THE DESIGN AND DEVELOPMENT OF A PS/2 MOUSE CONTROLLER AND MULTIPLE I/O BUS SYSTEM INTEGRATION

BY

NG KWONG CHEONG

A REPORT

SUBMITTED TO

University Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology

Department of Information Technology and Engineering

APRIL 2014

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: **The Design and Development of a PS/2 Mouse Controller and Multiple I/O Bus System Integration**

Academic Session: 2014/2015

I NG KWONG CHEONG

(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:
32AA, Jalan Lagenda 14,Taman Lagenda Mas, Kajang, Selangor

Supervisor's name

Date: 4th April 2014

Date: 4th April 2014

DECLARATION OF ORIGINALITY

I declare that this report entitled "**The Design and Development of a PS/2 Mouse Controller and Multiple I/O Bus System Integration**" 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 : __NG KWONG CHEONG___

Date : _____

Acknowledgement

On behalf of this project, I would like to express my deepest gratitude to my project supervisor, Mr. Mok Kai Ming who has been providing me invaluable guidance and constructive suggestions during the planning and development of this project. His willingness to give his time so generously has been much appreciated.

I would also like to express my appreciation to my family members who have been giving me endless support and encouragement since the starting of my undergraduate years. In addition, I would like to thank my course mates and friends who supported me throughout the entire course of this project.

Once again, it is my honour to say that the accomplishment of this project is indeed, due to the heartfelt support rendered by the people I have mentioned above. Their help and guidance is very much appreciated.

Abstract

As stated in the project title, the main objective of this project involves the design and development of a PS/2 Mouse Controller and Multiple I/O Bus System Integration. To fulfil this purpose, the investigation and analysis on previous works done by the senior in the design of Mouse Controller was also involved. In addition, there was a necessity for the investigation and analysis of the PS/2 Controller interface which includes the PS/2 Controller design, physical interface, low – level protocol, modes of operation, commands and extensions.

The design of the Mouse Controller was completed by the previous student. Its design is based on the bidirectional synchronous serial communication protocol, whereby it contains two wires for communication purposes. One is responsible for transmitting data to the PS/2 Controller while the other generates the clock signal to specify the data sent is valid and therefore can be retrieved.

Despite that, the project was incomplete because the implementation of PS/2 Controller has not been done in previous student's work. Therefore, this project is initiated to continue the work of modelling the PS/2 Controller and ensuring the PS/2 Controller can communicate with the Mouse Controller designed by the previous student. Therefore, the emphasis of this project is to focus on verifying the correctness on the functional behaviour of the newly designed PS/2 Controller. Thus, the research and design methodologies done by the previous student were studied and practiced on this project.

Lastly, after the verification is done, the newly designed PS/2 Controller will be integrated into 32-bit RISC processor which is ready in the Faculty of Information Technology, UTAR. A test program will be developed to test the compatibility, functionality and behaviour for the new 32-bit RISC processor with the integrated PS/2 Controller.

Table of Contents

List of Figures

List of Tables

List of Flow Charts

Chapter 1: Introduction

1.1 Background

1.1.1 Design Background

The Personal System/2, also known as PS/2 was originally IBM's third generation of personal computers released in 1987. Its production line was created in an attempt to recapture control of the PC market by introducing an advanced yet proprietary architecture. Although the PS/2 line was unsuccessful with the consumer market, many of the PS/2's innovation, such as the 16550 UART, 1440KB 3.5-inch floppy disk format, 72 – pin SIMMs, the PS/2 keyboard and mouse ports, and the VGA video standard, went on to become standards in the broader PC market(Wikipedia, 2012). Nowadays, the term "PS/2" refers to the innovation of IBM's third generation of personal computer.

PS/2 is a type of serial communication which is specifically used for user input devices. The design involves a controller, the mechanical and electrical information of the communication, and a device (OSDev.org, 2012). In personal computing, it is generally used to connect the keyboard and mouse for data transmission between the devices and the host (CPU). The PS/2 system contains two wires for communication: one is for transmitting data in a serial stream, while the other is for the clock information to specify when the data is valid and can be retrieved (Chu, 2008). Similar to the PS/2 keyboard system, the PS/2 mouse implements a bidirectional synchronous serial protocol. When the data and clock are high, the bus is "idle". This is the only state where the keyboard and mouse are allowed to transmit data. The CPU has priority over the bus and may inhibit communication at any time by holding the clock low (Chapweske, 2003).

The data sent from the device to the host is read on the falling edge of the clock signal; data sent from the host to the device is read on the rising edge. The device always generates the clock signal for both direction of communication(Chapweske, 2003). Therefore, the host must first inhibit communication from the device by pulling the clock low in order to send data. This is called a "Request to Send" state. When the device detected it, it will start to generate clock pulses (Chapweske, 2003). The data sent is arranged in bytes, consisting of 1 start bit, 8 data bits, 1 parity bit and 1 stop bit (Chapweske, 2003).

1.1.2 Problem Background

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

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

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

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

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

1.2 Problem Statement

So far, there is MIPS compatible ISA which includes the PS/2 mouse system, PS/2 keyboard system, basic memory, coprocessor 0 (CP0) and a Universal Asynchronous Receiver/Transmitter (UART) in contribution to the development of 32-bit RISC project. The functionality of the PS/2 keyboard system has been tested and verified. However, the existing PS/2 mouse system is incomplete as the PS/2 Controller that is attached to the CPU has not been developed yet,

causing the verification and testing of a fully functional PS/2 mouse system cannot be done. The existing PS/2 mouse system is designed to be integrated to a point-to-point bus system to the CPU, however modifications are need to be made to convert it into a shared bus system which supports multiple I/O's which in this case, the PS/2 mouse and PS/2 keyboard using bus arbitration.

1.3 Project Scope

The scope of this project includes the development of the PS/2 Controller which includes its specifications at architecture level and microarchitecture level. Apart from that, the modeling of the PS/2 Controller will be constructed using Verilog HDL (Hardware Descriptive Language).

The functional behavior verification of the PS/2 Controller will be done using functional verification techniques involving the construction of testbench based on bus functional model. Apart from that, the device will be tested to communicate with the readily available Mouse Controller. Lastly, the PS/2 Controller will be integrated to the bus system to implement the PS/2 mouse system into the 32-bit RISC processor. In addition, the Interrupt Service Routine (ISR) for the PS/2 Controller will be developed using MIPS assembly language.

1.4 Project Objectives

The objectives of the project are as follows:

- To analyse the specification of the PS/2 architecture and fulfilling its design requirements.
- To develop the RTL model of the PS/2 Controller, including the development of chip specification, microarchitecture specification and Verilog HDL
- To develop a verification strategy based on a bus functional model to perform functional verification on the PS/2 Controller.
- To integrate the PS/2 Controller into the bus system using the Wish Bone Master-to-Slave connection architecture.
- To write an Interrupt Service Routine (ISR) using MIPS Assembly Language for the PS/2 Controller.

1.5 Impact and Significance

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

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

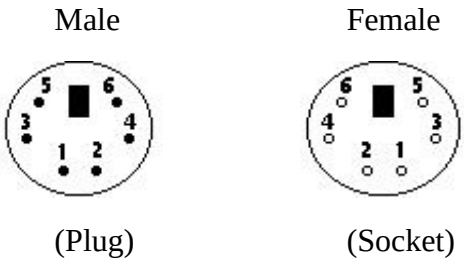
With the available well – developed basic 32-bit RISC RTL model (which has been fully functional verified), the verification environment and the design documents, researchers can develop their own specific RTL model as part of the development environment (whether directly modifying the internals of the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

Chapter 2: Literature Review

2.1 PS/2 Port and I/O Device (Mouse)

The PS/2 (Personal System/2) device interface was developed by IBM. It was released to the consumer market in 1987 and originally appeared in the IBM Technical Reference Manual (Chapweske, 2003).

The output pins for the PS/2 connector are shown below:



6-pin Mini-DIN (PS/2):
1 - Data 2 - Not Implemented
3 - Ground
4 - +5v 5 - Clock
6 - Not Implemented

Figure 2.1: The pins for PS/2 connector (Chapweske, 2003)

2.2 PS/2 Architecture

2.2.1 Power – On Reset

Mouse controller uses Power – On Reset architecture. When the power is on, it will trigger the reset pin and the Mouse Controller will reset the device. (Johnson, 1998)

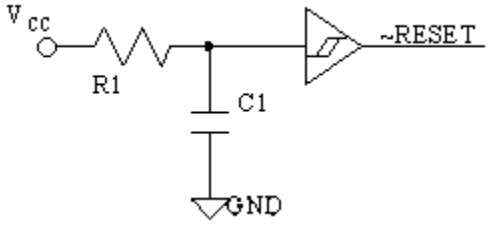


Figure 2.2.1: The Power – On Reset (Johnson, 1998)

2.2.2 Open Collector Interface

There are four pins on the connectors: Data, Ground, +5V, and Clock. The +5V is supplied by the PS/2 Controller (host) and the Ground is connected to the PS/2 Controller's electrical ground. The Data and Clock pins are both based on open collector architecture, which means they are normally held at a high logic level (logic 1) but can be easily pulled down to ground (logic 0). When the PS/2 Controller connects to a PS/2 device (Mouse or Keyboard), it should have large pull-up resistors on the Clock and Data lines. When pulling the line low, it will produce logic "0". When releasing the line, logic "1" will be produced instead. (Chapweske, 2003)

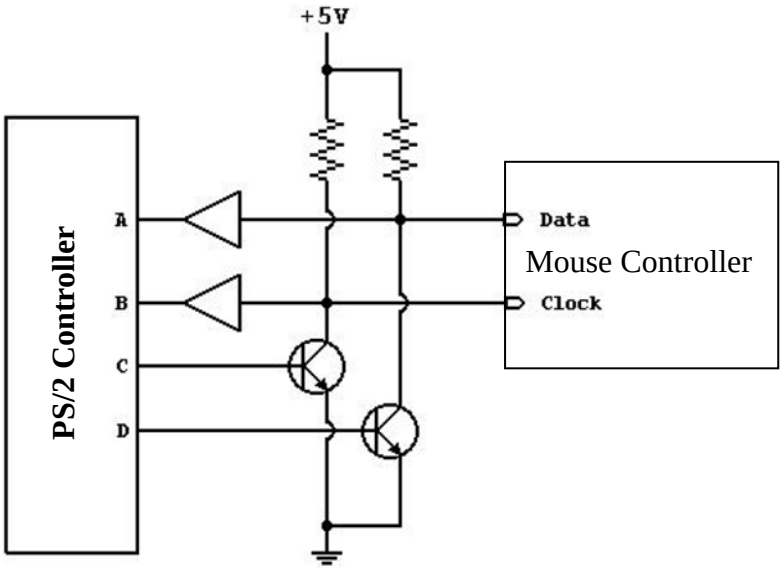


Figure 2.2.2: The Open Collector for Data and Clock Pins

2.3 Mouse Controller sends information to PS/2 Controller

The Mouse Controller always transmits data in 3 frames, which is 1 transaction, each. Each frame contains 1 start bit, 8 data bits, 1 parity bit and 1 stop bit. The transaction is shown below:

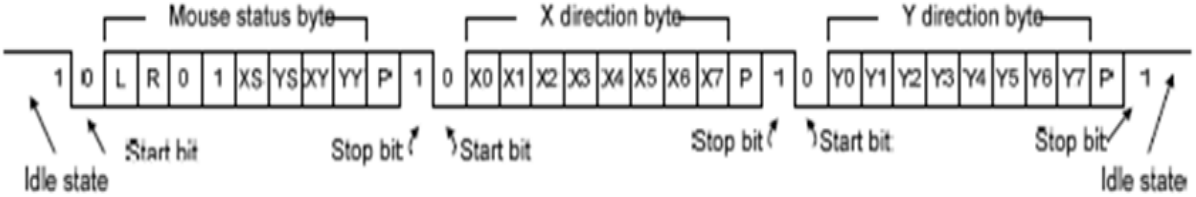


Figure 2.3.1: Mouse Controller data transmit format

The timing requirement of the Mouse Controller sends 1 frame of the packet to the PS/2 Controller is shown below:

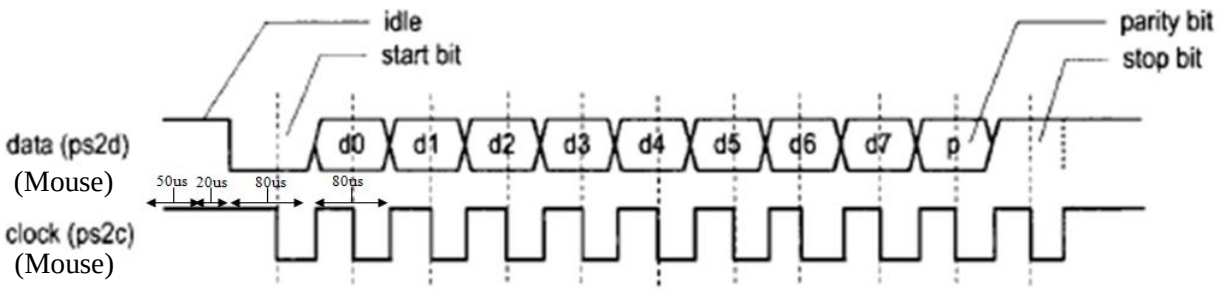


Figure 2.3.2: Timing requirement for Mouse Controller to PS/2 Controller

Mouse Controller is designed to detect two – dimensional motion on a surface. It measures the relative distance of movement and checks the status of the button. The Mouse Controller has three packets of information to be sent to the PS/2 Controller, whereby it transmits the packets continuously in a predesigned sampling rate.

The Mouse Controller always generates or toggles the Clock signal. When not toggling the Clock signal, it will remain high (weak pull-up). The Clock signal is bidirectional, which means the Mouse Controller can send the Clock signal through Clock line to the PS/2 Controller and PS/2 Controller can pull the Clock line too to send data to Mouse Controller. When the Data and Clock signal are both high, it is in “idle” state. This is the only state which the Mouse Controller is allowed to perform data transmission (Chapweske, 2003).

If the Mouse Controller wants to send data, it must make sure the Data and Clock signal are high (idle state). If the condition is met, it will first send the start bit (0). After that, the PS/2 Controller will read the data at the falling edge. The 8 data bits and 1 parity bit will be sent at the rising edge of the Clock signal. Once these 9 bits of data is sent, the Mouse Controller will send the following stop bit (1) to the PS/2 Controller.

All data is transmitted one byte at a line and each byte is sent in a frame consisting of 11 bits. These bits are:

- 1 start bit. This bit is always logic “0”.
- 8 data bits. They are sent in Little-Endian format (LSB is sent first).

- 1 parity bit (odd parity). It should be logic “1”. If not, error occurs.
- 1 stop bit. This bit is always logic “1”.

The parity bit is set (1) if there is an even number of 1’s in the data bits and reset (0) if there is an odd number of 1’s in the data bits. The numbers of 1’s in the data bits plus the parity bit are always added up to an odd number (odd parity). This is used for error detection. The PS/2 Controller must check this bit and if incorrect it should respond as it had received an invalid command.

Data is sent from the PS/2 Device (Mouse) to the PS/2 Controller (host) on the rising edge of the clock signal and PS/2 Controller reads the data on the falling edge of the clock signal. The clock frequency must be within the range of 10 – 16.7 kHz. In other words, the Clock must be high for 30 – 50 microseconds and low for 30 – 50 microseconds. In this design, the clock frequency is 12.5 kHz, which is high for 40 microseconds and low for 40 microseconds. The Mouse Controller always generates the clock signal, but the PS/2 Controller always has the ultimate control over communication.

The Mouse Controller sends out data packet to PS/2 Controller in the following format:

	MSB							**LSB**
Bit	7	6	5	4	3	2	1	0
1st Byte	Yo	Xo	Ys	Xs	1	M	R	L
2nd Byte	X7	X6	X5	X4	X3	X2	X1	X0
3rd Byte	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

L = Left Key Status Bit

(1 = Pressed; 0 = Released)

M = Middle Key Status Bit

(Reserved for standard Mouse Controller which has three button mouse)

(1 = Pressed; 0 = Released)

R = Right Key Status Bit

(1 = Pressed; 0 = Released)

X7 – X0 = Moving distance of X in two complements format

(Moving Left = Negative; Moving Right = Positive)

Y7 – Y0 = Moving distance of Y in two complements format

(Moving Down = Negative, Moving Up = Positive)

Xo = X Data Overflow bit (1 = Overflow)

Yo = Y Data Overflow bit (1 = Overflow)

Xs = X Data sign bit (1 = Negative)

Ys = Y Data sign bit (1 = Negative)

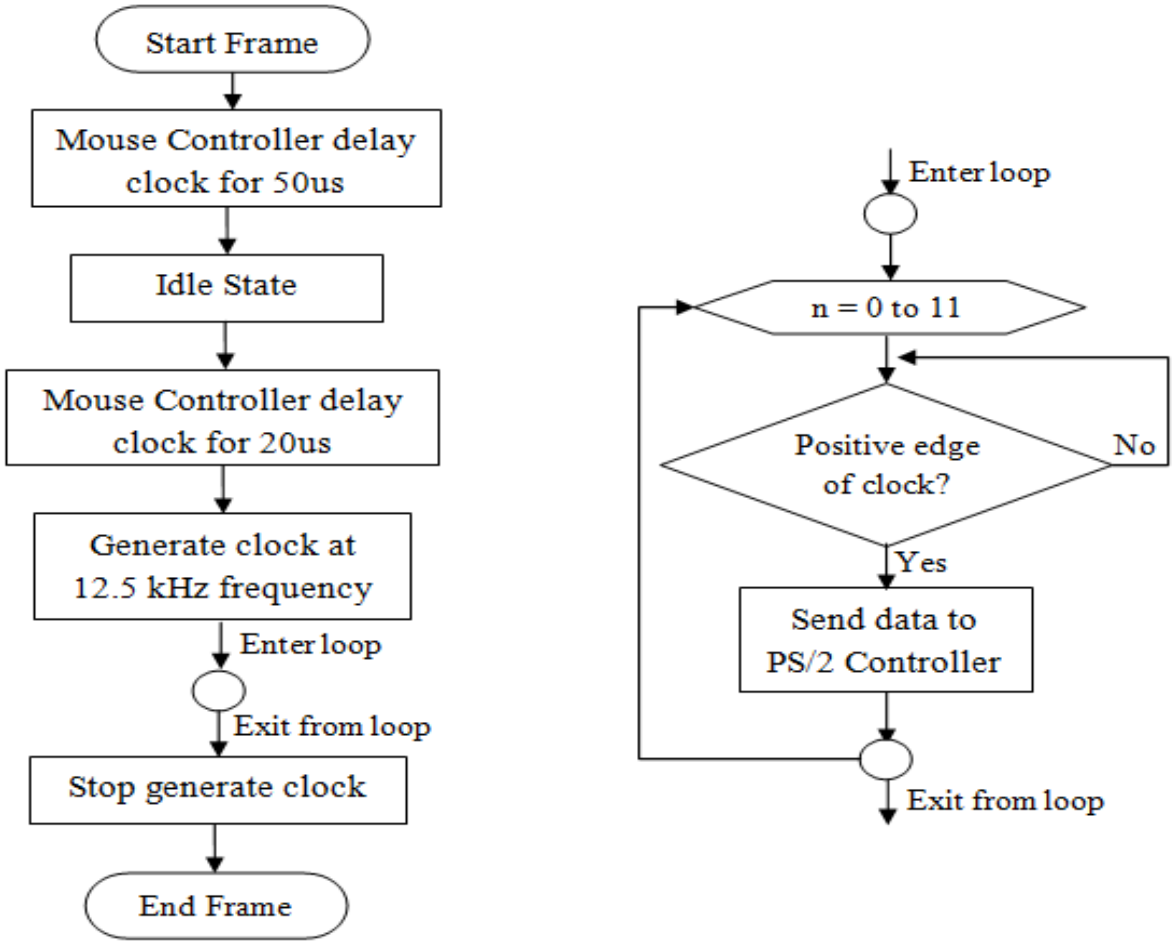
Table 2.3.1: Mouse Controller Data Packet Format

Code (Hex)	Command
FA	Acknowledge
AA	Self-Test passed
FC	Self-Test failed or Error
00	Device ID. Sent immediately after the Self-Test status command and is always 0x00

Table 2.3.2: Mouse Controller to PS/2 Controller data

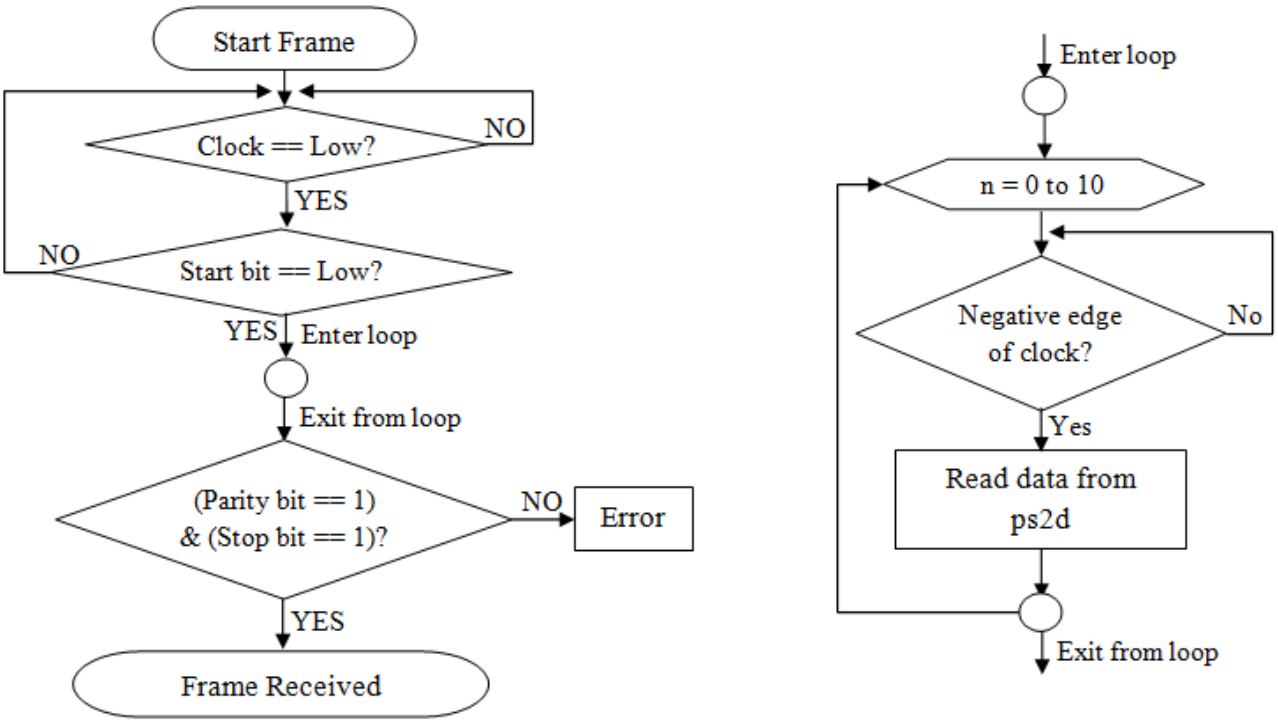
The flow charts of data transmission from Mouse Controller to PS/2 Controller are shown below:

On Mouse Controller side,



Flow Chart 2.3.1: Mouse Controller sends information to PS/2 Controller

On the PS/2 Controller side,



Flow Chart 2.3.2: PS/2 Controller receives data from Mouse Controller

The flow chart above only shows 1 transaction of 1 frame of data sent from the Mouse Controller to the PS/2 Controller, and there are 3 frames of data (1 packet) will be sent to PS/2 Controller, so the actual transmission process runs 3 times.

2.4 PS/2 Controller sends information to Mouse Controller

The timing requirement of PS/2 Controller sends 1 frame of the packet to Mouse Controller is shown below:

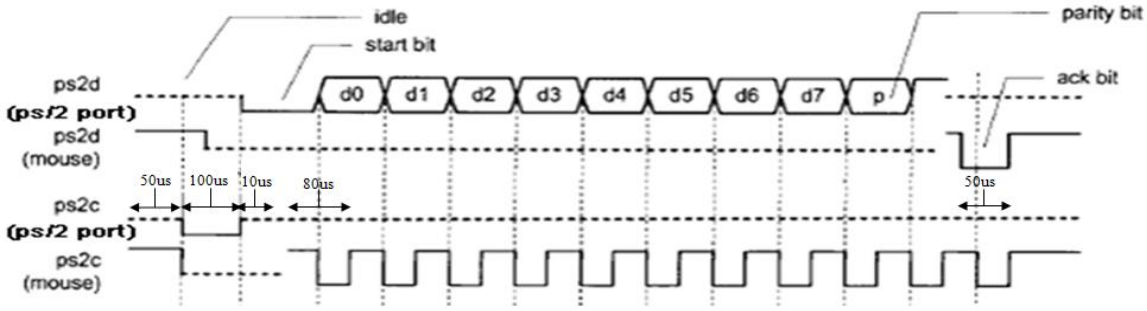


Figure 2.4.1: Timing Requirement for PS/2 Controller to Mouse Controller

The Mouse Controller is a bidirectional device. Thus, it can send data transmit data to PS/2 Controller and vice versa. In addition, the Mouse Controller always generates the Clock signal, but the PS/2 Controller has the ultimate control over the communication. When PS/2 Controller wishes to send data to Mouse Controller, it will control the Clock signal to low to reset the Clock signal and send data to Mouse Controller at the falling edge of the Clock signal. When the PS/2 Controller pulls the Clock signal to low and then pulls the Data line to low and releases the Clock line, it is a "Request to Send" state.

When the Mouse Controller detected the "Request to Send" state, it will begin toggling the Clock signal to receive and read the data from the PS/2 Controller. Differ to the Mouse Controller, the PS/2 Controller sends data at falling edge of the Clock signal, while the Mouse Controller reads the data at rising edge of the Clock signal.

All data is transmitted one byte at a time and each byte is sent in a frame consisting of 12 bits. These bits are:

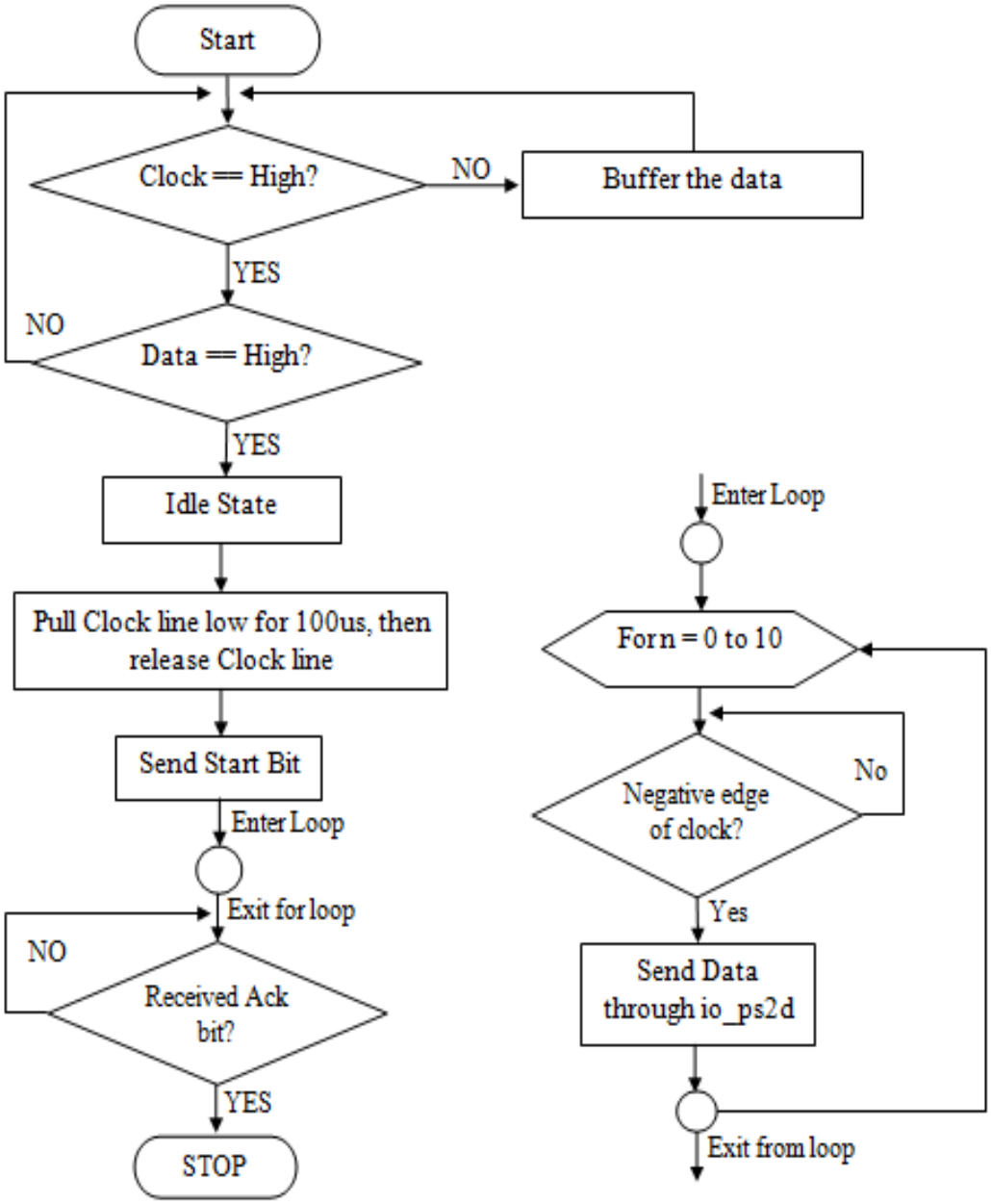
- 1 start bit. This bit is always logic "0".
- 8 data bits. They are sent in Little-Endian format (LSB is sent first).
- 1 parity bit (odd parity). It should be logic "1". If not, error occurs.
- 1 stop bit. This bit is always logic "1".
- 1 acknowledge bit.

The parity bit is set (1) if there is an even number of 1's in the data bits and reset (0) if there is an odd number of 1's in the data bits. The numbers of 1's in the data bits plus the parity bit are always added up to an odd number (odd parity). This is used for error detection. The Mouse Controller must check this bit and if incorrect it should respond as it had received an invalid command.

The clock frequency generated by the Mouse Controller must be in the range of 10 – 16.7 kHz. In other words, the clock must be high for 30 – 50 microseconds and low for 30 – 50 microseconds. In this design, the clock frequency is 12.5 kHz, which is high for 40 microseconds and low for 40 microseconds. As mentioned, the Mouse Controller always generates the clock signal, but the PS/2 Controller always has the ultimate control over communication.

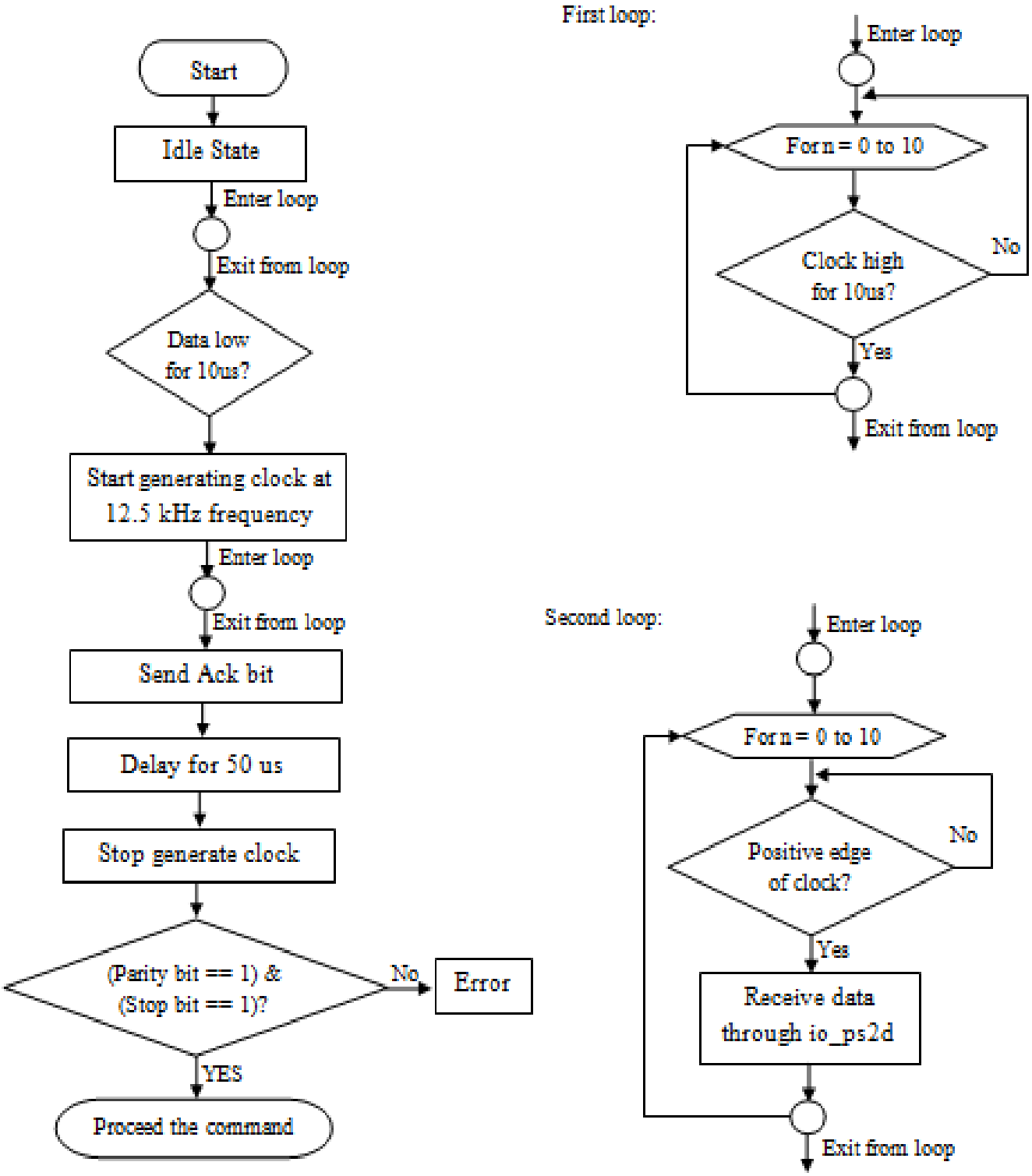
The flow charts of data transmission from the PS/2 Controller to Mouse Controller are shown below:

On PS/2 Controller side,



Flow Chart 2.4.1: PS/2 Controller sends information to Mouse Controller

On Mouse Controller side,



Flow Chart 2.4.2: Mouse Controller generates the Clock signal and receives information from PS/2 Controller

Table below shows the list of command PS/2 Controller sends to Mouse Controller

Code (Hex)	Command
FF	Reset. The Mouse Controller responds to this command with "Acknowledge" (0xFA) then enters Reset Mode.
FE	Resend. The PS/2 Controller sends this command whenever it receives invalid data from the Mouse Controller. The Mouse Controller responds by resending the last packet it sent to the PS/2 Controller. If the Mouse Controller responds to the "Resend" command with another invalid packet, the PS/2 Controller may either issues another "Resend" command or an "Error" command, cycles the Mouse Controller's power supply to reset the Mouse Controller. It may even inhibit communication (by pulling the Clock line low). The action depends on the PS/2 Controller.
F6	Set default. The Mouse Controller responds with "Acknowledge" (0xFA) then loads the following values: sampling rate = 100, resolution = 4 counts/mm, scaling = 1:1, disable data reporting. The Mouse Controller then resets its movement counters and enters Stream mode.
F5	Disable data report. The Mouse Controller responds with "Acknowledge" (0xFA) then disables data reporting and resets its movement counters. This only affects data reporting in Stream mode and does not disable sampling. Disabled Stream mode functions the same as Remote mode.
F4	Enable data report. The Mouse Controller responds with "Acknowledge" (0xFA) then enables data reporting and resets its movement counters. This command may be issued when the Mouse Controller is in Remote mode (or Stream mode), but it only affects data reporting in Stream mode.
F3	Set Sample rate. The Mouse Controller responds with "Acknowledge" (0xFA) then reads one more byte from the host. The Mouse Controller saves this byte as the new sample rate. After receiving the sample rate, it again responds with "Acknowledge" (0xFA) and resets its movement counters. Valid sample rates are 10, 20, 40, 60, 80, 100 and 200 samples/sec.

<table>
<tr><td>F2</td><td>Get device ID. The Mouse Controller responds with “Acknowledge” (0xFA) followed by its device ID (0x00 for the Mouse Controller). It should also reset its movement counters.</td></tr>
<tr><td>F0</td><td>Set Remote mode. The Mouse Controller responds with “Acknowledge” (0xFA) then resets its movement counters and enters Remote mode.</td></tr>
<tr><td>EE</td><td>Set Wrap mode. The Mouse Controller responds with “Acknowledge” (0xFA) then resets its movement counters and enters Wrap mode.</td></tr>
<tr><td>EC</td><td>Reset Wrap mode. The Mouse Controller responds with “Acknowledge” (0xFA) then resets its movement counters and enters the mode it was before Wrap mode (Stream mode or Remote mode).</td></tr>
<tr><td>EB</td><td>Read data. The Mouse Controller responds with “Acknowledge” (0xFA) then sends a movement data packet. This is the only way to read data in Remote mode. After the data packets have been successfully sent, it resets its movement counters.</td></tr>
<tr><td>EA</td><td>Set Stream mode. The Mouse Controller responds with “Acknowledge” (0xFA) then resets its movement counters and enters Stream mode.</td></tr>
<tr><td>E9</td><td>Status Request. The Mouse Controller responds with “Acknowledge” (0xFA) then sends the following 3 – byte status packet (then resets its movement counters).
<table>
<tr><td>Bit</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1st Byte</td><td>0</td><td>Mode</td><td>Enable</td><td>Scaling</td><td>0</td><td>Middle</td><td>Right</td><td>Left</td></tr>
<tr><td>2nd Byte</td><td colspan="8">Resolution</td></tr>
<tr><td>3rd Byte</td><td colspan="8">Sample Rate</td></tr>
</table>

Left, Right, Middle: 1 = Pressed, 0 = Released
Scaling: 1 = Scaling is 2:1, 0 = Scaling is 1:1
Enable: 1 = Enable data report, 0 = Disable data report</td></tr>
</table>

<table>
<tr><td></td><td>Mode: 1 = In Remote mode, 0 = In Stream mode
0xE9 only can be used in Remote mode or Stream mode</td></tr>
<tr><td>E8</td><td>Set resolution. The Mouse Controller responds with “Acknowledge” (0xFA) then reads one byte from the host and again responds with another “Acknowledge” signal before resetting its movement counters. The byte read from the host determines the resolution as the following values:
<table>
<tr><td>Value</td><td>Resolution</td></tr>
<tr><td>0x00</td><td>1 count /mm</td></tr>
<tr><td>0x01</td><td>2 counts /mm</td></tr>
<tr><td>0x02</td><td>4 counts /mm</td></tr>
<tr><td>0x03</td><td>8 counts /mm</td></tr>
</table></td></tr>
<tr><td>E7</td><td>Set scaling 2:1. The Mouse Controller responds with “Acknowledge” (0xFA) then enables 2:1 scaling.</td></tr>
<tr><td>E6</td><td>Set scaling 1:1. The Mouse Controller responds with “Acknowledge” (0xFA) then enables 1:1 scaling.</td></tr>
</table>

Table 2.4.1: Lists of command from PS/2 Controller to Mouse Controller

2.5 Operation Mode for Mouse Controller

Operation Mode	Description
Reset	The Mouse Controller enters Reset mode at power up or after receiving the 0xFF (Reset) command.
Stream	This is the default mode (after Reset mode) and is the mode in which most software uses the mouse. If the PS/2 Controller has previously set the Mouse Controller to Remote mode, it may re-enter Stream mode by sending the 0xEA (Set Stream mode) command to the Mouse Controller.
Remote	This is the mode where the Mouse Controller sends data packet (button + movement) to PS/2 Controller and may be entered by sending the 0xF0 (Set Remote mode) command to the Mouse Controller. In this mode, the data packet will only be received after

	request by sending the 0xEB (Read Data).
Wrap	This mode is not particularly useful except for testing the connection between the Mouse Controller and PS/2 Controller. Wrap mode may be entered by sending the 0xEE (Set Wrap mode) command to the Mouse Controller. To exit Wrap mode, the PS/2 Controller must issue the 0xFF (Reset) command or 0xEC (Reset Wrap mode) command. If the 0xFF (Reset) command is received, the Mouse Controller will enter Reset mode. If the 0xEC (Reset Wrap mode) command is received, the Mouse Controller will enter the mode it was prior to Wrap mode.

Table 2.5.1: The Operation Mode of the Mouse Controller

2.6 PS/2 Keyboard architecture

The PS/2 keyboard consists of a large matrix of keys, which are monitored by an on-board processor, known as the “keyboard encoder”. The encoder monitors which key(s) are being pressed or released and sends the data to the host (CPU). In addition, it takes care of all the de-bouncing and buffers any data in a 16-byte buffer. (Chapweske, 2003)

The keyboard’s processor is always busy monitoring the matrix of keys. If any key is being pressed, released, or held down, the keyboard will send a packet of information known as “scan codes” to the host. Scan codes can be categorized as two types: “make codes” and “break codes”. A make code is sent when a key is pressed or held down. A break code is sent when a key is released. As every key has its own unique make code and break code, the set of make and break codes for every key comprises a “scan code set”. According to the standard PS/2 protocol, there are three different scan code sets which is one, two, and three. By default, all modern keyboards follow scan code two. (Chapweske, 2003)

2.7 PS/2 Keyboard Scan Code

Keyboard is encoded by placing the key switches in a matrix of rows and columns. All rows and columns are periodically monitored by the keyboard encoder to detect any key state changes. If any state change is detected, the data is send serially to the PS/2 port from the keyboard using a

certain scan code. Each key has a unique scan code based on the key switch matrix row and column to identify the key pressed. Nevertheless, this depends on which scan code set the keyboard is following.

The type of key activities can be summarized as the following:

- When a key is pressed, the make code of the key is transmitted.
- When a key is pressed and held down continuously, known as being *typematic*, the make code is transmitted repeatedly at a specific rate. For every half a second, the keyboard transmits make code every 100ms after the key is held down.
- When a key is released, the break code is transmiited.

For example, when we press and release the 'A' key, the keyboard will transmit its make code and break code in the following fashion.

1C F0 1C

If we press and hold the key down for a while before releasing it, the make code will be transmitted repeatedly before the break code is sent.

1C 1C ... 1C F0 1C

Multiple keys can be pressed at the same time. An example would be pressing the shift key (make code is 0x12) and then the 'N' key, then releasing the 'N' key followed by the shift key. The transmitted code follows the make code and break code of the two keys:

12 31 F0 31 F0 12

Note: in this design there is no special code for lower or uppercase key.

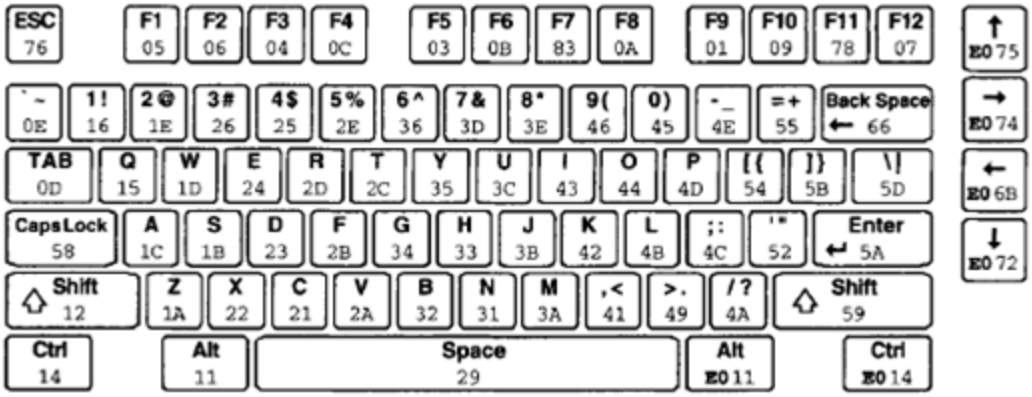


Figure 2.7: PS/2 keyboard scan code

Table below shows the structure of an IBM PS/2 keyboard scan code set 2.

KEY	MAKE	BREAK	---	KEY	MAKE	BREAK	---	KEY	MAKE	BREAK
A	1C	F0, 1C		9	46	F0, 46		[	54	F0, 54
B	32	F0, 32		`	0E	F0, 0E		INSERT	E0, 70	E0, F0, 70
C	21	F0, 21		-	4E	F0, 4E		HOME	E0, 6C	E0, F0, 6C
D	23	F0, 23		=	55	F0, 55		PG UP	E0, 7D	E0, F0, 7D
E	24	F0, 24		\	5D	F0, 5D		DELETE	E0, 71	E0, F0, 71
F	2B	F0, 2B		BKSP	66	F0, 66		END	E0, 69	E0, F0, 69
G	34	F0, 3B		SPACE	29	F0, 29		PG DN	E0, 7A	E0, F0, 7A
H	33	F0, 33		TAB	0D	F0, 0D		U ARROW	E0, 75	E0, F0, 75
I	43	F0, 43		CAPS	58	F0, 58		L ARROW	E0, 6B	E0, F0, 6B
J	3B	F0, 3B		L SHIFT	12	F0, 12		D ARROW	E0, 72	E0, F0, 72
K	42	F0, 42		L CTRL	14	F0, 14		R ARROW	E0, 74	E0, F0, 74
L	4B	F0, 4B		L GUI	E0, 1F	E0, F0, 1F		NUM	77	F0, 77
M	3A	F0, 3A		L ALT	11	F0, 11		KP /	E0, 4A	E0, F0, 4A
N	31	F0, 31		R SHIFT	59	F0, 59		KP *	7C	F0, 7C
O	44	F0, 44		R CTRL	E0, 14	E0, F0, 14		KP -	7B	F0, 7B
P	4D	F0, 4D		R GUI	E0, 27	E0, F0, 27		KP +	79	F0, 79
Q	15	F0, 15		R ALT	E0, 11	E0, F0, 11		KP EN	E0, 5A	E0, F0, 5A
R	2D	F0, 2D		APPS	E0, 2F	E0, F0, 2F		KP .	71	F0, 71
S	1B	F0, 1B		ENTER	5A	F0, 5A		KP 0	70	F0, 70
T	2C	F0, 2C		ESC	76	F0, 76		KP 1	69	F0, 69
U	3C	F0, 3C		F1	05	F0, 05		KP 2	72	F0, 72
V	2A	F0, 2A		F2	06	F0, 06		KP 3	7A	F0, 7A
W	1D	F0, 1D		F3	04	F0, 04		KP 4	6B	F0, 6B
X	22	F0, 22		F4	0C	F0, 0C		KP 5	73	F0, 73
Y	35	F0, 35		F5	03	F0, 03		KP 6	74	F0, 74
Z	1A	F0, 1A		F6	0B	F0, 0B		KP 7	6C	F0, 6C

0	45	F0, 45		F7	83	F0, 83		KP 8	75	F0, 75
1	16	F0, 16		F8	0A	F0, 0A		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	01	F0, 01		]	5B	F0, 5B
3	26	F0, 26		F10	09	F0, 09		;	4C	F0, 4C
4	25	F0, 25		F11	78	F0, 78		’	41	F0, 41
5	2E	F0, 2E		F2	07	F0, 07		,	52	F0, 52
6	36	F0, 36		PRNT SCRN	E0, 12, E0, 7C	E0, F0, 7C, E0, F0, 12		.	49	F0, 49
7	3D	F0, 3D		SCROLL	7E	F0, 7E		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	E1, 14, 77, E1, F0, 14, F0, 77	-NONE-				

Table 2.7: PS/2 Keyboard Scan Code Set 2

2.8 I/O Controller Interface with I/O Device

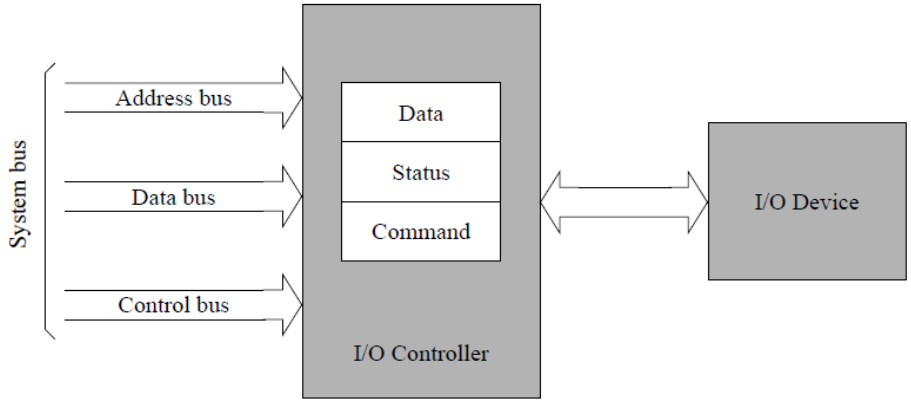


Figure 2.8.1: The Block Diagram of a Generic I/O Device Interface (Dandamudi, 2002)

Computer – use device are also called as peripheral device. There are two main purposes for I/O device: to communicate with the outside world and store data. Regardless of the intended purpose of the I/O device, all communications with these devices must involve the system bus. However, they are not directly connected to the system bus. Instead, there is usually an I/O Controller that functions as an interface between the system bus and the I/O device.

There are two main reasons for using an I/O controller. Firstly, different I/O devices exhibit different characteristics. If they are connected correctly, the CPU would have to understand and respond appropriately to each I/O device. This would cause the CPU to spend a lot of time interacting with each I/O device and spend less time executing user programs. Thus, the I/O controller is introduced to provide necessary low – level commands and data for proper operation of the associated I/O device. For complex I/O devices such as disk drives, there are special I/O control chips available.

Secondly, the amount of electrical power used to send signals on the system bus is very low. Thus, the cable connecting the I/O device has to be very short (a few centimeters at most). I/O controllers typically contain driver hardware to send current over long cables that connect the I/O devices. I/O controllers typically have three types of internal registers – a data register, a command register and a status register. When the CPU wants to interact with an I/O device, it communicates only with the associated I/O controller. (Dandamudi, 2002)

The interface of the PS/2 Controller connected to the Mouse Controller is shown as below:

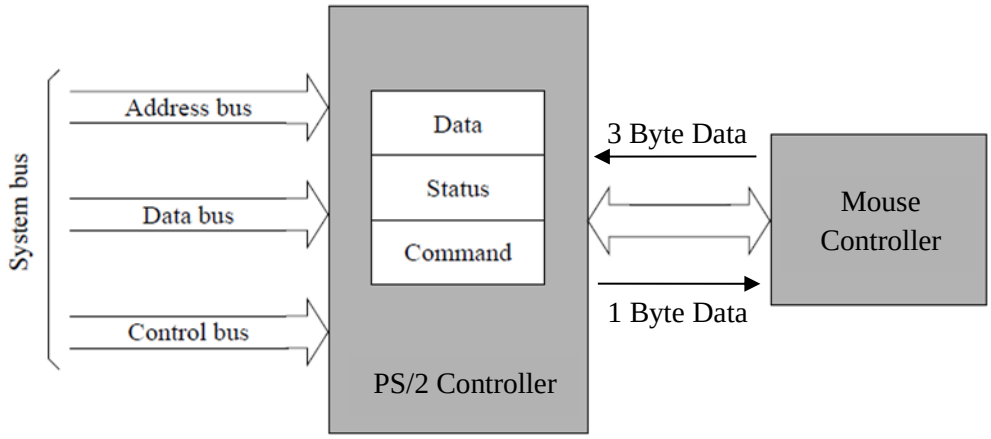


Figure 2.8.2: The Block Diagram of PS/2 Controller Interface with Mouse Controller

2.9 I/O Data Transfer

There are various ways I/O devices can be accessed by a system. The data transfer process involves two distinct phases, which are data transfer phase and the end –notification phase.
The data transfer phase transmits data between the memory and I/O device. This can be done by the programmed I/O or direct memory access (DMA). The end – notification informs the processor that the data transfer has been completed. The processor gets this information either by an interrupt or through the programmed I/O mechanism. (Dandamudi, 2002).

To understand I/O data transfer, we have to look at three basic techniques

i. Programmed I/O,
ii. Interrupt – driven I/O,
iii. DMA

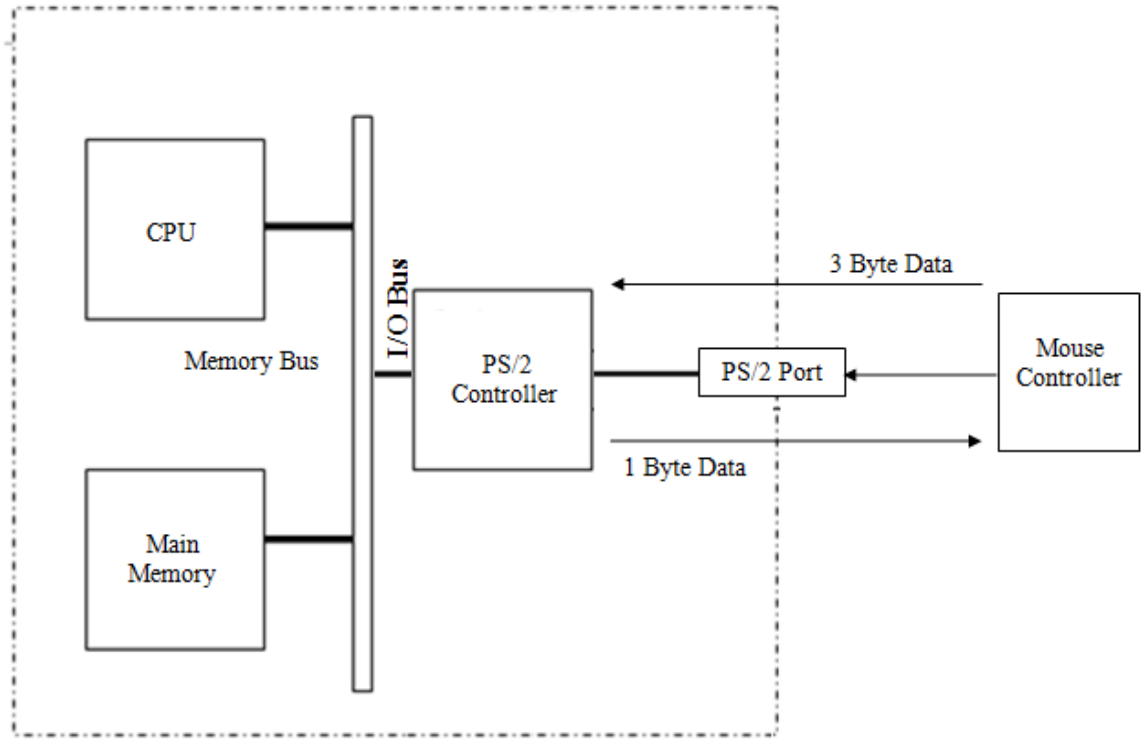


Figure 2.9.1: The Programmed I/O Design

i) Programmed I/O Mechanism

Programmed I/O involves the processor in the I/O data transfer. After the command is sent to the I/O devices, the processor will send signal to the I/O controller repeatedly to ensure whether the task has been completed or not. It repeatedly checks to confirm if a particular condition is true. Typically, it busy – waits until the condition is true. From this brief description, it is clear that the programmed I/O mechanism wastes processor time. (Dandamudi, 2002)

ii) Interrupt – driven I/O Mechanism

In the same situation, after the command is sent to the I/O devices, the processor assigns a task to an I/O controller and resumes its pending work. When the task is completed, the I/O controller notifies the processor by using an interrupt signal. Obviously, this is a better way of using the processor though an interrupt – driven mechanism requires hardware support, which is provided by all processors. (Dandamudi, 2002)

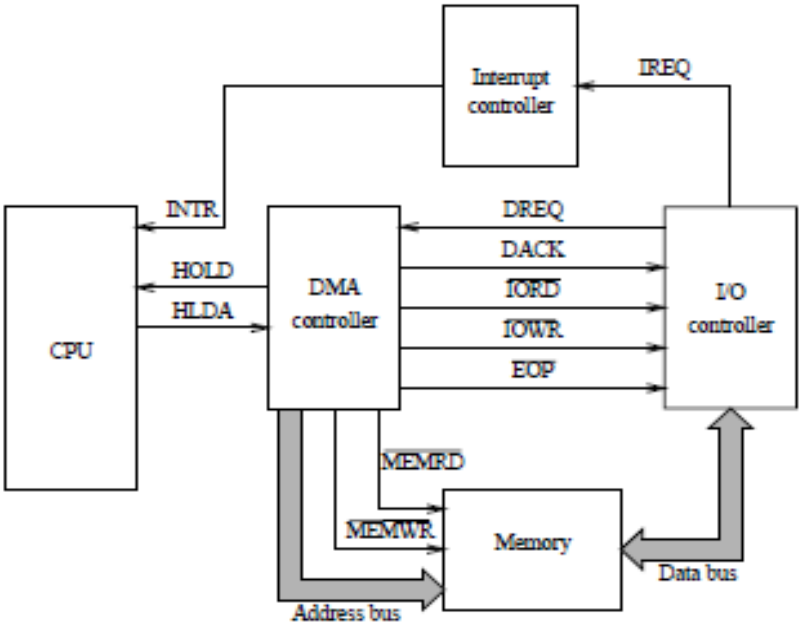


Figure 2.9.2: The DMA Design

iii) DMA

The last technique, DMA, relieves the processor of the low – level data transfer chore. DMA is used for bulk data transfer. For example, in an interrupt – driven I/O, the task assigned could be a DMA request to transfer data from a disk drive. Typically, a DMA controller oversees the data transfer. When the specified transfer is complete, the processor is notified by an interrupt signal. (Dandamudi, 2002)

DMA is implemented by using a DMA controller. The DMA controller acts as a slave to the processor and receives data transfer instructions from the processor. For example, to read a block of data from an I/O device, the CPU sends the I/O device number, main memory buffer address, number of bytes to be transferred, and the direction of transfer (I/O to memory or memory to I/O). After the DMA controller has received the transfer instruction, it requests the bus. Once the DMA controller becomes the bus master, it generates all bus control signals to facilitate the data transfer. The DMA transfer not only relieves the processor from the data transfer chore but also causes the data transfer process to be more efficient by transferring data directly from the I/O device to memory. (Dandamudi, 2002)

2.10 Bus System

Memories and input-output devices are usually interfaced to a microprocessor through a tri-state bus. To assure proper data transfer on the bus, the timing characteristics of both the microprocessor bus interface and the devices must be carefully considered. The bus system comprises of 3 major components:

- Data bus
- Address bus
- Control bus

Figure below shows a generalized I/O structure connecting CPU with other peripheral devices.

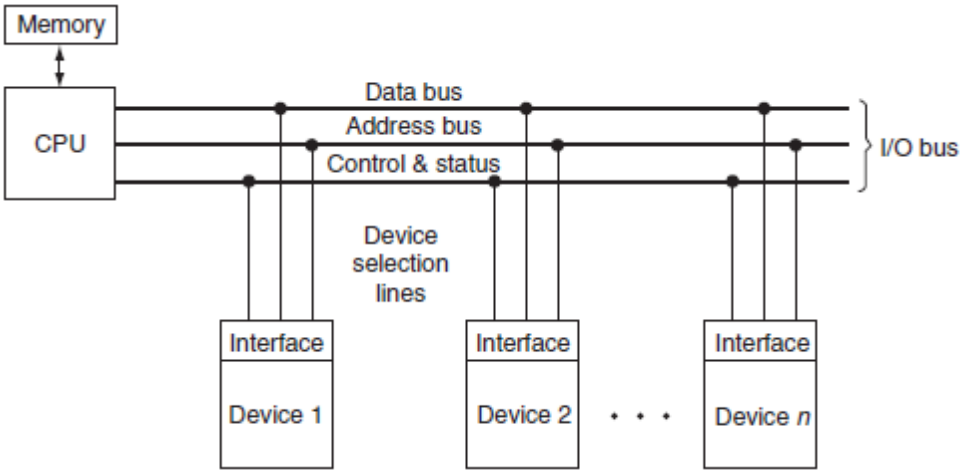


Figure 2.10: General I/O structure (Shiva, 2008)

Memories are usually interfaced to the CPU through a memory bus that consists of address, data, and control lines. It is also known as a "North Bridge". Other peripherals such as I/O devices communicate with the CPU over the I/O bus, which is also known as the "South Bridge". The data transfer rate of the memory bus is much higher than the I/O bus.

Each device has its own decoded address which is carried on the address bus. Hence, only the device whose address matches that on the address bus will participate in its corresponding I/O operation. The data bus is bidirectional. The control bus carries control signals such as READ,

WRITE, and so on. In addition, several status signals such as DEVICE BUSY and ERRORS originating in the device interface also form a part of the control bus (Shiva, 2008).

2.10.1 Bus Structures

In practice, a bus structure can be realized by using either tri-state buffers or multiplexers. A bus is called a tri-state bus when using tri-state buffers and a multiplexer-based bus when using multiplexers (Lin, 2011).

i) Tri-state Bus

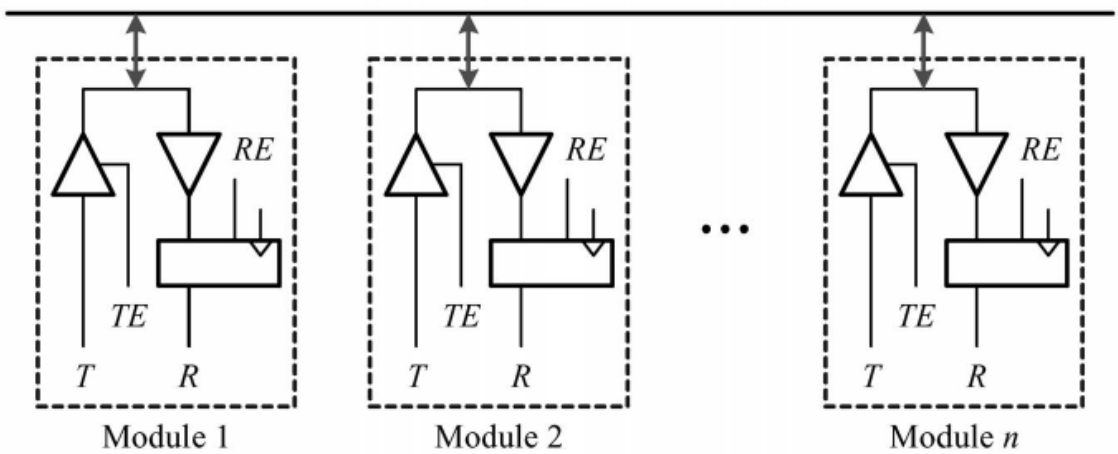


Figure 2.10.1.1: Typical Tri-state Bus Structure (Lin, 2011)

A typical tri-state bus is used in digital systems when there are n modules connected to the bus. Each module is connected to the bus through a bidirectional interface that enables it to drive a signal T to the bus when then transmit enable control signal TE is asserted. When the receive enable control signal RE is asserted, the module is enabled to sample a signal off the bus onto an internal signal R (Lin, 2011).

Using a bus structure may not be suitable for some applications, especially when the capacitive loading of the driver within the bidirectional interface of the active module is large. In the figure above, each transmit buffer needs to drive an amount of $n\ x\ (C_{bout} + C_{bin})$ capacitive load, where C_{bout} and C_{bin} are the output capacitance of the tri-state output buffer and the input

capacitance of the input buffer, respectively. This amount of capacitive load may be intolerant in some applications (Lin, 2011).

ii) Multiplexer-Based Bus

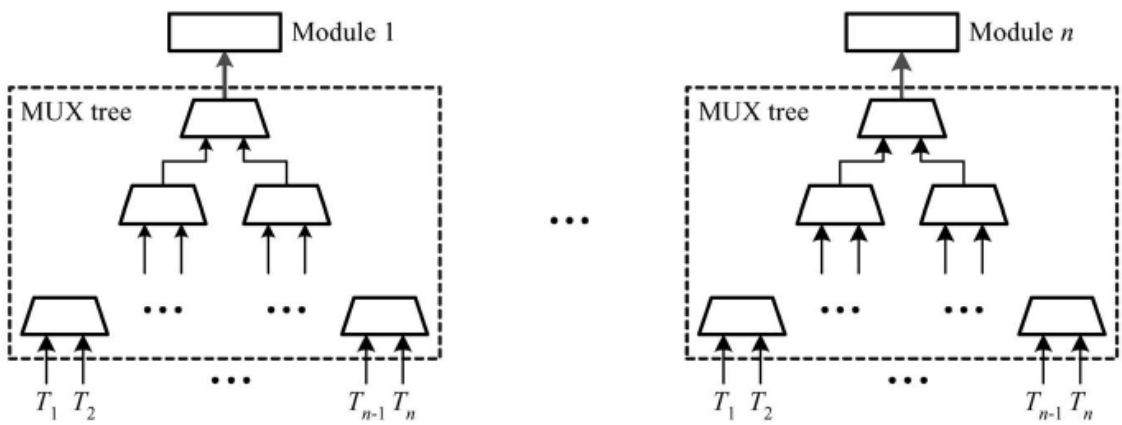


Figure 2.10.1.2: Typical multiplexer-based bus structure (Lin, 2011)

A multiplexer-based bus structure is used to avoid large amount of capacitive load in some applications. From the figure above, the output signals T_i of n modules are routed to their destination through a multiplexer tree. Compared to a tri-state bus structure, the propagation delay of a multiplexer tree is less when the number of modules attached to it is large enough. Thus, it is more often used instead of a tri-state bus for a better performance (Lin, 2011).

2.10.2 Bus System Interfacing

Typically, the bus system can be interfaced point-to-point with the other peripherals such as memory and I/O controllers. Figure below shows an example of a bus interface unit connected to an I/O system.

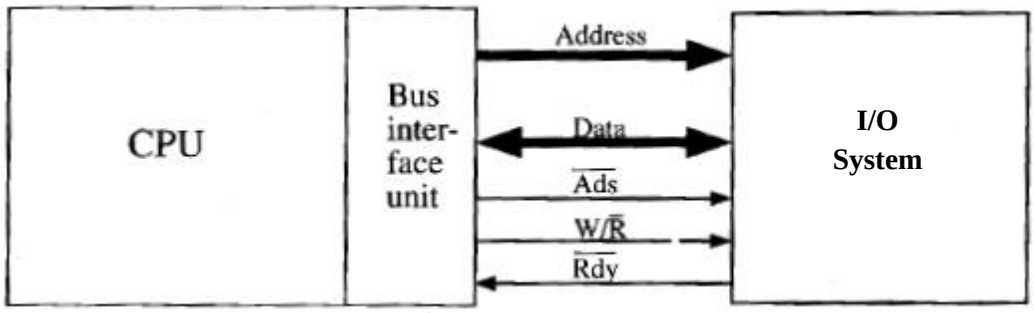


Figure 2.10.2.1: Bus interface connected to I/O system (Charles H. Roth, 1998)

The address bus signal is a unidirectional signal given from the bus interface to the I/O system, while the data bus signal is bidirectional. In addition, control lines such as Write/Read, Ready, and Address Strobe are also needed for basic read and write operation.

When the CPU wants to write to I/O system, it needs to follow a sequence of events:

1. The CPU outputs an address on the address bus and asserts $\overline{Ads}$ (address strobe) to indicate a valid address on the bus.
2. CPU places data on the data bus and asserts $W/\overline{R}$ (write/$\overline{read}$) to initiate writing the data. The I/O system asserts $\overline{Rdy}$ (ready) to indicate that the data transfer is ready

For reading, step (1) is the same, but in step (2) the I/O system places data on the data bus and are stored inside the CPU when the memory asserts $\overline{Rdy}$.

The interface signals for the bus interface unit are shown in figure 2.8.3 below. This is a simplified version of a complex bus interface unit, where only the signals needed to run the basic read and write bus cycles are included.

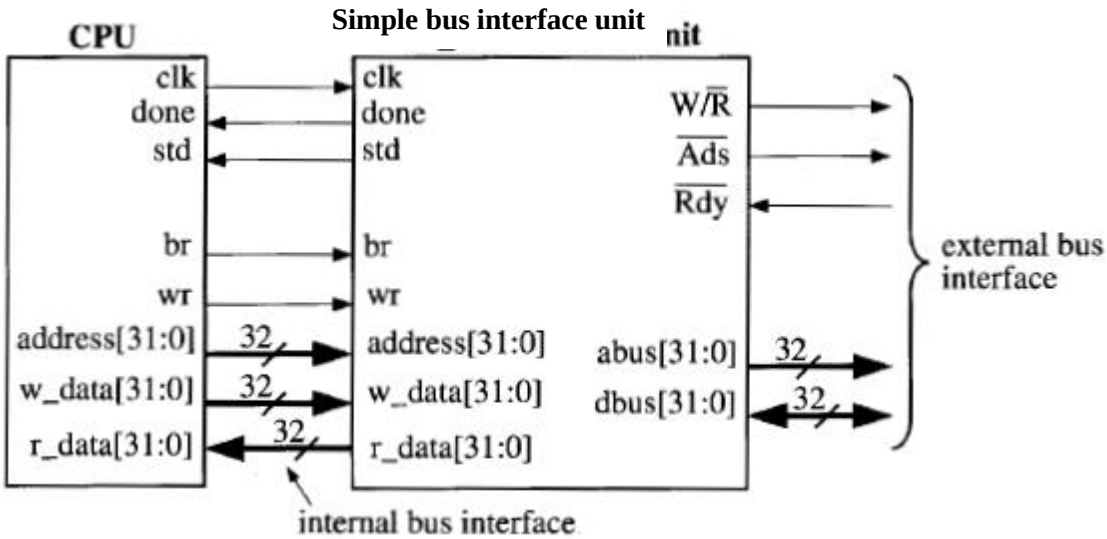
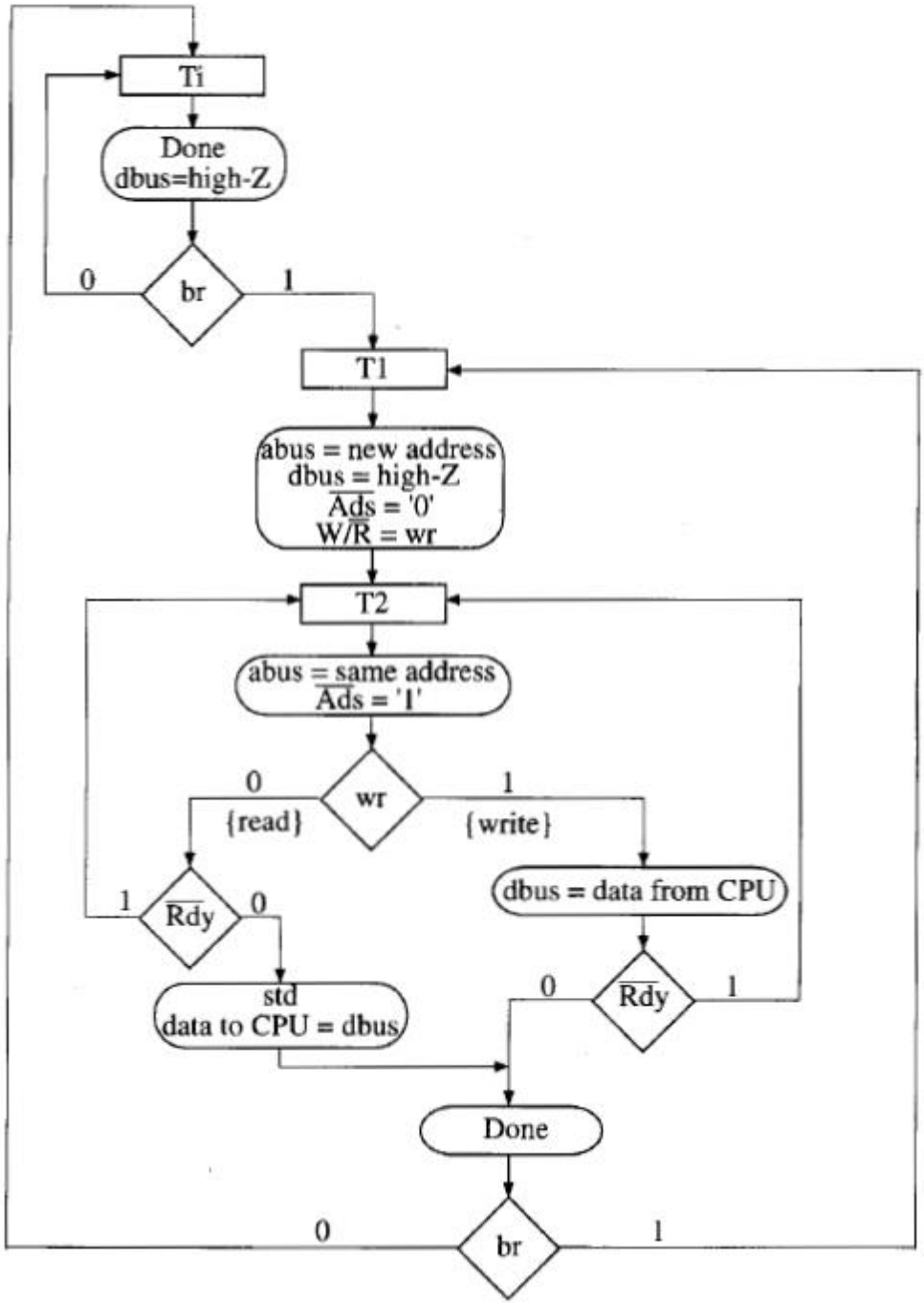


Figure 2.10.2.2: Simple bus interface unit interfaced with CPU. (Charles H. Roth, 1998)

The internal bus interface shows only those signals needed for transferring data between the bus interface unit and the CPU. If the CPU needs to write data to a memory attached to the external bus interface, it requests a write cycle by setting *br* (bus request) to 1 and *wr* to 1. If the CPU needs to read data, it requests a read cycle by setting *br* to 1 and *wr* to 0. Whe the write or read cycle is complete, the bus interface unit returns *done* to 1 to the CPU.

The state machine of the bus interface unit is shown in the flow chart below. In state Ti, the bus interface is in idle state, and the data bus is driven in high impedance state (high-Z). When a bus request (*br*) is received from the CPU, the controller goes to state T1. in T1, the new address is driven onto the address bus, and the address strobe signal, $\overline{Ads}$ is set to 0 to indicate a valid address on the bus, The write-read signal ($W/\bar{R}$) is set to low for a read cycle or high for a write cycle, and the controller goes to state T2. In T2, $\overline{Ads}$ returns to 1. For a read cycle, *wr* is set to 0 and the controller waits for the ready signal,$\overline{Rdy}$ to be 0, which indicates valid data is available from the memory, and then store data signal (*std*) is asserted to indicate that the data should be stored in the CPU. For a write cycle, *wr* is set to 1 and data from CPU is placed on the data bus. The controller then waits for $\overline{Rdy}$ to be 0 to indicate that the data has been in memory. The done signal, *done* is asserted when $\overline{Rdy}$ is set to 0 to indicate that the write or read cycle is completed. After the read or write cycle is completed, the controller goes back to Ti state if no bus request is

pending. Otherwise, it goes to state T1 to initiate another read or write cycle. The *done* signal remains on in Ti. (Charles H. Roth, 1998)



Flow Chart 2.10.2.1: State machine for simple bus interface unit. (Charles H. Roth, 1998)

2.11 WISHBONE Architecture

The WISHBONE System-on-Chip (SOC) Interconnection Architecture (generally known as WISHBONE) is a flexible design methodology for use with semiconductor intellectual property (IP) cores. It is used to foster design reuse by alleviating System-on-Chip integration problems by creating a common interface between IP cores. The advantages of this architecture include improving the portability and reliability of the system, and results in faster time-to-market for the end user (Peterson, 2010).

WISHBONE architecture was introduced as an intention as a general purpose interface. It defines the standard data exchange between IP core modules and does not regulate the application specific functions of the IP core. In fact, the WISHBONE architecture is analogous to a microcomputer bus in such that they both offer:

- A flexible integration solution that can be tailored to a specific application.
- A variety of bus cycles and data path widths to solve various system problems
- Allow products to be designed by a variety of suppliers (thereby driving down price while improving performance and quality).

However, traditional microcomputer buses are normally handicapped for use as a System-on-Chip interconnection as they are designed to drive long signal traces and connector systems which are highly inductive and capacitive. In contrast, WISHBONE architecture is much simpler and faster as well as having a rich set of interconnection resources, which do not exist in microcomputer buses (Peterson, 2010).

WISHBONE utilizes MASTER/SLAVE architecture. The functional modules with MASTER interfaces initiate data transactions to participating SLAVE interfaces. An idea of the communication between the MASTERs and SLAVEs is shown in the figure below.

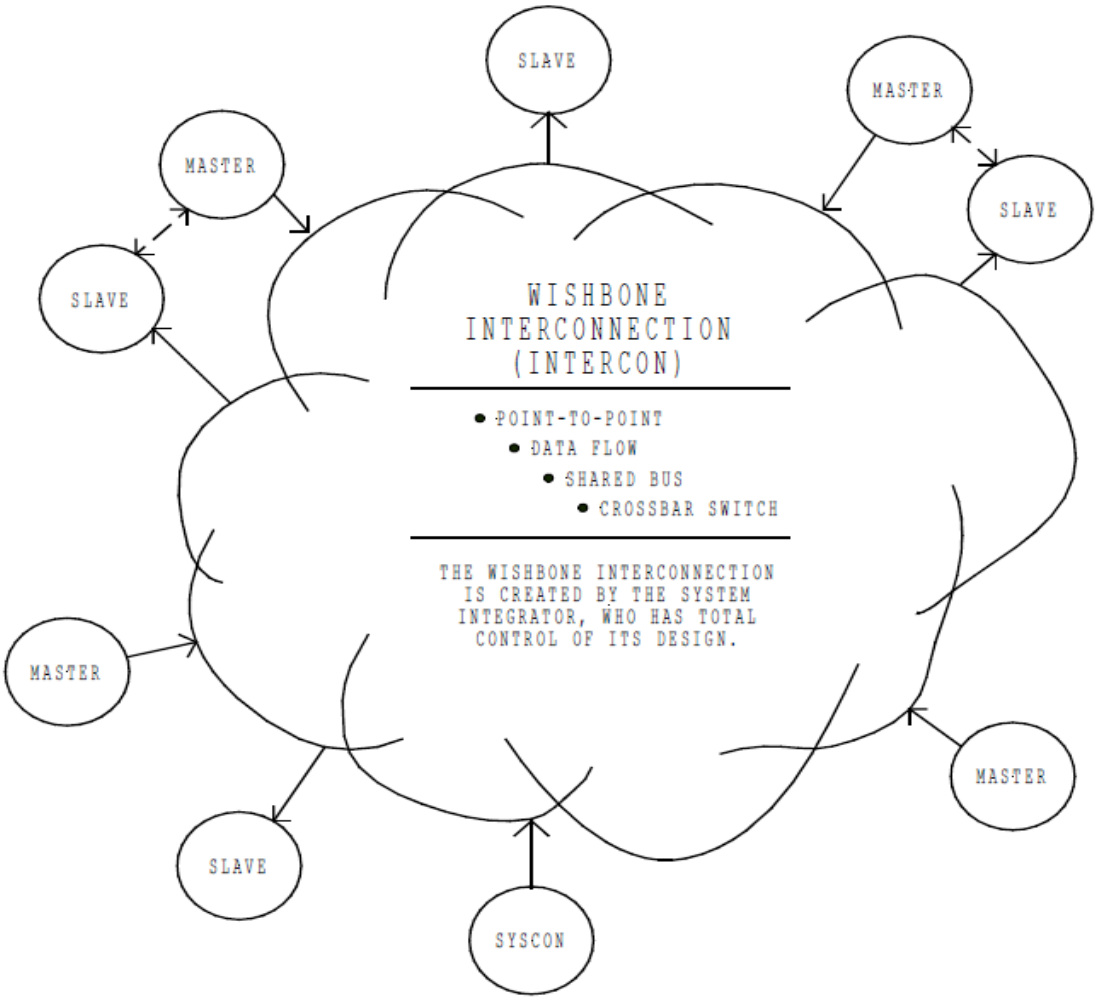


Figure 2.11: The WISHBONE MASTER/SLAVE interconnection (Peterson, 2010)

The MASTERs and SLAVEs usually communicate with each other through an interconnection (INTERCON), which is conceptualized as a 'cloud' that contains circuits. Different from traditional microcomputer buses, WISHBONE uses variable interconnection, a new scheme that allows the interconnection network to be changed by the system integrator to suit one's own requirements. This is possible because integrated circuit (IC) chips have interconnection paths that can be adjusted. These are very flexible, and take the form of logic gates and routing paths. These can be 'programmed' into the chip using variety of tools such as using hardware descriptive languages like Verilog or VHDL (Peterson, 2010).

2.11.1 Types of WISHBONE interconnection

There are 4 defined types of WISHBONE interconnection. They include:

i. Point-to-point
ii. Data flow
iii. Shared bus
iv. Crossbar switch

i) Point-to-point Interconnection

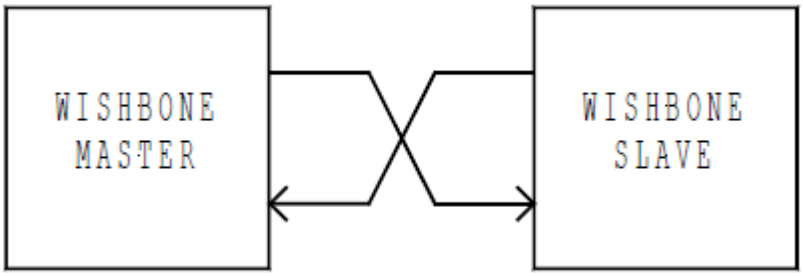


Figure 2.11.1.1: Point-to-point interconnection (Peterson, 2010)

The point-to-point interconnection is the simplest way to connect two WISHBONE IP cores together. It allows a single MASTER interface to connect to a single SLAVE interface. The MASTER interface could be on microprocessor IP core, and the SLAVE interface could be on a serial I/O port.

ii) Data Flow Interconnection

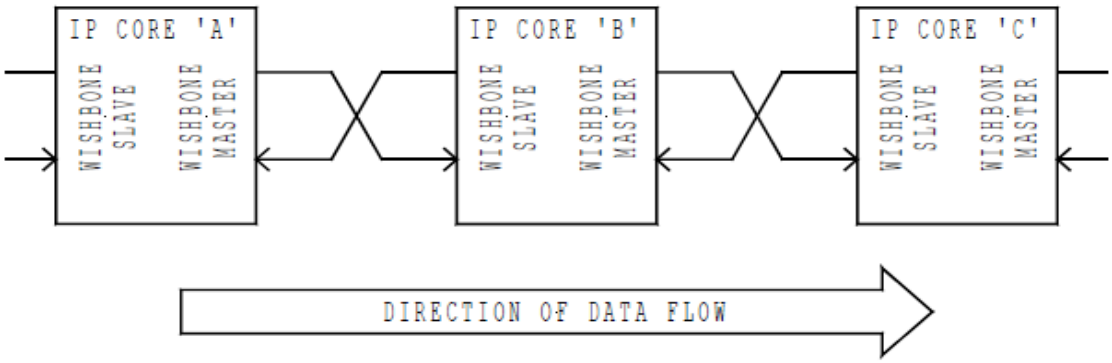


Figure 2.11.1.2: Data flow interconnection (Peterson, 2010)

The data flow interconnection is used when data is processed in a sequential manner, where each IP core in the data flow architecture has both a MASTER and a SLAVE interface. Data flows from core-to-core. This process is also known as pipelining.

The data flow architecture exploits parallelism, thereby speeding up execution time. For instance, if each of the IP cores in the figure represents a floating point processor, then the system has 3 times the number crunching potential of a single unit, assuming that each IP core takes an equal amount of time to solve its problem, and that the problem can be solved in a sequential manner.

iii) Shared Bus Interconnection

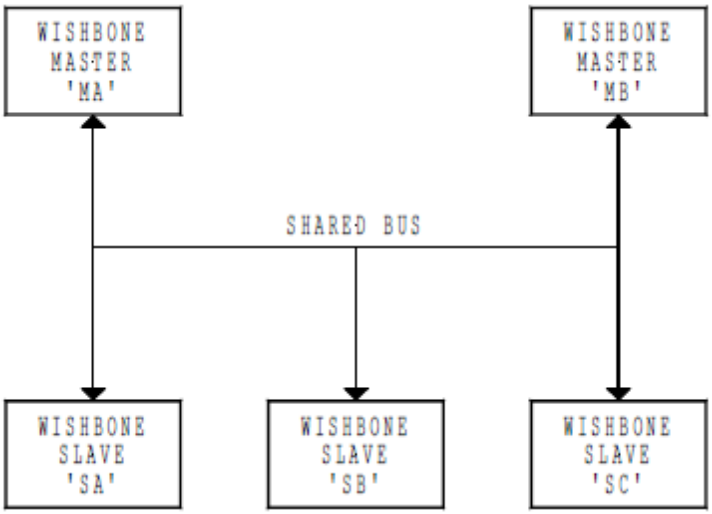


Figure 2.11.1.3: Shared bus interconnection (Peterson, 2010)

This interconnection is normally used to connect 2 or more MASTERs with one or more SLAVEs. As shown in figure 2.9.1.3, a MASTER initiates a bus cycle to a target SLAVE. The target SLAVE then participates in 1 or more bus cycles with the MASTER.

The shared bus interconnection uses an arbiter to determine when a MASTER may gain access to the shared bus. It also decides how each MASTER accesses the shared resource. The type of arbiter is completely defined by the system integrator, whether using a priority-based or round robin type.

The main advantage to this technique is that it is relatively compact, whereby it needs fewer logic gates and routing resources than other configurations, especially the crossbar switch. However, the main drawback of this technique is that the MASTERs may have to wait for a period of time before gaining access to the interconnection, which degrades the overall speed of the data transfer.

iv) Crossbar Switch Interconnection

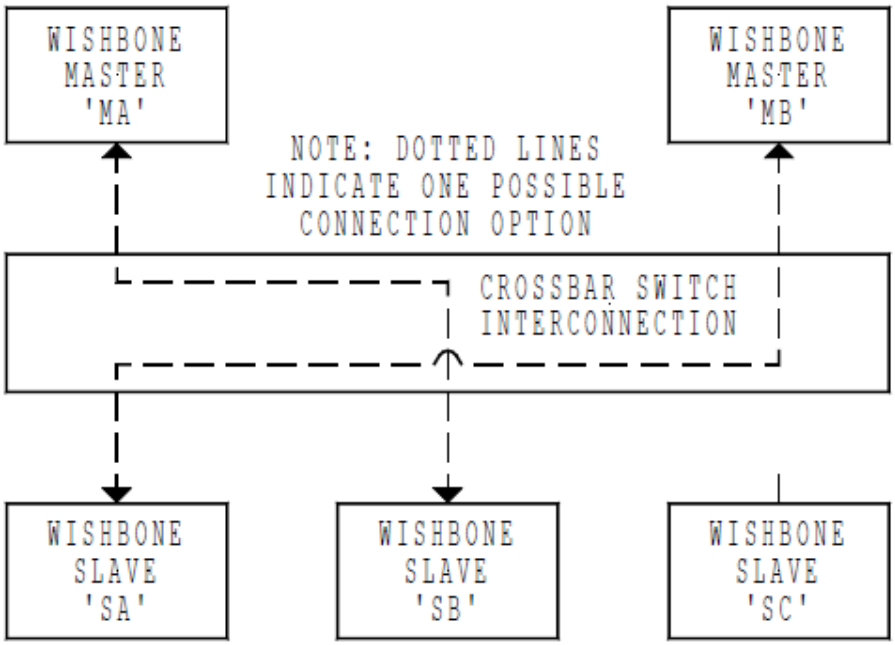


Figure 2.11.1.4: Crossbar Switch interconnection (Peterson, 2010)

The crossbar switch interconnection is used to connect 2 or more WISHBONE MASTERs together so that each can access 2 or more SLAVEs. Using this technique, a MASTER can initiate an addressable bus cycle to a target SLAVE. Similar to shared bus interconnection, an arbiter is also used to determine when each MASTER may gain access to the indicated SLAVE. The difference is the crossbar switch allows more than 1 MASTER to use the interconnection, as long as 2 MASTERs don't access the same SLAVE at the same time.

Under this method, each master arbitrates for a 'channel' on the switch. Once this is established, data is transferred between the MASTER and the SLAVE over a private communication link.

Overall, the crossbar switch technique has a higher data transfer rate than shared bus mechanisms. It can also be expanded to support extremely high data transfer rates. One disadvantage is that more interconnection logic and routing resources are required than shared bus systems.

2.11.2 WISHBONE architecture signal description

There are some signals that are commonly used in the WISHBONE interconnect. This section describes the signals that are used between the MASTER and SLAVE interfaces.

2.11.2.1 System controller (SYSCON) module signals

Signal	Name	Type	Description
System clock	CLK_O	Output	• Coordinate all activities for internal logic within the WISHBONE interconnect. • Connect to the clock input on MASTER and SLAVE interfaces.
Reset	RST_O	Output	• Force all WISHBONE interfaces to restart and initialize all internal self-starting machines. • Connect to the reset input on MASTER and SLAVE interfaces.

Table 2.11.2.1: System controller module signals (Peterson, 2010)

2.11.2.2 MASTER and SLAVE interfaces common signals

Signal	Name	Type	Description
Clock	CLK_I	Input	• Coordinate all activities for the internal logic within the WISHBONE interconnect. • All output signals are registered at the rising edge of clock signal. • All input signals are stable before the

			rising edge of clock signal.
Data input array	DAT_I()	Input	• Pass binary data. Array boundaries are determined by port size with a maximum size of 64-bits
Data output array	DAT_O()	Output	• Pass binary data. Array boundaries are determined by port size with a maximum size of 64-bits.
Reset	RST_I	Input	• Force the WISHBONE interface to restart and initialize all internal self-starting state machines. • Only resets the WISHBONE interface.
Data tag type	TGD_I	Input	• Contain information that is associated with data input array, and is qualified a strobe signal. • Simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. • Examples are parity correction, error correction and time stamp information.
Data tag type	TGD_O	Output	• Contain information that is associated with data input array, and is qualified a strobe signal. • Simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. • Examples are parity correction, error correction and time stamp information.

Table 2.11.2.2: MASTER and SLAVE interfaces common signals (Peterson, 2010)

2.11.2.3 MASTER signals

Signal	Name	Type	Description
Acknowledge	ACK_I	Input	• When asserted, indicates the normal termination of a bus cycle.
Address output array	ADR_O()	Output	• Pass a binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size and granularity.
Select output array	SEL_O()	Output	• Indicate where valid data is expected on the DAT_I() during READ cycles, and where it is placed on DAT_O() during WRITE cycles. • Array boundaries are determined by granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of SEL_O(7..0).
Cycle output	CYC_O	Output	• When asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all cycles. • Useful for interfaces with multi-port interfaces (such as dual port memories). In these cases, the CYC_O signal requests use of a common bus from an arbiter.
Error detect	ERR_I	Input	• Indicate an abnormal cycle termination. The source of the error, and the response generated by the

			MASTER is defined by the IP core supplier.
Strobe	STB_O	Output	• Indicate a valid data transfer cycle. Used to qualify various other signals on the interface such as SEL_O.
Address tag type	TGA_O()	Output	• Contain information associated with address lines, and is qualified by signal STB_O. • Simplify the task of defining new signals because their timing (in relation to every bus cycle) is defined by this specification.
Cycle tag type	TGC_O()	Output	• Contain information associated with bus cycles, and is qualified by signal CYC_O. For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified. They can also be used to discriminate between WISHBONE SINGLE, BLOCK and RMW cycles.
Write enable	WE_O	Output	• Indicate whether the current local bus cycle is a READ or WRITE cycle, and is asserted during WRITE cycles.

Table 2.11.2.3: MASTER signals (Peterson, 2010)

2.11.2.4 SLAVE signals

Signal	Name	Type	Description
Acknowledge	ACK_O	Output	• When asserted, indicates the termination of a normal bus cycle.
Address input array	ADR_I()	Input	• Pass a binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size.
Cycle input	CYC_I	Input	• When asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles.
Select input array	SEL_I()	Input	• Indicate where valid data is placed on the DAT_I() signal array during WRITE cycles, and where it should be present on the DAT_O() signal array during READ cycles. • Array boundaries are determined by the granularity of a port.
Error detect	ERR_O	Output	• Indicate an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier.
Strobe	STB_I	Input	• When asserted, indicates that the SLAVE is selected. A SLAVE shall respond to other WISHBONE signals only when STB_I is asserted (except for reset signal, RST_I).

			• The SLAVE asserts either ACK_O or ERR_O signals in response to every assertion of STB_I signal.
Address tag type	TGA_I()	Input	• Contain information associated with address lines ADR_I(), and is qualified by signal STB_I. • Simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification.
Cycle tag type	TGC_I()	Input	• Contain information associated with bus cycles, and is qualified by signal CYC_I. For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified with the cycle tag. They can also be used to discriminate between WISHBONE SINGLE, BLOCK and RMW cycles.
Write enable	WE_I	Input	• Indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is de-asserted during READ cycles, and is asserted during WRITE cycles.

Table 2.11.2.4: SLAVE signals (Peterson, 2010)

Note that the signal names mentioned in this section is just a convention for convenience purpose in this paper. They may be different in response to the modules designed by the designers. In addition, not all signals are necessary to be implemented in every design. However, there are some rules that must be followed as shown below (Peterson, 2010):

- All WISHBONE interface signals must use active high logic.

- As a minimum, the MASTER interface must include the following signals: ACK_I, CLK_I, CYC_O, RST_I, and STB_O. All other signals are optional.
- As a minimum, the SLAVE interface must include the following signals: ACK_O, CLK_I, CYC_I, STB_I, and RST_I. All other signals are optional.
- The signals must allow MASTER and SLAVE interfaces to support either one of the WISHBONE interconnect technique.
- The signals must allow three basic types of the bus cycle. These include SINGLE READ/WRITE, BLOCK READ/WRITE and RMW (read-modify-write) bus cycles.
- All signals on MASTER and SLAVE interfaces are either inputs or outputs, but are never bi-directional. However, it is permissible to use bi-directional signals in the interconnection logic if the target device supports it.
- A handshaking mechanism allows a used so that either the MASTER or the participating SLAVE interface can adjust the data transfer rate during a bus cycle. This allows the speed of each cycle to be adjusted by either the MASTER or SLAVE interface. This means that all WISHBONE bus cycles run at the speed of the slowest interface.
- The handshaking mechanism allows a participating SLAVE to accept a data transfer, reject a data transfer with an error. The SLAVE does this by generating the ACK_O, ERR_O signals. Every interface must support the ACK_O signal, but the error signal is optional.

2.11.3 WISHBONE Classic Bus Cycles

In WISHBONE architecture, the MASTER and SLAVE interfaces are interconnected with a set of signals that permit them to exchange data. These signals are cumulatively known as a bus, and are contained within a functional module called the INTERCON. Address, data and other information is impressed upon the bus in the form of bus cycles.

There are three types of bus cycles:

i. Single READ/WRITE cycle
ii. BLOCK READ/WRITE cycle
iii. READ-MODIFY-WRITE (RMW) cycle

i) Single READ/WRITE cycle

The single read/write cycle performs one data transfer at a time. These are the basic cycles used to perform data transfers on the WISHBONE interconnect.

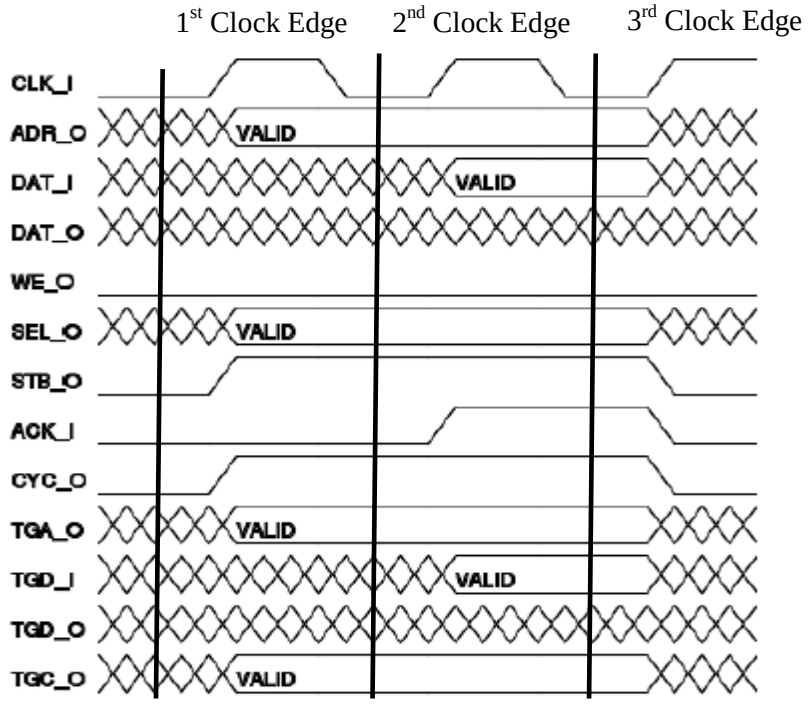


Figure 2.11.3.1: Standard single READ cycle (WISHBONE B4)

In a standard single READ cycle, the bus protocol works as follows:

- 1st clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER de-asserts WE_O to indicate a READ cycle.
  - MASTER presents back select SEL_O() to indicate where it expects data.
  - MASTER asserts CYC_O and TGC_O() to indicate the start of the cycle.
  - MASTER asserts STB_O to indicate the start of the phase.

- 2nd clock edge
  - SLAVE decodes inputs and responding SLAVE asserts ACK_I.
  - SLAVE presents valid data on DAT_I() and TGD_I().
  - SLAVE asserts ACK_I in response to STB_O to indicate valid data.
  - MASTER monitors ACK_I, and prepares to latch data on DAT_I().
    [Note: SLAVE may insert wait states before asserting ACK_I, thereby allowing it to throttle the cycle speed. Any number of wait states may be added]

- 3rd clock edge
  - MASTER latches data on DAT_I().
  - MASTER negates STB_O and CYC_O to indicate the end of cycle.
  - SLAVE de-asserts ACK_I in response to negated STB_O.

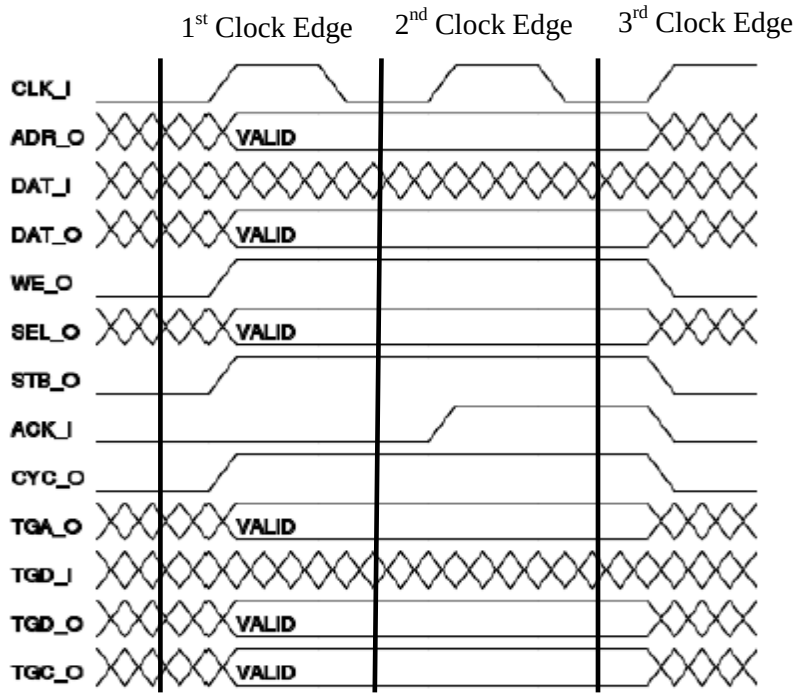


Figure 2.11.3.2: Standard single WRITE cycle (Peterson, 2010)

In a standard WRITE cycle, the bus protocol works as follow:

- 1st clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER presents a valid data on DAT_O() and TGD_O().
  - MASTER asserts WE_O to indicate a WRITE cycle.
  - MASTER presents bank select SEL_O() to indicate where it sends data.
  - MASTER asserts CYC_O and TGC_O() to indicate the start of the cycle.
  - MASTER asserts STB_O to indicate the start of the phase.

- 2nd clock edge
  - SLAVE decodes inputs, and responding SLAVE asserts ACK_I.
  - SLAVE prepares to latch data on DAT_O() and TGD_O().
  - SLAVE asserts ACK_I in response to STB_O to indicate latched data.
  - MASTER monitors ACK_I, and prepares to terminate the cycle.
    [Note: SLAVE may insert wait states before asserting ACK_I, thereby allowing it to throttle the cycle speed. Any number of wait states may be added]

- 3rd clock edge
  - SLAVE latches data on DAT_O() and TGD().
  - MASTER de-asserts STB_O and CYC_O to indicate the end of the cycle.
  - SLAVE de-asserts AKC_I in response to negated STB_O.

ii) BLOCK READ/WRITE cycle

The BLOCK transfer cycles perform multiple data transfers. They are very similar to single READ and WRITE cycles, but have a few special modifications to support multiple transfers.

During BLOCK cycles, the interface basically performs SINGLE READ/WRITE cycles. However, the BLOCK cycles are modified somewhat so that these individual cycles (called phases) are combined together to form a single BLOCK cycle, which is useful when multiple MASTERS are used on the interconnect (Peterson, 2010).

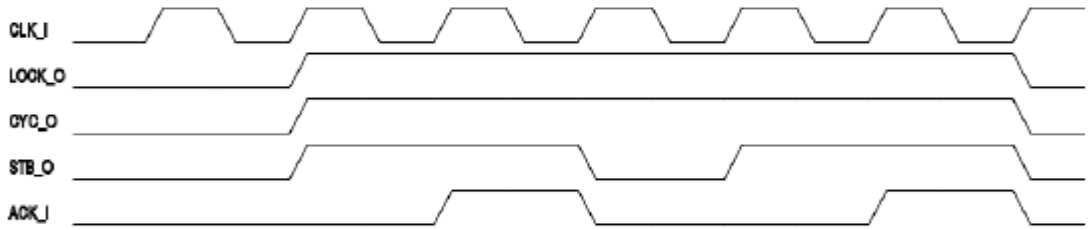


Figure 2.11.3.3: Use of CYC_O signal during BLOCK cycles (Peterson, 2010)

From figure above, the CYC_O signal is asserted for the duration of a BLOCK cycle. It can be used to request permission to access a shared resource from a local arbiter. To hold the access until the end of the cycle, a signal to request the complete ownership of the bus, called LOCK_O is introduced. When this signal is asserted, it indicates that the current bus cycle is uninterruptible. During each of the data transfer phases (within the block transfer), the normal handshaking protocol between STB_O and ACK_I is maintained (WISHBONE b4).

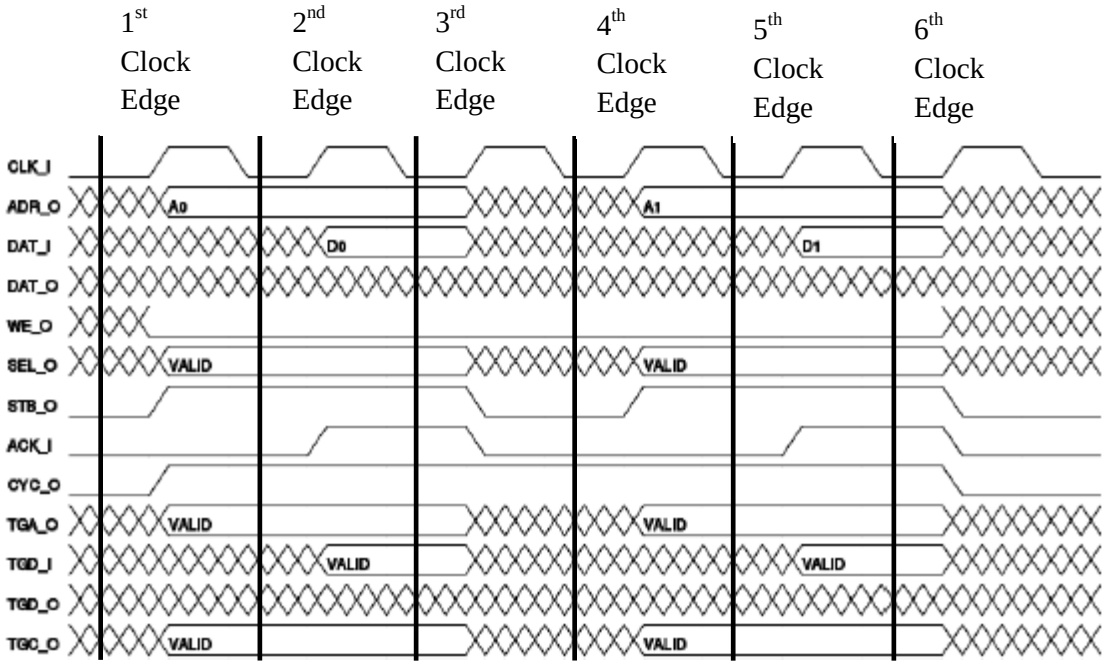


Figure 2.11.3.4: Standard BLOCK READ cycle (Peterson, 2010)

The protocol for a standard BLOCK READ cycle works as follows:

- 1st clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER de-asserts WE_O to indicate a READ cycle.
  - MASTER presents bank select SEL_O to indicate where it expects data.
  - MASTER asserts CYC_O and TGC_O to indicate the start of the cycle.
  - MASTER asserts STB_O to indicate the start of the first phase.
    [Note: the MASTER asserts CYC_O and/or TGC_O() at, or any time before, 2nd clock edge.]

- 2nd clock edge
  - SLAVE decodes inputs, and responding SLAVE asserts ACK_I.
  - SLAVE presents valid data on DAT_I() and TGD_I().

- MASTER monitors ACK_I, and prepares to latch DAT_I() and TGD_I().

- 3rd clock edge
  - MASTER latches data on DAT_I and TGD_I().
  - MASTER de-asserts STB_O to introduce a wait state.

- 4th clock edge
  - MASTER presents a valid address on ADR_O() and TGA().
  - MASTER de-asserts WE_O to indicate a READ cycle.
  - MASTER presents bank select SEL_O() to indicate where it expects data.
  - MASTER asserts STB_O.

- 5th clock edge
  - SLAVE decodes inputs, and responding SLAVE asserts ACK_I.
  - SLAVE presents valid data on DAT_I() and TGD_I().

- 6th clock edge
  - MASTER latches data on DAT_I() and TGD_I().
  - MASTER terminates cycle by negating STB_O and CYC_O.

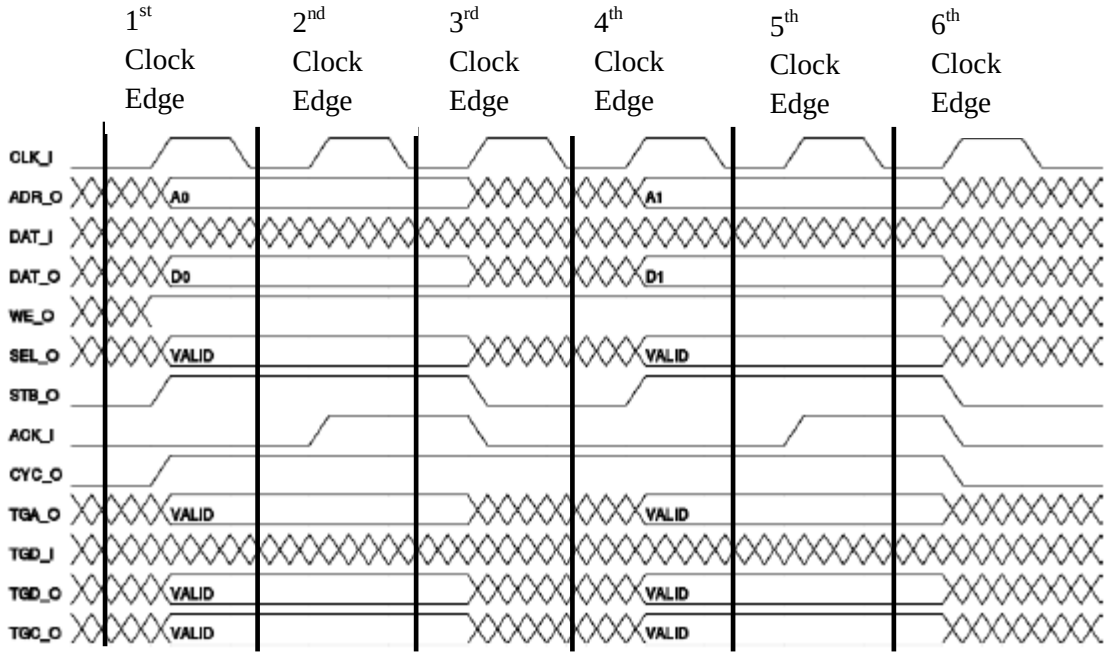


Figure 2.11.3.5: Standard BLOCK WRITE cycle (Peterson, 2010)

The protocol for a BLOCK WRITE cycle works as follows:

- 1st clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER asserts WE_O to indicate a WRITE cycle.
  - MASTER presents bank select SEL_O() to indicate where it sends data.
  - MASTER asserts CYC_O and TGC_O to indicate the start of the cycle.
  - MASTER asserts STB_O to indicate the start of the first phase.
    [Note: the MASTER asserts CYC_O and/or TGC_O at, or any time before 2nd clock edge.]

- 2nd clock edge
  - SLAVE decodes inputs, and responding SLAVE asserts ACK_I.

- 3rd clock edge
  - MASTER monitors ACK_I.
  - MASTER de-asserts STB_O to introduce a wait state.

- 4th clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER asserts WE_O to indicate a WRITE cycle.
  - MASTER presents bank select SEL_O() to indicate where it sends data.
  - MASTER asserts CYC_O and TGC_O() to indicate the start of the cycle.
  - MASTER asserts STB_O to indicate the start of the second phase.

- 5th clock edge
  - MASTER presents a valid address on ADR_O() and TGA_O().
  - MASTER de-asserts WE_O to indicate a READ cycle.
  - MASTER presents bank select SEL_O() to indicate where it expects data.

- 5th clock edge
  - SLAVE decodes inputs, and responding SLAVE asserts ACK_I.

- 6th clock edge
  - MASTER monitors ACK_I.
  - MASTER terminates cycle by negating STB_O and CYC_O.

iii) READ-MODIFY-WRITE (RMW) cycle

The RMW cycle is used for indivisible semaphore operations. During the first half of the cycle a single read data transfer is performed. During the second half of the cycle a write data transfer is performed. The CYC_O signal remains asserted during both halves of the cycle.

It is possible for the MASTER and SLAVE to be designed so that they do not support the RMW cycles.

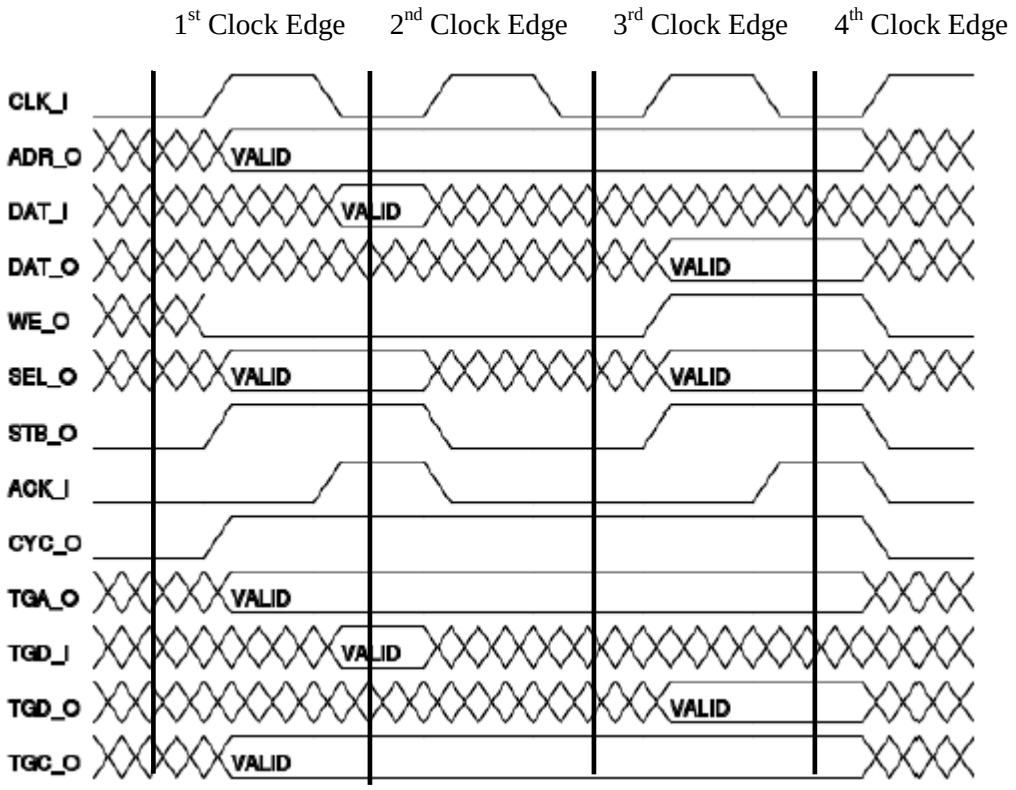


Figure 2.11.3.6: Standard RMW cycle (Peterson, 2010)

A standard RMW cycle protocol works as follows:

- 1st clock edge
  - MASTER presents ADR_O() and TGA_O().
  - MASTER de-asserts WE_O to indicate a READ cycle.
  - MASTER presents bank select SEL_O() to indicate where it expects data.
  - MASTER asserts CYC_O and TGC_O() to indicate the start of the cycle.
  - MASTER asserts STB_O.
    [Note: the MASTER asserts CYC_O and/or TGC_O at, or any time before 2nd clock edge. The use of TAGN_O is optional.]

- SETUP, 2nd edge
  - SLAVE decodes inputs, and responds by asserting ACK_I.
  - SLAVE presents valid data on DAT_I() and TGD_I().
  - MASTER monitors ACK_I, and prepares to latch DAT_I() and TGD_I().

- 2nd clock edge
  - MASTER latches data on DAT_I() and TGD_I().
  - MASTER de-asserts STB_O to introduce a wait state.

- SETUP, 3rd edge
  - SLAVE de-asserts ACK_I in response to STB_O.
  - MASTER asserts WE_O to indicate a WRITE cycle.
    [Note: any number of wait states can be inserted by the MASTER at this point.]

- 3rd clock edge
  - MASTER presents WRITE data on DAT_O() and TGD_O().
  - MASTER presents new bank select SEL_O() to indicate where it sends data.
  - MASTER asserts STB_O.

- SETUP, 4th edge
  - SLAVE decodes inputs, and responds by asserting ACK_I.
  - SLAVE prepares to latch data on DAT_O() and TGD_O().
  - MASTER monitors ACK_I, and prepares to terminate the data phase.
    [Note: any number of wait states can be inserted by the SLAVE at this point.]

- 4th clock edge
  - SLAVE latches data on DAT_O() and TGD_O().
  - MASTER de-asserts STB_O and CYC_O indicating the end of the cycle.
  - SLAVE de-asserts ACK_I in response to negated STB_O.

Chapter 3: Design Methodology and Development Tools

3.1 Design Tools

There are lots of design tools that can create and edit Verilog HDL files. Among them, there are 3 famous Verilog HDL design tools with rich graphical user interface (GUI) which will be discussed and compared in this chapter.

3.1.1 Altera Quartus II

The Quartus II development software is designed by Altera. This software provides a complete design environment for system – on – a – programmable – chip (SOPC) design. Besides that, it also ensures easy design entry, fast processing, and straightforward device programming. It offers a rich graphical user interface (GUI) complemented with an illustrated, easy – to – use online Help system. (Altera, 2012)

The Quartus II software can combine different types of design files into a hierarchical project, choosing the design entry format that works best for each functional block. Moreover, the Quartus II software can create block diagrams that describes at a high – level, then uses additional block diagrams, schematics, AHDL Text Design Files (.tdf), EDIF Input Files (.edf), VHDL Design Files (.vhd) , and Verilog HDL (.v) to create lower – level design components. (Altera, 2012)

The Quartus II software can work with multiple files at the same time, editing multiple design files to transfer information between them, while simultaneously compiling or simulating another project. It can also view an entire hierarchy of design files and move smoothly from one hierarchical level to another. (Altera, 2012)

3.1.2 Synopsys VCS

VCS development software is designed by Synopsys. This software is based on multi – core technology, which can delivers a 2x verification speed – up that helps users find design bugs early in the product development cycle. VCS multi – core technology cuts down verification time by running the design, testbench, assertions, coverage and debug in parallel on machines with multiple cores. (Synopsys, 2012)

VCS supports all popular design and verification languages including Verilog HDL, VHDL, SystemVerilog, OpenVera, and SystemC™ and the VMM, OVM, and UVM™ methodologies which help VCS users to develop high – quality designs. The VCS solution's advanced bug – finding technologies include full – featured Native Testbench (NTB), complete assertions, comprehensive code and functional coverage to find more design bugs faster and easier. (Synopsys, 2012)

Additionally, the VCS solutions powerful debug and visualization environment minimizes the turnaround time to find and fix design bugs. VCS with MVSIM and MVRC delivers innovative voltage – aware verification techniques to find bugs related to modern low power designs. (Synopsys, 2012)

3.1.3 Mentor Graphics ModelSim XE III 10.1a

Mentor Graphics ModelSim XE III 10.1a is created by Mentor Graphics and Xilinx. This development software enables users to verify the hardware descriptive language (HDL) source code, behavioural, functional and timing simulation of the designs. It includes a complete HDL simulation and debugging environment providing 100% VHDL and Verilog language coverage, a source code viewer/editor, waveform viewer, design structure browser, list window and a host of other features designed. (ModelSim, 2012)

This software has a Student Edition (SE) which is a freeware but it is limited to 10,000 lines of code. Although it has its limitation, it will not affect this project because 10,000 lines of code are sufficient for the purpose of this project.

The ModelSim XE III 10.1a has 30% of Professional Edition (PE) which has the performance (speed) of the simulation engine. It does not slow down the time needed to compile the design (VCOM/VLOG) or to load the design in MXE III (VSIM). The slowdown occurs during simulation. For example, a design that takes 20 seconds to run in PE, will take approximately 60 seconds to run in MXE III. (ModelSim, 2012)

To choose the most appropriate design tools in this project, some factors such as language supported, user-friendly environment, affordability and performance need to be put in a serious consideration. Since most the simulators support system level and RTL design, some minor comparisons between the simulators discussed in this paper are shown in the table below.

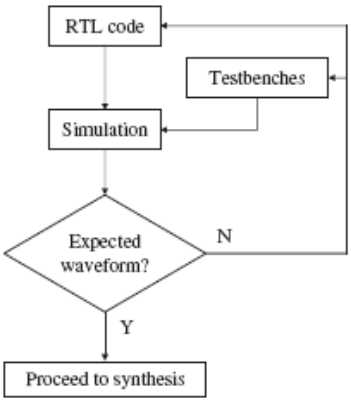
Simulator Chosen / Factors Considered	Altera Quartus II	ModelSim	VCS
Company	ALTERA®	Mentor Graphics®	SYNOPSYS®
Language Supported	VHDL Verilog HDL	VHDL-2002 V2002 SV2005	VHDL-2002 V2001 SV2005
Platform Supported	-Windows XP/7 -Linux	-Windows XP/Vista/7 -Linux	-Linux
Affordability	No	Yes (SE Edition only)	No

Table 3.1: Comparison between simulators chosen

Based on the comparison table above, it is clearly stated that the software which is going to be used in this project is Mentor Graphics ModelSim XE III 10.1a; this is because ModelSim XE III 10.1a is free license software. Moreover, I've been practicing this software since my sophomore year. Thus, I've been familiar with the software and do not need to learn other development software from scratch.

3.2 Design Method

Designing a PS/2 Controller involves two levels which are the architecture level and register transfer level (RTL). It needs to design the I/O unit of the PS/2 Controller by using Verilog code to describe the algorithm and data flow. After completing the Verilog code, the testbench is used to simulate and test the Verilog code whether it is functionally correct or not.



Flow Chart 3.2.1: Design Flow

A top down design approach was used in the design accordingly to the functionality in this project. Specification is the beginning of the design methodology. The functionality and feature of the PS/2 Controller are defined. The design is made to meet the specification at first. Secondly, the RTL coding is written based on the functionality of the PS/2 Controller's design. When the RTL code is completed, the next step is to create the testbench using Verilog to simulate the design. If the expected waveforms are not correct, the RTL code needs to be corrected repeatedly until the correct waveforms are generated.

Notes: The synthesis process is not performed in this project. Hence, the formal verification on the synthesis netlist will not be used.

Chapter 4 System Specification

4.1 Naming Convention

Module - [lvl]_[mod. name]

Instantiation - [lvl]_[abbr. mod. name]

Pin - [lvl]_ [type]_[abbr. mod. name]_[pin name]

- [lvl]_ [type]_[abbr. mod. name]_[stage]_[pin name]

Abbreviation

	Description	Case	Available	Remark
lvl	level	lower	c: Chip u: Unit b: Block	
mod. name	module name	lower all	any	
abbr. mod. name	abbreviated module name	lower all	any	maximum 3 characters
type	pin type	lower	o: output i: input f-: function	
stage	stage name	lower all	if, id, ex, mem, wb	
pin name	pin name	lower all	any	several words are separated by “-”

Table 4.1: Naming Convention

Chapter 5 Microarchitecture Specification (Unit Level)

5.1 Microarchitecture (unit level) of RISC32 processor

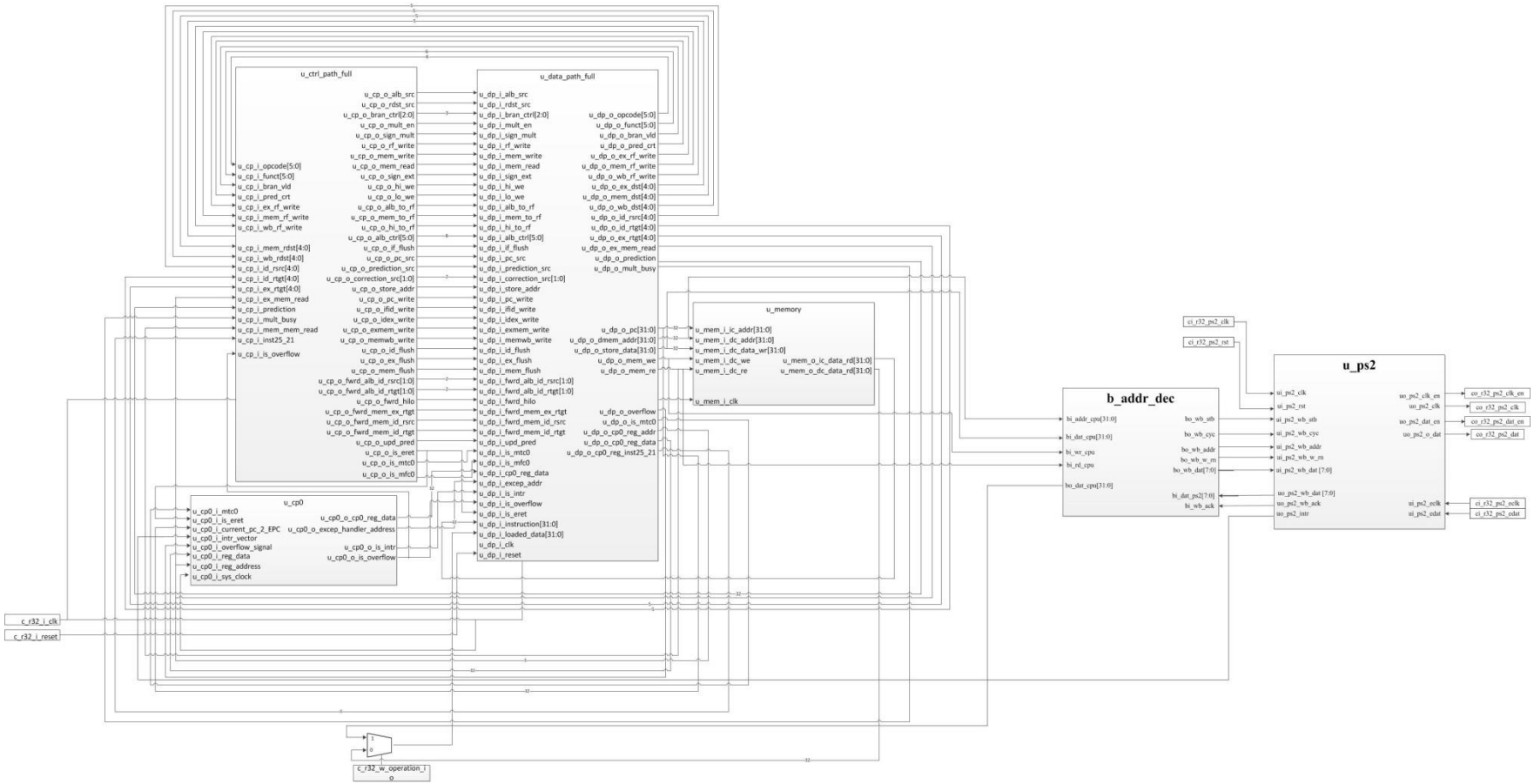


Figure 5.1: Microarchitecture (unit level) of RISC32 processor

5.2 Design Hierarchy

Chip Partitioning at System Level	Unit Partitioning at Architecture Level	Block and Functional Block Partitioning at RTL Level (Microarchitecture level)	Sub-block
c_risc32_full	u_data_path_full	b_reg_file	
		b_alb_32	
		b_mult_32	add_lvl1
			adder_lvl1_firstrow
			Add_lvl1_lastrow
			adder_lvl2
			adder_lvl2_lastrow
			adder_lvl3
			adder_lvl4
			adder_lvl5
			sub_lvl1_lastrow
		b_branch_pred	
	u_ctrl_path_full	b_alb_ctrl	
		b_iag_ctrl	
		b_main_ctrl	
		b_fwrd	
		b_itld_ctrl	
	u_memory	b_cache (for instruction)	
		b_cache (for data)	
	u_cp0	b_cp0_dc	
		b_cp0_regfile	
	u_ps2	**b_transmit**	
		b_receive	
		b_wb_if	
		b_synch	

Table 5.2: Design hierarchy of a PS/2 mouse system integration to RISC32 processor

5.3 Datapath Unit

5.3.1 Datapath Unit's Interface

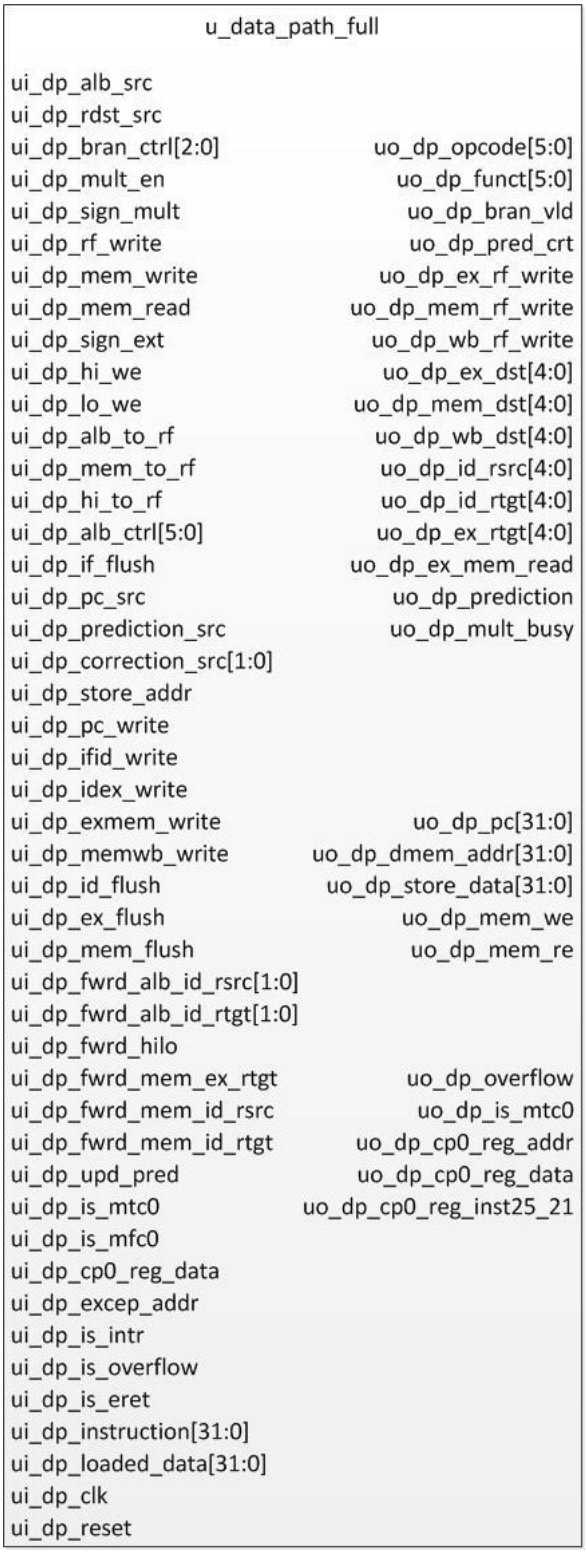


Figure 5.3.1: Full RISC32's Datapath Unit

5.3.4 Microarchitecture for Datapath Unit

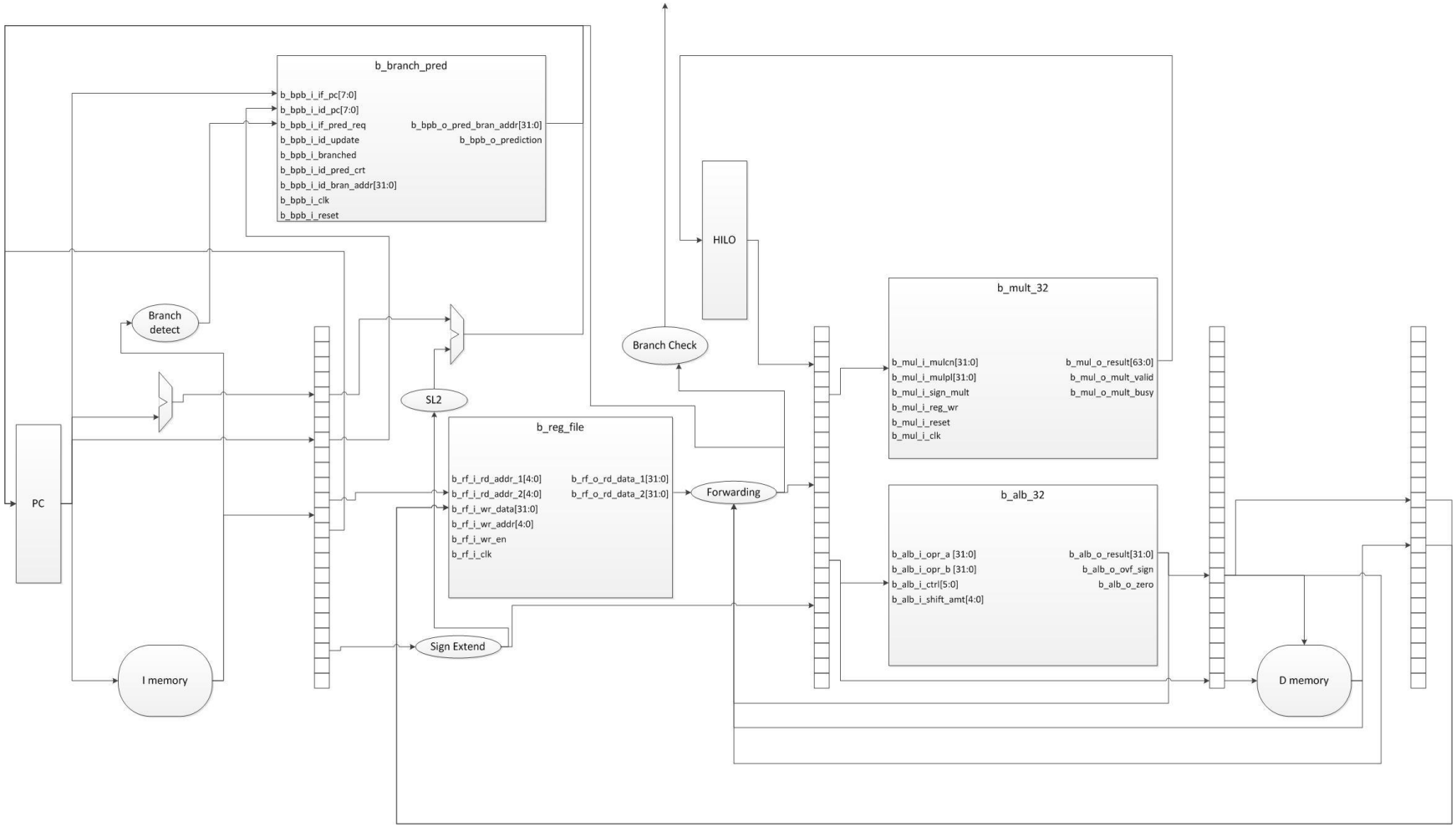


Figure 5.3.4: Microarchitecture for Datapath Unit

5.4 Control Path Unit

5.4.1 Control Path's Unit Interface

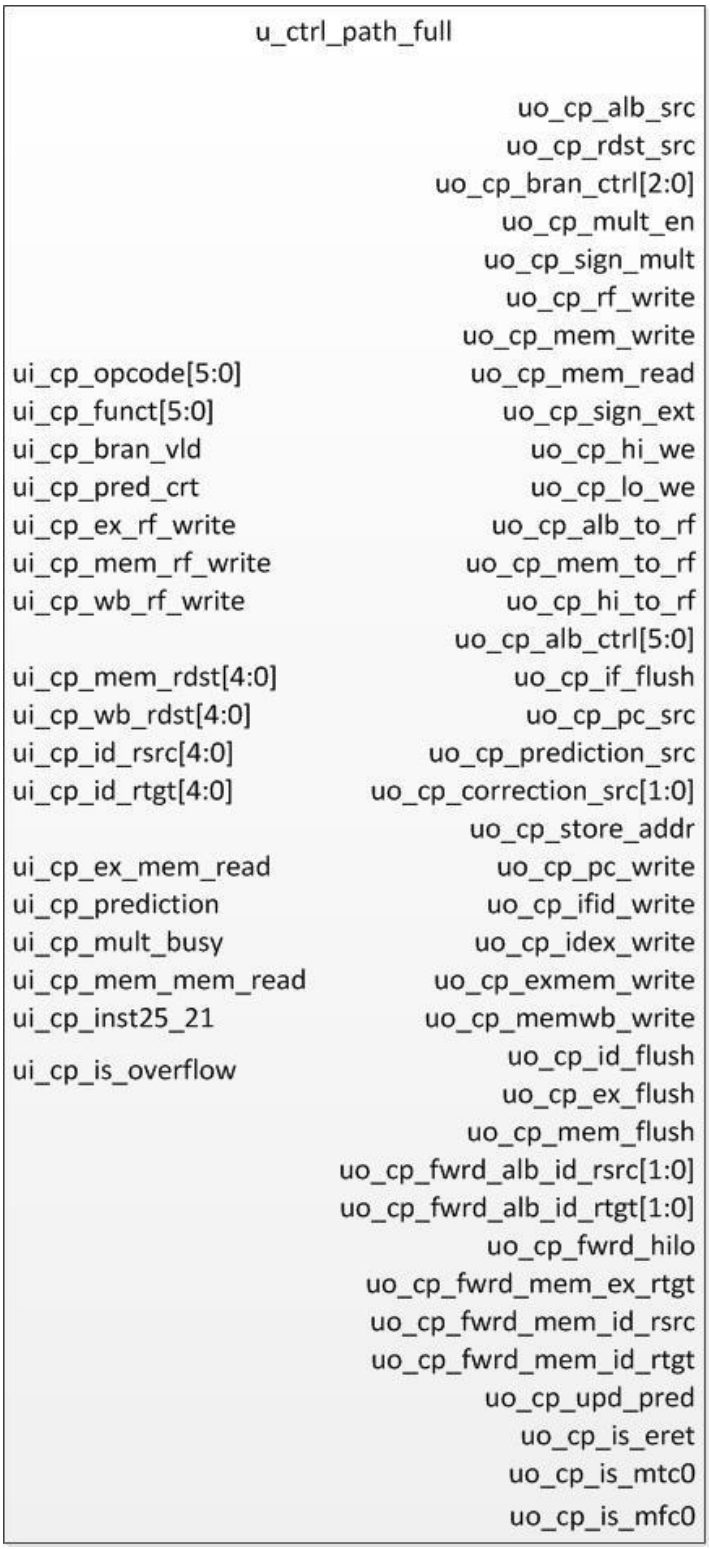


Figure 5.4.1: Full RISC32's Control Path Unit

5.4.2 Microarchitecture for Control Path Unit

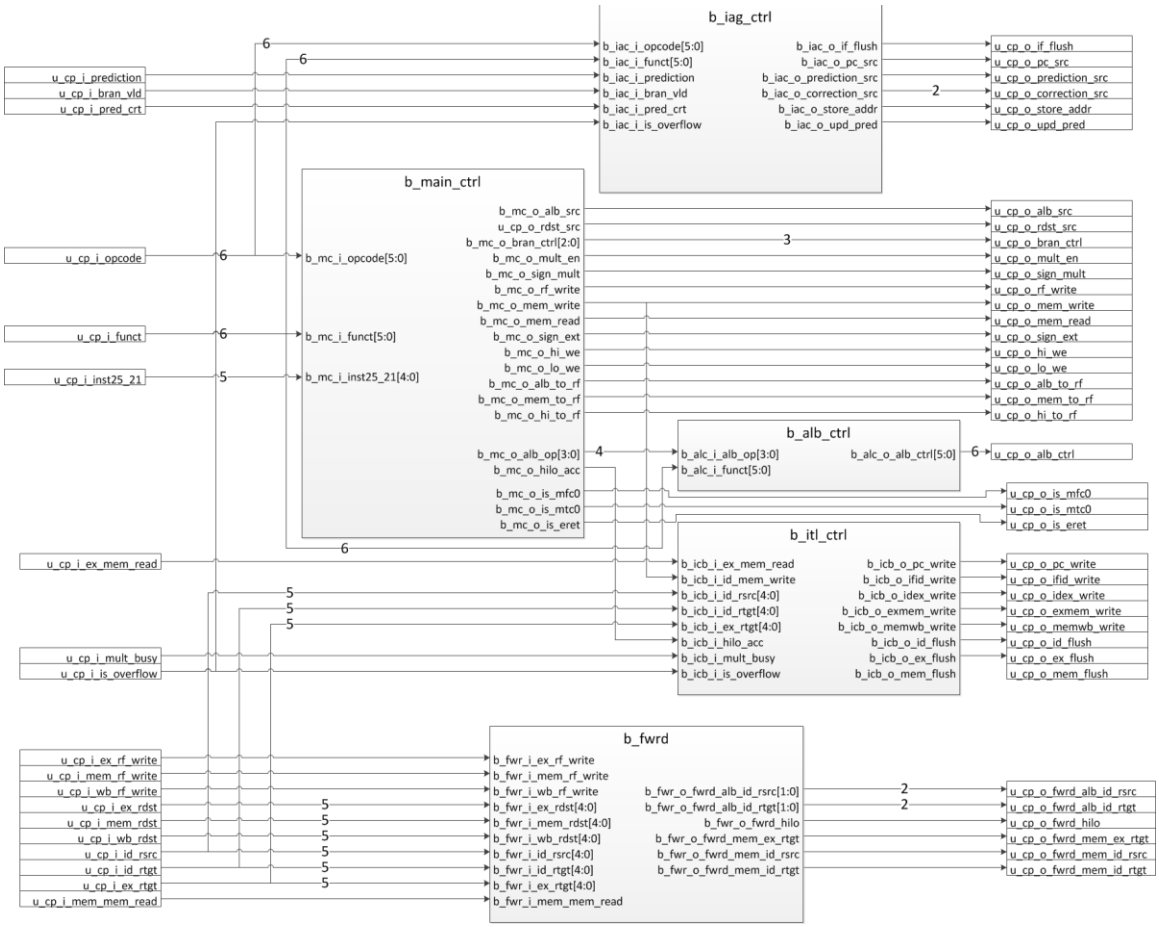


Figure 5.4.2: Microarchitecture for Full RISC32's Control Path Unit

5.5 Memory Unit

5.5.1 Memory Unit's Interface

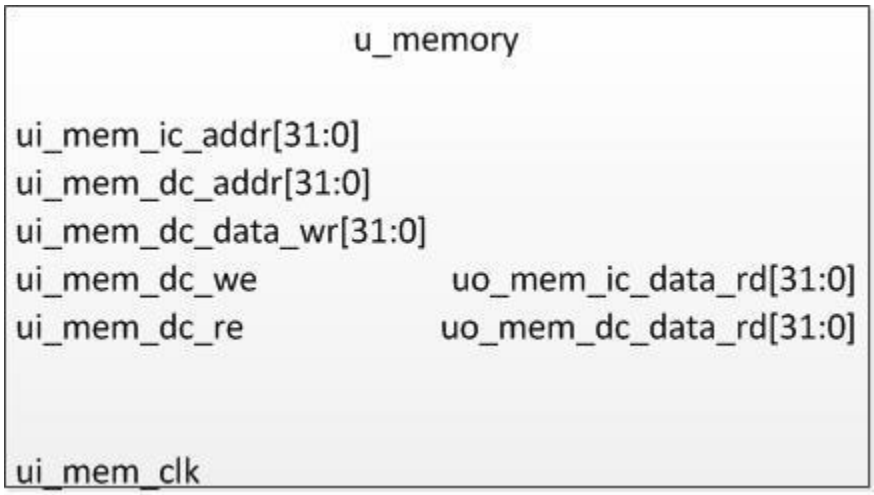


Figure 5.5.1: Memory Unit's interface

5.6 Co-Processor 0 Unit

5.6.1 Co-Processor 0 Unit's Interface

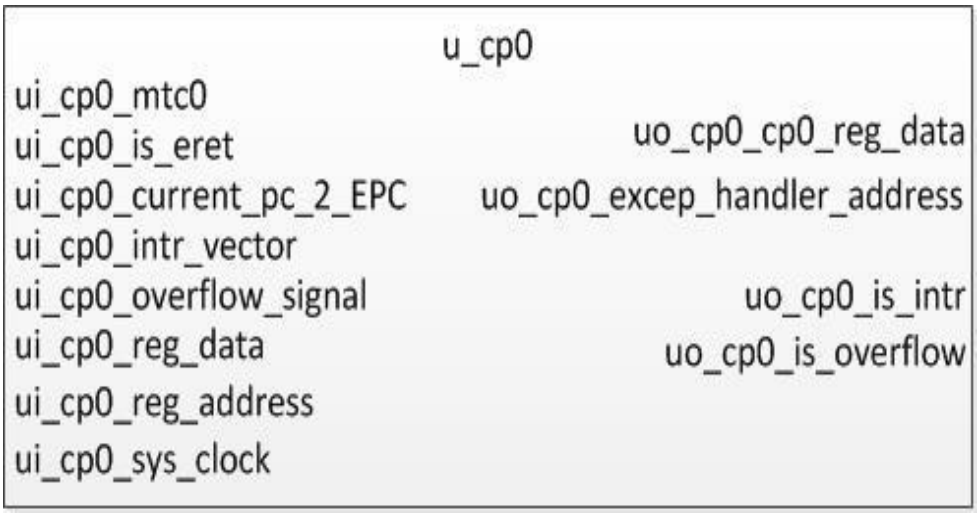


Figure 5.6.1: Co-Processor 0 Unit's Interface

The Co-Processor 0 (CP0) is a unit used to process and store exception and interrupt information. It plays a major role in exception/interrupt handling mechanism. Once the instruction *mtc0* is decoded, the control signal will travel along along the pipeline and

reach to CP0 at stage WB. Both register write, either to CP0 register file (via instruction *mtc0*) or CPU register file (via instruction *mfc0*) will be completed in CPU stage WB. Once the instruction *eret* is decoded, the control signal will immediately output by Control Unit to CP0.

5.6.3 Microarchitecture for Co-Processor 0 Unit

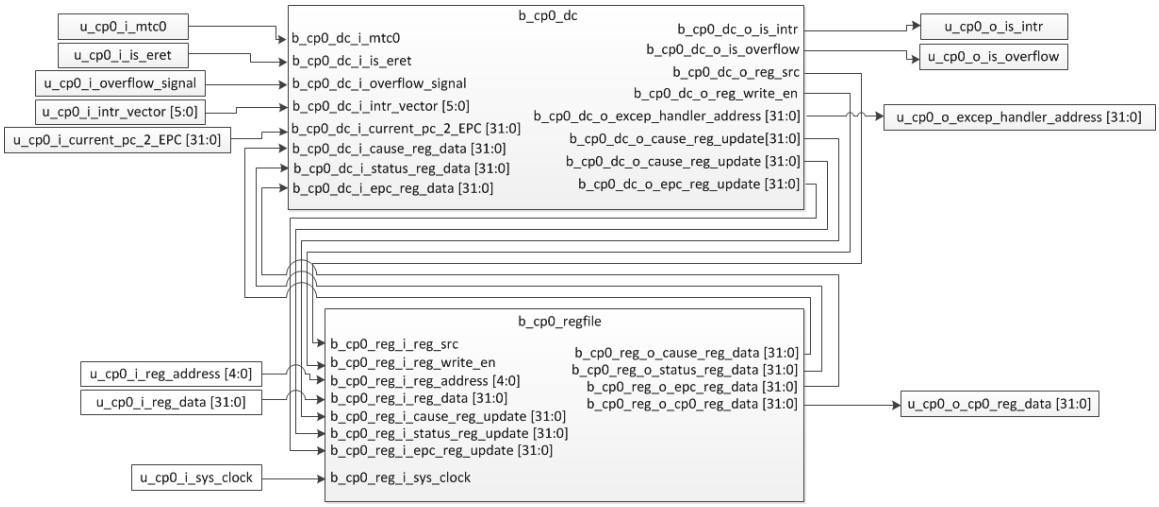


Figure 5.6.4: Microarchitecture for Co-Processor 0 Unit

5.7 PS/2 Controller Unit (u_ps2)

5.7.1 PS/2 Controller Unit's interface

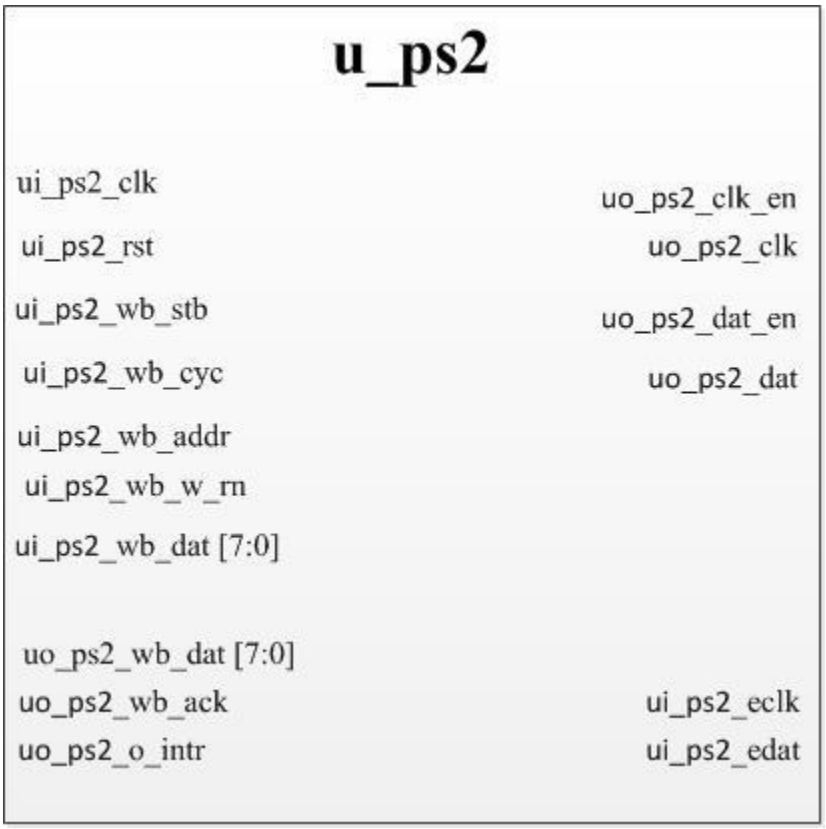


Figure 5.7.1: PS/2 Controller's Unit interface

5.7.2 Input pins description

Pin Name: ui_ps2_clk	Source → Destination : External Source → u_ps2	Registered: No
Pin Function: Clock signal for u_ps2.		
Pin Name: ui_ps2_rst	Source → Destination : External Source → u_ps2	Registered: No
Pin Function: Reset signal for u_ps2. When asserted, resets the whole unit of u_ps2.		
Pin Name: ui_ps2_wb_stb	Source → Destination : External Source → u_ps2	Registered: No
Pin Function:		

Strobe signal for u_ps2. When asserted, indicates that u_ps2 is selected to respond to other WISHBONE signals (except for reset signal).		
Pin Name: ui_ps2_wb_cyc	Source → Destination : External Source → u_ps2	Registered: No
Pin Function: Cycle signal for u_ps2. When asserted, indicates the start of a valid WISHBONE data transfer cycle.		
Pin Name: ui_ps2_wb_addr	Source → Destination : Datapath Unit → u_ps2	Registered: No
Pin Function: Address bus signal, used to select an internal register of the device from: Asserted = WISHBONE Control register (WCREG) De-asserted = WISHBONE Data register (WDREG)		
Pin Name: ui_ps2_wb_dat[7:0]	Source → Destination : Memory Unit → u_ps2	Registered: No
Pin Function: Data signal sent from Memory Unit.		
Pin Name: ui_ps2_wb_w_rn	Source → Destination : Control Path Unit → u_ps2	Registered: No
Pin Function: Write enable signal for u_ps2. Used to indicate whether current bus cycle is a Read or Write cycle. Asserted = Write De-asserted = Read		
Pin Name: ui_ps2_eclk	Source → Destination : Mouse Controller → u_ps2	Registered: No
Pin Function: Clock input signal from Mouse Controller.		
Pin Name: ui_ps2_edat	Source → Destination : Mouse Controller → u_ps2	Registered: No

Pin Function: Data signal from Mouse Controller.

Table 5.7.2: PS/2 Controller Unit's Input Pin Description

5.7.3 Output Pin Description

Pin Name: uo_ps2_clock_en	Source → Destination: u_ps2 → External	Registered: No
Pin Function: Tri-state enable signal for bidirectional clock signal between Mouse Controller and u_ps2.		
Pin Name: uo_ps2_clk	Source → Destination: u_ps2 → External	Registered: No
Pin Function: Clock output signal to control communication between Mouse Controller and u_ps2.		
Pin Name: uo_ps2_dat_en	Source → Destination: u_ps2 → External	Registered: No
Pin Function: Tri-state enable signal for bidirectional data signal between Mouse Controller and u_ps2.		
Pin Name: uo_ps2_dat	Source → Destination: u_ps2 → External	Registered: No
Pin Function: Data output signal from u_ps2 to Mouse Controller		
Pin Name: uo_ps2_wb_dat[7:0]	Source → Destination: u_ps2 → Memory Unit	Registered: No
Pin Function: Data output signal sent from u_ps2 to Memory Unit		
Pin Name: uo_ps2_wb_ack	Source → Destination: u_ps2 → CP0 Unit	Registered: No
Pin Function: Standard WISHBONE acknowledgement signal.		

Asserted = the PS/2 controller has finished execution of the requested action and the current bus cycle is terminated.		
Pin Name: uo_ps2_intr	Source → Destination: u_ps2 → CP0 Unit	Registered: No
Pin Function: Interrupt signal that is used to alert CP0 Unit to the presence of data received from Mouse Controller. It will not be asserted if the parity bit of the byte received is not correct. Asserted = Signifies that 1 byte of data has been received from Mouse Controller.		

Table 5.7.3: PS/2 Controller Unit's Output Pin Description

5.7.4 Microarchitecture for PS/2 Controller Unit

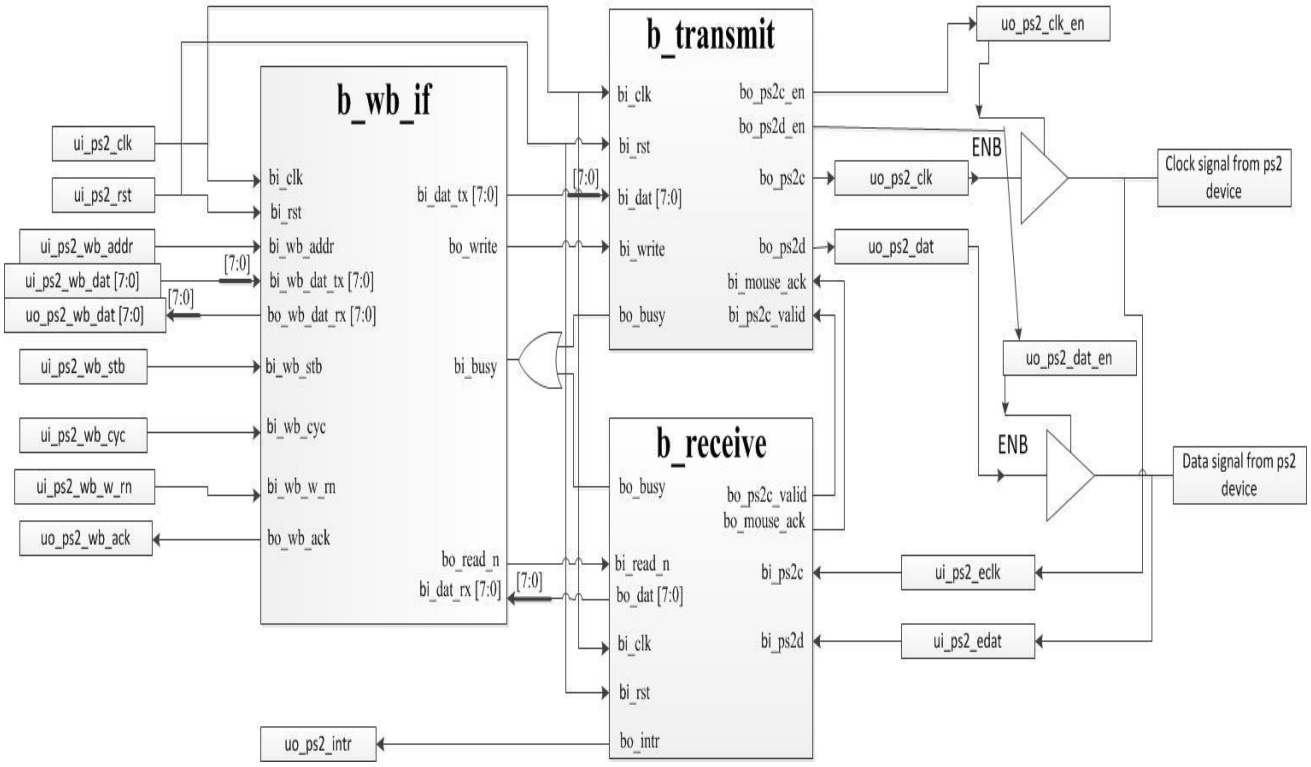


Figure 5.7.4: Microarchitecture for PS/2 Controller Unit

5.7.5 Internal Operation

Test Case	Test Function	Test vector	Expected results
1. System reset.	To initialize the PS/2 Controller Unit.	• ui_ps2_i_rst is asserted for 2 clock cycles, then de-asserted	• uo_ps2_clk_en = 1’b0. • uo_ps2_clk = 1’b1 • uo_ps2_dat_en = 1’b0 • uo_ps2_dat = 1’b1
2. Output WISHBONE signals and enable WRITE cycle (PS/2 Controller transmits)	To enable PS/2 Controller Unit to send data to PS/2 mouse	• ui_ps2_i_wb_addr = 1'b1 • ui_ps2_i_wb_cyc = 1'b1 • ui_ps2_i_wb_stb = 1'b1 • ui_ps2_i_wb_w_rn = 1'b1 • ui_ps2_i_wb_dat = 8’hFA	• uo_ps2_clk_en = 1’b0. • uo_ps2_clk = uo_ps2_clk • uo_ps2_dat_en = 1’b1 • uo_ps2_dat <= ui_ps2_wb_dat[bit_count], bit_count <= bit_count + 1
3. Output WISHBONE signals and enable READ cycle for 3 cycles (PS/2 Controller receives)	To enable PS/2 Controller to receive 3 packets of data from PS/2 mouse	• ui_ps2_i_wb_addr = 1'b1 • ui_ps2_i_wb_cyc = 1'b1 • ui_ps2_i_wb_stb = 1'b1 • ui_ps2_i_wb_w_rn = 1'b0 • ui_ps2_i_edat <= test_receive_data[bit_count], bit_count <= bit_count + 1	• uo_ps2_intr is asserted for 13 clock cycles after stop bit is detected, then de-asserted • uo_ps2_wb_dat outputs the 8 data bits received from PS/2 mouse

Table 5.7.5: Internal Operation for PS/2 Controller Unit

5.7.6 Simulation results

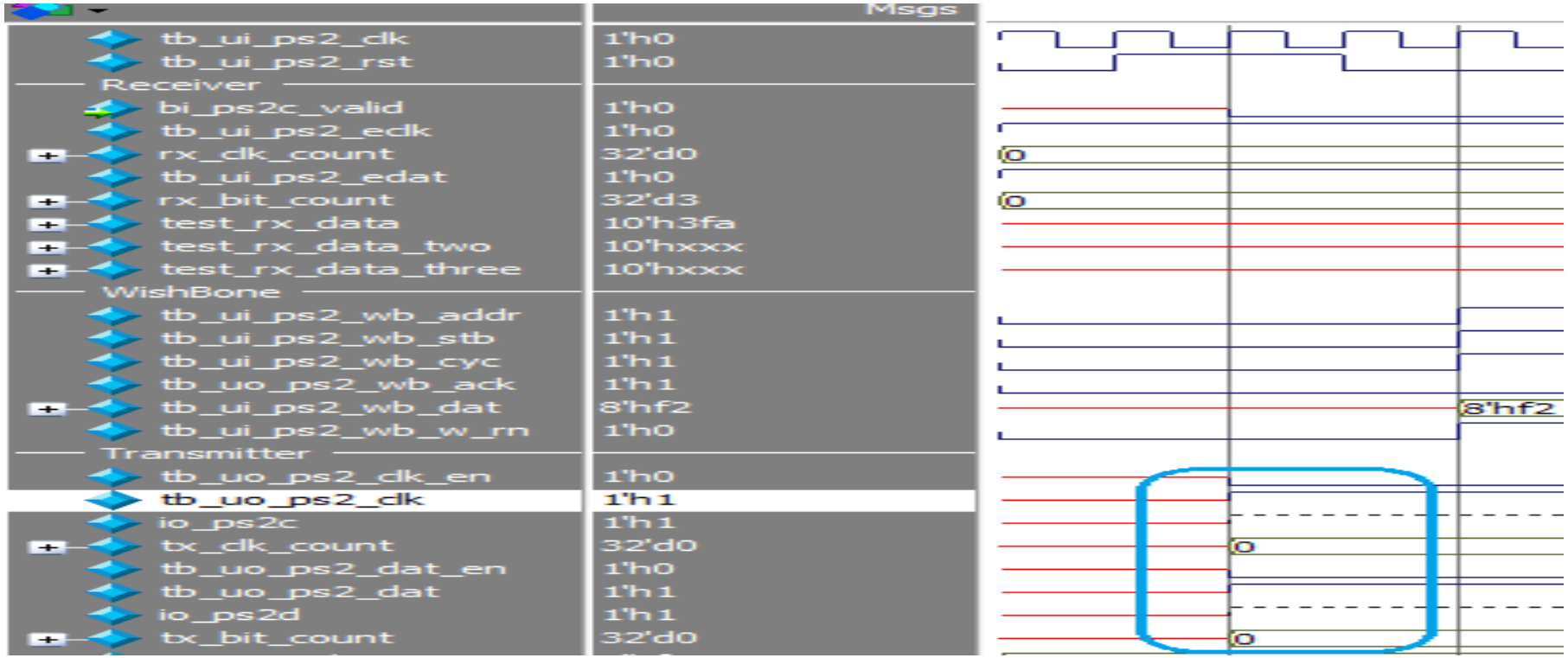


Figure 5.7.6.1: PS/2 Controller Unit simulation result (a)

ui_ps2_rst is asserted for 2 clock cycles to initialize the output signals uo_ps2_clk_en, uo_ps2_clk, uo_ps2_dat_en and uo_ps2_dat

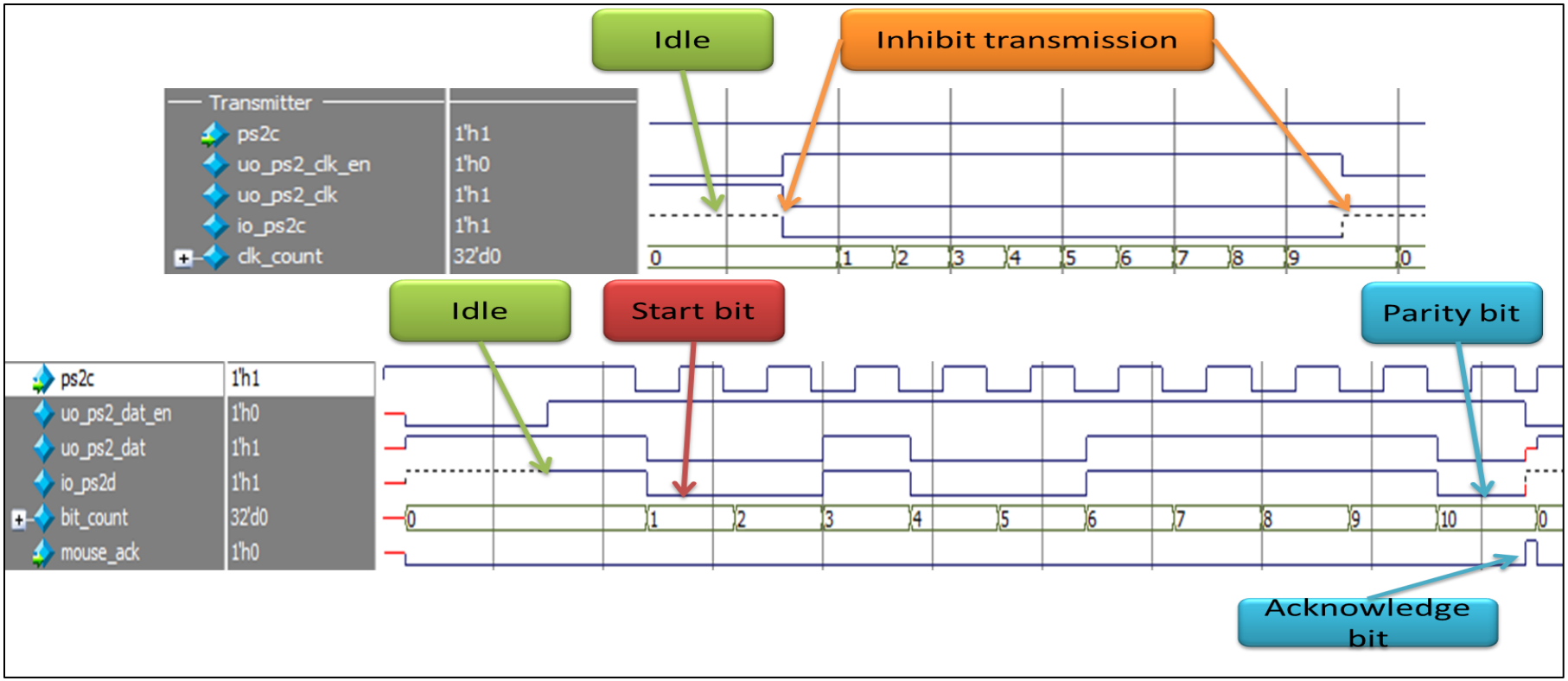


Figure 5.7.6.2: PS/2 Controller Unit simulation result (b)

From figure above, the clock signal, io_ps2c is held at logic 1 when it is in idle state. When the PS/2 Controller wants to transmit command, it will inhibit transmission for 10 clock cycles (100 microseconds) first. After that, the data signal, io_ps2d will first send the start bit which is always “0”, followed by the 8 bits command and the parity bit. After it has received an acknowledge bit from the Mouse Controller, the data signal will be in idle state.

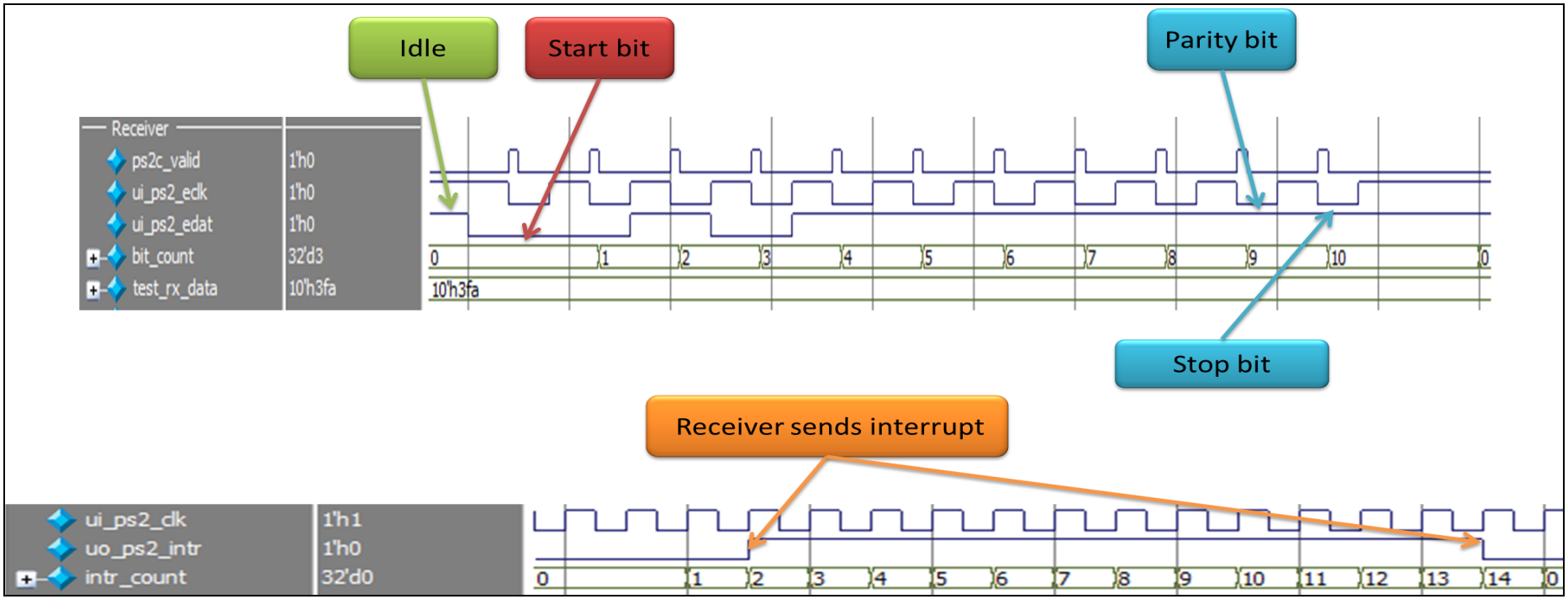


Figure 5.7.6.3: PS/2 Controller Unit simulation result (c)

Based on figure 5.7.6.3, the input signals: ui_ps2_eclk and ui_ps2_edat is at logic 1 which is in idle state. When the PS/2 Controller receives data from the Mouse Controller, ui_ps2_eclk will begin to toggle and the start bit (always "0") is received first, followed by the 8 data bits in LSB format and the parity bit. Finally, the stop bit (always "1") is received by the PS/2 Controller. If the transmission is successful and the parity bit is correct, the PS/2 Controller will send out an interrupt signal, uo_ps2_intr to the cp0 block for 13 clock cycles.

Chapter 6 Microarchitecture Specification (Block Level)

6.1 The Receiver Block

6.1.1 Receiver's Block Interface

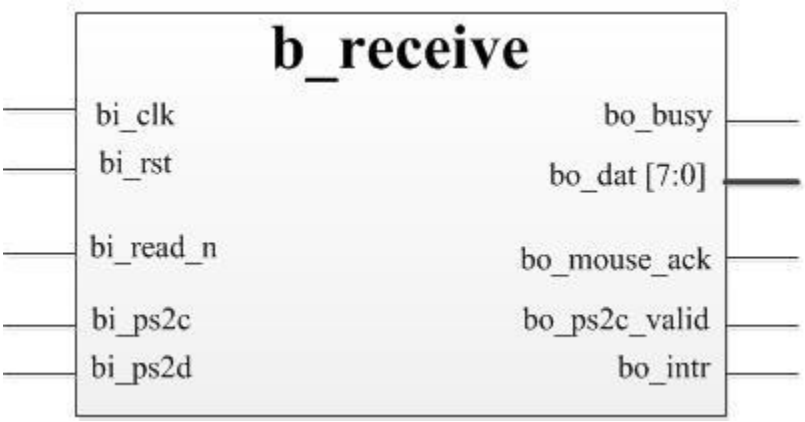


Figure 6.1.1: Receiver's Block interface

6.1.2 Input pin description

Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_clk	External Source → b_receive	1 bit	High	No
Pin Function: Clock signal for b_receive.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_rst	External Source → b_receive	1 bit	High	No
Pin Function: Reset signal for b_receive. When asserted, b_receive will be in idle mode.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_read_n	b_wb_if → b_receive	1 bit	Low	No
Pin Function: Enable signal to enable b_receive to receive data from Mouse Controller.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_ps2c	Mouse Controller → b_receive	1 bit	High	No
Pin Function: Clock input signal from Mouse Controller. It is activated when b_receive is receiving data from Mouse Controller.				

Pin Name: bi_ps2d	Source → Destination: Mouse Controller → b_receive	Size: 1 bit	Active: High	Registered: No
Pin Function: Data input signal from Mouse Controller. It receives 10 bytes of data in serial form.				

Table 6.1.2: Receiver Block Input Pin Description

6.1.3 Output pin description

Pin Name: bo_busy	Source → Destination: b_receive → b_wb_if	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Status signal to inform b_wb_if that a READ cycle is in progress. When asserted, b_transmit should be in idle mode.				
Pin Name: bo_dat_tx[7:0]	Source → Destination: b_receive → b_wb_if	Size: 8 bits	Active: High	Registered: Yes
Pin Function: Sends received data to b_wb_if. When the parity bit received is not correct, b_receive will not send the data.				
Pin Name: bo_mouse_ack	Source → Destination: b_receive → b_transmit	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Status signal to inform b_transmit that the Mouse Controller has received the command b_transmit has transmitted. It is asserted when b_receive receives an acknowledge signal from Mouse Controller.				
Pin Name: bo_ps2c_valid	Source → Destination: b_receive → b_transmit	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Output a valid signal when detected a negative edge of bi_ps2c.				
Pin Name: bo_intr	Source → Destination: b_receive → CP0 unit	Size: 1 bit	Active: High	Registered: Yes
Pin Function:				

Interrupt signal that is used to alert CP0 Unit to the presence of data received from Mouse Controller. It will not be asserted if the parity bit of the byte received is not correct. Asserted = Signifies that 1 byte of data has been received from Mouse Controller.

Table 6.1.3: Receiver Block Output Pin Description

6.1.4 Functionalities and Feature

1. Receive 8 bits of data from PS/2 Mouse by receiving by the least significant bit first.
2. Able to perform a parity check before sending the received data to WISHBONE interface block.
3. Able to send an interrupt signal to alert the processor about the presence of data.
4. Able to receive an acknowledgement signal from PS/2 Mouse to signify that PS/2 Mouse has successfully received command from Transmitter Block.

6.1.5 Internal Operation

Test Case	Test Function	Test vector	Expected results
4. Assert high bi_rst and de-assert low.	To initialize the receiver block.	• bi_rst is asserted	• bo_busy = 1’b0. • bo_mouse_ack = 1’b0 • bo_intr = 1b0
5. Assert high bi_ps2d while bi_read_n is low	To test receiver whether can send an acknowledge signal or not	• bi_read_n is de-asserted for 1 clock cycle	• bo_busy = 1’b0 • bo_mouse_ack = 1’b1 • bo_intr = 1’b0

6. Generate bi_ps2c at 80us when sending test data to bi_ps2d	To create a valid signal for edge-detect circuit when receiving data from bi_ps2d	• repeat(10) begin
		repeat(4) @(posedge bi_clk);
		bi_ps2c = 1'b0;
		repeat(4) @(posedge bi_clk);
		bi_ps2c = 1'b1;
		end
		• repeat(10) @(posedge bi_ps2c)begin
		bi_ps2d <= test_data[bit_count];
		bit_count <=
			• bo_busy = 1’b1
			• bo_dat_tx = test_data
			• bo_mouse_ack = 1’b0
			• bo_intr is asserted for 10 clock cycles

		bit_count + 1; end	
7. Repeat test case 3 for three times	To test whether receiver can complete receiving 3 data packets		• bo_busy = 1'b1 • bo_dat_tx = test_data • bo_mouse_ack = 1'b0 • bo_intr is asserted for 10 clock cycles

Table 6.1.5: Internal operation for Receiver Block

6.1.6 Finite State Machine

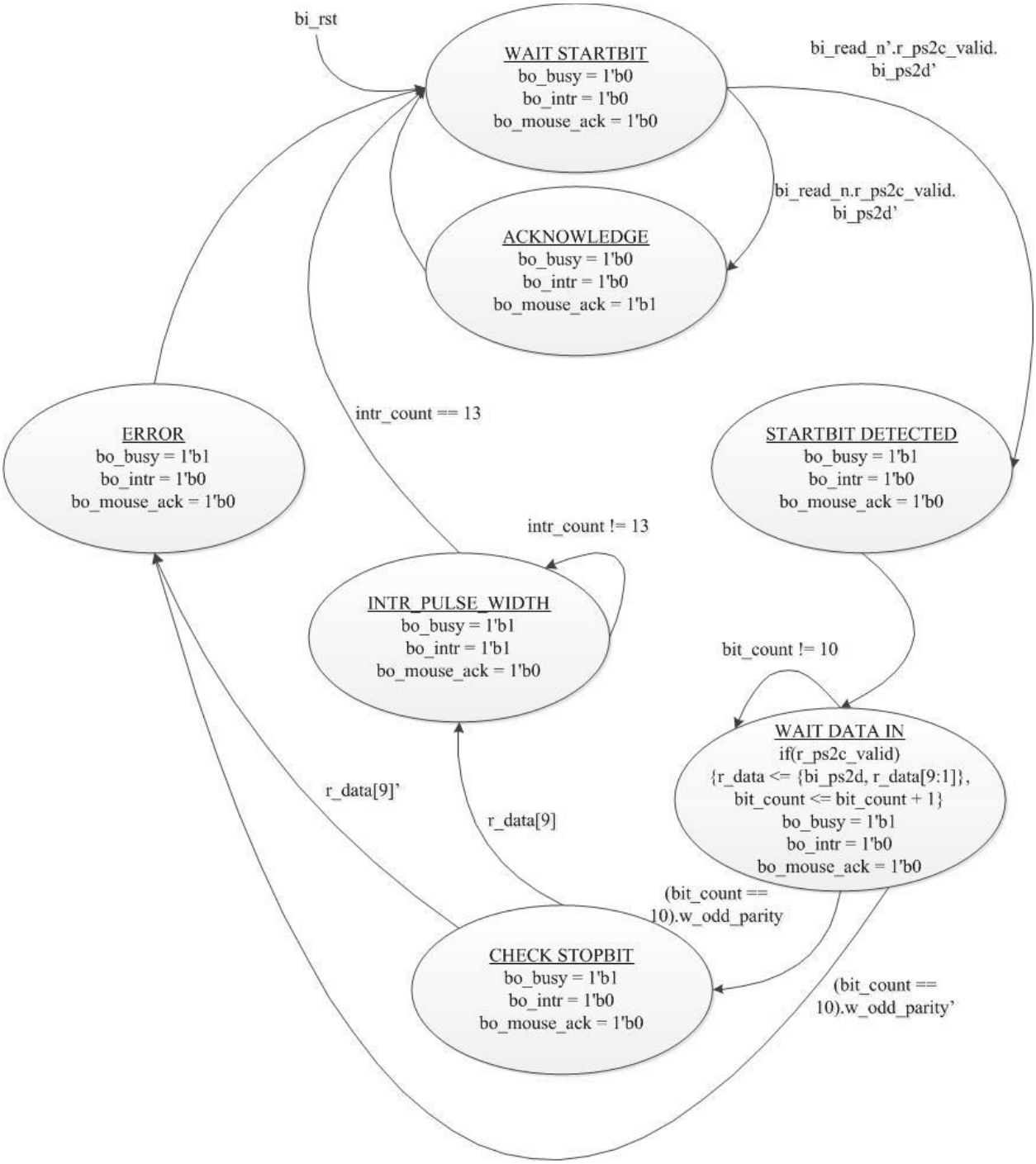


Figure 6.1.6: Receiver Block Finite State Machine

6.1.7 Simulation Results

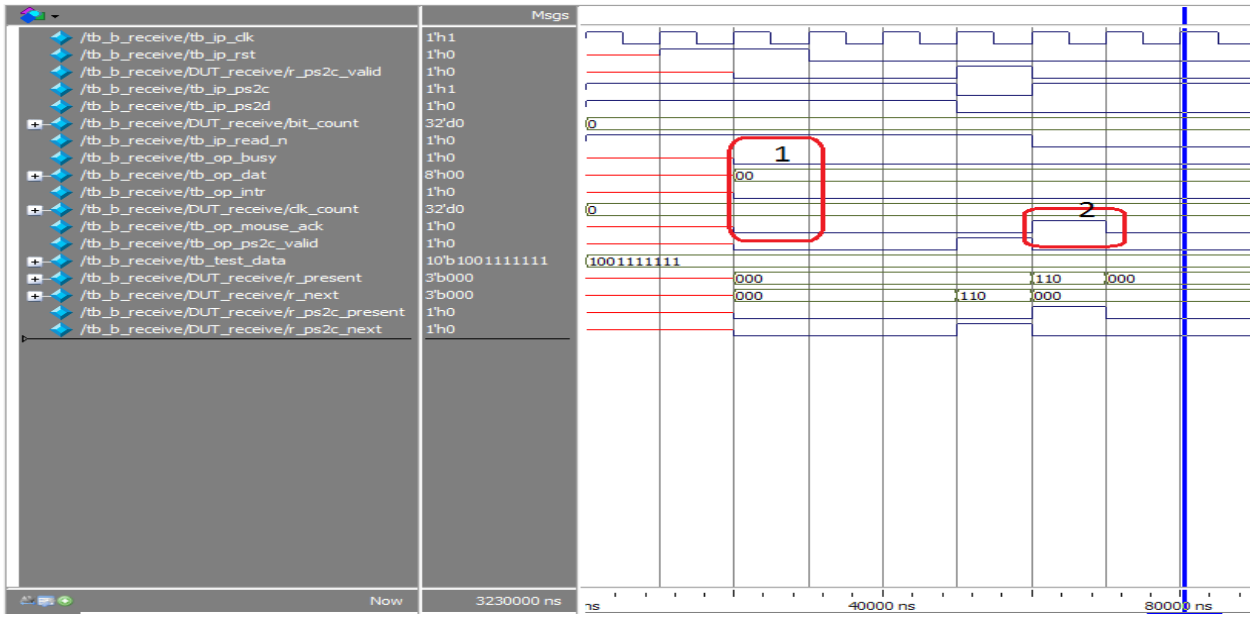


Figure 6.1.7.1: Receiver Block simulation result (a)

At (1), bi_rst is asserted.

At (2), bo_mouse_ack is asserted.

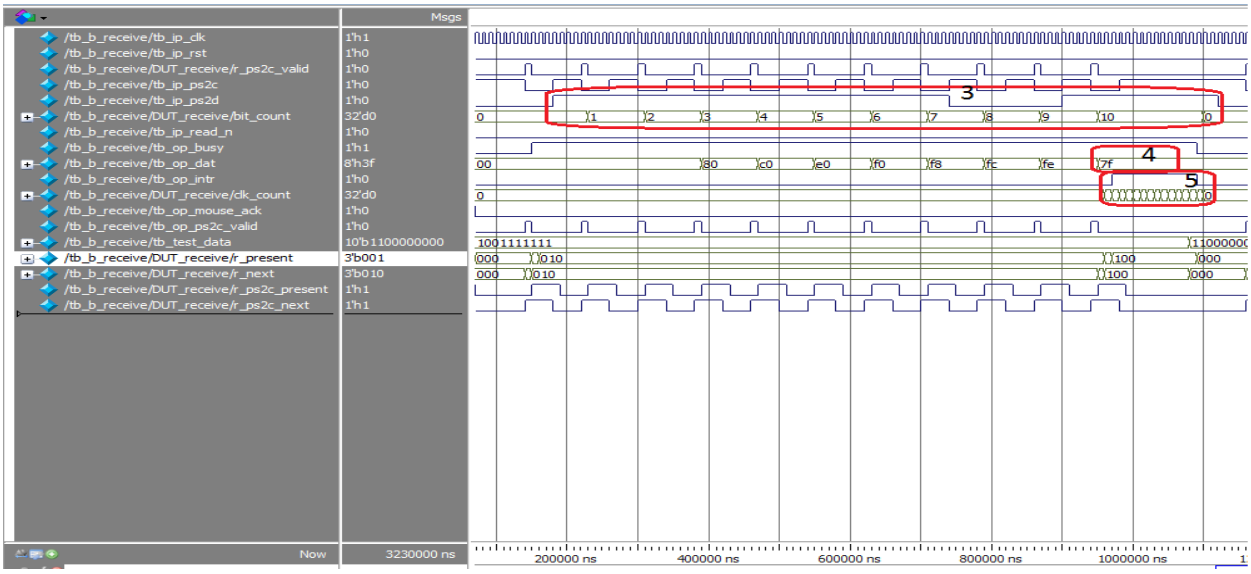


Figure 6.1.7.2: Receiver block simulation result (b)

At (3), data is being received into bi_ps2d

At (4), 8-bits data is output

At (5), bo_intr is asserted for 13 clock cycles.

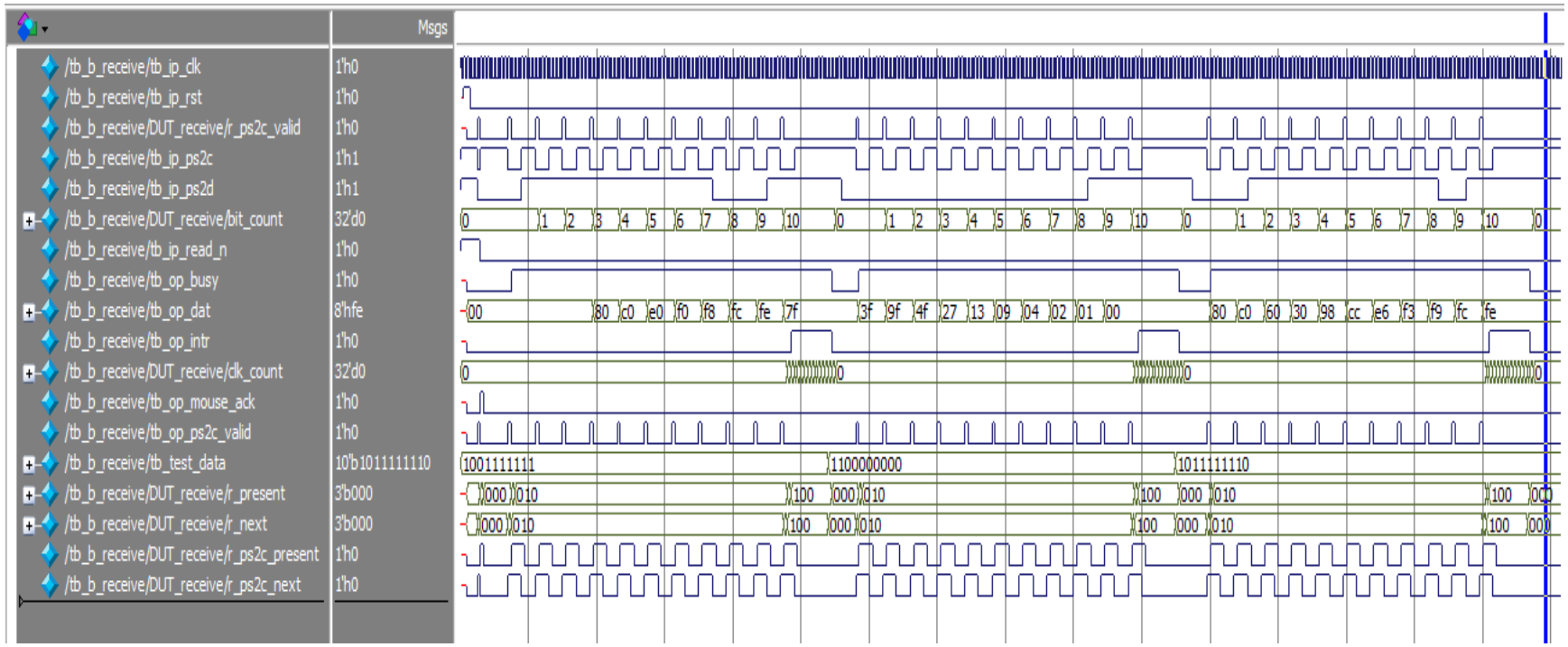


Figure 6.1.7.3: Receiver block simulation result (c)

Figure shows the full data transmission of the Receiver block. The Receiver block has successfully received 3 data packets and able to output an interrupt signals for 13 clock cycles for each transmission.

6.2 The Transmitter Block

6.2.1 Transmitter Block Interface

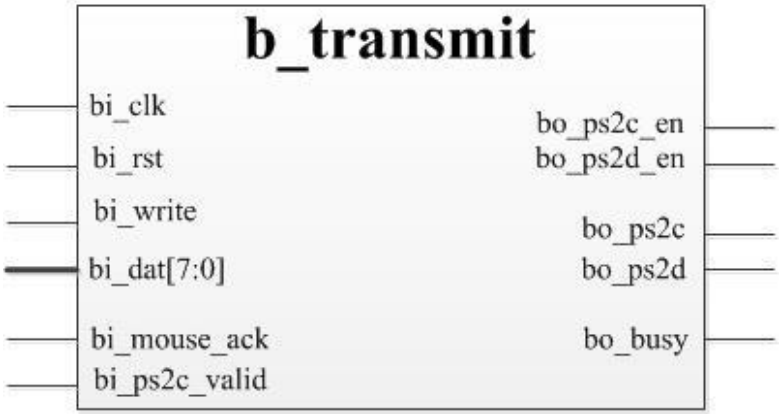


Figure 6.2.1: Transmitter Block interface

6.2.2 Input pin description

Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_clk	External Source → b_transmit	1 bit	High	No
Pin Function: Clock signal for b_transmit.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_rst	External Source → b_transmit	1 bit	High	No
Pin Function: Reset signal for b_transmit. When asserted, b_transmit will be in idle mode.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_write	b_wb_if → b_transmit	1 bit	High	No
Pin Function: Enable signal to enable b_transmit to send command to Mouse Controller.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_dat[7:0]	b_wb_if → b_transmit	8 bits	High	No
Pin Function: Data signal for b_transmit to receive the command from b_wb_if.				
Pin Name:	Source → Destination:	Size:	Active:	Registered:
bi_mouse_ack	b_receive → b_transmit	1 bit	High	No

Pin Function: Status signal to inform b_transmit that the Mouse Controller has received the command b_transmit has transmitted. It is asserted when b_receive receives an acknowledge signal from Mouse Controller.				
Pin Name: bi_ps2c_valid	Source → Destination: b_receive → b_transmit	Size: 1 bit	Active: High	Registered: No
Pin Function: Input valid signal from b_receive when detected a negative edge of bi_ps2c.				

Table 6.2.2: Transmitter Block Input Pin Description

6.2.3 Output pin description

Pin Name: bo_ps2c_en	Source → Destination: b_transmit → External	Size: 1 bit	Active: High	Registered: No
Pin Function: Tri-state enable signal for bidirectional clock signal between Mouse Controller and b_transmit.				
Pin Name: bo_ps2d_en	Source → Destination: b_transmit → External	Size: 1 bit	Active: High	Registered: No
Pin Function: Tri-state enable signal for bidirectional data signal between Mouse Controller and b_transmit				
Pin Name: bo_ps2c	Source → Destination: b_transmit → Mouse Controller	Size: 1 bit	Active: High	Registered: No
Pin Function: Clock output signal to Mouse Controller.				
Pin Name: bo_ps2d	Source → Destination: b_transmit → Mouse Controller	Size: 1 bit	Active: High	Registered: No
Pin Function: Data output signal to Mouse Controller.				
Pin Name: bo_busy	Source → Destination: b_transmit → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Status signal to inform b_wb_if that a WRITE cycle is in progress. When asserted, b_receive				

should be in idle mode.

Table 6.2.3: Transmitter Block Output Pin Description

6.2.4 Functionalities and Features

1. Transmit 11 bits of command to PS/2 Mouse.

2. Control the bidirectional clock signal and data signal between Mouse Controller and b_transmit.

3. Able to receive an acknowledgement signal from Receiver Block to signify that the previous command has been successfully received from PS/2 Mouse.

4. Able to receive a valid signal from Receiver Block to detect a negative edge of PS/2 clock signal to send data.

6.2.5 Internal Operation

Test Case	Test Function	Test Vector	Expected results
1. Assert high bi_rst for 2 clock cycles and de-assert low.	To initialize the transmitter block to idle state.	• bi_rst is asserted	bo_ps2c_en = 1’b0 bo_ps2d_en = 1’b0 bo_ps2c = 1’b1 bo_ps2d = 1’b1 bo_busy = 1’b0
2. Assert high bi_write	To enable transmitter block to inhibit transmission with PS/2 mouse for 10 clock cycles	• bi_write is asserted	bo_ps2c_en = 1’b1 bo_ps2d_en = 1’b1 bo_ps2c = 1’b0 bo_ps2d = 1’b1 bo_busy = 1’b1

3. De-assert low bi_ps2c_valid for 8 clock cycles then assert high bi_ps2c_valid. Repeat it for 10 times.	To enable transmitter block to send Start Bit , 8 data bits and parity bit to PS/2 mouse	• repeat(11) begin @(posedge bi_clk)
		bi_ps2c_valid = 1'b1;
		repeat(7) @(posedge bi_clk)
		bi_ps2c_valid = 1'b0;
		end
			bo_ps2c_en = 1’b0
			bo_ps2d_en = 1’b1
			bo_ps2c = bo_ps2c
			bo_ps2d = bi_dat[bit_count]
			bo_busy = 1’b1

4. Assert high bi_mouse_ack after sending the stop bit.	To notice transmitter block that PS/2 mouse received the command successfully	• bi_mouse_ack is asserted after transmitting stop bit	bo_ps2c_en = 1’b0 bo_ps2d_en = 1’b0 bo_ps2c = 1’b1 bo_ps2d = 1’b1 bo_busy = 1’b0

Table 6.2.5: Internal operation for Transmitter Block

6.2.6 Finite State Machine

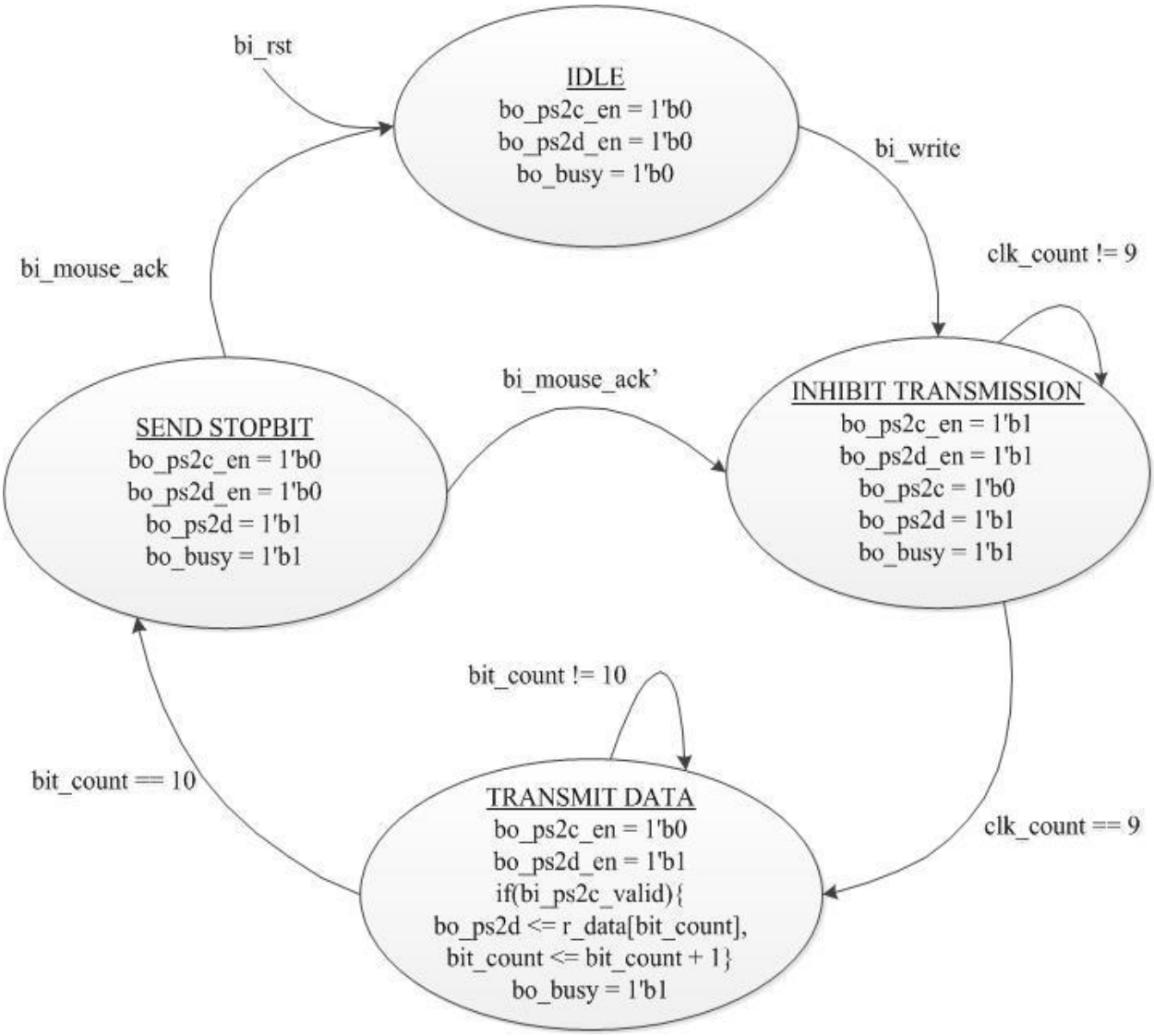


Figure 6.2.6: Transmitter Block Finite State Machine

6.2.7 Simulation result

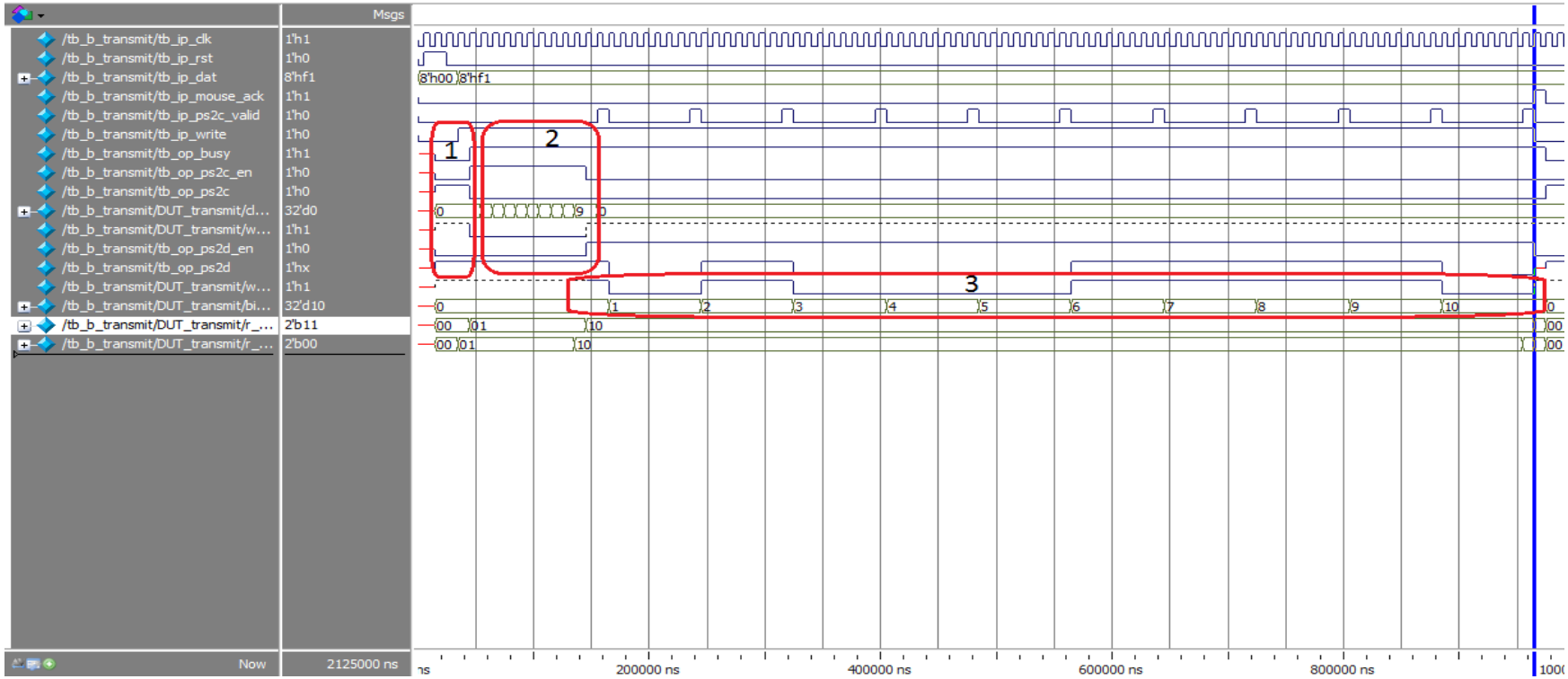


Figure 6.2.7.1: Transmitter Block simulation result (a)

At (1), bi_rst is asserted. All output signals are initialized.

At (2), bi_write is asserted. bo_ps2c is pull down as 1'b0 to inhibit transmission.

At (3), data is being transmitted.

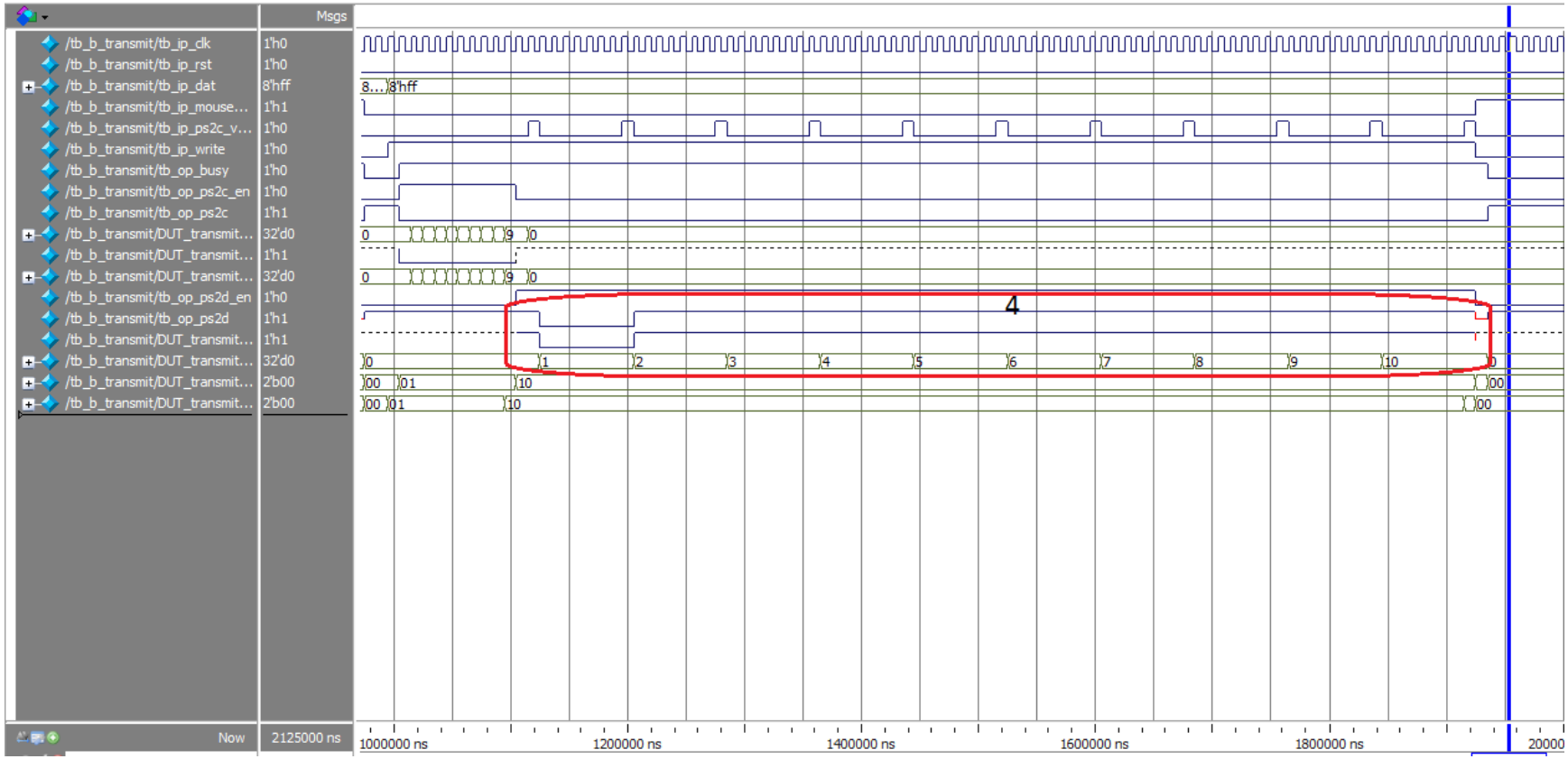


Figure 6.2.7.2: Transmitter block simulation result (b)

At (4), second data is being transmitted.

6.3 The WISHBONE interface Block

6.3.1 WISHBONE interface Block interface

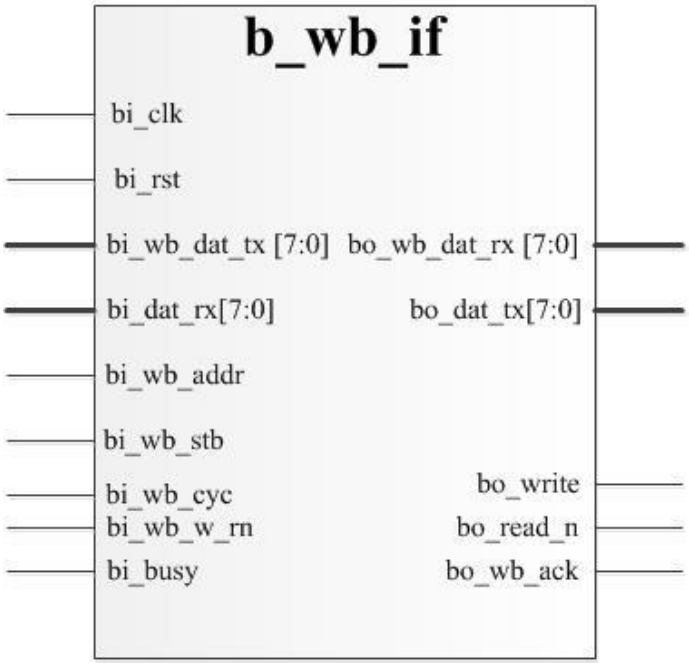


Figure 6.3.1: WISHBONE interface Block interface

6.3.2 Input pin description

Pin Name: bi_clk	Source → Destination: External Source → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Clock signal for b_wb_if.				
Pin Name: bi_rst	Source → Destination: External Source → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Reset signal for b_wb_if. When asserted, b_wb_if will be in idle mode.				
Pin Name: bi_wb_dat_tx[7:0]	Source → Destination: Datapath Unit → b_wb_if	Size: 8 bits	Active: High	Registered: Yes
Pin Function: Data input signal for b_wb_if. It receives command data from Datapath Unit and sends it to b_transmit when it is in WRITE cycle.				

Pin Name: bi_dat_rx[7:0]	Source → Destination: b_receive → b_wb_if	Size: 8 bits	Active: High	Registered: Yes
Pin Function: Data input signal for b_wb_if. It receives 8 bits of data from b_receive when it is in a READ cycle.				
Pin Name: bi_wb_addr	Source → Destination: Datapath unit → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Address bus signal, used to select an internal register of the device from: Asserted = WISHBONE Data register (WDREG) De-asserted = WISHBONE Control register (WDREG)				
Pin Name: bi_wb_stb	Source → Destination: External Source → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Strobe signal for b_wb_if. When asserted, indicates that it is selected to respond to other WISHBONE signals. (except for reset signal)				
Pin Name: bi_wb_cyc	Source → Destination: External Source→ b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Cycle signal for b_wb_if. When asserted, indicates the start of a valid WISHBONE data transfer cycle.				
Pin Name: bi_wb_w_rn	Source → Destination: Control Path Unit → b_wb_if	Size: 1 bit	Active: High	Registered: No
Pin Function: Write enable signal for b_wb_if. Used to indicate whether current bus cycle is a Read or Write Cycle. Asserted = Write De-asserted = Read				
Pin Name:	Source → Destination:	Size:	Active:	Registered:

bi_busy	b_trasnmit, b_receive → b_wb_if	1 bit	High	No
Pin Function: Status signal from b_transmit and b_receive to b_wb_if.When asserted, indicates that either one is performing a Write or Read cycle.				

Table 6.3.2: WISHBONE interface Block Input Pin Description

6.3.3 Output pin description

Pin Name: bo_wb_dat_rx[7:0]	Source → Destination: b_wb_if → Datapath Unit	Size: 8 bits	Active: High	Registered: Yes
Pin Function: Data output signal. It sends 8 bits of data received from b_receive to Datapath Unit. If the parity bit received is incorrect, the data will not be sent.				
Pin Name: bo_dat_tx[7:0]	Source → Destination: b_wb_if → b_transmit	Size: 8 bits	Active: High	Registered: Yes
Pin Function: Data output signal. It sends 8 bits of command to b_transmit.				
Pin Name: bo_write	Source → Destination: b_wb_if → b_transmit	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Enable signal to start a valid WRITE cycle.				
Pin Name: bo_read_n	Source → Destination: b_wb_if → b_receive	Size: 8 bits	Active: Low	Registered: Yes
Pin Function: Enable signal to start a valid READ cycle.				
Pin Name: bo_wb_ack	Source → Destination: b_wb_if →	Size: 1 bit	Active: High	Registered: No
Pin Function: Standard WISHBONE acknowledgement signal. Asserted = the PS/2 controller has finished execution of the requested action and the current bus				

cycle is terminated.

Table 6.3.3: WISHBONE interface Block Output Pin Description

6.3.4 Functionalities and Features

1. Transmit 8 bits of command from processor to the Transmitter Block.
2. Receive 8 bits of data from Receiver Block to the processor.

6.3.5 Internal Operation

Test Case	Test Function	Test Vector	Expected results
1. Assert high bi_rst for 2 clock cycles and de-assert low	To initialize WISHBONE interface Block to idle state.	• bi_rst is asserted	• bo_write = 1’b0 • bo_read_n = 1’b0 • bo_wb_ack = 1’b0
2. Assert high bi_wb_addr, bi_wb_cyc and bi_wb_stb while de-assert low bi_wb_w_rn and bi_busy	To enable WISHBONE interface Block to enter READ cycle	• bi_wb_addr = 1'b1; • bi_wb_w_rn = 1'b0; • bi_busy = 1'b1; • bi_wb_cyc = 1'b1; • bi_wb_stb = 1'b1; • bi_dat_rx = 8'b1111_1010	• bo_wb_dat_rx = bi_dat_rx • bo_write = 1’b0 • bo_read_n = 1’b1 • bo_wb_ack = 1’b1
3. Assert high bi_addr, bi_cyc, bi_stb and bi_w_rn	To enable WISHBONE interface Block to enter WRITE cycle	• bi_wb_addr = 1'b1; • bi_wb_w_rn = 1'b1; • bi_busy = 1'b1; • bi_wb_cyc = 1'b1; • bi_wb_stb = 1'b1; • bi_wb_dat_tx = 8'b1111_0101;	• bo_dat_tx = bi_wb_dat_tx • bo_write = 1’b1 • bo_read_n = 1’b0 • bo_wb_ack = 1’b1

Table 6.3.5: Internal operation for WISHBONE interface Block

6.3.6 Finite State Machine

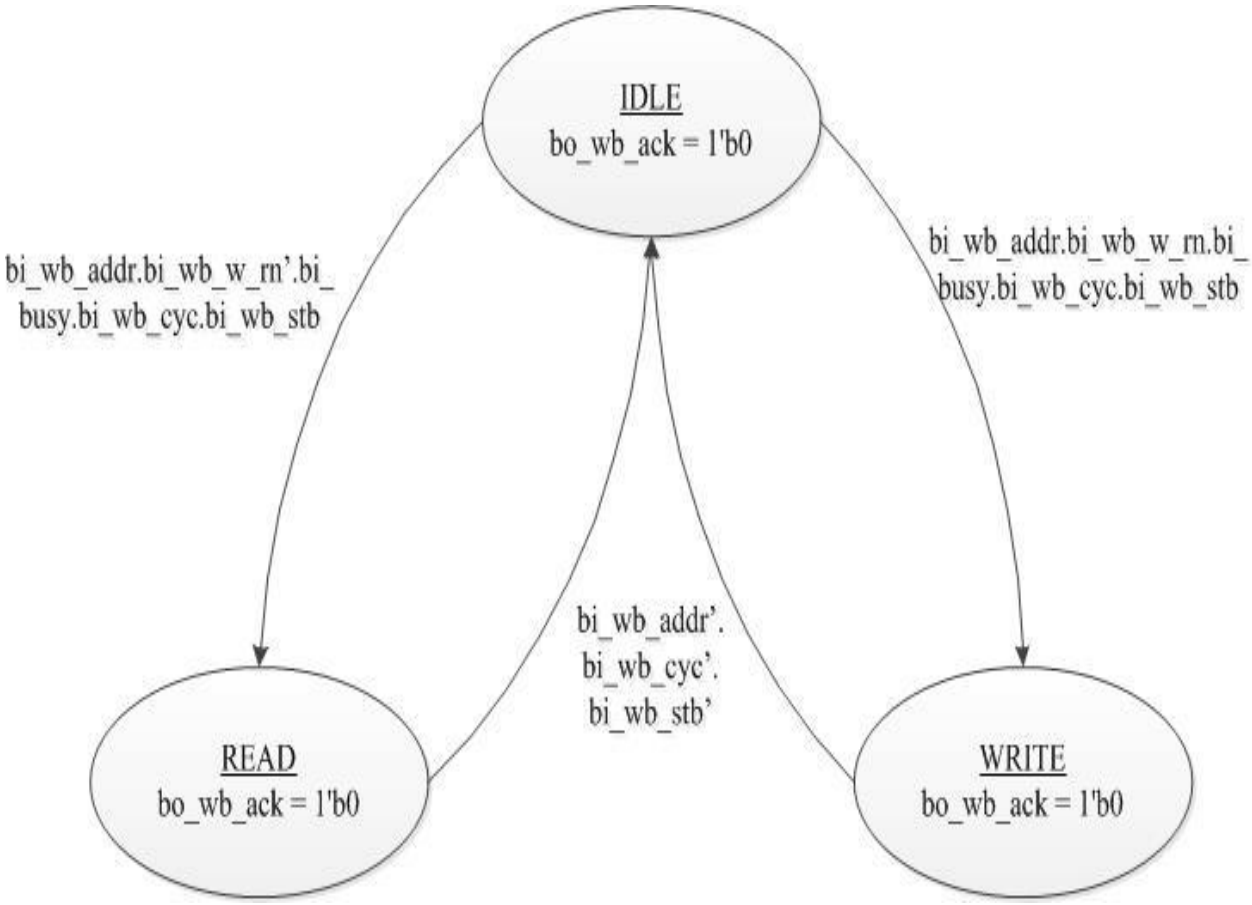


Figure 6.3.6: WISHBONE interface Block Finite State Machine

6.3.7 Simulation results

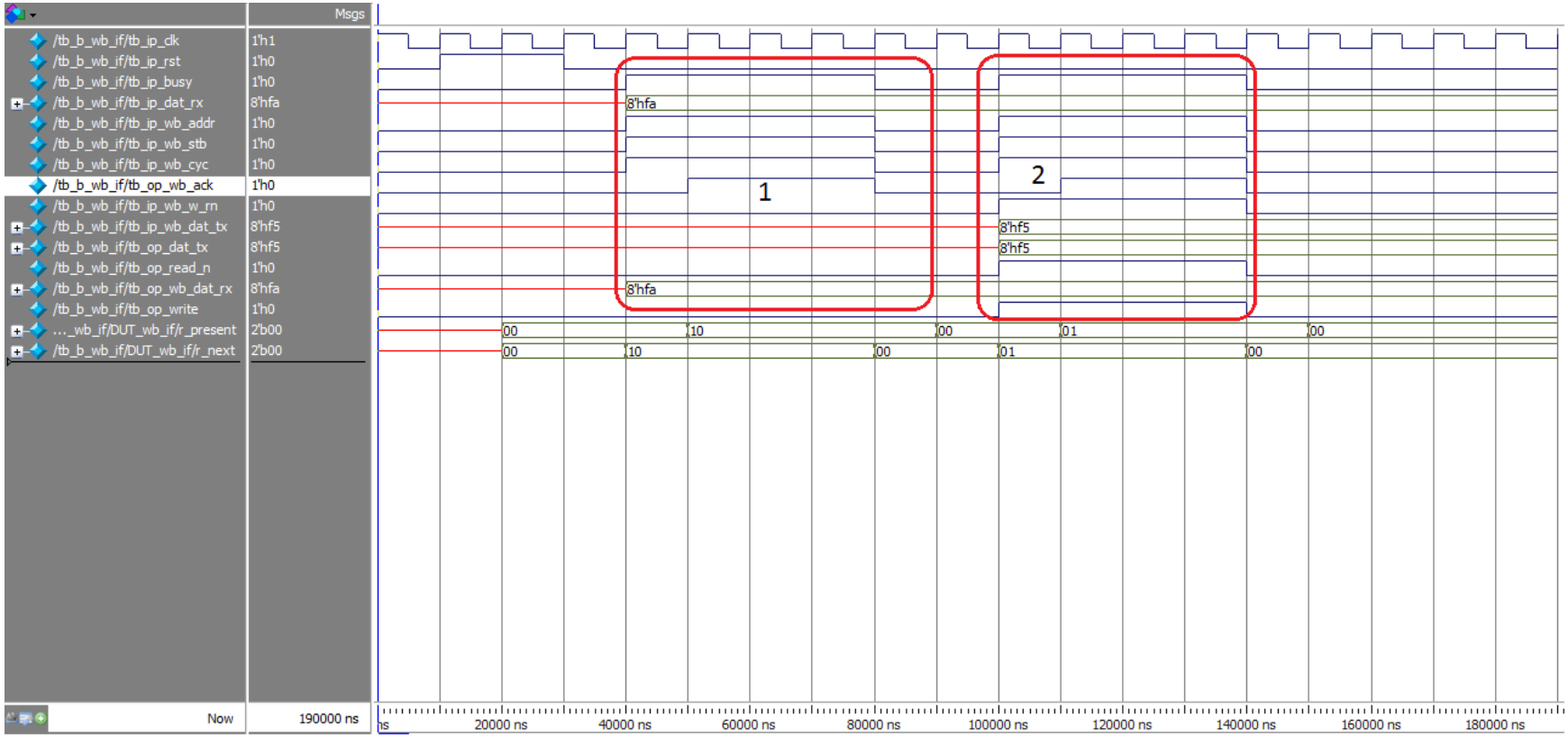


Figure 6.3.7: WISHBONE interface Block simulation result

At (1), READ cycle begins

At (2), WRITE cycle begins

6.4 The address decoder

6.4.1 Address decoder interface

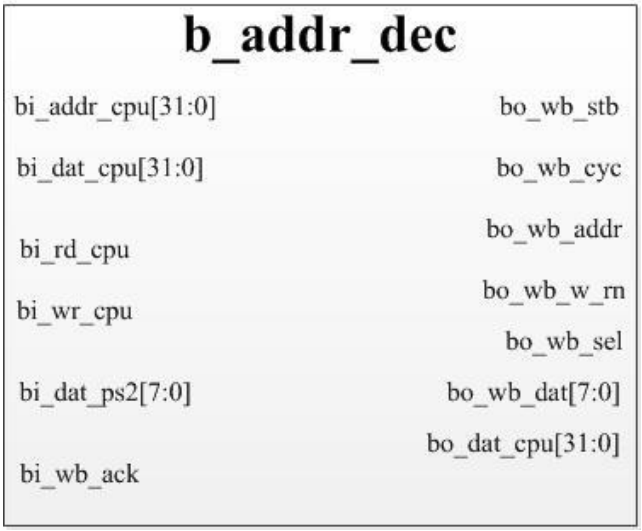


Figure 6.4.1: Address decoder interface

The address decoder decodes the control signals from CPU (bi_addr_cpu, bi_rd_cpu, bi_wr_cpu) and WISHBONE interface Block (bi_wb_ack) to output the appropriate WISHBONE compatible signals to the WISHBONE interface Block.

Input Signals			**Output signal**
bi_wb_ack	bi_rd_cpu	bi_wr_cpu	bo_wb_w_rn
1	x	x	x
0	0	0	x
0	0	1	1
0	1	0	0
0	1	1	x

Table 6.4.1.1: WISHBONE write/read_not signal decoding table

Input signal	**Output signals**		
bi_addr_cpu	bo_wb_stb	bo_wb_cyc	bo_wb_addr
<260 or >260	0	0	0
260	1	1	1

Table 6.4.1.2: WISHBONE output signal decoding table

6.4.2 Input pin description

Pin Name: bi_addr_cpu	Source → Destination: RISC32CPU → b_addr_dec	Size: 32 bit	Active: High	Registered: No
Pin Function: 32-bit address signal from CPU				
Pin Name: bi_dat_cpu	Source → Destination: RISC32CPU →b_addr_dec	Size: 32 bit	Active: High	Registered: No
Pin Function: 32-bit data signal from CPU				
Pin Name: bi_rd_cpu	Source → Destination: RISC32CPU → b_addr_dec	Size: 1 bit	Active: High	Registered: No
Pin Function: Read signal from CPU				
Pin Name: bi_wr_cpu	Source → Destination: RISC32CPU → b_addr_dec	Size: 1 bit	Active: High	Registered: No
Pin Function: Write signal from CPU				
Pin Name: bi_dat_ps2	Source → Destination: b_wb_if → b_addr_dec	Size: 8 bit	Active: High	Registered: No
Pin Function: 8-bit data signal from PS/2 Controller				
Pin Name: bi_wb_ack	Source → Destination: b_wb_if → b_addr_dec	Size: 1 bit	Active: Low	Registered: No
Pin Function: WISHBONE acknowledge signal from PS/2 Controller				

Table 6.4.2: Address Decoder Input Pin Description

6.4.3 Output pin description

Pin Name: bo_wb_stb	Source → Destination: b_addr_dec → b_wb_if	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Strobe output signal to b_wb_if				
Pin Name: bo_wb_cyc	Source → Destination: b_addr_dec → b_wb_if	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Cycle output signal to b_wb_if				
Pin Name: bo_wb_addr	Source → Destination: b_addr_dec → b_wb_if	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Address output signal to b_wb_if				
Pin Name: bo_wb_w_rn	Source → Destination: b_addr_dec → b_wb_if	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Write enable output signal to b_wb_if. Used to indicate whether current bus cycle is a Read or Write Cycle. Asserted = Write De-asserted = Read				
Pin Name: bo_wb_sel	Source → Destination: b_addr_dec → b_wb_ic	Size: 1 bit	Active: High	Registered: Yes
Select enable output signal to b_wb_ic. Used to determine which PS/2 external device is selected to perform transaction. Asserted = PS/2 keyboard is selected De-asserted = PS/2 mouse is selected				
Pin Name: bo_wb_dat	Source → Destination: b_addr_dec → b_wb_if	Size: 8 bit	Active: High	Registered: Yes
Pin Function: 8-bit data signal to b_wb_if				

Pin Name:	Source → Destination:	Size:	Active:	Registered:
bo_dat_cpu	b_addr_dec → CPU	32 bit	High	Yes
Pin Function: 32-bit data signal to RISC32PU				

Table 6.4.3: Address Decoder Output Pin Description

6.4.4 Functionalities and Features

1. Decodes the input signals from RISC32 CPU and output WISHBONE compatible signals to the WISHBONE interface Block of the PS/2 Controller.
2. Receives data from PS/2 Controller and sends to the memory according to the designated address.

6.4.5 Microarchitecture of address decoder

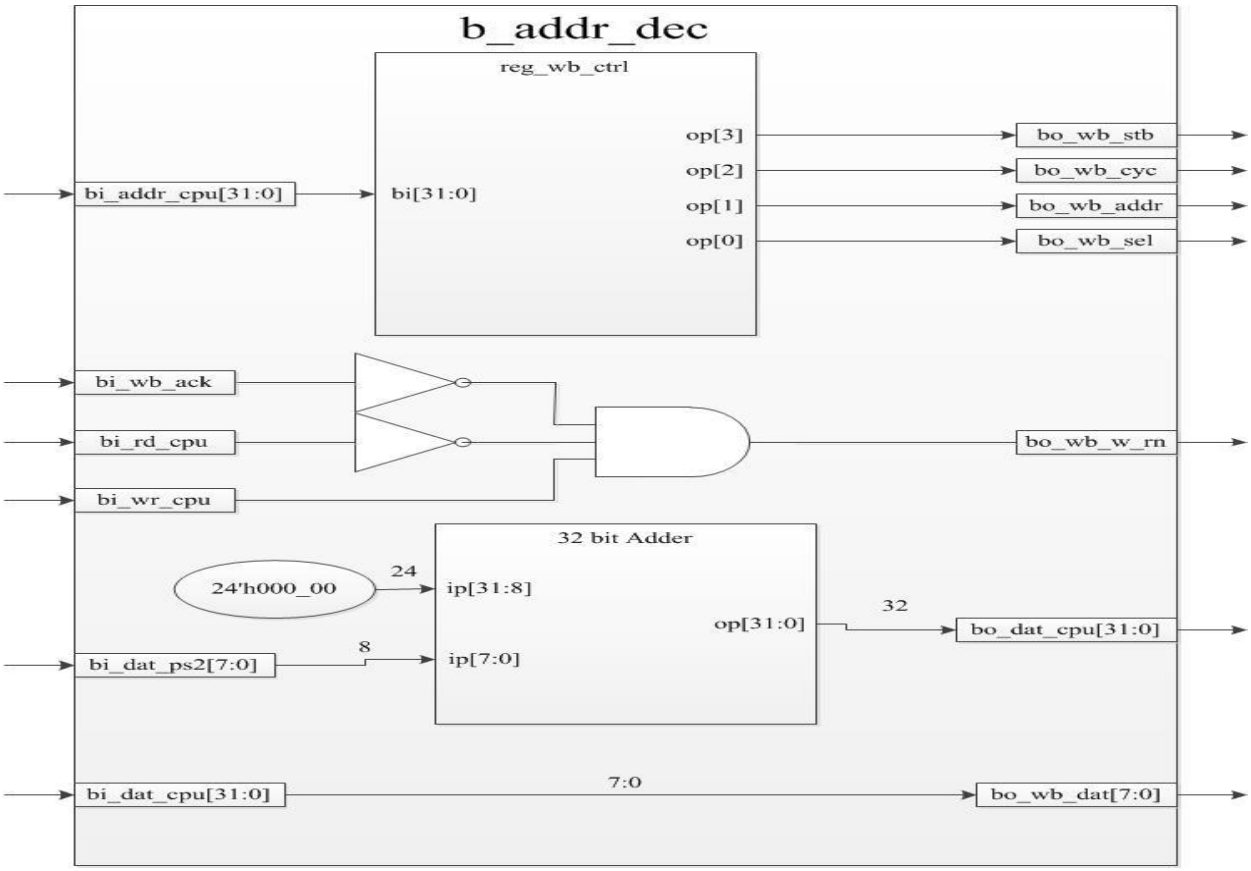


Figure 6.4.5: Microarchitecture for address decoder

6.4.6 Internal Operation

Test Case	Test Function	Test Vector	Expected results
1. Assert high bi_wr_cpu and other input signals	To enable address decoder to write data into the WISHBONE interface Block	• bi_addr_cpu = CPU_DATA_ADDR; • bi_rd_cpu = 1'b0; • bi_wr_cpu = 1'b1; • bi_wb_ack = 1'b0; • bi_dat_cpu = 32'h0000_00FF;	• bo_wb_stb = 1’b1 • bo_wb_cyc = 1’b1 • bo_wb_addr = 1’b1 • bo_wb_w_rn = 1’b1 • bo_wb_dat = 8’hFF
2. Assert low bi_rd_cpu and assert high other input signals	To enable address decoder to read data from WISHBONE interface Block	• bi_addr_cpu = CPU_DATA_ADDR; • bi_rd_cpu = 1'b0; • bi_wr_cpu = 1'b1; • bi_wb_ack = 1'b0; • bi_dat_ps2 = 8’hFA;	• bo_wb_stb = 1’b1 • bo_wb_cyc = 1’b1 • bo_wb_addr = 1’b1 • bo_wb_w_rn = 1’b0 • bo_dat_cpu = 32’h0000_00FA

Table 6.4.6: Internal operation for Address Decoder

6.5 The Synchronizer Block

6.5.1 Synchronizer Block interface

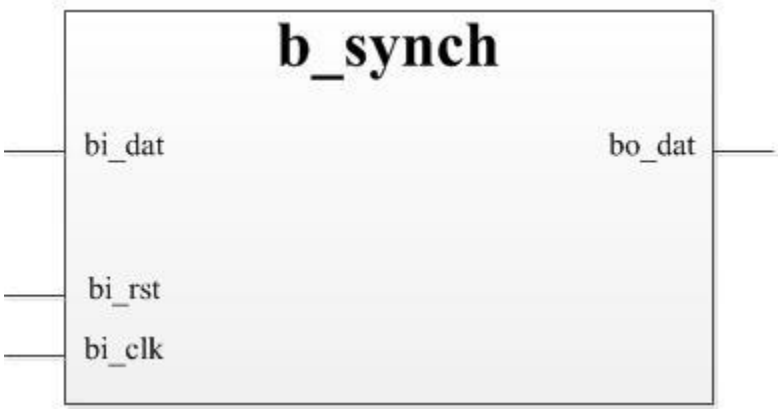


Figure 6.5.1: Synchronizer Block interface

6.5.2 Input pin description

Pin Name: bi_clk	Source → Destination: External Source → b_synch	Size: 1 bit	Active: High	Registered: No
Pin Function: Clock signal for b_synch				
Pin Name: bi_rst	Source → Destination: External Source → b_synch	Size: 1 bit	Active: High	Registered: No
Pin Function: Reset signal for b_synch.				
Pin Name: bi_dat	Source → Destination: Datapath Unit → b_synch	Size: 1 bit	Active: High	Registered: Yes
Pin Function: Load signal into b_synch				

Table 6.5.2: Synchronizer Block Input Pin Description

6.5.3 Output pin description

Pin Name:	Source → Destination:	Size:	Active:	Registered:
bo_dat_tx	b_synch → other block module	1 bit	High	Yes
Pin Function: Synchronize output signal				

Table 6.5.3: Synchronizer Block Output Pin Description

6.5.4 Functionalities and Features

1. Synchronize all input signals.
2. Filter glitches.

6.5.5 Internal Operation

Test Case	Function	Expected Result
Assert bi_rst. • Assert bi_rst for 3 clock cycles.	To initialise	bo_dat_tx will be initialized to 0 or initial stage. b_synch will not operate.
Assert any input signal after the TCS is reset. • Assert bi_dat for 3 clock cycles.	To test the functionality of b_synch	The input signal will be synchronized. b_synch will generate output signals to the respective block module.

Table 6.5.5: Internal operation for Synchronizer Block

Chapter 7 Verification Specification

7.1 Test Program for I/O Serial Communication

The RISC32 CPU utilizes a memory mapped I/O that reserves specific memory address from 0x8000 0000 to 0x8000 000C to communicate with I/O modules. Any I/O communication with the RISC32 CPU by input or output data from these memory locations is conducted in 32-bits in for each location. The memory location can be accessed by using instruction load word (*lw*) and store word (*sw*) to read/write data from/to I/O.

A test program is written to examine the integration of the I/O serial communication into enhanced RISC32 architecture, which the RISC32 CPU will interface with the PS/2 Controller via the address decoder. In the test program, the CPU will first send 8 bits of command to the address decoder before running a looping program to mimic user program running. After transmitting the command byte, it will receive the 3 bytes of data from the PS/2 Controller after the PS/2 Controller triggers an interrupt to the CPU. The command byte is 8'hFF and the data bytes received are: 8'hFA, 8'hAA, and 8'h00.

I/O interrupt happens when the PS/2 Controller's interrupt signal triggers the interrupt handling mechanism in CP0 block, which dispatches RISC32 CPU to jump into exception handler. The exception handler manages the software handling and it will investigate the exception causes and jump to the appropriate ISR.

ISR is the place where the interrupt is served. The instruction is issued to send data from the PS/2 Controller and places it into the register file. After the data is loaded, the data will be saved into the data memory. The "exception return" instruction (*eret*) will be called and the CPU will resume to user program as before.

7.1.1 Test Program

```
. .ktext 0x00400024
#------------------PS/2 Wish Bone Data Register Initialization-------------------------
        lw $t3, 260($s0)        #set Wish Bone data register
#------------------------CP0 Registers Initialization-------------------------------------
        lw $s1, $0, $0          #reg[s1] = 0x00000000
        mtc0 $s1, $13           #Initialize cause register, $cp0_cause = $13
        ori  $s1, $0, 0xff01    #reg[s1] = 0x0000ff01
        mtc0 $s1, $12           #Initialize status register, $cp0_status = $12
#-----------------Jump to Transmit Routine address, 0x800001d0----------------------
        lui $t9, 0x8000          #reg[t9] = 0x80000000
        ori $t9, 0x01d0          #reg[t9] = 0x800001d0
        jal $t9                  #jump to address 0x800001d0
#---------------------------------Looping Program----------------------------------------
loop:   ori $t4, $0, 0x0104      #reg[t4] = 0x00000104
        nop                      #no operation
        j loop                   #jump back to loop
#---------------------------------Exception Handler-----------------------------------------
.ktext 0x80000180
exceptionhandler:
        mfc0 $k0, $13            #move cause register to $k0
        mfc0 $k1, $12            #move status register to $k1
        nop                      #delay to prevent data hazard
        andi $t2, $k0, 0x007c #extract code Excode
        andi $k0, $k0, 0xff00 #extract cause register IP field
        and $k0, $k1, $k0        #check whether interrupt is mask
        beq $k0, $0, EXIT        #IP field is not set, ignore interrupt
        beq $t2, $0, ps2rx       #branch to external interrupt
```

```
EXIT:
        eret                    #exception return
#----------------------------------ISR----------------------------------------------
ps2rx:
        lw $s1, 260($s0)        #receive first data
        nop                     #delay
        lw $s2, 260($s0)        #receive second data
        nop                     #delay
        lw $s3, 260($s0)        #receive final data
        nop                     #delay
        sw $s1, 0($s0)          #MEM[10000000] = reg[s1]
        sw $s2, 4($s0)          #MEM[10000004] = reg[s2]
        sw $s3, 8($s0)          #MEM[10000008] = reg[s3]
#----------------------------------Transmit Data-------------------------------------
ps2tx:
        ori $t6, $0, 0x00ff     #reg[t6] = 0x000000ff
        sw $t6, 260($s0)        #MEM[00000104] = reg[t6]
        jr $ra,                 #jump back to return address
```

7.2 Verification Result

7.2.1 Register content in Register File

Figure below describes the content in Register File of RISC32 CPU:

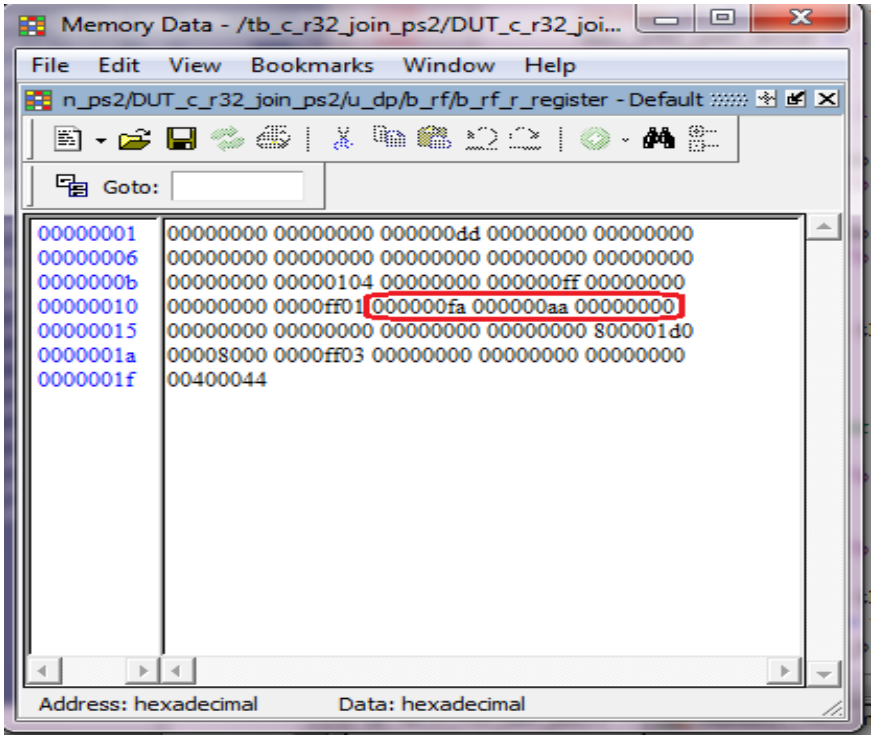


Figure 7.2.1: Register content in Register File

7.2.2 Data memory content in data memory

Figure below shows the data memory content of RISC32 CPU:

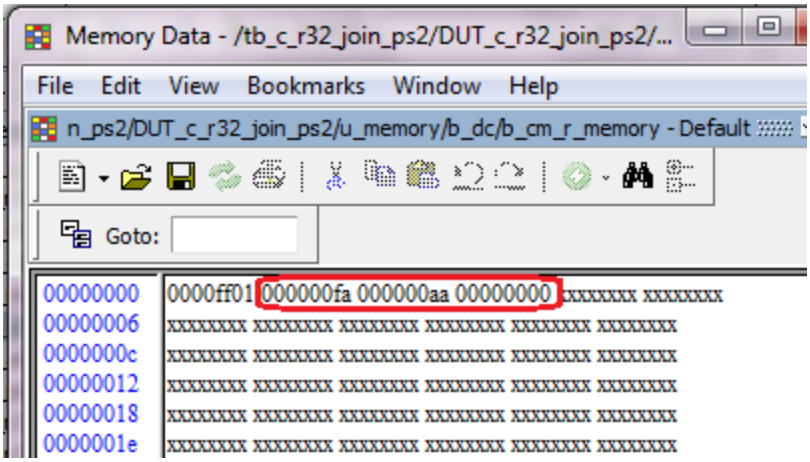


Figure 7.2.2: Data memory content in data memory

7.2.3 Waveform and explanation for RISC32 interrupt

The figure below shows the waveform when interrupt handling occurs:

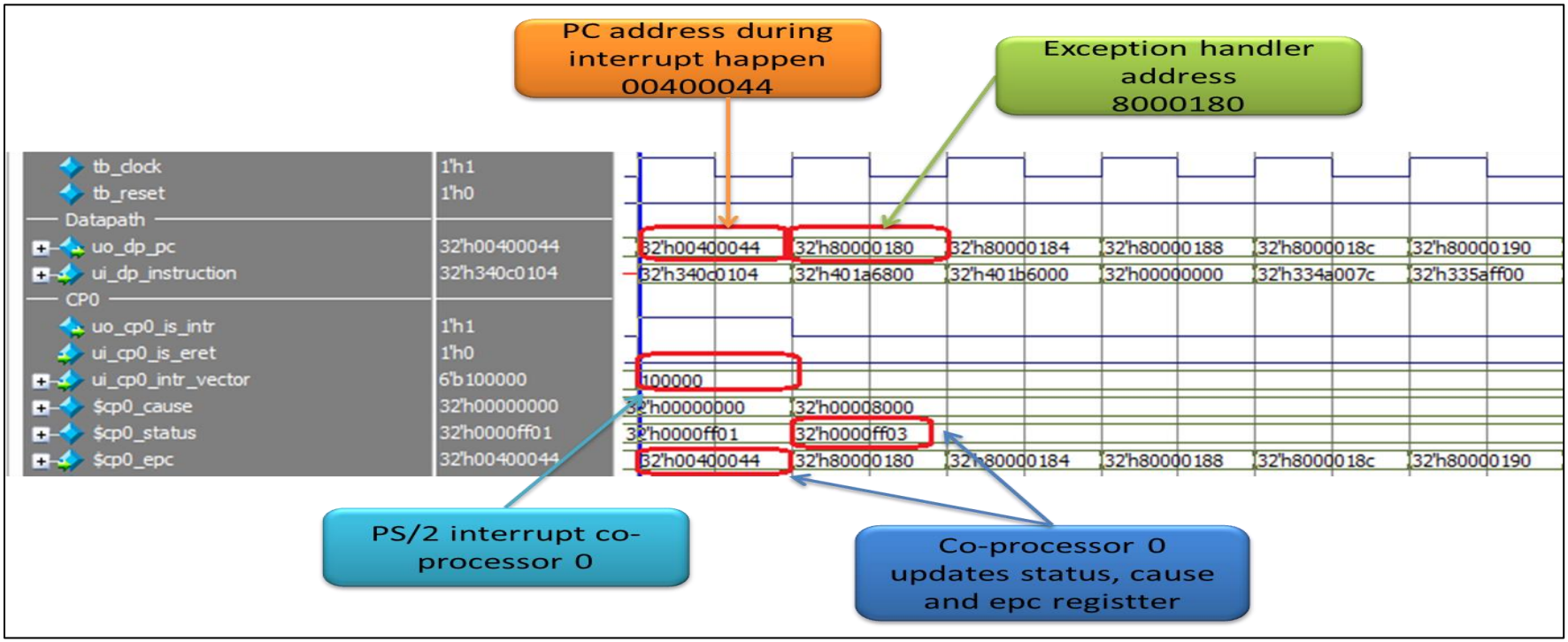


Figure 7.2.3.1: RISC32 interrupt handling process

When the PS/2 Controller sends an interrupt to the RISC32 CPU, the Co-processor 0 will check the status register ($cp0_status) bit [0] to check whether interrupt is enabled. CP0 will then update the status register into kernel mode, update the exception code in cause register ($cp0_cause) and store the next PC address in the EPC register ($cp0_epc). This will cause the RISC32 CPU to jump to the exception handler address which is 80000180.

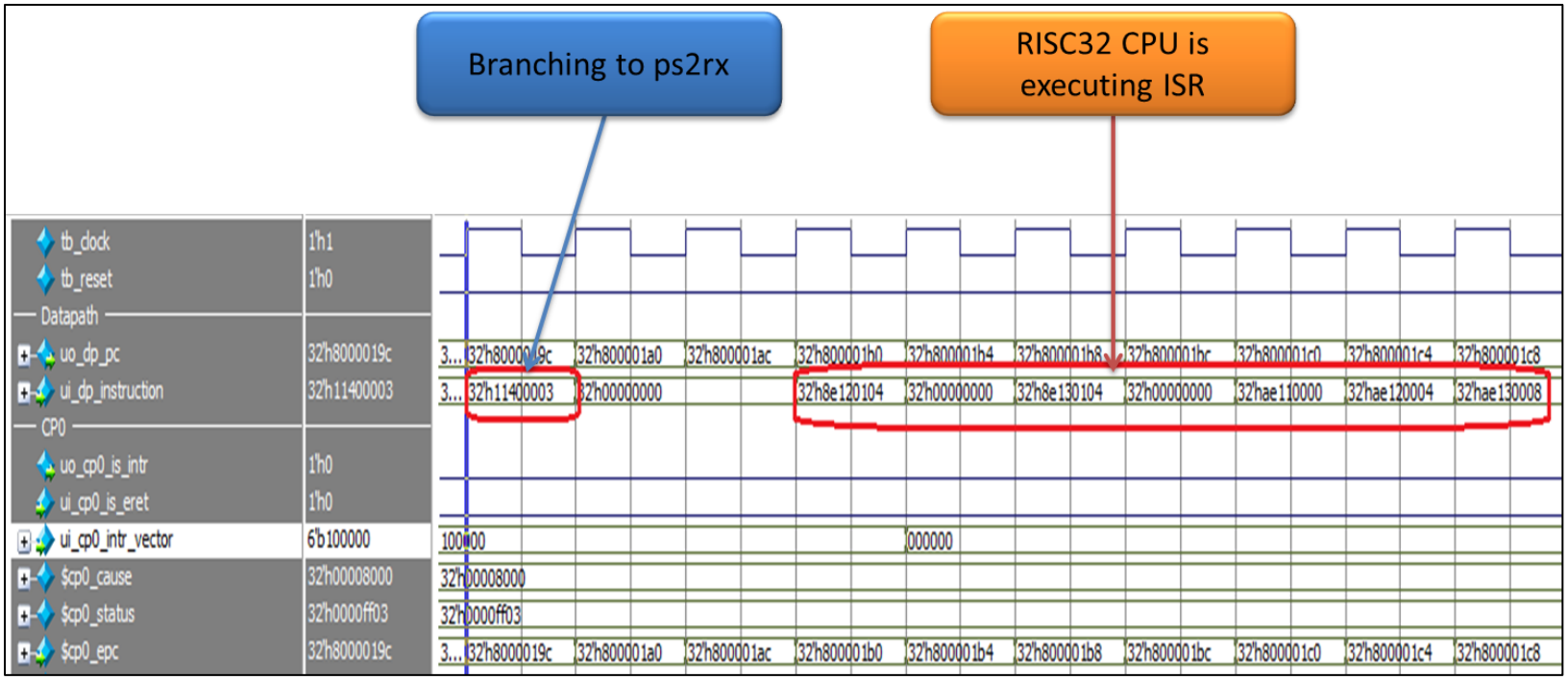


Figure 7.2.3.2: Exception handler jumps to appropriate ISR

Based on the figure above, the exception handler will extract the exception code and examine the interrupt pending bit to check whether it should ignore or run the interrupt, then jump to specific ISR based on exception code in cause register.

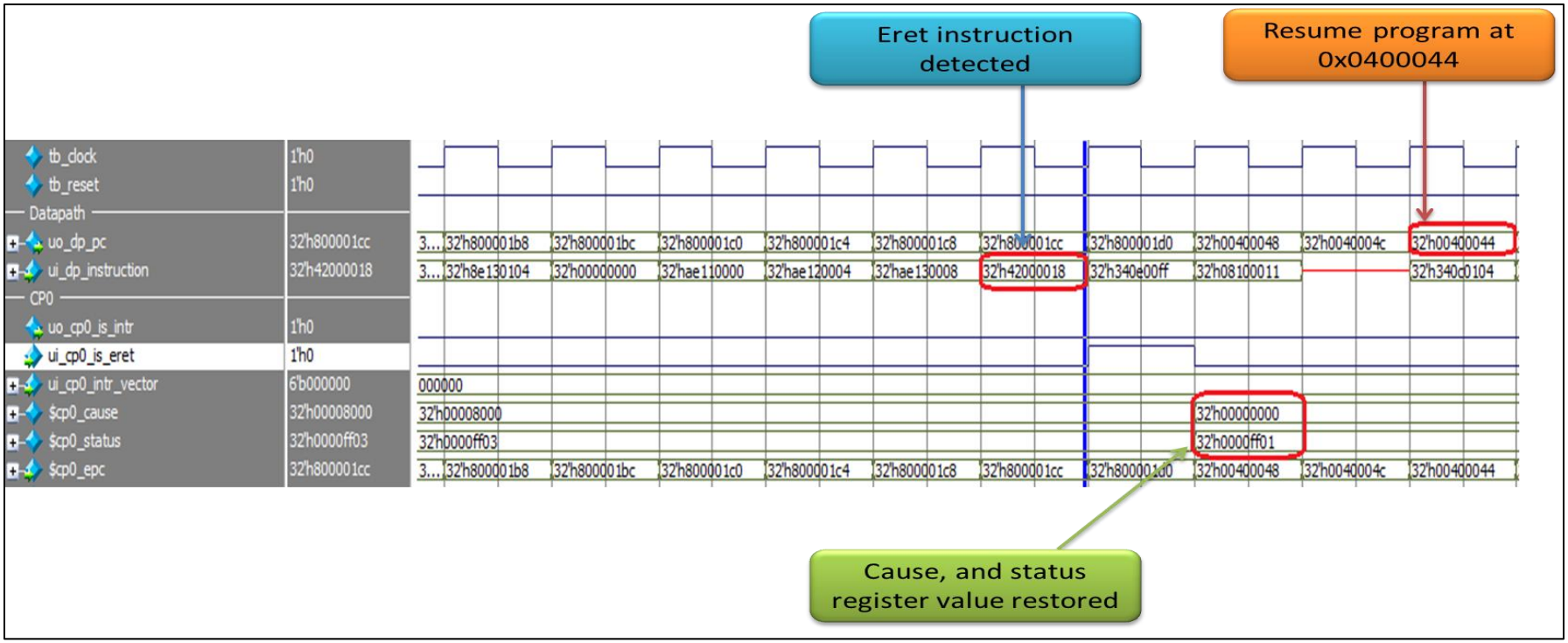


Figure 7.2.3.3: Return from exception

When the RISC32 CPU has finished executing the interrupt handling and start executing the instruction *eret*, Co-processor 0 will update $cp0_status by switching to kernel mode to user mode in bit [1], clearing the interrupt pending bit and exception code in $cp0_cause and finally will output the PC value in $cp0_epc to main core. The program will jump back to the PC value based on the value in $cp0_epc to resume program.

Chapter 8 Conclusion and Discussion

8.1 Conclusion

A PS/2 Controller has been successfully modelled and tested its functionality. With an address decoder designed to produce WISHBONE compatible output signals to the PS/2 Controller, both of them have been integrated into the RISC32 architecture. Hence, the 32-bit RISC CPU can communicate with the PS/2 Controller using instructions *lw* to transmit command to the PS/2 Controller and *sw* to receive 8 bits of data from the PS/2 Controller. The I/O serial communication follows the protocol mentioned in Chapter 2 of this project.

The integration of PS/2 Controller into enhanced RISC32 architecture has been accomplished, as shown in Chapter 4 In addition; the address decoder for the PS/2 Controller was modelled using Verilog HDL based on the developed microarchitecture (block level) specifications as shown in Chapter 6. The full integration verification has also completed as shown in Chapter 6. Moreover, the software handling part, which are the Exception Handler and also the Interrupt Service Routine (ISR) are proved to be working. The data from I/O was successfully transferred to the memory.

Based on the following table, the list of objectives stated in Chapter 1 has been achieved:

Objectives	Status
Study of existing PS/2 architecture	Enhanced
Development of the RTL model of the PS/2 Controller	Enhanced
Integration of the PS/2 Controller into the bus system using Wish Bone Master-to-Slave connection architecture	Enhanced
Interrupt Service Routine (ISR) for PS/2 Controller	Enhanced

Table 8.1: Enhancement Outcome

8.2 Discussion and Future Work

The Mouse Controller that was developed by senior wasn't tested successfully for its functionality; hence it was not integrated into the RISC32 architecture to test the serial communication between PS/2 Controller and the Mouse Controller. The reason for this failure is due to the design faults on the Mouse Controller modelled by the senior, as the finite state machine (FSM) for the Mouse Controller does not follow the proper design rules. The Mouse Controller design should be revised and improved so that a proper test of serial communication can be conducted in future.

Lastly, the PS/2 Controller for the keyboard has not been developed, hence it the future design, the RISC32 CPU can be enhanced to connect to both the PS/2 Controller for mouse and for the keyboard via the I/O bus using either a bus arbitration system or a Wish Bone interconnects.

For future, the Keyboard Controller and the Mouse Controller can be remodelled to communicate with their corresponding PS/2 Controller to create a complete PS/2 system environment. In addition, exception handling can be implemented to arithmetic overflow exception, breakpoint exception and others.

BIBLIOGRAPHY

WIKIPEDIA (2012) IBM Personal System/2.

OSDEV.ORG (2012) PS/2.

CHU, P. P. (2008) *FPGA Protoyping by Verilog Examples*, John Wiley & Sons, Inc.

CHAPWESKE, A. (2003) The PS/2 Mouse/Keyboard Protocol.

JOHNSON, H. (1998) Power - On - Reset.

DANDAMUDI, S. P. (2002) *Fundamentals of Computer Organization and Design,* New York, United States of America, Springer.

ALTERA (2012) Welcome to the Quartus II Software.

MODELSIM (2012) ModelSim PE Student Edition - HDL Simulation.

SYNOPSYS (2012) VCS.

ASHENDEN, P. J. (2008) *Digital Design: An Embedded Systems Approach Using Verilog,* MU, United States of America, Morgan Kaufmann.

CHONNAD, S. & BALACHANDER, N. (2004) *Verilog Frequently Asked Question: Languages, Applications and Extensions,* Boston, Springer Science + Business Media, Inc.

SHIVA, S. G. (2008) *Computer Organization, Design, and Architecture,* New York, United States of America, CRC Press.

WILSON, P. (2007) *Design Recipes for FPGA Examples*, ELSEVIER.

SHIVA, S. G. (2008) *Computer Organization, Design, and Architecture,* New York, United States of America, CRC Press.

LIN, M.-B. (2011) *Digital System Designs and Practices Using Verilog HDL and FPGAs,* Singapore, John Wiley & Sons (Asia) Pte Ltd.

PETERSON, W. D. (2010) WISHBONE System-On-Chip (SoC) Interconnection Architecture Portable IP Cores. IN HERVEILLE, R. (Ed.), OpenCores.

Appendix A Bibliography

ALTIUM (2008) WB_INTERCON Configurable Wishbone Interconnect.

B-1RISC32 Processor integrated with PS/2 Controller Unit and Address Decoder

```
//#######################################################################
#####
//Filename: c_risc32_join_ps2.v
//Date created: 14/2/2014
//Author : ?
//Modified by : Ng Kwong Cheong
//Description: This module is the full chip module.
//#######################################################################
#####

//Load macro file
`include "macro.v"

//Load design for cp0
`include "cp0/Microarch/b_cp0_dc.v"
`include "cp0/Microarch/b_cp0_regfile.v"
`include "cp0/u_cp0.v"

//Load design for control path
`include "ctrlpath/Microarch/b_alb_ctrl.v"
`include "ctrlpath/Microarch/b_fwrd.v"
`include "ctrlpath/Microarch/b_iag_ctrl.v"
`include "ctrlpath/Microarch/b_itl_ctrl.v"
`include "ctrlpath/Microarch/b_main_ctrl.v"
`include "ctrlpath/u_ctrl_path_full.v"

//Load design from data path
`include "datapath/Microarch/mult/add_lvl1_lastrow.v"
`include "datapath/Microarch/mult/adder_lvl1.v"
`include "datapath/Microarch/mult/adder_lvl1_firstrow.v"
`include "datapath/Microarch/mult/adder_lvl2.v"
`include "datapath/Microarch/mult/adder_lvl2_lastrow.v"
`include "datapath/Microarch/mult/adder_lvl3.v"
`include "datapath/Microarch/mult/adder_lvl4.v"
`include "datapath/Microarch/mult/adder_lvl5.v"
`include "datapath/Microarch/mult/sub_lvl1_lastrow.v"
`include "datapath/Microarch/b_alb_32.v"
`include "datapath/Microarch/b_branch_pred.v"
`include "datapath/Microarch/b_mult_32.v"
`include "datapath/Microarch/b_reg_file.v"
`include "datapath/u_data_path_full.v"

//Load design from memory
`include "memory/Microarch/b_cache.v"
`include "memory/u_memory.v"

//Load design from PS/2 mouse
`include "fyp_ps2_mouse/b_receive.v"
`include "fyp_ps2_mouse/b_transmit.v"
`include "fyp_ps2_mouse/b_wb_if.v"
`include "fyp_ps2_mouse/b_synch_single.v"
```

```
`include "fyp_ps2_mouse/b_synch_full.v"
`include "fyp_ps2_mouse/b_addr_decoder.v"
`include "fyp_ps2_mouse/u_ps2.v"

module c_r32_join_ps2
  #(parameter
    CPU_MS_DATA_ADDR = `WIDTH_WORD'h0000_0104)
  (output                          co_r32_ps2_clk_en, //PS/2 clock output
enable
   output                          co_r32_ps2_clk,    //PS/2 clock output
   output                          co_r32_ps2_dat_en, //PS/2 data output
enable
   output                          co_r32_ps2_dat,    //PS/2 data output
   input                           ci_r32_ps2_clk,
   input                           ci_r32_ps2_rst,
   input                           ci_r32_ps2_eclk,   //PS/2 device clock
input
   input                           ci_r32_ps2_edat    //PS/2 device data
input
      );

  wire  [`WIDTH_OPCODE - 1:0]      c_r32_opcode;
  wire  [`WIDTH_FUNCT  - 1:0]      c_r32_funct;
  wire                             c_r32_prediction;
  wire                             c_r32_bran_vid;

  wire  [`WIDTH_WORD - 1:0]        c_r32_pc;
  wire  [`WIDTH_WORD - 1:0]        c_r32_dmem_addr;
  wire  [`WIDTH_WORD - 1:0]        c_r32_store_data;

  wire                             c_r32_alb_src;
  wire                             c_r32_rdst_src;
  wire  [`WIDTH_BRAN_CTRL - 1:0]   c_r32_bran_ctrl;
  wire                             c_r32_sign_mult;
  wire                             c_r32_rf_write;
  wire                             c_r32_mem_write;
  wire                             c_r32_mem_read;
  wire                             c_r32_sign_ext;
  wire                             c_r32_hi_to_rf;
  wire                             c_r32_hi_we;
  wire                             c_r32_lo_we;
  wire                             c_r32_alb_to_rf;
  wire                             c_r32_mult_en;
  wire                             c_r32_mem_to_rf;

  wire  [`WIDTH_ALB_CTRL - 1:0]    c_r32_alb_ctrl;
  wire      [4:0]
      c_r32_dp_inst25_21;

  wire                             c_r32_if_flush;
  wire                             c_r32_pc_src;
  wire                             c_r32_prediction_src;
  wire  [1:0]                      c_r32_correction_src;
```

```
wire                            c_r32_store_addr;

wire  [`WIDTH_WORD - 1:0]       c_r32_instruction;
wire  [`WIDTH_WORD - 1:0]       c_r32_loaded_data;
wire                            c_r32_mem_re;
wire                            c_r32_mem_we;

//forwarding
wire  [`WIDTH_FWRD_ALB - 1:0]   c_r32_fwrd_alb_id_rsrc;
wire  [`WIDTH_FWRD_ALB - 1:0]   c_r32_fwrd_alb_id_rtgt;
wire                            c_r32_fwrd_hilo;
wire                            c_r32_fwrd_mem_ex_rtgt;
wire                            c_r32_fwrd_mem_id_rsrc;
wire                            c_r32_fwrd_mem_id_rtgt;
wire                            c_r32_ex_rf_write;
wire                            c_r32_mem_rf_write;
wire                            c_r32_wb_rf_write;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_ex_rdst;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_mem_rdst;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_wb_rdst;

//forwarding and interlock control
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_id_rsrc;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_id_rtgt;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_ex_rtgt;

//interlock control
wire                            c_r32_pc_write;
wire                            c_r32_ifid_write;
wire                            c_r32_idex_write;
wire                            c_r32_exmem_write;
wire                            c_r32_memwb_write;
wire                            c_r32_id_flush;
wire                            c_r32_ex_flush;
wire                            c_r32_mem_flush;
wire                            c_r32_ex_mem_read;
wire                            c_r32_mult_busy;

//branch prediction
wire                            c_r32_upd_pred;

//cp0 added wire
//control unit
wire                            c_r32_ctrl_is_mfc0;
wire                            c_r32_ctrl_is_mtc0;
wire                            c_r32_ctrl_is_eret;

//datapath
wire                            c_r32_dp_is_mtc0;
wire  [`WIDTH_WORD - 1:0]       c_r32_dp_cp0_reg_data;
wire  [`WIDTH_REG_ADDR - 1:0]   c_r32_dp_cp0_reg_addr;
wire                            c_r32_dp_is_overflow;
```

```
  //cp0
  wire  [`WIDTH_WORD - 1:0]       c_r32_cp0_reg_data;
  wire  [`WIDTH_WORD - 1:0]       c_r32_cp0_excep_addr;
  wire                            c_r32_cp0_is_intr;
  wire                            c_r32_cp0_is_overflow;
  wire  [5:0]                     c_r32_intr_vector;

  //address_decoder
  wire                            c_r32_wb_addr;
  wire  [`PS2_WORD - 1:0]         c_r32_wb_dat;
  wire                            c_r32_wb_stb;
  wire                            c_r32_wb_cyc;
  wire                            c_r32_wb_w_rn;
  wire                            c_r32_wb_ack;
  wire  [`PS2_WORD - 1:0]         c_r32_dat_ps2;

  //ps2_ctrl
  wire                            c_r32_ps2_interrupt;
  wire                            c_r32_ps2_io_ps2c;
  wire                            c_r32_ps2_io_ps2d;

  //data select for ps2
  wire                            c_r32_operation_io;
  wire     [`WIDTH_WORD - 1:0]                    c_r32_dcrddata;
  wire  [`WIDTH_WORD - 1:0]       c_r32_data_io;

  pullup(c_r32_ps2_io_ps2c);
  pullup(c_r32_ps2_io_ps2d);
  assign c_r32_ps2_io_ps2c = co_r32_ps2_clk_en ? co_r32_ps2_clk : 1'bz;
  assign c_r32_ps2_io_ps2d = co_r32_ps2_dat_en ? co_r32_ps2_dat : 1'bz;
      assign c_r32_operation_io = c_r32_dmem_addr>=CPU_MS_DATA_ADDR &&
c_r32_dmem_addr<=CPU_MS_DATA_ADDR;
      assign c_r32_dcrddata = c_r32_operation_io ? c_r32_data_io :
c_r32_loaded_data;
  assign c_r32_intr_vector = {c_r32_ps2_interrupt, 5'b0};

  u_ctrl_path_full u_control
  (//main control signal
   .uo_cp_alb_src(c_r32_alb_src),
   .uo_cp_rdst_src(c_r32_rdst_src),
   .uo_cp_bran_ctrl(c_r32_bran_ctrl),
   .uo_cp_mult_en(c_r32_mult_en),
   .uo_cp_sign_mult(c_r32_sign_mult),
   .uo_cp_rf_write(c_r32_rf_write),
   .uo_cp_mem_write(c_r32_mem_write),
   .uo_cp_mem_read(c_r32_mem_read),
   .uo_cp_sign_ext(c_r32_sign_ext),
   .uo_cp_hi_we(c_r32_hi_we),
   .uo_cp_lo_we(c_r32_lo_we),
   .uo_cp_alb_to_rf(c_r32_alb_to_rf),
   .uo_cp_hi_to_rf(c_r32_hi_to_rf),
   .uo_cp_mem_to_rf(c_r32_mem_to_rf),
   .uo_cp_is_mtc0(c_r32_ctrl_is_mtc0),
```

```
    .uo_cp_is_mfc0(c_r32_ctrl_is_mfc0),
    .uo_cp_is_eret(c_r32_ctrl_is_eret),
    //alb
    .uo_cp_alb_ctrl(c_r32_alb_ctrl),
    //branch control signal
    .uo_cp_if_flush(c_r32_if_flush),
    .uo_cp_pc_src(c_r32_pc_src),
    .uo_cp_prediction_src(c_r32_prediction_src),
    .uo_cp_correction_src(c_r32_correction_src),
    .uo_cp_store_addr(c_r32_store_addr),
    .uo_cp_upd_pred(c_r32_upd_pred),
    //forwarding
    .uo_cp_fwrd_alb_id_rsrc(c_r32_fwrd_alb_id_rsrc),
    .uo_cp_fwrd_alb_id_rtgt(c_r32_fwrd_alb_id_rtgt),
    .uo_cp_fwrd_hilo(c_r32_fwrd_hilo),
    .uo_cp_fwrd_mem_ex_rtgt(c_r32_fwrd_mem_ex_rtgt), //mem-to-mem copy
    .uo_cp_fwrd_mem_id_rsrc(c_r32_fwrd_mem_id_rsrc),
    .uo_cp_fwrd_mem_id_rtgt(c_r32_fwrd_mem_id_rtgt),
    //interlock control
    .uo_cp_pc_write(c_r32_pc_write),
    .uo_cp_ifid_write(c_r32_ifid_write),
    .uo_cp_idex_write(c_r32_idex_write),
    .uo_cp_exmem_write(c_r32_exmem_write),
    .uo_cp_memwb_write(c_r32_memwb_write),
    .uo_cp_id_flush(c_r32_id_flush),
    .uo_cp_ex_flush(c_r32_ex_flush),
    .uo_cp_mem_flush(c_r32_mem_flush),
    //main control
    .ui_cp_opcode(c_r32_opcode),
    .ui_cp_funct(c_r32_funct),
    .ui_cp_inst25_21(c_r32_dp_inst25_21),
    //branch
    .ui_cp_prediction(c_r32_prediction),
    .ui_cp_bran_vld(c_r32_bran_vid),
    .ui_cp_pred_crt(c_r32_pred_crt),
    //interlock control
    .ui_cp_ex_mem_read(c_r32_ex_mem_read),
    .ui_cp_mult_busy(c_r32_mult_busy),
    .ui_cp_is_overflow(c_r32_cp0_is_overflow),
    //interlock control and forwarding
    .ui_cp_id_rsrc(c_r32_id_rsrc),
    .ui_cp_id_rtgt(c_r32_id_rtgt),
    .ui_cp_ex_rtgt(c_r32_ex_rtgt),
    //forwarding
    .ui_cp_ex_rf_write(c_r32_ex_rf_write),
    .ui_cp_mem_rf_write(c_r32_mem_rf_write),
    .ui_cp_wb_rf_write(c_r32_wb_rf_write),
    .ui_cp_mem_mem_read(c_r32_mem_re),
    .ui_cp_ex_rdst(c_r32_ex_rdst),
    .ui_cp_mem_rdst(c_r32_mem_rdst),
    .ui_cp_wb_rdst(c_r32_wb_rdst));

  u_data_path_full u_dp
```

```
  (//cp0
   .uo_dp_is_mtc0(c_r32_dp_is_mtc0),
   .uo_dp_cp0_reg_addr(c_r32_dp_cp0_reg_addr),
   .uo_dp_cp0_reg_data(c_r32_dp_cp0_reg_data),
   .uo_dp_inst25_21(c_r32_dp_inst25_21),
   //main control
   .uo_dp_opcode(c_r32_opcode),
   .uo_dp_funct(c_r32_funct),
   //branch control
   .uo_dp_prediction(c_r32_prediction),
   .uo_dp_bran_vld(c_r32_bran_vid),
   .uo_dp_pred_crt(c_r32_pred_crt),
   //alb
   .uo_dp_overflow(c_r32_dp_is_overflow),
   //memory unit
   .uo_dp_pc(c_r32_pc),
   .uo_dp_dmem_addr(c_r32_dmem_addr),
   .uo_dp_store_data(c_r32_store_data),
   .uo_dp_mem_we(c_r32_mem_we),
   //mem unit and forwarding
   .uo_dp_mem_re(c_r32_mem_re),
   //interlock control
   .uo_dp_ex_mem_read(c_r32_ex_mem_read),
   .uo_dp_mult_busy(c_r32_mult_busy),
   //interlock control and forwarding
   .uo_dp_id_rsrc(c_r32_id_rsrc),
   .uo_dp_id_rtgt(c_r32_id_rtgt),
   .uo_dp_ex_rtgt(c_r32_ex_rtgt),
   //forwarding
   .uo_dp_ex_rf_write(c_r32_ex_rf_write),
   .uo_dp_mem_rf_write(c_r32_mem_rf_write),
   .uo_dp_wb_rf_write(c_r32_wb_rf_write),
   .uo_dp_ex_rdst(c_r32_ex_rdst),
   .uo_dp_mem_rdst(c_r32_mem_rdst),
   .uo_dp_wb_rdst(c_r32_wb_rdst),
   //main control
   .ui_dp_alb_src(c_r32_alb_src),
   .ui_dp_rdst_src(c_r32_rdst_src),
   .ui_dp_bran_ctrl(c_r32_bran_ctrl),
   .ui_dp_mult_en(c_r32_mult_en),
   .ui_dp_sign_mult(c_r32_sign_mult),
   .ui_dp_rf_write(c_r32_rf_write),
   .ui_dp_mem_write(c_r32_mem_write),
   .ui_dp_mem_read(c_r32_mem_read),
   .ui_dp_sign_ext(c_r32_sign_ext),
   .ui_dp_hi_we(c_r32_hi_we),
   .ui_dp_lo_we(c_r32_lo_we),
   .ui_dp_alb_to_rf(c_r32_alb_to_rf),
   .ui_dp_hi_to_rf(c_r32_hi_to_rf),
   .ui_dp_mem_to_rf(c_r32_mem_to_rf),
   //alb
   .ui_dp_alb_ctrl(c_r32_alb_ctrl),
   //branch control signal
```

```
    .ui_dp_if_flush(c_r32_if_flush),
    .ui_dp_pc_src(c_r32_pc_src),
    .ui_dp_prediction_src(c_r32_prediction_src),
    .ui_dp_correction_src(c_r32_correction_src),
    .ui_dp_store_addr(c_r32_store_addr),
    .ui_dp_upd_pred(c_r32_upd_pred),
    //forwarding
    .ui_dp_fwrd_alb_id_rsrc(c_r32_fwrd_alb_id_rsrc),
    .ui_dp_fwrd_alb_id_rtgt(c_r32_fwrd_alb_id_rtgt),
    .ui_dp_fwrd_hilo(c_r32_fwrd_hilo),
    .ui_dp_fwrd_mem_ex_rtgt(c_r32_fwrd_mem_ex_rtgt),
    .ui_dp_fwrd_mem_id_rsrc(c_r32_fwrd_mem_id_rsrc),
    .ui_dp_fwrd_mem_id_rtgt(c_r32_fwrd_mem_id_rtgt),
    //interlock control
    .ui_dp_pc_write(c_r32_pc_write),
    .ui_dp_ifid_write(c_r32_ifid_write),
    .ui_dp_idex_write(c_r32_idex_write),
    .ui_dp_exmem_write(c_r32_exmem_write),
    .ui_dp_memwb_write(c_r32_memwb_write),
    .ui_dp_id_flush(c_r32_id_flush),
    .ui_dp_ex_flush(c_r32_ex_flush),
    .ui_dp_mem_flush(c_r32_mem_flush),
    //memory unit
    .ui_dp_instruction(c_r32_instruction),
    .ui_dp_loaded_data(c_r32_dcrddata),
    //cp0
    .ui_dp_is_mtc0(c_r32_ctrl_is_mtc0),
    .ui_dp_is_mfc0(c_r32_ctrl_is_mfc0),
    .ui_dp_cp0_reg_data(c_r32_cp0_reg_data),
    .ui_dp_excep_addr(c_r32_cp0_excep_addr),
    .ui_dp_is_intr(c_r32_cp0_is_intr),
    .ui_dp_is_overflow(c_r32_cp0_is_overflow),
    .ui_dp_is_eret(c_r32_ctrl_is_eret),
    //system signal
    .ui_dp_clk(ci_r32_ps2_clk),
    .ui_dp_reset(ci_r32_ps2_rst));

  u_memory u_memory
   (.uo_mem_ic_data_rd(c_r32_instruction),
    .uo_mem_dc_data_rd(c_r32_loaded_data),
    .ui_mem_ic_addr(c_r32_pc),
    .ui_mem_dc_addr(c_r32_dmem_addr),
    .ui_mem_dc_data_wr(c_r32_store_data),
    .ui_mem_dc_we(c_r32_mem_we), //to dc
    .ui_mem_dc_re(c_r32_mem_re), //to dc
    .ui_mem_clk(ci_r32_ps2_clk));

  u_cp0 u_cp0
  (.ui_cp0_mtc0(c_r32_dp_is_mtc0),
   .ui_cp0_is_eret(c_r32_ctrl_is_eret),
   .ui_cp0_current_pc_2_EPC(c_r32_pc),
   .ui_cp0_intr_vector(c_r32_intr_vector),
   .ui_cp0_overflow_signal(c_r32_dp_is_overflow),
```

```
    .ui_cp0_reg_data(c_r32_dp_cp0_reg_data),
    .ui_cp0_reg_address(c_r32_dp_cp0_reg_addr),
    .ui_cp0_sys_clock(ci_r32_ps2_clk),
    .uo_cp0_cp0_reg_data(c_r32_cp0_reg_data),
    .uo_cp0_excep_handler_address(c_r32_cp0_excep_addr),
    .uo_cp0_is_intr(c_r32_cp0_is_intr),
    .uo_cp0_is_overflow(c_r32_cp0_is_overflow));

    b_addr_decoder b_addr_decoder
    (.bo_wb_stb(c_r32_wb_stb),
     .bo_wb_cyc(c_r32_wb_cyc),
     .bo_wb_addr(c_r32_wb_addr),
     .bo_wb_w_rn(c_r32_wb_w_rn),
     .bo_wb_dat(c_r32_wb_dat),
     .bo_dat_cpu(c_r32_data_io),
     .bi_dat_ps2(c_r32_dat_ps2),
     .bi_dat_cpu(c_r32_store_data),
     .bi_addr_cpu(c_r32_dmem_addr),
     .bi_wb_ack(c_r32_wb_ack),
     .bi_rd_cpu(c_r32_mem_re),
     .bi_wr_cpu(c_r32_mem_we));

    u_ps2 u_ps2
    (.uo_ps2_clk_en(co_r32_ps2_clk_en),
     .uo_ps2_clk(co_r32_ps2_clk),
     .uo_ps2_dat_en(co_r32_ps2_dat_en),
     .uo_ps2_dat(co_r32_ps2_dat),
     .uo_ps2_wb_dat(c_r32_dat_ps2),
     .uo_ps2_wb_ack(c_r32_wb_ack),
     .uo_ps2_intr(c_r32_ps2_interrupt),
     .ui_ps2_clk(ci_r32_ps2_clk),
     .ui_ps2_rst(ci_r32_ps2_rst),
     .ui_ps2_wb_stb(c_r32_wb_stb),
     .ui_ps2_wb_cyc(c_r32_wb_cyc),
     .ui_ps2_wb_addr(c_r32_wb_addr),
     .ui_ps2_wb_w_rn(c_r32_wb_w_rn),
     .ui_ps2_wb_dat(c_r32_wb_dat),
     .ui_ps2_eclk(ci_r32_ps2_eclk),
     .ui_ps2_edat(ci_r32_ps2_edat));

endmodule
```

B-2 Test Bench

```
//#########################################################################
//Filename: tb_c_risc32_join_ps2.v
//Date created: 28/2/2014
//Author : Ng Kwong Cheong
//Description: Testbench for the full chip module.
//#########################################################################
```

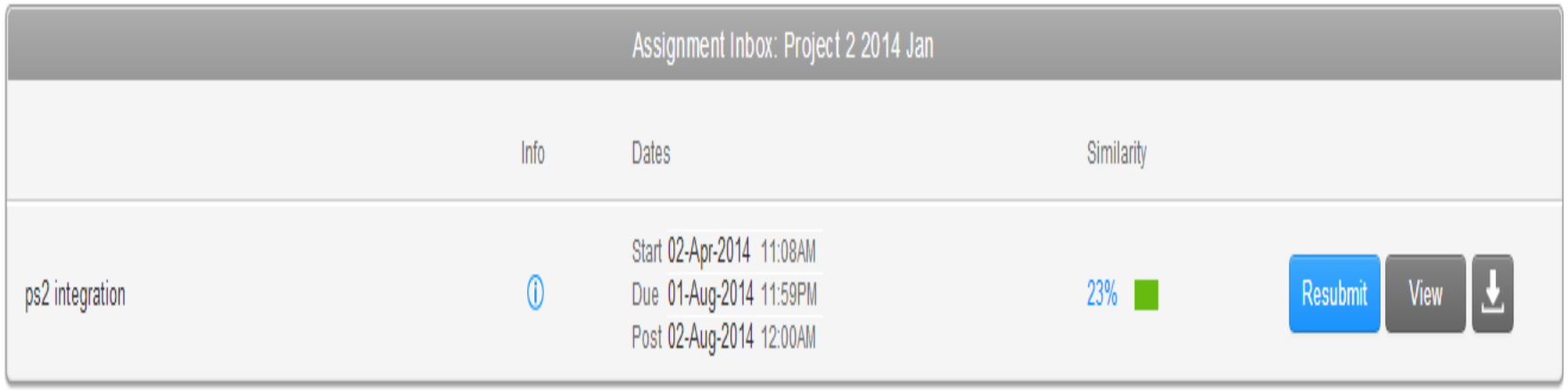
```
module tb_c_r32_join_ps2
      ();

      //Wire declarations
      wire                    tbo_r32_ps2_clk_en;
      wire                    tbo_r32_ps2_clk;
      wire                    tbo_r32_ps2_dat_en;
      wire                    tbo_r32_ps2_dat;

      //Register declarations
      reg                     tbi_r32_ps2_clk;
      reg                     tbi_r32_ps2_rst;
      reg                     tbi_r32_ps2_eclk;
      reg                     tbi_r32_ps2_edat;

      //Test data buffer declarations
      reg   [9:0]       test_rx_data;
      reg   [9:0]       test_rx_data_two;
      reg   [9:0]       test_rx_data_three;

      //Flag register declarations
      reg               rx_flag_start, rx_flag_stop;
      reg               rx_flag_start_two, rx_flag_stop_two;
      reg               rx_flag_start_three, rx_flag_stop_three;

      //Counter declarations
      integer           location;
      integer           rx_bit_count = 0;
      integer           rx_bit_count_two = 0;
      integer           rx_bit_count_three = 0;

      c_r32_join_ps2   DUT_c_r32_join_ps2
      (.co_r32_ps2_clk_en(tbo_r32_ps2_clk_en),
       .co_r32_ps2_clk(tbo_r32_ps2_clk),
       .co_r32_ps2_dat_en(tbo_r32_ps2_dat_en),
       .co_r32_ps2_dat(tbo_r32_ps2_dat),
       .ci_r32_ps2_clk(tbi_r32_ps2_clk),
       .ci_r32_ps2_rst(tbi_r32_ps2_rst),
       .ci_r32_ps2_eclk(tbi_r32_ps2_eclk),
       .ci_r32_ps2_edat(tbi_r32_ps2_edat));

      initial tbi_r32_ps2_clk = 1'b0;
      always #5000 tbi_r32_ps2_clk = ~tbi_r32_ps2_clk;

      initial begin
            //Signals initialization
            tbi_r32_ps2_rst = 1'b0;
            tbi_r32_ps2_eclk = 1'b1;     //standard signal
            tbi_r32_ps2_edat = 1'b1;     //standard signal
            rx_flag_start = 1'b0;
            rx_flag_stop = 1'b0;
            rx_flag_start_two = 1'b0;
            rx_flag_stop_two = 1'b0;
```

```
            rx_flag_start_three = 1'b0;
            rx_flag_stop_three = 1'b0;

            //Read file and load to memory
            $readmemh("ps2_test_hex.txt",
DUT_c_r32_join_ps2.u_memory.b_ic.b_cm_r_memory);
            $readmemh("ps2_test_data.txt",
DUT_c_r32_join_ps2.u_memory.b_dc.b_cm_r_memory);
            $readmemh("ps2_test_rf.txt",
DUT_c_r32_join_ps2.u_dp.b_rf.b_rf_r_register);

           //Display loaded data
            for(location = 0; location < 4096; location = location +
1)begin
                  $display("Memory [%0d]\t = \t%h", location,
DUT_c_r32_join_ps2.u_memory.b_ic.b_cm_r_memory[location]);
            end

            //System reset
            @(posedge tbi_r32_ps2_clk)
            tbi_r32_ps2_rst = 1'b1;

            repeat(3) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_rst = 1'b0;

            //Receive data from PS/2 mouse
            @(posedge tbi_r32_ps2_clk)
            test_rx_data = 10'b11_1111_1010; //8'hFA, acknowledge byte,
parity bit = 1'b1

            repeat(4) @(posedge tbi_r32_ps2_clk);
            rx_flag_start = 1'b1;

            repeat(10) begin
                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b0;

                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b1;
            end

            rx_flag_stop = 1'b1;//Generate stop bit after sending data +
parity bits

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b0;

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b1;


            //Receive second data from PS/2 mouse
            @(posedge tbi_r32_ps2_clk)
```

```
            test_rx_data_two = 10'b11_1010_1010; //8'hAA, Power-on basic
assurance test "passed" result, parity bit = 1'b1

            repeat(4) @(posedge tbi_r32_ps2_clk);
            rx_flag_start_two = 1'b1;

            repeat(10) begin
                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b0;

                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b1;
            end

            rx_flag_stop_two = 1'b1; //Generate stop bit after sending
data + parity bits

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b0;

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b1;

            //Receive last data from PS/2 mouse
            @(posedge tbi_r32_ps2_clk)
    test_rx_data_three = 10'b11_0000_0000; //8'h00, PS/2 mouse ID, parity
bit = 1'b1

            repeat(4) @(posedge tbi_r32_ps2_clk);
            rx_flag_start_three = 1'b1;

            repeat(10) begin
                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b0;

                  repeat(4) @(posedge tbi_r32_ps2_clk);
                  tbi_r32_ps2_eclk = 1'b1;
            end

            rx_flag_stop_three = 1'b1; //Generate stop bit after sending
data + parity bits

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b0;

            repeat(4) @(posedge tbi_r32_ps2_clk);
            tbi_r32_ps2_eclk = 1'b1;

            //Stop operation
            repeat(80) @(posedge tbi_r32_ps2_clk);
            $stop;
      end
```

```
      /*********Receiver block for receiving first data**********/
      initial begin
            tbi_r32_ps2_edat = 1'b1;

            wait(rx_flag_start)
            tbi_r32_ps2_edat = 1'b0;

            repeat(10) @(posedge tbi_r32_ps2_eclk)begin
                  tbi_r32_ps2_edat <= test_rx_data[rx_bit_count];
                  rx_bit_count <= rx_bit_count + 1;
            end

            wait(rx_flag_stop)
            tbi_r32_ps2_edat = 1'b1;
      end
      /*********************************************************/

      /*********Receiver block for receiving second data*********/
      initial begin
            tbi_r32_ps2_edat = 1'b1;

            wait(rx_flag_start_two)
            tbi_r32_ps2_edat = 1'b0;

            repeat(10) @(posedge tbi_r32_ps2_eclk)begin
                  tbi_r32_ps2_edat <= test_rx_data_two[rx_bit_count_two];
                  rx_bit_count_two <= rx_bit_count_two + 1;
            end

            wait(rx_flag_stop_two)
            tbi_r32_ps2_edat = 1'b1;
      end
      /*********************************************************/

      /*********Receiver block for receiving last data***********/
      initial begin
            tbi_r32_ps2_edat = 1'b1;

            wait(rx_flag_start_three)
            tbi_r32_ps2_edat = 1'b0;

            repeat(10) @(posedge tbi_r32_ps2_eclk)begin
                  tbi_r32_ps2_edat <=
test_rx_data_three[rx_bit_count_three];
                  rx_bit_count_three <= rx_bit_count_three + 1;
            end

            wait(rx_flag_stop_three)
            tbi_r32_ps2_edat = 1'b1;
      end
 /*********************************************************/

endmodule
```

Appendix C Turnitin result



Assignment Inbox: Project 2 2014 Jan

	Info	Dates	Similarity	
ps2 integration	ⓘ	Start 02-Apr-2014 11:08AM Due 01-Aug-2014 11:59PM Post 02-Aug-2014 12:00AM	23%	Resubmit View