Design and Implementation of Fetch, Decode and Issue Stages for RISC-V Based 32/64 bit Superscalar Processor

A Project Report

submitted by

SACHIN RAMRAO WAGHMARE

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

MAY 2014

THESIS CERTIFICATE

This is to certify that the project report titled **Design and Implementation of Fetch, Decode and Issue Stages for RISC-V Based 32/64 bit Superscalar Processor**, submitted by **Sachin R. Waghmare**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this project report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

Successful completion of this project work would not have been possible without the guidance, support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to them.

First and foremost I offer my earnest gratitude to my guide,***Dr. V. Kamakoti*** who has supported me throughout my project work with his patience and knowledge while allowing me freedom and flexibility to follow my own thought process.

My special thanks to ***Mr. G.S. Madhusudan*** who patiently listened, evaluated, criticized and guided me periodically. The invaluable inputs and suggestions from him were instrumental in steering the project work in the right direction.

I would like to thank my faculty advisor ***Dr. Deleep R.Nair*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to project team members Neel, Naveen, Rishi Naidu, Arjun, Senthil, Sarath, Chaitanya, Keerthi for their help and support.

ABSTRACT

KEYWORDS: Fetch ; Decode; Issue; ICOUNT; Instruction Queue, TLB

In modern superscalar processor the issue bandwidth is high compared to the fetch bandwidth. In this case feeding the execution units optimal or useful instructions becomes very important. In other words the fetch policy affects the performance of a processor to a great extent. In this thesis I have explained the design of the Fetch, Decode and Issue stages which minimize the IQ clog conditions at the expense of separate IQs for separate threads and its control logic which also enables the independent flow of the threads through the pipeline. To provide the execution unit with most useful instructions the thread select logic is divided into two stages which enables the control logic to work on more recent data.

The Fetch and Issue policies used are ICOUNT and Round Robin. First of which is performance oriented and later one leads to simpler design.Finally synthesis reports of the design are provided on both FPGA and ASIC platforms.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

IQ	Instruction Queue
TID	Thread ID
CPU	Central Processing Unit
FGMT	Fine Grained Multithreading
CGMT	Coarse Grained Multithreading
SMT	Simultaneous Multithreading
PC	Program Counter
FPU	Floating Point Unit
ALU	Arithmetic Logic Unit
BSV	Bluespec System Verilog
ICOUNT	Instruction Count
BRCOUNT	Branch Count
RS	Reservation Station
RR	Round Robin
ROB	Reorder Buffer
IPC	Instructions per Clock Cycle
MP	Multiprocessing
FU	Functional Unit
TLB	Translational Look Aside Buffer

CHAPTER 1

INTRODUCTION

1.1 Processor Architecture

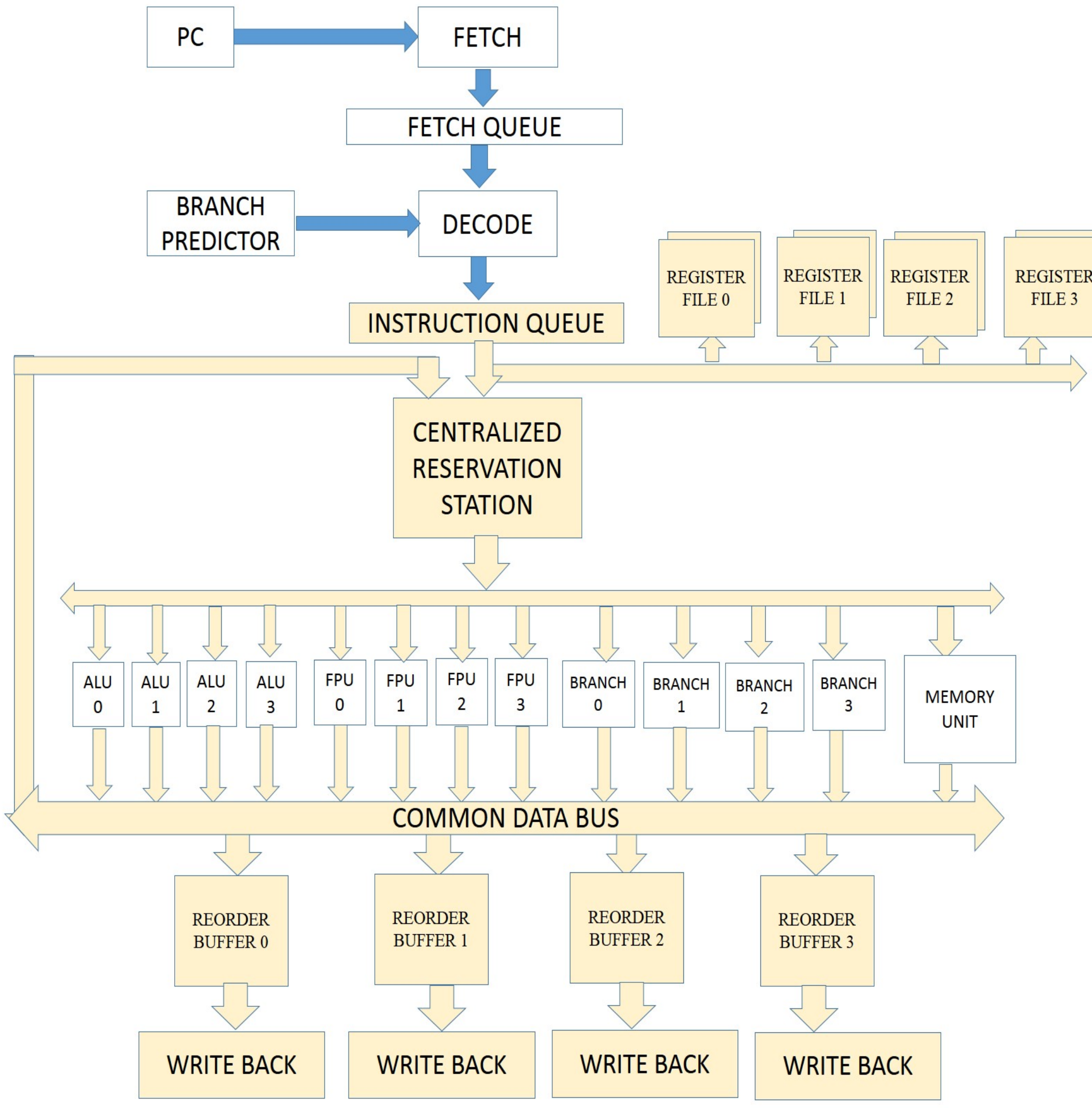


Figure 1.1: Microarchitectural description of the Superscalar Processor

1.1.1 Overview of Processor

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Dept. of IIT Madras has been actively involved in building a Superscalar processor for Server applications. The proposed processor is a 64 bit, single core, quad threaded superscalar processor. The processor strictly follows RISC-V Instruction set Architecture (ISA). The entire design of the processor is done using a Hardware Description Language (HDL) called Bluespec System Verilog (BSV).

The CPU core is based on the Tomasulo Algorithm. The Microarchitecture of the CPU core is an Out of order microprocessor that is capable of fetching, decoding and issuing 4 instructions every clock cycle. A Centralized Reservation station is implemented to utilize the reservation station entries in a more efficient manner. The Execution units are duplicated to support for Quad issue. The Common Data Bus (CDB) used for Operand forwarding can forward the results of at most 4 execution units. The Reorder buffers are designed to commit maximum of four instructions in every clock cycle. My project work involves the design of issue, dispatch, CDB and commit stage related to the CPU core.

1.1.2 My Contribution

I contributed to the fetch, decode and issue stages of the superscalar processor. The main challenge in implementing these stages was doing it for a hardware supported multi-threaded processor capable of handling four simultaneous threads. The main feature of my work is separate IQs for multiple threads which result in less IQ clog conditions and divided logic for selecting thread for fetch and for issue which results in more useful instructions every cycle. I have implemented the quad thread SMT core in two policies which are Round-Robin and ICOUNT.The first one results in less complexity design but lacks in performance when compared to ICOUNT.Also these stages are also implemented separately for 32 bit and 64 bit processor. I have also verified the functional correctness of these units using specially designed test benches which were able to provide processor like environment to these stages.Finally I synthesized these units on both FPGA and ASIC platforms synthesis results of which are provided in results section.

1.2 Introduction to Multithreading Techniques

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques.What is achieved is thread-level parallelism, which may provide for greatly improved performance when married to instruction-level parallelism.

1.2.1 Fine Grained Multithreading

This technique switches between threads on each clock, causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles. The primary disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread, since a thread that is ready to execute without stalls will be delayed by instructions from other threads. It trades an increase in multithreaded throughput for a loss in the performance (as measured by latency) of a single thread.

1.2.2 Coarse Grained Multithreading

It was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two or three cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: It is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs the pipeline will see a bubble before the new thread begins executing. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time. Several research projects have explored coarse grained multithreading, but no major current processors use this technique.

1.2.3 Simultaneous Multithreading

It is a variation on finegrained multithreading that arises naturally when fine-grained multi-threading is implemented on top of a multiple-issue, dynamically scheduled processor. As with other forms of multithreading, SMT uses thread-level parallelism to hide long-latency events in a processor, thereby increasing the usage of the functional units. The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

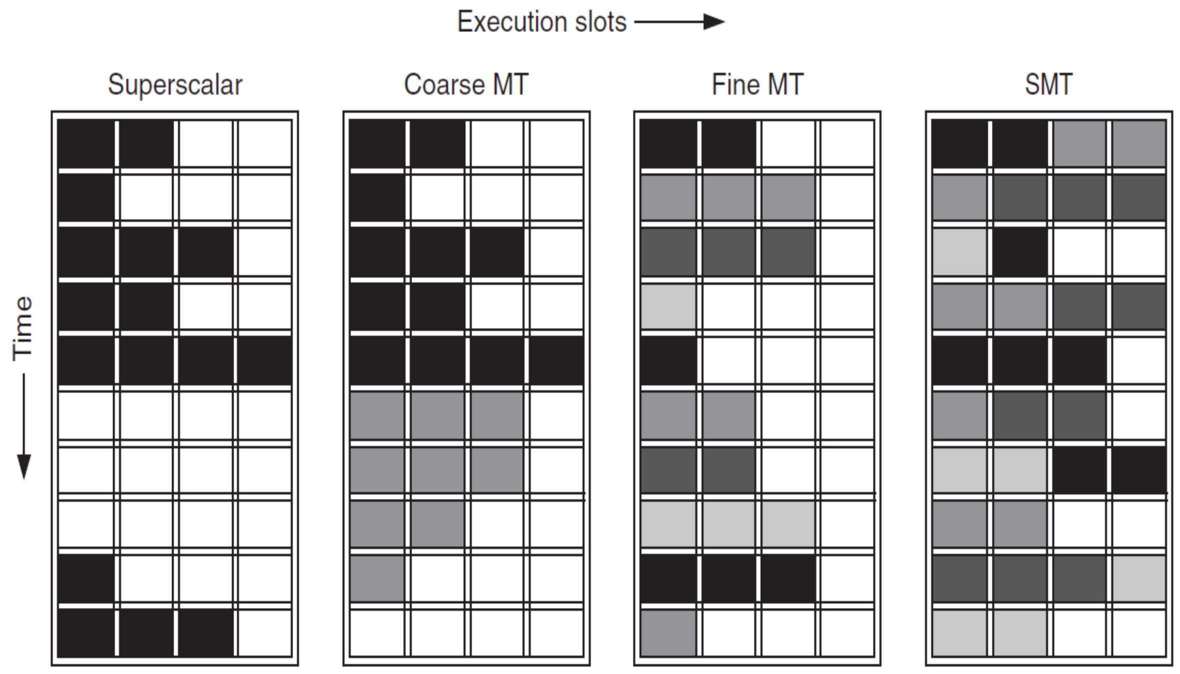


Figure 1.2: Multithreading techniques comparison

1.3 Objective

- To design a Fetch unit to support quad thread SMT.
- To design a Decode unit capable of decoding RISC V ISA instructions in processor clock cycle.
- To design a Issue unit capable of selecting instructions from a thread of highest IPC.

1.4 Overview of Content

Chapter 2 describes the SMT basics and compares different SMT configurations.Then an SMT selection metric is described which determines the optimal SMT configuration.Then a note on related work is provided and caparison between SMT and Multiprocessing is provided.Finally the chapter ends with description of the Bluspec System Verilog.

Chapter 3 starts with overview of the architecture with a diagrammatic description. Then instruction fetch basics are provided.Then a detailed description on the implementation is given. After that various fetch policies are presented which is followed by the IQ description.Next Decode stage is described and the chapter ends with state diagram of thread.

Chapter 4 describes the verification steps followed.It first describes a general verification model in BSV.Then specific test cases and various modifications to the test benches are described for every stage and a list of verified functionalities is given. And in the end the synthesis reports are provided both for FPGA and ASIC.

Chapter 5 provides the conclusion and a short note on future work

CHAPTER 2

BACKGROUND

2.1 Simultaneous Multithreading

The VLSI technologies of the past few years, while giving significant increases in transistor density, have not been able to deliver corresponding increases in transistor performance. One of the architectural techniques used in improving the overall performance of a wide range of applications has been Simultaneous Multithreading (SMT) [1]. It is designed to improve CPU utilization by exploiting both instruction-level parallelism and thread-level parallelism. The first extensive use of SMT in a commercial processor design was for Alpha 21464 (EV8), which was slated for 2004, but did not make it to the market. Intels first SMT-capable processor (marketed as Hyper-Threading) was the Foster-based Xeon in 2002, and in 2008 Intel reintroduced SMT with the Nehalem-based Core i7. IBM designed a fairly sophisticated SMT processor, POWER5 [2], by enabling dynamically managed levels of priority for hardware threads. The POWER5 processors were available in May 2004. Since then, IBM has developed the next two generations of POWER processors with further sophistications in resource allocations. In SMT, the processor handles a number of instruction streams (typically small number) from different threads in the same cycle. The execution context, like the program counter, is duplicated for each hardware thread, while most CPU resources, such as the execution units, the branch prediction resources, the instruction fetch and decode units and the cache, are shared competitively among hardware threads. In general the processor utilization increases because there are more instructions available to fill execution units and because instructions from other hardware threads can be executed while another instruction is stalled on a cache miss. Since the threads share some of the key resources, it is performance efficient to schedule threads with anti correlated resource requirements. From the software perspective, the resource allocation to a thread becomes the focal point for SMT specific performance improvement. Several studies have shown that SMT does not always improve the performance of applications [3], [4], [5]. The performance gains from SMT vary depending on a number of factors: The scalability of the workload, the CPU resources used by the workload, the instruction mix of the workload, the cache footprint of the workload, the degree of sharing among the software threads, etc. Fig.2.1 shows the performance of three benchmarks with and without SMT (4-way SMT) on the 8-core POWER7 microprocessor. We first run the application with eight threads at SMT1. Then we quadruple the number of threads and enable SMT4. Note that for Equake, SMT4 degraded the performance of the application, while it improved the performance of EP. MG's performance was oblivious to whatever SMT level was used.

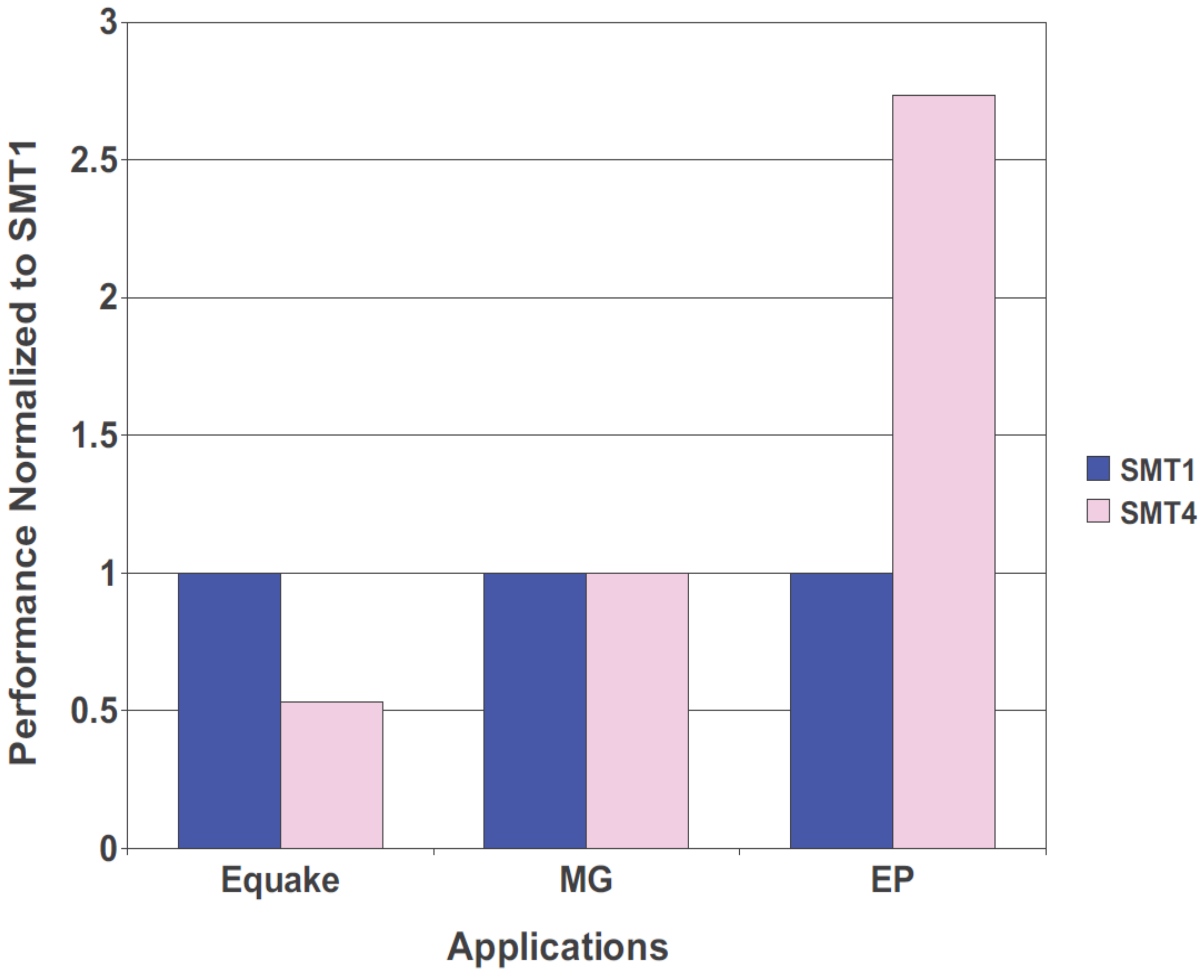


Figure 2.1: Multithreading techniques comparison[5]

2.2 The SMT Selection Metric

The rationale behind the SMT-selection metric is based on how well the instructions of a workload can utilize the various pipelines of a processor during each cycle. An ideal workload for SMT would have a good mix of instructions that are capable of filling all available functional units at each cycle. Figure 2.2 shows the pipeline of a generic processor core. In each cycle, the processor fetches from the instruction cache a fixed number of instructions. These instructions are then decoded and buffered. As resources become available, instructions can be dispatched/issued to the various execution/ functional units in an out-of-order manner. Issue ports are the pathways through which instructions are issued to the various functional units, which can operate independently. If the instructions that are issued consist of a mix of load, store, branch, integer, and floating point instructions and there are little data dependencies between them, then all functional units will be able to be used concurrently, hence increasing the utilization of the processor. We define the term ideal SMT instruction mix to mean a mix of instructions that is proportional to the number and types of the processors issue ports and functional units. With an ideal mix, the processor is able to execute the maximum number of instructions supported. In order for SMT to increase utilization there needs to be instructions available from all the hardware contexts to use as many issue ports as possible. Consider a multithreaded application whose vast majority of instructions are fixed point (integer) instructions. Running the application with more hardware contexts will not help because the fixed point units were already occupied most of the time with one hardware context. On the other hand, if we have an application with an ideal SMT instruction mix, then SMT should improve performance since the processor will have more opportunities to fill all the execution units. Since SMTsm must be able to predict whether an application benefits from additional SMT resources as we increase the number of threads, it must also include some measure of scalability within the application itself. After all, if there are software-related scalability bottlenecks, the application will not run better with increased number of threads irrespective of hardware. We observe that instruction mix, which is crucial for predicting hardware resource utilization in SMTsm, is also a good indicator of the applications scalability. An application that spends significant time spinning on locks will have a large percentage of branch instructions and a high deviation from the ideal SMT mix. Equation 1 shows how to calculate the SMTsm metric for the generic processor discussed above, where Pi denotes a unique issue port, N is the total number of issue ports, DispHeld is the fraction of cycles the dispatcher was held due to lack of resources, TotalTime is the wall-clock time elapsed, and AvgThrdTime is the average time spent by each hardware thread. Smaller metric values indicate greater preference for a higher SMT. The metric consists of three factors: i) the instruction mixs deviation from an ideal SMT instruction mix, ii) the fraction of cycles that the dispatcher was held due to lack of resources, and iii) the ratio of the wall- clock time elapsed to the average CPU time elapsed per thread. fPi is the fraction of instructions that are issued to Pi. For example, to calculate fP1, the number of instructions issued through port 1 is divided by the total number of instructions.

$$\text{SMT}_{sm} = \left(\sum_{u=0}^{N-1} (f_{pi} - 1/N)^2\right)^{1/2}$$

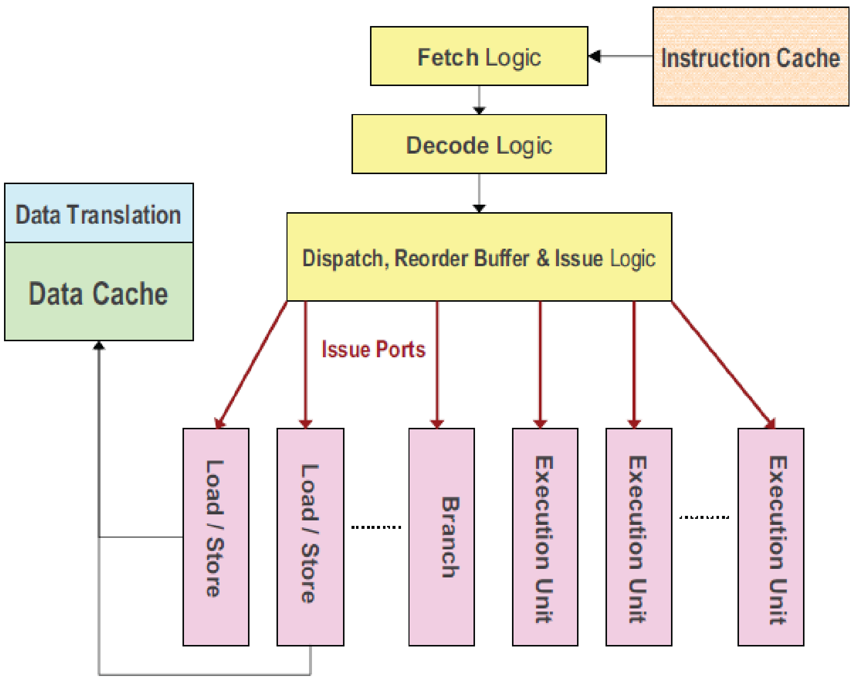


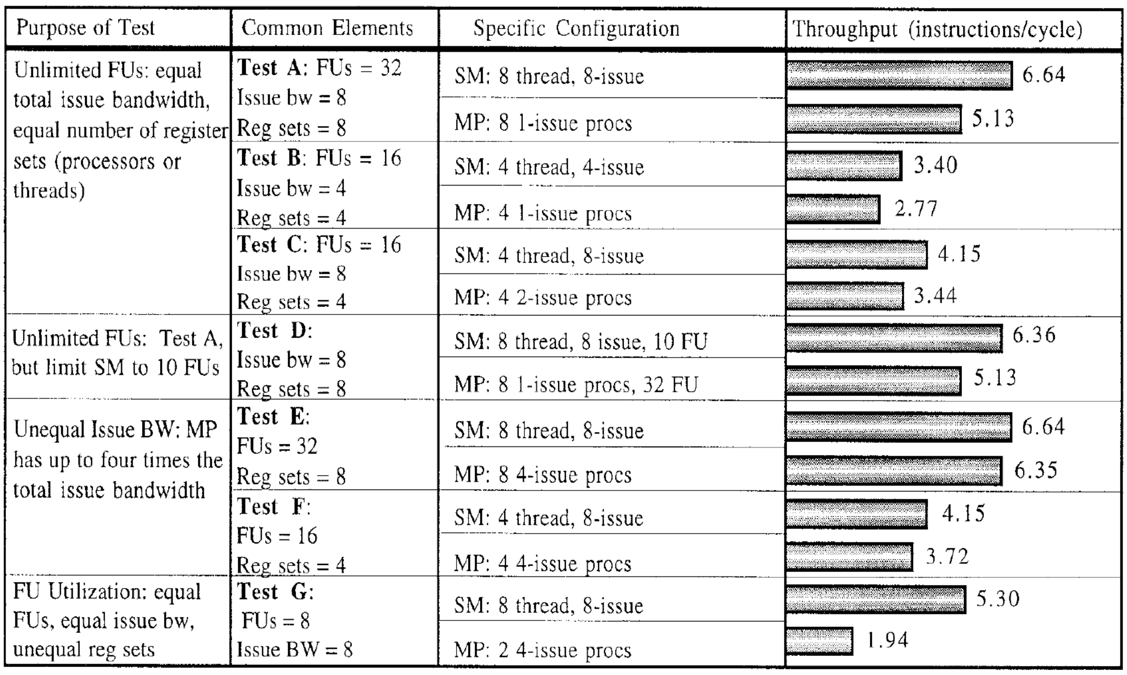
Figure 2.2: A generic processor execution engine

2.3 Related Work

Tullsen [2] run a variety of experiments for different Round-Robin (RR) configurations and found RR.2.8, which fetches up to eight instructions from every two threads in round-robin fashion, to be the best policy. Then they explored several fetch policies that assign fetch priority to threads according to various criteria, and compared performances of these policies to that of RR.2.8 scheme. Their results show that the best priority policy is ICOUNT, in which priority is assigned to a thread according to the number of instructions it has in decode, rename, and issue stages (issue queues) of the pipeline. Threads with the fewest such instructions are given the highest priority for fetch. The rationale is that such threads may be marking more forward process than others. It can prevent one thread from clogging the issue queue, and provide a mix of threads in the issue queue to increase parallelism. With their experiments, the ICOUNT.2.8, the same as RR.2.8 but favor the threads with two highest priorities, enhance system performance for 37 percent (IPC from 3.9 to 5.4). From then on, many researches improve on ICOUNT policy for better performance. Luo[3] proposed Unready Count Gating, Data Miss Gating and Predictive Data Miss Gating policies to reduce IQ clog. Limousin[4] introduced instruction prediction in their policy. If an instruction is predicted to bring L2 cache miss, its sequent instructions in the thread will be limited to be fetched. In the modified policy proposed by Tullsen [5], if a L2 cache miss is detected, all dependent instructions in front-end of the pipeline is flushed. Knijnenburg [6] classified branch instructions into easy-predict and hard-predict. Branch prediction is only conducted on instructions belong to easypredict. Thread owns hard-predicted instruction is stopped fetching. Ali[7] proposes several front-end policies thatreduce the required integer and floating point issue queue sizes in SMT processors. Their schemes are based on ICOUNT.2.8 policy, and provide some gating mechanisms on the basic policy according to the number of the instructions that are not ready in the issue queue, or the L1 cache miss rate of a given thread. Shin[8] proposed Adaptive Dynamic Thread Scheduling mechanism. He[9] proposed the IPC Based Fetch Policy. The basic idea of the policy is to select two threads with least instructions in the instruction queue and feed as many as the needed number of instructions to every selected thread, up to eight in total. Some other researches[10] focus on thread scheduling methods. In their methods, they take some factors into account when fetching instructions, such as response time, cache miss rate, number of IQ conflicts etc. Although these methods do help with their considered factors, overall performances of the methods in SMT processor never exceed the result of ICOUNT policy which takes the overall performance as its final designing objective.

2.4 Simultaneous Multithreading versus Single-Chip Multiprocessing

As chip densities continue to rise, single-chip multiprocessors will provide an obvious means of achieving parallelism with the available real estate. This section compares the performance of simultaneous multithreading to small-scale, single-chip multiprocessing (MP). On the organizational level, the two approaches are extremely similar: both have multiple register sets, multiple functional units, and high issue bandwidth 'on a single chip. The key difference is in the way those resources are partitioned and scheduled: the multiprocessor statically partitions resources, devoting a fixed number of functional units to each thread; the SM processor allows the partitioning to change every cycle. Clearly, scheduling is more complex for an SM processor; however, we will show that in other areas the SM model requires fewer resources, relative to multiprocessing, in order to achieve a desired level of performance. For these experiments, we tried to choose SM and MP configurations that are reasonably equivalent, although in several cases we biased in favor of the MP. For most of the comparisons we keep all or most of the following equal: the number of register sets (i.e, the number of threads for SM and the number of processors for MP), the total bandwidth, and the specific functional unit configuration. A consequence of the last item is that the functional unit configuration is often optimized for the multiprocessor and represents an inefficient configuration for simultaneous multithreading. All experiments use 8 KB private instruction and data caches (per thread for SM, per processor for MP), a 256 KB 4-way set-associative shared second-level cache, and a 2 MB direct-mapped third-level cache. We want to keep the caches constant in our comparisons, and this (private and D caches) is the most natural configuration for the multiprocessor. We evaluate MPs with 1, 2, and 4 issues per cycle on each processor. We evaluate SM processors with 4 and 8 issues per cycle; however we use the SM:Four Issue model for all of our SM measurements (i.e., each thread is limited to four issues per cycle). Using this model minimizes some of the inherent complexity differences between the SM and MP architectures. For example, an SM:Four Issue processor is similar to a single-threaded processor with 4 issues per cycle in terms of both the number of ports on each register file and the amount of inter-instruction dependence checking. In each experiment we run the same version of the benchmarks for both configurations (compiled for a 4-issue, 4 functional unit processor, which most closely matches the MP configuration) on both the MP and SM models; this typically favors the MP. We ,must note that, while in general we have tried to bias the tests in favor of the MP, the SM results may be optimistic in two respects- the amount of time required to schedule instructions onto functional units, and the shared cache access time. The impact of the former is small. The distance between the load store units and the data cache can have a large impact on cache access time. The multiprocessor, with private caches and private load store units, can minimize the distances between them. Our SM processor cannot do so, even with private caches, because the load store units are shared. However, two ultimate configurations could eliminate this difference. Having eight load/store units (one private unit per thread, associated with a private cache) would still allow LIS to match MP performance with fewer than half the total number of MP functional1 units (32 vs. 15). Or with 4 load store units and 8 threads, we could statically share a single cache i load store combination among each set of 2 threads. Threads 0 and 1 might share one load tore unit, and all accesses through that load/store unit would go to the same cache, thus allowing us to minimize the distance between cache and load store unit, while still allowing resource sharing. Figure 2.3 shows the results of our SM/MP comparison for various configurations. Tests A, B, and C compare the performance of the two schemes with an essentially unlimited number of functional units (FUs); i.e., there is a functional unit of each type available to every issue slot. The number of register sets and total issue bandwidth are constant for each experiment, e.g., in Test C, a 4 thread, 8-issue SM and a 4-processor, 2-issue-per-processor MP both have 4 register sets and issue up to 8 instructions per cycle. In these models, the ratio of functional units (and threads) to issue bandwidth is high, so both configurations should be able to utilize most of their issue bandwidth. Simultaneous multithreading, however, does so more effectively. Test D repeats test A but limits the SM processor to a more reasonable configuration (the same 10 functional unit configuration used throughout this paper). This configuration outperforms the multiprocessor by nearly as much as test A, even though the SM configuration has 22 fewer functional units and requires fewer forwarding connections. In tests E and F, the MP is allowed a much larger total issue bandwidth. In test E, each MP processor can issue 4 instructions per cycle for a total issue bandwidth of 32 across the 8 processors; each SM thread can also issue 4 instructions per cycle, but the 8 threads share only 8 issue slots. The results are similar despite the disparity in issue slots. In test F, the 4-thread, 8-issue SM slightly outperforms a 4-processor, 4-issue per processor MP, which has twice the total issue bandwidth. Simultaneous multithreading performs well in these tests, despite its handicap, because the MP is constrained with respect to which 4 instructions a single processor can issue in a single cycle. Test G shows the greater ability of SM to utilize a fixed number of functional units. Here both SM and MP have 8 functional units and 8 issues per cycle. However, while the SM is allowed to have 8 contexts (8 register sets), the MP is limited to two processors (2 register sets), because each processor must have at least 1 of each of the 4 functional unit types. Simultaneous multithreading's ability to drive up the utilization of a fixed number of functional units through the addition of thread contexts achieves more than 23 times the throughput. These comparisons show that simultaneous multithreading outperforms single-chip multiprocessing in a variety of configurations because of the dynamic partitioning of functional units. More important, SM requires many fewer resources (functional units and instruction issue slots) to achieve a given performance level. For example, a single thread, 8-issue SM processor with 10 functional units is 24 percent faster than the 8-processor, single-issue MP (Test D), which has identical issue bandwidth but requires 32 functional units; to equal the throughput of that 8-thread 8-issue SM, an MP system requires eight 4-issue processors (Test E), which consume 32 functional units and 32 issue slots per cycle. Finally, there are further advantages of SM over MP that are not shown by the experiments: Performance with few threads - These results show only the performance at maximum utilization. The advantage of SM (over MP) is greater as some of the contexts (processors) become unutilized. An idle processor leaves Up of an MP idle, while with SM, the other threads can expand to use the available resources. This is important when (1) we run parallel code where the degree of parallelism varies over time, (2) the performance of a small number of threads is important in the target environment, or (3) the workload is sized for the exact size of the machine (e.g., 8 threads). In the last case, a processor and all of its resources is lost when a thread experiences a latency orders of magnitude larger than what we have simulated (e.g., IO). o Granularity and flexibility of design - Our configuration options are much richer with SM, because the units of design have finer granularity. That is, with a multiprocessor, we would typically add computing in units of entire processors. With simultaneous multithreading, we can benefit from the addition of a single resource, such as a functional unit, a register context, or an instruction issue slot; furthermore, all threads would be able to share in using that resource. Our comparisons did not take advantage of this flexibility. Processor designers, taking full advantage of the configurability of simultaneous multithreading, should be able to construct configurations that even further out-distance multiprocessing. For these reasons, as well as the performance and complexity results shown, we believe that when component densities permit us to put multiple hardware contexts and wide issue bandwidth on a single chip, simultaneous multithreading represents the most efficient organization of those resources.



Purpose of Test	Common Elements	Specific Configuration	Throughput (instructions/cycle)
Unlimited FUs: equal total issue bandwidth, equal number of register sets (processors or threads)	**Test A**: FUs = 32 Issue bw = 8 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 1-issue procs	5.13
	Test B: FUs = 16 Issue bw = 4 Reg sets = 4	SM: 4 thread, 4-issue	3.40
		MP: 4 1-issue procs	2.77
	Test C: FUs = 16 Issue bw = 8 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 2-issue procs	3.44
Unlimited FUs: Test A, but limit SM to 10 FUs	**Test D**: Issue bw = 8 Reg sets = 8	SM: 8 thread, 8 issue, 10 FU	6.36
		MP: 8 1-issue procs, 32 FU	5.13
Unequal Issue BW: MP has up to four times the total issue bandwidth	**Test E**: FUs = 32 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 4-issue procs	6.35
	Test F: FUs = 16 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 4-issue procs	3.72
FU Utilization: equal FUs, equal issue bw, unequal reg sets	**Test G**: FUs = 8 Issue BW = 8	SM: 8 thread, 8-issue	5.30
		MP: 2 4-issue procs	1.94

Figure 2.3: SMT and MP Comparison[7]

2.5 Bluespec System Verilog

2.5.1 Building a design in BSV

Figure: 2.4 illustrates the various steps involved in building a design in Bluespec System Verilog.

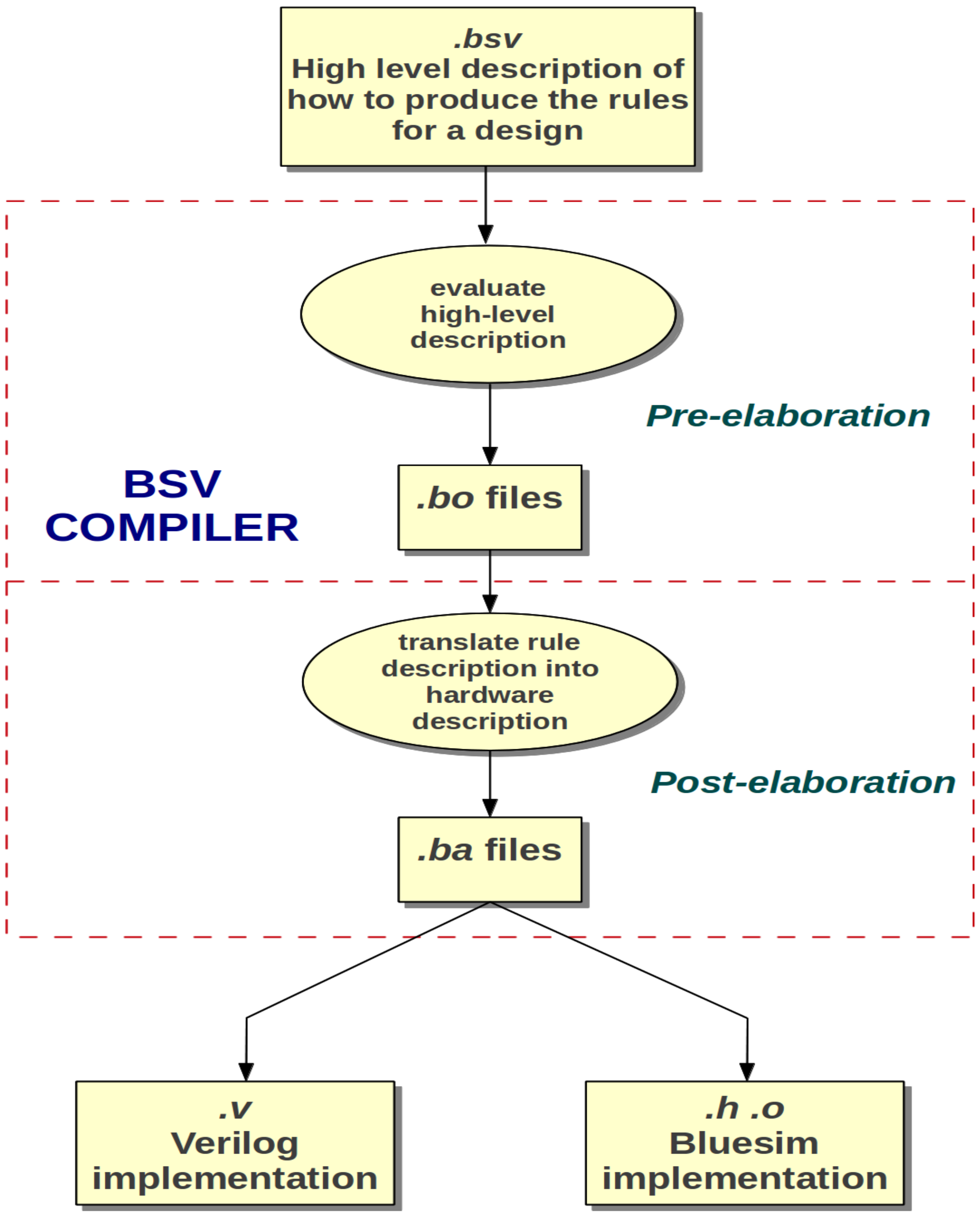

.bsv
High level description of how to produce the rules for a design
evaluate high-level description
Pre-elaboration
BSV COMPILER
.bo files
translate rule description into hardware description
Post-elaboration
.ba files
.v
Verilog implementation
.h .o
Bluesim implementation

Figure 2.4: Building a design in BSV

- Designer writes a BSV program. It may optionally include Verilog, SystemVerilog, VHDL,and C components.
- The BSV program is compiled into a Verilog or Bluesim specification. This step has two distinct stages:
    1. pre-elaboration - parsing and type checking
    2. post-elaboration - code generation
- The compilation output is either linked into a simulation environment or processed by a synthesis tool.

2.5.2 Rules

BSV does not have always blocks like Verilog. Instead, rules are used to describe all behavior (how state evolves over time). Rules are made up of two components:

Rule Condition: a Boolean expression which determines if the rule body is allowed to execute(*"fire"*).

Rule Body: a set of actions which describe the state updates that occur when the rule fires.

We can logically think of a rule's execution as ***instantaneous, complete and ordered*** w.r.t execution of all other rules.

Instantaneous:

- Conceptually, all the actions in the rule body occur at a single,common instant - there is no sequencing of actions within a rule.

Complete:

- When fired, the entire rule body executes. There is no concept of "partial" execution of a rule body.

Ordered:

- Each rule execution conceptually occurs either before or after every other rule execution, but never simultaneously.

Some constraints are imposed on rules. These constraints are:

- Each rule fires at most once within a clock.
- Certain pairs of rules, which we will call conflicting, cannot both fire in the same clock.

2.5.3 Module hierarchy and Interfaces

In BSV, a module's interface is an abstraction of its verilog port list. In BSV, the interface declaration and module declaration are separate. So, a common interface can be used by several modules, with out having to repeat the declaration in each of its implementation modules.

An interface declaration specifies the methods provided by every module that provides the interface,but does not specify the methods implementation. The implementation of the interface methods can be different in each module that provides that interface. The definition of the interface and its methods is contained in the providing module.
BSV classifies interface methods into three types:

- ***Value Methods***: These are methods which return a value to the caller, and have no"*actions*"(i.e., when these methods are called, there is no change of state, no side-effect).
- ***Action Methods***: These are methods which cause actions (state changes) to occur. One may consider these as input methods, since they typically take data into the module.
- ***Action Value Methods***: These methods couple Action and Value methods, causing an action (state change) to occur and they return a value to the caller.

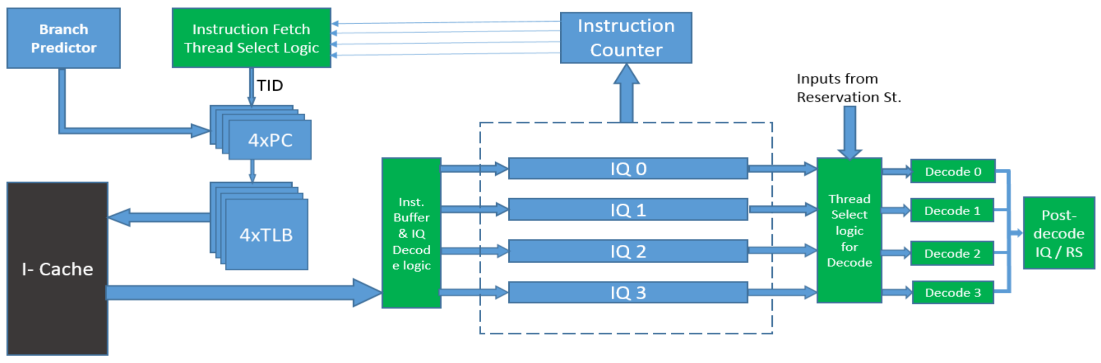
Every module uses the interface(s) just below it in the hierarchy. Every module provides an interface to the module above it in the hierarchy.

CHAPTER 3

ARCHITECTURE DESCRIPTION

3.1 Architecture Overview

Following figure 3.1 shows the block diagram for Fetch and Decode mechanisms:



Branch Predictor
Instruction Fetch Thread Select Logic
Instruction Counter
TID
4xPC
4xTLB
I- Cache
Inst. Buffer & IQ Decode logic
IQ 0
IQ 1
IQ 2
IQ 3
Inputs from Reservation St.
Thread Select logic for Decode
Decode 0
Decode 1
Decode 2
Decode 3
Post-decode IQ / RS

Figure 3.1: Microarchitecture Overview

3.2 Instruction Fetch

When designing a high-performance processor, it is important to keep all parts of the processor balanced, avoiding bottlenecks whenever possible. For example, as shown in Fig.3.2 , if we design a high-performance processor capable of executing five ALU operations at once, it is also important to ensure that we can feed the ALU stage and retire those instructions without stalling the pipeline. This means fetching and decoding at least five instructions per cycle, to keep the ALU stage busy, and writing results and graduating instructions at a fast enough rate.

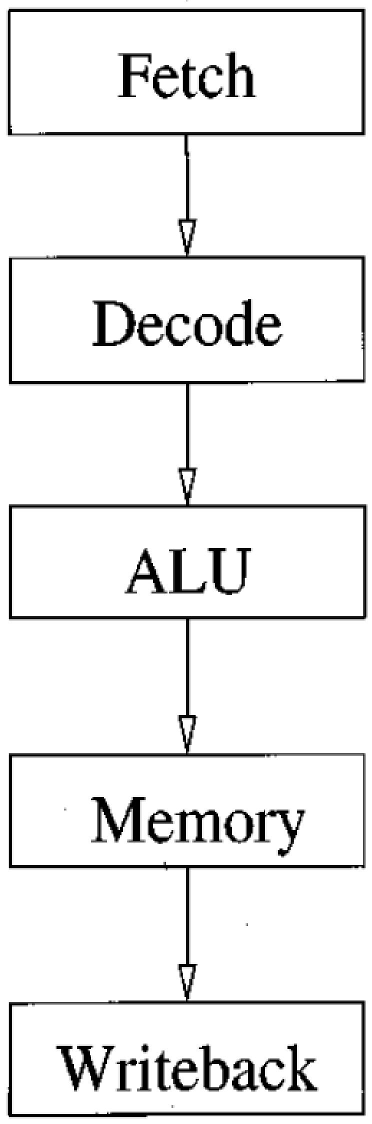


Figure 3.2: Basic five stage pipeline

But the fetch stage does not behave like other pipeline stages in the sense that it can not be widened by simply replicating it, or adding more functional units. It has to follow the control path defined by branch instructions, which have not been executed yet. The fetch stage first evolved to include branch prediction, and used it to fetch instructions from speculative execution paths. This ability to follow speculative paths independently of the execution stages leads to a decoupled view of the processor, as shown in Fig. 3.3. The fetch engine reads instructions from memory and places them in an instruction buffer.

Then,an execution engine reads instructions from the buffer and generates the required results, providing feedback to the fetch.

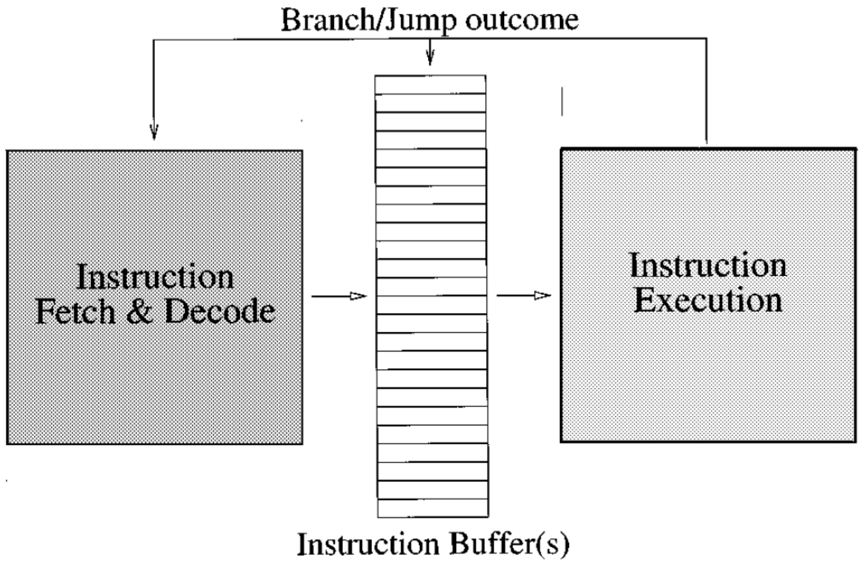


Figure 3.3: Decoupled view of the processor: a fetch engine produces instructions, and an execution engine consumes them

Superscalar processors attempt to obtain higher execution performance by exploiting instruction level parallelism (ILP), which basically means replicating the different pipeline stages to execute several instructions in parallel. In order to execute multiple instructions per cycle, it becomes imperative to fetch multiple instructions per cycle. But simply duplicating the number of functional units in the fetch stage does not resolve the problem. The fetch engine needs to increase its capabilities and read a full basic block of instructions from memory in a single cycle. With superscalar processors, a single-cycle delay represents an undetermined number of wasted instructions, which increases the importance of an accurate branch prediction mechanism. The performance metric changes from branch execution cost to branch execution penalty, which measures the number of wasted cycles due to a branch instruction. Given that the number of instructions executed per cycle varies depending on the available ILP, the number of lost cycles can not be easily translated to the number of lost instructions.

3.3 Detailed Description

The entire mechanism is built to support 4 thread SMT and hence we can see the 4 PCs being maintained separately for every thread. As can be seen from the figure3.1 we decided to keep the I-Cache to be physically addressed and hence the address is transformed from virtual to physical before being sent to the I-Cache. The address translation takes place with the help of TLB and to improve performance we decided to keep the TLBs as well separate for every thread as can be seen from the figure3.1. The address translation takes place as follows:

1. Let a be the value of the ptbr register, and let i = LEVELS-1.(LEVELS equals 2 for RV32Sv32 or 3 for RV64Sv43.)

2. Let pte be the value of the PTE at address a + va.vpn[i] x PTESIZE. (PTESIZE equals 4 for RV32Sv32 or 8 for RV64Sv43.)

3. If pte.v=0, signal an address error.

4. If pte.t=00, determine if the requested memory access is allowed by the permission bits. If not, signal an address error. Otherwise, the translation is successful and the physical address is given as follows:

- pa.pgoff=va.pgoff.
- If i is greater than 0, then this is a superpage translation and pa.ppn[i-1:0]=va.vpn[i-1:0].
- pa.ppn[LEVELS-1:i]=pte.ppn[LEVELS-1:i]

5. If i=0, signal an address error. Otherwise, let i=i-1, let a=pte.ppn x PAGESIZE, and go to step 2. (PAGESIZE equals 2 $^{12}forRV32Sv32or2^{12}forRV64Sv43.)$

3.4 Instruction Fetch Thread Select Logic

The Instruction Fetch Thread Select Logic (IFTSL) determines the next thread to be selected for fetch using the implemented logic. Two techniques have been implemented to do that which are:

3.4.1 Round Robin

In order to schedule processes fairly, a round-robin scheduler generally employs time-sharing, giving control to each thread in a circular fashion on every clock cycle. In case the thread is not ready because of long latency event like cache miss in that case the particular thread will be skipped and the next available ready thread will be selected.

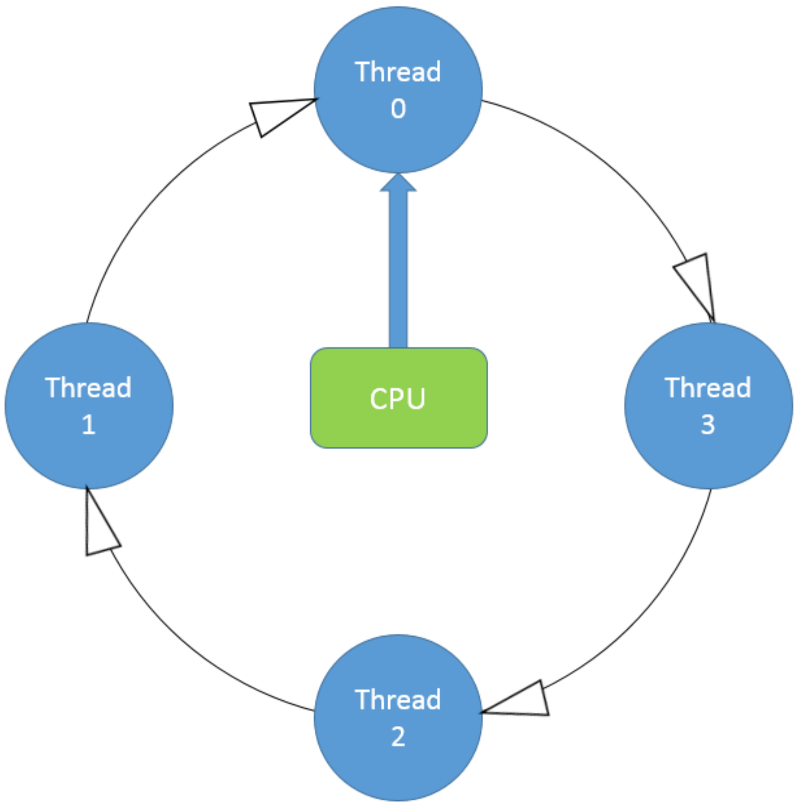


Figure 3.4: Round Robin

3.4.2 ICOUNT

In this policy we keep track of number of instructions in the instruction queue of every thread and hence the name Instruction COUNT. The thread with least number of instructions in the instruction queue is selected for fetch. The rationale behind this is if a thread has lower data dependency as well as structural dependency then the instructions from that thread will be issued more easily in other words the instruction flow rate of that thread will be higher compared to other threads which will result in less number of instructions in the instruction queue for that thread. This achieves three purposes:

- it prevents any one thread from filling the IQ
- it gives highest priority to threads that are moving instructions through the IQ most efficiently
- it provides a more even mix of instructions from the available threads, maximizing the parallelism in the queues

Following figure3.5 shows the hardware required to implement this policy:

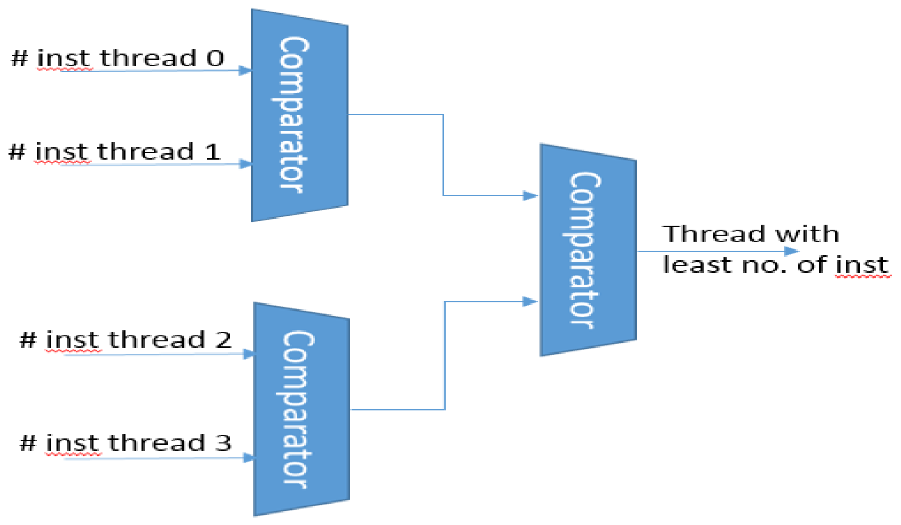


Figure 3.5: ICOUNT

Selection of one of these policies will depend upon area of application. For example where performance is not much important there we can use Round-Robin policy which is much simpler than ICOUNT and also chip area required for Round-Robin will be less. But where performance is most important there its better to use the ICOUNT policy because it gives better results in terms of performance. Like Round-Robin and ICOUNT some other polies are:

3.4.3 BRCOUNT

Here we attempt to give highest priority to those threads that arc least likely to be on a wrong path. We do this by counting branch instructions that are in the decode stage, the rename stage, and the instruction queues, favoring those with the fewest unresolved branches.

3.4.4 MISS COUNT

This policy detects an important cause of IQ clog. A long memory latency can cause dependent instructions to back up in the IQ waiting for the load to complete, eventually filling the queue with blocked instructions from onc thread. This policy prevents that by giving priority to those threads that have the fewest outstanding D cache misses.

3.4.5 IQPOSN

Like ICOUNT, IQPOSN strives to minimize IQ clog and bias toward efficient threads. It gives lowest priority to those threads with instructions closest to the head of either the integer or floating point instruction queues (the oldest instruction is at the head of the queue). Threads with the oldest instructions will be most prone to IQ clog, and those making the best progress will have instructions farthest from the head of the queue. This policy does not require a counter for each thread, as do the previous three policies.

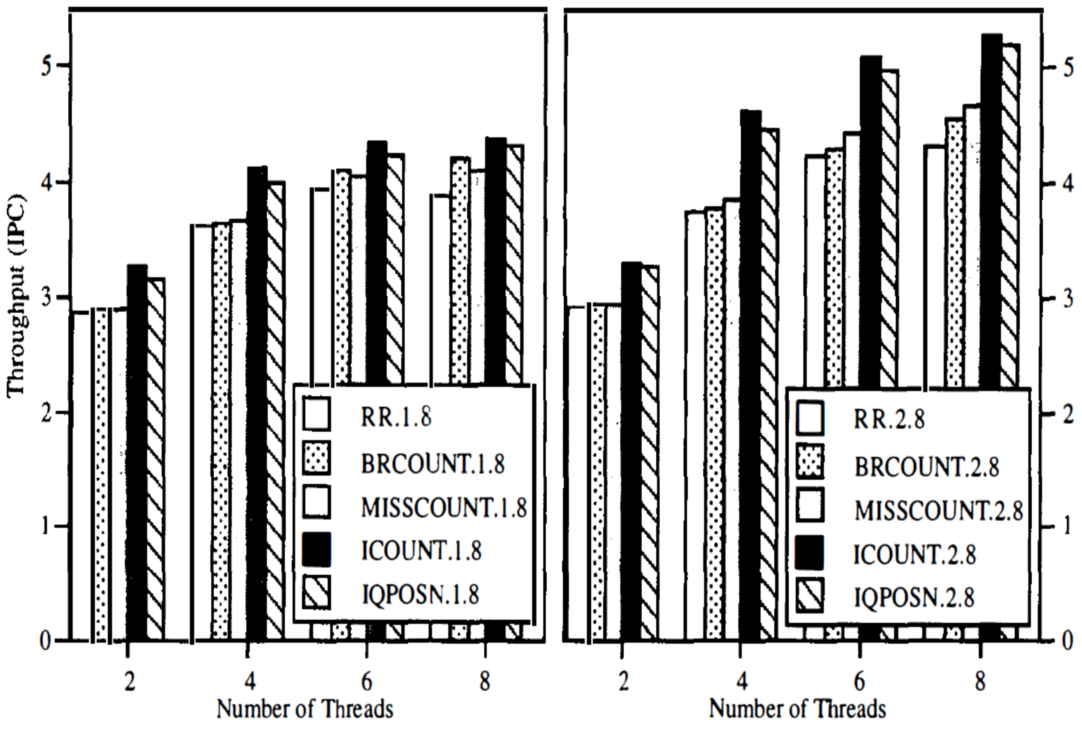


Figure 3.6: Comparison between various multithreading policies[2]

3.5 Instruction Queue

IQ is used to store instructions before they are issued to the functional units. Traditionally this IQ is kept as single register array or sometimes it is divided into two, one for integer instructions and another for FP instructions. But problem with these techniques is that they lead to IQ clog conditions. IQ clog occurs when the head of the IQ contains instructions from those threads which are having dependencies due to which the instructions for that thread cannot be issued and which makes instructions from other threads also to wait until that particular threads dependencies are cleared. One more problem with these kind of techniques is that when branch miss prediction occurs all the instructions from all the threads need to be flushed. Solution to all these problems is having separate instruction queues for separate threads as can be seen in figure3.1 because of which the threads will be able to flow through the pipeline independent of other threads. Following figure3.7 gives details about the Instruction Queue:

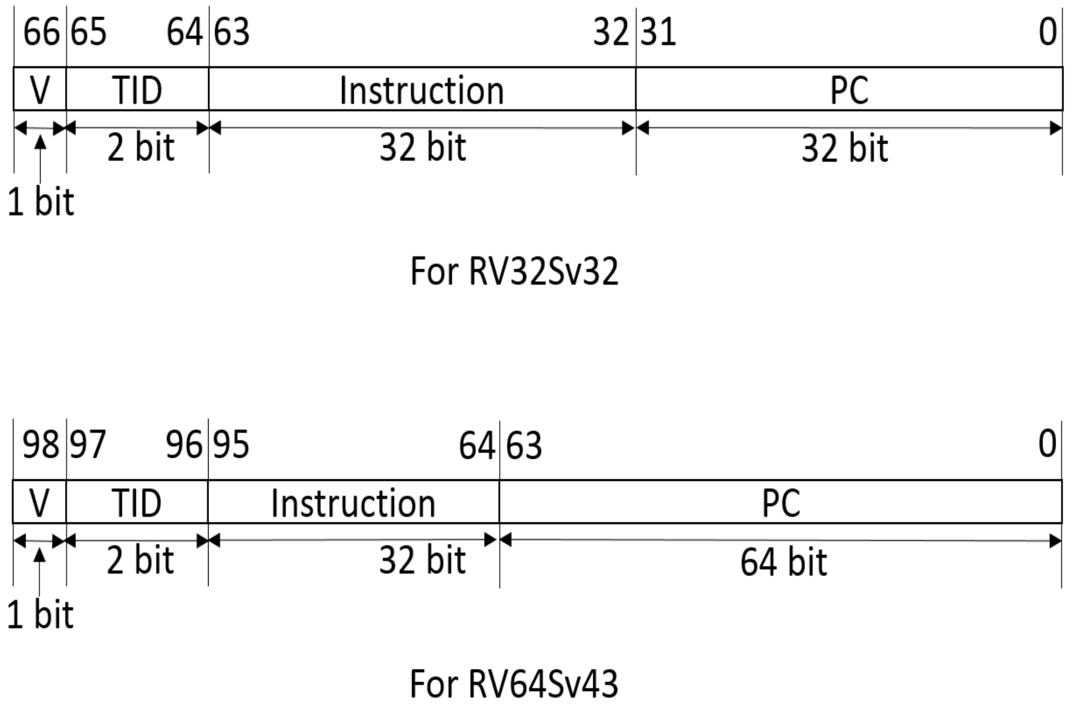


Figure 3.7: IQ Entry Details

As in the figure3.7 in both the architectures instruction queue entry contains four fields which are

- **V**: This v stands for valid bit. This bit is used to check whether the data in a particular entry is valid or not.
  - 0 = Data is invalid
  - 1 = Data is valid
- **Thread ID (TID)**: This field indicates which hardware thread does this instruction belong to. As shown in figure3.7 this field is two bits because two bits are sufficient to represent 4 threads.
- **Instruction**: This field is 32 bits wide and used to store the instruction received from cache.

- **Program Counter (PC)**: This field unlike other fields has different no. of bits for different architectures. For 32 bit version this field is 32 bit wide which contains the virtual address of the corresponding instruction. For 64 bit version this field is 64 bit wide which contains the virtual address of the corresponding instruction.

3.6 Thread Select Logic for Decode (TSLD)

Similar to IFTSL any of the thread selection policies mentioned above can be used for this purpose. But the difference here is that instead of number of instructions in the instruction queues, the number of instructions in the reservation station are taken into account. The main reason for separately choosing thread to fetch and thread to issue is that if we choose thread only at the time of fetch then by the time instructions reach functional units that is after 3-4 cycles they might not be use full enough. So by separating the thread selection logic for issue this delay is effectively reduced to 1-2 cycles. By this means more use full instructions are provided to reservation station every clock cycle.

3.7 Decode

Processors make use of instruction pipelining to speed up execution. In essence, pipelining involves breaking up the instruction cycle into a number of separate stages that occur in sequence, such as fetch instruction, decode instruction, determine operand addresses, fetch operands, execute instruction, and write operand result. Instructions move through these stages, as on an assembly line, so that in principle, each stage can be working on a different instruction at the same time.The occurrence of branches and dependencies between instructions complicates the design and use of pipelines.

Many times the decode forms the critical path in a processor pipeline. So to avoid that partial decoding of instruction is done which not only takes lesser delay but also eases the issue process. Before going into complete details it is advised to go through appendix A which provides a brief introduction to RISC-V ISA.As shown in figure3.1 , 4 decodes are used so as to decode 4 instructions simultaneously in order to maintain a high instruction flow rate through the pipeline. The decoder is based on the fact that there will be only 6 types of different execution units :

- ALU
- MEM (LOAD-STORE)
- BRANCH
- SPFPU
- DPFPU
- SUPERVISOR

Following figure3.8 shows the decoder as a black box and input and outputs:

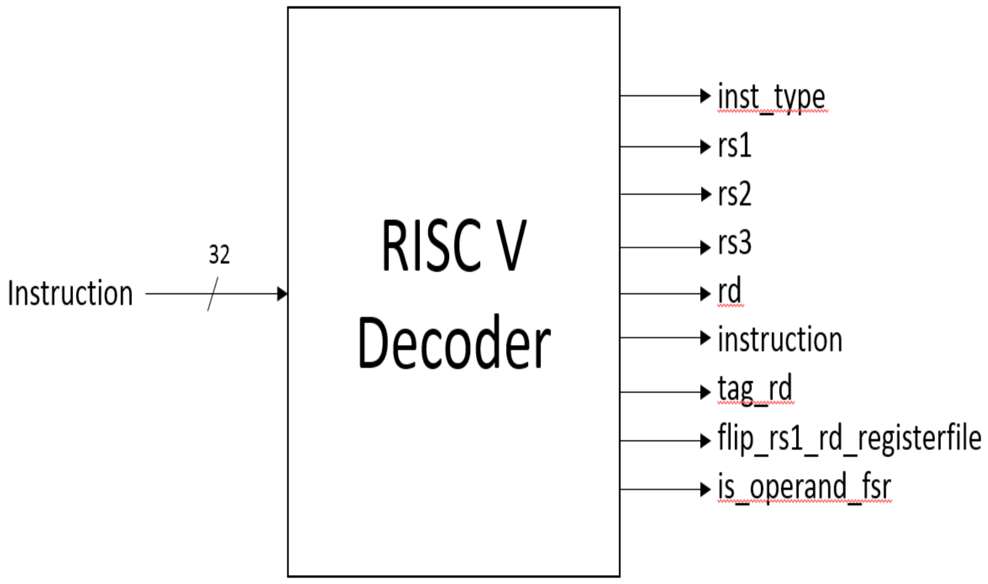


Figure 3.8: RISC V ISA Decoder

As can be seen from the figure the decoder is just a combinational block which analyzes the instruction and gives information about the processing and path of the instruction in terms of following fields:

inst type : This field is used to hold the instruction type. That is this field can have any value between ALU, MEM, BRANCH, SPFPU, DPFPU, NOP, SUPERVISOR, AMO which defines the functional unit required by the instruction and its flow through the pipeline.

rs1 : This field is used to store the address of the source operand 1 which is a 5 bit field as the number of registers supported by RISC V are 32.

rs2 : Similar to rs1 this field is used to store the address of the source oprand 2.

rs3 : Similar to rs1 and rs2 this field is used to store the address of the source oprand 3.

rd : Similar to rs1,rs2 and rs3 this field is used to store the address of the destination oprand.

instruction : As discussed earlier the decode unit decodes the instruction partially just to ease the issue stage logic. Because of this some information is retrieved from the instruction in the execution stage so its important to forward instruction through the pipeline that is what this field is used for.

tag rd : Sporadically it is important to tag the destination register and this field is used to answer the question do we need to tag the destination register while issue?

- 0= tag is not needed
- 1= tag is needed

flip rs1 rd registerfile : This 2 bit field is used to ease the fetch logic. These two bits give information about which register file is to be accessed on receiving the register address. If bit0 is 1 flip the register file of the rd than its conventional one. if bit1 is 1 then flip the register file for rs1 than its conventional one. Conventional ones are defined based on the type of instruction - integer or floating.

is operand fsr : Some instructions operate on fsr in that case it is easier to handle those instructions with help of this single bit field which is when 1 means operand is fsr for FSSR, FRSR operations. 0 means no fsr required. This will make the operand2 as FSR.

3.8 Thread State Diagram

A thread can be in any one of the four defined states at a given time. The 4 states are:

1. Available State: This state implies that curently the hardware thread is empty and any software thread can be assigned to it.

2. Wait State: This state implies that the hardware thread is assigned to a software thread (and it can not be assigned to a new software thread until current one finishes), but it is waiting for some event like completion of previous instruction for the operands or cache or functional unit. Importance of this state is that it tries to avoid the IQ full condition by not fetching the instructions for the threads which are in wait state.

3. Ready state: The thread transits from wait state to the ready state when the event for which the thread was waiting occurs. In this state also the instructions are not being fetched for the thread. The ready state signifies that the instructions for that thread can be issued to next blocks.

4. Running State: This state indicates that the thread instruction issue is not stalled and the instructions for that thread can be fetched on next cycle.

So, in this we can see that instructions can be fetched only for the threads which are in running state. Now at a given time multiple threads can be in running state and the cache bandwidth is limited to 2 threads per cycle. Which demands for a thread selection logic and for this particular implementation, RR2.8 is used as stated earlier. According to it the two least recently used threads are selected every cycle from the running threads and instructions are fetched for them. State transition diagram for this can be drawn as:

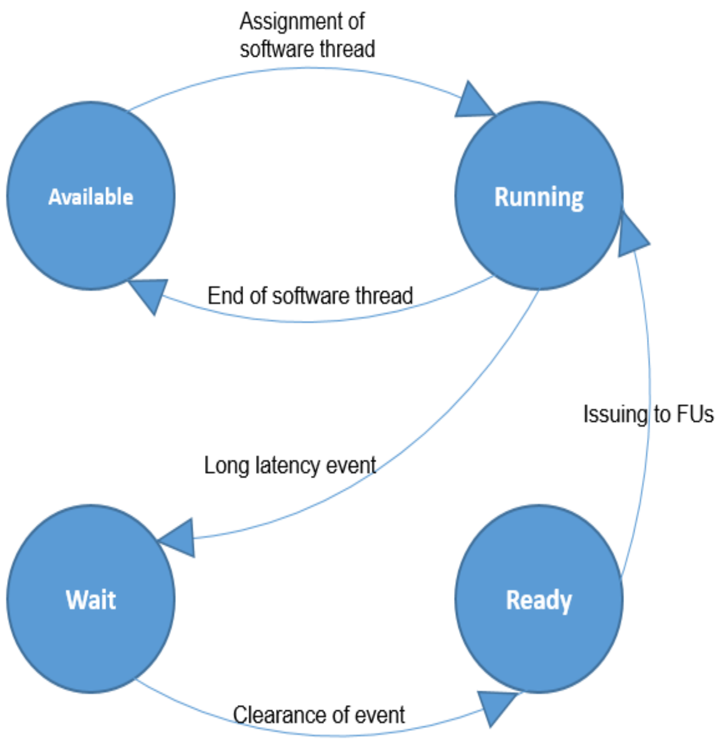


Figure 3.9: Thread State Diagram

To store the state of a thread the TCB reserves 2 bits per thread. But for readability an enum data type is declared named Thread state type.In TCB the entity which holds the current state of the thread is declared as rg thread state. To modify the state of a thread seperate method is given for every thread named thread0 state update. The 0 in the method name can be replaced by any number between 1 to 7 to update state for any thread.

CHAPTER 4

VERIFICATION and RESULTS

This chapter deals about how the processor design is verified in Bluespec System Verilog. It also describes about the test cases generated to verify the design.

4.1 Verification Setup

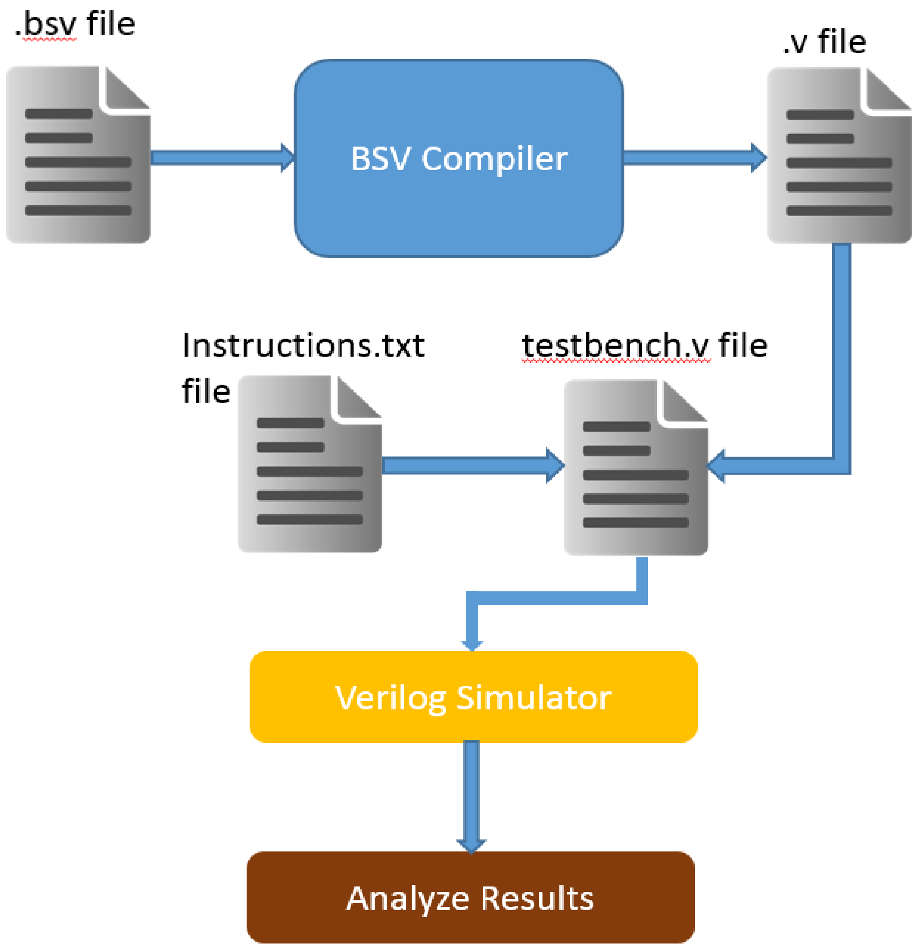


Figure 4.1: Verification flow

The Bluespec code that is written in .bsv format is given to the Bluespec compiler in the Bluespec Development Workstation (BSW). The Bluespec Compiler compiles and generates the Verilog code in .v format. The Verilog code is simulated in the Xilinx ISE Verilog simulator to check for the functionality of the design. The Verilog code is also synthesized in Xilinx ISE to get the clock frequency and to know about the amount of hardware generated by the design. Figure4.1 shows the verification flow in Bluespec System Verilog.

The functionality of the design is verified by using a test bench which provides a set of instructions given to the main module. display statements were written in the code in all the necessary places to monitor the functionality of the modules in various clock cycles.

4.2 Verification Strategy

The verification process mainly focused on verifying the functional correctness of the developed stages.There were two approaches to verify the developed stages.One of which was to verify all the stages together as a single unit and another one was to verify every stage independently.I chose to go with the later one because if we chose to verify all the stages as a single unit it would have been easy to check functionalities which are limited to single cycle behaviour like output of decode stage or switching the threads on given contitions. But verifying multicycle behaviour would have been very difficult which will get cleared from the following stage wise strategies.

4.2.1 Fetch Stage

During fetch stage it is very important to check whether the instructions are received and stored in the instruction queues in the correct sequence or not.Also it is very important that the instructions from a particular thread to go into the instruction queue allotted for that thread. To verify that I created special instructions which contained information about both the address of the instruction and thread of the instruction which resulted in easier and less prone to error analysis.The functionalities checked during the verification of fetch stage are:

- Filling of Instruction Queues.
- Correct sequence of instructions.
- Accurate address translation.
- Freeing instruction queues from unnecessary instructions.
- Simultaneous behaviour of multiple instruction queues.
- Instruction Fetch Thread Select Logic.
- Instruction address increment or decrement.

4.2.2 Decode stage

Though decode logic looks simple verifying it is definitely not an easy task.Because with over 150 instructions and that to combined with fields like source operand destination, operand immediate data, etc. the possible combinations may go beyond thousands. But here also dividing the verification task into groups and choosing appropriate instructions can give the required confidence. The functionalities checked during the verification of fetch stage are:

- Instruction decoding to correct functional unit.
- Decoding of operands.
- Handling of immediate data.
- Special fields related to register files and fsr.

4.2.3 Issue Stage

Here in this issue is just about selecting the thread for issuing and filling the corresponding instructions into the decode stage which later fills the post decode IQ.Now nuisance here is thread selection depends on the the number of instructions in the reservation station which is very difficult to predict because is mostly depends on the application software. Still assuming a mix of all kinds of conditions following functionalities are verified.

- Thread Select Logic for Decode.
- Filling up of post decode IQ.
- Retrieving instructions from IQs.
- Shifting of IQ.

Files associated with the design:

Top Module:

fetch.bsv

Other modules:

riscv decoder.bsv

iqbuffer.bsv

4.3 Synthesis Report

The design has been synthesized on both FPGA and ASIC platforms. For FPGA synthesis Xilinx ISE was used to synthesize on ***Virtex 6 XC6VLX240T-FF1156*** kit and for ASIC synthesis synopsys was used with library **uk65lscllmvbbl 110c0 bc**. Both the synthesis reports are given in following sections.

4.3.1 FPGA Synthesis Report

The slice utilization and timing summary are provided below.

Device Utilization Summary :

Selected Device: 6vlx240tff1156-1

Slice Logic Utilization:

Number of Slice Registers : 3582 out of 301440

Number of Slice LUTs : 11887 out of 150720

Number used as Logic : 11887 out of 150720

Number of fully used LUT-FF pairs: 3546 out of 11923

Timing Summary :

Minimum period: 4.015ns

Maximum Frequency: 249.054MHz

Minimum input arrival time before clock: 0.698ns

Maximum output required time after clock: 1.123ns

4.3.2 ASIC Synthesis Report

Technology used: **Synopsys 65 nm**

Library used: **uk65lscllmvbbl 110c0 bc**

Area Utilization Summary :

Number of ports: 583

Number of nets: 35140

Number of cells: 35043

Number of combinational cells: 28879

Number of sequential cells: 6164

Number of buf/inv: 2975

Number of references: 281

Combinational area: 67679.640840

Buf/Inv area: 5712.480133

Noncombinational area: 44869.679406

Total cell area: 112549.320246

Timing Summary :

Minimum period: 600ps

Maximum Frequency: 1.66 GHz

Power Consumption:

Cell Internal Power: 77.0066 mW

Net Switching Power: 2.6685 mW

Total Dynamic Power: 79.6751 mW

Cell Leakage Power: 192.5592 uW

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In SMT processors, as the number of competing threads is increasing, instruction throughput and performance are largely impacted by fetch policy. With high issue bandwidth of the designed processor it is very important to feed the pipeline with useful instructions every cycle.To do that the thread selection logic is implemented in two stages which are thread select logic for fetch and thread select logic for issue.Separate instruction queues are designed for every hardware thread which results in independent flow of the thread through the pipeline.A special single thread execution mode is also designed in which the processor will support a single hardware thread.This mode is designed so that the multitheading overhead will not affect single thread performance.The decode unit is designed in such a way that it decodes the instruction partially but still providing all the information needed till it reaches execution unit, which results in optimum decode delay.

Using above mentioned techniques Fetch, Decode and Issue units are developed for a quad thead SMT processor in two variants 32 bit and 64 bit.The design is synthesized on both FPGA and ASIC platforms.On FPGA the maximum frequency achieved is 249 MHz and on ASIC its 1.66GHz.

5.2 Future Work:

The existing design was developed for providing maximum performance and speed. Which resulted in a higher area requirement and power budget.To enhance the processor further following steps can be taken

- Optimizing the area of the design.
- Optimizing the power consumption.

APPENDIX A

RISC-V ISA FEATURES

A.1 RISC-V ISA Overview

The RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISA is very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. The base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional supervisor-level operations), and so provides a convenient ISA and software toolchain around which more customized processor ISAs can be built. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are two base integer variants, RV32I and RV64I, described in Chapters 2 and 3, which provide 32-bit or 64-bit user-level address spaces respectively. Hardware implementations and operating systems might provide only one or both of RV32I and RV64I for user programs.

A.2 Instruction Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction parcels in length and parcels are naturally aligned on 16-bit boundaries. The standard compressed ISA extension described in Chapter 8 reduces code size by providing compressed 16-bit instructions and relaxes the alignment constraints to allow all instructions (16 bit and 32 bit) to be aligned on any 16-bit boundary to improve code density. Figure A.1illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to 11. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to 00, 01, or 10. Standard instruction- set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in FigureA.1 Instruction lengths between 80 bits The base RISC-V ISA has a little-endian

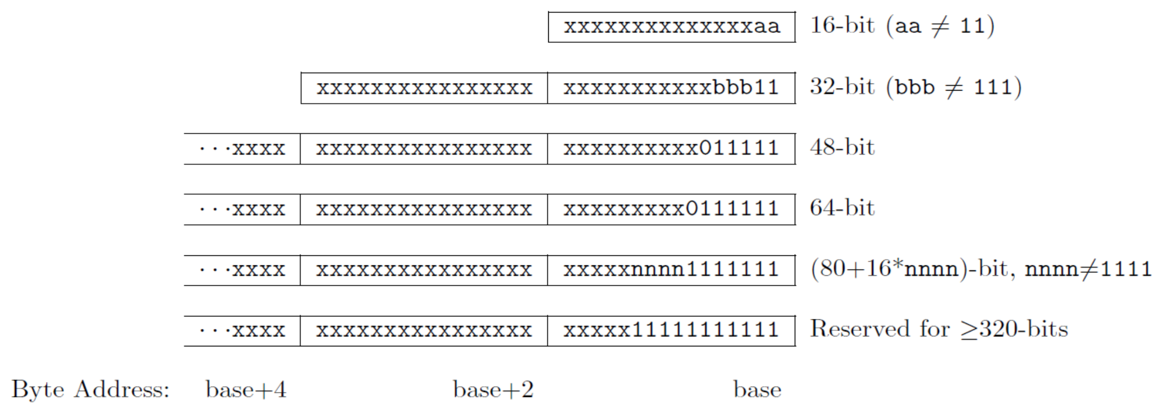


Figure A.1: Instruction length encoding

memory system, but non-standard variants can provide a big-endian or bi-endian memory system. Instructions are stored in memory with each 16-bit parcel stored in a memory halfword according to the implementation's natural endianness. Parcels comprising one instruction are stored at increasing halfword addresses, with the lowest addressed parcel holding the lowest numbered bits in the instruction speciation, i.e., instructions are always stored in a little-endian sequence of parcels regardless of the memory system endianess.

A.3 Programmers' Model for Base Integer Subset

Figure A.2 shows the user-visible state for the base integer subset. There are 31 general-purpose registers x1x31, which hold integer values. Register x0 is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register x1 to hold the return address on a call. For RV32, the x registers are 32 bits wide, and for RV64, they are 64 bits wide. This document uses the term XLEN to refer to the current width of an x register in bits (either 32 or 64). There is one additional user-visible register: the program counter pc holds the address of the current instruction.

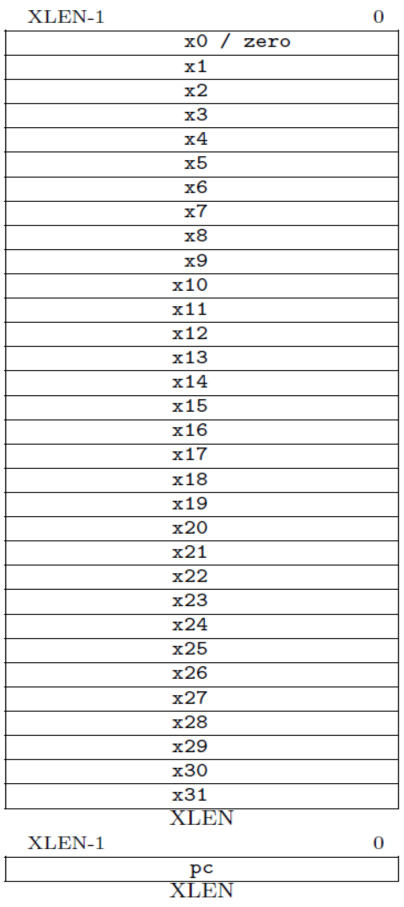


Figure A.2: Programmers' Model for Base Integer Subset

A.4 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in FigureA.3. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated if the PC is not four-byte aligned on an instruction fetch.

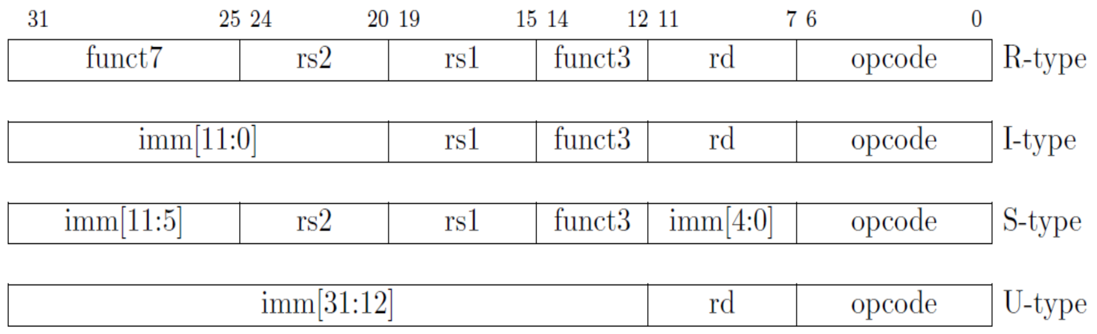


Figure A.3: Base Instruction Formats

A.5 Opcode Map

FigureA.4 shows a map of the major opcodes for RVG. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. Opcodes marked as reserved should be avoided for custom instruction set extensions as they might be used by future standard extensions. Major opcodes marked as custom-0 and custom-1 will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked custom-2/rv128 and custom-3/rv128 are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Figure A.4: Opcode Map

APPENDIX B

TRANSLATIONAL LOOKASIDE BUFFER

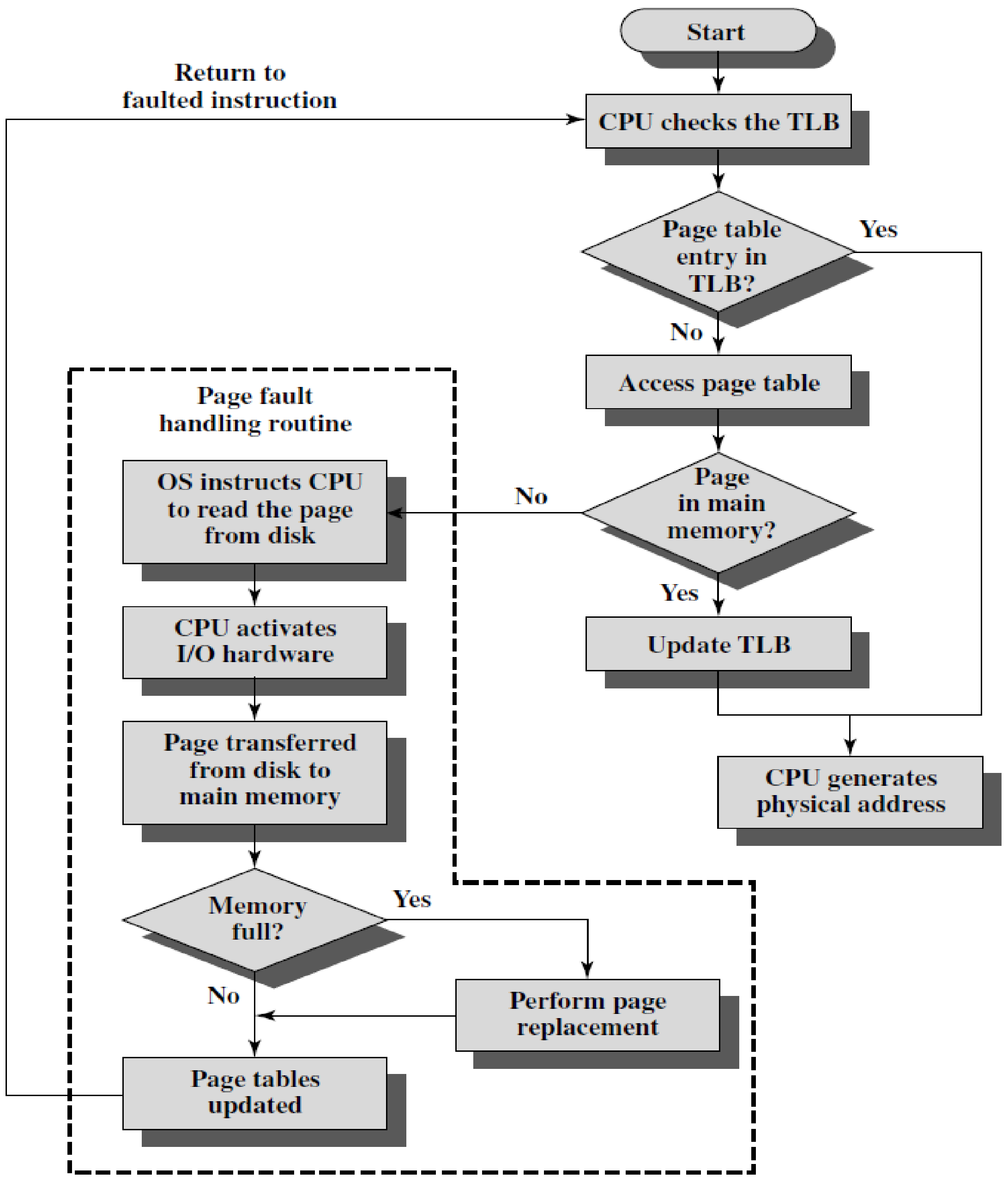

Start
Return to faulted instruction
CPU checks the TLB
Page table entry in TLB?
Yes
No
Access page table
Page fault handling routine
OS instructs CPU to read the page from disk
No
Page in main memory?
Yes
Update TLB
CPU activates I/O hardware
Page transferred from disk to main memory
CPU generates physical address
Memory full?
Yes
No
Perform page replacement
Page tables updated

Figure B.1: Operation of Paging and Translation Lookaside Buffer (TLB)

In principle every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry, and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time.To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called a translation lookaside buffer (TLB).This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. FigureB.1 is a flowchart that shows the use of the TLB. By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache.

Note that the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in FigureB.2. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present. If so, it is returned to the processor. If not, the word is retrieved from main memory. The reader should be able to appreciate the complexity of the processor hardware involved in a single memory reference.The virtual address is translated into a real address. This involves reference to a page table, which may be in the TLB, in main memory, or on disk.The referenced word may be in cache, in main memory, or on disk. In the latter case, the page containing the word must be loaded into main memory and its block loaded into the cache. In addition, the page table entry for that page must be updated.

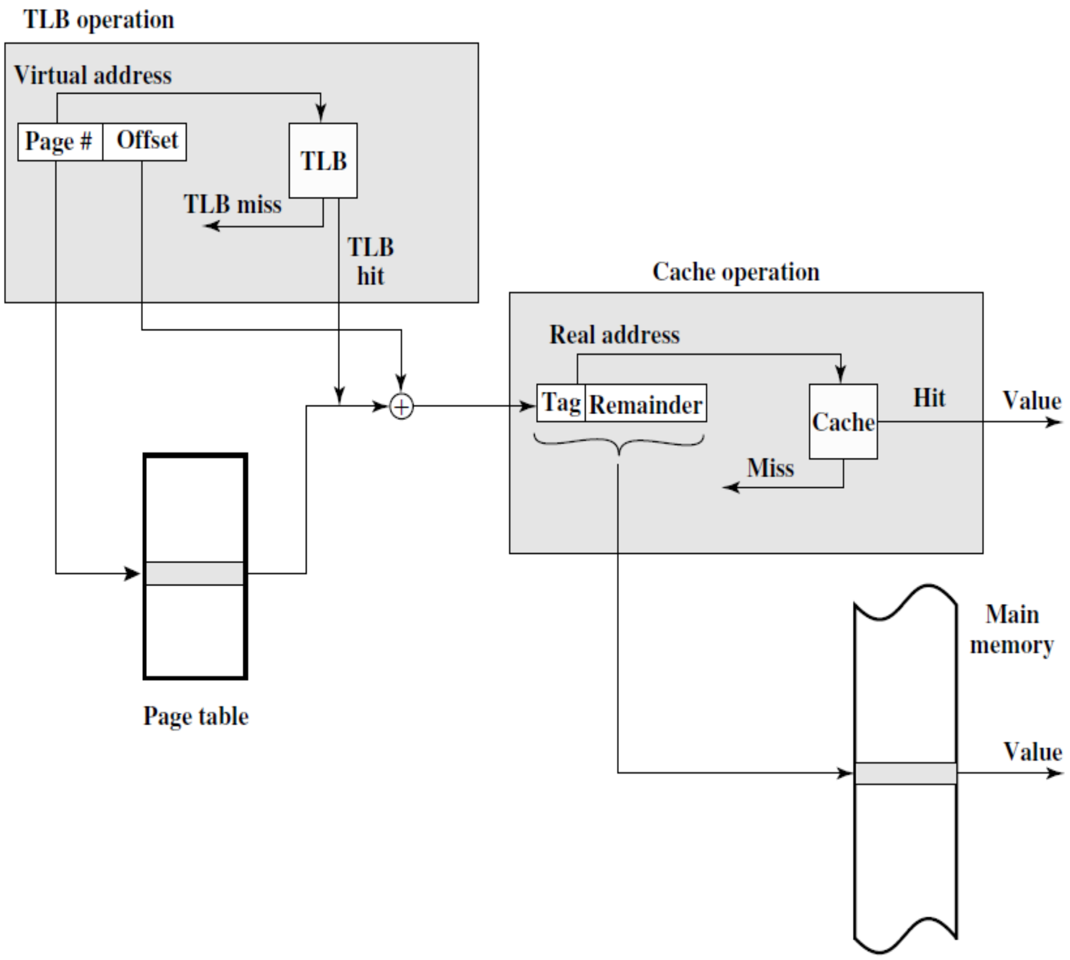


Figure B.2: Translation Lookaside Buffer and Cache Operation

REFERENCES

[1] Sohi G., Roth A. Speculative Multithreaded Processors. IEEE Computer, 2001,34(4): 66-71.

[2] Tullsen D., Eggers S., Emer J., Levy H., Lo J., Stamm R.. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. Proc. 23rd Annual International Symposium on Computer Architecture, 1996:191-202.

[3] Luo Kun, Franklin Manoj, Mukherjee Shubhendu S., Sezne Andre. Boosting SMT Performance by Speculation Control. Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01),2001:9-16.

[4] Limousin C., Sebot J., Vartanian A., Drach-Temam N.. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. Proc. ICS, 2001:236-245.

[5] Tullsen D.M. Brown J.A.. Handling long-latency loads in a simultaneous multi-threading processor. Proc. Of MICRO34,2001:172-180.

[6] Knijnenburg P.M.W., Ramirez A., Latorre F., Larriba J., Valero M.. Branch Classification to Control Instruction Fetch in Simultaneous Multithreaded Architectures. Proceedings of the International Workshop on Innovative Architecture for Future Generation High- Performance Processors and Systems (IWIA02), 2002:67-76.

[7] El-Moursy Ali, Albonesi David H.. Front End Policies for Improved Issue Efficiency in SMT Processors. Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA-903),2003:25-33.

[8] Shin Chulho, Lee Seong-Won, Gaudiot Jean-Luc. Dynamic Scheduling Issues in SMT Architectures .Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS03),2003: 187-192.

[9] HE Li-Qiang, LIU Zhi-Yong. An Effective Instruction Fetch Policy for Simultaneous Multithreaded Processors. Proceedings of the 17th International Conference on High Performance Computing and Grid in Asia Pacific Region(HPCAsia04),2004:243-248.

[10] S. Parekh, S. J. Eggers, H. M. Levy et al. Thread-Sensitive Scheduling for SMT Processors. Technical report, Dept. of Computer Science, University of Washington, 2000.

[11] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. 22nd Annul Computer Measurement Group Conference, December 1996.

[12] Li Ying. Analysis of Instruction Flow Characterization and Instruction Fetching Approach of Simultaneous Multi-Threading Architecture. [Thesis for Doctor Degree of NWPU]. 2004.

[13] J. L. Hennessy and D. A. Patterson, Computer Architecture (5th Edition). Morgan Kaufmann, 2012.

[14] R. S. Nikhil and K. Czeck, BSV by Example. Bluespec, Inc, 2010.

[15] Bluespec, Inc, Bluespec System Verilog Reference Guide, revision: 17 ed., 2012.

[16] User Level RISC-V ISA, University of California, Berkeley, revision: 2 ed., 2014.