UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Santa Cruz Instruction Processor With Scoreboarding

Permalink
https://escholarship.org/uc/item/79d417gs

Author
Srivatsaa, Vidyuth

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SANTA CRUZ INSTRUCTION PROCESSOR WITH SCOREBOARDING

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Vidyuth Srivatsaa

June 2013

The Thesis of Vidyuth Srivatsaa
is approved:

____
Professor Jose Renau, Chair

____
Professor Alexandre Brandwajn

____
Mr. Nils Graef

____
Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by
Vidyuth Srivatsaa
2013

Table of Contents

List of Figures v

List of Tables vi

Abstract vii

Dedication viii

Acknowledgments ix

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Superscalar Processors . . . 2
- 1.3 Stage Implementation . . . 2
- 1.4 SCCORE ISA . . . 3
- 1.5 SCIPS Implementation . . . 6

2 Fetch Unit 7

3 Scoreboard 8
- 3.1 Register File . . . 8
- 3.2 Pipelining Hazards . . . 11
- 3.3 Instruction Scheduling in SCIPS . . . 13
- 3.4 Scoreboard verification . . . 14

4 Execute Unit 16
- 4.1 Forwarding . . . 16

5 Data Cache 19
- 5.1 Introduction to Caches . . . 19
- 5.2 Data cache stages . . . 23
- 5.3 Data cache FSM . . . 25
- 5.4 Data cache verification . . . 27

6 Evaluation **29**

7 Conclusion **31**

A Glossary **32**

B Testing and Synthesis **34**

Bibliography **35**

List of Figures

1.1 Simple 2-issue Superscalar pipeline 2
1.2 Retry protocol. 3
1.3 Stage implementation without Retry. 4
1.4 Stage implementation with Retry. 4
1.5 SPARC v8 instruction format 5
1.6 SCIPS in a nutshell 6

3.1 Top level of Scoreboard module 9
3.2 Structure of the Register File 10

4.1 Top level of the Execute unit in SCIPS 17
4.2 Register Forwarding 17
4.3 Forwarding in the Execute unit in SCIPS 18

5.1 Data cache in SCIPS 21
5.2 Top level of data cache in SCIPS 22
5.3 Illustration of skewed associative cache hashing among banks in SCIPS 22
5.4 Cache address structure 23
5.5 Data cache bank structure 24

List of Tables

6.1 Scoreboard synthesis results 29
6.2 Execute Unit synthesis results 30
6.3 Data Cache synthesis results 30
6.4 SCIPS core synthesis results 30

Abstract

Santa Cruz Instruction Processor with Scoreboarding

by

Vidyuth Srivatsaa

This thesis describes Santa Cruz Instruction Processor with Scoreboarding (SCIPS) which is an aggressive 64-bit 2-way superscalar processor with Scoreboard logic implementing the SCCORE ISA designed in System Verilog.

The SCIPS consists of a Scoreboard unit - which serves as the control section to stream instructions to the Execute unit resolving potential hazards; the Execute unit which comprises of 5 different functional units, including a skewed associative data cache.

SCIPS is a flexible pipelined design owing to the *Stage implementation* which tolerates any latency in the pipeline, allowing SCIPS to be re-pipelined at will, helping in significant power savings besides performance improvements.

Dedicated to my close family who have constantly supported and encouraged me.

Acknowledgments

First, and foremost, I would like to thank my adviser, Professor Jose Renau. He has imparted me with excellent knowledge and remarkable support to help me accomplish what I have. He has provided many ideas which has helped me expand my breadth and depth of knowledge in Computer Architecture. Thanks for being patient and always being available for any help!

I would like to thank my thesis reading committee members, Prof. Brandwajn and Mr. Nils Graef, Senior Engineering Manager at LSI for taking the time to read my thesis and for providing valuable feedback in a short notice.

I would also like to thank Jason and Pranav for working on the execute stage; James for helping with the Scoreboard; Elnaz for the Stage implementation.

I would also like to thank Carol for helping me on many occasions.

Lastly, and most importantly, I would like to thank my parents, grandparents and my sister for their constant motivation and encouragement. Big thanks goes to my Uncle, Aunt and Anika.

Chapter 1

Introduction

Mobile processors are ubiquitous owing to a great demand in handheld devices. The challenging aspect in this is to have an energy efficient solution while not compromising on performance.

SCIPS is a general purpose processor implementing the SCCORE ISA maintaining a low power profile and a low silicon footprint, while targeting a good performance point.

1.1 Overview

This chapter briefly introduces the different units of this processor. The chapter also talks about techniques used in the processor design such as the Stage Implementation, has an overview of the Instruction Set Architecture and briefly goes over pipelining, forwarding and superscalar processors.

Chapters 2, 3, 4 and 5 covers the different SCIPS blocks individually. Chapter 2 is about the instruction fetch stage, chapter 3 explains the dynamic scheduling scheme called Scoreboarding used in SCIPS. Chapter 3 briefly covers the Execute unit which is the compute engine for the processor. Chapter 5 elucidates the data cache and goes over the details of a skewed-associative cache that was used in this processor.

Chapter 6 gives a summary of implementation results of the SCIPS core and its sub-blocks from a synthesis tool.

1.2 Superscalar Processors

SCIPS is a superscalar processor. A superscalar processor implements instruction level parallelism within a single processor thereby allowing faster throughput than would otherwise be possible at the given clock rate.

Superscalar architectures include all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage. SCIPS issues 2 instructions in a cycle, so is a 2-way superscalar processor. Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel. Figure 1.1 shows an ideal pipeline flow of such a 2-way superscalar processor.

The issues with superscalar processors are that they are limited to the amount of instruction-level parallelism, complexity and time cost of the dispatcher and associated dependency checking logic, and the extra overhead for branch instruction processing.

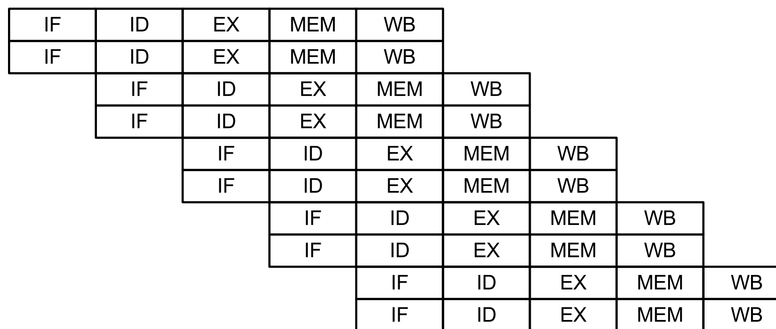


Figure 1.1: Simple 2-issue Superscalar pipeline

1.3 Stage Implementation

The Stage implementation can be understood as a handshaking mechanism which tolerates latency in computations in each module in the system and in communications between the modules. This will allow for transformations which can be used in optimization techniques such as re-pipelining an unbalanced design.

The stage module can be thought of as a FIFO of size 1. Each data input has a corresponding Valid signal, so when the output data comes with an invalid signal, that corresponding

data is discarded and not stored anywhere. In cases where the data does come in with a Valid signal, but gets a Retry from the next stage, then the data is held until that Retry signal is released. When the current module issues a Retry signal to the previous stage simply means that it cannot accept any new data and requests the data to be sent again.

A basic module to module communication with stage signals where one module wants to send data to an another is shown in figure 1.2.

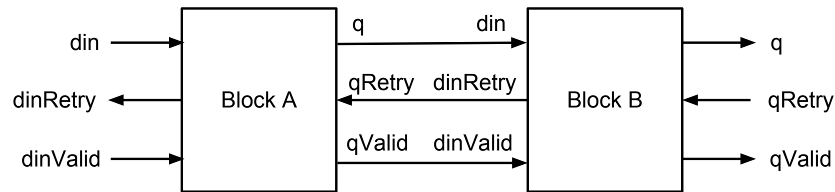


Figure 1.2: Retry protocol.

To illustrate this functionality, below are two timing diagrams, figure 1.3 where the test block doesn't get a Retry on incoming data, and in figure 1.4 when it gets a Retry on incoming data for a couple of clock cycles.

While the stage implementation has a slightly additional area overhead, it eases the pipeline complexity as an entire pipeline can be held in the Retry state without having to worry about losing data. Additionally, the pipeline need not have to be balanced as this implementation gives us the flexibility to have parallel pipelines of different lengths.

1.4 SCCORE ISA

The Instruction Set Architecture (ISA) serves as the boundary between hardware and software. It gives us the ability to precisely describe how to invoke and access the different operations, modes and storage locations supported by the hardware.

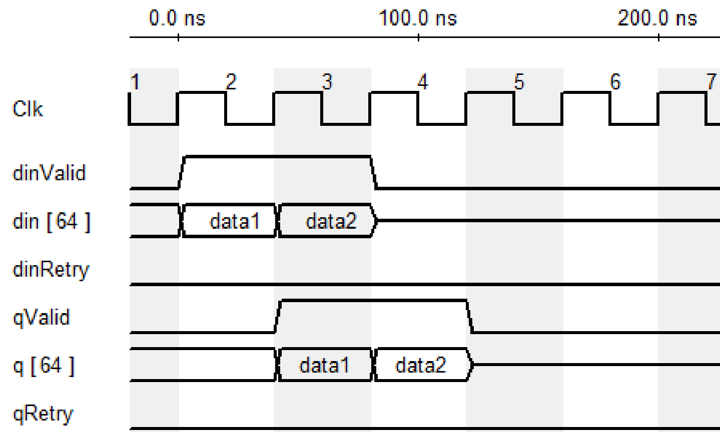


Figure 1.3: Stage implementation without Retry.

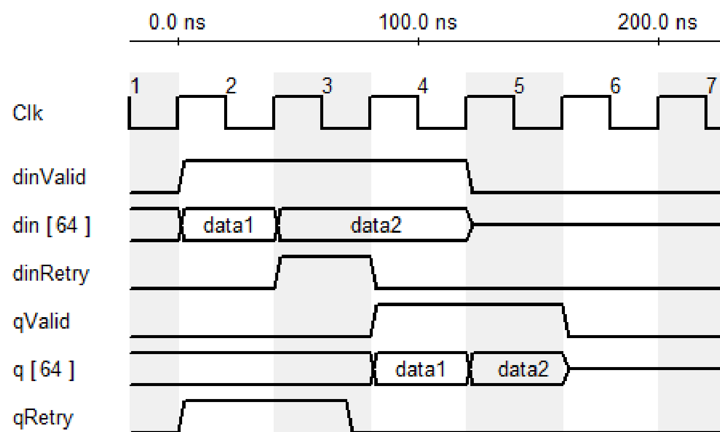


Figure 1.4: Stage implementation with Retry.

The SCIPS implements the pre-decoded SCCORE micro-ops. The SPARCv8 and ARM ISA's are supported to be decoded to the SCCORE ISA. It is a vectored ISA which includes support for ARM's Thumb instruction set which improves the compiled code density by using a compact 16-bit encoding for a subset of ARM instructions.

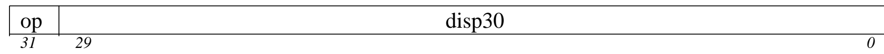
Pre-decoded μOP's include simple ALU, branch ALU, Load/Store ALU, Complex ALU instructions with support for conditional moves. The opcode in SCCORE is a byte wide.

There are 32 integer registers and 32 floating point registers. Additionally, the ISA has support for conditional codes, and hence has an ICC register. There is a floating point

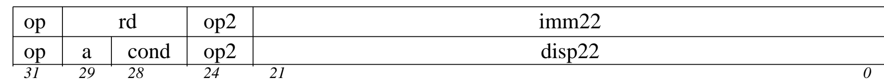
rounding register which specifies the different rounding modes.

The instruction format which is an implementation of the SPARCv8 [12] is shown in 1.5.

Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (*op* = 2 or 3): Remaining instructions

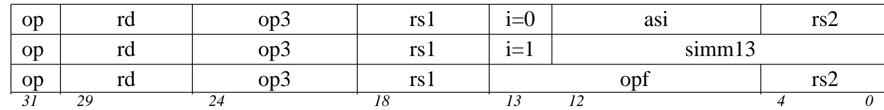


Figure 1.5: SPARC v8 instruction format

1.5 SCIPS Implementation

The overall block digram for SCIPS is below.

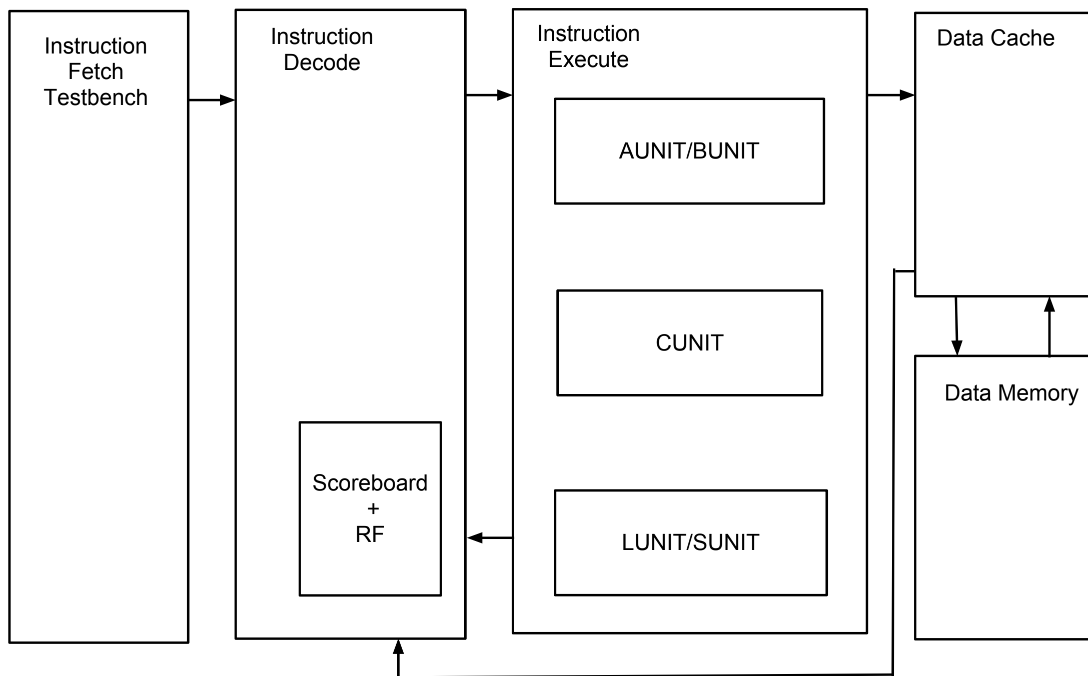


Figure 1.6: SCIPS in a nutshell

The instruction fetch is done with the help of a testbench. The instruction decode unit which comprises the scoreboard, gets the 2 fetch packets from the fetch. This stage resolves all the pipelining hazards and sends the instruction to the appropriate functional unit in the execute stage. The execute stage does all the calculations and sends the results for write back to the register file which is maintained by the scoreboard. It also makes available the forwarded results. In cases of a load or a store instruction, the execute unit provides the data cache with the address, data and memory operation to load/store, acting as a quick lookup to the main memory.

Chapter 2

Fetch Unit

The SCIPS fetch interface is mostly an instruction cache with a very rudimentary branch predictor.

The instruction fetch unit contains the program counter (PC) which makes requests to the instruction memory. When it gets back the response it is passed on to the next stage, the scoreboard unit for address decode.

The fetch keeps providing consecutive instructions until it receives a branch misprediction. In the event of a misprediction, the fetch unit flushes all the current requests and starts to fetch the newly requested PC address. The branch predictor basically chooses between PC+4 and PC+8.

The instruction memory is part of the testbench which takes requests from the Instruction Fetch unit either at the same cycle or later depending on the Retry value responding with a list of instructions starting at the requested address.

Chapter 3

Scoreboard

Scoreboarding is a hardware based dynamic instruction scheduling technique to achieve pipeline efficiency. It is analogous to a book-keeping technique which allows instructions to execute whenever current instructions are not dependent on the previous ones and no structural hazards are present. The scoreboard is a central unit that exchanges information with the instruction fetch stage and the execute stages.

The scoreboard logic takes care of the data dependencies including Write-after-Read and Write-after-Write. The typical stages in a scoreboard based processor are the following:

1. Instruction fetch.
2. Instruction dispatch to an instruction queue. It waits in the queue until its input operands are available.
3. Once made available, the instruction is executed by the Scoreboard's functional unit.
4. The result is written to the destination register in the register file.

The top level of the Scoreboard module is shown in figure 3.1.

3.1 Register File

The register file contains all the processor registers mostly with multiple dedicated read and write ports. It is the register file that ensures the value of the registers have the correct

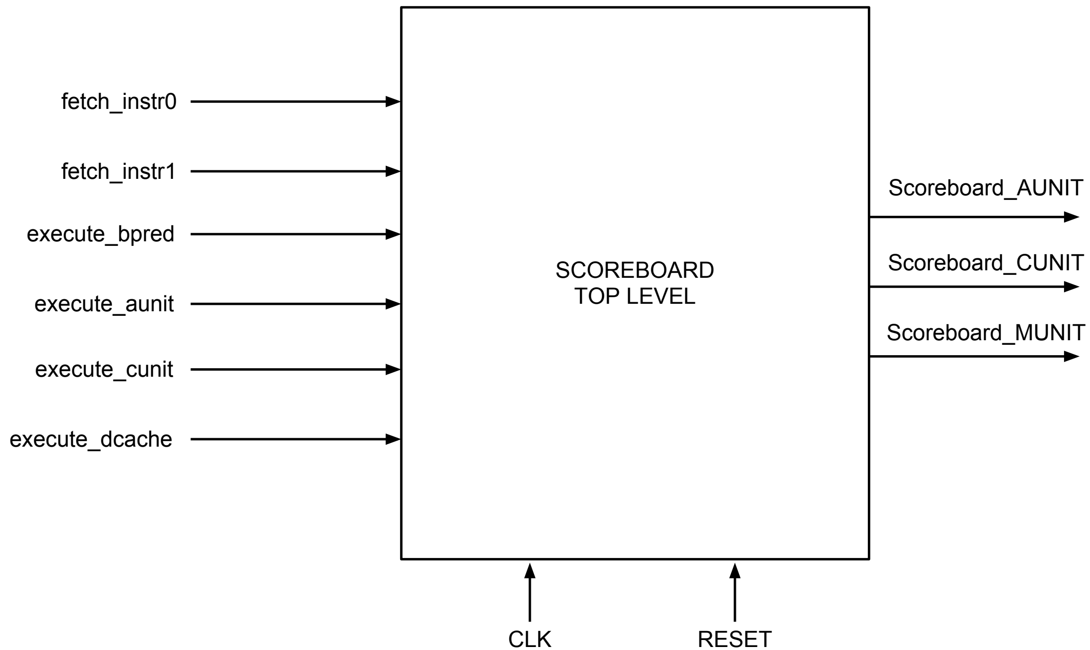


Figure 3.1: Top level of Scoreboard module

values, synchronizing the output destinations of the instruction input. In SCIPS, the register file is maintained by the Scoreboard module.

The register file used in SCIPS is a 4-read 3-write RAM module and is shown in figure 3.2. The three write ports are used to write the results from the different Execute units (AUNIT, CUNIT, LUNIT and SUNIT.) The four read ports are used to read the data stored in the register file. The first two ports are for the source registers of the first fetch pipeline (*src1* and *src2* respectively.) The other two ports similarly are for the lower and higher order source registers of the second fetch pipeline. On any read, the output on the read port will have the sign extended result of the respective source address.

Register file FSM's used:

1. FETCH_NORMAL_REG_QUEUE : This is the default state of the register state machine.

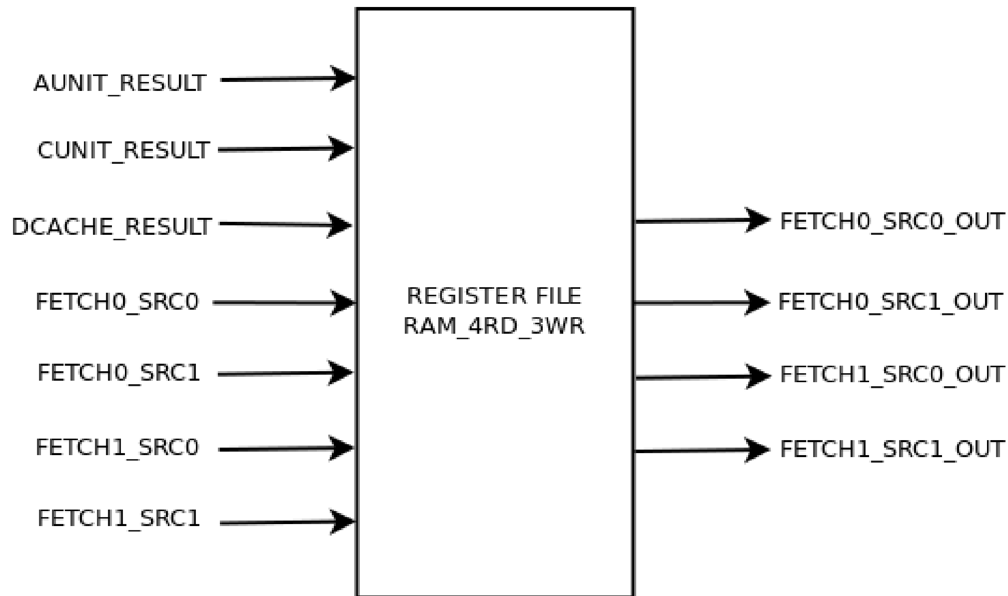


Figure 3.2: Structure of the Register File

Check if instruction is valid. If so, check whether the desired source value is on the register file bus. Read the register file, checking if retry is set. The register file bus needs to be monitored in case the result just shows up. For the case when register is R0, the execute unit is ready for the current queue and the instruction is ready to fetch the next. When register is in the Execute unit, monitor the register file bus in case the result just shows up.

2. EXE_RETRY_SET_WAIT : When the Execute unit is busy, or in other words, when Retry is asserted by Execute, the FSM jumps to this state. Once in this state, it checks if the retry from Execute is de-asserted, which indicates a Execute is done. If so, and if the instruction fetch for that source is also ready, then the state is transferred back to FETCH_NORMAL_REG_QUEUE. If the fetch is still not ready, we need to wait for a return from the Fetch, so the FSM is transferred to EXE_DONE_WAIT_FETCH_RETURN state.

3. FETCH_FROM REGFILE_WAIT : When the requested register is not in the Execute unit, we need to request it from register file. Once the register request is sent, we wait for result to show up. If the Execute sends back a Retry, the FSM goes to EXE_RETRY_SET_WAIT.

If the Fetch is ready, then it transitions to the default FETCH_NORMAL_REG_QUEUE. Otherwise, if the Execute is ready, but the instruction pipe is not yet ready to fetch, then it jump to EXE_DONE_WAIT_RETURN_FETCH state.

4. FETCH_FROM REGFILE_BUSY : When the register file sets a Retry, the FSM transferred to this state. Once that retry is de-asserted, a new request is initiated, and the FSM jumps back to FETCH_FROM_REGFILE_WAIT state.

5. MONITOR_REGFILE : When the register requested is in Execute unit, the register file bus needs to be checked if the result is written back. The result is monitored and latched on to when it shows up.

6. EXE_DONE_WAIT_RETURN_FETCH : Once the Execute unit is done with the current request, but the instruction pipe is not ready to fetch, the FSM is brought into this state. Once the Fetch pipe is ready, the FSM jumps to FETCH_NORMAL_REG_QUEUE state.

The above states are applicable for a single register routine. This is used for all the four source registers, namely, the first fetch pipe's *src1* and *src2*, and the second fetch pipe's *src1* and *src2*. However, the latter two source registers may have a dependency upon the first two due to Read-after-Write (RAW), so an additional state is required for them to stall until the dependency is solved.

3.2 Pipelining Hazards

The scoreboard handles different pipelining hazards such as data hazards, control hazards and structural hazards.

Data hazards occur when instructions that have data dependence modify data in different stages of a pipeline. They must be prevented since they would result in a race condition otherwise, resulting in unexpected output. There are three situations in which a data hazard can occur, that would be handled by the scoreboard.

Read after write (RAW) refers to the situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline. In this implementation, as the source operands are checked, and

will be held in Retry until the write on that register is completed by the earlier instruction until it's new content is available.

```
Instruction i        R2 <- R1 + R3
Instruction i+1      R4 <- R2 + R3
```

Write after Read (WAR) refers to the situation where an instruction tries to write to a destination register before it is read by previous instruction. The scoreboard ensures to preventing such hazards as only valid values of source operands are placed in the instruction pipe where they cannot be overwritten.

```
Instruction i        R4 <- R1 + R3
Instruction i+1      R3 <- R1 + R2
```

Write after Write (WAW) refers to the situation where an instruction tries to write an operand before it is written by previous instruction.

```
Instruction i        R2 <- R4 + R7
Instruction i+1      R2 <- R1 + R2
```

To keep track of WAW, there is a bit vector at decode, which is set when the instruction finishes decode and is cleared once it finishes execute. In the event when the WAW bit is set for a particular register by some previous instruction, the current instruction will be stalled at decode.

Branch instructions are those that directly affect the control logic of the processor. If a branch instruction is currently being executed, there is a stall in decode until the branch predict flag is received.

If the first instruction is a branch and is taken, then that instruction is executed. If the branch was not taken then that pipe is flushed, wait for a cycle and then read the two instructions again.

Where a branch instruction is scheduled in the second fetch pipeline, an external signal is used to notify if the execute units are empty, since monitoring the last instruction wont help.

If a floating point instruction enters, followed by a simple increment instruction, AUNIT is monitored for when it finishes, but not necessarily CUNIT. Similarly, since AUNIT handles jumps, it could easily finish before CUNIT.

In cases where we need to send to execute instructions following the branch before knowing if the branch was correctly predicted, the following instructions needs to be remembered and any writes to the register file or memory should not be performed until that branch is resolved. Otherwise the instructions are allowed to be executed and the order doesn't matter.

There is a state machine in the Scoreboard module which interfaces the fetch FSM to the external Execute unit. It helps in choosing between the two fetch pipelines. This two state FSM simply decides if the output is valid or invalid for each of the pipelines. The output is valid when the Execute unit de-asserts its retry and the register file FSM and fetch are ready.

3.3 Instruction Scheduling in SCIPS

SCIPS being a dual issue processor, with two pipelines. When two instructions are issued, the earlier pipeline will always contain the older instruction in program order and the latter pipeline will contain the newer instruction. So, when an older instruction in the first pipeline cannot be issued, the instruction in second pipeline will not be issued despite of being free from any hazards. In such cases the ”Retry” signal is raised.

This in-order instruction issue and retirement by writing back to the register file helps in completely preventing WAR hazards and keeping track of WAW hazards and recovering easily from any flush condition.

Result data is forwarded from the outputs of AUNIT, BUNIT, CUNIT and the data cache. This data is available for any instruction that requires it as soon as it is produced. Additionally, once this data is available for forwarding, it will continue to be available while the instruction is in flight until its result is written to the register file. This way the read after write hazards can be mitigated.

One approach to restart relies on processor hardware to maintain a simple, well-defined restart state that is consistent with the state of a sequentially-executing processor [11]. In this case, restarting after a point of incorrect execution requires only a branch (or similar change of control flow) to the point of the exception, after the cause of the exception has been corrected. A processor providing this form of restart state is said to support precise exceptions

(or interrupts). The problem with normal pipelining is when there are data hazards, there will be stall.

If the pipeline can (or must) be stopped, all the instructions before the faulty (or intended) instruction must be completed. All the instructions after it must not be completed and be able to restart the execution from the faulty (or intended) instruction. The scoreboard mechanism helps to stall in such cases until no exception of prior long-latency instruction is guaranteed.

Below is such an example where DIVISION operation, which is a considerably long-latency operation, ends in an exception. The next two are simple ADD instructions, that would complete even before the division ends up with an exception, and hence must never happen. Using the scoreboard mechanism, this issue can be successfully prevented.

```
DIV F0, F1, F2 (exception)
ADD F3, F1, F5 (completed)
ADD F6, F6, F7 (completed)
```

It is good to have precise exceptions because it enables us to easily recover from exceptions such as page faults. It also helps enabling traps into software and aids in easier software debugging.

3.4 Scoreboard verification

We have seen that the scoreboard unit interfaces with the execute unit and the data cache, also containing the register file. Hence, the following were tested for this block.

Firstly, it is easier to simply test the Register file separately. When the source operand is 0, it implies the R0 register and is ignored. When the WAW bit on a register is observed to be set, there should be no update to the register file. When the data valid bit is set, it implies an update to the register file, and the data needs was checked if it gets the correct value. If the WAW bit is not set, on a register file read, the correct value is expected.

Next, the instruction fetch part of the scoreboard was tested for. When both the fetch pipes are invalid, the scoreboard waits for a valid instruction. When both the fetch pipes are valid, both instructions were scheduled, and on any dependencies, the FSM jumped to the right states to stall accordingly. When the first fetch pipe is valid, but the second is invalid, the first

was sent to execute. When the first fetch pipe is invalid, but the second is valid, the scoreboard stalled waiting for the first to go valid.

If the first instruction is a branch and is taken, that instruction was executed. If the branch was not taken then first fetch pipe was flushed, and after a clock cycle the instructions were read in again.

If the first instruction is a non-branch while the second is a branch, then the control waits to see if the branch was taken. When the branch was taken, the scoreboard sent it to execute. When it was not taken, the FSM waits for one cycle.

Chapter 4

Execute Unit

SCIPS is a clustered architecture having five main units which comprise the Execute unit. This Execute unit is responsible for all the calculations of the processor working as the functional unit of the Scoreboard.

The arithmetic unit (AUNIT) is responsible for all the simple arithmetic operations such as basic *ADD*, *SUB*, *SHIFT*, *AND*, *XOR*, etc. The branch unit (BUNIT) is responsible for branch operations such as Branch if not equal to zero (BNE), Branch if greater than or equal to zero (*BGE*) etc. The complex unit (CUNIT) contains a full IEEE compliant floating point unit (FPU) which handles all the floating point operations such as floating point *ADD*, *SUB*, *MULT*, *DIV* and square root. Besides floating point *MULT* and *DIV*, the CUNIT also does the simple multiplication and division operations. The memory units are split between the load (LUNIT) and store (SUNIT) unit and interface with the data cache which in turn handles all load and store operations. All these units are all single ported in SCIPS.

The top level of the Execute module can be seen in the figure 4.1

4.1 Forwarding

The execute unit implements the forwarding logic by feeding back the output of the instruction to the previous stage of the pipeline as soon as the output of that instruction is made available.

Figure 4.2 illustrates register forwarding where the result data after a successful execution of an instruction is forwarded to be used by the immediately following instruction.

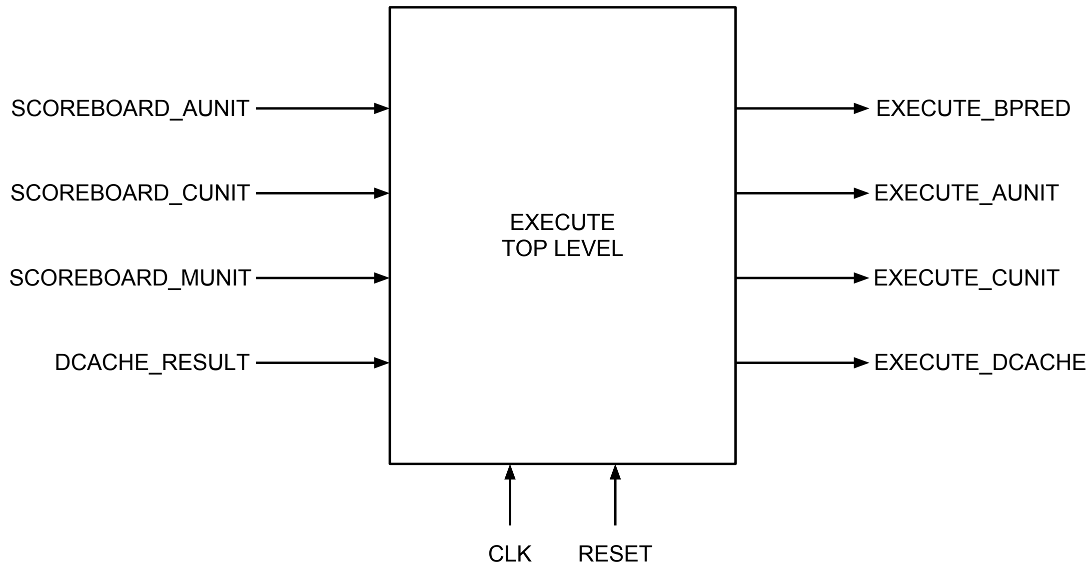


Figure 4.1: Top level of the Execute unit in SCIPS

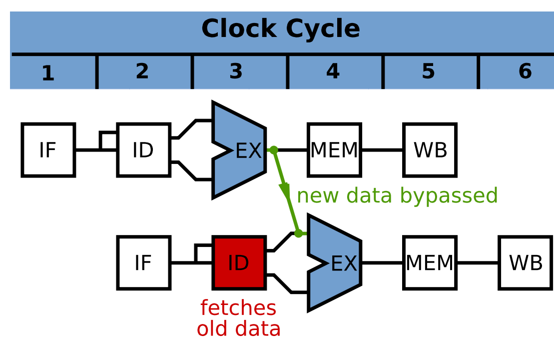


Figure 4.2: Register Forwarding[1]

The following figure 4.3 shows the forwarding implementation for the execute stage.

[1] http://en.wikipedia.org/wiki/File:Data_Forwarding_(One_Stage).svg

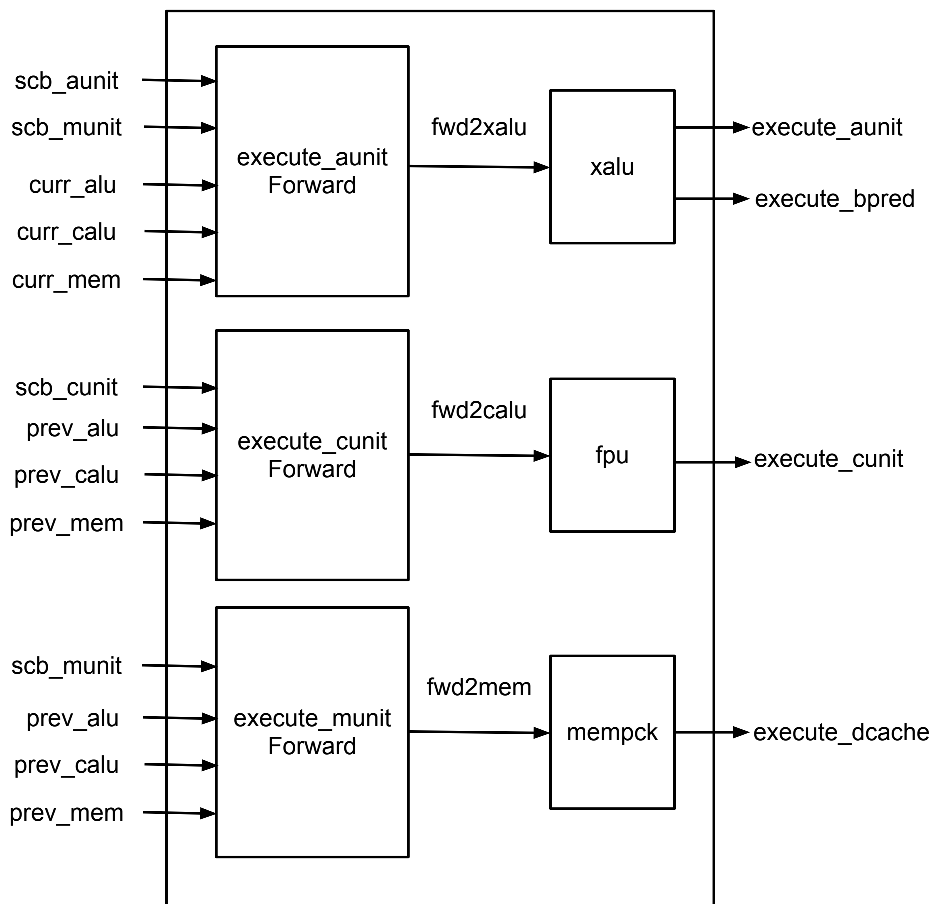


Figure 4.3: Forwarding in the Execute unit in SCIPS

Chapter 5

Data Cache

5.1 Introduction to Caches

Cache is a relatively small memory that are high speed since it eliminates the access to the main memory from the CPU. By acting as a gateway to memory, cache serves to store data most recently used by the CPU. Accessing RAM is comparably slow compared to the clock speed of a CPU, and thus it is very advantageous to have the required data as close as possible (on the same chip as the CPU). Registers on the CPU do this as well but are limited in number. To limit the amount of latency the majority of memory accesses should be through the CPU cache, thereby preventing stalls on the CPU. Processor access to memory that is not in the cache is called a cache miss, and when its found, is called a cache hit. Most caches in processors use set-associative caching, which is a compromise between direct mapped cache and a fully associative cache. SCIPS uses a 3-way skewed associative caching, where a memory address can map to multiple cache addresses by employing distinct indexing mechanisms for various ways. This further reduces cache misses, however coming at the cost of latency used to calculate the hashed addresses.

Banks are often used in caches as they are useful in many ways. Banking storage is substantially less expensive than adding access ports. For a common case where two accesses are unlikely to map the same bank out of n banks, it also helps in reducing the power consumption since the other banks need not be accessed. However, banking can introduce structural hazard when there is a bank conflict, but this is mitigated by the stage implementation. SCIPS has an 8-bank implementation of the data cache.

Skewed associativity was introduced for caches by Seznec [1]. Seznec, along with Bodin and others, examined the concept in further detail in several papers [1] [5] [6] [9] and Michaud [8] described it from a theoretical statistical point of view. Seznec also elaborated on the choice of replacement strategies for skewed caches and BTB's [10].

Skew-associative caches typically exhibit lower conflict misses and higher utilization than a set-associative cache with the same number of ways [6].

The behaviour of skewed associative caches enhances the performance of blocked algorithms that iterate computations on a small working set smaller than the cache size in which interference misses may degrade performance a lot [7].

In this skew cache, way 0 is indexed using the lower address bits just like in a direct-mapped cache. Meanwhile, the indices for way 1 and way 2 are generated using a hashing function. For way 1, the address bits are XOR'ed with the address bits left-shifted by 1. Finally, for way 2, the address bits XOR'ed with the sum of address bits left-shifted by 1 and masked value of address bits.

$$H_0(A) = A$$

$$H_1(A) = A \oplus (A \gg 1)$$

$$H_2(A) = A \oplus ((A \gg 1) + (A \;\&\&\; \mathit{0xFFFF}))$$

However, skewed associative caching complicates the replacement policy for when a memory address is swapped out. State will be kept to determine the last recently used cache location usually. Th top level of data cache used in SCIPS can be seen in the figure 5.2

LRU replacement is considered an efficient approach for the cache replacement policy. However, implementing it for a skewed associative cache is tricky since it is difficult to get concise information to associate with a cache line that would allow a simple hardware implementation (as the set of lines on which a line has to be replaced vary with the new line to be introduced, the information needed in order to determine the latest line referenced in the set is the complete date of reference.)

A simple replacement policy which requires only one tag bit per cache line (recently used) can be more effective and simpler to implement.The other policy is to introduce some randomness in selection. Using such a pseudo-random policy however induces more misses on caches than the LRU policy.

The replacement reason is that if a line is evicted from the skew cache and then re-

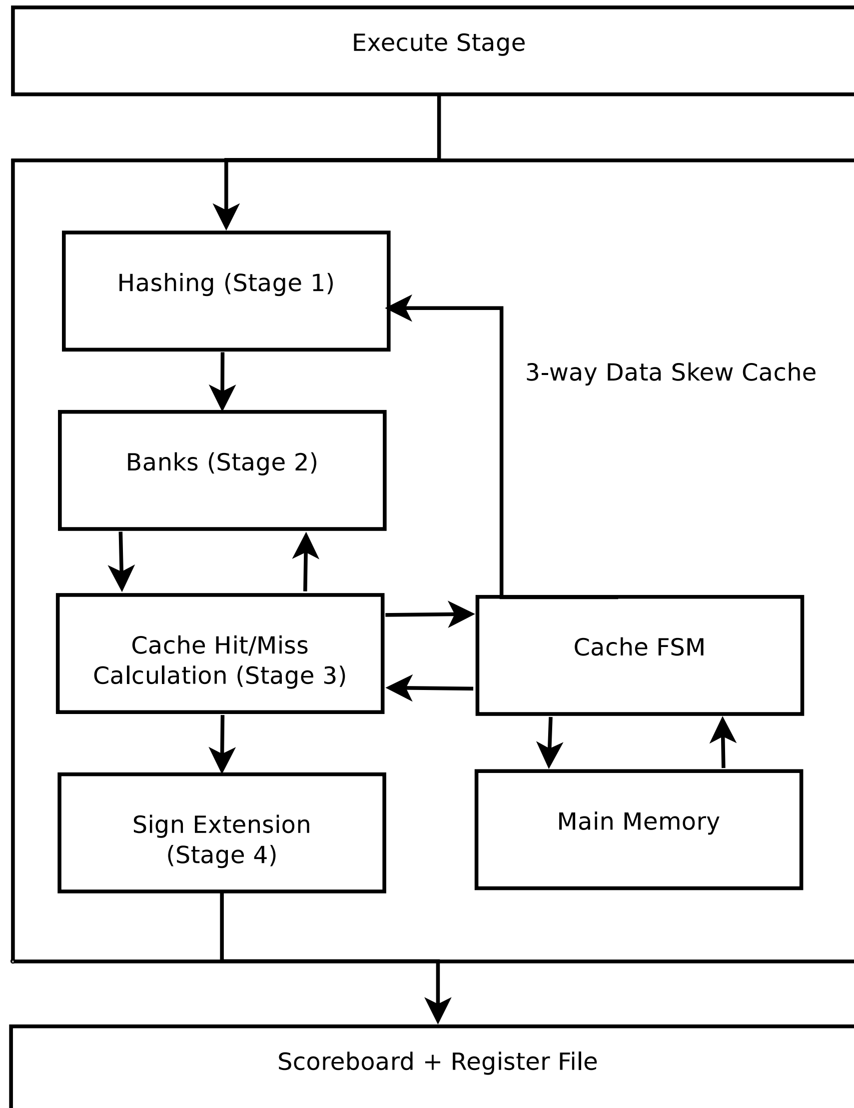


Figure 5.1: Data cache in SCIPS

referenced, it might occupy a different location than it did the first time. For the sake of tradeoff, we choose a modified version of MRU replacement, which is the opposite of the LRU policy. With this policy, a bit is set on a cacheline access. When it comes to the point of a replacement, if the replacement bit value is low, then that line is evicted, and if two of the three possible ways have their replacement bit low, then the evicted line is randomly chosen among the two. These replacement bits are cleared time to time to avoid stagnation.

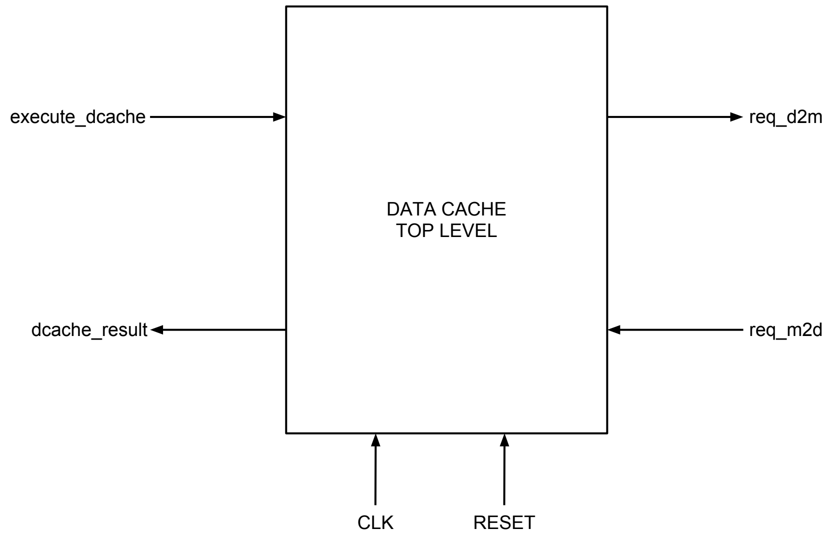


Figure 5.2: Top level of data cache in SCIPS

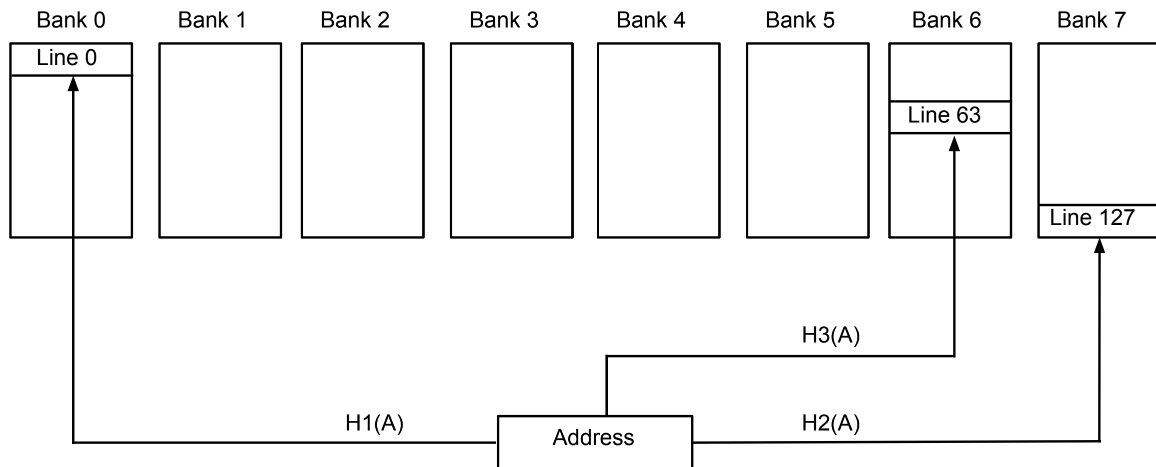


Figure 5.3: Illustration of skewed associative cache hashing among banks in SCIPS

5.2 Data cache stages

There are 4 stages in the structure of this data cache which can be seen in the figure 5.1. Data memory is similar to Instruction memory described earlier. It is a part of the testbench which handles requests from LUNIT/SUNIT and responds back with appropriate data on read requests that aren't already on the data cache.

The different stages are described below:

Stage 1 (Hashing)

1. When the execute unit requests an address load, the data cache gets this address. It splits this address into tag, index, bank offset, word offset and byte offset as shown below.

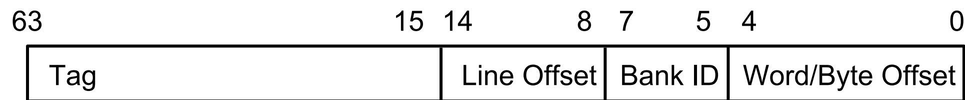


Figure 5.4: Cache address structure

2. The address is hashed for usage in different ways of the skew cache. For way0 the address is as is, behaving like a direct-mapped cache. For way1, the address XOR'ed with a shuffle, both of which are cheap hardware operations, and for way 2, the address is combination of XOR, shuffling and a simple addition with bit masking, as shown previously.

3. The destination register and the memory opcode are copied and will be forwarded to the next stage.

4. Depending on the memory operation, the cache's stage 1 decodes whether it's a load or store instruction from the fourth bit of the memory opcode. Accordingly, when the bit is high it decodes to a store, while is a load instruction if the bit is low.

5. With the stage implementation, it accepts new requests only when the valid bit is set by the execute unit.

6. Additionally, the result from the cache FSM is forwarded to this stage to write to SRAM banks in the next cycle.

7. From the MRU bit value from earlier stages, the appropriate hash is used to index this replacement candidate to be stored in the bank.

Stage 2 (SRAM Banks)

1. Stage 2 is essentially where the SRAM banks for tags and data reside. Additionally, the replacements bits for each of cache line for every bank is maintained here, and looked up.

2. It gets the hashed bank values which comprises the bank vectors.

3. Each bank can send a retry at any time, and in such cases it gets propagated to the previous stage.

4. Every tag and data in each of the banks have a valid signal as well, and they all get a retry from the next stage.

5. This data cache bank structure is shown below in 5.5

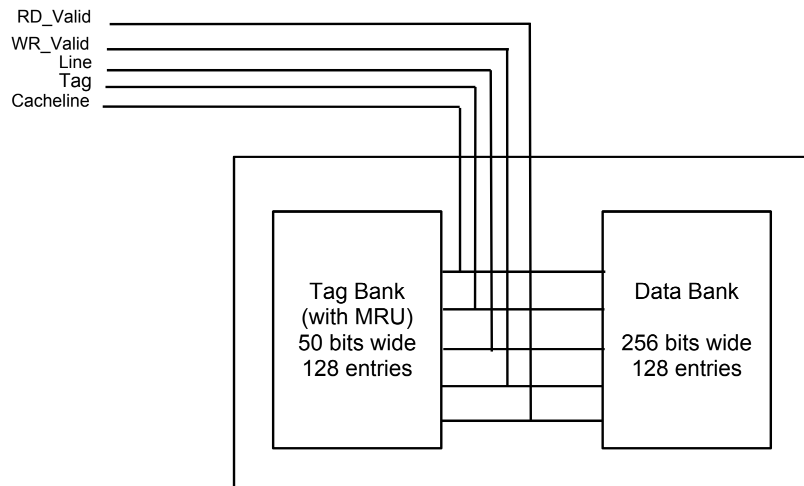


Figure 5.5: Data cache bank structure

Stage 3 (Hit or Miss calculation)

1. In this stage, the cache evaluates whether there was a hit or a miss in the cache.
2. To check this, the cache requesting address' tag is compared with the tag that resides in each of the banks. The lookup takes place in parallel, and if there is a match, the logic for hit signal is raised. If there's no match, then it's considered as a cache miss and the logic for miss signal is raised.
3. In the event of a cache read hit, it moves on to the next stage. In all other cases, such as, read miss, write hit and write miss, the flow proceeds to an FSM explained in the next part.
4. Additionally, on a hit, it updates the particular cache line that it recently had a hit, which will be used as basis for cache line replacement.

Stage 4 (Sign extension)

1. This is the final stage where the sign extension is carried out.
2. The cache request enters this stage only when there was a load request, after processing the hit in the cache.
3. According to the byte offset, the appropriate byte(s) are loaded to the result. This is sent to the scoreboard unit for it to write back to the register file.

5.3 Data cache FSM

The L1 data cache FSM can go to either of the following 5 states:

1. Idle
2. Read Miss
3. Write Hit
4. Write Miss
5. Memory

IDLE state is the default state of the FSM. When there's a reset, the FSM jumps to IDLE state. When theres no activity, the FSM transitions to this default state.

On a read miss, we copy the address of the cache miss to the data memory. The memory command is updated with READLINE. Once the memory request Valid flag is asserted, the FSM jumps to the Memory state. If not, the FSM is going to check if there was a miss and whether the currently handled instruction was a load, and then the FSM will continue to stay in the Read Miss state. This will happen when the dcache FSM block gets a Retry or if the memory is not ready yet, then Read Miss state will have to wait. If neither of these are the case, then the FSM jumps to the default IDLE state.

On a write hit, we copy the address of the cache hit to the memory. The memory command is updated with PUSHLINE, since the cache wants to write the line that was written in the cache to the memory. Next, we check against the bank offset to copy the respective data from the bank to the data memory data.

Once the memory is idle and holds up its Valid bit, the state transitions to MEMORY state. If it's not Valid yet, the cache checks if it still has the cache hit and if the currently handled instruction is a store, and in that case the state will remain in Write Hit. Otherwise, the state transitions to the default IDLE state.

A write miss occurs when execute wants to write some data in a particular address location, and the tag corresponding to the particular address is not available in the cache. In such a case, the FSM state jumps to Write Miss. Initially, the address in case is passed on to the memory using the cache-to-memory request. The memory command READLINE is set on the cache-to-memory request's command for updating it. Next, the bank offset is checked and matched as to which bank has the data, and that data is passed on to cache-to-memory request's data.

Once memory is idle and holds its Valid bit, the state jumps to memory. However, if it's blocked and can't move forward due to some pending Retry, we check for a cache hit again and if the instruction was a store request, and continue to stay in the Write Hit state. Otherwise, we move to IDLE state.

We have seen that when the cache comes across a read miss, write hit or a write miss, the cache FSM transitions to the respective states and eventually transitions to the MEMORY state.

Now, in this MEMORY state, irrespective of the different possible states that the FSM is in, once the memory request Valid flag is raised, we can derive the data, address and request size for the miss handler results data structure. Additionally, in case the currently handled instruction is a store instruction, then the request size is saved.

The memory can also send the data cache an invalidate signal to notify that it has an updated entry and that whatever is in the cache is outdated. In such a case the memory signals an INVALIDATE command so that the cache sees it to invalidate the line entry. It sends an acknowledgment for the same with a NODATA_ACK command.

5.4 Data cache verification

The data cache was tested for a number of test scenarios. The basic test is writing to an address and reading from an address. Next, it was tested if was able to snoop the bus properly, for hits, misses and back-to-back misses.

More specifically for the skewed cache, testing whether the addresses have been hashed correctly needs to be done. Importantly, we need to check if the replacement candidates were correctly implemented.

To test the FSM, the initial loads will have a cache miss. If the cache gets a new load request in the kth clock cycle, at $k+1^{th}$ clock, the appropriate cache tag banks get the lookup request. In the next cycle, depending on the tag from the banks and the actual tag, a cache miss occurs, and the FSM gets a new request. In cycle $k+3$, the FSM jumps from the RD_MISS state as the tag was not found in the banks, and jumps to the MEMORY state where the data memory fills in the cache with the load value several cycles later. Both stage 1 and stage 3 gets this new load data. Stage 1 sends it to store to the bank, and simultaneously, Stage 3 sends it for sign-extension to later send it out to the Scoreboard.

When a load to the same address is requested again at cycle j, bank lookup request is done at cycle $j+1$, and in the next cycle, the cache evaluates to a hit, gets the data, and in $j+3$ goes through sign-extension. Eventually, at $j+4^{th}$ clock cycle, the result is sent out to the Scoreboard.

All store requests goes to the memory. If the corresponding tag was not found in the cache, the state machine goes to WR_MISS state and then onto MEMORY state. If the tag was found in one of the banks, then it goes to WR_HIT and the onto MEMORY. This is because the

newly stored data needs to be updated in the data memory as well.

To test the eviction mechanism, the cache was able to replace cachelines for those whose MRU bit was unset. On hits, these bits were set to indicate that the eviction must not consider them for eviction.

Chapter 6

Evaluation

The test scenarios for the SCIPS blocks were written in C/C++ and interfaced with the RTL through PLI (Programmable Language Interface.) The testbench has functions to set the DUT input parameters and another function to check if the hardware and software results matches.

The SCIPS core was synthesized using Synopsys Design Compiler using the SAED32 32nm process library under normal operating conditions. The synthesis results are indicated in Table below. The design was targeted for a frequency of 1 GHz with the critical path in the design running through the complex unit.

Below are the synthesis results for the main blocks in the processor and the overall SCIPS core put together.

Parameter	Synthesis Results
Time period (ns)	0.66
Maximum Frequency(GHz)	1.50
Non-combinational Area (μm^2)	79707.18
Combinational Area (μm^2)	163408.24
Total Area (μm^2)	243115.42
Average clock toggle rate	0.2088
Average register gating efficiency (%)	79.1
Total Power (mW)	559.48

Table 6.1: Scoreboard synthesis results

Parameter	Synthesis Results
Time period (ns)	1.00
Maximum Frequency(GHz)	1.00
Non-combinational Area (μm^2)	59878.11
Combinational Area (μm^2)	142549.11
Total Area (μm^2)	202427.22
Average clock toggle rate	0.97496
Average register gating efficiency (%)	2.5
Total Power (mW)	449.48

Table 6.2: Execute Unit synthesis results

Parameter	Synthesis Results
Time period (ns)	0.91
Maximum Frequency(GHz)	1.10
Non-combinational Area (μm^2)	1699208.32
Combinational Area (μm^2)	816380.67
Total Area (μm^2)	2515588.99
Average clock toggle rate	0.1558
Average register gating efficiency (%)	84.4
Total Power (mW)	1290

Table 6.3: Data Cache synthesis results

Parameter	Synthesis Results
Time period (ns)	1.00
Maximum Frequency(GHz)	1.00
Non-combinational Area (μm^2)	1756144.45
Combinational Area (μm^2)	946537.46
Total Area (μm^2)	2702681.91
Average clock toggle rate	0.1854
Average register gating efficiency	81.46
Total Power (mW)	2540

Table 6.4: SCIPS core synthesis results

Chapter 7

Conclusion

This thesis describes the processor architecture of SCIPS which can tolerate latencies. It simplifies pipelining complexities at the cost of a minor area overhead, capable of achieving the targeted performance. The SCIPS core reaches upto 1 GHz using the SAED32 process consuming a total area of 2.7 mm^2.

Appendix A

Glossary

Santa Cruz Instruction Processor with Scoreboarding (SCIPS). – SCIPS is the in-order processor been described in this thesis.

Program Counter (PC). – PC is the program counter which is also the instruction address.

Device Under Test (DUT). – DUT is the device under test.

Programmable Logic Interface (PLI). – PLI is the way by which the high level language talks with a low level language.

Read After Write (RAW). – RAW is a data hazard in pipeline where the current instruction has to read an operand only after the previous instruction finishes writing to it.

Write After Write (WAW). – WAW is a data hazard in pipeline where the current instruction tries to write an operand before it is written by previous instruction.

Write After Read (WAR). – WAR is a data hazard in pipeline where the current instruction tries to write a destination before it is read by previous instruction.

Finite State Machine (FSM). – FSM is a state machine used to design digital logic circuits, which can be conceived as an abstract machine that can be in one of a finite number of states.

Arithmetic Unit (AUNIT). – The AUNIT is responsible for all the simple arithmetic operations.

Branch Unit (BUNIT). – The BUNIT is responsible for all the branch operations.

Complex Unit (CUNIT). – The CUNIT is responsible for all the simple floating point operations.

Load Unit (LUNIT). – The LUNIT is responsible for all the load operations.

Store Unit (SUNIT). – The SUNIT is responsible for all the store operations.

Least Recently Used (LRU). – Least recently used is one of the commonly used replacement policies for cache replacements.

Most Recently Used (MRU). – Most recently used is one of the replacement policies for cache replacements.

Appendix B

Testing and Synthesis

SCIPS adopts a Ruby Makefile for running tests and synthesis. To run a test on the command line, do the following.

```
rake test:scips_tb
```

The following command line kicks off the synthesis for the SCIPS core using Design compiler on 32nm library with a constrained frequency of 1 GHz.

```
rake asic:scips frequency=1000 tech=32 suite=dc
```

To get the clock gating efficiency for the SCIPS core, we first need to have a dump of the design containing switching activities. To do a SAIF dump, use the following

```
rake test:scips SAIF=1 args=``+clk=2``
```

Once we have the switching activity file, the following command kicks off a PrimeTime run which reports the consumed power along with clock gating information.

```
rake asic:scips frequency=1000 tech=32 suite=pt
```

Bibliography

[1] Andre Seznec. A case for two-way skewed-associative caches. In *International Symposium on Computer Architecture, 1993. Proceedings*, pages 159 – 192, 1993.

[2] A. Seznec and F. Bodin. *Lecture Notes in Computer Science*. Parle-Springer-Verlag, volume 9, pages 305 – 316, 1993.

[3] Daniel Sanchez and Christos Kozyraki. The ZCache: Decoupling Ways and Associativity. In *IEEE MICRO 43*, pages 159 – 192, 2010.

[4] Thorild Selen. Reorganisation in the skewed-associative TLB. Masters Report, Uppsala University, 2004.

[5] F. Bodin and Andre Seznec. Cache Organization influence on loop blocking. Technical Report, IRISA, 1994.

[6] F. Bodin and A. Seznec. Skewed Associativity enhances performance predictability. In *International Symposium on Computer Architecture, 1995. Proceedings*, pages 265 – 274, 1995.

[7] M. Lam, E. Rothberg and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *IEEE ASPLOS*, volume IV, pages 159 – 192, 1991.

[8] Pierre Michaud. A Statistical Model of Skewed-Associativity. *High Performance Computer Architecture*, volume VII, pages 19 – 24, 2001.

[9] Nathalie Drach, Alain Gefflaut, Philippe Joubert and Andre Seznec. About Cache Associativity in low-cost shared memory multi-microprocessors. Technical Report, IRISA, France, 1993.

[10] Mathias Spjuth, Martin Karlsson and Erik Hagersten. Skewed Caches from a Low-Power Perspective Proceedings of the 2nd conference on Computing frontiers, pages 152 – 160, 2005.

[11] J. Smith and A. Pleszkun. Implementation of precise interrupts in pipelined processors. In *International Symposium on Computer Architecture, 1993. Proceedings*, pages 36 – 44, 1985.

[12] The SPARC Architecture Manual Version 8. http://www.sparc.com/standards/V8.pdf