THE DEVELOPMENT OF AN EXCEPTION SCHEME FOR 5-STAGE PIPELINE RISC PROCESSOR

BY

Goh Jia Sheng

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfilment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology
(Perak Campus)

JAN 2019

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**: ____________________

THE DEVELOPMENT OF AN EXCEPTION SCHEME FOR 5-STAGE PIPELINE RISC PROCESSOR

BY

Goh Jia Sheng

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfilment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology
(Perak Campus)

JAN 2019

DECLARATION OF ORIGINALITY

I declare that this report entitled "**THE DEVELOPMENT OF AN EXCEPTION SCHEME FOR 5-STAGE PIPELINE RISC PROCESSOR**" 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	:	________________________
Date	:	________________________

ACKNOWLEDGEMENTS

I would like give my deepest appreciation to my project supervisor, Mr. Mok Kai Ming for providing me an opportunity to involve in the Computer Architecture development. Thank you for your patience and invaluable guidance and suggestion throughout this project.

Next, I would like to thanks my family members for giving me endless support and encouragement throughout my undergraduate study. Furthermore, I would like to thanks my course mates and friends who has supported me throughout the entire project.

Finally, I appreciate all the guidance and support that provided by people that I mentioned above. All the supports and helps have contributed to the accomplishment of this project.

ABSTRACT

Exception classified into two types, which are the internal exception and external exception. Normally, we called internal exception as trap and External exception as interrupt. Exception makes the 5-stage pipeline processor more complicated because the exception is difficult to handle in pipeline processor due the overlapping instruction characteristics. The exception will cause abnormal program flow, and when exception occur, we need to provide some operation to overcome the problem. The IoT SoC processor will used for this project purpose. Up-to-date, the processor has a few I/O modules integrated namely the UART, GPIO and SPI. It also has a co-processor and programmable interrupt controller to handle the exceptions. The handling of the exceptions was half-planned, however, not up to a high confidence level. Therefore, this project is initiated to develop an exception handling scheme to handle the multiple interrupt (including nested interrupts) occurrence. Interrupt can occur at any time, and the timing to capture the data is critical. For example, when the UART and SPI received the data at the same time, both module will raise the interrupt flag concurrently. Therefore, we need a plan to schedule which one need to be serve first. The situation is further complicated when the multiple nested interrupts and traps occurs concurrently. With the availability of the exception-handling scheme, it is straightforward to resolve the conflicts among the mentioned exceptions. In addition, it will be easier to plan ahead to integrate new devices without having to worry about buggy exception handling.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic and Logic Unit
CP0	Core Processor 0
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IOT	Internet of Things
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MIPS	Microprocessor without Interlocked Pipelined Stage
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SOC	System-On-Chip

Chapter 1 Introduction

1.1 Background Information

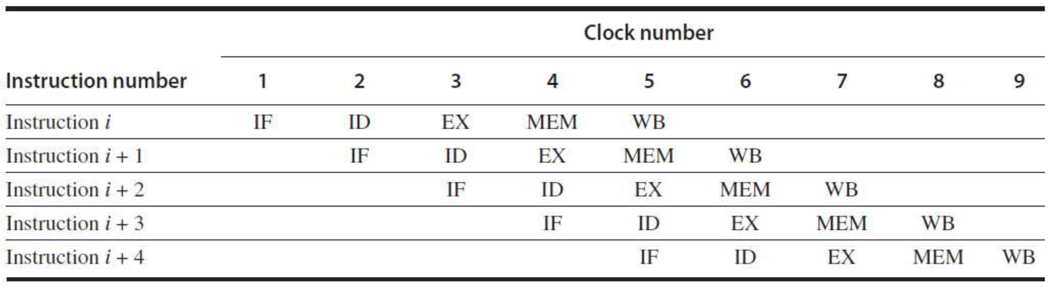
1.1.1 RISC

RISC is short for Reduced Instruction Set Computing that developed and introduced by IBM in 1980 and coined by David Patterson. John Cocke of IBM Research in Yorktown, New York, originated the RISC concept in 1974 by proving that about 20% of the instructions in a computer did 80% of the work. Therefore, RISC using simple and small instruction set hence less hardware needed, so the system can operate at higher speeds and low-power consumption, and this makes the processor easier to build and test. RISC has four philosophy (Mok, 2009):

- Fixed instruction lengths.
- Load-store instruction sets.
- Limited number of addressing modes.
- Limited number of operations.

1.1.2 MIPS

MIPS short for Microprocessor without Interlocked Pipelined Stage is the Microprocessor based on the Reduced Instruction Set Computer (RISC) architecture. MIPS initiated in 1981 by a team led by John L. Hennessy and come out conclusion in the year 1984. Recently, MIPS implement in the digital home, networking, embedded system, Internet of things and mobile applications. At the pass, MIPS used in video game consoles such as Sony PlayStation, PlayStation and PlayStation Portable. The MIPS ISA based on a 32-bit word. MIPS support 32-bit addressing (word-addressed). MIPS is a load-store architecture that means it can perform load and store operation between memory and registers and ALU operation between registers. MIPS is a modular architecture it contains coprocessors 0(CP0) which handle the exception and coprocessors 1(CP1) which handle the floating-point operation (Mok, 2009). The details of MIPS architecture and relative information can found in a book, which name Computer Organization and Design: The Hardware/ Software Interface (Patterson and Hennessy, 2008). MIPS processors operate by breaking instruction execution cycle into multiple small independent stages and this technic call pipelining. Figure 1.1 shown the MIPS 5-stage pipelining



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figure 1.1: MIPS five-stage pipelining (Patterson & Hennessy, 2002, p.A-7)

The instruction execution is divided to 5 stages, IF (“Instruction Fetch”), ID (“Instruction Decode”), EX (“Instruction Execution”), MEM (“Memory access”) and WB (”Write Back”).

- IF: Send the program counter (PC) to instruction memory, fetch the instruction from the instruction memory/instruction cache (I-cache) and update the PC by adding 4 (instruction is 4 bytes).
- ID: Decodes the instruction and read the corresponding register for CPU use.
- EX: Performs an arithmetic or logical operation.
- MEM: Write or Read a data from the data memory (D-cache) only the instruction load and store will use this stage.
- WB: store the value obtained from an operation back to the register file.

1.1.3 Exception

Exception is an event other than branches or jump that change the normal flow of instruction execution (Patterson & Hennessy, 2008, p.384). Exception was classify into two type, which are synchronous exception and asynchronous exception. Asynchronous exception is the exception that occurs with no relation to the program executed such as I/O requests while synchronous exception is exception that occurs at the same place every time the program executed with the same data and memory allocation, example for synchronous exception are arithmetic overflow, undefined instruction, and page fault. (Patterson & Hennessy, 2002, p.A-40)

1.1.4 Interrupt

An interrupt is an external event that changes the normal flow of instruction execution (Patterson & Hennessy, 2008, p.384). Interrupts are the asynchronous exception. Example for the Asynchronous event is I/O device request, power failure and Hardware malfunction. The asynchronous exception usually handled after the completion of the current instruction, which makes them easier to handle. Coprocessor 0 (CP0) system control coprocessor will handle these interrupts.

1.1.5 Trap

The trap is an internal event that changes the normal flow of instruction execution. The trap is the synchronous exception. Example for the synchronous event is invoked operating system, tracing instruction execution, breakpoint, arithmetic overflow, page fault, misaligned memory accesses, memory protection violations and using undefined instruction.

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 designed microprocessor as Intellectual Property or IP for commercial purpose. The microprocessor IP includes information on the entire design process for the front-end (modelling and verification) and back-end (physical design) integrated circuit (IC) design. These are trade secrets of a company and certainly not made available in the market at an affordable price for research purpose.
- Several freely available microprocessor cores can found in internet, most of them can found at OpenCores (http://www.opencores.org/). Unfortunately, these processors do not implement the entire MIPS Instruction Set Architecture (ISA) and lack comprehensive documentation. This makes them unsuitable for reuse and customization.
- The verification specification for a freely available RISC microprocessor core that is available on the Internet is not well developed and incomplete. Therefore, without a good verification specification, the verification process will be slow and hence, will slow 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 functionally proven before the physical design phase can proceed smoothly. Otherwise, if the front-end design has to be changed, the physical design process has to be re-design.

This project will aim to provide solutions to the above problems by creating a 32-bit RISC core-based development environment to assist research work in the area of soft-core and application specific hardware modelling. In the RISC32 project, the project divided into several units based on MIPS architecture.

1.3 Problem Statement

Currently, a team from FICT has designed an IoT SoC processor based on a subset of the MIPS ISA. The processor supports three type of communication interface, which are the UART, SPI and GPIO. The UART, SPI and GPIO have integrated into the IoT processor using I/O mapped technique. The individual test cases for each I/O have been conducted but have not gone through thorough multiple and nested exception verification and there are lack of well-defined exception handler scheme to manipulate the multiple interrupt occurrence and traps. Exception makes the 5-stage pipelining processor more complicated because the exception is hard to be handled in pipeline processor due the overlapping instruction characteristics (Patterson & Hennessy, 2002, p.A-37). The exception causes the instruction to stop executing in the middle of execution. To handle Exception, first, we need to detect the exception, what is the cause, when it occurs, how to handle it and what to do after exception. For 5-stage pipeline processor, handling exception is more difficult when multiple exceptions occur at the same time (clock cycle). Fortunately, the cause of exception can be determine based on the stage where by an instruction cause exception. On the other hand, if the multiple exceptions occur in same time, we need to come up with a plan to determine which exception we need to serve first to ensure smooth running of the program. The exception will also occur out-of-order that means out of the instruction execution order, this makes exception more difficult to handle. After handling the exception, there is two alternative, which is terminate the program or return to the program. When returning to the program, the problem is where the program needs to restart at the user program, the branch delay slot also makes a return from the exception to the user program more complicated. When the exception was in execution, there is possibly another exception occurs, this also known as a nested exception. The main purpose for this project is to develop an exception scheme to handle various type of exception in order to ensure future reliable I/Os integration and smooth running of the user program.

1.4 Project Scope

The project scope includes the development of an exception handler scheme for interrupt conflict and the nested interrupt resolution. The exception scheme also needs to be verified its functionality through simulation by write the test code to trigger the interrupt individually. After that, trigger the multiple I/O interrupt and trap make sure the exception scheme well function. In addition, physical synthesis the RISC32 IoT processor on FPGA board will conducted to verify the correctness of the exception scheme.

1.5 Project Objectives

The Project Objectives are as shown below:

- To develop an exception scheme for RISC32 IOT processor.
- To develop a test bench to verify the exception handle and Interrupt Service Routine (ISR) code.
- To synthesize the RISC32 IoT Processor and carry out physical tests on the I/O function.

1.6 Impact and significance

As a summary of 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, test bench architectures etc.
- A complete physical design in Field Programmable Gate Array (FPGA) with documented timing and resource usage information.

With the available of well-defined exception handler scheme, it can build up high confident level to extend the IoT SoC processor. It can allow us to add-on extra communication interface on processor. For instance, integration of ADC (analogue to digital converter) to the processor without having to worry about the data conflicting. Consequently, the research work could be done easier and speed up significantly.

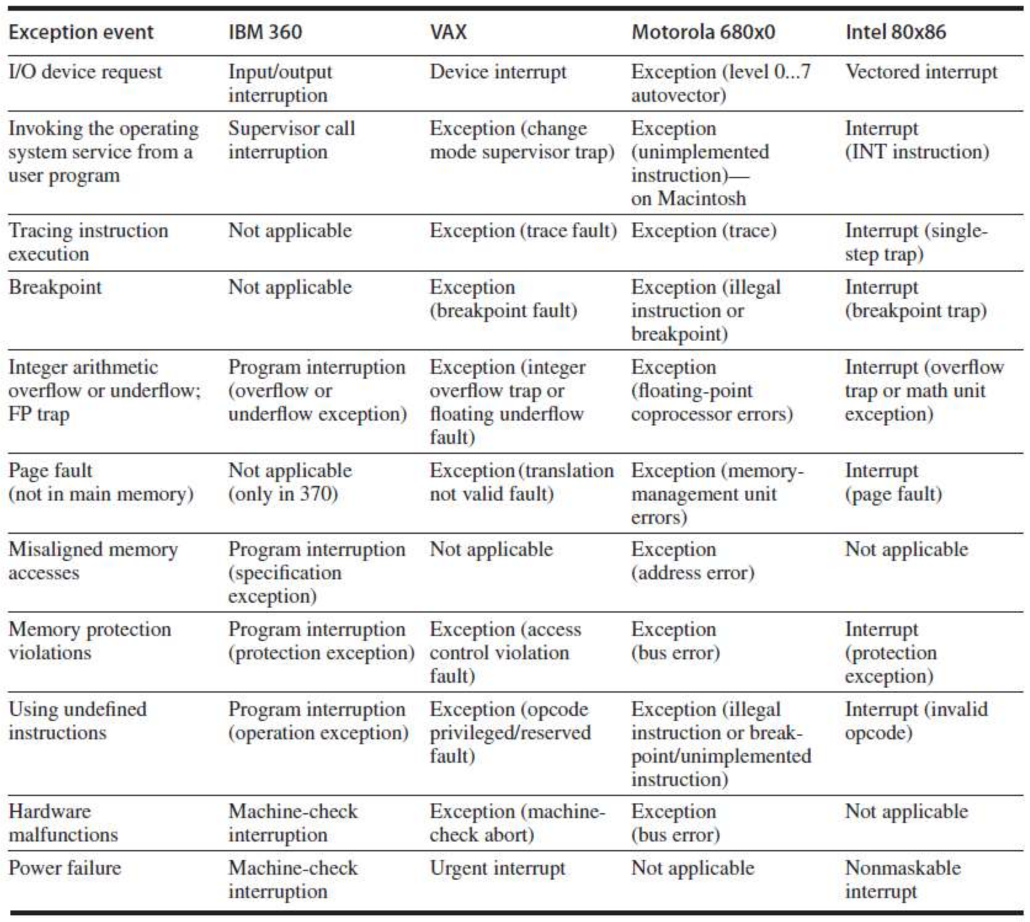
Chapter 2 Literature Review

2.1 Exception

Exception is an event other than branch or jump that change the normal flow of instruction execution. The type of exception are listed below :- (Patterson & Hennessy, 2002, pp.A-38-A39)

- I/O device request
- Invoking an operating system service from a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

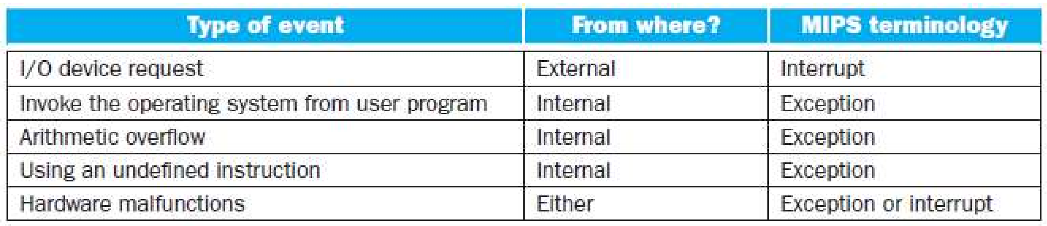
Different Architecture using different terminology to describe the exception. Figure2.1 show the different name for the common exception event.



Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (level 0...7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)— on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

Figure 2.1 Different architecture use different names to represent common exception event. IBM and Intel using interrupt for every exception event. Motorola using exception while Vax using both interrupt and exception. (Patterson & Hennessy, 2002, p.A-40).

In MIPS, it classify type of exception event into external and internal. External exception event name interrupt while internal exception event name exception. Figure 2.2 show that the exception event for MIPS terminology.



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

Figure 2.2 MIPS terminology to differentiate type of exception event. (Patterson & Hennessy, 2008, p.385).

2.2 Characteristic of exception

Exception can be classify based on its characteristic. Figure 2.3 show the exception event and its characteristics. The five main independent characteristic are (Patterson & Hennessy, 2002, pp.A-40-A-41):

- Synchronous Vs Asynchronous

Asynchronous exception is the exception that occurs with no relation to the program executed such as I/O requests while synchronous exception is exception that occurs at the same place every time the program executed with the same data and memory allocation, example for synchronous exception are arithmetic overflow, undefined instruction, and page fault. Asynchronous usually handle after the current instruction complete execute.

- User requested Vs Coerced

User requested event is the user request it to happen, for instance, “syscall”. User requested actually not really exception because it is predictable but the only method to create the event is to cause exception. Coerced exception is an unpredictable event that not under the user control.

- User maskable Vs user nonmaskable

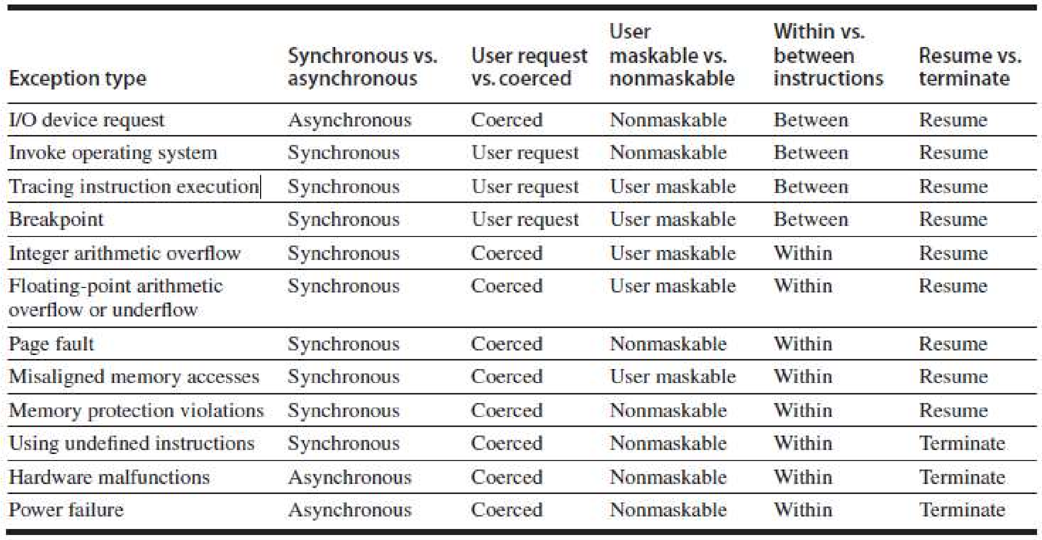
If the exception can disable by user program the event is user maskable event. Otherwise, it is nonmaskable event.

- Within Vs between instructions

If the exception event occur and stop the current executing instruction in the pipeline then the event is classify “within”. If the exception event allow the current executing instruction to complete, then only serve the exception event then the event id classify “between”.

- Resume Vs Terminate

After handling the exception event, there are two alternative way, which is return to the user program, or terminate the current program.



Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure 2.3 Exception event and its characteristics. (Patterson & Hennessy, 2002, p.A-42).

2.3 Precise exception Vs Imprecise exception

➢ Precise exception

Precise exception means that when the exception occurred, the instruction causes the exception (instruction victim) will be recorded. There are able to draw a line between the instruction before the instruction victim and the instruction after the instruction victim. Beside, all the instruction before the instruction victim will executed while all the instruction after the instruction victim will flushed out from pipeline. This method make programmer work more easy because they can ignore the timing effect of the CPU implementation.

The feature provided with precise exception are (Sweetman, 2007, pp.107-108):

- Unambiguous proof of guilt: After the exception, exception will return to the user program by load the value from EPC register into PC. EPC will always point to the instruction that cause the exception. However, EPC also will point to the preceding branch instruction if the BD in cause register was set.
- Exceptions appear in instruction sequence: For pipeline processor, multiple exception will occur in the same time in different stage of execution. For instance, the load instruction (lw) cause the Memory Translation exception in the MEM stage (4th stage of the pipeline) and at the same time, a later instruction hit an exception in the ID stage (2nd stage of the pipeline), this will cause the out of order exception. The later instruction arise the exception earlier than the prior one. To avoid this problem, an exception detected early but no perform the operation immediately, the exception event just marked and passed until end of the MEM stage.
- Subsequent instructions nullified: Because of pipelining, instructions following the victim instruction have been started and inside pipeline. However, MIPS guarantee that, the instruction following the victim instruction will not have effect toward the register file or CPU and return to the user program just like exception no occur.

➢ Imprecise exception

The imprecise exception mean that when the exception occur we cannot precisely tell where we need to return after exception. For instance: (Patterson & Hennessy, 2002, p.A-54).

1. DIV.D   F0,F2,F4
2. ADD.D F10,F10,F8
3. SUB.D F12,F12,F14

ADD.D and SUB.D expected to complete before the DIV.D because DIV.D need more cycle to complete compare to ADD.D and SUB.D. This also known as out-of-order completion. Suppose SUB.D cause an arithmetic exception at the point where ADD.D completed but DIV.D has not completed. This result in imprecise exception because it cannot precisely tell that where should return after the exception.
Another example (Zjueducn, n.d., p.13)

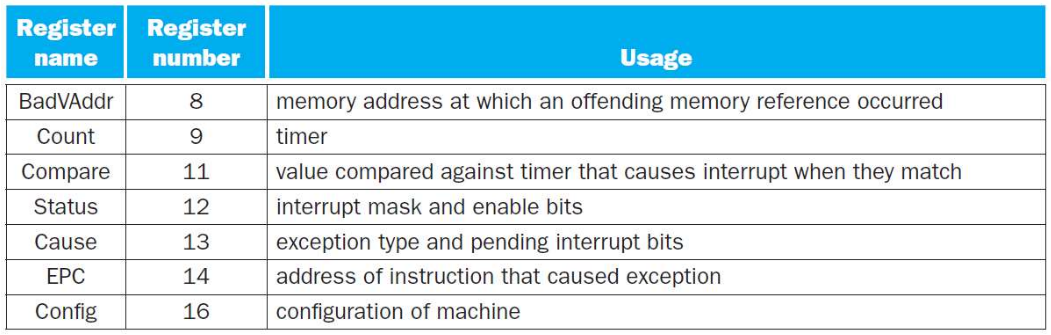
1. Mult r1,r2,r3 ;Multiply take 10 cycles
2. Add r10,r11,r12 ;Add take 5 cycle

Add will complete before the multiply. If the multiply cause an arithmetic exception, but add has already update the value in r10. This result imprecise exception.
In general, Imprecise exception always involve when there are instruction take multiple cycle to complete For instance, instruction involve in floating point , multiply and divide. Imprecise exception are harder to handle compare to precise exception.

2.4 Exception handler Scheme for MIPS

2.4.1 Coprocessor 0

In MIPS, there have two Coprocessor, which are Coprocessor 0 and Coprocessor 1. In this project, we are more interesting in Coprocessor 0. Coprocessor 0 also known as system control coprocessor, it handle the exception and interrupt by records the information that correspond exception event. Coprocessor 0 has its own registers files. Figure 2.4 shown the coprocessor 0's registers and its usage.



Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

Figure 2.4 coprocessor 0's registers and its usage. (Patterson & Hennessy, 2008, p.B-33).

2.4.2 BadVaddr register

BadVaddr Register will store the referenced memory location's address if the instruction that caused exception made a memory access.

2.4.3 Count Register and Compare Register

Count Register act as a timer, increment at a fixed period. When the value in the Count Register count until the value in the Compare register, it will raise a hardware interrupt.

2.4.4 Status Register

Status Register used to indicate the exception details. Figure 2.5 show the Status Register and its field.

It made up by 4 field:

- Interrupt Mask (Status Register[15:8])

-There are 6 bit for hardware and 2 bit for software interrupt level

-Mask bit = 1, when the interrupt is enable.

-Mask bit=0, when the interrupt is disable.

- Interrupt occur when both interrupt mask (Status Register) and Interrupt Pending (Cause Register) was asserted.

- When interrupt raise, correspond Interrupt pending bit will asserted but it will not be served when the Interrupt Mask disable.

- User Mode /Kernel mode(Status Register[4])

Status Register [4] =0, the processor running in kernel mode.

Status Register [4] =1, the processor running in user mode.

- Exception Level (Status Register[1])

-Normally 0.

-Set to 1 when exception event happen.

- To prevent the multi-level exception by prevent other exception event changing the EPC value.

- Should reset after finish exception.

- Interrupt Enable (Status Register[0])

Status Register [0] = 1, interrupt enable.

Status Register [0] = 0, interrupt disable.

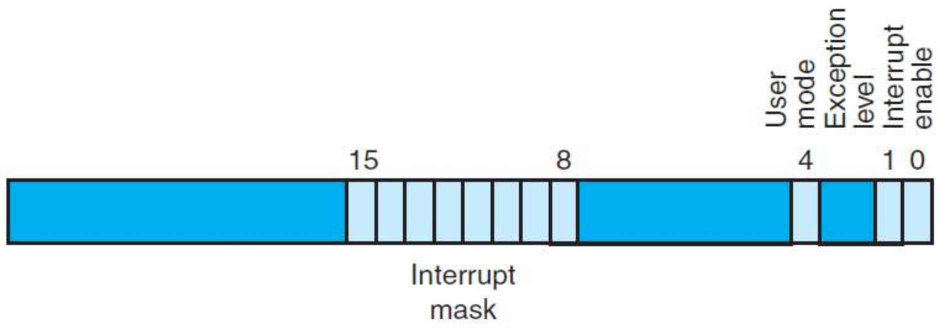


Figure 2.5: Status Register. (Patterson & Hennessy, 2008, p.B-35)

2.4.5 Cause Register

Cause register is use to determine the causes for the exception. Figure 2.6 show that the Cause register and its field.

- Branch Delay(Cause Register[31])

- Cause Register [31] = 1, when the exception occur inside in branch/ jump instruction.

- EPC store the branch/jump instruction instead of the instruction cause the exception.

-exception handler must look at EPC+4 for the offending instruction.

- Pending interrupt (Cause Register[15:8])

-Pending bit = 1, when the exception occur but no serve.

-mainly use to handle multiple exception occur at a same time.

- Exception Code (Cause Register[6:2])

-use the indicate the causes of exception

-the exception code shown in Figure 2.7.

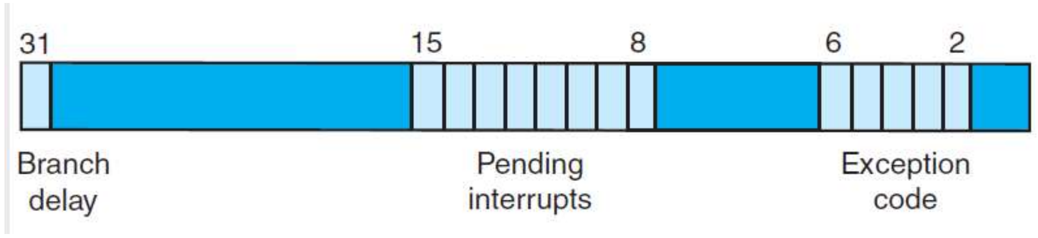
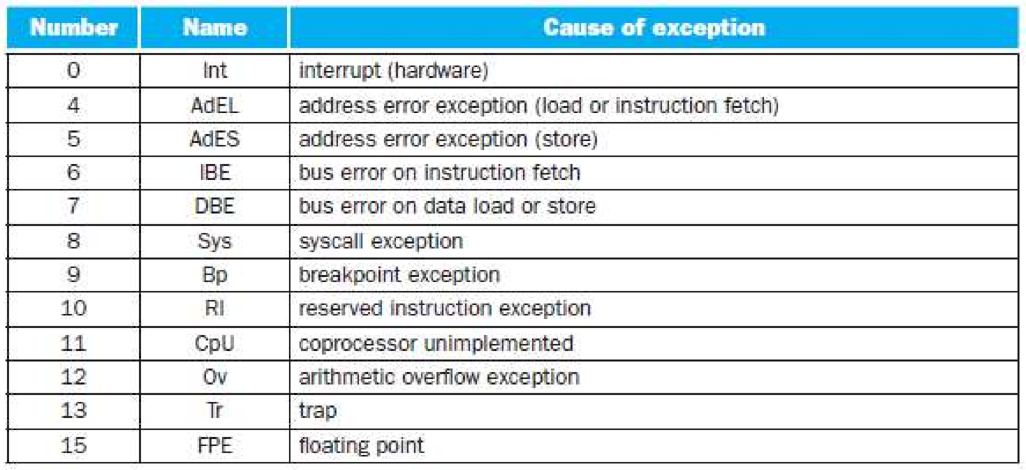


Figure 2.6: Cause Register (Patterson & Hennessy, 2008, p.B-35).



Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Figure 2.7: Exception code (Patterson & Hennessy, 2008, p.B-35).

2.4.6 EPC Register

- Store the instruction address that causes the exception occur.
- If BD (Cause Register [31] was set, when the exception occur, the branch / jump, instruction was load into EPC Register instead the instruction cause exception.

2.4.7 Instruction associate with exception handling

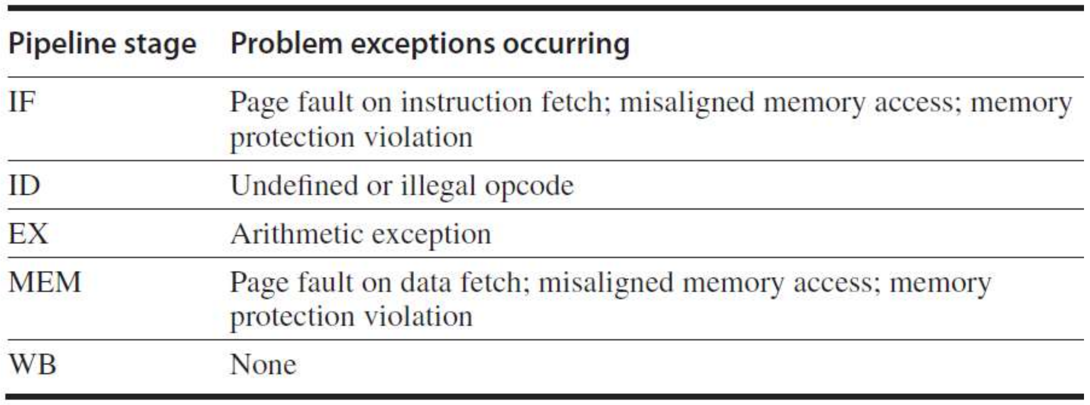
Some instruction are dedicated build to access the Register in CP0, because CP0 does not implement ALU unit to carry out the operation, so, the data need to move to the CPU for compute and move it back to CP0.The instruction and its function are list below:- (Sweetman, 2007, p.55).

```
mtc0 <register in CPU>, < destination in CP0>       #move data from CPU to CP0
mfc0 < register in CPU >, < source in CP0>          #move data from CP0 to CPU
eret                                                 # return from exception
```

When the bit 4(user mode/ kernel mode) in the status register was set, it means that the program is in user mode, it can use all the general-purpose register in CPU for data transfer. However, when enter the exception handler, normally the program execute in kernel mode, register k0 and k1 reserved for kernel usage. For instruction "eret", it return form exception by load the EPC value into PC.

2.4.8 Step-by-step how MIPS handle Exception

1. Determine which instruction that cause the exception. For MIPS, there are multi instruction in the pipeline processor within a single clock cycle. At different pipeline stage, it will arise different exception. The detail shown in Figure2.8:



Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure 2.8: Exceptions that may occur in the MIPS pipeline. (Patterson & Hennessy, 2002, p.A-44).

2. Stop the offending instruction and let the prior instruction finished
3. Flush the offending instruction and the all instruction inside the pipeline stage.
4. Load PC value into EPC, for determine the cause or return from exception.
5. Load the 0x80000180 into PC. 0x80000180 is the single entry point for all exceptions in MIPS architecture.
6. Determine the cause by using the information inside Cause Register.
7. Pass the work to Operating system, Operating system will handle the case. In other word, jump to the interrupt service routine. OS will handle the cases by :
   - Terminate the program and display the reason.
     -undefined instruction
     -hardware failure
     -arithmetic overflow
   - Perform the desired task and return to program from exception
     -I/O device request
     -system service call
8. Return from exception by load EPC+4 into PC.

The flow of the handle the exception shown in the Figure 2.9.

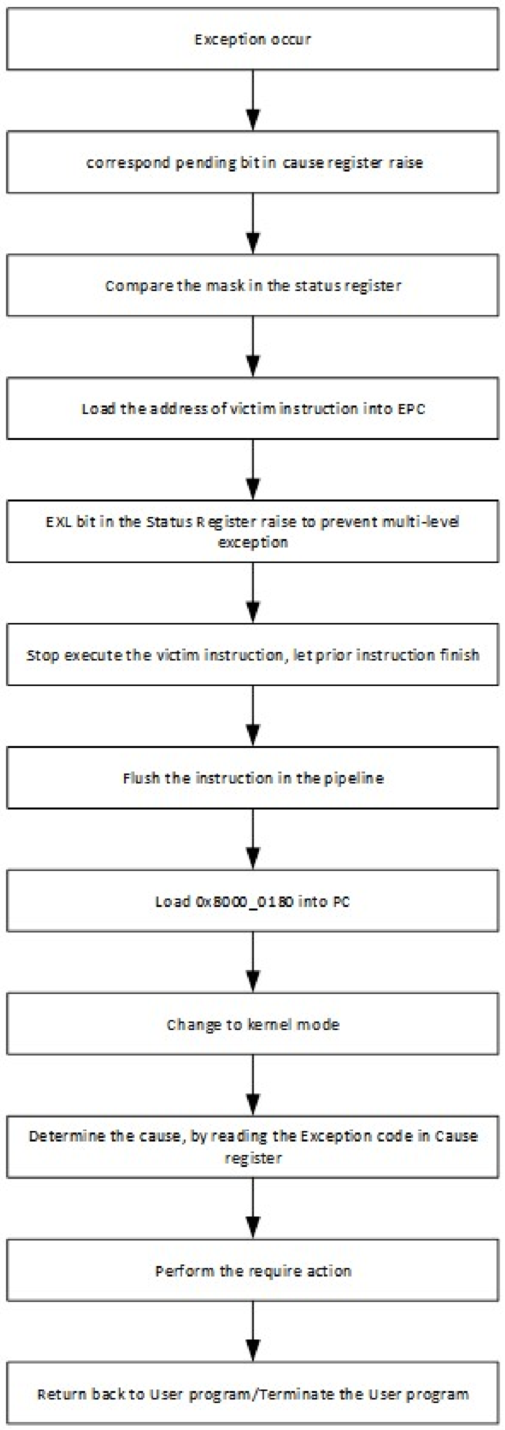

Exception occur
correspond pending bit in cause register raise
Compare the mask in the status register
Load the address of victim instruction into EPC
EXL bit in the Status Register raise to prevent multi-level exception
Stop execute the victim instruction, let prior instruction finish
Flush the instruction in the pipeline
Load 0x8000_0180 into PC
Change to kernel mode
Determine the cause, by reading the Exception code in Cause register
Perform the require action
Return back to User program/Terminate the User program

Figure 2.9 Flow chart for handle an exception.

For above solution, it look completely fine but it only can handle one level exception, which means that it cannot interrupt when inside the exception handler. Sometime we need to interrupt when interrupt is serving this also known as multi-level exception or Nested Exception.

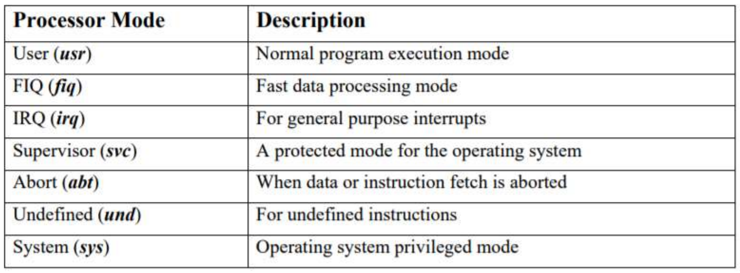
2.5 Nested Exception

Nested Exception is the permit for other exception to occur when the system are serving an exception. When exception occur, CP0 will write to the cause register, status register and the EPC. For nested exception, value in the cause register, status register and EPC was expected to be overwrite (Sweetman, 2007, pp.114-115). To support the nested exception, we need to store the value in cause register, status register and EPC value inside the stack. However, interrupt will also occur when copying the value to the stack. To solve this problem, we need to disable all interrupt when copying the value to the stack. We can implement the Interrupt Priority Level (IPL) by control the masking value of Status register to disable the further interrupt. However, the interrupt resource was limited, interrupt have a chance to occur when changing the value in Status Register causes the Status Register to be overwrite. This problem also known as the Race Condition. To solve this problem, we need to make the program mutual exclusion by using the software way, which is semaphores, to allow atomic changes of Status register.

2.6 Exception handler scheme for ARM processor

2.6.1 Processor Mode for ARM

The ARM processor internally has seven different modes of operation, which are, User mode, FIQ mode, IRQ mode, Supervisor mode, Abort mode, Undefined mode, System mode. The following figure summarizes the seven modes.



Processor Mode	**Description**
User (***usr***)	Normal program execution mode
FIQ (***fiq***)	Fast data processing mode
IRQ (***irq***)	For general purpose interrupts
Supervisor (***svc***)	A protected mode for the operating system
Abort (***abt***)	When data or instruction fetch is aborted
Undefined (***und***)	For undefined instructions
System (***sys***)	Operating system privileged mode

Figure 2.10 ARM Processor Mode

For user mode, it used for normal program execution. FIQ mode used for interrupts requiring fast response for instance data transfer with DMA. IRQ mode used for general-purpose interrupts for example I/O interrupt. Supervisor mode used when operating system support needed. Abort mode used when data or instruction fetch have aborted. Undefined mode used when undefined instruction fetched. System mode is the Operating system privilege mode for users.

2.6.2 ARM Exception

ARM have support few type of exception, which are Fast Interrupt Request, Interrupt Request, Software interrupt (SWI) and Reset, Prefetch Abort and Data Abort and Undefined Instruction. Figure 2.11 summaries the type of exception support by ARM. Figure 2.12 shown the priority level for ARM exception.

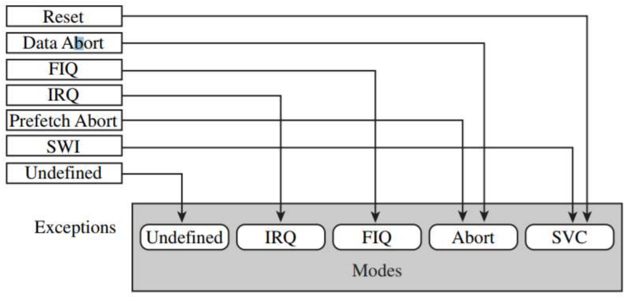
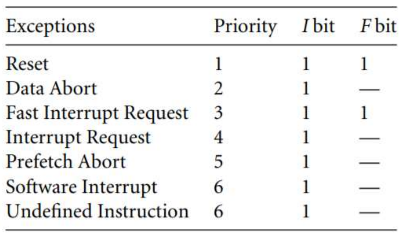


Figure 2.11 ARM processor exceptions and associated modes. (Sloss, Symes&Wright, 2004, p.319).



Exceptions	Priority	*I* bit	*F* bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instruction	6	1	—

Figure 2.12 Exception priority levels for ARM. (Sloss, Symes&Wright, 2004, p.319).

2.6.3 Entering and exiting an exception handler.

Sloss, Symes and Wright (2004) list out the step of ARM processor to handle an exception. First, preserve the address of the next instruction, copy the Current Program Status Register (CPSR) to Saved Program Status Register (SPSR) and the Program counter to the Link Register (LR). Next, force the CPSR mode bits to a value depending on the raised exception, force the Program counter (PC) to fetch the next instruction from the exception vector table. Now the handler is running in the mode associated with the raised exception. When handler is done, the CPSR restored from the saved SPSR. PC restored with the value of (LR – offset) and the offset value depends on the type of the exception. Last, clear the interrupt disable flags if they were set.

2.6.4 ARM Interrupt handling schemes

Sloss, Symes and Wright (2004) has introduce some interrupt handling scheme, which are, non-nested interrupt handling, nested interrupt handling, re-entrant interrupt handling, prioritized simple interrupt handling, prioritized standard interrupt handling, prioritized direct interrupt handling and prioritized grouped interrupt handling.

➢ **Non-nested interrupt handling**

This non-nested interrupt handling is the simplest scheme, it only allow one interrupt occur in concurrently. Once the processor received an interrupt, it will disable other interrupt and save the current context into SPSR. After that, jump to the exception handler to identify the interrupt source and jump to appropriate Interrupt service routine (ISR). After service the interrupt, restore the context from SPSR and re-enable the interrupt. The flow chart for non-nested interrupt handling shown in figure below.

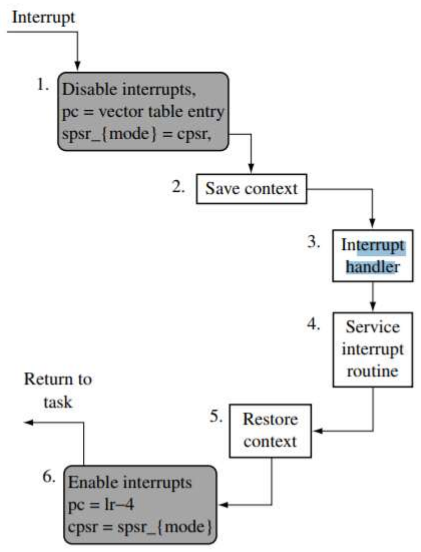

Interrupt
1. Disable interrupts, pc = vector table entry spsr_{mode} = cpsr,
2. Save context
3. Interrupt handler
4. Service interrupt routine
5. Restore context
6. Enable interrupts pc = lr–4 cpsr = spsr_{mode}
Return to task

Figure 2.13 Non-nested interrupt handling. (Sloss, Symes&Wright, 2004, p.334).

These non-nested interrupt handling scheme are not suitable for complex embedded system which has multiple interrupt occurrences and it has high interrupt latency but it is easy to implement and debug.

➢ **Nested interrupt handling**

The scheme can support multiple interrupt in the same time. This achieved by re-enabling interrupts before the interrupt has fully served. This feature will increases the complexity of the system but improves the latency. The scheme should be designed carefully to ensure the context saving and restoration from being interrupted. The goal of nested handling is to respond to interrupts quickly. The flow chart for nested interrupt handling shown in figure below.

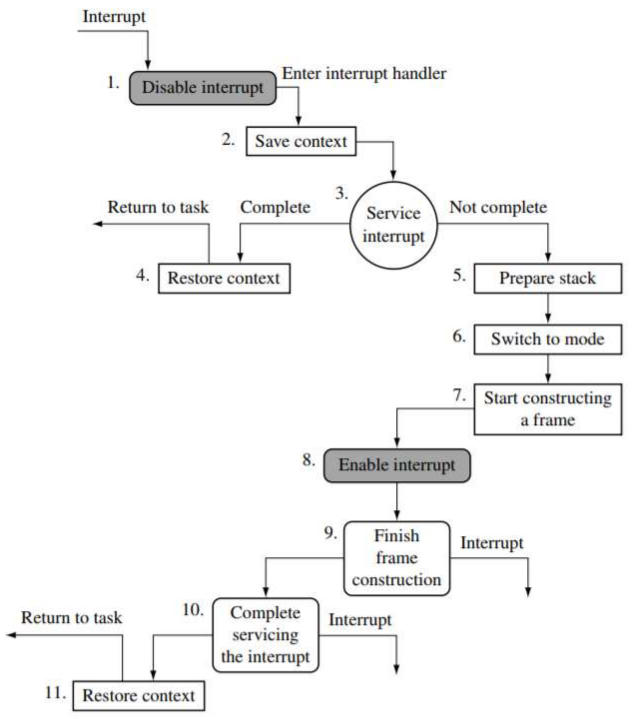


Figure 2.14 Nested interrupt handling. (Sloss, Symes & Wright, 2004, p.337).

The disadvantage of scheme is that it does not differentiate interrupts by priorities, so lower priority interrupt can block higher priority interrupts, it will cause deadlock.
The advantage is it can handle multiple interrupt in the same time and improve latency.

➢ **<u>Re-entrant Interrupt Handling</u>**

The difference between this scheme and the nested interrupt handling is re-enable interrupts earlier on the re-entrant interrupt handler compare to the nested interrupt handling. This can reduce interrupt latency. The external interrupt is clear before re-enabling interrupts to protect the system from infinite interrupt sequence. This is done by a using a mask in the interrupt controller. By using this mask, prioritizing interrupts is possible but this handler is more complex. The flow chart for Re-entrant interrupt handling shown in figure below.

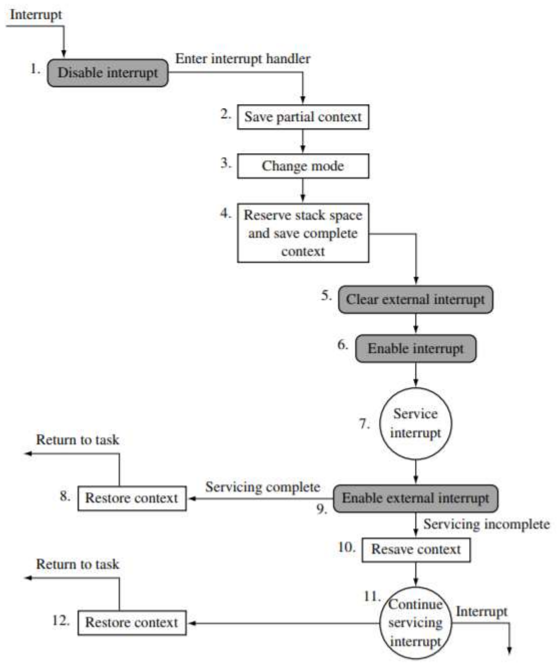


Figure 2.15 Re-entrant interrupt handling. (Sloss, Symes & Wright, 2004, p.343).

The advantage of this scheme are it can handle multiple interrupt with the differing priority level and it provide low latency but the scheme will be more difficult to build.

➢ **<u>Prioritized Simple Interrupt handling</u>**

In this scheme, the handler will associate a priority level with a particular interrupt source. A higher priority interrupt will take precedence over a lower priority interrupt. (Sloss, Symes&Wright, 2004, p.319). Handling prioritization can done by means of software or hardware. In case of hardware prioritization, the handler is simpler to design because the interrupt controller will give the interrupt signal of the highest priority interrupt requiring service. However, on the other side, the system needs more initialization code at start-up since priority level tables have to construct before the system switched on. When an interrupt signal raised, a fixed amount of comparisons will be compare with the available set of priority levels. The flow chart for Prioritized Simple interrupt handling shown in figure below.

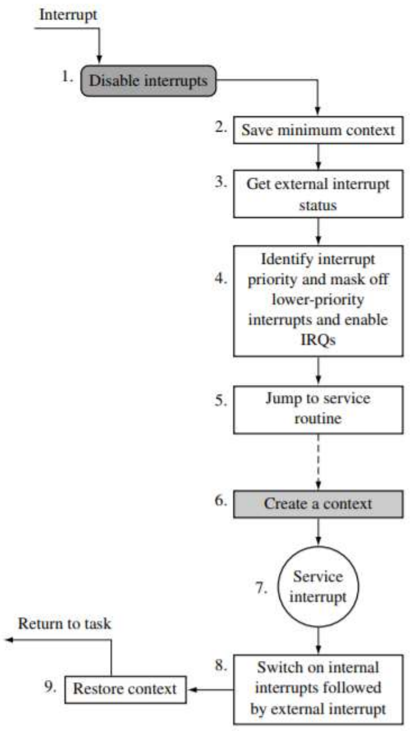

Interrupt
1. Disable interrupts
2. Save minimum context
3. Get external interrupt status
4. Identify interrupt priority and mask off lower-priority interrupts and enable IRQs
5. Jump to service routine
6. Create a context
7. Service interrupt
8. Switch on internal interrupts followed by external interrupt
9. Restore context
Return to task

Figure 2.16 Prioritized Simple interrupt handling. (Sloss, Symes & Wright, 2004, p.348).

The advantage for this scheme is it can handles prioritized interrupts and low interrupt latency. The low priority interrupt cannot take the precedence over the higher priority interrupt, with this feature, it solve the deadlock problem. The disadvantage for this scheme is the time taken to get to a low-priority service routine is same, as high-priority service routine and it cannot support multiple interrupt occurrence.

➢ **<u>Prioritized Standard Interrupt Handling</u>**

This scheme is the alternative of prioritized simple interrupt handler. It has the advantage of low interrupt latency for higher priority interrupts than the lower priority interrupt. The flow chart for Prioritized Simple interrupt handling shown in figure below.

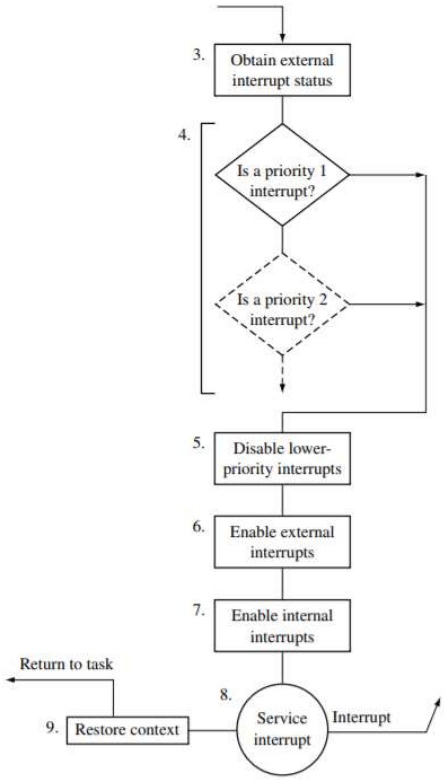

3.
Obtain external interrupt status
4.
Is a priority 1 interrupt?
Is a priority 2 interrupt?
5.
Disable lower-priority interrupts
6.
Enable external interrupts
7.
Enable internal interrupts
8.
Service interrupt
Interrupt
9.
Restore context
Return to task

Figure 2.17 Prioritized Standard interrupt handling. (Sloss, Symes & Wright, 2004, p.353).

- **<u>Prioritized Direct Interrupt Handling</u>**

There are two different between the prioritized direct interrupt handler and the prioritized standard interrupt handler. Some of the processing move to the individual ISR from the handler. Each individual ISR have the responsible to mask out the lower priority interrupt. This type of handler is relatively simple since the masking done by the individual ISR, but there are code duplication in each individual ISR since each interrupt service routine have to mask out the lower-priority interrupt that is same operation.

- **<u>Prioritized Grouped Interrupt Handler</u>**

This handler designed to handle large amount of interrupts by grouping interrupts together and forming a subset that can have a priority level. This way of grouping reduces the complexity of the handler since it does not scan through every interrupt to determine the priority. If the prioritized grouped interrupt handler is well design, it will improve the overall system response times dramatically, on the other hand if it is badly design such that interrupts are not group well, and then some important interrupts will dealt as low priority interrupts and vice versa. The most complex and possibly critical part of such scheme is the decision on which interrupts should be group together. The advantage for this scheme are can handle a large number of interrupts, and reduces the response time since the time taken to determine the priority level is shorter but it is difficult to group the interrupt.

Chapter 3 Proposed Methods / Technologies Involved

3.1 General Work Flow

The basic approach to develop an exception scheme is to identify type of the exception need to be support. After that, develop an Exception scheme that can handle multiple interrupt. Next, set up the test benches by using Verilog HDL to verify the exception scheme. Simple verification can done by trigger the I/Os interrupt and exception individually. After the IoT processor has passed through the individual test, a definitive exception-handling scheme can be derive from the combination of the various type of exception. If the scheme not functioning well, the exception scheme needs to redesign and go through the process again. If the scheme works correctly, then the work can be document. Next, the IoT processor is ready to synthesize onto an FPGA board for physical tests to conclude the earlier laid down experiments / tests. The Design Flow shown in Figure 3.1.

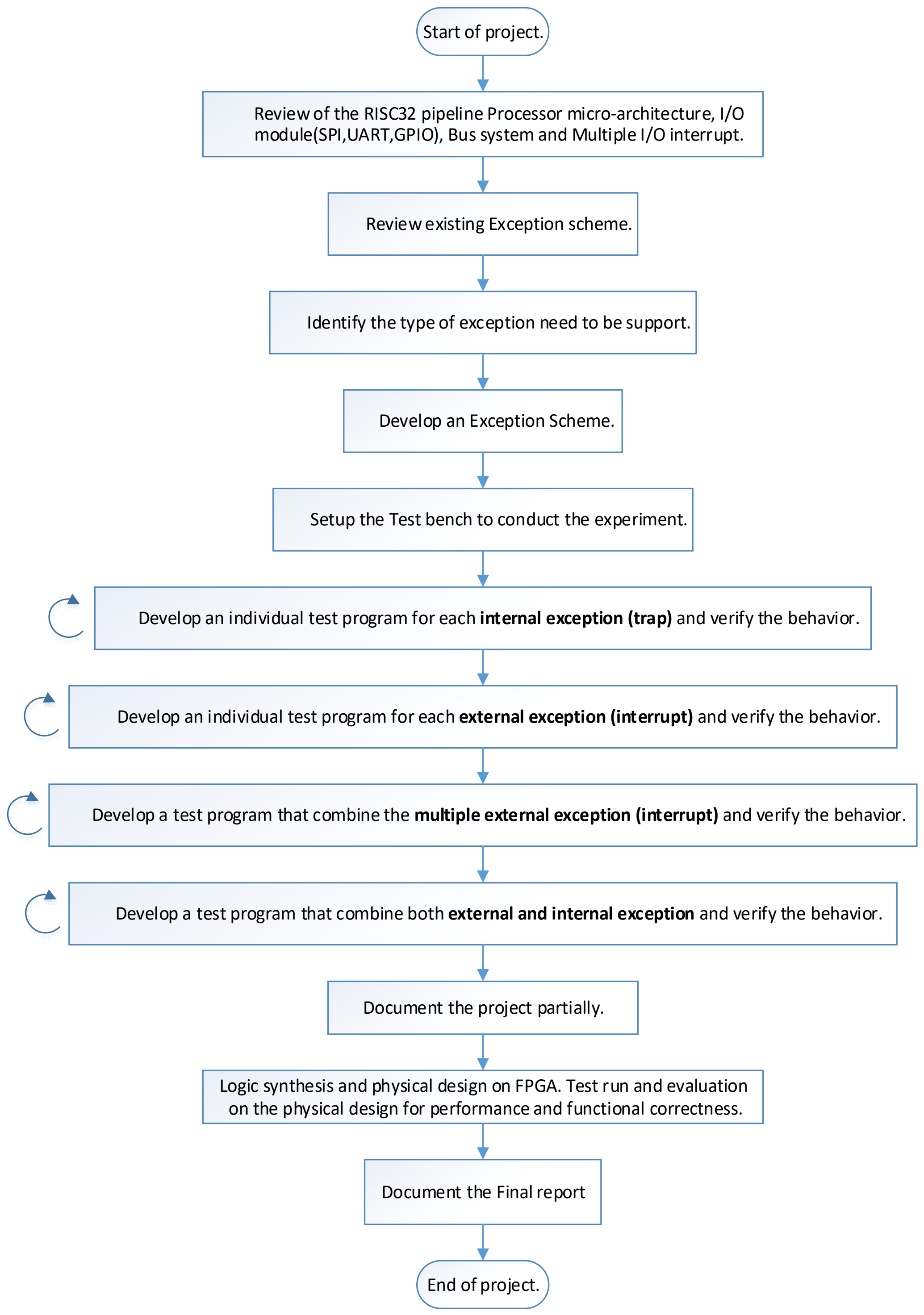

Start of project.
Review of the RISC32 pipeline Processor micro-architecture, I/O module(SPI,UART,GPIO), Bus system and Multiple I/O interrupt.
Review existing Exception scheme.
Identify the type of exception need to be support.
Develop an Exception Scheme.
Setup the Test bench to conduct the experiment.
Develop an individual test program for each internal exception (trap) and verify the behavior.
Develop an individual test program for each external exception (interrupt) and verify the behavior.
Develop a test program that combine the multiple external exception (interrupt) and verify the behavior.
Develop a test program that combine both external and internal exception and verify the behavior.
Document the project partially.
Logic synthesis and physical design on FPGA. Test run and evaluation on the physical design for performance and functional correctness.
Document the Final report
End of project.

Figure 3.1 Design flow of the project

3.2 Design Tools

Since this project will be using Verilog HDL to model the test bench to verify the functionality, we will be discussing commonly used design software that can support Verilog HDL. Some simulator are shows in Table 3.1:

Simulator	ModelSim	VCS	Quartus II
Company	Mentor Graphics	SYNOPSYS Predictable Success	ALTERA
Language Supported	VHDL-2002 V2001 SV2005	VHDL-2002 V2001 SV2005	VHDL-2002 V2001 SV2005
Platform Supported	-Windows XP/Vista/7/8/10 -Linux	Linux	-Windows XP/7/8 -Linux
Availability for free	YES (Student Edition only)	No	No

Table 3.1 Comparison between simulation tools. (Mentor Graphics, n.d.), (Synopsys, n.d.), (Altera, n.d.).

Based on the comparison above, it is clear that ModelSim from Mentor Graphic is the best choice as a simulation tool for this project because they offer free license (180 days) for Student Edition. There will some limitation for the student edition but it is sufficient for this project. However, the other two simulation tools provide better feature compare to ModelSim but the price is too expensive, it is not affordable for a student. ModelSim also provided freely in the computer laboratory by Universiti Tunku Abdul Rahman.

<u>Mentor Graphics ModelSim PE Student Edition 10.4a (Mentor Graphics, n.d.)</u>

ModelSim PE Student Edition 10.4a is the latest version and it offers a free license for academic purpose. It supports VHDL and Verilog HDL designs but not mixed and it has a friendly GUI with TCL interface. Since it is free, so the free version has no customer support, but there are a lot of learning resources are available on the internet and a forum for discussion.

<u>PC Spim</u>

PC Spim is a simulator that provides a MIPS environment to simulate MIPS programs. It supported almost the entire MIPS assembly language and a build-in simple debugger. In this project, the test code, boot loader, interrupt service routine and exception handler will write in MIPS assembly language and simulated with PC spim before load into the RISC32 IoT processor.

<u>Xilinx Vivado Design Suite- HLx Editions (Xilinx.com, n.d.)</u>

Xilinx Vivado used for synthesis and analysis of HDL designs. It allow the developer to synthesize their designs on to FPGA board, analysis RTL schematic diagrams, run the simulation, perform timing analysis, and load the bit stream to the target device. The FPGA board that used in this project is Arty A7: Artix-7 FPGA Development Board.

<u>Arty A7: Artix-7 FPGA Development Board (Digilent, n.d.)</u>

Arty is a ready-to-use development board and designed based on the Artix-7™ FPGA from Xilinx. It contain 256MB DDR3L and 16MB Quad-SPI Flash. For peripheral, Arty supported by the UARTs, SPIs, IICs, and an Ethernet MAC. It also contain 4 Pmod connector for expansion the connection. There are also some interaction and sensory devices such as 4 Switches, 4 Buttons, 1 Reset Button, 4 LEDs and 4 RGB LEDs.

3.3 System Overview

The Figure 3.2 shown the block diagram of the IoT SoC Processor and Table 3.2 shown hardware features of the IoT SoC Processor

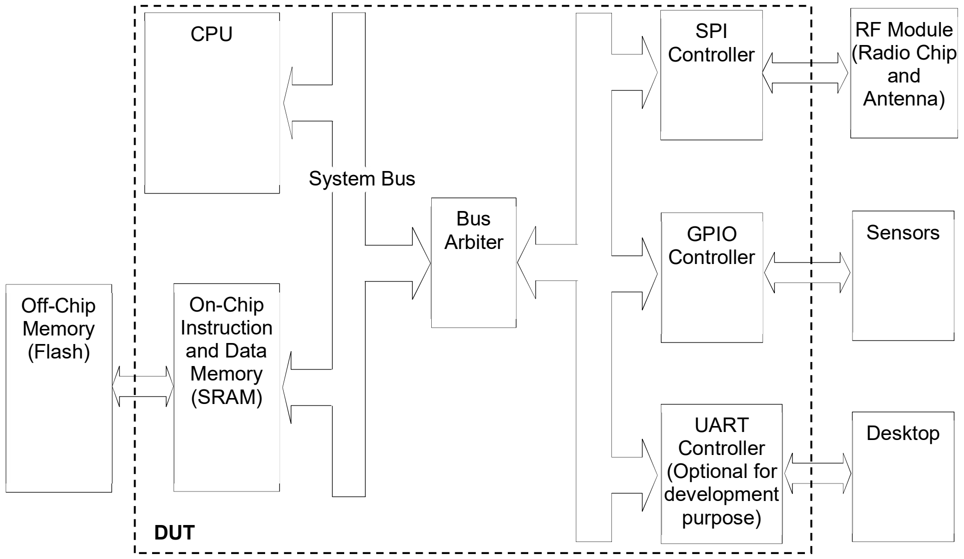

CPU
System Bus
Bus Arbiter
SPI Controller
RF Module (Radio Chip and Antenna)
GPIO Controller
Sensors
Off-Chip Memory (Flash)
On-Chip Instruction and Data Memory (SRAM)
UART Controller (Optional for development purpose)
Desktop
DUT

Figure 3.2 Block diagram of the IoT SoC Processor

CPU Structure	Pipeline
Instruction cycle	5, overlapping
CPU features	Control unit Data-path unit Branch predictor (64 entries 4 ways associative) Pipeline registers Hazard circuitry Interlock circuitry
Memory features	4kBytes boot ROM, 128kBytes user access flash, 8kBytes RAM (Data & Stack), 1kBytes I-cache, 32Bytes d-cache, 512Bytes Memory Mapped I/O Register
Communication interface features	UART, SPI, 32 GPIO pins

Table 3.2 Hardware features of the IoT SoC Processor

The figure above shows the system overview of this project. RISC32 IoT processor made up of 3 major part, which are, Central processing Unit (CPU), memory system and I/O System. The IoT processor based on pipeline architecture with build-in coprocessor 0 and Programmable interrupt controller to handle the exception. The CPU is compatible to the 5-stage 32-bit MIPS Instruction Set Architecture (ISA).

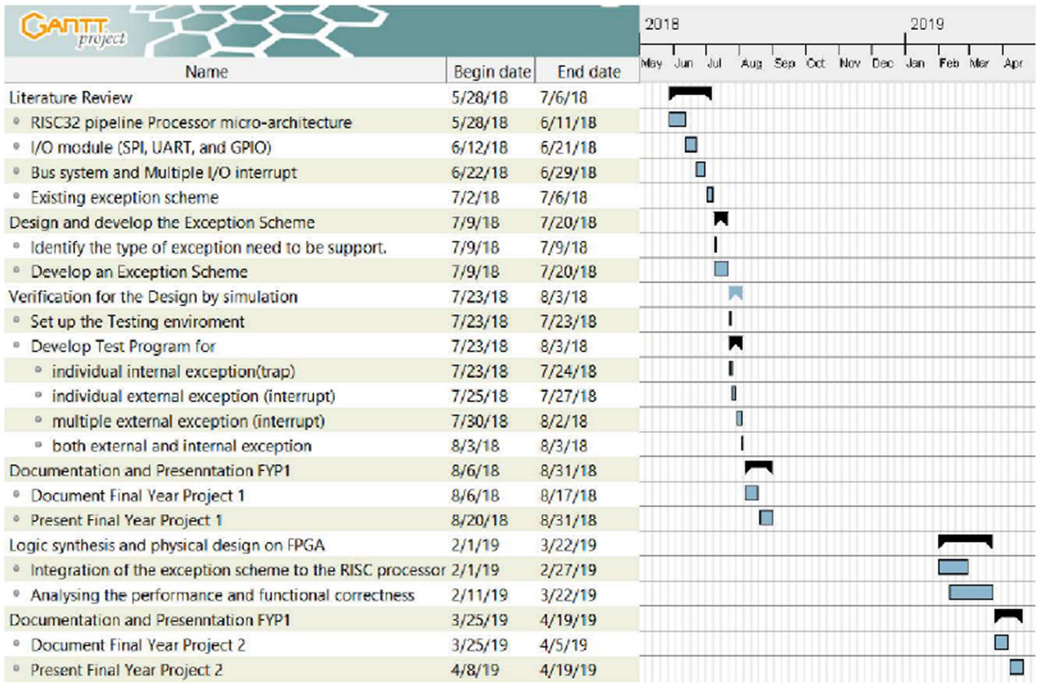
Memory unit will used to store the system code, user program and data. The I/O register mapped to the memory unit because I/O mapped technique used.

The I/O System of IoT Processor consist of SPI controller, GPIO controller, and UART controller. These controllers will responsible for data transmission between IoT processor and the external device, for example, sensors, wireless modules, personal computers.

The bus system will connect between CPU and I/O devices. Any data transmission between the CPU and the I/O device will pass through the bus system.

3.4 Timeline

Figure below show the Gantt chart for FYP 1 and FYP2.



Name	Begin date	End date
Literature Review	5/28/18	7/6/18
▪ RISC32 pipeline Processor micro-architecture	5/28/18	6/11/18
▪ I/O module (SPI, UART, and GPIO)	6/12/18	6/21/18
▪ Bus system and Multiple I/O interrupt	6/22/18	6/29/18
▪ Existing exception scheme	7/2/18	7/6/18
Design and develop the Exception Scheme	7/9/18	7/20/18
▪ Identify the type of exception need to be support.	7/9/18	7/9/18
▪ Develop an Exception Scheme	7/9/18	7/20/18
Verification for the Design by simulation	7/23/18	8/3/18
▪ Set up the Testing enviroment	7/23/18	7/23/18
▪ Develop Test Program for	7/23/18	8/3/18
▪ individual internal exception(trap)	7/23/18	7/24/18
▪ individual external exception (interrupt)	7/25/18	7/27/18
▪ multiple external exception (interrupt)	7/30/18	8/2/18
▪ both external and internal exception	8/3/18	8/3/18
Documentation and Presenntation FYP1	8/6/18	8/31/18
▪ Document Final Year Project 1	8/6/18	8/17/18
▪ Present Final Year Project 1	8/20/18	8/31/18
Logic synthesis and physical design on FPGA	2/1/19	3/22/19
▪ Integration of the exception scheme to the RISC processor	2/1/19	2/27/19
▪ Analysing the performance and functional correctness	2/11/19	3/22/19
Documentation and Presenntation FYP1	3/25/19	4/19/19
▪ Document Final Year Project 2	3/25/19	4/5/19
▪ Present Final Year Project 2	4/8/19	4/19/19

Figure 3.3 Gantt chart For FYP1 and FYP2.

Chapter 4 System Specification

4.1 System Overview

The IoT SoC processor is made up by 3 major parts, which are, Central Processing Unit (CPU), memory system and I/O system. The CPU is the subset of the 5-stage pipeline 32-bit MIPS Instruction Set Architecture (ISA). It supports up to 50 instructions, included arithmetic, logical, data transfer, program control and system instruction classes. The memory system consists of a 2-level memory hierarchy. First level consists of cache, Boot ROM and Data and Stack RAM and second level consists of Flash memory. The I/O system consists of GPIO controller, SPI controller, UART controller and Priority Interrupt controller. The I/O system integrated with CPU through Wishbone B4 standard bus interface (OpenCores, 2010). GPIO, SPI and UART controllers are used to data transfer with the external devices, for example, sensors, wireless modules, personal computers etc. The Priority Interrupt controller used as an external interrupt controller to handle multiple interrupt occurrences based on priority level. It collaborate with coprocessor 0 to handle the exception. Figure4.1 shows the architecture of the IoT SoC Processor. Table 4.1 shows the hardware feature of the IoT SoC Processor.

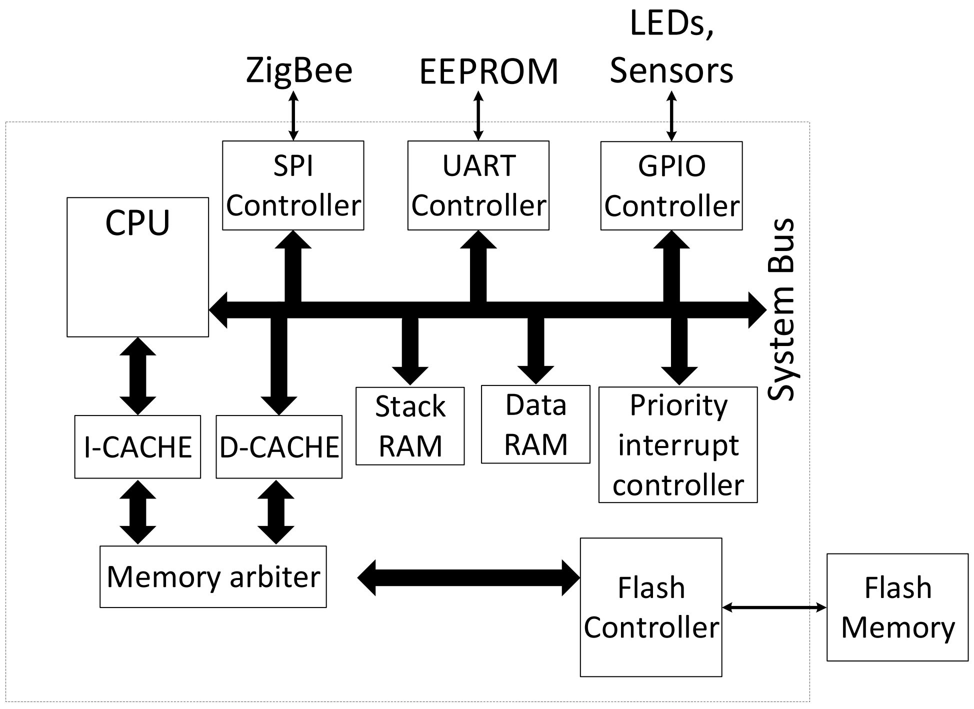


Figure 4.1: Architecture of the IoT SoC Processor.

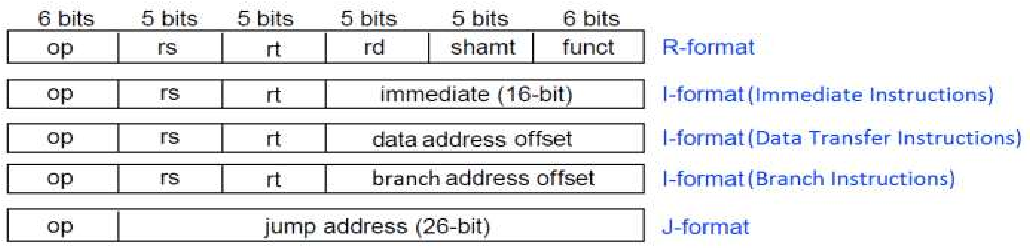
CPU Structure	Pipeline
Instruction cycle	5, overlapping
CPU features	Control unit Data-path unit Branch predictor (64 entries 4 ways associative) Pipeline registers Hazard circuitry Interlock circuitry
Memory features	4kBytes boot ROM, 128kBytes user access flash, 8kBytes RAM (Data & Stack), 1kBytes i-cache, 32Bytes d-cache, 512Bytes Memory Mapped I/O Register
Communication interface features	UART, SPI, 32 GPIO pins

Table 4.1: Hardware features of the IoT SoC Processor.

4.2 MIPS ISA

4.2.1 Instruction Format

There are 3 Instruction format which are, R-format, I- format and J-format. Each MIPS instruction must belong to one of these formats. Figure 4.2 shows the MIPS instruction format.



6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
op	rs	rt	rd	shamt	funct	R-format
op	rs	rt	immediate (16-bit)			I-format(Immediate Instructions)
op	rs	rt	data address offset			I-format(Data Transfer Instructions)
op	rs	rt	branch address offset			I-format(Branch Instructions)
op	jump address (26-bit)					J-format

Figure 4.2 Instruction Format

4.2.2 Addressing modes

There are six addressing modes, which are register addressing mode, immediate addressing mode, base addressing mode, pc-relative addressing mode, pseudo-direct addressing mode and Register direct addressing mode.

A) Register addressing mode(R-format)

Operand are in a system register. Perform operation based on function field. Action on Source and target register and store the result back to destination register.

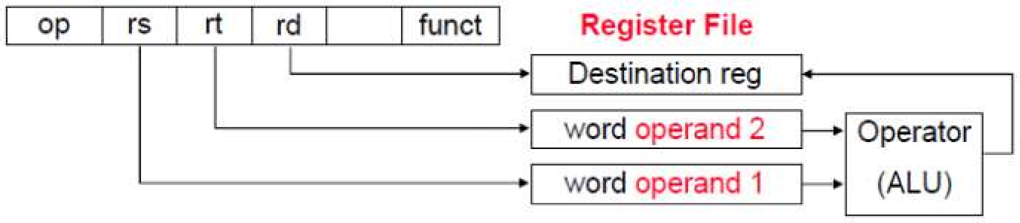


Figure 4.3 Register Addressing mode

B) Immediate addressing mode (I-format)

The operand is inside the instruction (data-value). Perform operation on source register and immediate value and store the result back into target register

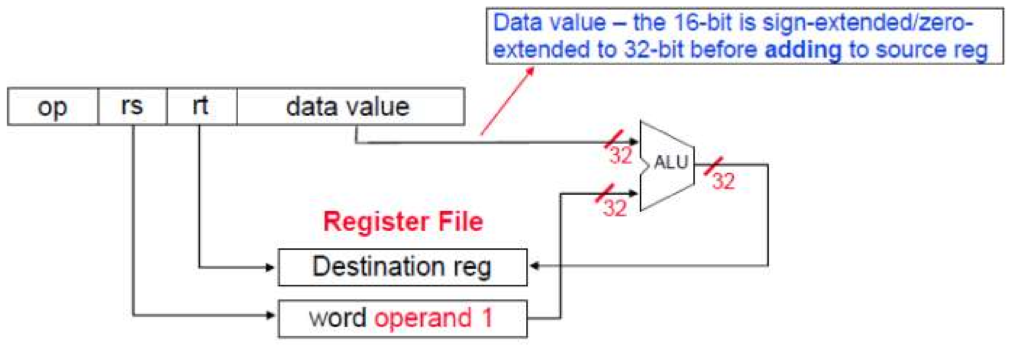

Data value – the 16-bit is sign-extended/zero-extended to 32-bit before adding to source reg
op
rs
rt
data value
32
ALU
32
32
Register File
Destination reg
word operand 1

Figure 4.4 Immediate Addressing mode

C) Base addressing mode (I-format)

Perform operation on source register and data address offset. The calculated result used as address to access the data memory to load/store data to/from target register.

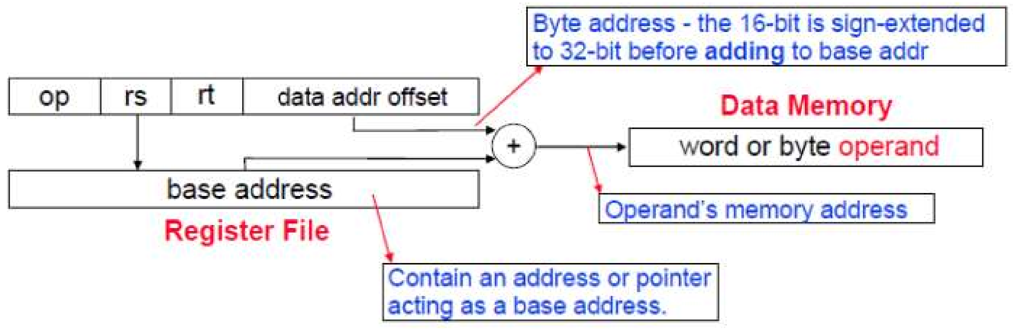

Byte address - the 16-bit is sign-extended to 32-bit before adding to base addr
op
rs
rt
data addr offset
Data Memory
+
word or byte operand
base address
Operand's memory address
Register File
Contain an address or pointer acting as a base address.

Figure 4.5 Base Addressing mode

D) PC-relative addressing (I-format)

Perform comparison on source and target register to determine branch taken or untaken, the immediate value is uses to calculate the branch target.

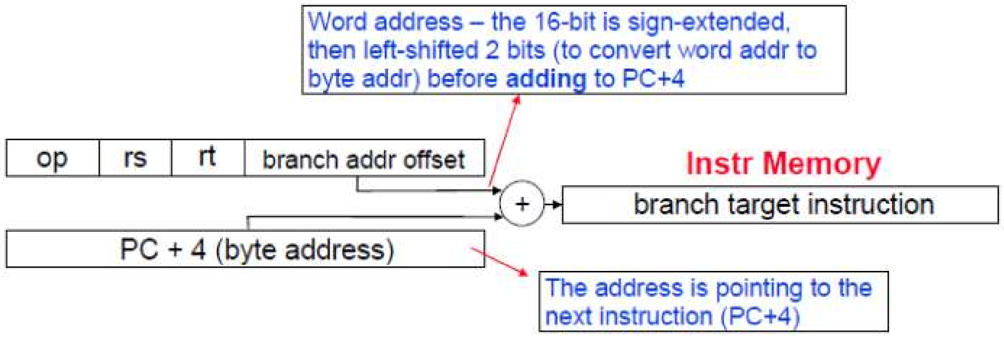

Word address – the 16-bit is sign-extended, then left-shifted 2 bits (to convert word addr to byte addr) before adding to PC+4
op
rs
rt
branch addr offset
+
Instr Memory
branch target instruction
PC + 4 (byte address)
The address is pointing to the next instruction (PC+4)

Figure 4.6 PC-Relative Addressing mode

E) Pseudo-direct addressing (J-format)

Perform operation by concatenating the upper bits of PC with the jump address offset, to calculate the jump target.

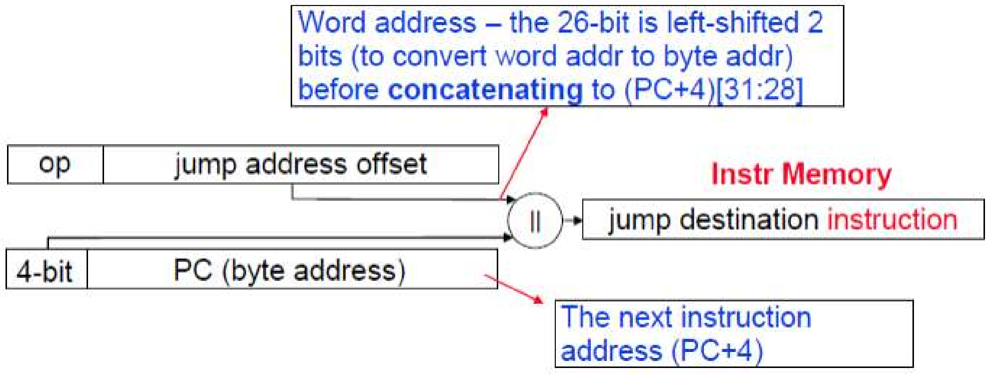

Word address – the 26-bit is left-shifted 2 bits (to convert word addr to byte addr) before concatenating to (PC+4)[31:28]
op
jump address offset
||
Instr Memory
jump destination instruction
4-bit
PC (byte address)
The next instruction address (PC+4)

Figure 4.7 Pseudo-Direct Addressing mode

F) Register direct addressing mode (J-format)

Take the value from the source register and force it into PC.

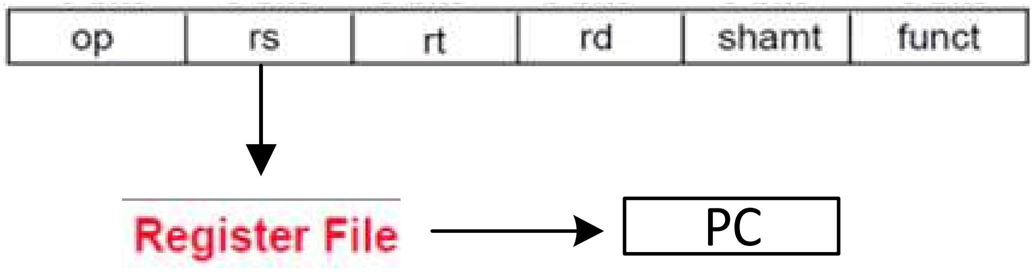


Figure 4.8 Register Direct Addressing mode

4.2.3 Instruction Supported

No	Instruction	opcode[31:26]	rs[25:21]	rt[20:16]	rd[15:11]	shamt[10:6]	funct[5:0]
		opcode[31:26]	rs[25:21]	rt[20:16]	immediate[15:0]		
		opcode[31:26]	address[25:0]				
1	add	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100000
2	addu	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100001
3	sub	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100010
4	subu	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100011
5	mult	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	011000
6	multu	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	011001
7	mfhi	000000	00000	00000	[xxxxx]	00000	010000
8	mflo	000000	00000	00000	[xxxxx]	00000	010010
9	and	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100100
10	or	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100101
11	xor	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100110
12	nor	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	100111
13	sll	000000	00000	[xxxxx]	[xxxxx]	[xxxxx]	000000
14	srl	000000	00000	[xxxxx]	[xxxxx]	[xxxxx]	000010
15	sra	000000	00000	[xxxxx]	[xxxxx]	[xxxxx]	000011
16	slt	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	101010
17	sltu	000000	[xxxxx]	[xxxxx]	[xxxxx]	00000	101011
18	jr	000000	[xxxxx]	00000	00000	00000	001000
19	jalr	000000	[xxxxx]	00000	[xxxxx]	00000	001001
20	syscall	000000	00000	00000	00000	00000	001100
21	mtc0	010000	00100	[xxxxx]	[xxxxx]	00000	000000
22	mfc0	010000	00000	[xxxxx]	[xxxxx]	00000	000000
23	eret	010000	00001	00000	00000	00000	011000
24	addi	001000	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]		
25	addiu	001001	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]		
26	andi	001100	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]		
27	ori	001101	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]		
28	xori	001110	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]		

29	lui	001111	00000	[xxxxx]	[xxxxxxxxxxxxxxxx]
30	lw	100011	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
31	lwl	100010	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
32	lwr	100110	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
33	lh	100001	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
34	lhu	100101	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
35	lb	100000	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
36	lbu	100100	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
37	sw	101011	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
38	swl	101010	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
39	swr	101110	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
40	sh	101001	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
41	sb	101000	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
42	slti	001010	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
43	sltiu	001011	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
44	beq	000100	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
45	bne	000101	[xxxxx]	[xxxxx]	[xxxxxxxxxxxxxxxx]
46	blez	000110	[xxxxx]	00000	[xxxxxxxxxxxxxxxx]
47	bgtz	000111	[xxxxx]	00000	[xxxxxxxxxxxxxxxx]
48	j	000010	[xxxxxxxxxxxxxxxxxxxxxxxxxx]		
49	jal	000011	[xxxxxxxxxxxxxxxxxxxxxxxxxx]		

Table 4.2 Instruction Supported

4.3 Functional View of the RISC32 Pipeline Processor

MIPS processor break the instruction execution cycle into 5-stage which is IF, ID, EX, MEM and WB stage and this technic call pipelining. Figure 4.9 shows the hardware component allocate in each pipeline stage.

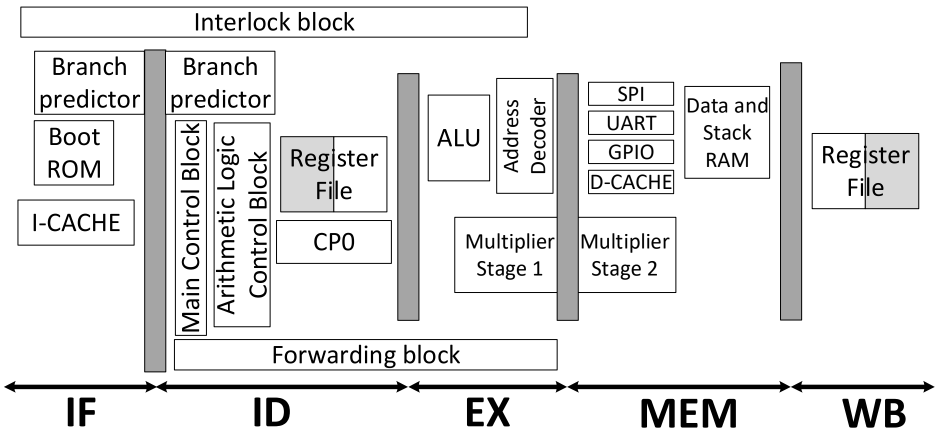


Figure 4.9 Functional view of the RISC32 processor

At IF stage, instruction fetched from the Boot ROM or I-CACHE and pass through the IF/ID pipeline. If the cache miss happed, the I-cache will send a signal to stall whole processor until respective instruction was fetch into the I-cache.

At ID stage, the control signal will be compute by decoding the instruction. The Main Control block and the Arithmetic Logic Control block will decode the instruction and send output signals. Output signals from both hardware components will pass through to the ID/EX pipeline and the remaining hardware components in the ID stage, which are, Register File, Forwarding block, Coprocessor 0, Branch Predictor and Interlock blocks.

At EX stage, ALU block covers all the operation except the multiplication operation. Multiplier block starts the multiplication operation at EX stage and requires 2 clock cycles (EX and MEM stages) to perform a multiplication operation on two 32-bit operands.

At the MEM stage, the load/store instruction will access the memory component, which are, D-cache, Data and Stack RAM and I/O register.

At WB stage, the computed result will write back to the register file.

4.4 Memory Map

Figure 4.10 shows the Memory map. Table 4.3 shows the details of the Memory allocation.

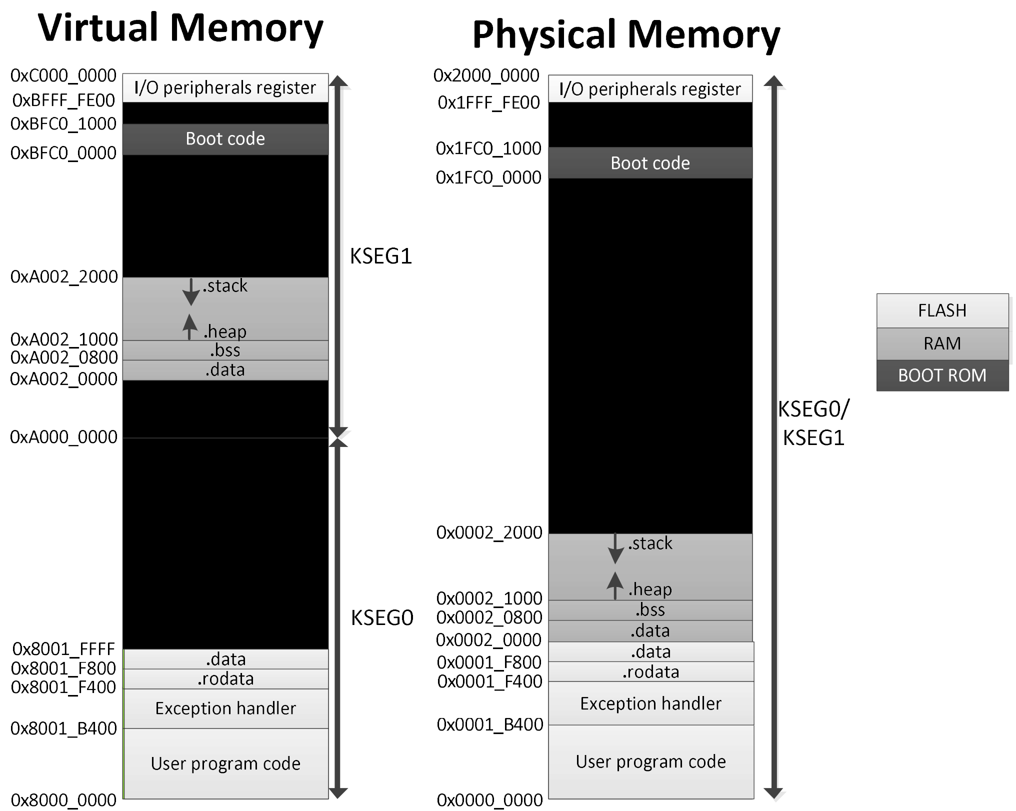


Figure 4.10 Memory Map

Purpose	Description	Size
I/O Peripherals register	External I/O device registers (I/O mapped technique)	512Bytes
Boot code	Start-up code which keep the system configuration(Boot loader)	4kBytes
Stack	Use for argument passing	8kBytes
Heap	Dynamic memory allocation such as malloc()	
Exception handler	Exception handler code and ISR	16kBytes
User Program Code	Store User Program Code	128kBytes

Table 4.3 Memory map description

When a processor start-up, the boot loader program stored in the boot ROM should perform the following actions:

1) Set up the Register File block registers value
2) Copy *.data* content from Flash memory to the Data RAM
3) Jump to user program code located at 0x8000_0000 (virtual address)

The data in *.data*, *.bss*, *.stack*, *.heap* and I/O peripherals registers can be accessed using load and store instructions.

When Exception occur, the program should jump into the single entry point of the Exception handler (0x8001_B400). After that, identify the cause of the exception and jump to the respective Interrupt service routine (ISR). After serving the exception, should jump back to the user program to continue execution.

4.5 RISC32 Pipeline Processor Hierarchy.

Table 4.4 shows the RISC32 pipeline processor hierarchy.

Chip Level	Unit Level (Micro-Architecture Level)	Block Level (Micro-Architecture Level)	Sub-block
crisc	Data-path unit (udata_path)	Branch Predictor block (bbp_4way)	
		Register File block (brf)	
		Forwarding block (bfw_ctrl)	
		Interlock block (bitl_ctrl)	
		CP0 block (bcp0)	
		ALU block (balb)	
		Multiplier Block (bmult32)	adder_lvl1_firstrow
			adder_lvl1
			add_lvl1_lastrow
			sub_lvl1_lastrow
			adder_lvl2
			adder_lvl2_lastrow
			adder_lvl3
			adder_lvl4
			adder_lvl5
		Address Decoder block (baddr_decoder)	
	Control-path unit (uctrl_path)	Main Control block (bmain_ctrl)	
		Arithmetic Logic Control block (balb_ctrl)	
	Cache unit (ucache)	Cache Controller block (bcache_ctrl)	
		Cache RAM block (bcache_ram)	
		FIFO Controller (bfifo_ctrl)	
		FIFO block(bfifo)	
	Flash Controller Unit (ufc)	Flash Controller Clock Generator block (bfc_clk_gen)	
		Flash Controller FSM block (bfc_fsm)	
		Flash Controller Transmitter block (bfc_TX)	
		Flash Controller Receiver block (bfc_RX)	
		FIFO block (bfc_FIFO)	

	Data and Stack RAM unit (uram)		
	UART Controller unit (uuart)	UART Baud Clock Generator block (bclkctr)	
		UART Receiver block (brx)	sbrx_ctr
			asynfifo_r1_3
			fifomem_b1_1
			graycntr_r1_3
			synchronizer
		UART Transmitter block (btx)	sbtx_ctr
			asynfifo_r1_3
			fifomem_b1_1
			graycntr_r1_3
			synchronizer
	SPI Controller unit (uspi)	SPI Clock Generator block (bclk_gen)	
		SPI Receiver block (bRX)	
		SPI Transmitter block (bTX)	
		FIFO block (bFIFO)	
		SPI Input Output Control block (bio_ctrl)	
	GPIO Controller unit (ugpio)		
	Priority Interrupt Controller unit (upi_ctrl)	Priority Resolver block (bpic_resolver)	
	Boot ROM unit (uboot_rom)		
	Memory Arbiter unit (umem_arbiter)		

Table 4.4 RISC32 processor hierarchy.

Chapter 5 Analysis of the I/O system

5.1 I/O System

The I/O system consists of GPIO controller, UART controller, SPI controller and Priority interrupt controller (PIC). There are integrate to the CPU through the Wishbone B4 standard bus interface (OpenCores, 2010). The GPIO controller, UART controller and SPI controller used to communicate with the external device and data transmission.

The GPIO controller, UART controller, SPI controller and Priority interrupt controller (PIC) integrated to the CPU by using the Memory-mapped I/O technique. The starting address for I/O Map is 0xbffffe00. For convenience, register $S0 was programme to store the starting address of the I/O Map. There are allow the user use load-store instruction to access the I/O register.

The UART controller used for asynchronous serial data communication between another UART devices. The SPI controller used for high-speed serial data communication between the SPI interfaces devices. It developed with 4 wires, which are Master out Serial in (MOSI), Master in Serial out (MISO), Slave Select (SS) and SPI clock (SCLK), and 4 modes of serial data communication. The General Purpose Input/output (GPIO) Controller is 32-bits I/O port. Each of the pin can be set as either input or output by configure the GPIODIR register. The GPIO Controller can be used for interact with the external devices. For example, blinking LEDs, debugging, digital input. (Kiat, 2018, pp. 91-108).

The Priority Interrupt Controller (PIC) is an external interrupt controller to handle the multiple interrupt occurrence based on interrupt priority level. Priority Interrupt Controller unit work with core processor 0 (CP0) to handle the exception. The Priority Interrupt Controller can take up the 8 interrupt source. The currently connected interrupt sources are SPI controller, UART controller and CP0 timer. There are four interrupt priority levels (IPL) can be set for each interrupt source. The highest priority interrupt will take precedence over a lower priority interrupt. (Kiat, 2018, pp. 109-112).

5.2 Micro-architecture for I/O system.

Figure 5.1 below shows the micro-architecture for the I/O system.

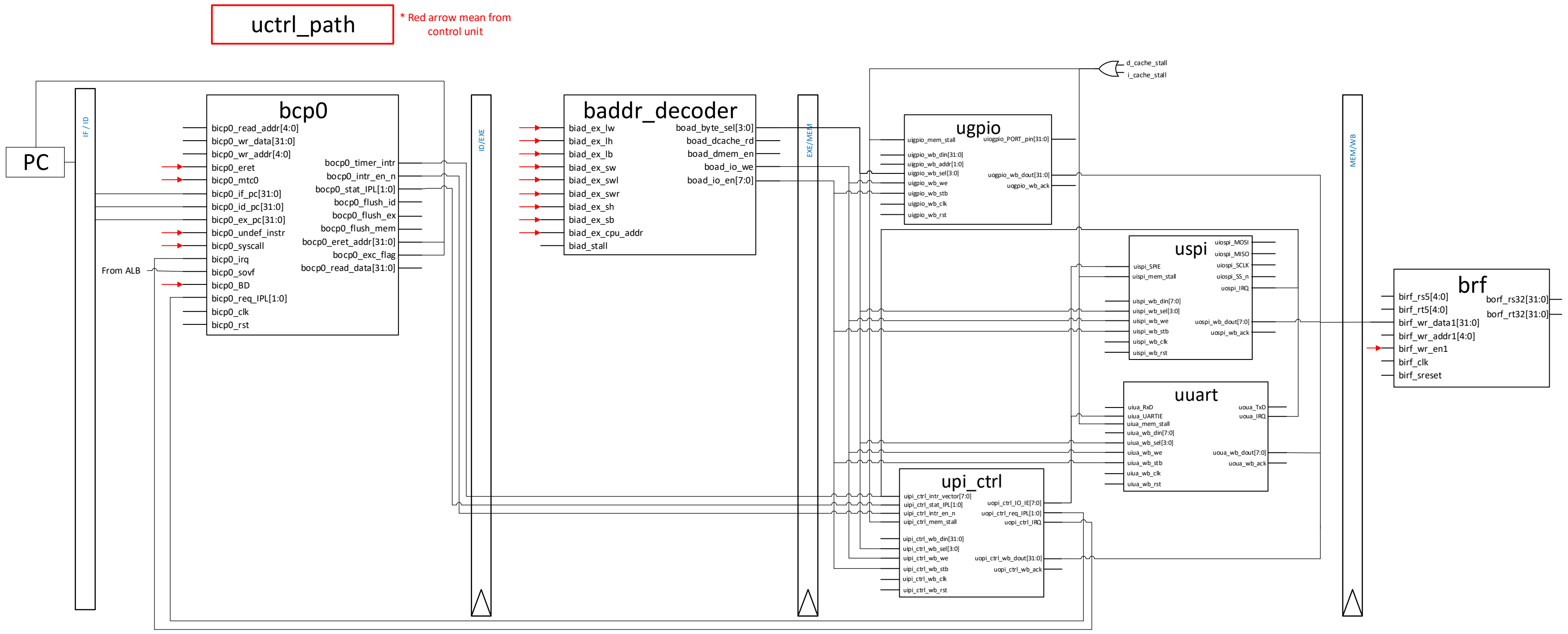


Figure 5.1 Micro-architecture for I/O system.

5.3 Co-processor 0

Co-processor 0 (CP0) is responsible to record and process the exception information. The CP0 block is able to process sign-overflow exception, undefined instruction exception, syscall exception and I/Os interrupt. Table 5.1 shows the details of resolving simultaneous exception occurrence.

Exception event at the same clock cycle		**Exception event occurring at which Stage?**	**Occurs at branch delay slot?**	**Return Address ($EPC)**	**Pipeline registers flush?**
Interrupt Request (IRQ)	**Other Exception**				
No	Overflow	EX stage (ALU block)	-	ID stage's PC	IF/ID, ID/EX and EX/MEM
No	Undefined Instruction	ID stage	-	IF stage's PC	IF/ID and ID/EX
No	Syscall	ID stage	-	IF stage's PC	IF/ID
Yes	-	-	No	IF stage's PC	IF/ID
Yes	-	-	Yes	ID stage's PC	ID/EX
Yes	Overflow	Overflow-EX stage	-	EX stage's PC	IF/ID, ID/EX and EX/MEM
Yes	Undefined Instruction	Undefined Instruction-ID stage	-	ID stage's PC	IF/ID and ID/EX
Yes	Syscall	Syscall-ID stage	-	ID stage's PC	IF/ID

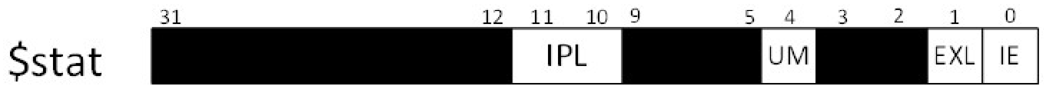
Table 5.1 Details for exception event.

<u>CP0 Register sets</u>

Table 5.2 shows the CP0 Register Sets. Table 5.3 show the each field for status register. Table 5.4 shows the each field for cause register.

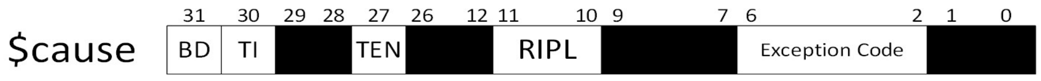
Register No.	Code	Register Name	Usage
0 - 8	00000 - 01000	RESERVED	RESERVED
9	01001	$count	count up every CPU cycle
10	01010	RESERVED	RESERVED
11	01011	$compare	Used with $count register to form a timer
12	01100	$stat	Store the control and status of exceptions
13	01101	$cause	Store the cause of exceptions
14	01110	$epc	Store exception return address
15 - 31	01111 - 11111	RESERVED	RESERVED

Table 5.2 Conventional usage of CP0 registers



Register	bit	usage
$stat	[31:12]	RESERVED
	IPL[11:10]	store current interrupt priority level
	[9:5]	RESERVED
	UM[4]	1=user mode, 0=kernel mode
	[3:2]	RESERVED
	EXL[1]	Exception level 1=exception occurs, disable further exception to occur 0=no exception occurs
	IE[0]	1=Interrupt enable 0=Interrupt disable

Table 5.3 Details of the Status register



Register	bit	usage
$cause	BD[31]	Indicate branch delay
	TI[30]	1=enable timer interrupt 0=disable timer interrupt
	[29:28]	RESERVED
	TEN[27]	CP0 Timer, $count disable control
	[26:12]	RESERVED
	RIPL[11:10]	User Define priority level for the active interrupt
	[9:7]	RESERVED
	Exception code [6:2]	encodes reasons for the exception 0=Interrupt 4=AdEL, address error trap (load or instruction fetch) 5= AdES, address error trap (store) 6=lBE, bus error on instruction fetch trap 7=DBE, bus error on data load or store trap 8=Sys, syscall trap 9=Bp, breakpoint trap 10=Rl, undefined instruction trap 12=Ov, arithmetic overflow trap
	[1:0]	RESERVED

Table 5.4 Details of the cause register.

5.4 Exception event.

Table 5.5 shows the Characteristic of RISC32 exception event. Refer to Chapter 2.2 for detail explanation of the exception event characteristics.

Exception Type	Synchronous Vs Asynchronous	User requested Vs Coerced	User maskable Vs user nonmaskable	Within Vs between instructions	Resume Vs Terminate
Exception					
#AdEL(address error exception (load or instruction fetch))	*	*	*	*	*
#AdES(address error exception (store))	*	*	*	*	*
#IBE(bus error on instruction fetch)	*	*	*	*	*
#DBE(bus error on data load or store)	*	*	*	*	*
sys(syscall exception)	Synchronous	user request	nonmaskable	Within	Resume
#Bp(breakpoint exception)	*	*	*	*	*
Ov(arithmetic overflow exception)	Synchronous	Coerced	nonmaskable	Within	Resume
RI(Reserved instruction/Undefined instruction)	Synchronous	Coerced	nonmaskable	Within	Resume
Interrupt					
UART	Asynchronous	Coerced	User maskable	Between	Resume
SPI	Asynchronous	Coerced	User maskable	Between	Resume
GPIO	*	*	*	*	*
Timer	*	*	*	*	*

*pending
#not implemented

Table 5.5 Exception event and its characteristics for RISC32 IoT Processor.

5.5 Priority for Exception Event

The characteristic for 5-stage pipeline processor is overlapping the instruction. Therefore, it is possible multiple exception occur in the same clock cycle. To handle this problem, there are needed a priority scheme to schedule the exception event when multiple exception event conflicting. Table 5.6 shows the priority for exception event

Exception event	**Priority(in ascending order)**		**Remark**
Internal			
Syscall	3(Lowest priority)		Syscall and Undefined instruction will occur in the ID stage, It will not conflict with each other.
Undefined Instruction	3(Lowest priority)		
Sign-Overflow	2		
External(IRQ)	1(Highest priority)		
INT_1		Highest priority	The priority between the External Exception (interrupt) is determine by the User by setting the interrupt priority level in the PICIPLLO[7:0] and PICIPLHI[7:0] of Programmable interrupt controller. (Kiat, 2018, pp. 109-112).
UART(INT_2)		↑	
SPI (INT_3)		↑	
INT_4		↑	
INT_5		↑	
INT_6		↑	
Timer (INT_7)		Lowest Priority	

Table 5.6 Priority for Exception Event

5.6 Exception Handler Scheme

The exception handler start at address 0x8001b400. When exception occur, CP0 will send a signal to flush the pipeline register based on the exception event (refer to Table 5.1). After that, exception flag will assert by CP0. Next, save the respective PC to the $epc for return purpose. After that, jump to the exception handler (0x8001b400). Figure 5.2 shows the flow of the exception handler.

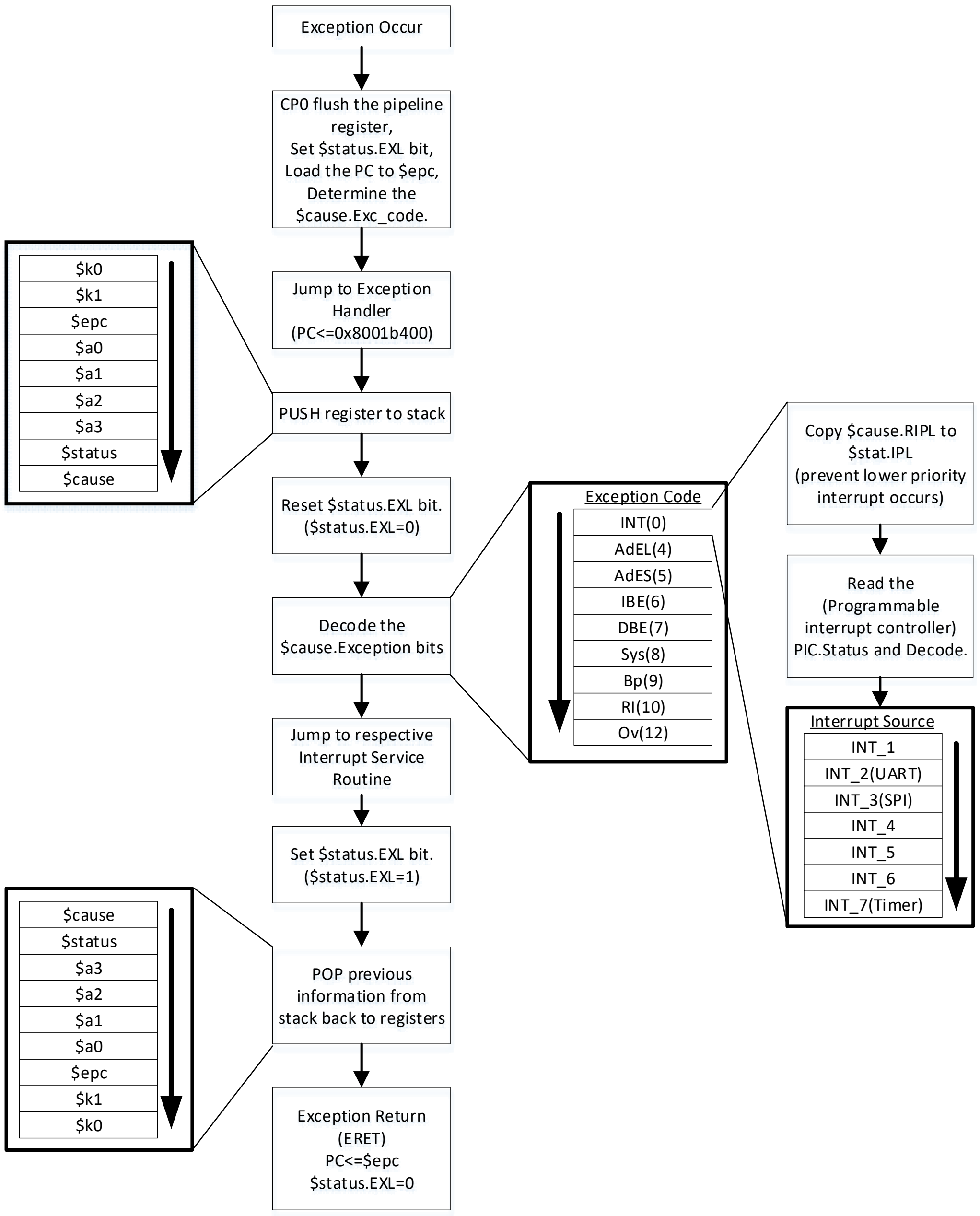

Exception Occur
CP0 flush the pipeline register, Set $status.EXL bit, Load the PC to $epc, Determine the $cause.Exc_code.
Jump to Exception Handler (PC<=0x8001b400)
PUSH register to stack
$k0
$k1
$epc
$a0
$a1
$a2
$a3
$status
$cause
Reset $status.EXL bit. ($status.EXL=0)
Decode the $cause.Exception bits
Exception Code
INT(0)
AdEL(4)
AdES(5)
IBE(6)
DBE(7)
Sys(8)
Bp(9)
RI(10)
Ov(12)
Copy $cause.RIPL to $stat.IPL (prevent lower priority interrupt occurs)
Read the (Programmable interrupt controller) PIC.Status and Decode.
Interrupt Source
INT_1
INT_2(UART)
INT_3(SPI)
INT_4
INT_5
INT_6
INT_7(Timer)
Jump to respective Interrupt Service Routine
Set $status.EXL bit. ($status.EXL=1)
POP previous information from stack back to registers
$cause
$status
$a3
$a2
$a1
$a0
$epc
$k1
$k0
Exception Return (ERET) PC<=$epc $status.EXL=0

Figure 5.2 Flow of the Exception Handler

Nested interrupt Handling Scheme

Figure 5.3 shows the flow of exception handling, when there are another interrupt request during the execution of the ISR.

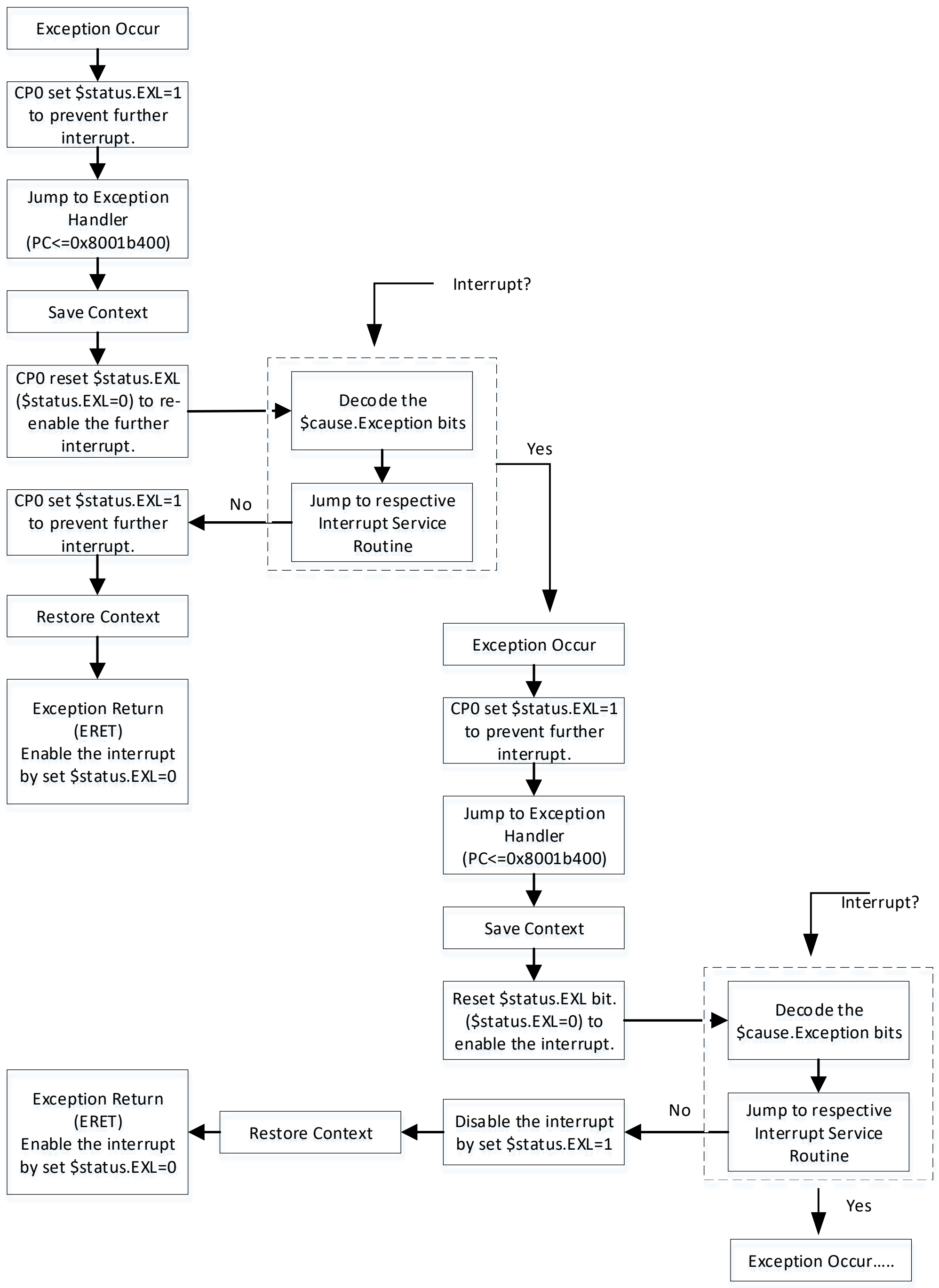

Exception Occur
CP0 set $status.EXL=1 to prevent further interrupt.
Jump to Exception Handler (PC<=0x8001b400)
Save Context
CP0 reset $status.EXL ($status.EXL=0) to re-enable the further interrupt.
Interrupt?
Decode the $cause.Exception bits
Jump to respective Interrupt Service Routine
Yes
No
CP0 set $status.EXL=1 to prevent further interrupt.
Restore Context
Exception Return (ERET) Enable the interrupt by set $status.EXL=0
Exception Occur
CP0 set $status.EXL=1 to prevent further interrupt.
Jump to Exception Handler (PC<=0x8001b400)
Save Context
Reset $status.EXL bit. ($status.EXL=0) to enable the interrupt.
Interrupt?
Decode the $cause.Exception bits
Jump to respective Interrupt Service Routine
No
Disable the interrupt by set $status.EXL=1
Restore Context
Exception Return (ERET) Enable the interrupt by set $status.EXL=0
Yes
Exception Occur.....

Figure 5.3 Nested interrupt Handling Scheme

Chapter 6 Verification Specification

6.1 Test cases

Test Case 1: Individual Trap

In Test Case 1, simulation on Individual Trap had conducted. In this project, the individual Trap cover the Sign-overflow, undefined instruction and Syscall. The expected output for this test case are as shown below:

i) Each individual Trap event occur
ii) Jump to the exception handler
iii) Jump to the respective ISR.
iv) Return to the user program.

Test Case 2: Multiple Trap

In Test Case 2, Multiple Trap event intentionally created by mixed the individual trap event in the same clock cycle. There are only two possible combination:

- Sign-overflow (EX stage) and Undefined instruction (ID stage)
- Sign-overflow (EX stage) and Syscall (ID stage)

The expected output for this test case is the Exception occur in the EX stage will be serve prior then the Exception occur in ID stage.

Test Case 3: Individual interrupt

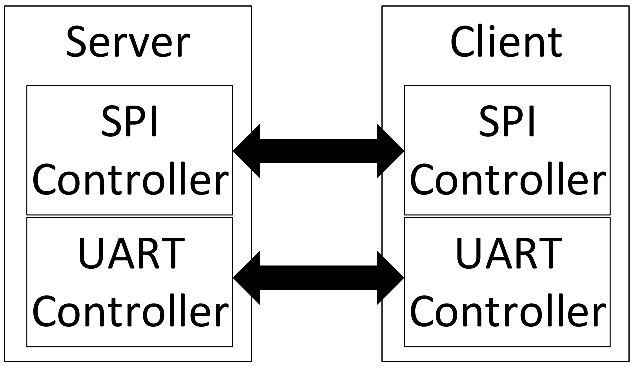


Figure 6.1 Connection between server and client

In Test Case 3, the connection between server and client had established as figure 6.1. In this project, the individual interrupt covers the UART interrupt and SPI interrupt. Client will keep sending the data to the Server through the SPI and UART to generate the interrupt to the Server side. In this test case, only one of the UART or SPI will turn on. The expected output for this test case are shows as below:

i) After the processor receive a data, it will raise the interrupt flag.
ii) Jump to the exception handler
iii) Jump to the respective ISR.
iv) Return to the user program.

Test Case 4: Multiple interrupt and Multiple Trap.

In Test Case 4, the connection was same like test case 3 but the UART and SPI on client will keep transmit the data at the same time in order to generate the interrupt to the Server side. In the same time, multiple trap occurrences had intentionally created at the server side to simulate the exception event clashing behaviour. The expected output is to ensure each exception event have been served and executed their ISR respectively.

6.2 MIPS assembly code

Test Case 1: Individual Trap and Test Case 2: Multiple Trap

LABEL	INSTRUCTION	COMMENTS
	.text 0x00400000	
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
Test Case 1: Individual Trap – Sign-Overflow		
sovf:	addi $s0, $zero, 1	#$s0 = 1
	sll $s0, $s0, 30	#$s0 = 1073741824, $s0[30] = 1, others=0
	addi $s1, $zero, 1	#$s1 = 1
	sll $s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0
	add $s2, $s0, $s1	#sign overflow, $s0[30]=$s1[0] && $s0[30]!=$s2[31]
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
Test Case 1: Individual Trap – undefined instruction		
u_inst:	sll $zero, $zero, 0	#undefined instruction
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	li $v0,1	#print integer
	li $a0,5	#print 5
Test Case 1: Individual Trap – Syscall		
	syscall	#syscall
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
	sll $zero, $zero, 0	#nop
Test Case 2: Multiple Trap - Sign-overflow (EX stage) and Undefined instruction (ID stage)		
sovf_uinst:	addi $s0, $zero, 1	#$s0 = 1
	sll $s0, $s0, 30	#$s0 = 1073741824, $s0[30] = 1, others=0
	addi $s1, $zero, 1	#$s1 = 1
	sll $s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0

	add	$s2, $s0, $s1	#sign overflow, $s0[30]=$s1[0] && $s0[30]!=$s2[31] at ex stage
	sll	$zero, $zero, 0	#undefined instruction
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
Test Case 2: Multiple Trap - •Sign-overflow (EX stage) and Syscall (ID stage)			
sovf_syscall:	addi	$s0, $zero, 1	
	sll	$s0, $s0, 30	#$s0 = 1073741824, $s0[30] = 1, others=0
	addi	$s1, $zero, 1	#$s1 = 1
	sll	$s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0
	add	$s2, $s0, $s1	#sign overflow, $s0[30]=$s1[0] && $s0[30]!=$s2[31] at ex stage
	syscall		#syscall
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
here:	j here		#forever loop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop

Test Case 3: Individual interrupt

- UART interrupt

Server

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	$s0=bfff_fe00
	addi	$s3, $zero, 1	$s3=1
	add	$s1, $zero, $zero	$s1=0
	addi	$t0, $zero, 0x1	Enable GPIO[16]
	sll	$t0, $t0, 16	
	sw	$t0, 20($s0)	
	addi	$t0, $zero, 0x0004	
	sll	$t0, $t0, 16	
	sw	$t0, 32($s0)	Enable UARTIE at PIC.PICMASK[2]
	addi	$t0, $zero, 0xC2	Configure UART.UARTCR=1100_0010,
	sb	$t0, 40($s0)	UARTEN=1, RXCIE=1,BAUD=010
GPIO:	xori	$s1, $s1, 1	Toggle GPIO[16]
	sll	$s2, $s1, 16	
	sw	$s2, 24($s0)	
start_timer:	ori	$t1, $zero, 0x0500	Create delay
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	
	mtc0	$zero, $9	
	mtc0	$zero, $13	
	mfc0	$t0, $9	
poll_timer:	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, GPIO	($s3=1)!=0 branch to GPIO
	nop		
	nop		
	nop		
	nop		

Client

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	$s0=bfff_fe00
	addi	$s3, $zero, 1	$s3=1
	add	$s1, $zero, $zero	$s1=0
	addi	$t0, $zero, 0x1	#GPIO setting #GPIOEN=1000_0000
	sll	$t0, $t0, 16	
	sw	$t0, 20($s0)	
	addi	$t0, $zero, 0xA2	#UARTCR=1010_0010, UARTEN=1,TXEIE=1
	sb	$t0, 40($s0)	
UART_restart:	addi	$t1, $zero, 0x11	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x22	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x33	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x44	
	sb	$t1, 42($s0)	Put data into UARTTDR
check_TXEF1:	lbu	$t1, 41($s0)	Check the transmit flag
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1,$zero, check_TXEF1	
	addi	$t1, $zero, 0x55	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x66	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x77	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0x88	
	sb	$t1, 42($s0)	Put data into UARTTDR
check_TXEF2:	lbu	$t1, 41($s0)	Check the transmit flag
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1,$zero, check_TXEF2	
	addi	$t1, $zero, 0x99	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0xAA	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0xBB	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0xCC	
	sb	$t1, 42($s0)	Put data into UARTTDR

check_TXEF 3:	lbu	$t1, 41($s0)	Check the transmit flag
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1,$zero, check_TXEF3	
	addi	$t1, $zero, 0xDD	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0xEE	
	sb	$t1, 42($s0)	Put data into UARTTDR
	addi	$t1, $zero, 0xFF	
	sb	$t1, 42($s0)	Put data into UARTTDR
check_TXEF 4:	lbu	$t1, 41($s0)	Check the transmit flag
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1,$zero, check_TXEF4	
	xori	$s1, $s1, 1	Toggle the GPIO
	sll	$s2, $s1, 16	
	sw	$s2, 24($s0)	
start_timer:	addi	$t1,$zero,0x500	Create delay
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	
	mtc0	$zero, $9	
	mtc0	$zero, $13	
poll_timer:	mfc0	$t0, $9	
	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, UART_restart	($s3=1)!=0 branch to GPIO
	nop		
	nop		
	nop		
	nop		

➢ SPI interrupt

Server

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	$s0=bfff_fe00
	addi	$s3, $zero, 1	$s3=1
	add	$s1, $zero, $zero	$s1=0
	addi	$t0, $zero, 0x1	Enable GPIO[16]
	sll	$t0, $t0, 16	
	sw	$t0, 20($s0)	
	addi	$t0, $zero, 0x0008	
	sll	$t0, $t0, 16	
	sw	$t0, 32($s0)	Enable SPIIE at PIC.PICMASK[3]
	addi	$t0, $zero, 0x87	#SPI setting = 10000111 = 0x87 -> SPE = 1,MSTR = 0, MODE 0,Baud=0111
	sb	$t0,36($s0)	#control reg
	addi	$t0, $zero, 0x0a	#clear SPISR=00001010, RXFHE=1 RXFIE=1
	sb	$t0,37($s0)	#status reg
GPIO:	xori	$s1, $s1, 1	Toggle GPIO[16]
	sll	$s2, $s1, 16	
	sw	$s2, 24($s0)	
start_timer:	ori	$t1, $zero, 0x0500	Create delay
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	
	mtc0	$zero, $9	
	mtc0	$zero, $13	
	mfc0	$t0, $9	
poll_timer:	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, GPIO	($s3=1)!=0 branch to GPIO
	nop		
	nop		
	nop		
	nop		

Client

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	$s0=bfff_fe00
	addi	$s3, $zero, 1	$s3=1
	addi	$s4, $zero, 0x40	Lower bound
	addi	$s5, $zero, 0x5B	upper bound
	add	$s1, $zero, $zero	$s1=0
	addi	$t0, $zero, 0x1	Enable GPIO[16]
	sll	$t0, $t0, 16	
	sb	$t0, 20($s0)	
	add	$s6, $zero, $s4	move lower bound to $s6
start_SPI:	addi	$s6, $s6, 0x01	$s6=$s6 +1
	bne	$s6, $s5, no_reset	if no reach upper bound, branch to no reset
	add	$s6, $zero, $s4	move lower bound to $s6
	addi	$s6, $s6, 0x01	$s6=$s6 +1
no_reset:	sb	$s6, 38($s0)	Store data to SPITDR
	addi	$t0, $zero, 0x01	#clear SPISR=00000001, TXEHE=1
	sb	$t0, 37($s0)	#status reg
	addi	$t0, $zero, 0xc7	#SPI setting = 11000111 = 0xC7 -> SPE = 1,MSTR = 1, MODE 0,Baud=0111
	sb	$t0, 36($s0)	#control reg
start_timer:	xori	$s1, $s1, 1	Toggle GPIO[16]
	sll	$s2, $s1, 16	
	sw	$s2, 24($s0)	
	ori	$t1, $zero, 0x1000	Create delay
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	
	mtc0	$zero, $9	
	mtc0	$zero, $13	
poll_timer:	mfc0	$t0, $9	
	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, start_SPI	($s3=1)!=0 branch to GPIO
	nop		
	nop		
	nop		
	nop		

Test Case 4: Multiple interrupt and Multiple Trap.

Server

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	
	addi	$s3, $zero, 1	
	add	$s4, $zero, $zero	
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 16	
	sw	$t0, 20($s0)	
	addi	$t0, $zero, 0x000c	#PIC MASK =00001100 SPIE=1 UARTIE=1
	sll	$t0, $t0, 16	
	sw	$t0, 32($s0)	
	addi	$t0, $zero, 0x87	#SPI setting = 10000111 = 0xC7 -> MSTR = 0, SPE = 1, MODE 0
	sb	$t0,36($s0)	#SPI control reg
	addi	$t0, $zero, 0x0a	#clear SPISR=00001010, RXFHE=1 RXFIE=1
	sb	$t0,37($s0)	#SPI status reg
	addi	$t0, $zero, 0xC2	
	sb	$t0, 40($s0)	#UART control reg=1100 0010, UARTEN=1, RXCIE=1, Baud mode=010
GPIO:	xori	$s4, $s4, 1	#pooling GPIO
	sll	$s5, $s4, 16	
	sw	$s5, 24($s0)	
Sign-overflow			
sovf:	addi	$s6, $zero, 1	#$s6 = 1
	sll	$s6, $s6, 30	#$s6 = 1073741824, $s6[30] = 1, others=0
	addi	$s1, $zero, 1	#$s1 = 1
	sll	$s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0
	add	$s2, $s6, $s1	#sign overflow, $s6[30]=$s1[0] && $s6[30]!=$s2[31]
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
Undefined instruction			
u_inst:	sll	$zero, $zero, 0	#undefined instruction set to 0xffff_ffff
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop

	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	li	$v0,1	#print integer
	li	$a0,5	#print 5
Syscall			
	syscall		#syscall
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
	sll	$zero, $zero, 0	#nop
Sign-overflow and undefined instruction			
sovf_uinst:	addi	$s6, $zero, 1	#$s6 = 1
	sll	$s6, $s6, 30	#$s6 = 1073741824, $s6[30] = 1, others=0
	addi	$s1, $zero, 1	#$s1 = 1
	sll	$s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0
	add	$s2, $s6, $s1	#sign overflow, $s6[30]=$s1[0] && $s6[30]!=$s2[31]
	sll	$zero, $zero, 0	#undefined instruction set to 0xffff_ffff
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
Sign-overflow and syscall			
sovf_syscall	addi	$s6, $zero, 1	#$s6 = 1
	sll	$s6, $s6, 30	#$s6 = 1073741824, $s6[30] = 1, others=0
	addi	$s1, $zero, 1	#$s1 = 1
	sll	$s1, $s1, 30	#$s1 = 1073741824, $s1[30] = 1, others=0
	add	$s2, $s6, $s1	#sign overflow, $s6[30]=$s1[0] && $s6[30]!=$s2[31]
	syscall		
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
	sll	$zero, $zero, 0	
start_timer:	ori	$t1, $zero, 0x0500	
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	

	mtc0	$zero, $9	
	mtc0	$zero, $13	
poll_timer:	mfc0	$t0, $9	
	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, GPIO	
	nop		
	nop		
	nop		
	nop		

Client

LABEL	INSTRUCTION		COMMENTS
	.text 0x00400000		
setting:	lui	$s0, 0xbfff	
	ori	$s0, $s0, 0xfe00	
	addi	$s3, $zero, 1	
	add	$s1, $zero, $zero	
	addi	$t0, $zero, 0x1	#GPIO setting
	sll	$t0, $t0, 16	
	sw	$t0, 20($s0)	#GPIOEN=1000_0000
	addi	$t0, $zero, 0xA2	
	sb	$t0, 40($s0)	#UARTCR=1010_0010, UARTEN=1,TXEIE=1
	addi	$s4, $zero, 0x40	#SPI lower bound
	addi	$s5, $zero, 0x5B	#SPI upper bound
	add	$s6, $zero, $s4	#Set the lower bound to s6
	addi	$t0, $zero, 0x01	
	sb	$t0, 37($s0)	#SPI Status register
	addi	$t0, $zero, 0xc7	
	sb	$t0, 36($s0)	#SPI Control Register
UART_SPI _restart:	addi	$t1, $zero, 0x11	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x22	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x33	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x44	
	sb	$t1, 42($s0)	
SPI_transm it:	sb	$s6, 38($s0)	
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	
	addi	$s6, $s6, 0x01	
check_TX EF1:	lbu	$t1, 41($s0)	
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1, $zero, check_TXEF1	
	addi	$t1, $zero, 0x55	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x66	

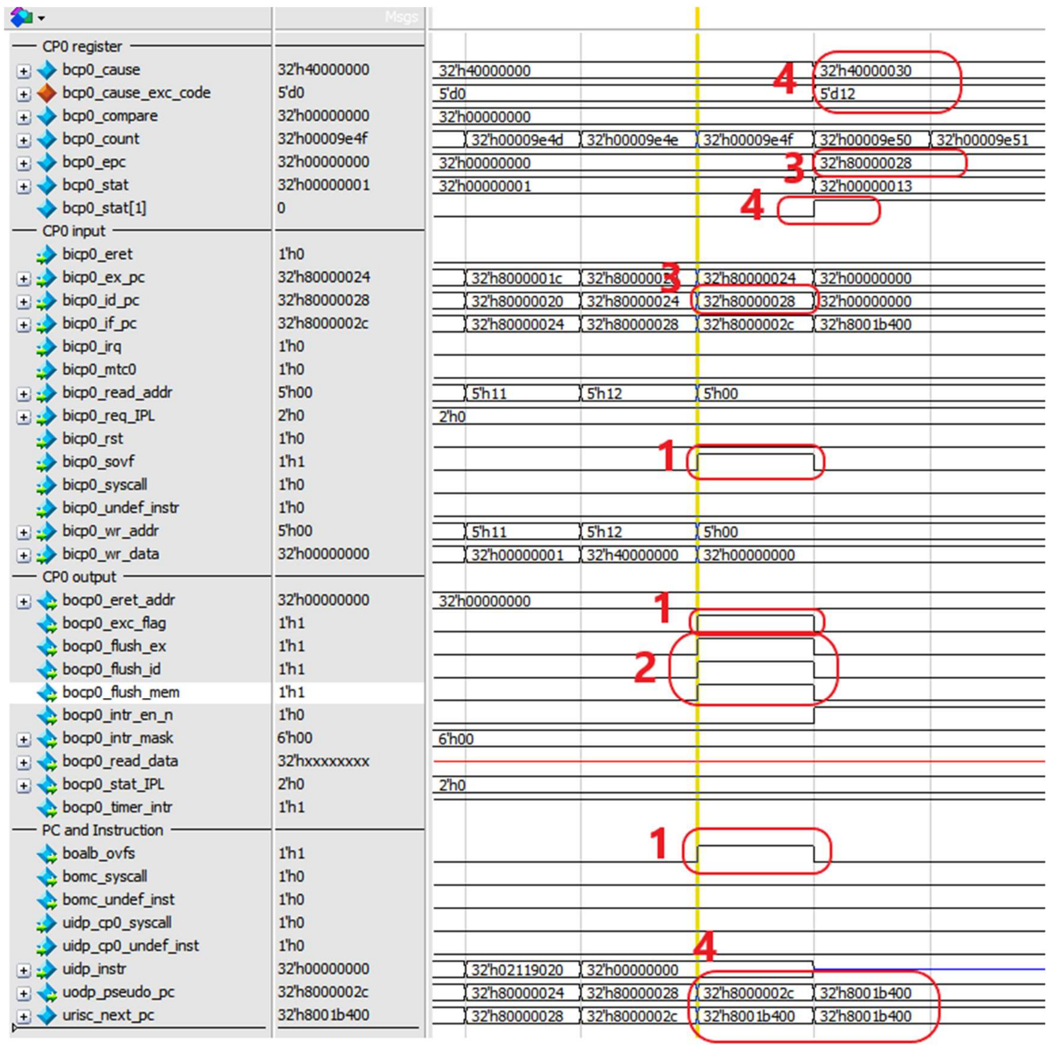
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x77	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0x88	
	sb	$t1, 42($s0)	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
check_TX EF2:	lbu	$t1, 41($s0)	
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1, $zero, check_TXEF2	
	addi	$t1, $zero, 0x99	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xAA	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xBB	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xCC	
	sb	$t1, 42($s0)	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
check_TX EF3:	lbu	$t1, 41($s0)	
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1, $zero, check_TXEF3	
	addi	$t1, $zero, 0xDD	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xEE	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xFF	
	sb	$t1, 42($s0)	
	addi	$t1, $zero, 0xCC	
	sb	$t1, 42($s0)	
	sb	$s6, 38($s0)	#send SPI data

	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
	sb	$s6, 38($s0)	#send SPI data
	addi	$s6, $s6, 0x01	
check_TXEF4:	lbu	$t1, 41($s0)	
	sll	$t1, $t1, 25	
	srl	$t1, $t1, 31	
	beq	$t1, $zero, check_TXEF4	
GPIO_toogle:	xori	$s1, $s1, 1	
	sll	$s2, $s1, 16	
	sw	$s2, 24($s0)	
start_timer:	addi	$t1,$zero,0x500	
	addi	$t0, $zero, 0x1	
	sll	$t0, $t0, 27	
	mtc0	$t0, $13	
	mtc0	$zero, $9	
	mtc0	$zero, $13	
poll_timer:	mfc0	$t0, $9	
	sub	$t0, $t1, $t0	
	bgtz	$t0, poll_timer	
	bne	$s3, $zero, UART_SPI_restart	
	nop		
	nop		
	nop		
	nop		

6.3 Simulation Result

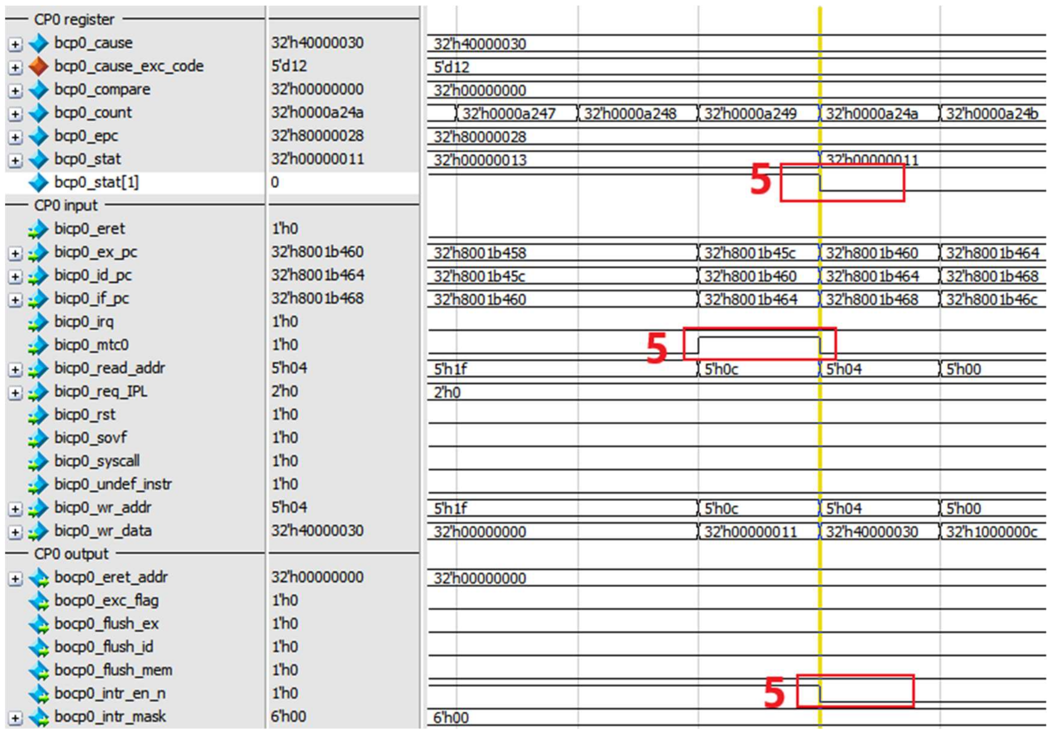
Test Case 1: Individual Trap

Sign-overflow



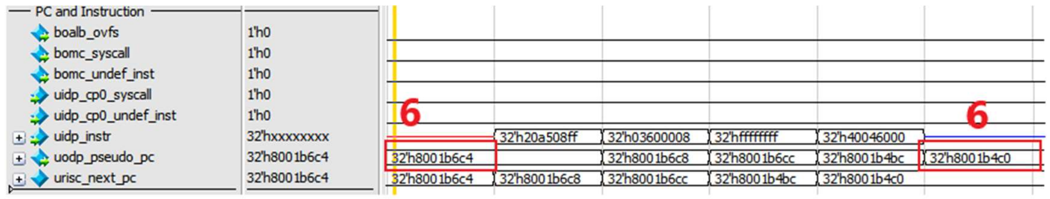
```
CP0 register
bcp0_cause              32'h40000000
bcp0_cause_exc_code     5'd0
bcp0_compare            32'h00000000
bcp0_count              32'h00009e4f
bcp0_epc                32'h00000000
bcp0_stat               32'h00000001
bcp0_stat[1]            0
CP0 input
bicp0_eret              1'h0
bicp0_ex_pc             32'h80000024
bicp0_id_pc             32'h80000028
bicp0_if_pc             32'h8000002c
bicp0_irq               1'h0
bicp0_mtc0              1'h0
bicp0_read_addr         5'h00
bicp0_req_IPL           2'h0
bicp0_rst               1'h0
bicp0_sovf              1'h1
bicp0_syscall           1'h0
bicp0_undef_instr       1'h0
bicp0_wr_addr           5'h00
bicp0_wr_data           32'h00000000
CP0 output
bocp0_eret_addr         32'h00000000
bocp0_exc_flag          1'h1
bocp0_flush_ex          1'h1
bocp0_flush_id          1'h1
bocp0_flush_mem         1'h1
bocp0_intr_en_n         1'h0
bocp0_intr_mask         6'h00
bocp0_read_data         32'hxxxxxxxx
bocp0_stat_IPL          2'h0
bocp0_timer_intr        1'h1
PC and Instruction
boalb_ovfs              1'h1
bomc_syscall            1'h0
bomc_undef_inst         1'h0
uidp_cp0_syscall        1'h0
uidp_cp0_undef_inst     1'h0
uidp_instr              32'h00000000
uodp_pseudo_pc          32'h8000002c
urisc_next_pc           32'h8001b400
```

1) When detected the sign-overflow at ALU block (EX stage), hardware will raise the bocp0_exc_flag.
2) Flush the IF/ID, ID/EXE, EXE/MEM pipeline.
3) Put the ID stage's PC to the $epc for return purpose after handler the exception.
4) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, sign-overflow the exception code is 12, which will write into the CP0 $cause by hardware.



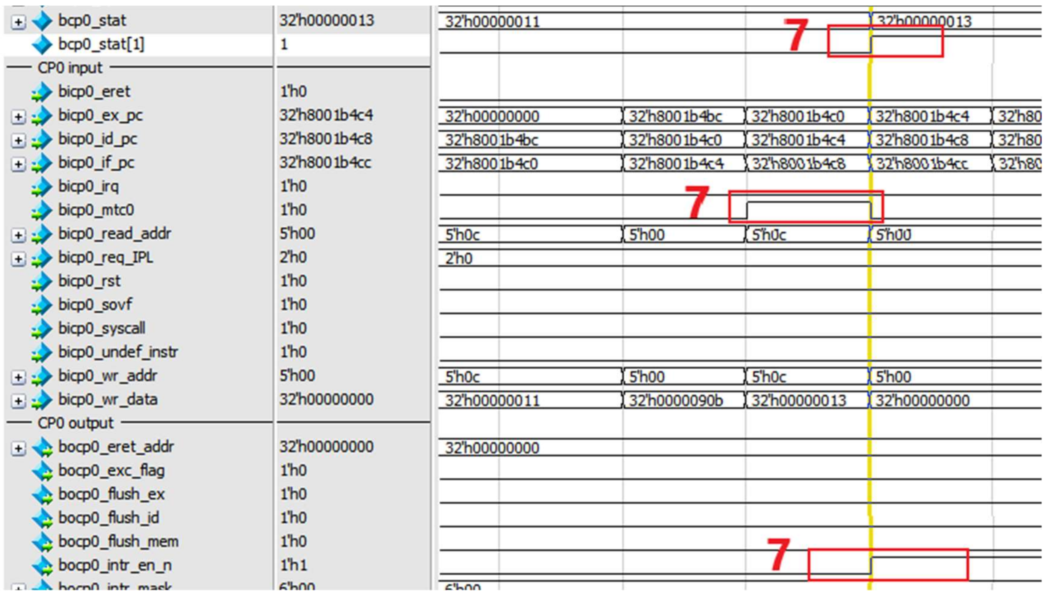
```
CP0 register
bcp0_cause            32'h40000030
bcp0_cause_exc_code   5'd12
bcp0_compare          32'h00000000
bcp0_count            32'h0000a24a
bcp0_epc              32'h80000028
bcp0_stat             32'h00000011
bcp0_stat[1]          0
CP0 input
bicp0_eret            1'h0
bicp0_ex_pc           32'h8001b460
bicp0_id_pc           32'h8001b464
bicp0_if_pc           32'h8001b468
bicp0_irq             1'h0
bicp0_mtc0            1'h0
bicp0_read_addr       5'h04
bicp0_req_IPL         2'h0
bicp0_rst             1'h0
bicp0_sovf            1'h0
bicp0_syscall         1'h0
bicp0_undef_instr     1'h0
bicp0_wr_addr         5'h04
bicp0_wr_data         32'h40000030
CP0 output
bocp0_eret_addr       32'h00000000
bocp0_exc_flag        1'h0
bocp0_flush_ex        1'h0
bocp0_flush_id        1'h0
bocp0_flush_mem       1'h0
bocp0_intr_en_n       1'h0
bocp0_intr_mask       6'h00
```

5) After push the register information into Stack, $status [1] will pull down by user, to enable further interrupt.



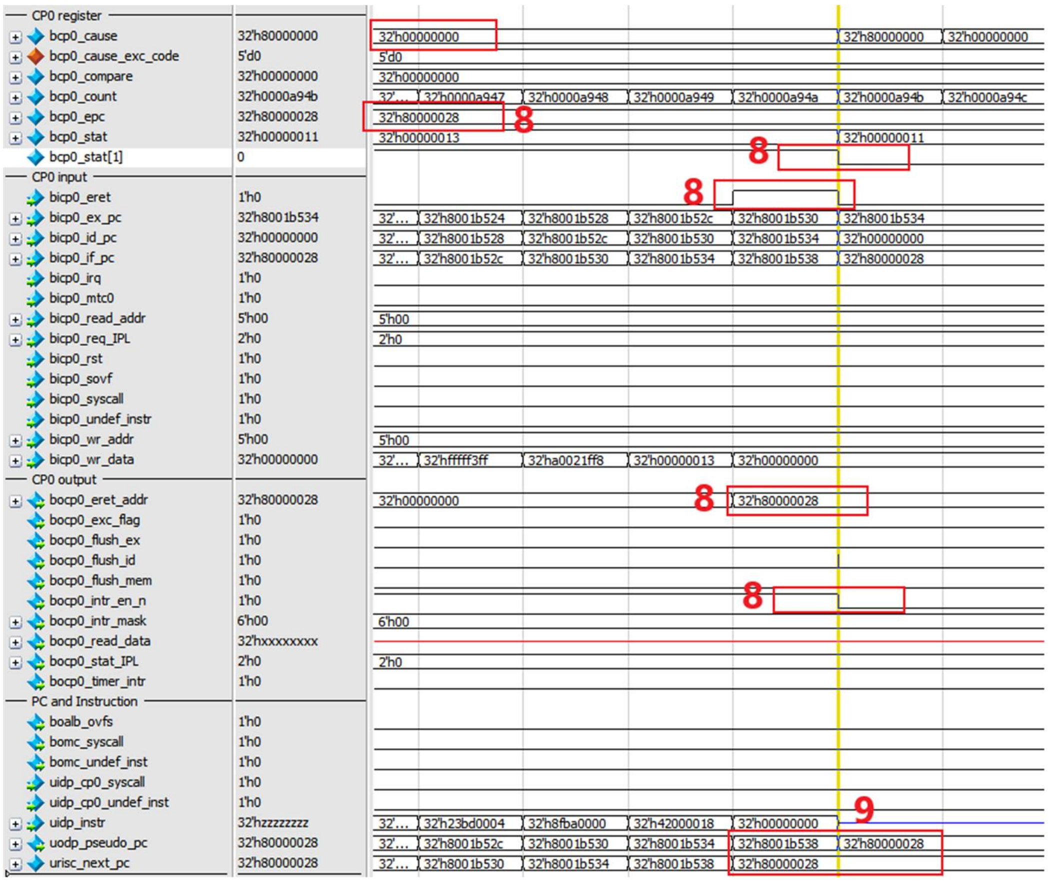
```
PC and Instruction
boalb_ovfs        1'h0
bomc_syscall      1'h0
bomc_undef_inst   1'h0
uidp_cp0_syscall  1'h0
uidp_cp0_undef_inst 1'h0
uidp_instr        32'hxxxxxxxx
uodp_pseudo_pc    32'h8001b6c4
urisc_next_pc     32'h8001b6c4
```

6) The exception code in $cause will decode and branch to the respective ISR. For, sign-overflow the ISR start at 0x80016c4. After ISR, it will branch to pop data section (0x8001b4c0).



bcp0_stat	32'h00000013
bcp0_stat[1]	1
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8001b4c4
bicp0_id_pc	32'h8001b4c8
bicp0_if_pc	32'h8001b4cc
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h00
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h0
bicp0_wr_addr	5'h00
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h00000000
bocp0_exc_flag	1'h0
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h0
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h1
bocp0_intr_mask	6'h00

7) Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.

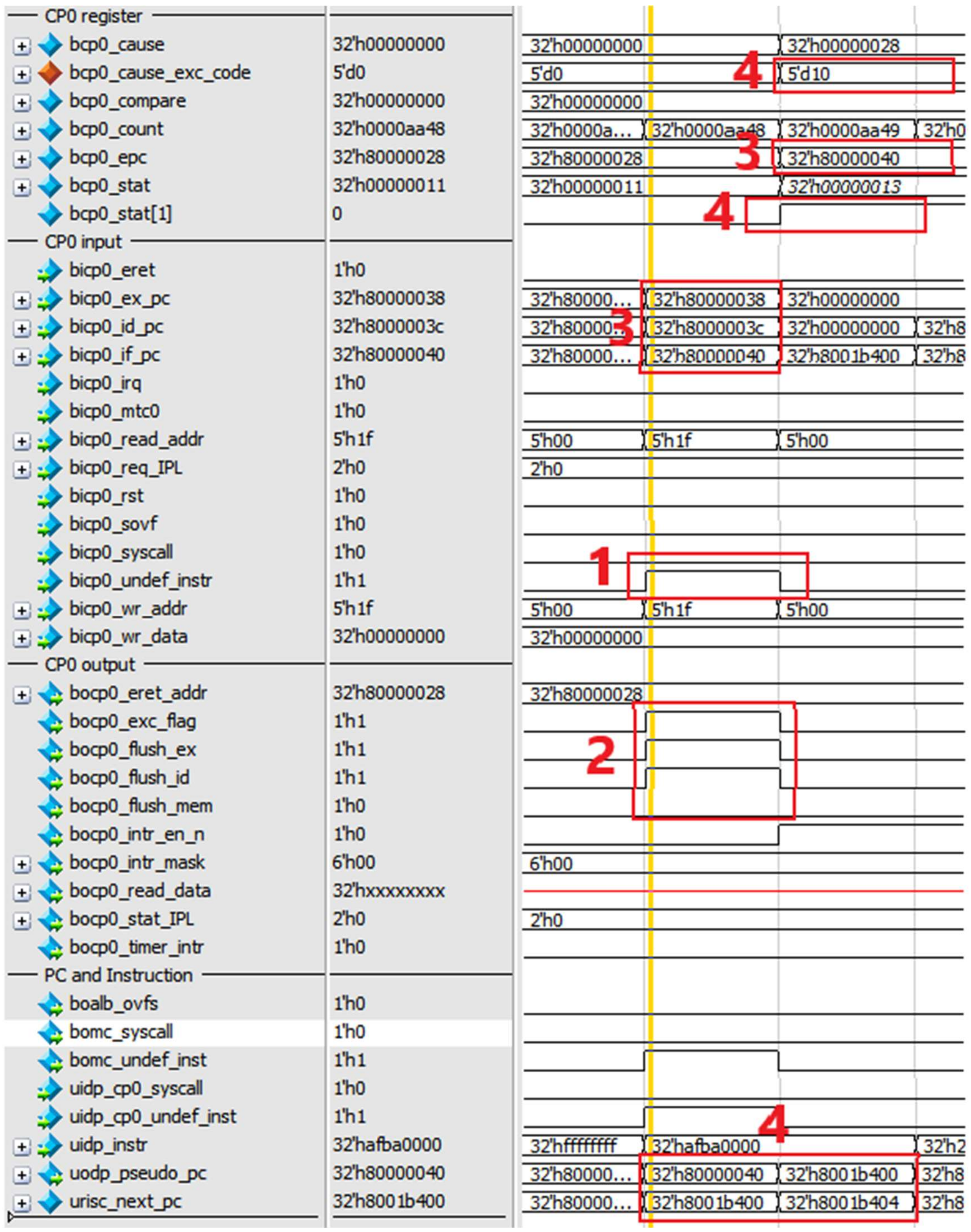


CP0 register	
bcp0_cause	32'h80000000
bcp0_cause_exc_code	5'd0
bcp0_compare	32'h00000000
bcp0_count	32'h0000a94b
bcp0_epc	32'h80000028
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8001b534
bicp0_id_pc	32'h00000000
bicp0_if_pc	32'h80000028
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h00
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h0
bicp0_wr_addr	5'h00
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h80000028
bocp0_exc_flag	1'h0
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h0
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h0
bomc_undef_inst	1'h0
uidp_cp0_syscall	1'h0
uidp_cp0_undef_inst	1'h0
uidp_instr	32'hzzzzzzzz
uodp_pseudo_pc	32'h80000028
urisc_next_pc	32'h80000028

8) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.

9) Jump back to the User program in the next clock cycle.

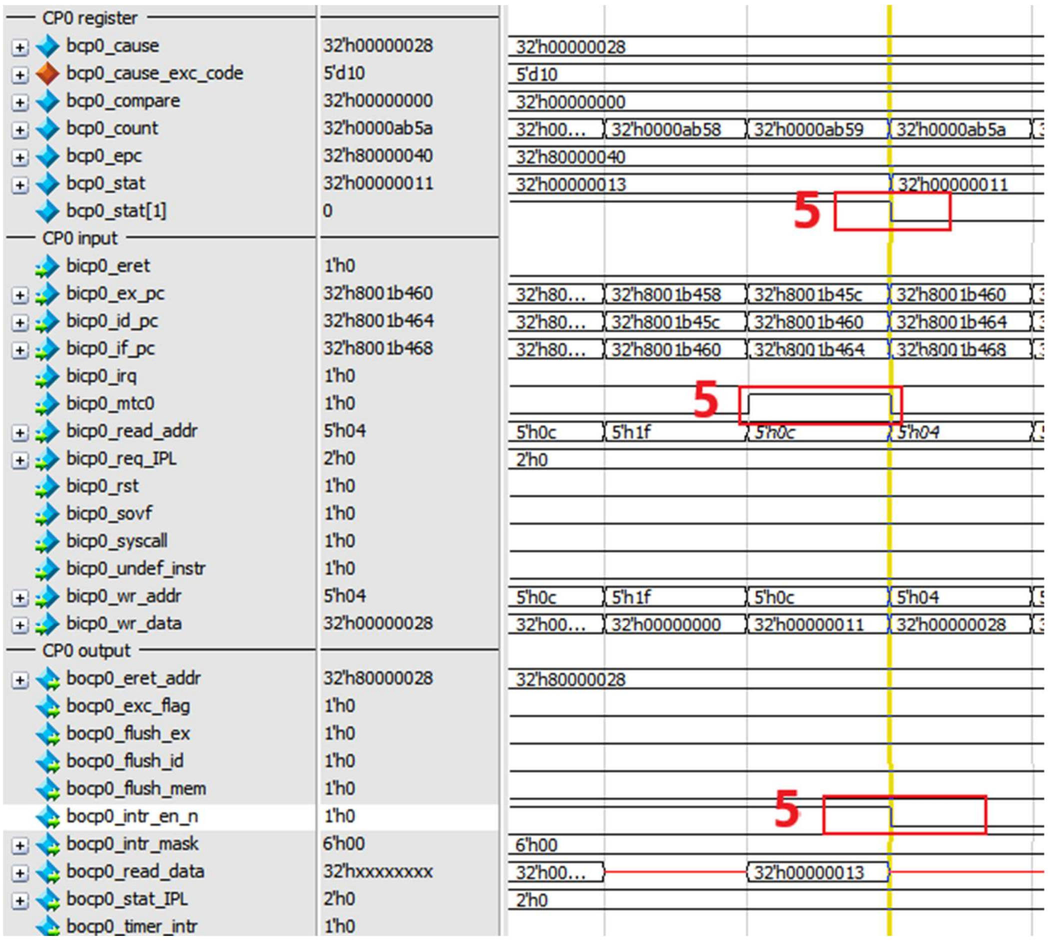
Undefined instruction



CP0 register	
bcp0_cause	32'h00000000
bcp0_cause_exc_code	5'd0
bcp0_compare	32'h00000000
bcp0_count	32'h0000aa48
bcp0_epc	32'h80000028
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h80000038
bicp0_id_pc	32'h8000003c
bicp0_if_pc	32'h80000040
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h1f
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h1
bicp0_wr_addr	5'h1f
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h80000028
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h1
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h0
bomc_undef_inst	1'h1
uidp_cp0_syscall	1'h0
uidp_cp0_undef_inst	1'h1
uidp_instr	32'hafba0000
uodp_pseudo_pc	32'h80000040
urisc_next_pc	32'h8001b400

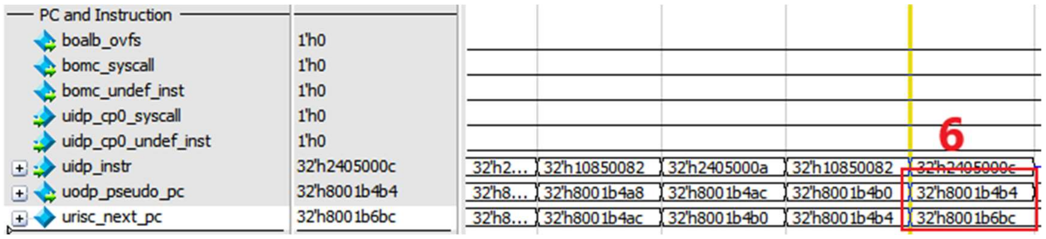
1) When detected the undefined instruction at Main control unit (ID stage), hardware will raise the bocp0_exc_flag.
2) Flush the IF/ID, ID/EXE pipeline.
3) Put the IF stage's PC to the $epc for return purpose after handler the exception.
4) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, undefined

instruction the exception code is 10, which will write into the CP0 $cause by hardware.



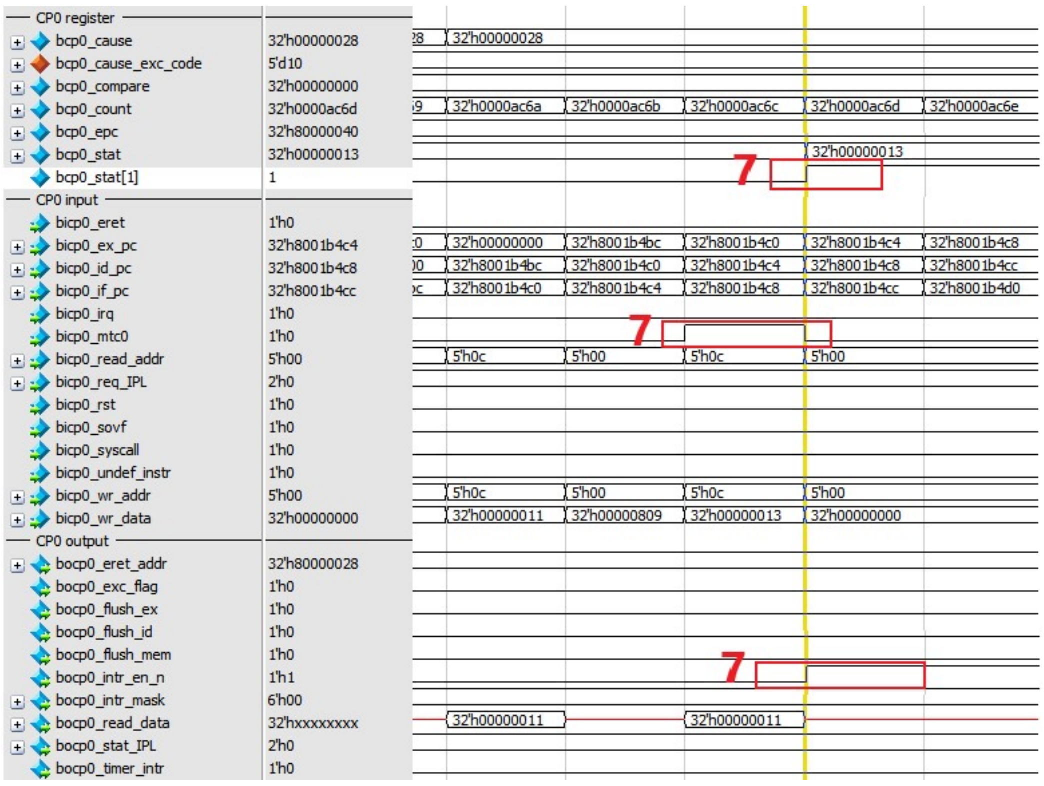
CP0 register	
bcp0_cause	32'h00000028
bcp0_cause_exc_code	5'd10
bcp0_compare	32'h00000000
bcp0_count	32'h0000ab5a
bcp0_epc	32'h80000040
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8001b460
bicp0_id_pc	32'h8001b464
bicp0_if_pc	32'h8001b468
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h04
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h0
bicp0_wr_addr	5'h04
bicp0_wr_data	32'h00000028
CP0 output	
bocp0_eret_addr	32'h80000028
bocp0_exc_flag	1'h0
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h0
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0

5) After push the register information into Stack, $status [1] will pull down by user, to enable further interrupt.



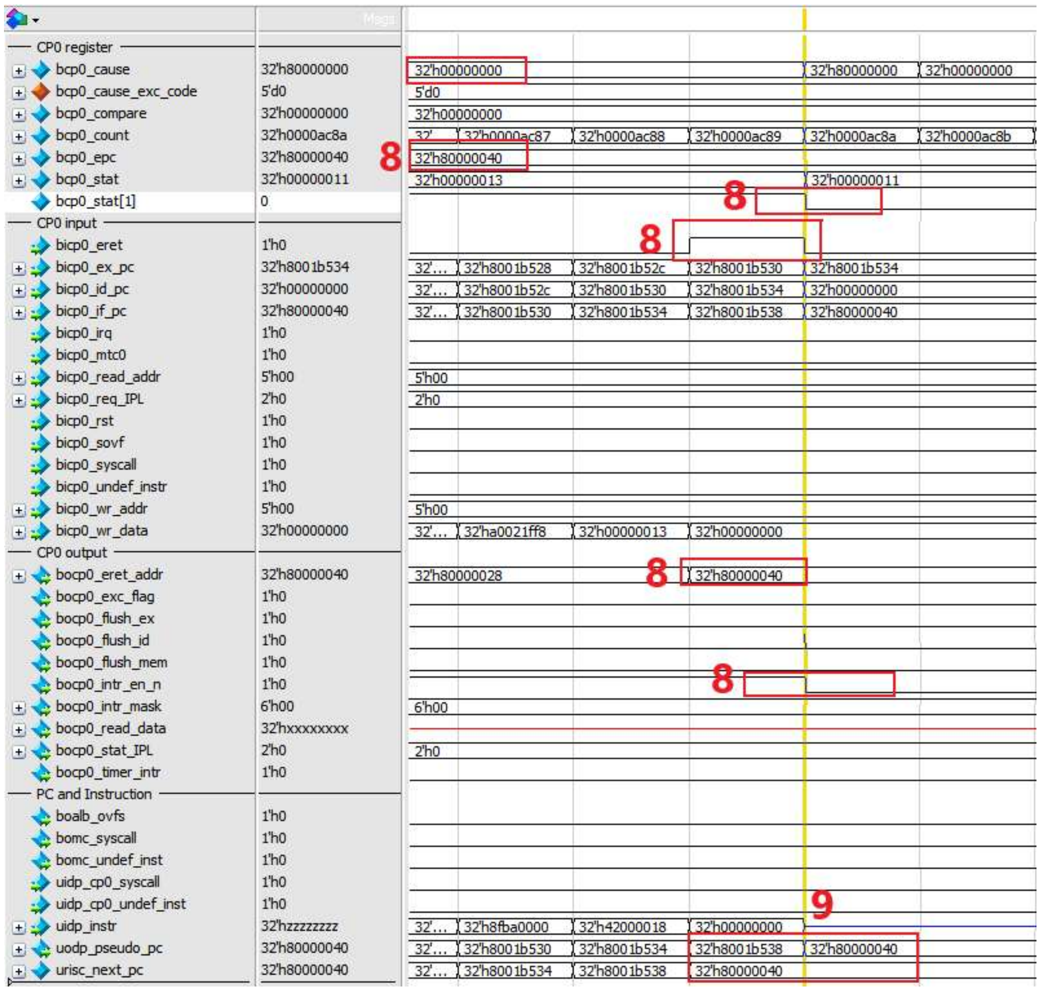
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h0
bomc_undef_inst	1'h0
uidp_cp0_syscall	1'h0
uidp_cp0_undef_inst	1'h0
uidp_instr	32'h2405000c
uodp_pseudo_pc	32'h8001b4b4
urisc_next_pc	32'h8001b6bc

6) The exception code in $cause will decode and branch to the respective ISR. For undefined instruction, the ISR start at 0x8001b6bc. After ISR, it will branch to pop data section (0x8001b4c0).



```
CP0 register
bcp0_cause             32'h00000028
bcp0_cause_exc_code    5'd10
bcp0_compare           32'h00000000
bcp0_count             32'h0000ac6d
bcp0_epc               32'h80000040
bcp0_stat              32'h00000013
bcp0_stat[1]           1
CP0 input
bicp0_eret             1'h0
bicp0_ex_pc            32'h8001b4c4
bicp0_id_pc            32'h8001b4c8
bicp0_if_pc            32'h8001b4cc
bicp0_irq              1'h0
bicp0_mtc0             1'h0
bicp0_read_addr        5'h00
bicp0_req_IPL          2'h0
bicp0_rst              1'h0
bicp0_sovf             1'h0
bicp0_syscall          1'h0
bicp0_undef_instr      1'h0
bicp0_wr_addr          5'h00
bicp0_wr_data          32'h00000000
CP0 output
bocp0_eret_addr        32'h80000028
bocp0_exc_flag         1'h0
bocp0_flush_ex         1'h0
bocp0_flush_id         1'h0
bocp0_flush_mem        1'h0
bocp0_intr_en_n        1'h1
bocp0_intr_mask        6'h00
bocp0_read_data        32'hxxxxxxxx
bocp0_stat_IPL         2'h0
bocp0_timer_intr       1'h0
```

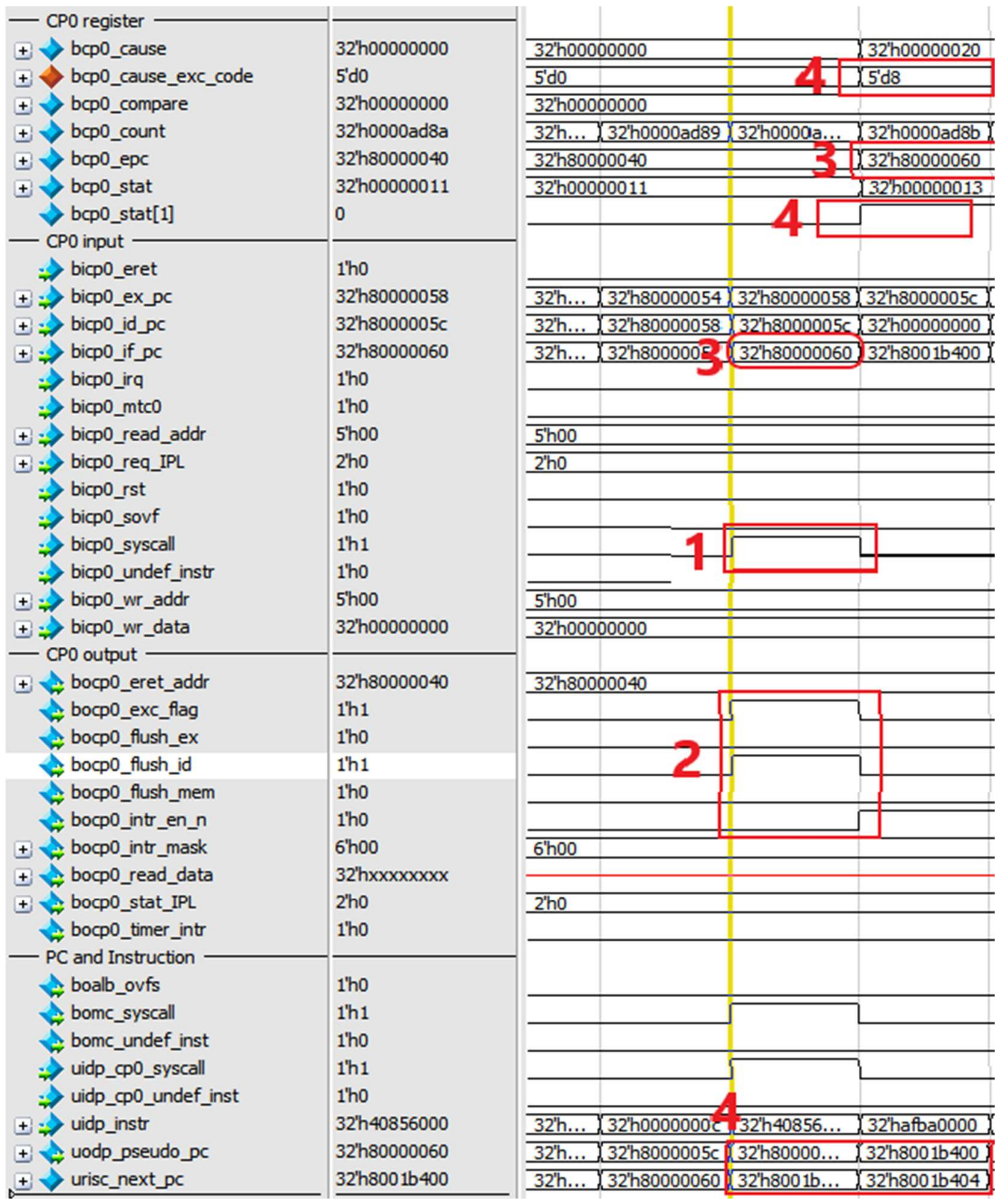
7) Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.



```
CP0 register
bcp0_cause              32'h80000000
bcp0_cause_exc_code     5'd0
bcp0_compare            32'h00000000
bcp0_count              32'h0000ac8a
bcp0_epc                32'h80000040
bcp0_stat               32'h00000011
bcp0_stat[1]            0
CP0 input
bicp0_eret              1'h0
bicp0_ex_pc             32'h8001b534
bicp0_id_pc             32'h00000000
bicp0_if_pc             32'h80000040
bicp0_irq               1'h0
bicp0_mtc0              1'h0
bicp0_read_addr         5'h00
bicp0_req_IPL           2'h0
bicp0_rst               1'h0
bicp0_sovf              1'h0
bicp0_syscall           1'h0
bicp0_undef_instr       1'h0
bicp0_wr_addr           5'h00
bicp0_wr_data           32'h00000000
CP0 output
bocp0_eret_addr         32'h80000040
bocp0_exc_flag          1'h0
bocp0_flush_ex          1'h0
bocp0_flush_id          1'h0
bocp0_flush_mem         1'h0
bocp0_intr_en_n         1'h0
bocp0_intr_mask         6'h00
bocp0_read_data         32'hxxxxxxxx
bocp0_stat_IPL          2'h0
bocp0_timer_intr        1'h0
PC and Instruction
boalb_ovfs              1'h0
bomc_syscall            1'h0
bomc_undef_inst         1'h0
uidp_cp0_syscall        1'h0
uidp_cp0_undef_inst     1'h0
uidp_instr              32'hzzzzzzzz
uodp_pseudo_pc          32'h80000040
urisc_next_pc           32'h80000040
```

8) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.
9) Jump back to the User program in the next clock cycle.

<u>Syscall</u>



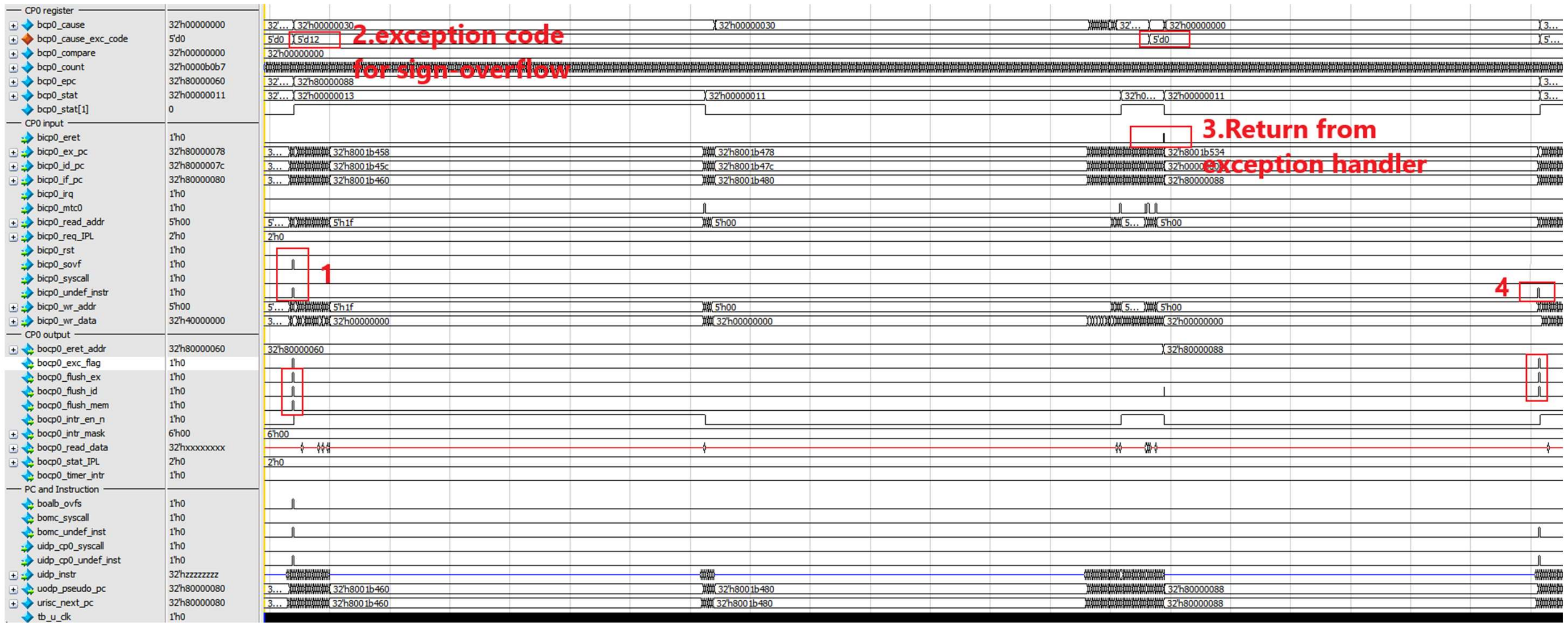
CP0 register	
bcp0_cause	32'h00000000
bcp0_cause_exc_code	5'd0
bcp0_compare	32'h00000000
bcp0_count	32'h0000ad8a
bcp0_epc	32'h80000040
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h80000058
bicp0_id_pc	32'h8000005c
bicp0_if_pc	32'h80000060
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h00
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h1
bicp0_undef_instr	1'h0
bicp0_wr_addr	5'h00
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h80000040
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h1
bomc_undef_inst	1'h0
uidp_cp0_syscall	1'h1
uidp_cp0_undef_inst	1'h0
uidp_instr	32'h40856000
uodp_pseudo_pc	32'h80000060
urisc_next_pc	32'h8001b400

32'h00000000
32'h00000020
5'd0
4
5'd8
32'h00000000
32'h... 32'h0000ad89 32'h0000a... 32'h0000ad8b
32'h80000040
3
32'h80000060
32'h00000011
32'h00000013
4
32'h... 32'h80000054 32'h80000058 32'h8000005c
32'h... 32'h80000058 32'h8000005c 32'h00000000
32'h... 32'h8000005 3 32'h80000060 32'h8001b400
5'h00
2'h0
1
5'h00
32'h00000000
32'h80000040
2
6'h00
2'h0
32'h... 32'h0000000c 4 32'h40856... 32'hafba0000
32'h... 32'h8000005c 32'h80000... 32'h8001b400
32'h... 32'h80000060 32'h8001b... 32'h8001b404

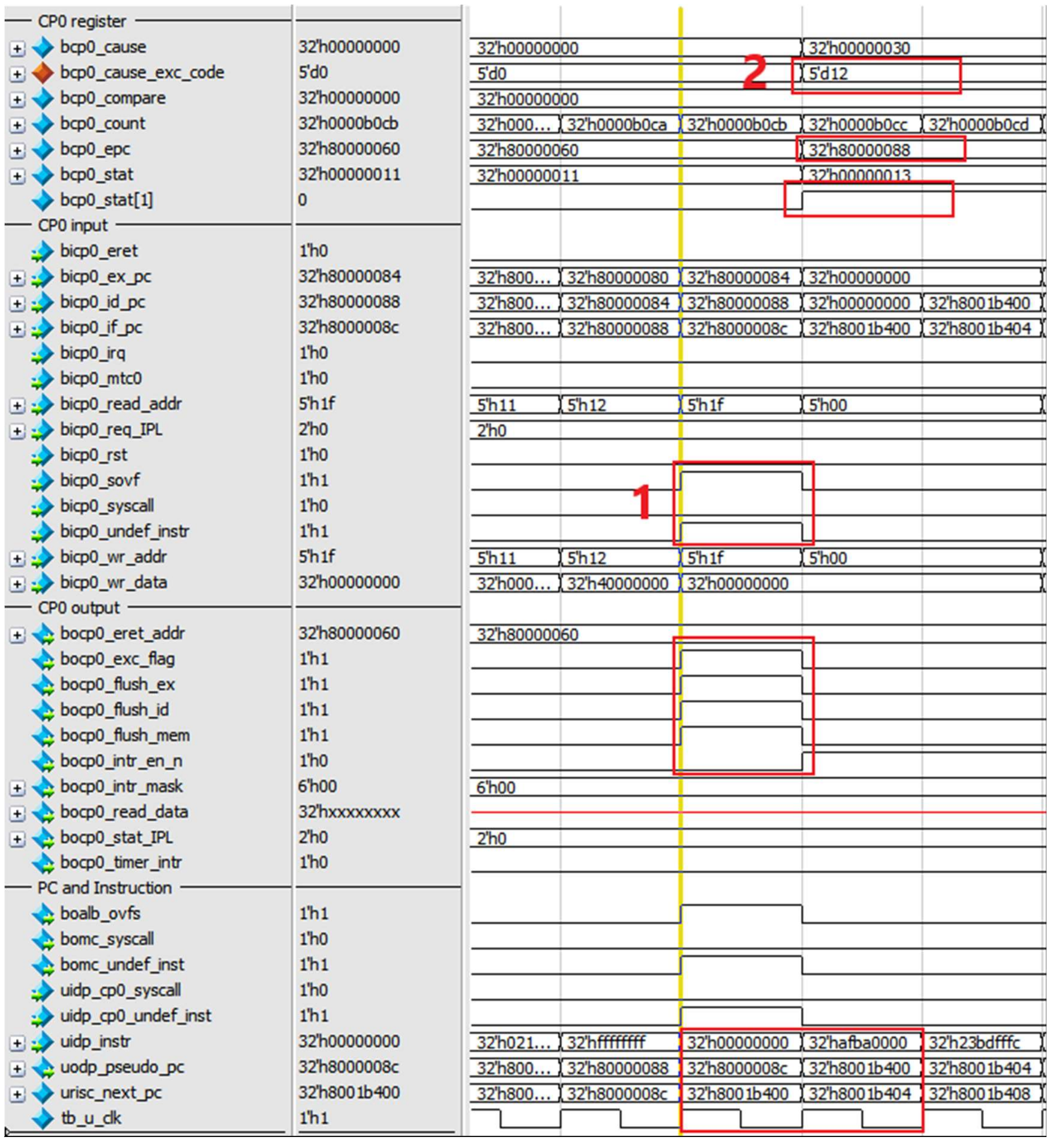
1) When detected the Syscall at Main control unit (ID stage), hardware will raise the bocp0_exc_flag.
2) Flush the IF/ID pipeline.
3) Put the IF stage's PC to the $epc for return purpose after handler the exception.
4) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, Syscall the exception code is 8, which will write into the CP0 $cause by hardware.
5) Step 5 –Step 9 was similar to the undefined instruction. The only different is the address of the ISR.

Test Case 2: Multiple Trap

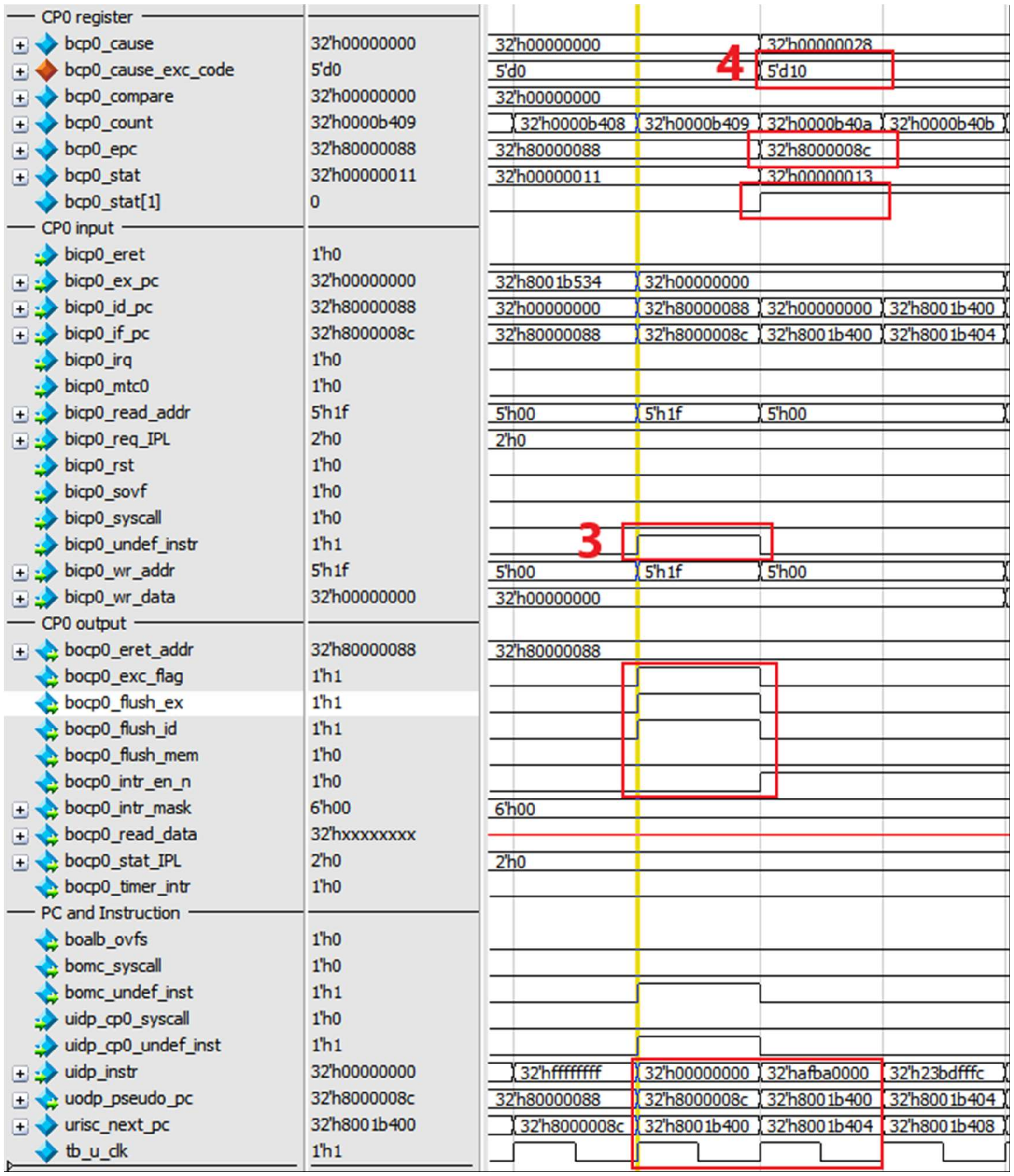
Sign-overflow (EX stage) and Undefined instruction (ID stage)



When sign-overflow (Ex stage) and undefined instruction (ID stage) occur in the same clock cycle (label "1"). Sign-overflow will be handler prior (label "2"). After return from the Sign-Overflow (label "3"), it will handler the undefined instruction in the next clock cycle (label "4").



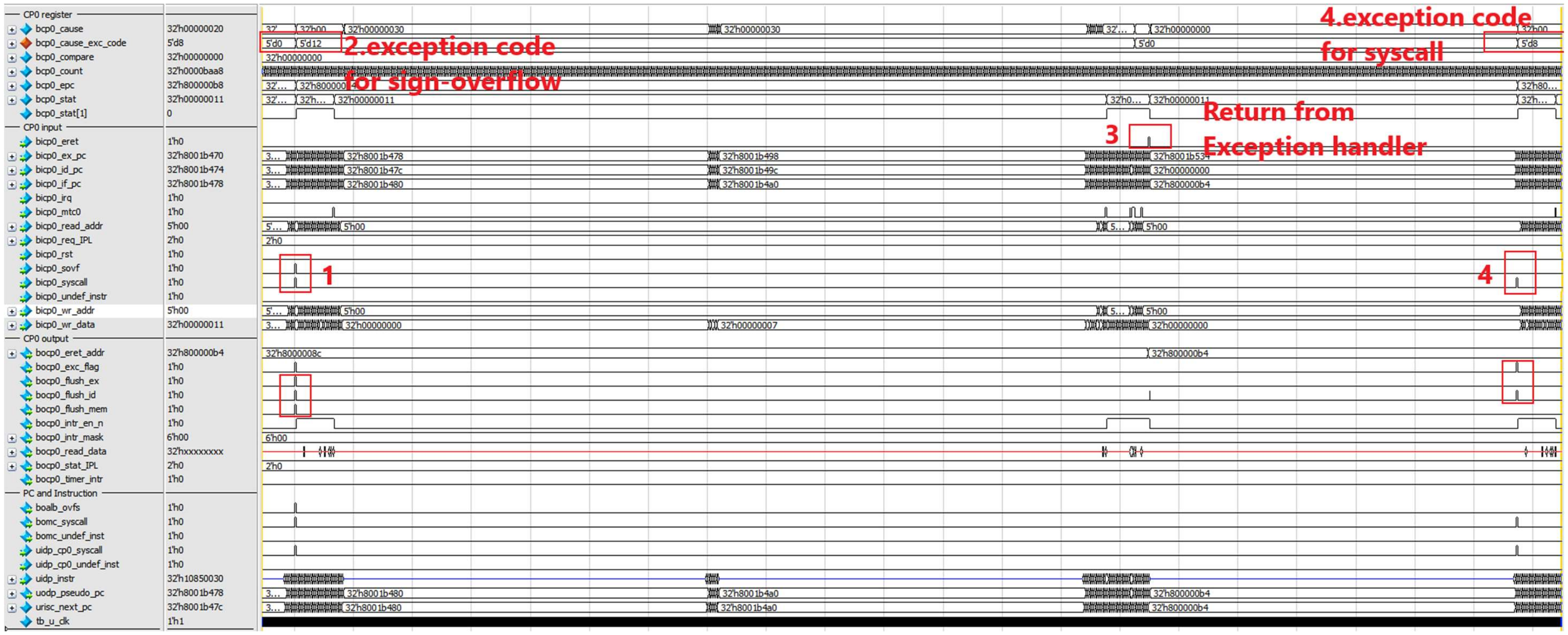
We can clearly see that when both exception occur in the same clock cycle (label "1"), sign-overflow will be handle because the exception Code is 12(label "2"), which is sign-overflow exception.



CP0 register	
bcp0_cause	32'h00000000
bcp0_cause_exc_code	5'd0
bcp0_compare	32'h00000000
bcp0_count	32'h0000b409
bcp0_epc	32'h80000088
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h00000000
bicp0_id_pc	32'h80000088
bicp0_if_pc	32'h8000008c
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h1f
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h1
bicp0_wr_addr	5'h1f
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h80000088
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h1
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h0
bomc_undef_inst	1'h1
uidp_cp0_syscall	1'h0
uidp_cp0_undef_inst	1'h1
uidp_instr	32'h00000000
uodp_pseudo_pc	32'h8000008c
urisc_next_pc	32'h8001b400
tb_u_clk	1'h1

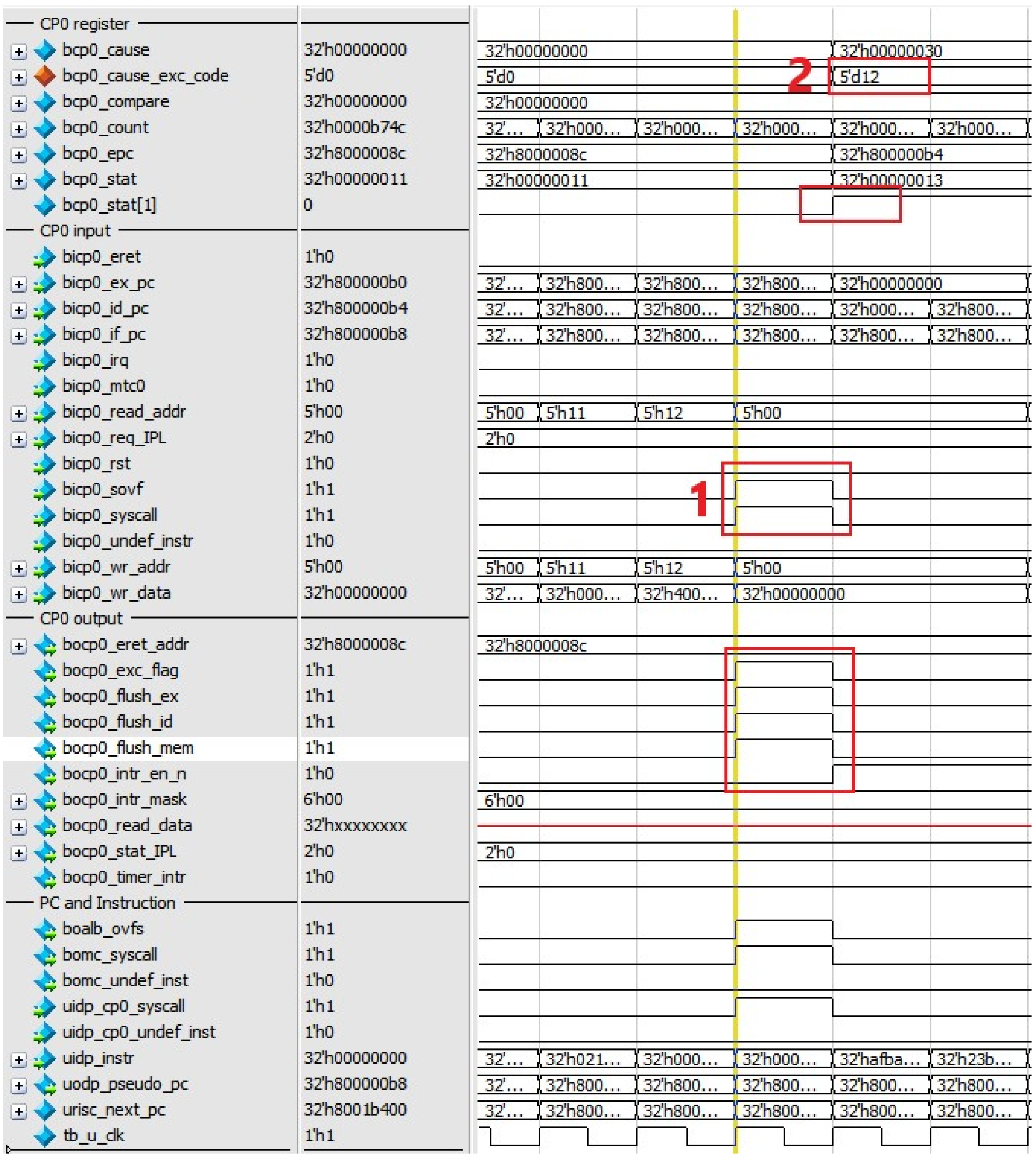
After return from the sign-overflow exception, the undefined instruction will be handle (label "3"). We can clearly see that the Exception Code is 10, which is undefined instruction (label "4").

<u>Sign-overflow (EX stage) and Syscall (ID stage)</u>


2.exception code for sign-overflow
4.exception code for syscall
3
Return from Exception handler
1
4

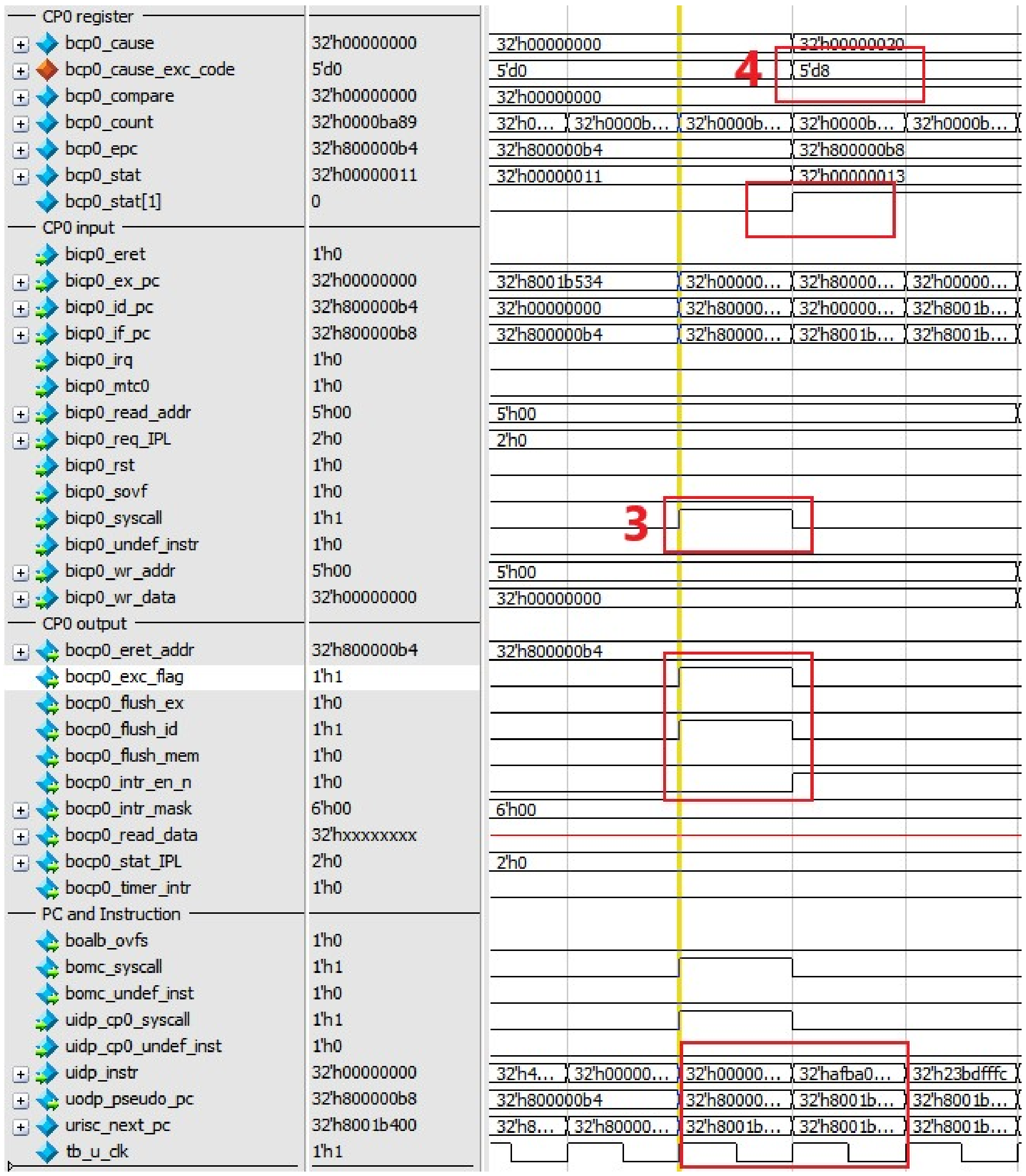
```
CP0 register
bcp0_cause              32'h00000020
bcp0_cause_exc_code     5'd8
bcp0_compare            32'h00000000
bcp0_count              32'h0000baa8
bcp0_epc                32'h800000b8
bcp0_stat               32'h00000011
bcp0_stat[1]            0
CP0 input
bicp0_eret              1'h0
bicp0_ex_pc             32'h8001b470
bicp0_id_pc             32'h8001b474
bicp0_if_pc             32'h8001b478
bicp0_irq               1'h0
bicp0_mtc0              1'h0
bicp0_read_addr         5'h00
bicp0_req_IPL           2'h0
bicp0_rst               1'h0
bicp0_sovf              1'h0
bicp0_syscall           1'h0
bicp0_undef_instr       1'h0
bicp0_wr_addr           5'h00
bicp0_wr_data           32'h00000011
CP0 output
bocp0_eret_addr         32'h800000b4
bocp0_exc_flag          1'h0
bocp0_flush_ex          1'h0
bocp0_flush_id          1'h0
bocp0_flush_mem         1'h0
bocp0_intr_en_n         1'h0
bocp0_intr_mask         6'h00
bocp0_read_data         32'hxxxxxxxx
bocp0_stat_IPL          2'h0
bocp0_timer_intr        1'h0
PC and Instruction
boalb_ovfs              1'h0
bomc_syscall            1'h0
bomc_undef_inst         1'h0
uidp_cp0_syscall        1'h0
uidp_cp0_undef_inst     1'h0
uidp_instr              32'h10850030
uodp_pseudo_pc          32'h8001b478
urisc_next_pc           32'h8001b47c
tb_u_clk                1'h1
```

When sign-overflow (Ex stage) and Syscall (ID stage) occur in the same clock cycle (label "1"). Sign-overflow will be handler prior (label "2"). After return from the Sign-Overflow (label "3"), it will handler the Syscall in the next clock cycle (label "4").



```
CP0 register
bcp0_cause              32'h00000000
bcp0_cause_exc_code     5'd0
bcp0_compare            32'h00000000
bcp0_count              32'h0000b74c
bcp0_epc                32'h8000008c
bcp0_stat               32'h00000011
bcp0_stat[1]            0
CP0 input
bicp0_eret              1'h0
bicp0_ex_pc             32'h800000b0
bicp0_id_pc             32'h800000b4
bicp0_if_pc             32'h800000b8
bicp0_irq               1'h0
bicp0_mtc0              1'h0
bicp0_read_addr         5'h00
bicp0_req_IPL           2'h0
bicp0_rst               1'h0
bicp0_sovf              1'h1
bicp0_syscall           1'h1
bicp0_undef_instr       1'h0
bicp0_wr_addr           5'h00
bicp0_wr_data           32'h00000000
CP0 output
bocp0_eret_addr         32'h8000008c
bocp0_exc_flag          1'h1
bocp0_flush_ex          1'h1
bocp0_flush_id          1'h1
bocp0_flush_mem         1'h1
bocp0_intr_en_n         1'h0
bocp0_intr_mask         6'h00
bocp0_read_data         32'hxxxxxxxx
bocp0_stat_IPL          2'h0
bocp0_timer_intr        1'h0
PC and Instruction
boalb_ovfs              1'h1
bomc_syscall            1'h1
bomc_undef_inst         1'h0
uidp_cp0_syscall        1'h1
uidp_cp0_undef_inst     1'h0
uidp_instr              32'h00000000
uodp_pseudo_pc          32'h800000b8
urisc_next_pc           32'h8001b400
tb_u_clk                1'h1
```

We can clearly see that when both exception occur in the same clock cycle (label "1"), sign-overflow will be handle because the exception Code is 12(label "2"), which is sign-overflow exception.

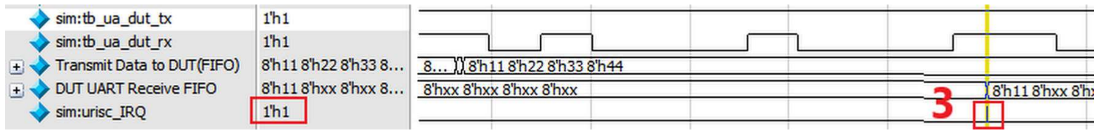


CP0 register	
bcp0_cause	32'h00000000
bcp0_cause_exc_code	5'd0
bcp0_compare	32'h00000000
bcp0_count	32'h0000ba89
bcp0_epc	32'h800000b4
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h00000000
bicp0_id_pc	32'h800000b4
bicp0_if_pc	32'h800000b8
bicp0_irq	1'h0
bicp0_mtc0	1'h0
bicp0_read_addr	5'h00
bicp0_req_IPL	2'h0
bicp0_rst	1'h0
bicp0_sovf	1'h0
bicp0_syscall	1'h1
bicp0_undef_instr	1'h0
bicp0_wr_addr	5'h00
bicp0_wr_data	32'h00000000
CP0 output	
bocp0_eret_addr	32'h800000b4
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
bocp0_intr_en_n	1'h0
bocp0_intr_mask	6'h00
bocp0_read_data	32'hxxxxxxxx
bocp0_stat_IPL	2'h0
bocp0_timer_intr	1'h0
PC and Instruction	
boalb_ovfs	1'h0
bomc_syscall	1'h1
bomc_undef_inst	1'h0
uidp_cp0_syscall	1'h1
uidp_cp0_undef_inst	1'h0
uidp_instr	32'h00000000
uodp_pseudo_pc	32'h800000b8
urisc_next_pc	32'h8001b400
tb_u_clk	1'h1

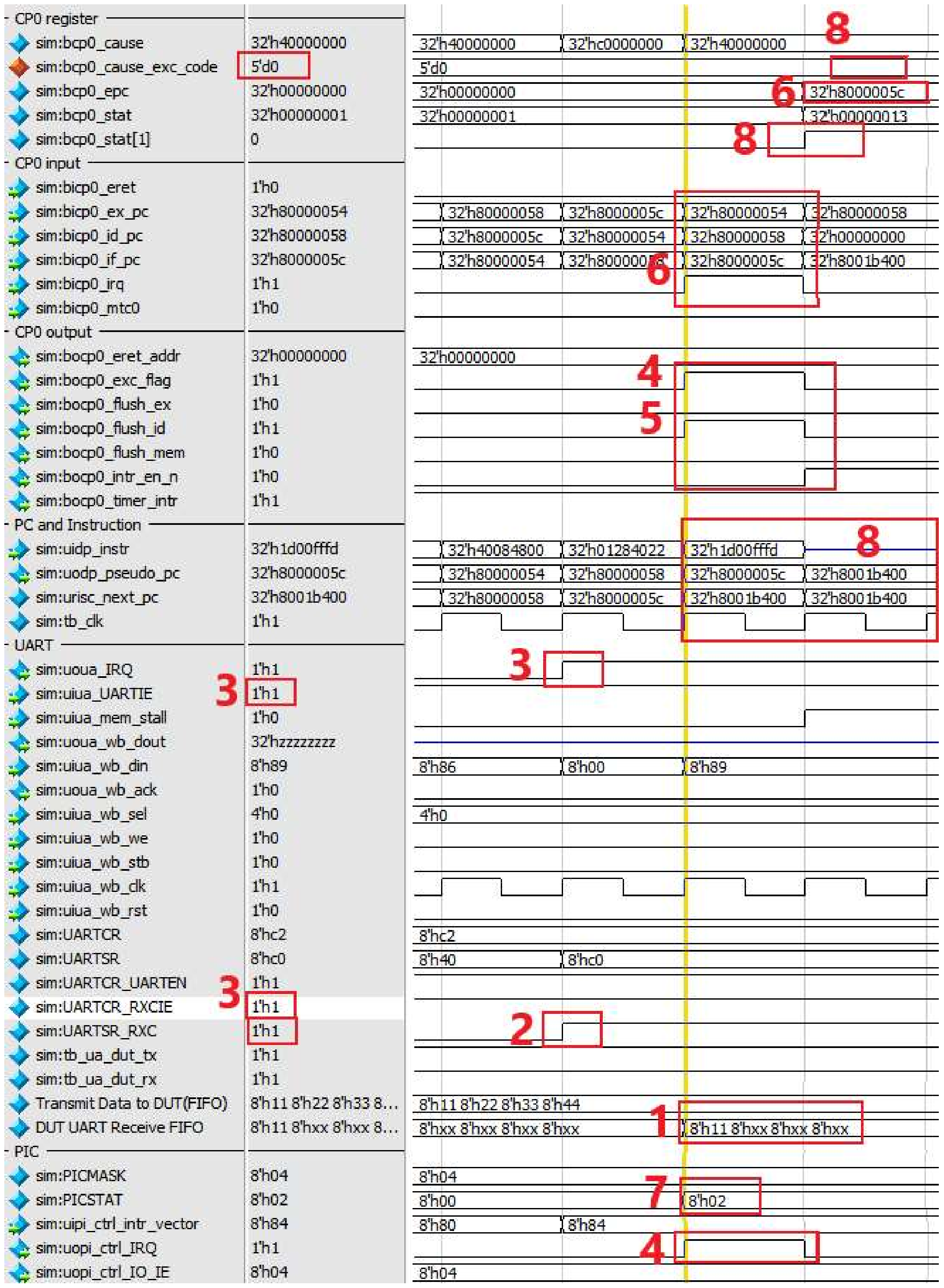
After turn from the sign-overflow exception, the Syscall will be handle (label "3"). We can clearly see that the Exception Code is 8, which is Syscall instruction (label "4").

<u>Test Case 3: Individual interrupt</u>

<u>UART interrupt</u>

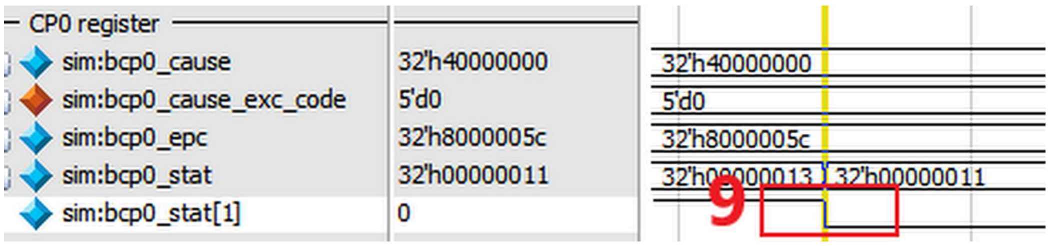


sim:tb_ua_dut_tx	1'h1
sim:tb_ua_dut_rx	1'h1
Transmit Data to DUT(FIFO)	8'h11 8'h22 8'h33 8...
DUT UART Receive FIFO	8'h11 8'hxx 8'hxx 8...
sim:urisc_IRQ	1'h1



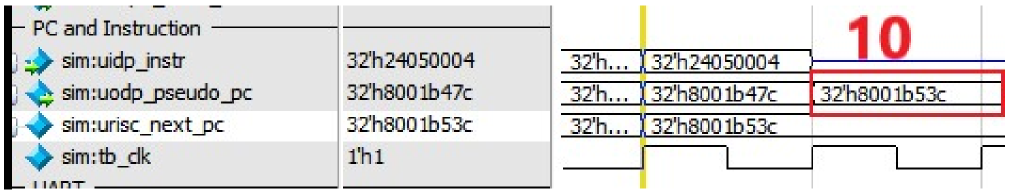
CP0 register	
sim:bcp0_cause	32'h40000000
sim:bcp0_cause_exc_code	5'd0
sim:bcp0_epc	32'h00000000
sim:bcp0_stat	32'h00000001
sim:bcp0_stat[1]	0
CP0 input	
sim:bicp0_eret	1'h0
sim:bicp0_ex_pc	32'h80000054
sim:bicp0_id_pc	32'h80000058
sim:bicp0_if_pc	32'h8000005c
sim:bicp0_irq	1'h1
sim:bicp0_mtc0	1'h0
CP0 output	
sim:bocp0_eret_addr	32'h00000000
sim:bocp0_exc_flag	1'h1
sim:bocp0_flush_ex	1'h0
sim:bocp0_flush_id	1'h1
sim:bocp0_flush_mem	1'h0
sim:bocp0_intr_en_n	1'h0
sim:bocp0_timer_intr	1'h1
PC and Instruction	
sim:uidp_instr	32'h1d00fffd
sim:uodp_pseudo_pc	32'h8000005c
sim:urisc_next_pc	32'h8001b400
sim:tb_clk	1'h1
UART	
sim:uoua_IRQ	1'h1
sim:uiua_UARTIE	1'h1
sim:uiua_mem_stall	1'h0
sim:uoua_wb_dout	32'hzzzzzzzz
sim:uiua_wb_din	8'h89
sim:uoua_wb_ack	1'h0
sim:uiua_wb_sel	4'h0
sim:uiua_wb_we	1'h0
sim:uiua_wb_stb	1'h0
sim:uiua_wb_clk	1'h1
sim:uiua_wb_rst	1'h0
sim:UARTCR	8'hc2
sim:UARTSR	8'hc0
sim:UARTCR_UARTEN	1'h1
sim:UARTCR_RXCIE	1'h1
sim:UARTSR_RXC	1'h1
sim:tb_ua_dut_tx	1'h1
sim:tb_ua_dut_rx	1'h1
Transmit Data to DUT(FIFO)	8'h11 8'h22 8'h33 8...
DUT UART Receive FIFO	8'h11 8'hxx 8'hxx 8...
PIC	
sim:PICMASK	8'h04
sim:PICSTAT	8'h02
sim:uipi_ctrl_intr_vector	8'h84
sim:uopi_ctrl_IRQ	1'h1
sim:uopi_ctrl_IO_IE	8'h04

1) The slave device transmit the data to the DUT.
2) After the DUT received 1 byte of data, it will rise the UART_RXC flag.
3) The trigger the interrupt request (IRQ), UART interrupt enable (UARTIE) and UART Received interrupt enable (UARTCR_RXCIE) must be set to high.
4) When IRQ occur, hardware will rise the bocp0_exc_flag
5) Flush the IF/ID pipeline
6) Load the IF stage's PC into $epc for return purpose after handler the exception.
7) PICSTAT will update the value that correspond to the IRQ source. For UART the value is 2.
8) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, Interrupt request the exception code is 0, which will write into the CP0 $cause by hardware.



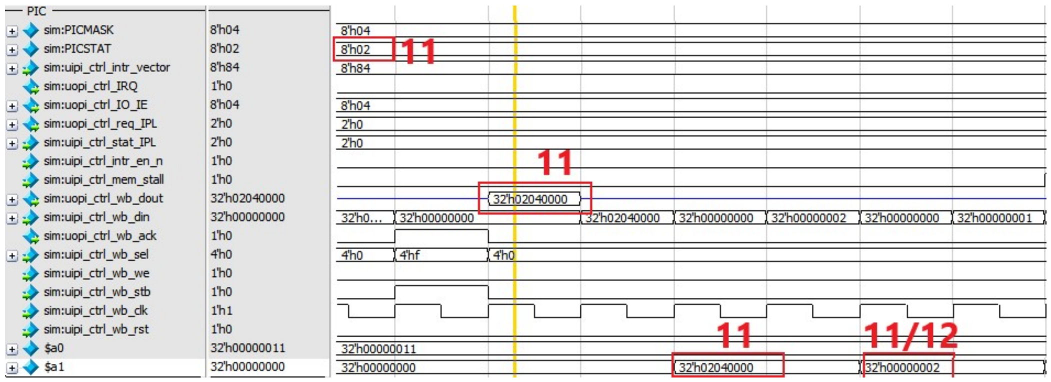
CP0 register		
sim:bcp0_cause	32'h40000000	32'h40000000
sim:bcp0_cause_exc_code	5'd0	5'd0
sim:bcp0_epc	32'h8000005c	32'h8000005c
sim:bcp0_stat	32'h00000011	32'h00000013 / 32'h00000011
sim:bcp0_stat[1]	0	

9) After store the register information into the stack, $status [1] will pull down by user, to enable further interrupt.



PC and Instruction		
sim:uidp_instr	32'h24050004	32'h... / 32'h24050004
sim:uodp_pseudo_pc	32'h8001b47c	32'h... / 32'h8001b47c / 32'h8001b53c
sim:urisc_next_pc	32'h8001b53c	32'h... / 32'h8001b53c
sim:tb_clk	1'h1	

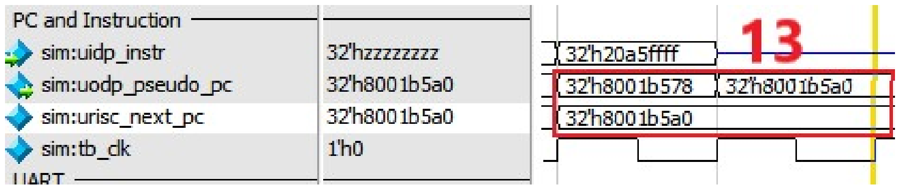
10) The exception code in $cause will decode and branch to the respective ISR. For IRQ, the ISR start at 0x8001b53c.



```
PIC
sim:PICMASK                 8'h04
sim:PICSTAT                 8'h02
sim:uipi_ctrl_intr_vector   8'h84
sim:uopi_ctrl_IRQ           1'h0
sim:uopi_ctrl_IO_IE         8'h04
sim:uopi_ctrl_req_IPL       2'h0
sim:uipi_ctrl_stat_IPL      2'h0
sim:uipi_ctrl_intr_en_n     1'h0
sim:uipi_ctrl_mem_stall     1'h0
sim:uopi_ctrl_wb_dout       32'h02040000
sim:uipi_ctrl_wb_din        32'h00000000
sim:uopi_ctrl_wb_ack        1'h0
sim:uipi_ctrl_wb_sel        4'h0
sim:uipi_ctrl_wb_we         1'h0
sim:uipi_ctrl_wb_stb        1'h0
sim:uipi_ctrl_wb_clk        1'h1
sim:uipi_ctrl_wb_rst        1'h0
$a0                         32'h00000011
$a1                         32'h00000000
```

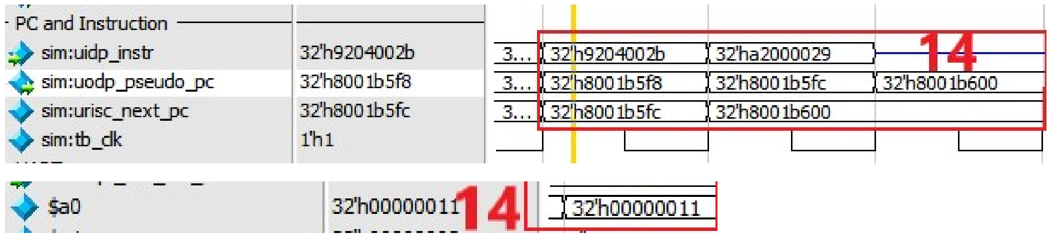
11) After Jump to the External interrupt ISR, exception handler will load the PICSTAT from Programmable interrupt controller (PIC) to $a1.

12) Decode the PICSTAT to figure out the IRQ trigger by which interrupt source. For UART the code is 2.



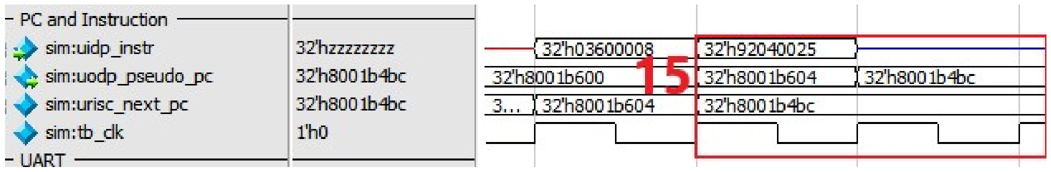
```
PC and Instruction
sim:uidp_instr         32'hzzzzzzzz
sim:uodp_pseudo_pc     32'h8001b5a0
sim:urisc_next_pc      32'h8001b5a0
sim:tb_clk             1'h0
UART
```

13) After that, it will branch to the respective interrupt source's ISR. For UART, the starting address for ISR is (0x8001b5a0).



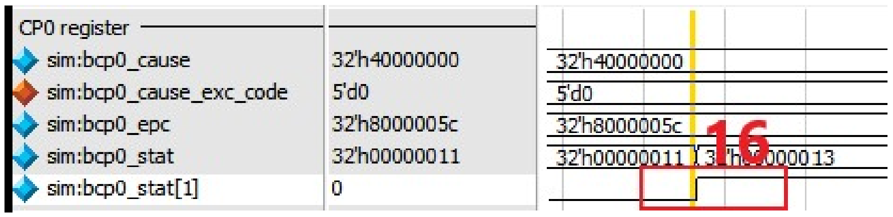
```
PC and Instruction
sim:uidp_instr         32'h9204002b
sim:uodp_pseudo_pc     32'h8001b5f8
sim:urisc_next_pc      32'h8001b5fc
sim:tb_clk             1'h1

$a0                    32'h00000011
```

14) UART Receive interrupt's ISR will load the data from UARTRDR to the $a0.



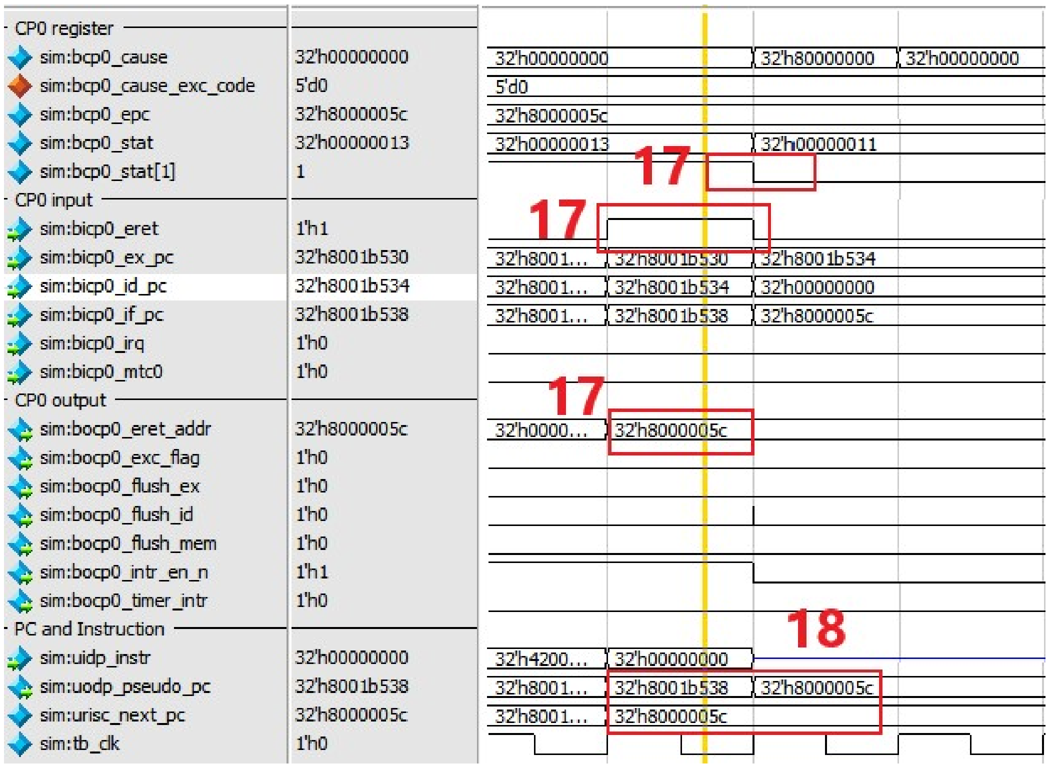
```
PC and Instruction
sim:uidp_instr         32'hzzzzzzzz
sim:uodp_pseudo_pc     32'h8001b4bc
sim:urisc_next_pc      32'h8001b4bc
sim:tb_clk             1'h0
UART
```

15) After ISR, it will branch to pop data section (0x8001b4bc).



```
CP0 register
sim:bcp0_cause            32'h40000000
sim:bcp0_cause_exc_code   5'd0
sim:bcp0_epc              32'h8000005c
sim:bcp0_stat             32'h00000011
sim:bcp0_stat[1]          0
```

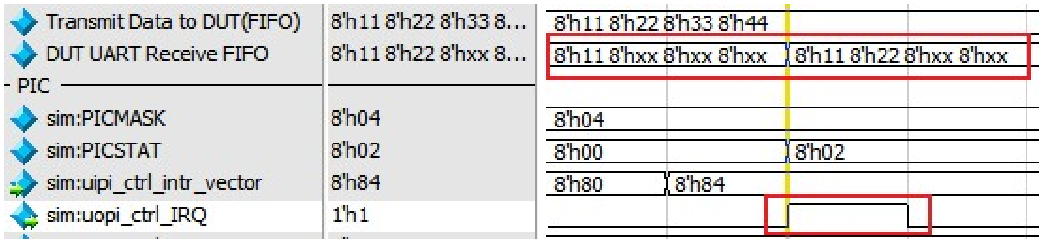
16) Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.



CP0 register	
sim:bcp0_cause	32'h00000000
sim:bcp0_cause_exc_code	5'd0
sim:bcp0_epc	32'h8000005c
sim:bcp0_stat	32'h00000013
sim:bcp0_stat[1]	1
CP0 input	
sim:bicp0_eret	1'h1
sim:bicp0_ex_pc	32'h8001b530
sim:bicp0_id_pc	32'h8001b534
sim:bicp0_if_pc	32'h8001b538
sim:bicp0_irq	1'h0
sim:bicp0_mtc0	1'h0
CP0 output	
sim:bocp0_eret_addr	32'h8000005c
sim:bocp0_exc_flag	1'h0
sim:bocp0_flush_ex	1'h0
sim:bocp0_flush_id	1'h0
sim:bocp0_flush_mem	1'h0
sim:bocp0_intr_en_n	1'h1
sim:bocp0_timer_intr	1'h0
PC and Instruction	
sim:uidp_instr	32'h00000000
sim:uodp_pseudo_pc	32'h8001b538
sim:urisc_next_pc	32'h8000005c
sim:tb_clk	1'h0

17) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.

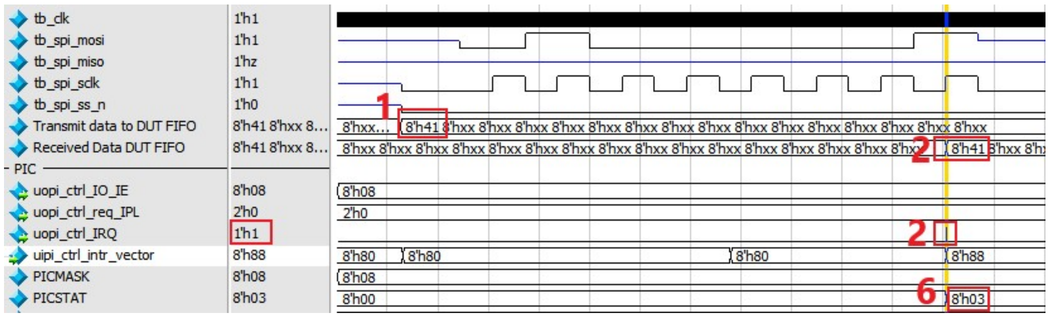
18) Jump back to the User program in the next clock cycle.



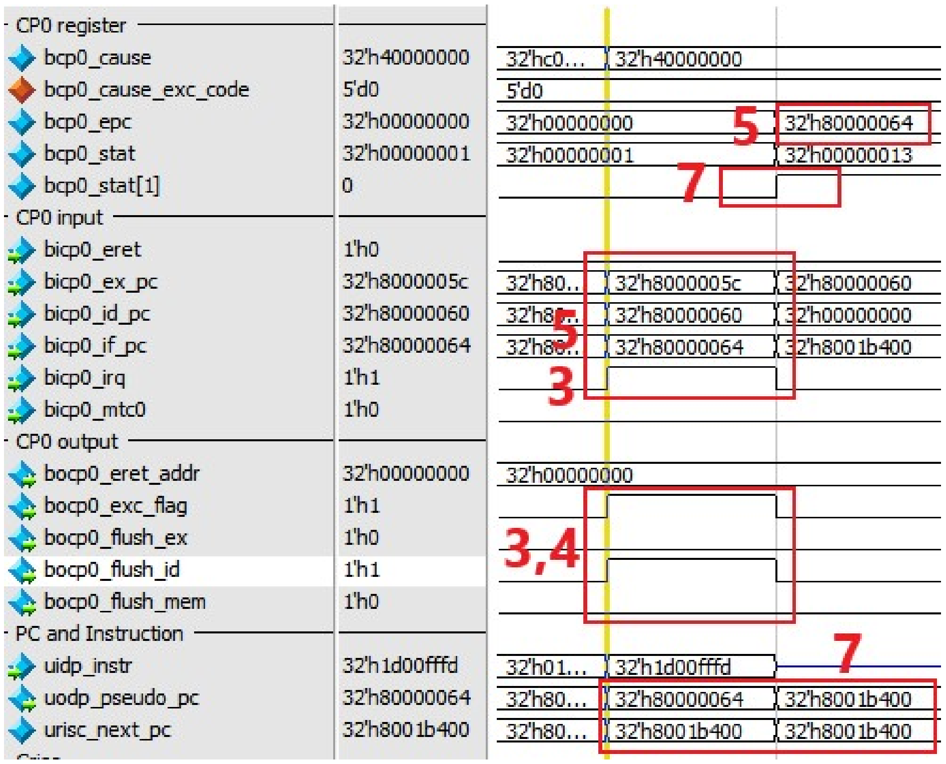
Transmit Data to DUT(FIFO)	8'h11 8'h22 8'h33 8...
DUT UART Receive FIFO	8'h11 8'h22 8'hxx 8...
PIC	
sim:PICMASK	8'h04
sim:PICSTAT	8'h02
sim:uipi_ctrl_intr_vector	8'h84
sim:uopi_ctrl_IRQ	1'h1

The Second byte of Data received by DUT and hardware will trigger the IRQ. The step will perform same as step 1 until step 18.

<u>SPI interrupt</u>

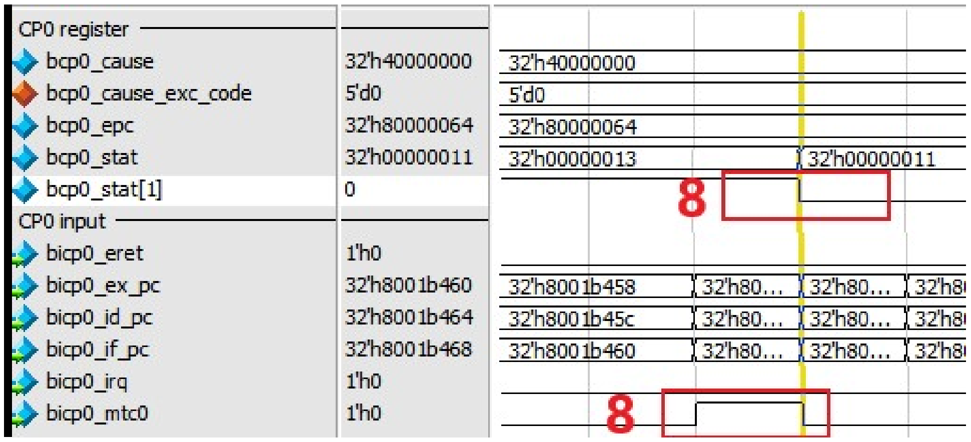


tb_clk	1'h1
tb_spi_mosi	1'h1
tb_spi_miso	1'hz
tb_spi_sdk	1'h1
tb_spi_ss_n	1'h0
Transmit data to DUT FIFO	8'h41 8'hxx 8...
Received Data DUT FIFO	8'h41 8'hxx 8...
- PIC	
uopi_ctrl_IO_IE	8'h08
uopi_ctrl_req_IPL	2'h0
uopi_ctrl_IRQ	1'h1
uipi_ctrl_intr_vector	8'h88
PICMASK	8'h08
PICSTAT	8'h03



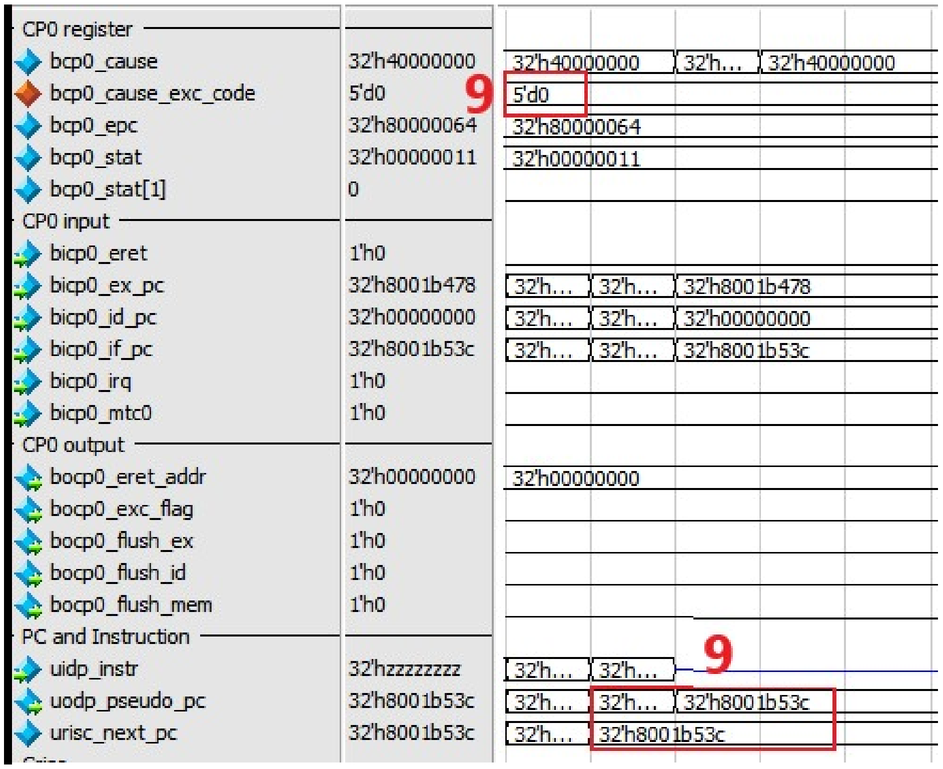
- CP0 register	
bcp0_cause	32'h40000000
bcp0_cause_exc_code	5'd0
bcp0_epc	32'h00000000
bcp0_stat	32'h00000001
bcp0_stat[1]	0
- CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8000005c
bicp0_id_pc	32'h80000060
bicp0_if_pc	32'h80000064
bicp0_irq	1'h1
bicp0_mtc0	1'h0
- CP0 output	
bocp0_eret_addr	32'h00000000
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
- PC and Instruction	
uidp_instr	32'h1d00fffd
uodp_pseudo_pc	32'h80000064
urisc_next_pc	32'h8001b400

1) The slave device transmit the data to the DUT.
2) After the DUT received 1 byte of data, it will rise the IRQ.
3) When IRQ occur, hardware will rise the bocp0_exc_flag
4) Flush the IF/ID pipeline
5) Load the IF stage's PC into $epc for return purpose after handler the exception.
6) PICSTAT will be update the value that correspond to the IRQ source. For SPI the value is 3.
7) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, Interrupt request the exception code is 0, which will write into the CP0 $cause by hardware.



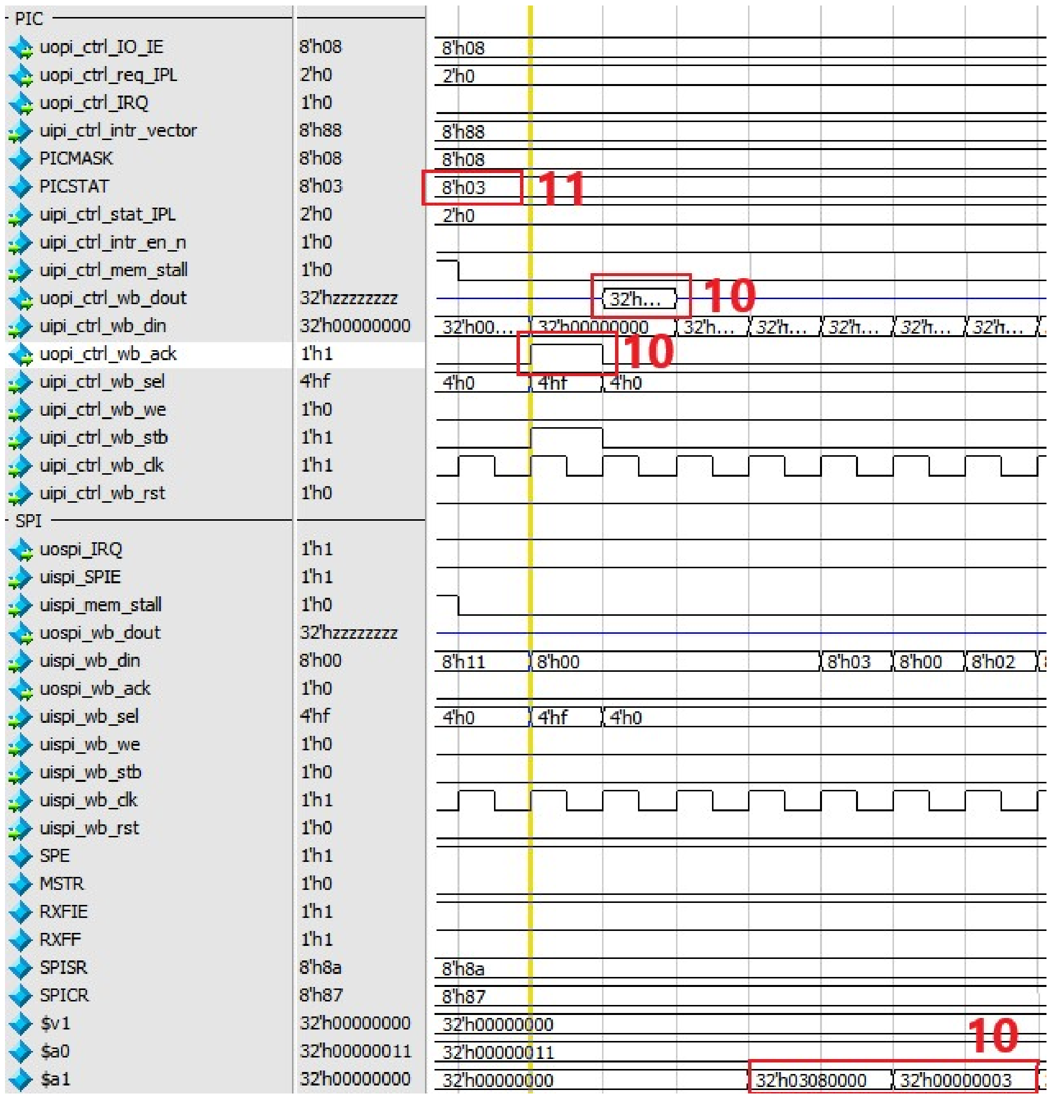
CP0 register	
bcp0_cause	32'h40000000
bcp0_cause_exc_code	5'd0
bcp0_epc	32'h80000064
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8001b460
bicp0_id_pc	32'h8001b464
bicp0_if_pc	32'h8001b468
bicp0_irq	1'h0
bicp0_mtc0	1'h0

8) After store the register information into the stack, $status [1] will pull down by user, to enable further interrupt.



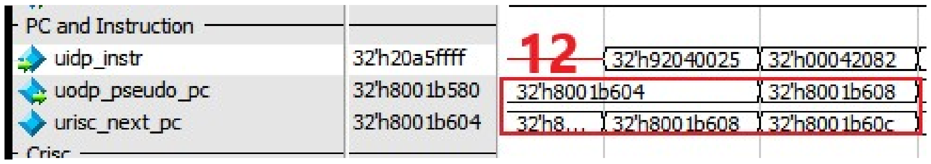
CP0 register	
bcp0_cause	32'h40000000
bcp0_cause_exc_code	5'd0
bcp0_epc	32'h80000064
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_eret	1'h0
bicp0_ex_pc	32'h8001b478
bicp0_id_pc	32'h00000000
bicp0_if_pc	32'h8001b53c
bicp0_irq	1'h0
bicp0_mtc0	1'h0
CP0 output	
bocp0_eret_addr	32'h00000000
bocp0_exc_flag	1'h0
bocp0_flush_ex	1'h0
bocp0_flush_id	1'h0
bocp0_flush_mem	1'h0
PC and Instruction	
uidp_instr	32'hzzzzzzzz
uodp_pseudo_pc	32'h8001b53c
urisc_next_pc	32'h8001b53c

9) The exception code in $cause will decode and branch to the respective ISR. For IRQ, the ISR start at 0x8001b53c.

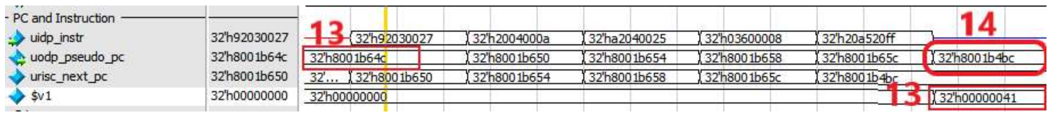


10) After Jump to the External interrupt ISR, exception handler will load the PICSTAT from Programmable interrupt controller (PIC) to $a1.

11) Decode the PICSTAT to figure out the IRQ trigger by which interrupt source. For SPI the code is 3.

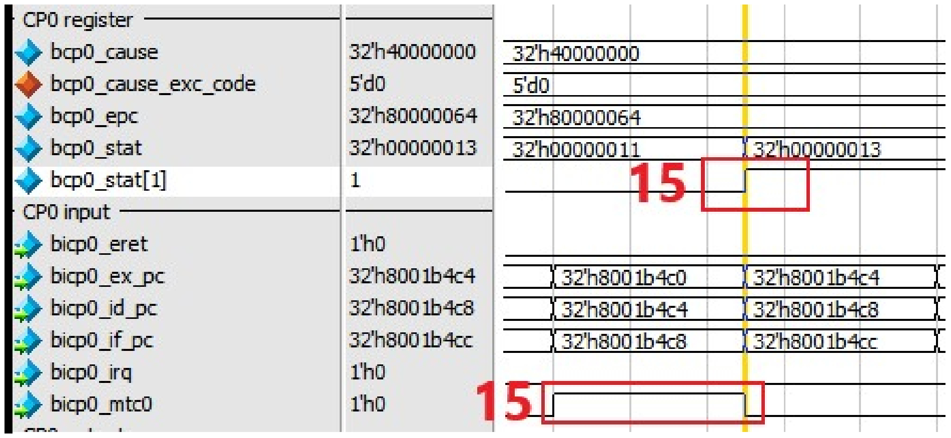


12) After that, it will branch to the respective interrupt source's ISR. For SPI, the starting address for ISR is (0x8001b604).

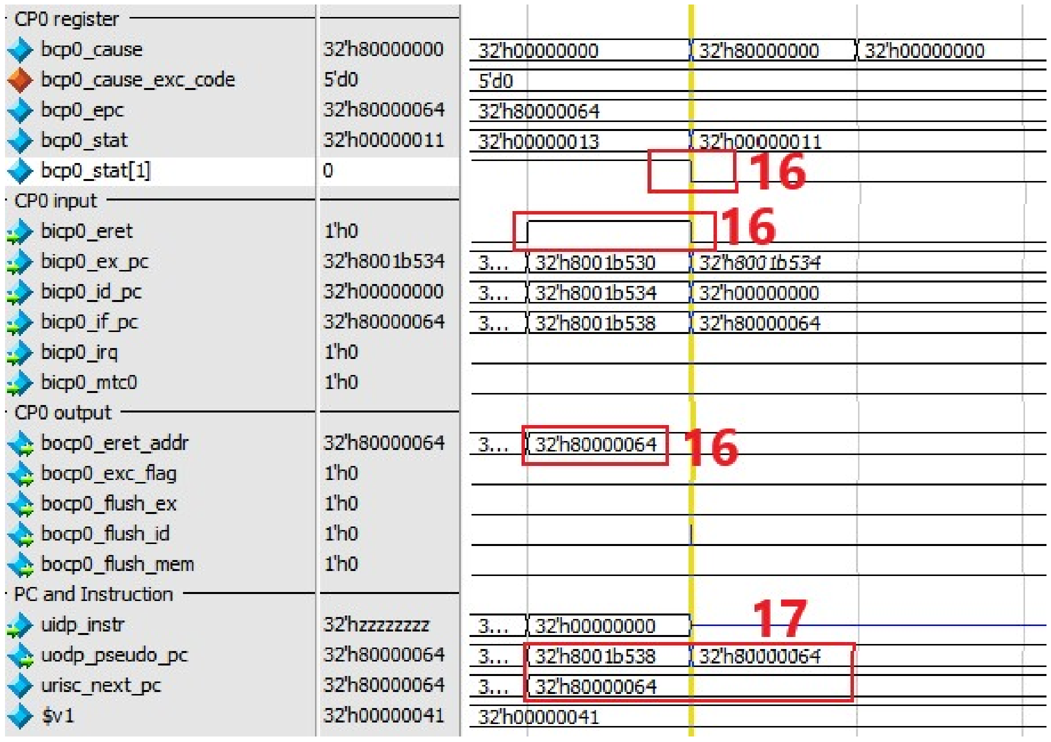


13) SPI Receive interrupt’s ISR will load the data from SPIRDR to the $v1.

14) After ISR, it will branch to pop data section (0x8001b4bc).

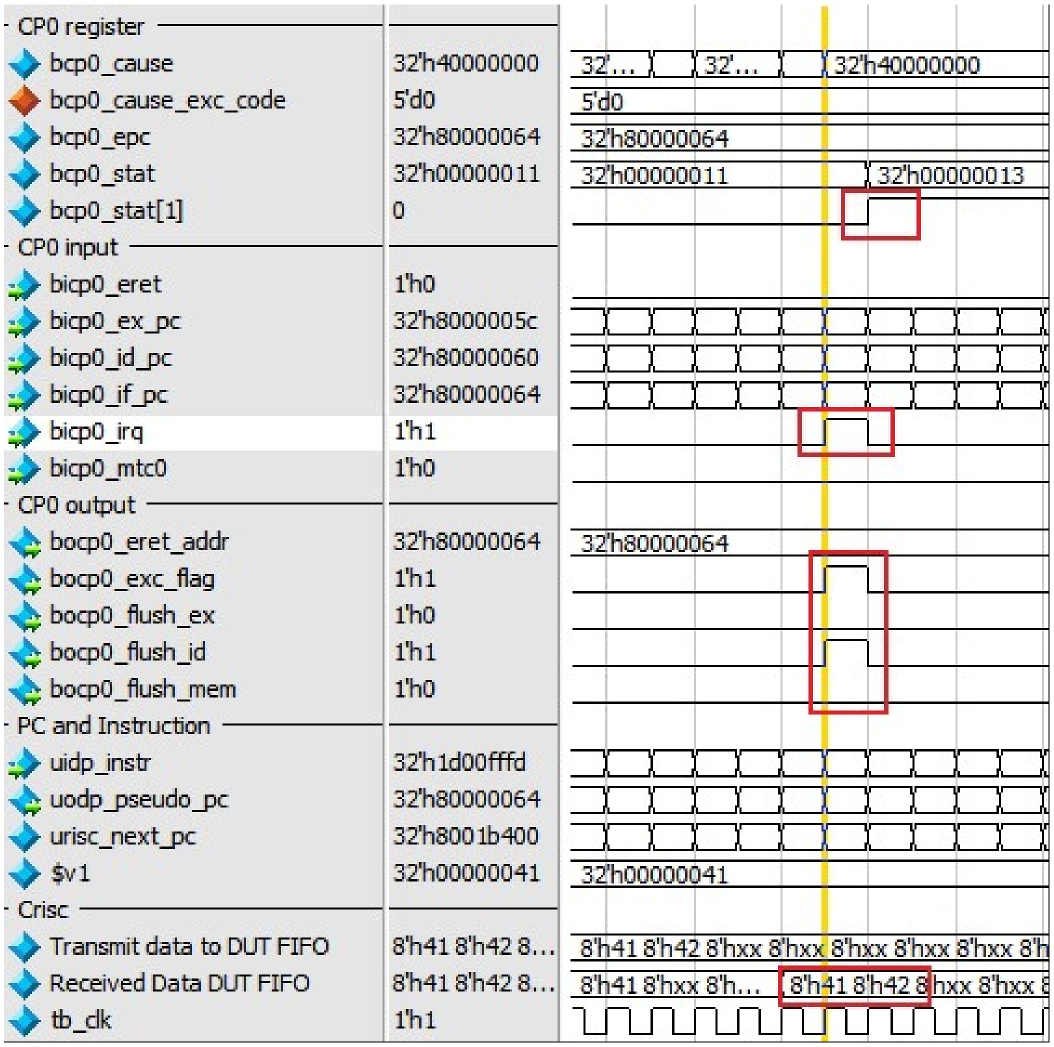


15) Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.



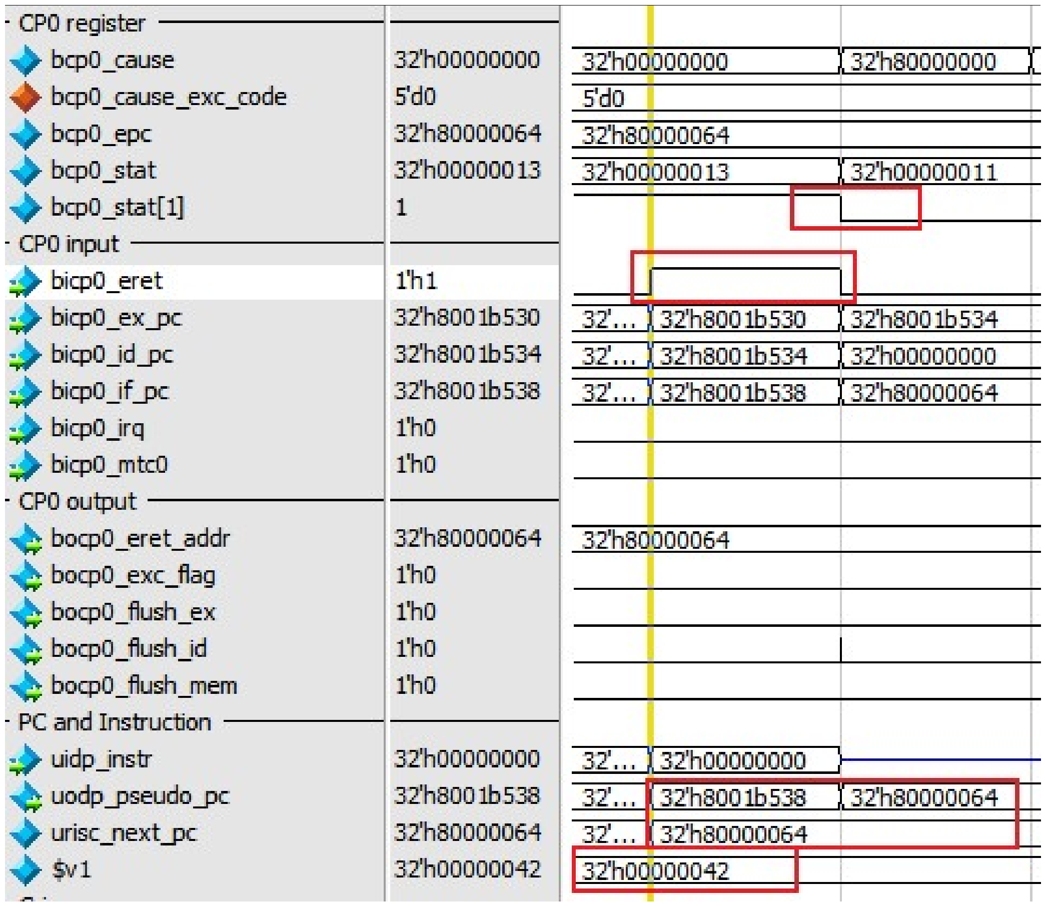
16) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.

17) Jump back to the User program in the next clock cycle.



```
- CP0 register
bcp0_cause                  32'h40000000
bcp0_cause_exc_code         5'd0
bcp0_epc                    32'h80000064
bcp0_stat                   32'h00000011
bcp0_stat[1]                0
- CP0 input
bicp0_eret                  1'h0
bicp0_ex_pc                 32'h8000005c
bicp0_id_pc                 32'h80000060
bicp0_if_pc                 32'h80000064
bicp0_irq                   1'h1
bicp0_mtc0                  1'h0
- CP0 output
bocp0_eret_addr             32'h80000064
bocp0_exc_flag              1'h1
bocp0_flush_ex              1'h0
bocp0_flush_id              1'h1
bocp0_flush_mem             1'h0
- PC and Instruction
uidp_instr                  32'h1d00fffd
uodp_pseudo_pc              32'h80000064
urisc_next_pc               32'h8001b400
$v1                         32'h00000041
- Crisc
Transmit data to DUT FIFO   8'h41 8'h42 8...
Received Data DUT FIFO      8'h41 8'h42 8...
tb_clk                      1'h1
```

The Second byte of Data received by DUT and hardware will trigger the IRQ. The step will perform same as step 1 until step 17.

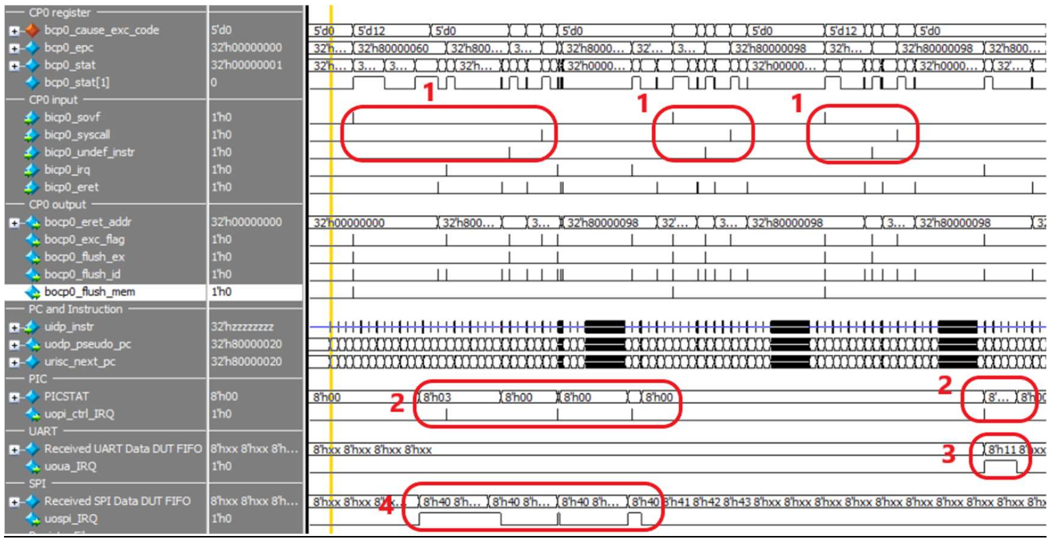


```
- CP0 register
  bcp0_cause            32'h00000000
  bcp0_cause_exc_code   5'd0
  bcp0_epc              32'h80000064
  bcp0_stat             32'h00000013
  bcp0_stat[1]          1
- CP0 input
  bicp0_eret            1'h1
  bicp0_ex_pc           32'h8001b530
  bicp0_id_pc           32'h8001b534
  bicp0_if_pc           32'h8001b538
  bicp0_irq             1'h0
  bicp0_mtc0            1'h0
- CP0 output
  bocp0_eret_addr       32'h80000064
  bocp0_exc_flag        1'h0
  bocp0_flush_ex        1'h0
  bocp0_flush_id        1'h0
  bocp0_flush_mem       1'h0
- PC and Instruction
  uidp_instr            32'h00000000
  uodp_pseudo_pc        32'h8001b538
  urisc_next_pc         32'h80000064
  $v1                   32'h00000042
```

32'h00000000
32'h80000000
5'd0
32'h80000064
32'h00000013
32'h00000011
32'... 32'h8001b530 32'h8001b534
32'... 32'h8001b534 32'h00000000
32'... 32'h8001b538 32'h80000064
32'h80000064
32'... 32'h00000000
32'... 32'h8001b538 32'h80000064
32'... 32'h80000064
32'h00000042

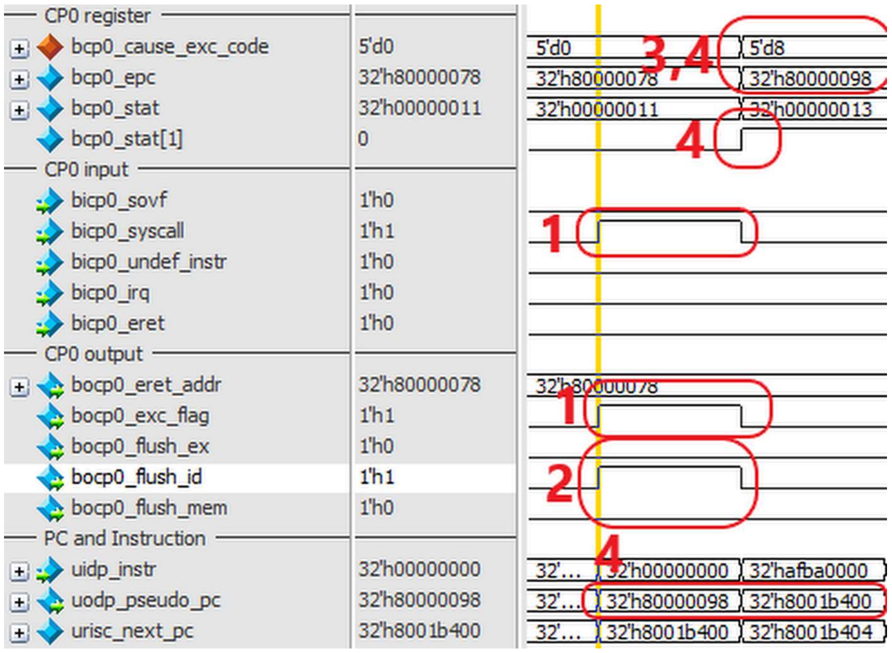
After perform the step 1 until 17, the value in $v1 is 42.

Test Case 4: Multiple interrupt and Multiple Trap.

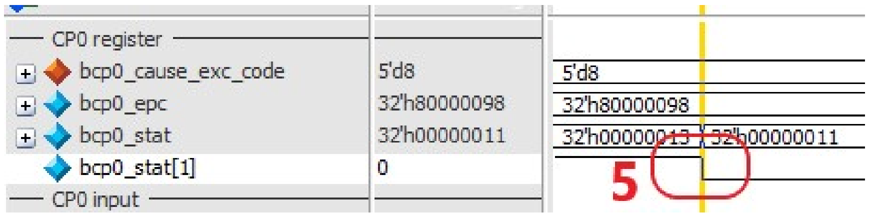


In figure above, the label "1" indicate that the internal exception event (Sign-overflow, undefined instruction and syscall) occur continuously on the Server. The Label "2" show that the programmable interrupt controller (PIC) generate the interrupt request (IRQ) to the core processor 0(CP0) based on the UART and SPI Interrupt Request (label 3 and label 4). When the Server received one byte of data from the Client through the SPI and UART, SPI controller and UART controller will generate an interrupt to the PIC, PIC will manipulate the multiple interrupt occurrence based on the interrupt priority level that pre-set by user and generate the interrupt request to CP0. In this test case, we are more interesting about the Nested Exception and the Exception Conflicting between the Trap and Interrupt.

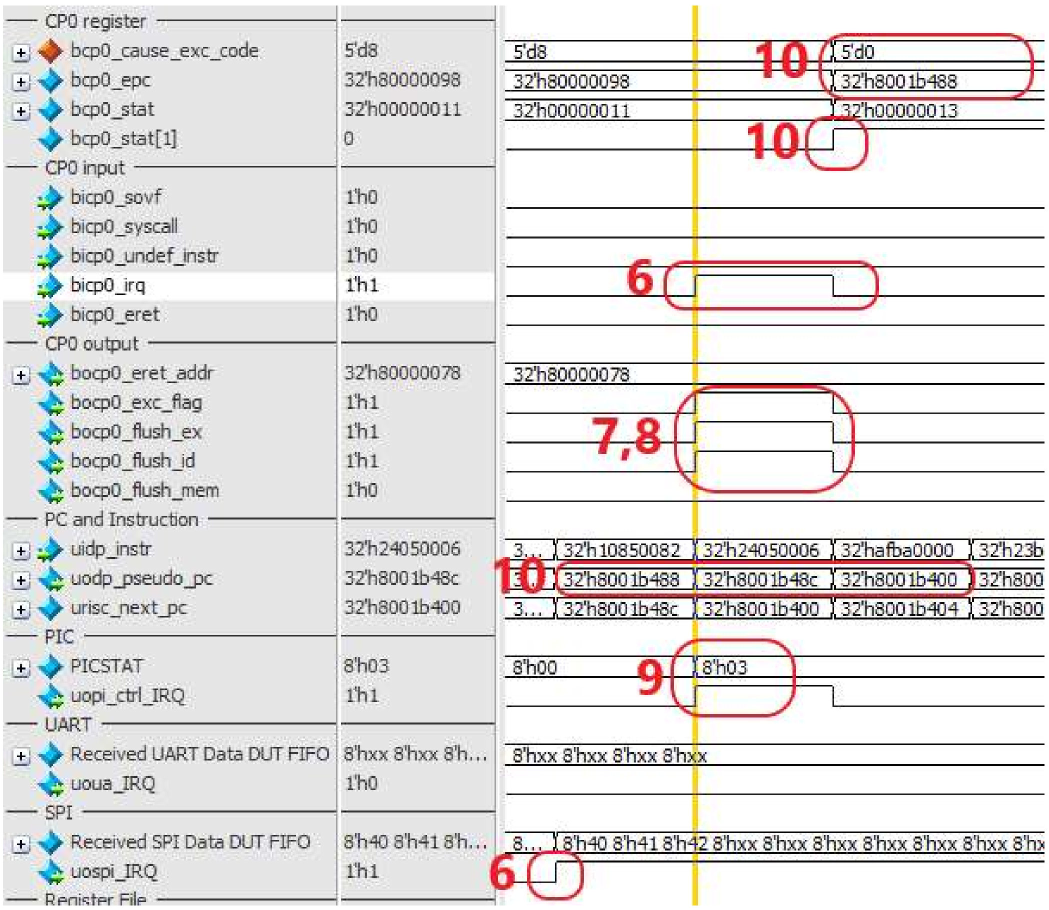
Nested Exception



1) When detected the syscall at Main control unit (ID stage), hardware will raise the bocp0_exc_flag.
2) Flush the IF/ID.
3) Put the IF stage's PC to the $epc for return purpose after handler the exception.
4) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, syscall the exception code is 8, which will write into the CP0 $cause by hardware.



5) After push the register information into Stack, $status [1] will pull down by user, to enable further interrupt. The exception code in $cause will decode and branch to the respective ISR.



CP0 register	
bcp0_cause_exc_code	5'd8
bcp0_epc	32'h80000098
bcp0_stat	32'h00000011
bcp0_stat[1]	0
CP0 input	
bicp0_sovf	1'h0
bicp0_syscall	1'h0
bicp0_undef_instr	1'h0
bicp0_irq	1'h1
bicp0_eret	1'h0
CP0 output	
bocp0_eret_addr	32'h80000078
bocp0_exc_flag	1'h1
bocp0_flush_ex	1'h1
bocp0_flush_id	1'h1
bocp0_flush_mem	1'h0
PC and Instruction	
uidp_instr	32'h24050006
uodp_pseudo_pc	32'h8001b48c
urisc_next_pc	32'h8001b400
PIC	
PICSTAT	8'h03
uopi_ctrl_IRQ	1'h1
UART	
Received UART Data DUT FIFO	8'hxx 8'hxx 8'h...
uoua_IRQ	1'h0
SPI	
Received SPI Data DUT FIFO	8'h40 8'h41 8'h...
uospi_IRQ	1'h1
Register File	

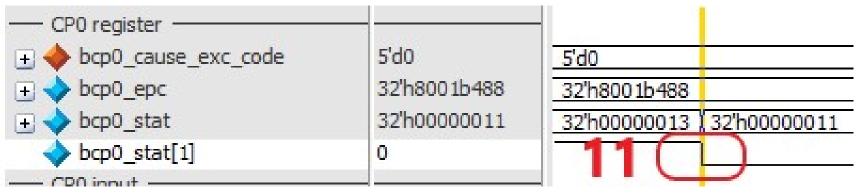
6) During decode the exception code, a byte of SPI data had received and generate an interrupt request. This is where Nested Exception event occur.

7) When IRQ occur, hardware will rise the bocp0_exc_flag and flush the IF/ID pipeline

8) Load the IF stage's PC into $epc for return purpose after handler the exception in this case the IF stage's PC is the PC in the exception handler.

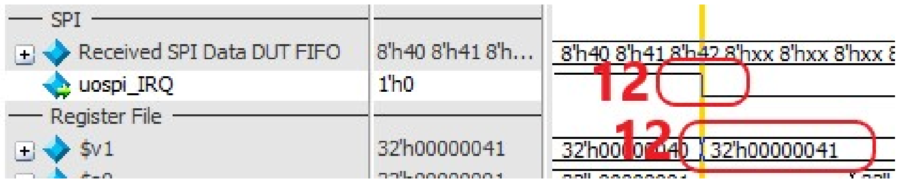
9) PICSTAT will be update the value that correspond to the IRQ source. For SPI the value is 3.

10) Jump to exception handler (0x8001b400) in the next clock cycle and raise the exception flag in CP0 $status [1] to disable further exception occur. For, Interrupt request the exception code is 0, which will write into the CP0 $cause by hardware.



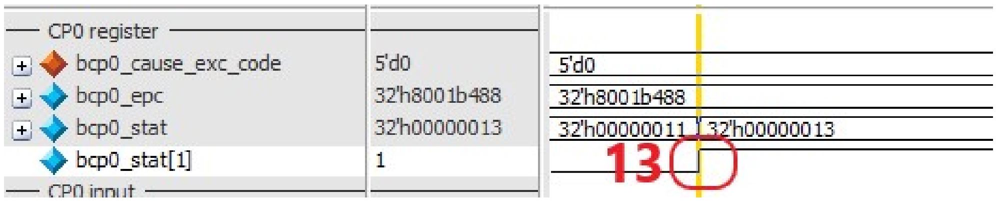
CP0 register		
+	bcp0_cause_exc_code	5'd0
+	bcp0_epc	32'h8001b488
+	bcp0_stat	32'h00000011
	bcp0_stat[1]	0

11) After store the register information into the stack, $status [1] will pull down by user, to enable further interrupt. The exception code in $cause and PICSTAT in PIC will decode and branch to the respective ISR.



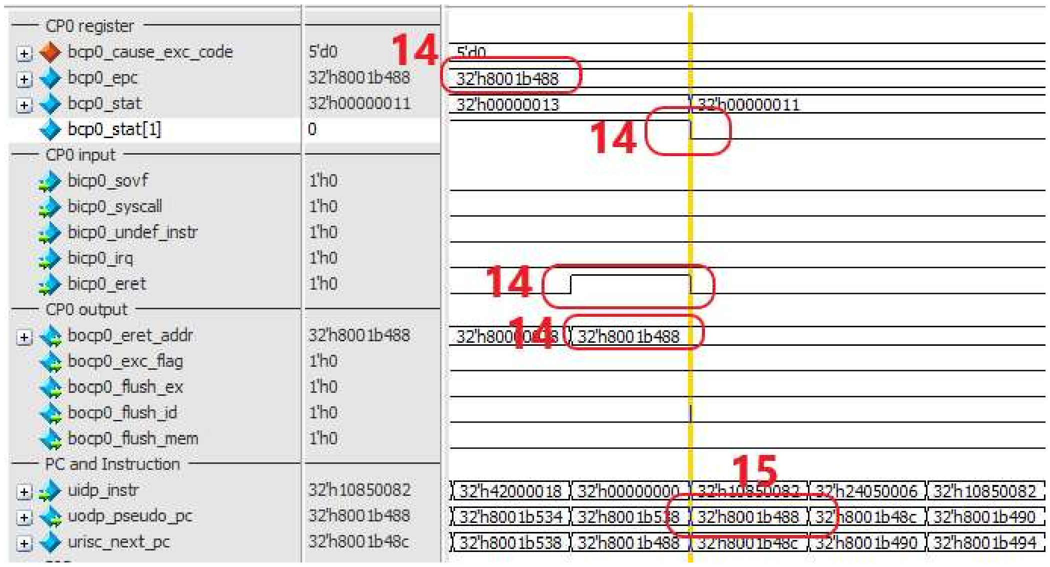
SPI		
+	Received SPI Data DUT FIFO	8'h40 8'h41 8'h...
	uospi_IRQ	1'h0
Register File		
+	$v1	32'h00000041

12) SPI Receive interrupt's ISR will load the data from SPIRDR to the $v1. The SPI interrupt request was de-asserted.



CP0 register		
+	bcp0_cause_exc_code	5'd0
+	bcp0_epc	32'h8001b488
+	bcp0_stat	32'h00000013
	bcp0_stat[1]	1

13) After ISR, it will branch to pop data section. Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.

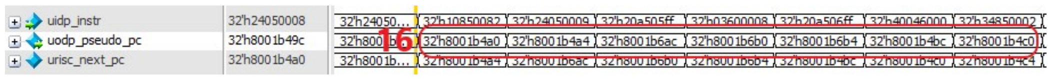


CP0 register		
+	bcp0_cause_exc_code	5'd0
+	bcp0_epc	32'h8001b488
+	bcp0_stat	32'h00000011
	bcp0_stat[1]	0
CP0 input		
	bicp0_sovf	1'h0
	bicp0_syscall	1'h0
	bicp0_undef_instr	1'h0
	bicp0_irq	1'h0
	bicp0_eret	1'h0
CP0 output		
+	bocp0_eret_addr	32'h8001b488
	bocp0_exc_flag	1'h0
	bocp0_flush_ex	1'h0
	bocp0_flush_id	1'h0
	bocp0_flush_mem	1'h0
PC and Instruction		
+	uidp_instr	32'h10850082
+	uodp_pseudo_pc	32'h8001b488
+	urisc_next_pc	32'h8001b48c

14
5'd0
32'h8001b488
32'h00000013
32'h00000011
14
14
14
32'h8001b488
15
32'h42000018 32'h00000000 32'h10850082 32'h24050006 32'h10850082
32'h8001b534 32'h8001b538 32'h8001b488 32'h8001b48c 32'h8001b490
32'h8001b538 32'h8001b488 32'h8001b48c 32'h8001b490 32'h8001b494

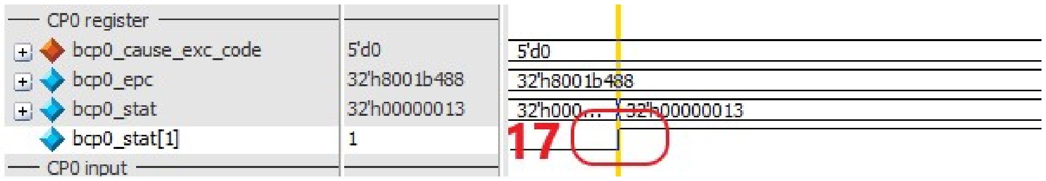
14) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.

15) Jump back to the previous exception handler in the clock cycle.



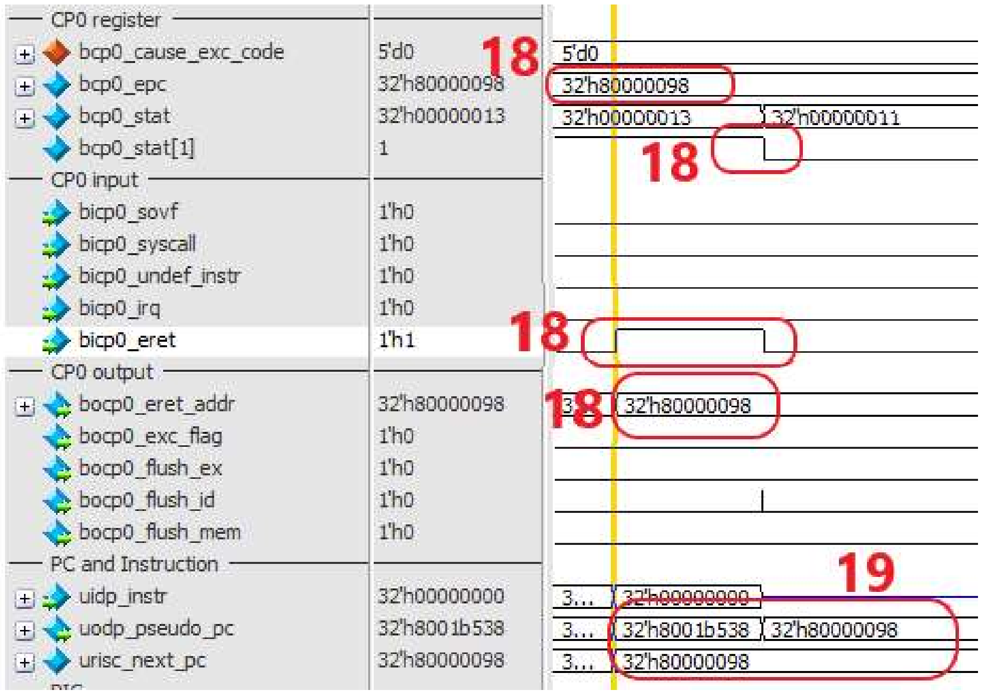
```
uidp_instr       32'h24050008  32'h24050... 32'h10850082 32'h24050009 32'h20a505ff 32'h03600008 32'h20a506ff 32'h40046000 32'h34850002
uodp_pseudo_pc   32'h8001b49c  32'h8001b... 32'h8001b4a0 32'h8001b4a4 32'h8001b6ac 32'h8001b6b0 32'h8001b6b4 32'h8001b4bc 32'h8001b4c0
urisc_next_pc    32'h8001b4a0  32'h8001b... 32'h8001b4a4 32'h8001b6ac 32'h8001b6b0 32'h8001b6b4 32'h8001b4bc 32'h8001b4c0 32'h8001b4c4
```

16) After return, the exception handler will continue to decode exception code and branch to the respective ISR. For syscall, the ISR start at 0x8001b6ac. After ISR, it will branch to pop data section (0x8001b4bc).



```
CP0 register
bcp0_cause_exc_code   5'd0           5'd0
bcp0_epc              32'h8001b488   32'h8001b488
bcp0_stat             32'h00000013   32'h000...  32'h00000013
bcp0_stat[1]          1
CP0 input
```

17) Before pop the data from stack, the $status [1] will raise by user to disable further interrupt when pop the data.



```
CP0 register
bcp0_cause_exc_code   5'd0           5'd0
bcp0_epc              32'h80000098   32'h80000098
bcp0_stat             32'h00000013   32'h00000013  32'h00000011
bcp0_stat[1]          1
CP0 input
bicp0_sovf            1'h0
bicp0_syscall         1'h0
bicp0_undef_instr     1'h0
bicp0_irq             1'h0
bicp0_eret            1'h1
CP0 output
bocp0_eret_addr       32'h80000098   3... 32'h80000098
bocp0_exc_flag        1'h0
bocp0_flush_ex        1'h0
bocp0_flush_id        1'h0
bocp0_flush_mem       1'h0
PC and Instruction
uidp_instr            32'h00000000   3... 32'h00000000
uodp_pseudo_pc        32'h8001b538   3... 32'h8001b538  32'h80000098
urisc_next_pc         32'h80000098   3... 32'h80000098
```

18) After pop the Data, the exception code in $cause will clear by user and eret will asserted. eret will pull down the $status [1] and put the $epc into pc.

19) Jump back to the User program in the next clock cycle.

Exception Conflicting between the Trap (Sign-Overflow) and Interrupt (UART).

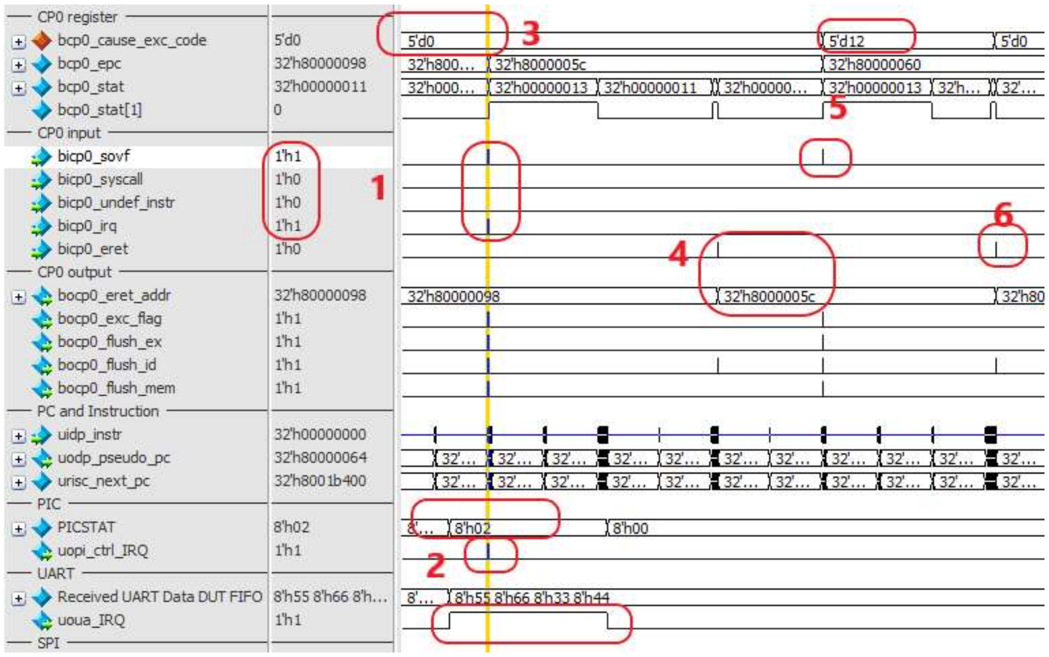


Figure above show that the exception scheme when exception conflicting between the sign-overflow and the interrupt request. In label "1", we can clearly observed that the Sign-overflow and the interrupt request had asserted in the same clock cycle. The interrupt request is asserted by the UART controller when receive one byte of data from the client (label "2"). During this kind of situation, the priority will goes to the interrupt request. In label "3" we can observed that the value of exception code in the CP0 $cause is "0", which is IRQ. After Serving the IRQ, "eret" will asserted and return to the user program (label "4"). The Sign-overflow asserted again, and jump to the exception handler in next clock cycle. In label "5", we can observe that the exception code is 12. In label "6" show that the return to the user program after handle the sign-overflow event.

<u>Exception Conflicting between the Trap (Sign-Overflow, Undefined instruction) and Interrupt (UART).</u>

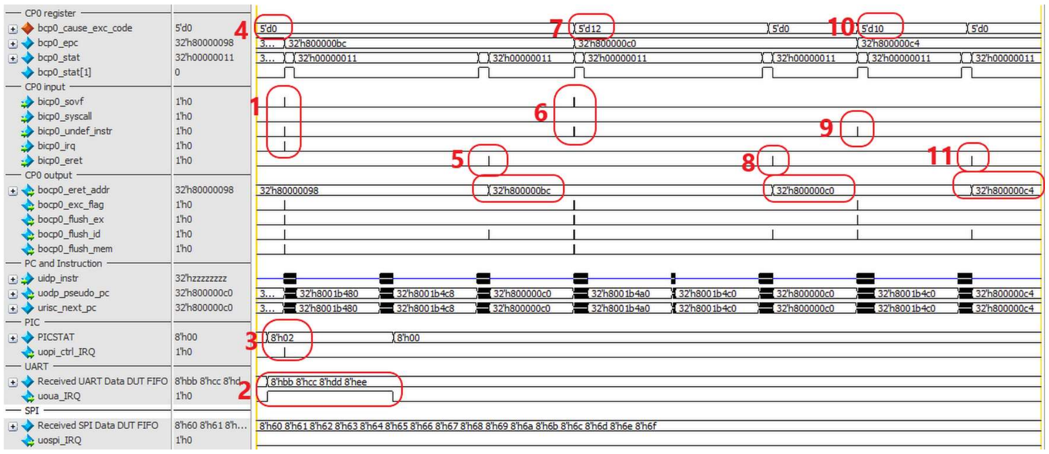


Figure above show that the flow when exception conflicting between the sign-overflow, undefined instruction and the interrupt request. In label "1", we can clearly observed that the sign-overflow, undefined instruction and the interrupt request was assert in the same clock cycle. The interrupt request is asserted by the UART controller when receive one byte of data from the client (label "2"). In label "3", show that the register PICSTAT is "2" which is UART interrupt request. When conflicting occur, the priority will goes to IRQ, in label "4" we can observed that the value of exception code in the CP0 $cause is "0", which is IRQ. After Serving the IRQ, "eret" will asserted and return to the user program (label "5"). In label "6" shows that the conflicting between the sign-overflow and undefined instruction. The sign-overflow will handle prior than the undefined instruction. In label "7", the value of exception code in the CP0 $cause is 12, which is sign-overflow. After serving the sign-overflow, "eret" will asserted and return to the user program (label "8"). Lastly, the undefined instruction will be handle after the sign-overflow. In label "10", we can observe that the exception code is 10, which is undefined instruction. In label "11" show that the return to the user program after handle the undefined instruction event. The sequence for handle the exception event when multiple exception event happen in same clock cycle is interrupt request (IRQ) > sign-overflow > undefined instruction.

6.4 Test Bench

```
`timescale 1ns / 10ps
`default_nettype none
`define  TEST_CODE_PATH "DUT_TESTCODE.txt"
`define                    EXC_HANDLER "exception_handler.txt"
`define  TEST_CODE_PATH_2 "SLAVE_TESTCODE.txt"

module tb_r32_pipeline();
//declaration
//======= INPUT =======
// System signal

reg              tb_clk;
reg              tb_rst;

wire    tb_spi_mosi;
wire    tb_spi_miso;
wire    tb_spi_sclk;
wire    tb_spi_ss_n;

wire    tb_fc_sclk;
wire    tb_fc_ss;
wire    tb_fc_MOSI;
wire    tb_fc_MISO1;
wire    tb_fc_MISO2;
wire    tb_fc_MISO3;

wire    tb_fc_sclk_2;
wire    tb_fc_ss_2;
wire    tb_fc_MOSI_2;
wire    tb_fc_MISO1_2;
wire    tb_fc_MISO2_2;
wire    tb_fc_MISO3_2;

wire [31:0]             tb_GPIO;
wire [31:0]             tb_GPIO_2;

wire    tb_ua_dut_tx;
wire    tb_ua_dut_rx;

//*********** INSTANTIATION ************
crisc
dut_c_risc
(
//======= INPUT =======
//GPIO
.urisc_GPIO(tb_GPIO),

//SPI controller
.uiorisc_spi_mosi(tb_spi_mosi),
.uiorisc_spi_miso(tb_spi_miso),
.uiorisc_spi_sclk(tb_spi_sclk),
.uiorisc_spi_ss_n(tb_spi_ss_n),

//UART controller
.uorisc_ua_tx_data(tb_ua_dut_tx),
//.uorisc_ua_rts(),
.uirisc_ua_rx_data(tb_ua_dut_rx),
//.uirisc_ua_cts(1'b0),

//FLASH controller
.uorisc_fc_sclk(tb_fc_sclk),
.uiorisc_fc_MOSI(tb_fc_MOSI),
.uirisc_fc_MISO1(tb_fc_MISO1),
.uirisc_fc_MISO2(tb_fc_MISO2),
.uirisc_fc_MISO3(tb_fc_MISO3),
.uorisc_fc_ss(tb_fc_ss),

// System signal
.uirisc_clk_100mhz(tb_clk),
.uirisc_rst(tb_rst)
);

s25fl128s SPI_flash
(.SI(tb_fc_MOSI), //IO0
.SO(tb_fc_MISO1), //IO1
.SCK(tb_fc_sclk),
.CSNeg(tb_fc_ss),
.RSTNeg(tb_rst),
.WPNeg(tb_fc_MISO2), //IO2
.HOLDNeg(tb_fc_MISO3));  //IO3

crisc
dut_c_risc_2//Client
(
//======= INPUT =======
//GPIO
.urisc_GPIO(tb_GPIO_2),

//SPI controller
.uiorisc_spi_mosi(tb_spi_mosi),
.uiorisc_spi_miso(tb_spi_miso),
.uiorisc_spi_sclk(tb_spi_sclk),
.uiorisc_spi_ss_n(tb_spi_ss_n),

//UART controller
.uorisc_ua_tx_data(tb_ua_dut_rx),
//.uorisc_ua_rts(),
.uirisc_ua_rx_data(tb_ua_dut_tx),
//.uirisc_ua_cts(1'b0),

//FLASH controller
.uorisc_fc_sclk(tb_fc_sclk_2),
.uiorisc_fc_MOSI(tb_fc_MOSI_2),
.uirisc_fc_MISO1(tb_fc_MISO1_2),
.uirisc_fc_MISO2(tb_fc_MISO2_2),
.uirisc_fc_MISO3(tb_fc_MISO3_2),
.uorisc_fc_ss(tb_fc_ss_2),

// System signal
.uirisc_clk_100mhz(tb_clk),
.uirisc_rst(tb_rst)
);
```

```
s25fl128s SPI_flash_2
(.SI(tb_fc_MOSI_2), //IO0
.SO(tb_fc_MISO1_2), //IO1
.SCK(tb_fc_sclk_2),
.CSNeg(tb_fc_ss_2),
.RSTNeg(tb_rst),
.WPNeg(tb_fc_MISO2_2), //IO2
.HOLDNeg(tb_fc_MISO3_2));  //IO3

//***********************************
//Clock waveform generation
initial tb_clk <= 1'b1;
always #25 tb_clk =~ tb_clk; //assume 20MHz

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Signals initialization.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//========================
//read memory to get instruction
initial begin
$readmemh(`TEST_CODE_PATH,tb_r32_pipeline.SPI_flash.Mem);
$readmemh(`EXC_HANDLER,tb_r32_pipeline.SPI_flash.Mem);
$readmemh(`TEST_CODE_PATH_2,tb_r32_pipeline.SPI_flash_2.Mem);
$readmemh(`EXC_HANDLER,tb_r32_pipeline.SPI_flash_2.Mem);
tb_rst = 1'b1;
repeat(1)@(posedge tb_clk);
tb_rst = 1'b0;
repeat(10000)@(posedge tb_clk);
tb_rst = 1'b1;
repeat(12000000)@(posedge tb_r32_pipeline.dut_c_risc.urisc_clk);

$stop;
end
endmodule
```

Chapter 7 Synthesis on FPGA

After the Simulation by using ModelSim simulator, the functionality of the Exception Scheme shown positive result. In this Chapter, we will discuss about synthesis the RISC32 processor on the ARTY Artix-7 FPGA Development Board by using Xilinx Vivado 2017.2. Besides, an experiment had conducted on the RISC32 processor by connecting two ARTY Artix-7 FPGA Development Board. The objective of this experiment is to test run and evaluation on the physical design for functional correctness.

7.1 Pin Allocation

The Xilinx Design Constraints (XDC) shown in Table 7.1. It has been set for the implementation of RISC32 processor on the ARTY Artix-7 FPGA Development Board.

Group	Design pin	Xilinx ARTY 4 DDR FPGA pin	Remark
Global	uirisc_clk_100mhz	E3	
	uirisc_rst	C2	
Quad SPI Flash Controller	uiorisc_fc_MOSI	K17	
	uirisc_fc_MISO1	K18	
	uirisc_fc_MISO2	L14	
	uirisc_fc_MISO3	M14	
	uorisc_fc_ss	L13	
SPI Controller	uiorisc_spi_miso	G1	ChipKit SPI
	uiorisc_spi_mosi	H1	
	uiorisc_spi_sclk	F1	
	uiorisc_spi_ss_n	C1	
UART Controller	uorisc_ua_tx_data	U16	
	uirisc_ua_rx_data	V15	
GPIO Controller	urisc_GPIO[0]	G13	Pmod Header JA
	urisc_GPIO[1]	B11	
	urisc_GPIO[2]	A11	
	urisc_GPIO[3]	D12	
	urisc_GPIO[4]	D13	
	urisc_GPIO[5]	B18	
	urisc_GPIO[6]	A18	
	urisc_GPIO[7]	K16	
	urisc_GPIO[8]	E15	Pmod Header JB
	urisc_GPIO[9]	E16	
	urisc_GPIO[10]	D15	
	urisc_GPIO[11]	C15	
	urisc_GPIO[12]	J17	
	urisc_GPIO[13]	J18	
	urisc_GPIO[14]	K15	
	urisc_GPIO[15]	J15	
	urisc_GPIO[16]	U12	Pmod Header JC
	urisc_GPIO[17]	T10	Connected to LED for Observation
	urisc_GPIO[18]	V10	Pmod Header JC
	urisc_GPIO[19]	V11	
	urisc_GPIO[20]	U14	

	urisc_GPIO[21]	V14	
	urisc_GPIO[22]	T13	
	urisc_GPIO[23]	U13	
	urisc_GPIO[24]	D4	Pmod Header JD
	urisc_GPIO[25]	D3	
	urisc_GPIO[26]	F4	
	urisc_GPIO[27]	F3	
	urisc_GPIO[28]	E2	
	urisc_GPIO[29]	D2	
	urisc_GPIO[30]	H2	
	urisc_GPIO[31]	G2	
Testing Pin	test_urisc_intr_spi	E1	Connected to led for observation
	test_urisc_intr_uart	G6	
	test_urisc_mc_syscall	J4	
	test_urisc_mc_undef_inst	J2	
	test_urisc_ex_ovfs	H6	
	test_urisc_IRQ	H5	
	test_urisc_mc_eret	J5	
	test_urisc_cp0_exc_flag	T9	

Table 7.1: Pin allocation on ARTY Artix-7 FPGA Development Board

7.2 Setting up the Test Environment for Functionality Test

The RISC32 IoT processor was synthesis on FPGA board and the experiment conducted by connect two FPGA board together. The Experiment set up shown as Figure 7.1. In Figure 7.1, there are one board label as Server and another label as Client. The UART and SPI for both FPGA board was connect to each other for data transmission. In order to increase observability, there are some internal pin was pull out to the top-layer and connected to LED and. The clock was lower down to 1k Hz. The Connected Pin function shown in Figure 7.2.

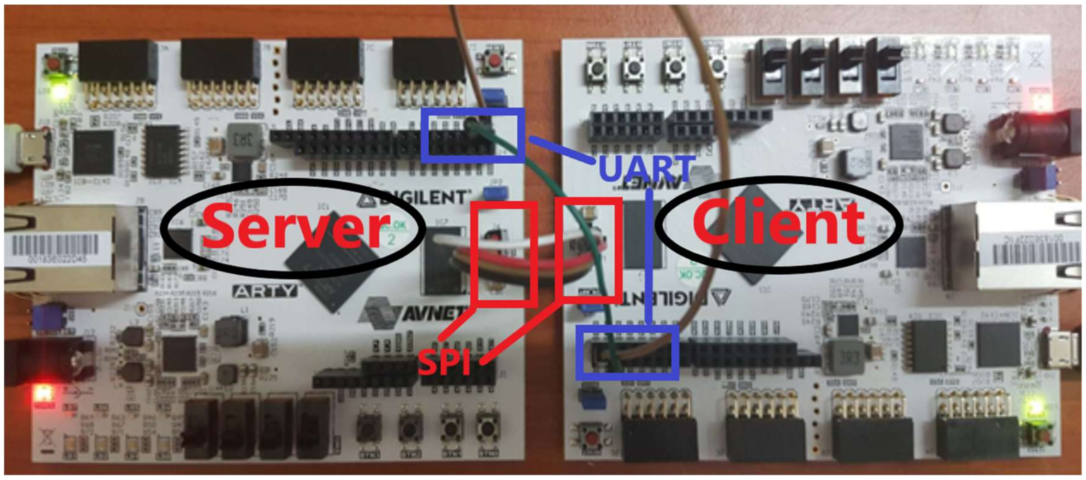


Figure 7.1 Test Environment set up.

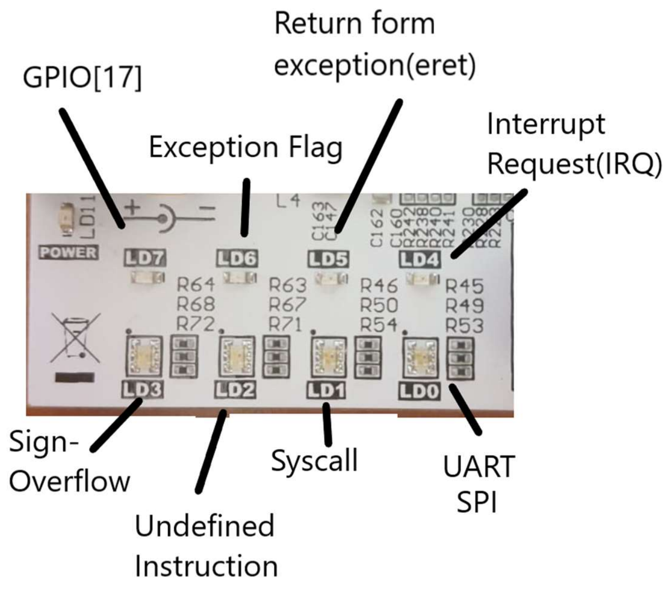


Figure 7.2 Functionality of the LED Pin Connected.

At the Server, the user program was programme to generate the Internal Exception Event (Trap) such as Sign-overflow (Detected at EX stage), Undefined Instruction (Detected at ID stage) and Syscall (Detected at ID stage). The Internal Exception Event will loop until the power off or reset button was press.

For the Client, the user program was programme to transmit the data to the Server through UART and SPI continuously. When the UART or SPI on Server board received a data, UART or SPI will generate an Interrupt Request on Server board and jump to the Exception handler.

When enter to the exception handler, the GPIO [17] will light up to indicate that the program execution was in the Exception handler. The GPIO [17] will turn off when it finished the Interrupt Service Routine and exit the exception handler return to the user program.

When there is the exception event occur, the LD6 will blinking, this is because the assertion of the Exception Flag for one clock cycle. In this Project, The exception event included the sign-overflow, undefined instruction, syscall, and interrupt request. The LD5 will blink when an "eret" instruction was decoded. "eret" is the only instruction that make the CPU return from Exception handler to the user program. It only appear in the last line of the Exception Handler. When LD5 blinking, we know that the exception event was served and ready return to user program.

The LD4 will blink when there are an Interrupt Request. In this experiment, client will transmit the data continuously. The LD0 in server board used to indicate that the data received from client. LD0 will turn to blue when SPI on server received a byte of data from the Client and it will turn to red when UART on server received a byte of data from the Client. LD0 will turn off when the Interrupt Flag de-asserted by the Exception handler.

The LD3 will blink when there are a sign-overflow exception event. The LD2 will blink when there are an undefined instruction exception event. The LD1 will blink when there are a syscall instruction.

Chapter 8 Conclusion & Future Enhancement

8.1 Conclusion

The exception handler scheme for interrupt conflicting and the nested interrupt resolution have successfully implemented into the RISC32 IoT processor. In this project, the exception events cover the sign-overflow, undefined instruction, syscall and external interrupt request. There are two communication I/O supported the interrupt, which are UART and SPI. The Priority interrupt Controller (PIC) is to handle the multiple interrupt occurrences based on priority level. It collaborate with coprocessor 0 to handle the exception event. With the availability of the well planning exception scheme, it is straightforward to resolve the conflicts among the exceptions event. In addition, it will be easier to integrate new devices without having to worry about exception handling.

In Chapter 2, the Exception scheme for ARM processor and MIPS have been review. In Chapter 3, we have discuss about the basic approach for this project. In Chapter 4, the system specification have discussed. In Chapter 5, we have perform the analysis on the I/O system. In addition**, the exception scheme for RISC32 IoT processor have developed** in chapter 5.

The **test bench has been model by using the VerilogHDL and simulated by using the ModelSim in order to verify the functionality of exception handle and Interrupt Service Routine (ISR) code**. The MIPS assembly code have coded to trigger each individual exception. The behaviour verification for exception handle scheme has been carry out by trigger different possible combination exception event to ensure the robustness of the exception scheme. The simulation result shown in Chapter 6.

Lastly, **the RISC32 IoT processor has successfully synthesized onto the ARTY Artix-7 FPGA Development Board by using Xilinx Vivado 2017.2**. An Experiment conducted to tests on the I/O function physically in order to ensure the functionality of the exception scheme.

8.2 Future Enhancement

In this project, the exception event only cover the sign-overflow, undefined instruction, syscall and interrupt request. For future development, more exception event should added such as breakpoint exception, address error exception and bus error exception in order to make the RISC32 IoT Processor more complete. The ISR code need to rewrite to make the user program more value added.

With the well-developed exception scheme for RISC32 IoT processor, the I/O system become more stable to use. It provided a high-confident level to integrated new I/O. For future, new I/O module can integrated to the RISC32 IoT processor such as Analog-to-Digital Convertor (ADC) and cryptography engine.

References

Altera. no date. Three Intel® Quartus® Prime Software Editions to Meet Your System Design Requirements. [Online]. [2 April 2018]. Available from: https://www.altera.com/downloads/download-center.html

Digilent. (n.d.). Arty A7: Artix-7 FPGA Development Board for Makers and Hobbyists. [online] Available at: https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/.

Hennessy, J.L.H & Patterson, D.A.P (2002). Computer Architecture: A Quantitative Approach. (3rd ed.). San Francisco: Morgan Kaufmann.

Kiat, W.P. (2018). The design of an FPGA-based processor with reconfigurable processor execution structure for internet of things (IoT) applications. Available at: http://eprints.utar.edu.my/3146/1/CEA-2019-1601206-1.pdf [Accessed 6 Apr. 2019].

Mentor Graphic. no date. ModelSim PE Student Edition. [Online]. [2 April 2018]. Available from: https://www.mentor.com/company/higher_ed/modelsim-student-edition

Mok, K.M (2009). Computer Organisation and Architecture Lecture Notes. Universiti Tunku Abdul Rahman, Faculty of Information and Communication Technology.

Opencores (2010). Wishbone B4. [Online]. [4 August 2018]. Available from: https://cdn.opencores.org/downloads/wbspec_b4.pdf

Patterson, D.A.P & Hennessy, J.L.H (2008). Computer Architecture and Design Computer Organization and Design: The Hardware/software Interface. (4th ed.). Canada: Morgan Kaufmann.

Sloss, A.N, Symes, D & Wright, C (2004). ARM System Developer's Guide : Designing and Optimizing System Software. : Morgan Kaufmann.

Sweetman, D.S (2007). See MIPS Run . (2nd ed.). United States: Morgan Kaufmann.

Synopsys. no date. VCS. [Online]. [2 April 2018]. Available from: https://www.synopsys.com/verification/simulation/vcs.html

Xilinx.com. (n.d.). Vivado Design Suite. [online] Available at: https://www.xilinx.com/products/design-tools/vivado.html#overview.

Zjueducn. no date. Lecture 4 for pipelining. [Online]. [2 April 2018]. Available from: http://arc.zju.edu.cn/static/download/arch/v4_arch_9.pdf