Exception Handling for 5-Stage Pipeline Micro-Architecture

BY

ARTHUR PUAN CHOK HO

Supervised by

Mr. Mok Kai Ming

A Report

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

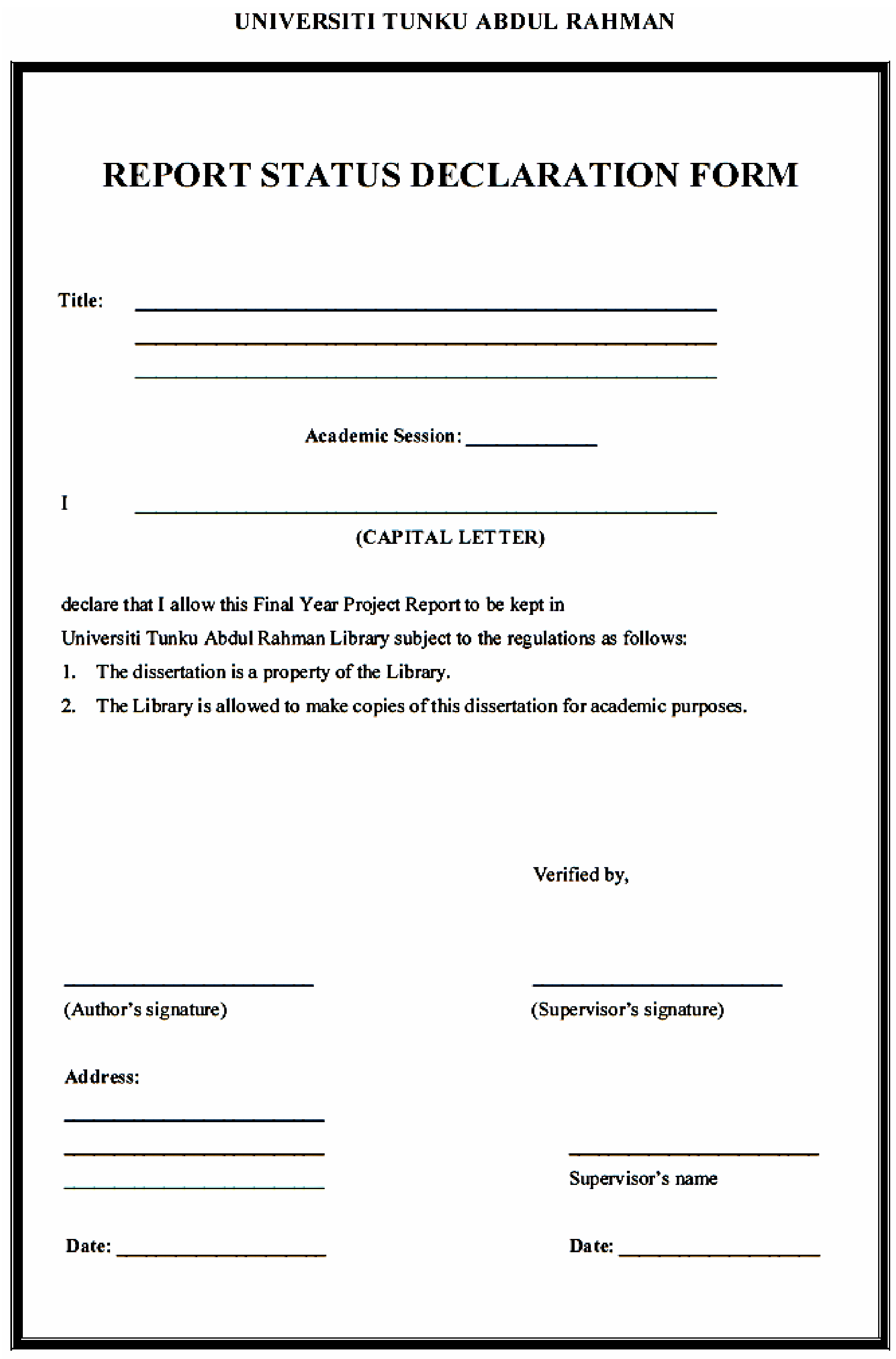
for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology
(Perak Campus)

JUNE 2015



UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: ____
____
____

Academic Session: ____________

I ____
(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

_________________________	_________________________
(Author's signature)	(Supervisor's signature)
Address:	
_________________________	
_________________________	_________________________
_________________________	Supervisor's name
Date: ____________________	Date: ____________________

DECLARATION OF ORIGINALITY

I declare that this report entitled "**Exception Handling for 5-Stage Pipeline Micro-Architecture**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : __________________________

Name : _____________________________

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my final year project supervisor, Mr Mok Kai Ming for his guidance throughout the whole project with good patience and also clear explanation of all sorts of problems to help me to complete my project. This project might not be able to finish without his guidance. A million thanks to you for spending so much time to guide me throughout the project.

Secondly, I want to say thank you to my friends who give a lot of help when I face problem, comfort me when I am stressed, encourage and support me during hard times.

Lastly, I must say thanks to my parents and my family for their love, unconditional support and continuous encouragement throughout the course.

ABSTRACTS

This project, as stated in the title of the project, is to develop exception handling for 5-stage pipeline Micro-Architecture. Coprocessor (CP0) is implemented independently to receive and decode exception and interrupt signal which come from CPU or external device. In this project, coprocessor 0 (CP0) is developed. Component and exception/interrupt handling mechanism for software and hardware are developed. CP0 architecture and micro-architecture specification will be developed and implemented with HDL (Hardware Description Language) VERILOG. A series of test cases are developed to verify CP0 functionality. Lastly, the CP0 will integrate with RISC32 core. Test program will be developed to verify CP0 functionality. An exception handler is also developed to complete the whole exception/interrupt handling mechanism. Exception handler will examine the causes and dispatch CPU to appropriate ISR.

Table of Contents

List of Figures

List of Table

List of Abbreviation

MIPS	Microprocessor without Interlocked Pipeline Stages
RISC	Reduced Instruction Set Computing
CPU	Central Processing Unit
CP0	Coprocessor 0
IP	Intellectual Property
ISA	Instruction Set Architecture
RTL	Register Transfer Level
I/O	Input output
ISR	Interrupt service routine

Chapter 1 Introduction

1.1 Background Information

1.1.1 MIPS – RISC Processor

MIPS(Microprocessor without Interlocked Pipeline Stages) is a RISC(Reduced Instruction Set Computing) based processor developed by MIPS Technologies which executes instructions directly using hardware implementation without microprogrammed control. MIPS implementations are primarily used in embedded systems such as routers, residential gateways in networking and video games console such as Sony Playstation, Playstation 2, Playsation Portable and later used in many of Silicon Graphics computer products.

1.1.2 Coprocessor 0 (CP0)

MIPS exceptions/interrupts are handled by a peripheral device to the CPU known as coprocessor 0 (CP0). It is named as Coprocessor because it is separated from the main core and it is meant to handle the exceptional situation. Coprocessor 0 contains several registers used to configure exception handling and report the status of current exceptions. When exception occurs, CPU will suspend normal instruction execution and CP0 will save the exception states. The exception handler will examine the cause and handle the exception by providing appropriate service [1].

1.1.4 Exception

Exception is something that disrupts the normal flow of instruction. There are two type of exception which is asynchronous exception and synchronous exception. Asynchronous exception is the exception that occurs with no relation to the program executed while synchronous exception is exception that occurs at the same place every time the program is executed with the same data and memory allocation.

1.1.4 Interrupt

Interrupt is an event external to the current instruction execution that causes a change to the normal flow of instruction execution. Interrupts are asynchronous exception. Interrupts can be either software or hardware. For hardware interrupts, external devices such as mouse, keyboard, printers need CPU service. These interrupts are handled by Coprocessor 0 (CP0).

1.2 Project Motivation

A 32-bit 5-stage pipeline RISC soft-core can be advantageous in creating a core–based environment to assist research and development work in the area of developing Intellectual Properties (IP) cores. However, there are limitations in obtaining such workable core-based design environment.

Microchip design companies develop microprocessors cores as IP for commercial purposes. The microprocessor IP includes information on the entire design process for the front – end (modelling and verification) and back – end (layout and physical design) IC design. These are trade secrets of a company and certainly not made available in the market at an affordable price for research purposes.

Several freely available microprocessor cores are freely available from source such as the miniMIPS (www.opencores.org), the PH processor (Leicester University), uCore, Yellow Star (Manchester University), etc. Unfortunately, these processors do not implement the entire MIPS Instruction Set Architecture (ISA) and lack of comprehensive documentation. This makes them unsuitable for reuse and customization.

Verification is vital for proving the functionality of any digital design. The microprocessor cores mentioned above are handicapped by incomplete and poorly developed verification specifications. This hampers the verification process, slowing down the overall design process.

The lack of well – developed verification specifications for these microprocessor cores will inevitably affect the physical design phase. A design needs to be functionally proven before the physical design can proceed smoothly. Otherwise, if the front – end design needs to be changed, the physical process also needs to be redone.

The RISC32 project will look into all the above problems and create a 32-bit RISC core-based development environment to assist research work in the area of application specific hardware modeling. In the RISC32 project, it is divided into several units based on MIPS architecture.

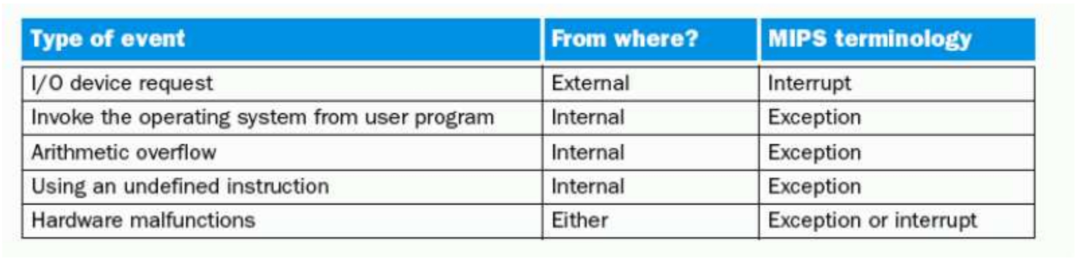
Chapter 2 Literature Review

2.1 Exception and interrupt

In MIPS, exception is described as something that disrupts the normal flow of execution. Exception is divided into two types which are asynchronous exception and synchronous exception. Asynchronous exception is exception that occurs with no relation to the program executed and normally caused by hardware such as I/O module called interrupt, memory error and power supply failure. Synchronous exception occurs every time when a program executed with the same data and same memory allocation which normally caused by arithmetic overflow, undefined instruction and traps.

In detail, exception can be differentiated into:

1. External events: event outside CPU core which send interrupt signal to CPU to get attention. These are interrupts. Interrupts are used to direct the attention of the CPU to some external event [2]. Interrupts are the only exception conditions that arise from something independent from CPU's normal instruction execution.
2. Memory translation exception: happens when memory address decoding error, a program tried to write to a write-protected page.
3. Program or hardware-detected error: arise when nonexistent instruction is detected (invalid instruction format), instruction that illegal in user-privilege is used, co-processor instruction executed when appropriate status register flag is disabled and integer overflow.
4. Data integrity problem: caused by bus to bus transferred data error (parity errors)
5. System calls and traps: caused by instruction themselves, for example system call instruction, conditional traps and breakpoints. These instructions are used to generate exception to interrupt the program for certain purpose.



Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Table 2.1 Examples of event and type of exception [3].

2.1.1 Precise Exception

MIPS architecture implements precise exception. Precise exception means when exception occurred, the particular exception will only point to the instruction that cause the exception. Besides, all the instructions before will be executed while all the later instructions will not be executed. This method eases the programmer work because they can ignore the timing effect of the CPU implementation.

The features provided with Precise Exception are:

- Unambiguous proof of cause: EPC will point only to the instruction that cause the exception. However, EPC might also point to the preceding branch for an instruction is in a branch delay slot, but will signal occurrence of this using the *BD* bit.
- Exceptions are seen in instruction sequence: In pipeline CPU, exception can arise in several different stage of execution. For example, a lw(load word) instruction suffer Memory Translation Exception will only arise exception signal in MEM stage (4th stage in pipeline), but at the same time, a later instruction that cause decoding error in ID stage (Instruction decode, 2nd stage in pipeline CPU) will arise exception first. To avoid this problem, MIPS only serve the exception if all previous instruction is complete successfully.
- Subsequent Instructions Nullified: Because of pipelining, instructions lying in sequence after the victim at EPC have been started. But MIPS guarantee no effect on visible register or CPU after return from Exception Handler.

2.1.2 Vectored interrupt

Vectored interrupt is an interrupt handling method in which the causes of interrupt will directly affect the address to be dispatch. Each interrupt input will be given a unique address, corresponding to its causes. This interrupt handling method is not implemented in MIPS processor.

2.1.3 Interrupt Service Routine (ISR)

Interrupt service routine is software that hardware or software invokes in response to an interrupt [4]. Interrupt service routine then will examine the interrupt and determine ways to handle it. After handle the interrupt, it will return from interrupt and then continue the program execution.

2.1.4 Interrupt Processing

Most processors generally share the same process of interrupt processing but only minor differences in how the processors save their status and call the interrupt service routine. When an interrupt occurs, the processor will finish the current instruction and store status and return address. Then the processor will call the corresponding interrupt service routine and start executes the interrupt service routine. Finally, when the processor done the execution of interrupt service routine, it will return from the interrupt and resume the program execution.

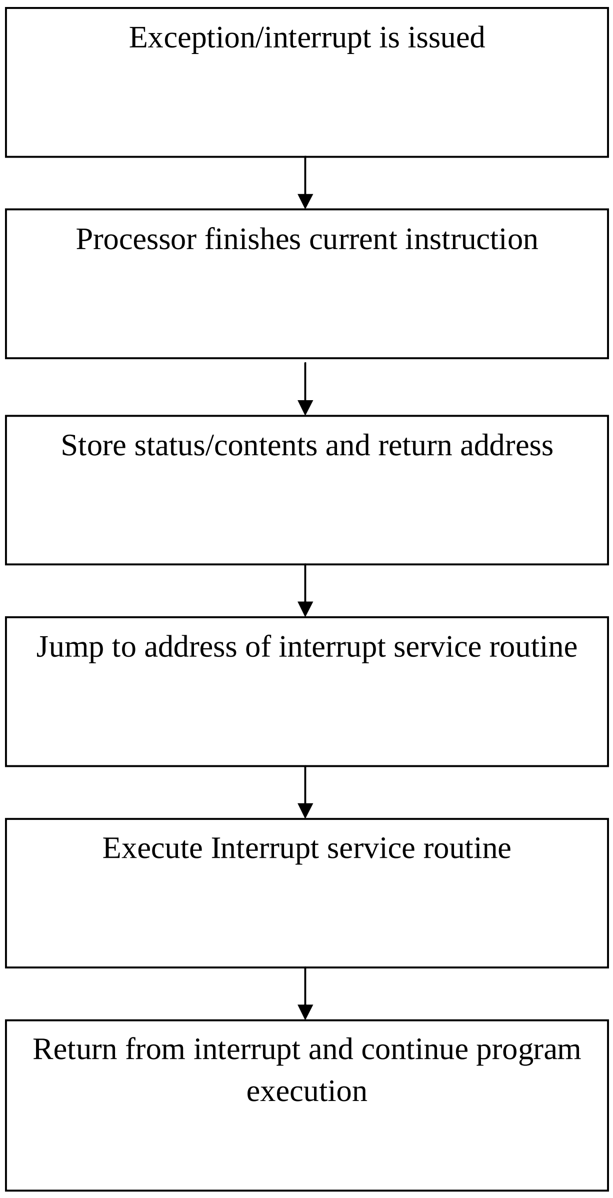

Exception/interrupt is issued
Processor finishes current instruction
Store status/contents and return address
Jump to address of interrupt service routine
Execute Interrupt service routine
Return from interrupt and continue program execution

Figure 2.1 Interrupt Handling Process

2.1.5 Exception Handling

Any MIPS exception handler routine has to go through the same stages.

- Bootstrapping: When enter the exception handler, very little of the state of the interrupted program has been saved. So the first step is make yourself a enough room to do whatever you want without overwriting something vital to the software that has just been interrupted [2].
- Dispatching different exceptions: Get the exception code from the cause register. It tells what and why the exceptions occur [2].
- Constructing the exception processing environment: Complex exception handling routines will probably be written in high level language and will want to be able to use standard library routines. A piece of stack memory that isn't be used by any other piece of software has to be provided and save all the values of any CPU registers that might be vital to the interrupted program and that called subroutines are allowed to change [2].
- Processing the exception: Here is where you can do whatever you like.
- Preparing to return: Return into low level dispatch code from subroutine (high level function). Saved registers are restored and CPU return to its safe (kernel mode, exceptions off) state by changing status register value to its postexception value [2].
- Returning from an exception: instruction eret is used. It clears the status register EXL bit and return to the address stored in EPC [2].

2.1.6 Exception Handling by MIPS

When exception occurs, CPU suspends normal instructions execution. CP0 will save the exception states. CP0 records the cause of exception in cause register. It then saves the return address in exception program counter (EPC). Processor will go into kernel mode. MIPS fixed the exception handling code at 0x8000 0180 where the exception handler examine the cause of exception and jump to a more specific code also in kernel to handle exception. Lastly, eret is used to resume normal program execution if not terminated.

2.2 Coprocessor 0 (CP0)

MIPS coprocessor 0 is a piece of hardware implementation that implements independently from the main processor core that functions to handle interrupt from hardware or exception from program or the instruction that is executing.

2.2.1 MIPS CP0 Implemented Register

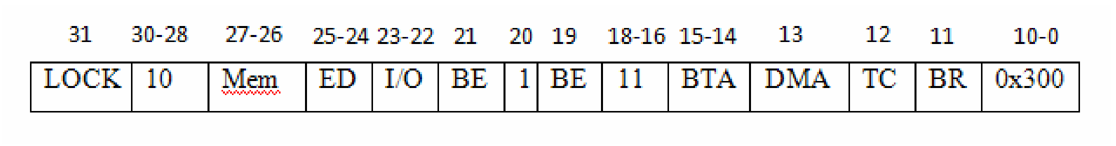
For commercial product, MIPS co-processor 0 implemented 32 registers according to MIPS R4000 specification [5]. There is only few registers which are important for all type of processor while most of the registers implemented only for dedicated function of specific processor. The registers that are important for all processors are Status Register (SR), Exception Program Counter (EPC) , Cause Register, Count Register, Compare Register, Bus Control Register (BusCtr), Port Size Register and Bad Virtual Address (BadVAddr Register).

Name	Register no.	Usage
BusCtrl	$2	Configure bus interface signals. Needs to be setup to match hardware implementation
BadVAddr	$8	Offending memory reference
Count	$9	Current Timer, which increment every 10ms
PortSize	$10	Used to flag some program address regions as 8-bit or 16-bits wide. Must be programmed to match hardware implementation.
Compare	$11	Interrupt when Count Register ≡ Compare Register
Status	$12	Interrupt mask, enable bits and status when exception occurred
Cause	$13	Exception type and pending interrupt
EPC	$14	Address of instruction that caused exception

Table 2.2 Standard CP0 register and usage

2.2.2 BusCtrl Register

This register configures buses in a cheap and simple way, without involving extra circuitry. Figure below shows the layout of BusCtrl register.



31	30-28	27-26	25-24	23-22	21	20	19	18-16	15-14	13	12	11	10-0
LOCK	10	Mem	ED	I/O	BE	1	BE	11	BTA	DMA	TC	BR	0x300

Figure 2.2 Bus control **(**BusCtrl**)** register.

- Lock [31]: Is used to prevent changes of the register field after initialized is done. Clear when system is reset.

- 10 (can be other value) [30:28]: Specified bit pattern is written in this field.

- Mem [27:26]: “MemStrobe Control”. Set in bit position 27 is to enable memory reads while set in bit position 26 is to enable memory write.

- ED [25:24]: “ExtDataEn control”. Encoded as for memory. In order to make this pin as output, BR[10] field must be zero[R3000 spec].

- IO [23:22]: “IOStrobe control”. Encoded as for memory. In order to make this pin as output, BR[10] field must be zero[R3000 spec].

- BE16: “BE16(1:0) read control”. “0” to make these pins active on write cycles only[R3000 spec].

- BE: “BE(3:0) read control”. “0” to make these pins active on write cycles only [R3000 spec].

- BTA [15:14]: “Bus Turn Around Time”. Program with a binary number between 0 and 3 for 0-3 cycle of guaranteed delay between the end of a read cycle and the start of the address phase of the next cycle. This field enables the use of devices with slow tri-state time, and enables the system designer to save cost by omitting data transceivers [R3000 spec].

- DMA [13]: “DMA Protocol Control”. When is set, CPU uses its DMA control pins to communicate its desire for the bus even while a DMA is in progress[R3000 spec].

- TC [12]: “TC Negation Control”. TC is the output pin which is activated when the internal timer register *Count* reaches the value stored in *Compare*. Clear this field make TC pin just pulse for a couple of clock periods; set this field TC pin will be asserted on a compare and remain asserted until software explicitly clears it (BY re-writing *Compare* with any value) [R3000 spec].

- BR[11]: “SBrCond(3:2) control”. Clear to recycle the SBrCond(3:2) pins as IOStrobe and ExtDateEn respectively [R3000 spec].

2.2.3 BadVAddr Register

This register is used to store memory address where the exceptions were occurred. For example, instruction LW (Load Word) from data memory address X which lead to memory address translate error (which the address provided is not valid, wrongly aligned or outside the range that supposed to be) will store in this register.

2.2.4 Count and Compare Register

This is a 24-bit counter/timer that running at CPU cycle rate. Count register is counting up and reset to zero when the count has reached the value in the Compare register. When Count Register is reset, TC in BusCtrl register will asserted high for a clock cycle. This is meant to generate an interrupt signal when TC is connected to interrupt input. After reset, the Compare register value will set to 0xFF_FFFF, which is maximum value of 24-bit, hence, the counter can runs up to 2^{24}-1 (1677215).

2.2.4 PortSize Register

This register is used to flag different part of the program address space for accesses to 8, 16 or 32 bit wide memory. This register must be programmed to match hardware implementation.

2.2.4 Status Register (SR)

This register contains the interrupt masking bit which enable/disable particular interrupt and status information. The layout of status register is shown below.

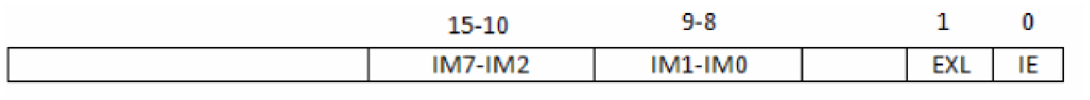


Figure 2.3 Status Register Layout

status[15:10] is external hardware Interrupt Masking bit, which used to enable/disable interrupt level. For example, if IM*i* is set to "1", interrupt of level *i* is enable. On the other hand, status[9:8] (IM1-IM0) is software writeable bit, which allowed software to mask/unmask the interrupt level.

status[1] is exception level (EXL) bit, is used to determine whether the processor is in kernel or user mode. It is set by any exception. When set to "1", it indicates that the processor is in kernel mode, and hence, disables all the interrupt. When set to "0", it indicates that processor is in user mode, which allows interrupt happen.

Status[0] is global interrupt enable (IE). When it is set to "1", processor permits interrupt; else no interrupt will be permitted. This bit usually configured by OS to control the process whether accept interrupt or not.

2.2.5 Cause Register

Cause of any exception and pending exception are stored in Cause Register. The exception code is stored as an unsigned integer in cause[6:2] while pending exception is stored in cause[15:8]. Figure below shows the Cause Register layout.

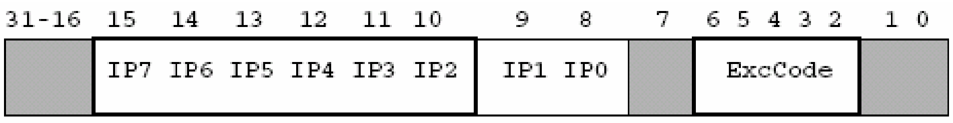
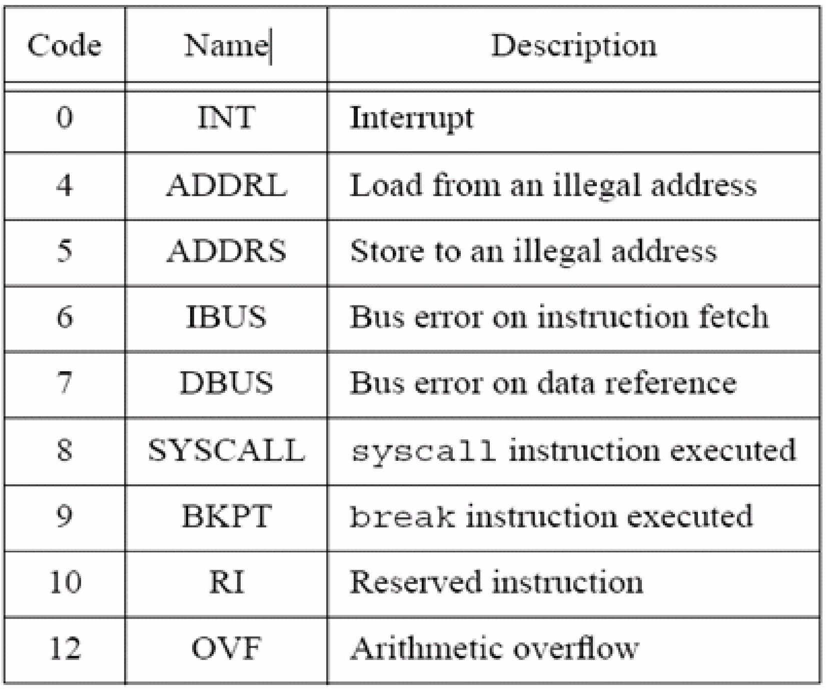


Figure 2.4 Cause Register Layout

When an interrupt occurs, the particular interrupt level field will be set to “1” and clear after the interrupt is served. However, SPIM only simulate 6 out of 8 pending interrupt which is IP7-IP2. IP1 and IP0 are software interrupt bit and not visible in SPIM.

The exception code, cause[6:2] is used to indicate the cause of particular interrupt. Table Below shows the exception code implemented by SPIM corresponding to different exception causes.



Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	`syscall` instruction executed
9	BKPT	`break` instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

Table 2.3 Exception code and causes [6]

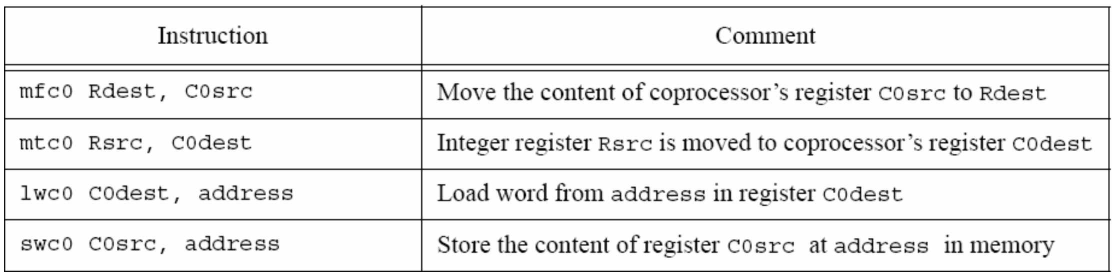
The left over part, exception code 1-3 is reserved for virtual memory (TLB exception), exception code 11 is to indicate particular coprocessor is missing while exception code above 12 are used for floating point exception or reserved.

2.2.6 EPC

EPC is used to store the return point when return from exception. In other word, EPC is used to store the address of the instruction that cause exception.

2.2.7 Instructions Associate with Exception Handling

Some instructions are dedicated to access the register in MIPS coprocessor 0. In order to access coprocessor 0 register, the code must have kernel privilege. Table below shows the instruction and usage of the instructions.



Instruction	Comment
`mfc0 Rdest, C0src`	Move the content of coprocessor's register `C0src` to `Rdest`
`mtc0 Rsrc, C0dest`	Integer register `Rsrc` is moved to coprocessor's register `C0dest`
`lwc0 C0dest, address`	Load word from `address` in register `C0dest`
`swc0 C0src, address`	Store the content of register `C0src` at `address` in memory

Figure 2.5 Instruction used to access CP0 register [6].

2.3 MIPS Memory Map

Memory map in MIPS is not implemented in hardware. It just a convention followed by most programmer. However, in real world, this convention is applied to almost all the MIPS CPU. Figure below shows MIPS memory map.

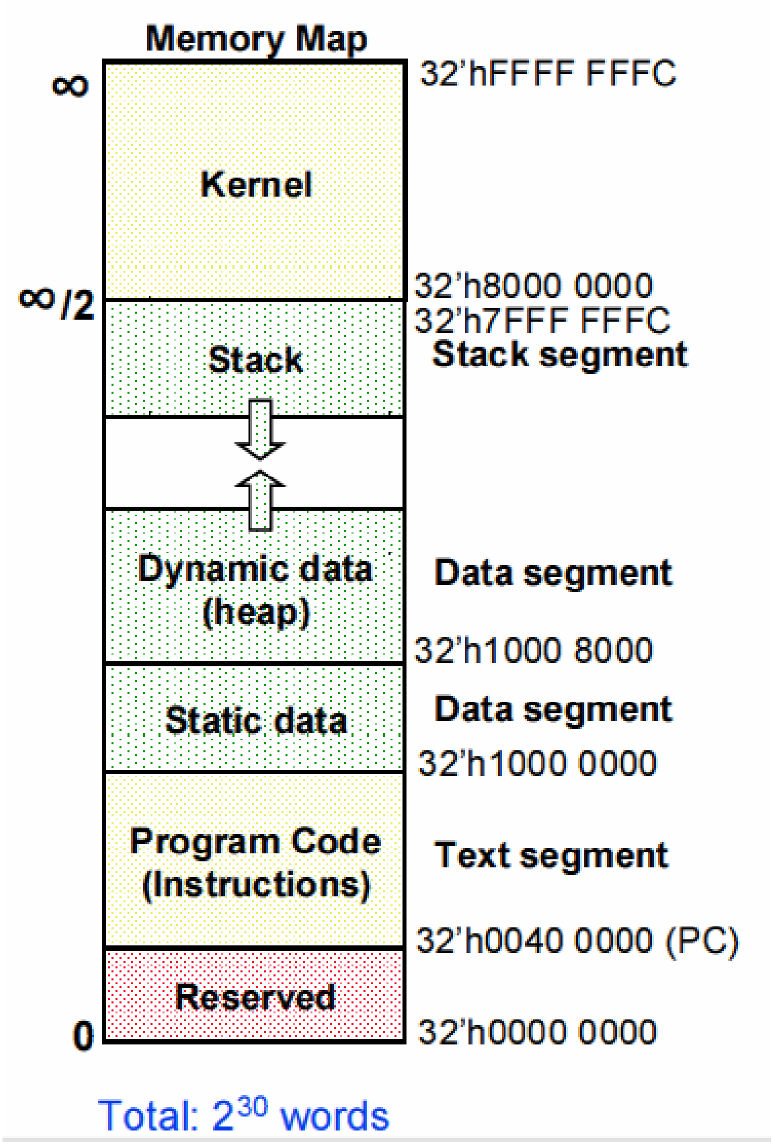


Figure 2.6 MIPS Memory Map [1]

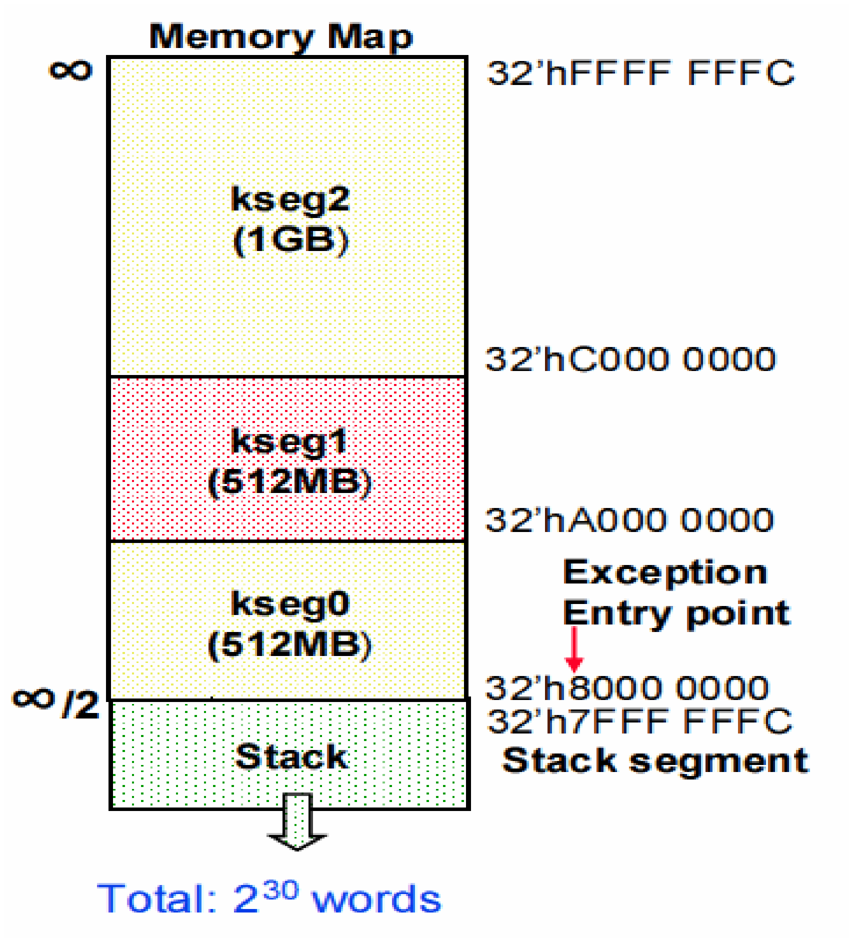


Figure 2.7 Kernel Segment [1]

MIPS partition memory into 2 major parts, User Space and Kernel Space, where 0x0000_0000 to 0x7FFF_FFFC is user space and everything above 0x7FFF_FFFC is kernel space. Below are the functions of each memory space.

0x0000_0000 – 0x7FFF_FFFC: This lower 2GB memory allocation is permitted in user mode. It contains stack segment, data segment, text segment and reserved memory space.

kseg0 0x8000_0000 – 0x9FFF_FFFC: This 512MB memory region is normally accessed through cache. Exception and page table base register are allocated here. Here is also the exception entry point (software exception handling).

kseg1 0xA000_0000 – 0xBFFF_FFFC: Boot ROM, 512MB memory region which initialize CPU after reset.

kseg2 0xC000_0000 – 0xFFFF_FFFC: Kernel module, only accessible in kernel mode.

Chapter 3 Problem Statement, Project Scope and Objectives

3.1 Problem Statement

Up to date, a new micro architecture is re-implemented. There is a need to port CP0 to the newly implemented micro architecture. The new micro architecture only contains datapath unit, control unit, memory unit. CP0 is added to the new micro architecture in this project. After port in CP0 into the new micro architecture, the functionality is verified again to make sure every part is functioning properly. Kernel text segment cache and kernel data segment cache is added to the processor to store exception handler instruction. This project aims to implement CP0 which is able to handle 4 exceptions which are sign overflow, undefined instruction, syscall and I/O interrupt. The CP0 needed to be implemented to handle exception which may be caused by hardware or software. Hence, this RISC32 Coprocessor project is initiated.

3.2 Project Scope

This project aims design a coprocessor 0, implements into RISC32 microprocessor. The coprocessor worked as exception handler to handle exception for 5-stage pipeline microarchitecture. Specifications at architecture level and micro architecture level will be developed and the modeling of design will be constructed using Verilog. Functional behavior verification will be constructed using testbench and lastly integrate the design into 32-bit RISC processor. A set of test program will be used to verify the whole system. Lastly a report will be written. The report will document chip specification, architecture specification, microarchitecture specification, verification specification, test plan and verification result.

3.3 Project Objective

The objectives of this project are as follows:

- To analyze the existing implementation cp0 done by senior.
- To develop a new coprocessor 0.
- To develop a testbench to verify the functionality of coprocessor 0.
- To develop exception handler to handle exceptions.
- To design kernel text segment cache and kernel data segment cache.
- To integrate Coprocessor 0 into the RISC-32 processor.
- To develop test program to test the functionality of coprocessor 0 to handle 4 type of exceptions which are sign overflow, undefined instruction, syscall and I/O interrupt.

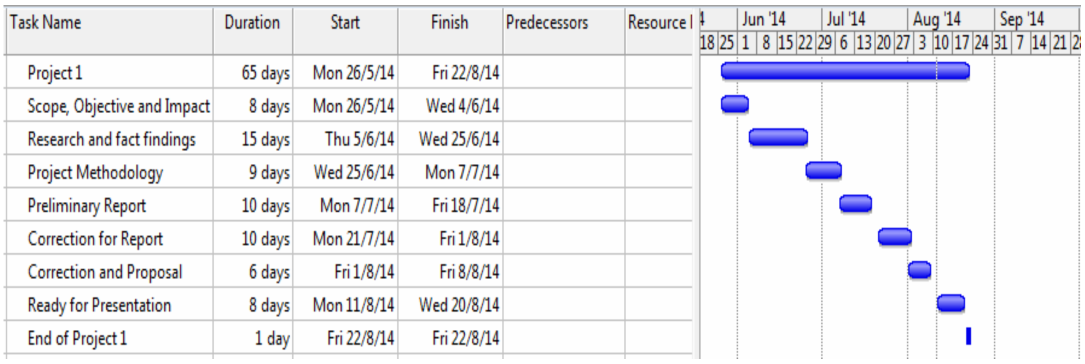
3.4 Impact and Significance

As a summary to the problem statement, there is a lack of well-developed and well-founded 32-bit RISC microprocessor core-based development environment. The development environment refers to the availability of the following:

- A well-developed design document, which includes the chip specification, architecture specification and micro-architecture specification.
- A fully functional well-developed 32-bit RISC architecture core in the form of synthesis-ready RTL written in Verilog HDL.
- A well-developed verification environment for the 32-bit RISC core. The verification specification should contain suitable verification methodology, verification techniques, test plans, testbench architectures etc.
- A complete physical design in Field Programmable Gate Array (FPGA) with documented timing and resource usage information.

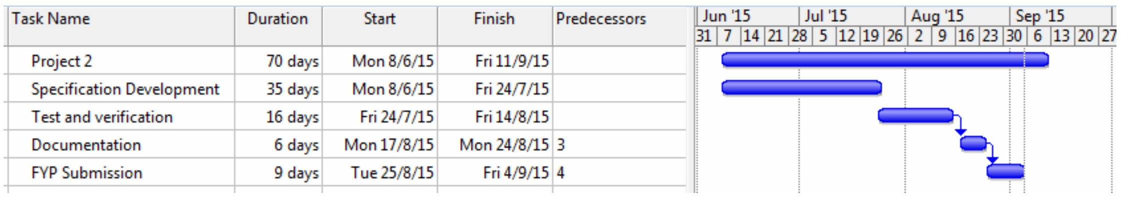
With the available well-developed basic 32-bit RISC RTL model (which has been fully functional verified), the verification environment and the design documents, researchers can develop their own specific RTL model as part of the development environment (whether directly modifying the internals of the processor or interface to the processor) and can quickly verify their model to obtain results, without having to worry about the development of the verification environment and the modeling environment. This can speed up the research work significantly. For example, a researcher may have developed an image-processing algorithm and modified the algorithm to obtain a structure that suits the hardware implementation. The structure can be modeled in Verilog as part of a specialized datapath or as a coprocessor interfacing to the RISC processor.

3.5 Project Plan



Task Name	Duration	Start	Finish	Predecessors	Resource I
Project 1	65 days	Mon 26/5/14	Fri 22/8/14		
Scope, Objective and Impact	8 days	Mon 26/5/14	Wed 4/6/14		
Research and fact findings	15 days	Thu 5/6/14	Wed 25/6/14		
Project Methodology	9 days	Wed 25/6/14	Mon 7/7/14		
Preliminary Report	10 days	Mon 7/7/14	Fri 18/7/14		
Correction for Report	10 days	Mon 21/7/14	Fri 1/8/14		
Correction and Proposal	6 days	Fri 1/8/14	Fri 8/8/14		
Ready for Presentation	8 days	Mon 11/8/14	Wed 20/8/14		
End of Project 1	1 day	Fri 22/8/14	Fri 22/8/14		

Figure 3.1 Gantt Chart for Project 1



Task Name	Duration	Start	Finish	Predecessors
Project 2	70 days	Mon 8/6/15	Fri 11/9/15	
Specification Development	35 days	Mon 8/6/15	Fri 24/7/15	
Test and verification	16 days	Fri 24/7/15	Fri 14/8/15	
Documentation	6 days	Mon 17/8/15	Mon 24/8/15	3
FYP Submission	9 days	Tue 25/8/15	Fri 4/9/15	4

Figure 3.2 Gantt Chart for Project 2

Chapter 4 Methods/Technologies Involved

4.1 Design Methodology

Design methodology is the method of development of a system. It provides guideline to successfully carry out a design work. Good Design methodology needs to ensure correct functionality, catching bugs early, satisfaction of performance and power goals, good documentation [7]. There are two type of design methodology which are Top-down and Bottom-up. In this project, Top-down design methodology is used since digital system always uses the abstraction concepts to simplify the design process.

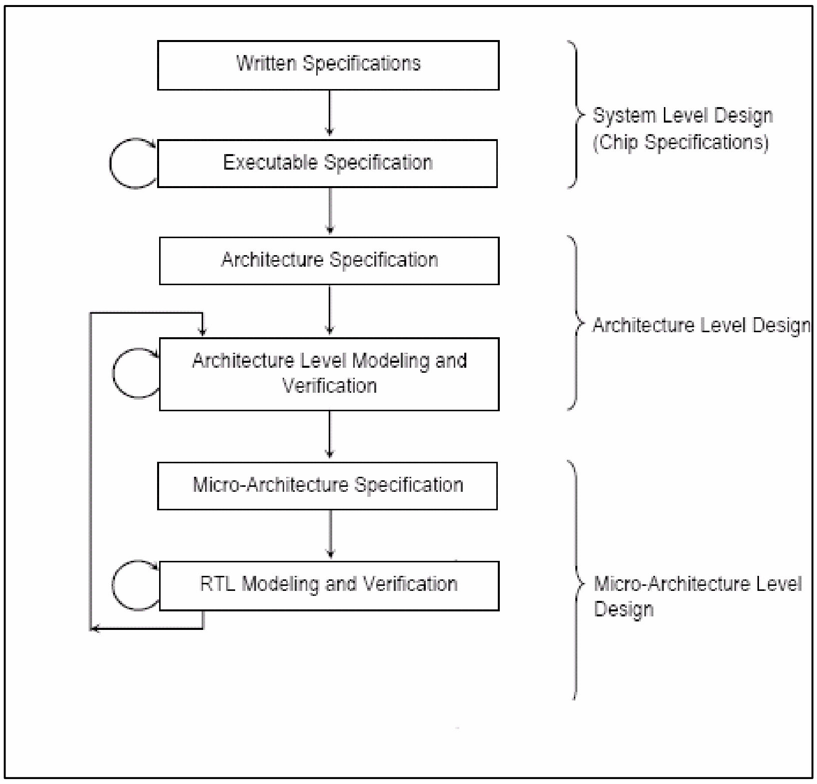

Written Specifications
Executable Specification
Architecture Specification
Architecture Level Modeling and Verification
Micro-Architecture Specification
RTL Modeling and Verification
System Level Design (Chip Specifications)
Architecture Level Design
Micro-Architecture Level Design

Figure 4.1 General Design Flow without Logic Synthesis and Physical Design **[**8**].**

4.1.1 System Level Design

System level design includes Written Specifications and Executable Specification is level where chip specifications are developed.

Written Specification- using English to write out the function, performance, cost and time constrain of a design.

Executable Specification-features and functionalities are described in high level programming language such as System C, Verilog.

4.1.2 Architecture Level Design

Architecture level design includes Architecture Specification and Architecture Level Modelling and Verification.

Architecture Specification- describes the internal of a chip and may contain design hierarchy, functional partitioning of the chip into units and inter-unit signaling and worst case timing.

Architecture Level Modelling and Verification- Algorithms are developed based on information from architecture specification to model the units that make up the architecture. The algorithms are then coded using hardware description language (HDL). Each unit is verified for functional correctness.

4.1.3 RTL Design

Micro-Architecture Specification- describes internal of a unit and may include:

- Unit interfaces and I/O pin description
- Unit functionality description
- Unit internal operation, function table etc to assist test plan
- Timing requirement
- Test plan
- Unit functional partitioning into blocks and inter-blocks signaling
- For each blocks/sub-blocks, may include
    i. Block interfaces and I/O pin description
    ii. A description of functionality of each block
    iii. Internal operation such as function table and text description
    iv. Finite-state machine (FSM) and Algorithmic-state machine (ASM)
    v. Timing requirements
    vi. Test Plan

RTL Modeling and Verification- RTL coding can begin after the micro-architecture specification has been developed. After models have been coded, they are verified for functional correctness.

4.2 Design Tools

Since this project is implemented using Verilog HDL, simulation tools that support Verilog HDL is needed. There are a lot simulation tools created by different companies which has their own advantages and disadvantages. Among them, 3 of the famous simulation tools will be discussed here.

(i) VCS

-Developed by Synopsys

-based on multi-core technology which cuts down verification time

-supports all popular design and verification languages

- Powerful debug and visualization environment

(ii) ModelSim

-Developed by Mentor Graphics

-Complete HDL simulation and debugging environment

-Provide Student Edition (SE) which limits to 10,000 lines of code

(iii) Quartus II

-Developed by Altera

-Provides complete design environment for system on a programmable chip (SOPC)

-Can work with multiple files at the same time.

Simulator	VCS	ModelSim	Quartus II
Company	SYNOPSYS Predictable Success	Mentor Graphics	ALTERA
Language Supported	VHDL-2002 V2001 SV2005	VHDL-2002 V2002 SV2005	VHDL Verilog HDL
Platform Supported	Linux	-Windows XP/Vista/7/8 -Linux	-Windows XP/7/8 -Linux
Availability for free	No	YES (SE Edition only)	No

Table 4.1 Comparison Between Simulators

Based on Table 4.1, ModelSim PE Student Edition 10.2b is chosen because it is available for free while other simulators may need to pay for license which may not be affordable.

4.3 Exception/Interrupt Handling Mechanism

4.3.1 Exception/Interrupt Handling

Exception/interrupt handling is very vital for a processor to function well and interface with external devices. Without exception/interrupt handling mechanism, processor may not be able to deal with event which is not belongs to normal program execution. For example, arithmetic overflow, syscall function, undefined instruction and external device interrupt request.

4.3.2 RISC32 Exception/Interrupt Handling Mechanism

RISC32 exception/interrupt handling mechanism is categorized into 2 parts, which is hardware handling and software handling.

4.3.2.1 Hardware Handling

When exception/interrupt arises in processor, CP0 will first check the EXL and IE bit of the status register (status [1] and status [0]). Exception/interrupt will be prohibited if EXL bit is set to “1”. All interrupt can be further prohibited if IE bit is set to “0”, which means no interrupt is allowed.

After ensure the exception/interrupt can be process, CP0 will update the cause register with value associate to the causes. Then current program counter (PC) value will be stored in EPC as a return point after exception handling. Exception handler entry address will be output to PC to force the processor into exception handler. Finally, EXL bit will be set to “1” to prevent any other exception/interrupt service.

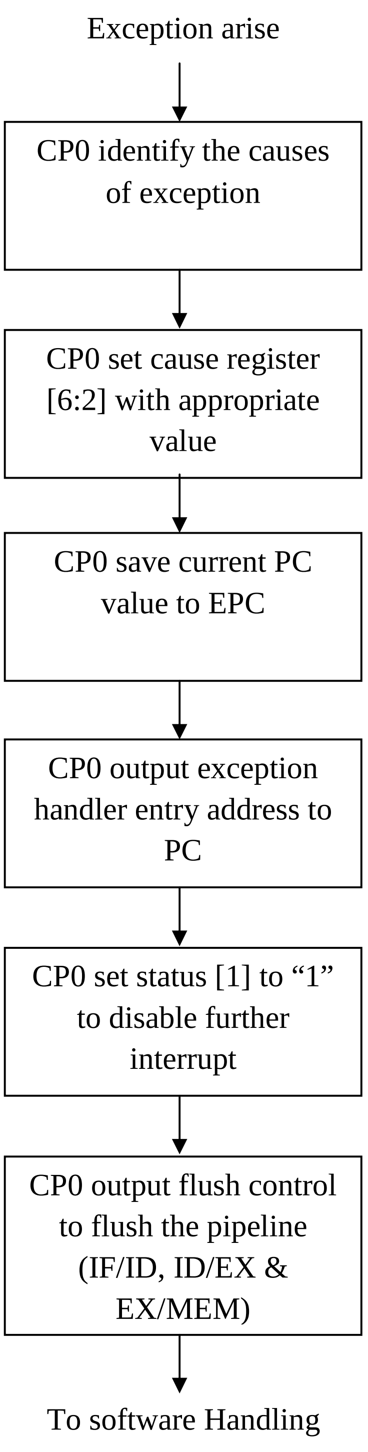

Exception arise
CP0 identify the causes of exception
CP0 set cause register [6:2] with appropriate value
CP0 save current PC value to EPC
CP0 output exception handler entry address to PC
CP0 set status [1] to "1" to disable further interrupt
CP0 output flush control to flush the pipeline (IF/ID, ID/EX & EX/MEM)
To software Handling

Figure 4.2 Hardware Exception Handling Flow Chart

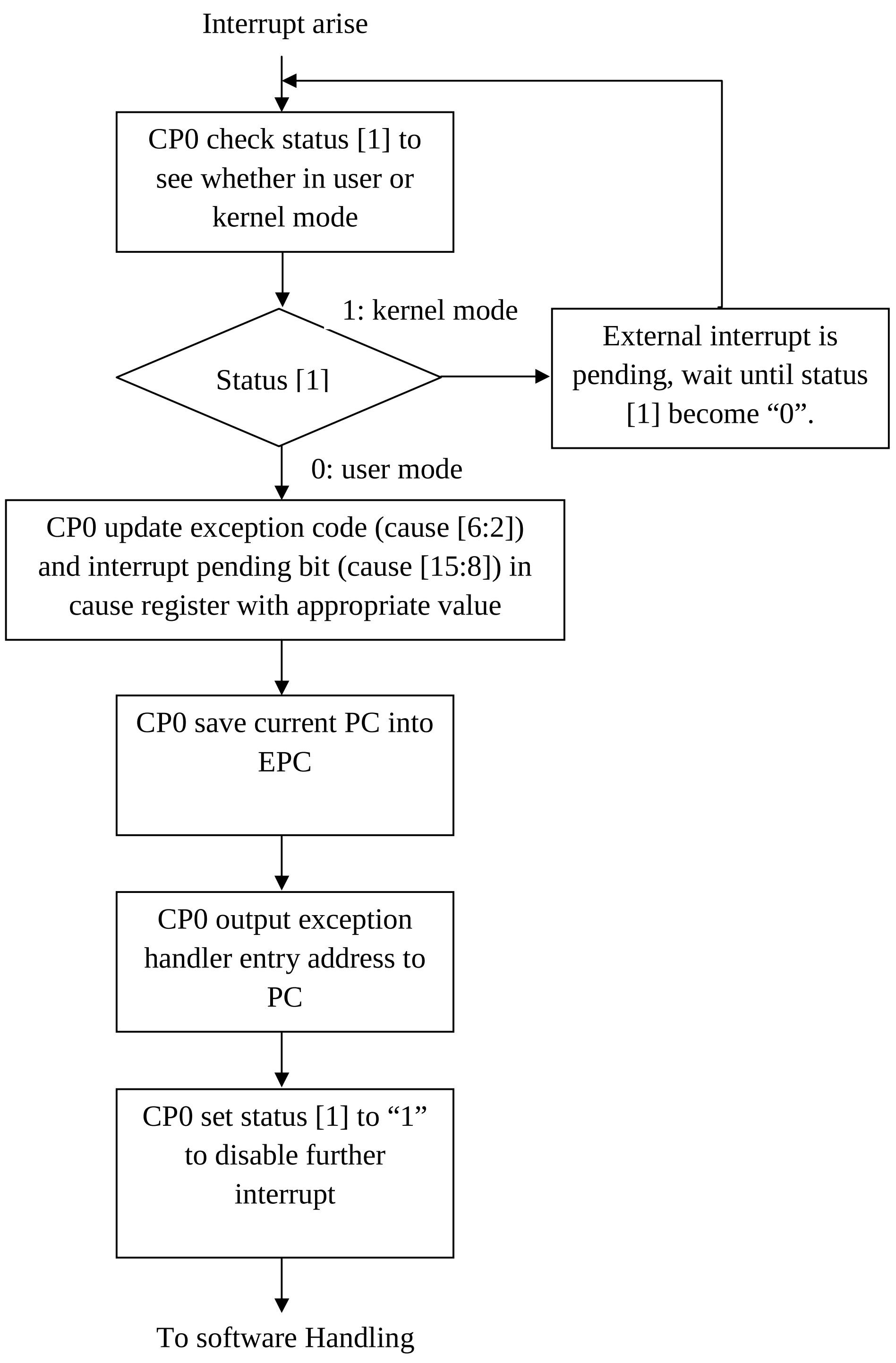

Interrupt arise
CP0 check status [1] to see whether in user or kernel mode
Status [1]
1: kernel mode
External interrupt is pending, wait until status [1] become “0”.
0: user mode
CP0 update exception code (cause [6:2]) and interrupt pending bit (cause [15:8]) in cause register with appropriate value
CP0 save current PC into EPC
CP0 output exception handler entry address to PC
CP0 set status [1] to “1” to disable further interrupt
To software Handling

Figure 4.3 Hardware Interrupt Handling Flow Chart

4.3.2.1 Software Handling

In software handling, cause of exception/interrupt will be determined in Exception Handler and appropriate service routine will be selected. The handler code is placed in kernel segment in memory, which starts from 0x8000_0180.

The sequence of process:

1. Save register state into memory stack to clear up register space so that register can be used by interrupt process.

2. Load CP0 cause register into register k0 using instruction mfc0.

3. Extract and mask the exception code (cause [6:2]) to determine the cause of exception.

4. Jump to appropriate service routine.

5. Clear cause register and reset status register.

5. Restore register from memory stack.

6. Return to user program execution using instruction eret.

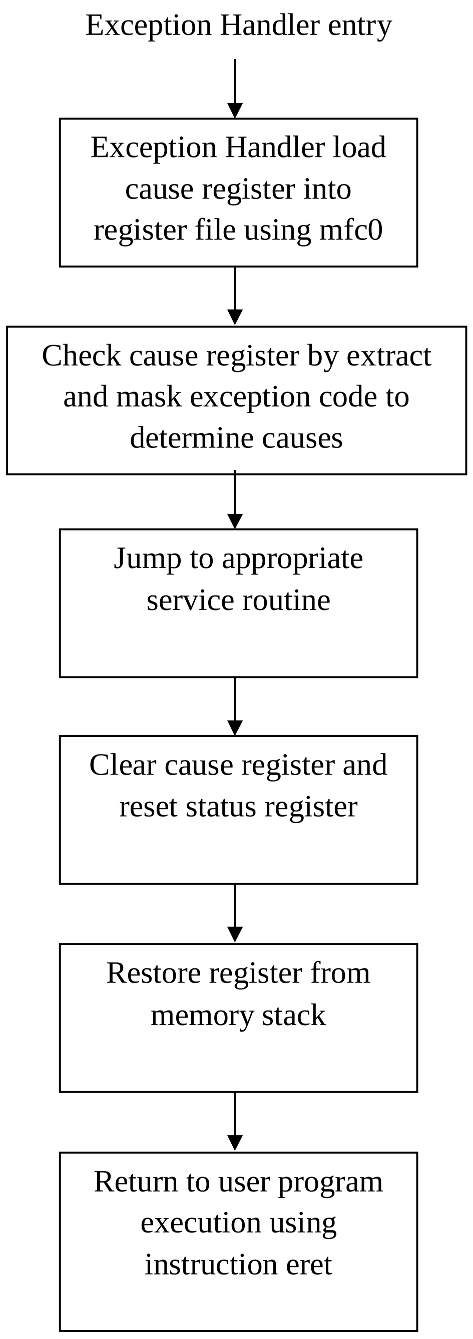

Exception Handler entry
Exception Handler load cause register into register file using mfc0
Check cause register by extract and mask exception code to determine causes
Jump to appropriate service routine
Clear cause register and reset status register
Restore register from memory stack
Return to user program execution using instruction eret

Figure 4.4 Software Exception Handling Flow Chart

4.4 Instruction used in Exception/Interrupt Handling

Two register accessing instructions are mtc0 and mfc0 to transfer data between CP0 register and processor register. A special instruction eret is used to return processor to normal program. The instruction format is shown below:

mfc0 rt,rd : move data from CP0 register to CPU register.

31-26	25-21	20-16	15-11	10-0
010000	00000	rt	rd	00000000000

Figure 4.5 mfc0 Instruction Format

mtc0 rt,rd : move data from CPU register to CP0 register.

31-26	25-21	20-16	15-11	10-0
010000	00100	rt	rd	00000000000

Figure 4.6 mtc0 Instruction Format

31-26	25	24-6	5-0
010000	1	0000000000000000000	011000

Figure 4.7 EPC Instruction Format

4.5 Pipeline Flushing

When an exception occurs, there are different instructions in different pipeline stages. Since, we don't want the instruction causing exception to save any of its state into processor, either memory or register file, we need to use pipeline flushing to9 flush the pipeline stage. When exception occurs, CP0 will receive signal from processor, decode and update register and output exception handler address to PC. At the same time, CP0 will send signal to flush the pipeline. Only instruction come after the exception is flushed. The instructions before the instruction which caused exception is left to be finished. For sign overflow, IF/ID, ID/EX and EX/MEM are flushed. For undefined instruction, IF/ID and ID/EX are flushed while for syscall, only IF/ID need to be flushed. I/O interrupt does not need to flush the pipeline but let the normal instruction finishes only jump to exception handler to handle interrupt. Pipeline flushing is just simply change all the control signal in pipeline to "0" which turn the instruction into nop (no operation). Thus, it can prevents faulty value to be written into register or memory.

Chapter 5 System Specification

5.1 System Feature

	RISC32 with CP0
Dummy Instruction Cache (KB)	16
Dummy Data Cache (KB)	16
Data width (bits)	32
Instruction width (bits)	32
General Purpose Register	32
Special Purpose Register	HILO, PC
Co-Processor Register	**32**
Pipelined Stage	5
Data Hazard Handling	Yes
Control Hazard Handling	Yes
Interlock Handling	Yes
Exception Handling	**Yes (4)**
Data Dependency Forwarding	Yes
Branch Prediction	Dynamic – 2bits scheme
Multiplication (size of multiplier and multiplicand)	yes – 32 bits
Branch Delay Slot	Not supported
Instruction supported	**40**

Table 5.1 RISC32 Features

5.1.1 System Functionality

1. Divide execution of instruction into 5 stages:

- IF(Instruction Fetch)	Fetch instruction from instruction cache into the datapath.
- ID(Instruction Decode)	Decode instruction and fetch $rs & $rt registers.
- EX(Execute)	Execute instruction in the ALB.
- MEM(Memory)	Access data cache, load or store.
- WB(Write Back)	Write back the result to the register file.

2. Resolve data hazard by data forwarding.

3. Resolve load-use instructions problem using stalling.

4. Resolve structural hazards using separating data and instruction cache.

5. Resolve control hazards by branch prediction.

6. Resolve exception/interrupt using CP0 and exception handler.

5.2 Operating Procedure

1. Start the system.
2. Porting sequence of instruction into instruction cache.
3. Reset the system for at least 2 clocks.
4. After the reset, the system will automatically fetch and run the program inside instruction cache.
5. Observe the waveform from the development tools (Modelsim).

5.3 Naming Convention

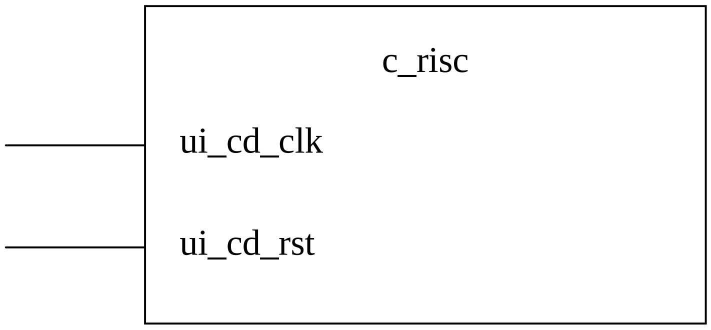
Module	– [lvl]_[mod. name]
Instantiation	– [lvl]_[abbr. mod. name]
Pin	– [lvl][type]_[abbr. mod. name]_[pin name]
Signal	– [type]_[abbr. mod. name]_<stage>_[pin name]

Abbreviation	Description	Case	Available	Remark
lvl	level	lower	c: Chip u: Unit b: block	
mod. name	Module name	Lower all	Any	
abbr. Mod. name	Abbreviated module name	Lower all	Any	Maximum 3 characters
Type	Pin type	Lower	o: output i: input r: register w: wire	
Stage	Stage name	Lower all	If, id, ex, mem, wb	Optional
Pin name	Pin name	Lower all	any	Several word separate by "_"

Table 5.2 Naming Convention

5.4 RISC32 Pipeline Processor with CP0 and I/O Description

5.4.1 Processor Interface



5.4.2 I/O Pin Description

c_risc
Input:
Pin name : ui_cd_clk Pin Class : Global Registered: Yes Source->Destination: External →c_risc Pin Function: Provide clock signal for the pipeline processor.
Pin name : ui_cd_rst Pin Class : Global Registered: Yes Source->Destination: External →c_risc Pin Function: Provide reset signal for the pipeline processor.

Table 5.3 RISC32 Processor I/O Description

5.5 Memory Map

Purpose	start address	Direction	Segment
Kernel module	0xC000 0000	Up	Kseg2
Boot Rom		Up	Kseg1
I/O register(if below 512MB)	0xA000 0000	Up	
Direct view of memory to 512MB linux kernel code and data		Up	Kseg0
Exception Entry point	0x8000 0000	Up	
Stack	0x7FFF FFFC	Down	Kuseg
Program heap	0x1000 8000	Up	
Dynamic library code and data	0x1000 0000	Up	
Main program	0x0040 0000	Up	
Reserved	0x0000 0000	Up	

Table 5.4 Memory Map

5.5.1 Memory map description

Kernel module

-Accessible by kernel*

Boot Rom

-Start-up ROM which keep the system configuration*

I/O registers (if below 512MB)

-External IO device register*

Direct view of memory to 512MB linux kernel code and data

-Memory allocation to view linux kernel code and data*

Exception Entry point

-Software exception handling *

Stack

-Use for argument passing

Program heap

-Dynamic memory allocation such as malloc()

Dynamic library code and data

-Data segment which is access by variable

Main program

-Text segment which contain the main program

Note *: required CP0

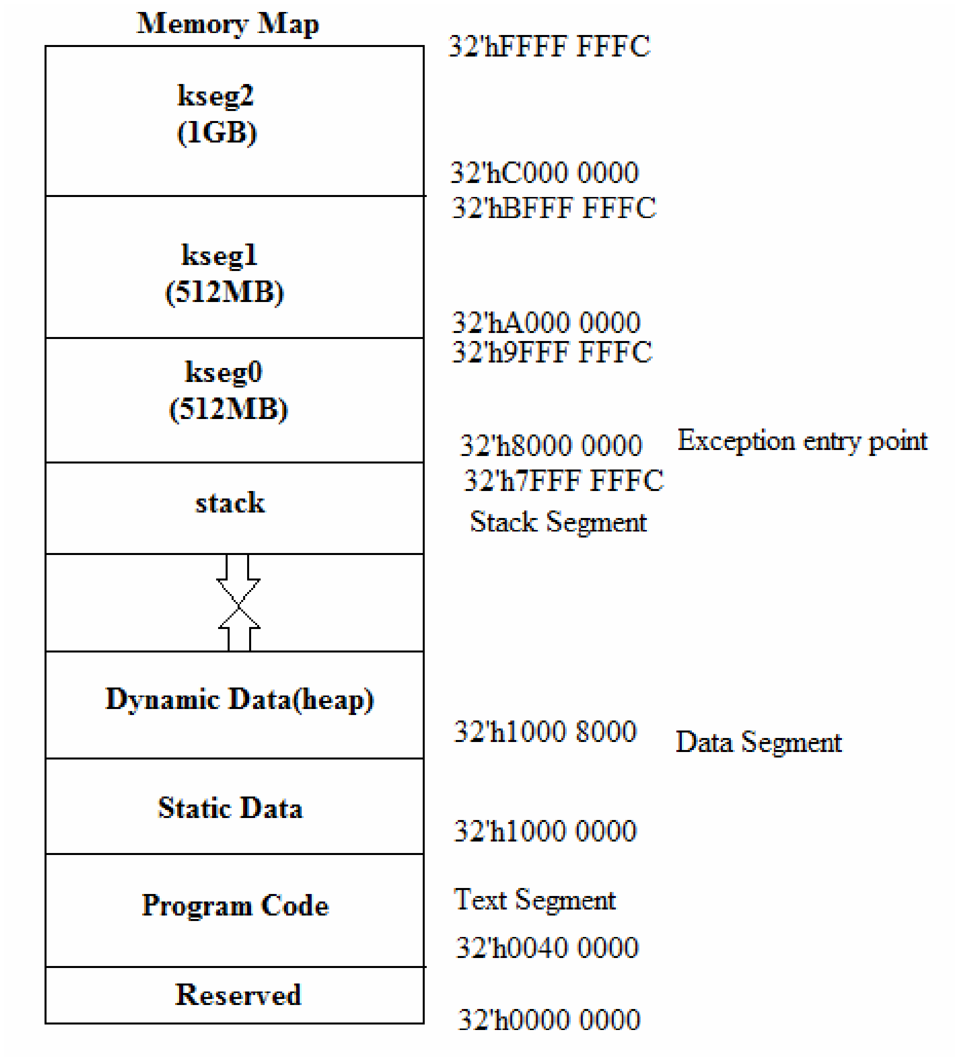


Figure 5.1 Memory Map

However, due to the limitation of modelsim student edition which only support up to 8k memory, the cache size will set text segment from 32'h0040_0000 to 32'h0040_1FFC, data segment from 32'h1000_0000 to 32'h1000_1FFC, kernel text segment from 32'h8000_0000 to 32'h8000_1FFC and kernel data segment from 32'h9000_0000 to 32'h9000_0FFC.

5.6 System Register

5.6.1 General Purpose Register

Width : 32-bits

Size : 32 units

Retrieving method : 5-bits address as index

Name	Address	Use	Preserved Across A Call?
$zero	0	Constant Value 0 (hardwired)	N.A.
$at	1	Assembler Temporary	No
$v0 - $v1	2 - 3	Value for Function Results and Expression Evaluation	No
$a0 - $a3	4 - 7	Arguments	No
$t0 - $t7	8 – 15	Temporaries	No
$s0 - $s7	16 - 23	Saved temporaries	Yes
$t8 - $t9	24 – 25	Temporaries	No
$k0 - $k1	26 -27	Reserved for OS kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Table 5.5 General Purpose Register

5.6.2 Special Purpose Register

Width : 32-bits

Size : 2 units

Retrieving method : Via instructions: MFHI, MTHI, MFLO, MTLO, MULT or MULTU

Name	definition	location in double [64:0]
HI	Most Significant Word	Double [63:32]
LO	Least Significant Word	Double [31:0]

Table 5.6 Special Purpose Register

5.6.3 Program Counter Register

Width	: 32-bits
Size	: 1 unit
Retrieving method	: Control by instruction address generator control.

5.6.4 CP0 Register

Name	Address	Use
$b_cp0_stat	12	Interrupt mask, enable bits and status when exception occurred
$b_cp0_cause	13	Exception type and pending interrupt
$b_cp0_epc	14	Address of instruction that caused exception

Table 5.7 CP0 Register

5.7 Instruction Formats and Addressing Modes

5.7.1 Instruction Formats

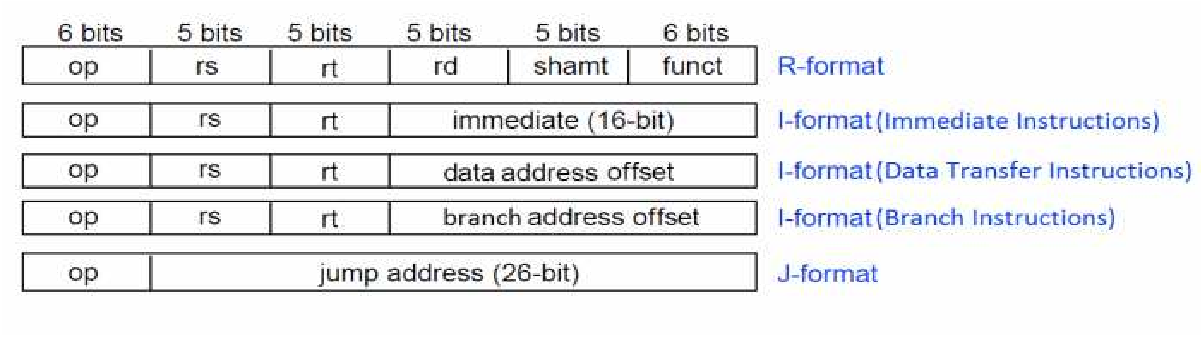


Figure 5.2 Instruction Format

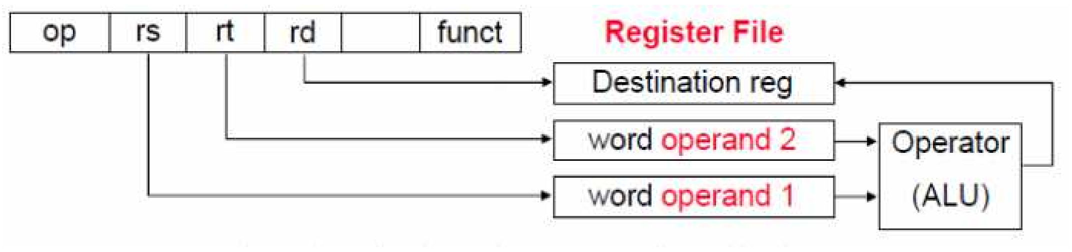
Abbreviation	Definitiion	Width
op	Operation code	6
rs	Source register	5
rt	Target register	5
rd	Destination register	5
shamt	Shift amount	5
funct	Function field	6
immediate	Immediate	16
data address offset	Data address offset	16
branch address offset	Branch address offset	16
jump address	Jump address	26

Table 5.8 Instruction Format Definition

5.7.2 Addressing Modes

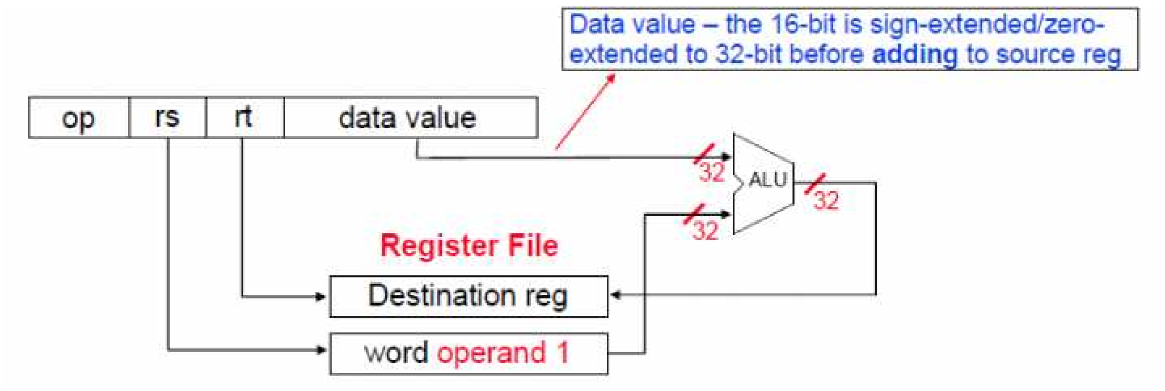
a) R-format

Register addressing: Perform operation on source and target register and store the result into destination register.

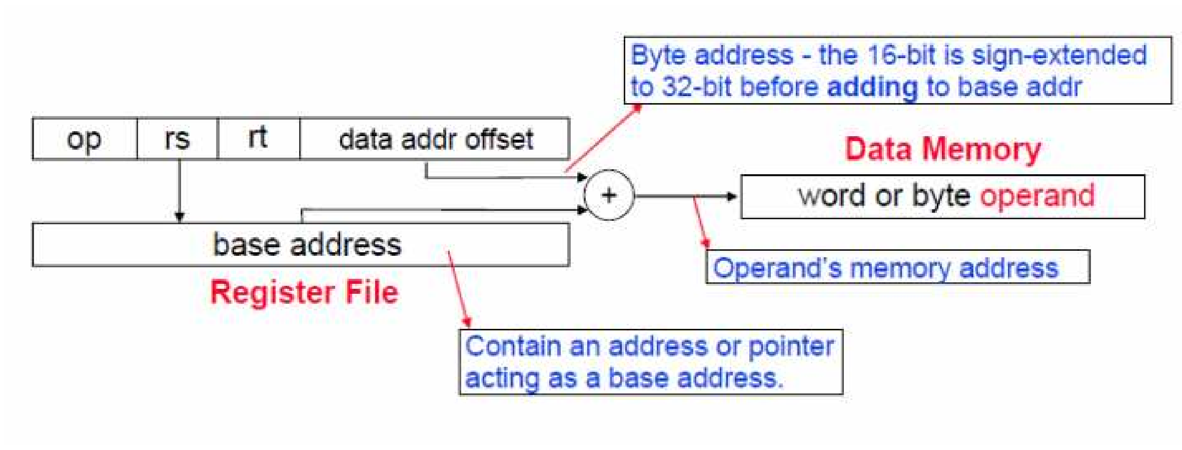


b) I-format

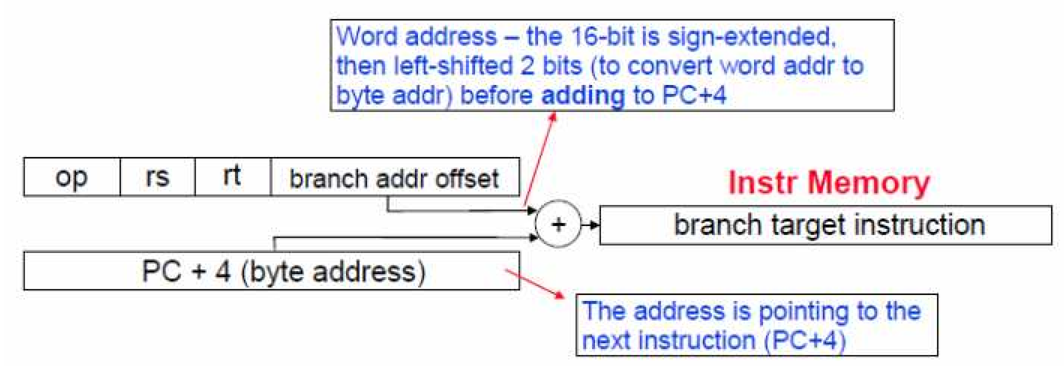
i. Immediate addressing: Perform operation on source register and immediate and store the result into target register.


Data value – the 16-bit is sign-extended/zero-extended to 32-bit before adding to source reg
op
rs
rt
data value
32
32
ALU
32
Register File
Destination reg
word operand 1

ii. Based displacement addressing: Perform operation on source register and immediate, the result is then uses as address to access the data memory to load/store data to/from target register.

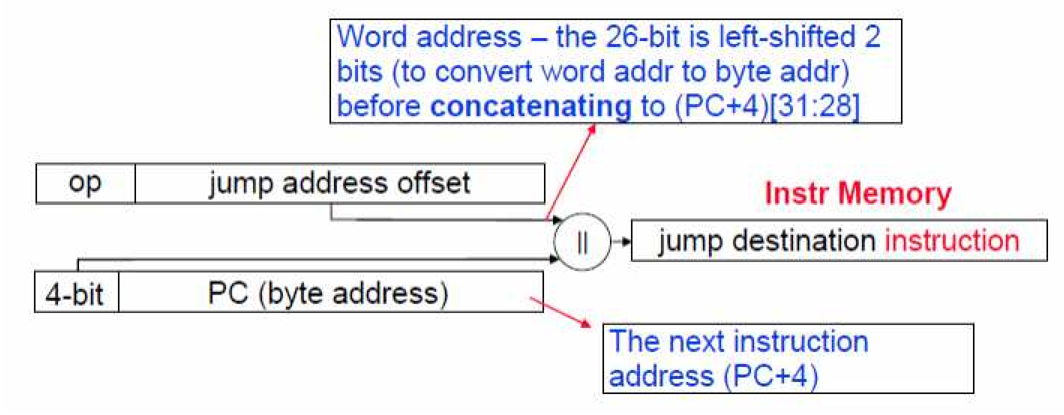

Byte address - the 16-bit is sign-extended to 32-bit before adding to base addr
op
rs
rt
data addr offset
Data Memory
+
word or byte operand
base address
Operand's memory address
Register File
Contain an address or pointer acting as a base address.

iii. PC-relative addressing: Perform operation on source and target register to determine next PC condition, the immediate is uses as address offset for next PC.


Word address – the 16-bit is sign-extended, then left-shifted 2 bits (to convert word addr to byte addr) before adding to PC+4
op
rs
rt
branch addr offset
Instr Memory
+
branch target instruction
PC + 4 (byte address)
The address is pointing to the next instruction (PC+4)

c) J-format

Pseudo-direct addressing: Perform operation by concatenating the upper bits of PC with the jump address.


Word address – the 26-bit is left-shifted 2 bits (to convert word addr to byte addr) before concatenating to (PC+4)[31:28]
op
jump address offset
Instr Memory
||
jump destination instruction
4-bit
PC (byte address)
The next instruction address (PC+4)

5.8 Supported Instructions Set

Instruction	Format	Addr. Mode	Machine Language						Register Transfer Notation	Assembly Format	Over flow
			OpCo de	Rs	Rt	Rd	Shamt	Func			
sll	R	Register	0x00	0	$rt	$rd	n	0x01	R[rd] =R[rs] << n	sll $rd, $rt, n	no
srl	R	Register	0x00	0	$rt	$rd	n	0x03	R[rd] =R[rs] >> n	srl $rd, $rt, n	no
sra	R	Register	0x00	0	$rt	$rd	n	0x04	R[rd] =R[rs] >>> n	sra $rd, $rt, n	no
jr	R	Register	0x00	$rs	0	0	0	0x0 A	PC = R[rs]	jr $rs	no
jalr	R	Register	0x00	$rs	0	0	0	0x0 B	PC = R[rs], R[31] = PC + 4	jalr $rs	no
mfhi	R	Register	0x00	0	0	$rd	0	0x10	R[rd] = HI	mfhi $rd	no
mthi	R	Register	0x00	$rs	0	0	0	0x11	HI = R[rs]	mthi $rs	no
mflo	R	Register	0x00	0	0	$rd	0	0x12	R[rd] = LO	mflo $rd	no
mtlo	R	Register	0x00	$rs	0	0	0	0x13	LO = R[rs]	mtlo $rs	no
mult	R	Register	0x00	$rs	$rt	0	0	0x24	HILO = R[rs] * R[rt]	mult $rs, $rt	no
multu	R	Register	0x00	$rs	$rt	0	0	0x24	HILO = U(R[rs]) * U(R[rt])	multu $rs, $rt	no
add	R	Register	0x00	$rs	$rt	$rd	0	0x20	R[rd] = R[rs] + R[rt]	add $rd, $rs, $rt	yes
addu	R	Register	0x00	$rs	$rt	$rd	0	0x21	R[rd] = U(R[rs]) + U(R[rt])	addu $rd, $rs, $rt	no
sub	R	Register	0x00	$rs	$rt	$rd	0	0x22	R[rd] = R[rs] - R[rt]	sub $rd, $rs, $rt	yes
subu	R	Register	0x00	$rs	$rt	$rd	0	0x23	R[rd] = U(R[rs]) - U(R[rt])	subu $rd, $rs, $rt	no
and	R	Register	0x00	$rs	$rt	$rd	0	0x24	R[rd] = R[rs] & R[rt]	and $rd, $rs, $rt	no
or	R	Register	0x00	$rs	$rt	$rd	0	0x25	R[rd] = R[rs] | R[rt]	or $rd, $rs, $rt	no
xor	R	Register	0x00	$rs	$rt	$rd	0	0x26	R[rd] = R[rs] ^ R[rt]	xor $rd, $rs, $rt	no
nor	R	Register	0x00	$rs	$rt	$rd	0	0x27	R[rd] = ~(R[rs] | R[rt])	nor $rd, $rs, $rt	no
slt	R	Register	0x00	$rs	$rt	$rd	0	0x2 A	R[rd] = (R[rs] < R[rt]) ? 1 : 0	slt $rd, $rs, $rt	no
sltu	R	Register	0x00	$rs	$rt	$rd	0	0x2 B	R[rd] = (U(R[rs]) < U(R[rt])) ? 1 : 0	sltu $rd, $rs, $rt	no
j	J	Pseudo-	0x02	JumpAddr (Label)					PC = {(PC+4) [31:28],	j label	no

		Direct					JumpAddr, 2’b00}		
jal	J	Pseudo-Direct	0x03	JumpAddr (Label)			PC = {(PC+4) [31:28], JumpAddr, 2’b00} R[31] = PC + 4	jal label	no
beq	I	PC-Relative	0x04	$rs	$rt	BranchAddr (Label)	PC = (R[rs] == R[rt]) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	beq $rs, $rt, label	no
bne	I	PC-Relative	0x05	$rs	$rt	BranchAddr (Label)	PC = (R[rs] != R[rt]) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	bne $rs, $rt, label	no
blez	I	PC-Relative	0x06	$rs	0	BranchAddr (Label)	PC = (R[rs] <=0) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	blez $rs, $rt, label	no
bgtz	I	PC-Relative	0x07	$rs	0	BranchAddr (Label)	PC = (R[rs] > 0) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	bgtz $rs, $rt, label	no
addi	I	Immediate	0x08	$rs	$rt	Imm	R[rt] = R[rs] + SE(Imm)	addi $rt, $rs, imm	yes
addiu	I	Immediate	0x09	$rs	$rt	Imm	R[rt] = U(R[rs]) + U(ZE(Imm))	addiu $rt, $rs, imm	no
slti	I	Immediate	0x0A	$rs	$rt	Imm	R[rt] = (R[rs] < SE(Imm)) ? 1 : 0	slti $rt, $rs, imm	no
sltiu	I	Immediate	0x0B	$rs	$rt	Imm	R[rt] = (U(R[rs]) < U(SE(Imm))) ? 1 : 0	sltiu $rt, $rs, imm	no
andi	I	Immediate	0x0C	$rs	$rt	Imm	R[rt] = R[rs] & ZE(Imm)	andi $rt, $rs, imm	no
ori	I	Immediate	0x0D	$rs	$rt	Imm	R[rt] = R[rs] | ZE(Imm)	ori $rt, $rs, imm	no
xori	I	Immediate	0x0E	$rs	$rt	Imm	R[rt] = R[rs] ^ ZE(Imm)	xori $rt, $rs, imm	no

lui	I	Immediate	0x0F	$rs	$rt	Imm			R[rt] = Imm << 16	lui $rt, imm	no
lw	I	Based-Displacement	0x23	$rs	$rt	Imm			R[rt] = MEM[R[rs] + SE(Imm)]	lw $rt, imm($rs)	no
sw	I	Based-Displacement	0x2B	$rs	$rt	Imm			MEM[R[rs] + SE(Imm)] = R[rt]	sw $rt, imm($rs)	no
mfc0		**Register**	**0x10**	**0x00**	**$rt**	**$rd**	**0x00**	**0x00**	**R[rt] = R[rd] (from CP0)**	**mfc0 $rt, $rd**	**no**
mtc0		**Register**	**0x10**	**0x04**	**$rt**	**$rd**	**0x00**	**0x00**	**R[rd] (from CP0) = R[rt]**	**mtc0 $rt, $rd**	**no**
eret		**Register**	**0x10**	**0x10**	**0x00**	**0x00**	**0x00**	**0x18**	**PC = R[epc] (from CP0)**	**eret**	**no**

Table 5.9 Instruction Set

Chapter 6 Micro Architecture Specification

6.1 Design Hierarchy and Partitioning

Chip Partitioning (Top Level) at Architecture Level	Unit Partitioning at Micro-Architecture Level	Block and Functional Block Partitioning at RTL (Micro-Architecture Level)
RISC32 Pipeline Processor (c_risc)	Datapath (u_dp)	Branch Predictor (b_bp_4way)
		Register File (b_rf)
		Interlock Control (b_itl_ctrl)
		Forward Control (b_fw_ctrl)
		32-bit Multiplier (b_mult32)
		ALB (b_alb)
		Coprocessor0(b_cp0)
	Controlpath (u_cp)	Main Control (b_main_ctrl)
		ALB Control (b_alb_ctrl)
	Cache (u_cache)	Cache(u_cache)

Table 6.1 Design Hierarchy

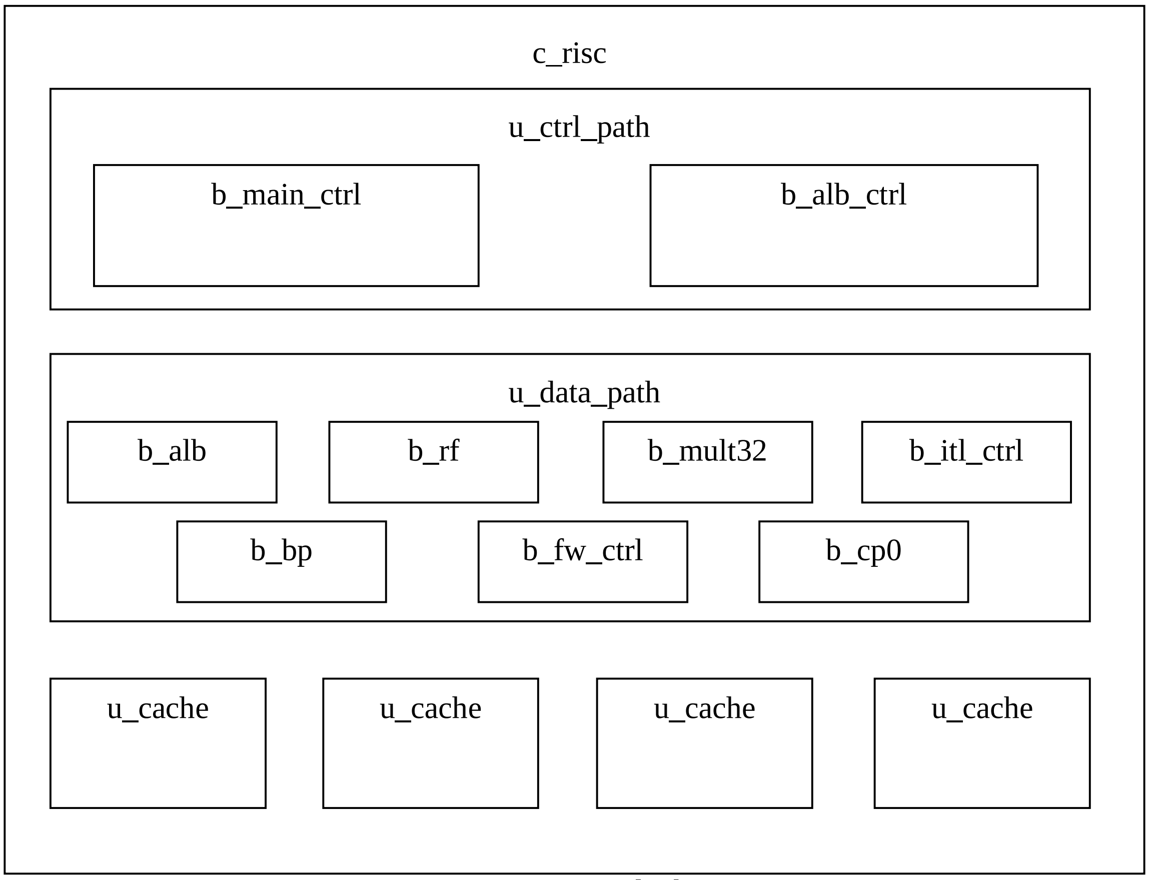
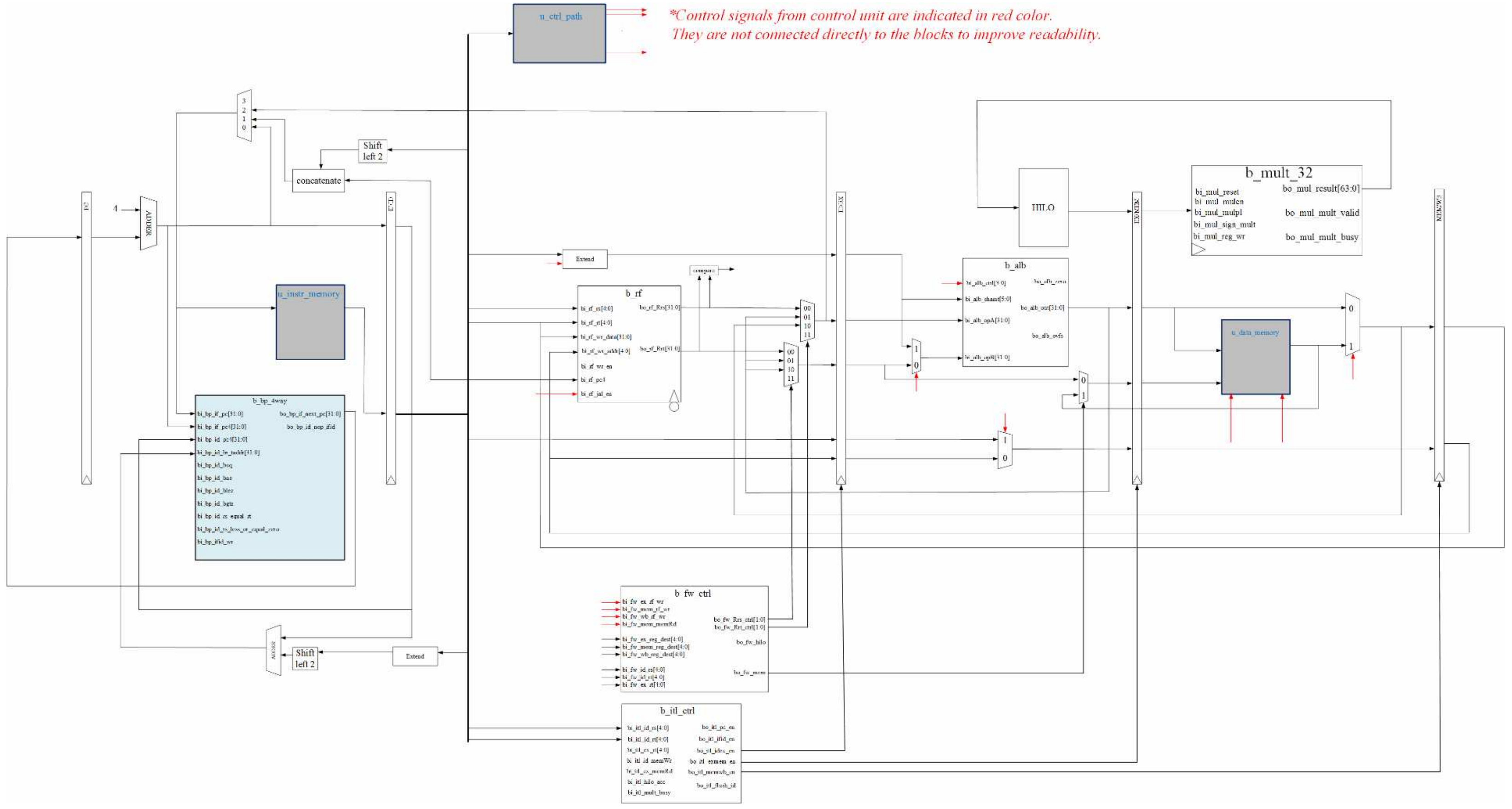


Figure 6.1 Block Partitioning

6.2 Micro-Architecture (Block Level)

6.2.1 Micro-Architecture without CP0



u_ctrl_path
*Control signals from control unit are indicated in red color.
They are not connected directly to the blocks to improve readability.
Shift left 2
concatenate
ADDER
u_instr_memory
b_bp_4way
Extend
b_rf
b_alu
HILO
b_mult_32
u_data_memory
b_fw_ctrl
b_itl_ctrl

Figure 6.2 System Micro-Architecture Without CP0

6.2.2 Micro-Architecture with CP0

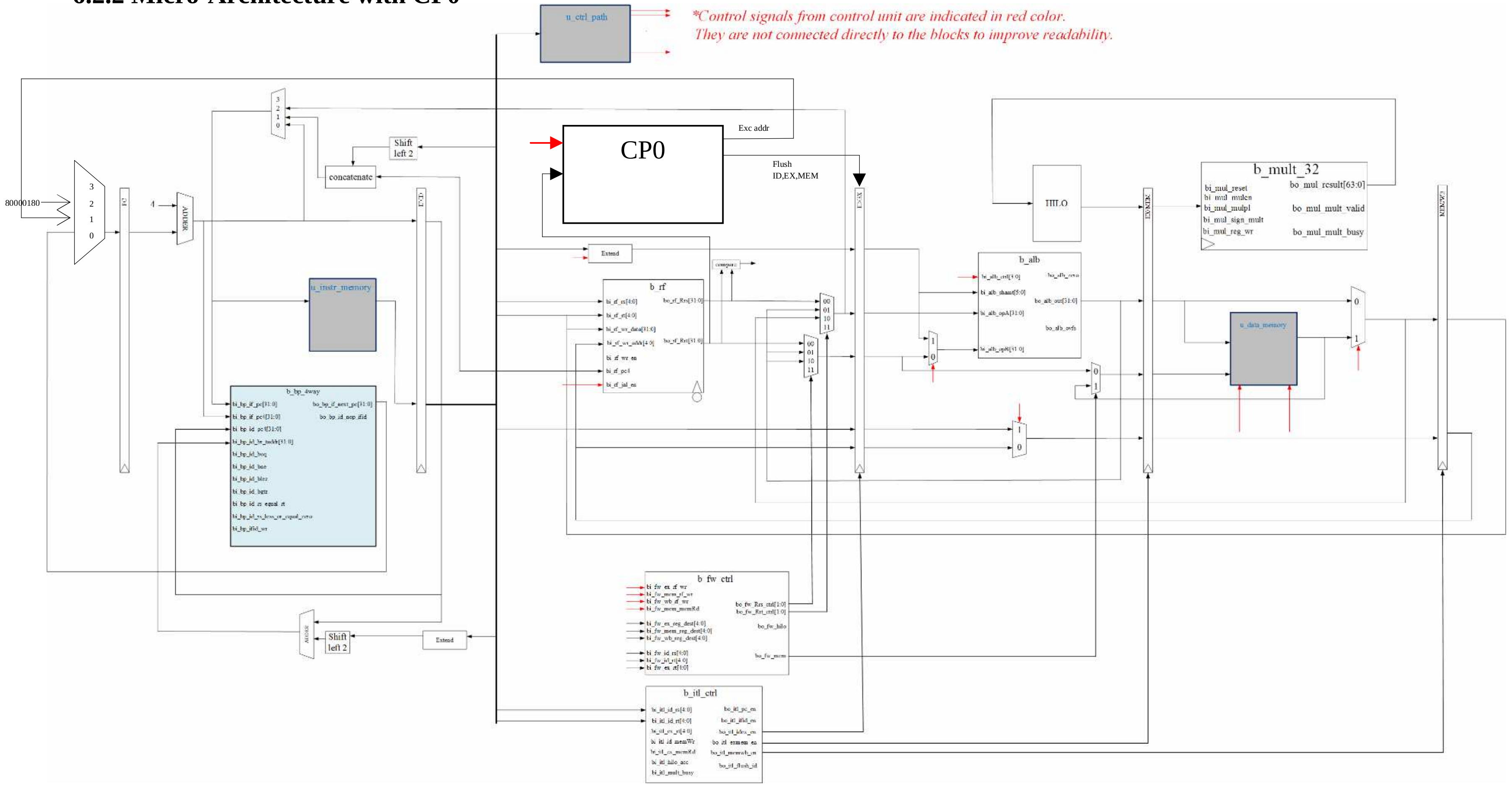
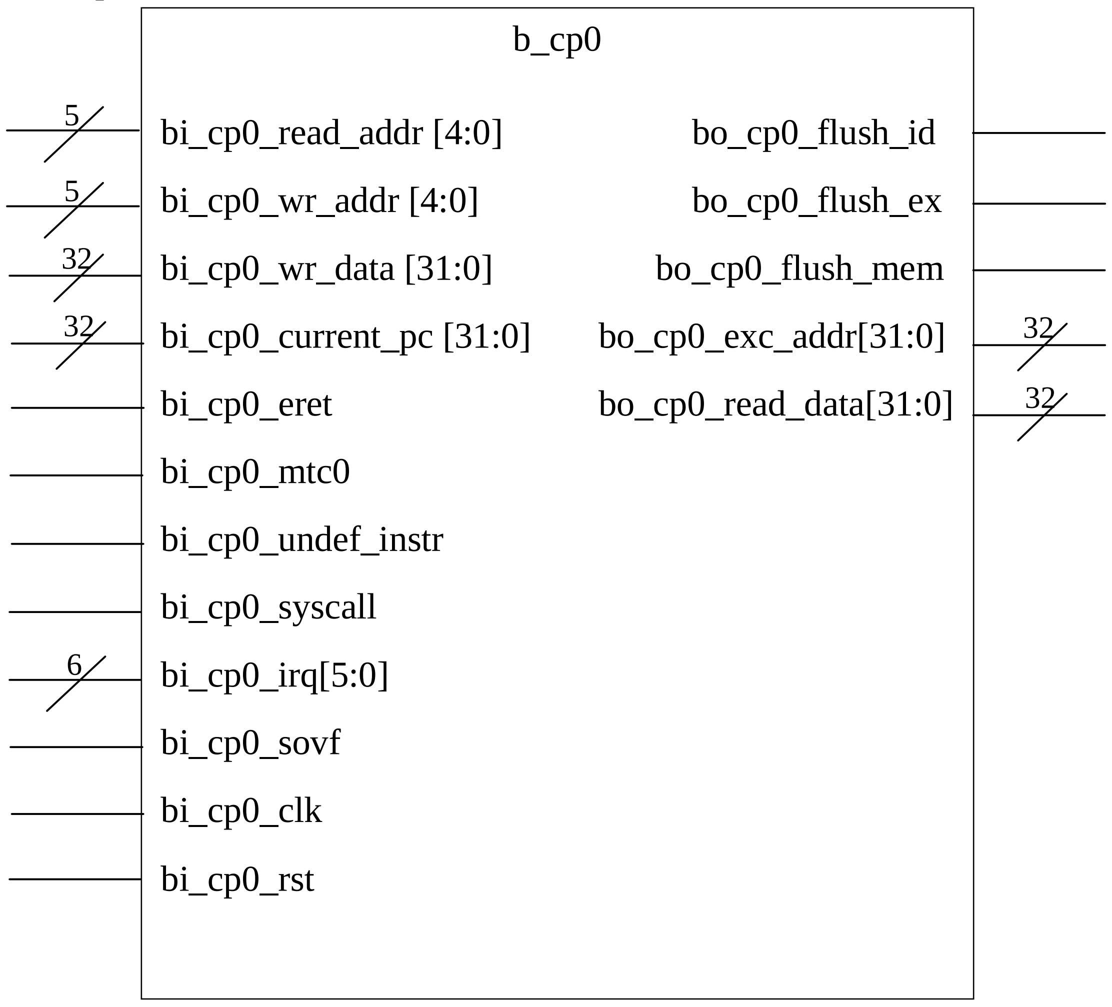


Figure 6.3 System Micro-Architecture With CP0

6.3 Coprocessor 0 Block

6.3.1 Coprocessor 0 Block interface



6.3.2 I/O Pin Description

b_cp0
Input:
Pin name : bi_cp0_read_addr[4:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(ID) → b_cp0 **Pin Function**: 5 bit rd address to indicate CP0 register file location.
Pin name : bi_cp0_wr_addr[4:0] **Pin Class** : Address **Registered**: No **Source->Destination**:Datapath(ID) → b_cp0 **Pin Function**: 5 bit rd address to indicate CP0 register file location.
Pin name : bi_cp0_wr_data[31:0] **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_cp0 **Pin Function**: 32 bit data to be stored to CP0 register file.
Pin name : bi_cp0_current_pc[31:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(ID) → b_cp0 **Pin Function**: 32 bit current Program Counter (PC) value.
Pin name : bi_cp0_eret **Pin Class** : Control **Registered**: No **Source->Destination**: Control → Datapath → b_cp0 **Pin Function**: Indicate current instruction is eret when asserted high.
Pin name : bi_cp0_mtc0 **Pin Class** : Control **Registered**: No **Source->Destination**: Control → Datapath → b_cp0 **Pin Function**: Indicate current instruction is mtc0 when asserted high.
Pin name : bi_cp0_undef_instr **Pin Class** : Control **Registered**: No **Source->Destination**: Control → Datapath → b_cp0 **Pin Function**: Indicate current instruction is undefined when asserted high.

Pin name : bi_cp0_syscall **Pin Class** : Control **Registered**: No **Source->Destination**: Control → Datapath → b_cp0 **Pin Function**: Indicate current instruction is syscall when asserted high.
Pin name : bi_cp0_irq[5:0] **Pin Class** : Control **Registered**: No **Source->Destination**: Datapath → b_cp0 **Pin Function**: Each bit indicates interrupt signal from external device.
Pin name : bi_cp0_sovf **Pin Class** : Control **Registered**: No **Source->Destination**: Control → Datapath → b_cp0 **Pin Function**: Indicate sign overflow has occurred when asserted high.
Pin name : bi_cp0_clk **Pin Class** : Global **Registered**: No **Source->Destination**: System → b_cp0 **Pin Function**: Clock signal for CP0
Pin name : bi_cp0_rst **Pin Class** : Global **Registered**: No **Source->Destination**: System → b_cp0 **Pin Function**: Reset signal for the CP0.
Output:
Pin name : bo_cp0_flush_id **Pin Class** : Control **Registered**: No **Source->Destination**: b_cp0 → Datapath **Pin Function**: Flush IF/ID pipe when asserted high.
Pin name : bo_cp0_flush_ex **Pin Class** : Control **Registered**: No **Source->Destination**: b_cp0 → Datapath **Pin Function**: Flush ID/EX pipe when asserted high.
Pin name : bo_cp0_flush_mem **Pin Class** : Control **Registered**: No **Source->Destination**: b_cp0 → Datapath **Pin Function**: Flush EX/MEM pipe when asserted high.

Pin name : bo_cp0_exc_addr[31:0]
Pin Class : Address
Registered: No
Source->Destination: b_cp0 → Datapath
Pin Function: Contain EPC value to be passed to PC.
Pin name : bo_cp0_read_data[31:0]
Pin Class : Data
Registered: No
Source->Destination: b_cp0 → Datapath
Pin Function: Data read out from CP0 register.

Table 6.2 CP0 I/O Pin Description

6.3.3 Internal Operation

CP0 is a block that used to process and store exception/interrupt information. CP0 is placed in ID stage. Once mtc0 is decoded, the control signal, address and dat will travel straight into CP0 block, which means CP0 will process and store exception/interrupt information at next clock cycle. While for mfc0, the signal and address will go straight into CP0 block in ID stage but the data will travel towards the end of WB stage then only it will store into register file at negative edge of clock. The data cannot direct store into register in ID stage to prevent 2 data write into register at the same time (one from CP0 output, one from WB stage).

Overflow Exception: Detected by ALB block in stage EX and overflow signal is generated. Flush If/ID, ID/EX and EX/MEM pipe to prevent wrong update of information into register file and data cache. Then jump to exception handler to handle exception.

Undefined Instruction Exception: Detected in stage ID by main control and undefined_instr signal is generated. Flush IF/ID and EX/MEM pipe to prevent update wrong information into register file and data cache. Then jump to exception handler to handle exception.

Syscall Exception: Detected in stage ID by main control and syscall signal is generated. Flush IF/ID pipe to prevent update wrong information into register file and data cache. Then jump to exception handler to handle exception.

I/O interrupt: Interrupt is asynchronous relative to a program execution. No need pipeline flushing, but let the current instruction in the stages complete their execution. Then jump to exception handler to handle exception.

Chapter 7 Verification

7.1 CP0 Test Program

The following program is designed to verify the functionality of CP0 block.

Instruction Address	Instruction Code	Instruction	Explanation
0x00400024	00008824	and $s1,$0,$0	Initialize cause register. Cause register = $13.
0x00400028	00000000	sll $zero,$zero,0	
0x0040002C	00000000	sll $zero,$zero,0	
0x00400030	40916800	mtc0 $s1,$13	
0x00400034	34116601	ori $s1,$0,0xff01	Initialize status register. status register = $12.
0x00400038	00000000	sll $zero,$zero,0	
0x0040003C	00000000	sll $zero,$zero,0	
0x00400040	40916000	mtc0 $s1,$12	
0x00400044	00000000	sll $zero,$zero,0	Nop
0x00400048	3C107FFF	lui $s0,0x7fff	$s0 = 7fff_0000
0x0040004C	3612FFFF	ori $s2,$s0,0xffff	$s2 = 7fff_ffff
0x00400050	3C148000	lui $s4,0x8000	$s4 = 8000_0000
0x00400054	36940300	ori $s4,$s4,0x0300	$s4 = 8000_0300 Address of selected case item in exception handler.
0x00400058	3C158000	lui $s5,0x8000	$s5 = 8000_0000
0x0040005C	36B501C0	ori $s5,$s5,0x01c0	$s5 = 8000_01C0 Address of clean_up in exception handler
0x00400060	00000000	sll $zero,$zero,0	Nop
0x00400064	FFFFFFFF	Undefined instruction	Creating undefined instruction to test exception.
0x00400068	0000000C	Syscall	Creating syscall to test exception.
0x0040006C	02524020	add $t0,$s2,$s2	Sign overflow
0x00400070	00000000	sll $zero,$zero,0	Nop
0x00400074	20120012	addi $s2,$zero,19	Create normal instruction and I/O interrupt occurs at same time.
0x00400078	00000000	sll $zero,$zero,0	Nop

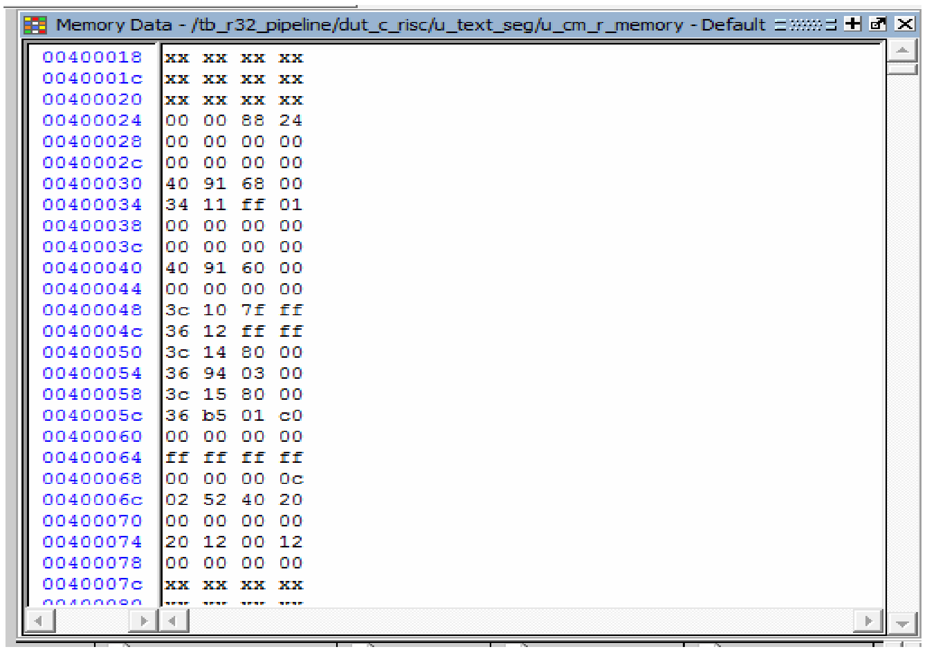
7.1.1 Exception Handler

Instruction Address	Instruction Code	Instruction	Explanation
0x80000180	0020D820	add $k1,$at,$0	Save register in data memory so exception handler can use it.
0x80000184	AC040000	sw $a0,0($zero)	
0x80000188	AC050004	sw $a1,4($zero)	
0x8000018C	401A6800	mfc0 $k0,$13	Move casue register to $k0
0x80000190	00000000	sll $zero,$zero,0	Nop
0x80000194	00000000	sll $zero,$zero,0	Nop
0x80000198	001A2082	srl $a0,$k0,2	Extract ExcCode field
0x8000019C	3084001F	andi $a0,$a0,0x1f	Mask cause register [6:2]
0x800001A0	0080502A	slt $t2,$a0,$0	Test if $a0 < 0
0x800001A4	15400006	bne $t2,$0,clean_up	If $a0 < 0 branch to clean_up
0x800001A8	288A000D	slti $t2,$a0,13	Test if $a0 > = 13
0x800001AC	11400004	beq $t2,$0,clean_up	If $a0 >= 13 branch to clean_up
0x800001B0	00844820	add $t1,$a0,$a0	Turn $a0 into byte address.
0x800001B4	01294820	add $t1,$t1,$t1	$t1 = 4*$t1
0x800001B8	01344820	add $t1,$t1,$s4	Determine address of ISR
0x800001BC	01200008	jr $t0	Jump to the address of selected case item
clean_up			
0x800001C0	401A7000	mfc0 $ko,$14	Move EPC to register file
0x800001C4	00000000	sll $zero,$zero,0	Nop
0x800001C8	00000000	sll $zero,$zero,0	Nop
0x800001CC	275A0004	addiu $k0,$k0	EPC + 4. Donot re-execute faulting instruction when return
0x800001D0	00000000	sll $zero,$zero,0	Nop
0x800001D4	00000000	sll $zero,$zero,0	Nop
0x800001D8	409A7000	mtc0 $k0,$14	Update EPC
0x800001DC	40806800	mtc0 $0,$13	Clear cause register
0x800001E0	401A6000	mfc0 $k0,$12	Move status register to register file

0x800001E4	00000000	sll $zero,$zero,0	Nop
0x800001E8	00000000	sll $zero,$zero,0	Nop
0x800001EC	335AFFFD	andi $k0,0xfffd	Mask status register
0x800001F0	375A0001	ori $k0,0x1	Set status[0] to 1
0x800001F4	00000000	sll $zero,$zero,0	Nop
0x800001F8	00000000	sll $zero,$zero,0	Nop
0x800001FC	409A6000	mtc0 $k0,$12	Update status register
0x80000200	8C040000	lw $a0,0($zero)	Restore register value
0x80000204	8C050004	lw $a1,4($zero)	Restore register value
0x80000208	03600820	add $at,$k1,$0	Restore register value
0x8000020C	42000018	eret	Eret. Return to normal program execution
0x80000210	00000000	sll $zero,$zero,0	Nop
TEST CASE SELECTED ITEM			
0x80000300	02A00008	jr $s5	Jump to clean_up
0x80000304	02A00008	jr $s5	Jump to clean_up
0x80000308	02A00008	jr $s5	Jump to clean_up
0x8000030C	02A00008	jr $s5	Jump to clean_up
0x80000310	02A00008	jr $s5	Jump to clean_up
0x80000314	02A00008	jr $s5	Jump to clean_up
0x80000318	02A00008	jr $s5	Jump to clean_up
0x8000031C	02A00008	jr $s5	Jump to clean_up
0x80000320	02A00008	jr $s5	Jump to clean_up
0x80000324	02A00008	jr $s5	Jump to clean_up
0x80000328	02A00008	jr $s5	Jump to clean_up
0x8000032C	02A00008	jr $s5	Jump to clean_up
0x80000330	02A00008	jr $s5	Jump to clean_up

7.1.2 Simulation Result

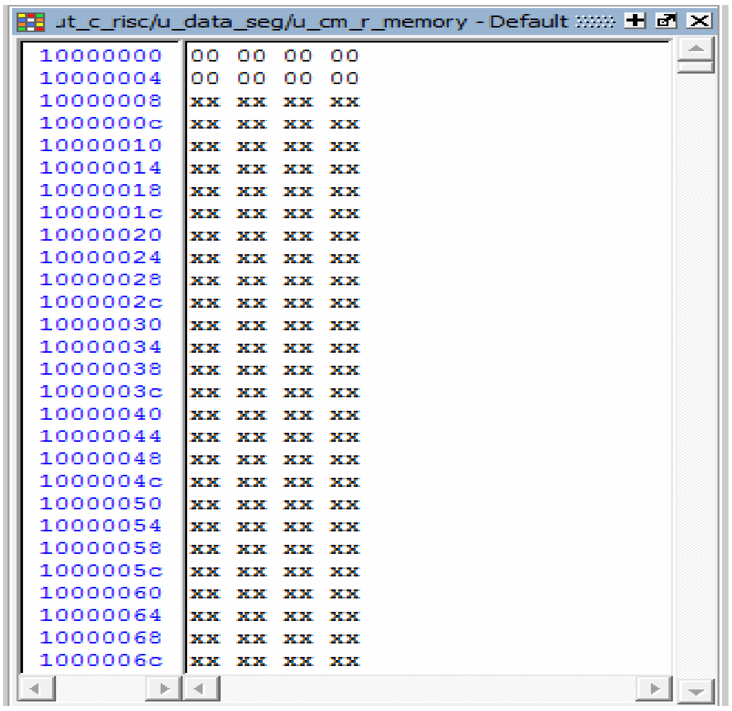
Instructions Cache (Text Segment)



Memory Data - /tb_r32_pipeline/dut_c_risc/u_text_seg/u_cm_r_memory - Default

```
00400018  xx xx xx xx
0040001c  xx xx xx xx
00400020  xx xx xx xx
00400024  00 00 88 24
00400028  00 00 00 00
0040002c  00 00 00 00
00400030  40 91 68 00
00400034  34 11 ff 01
00400038  00 00 00 00
0040003c  00 00 00 00
00400040  40 91 60 00
00400044  00 00 00 00
00400048  3c 10 7f ff
0040004c  36 12 ff ff
00400050  3c 14 80 00
00400054  36 94 03 00
00400058  3c 15 80 00
0040005c  36 b5 01 c0
00400060  00 00 00 00
00400064  ff ff ff ff
00400068  00 00 00 0c
0040006c  02 52 40 20
00400070  00 00 00 00
00400074  20 12 00 12
00400078  00 00 00 00
0040007c  xx xx xx xx
```

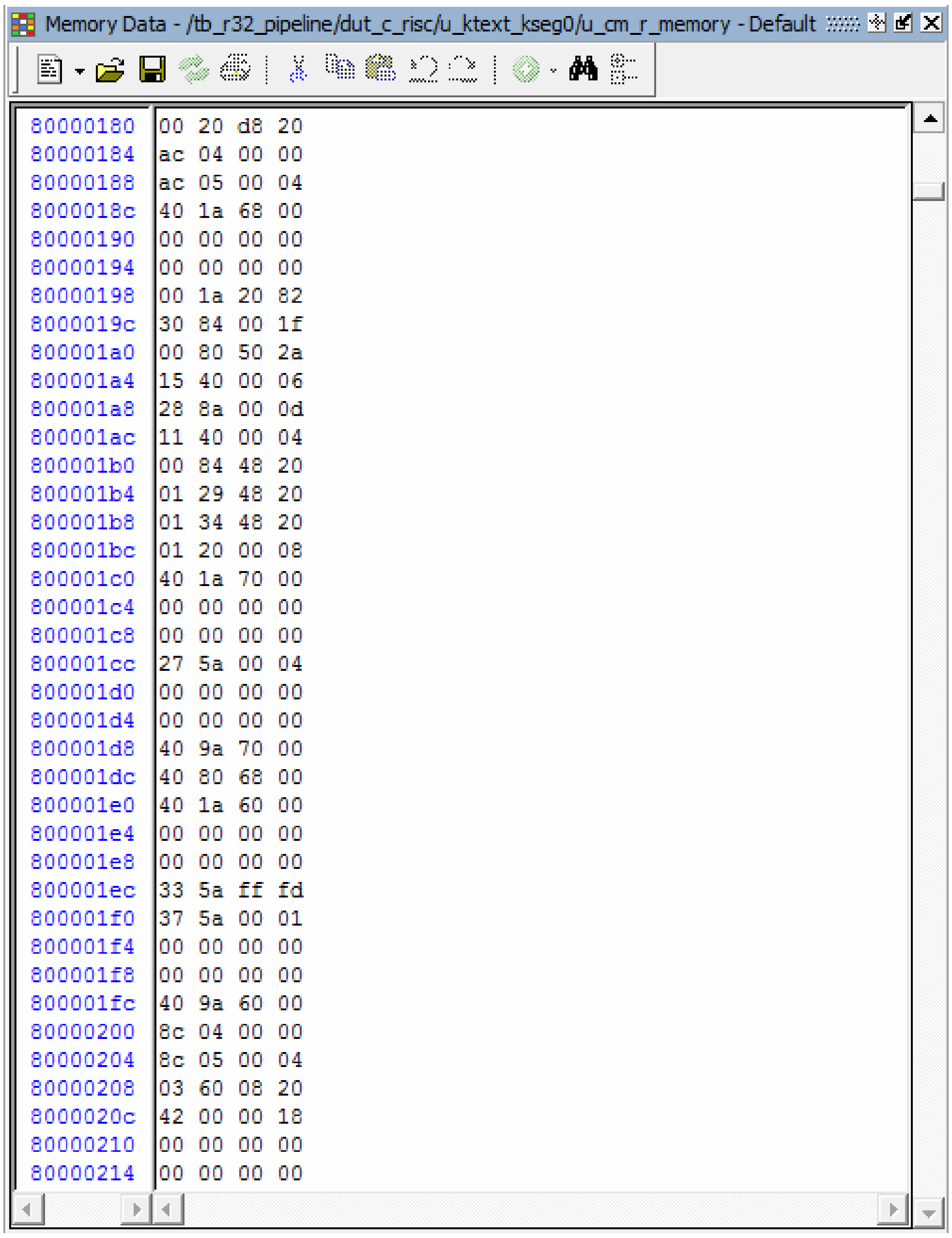
Data Cache (Data Segment)



ut_c_risc/u_data_seg/u_cm_r_memory - Default

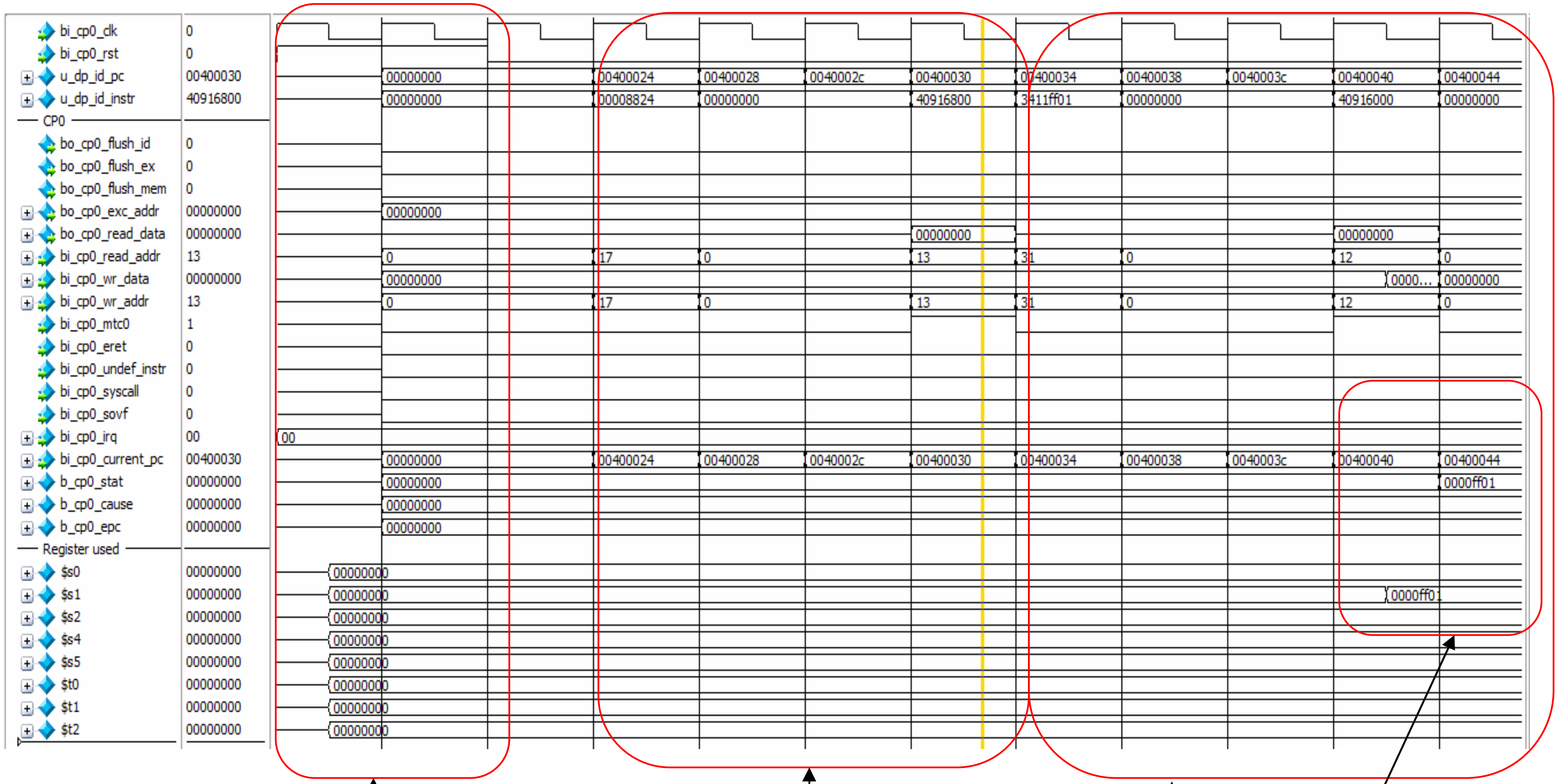
```
10000000  00 00 00 00
10000004  00 00 00 00
10000008  xx xx xx xx
1000000c  xx xx xx xx
10000010  xx xx xx xx
10000014  xx xx xx xx
10000018  xx xx xx xx
1000001c  xx xx xx xx
10000020  xx xx xx xx
10000024  xx xx xx xx
10000028  xx xx xx xx
1000002c  xx xx xx xx
10000030  xx xx xx xx
10000034  xx xx xx xx
10000038  xx xx xx xx
1000003c  xx xx xx xx
10000040  xx xx xx xx
10000044  xx xx xx xx
10000048  xx xx xx xx
1000004c  xx xx xx xx
10000050  xx xx xx xx
10000054  xx xx xx xx
10000058  xx xx xx xx
1000005c  xx xx xx xx
10000060  xx xx xx xx
10000064  xx xx xx xx
10000068  xx xx xx xx
1000006c  xx xx xx xx
```

Exception Handler (ktext Segment)


Memory Data - /tb_r32_pipeline/dut_c_risc/u_ktext_kseg0/u_cm_r_memory - Default

```
80000180  00 20 d8 20
80000184  ac 04 00 00
80000188  ac 05 00 04
8000018c  40 1a 68 00
80000190  00 00 00 00
80000194  00 00 00 00
80000198  00 1a 20 82
8000019c  30 84 00 1f
800001a0  00 80 50 2a
800001a4  15 40 00 06
800001a8  28 8a 00 0d
800001ac  11 40 00 04
800001b0  00 84 48 20
800001b4  01 29 48 20
800001b8  01 34 48 20
800001bc  01 20 00 08
800001c0  40 1a 70 00
800001c4  00 00 00 00
800001c8  00 00 00 00
800001cc  27 5a 00 04
800001d0  00 00 00 00
800001d4  00 00 00 00
800001d8  40 9a 70 00
800001dc  40 80 68 00
800001e0  40 1a 60 00
800001e4  00 00 00 00
800001e8  00 00 00 00
800001ec  33 5a ff fd
800001f0  37 5a 00 01
800001f4  00 00 00 00
800001f8  00 00 00 00
800001fc  40 9a 60 00
80000200  8c 04 00 00
80000204  8c 05 00 04
80000208  03 60 08 20
8000020c  42 00 00 18
80000210  00 00 00 00
80000214  00 00 00 00
```

Test Program waveform



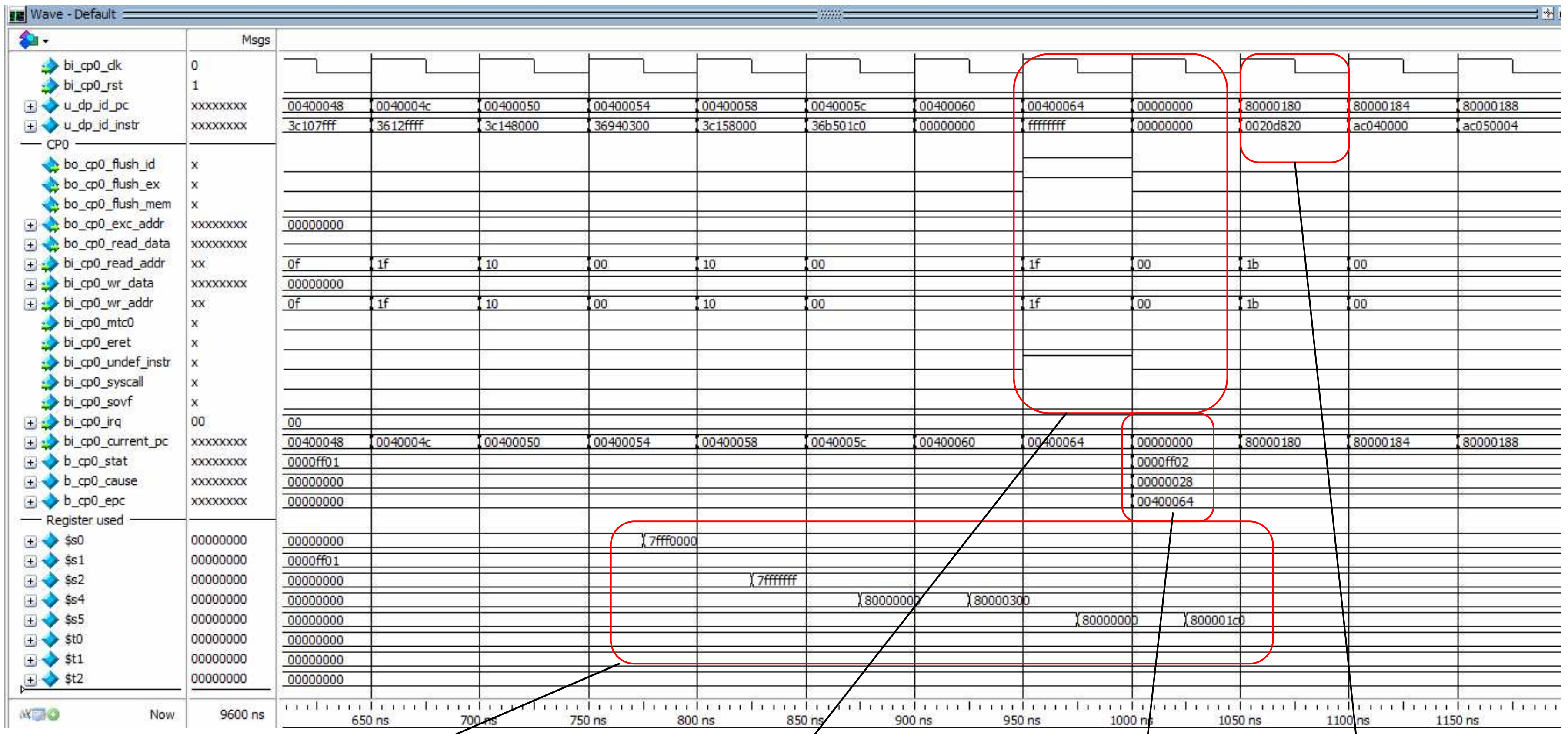
bi_cp0_clk	0
bi_cp0_rst	0
u_dp_id_pc	00400030
u_dp_id_instr	40916800
CP0	
bo_cp0_flush_id	0
bo_cp0_flush_ex	0
bo_cp0_flush_mem	0
bo_cp0_exc_addr	00000000
bo_cp0_read_data	00000000
bi_cp0_read_addr	13
bi_cp0_wr_data	00000000
bi_cp0_wr_addr	13
bi_cp0_mtc0	1
bi_cp0_eret	0
bi_cp0_undef_instr	0
bi_cp0_syscall	0
bi_cp0_sovf	0
bi_cp0_irq	00
bi_cp0_current_pc	00400030
b_cp0_stat	00000000
b_cp0_cause	00000000
b_cp0_epc	00000000
Register used	
$s0	00000000
$s1	00000000
$s2	00000000
$s4	00000000
$s5	00000000
$t0	00000000
$t1	00000000
$t2	00000000

Reset for 2 clock cycle

Initialize cause register to 0.
Move $s1 to cause $13.
$s1 = 0

Initialize status register to 0xff01.

Immediately update after $s1 become 0xff01

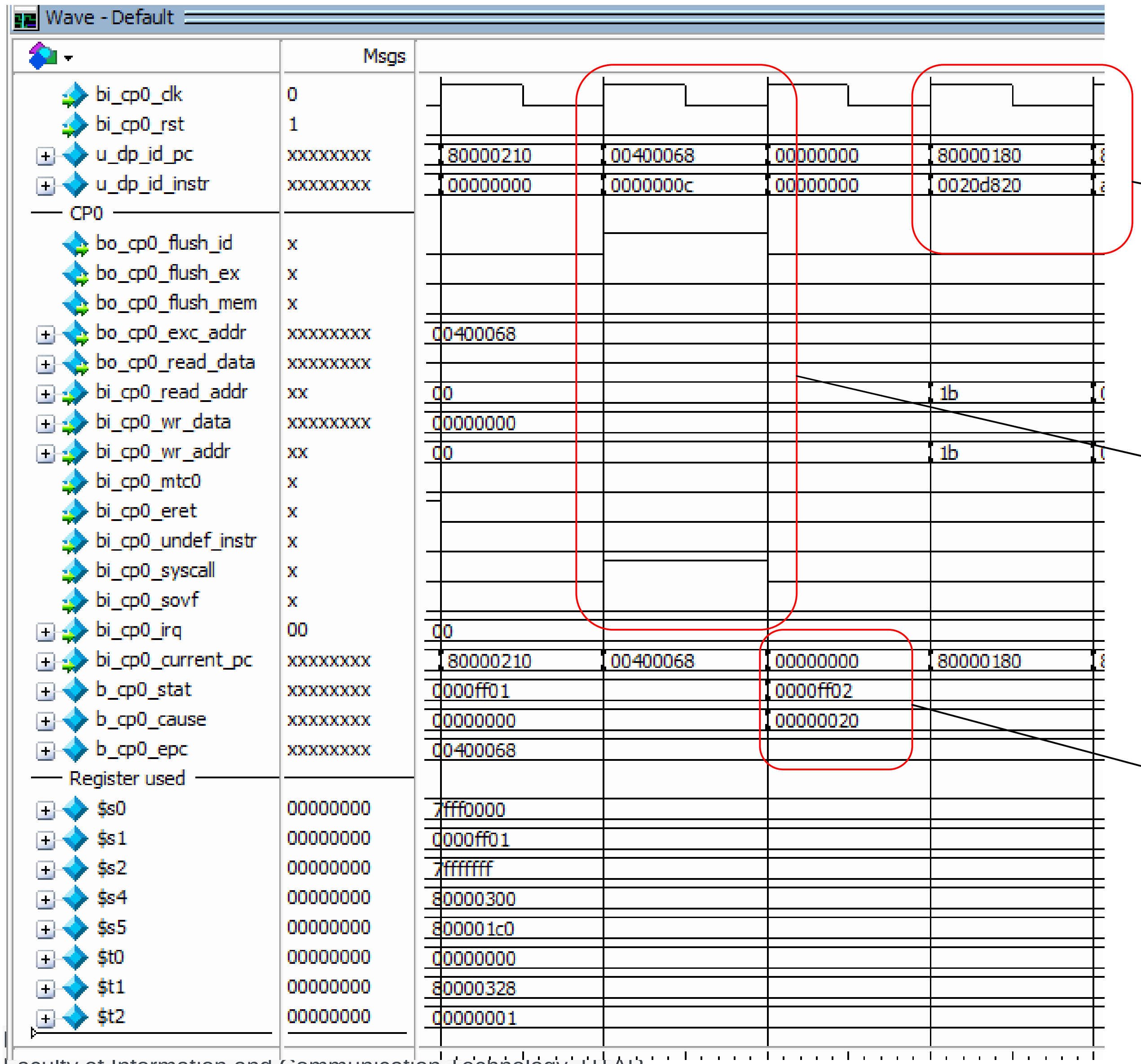


Using load upper immediate to create 7fff_ffff, 80000300 and 800001c0 and save in register $s2, $s4 and $s5

Undefined instruction detected. Flush IF/ID and ID/EX pipe

Update status, cause and EPC register

Go to Exception Handler at address 80000180

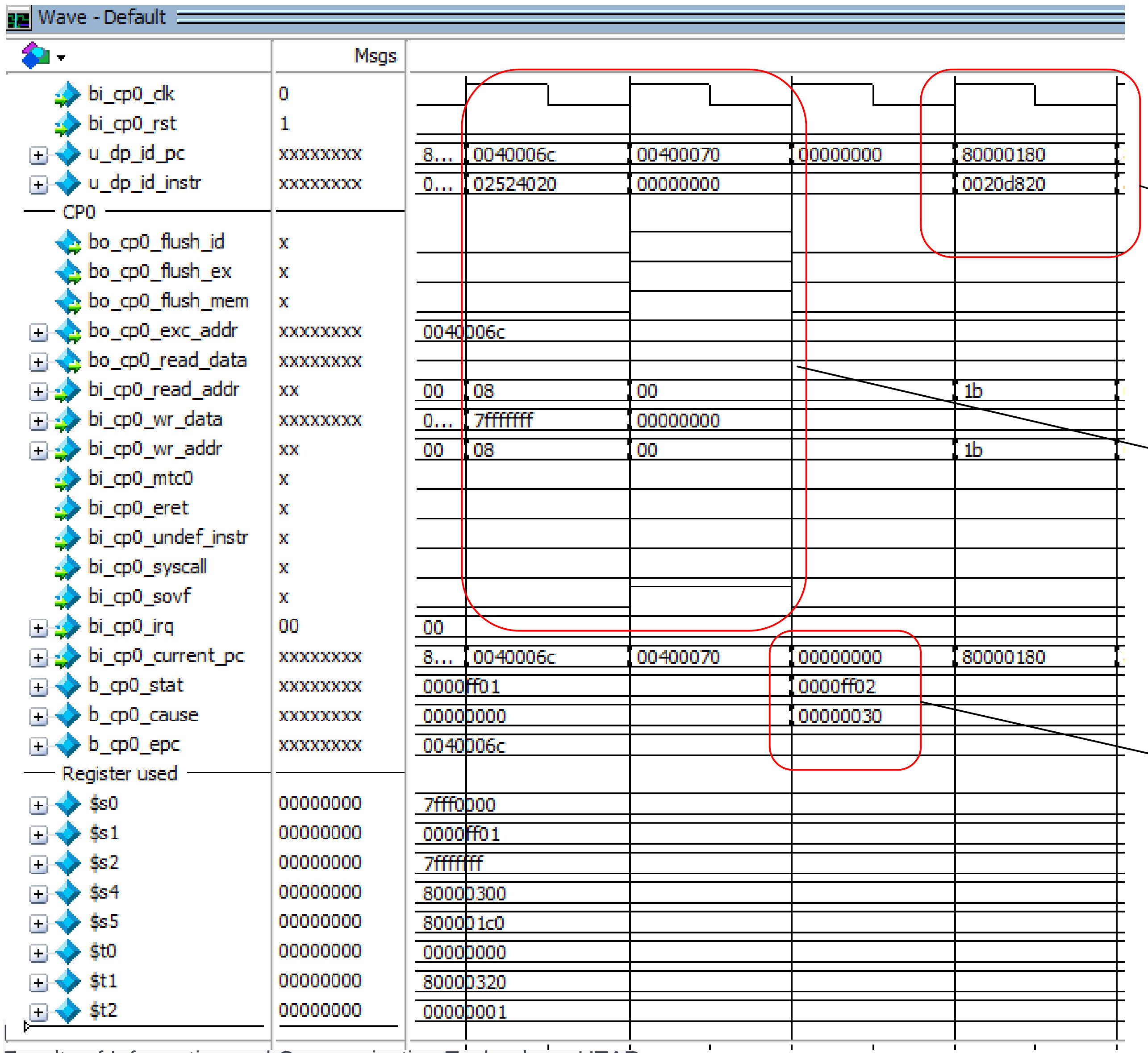


	Msgs
bi_cp0_clk	0
bi_cp0_rst	1
u_dp_id_pc	xxxxxxxx
u_dp_id_instr	xxxxxxxx
CP0	
bo_cp0_flush_id	x
bo_cp0_flush_ex	x
bo_cp0_flush_mem	x
bo_cp0_exc_addr	xxxxxxxx
bo_cp0_read_data	xxxxxxxx
bi_cp0_read_addr	xx
bi_cp0_wr_data	xxxxxxxx
bi_cp0_wr_addr	xx
bi_cp0_mtc0	x
bi_cp0_eret	x
bi_cp0_undef_instr	x
bi_cp0_syscall	x
bi_cp0_sovf	x
bi_cp0_irq	00
bi_cp0_current_pc	xxxxxxxx
b_cp0_stat	xxxxxxxx
b_cp0_cause	xxxxxxxx
b_cp0_epc	xxxxxxxx
Register used	
$s0	00000000
$s1	00000000
$s2	00000000
$s4	00000000
$s5	00000000
$t0	00000000
$t1	00000000
$t2	00000000

Go to Exception Handler at address 80000180

Syscall instruction detected. Flush IF/IF pipeline.

Update status, cause and EPC register

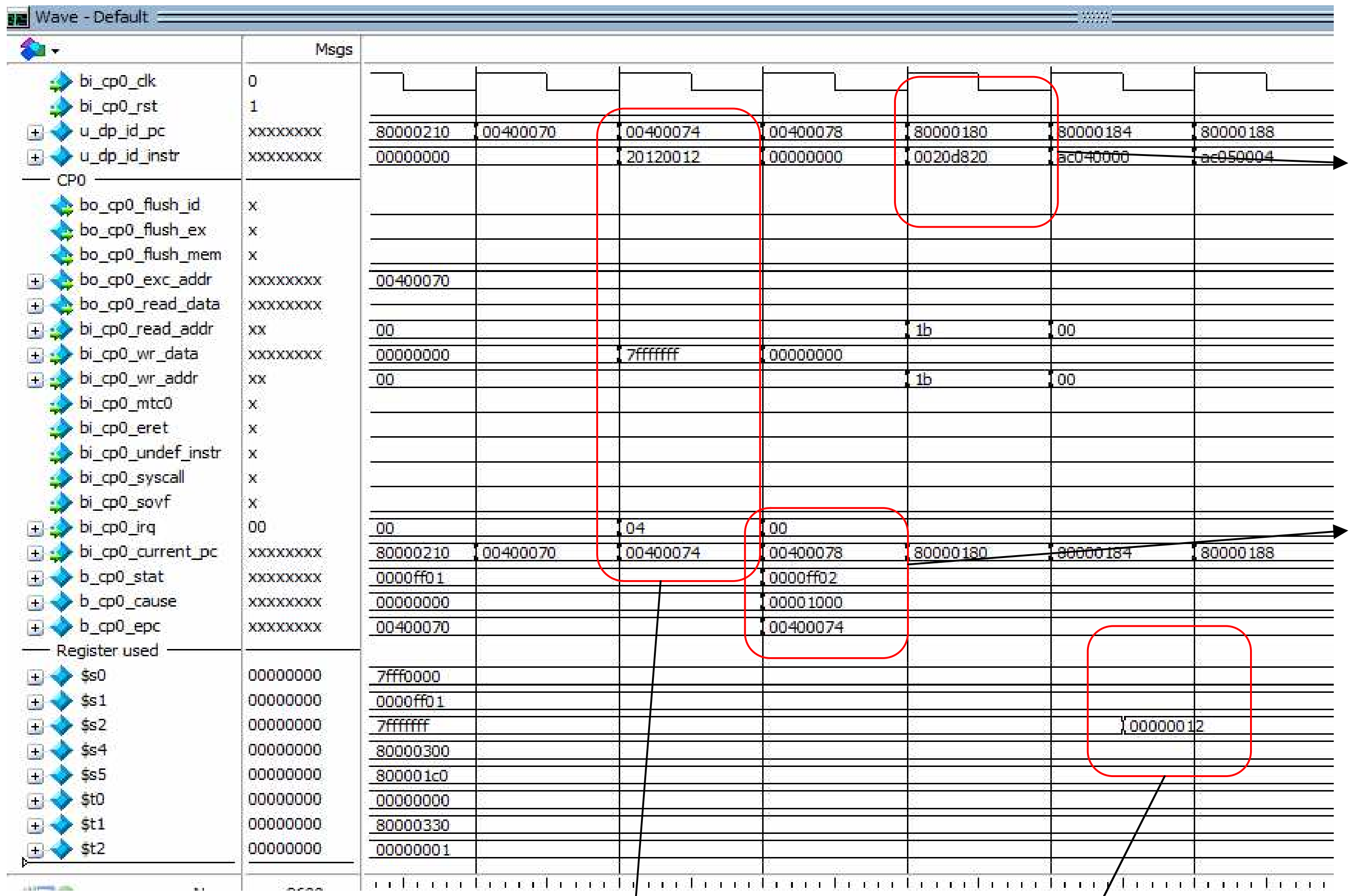

Wave - Default

	Msgs					
bi_cp0_clk	0					
bi_cp0_rst	1					
u_dp_id_pc	xxxxxxxx	8...	0040006c	00400070	00000000	80000180
u_dp_id_instr	xxxxxxxx	0...	02524020	00000000		0020d820
CP0						
bo_cp0_flush_id	x					
bo_cp0_flush_ex	x					
bo_cp0_flush_mem	x					
bo_cp0_exc_addr	xxxxxxxx	0040006c				
bo_cp0_read_data	xxxxxxxx					
bi_cp0_read_addr	xx	00	08	00		1b
bi_cp0_wr_data	xxxxxxxx	0...	7fffffff	00000000		
bi_cp0_wr_addr	xx	00	08	00		1b
bi_cp0_mtc0	x					
bi_cp0_eret	x					
bi_cp0_undef_instr	x					
bi_cp0_syscall	x					
bi_cp0_sovf	x					
bi_cp0_irq	00	00				
bi_cp0_current_pc	xxxxxxxx	8...	0040006c	00400070	00000000	80000180
b_cp0_stat	xxxxxxxx	0000ff01			0000ff02	
b_cp0_cause	xxxxxxxx	00000000			00000030	
b_cp0_epc	xxxxxxxx	0040006c				
Register used						
$s0	00000000	7fff0000				
$s1	00000000	0000ff01				
$s2	00000000	7fffffff				
$s4	00000000	80000300				
$s5	00000000	800001c0				
$t0	00000000	00000000				
$t1	00000000	80000320				
$t2	00000000	00000001				

Go to Exception Handler at address 80000180

Add 7fff_ffff with 7fff_ffff. Sign overflow detected. Flush IF/ID, ID/EX and EX/MEM pipe.

Update status, cause and EPC register

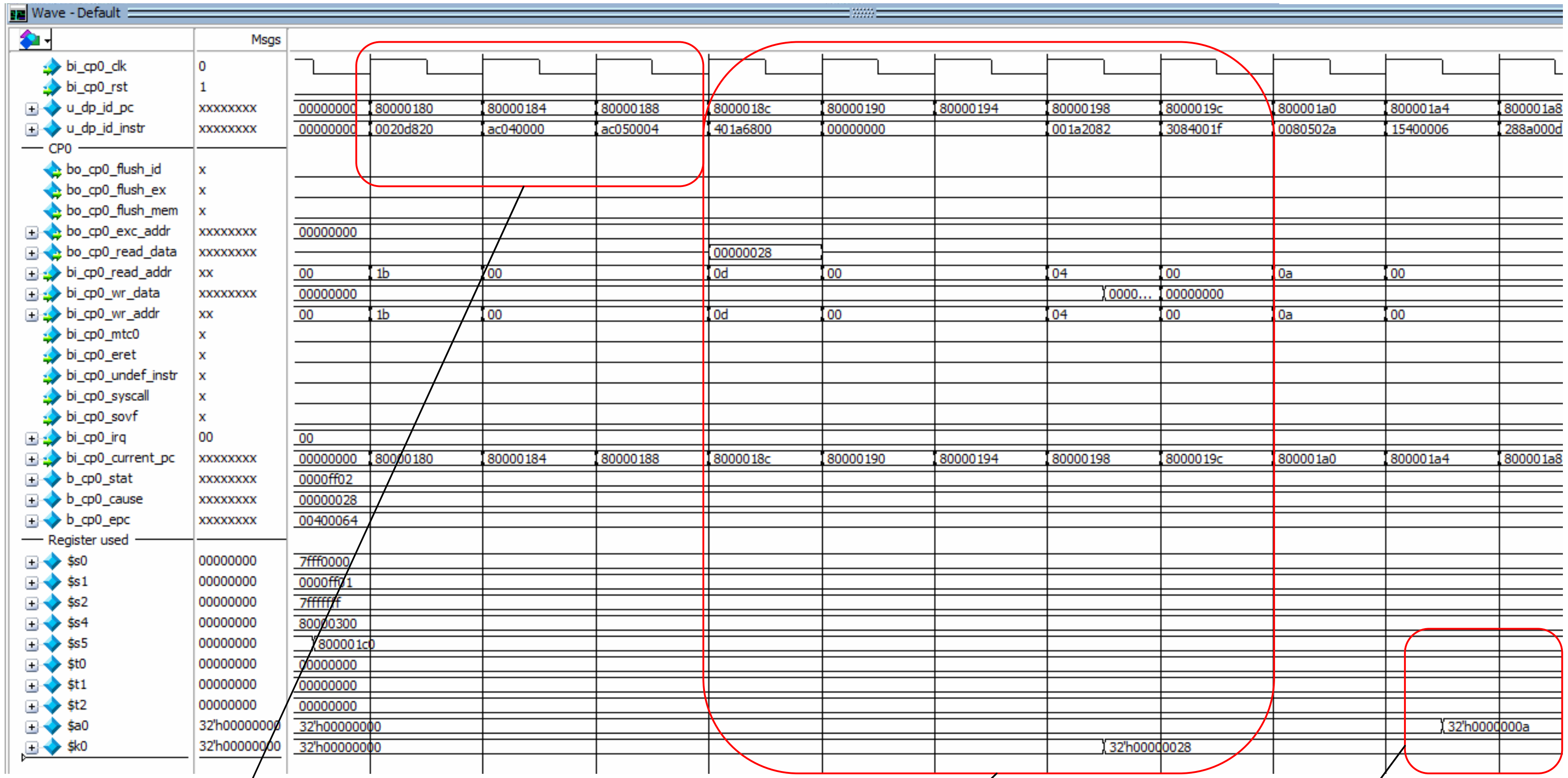

Wave - Default
Msgs
bi_cp0_clk 0
bi_cp0_rst 1
u_dp_id_pc xxxxxxxx 80000210 00400070 00400074 00400078 80000180 80000184 80000188
u_dp_id_instr xxxxxxxx 00000000 20120012 00000000 0020d820 ac040000 ac050004
CP0
bo_cp0_flush_id x
bo_cp0_flush_ex x
bo_cp0_flush_mem x
bo_cp0_exc_addr xxxxxxxx 00400070
bo_cp0_read_data xxxxxxxx
bi_cp0_read_addr xx 00 1b 00
bi_cp0_wr_data xxxxxxxx 00000000 7fffffff 00000000
bi_cp0_wr_addr xx 00 1b 00
bi_cp0_mtc0 x
bi_cp0_eret x
bi_cp0_undef_instr x
bi_cp0_syscall x
bi_cp0_sovf x
bi_cp0_irq 00 00 04 00
bi_cp0_current_pc xxxxxxxx 80000210 00400070 00400074 00400078 80000180 80000184 80000188
b_cp0_stat xxxxxxxx 0000ff01 0000ff02
b_cp0_cause xxxxxxxx 00000000 00001000
b_cp0_epc xxxxxxxx 00400070 00400074
Register used
$s0 00000000 7fff0000
$s1 00000000 0000ff01
$s2 00000000 7fffffff 00000012
$s4 00000000 80000300
$s5 00000000 800001c0
$t0 00000000 00000000
$t1 00000000 80000330
$t2 00000000 00000001

Go to Exception Handler at address 80000180

Update status, cause and EPC register

Execute normal instruction. Add 19 to $s2. At the same time I/O interrupt come in. Processor finishes normal instruction and goes to exception handler.

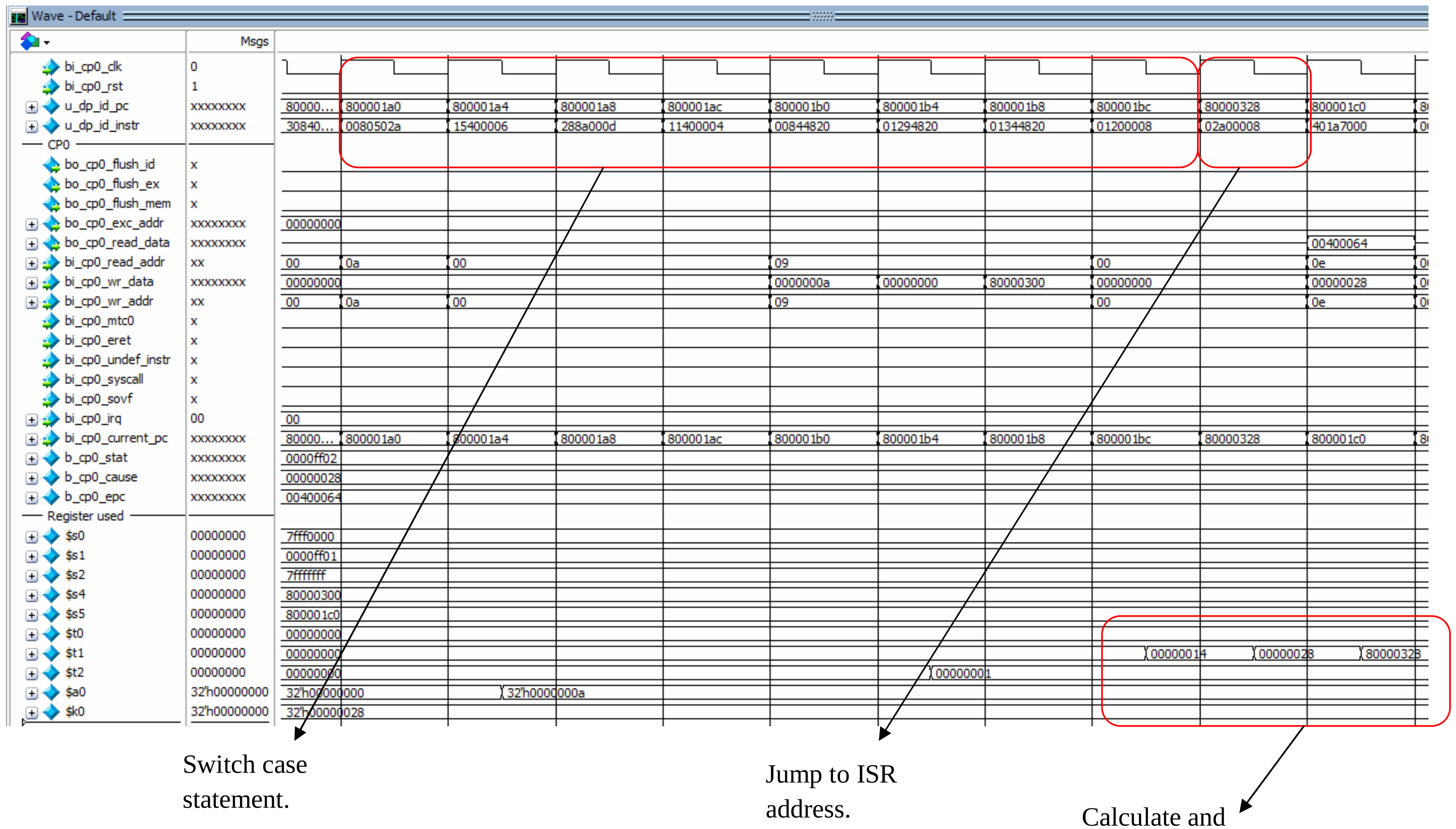
Exception Handler Waveform



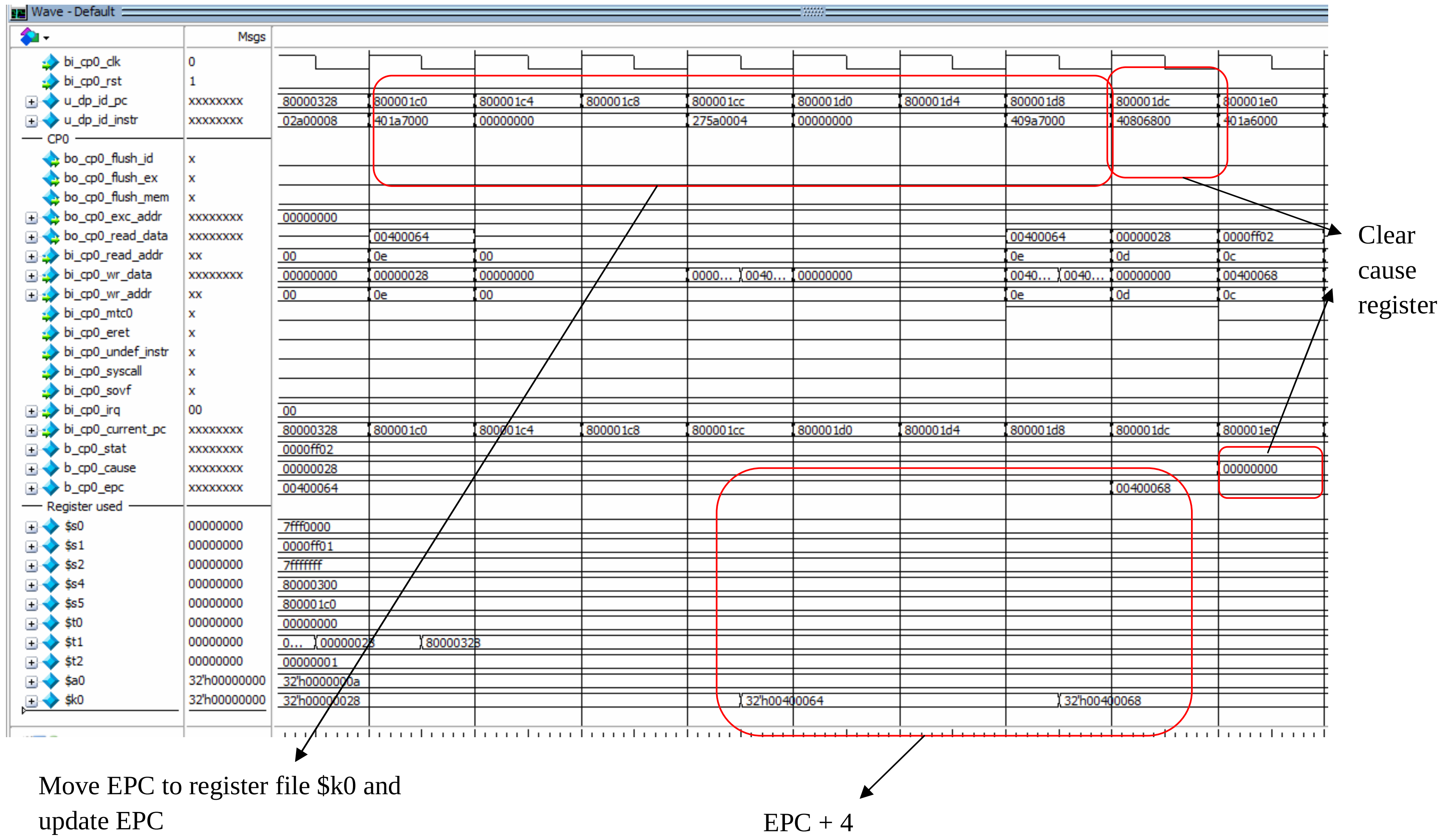
Save register $a0 and $a1 before use.

Move cause register to $k0, extract exception code and mask it.

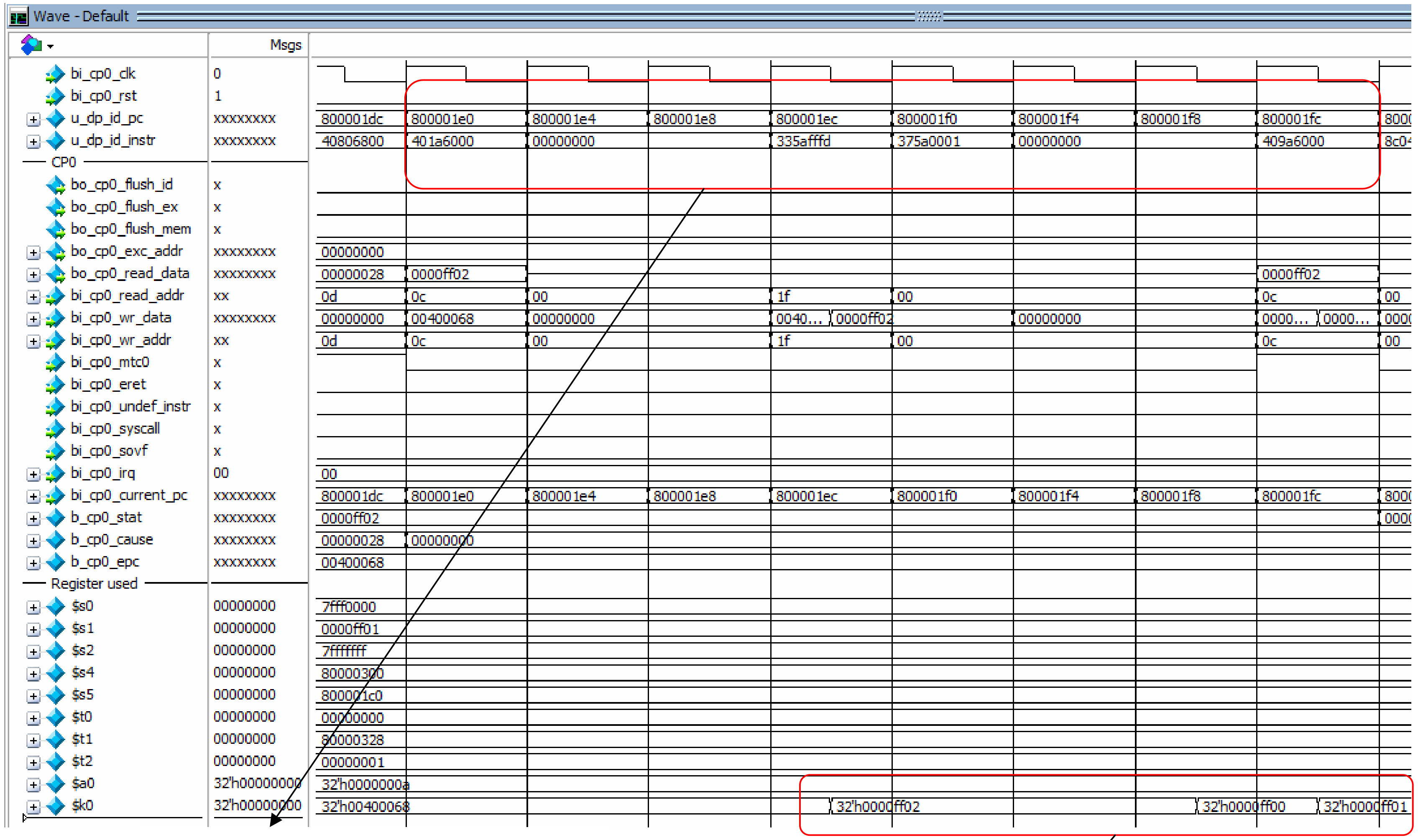
Extracted and masked value of exception code.



Signal	Msgs
bi_cp0_clk	0
bi_cp0_rst	1
u_dp_id_pc	xxxxxxxx
u_dp_id_instr	xxxxxxxx
CP0	
bo_cp0_flush_id	x
bo_cp0_flush_ex	x
bo_cp0_flush_mem	x
bo_cp0_exc_addr	xxxxxxxx
bo_cp0_read_data	xxxxxxxx
bi_cp0_read_addr	xx
bi_cp0_wr_data	xxxxxxxx
bi_cp0_wr_addr	xx
bi_cp0_mtc0	x
bi_cp0_eret	x
bi_cp0_undef_instr	x
bi_cp0_syscall	x
bi_cp0_sovf	x
bi_cp0_irq	00
bi_cp0_current_pc	xxxxxxxx
b_cp0_stat	xxxxxxxx
b_cp0_cause	xxxxxxxx
b_cp0_epc	xxxxxxxx
Register used	
$s0	00000000
$s1	00000000
$s2	00000000
$s4	00000000
$s5	00000000
$t0	00000000
$t1	00000000
$t2	00000000
$a0	32'h00000000
$k0	32'h00000000



```
bi_cp0_clk          0
bi_cp0_rst          1
u_dp_id_pc          xxxxxxxx   80000328  800001c0  800001c4  800001c8  800001cc  800001d0  800001d4  800001d8  800001dc  800001e0
u_dp_id_instr       xxxxxxxx   02a00008  401a7000  00000000            275a0004  00000000            409a7000  40806800  401a6000
CP0
bo_cp0_flush_id     x
bo_cp0_flush_ex     x
bo_cp0_flush_mem    x
bo_cp0_exc_addr     xxxxxxxx   00000000
bo_cp0_read_data    xxxxxxxx             00400064                                                    00400064  00000028  0000ff02
bi_cp0_read_addr    xx         00        0e        00                                                0e        0d        0c
bi_cp0_wr_data      xxxxxxxx   00000000  00000028  00000000            0000...  0040...  00000000    0040...  0040...  00000000  00400068
bi_cp0_wr_addr      xx         00        0e        00                                                0e        0d        0c
bi_cp0_mtc0         x
bi_cp0_eret         x
bi_cp0_undef_instr  x
bi_cp0_syscall      x
bi_cp0_sovf         x
bi_cp0_irq          00         00
bi_cp0_current_pc   xxxxxxxx   80000328  800001c0  800001c4  800001c8  800001cc  800001d0  800001d4  800001d8  800001dc  800001e0
b_cp0_stat          xxxxxxxx   0000ff02
b_cp0_cause         xxxxxxxx   00000028                                                                                  00000000
b_cp0_epc           xxxxxxxx   00400064                                                                        00400068
Register used
$s0                 00000000   7fff0000
$s1                 00000000   0000ff01
$s2                 00000000   7fffffff
$s4                 00000000   80000300
$s5                 00000000   800001c0
$t0                 00000000   00000000
$t1                 00000000   0...  00000028  80000328
$t2                 00000000   00000001
$a0                 32'h00000000   32'h0000000a
$k0                 32'h00000000   32'h00000028        32'h00400064                    32'h00400068
```

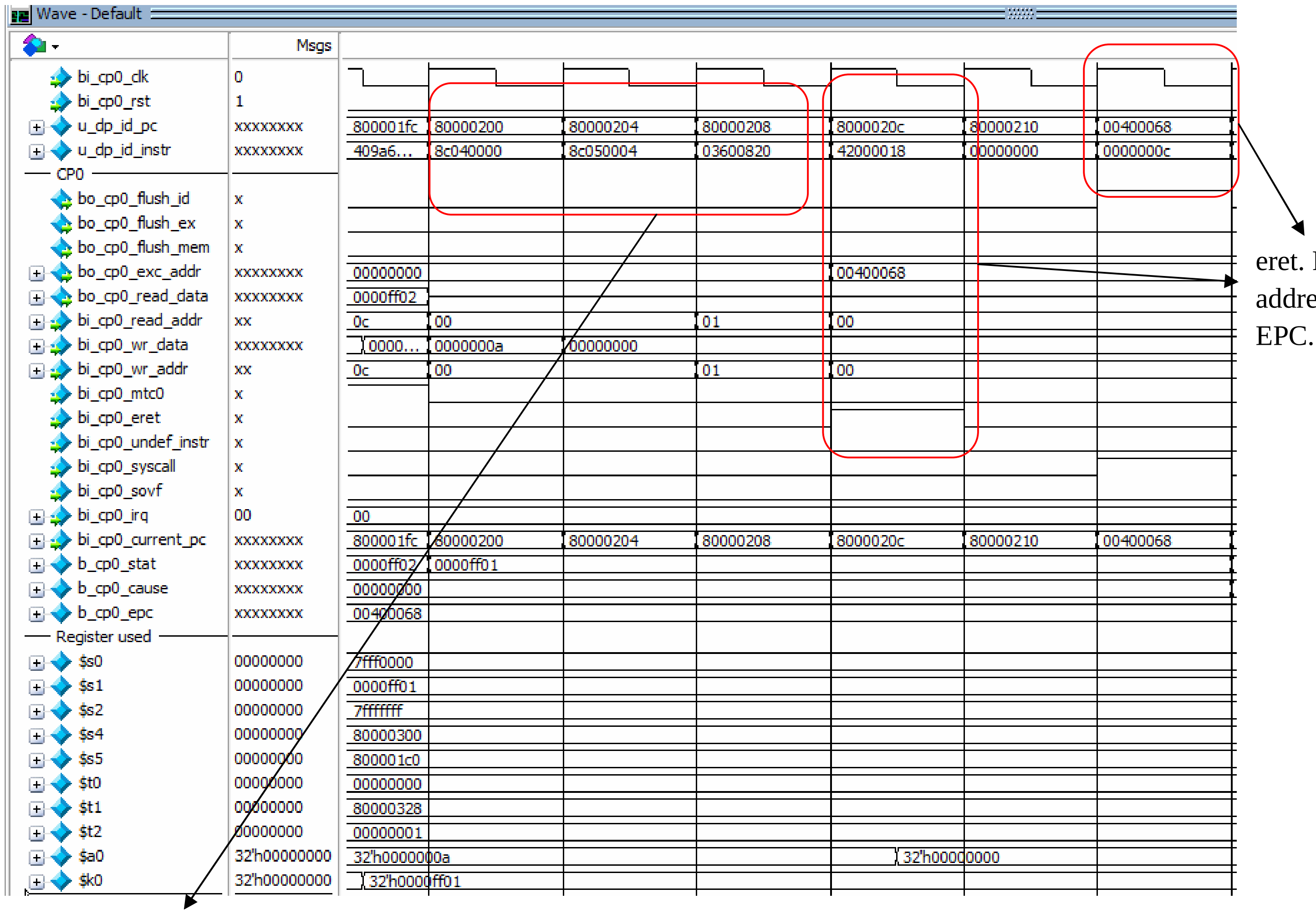


Wave - Default

Signal	Msgs	Values
bi_cp0_clk	0	
bi_cp0_rst	1	
u_dp_id_pc	xxxxxxxx	800001dc, 800001e0, 800001e4, 800001e8, 800001ec, 800001f0, 800001f4, 800001f8, 800001fc, 800
u_dp_id_instr	xxxxxxxx	40806800, 401a6000, 00000000, 335afffd, 375a0001, 00000000, 409a6000, 8c0
CP0		
bo_cp0_flush_id	x	
bo_cp0_flush_ex	x	
bo_cp0_flush_mem	x	
bo_cp0_exc_addr	xxxxxxxx	00000000
bo_cp0_read_data	xxxxxxxx	00000028, 0000ff02, 0000ff02
bi_cp0_read_addr	xx	0d, 0c, 00, 1f, 00, 0c, 00
bi_cp0_wr_data	xxxxxxxx	00000000, 00400068, 00000000, 0040..., 0000ff02, 00000000, 0000..., 0000..., 000
bi_cp0_wr_addr	xx	0d, 0c, 00, 1f, 00, 0c, 00
bi_cp0_mtc0	x	
bi_cp0_eret	x	
bi_cp0_undef_instr	x	
bi_cp0_syscall	x	
bi_cp0_sovf	x	
bi_cp0_irq	00	00
bi_cp0_current_pc	xxxxxxxx	800001dc, 800001e0, 800001e4, 800001e8, 800001ec, 800001f0, 800001f4, 800001f8, 800001fc, 800
b_cp0_stat	xxxxxxxx	0000ff02, 000
b_cp0_cause	xxxxxxxx	00000028, 00000000
b_cp0_epc	xxxxxxxx	00400068
Register used		
$s0	00000000	7fff0000
$s1	00000000	0000ff01
$s2	00000000	7fffffff
$s4	00000000	80000300
$s5	00000000	800001c0
$t0	00000000	00000000
$t1	00000000	80000328
$t2	00000000	00000001
$a0	32'h00000000	32'h0000000a
$k0	32'h00000000	32'h00400068, 32'h0000ff02, 32'h0000ff00, 32'h0000ff01

Move status register value to register file $k0 and reset value of status register.

Reset status register to enable interrupt



Wave - Default

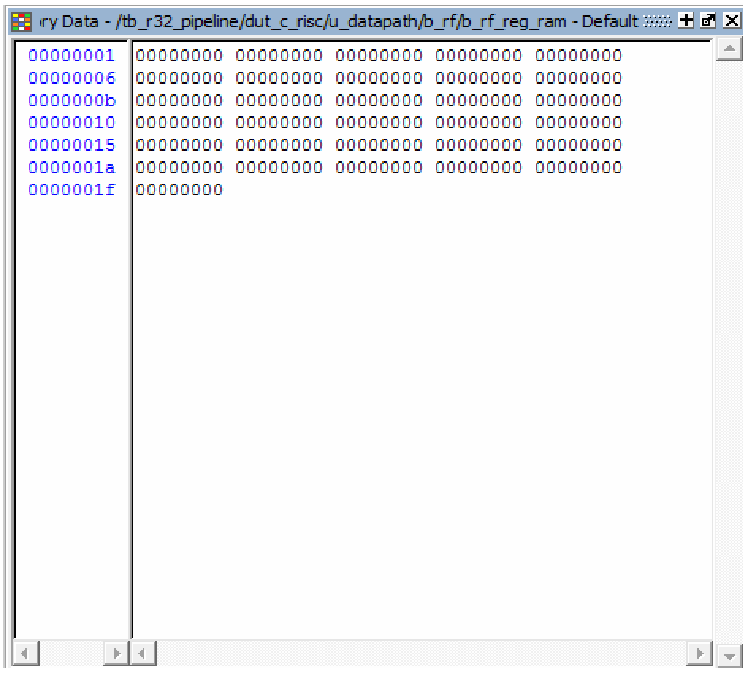
Signal	Msgs	Values
bi_cp0_clk	0	
bi_cp0_rst	1	
u_dp_id_pc	xxxxxxxx	800001fc, 80000200, 80000204, 80000208, 8000020c, 80000210, 00400068
u_dp_id_instr	xxxxxxxx	409a6…, 8c040000, 8c050004, 03600820, 42000018, 00000000, 0000000c
CP0		
bo_cp0_flush_id	x	
bo_cp0_flush_ex	x	
bo_cp0_flush_mem	x	
bo_cp0_exc_addr	xxxxxxxx	00000000, 00400068
bo_cp0_read_data	xxxxxxxx	0000ff02
bi_cp0_read_addr	xx	0c, 00, 01, 00
bi_cp0_wr_data	xxxxxxxx	0000…, 0000000a, 00000000
bi_cp0_wr_addr	xx	0c, 00, 01, 00
bi_cp0_mtc0	x	
bi_cp0_eret	x	
bi_cp0_undef_instr	x	
bi_cp0_syscall	x	
bi_cp0_sovf	x	
bi_cp0_irq	00	00
bi_cp0_current_pc	xxxxxxxx	800001fc, 80000200, 80000204, 80000208, 8000020c, 80000210, 00400068
b_cp0_stat	xxxxxxxx	0000ff02, 0000ff01
b_cp0_cause	xxxxxxxx	00000000
b_cp0_epc	xxxxxxxx	00400068
Register used		
$s0	00000000	7fff0000
$s1	00000000	0000ff01
$s2	00000000	7fffffff
$s4	00000000	80000300
$s5	00000000	800001c0
$t0	00000000	00000000
$t1	00000000	80000328
$t2	00000000	00000001
$a0	32'h00000000	32'h0000000a, 32'h00000000
$k0	32'h00000000	32'h0000ff01

eret. Return to address saved in EPC.

Restore Register

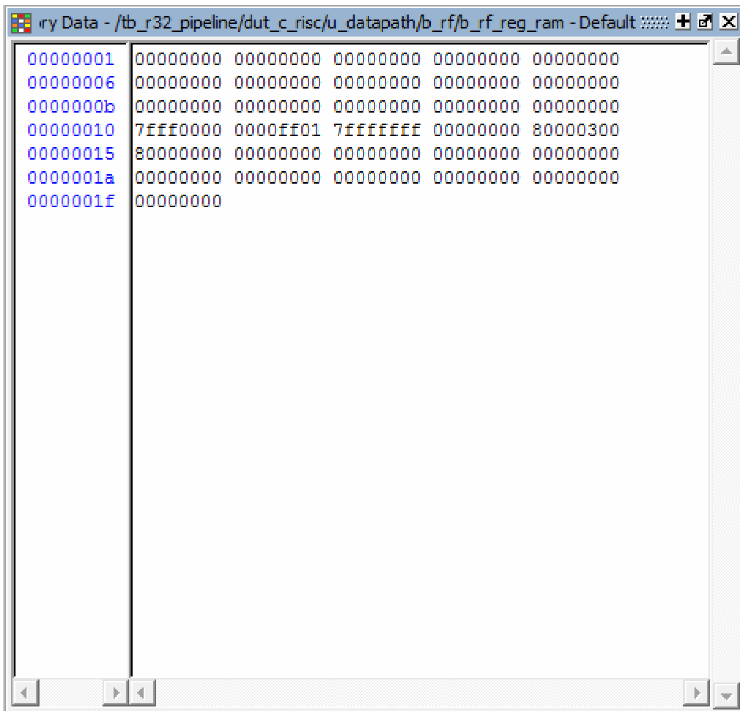
Register File

Register file when first start up and reset.



```
ry Data - /tb_r32_pipeline/dut_c_risc/u_datapath/b_rf/b_rf_reg_ram - Default
00000001  00000000 00000000 00000000 00000000 00000000
00000006  00000000 00000000 00000000 00000000 00000000
0000000b  00000000 00000000 00000000 00000000 00000000
00000010  00000000 00000000 00000000 00000000 00000000
00000015  00000000 00000000 00000000 00000000 00000000
0000001a  00000000 00000000 00000000 00000000 00000000
0000001f  00000000
```

Register file after $s0, $s1, $s2, $s4, $s5 is written with value. (partial test program before exception)



```
ry Data - /tb_r32_pipeline/dut_c_risc/u_datapath/b_rf/b_rf_reg_ram - Default
00000001  00000000 00000000 00000000 00000000 00000000
00000006  00000000 00000000 00000000 00000000 00000000
0000000b  00000000 00000000 00000000 00000000 00000000
00000010  7fff0000 0000ff01 7fffffff 00000000 80000300
00000015  80000000 00000000 00000000 00000000 00000000
0000001a  00000000 00000000 00000000 00000000 00000000
0000001f  00000000
```

Chapter 8 Conclusion

8.1 Conclusion

Exception/interrupt handling is essential for a processor to work and function properly. Without exception handling, processor may not be able to handle and solve exception/interrupt.

Coprocessor 0 implemented in this project is able to solve this problem. CP0 implemented is capable of save the return address, and decode the cause of incoming interrupt. As for software handling part, the exception handler is working well and proven to be working.

There are 3 instructions added to the RISC32 architecture to support exception/interrupt handling mechanism. The 3 instructions added are mfc0, mtc0 and eret. Modification of datapath unit and control unit are done to enable the processor to decode and process the added instruction.

There are 2 caches added to the RISC32 architecture also which are kernel segment and kernel data segment to enable exception handler to work.

Lastly, all the objective of this project is achieved. A complete interrupt handling mechanism is developed and proven to be work well. The Co-processor 0 is developed in RTL (Register Transfer Level) form and modeled in synthesizable Verilog. The Co-processor 0 functionality is verified as well.

8.2 Discussion and Future Work

The exception/interrupt handling developed is not yet in complete and perfect form. It currently only able to process 4 type of exceptions which are undefined instruction, syscall, sign overflow and I/O interrupt. The future improvement should include more functionality, let the CP0 able to process all kind of exception and interrupt. Other than that, the software part for exception handling is not yet completed. The interrupt service for each exception/interrupt is not developed. It currently only able to detect, examine the cause and then return to normal instruction execution. Kernel programmer will be responsible to develop the whole exception handling mechanism to handle exception.

Furthermore, due to limitation of simulation tool, modelsim student edition, the memory map used in this project is not able to use standard memory map convention. The memory limitation only allow up to 8k memory. The system should simulate as standard practice, which similar to industrial product.

Lastly, the CP0 currently developed can only handle single exception/interrupt at every particular time. Improvement should be done to allow CP0 to process with more the 1 exception/interrupt at the same time. This will be able to speed up processor's processing power.

REFERENCE

[1] K.M Mok, *Computer Organisation and Architecture Notes,* University of Tunku Abdul Rahman, Faculty of Information and Communication Technology, 2009.

[2] Sweetman, D. (2007). *See MIPS Run*. San Francisco: Morgan Kaufmann.

[3] David A. Patterson and John L. Hennessy (Edition 2004). *Computer Organization and Design: The Hardware/Software Interface 3rd*, Morgan Kaufman, 2004

[4] https://msdn.microsoft.com/en-us/library/ms892408.aspx [Accessed: 20 August 2015]

[5] Joe H., *"MIPS R4000 Microprocessor User's Manual", Second Edition*, 2011.

[6] Virgil B., Lab 7 - MIPS interrupt and exception handling, 1996

[7] W. Wolf, *FPGA-Based System Design*, 1st edition, Prentice Hall, 2004.

[8] K.M Mok, *Digital System Design Notes,* University of Tunku Abdul Rahman, Faculty of Information and Communication Technology, 2009.

Appendix

c_risc.v

```
`include "macro.v"
`default_nettype none

module c_risc
(// === in out port declaration =====
//OUTPUT
/*
// Input / Output from VGA controller


// Input / Output from PS2 Mouse controller


// Input / Output from PS2 Keyboard controller


// Input / Output from UART controller


*/
//INPUT
input wire              ui_cd_clk,
input wire              ui_cd_rst
);

//===================================
//======== internal wire ===========
//===================================
//main control signal
wire    u_cd_alb_src;
wire    u_cd_rd_src;
wire    u_cd_mult_en;
wire    u_cd_sign_mult;
wire    u_cd_rf_wr;
wire    u_cd_mem_wr;
wire    u_cd_mem_re;
wire    u_cd_sign_ext;
wire    u_cd_hi_wr;
wire    u_cd_lo_wr;
wire    u_cd_alb_to_rf;
wire    u_cd_hi_to_rf;
wire    u_cd_mem_to_rf;
wire    u_cd_jump;
wire    u_cd_jr;
wire    u_cd_jal;
wire    u_cd_jalr;
wire    u_cd_beq;
wire    u_cd_bne;
wire    u_cd_blez;
wire    u_cd_bgtz;

//alb
wire    [`ALB_CTRL_NB-1:0] u_cd_alb_ctrl;

//main control
wire    [`OPCODE_NB-1:0]   u_cd_opcode;
wire    [`FUNCT_NB-1:0]    u_cd_funct;
wire    [4:0]              u_cd_rs;

// Memory unit
wire [`WORD_NB-1:0]        u_cd_pc;
wire [`WORD_NB-1:0]        u_cd_dmem_addr;
wire [`WORD_NB-1:0]        u_cd_store_data;

// Mem unit and forwarding
reg [`WORD_NB-1:0]         u_cd_instr;
wire [`WORD_NB-1:0]        u_cd_text_instr;
wire [`WORD_NB-1:0]        u_cd_ktext_instr;
reg [`WORD_NB-1:0]         u_cd_loaded_data;
wire [`WORD_NB-1:0]        u_cd_loaded_ndata;
wire [`WORD_NB-1:0]        u_cd_loaded_kdata;

//datapath output
wire                       u_dp_mem_re;
wire                       u_dp_mem_wr;

//IO(UART)
reg                        u_cd_intr_uart;
//IO(PS2 Mouse)
reg                        u_cd_intr_ps2_mouse;
//IO(PS2 Keyboard)
reg                        u_cd_intr_ps2_keyboard;

//cp0 wire
//control
wire                       u_cd_mfc0;
wire                    u_cd_mtc0;
wire                    u_cd_eret;
wire                    u_cd_syscall;
wire                    u_cd_undef_inst;

//====================================
//======== controlpath ===========
//====================================
u_ctrl_path u_control(              //control_u
  //output signal
  .uo_cp_alb_src(u_cd_alb_src),
  .uo_cp_rd_src(u_cd_rd_src),
  .uo_cp_mult_en(u_cd_mult_en),
  .uo_cp_sign_mult(u_cd_sign_mult),
  .uo_cp_rf_wr(u_cd_rf_wr),
  .uo_cp_mem_wr(u_cd_mem_wr),
  .uo_cp_mem_re(u_cd_mem_re),
  .uo_cp_sign_ext(u_cd_sign_ext),
  .uo_cp_hi_wr(u_cd_hi_wr),
  .uo_cp_lo_wr(u_cd_lo_wr),
  .uo_cp_alb_to_rf(u_cd_alb_to_rf),
  .uo_cp_hi_to_rf(u_cd_hi_to_rf),
  .uo_cp_mem_to_rf(u_cd_mem_to_rf),
  .uo_cp_jump(u_cd_jump),
  .uo_cp_jr(u_cd_jr),
  .uo_cp_jal(u_cd_jal),
  .uo_cp_jalr(u_cd_jalr),
  .uo_cp_beq(u_cd_beq),
  .uo_cp_bne(u_cd_bne),
  .uo_cp_blez(u_cd_blez),
  .uo_cp_bgtz(u_cd_bgtz),
  .uo_cp_alb_ctrl(u_cd_alb_ctrl),
  .uo_cp_mfc0(u_cd_mfc0),
  .uo_cp_mtc0(u_cd_mtc0),
  .uo_cp_eret(u_cd_eret),
  .uo_cp_syscall(u_cd_syscall),
  .uo_cp_undef_inst(u_cd_undef_inst),

  //input signal
  .ui_cp_opcode(u_cd_opcode),
  .ui_cp_funct(u_cd_funct),
  .ui_cp_rs(u_cd_rs));

//====================================
//======== datapath ===========
//====================================
u_data_path u_datapath
(//*********** INSTANTIATION *************
 //======= OUTPUT =======
 // Main control
 .uo_dp_opcode(u_cd_opcode),
 .uo_dp_funct(u_cd_funct),
 .uo_dp_rs(u_cd_rs),

 // Memory unit
 .uo_dp_im_addr(u_cd_pc),
 .uo_dp_dm_addr(u_cd_dmem_addr),
 .uo_dp_dm_store(u_cd_store_data),
 .uo_dp_mem_wr(u_dp_mem_wr),
 .uo_dp_mem_re(u_dp_mem_re),

 //======= INPUT =======
 // Main control
 .ui_dp_alb_src(u_cd_alb_src),                    //aluSrc
 .ui_dp_rd_src(u_cd_rd_src),            //regDst
 .ui_dp_mult_en(u_cd_mult_en),
 .ui_dp_sign_mult(u_cd_sign_mult),
 .ui_dp_rf_wr(u_cd_rf_wr),   //regWr
 .ui_dp_mem_wr(u_cd_mem_wr),     //mem_wr
 .ui_dp_mem_re(u_cd_mem_re),       //mem_re
 .ui_dp_sign_ext(u_cd_sign_ext),      //extOp
 .ui_dp_hi_wr(u_cd_hi_wr),
 .ui_dp_lo_wr(u_cd_lo_wr),
 .ui_dp_hi_to_rf(u_cd_hi_to_rf),
 .ui_dp_mem_to_reg(u_cd_mem_to_rf),       //mem_to_reg0
 .ui_dp_beq(u_cd_beq),
 .ui_dp_bne(u_cd_bne),
 .ui_dp_blez(u_cd_blez),
 .ui_dp_bgtz(u_cd_bgtz),
 .ui_dp_jump(u_cd_jump),
 .ui_dp_jr(u_cd_jr),
 .ui_dp_jalr(u_cd_jalr),
 .ui_dp_jal(u_cd_jal),

 //ALB
 .ui_dp_alb_ctrl(u_cd_alb_ctrl),        //aluCtr
 // Memory_unit
 .ui_dp_instr(u_cd_instr),                          //instr
 .ui_dp_loaded_data(u_cd_loaded_data),//pc4
 //cp0
 .ui_dp_intr_vector({1'b0,1'b0,1'b0,u_cd_intr_uart,u_cd_intr_ps2_mouse,u_cd_intr_ps2_
keyboard}),
 .ui_dp_cp0_mfc0(u_cd_mfc0),
 .ui_dp_cp0_mtc0(u_cd_mtc0),
 .ui_dp_cp0_eret(u_cd_eret),
 .ui_dp_cp0_syscall(u_cd_syscall),
 .ui_dp_cp0_undef_inst(u_cd_undef_inst),
 // System signal
 .ui_dp_clk(ui_cd_clk),
 .ui_dp_rst(ui_cd_rst)
);


//===================================
//======== memory unit ===========
//===================================

  u_cache
  u_text_seg(
  .uo_cm_rd_data (u_cd_text_instr),
  .ui_cm_addr (u_cd_pc),
  .ui_cm_wr_data ({`WORD_NB{1'b0}}),
  .ui_cm_re (1'b1),      // i-cache always read
  .ui_cm_wr (1'b0),
  .ui_cm_clk (ui_cd_clk));

  u_cache
  u_data_seg(
  .uo_cm_rd_data (u_cd_loaded_ndata),
  .ui_cm_addr (u_cd_dmem_addr),
  .ui_cm_wr_data (u_cd_store_data),
  .ui_cm_re (u_dp_mem_re),
  .ui_cm_wr (u_dp_mem_wr),
  .ui_cm_clk (ui_cd_clk));

  u_cache
  u_ktext_kseg0(
  .uo_cm_rd_data (u_cd_ktext_instr),
  .ui_cm_addr (u_cd_pc),
  .ui_cm_wr_data ({`WORD_NB{1'b0}}),
  .ui_cm_re (1'b1),      // i-cache always read
  .ui_cm_wr (1'b0),
  .ui_cm_clk (ui_cd_clk));

  u_cache
  u_kdata_kseg0(
  .uo_cm_rd_data (u_cd_loaded_kdata),
  .ui_cm_addr (u_cd_dmem_addr),
  .ui_cm_wr_data (u_cd_store_data),
  .ui_cm_re (u_dp_mem_re),
  .ui_cm_wr (u_dp_mem_wr),
  .ui_cm_clk (ui_cd_clk));

endmodule
```