ECE 612

Design of Digital Integrated Circuits

AMR AHMED EL-HOSINY
MOHAMED MAHER MAHMOUD

Communication and Electronics Department

Dr. Sameh A. Ibrahim

6/27/2014

Table of Contents

Introduction

Multiplier Design Project: It is required to design a functional, compact, fast, energy-efficient 32x32 radix-4 modified-Booth-encoded Wallace-tree multiplier as shown in figure (1).

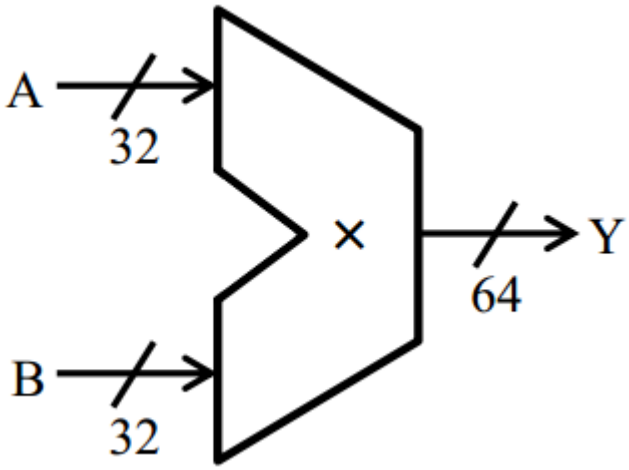

Fig.1: 32x32 multiplier symbol

It is required to design this multiplier and get results for functionality behavior, area report, power report and timing reports for critical path delay. Also, it is required to compare these results with the default multiplier produced by synthesis tool.

Multiplier Design

Design Steps:

1- ***Booth Encoding***: For radix-4 multiplication, we should use Booth Encoding to get partial products of multiplication. In radix-4 we multiply every 2 bits from multiplier with multiplicand to get only one partial product. So, we will use the following tables to generate partial products.

x_{2i+1}	x_{2i}	$pp_i(Booth)$	$pp_i(Modified\ Booth)$	Equivalent
0	0	0	0	0
0	1	Y	Y	Y
1	0	2Y	4Y-2Y	-2Y &inc. next PP
1	1	3Y	4Y-Y	-Y &inc. next PP

Table 1

Inputs			Partial Products	Booth Select		
x_{2i+1}	x_{2i}	x_{2i-1}	pp_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	2Y	0	1	0
1	0	0	-2Y	0	1	1
1	0	1	-Y	1	0	1
1	1	0	-Y	1	0	1
1	1	1	-0	0	0	1

Table 2

And here for 32x32 multiplier we will get 17 partial products which will entre next step.

2- ***Wallace-tree Multiplier***: In this step we will try to rearrange partial products before entering (3:2 compressor). The following figures illustrate this arrangement step.

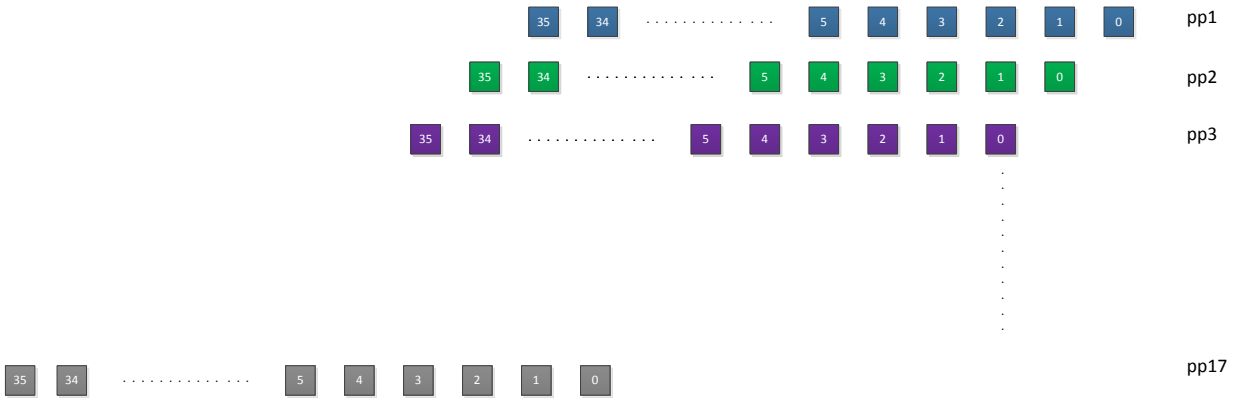

Fig.2: Generated partial products from Booth Encoding

The rearrangement process will focus on moving MSB bits of partial products from down toward up in the empty places of the above partial product. After finishing this step for all partial products we will get the partial products as in following figure.

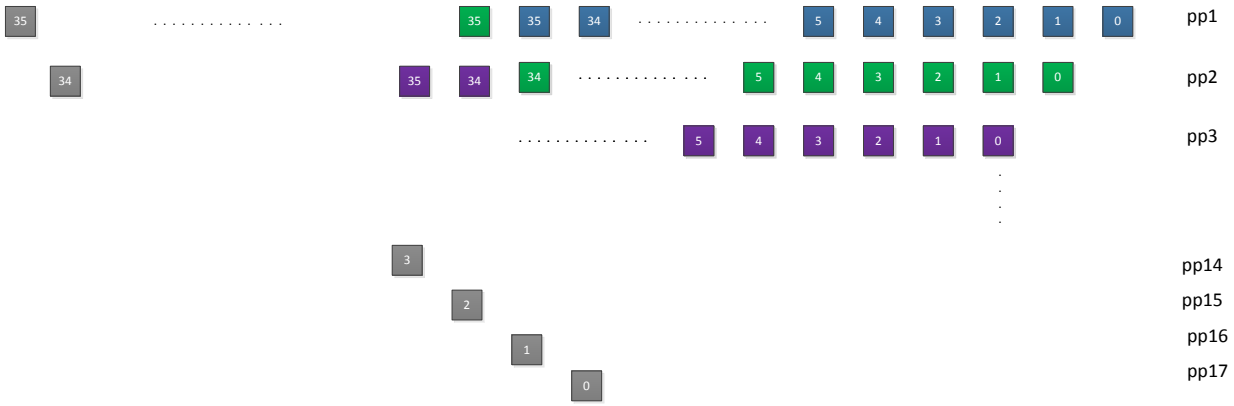

Fig.3: Partial products after rearrangement step

3- ***3:2 Compressor***: This is reduction step for partial products. We generate only 2 vectors instead of 3 vectors of partial products as shown in figure (4).

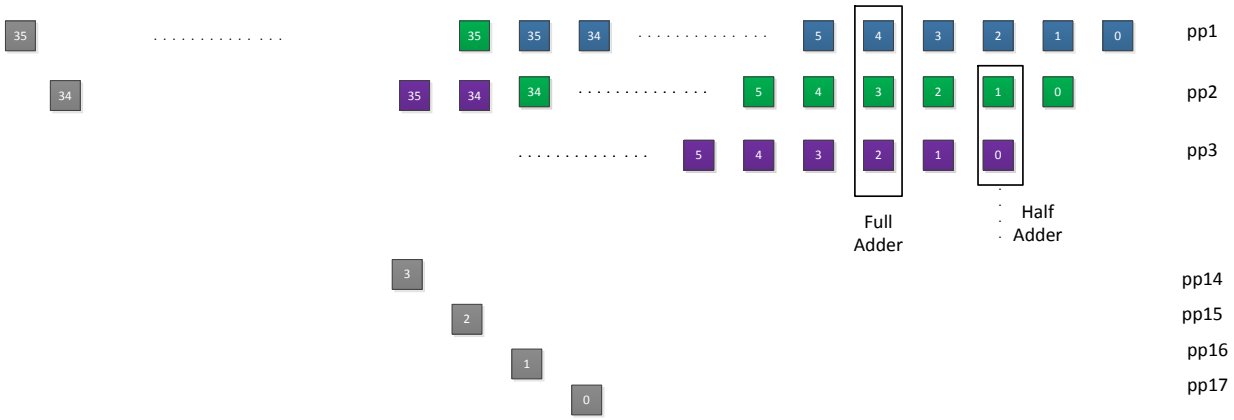

Fig.4: 3:2 Compressor for partial products

This compression process will be done till we get only 2 vectors each one of them consists of 64 bits to be ready for final step as shown in figure (5).

Fig.5: Final 2 vectors from 3:2 compressor step

4- ***Adder:*** This is the final step to get the output from multiplier. In this step we will make addition for the 2 final vectors from 3:2 compressor step. For this addition we will use kind of carry-lookahead tree adders which is Kogge-Stone Tree Adder. The advantage of Kogge-Stone Adder is fast Computation time but at cost of increased area. Carry-lookahead tree adders algorithm depends on generating carry tree at first then we can get final sum and carry. Generation of carry tree starts with preprocessing for input vectors to generate (p_i, g_i) from input vectors.

Where

$$p_i = A_i \; xor \; B_i$$

$$g_i = A_i \; and \; B_i$$

Then generate Carry look ahead network (P_{ij}, G_{ij}) from (P_i, G_i) and (P_j, G_j).

Where

$$P_{i:j} = P_{i:k+1} \; and \; P_{k:j}$$

$$G_{i:j} = G_{i:k+1} \; or \; (P_{i:k+1} \; and \; G_{k:j})$$

And finally to get sum and carry

$$sum_i = p_i \; xor \; carry_{i-1}$$

$$C_i = G_{i:0} \; or \; (c_{in} \; and \; P_{i:0})$$

The following figure shows the generation of carry tree and steps of adder.

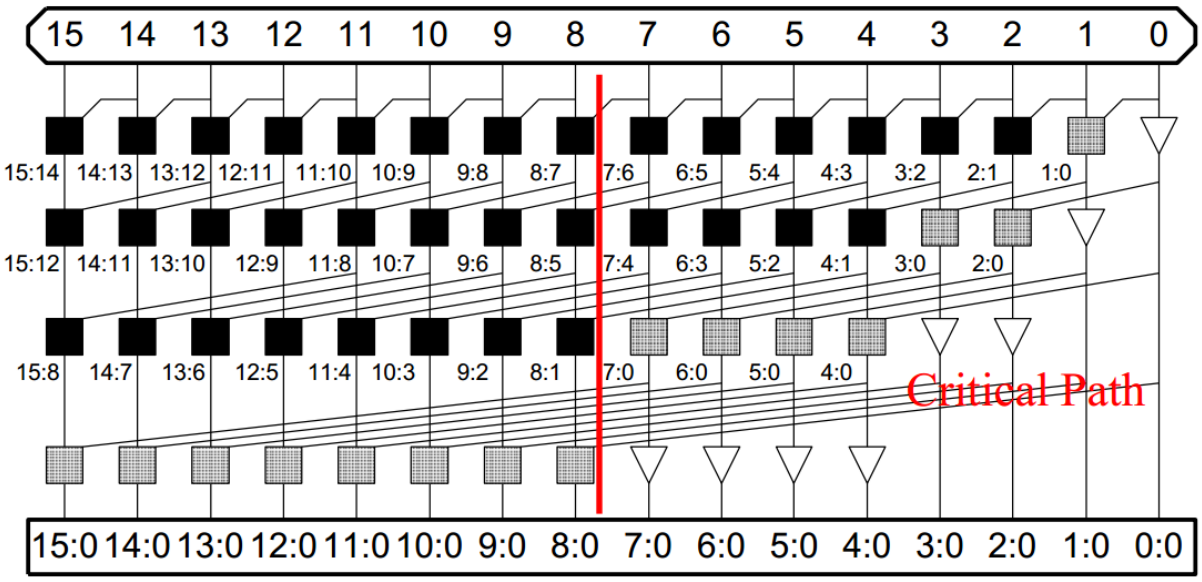

Fig.6: Kogge-Stone Tree Adder

Simulation Results

We have used a MATLAB script (m-file) and a C-program to generate random numbers of 32-bit width as input to the multiplier. The implemented test bench applies the random test cases to the multiplier; in parallel it calculates the multiplication results using behavioral multiplier statement. The test bench has a self-check flag "check_acc" which checks the multiplier output versus reference result calculated internally in the test bench. In case of any error, the "check_acc" is asserted '1' and remain '1' until the end of the simulation. Alternatively, the test bench writes the multiplier result to output file, in addition it calculates the multiplication result using behavioral multiplication statement. The result of the behavioral multiplication is written to file as a reference.

To verify the multiplier compare the two files:

- sim/multiplier_top_tb/results_out.txt
- sim/multiplier_top_tb/results_ref.txt

Implementation Design

The following figure shows the block diagram for architecture of digital blocks used in multiplier.

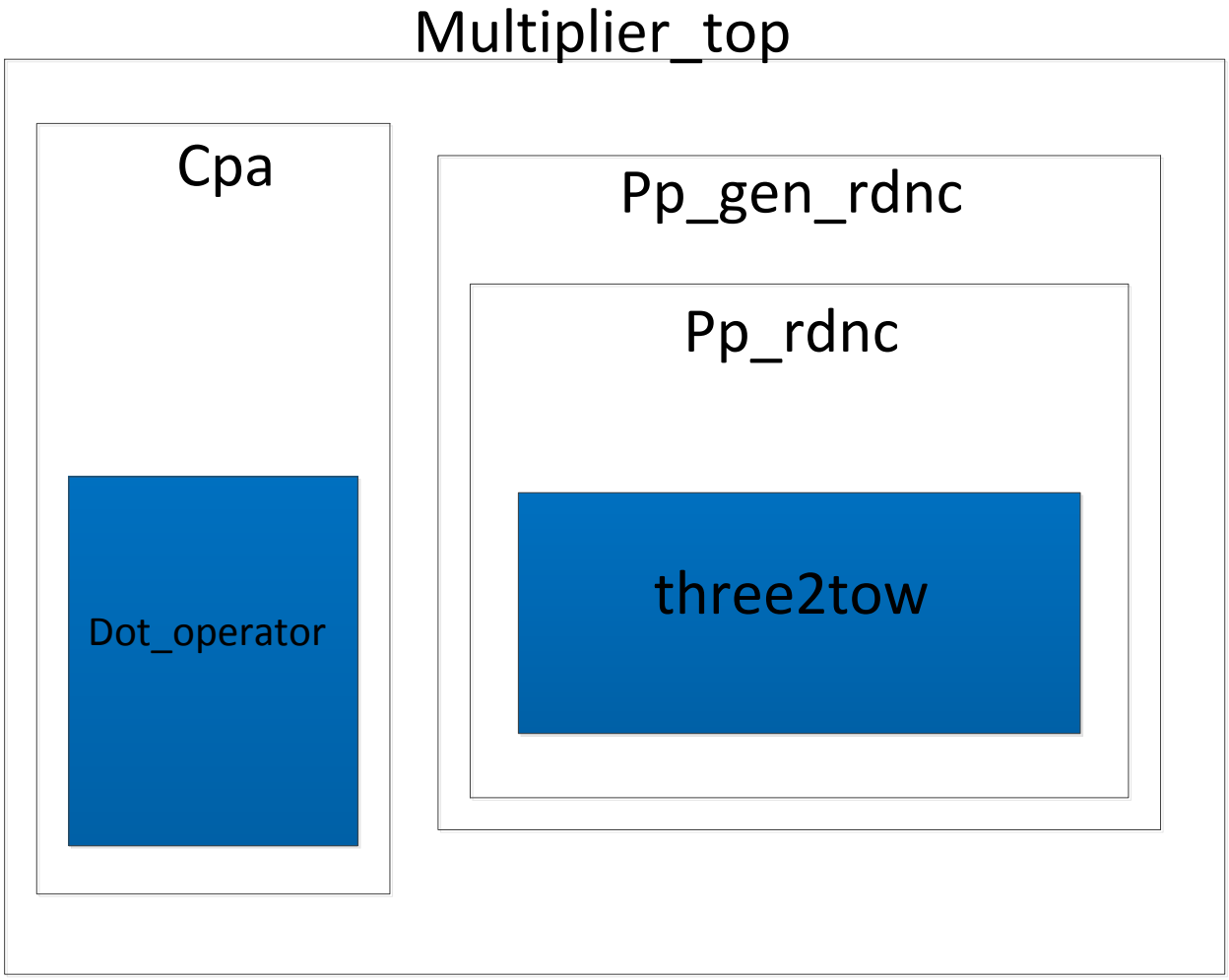

Fig.7: Block Diagram of design architecture

As shown in previous figure multiplier design is divided into two parts, one for generation of partial products and the other one for addition of remain two vectors using Kogge-Stone Adder. The first part of partial products generation includes partial products reduction using Wallace-tree the 3:2 compressor.

Simulation Command

The environment is set to run on Linux terminal as follows.

1. CD to scripts/sim
2. run this command => ***vsim -do sim.tcl***
3. Check the "check_acc" flag at the end of simulation or compare the results file from simulation output with the reference file.

4. To run simulation for the CPA, use the command => ***vsim –gtop=cpa_tb -do sim.tcl***

If it is required to run simulation on Windows follow following steps:

1. You have to change path to test cases files included in the test bench. Make it a relative of absolute path instead of the used environment variable
2. Create Modelsim project, include files listed inside source/tb/vhdl_src.f
3. Check the "check_acc" flag at the end of simulation or compare the results file from simulation output with the reference file.

And for running Xilinx Synthesis:

1. CD to synth/ise
2. use command => sh run.sh <top level>
3. ex : sh run.sh multiplier_top
4. Time constraint is set in the file multiplier.ucf

Functional Results

The following figures show the output from Modelsim after simulation for whole design. The first one shows that the output results from Modelsim are the same as the reference generated file. The second one also shows that the output results from Modelsim are the same as the reference generated file by checking at the end of simulation, check_acc is still zero indicating that all test cases have passed.

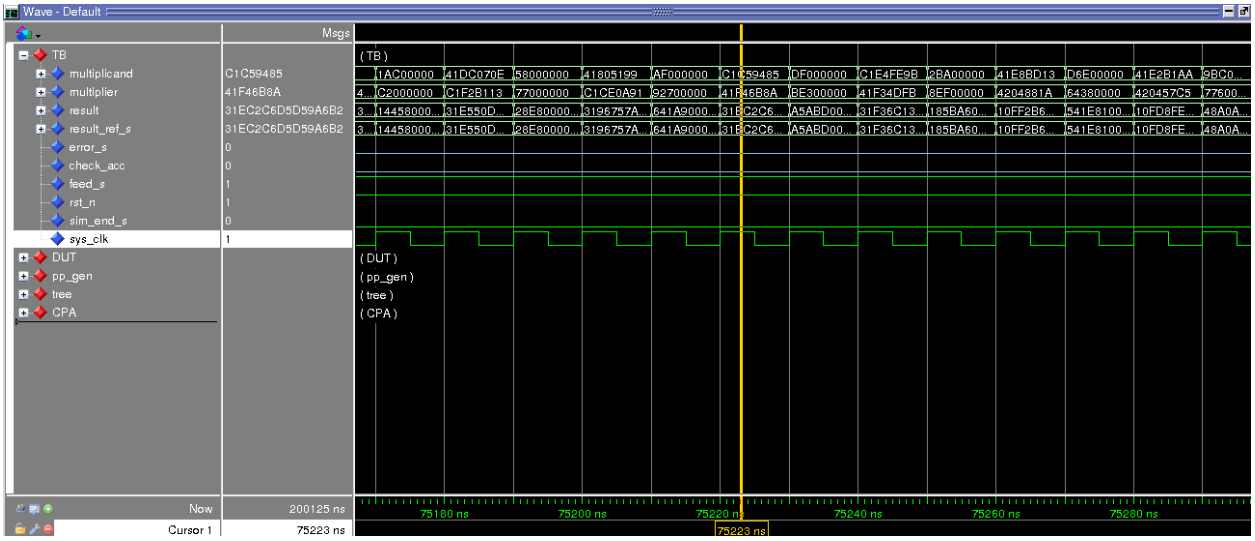

Fig.8: Simulation Result for Design (result = result_ref_s)

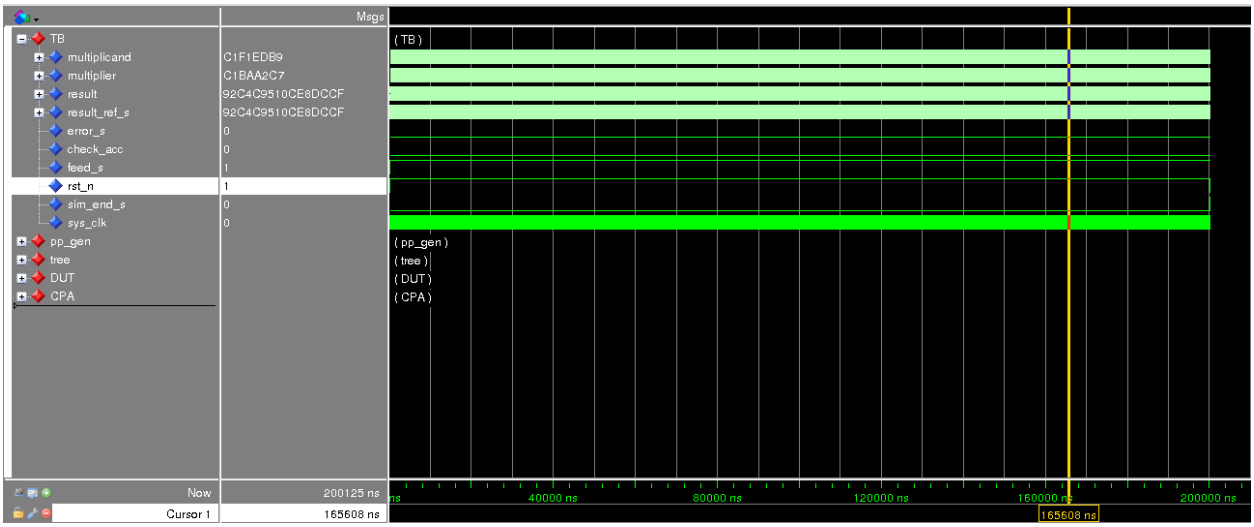

Fig.9: Simulation Result for Design (check_acc is still zero indicating that all test cases have passed)

Synthesis Results

Two synthesis tools were used to evaluate the results.

- Cadence RTL Compiler (RC). TSMC-0.18 process.
- Xilinx ISE. Spartan-6 device. Speed grade -3.
- The synthesis tool and environment is fixed for both designs.
- In cadence RC, we did not find a way to control the implementation style of the behavioral multiplier. In addition, the synthesis tool chooses a more optimized architecture as long as the constraint is set tighter.
- To achieve fair comparison as much as possible, several synthesis iterations were performed to achieve maximum working frequency for our multiplier with Tclk = 5600 ps. The same constraint was applied to the behavioral multiplier during synthesis.
- Using Xilinx ISE, the default multiplier implementation is done using DSP blocks. To achieve a reasonable and fair comparison, an attribute was added to the behavioral multiplier RTL to direct the tool to use LUT's for multiplier implementation instead of DSP modules.
- We have run synthesis on both multiplier architectures in the evaluation mode of the tool. The evaluation mode is intended to achieve the best timing closure without setting a timing constraint.
- Our multiplier architecture has achieved a higher operational frequency in the evaluation mode. To make sure of result, we have applied timing constraint and continued the flow down to place and route.
- Unlike Cadence flow, the multiplier architecture implemented by ISE is fixed. It's not changed according to set constraints.

Performance Analysis

Xilinx Flow Results

Xilinx synthesizer implements carry propagation addition using dedicated carry logic. The carry logic is a fast routing used for carry propagation paths. The routing resources of the carry logic is faster routing compared to normal routing between LUT's.

Adders are implemented in Xilinx FPGA in a carry ripple style. They rely on the optimized carry propagation path routing to minimize the ripple carry delay. The delay of the ripple carry addition has a linear dependence on the adder precision.

At the end of the multiplier reduction tree, we have to add two vectors each of 64 bit width. Xilinx uses ripple carry addition while we are using kogge-stone adder. The delay of kogge-stone adder has a log-scale dependence on the adder precision.The resultant delay of the kogge-stone adder is less than adder implemented by Xilinx synthesizer despite the fast carry logic used. This is one of the reasons why our multiplier architecture is faster under Xilinx environment.

In the following results, our multiplier architecture will be given the name "multiplier_top" where the behavioral multiplier is mentioned under the name "auto_multiplier". These names are their HDL respective names.

Cadence Flow Results

The two multipliers were synthesized using same timing constraint. However, our multiplier architecture reports higher total power due to increased number of gates. Larger number of gates means larger number of gates switching => increased dynamic power. Larger number of gates will result as well in increased leakage power.

Synthesis Reports

Xilinx ISE post synthesis results. No timing constraint entry. Timing estimates is calculated in performance evaluation mode.

	Area(LUT)	Frequency(MHz)
multiplier_top	1887	143
auto_multiplier	1577	131

Table 3

Xilinx ISE post PAR results. Timing constraints set.

	Area(LUT)	Frequency(MHz)	Power(mW)
multiplier_top	1874	145	234.79
auto_multiplier	1141	125	113.18

Table 4

Cadence RC post synthesis results.

	Area(cells)	frequency	Total Power(uW)
multiplier_top	7452	178	28240.141
auto_multiplier	6557	178	21247.131

Table 5

Directory Structure

- The following figure shows the directory structure for files and folders of our project.

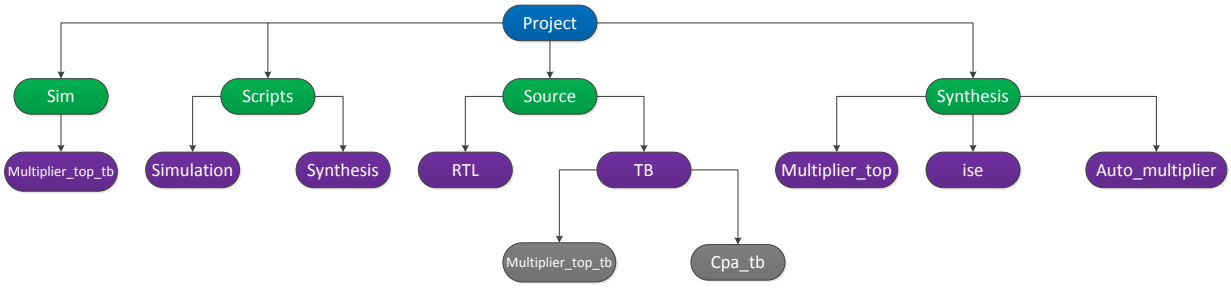

Fig.10: Directory Structure of project files

And here is the description of every folder in structure

- sim/multiplier_top_tb
  - The simulation output and reference directory
- scripts/sim
  - sim.tcl => modelsim simulation script.
- scripts/synthesis
  - Cadence synthesis scripts.
- source/tb/
  - multiplier_top_tb.vhd => multiplier test bench.
  - cpa_tb => kogge stone adder test bench.
  - vhdl_src.f => listing file for HDL sources, used by simulator for compilation.
- source/tb/multiplier_top_tb/
  - gen_vec.m => generate random test cases for multiplier using MATLAB.
  - gen_vec.c => generate random test cases for multiplier using C programming language.
  - multiplier.txt, multiplicand.txt => input operands.
  - tc_cfg.do, wave.do => used by Modelsim during simulation.
  - results.txt => reference results generated by MATLAB script.
- source/tb/cpa_tb
  - opa.txt, opb.txt => random 16-bit operands to test CPA with width=16
  - tc_cfg.do, wave.do => used by Modelsim during simulation.
  - gen_add_vec.m => generate random test cases for CPA.
  - results.txt => reference results generated by MATLAB script.
- source/rtl
  - Include all synthesizable HDL code used in the project.
- synth/multiplier_top

    - rpt/* => multiplier synthesis reports.
- synth/auto_multiplier
    - rpt/* => behavioral multiplier synthesis reports.
- synth/ise
    - Xilinx-ISE synthesis script and synthesis results.