ABSTRACT

SABHARWAL, SAHIL. Microarchitectural Implementation of the MIPS System Coprocessor in FabScalar-generated Superscalar Cores. (Under the direction of Dr. Eric Rotenberg.)

FabScalar is a toolset that allows automatic generation and verification of synthesizable RTL cores with varied configurations. It is mainly aimed at vastly simplifying the design of heterogeneous multi-core systems which possess several advantages over homogeneous multi-cores. Harnessing the full potential of a heterogeneous system requires both a clever architecture and substantial system software support. Although significant amount of progress has been made in architecting such a system, much needs to be accomplished on the system software front. The FabScalar MIPS toolset that has full software tool chain support for MIPS provide the platform for booting operating system kernel, running applications as Linux processes and to perform system level research. The FabScalar MIPS design, with MIPS COP 0 support, is an attempt at providing a toolset for such research. COP 0 empowers FabScalar designs to run preemptive, multitasking operating systems, such as Linux, by providing system level support such as virtual memory and a mechanism to handle exceptions and interrupts. This allows for research in the diverse fields of power management, thread and process scheduling and SMT in a real heterogeneous system. In this thesis, we architect and implement the COP 0 in the FabScalar MIPS. There are three facets to this work , 1) Understanding the Linux kernel booting process and compiling and building the kernel image binary that can be booted on FabScalar toolset. 2) Emulating serial devices in C++ which is required for running a kernel on the design. 3) Design and implementation of Coprocessor 0 module, which mainly consists of TLB/Address translation mechanism and exception handling mechanism,

leveraging the existing structures. The results of designing such a system are discussed along with challenges faced and solutions reached while designing it.

Microarchitectural Implementation of the MIPS System Coprocessor in FabScalar-generated Superscalar Cores

by
Sahil Sabharwal

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2013

APPROVED BY:

_____	_____
Dr. Eric Rotenberg Committee Chair	Dr. Huiyang Zhou
_____	
Dr. Gregory Byrd	

DEDICATION

To my parents, grandparents and my brother...

BIOGRAPHY

Sahil Sabharwal was born in New Delhi, India, on 6th February, 1988. He recieved his Bachelor of Technology degree from Kurukshetra University, Haryana, India, in 2010. He joined the North Carolina State University in fall, 2011, to pursue Master of Science degree in the field of Computer Engineering. He will be receiving his Master of Science degree with the defense of this thesis.

He will be joining IBM, Austin, System Architecture and Performance group (STG) from August 2013.

ACKNOWLEDGMENTS

I express my sincere gratitude and thank my advisor, Dr. Eric Rotenberg for his constant guidance and support throughout my thesis.

I would like to thank Dr. Gregory Byrd and Dr. Huiyang Zhou for agreeing to serve on my thesis committee. Many thanks to my family and friends for their support and blessings in achieving my dreams and goals.

I would also like to thank my friends Rangeen Basu Roy Chowdhury and Amro Awad for keeping me motivated.

This research was supported by NSF grants CCF-1218608 and CCF-1018517. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Motivation and Contribution

Our research group has developed a toolset known as FabScalar [1] which automates the generation of synthesizable RTL processor cores of different widths and depths. The FabScalar toolset was originally designed for PISA [7] which is MIPS look-alike instruction set architecture but recently we have moved to commercial instruction set architecture MIPS that has rich system support. We migrated from FabScalar PISA [7] to FabScalar MIPS[1] so that we can utilize the full MIPS software tool chain (Compiler/Operating System) for booting operating system and ultimately for systems related research on a FabScalar toolset.

The current FabScalar MIPS version boasts support for the integer and floating point MIPS instructions. Although this is sufficient for running standalone executables, the absence of virtual memory support and exception handling mechanism will prevent operating system from running on the design. This will prevent us from leveraging the software ecosystem that MIPS has to offer and hence hinders the use of FabScalar toolset in system level research. My contribution in the FabScalar MIPS toolset is the design and development of COP 0, and to provide the device functional support necessary for running a real world operating system such

[1] My colleagues Brandon Dwiel, Elliot Forbes and Ashlesha V. Shashtri were involved in developing and designing the FabScalar MIPS version. Ashlesha in particular was involved in adding MIPS floating point instruction support in FabScalar MIPS toolset.

as Linux. To boot an operating system on the MIPS CPU, system COP 0 is required. COP 0 is a MIPS ISA subset that provides an abstraction of the functions necessary for system level support such as kernel code, exception handling and memory management unit/Address Translation. The MIPS coprocessor 0 ISA subset is integrated with the FabScalar toolset MIPS 32 release2. The OS kernel also assumes the availability of some serial devices which we emulated in the FabScalar framework. The devices that we have emulated are Interrupt Controller 8259 and ns16550 Serial I/O.

1.2 Coprocessor 0 Module

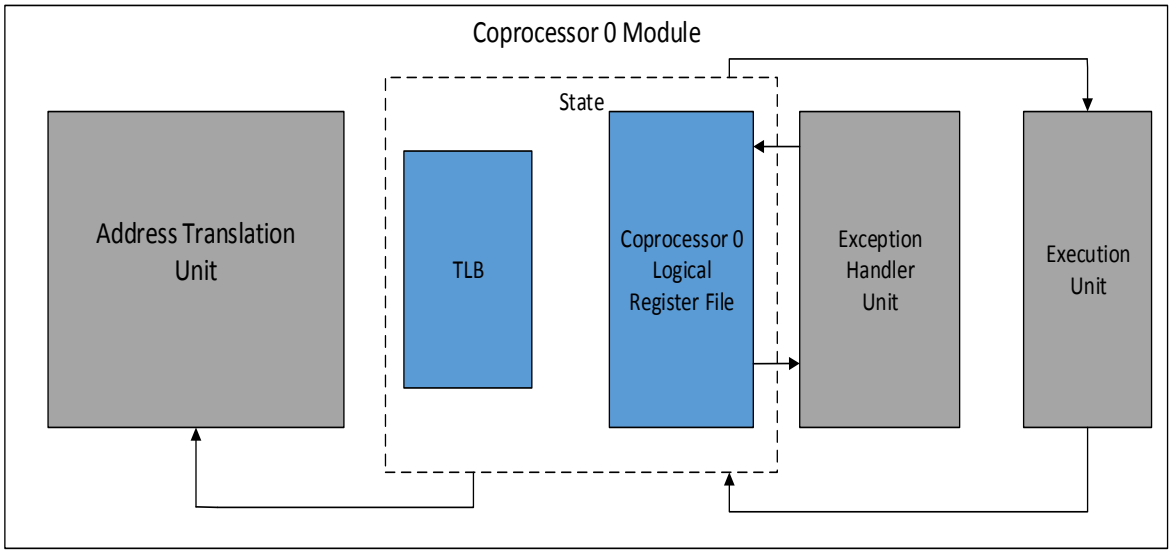

Figure 1. High Level Diagram of COP 0 Module

In this section we explain, at a high level, the description of Coprocessor-0 (COP 0) and the strategies which we have deployed to implement COP 0 logic in canonical pipeline type processor.

The COP 0 module mainly has four components, a) State of COP 0, b) Address Translation Unit, c) Exception Handling Unit, d) Execution Unit

1.2.1 State of COP 0

The architectural state of COP 0 at any instant of time is depicted by

- 32 COP 0 Registers

  It consists of 32 registers, each 32 bit wide. Some of the important uses of these registers are in address translation process, maintaining the status and cause of exceptions at any given point of time and configuring the caches and TLB.

- **T**ranslational **L**ookaside **B**uffer (TLB)

  COP 0 has a unified software-managed TLB which can be configured for 16 to 64 entries. Each entry contains the virtual address and the corresponding physical address.

1.2.2 Address Translation Unit

This unit is responsible for translating the virtual addresses into the physical addresses. This unit interacts with the Fetch Stage and AGEN (Execute Stage) and translate their virtual address to physical address before they access memory hierarchy.

1.2.3 Exception Handling Unit

When the posted exception or instruction with ERET bit high reaches the head of Active List, Retire Stage signals this unit with the cause (exception or ERET instruction) and if the exception is signaled, type of exception is also signaled along with it. Depending upon the signal, this unit is responsible for redirecting the Program Counter to the correct address.

1.2.4 Execution Unit

This is the main execution machinery responsible for the correct execution of COP 0 instructions.

Classification of Instructions into Groups

The instructions are classified into different groups which are based on the different strategies that we have employed to handle them correctly in the processor pipeline.

- Group1 (Move Instructions):

  MTC0: move the contents of a general register to a coprocessor 0 register.

  MFC0: move the contents from coprocessor 0 register to a general register.

- Group 2 (TLB Management Instructions):

  TLBW: The TLB entry pointed to by the Index register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers.

  TLBR: The EntryHi, EntryLo0, EntryLo1, and PageMask registers are loaded with the contents of the TLB entry pointed to by the Index register.

TLBP: The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register.

ERET: To return from interrupt, exception, or error trap. It restores the PC (from Coprocessor 0 register **EPC)** from which the processor start executing after handling exceptions/interrupts.

- Unimplemented Instructions:

  RDPGPR: Read GPR from previous shadow set.

  WRPGPR: Write to GPR in previous shadow set.

  MFMC0: further encoding based on bit 5. Disable Interrupt (DI), Enable Interrupt (EI)

Strategy for implementing instructions

There are several ways of handling these instructions depending upon the type of COP 0 state being updated. We didn't want to rename these instructions as renaming the TLB instructions that affect address translations would have not been the clean way to handle the coprocessor instructions that manages the system related operations of main processor. Renaming these instructions would have also bring the additional complexity as the TLB related instructions don't have explicit source and destinations registers. These instructions read and write to coprocessor 0 registers EnrtyLo1, EnrtyLo2, ASID and Index register.

As coprocessor 0 instructions are not renamed, implementing these instructions correctly with respect to the out of order backend pipeline stages was a challenging task because of which we came up with different strategies and implemented the simplest one.

1. Serializing: When the instruction is encountered in the dispatch packet that tries to change the COP 0 state, we heavily serialize the execution of such instruction using Finite State Machine in Dispatch Stage.
2. Scoreboard Technique: This technique maintains the scoreboard for COP0 instruction in the issue stage. We handle TLB instructions and COP0 Read/Write instructions bit differently as TLB instruction can affect the way address translations are done. We maintain Scoreboard to check for any pending COP 0 instruction in the Issue Queue. The details about these techniques are discussed in Chapter-6, section 6.2.

1.3 Location in pipeline

There are several ways in which COP 0 can be integrated with the current FabScalar canonical pipeline. One of the ways is to create a separate execution lane for COP0 instructions which would create additional logic and unnecessary complexity. We have used Load Store Unit (LSU) execution lane for logically placing the COP 0 module so that we can leverage the existing structures.

The rationale behind choosing the LSU lane is that it already has access to physical register file with two read ports and one write port. The access to the Physical Register File by LSU execution lane supported the execution of instructions that communicate data between

general purpose CPU logical registers and COP0 registers. This lane is also closer to and interfaces with the memory hierarchy (D-Cache, Load/Store lane and Main Dram memory). Since loads and stores come to the LSU execution lane for address generation, logically placing COP 0's TLB there for virtual address to physical address translation provided much clearer and less complex interaction between the memory hierarchy and the load/store queue. As MIPS has unified TLB for both I-Cache and D-cache, the fetch stage also interacts with COP0's Address Translation module every cycle for translating its PC address into physical address. Hence, in the physical design layout, fetch block will be closer to AGEN unit. The COP 0 module is responsible for signaling the exception high signal and the type of exception like TLB miss, Syscall exception, etc, to the appropriate entry of the instruction causing exception in the active list using alId value. As the LSU lane already interacts with active list for updating execution flags and exception flag field, placing COP0 module in LSU execution lane enabled clearer implementation and interaction of exception/interrupt handling module with the active list.

CHAPTER 2
BACKGROUND

The COP 0 design was implemented on the existing FabScalar infrastructure [1] which enables the automatic generation of synthesizable RTL superscalar cores with canonical pipeline stages of different widths and depths.

2.1 FabScalar Template

The FabScalar toolset provides a co-simulation environment in which two superscalar processor simulators run in conjunction with each other. One is written in Verilog Hardware Description Language and the other is written in C++ that performs functional execution of instructions. The C++ functional simulator fetches and executes one instruction at a time in program order and is used to initiate the FabScalar RTL simulator by initializing its memory and architectural state. The C++ functional simulator is also used for functionally verifying and debugging the Verilog simulation instruction results by comparing with its own results at the time of retirement.

FabScalar canonical pipeline template as shown in Figure 2 consists of nine canonical pipeline stages. The current FabScalar template supports integer and floating point instructions set as shown in Figure 3.The instruction fetch is divided into Fetch-1 stage and Fetch-2 stage. Ftech-1 stage consists of next PC logic that generates the address from which the next instruction is to be fetched and contains structures like Branch Predictor (BP), Branch Target Buffer (BTB) and Return

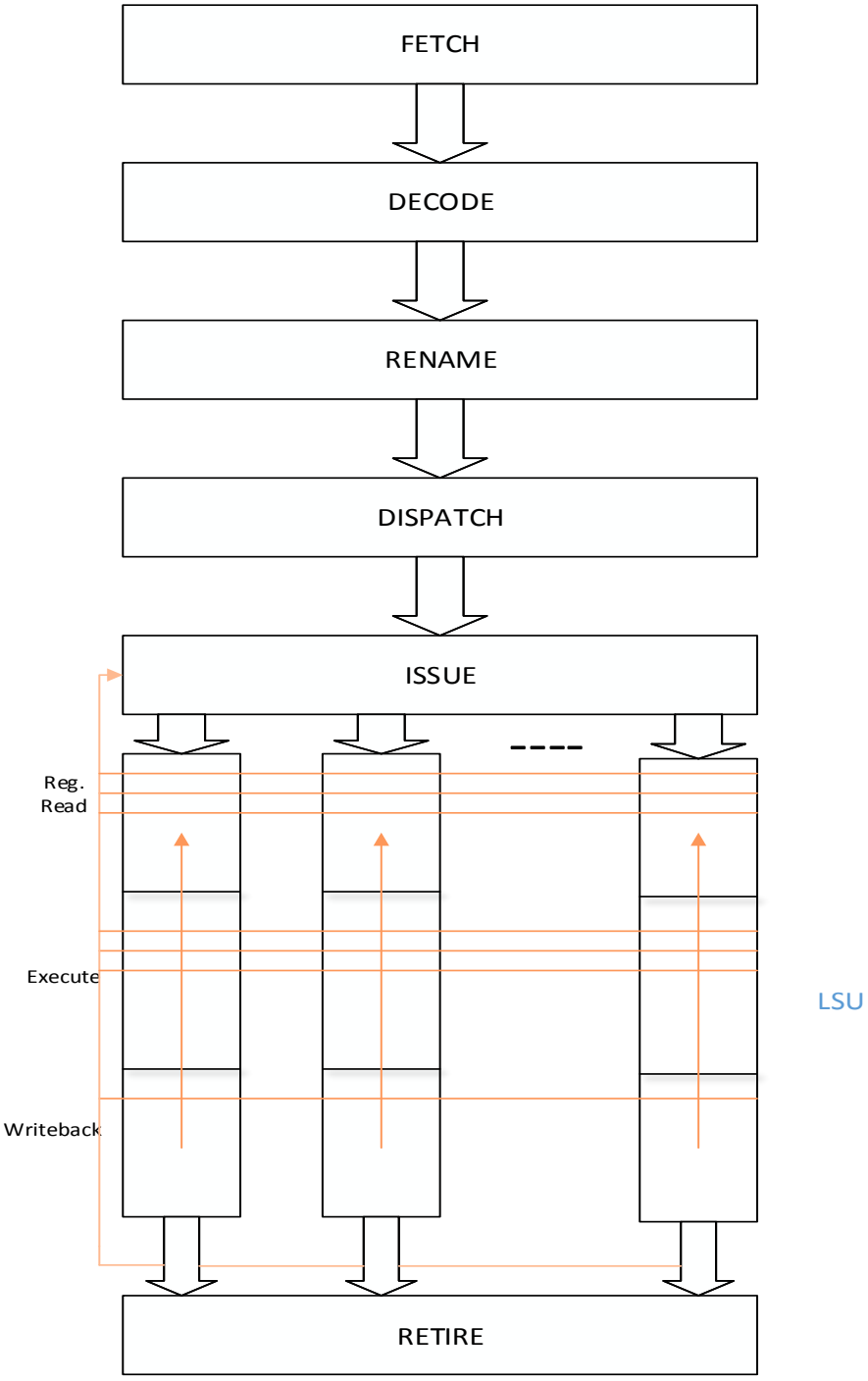

Figure 2. FabScalar Template

Address Stack (RAS). Fetch-2 stage mainly consists of a control queue, which updates the Branch Predictor and BTB with branch results at the time when branch commits from active list, and Pre-Decode Block which pre-decodes the branch in the fetch bundle.

The decoder stage decodes in parallel the incoming instruction binaries and generates the packet per instruction which contains information about instruction and sends it to Instruction Buffer which is a circular FIFO. The decoder logic has the capability to split the instruction into three micro-instructions, mainly to support floating point instruction execution. At the time when backend is stalled, Instruction Buffer allows continuous fetching and decoding of new instructions every cycle by buffering them in the queue.

The packet from the head of the buffer is popped and go to rename stage if the back end is ready. This stage contains two instances for rename machinery, one for integer instructions and other for floating point instructions. The source operands are renamed by getting their current physical register mapping from Rename Map Table (RMT) while destination register if valid is renamed by popping the new physical register mapping from the free list queue that maintains the mapping of physical register that are not in use i.e neither in active list nor in Architecture Map Table (AMT). In case of branch misprediction, exceptions or external interrupts, the mapping of the RMT is restored by copying AMT to RMT. The dispatch bundle formed after renaming is sent to dispatch stage.

The dispatch stage is responsible for creating packets for Integer and Floating point Issue Queue, Active List and Load Store Queue and dispatch them into the relevant queues after checking the availability of the required number of instruction in each queue. The dispatch

logic is designed such that it splits the bundle into two whenever it sees the mix of floating point and integer instructions in the dispatch bundle. One bundle contains integer instructions that are steered into Integer Issue Queue and the other contains floating point instructions that are steered to Floating Point Issue Queue. This stage also contains Execution pipe scheduler that assigns each instruction with the execution lane number based on its opcode decoded in decode stage. This simplifies the select logic in the issue queue when instruction arbitrates for the lane.

The Issue queue stage separates the in order front end from the out of order back end. The instructions that are dispatched gets written into the Issue queue Payload RAM. The entries for new instructions are created and maintained by issue queue free list. The select logic selects instructions to issue queue every cycle if their source operands are ready and the execution lane for which they are selected to execute in is available. After the instructions are issued from the issue queue, they wake up their dependents and set the ready bit in physical register ready bit array corresponding to its physical register destination.

In the register read stage the instructions read their source register values, either from the corresponding entry in the physical register file or from the bypasses. The unified FabScalar Template consists of separate Integer and Floating Point physical register file. The Floating point PRF is used by floating point instructions and each entry is 64 bits wide as compared to the 32 bit-wide entry of Integer PRF.

After instruction gets executed in the execute stage, the result is written into the physical register file and also broadcasted along bypasses in the write back stage.

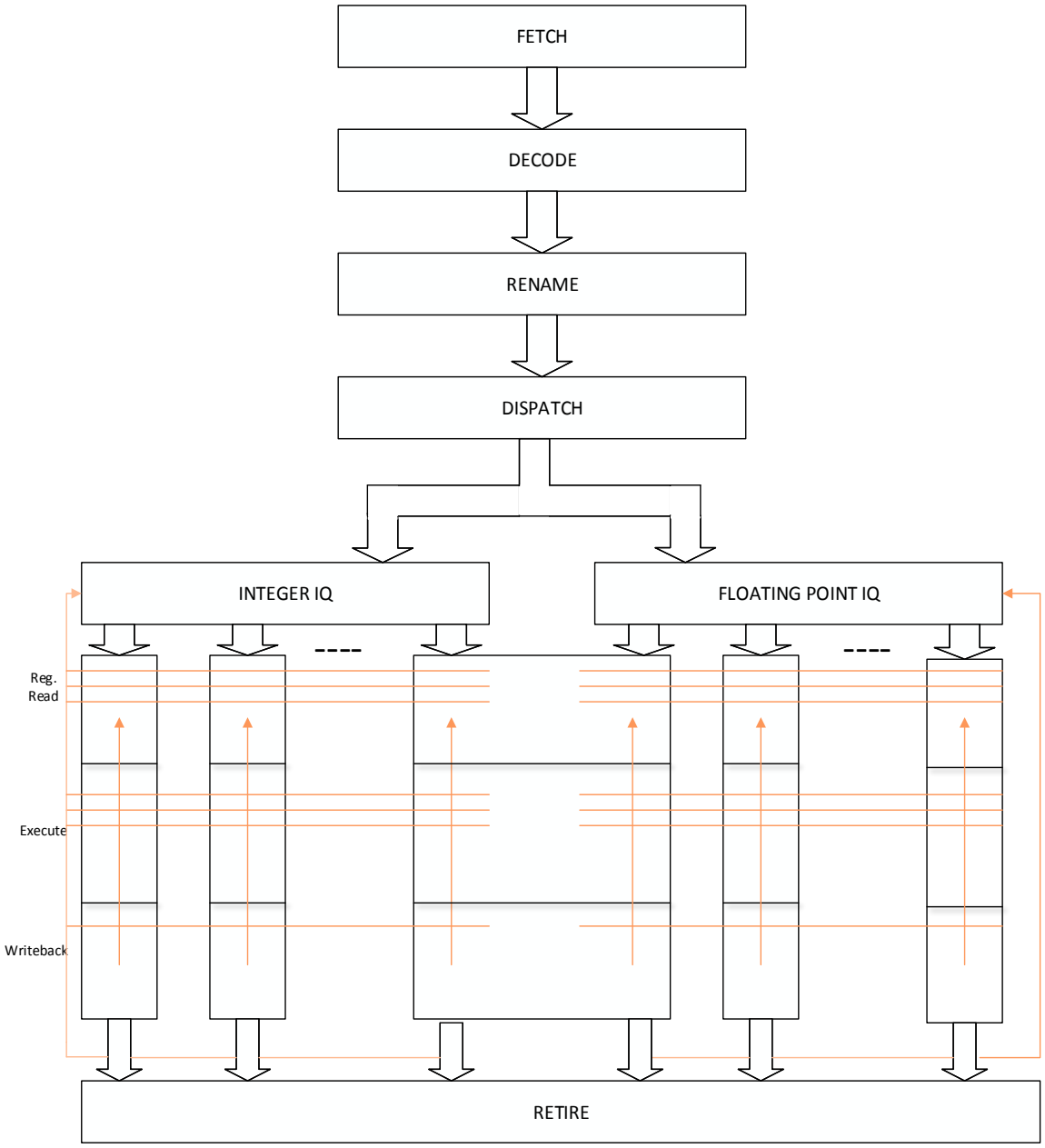

Figure 3. FabScalar with Floating Point Support

The instructions are committed in order from the head of active list. It maintains the execution flags and control flags for each instructions which it checks before commit. If the instruction that is committing has a valid destination, the corresponding entry in the AMT is updated with the instruction's destination physical register mapping, which in turn causes the current mapping of the logical register in the AMT to get free and is pushed at the tail of physical register free list.

I initially implemented COP 0 functionality in C++ functional simulator and emulated serial I/O ns16550 and UART controller 8250 as a class module in C++ language. After I successfully booted Linux kernel on the C++ functional simulator, I designed COP 0 in Verilog RTL that has Address Translation unit, exception handler and a register file containing 32 logical registers.

2.2 FabScalar Template with COP 0 Support

This section provides the high level description of the changes made to the FabScalar template structure to integrate COP 0. The Figure 4 shows the FabScalar template with COP 0 module integrated that provides system level support for operating system related research.

1. In the fetch-1 stage, the next PC logic is changed such that next PC comes from the address stored in COP 0 register Exception Program Counter (EPC) when ERET instruction is encountered in execute stage.

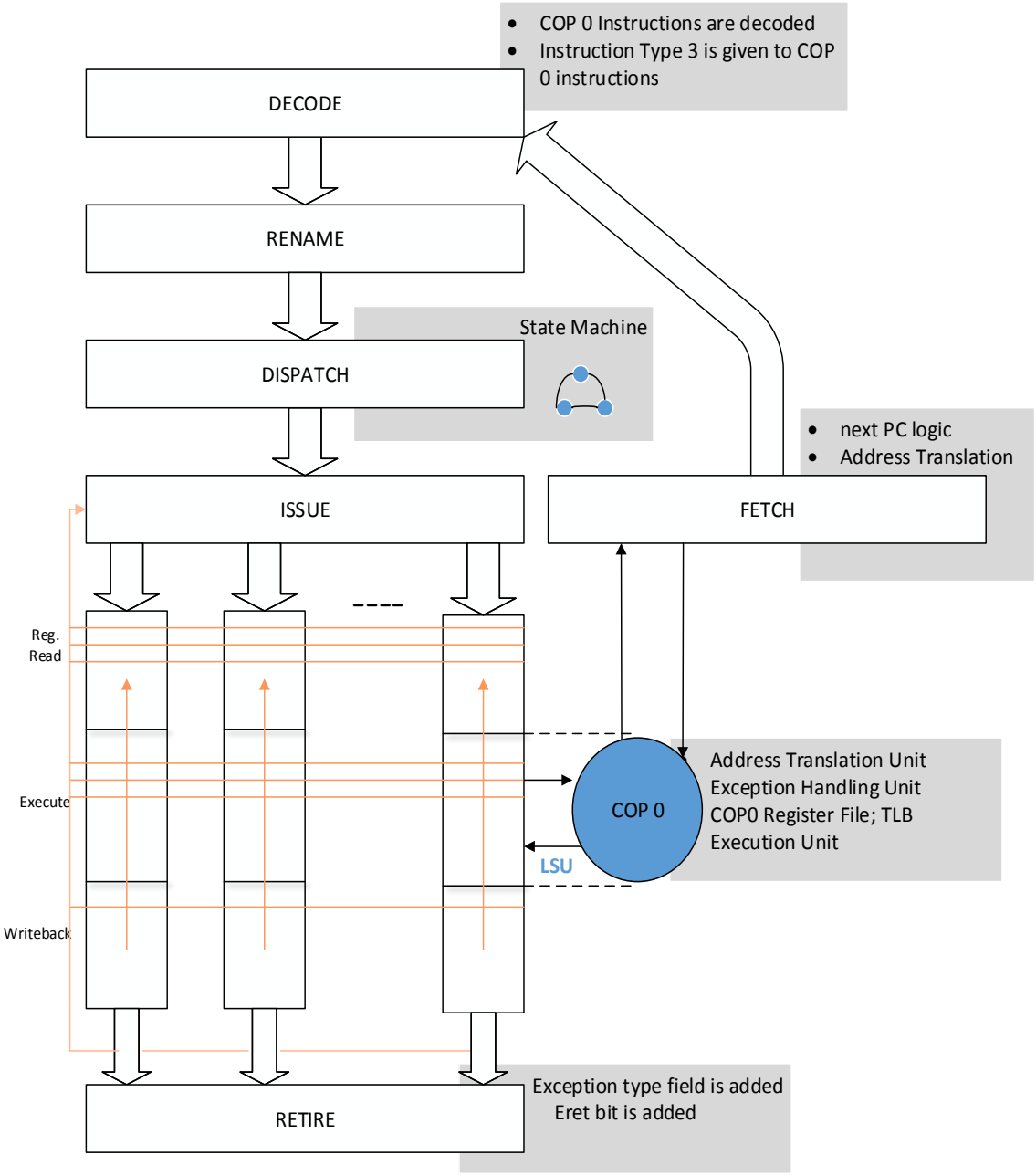

Figure 4. FabScalar template with COP 0 Integrated

2. In the decode stage, the INSTRUCTION TYPE3 is given to the COP 0 instructions such that they get issued to LSU execution lane in the issue stage. The decode packet is modified to include COP 0 destination and source fields. The COP 0 instructions and their registers are decoded.

3. The dispatch logic is modified to include the state machine that stalls the frontend and does the partial dispatch when the COP 0 instruction is encountered in the dispatch packet that tries to change the state of the COP 0.

4. The Load-Store execution lane is the place where most of COP 0 logic and its state resides. It consists of the following main structures:

- Address Translation Unit is added to translate virtual addresses into physical addresses. It interface with fetch-1 pipeline stage and AGEN unit to convert program counter and load-store addresses. The detailed explanation of the implementation is given in Chapter4 Section 4.4.
- The COP 0 register file consisting of 32 registers is added. Each register is 32 bit wide and it interfaces with register read stage write back stage which writes and reads the values from register file for MTC0 and MFC0 instructions.
- Exception Handler is responsible for redirecting the PC of the processor to the software exception handler base address and saving the state of some of the COP 0 registers. The more detailed explanation can be found in Chapter 5, Section 5.2.

CHAPTER 3
BOOTING LINUX

There were several steps involved in porting Linux to our target platform i.e Fabscalar MIPS. In this chapter we have explained the high level kernel source tree structure, steps involved in building kernel image binaries with various configuration options we selected for our target machine (MIPS), devices that we needed to emulate for successfully booting Linux and kernel start up process and brief description about important functions that are involved in this process.

3.1 Kernel Source Tree

One of the most important steps is to have a basic understanding of the kernel source tree organization. The top directory of kernel source code has following subdirectories:

1. arch: This directory contains all the architecture related code. Each architecture (MIPS, X86 and ARM) subdirectory has four main subdirectories:

- kernel, which contains the architecture-specific kernel code.
- mm, which contains the architecture-specific memory management code.
- lib, which contains architecture specific library code (vsprintf and so on)
- MY_PLATFORM (Qemu), which contains platform-specific code.

2. documentation-This subdirectory contains the documentation for the kernel

3. drivers-This subdirectory contains code for the device drivers. Each type of device has further subdirectories, such as char, block, net, and so on.
4. fs-This directory contains the file system code. This has further sub-directories for each supported file system (ext2, proc, etc).
5. include-The include subdirectory contains the include files for the kernel. It has further subdirectories for common include files (for all architectures), one for every architecture supported, and a couple of other subdirectories.
6. init-This directory contains the initialization code for the kernel.
7. kernel-This directory contains the main kernel code.
8. lib-This directory contains the library code of the kernel.
9. mm-This directory contains the memory management code.

3.2 Kernel Startup Process

The system startup process and the functions involved in the Linux booting process with their brief description are as follows:

- Set desired endian mode
- Clear the BEV bit
- Set the TLB bit
- Jump to cpu_probe and return
- Set up stack for kernel
- Clear bss

- Jump to prom_init and return
- Jump to loadmmu and return
- Disable coprocessors that is set to be disabled in COP0 configuration register.
- Jump to start_kernel

The configuration (Config) register needs to be initialized properly. The desired endian mode should be set by setting BE bit in Configuration register which for our case is set to little-endian mode. The bootstrap exception vector bit needs to be cleared in Status register to make sure that the normal exception vectors are taken by processor in case of exception. The MT (MMU Type bits 9:7)) field in Configuration register is set to standard TLB (encoding of 1) to make sure that TLB-based address translation is used.

The next step is to probe for the cputype. $(TOPDIR)/include/asm/bootinfo.h contains entries for the cputype (MYCPU) and machine group (MY_MACH_GROUP). The mips_cputype variable has to be updated in the cpu_probe function. This value is used later to determine the exception handling and MMU routines that need to be loaded for the given CPU, as well as to get the CPU information in the /proc file system.

The prom_init() function, which is part of $(TOPDIR)/arch/mips/qemu/prom.c, modifies the command line string to add parameters that need to be passed to the kernel from the bootloader. The machine group and upper bound of usable memory are set up here.

```
asmlinkage void __init start_kernel(void)
{
      char * command_line;

       /*
        * Interrupts are still disabled. Do necessary setups, then
        * enable them
        */
       lock_kernel();
      printk(linux_banner);
      setup_arch(&command_line, &memory_start, &memory_end);
      memory_start = paging_init(memory_start, memory_end);
      trap_init();
      init_IRQ();
      sched_init();
      time_init();
      parse_options(command_line);
      .
      .
      .
}
```

Figure 5. Psuedo code for start Kernel function

setup_arch()

This function is in $(TOPDIR)/arch/mips/kernel/setup.c. The board-specific setup function is called from here. The command line string and memory start and memory end are passed over to the caller of this function. The start and end addresses for the linked-in ramdisk image are updated here.

trap_init()

This function loads the exception handler code to the KSEG0 vector location which for MIPS is at 0x80. The interrupt handling code is located at $(TOPDIR)/arch/MY_ARCH/MY_PLATFORM/irq.c. Generally the systems have dedicated exception handler for handling various exceptions/interrupts. The exception handler code is required to service exception which causes the transfer of control to the operating system.

3.3 Building Kernel Image

The Makefile in the $(TOPDIR) either has to have ARCH (architecture) properly defined or it can also set during invocation of make: make ARCH=mips. The Makefile in $(TOPDIR)/arch/mips/boot has to have CROSS_COMPILE set to mips-linux-gnu (MIPS little-endian cross-compiler tool-set) and LOADADDR (address at which the kernel image would be loaded) defined as per the configuration. The ARCH is by default set to the host machine which for our case is i386 and automatically the binaries are compiled for the same. The options ARCH and CROSS_COMPILE can be overridden through command line option: make ARCH=mips CROSS_COMPILE=mips-linux-gnu- menuconfig.

The Figure 6 shows the configuration window that popes up after this command. The kernel has to be configured using to the barest minimum needs which for our case is mips32 release 1, little endian, serial driver, CPU type = Qemu, ramdisk driver, kernel support for elf binary, ext2 file system. If additional configuration options have to be added, the $(TOPDIR)/arch/mips/config.in file has to be modified. You would need to have a config

option for your platform (CONFIG_MYPLATFORM) to include code that is specific to your platform. Then do a "make dep" to link up all the dependencies and finally a "make vmlinux V=1" to produce the elf binary kernel image “vmlinux”. The command V=1 enables the compilation with more focus on warnings and less verbose as default.

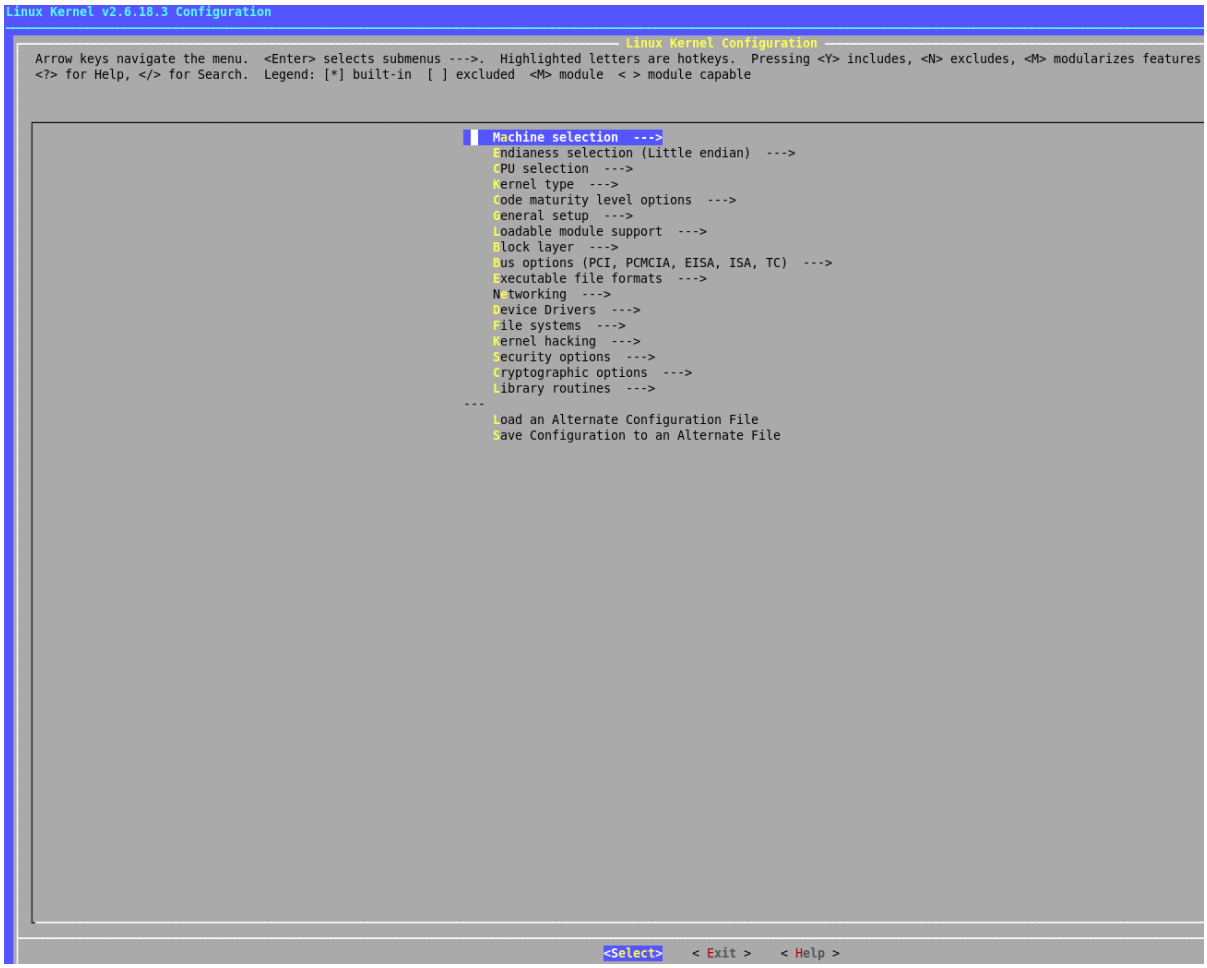

Figure 6 Kernel Compilation - makeconfig GUI

3.4 Initial Ram Disk Image Loader

The initial ram disk image known as initrd image in Linux is a temporary root file system that supports boot process. It has various drivers and executables that enables the real root file system to get mounted. It is usually stored as the compressed image and can also be modified to include user level applications. The initrd image loader was written to load the image directly to the memory by passing the memory setting file through the command line.

The memory setting file looks like this:

@mem 00800000 image/initrd.gz

@mem 00fffef0 image/arg-qemu.bin

After the real file system is mounted, the initrd RAM disk is unmounted and its memory gets freed.

3.5 Emulating Devices

We emulated 8259 like Programmable Interrupt Controller (PIC) and Serial IO and they both are initiated under one C++ class IsaIO which implements ISA bus like structure. Serial ports must be visible as tty devices from user space applications. Hardware ttys (eg. Serial lines, text mode console) have one end connected to hardware and one end connected to software (eg. shells). The SimMips [2] simulator was very helpful in providing system level support to our toolset infrastructure as we leveraged their device emulation functions.

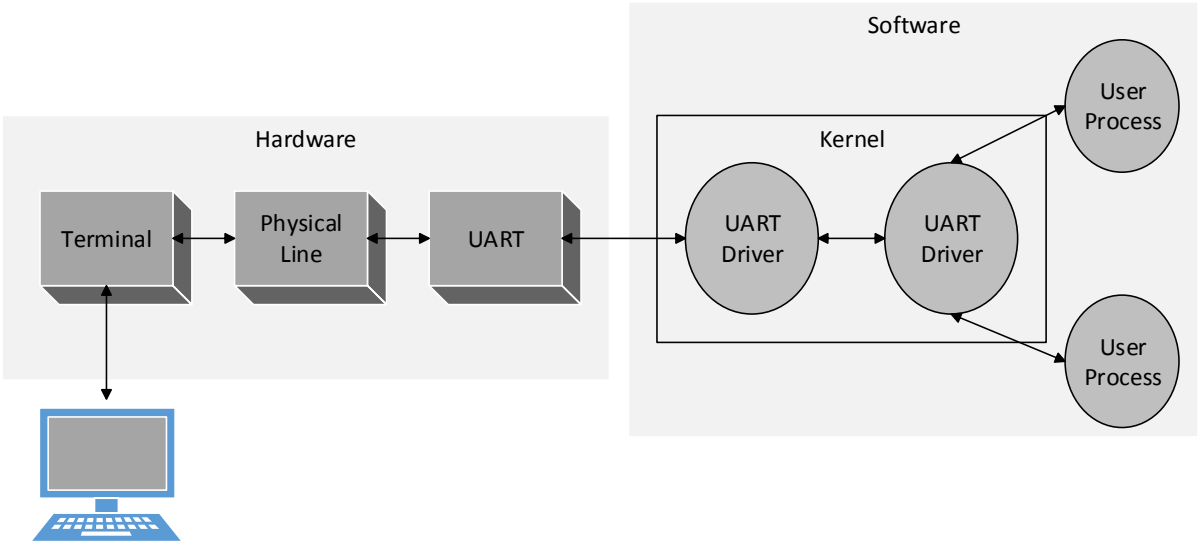

Figure 7. Serial Device Software-Hardware Interface

A user enters at terminal which is connected to UART (Universal Asynchronous Receiver Transmitter) through wires. The linux operating system has UART driver that is responsible for physical transmission of bytes and parity checks. In our simulator infrastructure we implemented two 8259 like UARTs that emulates physical UART functionality. We used termios functions in our simulator that describes a terminal interface provided to control asynchronous communication ports. These functions provides the mechanism to control from userspace serial port parameters such as speed, parity, byte size, stop bit, hardware handshake and etc.

3.6 Kernel Command Line Arguments

There are few command line arguments to the kernel that we used and passed these arguments as the separate file to the simulator that loaded these arguments at a specified location in the memory:

- Console: It allows the early boot messages to be printed and can be enable using the console=kernel argument. This console setting argument is used to specify the serial console port as ttyS0 which is the primary hardware terminal attached to the first serial port of the host.
- rd_start: The base address of the loaded ramdisk image.
- rd_size: The size of the loaded ramdisk image.

3.7 Boot Message

The command line given to run the simulator is: ./funcsim - -job job mem_fabscalar.txt.

The job file specifies the kernel image binary that needs to be booted and the file mem_fabscalar.txt specifies the initrd ramdisk image and the kernel argument binary file with the addresses at which they should be loaded in memory by ramdisk loader. On running this command we should see the boot messages as shown in Figure 8 on our terminal.

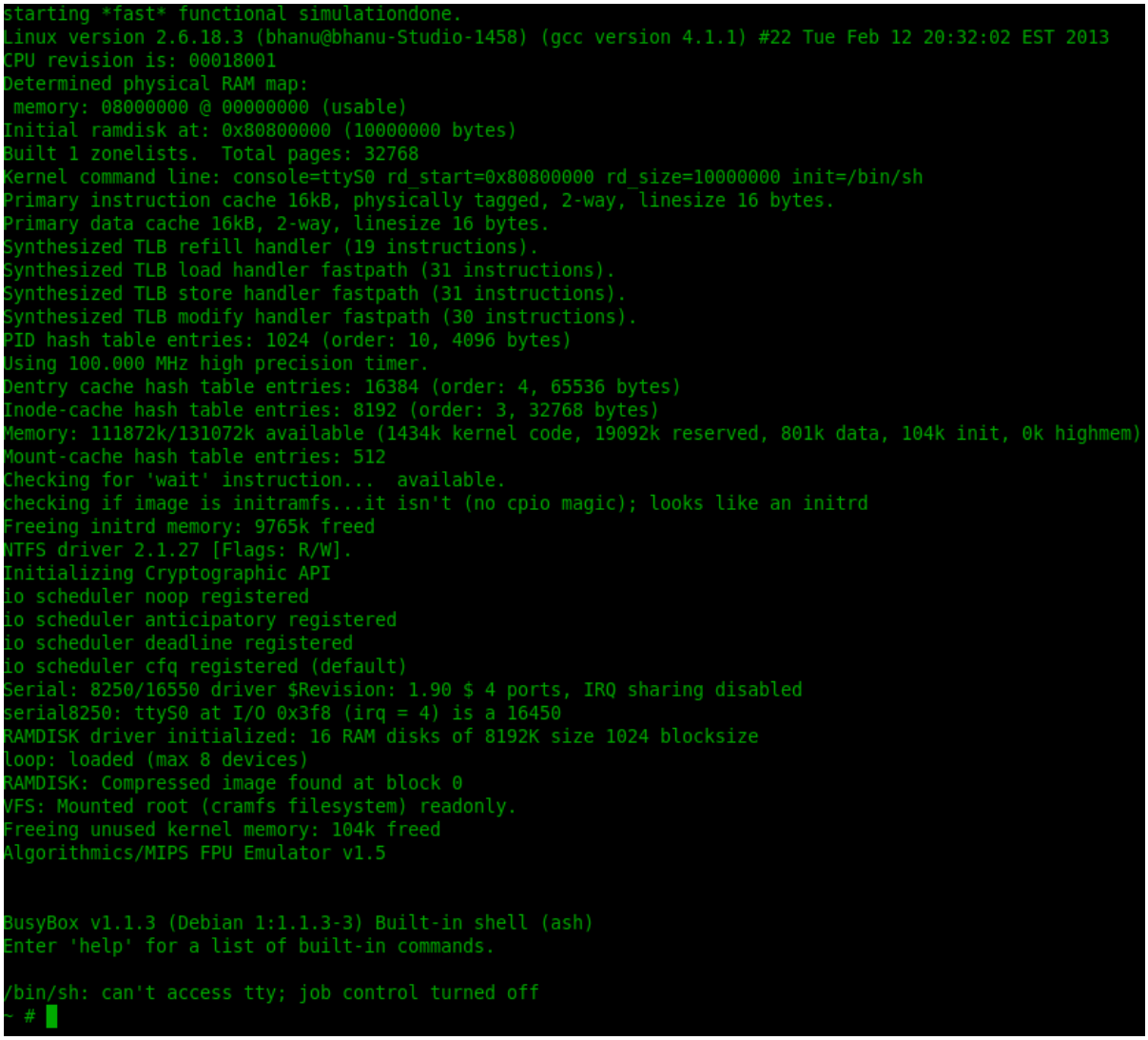

Figure 8. Kernel Boot Message on Shell

CHAPTER 4
ADDRESS TRANSLATION UNIT

The COP 0 structure is modular in nature and has been added with configurable option. It can be disabled from FabScalarParam.sv (Parameter setting file) which will enable to execute application binary without COP 0 and Linux support which means no address translation (Virtual address to Physical address) and no software exception/interrupt handling mechanism. The following section explains the address translation process which is used by fetch and execute stage.

4.1 Virtual Address Space

When virtual address reaches address translation unit, it checks for the address range and the segment in which the virtual address falls. The virtual address space is divided into total of five segments as shown in Figure 9. The segments can be either Mapped or Unmapped. The Mapped addresses are those that are translated through the Translation Look aside Buffer (TLB), whereas unmapped addresses are those which are not translated through TLB but instead it undergoes a fixed translation which provides a window into the lower portion of physical address, starting physical address zero with the size corresponding to the size of unmapped segment. Any reference to the uncached segment will bypass all the levels of cache hierarchy and will allow the direct access to memory.

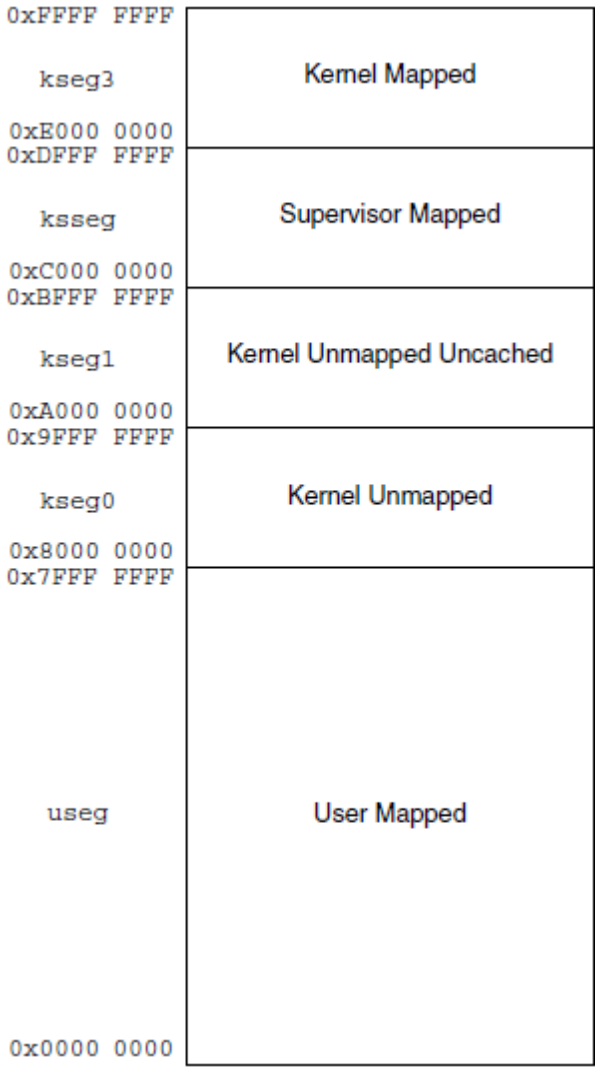

Figure 9. Virtual Address Range

Table 1. Physical Address Generation using Fixed Translation

Segment Name	Virtual Address Range	Physical Address Generation
Kseg1	0xBFFF FFFF through 0xA000 0000	0x1FFF FFFF through 0x0000 0000
Kseg0	0x9FFF FFFF through 0x8000 0000	0x1FFF FFFF through 0x0000 0000

The Table 1. shows how the virtual address (Unmapped Address) gets translated into physical address if it falls under fixed address translation.

4.2 TLB

Before explaining how Mapped addresses gets translated using Translation Look aside Buffer, it is necessary to know how the TLB is organized, what are its contents. Figure 10 shows the contents of the TLB. MIPS doesn't have separate TLBs i.e I-TLB for fetching instructions from memory and D-TLB for accessing data from memory, instead it has unified TLB for both purposes.

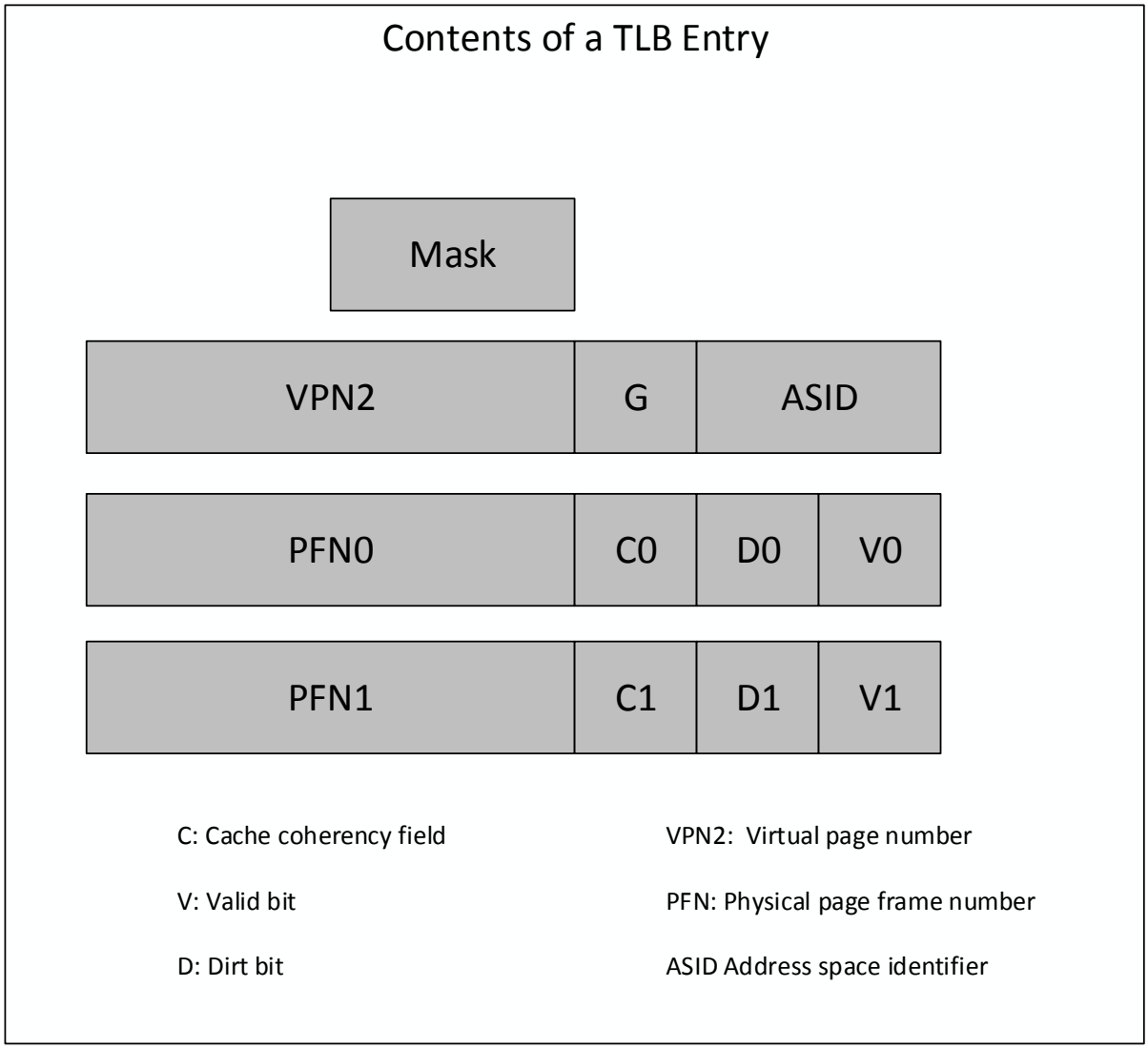

Figure 10. Translation Look Aside Buffer Entry

4.2.1 TLB Organization

The TLB is a fully associative structure and the number of entries that TLB should have can be configured using Coprocesso-0 Configuration register (MMU) field. The maximum entry it

can be configure for is 2pow6 64. The each entry in the TLB contains virtual page number (VPN2, the virtual page number/2 since each TLB entry maps to two aligned pair of physical pages), ASID for address space identification, global bit (G) which overrides the ASID comparison during translation as Operating Systems may sometimes want to associate same virtual addresses with all the processes spawned by it, pair of aligned physical page frame number PFN corresponding to the even and odd pages of the pair. The fields in the TLB entry corresponds to the fields in the COP 0 registers EntryHi, EntryLo0 and EntryLo1. The instruction TLBW (TLB Write) instruction uses registers EntryLo0 and EnrtyLo1 to fill the entries corresponding to the PFN0 and PFN1 fields, whereas TLBR (TLB Read) instruction reads the content of corresponding PFN0 and PFN1 fields into EnrtyLo0 and EnrtyLo1 registers.

4.2.2 Working of TLB Translation

The translation process involving virtual address to physical address translation contains two logical components:

- Comparison Section (TLB CAM Structure)
- Physical Translation Section (TLB Payload RAM Structure)

TLB CAM:

This module is used to find the matching entry in the TLB corresponding to the virtual address that is broadcasted to it. Each entry in the CAM is of TlbCamPkt (TLB CAM Packet) type. The Figure 11 shows the structure of TLB CAM Packet.

```
typedef struct packed {
      logic [`SIZE_DATA-1:0]          vpn2;
      logic [`SIZE_DATA-1:0]          pageMask;
      logic [`SIZE_DATA-1:0]          asId;
      logic                           global;
} TlbCamPkt;
```

Figure 11. TLB CAM Packet

The matching condition looks like:

if((camReg[i].vpn2 & (~camReg[i].pageMask)) == (page0_i & (~camReg[i].pageMask)) &((camReg[i].asId == id) | (camReg[i].global))), where page0_i is the virtual address coming into the module. The virtual address is compared against each vpn2 entry of the CAM by masking out the bits required for page offset from both. Upon successful match of TLB entry, the corresponding bit of the match vector is set to one.

Figure 12 shows the CAM structure involving comparators. The figure has been simplified to show only one virtual address being fed to the TLB CAM. In actual the number of virtual

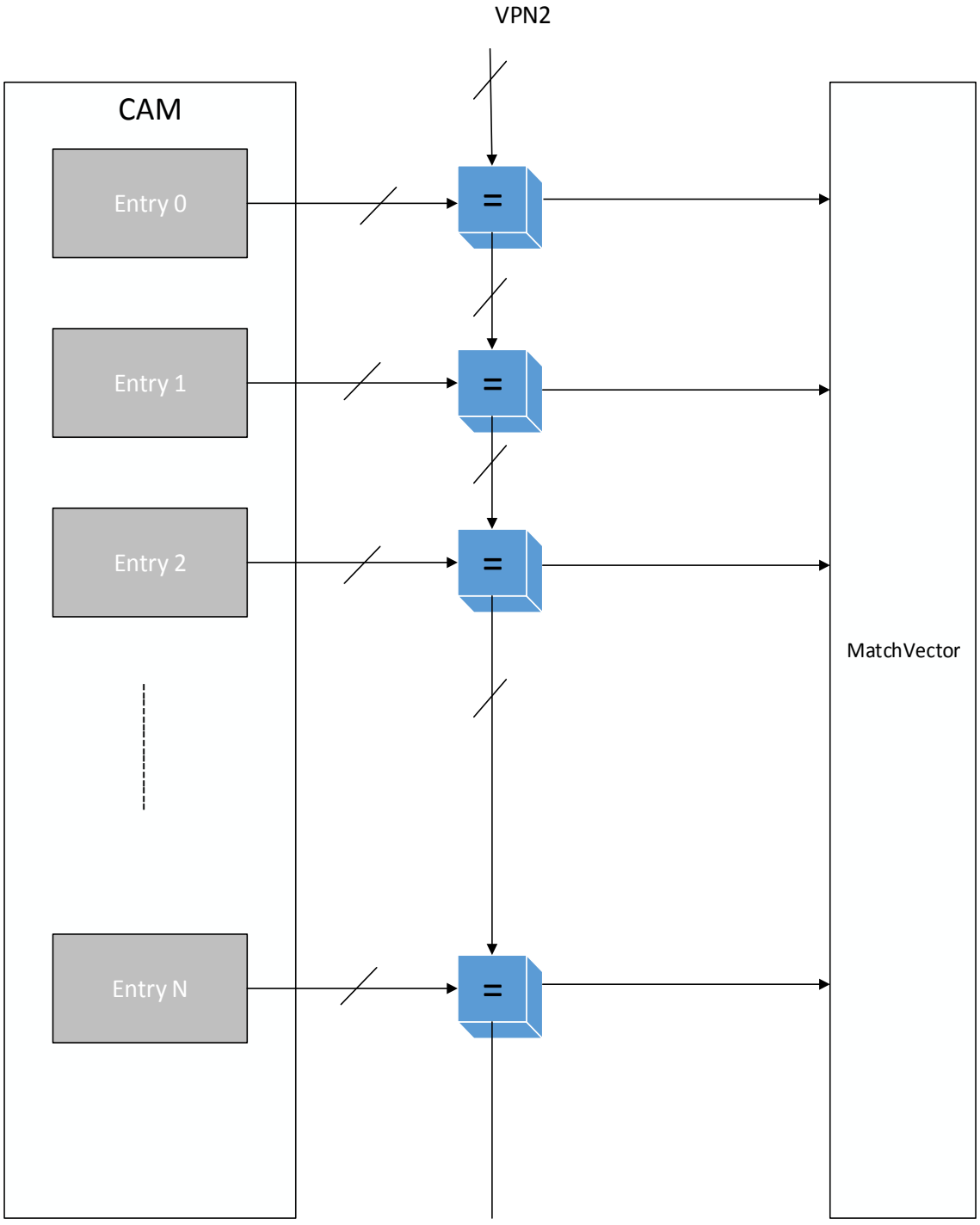

Figure 12. TLB CAM Structure

plus one. The plus one is for the case when load/store instruction in LSU access the address translation module for the virtual to physical address translation before accessing memory.

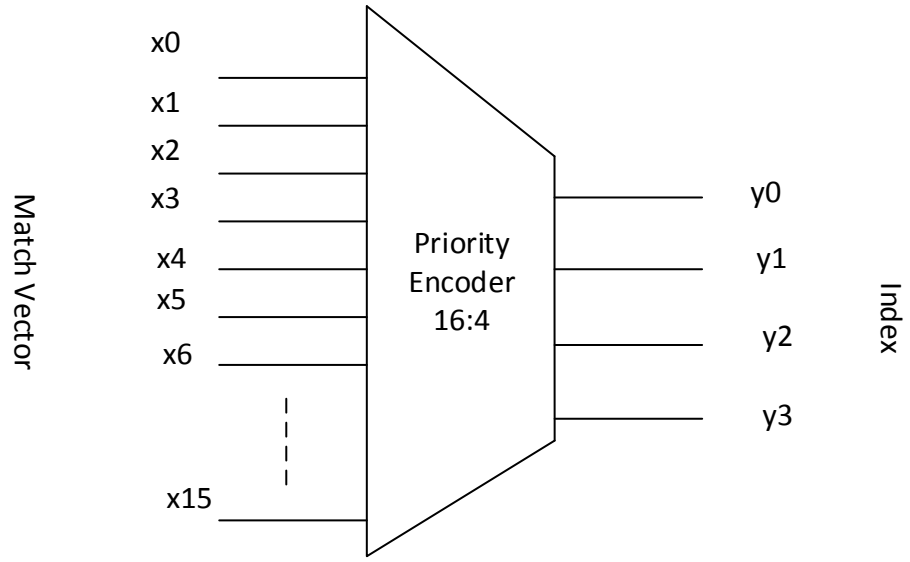

Figure 13. Priority Encoder for generating index into the TLB

The output of this module is the match vector (MatchVector) which is fed to the 16 X 4 priority encoder to find out the index of the entry that has successfully matched with the broadcasted virtual address.

TLB Payload RAM:

It is the RAM that contains the data or payload that is needed for virtual to physical address translation. The Figure 14 below shows the TLB payload packet which will be contained in each TLB entry.

```
typedef struct packed {
      logic [`SIZE_DATA-1:0]          pageMask;
      logic [`SIZE_DATA-1:0]          pageShift;
      logic [`SIZE_DATA-1:0][1:0]     pfn;
      logic [1:0]                     valid;
      logic                           dirty;
      logic [1:0]                     cache;
} TlbPayloadPkt;
```

Figure 14. TLB Payload Packet

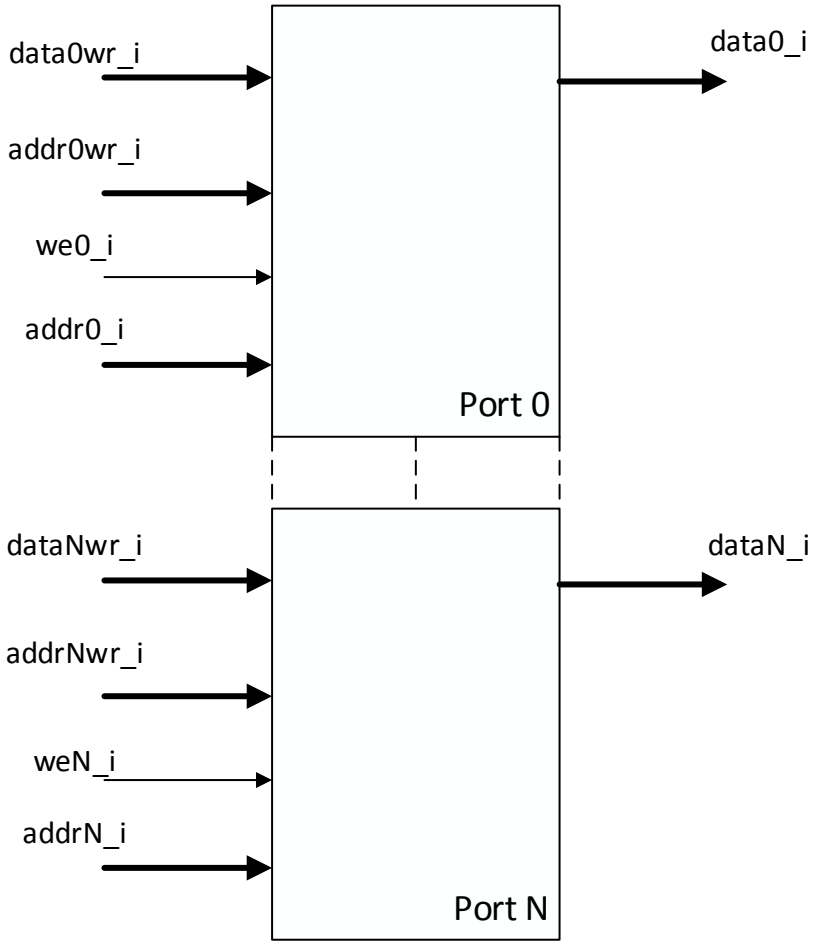

Figure 15. Multiported RAM for TLB Payload

Above shown Figure 15 shows the multi ported RAM which receives the index of the entry from the priority encoder and reads the payload from that location to get the physical address corresponding to that index. As each entry contains the even and odd pages, it reads either PFN0 or PFN1 depending upon whether the even/odd bit is set or not in the virtual address. The even /odd bit is the MSB bit of the LSB bits taken out from the virtual address for page offset.

Table 2. Physical Address Generation using TLB

Page Size	**Even/Odd Select**	**Physical Address Generation**
4 K Bytes	VA_{12}	$PFN_{(PABITS-1)-12..0} \mid\mid VA_{11..0}$
16K Bytes	VA_{14}	$PFN_{(PABITS-1)-12..2} \mid\mid VA_{13..0}$
64K Bytes	VA_{16}	$PFN_{(PABITS-1)-12..4} \mid\mid VA_{15..0}$

CHAPTER 5
EXCEPTION HANDLING UNIT

In this chapter we have explained the general process of handling exception and starting the handler with the right address. This unit is tightly coupled with the retire stage which signals this unit in an event of instruction reaching the head of active list with either exception flag or eret flag set.

5.1 Exception Handling Process

The normal execution of the program is interrupted which redirects the processor's program counter to the address where the exception handler resides in the memory. The cause of interruption can be due to the byproduct of instruction execution, such as Address Error exception, TLB miss exception, System Call, etc, or can be due to event that is not directly related to the execution of instructions, such as an external interrupt.

In an event of exceptions/interrupts, the following steps takes place:

- Processor saves certain COP 0 state like COP 0's EPC register gets loaded with PC at which execution will be restarted, BadVAddr register gets loaded with the failing address for exceptions like Address Error Exception, TLB Refill Exception, etc.

- It then signals fetch stage to redirect its Program Counter and provide it with the address of software exception handler.

The states to be saved and the address of the software exception handler depends upon the type of exception and the current state of the processor.

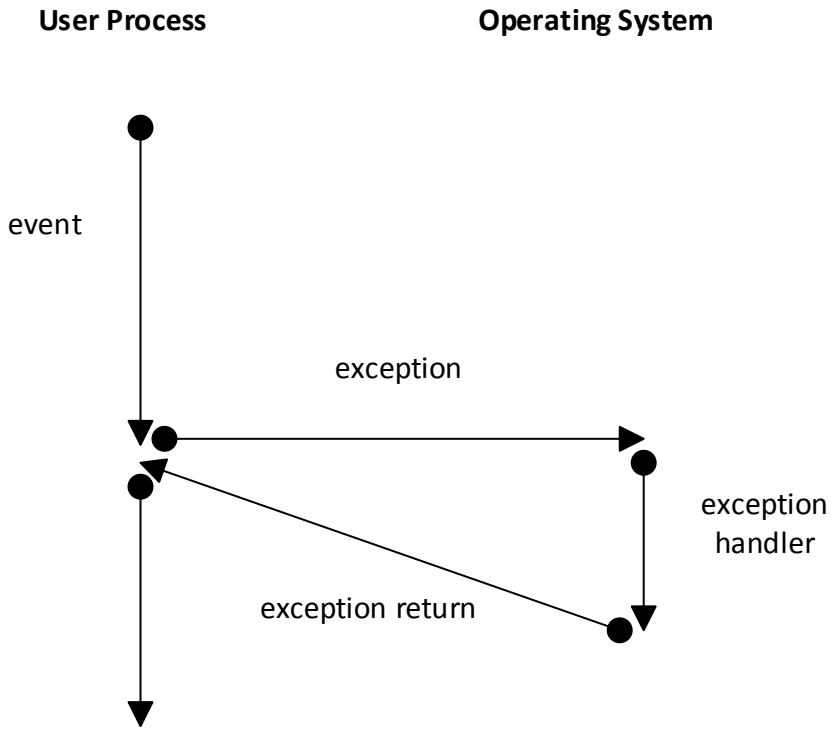

Figure 16. Exception Process

The Figure 16 shows the general process that is followed in case when exception has occurred.

The complete exception handling process has two aspects:

1. Starting the exception handler by redirecting Fetch Stage with the address of the software handler.

2. Returning from the exception when ERET instruction is encountered by redirecting Fetch Stage with the address stored in Exception Program Counter (EPC) register.

The Active List packet is modified to include extra fields:

- exception type
- eret flag

5.2 Exception Handler Address Generation

Address of any exception is the combination of vector base address and vector offset. In release 2 of MIPS architecture, software is allowed to specify the vector base address through COP 0 Exception vector base register (EBase) when Status register's BEV bit is zero, while in release 1 of the architecture the vector base address was fixed to 0x8000 0000. The exception vector offset depends on the type of exception.

Table 3. Exception handler base address

Exception	**Status BEV**	
	0	**1**
Reset, Soft Reset, NMI	0xBFC0 0000	
other	For release 1: 0x8000 0000 For release 2: EBase 31..12 || 0x000 *EBase 31..30 have fixed value 0b10	0xBFC0 0200

Table 4. Exception vector offset

Exception	**Vector Offset**
TLB Refill, EXL = 0	0x000
General Exception	0x180
Interrupt	0x200
Reset, Soft Reset, NMI	None

CHAPTER 6
EXECUTION UNIT

This chapter includes the detail description about the execution machinery of COP 0 and different strategies for handling COP 0 instructions in the pipeline. One strategy is better than the other in terms of executing in parallel COP 0 instructions with CPU instructions but will be more complex to implement in design.

6.1 Instruction Execution Machinery

This unit is the main execution machinery of COP 0 instructions which involves strategies deployed to handle them correctly with respect to out of order back end pipeline stages of processor.

MFC0

The contents of the COP 0 register which is specified by the combination of rd and sel is moved to the general register specified by rt field in the instruction.

Operation:

data = RF[rd,sel]

GPR[rt] = data

MTC0

The contents of the general register specified by rt field of the instruction is moved to the COP 0 register which is specified by combination of rd and sel.

Operation:

data = GPR[rt]

RF[rd,sel] = data

TLBR

The instruction TLBR causes the contents of the TLB entry pointed by the Index register to get loaded into the EntryHi, EntryLo1, EntryLo0 and PageMask register.

Operation:

RF[EntryLo0] = TLB[RF[index]].PFN0

RF[EntryLo1] = TLB[RF[index]].PFN1

RF[PageMask] = TLB[RF[index]].Mask

RF[EntryHi] = TLB[RF[index]].VPN2

TLBW

The TLB entry pointed by Random register if the instruction is TLBWR or Index register if the instruction is TLBWI gets written with the contents of the EntryHi, EntryLo1, EntryLo0 and PageMask registers.

Operation:

TLB[RF[index]].PFN0 = RF[EntryLo0]

TLB[RF[index]].PFN1 = RF[EntryLo1]

TLB[RF[index]].Mask = RF[PageMask]

TLB[RF[index]].VPN2 = RF[EntryHi]

TLBP

The TLB probe instruction causes the Index register to get loaded with the address of the TLB entry whose contents match the contents of the EntryHi register.

Operation:

```
for i in 0…TLBEntries-1
    if( (TLB[i].VPN2 & !TLB[i].Mask) == (RF[EntryHi] & !TLB[i].Mask) )
    RF[index] = i
     endif
endfor
```

6.2 Strategies for implementing COP 0 instructions in OOO pipeline context

We have proposed here four different strategies for handling COP 0 instructions in the processor pipeline and have implemented the simplest one.

1. Fully Serializing the COP 0 Write Instructions: This is the simplest and the least complex logic to handle these instructions. We let the instructions that are reading from

COP 0 state to issue and execute out of order normally. We heavily serialize the execution of instructions that write the state of COP 0. Whenever the write instructions (TLBW, MTCO) are encountered in the dispatch packet, we stall the frontend and partially dispatch the bundle by making the instructions before the first COP0 write instructions as valid and inserting NOPs in place of COP0 write instruction and the instructions after it in the dispatch packet. We then wait for the Active List to be empty which means letting all the instructions before the COP0 write instructions to drain out from the pipeline. After all the instructions drain out from the backend of the pipeline and active list becomes empty we then dispatch the COP0 write instruction into the issue queue and wait for it to drain out of pipeline, after which we flush the front end, recover RMT from AMT and redirect the fetch unit to the PC just after the COP0 write instruction. The squashing and recovering of instructions after COP 0 write is necessary because if there are address translation side-effects due to the COP 0 write, then instructions fetched speculatively after it are potentially incorrectly translated. This way we have ensured that instruction reading COP0 registers can issue and complete out of order as its destination (CPU register) will get renamed and all register dependencies with respect to this register will be taken care of by the existing renaming machinery. There is no overlapping possible with the COP 0 instruction with this technique.

2. Coarse grain Scoreboard Technique: This technique uses the single bit indicating whether or not there is another COP0 instruction in the Issue Queue. If the instruction packet in the dispatch stage contains COP0 instruction, check for another COP0 instruction in the issue queue. If there is a pending COP0 instruction in the issue queue we stall the frontend and allow the instructions that are before COP0 instruction in the dispatch packet to issue. After the COP0 instruction gets issued from the issue queue, we dispatch the remaining instructions in the bundle including COP0 instruction to the issue queue and un-stall the frontend. The overlapping of CPU instructions with single COP 0 instruction is possible with this technique, but not with multiple COP 0 instructions.

3. Fine grain Scoreboard Technique: This technique maintains one bit per COP0 register. If the instruction packet in the dispatch stage contains COP0 instruction, check for the COP0 instruction in the issue queue that has either same destination or source register as the COP 0 instruction in the dispatch stage. If there is a pending COP0 instruction in the issue queue using same register, we stall the frontend and allow the instructions that are before COP0 instruction in the dispatch packet to issue. After the COP0 instruction gets issued from the issue queue, we dispatch the remaining instructions in the bundle including COP0 instruction to the issue queue and un-stall the frontend. The overlapping of CPU instructions and multiple COP 0 instructions to different registers is possible with this technique.

4. Renaming COP 0 instructions: By renaming the COP 0 registers in the rename stage, we can remove false dependencies between the COP 0 instructions. The zest behind not renaming COP registers is kind of hard to explain. We choose not to rename them, as the fetch stage will have different view of address translation for dynamic instruction stream which doesn't make sense. In addition to the overlapping of instructions provided by Fine grain approach, this technique allows the overlapping of COP 0 instructions to the same registers.

Table 5. Strategies for handling COP 0 Instructions

Type of State Update	**Solution**	**What can overlap?**
Global: affects address translations	fully serialize	nothing
Local: does not affect address translations	fully serialize	nothing
	coarse-grain scoreboard	CPU instructions + single COP 0 instruction
	fine-grain scoreboard	+ multiple COP 0 instructions as long as they write to different registers
	Renaming	+ COP 0 instructions to same registers

Strategies depends upon the type of COP 0 state being updated i.e Global update that affects the address translation, Local update that doesn't affect address translation. We handle the TLB instructions and COP 0 read/write instructions bit differently as TLB instructions cause global update of COP 0 state. Hence, TLB instructions are handled only using fully serializing technique but strategies to handle COP 0 read/write instructions that only causes local state update of COP 0 can be optimized for performance. The different strategies for handling COP 0 instructions and the overlapping possible with each of them depending upon the type of COP 0 state update is shown in Table 5.

CHAPTER 7

PIPELINE STAGES AFFECTED

In this chapter we have explained the stages that are affected and the changes that have taken place within those stages due to the addition of COP 0 module in FabScalar design.

7.1 Fetch Stage

The function of this pipeline stage is to fetch new instructions from memory every cycle and predict the next PC (the address from which it should fetch the next instruction) of the instruction. Since this stage interacts with memory hierarchy, it needs to go through COP 0's Address Translation unit for translating its virtual Program Counter (PC) into physical Program Counter before it can access memory hierarchy. The number of translations required per cycle depends upon the fetch width of the processor. The process of address translation is explained in this Chapter 4.

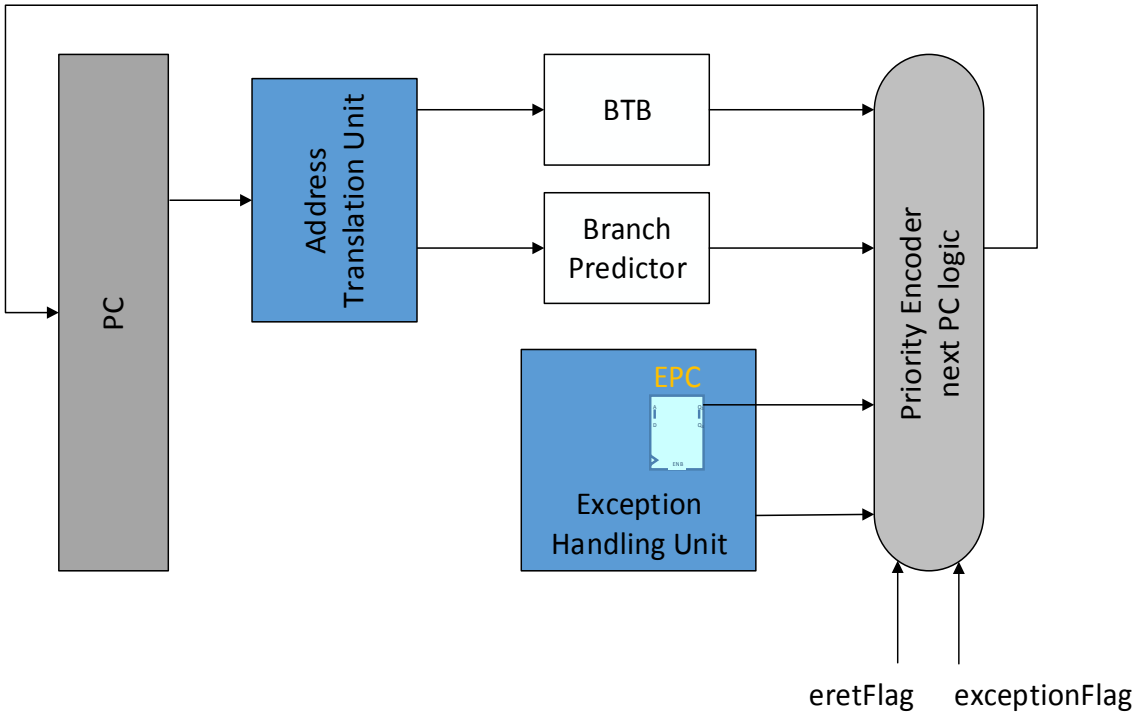

Figure 17. Next PC logic of Fetch-1 Stage

The existing next PC logic is modified such that it will redirect it's program counter to the address stored in COP 0 Exception Program Counter (EPC) register when the COP 0 instruction is encountered in execute stage by setting it's priority as highest among other causes that redirects program counter/fetch unit with different address rather than the next sequential address. The EPC register is written with the program counter of the instruction that has caused the exception before redirecting the processor to the exception handler address. In case the exception has occurred which is detected at the retire stage at the time of commit it will override the next PC logic and will redirect the processor's PC to the exception handler address.

The exception handler's address will depend upon whether the processor is running in normal mode or the bootstrap mode. The detail about the exception handler mechanism is explained in Chapter 5, section 5.1.

7.2 Decode Stage

This stage in the pipeline of the processor is responsible for the encoding of incoming instruction binaries from the Fetch-2 stage. Earlier, the FabScalar-MIPS was design to implement integer instructions and floating point instructions and we extended this existing design with COP 0 instructions as per MIPS instruction set architecture [4].

The basic features of COP 0 instructions semantics are as below while more detail information pertaining to this can be found in MIPS ISA manual [4]:

- There are 32 COP 0 architectural registers, each 32 bit in width, with important ones include EntryHi, EntryLo, Status, Cause, Config, and EPC.
- COP 0 instructions has either one source register as general CPU register and one destination register as COP0 or vice-versa. The TLB instructions have implicit COP0 register as source and destination register.

The decode packet was modified to include the additional information pertaining to COP 0 instruction.

```
`ifdef COPROC0
    logic [`SIZE_CP0_LOGICAL_LOG-1:0]    cp0Src;
    logic [`SIZE_CP0_LOGICAL_LOG-1:0]    cp0SrcValid;

    logic [`SIZE_CP0_LOGICAL_LOG-1:0]    cp0Dest;
    logic                                cp0DestValid;
`endif
```

Figure 18. Decode Packet

Move Instructions:

Instructions involving data transfer between general CPU registers and COP0 registers. The table below shows the move instructions involving COP0 register as specified by MIPS ISA [4]:

Table 6. Types of COP 0 Move Instructions

Mnemonic	Instruction
MFC0	Move from coprocessor -0 register
MTC0	Move to COP 0 register

The Figure 19 below shows the semantics of the COP0 MFC0 instruction.

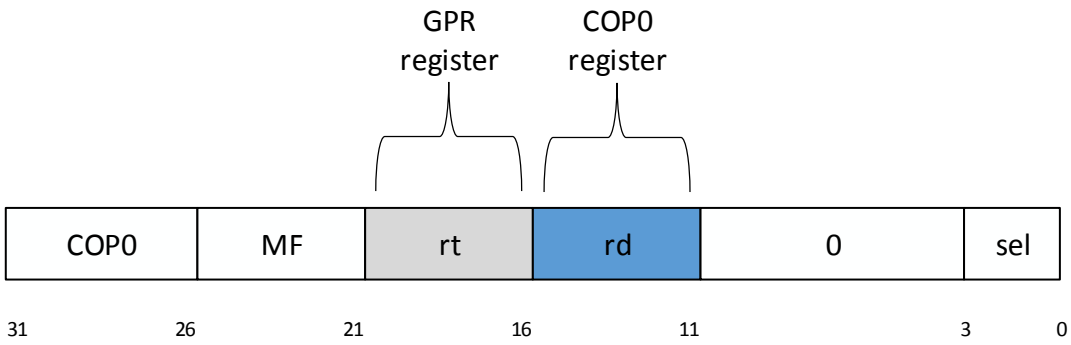

Figure 19. MFC0 Instruction Semantics

The Figure 20 below shows the semantics of the COP0 MTC0 instruction.

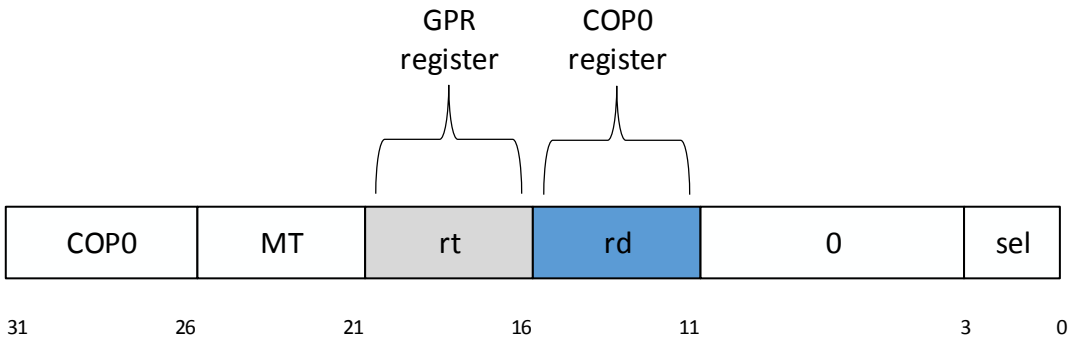

Figure 20. MTC0 Instruction Semantics

TLB Instructions:

The table below shows the TLB instructions we implemented as specified by MIPS ISA [4]:

Table 7. TLB Instructions

Mnemonic	Instruction
TLBWR	Write a TLB entry indexed by COP0 Random register
TLBWI	Write a TLB entry indexed by COP0 Index register
TLBR	Read a TLB entry pointed by Index register
TLBP	Probe TLB to find a matching entry

The TLB instructions doesn't have explicit source or destination register encoded in the instruction binary, instead it has implicit registers as its source and destination. Figure 21 below shows the semantics of TLB instructions.

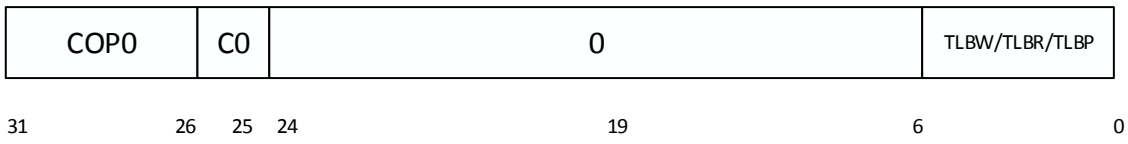

Figure 21. TLB Instruction Semantics

The rename of general purpose register is done using the existing rename machinery and mechanism. We decided not to rename the COP 0 registers as some of these registers are involved in address translation process and renaming them would have not been the clear way

of implementing the design especially for the fetch stage which will be accessing the COP 0 registers every cycle for generating physical address to fetch instructions from memory.

7.3 Dispatch Stage

The dispatch stage basically separates the in order front end and out of order backend pipeline stages of OoO processors. This stage is responsible for dispatching the instructions into the Integer Issue Queue, Floating Point Issue Queue, Active List/ROB, Load and Store Queues after checking the availability for required space for the dispatch bundle. If the required space is not available in either of the structure, the dispatch stage is stalled till the required space becomes available again.

In the decode stage we assign instruction type to an instruction depending upon its opcode which will be used by execution pipe scheduler to pre assign the execution lane number to the instruction which enables the clear and simple implementation of lane arbitration logic and select logic. We set the instruction type of COP 0 instruction such that they always assigned to execute and arbitrate for LSU lane as the COP 0 module resides in LSU execution lane.

For integer and floating point instructions we take care that they maintain all dependencies i.e WAW, WAR and RAW dependencies by renaming their destination register to a physical register of physical register file which is bigger than logical register file. This way we removes false dependencies between instructions and also ensures that even if they execute out of order with respect to each other they will execute by taking the correct value of their source operands.

We implemented the strategy to ensure that COP 0 instructions gets executed correctly with respect to out of order back end semantics of a processor.

We implemented Finite State Machine in the dispatch stage to heavily serialize the execution of COP 0 instructions that changes the state of COP 0. We let the MFC0 instruction (that only reads the COP 0 register and doesn't tries to change its state) to execute normally without implementing special strategy for handling it. The MFC0 instruction has one COP 0 register as source and one General purpose register as destination. As its destination register is GPR register, it will get renamed to a physical register in the rename stage and will execute correctly in out of order backend by writing the result first to the physical destination register and then committing the physical register mapping to AMT when MFC0 instruction reaches the head of Active list.

However for other COP 0 instructions that changes the state of COP 0 either explicitly through MTC0 instructions or implicitly through TLB instructions, special strategy is implemented to handle them correctly. The Finite State Machine that is designed to heavily serialize these type of instructions is shown in Figure 22.

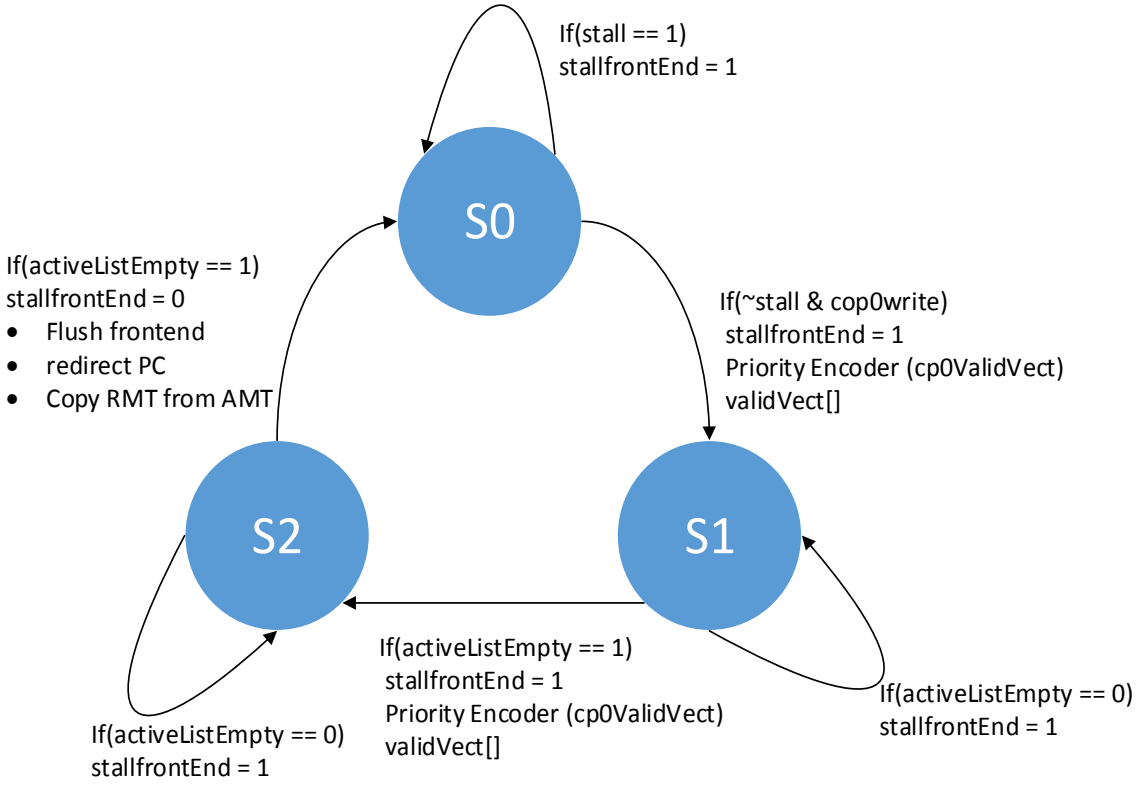

Figure 22. State Machine designed for dispatch stage

Working of FSM:

We designed three state FSM: S0, S1 and S2. The FSM will be set to the initial state S0 upon observing the reset signal. The stall signal in the dispatch stage gets high if either issue queue, load queue, store queue or active list doesn't have require number of free entries for number of instructions equal to the dispatch width of the processor core. The machine will remain in state S0 till the stall signal remains high stalling the dispatch for that amount of time. When stall signal becomes low i.e when the required number of entries which is equal to the dispatch

width becomes available in each of the queue to accommodate instructions in the dispatch bundle and the cop0write[2] signal is low indicating that there is no instructions in the dispatch bundle that will change the state of COP 0, we will set the valid bit for each instruction in the dispatch bundle. The cop0write signal goes high whenever there is a COP 0 instruction in the bundle that tries to change the state of COP 0. When the cop0write instruction is encountered in the dispatch bundle, the machine goes through two step process which are explained below. If the stall signal is low and the cop0write signal is high, the machine will get into the serialize mode and will.

Step 1:

When the cop0write signal gets high and the stall signal is low the state machine will advance its state to S1. The priority encoder is employed to generate the valid vector to make the instructions before the cop0write instruction in the bundle valid and make the instructions after the cop0write including itself as invalid. The Figure 23 below shows how this process works. The dispatch bundle as shown in Figure 23.a contains add, sub, div and tlbr as a last instruction in the dispatch bundle. After priority encoding, the instructions add, sub and div will become valid and tlbr as invalid as shown in Figure 23.b**.**

[2] We are referring to cop0write instruction as any COP 0 Instruction that changes its architectural state.

Figure 23. a) Dispatch packet containing TLB Read Instruction. b) Dispatch packet after making TLB Read as invalid using priority encoder.

The front end is stalled and the bundle containing nops is now dispatched to the issue queue and the state is advanced to state S1. As the frontend is stalled, the ren2dispatch register will store the same packet containing the cop0write instruction. The state machine will remain in state S1 till all the instructions in the backend of the processor drains out of the pipeline making the active list empty which will cause the activeListEmpty signal to go high.

Step 2:

When the activeListEmpty signal becomes high, we dispatch the cop0write instruction by making it as valid and all other instructions as invalid/nop as shown in Figure 24, the front end is stalled and the state machine is advanced to state S2 and will remain in this state till the activelistEmpty signal becomes high again or the cop0write instructions drains out of the pipeline.

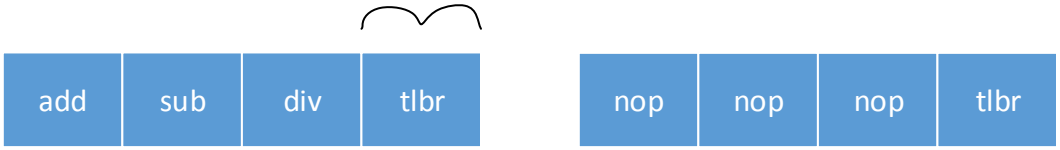

Figure 24. a) Original dispatch packet b) Dispatch packet after inserting nops with TLBR

When the active list becomes empty the state of the machine is advanced back to state S0, the frontend is flushed, then unstalled, redirect the program counter and the RMT is restored by copying AMT to RMT.

The Issue Queue holds the instructions till their source operands are not ready or the execution lane for which the instruction is selected to execute by the execution pipe scheduler becomes available.

7.4 Retire Stage

There are two aspects to discuss about Retire Stage, i.e other stages interaction with it and its interaction with COP 0's Exception handling unit.

1. Other Stages -> Retire Stage interactions: The exception can be set either in frontend pipeline stages or in execution stage. If an exception is detected in the frontend pipeline stage, exception information flows with the instruction to the dispatch stage. The active list's exception bit and exception type are set when the instructions is pushed onto the active list. If an exception is set in the execution stage, exception bit and exception type gets set in the instruction's active list entry. When ERET instruction is executed in the execution stage, eret flag is set in the instruction's Active list entry.
2. Retire Stage-> COP 0 interaction: When posted exception reaches head of Active List, the pipeline gets flushed and it signals COP 0 Exception Handling Unit with a) exception, b) exception type which redirects the fetch stage with the address of

exception handler. In case of instruction that has eret flag set reaches the head of Active List, pipeline gets flushed and it signals COP 0 about ERET.

CHAPTER 8

COP 0 within LSU LANE

Most of the COP 0's logic structure and its architecture state resides in the LSU lane. From the name we can make out that this is the lane in which store instructions and load instructions gets executed but as this lane contains COP 0 module we modified the design such that execution pipe scheduler in the dispatch stage sets the LSU lane for COP 0 instructions to schedule or issue in by the issue stage select/lane arbitration logic. Loads and stores also gets dispatch to Load queue and Store queue by the dispatch stage.

The register read stage contains the integer physical register file and floating point register file that holds the instruction results. The COP 0 register file that contains 32 registers is added. The COP 0 register file is implemented as an SRAM that has. Loads and stores reads and writes to either integer or floating point physical register file while only COP 0 instructions accesses the COP 0 register file to read the value.

The address generation unit in the execute stage is responsible for calculating the memory addresses for load and store instructions. After the memory address is computed by the AGEN unit, the memory type packet which contains necessary information about the instruction is sent to the LSU unit through Agen-lsu latch. The FabScalar template has been changed such that now after AGEN computes the address which is virtual, the memory packet first goes through the COP 0's address translation unit which translates the virtual memory address into physical address and after updating the memory packet's address field

it sends the memory packet to LSU unit. The LSU unit is responsible for load disambiguation and store to load forwarding. The loads accesses the LSU unit for address dependency check while the store accesses the store queue to put its address and the store value, it also broadcasts its address to any disambiguation stalled loads. A store value remains in the store queue and doesn't commit to the memory until the store reaches the head of active list.

In write back stage the result of the load instruction gets written into the physical register file and it also broadcasts the load's destination tag along with the value on the bypass which is used by the instructions in the execute stage that has its source matched with the broadcasted tag. If the instruction has the COP 0 register as its destination then the result gets directly written into the COP 0 register file by bypassing all other write back stage logic.

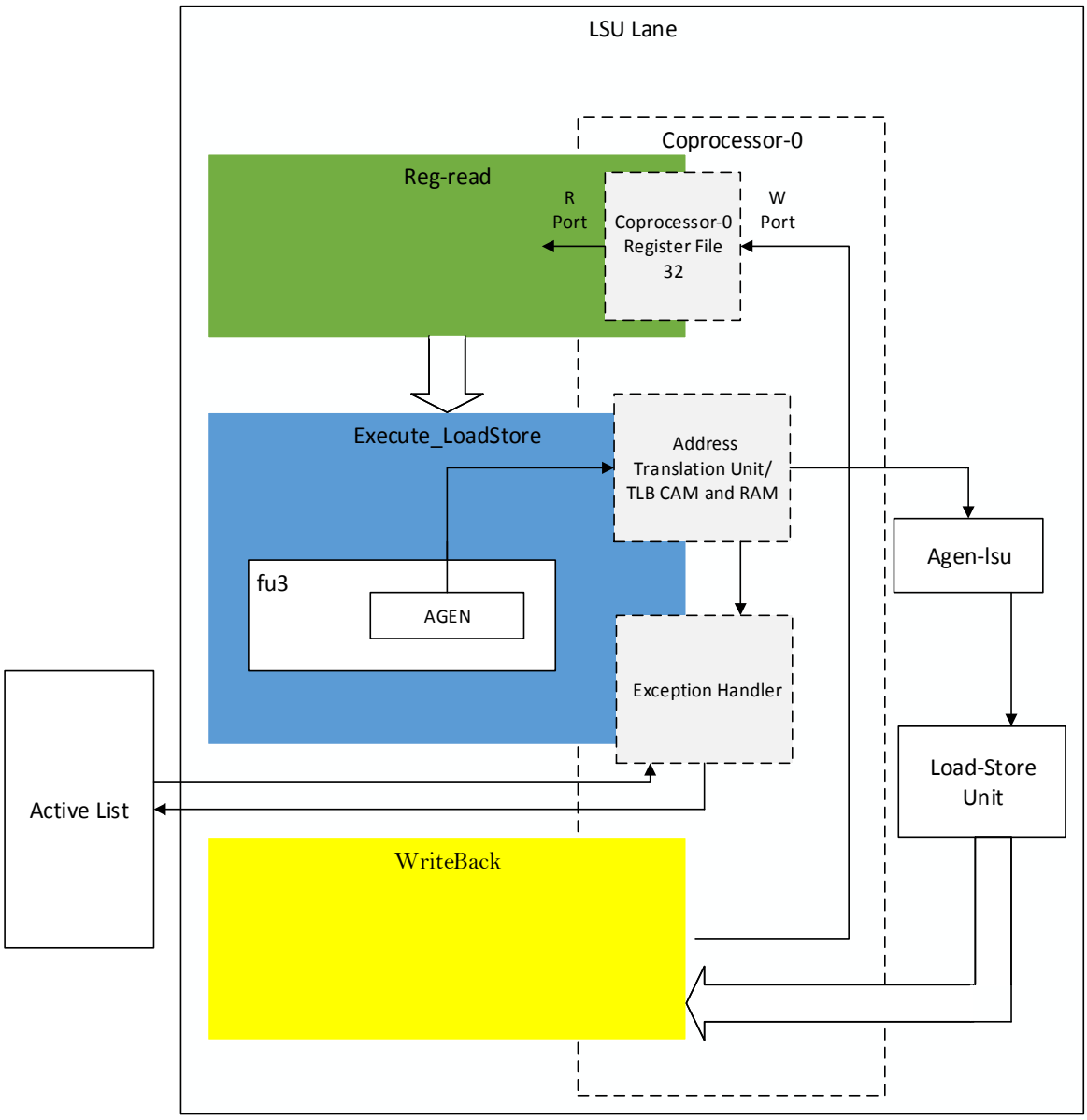

Figure 25. Load-Store Lane

The new port connection is made in the active list module with the name exceptionType for interfacing the retire stage with the COP 0's exception handling module. In an event of an

exception or interrupt, the active list entry of the instruction is written with the type of exception along with the exception flag using by COP 0 module. When the instruction that caused the exception reaches the head of active list, the exception type is send to the COP 0 module which redirects the fetch unit to the address of the exception handler depending upon the type of exception and saves the state for some of the COP 0 registers. The instructions commit from the head of active list and updates the AMT with their destination physical register mapping if the instruction's destination is valid. There are separate AMT for integer and floating point instruction.

When the instruction commit at the retire stage, the testbench calls the checker function which verifies the program counter and the destination result if valid. The checker currently checks for the floating point destination values for floating point instructions and integer destination values for integer instructions. It is modified to include the checking mechanism for the COP 0 instructions that may have COP 0 register as destination.

CHAPTER 9
ANALYSIS AND RESULTS

In this chapter we have presented the methodology we followed for performing functional verification of the RTL simulator and also gathered performance numbers for various runs. The performance is measured in instructions per cycle (IPC).

9.1 Simulation Methodology

The Figure 26 shows the co-simulation setup that is used to verify the results of RTL simulator with the results of C++ functional simulator. The functional simulator is also used to initiate the processor state and the memory of RTL simulator using DPI calls from the testbench.

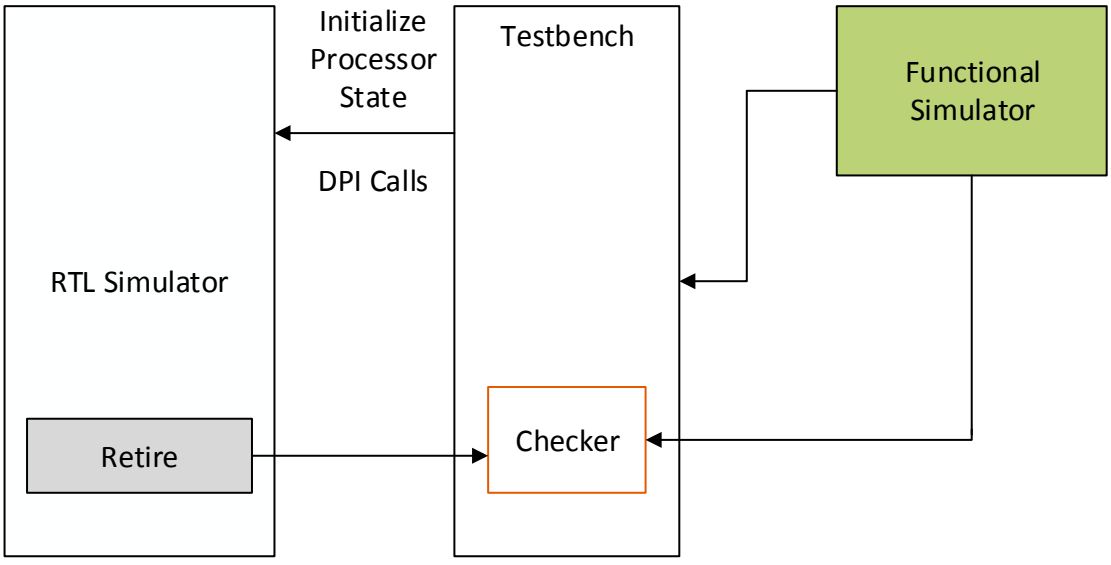

Figure 26. RTL and C++ co-simulation environment.

At the retire stage during instruction commit, the checker function is called from the testbench using DPI for verifying RTL simulator's PC and the result with the corresponding functional simulator output.

After we successfully booted Linux image on C++ functional simulator which ensured the correct functionality of address translation and exception handling functions in C++, it was used as a golden reference in FabScalar's co-simulation infrastructure. The functional correctness of the RTL was progressively verified in three steps and each step builds upon the previous one:

1. Targeted verification using synthetic kernels: In this we created micro kernels which were targeted to test specific COP 0 functionalities. The instructions that were targeted and stressed were move instructions which ensured that the correct data is transferring between COP 0 register file and General purpose register file.

Table 8. Instructions used to create micro kernels

Functionality	Assembly instruction	correctness
MTC0	mtc0 $k1, $12	✓
MFC0	mfc0 $k0, $14	✓
TLBWI	tlbwi	✓

2. Full COP 0 verification using Linux kernel image: We ran Linux kernel image on FabScalar and was able to fully boot it on C++ functional simulator and partially on RTL simulator (79,923 instructions). Co-simulation environment of our toolset ensures the functional correctness of 79,923 instructions that committed on RTL simulator.
3. System verification with Linux applications: Although the Linux has not completely booted on RTL simulator but once it's successfully booted, spec2000 benchmarks can run as processes in Linux to verify correct functionality of Linux process and virtual memory management with our COP 0 design.

9.2 Performance Results

Once the RTL was functionally verified, performance data was collected and analyzed by measuring the IPC for first two runs i.e targeted synthetic micro kernels and Linux kernel. The graph in Figure 27 shows the Instruction per cycle of a Linux kernel and the micro kernels which we created to test for targeted functionality of COP 0 design.

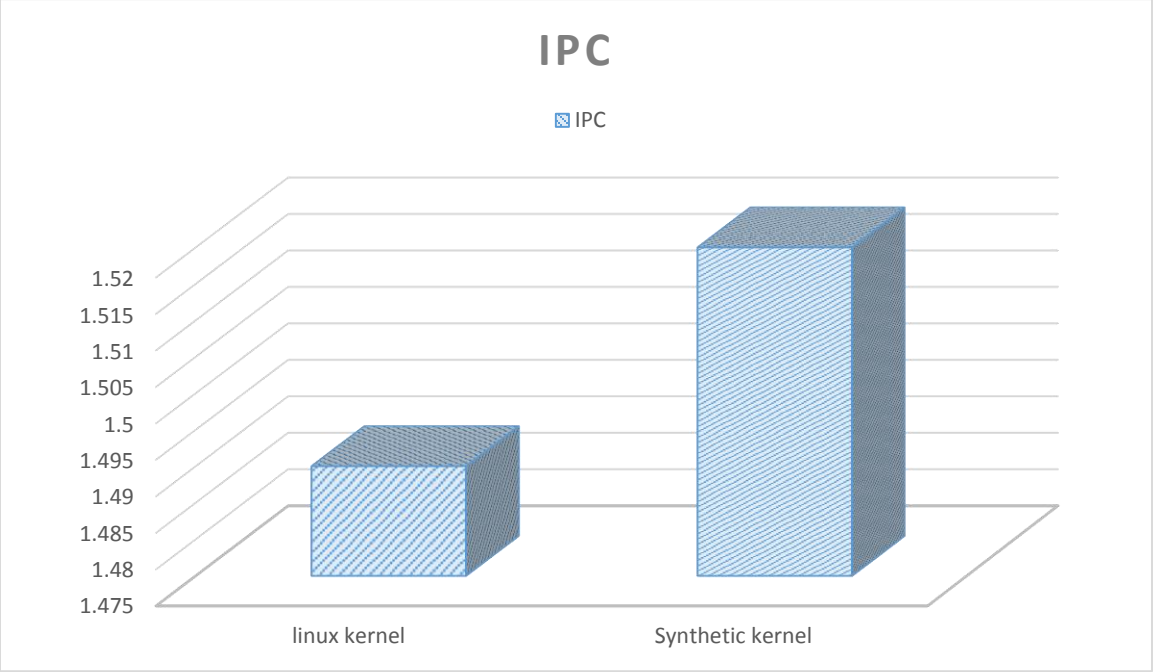

Figure 27. Graph showing IPC of Linux kernel and micro Kernel.

9.3 Instruction Breakdown

The pie chart shown in Figure 28 shows the percentage of COP0 instructions over a total of 340471207 instructions that booted on functional simulator and Figure 29 shows the percentage of COP 0 instructions over a total of 79923 instructions that booted on RTL simulator. We have further break down the COP 0 instructions to show the distribution of the MFC0 and MTC0 instructions and which instruction out of the two is stressed more by running Linux image on functional simulator. The pie chart in Figure 30 shows the COP 0 instruction distribution which shows that MFC0 instructions are slightly more stressed than MTC0 instructions.

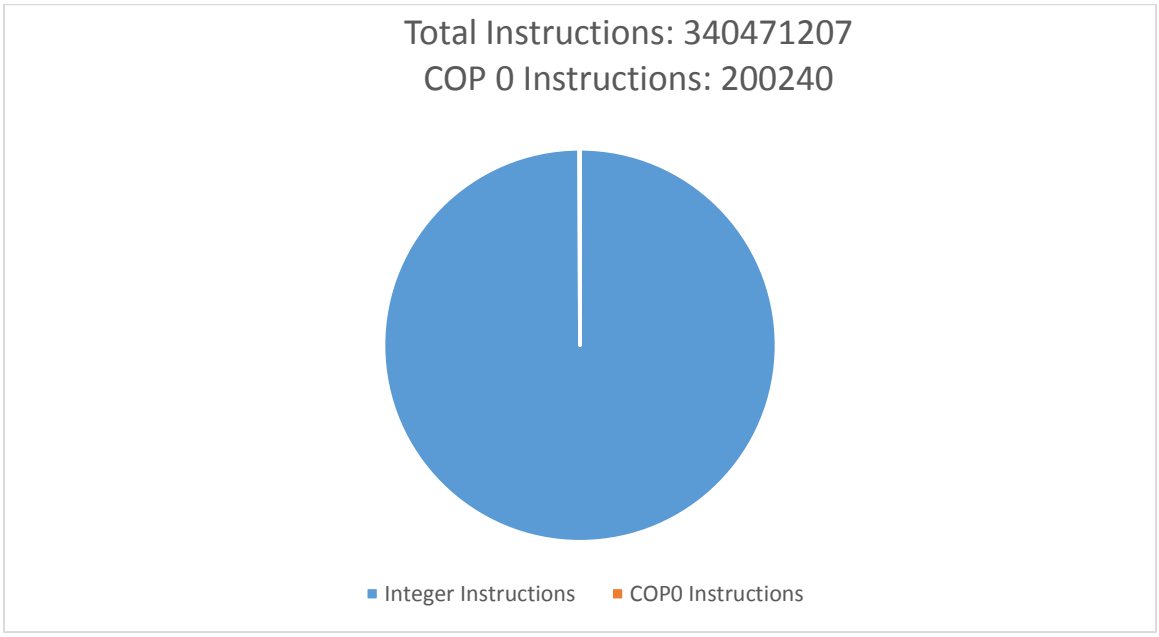

Figure 28. Linux kernel instruction breakdown for functional simulator.

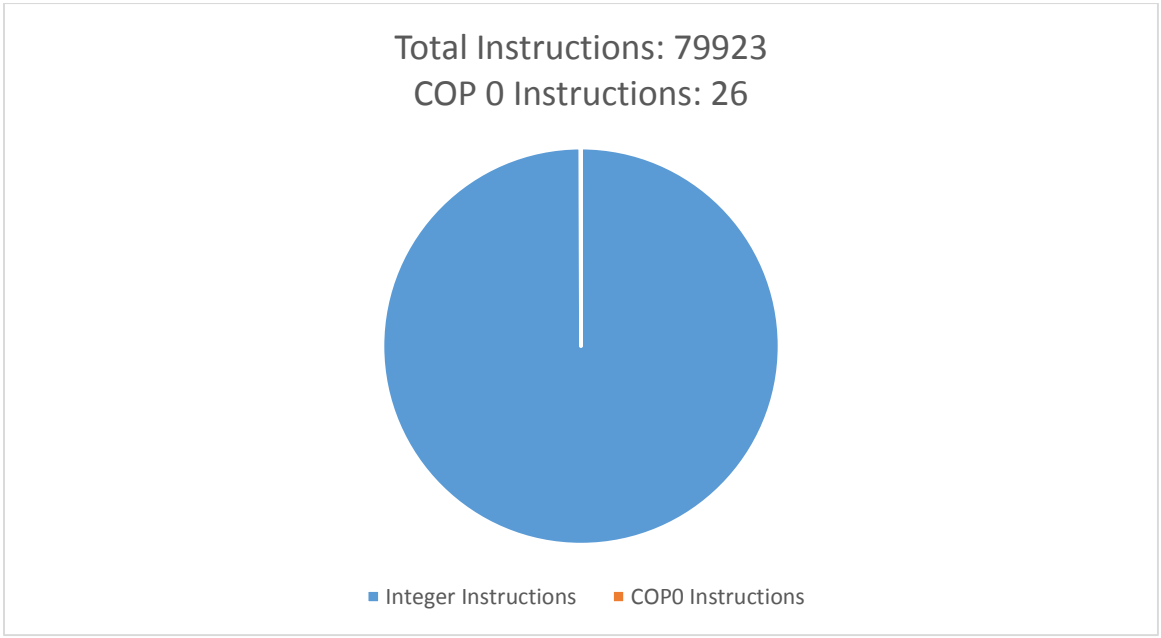

Figure 29. Linux Kernel instruction breakdown for RTL simulator

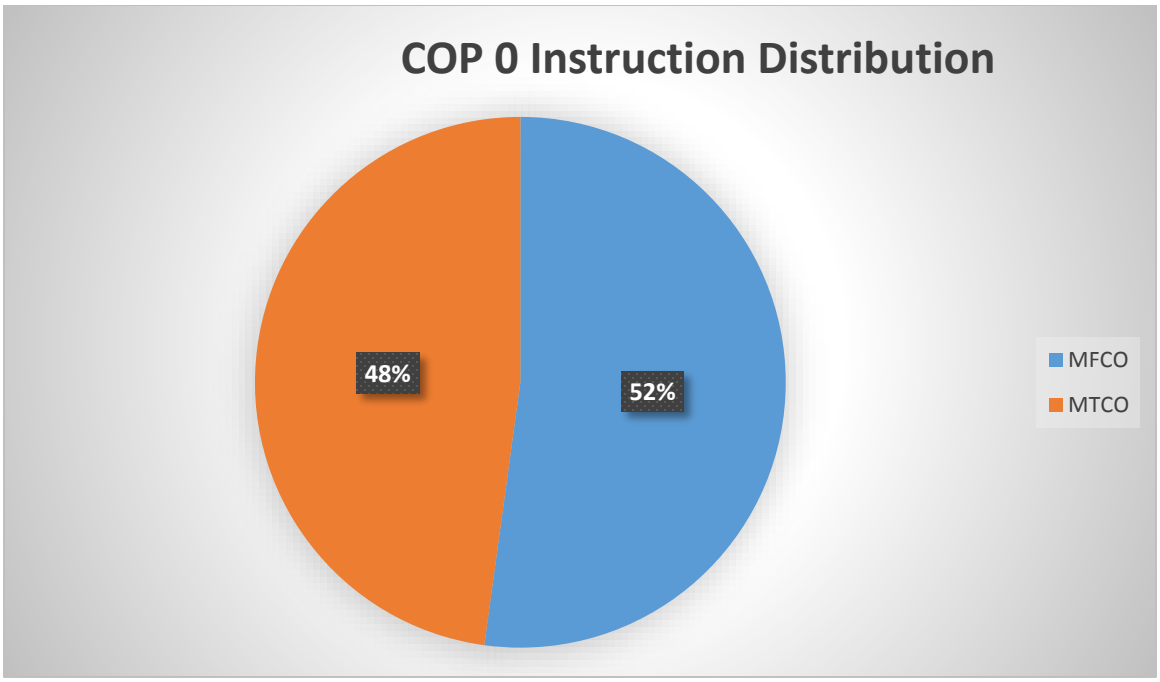

Figure 30. COP 0 instruction distribution.

CHAPTER 10
SUMMARY AND FUTURE WORK

This thesis presents an interesting work of implementing Coprocessor-0 (COP 0) ISA which is subset of MIPS ISA. The FabScalar toolset is extended with COP 0 and serial device support for booting Linux on it.

FabScalar is a toolset that allows automatic generation and verification of synthesizable RTL cores with varied configurations. It is mainly aimed at vastly simplifying the design of heterogeneous multi-core systems which possess several advantages over homogeneous multi-cores. Harnessing the full potential of a heterogeneous system requires both a clever architecture and substantial system software support. Although significant amount of progress has been made in architecting such a system much needs to be accomplished on the system software front. The FabScalar MIPS design, with support for MIPS COP 0, is an attempt at providing a platform for such research. COP 0 empowers FabScalar designs to run preemptive, multitasking operating systems, such as Linux, by providing support for virtual memory and mechanism to handle exceptions and interrupts. This allows for research in the diverse fields of power management, thread and process scheduling and SMT in a real heterogeneous system. In this work, we architect and implement the COP 0 in the FabScalar MIPS design. This involves modifying the MIPS functional simulator and RTL, emulating devices and improving co-simulation framework. The results of designing such a system are discussed along with challenges faced and solutions reached while designing it.

The first step towards the future work should be getting Linux booted on RTL simulator. Performance of benchmarks running as Linux processes can be compared against the same benchmarks running as stand-alone executables on the same design without COP 0 support. More performance analysis such as studying the impact of exceptions on the core performance can be done further to see some interesting and important results. After synthesizing our RTL design that has COP 0 module integrated, we can calculate the impact on cycle time, power and area due to the changes made in the FabScalar template.

REFERENCES

[1] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template", Proceedings of the 38th Annual International Symposium on Computer Architecture. ACM, 2011, pp. 11. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000067

[2] Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise "SimMips A MIPS System Simulator".

[3] MIPS® Architecture For Programmers Volume III-A: Introduction to the MIPS32® Architecture.

[4] MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture.

[5] MIPS® Architecture For Programmers Volume II-A: Introduction to the MIPS32® Architecture.

[6] Linux Porting Guide. [Online]. Available: http://www.embedded.com/design/embedded/4023297/Linux-Porting-Guide

[7] SimpleScalar Simulator Toolset. [Online]. Available: http://www.simplescalar.com/