 a cell.
######Streamout is finished!
encounter 1>
```

Figure 12: Reports from Cadence Encounter

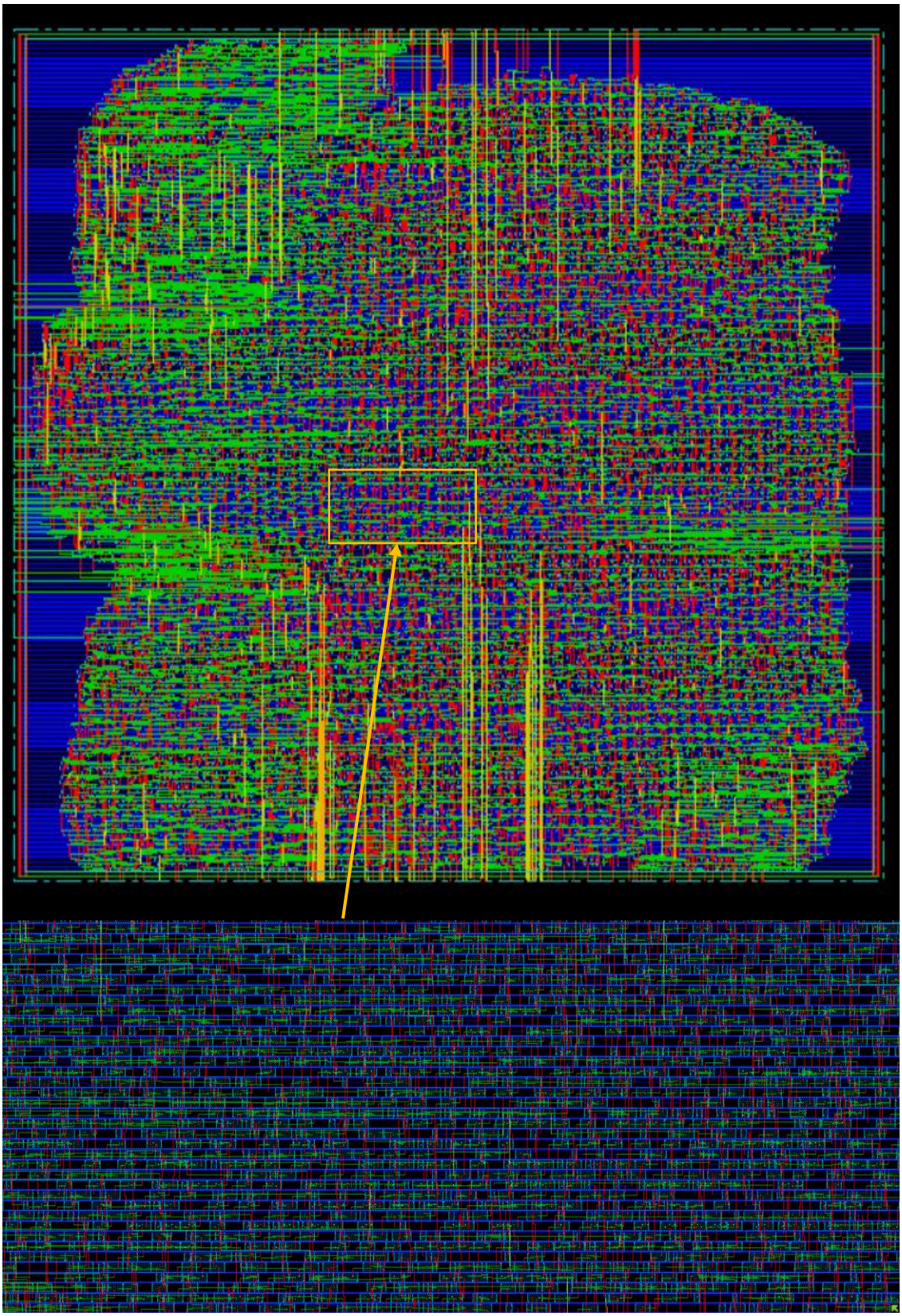


Figure 13: (TOP) Final CMOS Layout. (BOTTOM) Zoomed in section

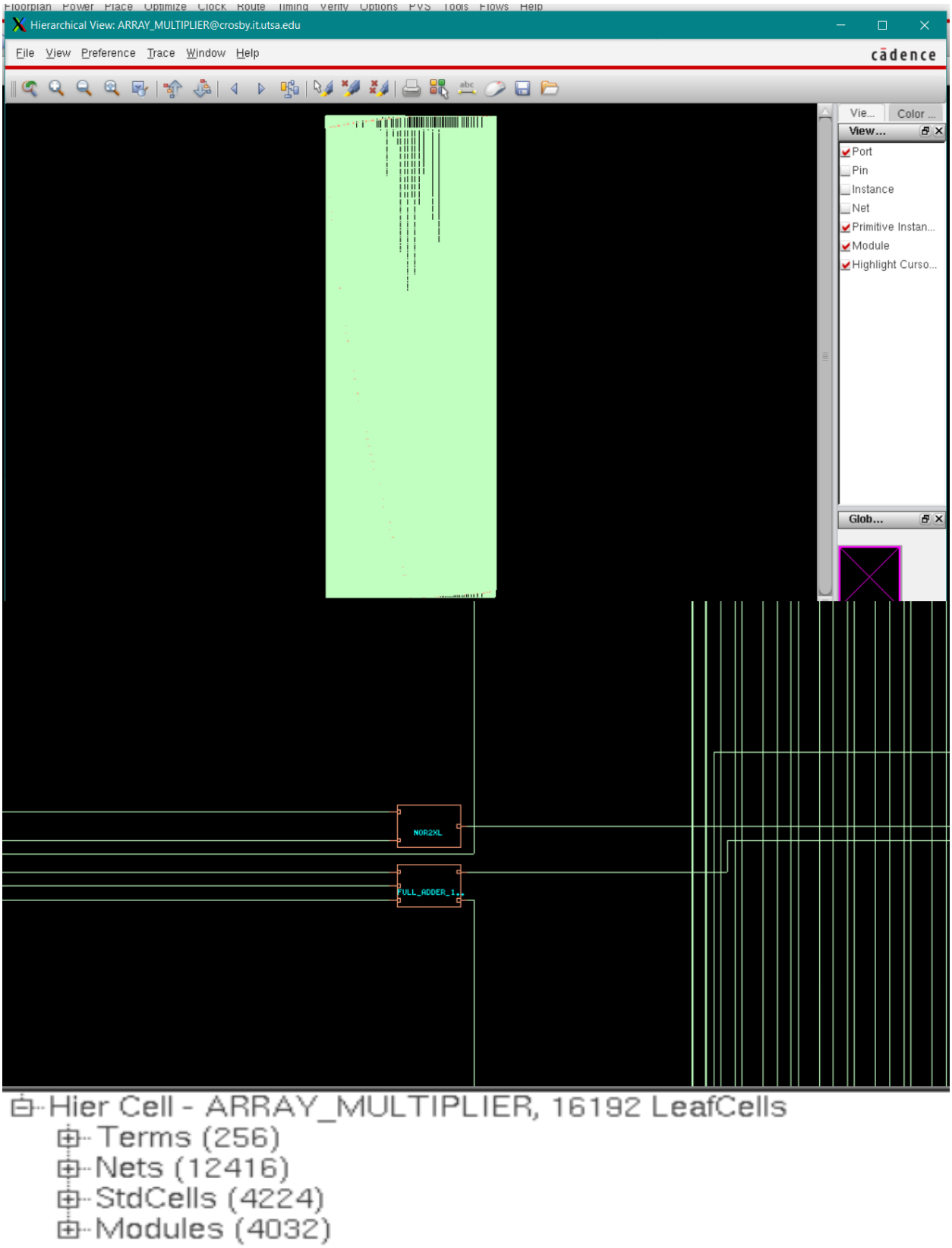


Figure 14: Schematic views of the 64x64 Array Multiplier and Cell Count

16-BIT SYNCHRONOUS UP-DOWN COUNTER

Architecture Design

The 16-bit synchronous counter is also designed with a variable width approach for better repurposing of the logic. This approach is also a great way to implement in an FPGA, but the CMOS layout for the integrated circuit is presented in Cadence Encounter. First, however, the architecture is referenced from an online source and is presented below in Figure 15.

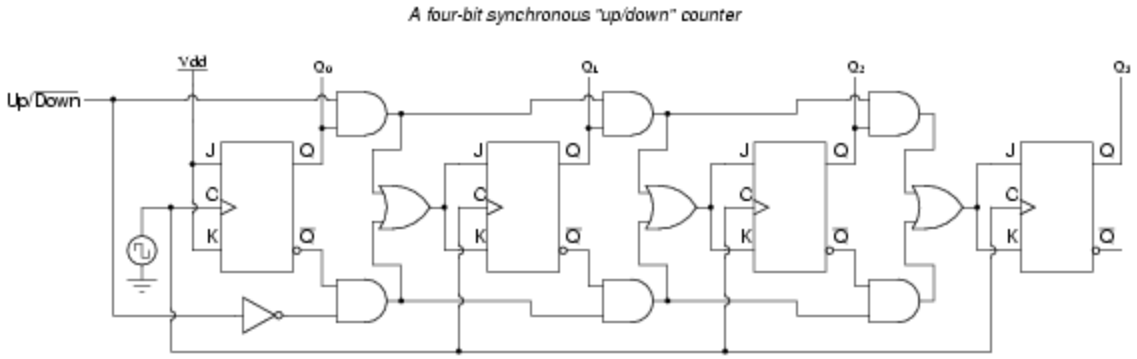


Figure 15: 4-bit synchronous up-down architecture from vlsiencyclopedia.com

In this design, an overall structural approach is also used, and the first step is to design a JK flip-flop module. Although the author is aware T flip-flops (TFF) could be used since the JK inputs for all FF's are shorted together, the JKFF is designed as a learning exercise. Figure 16 demonstrates the JKFF module with asynchronous reset designed with a behavioral approach.

```
13  always@(posedge clk, posedge clr) begin
14      if(clr) begin
15          q = 1'b0;
16      end
17      else begin
18          case({j,k})
19              2'b00: q = q;
20              2'b01: q = 1'b0;
21              2'b10: q = 1'b1;
22              2'b11: q = ~q;
23          endcase
24      end
25  end
```

Figure 16: JKFF Module in behavioral approach

Now, similar to the Array Multiplier design, the synchronous counter is developed to take in a parameter to dictate how many bits to generate. A generate statement is therefore used to create the logic necessary to define the N bit counter. The design is structured into columns and a for loop is used to cycle through from the Least significant bit (LSB) to the Most significant bit (MSB) of the counter. Like the multiplier design, there are also three cases to check in each iteration: the first section, the second section, and the remaining sections. The architecture is presented again to better formulate the algorithm to create the variable width logic in Figure 17.

For an N-bit synchronous counter, there will be N-2 remaining sections, and the other two will be the first and second sections respectively. The argument behind this choice lies in the inputs to the AND gates of the yellow section shown in Figure 17—they are connected to the control lines, while the “remaining sections” are connected to i-1 sections’ and gates. Figure 18 shows a snippet of the logic used to make the counter variable width.

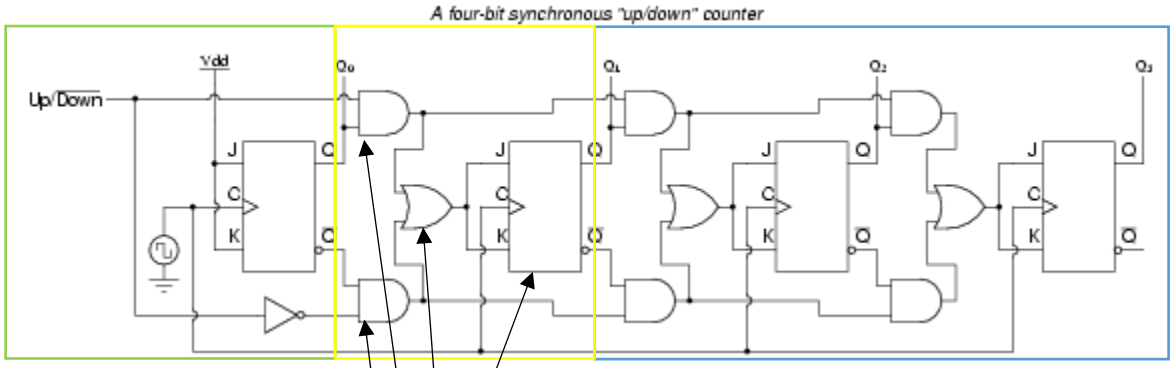


Figure 17: Loop iterations to generate variable width counter (GREEN = First section) (Yellow = Second section) (Blue = Remaining sections)

```
41          else if(i==1) begin
42              JKFF INST_JK (
43                  .clk(clk),
44                  .j(jk[i]),
45                  .k(jk[i]),
46                  .clr(rst),
47
48                  .q(q[i])
49              );
50
51              and(awire[i][1], ctrl, q[i-1]);
52              and(awire[i][0], ~ctrl, ~q[i-1]);
53              or(jk[i], awire[i][1], awire[i][0]);
54          end
```

Figure 18: Second section generation of logic

The loop begins with index=0 and this marks the first section outlined in green in Figure 17. Once the second section begins, the inputs to the AND and OR gates are the previous sections’ outputs and are marked by i-1 indexes as shown in Figure 18. The simulation results are presented next.

Simulation and Results

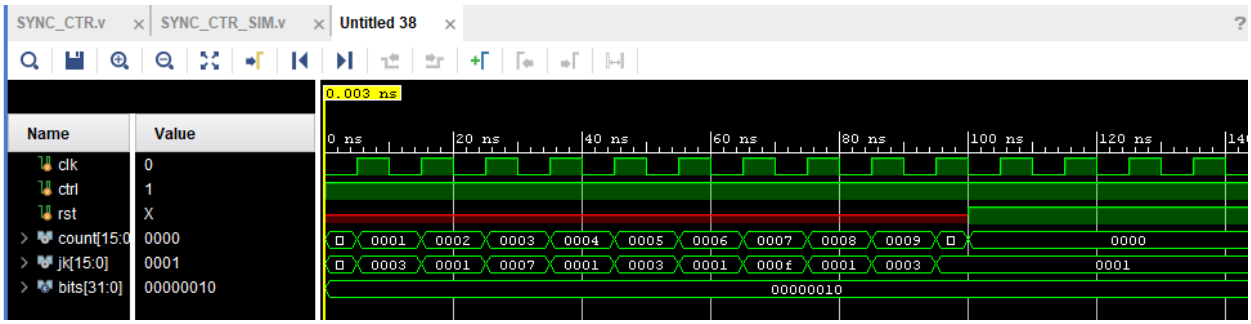


Figure 19: Reset after 100NS, Counting to 10

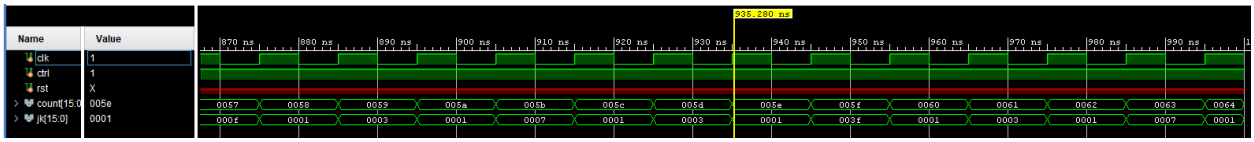


Figure 20; Counting up to 100d=0x64 in 1us

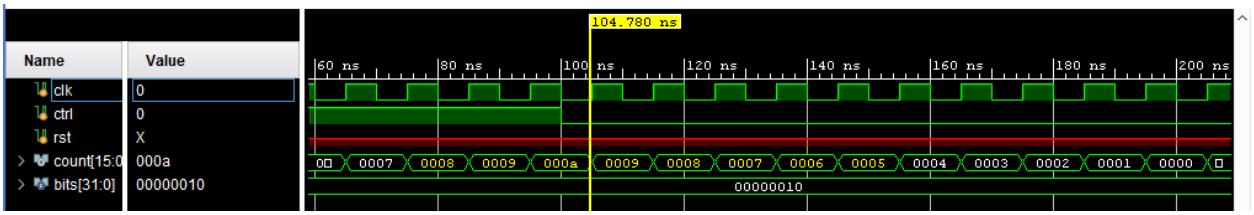
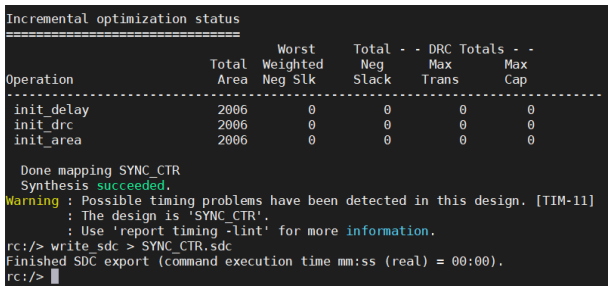
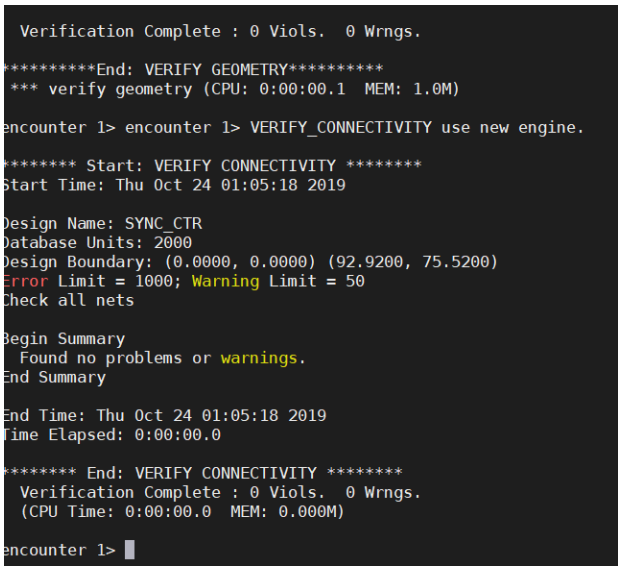
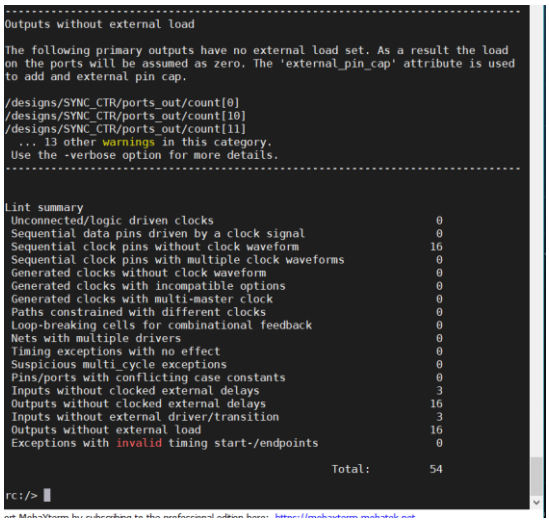
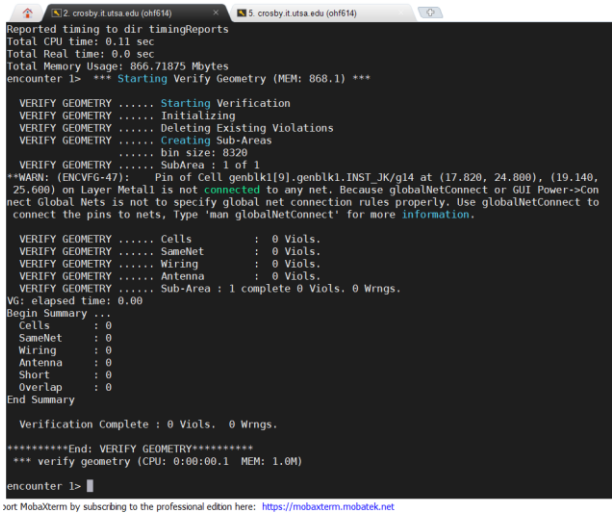


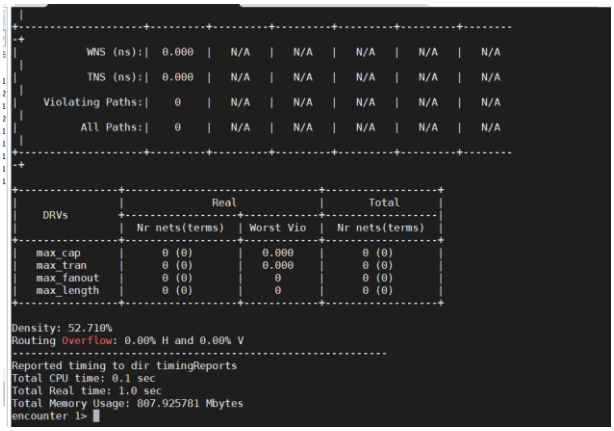
Figure 21: Counting up to 0xA then down again











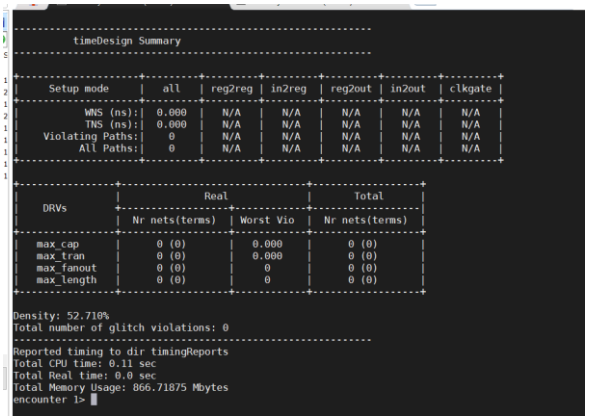


Figure 22: Various Design reports for the 16-bit Synchronous Counter

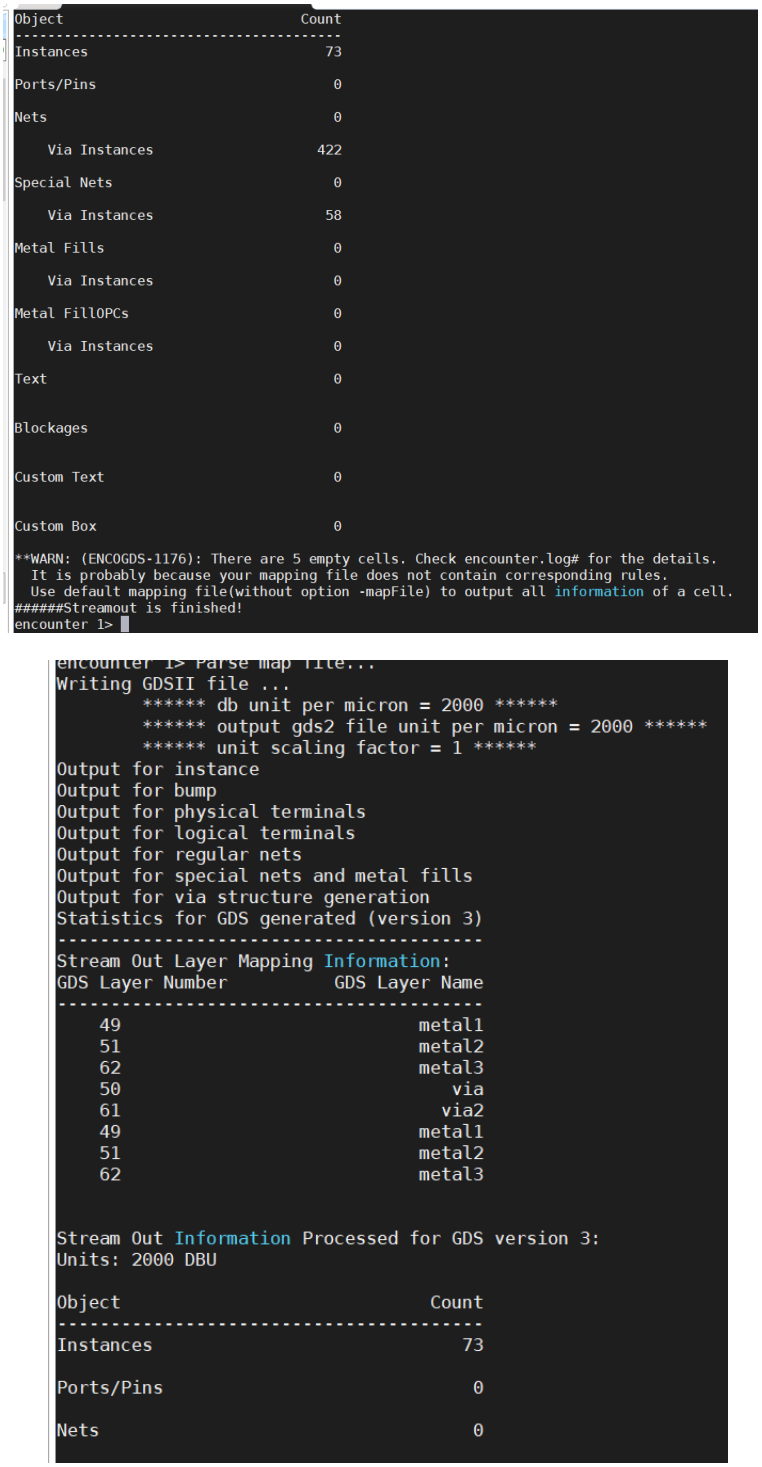


Figure 23: GDS Report

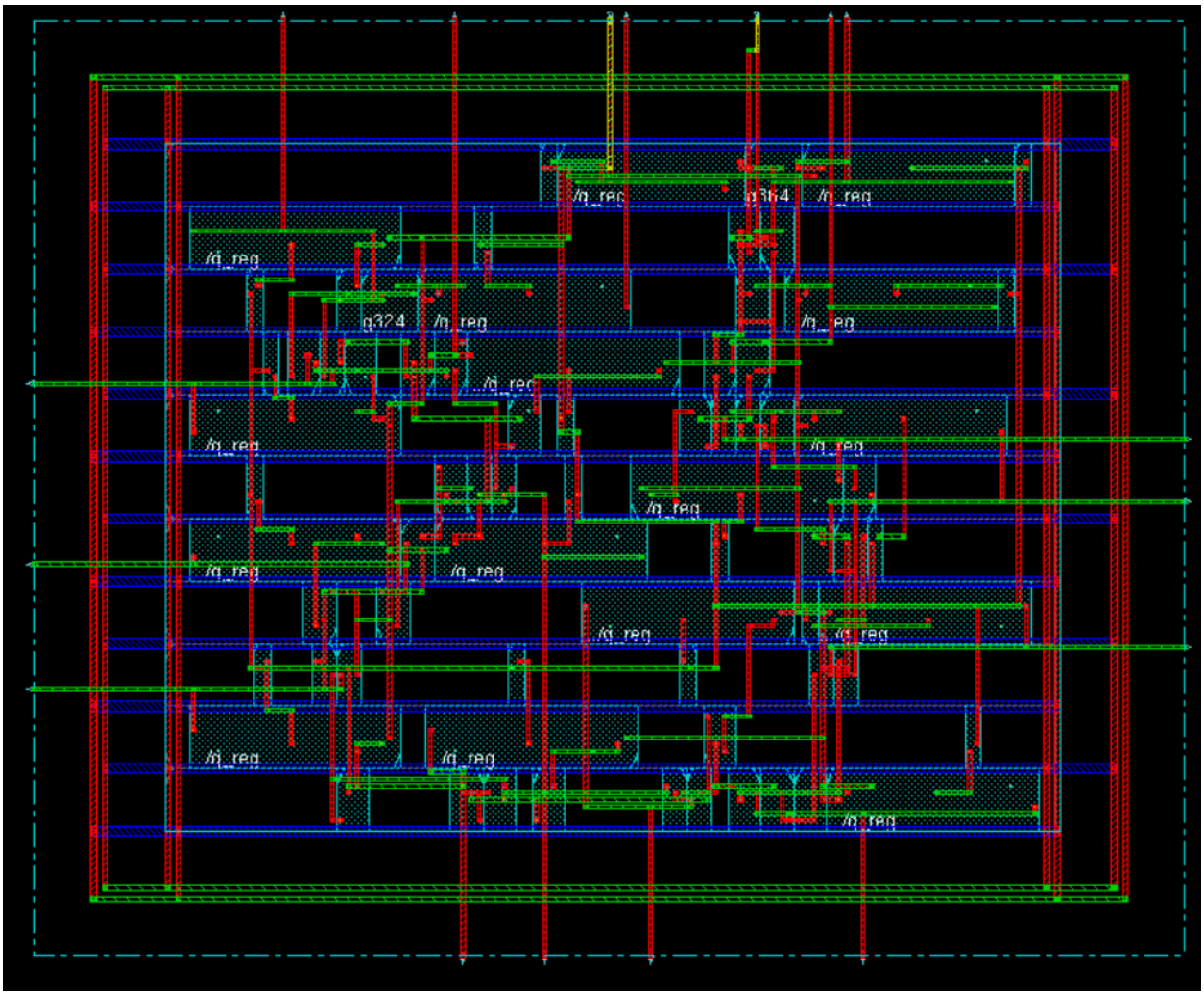


Figure 24: Final IC Layout

Conclusion

Designing both the multiplier and counter proved a great exercise to strengthen my structural design skills as well as IC design skills, specially the Array Multiplier. For the next assignments, I would like to try both structural and behavioral approaches to better compare the synthesis of the Verilog code by the Cadence RTL Compiler.