# SYNTHESIS OF THE KESTREL MULTISCALAR PROCESSOR

by

**Padmaja Nandula**

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science

(Electrical and Computer Engineering)

at the

**UNIVERSITY OF WISCONSIN – MADISON**

1998

I certify that I have read this thesis and certify that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Master of Science.

Advisor: ____________________ Date: ____________________

Gurindar S. Sohi

# Dedication

To

అమ్మా, తాతగారు, అమ్మమ్మా, ఆనన్ద్, ఆకాశ్...

# Abstract

Majority of the current day software is written with the assumption of sequential execution. State-of-the-art microprocessors, therefore, search for independent instructions in a dynamic instruction window and attempt to execute them simultaneously (*ILP*). The multiscalar processors, on the other hand, extract parallelism by dividing a program into tasks. The Kestrel processor is an implementation of this multiscalar paradigm.

This thesis details techniques used for the synthesis and static timing analysis in the context of the Kestrel multiscalar processor. The issues related to the synthesis of the Kestrel processor are discussed in detail in this thesis.

Various tools and scripts were used to automate the process of synthesis. The *ASIC* synthesis of large blocks of the design, described using the Verilog model, served as an input to the floorplanning tool. Interconnect delays and timing information modeled on the basis of data extracted from the high-level floorplans were used to specify design constraints for different submodules. The constraints specified on the design included input and output delays, input loads, and output drive constraints. Various compile-time optimization options of flattening and structuring the logic have been used and studied. Final timing estimates were based on the reports obtained after static timing analysis using *Primetime* and *Design Analyzer*. The delay (*SDF*), and parasitic capacitance information (*SPEF*) extracted from the floorplan was used to specify the delay and load constraints on the ports.

The target clock cycle time for the Kestrel processor in a $0.5\mu$ technology was $12ns$. After several iterations of the synthesis and design process, majority of the design

(except data memory system) satisfies a clock of $15ns$. The data memory system ($DMS$) is currently at a clock cycle time of $16ns$ and is being constantly optimized to get closer to the target cycle time of $12ns$. The area of the processor designed is estimated at $16cm^2$.

# Acknowledgements

I would like to thank my adviser and mentor Prof. Sohi and Prof. Smith for giving me the opportunity to work on this immensely interesting project and for their help and guidance throughout my Master's program.

Thanks also to Prof. Kime and Prof. Parmesh Ramanathan who played a key role in the implementation of the Kestrel project. My special thanks to the Kestrel team members and occupants of 3652, Anand, Matt, Eric, Rick, Quinn, Sastry, Tim, Craig, Zheng Li, and Yanos for helping me with various things at different stages of the project and making my stay at UW-Madison a memorable one. Thanks to Bruce Orchard and the people at the computer systems laboratory for their help with the tools and hardware throughout the course of the project. Finally, thanks to the faculty of the ECE department, Prof. Saluja, Prof. Chalasani, for helping me shape my future.

This work was supported in part by NSF Grant MIP-9505853 and by the U.S Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The view and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of the U.S Army Intelligence Center and Fort Huachuca, or the U.S. Government.

# Abbreviations

| Abbreviation | Definition |
|---|---|
| $\mu$ | micron |
| *cm* | centi-meter |
| *i-cache* | Instruction Cache |
| *mux* | multiplexor |
| *muxes* | multiplexors |
| *ns* | nano-second |
| *ALU* | Arithmetic Logic Unit |
| *ARB* | Address Resolution Buffer |
| *ASIC* | Application Specific Integrated Circuit |
| *BHT* | Branch History Table |
| *BIU* | Bus Interface Unit |
| *BTB* | Branch Target Buffer |
| *D$* | Data Cache |
| *DMS* | Data Memory System |
| *DMU* | Data Miss Unit |
| *DRF* | Distributed Register File |
| *EX* | Execute |
| *HDL* | Hardware Description Language |
| *I$* | Instruction Cache |

Continued...

| Abbreviation | Definition |
|---|---|
| *ID* | Instruction Decode |
| *IF* | Instruction Fetch |
| *ILP* | Instruction Level Parallelism |
| *IMU* | Instruction Miss Unit |
| *IPC* | Instructions Per Cycle |
| *L1* | Level one |
| *L2$* | Level two Cache |
| *LDHD* | Load Header instruction |
| *LFB* | Line Fill Buffer |
| *MEM1* | First stage of memory access |
| *MEM2* | Second stage of memory access |
| *PC* | Program Counter |
| *PE* | Processing Element |
| *PE_INTEG* | Processing Element with all stages of the pipeline |
| *PLB* | Primary Lookaside Buffer |
| *PLBMU* | PLB Miss Unit |
| *SDF* | Standard Delay Format |
| *SEQ* | Sequencer |
| *SPEF* | Standard Parasitic Exchange Format |
| *SPRU* | Special Purpose Register Unit |
| *RCL* | Restart Control Logic |
| *T$* | Task Cache |

Continued...

| Abbreviation | Definition |
| --- | --- |
| *TLB* | Translation Lookaside Buffer |
| *TLBP* | TLB Probe |
| *VLIW* | Very Large Instruction Word |
| *WB* | Write Back |

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Multiscalar Paradigm

Programs are traditionally written with the assumption that they would be executed in the same order as the instructions appear in the program. To achieve higher performance, modern day processors attempt to execute multiple instructions simultaneously (instruction level parallelism). Superscalar and *VLIW* processors belong to this category of machines that identify instruction level parallelism and exploit it to improve performance.

The Multiscalar paradigm offers a new approach to the problem of executing multiple instructions at a time. The Multiscalar paradigm [3] is a processing paradigm for extracting large quantities of instruction level parallelism from ordinary high level language programs.

The Multiscalar approach divides a sequential program into tasks, where a task could be, a part of a basic block, an entire basic block, multiple basic blocks, or a loop iteration. A task is assigned to one of the processing elements with the initial program counter of the task. This enables multiple tasks to be executed in parallel on various processing elements, resulting in a higher execution rate.

Multiscalar processors can be described as a collection of processing units with a sequencer assigning tasks to the processing units as shown in Figure 1.1. Key to the multiscalar paradigm is communication of data and control information among

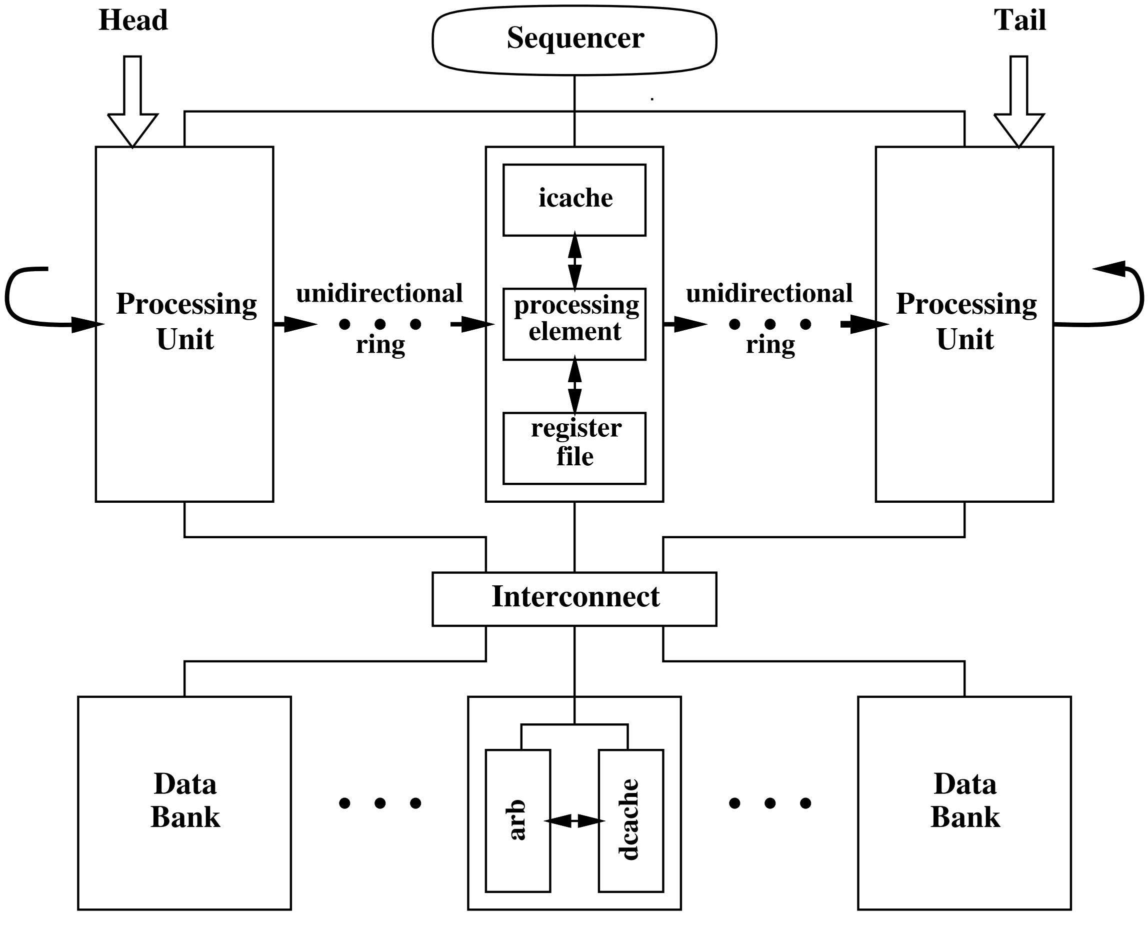


Figure 1.1: Schematic Diagram of Multiscalar Processor [3]

the parallel processing units. To maintain the appearance of sequential execution a loose sequential order is enforced on the processing elements by organizing them as a unidirectional ring (thus enforcing an order on the tasks executed). Multiscalar processors perform control as well as data speculation. The tasks are committed from the queue in same order as they are assigned, thus ensuring sequential semantics. For memory operations, an Address Resolution Buffer (*ARB*) is implemented to hold speculative operations, detect violations and initiate corrective actions.

Multiscalar processors use a combination of hardware and software techniques to achieve the goal of high *ILP*.

## 1.2 Kestrel Implementation

The Kestrel Architecture [4, 6] is an implementation of the multiscalar paradigm described in section 1.1. Top level diagram of the Kestrel processor is shown in figure 1.2. This figure shows the various Verilog modules at different levels of hierarchy in the design.

The following sections give a brief sketch of the design of various units[1] (done using Verilog HDL). The synthesis details of these units would be discussed in subsequent sections.

### 1.2.1 Processing Element

The Processing elements (figure 1.2) are single issue, in-order processing cores. The Kestrel implementation of multiscalar paradigm has four processing elements in a ring

[1]This description has been taken in part from the Kestrel Design Specification Manual.

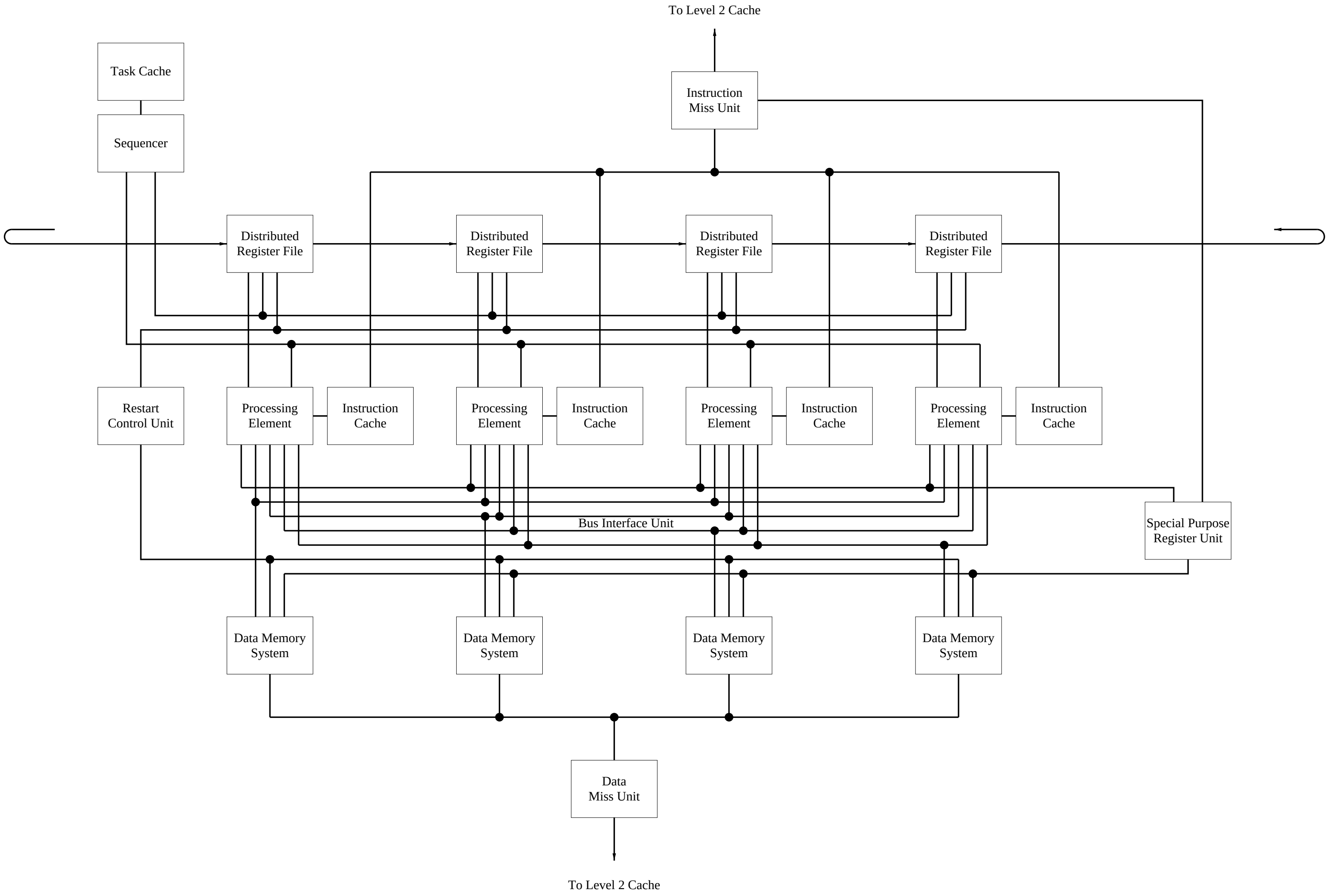


Figure 1.2: Top Level Diagram of the Kestrel Processor [4]

network. Tasks are assigned to each of these *PEs*, speculatively or non-speculatively. If the speculated task assigned to any of the *PEs* turns out to be misspeculated, the entire work done by that *PE* is squashed to ensure correct operation.

Each processing element has its own instruction cache and register file. The instruction cache (*I$*) has been synthesized as a part of the processing element. The processor state exists in both the distributed register file (*DRF*) and special purpose register unit (*SPRU*). The processing elements access the memory subsystem through the data memory system (*DMS*).

### 1.2.2 Instruction Miss Unit

The instruction miss unit (shown in figure 1.3) handles communication between the level one (*L1*) instruction caches and the level two (*L2*) unified cache. The *L1* instruction misses are sent to the instruction miss unit (*IMU*) from each *PE*, and in the *IMU*, they are arbitrate for the miss handler with the head getting the highest priority and others depending on their position in the ring from head to tail.

The instruction miss handler (figure 1.3) consists of a 16 entry line-fill buffer (*LFB*) that holds recently requested cache lines to prevent unnecessary compulsory misses. One line of the *LFB* consists of 284 bits (256 bits cache data, 27 bits address tag, and a valid bit).

### 1.2.3 Bus Interface Unit

The bus interface unit (*BIU*) behaves like a crossbar between the *PEs* and the *DMS* banks for load, store, cache and *TLB* requests. This unit accepts inputs from all the

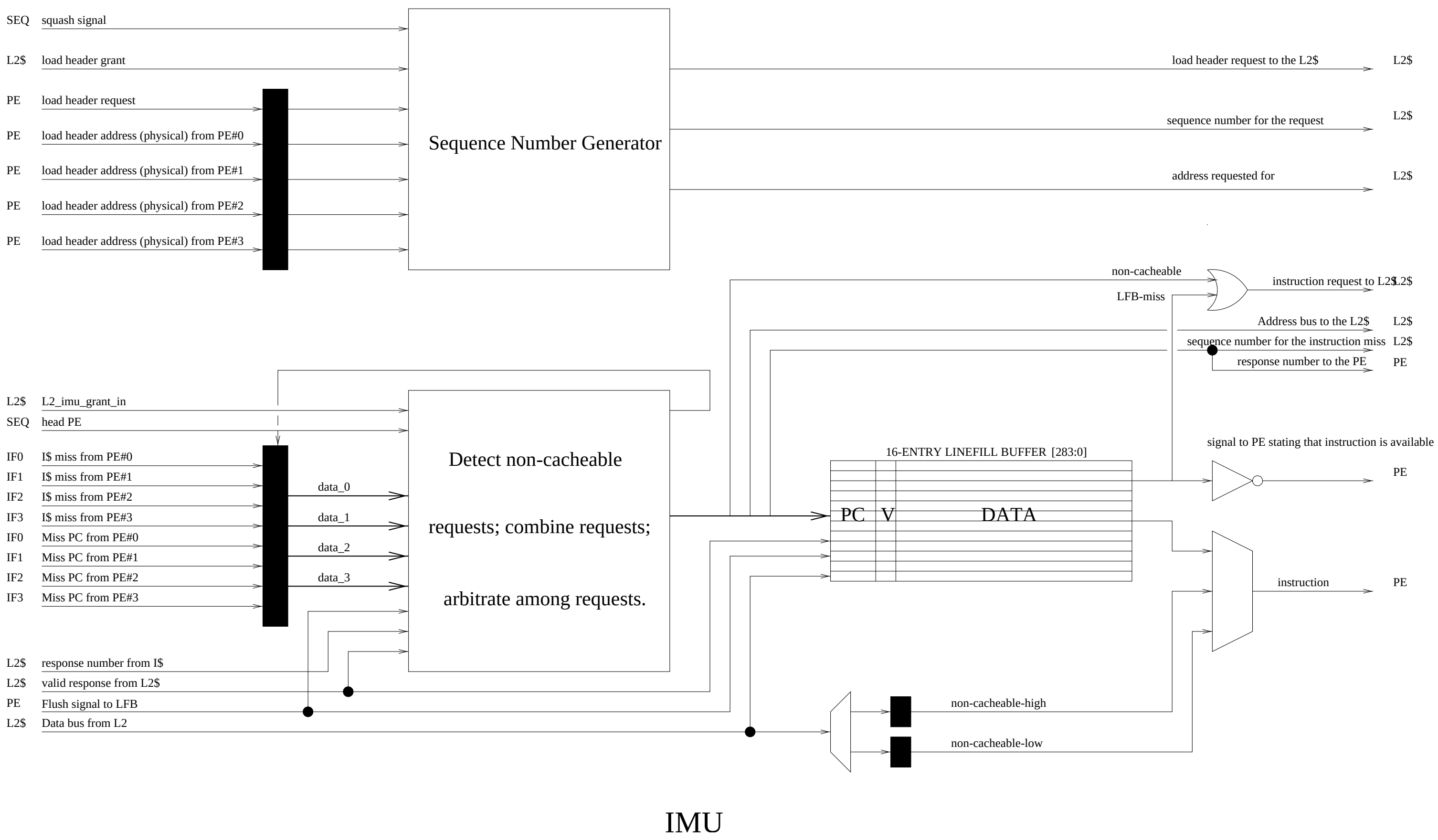


Figure 1.3: Top Level Diagram of the instruction miss unit [4]

*PEs* and has interfaces with all the *DMS* banks. The *BIU* arbitrates among the *PE* requests, with the priority depending on the proximity of the *PE* to the head. *BIU* also handles *TLB* requests to the *DMS* banks.

### 1.2.4 Special Purpose Register Unit

The special purpose register unit contains 32 registers of different names that store the centralized processor state. One of the special purpose registers is the Suppress register that is replicated for each of the processing elements. Other registers appear once in the design.

This unit also contains the translation lookaside buffer (*TLB*) logically, as it handles accesses and updates to the *TLB*.

### 1.2.5 Data Memory System

The data memory system consists of four interleaved memory banks. These banks consist of the data cache (*L1*), *ARB*, and the *TLB*. The data memory system manages all the load and store requests from the *PEs*.

### 1.2.6 Data Miss Unit

The data miss unit (*DMU*) handles communication between the data memory system and the level two cache.

### 1.2.7 Sequencer

The sequencer initiates both speculative and non-speculative tasks on the *PEs*. *SEQ* also handles commits and squashes of tasks. Additionally, the sequencer reads and writes the task header information from and to the task cache (*T$*). The restart control logic (*RCL*), synthesized as a submodule of sequencer, orchestrates the restarting of the processing elements due to misspeculated data accesses (Both memory and register misspeculations).

### 1.2.8 Distributed Register File

The distributed register file (*DRF*) serves as a local register file for a given processor and handles register value communication and synchronization between processing elements.

## 1.3 Micro-architecture Parameters

Table 1.1 summarizes the key micro architecture parameters used in the design of the Kestrel processors.

## 1.4 Synthesis

Synthesis is the process of generating gate-level netlist for a design from its *HDL* description. Synthesis includes both reading in the source Verilog code and optimizing it. Synthesis terminology and the tools used are described in Appendix C.

After designing and verifying the functionality of the processor, it is essential to

| Parameter | Value |
|---|---|
| No. of *PEs* | 4 |
| *I$* width | 256 bits |
| *I$* depth | 512 lines |
| *I$* address width | 9 bits |
| *BTB* width | 50 bits |
| *BTB* depth | 1024 lines |
| *BTB* address width | 10 bits |
| *BHT* width | 2 bits |
| *BHT* depth | 1024 lines |
| *BHT* address width | 10 bits |
| *Data Cache(D$)* width | 128 bits |
| *Data Cache(D$)* depth | 2048 lines |
| No. of registers in *DRF* | 32 |

Table 1.1: Micro-architecture specifications of the Kestrel processor

evaluate the feasibility of the design. This was done by performing a detailed timing and area analysis of the design.

The Verilog model was synthesized on a hierarchical basis. All the submodules were synthesized with estimated input and output constraints. These units were further linked together to get the top level processor estimates. The goal of synthesis was to evaluate the processor in the context of timing. The synthesis was carried out using compile time synthesis options of structuring and flattening the logic. The details of synthesis in general and the synthesis of the Kestrel processor in particular are discussed in detail in this thesis. The work presented here is a group effort and references have been given at appropriate places.

This thesis is organized into several chapters. Chapter 2 describes the steps of the synthesis process for the Kestrel multiscalar design. Timing reports generated using *Synopsys* and *Primetime* (refer Appendix C) were used to identify critical paths in the design. This information was further used to optimize the design to satisfy the timing requirements. A description of the nature of these changes and some specific changes have been listed in chapter 3. Chapter 4 presents the results of the synthesis process. Finally, the conclusions and future directives are given in chapter 5.

Some of the scripts used in the project are given in Appendix B. Description of the tools, reporting techniques and synthesis terminology used throughout the thesis are covered in Appendix A and C.

# Chapter 2

# Synthesis of the Kestrel Processor

Synthesis is the process of obtaining a gate level netlist for the design and analyzing it for timing and area requirements. The Verilog code is the primary input to the synthesis process. The synthesis process also takes as input the design environment, constraints, design rules, technology libraries, and compile strategy. Synthesis with the above parameters results in a gate-level netlist describing the design. Timing reports on the netlist are then debugged and analyzed to determine if the constraints are met. If not, the constraints and/or the HDL code are modified and synthesized again.

The next step is to perform a floorplan, that gives the standard delay and the parasitic capacitance information. This information is then back annotated onto the design to perform post-layout simulation and timing verification. If the results are not within the design goals, requisite changes are made and the synthesized again till the goals are achieved.

For a large design it is therefore necessary to perform many passes of the synthesis process, fine tuning the parameters based on the timing reports and area reports.

The synthesis of the Kestrel processor also went through several iterations of synthesis and design changes to attain the required cycle time of $12ns$. *Synopsys* and *Cadence* design tools had been used for modelling and synthesis of the design.

Figure 2.1 shows the flow of the physical design for the Kestrel Processor. The description of each of the steps shown in the figure has been presented in the following sections.

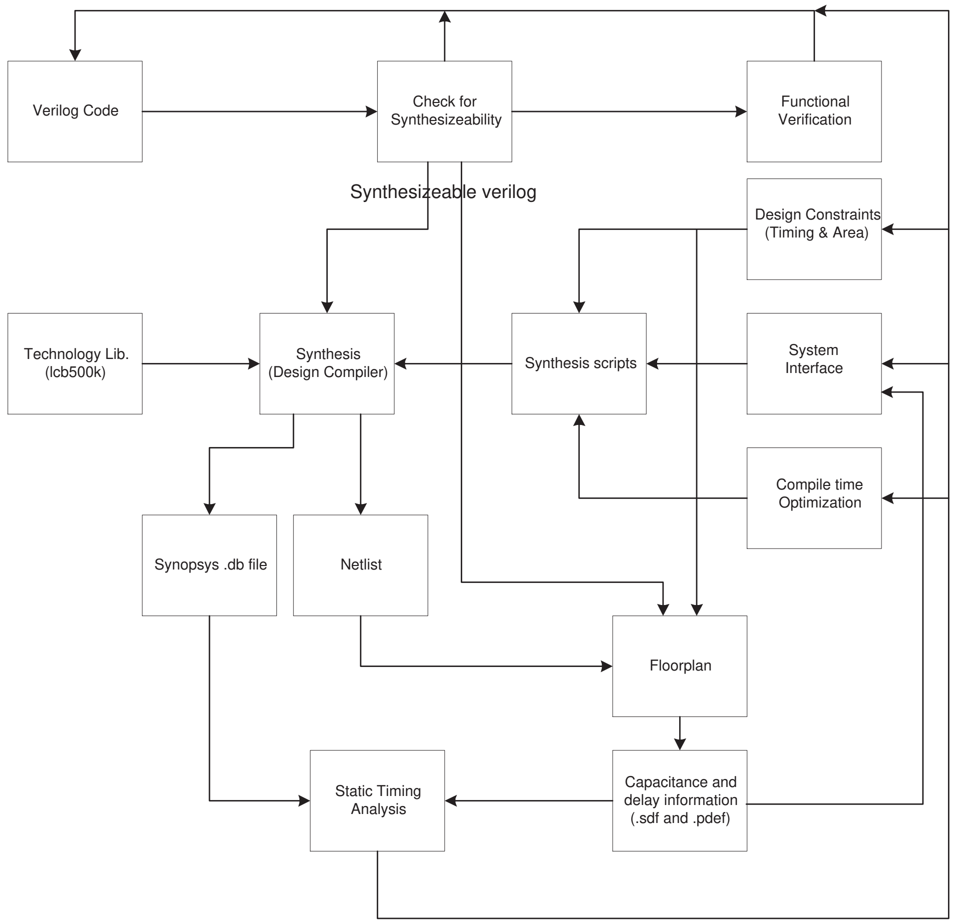


Figure 2.1: Physical design flow of the Kestrel processor

## 2.1 Check for Synthesizeability

Check for synthesizeability entails verifying that the Verilog code is completely synthesizeable and also partitioned into logical blocks (or modules) for the purpose of synthesis.

**Synthesizeability** To make the Verilog synthesizeable the library cells and memory cells were mapped to the library(lcb500k) [7] being used. The entire processor is currently synthesizeable. Only non-synthesizeable units are the L2 cache and the multiply, divide, and floating point units in the processing elements.

**Partitioning** The Kestrel design is completely hierarchical and the entire design was divided into twenty nine submodules. Various parameterizable library modules for flip-flops, multiplexors and memory units were written in Verilog for the convenience of the designers. Additional regrouping was done for the purpose of physical design. The bidirectional busses were made a separate module[1], and glue logic at the top-level was grouped into a submodule. Figure 2.2 shows the hierarchical structure of the Kestrel design.

Partitioning of the design helped achieve good synthesis results, reduce compile time and simplify constraints and script files.

## 2.2 Technology Libraries

Technology libraries are used to implement a gate-level design. These libraries consist of a set of vendor-specific cells, their names, delays and pin loading. Design rule constraints are also specified in the technology library.

The Kestrel processor was synthesized using the $0.5\mu$ *LSI-Logic lcb500k* [7] libraries. The library cells were mapped to the *lcb500k* libraries for synthesis. Design rule constraints for maximum fanout and maximum capacitance were also specified in the synthesis scripts for high fanout signals, as the synthesis was carried out in a

[1] Details in Chapter 3.

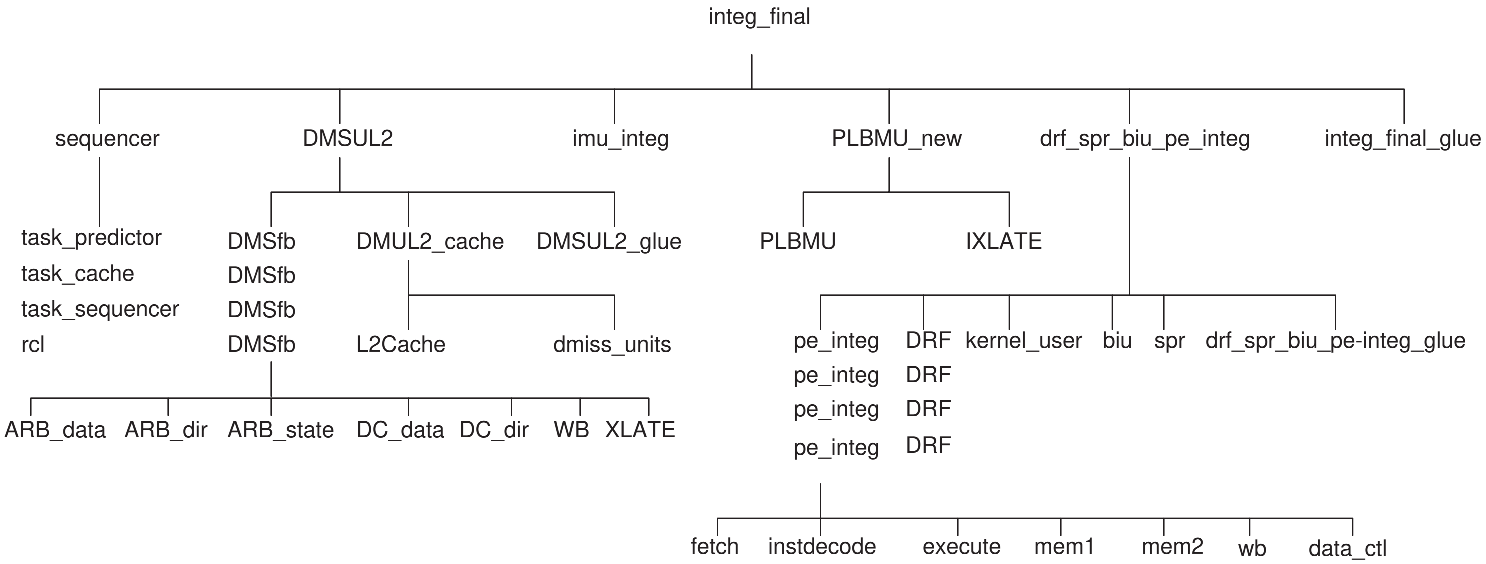


Figure 2.2: Logical Hierarchy of Kestrel design

hierarchical fashion and compilation was not done at the topmost level. This enabled synthesis tools to insert appropriate buffers as the fanout outside the boundary of the submodule under consideration was not known to the tool.

## 2.3 Synthesis Scripts

Apart from the technology libraries and synthesizeable Verilog, synthesis scripts, specifying the constraints, interface, and compile time options, also serve as an input to the synthesis process. An outline of the components comprising the synthesis scripts has been given below.

### 2.3.1 System Interface

System interface refers to the attributes and constraints that define the interaction of various modules and the environment surrounding the design submodules.

**Wire Load Models** Initially the custom wire load models created using the data from the floorplan were used for synthesis. These wire load models were later modified on the basis of the net load estimations made using path timing and net capacitance and loading information [2].

**Submodule Interfaces** The interface of the various submodules was specified by defining the drive characteristics for the input ports. Loads obtained from the extracted capacitance and delay information after floorplan were used to specify estimated loads on the output ports. Further, fanout loads on the output ports were specified to handle the buffering, in case of high fanout outside the sub-module being synthesized. Buffer tress were automatically generated by *Design Compiler*[2] for signals with high fanout and loads.

It is important to note that the drive strength for heavily loaded driving ports, such as clock, bus error and reset was left at zero, so that the synthesis tool does not buffer these nets.

### 2.3.2 Design Constraints

Design constraints refer to circuit characteristics, such as timing and area, that define the synthesis goals. The design constraints are used to optimize and implement the design.

**Timing Constraints** Timing constraints indicate the required performance of the design. The Kestrel processor was synthesized for a cycle time of $12ns$. The Kestrel synthesis process assumes a perfect clock with no skew. Further, input and output delay constraints were specified to achieve the clock cycle requirements. These constraints were estimates, that were rectified based on the timing reports obtained. Point-to-point exceptions were also identified and specified as false paths. Finally, scripts were written to apply these constraints and synthesize.

**Area Constraints** Design area consists of the area of each component and the net. Area constraints are specified to limit the area of the entire design. Current effort of the Kestrel processor is to satisfy the cycle time requirements. Though

[2]Details of the tools used are given in Appendix C.

the design of the processor was done with area considerations in mind, no area constraints were specified in the synthesis of the design.

### 2.3.3 Compile-time Optimizations

Compile-time optimizations include selecting the compile strategy and then optimization of the design.

#### Compile Strategy

Compile strategy has to be determined prior to constraining or compiling the design. The selection of a compile strategy depends on the characteristics of the design to be synthesized. Figure 2.3 shows the strategy to be used depending upon the features of the design.

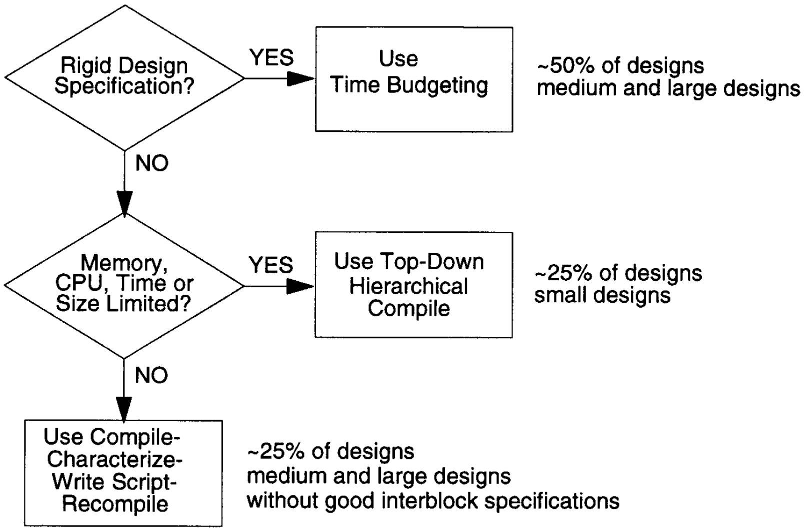


Figure 2.3: Compile Strategy Decision flowchart [5]

Kestrel is a large design organized in a hierarchical manner. The compilation for Kestrel was done at the submodule level using the estimates for output drive and input load. The top level integration was done by linking all the submodules. Synthesis optimization was performed at the lower levels and not the top most level. This approach was selected because of memory limitations. The compilation of the entire Kestrel design at the top level would consume a lot of memory and was not physically feasible.

### Design Optimization

The process of optimization consists of resolving multiple instances and using various optimization cost functions to achieve required results. It is necessary to resolve multiple instances because many designs reference a sub-design[3] more than once, and the environment for each instance might be different.

The Kestrel design has been optimized by iteratively synthesizing the submodules with different compile time optimizations. Various techniques of resolving multiple instances are:

- **Uniquify** copies and renames the design for each instance. This enables optimization of every instance in a way based on its environment. This method was used to resolve multiple instances if the environments around the design instances differed significantly.

- **Ungroup** is similar to uniquify but also removes levels of hierarchy. The method of ungrouping results in a flattened net-list. This approach provides the best synthesis results, but with more memory and longer synthesis time.

[3]For example: an adder submodule being referenced more than once in the processor pipeline.

- **Compile-once-don't-touch** attribute preserves the sub-design during optimization. This technique uses less memory, but the objects that have don't-touch set cannot be ungrouped. It was used in the Kestrel synthesis at the higher levels of hierarchy, where similar environments surround the design instances. For instance, All the *PEs* have a similar interface.

Uniquify and Compile-once-don't-touch options have been used at higher levels of the hierarchy because they require less memory. Ungroup, on the other hand, has been used at lower levels of the hierarchy (like the stages of the processor pipeline) to flatten the logic. The techniques used for the Kestrel processor are shown in Figure 2.4.

After resolving multiple references, logic-level optimization and gate level mapping of the design was performed using the *Design Analyzer* compile commands. The flattening and structuring options control the logic level optimization and compile command options control the gate level mapping. Figure 2.5 shows the commands used at different levels of the design hierarchy. The map effort was maintained at *medium* in majority of the cases.

The options for logic level optimization are:

- **Structuring** is an optimization technique that identifies sub-functions that can be factored and turns them into intermediate variables to reduce design equations.

  Structuring results in a reduction in area but not necessarily a fast circuit. Structured logic can further be optimized using boolean optimizations and timing driven optimization. Timing driven structuring does structuring only where the

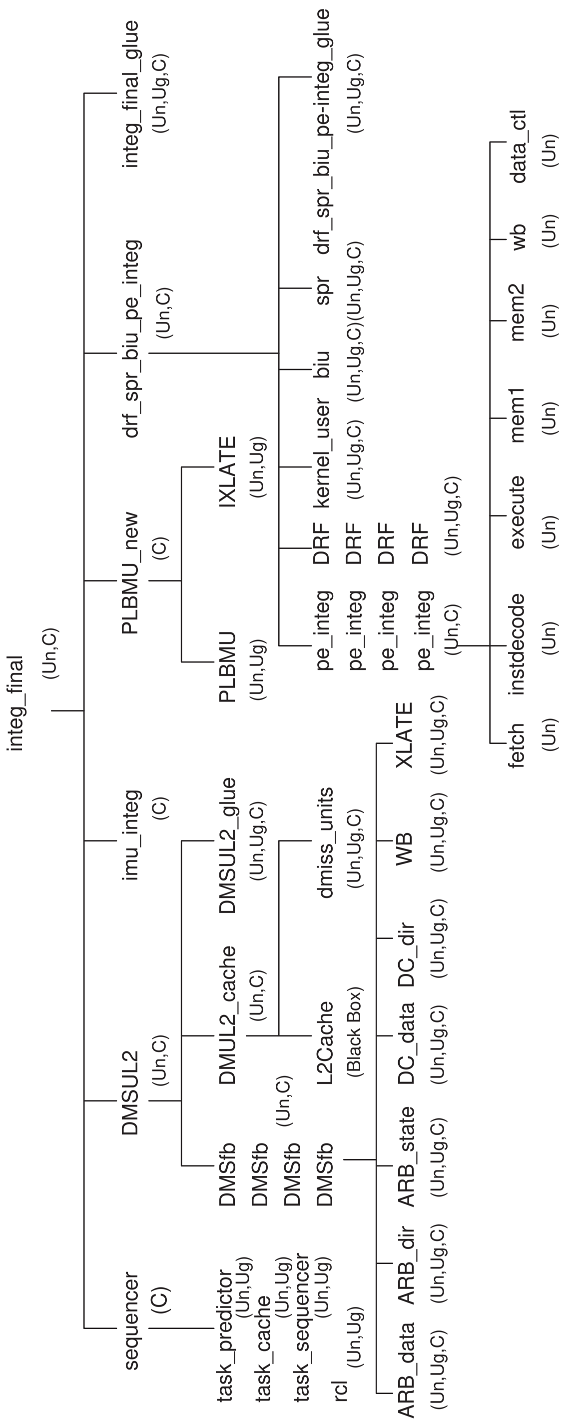


Figure 2.4: Methods used for resolving multiple instances in the Kestrel design. Ug: ungroup; Un: Uniquify; C: Compile-once-don't-touch

timing constraints are not critical.

- **Flattening** is an optimization technique that attempts to convert the design into a two-level sum-of-products representation. Flattened logic is generally fast because it is just two levels of combinational logic, but requires a large amount of CPU time and area.

  Flattening results in a faster circuit with larger area, whereas, structuring gives a slower circuit with lesser area. These techniques are thus a compromise between area and speed where a reduction in delay (increase in speed) results in an increase in area as shown in Figure 2.6.

  Flattening should therefore be used in more critical paths and structuring where area becomes a more critical issue over time.

- **Boundary Optimization** is a feature that enables optimization across boundaries. This option was used for synthesis at all levels where compilation was done for the Kestrel processor.

## 2.4 Synthesis

This block takes as input the technology libraries, synthesizeable Verilog, and the synthesis scripts. The outputs of the synthesis process are the netlist and the *Synopsys* database files (.db[4]), that were used for floorplan and static timing analysis.

[4] A format in which the *Design Compiler* saves the synthesized design.

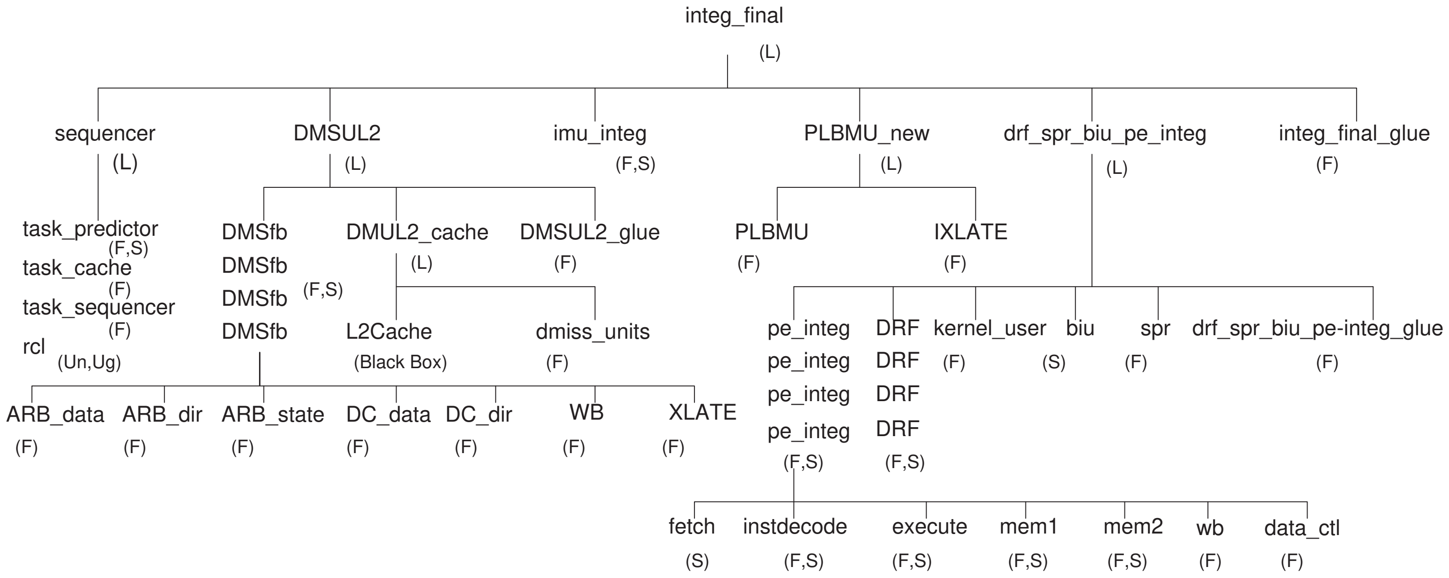


Figure 2.5: Compile commands used in the Kestrel design. S: Structure; F: Flatten; L: Link

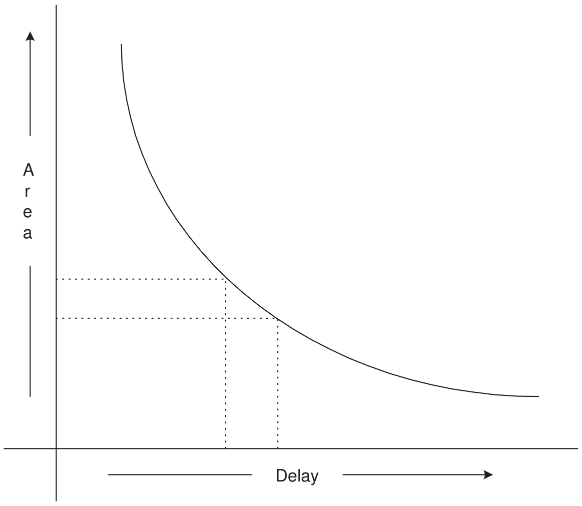


Figure 2.6: Design Space curve

## 2.5 Floorplan

The Cadence *floorplan manager* was used to place and route the Kestrel design [2]. The netlist obtained from synthesis was used to route the design. The final floorplan of the processor is shown in Figure 2.7. The small blocks in the floorplan that are not labelled are the bidirectional bus units and the glue logic.

The standard delay information and parasitic capacitances were extracted from the floorplan and used for static timing analysis.

## 2.6 Static Timing Analysis

The final step in the process of synthesis is analysis of timing and design problems. Preliminary analysis was to check the design for the presence of any unmapped components or unresolved references. Having verified that there were no such errors, gate level netlist obtained after synthesis was used to extract timing reports. Other than

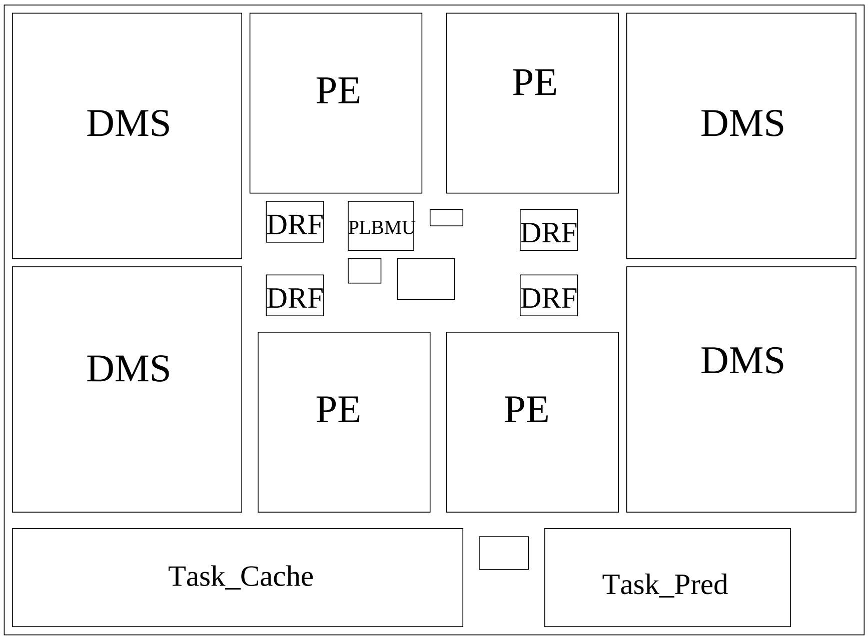


Figure 2.7: Floorplan of the Kestrel processor [2]

*Design Analyzer*, *Primetime* was used for the purpose of reporting. Appendix C lists the commands used for static analysis in the tools. These reports were studied to determine if the design met the specified constraints.

Critical paths were identified and design optimizations made to meet the timing specifications. The results of static timing analysis were used to redefine/refine the timing constraints and compile time optimizations.

Thus the process of design, constraints specification and synthesis was performed iteratively until the desired results were obtained. The following chapter discusses in detail the key design changes made for the purpose of synthesis. The results of timing analysis are given in the subsequent chapters.

# Chapter 3

# Design Optimizations for Synthesis

The synthesis process required the Verilog model to be organized according to the physical hierarchy. Also, timing reports generated after synthesis highlighted several critical paths that did not meet the cycle time constraints. This therefore led to several design optimizations in the Verilog model.

This chapter covers the key optimizations made to the various modules of the Kestrel processor. The implications of these changes on synthesis and functionality of the units has also been discussed. The changes have been organized into groups based on the unit or module they effect.

## 3.1 Top-Level

This sections includes the changes that effect all or most of the submodules in the design.

### 3.1.1 Bidirectional Busses

This is an example of difference in logical and physical hierarchy. In the Verilog design, the inputs and outputs of the bidirectional busses were in different modules. The presence of these three-state drivers resulted in a large number of false paths that could not be physically specified. This problem was noticed in the *BIU*, that had a bidirectional bus coming in/going out corresponding to each of the *DMS* banks and

*PEs.*

All the bidirectional busses could logically be represented as a pair of busses, one read and one write with a write enable. The Verilog model was therefore changed, to add one input port (read bus) and two output ports (write enable and write bus), in place of the bidirectional busses. The bidirectional busses were a different module implemented separately for functional verification and physical synthesis.

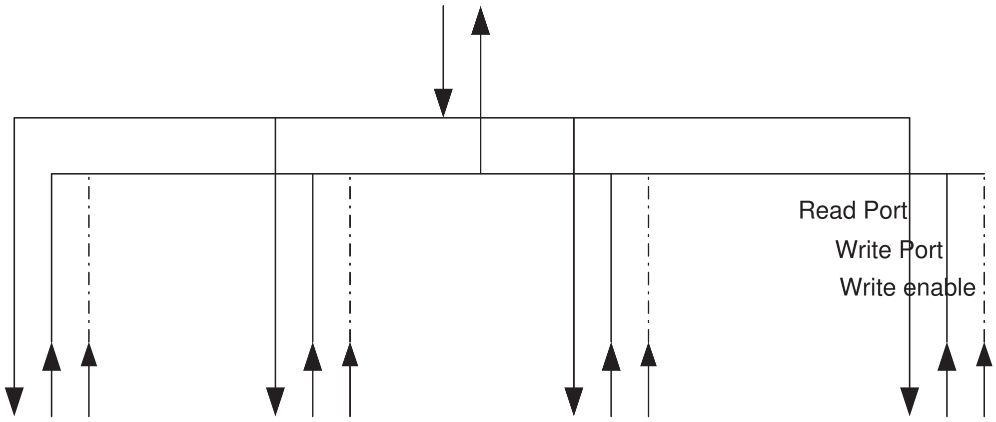


Figure 3.1: Model of bidirectional bus [2]

Figure 3.1 represents the logical representation of the bidirectional busses, where each pair of data busses were actually considered as one bus for physical synthesis purposes.

### 3.1.2 Ring Order

The processing elements and the distributed register file are organized as a directed ring in the multiscalar processor. The convention for the direction of this ring was changed from 0-1-2-3 to 3-2-1-0. Though this change did not effect the synthesis results directly, it brought about several changes in the design of the Kestrel implementation.

After reset, the first task was initially assigned to *PE#0* according to the old convention, whereas it was changed to *PE#3* in the new convention. Major change was in places where the order of the *PEs* mattered. The arbitration logic was implemented as the head *PE* getting the highest priority and the other *PEs* according to their distance from the head. With the change in the ring order the logic had to be changed.

It is important to note that these changes did not bring about differences in the static timing analysis, but called for the re-synthesis of the modules as the Verilog had changed.

### 3.1.3 Incrementers

The Verilog implementation of Kestrel consisted of several incrementers implemented as a behavioral model. Synthesis tools generated ripple carry adders to represent these incrementers. These incrementers were very slow and were on the critical path of the design.

Carry look ahead adders of different widths were therefore designed to replace all the incrementers in the design.

### 3.1.4 Disable *write enable* to *data output* Path in *SRAMS*

The data being read on the output port does not depend on the write signal. Furthermore, the design specifies that only one cache access is performed in a clock cycle. But, the synthesis tool would report the path from the *write enable* signal to the data output as critical.

Such a path should not exist in the design, therefore it was disabled for timing purposes.

This problem could have been solved by doing case analysis for various combinations of read and write accesses or disabling the write enable to data output path in the *SRAMS*. The second option was chosen as the number cases to be considered were too high.

## 3.2 Instruction Miss Unit

The instruction miss unit (block diagram shown in Figure 3.2) consists of a line-fill buffer that has sixteen most recently accessed cache lines. The line-fill buffer is thus a fully associative cache that has 128-bit wide cache lines. In case of an *i-cache* miss the sixteen entries are searched for a match before going to the *L2* cache.

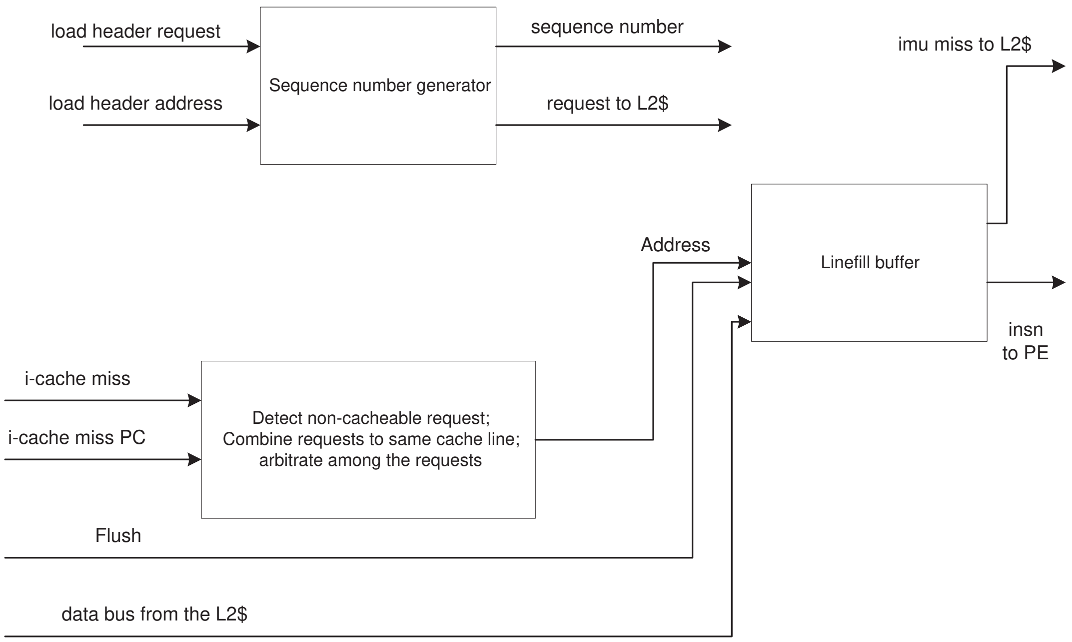


Figure 3.2: Block diagram of the instruction miss unit

The buffer logic is implemented as a collection of 16-to-1 muxes, each 128 bit wide. Initially these muxes were implemented totally in logic and the synthesis tool could not optimize it properly. To solve this problem a library module for a 16-to-1 mux was designed and mapped to technology cells to speed the synthesis process and get better results.

## 3.3 Processing Element

This section covers the key modifications made to the design of the processing element to meet the specified timing constraints.

### 3.3.1 Memory Instructions in the Execute Stage

The load and store memory requests arbitrate for the bus in the execute stage of the pipeline. The result of this request is available in two cycles (*MEM2* stage), if there is a hit in the *ARB* (address resolution buffer) or the data cache.

In the initial Kestrel design, a request was sent to the *DMS* only if the instruction was not stalling and didn't raise an exception. The stall and exception are known much later in the cycle, as stall in-turn depends on the earlier request being serviced.

Consider the scenario of three back to back memory instructions (shown in Figure 3.3 & 3.4). The first memory instruction would stall in the *MEM2* stage until it receives a hit from the *DMS*, thus stalling the instructions in *MEM1* and *EX* stages.

To break this critical path, the request was sent from the execute stage irrespective of stalls or exception. If the instruction in the execute stage won arbitration but stalled or raised an exception, it sent a "hosed" signal to the *DMS* indicating that the

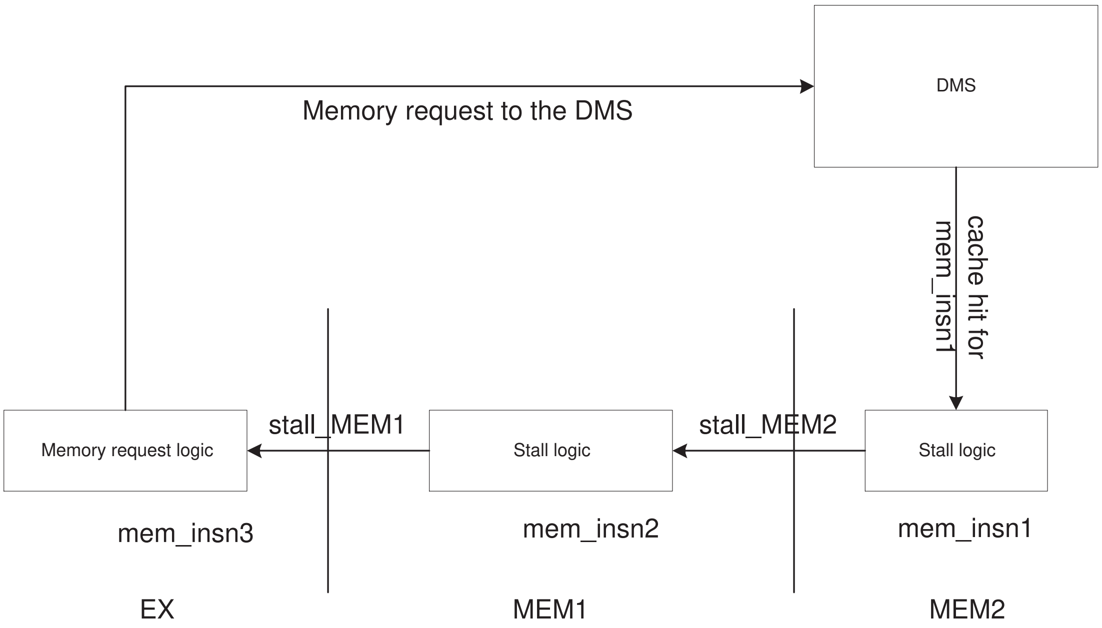


Figure 3.3: Old implementation of the memory requests

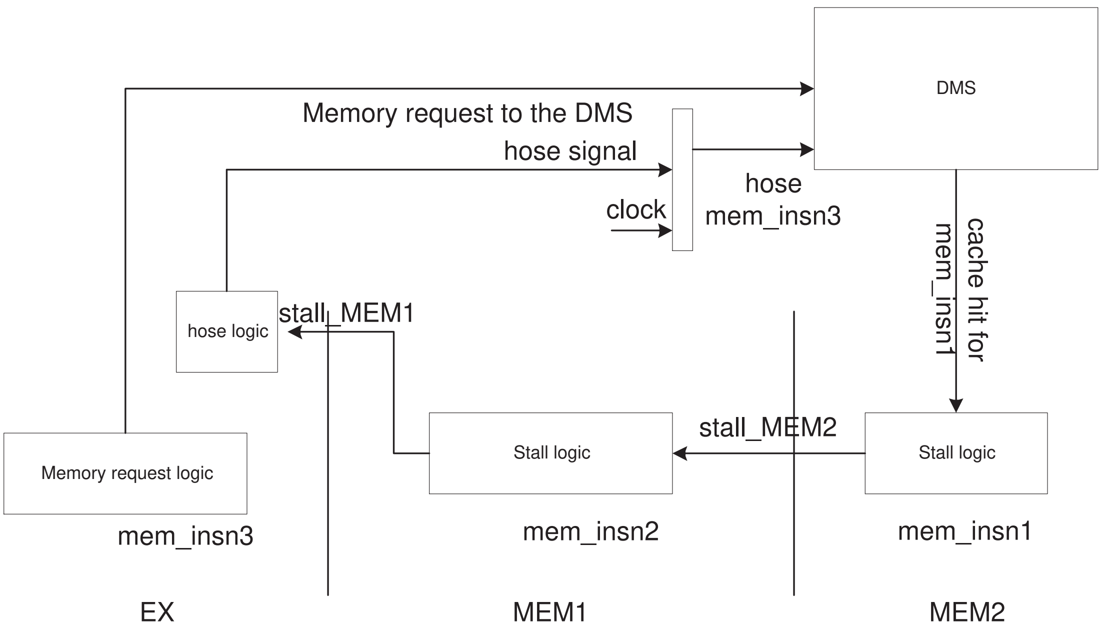


Figure 3.4: Improved implementation of the memory requests

instruction would not make it to the *MEM2* stage in two cycles.

### 3.3.2 TLB Instructions

*TLB* requests are sent to the *DMS* in the *MEM1* stage of the pipeline. The stall logic for these instructions was changed to stall them in the *MEM1* stage until the pipe ahead of them was empty. This had been done because, in case of a load followed by a *TLB* instruction, there would be a reply from the memory unit and a *TLB* request in the same cycle. This turned out to be a critical path and had to be broken.

Also, the architecture specifies that all the instructions commit in the *WB* stage, while *TLB* instructions were being committed in the *MEM1* stage. If the pipe ahead of the *TLB* instruction was made empty, it was equivalent to committing in the write back stage.

This change introduced some extra ports and additional conditions to the stall logic.

### 3.3.3 Address Selection for Memory Instructions

Address calculation for memory instructions is performed in the execute stage of the pipeline. Earlier the address bus going to the *BIU* got its value from the result selection unit, that primarily consisted of a set of multiplexors to select the final result based on the instruction type.

In case of a load store instruction, always the *ALU* output is selected and the implementation created false dependencies with the overflow and negative output of the *ALU*. This dependency was removed by assigning the value of the *ALU* result straight to the address bus.

### 3.3.4 Start, Head and Tail Signal

The Start signal comes to the *PE* from the sequencer. In the floorplan for the Kestrel processor, the sequencer is very far from the *PEs* and more than $\frac{2}{3}rd$ of the cycle is consumed by the interconnect delays. This signal comes to the *PE* once at the beginning of every task and is not critical. This was therefore latched at the input to solve timing problems.

Similarly head and tail signals also come from the sequencer. These signals were also latched at the level of the *DRF*, *SPR*, *BIU* and *PE* integration.

### 3.3.5 Instruction from the *IMU*

In the earlier implementation the path from *i-cache* miss to the next *i-cache* miss via the *IMU* was latched only at the input to the *IMU* as shown in Figure 3.5. This led to obvious timing violations as the interconnect delay between these units is $\frac{1}{3}rd$ of the clock cycle, thus leaving only $4ns$ for cache lookup and logic in the *IMU*. The instruction from the *IMU* was therefore latched at the input to the fetch stage (shown in Figure 3.6) of the processing element.

The introduction of latches in the *PE* led to changes in the logic of *IMU* as well and a need to verify its functionality.

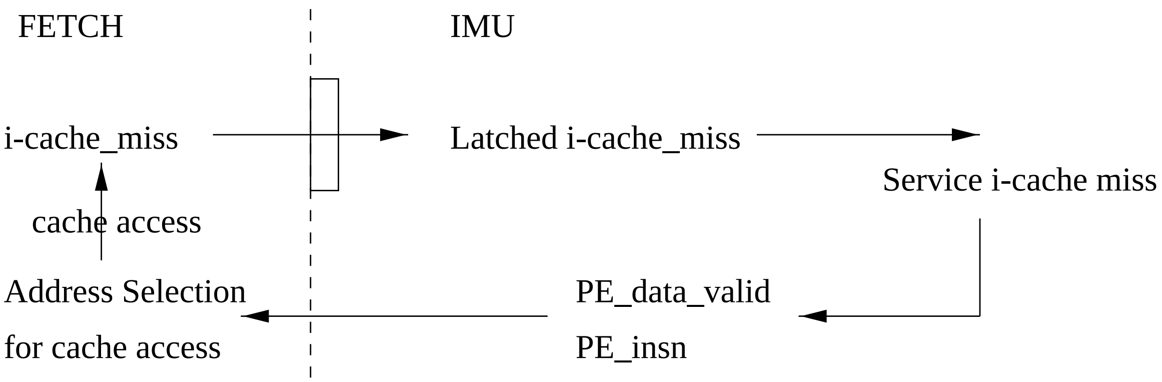


Figure 3.5: Old configuration of the instruction miss logic

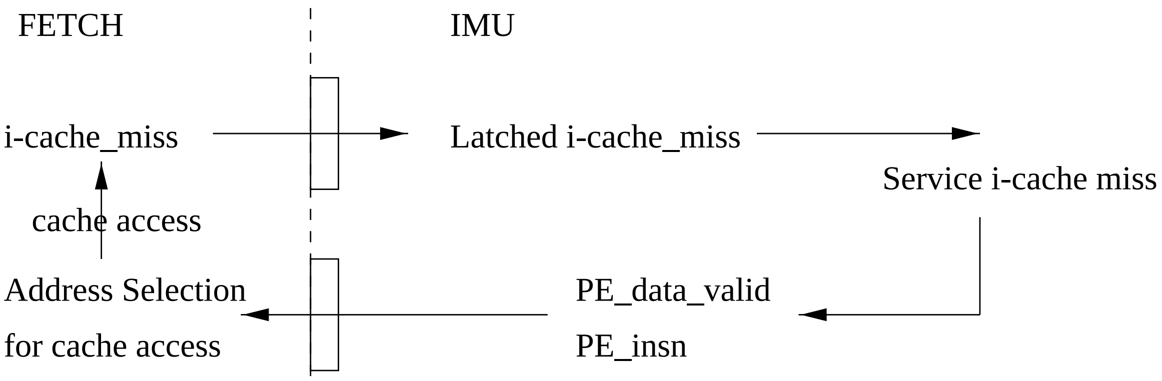


Figure 3.6: Improved configuration of the instruction miss logic

### 3.3.6 Load Header (*LDHD*) Instruction

The design specification indicated that the virtual address of a *LDHD* instruction should go through the translation process in the *IMU*, but this was not implemented in the Verilog model. Thus the address being sent to the *IMU* was virtual address and not the physical.

This problem was solved by introducing a *PLB* (Primary Lookaside Buffer) in the *ID* stage of the pipeline. It was chosen to implement the translation logic in the *PE* instead of the *IMU* because the *PLBMU* already had the capacity to handle multiple *PLB* miss requests from different *PEs*.

This change involved addition of new logic to the pipeline and additional ports to the *PLB* miss unit.

### 3.3.7 Forward Signal from the Execute Stage

The forward signal from the execute stage indicates to the *DRF* that the value produced needs to be forwarded. The forwarding of the data is done only if the execute stage is valid and not stalling. The stall logic depends on the arbitration logic for memory instructions. Therefore,the logic designed had a dependence on the arbitration won signal from the *BIU*.

Memory instructions in the execute stage never forward data as they perform address calculation. The Verilog was modified to remove this false path that was being reported as critical by *Design Analyzer*.

### 3.3.8 Memory Request from *MEM1* Stage

In the initial design, memory requests were sent only from the execute and *MEM2* stage of the pipeline. If a request was cancelled in the *MEM1* stage, it was resent only when it reached the *MEM2* stage. The design was changed to send requests from the *MEM1* stage as well. This change was done for improving the *IPC* and not synthesis results.

### 3.3.9 Special Adder for 5th and 6th Bits of Address

The 5th and 6th bits of the address specify the memory bank in case of a load store instruction. The calculation of these bits was on the critical path of arbitration for the bus. The adder used in the design was a carry look ahead adder with the carry rippling after every 4-bits.

A special adder with no ripples to calculate the 5th and 6th bits was therefore designed to reduce the combinational logic time.

### 3.3.10 Stall Logic

A memory instruction stalls in the *EX* stage until it wins bus arbitration. It takes $\frac{2}{3}s$ of the cycle to determine whether the instruction is going to stall. The fetch stage would also stall in case of a stall in the *EX* stage, if there is no invalid instruction between the two. The determination of the next program counter for instruction fetch thus depends on the stall signal. The stall logic was therefore on the critical path.

The stall from the *EX* was changed not to include that stall because of arbitration. Additional ports were added to *IF* and *ID* stage of the pipeline to indicate there was a memory instruction in the *EX* stage and the arbitration signal was sent straight to these stages. This added additional logic but removed the interconnect delay. The schematic diagram of both the configurations is shown in Figure 3.7 & Figure 3.8.

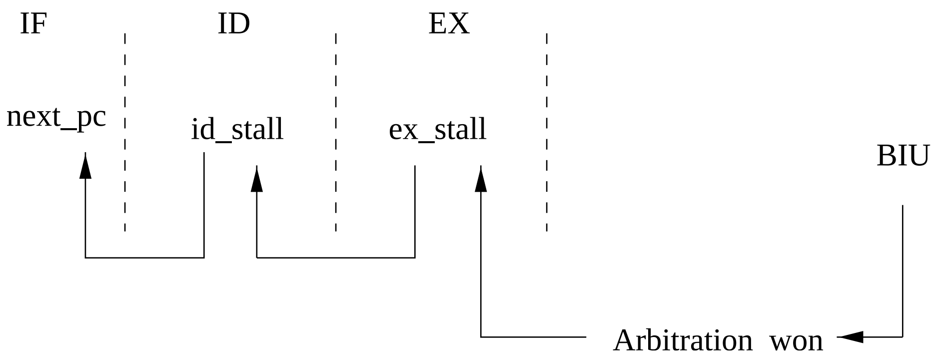


Figure 3.7: Old configuration of the stall logic

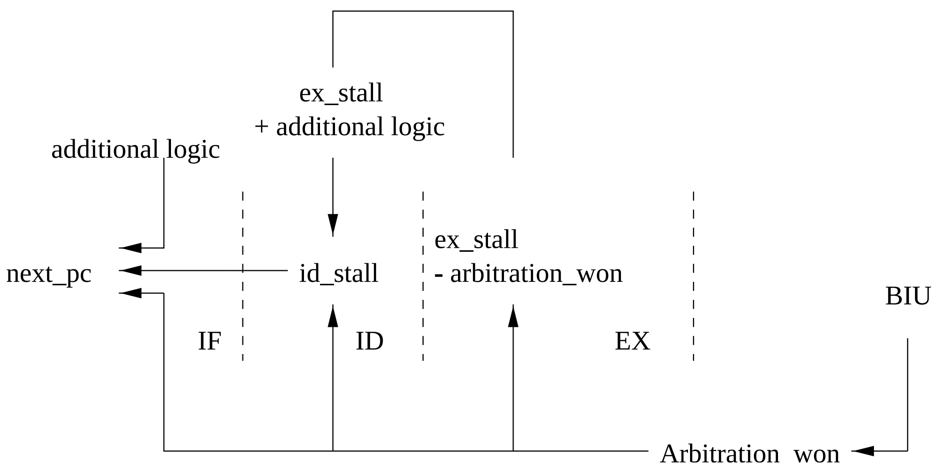


Figure 3.8: Improved configuration of the stall logic

### 3.3.11 *TLB* Exception

The implementation of Kestrel had some false paths originating from the *TLB* exceptions. The Verilog was modified to eliminate these paths by removing false dependences. A brief description of the main changes made is outlined below.

**BTB** is not modified by a memory instruction. In the earlier implementation, the modification of the *BTB* was dependent of the fact that there should be no exception in the pipeline. As memory instructions do not modify the *BTB*, the dependence on *TLB* exception (only caused by memory instructions) could be removed.

**PE** need not send a "hosed" signal[1] to *DMS* in case of a *TLB* exception. This is not needed as the *TLB* exception is generated by the *DMS* itself, and the unit takes care of it.

**TLB request** sent by the *PE* need not depend on *TLB* exception, this is taken care of by the *DMS*. The *TLB* exception refers to the exception caused by the instruction in the *MEM1* stage. As, the pipeline ahead of the *TLB* instruction is always empty when the request is sent there is no instruction ahead of it that can cause an exception.

## 3.4 Latches at the Input and Output of *DMU*

The *DMU* is the miss unit that services the data cache misses. In the static timing analysis of the top-level integration of the processor, there were several paths through the *DMU* violating the timing constraints.

This unit is a non-critical unit and therefore latches were added at the input and output of every signal of the *DMU*. The critical paths observed after this change are discussed in the results section. This change effects the functionality of the unit,

[1]Indicates that the instruction will not be in the *MEM2* stage in the next cycle.

but the *DMU* can easily be redesigned to function properly with additional stages of pipeline.

## 3.5 Data Memory System

The *DMS* is the most critical unit in the Kestrel processor. Several modifications to the design have been made in this unit based on the static timing analysis results[2]. A few of these changes are outline below.

### 3.5.1 Reorganization of Verilog

The synthesis tool does not do a good job of synthesis, in case of large blocks of random logic. This behavior was observed in the data memory system. The solution to this problem was to divide the logic into smaller units and specify strict constraints on these submodules. Better synthesis results (w.r.t. timing) were obtained when such restructuring of the logic was done.

### 3.5.2 Table Macros

It was noticed that synthesis tools are not efficient at optimizing the logic represented as table macros. For example, in the *DMS* the signal "write_store_5" is part of a table macro. The table consists of many unrelated signals, as a result, many of the inputs to the "write_store_5" logic are don't-cares. In one case "direct_grant", a very critical signal, ends up in the prime implicants for "write_store_5", which in reality has nothing to do with this signal.

[2] These changes made by Eric Rotenberg and Craig Zilles have been included here for completeness.

The solution to this problem was to manually implement the logic for "write_store_5".

### 3.5.3 *TLB* Probe Instruction

The virtual address selection *mux* in the *DMS* selects between the virtual address for a *TLBP* instruction and other memory instructions. The control signal for this multiplexor was originally the "TLBP_in" signal that created a dependence between the *TLB* exception for a memory request and "TLBP_in". This path was broken by changing the control signal of the multiplexor.

# Chapter 4

# Synthesis Results

The static timing analysis of the processor was performed using the *Primetime* and *Design Analyzer* timing tools. This chapter discusses in detail the timing results and highlights the critical paths observed in various units and at the top-level of the design. The synthesis of the processor was not optimized for area, but the area estimates available are also be presented here.

The Design attributes specified for the synthesis of the Kestrel processor are:

| Design Attribute | Value |
|---|---|
| temperature_max | $25^oC$ |
| voltage_max | 3.3V |
| wire_load_model_max | B0.5X0.5 |
| wire_load_model_library_max | lcb500kv |

## 4.1 Area Estimates

The area estimates for the units synthesized are listed in the Table 4.1. The area is represented in terms of $mm^2$. The synthesis tools give the area in terms of *LSI* [7] cell equivalents, and the conversion from LSI cells to die area is between $20,000$ and $25,000$ cells per $mm^2$ [2]. The total area of the design calculated using the data in Table 4.1 was about $16.36cm^2$. The area calculated includes the overhead for global routing(15%) and excludes the overlapping modules. The area for the *L2* cache was

| Design module | Approximate area in $mm^2$ |
|---|---|
| *DMS* | 206.76 |
| *DMU* | 3.66 |
| *IMU* | 26.72 |
| *PLBMU* | 7.14 |
| *BIU* | 2.13 |
| *SPR* | 1.55 |
| *PE* | 83.69 |
| *GLUE LOGIC* | 0.28 |
| *SEQ* | 219.66 |

Table 4.1: Area estimates for Kestrel [2]

not estimated as it was never synthesized. The synthesis process assumes the *L2* cache as a black box.

## 4.2 Timing Analysis

Timing analysis is primarily used to ensure that the specific timing requirements in a design are satisfied. Static timing analysis tools check the timing of all possible paths in a design against the design requirements. Delays of each gate and interconnect are calculated, and critical paths traced using the minimum and maximum arrival times to the points of interest (Usually primary input and output ports).

The *Primetime* and *Design Analyzer* timing tools (refer to Appendix C) were used for the purpose of static timing analysis of the Kestrel processor. This section presents the results obtained, in terms of slacks (difference between the required path delays and the actual path delays) observed and the critical paths identified. These details are given for the each of the submodules and the top-level integrated processor synthesis. An example of the timing report is given in Appendix C.

It is important to note that the synthesis and timing analysis were done for a clock cycle time of $12ns$.

### 4.2.1 *IMU*

The instruction miss unit violates the clock cycle timing requirements by $1.08ns$. The distribution of the slacks for the endpoints are shown in Figure 4.1. The x-axis represents the slacks in nanoseconds and the y-axis represents the number of endpoints.

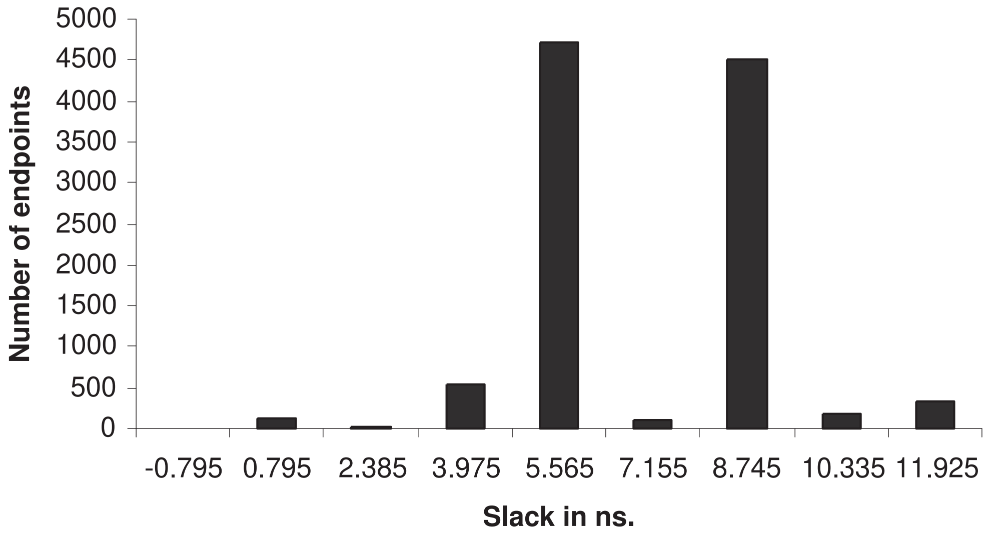


Figure 4.1: Endpoint slacks for the IMU

The critical path for the *IMU* is the logic involved in determining "PE_data_valid_out". This signal tells the *PE* that the miss request has been serviced and the instruction is available.

It can be inferred from Figure 4.1 that very few endpoints violate the cycle time requirement. The number shown in the Figure appears to be significant because every bit of a bus is interpreted as a different endpoint though they represent the same path (in most cases). The timing results thus report the same path (with different bits of a bus) as critical.

### 4.2.2 *BIU*

This unit primarily consists of arbitration logic that cannot be flattened. The module violates the clock cycle requirement by $1.73ns$. The violation is roughly 26 % over the specified clock cycle time. The endpoint slack distribution for the BIU is shown in Figure 4.2.

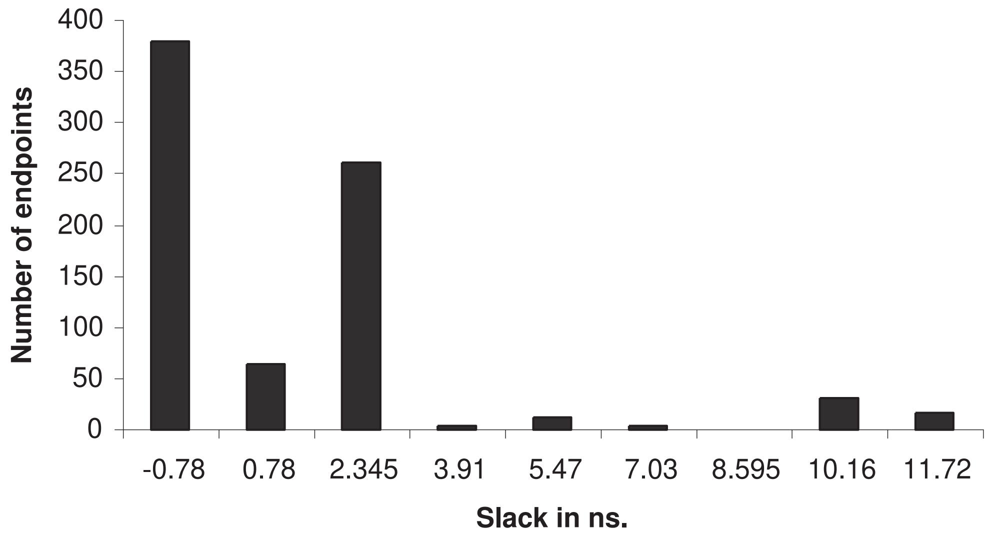


Figure 4.2: Endpoint slacks of the bus interface unit

The critical path is to the endpoint "DMS3_cntl_out" (control bus to the *DMS*). This path becomes the critical path because the request coming from the *PE* takes about $6ns$. The arbitration for the bus is done after the request is received by the

*BIU*.

Another critical path observed in this unit was from the address for the memory request to the "arbitration_won" (going to the *PE* indicating that it won arbitration for the bus). The adder in the execute stage of the pipeline was optimized to reduce the time taken for the calculation of the address bits used in arbitration. In-spite of the change made, this path shows up as one of the critical paths in the timing analysis.

### 4.2.3 *SPR*

The special purpose register is a small unit that keeps the state of the processor. The synthesis results showed that this unit meets the timing specifications. The slack distribution for the *SPR* is shown in Figure 4.3.

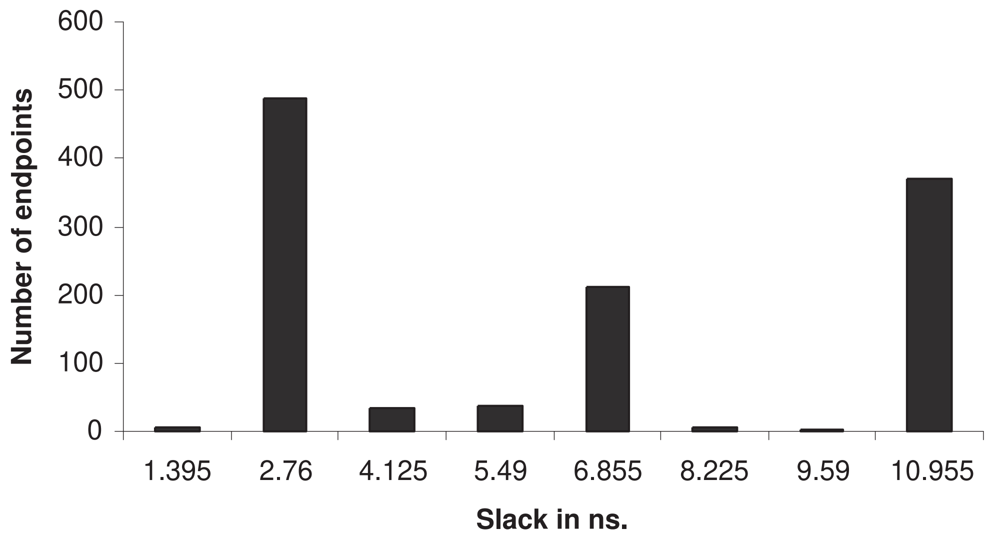


Figure 4.3: Endpoint slacks of the special purpose register unit

The most critical path meets the timing with a positive slack of $0.71ns$.

### 4.2.4 *DMS*

The data memory system violates the timing constraints by $4.16ns$. Efforts are underway to improve the timing on this unit. The slack distribution for the memory system is shown in Figure 4.4.

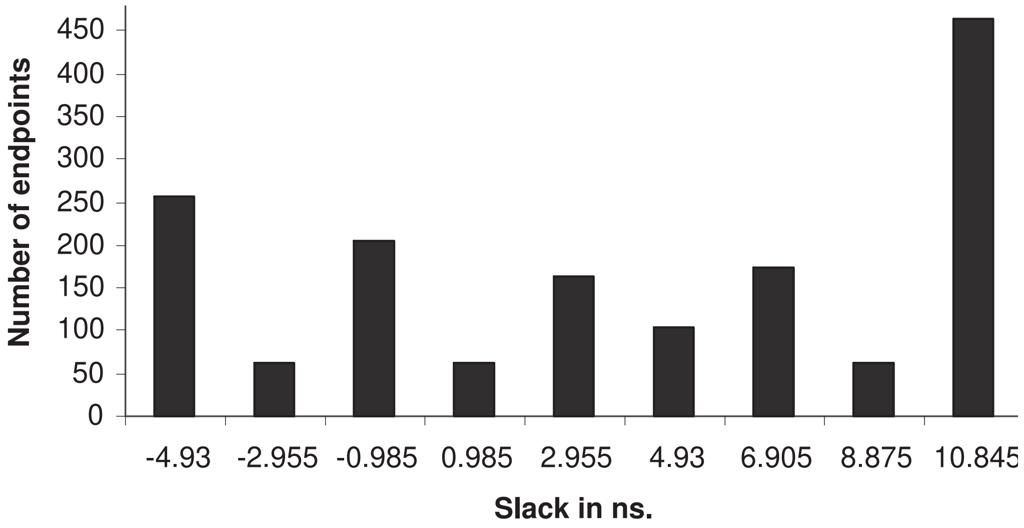


Figure 4.4: Endpoint slacks of the data memory system

### 4.2.5 *SEQ*

The sequencer is also violates the clock cycle requirement by less than $3ns$. The slack for this unit is negative $2.63ns$. The distribution of the slacks is shown in Figure 4.5.

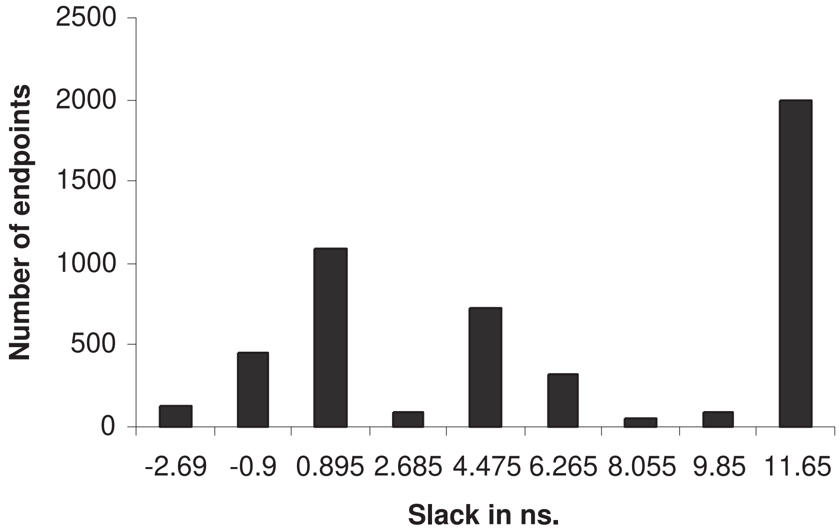


Figure 4.5: Endpoint slacks of the sequencer

### 4.2.6 *DRF*

The synthesis results indicate that the *DRF* meets the clock cycle requirements. The slack distribution for this unit is shown in Figure 4.6.

The most critical path meets the timing with a positive slack of $0.50ns$.

### 4.2.7 *PE*

The processing element consists of 6 stages of pipeline. Each of the stages has been optimized individually and integrated at the top-level. The top-level *PE_INTEG* has further been compiled to optimize the netlist.

The timing analysis shows that the processing cores violate the timing specifications by $2.7ns$. The slack distribution for the unit is shown in Figure 4.7.

The adder in the execute stage comes in the critical path of the unit. The results

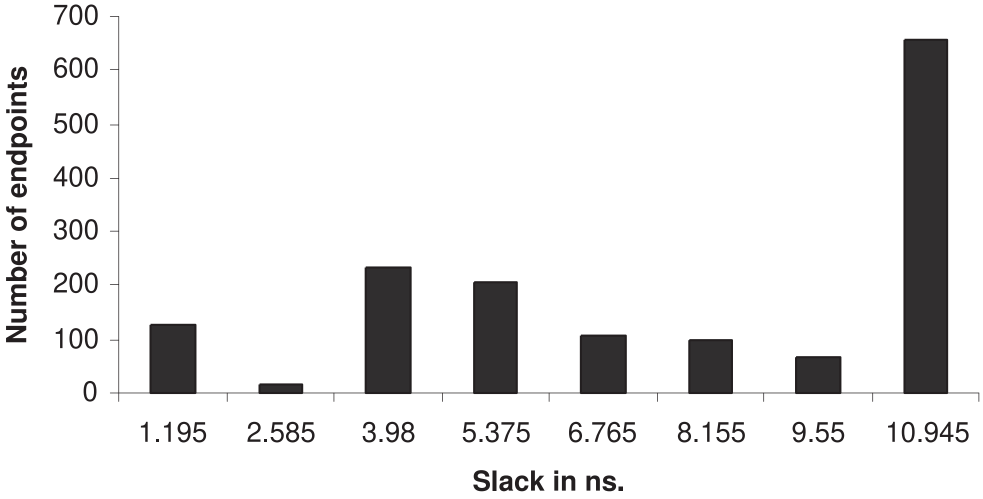


Figure 4.6: Endpoint slacks of the distributed register file

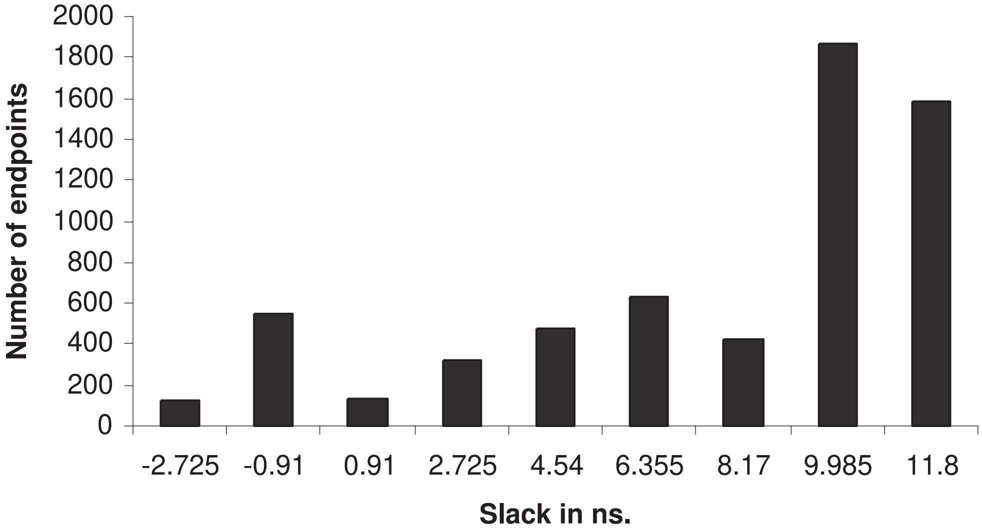


Figure 4.7: Endpoint slacks of the processing element

of the adder could cause exceptions (like overflow exception), which in turn effect the stall logic. The stall logic in case of memory instructions was decoupled, but the stall in other cases was implemented as a daisy chain. This created a dependence between the "next_pc" (next program counter determined in the fetch stage of the pipeline) and the adder output.

The load store path was also identified as critical by the timing analysis tool. This path became critical because of the amount of logic as well as the interconnect delays associated with it. The path traced by a load/store request is shown in Figure 4.8.

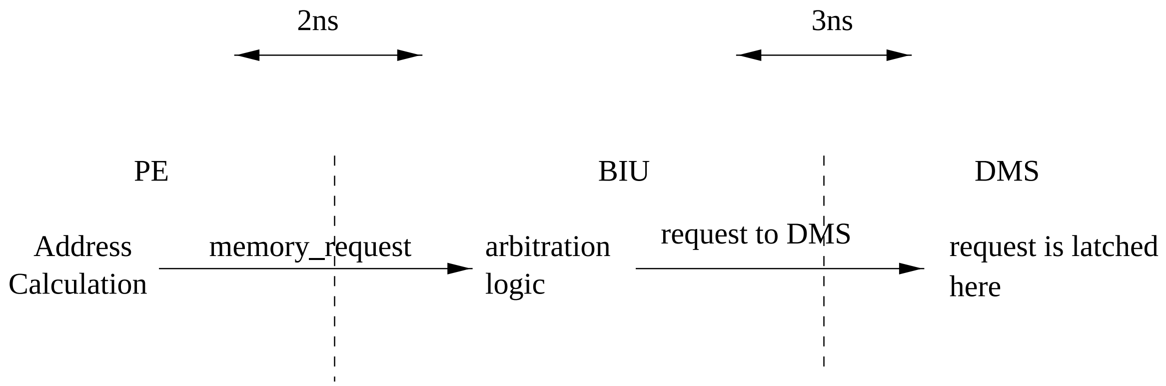


Figure 4.8: Path traced by the memory requests

### 4.2.8 *DRF*, *SPR*, *BIU*, and *PE* Integration

The four processing cores, corresponding register files, *BIU*, and *SPR* were combined at the top-level and timing analysis was done at this level of integration. This unit was violating the cycle time requirement of $12ns$ by $2.7ns$. The slack distribution for the entire unit is plotted in Figure 4.9

The critical path at this level of the design was the forward logic that goes from the *PE* to the *DRF*. The path starts from the adder that takes majority of the cycle for

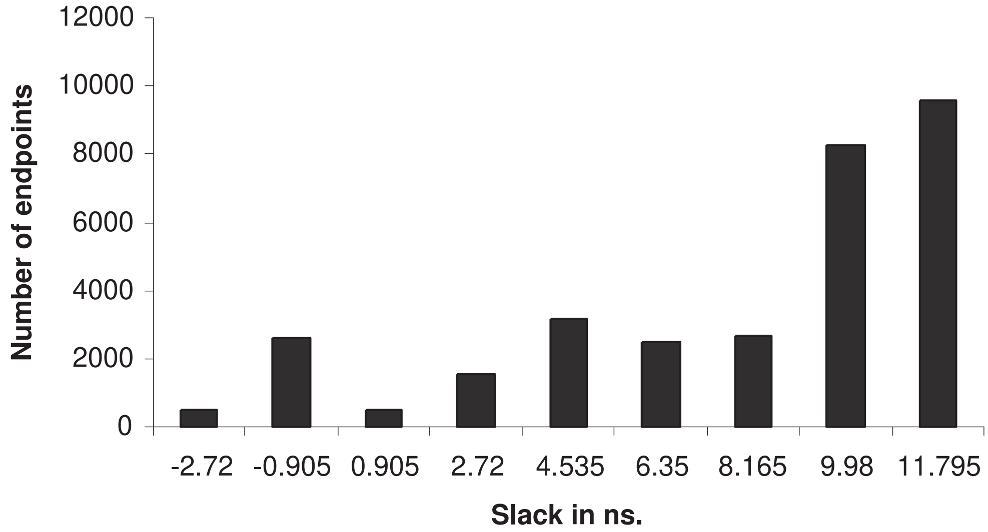


Figure 4.9: Endpoint slacks of the *DRF_SPR_BIU_PE_INTEG*

calculating the result. The stall in the execute stage determines whether to forward or not, which in turn depends on the exception output of the adder.

Further optimizations are being done on this unit and other underlying submodules to achieve the target cycle time of $12ns$.

### 4.2.9 Top-Level Integration

The top level-integration of all the units was used for the timing analysis of the processor. The slack distribution at this level is shown in Figure 4.10.

The current cycle time for the integrated processor is about $18.7ns$,that is a negative slack of $6.7ns$. Efforts are underway to reduce this slack by identifying the critical paths and making design optimizations to reduce the cycle time.

The critical path being observed initiates in the *PE* and goes through the *DMS*.

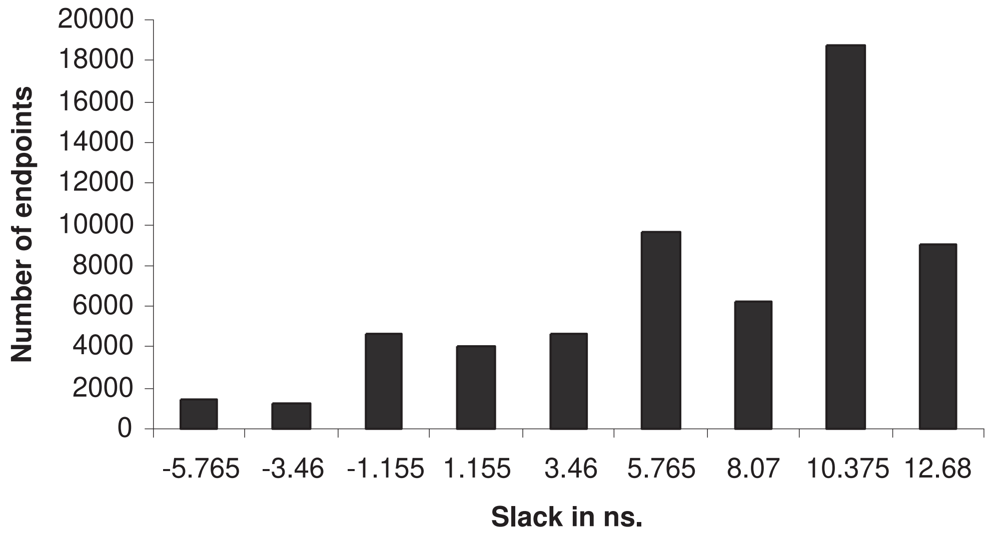


Figure 4.10: Endpoint slacks of *INTEG_FINAL*

The multiplexor that selects the physical address for tag match in *DMS*, has its control signal dependent on the more critical "TLB_op", thus making this path critical. The cycle time could further be improved by removing this dependency.

The results presented above can thus be improved by iteratively making changes to the design and optimizing the more critical paths for timing. The following chapter suggests measures that should have been taken in the process of design to meet the timing requirements, and highlights how the problems faced could be eliminated.

# Chapter 5

# Conclusions

## 5.1 Overview of the Work Done

This thesis includes a detailed description of the process used for the synthesis of the Kestrel processor. The *Synopsys* and *Cadence* tools were used for the purpose of synthesis and analysis. Several techniques and compile time optimizations have been discussed and applied to the Kestrel processor. The processor has been synthesized for a clock cycle time of $12ns$. The following sections outline the current status, the problems faced in trying to achieve the specified goal, and suggestions for further improvement.

The Kestrel processor uses $0.5\mu$ ASIC technology. The area estimates for the processor were obtained from the floorplanning and synthesis tools. Timing analysis done indicate that the implementation is feasible and can be optimized with considerable effort to meet the constraints.

## 5.2 Current Status

The target cycle time for the Kestrel multiscalar processor is $12ns$. Current limitation is the DMS unit, that has a negative slack of $4.2ns$. All the other units meet the constraints with a negative slack of a few nanoseconds. The top level integration has a negative slack of about $6.7ns$. This slack refers to the intermodule paths, that do not show up when individual modules are timed. Efforts are underway to identify these paths and reduce the slack on these paths.

The floating point unit in the processor has not been designed and synthesized. The addition of this unit might result in an increase of the cycle time by a few nanoseconds.

The final area estimate for the processor has been made as $16.36cm^2$. This area does not include the floating point execution units and the L2 cache. It is important to note that the *ASIC* standard cell designs (used for Kestrel) are typically slower and larger than the same circuits designed with full custom circuits. Current area estimates scaled to projected technologies indicate that the Kestrel die would use only 5% of the area in a $0.07\mu$ technology [2].

## 5.3 Pitfalls & Future Directives

This section points out the problems faced during synthesis and possible solutions to those problems. Some future directives to improve the synthesis results are also presented here.

In the ideal project flow, the output of synthesis should direct the design, whereas in Kestrel the design was done without any input from the synthesis process. The design was later modified to suit the timing requirements. Many of the problems could be solved by a better structuring of the Verilog code. Synthesis tool is incapable of optimally synthesizing large blocks of random logic. Logical regrouping of such logic by the designers into smaller sub-blocks at the outset would have saved the time spent in redesigning and re-organizing the logic in later stages of the project.

The major problem in synthesis was caused by the glue logic at the top level of the design hierarchy. The glue logic had to be regrouped into one submodule. This regrouping was done for synthesis purposes. The synthesis would have been much simpler and easier to constrain if there was no such floating logic between the submodules.

Adherence to good naming conventions would have helped save a lot of initial synthesis and debugging time. If the instances of modules were given more meaningful names, it would have been easier to identify the paths indicated by the timing reports.

Module interfaces in the synthesis scripts were not updated every-time there was a change in the design. This incompatibility between the synthesis scripts and the Verilog description caused a lot of wastage of time. Tools to automatically reflect such changes should be implemented in early stages of the project.

The constraints specified for the inputs and outputs of different modules, were estimated delays obtained from the floorplan. A more accurate timing analysis could be performed if the *SDF* (standard delay format) and capacitance information could be back annotated on the design. The *SDF* file available is of a different version and attempts are being made to convert to the acceptable format.

The synthesis of the Kestrel processor assumes a perfect clock with no skew. It would be nice to introduce skew for more realistic results.

The current implementation of the Kestrel synthesis uses custom wire load model. Various other wire load models must be tried and an optimal model for the design should be determined.

# Appendix A

# Synthesis Terminology

This appendix summarizes the terminology used in synthesis [8] and its basic concepts.

**Synthesize**

The process that generates a gate-level netlist from the *HDL* source code. Synthesis includes both reading the *HDL* code and optimizing the code.

**Optimize**

The step in the synthesis process that attempts to implement a combination of library cells to meet the functional, speed, and area requirements of a design.

**Compile**

The command that executes the optimization step. After reading the design and specifying constraints, the design is compiled.

**Forward Annotation**

Passing timing constraints to the place and route (layout) tool.

**Backward Annotation**

Process of passing physical design information from the physical design environment to the synthesis environment.

**Technology Libraries**

The technology libraries contain a vendor-specific set of leaf cells. The cells are defined by their functions, timing, wiring and wiring size. A target library is a technology library that *Design Compiler* maps to during optimization.

**Symbol Libraries**

The symbol library contains definitions of the graphic symbols that represent cells in the design schematics.

**Design**

Designs are circuit descriptions that perform logical functions. Logic-level designs are represented as sets of boolean equations, gate-level designs, such as net-lists are represented as interconnect cells. Designs can be hierarchical or flat.

**Hierarchical Design**

Hierarchical designs contain one or more designs. Designs within a hierarchical design are called cells or subdesigns. Each subdesign can further contain a hierarchy of designs in it.

**Flat Design**

Flat design does not consist of any subdesigns. It is completely represented using leaf cells.

### Design Rules

Design rules are technology requirements that cannot be violated. They are rules set up by the *ASIC* vendors to ensure that the product meets vendor specifications and works as intended.

### Design Constraints

Design constraints are design goals represented as measurable circuit characteristics, such as timing and area.

### Cells

A cell is an instantiation of a design or subdesign within another design. Each time a cell is used in a design, it becomes a design instance. A design can contain multiple instances of the same cell.

### References

A reference is a pointer to a cell or subdesign in its parent design. Cells used multiple times in a design point to the same reference.

### Nets

Nets (networks) are the wires that connect ports to pins and pins to each other.

### Ports

Ports are the inputs and outputs of a design.

**Pins**

Pins are the input and output of the cells within a design (such as gates and flip-flops). The ports of a subdesign are pins within its parent design.

**Routing**

Routing is the network of connections between logic gates.

**Layout**

A complete layout is a geometric representation of a chip, overlaid with the geometry of a physical design.

**Place and Route**

Placing netlist gates at physical coordinates on the chip layout (with no overlaps) and then physically connecting (route) the gates.

**Chip**

Physical implementation of a design, usually built for silicon. Also known as a die.

# Appendix B

# Scripts

This appendix shows a listing sample of the scripts used for synthesis and reporting purposes. Synthesis scripts are from *Design Analyzer*, whereas timing and area reports have been obtained using the *Primetime* tool. A sample timing report obtained using *Primetime* is also shown in the end.

## B.1 Synopsys Scripts

```
/*************************************************************/
/* PE_INTEG synthesis script  */
/* command line is dc_shell -f PE_INTEG.script */
/* create the WORK directory */

sh mkdir WORK
sh mkdir db

define_design_lib WORK -path "./WORK"

/* read in kestrel libraries in uncompiled form */
include libread.script
include WE_disable.script

read -format db {"./db/mem1.db"}
```

```
read -format db {"./db/mem2.db"}
read -format db {"./db/fetch.db"}
read -format db {"./db/execute.db"}
read -format db {"./db/instdecode.db"}
read -format db {"./db/wb.db"}

analyze -format verilog -lib WORK {"PE/INTEGRATION/pe_integ.v"}
analyze -format verilog -lib WORK {"PE/MISCEL/data_ctl.v"}
elaborate pe_integ -arch "verilog" -lib WORK -update
elaborate data_ctl -arch "verilog" -lib WORK -update

set_dont_touch "mem1.db:mem1"
set_dont_touch "mem2.db:mem2"
set_dont_touch "fetch.db:fetch"
set_dont_touch "execute.db:execute"
set_dont_touch "instdecode.db:instdecode"
set_dont_touch "wb.db:wb"

create_schematic -size infinite    -gen_database

current_design pe_integ

/* ---------------------- SET CLOCK ------------------------- */
create_clock -period 12 -waveform {0 6} find(port,"clk")
uniquify

/* ---------------------- SET ATTRIBUTES -------------------- */
set_input_delay 9 -max -clock "clk" find(port,"arbitrate_in")
```

```
set_input_delay 3 -max -clock "clk" find(port,"resend_w_h_in")
set_input_delay 3 -max -clock "clk" find(port,"resend_in")
set_input_delay 3 -max -clock "clk" find(port,"priority_resend_in")
set_output_delay 6 -max -clock "clk" find(port,"data_addr_out")
set_output_delay 8 -max -clock "clk" find(port,"data_addr_out[5]")
set_output_delay 8 -max -clock "clk" find(port,"data_addr_out[6]")
set_output_delay 6 -max -clock "clk" find(port,"data_cntl_out")
set_output_delay 10 -max -clock "clk" find(port,"data_cntl_out[5]")
set_output_delay 10 -max -clock "clk" find(port,"data_cntl_out[6]")

/* ---------------------- NET LOADS ------------------------- */
/* ---- INPUTS ---- */
set_driving_cell -cell BUFA -library lcb500kv -pin "Z" -no_design_rule
find(port, "arbitrate_in");
set_driving_cell -cell BUFA -library lcb500kv -pin "Z" -no_design_rule
find(port, "priority_resend_in");
set_driving_cell -cell BUFA -library lcb500kv -pin "Z" -no_design_rule
find(port, "flush_cache_in");
set_driving_cell -cell BUFA -library lcb500kv -pin "Z" -no_design_rule
find(port, "resend_in");
set_driving_cell -cell BUFA -library lcb500kv -pin "Z" -no_design_rule
find(port, "resend_w_h_in");

set_max_fanout 1 find(port, "head_1_in");
set_max_fanout 1 find(port, "start_1_in");
set_max_fanout 1 find(port, "restart_PE_1_in");
/* ---- OUTPUTS ---- */
set_load 0.43 find(port,"data_addr_out")
```

```
set_load 0.43 find(port,"data_cntl_out")

current_design pe_integ

/* ---------------------- SYNTHESIZE ------------------------ */
set_flatten false
set_flatten -effort medium
set_flatten -minimize single_output
set_flatten -phase false
set_structure true -timing true
set_ungroup current_design true
set_local_link_library {lcb500kv.db}
set_wire_load "B0.5X0.5" -library "lcb500kv" -mode "enclosed"
compile -map_effort medium

/* ---------------------- SAVE ------------------------------ */
ungroup -flatten -all
write -format db -hierarchy -output "./db/pe_integ.db" \
{"pe_integ.db:pe_integ"}

/* ----------------------Reporting--------------------------  */
check_design > ./pe_integ_report
report_area > ./reports/pe_integ.area
report_timing -path full -delay max -nworst 100 -max_paths 100 > \
./reports/pe_integ.timing

exit
```

```
/**************************************************************/
/* INTEG_FINAL synthesis script  */
/* command line is dc_shell -f INTEG_FINAL.script */

/* create the WORK directory */
sh mkdir WORK
sh mkdir db

define_design_lib WORK -path "./WORK"

/* read in kestrel libraries in uncompiled form */
include libread.script
include WE_disable.script

read -format db {"./db/drf_spr_biu_pe_integ.db"}
read -format db {"./db/PLBMU_new.db"}
read -format db {"./db/sequencer.db"}
read -format db {"./db/DMSUL2.db"}
read -format db {"./db/imu_integ1.db"}
read -format db {"./db/integ_final_glue.db"}
read -format db {"./db/bidir_1to1_64_bus.db"}
read -format db {"./db/bidir_5to1_8_bus.db"}
read -format db {"./db/bidir_4to1_54_bus.db"}
read -format db {"./db/bidir_4to1_128_bus.db"}
analyze -format verilog -lib WORK {"TOP/integ_final.v"}
elaborate integ_final -arch "verilog" -lib WORK -update

set_dont_touch "bidir_1to1_64_bus.db:bidir_1to1_64_bus"
```

```
set_dont_touch "bidir_5to1_8_bus.db:bidir_5to1_8_bus"
set_dont_touch "bidir_4to1_54_bus.db:bidir_4to1_54_bus"
set_dont_touch "bidir_4to1_128_bus.db:bidir_4to1_128_bus"
set_dont_touch "drf_spr_biu_pe_integ.db:drf_spr_biu_pe_integ"
set_dont_touch "PLBMU_new.db:PLBMU_new"
set_dont_touch "sequencer.db:sequencer"
set_dont_touch "DMSUL2.db:DMSUL2"
set_dont_touch "imu_integ1.db:imu_integ1"
set_dont_touch "integ_final_glue.db:integ_final_glue"

create_schematic -size infinite    -gen_database

/* Set the current_design */
current_design integ_final
link

/* ---------------------- SET CLOCK --------------------- */
create_clock -name "gclk" -period 12 -waveform {0 6} find(port,"gclk")

/* ---------------------- SAVE -------------------------- */
write -format db -hierarchy -output "./db/integ_final.db" \
{"integ_final.db:integ_final"}
exit
```

## B.2 Primetime Scripts

```
/*----------------------- PE_INTEG -------------------------------*/
# pt_shell   example script for timing analysis.
```

```
#  read this script in using "source" scriptname
#  or else start up using "pt_shell -f thisfile"

# page the output
set sh_enable_page_mode true

# now relink the design
set link_path"* mergener/synopsys/LCB500K_LIB/lcb500kv.db"

# read in the db for the design
read_db /p/multiscalar/padmaja/synopsys/db/pe_integ.db

link_design pe_integ

create_clock -period 12 [get_ports clk]

source /p/multiscalar/mergener/synopsys/scripts/WE_disable.script
report_design
report_reference

# this check to see if all ports are constrained, It will fail if
# any port in the design is not constrained. That is OK and expected.
check_timing

# this will start to do the timing analysis.
report_timing

check_timing
```

```
report_timing -delay max -path_type full -max_path 2

quit

/*--------------------- DRF SPR BIU PE INTEG ----------------------*/
# pt_shell   example script for timing analysis.
#  read this script in using "source" scriptname
#  or else start up using "pt_shell -f thisfile"

# page the output
set sh_enable_page_mode true

# now relink the design
set link_path"* mergener/synopsys/LCB500K_LIB/lcb500kv.db"

# read in the db for the design
read_db /p/multiscalar/padmaja/synopsys/db/bidir_1to1_64_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/bidir_4to1_128_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/bidir_4to1_54_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/bidir_5to1_8_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/drf_spr_biu_pe_integ.db

link_design bidir_1to1_64_bus
link_design bidir_4to1_128_bus
link_design bidir_5to1_8_bus
link_design bidir_4to1_54_bus
link_design drf_spr_biu_pe_integ
```

```
create_clock -period 12 [get_ports clk]

report_design
report_reference > REPORTS/ref_drf_spr_biu

# this check to see if all ports are constrained, It will fail if
# any port in the design is not constrained. That is OK and expected.
check timing

# this will start to do the timing analysis.
report_timing -delay max -path_type full -max_path 2
quit

/*------------------------- INTEG_FINAL --------------------------*/
# pt_shell   example script for timing analysis.
#  read this script in using "source" scriptname
#  or else start up using "pt_shell -f thisfile"

# page the output
set sh_enable_page_mode true

# now relink the design
set link_path"* mergener/synopsys/LCB500K_LIB/lcb500kv.db"

# read in the db for the design
read_db /p/multiscalar/padmaja/synopsys/db/bidir_1to1_64_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/bidir_4to1_128_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/bidir_4to1_54_bus.db
```

```
read_db /p/multiscalar/padmaja/synopsys/db/bidir_5to1_8_bus.db
read_db /p/multiscalar/padmaja/synopsys/db/integ_final.db

link_design bidir_1to1_64_bus
link_design bidir_4to1_128_bus
link_design bidir_5to1_8_bus
link_design bidir_4to1_54_bus
link_design integ_final

create_clock -period 12 [get_ports gclk]

report_design
report_reference > REPORTS/ref_integ_final

check_timing
report_timing -delay max -path_type full -max_path 2

quit
```

## B.3 Timing Report

Given below is a typical timing report generated using the *Primetime* tool. The *Startpoint* and *Endpoint* represent the starting and ending of the path. The timing constraints specified on the Kestrel design are all w.r.t. the clock edge.

An output external delay of $10ns$ has been specified on the output port data_cntl_out therefore, the data required time is $2ns$, whereas, the logic takes $3.85ns$, violating the constraints by $1.85ns$.

```
Startpoint: mem2_r5/valid5_r12/r1/data_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: data_cntl_out[5]
          (output port clocked by clk)
Path Group: clk
Path Type: max

Point                                              Incr       Path
--------------------------------------------------------------------
clock clk (rise edge)                              0.00       0.00
clock network delay (ideal)                        0.00       0.00
mem2_r5/valid5_r12/r1/data_reg[0]/CP (FD1SQA)      0.00       0.00 r
mem2_r5/valid5_r12/r1/data_reg[0]/Q (FD1SQA)       0.46       0.46 r
mem2_r5/valid5_r12/U4/Z (BUFB)                     0.60       1.06 r
mem2_r5/U47/Z (N1L)                                0.49       1.55 f
mem2_r5/U77/Z (NR2A)                               0.22       1.77 r
mem2_r5/U88/Z (ND3A)                               0.23       2.00 f
mem2_r5/U97/Z (NR2A)                               0.20       2.20 r
mem2_r5/U103/Z (ND3A)                              0.19       2.39 f
mem2_r5/U106/Z (AO3A)                              0.26       2.65 r
mem2_r5/cb_cntl_bus_5_out[6] (mem2)                0.00       2.65 r
data_ctl_r7/U20/Z (N1B)                            0.16       2.81 f
data_ctl_r7/U17/Z (N1B)                            0.09       2.90 r
data_ctl_r7/U19/Z (N1C)                            0.07       2.97 f
data_ctl_r7/U14/Z (ND3B)                           0.10       3.08 r
data_ctl_r7/U11/Z (N1B)                            0.11       3.19 f
```

```
data_ctl_r7/r4/U14/Z (N1D)                        0.09       3.28 r
data_ctl_r7/r4/U13/Z (N1B)                        0.07       3.35 f
data_ctl_r7/r4/U20/Z (ND2B)                       0.08       3.43 r
data_ctl_r7/r4/U27/Z (ND3B)                       0.14       3.57 f
U17/Z (N1B)                                       0.11       3.68 r
U6/Z (N1C)                                        0.17       3.85 f
data_cntl_out[5] (out)                            0.00       3.85 f
data arrival time                                            3.85

clock clk (rise edge)                            12.00      12.00
clock network delay (ideal)                       0.00      12.00
output external delay                           -10.00       2.00
data required time                                           2.00
---------------------------------------------------------------------
data required time                                           2.00
data arrival time                                           -3.85
---------------------------------------------------------------------
slack (VIOLATED)                                            -1.85
```

# Appendix C

# Tools and Reporting Techniques

This appendix gives a brief overview of the synthesis tools used and the commands for timing analysis of the design. The primary synthesis tool used was *Design Compiler* and *Primetime* was used for static timing analysis. *Synopsys Floorplan Manager* was used for the floorplanning of the design.

## C.1 Design Compiler

*Design Compiler* is the core of the *Synopsys* synthesis product family. It provides constraint driven sequential optimization, and synthesizes the HDL description into a technology dependent gate level design. *Design Compiler* optimizes logic design for speed, area, and routability.

*Design Analyzer* is the graphical interface of *Design Compiler*, where as the Unix shell interface is called *dc_shell*.

### C.1.1 How does *Design Compiler* Optimize a Design [1]?

*Design Compiler* works at two levels: the logic level and the gate level. Optimization occurs at both the levels of logic. Figure C.1 summarizes the process of optimization in *Design Compiler*.

The *HDL* description is first read in and converted to a logic level description. The *Design Compiler* applies logic level optimization (flattening and structuring) and

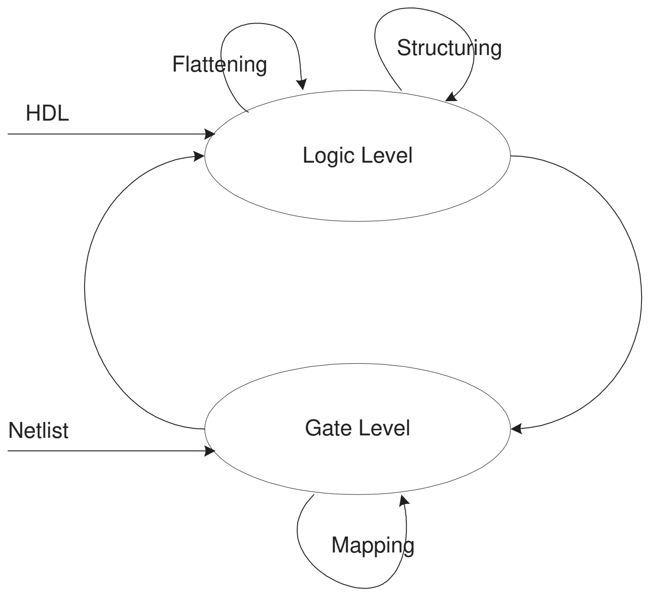


Figure C.1: Optimization process in the *Design Compiler*

maps the resulting structure to gates. Alternatively, if the primary input is a netlist, *Design Compiler* extracts logic equations from it, and optimization is then applied and mapped back to gates.

In a hierarchical design, different levels of the hierarchy can be read from different sources. *Design analyzer* also accepts a design expressed as a combination of gate-level and logic-level constructs.

The *Design Compiler* moves the design between the logic-level and gate-level representations to take advantage of different optimization techniques. The optimization cycle continues till the satisfactory implementation is achieved or no further improvements are possible with the current design.

The logic-level and gate-level optimizations can be controlled using the commands *set_structure*, *set_flatten* or *set map_effort*.

## C.2 Primetime

*Primetime* is a *Synopsys* tool used to generate timing and area reports. This tool can be used to plot the capacitance information, delay information and path based slacks, among other things. Some of the reporting commands used in the Kestrel project are given in the following section.

## C.3 Floorplan Manager

*Floorplan Manager* is a tool in the *Design compiler* family. This tool provides means to produce predictable results at the physical level from the logical implementation. The information from the design (constraints) can be forward annotated onto the design using this tool.

## C.4 Reporting Commands

Various reporting and optimization commands of *Design Analyzer* (*Design Compiler*) and *Primetime* used in the Kestrel project are outline below.

### C.4.1 Design Analyzer

**read** Reads designs into dc_shell.

**analyze** Analyzes *HDL* files and stores the intermediate format for the *HDL* description in the specified library.

**elaborate** Builds a design from the intermediate format of a Verilog module.

**include** Executes a script of dc_shell commands.

**create_schematic** Generates a schematic for the current design.

**current_design** Sets the working design in dc_shell.

**uniquify** Removes multiply-instantiated hierarchy in the current_design by creating a unique design for each cell instance.

**create_clock** Creates a clock object and defines its waveform in the current design.

**set_input_delay** Sets input delay on pins or input ports relative to a clock signal.

**set_output_delay** Sets output delay on pins or output ports relative to a clock signal.

**set_wire_load** Sets the wire loading model for the current_design or for the specified cluster or ports.

**set_false_path** Marks paths between specified points false.

**set_local_link_library** local_link_library attribute is set to the specified file. The local_link_library files are not loaded or checked until the link_library is used.

**set_driving_cell** Sets attributes on input or inout ports of the current_design, specifying that a library cell or pin will drive the ports.

**set_load** Sets the load attribute to a specified value on specified ports and nets.

**set_flatten** Sets or removes the flatten attribute on specified designs or on the current_design, to enable or disable the flattening optimization step during compile.

**set_structure** Sets various structure attributes on a design or on a list of designs, to determine whether and how the designs are structured during compile.

**set_ungroup** Sets the ungroup attribute on specified designs, cells, or references, indicating that they are to be ungrouped during compile

**report_timing** Displays timing information about a design.

**report_area** Displays area information and statistics for the design of the current_instance, if set; or for the current_design otherwise.

**report_reference** Displays information about references in the current_instance, if set; or in the current_design otherwise.

### C.4.2 Primetime

**link_path** Specifies a list of libraries, design files, and library files used during linking. The link_design command looks at those files and tries to resolve references in the order of specified files.

**read_db** Reads one or more design or library db files.

**link_design** Performs a name-based resolution of design references for the specified design_name or the current design.

**source** Reads a file and evaluates it as a script.

**create_clock** Creates a clock object.

**report_design** Lists information about the attributes on the current_design.

**report_reference** Displays information about all references in the current instance or current design.

**set_case_analysis** Specifies that a port or pin is at a constant logic value 1 or 0, or is considered with a rising or falling transition.

**check_timing** Shows possible timing problems for design

**report_timing** Reports timing paths.

# Bibliography


[1] Synopsys Inc., *Design Compiler Reference Manual: Optimization and Timing Analysis*, August 1997.

[2] M. Mergener, "Estimation of processor microarchitecture timing," Master's thesis, University of Wisconsin-Madison, June 1998.

[3] G.S.Sohi, S.E.Breach, and T.N.Vijaykumar, "Multiscalar processors," in *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995.

[4] Department of Electrical and Computer Engineering & Computer Sciences Department, University of Wisconsin-Madison, *Kestrel K-1 Design Specification*, March 1997.

[5] Synopsys Inc., *Guidelines for successful Logic Synthesis*, August 1997.

[6] Department of Electrical and Computer Engineering & Computer Sciences Department, University of Wisconsin-Madison, *Kestrel Architecture Reference Manual*, March 1997.

[7] LSI Logic Corporation, *LCB500K Cell-Based ASIC Products Databook*, September 1997.

[8] Synopsys Inc., *Design Compiler Reference Manual: Fundamentals*, August 1997.

[9] M.Franklin, *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, 1993.

[10] E. Sternheim and R. Singh, *Digital Design and Synthesis with Verilog HDL.* San Jose, CA, USA: Automata Publishing Company, 1993.

[11] S. Palnitkar, *Verilog HDL A Guide to Digital Design and Synthesis.* Upper Saddle River, NJ 07458: Prentice Hall Publishing Company, 1996.

[12] J.L.Hennesey and D.A.Patterson, *Computer Architecture A Quantative Approach.* San Mateo, California: Morgan Kaufmann Publishers Inc, 1995.

[13] T.N.Vijaykumar, *Compiling for the Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison, January 1998.

[14] M.Kantrowitz and L.M.Noack, "Functional verification of a multiple-issue, pipelined, superscalar alpha processor-the alpha 21164 cpu chip," *Digital Technical Journal*, vol. 7, no. 1, 1995.

[15] Department of Electrical and Computer Engineering & Computer Sciences Department, University of Wisconsin-Madison, *Verilog Style Guide*, November 1996.

[16] Cadence Inc., *Verilog-XL Reference Manual*, 2.5 ed.

[17] Synopsys Inc., *Design Analyzer Reference Manual*, February 1996.

[18] Synopsys Inc., *Design Compiler Family Reference Manual*, January 1996.

[19] L. Wall, *Programming Perl.* O Reilly, 1996.

[20] S.Palacharla, N.P.Jouppi, and J.E.Smith, "Complexity effective superscalar processors," in *Proc. International Symposium on Computer Architecture*, 1997.