To the Graduate Council:

I am submitting herewith a thesis written by Gabriel Cozmin Chereches entitled "Design and Verification of a Reusable Self-Reconfigurable Gate Array Architecture". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Don Bouldin

Major Professor

We have read this thesis
and recommend its acceptance:

Dr. Gregory Peterson

Dr. Itamar Elhanany

Accepted for the Council:

Anne Mayhew

Vice Chancellor and Dean of
Graduate Studies

(Original signatures are on file with official student records.)

Design and Verification of a Reusable Self-Reconfigurable Gate Array Architecture

A Thesis
Presented for the
Master of Science Degree

University of Tennessee, Knoxville

Gabriel Cozmin Chereches

December 2005

to titu (my father) and to ron

Acknowledgments

I would like to thank my academic and thesis advisor Dr. Donald W. Bouldin for his support and guidance throughout my graduate school and for taking an interest in me pursuing this research and providing me with the facilities in the Microelectronic System Research Lab at the University of Tennessee. I would like to thank Dr. Gregory D. Peterson and Dr. Itamar Elhanany for their interest in this work and for serving on my thesis committee.

I would like to acknowledge the University of Southern California and the University of Trento (Italy) for coming up with such an innovative Self-Reconfigurable device which caught our attention (Dr. Bouldin and I) – thanks to R. Sidhu, A. Mei, and V. Prasanna.

I would like to thank Dr. Roger Parsons, Director of the Engage Engineering Fundamentals Program at the University of Tennessee, for offering me the Graduate Teaching Assistant position and allowing me to be part of one of the nation's most innovative freshman engineering programs. I also would like to thank Dr. Fred Gilliam, the former Associate Dean of the College of Engineering, for his guidance throughout my education.

I would like to thank the Athletic Department at the University of Tennessee for supporting me while in graduate school by awarding me the Herman Hickman Memorial Scholarship. Dave, JT, Joe, and all my coaches, have made a difference and I would like to express my appreciation.

I would like to thank the Environmental Systems Corporation, especially Adam Engle, for offering me a great job while being tied up with my academic goals.

I am very thankful for having a great family – Magdalena (my mother), Anca (my sister), Lucian (my brother), and Jim Hoyle, thanks for believing in me.

I would like to thank the class of ECE 652 (spring 2003) for sharing their knowledge and support. And I would like to express my appreciation for those that have been there for me – and in no particular order: Lee Blask, Bree, Joel, Bird, M. Burton, Wei, Akila, Scott, Kay Shanahan, and Tyler Johnson.

Abstract

This thesis presents the design and verification of a Self-Reconfigurable Gate Array architecture *(SRGA-UT)* created for reuse, and available with a step-by-step tutorial and comprehensive documentation.

The original SRGA [1], created at the University of Southern California, is an innovative architecture for a reconfigurable device that allows single cycle context switching and single cycle random access to a unified on-chip configuration/data memory. The key architecture that enables the above two features is the use of a mesh of trees based interconnect with logic cells and memory blocks at the leaf nodes and identical switches at the parent nodes.

The *SRGA-UT* was adapted by making necessary modifications to the original design, to be implemented using the available University of Tennessee electronic design automation tools. An 8x8 array of PEs (Processing Elements) was synthesized and routed targeting a standard cell library for a 0.18 µm process. The synthesized design can store eight configuration contexts in each PE (this number can be modified by editing the Verilog files). The place and route generated a core-chip size of 5,413,300 μm^2, and contains 354,053 number of gates. The step-by-step tutorial demonstrates that the *SRGA-UT* design is capable to switch context and perform memory access operations in a single clock cycle.

ModelSim tools were used for verification and simulation at all levels, Design Compiler executed the synthesis and created the netlist design, and First Encounter SoC performed the place and route and created the delay constraints.

Table of Contents

Chapter 1 Introduction **1**

1.1 Thesis Goals 1

1.2 Outline of Thesis 2

Chapter 2 Background **4**

2.1 Reconfigurable Technology 4

2.2 Design Reuse 5

2.3 *SRGA-USC* 6

Chapter 3 Component Background **10**

3.1 Components Overview 10

3.2 Registers 10

3.3 Configuration Word 13

3.4 Logic Cell 16

3.5 Memory Cell 20

3.6 Switch Structure 23

3.7 PE Structure 25

3.8 2x2 Array 27

3.9 8x8 Array 27

Chapter 4 *SRGA-UT* Implementation **29**

4.1 *SRGA-UT* Overview 29

4.2 EDA Tools 29

4.3 Setting up Files 31

4.4 8x8 Array Step-by-Step Tutorial 32

Array 8x8 Pre-Synthesis 36

Array 8x8 Synthesis 40

First Encounter Tools 43

Step 1: Setting up the files 43

Step 2: Import the Design 43
Step 3: Specify the Chip Size 44
Step 4: Power Planning 44
Step 5: Global Net Connections 46
Step 6: Standard Cell Placement 46
Step 7: Add Filler Cells 48
Step 8: Route Power 48
Step 9: Final Route 48
Step 10: Extract RC 50
Step 11: Calculate Delay 50
Step 12: Results, Save, and Restore Design 51
Final Layouts 51
Array 8x8 Post Layout Simulation 51

Chapter 5 Implementing the *SRGA-UT* Sub-Designs 55

5.1 Logic Cell Implementation 55
Logic Cell Pre-Synthesis 56
Logic Cell Synthesis 57
Logic Cell Place and Route 59
5.2 Memory Cell Implementation 63
5.3 Switch Implementation 66
5.4 PE Implementation 68
5.5 2x2 Array Implementation 72

Chapter 6 *SRGA-UT* Results, Conclusion and Future Possibilities 76

6.1 Results 76
6.2 Conclusion and Future Possibilities 78

Reference 79

VITA 81

List of Figures

Figure 2.2.1: Design for Reuse Diagram [5] 6
Figure 2.3.1: SRGA Architecture 7
Figure 3.2.1: OR and MOR Registers 11
Figure 3.2.2: Periphery Registers (a) N x N Array of PEs [1] (b) PE 12
Figure 3.3.1: Configuration Word Format (a) Diagram and (b) From Configuration File 14
Figure 3.3.2: Full Add Configuration (a) Truth Table (b) LUT 15
Figure 3.3.3: Full Subtract Configuration (a) Truth Table (b) LUT 15
Figure 3.4.1: (a) LUT Structure and (b) Flip-Flop Structure 16
Figure 3.4.2: Logic cell Structure [1] 17
Figure 3.4.3: Internal Functions of the LUT 18
Figure 3.4.4: Input and Output Set of Wires for the Logic Cell Muxes 19
Figure 3.5.1: Memory Block Structure [1] 21
Figure 3.5.2: Defining the Size of the Memory Array 21
Figure 3.6.1: Switch Structure 24
Figure 3.7.1: PE Structure [1] 26
Figure 3.8.1: Structure of the 2x2 Array 28
Figure 3.9.1: Structure of the 8x8 Array 28
Figure 4.2.1: SRGA-UT Design Flow 31
Figure 4.4.1: Configuration Word Format for the 8x8 Array 33
Figure 4.4.2: Importing Configuration Word Section of *test_array8x8.v* 35
Figure 4.4.3: Assigning Inputs to the 8x8 Array 35
Figure 4.4.4: Array 8x8 Pre-Synthesis - Loading Memory Blocks 37
Figure 4.4.5: Array 8x8 Full Add Demo Schematic 38
Figure 4.4.6: Array 8x8 Pre-Synthesis - Applications 39
Figure 4.4.7: Array 8x8 Pre Synthesis Workspace 40

Figure 4.4.8: Array 8x8 Synthesis - Loading Memory Blocks 42
Figure 4.4.9: Array 8x8 Pre-Synthesis - Applications 42
Figure 4.4.11: Encounter Tools - Importing Design 44
Figure 4.4.12: Encounter Tools - Specify Chip Size 45
Figure 4.4.13: Encounter Tools - Power Planning 45
Figure 4.4.14: Encounter Tools - Global Net Connections 46
Figure 4.4.15: Encounter Tools - Standard Cell Placement 47
Figure 4.4.16: Encounter Tools - Cell Placement Views 47
Figure 4.4.17: Encounter Tools - Add Filler Cells 48
Figure 4.4.18: Encounter Tools - Route Power 49
Figure 4.4.19: Encounter Tools - Final Route 49
Figure 4.4.20: Encounter Tools - (1) Extract RC and (2) Calculate Delay 50
Figure 4.4.21: Encounter Tools - Final Pace and Route of 8x8 Array 52
Figure 4.4.22: Array 8x8 Post Layout - Loading Memory Blocks 53
Figure 4.4.23: Array 8x8 Post Layout – Applications 54
Figure 5.1.1: Section of Test-bench for Logic Cell Block 56
Figure 5.1.2: Logic Cell - Pre-Synthesis Simulation 58
Figure 5.1.3: Logic Cell - Post-Synthesis Simulation 60
Figure 5.1.4: Logic Cell Layout after WRoute 61
Figure 5.1.5: Logic Cell - Post-Layout Simulation 62
Figure 5.2.1: Section of Test Bench for Memory Cell Block 64
Figure 5.2.2: Memory Cell Testing 65
Figure 5.3.1: Section of Test Bench for Switch Module 67
Figure 5.3.2: Switch Testing 67
Figure 5.4.1: PE Testing – Loading Configuration 69
Figure 5.4.2: PE Testing – Operations 70
Figure 5.5.1: 2x2 Array Testing – Loading Configuration 73
Figure 5.5.2: 2x2 Array Testing – Operations 74
Figure 6.1.1: 8x8 SRGA-UT Results 77

List of Acronyms

CCR – Current Context Register
CMAR – Context and Memory Address Register
CMR – Column Mask Register
CSMR – Context Switch Mask Register
DCR – Destination Column Register
DRMR – Data Restore Mask Register
DRR – Destination Row Register
ECE – Electrical and Computer Engineering
EDA – Electronic Design Automation
FF – Flip-Flop
FPGA – Field Programmable Gate Array
GDS – Graphic Design System
HDL – Hardware Description Language
IC – Integrated Circuit
IP – Intellectual Property
I/O – Input/Output
LIN – Logic Interconnection Network
LUT – Lookup Table
MIN – Memory Interconnection Network
MOR – Memory Operation Register
OR – Operation Register
PE – Processing Element
RAM – Random Access Memory
RMR – Row Mask Register
RTL – Register Transfer Level
SCR – Source Column Register
SDF – Standard Delay Format

SoC – System on Chip

SRGA – Self-Reconfigurable Gate Array

SRGA-USC – Self-Reconfigurable Gate Array - University of Southern California

SRGA-UT – Self-Reconfigurable Gate Array - University of Tennessee

SRR – Source Row Register

Chapter 1: Introduction

1.1 Thesis Goals

The objective of this thesis was to use the open code of the Self-Reconfigurable Gate Array (SRGA) architecture, created by the Department of EE-Systems at University of Southern California and the Department of Mathematics at the University of Trento (Italy), and make the minimum adjustments necessary to adapt and implement using the available EDA (electronic design automation) tools at the Department of ECE at the University of Tennessee. The outcome of this thesis was to end up with a *SRGA-UT* design for reuse, accessible with a step-by-step tutorial and comprehensive documentation.

To achieve this goal, a great deal of knowledge of the *SRGA-USC* architecture was required. The first step was to examine the RTL (register transfer level) design, make the appropriate adjustments and pass the pre-synthesis verification stage. The second challenge was to synthesize the RTL design, create the netlist and pass the synthesis verification stage. And the last step was to generate the place and route from the netlist design, produce the timing delay files, and test the final design for proper functionality.

The EDA tools used to implement the SRGA were: ModelSim for verification and simulation at all stages, Design Compiler for synthesis and creating the netlist, and First Encounter for performing the place and route and creating the delay files.

1.2 Outline of Thesis

The introduction states the thesis goals. An open self-reconfigurable design, *SRGA-USC*, developed at the University of Southern California was taken and implemented using the available EDA tools at the University of Tennessee. The intent was to make the minimum changes, while the focus was to develop a *SRGA-UT* design for reuse with complete documentation and a step-by-step implementation tutorial.

The second chapter will present a background of the IC (integrated circuit) technology – focusing on the reconfigurable devices such as the *SRGA-USC*, an innovative architecture for a self-reconfigurable device that allows single cycle context switching and single cycle random access to the on-chip configuration memory. This chapter also describes the importance of the design for reuse techniques which can greatly decrease the time-to-market by reducing the design cycle and the manufacturing cycle.

The third chapter illustrates the background of each component of the SRGA architecture including the changes made to the original *SRGA-USC* design to adapt and implement it at the University of Tennessee. The *SRGA-UT* design retains the overall original architecture but was mostly adjusted to be able to pass the verification stages of synthesis and place and route.

The fourth chapter provides a step-by-step tutorial of an 8x8 *SRGA-UT* array. The steps will confirm the pre-synthesis RTL verification (using ModelSim), the synthesis process (using Design Compiler) and verification of the netlist (using ModelSim), and the place and route process (using First Encounter) and verification of the delay constraints (using ModelSim).

The fifth chapter shows the implementation of the 8x8 array sub-blocks in the same manner as the tutorial. Some blocks were small enough to be proven just by verifying the pre-synthesis RTL step.

The sixth chapter will describe the results of the *SRGA-UT* implementation and followed by the conclusion remarks and suggestions for future possibilities. The overall thesis provides a detailed documentation for a reusable design.

Chapter 2: Background

2.1 Reconfigurable Technology

Reconfigurability denotes the potential of a system to dynamically change its behavior usually in response to changes in its environment. In the computing world, the Field Programmable Gate Arrays (FPGAs) are the most popular means of accomplishing reconfigurability. An FPGA consists of an array of programmable logic elements and programmable interconnects. The logic elements can be logic gates (AND, OR, XOR, Invert), lookup tables (memory usually RAM), or flip-flops [2]. The interconnects allow the logic elements to be connected as needed by the design.

The logic elements and interconnects can be programmed by the customer so that the FPGA can perform a certain functionality. This functionality can be reconfigured to suit new application requirements desired by the customer by writing appropriate bits into the configuration memory. The challenges with most FPGAs however, whether they are reconfigured at compile time or runtime, is that they require an external source to execute the reconfiguration.

A device, that is capable to generate configuration bits at runtime and use them to modify its own configuration, exhibits self-reconfiguration. A self-reconfigurable device needs to be able to store multiple contexts of configuration information and context switch between them. The configured logic should be able to access any of the contexts of information stored and perform self-reconfiguration by modifying the contents of the information stored. When the configured logic has made the modifications to the configuration information, the device should be able to switch context to any of the contexts of configuration stored. For an efficient self-reconfiguration to occur, the device should be able

to configure logic to perform: (1) fast context switching and (2) fast random access of the configuration information stored. The *SRGA-USC* [1] was designed to perform the fast context switching and fast random access of the configuration stored in one clock cycle. The *SRGA-USC* is capable of storing eight contexts of configuration information (this number is editable).

Another device that is capable of storing four contexts of configuration information on-chip and switch between them on a clock cycle basis is the Sanders CSRC [3]. This device can also load configurations while other contexts are active. Despite the fast switching and loading capabilities, it only provides serial configuration memory access which can take hundreds of clock cycles to access a particular location. The Berkeley HSRA [4] is capable of accessing the configuration information in a fast manner but it takes hundreds of clock cycles to switch context.

2.2 Design Reuse

Time-to-market is a crucial aspect for the survival of many IC manufacturing companies in such a competitive environment. Time-to-market may be optimized by reducing the design cycle and by reducing manufacturing cycle. The design cycle may be greatly reduced and the quality of the designs may be increased by providing designs with reuse [5]. IP (intellectual property) blocks such as configurable I/O (input/output), power and ground grids, block RAMs (random access memories), and timing generators were some of the first blocks created for reuse as sub-designs in larger projects such as SoCs (system-on-chips). If a sub-design were to be developed from scratch, it would take a lot more time than to adapt the reusable block. Figure 2.2.1 shows that without planned design reuse, the total time for development is proportional to the

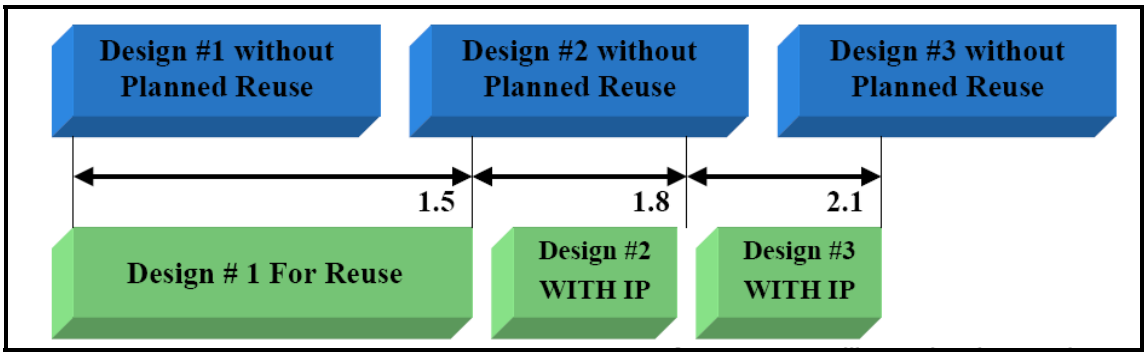


Figure 2.2.1: Design for Reuse Diagram [5]

number of sub-designs, given that each module has the same complexity. During development with planned design reuse, the designer will spend more time in creating a reusable block by providing comprehensive documentation and more adaptable interface.

The design for reuse techniques are closely studied in today's SoC development. A basic combination of such reusable features is known as a platform. The platform used to implement a SoC greatly impacts all of the issues and is the fundamental decision the hardware designers must make at the start of each new project. By 2010 the percentage of IP contained in a System-on-Chip application is predicted to grow to 95% [6].

One of the goals for this thesis is to end up with a *SRGA-UT* design for reuse, accessible with a step-by-step tutorial and comprehensive documentation.

2.3 *SRGA-USC*

The original Self-Reconfigurable Gate Array Architecture [1] is an open core design implemented by the Department of EE-Systems at University of Southern California and the Department of Mathematics at the University of Trento (Italy).

The reconfigurable device allows single cycle context switching and single cycle random access to the unified on-chip memory which stores the configuration data. Both features are necessary for efficient self-reconfiguration. The context switching feature permits arbitrary regions of the chip to selectively switch context. The memory access feature allows data transfer between logic cells and memory locations as well as between memory locations. A mesh of trees based interconnects with logic cells and memory blocks at the leaf nodes and identical switches at the other nodes make it possible to perform the above features. Figure 2.3.1 shows the basic SRGA architecture of a 4x4 array of PEs.

The architecture can be of any N x N array of PEs. The PE sits at the leaf node of the mesh of trees interconnects and is composed of a logic cell, memory block, and memory interface. Each switch is identical except that some switches are connected in a column mesh network and some are connected in a row mesh network. Each PE is connected to a row switch and a column switch.

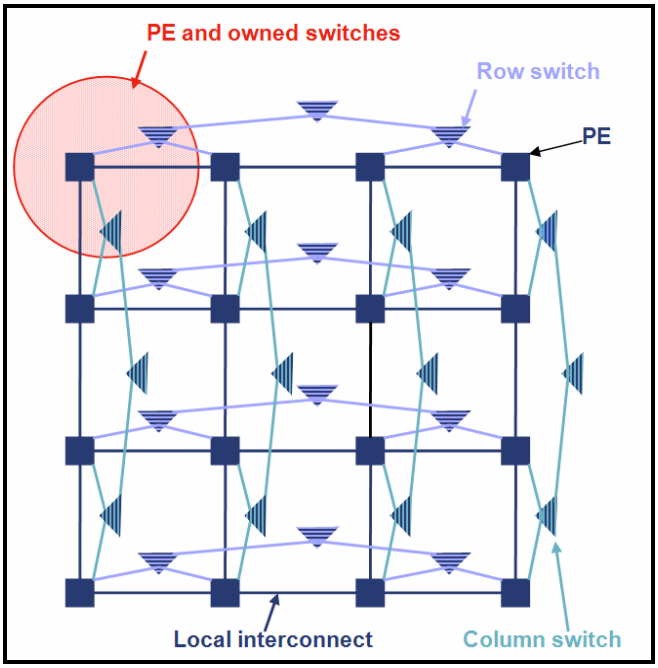


Figure 2.3.1: SRGA Architecture

The connection between the PEs through the switches is configured using two methods. The first method serves the same purpose as the interconnection network in a typical FPGA, connecting together the logic cells as specified by the configuration bits controlling the network switches. The second method is used for performing data transfer during the memory access operations. For this method, only one wire that carries signals in either direction is used for each connection.

The logic cell of each PE can connect directly to the four neighbor's logic cells. Fore example, the south output of a PE will become the north input of the neighbor PE directly to the south of the original PE – and/or – the east output of a PE will become the west input of the neighbor PE directly to the east of the original PE.

The SRGA contains three global registers – their contents are broadcast to all PEs, and four periphery registers – located along the boundary of the N x N PE array. The purpose of the global registers is to specify – the operation to be initiated, the source and destination of the data, and the context to switch to or the memory address to be accessed. The purpose of the periphery registers is to control which rows or columns will be the source or destination during the different functionalities. See section 3.2 for a more detailed description of the registers.

The Memory block stores the different configuration words which are 77 bits long each. During a memory access operation, the contents of the configuration words can be transferred between rows or between columns. All memory access operations complete in a single clock cycle. The operation can be a memory read (memory to logic cells), memory write (logic cells to memory), or memory transfer (memory to memory) depending on the contents of the global registers.

The memory interface generates the proper inputs to the memory block and to some extent to the logic cell by taking the register signals and combining them through basic logic gates to create the appropriate signals.

Chapter 3: Component Background

3.1 Components Overview

This chapter will describe in details the structure of each component of the *SRGA-UT*, which includes the changes made to the original *SRGA-USC* design. The changes, which will be pointed out throughout the next three chapters, were necessary for the design to be adapted and able to implement using the available EDA tools. The *SRGA-UT* design retains the overall original architecture but was mostly adjusted to be able to pass the verification stages of pre-synthesis, synthesis and place and route.

The *SRGA-UT* architecture consists of an array of N x N array of PEs. The PE sits at the leaf node of the mesh of trees interconnects and is composed of a logic cell, memory block, and memory interface (described later in this chapter). Each switch is identical except that some switches are connected in a column mesh network and some are connected in a row mesh network. Each PE is connected to a row switch and a column switch.

The N x N array of PEs is configured by inputting configuration contexts (described in section 3.3) into the memory array of each PE – which can store eight contexts of configuration. Once the memory is loaded, the *SRGA-UT* can switch context and perform memory operations in a single clock cycle. This is done through a number of global and periphery registers.

3.2 Registers

The *SRGA-USC* contains three global registers and four periphery registers. The memory interface module takes signals from the registers and creates the

internal signals *wrMem*, *WrLog*, *switchContext*, *data_In*, *Rmi*, and *Cmi*. These signals are used to perform context switching and memory operations.

The three global registers are the Operation Register (OR), the Memory Operation Register (MOR), and Context and Memory Address Register (CMAR). OR seen in figure 3.2.1 (a) is a 2-bit register that specifies what operation will be initiated in the next clock cycle. MOR seen in figure 3.2.1 (b) is a 2-bit register that specifies the source and destination of the data transfer for the next clock cycle when OR indicates memory access. Note: The order of the OR and MOR bits were changed in the figure bellow for consistency with the verilog code. CMAR, depending on the OR contents, specifies the context to switch to or the memory address to be accessed in the next clock cycle. CMAR consists of two fields: (1) the context field and (2) the offset field. The context field points to the memory column to be accessed. In testing this design, *nc* equals eight (8), the number of memory columns for the memory array. *nc* is defined and can be edited in the memory verilog files found in the /srga/tools/rtl/mem/ directory.

OR[0]	OR[1]	Operation
0	0	No operation
0	1	Context switch
1	0	Row memory access
1	1	Column memory access

(a)

MOR[0]	MOR[1]	Source and destination
0	0	Memory to memory
0	1	Memory to logic (read)
1	0	Logic to memory (write)
1	1	Logic to logic

(b)

Figure 3.2.1: OR and MOR Registers

The size of the context field can be calculated as: (2^X = *nc*) where *X* is the context field number of bits which equals to 3 (2^3 = 8). The offset field points to the memory row to be accessed. In the design, *cs* equals eighty (80), the number of output bits from the *Row Demux* (see memory cell section 3.5). The first seventy-seven (77) bits of *cs* are the number of rows for the memory array. *cs* can also be defined as the number of bits required to configure a logic cell and its two owned switches. Each memory array block consists of *nc* x *cs* bits.

The four periphery registers are SCR, SRR, DCR/RMR, and DRR/CMR, shown in figure 3.2.2 (a) and (b). All the periphery registers are N bits long to match the size of the N x N array of PEs.

SCR (Source Column Register), a set bit implies that the corresponding PE column will be the source for the next column memory access.

SRR (Source Row Register), a set bit implies that the corresponding PE row will be the source for the next row memory access.

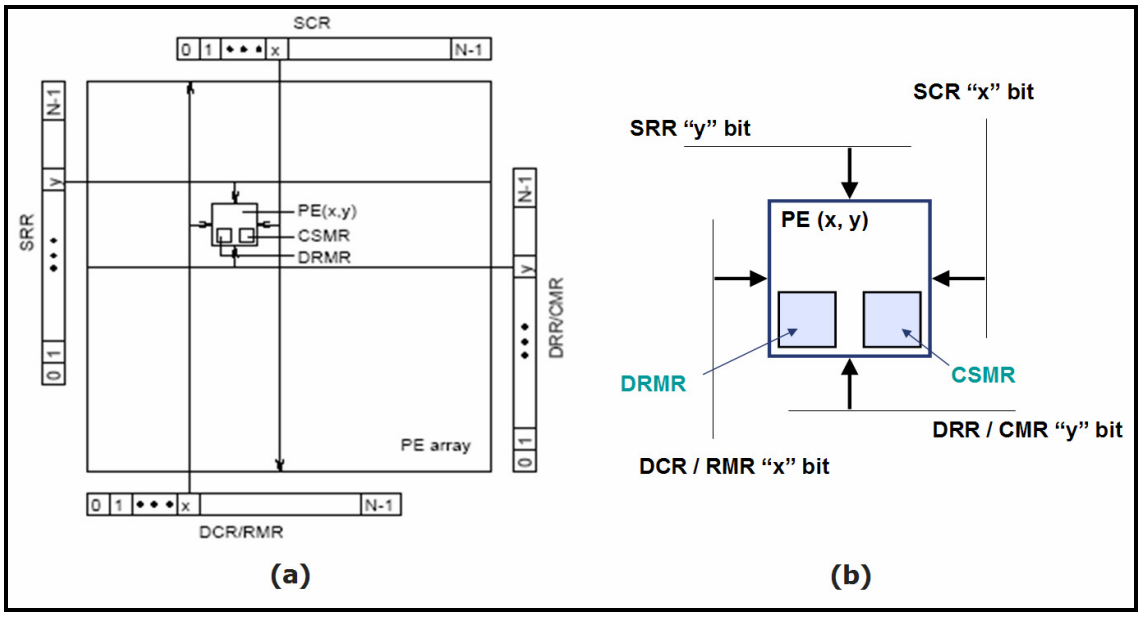


Figure 3.2.2: Periphery Registers (a) N x N Array of PEs [1] (b) PE

DCR/RMR (Destination Column Register)/(Row Mask Register), a set bit implies that the corresponding PE column will be the destination for the next memory access – or – during a row memory access, no data will be transferred for the corresponding column.

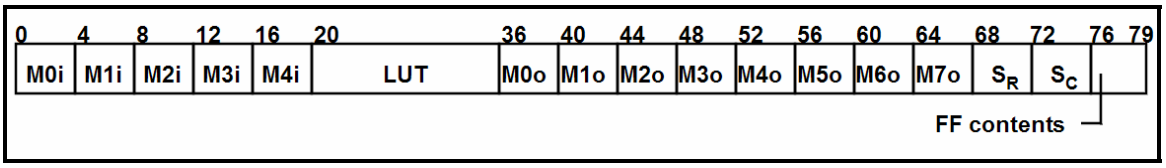
DRR/CMR (Destination Row Register)/(Column Mask Register), a set bit implies that the corresponding PE row will be the destination for the next memory access – or – during a column memory access, no data will be transferred for the corresponding row.

Each PE also contains two memory mapped registers CSMR and DRMR to give the design more flexibility. CSMR (Context Switch Mask Register), a set bit will not allow the corresponding PE to switch context even when a context switch operation occurs. DRMR (Data Restore Mask Register), a set bit prevents the corresponding flip-flop contents from being restored when a context switch operation occurs.

3.3 Configuration Word

The configuration word that is loaded to the memory array is 80-bit long. Figure 3.3.1 (a) and (b) shows the format of the configuration word. The word is composed of: select 4-bit input for five input muxes (M0i to M4i), configuration 16-bit input for the LUT, select 4-bit input for eight output muxes (M0o to M7o), select 4-bit input for the row switch (S_R), select 4-bit input for the column switch (S_C), and four bit contents of the FF.

For testing, different configuration word files were created and stored in the /srga/config_files/ directory, and called by the test-bench of each module.



(a)

```
1|Full adder
2|          1         2         3         4         5         6         7
3|01234567890123456789012345678901234567890123456789012345678901234567890123456789
4|00000011010000010010111010001001011010101011010000110000110011010010000000000000
5|iiiiiiiiiiiiiiiiiiiillllllllllllllllooooooooooooooooooooooooooooooooesssesssseeeef
6|0000111122223333444ttttttttttttttttt00001111222233334444555566667777errreccceeeef
```

(b)

Figure 3.3.1: Configuration Word Format (a) Diagram and (b) From Configuration File

Note: Only three bits $S_R[69:71]$ and $S_C[73:75]$ are used for the switches, and only one bit *FF[79]* is used for the flip-flop. Part (a) of the figure 3.3.1 was edited to be consistent with the configuration word format from the verilog code.

The four input muxes M0i to M3i generate the select bits for the LUT. For a detailed illustration of how the LUT works, refer to section 3.4. To test this design, 16-bit configuration words were created for a full adder and a full subtractor. The format of the LUT configuration is demonstrated in figure 3.3.2 and figure 3.3.3. It can be seen that the output of the truth table forms the configuration bits for the LUT.

The LUT can perform two boolean functions of three inputs and one output each and one boolean function of four inputs and one output. The truth tables are color coded to show how the configuration for the LUT must be applied. A binary and hexadecimal number format is shown which will be useful when generating the ModelSim simulations.

Carry-In bit L2i	Input bit L1i	Input bit L0i	Carry-Out bit L1o	Sum bit L0o
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a)

	Carry-Out bit L1o		Sum bit L0o	
Binary Format	1110	1000	1001	0110
Hex Format	e	8	9	6
LUT Configuration word	lutconfig[15:8]		lutconfig[7:0]	

(b)

Figure 3.3.2: Full Add Configuration (a) Truth Table (b) LUT

Borrow-In bit L2i	Input bit L1i	Input bit L0i	Borrow-Out bit L1o	Difference bit L0o
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

(a)

	Borrow-Out L1o		Difference L0o	
Binary Format	1101	0100	1001	0110
Hex Format	d	4	9	6
LUT Configuration word	lutconfig[15:8]		lutconfig[7:0]	

(b)

Figure 3.3.3: Full Subtract Configuration (a) Truth Table (b) LUT

3.4 Logic Cell

The logic cell block shown in figure 3.4.2 consists of a 16-bit LUT, a Flip-Flop (FF), five input muxes (M0i, M1i, M2i, M3i, and M4i), and eight output muxes (M0o, M1o, M2o, M3o, M4o, M5o, M6o, and M7o). The input muxes (M0i, M1i, M2i, and M3i) are used to generate the four control bits *(L0i, L1i, L2i,* and *L3i)* for the LUT. See also figure 3.4.1 (a). The input mux M4i generates the signal *(L4i)*, the input to the Flip-Flip when the Switch_context signal is '0'. When the Switch_context signal is '1', the context_state is the input to the Flip-Flop. See also figure 3.4.1 (b).

The LUT can perform two boolean functions of three inputs and two outputs or one boolean function of four inputs and one output. As can be seen from the logiccell.v section in figure 3.4.3, the mux8 *u_lut0* and *u_lut1* instances will perform the two boolean functions of three inputs and two outputs.

$f(L0i, L1i, L2i) = L0o$

$f(L0i, L1i, L2i) = L1o$

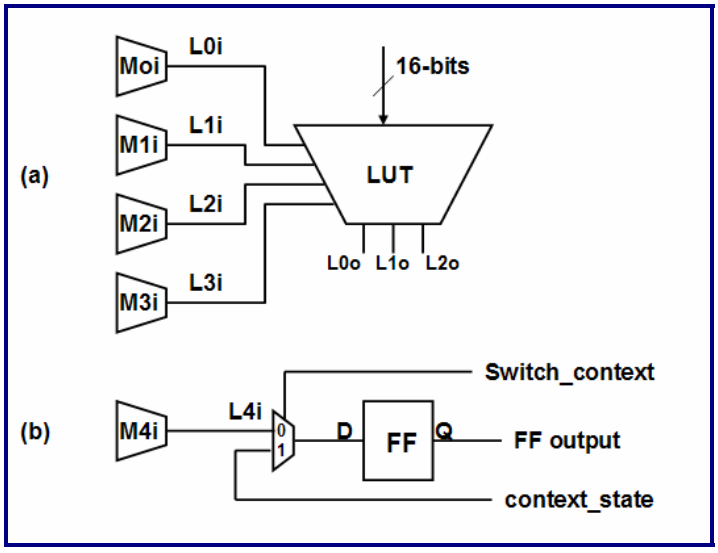


Figure 3.4.1: (a) LUT Structure and (b) Flip-Flop Structure

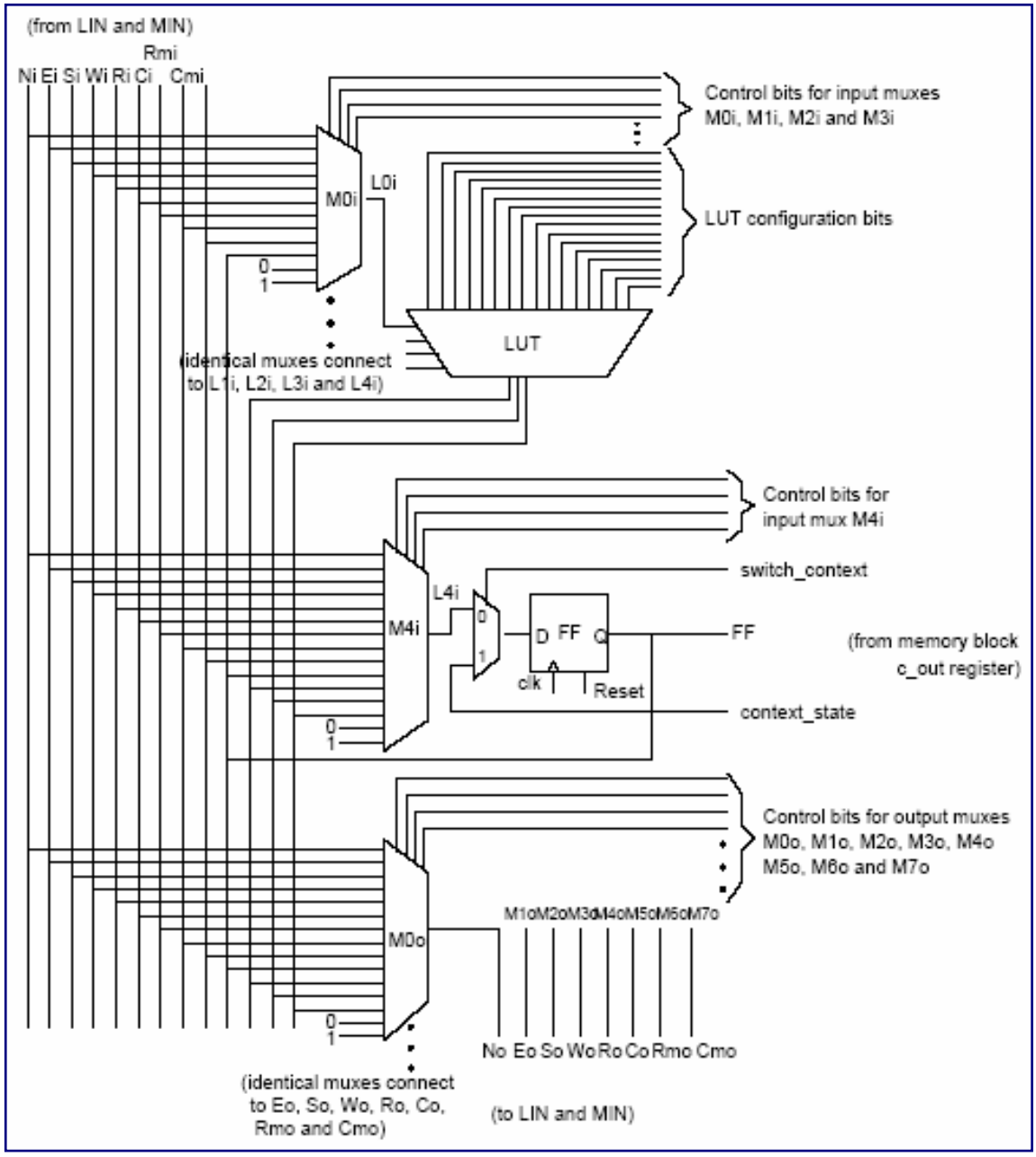


Figure 3.4.2: Logic cell Structure [1]

```
assign  sel_lut0 = {li2,li1,li0};
assign  sel_lut1 = {li2,li1,li0};
assign  sel_lut2 = li3;
assign  lutconfig0 = lutconfig[7:0];
assign  lutconfig1 = lutconfig[15:8];

mux8  u_lut0 (.sel      (sel_lut0),
              .dataIn   (lutconfig0),
              .dataOut  (lo0));

mux8  u_lut1 (.sel      (sel_lut1),
              .dataIn   (lutconfig1),
              .dataOut  (lo1));

mux2  u_lut2 (.Y      (lo2),
              .A      (lo0),
              .B      (lo1),
              .S0     (sel_lut2));
```

Figure 3.4.3: Internal Functions of the LUT

The instance *u_lut2* of mux2 takes the signals *L0o* and *L1o* and generates the output *L2o* depending on the state of *Li3*. This is the internal function of the LUT that performs the one boolean function of four inputs and one output.

f (L0i, L1i, L2i, L3i) = L2o

The signal *L0i* from figure 3.4.2 is equivalent to li0 from figure 3.4.3. The same applies to *L1i*, *L2i*, *L3i*, *L0o*, *L1o*, and *L2o*.

The inputs to the PE *(Ni, Ei, Si, Wi, Ri, Ci, Rmi, Cmi)* are grouped together in the *logiccell.v* module to form the 8-bits *inBus* set of wires. See figure 3.4.4. The 16-bits *inMuxBus* is the set of wires which groups together the *inBus*, *lo3* (the output Q from the Flip-Flop), and other strategically placed ones and zeros.

inMuxBus[15:0] = {1, 0, lo3, 0, 0, 0, 0, 0, inBus}
inMuxBus[15:0] = {1, 0, lo3, 0, 0, 0, 0, 0, Cmi, Rmi, Ci, Ri, Wi, Si, Ei, Ni}

```
assign  inBus = {cmi, rmi, ci, ri, wi, si, ei, ni};
assign  inMuxBus = {1'b1, 1'b0, lo3, 5'b0, inBus};
assign  outBus = {lo2, lo1, lo0, 2'b0, inBus};
assign  outMuxBus ={1'b1, 1'b0, lo3, outBus};
```

(a)

Binary and Decimal number for the control bits of each mux	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
inMuxBus[15:0]	1	0	lo3	0	0	0	0	0	Cmi	Rmi	Ci	Ri	Wi	Si	Ei	Ni
outMuxBus[15:0]	1	0	lo3	L2o	L1o	L0o	0	0	Cmi	Rmi	Ci	Ri	Wi	Si	Ei	Ni

(b)

Figure 3.4.4: Input and Output Set of Wires for the Logic Cell Muxes

The *inMuxBus* is the input to each of the four input muxes M0i, M1i, M2i, and M3i. This architecture allows the flexibility to use any of the signals in the *inMuxBus* to become the control bits *L0i*, *L1i*, *L2i*, and *L3i* for the LUT. This is done by the control bits for the input muxes which come from the configuration words. For example, the input *Ni* can be the output for all the muxes (including output muxes discussed later in this section) if the control bits from the configuration word for each mux are set to '0000'.

The 13-bits *outBus* is the set of wires which groups together the *inBus*, the outputs from the LUT (*L0o*, *L1o*, and *L2o*), and other strategically placed zeros.

outBus[12:0] = {L2o, L1o, L0o, 0, 0, inBus}

The 16-bits *outMuxBus* is the set of wires which groups together the *outBus*, *lo3* (the output Q from the Flip-Flop), and other strategically placed ones and zeros.

outMuxBus[15:0] = {1, 0, lo3, outBus}
outMuxBus[15:0] = {1, 0, lo3, L2o, L1o, L0o, 0, 0, Cmi, Rmi, Ci, Ri, Wi, Si, Ei, Ni}

The *outMuxBus* is the input to the input mux M4i and to the output muxes (M0o, M1o, M2o, M3o, M4o, M5o, M6o, and M7o). In this manner any of the signals in the *outMuxBus* can become the output for the muxes depending on the control bits from the configuration word.

The complete flexibility in configuring connections allows the LUT and Flip-Flop to be used while other signals are routed through the logic cell – to perform operations as inputs for other PEs.

3.5 Memory Cell

The memory block shown in figure 3.5.1 consists of a memory cell array, a row decoder, a column decoder, three current context registers (CCR) with three context field muxes, a configuration word register, and a data-out row mux. The memory cell array block *(memArray.vhd)* was redesigned to reduce the number of instances. The original memory array block *(memArray.v)* instantiated seventy-seven (77) sub-blocks of *storageCellRow*. The *storageCellRow* block then instantiated eight (8) *storageCell* sub-blocks which also contains other instances. When used with the 8x8 array of PEs, the large number of instances for the design would surpass the allocated amount of files allowed in a folder. This happens when running the synthesis of the 8x8 array. The new *memArray.vhd* block was created with one instance *memArray*, and also by using the mixed compile method, the design was successfully synthesized.

The memory cell array is internally arranged as a *nc* columns of *cs* storage cell rows. The *nc* and *cs* can be edited in the module *memArray.vhd* to become any memory array size as seen in figure 3.5.2. In this design the size of *nc* = 8 [*colSelWidth* to 0] and the size of *cs* = 77 [*rowSelWidth* to 0].

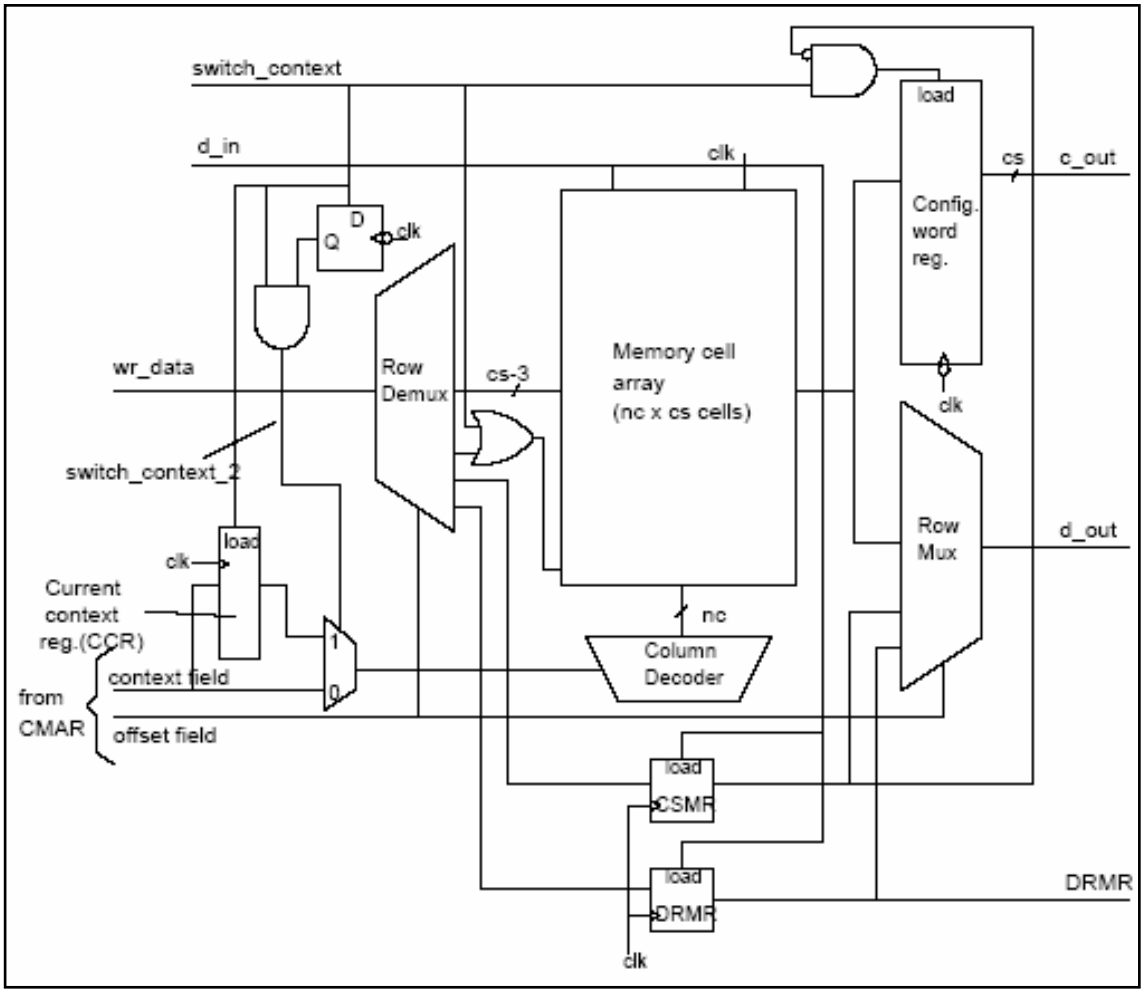


Figure 3.5.1: Memory Block Structure [1]

```
`define colSelWidth 7
`define rowSelWidth 76
wire [`colSelWidth:0] colSel;
wire [`rowSelWidth:0] rowSel;
```

Figure 3.5.2: Defining the Size of the Memory Array

Each column can store a configuration word in this case 77-bits long, thus the memory array can store eight different configuration words. In testing the 8x8 array, only four (4) configuration contexts are used thus leaving the last four columns of memory vacant in each PE. By editing this in the future, the number of gates in each PE will be reduces by about one third.

The row decoder *u_rowdecode* is instantiated in the *memcell.v* and it selects the row to write the data to. The column decoder *u_coldecode* is instantiated in the *memcell.v* and it selects the column to write data to. The row decoder and column decoder together select a specific memory cell to be accessed.

The three CCRs, *u_10EDFFTRX1*, *u_11EDFFTRX1*, *u_12EDFFTRX1*, and the three context field muxes *u31_MX2X1*, *u32_MX2X1*, *u33_MX2X1*; are instantiated in the *memcell.v*. Their functionality is to perform a context switch operation in a single clock cycle. This is done as follows: At the positive edge which marks the beginning of the next clock cycle, the CMAR and OR contents are registered and broadcast to all the memory blocks. In each memory block, in the first half of the clock cycle, the new configuration is loaded into the configuration word register (for details refer to figure 3.5.1 above) when *switch_context* is "1" and *switch_context_2* is "0" *(swith_context_2* is *switchContextHalf* in *memcell.v)*. In this manner, the context field of CMAR gets applied to the column decoder thus selecting the memory column to place the new configuration. At the negative edge of the clock cycle, the new configuration word gets loaded into the configuration word register and ready to be used.

The configuration word register *u_reg_77bnlr* is instantiated in the *memcell.v* and it loads and stores the configuration word from the memory array that is selected by the column decoder. This is done internally in the *memArray.vhd*.

When a switch context occurs, the load signal to the configuration word register will enable the new selected memory column to be loaded in a single clock cycle.

The output of the configuration word register sends the 16-bit configuration to the LUT, the control 4-bit configuration to each of the logic cell muxes, the configuration bits for the switches, and the signal to the Flip-Flop for the generation of the context state.

The data-out row mux *u_mux80* is instantiated in *memcell.v* and it performs the bit transfers during memory operations. The offset field signal will select the memory bit (row) of the currently used configuration word (memory column). In this manner a single memory cell can be transmitted to another memory block to change its configuration. The signals CSMR and DRMR are inputs to the data-out row mux and can also be accessed through memory operations.

3.6 Switch Structure

The switch is the most important part of the mesh of trees network because the switches together with connecting wires create the mesh network. Each PE is connected to one row and one column switch. The row and column switches are identical. Each switch is connected to two child nodes and a parent node where the child nodes can be other switches or PEs. The switch shown in figure 3.6.1 is composed of two parts: (1) the Logic Interconnection Network (LIN) and (2) the Memory Interconnection Network (MIN).

The LIN is composed of three muxes with select inputs (*c_out[68]*, *c_out[69]*, and *c_out[70]* for the row switch, and *c_out[72]*, *c_out[73]*, and *c_out[74]* for the column switch) coming from the configuration word.

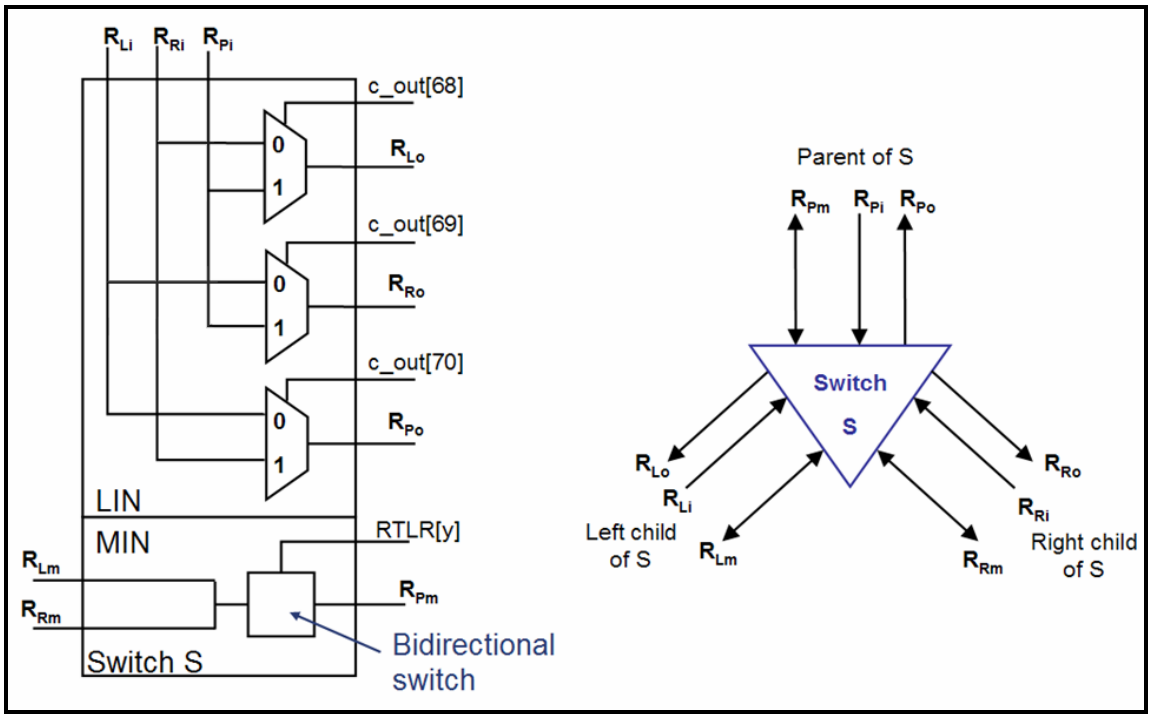


Figure 3.6.1: Switch Structure

This setup allows any input to be connected to any output without restrictions except connecting an input to its output pair. This way a signal is not routed back where it came from.

The MIN part is composed of a bidirectional tri-state circuit where the wires connected to it can flow signals in both directions. In this manner, by opening all switches at a particular network level, a memory tree can be divided into multiple smaller trees. The wires from the child connections R_{Lm} and R_{Rm} are connected together thus any signal coming from any parent or child node will be transmitted to all nodes. The tri-state circuit will determine which way the signal flows.

Note: The schematic connections to the LIN muxes were changed in the schematic above (from the original) to be consistent with the verilog code.

3.7 PE Structure

The PE block shown in figure 3.7.1 consists of a logic cell, a memory cell, a memory interface, a row switch, and a column switch. At this level all the modules are connected together and all the inputs and outputs to all the modules, including registers will go through the PE. The register signals are processed by the memory interface which will create the internal signals *wrMem*, *WrLog*, *switchContext*, *data_In*, *Rmi*, and *Cmi*. These signals are used to perform context switching and memory operations.

The logic cell connections through the PE are: The LIN nearest neighbors' connections (inputs *Ni*, *Ei*, *Si*, *Wi*, and outputs *No*, *Eo*, *So*, *Wo*). The mesh network connections (inputs *Ri*, *Ci*, *Rmi*, *Cmi*, and outputs *Ro*, *Co*, *Rmo*, *Cmo*).

Configuration bits from the memory cell (configuration word register) are connected through *c_out* (*cOut* in the RTL code) to the select bits for the logic cell's muxes and input to the LUT. Also, configuration bits are passed to the row and column switches.

The inputs and outputs *Cm* or *Rm* are connected together. This setup indicates the bidirectional connectivity with the MIN part of the switch. If the direction is input to PE, the *Cm* or *Rm* will be input signals shown on the upper left-hand side of figure 3.7.1. If the direction is output from PE, the *Cm* or *Rm* will be output signals shown on the lower right-hand side of figure 3.7.1. The test-bench inputs data for the configuration word file to the PE (logic cell and memory cell) through one of the inputs *Cm* or *Rm* depending whether a column or row operation is expected.

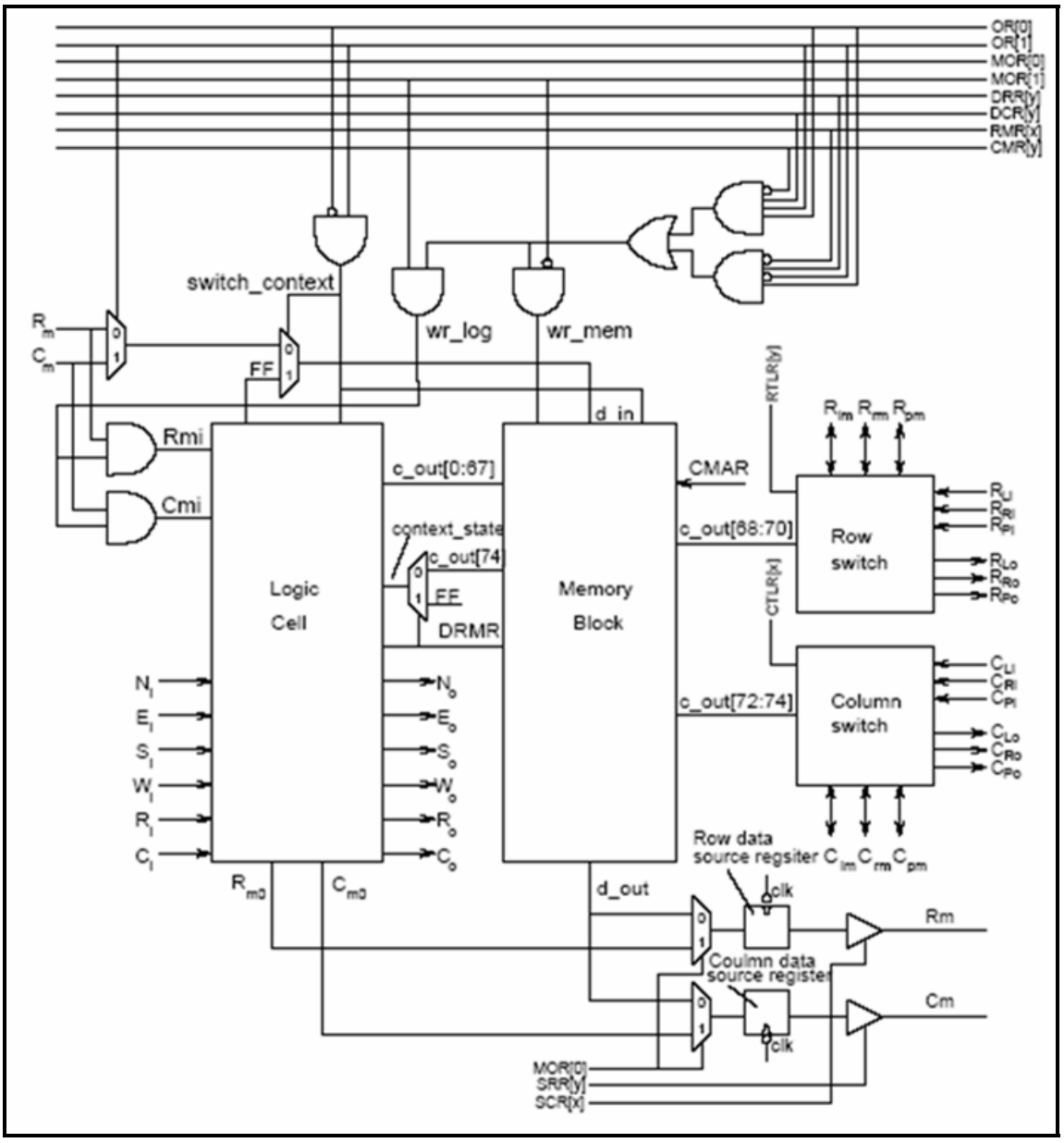


Figure 3.7.1: PE Structure [1]

3.8 2x2 Array

The 2x2 array module shown in figure 3.8.1 consists of four instances of the PE block. Note: Only one column switch is shown for a clearer diagram. The four instances are *u00_pe*, *u01_pe*, *u10_pe*, and *u11_pe*, which are joined together in the *array2x2.v* file. Each PE is connected to one row switch and one column switch. The external connections such as *No00*, *Wi01*, *Rpm*, will become the inputs and outputs to the 2x2 array while the internal interconnects such as *Wo00* will become *Ei01*. The other I/O to the design are the three global registers (OR, MOR, CMAR), and four periphery registers.

3.9 8x8 Array

The 8x8 array block shown in figure 3.9.1 consists of four instances of the 4x4 array. The four instances are *u00_array4x4*, *u01_array4x4*, *u10_array4x4*, and *u11_array4x4*, which are joined together in the *array8x8.v* file. The local interconnects for each of the 4x4 arrays are connected in the same manner that four PEs are connected together to form a 2x2 array. The same way that *Eo* for PE00 connects to the *Wi* for PE01 in a 2x2 array, *Eo* for PE00_0101 connects to the *Wi* for PE01_0000. For more details see the local interconnections in the *array8x8.v* file located in the /srga/tools/rtl/ directory. There are also eight row and eight column switches added to the design. In figure 3.9.1 only switches in rows one and five and columns four and eight are shown for a clearer schematic. These switches are the parent switches in the 8x8 array design. The inputs and outputs of the 8x8 array are connected to the outer PEs, parent switches, three global registers (OR, MOR, CMAR), and four periphery registers.

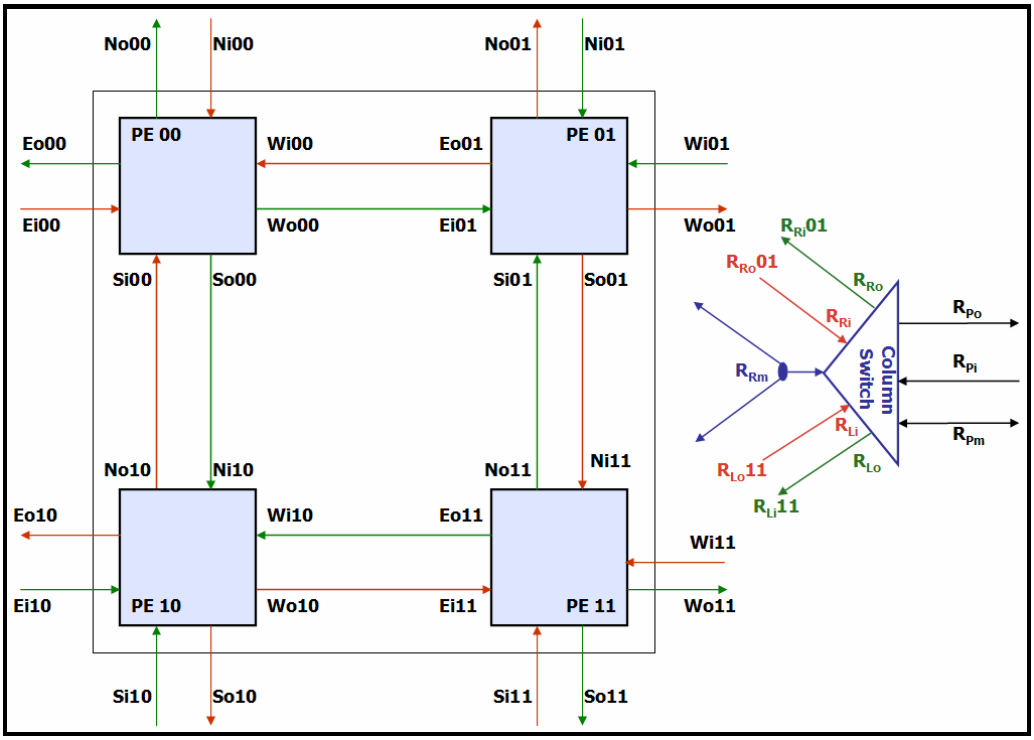


Figure 3.8.1: Structure of the 2x2 Array

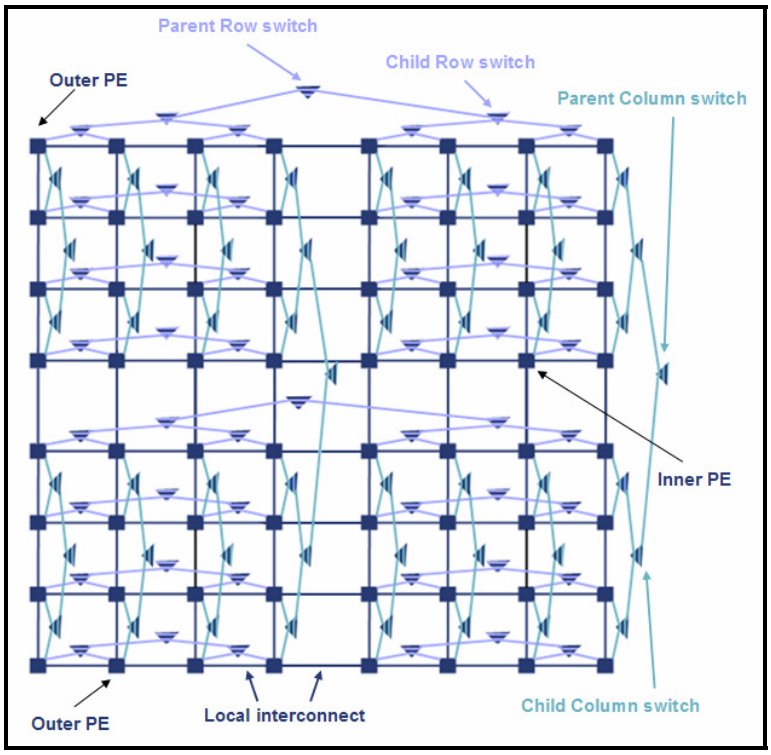


Figure 3.9.1: Structure of the 8x8 Array

Chapter 4: *SRGA-UT* Implementation

4.1 *SRGA-UT* Overview

This chapter provides a step-by-step tutorial of an 8x8 *SRGA-UT* array. The steps will confirm the pre-synthesis RTL verification (using ModelSim), the synthesis process (using Design Compiler) and verification of the netlist (using ModelSim), and the place and route process (using First Encounter) and verification of the delay constraints (using ModelSim).

There are several new RTL designs that were created to simplify the testing of the design. The synchronous D Flip-Flip *dffsync.vhd* file, located in the /tools/rtl/logiccell/ directory, was created to replace the tsmc18 version to eliminate timing errors. The memory array *memArray.vhd* file, located in the /tools/rtl/mem/ directory, was created to replace the *memArray.v* to reduce the number of instances when synthesizing the memory cell.

In chapter 5 the implementation of the *SRGA-UT* sub-designs is described.

4.2 EDA Tools

For the 8x8 array *SRGA-UT* design some of the EDA tools available at the ECE Department at University of Tennessee will be utilized to explore design alternative and enhance productivity. The EDA tools used are ModelSim from Mentor Graphics, Design Compiler from Synopsys, and First Encounter from Cadence.

The ModelSim tools were used for design testing and verification done at the pre-synthesis level (HDL designs), post-synthesis level (gate-level netlist), and post place and route level (SDF timing) [7].

The Design Compiler (dc_shell) was used for logic synthesis, which is the process of converting a design description written in a hardware description language such as Verilog and VHDL into an optimized gate-level netlist targeting the tsmc18 libraries. The mixed compile method was used, where the top-down and bottom-up strategies are simultaneously applied [8]. The top-down compile is the most used strategy where the top-level design and all its sub-designs are compiled together. The bottom-up compile strategy compiles the sub-designs separately and then incorporates them in the top-level design. The top-level constraints are applied, and the design is checked for violations.

The Mixed compile or bottom-up methods must be used to synthesize the *SRGA-UT* design. This is because of the large number of instances present in the memory cell module. By running only the top-down approach for a design of 2x2 array and higher, the available number of files (~32,000) that can be used in a folder or the amount of memory (~4000M) allocated by the internal CPU for each user, will be surpassed and the synthesis will crash. The synthesis scripts can be found in the /synthesis/synopsys/ directory of each module.

First Encounter was used to generate the place and route steps, targeting the TSMC 0.18-micron technology, and extract the SDF file which contains net delays and cell delays [9]. This is done by going through a series of steps, described in section 4.4, where the netlist is transformed into the graphic design system (GDS) format for 2D layout display.

The design flow for implementing the *SRGA-UT* is shown in figure 4.2.1.

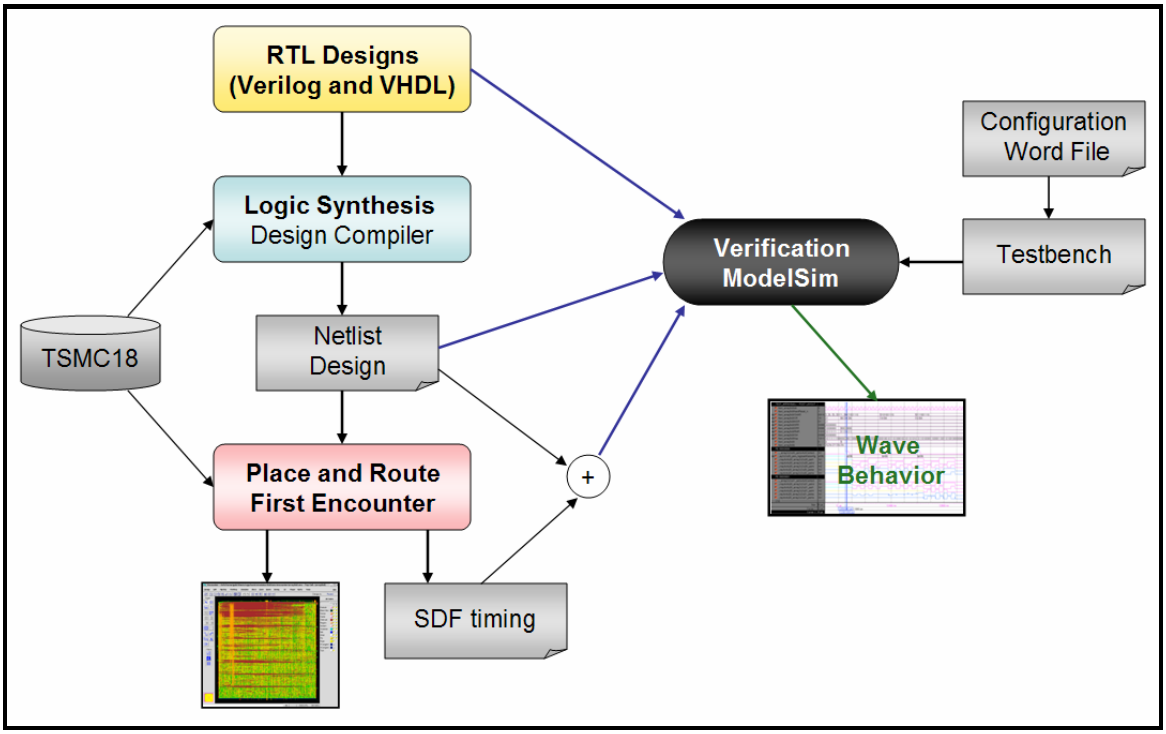


Figure 4.2.1: SRGA-UT Design Flow

4.3 Setting up Files

There are two files needed to setup the *SRGA-UT*: (1) the script ***start-SRGA-UT*** and (2) the zipped file ***srga-ut.tar.gz***. The files can be obtained from Dr. Bouldin at the ECE department of the University of Tennessee. The first step is to copy the two files to the directory where the *SRGA-UT* is going to be initiated. The second step is to run the script *start-SRGA-UT*.

» **start-SRGA-UT**

The script will unzip the *srga-ut.tar.gz* which will setup the folders and files in the following format.

/srga/	main directory
/srga/documentation/USC/	original project and files from USC

/srga/documentation/UT/	*SRGA-UT* documentation and libraries
/srga/config_files/	configuration word files
/srga/tools/rtl/	RTL designs (verilog and vhdl)
/srga/tools/simulator/lib/	tsmc18, synopsys, and encounter library files
/srga/tools/simulator/logiccell/	logic cell testing folders, scripts, and files
/srga/tools/simulator/memcell/	memory cell testing folders, scripts, and files
/srga/tools/simulator/switch/	switch testing folders, scripts, and files
/srga/tools/simulator/PE/	PE testing folders, scripts, and files
/srga/tools/simulator/2x2/	array 2x2 testing folders, scripts, and files
/srga/tools/simulator/8x8/	array 8x8 testing folders, scripts, and files

4.4 8x8 Array Step-by-Step Tutorial

The file for testing the functionality of the 8x8 array block is the verilog file *test_array8x8.v* which is located in the /srga/tools/simulator/8x8/ directory. A number of predefined configuration words are provided by the configuration file *8x8_config_words.cnf* located in the /srga/config_files/ directory. Line number five in figure 4.4.1 is a configuration word taken from the configuration file and can be described as follows:

- x – Eight bits from 0 to 7 are the horizontal periphery bit registers for the 8x8 array. A bit "1" selects the specific column.
- y – Eight bits from 8 to 15 are the vertical periphery bit registers for the 8x8 array. A bit "1" selects the specific row.
- c – Three bits from 17 to 19 are the context field from the CMAR register. The context field selects the memory column to perform a memory access operation.

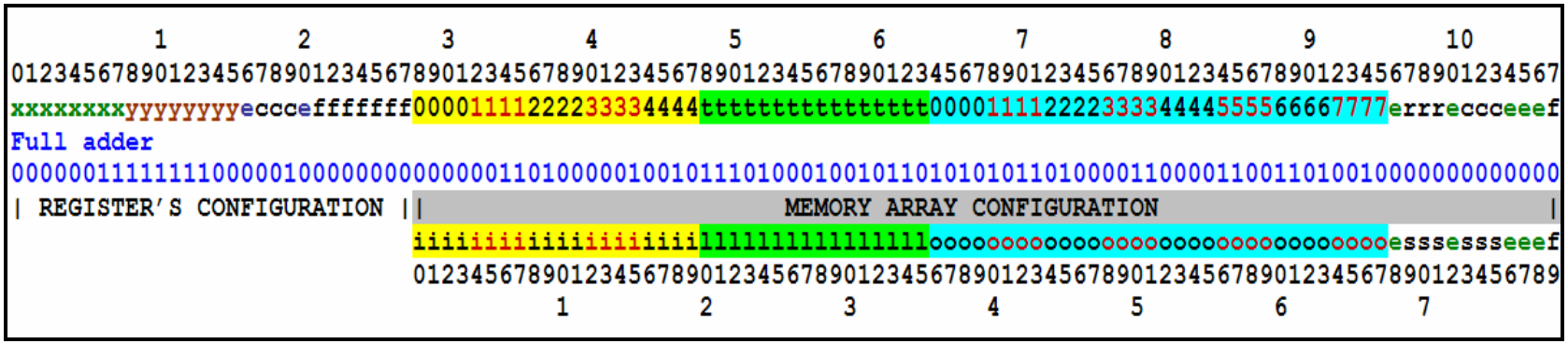


Figure 4.4.1: Configuration Word Format for the 8x8 Array

e – Indicates an extra bit.

f – Seven bits from 21 to 27 are the offset field from the CMAR register. The offset field selects the memory row to perform a memory access operation.

i – The five sets of four bits each from 28 to 47 are the select bits for the input muxes M0i to M4i. The first four sets of four bits from 28 to 43, select the four select bit inputs, L0i to L3i, to the LUT. The fifth set of four bits from 44 to 47, select the first input to the switch_context mux.

t – Sixteen bits from 48 to 63 are the input bits for the LUT. In this case "1110100010010110" in binary or "e896" in hex is the configuration for a full adder.

o – The eight sets of four bits each from 64 to 97 are the select bits for the output muxes M0o to M7o. The outputs from these muxes are No, Eo, So, Wo, Ro, Co, Rmo, and Cmo respectively. The No, Eo, So, and Wo are connected to the neighbor PEs while Ro, Co, Rmo, and Cmo are connected to the logic and memory interconnects of the owned switches.

r – Three bits from 97 to 99 are the configuration bits for the row owned switch.

c – Three bits from 101 to 103 are the configuration bits for the column owned switch.

f – One bit 107. Only bit 0 is used to configure the content of the Flip-Flop.

On line eight of the test-bench *test_array8x8.v* the number of configuration words is defined *(`define CONFIGWORDS 5)*. In this case, the number of *CONFIGWORDS* comes from the configuration file *(8x8_config_words.cnf)*. Figure 4.4.2 is the section of the test-bench that takes the configuration words and distributes the different sections of each word to the 8x8 memory array and registers. Figure 4.4.3 shows the method of assigning the truth table inputs *ni*, *wi*, and *ri*, to perform the full addition and subtraction operations.

```
/*****************Load config.*****************/
$readmemb("../../../../../config_files/8x8_config_words.cnf", mem);
/**********************************************/

       OR<=2'b01;
       MOR<=2'b01;
       SRR<=`array_side'b00000000;
       for(i=0;i<`CONFIGWORDS;i=i+1)
       begin
          tmp=mem[i];
          DRR=`array_side'b00000000;
          //DRR[`array_side-tmp[8:15]-1]=1'b1;  //this will force bit 0 to be 1
          DRR[`array_side-1:0]=tmp[8:15];
          RMR=`array_side'b11111111;
          //RMR[`array_side-tmp[0:7]-1]=1'b0;  //this will force bit 0 to be 0
          RMR[`array_side-1:0]=tmp[0:7];
          CMAR[9:7]=tmp[17:19];
          CMAR[6:0]=tmp[21:27];
          if(tmp[28]!==1'bx)
            begin
              CM[0]<=tmp[28];
              CM[1]<=tmp[28];
              CM[2]<=tmp[28];
              CM[3]<=tmp[28];
              CM[4]<=tmp[28];
              CM[5]<=tmp[28];
              CM[6]<=tmp[28];
              CM[7]<=tmp[28];
              repeat (1) @ (posedge clk);
            end
          for (j=1;j<79;j=j+1)
          begin
             CMAR[6:0]=CMAR[6:0]+1;
             if(tmp[28+j]!==1'bx)
               begin
                  CM[0]<=tmp[28+j];
                  CM[1]<=tmp[28+j];
                  CM[2]<=tmp[28+j];
                  CM[3]<=tmp[28+j];
                  CM[4]<=tmp[28+j];
                  CM[5]<=tmp[28+j];
                  CM[6]<=tmp[28+j];
                  CM[7]<=tmp[28+j];
                  repeat (1) @ (posedge clk);
               end
          end
       end // for (i=0;i<`CONFIGWORDS;i=i+1)
```

Figure 4.4.2: Importing Configuration Word Section of *test_array8x8.v*

```
//-------------------------------------------------------------------------------------------
//     FULL SUBTRACTOR

       CMAR[9:7]<=3'b010; // memory column "010" has LUT config for full adder
       repeat (1) @ (posedge clk);

       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b0;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b0;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b0;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b0;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b1;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b1;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b1;
       repeat (1) @ (posedge clk);
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b1;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b1;

//-------------------------------------------------------------------------------------------
//     FULL ADDER - TESTING PE00000000  AND PE00000001

       CMAR[9:7]<=3'b011; // memory column "011" has LUT config for full adder
       repeat (1) @ (posedge clk);

       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.wi=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u00_pe.u_logiccell.ri=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u01_pe.u_logiccell.ni=1'b0;
       force test_array8x8.u_array8x8.u00_array4x4.u00_array2x2.u01_pe.u_logiccell.ri=1'b0;
       repeat (1) @ (posedge clk);
```

Figure 4.4.3: Assigning Inputs to the 8x8 Array

Line seventy-seven stores the configuration words into a temporary system memory. The OR, MOR, and SRR registers are then setup to allow memory write. The FOR loop creates a counter for the number of configuration words and a counter for the amount of bits in a configuration word.

Array 8x8 Pre-Synthesis

To test the RTL design using ModelSim simulation, run the script *1-presynth-8x8* by following the next steps:

- **» cd /simulator/8x8/presynth/**
- **» 1-presynth-8x8**

This will bring up the ModelSim main and wave windows. After reviewing the *test_array8x8.v* signals in the wave window, other signals such as the *array8x8* signals can be added to the wave from the main window by highlighting the *u_array8x8* in the Workspace area, then right click and Add to Wave.

When the signals appear in the wave window, at the main window prompt type ***restart***, then click Restart on the next window. To simulate all the signals, at the main window prompt type ***run 15000000.*** To load the signals from figure 4.4.4, open *presynthesis_8x8_load_TB.do* from the ModelSim wave window, located in /8x8/presynth/modelsim/ directory.

Bits "1" in the DRR registers and bits "0" in the RMR register, indicate the array 8x8 rows that are allowed to have a memory operation. Section A of figure 4.4.4 illustrates the first configuration word being loaded into the memory of the PEs allowed by DRR and RMR. The first configuration word is no operation (all bits are zeros) and it is stored in column address "000" of the memory array block shown by the *contextAdr* signal.

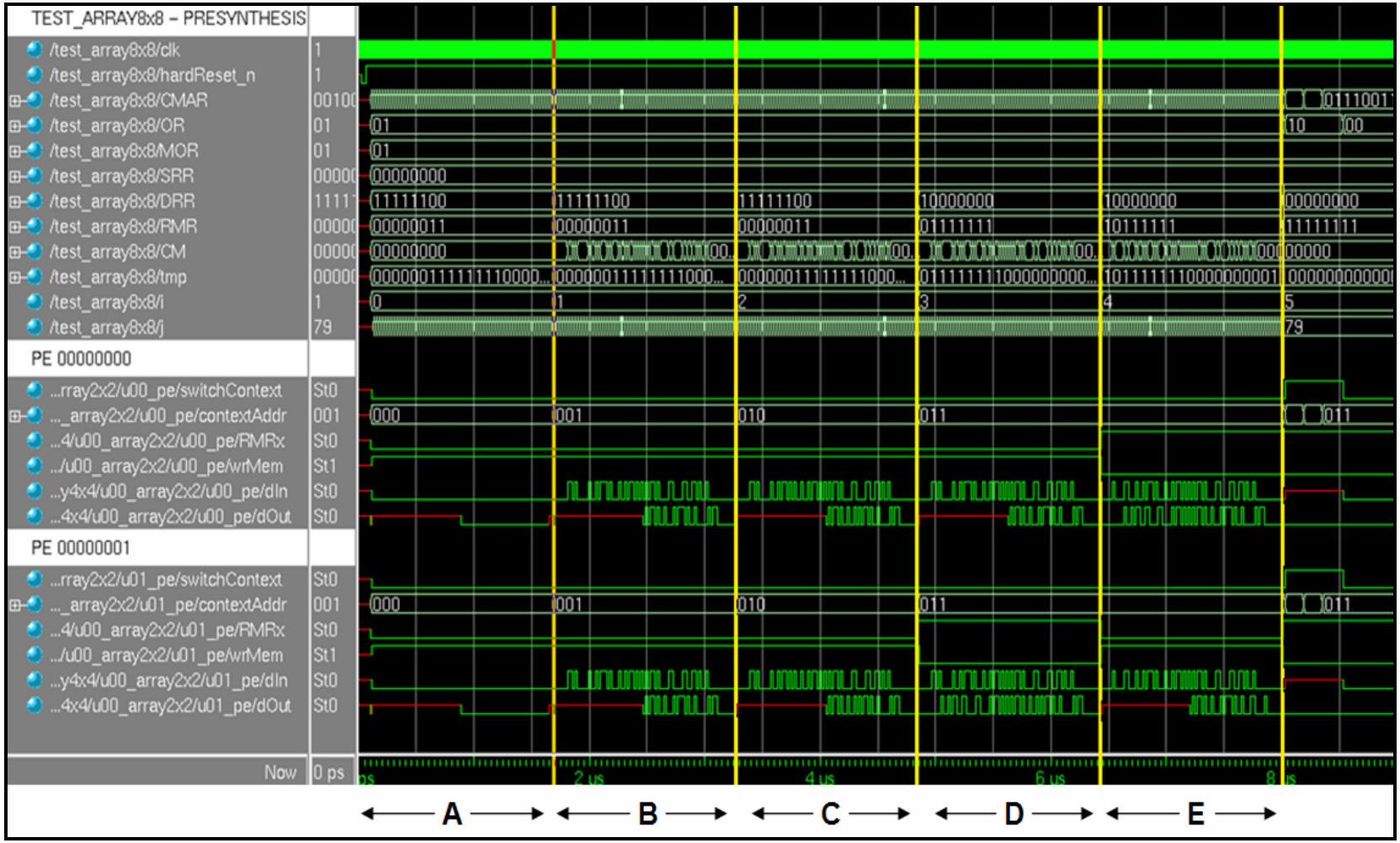


Figure 4.4.4: Array 8x8 Pre-Synthesis - Loading Memory Blocks

Section B and C illustrate the second and third configuration words being loaded in the same manner as the first word. The second word is configured for a full adder and stored in column address "001" of the memory array block. The test behavior is shown in figure 4.4.6 section X with *ni*, *wi*, and *ri* being the truth table inputs, while the orange signals *no* and *eo* being the sum and carry-out respectively. The third word is configured for a full subtract operation and it is stored in address "010". The behavior of the waves can be examined in section Y of figure 4.4.6 with the same inputs and outputs as part X where and *eo* being the difference and borrow-out respectively.

Sections D and E of figure 4.4.4 illustrate the fourth and fifth configuration words being loaded. This is a unique test case since PE0 and PE1 in the first row of the 8x8 array are configured to perform two-bit full adder also illustrated in figure 4.4.5. The fourth word is loaded only into the memory address "011" of PE0 which is controlled by the *DRR*, *RMR* and *wrMem* signals. The configuration word selects the *Ni* and *Wi* of PE0 to be the addition bits and *Ri* to be the Carry-In bit, while *No* is the Sum and *Eo* is the Carry-Out. The fifth word is loaded only into the memory address "011" of PE1. The configuration word selects the *Ni* and *Ri* to be the addition bits and *Wi* becomes the Carry-In bit which is connected to the *Eo* Carry-Out of PE0. *No* is the Sum bit and *Eo* is the Carry-Out bit which will connect to the west input of PE2. Note: *ni* (from code) and *Ni* (from schematic) are identical signals; and the same applies to all other signals.

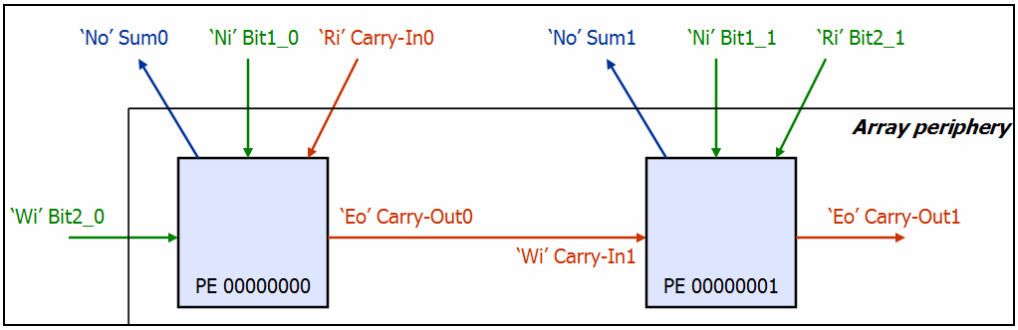


Figure 4.4.5: Array 8x8 Full Add Demo Schematic

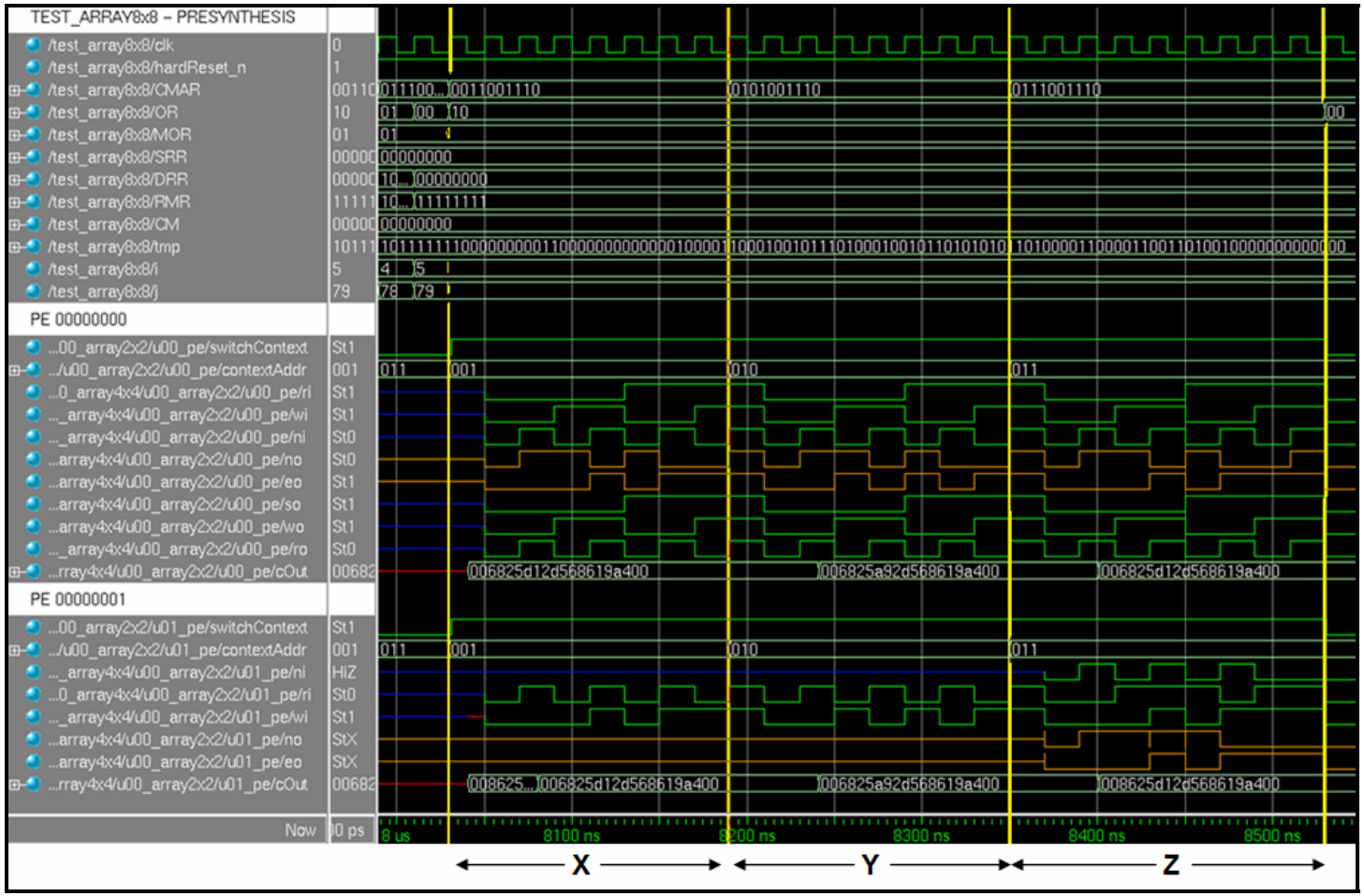


Figure 4.4.6: Array 8x8 Pre-Synthesis - Applications

To load the signals from figure 4.4.6, open *presynthesis_8x8_application.do* from /8x8/presynth/modelsim/ directory. When the *contextAddr* changes, in this case from full add to subtract and back to add, it takes one clock cycle for the device to switch to each context.

Figure 4.4.7 shows the filenames and their full-path, and the instances and their design modules for the array 8x8 design.

Array 8x8 Synthesis

To create the gate-level netlist design, follow the next steps:

- **» cd /simulator/8x8/synthesis/**
- **» 2-synth-array8x8**

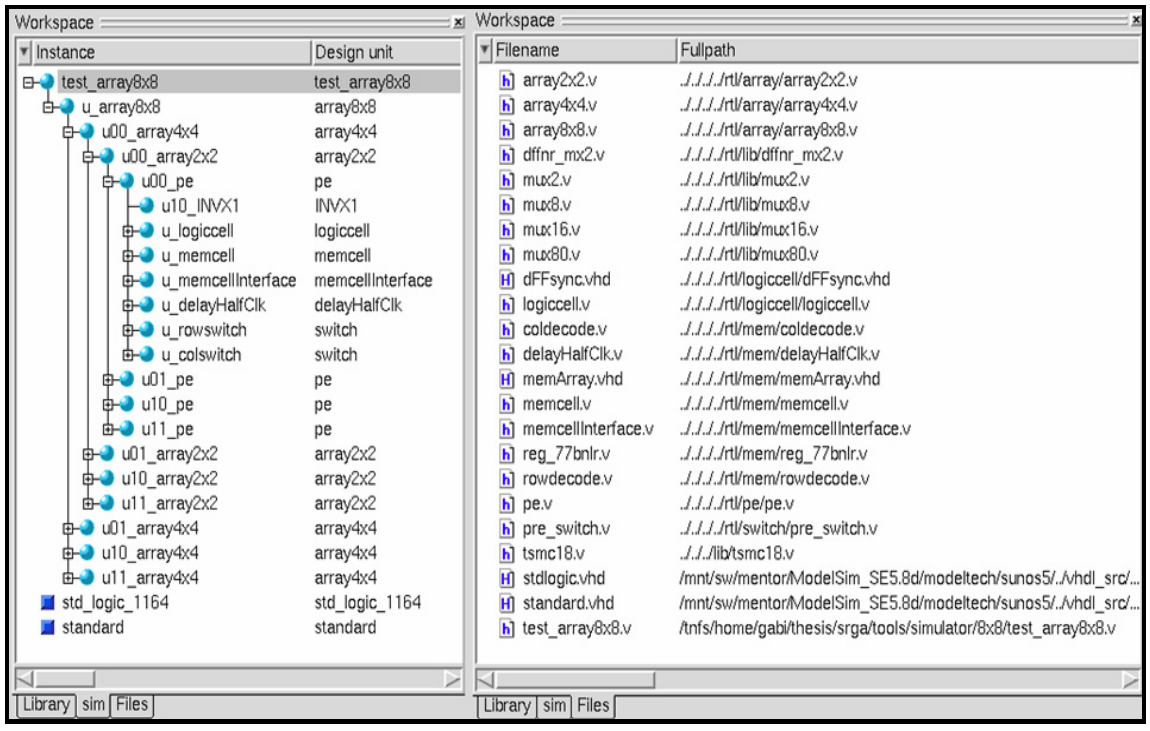


Figure 4.4.7: Array 8x8 Pre Synthesis Workspace

At first, the script will copy the necessary library files to the /8x8/synthesis/synopsys/ directory and then the dc_shell synthesis tool from Synopsys [8] is initiated. The dc_shell executes the script *synth-array8x8.scr* located in the /synopsys/ directory. The details from this step are written to the file *info_8x8_synthesis.txt* also located in the /synopsys/ directory. The synthesis tools will create the verilog netlist file ***array8x8-synth.v*** and the delay file ***array8x8-synth.sdf*** which are written in the /synopsys/ folder.

The next step is to edit the verilog netlist file. At lines 407 change from *rpm00(1'b0)* to *rpm00(rpm0000)* and at line 408 change from *rpm10(1'b0)* to *rpm00(rpm0010)*.

The netlist is used to test the post synthesis design using ModelSim simulation by following the next steps:

- **cd /simulator/8x8/synthesis/**
- **3-post-synth-sim-array8x8**

This will bring up the ModelSim main and wave windows. To test and review the behavior of the signals, follow some of the steps described in the pre-synthesis section. To load the signals from figure 4.4.8 and 4.4.9, open the wave file *synthesis_8x8_final.do* from the /8x8/synthesis/modelsim/ directory. To review the behavior of the waves, follow the instructions in the pre-synthesis simulations.

Note: It is a good idea to go into the /modelsim/ folders after testing the design and delete the work/ directory and the transcript, vsim.wlf, and workingExclude.cov files. This will save space to be able to run other tests.

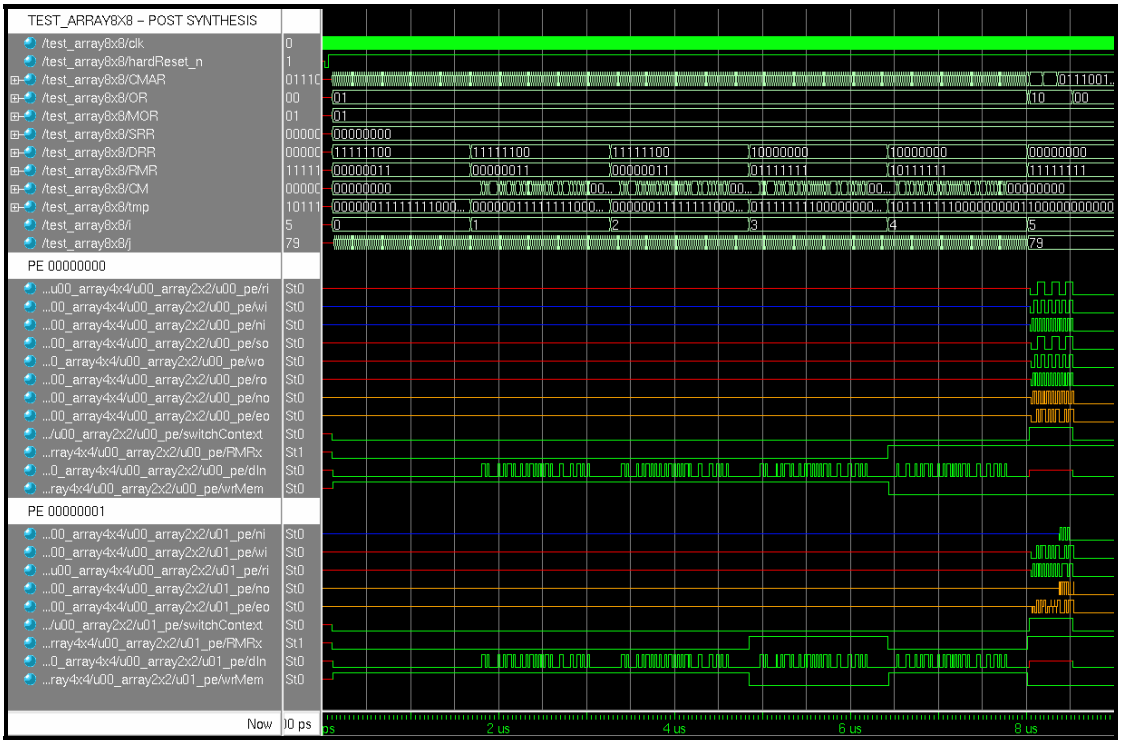


Figure 4.4.8: Array 8x8 Synthesis - Loading Memory Blocks

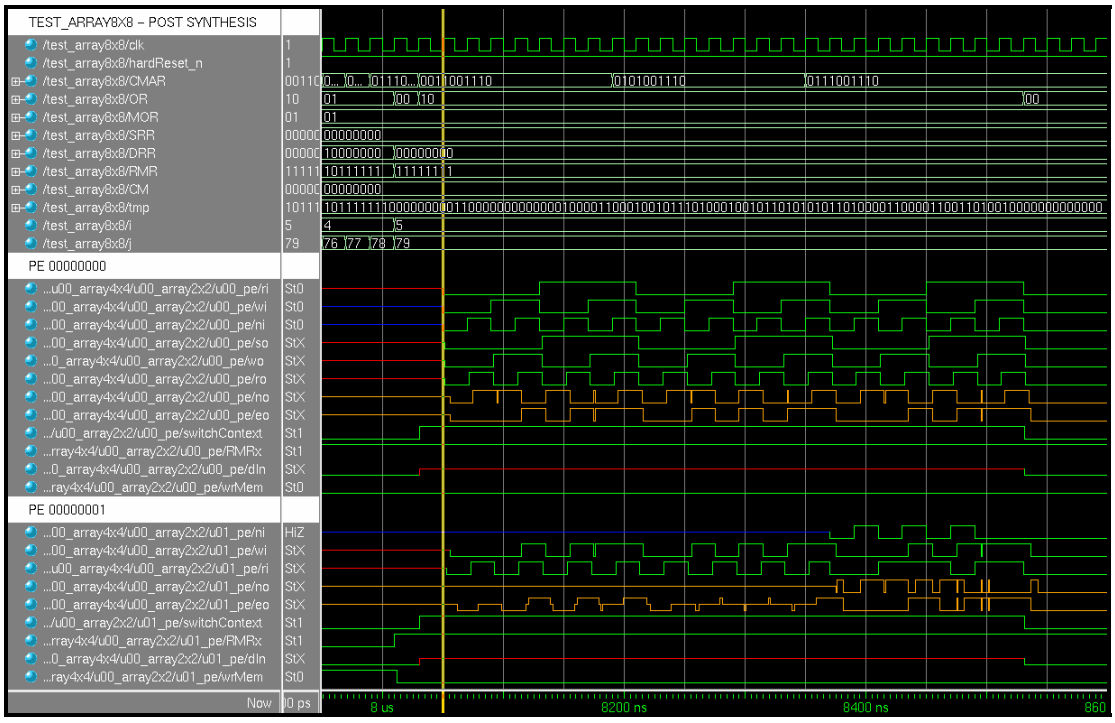


Figure 4.4.9: Array 8x8 Pre-Synthesis - Applications

First Encounter Tools

This Section is the tutorial to generate the automatic place and route of the design. The process itself is very elaborate and to discuss the details of developing each step is out of the scope of this project. However a generalized step by step tutorial of using the encounter tools to crate the place and route is provided bellow.

Step 1: Setting up the files

Initiate the Encounter tools:

- **» cd /simulator/8x8/asic/**
- **» 4-start-encounter**

The script *4-start-encounter* will copy the necessary encounter libraries to the /8x8/asic/encounter/ directory and will bring up the encounter window.

Step 2: Import the Design

From the encounter window (see figure 4.4.11), select **Design** -> **Design Import...** Complete the **Design** and **Power** sections from the Design Import window. In the Design section, to choose the files click on the dotted tab to the right of the text boxes. For the Top Cell you can check the Auto Assign or you can enter your own name. In the Power section, name the Power Nets as VDD and Ground Nets as VSS. At this point you can save the initial setup. Click **OK**

After the design is imported, the core area of the chip should be seen and if zoomed out (shift-z), the top level module (four instances of the 4x4 array) should be seen in purple objects. A single module can be selected by single clicking the object. By using "shift-g" the object can be ungroup and all the modules that belong to the object should be seen. You can use the "shift-g" procedure further to another layer of hierarchy.

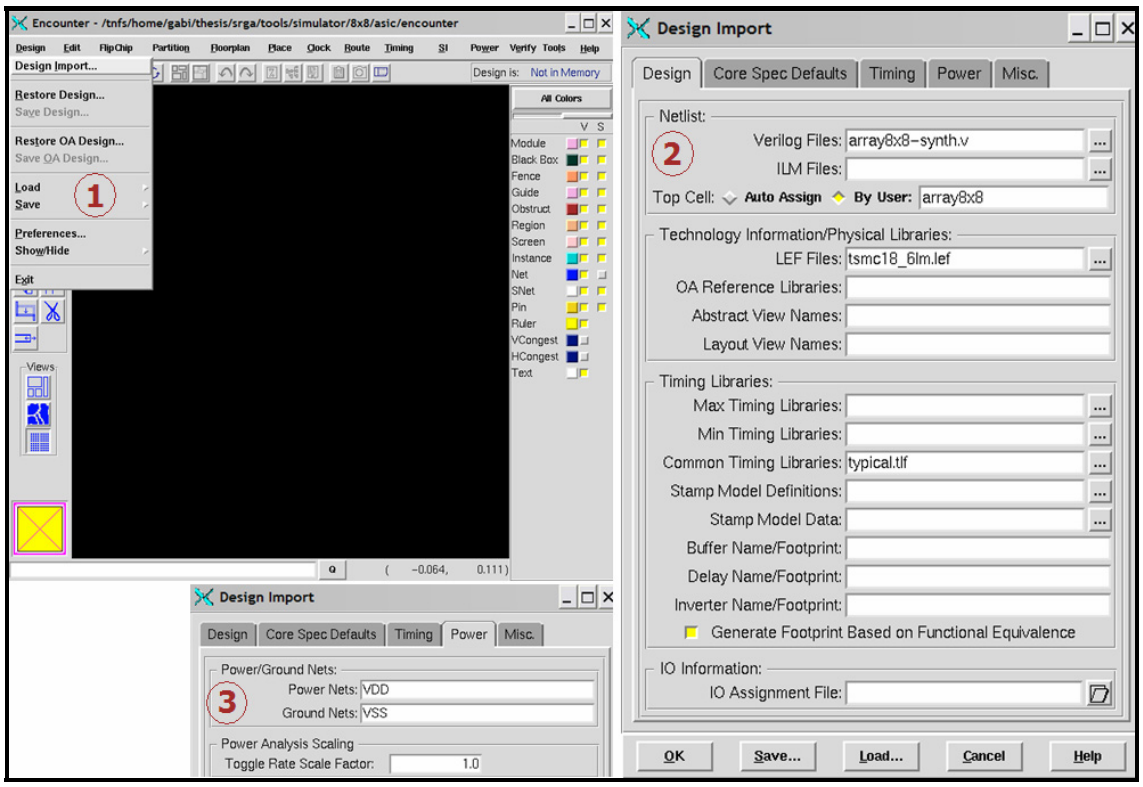


Figure 4.4.11: Encounter Tools - Importing Design

To re-group the hierarchy, select one of the child modules and press "g". The refresh button can redraw the design at any time.

Step 3: Specify the Chip Size

From the encounter window (see figure 4.4.12), select **Floorplan** -> **Specify Floorplan...** Change the Margins "Core to IO Boundary" to 40 for all directions. Click **OK** to apply the change.

Step 4: Power Planning

From the encounter window (see figure 4.4.13), select **Floorplan** -> **Power Planning** -> **Add Rings...** The width and the spacing for the rings are an option that could be changed. Everything else stays as default. Click **OK.**

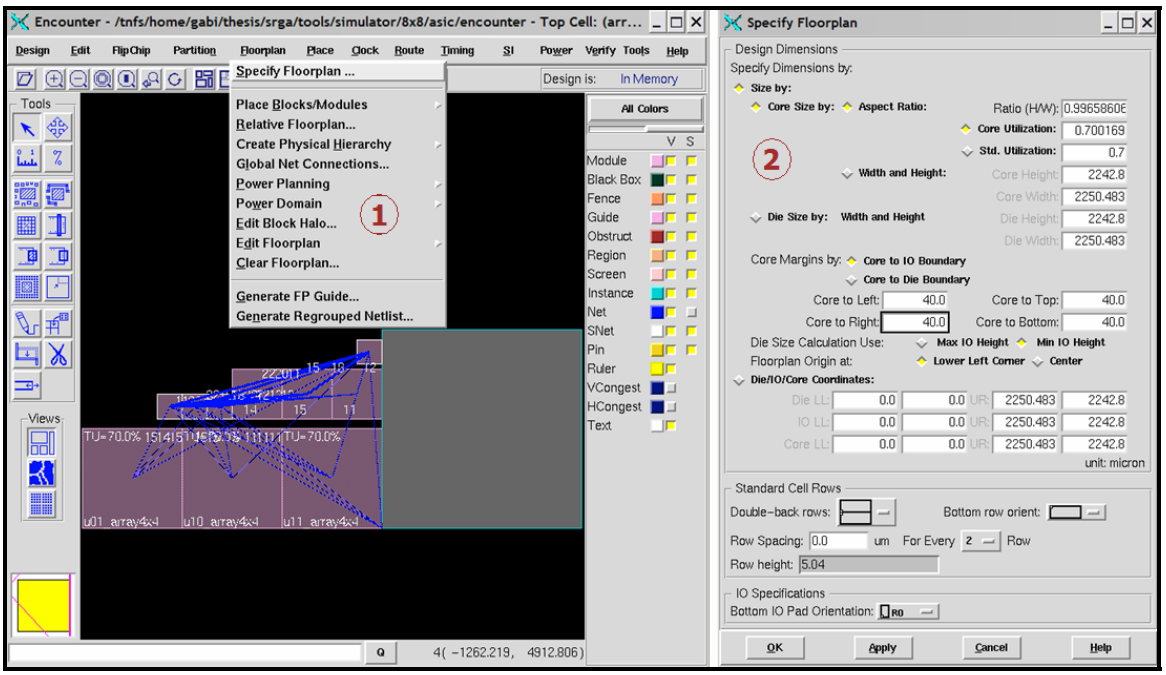


Figure 4.4.12: Encounter Tools - Specify Chip Size

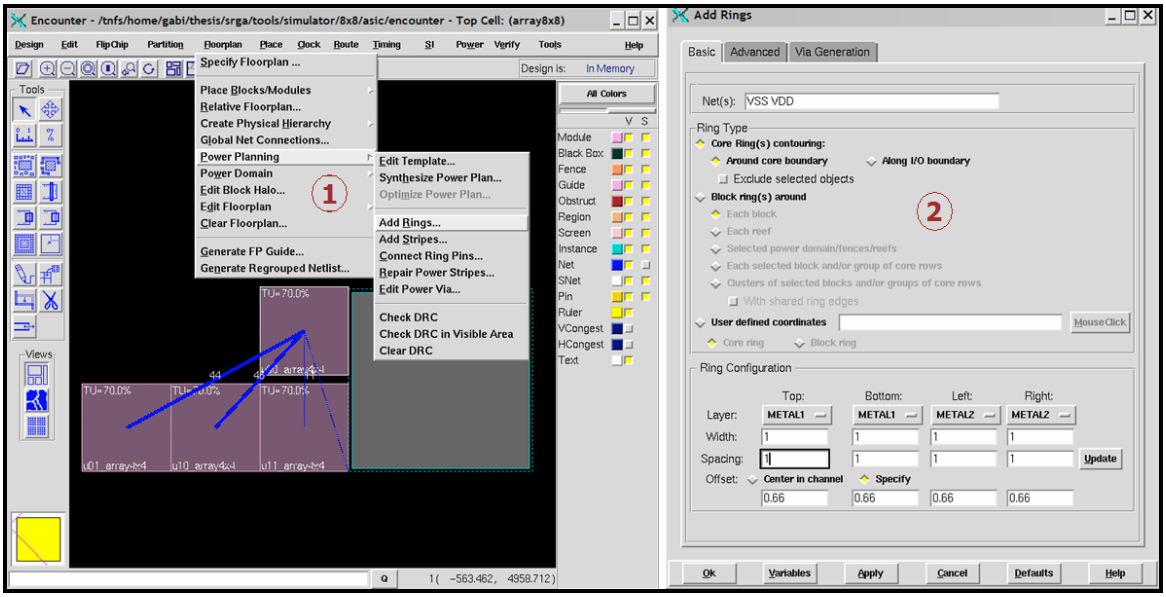


Figure 4.4.13: Encounter Tools - Power Planning

Step 5: Global Net Connections

From the encounter window (see figure 4.4.14), select **Floorplan** -> **Global Net Connections...** This connects the VDD and VSS pins to the global power nets. Fill out the form of part 2 and click **Add to List**. After the pins VDD and VSS are seen in the Connection List, click **Apply** and then **Close**.

Step 6: Standard Cell Placement

From the encounter window (see figure 4.4.15), select **Place** -> **Place...**, use the default of Medium Effort and click **OK**. This step takes some time (~8min).

At this point the three different placement views can be seen from figure 4.4.16. The first view is the Floorplan view. The blue lines show the connections between the different modules and the connections to the I/O pins of the floorplan. The second view is the Amoeba view. This shows the outlines of the different modules. Any module can be selected and use "shift-g" to view the next layer hierarchy. The third view is the Physical view. This is where the standard cells can be seen by zooming.

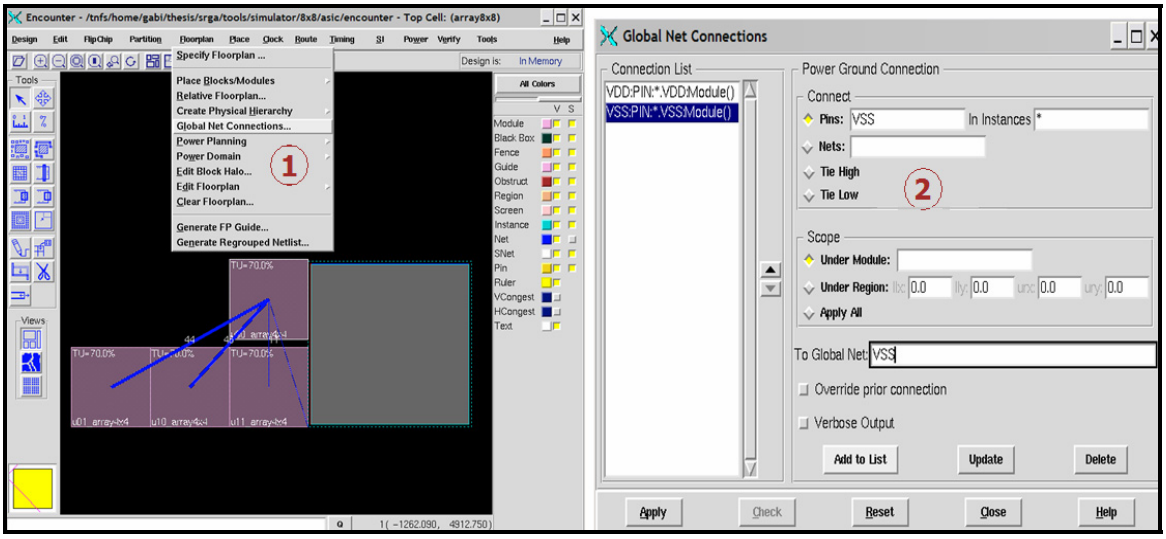


Figure 4.4.14: Encounter Tools - Global Net Connections

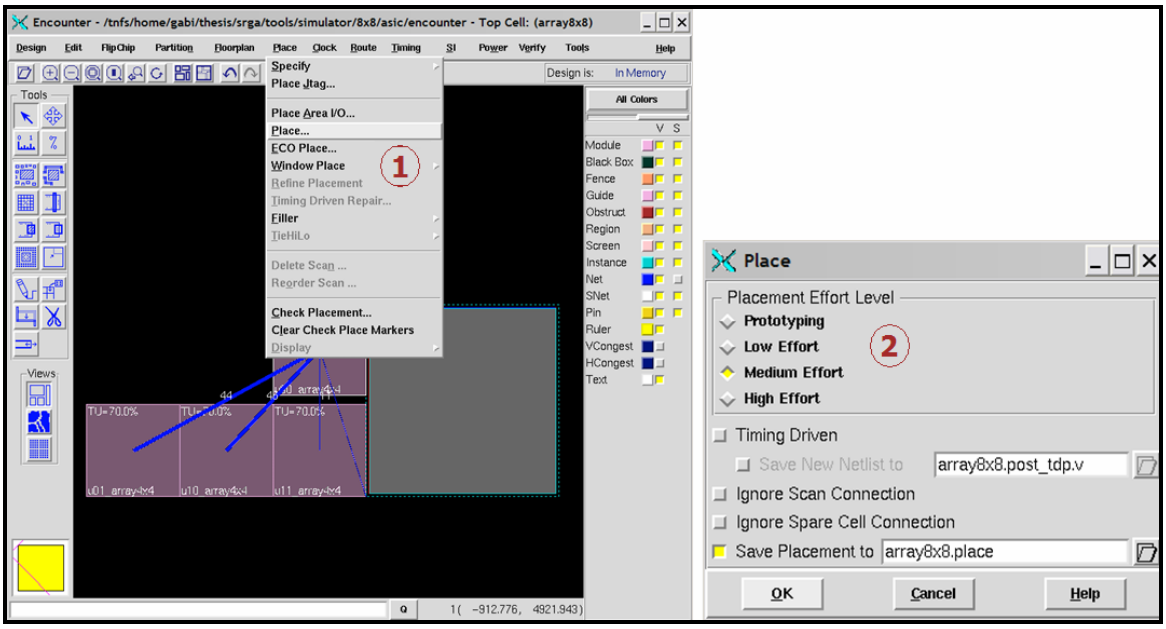


Figure 4.4.15: Encounter Tools - Standard Cell Placement

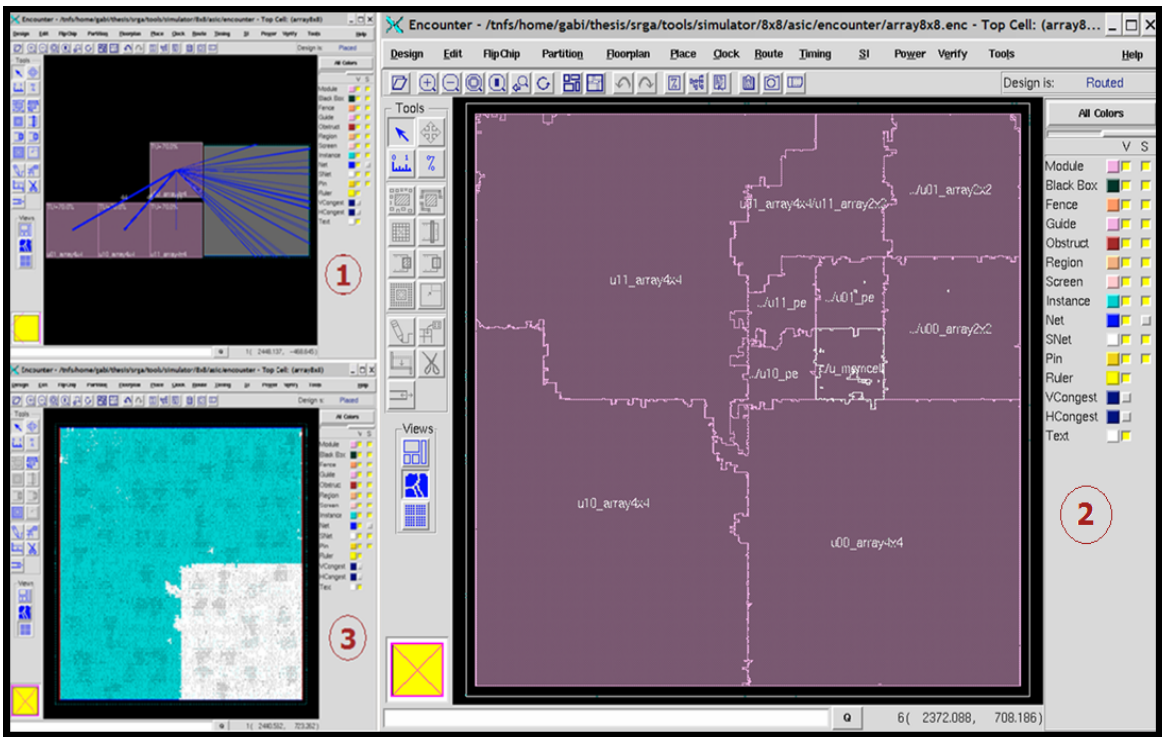


Figure 4.4.16: Encounter Tools - Cell Placement Views

Step 7: Add Filler Cells

From the encounter window (see figure 4.4.17), select **Place** -> **Filler** -> **Add Filler...** In the Add Filler window (part 2) change the Cell Name to **FILL1**, and the Prefix to **fill**. Click **OK**.

Step 8: Route Power

From the encounter window (see figure 4.4.18), select **Route** -> **SRoute...** When the SRoute window comes up, uncheck **Block pins**, **Pad pins**, and **Pad rings**. Click **OK**. The power strips should be seen in the layout.

Step 9: Final Route

From the encounter window (see figure 4.4.19), select **Route** -> **WRoute...** Leave everything as default. Click **OK**. This part may take some time (~20 to 30min).

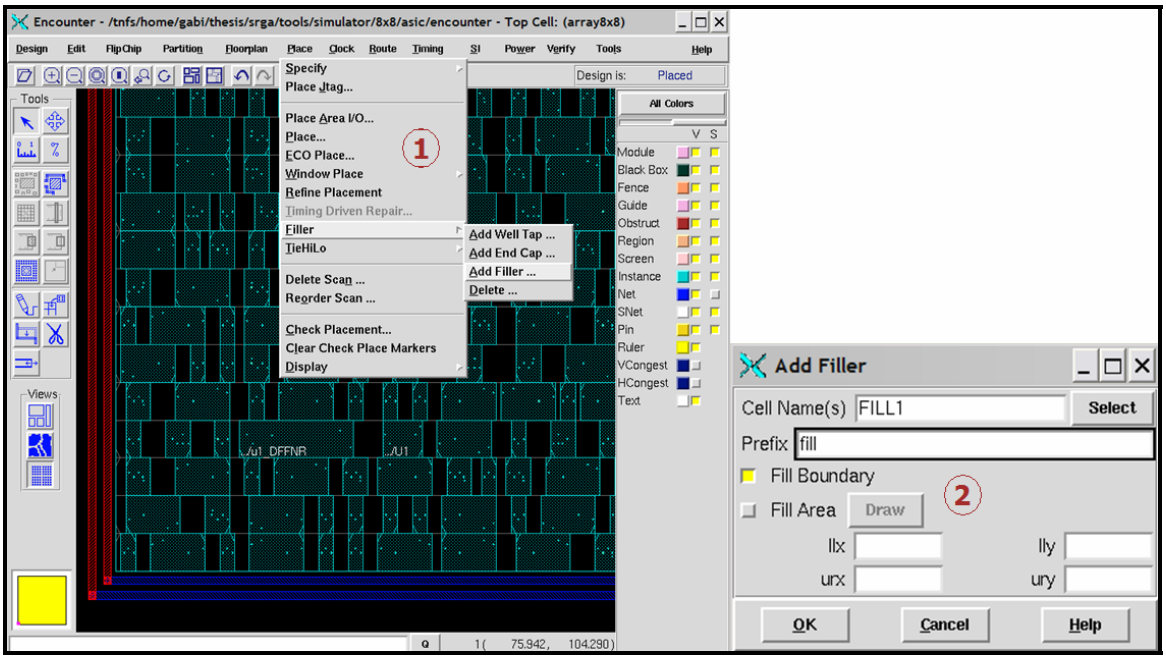


Figure 4.4.17: Encounter Tools - Add Filler Cells

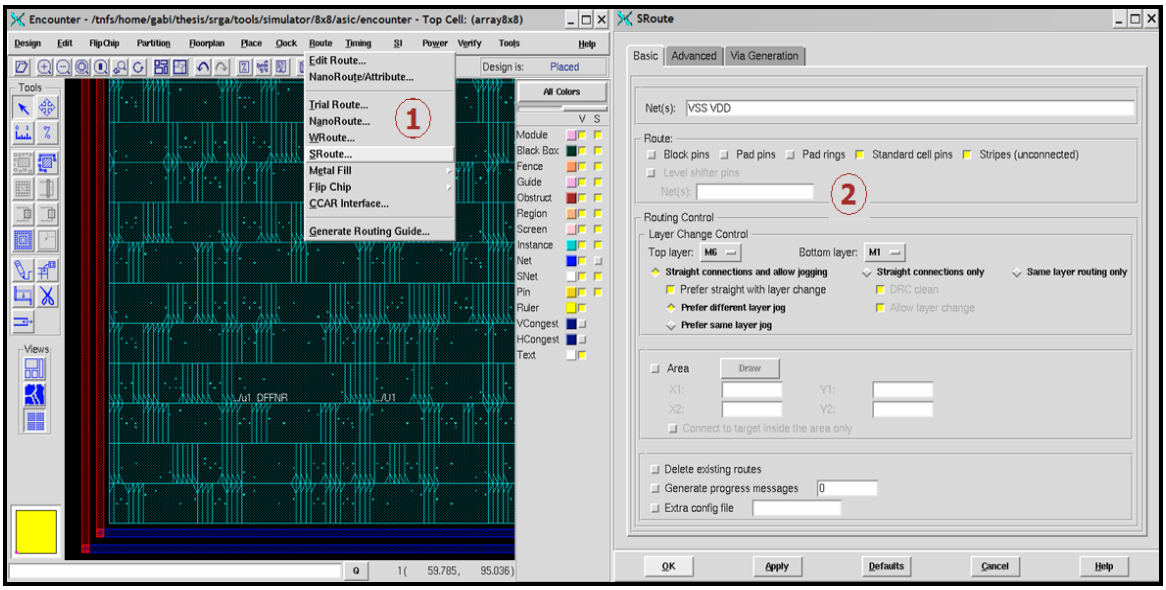


Figure 4.4.18: Encounter Tools - Route Power

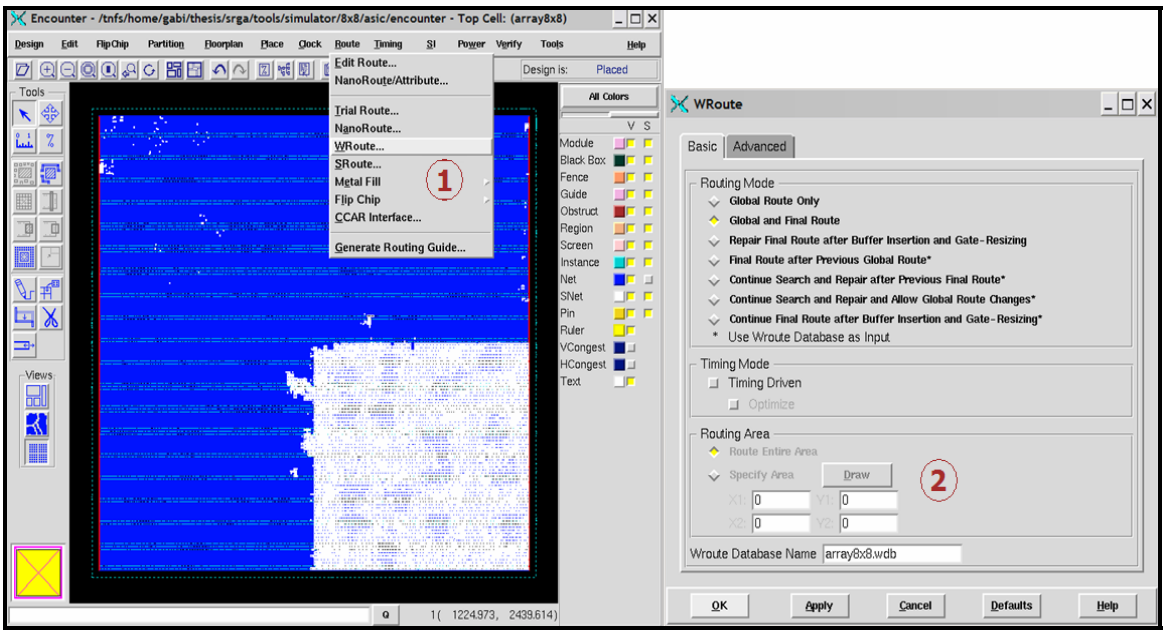


Figure 4.4.19: Encounter Tools - Final Route

Step 10: Extract RC

From the encounter window (see figure 4.4.20 1a and 1b), select **Timing** -> **Extract RC…** You can check any of the boxes to save any file then click **OK**.

Step 11: Calculate Delay

From the encounter window (see figure 4.4.20 2a and 2b), select **Timing** -> **Calculate Delay…** In the Calculate Delay window change the name for the SDF Output File to ***array8x8-encounter.sdf***. Click **OK**. This file will be stored in the /8x8/asic/encounter/ directory and it is the final file that includes the timing delay for the place and route design.

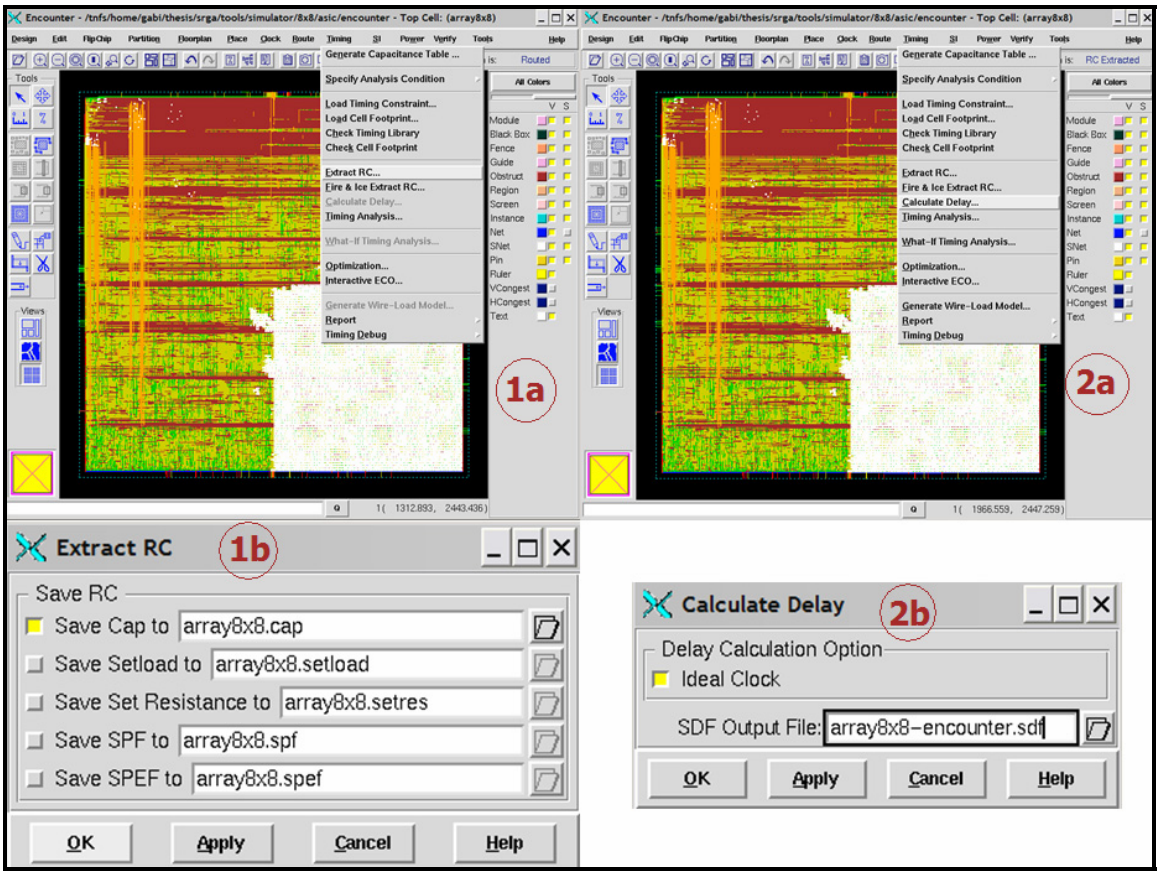


Figure 4.4.20: Encounter Tools - (1) Extract RC and (2) Calculate Delay

Step 12: Results, Save, and Restore Design

This step is optional although it will save a lot of time. From the encounter window select **Tools** -> **Gate Count Report...** – and – **Summary Report...** Save the reports and view the results. The gate count report will provide the number of gates, number of cells, and the area of the gate for each module. The summary report will provide design statistics, chip utilization, module information, and wire information. The chip utilization will indicate the core size, the chip size, and the number of cell rows.

From the encounter window select **Design** -> **Save Design...** This will save the design by default as *array8x8.enc*. To load the place and route design in the future, select **Design** -> **Restore Design...** and select *array8x8.enc*.

Final Layouts

The final layouts can be seen in figure 4.4.21. The floorplan view shows the individual blocks connected by the blue lines to the left and the core to place all the individual blocks to the right. The Amoeba view shows the individual modules placed in the core. Here you can see the four instances of the 4x4 array, where instance u01_array4x4 further shows the different modules. The Physical view shows the final layout with the place and route.

Array 8x8 Post Layout Simulation

Before using the ModelSim tools to test the place and route timing behavior, the timing file *array8x8-encounter.sdf* must be edited on line 15 from (CELLTYPE " ") to (CELLTYPE "array8x8"). One simple way to open the file is to rename it with a *.txt* extension.

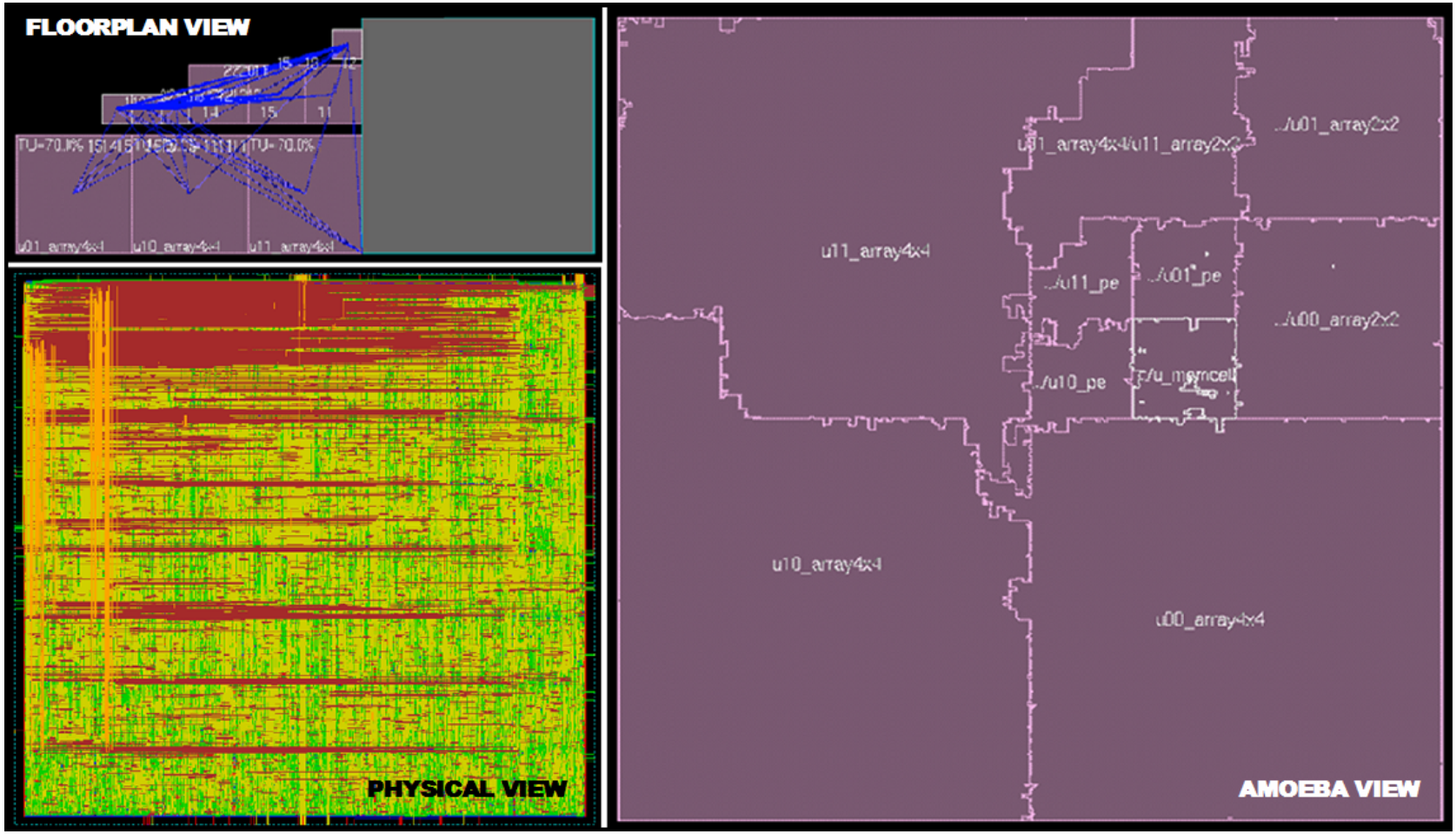


Figure 4.4.21: Encounter Tools - Final Pace and Route of 8x8 Array

To test the timing generated by the pace and route, follow the next steps to initiate the ModelSim tools:

- **» cd /simulator/8x8/asic/**
- **» 5-post-layout-sim-array8x8**

The script *5-post-layout-sim-array8x8* copies the timing file *array8x8-encounter.sdf* from the /encounter/ to the /modelsim/ directory and simulates the netlist design *array8x8-synth.v* created by the synthesis tools. To test and review the behavior of the signals, follow some of the steps described in the pre-synthesis section.

To load the signals in figure 4.4.22 and 4.4.23, from the wave window, open the wave file *post-layout-array8x8_final.do*.

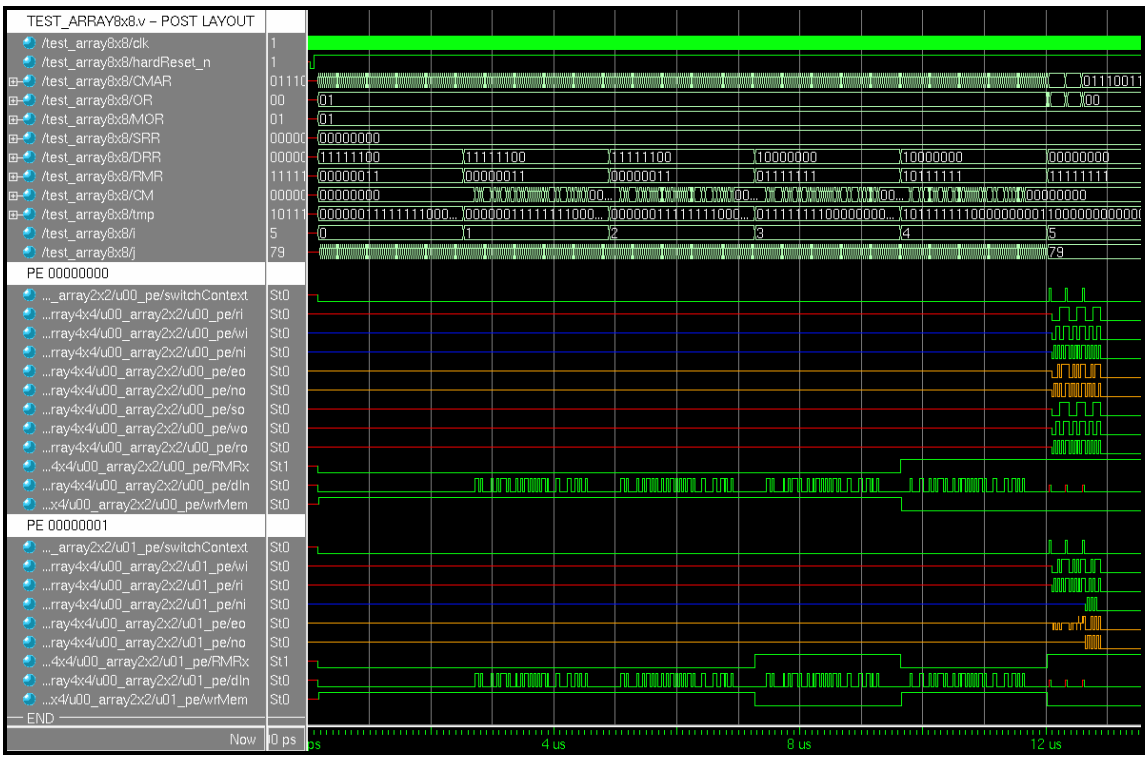


Figure 4.4.22: Array 8x8 Post Layout - Loading Memory Blocks

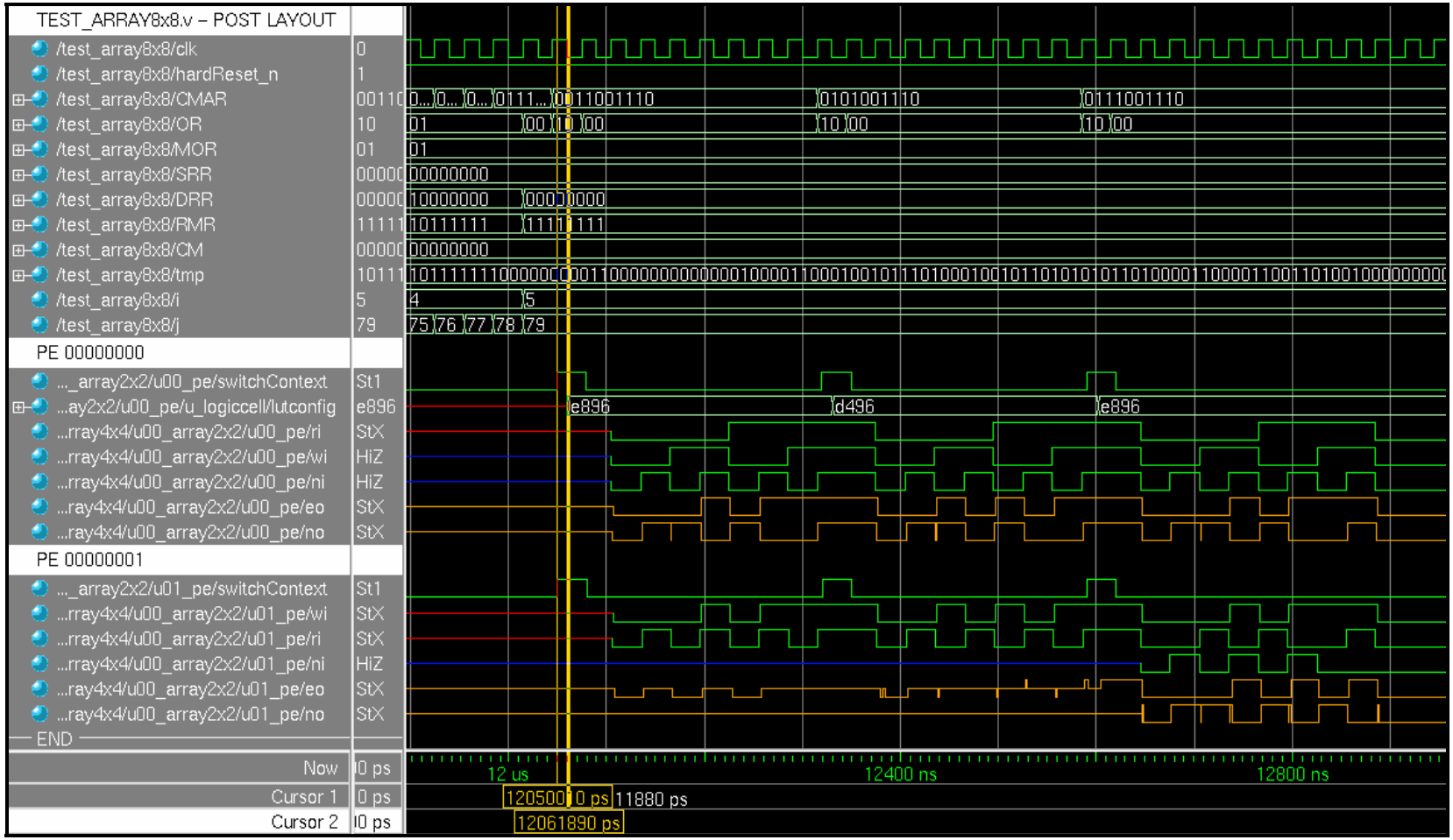


Figure 4.4.23: Array 8x8 Post Layout – Applications

Chapter 5: Implementing the *SRGA-UT* Sub-Designs

5.1 Logic Cell Implementation

The file for testing the functionality of the logic block is *test_logiccell.v* and it is located in the /srga/tools/simulator/logiccell/ directory. The testing objective is to assign inputs to the logic block as if they were coming through the corresponding PE and view the behavior of the internal logic. The applications to be tested are the full addition and full subtraction discussed in section 3.3. By knowing the location of the input bits from the *inMuxBus[15:0]* and *outMuxBus[15:0]*, the configuration bits are selected strategically (see also figure 5.1.1).

Sel_mi0 will select *ni* to be the first input *L0i* for the LUT. *Sel_mi1* will select *wi* to be the second input *L1i* for the LUT. *Sel_mi2* will select *ri* to be the third input *L2i* for the LUT.

At line 120 of the *test_logiccell.v*, the configuration word for a full adder is passed to the *lutconifig* which will be the application for the truth table inputs in lines 128 to 135. The outputs for the full adder are given by *sel_mo0* which selects *Lo0* (sum) to be the output for M0o mux, and *sel_mo1* which selects *Lo1* (carry-out) to be the output for M1o mux. These outputs will be the North output *No* and East output *Eo* for the corresponding PE.

At line 141, the configuration context is switched to full subtraction. The inputs and outputs for the full subtraction operation are utilized in the same manner as in the addition part. In this case *ni* becomes input bit one, *wi* becomes input bit two, *ri* becomes the borrow-in, *No* becomes *Lo0* the difference bit, and *Eo* becomes *Lo1* the borrow-out bit.

```
101       sel_mi0<=NI; //setect M0i to be input NI - this is input L0i to LUT
102       sel_mi1<=WI; //setect M1i to be input WI - this is input L1i to LUT
103       sel_mi2<=RI; //setect M2i to be input RI - this is input L2i to LUT
104       sel_mi3<=EI; //setect M3i to be input EI
105       sel_mi4<=SI; //setect M4i to be input SI
106
107       sel_mo0<=LO0; //select the output of M0o for No to be LUT output Lo0
108       sel_mo1<=LO1; //select the output of M1o for Eo to be LUT output Lo1
109       sel_mo2<=RI; //setect the output of M2o for So to be input RI
110       sel_mo3<=WI; //setect the output of M3O for Wo to be input WI
111       sel_mo4<=NI; //setect the output of M4o for Ro to be input NI
112       sel_mo5<=LO2; //setect the output of M5o for Co to be LUT output Lo2
113       sel_mo6<=LO3; //setect the output of M6o for Rmo to be FF output lo3
114       sel_mo7<=SI; //setect the output of M7o for Cmo to be input SI
115
116
117 //-------------------------------------------------------------
118 //    configuration word of LUT for Full Adder hex: e896
119 //
120       lutconfig<=16'b1110100010010110;
121
122 //    Inputs of the truth table for Full Adder
123 //    ni - is the input bit L0i
124 //    wi - is the input bit L1i
125 //    ri - is the Carry-In bit L2i
126
127
128       ni<=0; wi<=0; ri<=0; repeat (1) @ (posedge clk) ;
129       ni<=1; wi<=0; ri<=0; repeat (1) @ (posedge clk) ;
130       ni<=0; wi<=1; ri<=0; repeat (1) @ (posedge clk) ;
131       ni<=1; wi<=1; ri<=0; repeat (1) @ (posedge clk) ;
132       ni<=0; wi<=0; ri<=1; repeat (1) @ (posedge clk) ;
133       ni<=1; wi<=0; ri<=1; repeat (1) @ (posedge clk) ;
134       ni<=0; wi<=1; ri<=1; repeat (1) @ (posedge clk) ;
135       ni<=1; wi<=1; ri<=1; repeat (1) @ (posedge clk) ;
136
137
138 //-------------------------------------------------------------
139 //    configuration word of LUT for full subtractor hex: d496
140 //
141       lutconfig<=16'b1101010010010110;
```

Figure 5.1.1: Section of Test-bench for Logic Cell Block

Logic Cell Pre-Synthesis

To test the RTL design using ModelSim simulation, run the script *1-presynth-logiccell* by following the next steps:

- **» cd /simulator/logiccell/presynth/**
- **» 1-presynth-logiccell**

This will bring up the ModelSim main and wave windows. After reviewing the *test_logiccell.v* signals in the wave window, the logic cell signals can be added to the wave from the main window by highlighting the *u_logiccell* in the Workspace area, then right click and choose Add and Add to Wave. When the signals appear in the wave window, at the main window prompt type ***restart***, then click Restart on the next window. To simulate all the signals, at the main window prompt type ***run 5000.***

In the wave window select the signals *sel_mi0* to *sel_mo7*, right click to select Radix and Unsigned. This will allow you to see the decimal values for each mux select signals to be consistent with the *test_logiccell.v* file. Also, right click and select Radix and Hexadecimal for the *lutconfig*, *lutconfig0*, and *lutconfig1* signal. The wave that is being edited at this point can be saved and then opened in the future testing. From the wave window, select File and click Save and Format. To load the saved wave format, first select all the signals in the wave window and delete them. Then, choose File and click Open and Format, and select the *filename.do* (filename being the name of the saved file). Then redo the ***restart*** and ***run 5000*** steps to view the behavior of the waves.

To load the signals from figure 5.1.2, open and run *presynthesis_logiccell.do* from /logiccell/presynth/modelsim/ directory. Section A tests the full addition with the orange waves *eo* and *no* being the carry-out and sum bits. Section B tests the full subtraction with the orange waves *eo* and *no* being the borrow-out and difference bits. In the logic cell, when the *lutconfig* signal changes, the context switches from add to subtract instantaneously. One clock cycle is needed for the context switch when the new configuration is loaded from the memory (described in the PE implementation section).

Logic Cell Synthesis

To create the gate-level netlist design, follow the next steps:

- **» cd /simulator/logiccell/synthesis/**
- **» 2-synth-logiccell**

At first, the script will copy the necessary library files to the /logiccell/synthesis/synopsys/ directory and then the dc_shell synthesis tool from Synopsys [8] is initiated.

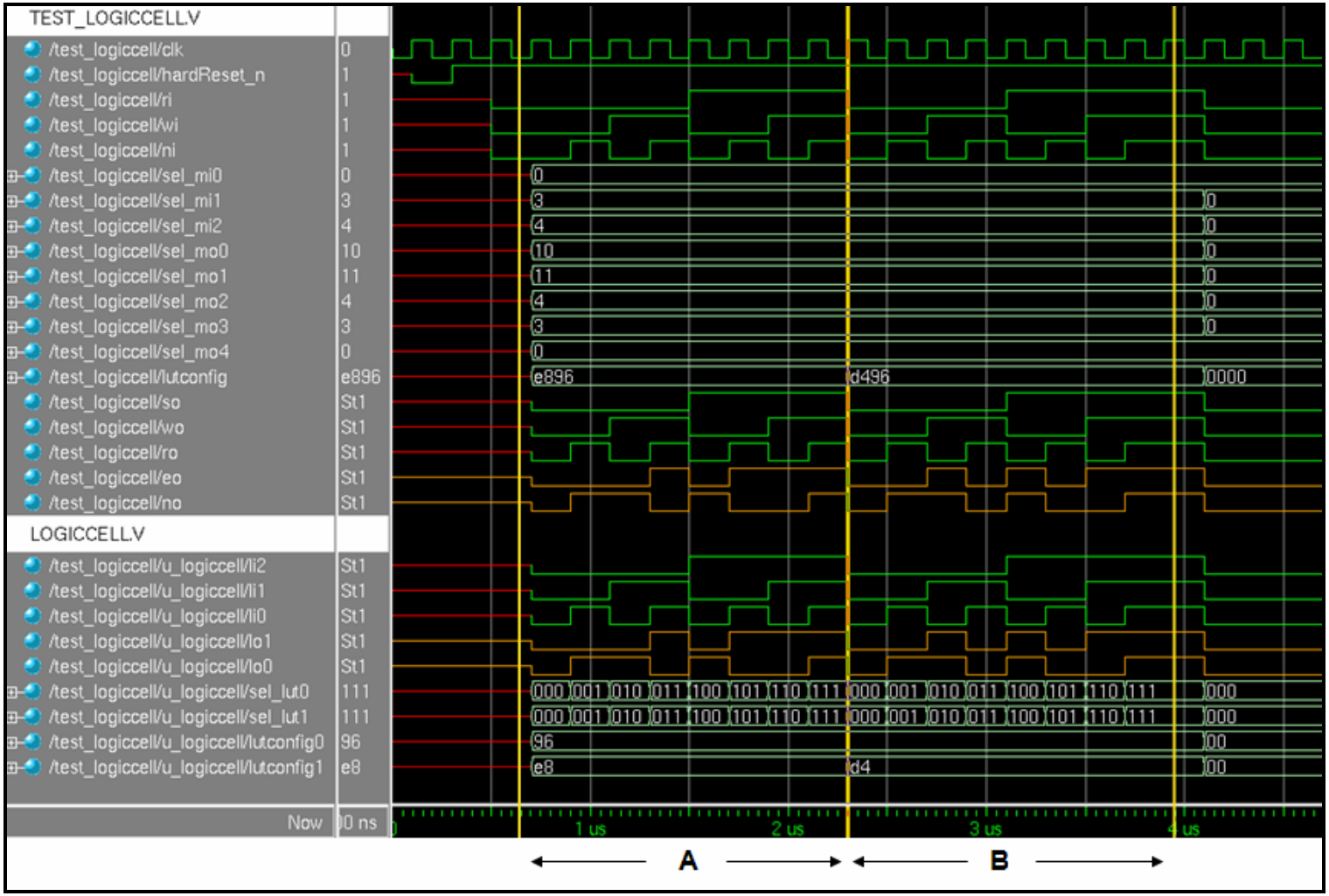


Figure 5.1.2: Logic Cell - Pre-Synthesis Simulation

The dc_shell executes the script *synth-logiccell.scr* located in the /synopsys/ directory. The details from this step are written to the file *info_logiccell_synthesis.txt* also located in the /synopsys/ directory. The synthesis tools will create the verilog netlist file *logiccell-synth.v* and the delay file *logiccell-synth.sdf* which are written in the /synopsys/ folder. These two files are used to test the netlist design using ModelSim simulation by following the next steps:

- **cd /simulator/logiccell/synthesis/**
- **3-post-synth-sim-logiccell**

This will bring up the ModelSim main and wave windows. To test and review the behavior of the signals, follow some of the steps described in the pre synthesis section. To load the signals from figure 5.1.3, open the wave file *synthesis-logiccell.do* from the /logiccell/synthesis/modelsim/ directory.

Section A from figure 5.1.3 tests the full addition with the orange waves *eo* and *no* being the carry-out and sum bits. Section B tests the full subtraction with the orange waves *eo* and *no* being the borrow-out and difference bits.

Logic Cell Place and Route

The First Encounter tools are used to generate the placement and routing of the netlist. To initiate the Encounter tools, follow the next steps:

- **cd /simulator/logiccell/asic/**
- **4-start-encounter**

The script *4-start-encounter* will copy the necessary encounter libraries to the /logiccell/asic/encounter/ directory and will bring up the encounter window.

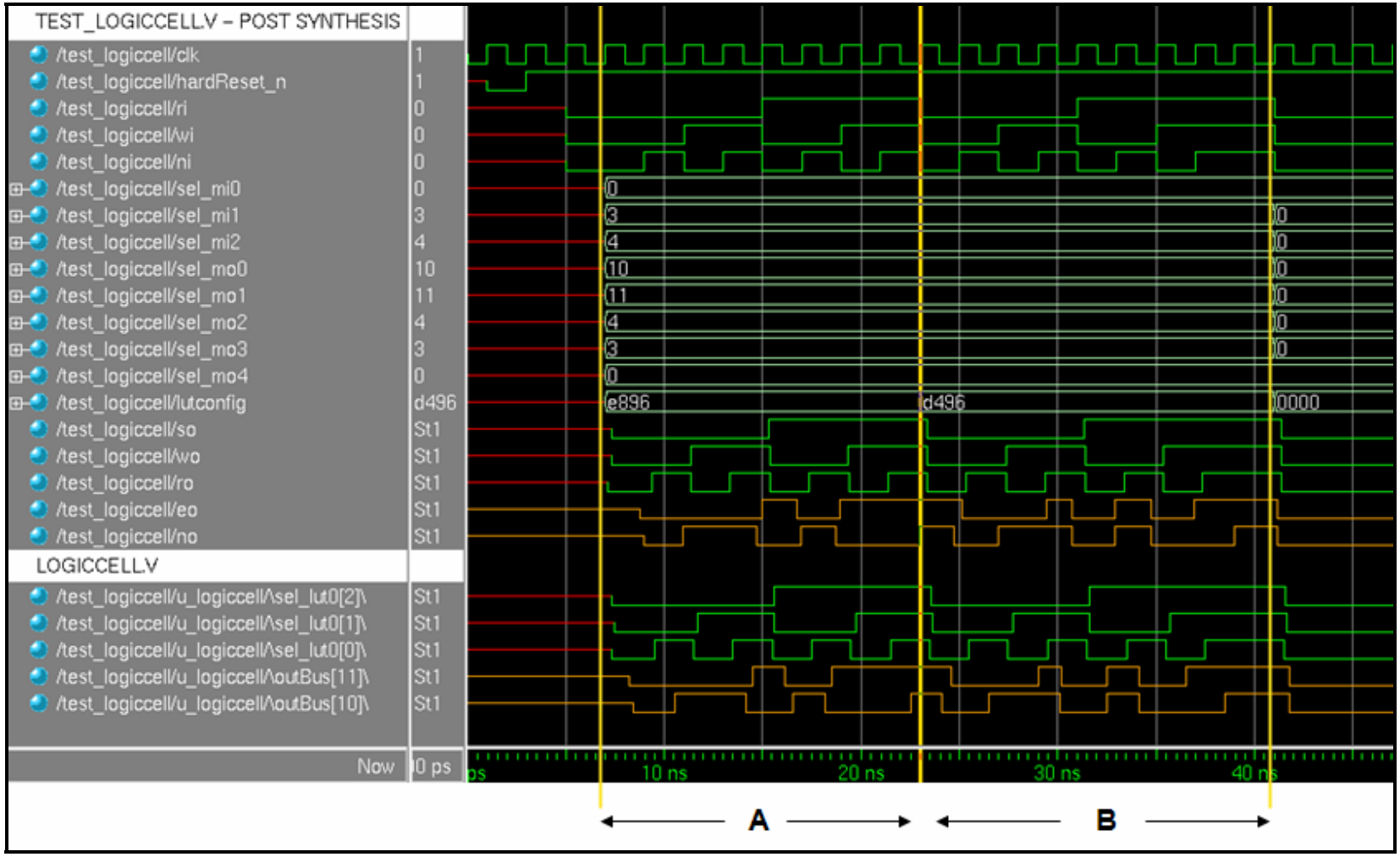


Figure 5.1.3: Logic Cell - Post-Synthesis Simulation

Follow the step by step tutorial in section 4.4, except when necessary enter "logiccell" instead of "array8x8". A generalized procedure for generating the logic cell placement and routing is listed bellow:

- Import Design – Design: netlist, LEF file, and timing library
- Import Design – Power Nets: VDD, VSS
- Floorplan – Specify Floorplan
- Floorplan – Power Planning – Add Rings
- Floorplan – Global Net Connections
- Place – Place
- Place – Filler – Add Filler
- Route – SRoute
- Route – WRoute
- Timing – Extract RC
- Timing – Calculate Delay – edit SDF Output File: ***logiccell-encounter.sdf***

Figure 5.1.4 shows the logic cell layout after placement and routing.

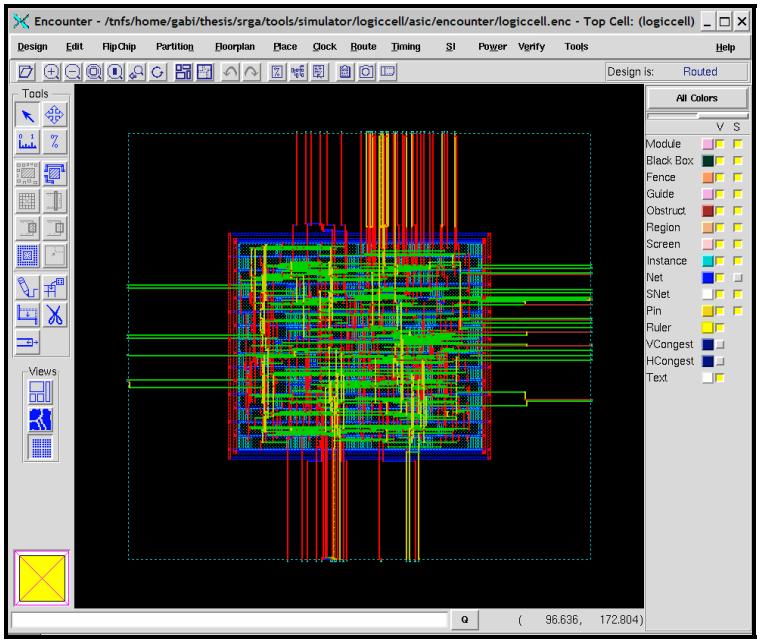


Figure 5.1.4: Logic Cell Layout after WRoute

The timing file created by place and route is *logiccell-encounter.sdf* and it is written in the logiccell/asic/encounter/ directory. The next step is to edit line 15 of the *logiccell-encounter.sdf* file from (CELLTYPE " ") to (CELLTYPE "logiccell").

To test the timing generated by the pace and route, follow the next steps to initiate the ModelSim tools:

- **» cd /simulator/logiccell/asic/**
- **» 5-post-layout-sim-logiccell**

The script *5-post-layout-sim-logiccell* copies the timing file from the /encounter/ to the /modelsim/ directory and simulates the netlist design created by the synthesis tools. To test and review the behavior of the signals, follow some of the steps described in the pre-synthesis section. The inputs and outputs are the same as in the pre-synthesis and synthesis sections. To load the signals in figure 5.1.5, from the wave window, open the wave file *encounter-logiccell.do*.

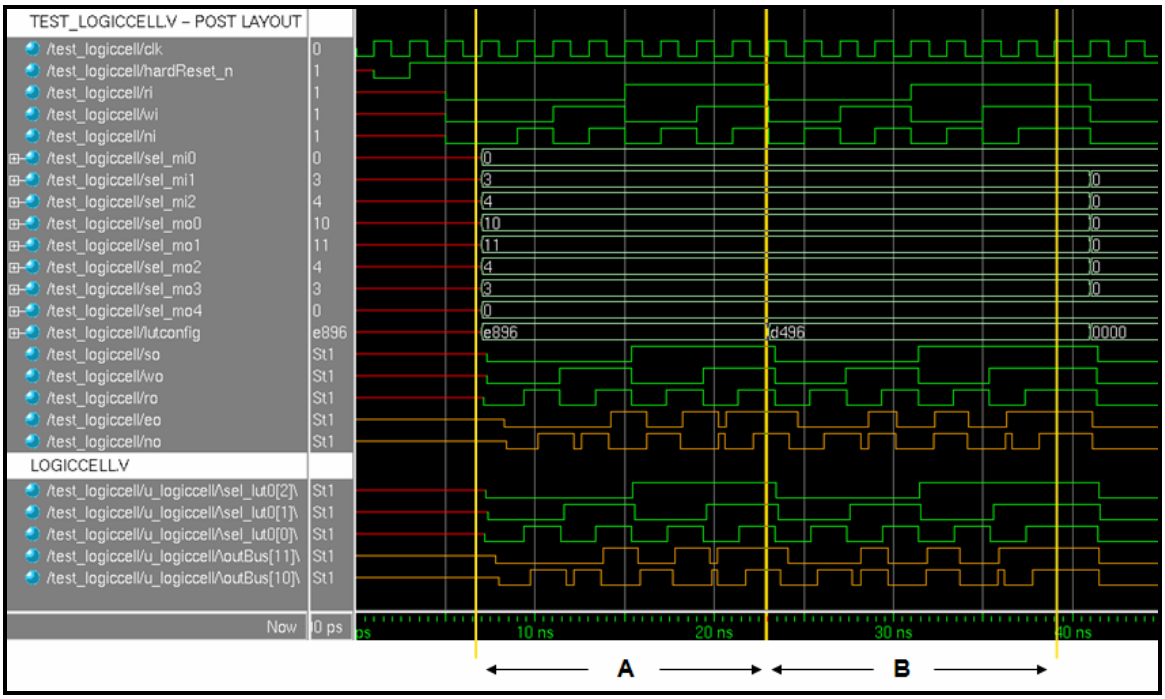


Figure 5.1.5: Logic Cell - Post-Layout Simulation

5.2 Memory Cell Implementation

Implementing the Memory Block follows the same format as the tutorial section 4.4. For this section, only a brief description will be demonstrated.

The memory array block was redesigned in the *memArray.vhd* to be synthesizable. The original *memArray.v* instantiates 77 rows of the sub-block *storageCellRow*. The *storageCellRow* block then instantiates 8 *storageCell* which also contains other sub-blocks. When used with the 8x8 array of PEs, the large number of instances for the design would surpass the allocated amount of files allowed in a folder. This happens when running the synthesis of the 8x8 array. The new *memArray.vhd* block was created with one instance *memArray*, and together with using the mixed compile method, the design was synthesized.

The file for testing the functionality of the memory block is *test_memcell.v* and it is located in the /srga/tools/simulator/memcell/ directory. The operations loaded to the memory array are the full addition and full subtraction discussed in section 3.3. In the test-bench figure 5.2.1 the configuration file *memcell.cnf* is loaded on line 70. After the configuration contexts were loaded into the memory, the *C_out* and *d_out* outputs were verified for proper functionality – illustrated from line 100 in figure 5.2.1.

To test the RTL design using ModelSim simulation, run the script *1-presynth-memcell* by following the next steps:

- **» cd /simulator/memcell/presynth/**
- **» 1-presynth-memcell**

```
67    switchContext<=1'b0;
68
69       //Load config.
70       $readmemb("../../../../../config_files/memcell.cnf", mem);
71       MOR0<=1'b1;
72       SRRy<=1'b0;
73
74       // i counts for the number of config words
75       for(i=0;i<=`CONFIGWORDS;i=i+1)
76       begin
77          tmp=mem[i];
78          wrData=1'b1;
79          contextAddr[2:0]=tmp[2:4]; //ccc part in the memcell.cnf
80          offset[6:0]=tmp[5:11];  //fffffff part in the memcell.cnf
81          if(tmp[12]!==1'bx)
82            begin
83               dIn<=tmp[12];
84               repeat (1) @ (posedge clk);
85            end
86
87          for (j=1;j<79;j=j+1)
88            begin
89             offset[6:0]=offset[6:0]+1;
90             if(tmp[12+j]!==1'bx)
91               begin
92                  dIn<=tmp[12+j];
93                  repeat (1) @ (posedge clk);
94               end
95          end
96       end  // for (i=0;i<`CONFIGWORDS;i=i+1)
97
98       MOR0<=1'b0;
99
100 // The switchContext = 1 will enable the load for config word reg. thus outputs c_out
101    switchContext<=1'b1;
102    contextAddr[2:0]<= 3'b000;
103    repeat (30) @ (posedge clk);
104
105 // the offset fied will enable d_out depending if there is a value at the offset
106    switchContext<=1'b0;
107    offset[6:0]<=7'b0110100;
108    repeat (30) @ (posedge clk);
109    offset[6:0]<=7'b0110101;
110    repeat (30) @ (posedge clk);
111    offset[6:0]<=7'b0110110;
112    repeat (30) @ (posedge clk);
113    offset[6:0]<=7'b0000000;
114
```

Figure 5.2.1: Section of Test Bench for Memory Cell Block

The behavior of the memory cell is shown in figure 5.2.2. The output *cOut* and *dOut* (in the code) are the same as *C_out* and *d_out* (in the schematic). *C_out* output the 77-bit configuration word when the MOR register becomes "0" and the *switchContext* signal becomes "1". The *d_out* signal will output the memory 1-bit that the *offset* signal (memory row) points to.

To create the gate-level netlist design, follow the next steps:

- **» cd /simulator/memcell/synthesis/**
- **» 2-synth-memcell**

The dc_shell executes the script *synth-memcell.scr* located in the /synopsys/ directory. The details from this step are written to the file *info_memcell_synthesis.txt* also located in the /synopsys/ directory. The synthesis tools will create the verilog netlist file *memcell-synth.v* and the delay file *memcell-synth.sdf* which are written in the /synopsys/ folder.

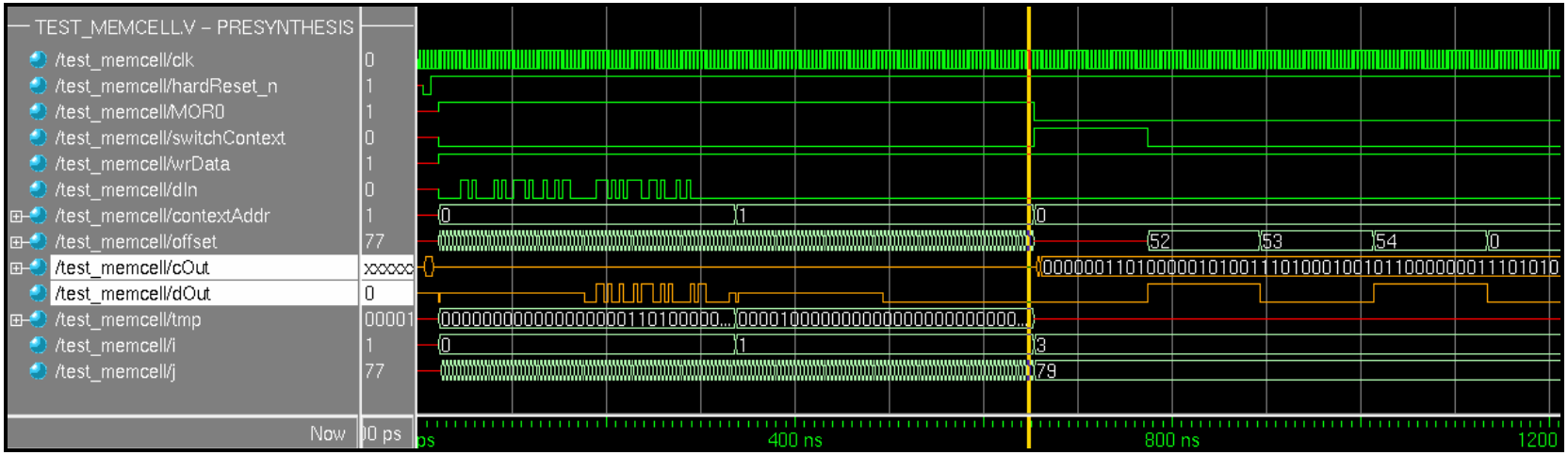


Figure 5.2.2: Memory Cell Testing

The netlist design is tested using ModelSim by following the next steps:

- **» cd /simulator/memcell/synthesis/**
- **» 3-post-synth-sim-memcell**

To initiate the Encounter tools, follow the next steps:

- **» cd /simulator/memcell/asic/**
- **» 4-start-encounter**

Follow the encounter section of the tutorial, except when necessary enter "memcell" instead of "array8x8" to generate the place and route and output the timing file ***memcell-encounter.sdf***. The next step is to edit line 15 of the *memcell-encounter.sdf* file from (CELLTYPE " ") to (CELLTYPE "memcell"). To test the timing generated by the pace and route, follow the next steps to initiate the ModelSim tools:

- **» cd /simulator/memcell/asic/**
- **» 5-post-layout-sim-memcell**

5.3 Switch Implementation

The switch is tested only at the RTL level because the block is very small – it only consists of three 2-bit muxes and a bidirectional switch. The test file for the switch module *test_switch.v* was designed to test the truth table illustrated in figure 5.3.1. The results can be seen in figure 5.3.2 from ModelSim simulations. To load the signals in figure 5.3.2, open the file *presynthesis_switch.do* in the /simulator/switch/presynth/modelsim/ directory. To test the RTL design using ModelSim simulation, run the script *1-presynth-memcell* by following the next steps:

- **» cd /simulator/switch/presynth/**
- **» 1-presynth-switch**

```
48 // truth table for the LIN s[2:0]
49 // for more details see switch.v
50 //        s[210]                          //----------outputs--------
51     s<=0000;   pi<=0; ri<=0; li<=0;       //po=li=0, ro=pi=0, lo=pi=0
52     repeat (1) @ (posedge clk) ;
53     s[2:0]<=100;   pi<=0; ri<=0; li<=1;   //po=li=0, ro=li=1, lo=pi=0
54     repeat (1) @ (posedge clk) ;
55     s[2:0]<=111;   pi<=0; ri<=1; li<=1;   //po=ri=1, ro=li=0, lo=ri=0
56     repeat (1) @ (posedge clk) ;
57     s[2:0]<=011;   pi<=1; ri<=1; li<=0;   //po=li=0, ro=li=1, lo=ri=1
58     repeat (1) @ (posedge clk) ;
59     s[2:0]<=000;   pi<=0; ri<=0; li<=0;
60     repeat (1) @ (posedge clk) ;
61
62 // truth table for MIN s[3]
63     s[3]<=1;  lr<=0;                  //p=0
64     repeat (1) @ (posedge clk) ;
65     s[3]<=1;  lr<=1;                  //p=1
66     repeat (1) @ (posedge clk) ;
67     s[3]<=1;  lr<=1;                  //p=1
68     repeat (1) @ (posedge clk) ;
69     s[3]<=0;  lr<=1;                  //p=0
70     repeat (1) @ (posedge clk) ;
71     s[3]<=0;  lr<=0;                  //p=0
72     end
```

Figure 5.3.1: Section of Test Bench for Switch Module

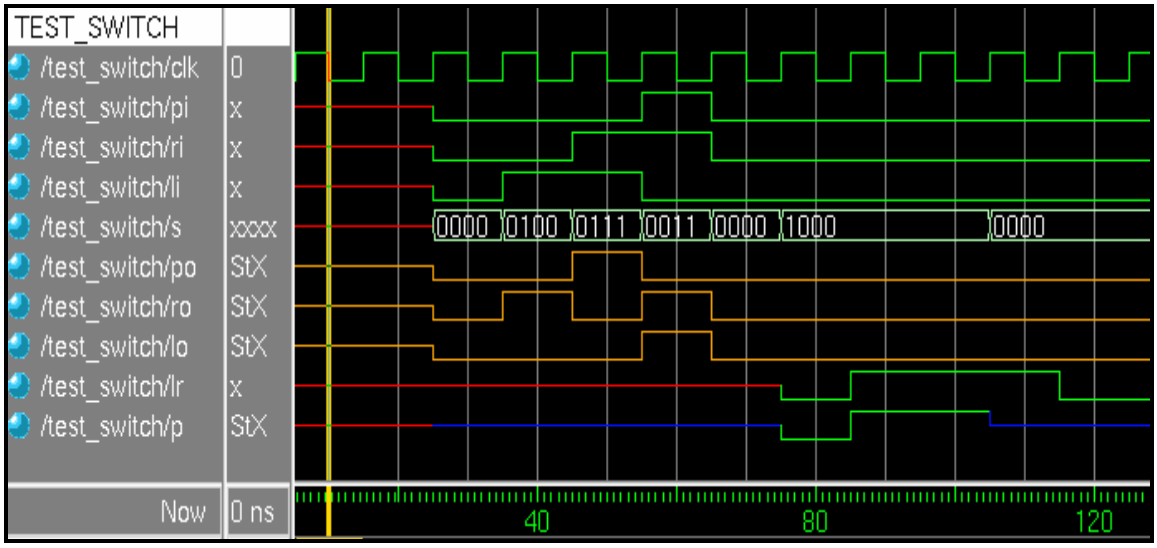


Figure 5.3.2: Switch Testing

5.4 PE Implementation

The PE block consists of a logic cell, a memory cell, a memory interface, a row switch, and a column switch. At this level all the modules are connected together and all the inputs and outputs to all the modules, including registers will go through the PE. The register signals are processed by the memory interface which will create the internal signals *wrMem*, *WrLog*, *switchContext*, *data_In*, *Rmi*, and *Cmi*. These signals are used to perform context switching and memory operations.

The file for testing the functionality of the PE block is *test_pe.v* and it is located in the /srga/tools/simulator/pe/ directory. The test-bench inputs data for the configuration word file to the PE (logic cell and memory cell) through one of the inputs *Cm* or *Rm* depending whether a column or row operation is expected. The PE is using the same format configuration file as the 8x8 tutorial for consistency; except that the registers are set up only for one PE instead of eight. The operations that are being tested are the full add and full subtract. The PE verifies the functionality of all the sub-blocks which is illustrated in figure 5.4.1 and figure 5.4.2.

To test the RTL design using ModelSim simulation, run the script *1-presynth-pe* by following the next steps:

- **» cd /simulator/pe/presynth/**
- **» 1-presynth-pe**

The wave windows that can be viewed at this stage from ModelSim are *pe_load_mem.do* and *pe_operation.do.*

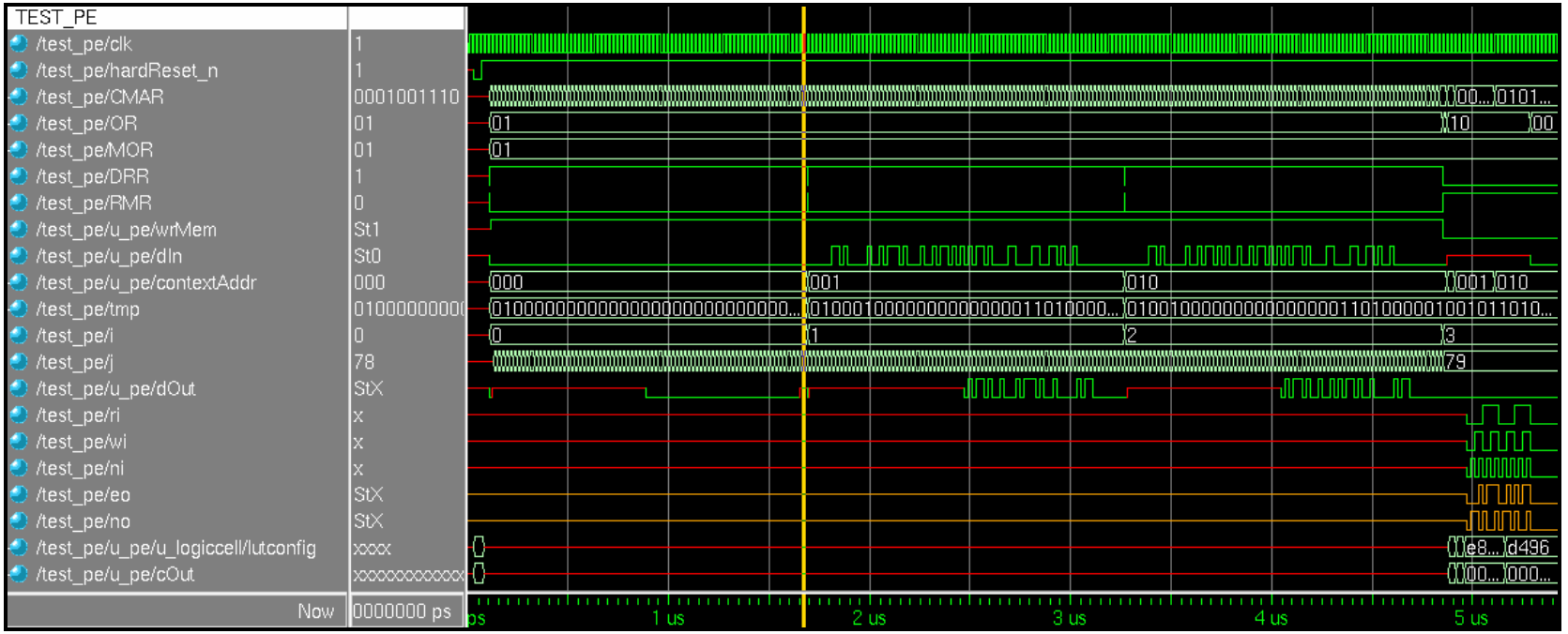


Figure 5.4.1: PE Testing – Loading Configuration

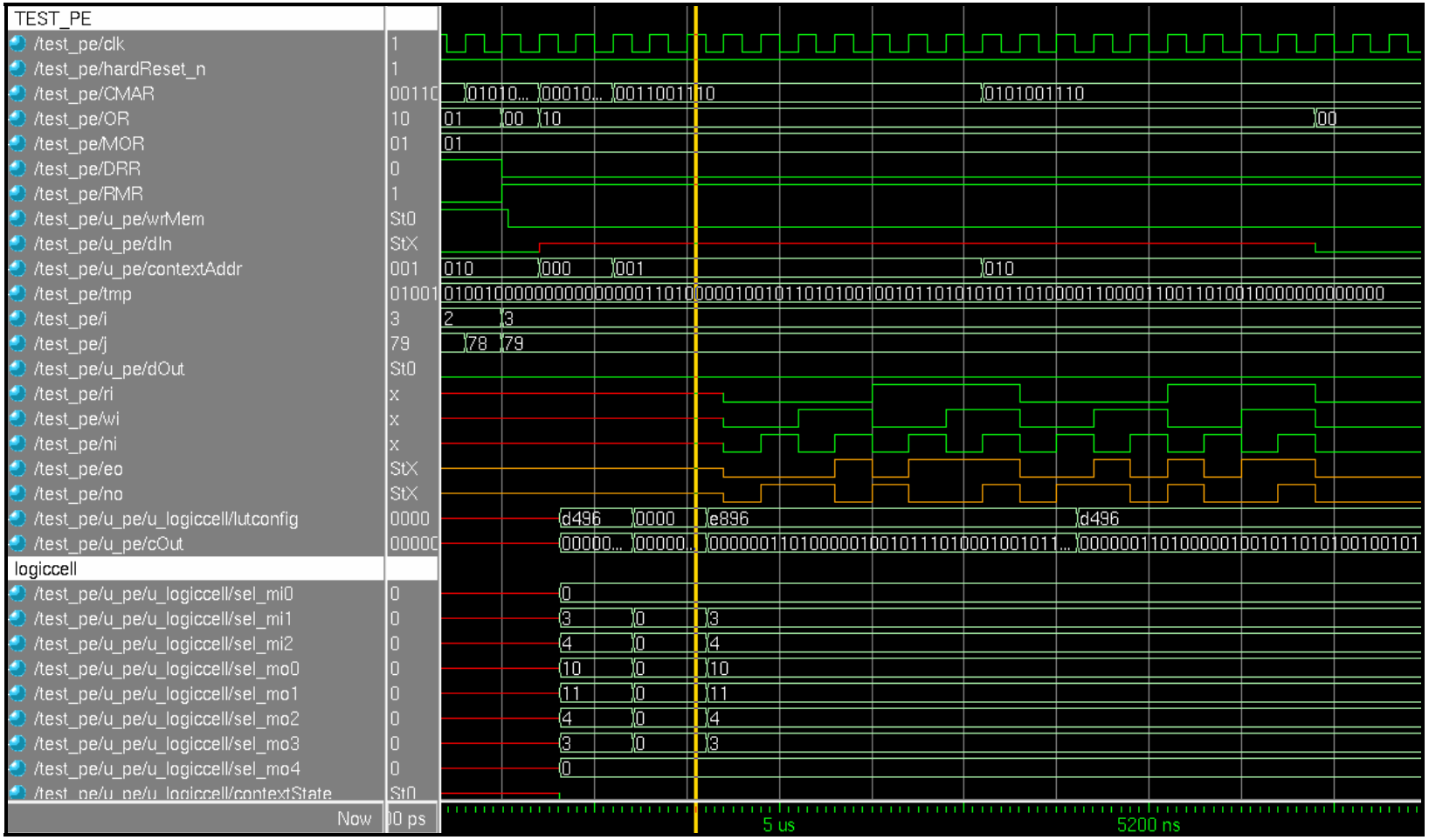


Figure 5.4.2: PE Testing – Operations

To create the gate-level netlist design, follow the next steps:

- **» cd /simulator/pe/synthesis/**
- **» 2-synth-pe**

The dc_shell executes the script *synth-pe.scr* located in the /synopsys/ directory. The details from this step are written to the file *info_pe_synthesis.txt* also located in the /synopsys/ directory. The synthesis tools will create the netlist file *pe-synth.v*, and using ModelSim it can be tested by following the next steps:

- **» cd /simulator/pe/synthesis/**
- **» 3-post-synth-sim-pe**

The wave window that can be viewed at this stage from ModelSim is *pe.do*.

To initiate the Encounter tools, follow the next steps:

- **» cd /simulator/pe/asic/**
- **» 4-start-encounter**

The necessary encounter libraries will be copied to the /pe/asic/encounter/ directory and will bring up the encounter window. Follow the step by step tutorial in section 4.4, except when necessary enter "pe" instead of "array8x8" to generate the place and route and output the timing file ***pe-encounter.sdf***.

The next step is to edit line 15 of the *pe-encounter.sdf* file from (CELLTYPE " ") to (CELLTYPE "pe").

To test the timing generated by the pace and route, follow the next steps to initiate the ModelSim tools:

- **» cd /simulator/pe/asic/**
- **» 5-post-layout-sim-pe**

The wave window that can be viewed at this stage from ModelSim is *pe.do*.

5.5 2x2 Array Implementation

The 2x2 array module consists of four instances of the PE block. Each PE is connected to one row switch and one column switch. The file for testing the functionality of the 2x2 array block is *test_array2x2.v* and it is located in the /srga/tools/simulator/2x2/ directory. The configuration word file for the 2x2 array is *2x2_config_words.cnf* located in the /srga/config_files/ directory. The format of the configuration word is the same as the PE's except there are 2-bis for the registers DRR and RMR instead of one bit.

In the *test_array2x2.v* file, the signal *CM* transfers the bits form the configuration word to *CM_tri*. The inouts *cpm10* and *cpm11* (from array2x2.v) are connected to *CM_tri[0]* and *CM_tri[1]* respectively. In *array2x2.v*, the signals *cpm10* and *cpm11* become the parent inout for each of the two column switches. Each switch then sends the configuration word bits to the *Cm* signal of each PE (see section 5.4 for the functionality of the PE). If the registers indicated a row operation, the configuration bits would be sent to *Rm* of each PE.

The 2x2 array verifies the functionality of all the sub-blocks which is illustrated in figure 5.5.1 and figure 5.5.2.

To test the RTL design using ModelSim simulation, run the script *1-presynth-2x2* by following the next steps:

- **» cd /simulator/2x2/presynth/**
- **» 1-presynth-2x2**

The wave windows that can be viewed at this stage from ModelSim are *2x2_load_mem.do* and *2x2_application.do.*

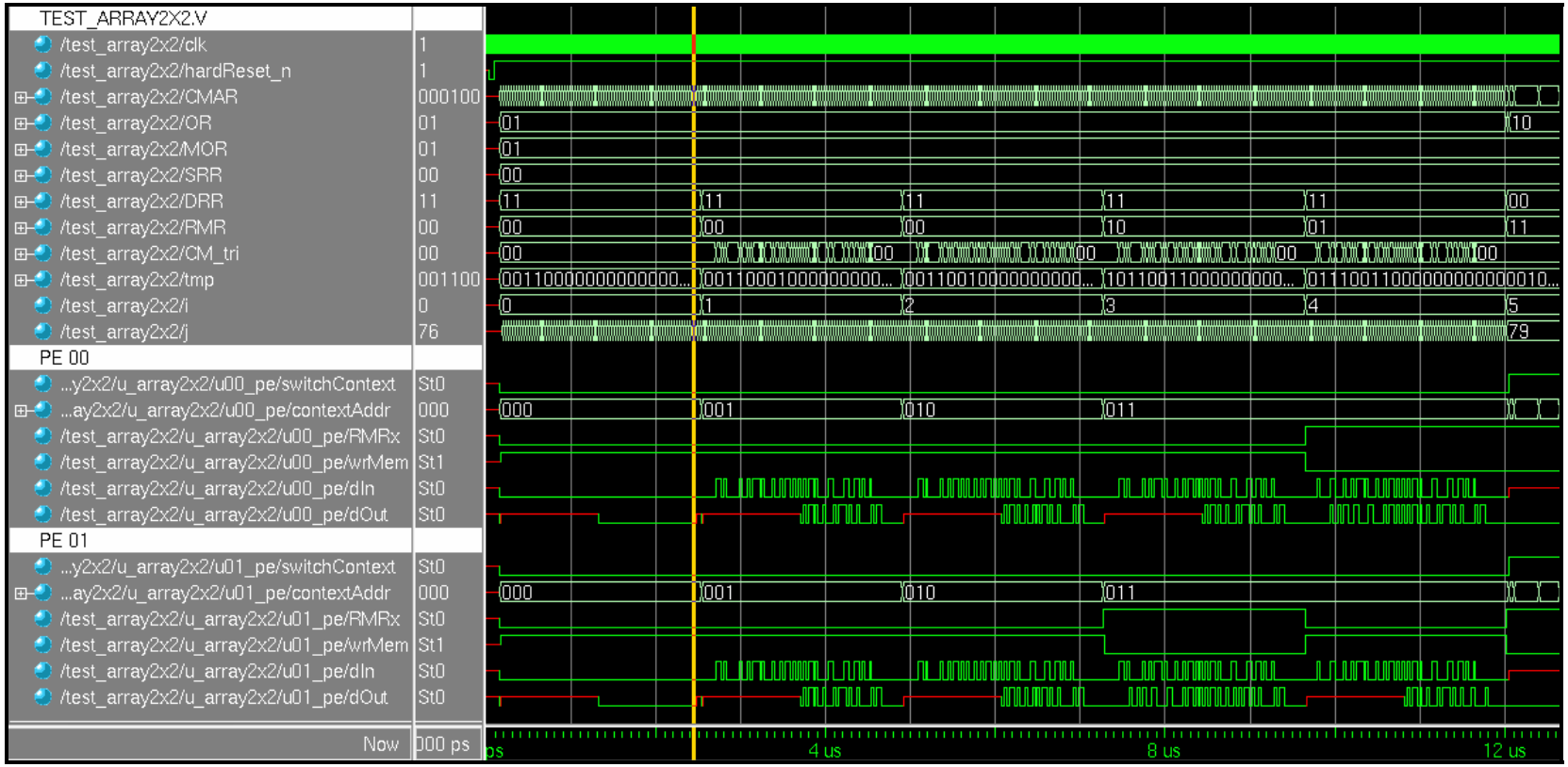


Figure 5.5.1: 2x2 Array Testing – Loading Configuration

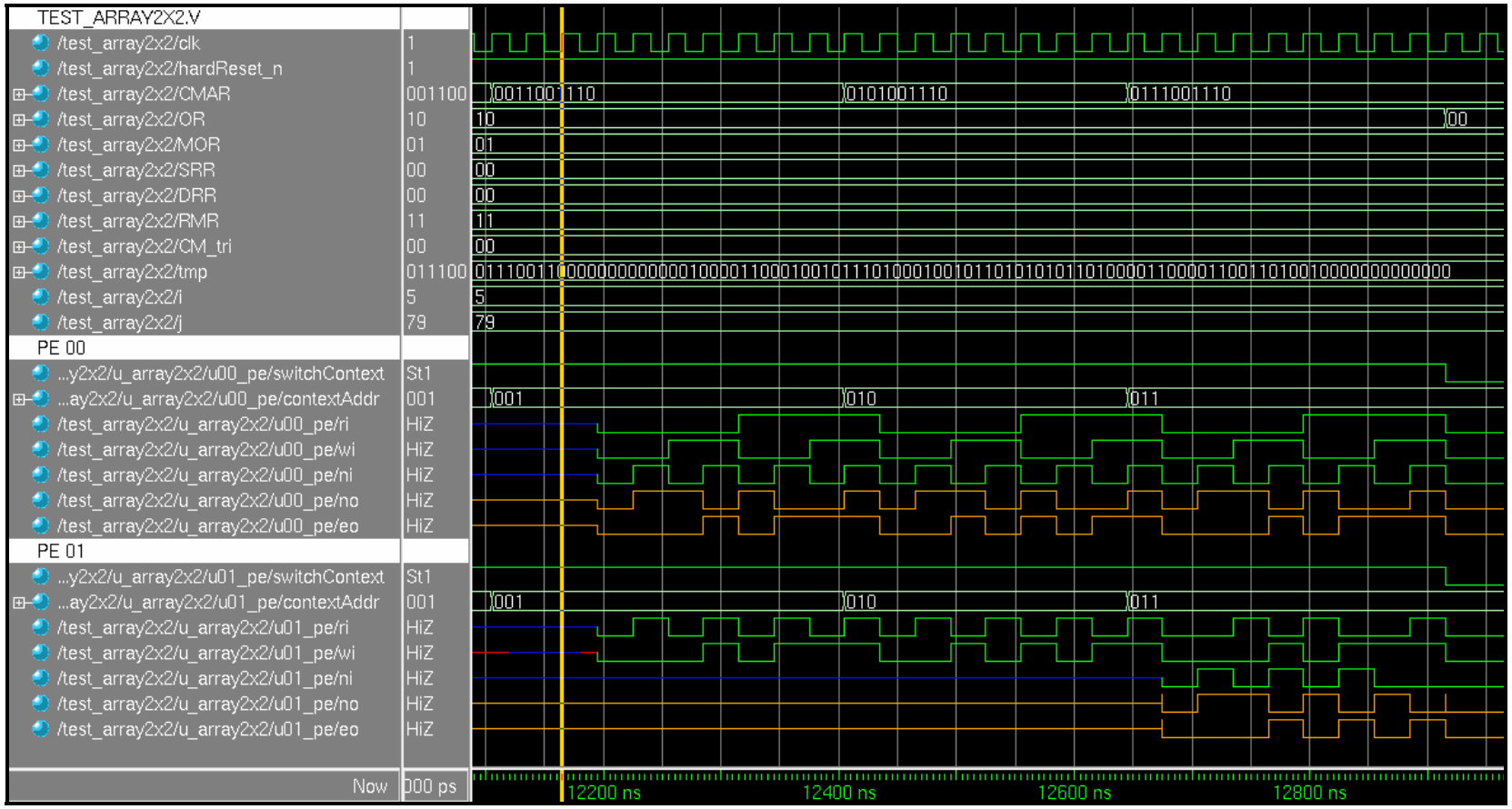


Figure 5.5.2: 2x2 Array Testing – Operations

To create the gate-level netlist design, follow the next steps:

- **» cd /simulator/2x2/synthesis/**
- **» 2-synth-2x2**

The dc_shell executes the script *synth-2x2.scr* located in the /synopsys/ directory. The details from this step are written to the file *info_2x2_synthesis.txt* also located in the /synopsys/ directory. The synthesis tools will create the netlist file *2x2-synth.v*, and it can be tested by following the next steps:

- **» cd /simulator/2x2/synthesis/**
- **» 3-post-synth-sim-2x2**

The wave window that can be viewed at this stage from ModelSim is *2x2.do*.

To initiate the Encounter tools, follow the next steps:

- **» cd /simulator/2x2/asic/**
- **» 4-start-encounter**

The necessary encounter libraries will be copied to the /2x2/asic/encounter/ directory and will initiate the encounter window. Follow the step by step tutorial in section 4.4, except when necessary enter "array2x2" instead of "array8x8" to generate the place and route and output the timing file ***2x2-encounter.sdf***.

The next step is to edit line 15 of the *2x2-encounter.sdf* file from (CELLTYPE " ") to (CELLTYPE "array2x2").

To test the timing generated by the pace and route, follow the next steps to initiate the ModelSim tools:

- **» cd /simulator/2x2/asic/**
- **» 5-post-layout-sim-2x2**

The wave window that can be viewed at this stage from ModelSim is *2x2.do*.

Chapter 6: *SRGA-UT* Results, Conclusion and Future Possibilities

6.1 Results

The objective of this thesis was to use the open code of the *SRGA-USC* design, created by the Department of EE-Systems at University of Southern California and the Department of Mathematics at the University of Trento (Italy), and implement it using the available EDA tools at the Department of ECE at the University of Tennessee. To achieve this goal, a great deal of knowledge of the original *SRGA-USC* architecture was required. The first step was to examine the RTL design, make the appropriated adjustments and pass the pre synthesis verification stage. The second challenge was to synthesize the RTL design, create the netlist and pass the synthesis verification stage. And the last step was to generate the place and route from the netlist design, produce the timing delay files, and test the final design for proper functionality.

The EDA tools used to implementing the *SRGA-UT* were: ModelSim for verification and simulation at all stages, Design Compiler for synthesis and creating the netlist, and First Encounter for performing the place and route and creating the delay files.

The complete SRGA architecture was describe in several thousand lines of Verilog and VHDL code. The synthesis and place and route were done using a standard cell library for a 0.18 µm process. The synthesized design can store 8 configuration contexts in each PE (this number is editable in the memory cell verilog files). The final implemented module has an 8x8 array of PEs. The number of gates and area for each module are shown in figure 6.1.1.

Module	Gates	Cells	Area* (μm^2)
Array 8x8	354,053	145,540	3,533,166
Array 4x4	88,512	36,384	883,279
Array 2x2	22,128	9,096	220,820
PE	5,532	2,274	55,205
Memory Cell	5,014	2,181	50,036
Memory Array	3,810	1,828	38,021
Logic Cell	476	74	4,750
* Area based on the Gate area of 9.9792 μm^2			

Figure 6.1.1: 8x8 SRGA-UT Results

The place and route generated a chip size of 5,413,300 μm^2.

The results obtained throughout the implementation of the *SRGA-UT*, demonstrated that the design was capable to switch context and perform memory access operations in a single clock cycle. The minimum clock cycle that was required to verify the design was 30ns. Thus the *SRGA-UT* design can be expected to operate at a frequency of 33MHz.

There were other tools used/tried to implement the *SRGA-UT*. The VCS simulating tools from Synopsys [10] were also used successfully at UT. The *SRGA-USC* design came with scripts to test their design using VCS tools. Since the ModelSim tools are more frequently used in the Microelectronic System Research Lab at the University of Tennessee, all the testing and verification was done using ModelSim.

The Silicon Ensemble tools [11] were also used to test the *SRGA-UT* place and route design. The SE tools were able to create the layout of all the modules in the design but did not produce the timing *(.sdf)* files for the 2x2 and 8x8 arrays.

6.2 Conclusion and Future Possibilities

The objective of the thesis was met, by exploring the original design, making the necessary changes, and using the available EDA tools to generate the results. A step-by-step tutorial was created for the 8x8 *SRGA-UT* using the standard cell library for a 0.18 µm process. The functionality of the *SRGA-UT* to perform context switching and memory access operations in a single clock cycle were confirmed. The *SRGA-UT* design has proven the importance of the design for reuse techniques.

Following are future possibilities that could be explored:

(1) One of the disadvantages of having such a large amount of gates for the memory array 3,810 is that it takes the most amount of space on the layout. This is because every PE in any combination of array size contains this number of gates for the memory array, which is implemented to store eight (8) configuration contexts. One option is to edit the memory cell code to store less configuration contexts, thus reducing the memory size for each PE – and/or – another possibility could be to create a global memory array, or a combination of global memory arrays. The global memory could store the configuration for the applications intended to be used. The global memory could be utilized through a network of muxes to attach each PE in a similar way as it is connected now. This could allow the *SRGA-UT* to maintain its functionality and greatly reduce the number of gates for the design. The area for each PE could be reduced by more than half thus improving the size, frequency, and possible cost of production.

(2) A second future possibility could be to implement the 16x16 array of PEs.

(3) A third future possibility could be to incorporate the *SRGA-UT* with the University of Tennessee SoC open core *Volunteer SoC* platform [12].

Reference:

[1] R. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna, "*A Self-Reconfigurable Gate Array Architecture*", in *Proc. Of the International Conference on Field Programmable Logic and Applications,* Sep. 2000, pp. 106-120.

[2] Wikipedia, the Free Encyclopedia, *Field-Programmable Gate Array*, [Online] Available: http://en.wikipedia.org.

[3] S. M. Scaler, and J. R. Vázquez, Sanders, A Lockheed Martin Company, "*The Design and Implementation of a Context Switching FPGA*", in *Proc. of the FPGAs in Custom Computing Machines*, Apr. 1998, pp. 78-85.

[4] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "*High-speed, hierarchical synchronous reconfigurable array*", In *Proc. of the International Symposium of Field Programmable Gate Array*, Feb. 1999 pp.69-78.

[5] Dr. Bouldin, Don. ECE 551/552/651/652 Class Notes. [Online] Available: http://www-ece.engr.utk.edu/ece/bouldin_courses/index.html

[6] Rick Mosher. AMI Semiconductor, *Structured ASIC Based SoC Design*, [Online] Available: http://www.us.design-reuse.com/articles/article7058.

[7] Mentor Graphics Corporation, *ModelSim SE Tutorial*, Version 5.7a Published: Jan. 2003.

[8] Synopsys, *Design Compiler User Guide*, Version W-2004.12, Dec. 2004.

[9] Cadence Design Systems, *Encounter Digital IC Design Platform*, [Online] Available: http://www.cadence.com/products, Unpublished.

[10] Synopsys, *VCS/VCSi User Guide*, Version 7.0.1, Apr. 2003.

[11] Mississippi State University, Computational Science and Engineering, *"Design_flow_se.ppt"*, [Online] Available: http://www.erc.msstate.edu/mpl /education/cadence/standard_cell/downloads.html

[12] Bouldin, Don. and R. Srivastava. "*An open System-on-Chip Platform for Education*", in *Proc. of 2004 European Workshop on Microelectronics Education (EWME)*, Lausanne, Switzerland. Apr. 15-16, 2004.

VITA

Gabriel Cozmin Chereches was born in Bacau, Romania. He started school at the age of seven in Bacau. At the age of nine he started taking swimming and diving lessons and not too long after he became the Diving National Champion. He went to school part time while he was traveling with the national team to compete in the international diving events. At the age of fourteen, in 1992, he competed in his first Olympic Games in Barcelona (Spain). At the age of fifteen, in 1993, after becoming the Junior World Champion at the ten meter platform in London (UK), he moved to the United States to continue his training and education. He attended high school at the Awty International School in Houston Texas. During his high school he participated in the 1996 Olympic Games in Atlanta (USA) on the ten meter platform. After graduating from Awty, he joined the University of Tennessee swimming and diving team to participate in the collegiate athletics and to receive an advanced education. As a college student athlete, he became NCAA All-American nine times, SEC Champion five times, and participated in the 2000 Olympic Games in Sydney (Australia). He graduated from the University of Tennessee with a Bachelors of Science in Electrical and Computer Engineering in the spring of 2002. In the fall of 2002, he started his Graduate studies in Electrical and Computer Engineering at the University of Tennessee. As a graduate student, he worked as a Graduate Teaching Assistant for the Engage Freshman Engineering Program and also worked part time at the Environmental Systems Corporation. In the fall of 2004 he started working full time at the Environmental Systems Corporation and graduating with a Masters of Science in December 2005…