12-2006

# Using Physical Compilation to Implement a System on Chip Platform

Pradeep M. Chimakurthy
*University of Tennessee - Knoxville*

To the Graduate Council:

I am submitting herewith a thesis written by Pradeep M. Chimakurthy entitled "Using Physical Compilation to Implement a System on Chip Platform." 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 requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Gregory Peterson, Itamar Elhanany

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

To the Graduate Council:

I am submitting herewith a thesis written by Pradeep M Chimakurthy entitled "Using Physical Compilation to Implement a System on Chip Platform." 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.

Donald Bouldin
Major Professor

We have read this thesis
and recommend its acceptance:

Gregory Peterson

Itamar Elhanany

Accepted for the Council:
Linda Painter
Interim Dean of Graduate Studies

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

# USING PHYSICAL COMPILATION TO IMPLEMENT A SYSTEM ON CHIP PLATFORM

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

PRADEEP M CHIMAKURTHY

DECEMBER 2006

# Acknowledgement

First, I would like to thank my advisor Dr. Donald Bouldin for his advice, constant encouragement and quick response to all my queries. I am also thankful to him for giving me access to a wide range of resources from a workshop to support sessions and the flow scripts, all of which were very vital to this work. I would also like to thank Dr. Gregory Peterson and Dr. Itamar Elhanany for agreeing to serve on my committee.

I am extremely grateful to Dr. Jay Whelan and Ms. Julia Gouffon of the Affymetrix Core Facility for partially supporting my education. Thanks, Julia for your constant advice and help.

I would like to thank Tushti Marwah and Wei Jiang for sharing components of their work on the SoC platform with me. I would also like to thank Synopsys, Inc. and our Synopsys university representative Ms. Troy Wood for giving me access to a workshop and the Galaxy Reference Flow scripts.

I would like to thank all my wonderful friends for their constant encouragement and support. The wonderful dinners, enjoyable discussions and captivating drives will always evoke pleasant memories. Each of you have thought me something new.

Finally, I would like to thanks my parents Sri Chimakurthy Sri Krishna Mohan Rao, Srimati Chimakurthy Vijayalakshmi and brother Prasanth without whom I would have achieved nothing. This thesis is dedicated to them.

# Abstract

The goal of this thesis was to setup a complete design flow involving physical synthesis. The design chosen for this purpose was a system-on-chip (SoC) platform developed at the University of Tennessee. It involves a Leon Processor with a minimal cache configuration, an AMBA on-chip bus and an Advanced Encryption Standard module which performs decryption.

As transistor size has entered the deep submicron level, iterations involved in the design cycle have increased due to the domination of interconnect delays over cell delays. Traditionally, interconnect delay has been estimated through the use of wire-load models. However, since there is no physical placement information, the delay estimation may be ineffective and result in increased iterations. Hence, placement-based synthesis has recently been introduced to provide better interconnect delay estimation. The tool used in this thesis to implement the system-on-chip design using physical synthesis is Synopsys Physical Compiler. The flow has been setup through the use of the Galaxy Reference Flow scripts obtained from Synopsys.

As part of the thesis, an analysis of the differences between a physically synthesized design and a logically synthesized one in terms of area and delay is presented.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1: Overview

## 1.1 Introduction

A fundamental observation in integrated electronics is Moore's Law which predicted that the components per chip or complexity would approximately double every two years resulting in reduced cost per function and dramatic improvement of system performance [1].

Integrated circuits have gone from having a few transistors known as Small Scale Integration (SSI) to hundreds of thousands of transistors and beyond known as Very Large Scale Integration (VLSI) [2]. Integrated circuits are used to build a huge variety of applications ranging from high performance computers to low-power handheld devices.

As designer productivity began to lag behind the available chip capacity, reusing components from earlier designs and integrating multiple components together on a single chip became advantageous. This approach was helpful in handling the increased chip complexity by reducing the design cycle time which translated into a reduced time to market. However, initially the components of SoCs did not have standard interfaces or characterization. This led to the development of intellectual property (IP) cores which could be reused. These cores are extensively simulated and characterized resulting in a reduction of the time required to use them compared to designing them from scratch.

A typical SoC architecture consists of a Central Processing Unit (CPU), Input/Output (I/O) interfaces, memory and peripherals along with a bus to enable communication among them [3]. A combination of this architecture along with a set of IP blocks is called a platform. Thus, a designer can easily derive a design

by choosing a set of components from the platform or setting parameters of the libraries' reconfigurable components [4].

The numbers of transistors on a chip have increased due to a reduction in transistor size. This reduced size also has translated to an increase in the switching speeds because of lower threshold voltages. But once the minimum feature size is less than 0.35 µm (also referred to as a deep submicron process), implementation becomes an even greater challenge. Various effects are encountered including signal coupling between adjacent metal lines or crosstalk between metal layers, increased interconnect delay and lower operating voltage to limit power dissipation which results in higher leakage currents[5].

## 1.2 Project Motivation

The process of converting a series of system specifications into a layout is called the design cycle. The traditional design cycle involved separate domains of logical and physical design. Initially, gate delays were a dominant portion of the total delay and hence the estimation of interconnect through statistical wire-load models was fine [5]. However, as we enter the deep submicron realm, interconnect becomes the dominating factor in the total delay as evidenced in Figure 1.1 [5].

Additionally, it has been shown that a wire-load model without any coarse placement information should have significant error. It was also shown that as the transistor size is reduced, the error in wire-load models impacts the stage delays to a greater extent. The major problem was shown to be weak drivers on long interconnects. Hence, placement data can help resolve this error in the interconnect estimation to an acceptable level [6].

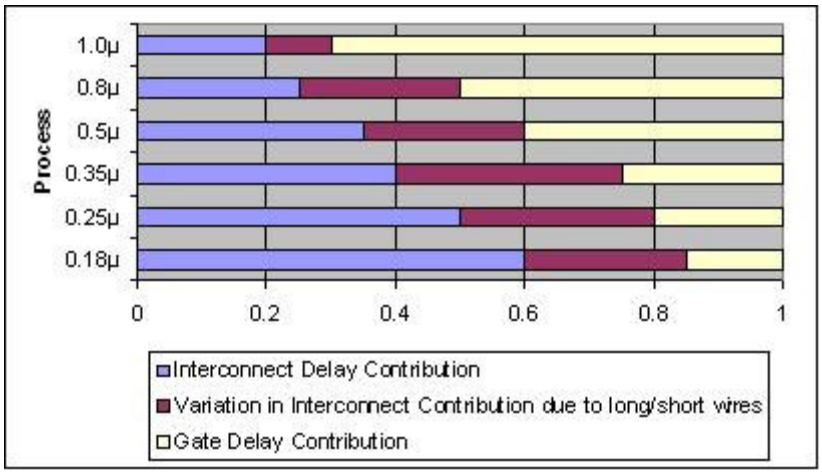


Figure 1.1: Process Size and Normalized Delay Contributions

In case of improper interconnect estimation, multiple iterations between the logical and physical domains will be required and achieving timing closure becomes harder. This in turn translates into an increase in the time to market. It has been estimated that a one-month increase in the time to market can result in a loss of ten percent of the potential revenue [7]. Hence, it becomes very important to achieve timing closure quickly.

Thus the use of physical synthesis, which is a combination of synthesis with placement, should result in a better interconnect estimation as physical information about the cells is known. This will allow appropriate drive strength selection for a particular cell. Hence, it should result in reduced design iterations.

## 1.3 Thesis Goals

The research for this thesis involved implementing a baseline SoC platform through both logic synthesis and physical synthesis. The baseline SoC platform developed as part of previous theses work is modified to include only the Leon processor and an Advanced Encryption Standard (AES) Block. The Galaxy

Reference Flow (GRF) scripts obtained from Synopsys were customized to implement the design flows.

The design targeted the IBM7RF 180-nm process and hence the required Milkyway libraries had to be created and integrated into the Galaxy Reference Flow. Once both the logic and physical synthesis flows were implemented, a comparison of the two designs in terms of slack and area were determined. Finally a tutorial was prepared about using Synopsys tools in both the design flows.

## 1.4 Thesis Outline

In this chapter a brief introduction about the relevance and necessity of physical synthesis is presented. Chapter Two is a review of the literature that shows the advantages and design improvements when synthesis and placement are combined. In Chapter Three, the Volunteer SoC platform and its components are discussed. In Chapter Four, the features of the Synopsys tools used to implement the design flow are presented. Chapter Five has the implementation details. Finally Chapter 6 concludes with the results and future goals.

# Chapter 2: Background

## 2.1 Traditional and Physical Design Cycles

A hardware description language like VHDL (Very High-Speed Integrated Circuit Hardware Description Language) or Verilog may be used to describe a design from its specifications. After verifying its functional operation through simulation, the design is mapped into a series of gates and optimized based on its technology library resulting in a gate-level net-list. However, the interconnect delays are unknown and hence are estimated through the use of statistical wire-load models which are based on the fan-out of each net. This conversion of the design into a gate-level net-list is called logic synthesis.

Following logic synthesis, the design is floorplanned to estimate positions of the various blocks of the chip and its power structures. This stage is useful to fix the positions of macros used in the design and to decide regions where no wires or cells should be placed so as to avoid high congestion. Placement, which is the next step, involves allocation of physical locations to the standard cells. This is followed by clock distribution and the building of clock trees in the Clock Tree Synthesis step. The next step involves actually laying down the metal interconnect to connect the cells and is called routing. Finally, the parasitics of the circuit are extracted and back-annotated to the routed net-list to verify design operation at the required frequency. In case of a hierarchical design, each block is separately taken through from logic synthesis to routing and finally all these blocks are interconnected at the top level.

Unfortunately, once the entire design cycle is completed, there is an increased chance that the design will not meet its timing constraints after routing. Hence, the implemented design will have to be extracted and these values back-

annotated to replace the wire-load models. But once the new net-list is taken through physical design, the interconnect values between the gates change and hence again an error may occur and multiple iterations may be required to achieve timing sign-off.

The major difference in the physical design cycle is the introduction of physical synthesis which is a combination of placement and synthesis. Once synthesis using wire-load models is completed, the design is floorplanned and placed. But as part of the placement, the physical synthesis tool will optimize the design based on the physical location of the cells. Hence the design cycle is more likely to require fewer iterations.

## 2.2 Previous Work

The concept of physical synthesis has been around for several years. This section summarizes some of the results motivating the use of physical synthesis.

The summary of an early paper that discussed a Placement-Driven Synthesis (PDS) approach is described below [8]. The wire capacitance models are used as part of synthesis tools to estimate interconnect based on the fan-out. These estimates differ before and after physical design. Thus, actual critical length paths may be longer than expected leading to a negative slack and multiple iterations between synthesis and placement. As part of their analysis, it was shown that the estimated capacitance for a particular net cannot be precise as there is a wide spread of actual capacitance.

Hence, the paper [8] suggests an approach where synthesis can run every time placement is done but constrained so that it can change only the power levels of the circuits based on the more accurate net capacitance available. At the time the authors wrote this paper, wire and gate delays were close to each other and

improvements were noted when placement-driven synthesis was used. It should certainly be appropriate today when interconnect delays dominate gate delays [5]. Their results, shown in the Table 2.1, compare area and slack amongst other factors between placement-driven synthesis and traditional synthesis and placement approaches [8]. The authors noted that the two-fold increase in CPU time was an acceptable tradeoff to improve the timing convergence of microprocessors.

A later paper [9] discussed the use of placement transformations on partitions of a mainframe processor with the goal of timing optimization. In the Transformational Placement and Synthesis (TPS) approach, coarse optimization is done during synthesis followed by detailed and aggressive optimization during placement. The transformations included scan chain optimization to reduce the scan chain length by connecting the scan chain based on the physical locations of the scan cells. The results were then compared with a Synthesis-Placement and Re-synthesis (SPR) approach. Their results showed an improvement in slack and area. The other major observation in their paper was that timing improvement and closure was achieved through the use of a single cycle for TPS relative to multiple iterations for SPR.

A more recent comparison was done for a design fabricated in a 180-nm process [10]. The physical synthesis methodology and additional optimizations used to implement high-performance microprocessors are discussed. The chip was divided into hierarchical blocks and after implementing each block, the noise and timing information was passed up the hierarchy.

Initially wire-load models were used and once the blocks were stable, physical synthesis was done to obtain optimal placement along with timing, area and power optimization.

Table 2.1: Comparison between PDS and Placement Approach

| | Approach | Area (Gate Size) | Worst Slack(ps) | Electrical Violations | Normalized Wire length w.r.t STSP | CPU time (sec) |
|---|---|---|---|---|---|---|
| Design 1 | PDS | 7078 | -437 | 2 | 1.005 | 521 |
| | STSP | 7061 | -501 | 3 | 1.000 | 306 |
| Design 2 | PDS | 21768 | -1154 | 32 | 1.01 | 2077 |
| | STSP | 21653 | -2528 | 190 | 1.00 | 1092 |
| Design 3 | PDS | 33051 | -1720 | 461 | 0.995 | 2695 |
| | STSP | 32699 | -3372 | 644 | 1.00 | 1470 |
| Design 4 | PDS | 30479 | -895 | 394 | 0.999 | 1488 |
| | STSP | 30443 | -1339 | 493 | 1.000 | 885 |
| Design 5 | PDS | 34957 | -1882 | 225 | 1.009 | 2619 |
| | STSP | 35755 | -2889 | 436 | 1.000 | 1486 |
| Design 6 | PDS | 3147 | -226 | 4 | 1.017 | 177 |
| | STSP | 3145 | -274 | 3 | 1.00 | 85 |
| Design 7 | PDS | 11007 | -484 | 18 | 0.974 | 688 |
| | STSP | 11068 | -513 | 48 | 1.00 | 358 |
| Design 8 | PDS | 29947 | -1090 | 272 | 1.007 | 2796 |
| | STSP | 29525 | -3119 | 535 | 1.00 | 1340 |

Low Vt gates were used to achieve higher performance on timing critical paths. It is mentioned that the use of these optimizations can only be determined after placement and hence could be used during physical synthesis when timing and load estimates are precise.

Better interconnect estimation as part of physical synthesis is considered a result of the placement information. Hence, the difference between a Steiner model used to measure wiring distances and the final routing of nets has been analyzed. It is shown that by removing the shortest 10-20% of the nets, large error percentages disappear. The errors due to these short nets do not affect the delay much. Hence, the Steiner length approximation is precise enough to be used as part of the physical synthesis phase [10].

The authors also note that the clock consumes 70% of the power in the processor due to the last clock driver stages which drive the latches but may have been placed far away from them. This results in larger wire length and hence increased capacitance leading to greater power consumption. After global placement, an initial clock network is created by the authors' physical synthesis algorithm and then by moving the latches closer to the driver stages, power optimization can be done. When this clock optimization is combined with physical synthesis, the authors note that the logic can be optimized to account for the placement changes made by the insertion of the clock buffers [10].

Another optimization typically used as part of physical synthesis is circuit relocation wherein timing critical circuits are moved so that the net capacitance to be driven is reduced and the buffer insertion on long wires is avoided by redistributing distances between logic gates. Another frequently used optimization is the remapping of the technology-mapped gates. Yet another technique is to rebuild the buffer trees as part of physical synthesis since buffer

insertion prior to placement can cause high congestion and degraded performance due to the poor topology of the trees.

Figure 2.1 from their paper [10] is used to compare the slack and area results on using physical synthesis and when alternating between placement and synthesis. They conclude that a majority of the points show both slack and area improvements and that sometimes allowing a small area penalty resulted in significant slack improvement.

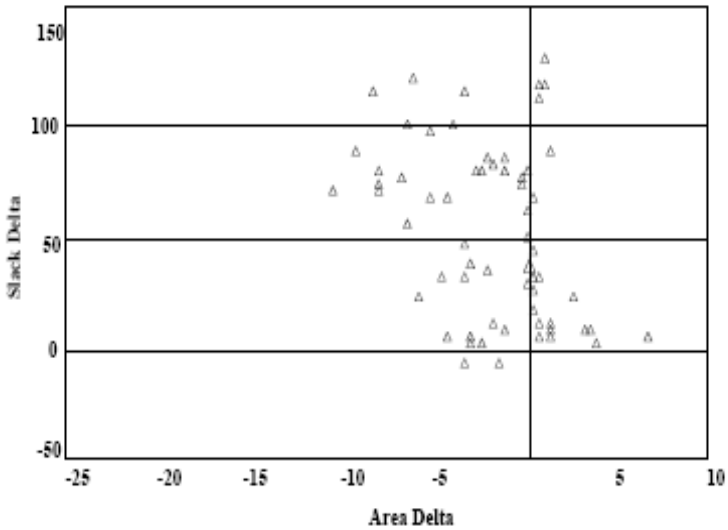


Figure 2.1: Slack and Area Comparison

# Chapter 3: System on Chip Platform

The Volunteer SoC Platform was developed as part of a graduate course at the University of Tennessee, Knoxville. Initially, IP cores were obtained or generated and verified individually. They were then modified to have standard interfaces so that they could be integrated with a Leon processor model [11]. Some modules were then integrated with the processor through its AMBA bus to create a platform as part of a previous thesis [12]. This platform consisted of an open core Advanced Encryption Standard (AES) block and Fast Fourier Transform and Finite Impulse Response cores developed by other students. As part of a later thesis [13], a baseline configuration was created using minimal cache and the open core AES block replaced with one that was developed in-house as part of a cryptographic project [14]. This baseline platform was used in this thesis to test the physical synthesis flow.

## 3.1 Leon Processor

Leon is a SPARC V8 processor and its VHDL model can be obtained online for free [11]. Its features include separate instruction and data caches, a hardware multiplier and divider, interrupt controller, two UARTs (Universal Asynchronous Receiver Transmitters), two timers, a memory controller and Ethernet and PCI (Peripheral Control Interface) interfaces. It consists of two different buses from the Advanced Microcontroller Bus Architecture (AMBA), namely the Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB) through which new modules can be added to the processor. The model has been synthesized and simulated using various tools for various technologies. The block diagram of the Leon processor is shown in Figure 3.1 [11].

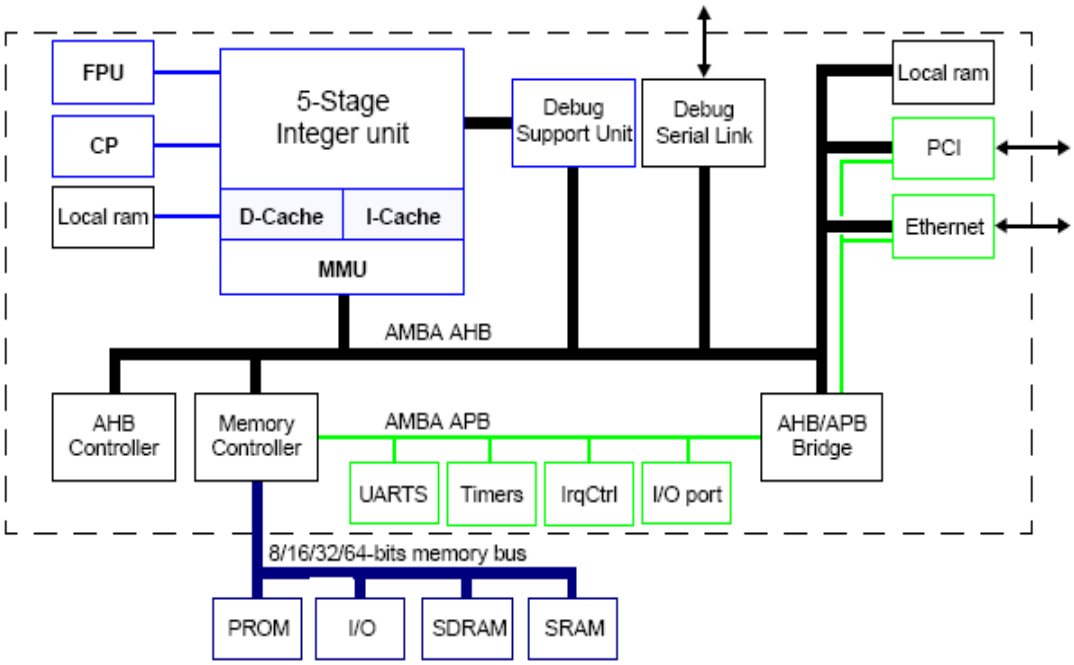


Figure 3.1: Leon 2-1.0.30-xst Processor Block Diagram

The Leon model can be configured using a graphic configuration tool which creates a *device.vhd* file in the Leon folder when the 'make dep' command is run. This action combined with the *config.vhd* file setups the Leon for a particular configuration. To obtain a baseline platform, a minimal cache size was chosen. Also both the Ethernet and PCI interfaces were not enabled. Both the instruction and data caches were configured to be direct-mapped having 8-bit line sizes. We also needed to generate a cache tag RAM since each line in the cache has a tag associated with it. Synchronous single-port RAM cells were used to implement both the data and tag caches. The technology-mapping file to be used for the caches and I/O with the IBM7RF technology was created as part of an earlier thesis [15]. Table 3.1 provided in the Leon User Manual was used to determine the cache sizes [11].

Table 3.1: Selection of RAM Size

| Cache Set Size | Words / Line | Tag Ram | Data Ram |
|---|---|---|---|
| 1Kbyte | 8 | 32x30 | 256x32 |
| 1Kbyte | 4 | 64x26 | 256x32 |
| 2Kbyte | 8 | 64x29 | 512x32 |
| 2Kbyte | 4 | 128x25 | 512x32 |
| 4Kbyte | 8 | 128x28 | 1024x32 |
| 4Kbyte | 4 | 256x24 | 1024x32 |
| 8Kbyte | 8 | 256x27 | 2048x32 |
| 8Kbyte | 4 | 512x23 | 2048x32 |
| 16Kbyte | 8 | 512x26 | 4096x32 |
| 16Kbyte | 4 | 1024x22 | 4096x32 |

For the Integer Unit, the default eight register windows were chosen. A register is used to store the temporary values during the execution of a program. A group of eight registers is called a window. Register windows allow multiple parts of the program to access its own group of registers during procedure calls. Hence, the program can have a chain of eight procedure calls deep without saving the register contents to the memory. In the case of the SPARC architecture, any part of the program can access 32 registers (8 global registers + 3 register windows). A register window shifts by 16 registers for each procedure call since 8 registers that are output in the previous level act as input registers for the current level [16]. Hence, the total number of registers required for the 8 register windows is 8 + 24*1+ 16*6 +8*1 = 136. Since the Integer Unit file requires one 32-bit write port and two 32-bit read ports, we use two parallel dual-port RAM cells each with a size of 136 x 32. Artisan RAM generators were used to generate both the single-port and dual-port memories.

## 3.2 Advanced Microprocessor Bus Architecture [17, 18]

Advanced Microprocessor Bus Architecture (AMBA) is the on-chip bus protocol from ARM. Other bus protocols available are Core Connect from IBM and the Wishbone bus from Silicore. AMBA consists of the following:

a) Advanced High-performance Bus (AHB) - It is a high performance bus used to connect processors, on-chip memory and memory controllers.

b) Advanced System Bus (ASB) - It is an alternative system bus used when the high performance features of AHB are not required. It does not have the burst transfer and split transaction capabilities of the AHB. Burst transfer allows one or more consistent width data transactions to an incremental region of address space. Split transactions improve bus utilization by ensuring that the arbiter will allow other masters to access the bus until the slave can complete a transfer.

c) Advanced Peripheral Bus (APB) - It is a low power and low performance bus used for peripherals. The APB interface has reduced complexity.

Figure 3.2 illustrates a typical AMBA bus system.

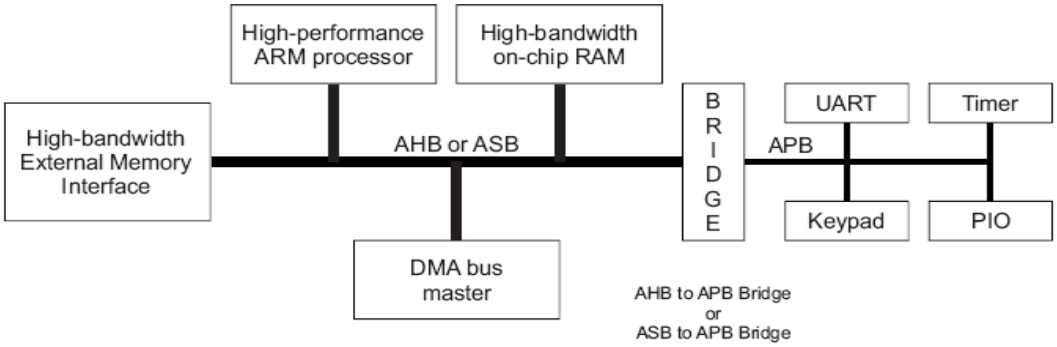


Figure3.2: Typical AMBA System [17]

One of the major specifications noted was that AMBA was derived to be technology-independent so that it could be migrated across various processes. In the case of the Leon processor model, the AHB and APB buses have been used. Both the AHB and APB bus cycles are defined from one rising edge to the next rising edge. The high bandwidth AHB and low bandwidth APB are linked via a bridge which connects the master to the peripheral bus slaves.

### 3.2.1 Advanced High-performance Bus (AHB)

The AHB can have multiple bus masters and usually the processor is the master on this bus. The APB bridge and internal memories are usually slaves on the AHB. It has an arbiter which ensures that there is only one bus master at any time and routes the address and control signals to all the slaves. A centralized decoder controls the data reading and also selects appropriate signals for the slave involved in the transfer. The bus has separate read (HRDATA) and write (HWDATA) buses. The HWRITE signal is an active high signal used to indicate the direction of data transfer. The bus master can also lock the bus with the arbiter for a certain number of transfers. Figure 3.3 can be used to understand the AHB bus cycle.

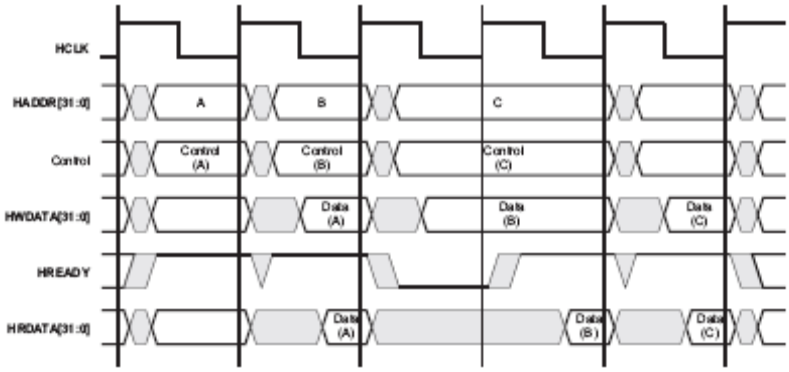


Figure 3.3: AHB Bus Cycle with Wait States [17]

Initially, a master requests the bus and the arbiter decides which master is assigned control of the bus. Normally a master can complete all the transfers using a single burst but the arbiter has the ability to break the burst. The transfer consists of a single cycle which cannot be extended and during this time, the address and control signals are sent. This is followed by one or more data cycles which can be extended using the HREADY signal. A high HREADY signal indicates data transfer can be completed. When the HREADY signal is driven low by a slave, wait states are introduced and the slaves can sample or read data for extra time. While the bus master holds the data stable throughout the extended cycle for a write operation, the read data is available just when the transfer is about to complete. Pipelining is implemented as the address phase of the current transfer occurs during the data transfer of the previous phase.

### 3.2.2 Advanced Peripheral Bus (APB)

The bridge is the only master on the APB and all other peripherals on it are slaves. It converts data from the AHB bus into a suitable format for the APB slaves and also generates the corresponding select signals for the slaves (PSELx) based on the address. Its operation involves a Setup state in which it remains for one cycle, following which it enters the Enable state which also lasts a single cycle. In the Setup state, the PWRITE signal is made high or low depending on the direction of transfer and the address is put onto PADDR. It has a read (PRDATA) data bus which is driven when the PWRITE signal is low while the write (PWDATA) data bus is driven when the PWRITE signal is high. The PENABLE signal is asserted to indicate that the Enable state is taking place. The address, data and control signals remain valid throughout the Enable state. The PENABLE will be unasserted to indicate the end of the state. The PSELx will also be unasserted unless there is another transfer to be made.

### 3.2.3 Bus Operation on Interfacing APB and AHB

When the AHB is interfaced to the APB, the address is sampled by the bridge and broadcast along with selecting the appropriate peripheral. During the Setup cycle, the HREADY will be asserted low. Following the Setup cycle, the PENABLE is made high and the data peripheral must provide the read data during this enable cycle which can be routed back onto the AHB so that the bus master can sample it at the fourth clock edge after transfer was initiated.

For high frequency systems, the read data is registered by the bridge and transmitted on the AHB for the bus master during the fourth clock cycle introducing an extra wait state. In the case of a write operation, the bridge will sample the address and data and hold them for the peripheral throughout the write cycle. It is also noted that the bridge requires two address registers so that it can sample the next address on the AHB while the current write transfer occurs on the peripheral bus [17].

## 3.3 Advanced Encryption Standard Block

AES is an encryption standard adopted by the US government. It has a fixed block size of 128 bits and can have a key size of 128, 192 or 256 bits. This block had been generated in-house as part of a cryptographic project [14] and had been simulated and verified using the Xilinx Virtex 1000E FPGA. This is the only user IP block integrated as part of the SoC in this thesis. A key size of 128 bits has been used and only decryption was performed with the encrypted text given as an input.

.

## 3.4 Artisan RAM

The required synchronous block RAMs for the design were generated using Artisan RAM generators. We needed the single-port RAM generator for the cache and tag RAMs and the dual-port RAM generator for the integer unit. For the Artisan RAM, the Chip ENable pin (CEN) needs to be low and then based on the Write Enable (WEN) pin, a read (high) or write (low) occurs. As part of a previous thesis [12], it was found that these RAMs cannot be directly integrated with the Leon processor. In the case of the RAM cycles, the memory address should already be present before the rising clock edge. However, the Leon processor loads the address and data at the rising edge of the clock resulting in a cache failure. This is illustrated using Figure 3.4 and Figure 3.5 respectively. To avoid this, a wrapper was created which allows the RAM to be interfaced with the processor without failing.

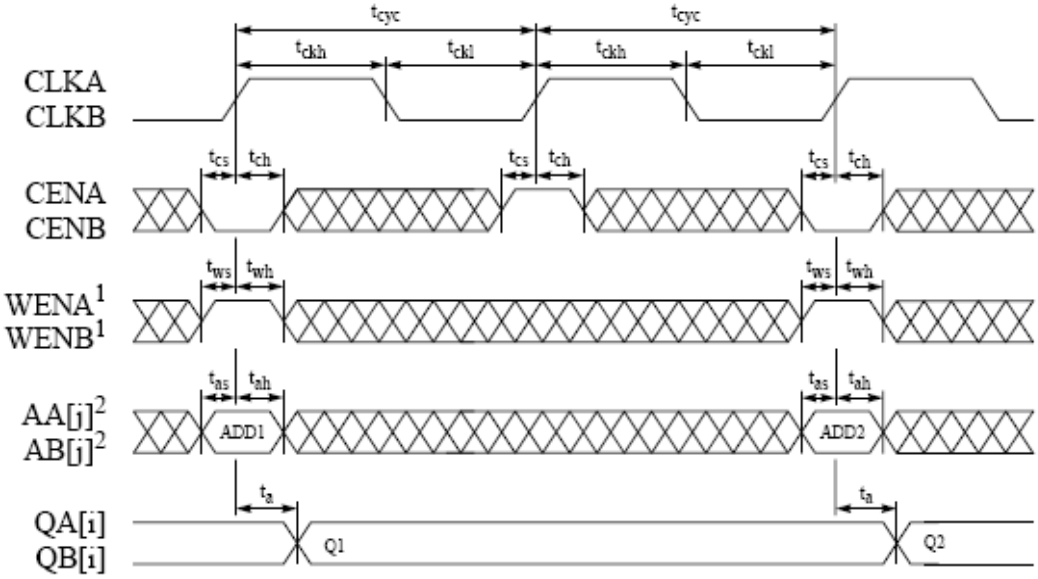


Figure 3.4: Dual Port SRAM Read Cycle [19]

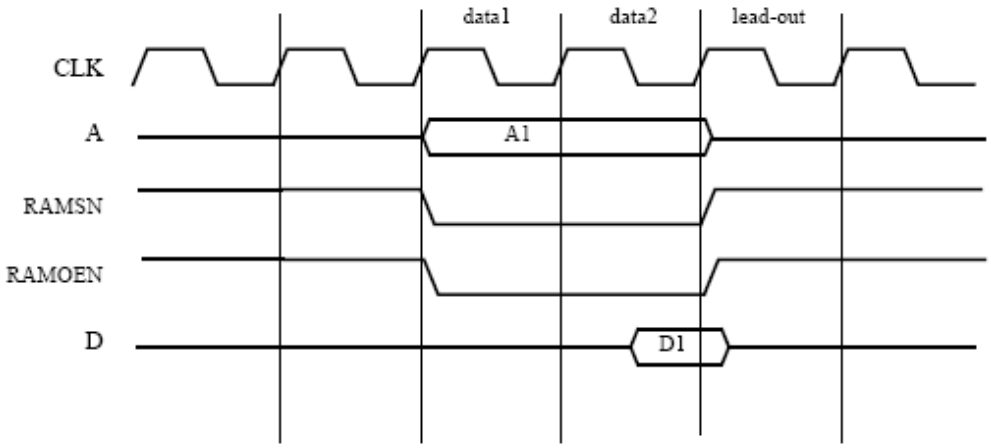


Figure 3.5: Leon Processor Read Cycle [11]

## 3.5 LEON/ERC32 GNU Cross- Compiler System (LECCS)

LECCS is a free multi-platform cross-compilation system provided by Gaisler Research. It is based on gcc and the Real-Time Executive for Multiprocessor Systems kernel and allows compilation of C/C++ programs so that they can be run on Leon [20]. The C program to be run is compiled and loaded into *ram.dat* which is then read by the test bench. Figure 3.6 shows the analogy between gcc and LECCS.

## 3.6 Volunteer SoC Integration [12, 13]

For the IP block to be easily integrated, it was decided that the blocks would have a 32-bit data and address width. The registers would be initialized using a reset signal and there would be a GO signal to inform the IP to start its operation and it would issue a DONE signal once it was done.

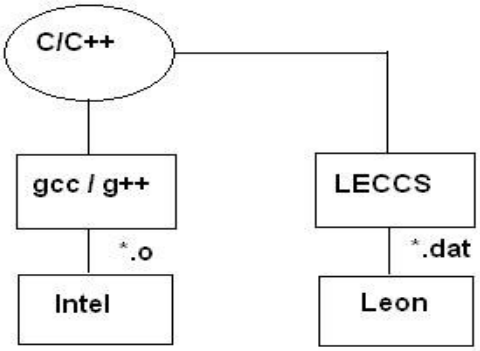


Figure 3.6: Analogy Between gcc and LECCS [12]

The IP block was then interfaced to the AMBA bus through another wrapper that would allow it to be either an AHB master or an APB slave. The *ambacomp.vhd* file was modified so that the IP block could be added as an AMBA component. From Table 3.2 it can be seen that the slaves on the APB need to mapped in the address range 0x80000000 - 0x8FFFFFFF which corresponds to the APB bridge. Hence the *apbmst.vhd* file was modified to map the AES block into the address range 0x80000300 - 0x800003FF. In the *mcore.vhd* file, the AES block was added such that it has a priority index of 1 on the AHB. The default master on the AHB is the processor with an index of 0. Hence, the AES was assigned a higher priority than Leon on the AHB.

The final derivative configuration of the Volunteer SoC platform used for physical compilation is shown in Figure 3.7. Thus the AES block is setup so that it receives control signals over the Advanced Peripheral Bus and transfers its data on the Advanced High-Performance Bus.

Table3.2: AHB Address Allocation [11]

| Address Range | Size | Mapping | Module |
|---|---|---|---|
| 0x00000000 - 0x1FFFFFFF | 512M | PROM | Memory Controller |
| 0x20000000 - 0x3FFFFFFF | 512M | Memory Bus I/O | Memory Controller |
| 0x40000000 - 0x7FFFFFFF | 1G | SRAM / SDRAM | Memory Controller |
| 0x80000000 - 0x8FFFFFFF | 256M | On-Chip Registers | APB Bridge |
| 0x90000000 - 0x9FFFFFFF | 256M | Debug Support Unit | DSU |
| 0xB0000000 - 0xB001FFFF | 128M | Ethernet registers | Ethernet |

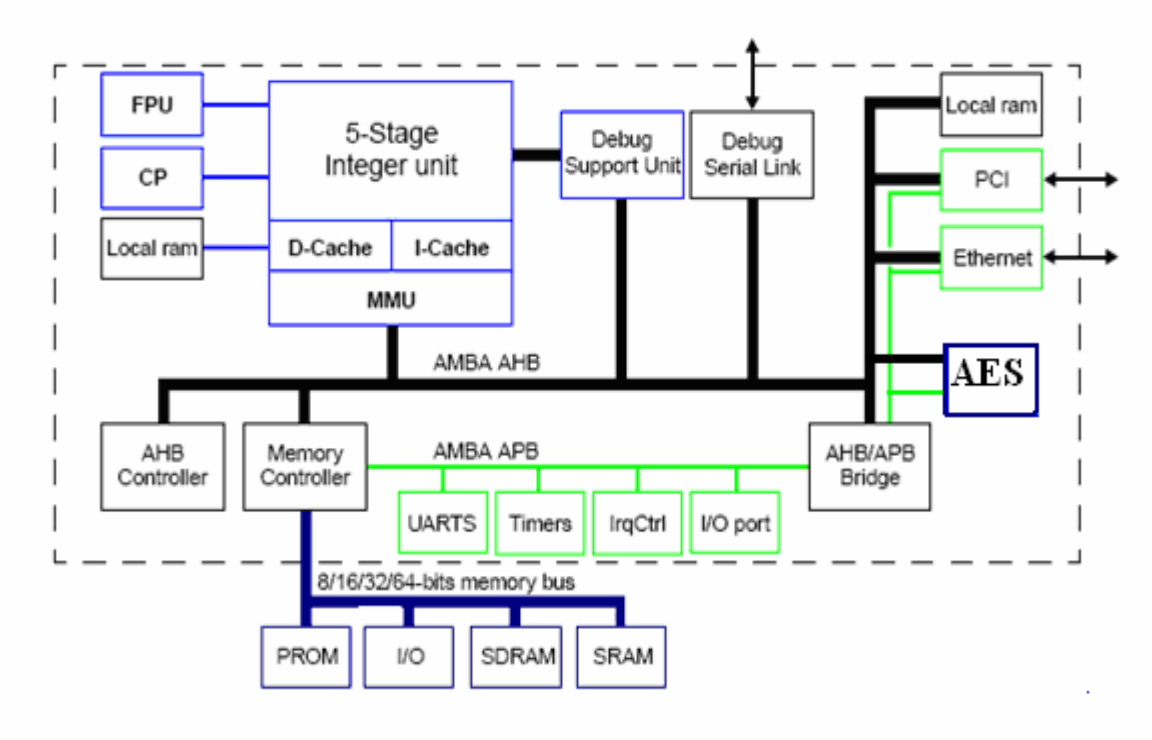


Figure 3.7: SOC Platform Used for Physical Compilation

# Chapter 4: Design Flow Components

## 4.1 Galaxy Reference Flow

The Galaxy Reference Flow (GRF) is a reference design flow that allows a Register Transfer Level (RTL) design to be taken through physical design using an automated set of customizable scripts [21]. It can be used to implement the best methodologies and harness the features of the below main tools.

1) Design Compiler for synthesis and DFT insertion using DFT Compiler
2) JupiterXT for generating multiple floor-plans, macro placement and power planning
3) Physical Compiler for placement and optimization
4) Astro for Clock Tree Synthesis (CTS) and routing
5) IC-Compiler for placement, CTS and routing

The IC-Compiler tool can be used to replace the Physical Compiler and Astro tools in the design flow. It also can be used to work with additional helper tools such as Formality for formal verification, Prime Power for gate-level power analysis, Star-RXCT for extraction and Prime Time for static timing analysis and sign-off at various stages of the design flow. The GRF can be used in either the full or stand-alone modes. The full mode consists of multiple tools including helper tools while the stand-alone mode is used for working with a single tool as a subset of the Galaxy flow. As part of this thesis, we used version 2.1 of the GRF with version 2005-09 of the main tools. The GRF requires the Milkyway database for the libraries, which can be used by all the tools and hence they needed to be created prior to the first usage.

## 4.2 Design Compiler

Design Compiler (DC) is a standard synthesis tool from Synopsys which is widely used across the Application Specific Integrated Circuit (ASIC) industry. It is used to convert the Hardware Description Language (HDL) code into an optimized gate-level net-list. The HDL files in our case were read in using the Presto HDL compiler which is the newer version and the only 64-bit implementation for the HDL compiler.

The HDL code is first mapped into a technology-independent generic library (GTECH) which consists of basic logic gates and flip-flops and the DesignWare library which contains complex cells such as adders and comparators [22].The net-list is then optimized to achieve minimum area and to comply with user-defined constraints while mapping the logic gates to those defined in the technology library. It has the ability to perform power optimization using clock gating through the use of Power Compiler and Design for Test insertion using DFT compiler.

The traditional way of estimating interconnect was through the use of statistical wire-load models which could be used to apply a certain model based on the area of the design block. Hence while writing the constraints for the tool they were about 15-20% more stringent than the required implementation constraints. This is followed by performing the physical implementation of the design and then re-synthesizing the design using the back-annotated parasitics. This was followed by using the physical database as an input while performing synthesis.

The latest Design Compiler Ultra uses topographical technology which it shares with IC-Compiler as part of synthesis to build a virtual layout and avoids the use of any wire-load models. This helps achieve a logic net-list that has a good correlation in timing and area with a post-layout net-list. It has the ability to read

in a floorplan or can use floorplan constraints given by the user. The ability to use a virtual layout allows better scan chain reordering and helps reduce congestion. DC-Ultra also has the capability to perform virtual clock-tree synthesis which provides more accurate power estimation [23]. The physical library information is provided in the Milkyway Database format.

## 4.3 JupiterXT

JupiterXT is a design planning tool from Synopsys that allows a fast exploration of the design and can obtain a detailed and optimized floorplan for both flat and hierarchical design styles [24]. It can apply its virtual flat placement algorithms to gate-level net-lists to place macros and standard cells simultaneously. “It shares its placement, routing and timing analysis engines with Physical Compiler and Astro ensuring faster convergence in obtaining correct floorplans” [24]. As part of its Virtual Flat Placement (VFP), we can use the explore mode which generates multiple output views that can be analyzed prior to deciding the final location of the macro placement. JupiterXT can also use In-Place Optimization to provide an early assessment of whether timing can be met.

The Power Network Synthesis (PNS) capability can be used to estimate the electromigration and voltage drop problems that can be encountered later in the tool flow and avoid signal integrity issues. This can be used after the die size is determined to formulate the power plan structure involving power rings and straps so that the design meets the power budget requirement. However, power pad information is crucial when running PNS [25].

The tool has an additional command *search_die* which can be used after the initial floorplanning and virtual flat placement to search different die size estimates based on the core/cell utilization and determine the minimal die size for

a design net-list. If the design is easy to route, it reduces the die size else if the design is not routable the core size is increased [24].

JupiterXT can also be used to perform clock planning and then estimate the insertion delays and skew of clock nets. Finally it can perform prototype global routing to estimate the wire-length and routing resource usage and check whether the design is routable.

## 4.4 Physical Compiler

Physical Compiler is the physical synthesis tool from Synopsys which integrates both placement and synthesis to achieve quicker convergence, thus reducing the design cycle time as the iterations between synthesis and placement are reduced. It extends synthesis by performing location based optimization and timing driven placement together [26]. It uses Steiner routing as part of its routing estimation to ensure better timing information and it also reduces routing congestion.

The parasitic RC (Resistance Capacitance) for interconnect is estimated through the use of TLU+ models, generated using STAR-RCXT an extraction tool from synopsys. TLU+ contains resistance and capacitance look up tables and model ultra deep submicron process effects. The Distributed Physical Synthesis (DPS) capability allows a large design to be automatically split into partitions and run simultaneously on multiple systems to achieve quicker results.

A floorplan is required for the physical synthesis tool and can be read in using either the Design Exchange Format (DEF) or the Physical Design Exchange Format (PDEF). Alternatively, a basic floorplan can be created using its RTL performance prototyping capability through its minimum physical constraints [27].

Thus Physical Compiler can achieve a placed-gates design from either HDL code or a gate-level netlist.

Initially as part of DFT insertion, scan chains are connected based on their instance names which can cause the chain length to be longer than required and result in congestion if two nearby instances are placed far apart. Physical Compiler combined with DFT compiler has the ability to disconnect these before placement and re-stitch the scan chains based on order or placement. This reduces the scan chain length and also helps improve congestion.

It can perform power optimization as it is integrated with Power Compiler. This can be done through the use of clock gating. Another approach is the use of Multi-Threshold Voltage cells for reducing the leakage or static power. Thus low threshold cells which have a large leakage current but switch quickly can be used for timing critical parts while high threshold cells are used for the non critical paths ensuring power savings. It can perform dynamic power optimization when switching activity information is provided.

## 4.5 Astro

This physical implementation tool from Synopsys can perform placement and optimization, Clock Tree Synthesis and Routing [28]. It can address affects such as crosstalk, IR (current-resistance) drop and electromigration. Astro can perform distributed routing and hence reduce implementation time. It can also handle multiple voltage designs during implementation. It also uses TLU+ models to address ultra deep submicron process effects [29]. Astro also has the capability to generate a floorplan and perform power planning.

The methodology recommended by Synopsys for Astro allows the user to setup an automatic flow through the use of customizable scripts [28]. Astro can also

perform placement-based scan chain re-ordering which reduces the scan chain length and improves congestion. Astro can initially remove wire-load model effects by downsizing gates and removing buffers along paths with positive slack and then uses virtual Steiner routing to estimate timing and optimize the net-list. During congestion analysis after placement, it can use global routing to check the demand for wire tracks and achieve more accurate estimation.

After placement, Astro can perform power optimization by removing and sizing the buffers. Astro then uses Clock Tree Synthesis (CTS) to build clock trees for minimum skew and if required can use its useful skew optimization feature to increase or decrease the size of clock buffers in the positive slack paths to meet setup timing. It has a clock tree browser for viewing the clock tree structures [28].

After CTS, Astro can perform routing, followed by search and repair to fix design rule violations, post-route optimization and finally Engineering Change Order (ECO) routing. Once routing is completed it can also be used to optimize the design by reducing the wire-length and the number of vias. Astro can perform design rule checking for designs above 130-nm using a subset of the design rules used by the Synopsys Hercules tool and uses advanced design rule checking for designs below 90-nm.

# Chapter 5: Implementation

## 5.1 Milkyway Library Creation

The required details to setup a Milkyway Library have been obtained from the Synopsys Milkyway Manual [30] and Synopsys Solvnet Articles [31]. The library creation was done using Astro. The basic files for any technology are given below:

a) *Layout Exchange File (lef)*
b) *Technology File (tf)*
c) *lef_layer_tf_number_map.pl* –script that maps the lef file and the tech file to find the Milkyway layer number for a particular layer in the lef [31]. The Milkyway layer number is usually defined in the tech file. This prints out only the routing layer and poly numbers but we can associate other members as the script prints them out from the tech file earlier.

The process for creating a Milkyway Library in brief using Astro is given below:

a) Choose *Data Prep* in the *Astro menu* and then *Create Library*. In the form enter *library name, technology file name* and remember to *set the case sensitive* option.
b) *Open* the *Library* and in the scheme command enter *read_lef*. This is an automated series of steps in the library creation. In this form enter the *name of the tech lef file* (if the lef file contains technology information), *cell lef file* (lef file name) and the *layer mapping file* obtained using the perl script. The series of steps involved are Extracting blockage, Pin and via (to create the abstracted view), setting P & R boundary and finally defining wire tracks. *If* the *wire tracks* information is *not present* enter information

about the metal layer offsets, metal directions (*axgDefineWireTracks*). Finally one can check the wire track information (*axgCheckWireTrack*).

c) Attach the Logic Models (.db) files using *read_lib* command and then selecting logical. Then browse and select the maximum, typical and minimum logic model files to import.

Additional files used are the *Tluplus* models which can be generated using the *Interconnect Technology File (itf)* which can be extracted from the *.nxtgrd* file used as input for Star-RCXT. Additionally a *mapping file between the tech file and the itf file* is to be created by the user for the below two headings:

conducting_layers

via_layers

Then using the *grdgenxo* script available with Star-RCXT one can generate the models. To generate models for a particular corner instead of typical we just need to include a *one line format file* with the –f option of grdgenxo which just *mentions the operating condition* (ex: OPCOND MAX).The usage is given below:

*grdgenxo -itf2TLUPlus -i <itf_file> [-f <format_file>] -o <TLUPlus_file>*

Once the *Tluplus models* have been generated they can be *attached* to the *Milkyway library* by *including the mapping file*. Thus the reference library created as a Milkyway database can be used with the JupiterXT, Physical Compiler and Astro tools.

## 5.2 Artisan RAM generation

As discussed in Chapter-3 we require synchronous RAMs to implement our register file and caches. These are generated using the Dual and Single port RAM generators from Artisan. The required parameters to generate them are provided in Table 5.1.

Table 5.1: Artisan RAM Generator Parameters [13]

| Field | IU Rams | Cache Ram | Tag Ram |
|---|---|---|---|
| Instance name | dpram136x32_inst | ram256x32_inst | ram32x30_inst |
| Words | 136 | 256 | 32 |
| Bits | 32 | 32 | 30 |
| Frequency (MHz) | 50 | 50 | 50 |
| Multiplexer | 4 | 8 | 4 |
| Library Name | DPRAM1 | RAM3 | RAM2 |

For each of the above block memories we generate the verilog model, tlf, synopsys lib models which will then be converted into Synopsys db files and the vclef footprint, used to create the physical Milkyway database.

## 5.3 Leon Processor Configuration

The processor can be configured using a graphical configuration tool based on linux kernel tkconfig scripts [11]. As part of the configuration we use caches of size 1KB with line size of 8 bits. We also do not use the multiplier and divider. As we will target the IBM7RF 180-nm process, we have to modify the target technology in the generated *device.vhd* file which contains the configuration. Also we change the number of masters on the AHB bus to two while having the Leon processor as the default master. The IBM process is added in the *target.vhd* file as a target technology. Additionally a file that had been created previously [15] and contains information about the RAMs and I/O required for the IBM7RF process is used. Also, since we decided to implement the SoC without any pads in this thesis, the *leon.vhd* was modified to remove the I/O pad instances and the clock generator since we are not dividing the clock. Then the makefile required for simulation was modified accordingly to use the appropriate Leon and AES files.

## 5.4 Design Flow

The SoC implementation was done using a flat flow instead of a hierarchical one. Functional simulation was done using ModelSim at the pre-synthesis, post-synthesis and post-layout stages. At the post-synthesis and post-layout stages, SDF back-annotation was done. Design Compiler was used for synthesis, Jupiter-XT for floor-planning, power planning and macro placement, Physical Compiler for physical synthesis and finally Astro for placement, CTS and routing. To compare the results with and without the presence of the physical synthesis tool, two paths were taken in the design flow. In one path, placement combined with synthesis (Physical Synthesis) is done using Physical Compiler. As part of the second flow, placement involving net-list optimization was done in Astro. In both of the flows, Clock Tree Synthesis and Routing are done in Astro. The design flows implemented are shown in Figure 5.1.

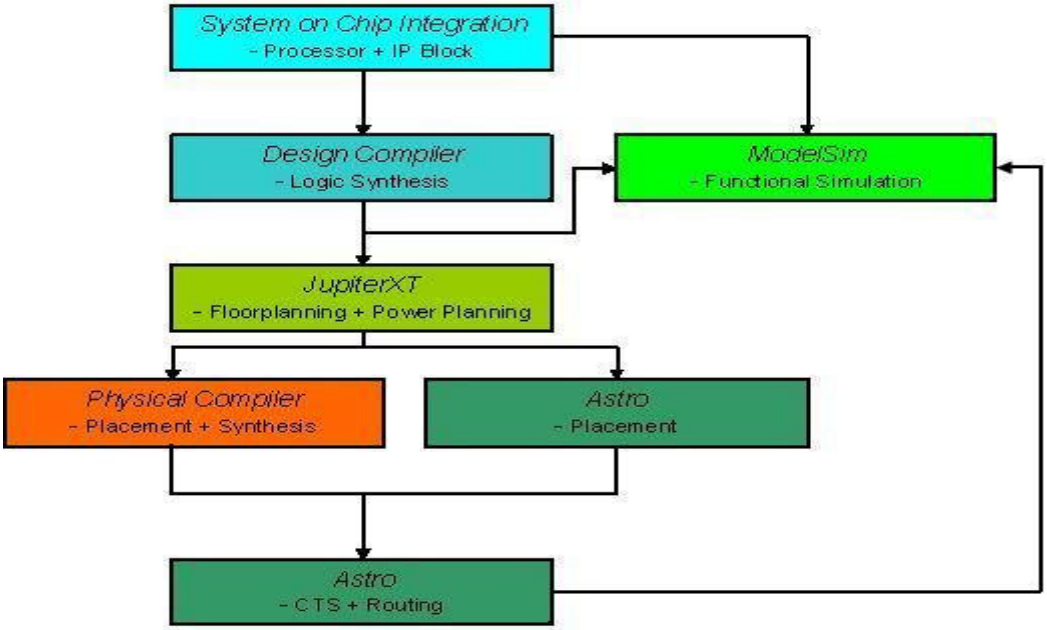


Figure 5.1: Design Flow

### 5.4.1 Synthesis

As part of synthesis, the operating condition was set for both the maximum and minimum conditions. The synthesis script was customized from the one provided with the Leon processor [11]. The target library for the design during synthesis was the slow corner library. Initially the RAM VHDL interfaces were synthesized using the RAM db files. Then the Leon processor was synthesized using these interfaced Verilog files. We had to enable the *use presto* variable to read in the VHDL. Initially, synthesis was attempted for a frequency of 111.11 MHz but timing was not met. Hence, the design was targeted for a frequency of 100 MHz. The Standard Delay Format (SDF) file and Verilog net-list were produced by Design Compiler. This SDF was then modified using a perl script provided by Artisan so that it could be simulated with the Artisan Verilog models. The back-annotation and simulation are shown in Figure 5.2 and Figure 5.3.

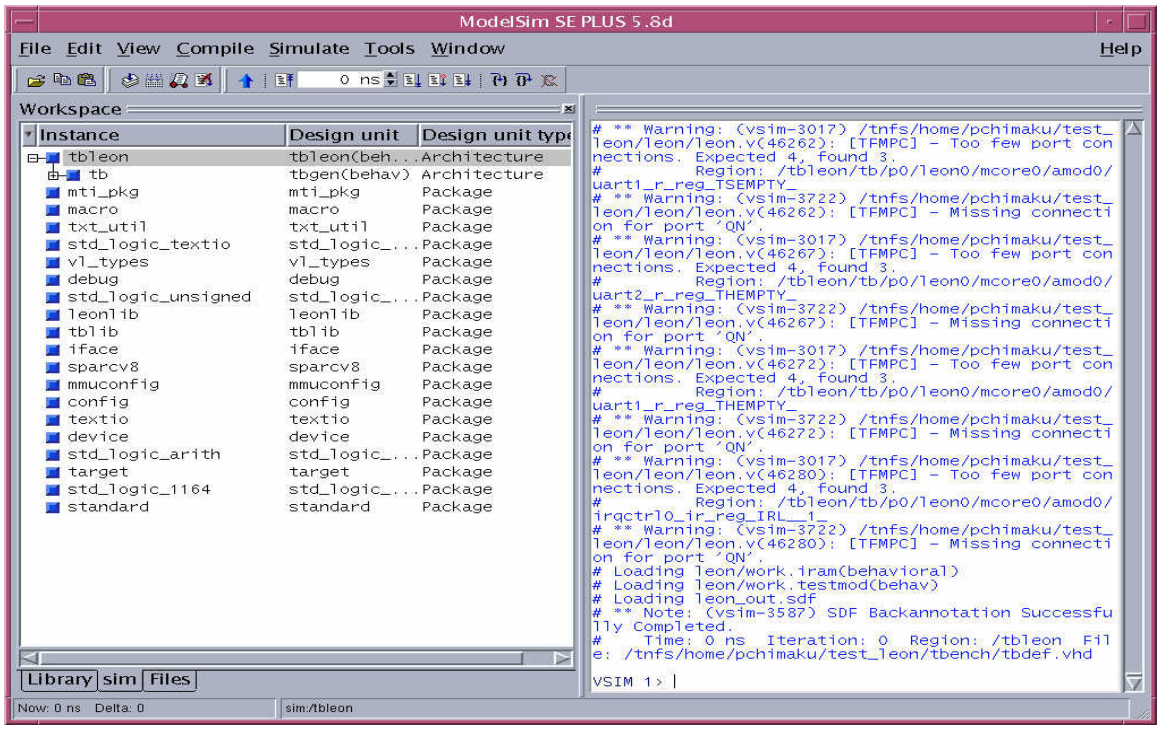


Figure 5.2: Post-Synthesis SDF Back Annotation

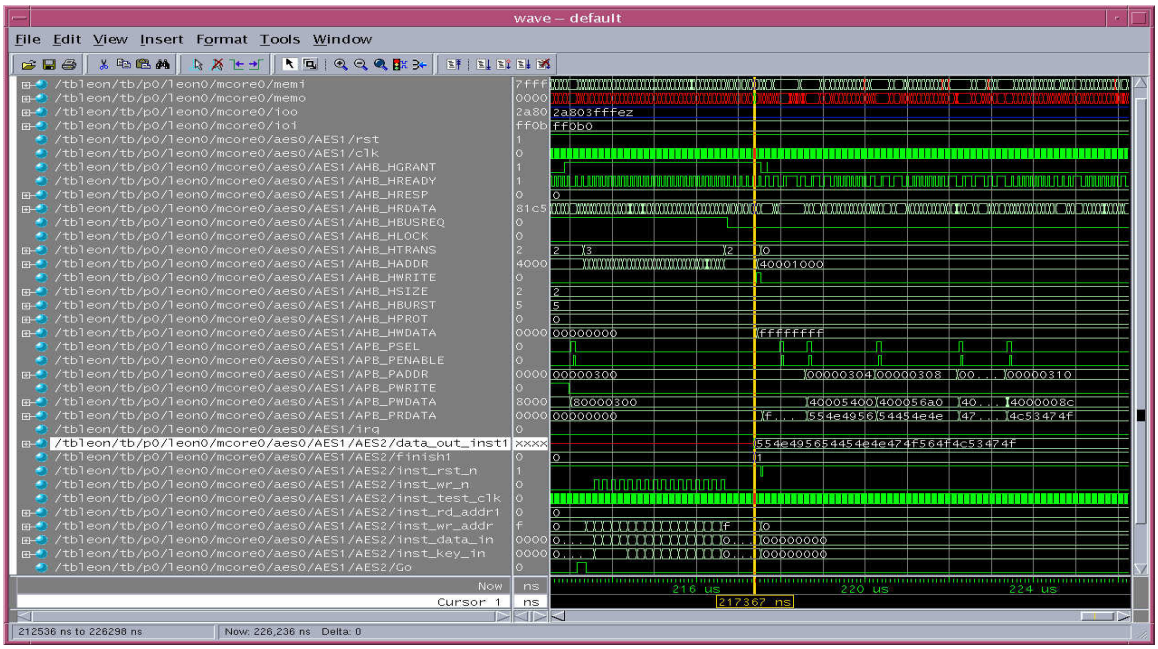


Figure 5.3: Post-Synthesis Simulation

### 5.4.2 Physical Implementation

From this stage we used the Galaxy Reference Flow scripts for the remaining implementation. Hence the steps to setup the GRF scripts are listed below [21].

1. Untar the downloaded file [*gtar xvfz G.tar*]
2. Copy the modified *setup.csh* which points to the tool executables and licenses.
3. The *.grf_reference_files/ASTRO/starxt/cmd* directory is missing the file named *14.extract_handoff_sta_starxt.txt* as the flow command language file in the *Astro.fcl* file requires this to proceed with the setup. Copy it from the extracted directory into the above location.
4. Source the *setup.csh* and then use the below command to setup the directories for any or multiple modules:

*grf_utilities/grf_setup module*

5. Once the tools are setup using the *grf_utilities/grf_setup* command, one has to select variables for the tool. A list and description for the variables associated with each tool in the GRF can be obtained from the help using the command *gh –var variable name*. The suggested workflow from the GRF User Guide is to setup the initial variables (*gep –i*). After that I would use (*gep –t all*) to setup other variables.

6. Use the *project_setup* file provided for each tool. If a variable is defined in the *project_setup* file, it is used instead of the default value in *project_setup.defaults* file.

7. Hence to implement the above design flow, four GRF directories will be required. Two directories will be used for JupiterXT, the third for Physical compiler and Astro while the last is just for Astro.

### 5.4.3 Floorplanning and Macro Placement using JupiterXT

1. Create the GRF directories for JupiterXT using *grf_utilities/grf_setup jxt*

2. Copy the *4.initial_place_jxt.scheme* and *6.feasibility_jxt.scheme* given into the *cmd.user* directory of the JXT directory

3. Copy the *project_setup* file into the *GRF2.1.release.without_examples* directory of Jupiter. The only difference between the two provided *project_setup* is the values of height and width for the core. In case of the physical compiler and astro flow, the core is defined as 1425µ x1425µ while for the astro flow it is 1485µ x 1485µ.

4. Copy the netlist file *leon.v* and the design constraints *leon.sdc* files from the design compiler directory into the *JXT/input_data.*

5. Running the *gmake default* command in the *JXT* directory will perform the following steps:

a) Create a *Milkyway Library* for the design and a *.scheme* file with tool variables

b) Generate a floor-plan for the design

c) Perform an initial and incremental placement so that the design can be placed and ensuring that the floorplan parameters are sufficient.

d) As part of the feasibility script, we will create the power structures and also perform a trial CTS (clock tree synthesis) and routing without modifying the placed design. Finally the floor-plan is dumped out with only the macro cells and power structures. The floorplans used are shown in Figure 5.4 and Figure 5.5.

The difference from the GRF scripts in this section was that the congestion driven option was chosen for the virtual flat placement (VFP) in JupiterXT. Initially while iterating with the explore mode that provides multiple floorplan candidates, congestion driven would always give the best results in our case compared with the other options. Additionally Power Network Synthesis which can estimate power requirement and build the power rings and stripes was not used.

The power rings for macros were built according to the floorplan and the commands to build them were added into the scripts. The approach for floorplanning involved decreasing the utilization value from 95% until placement could be achieved. Using information from a solvnet article to improve routing performance in Astro, the Global Route Cell overflow information after placement which helps analyze congestion was used to decide which floorplan can be used to proceed further. The above information is available in the place and route summary created by the GRF in the *JXT/report* directory after incremental placement namely *incremental_place.sum*.

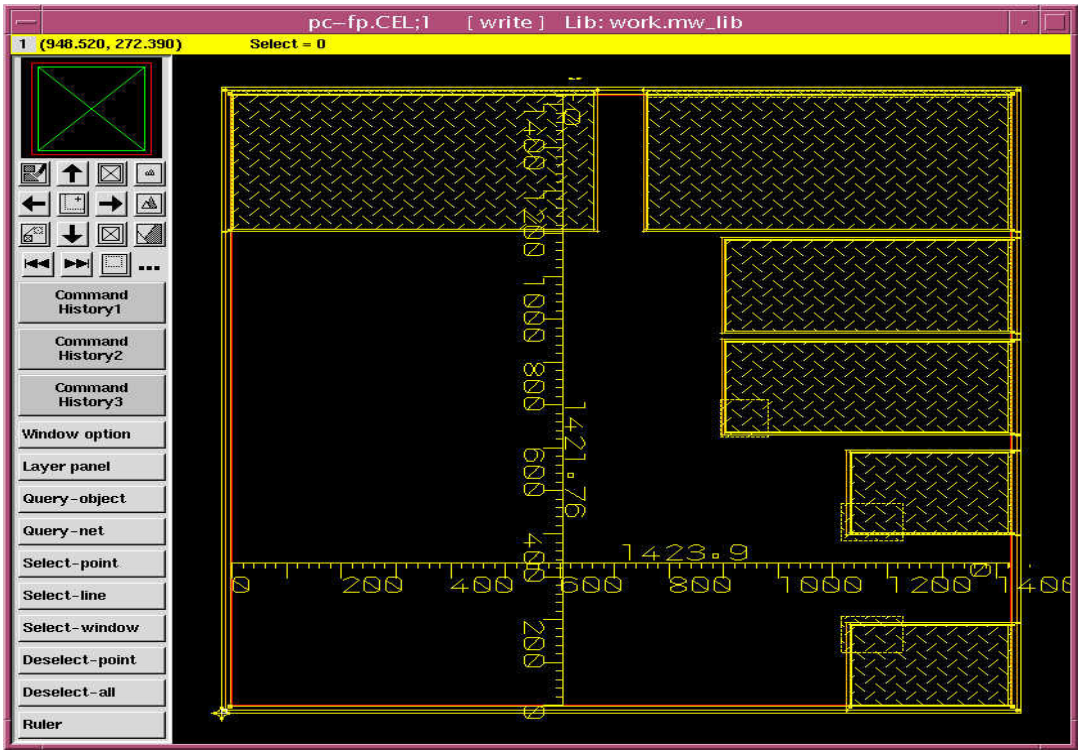


Figure 5.4: Floorplan for the Physical Compiler Tool Flow

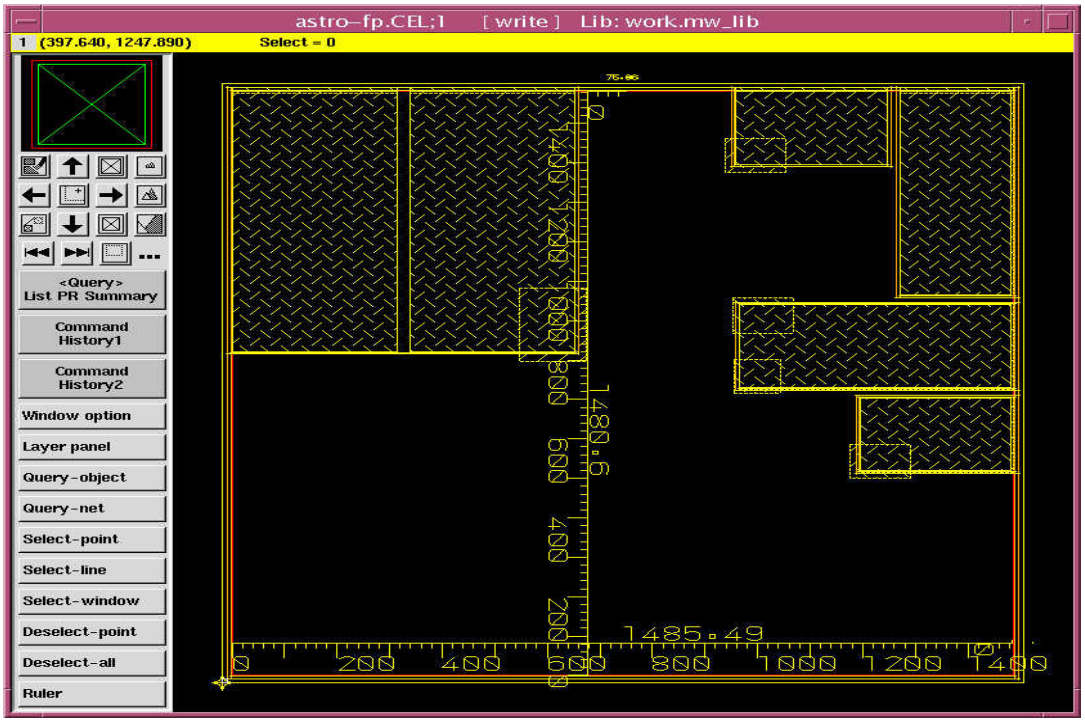


Figure 5.5: Floorplan for the Astro Only Tool Flow

### 5.4.4 Implementation using Physical Compiler and Astro

1. Create directories for Physical Compiler using the command

*grf_utilities/grf_setup pc astro*

2. Copy the given *2.physopt_pc.tcl* into the *cmd.user* directory of the *PC* directory. The *cmd.user* directory is first checked for any customized user scripts before scripts are read from the default *cmd* directory.

3. Copy the *project_setup* file into the *GRF2.1.release.without_examples* directory of Physical Compiler.

4. Copy the netlist file *leon.v* from the design compiler directory and the provided *leon.sdc* constraints file for Physical Compiler into the *PC/input_data.*

5. Running the *gmake default* command in the *PC* directory will perform the following steps:

a) Create a *Milkyway Library* for the design after reading the floorplan as shown in Figure 5.6.

b) Perform Physical Synthesis and optimization which involves Automatic High Fan-out Synthesis to rebuild the buffer trees. The placed design is shown in Figure 5.7.

c) It can perform placement based scan optimization and Power Optimization as part of this flow by setting the following variables namely *ENABLE_POWER_OPT* and *ENABLE_SCAN_FLOW* to true. However we do not do either as part of this SOC flow. However the scan flow was used to create a test-ready design for another internal tutorial.

d) Writes out the verilog netlist *leon.v* as the data mode is set to ASCII, and the placed design *leon.def* in the *PC/output_data* directory

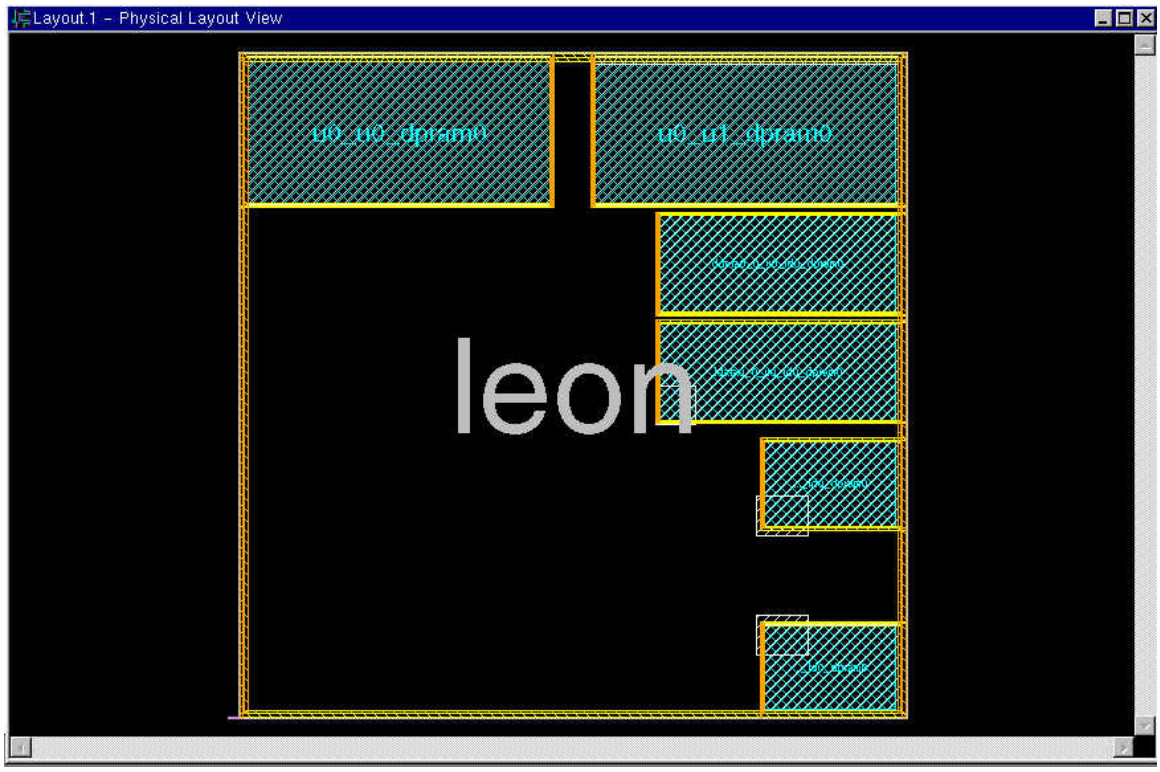


Figure 5.6: Input for Physical Compiler

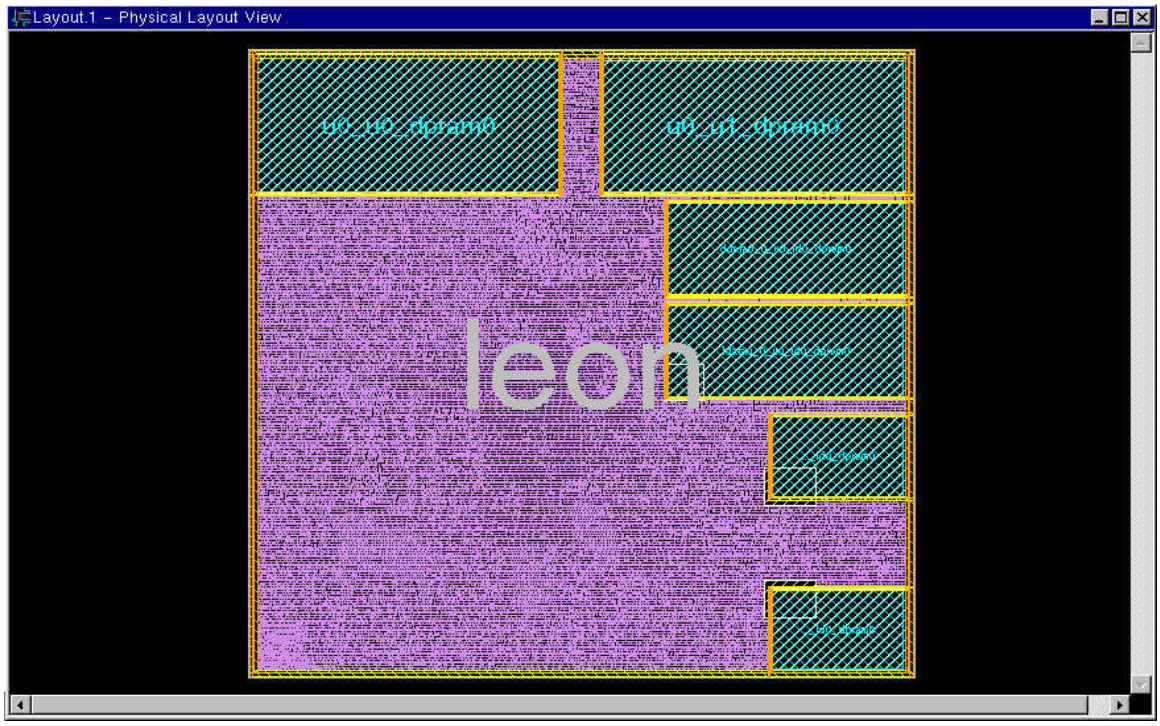


Figure 5.7: Output from Physical Compiler

6. In the top directory, namely *GRF2.1.release.without_examples*, run the following command to propagate the input data for astro: *gp astro*

7. Copy the provided design constraints file for astro *leon.sdc* into *astro/input_data*.

8. Copy *6.route_design_astro.scheme* and *7.post_route_astro.scheme* into the *astro/cmd.user* directory

9. Running the next command in the astro directory will perform the following functions: *gmake write_verilog_astro*

a) Creates the Milkyway view for the design as shown in Figure 5.8 and a .scheme file with the tool variables

b) Repairs the hierarchy and avoids placement as it has been done in Physical Compiler

c) Performs Clock Tree Synthesis to achieve minimum skew and then optimizes to correct hold time violations and meet setup timing. The distribution of the clock before and after clock tree synthesis as shown with the clock browser is shown in Figure 5.9 and Figure 5.10.

d) Once Post-CTS optimization is done, routing is performed in the following sequence. First the clock nets are routed followed by global routing during which nets are assigned to metal layers and global routing cells. This is followed by track assignment during which metal traces are laid down. After this detail routing is done to fix the DRC violations by working on fixed size segments called Sboxes and then Search and Repair fixes the remaining DRC violations by varying the size of the S-box [32]. Finally post route optimization and Engineering Change Order Routing is performed. The routed design is shown in Figure 5.11.

e) Writes out the verilog netlist *leon.v* which we use for post-layout simulation.

10. Use the command *"load sdc-out.cmd"* after opening Astro to dump out the SDF file.

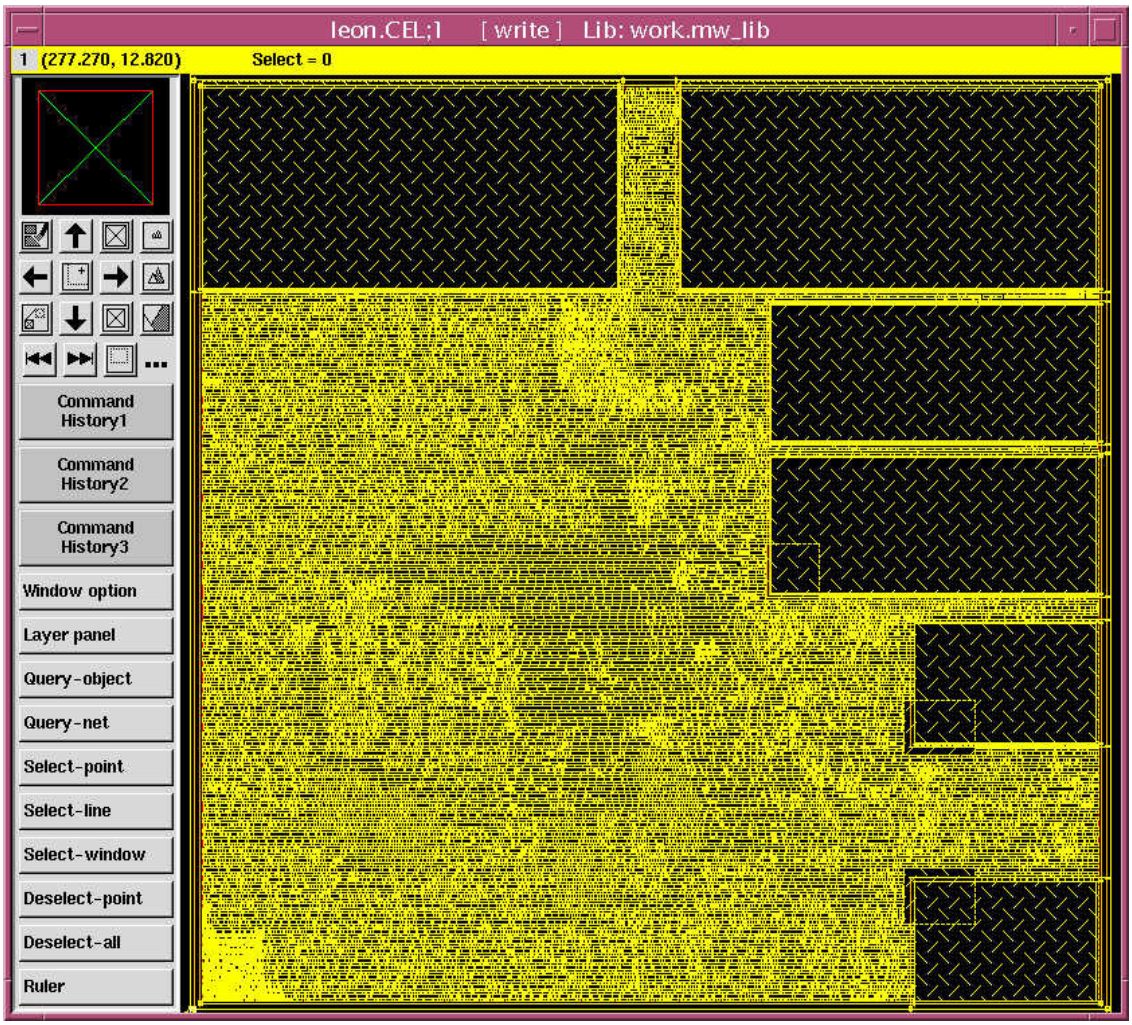


Figure 5.8: Input for Astro

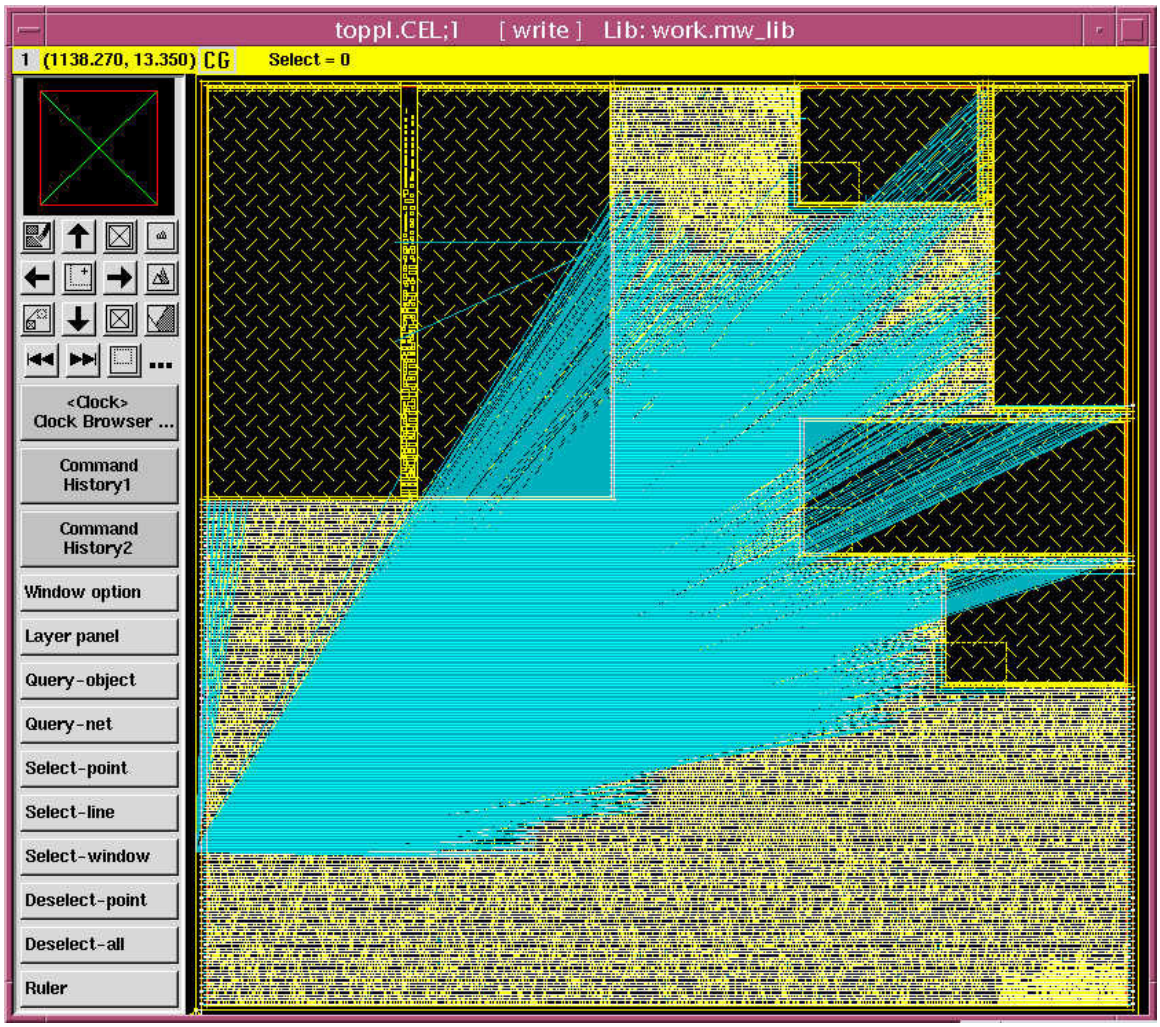


Figure 5.9: Pre-CTS Clock Distribution

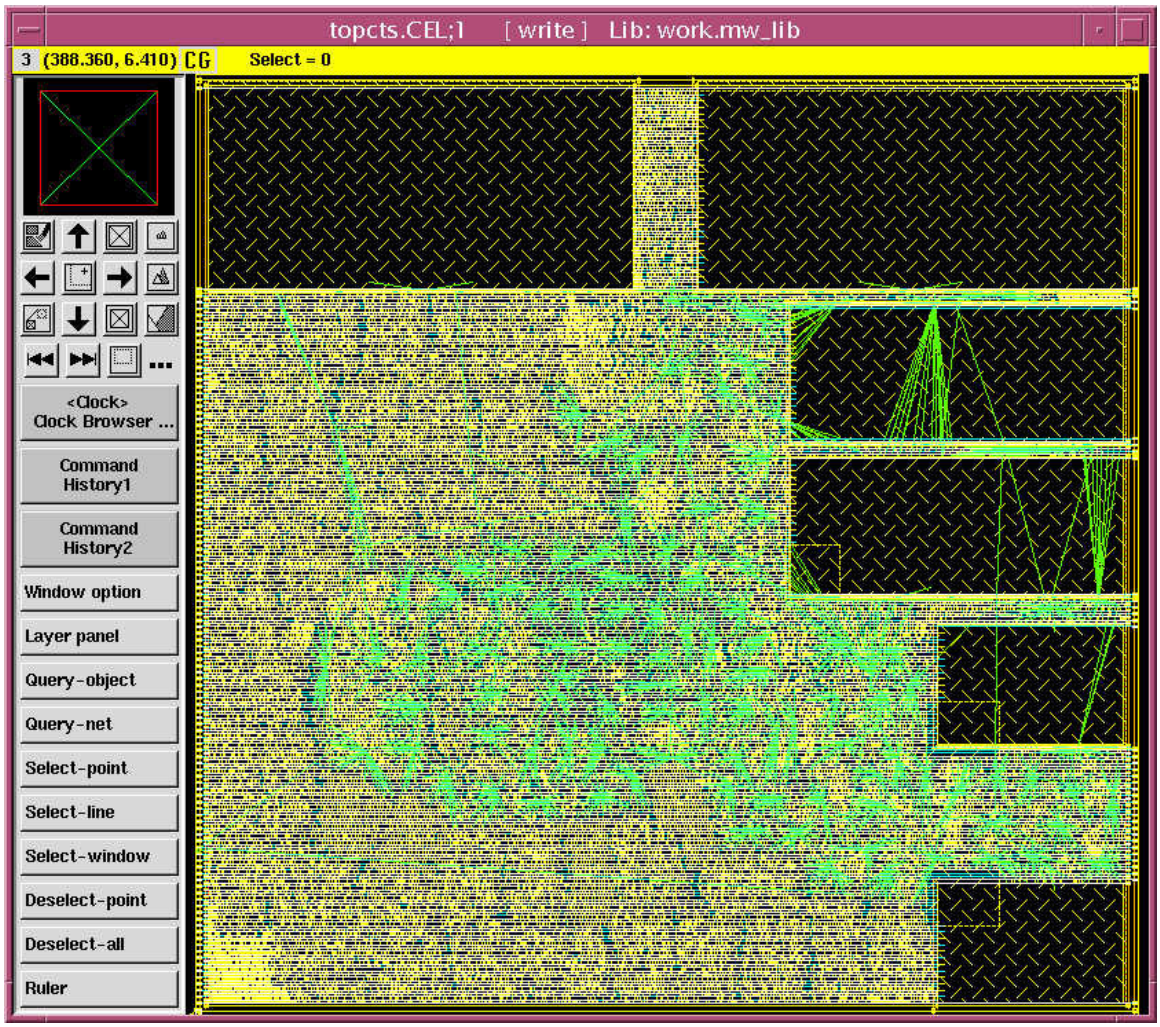


Figure 5.10: Post-CTS Clock Distribution

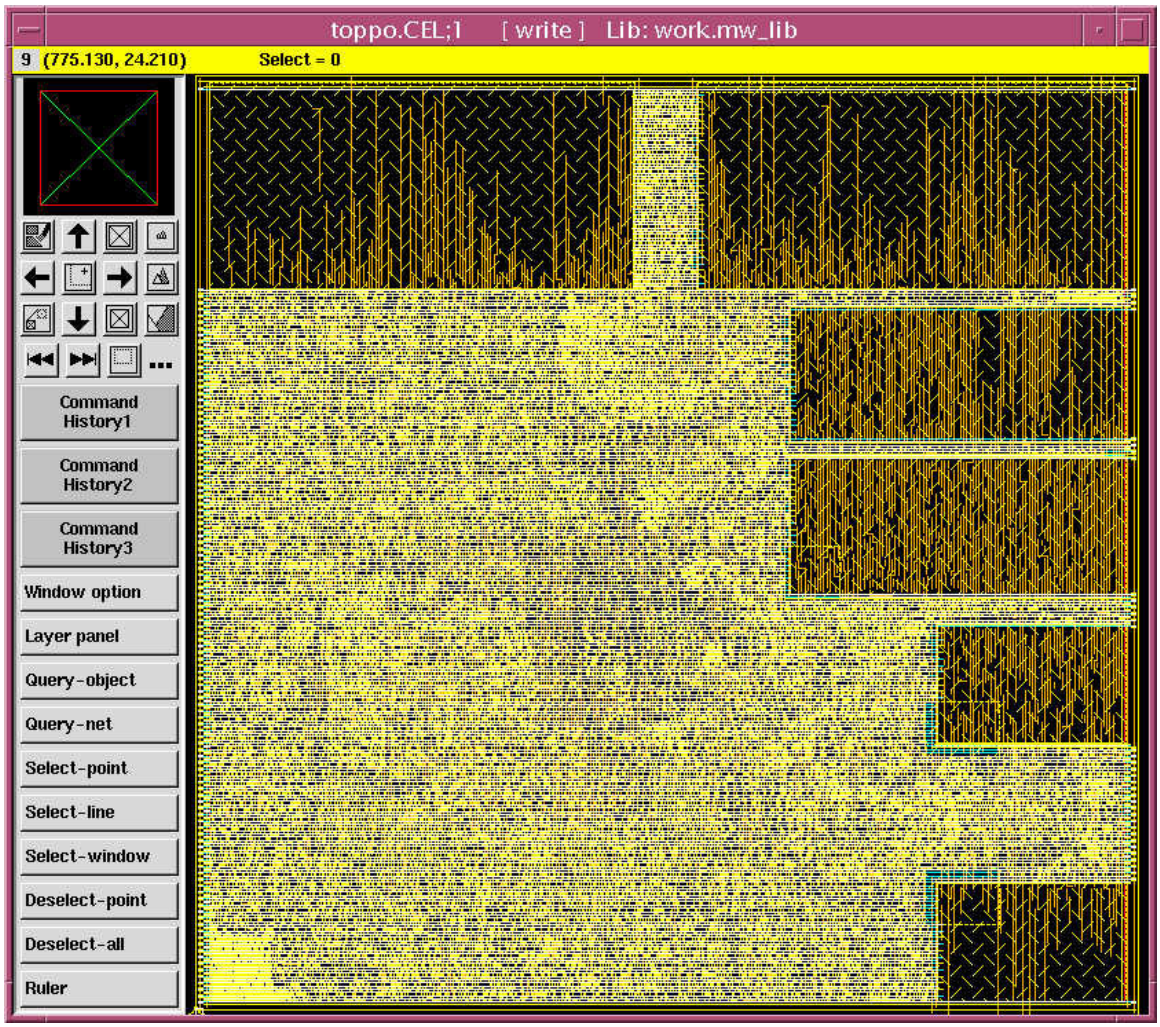


Figure 5.11: Layout after Routing in Astro

### 5.4.5 Implementation using Astro

1. Create directories for Astro using *grf_utilities/grf_setup astro*

2. Copy the provided files astro_setup.scheme, *6.route_design_astro.scheme* and *7.post_route_astro.scheme* into the *astro/cmd.user* directory

3. Copy the *project_setup* file into the *GRF2.1.release.without_examples* directory of Astro.

4. Copy the netlist file *leon.v* from the design compiler directory and the provided *leon.sdc* constraints file for Astro into the *astro/input_data.*

5. Running the *gmake write_verilog_astro* command in the *ASTRO* directory will perform the following steps:

a) Create a *Milkyway View* for the design after reading the floorplan and repairs hierarchy information.

b) Perform placement and optimization by initially removing the WLM effects by setting the RC to zero and again performing logic synthesis and removing buffers along positive setup paths. High fan-out buffer trees are built on the basis of a quick placement. It then performs placement on the basis of Virtual routing and does optimization to meet setup timing and also introduces buffers to meet the transition and capacitance constraints [32].The design after placement in Astro is shown in Figure 5.12.

c) Performs Clock tree synthesis followed by routing as in the earlier design flow.

d) The routed design is shown in Figure 5.13.

e) Finally the verilog netlist is written out and again the SDF is dumped out using the *sdc-out.cmd*.

In case of both the design flows a Design Rule Check (DRC) was performed, the rules used are a subset of the Hercules runset. The Error browser used to check for any errors in the layout implemented using Astro is shown in Figure 5.14.

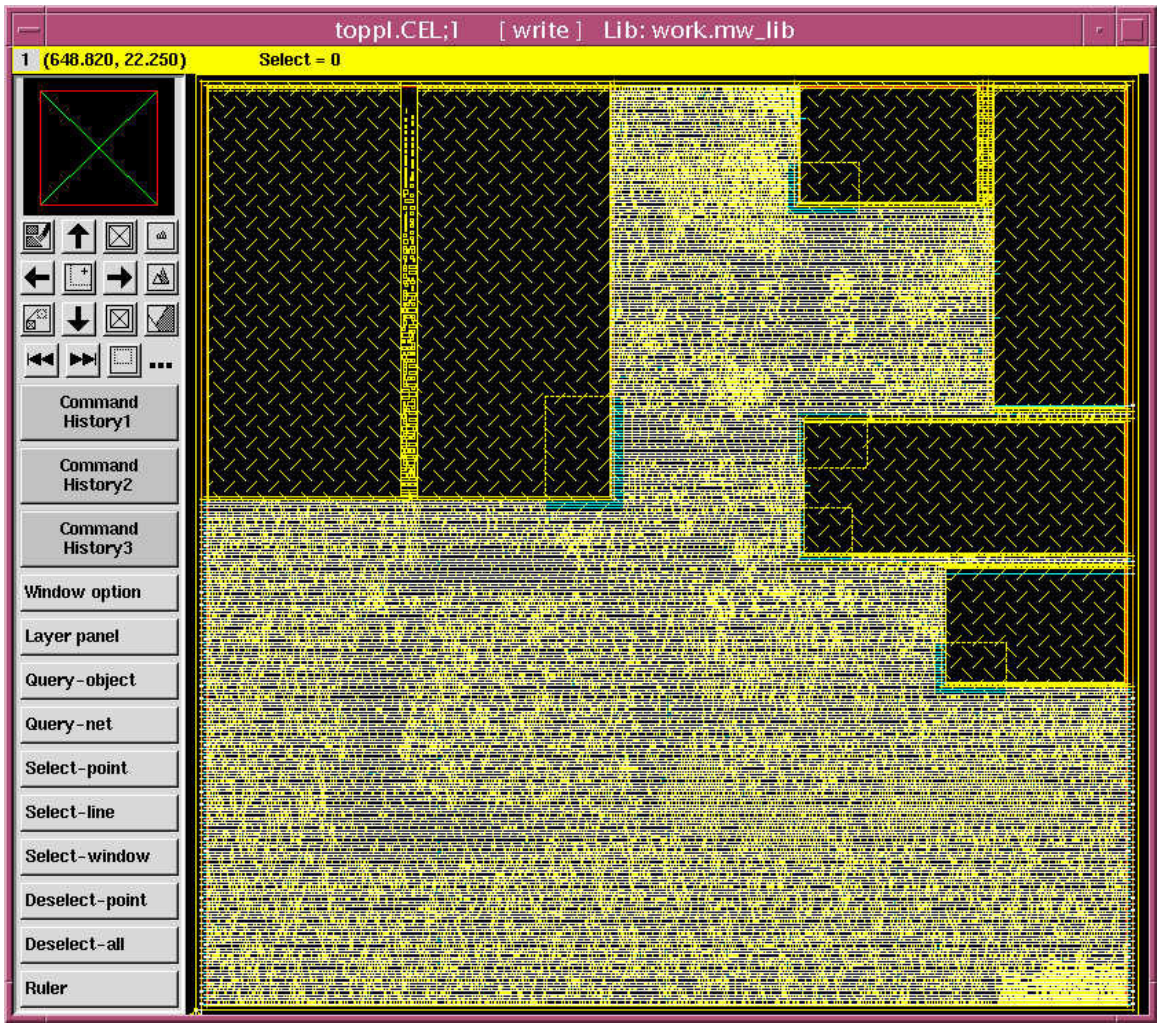


Figure 5.12: Placed Design in Astro Only Flow

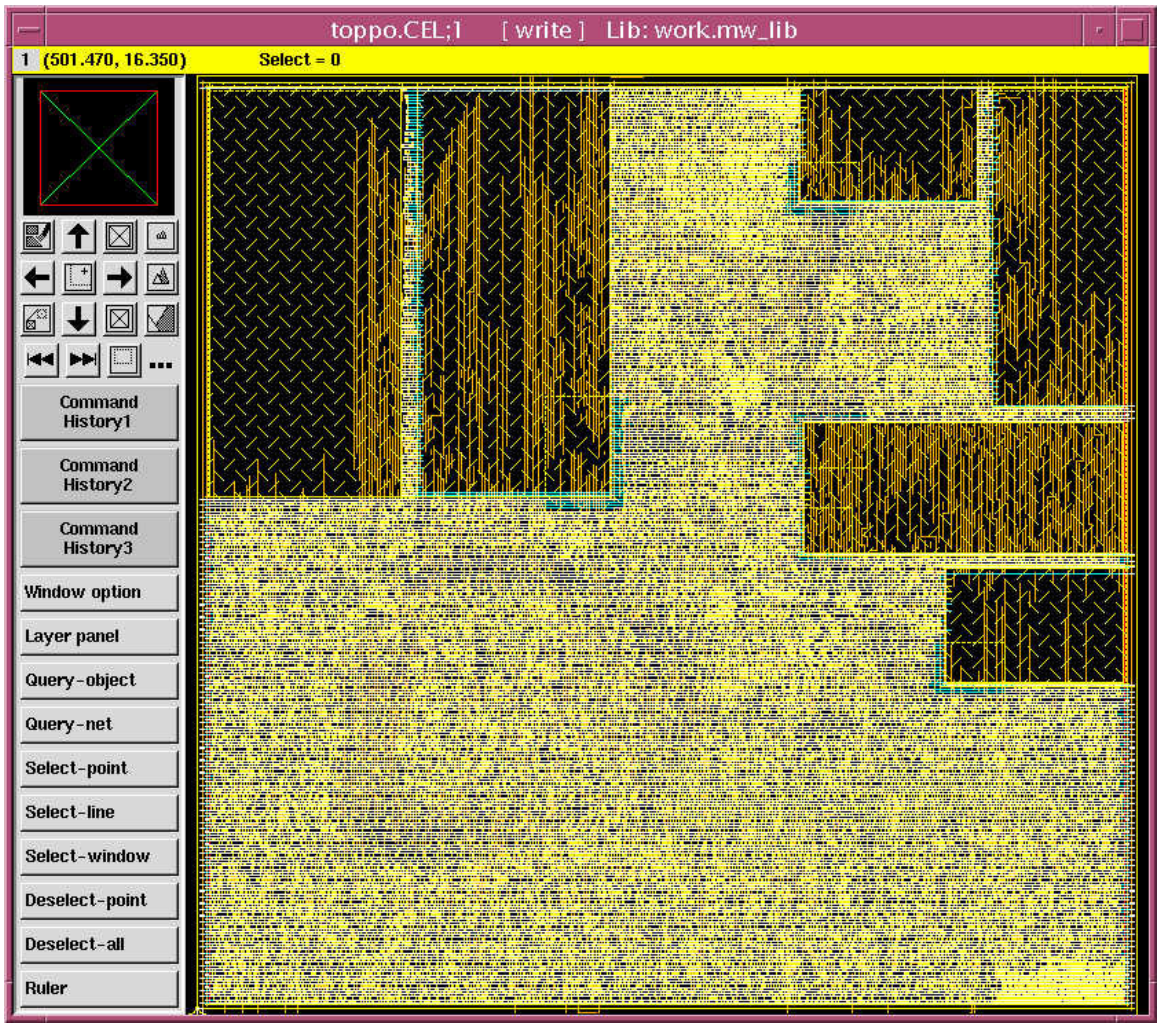


Figure 5.13: Routed Design in Astro Only Flow

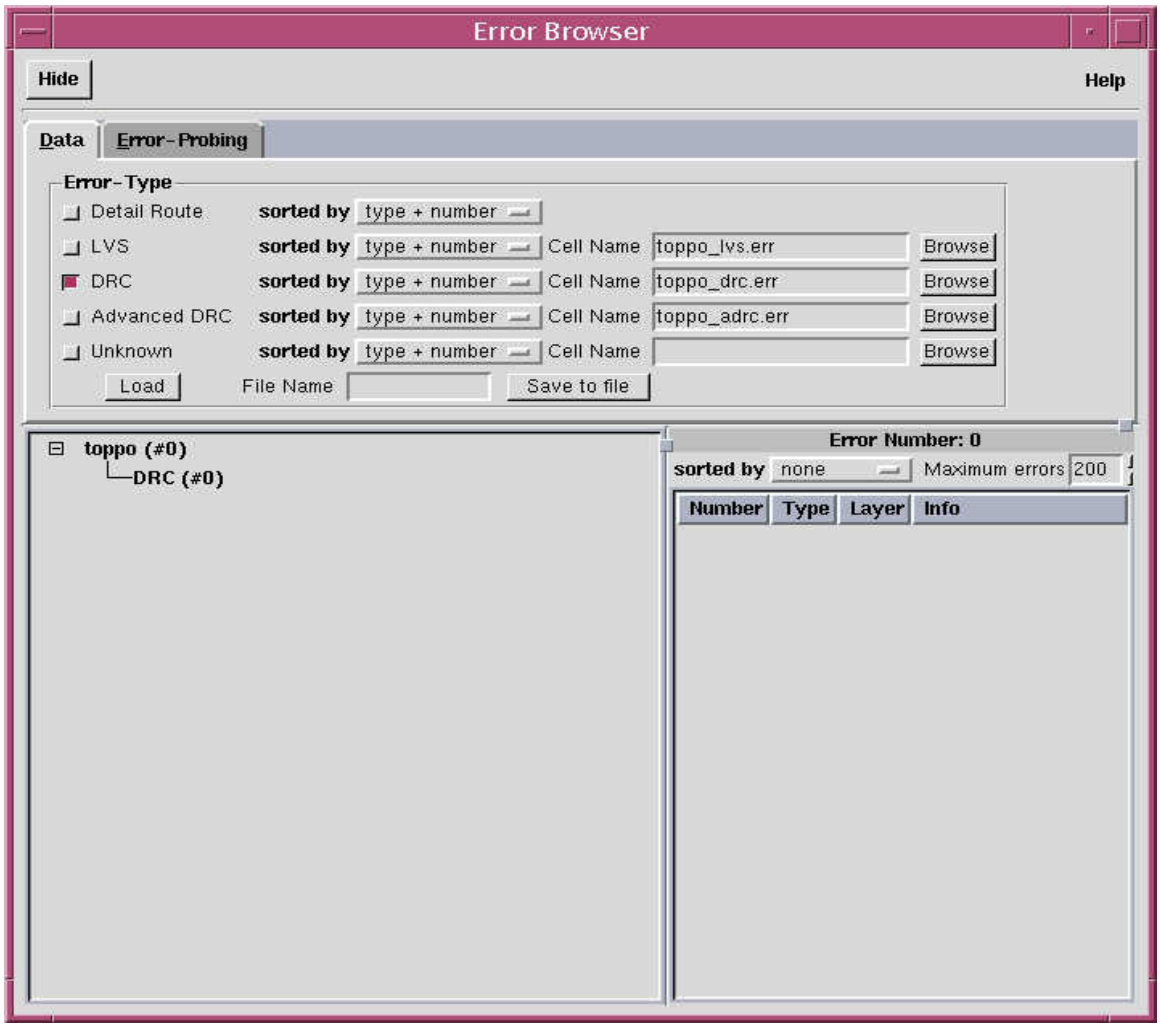


Figure 5.14: Error Browser to Check for DRC Errors

### 5.4.6 Post Layout Simulation

Once the SDF files were dumped from Astro, they were sent through the Artisan perl script. Finally the SDF was sent through another perl script that removes the RECOVERY timing check in the SDF as the timing check was not found in the verilog simulation files. Back annotation was successful for both maximum and minimum operating conditions. The post layout simulation is shown in Figure 5.15.

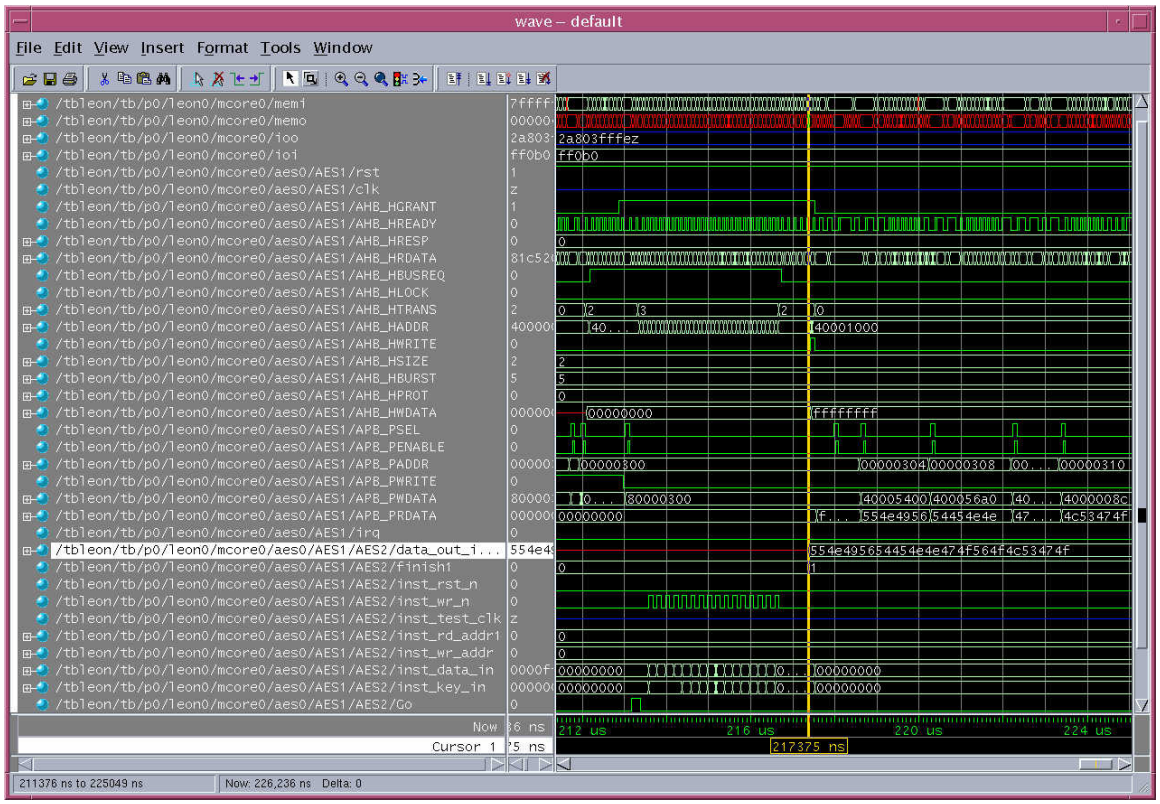


Figure 5.15: Post Layout Simulation with SDF Back Annotation

# Chapter 6: Conclusion

## 6.1 Results

As the physical design flow was being setup, an attempt was made to compare the two designs in terms of delay and area. The design could be implemented for a frequency of 100 MHz in both of the flows. Hence the only remaining parameter left was area. Also this meant there were no iterations between synthesis and physical implementation in our case. While the design implemented using physical synthesis used a total core area of ~2.02 $mm^2$, the second flow required a minimum area of ~2.2 $mm^2$. Hence the physical synthesis flow was implemented using 8% less core area than for the second flow. This also meant that the design implementation time was less for the physical synthesis flow as the second flow required a greater number of physical implementation iterations.

The total macro cell area comprised ~0.73 $mm^2$ for the four macros used. Additionally the physical synthesis flow with 34,613 standard cell instances used 372,506 transistors and required an area of ~0.82 $mm^2$ compared to the traditional flow which used 34,419 standard cell instances with 373,745 transistors and an area of ~0.83 $mm^2$. However, the total interconnect length for the physical synthesis flow was 2.7 mm compared to 2.47 mm for an increase of about 10%.

Hence the physical synthesis flow resulted in a design with reduced area and transistor count along with lesser iterations.

## 6.2 Conclusion

Thus the following tasks were completed as part of this thesis.

- Setup the physical synthesis flow using the Galaxy Reference Flow scripts from Synopsys.
- Created the Milkyway library for the IBM7RF technology.
- Implemented the physical and logic synthesis flows for a small design and for a system-on-chip.
- Compared the designs generated by the two flows.
- Prepared a tutorial to help universities setup and use the physical synthesis flow.

## 6.3 Future Work

It would be interesting to use the design implementation for much smaller feature sizes such as 90-nm and in a hierarchical approach. This would permit optimization of each block before integration and result in a more thorough flow comparison. Power optimization could also be considered as well as addressing other effects such as electromigration, IR drop and crosstalk. Primetime could be integrated with the above flow to perform static timing analysis.

# LIST OF REFERENCES

[1] Moore, G.E., “Progress in digital integrated electronics”, Electron Devices Meeting, 1975 International Volume 21, 1975 Page(s):11 – 13

[2] Integrated circuit, [online] Available: http://en.wikipedia.org/wiki/Integrated_circuit

[3] Bergamaschi, R.A.; Cohn, J., “The A to Z of SoCs” IEEE/ACM International Conference on Computer Aided Design 2002, 10 - 14 Nov. 2002, Page(s):791 - 798

[4] Sangiovanni-Vincentelli, A.; Martin, G., “Platform-based design and software design methodology for embedded systems”, Design & Test of Computers, IEEE Volume 18, Issue 6, Nov.-Dec. 2001 Page(s):23 - 33

[5] Kurt Keutzer; A. Richard Newton; Narendra Shenoy, “The future of logic synthesis and physical design in deep-submicron process geometries”, Proceedings of the 1997 International Symposium on Physical Design, April 1997, Page(s):218-224

[6] Padmini Gopalakrishnan; Altan Odabasioglu; Lawrence Pileggi; Salil Raje, “Overcoming wire-load model uncertainty during physical design”, Proceedings of the 2001 international symposium on Physical design, 2001 Page(s): 182 - 189

[7] Smith, M. J. S., “Application-Specific Integrated Circuits,” Reading. Addison - Wesley, Boston, MA, 1997.

[8] Hojat, S.; Villarrubia, P., “An integrated placement and synthesis approach for timing closure of PowerPC microprocessors”, Proceedings of the 1997 International Conference on Computer Design, 1997, 12-15 Oct. 1997, Page(s):206 – 210

[9] Donath, W.; Kudva, P.; Stok, L.; Villarrubia, P.; Reddy, L.; Sullivan, A., “Transformational placement and synthesis”, Proceedings of the 2000 Design, Automation and Test in Europe Conference and Exhibition, 2000, 27-30 March 2000 Page(s):194 - 201

[10] Yiu-Hing Chan; Kudva, P.; Lacey, L.; Northrop, G.; Rosser, T., “Physical synthesis methodology for high performance microprocessors”, Proceedings of the Design Automation Conference, 2003, 2-6 June 2003, Page(s):696 – 701

[11] Jiri Gaisler, Gaisler Research. The LEON-2 Processor: User’s Manual. [Online]. Available: http://www.gaisler.com/.

[12] Srivastava, R.,”Development of An Open Core System-on-Chip Platform”, M.S. Thesis, University of Tennessee, August 2004.

[13] Jiang, W.,”Enhancing System-on-Chip Verification using Embedded Test Structures”, M.S. Thesis, University of Tennessee, December 2005.

[14] Fields, S., “Hardware Design and Implementation of Role-Based Cryptography", M.S. Thesis, University of Tennessee, Dec. 2005.

[15] Marwah, T., “System-on-Chip Design and Test with Embedded Debug Capabilities", M.S. Thesis, University of Tennessee, August, 2006.

[16] Register Window, [Online] Available: http://en.wikipedia.org/wiki/Register_window

[17] AMBA Specifications Rev 2.0, ARM Limited, (1999). [Online] Available: http://www.arm.com/products/solutions/AMBA_Spec.html

[18] AMBA Bus System, [Online] Available: http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=4165

[19] Artisan Standard Library 0.13um - 0.25um SRAM Generator User Manual, ARM Limited, June 2005, Revision 2005q2v3.

[20] LECCS Manual, [Online] Available:http://www.gaisler.com/doc/leccs-1.1.5.pdf

[21] Galaxy Reference Flow User Guide Version 2.1, Synopsys Inc., April 2006.

[22] Design Compiler User Guide, Synopsys Inc., Version Y-2006.06, June 2006

[23] Priti Vijayvargiya, “Design Compiler Topographical Technology: Enhancing Synthesis Predictability and Productivity”, Synopsys Insight Volume 1, Issue 2, AUGUST - SEPTEMBER, 2006. [Online] Available: http://www.synopsys.com/news/pubs/insight/2006/art3_dctopo_v1s2.html?NLC-insight&Link=Aug06_V1-I2_Art3

[24] JupiterXT, [Online] Available: http://synopsys.com/products/jupiterxt/jupiterxt.html

[25] JupiterXT Virtual Flat Flow User Guide, Synopsys Inc., Version Y-2006.06, June 2006

[26] Physical Compiler User Guide, Volume 1, Synopsys Inc., Version X-2005.09, December 2005

[27] Physical Compiler Data Sheet, [Online] Available: http://synopsys.com/products/unified_synthesis/unified_synthesis_ds.pdf

[28] Astro Data Sheet, [Online] Available: http://www.synopsys.com/products/astro/astro_ds.pdf

[29] Astro User Guide, Synopsys Inc., Version X-2005.09, September 2005

[30] Milkyway Environment Data Preparation User Guide Version X-2005.09, September 2005

[31] Solvnet, [Online] Available: www.solvnet.synopsys.com

[32] Astro1 Workshop Student Guide, 2005-09, Synopsys Customer Education Services

# Vita

Pradeep M Chimakurthy was born in Kakinada, India on November 29th 1981. He received his Bachelor of Engineering degree in Electronics and Communication from the University of Madras in December 2003. He joined the University of Tennessee, Knoxville in August 2004 and received his Master of Science degree in Electrical Engineering in December 2006 under the guidance of Dr. Donald Bouldin. His areas of interest include Digital VLSI design, Computer Architecture and Digital Communications.