Design of Pacman with Debug Logic

DINESH KOTHAMASU
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Design of Pacman with Debug Logic

Master Thesis

By

Dinesh Kothamasu

Master of Science in System on Chip

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

2017

Abstract

This Thesis work was performed at Ineda System Pvt Ltd, Hyderabad, India.

Pacman is an interrupt controller and priority resolver with 16x8 input interrupt lines. The main objective of this Master Thesis is to upgrade 16x8 interrupt controller and priority resolver to 128x8 input interrupt lines and adding a debug feature for this customised processor which has its own instruction set. A debugger is used to debug the target programme.

In this thesis, the upgradation of Pacman and design of debugging features such as halt, break point, single step are implemented at the Register Transfer Level (RTL) in the processor. The processor is integrated with Memory, JtagtoAHB, System Register modules and the Advanced High-performance Bus (AHB) Arbiter. The design is verified using System Verilog test bench to test the functional correctness of the design and validated the design in FPGA environment.

Acknowledgements

Table of Contents

Introduction

1.1 Overview

Interrupts are given to the processor through hardware which is known as an Interrupt Controller. The most popular interrupt controller is 8259. This 8259 interrupt controller has 8 input lines which can take inputs from 8 devices and feed them to the processor. The processor stops its execution and takes care of these interrupts due to the processor losing its performance.

To overcome this problem, Advanced RISC Machine (ARM) designed Cortex M processors which can handle and resolve the interrupts without depending upon the primary processor. Similarly, we designed a Pacman interrupt controller which is a low power efficient and can act as a secondary processor. The Pacman prioritises the interrupts and resolves them without disturbing the primary processor. Due to this, we can utilise the processor to its maximum. I also implemented debug logic in Pacman where we can debug Pacman using Catalyst. Pacman supports halt, break point, and single step.

The break point can break the code and stop the execution at the required address. Single step can step and debug each line of the code.

Before break point, the programmer has only two states. The initial state of the application before it ran, and the final state of the application.

As software moves ahead, there is much improvement in the debugging features also. The ability to see the original code to be able to set a

breakpoint is added. Better ways to dump the memory and look at the changes in the memory are added. The next big thing in debuggers is Turbo Pascal with Integrated Development Environment, and this was introduced by Borland with the ability to debug, edit, and compile within the same system.

Modern Debuggers and IDE's widely used are VC++ (Visual), Eclipse (Visual), GDB (Command-Line), etc

Due to these type of IDE and Debuggers, we can debug complex programmes. However, there is still much research needed in debugging. Since more than 50% of the time is spent by programmers in debugging[1].

1.2 Scope of the Thesis Project

This Master thesis is at Ineda Systems Pvt. Ltd., aiming for the upgradation of 16x8 Pacman Interrupt Controller to 128x8 and design of a debugger for the Pacman. Due to the implementation of the debugging feature into Pacman, the customer will be able to debug the Pacman which make things easier for the customer. There are several different types of debuggers available in the market out of which Bus Blaster is an open source debugger and Catalyst which is developed by Ineda Systems that helps only in reading and writing to the Processor. Here we are using Catalyst Hardware for debugging the Pacman using Genie Software. The other debuggers available in the market are Segger, Lauterbach, and the like, which are not open source and work for ARM, MIPS, etc.

This Master thesis project mainly deals with the upgradation of Pacman and implementing a debugging feature that can be tested on FPGA through Catalyst.

1.2.1 Target of the Thesis Project

The target of the thesis project is to design a debugger for the Pacman. The design requirements are listed as follows:

➢ Understand the working of the Pacman at the RTL Level in Verilog and upgrading it from 16x8 to 128x8 input interrupt lines, and adding a debug logic.

➢ Understand the working of Jtag and Advanced High Performance Bus (AHB) and integrated Pacman, JtagtoAHB, and SRAM with the help of AHB Bus.

➢ Verifying the whole system using UVM Methodologies.

➢ Generate a bit file for the whole system on VC707 FPGA Board using Vivado tool.

➢ Debugging it through Genie Software using Catalyst.

The whole thesis work consists of Designing, Verification, Bit File Generation and Software.

1.2.2 Organisation of the Thesis Project

This report is divided into chapters as explained below:

➢ Chapter 2 gives a summary of the whole flow of the project.

➢ Chapter 3 gives sample details about the Pacman architecture.

➢ Chapter 4 gives sample details about the tools used and the design flow of the project.

➢ Chapter 5 gives details about the simulation and synthesis results of the project.

➢ Chapter 6 concludes the report and discusses the future work.

➢ Appendix A gives details about the functioning of Jtag; Appendix B describes the AHB architecture and its working, and Appendix C describes the instruction set architecture of Pacman.

PACMAN Subsystem

Figure 2.1 explains the flow of the whole subsystem. In the Host PC block. I am using a Genie command-line which helps in debugging Pacman through the Catalyst board. The Catalyst board is connected by USB to the Host PC, and the VC707 board is connected to Catalyst through Jtag.
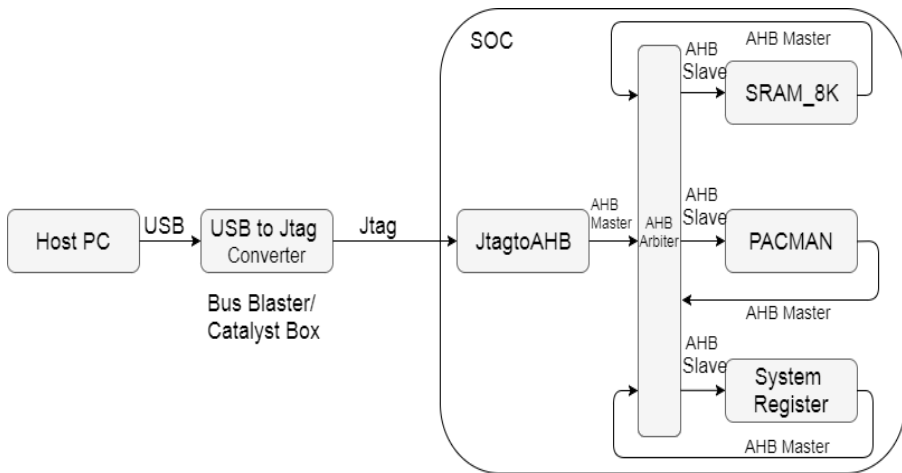
Figure 2.1: Pacman Subsystem

Figure 2.2 shows the Catalyst board designed by Ineda Systems which is used for accessing registers of different processors for reading and writing. The generated bit file is flashed into the VC707 FPGA board. The bit file consists of JtagtoAHB Module, SRAM, Pacman and System Registers. JtagtoAHB acts as AHB Master. This module takes serial inputs which are given to Jtag and is converted to AHB and given to the AHB bus. SRAM, Pacman and System Register acts as AHB Slave. Memory is allocated to each module, SRAM is allocated with 8K memory and can be accessed with

address in between 0x00000000 to 0x00001FFF. Pacman is allocated with 4K memory and can be accessed with address in between 0x00002000 to 0x00002FFF. System Register is allotted with 4K memory and can be accessed with address in between 0x00003000 to 0x00003FFF.
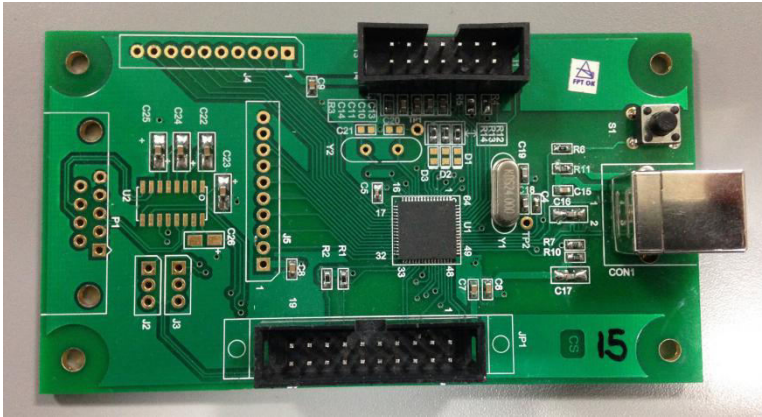
Figure 2.2: Catalyst

SRAM is a memory module taken and integrated into the system. We can load the microcode into the given address of the memory. Microcode is used to resolve the interrupts. Pacman is a 128x8 interrupt controller which can handle up to 128 interrupt lines and supports round-robin and fixed priority. We get interrupts from peripherals since we do not have any peripherals in the system to generate interrupts. We choose a System Register module which helps in generating an interrupt to the Pacman.

PACMAN Architecture

Pacman is an interrupt controller and priority resolver which can handle up to 128 interrupt lines as input, which is active high-level sensitive. Out of these 128 interrupt lines, the user can select up to 8 lines as interrupt vectors which can be serviced with the help of the microcode. Pacman acts as a secondary processor which helps in reducing the burden on the main CPU.

3.1 Feature List

Pacman controller supports the following features:

➢ Up to 128 interrupt lines.

➢ Up to 8 active interrupt lines from the list of input interrupt lines.

➢ The round-robin and fixed priority to service the selected vectors. The selection is programmable.

➢ One AHB Slave to configure the controller register bits.

➢ One AHB Master to fetch the microcode from the system memory.

➢ Four input lines as Wait for the event.

➢ Four output lines as General Purpose Output (GPO).
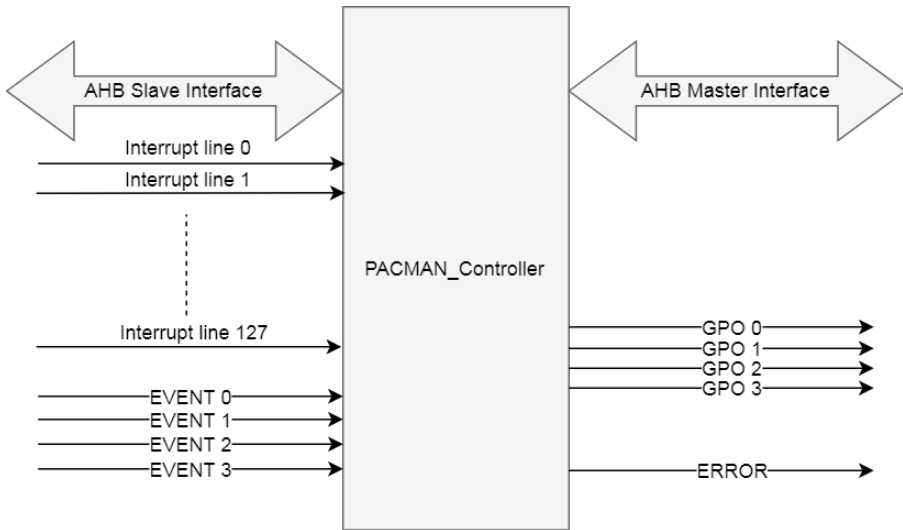
Figure 3.1: Pacman controller

3.2 Interfaces

➤ One AHB Slave interface, which has write and read capabilities, to access and programme control registers and read status/debug registers from CPU.

➤ One AHB Master interface, which has write and read capabilities, to read the microcode that is stored in the system memory and to write data caused by store instructions.

➤ There are 4 General Purpose Output (GPO) lines which can be programmed to '1' or '0'.

➤ There are 4 input lines which can be used as Wait for Event(s).

➤ There are 128 interrupt lines, out of which maximum of 8 interrupt lines can be selected and routed to execute the corresponding microcode in a round-robin manner or fixed priority encoding with priority as Vector 0 having the highest priority and Vector 7 as lowest priority.

8

➢ One output line called "ERROR" which goes high when there is a problem while fetching or executing the microcode. The CPU can read the debug register via slave AHB interface to understand the cause of the error.

3.3 Clock and Reset

The controller operates on a single AHB clock domain and expects synchronous reset signal as input.

3.4 Functional Overview

Pacman design is split into two functional blocks which are:

-Prefetch Buffer

-Interrupt Router and Priority Resolver Block

-Execution Unit

3.4.1 Prefetch Buffer

The prefetch module in the design is interfaced with the AHB bus master. This module is mainly used to handle the read and write transfers onto the AHB master. The main aim of this module is to get the instructions beforehand so that the waiting time for the Pacman is reduced. This module handles the 8-beat, 4-beat and 1-beat transfers that are explained in Appendix B.

Figure 3.2 explains the state machine of the prefetch module. The prefetch buffer function is based on a 7-state FSM. The seven states are S_idle, S_ready, S_8beat, S_4beat, S_1beat, S_busy, and S_wait.   Upon active low reset, the prefetch buffer enters into the S_idle state. Upon start of the programme, the FSM enters into the S_ready state. In   S_ready
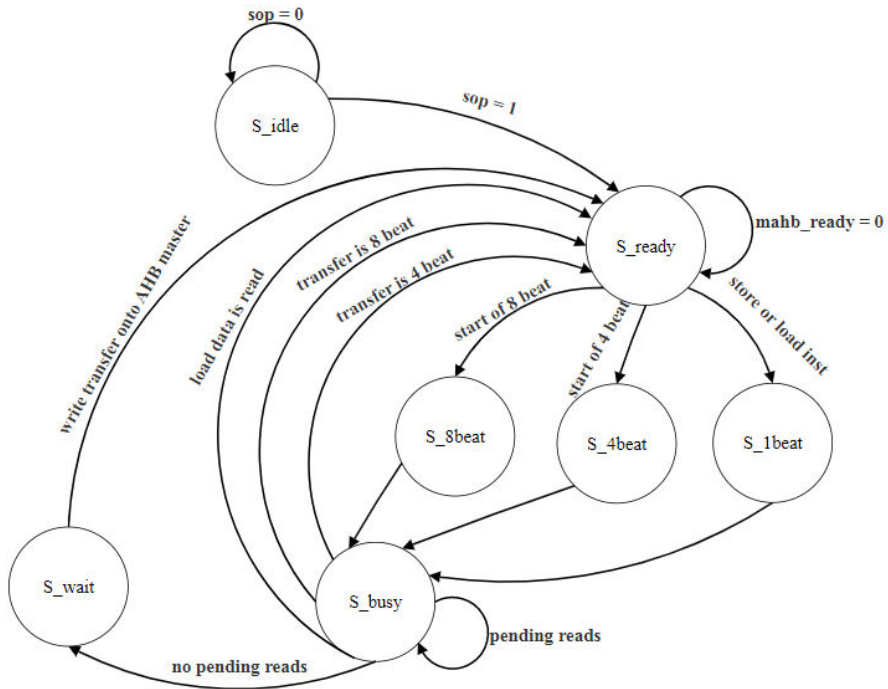
Figure 3.2: Prefetch state of FSM

state if the AHB bus is not ready for the transfer it will remain in the same state. If AHB bus is ready for the transfer depending on the condition enabled it will enter into that particular state. If the 8-beat transfer is enabled, it will enter into the S_8beat state. If the 4-beat transfer is enabled, it will enter into the S_4beat state. If either store or load instruction is enabled, it will enter into the S_1beat state. After entering the particular state, it will set that particular transfer state to '1' and will move onto S_busy state. In S_busy state read data transfer takes place and if there are no pending reads and load data is read, it will again move back to the S_ready state. In S_busy state if there are no pending reads available it will jump to the S_wait state and takes care of write transfer onto AHB master.

3.4.2 Interrupt Router and Priority Resolver

Once the controller programming is done, whenever an interrupt line is received at the input of Pacman, and if it is enabled, the interrupt router directs the interrupt line to the appropriate interrupt vector that is selected by the routing bits of the corresponding interrupt line input.
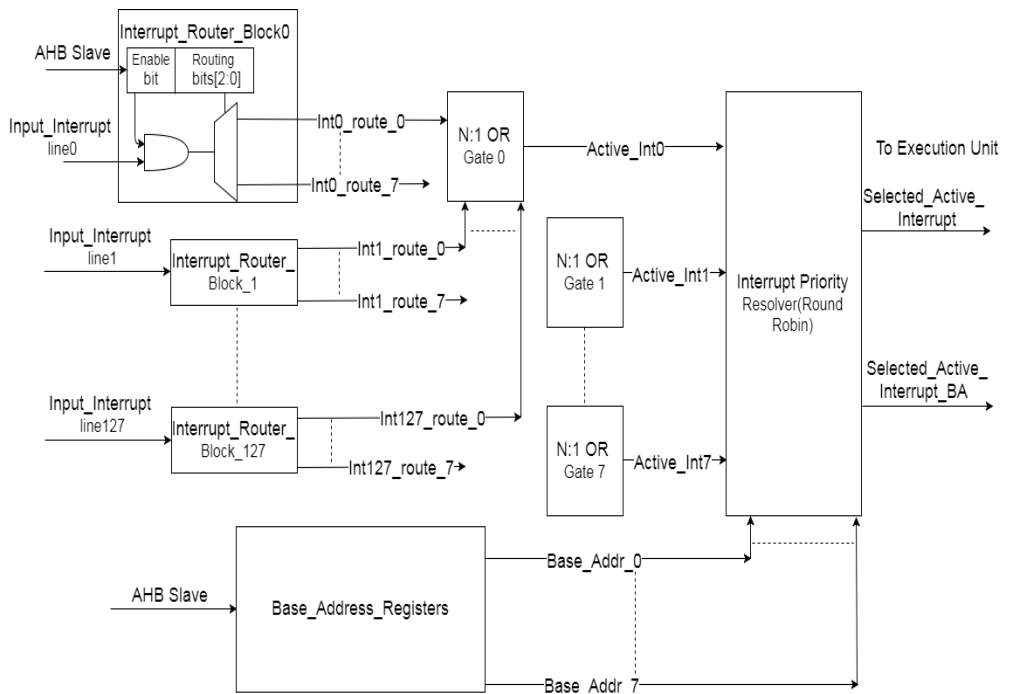

Figure 3.3: Interrupt router and priority resolver

By default, the controller does a round-robin style interrupt handling of the vectors. The controller can be programmed to do a fixed priority vector handling with Vector 0 having the highest priority and Vector 7 having the lowest priority.

Once the vector to be serviced is selected, this block enables the execution unit to process the microcode by providing the Base Address from which the microcode is fetched and executed.

3.4.3 Execution Unit

On receiving the start signal from the "interrupt router and priority resolver" block, the execution unit starts fetching the microcode from the Base Address provided. The execution of microcode continues until the END instruction is executed.

Upon executing the END instruction, the execution unit signals the "interrupt router and priority resolver" block that is ready to process the next vector in the queue.

The process of resolving the priority of vectors and executing the corresponding microcode will be continuing as long as the controller is enabled or till there is an error occurs in fetching and executing the microcode.

In case of an error, the controller provides some debug information which can access via AHB slave interface.

Once the FSM goes into the Error state because of instruction execution error or instruction prefetch error, the FSM sits in error states which can be recovered only by giving Soft Reset bit set to Pacman controller logic.

Figure 3.4 explains the state machine of the execution unit. The execution unit is based on a 5-state FSM. The five states are S_idle, S_busy, S_fetch, S_wait, and S_error. Upon active low reset, the execution unit enters into the S_idle state. Upon the start of the programme, the FSM will move to the S_busy state. In this state it checks if the prefetch buffer is empty it will continue in the S_busy state, if the prefetch buffer is not empty then the

FSM will move to the S_fetch state. In the S_fetch state, it checks different conditions, out of which if the jump instruction is executing it will again move back to the S_busy state. If there is any memory transfer starting in the S_fetch state, then the FSM will move to the S_wait state and waits in this state until the memory transfer is done. In the S_fetch state, it also checks the prefetch buffer if prefetch buffer is empty the FSM will move to the S_busy state while if prefetch buffer is not empty, it will stay in the same state. When the FSM enters S_wait state due to memory transfer it
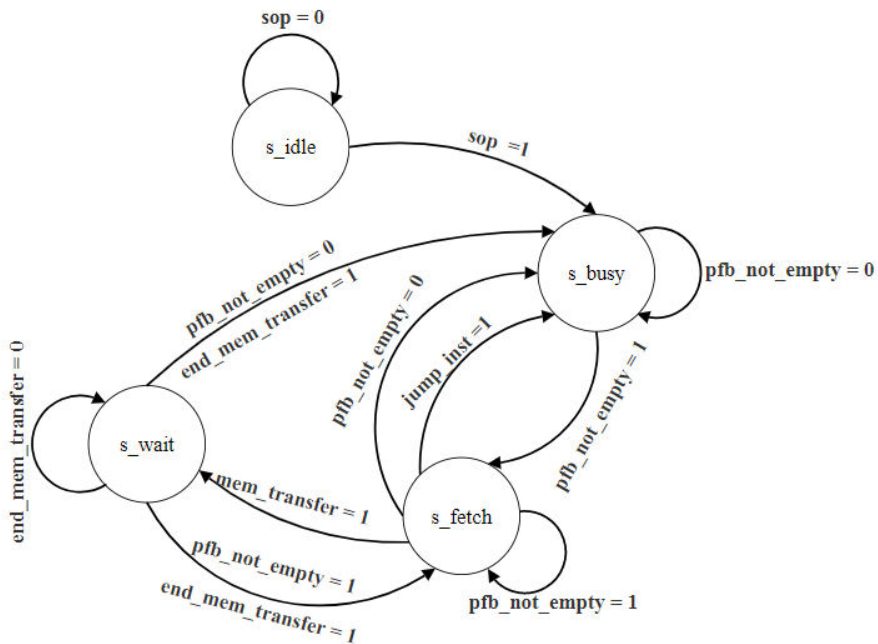

Figure 3.4: execution state of FSM

stays in the S_wait state until the memory transfer is done. If the memory transfer is done and if there is a data available in the prefetch buffer then the FSM will move to S_fetch state, if there is no data available in the prefetch buffer then the FSM moves to the S_busy state. If an error occurs during

13

fetching or executing the microcode, the FSM jumps directly into the S_error state.

3.5 Instructions

The instructions supported by Pacman are 8 bit wide, out of which upper 5 bits (Opcode) are allocated for instruction decode and lower 3 bits are allocated for operands. Instructions in Pacman are of variable length. MOVI, LDI, STI take 5 bytes. JUMP and JUMPC take 2 bytes, and all other instructions take 1 byte. Instruction set architecture is given in Appendix C.

3.6 Debug

The debug logic in the Pacman processor provides:

    1) Halt mode

    2) Two Hardware Address Break Points

    3) Single Step mode

3.6.1 Halt Mode Register

Halt Mode Register is used to halt the processor immediately irrespective of the instruction executed by the processor. This is a 32-bit register with halt_bit as a bit '0', halted as a bit '1' and the rest of the bits are tied to '0'. To halt the processor halt_bit should be programmed to '1' and to release the processor from halt state the bit should be programmed to '0' so that the Pacman starts executing from the point where it stopped. The halted bit gives the status of the processor.
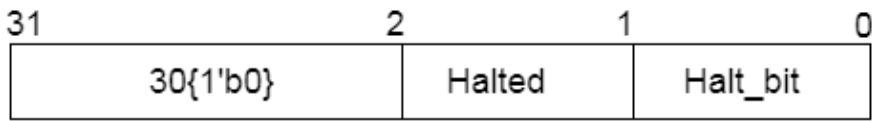
Figure 3.5: Halt Register

Halt_bit is a Read/Write bit while the halted bit is a Read-only bit.

The Halt mode register can be accessed through AHB bus with the address 0x000022C0.

Programming the register with "0x00000001" will halt the processor and disable halt mode we need to programme the register with "0x00000000".

3.6.2 Address Break Point Register

A breakpoint is a mechanism provided by debuggers to identify an address at which programme execution is to be halted[9]. Break Points are inserted by programmers to inspect the register content's, memory locations in the program execution to test the program and make sure that it is operating correctly. Break Points are removed after the program is tested. Two Address Break Point registers are implemented, and a Clear Break Point Enable register is used to control these two registers.

Address Break Point Register and Clear Break Point Enable register are 32-bit registers. Break Point Address Register is Read/Write register. Clear Break Point Enable register, a 32-bit register with 0th bit as Clear Break Point Enable and the rest of the bits are tied to '0'. Address Break Point Register is given with the address at which we need to put a break point and when this address equals with the PC value the processor halts at that particular address.
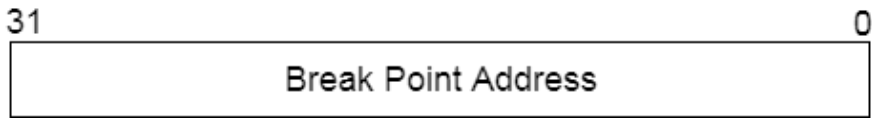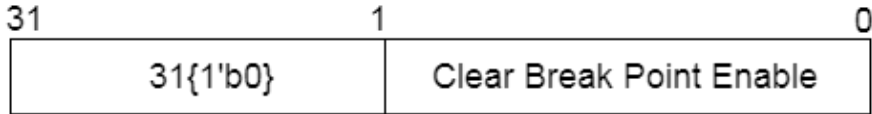
Figure 3.6: Break Point address Register

Figure 3.7: Clear Break Point Enable Register

The Reset value for Address Break Point register is "0xFFFFFFFF". The Address Break Point register1 can be accessed through AHB bus with the address 0x000022EC while the Address Break Point register2 can be accessed at address 0x000022F4. The two registers can be programmed at once with different addresses. To clear this break point, we need to program Clear Break Point Enable register with 0x00000000, and the processor starts executing from the halted point. The halted bit in the halt register gives the status of the processor.

3.6.3 Single Step Register

Single Step Register is a 32-bit register which has 3 bits SStep_en, SStep_go, SStep_ack and rest of the bits are tied to '0'. SStep_en, SStep_go are Read/Write bits while SStep_ack bit is a Read-only bit. The Single Step Register can be accessed through AHB bus with the address 0x000022F8.

To enable Single Stepping in Pacman, we need to program SStep_en bit to '1' which helps Pacman to enter into single step mode. Pacman after entering into the single step mode we need to program SStep_go bit to '1' each time so we can single step and debug each line of Pacman. SStep_ack gives the status of the Pacman.

16

For Pacman to disable single step mode, we need to Program SStep_en bit to '0'.

31	3	2	1	0
29{1'b0}	SStep_ack	SStep_go	SStep_en
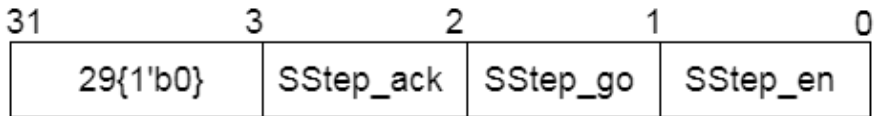
Figure 3.8: Single Step Register

I enabled the ability to read the values of all Registers, Accumulator and Carry flag in debug mode through AHB bus.

Tools and Design Flow

The process involves Behavioural simulation, RTL synthesis, FPGA synthesis and Verifying on Genie software. Different tools are used at each stage.

Figure 4.1: Design Flow

Figure 4.1 explains the high-level design flow of FPGA. In a practical situation, each step in the figure may be split into several smaller steps, and the design flow is iterated to ensure the functional correctness of the design.

4.1 Behavioural Simulation

The RTL coding for the Pacman processor was done using Verilog. The behaviour of the design is simulated using the HDL simulation tool Incisive from Cadence design systems. The codes are writ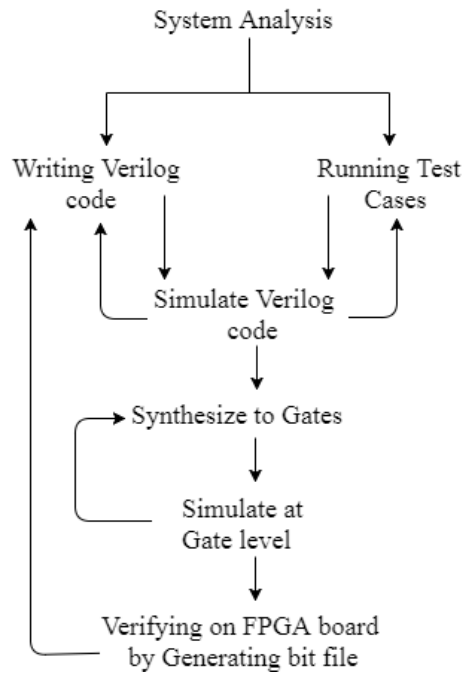ten in a Vim editor a SOC file list was generated which gives a path to all the modules in a single file. I gave this file as input to the Incisive which compiles and executes the Verilog code and displays waveform. These waveforms are used only for the functional verification of the design. There modules are validated using test bench which is written in Verilog, test bench provides stimulus to the processor and generates output.

4.2 ASIC Synthesis

The RTL code synthesised using the synthesis tool RTL Compiler (RC) from Cadence. The RTL synthesis was done at 22nm technology. The synthesis creates a gate-level circuit based on the RTL model, which meets design constraints such as timing, area and power consumption. The constraints are provided to the tool by a script file, which contains command specific to the tool. The tool synthesises the circuit using the standard cells, providing standard functionality such as logical operations, adders, flip-flops, buffers. The standard cells are provided in the form of a library. The synthesis script consists of configuration variables, library variables, read design, constraint, compile, reports, write design. The various steps involved in the design are[6]:

- The designer describes design at a high-level using RTL which can be in Verilog or VHDL.

- The RTL is converted by the synthesis tool to unoptimised, unintended and internal representation.

- The logic is now optimised to remove the redundant logic from the synthesis tool.

- The optimised logic is now represented in the form of gates, using the cells provided by the technology library.

- The technology library contains library cells designed by the foundry.

- Design constructs typically include Area, Timing, and Power.

  1) Timing: The design should meet certain timing requirements. The timing analyser checks the timing.

  2) Area: We need to optimise as much area as possible for any design.

  3) Power: The power dissipation of the design should not exceed the threshold.

- After mapping the technology constructs, an optimised gate-level netlist is generated.

- The designer modifies the RTL or re-constrain the design until the design meets the desired results.

4.3 FPGA Synthesis

FPGA Synthesis was done using Vivado tool. Vivado tool is timing driven and optimised for performance and memory usage. Vivado design suite solution is native TCL with support from Synopsis Design Constructs (SDC) and Xilinx Design Constructs (XDC) formats. Vivado tool supports Verilog, VHDL and System Verilog for synthesis enables easier FPGA adoption.

The main design flow features are:
- Vivado Synthesis
- Vivado Implementation
- Bitstream generation

4.3.1 Vivado Synthesis

Synthesis is a process of transforming RTL design into a gate-level representation. The tool manages the run data automatically, allowing repeated run attempts with varying Register Transfer Level (RTL) source versions, target devices, synthesis or implementation and timing or physical constraints[7]. The netlist generated by the tool can be regarded as efficient netlist due to logic optimisation, register load balancing and other techniques to enhance timing performance.

4.3.2 Vivado Implementation

The Vivado design suite implementation process transforms a logical netlist into a place and route design, followed by bitstream generation. The implementation process consists of the following sub-process[8]:

Opt Design: Optimises the logical design to make sure that it fits into the target device.

Power Opt Design: Optimises the logical design to reduce the power so that it satisfies the demands of the target device. This process can be optional.

Place Design: In this step, the design is placed onto the target device.

Route Design: In this step, the design is routed onto the target device.

Write Bit Stream: This is the last step and generates bitstream to the target device.

4.3.3 Bit File Generation

FPGA has hundreds of components. To access a particular pin, we need to create a .xdc constraints file. This constraint file will let the tool know about the access to that particular pin. To assign a pin, we use keywords "set_property PACKAGE_PIN" followed by the pin location.

In figure 4.2 "tdo_pad_o" and "trst_pad_i" are the two port pins that are connected to R32 and V35 pins on the FPGA board.

```
set_property PACKAGE_PIN R32 [get_ports tdo_pad_o]
set_property PACKAGE_PIN V35 [get_ports trst_pad_i]
```

Figure 4.2: Package Pin

In figure 4.3, LVCMOS is low voltage metal oxide semi-conductor and is a low voltage class of CMOS technology. This is an IO Standard. Device geometries are decreased for an integrated circuit to obtain better performance and low voltage. Due to these, the input voltages are also decreased. LVCMOS is one of the power supply voltage and interface standard that was defined for decreased voltages.

```
set_property IOSTANDARD LVCMOS18 [get_ports tdo_pad_o]
set_property IOSTANDARD LVCMOS18 [get_ports trst_pad_i]
```

Figure 4.3: IO Standard

The bit file was generated after setting up the xdc constraint file and adding the required Verilog files into the design. The generated bit file is flashed onto the VC707 board using iMPACT tool.

Simulation and Synthesis Results

Verification is a process to demonstrate the functional correctness of the design. In the design cycle, almost 70% of the time is spent on verification. Every increasing complexity of the design makes verification harder. Verification is always on a critical path for product design.

Test benches are written to simulate the design. Test benches are written in OVM, UVM, Verilog and System Verilog. In this project, I wrote a test bench in System Verilog. We need to write different test cases and test the design to make sure that the design is bug-free. The test cases are written in 'C', and the register access is also done through this file. The test cases are then converted into System Verilog by using API and DPI. Whenever there is a change in the 'C' file, we need to run the perl script so that the change in the 'C' file is effected during simulation. I did two types of verification simulation verification and FPGA verification.

5.1 Simulation

In Figure 5.1, we can see the two registers, break point address 1 and break point address 2, are given with break point addresses 0x00000006 and 0x00000019. Whenever the Programme Counter hits 0x00000006, we can see Pacman is in a halt state until the break point enable register is disabled. When the break point enable register is disabled, Pacman starts running until the Programme Counter hits the address in the second break point address register. When the Programme Counter hits the second break point address, the Pacman halts. The Pacman execution starts again when the

required registers are disabled. During this break point mode, we can read the registers.
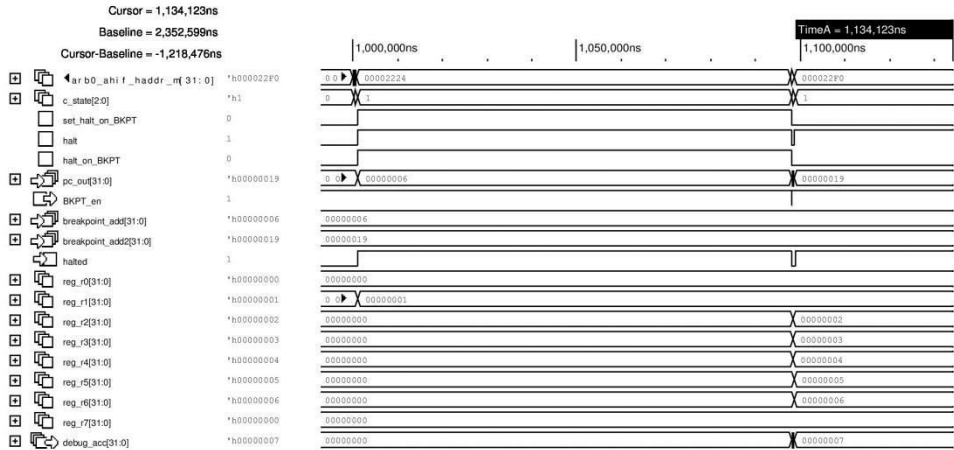

Figure 5.1: Break Point

In Figure 5.2, we can see that the Pacman is in single step mode and whenever the single step go signal is enabled, the Pacman is single stepping. We can also read registers during single stepping.
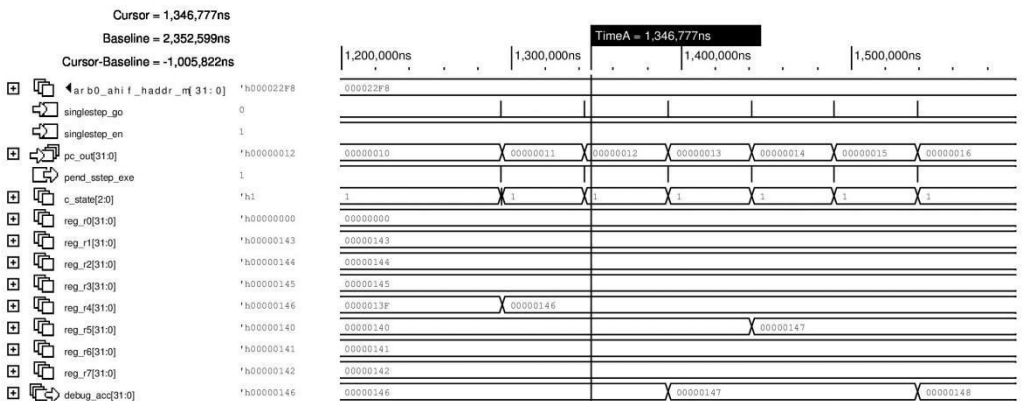

Figure 5.2: Single step

5.2 ASIC Synthesis

The design ran at frequency 125Mhz and 22nm technology.

Gate count is the number of gates used by each module which was determined by the synthesis tool. The results are tabulated in Table 5.1. The gate count depends on the number used in the design and also depends on the optimisation of the tool. Each gate is equivalent to 2-input NAND gate which is equal to 4 transistors.

Gate count = Total Area/(Area of $Nand_2$ gate)
Area of $Nand_2$ gate is 0.19968

Module	Area	Gate Count
Pacman top	10051	50336
Interrupt router top	4709	23582
Execution Unit	1966	9845
AHB Master	1878	9405
Prefetch buffer	1377	6896

Table 5.1: Gate Count

5.3 FPGA Synthesis

FPGA Synthesis was run using the tool, Vivado 2016.3. All the required Verilog files and xdc constraint file were added to the design. FPGA Synthesis, Implementation, and Writing Bit Stream were done using the Vivado tool. The utilisation report on FPGA is given in Table 5.2. We can

clearly see that the design is utilising 56.94% of all the LUTs available on FPGA.

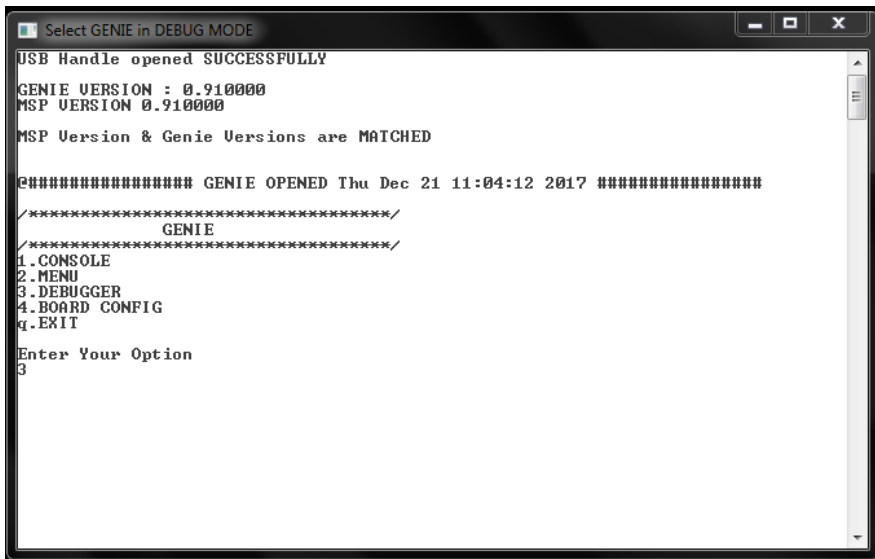| Site type	Used	Fixed	Available	Util%
Slice LUTs	172882	0	303600	56.94
LUT as Logic	172880	0	303600	56.94
LUT as Memory	2	0	130800	0.01
LUT as Distributed RAM	0	0		
LUT as Shift Register	2	0		
Slice Registers	68637	0	607200	11.30
Register as Flip Flop	68637	0	607200	11.30
Register as Latch	0	0	607200	0.00
F7 Muxes	17607	0	151800	11.60
F8 Muxes	8736	0	75900	11.51

Table 5.2: Utilisation Report

The timing summary for the design is given as for a setup time the Worst Negative Slack (WNS) is 7.110ns while the Total negative slack is 0ns. For a Hold time, the WNS is 0.095ns while the Total negative slack is 0ns. For a

Pulse width, the Worst Pulse width negative slack is 1.100ns while the Total pulse width negative slack is 0ns.

5.4 Genie

The genie tool is developed by Ineda Systems which works with Catalyst. In Figure 5.3, there are different functioning options the tool supports, and I selected the debugger option to debug Pacman.

Figure 5.3: Genie in Debug Mode

Selecting the debugger option enters genie into debug mode. In Figure 5.3, by giving the option $jtagid to genie gives the id of the flashed programme onto the FPGA to ensure that we are accessing the correct programme. As I said earlier, the address between 0x00000000 to 0x00001FFF are used to access SRAM. Here I am accessing SRAM and writing the microcode into the SRAM.

Figure 5.4: Genie tool writing Micro code

In Figure 5.4, I am programming to enable the Interrupt 0 Register which can be accessed through address 0x00002000. The address 0x000022EC gives access to Address Break Point register1 which is programmed with 0x00000010. The address 0x000022F4 gives access to Address Break Point register2 which is programmed with address 0x00000014.

The address 0x00003000 gives access to the system register when programmed this register with '1' generates an interrupt to the Pacman. The address 0x00002224 gives access to the Pacman control register start bit by enabling this bit to '1' Pacman starts working.

The address 0x00002230 gives Read-only access to the programme counter. We can check here if the Pacman is halted at the desired location. During this process, we can read the values present in Registers R0-R7 and can also check the status of the Carry flag.
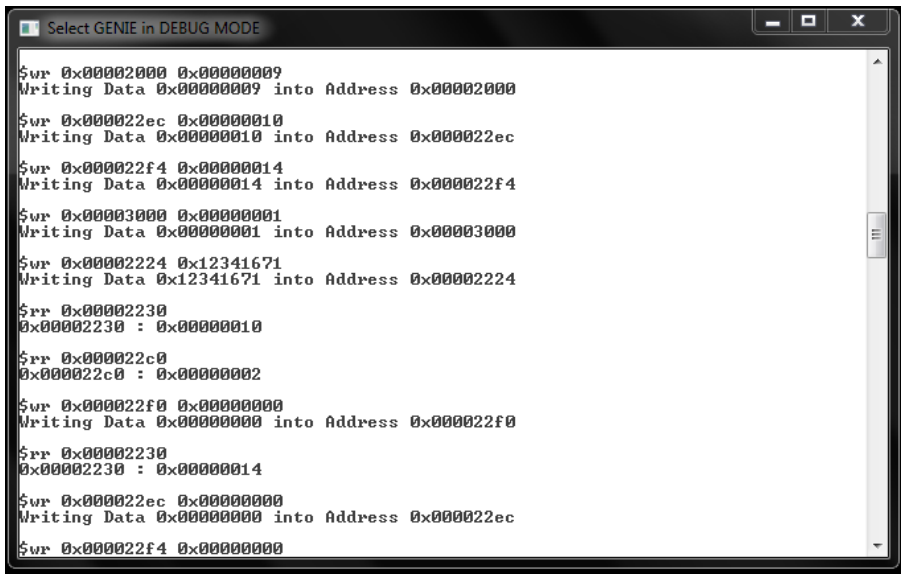
Figure 5.5: Genie tool in Break Point mode
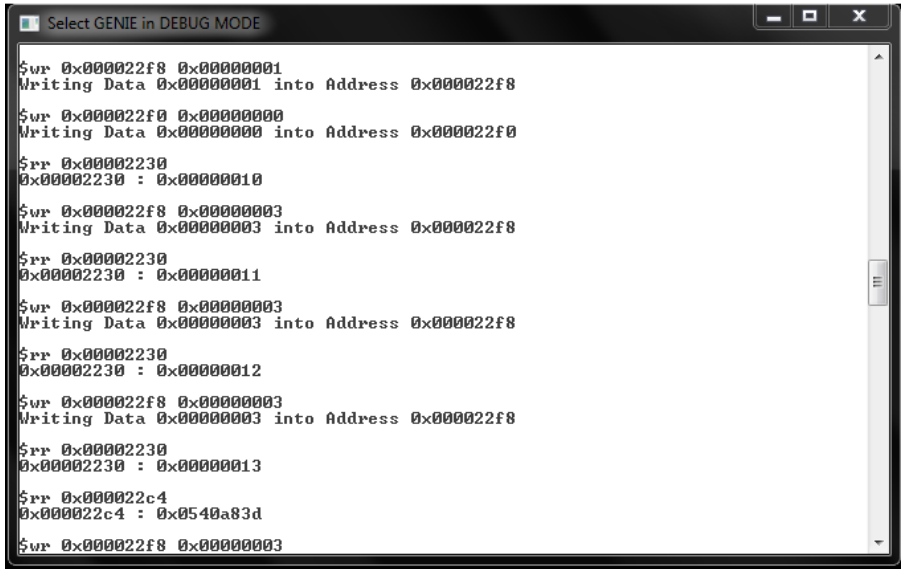
The address 0x000022F0 will give access to the clear break point register. To disable the break point we need to programme this register with 0x00000000. Pacman halts again whenever it hits the second break point which is at address 0x00000014.

We can see in Figure 5.5 the address 0x00002230 which is programme counter shows that the Pacman was halted at the desired location. Again to disable this break point we need to programme the clear break point register with 0x00000000.

If the microcode is in a loop, and if the programme counter again hits any one of the addresses of the break point address register, it once more halts. To disable this feature, we need to program break point address registers with 0x00000000.

Figure 5.6 shows the working of a single step. The address 0x000022f8 gives access to the single step register. To enter into the single step, we need

to program this register with 0x00000001. For single stepping, we need to program this register with 0x00000003.

Figure 5.6: Genie tool in single step mode

Conclusion and Future Work

An interrupt controller which can prioritise the interrupts and can also resolve the interrupts without the help of the main processor has been designed. It can also act as a secondary processor in the system. It is a low power processor designed with a lesser number of gates.

One important feature to be noted in the implemented design is the debug feature where the interrupt controller and priority resolver supports halt, break point, and single step modes. The main purpose of this interrupt controller is to give more leverage to the processor without disturbing its functioning during an interrupt.

The debugger used in this project is Catalyst, helps in debugging Pacman through Genie tool. In Future, we can add GDB and Open OCD plugins to the Pacman. This helps in debugging Pacman through Bus Blaster using Eclipse.

JTAG

The Joint Test Action Group (JTAG) was formed in 1985 to create Printed Circuit Board (PCB) and Integrated Circuit (IC) standards. The latest version of their proposal was approved by the Institute of Electrical and Electronic Engineers (IEEE) as IEEE STD. 1149.1-2013, IEEE Standard Test Access Port and Boundary-Scan Architecture. The standard was created to test the devices functionality and component interconnects. This chapter is intended to give the reader enough understanding and operation of Jtag.

Jtag is hardware which helps your computer to communicate directly to the chip on the board. Jtag is used for Debugging, Programming, and testing on all embedded devices.

A.1 JTAG Interface Signals

Signal	Description
Test Clock (TCK)	Serial clock signal
Test Mode Select (TMS)	Controls the Jtag state machine
Test Data Input (TDI)	Serial data input to the design
Test Data Output (TDO)	Serial data output from the design
Test Reset (nTRST)	Optional reset signal

Table A.1: JTAG interface signals

Table A.1 shows the signals defined by Jtag standard[2].

TCK - Test Clock: The TCK pin is used to load test data from the TMS pin, the test data on the TDI pin on the rising edge of TCK, the test data on the TDO pin on the falling edge of TCK.

TMS - Test Mode Select: The TMS pin on the Jtag is the input pin which clocks through on the rising edge of TCK determines the state of the TAP Controller.

TDI - Test Data Input: The TDI pin on the Jtag is the connection on to which the test data is passed. TDI is fed with the serial input data which is then fed either into the instruction register or data register depending on the state of the TAP Controller.

TDO - Test Data Output: The TDO pin delivers the serial data which is either from the instruction register or data register depending on the state of the TAP Controller. The data on the TDO pin is from the TDI pin with data shifted by a number of clock cycles, depending on the length of an internal register.

TRST - Test Reset: The TRST pin is the optional active low test reset pin on Jtag. This pin permits active asynchronous TAP controller initialisation without affecting other device or system logic.

A.2 JTAG State Machine

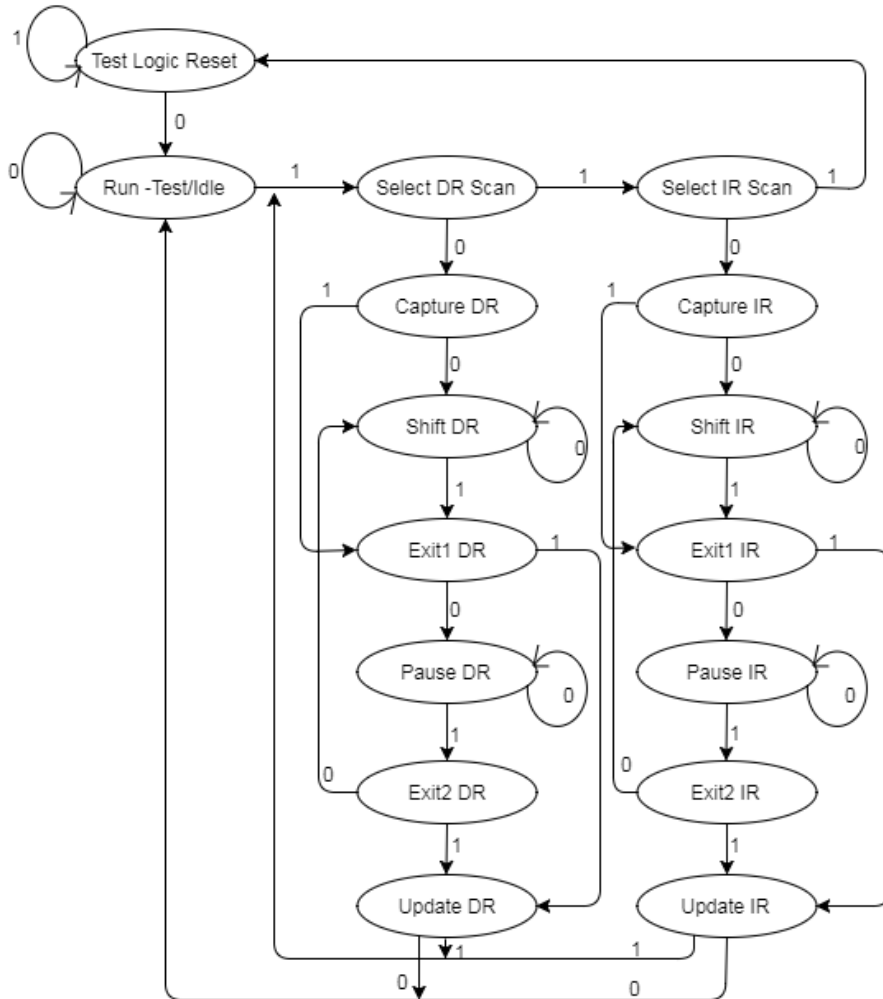**Figure A.1:** Jtag State Machine

All Jtag operations are controlled by a State Machine. The state machine is driven by Test Mode Select (TMS) signal and is clocked by the rising edge of Test Clock (TCK). When a test session is initiated, the bus master has to initialise all connected TAP controllers by putting them into Test Logic Reset (TLR) state. TLR is set either by forcing the nTRST to active low or

by clocking the TCK five times with the TMS kept high which will set the TLR state. Once the TLR state is set, the identification register (IDCODE) or the bypass register is set. If the TMS is low on the rising edge of TCK, then the TLR state is moved to the Run Test/Idle state. Depending on the currently selected instruction either the test runs or it remains idle. From Run Test/Idle state if TMS is '1' Select Data Register (DR) Scan state is reached which is similar to the Select Instruction Register (IR) Scan state. Exit1 DR/IR, Exit2 DR/IR are temporary states where no operation occurs, used to select different paths in the state machine. In Capture DR state the currently selected test data register is parallel loaded if the data is appropriate or else the data is not loaded from the register. In Capture IR state a fixed value of b'01 is loaded into the least significant bits of IR, design specific values are put into the remaining IR bits. Once the Shift DR or Shift IR state is reached, the TAP Controller takes TMS low and starts outputting the data on the falling edge of TCK. The device under test will sample TDI on the rising edge of TCK and will stay in shift IR until TMS is low. No test operations occur when the TAP Controller is in Pause DR/IR state. On the falling edge of TCK in the Update DR state, the current value of the register is latched output if this is required for currently selected test data registers. Similarly, with the falling edge of TCK in the Update IR state, the current value of the register is latched out. Latching a new value on the IR parallel outputs makes this value the new current instruction[3].

A.3 JTAG Instructions

The Jtag standard requires several instructions on the device, but most of these are unimportant for Pacman Debugging. The following instructions are used in ARM debugging systems

IDCODE: IDCODE is used to select the device identification register instead of the Data register. This IDCODE is given a particular binary value which helps in identifying the chip. The binary value of IDCODE for a particular chip is set by the designer.

BYPASS: BYPASS register is a single bit register that is placed in between TDI and TDO which helps in passing the information from TDI to TDO. This instruction allows checking other devices in the Jtag without any unnecessary over the head[3].

EXTEST: The EXTEST instruction makes the boundary-scan register as the current test data register. Signals that are driven from outside of the component are loaded into the boundary-scan register during the falling edge of TCK in Capture DR state, and the signals that are loaded from the component are loaded into the boundary-scan register during the falling edge of TCK in Update DR state. This allows signals from the system to the component to be captured, and known values to be applied to signals driven from the component to the system[3].

INTEST: The INTEST instruction also selects boundary-scan register but is used to select the instructions that are driven out of the component, and known values applied to signals driven into the component.

Advanced High-performance Bus

AHB is a new generation of Advanced Microcontroller Bus Architecture (AMBA) bus by ARM which is intended to address the requirements of high-performance synthesisable designs. AHB is for high-performance, high clock frequency systems including:

- burst transfers

- split transactions

- single cycle bus master handover

- non-tristate implementation

- wider data bus configuration (64/128 bits)

B.1 Bus Interconnection

The AMBA AHB bus protocol is designed to be used with a central multiplexer interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexer, which selects the appropriate signals from the slave that is involved in the transfer[4].

The AHB Arbiter that I am using in the design can handle up to 16 Masters and 16 Slaves. JtagtoAHB acts as Master 1 while SRAM acts as Slave 1, Pacman acts as Slave 2, System Register acts as Slave 3. SRAM is assigned with memory 8K which can be accessed through address location from

0x00000000 to 0x00001FFF. Pacman is assigned with memory 4K which can be accessed through address location from 0x00002000 to 0x00002FFF. System Register is assigned with memory 4K which can be accessed through address location from 0x00003000 to 0x00003FFF. All the remaining Master and Slaves are disabled.

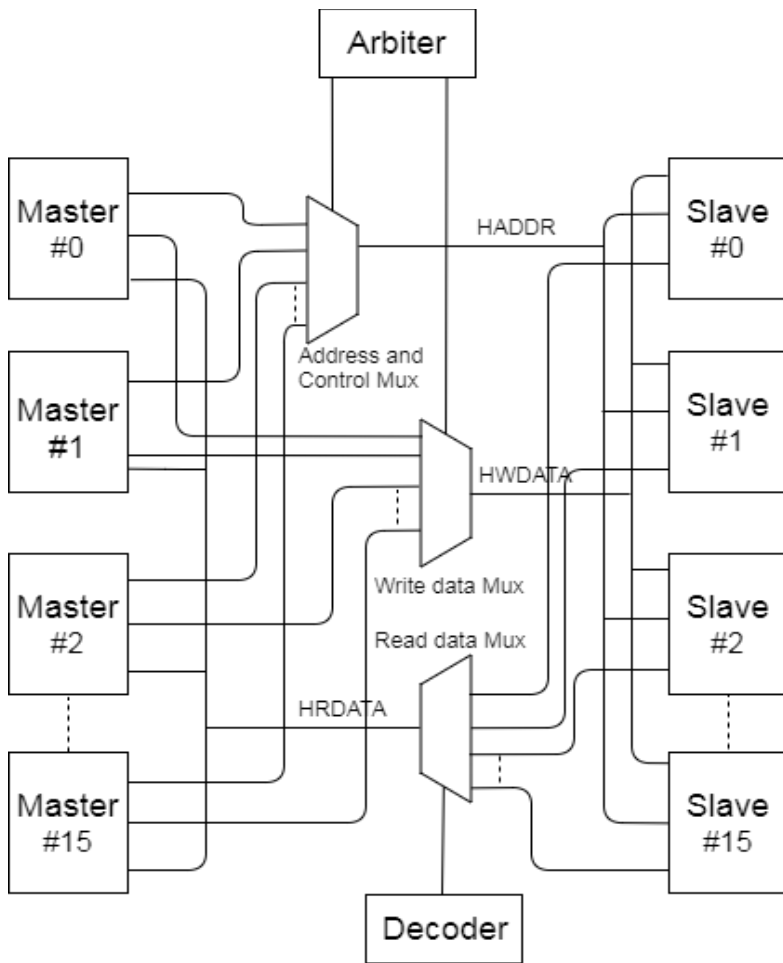Figure B.1 illustrates the structure of an AHB design with 16 Masters and 16 Slaves.

Figure B.1: Multiplexer Interconnection

B.2 AHB Operation

Before the AHB transfer, the bus master needs to be granted access to the bus. This process is started by the master by sending a request signal to the arbiter. Then the arbiter grants access and indicates master when to use the bus.

The granted master starts AHB transfer by driving the address and control signal. These signals provide the required information and also indicates if the transfer forms part of a burst. There are two types of burst transfers.

- Incrementing burst, which does not wrap at address boundaries
- Wrapping burst, which wraps at address boundaries.

A write data bus is used to write data from Master to Slave, while read data bus is used to read data from Slave to Master.Every transfer consists of address and control signal, one or more cycles for the data.

The address cannot be extended therefore all slaves will sample address. The data, however, can be extended with the help of HREADY signal. When LOW, these signal insert wait states, which allow extra time for the slaves.

During the transfer, the slave shows the status using the signal HRESP. HRESP is a two-bit signal. There are three types of response states:

OKAY - The OKAY response indicates that the transfer is normally going and the HREADY signal goes high indicating that the transfer has completed successfully.

ERROR - The ERROR response indicates that the error occurred during the transfer and transfer has been unsuccessful.

RETRY AND SPLIT - Both RETRY and SPLIT indicates that the transfer will not complete immediately, but the bus master should continue to attempt the transfer continuously.

In normal operation, the transfer is completed in particular burst before the arbiter grants bus access to another master.

B.2.1 Basic Transfer

AHB transfer consists of two distinct sections:

- The address phase lasts for one clock cycle.

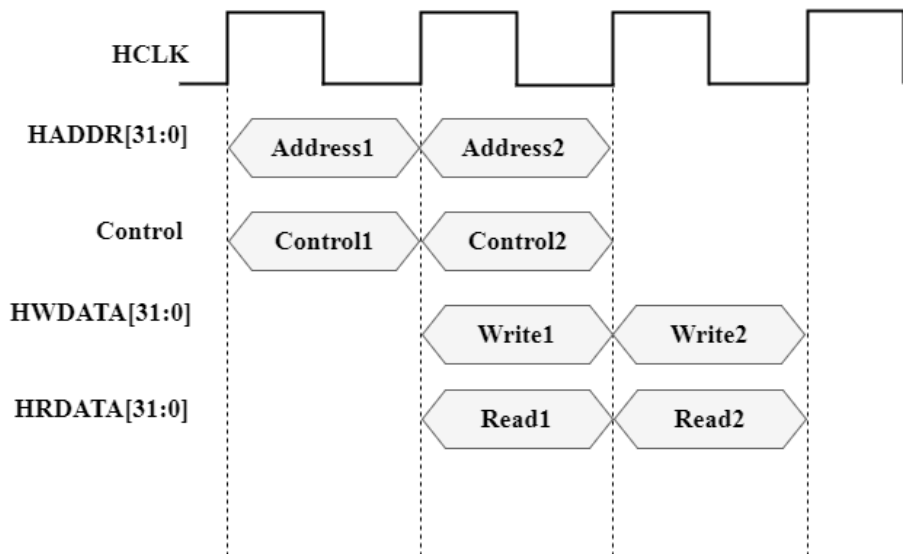- The data phase can last for more than one clock cycle with the help of HREADY signal.

Figure B.2: AHB Basic Transfer

B.2.2 Burst operation

Four, eight, and sixteen beat bursts are defined in the AHB protocol as well as the undefined length and single transfers. Both incrementing and wrapping bursts are supported in this protocol.

- Incrementing burst access sequential locations with the address of each transfer in the burst being an increment to the previous address[5].

- An increment burst can be of any length, but the upper limit is set by the fact that it must not cross the 1KB boundary.

- Wrapping burst, if the start address of the transfer is not aligned to the total number of bytes in the burst, then the address of the transfer in the burst will wrap when the boundary is reached.

  For example, a 4-beat wrapping burst of word access will wrap at 16-byte boundaries.

  Therefore, if the start of the transfer is 0x34, then it consists of 4 transfers to addresses 0x34,0x38,0x3C and 0x40.

Burst information is provided using HBURST[2:0] signal and the eight possible types are defined in Table B.1.

HBURST	TYPE	DESCRIPTION
000	SINGLE	Single Transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat increment burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat increment burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat increment burst

Table B.1: Transfer Types

Burst must not cross 1KB address boundary. Therefore, it is important to make sure that the masters do not start with a fixed length incrementing burst which would cause this to cross the address boundary.

All transfers within an address boundary must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries; halfword transfers must be aligned to halfword address boundaries.

B.3 AHB Arbiter

The arbitration mechanism is to make sure that only one master has access to the bus at one time. The arbiter performs this function by observing different requests that it received from different masters and deciding which is currently the highest priority master requesting the bus.

Each master also generates HLOCK signal which is used to indicate that the master required exclusive access to the bus.
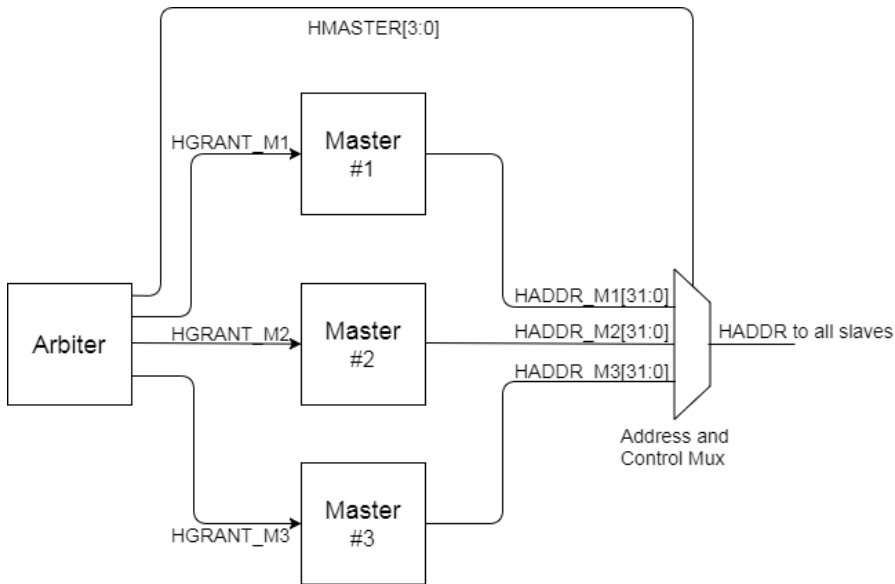
Figure B.3: Bus master grant signal

B.3.1 Arbitration Signals

HBUSREQx - The bus request signal is used by a bus master to request access to the bus.

HLOCKx - The lock signal is asserted by a master at the same time as the bus request signal. This indicates to the arbiter that the master is performing a number of indivisible transfers and the arbiter must not grant any other bus master access to the bus.

HGRANTx - The grant signal is generated from the arbiter and indicates that the appropriate master is current highest priority master requesting the bus.

HMASTER[3:0] - This signal is used by the arbiter to indicate which master is currently granted the bus and this signal is used to control the address and control multiplexer.

HMASTLOCK - This signal is used by the arbiter to indicate that the current transfer is a part of locked sequence

B.4 AHB Decoder

A decoder is used to provide an HSEL signal for each slave on the bus. A slave must sample the address, control signal and HSEL signal when HREADY signal is high, indicating the current transfer is completing.

Under certain circumstances, it is possible that HSEL will be asserted when HREADY is low, but the selected signal will have changed by the time the current transfer completes. A minimum address space that can be allocated for a slave is 1KB. All bus masters are designed in a way such that they will not perform incrementing burst over 1KB[4].
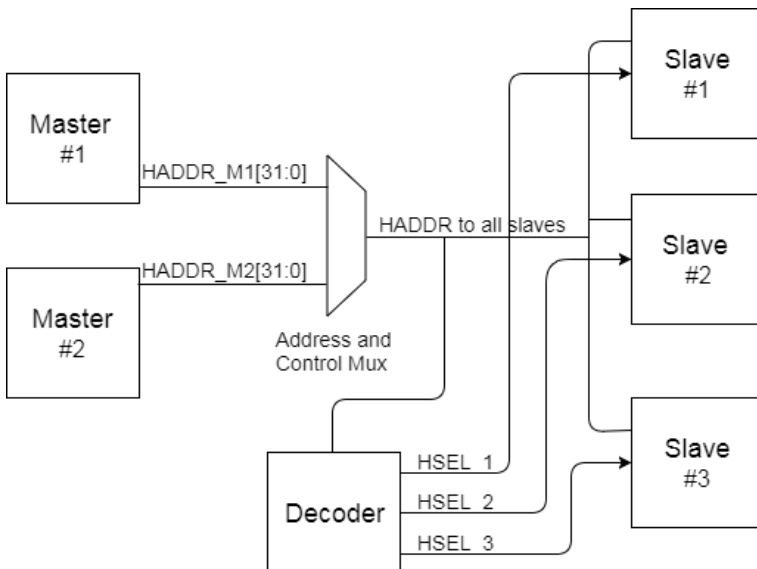
Figure B.4: Slave select signals

Instruction Set Architecture

C.1 Instruction Set

Sl. no	Instruction	Description
1	NOP	No Operation
2	END	End of Programme
3	SETB	Sets GPO with '0' or '1'
4	WAIT	Wait for event port to become '1'
5	LD	Load data to accumulator from the address given by the register
6	ST	Stores data from accumulator to the address given by the register
7	SUBA	Subtract accumulator from the register. The result is stored in accumulator
8	MOVI	Moves 32 bit data into accumulator or into registers
9	STI	Stores data from accumulator or registers from the address specified
10	LDI	Loads data to accumulator or registers from the address specified
11	JUMP	The offset is signed value. Jump to a word-aligned address from the current location
12	JUMPC	The offset is signed value. On condition flag set by previous instruction, jump to word-aligned address from the current

		location
13	ADD	Add register to accumulator
14	SUB	Subtract register from accumulator
15	AND	Logical AND Rn with accumulator
16	OR	Logical OR Rn with accumulator
17	XOR	Logical XOR Rn with accumulator
18	GT	If [acc] > [Rn] sets conditional flag to '1'
19	LT	If [acc] <[Rn] sets conditional flag to '1'
20	EQ	If [acc]=[Rn] sets conditional flag to '1'
21	EQZ	If [acc]=32'd0 sets conditional flag to '1'
22	LS	Left Shift accumulator along with conditional flag bit
23	RS	Right Shift accumulator along with condition flag bit
24	MOVF	Move data from register to accumulator
25	MOVT	Move data to register from accumulator
26	CLR	Clear accumulator contents
27	ADDI	Increments accumulator contents by 2^n
28	SUBI	Decrements accumulator contents by 2^n

Bibliography

[1]   K.Yash. History of software bugs and debugger.
       http://www.ksyash.com/2011/01/178/ [Online; Accessed16- Oct-
       2017]

[2]   http://www.interfacebus.com/Design_Connector_JTAG_Bus.html
       [Online Accessed; Jtag Interface signals]

[3]   https://www.xjtag.com/about-jtag/jtag-a-technical-overview/
       [Online Accessed; Jtag Instructions and State Machine]

[4]   http://soc.eecs.yuntech.edu.tw/Course/SoC/doc/amba.pdf
       [Online Accessed; AMBA Specification, AHB Bus]

[5]   http://mapl.nctu.edu.tw/course/ESL_2008/files/Lecture10.pdf
       [Online Accessed; AHB PPT]

[6]   http://eacharya.inflibnet.ac.in/data-server/eacharya-
       documents/53e0c6cbe413016f23443704_INFIEP_33/13/LM/33-13-
       LM-V1-S1__synthesis_design_flow.pdf [Online Accessed; Synthesis
       Design Flow]

[7]   https://www.xilinx.com/support/documentation/sw_manuals/xilinx
       2016_3/ug901-vivado-synthesis.pdf [Online Accessed; Vivado
       Synthesis PDF]

[8]   https://www.xilinx.com/support/documentation/sw_manuals/xilinx
       2014_1/ug904-vivado-implementation.pdf[Online Accessed; Vivado
       Implementation PDF]

[9]   http://processors.wiki.ti.com/index.php/How_Do_Breakpoints_Work
       [Online Accessed; Working of Break Points]