ASIC DESIGN OF THE OPENSPARC T1 PROCESSOR CORE

By

Mohamed Mahmoud Mohamed Farag

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2013

ASIC DESIGN OF THE OPENSPARC T1 PROCESSOR CORE

By
Mohamed Mahmoud Mohamed Farag

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Under the Supervision of

Prof. Dr. Serag El-Din Habib	Dr. Hossam A. H. Fahmy
Professor of Electronics Electronics and Communications Department Faculty of Engineering, Cairo University	Associate Professor Electronics and Communications Department Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2013

ASIC DESIGN OF THE OPENSPARC T1 PROCESSOR CORE

By
Mohamed Mahmoud Mohamed Farag

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Approved by the
Examining Committee

____
Prof. Dr. El Sayed Mostafa Saad, External Examiner

____
Prof. Dr. Ibrahim Mohamed Qamar, Internal Examiner

____
Prof. Dr. Serag El-Din Habib, Thesis Main Advisor

____
Dr. Hossam A. H. Fahmy, Thesis Advisor

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2013

Engineer's Name:	Mohamed Mahmoud Mohamed Farag
Date of Birth:	29/12/1985
Nationality:	Egyptian
E-mail:	eng.mohamedfarag@gmail.com
Phone:	01068823040
Address:	7 Ramzy Farag Street – Al Haram
Registration Date:	…./…./……..
Awarding Date:	6 / 1 / 2013
Degree:	Master of Science
Department:	ELECTRONICS AND COMMUNICATIONS ENGINEERING

Insert photo here

Supervisors:

Prof. Dr. Serag El-Din Habib
Dr. Hossam A. H. Fahmy

Examiners:

Prof. El Sayed Mostafa Saad	(External examiner)
Prof. Ibrahim Mohamed Qamar	(Internal examiner)
Porf. Serag El-Din Habib	(Thesis main advisor)
Porf. Hossam A. H. Fahmy	(Thesis advisor)

Title of Thesis:

ASIC Design Of The OpenSPARC T1 Processor

Key Words:
ASIC; SPARC; Layout; Processor; Design

Summary:

The objective of this thesis is to carry out an ASIC design of the OpenSPARC T1 processor core using the 130nm CMOS technology. Starting from the open-source RTL description of the OpenSPARC T1 processor core, several modifications like memory mapping, reducing the processor threads and suppressing the test pins were made in order to reduce the processor size to fit into a 4x4 mm^2 die. The correct functionality of the modified RTL description of this processor was verified against over 500 test scripts given by SUN Inc. Next, to convert the design from the RTL form to its gate level equivalent the design was synthesized. Subsequently, the design was physically implemented using the Place and Route flow, including the normal steps like floorplanning, placement, optimization, clock tree synthesis (CTS) and routing. Finally, the design was verified by a series of verification and performance evaluation tests to guarantee its functionality and performance. The designed processor core supports two threads and runs at a 100MHz clock frequency. It occupies an area of 16 mm^2, including pads.

Acknowledgments

First I thank ALLAH (God) for being to my side during all the time I spent working on this thesis, HE blessed me with the strength and guidance that I needed to complete it.

I would like to give my sincere appreciation to my parents for always giving me the moral support that I needed to continue working on this thesis and always praying for me.

I would like to give a very special thank you to my wife who stood to my side and withstood a lot of burden during my work on this project.

I would like to thank my manager at work "Ahmed Gharieb" for his endless support and understanding. I would like also to thank my Mentor Graphics family. Special thanks to "Mohamed Ali" my previous manager for being there for me whenever I needed him.

I would like to thank my suppervisors "Dr. Serag" and "Dr. Hossam" for their support, guidance and understanding that they gave me throughtout my research.

Finally I acknowledge MOSIS for providing the PDK of the IBM 130nm CMOS process under the MEP account number 4960.

Contents

Acknowledgments **V**

Table of contents **VI**

List of figures **XII**

List of tables **XV**

List of symbols and abbreviations **XVI**

Abstract **XXV**

1 Introduction **1**

1.1 SPARC architecture 1

1.1.1 SPARC V8 vs V9 2

1.1.2 SPARC-V9 Processor 4

1.1.3 SPARC-V9 Instructions 4
1.1.4 SPARC-V9 Traps 6
1.1.5 SPARC-V9 Data Formats 7
1.1.6 SPARC-V9 Registers 8
1.1.7 SPARC-V9 Memory Models 8
1.1.8 SPARC-V9 Operation 11
1.2 Thesis Objective 12
1.3 Thesis Map 13

2 OpenSparc T1 **15**

2.1 OpenSPARC T1 architecture 16
2.1.1 SPARC Core 19
2.1.2 Floating-Point Unit 22
2.1.3 CPU-Cache Crossbar 22
2.1.4 L2-Cache 24
2.1.5 DRAM Controller 25
2.1.6 I/O Bridge 26
2.1.7 J-Bus Interface 26
2.1.8 Serial System Interface 27
2.1.9 Electronic Fuse 27
2.2 OpenSPARC T1 core 28
2.2.1 Instruction fetch unit (IFU) 30

2.2.2 Execution unit (EXU) 32
2.2.3 Load store unit (LSU) 34
2.2.4 Trap logic unit (TLU) 36
2.2.5 Stream processing unit (SPU) 38
2.2.6 Memory management unit (MMU) 39
2.2.7 Floating-point frontend unit (FFU) 41
2.3 OpenSPARC T1 design history 43
2.3.1 Xilinx FPGA design 1 [6] 43
2.3.2 Xilinx FPGA design 2 [6] 44
2.3.3 UltraSparc T1 (Niagara) release [6] 44
2.3.4 Commercial processors based on the UltraSparc architecture [1] 46
2.4 Proposed Design 48

3 OpenSPARC T1 core Implementation 51
3.1 RTL preparation and functional verification 51
3.2 Synthesis 61
3.3 Design Floorplanning 63
3.4 Design Placement 68
3.5 CTS 71
3.6 Routing 73
3.7 Post Layout Verification 74

4 Results **76**

4.1 Timing results 77

4.2 Physical results 87

4.3 Design comparison 93

4.3.1 Processor size 94

4.3.2 Processor speed 96

5 Conclusion **98**

5.1 Future work 100

5.1.1 Tighten the design constraints 101

5.1.2 Custom implement the IRF memory module . . 101

5.1.3 Optimize clock network 101

5.1.4 Use smaller technology node 103

A Digital design ASIC Flow **104**

A.1 RTL preparation 105

A.2 RTL functionality Verification 109

A.3 Logic Synthesis 112

A.4 Gate Level functionality Verification 115

A.5 Floorplaning 116

A.6 Placement 131

A.7 Clock Tree Synthesis 136

A.8 Routing 142
A.9 Post layout verification 145

References **148**

List of Figures

1.1 Memory Models from Least Restrictive (RMO) to Most Restrictive (TSO) [2] 10

2.1 OpenSPARC T1 Processor Block Diagram [5] 18

2.2 SPARC Core Pipeline [5] 21

2.3 CCX Block Diagram [5] 24

2.4 SPARC Core Block Diagram [5] 28

2.5 Execution Unit Diagram [5] 33

2.6 LSU Pipeline Graph [5] 35

2.7 TLU Role With Respect to All Other Backlogs in a SPARC Core [5] 37

2.8 MMU and TLBs Relationship [5] 40

2.9 Top-Level FFU Block Diagram [5] 42

2.10 Sun UltraSPARC T1 (Niagara 8 Core) [6] 46

3.1 Processor blocks rough estimation [5] 64
3.2 Design Power Grid 65
3.3 Macro placement 67
3.4 Empty spaces used up by Macros 68
3.5 Macro Power Connection 69
3.6 Top blocks placement 70

4.1 Setup slack distribution 79
4.2 Hold slack distribution 80
4.3 Worst setup timing path. 82
4.4 Worst hold timing path. 83
4.5 Area percentage distribution. 87
4.6 Die area distribution. 88
4.7 Cell density histogram. 89
4.8 UltraSPARC T1 processor core [8] 95

5.1 IRF block area highlighted. 102

A.1 Digital ASIC Design Flow 105
A.2 Simulation VS Formal Verification [17] 111
A.3 Synthesis Stage 114
A.4 Floorplan of the design [21] 122
A.5 Pad driven VS Core driven [22] 124

A.6 Power grid of the design [21] 128
A.7 Example of macro placement [21] 130
A.8 Global router G-cells [21] 136
A.9 Tools used in design flow 147

List of Tables

3.1 Reduction in the processor core 58

4.1 Setup and hold timing path groups 78
4.2 Worst 10 setup paths. 81
4.3 Worst 10 hold paths. 81
4.4 CTS Properties. 84
4.5 Level Tree Histogram 85
4.6 Rise and Fall CTS latency. 86
4.7 Design routing statistics. 90
4.8 Clock routing statistics. 91
4.9 Power analysis. 92
4.10 Design comparison table 93

List of symbols and abbreviations

ALU	Arithmetic and Logic Unit
ASI	Address Space Identifier register
ASIC	Application-Specific Integrated Circuit
ASRs	Ancillary State Registers
CAM	Content Addressable Memory
CANRESTORE	Restorable Windows Register
CANSAVE	Savable Windows Register
CCR	Condition Codes Register
CCX	CPUcache Crossbar

CLEANWIN	Clean Windows Register
CSR	Control Status Registers
CTIs	Control-Transfer Instructions
CTL	Control Block
CTS	Clock Tree Synthesis
CWP	Current Window Pointer register
D-cache	Data Cache
DCD	Data Cache Data Array
DDR	Double Data Rate
DEF	Design Exchange Format
DFM	Design For Manufacturability
DIMM	Dual In-Line Memory Modules
DMA	Direct Memory Access
DP	Data-Path
DRAM	Dynamic Random Access Memory

DRC	Design Rule Checks
e-Fuse	Electronic Fuse
ECC	Error Correction Code
ECL	Execution Control Logic
ESD	Electrostatic Discharge
EXU	Execution Unit
F registers	Floating-point working registers
FFU	Floating-Point Frontend Unit
FPGA	Field-Programmable Gate Array
FPop	Floating Point operate
Fpop	Floating-Point Operation
FPRS	Floating-Point Registers State Register
FPU	Foating-Point Unit
FQ	Floating-Point Deferred-Trap Queue
FRF	Floating-Point Register File

FSR	Floating-Point State Register
G-cells	Global Cells
GSR	Graphics State Register
HFNS	High Fanout Net Synthesis
I-cache	Instruction Cache
ICD	Instruction Cache Data
IDCT	Instruction and Data Cache Tag
IDIV	Integer Divider
IFU	Instruction Fetch Unit
IMUL	Integer Multiplier
IOB	I/O Bridge
IRF	Integer Register File
ITLB	Instruction Translation Lookaside Buffer
IU	Integer Unit
JBI	J-Bus Interface

JBI	JBUS Interface
LEF	Library Exchange Format
LIB	Synopsys Liberty Format
LRU	Least Recently Used
LSU	Load/Store Unit
LVS	Layout Versus Schematic
MA	Modular Arithmetic
MMU	Memory Management Unit
NIR	Next Instruction Register
nPC	Next Program Counter register
opcodes	Operation Codes
OTHERWIN	Other Windows Register
P&R	Place and Route
PC	Program Counter register
PCX	Processor Cache-Crossbar

PIC	Performance Instrumentation Counters
PIL	Processor Interrupt Level register
PIO	Programmed Input/Output
PLI	Programmable Logic Interface
PSO	Partial Store Order
PSTATE	Processor State register
PTF	Pinnacle Timing Formate
R registers	Integer working registers
RAM	Randomly Addressable Memory
RF	Register File
RISC	Reduced Instruction Set Computing
RMO	Relaxed Memory Order
RSA	Ron Rivest, Adi Shamir and Leonard Adleman
SDC	Synopsys Design Constraints
SDF	Standard Delay Format

SHFT	Shifter
SPARC	Scalable Processor ARChitecture
SPARC-V9	SPARC Version 9
SPU	Stream Processing Unit
SSI	Serial System Interface
STA	Static Timing Analysis
TBA	Trap Base Address register
TICK	Hardware Clock-Tick Counter Register
TIR	Thread Instruction Register
TL	Trap Level register
TLB	Translation Lookaside Buffer
TLU	Trap Logic Unit
TNPC	Trap Next Program Counter register
TNS	Total Negative Slack
TPC	Trap Program Counter register

TSB	Translation Storage Buffers
TSO	Total Store Order
TSTATE	Trap State Register
TT	Trap Type register
V-bit	Valid Bit
VCS	Verilog Compiler and Simulator
VER	Version Register
VIS	Visual Instruction Set
Vth	Threshold Voltage
WNS	Worst Negative Slack
WSTATE	Window State Register
Y	Y register

Abstract

OpenSPARC T1 is the first open-source, multi-threaded and multi-cored processor developed by SUN micro-systems. The objective of this thesis is to carry out an ASIC design of the core of this processor using the 130nm CMOS technology.

Starting from the open-source RTL description of the OpenSPARC T1 processor core, several modifications like memory mapping, reducing the processor threads and suppressing the test pins were made in order to reduce the processor size to fit it into a 4 x 4 mm^2 die area as required by the technology supplier. The correct functionality of the modified RTL description of this processor was verified against over 500 test scripts given by SUN Inc.

Next, to convert the design from the RTL form to its gate level equivalent the design was synthesized. Subsequently, the design was physically implemented using the Place and Route flow, including the normal steps like floorplanning, placement, optimization, clock tree synthesis (CTS) and routing. Finally, the design was verified by a series of verification and performance evaluation tests to guarantee its functionality and performance.

The designed processor core supports two threads and runs at a 100MHz clock frequency. It occupies an area of 16 mm^2, including pads. The performance metrics of the designed core are next compared to relevant results in the published literature.

Chapter 1

Introduction

This chapter introduces the basic knowledge and brief background of topics related to this thesis. It also describes the objective behind this thesis and a map for this documentation in order to make it easier to navigate through different topics covered by this documentation.

1.1 SPARC architecture

SPARC stands for a **S**calable **P**rocessor **ARC**hitecture [1]. SPARC is a microprocessor specification created by the SPARC Architecture Committee of SPARC International. SPARC is not a chip; it is an architectural specification that can be implemented as a microprocessor by anyone having a license from SPARC International. SPARC has been implemented in processors used in a range of computers from laptops to supercomputers. SPARC is based on the RISC (Reduced Instruction Set Computing) I and II designs engineered at the University of California at Berkeley. In this thesis work we will be focusing only on version 9 (SPARC-V9), as this is the version implemented by the OpenSPARC T1 design.

1.1.1 SPARC V8 vs V9

SPARC-V9 does not replace the SPARC-V8 architecture; it is complimentary to it. SPARC-V9 was architectured to be a higher-performance peer to SPARC-V8. Application software for the 32-bit SPARC-V8 architecture can execute, unchanged, on SPARC-V9 systems, no special compatibility mode is needed. The SPARC-V9 architecture differs from the SPARC-V8 in six main areas: the trap model, data formats, the registers, alternate address space access, the instruction set, and the memory model.

Trap Model: SPARC-V9 architecture supports four or more levels of traps compared to one level supported by the SPARC-V8 architecture.

Data Formats: Data formats for extended (64-bit) integers have been added to the SPARC-V9 architecture. Also Little-Endian Support has been added to the existing Big-Endian.

Registers: A lot of register changes have been made, some registers were totaly removed, some were added, contents of some have been changed and some were widened from 32 to 64 bits. For complete list of register changes please refer to [2].

Alternate Space Access: In SPARC-V8 architecture, access to all alternate address spaces is privileged. In SPARC-V9 architecture, load and store alternate instructions to one-half of the alternate spaces can now be used in user code (nonprivileged).

Instruction Set: Instructions for the SPARC-V9 architecture now process 64 bit values. Some new instructions were added to provide support for 64-bit operations and/or addressing, support the new trap model, support implementation of higher-performance systems, and support memory synchronization. Other instructions have been changed or deleted. For complete list of instructions changes please refer to [2].

Memory Model: SPARC-V9 architecture introduces a new memory model called Relaxed Memory Order (RMO). This model allows the CPU hardware to schedule memory accesses such as loads and stores in nearly any order, as long as the program computes the correct answer. Leading to much faster memory operations and better performance.

1.1.2 SPARC-V9 Processor

A SPARC-V9 processor logically consists of an integer unit (IU) and a floating-point unit (FPU). This organization allows the concurrency between integer and floating-point instruction execution. The integer unit contains the general-purpose registers and controls the overall operation of the processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU. FPU is a processing unit that contains the floating-point registers and performs floating-point operations.

1.1.3 SPARC-V9 Instructions

SPARC-V9 instructions fall into the following basic categories:

- Memory access

- Integer operate
- Control transfer
- State register access
- Floating-point operate
- Conditional move
- Register window management

Memory access instructions are the instructions related to the memory load and store operations of the processor. **Integer operate** instructions are the arithmetic/logical/shift instructions performed by the processor which perform arithmetic, tagged arithmetic, logical, and shift operations. **Control-transfer** instructions (**CTI**s) include PC-relative branches and calls, register-indirect jumps, and conditional traps. **State Register Access** are instructions that controls the reading and writing of contents to the state registers. **Floating-point operate** (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. **Conditional move** instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or upon the contents of an integer register. **Register Window Management** instructions are used to manage the register windows. Instructions are encoded in four major 32-bit formats and several minor formats.

1.1.4 SPARC-V9 Traps

Trap is the action taken by the processor when it changes the instruction flow in response to the presence of an **exception** (ex: an interrupt). A trap behaves like an unexpected procedure call. It causes the hardware to

1. Save certain processor state (program counters, trap type ....etc) on a hardware register stack.
2. Enter privileged execution mode with a predefined PSTATE.
3. Begin executing trap handler code in the trap vector.

Normally the processor behaves as the following, before executing each instruction, it determines if there are any pending exceptions or interrupt requests. If there are pending exceptions or interrupt requests, the processor selects the highest-priority exception or interrupt request and causes a trap. which means that an exception is a condition that makes it impossible for the processor to continue executing the current instruction stream without software intervention. After the trap handler finishes, it uses either a DONE or RETRY instruction to return.

1.1.5 SPARC-V9 Data Formats

The SPARC-V9 architecture recognizes these fundamental data types:

- Signed Integer: 8, 16, 32, and 64 bits
- Unsigned Integer: 8, 16, 32, and 64 bits
- Floating Point: 32, 64, and 128 bits

The widths of the data types are:

- Byte: 8 bits
- Halfword: 16 bits
- Word: 32 bits
- Extended Word: 64 bits
- Tagged Word: 32 bits (30-bit value plus 2-bit tag)

- Doubleword: 64 bits
- Quadword: 128 bits

1.1.6 SPARC-V9 Registers

A SPARC-V9 processor includes two types of registers: general-purpose, or working data registers, and control/status registers. As the name indicates the general purpose registers are registers used to store data during the normal operation of the processor, it is divided into integer and floating point registers. The control/status registers are special purpose registers that defines a certain state of the processor or to control certain operations carried out by the processor.

1.1.7 SPARC-V9 Memory Models

The SPARC-V9 **memory models** define the way the memory operates. The instruction set criteria require that loads and stores seem to be performed in the order in which they appear in the dynamic control flow of the program. The actual order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

The memory models are similar for both uniprocessor and shared-memory multiprocessors. Formal memory models are needed in order to precisely define the interactions between multiple processors and input/output devices in a shared-memory configuration. Programming shared-memory multiprocessors requires a deep understanding of the operative memory model and the ability to specify memory operations at a low level in order to construct programs that can safely and reliably coordinate their activities.

The SPARC-V9 architecture is a model that specifies the behavior observable by software on SPARC-V9 systems. Therefore, access to memory can be implemented in any manner, as long as the behavior observed by software conforms to that of the models.

The SPARC-V9 architecture defines three different memory models: **Total Store Order (TSO)**, **Partial Store Order (PSO)**, and **Relaxed Memory Order (RMO)**.

Figure 1.1 shows the relationship of the various SPARC-V9 memory models, from the least restrictive to the most restrictive. Programs written assuming one model will function correctly on any included model.

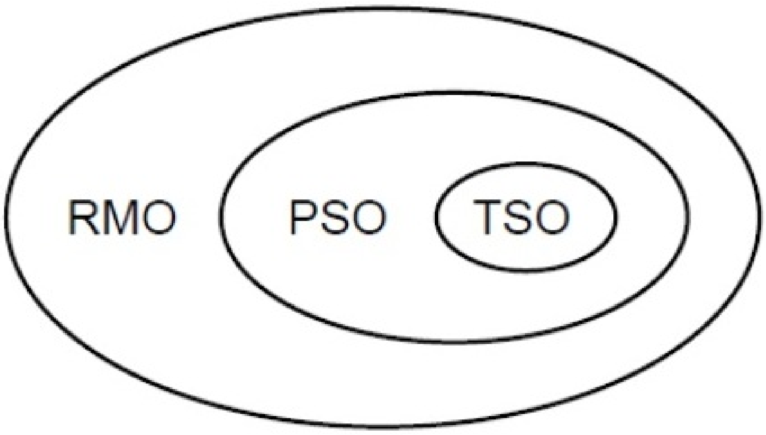


Figure 1.1: Memory Models from Least Restrictive (RMO) to Most Restrictive (TSO) [2]

SPARC-V9 provides multiple memory models so that:

- Implementations can schedule memory operations for high performance.
- Programmers can create synchronization primitives using shared memory.

There is no preferred memory model for SPARC-V9. Programs written for Relaxed Memory Order will work in Partial Store Order and Total Store Order. Programs written for Partial Store Order will work in Total Store Order. Programs written for a weak model, for example RMO, may execute more quickly, since the model contains more scheduling opportunities, but may also require extra instructions to ensure synchronization. Multiprocessor programs written for a stronger model will have unpredictable behavior if run in a weaker model.

1.1.8 SPARC-V9 Operation

The instruction at the memory location specified by the program counter is fetched and then executed. Instruction execution may change program-visible processor and/or memory state. As a side-effect of its execution, new values are assigned to the program counter (PC) and the next program counter (nPC).

An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is transfered to a trap table.

If a trap does not occur and the instruction is not a control transfer, the next program counter (nPC) is copied into the PC and the nPC is incremented. If the instruction is a control-transfer instruction, the next program counter (nPC) is copied into the PC and the target address is written to nPC.

1.2 Thesis Objective

ASIC design has become the main focus for any person or company seeking to develop an electronic product that can perform effectively and incorporates a lot of functionality and options. Due to the huge competition in this field and huge customer demands the technology in this field is developing at a fast pace towards smaller, faster and more complex devices which is making it harder for a newcomer to coupe up with this field without learning and understanding the basics first. The objective of this thesis is to carry out an ASIC design of the OpenSPARC T1 processor core using the 130nm CMOS technology.

The best way to achieve this objective is to define a good starting point for the flow and then proceed downstream the flow exploring each step at a time till a defined endpoint is reached. In the case of this thesis work a well tested real design (OpenSPARC T1 processor core) in its RTL form was chosen as the starting point and the target was to process this design to produce an output that can be fabricated using a real technology process, passing through all necessary design steps needed to reach this output.

1.3 Thesis Map

This thesis documentation is divided into five chapters and an Appendix. Chapter1 is an introductory chapter that covers some background knowledge that is needed throughout this documentation also it covers the thesis objective and this thesis map. Chapter 2 describes the OpenSPARC T1 design architecture which is the design this thesis work is based on, also it mentions the available resources that were used. Chapter 2 also surveys previous real physical designs of the processor and states the differences between them and the design achieved by this thesis work in section 2.4 of the chapter.

Appendix A gives some background on each step of the ASIC design flow in general not depending on a specific design. Chapter 3 describes how I applied the flow described in appendix A on my design. Chapter 4 gathers all the achieved results from applying the flow on my design and makes some analysis on them. Also chapter 4 contains a comparison between my results to other real designs results.

Chapter 5 concludes the work achieved by this thesis and mention some future work.

Chapter 2

OpenSparc T1

Sun Microsystems began shipping the UltraSPARC T1 chip multi-threaded (CMT) processor in December 2005. Sun surprised the industry by announcing that it would not only ship the processor but also open-source the RTL design of this processor. By March 2006, UltraSPARC T1 had been open-sourced in a distribution called OpenSPARC T1 [3]. In this chapter we will explore in more details the components and architecture of this processor and then will focus on its core as this is the main component this thesis work circles around.

2.1 OpenSPARC T1 architecture

The OpenSPARC T1 processor is a highly integrated processor that implements the 64-bit SPARC V9 architecture. The OpenSPARC T1 processor contains eight SPARC processor cores, which each have full hardware support for four threads. Each SPARC core has an instruction cache, a data cache, and a fully associative instruction and data translation lookaside buffers (TLB). The eight SPARC cores are connected through a crossbar to an on-chip unified level 2 cache (L2-cache).

The four on-chip dynamic random access memory (DRAM) controllers directly interface to the double data rate-synchronous DRAM (DDR2 SDRAM). Additionally, there is an on-chip J-Bus controller that provides an interconnect between the OpenSPARC T1 processor and the I/O subsystem.

The features of the OpenSPARC T1 processor include [4]:

- 8 SPARC V9 CPU cores, with 4 threads per core, for a total of 32 threads
- 132 Gbytes/sec crossbar interconnect for on-chip communication

- 16 Kbytes of primary (Level 1) instruction cache per CPU core
- 8 Kbytes of primary (Level 1) data cache per CPU core
- 3 Mbytes of secondary (Level 2) cache - 4 way banked, 12 way associative shared by all CPU cores
- 4 DDR-II DRAM controllers - 144-bit interface per channel, 25 GBytes/sec peak total bandwidth
- IEEE 754 compliant floating-point unit (FPU), shared by all CPU cores

External interfaces:

- J-Bus interface (JBI) for I/O - 2.56 Gbytes/sec peak bandwidth, 128-bit multiplexed address/data bus
- Serial system interface (SSI) for boot PROM

Figure 2.1 shows a block diagram of the OpenSPARC T1 processor illustrating the various interfaces and integrated components of the chip.

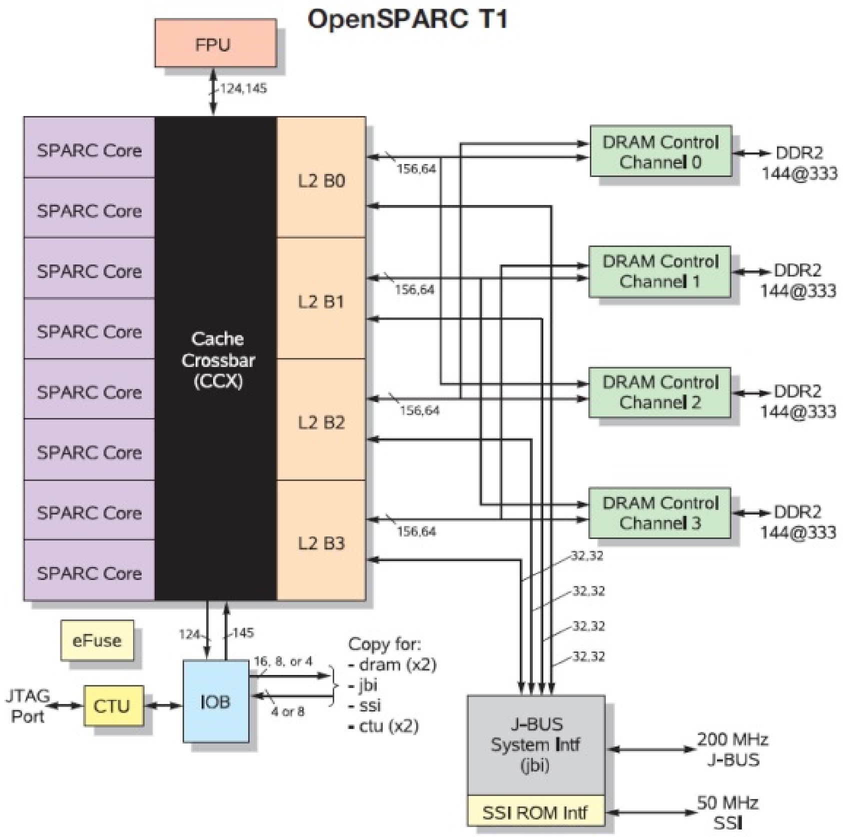


Figure 2.1: OpenSPARC T1 Processor Block Diagram [5]

OpenSPARC T1 includes the following components:

1. 8 SPARC Cores

2. Floating-Point Unit

3. CPU-Cache Crossbar
4. L2-Cache
5. DRAM Controller
6. I/O Bridge
7. J-Bus Interface
8. Serial System Interface
9. Electronic Fuse

In the following sections briefly describe each component.

2.1.1 SPARC Core

Each SPARC core has hardware support for four threads. This support consists of a full register file (with eight register windows) per thread, with most of the address space identifiers (ASI), ancillary state registers (ASR), and privileged registers replicated per thread. The four threads share the instruction, the data caches, and the TLBs. Each instruction cache is 16 Kbytes with a 32-byte line size. The data caches are write through, 8 Kbytes, and have a 16-byte line size. The TLBs include an autodemap feature which enables the multiple threads to update the TLB without locking. [5]

Each SPARC core has single issue, six stage pipeline. These six stages are:

1. Fetch
2. Thread Selection
3. Decode
4. Execute
5. Memory
6. Write Back

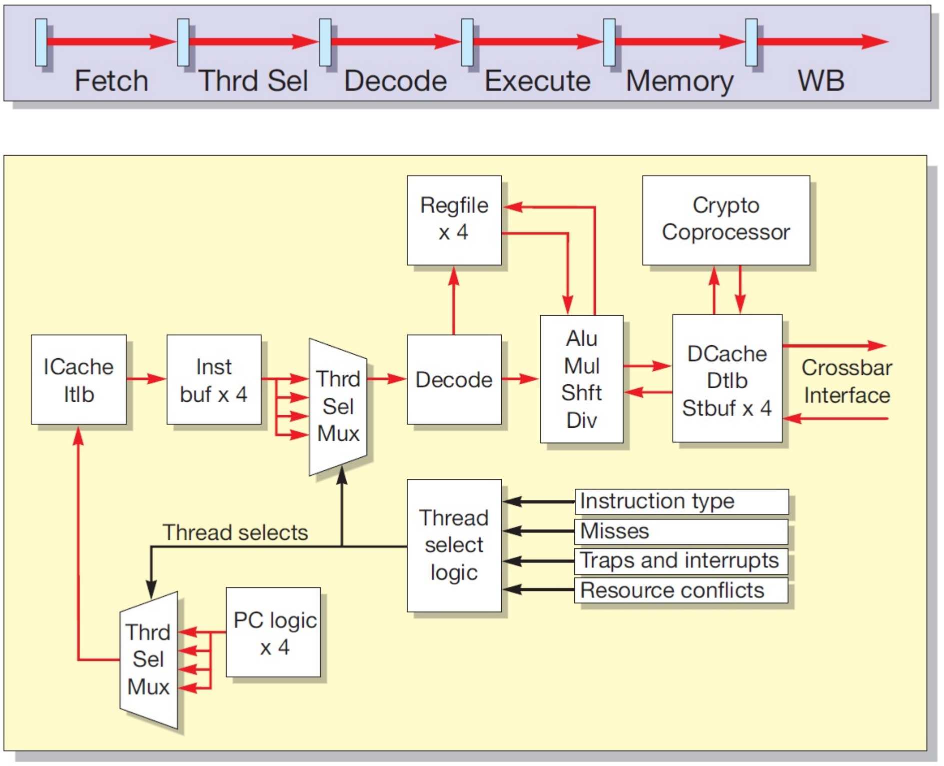


Figure 2.2: SPARC Core Pipeline [5]

Each SPARC core has the following units:

1. Instruction fetch unit (IFU) includes the following pipeline stages - fetch, thread selection, and decode. The IFU also includes an instruction cache complex.

2. Execution unit (EXU) includes the execute stage of the pipeline.

3. Load/store unit (LSU) includes memory and writeback stages, and a data cache complex.
4. Trap logic unit (TLU) includes trap logic and trap program counters.
5. Stream processing unit (SPU) is used for modular arithmetic functions for crypto.
6. Memory management unit (MMU).
7. Floating-point frontend unit (FFU) interfaces to the FPU.

2.1.2 Floating-Point Unit

A single floating-point unit (FPU) is shared by all eight SPARC cores. The shared floating-point unit is sufficient for most commercial applications in which typically less than 1% of the instructions are floating-point operations [5].

2.1.3 CPU-Cache Crossbar

The eight SPARC cores, the four L2-cache banks, the I/O Bridge, and the FPU all interface with the crossbar.

FIGURE 2.3 displays the crossbar block diagram. The CPUcache crossbar (CCX) features include:

- Each requester queues up to two packets per destination.
- Three stage pipeline - request, arbitrate, and transmit.
- Centralized arbitration with oldest requester getting priority.
- Core-to-cache bus optimized for address plus double word store.
- Cache-to-core bus optimized for 16-byte line fill. 32-byte Is line fill delivered in two back-to-back clocks.

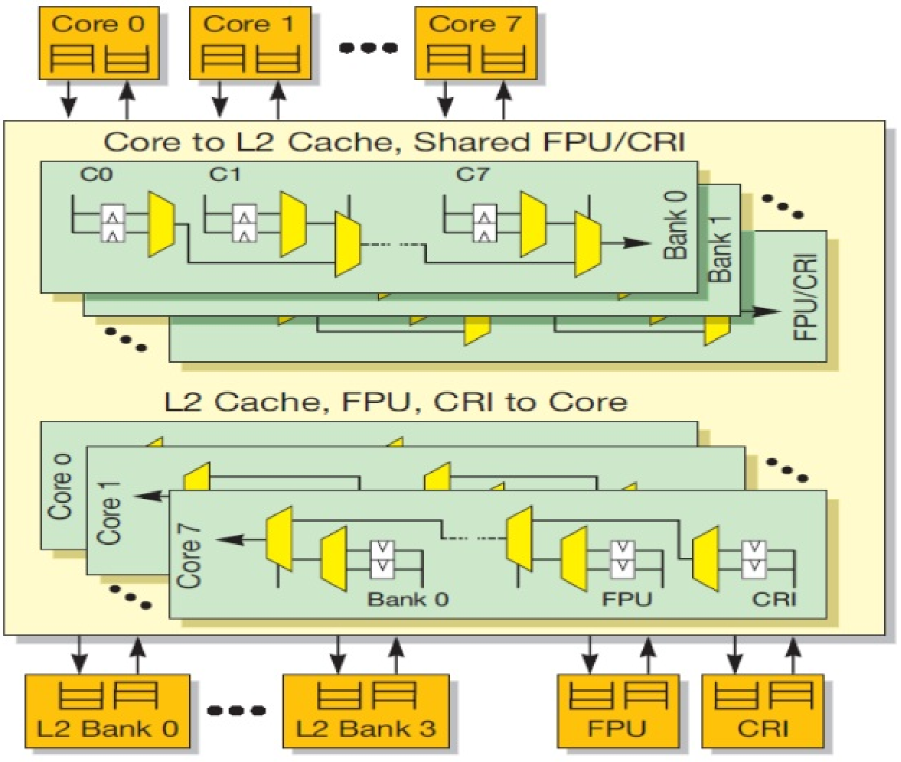


Figure 2.3: CCX Block Diagram [5]

2.1.4 L2-Cache

The L2-cache is banked four ways, with the bank selection based on the physical address bits 7:6. The cache is 3-Mbyte, 12-way set-associative with pseudo-least recently used (LRU) replacement (the replacement is based on a used bit scheme). The line size is 64 bytes. Unloaded access time is 23 cycles for an L1 data cache miss and 22 cycles for an L1 instruction cache miss.

L2-cache has a 64-byte line size, with 64 bytes interleaved between banks. Pipeline latency in the L2-cache is 8 clocks for a load, 9 clocks for an I-miss, with the critical chunk returned first. 16 outstanding misses per bank are supported for a 64 total misses. Coherence is maintained by shadowing the L1 tags in an L2-cache directory structure (the L2-cache is a point of global visibility). DMA from the I/O is serialized with respect to the traffic from the cores in the L2-cache. [5]

2.1.5 DRAM Controller

The OpenSPARC T1 processor DRAM controller is banked four ways, with each L2 bank interacting with one DRAM controller bank. Each DRAM controller bank must have identical dual in-line memory modules (DIMM) installed and enabled. The OpenSPARC T1 processor uses DDR2 DIMMs and can support one or two ranks of stacked or unstacked DIMMs. Each DRAM bank/port is two-DIMMs wide (128-bit + 16-bit ECC). All installed DIMMs must be identical, and the same number of DIMMs (ranks) must be installed on each DRAM controller port. The DRAM controller frequency is an exact ratio of the core frequency, where the core frequency must be at least three times the DRAM controller frequency. The double data rate (DDR) data buses transfer data at twice the frequency of the DRAM controller frequency. The OpenSPARC T1 processor can support memory sizes of up to 128 Gbytes with a 25 Gbytes/sec peak bandwidth limit. Memory access is scheduled across 8 reads plus 8 writes.

2.1.6 I/O Bridge

The I/O bridge (IOB) performs an address decode on I/O-addressable transactions and directs them to the appropriate internal block or to the appropriate external interface (J-Bus or the serial system interface). Additionally, the IOB maintains the register status for external interrupts.

2.1.7 J-Bus Interface

The J-Bus interface (JBI) is the interconnect between the OpenSPARC T1 processor and the I/O subsystem. The J-Bus is a 200 MHz, 128-bit wide, multiplexed address or data bus, used mainly for direct memory access (DMA) traffic, plus the programmable input/output (PIO) traffic used to control it. The J-Bus interface is the functional block that interfaces to the J-Bus, receiving and responding to DMA requests, routing them to the appropriate L2 banks, and also issuing PIO transactions on behalf of the processor threads and forwarding responses back.

2.1.8 Serial System Interface

The OpenSPARC T1 processor has a 50 Mbyte/sec serial system interface (SSI) that connects to an external application-specific integrated circuit (ASIC), which in turn interfaces to the boot read-only memory (ROM). In addition, the SSI supports PIO accesses across the SSI, thus supporting optional control status registers (CSR) or other interfaces within the ASIC.

2.1.9 Electronic Fuse

The electronic fuse (e-Fuse) block contains configuration information that is electronically burned-in as part of manufacturing, including part serial number and core available information.

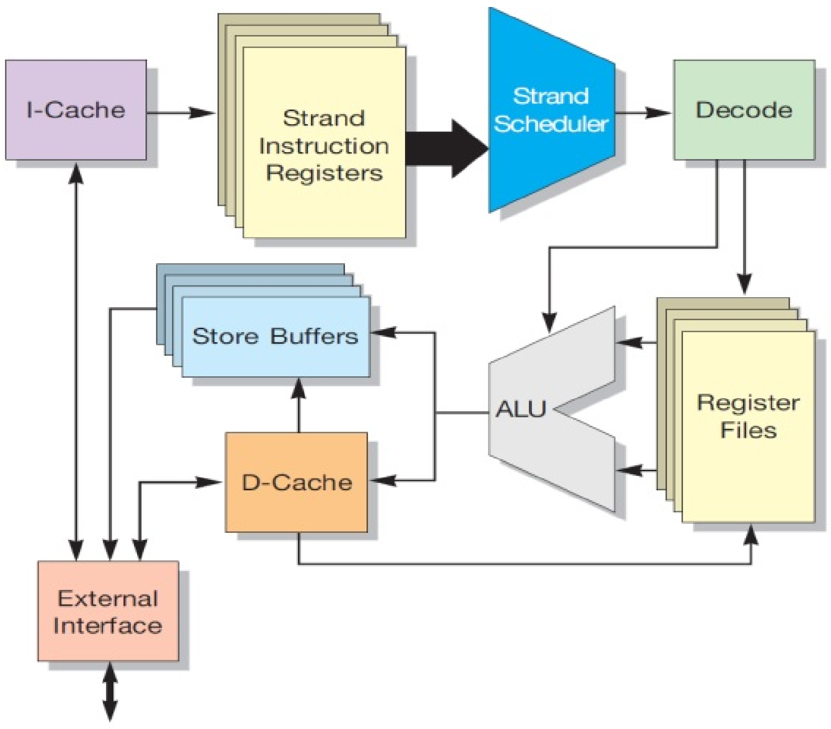


Figure 2.4: SPARC Core Block Diagram [5]

2.2 OpenSPARC T1 core

FIGURE 2.4 presents a high-level block diagram of a SPARC core. OpenSPARC T1 processor is considered to be one of few real-life commercial complex designs that has been open sourced, which means that the design has been thoroughly tested and verified by the designers, which also means that the test suites and design scripts that have been used are to some extent available for reference and reuse. Taking these points into consideration and adding to them the availability of good documentation and guides explains why this design was chosen for this thesis work.

In this section we will discuss in more details the internal architecture of the openSPARC T1 core.

The SPARC core is divided into 7 main blocks which are:

1. Instruction fetch unit (IFU).
2. Execution unit (EXU).
3. Load/store unit (LSU).
4. Trap logic unit (TLU).
5. Stream processing unit (SPU).
6. Memory management unit (MMU).
7. Floating-point frontend unit (FFU).

2.2.1 Instruction fetch unit (IFU)

The instruction fetch unit (IFU) is responsible for maintaining the program counters (PC) of different threads and fetching the corresponding instructions. The IFU also manages the level 1 I-cache and the instruction translation lookaside buffer (ITLB), as well as managing and scheduling the four threads in the SPARC core. The SPARC core pipeline is located in the IFU, which controls instruction issue and instruction flow in the pipeline. The IFU decodes the instructions flowing through the pipeline, schedules interrupts, and it implements the idle/resume states of the pipeline. The IFU also logs the errors and manages the error registers.

There are six stages in a SPARC core pipeline:

- Fetch - F-stage
- Thread selection - S-stage
- Decode - D-stage
- Execute - E-stage
- Memory - M-stage
- Writeback - W-stage

The I-cache access and the ITLB access take place in fetch stage. A selected thread (hardware strand) will be picked in the thread selection stage. The instruction decoding and register file access occur in the decode stage. The branch evaluation takes place in the execution stage. The access to memory and the actual writeback will be done in the memory and writeback stages. FIGURE 2.2 illustrates the SPARC core pipeline and support structures.

The thread selection policy is as follows : switch between the available threads every cycle giving priority to the least recently executed thread. The threads may become unavailable due to the long latency operations like loads, branch, MUL, and DIV, as well as to the pipeline stalls like cache misses, traps, and resource conflicts. The loads are speculated as cache hits, and the thread is switched-in with lower priority.

Instruction cache complex has a 16-Kbyte data, 4-way, 32-byte line size with a single ported instruction tag. It also has dual ported (1R/1W) valid bit array to hold cache line state of valid/invalid. There is a fully associative instruction TLB with 64 entries. The buffer supports the following page sizes: 8 Kbytes, 64 Kbytes, 4 Mbytes, and 256 Mbytes. The TLB uses a pseudo least recently used (LRU) algorithm for replacement. Two instructions are fetched each cycle, though only one instruction is issued per clock, which reduces the instruction cache activity and allows for an opportunistic line fill. The integer register file (IRF) of the SPARC core has 5 Kbytes with 3 read/2 write/1 transport ports. There are 640 64-bit registers with error correction code (ECC). Only 32 registers from the current window are visible to the thread [5]. The processor core supports eight register windows per thread.

2.2.2 Execution unit (EXU)

The execution unit (EXU) contains these four subunits - arithmetic and logic unit (ALU), shifter (SHFT), integer multiplier (IMUL), and integer divider (IDIV).

FIGURE 2.5 presents a top level diagram of the execution unit. The arithmetic and logic unit (ALU) executes arithemetic and logic operations such as - ADD, SUB, AND, NAND, OR, NOR, XOR, XNOR, and NOT. The ALU is also reused for branch address and virtual address calculation. MUL is the integer multiplier unit (IMUL), and DIV is the integer divider unit (IDIV). IMUL includes the accumulate function for modular arithmetic. The latency of IMUL is 5 cycles, and the throughput is 1-half per cycle. IDIV contains a simple

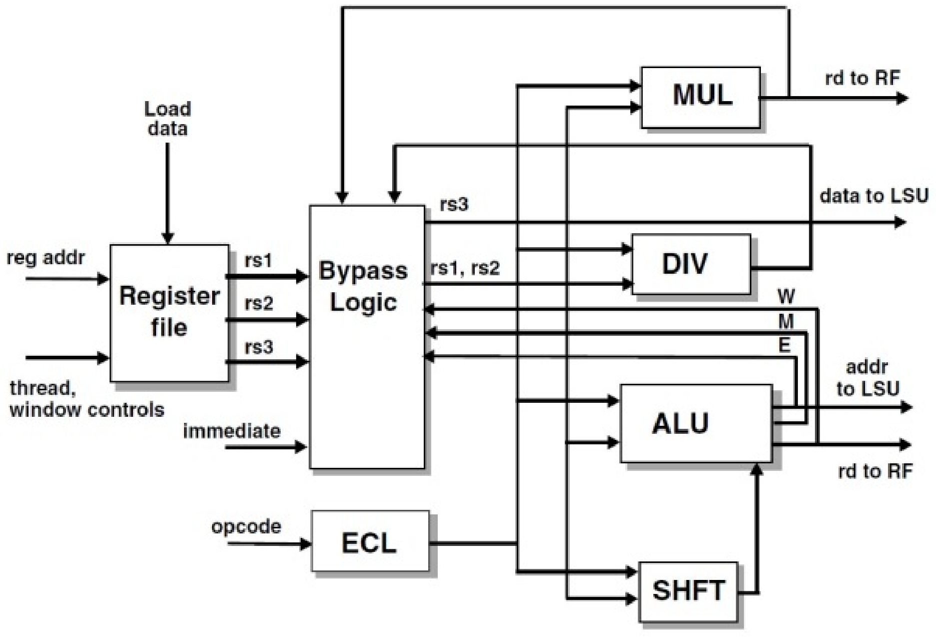


Figure 2.5: Execution Unit Diagram [5]

non-restoring divider, and it supports one outstanding divide operation per core. When either IMUL or IDIV is occupied, a thread issuing a MUL or DIV instruction will be rolled back and switched out.

The shifter block (SHFT) implements the 0 - 63-bit shift [5].

The execution control logic (ECL) block generates the necessary select signals that control the multiplexors, keeps track of the thread and reads the operands of each instruction, and implements the bypass logic. The ECL also generates the write-enables for the integer register file (IRF). The bypass logic block does the operand bypass from the E (Execute), M (Memory), and W (Writeback) stages to the D (Decode) stage. Results of long latency operations such as load, mul, and div, are forwarded from the W (Writeback) stage to the D (Decode) stage.

2.2.3 Load store unit (LSU)

The load store unit (LSU) processes memory-referencing operation-codes (opcodes) such as various types of loads, various types of stores ...etc. The threaded architecture of the LSU can process four loads, four stores, one fetch, one FP operation, one stream operation, one interrupt, and one forward packet. Therefore, thirteen sources supply data to the LSU. The LSU implements the ordering for memory references, whether locally or not. The LSU also enforces the ordering for all the outbound and inbound packets.

There are four stages in the LSU pipeline. FIGURE 2.6 shows the different stages of the LSU pipeline.

The cache access set-up and the translation lookaside buffer (TLB) access set-up are done during the pipeline's E-stage (execution). The

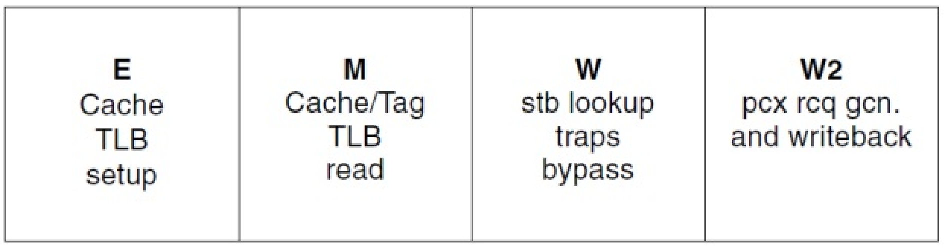


Figure 2.6: LSU Pipeline Graph [5]

cache/tag/TLB read operations are done in the M-stage (memory access). The W-stage (writeback) supports the look-up of the store buffer, the detection of traps, and the execution of the data bypass. The W2-stage (writeback-2) is for generating PCX requests and writebacks to the cache.

The LSU includes an 8-Kbyte D-cache, which is a part of the level 1 cache shared by four threads. The 8-Kbyte level 1 (L1) D-cache is 4-way set-associative, and the line size is 16 bytes. The D-cache has a single read and write port (1 RW) for the data and tag array. The valid bit (V-bit) array is dual ported with one read port and one write port (1R/1W). The valid bit array holds the cache line state of valid or invalid [5].

2.2.4 Trap logic unit (TLU)

The trap logic unit (TLU) has support for six trap levels. Traps cause pipeline flush and thread switch until trap program counter (PC) becomes available. The TLU also has support for up to 64 pending interrupts per thread. The TLU is in a logically central position to collect all of the traps and interrupts and forward them. Fig. 2.7 illustrates the TLU role with respect to all other backlogs in a SPARC core.

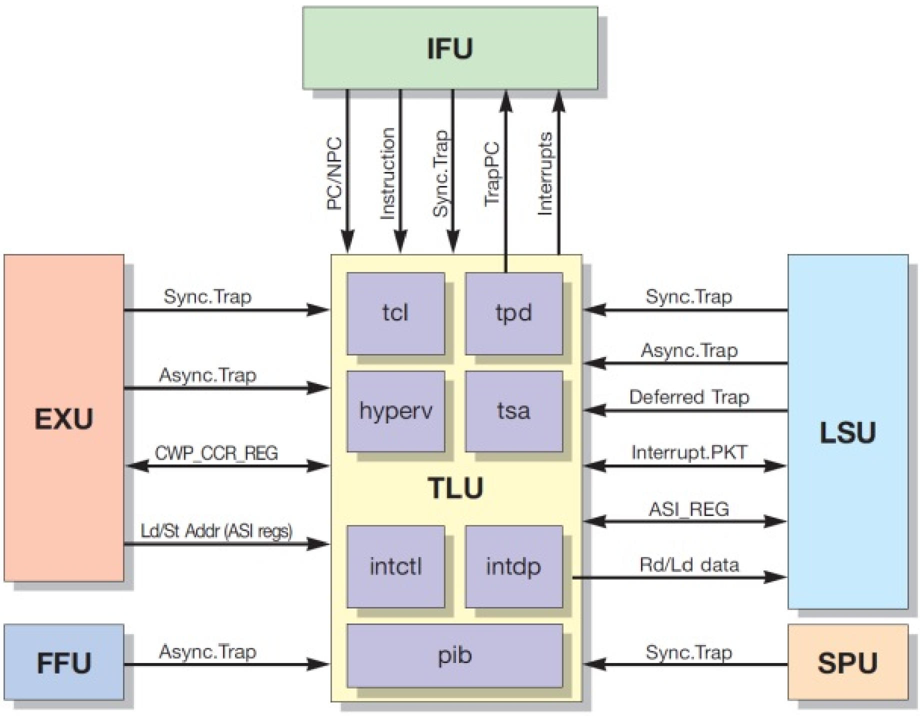


Figure 2.7: TLU Role With Respect to All Other Backlogs in a SPARC Core [5]

The following list highlights the functionality of the TLU:

- Collects traps from all units in the SPARC core.
- Detects some types of traps internal to the TLU.
- Resolves the trap priority and generates the trap vector.

- Sends flush-pipe to other SPARC units using a set of non-LSU traps.
- Maintains processors state registers.
- Manages the trap stack.
- Restores the processor state from the trap stack on done or retry instructions.
- Implements an inter-thread interrupt delivery.
- Receives and processes all types of interrupts.
- Maintains tick, all tick-compares, and the SOFT-INT related registers.
- Generates timer interrupts and software interrupts (interrupt-level-n type).
- Maintains performance instrumentation counters (PIC).

2.2.5 Stream processing unit (SPU)

The SPARC core is equipped with a stream processing unit (SPU) supporting the asymmetric cryptography operations (public-key RSA) for up to a 2048-bit key size. The SPU shares the integer multiplier with the execution unit (EXU) for the modular arithmetic (MA) operations. While the SPU facility is shared among all threads of a SPARC core, only one thread can use the SPU at a time. The SPU operation is set up by storing a thread to a control register and then returning to normal processing. The SPU will initiate streaming load or streaming store operations to the level 2 cache (L2) and compute operations to the integer multiplier. Once the operation is launched, it can operate in parallel with SPARC core instruction execution. The completion of the operation is detected by polling (synchronous way) or by interrupt (asynchronous way). [5]

2.2.6 Memory management unit (MMU)

The memory management unit (MMU) maintains the contents of the instruction translation lookaside buffer (ITLB) and the data translation lookaside buffer (DTLB). The ITLB resides in instruction fetch unit (IFU), and the DTLB resides in load and store unit (LSU). Fig. 2.8 shows the relationship among the MMU and the TLBs.

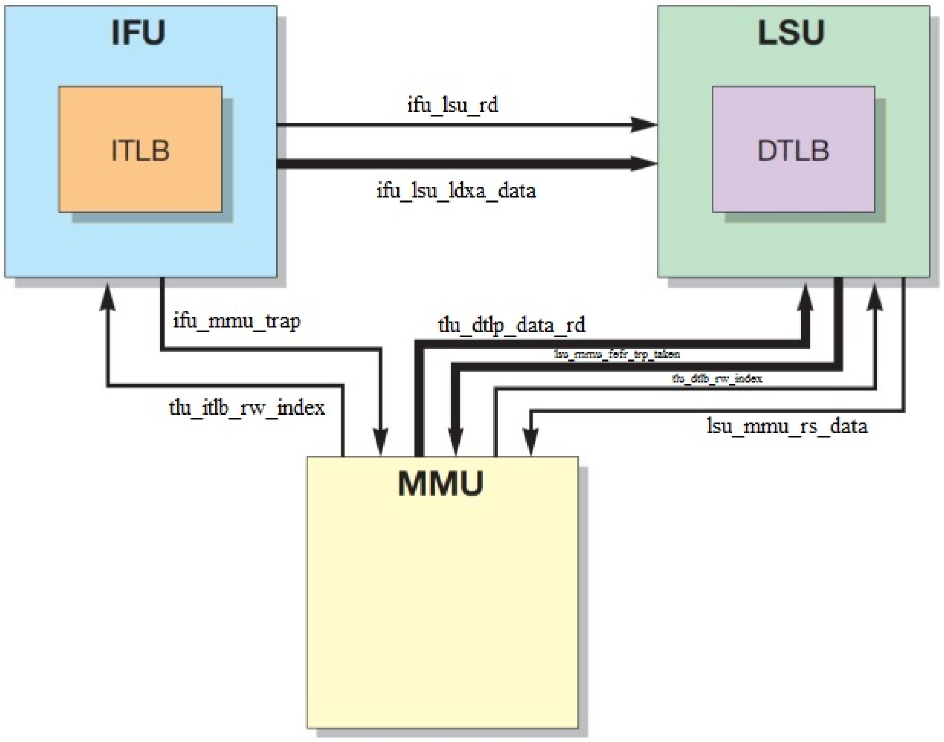


Figure 2.8: MMU and TLBs Relationship [5]

The MMU interacts with TLBs to maintain the content of TLBs. The system software manages the content of MMU by way of three kinds of operations - reads, writes, and demap. All TLB entries are shared among the threads, and the consistency among the TLB entries is maintained through auto-demap. The MMU is responsible for generating the pointers to the software translation storage buffers

(TSB), and it also maintains the fault status for the various traps. The translation lookaside buffer (TLB) consists of content addressable memory (CAM) and randomly addressable memory (RAM). CAM has one compare port and one read-write port (1C1RW), and RAM has one read-write port (1RW). [5]

2.2.7 Floating-point frontend unit (FFU)

The floating-point frontend unit (FFU) decodes floating-point instructions and it also includes the floating-point register file (FRF). The FFU also maintains the floating-point state register (FSR) and the graphics state register (GSR). Some of the floating-point instructions like move, absolute value, and negate are implemented in the FFU, while the others are implemented in the FPU.

The FFU is composed of four blocks - the floating-point register file (FFU-FRF), the control block (FFU-CTL), the data-path block (FFU-DP), and the VIS execution block (FFU-VIS). Fig. 2.9 shows a block diagram of the FFU illustrating these four subblocks.

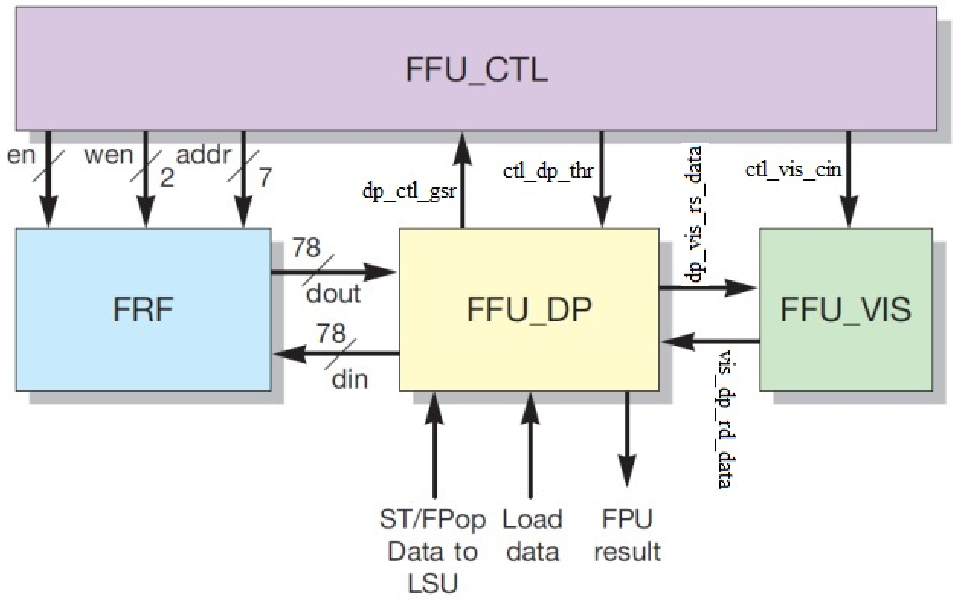


Figure 2.9: Top-Level FFU Block Diagram [5]

The following steps are taken when the FFU detects a floating-point operation (Fpop):

- The thread switches out.
- The Fpop is further decoded and the FRF is read.
- Fpops with operands are packetized and shipped over the crossbar to the FPU.

- The computation is done in the FPU and the results are returned by way of the crossbar.
- Writeback completed to the FRF and the thread restarts.

2.3 OpenSPARC T1 design history

There have been several designs by researchers to physically implement the OpenSparc T1 processor [5]. Below we briefly desribe three of these designs. The first two target Xilinx FPGAs, while the third is the famous ASIC design: The Niagra UltraSPARC T1 processor.

2.3.1 Xilinx FPGA design 1 [6]

In 2006 Sun and Xilinx began OpenSPARC collaboration. They started with a single threaded version of the OpenSPARC T1 core and was able to download it to the Xilinx XC4VFX6 FPGA. Below is a brief specification list of this design.

- Processor configuration: Single thread, No modular arithmetic (MA) (i.e. No SPU), reduced TLB
- Size: 40K Virtex-2/4 LUTs, 30K Virtex-5 LUTs

- FPGA type: Xilinx XC4VFX6
- Performance: Meets 20ns cycle time (50MHz)

2.3.2 Xilinx FPGA design 2 [6]

SUN and Xilinx continued collaboration and was able to download a 4 threaded version of the OpenSPARC T1 core to the Xilinx XC5VLX110T FPGA. Below is a brief specification list of this design.

- Processor configuration: Four threads, No modular arithmetic (MA) (i.e. No SPU), 16-entry TLB
- Size: 69K Virtex-2/4 LUTs, 51K Virtex-5 LUTs
- Block RAMs used: v4: 127, v5: 115
- FPGA type: Xilinx XC5VLX110T
- Performance: Runs at 10 MHz

2.3.3 UltraSparc T1 (Niagara) release [6]

In 14 November 2005, Sun Microsystems released a full ASIC implementation of the OpenSPARC T1 opensource RTL design. This processor was called the UltraSPARC T1 and was codenamed "Niagara". Below is a short specification list of this processor:

- Processor frequency: 1 - 1.4 GHZ
- Processor cores: 8
- Fabrication Process: 90nm
- Die Size: 340 mm sqr
- processor L1 cache size: 16 Kbytes
- processor L2 cache size: 3072 Kbytes
- processor power: 72 Watt

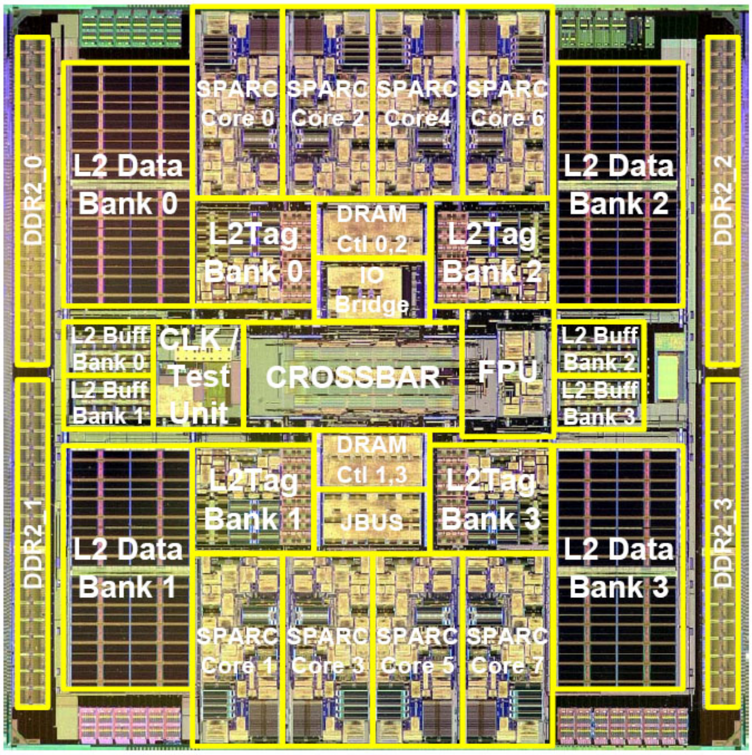


Figure 2.10: Sun UltraSPARC T1 (Niagara 8 Core) [6]

2.3.4 Commercial processors based on the Ultra-Sparc architecture [1]

The below table shows a comparison between diffrent commercial processors based on the UltraSparc architecture.

Name (codename)	Frequency (MHz)	Arch. version	Year	Total threads	Process (µm)	Transistors (millions)	Die size (mm²)	IO Pins	Power (W)	Voltage (V)	L1 Dcache (KiB)	L1 Icache (KiB)	L2 Cache (KiB)	L3 Cache (KiB)
UltraSPARC (Spitfire)	143–167	V9	1995	1×1=1	0.47	3.8	315	521	30	3.3	16	16	512-1024	none
UltraSPARC (Hornet)	200	V9	1998	1×1=1	0.42	5.2	265	521	--	3.3	16	16	512-1024	none
UltraSPARC IIs (Blackbird)	250–400	V9	1997	1×1=1	0.35	5.4	149	521	25	2.5	16	16	1024 or 4096	none
UltraSPARC IIs (Sapphire-Black)	360–480	V9	1999	1×1=1	0.25	5.4	126	521	21	1.9	16	16	1024–8192	none
UltraSPARC IIi (Sabre)	270–360	V9	1997	1×1=1	0.35	5.4	156	587	21	1.9	16	16	256–2048	none
UltraSPARC IIi (Sapphire-Red)	333–480	V9	1998	1×1=1	0.25	5.4	--	587	21	1.9	16	16	2048	none
UltraSPARC IIe (Hummingbird)	400–500	V9	1999	1×1=1	0.18 Al	--	--	370	13	1.5-1.7	16	16	256	none
UltraSPARC IIi (IIe+) (Phantom)	550–650	V9	2000	1×1=1	0.18 Cu	--	--	370	17.6	1.7	16	16	512	none
UltraSPARC III (Cheetah)	600	V9 / JPS1	2001	1×1=1	0.18 Al	29	330	1368	53	1.6	64	32	8192	none
UltraSPARC IIIi (Jalapeño)	1064–1593	V9 / JPS1	2003	1×1=1	0.13	87.5	206	959	52	1.3	64	32	1024	none
UltraSPARC IV (Jaguar)	1050–1350	V9 / JPS1	2004	1×2=2	0.13	66	356	1368	108	1.35	64	32	16384	none
UltraSPARC IV+ (Panther)	1500–2100	V9 / JPS1	2005	1×2=2	0.09	295	336	1368	90	1.1	64	64	2048	32768
UltraSPARC T1 (Niagara)	1000–1400	V9 / UA 2005	2005	4×8=32	0.09	300	340	1933	72	1.3	8	16	3072	none
UltraSPARC T2 (Niagara 2)	1000–1600	V9 / UA 2007	2007	8×8=64	0.065	503	342	1831	95	1.1–1.5	8	16	4096	none

2.4 Proposed Design

This section describes in more details a proposed implementation of the OpenSPARC T1 processor core. There is an incomplete implementation of the OpenSPARC T1 processor core that stops at the synthesis stage. This implementation was made by Professor Jose Renau from University of California at Santa Cruz [7]. His implementation was to take the OpenSPARC T1 RTL blocks as input and synthesizes them to gates using Synopsys Design Compiler. Using TSMC 65 nm technology available to Universities from Synopsys, the design can be run at 950 MHz with no caches.

Work on this thesis project started targeting a complete ASIC design of the OpenSPARC T1 processor core. The first design issue appeared was to find a complete ASIC technology library that can be used to implement the processor. The only complete (Includes memory generators and standard cells GDS description) ASIC library that was available to work with at the time work started on this thesis was the IBM 130 nm CMOS technology. The PDK (process design kit) for this technology was available through MOSIS MEP account # 4960.

The second issue appeared was that in order to produce a design that can be fabricated under this educational program offered by Mosis, the design dimensions should not exceed 4 mm X 4mm. Taking into consideration the core size and the technology used for fabrication it was nearly impossible to produce an ASIC that can function properly under these conditions. To make sure that this is true, direct scaling of the area of one core of the published UltarSPARC T1 processor [8] at 90 nm technology to a 130nm technology shows that it needs an area of 23 mm2 and that does not include the PADs. So a decision was made to make some functionality sacrifices in order to reduce the size of the core.

The functionality modifications that were made can be summarized in the following points:

- Changing the number of threads from four to two. This change will not affect the operation of the core, it will only reduce its performance and reduces the size of the netlist by up to 20%. The external interfaces of the chip are not affected by this change.

- Reducing the TLB size. This reduction reduces the size of the design. This reduction will not affect the core functionality but it will reduce the TLB hit rates, which will reduce the performance.

- Removing the Stream Processing Unit (SPU). This removal reduces the size of the design. The SPU is used to accelerate cryptographic functions. It contains a modular arithmetic unit. All modular arithmetic operations will not be supported after this removal.

These modifications are suggested by reference [9] and are implemented by the first Xilinx FPGA example in the previous section. More details on these modifications can be found in chapters 3 of this thesis documentation and in reference [9].

Moving on with the design many other challenges surfaced and were handled, until a complete ASIC component was produced, Appendix A describes in more details the design flow that was used to reach this final stage.

Chapter 3

OpenSPARC T1 core Implementation

In this chapter we will describe in more details the attempt to implement the OpenSPARC T1 core following the flow described in appendix A.

3.1 RTL preparation and functional verification

The OpenSPARC T1 design RTL code is available from Sun Microsystems under a GPLv2 license for public use. It is provided in a suite that also includes its verification environment and simulation tests that test its functionality and performance. So first this verification environment was setted up and the available tests were ran to make sure that the design was functioning properly as intended.

The provided verification environment of OpenSPARC T1 processor is based on Cadence VCS verification tool. All tests run correctly on this environment. Additionally, this environment was modified to run on Mentor Graphics Questa, which was more readily available to me.

After verifying the RTL code, preparations for the ASIC flow started, I started by replacing the memory modules from the design by memory blocks from the memory generator of the ASIC vendor. The OpenSPARC T1 core contains 11 memory modules which are [10]:

1. Instruction Cache Data (ICD).
2. Data Cache Data Array (DCD).
3. Instruction and Data Cache Tag (IDCT).
4. Floating-Point Register File (FRF).
5. Integer Register File (IRF).

6. Translation Lookaside Buffer (TLB).
7. Store Buffer CAM.
8. Register File 16 x 32 (RF16x32).
9. Register File 16 x 160 (RF16x160).
10. Register File 32 x 80 (Rf32x80).
11. Register File 32 x 152 (RF32x152).

Two main difficulties appeared during this conversion process:

1. Each of these memory modules are built up using registers, the coding style that was used by these blocks are written in a way that made replacing the registers with memory blocks a hard task without recoding the full memory block. For example the already existing memory interface logic was designed to be compatible with the memory in its register form where groups of those registers were handled separately at different clock edges. This was mainly done to ease the coding of such memory blocks. It has no impact on performance. A lot of modifications and sometimes a complete rebuild were made to the interface logic of these memory blocks to retain the same functionality and performance. (Example: ICD and DCD blocks)

2. Due to the restrictions on the size of the memory blocks that can be generated by the memory generator, some adjustments were made (ex: partitioning or grouping) to the memory block to achieve a block size that can be generated by the generator.

The supplied design kit that is available from the technology provider contains four types of memory generators which are single port SRAM, double port SRAM, single port register file, and double port register file. Each one of these generators is used to generate a certain type of memory depending on its functionality. The following is the list of memories used for each memory type:

1. Single-Port SRAM: ICD, DCD.
2. Dual Port SRAM: FRF, IDCT.
3. Two-Port Register File: RF16x32, RF16x160, Rf32x80, RF32x152

The memory conversion process was as follows:

1. Understand the current implementation, functionality and timing of the memory module.
2. Generate a memory block of the same size and type of the existing memory module.

3. If there is a size restriction by the generator on the memory block, then the memory is divided into symmetrical smaller memory blocks.

4. Try to replace the register implementation with the generated memory block without modifying the interface logic as much as possible.

5. If needed the interface logic can be modified or completely replaced to preserve the functionality of the memory module using the generated memory block.

I was successfully able to replace 8 of the 11 modules by memory blocks from the generator; the remaining 3 blocks could not be replaced due to the following reasons:

1. The Store Buffer CAM memory module contains a CAM (content addressable memory), which need a special memory generator that is not supplied by the ASIC vendor.

2. The Integer Register File (IRF) structure requires the access of different register files in the same clock cycle which cannot be implemented using a memory block without adding huge overhead which will make the use of memory blocks of no benefit.

3. The Translation Lookaside Buffer (TLB) is a fully associative memory, it consists of content addressable memory (CAM) which cannot be replaced by a memory block.

These three memory modules were left in their original register-based structural description provided by Sun Microsystems [5] without any modification.

To make sure that the core function was not altered during these modifications, each of the modified memory blocks was tested separately using the verifications suite. After that all the modifications were imported to the design, and then the full RTL design was tested to make sure that everything is working as intended.

Another modification that was made to the design at this stage was the addition of Pads. The design contains 330 signal I/O pins, each of these signal pins needs their own signal Pads. In addition to that, the design needs power Pads. Given the size of the chip and the size of these signal Pads, It was impossible to fit those Pads in the chip area. Some pins in the design were hard wired to constant values. For example the CPU ID pins that give an ID for the CPU core when this core is used in the full OpenSPARC implementation that has 8 cores. These pins are of no use if the core is used separately, so it is OK to hardwire it. Another example of pins that were hard wired are the SCAN pins, doing so removed the functionality of hardware testing from the processor, this will not affect any main functionality of the processor but will disable hardware testing of the processor after the processor is fabricated. This is a sacrifice that needed to be made to enable the processor to fit in the 4x4 mm^2 chip area.

As mentioned before the technology I am using to implement the processor core is the IBM 130 nm supplied from Mosis under an educational license. To fabricate any chip at Mosis using this technology under this educational license there is a limitation on the size of the chip core area to be less than or equal to 4x4 mm^2. Driven by this constraint I had to limit my design size to 4x4 mm^2, this should include all design components (i.e. pads, cells, wiring etc). My first trials to fit the full processor core without any size reduction failed, this is the reason behind the size reduction modifications described in the previous section.

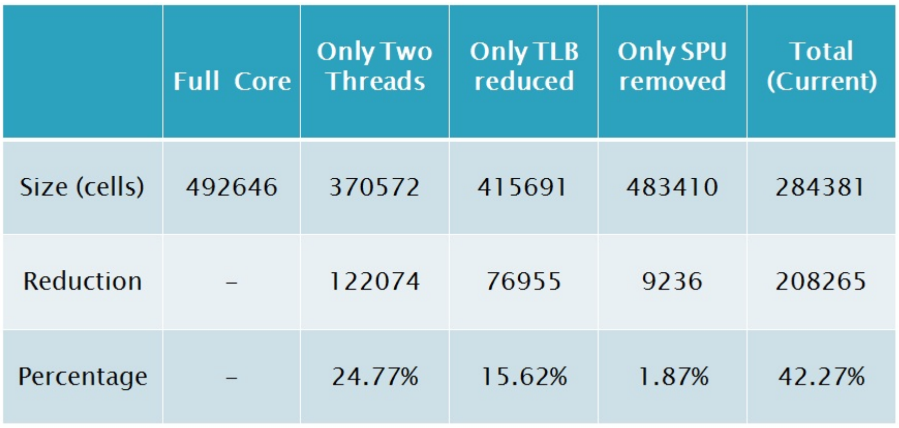
Several modifications were made to the design in order to reduce its size; as suggested in [9]:

1. Changing the number of threads from four to two.

2. TLB size reduction from 64 entries to 8 entries.

3. Removing the Stream Processing Unit (SPU).

Table 3.1 shows the reduction in size for each of the above points.



	Full Core	Only Two Threads	Only TLB reduced	Only SPU removed	Total (Current)
Size (cells)	492646	370572	415691	483410	284381
Reduction	–	122074	76955	9236	208265
Percentage	–	24.77%	15.62%	1.87%	42.27%

Table 3.1: Reduction in the processor core

As mentioned before these modifications are essential to reduce the size of the design to be able to fit it in the design core. These modifications will reduce the performance of the design but will preserve most of its functionality. A hardware thread is a collection of processor resources where a group of these resources is unique to a certain thread and the other group is shared among other threads. These resources are used to execute threaded program code simultaneously.

The processor core has multiple numbers of threads; each thread resources include its registers, a portion of I-fetch data-path, store buffer, and miss buffer. The shared resources of multiple threads include the pipeline registers and data-path, caches, translation lookaside buffers (TLB), and execution unit of the SPARC Core pipeline.

The number of threads was reduced from four to two by removing the extra resources needed by the extra threads, for example the instruction fetch unit (IFU) is responsible for maintaining the program counters (PC) of different threads, as well as managing and scheduling all the threads in a SPARC core. The IFU includes instruction buffer which includes two instruction registers per thread, the thread instruction register (TIR) and the next instruction register (NIR). The IFU also includes the integer register file (IRF), each thread requires 128 registers for the eight windows (with 16 registers per window), and four sets of global registers with eight global registers per set. There are 160 registers per thread. Removing the extra resources of the extra threads I was able to reduce the number of threads from four to two. The following is a list of the design main blocks that needed modifications in the same way:

1. Instruction Fetch Unit (IFU).

2. Load Store Unit (LSU).
3. Floating-Point Frontend Unit (FFU).
4. Execution Unit (EXU).
5. Trap Logic Unit (TLU).

The code of the SPARC core contains switches that guided me to the portion of the code that needed to be modified in each of the above design blocks. The modification to the design blocks had to be made all at once before any verification tests could be run to test the functionality of the modified blocks.

Another modification that was made to reduce the size of the core was to reduce the TLB size. There are two TLBs used in the processor core, the instruction TLB (ITLB) resides in instruction fetch unit (IFU), and the data TLB (DTLB) resides in load and store unit (LSU). In this modification both the instruction TLB size and data TLB size were reduced from 64 entries to 8 entries. The reduction of the TLB size will not affect the functionality of the processor but will reduce the TLB hit rates, which will in fact reduce the performance.

The Stream Processing Unit (SPU) of the core was also removed in order to reduce the core size; The SPU is used to accelerate cryptographic functions of the core. SPU contains a modular arithmetic unit. Removing it will remove this functionality from the processor core. This sacrifies is made in order to reduce the core area.

After applying these modifications to the design, the design was verified using the supplied verification suite. At the beginning there were several failures in the verification tests. After investigating these failures, I found that some of these failures were due to coding mistakes which were fixed and these tests passed. The rest of failures were in the tests that were intended to work on four threads, but due to my modifications they were failing, I manually checked all of the failing tests and verified that these failures were of the same reason. These failures were waived away as they were expected to happen.

3.2 Synthesis

To synthesis the design, the Design Compiler [11] tool from synopsis was used. The Design Compiler is considered the best synthesis tool in the market in the time of writing this thesis documentation. The synthesis process was done in two main stages because the size of the design is big and the resources (computer machines) that were available to me to do the job were not powerful enough to handle the full design at once.

The first stage of the synthesis process was to synthesize each of the small design blocks separately. This was done using an automated script [12] that is supplied by the processor design kit. This script automatically loops on all the design blocks and synthesizes them using a generic synthesis script that is parameterized by a set of variables stored in the run directory of each of these design blocks. The outcome of this stage is a design database for each of the design blocks. Some of the design blocks were not included in the automated script due to their different nature. Examples of these blocks include the memory modules, which were synthesized separately using separate synthesis scripts.

The objective of the second stage of the synthesis process is to generate the top level synthesized netlist. This was done by writing a synthesis script that loads the synthesized memory blocks and then marks them as don't modify blocks. This marking forces the synthesis engine treat those design blocks as fixed design components and will not change them. Then, the top-level netlist is loaded, and the design is synthesized completely. The outcome of this stage is a synthesized netlist that describes the design completely.

3.3 Design Floorplanning

I am using Olympus SOC [13] from Mentor Graphics for the P&R flow. I started the floorplan stage by importing the required design files LEF, LIB, Verilog, PTF and SDC. Then, I made a Zero RC optimization run to make sure that there are no constraints issues. Then, I defined the chip area to be 4X4 mm, after that I defined the pads area and the core area. To arrange the pads location, first I had to estimate roughly where the design blocks will be placed depending on their relation to each other so as to guarantee the minimum wire length between the pad and the design elements. Fig. 3.1 shows a rough estimation of the blocks location found in the processor documentation as a guideline.

To order the design pads a location constraint file had to be written to guide the tool as to where to place the pads. This file contains information like on which side should the pad be placed and what is the order of the pads placement on each side. After that, the tool is instructed to place the pads using this constraint file. After placement the pads are marked as fixed location so they are not moved later in

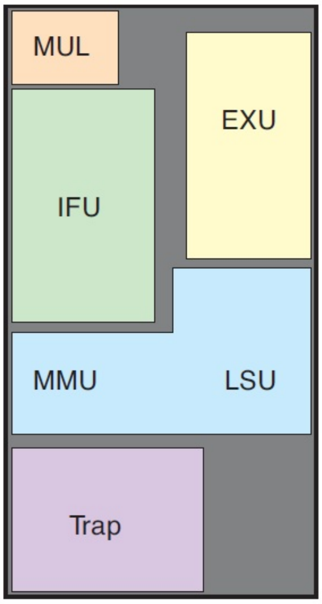


Figure 3.1: Processor blocks rough estimation [5]

the design.

The core area is defined as the remaining chip area after the pad placement. Usually there is a small gap that is left between the core area and the pads. This gap is needed to insure easy route access to the pad pins. Due to the tight area constraint this gap was ignored to use its space for cell placement. This will increase the risk that the pad pins get blocked. So caution during placement stage must be taken to avoid this.

The next step now is to define the power grid. Power rings and stripes were implemented using Metal 5 and Metal 6 of the technology, after that power rails on Metal 1 were implemented, then the rails, rings and stripes were connected together to form the power grid. This grid was then connected manually to the supply pads placed on the top and bottom sides of the chip.

Figure 3.2 shows the power grid design.

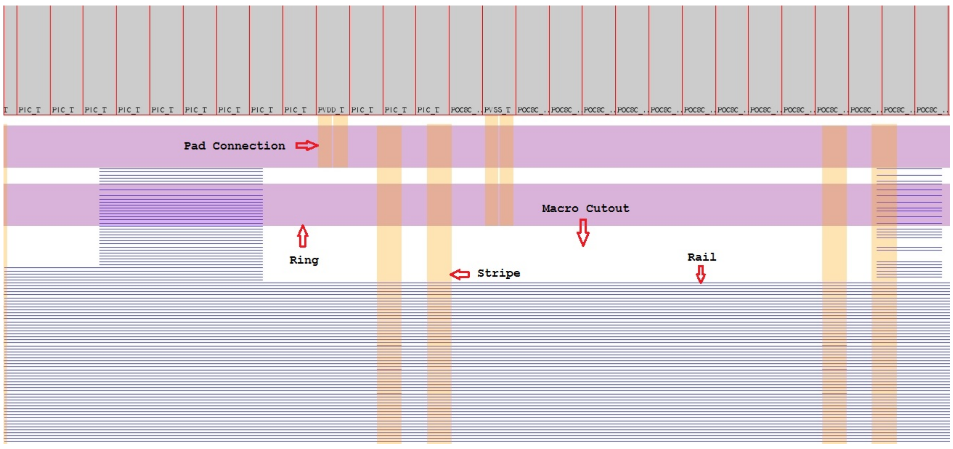


Figure 3.2: Design Power Grid

Macro cells (memory blocks) were then placed. To find the best location of where to place the Macro cells I used the following technique. First, the tool automatically guess the initial location of each block. The tool uses information from the design hierarchy and timing calculations to determine these location. After that knowing the relation between each of these blocks I manually modified its location to give better connectivity to the rest of the design, In order to get the best use of the available area, I pushed the memory blocks near to the top and bottom chip edges to occupy locations where the standard cells will not be placed (no standard cell rows). Macro cells are then marked as fixed and connected to the power grid manually.

Figure 3.3 shows the Macro placement of the design.

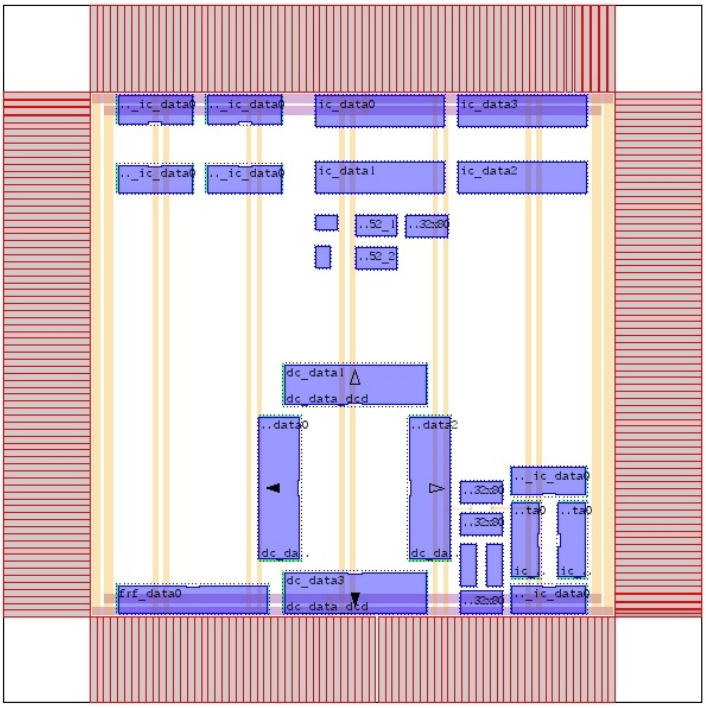


Figure 3.3: Macro placement

Figure 3.4 shows the empty space used up by macros.

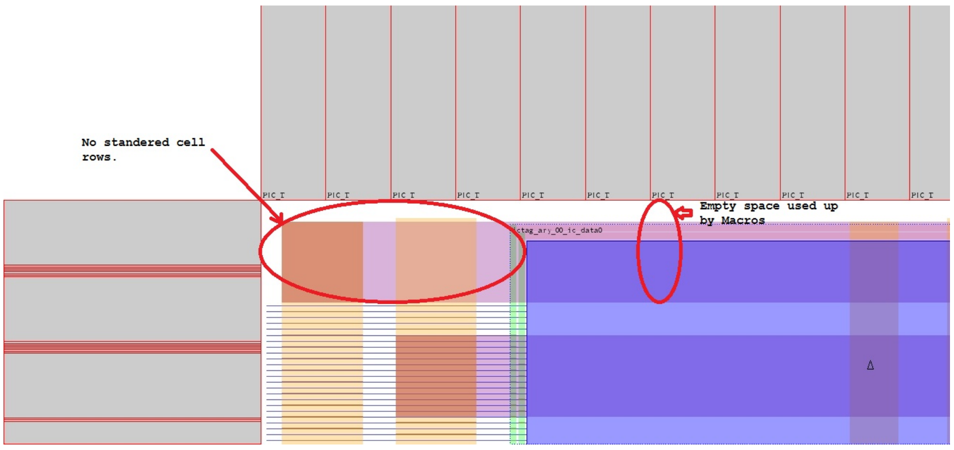


Figure 3.4: Empty spaces used up by Macros

Figure 3.5 shows an example of the power connection of the macros.

3.4 Design Placement

As a preparation step for placement, High Fan Out Net Synthesis (HFNS) must be run first. In HFNS all high fan-out nets except clock nets are buffered to reduce the fan-out of each of them, the tool has a threshold value for the number of fan-outs above which this net is considered a high fan-out net. This value is set to 128 in our design, which is the default value set by the tool. This value can be changed by the user. If after HFNS there is still timing issues due to nets driving

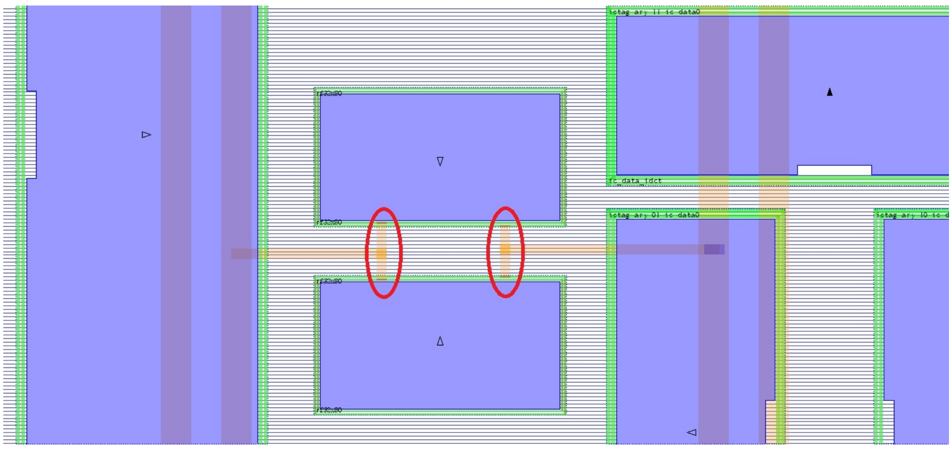


Figure 3.5: Macro Power Connection

large number of cells, the optimizer can easily fix them. There were 112 high fan-out nets in my design, the tool synthesized and buffered those nets.

The design has no regions or partitions, the design will be totally placed by the tool in the same run guided by the hierarchal information that lies in the design netlist. Placement is done in two main steps, global placement and detailed placement. In global placement the design standard cells will be roughly placed in the core area, after that the detail placer will legalize the locations of these cells so as not to create DRC violations.

Figure 3.6 shows the cell placement of the top design blocks.

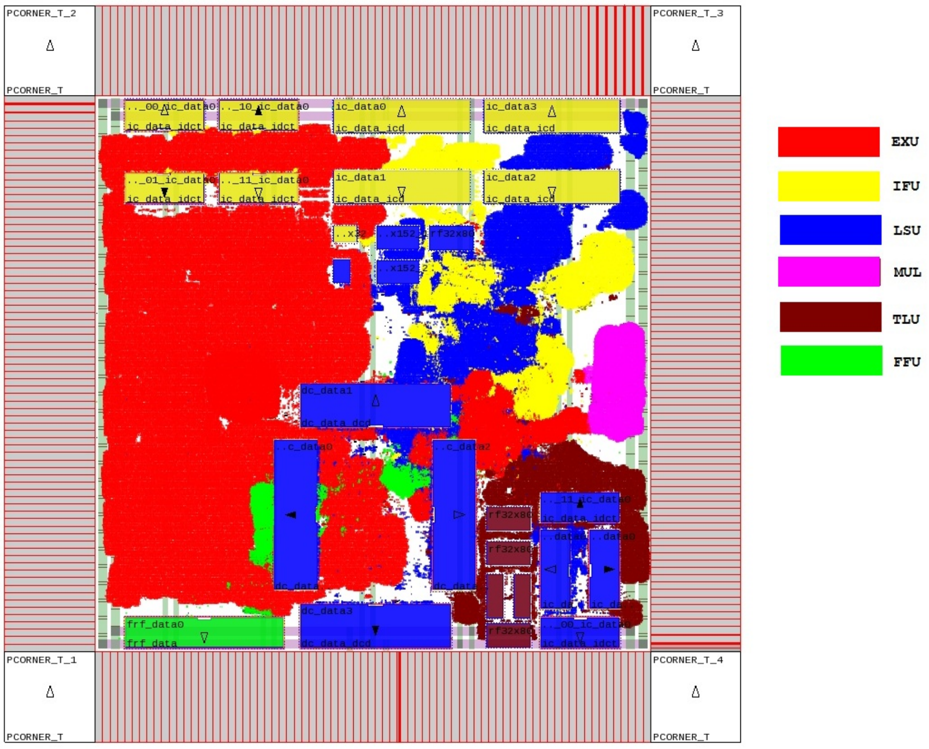


Figure 3.6: Top blocks placement

From the above picture you can notice that the tool has placed the cells in a different distribution than specified in Fig. A.4. This is expected as the distribution of the pads and Macro cells are different from the one assumed by Fig. A.4. Figure A.4 is just a guide and is not a real implementation. Also you can notice the size difference in the block sizes and again this is because the blocks implementation is different from the ones assumed in Fig. A.4. Changing any design environment like pads, Macro cells or block implementation will definitely change the placer behavior while placing the design.

3.5 CTS

As mentioned in appendix A the CTS stage is divided into three steps. In the first step we do a pre-CTS optimization on the design. The first optimization run in the pre-CTS step is to optimize the max transition and max capacitance violations, after that global optimization and local optimization is run, the designer must note that the run time of the pre-CTS stage is big as the design has not been optimized before. The designer must keep an eye on the optimization process while running and make sure that timing is converging. If timing is not converging then the designer must stop the optimization process and inspect the design manually for problems that could obstruct the optimization process. Examples of these issues are:

1. Optimization buffers are set to don't use in the library, or no enough buffer driver strength varieties for the optimizer to choose from.

2. No enough space or high utilization of cells in a certain area of the design where the optimizer needs to place buffers.

3. Non realistic constraints or over tightened constrains.

After the pre-CTS optimization, the design is ready now for the CTS process. In our design we have only one clock domain, therefore the full CTS tree will be built in one compilation run. The CTS engine uses buffers with equivalent rise and fall propagation delays, so the designer must make sure to enable the use of these buffers and define them to the CTS engine. The designer must also specify some CTS specs if needed for the CTS engine to follow, like for example maximum fan-out, maximum transition, or maximum capacitance.

After the tool builds the clock tree, the clock is finally routed using all available metal layers to guarantee short connections and lowest delays. The clock network is then marked as fixed so that its elements and routing will not be altered in the remaining flow stages to keep its timing constant.

Now realistic timing information of the clock network is available so a post-CTS optimization flow is needed to fix hold violations and any remaining setup violations. In some cases due to bad definition of some constraints a conflict could occur on a path where it is violating both setup and hold. In this case this path is not fixable using optimization and the designer must fix the timing constraint manually. This kind of conflict could lead to long run time of the post-CTS optimization engine because it is trying to fix a non fixable timing path.

3.6 Routing

Routing was done using all 6 metal layers; the routing stage was done in 2 steps, track routing and final routing. Before starting the routing process the existing global routes were optimized to reduce both timing and congestion, this was done by launching several global routing optimization runs on the design.

Next, track routing was used to replace the global routes with real metal layers, which was then optimized using final route. Several iterations of final routing were needed to fix all DRC errors. The final router used tricks like off-grid routing, and non-preferred metal direction routing to fix hard to fix DRC errors.

3.7 Post Layout Verification

In post layout verification the design must be verified to make sure that it is functionally operating and must be checked physically to make sure that it meets all fabrication rules. The design was tested functionally by two methods; first it was tested using formal verification, where the final netlist of the design was loaded into the formal verification tool and compared to the netlist that was used to initially implement the design. Theoretically they should match because the only change that was made to the netlist was the addition of optimization buffers which will not affect the functionality in anyway, and that was proved from the verification results.

To verify the design using simulation, it was a tricky job, as the supplied test suite has its own monitors that monitors the design signals to check if the applied test passed or not. And since the design has undergone a lot of structural modification these monitors could no longer find the signals to monitor. To overcome this issue a modification to the testing environment was made. First a new monitor was added to the original testing environment, the function of this monitor is to dump the signal values of the processor core external pins on every clock cycle, either for input or output. Then this new test environment was run on the original available tests to dump the values of the signals for each test.

After that a new testing environment was build, the objective of this testing environment is to apply the input values dumped from the monitor to the input pins of the processor, and then compare the output of the processor output pins with the dumped outputs of the monitor. Theoretically if the two design match then the output of the final processor must match the output of the original processor, and this was proven by the simulation results. During this simulation test SDF file was loaded to account for design delays.

To physically check the design, the design was tested for DRC and LVS violations. DRC was checked by loading the DRC rule file provided by the FAB into the DRC checking tool and then the tool was run to make sure that the design passes all rules. Some violation were found, these violations were fixed using an iteration of final routing.

LVS was checked by extracting the netlist of the design using the extraction tool and then comparing the extracted netlist to the original design netlist, no violations were found after LVS checking.

Chapter 4

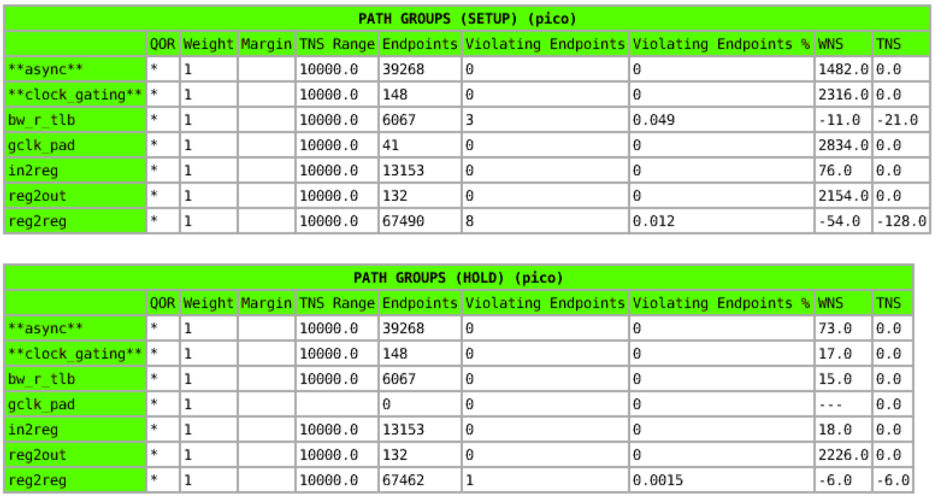
Results

In this chapter we will review all the design results that were achieved from running the design flow. The results are divided into two main groups physical results and timing results. In the physical results section we will display results related to the physical implementation of the design like area, wiring ...etc. In the timing result section we will review the design timing data like slack, delays ...etc. We will also analyze some of these results and explain their reasons.

4.1 Timing results

The design timing was tested using static timing analysis (STA). In STA you define the design constraints and then test the design and see if it passes them. These constraints are the same constraints used to build up the design. The design was constrained to run on 100 MHz clock. This frequency was chosen based on several trial runs that were made with higher frequencies but failed to complete the flow cleanly. These runs were made by tightening the timing constraints and increasing the clock frequency. Then the full ASIC flow is applied on the design. Then timing is checked through STA. The 100 MHz frequency is not the exact maximum speed the design can reach but it is a rough estimate. Tables 4.1 shows the setup and hold path_groups_timing report of the design.

Path groups are groups of timing endpoints that have a common property. This is a technique used by the tool in order to categorize timing endpoints into several groups for better readability to ease the process of analyzing the results. From the tables above we can see some specified groups; the main ones are in2reg, reg2reg and reg2out. In2reg path group is the group of endpoints that start from an input pin and ends at a register, reg2reg is the group of endpoints that



PATH GROUPS (SETUP) (pico)									
	QOR	Weight	Margin	TNS Range	Endpoints	Violating Endpoints	Violating Endpoints %	WNS	TNS
async	*	1		10000.0	39268	0	0	1482.0	0.0
clock_gating	*	1		10000.0	148	0	0	2316.0	0.0
bw_r_tlb	*	1		10000.0	6067	3	0.049	-11.0	-21.0
gclk_pad	*	1		10000.0	41	0	0	2834.0	0.0
in2reg	*	1		10000.0	13153	0	0	76.0	0.0
reg2out	*	1		10000.0	132	0	0	2154.0	0.0
reg2reg	*	1		10000.0	67490	8	0.012	-54.0	-128.0

PATH GROUPS (HOLD) (pico)									
	QOR	Weight	Margin	TNS Range	Endpoints	Violating Endpoints	Violating Endpoints %	WNS	TNS
async	*	1		10000.0	39268	0	0	73.0	0.0
clock_gating	*	1		10000.0	148	0	0	17.0	0.0
bw_r_tlb	*	1		10000.0	6067	0	0	15.0	0.0
gclk_pad	*	1			0	0	0	---	0.0
in2reg	*	1		10000.0	13153	0	0	18.0	0.0
reg2out	*	1		10000.0	132	0	0	2226.0	0.0
reg2reg	*	1		10000.0	67462	1	0.0015	-6.0	-6.0

Table 4.1: Setup and hold timing path groups

starts at a register and ends at another register. Likewise reg2out is the group of endpoints that start at a register and ends at an output port.

WNS and TNS columns of the above table specify the worst and total negative slack. Slack is the difference between the required delay time calculated from the defined constraint and the actual delay time extracted from the design. A negative slack number means that the design is violating by the value of this number. WNS is the slack of the worst violating path. TNS is the sum of all the negative slacks in the path group. From the tables above the WNS and TNS are not zero but small values, this is considered acceptable design closure state, the remaining violations can be manually fixed before the design is shipped for fabrication. Figures 4.1 and 4.2 show the setup and hold slack distributions of the design.

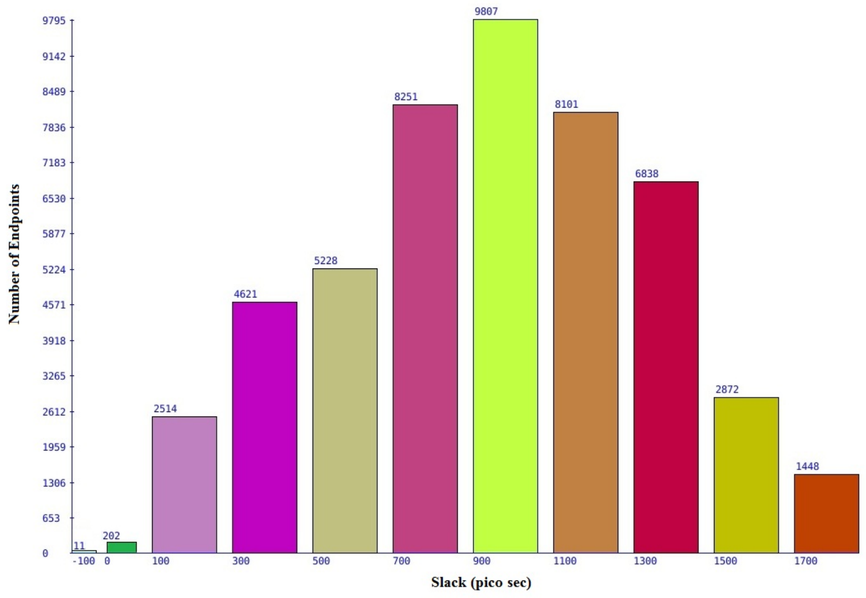


Figure 4.1: Setup slack distribution

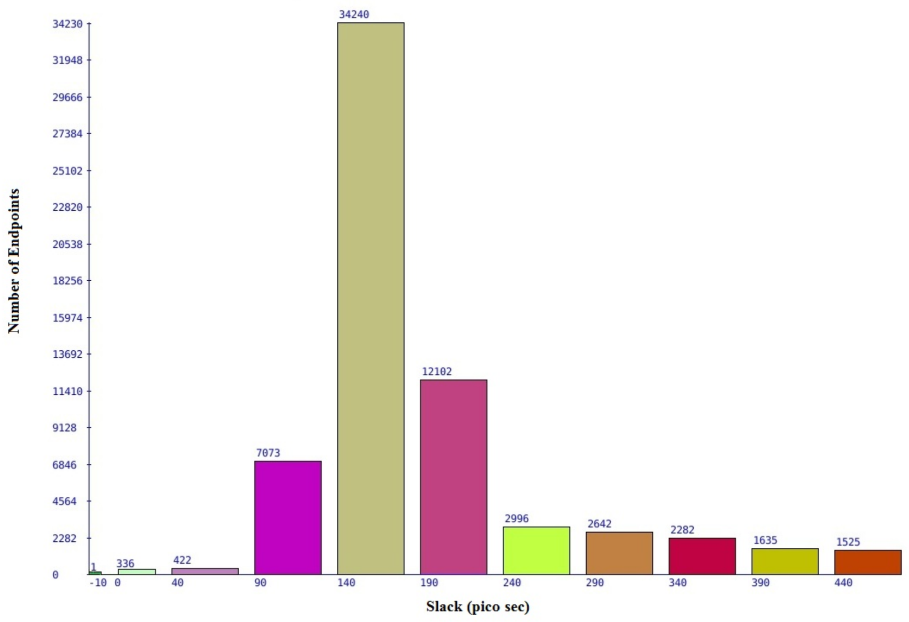
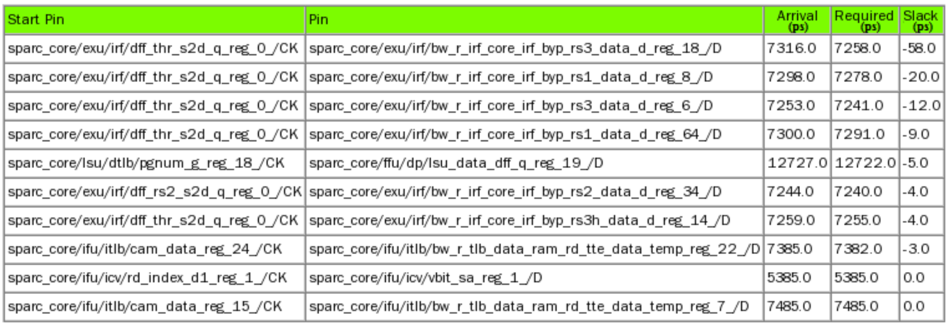


Figure 4.2: Hold slack distribution

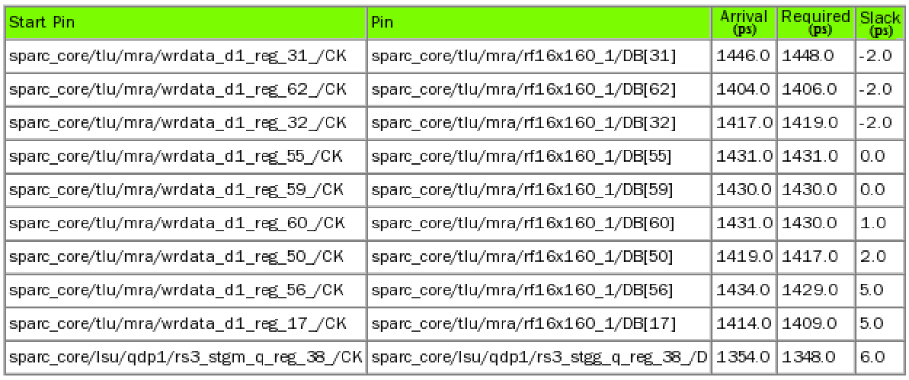
The slack distribution figures show the relation between the number of endpoints (y-axis) and the slack (x-axis). You can notice from the figures that the median of the graph lies in the positive side with a small margin, which indicated that the design is passing the specified design constraints. This also shows that the design is just passing the applied timing constraints and cannot be improved more to achieve better performance.

Table 4.2 shows the worst 10 setup timing path and table 4.3 shows the worst 10 hold timing path.



Start Pin	Pin	Arrival (ps)	Required (ps)	Slack (ps)
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs3_data_d_reg_18_/D	7316.0	7258.0	-58.0
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs1_data_d_reg_8_/D	7298.0	7278.0	-20.0
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs3_data_d_reg_6_/D	7253.0	7241.0	-12.0
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs1_data_d_reg_64_/D	7300.0	7291.0	-9.0
sparc_core/lsu/dtlb/pgnum_g_reg_18_/CK	sparc_core/ffu/dp/lsu_data_dff_q_reg_19_/D	12727.0	12722.0	-5.0
sparc_core/exu/irf/dff_rs2_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs2_data_d_reg_34_/D	7244.0	7240.0	-4.0
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs3h_data_d_reg_14_/D	7259.0	7255.0	-4.0
sparc_core/ifu/itlb/cam_data_reg_24_/CK	sparc_core/ifu/itlb/bw_r_tlb_data_ram_rd_tte_data_temp_reg_22_/D	7385.0	7382.0	-3.0
sparc_core/ifu/icv/rd_index_d1_reg_1_/CK	sparc_core/ifu/icv/vbit_sa_reg_1_/D	5385.0	5385.0	0.0
sparc_core/ifu/itlb/cam_data_reg_15_/CK	sparc_core/ifu/itlb/bw_r_tlb_data_ram_rd_tte_data_temp_reg_7_/D	7485.0	7485.0	0.0

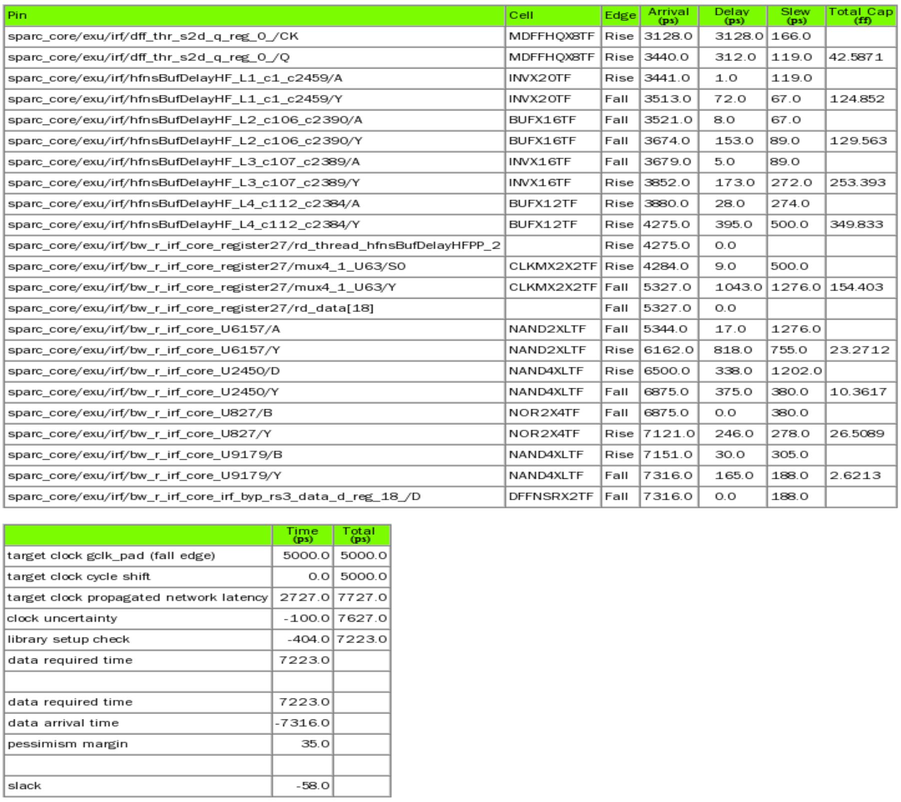
Table 4.2: Worst 10 setup paths.



Start Pin	Pin	Arrival (ps)	Required (ps)	Slack (ps)
sparc_core/tlu/mra/wrdata_d1_reg_31_/CK	sparc_core/tlu/mra/rf16x160_1/DB[31]	1446.0	1448.0	-2.0
sparc_core/tlu/mra/wrdata_d1_reg_62_/CK	sparc_core/tlu/mra/rf16x160_1/DB[62]	1404.0	1406.0	-2.0
sparc_core/tlu/mra/wrdata_d1_reg_32_/CK	sparc_core/tlu/mra/rf16x160_1/DB[32]	1417.0	1419.0	-2.0
sparc_core/tlu/mra/wrdata_d1_reg_55_/CK	sparc_core/tlu/mra/rf16x160_1/DB[55]	1431.0	1431.0	0.0
sparc_core/tlu/mra/wrdata_d1_reg_59_/CK	sparc_core/tlu/mra/rf16x160_1/DB[59]	1430.0	1430.0	0.0
sparc_core/tlu/mra/wrdata_d1_reg_60_/CK	sparc_core/tlu/mra/rf16x160_1/DB[60]	1431.0	1430.0	1.0
sparc_core/tlu/mra/wrdata_d1_reg_50_/CK	sparc_core/tlu/mra/rf16x160_1/DB[50]	1419.0	1417.0	2.0
sparc_core/tlu/mra/wrdata_d1_reg_56_/CK	sparc_core/tlu/mra/rf16x160_1/DB[56]	1434.0	1429.0	5.0
sparc_core/tlu/mra/wrdata_d1_reg_17_/CK	sparc_core/tlu/mra/rf16x160_1/DB[17]	1414.0	1409.0	5.0
sparc_core/lsu/qdp1/rs3_stgm_q_reg_38_/CK	sparc_core/lsu/qdp1/rs3_stgg_q_reg_38_/D	1354.0	1348.0	6.0

Table 4.3: Worst 10 hold paths.

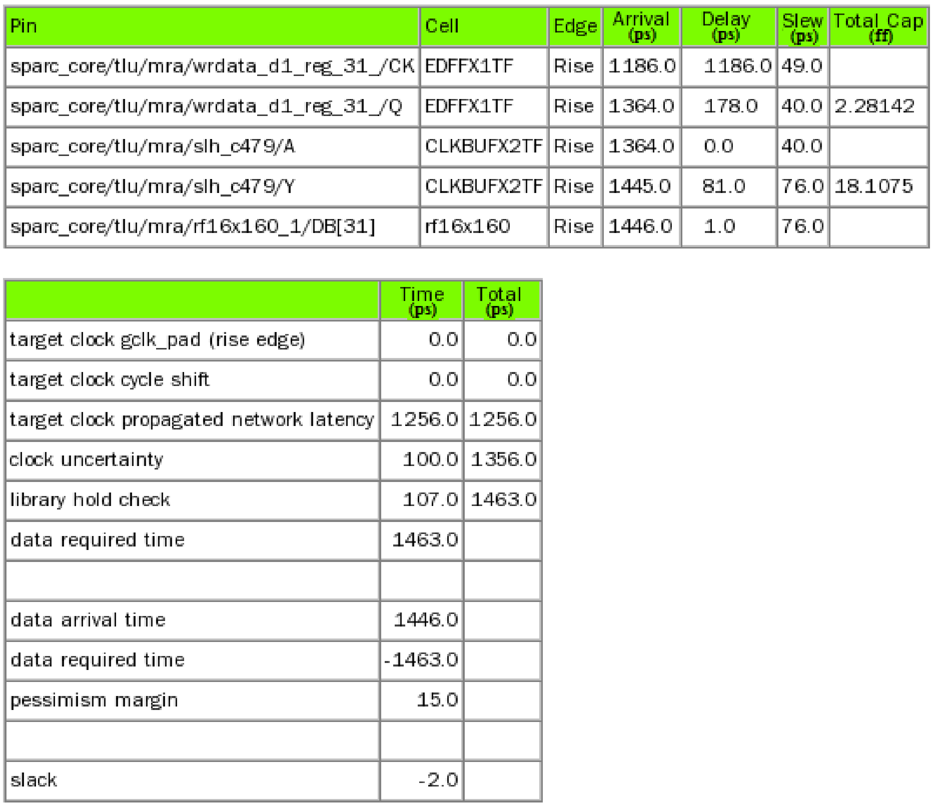
Figure 4.3 shows setup worst timing path and figure 4.4 shows hold worst timing path.



Pin	Cell	Edge	Arrival (ps)	Delay (ps)	Slew (ps)	Total Cap (ff)
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/CK	MDFFHQX8TF	Rise	3128.0	3128.0	166.0	
sparc_core/exu/irf/dff_thr_s2d_q_reg_0_/Q	MDFFHQX8TF	Rise	3440.0	312.0	119.0	42.5871
sparc_core/exu/irf/hfnsBufDelayHF_L1_c1_c2459/A	INVX20TF	Rise	3441.0	1.0	119.0	
sparc_core/exu/irf/hfnsBufDelayHF_L1_c1_c2459/Y	INVX20TF	Fall	3513.0	72.0	67.0	124.852
sparc_core/exu/irf/hfnsBufDelayHF_L2_c106_c2390/A	BUFX16TF	Fall	3521.0	8.0	67.0	
sparc_core/exu/irf/hfnsBufDelayHF_L2_c106_c2390/Y	BUFX16TF	Fall	3674.0	153.0	89.0	129.563
sparc_core/exu/irf/hfnsBufDelayHF_L3_c107_c2389/A	INVX16TF	Fall	3679.0	5.0	89.0	
sparc_core/exu/irf/hfnsBufDelayHF_L3_c107_c2389/Y	INVX16TF	Rise	3852.0	173.0	272.0	253.393
sparc_core/exu/irf/hfnsBufDelayHF_L4_c112_c2384/A	BUFX12TF	Rise	3880.0	28.0	274.0	
sparc_core/exu/irf/hfnsBufDelayHF_L4_c112_c2384/Y	BUFX12TF	Rise	4275.0	395.0	500.0	349.833
sparc_core/exu/irf/bw_r_irf_core_register27/rd_thread_hfnsBufDelayHFPP_2		Rise	4275.0	0.0		
sparc_core/exu/irf/bw_r_irf_core_register27/mux4_1_U63/S0	CLKMX2X2TF	Rise	4284.0	9.0	500.0	
sparc_core/exu/irf/bw_r_irf_core_register27/mux4_1_U63/Y	CLKMX2X2TF	Fall	5327.0	1043.0	1276.0	154.403
sparc_core/exu/irf/bw_r_irf_core_register27/rd_data[18]		Fall	5327.0	0.0		
sparc_core/exu/irf/bw_r_irf_core_U6157/A	NAND2XLTF	Fall	5344.0	17.0	1276.0	
sparc_core/exu/irf/bw_r_irf_core_U6157/Y	NAND2XLTF	Rise	6162.0	818.0	755.0	23.2712
sparc_core/exu/irf/bw_r_irf_core_U2450/D	NAND4XLTF	Rise	6500.0	338.0	1202.0	
sparc_core/exu/irf/bw_r_irf_core_U2450/Y	NAND4XLTF	Fall	6875.0	375.0	380.0	10.3617
sparc_core/exu/irf/bw_r_irf_core_U827/B	NOR2X4TF	Fall	6875.0	0.0	380.0	
sparc_core/exu/irf/bw_r_irf_core_U827/Y	NOR2X4TF	Rise	7121.0	246.0	278.0	26.5089
sparc_core/exu/irf/bw_r_irf_core_U9179/B	NAND4XLTF	Rise	7151.0	30.0	305.0	
sparc_core/exu/irf/bw_r_irf_core_U9179/Y	NAND4XLTF	Fall	7316.0	165.0	188.0	2.6213
sparc_core/exu/irf/bw_r_irf_core_irf_byp_rs3_data_d_reg_18_/D	DFFNSRX2TF	Fall	7316.0	0.0	188.0	

	Time (ps)	Total (ps)
target clock gclk_pad (fall edge)	5000.0	5000.0
target clock cycle shift	0.0	5000.0
target clock propagated network latency	2727.0	7727.0
clock uncertainty	-100.0	7627.0
library setup check	-404.0	7223.0
data required time	7223.0	
data required time	7223.0	
data arrival time	-7316.0	
pessimism margin	35.0	
slack	-58.0	

Figure 4.3: Worst setup timing path.



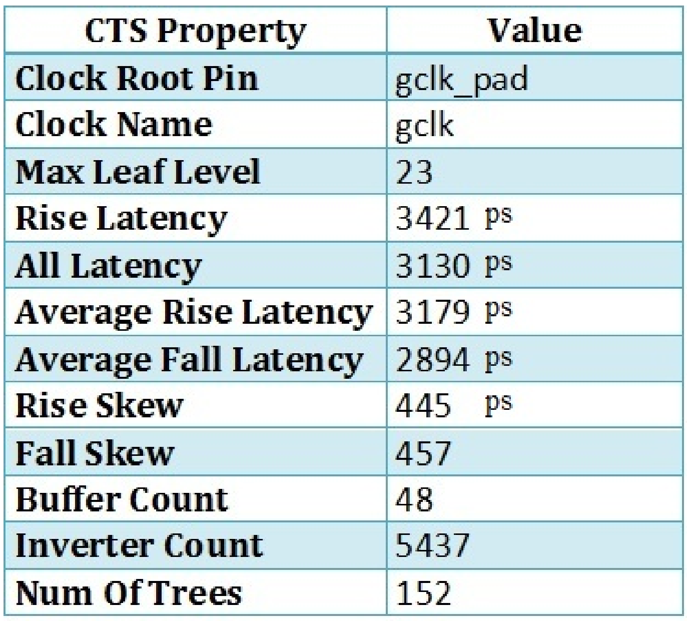
Pin	Cell	Edge	Arrival (ps)	Delay (ps)	Slew (ps)	Total Cap (ff)
sparc_core/tlu/mra/wrdata_d1_reg_31_/CK	EDFFX1TF	Rise	1186.0	1186.0	49.0	
sparc_core/tlu/mra/wrdata_d1_reg_31_/Q	EDFFX1TF	Rise	1364.0	178.0	40.0	2.28142
sparc_core/tlu/mra/slh_c479/A	CLKBUFX2TF	Rise	1364.0	0.0	40.0	
sparc_core/tlu/mra/slh_c479/Y	CLKBUFX2TF	Rise	1445.0	81.0	76.0	18.1075
sparc_core/tlu/mra/rf16x160_1/DB[31]	rf16x160	Rise	1446.0	1.0	76.0	

	Time (ps)	Total (ps)
target clock gclk_pad (rise edge)	0.0	0.0
target clock cycle shift	0.0	0.0
target clock propagated network latency	1256.0	1256.0
clock uncertainty	100.0	1356.0
library hold check	107.0	1463.0
data required time	1463.0	
data arrival time	1446.0	
data required time	-1463.0	
pessimism margin	15.0	
slack	-2.0	

Figure 4.4: Worst hold timing path.

The above timing reports show the worst 10 design timing path and the worst path in details for setup and the equivalent for hold, from the setup reports you can notice that most of the worst paths lay inside the IRF module. This shows that the IRF block is a critical block. To improve the performance of the design paths like these must be enhanced, this is discussed in more details in chapter 6.

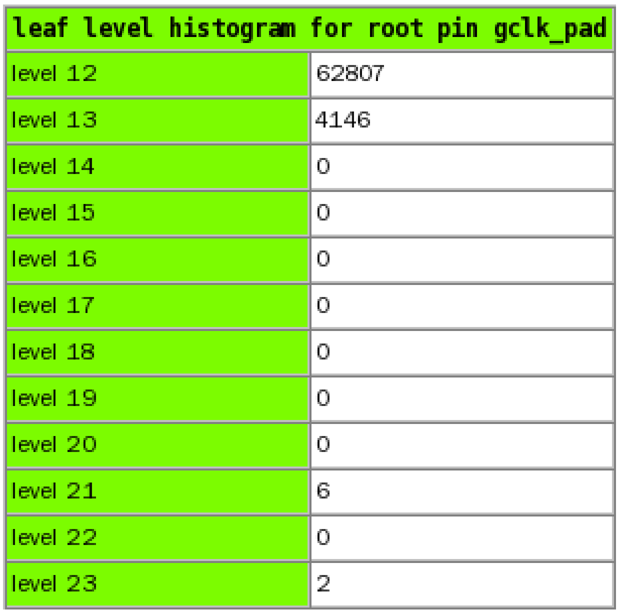
Clock network timing is another factor that affects timing closure of the design, it is important to build a proper clock network in order to get the maximum frequency for the clock. Below we will review some timing results related to the clock. Table 4.4 shows the CTS report table of the design.



CTS Property	Value
Clock Root Pin	gclk_pad
Clock Name	gclk
Max Leaf Level	23
Rise Latency	3421 ps
All Latency	3130 ps
Average Rise Latency	3179 ps
Average Fall Latency	2894 ps
Rise Skew	445 ps
Fall Skew	457
Buffer Count	48
Inverter Count	5437
Num Of Trees	152

Table 4.4: CTS Properties.

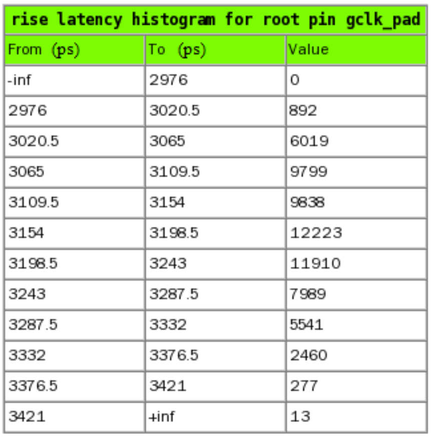
From the above table we can see that the clock tree is built mostly using inverters. This is mostly done in order to reduce the clock tree pulse width degradation issue that occurs on long clock paths. Table 4.5 shows the clock tree level histogram. This histogram shows the distribution of the clock leaf pins on the clock tree levels. We notice that the max tree levels is 23 but actually as shown from the below histogram most of the leafs lay on level 12 and 13 of the clock network.



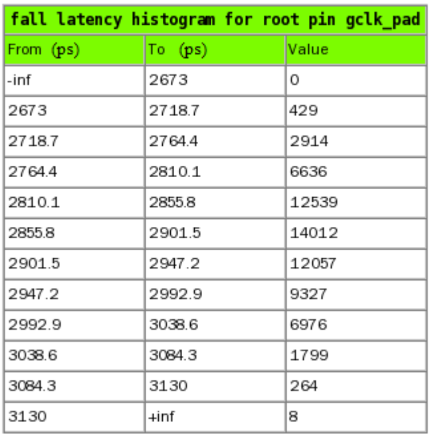
leaf level histogram for root pin gclk_pad	
level 12	62807
level 13	4146
level 14	0
level 15	0
level 16	0
level 17	0
level 18	0
level 19	0
level 20	0
level 21	6
level 22	0
level 23	2

Table 4.5: Level Tree Histogram

The above histogram shows that the clock tree has 23 levels, but truly most of the clock network is between 13 or 12 levels deep. This shows that there are some outlier clock network. If these outliers are handled then the clock network could be reduced and therefore the clock latency could decrease giving a possibility of performance enhancement, as discussed in chapter 6.



rise latency histogram for root pin gclk_pad		
From (ps)	To (ps)	Value
-inf	2976	0
2976	3020.5	892
3020.5	3065	6019
3065	3109.5	9799
3109.5	3154	9838
3154	3198.5	12223
3198.5	3243	11910
3243	3287.5	7989
3287.5	3332	5541
3332	3376.5	2460
3376.5	3421	277
3421	+inf	13



fall latency histogram for root pin gclk_pad		
From (ps)	To (ps)	Value
-inf	2673	0
2673	2718.7	429
2718.7	2764.4	2914
2764.4	2810.1	6636
2810.1	2855.8	12539
2855.8	2901.5	14012
2901.5	2947.2	12057
2947.2	2992.9	9327
2992.9	3038.6	6976
3038.6	3084.3	1799
3084.3	3130	264
3130	+inf	8

Table 4.6: Rise and Fall CTS latency.

Table 4.6 shows the CTS latency report of the design. For proper operation of the design the clock signal must reach all clock sinks at the same time. Clock latency is the delay of the clock signal from its source pin to its destination pin. Clock latency would ideally be the same for all clock destination pins. If a big difference occurs between the latencies the design might not function properly. The difference in the latencies is called clock skew.

4.2 Physical results

In this section we will review the physical results of implementing the processor.

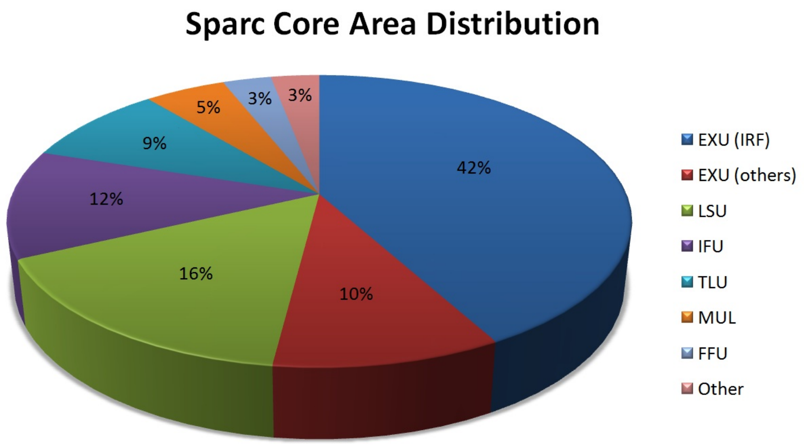


Figure 4.5: Area percentage distribution.

The design consists of about 285 K cells. Figure 4.5 shows area percentage distribution for each of the design main blocks, from this figure we can see that the EXU, LSU and the IFU are the three major design blocks.

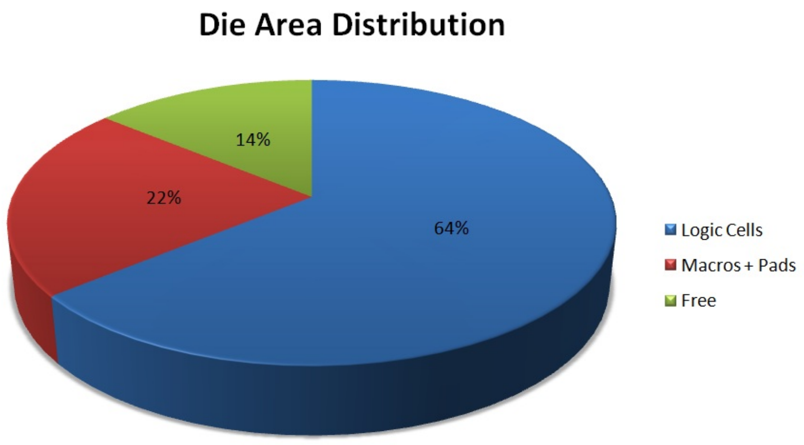


Figure 4.6: Die area distribution.

Figure 4.6 shows the die area distribution of the design. This figure shows that the utilization of the design in the specified die area is 86%, the remaining 14% percent is left to reduce the routing congestion. Because if the design was 100% utilized, the cells would be placed side to side with each other, and will make reaching the cell pins without violating DRCs a very hard task . After the design is routed the remaining free design space can be filled with spare cells. Spare cells are replacement cells that are only connected to the design if the designer wants to make a last minute small change in the design after the design has been shipped for fabrication but still the metal mask has not been fabricated yet.

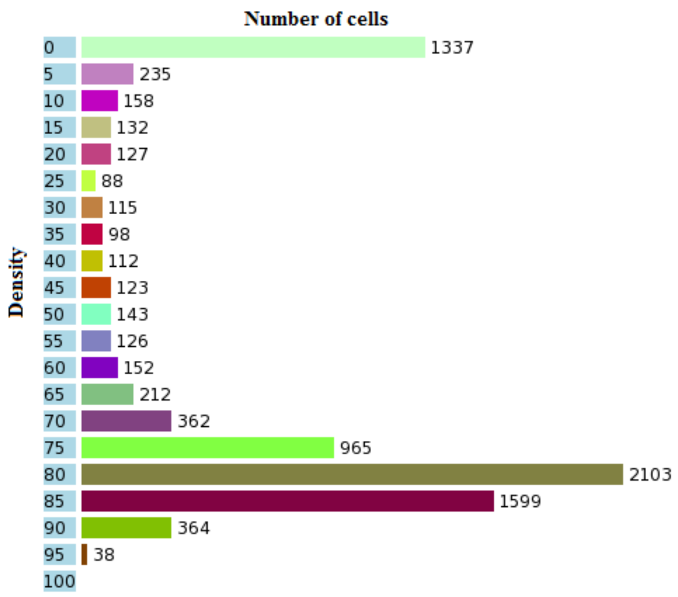
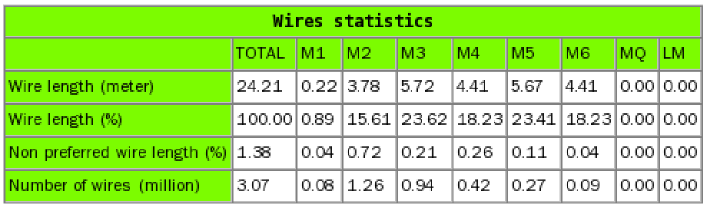


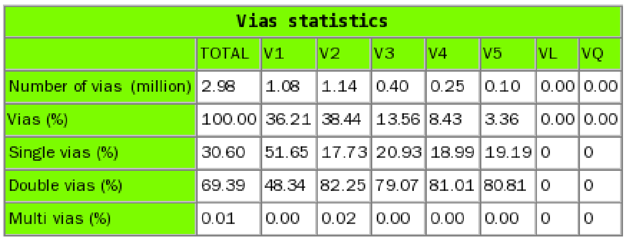
Figure 4.7: Cell density histogram.

Figure 4.7 shows placement cell density histogram. Cell density histogram shows the utilization distribution of standard cells in every placement bins. The design area is divided into small placement squares called placement bins. For good distribution of the standard cell over the die area the designer tries to use all available design bins. As shown from the above graph, most bins are 75% to 85% utilized which indicates that the placer has done a good job. Note that there are some bins that has 0% utilization, those are mostly bins that lie under the memory blocks, as these areas cannot be occupied by standard cells as they already contain the memory blocks.



Wires statistics

	TOTAL	M1	M2	M3	M4	M5	M6	MQ	LM
Wire length (meter)	24.21	0.22	3.78	5.72	4.41	5.67	4.41	0.00	0.00
Wire length (%)	100.00	0.89	15.61	23.62	18.23	23.41	18.23	0.00	0.00
Non preferred wire length (%)	1.38	0.04	0.72	0.21	0.26	0.11	0.04	0.00	0.00
Number of wires (million)	3.07	0.08	1.26	0.94	0.42	0.27	0.09	0.00	0.00

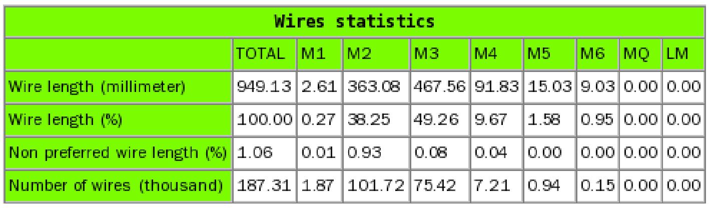


Vias statistics

	TOTAL	V1	V2	V3	V4	V5	VL	VQ
Number of vias (million)	2.98	1.08	1.14	0.40	0.25	0.10	0.00	0.00
Vias (%)	100.00	36.21	38.44	13.56	8.43	3.36	0.00	0.00
Single vias (%)	30.60	51.65	17.73	20.93	18.99	19.19	0	0
Double vias (%)	69.39	48.34	82.25	79.07	81.01	80.81	0	0
Multi vias (%)	0.01	0.00	0.02	0.00	0.00	0.00	0	0

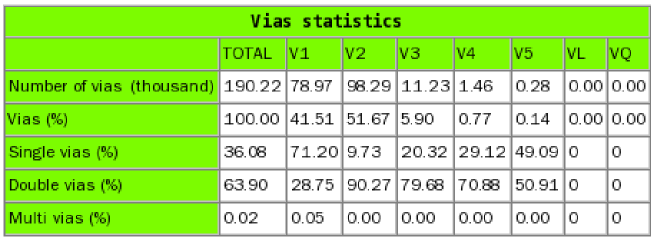
Table 4.7: Design routing statistics.

Table 4.7 shows design routing statistics. From this table you can notice that the design routing was done mostly using metals from layer 2 to 6, this is expected as M1 is mostly consumed in power rails and internal connections of the standard cells. You could also notice that the design has 70% double VIAs, which is a good indicator that the design is DFM (Design For Manufacturability) friendly, which means that the metal layers bonds are strengthened using double VIAs to withstand fabrication issues. This means that the design yield will increase. Yield is the percent of working chips to total number of chips on a single wafer.



Wires statistics

	TOTAL	M1	M2	M3	M4	M5	M6	MQ	LM
Wire length (millimeter)	949.13	2.61	363.08	467.56	91.83	15.03	9.03	0.00	0.00
Wire length (%)	100.00	0.27	38.25	49.26	9.67	1.58	0.95	0.00	0.00
Non preferred wire length (%)	1.06	0.01	0.93	0.08	0.04	0.00	0.00	0.00	0.00
Number of wires (thousand)	187.31	1.87	101.72	75.42	7.21	0.94	0.15	0.00	0.00

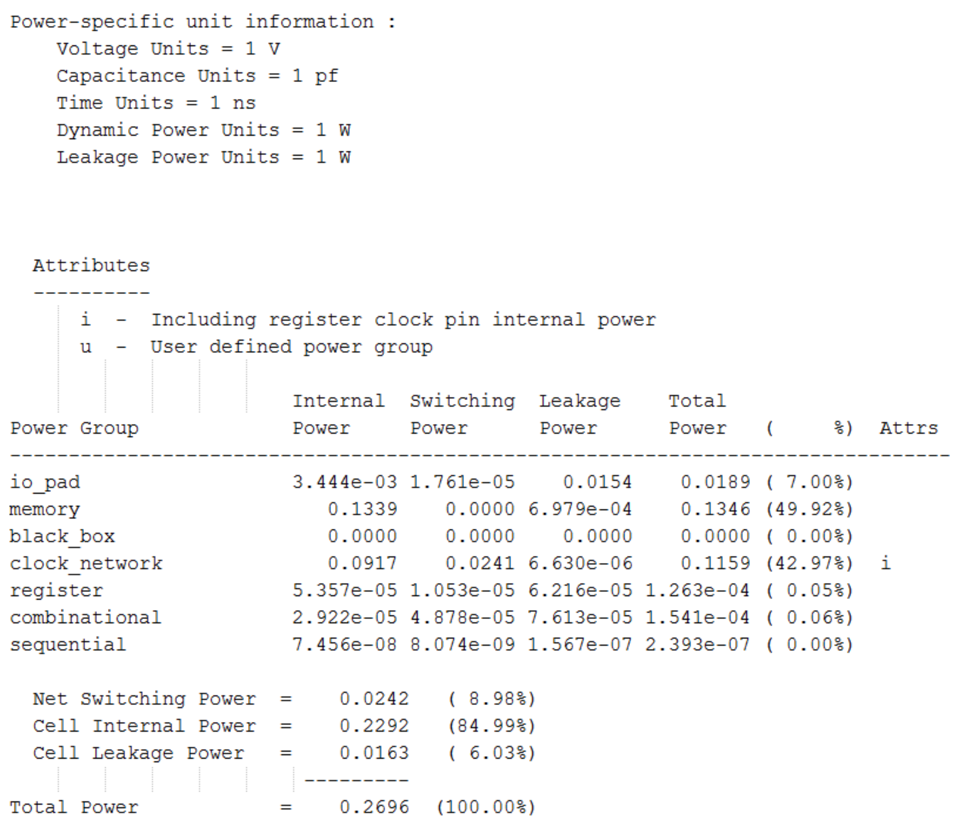


Vias statistics

	TOTAL	V1	V2	V3	V4	V5	VL	VQ
Number of vias (thousand)	190.22	78.97	98.29	11.23	1.46	0.28	0.00	0.00
Vias (%)	100.00	41.51	51.67	5.90	0.77	0.14	0.00	0.00
Single vias (%)	36.08	71.20	9.73	20.32	29.12	49.09	0	0
Double vias (%)	63.90	28.75	90.27	79.68	70.88	50.91	0	0
Multi vias (%)	0.02	0.05	0.00	0.00	0.00	0.00	0	0

Table 4.8: Clock routing statistics.

Table 4.8 shows design clock routing statistics. You can notice that most of the routing is done on metal 2 and 3, this is expected as the clock network was routed separately in the beginning where all metal layers were available for routing. The router chose to route on lower metal layers to reduce the jogs on several metal layers and by that reduces the VIA count which will reduce the load on the wires and therefore reduces the resistance and increase the clock signal speed.



```
Power-specific unit information :
    Voltage Units = 1 V
    Capacitance Units = 1 pf
    Time Units = 1 ns
    Dynamic Power Units = 1 W
    Leakage Power Units = 1 W


  Attributes
  ----------
      i  -  Including register clock pin internal power
      u  -  User defined power group

                        Internal  Switching  Leakage    Total
Power Group             Power     Power      Power      Power    (     %)  Attrs
--------------------------------------------------------------------------------
io_pad                  3.444e-03 1.761e-05    0.0154    0.0189 ( 7.00%)
memory                     0.1339    0.0000 6.979e-04    0.1346 (49.92%)
black_box                  0.0000    0.0000    0.0000    0.0000 ( 0.00%)
clock_network              0.0917    0.0241 6.630e-06    0.1159 (42.97%)  i
register                5.357e-05 1.053e-05 6.216e-05 1.263e-04 ( 0.05%)
combinational           2.922e-05 4.878e-05 7.613e-05 1.541e-04 ( 0.06%)
sequential              7.456e-08 8.074e-09 1.567e-07 2.393e-07 ( 0.00%)

  Net Switching Power  =    0.0242   ( 8.98%)
  Cell Internal Power  =    0.2292   (84.99%)
  Cell Leakage Power   =    0.0163   ( 6.03%)
                          ---------
Total Power            =    0.2696  (100.00%)
```

Table 4.9: Power analysis.

Tables 4.9 shows the power analysis of the full design. This is based on switching activity with average static probability of 50% and average toggle rate of 10%. From the numbers you can see that most of the dissipation occurs in the clock network, memory blocks and I/O pads. Leakage power dissipation is leading in the I/O pads, Internal power (dynamic power dissipated within the memory boundary) and dynamic power dissipation is leading in the clock network.

4.3 Design comparison

In this section I will compare my design to the previously listed FPGA designs [6], the UltraSparc T1 (Niagara) release [14] and the FPGA design made by M. Merzban [15]. Table 4.10 shows this comparison.

	IMP. SPARC	UltraSPARC T1 (Niagara) core [17]	FPGA Design 1 [1]	FPGA Design 2 [1]	FPGA Design 3 (M. Merzban) [18]
Configuration	2 Threads / No SPU / Reduced TLB	4 Threads / SPU / Full TLB	1 Thread / No SPU / Reduced TLB	4 Threads / No SPU / Reduced TLB	1 Thread / No FFU / No SPU / Reduce SB / Reduced IRF / Reduce MX / No PC / Reduced MP / Reduced CCX
Technology	IBM 130nm CMOS	Texas Instruments 90nm CMOS	FPGA: Xilinx XC4VFX6	FPGA: Xilinx XC5VLX110T	FPGA: Xilinx XC5VLX110T
Processor Size	9 mm^2	11 mm^2	30K Virtex-5 LUTs	51K Virtex-5 LUTs	69K Virtex-5 LUTs (2 cores)
Speed	100 MHZ	1 GHZ	50 MHZ	10 MHZ	75 MHz

SB = Store Buffer MX = Multiplexer PC = Parity Check MP = Multiplier

Table 4.10: Design comparison table

4.3.1 Processor size

It is hard to compare the design size of the implemented design to the FPGA designs, as it is hard to estimate the size of the LUTs of the FPGAs. So the implemented design is compared to the Ultra-Sparc T1 (Niagara) release. Taking into consideration the difference in technology between my design and the UltraSparc T1 (Niagara) design, scaling of my design size is needed in order to have a reasonable comparison.

Figure 4.8 shows the layout of the UltraSPARC T1 processor core. From the processor information that is available, the processor core size is 11 mm^2 [6]. My implemented design is 9 mm2 on a 130 nm technology, direct scaling this to 90 nm technology yields to , ((90 x 90) / (130 x130)) x 9 = 4.31 mm^2, this shows that my design is smaller in size. However, my implementation has undergone some size reduction which is not the case in the UltraSPARC T1 core. From table 3.1 you can see that there was a 42.27% design size reduction, adding this to my design size we get 4.31 /(1 - 0.4227) = 7.46 mm^2. This shows that my design is competitive with the area of UltraSparc T1 (Niagara) release. It must be noted that the smaller design size was gained by relaxing the timing constraints this made it easier for

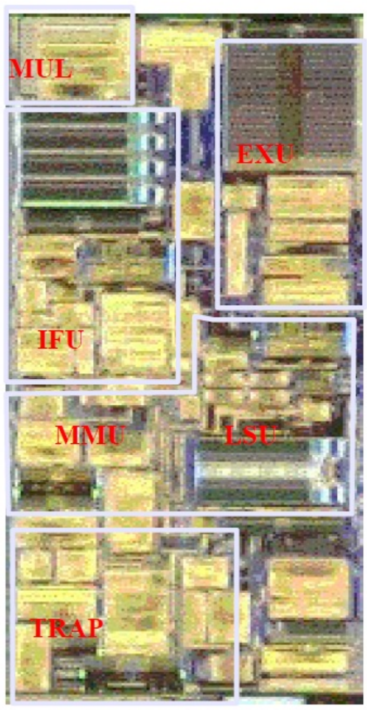


Figure 4.8: UltraSPARC T1 processor core [8]

the tool placer and router to achieve this area target. As mentioned before that this area restriction is imposed by the fabrication facility in order to fabricate the design under educational terms. If this area constraint is relaxed the designer may be able to achieve higher design performance.

4.3.2 Processor speed

Comparing the implemented design to the FPGA designs we can see that the implemented design is faster, which is expected as ASIC is used compared to FPGA. The UltraSPARC T1 processor runs on 1 GHz clock, compared to the 100 MHz of the implemented design. This huge difference in the performance is due to the following reasons:

1. The UltraSPARC T1 uses 90 nm technology which is faster than the 130 nm I am using.

2. The maximum frequency of the artisan standard cell library used with this design is limited to 1GHz.

3. The UltraSPARC T1 has custom design for the memories, and the input and output drivers for the chip I/Os [16], which means that all the memory blocks and I/Os are optimized for high performance, compared to regular standard cells and memory generators that was used in the implemented design.

4. Size Limitation by the factory on my design to be 4x4 mm, which is not the case in the UltraSparc T1.

5. The use of PLLs on chip to generate the clock [4], will boost the clock frequency of UltraSPARC T1, compared to external clock

source that is used.

6. 9 Layers of metal interconnects are used by the UltraSPARC T1 compared to the 6 layers used in my design. Which will allow more routing resources for the UltraSPARC T1 which will reduce congestion.

Chapter 5

Conclusion

In this thesis a complete ASIC design of the reduced OpenSPARC T1 processor core was implemented. This multi-threaded core conforms to the SPARC V9 ISA and is characterized by the following specifications:

1. The design was implemented using IBM 130 nm CMOS technology.
2. It has the area of 9 mm^2.
3. Consists of two threads.
4. Operates on 100 MHz frequency.
5. Consumes 0.2696 Watt of power.

6. Instruction cache size is 16 Kbytes.

7. Data cache size is 8 Kbytes.

In order to design this process core the following 9 main steps was executed:

1. RTL preparation: This included steps like core area reduction, memory mapping and PADs insertion.

2. RTL functional verification: Verifyng the core functionality after the modifications that were done in the previous step.

3. Synthesis: Changing the design from behavioral description to physical implementation.

4. Gate level verification: Verifyng that the conversion in the synthesis step did not introduce errors.

5. Floorplaning: Planning the layout of the design. This includes steps like Macro placement and power grid planning.

6. Placement: Placing the standard cells of the design in appropriate locations. This step also includes design pre-optimization.

7. CTS: Builds the clock network of the design. This step also contains clock routing.

8. Routing: Connects the design pins together. This stop also includes post layout optimization.

9. Post layout verification: Verify that the previous physical implementation and optimization steps did not introduce errors.

It is not necessary in order to implement any design to follow the same exact steps that was discussed in this thesis documentation. The designer has the total freedom to change, add or remove any step of the flow depending on his design nature and specs and the way he sees it is appropriate to implement his design. The 9 main steps of the flow discussed in this thesis documentation are the most commonly used to implement any design.

5.1 Future work

There are several ideas of improvements that can be made on the design to enhance its performance and increase its speed. Below we will discuss some of these ideas.

5.1.1 Tighten the design constraints

The first and simplest way to improve the design performance is to tighten the design constraints. This approach is not expected to make huge improvements on the design. Because as in Figure 4.1 there is only small room for improvement.

5.1.2 Custom implement the IRF memory module

The IRF module in the SPARC core is considered the biggest module in the design, it accounts for about 50% of the design area as shown in Figure 5.1.

From the most critical timing path report in Figure 4.3 we can see that the path goes through the IRF block, and as mentioned in chapter 4 this block was not implemented using memory blocks. So if this block was custom implemented to reduce its area and increase its speed then the performance of the design will increase.

5.1.3 Optimize clock network

From table 4.5 there exists some outlier clock networks that are up to 23 level deep. If these outliers could be optimized like for example

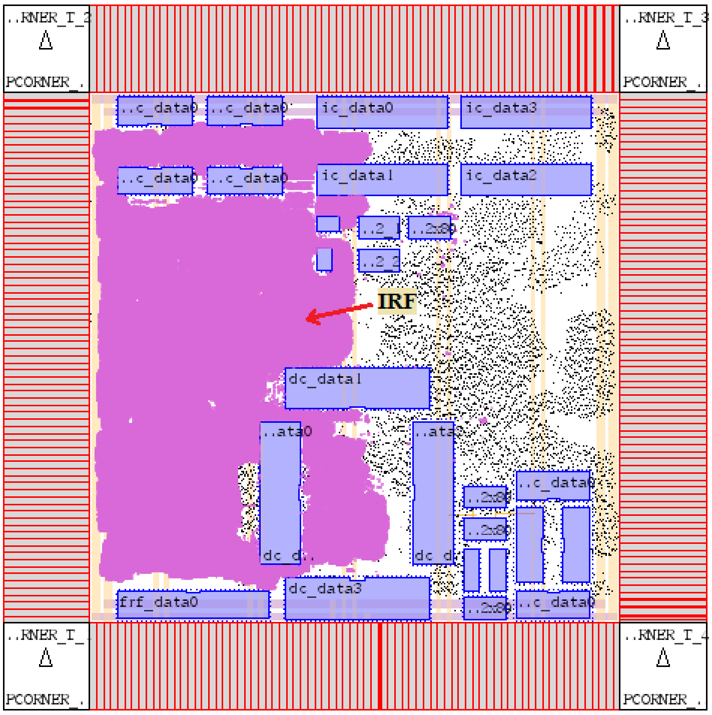


Figure 5.1: IRF block area highlighted.

the clock elements moved closer to the clock source, then the number of clock levels will decrease leading to a reduction in the clock latency which will increase the performance of the clock network and accordingly the performance of the design will increase.

5.1.4 Use smaller technology node

At the time this thesis started the only complete design kit (Includes Memory Generators) that was available was the IBM 130 nm technology node. Using a smaller technology node will for sure reduce the design area and increase the design speed, this is because the cells, pads and interconnect delays will become smaller. Using a smaller technology node has its own disadvantages, like for example crosstalk effect will increase and static power dissipation will also increase, new techniques to overcome these problems can be investigated.

Appendix A

Digital design ASIC Flow

Every design process has to go through a set of design steps to reach the required outcome. This set of steps may differ from design to design depending on the environment and the architecture of that design. But there are always some main design steps that the design cannot be accomplished until they are done. Also these steps depend on each other so they must be done in a certain order. In this chapter we will introduce the basics of the digital design ASIC flow (steps).

In this chapter we will introduce the main design steps in addition to most of the additional design steps that came to my attention while researching for this thesis work. The design flow described below may differ from one design to another or even from one designer to another depending on what each designer thinks is the best for that particular design. Figure A.1 shows the main design flow for the digital ASIC design.

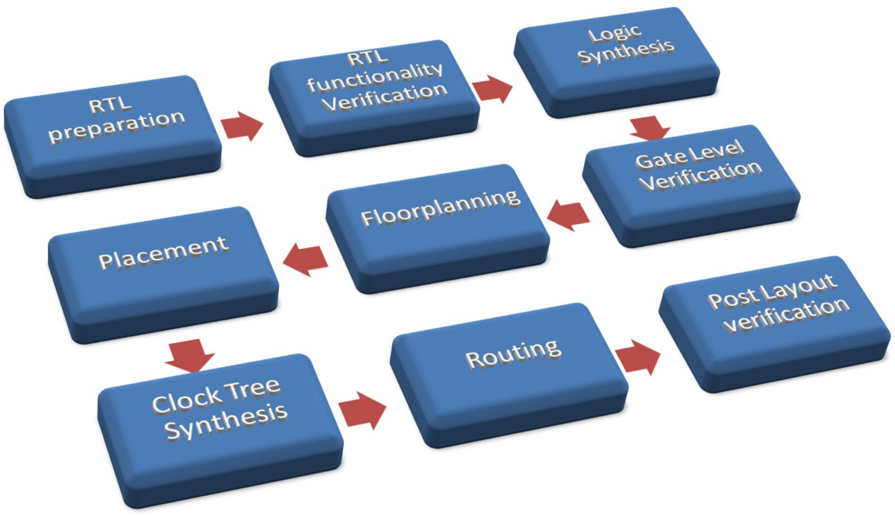


Figure A.1: Digital ASIC Design Flow

A.1 RTL preparation

RTL preparation stage is where we get familiar to the design in its RTL form. To make it easier to handle very large designs, usually the design is divided into several blocks, and each block is divided into several sub blocks depending on the design size and complexity, where several groups of engineers work on each block separately and then combine these blocks in one top level design. In case of very large designs, a separate group of engineers work only on this top level (usually called chip assembly).

Verilog / VHDL formats are mainly used to represent the design files at this stage. Those files define the functionality intended by the design. If the functionality has already been verified for each of those files, no changes are made to them, except in the existence of memory structures. There are different methods to implement memory structures; the simplest is the use of registers but the commonly used are memory blocks. Using registers is not efficient for many reasons especially for large memory sizes. Some of these reasons are:

1. Area: Using registers to represent large memory blocks is not efficient as a register contains a large number of transistors compared to a memory cell of a memory block.

2. Power: A register consumes a lot of power compared to a memory cell of a memory block.

3. P&R: Placement and routing of millions of memory registers are

very difficult compared to the placement and routing of a single memory block.

Memory blocks are considered as black boxes in the design. Each ASIC technology vendor has his own way of supplying the information needed by the designers to implement those memory blocks. In most cases it comes in the form of memory generators (A computer program), where the designer enters the specification of the memory block to the generator, and the generator generates the required design files. One of these files contains the Verilog / VHDL implementation of the memory block. The designer then has to instantiate the memory block as an instance in his design, and connects its interface pins to the rest of his design.

There are limitations on the sizes of memory blocks that the generator can generate; this reflects the restriction of the memory sizes the ASIC vendor can supply. There are also different types of memories that the designer can choose from to suite his design needs, examples are single port SRAMs, double port register files ...etc. Usually the technology vendor supplies a specification document with each memory generator to describe in details the functionality, architecture and size limitation of memory blocks that can be generated using this generator. The designer then uses this document as a guide to generate the required memory blocks.

After the designer has replaced all the memory structures in his design with memory blocks the designer has to connect the interface pins of his design to pads. This step is only done on the top level design, if the designer was working on a block that is instantiated in the top level then this step is not required as the top design will be connected to pads by the top level design engineer.

Pads are gateways that will interface the chip level external pins to the design internal pins. There are two main types of pads, power pads and general purpose input/output pads. As the name indicates power pads are used to supply the internal ASIC core with power from external sources and the general purpose pads are used to connect internal input /output signal pins to external signal sources. Usually the ASIC technology vendor provides the designer a document that describes the functionality and specs of each pad available in the design kit.

Pads can be connected to the design either automatically or manually. The automatic connection is done by the synthesis tool where the designer specifies to the tool the available pads and the tool automatically chooses the best pad and makes the connection to the pins automatically. The manual connection is done by the designer where he instantiates the top level design in a new top level module and then instantiate the pads that will be used from the library and then makes the connection between the design pins and pad pins manually.

After completing the pads insertion in the design the design is now ready for the next implementation step.

A.2 RTL functionality Verification

After modifying the design (If needed) in the RTL preparation stage, we need to make sure that the design functionality has not been altered. There are two main methods to do that, either to use simulation or formal verification or both. In simulation the design is tested by applying some test patterns to the inputs of the design, and then the outputs of the design are compared to a golden set of expected outputs. Usually the design runs through a battery of tests, where each test is designed to test a certain part or functionality of the design by applying the appropriate input patterns to the inputs of the design and then comparing the outputs with the expected values.

Formal verification is mainly used to compare a design in different stages, where we have a golden design file which we are sure it is functionally correct and we have a modified version of that design and we need to make sure that it is functionally equivalent. During formal verification each of the designs is reduced to a certain set of equations (properties), and then these equations are compared together to make sure they are equivalent.

The main difference between simulation based verification and formal verification is that the simulation requires input vectors and formal verification does not. Another difference is that formal verification is more complete, in the sense that it does not miss any point in the input space, a problem from which simulation based verification suffers. However, this strength of formal verification sometimes leads to the misconception that once a design is verified formally, the design is 100% free of bugs.

Simulating a vector can be viewed as verifying a point in the input space. With this view, simulation based verification can be seen as verification through input space sampling. Unless all points are sampled, there exists a possibility that an error escapes verification. As opposed to working at the point level, formal verification works at the property level. Given a property, formal verification exhaustively searches all possible input and state conditions for failures. If viewed from the perspective of output, simulation-based verification checks one output point at a time; formal verification checks a group of output points at a time (a group of output points make up a property). Figure A.2 illustrates this comparative view of simulation-based verification and formal verification.

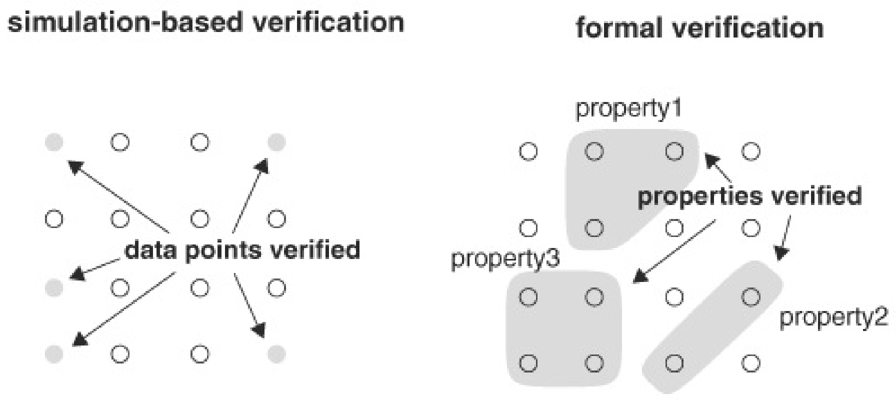


Figure A.2: Simulation VS Formal Verification [17]

With this perspective, the formal verification methodology differs from the simulation-based methodology by verifying groups of points in the design space instead of points. Therefore, to verify completely that a design meets its specifications using formal methods, it must be further proved that the set of properties formally verified collectively constitutes the specifications.

Usually at this stage of the flow there is no reference design that can be used by the formal verification tool to compare with, therefore it is more often that simulation is used to verify the design. It is also common that the design is verified with the same simulation environment used by the designers who wrote the RTL code to verify it, making only minor changes to it if needed.

For the designer to be able to verify his complete design in case of the existence of memory blocks (mentioned in the previous section), he must use the behavioral description of these blocks written in RTL form (verilog or VHDL). This description is supplied to the designer in the form of .v or .vhd files by the memory generator that was used to generate those blocks.

A.3 Logic Synthesis

Logic synthesis is the process of converting the design from its behavioral form (RTL code) to its physical form (Gate level netlist) [18]. In this process the synthesis tool maps each functionality from the design to its equivalent physical structure using the technology files supplied by the ASIC vendor. These physical structures are composed of one or more standard cells from the ASIC library. Standard cells are the basic building blocks that are supplied by the ASIC vendor that the designer uses to implement any functionality in his design. The mapping process that is done by the synthesis tool is an automatic one; the designer only guides the synthesis tool to do the mapping as desired. This guidance comes in the form of constraints that the designer specifies for the tool to obey, for example the designer can set some standard cells as don't use by the tool.

Synthesizing a design is an iterative process and begins with defining timing constraints for each block of the design. These timing constraints define the relationship of each signal with respect to the clock input for a particular block. In addition to the constraints, a file defining the synthesis environment is also needed. The environment file specifies the technology cell libraries and other relevant information that synthesis tool uses during synthesis.

Figure A.3 describes the synthesis process.

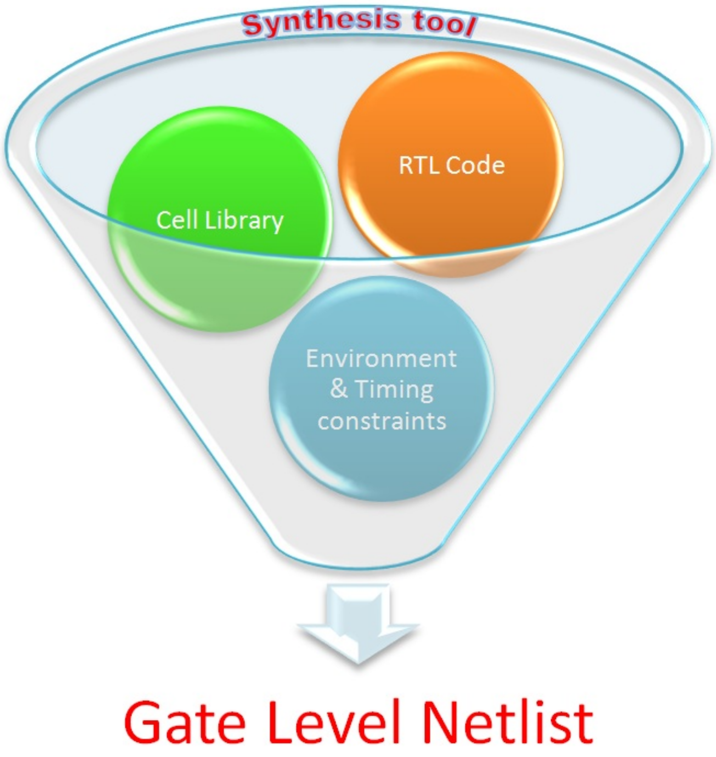


Figure A.3: Synthesis Stage

Usually the synthesis tool does a first iteration of initial fast mapping of the full design then it 'does other several iterations trying to optimize the outcome targeting different objectives like area, timing, power ..etc. These objectives are defined in the form of tool commands and constraints then the tool is instructed using its commands to optimize the output for these different objectives using these constraints.

Memory blocks (mentioned in the RTL preparation stage) are treated as black boxes and are not synthesized; only their timing information is taken into consideration during timing optimization in the synthesis process. This timing information is supplied to the synthesis tool in separate files (.lib or .db) which are supplied by the memory generator.

The outcome from this synthesis process is a gate level netlist of the full design and a constraint file that defines the timing constraints of the full design that is needed for the design to function correctly. The synthesis process is very memory and processing power consuming; therefore the design is usually synthesized each block separately and then is collected together in one top level gate netlist.

A.4 Gate Level functionality Verification

After synthesizing the design we must make sure that its functionality has not been altered during the synthesis process, this is done in this stage of the flow. The gate level netlist generated by the synthesis process is compared to the golden design in its RTL form using the formal verification tool as discussed in section 3.2. Simulation is not commonly used at this stage as the design structure has undergone a lot of changes in the synthesis process. For example a lot of internal signals and connections have been modified during the synthesis process which will cause the simulation test bench fail if it was monitoring any of those signals.

Although the synthesis process is automated and the designer do not interfere much in it, the bad coding style used by the designer could cause the tool to misinterpret the RTL code, causing mismatch between what the designer wants and what the synthesis tool understood. For example synthesis tools infer combinational or latching logic from an always block with a sensitivity list that does not contain the Verilog keywords posedge or negedge. For more examples please refer to [19].

A.5 Floorplaning

This stage marks the start of the Place and Route flow [20], where the design will physically be implemented in the form of a layout that can be fabricated. In this stage we start by collecting and loading all the design files that we generated from the previous stages or technology related files. Below is a list with brief description of each file needed at this stage:

1. LEF (Library Exchange Eormat) files: LEF files are technology specific files contain the definition of the technology metal layers, DRCs governing those metal layers and metal abstract of each standard cell in the library. Also if there are memory blocks in the design, each of these memories will have its own LEF file which can be obtained from the memory generator as discussed earlier.

2. LIB (Synopsys Liberty Format) files: LIB files, which are also technology specific files, contain timing information of each of the standard cells in the library at different corners. Also similar to LEF files, each memory block has its own LIB files at different corners.

3. Verilog/VHDL netlist: This is the output file from the synthesis stage. It defines all design elements and the connection between them.

4. SDC (Synopsys Design Constraints) files: These files are also outputs from the synthesis stage. SDC files define the timing constrains the design must obey to function correctly.

5. Interconnect specification file: This file is a technology specific file, but may have different formats depending on the P&R

(Place and Route) tool used. Also some P&R tool vendors supply conversion scripts to convert a well known interconnect specification file to its own format. This file contains specifications and parameters for each metal layer so the tool can use this information and calculate the resistance and capacitance of each interconnect.

6. DEF (Design Exchange Format) files (optional): DEF files can define any physical structures. It can be used for example to define a floorplan that might have been made by a third party tool, or can be used to specify a partition that went through the P&R flow separately. It is optional because it is not needed if the designer will start the design from scratch.

Each of the above files is usually loaded in the above sequence, but the designer should make sure before loading those files that he has already defined the modes and corners of the design. A design mode specifies the mode of operation of the design, for example we might have a normal operation mode (usually called Functional) and a test mode (usually called Bist). When the mode is enabled the tool understands that the designer wants to do all his timing analysis when the design is functioning in a certain way, this is done by enabling certain switches in the design by setting ones or zeros on some design inputs, this is done in the SDC file that is loaded. Inside the SDC file there is a command called set_case_analysis that does the job of applying ones or zeros to certain design pins/ports. That is why it is necessary while loading the SDC file we must specify which mode is this constraint for.

Design corners specify the operating conditions (ex. Temperature, Voltage ...etc) that the design will operate on. Usually there are two main corners (Best and Worst), but as the technology node drops in size this number multiplies. For each corner we have separate LIB files that specifies the timing calculations in that corner, that is why the designer should make sure that while loading the LIB files he specifies which corner is those files for. Not all P&R tools can handle all corners and modes in one working session, so in this case the designer should test his timing values in each combination of modes and corners to make sure that his design is timing clean in all design cases.

At this stage the designer can test his design timing constraints using an approach called Zero RC Timing, this is done to make sure that the design constraints are valid and logical. In this approach all the interconnect delays are set to zero, and timing is calculated using only cell delays. For the constraints to be valid the design must pass this test without any violations because if there is any violation it will only get worse when interconnect delays are added later in the flow. Some optimization may be needed at this stage to solve some small timing violations, for example if a cell driving strength is smaller than required the optimizer can replace those cells with ones which have a higher driving strength. Now the design is ready for floorplaning.

Floorplanning is the initial planning of the chip layout, in this stage the designer specifies roughly where each part of his design will be located on the chip die. The designer will also decide how he will implement the design depending on the design size and the strategy that he will follow, for example if the design is small he could simply flatten all the design hierarchies and let the P&R tool automatically places all the design elements in one area and then connect them by routing. If the design is large he could choose to divide it into smaller partitions and he could either work on those partition separately as separate designs (blocks) and then import them in the top level or he could work on them all at once in the same context of the top level, in this case the tool will deal with each partition as separate design but will work on them at the same time.

Whichever flow the designer chooses to build his design he must first start by specifying the dimensions (area) of the chip. The designer can directly specify the X and Y dimensions, or he can specify the dimensions as a utilization factor and an aspect ratio. In the latter case the tool calculates the estimate area by summing up the area of each element in the design and then expands this area until these elements occupy the utilization factor the designer specified. The aspect ratio defines the ratio of the length to the width of the design. The area of the chip could be either core driven or pad driven. In case of core driven design, the core cells controls the size of the chip, and this is usually the case for big designs. For small designs, the pads inner circumference is what controls the chip size.

Figure A.4 shows an example of a design floorplan.

Pads are circuits that translate the signal levels used in the ASIC core to the signal levels used outside the ASIC. Additionally, the pads circuits clamp signals to the power and ground rails to limit the voltage at the external connection to the ASIC pads. This clamping reduces signal overshoot and prevents damage from Electrostatic Discharge (ESD). In most cases the technology provider provides different type of pads to satisfy each of the design requirement. In case of core

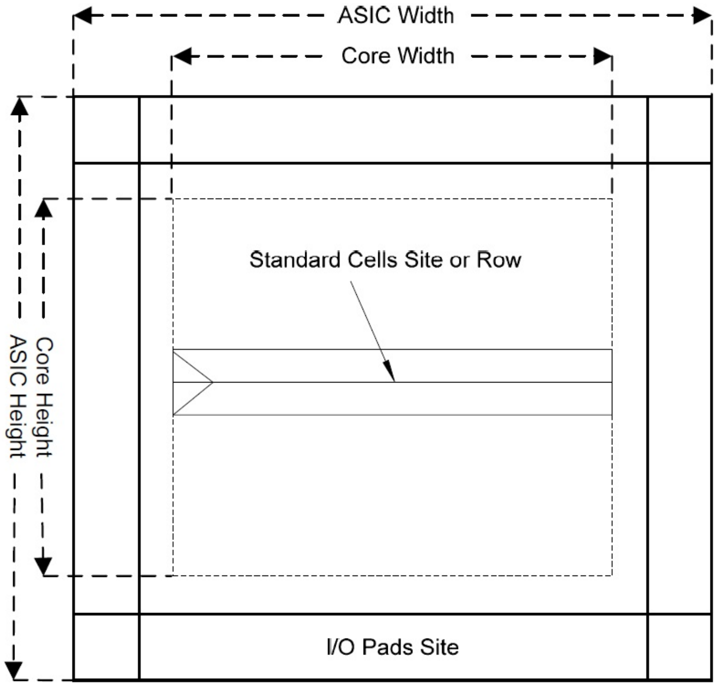


Figure A.4: Floorplan of the design [21]

driven designs, pads are thick and short so that it occupies more lateral dimension. In case of pad driven designs, pads are thin and tall to occupy more perpendicular dimension to reduce the design size as much as possible. The designer places his choice of pads in the area surrounding the die, pad placement is usually done automatically by the P&R tool, but it must be constrained to guarantee the ordering and the location of each pad. For example any pad can be placed on any of the four sides of the chip top, bottom, right or left. The designer must intelligently choose the correct location in order to reduce the connection length between the pad and the design components that will be placed latter in the design. Some P&R tools does an initial pad location estimation derived by the design information it has, but this is only a rough estimation and the designer who has a deeper understanding of the design should adjust this initial estimation to a more intelligent one. Also the designer must take care of ordering the pads correctly to ensure that related pads are placed close to each other, for example if the design has a bus on its interface, the designer must make sure that he orders his bus pads correctly to ensure good distribution and symmetry of the components that will be placed latter on in the design flow.

Figure A.5 shows an example for pad driven and core driven designs.

After defining the chip area the designer then starts defining partitions or regions in the core area if needed. This definition is done manually using the P&R tool GUI or commands. Partition defines a group of design elements grouped together. As mentioned before, partitions are made to enable separate parts of the design to be imple-

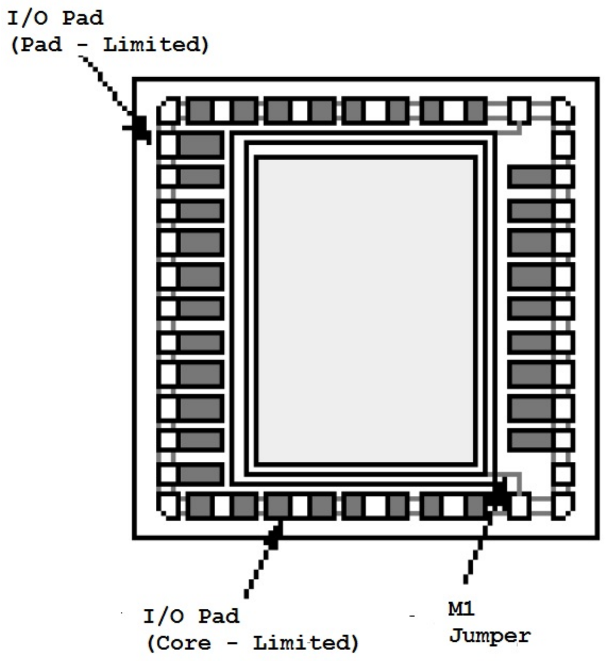


Figure A.5: Pad driven VS Core driven [22]

mented separately as small sub designs. Regions define certain areas in the chip where you can place certain design elements close together, unlike partitions regions are only visible to the placer engine of the P&R tool. The region boundaries could be either hard or soft, in case of soft boundaries the P&R tool placer engine can either place region elements outside of the region boundaries or place top level elements inside the region boundaries, and this could be controlled by certain switches in the tool. In case of hard boundaries the tool only places the region elements inside the region boundaries and keeps everything else out. This hard boundaries approach is mostly used in case the design uses multiple voltage supplies. In this case, these regions act as voltage islands where each of those islands can be powered by separate voltage supplies. If an element of an island is placed outside the island boundaries it will not function properly or might not function at all.

After creating the design regions and partitions the designer then creates the design rows, rows are hypothetical locations that specify where the design standard cells will be located, and they are used to guide the design placer to only place cells on these rows. Rows are separated by equal spaces depending on the cell height. In general each technology library has a fixed standard cell height. In some cases the technology library might also contain cells of other heights but these heights are usually multiples of the normal height, this is done to enable these cells to be placed on the same rows as the normal cells expect that they will occupy more than one row.

The next step is to build the power grid. We start by defining the power rings. Power rings are metal wires that surrounds the chip and are used to supply power, the metal wires used to build those rings are usually the thick metals of the technology (ex: Metal 7 and 8, in a 8 layer technology), this is done to reduce the power dissipation, guarantee good connectivity and keeps the thin lower metal layers for signal routing to reduce the routing area needed. Two metal layers are used because usually each metal layer has one preferred direction that most of its routes are routed in that direction. In critical cases where DRC violation cannot be resolved, the tool might try to use the non preferred direction to resolve the violation. In simplest case we have at least two power rings one for power (VDD) and the other for ground (GND), for multi-voltage designs we might have more. Power rings are then connected directly to power pads to supply the rings with power.

After connecting the power rings, the designer then creates power stripes, power stripes are vertical metal wires that are placed over the core area and are connected to the power rings. Power stripes are used to distribute the power from the rings to the rest of the design area. Each of the power supplies has its own stripes; also power stripes are created on thick metal layers similar to the power rings. After creating the stripes the designer creates power rails, power rails are horizontal metal wires that are used to supply power to the standard cells. The connection between the power rails and the standard cells is done through abutment, where standard cells has its power pins at the top and bottom of the cell, and power rails are placed at the top and bottom of each cell row, so when those cells are placed on the rows they automatically get connected to the power grid. Power rails are connected to both the ring and the stripes to ensure even power distribution along the rail length.

The power grid gets its power from power pads, the connection of power pads and the power ring is done manually or automatically depending on the implementation tool. Signal pads need also to be supplied with power; this is done through abutment of the pads to each other forming their own power ring, where each pad has its supply pins in the same location across the pad width, which guarantees that when those pads are put together they form a ring. If empty spaces exists between adjacent pads in case of core driven designs, this space is filled by special type of pads called pad fillers, the main purpose of those fillers is to complete the pads ring structure. Similarly at the four corners of the chip there exists a special type of cells called corner cells that does the same functionality of connecting the ring structure.

Figure A.6 shows an example of a floorplan power grid.

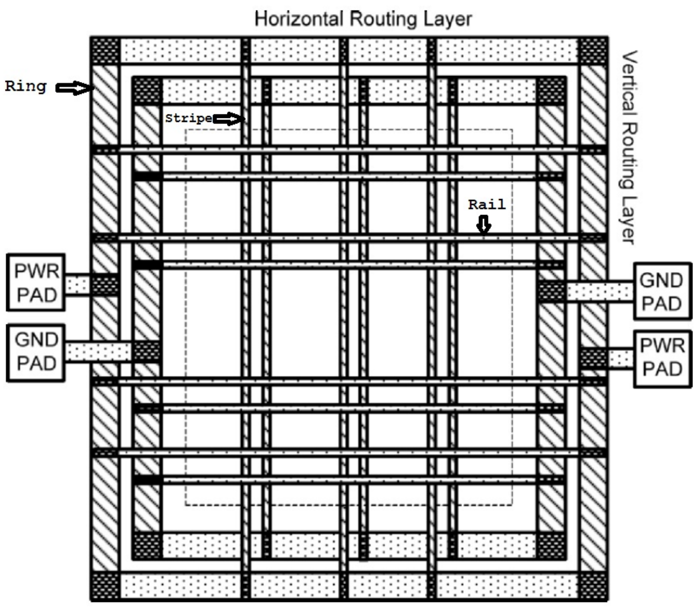


Figure A.6: Power grid of the design [21]

The floorplan is ready now to receive the design components; the first components that should be placed in the floorplan are macro blocks. Macro blocks could be either memory blocks or any physical block that is not a normal standard cell. Macro blocks could be either placed automatically or manually. If macro blocks are placed automatically, the tool tries to roughly estimate the location that will reduce the connection length and reduce the congestion of the design. If macro blocks are placed manually, the designer should try to achieve the same goals. The best practice in most cases is to let the tool roughly estimates the initial locations of the blocks and then the designer with his deeper view of the design adjusts these locations. The designer must make sure that the placed blocks does not block its pins and its orientation are correct to guarantee that routing could reach the block pins in the preferred metal direction. Also the designer must make sure not to leave narrow channels between blocks, as this will cause a lot of problems if standard cells got placed in them. An example of these problems is the congestion of routing wires at the entrance point to this channels. The macro blocks are then marked as fixed location so that it is not moved later in the flow.

Figure A.7 shows an example of macro placement in a design.

If any standard cell rows existed under the location where the macro blocks are placed, these rows must be removed using the tool row cutter or in another approach the locations where macros exists are covered by hard placement blockages, this is done to ensure that standard cell are not mistakenly placed in these locations causing DRC (design rule checks) errors.

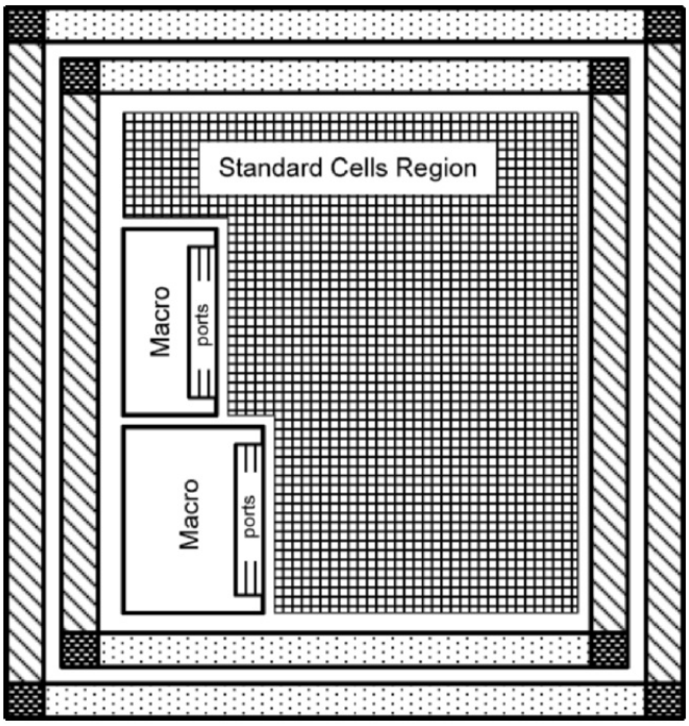


Figure A.7: Example of macro placement [21]

After placing the macro blocks they need to be separately connected to the power grid. Macro blocks can have its own power ring that is internally connected to the internal macro structure. In this case the designer must make sure that this ring is connected to his power grid. Alternatively, the macro block has large supply ports and also the designer must make sure to connect these ports correctly to his power grid. These connections could either be done automatically or manually depending on the implementation tool capabilities.

A.6 Placement

In this stage all the remaining physical cells of the design get placed on rows that were created in the floorplaning stage. But before the designer starts placing the standard cells a preparation stage is needed. This preparation step is called high fan out net synthesis (HFNS). In this step the tool tries to reduce as much as possible the high fan out nets. High fan out nets are nets that has a large number of sinks connected to one driver. This leads to huge delays at this driver output due to the large capacitive load that those sinks imply. The tool divides those sinks into small groups each derived by a newly instantiated buffer and connects the input of those buffers to the original driver. The tool can repeat this process on the newly instantiated buffers to reduce the fan out, this will build stages of buffer drivers. This process of HFNS is done before placement as it introduces a lot of buffers that needs to be placed near its sinks, and it also solves several timing issues and can eliminate critical paths not critical any more. The only exclusion to the HFNS process is the clock tree, although the clock network drives a huge number of load but this will be handled in the CTS (clock tree synthesis) stage, for the time being the clock network will be considered as ideal net, i. e., a net that has zero interconnect delays. Another example of ideal nets is the reset network; this also could be considered as a clock network and be handled in the CTS stage. Now the design is ready to be placed.

Placement is done in two main steps Global Placement and Local Placement. In global placement the tool does its calculations and roughly places the standard cells. The global placement process could be either driven by timing or congestion or both. If timing goal was only specified the tool tries to place the cells in a way that reduces the timing delays on timing constraint paths in order to reduce the overall timing violations. If congestion goal was specified the tool tries to place cells in a way that reduces the overall design routing congestion. If both goals are specified the tool tries to compromise its placement in a way that satisfies both goals as much as possible. In global placement the cells are scattered all over the core area, the global placer does not take care of any DRC violations, so it is normal that you could find cells placed over each other or cells not placed on the rows correctly.

In global placement stage the tool follows the guide lines of regions and partitions specified by the designer in the floorplan stage. The global placer engine uses the information of the design hierarchy (if available) to place related standard cells close to each other to reduce both timing and congestion as much as possible. Usually the global placement process is done in iterations, the quality of the placement improves proportionally with the increase of this number of iterations because the engine uses some information from the previous iteration to enhance the current one. The designer must take care that the increase of the number of iterations will lead to the increase of placement runtime.

After the tool initially placed the design in the global placement stage, the tool must now complete the placement process by using detailed placement. In the detailed placement the tool starts to legalize the location of each standard cell to make sure that it does not violate any DRC rule and does not overlap with another cell. The detailed placement process can either be timing driven or congestion driven or both, for example: in timing driven objective if two cells are overlapping the detailed placement engine will calculate the timing delay of each path that contains each of the standard cells, and then the engine will make a decision to move the cell that will best benefit the overall timing. In congestion driven detailed placement the engine tries to introduce gaps between standard cells in order to increase routing resources as much as possible.

After the tool finishes placing the design the designer must invoke the tool's placement checker in order to check for placement violation that the detailed placer might have missed. An example of these violations is the placement of cells on rows not powered by the power grid due to floorplan mistakes, or in case of multi-voltage designs the placer could mistakenly place cells of different power domain in wrong power domain regions. Also the designer could invoke the DRC checker engine to make sure that there are no DRC violations between the placed cells and the existing power grid routing.

For the tool to roughly estimate the interconnect delays during the timing calculations that it does in the placement stage, the tool routes the design using the global routing engine. In global routing process the engine divides the core area into small cells called global cells (G-cells), then it connects the center of those cells with each other using available routing resources (metal layers) depending on the standard cells that is located in each global cell. For example: If two standard cells are connected together logically and each of them is located in separate global cells physically then the global router will connect the centers of these two cells together. But if the standard cells were located in the same global cell then the global router will not connect them. Global routes are only estimate routes that help the tool roughly calculates the interconnect delays. Global routing engine should be fast because at this early stage of the design there is a lot of cell movements and there by a lot of routing modification. The quality of global routing depends on how well those routes will correlate to the final routing of the design. The quality of global routing can increase by reducing the size of the G-cells but this will lead to the increase of run time.

Figure A.8 shows an example of G-cells of the global router.

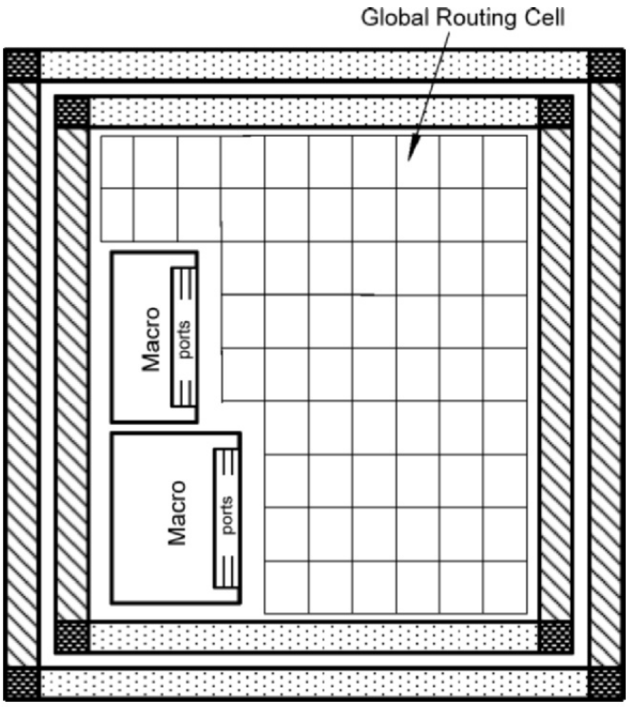


Figure A.8: Global router G-cells [21]

A.7 Clock Tree Synthesis

Clock tree synthesis (or CTS) stage is divided into three main stages, which are pre-CTS optimization stage, CTS stage and post-CTS optimization stage. Pre-CTS optimization stage is the first stage of optimizing the design. Usually optimization is done in two main steps global optimization and local optimization. In global optimization the optimization engine tries to locate and solve bottle neck timing violations across the design. Bottle neck timing violations are timing paths that are violating timing constraints requirement due to a common timing path between all these paths, if this common timing path is fixed all the violating timing paths that includes this timing path are fixed. For example if a driver is driving ten endpoints and this driver has a driving strength of one, then all the ten endpoints (timing paths) are violating timing constraints due to the big delay introduced on those ten paths by the driver due to its small driving strength. So if the optimizer replaced this driver with a driver of greater driving strength, this introduced big delay will disappear, which will make those ten endpoints meet the timing constraints requirement and will no longer be violating.

In detailed optimization, the optimizer handles each violating timing path separately and tries to fix it. The run time of the detailed optimization step is directly proportional with the number of violating paths; therefore it is always advised to run the global optimization step first to solve as much as possible bottle neck violations before running the detailed optimization. Usually you could specify different objectives for the optimizer to work on; examples of these objectives are maximum transition, maximum capacitance, setup and hold violations [23]. Where the optimizer tries to work on each of these objectives separately or at the same time to reduce these violations as much as possible. It is also a good practice to start the first optimization runs using maximum transition and maximum capacitance objectives only, this is because a lot of timing violations are due to either capacitance or transition violations because when the timing engine tries to calculate timing delays it uses transition and capacitance to figure out the value of the delays from the timing tables of the library, if the values of either the transition or capacitance are out of the range of the timing tables the tool extrapolates in order to calculate the timing delay, usually the delay calculated depending on this extrapolation process is not accurate causing timing violations.

The designer can run the optimization step several times as long as the timing is converging. If timing stopped converging and there still exists timing violations then the designer should try to look at this violations manually. The problem could be due to a real issue in the design preventing it from being fixed or due to tight timing constraint. Either way the designer should investigate more and try to solve the problem. In pre CTS optimization stage the designer 'does not optimize the design for hold violations as the clock tree has not been built yet and the delays on the clock paths are not calculated accurately. After each optimization run the designer go through an incremental placement and routing run to fix the location and correct the connections of newly introduced or modified design elements. This is done to ensure that the timing information calculated by the timing engine is accurate.

After completing the pre-CTS optimization stage the designer can now build his clock tree network. The main objective of the CTS stage is to reduce as much as possible the latency between the clock sources and clock sinks and to eliminate as much as possible clock skew between clock sinks. Clock skew is the difference in the arrival time between two sequentially-adjacent registers. This difference must be as small as possible to insure that the clock reaches all its sinks at the same time. We begin defining the clock tree by specifying its specs like defining the maximum transition of the clock signal, maximum capacitance, maximum fanout, maximum latency, maximum skew. We also introduce some extra constraints like defining clock uncertainty which is the uncertainty of the clock edge to encounter for any clock disturbance from the clock source.

During the CTS stage a special buffers (inverters) are used to build the clock network. This buffers are characterized by having equal rise and fall propagation delays, this is important to be able to balance the clock network accurately. These buffers are not used in the normal timing optimization as usually these buffers has a bigger size than normal buffers. The equal rise and fall times of these buffers are needed so as to minimize the clock pulse narrowing issue, where the clock pulse width gets smaller every time it passes through a buffer, also to overcome this problem CTS engines can use inverters instead of buffers.

Every CTS engine has its own algorithms to build the clock network, but there are common strategies that can be used to guide the tool to build a better network. For example the designer can manually divide the network into levels, where he could balance the endpoints in each level separately. This balance can be done by setting a special tool property on the endpoints that is not included in this level so that the tool ignores them and do not try to balance them with the rest of the endpoints. After balancing a certain level the designer then instruct the tool to set this level as fixed so as not to modify it, and then the designer can move to the next level and repeats the same process. Another approach that can be used in some tools is what is called skew groups, where the designer collects all the endpoints that need to be balanced together in groups, and then the designer instructs the tool to balance the elements of these groups together in one run.

After completing the CTS stage the design must go through incremental placement and routing to fix the locations and connections of all the newly inserted buffers and inverters. At this stage the clock network is no longer ideal and the delays on the clock network can be calculated accurately. It is also a common practice to route the clock network completely at this stage. To achieve that the clock is routed using real metal connections. This is done to ensure two points, first that the clock routing is done before routing the rest of the design so that it is not interrupted by any other routes and the connections are short and straight as much as possible to reduce the interconnect delays on these routes. The second point is that the use of these real metal connections will ensure accurate delay calculations on these connections which is critical at this point.

Now that the clock network is in place a final optimization stage is needed to optimize the timing violations that were introduced due to the modifications that occurred in the CTS stage. The post CTS optimization stage takes care of that, it also fixes hold violations that was ignored in the pre CTS stage as the clock network is now in place and delays can now be accurately calculated.

A.8 Routing

This is the final main stage of the P&R flow. In this stage all the design pins are hooked up using real metal connections. Usually routing process is done in two steps, in the first step the routing engine connects the design pins together using metal connections, in this step the routing engine tries as much as possible to connect all design pins without violating DRCs. The engine follows the route guides of the global router as much as possible but if it got stuck it can violate DRCs in order to make the connection. The outcome of this stage is a fully connected design but not a DRC free one.

The second step of routing is the final routing, where the routing engine tries to fix all the DRC violations that occurred in the previous routing step. In final routing the routing engine can use different techniques to solve DRC errors, example of those techniques are off-grid routing, non-preferred direction routing, wire tapering ...etc. The main objective of final routing is to reach a DRC free connected design. Several iterations of final route could be applied to the design as long as the number of DRC violations is decreasing. If the number of violations did not reach zero and are not decreasing with final route iterations the designer must manually look at these violations. There might be a design issue (for example a routing blockage) that might be preventing the tool from fixing the DRC error.

Before the designer goes into fixing all the DRC errors, he must check first that timing has not changed due to the routing stage and there are no timing violations. If there exists timing violations then the designer must go through a post route timing optimization stage to fix those errors first. The post route optimization stage differs from the normal optimization stage as it uses techniques that help to fix timing without disturbing the current state of the design. Examples of these techniques are white space optimization, where the tool adds the extra optimization buffers in white spaces existing between the already placed cells. Another technique is footprint optimization where the optimizer size up the existing standard cells using cells of exactly the same size.

After fixing all timing violations and all DRC errors the designer must also check LVS (Layout Versus Schematic) errors. In LVS the tool extracts all the design connections and constructs a netlist of the design using this extraction, then it compares this netlist with the original design netlist to make sure that all the connections in the design is made accurately. Now the design is almost ready for fabrication, some final steps are needed, examples of these steps are filler cells insertion and metal fill insertion. In filler cell insertion the free remaining spaces in the core area are filled with special cells called filler cells. These cells are needed to guarantee the continuity of N-well and implant (VDD and GND) layers on the standard cell rows. Metal filling is needed to ensure good distribution of metal density all over the die area. In metal filling process the tool adds metal pieces to areas where there is no metal. The tool must make sure not to violate any DRCs. These metal pieces are connected to the ground supply. Metal filling can introduce extra capacitance to the design. The designer should recheck his timing figures after metal fill.

There are some tricks the designer can use to improve the quality of his design, like replacing single cut VIAs with double cut VIAs; also the designer could use techniques like wire spreading to reduce the effects of crosstalk. He could even shield his clock network using grounded metal layers to reduce the effect of coupling. Also he could increase the width of metals used in clock routing in order to reduce the total resistance on these nets to speed up the clock network. All these tricks need to be accounted for in earlier design stages so as not to disturb the final routing of the design.

Now the design is ready to be shipped to the factory. The design is exported in a binary format called GDS. Also the design final netlist is exported for the final stage of post layout verification. Some other files are also needed for verification like interconnects capacitance and resistance extraction called Standard Parasitic Exchange Format (SPEF). Also interconnect delays are calculated and exported in a format called SDF (Standard Delay Format).

A.9 Post layout verification

This is the final stage of design checking before the design is shipped to the factory for fabrication. The design goes through two types of verification at this stage, functional verification and physical verification. In functional verification as mentioned before the design is functionally verified either by simulation or formal verification or both. As this is the final stage of building the design and no more editing to the design is allowed after the design is shipped to the factory, it is essential that the design is tested by all means of verification available. Therefore at this stage the design is verified by simulation and formal verification.

Also to take timing into consideration during simulation the designer loads the real interconnect delays into the simulation tool so that the tool can account for the effect of these delays during the verification process. Usually this delays comes in a format called Standard Delay Format (SDF), this file is generated by the P&R flow.

The physical verification process includes DRC checking, LVS checking and any other checking that is implied by the factory. Usually the fabrication factory requires the design to pass certain checking before it is shipped for fabrication. The designer must make sure that his design passes those tests. Although the design has undergone some of this checking at the end of the P&R flow it is always a good practice to repeat this test using different tools or using a more specialized tool. This re-check is done to make sure that no errors has been missed out due to any tool calibration mistake.

After completing both type of checking the designer can now ship his design for fabrication. Usually the designer ships his design to the factory and then the factory start processing it and gives the designer the possibility to make some final modifications to certain metal layers before a final cutoff date after which no modifications are allowed to any design layer.

To conclude, this chapter covered a complete digital ASIC design flow, and as mentioned before that this is not a fixed flow. Every designer can modify the flow to suite his design and his needs. In this chapter we tried to cover the main design flow stages in a level of detail that tries to compromise between simplicity and knowledge. Figure A.9 shows the tools used in each design step.

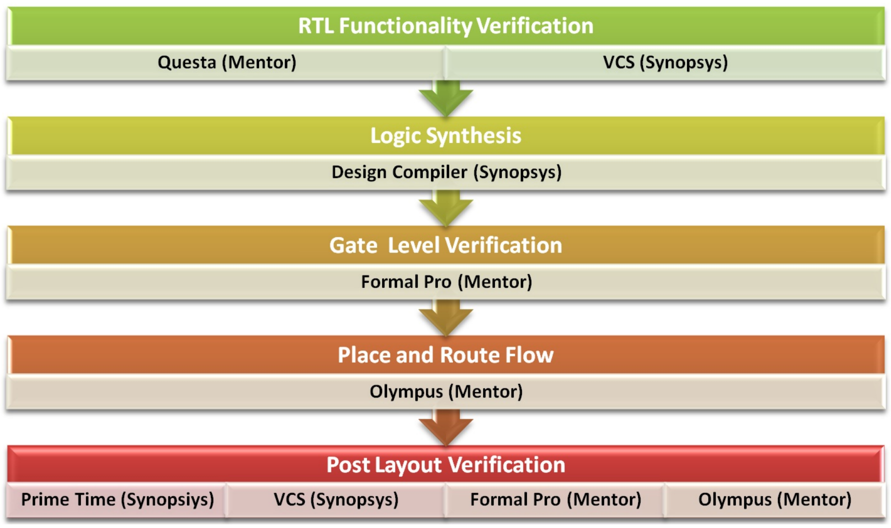


Figure A.9: Tools used in design flow

References

[1] Wikipedia, “SPARC”, *(http://en.wikipedia.org/wiki/SPARC)*, Accessed [26-8-2012].

[2] David L. Weaver and Tom Germond, “The SPARC Architecture Manual”, *PTR Prentice Hall*, Version 9. 1994.

[3] Sun Microsystems, “OpenSPARC”, *(http://www.opensparc.net/about.html)*, Accessed [28-8-2012].

[4] Sun Microsystems, “OpenSPARC T1 Processor Datasheet”, *Sun Microsystems, Inc*, March 2006.

[5] Sun Microsystems, “OpenSPARC T1 Microarchitecture Specification”, *Sun Microsystems, Inc*, April 2008.

[6] Durgam Vahia, “OpenSPARC T1 on Xilinx FPGAs”, *RAMP Retreat (http://ramp.eecs.berkeley.edu)*, Summer 2007. Accessed [26-8-2012].

[7] Shrenik Mehta, “Open Hardware Innovate with OpenSPARC”, *CANDE Workshop*, September 2006.

[8] Durgam Vahia, Thomas Thatcher, and Paul Hartke, "OpenSPARC T1 on Xilinx FPGAs", *RAMP Retreat (http://ramp.eecs.berkeley.edu)*, January 2008. Accessed [26-8-2012].

[9] David L. Weaver, "OpenSPARC Internals : OpenSPARC T1/T2 CMT Throughput Computing", *Sun Microsystems, Inc*, October 2008.

[10] Sun Microsystems, "OpenSPARC T1 Processor Megacell Specification", *Sun Microsystems, Inc*, March 2006.

[11] Synopsys, "Design Compiler Guide", *Synopsys, Inc*, December 2010.

[12] Sun Microsystems, "OpenSPARC T1 Processor Design and Verification Guide", *Sun Microsystems, Inc*, 2009.

[13] Mentor Graphics, "Olympus-SoC Manual", *Mentor Graphics Corporation*, May 2010.

[14] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun, "Niagara, a 32-way Multithreaded Sparc Processor", *IEEE Micro*, MarchApril 2005.

[15] Mohamed Hamdy Merzban, "Development of a Dual-core 64-bit MultiProcessor Based on the OpenSPARC Design", *Faculty of Engineering, Cairo University*, February 2011.

[16] Ana Sonia Leon and Denis Sheahan, "The U1traSPARC Ti: A Power-Efficient High-Throughput 32-Thread SPARC Processor", *Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian*, Nov 2006.

[17] William K. Lam, “Hardware Design Verification: Simulation and Formal Method-Based Approaches”, *Prentice Hall*, August 2005.

[18] Himanshu Bhatnagar, “Advanced ASIC Chip Synthesis Using Synopsys Design Compiler Physical Compiler and PrimeTime”, *Springer*, December 2001.

[19] Don Mills, “RTL Coding Styles That Yield Simulation and Synthesis Mismatches”, *Clifford E. Cummings Sunburst Design, Inc*, August 1999.

[20] Patrick Lee, “Introduction to Place and Route Design in VLSIs”, *Lulu.com*, December 2006.

[21] Khosrow Golshan, “Physical Design Essentials”, *Springer*, May 2007.

[22] Michael John and Sebastian Smith, “Application Specific Integrated Circuits”, *Addison-Wesley Professional*, June 1997.

[23] J. Bhasker and Rakesh Chadha, “Static Timing Analysis for Nanometer Designs”, *Springer*, April 2009.

الملخص

يعد "OpenSPARC T1" الذي تم تطويره بواسطة شركة "SUN" أول مُشغّل مفتوح المصدر، متعدد النويات و المسارات. ويهدف هذا البحث إلى تصميم دائرة متكاملة لنواة هذا المشغّل باستخدام تكنولوجيا "أكسيد – معدن – شبه موصل المتتامة" ١٣٠ نانو متر.

في البداية قمت بإدخال عدة تعديلات على التوصيف الإلكتروني المفتوح المصدر لنواة المُشغّل "OpenSPARC T1" ، ومنها: تحويل الذاكرة و تقليل مسارات المُشغّل و إلغاء وصلات الإختبار من أجل تقليل حجم المُشغّل لكي يتناسب مع شريحة مساحتها ٤x٤ مم كما هو مطلوب من قبل منشأة التصنيع. تم التحقق من سلامة الأداء الوظيفي للتوصيف الإلكتروني المعدل لهذا المشغّل بواسطة أكثر من ٥٠٠ إختبار مقدم من شركة "SUN".

ثم قمت بتوليف التصميم لتحويله من التوصيف الإلكتروني عند مستوى "نقل السجلات" إلى مستوى "البوابات" المعادل. بعد ذلك تم تنفيذ التصميم فعليا بإستخدام مجموعة خطوات التنفيذ الرقمي الملقبة ب "المواضع و المسارات"، بما في ذلك الخطوات العادية مثل "تخطيط الأرضية" ، "تحديد المواضع" ، "التحسين" ، "توليف تفريعات المنظّم" و "التوصيل". وأخيرا، تم التحقق من التصميم عن طريق سلسلة من الإختبارات الوظيفية و تقييم الأداء لضمان الفعالية والأداء.

إستطعت من خلال هذا البحث تصميم نواة مُشغّل OpenSPARC T1 مزدوج المسارات ويعمل على تردد ١٠٠ميجا هرتز و تبلغ مساحتة ١٦ مم^{٢} متضمنة الوسادات.

ضع صورتك هنا

مهندس:	محمد محمود محمد فرج
تاريخ الميلاد:	١٩٨٥\١٢\٢٩
الجنسية:	مصري
تاريخ التسجيل:	....\....\..........
تاريخ المنح:	٢٠١٣ \ ١ \ ١٦
القسم:	الإلكترونيات والاتصالات الكهربية
الدرجة:	ماجستير

المشرفون:

أ.د. سراج الدين السيد حبيب

أ.م.د. حسام علي حسن فهمي

الممتحنون:

أ.د السيد مصطفى سعد (الممتحن الخارجي)

أ.د إبراهيم محمد قمر (الممتحن الداخلي)

أ.د سراج الدين السيد حبيب (المشرف الرئيسي)

أ.م.د. حسام علي حسن فهمي (مشرف)

عنوان الرسالة:

تصميم دائرة متكاملة لنواة المشغّل OpenSPARC T1

الكلمات الدالة:

دائرة متكاملة ، SPARC ، تخطيط ، مُشغّل ، تصميم

ملخص الرسالة:

يعد "OpenSPARC T1" الذي تم تطويره بواسطة شركة "SUN" أول مُشغّل مفتوح المصدر، متعدد النويات و المسارات. ويهدف هذا البحث إلى تصميم دائرة متكاملة لنواة هذا المُشغّل باستخدام تكنولوجيا " أكسيد – معدن – شبه موصل المتتامة " ١٣٠ نانو متر. في البداية قمت بإدخال عدة تعديلات على التوصيف الإلكتروني المفتوح المصدر لنواة المشغّل "OpenSPARC T1" ، ومنها: تحويل الذاكرة و تقليل مسارات المشغّل و إلغاء وصلات الإختبار من أجل تقليل حجم المُشغّل لكي يتناسب مع شريحة مساحتها ٤x٤ مم. تم التحقق من سلامة الأداء الوظيفي للتوصيف الإلكتروني المعدل لهذا المُشغّل بواسطة أكثر من ٥٠٠ إختبار مقدم من شركة "SUN". ثم قمت بتوليف التصميم لتحويله من التوصيف الإلكتروني عند مستوى "نقل السجلات" إلى مستوى "البوابات" المعادل. بعد ذلك تم تنفيذ التصميم فعليا بإستخدام مجموعة خطوات التنفيذ الرقمي الملقبة ب "المواضع و المسارات"، بما في ذلك الخطوات العادية مثل "تخطيط الأرضية" ، "تحديد المواضع" ، "التحسين" ، "توليف تفريعات المنظّم" و "التوصيل". وأخيرا، تم التحقق من التصميم عن طريق سلسلة من الإختبارات الوظيفية و تقييم الأداء لضمان الفعالية والأداء. إستطعت من خلال هذا البحث تصميم نواة مُشغّل مزدوج المسارات ويعمل على تردد ١٠٠ميجا هرتز و تبلغ مساحتة ١٦ مم^{٢} متضمنة الوسادات.

تصميم دائرة متكاملة لنواة المشغّل OpenSPARC T1

اعداد

محمد محمود محمد فرج

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
الإلكترونيات والاتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور: السيد مصطفى سعد	الممتحن الخارجي
الاستاذ الدكتور: إبراهيم محمد قمر	الممتحن الداخلي
الاستاذ الدكتور: سراج الدين السيد حبيب	المشرف الرئيسى
الاستاذ الدكتور: حسام علي حسن فهمي	مشرف

كليـة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٣

تصميم دائرة متكاملة لنواة المشغّل OpenSPARC T1

اعداد

محمد محمود محمد فرج

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
الإلكترونيات والاتصالات الكهربية

تحت اشراف

أ.د سراج الدين السيد حبيب	أ.م.د. حسام علي حسن فهمي
قسم الإلكترونيات والاتصالات الكهربية كلية الهندسة ، جامعة القاهرة	قسم الإلكترونيات والاتصالات الكهربية كلية الهندسة ، جامعة القاهرة

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٣

تصميم دائرة متكاملة لنواة المشغّل OpenSPARC T1

اعداد

محمد محمود محمد فرج

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
الإلكترونيات والاتصالات الكهربية

كليــة الهندســـة - جامعـــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيـــة

٢٠١٣