TABLE DRIVEN ADAPTIVE,

EFFECTIVELY HETEROGENEOUS

MULTI-CORE ARCHITECTURE

By

SURPRIYA TIKE

Bachelor of Science in Electronics and

Telecommunication

Pune University

Pune, India

2007

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2011

TABLE DRIVEN ADAPTIVE,

EFFECTIVELY HETEROGENEOUS

MULTI-CORE ARCHITECTURE

Thesis Approved:

Dr. James E. Stine Jr.

Thesis Adviser

Dr. Chris Hutchens

Dr. Louis G. Johnson

Dr. Sheryl A. Tucker

Dean of the Graduate College

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Recent research and advancements with modern computer architecture emphasize heavily on the novel implementation techniques for performance enhancement. One of the techniques that have been under research is the exploitation of multi-core architectures. However, with increasing dynamic nature of the workloads of everyday computing, even general multi-core architectures seem to just touch an upper limit on the deliverable performance. This has paved way for meticulous consideration of heterogeneous multi-core architectures.

The notion behind the concept of heterogeneous multi-core architectures is smart allocation of workload to more equipped cores, which positively affects performance. Having multiple cores in a system, some dedicated for a special function certainly brings certain degree of flexibility in distributing workload more efficiently. However, these architectures can also face roadblocks in certain scenarios. Such architectures can be further enhanced by making the heterogeneity quality of the cores dynamic in nature. Making the cores dynamically heterogeneous implies that the cores can change their properties on-the-fly in accordance with the incoming loads.

This is quite significant, since the cores can change their properties independent of each other, and yet share processor resources on higher hierarchy levels. Some research and implementations have materialized in the past, which use workload monitoring to shut-off units for power optimizations. Some of these works also make use of Configuration ROMs or look-up tables being used for this purpose. Although, when managing multiple heterogeneous cores with ever-changing workloads, merely shutting of inactive units does not give the performance boost we are looking for.

This project, OSCAR (OSU Simulation for Computer Architecture Research) is an RTL based simulation environment for a heterogeneous multi-core architecture. The project implements four such dynamically configurable MIPS-like cores, which have shared memory resources on Level2 hierarchy through data coherency protocol. The dynamic configurations are done, by observing the nature of incoming workload for each core by deploying performance counters. The technique uses feedback from performance counters every couple of thousand instructions, and a configuration look-up table, to make the decisions for architecture changes.

Whilst this is a very powerful concept, future developments and expansions will make the architecture more proficient and promising. Another highlighting aim of the project is to make a framework of a modern multi-core architecture, which can be made available to the academia for further development.

CHAPTER II

BACKGROUND

In the past years, several platforms have been developed which actual showcase multi-core architectures or simulation environments for the same. Some of them are OpenSPARC T1/T2 by Sun Microsystems, Simics, SPIM, SimpleScalar, etc. Developers of these platforms encourage students to make use of the tools for academic purposes. However, looking at the necessity of the ongoing research focus in the field, the mentioned platforms pose certain difficulties in their usage.

2.1. Processor based platforms

The following entries describe academically available processor based platforms. Most of these are developed in high level languages, which is undoubtedly easier to deal with, but do not account for the real hardware investment.

2.1.1. OpenSPARC

Of all mentioned above, only UltraSPARC T1/T2 from the OpenSPARC project [1] is hardware based open-source development platform.

All others are instruction accurate simulators for MIPS-like architecture which are usually designed in high level languages such as C, C# or Java. UltraSPARC T1 (Niagara) released in 2005 was a 32/64 bit threaded microprocessor with 8 cores, and a total 32 threads. The downside of this architecture was the use of just one FPU unit shared between 8 cores. This problem was moderated but not solved in the UltraSPARC T2 (Niagara2). Niagara2 implemented one FPU for each of the 8 cores.

2.1.2. SimpleScalar

Dr. Todd Austin developed SimpleScalar [2] in 2001 while he was a PhD student at University of Wisconsin-Madison. The SimpleScalar architecture is derived from the MIPS-IV architecture. Several papers and articles published by him and his team point towards the need of an infrastructure for modeling and simulation.

These articles also emphasize on the need of such platforms to be open-source, and easily adaptable for academic use. Being written in a high level language, the tool provides limited detailing from implementation point of view.

2.1.3. SPIM

Dr. James Larus developed SPIM [3] (reversal of letters *MIPS*) while he was a professor at University of Wisconsin-Madison in 1990. The latest version being 9.0, is now being developed using Qt framework (cross-platform for Windows/Linux/Mac) and is also open-source. Although, since thus lacks the hardware aspect too, we will face same troubles with SPIM.

2.1.4. Simics

Simics [4] is a *full-system simulator*, which can be used to simulate complete computer systems. A full system simulator provides virtual hardware, which is completely independent of the host machine and its architecture. Simics can thus be used to create environments, which encompass multiple processors, memory units, network, and other interconnects. Thus, Simics has been developed to provide a very high level of abstraction as far as computer architecture is concerned. This makes Simics an unsuitable choice for detail designing and implementation of a processor with multicore architecture.

These simulators that are designed using high-level languages might provide a user with faster design cycles, with little or no co-relation to its hardware implementation. These simulators are effective when the designers need a platform to perform basic development and testing of new ideas on MIPS-like architectures.

Using instruction accurate simulators certainly implies that developers need lesser time for development and testing, but the accuracy and importance of actual hardware implementation cannot be overshadowed.

2.2. Other Related Work

One focus area for related research work has been the on-chip cache hierarchy. It has been suggested that an increasing percentage of die area is being devoted to caches, and that on-chip L1 caches can alone comprise over 40% of the total die power budget [5]. The 2007 International Technology Roadmap for Semiconductors (ITRS) also suggests that by 2014 more than 94% of the overall area of SoC designs will be occupied by memories [6]. Thus a lot of work is being done in optimizing the memory subsystems, instigating a tradeoff between increased performance, and reduced power requirements.

Some of such papers suggest shutting down some memory blocks when not needed, which results in power savings. One such paper [7] implements a technique called *Selective Cache Ways*, implements a set associative cache, and selectively disables data cache ways during phases of modest cache activity. This decision is controlled by keeping a check on *Performance Degradation Threshold* (PDT) for tolerable performance loss.

The advantage was that no flushing of cache was required when enabling/disabling the cache ways, which saves on the heavy time penalty. Enabling/disabling memory units eventually reflect impressive power numbers. But any increase in performance finally is gained at the expense of direct increased power.

Multiple level caches typically display unbalanced access patterns, since most data is serviced by L1 cache. Another paper takes a global ruling of turning off multiple tier caches which sit idle for long times [8]. A majority of these ideas involve exclusively reducing energy usage by disabling caches or redundant core resources to save on the associated leakage power [9] [10].

It has also been shown that using specialized low-leakage transistors can significantly help energy consumption in L1 and L2 caches [11]. Implementing such a technique will require either a manually routed memory block, or a compiler using such cells.

2.3. Proposed Solution

OSCAR (OSU Simulation for Computer Architecture Research) is an RTL based simulation environment for a heterogeneous multi-core architecture. The advantage of an RTL based model is that the hardware can be modeled very accurately. Using cycle accurate simulation environment provides the required insight into the detail operations of the cores. The target library used for the project is IBM65LPE. Virage memory compilers from Synopsys was used to generate the on-chip memory blocks.

The project is completely synthesizable, and the resultant netlist can be used to evaluate power and timing numbers for all hierarchically modules, including the memory macros for various test codes. Thus this multi-core architecture with two on-chip memory hierarchies and a shared FPU unit, which is all modeled using RTL, makes a great platform for future development.

Looking at conventional units in microprocessor architecture, memories, heavy computational units, and core-to-core interconnect busses are typically the ones to create bottlenecks. Thus, the objective was to boost performance by tweaking these units. This work implements an idea having unit similar to selective cache ways [7], which change their architecture dynamically between 1way, 2way and 4way within each core. The cores can also be configured in three modes – power, speed, and adaptive. These settings can be made in the testbench.

The *power* mode is the most conservative mode, where minimal resources are allocated, which would save on overall power, but may be effectively more time consuming. The other *speed* mode deploys the fastest resources on the chip, which may consume more power than in power mode.

The third *auto* mode is the highlighting aspect of this work. Keeping in mind, that the multiple cores will have independent workloads with varied requirements, the cores in this mode adapt dynamically to suit these needs. Various units in the architecture, including some specific to each core, are modified as per need. This includes the execute and memory stage of pipeline per core, and the global FPU unit.

The Dcache will restructure itself to act as power and speed driven from time-to-time, depending on load requirements. When working on the heavy computational units in the Execute stage of the datapath, a separate provision for fast and slow multipliers/dividers has been made. Thus, depending on requirement, one of them can be used, while the other one remains inactive. The FPU unit(s) is a global resource for the four cores. For increased FPU type workloads for more than one core, both the FPU units are turned on for faster servicing rates.

CHAPTER III

IMPLEMENTATION

The RTL of the architecture has been implemented in Verilog using structural style of hardware modeling. This is specifically important because the project aimed at designing units that didn’t use operators with huge untailored hardware for the computations.

3.1. Architecture Top Level description -

Four single threaded pipelined cores

4KB Level1 Dcache per core withy LRU (configurable)

4KB direct mapped Level1 Icache for each core

16KB 4-way set associative shared Level2 Dcache

16KB direct mapped shared Level2 Icache

Two floating-point units (configurable)

Arbiter logic between cores and Level2 cache and FPUs

Directory implementation for data coherency

Besides the components mentioned above, the architecture also has a shared L3 Dcache and L3 Icache, which is considered to be the main memory, and resides outside the chip.

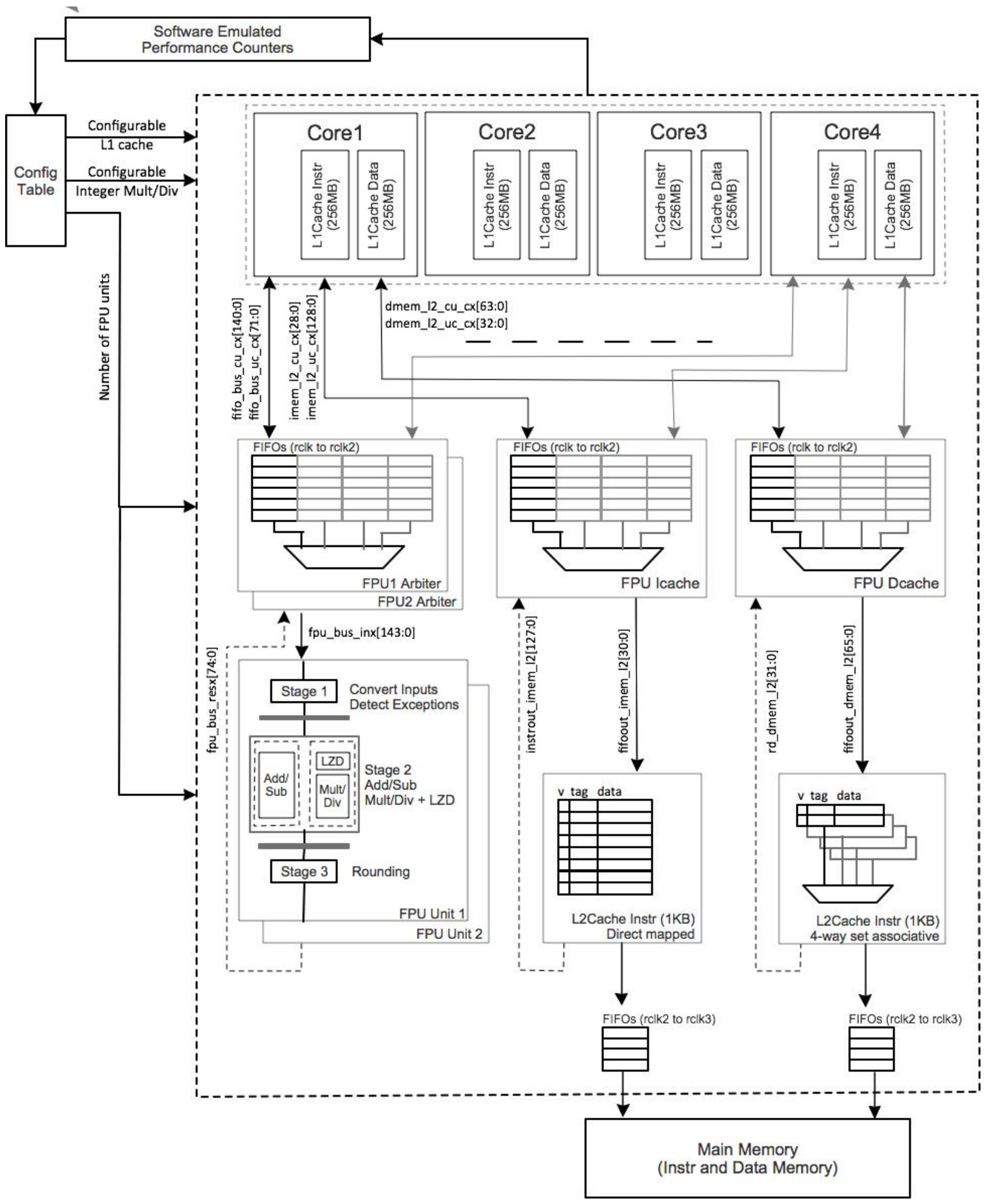


Figure 1 - Top Level Architecture

The top-level architecture consists of two levels of hierarchies. The top-level hierarchy illustrated in Figure 1 implements four MIPS-like cores. These are single-threaded cores with independent 4KB L1 data and instruction caches. Each core also has a standard 5-stage pipelined datapath and exception handler unit (coprocessor0).

The four cores share 16KB Level2 data and instruction caches, and FPU unit(s). The directory structure also operates parallel to the Level2 Dcache. The arbiter layer efficiently handles the complete interface between the Level2 units and the cores. The major components of this hierarchy are the huge memory components, which is why they can't use the same clock as the core frequency. All the units in this hierarchy use a separate clock domain with reduced frequency called rclk_2. The arbiter layer implements asynchronous FIFOs, which use writing clocks same as the core frequency (clk) and the reading clock same as the slower clock (rclk_2).

Besides the memory elements and the directory structure, the Level2 hierarchy also implements pipelined Floating Point Unit(s). The arbiter layer also facilitates the sharing of the PFU unit(s) between the cores. Besides the normal floating-point operations of add/sub, the FPU unit also implements units for high radix combined unit for mult/div operations on single and double precision floating point numbers (IEEE754-1985).

One of the highlighting features of the design is adaptability of the architecture for different types of workloads. The goal of this flexibility is to attain power and speed efficient models and provide quantitative analysis on design trade−offs, thus suggesting the best possible architectural configuration for a specific workload.

Architectural changes are done through a lookup table (similar to a Config ROM), which dynamically adapts to different architectural configuration with the help of performance counter. Various performance counters are deployed at various locations in hardware to constantly gauge the workload individually for all cores. The performance counters examine workloads for integer mult/div, floating point and memory intensive instruction windows. The size of this window can be very easily set in the testbench.

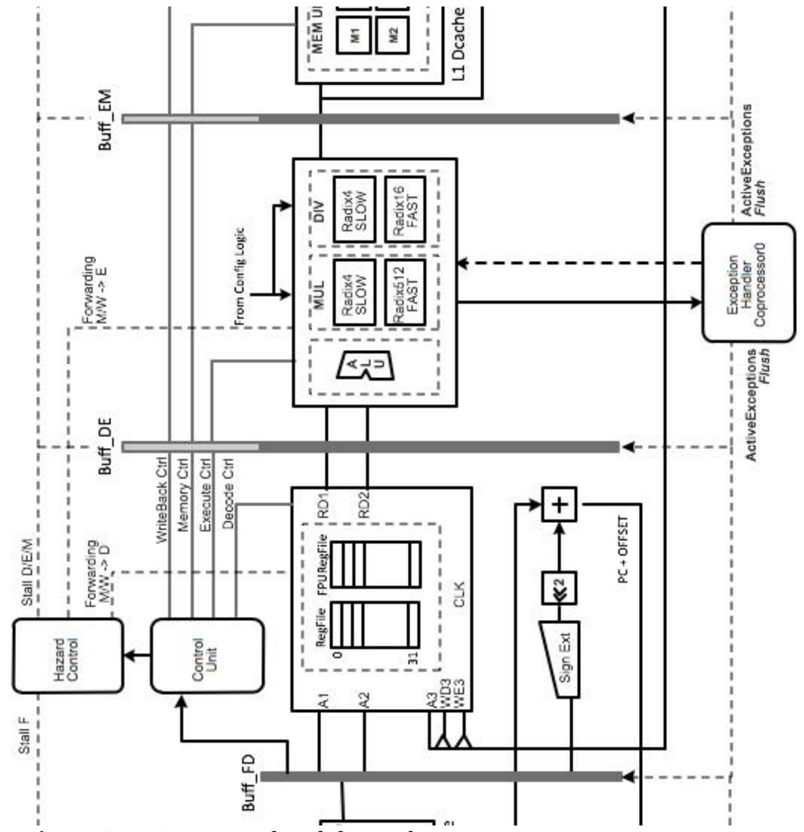


Figure 2 - 5 Stage pipelined datapath

The configurable components in architecture are -

a) Configurable number of FPU units

   Floating point intensive applications will enable FPU units (one or two) depending on demand. Each pipelined FPU unit comprises of a Radix512 mult−div unit, adder, and integer−float conversion unit. By default one FPU unit is enabled; second FPU unit is enabled when the workload of more than two cores increases beyond a threshold. The user can configure this threshold.

b) Dynamically configurable-way L1 cache for memory intensive applications.

   This decision is made independently for each core depending on needs. For a particular core, if Dcache misses exceed beyond a certain threshold, architecture for Level1 Dcache can be dynamically switched between direct mapped, 2way and 4way set associative. Again, the user can configure this threshold. By default, the Dcache is configured to be in direct mapped. The user can configure the L1 Dcache to be in direct mapped, 2way or 4way architecture, or let the decision be made dynamically according to the requirements.

c) Enabling and disabling high/ low performance integer units

This decision is made independently for each core depending on needs. The user can configure the cores for high performance vs. low performance integer multiply−divide units in testbench. The high performance unit uses Radix512 multiplier and Radix16 divider. The low performance unit uses Radix16 multiplier and Radix4 divider. Units are enabled/disabled through clock gating. By default, the low performance integer unit is enabled. The user can also choose to configure otherwise.

The flexibility added in the architecture for the availability of dynamic configuration does add a lot of redundant hardware. This definitely will influence area numbers. Although, the area count of the active hardware in a given configuration will be lesser, and is important. Although, since the redundant hardware will be mostly clock gated when unused, the power numbers should not be influenced beyond scope.

3.2. Performance Counters

Performance counters keep a measure of the workload in every instruction window (e.g.: 256 instructions for and 8-bit counter). Various performance counters are deployed in hardware to gauge workload for memory, FPU or integer mult/div intensive applications. These results from these counters are fed to the Config logic, where the architecture changes are decided for the next instruction window. The Config logic also contains several user-defined thresholds which when crossed, can trigger changes in architecture. Thus, some of the core configurations can be changed to enhance performance during run-time.

3.3. Single-threaded core

The chip consists of four single threaded pipelined cores, which share a common 16KB L2 Dcache, and 16KB L2 Icache, and FPU unit(s) through an arbiter. Each of the cores consists of basic blocks like the datapath, control unit, hazard control unit, and a coprocessor0 for exception handling. The following sub-sections will give descriptions about the various units of the architecture in details:

3.3.1. Datapath

The datapath is a classic five-stage pipeline consisting of following stages: fetch, decode, execute, memory, writeback. A detailed block diagram of the datapath is illustrated in Figure 2.

Buffers separate each of these stages. Fetch and Memory stages implement synchronous SRAM memories. The following sections describe the stages in the 5-stage pipeline.

3.3.1.1. Fetch

The fetch stage implements a synchronous direct-mapped SRAM, which has a capacity of 256x16B (4KB), thus storing 4 instructions in each line of SRAM. Level2 Icache is also designed in the similar manner. Thus, when an instruction miss is generated for an address, four instructions are fetched at a time.

J-type Instructions						
J (Jump)	JAL	JALR				
R-type Instructions						
NOP	SYSCALL	BREAK				
Add Subtract						
ADD	ADDU	ADDI	ADDIU	SUB	SUBU	
AND	ANDI	OR	ORI	XOR	XORI	NOR
Comparison type						
SLI	SLTI	SLTU	SLTIU			
SLL	SLLV	SRL	SRLV	SRA	SRAV	
Integer Multiply Divide						
MULT	MULTU	DIV	DIVU			
MFHI	MFLO					
I-type Instructions						
Load Store						

LW	LB	LUI	LH	LHU	LBU
SW	SB	SHW			
Branches					
BEQ	BNE				
BGTZ	BLTZ	BGEZ	BLEZ	BGEZAL	BLTZAL
Coprocessor Instructions					
Floating Point Instructions					
ADD.S	ADD.D	SUB.S	SUB.D		
DIV.S	DIV.D	MUL.S	MUL.D		
MOV.S	MOV.D				
NGT.S	NGT.D	ABS.S	ABS.D		
EQU.S	EQU.D	LT.D	LT.S	LE.D	LE.S
CVT.S.D	CVT.S.W	CVT.W.D	CVT.W.S	CVT.D.S	CVT.D.W
MTC0	MFC0	RFE			
MFC1	MTC1	CTC1	CFC1		
LWC1	SWC1				

Table 1 - Implemented Instruction Set

3.3.1.2. Decode

Decode stage has two register files, for integer and floating point numbers.

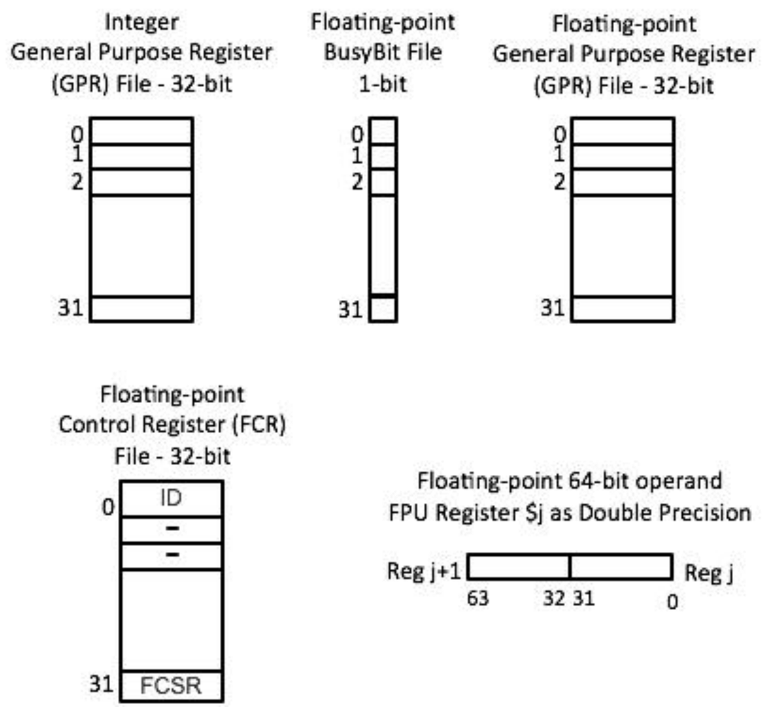


Figure 3 - Register File Set

This stage also has control unit logic, to evaluate the incoming instructions, and begin preparing the operands for the execute stage. Currently the architecture recognizes 87 instructions, which include both integer, and floating-point instructions. The currently implemented instruction set is illustrated in Table 1. Depending on the control signals coming from the control unit¸ the operands are prepared from the register file (integer register-file or floating-point register-file).

3.3.1.3. Execute

The execute stage consists of a basic ALU for integer add, subtract, shift, logical operations. There is a separate unit for integer multiplication and division. To provide certain degree of flexibility, there are two units of mult-div units placed per core – for high and low performance.

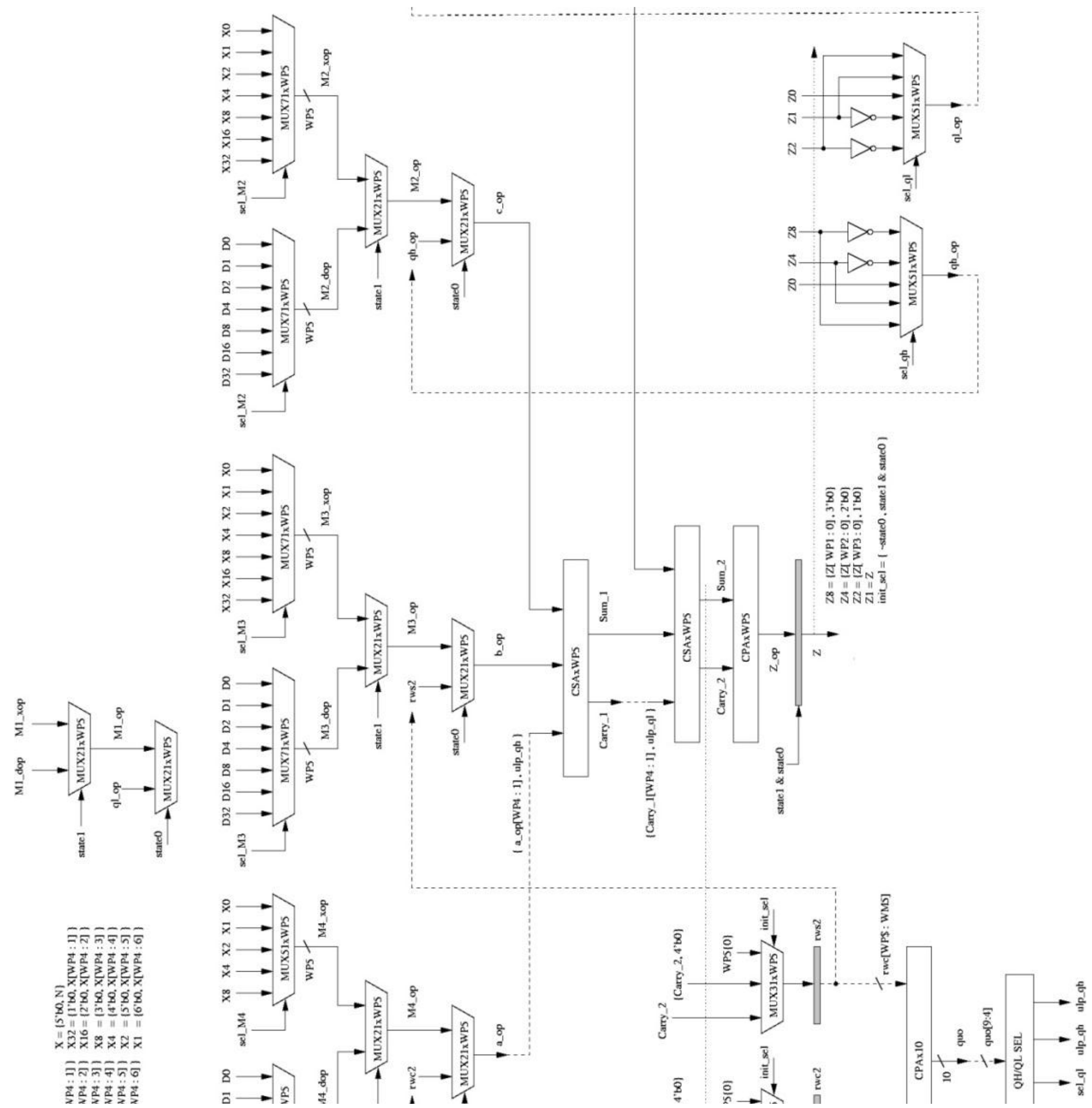


Figure 4 - Integer Radix16 Divider (fast)

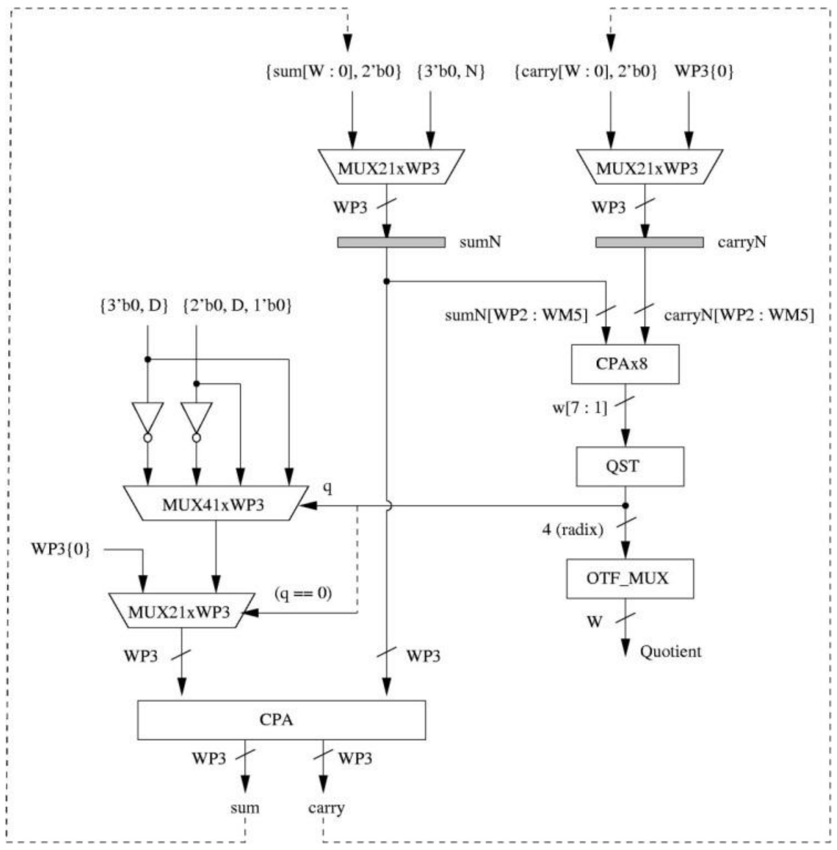


Figure 5 - Integer Radix4 Divider (slow)

The high performance unit uses Radix512 multiplier and Radix16 divider. The low performance unit uses Radix16 multiplier and Radix4 divider. Figure 5 and Figure 4 illustrate the block diagram for slow and fast dividers [12].

While the integer processing units are placed within the core, the floating-point units are placed outside the core and are shared between all four cores through the arbiter logic. For floating point instructions, the operands from the execute stage are routed to the floating point through the arbiter and asynchronous FIFOs.

3.3.1.4. Memory

The memory stage implements the other synchronous SRAM, which has a net capacity of 256x4Bx4 (4KB), thus storing 1 word of data in each line of SRAM. As shown in the Figure 6, the Dcache is implemented in 4 units of memory, each having a capacity of 256x4B (1KB).

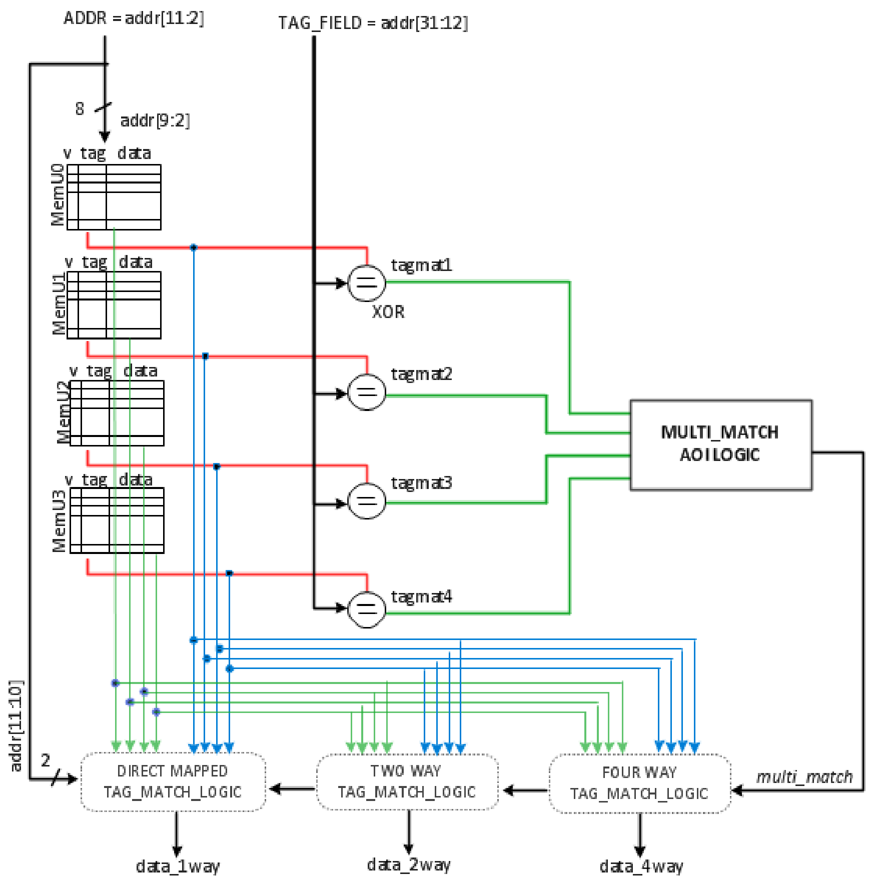
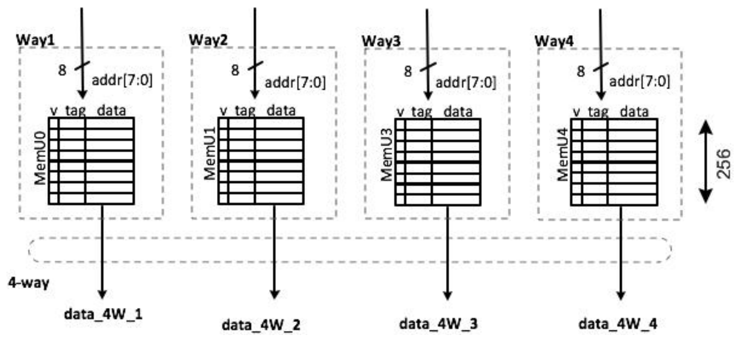
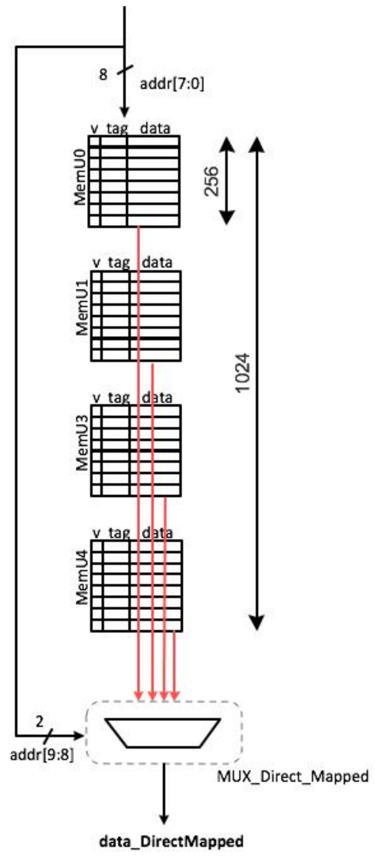


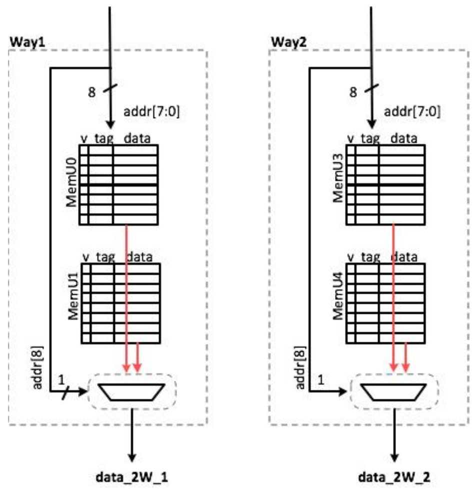
Figure 6 - Level1 Dcache architecture flexibility

These four units are arranged to behave as direct mapped, 2way or 4way set associative depending on requirement. The architecture for the memory unit can be decided by selecting the mode in the testbench. When the processor is set in the auto mode, the memory units virtually rearrange themselves and switch between 1way/2way/4way set associative cache system, depending on the signal from the Config Logic. In addition to this flexibility, the Level1 Dcache also implements LRU replacement policy (not shown in the figure) when the memory is configured as 2way or 4way set associative.



a) Four way set associative architecture





b) Direct Mapped architecture

c) Two way set associative architecture

Figure 7 - Memory Subsystem adapted into different architectures

Figure 7 displays the units in their 1way (power mode), 2way and 4way (speed mode) set associative configurations. In all the cases, effective cache size is maintained to be 1kB, thus, needing maximum 10bits for addressing. The number of tag bits needed for tag-match comparisons will differ in each of the modes - 4way needs most bits (22bits), and 1way needs least (20bits). Thus, for this system to work correctly, tag memory stores 22bits, thus satisfying all cases. Besides, another bit is required as the valid bit.

3.3.1.4.1. Challenge with adaptive memory subsystem - Implementation of *multi_match*

While dynamically switching the memory architecture improves the performance when required, this switching can result in limited data loss. No overhead losses are encountered when the memory adapts into a higher set associative architecture. With memory intensive phases, the memory adapts into a lower set associative architecture, which can result in some initial miss counts due to reduced *atomicity of ways*. This is resulted due to duplicate or invalid data entries identified in the new lower order associative memory system.

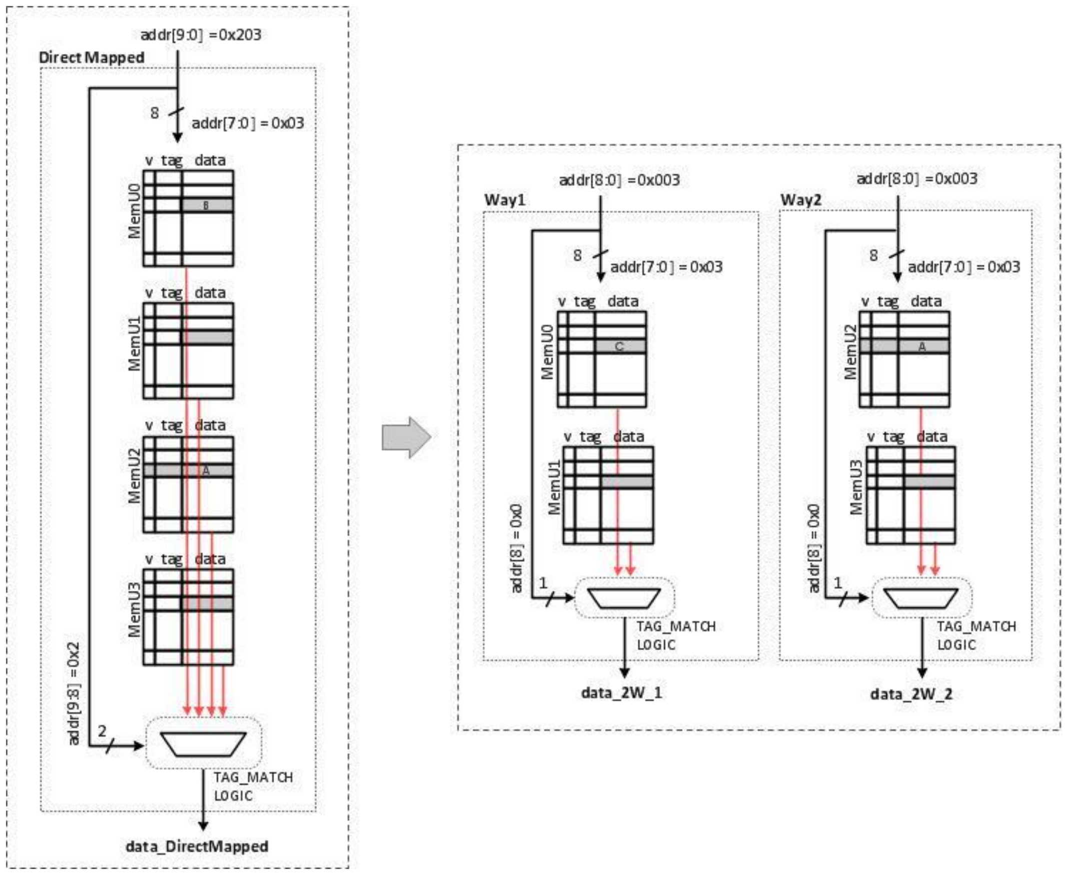


Figure 8 - Memory moving from lower to higher set associativity

The following example illustrates this performance loss. The experiment goes through five steps: a) store data A at address 0x200, b) store data B at 0x000, c) change memory architecture, d) store data C at 0x000, e) load data from 0x000. Figure12 shows the behavior of the memory units when architecture is changed from direct mapped to 2way set associative. From Figure 8 the architecture changes from direct mapped to 2way set associative. After a walkthrough of the five steps in the experiment are complete with the setup, the data can be read without any inconsistencies.

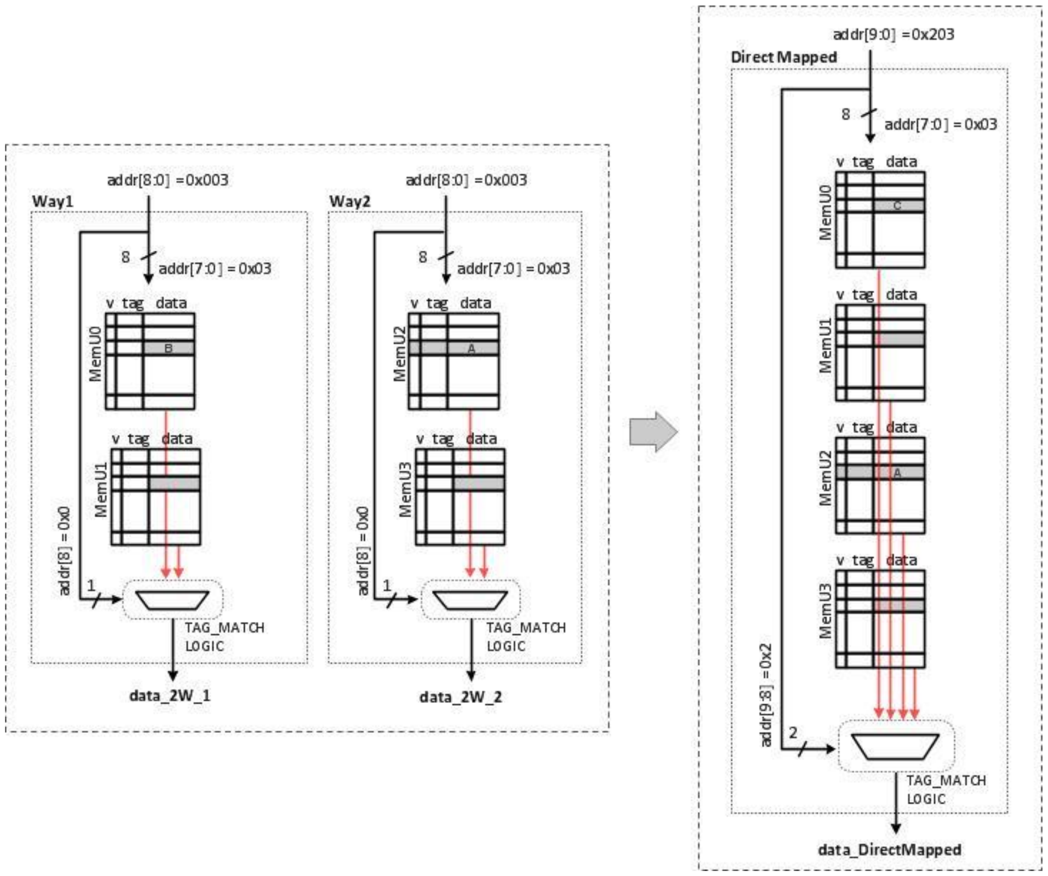


Figure 9 - Memory moving from higher to lower set associativity – encounters invalid entries

Now assume the set-up illustrated in Figure 9, where the memory architecture changes in the other direction, from two-way set associative to direct mapped. In this case, at the end of the 5th step when fetching data from the direct mapped memory two tag matches will be encountered. A technique needs to be implemented so the core does not use the invalid data entries from the cache.

One easy technique to handle this is flushing the cache units when the architecture change is made. Flushing the cache at every change can have very significant impact on the overall performance [13]. The impact is even greater with larger caches. OSCAR implements a technique where such cache entries are invalidated. As and when such invalid entries are identified, *multi_match* is raised, the entries are invalidated, and a *data_miss* is issued, and the true data is fetched from DcacheL2.

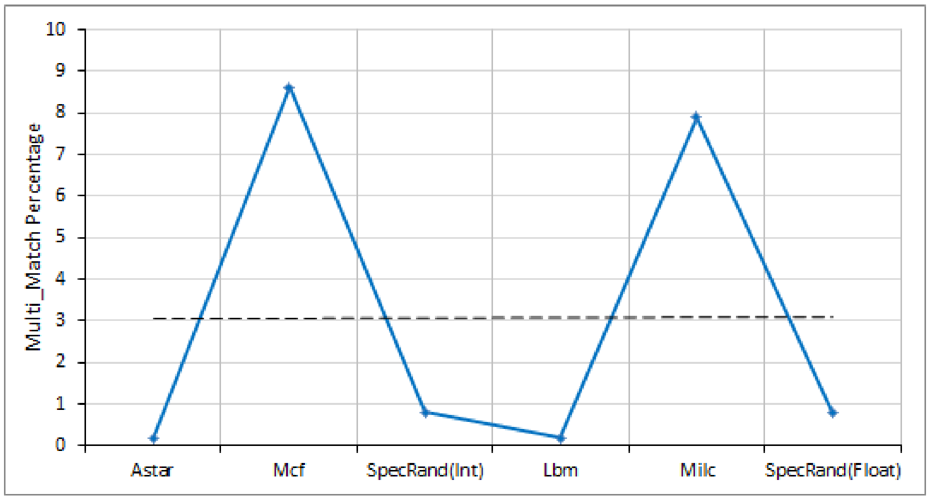


Figure 10 - *multi_match* penalty evaluation

The overhead hardware investment for generating multi_match is fairly compact, and comprises only of XOR gate arrays and and-or based logic. Signal multi_match is generated only when going from higher to lower order associativity and the data read is erroneously aliased in the four memory units with the new cache system.

It's necessary to gauge the impact of the multi_match on performance degradation. The frequency of such conflicts occurring and multi_match being raised vary depending on the type of workload being tested. Figure 10 displays the percentage of data misses generated due to *multi_match* for various benchmarks, while the cores were programmed in the *auto* mode. The average impact for integer and floating point is around 2.5% - 3%, which is quite less as compared to previous work.

3.3.1.5. Writeback

This stage will write back the data to the decode stage into the integer or floating point register-file. This stage gets data from memory stage (from Level1 Dcache or result from execute stage).

3.3.2. Hazard Unit

Hazard Unit takes care of data forwarding from memory and writeback stages to execute stage. It also takes care of data forwarding from writeback to decode stage. All this forwarding is not only managed for integer type instructions, but also for floating point instructions. This unit also generates stall signals for stages in datapath for all the exception conditions.

3.3.3. Coprocessor0

Exception unit determines the cause of exception in the normal flow of the program execution. Exception can be raised due to various reasons like external interrupts, system-call or break instruction, misaligned addresses while load/store, etc. When an exception occurs, Coprocessor0 is programmed to make the core to jump at exception routine, and then return back to the original code.

When any exception occurs, the Coprocessor0 raises a signal (activeexception), which is sent to the core. This signal will flush all the stages in the pipeline, and the core is diverted to the address where exception handler is located. After the exception handler is executed, the control is sent back to the return address.

3.4. L2 Data and Instruction Cache

Level2 Dcache is an on-chip resource, which is also shared between all four cores. This stage implements a synchronous 4-way set associative SRAM, which has a net capacity of 1024x4Bx4 (16KB). This hierarchy of memory directly fetches its contents from the main off-chip data memory.

The Level1 Dcache can be at most implemented as a 4way set-associative memory. This is why Level2 Dcache needs to be implemented at-least as 4-way set associative architecture. Level2 Icache is also an on-chip resource, which is shared between all four cores. This stage implements a synchronous direct-mapped SRAM, which has a net capacity of 1024x16B (16KB), thus storing 4 instructions (4 words) in each set of memory.

This hierarchy of memory directly fetches its contents from the main off-chip instruction memory. Owing to the bigger sizes, these memories are expected to run at clock speeds slower than the core frequency. Thus, these memory units run at slower clock speed (rclk_2). This is facilitated by the implementation of the asynchronous FIFOs in the arbiter logic.

The main data and instruction memories which are off-chip are expected to run yet slower clocks (rclk_3). Thus, the interface between the Level2 memory units and the external memories also uses asynchronous FIFOs.

3.5. Floating Point Units

This unit consists of two symmetrical pipelined FPU units. The FPU units will be enabled or disabled depending on the workload from cores. Each pipelined FPU unit as shown in Figure 11 comprises of a Radix512 mult-div unit, adder, and integer-float conversion unit. The computed results are sent back to the decode stage of the core. Each FPU unit is implemented as a three-stage pipeline, which are also buffered:

Stage1 – The operands into FPU unit come from the execute stage of the core, through the arbiter switch. OSCAR supports both single and double floating-point instructions. Thus, for simplicity, the core floating-point operation processing units like adder or mult-div units are configured to process double precision floating point numbers.

Thus, all incoming operands need to go through a unit called 'convert_inputs' which would convert all single precision floating-point numbers into double precision, and keep the double precision floating-point numbers untouched. After this, all the converted operands, along with the requested operation code must pass into a unit to check for any exceptions. This unit detects if there are any exceptions with the operands (like all-zero, NaN, +Infinity, -Infinity) or the operation type. Following this, the resultant operands and other control signals are buffered.

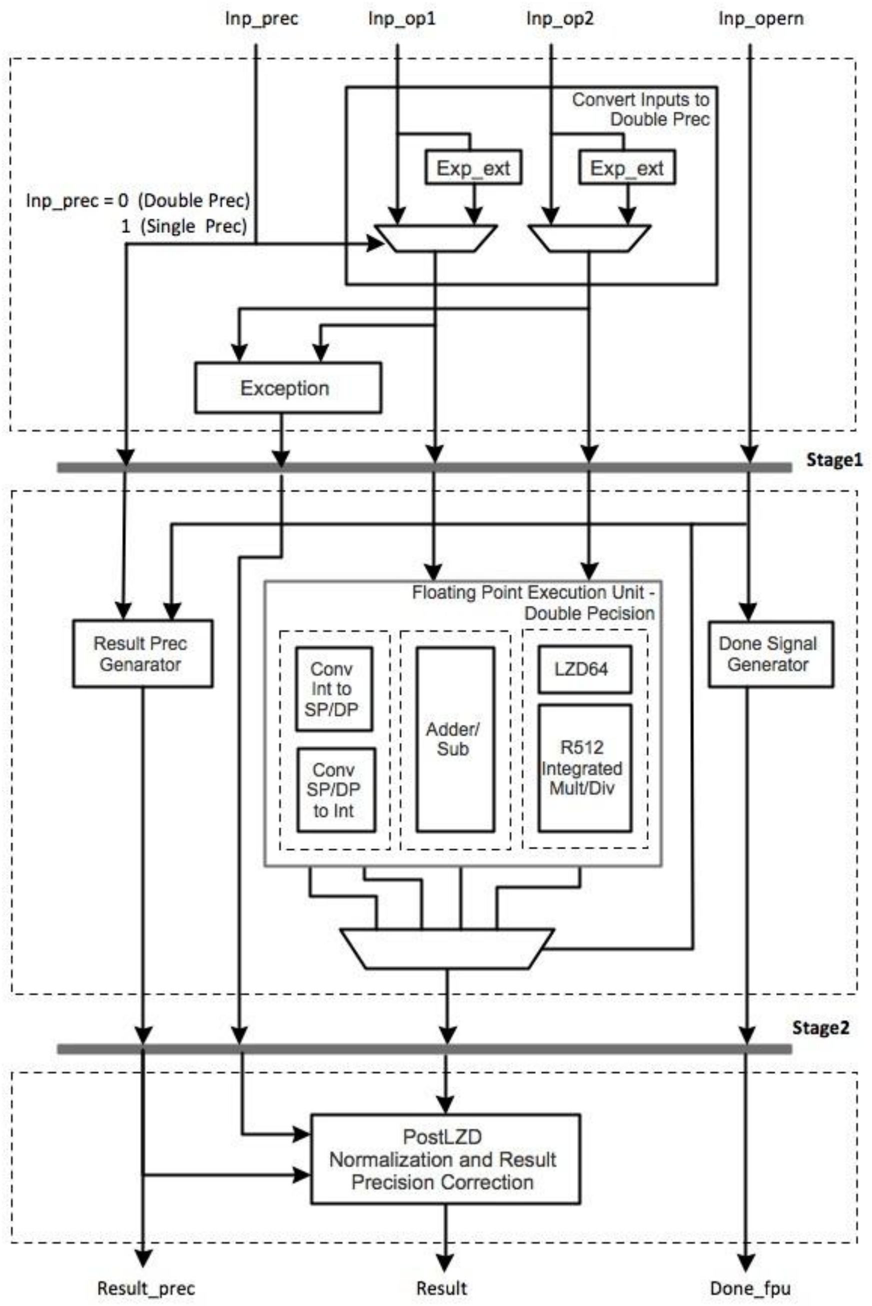


Figure 11 - Pipelined Floating Point Unit

Stage2 - Here, the operands can be passed to various units depending on the operation requested. The units can be mult/div, adder/sub, convert from single/double precision floating-point to integer, and convert from integer to single/double precision floating-point. Each of these units produces a done signal, which indicates that the result has been computed, and is ready to move to Stage3 of the pipeline.

If the desired operation is adder/sub, the operands are directly passed into the unit. The unit evaluates the sign, exponent and mantissa of the result. If the desired result operation is mult/div, then the incoming operands need to go through LZD unit, so there are no leading zeros in the mantissa of the two operands.

So, the two operands are passed through and LZD, and using the counts obtained from the LZDs, the two operands and their exponents are corrected. Thus, the output from LZD is only fed to the mult/div unit, while the inputs of the Stage2 are directly fed to the adder/sub unit.

If the required operation is unary, either convert from/to integer, the incoming operand is directly passed to either of the units. If a floating-point number (originally either single/double precision) is converted into integer, the 64-bit final result will be the 32-bit final integer appended with 32 zeros. If an integer is converted into a floating-point number (finally needed to be single/double precision), the generated result is anyway generated as a double-precision number.

The results are again buffered and passed on to the 3rd pipeline stage.

Stage3 – The final result (if a floating-point number) is again normalized. If the result is an integer, the result from Stage2 is passed on as it is. Now, when dealing with floating-point numbers in Stage2, the numbers are always processed as double-precision numbers.

If the final result after the operation is required to be single precision, the double-precision result is converted into single-precision number appended with 32 zeros. The final result from Stage3 of the pipeline is de-muxed back to the correct core's decoding stage. In the decode stage, the result is either stored in the integer/floating-point reg-file. Even though the FPU unit is located along with the Level2 cache hierarchy, it can still be easily run at the core frequency to enhance performance of the unit even more.

3.6. Arbiter Logic

The arbiter logic (Figure 12) is the crossbar logic, which connects four cores to the shared resources on Level2 hierarchy like Level2 Dcache, Level2 Icache, and FPU units. Each core is connected to the arbiter logic through asynchronous FIFO.

Bus Name	**Width**	**Source**	**Destination**	**Description**
imem_l2_cu_cx	[28:0]	Core	Arbiter	Instruction request from Core-CX to arbiter
imem_l2_uc_cx	[128:0]	Arbiter	Core	Instruction (4words) sent back to Core_CX
fifoout_imem_l2	[30:0]	Arbiter	L2 Icache	Request from selected core sent to L2 Icache
instrout_imem_l2	[127:0]	L2 Icache	Arbiter	Instruction (4words) sent from L2 Icache

Table 2 - Arbiter Bus - [Level1_Icache - Arbiter - Level2_Icache]

Bus Name	**Width**	**Source**	**Destination**	**Description**
dmem_l2_cu_cx	[63:0]	Core	Arbiter	Data request from Core-CX to arbiter
dmem_l2_uc_cx	[32:0]	Arbiter	Core	Data (1word) sent back to Core_CX
fifoout_dmem_l2	[65:0]	Arbiter	L2 Dcache	Request from selected core sent to L2 Dcache
rd_dmem_l2	[31:0]	L2 Dcache	Arbiter	Data (1words) sent from L2 Dcache

Table 3 - Arbiter Bus - [Level1_Dcache - Arbiter - Level2_Dcache]

Bus Name	**Width**	**Source**	**Destination**	**Description**
fifo_bus_cu_cx	[140:0]	Core	Arbiter	FPU request from Core_CX to arbiter
fifo_bus_uc_cx	[71:0]	Arbiter	Core	Result sent back to Core_CX
fpu_bus_inx	[143:0]	Arbiter	FPU Unit	Request from selected core sent to FPU unit
fpu_bus_resx	[74:0]	FPU Unit	Arbiter	Result sent from FPU Unit

Table 4 - Arbiter Bus - [Core_CX - Arbiter - Floating Point Unit]

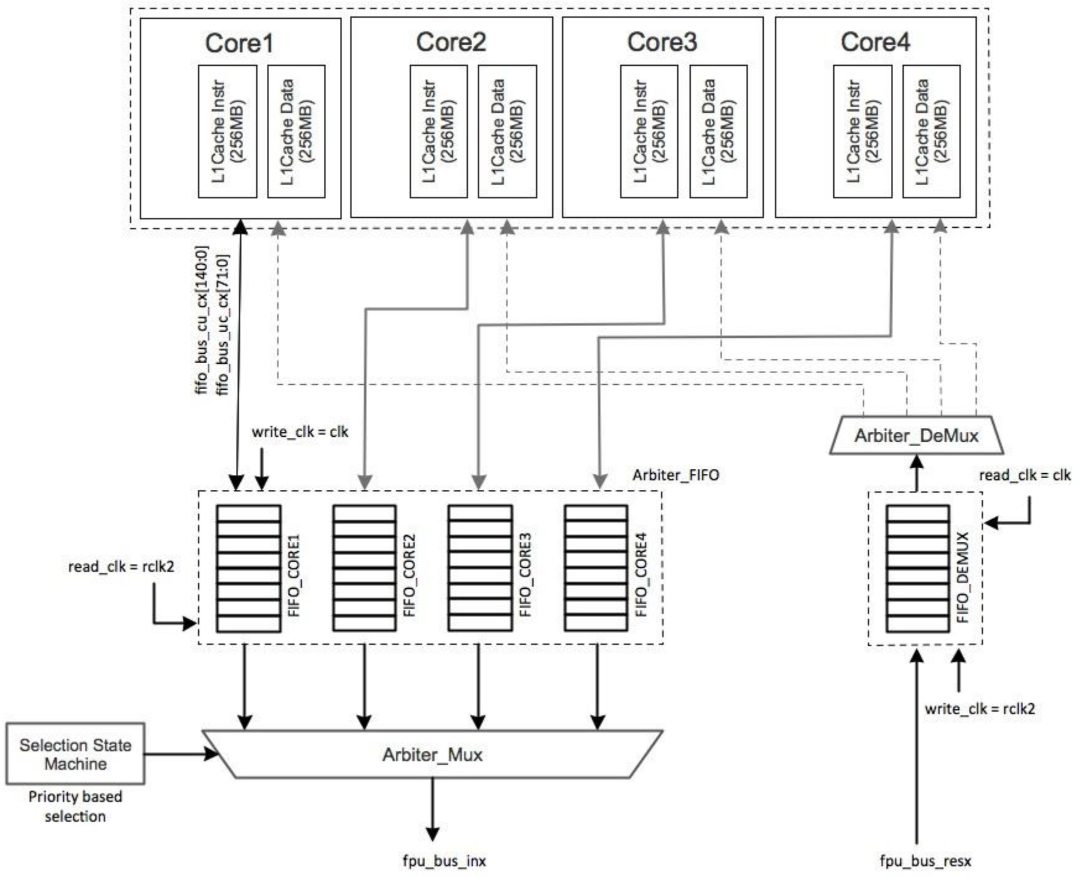


Figure 12 - Arbiter Logic

There are three sections in the arbiter logic – for Level2 Dcache, Level2 Icache and FPU units. Each core has three FIFO units connected to each of these three sections. Thus, for four cores, there are total 12 asynchronous FIFOs in the arbiter logic.

For each of the sections, the control is routed to the cores on first-come-first-served basis. In case when there are multiple cores with pending requests, the control is rotated between them in round-robin fashion.

The key feature of the arbiter is the implementation of asynchronous FIFOs, which allow reading and writing at different clock rates. Thus, while the core is running at maximum frequency (say 500MHz), Level2 hardware (Level2 Dcache, Icache) can run at slower speeds. The busses that flow in and out of arbiter are shown in Table 2, Table 3 and Table 4.

3.7. Asynchronous FIFO

The multi-core environment deploys three on-chip clocks – fastest for the core, slower for the Level2 memories, and floating point unit, and the slowest for the interface to the external memory module. Thus, when moving between levels, the data should be synchronized on the destination clock. Three kinds of issues occur with synchronization – metastability, data loss and data incoherency.

One way to do this is by using *multi flip-flop synchronizers*, which help with the metastability issue and work great for synchronizing fewer bits. Although, when multi flip-flop synchronizers are used with data busses, it might result in data loss due to probability of unequal transition times of the bits on the bus. Thus, a second choice of *recirculative MUX-based synchronizers* can be used for busses. In this system, a data enable bit is added to the bus, and this bit is allowed to pass through a multi flip-flop synchronizer. If and when this enable bit is latched on the destination clock, the remaining bus is allowed to be passed through.

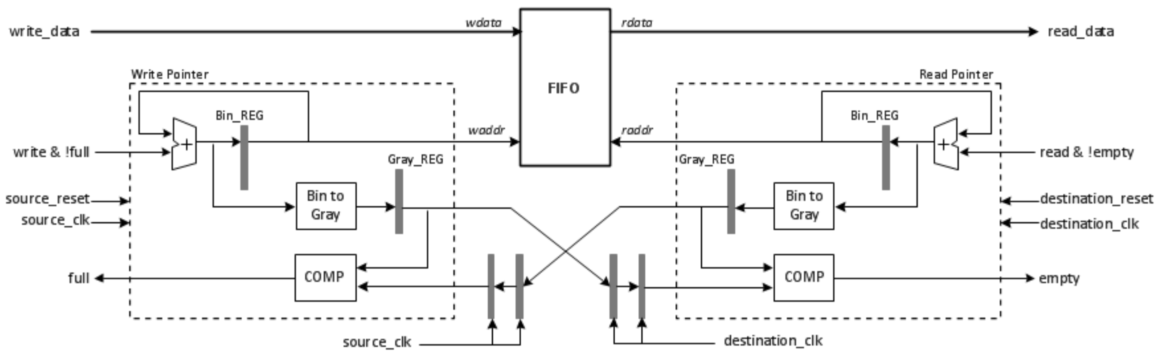


Figure 13 - Asynchronous FIFO with Gray code read/write pointers

Although, these methods are quite robust, they are effective only when the source and the destination clocks are derived from the same root clock. An asynchronous FIFO is rather more robust with asynchronous clocks, or clocks which are not derived from the same root clock.

Also, using gray codes for busses improves the performance of the synchronization circuit, by avoiding false intermediate stages during the synchronization of multiple bits across clock domain. Now, some control signals can be realistically converted as gray codes for the purpose of synchronization, but this cannot be done for all busses.

A simple solution that incorporates the use of asynchronous FIFO and gray codes is to keep the data busses as it is, but implement the read and write pointers using gray codes. Some articles suggest pros and cons of some implementation methods for FIFO based synchronizers. Analysis suggests that read and write pointers directly implemented using gray codes have certain difficulties [14].

Thus, original pointers in the source clock domain FIFO are implemented and maintained in binary, and converted into gray codes, just to facilitate the cross domain synchronization. The gray code pointers are then passed through flip-flops clocked to destination clock, which accomplishes one-way synchronization.

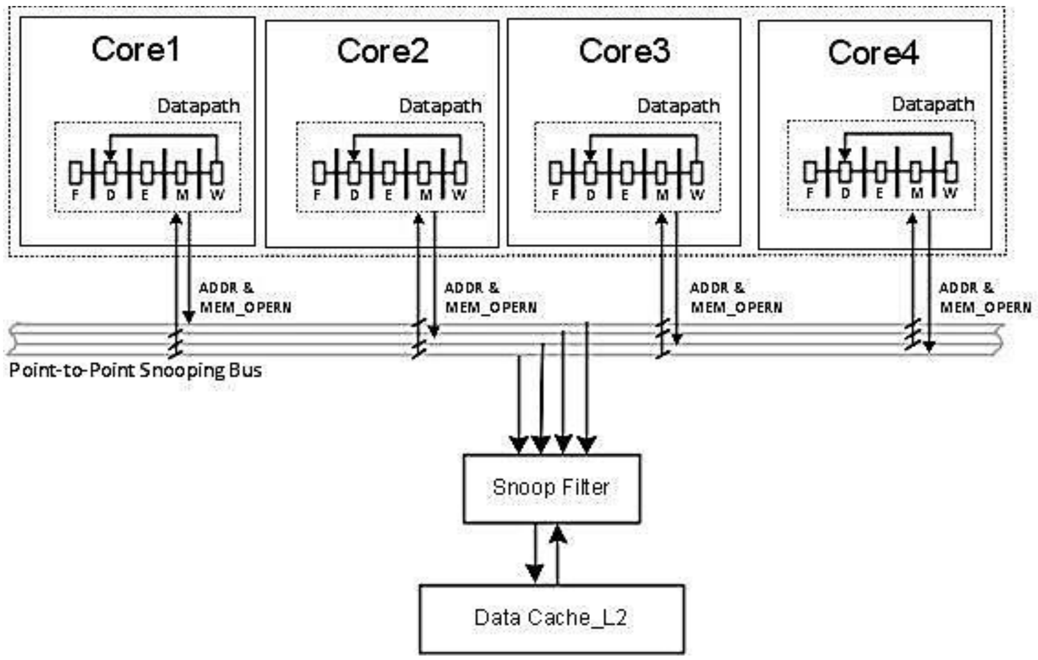


Figure 14 - Snooping based cache coherency

Like-wise, gray code pointers from the destination clock domain FIFO are registered at source clock to the source clock domain. These synchronized pointers are compared against the source clock domain pointers to evaluate the full and empty conditions of the FIFO. Figure 13 illustrates the block diagram of the implementation. The sequences of operations for synchronization are as detailed as below:

1. FIFO Register	wp_ns	→	wp_ps	@ wclk
	rp_ns	→	rp_ps	@ rclk
2. Gray conversion	wp_ns_gray	=	GRAY(wp_ns)	… (wp_ns >> 1) ^ wp_ns
	rp_ns_gray	=	GRAY(rp_ns)	
3. Gray Registers	wp_ns_gray	→	wp_ps_gray	@ wclk
	rp_ns_gray	→	rp_ps_gray	@ rclk
4. Syncing	wp_ps_gray	→	wp_sync	@ rclk
	rp_ps_gray	→	rp_sync	@ wclk
5. FULL/EMPTY	FULL	→	funct(wp_ns_gray, rp_sync)	
	EMPTY	→	funct(rp_ns_gray, wp_sync)	

3.8. Data Coherency

One of the dominant aspects of any multi-core architecture is cache coherency across multiple cores. Without cache coherency the cores will be unable to share common memory spaces when executing applications. Over times, three kinds of mechanisms have been developed for data coherency in multi-processor architectures; they are called directory-based coherence, snooping and snarfing. There are pros and cons to all the mechanisms.

3.8.1. Snooping

This mechanism (Figure 14) is implemented inside each core, and functions in the memory stage of the pipeline. With the snooping mechanism, each core monitors the address lines of other cores, for cache writing operations (store instructions). The core looks out for store operation being performed by other cores on locations that were already previously fetched.

When the core finds such an operation being performed, the core simply invalidates the entry in its own cache, and then continues the normal operation.

This directly implies that this mechanism is quite fast, but requires high bandwidth, since all memory operations need to be broadcasted to all other cores. So, as the number of cores increase, this local traffic also scales up, and will reflect as a significant impact on the cores' metrics.

3.8.2. Directory-based

The directory structure (Figure 15) functions along-side with Level2 Dcache, and it maintains a record of all entries, which are loaded into Level1 Dcache of all four cores. When a core makes changes to a memory location, which the other cores had also fetched previously, directory structure invalidates the entries for these other cores.

This indicates that this mechanism reduces the local traffic by quite an extent. Also, the mechanism is not as complex as snooping to implement, since all the busses run from the cores to the directory structure logic. This is why, directly structure protocol is more widely used for complex and large multi-core systems.

3.8.3. Snarfing

Snarfing is the least preferred choices for implementing cache coherency. This protocol is very similar to snooping. Although, here, instead of just snooping out for addresses, the cores also check for data changes, and then update its outdated copy if needed.

This makes snarfing the worst choice, because of the heavy local traffic (which was also inherent with snooping), but also because of the additional data bus running point-to-point between all cores.

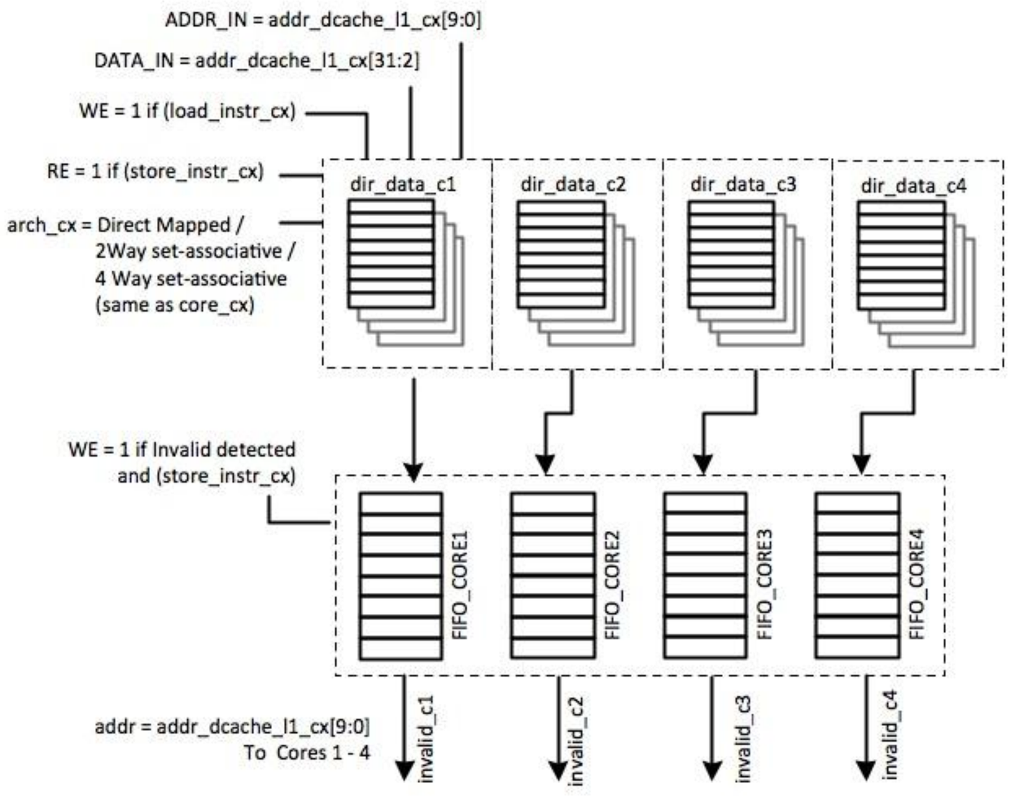

ADDR_IN = addr_dcache_l1_cx[9:0]
DATA_IN = addr_dcache_l1_cx[31:2]
WE = 1 if (load_instr_cx)
RE = 1 if (store_instr_cx)
arch_cx = Direct Mapped / 2Way set-associative / 4 Way set-associative (same as core_cx)
dir_data_c1
dir_data_c2
dir_data_c3
dir_data_c4
WE = 1 if Invalid detected and (store_instr_cx)
FIFO_CORE1
FIFO_CORE2
FIFO_CORE3
FIFO_CORE4
addr = addr_dcache_l1_cx[9:0] To Cores 1 - 4
invalid_c1
invalid_c2
invalid_c3
invalid_c4

Figure 15 - Directory Structure

Directory structure ensures data coherency in multi-core environment when data space is shared across cores. The directory consists of a structure with 4 memory blocks, each one of which keeps a track of the functioning of L1 Dcache of its respective core. Thus, each one of the four memory units is just as big as L1 Dcache of any core. The outputs of this memory structure are connected to 4 FIFOs, which are connected to the four cores. Functioning of the directory structure can be explained by the following example:

Given condition: L1 Dcache is structured as a 4KB direct mapped cache (1024x4B). Thus, the 1st memory block of the directory structure is of the same size and organization (direct mapped).

a. Now, say Core1 has a LW instruction for address 0x00001234. Data gets fetched from main memory into Level2 (if necessary) and finally from Level2 to Level1. While the data is being fetched from Level2, an entry is made in 1st memory block at address 0x08D (0x00001234 >> 2 for word offset = 0x0000048D).

b. Now, after a couple of cycles, Core2 has a SW instruction for the same address, implying that the address space is shared between the two cores. For a core, SW instruction is implemented in a write-through policy. Thus, when the store instruction comes upto the Level2 Dcache, it also makes a check in the directory memory blocks of other cores (1st, 3rd and 4th) to see if some other core has a copy of the data, which should be updated now.

c. Thus, when it understands that Core1 had fetched the data from the same address, it generates an invalidate signal for the address at Core1's Level1 Dcache. This request for invalidation is stored in a FIFO for Core1. When this request in the FIFO is serviced, it will stall all stages in the pipeline, and also invalidate the memory location in Level1 Dcache of Core1.

If multiple cores has fetched the same data, then there would be invalidates generated for multiple cores at once. Now, as described earlier, Level1 Dcache can be configured to be a direct mapped, 2way or 4way. Also, the memory block in directory structure must be an exact replication of the Level1 Dcache configuration.

Thus, when dynamically the Level1 Dcache changes its architecture from say, direct mapped to 2way set associative, the directory structure also must change its architecture likewise. Also, change of Level1 Dcache memory architecture for all four cores happens independently. Thus, the architecture changes in the four directory structure memory blocks also happen independently of each other.

The directory structure is also implemented as memory blocks similar to Level1 and Level2 caches. Thus, owing to its huge size, the directory structure is also facilitated to run at slower clock speeds as compared to core frequency

CHAPTER IV

TOOL FLOW AND DESIGN METHODOLOGY

Besides designing the complete architecture, the other challenge during the project was setting up tools to do design testing and hierarchical design planning for synthesis and floorplanning. The tool flow for the project as illustrated in Figure 16 was carried out in 4 major steps: testing and debugging, design synthesis, floorplanning, and final timing and power analysis. ModelSim was used for simulation and debugging. Design Compiler, Design Vision, IC Compiler and Primetime were used from the Synopsys tool-chain for synthesis, floor planning, and timing and power analysis. The standard cell library used for the project was IBM65 low power library provided by Virage Logic. The following flowchart explains the steps for the complete tool flow.

4.1. ModelSim

The highlighting aspect about the project is the ability of the front-end user to analyze the workload at every clock and dynamically configure some design units, so that the architecture delivers better performance in terms of either power or speed.

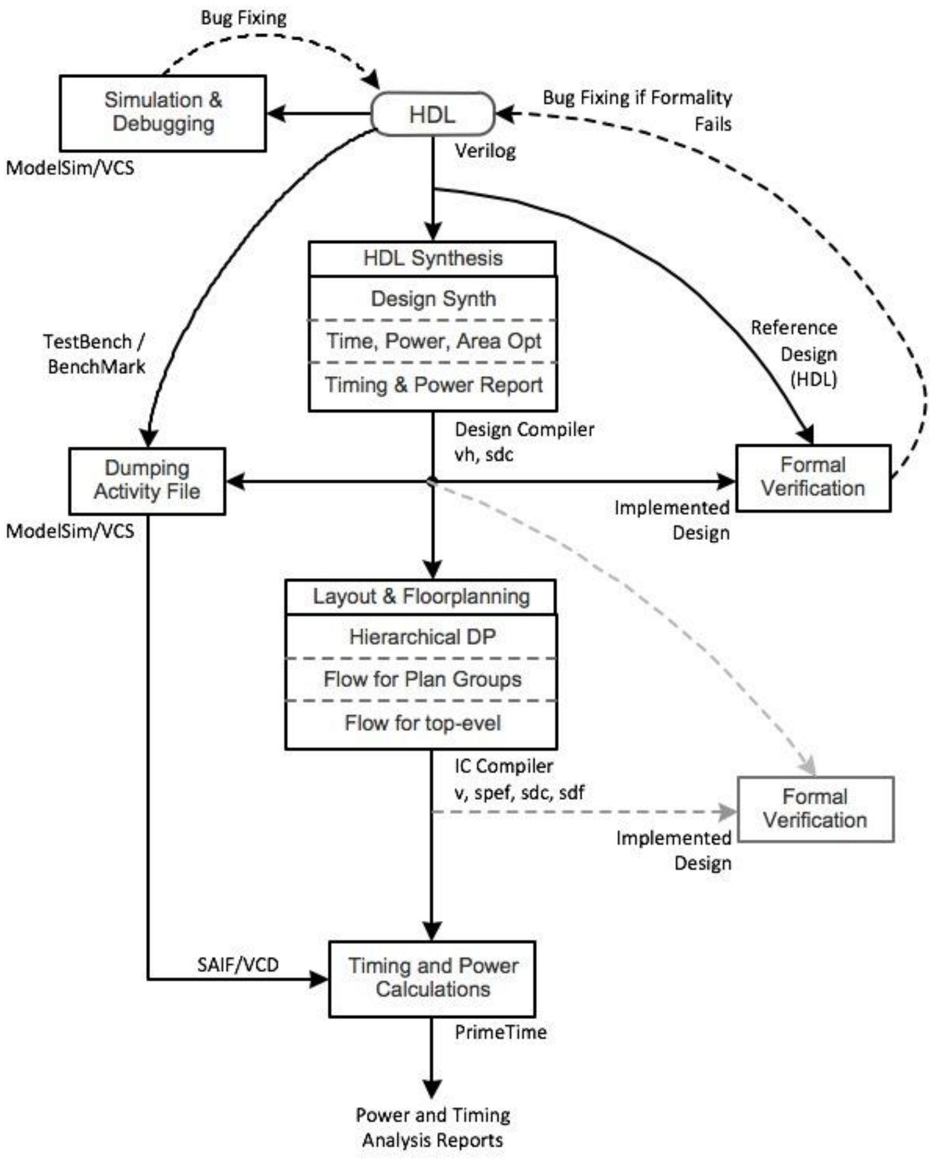
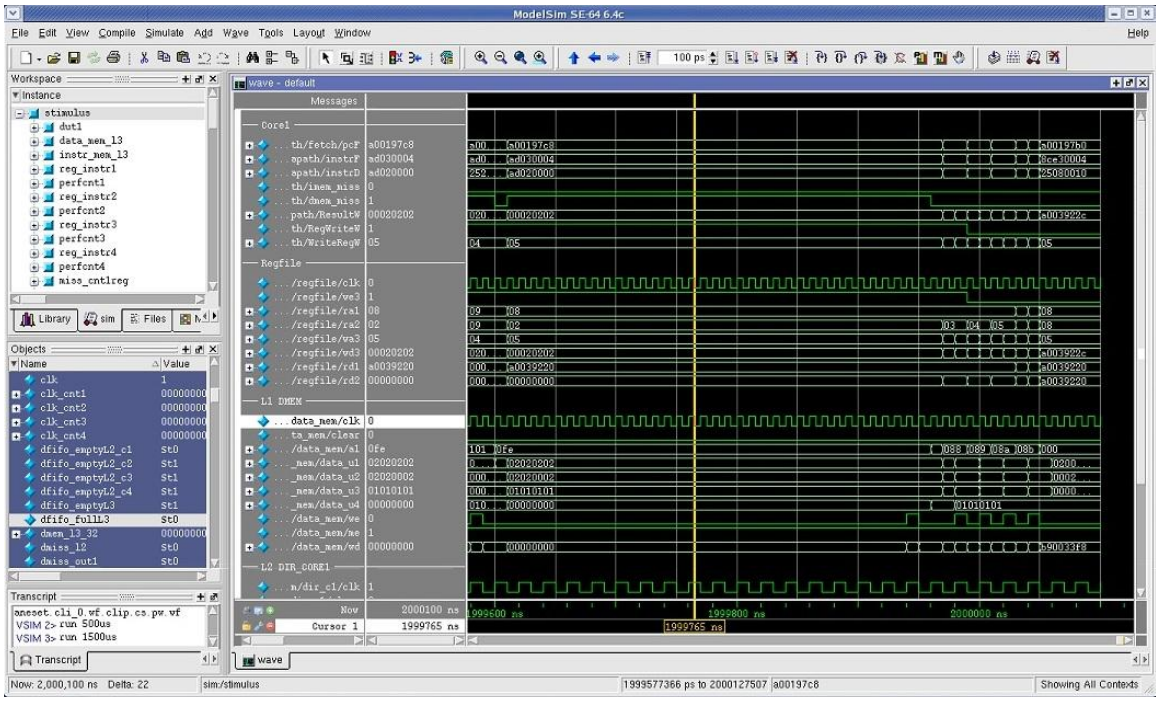


Figure 16 - Tool Flow Methodology

Many other similar hardware architecture models that have been designed and published have been written on platforms, which make use of high-level languages This may help prove the validity and feasibility of the proposed model; however, the accuracy of the obtained metrics and implementation techniques can be vague. This leads us to the need of a design platform, which should be easy to adapt to, should give the user good control over the design, but should also be realistic to the fabrication technologies. The complete designing of the project was done using RTL, which makes the hardware design very realistic.



```
stimulus
  dut1
  data_mem_l3
  instr_mem_l3
  reg_instr1
  perfcnt1
  reg_instr2
  perfcnt2
  reg_instr3
  perfcnt3
  reg_instr4
  perfcnt4
  miss_cntlreg
```

```
Name                 Value
clk                  1
clk_cnt1             00000000
clk_cnt2             00000000
clk_cnt3             00000000
clk_cnt4             00000000
dfifo_emptyL2_c1     St0
dfifo_emptyL2_c2     St1
dfifo_emptyL2_c3     St1
dfifo_emptyL2_c4     St1
dfifo_emptyL3        St1
dfifo_fullL3         St0
dmem_l3_32           00000000
dmiss_l2             St0
dmiss_out1           St0
```

```
Core1
...th/fetch/pcF      a00197c8
...apath/instrF      ad030004
...apath/instrD      ad020000
...th/imem_miss      0
...th/dmem_miss      1
...path/ResultW      00020202
...th/RegWriteW      1
...th/WriteRegW      05
Regfile
.../regfile/clk      0
.../regfile/we3      1
.../regfile/ra1      08
.../regfile/ra2      02
.../regfile/wa3      05
.../regfile/wd3      00020202
.../regfile/rd1      a0039220
.../regfile/rd2      00000000
L1 DMEM
...data_mem/clk      0
...ta_mem/clear      0
.../data_mem/a1      0fe
..._mem/data_u1      02020202
..._mem/data_u2      02020002
..._mem/data_u3      01010101
..._mem/data_u4      00000000
.../data_mem/we      0
.../data_mem/me      1
.../data_mem/wd      00000000
L2 DIR_CORE1
...m/dir_c1/clk      1
Now                  2000100 ns
Cursor 1             1999765 ns
```

```
aneset.cli_0.wf.clip.cs.pw.vf
VSIM 2> run 500us
VSIM 3> run 1500us
```

```
Now: 2,000,100 ns   Delta: 22   sim:/stimulus   1999577366 ps to 2000127507   a00197c8   Showing All Contexts
```

Figure 17 - ModelSim Simulation of RTL

ModelSim [14], which is a cycle-based simulator, has been used for simulation and testing of RTL Verilog code. There are several provisions made in the testbench for the user's benefit for debugging, and gathering run-time test results.

The dumped test results for each core contain information about instruction count, cycle count, information about the architecture configurations, and numbers which indicate nature of incoming workloads. These statistics are recorded in every '*instruction window*'. When working with benchmarks, these numbers prove to be very useful to plot graphs for analyzing the architecture with changing workloads.

As mentioned earlier, ModelSim proves very useful when simulating RTL code of the design. Although, to completely verify the functioning of the design, it is also be important to verify that the post-synthesis netlist with timing information also simulates without errors. Figure 17 shows a sample simulation on OSCAR.

4.1.1. Unit testing of module

Validating the behavioral functioning of the various units is undoubtedly very important before testing the complete design with heavy benchmark simulations. All the units were individually tested with manually created test codes.

Some codes from the David Harris's test-codes [15] set were also used to validate functioning of the design units. Some other test routines were also designed to test the cache coherency with directory implementation. After several iterations of locating and resolving issues, the code was tested with SPEC benchmarks.

4.1.2. Benchmark Testing

SPEC-CPU suites have benchmarks that can be used to test the CPU performance with applications, which quite resemble the present day workloads for a modern computer system. Thus, many researchers often use SPEC-CPU benchmark suites to evaluate their ideas and results. The project makes used SPEC-CPU2000 and SPEC-CPU2006 benchmarks for performance analysis.

4.1.3. Challenge with Benchmark Testing

Although, when working with benchmarks, it is very important to realize that even the most moderately sized benchmarks usually will have nothing less than 50billion instruction. The benchmarks were initially simulated using the synthesized netlist (which was needed to dump VCD - switching activity file). Although, ModelSim was very slow with the synthesized netlist, and it took the tool around 4-5 hours to run about 1us worth of simulation run-time. Also, besides the run-time, the other problem in keeping up with this technique was the huge space consumed by the VCD files. The estimated memory space needed for a VCD dump was 1GB for every 10us worth of simulation run-time

4.1.4. Solution

This is why the benchmarks need to be run on the RTL code directly, which makes the simulations on ModelSim run reasonably faster. A simulation run-time of 1ms usually will take around 1 hour in real-time. Thus, keeping in mind the size of the design, and the benchmarks, simulations were run with the RTL code, and not on the post-synthesis netlist.

4.1.5. Script and Setup

The ModelSim script file (.do) used for the project has been added in Appendix. The instructions for setting the environment and running the script is also given in the Appendix.

4.2. Design Compiler

The hierarchical design planning begins at the synthesis level. Design Compiler [16] and Design Vision from Synopsys were used for design synthesis, visually inspecting path groups, critical paths, and estimating timing and power numbers. The basic framework and file structure of the scripts can be downloaded from the Synopsys website.

4.2.1. Design Compiler flow

During synthesis, the RTL instantiations are broken down and mapped to the GTEH and DesignWare library components (if needed); the symbols are used extracted from the symbol library. If required, the design schematic can be reviewed using Design Vision to view the path groups for timing calculations, as in Figure 18. After the schematic review, the logic is optimized and the GTECH modules are mapped to technology library components. After the optimizations are done, the tool dumps the final synthesized netlist.

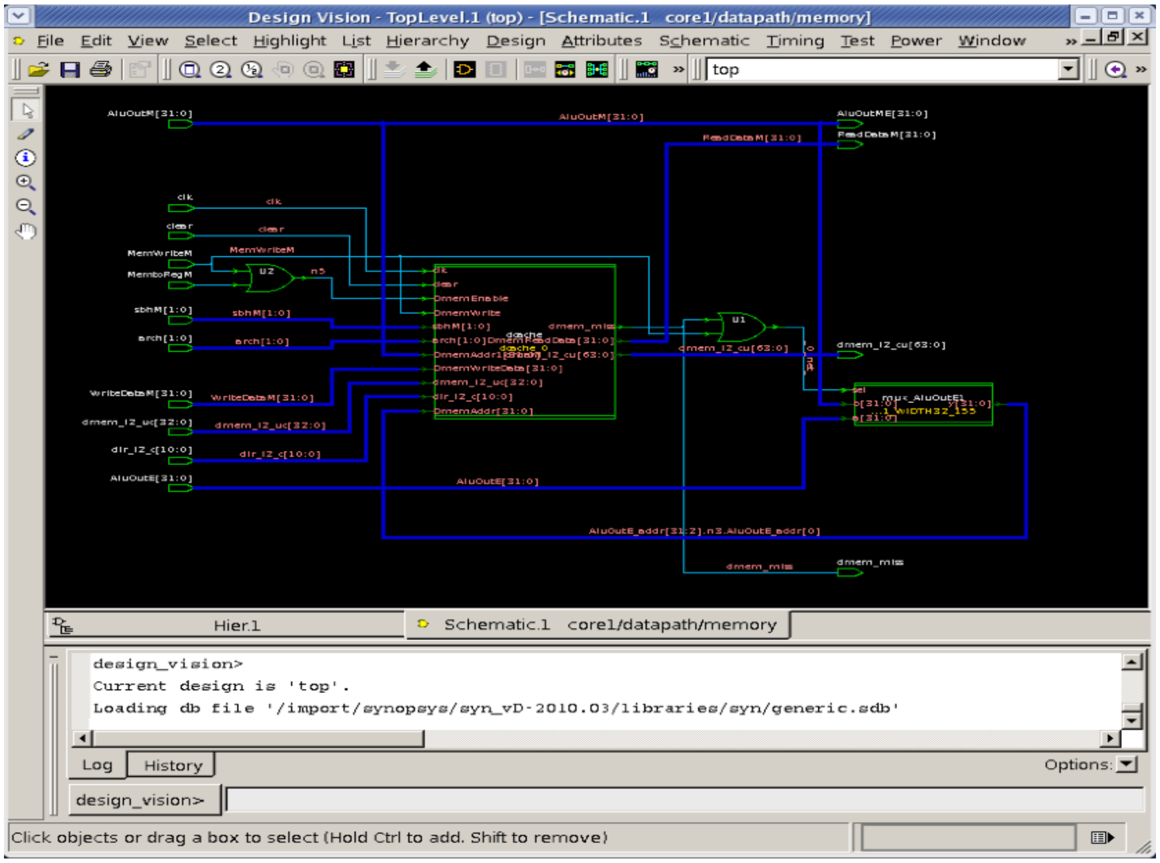
Besides the netlist, the other output files dumped by the tool are SDC (Synopsys Design Constraint), SDF (Standard Delay Format), DDC (Design Database), preliminary area, timing, and power reports. Commands for dumping all these formats, can be found in the Design Compiler scripts in the Appendix.

4.2.2. Design Compiler file formats

The *synthesized netlist* (.vh) generated by the tool is highly influenced by the choice of switches and other related options with the compile commands – *compile* and *compile_ultra*. The second command is very similar to the 1st one, except for some very few changes, which make it a push-button solution for timing-critical, high-performance designs. The second command does automatic ungrouping and boundary optimization across modules.

This can be particularly useful when synthesizing moderately sized designs. However, the user must be particularly careful when applying these switches directly to large designs. Command *compile_ultra* was used for this project. Also, it’s always preferable that the netlist is dumped in hierarchical manner. The commands for this are also mentioned in the scripts in the Appendix.

The *SDC (Synopsys Design Constraint)* is a Synopsys file format that is used to convey constraint information set by the user in Design Compiler to other compatible Synopsys tools (IC Compiler and PrimeTime). The user can set constraint information related to operating conditions, timing, power and area. Some of the design constraints are technology dependent and are directly taken from the technology library. Some examples include - set_max_capacitance, set_min_capacitance, set_max_fanout, etc. Other constraints can be customized by the user, e.g.: create_clock, set_input_delay, set_output_delay, set_max_delay, etc. Understanding and setting these constraints with Design Compiler is very important, and can be used to keep a check on the design.


Design Vision - TopLevel.1 (top) - [Schematic.1 core1/datapath/memory]
File Edit View Select Highlight List Hierarchy Design Attributes Schematic Timing Test Power Window
top
AluOutM[31:0]
clk
clear
MemWriteM
MemtoRegM
sbhM[1:0]
arch[1:0]
WriteDataM[31:0]
dmem_l2_uc[32:0]
dir_l2_c[10:0]
AluOutE[31:0]
ReadDataM[31:0]
AluOutME[31:0]
dmem_l2_cu[63:0]
dmem_miss
u2
n5
u1
dcache
mux_AluOutE1
WIDTH32_155
AluOutE_addr[31:2].n3.AluOutE_addr[0]
Hier.1
Schematic.1 core1/datapath/memory
Log
History
Options:
design_vision>
Click objects or drag a box to select (Hold Ctrl to add. Shift to remove)

```
design_vision>
Current design is 'top'.
Loading db file '/import/synopsys/syn_vD-2010.03/libraries/syn/generic.sdb'
```

Figure 18 - Design Vision showing the synthesized memory stage from datapath

SDF (Standard Delay Format) is a widely used by EDA tools to pass delay information between multiple tools for static timing analysis. An initial SDF for the synthesized netlist can be generated using Design Compiler, which can then be used with IC Compiler for timing driven placement. Although, if there are timing violations with the layout, the SDF file can be back annotated from the IC Compiler to DC Compiler to revise design synthesis and re-analyze worst case paths.

DDC (Design Database) is an internal Synopsys format, which can be used to view the post-synthesis design graphically using Design Vision. DDC file written from Design Compiler saves all design objects from the design, net names, and design constraints.

4.2.3. Script and Setup

Design Compiler needs inputs in the form of HDL code (Verilog). Along with the RTL input, the tool needs the technology library, symbol library and DesignWare library from Synopsys. The symbol and DesignWare libraries are needed if the design should be inspected graphically using Design Vision. The scripts for Design Compiler along with the instructions have been added in Appendix.

Before starting to work with the tool, it's very important that a compile strategy should be decided for a given project. In a top-down strategy, the top-level design and all its modules are compiled all together. Usually the top-down approach is preferred for moderately sized design.

With bottom-up strategy, sub-modules are constrained and synthesized individually. After this, these sub-modules are tagged as *'dont_touch'* when finally the top-level module is being synthesized. While the bottom-up strategy might seem to be quite time-consuming, it's usually preferred for large designs.

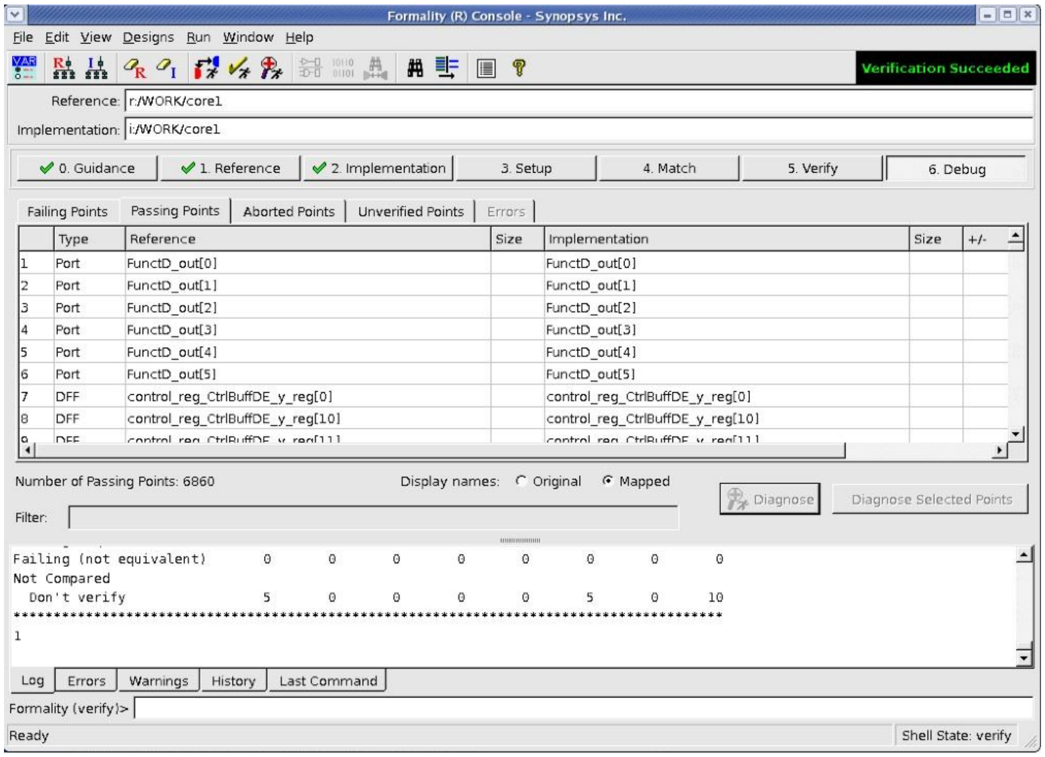
4.3. Formal Verification

When working with Design Compiler, depending on the switches, the tool might try to optimize some logic within modules, and also perform some optimization across hierarchical boundaries. When working with timing-critical designs that are relatively moderate in size, the user can be assured of the quality and validity of the synthesis done by the tool.

This implies that even after optimizations, the synthesized netlist does not behave differently than actual RTL. To evaluate this, rigorous simulation tests need to be performed on the synthesized netlist with huge input vectors.

4.3.1. Need for Formal Verification

For average sized designs, running post-synthesis simulations on cycle-based simulators such as ModelSim/VCS can also validate the same. This kind of testing becomes very tedious, time consuming and creates bottlenecks with large design, such as this project.



Reference: r:/WORK/core1
Implementation: i:/WORK/core1

	Type	Reference	Size	Implementation	Size	+/-
1	Port	FunctD_out[0]		FunctD_out[0]		
2	Port	FunctD_out[1]		FunctD_out[1]		
3	Port	FunctD_out[2]		FunctD_out[2]		
4	Port	FunctD_out[3]		FunctD_out[3]		
5	Port	FunctD_out[4]		FunctD_out[4]		
6	Port	FunctD_out[5]		FunctD_out[5]		
7	DFF	control_reg_CtrlBuffDE_y_reg[0]		control_reg_CtrlBuffDE_y_reg[0]		
8	DFF	control_reg_CtrlBuffDE_y_reg[10]		control_reg_CtrlBuffDE_y_reg[10]		
9	DFF	control_reg_CtrlBuffDE_y_reg[11]		control_reg_CtrlBuffDE_y_reg[11]		

Number of Passing Points: 6860
Display names: Original [✓ Mapped]
Filter:

```
Failing (not equivalent)       0       0       0       0       0       0       0       0
Not Compared
  Don't verify                 5       0       0       0       0       5       0      10
*****************************************************************************************
1
```

Formality (verify)>
Ready
Shell State: verify

Figure 19 - Formality – Successful Verification with post-synthesis netlist for Core1

These bottlenecks are usually cased due to heavy memory requirements of the simulation tool, which directly makes the process sluggish. A better way to do the same verification is to use a Synopsys tool called Formality [17]. The inputs needed for the tool are the original RTL code, the synthesized netlist, and technology libraries for mapping the two netlists. The advantage of using the tool is that it generates detailed reports for all the differences between the two netlists.

Even though the tool does not need any input vectors like the conventional cycle-based simulators, the results obtained from Formality can be claimed to be dependable. This implies only when rigorous testing has been performed on the original RTL code using simulators like ModelSim/VCS.

4.3.2. Script and Setup

Formality is a very easy and simple yet very resourceful tool, which can help save huge amount of reworks.

Knowing that the design is huge in size, this is especially important in the early design stages, since all the following stages are highly time-consuming. With Formality, the original RTL Verilog is called as the *reference design* and the synthesized netlist (in this case coming from the Design Compiler) is called as the *implemented design*. Along with designs, the technology libraries used for the synthesis also need to be imported. Following this, the two netlists are compared for the *match points*.

Once the match points are set, the tool is also set to run the verification. Once the verification is run, the tool tries to equate all the logic equivalence between the reference and the implemented netlists. After this, the reports of the verification stage can provide an insight into the differences if any.

It is important to know that the implemented design for Formality can also be taken from the netlist after the floorplanning is complete, that is the netlist dumped by IC Compiler. This netlist of course should be dumped without the physical only cells (such as tie-cells, internal diodes, etc.). The script and instructions for the tool have been given in the Appendix.

4.4. IC Compiler

IC Compiler [18] takes inputs from the Design Compiler that basically are the synthesized netlist, and preliminary timing information. The floor planning for large hierarchical designs can be a very time-consuming task.

4.4.1. IC Compiler flow

Design planning is the first and most important task in hierarchical floorplanning, especially when dealing with large designs. When doing design planning for hierarchical designs, it’s best to start with the hierarchical flow scripts available on the Synopsys website, as a basic framework. The scripts should definitely be customized to suit the project requirements.

4.4.1.1. Need for Hierarchical design methodology

The procedure starts with the identification of the potential hierarchies inside top-level module that can be converted into design plan-groups. Each of these plan-groups or sub-blocks is processed separately and will later be treated as soft macros inside the main top-level module. Thus, in the design flow, complete PnR routine takes place sequentially for all the sub-modules, and then finally for the top-level.

The hierarchical design planning serves multiple goals. One of the important advantages is that, the tool does not have to work with the complete design in memory. Thus, even if the number of PnR iterations increase with the number of design plan-groups, the process becomes inherently faster. Also, using hierarchical methodology allows the user to make late minor changes to just one of the plan-groups. Thus, again, instead of running IC Compiler for the complete design, just the particular sub-module can be update in the design.

4.4.1.2. Design flow for hierarchical design planning

The process is divided into three phases: design planning and committing plan-groups, running IC Compiler flow on all the plan-groups, and finally running IC Compiler flow on the top-level module. The user can consider some factors while deciding on plan-groups. Plan groups can be made depending on the functionality of the modules, e.g.: in case of the multi-core project, it would make a lot of sense to prepare plan-groups for the cores, the directory structure, and the FPU units. Another factor could be a module, which is likely to undergo design changes.

4.4.1.2.1. Phase 1 - Design planning and commit Plan-groups

After the potential plan groups have been decided and defined by the user in the script (common_setup.tcl), the floorplan needs to be initialized first with power planning for the chip. Figure12-15 shows some snapshots from Phase1.

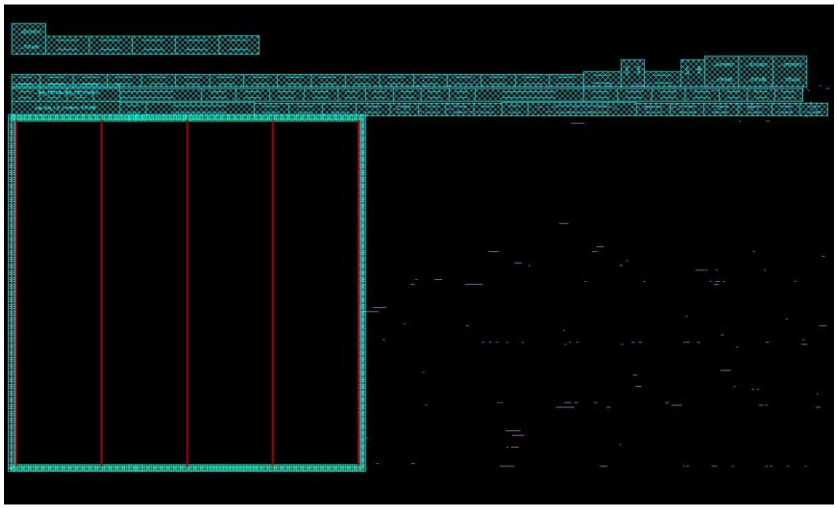


Figure 20 - Initialize floorplan for complete design

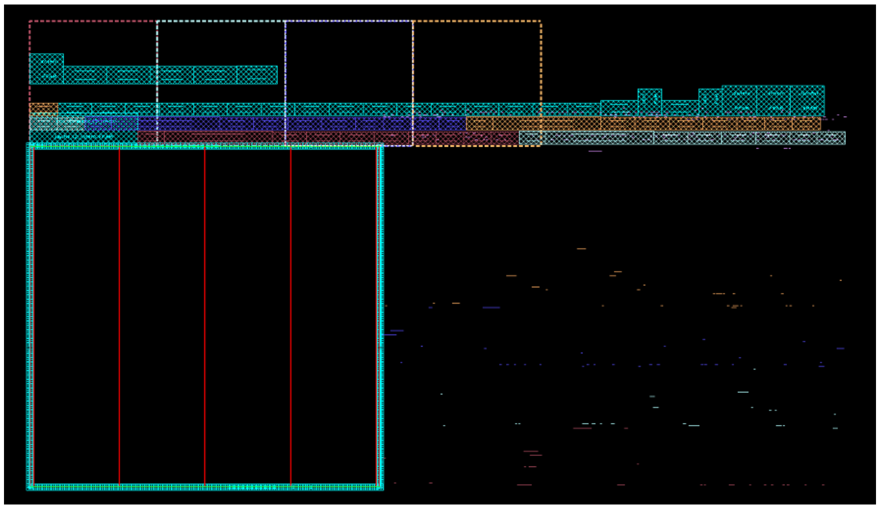


Figure 21 – Creating plan-groups for hierarchical sub-modules

After this, tentatively sized plan groups are made as per requirements. These plan-groups are then placed inside the floorplan along with their hard-macros (memory blocks), if any. The tool also needs to check the routability within the plan-group. If the design and route congestion is found to be within limits, pin assignments can then be made on the outer edge of the plan groups.

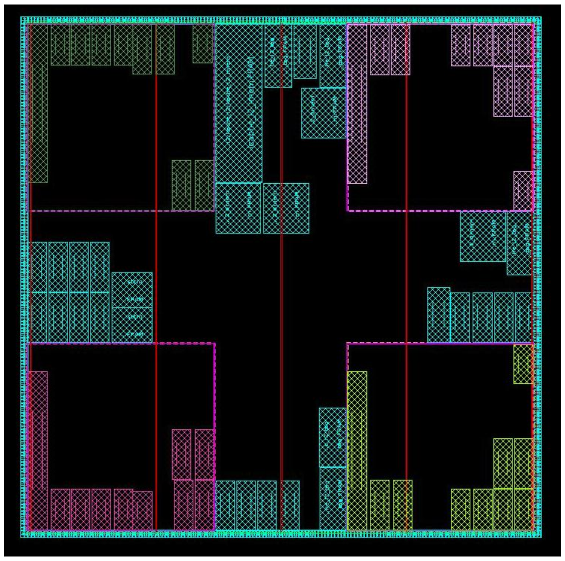


Figure 22 – Placement of plan-groups along with hard macros

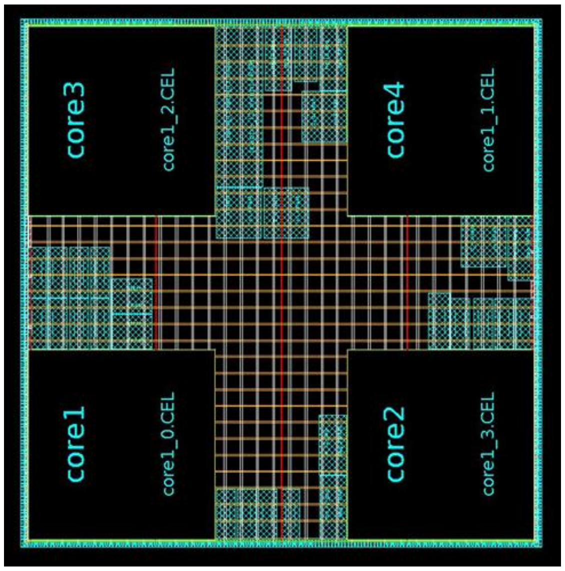


Figure 23 - Committing final plan-groups

Before committing, timing and area budgeting is done on the plan-groups. This ensures that the connections and congestion between these plan-groups is manageable in the top-level module. After this is done, the plan-groups can finally be committed. The script creates separate working folders, one for each plan-group.

4.4.1.2.2. Phase 2 - Design flow for all Plan-groups

In this phase, the user must navigate inside each of the folders, and perform the complete design flow for all the plan-groups from cell placement to post-route, and then chip-finish. Below are some of the important snapshots of the flow. Figure16-20 show some snapshots related to Phase2.

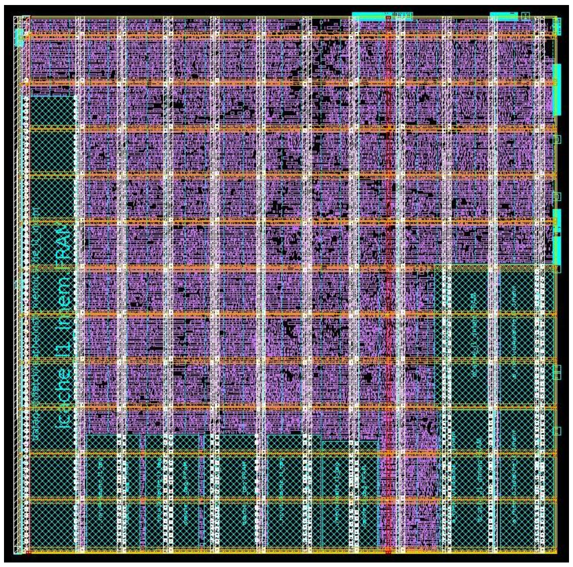
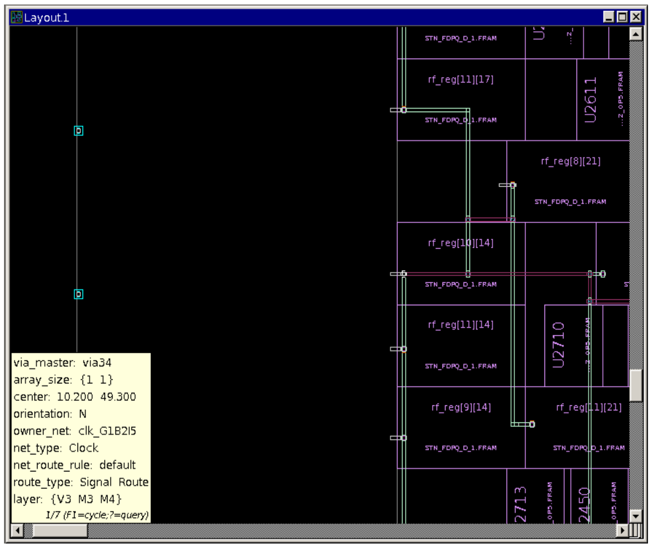


Figure 24 - Plan-group after placement



```
via_master:  via34
array_size:  {1  1}
center:  10.200  49.300
orientation:  N
owner_net:  clk_G1B2I5
net_type:  Clock
net_route_rule:  default
route_type:  Signal  Route
layer:  {V3  M3  M4}
        1/7 (F1=cycle;?=query)
```

Figure 25 - Zoom-in on the CTS stage showing clock routed on M3-M4

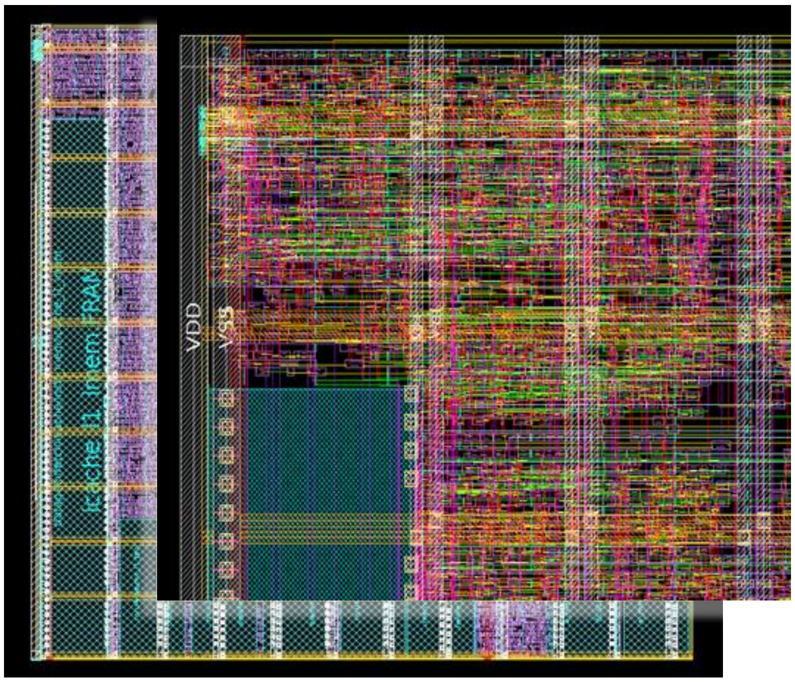
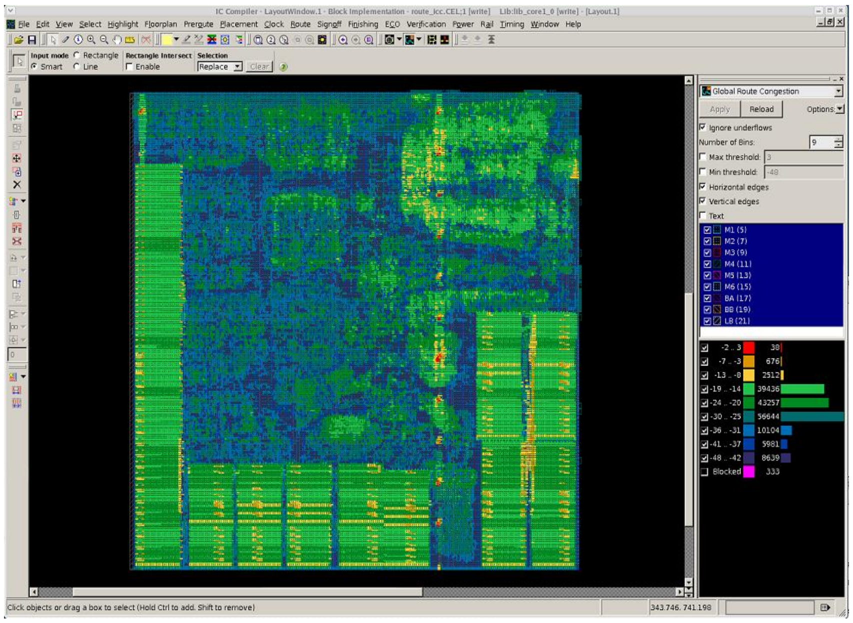


Figure 26 - Post-route Layout


IC Compiler - LayoutWindow.1 - Block Implementation - route_icc.CEL;1 [write] Lib:lib_core1_0 [write] - [Layout.1]
File Edit View Select Highlight Floorplan Preroute Placement Clock Route Signoff Finishing ECO Verification Power Rail Timing Window Help
Apply
Reload
Options:

Input mode: Smart
Rectangle Intersect: Enable ☐
Selection: Replace

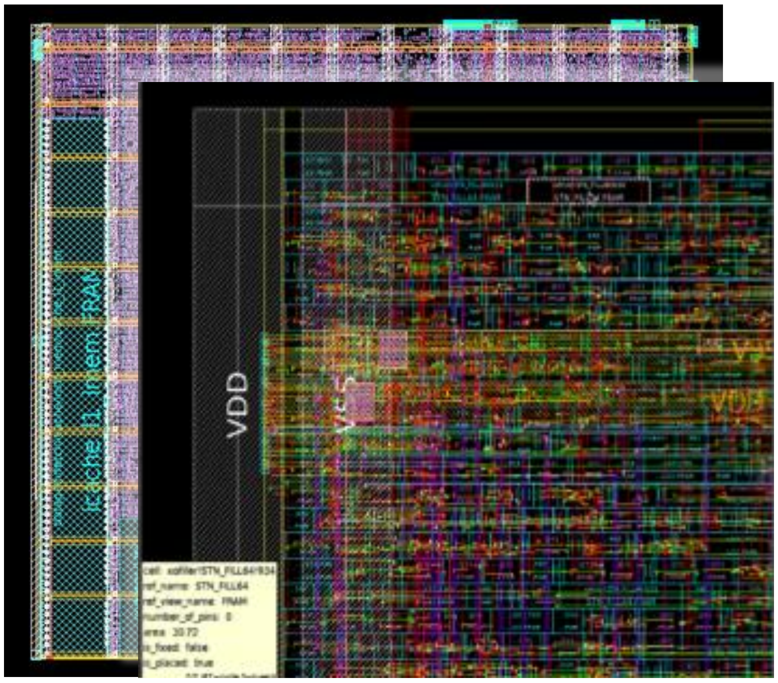
Global Route Congestion
☑ Ignore underflows
Number of Bins: 9
☐ Max threshold: 3
☐ Min threshold: -48
☑ Horizontal edges
☑ Vertical edges
☐ Text

☑ M1 (5)
☑ M2 (7)
☑ M3 (9)
☑ M4 (11)
☑ M5 (13)
☑ M6 (15)
☑ BA (17)
☑ BB (19)
☑ LB (21)

☑	-2 .. 3	🟥	38
☑	-7 .. -3	🟧	676
☑	-13 .. -8	🟨	2512
☑	-19 .. -14	🟩	39436
☑	-24 .. -20	🟩	43257
☑	-30 .. -25	🟦	56644
☑	-36 .. -31	🟦	10104
☑	-41 .. -37	🟦	5981
☑	-48 .. -42	🟪	8639
☐	Blocked	🟪	333

Click objects or drag a box to select (Hold Ctrl to add. Shift to remove)
343.746, 741.198

Figure 27 - Global Routing Congestion for plan-group



cell: xofiller!STN_FILL64!834
ref_name: STN_FILL64
ref_view_name: FRAM
number_of_pins: 0
area: 30.72
is_fixed: false
is_placed: true

Figure 28 - Chip Finish for plan-group

During this entire flow, Milkyway cells are saved for each step is saved in the work directory. This way, at the end of this phase, all the subgroups are fully ready with their Milkyway libraries, which can be readily used with the top-level module.

4.4.1.2.3. Phase 3 - Design flow for top-level module

Even if the majority portions of the design were divided into plan-groups, the design flow would still be needed for the top-level design. The only difference here is that the plan-groups for which the flow has already been done are now declared as macros with top-level design (for which we already have the Milkyway cells). Figure21-24 show figures related to Phase3.

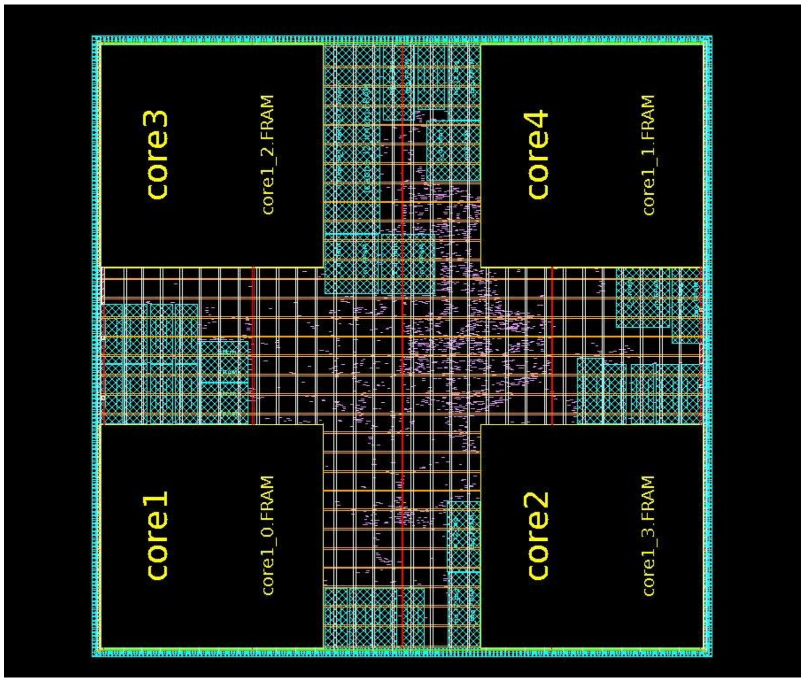
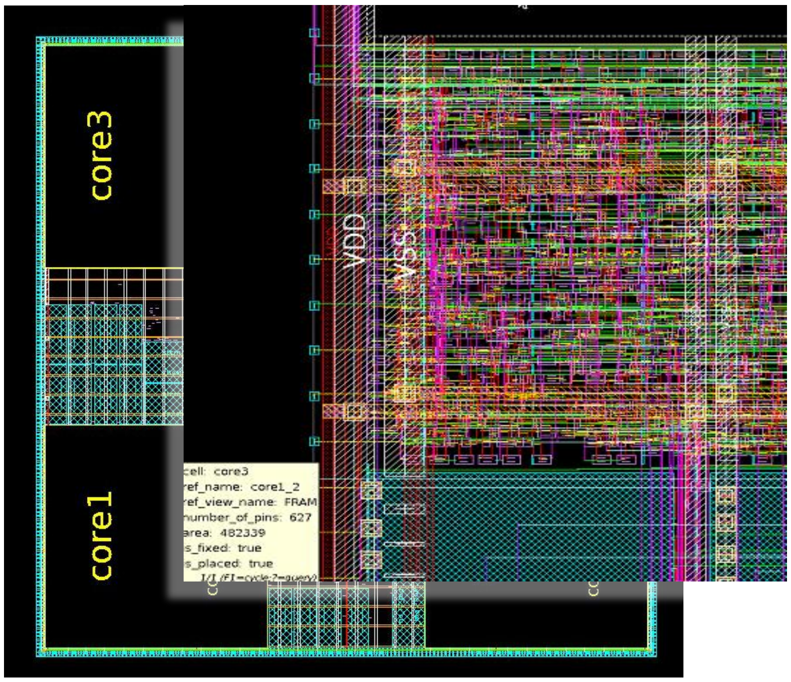


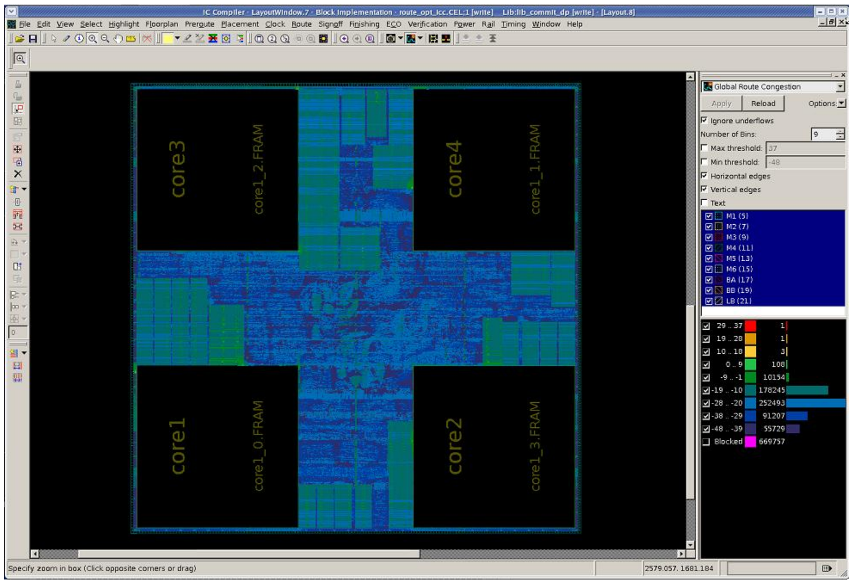
Figure 29 - Initialization of the top-level design



```
cell: core3
ref_name: core1_2
ref_view_name: FRAM
number_of_pins: 627
area: 482339
s_fixed: true
s_placed: true
```

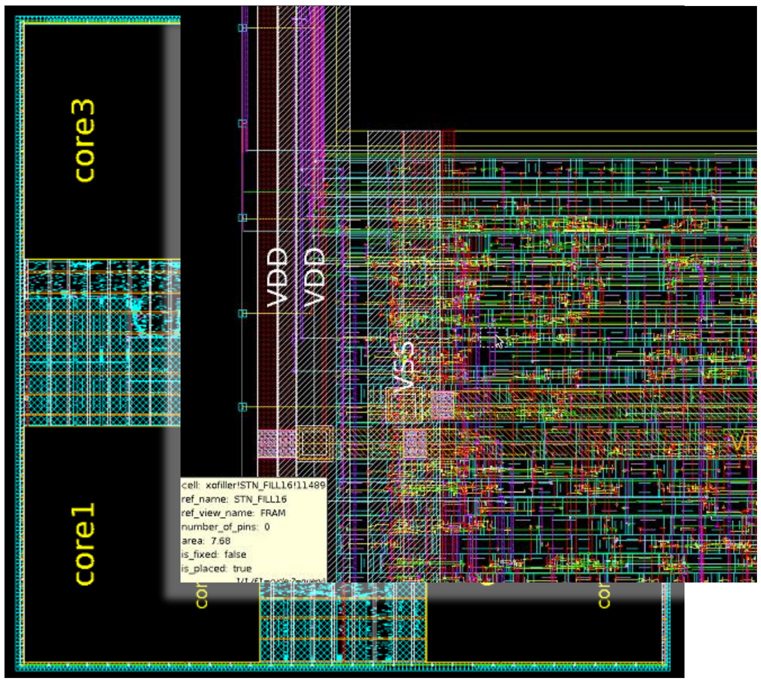
Figure 30 - Post Route of Top-level design

The final chip_finish script will generate the final GDSII, SDC, SDF, SPEF (Standard Parasitic Extraction Format), and final netlist (with and without physical cells).


IC Compiler - LayoutWindow.7 - Block Implementation - route_opt_icc.CEL;1 [write] Lib:lib_commit_dp [write] - [Layout.8]
File Edit View Select Highlight Floorplan Preroute Placement Clock Route Signoff Finishing ECO Verification Power Rail Timing Window Help
core3
core1_2.FRAM
core4
core1_1.FRAM
core1
core1_0.FRAM
core2
core1_3.FRAM
Global Route Congestion
Apply
Reload
Options:
Specify zoom in box (Click opposite corners or drag)
2579.057, 1681.184

```
[✓] Ignore underflows
Number of Bins: 9
[ ] Max threshold: 37
[ ] Min threshold: -48
[✓] Horizontal edges
[✓] Vertical edges
[ ] Text
[✓] M1 (5)
[✓] M2 (7)
[✓] M3 (9)
[✓] M4 (11)
[✓] M5 (13)
[✓] M6 (15)
[✓] BA (17)
[✓] BB (19)
[✓] LB (21)
[✓]  29 .. 37  🟥       1
[✓]  19 .. 28  🟧       1
[✓]  10 .. 18  🟨       3
[✓]   0 .. 9   🟩     108
[✓]  -9 .. -1  🟩   10154
[✓] -19 .. -10 🟦  178245
[✓] -28 .. -20 🟦  252493
[✓] -38 .. -29 🟦   91207
[✓] -48 .. -39 🟪   55729
[ ] Blocked    🟪  669757
```

Figure 31 - Global Routing Congestion for Top-level design



```
cell: xofiller!STN_FILL16!11489
ref_name: STN_FILL16
ref_view_name: FRAM
number_of_pins: 0
area: 7.68
is_fixed: false
is_placed: true
```

Figure 32 - Chip Finish of Top-level design

The netlist without the physical cells can be verified against the post-synthesis netlist (from Design Compiler) using tools like Formality. This ensures the correctness of the layout against the original netlist.

The post-layout netlist can also be used with the SDF (timing information from layout) can be used to evaluate *post-layout simulations*, if needed. The post-layout netlist and SPEF (RC parasitic extraction from layout) along with some activity file VCD (Value Change Dump) / SAIF (Switching Activity Interchange Format) can be used with Power Compiler to evaluate *post-layout power numbers*. Power Compiler can be invoked from DC Compiler, IC Compiler, Primetime, and Formality. [19]

Even after all the care taken, timing violations can still be encountered during any stage of the flow in 2nd or 3rd phase. In these cases, there are possibly two options – either change can be made in the RTL for the critical path(s), or the design constraints can be checked with the Design Compiler.

4.4.1.3. Graphical Analysis of Floorplan

Besides all the text reports generated by the IC Compiler, the final floorplan congestion can be graphically visualized in terms of cell density, and pin density. This graphical analysis is called '*Map Mode*'. The user can customize number of bins and thresholds for the color-coded bins. Figures26-27 show snapshots for the graphical analysis of cell and pin densities.

Various hierarchy levels can also be explored using a tool called Hierarchy browser that is built inside IC Compiler. Using Hierarchy browser, all or just some selected hierarchical cells, two examples of which are as shown in the Figures28-29. The Figure30 shows the location of top-level architecture units in the finalized floorplan.

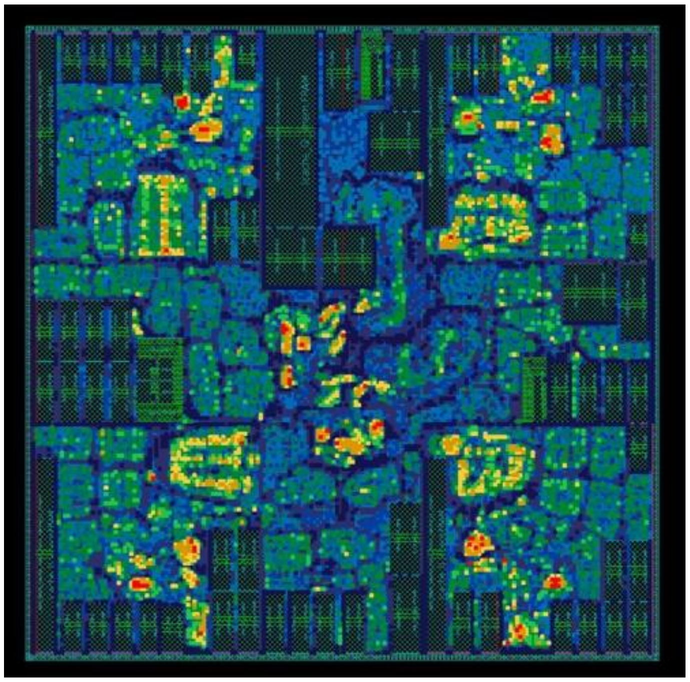


Figure 33 - Pin Density Analysis (Map Mode)

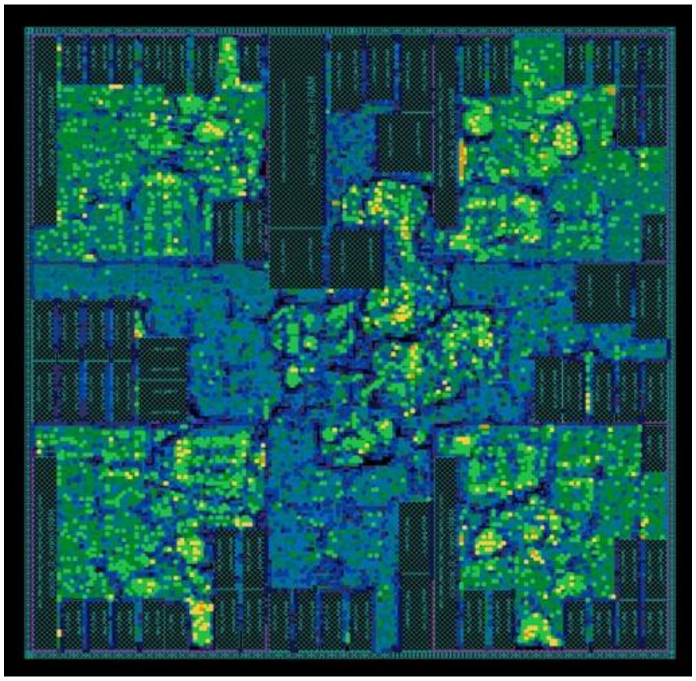


Figure 34 - Cell Density Analysis (Map Mode)

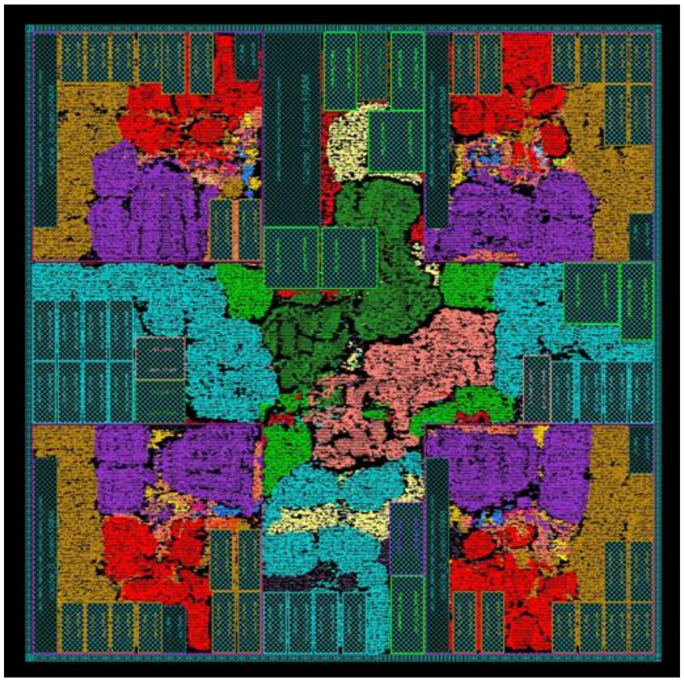


Figure 35 - Color coded hierarchy across top level and all plan groups

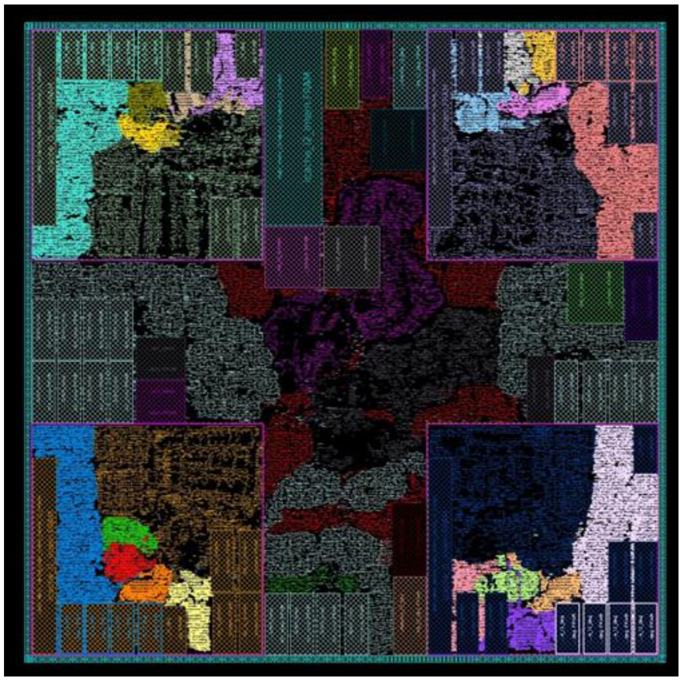


Figure 36 - Color-coded hierarchies inside cores

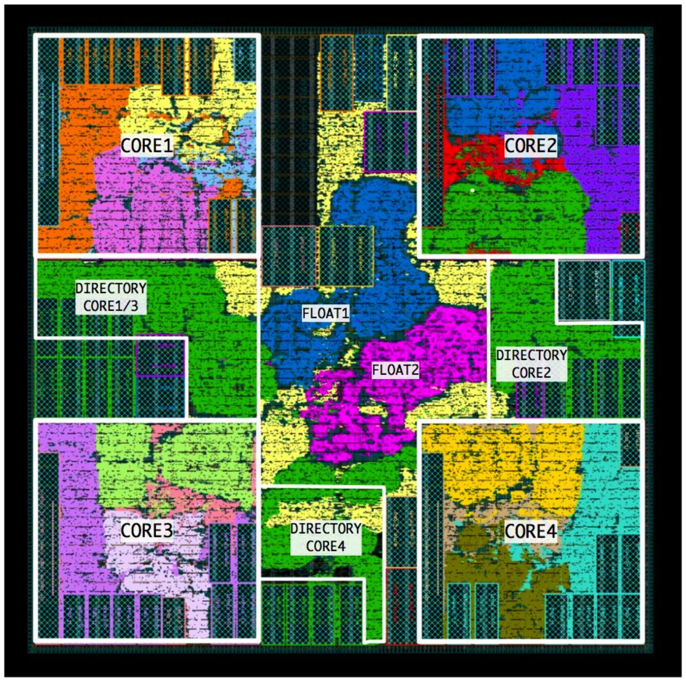


Figure 37 - Locations of architecture unit in top-level floorplan

4.4.2. Script and Setup

The basic framework and file structure of the scripts can be downloaded from the Synopsys website. These scripts provide a good start-up scripts set, but do not cover all commands or procedures that might be needed for a project. But the scripts can be easily modified to add custom codes.

Besides the basic inputs fetched from Design Compiler, IC Compiler needs some more technology related inputs. Most of these additional inputs are same as the ones, which were needed for Design Compiler. The other files needed by IC Compiler are Milkyway technology file (.tf), and TLUP (min and max). These files provide IC Compiler with all the information needed for doing the layout of the synthesized design. The scripts for IC Compiler along with the instructions have been added in Appendix.

CHAPTER V

CONCLUSION

The aim of the project is to create a realistic platform for development of MIPS-based multicore architectures, which could be adopted as a good platform for academic development. Unlike the software simulators, which provide very high level of abstraction, OSCAR (OSU Simulation for Computer Architecture Research) provides a very truthful insight into the accurate hardware investment. The project implements an RTL based structural model, which is completely synthesizable. The design can be used to perform timing and power analysis on all hierarchical modules, including the 3rd party memory macros.

Besides the direct utilization of this platform for academic purposes, the design basically offers a solid framework. The project implements four dynamically configurable MIPS-like cores, with shared memory resources and data coherency protocol. The table-driven approach improves performance by changing architecture on-the-fly using feedbacks from performance counters

The concept of the dynamically changing architecture to save power and also improving performance is very powerful.

However, there is a lot of scope for potential growth in this project, which will make the framework even robust and versatile. The cache blocks can be implemented with protocols such as MESI, MOESI (AMD), MESIF (Intel). Implementing branch prediction techniques will reflect in performance increase when testing with realistic workloads like the SPEC benchmarks.

For Level2 cache system, several other techniques, like *cache morphing* [20] can be used to improve performance of the shared memory resource. Implementing banked memories for Level2 cache can also help improve latencies.

REFERENCES

[1] S. Microsystems, "OpenSPARC T1 Microarchitecture Specification".

[2] T. Austin, "SimpleScalar LLC," [Online]. Available: http://www.simplescalar.com/.

[3] J. Larus, "SPIM A MIPS32 Simulator," [Online]. Available: http://pages.cs.wisc.edu/~larus/spim.html.

[4] "Wind River Simics," [Online]. Available: http://www.windriver.com/products/simics/.

[5] J. Montanaro, R. Witek, K. Anne, A. Black, E. Cooper, D. Dobberpuhl, P. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. Snyder, R. Stehpany and S. Thierauf, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *IEEE Journal of Solid-State Circuits,* vol. 31, no. 11, pp. 1703 - 1714, 1996.

[6] "International Technology Roadmap for Semiconductors (ITRS)," 2007.

[7] D. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *32nd Annual International Symposium on Microarchitecture (MICRO)*, 1999.

[8] G. Bournoutian and A. Orailoglu, "Dynamic, non-linear cache architecture for power-sensitive mobile processors," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, 2010.

[9] V. Kontorinis, A. Shayan, R. Kumar and D. Tullsen, "Reducing peak power with a table-driven adaptive processor core," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[10] D. Benitez, J. Moure, D. Rexachs and E. Luque, "A reconfigurable cache memory with heterogeneous banks," in *Design, Automation Test in Europe Conference Exhibition (DATE)*,

2010.

[11] H. Hanson, M. Hrishikesh, V. Agarwal, S. Keckler and D. Burger, "Static energy reduction techniques for microprocessor caches," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 11, no. 3, pp. 303-313, june 2003.

[12] J. E. Stine, A. Phadke and S. Tike, "A recursive-divide architecture for multiplication and division," in *2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011.

[13] A. Agarwal, J. Hennessy and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems,* pp. 393-431, Nov 1988.

[14] C. E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," in *Synopsys Users Group (SNUG)*, 2002.

[15] M. Graphics, "ModelSim Manual," [Online].

[16] HMC-MIPS. [Online]. Available: http://code.google.com/p/hmc-mips/.

[17] Synopsys, "Design Compiler User Guide," [Online].

[18] Synopsys, "Formality User Guide," [Online].

[19] Synopsys, "IC Compiler Design Planning User Guide," [Online].

[20] Synopsys, "PrimeTime User Guide," [Online].

[21] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. Irwin and Y. Xie, "MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy," in *IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[22] J. L. H. a. D. A. Patterson, Computer Architecture a Quantitative Approach, 4th Edition, San Mateo, CA: Morgan Kaufmann, 2012.

[23] M. Hill, "A case for direct-mapped caches," *Computer,* vol. 21, no. 12, pp. 25 -40, dec 1988.

[24] J. Heinrich, "MIPS R4000 Microprocessor Use’s Manual, Second Edition".

[25] S. Hangal and M. O'Connor, "Performance analysis and validation of the picoJava processor," *Micro, IEEE,* vol. 129, pp. 66-72, 1999.

[26] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Microarchitecture, 2001.*

MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on, 2001.

[27] D. Albonesi, R. Balasubramonian, S. Dropsbo, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook and S. Schuster, "Dynamically tuning processor resources with adaptive processing," *Computer,* vol. 36, no. 12, pp. 49 -58, dec. 2003.

APPENDICES

This appendix includes the script files and instructions to invoke and run commands to run the design flow for the hierarchical design planning for OSCAR. The Appendix is divided into several sections, each section dedicated for the scripts, which were customized for the project. The sections have been sequenced in the manner of execution.

Appendix A includes the following for ModelSim –

- Invocation commands for ModelSim
- Main script for ModelSim. (mips.do)
- Instructions on creating activity file VCD (Value Change Dump)

Appendix B covers following for Design Compiler –

- Invocation commands for Design Compiler
- Makefile script
- Source script for IBM_10LPE libraries (lib_10lpe.tcl)
- Main execution script for Design Compiler (compile_dc_ultra,tcl)

Appendix C includes the following for Formality –

- Invocation commands for Formality
- Main execution script for Formality (fm.tcl)

Appendix D coves the following for IC Compiler –

- Invocation commands for IC Compiler.
- Shell script to run complete hierarchical flow (run.sh)
- Common setup variables for hierarchical design planning (common_setup.tcl)
- TCL Script for initial hierarchical DP (create_plangroups.tcl)
- TCL Script to initialize floorplan for design (init_design.tcl)

Finally, Appendix E includes the following for PrimeTime –

- Invocation commands for PrimeTime
- Main execution script for PrimeTime (pt.tcl)

Since there are a lot of scripts in the actual script-set, only the ones with important modifications relevant to the project have been added. The other scripts are fairly similar to the ones which are available on the Synopsys website, and thus not included.

APPENDIX A

MODELSIM SCRIPTS

Invoking ModelSim

To invoke ModelSim with the mips.do file, use the following command from terminal -

```
=> vsim –do mips.do
```

Script - mips.do

```
#------------------------------------------------------------------------#
# Copyright 1991-2007 Mentor Graphics Corporation
# Modification by Oklahoma State University
# Use with Testbench
# James Stine, 2008
# Go Cowboys!!!!!!
# All Rights Reserved.
#
# Additional modifications by Surpriya Tike, 2011
#------------------------------------------------------------------------#
# THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION
# OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.
#------------------------------------------------------------------------#
onbreak {resume}
```

```
#-------------------------------------------#
# Create Library
#-------------------------------------------#
if [file exists work] {
#    vdel -all
}

vlib work


#-------------------------------------------#
# Create/Open Project
#-------------------------------------------#
if [file exists pasado.mpf] {
            project open pasado
            echo "Project opened"
            project compileoutofdate
} else
{
#           perl project_addfile.pl
#           project new ./ pasado
#           source project_addfile.tcl
            echo "New project created"
            vlog *.v ./backup/*.v +define+simulate=true
#           project compileall
}


#-------------------------------------------#
# Start and run simulation
#-------------------------------------------#
vsim  -novopt work.stimulus
view wave


#-------------------------------------------#
# Displays All Signals recursively
#-------------------------------------------#
add wave -hex /stimulus/dut1/core1/datapath/fetch/pcF
add wave -hex /stimulus/dut1/core1/datapath/instrD
add wave -hex /stimulus/dut1/core2/datapath/fetch/pcF
add wave -hex /stimulus/dut1/core2/datapath/instrD
add wave -hex /stimulus/dut1/core3/datapath/fetch/pcF
add wave -hex /stimulus/dut1/core3/datapath/instrD
add wave -hex /stimulus/dut1/core4/datapath/fetch/pcF
add wave -hex /stimulus/dut1/core4/datapath/instrD

#-------------------------------------------#

add wave -noupdate -divider -height 32 "Core1"
add wave -hex /stimulus/dut1/core1/datapath/fetch/pcF
add wave -hex /stimulus/dut1/core1/datapath/instrD
add wave -hex /stimulus/dut1/core1/datapath/ResultW
add wave -hex /stimulus/dut1/core1/datapath/RegWriteW
add wave -hex /stimulus/dut1/core1/datapath/WriteRegW
add wave -hex /stimulus/dut1/core1/datapath/imem_miss
add wave -hex /stimulus/dut1/core1/datapath/dmem_miss
add wave -hex /stimulus/dut1/core1/datapath/instrF

#------------------------------------------#

add wave -noupdate -divider -height 32 "L1 DMEM"
add wave -hex /stimulus/dut1/core1/datapath/memory/dcache/data_mem/*

#------------------------------------------#

add wave -noupdate -divider -height 32 "L2 DIR_CORE1"
add wave -hex /stimulus/dut1/directory/dir_dmem/dir_c1/*


#------------------------------------------#
add log -r /*

#------------------------------------------#
# Set Wave Output Items
#------------------------------------------#
TreeUpdate [SetDefaultTree]
WaveRestoreZoom {0 ps} {150 ns}

configure wave -namecolwidth 150
configure wave -justifyname right
configure wave -valuecolwidth 120
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2


# Run the Simulation
run 2000ns
wave seetime 2000ns
```

Script snippet – dumping VCD

When trying to dump VCD activity file using ModelSim, the following lines of code must be added into the testbench. The two main commands here are *$dump_file* and *$dump_vars*. These commands are called System Tasks and are derived from the IEEE1364 standard for Verilog.

```
initial
begin
        $dumpfile("top.vcd");
        $dumpvars(0,stimulus.dut1);

//      $sdf_annotate("top.sdf",dut1,,,"TYPICAL",,);
//      $sdf_annotate("core1_0.sdf",dut1,,,"MAXIMUM",,);
//      $sdf_annotate("core1_1.sdf",dut1,,,"MAXIMUM",,);
end
```

APPENDIX B

DESIGN COMPILER SCRIPTS

Invoking Design Compiler

To invoke Design Compiler with the compile_dc_ultra.tcl, refer the Makefile, or use the following command from terminal -

```
=> dc_shell-xg-t -64 - -f compile_dc_ultra.tcl
```

Script - Makefile

```
#/**************************************************/
#/* Makefile for Design Compiler – Synopsys        */
#/*                                               */
#/* Oklahoma State University - Go COWBOYS!!       */
#/**************************************************/
DC_EXEC = dc_shell-xg-t -64

# Source Options
OPTIONS =  -f compile_dc_ultra_10lpe.tcl


#************************************************
# Clean out all temporary files
#************************************************
.PHONY:
clean:
	rm -rf WORK alib-52 command.log
	rm -rf *.rep *.vh *.rpt filename* log* *.ddc *.sdc *.sdf default.svf .nfs*
	rm -rf verilog.tcl
```

```
sel_clean:
	rm -rf WORK alib-52 command.log
	rm -rf *.rep *.vh *.rpt filename* log* *.ddc *.sdc *.sdf default.svf


#**************************************************
# Compile a list of RTL files for synthesis
#**************************************************
verilog: sel_clean
	perl verilog.pl#**************************************************
# Invoke Design Compiler in 64 bit mode
#**************************************************
dc_shell: verilog
	date | tee log
	$(DC_EXEC) $(OPTIONS) | tee -a log
	date | tee -a log
```

Script – lib_10lpe.tcl

```
#/**************************************************/
#/* Library setup file for Synopsys – IBM10LPE     */
#/* See Makefile for invocation instructions       */
#/*                                                */
#/* Oklahoma State University - Go COWBOYS!!       */
#/**************************************************/

set DC_SHELL_DIR          [exec pwd]
set WORK_DIR                    $DC_SHELL_DIR/../verilog
set VIRAGE                "/import/vcag4/Virage/"
set SYNOPSYS_SYN_LIB            "/import/synopsys/syn_vD-2010.03/libraries/syn/"



#**************************************************
# Set search path for work directory and libraries
#**************************************************
set search_path [list     "./" \
                          "$WORK_DIR"\
                          "$VIRAGE/cp65npkssst/liberty/logic_synth/" \
                          "$VIRAGE/cp65npkssst/liberty/symbol/" \
                          "$VIRAGE/cp65npkhsst/liberty/logic_synth/" \
                          "$VIRAGE/cp65npkhsst/liberty/symbol/" \
                          "$VIRAGE/cp65npklsst/liberty/logic_synth/" \
                          "$VIRAGE/cp65npklsst/liberty/symbol/" \
                          "$SYNOPSYS_SYN_LIB" \
                          ]
```

```
#**************************************************
# Set Target and Synthtic libraries
#**************************************************
set target_library [list  \
                        cp65npkssst_tt1p2v25c.db \
                        cp65npkhsst_tt1p2v25c.db \
                        cp65npklsst_tt1p2v25c.db \
                        ${dcache_l1_dmem}.db \
                        ${dcache_l1_tag}.db \
                        ${dcache_l1_repl}.db \
                        ${dcache_l2_dmem}.db \
                        ${dcache_l2_tag}.db \
                        ${icache_l1_imem}.db \
                        ${icache_l1_tag}.db \
                        ${icache_l2_imem}.db \
                        ${icache_l2_tag}.db \
                        ${a0_sbtm}.db \
                        ${a1_sbtm}.db \
                        ${dir_cxou}.db \
                        ]


set synthetic_library       [list dw_foundation.sldb]
#**************************************************
# Set main link library
#**************************************************
set link_library  "$target_library $synthetic_library"
```

Script – compile_dc_ultra.tcl

```
#/**************************************************/
#/* Compile Script for Synopsys                  */
#/* See Makefile for invocation instructions       */
#/*                                              */
#/* This script uses IBM10LPE flow               */
#/**************************************************/


#**************************************************
# Define all verilog files for synthesis
#**************************************************
source verilog.tcl


#**************************************************
# Define top-level module and macro names
#**************************************************
set my_toplevel top
set dcache_l1_dmem     dcache_l1_dmem
set dcache_l1_repl     dcache_l1_repl
set dcache_l1_tag      dcache_l1_tag
set dcache_l2_dmem     dcache_l2_dmem
set dcache_l2_tag      dcache_l2_tag
set icache_l1_imem     icache_l1_imem
set icache_l1_tag      icache_l1_tag
set icache_l2_imem     icache_l2_imem
set icache_l2_tag      icache_l2_tag
set a0_sbtm            a0_sbtm
set a1_sbtm            a1_sbtm
set dir_cxou           dir_cxou


#***********************************************
# Timing and Basic Design Information
#***********************************************
set virtual 0
set fileformat verilog
set my_clock clk                          # Fastest clock for core frequency
set my_clock2 rclk_l2             # Clock for Level2 hierarchy
set my_clock3 rclk_l3             # Clock for main memory


set my_input_delay_ns 0
set my_output_delay_ns 0
set my_clk_freq_MHz 500
set my_clk2_freq_MHz 250
set my_clk3_freq_MHz 125


set my_period_ns  [expr 1000 / $my_clk_freq_MHz]
set my_period2_ns [expr 1000 / $my_clk2_freq_MHz]
set my_period3_ns [expr 1000 / $my_clk3_freq_MHz]


#***********************************************
# Switches to generate output files
#***********************************************
set make_db    0;# convert lib to db for macro
set write_v    1  ;# generates synthesized netlist
set write_sdc  1 ;# generates synopsys design constraint file for PnR
set write_ddc  1 ;# compiler file in ddc format
set write_sdf  1 ;# sdf file for back-annotated timing simulation
set write_rep  1 ;# generates estimated power, area and timing report


#***********************************************
# Convert .lib (memory from Virage Logic) to .db (Synopsys library format)
#***********************************************
if ($make_db) {
```

```
read_lib  [ format "%s%s" $dcache_l1_dmem ".lib"]
write_lib [ format "%s%s" $dcache_l1_dmem "_lib"]  -f db -o [ format "%s%s"
$dcache_l1_dmem ".db"]
read_lib  [ format "%s%s" $dcache_l1_repl ".lib"]
write_lib [ format "%s%s" $dcache_l1_repl "_lib"]  -f db -o [ format "%s%s" $dcache_l1_repl
".db"]
read_lib  [ format "%s%s" $dcache_l1_tag ".lib"]
write_lib [ format "%s%s" $dcache_l1_tag "_lib"]   -f db -o [ format "%s%s" $dcache_l1_tag
".db"]
read_lib  [ format "%s%s" $dcache_l2_dmem ".lib"]
write_lib [ format "%s%s" $dcache_l2_dmem "_lib"]  -f db -o [ format "%s%s"
$dcache_l2_dmem ".db"]
read_lib  [ format "%s%s" $dcache_l2_tag ".lib"]
write_lib [ format "%s%s" $dcache_l2_tag "_lib"]   -f db -o [ format "%s%s" $dcache_l2_tag
".db"]
read_lib  [ format "%s%s" $icache_l1_imem ".lib"]
write_lib [ format "%s%s" $icache_l1_imem "_lib"]  -f db -o [ format "%s%s" $icache_l1_imem
".db"]
read_lib  [ format "%s%s" $icache_l1_tag ".lib"]
write_lib [ format "%s%s" $icache_l1_tag "_lib"]   -f db -o [ format "%s%s" $icache_l1_tag
".db"]
read_lib  [ format "%s%s" $icache_l2_imem ".lib"]
write_lib [ format "%s%s" $icache_l2_imem "_lib"]  -f db -o [ format "%s%s" $icache_l2_imem
".db"]
read_lib  [ format "%s%s" $icache_l2_tag ".lib"]
write_lib [ format "%s%s" $icache_l2_tag "_lib"]   -f db -o [ format "%s%s" $icache_l2_tag
".db"]
read_lib  [ format "%s%s" $a0_sbtm ".lib"]
write_lib [ format "%s%s" $a0_sbtm "_lib"]        -f db -o [ format "%s%s" $a0_sbtm ".db"]
read_lib  [ format "%s%s" $a1_sbtm ".lib"]
write_lib [ format "%s%s" $a1_sbtm "_lib"]        -f db -o [ format "%s%s" $a1_sbtm ".db"]
read_lib  [ format "%s%s" $dir_cxou ".lib"]
write_lib [ format "%s%s" $dir_cxou "_lib"]       -f db -o [ format "%s%s" $dir_cxou ".db"]
}


#***********************************************
# Library setup, and reading design
#***********************************************
source ./lib_10lpe.tcl
define_design_lib WORK -path ./WORK
set verilogout_show_unconnected_pins "true"
set set_fix_multiple_port_nets "true"



# Analyze and elaborating design
analyze -f $fileformat $my_verilog_files
elaborate $my_toplevel -update
current_design $my_toplevel
list_instances
link
```

```
if {!$virtual} {
	create_clock -period $my_period_ns  $my_clock
	create_clock -period $my_period2_ns $my_clock2
	create_clock -period $my_period3_ns $my_clock3
} else
{
	set clk_name vclk
	create_clock -period $my_period_ns -name $clk_name
}


set_input_delay  $my_input_delay_ns -clock $my_clock [remove_from_collection [all_inputs]
[list  $my_clock2 $my_clock3 $my_clock]]
set_output_delay $my_output_delay_ns -clock $my_clock [all_outputs]
set_driving_cell -lib_cell STN_INV_S_8  [all_inputs]
set_fix_hold     [list  $my_clock2 $my_clock3 $my_clock]


#*********************************************
# Compile deisgn with optimization switches
#*********************************************
compile_ultra -no_boundary_optimization -no_autoungroup
group_path -name REGOUT -to [all_outputs]
group_path -name REGIN -from [all_inputs]
group_path -name FEEDTHROUGH -from [all_inputs] -to [all_outputs]


#*********************************************
# Dump reports
#*********************************************
set filename [format "%s%s"  $my_toplevel "_reportpath.rep"]
redirect $filename { report_path_group }
redirect check_design_ultra.rpt { check_design }


if  {$write_v} {
set filename [format "%s%s"  $my_toplevel "_ultra.vh"]
	define_name_rules verilog -allowed "a-z" -allowed "A-Z"
	define_name_rules verilog -add_dummy_nets -replacement_char "_" -restricted
"!@#$%^&*()||-/ "  -first_restricted "!@#$%^&*()||-/ "
	write -format verilog -hierarchy -output $filename
}


if  {$write_sdc} {
	set filename [format "%s%s"  $my_toplevel "_ultra.sdc"]
	write_sdc $filename
}
```

```
if {$write_sdf} {
	set filename [format "%s%s"  $my_toplevel "_ultra.sdf"]
	write_sdf $filename
}


if {$write_ddc} {
	set filename [format "%s%s"  $my_toplevel "_ultra.ddc"]
	write -format ddc -hierarchy -o $filename
}


if {$write_rep} {
	set filename [format "%s%s" $my_toplevel "_timing.rep"]
	redirect $filename { report_timing }
	set filename [format "%s%s" $my_toplevel "_area.rep"]
	redirect $filename { report_area }
	set filename [format "%s%s" $my_toplevel "_power.rep"]
	redirect $filename { report_power }
}
exit
```

APPENDIX C

FORMALITY SCRIPTS

Invoking Formality

To invoke Formality with the fm.tcl, use the following command from terminal -

```
=> fm_shell -64 –f fm.tcl
```

Script – fm.tcl

```
#/**************************************************/
#/* Formality Script for Synopsys                 */
#/* See Makefile for invocation instructions       */
#/*                                               */
#/* This script uses IBM10LPE flow                 */
#/**************************************************/

set my_toplevel top
set_app_var synopsys_auto_setup true
source lib_10lpe.tcl
source verilog.tcl


#************************************************
# Handling black-boxes
#************************************************
set_app_var hdlin_unresolved_modules black_box
```

```
#************************************************
# Reading SVF file(s)
#************************************************
set_svf default.svf


#************************************************
# Read technology libraries for implemented design
#************************************************
foreach tech_lib "${target_library} ${synthetic_library}" {
  read_db -technology_library $tech_lib
}


#************************************************
# Read reference design
#************************************************
read_verilog -r ${my_verilog_files} -work_library WORK
set_top r:/WORK/${my_toplevel}


#************************************************
# Read implemented design (Verilog/DDC/MilkyWay)
#************************************************
read_verilog -i ${my_toplevel}_ultra.vh
set_top i:/WORK/${my_toplevel}

# read_ddc -i ${RESULTS_DIR}/${DCRM_FINAL_DDC_OUTPUT_FILE}
# read_milkyway -i -no_pg -libname WORK -cell_name ${DCRM_FINAL_MW_CEL_NAME}
${mw_design_library}


#************************************************
# Match compare points and final verification
#************************************************
report_setup_status


match
report_unmatched_points > match.rep


verify
report_passing_points > passing_points.rep
report_failing_points > failing_points.rep


#************************************************
# Final Reporting
#************************************************
if { ![verify] }  {
  save_session -replace final_session
  report_aborted > abort.rep
  analyze_points -all > points.rep
}


exit
```

APPENDIX D

IC COMPILER SCRIPTS

Invoking IC Compiler

To invoke IC Compiler, refer to the Makefile or use the following command from terminal -

```
=> ic_shell -64 –f [tcl_file_name].tcl
```

Script – run.sh

```
#/**************************************************/
#/* Source script to run complete hierarchical     */
#/* design Planning for IC Compiler                */
#/*                                                */
#/**************************************************/
#!/bin/bash
newsyn          # Source synopsys script to setup environment
make hier_dp    # Run hierarchical design planning for deisgn


#*************************************************
# Perform Design Flow for each plan-group
#*************************************************
cd core1_0
make ic

cd ../core1_1
make ic
```

```
cd ../core1_2
make ic

cd ../core1_3
make ic


#**********************************************
# Perform final Design Flow for top-level
#**********************************************
cd ../top
cp -r ../core1_0/lib_core1_0 .
cp -r ../core1_1/lib_core1_1 .
cp -r ../core1_2/lib_core1_2 .
cp -r ../core1_3/lib_core1_3 .


make ic
```

Script – common_setup.tcl

```
#/*************************************************/
#/* Common source script to setup all libraries    */
#/*                                                */
#/* This script uses IBM10LPE flow                 */
#/*************************************************/
set ICC_SHELL_DIR              [exec pwd]
set WORK_DIR                   "$ICC_SHELL_DIR/../dc_shell"
set MW                         "/home2/stike/work/ic_compiler_flow"
set VIRAGE                     "/import/vcag4/Virage"
set TLU_PATH                   "/import/vlsi4/IBM_PDK/cmos10lpe/rel/
cmos10lpe_B_synopsys_20090316/synopsys/v.20090316/tluplus"


set DESIGN_NAME                "top";
set DESIGN_REF_DATA_PATH       ""  ;


#**********************************************
# Hierarchical Flow Design Variables
#**********************************************
set HIERARCHICAL_DESIGNS       "core1 core2 core3 core4";
set HIERARCHICAL_CELLS         "core1 core2 core3 core4";


#**********************************************
# Library Setup Variables
#**********************************************
```

```
set ADDITIONAL_SEARCH_PATH  [ list \
                        "$WORK_DIR"\
                        "$VIRAGE/cp65npkssst/liberty/logic_synth/" \
                        "$VIRAGE/cp65npkssst/liberty/symbol/" \
                        "$VIRAGE/cp65npkhsst/liberty/symbol/" \
                        "$VIRAGE/cp65npkhsst/liberty/logic_synth/"\
                        "$VIRAGE/cp65npklsst/liberty/logic_synth/"\
                        "$VIRAGE/cp65npklsst/liberty/symbol/"\
                        "/import/synopsys/syn_vD-2010.03/libraries/ syn/dw_foundation"\
                        "/import/synopsys/syn_vD-2010.03/libraries/ syn/"];


set TARGET_LIBRARY_FILES    [list \
                        cp65npkssst_tt1p2v25c.db \
                        cp65npkhsst_tt1p2v25c.db \
                        cp65npklsst_tt1p2v25c.db \
                        dcache_l1_dmem.db \
                        dcache_l1_tag.db \
                        dcache_l1_repl.db \
                        dcache_l2_dmem.db \
                        dcache_l2_tag.db \
                        icache_l1_imem.db \
                        icache_l1_tag.db \
                        icache_l2_imem.db \
                        icache_l2_tag.db \
                        a0_sbtm.db \
                        a1_sbtm.db \
                        dir_cxou.db];


set SYNTHETIC_LIBRARY       [list dw_foundation.sldb]
set ADDITIONAL_LINK_LIB_FILES    "$TARGET_LIBRARY_FILES
$SYNTHETIC_LIBRARY";
set MIN_LIBRARY_FILES           "";
set MW_REFERENCE_CONTROL_FILE    "";


set MW_REFERENCE_LIB_DIRS   [list \
                        $MW/convert_ssst/cp65npkssst_m08f2f0 \
                        $MW/convert_hsst/cp65npkhsst_m08f2f0 \
                        $MW/convert_lsst/cp65npklsst_m08f2f0 \
                        $ICC_SHELL_DIR/a0_sbtm \
                        $ICC_SHELL_DIR/a1_sbtm \
                        $ICC_SHELL_DIR/dcache_l1_dmem \
                        $ICC_SHELL_DIR/dcache_l1_tag \
                        $ICC_SHELL_DIR/dcache_l1_repl \
                        $ICC_SHELL_DIR/dcache_l2_dmem \
                        $ICC_SHELL_DIR/dcache_l2_tag \
                        $ICC_SHELL_DIR/dir_cxou \
                        $ICC_SHELL_DIR/icache_l1_imem \
                        $ICC_SHELL_DIR/icache_l1_tag \
                        $ICC_SHELL_DIR/icache_l2_imem \
                        $ICC_SHELL_DIR/icache_l2_tag];


set TECH_FILE              [list \
                        $MW/convert_ssst/cp65npkssst_m08f2f0.tf \
                        $MW/convert_hsst/cp65npkhsst_m08f2f0.tf \
                        $MW/convert_lsst/cp65npklsst_m08f2f0.tf \
                        ];#  Milkyway technology file


set MAP_FILE       "$TLU_PATH/cmos10lpe_9LB_6_02_00_00.astro_layername.map";
set TLUPLUS_MAX_FILE  "$TLU_PATH/cmos10lpe_9LB_6_02_00_00_FuncCmax.tluplus";
set TLUPLUS_MIN_FILE  "$TLU_PATH/cmos10lpe_9LB_6_02_00_00_FuncCmin.tluplus";


set MW_POWER_NET              "VDD" ;
set MW_POWER_PORT                        "VDD" ;
set MW_GROUND_NET                        "VSS" ;
set MW_GROUND_PORT                       "VSS" ;


set MIN_ROUTING_LAYER                    "M1"  ;
set MAX_ROUTING_LAYER                    "BA"  ;


set LIBRARY_DONT_USE_FILE                ""       ;



#**************************************************
# Multi-Voltage Common Variables
#**************************************************
set PD1                  "";    # Name of power/voltage area  1
set PD1_CELLS            "";        # Instances for power/voltage area 1
set VA1_COORDINATES      {};         # Coordinates for voltage area 1
set MW_POWER_NET1        "VDD1";  # Power net for voltage area 1
set MW_POWER_PORT1       "VDD" ;  # Power port for voltage area 1


set PD2                  "";    # Name of power/voltage area  2
set PD2_CELLS            "";        # Instances for power/voltage area 2
set VA2_COORDINATES      {};         # Coordinates for voltage area 2
set MW_POWER_NET2        "VDD2";  # Power net for voltage area 2
set MW_POWER_PORT2       "VDD" ;  # Power port for voltage area 2
```

Script – create_plangroups_dp.tcl

```
#/***********************************************/
#/* Script to create plan-groups for IC Compiler   */
#/*                                                */
#/* This script uses IBM10LPE flow                 */
#/***********************************************/
source -echo icc_setup.tcl
gui_set_current_task -name {Design Planning}


open_mw_lib $MW_DESIGN_LIBRARY
copy_mw_cel -from $ICC_FLOORPLAN_CEL -to $ICC_DP_CREATE_PLANGROUPS_CEL
open_mw_cel $ICC_DP_CREATE_PLANGROUPS_CEL
link
source  ./SCRIPTS/common_placement_settings_icc.tcl


## (Optional) Set ideal network on nets with fanout larger than the specified threshold
if {$ICC_DP_SET_HFNS_AS_IDEAL_THRESHOLD != ""} {
set hf_nets [all_high_fanout -nets -threshold
$ICC_DP_SET_HFNS_AS_IDEAL_THRESHOLD]
if { $hf_nets != "" } {
redirect /dev/null {set_load 0 -subtract_pin_load $hf_nets}
redirect /dev/null {set_ideal_network -no_propagate $hf_nets}
}
}


## Additional reporting before the major steps
if {$ICC_DP_VERBOSE_REPORTING} {
check_design -summary > ${REPORTS_DIR_DP_CREATE_PLANGROUPS} \
${ICC_DP_CREATE_PLANGROUPS_CEL}_pre.check_design.rpt
report_net_fanout -threshold 50 > ${REPORTS_DIR_DP_CREATE_PLANGROUPS} \
${ICC_DP_CREATE_PLANGROUPS_CEL}_pre.high_fanout.rpt
}


#**********************************************
# Create Plan-Groups
#**********************************************
if {[file exists [which $ICC_DP_PLANGROUP_FILE]]} {
source $ICC_DP_PLANGROUP_FILE
} elseif {$ICC_DP_PLAN_GROUPS != ""} {
create_plan_groups $ICC_DP_PLAN_GROUPS          -cycle_color \
-target_aspect_ratio 1
} else
{
echo "WARNING: Please create plan groups before continuing with hierarchical flow"
}
create_fp_plan_group_padding  -internal_widths {2 2 2 2} \
-external_widths {2 2 2 2} \
[get_plan_groups *]


#************************************************
# Placement Constraints on Plan-Groups
#************************************************
if {[all_macro_cells] != ""} {
if {$ICC_DP_FIX_MACRO_LIST eq ""} {
remove_dont_touch_placement [all_macro_cells]
} elseif {$ICC_DP_FIX_MACRO_LIST eq "skip"}
{
echo "remove_dont_touch_placement for macros is skipped"
} else
{
remove_dont_touch_placement [all_macro_cells]
set_dont_touch_placement $ICC_DP_FIX_MACRO_LIST
}
}

if {[file exists [which $CUSTOM_ICC_DP_PLACE_CONSTRAINT_SCRIPT]]}
{
source $CUSTOM_ICC_DP_PLACE_CONSTRAINT_SCRIPT}


#************************************************
# Shape and place all Plan-Groups
#************************************************
if {$ICC_DP_PLAN_GROUPS != "" && $ICC_DP_PLANGROUP_FILE == ""} {

set_fp_placement_strategy         -sliver_size 5 \
-adjust_shapes off \
-macros_on_edge off


set_host_options -max_cores $ICC_NUM_CORES
create_fp_placement -effort low -no_legalize -congestion_effort high


set area_top [get_attribute [get_core_area] bbox]
set area_top [join $area_top]
set core_leftend [lindex $area_top 0]
set core_rightend [lindex $area_top 2]


set core1_dimen [get_attribute -class plan_group core1 bbox]
set core1_dimen [join $core1_dimen]
set core1_x [lindex $core1_dimen 0]
set core1_y [lindex $core1_dimen 2]
set core1_dim [expr $core1_y - $core1_x]
move_objects    -x $core_leftend \
-y $core_leftend        core1


set core2_dimen [get_attribute -class plan_group core2 bbox]
set core2_dimen [join $core2_dimen]
set core2_x [lindex $core2_dimen 0]
set core2_y [lindex $core2_dimen 2]
set core2_dim [expr $core2_y - $core2_x]
move_objects    -x [expr $core_rightend - $core2_dim] \
-y $core_leftend core2


set core3_dimen [get_attribute -class plan_group core3 bbox]
set core3_dimen [join $core3_dimen]
set core3_x [lindex $core3_dimen 0]
set core3_y [lindex $core3_dimen 2]
set core3_dim [expr $core3_y - $core3_x]
move_objects    -x $core_leftend \
-y [expr $core_rightend - $core3_dim] core3


set core4_dimen [get_attribute -class plan_group core4 bbox]
set core4_dimen [join $core4_dimen]
set core4_x [lindex $core4_dimen 0]
set core4_y [lindex $core4_dimen 2]
set core4_dim [expr $core4_y - $core4_x]
move_objects    -x [expr $core_rightend - $core4_dim] \
-y [expr $core_rightend - $core4_dim] core4


save_mw_cel
set_object_fixed_edit [get_cells $HIERARCHICAL_CELLS] 1
}


#***********************************************
# Other placements - macros
#***********************************************
set_fp_placement_strategy -macros_on_edge on
set_fp_placement_strategy -adjust_shapes on
create_fp_placement -effort high -congestion_effort high


report_fp_placement > ${REPORTS_DIR_DP_CREATE_PLANGROUPS} \
${ICC_DP_CREATE_PLANGROUPS_CEL}_place.placement_rpt


if {$DFT && $ICC_DP_DFT_FLOW} {
optimize_dft -plan_group
redirect -file $REPORTS_DIR_DP_CREATE_PLANGROUPS \
${ICC_DP_CREATE_PLANGROUPS_CEL}_check_scan_chain.rpt {check_scan_chain}
```

```
}

save_mw_cel


# -----------------------------------------------------------------
# Floorplan Screeshot - Surpriya Tike
# -----------------------------------------------------------------
gui_start
set area_top                    [get_attribute [get_core_area] bbox]
gui_zoom                        -window [gui_get_current_window -types Layout -mru] \
-rect $area_top -fit
gui_zoom                        -window [gui_get_current_window -types Layout -mru] \
-fit -factor 0.7
gui_write_window_image          -window [gui_get_current_window -types Layout -mru] \
-file $SNAPSHOTS_DIR/$ICC_DP_CREATE_PLANGROUPS_CEL \
-format jpg
stop_gui


close_mw_lib
exit
```

Script – init_design_icc.tcl

```
#/**************************************************/
#/* Script to create plan-groups for IC Compiler   */
#/*                                                */
#/* This script uses IBM10LPE flow                 */
#/**************************************************/
source -echo ./icc_setup.tcl


if { $ICC_INIT_DESIGN_INPUT == "MW" } {
open_mw_cel $ICC_INPUT_CEL -library $MW_DESIGN_LIBRARY
if {$DFT && $ICC_DP_DFT_FLOW && !$ICC_SKIP_IN_BLOCK_IMPLEMENTATION} {
if {[file exists [which $ICC_IN_FULL_CHIP_SCANDEF_FILE]]} {
read_def $ICC_IN_FULL_CHIP_SCANDEF_FILE
} else
{
echo "SCRIPT-Error: $ICC_DP_DFT_FLOW is set to true but SCANDEF file
$ICC_IN_FULL_CHIP_SCANDEF_FILE is not found. Please investigate it"
}
}
} else
{
if { ![file exists [which $MW_DESIGN_LIBRARY/lib]] } {
if { [file exists [which $MW_REFERENCE_CONTROL_FILE]]} {
```

```
create_mw_lib \
-tech {$TECH_FILE} \
-bus_naming_style {[%d]} \
-reference_control_file $MW_REFERENCE_CONTROL_FILE \
$MW_DESIGN_LIBRARY
} else
{
create_mw_lib \
-tech {$TECH_FILE} \
-bus_naming_style {[%d]} \
-mw_reference_library $MW_REFERENCE_LIB_DIRS \
$MW_DESIGN_LIBRARY
}
}
}


if {$ICC_INIT_DESIGN_INPUT == "DDC" } {
open_mw_lib $MW_DESIGN_LIBRARY
suppress_message "UID-3"      ;# avoid local link library messages
import_designs ../dc_shell/$ICC_IN_DDC_FILE \
-format ddc -top $DESIGN_NAME \
-cel $DESIGN_NAME
unsuppress_message "UID-3"

if {$DFT && $ICC_DP_DFT_FLOW && !$ICC_SKIP_IN_BLOCK_IMPLEMENTATION} {
if {[file exists [which $ICC_IN_FULL_CHIP_SCANDEF_FILE]]} {
remove_scan_def
read_def $ICC_IN_FULL_CHIP_SCANDEF_FILE
} else
{
echo "SCRIPT-Error: $ICC_DP_DFT_FLOW is set to true but SCANDEF file
$ICC_IN_FULL_CHIP_SCANDEF_FILE is not found. Please investigate it"
}
}
}


if {$ICC_INIT_DESIGN_INPUT == "VERILOG" } {
open_mw_lib $MW_DESIGN_LIBRARY
set hdlin_preserve_sequential  "true"
read_verilog -verbose -top $DESIGN_NAME $ICC_IN_VERILOG_NETLIST_FILE
uniquify_fp_mw_cel
link
current_design $DESIGN_NAME
read_sdc $ICC_IN_SDC_FILE


if {$DFT && $ICC_DP_DFT_FLOW && !$ICC_SKIP_IN_BLOCK_IMPLEMENTATION} {
if {[file exists [which $ICC_IN_FULL_CHIP_SCANDEF_FILE]]} {
read_def $ICC_IN_FULL_CHIP_SCANDEF_FILE
```

```
} else
{
echo "SCRIPT-Error: $ICC_DP_DFT_FLOW is set to true but SCANDEF file
$ICC_IN_FULL_CHIP_SCANDEF_FILE is not found. Please investigate it"
}
}
}


if { [check_error -verbose] != 0} { echo "SCRIPT-Error, flagging ..." }
if {$DFT && $ICC_DP_DFT_FLOW && !$ICC_SKIP_IN_BLOCK_IMPLEMENTATION} {
redirect -file $REPORTS_DIR_INIT_DESIGN/
$DESIGN_NAME.full_chip_check_scan_chain.rpt {check_scan_chain}
}


if {$ICC_CTS_INTERCLOCK_BALANCING &&
[file exists [which $ICC_CTS_INTERCLOCK_BALANCING_OPTIONS_FILE]]} {
source -echo $ICC_CTS_INTERCLOCK_BALANCING_OPTIONS_FILE
}


if {$ICC_INIT_DESIGN_INPUT == "VERILOG" } {
set ports_clock_root {}
foreach_in_collection a_clock [get_clocks -quiet] {
set src_ports [filter_collection [get_attribute $a_clock sources] @object_class==port]
set ports_clock_root  [add_to_collection $ports_clock_root $src_ports]
}
group_path -name REGOUT -to [all_outputs]
group_path -name REGIN -from [remove_from_collection [all_inputs] $ports_clock_root]
group_path -name FEEDTHROUGH -from [remove_from_collection [all_inputs]
$ports_clock_root] -to [all_outputs]
}
remove_input_delay [get_clocks {rclk_l2 rclk_l3}]
remove_propagated_clock [all_fanout -clock_tree -flat]
remove_propagated_clock *


# Timing derate
if {$ICC_APPLY_RM_DERATING} {
set_timing_derate -early $ICC_EARLY_DERATING_FACTOR -cell_delay
set_timing_derate -late  $ICC_LATE_DERATING_FACTOR  -cell_delay
set_timing_derate -early $ICC_EARLY_DERATING_FACTOR -net_delay
set_timing_derate -late  $ICC_LATE_DERATING_FACTOR  -net_delay
}

if {$ICC_CRITICAL_RANGE != ""} {
echo $ICC_CRITICAL_RANGE ; set_critical_range $ICC_CRITICAL_RANGE
[current_design]}
if {$ICC_MAX_TRANSITION != ""} {
```

```
echo $ICC_MAX_TRANSITION ; set_max_transition $ICC_MAX_TRANSITION
[current_design]}
if {$ICC_MAX_FANOUT != ""} {
echo $ICC_MAX_FANOUT ; set_max_fanout $ICC_MAX_FANOUT [current_design]
}
if {$TLUPLUS_MIN_FILE == ""}
{
set TLUPLUS_MIN_FILE $TLUPLUS_MAX_FILE}
if {$TLUPLUS_MAX_EMULATION_FILE == ""} {
set_tlu_plus_files          -max_tluplus $TLUPLUS_MAX_FILE \
-min_tluplus $TLUPLUS_MIN_FILE \
-tech2itf_map $MAP_FILE
} else
{
if {$TLUPLUS_MIN_EMULATION_FILE == ""} {
set TLUPLUS_MIN_EMULATION_FILE $TLUPLUS_MAX_EMULATION_FILE
}
set_tlu_plus_files          -max_tluplus $TLUPLUS_MAX_FILE \
-min_tluplus $TLUPLUS_MIN_FILE \
-max_emulation_tluplus $TLUPLUS_MAX_EMULATION_FILE \
-min_emulation_tluplus $TLUPLUS_MIN_EMULATION_FILE \
-tech2itf_map $MAP_FILE
}

report_tlu_plus_files

if {$ICC_CTS_UPDATE_LATENCY &&
[file exists [which $ICC_CTS_LATENCY_OPTIONS_FILE]]} {
source -echo $ICC_CTS_LATENCY_OPTIONS_FILE
}


# ----------------------------------------------------------------------
# Creating floorplan
# ----------------------------------------------------------------------
if {$ICC_FLOORPLAN_INPUT != "DEF" } {
## Connect PG first before loading floorplan file or initialize_floorplan
if {[file exists [which $CUSTOM_CONNECT_PG_NETS_SCRIPT]]} {
source -echo $CUSTOM_CONNECT_PG_NETS_SCRIPT
} else
{
derive_pg_connection    -power_net $MW_POWER_NET \
-power_pin $MW_POWER_PORT \
-ground_net $MW_GROUND_NET \
-ground_pin $MW_GROUND_PORT \
-create_port top
}
}


# ----------------------------------------------------------------------
# Initialize floorplan - Surpriya Tike
# ------------------------------------------------------------------
if {$ICC_FLOORPLAN_INPUT == "CREATE"} {
if { [file exists [which $ICC_IN_PHYSICAL_ONLY_CELLS_CREATION_FILE]]} {
source $ICC_IN_PHYSICAL_ONLY_CELLS_CREATION_FILE}
if { [file exists [which $ICC_IN_PHYSICAL_ONLY_CELLS_CONNECTION_FILE]]} {
source $ICC_IN_PHYSICAL_ONLY_CELLS_CONNECTION_FILE}
if {[file exists [which $ICC_IN_TDF_FILE]]} {
read_pin_pad_physical_constraints $ICC_IN_TDF_FILE}


set initialize_floorplan_cmd "initialize_floorplan \
-left_io2core $LEFT_IO2CORE \
-bottom_io2core $BOTTOM_IO2CORE \
-right_io2core $RIGHT_IO2CORE \
-top_io2core $TOP_IO2CORE \
-row_core_ratio 1 "

if {$ICC_FLOORPLAN_CREATE_MODE == "DIMEN"}
{
lappend initialize_floorplan_cmd \
-control_type width_and_height \
-core_width $CORE_HEIGHT \
-core_height $CORE_WIDTH
} else
{
lapped initialize_floorplan_cmd \
-control_type aspect_ratio \
-core_aspect_ratio 1 \
-core_utilization $CORE_UTIL
}


echo $initialize_floorplan_cmd
eval $initialize_floorplan_cmd


# ------------------------------------------------------------------
# Power Rails and Stripes - Surpriya Tike
# ------------------------------------------------------------------
set POWER_SIGNALS           [ list $MW_POWER_NET $MW_GROUND_NET ];
set area_top                [get_attribute [get_core_area] bbox]
set area_top                [join $area_top]
set core_leftend            [lindex $area_top 0]
set core_rightend           [lindex $area_top 2]


create_rectangular_rings \
-nets $POWER_SIGNALS \
-offsets adjusted \
-left_offset 5 -right_offset 5 \
-bottom_offset 5 -top_offset 5\
-left_segment_width 5 -right_segment_width 5 \
-bottom_segment_width 5 -top_segment_width 5 \
-bottom_segment_layer M4 -top_segment_layer M4


create_power_straps \
-nets $POWER_SIGNALS \
 -direction vertical \
-configure groups_and_stop \
-layer M3 \
-width 3 \
-num_groups 5 \
-stop [expr $core_rightend -10] \
-start_at [expr $core_leftend + 10]
}


if {$ICC_FLOORPLAN_INPUT == "USER_FILE"} {
if {[file exists [which $ICC_IN_FLOORPLAN_USER_FILE]]} {
source $ICC_IN_FLOORPLAN_USER_FILE}}
if {[file exists [which $ICC_PHYSICAL_CONSTRAINTS_FILE]] } {
source $ICC_PHYSICAL_CONSTRAINTS_FILE}


## Also support for Well proximity effect (WPE) end cap cells
if {$ICC_H_CAP_CEL != "" } {
if {$ICC_V_CAP_CEL == ""} {
add_end_cap -respect_blockage -lib_cell $ICC_H_CAP_CEL
} else
{
add_end_cap     -respect_blockage -lib_cell $ICC_H_CAP_CEL \
-vertical_cells $ICC_V_CAP_CEL -fill_corner
}
}


source -echo ./SCRIPTS/common_optimization_settings_icc.tcl
source -echo ./SCRIPTS/common_placement_settings_icc.tcl


# ------------------------------------------------------------------
# Connect PG
# ------------------------------------------------------------------
if {[file exists [which $CUSTOM_CONNECT_PG_NETS_SCRIPT]]} {
source -echo $CUSTOM_CONNECT_PG_NETS_SCRIPT
} else
{
derive_pg_connection    -power_net $MW_POWER_NET \
-power_pin $MW_POWER_PORT \
-ground_net $MW_GROUND_NET \
-ground_pin $MW_GROUND_PORT
if {!$ICC_TIE_CELL_FLOW} {
derive_pg_connection   -power_net $MW_POWER_NET \
-ground_net $MW_GROUND_NET
-tie}
}
save_mw_cel -as $ICC_FLOORPLAN_CEL


# ------------------------------------------------------------------
# Floorplan Screeshot - Surpriya Tike
# ------------------------------------------------------------------
gui_start
set area_top                      [get_attribute [get_core_area] bbox]
gui_zoom                          -window [gui_get_current_window -types Layout -mru] \
-rect $area_top -fit
gui_zoom                          -window [gui_get_current_window -types Layout -mru] \
-fit -factor 0.7
gui_write_window_image            -window [gui_get_current_window -types Layout -mru] \
-file $SNAPSHOTS_DIR/$ICC_DP_CREATE_PLANGROUPS_CEL \
-format jpg
stop_gui


# ------------------------------------------------------------------
# Reports
# ------------------------------------------------------------------
if {$ICC_REPORTING_EFFORT != "OFF" } {
create_qor_snapshot -name $DBS/$ICC_FLOORPLAN_CEL
redirect -file $REPORTS_DIR_INIT_DESIGN/ $ICC_FLOORPLAN_CEL.qor_snapshot.rpt
{report_qor_snapshot -no_display}
}


if {$ICC_REPORTING_EFFORT != "OFF" } {
redirect -tee -file $REPORTS_DIR_INIT_DESIGN/$ICC_FLOORPLAN_CEL.sum
{report_design_physical -all -verbose}
set_zero_interconnect_delay_mode true
redirect -tee -file $REPORTS_DIR_INIT_DESIGN/$ICC_FLOORPLAN_CEL.zic.qor
{report_qor}
set_zero_interconnect_delay_mode false
set_check_library_options -all
redirect -file $REPORTS_DIR_INIT_DESIGN/check_library.sum {check_library}
}


exit
```

APPENDIX E

PRIMETIME SCRIPTS

Invoking PrimeTime

To invoke PrimeTime, use the following command from terminal -

```
=> pt_shell -64 –f [tcl_file_name].tcl
```

Script – pt.tcl

```
#/**************************************************/
#/* Source script for timing and power analysis    */
#/* PrimeTime                                      */
#/*                                                */
#/**************************************************/

# Please do not modify the sdir variable.
# Doing so may cause script to fail.
set sdir "."


source $sdir/SCRIPTS/common_setup.tcl
source $sdir/SCRIPTS/pt_setup.tcl


# make REPORTS_DIR
file mkdir $REPORTS_DIR
# make RESULTS_DIR
file mkdir $RESULTS_DIR
```

```
set read_parasitics_load_locations true
set power_enable_analysis true
set power_analysis_mode averaged
set report_default_significant_digits 3 ;
set sh_source_uses_search_path true ;
set search_path ". $search_path" ;


# -----------------------------------------------------------------
# Read Input Netlist(s)
# -----------------------------------------------------------------
set link_path "* $link_path"
read_verilog $NETLIST_FILES
current_design $DESIGN_NAME
link


# -----------------------------------------------------------------
# Power – annotate switching activity (VCD/SAIF)
# -----------------------------------------------------------------
#source $NAME_MAP_FILE
read_vcd -rtl $ACTIVITY_FILE -strip_path $STRIP_PATH
#read_saif  $ACTIVITY_FILE -strip_path $STRIP_PATH
report_switching_activity -list_not_annotated


# -----------------------------------------------------------------
# Back Annotation
# -----------------------------------------------------------------

if { [info exists PARASITIC_PATHS] && [info exists PARASITIC_FILES] } {
foreach para_path $PARASITIC_PATHS para_file $PARASITIC_FILES {
if {[string compare $para_path $DESIGN_NAME] == 0} {
read_parasitics -increment -format spef $para_file
} else
{
read_parasitics -path $para_path -format spef $para_file
}
}
}

report_annotated_parasitics -check >
$REPORTS_DIR/${DESIGN_NAME}_report_annotated_parasitics.report


#################################################################
#    Reading Constraints Section                          #
#################################################################
if  {[info exists CONSTRAINT_FILES]} {
foreach constraint_file $CONSTRAINT_FILES {
```

```
if {[file extension $constraint_file] eq ".sdc"} {
read_sdc -echo $constraint_file
} else
{
source -echo $constraint_file
}
}
}
if { [info exists AOCVM_FILES]} {
foreach aocvm_file $AOCVM_FILES {
read_aocvm $aocvm_file
}
}


#################################################################
#    Setting Derate and CRPR Section                            #
#################################################################
if { [string is double -strict $derate_clock_early_value ] } {
set_timing_derate $derate_clock_early_value -clock -early
}

if { [string is double -strict $derate_clock_late_value ] } {
set_timing_derate $derate_clock_late_value -clock -late
}

if { [string is double -strict $derate_data_early_value ] } {
set_timing_derate $derate_data_early_value -data -early
}

if { [string is double -strict $derate_data_late_value ] } {
set_timing_derate $derate_data_late_value -data -late
}


# -----------------------------------------------------------------
# Clock Tree synthesis section
# -----------------------------------------------------------------
set_propagated_clock [all_clocks]


# -----------------------------------------------------------------
# Power Analysis Section
# -----------------------------------------------------------------
check_power  > $REPORTS_DIR/${DESIGN_NAME}_check_power.report
update_power
report_power -hierarchy -levels 2 > \
$REPORTS_DIR/${DESIGN_NAME}_report_power.report


# -----------------------------------------------------------------
# Update timing and check_timing Section
# ------------------------------------------------------------------
update_timing -full
# Ensure design is properly constrained
check_timing -verbose > $REPORTS_DIR/ct.report


# ------------------------------------------------------------------
# Report timing section
# ------------------------------------------------------------------
report_timing    -slack_lesser_than 0.0 \
-delay min_max -nosplit \
-input -net -sign 4 > $REPORTS_DIR/rt.report
report_clock                 -skew -attribute > $REPORTS_DIR/rc.report
report_analysis_coverage > $REPORTS_DIR/rac.report


quit
```

VITA

Surpriya Tike

Candidate for the Degree of

Master of Science

Thesis: TABLE DRIVEN ADAPTIVE, EFFECTIVELY HETEROGENEOUS MULTI-CORE ARCHITECTURE

Major Field: Electrical Engineering

Biographical:

Education: Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December 2011. Received Bachelor of Engineering in Electronics and Telecommunication at University of Pune, Pune, Maharashtra, India in August 2007.

Experience: Employed by Oklahoma State University, Department of Electrical and Computer Engineering as a graduate teaching assistant in Fall 2009, Spring 2010 and Fall 2010, and as a graduate research assistant; Oklahoma State University, Department of Electrical and Computer Engineering in Spring 2011 and Fall 2011. Worked as a hardware design engineer at Aftek Ltd., Pune, India from August 2007 to May 2009.

Professional Memberships: Student Member, IEEE

Name: Surpriya Tike Date of Degree: December, 2011

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: TABLE DRIVEN ADAPTIVE, EFFECTIVELY HETEROGENEOUS MULTI-CORE ARCHITECTURE

Pages in Study: 95 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study:

Exploiting flexibilities and scope of multi-core architectures for performance enhancement is one of the highly used approaches used by many researchers. However, with increasing dynamic nature of the workloads of everyday computing, even general multi-core architectures seem to just touch an upper limit on the deliverable performance. This has paved way for meticulous consideration of heterogeneous multi-core architectures. Such architectures can be further enhanced, by making the heterogeneity of the cores dynamic in nature.

This work proposes techniques, which change configurations of these cores dynamically with workload. Thus, depending on requirements and pre-programmed preferences, each core can arrange itself to be power optimized or speed optimized. In addition, the project has been designed using RTL (Verilog) to provide completely realistic grip on the silicon investment. The project can be simulated using SPEC2000 and SPEC2006 benchmarks, and is completely synthesizable using IBM_LPE library for 65nm (IBM65LPE).

ADVISER'S APPROVAL: Dr. James E. Stine