A 32-Bit Parameterized Leon-3 Processor with Custom Peripheral Integration

By
Talal Khaliq
00000118396

Supervisor
Dr. Awais M. Kamboh
Department of Electrical Engineering

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Electrical Engineering (MS EE)-7

In
School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(November 2018)

Approval

It is certified that the contents and form of the thesis entitled "**A 32-Bit Parameterized Leon-3 Processor with Custom Peripheral Integration**" submitted by **Talal Khaliq** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Awais M. Kamboh**
Signature: ____________________

Date: ____________________

Committee Member 1: **Dr. Khawar Khurshid**

Signature: ____________________
Date: ____________________

Committee Member 2: **Dr. Ahmad Salman**

Signature: ____________________
Date: ____________________

Committee Member 3: **Dr. Salman Abdul Ghafoor**

Signature: ____________________
Date: ____________________

Abstract

The main purpose of this thesis is to integrate custom cores as peripherals as Co-Processors (CPs) to a processor. For this purpose, a 32-bit open source processor was be tested on an FPGA. For proof of concept, a simple open source 8-bit processor was selected to run on FPGA using custom instructions written in C. Thereafter, we tried to change specifications and peripherals of the peripherals like timers, UART, SPI etc. For 32-bit processor, starting with an open source processor design for Leon 3, the study involved synthesis of code, compilation of program, and test of pre-configured peripherals on an FPGA. Once a decent level of understanding was achieved, a new peripheral was integrated into the processor to enhance the processor's capabilities, and to adapt them for better performance in a given domain of applications. Using study of processor architecture of Leon 3, we tried to design our own processor with peripherals, memory management module and custom compiler.

Dedication

I dedicate this thesis to parents, wife and daughter.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Talal Khaliq**
Signature: ________________

Acknowledgment

I would like to thank my advisor and colleagues for their constant support in and research phase.

Table of Contents

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Problem Statement 1
1.3 Proposed Approach 2

2 Literature Review **3**
2.1 Introduction 3
2.2 Instruction Set and RISC 3
2.3 Peripheral Integration to Processor 4

3 8-bit Processor (8051 Microcontroller) **5**
3.1 Introduction 5
3.2 Oregano Systems mc8051 5
3.3 Tools Required for Synthesis and Simulation 6
3.4 mc8051 Top Module (Synthesis) 8
3.5 Work on Keil C51 (Microcontoller) 11
3.6 Conversion from HEX to COE 11
3.7 Synthesis and Implementation on FPGA 12
3.8 Simulation 15
3.9 Configuration of mc8051 for extra peripherals 15

4 Leon3 Introduction **18**
4.1 Introduction 18
4.2 Evaluation of Processors(SoC) 18
4.3 Evaluation of Bus Architectures 19
4.4 SPARC Version 8 ISA 20
4.5 Leon3 Introduction and Pipeline 20
4.6 AMBA Bus Architecture 24
4.7 Example Template Design 24
4.7.1 Library (Source Code) and Toolchain 25
4.7.2 Example Template Configuration and Implementation . 26

4.7.3 Configuration for Minimal, General Purpose and High Performance Processor 28
4.8 Software Development (BCC) 29
4.8.1 Software Development (GRMON Debugger) 30
4.8.2 Software Development (PROM Programmer) 30
4.8.3 Software Development (TSIM Simulator) 32

5 Leon3 Extension and Customization 33
5.1 Introduction 33
5.2 Library Structure 33
5.3 Understanding and Working of AMBA Bus 35
5.4 VHDL Generics and Link with Top Module 35
5.5 xconfig extension 36

6 Peripheral Interface (Introduction) 39
6.1 Introduction 39
6.2 Memory-Mapped Interface 39
6.3 Coprocessor Interface 41

7 Memory-Mapped Interface (AMBA APB) 42
7.1 Introduction 42
7.2 Advanced Peripheral Bus (APB) Architecture 42
7.3 IP Core (Co-Processor) APB Interface 43
7.4 Software Interface 47
7.5 Register Example 47

8 Multiplier and its APB Integration 51
8.1 Introduction 51
8.2 Shift and Add Multiplier 51
8.3 System Design and Behavioral Model with N Parametrization 53
8.4 Synthesis and Simulation in Xilinx 56
8.5 APB Integration 56
8.6 Multiplier APB Integration 57
8.7 Software Integration 59
8.8 Hardware / Software Verification: 62
8.9 N bit Parametrization for APB Interface 63
8.10 GUI Control 63

9 Memory-Mapped Interface (AMBA AHB) 66
9.1 Introduction 66
9.2 Advanced High-Performance Bus (AHB) 66

9.2.1 AHB Master . . . 67
9.2.2 AHB Slave . . . 67
9.2.3 AHB Decoder . . . 67
9.2.4 AHB Multiplexor . . . 69
9.3 Co-Processor AHB Slave Interface . . . 69
9.3.1 AHB Slave Data Transfer with enhanced features . . . 69
9.4 Software Interface . . . 70
9.5 Register Example . . . 72

10 AES-128 AHB Interface 75
10.1 Introduction . . . 75
10.2 AES-128 Synthesis and Simulation . . . 75
10.3 AHB Integration . . . 75
10.4 Software Integration . . . 79
10.5 Verification . . . 79

11 Conclusion and Future Work 82
11.1 Integration Examples . . . 82
11.2 Future Work . . . 82

List of Figures

2.1 von Neumann Architecture 4
2.2 Harvard Architecture 4

3.1 Oregano 8051 Core Top 7
3.2 Target 8051 Microcontroller in Keil c51 8
3.3 RAM Configuration in Xilinx 9
3.4 ROM Configuration in Xilinx 10
3.5 XRAM Configuration in Xilinx 10
3.6 8051 Top Module (Plan Ahead Pre-Synthesis) 11
3.7 BLINKY.c in Keil c51 IDE 12
3.8 HEX to COE Conversion in Command Prompt 13
3.9 COE file load in Xilinx Generated ROM Core 13
3.10 8051 Core RTL Schematic in Xilinx 14
3.11 Comparison on different design strategies 14
3.12 Fibonacci Code simulation on 8051 Core 15
3.13 REG51 Special Function Registers 16
3.14 Default I/Os:74 and C_IMPL_N_TMR=1 17
3.15 Default I/Os:82 and C_IMPL_N_TMR=2 17

4.1 Leon3 Integer Unit 22
4.2 Leon3 7 Stage Pipeline 23
4.3 AMBA Shared Single Bus 24
4.4 Leon 3 in Spartan 3E Template 25
4.5 Xilinx Vertex-5 ML507 27
4.6 Export Display to XWin Server 27
4.7 Leon3 Design Configuration GUI(xconfig) 28
4.8 Processor Comparison on Area Utilized and Timing 29
4.9 PROM hello.exe loaded and run 31
4.10 TSIM running hello.exe on Leon3 environment 32

5.1 Library showing scripts for different vendors 34

5.2 Library showing scripts for different files in AMBA folder . . . 34
5.3 New files in AMBA folder 35
5.4 Generation of components in Top Module 36
5.5 apb_example.in files 36
5.6 apb_example.in 36
5.7 apb_example.in.help 37
5.8 apb_example.in.h 38
5.9 apb_example.in included in config.in (Folder:ML50x) 38
5.10 Modified xconfig 38

6.1 Memory-Mapped Interface (AHB and APB) 40
6.2 Coprocessor Interface 40

7.1 AHB and APB Bus Control 43
7.2 AHB to APB Master Slave Interface 44
7.3 APB Slave 45
7.4 APB Data Write 45
7.5 APB Data Read 46
7.6 APB Slave (Wrapper) and Leon Interface 46
7.7 Software Memory Addressing 47
7.8 RTL of APB Register Wrapper 48
7.9 32-bit Register Synthesized RTL 49
7.10 32-bit Register Synthesized RTL APB Wrapper 49
7.11 Register Test Program 50
7.12 Test Verification 50

8.1 General Paper and Pencil Multiplication 52
8.2 4 by 4 multiplication with accumulator 52
8.3 Shift and Add Multiplication Example 53
8.4 System Level Design 54
8.5 Operation using States Counter 55
8.6 Top Level Design (RTL) 55
8.7 Synthesized Model (Xilinx) 56
8.8 Simulation of 8-bit Multiplier 57
8.9 APB Integration (Theora Hardware) 58
8.10 Multiplier Integration on AMBA APB Bus 58
8.11 mult dir 58
8.12 vhdlsyn.txt 59
8.13 Multiplier Component Leon3 Top 59
8.14 Multiplier APB Interface 60
8.15 32-bit Multiplier APB Interface 60

8.16 Address Pointer Declaration for Variables 60
8.17 Address Pointer Declaration for Variables (32-bit) 61
8.18 Example C Compiler Code of 16-bit 62
8.19 Compiler Code Verification 16-bit 63
8.20 N bit in Leon3 Top 63
8.21 in files in Multiplier Core 64
8.22 mult.in.vhd and mult.in.h 64
8.23 GUI Control config.in 65
8.24 Multiplier Configuration in xconfig 65

9.1 AHB Master 67
9.2 AHB Slave 68
9.3 AHB Decoder 68
9.4 AHB Data Write 70
9.5 AHB Data Read 70
9.6 AHB Slave and Leon Interface 71
9.7 Software Memory Interfacing 72
9.8 Register Test Program 73
9.9 Register initialization in Leon 73
9.10 Test Verification 74

10.1 AES Top Module 76
10.2 AES-128 Simulation in ModelSim 76
10.3 GRAES Registers 77
10.4 AES Wrapper (Interface) RTL 78
10.5 AES AHB Interface Synthesized Model 78
10.6 AES Pointer Variables in C 80
10.7 AES Pointer Variables in C 80
10.8 AES Pointer Variables in C 81

List of Tables

3.1 Description of Variables of 8051 Core 7

4.1 Comparison of Different 32-bit RISC Processors 19
4.2 Evaluation of Bus Architectures 19
4.3 Configuration of MP, GPP and HPP Processors 29

6.1 Coprocessor Interface vs Memory-Mapped Interface 41

7.1 APB Signals 43

8.1 4,8,16 and 32-bit Multiplier Variables and Their Latency . . . 57
8.2 Pointer Variables for Multiplier in C 61
8.3 Pointer Variables for Multiplier in C (32-bit) 62

9.1 AHB Signals 69

10.1 AES-128 Registers with Pointer Addressing 79

Chapter 1

Introduction

1.1 Background and Motivation

Since the introduction of System-on-Chips (SoCs) architecture separate chips for High Performance Hardware Accelerators were integrated into a single chip. Their ability to integrate multiple coporocessors and processor cores on a single with low power consumption and high performance made them an integral part of everyday electronics. Coprocessor integration is done using Memory-Mapped Interface or Coprocessor Interface. Usually, former is preferred for better maintainability and plug-and-play configuration.
General purpose processors are designed for general purpose tasks with sequential logic but there are certain scenarios where this may not suffice. [6] For this purpose, specialized hardware or coprocessor is designed to handle these tasks efficiently. For example, modern processors use floating-point unit as a coprocessor to handle floating point operations efficiently where Leon3 FPU is a prime example.

1.2 Problem Statement

Coprocessor designs normally concentrate on being used as a single independent unit rather than its interface with processors or bus architectures. [37]. Nowadays many powerful public domain IP cores are available for complicated component like 32 bit processor i.e. LEON3. It needs some expertise, work and experimentation to implement a hardware/software co-design project. We take an effort to present step-by-step description for implementing desired coprocessor or peripheral on LEON3 processor.

1.3 Proposed Approach

The main purpose of this thesis is to develop and extend simple RISC (Reduced Instruction Set Computer) based GPP (General Purpose Processor). In this academic project, a simple 32-bit RISC processor will be designed and tested on an FPGA. For proof of concept, a simple open source 8-bit processor will be selected to run on FPGA using custom instructions written in C. Thereafter, we shall try to change specifications and peripherals of the timer like timers, UART, SPI etc.
For 32-bit processor, starting with an open source processor design for Leon 3, the study will involve synthesis of code, compilation of program, and test of pre-configured peripherals on an FPGA. Once a decent level of understanding has been achieved, a new peripheral will be integrated into the processor to enhance the processor's capabilities, and to adapt them for better performance in a given domain of applications.
For integration Memory-Mapped Integration will be used as it preferred way of integration since the introduction of AMBA Bus Architecture. Also, Co-processor Integration will be studied so that proper distinction can be drawn between two types of techniques. For this purpose, we need benchmark tools to draw the difference.

Chapter 2

Literature Review

2.1 Introduction

A processor is a device capable of manipulating information in a way specified by sequence of instructions. This sequence of instructions (constituting an instruction set) may be altered to suit the application. A sequence of instructions is a machine controlled program. Each type of processor has a different instruction set meaning functionality of instructions varies. [38]

2.2 Instruction Set and RISC

Instruction set is processor's vocabulary for understanding instructions. Complex programs are broken down into instructions and again encoded in 1s and 0s (machine language) by the compiler. Processors read and execute these instructions [5]. There are two major approaches in instruction set architecture:

- Complex Instruction Set Architecture (CISC)
- Reduced Instruction Set Architecture (RISC)

CISC Architecture include processors like Intel x86, Motorola 68 series and National Semiconductor 32 series. RISC processors include Sun's SPARC, ARM, Microchip PIC and Atmel's AVR. Computer architecture types are devided into von Neumann and Harvard architecure. In von Neumann Architecure, memory (may be internal or external) of a processor contains instructions (with program counter) to be executed and data on which instructions are executed. Instructions are fetched (read) from the memory while data is both read and written to memory. von Neumann Architecture used

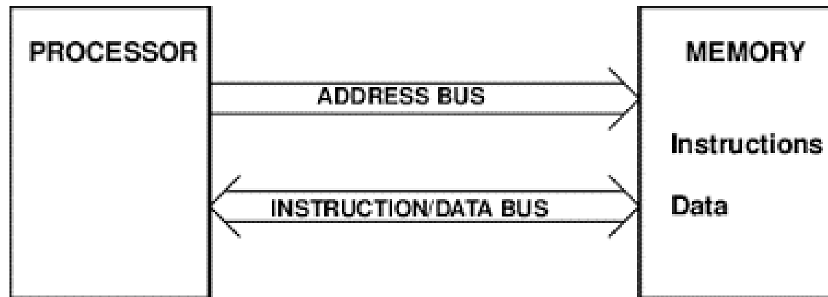

Figure 2.1: von Neumann Architecture

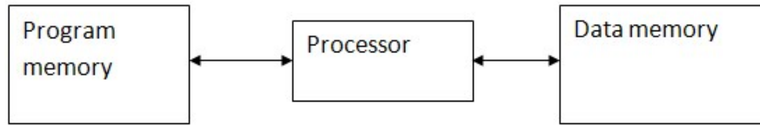

Figure 2.2: Harvard Architecture

by most CISC processors. 2.1
On the other side, in Harvard Architecture, instructions and data have different memory spaces with separate address, data and control buses. Separate memory spaces causes instructions and data fetch be executed independently. In our current study, we will stick with RISC Processor conforming Harvard Architecture. 2.2

2.3 Peripheral Integration to Processor

Coprocessors are being increasingly used for their higher throughput as compared to software based solutions. Their introduction is to add specific enhancements for variety of applications to General-Purpose Processors (GPPs). They are designed for specialized and resource intensive applications such as encryption/decryption [9], object tracking, complex signal processing, floating-point operations (Leon3 FPU), audio/video processing [24], CORDIC processor [34] etc. In SoCs, power consumption is lower as compared to separate chip for Coprocessors. For this, system-on-chips are designed with required coprocessors for reconfiguration to save power and bus architecture memory.

Chapter 3

8-bit Processor (8051 Microcontroller)

3.1 Introduction

For understanding of how a processor works and how it can be synthesized into FPGA, we chose open source that was compatible to Intel 8051 architecture [39]. There are many open source and commercial IP Core available. Open source 8051 IP Cores include Oregano Systems mc8051 [32], OpenCores' T51 and 8051 [36] while commercial IP Cores include Evatronix R8051XC2, e8051 and Digital Core Design DP8051CPU.
Of all the above mentioned 8051 cores, R8051XC2 is claimed to be fastest and fully-configurable 8051 achieving speed of 350 MHz. However, its code was not open source and meant for commercial purposes. For education, cores from Open Cores and Oregano Systems were to be used. Cores from Open Cores had one disadvantage that they were not easy to synthesize and documentation provided was not helpful. Thus, core for 8051 Microcontroller written in VHDL from Oregano Systems was chosen.

3.2 Oregano Systems mc8051

Its main features due to which it was chosen are as under:

- Open source VHDL code
- Instruction set compatible to 8051 microcontroller (Intel Architecture)
- Technology Independent (FPGA and ASIC)

- Extra Timer/counter and serial interface with addition of special function registers
- Parameterizeable via VHDL constants
- 256 bytes internal RAM
- 64 Kbytes ROM
- 64 Kbytes External RAM
- Its target IP Core was available in ARM Keil compiler for software programming

Its core can divided into:

1. Control Unit
2. ALU
3. Timer / Counter (Parameterizable)
4. Serial Interface (Parameterizable)

Control Unit is further divided into memory controller and Finite State Machine (FSM). Note that core does not contain any memory unit such as RAM or ROM to store instructions. This will be done during creation of top module in synthesis and simulation using selected target technology. Its list of variables is shown in Table 3.1 and Top Module 3.1. [33]

3.3 Tools Required for Synthesis and Simulation

1. For synthesis and simulation Xilinx ISE 14.5 was installed in Window 10 x64 bit computer and was configured for x64 XST Simulator (nt64).
2. For compilation of C Program for 8051, Keil c51 was installed which has built in target specification for Oregano 8051 Core 3.2. Here, after building C file (for example, BLINKY.c or Fibonacci.c), corresponding .hex file was created.
3. Hex to Bin converter
4. Bin to COE Converter (we will discuss it later on their purposes)

Table 3.1: Description of Variables of 8051 Core

Signal Name	Description
clk	System Clock
reset	Asynchronous reset for all Flip Flops
all_tx0_i	Timer 0 interrupt
all_tx1_i	Timer 1 interrupt
all_rxd_i	Receive data input for serial interface units
int0_i	Interrupt 0 input
int1_i	Interrupt 1 input
p0_i	Port 0 Input
p1_i	Port 1 Input
p2_i	Port 2 Input
p3_i	Port 3 Input
all_rxdwr_0	Data direction signal for bidirectional RXD input / output
all_txd_o	Transmit Data output for serial interface
all_rxd_o	Receive data output mode 0 operation for serial interface
p0_o	Port 0 output
p1_o	Port 1 output
p2_o	Port 2 output
p3_o	Port 3 output

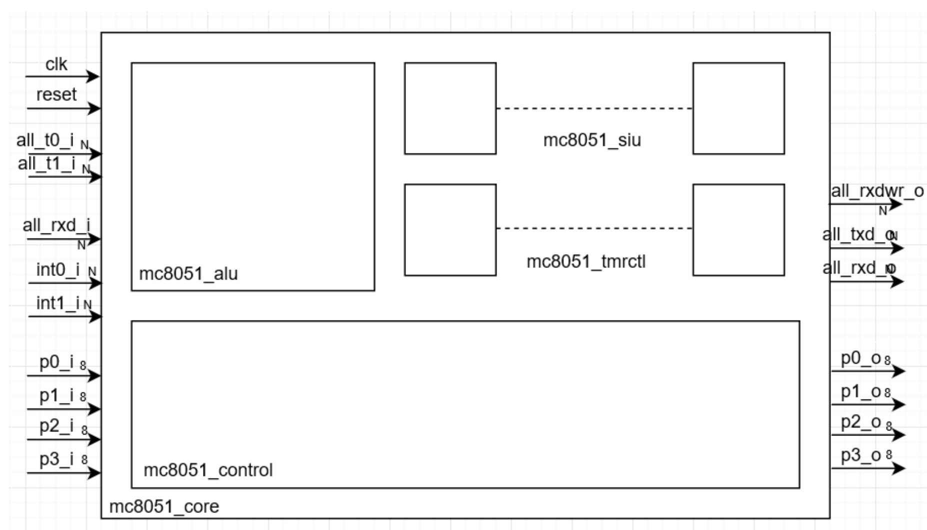

Figure 3.1: Oregano 8051 Core Top

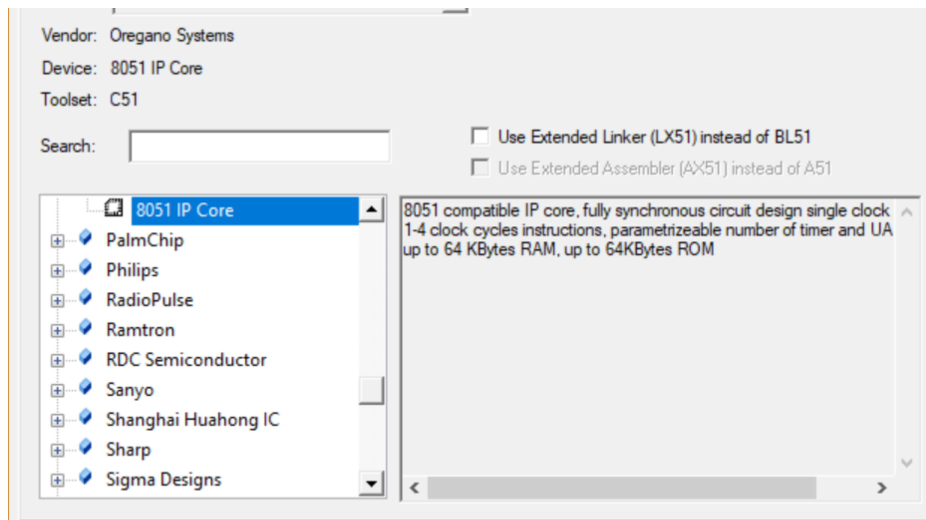

Figure 3.2: Target 8051 Microcontroller in Keil c51

3.4 mc8051 Top Module (Synthesis)

Two projects for 8051 were created in Xilinx ISE for synthesis and simulation. Spartan 3E (XC3S500E) was chosen for both simulation and synthesis. However, due to low IOBs (about 200 percent) in Spartan 3E during synthesis, we had to chose Vertex 5 (XC5VFX70T) Evaluation Board to work on. Top module for 8051 was written in VHDL, which used components of 8051 Core as well as memories such as 128 x 8 RAM 3.3, 64k x 8 ROM 3.4 and 64k x 8 External RAM 3.5. Memories were created from Core Generator in Xilinx ISE. Configuration for 128 x 8 bit RAM is as follows:

- Single Port RAM
- Minimum Area
- Read / Write Width: 8
- Write / Read Depth: 128
- Enable (ENA) Pin
- Write First
- Reset (RSTA)

Configuration for ROM is as follows:

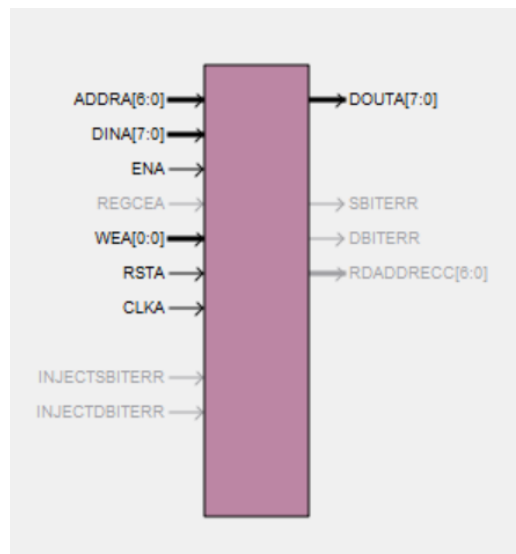

Figure 3.3: RAM Configuration in Xilinx

- Single Port ROM
- Minimum Area
- Read Width: 8
- Read Depth: 65536
- Always Enabled
- Load Init File (COE File)
- Use Reset (RSTA) pin

Configuration for XRAM (External RAM) is as follows:

- Single Port RAM
- Minimum Area
- Write / Read Width: 8
- Write / Read Depth: 65536
- Write First
- Always Enabled

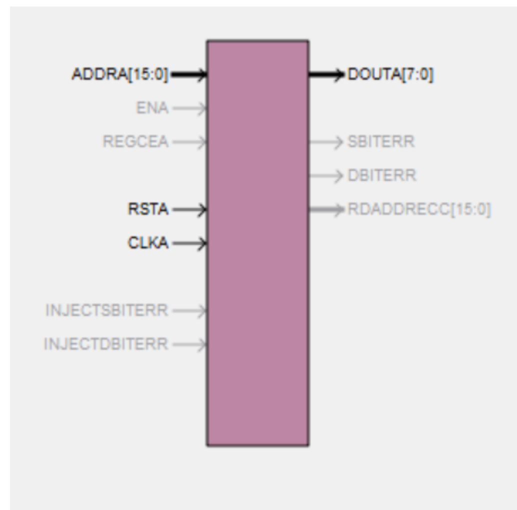

Figure 3.4: ROM Configuration in Xilinx

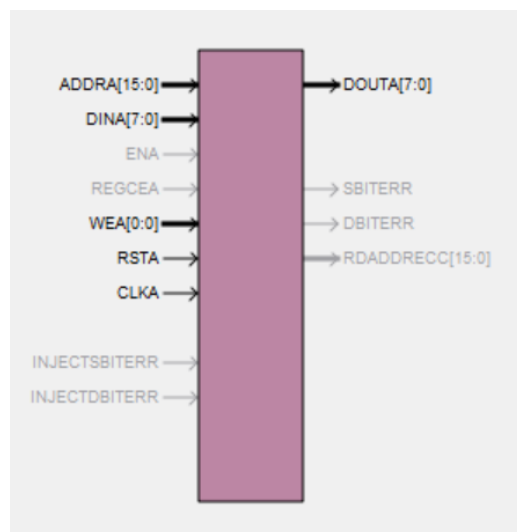

Figure 3.5: XRAM Configuration in Xilinx

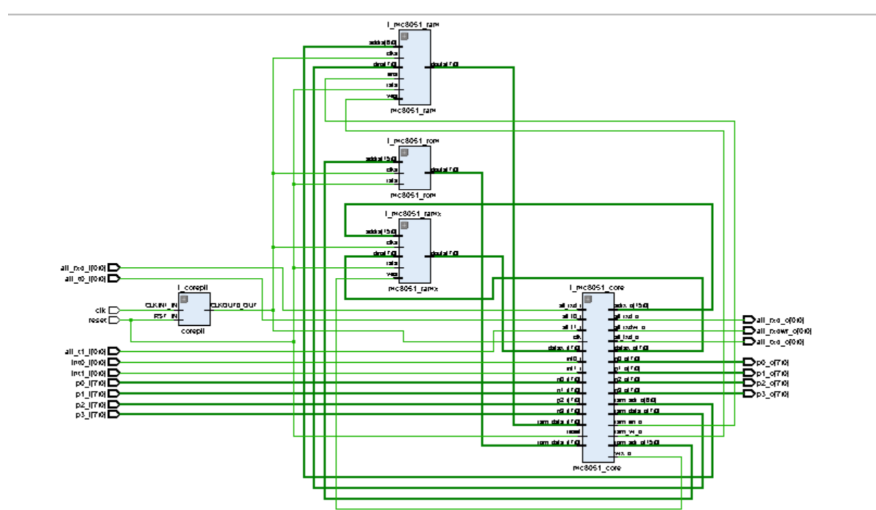

Figure 3.6: 8051 Top Module (Plan Ahead Pre-Synthesis)

- Use Reset (RSTA) Pin

Also, Phase Locked Loop (PLL) from Xilinx Core Generator was used to downgrade the speed from FPGA system clock of 100 MHz to desired frequency (11.675, 25 or 40 MHz). Its component was also called in top module. Architecture of Top Module generated from Plan Ahead (Pre-Synthesis) is shown in Figure 3.6. We can see Control Unit,ALU, Timer/Counter and Serial Interface in the system.

3.5 Work on Keil C51 (Microcontoller)

For 8051 Core to work on FPGA, we had to create HEX file from C file written for Oregano mc8051. It should be noted before compilation, the frequency of target core should be same as in PLL. The code used was for BLINKY 3.7, an example from Keil C51 after installation. After successful compilation and build, HEX file was created.

3.6 Conversion from HEX to COE

Normally, HEX file created is loaded into microcontroller ROM as instructions to execute a particular function. On FPGA, however, ROM created from Xilinx Core does not use HEX file. It rather loads Coefficient (COE) File. To convert HEX to COE file, there are some open source tools available but most are not compatible with 64 bit Windows. [10] [25] For this purpose,

```
10  // char code reserve [3] _at_ 0x23;   // when using on-chip UART for communication
11  // char code reserve [3] _at_  0x3;   // when using off-chip UART for communication
12
13  void wait (void)  {                    /* wait function */
14    ;                                    /* only to delay for LED flashes */
15  }
16
17  void main (void)  {
18    unsigned int i;                      /* Delay var */
19    unsigned char j;                     /* LED var */
20
21    while (1) {                          /* Loop forever */
22      for (j=0x01; j< 0x79; j<<=1)  {    /* Blink LED 0, 1, 2, 3, 4, 5, 6 */
23        P0 = j;                          /* Output to LED Port */
24        for (i = 0; i < 20000; i++)  {   /* Delay for 20000 Counts */
25         wait ();                        /* call wait function */
26        }
27      }
28
29      for (j=0x79; j> 0x01; j>>=1)  {    /* Blink LED 6, 5, 4, 3, 2, 1 */
30        P0 = j;                          /* Output to LED Port */
31        for (i = 0; i < 20000; i++)  {   /* Delay for 20000 Counts */
32         wait ();                        /* call wait function */
33        }
34      }
35    }
36  }
37
```

Figure 3.7: BLINKY.c in Keil c51 IDE

an alternative set to tools (Hex2bin and bin2COE) were introduced which convert HEX to bin file and then, bin to COE. These tools used and their working in Command Prompt are shown in Fig 3.8.
The resulting COE File is referenced by ROM Core before Core Synthesis 3.9. These are instructions for FPGA to perform once it is programmed into FPGA.

3.7 Synthesis and Implementation on FPGA

Before implementation, User Constraints (UCF) file was created in project. On board clock for FPGA is 100 MHz. Program loaded from HEX file running on default 12 MHz clock. Change in clock domains caused wrong results in LEDs shown as P0 of 8051 Core.
To deal with this problem, a PLL Core was introduced in between FPGA Clock and 8051 Core Clock. The resultant clock was matched for PLL and HEX file at: 11.675 MHz. Synthesized core is shown Fig 3.10. Once all problems were catered, programming file was generated and loaded into FPGA and was working smoothly. We tried with different clock speeds to check Timing and Power Utilization of Synthesis Process. 40MHz was highest clock speed possible achieved by 8051 Core. Comparison for 25 MHz and 40 MHz using different design strategies is given in Fig 3.11.

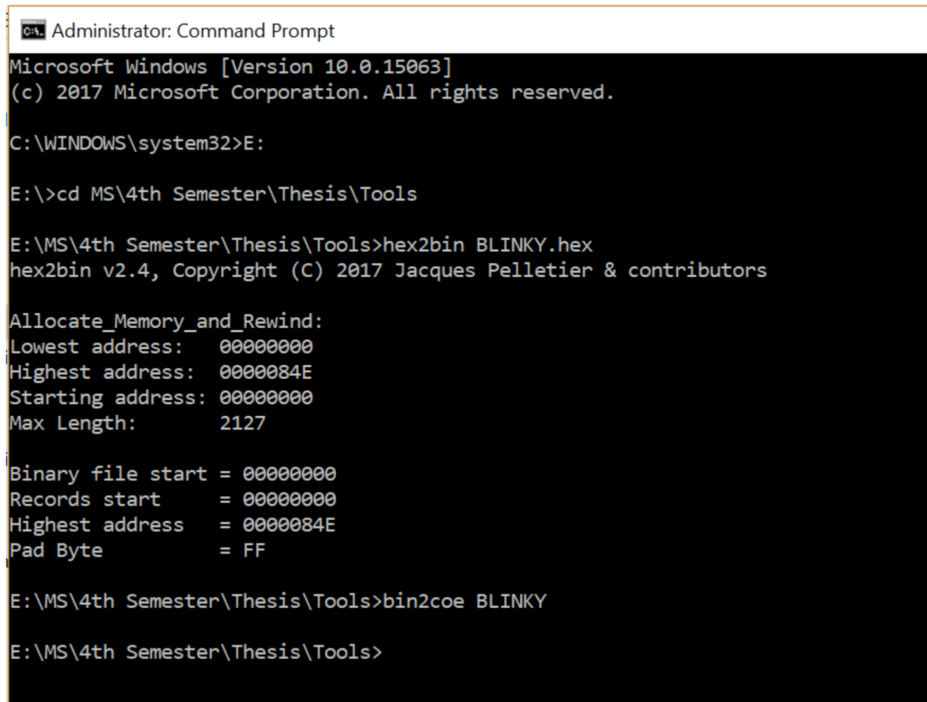

Figure 3.8: HEX to COE Conversion in Command Prompt

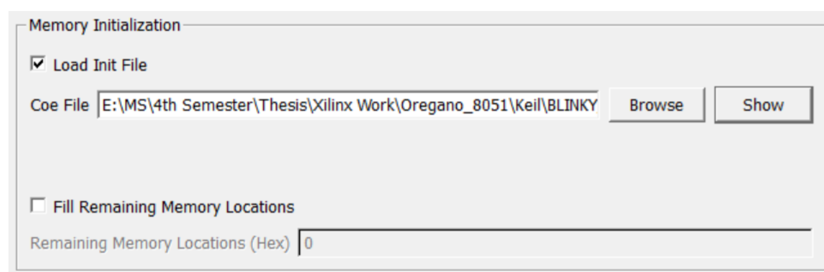

Figure 3.9: COE file load in Xilinx Generated ROM Core

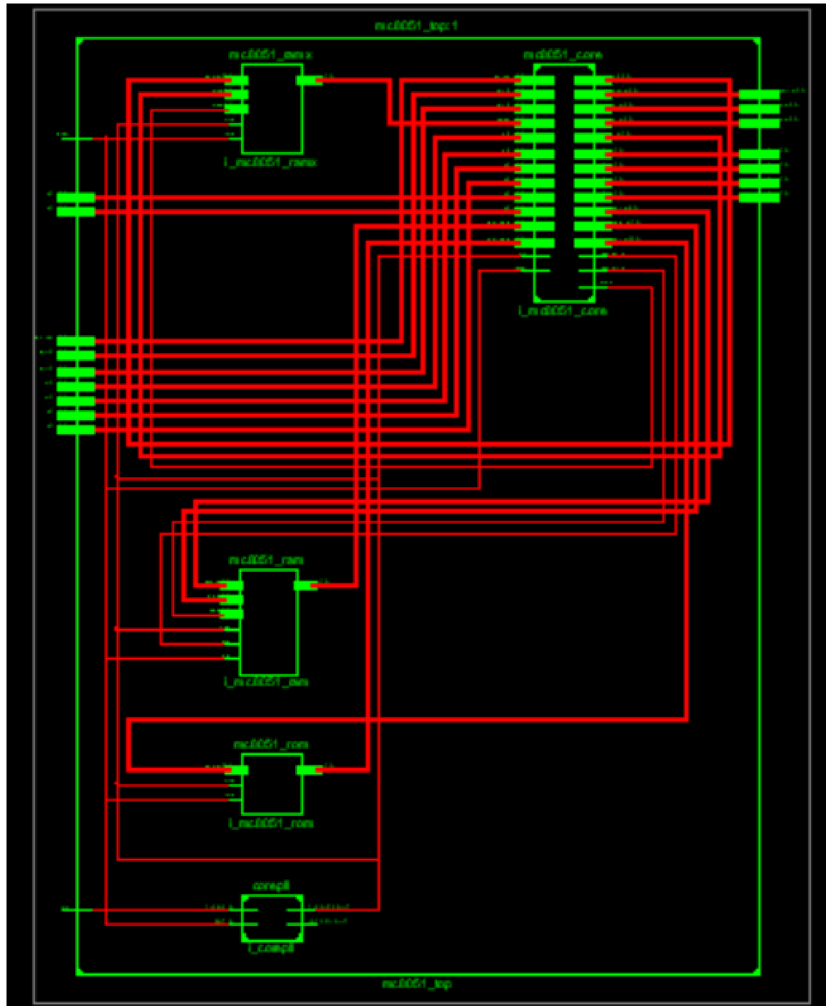

Figure 3.10: 8051 Core RTL Schematic in Xilinx

	11.765 MHz			25 MHz			40 MHz		
	Timing Performance	Balanced	Power	Timing Performance	Balanced	Power	Timing Performance	Balanced	Power
Synthesis Max Freq (MHz)	47.082	46.777	33.248	47.082	46.777	33.248	47.082	46.777	33.248
P&R Derived Freq (MHz)	20.15	26.348	18.504	25.043	26.51	25.061	36.085~	40.107	31.144~
Chip Power (mW) 30C	1189.44	1173	1163.49				1205.27	1208.67	1200.14
Chip Power (mW) 50C	1562.98	1546.34	1536.73	1577.78	1565.45	1555.63	1578.94	1582.42	1573.78

Figure 3.11: Comparison on different design strategies

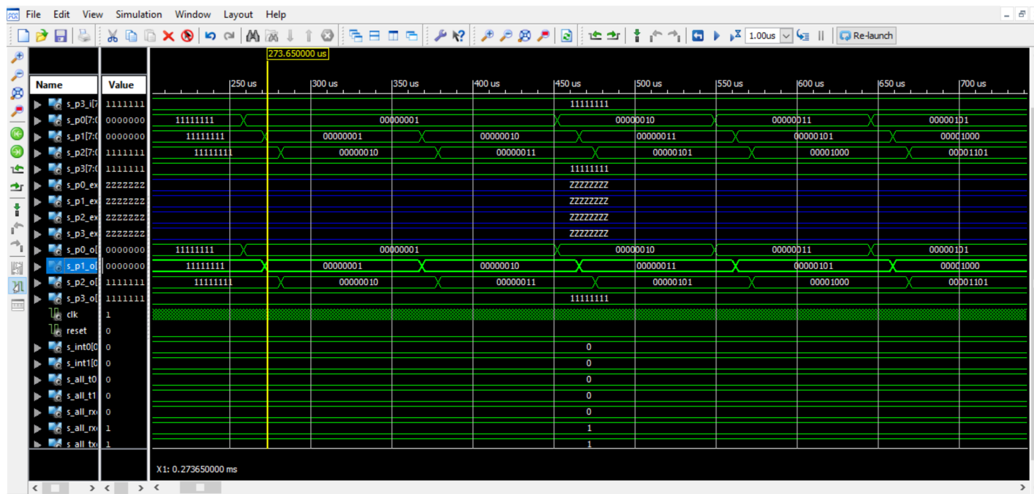

Figure 3.12: Fibonacci Code simulation on 8051 Core

3.8 Simulation

A local testbench was created for Fibonacci.c file which was loaded into a ROM similar to synthesis process. It was then simulated using Xilinx ISim. The output integer values were used in Port 0 (p0_o) 3.12.

3.9 Configuration of mc8051 for extra peripherals

The original microcontroller design offered only 2 timers, one serial and 2 external interrupt units. These can be changed in VHDL Core using some constants to increase or decrease the said peripherals. However, to decode registers of added peripherals (if any) without changing the address space of 8051 only two 8 bit registers are inferred as additional special function registers (SFRs). [33] These are TSEL (address 0x8Eh for timer/counter units) and SSEL (address 0x9Ah for serial interface units). If these registers point to a non existent device number, the default unit number 1 is selected. Efforts were made to be able to infer SFRs in Keil. REG51.H is referenced by C File in Keil. SFRs inferred is shown below:
As an example, 25 MHz synthesizable core was chosen. In this core, file named mc8051_p.vhd there is parameter named 'C_IMPL_N_TMR'. It can take values from 1 to 256. Its default value to set to 1. We changed its value to 2 which generated 2 extra timer units, 1 additional serial port and 1 additional external interrupt sources. Initial and custom (C_IMPL_N_TMR

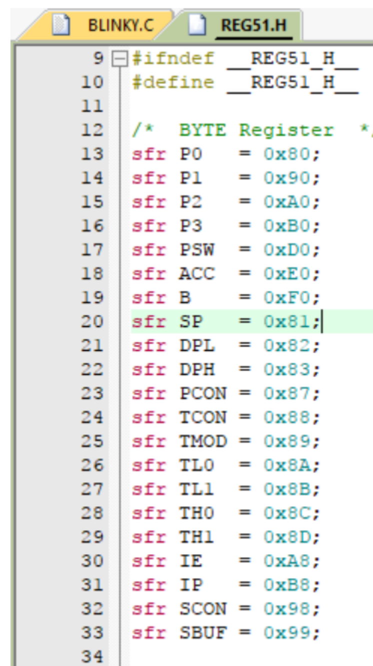

Figure 3.13: REG51 Special Function Registers

= 2 has 82 I/Os) peripheral diagram (pre-synthesized) is shown in Fig 3.14 and Fig 3.15 respectively:

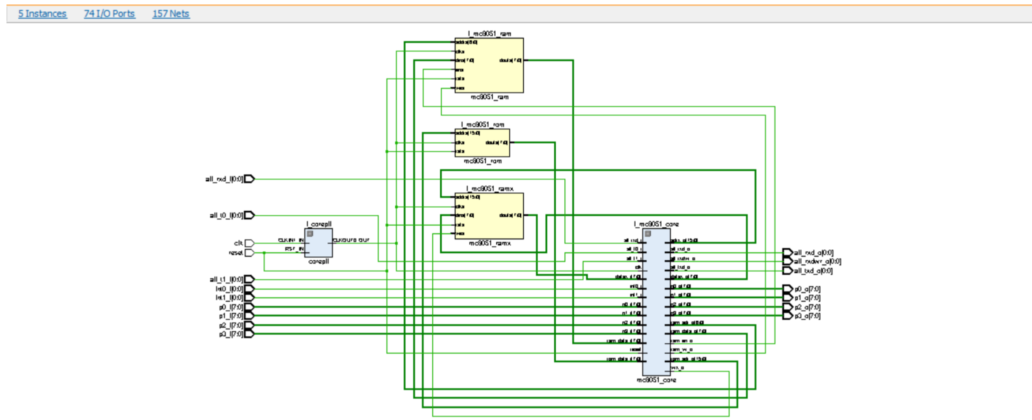

Figure 3.14: Default I/Os:74 and C_IMPL_N_TMR=1

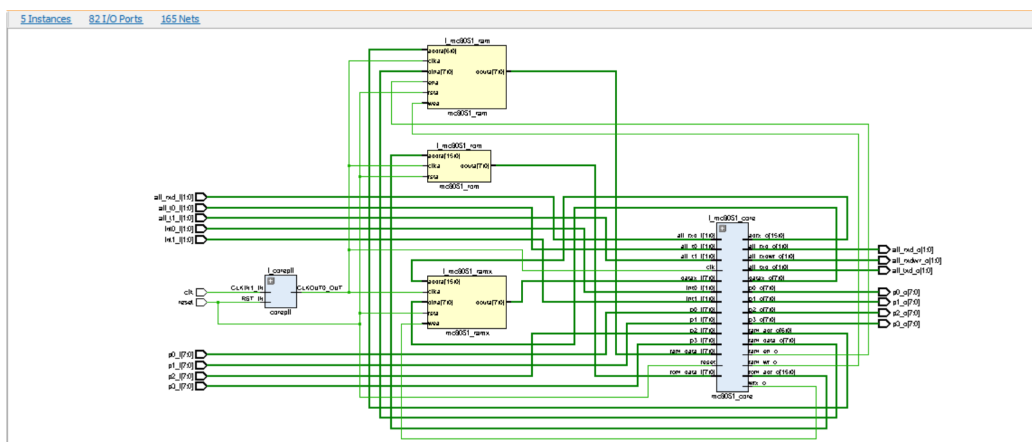

Figure 3.15: Default I/Os:82 and C_IMPL_N_TMR=2

Chapter 4

Leon3 Introduction

4.1 Introduction

The complexity of designing processors has increased overtime. Designing each and every hardware component of the system from scratch soon became impractical and expensive for most designers. Therefore, the idea of using pre-designed and pre-tested IP Cores in designs became an attractive alternative. Softcore processors are processors whose architecture and behavior are fully described using synthesizable Hardware Desciption Languages (HDL) like Verilog or VHDL. They can be easily synthesized to FPGA or ASIC. [22]Use of these processors has advantages like:

- Customizable
- Technology Independent
- Easily understandable

We will look for different open source and commercial IP Cores like in 8051 to come up with the best one for 32-bit RISC Processor which can be easily customized to our needs.

4.2 Evaluation of Processors(SoC)

There are many 32-bit processors available such as Altera Nios II, Xilinx MicroBlaze, Tensilica Xtensa, OpenCores OpenRISC 1200 and Gaisler Leon 3. Overall comparison has been drawn between them in Table 4.1. [23] [30] From above table 4.1, we can easily access that each processor has its advantages and disadvantages. Xtensa offers unlimited ISA customization but

Table 4.1: Comparison of Different 32-bit RISC Processors

Category	Nios II	MicroBlaze	Xtensa	Leon3
Max Frequency (MHz)	200 (FPGA)	200 (FPGA)	350 (ASIC)	400 (ASIC)
Cache	Upto 64 KB	Upto 64 KB	Upto 32 KB	Upto 256 KB
Pipeline Stages	6	3	5	7
Custom Instructions	Upto 256	None	Unlimited	None
Implementation	FPGA	FPGA	FPGA, ASIC	FPGA, ASIC
Open Source	No	No	No	Yes

Table 4.2: Evaluation of Bus Architectures

Feature	WishBone	AMBA	Avalon	CoreConnect
Open Architecture	Yes	Yes	Partial	Yes
Hierarchical	No	Yes	No	Yes
Pipelined	No	Yes	Yes	Yes
Arbitration	Yes	Yes	Yes	No
Data Transfer Hand Shaking	Yes	Yes	No	Yes
Data Transfer Pipelined	No	Yes	Yes	Yes
Split Transfer	N/A	Yes	No	Yes
Clocking	Yes	Yes	Yes	Yes
Frequency	User Defined	User Defined	User Defined	User Defined

it is also not open source and expensive. Similarly, OpenRISC has open source code but difficult to use to use with given technology. Leon 3, despite its ISA customization it excels all other departments. However, there are other problems to be explored also like bus architecture, software tools and compliant ISA.

4.3 Evaluation of Bus Architectures

There are namely four different bus architectures:

1. WishBone (OpenCores)
2. AMBA (ARM)
3. Avalon (Altera)
4. CoreConnect (IBM)

Their comparison is drawn below:

From here also, we can see that AMBA from ARM has quite a lot of advantages 4.2. However, WishBone has an edge of being adopted as primary bus for most open source designs. AMBA is the bus architecture used by Leon 3. We will check more details about it afterwards.

4.4 SPARC Version 8 ISA

If you choose a custom ISA, we have to create everything yourself:

- the chip architecture
- compiler
- OS and Appication Programmable Interfaces(APIs)
- cross-compilation

SPARC is an instruction set architecture (ISA), derived from a RISC lineage. As an architecture, SPARC allows for a spectrum of chip and system implemenetations at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial. SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations. SPARC implementations provide exceptionally high execution rates and short time-to-market development schedules. Its advantages are: [18]

- Open architecture without patent or license fees unlike Intel, MIPS and ARM
- Well designed
- Well documented
- Easy to implement
- Established software standard

4.5 Leon3 Introduction and Pipeline

The LEON3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC v8 architecture. The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. The full source code is available, allowing free and unlimited use for research and education. The LEON3 processor has the following features: [15]

- Compliant with SPARC V8 ISA
- 7-stage Pipeline
- Hardware Multiply, Divide and MAC units
- Floating Point Unit (FPU)
- Harvard Architecture (Separate Instruction and Data Cache)
- AMBA 2.0 AHB Bus Interface
- On-Chip Debug Support
- Multiprocessor Support
- Power Down and Clock Gating
- Fault tolerant version available for High Performance space applications
- Extensively configurable
- Tools available like simulators, compilers, debuggers and kernels

Leon 3 consists of following subsystems: [12]

1. Integer Unit (based on 7-Stage Pipeline Harvard Architecture) 4.1
2. Cache (Data and Instruction)
3. Floating Point Unit Coprocessor
4. Hardware Multiplier and Divider
5. Memory Management Unit
6. Debug Support Unit
7. Interrupt Controller

Integer Unit which is based on Harvard Architecture, implements the full SPARC V8 standard, including hardware multiply and divide instructions. The implementation is focused on high performance and low complexity. Register windows are set to 8 as default but are configurable as per SPARC standard (2-32). Integer Unit pipeline consists 7-stages which separate execution of data and instruction cache interface. Its 7-stage pipeline is shown in Fig 4.2. These can be summarized as:

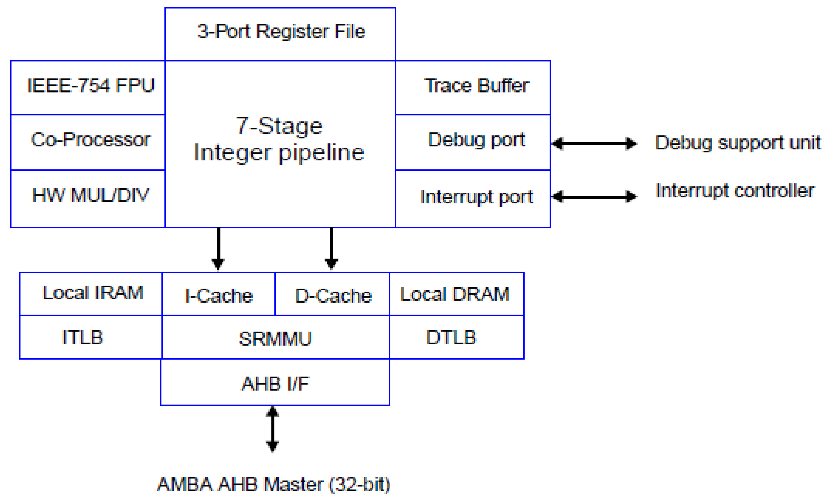

Figure 4.1: Leon3 Integer Unit

- FE (Instruction Fetch): If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU (Integer Unit).
- DE (Decode): The instruction is decoded and the CALL and Branch target addresses are generated.
- RA (Register access): Operands are read from the register file or from internal data bypasses.
- EX (Execute): ALU (Arithmetic Logic Unit), logical, and shift operations are performed. For memory operations (e.g. LD) and for JMPL/RETT, the address is generated.
- ME (Memory): Data cache is accessed. Store data read out in the execution stage is written to the data cache at this time.
- XC (Exception) Traps and interrupts are resolved. For cache reads, the data is aligned as appropriate.
- WR (Write): The result of any ALU, logical, shift, or cache operations are written back to the register file.

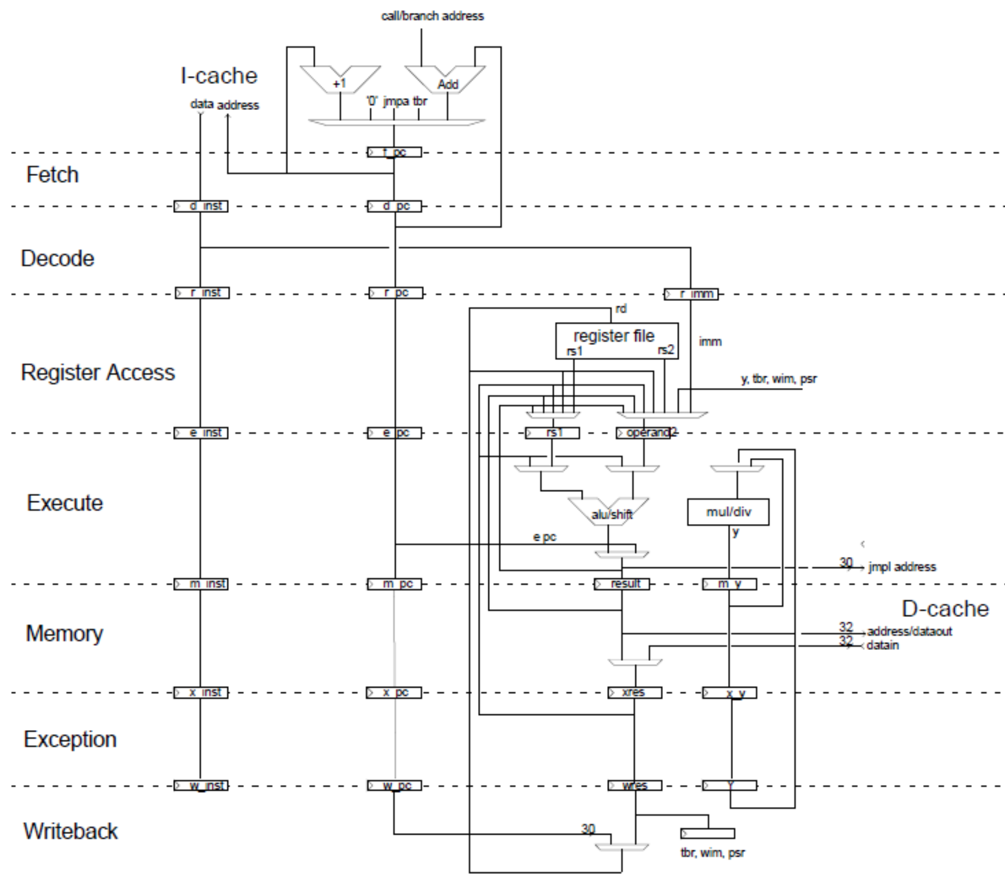

Figure 4.2: Leon3 7 Stage Pipeline

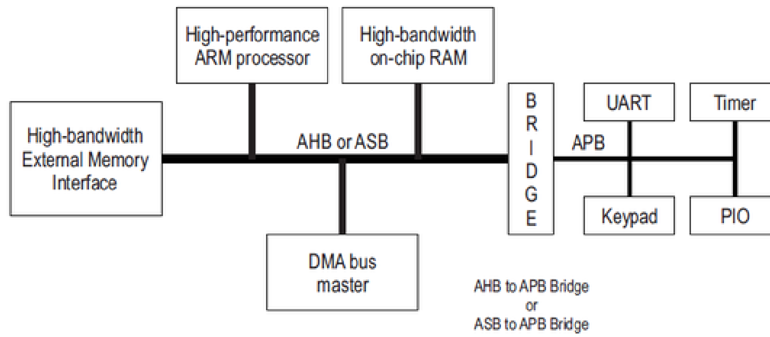

Figure 4.3: AMBA Shared Single Bus

4.6 AMBA Bus Architecture

Bus architecture is based on Advanced Microcontroller Bus Architecture (AMBA) introduced by ARM for RISC based processors. Its specification is used in the design of high performance processor SoC architectures. The typical AMBA bus system is shown in the figure 4.3, here there are two bus systems, one requiring high performance for the high speed components like, the internal memory, Direct Memory Access (DMA) and processor. On the other hand, peripherals, coprocessors or cores that do not need such high bandwidth are connected through to the low power bus via High-to-Low performance bridge. The former is called AHB (Advanced High Performance Bus) while latter is called APB (Advanced Peripheral Bus). They are discussed in detail in Chapters 7 9.

4.7 Example Template Design

Leon3 SoC architecture is based on AMBA Advanced High-Speed bus (AHB) as its bus architecture. All the components, memory and coprocessors including Leon3 is conncted to this bus. External memory is accessed through a combined PROM/IO/SRAM/SDRAM memory controller. Default template design of SoC includes peripherals like Ethernet, Serial and JTAG debug interface, UART, Interrupt Controller, CAN 2.0 and General Purpose I/O Ports. The design is be highly configurable as desired by use. Leon3 SoC is shown in Fig 4.4. [12]

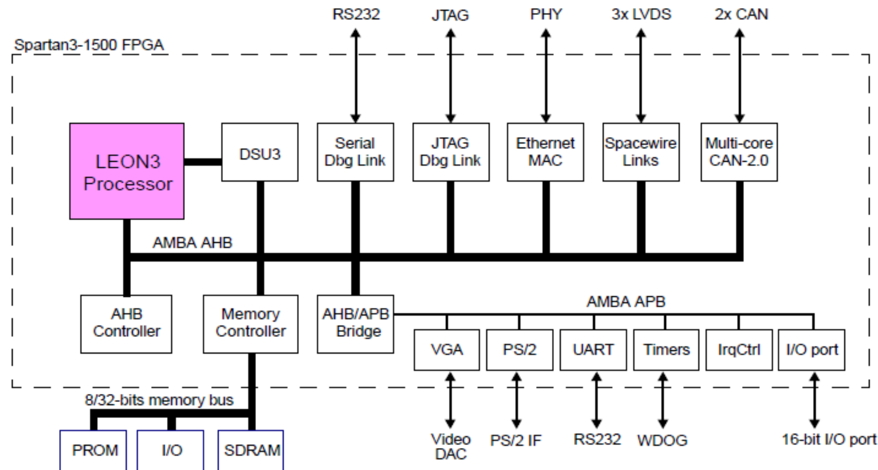

Figure 4.4: Leon 3 in Spartan 3E Template

4.7.1 Library (Source Code) and Toolchain

The complete design environment for LEON3 including all the IP cores can be downloaded from its website. Leon 3 design is integrated with template designs and other IP Cores in a single library file known as GRLIB. It is distributed as a zipped file and can be installed in any location on the host system. This library includes:

1. Make Files and script generators for shell commands (like bash or Cygwin)
2. Target FPGA Board designs from different companies like Altera, Xilinx etc
3. IP Cores including Leon 3
4. Example software files
5. FPGA and ASIC Technologies
6. Example template designs for Configuration and Synthesis

After installation of library, toolchain is required to use Leon 3. It is compatible in both Windows and Linux. However, Windows is preferred due to ease in installation. [22] It includes:

- Bare-C Compiler (BCC)

- Boot-Prom Builder (mkprom2)
- RTEMS Leon Cross Compiler (RTEMS)
- GRMON Debug Tool (GRMON2 Evaluation version)
- TSIM Simulator (Evaluation Version)

In windows environment, these tools are installed through a single installer file known as GRTOOLS where in Linux every file has to be installed separately. Also, during installation, environment variables in windows are set automatically. For Bare-C Compiler [13], Eclipse Kepler version 1.6 is installed during installation. Besides ease at installation, we preferred Windows because tools for Synthesis and Simulation (Xilinx and ModelSim) were already installed and their environment variables were set. For Linux, all these tools had to be installed from scratch.
However, for shell commands, Cygwin for windows was installed [17]. It emulates Linux Terminal in Windows. Cygwin has a major disadvantage that it is unable to launch ModelSim GUI. Also, it should be note to simulate the design using ModelSim, its professional edition should be installed. Student edition is not supported by Leon toolchain. Detailed work with each tool discussed above will be presented afterwards.

4.7.2 Example Template Configuration and Implementation

Leon3 system is usually implemented using example template designs included in design directory. We implement and try to LEON3 template design for the Xilinx ML50x (ML507) board 4.5 which was also used when we were using 8051. Implementation is done in five steps:

- Configuration of Leon design in xconfig
- Simulation of design
- Synthesis and Place Route
- Generate Bitstream
- Configure FPGA on board

Template design is based on mainly three files found in ML50x folder:

- config.vhd - a VHDL package containing design configuration parameters. Its is created and modified when using xconfig GUI tool.

Figure 4.5: Xilinx Vertex-5 ML507

```
talal@Talal-Laptop ~
$ cd /cygdrive/e/Work/Leon3/designs/leon3-xilinx-ml50x

talal@Talal-Laptop /cygdrive/e/Work/Leon3/designs/leon3-xilinx-ml50x
$ export DISPLAY=:0.0
```

Figure 4.6: Export Display to XWin Server

- leon3mp.vhd - top module of Leon3 SoC with instances of all components including Leon3 processor. It uses config.vhd to instantiate and use IP cores.

- testbench.vhd - testbench to simulate the desired SoC Architecture

In windows, we install Cygwin to replicate the Linux environment in Windows. During installation, make sure to install Tcl/Tk which is important for GUI launch. With cygwin installed, it is time to configure Leon using xconfig tool. Cygwin can be launched from Desktop and also XWIN server is required also for display. After XWin is successfully launched, following command is written in Cygwin to export Display to XWin server 4.6 Here we can see that we are in target design ML50x. Here by writing xconfig in cygwin shell calls for xconfig GUI as shown in Fig 4.7. Leon3 Configuration is broken down into:

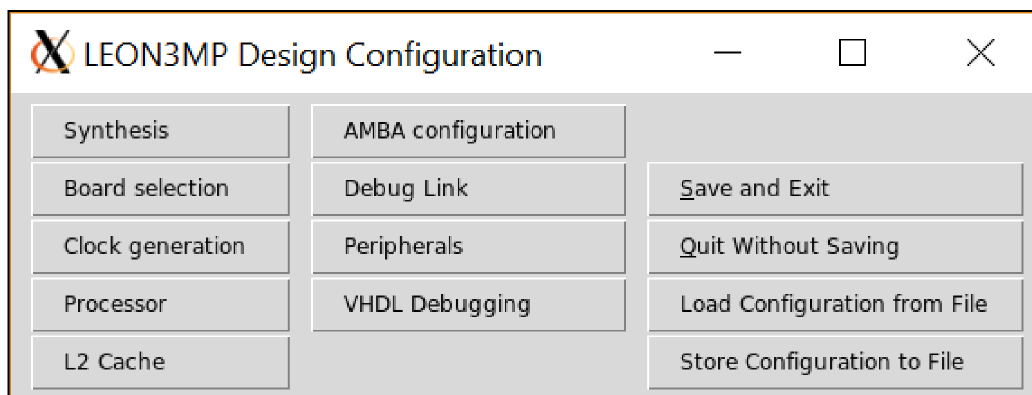

Figure 4.7: Leon3 Design Configuration GUI(xconfig)

- Synthesis: Target technology for FPGA and other technology related configurations. In this case, it is Xilinx.
- Board Selection: FPGA Board (Xilinx ML507) or ASIC Technology
- Clock Generation: PLL Generated for FPGA Board. Default is 60 MHz for 100 MHz Board.
- Processor: Main Processor configuration like number of processors, Integer Unit, FPU, MMU Configuration
- L2 Cache
- AMBA Bus Configuration
- Debug Link
- Peripherals: Memory Controller, On-Chip RAM/ROM, Ethernet, UART, Timer, VGA and Keyboard Interface, PCI Express
- VHDL Debugging

This default configuration is known as Minimal Processor. First we will try to simulate and synthesize the Minimal Processor and then, go for more high performance configurations.

4.7.3 Configuration for Minimal, General Purpose and High Performance Processor

Following table 4.3 describes the VHDL Generics to changed for Minimal (MP), General Purpose (GPP) and High Performance Processor (HPP) and

Table 4.3: Configuration of MP, GPP and HPP Processors

VHDL Generic	MP	GPP	HP	Description
dsu	0	1	1	Debug Support Unit
fpu	0	1	1	Floating Point Unit
v8	0	2	16#32#	Support for SPARC v8 MUL/DIV
nwp	0	2	4	Hardware Watchpoints
icen/ dcen	1	1	1	Processor Caches
irepl / drepl	2	2	2	Random replacement policy
dnsoop	0	6	6	Data Cache Snooping
mmuen	0	1	1	Memory Management Unit
tbuf	0	4	4	Trace Buffer
pwd	1	2	2	Power Down Mode
smp	0	0	1	SMP Support
bp	0	1	1	Branch Prediction
tlb_type	1	2	2	Look-a-side TLB Buffers
1ddel	1	1	1	1-cycle load delay
itlbnum / dtlbnum	-	8	16	MMU look-a side buffers

	Minimal	General Purpose	High Performance
Slice Registers (%)	18	22	23
Slice LUTs (%)	31	39	39
LUT-FF Pairs (%)	41	40	41
Bonded IOBs (%)	47	47	47
Block RAM/FIFO (%)	13	14	14
BUFG (%)	50	50	50
DCM_ADV (%)	50	50	50
DSP_48 (%)	0	0	0
Synth Frequency (MHz)	98.586	85.981	90.853

Figure 4.8: Processor Comparison on Area Utilized and Timing

then, synthesized on FPGA. These generics (or global variables) are updated in config.vhd file. [14] Area utilized on Vertex-5 ML507 and timing analysis for each processor in Fig 4.8.

4.8 Software Development (BCC)

To simulate (or emulate) the desired system or hardware performance for Leon3, we use embedded C program. It is also a good test of the software development environment. We use Bare-C Compiler (BCC) installed in the system. [13] BCC is a cross-compiler for LEON3 processors. It is based one the GNU compiler tools and the Newlib standalone C-library. The cross-

compiler system allows compilation of both tasking and non-tasking C and C++ applications. It supports hard and soft floating-point operations, as well as SPARC V8 multiply and divide instructions. We use example program 'Hello World'. BCC takes hello.c file and compiles it to output hello.exe. This executable file can be loaded into FPGA program using two methods:

- GRMON Debugger
- MKPROM2 PROM Programmer

4.8.1 Software Development (GRMON Debugger)

GRMON debug monitor uses debug interface to control the loading and running of compiled C program. It can also be launch using Windows Command Prompt. [7] GRMON has the follwing features:

- Read/Write all Registers and Memory
- Built-in disassembler and trace buffer management
- Loading and execution of GP applications
- Modern IDE Tools Management
- Remote connection to GNU debugger (GDB) (e.g. TSIM)

We use JTAG link which is also used for bit file programming of FPGA. However, for GRMON, compatible driver must be installed to use it. After successful link is established in JTAG, GRMON shell is launched in Command Prompt.
hello.exe compiled with Bare-C Compiler can be loaded into Leon 3 FPGA using GRMON debug link and its output is wrote back in the GRMON shell.

4.8.2 Software Development (PROM Programmer)

This method is used to program PROM of Leon 3 as boot loader before it is synthesized on FPGA same way as ROM was loaded with coefficient file in 8051. [11] Here, we use mkprom to output PROM file with loaded hello.exe file. First, it compiles and create PROM.out file. After successful PROM.out, file is loaded in PROM.srec bootloader file of Leon 3 which can be run as ModelSim testbench 4.9.

```
# LEON3 MP Demonstration design
# GRLIB Version 1.5.0, build 4164
# Target technology: inferred  , memory library: inferred
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Cobham Gaisler          LEON3 SPARC V8 Processor
# ahbctrl: mst1: Cobham Gaisler          AHB Debug UART
# ahbctrl: slv0: European Space Agency   LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Cobham Gaisler          AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Cobham Gaisler          Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Cobham Gaisler          Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Cobham Gaisler          Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Cobham Gaisler          AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Cobham Gaisler          General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 3
# gptimer3: Timer Unit rev 1, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 4, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# ahbuart7: AHB Debug UART rev 0
# leon3_0: LEON3 SPARC V8 processor rev 3: iuft: 0, fpft: 0, cacheft: 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
#  moving .text from 0x00001620 to 0x40000000
#  moving .data from 0x00007000 to 0x400059e0
# Hello World
#
```

Figure 4.9: PROM hello.exe loaded and run

```
tsim> load E:\Work\Leon3\designs\leon3-xilinx-ml50x\hello.exe
section: .text, addr: 0x40000000, size 42864 bytes
section: .data, addr: 0x4000a770, size 2960 bytes
read 450 symbols
tsim> run
starting at 0x40000000
Hello World

Program exited normally.
```

Figure 4.10: TSIM running hello.exe on Leon3 environment

4.8.3 Software Development (TSIM Simulator)

TSIM is a unique Leon 3 simulator which emulate its environment without the use of FPGA. ERC32 or LEON applications can be loaded and simulated using a Windows Command Prompt. A number of commands are available to examine data, insert breakpoints and advance simulation. [1]
Leon 3 can be loaded like GRMON System Information. This information can be changed to custom needs to emulate the required environment. hello.exe and any other program can be compiled using Bare-C Compiler again can be loaded and run 4.10.

Chapter 5

Leon3 Extension and Customization

5.1 Introduction

Using the knowledge of Leon 3 processor, we need to extend our work in customizing this processor. We will to study the factors and variables essential in the designing of this processor. There is different form of understanding required to achieve each form of customization [27].To add a peripheral or Co-Processor, we need:

- Library Structure
- Understanding and working of AMBA bus
- VHDL Generics and link with Leon3mp.vhd (Top Module)
- xconfig GUI Customization

5.2 Library Structure

Scripts generated search for VHDL libraries in libraries files source or lib/lib.txt. These contain paths to directories of IP Cores and Leon3 compiled as VHDL library. Their mapping is always as appear in compile order in libs.txt. 5.1 [12].
Each directory specified in the libs.txt contains the file dirs.txt, which contains paths to sub-directories containing the actual VHDL code. The sub-directories contains compile order of VHDL files to be synthesized or simulated in order or preference. 5.2.

```
talal@Talal-Laptop /cygdrive/e/Work/Leon3-ext/lib
$ cat libs.txt
#tech/dware
synplify
techmap
spw
eth
opencores
ihp
actel/core1553bbc
actel/core1553brt
actel/core1553brm
actel/corePCIF
gr1553
gaisler
esa
#nasa
fmf
spansion
gsi
```

Figure 5.1: Library showing scripts for different vendors

```
talal@Talal-Laptop /cygdrive/e/Work/Leon3-ext/lib/grlib/amba
$ cat vhdlsyn.txt
amba.vhd
devices.vhd
defmst.vhd
apbctrl.vhd
apbctrlx.vhd
apbctrldp.vhd
apbctrlsp.vhd
apb3ctrl.vhd
ahbctrl.vhd
ahbxb.vhd
dma2ahb_pkg.vhd
dma2ahb.vhd
ahbmst.vhd
ambaprot.vhd
```

Figure 5.2: Library showing scripts for different files in AMBA folder

```
talal@Talal-Laptop /cygdrive/e/Work/Leon3-Ext/lib/grlib/amba
$ ls
ahbctrl.vhd  amba.in    amba.in.help  amba.vhd     apbctrl.vhd
ahbmst.vhd   amba.in.h  amba.in.vhd   amba_tp.vhd  apbctrldp.vhd

talal@Talal-Laptop /cygdrive/e/Work/Leon3-Ext/lib/grlib/amba
$ cat vhdlsyn.txt
amba.vhd
devices.vhd
defmst.vhd
apbctrl.vhd
apbctrlx.vhd
apbctrldp.vhd
apbctrlsp.vhd
apb3ctrl.vhd
ahbctrl.vhd
ahbxb.vhd
dma2ahb_pkg.vhd
dma2ahb.vhd
ahbmst.vhd
ambaprot.vhd
example_ip_ahb.vhd

example_ip_apb.vhd
```

Figure 5.3: New files in AMBA folder

Why is this important? When scripts are generated during synthesis or simulation, the library is loaded with each file required for the processor and assembly system. When we create or add new peripheral, we update these scripts accordingly. It is done by updating target vhdlsyn.txt file with new peripheral file 5.3.

5.3 Understanding and Working of AMBA Bus

Detail understanding and working of AMBA bus and addition of Peripherals and Co-Processors to AHB and APB Bus is discussed it detail in Chapters 9 and 7 respectively.

5.4 VHDL Generics and Link with Top Module

VHDL Generics are global variables used as parameters saved in config.in. It creates a new variable which is used in config.vhd as parameter to generate component in leon3mp.vhd. To understand this, we first look config.in where it loads variables from different libraries containing .in files.

```
---------------------------------------------------------------------
---  System ACE I/F Controller ---------------------------------------
---------------------------------------------------------------------

 grace: if CFG_GRACECTRL = 1 generate
   grace0 : gracectrl generic map (hindex => 4, hirq => 3,
       haddr => 16#002#, hmask => 16#fff#, split => CFG_SPLIT)
     port map (rstn, clkm, clkace, ahbsi, ahbso(4), acei, aceo);
 end generate;
 nograce: if CFG_GRACECTRL /= 1 generate
   aceo <= gracectrl_none;
 end generate;

 sysace_mpa_pads : outpadv generic map (width => 7, tech => padtech)
   port map (sysace_mpa, aceo.addr);
 sysace_mpce_pad : outpad generic map (tech => padtech)
   port map (sysace_mpce, aceo.cen);
 sysace_d_pads : iopadv generic map (tech => padtech, width => 16)
   port map (sysace_d, aceo.do, aceo.doen, acei.di);
 sysace_mpoe_pad : outpad generic map (tech => padtech)
   port map (sysace_mpoe, aceo.oen);
 sysace_mpwe_pad : outpad generic map (tech => padtech)
   port map (sysace_mpwe, aceo.wen);
 sysace_mpirq_pad : inpad generic map (tech => padtech)
   port map (sysace_mpirq, acei.irq);
```

Figure 5.4: Generation of components in Top Module

```
talal@Talal-Laptop /cygdrive/e/Work/Leon3-ext/lib/gaisler/misc
$ ls apb_example.in*
apb_example.in  apb_example.in.h  apb_example.in.help  apb_example.in.vhd
```

Figure 5.5: apb_example.in files

5.5 xconfig extension

This module is the last but it uses information of all previous work which leads to customization of GUI shown. Each core has VHDL generics and header constants which are used in generation and configuration of its xconfig menu entries. As an example we will look at the apb_example. In figure 5.6 first line creates a boolean value for the variable CONFIG_I2CAHB which can be modified in GUI. If it is set to yes ('y') then the user can select select two more configuration options. One is width defined as integer and second is mask defined by hexadecimal value.

```
    bool 'Enable I2C to AHB bridge '         CONFIG_I2C2AHB
        if [ "$CONFIG_I2C2AHB" = "y" ]; then

    bool 'Enable APB interface '             CONFIG_I2C2AHB_APB
    hex 'AHB protection address (high) '     CONFIG_I2C2AHB_ADDRH 0000
    hex 'AHB protection address (low) '      CONFIG_I2C2AHB_ADDRL 0000
    hex 'AHB protection mask (high) '        CONFIG_I2C2AHB_MASKH 0000
    hex 'AHB protection mask (low) '         CONFIG_I2C2AHB_MASKL 0000
    bool 'Enable after reset '               CONFIG_I2C2AHB_APB
    hex 'I2C memory address '                CONFIG_I2C2AHB_SADDR 50
    hex 'I2C configuration address '         CONFIG_I2C2AHB_CADDR 51
fi
```

Figure 5.6: apb_example.in

```
CONFIG_I2C2AHB
    Say Y here to enable I2C2AHB

CONFIG_I2C2AHB_APB
    Say Y here to configure the core's APB interface

CONFIG_I2C2AHB_ADDRH
    Defines address bits 31:16 of the core's AHB protection area

CONFIG_I2C2AHB_ADDRL
    ...

CONFIG_I2C2AHB_MASKH
    ...

CONFIG_I2C2AHB_MASKL
    ...

CONFIG_I2C2AHB_SADDR
    ...

CONFIG_I2C2AHB_CADDR
    ...
```

Figure 5.7: apb_example.in.help

GUI also provides the help option for user assistance. The contents of the help box is defined in the file *.in.help. 5.7 apb_example.in.h and apb_example.in.vhd are used generation of VHDL generics as constants in config.vhd file for a design. config.vhd consists of options linked with core in sub menu entries and its integration with main SoC. After configuration is finished in GUI and xconfig is closed, variables the .in.vhd files for all cores are concatenated into one file. The contents of apb_example.in.h is: The menu entries to include in xconfig is defined for each template design in the file config.in. As an example we will look at the config.in file for the design leon3-xilinx-ml50x. In config.in we find the entry for the apb_example port (described in the previous section) as part of one of the submenus: These variables can be used to generate cores for apb_example in the same way as shown in Fig. The modified xconfig is shown below:

```
#ifndef CONFIG_I2C2AHB
#define CONFIG_I2C2AHB 0
#endif
#ifndef CONFIG_I2C2AHB_APB
#define CONFIG_I2C2AHB_APB 0
#endif
#ifndef CONFIG_I2C2AHB_ADDRH
#define CONFIG_I2C2AHB_ADDRH 0
#endif
#ifndef CONFIG_I2C2AHB_ADDRL
#define CONFIG_I2C2AHB_ADDRL 0
#endif
#ifndef CONFIG_I2C2AHB_MASKH
#define CONFIG_I2C2AHB_MASKH 0
#endif
#ifndef CONFIG_I2C2AHB_MASKL
#define CONFIG_I2C2AHB_MASKL 0
#endif
#ifndef CONFIG_I2C2AHB_RESEN
#define CONFIG_I2C2AHB_RESEN 0
#endif
#ifndef CONFIG_I2C2AHB_SADDR
#define CONFIG_I2C2AHB_SADDR 50
#endif
#ifndef CONFIG_I2C2AHB_CADDR
#define CONFIG_I2C2AHB_CADDR 51
#endif
#ifndef CONFIG_I2C2AHB_FILTER
#define CONFIG_I2C2AHB_FILTER 2
#endif
```

Figure 5.8: apb_example.in.h

```
mainmenu_option next_comment
comment 'Ethernet                  '
  source lib/gaisler/greth/greth.in
endmenu

mainmenu_option next_comment
comment 'UART, timer, I2C, SysMon, I/O port and interrupt controller'
  source lib/gaisler/uart/uart1.in
  source lib/gaisler/irqmp/irqmp.in
  source lib/gaisler/misc/gptimer.in
  source lib/gaisler/misc/grgpio.in
  source lib/gaisler/i2c/i2c.in
  source lib/gaisler/misc/grsysmon.in
  source lib/gaisler/misc/apb_example.in
endmenu
```

Figure 5.9: apb_example.in included in config.in (Folder:ML50x)

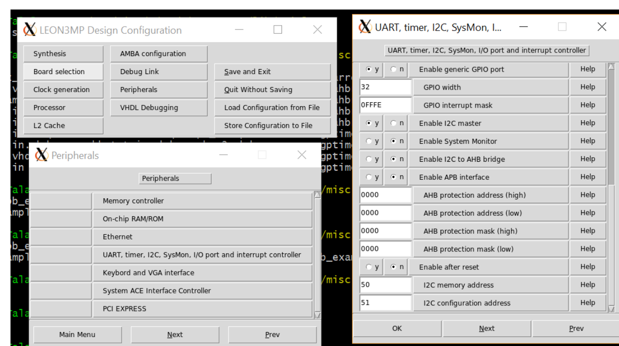

Figure 5.10: Modified xconfig

Chapter 6

Peripheral Interface (Introduction)

6.1 Introduction

For integration of custom peripheral or Coprocessor, hardware/software interface is used to enable communication between them. The software runs on a Leon3 which uses available resources on Processor and AMBA, while peripheral is linked to Leon3 through Memory Mapped Interface or Co-Processor Interface which shall be discussed shortly. In the former, local bus architecture (AMBA) is used for communication, interface and synchronization between Leon3, Peripheral and Shared Memory (if any) 6.1. A coprocessor interface u. The peripheral or coprocessor module is controlled with wrapper or interface using specialized software (with access of local registers on wrapper) on Leon3 6.2. [31]

6.2 Memory-Mapped Interface

A memory-mapped interface infer memory address (as shared memory) of Leon3 for interface between peripheral and software. It is generally more reliable and easy-to-use interface to be used for added hardware or peripheral. [19] In software, for addressing of shared memory pointers are declared. Its main advantages are:

- It is more general and easy-to-use.
- It's design cannot be locked to particular processor i.e. it can be used with any processor that supports AMBA.

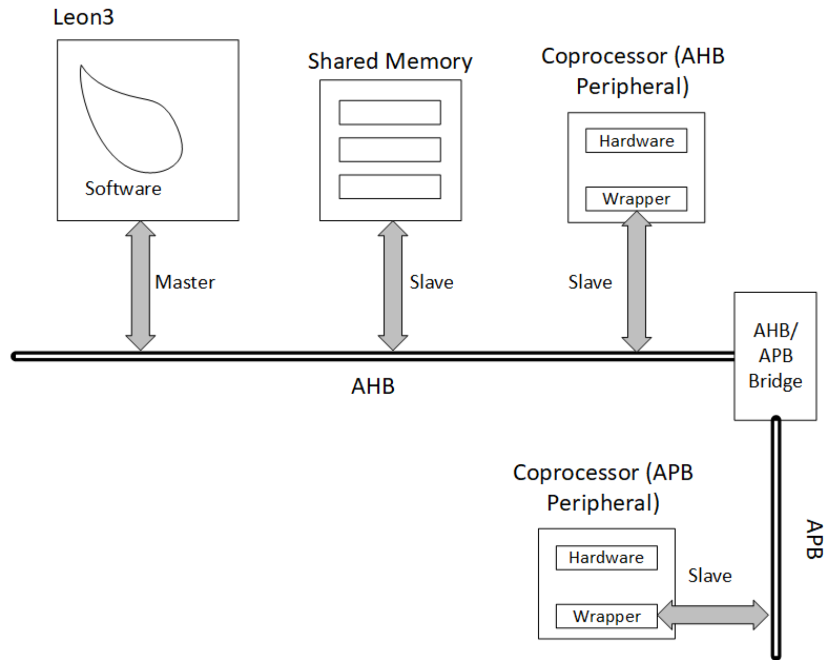

Figure 6.1: Memory-Mapped Interface (AHB and APB)

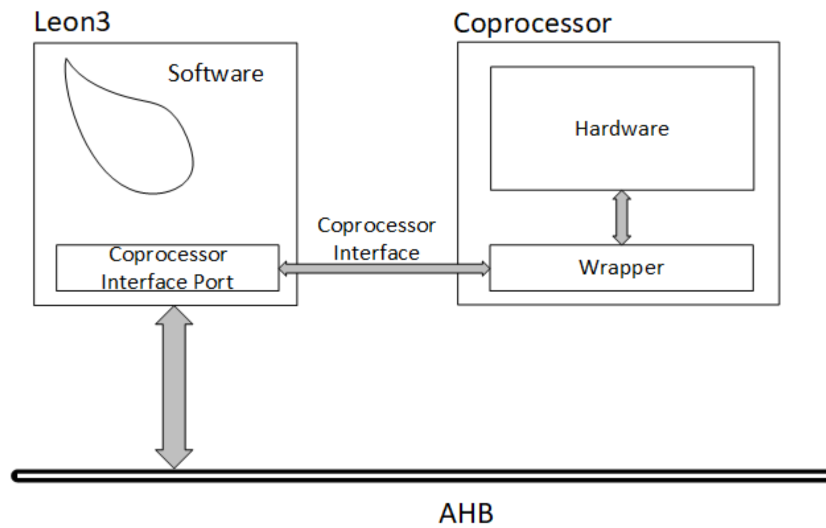

Figure 6.2: Coprocessor Interface

Table 6.1: Coprocessor Interface vs Memory-Mapped Interface

Factor	Coprocessor Interface	Memory-Mapped Interface
Addressing	Processor Specific	Bus Address
Connection	Point-to-Point	Shared
Latency	Fixed	Variable
Throughput	Higher	Lower

- Direct addressing of software to shared memory creates reliable software design.

6.3 Coprocessor Interface

In cases where high-throughput between the software and the custom hardware is needed, it makes sense to have a dedicated interface between Leon3 and peripheral. As illustrated in Fig 6.2 Coprocessor Interface uses a dedicated port on the processor which uses special instructions sometimes embedded in the processor pipeline. The coprocessor instruction set is different for each type of processor, since it depends on the processor. Its main advantages are:

- It has higher throughput
- It has fixed latency

Sadly not all processors have coprocessor interface. As an example, it was provided with Leon2 documentation but was removed from the Leon3 release, making the development of coprocessor much more difficult. There are only a few examples of coprocessor interface cores with Leon3. A classic example of a coprocessor is a floating-point calculation unit, which is interfaced with Leon3 Integer Unit pipeline. Brief difference between them is shown in table 6.1. [31]

Chapter 7

Memory-Mapped Interface (AMBA APB)

7.1 Introduction

The Advanced Peripheral Bus (APB) is part of the AMBA hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity. APB provides a low-power extension to the system bus which builds on AHB signals directly. [8]
The Advanced High Performance Bus (AHB) is a high speed bus suitable to connect units with high data rate. But, the problem is that IP Core (or Co-Processor) will be a Master on AHB bus and could overload the bus and lower the performance of LEON3. APB is slower than AHB but has following advantages:

1. Low Complexity
2. Low Power
3. Do not disturb the communication between Leon3 and Memory Controller

7.2 Advanced Peripheral Bus (APB) Architecture

APB bus is interface with AHB by AHB/APB bridge which works as AHB Slave. The AHB/APB bridge is the only APB master on one specific APB bus. More than one APB bus can be connected to one AHB bus, by means

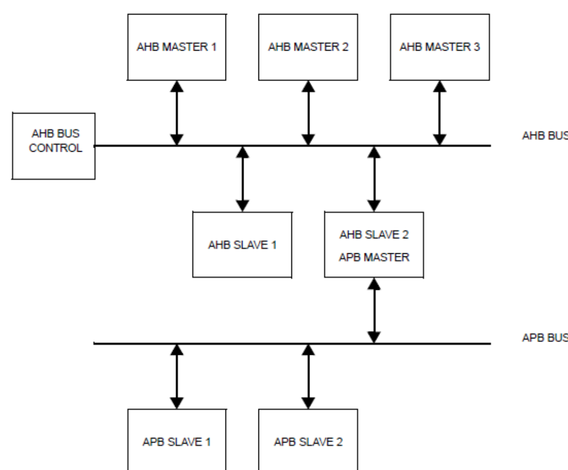

Figure 7.1: AHB and APB Bus Control

Table 7.1: APB Signals

Sr #	Name	Description
1	PCLK	Bus clock
2	PRESETn	APB reset
3	PADDR[31:0]	APB Address Bus
4	PSELx	APB Select
5	PENABLE	APB Strobe
6	PWRITE	APB Transfer Function
7	PRDATA [31:0]	APB Read Data bus
8	PWDATA [31:0]	APB Write Data bus

of multiple AHB/APB bridges. It is shown in figure 7.1. The access to the AHB slave input (AHBSI) is decoded and an access is made on APB bus. The APB master drives a set of signals grouped into a VHDL record called APBI which is sent to all APB slaves. The combined address decoder and bus multiplexer controls which slave is currently selected ('PINDEX' in case of APBI). The output record (APBO) of the active APB slave is selected by the bus multiplexer and forwarded to AHB slave output (AHBSO). [12] 7.2

7.3 IP Core (Co-Processor) APB Interface

Signals used APB interface (APBI and APBO) [3] are shown in table 7.1. APB interface can be as simple as a register which can be read and written

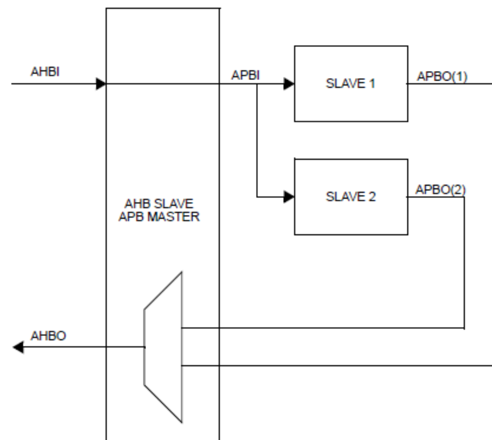

Figure 7.2: AHB to APB Master Slave Interface

through bus transfers on an on-chip bus. The register will be accessed when a given address address('PADDR'), or an address within a given range, appears on the bus. The memory address, and the related bus command, is analyzed by an address decoder [31]. APB address bus ('PADDR') works as a shared resource between software and hardware 7.3.
It works in three states which are [3]:

1. IDLE: Default State
2. SETUP: When a transfer is required the bus moves into the SETUP state, where the appropriate select signal PSELx, where Peripheral or Co-Processor is chosen, is asserted. It remains in this state for one clock cycle and move to the ENABLE state on the next rising edge of the PCLK.
3. ENABLE: In the ENABLE state the enable signal, PENABLE is asserted. The address, write and select signals all remain stable during the transition from the SETUP to ENABLE state. The ENABLE state also only lasts for a single clock cycle and after this state the bus will return to the IDLE state if no further transfers are required.

Timing diagram for write transfer is given 7.4.
Timing diagram for read transfer is given 7.5.
APB slaves have a simple, yet flexible, interface. It allows interface to be designed as per IP Core or Co-Processor requirements 7.6. For a write transfer the data can be latched at the following points:

- on either rising edge of PCLK, when PSEL is HIGH

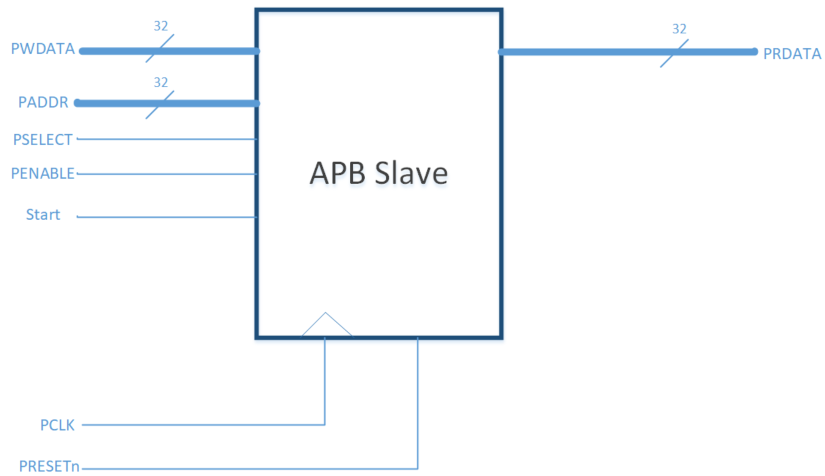

Figure 7.3: APB Slave

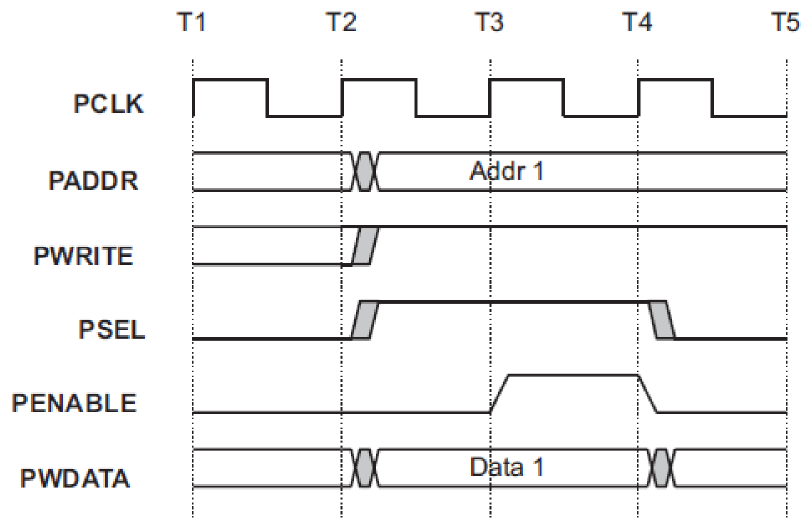

Figure 7.4: APB Data Write

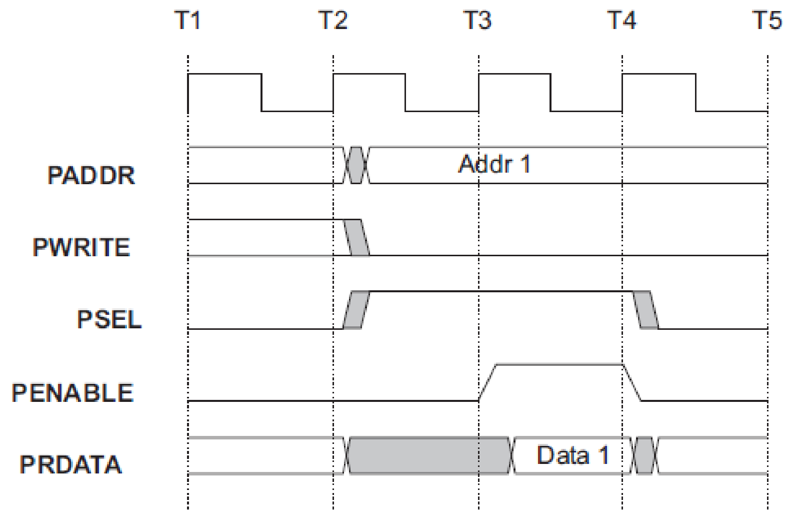

Figure 7.5: APB Data Read

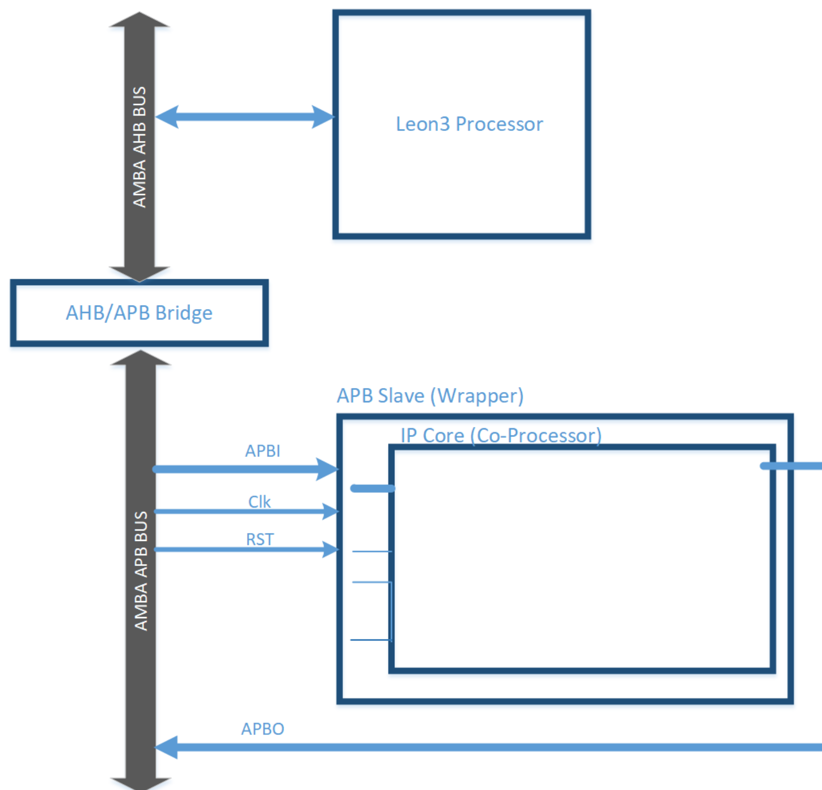

Figure 7.6: APB Slave (Wrapper) and Leon Interface

```
Int *MMRegister = (int*) 0x8000000;  //Base Address of Co-Processor Wrapper
// write the value '0xFF' into the register
*MMRegister = 0xFF;
// read the register
int value = *MMRegister;
```

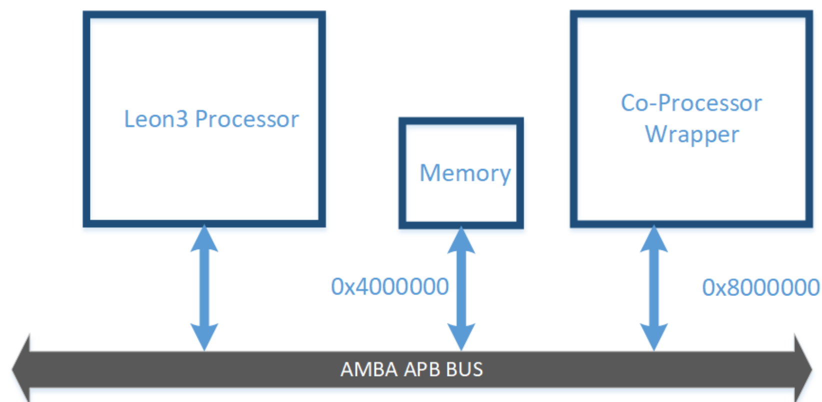

Figure 7.7: Software Memory Addressing

- on the rising edge of PENABLE, when PSEL is HIGH.

The select signal PSELx, the address PADDR and the write signal PWRITE can be combined to determine which register should be updated by the write operation. For read transfers the data can be driven on to the data bus when PWRITE is LOW and both PSELx and PENABLE are HIGH while PADDR is used to determine which register should be read.

7.4 Software Interface

In software, the representation of a register is easy to do using an initialized pointer. The base address of this pointer is determined by Slave bus index of the APB Peripheral or Co-Processor. For example, bus index for Slave (PINDEX=8) will be 0x80000800 with 256-bytes memory. Following diagram 7.7 gives us example how to architect software interface code.

7.5 Register Example

The IP core has one memory mapped 32-bit register that will be reset to zero. The register can be read or written from default PADDR. The core's bus index, base address and mask settings are configurable via VHDL generics

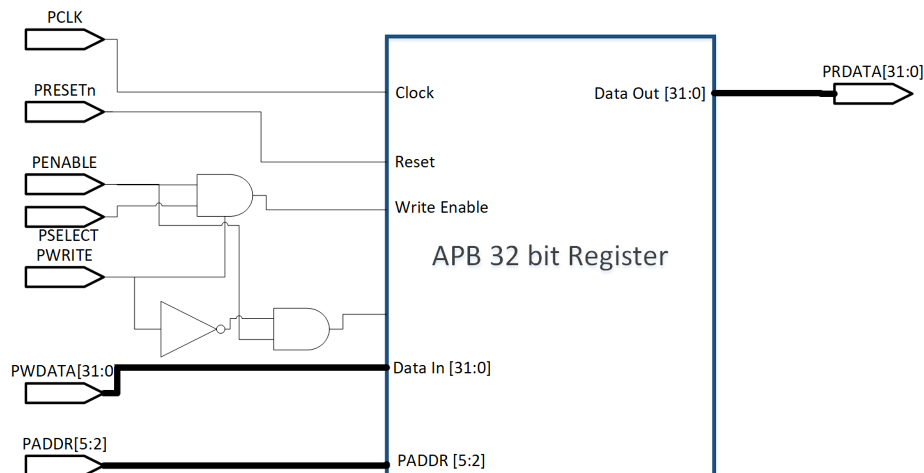

Figure 7.8: RTL of APB Register Wrapper

(PINDEX, PADDR, PMASK). The PADDR and PMASK VHDL generics are propagated to the APB bridge via the APBO.PCONFIG signal and the index is propagated via the APBO.PINDEX signal. These values are then used by the APB bridge to generate the APB address decode and slave select logic [11]. It is shown in Fig 7.8.
Synthesized RTL of Register is shown in Fig 7.9.
Synthesized RTL of Register Wrapper is shown 7.10.
Its software interface defined in figure 7.11 can be written as:
For Hardware / Software Verification, we use MKPROM which simulates testbench in ModelSim and runs compiled programs. It is a utility program which converts a LEON RAM application image into a bootable ROM image. The resulting bootable ROM image contains system initialization code, an application loader and the RAM application itself. [11]
Its main advantage is that we can simulate and verify our IP Core before implementing it on FPGA. Result of compiled code is shown in figure 7.12.

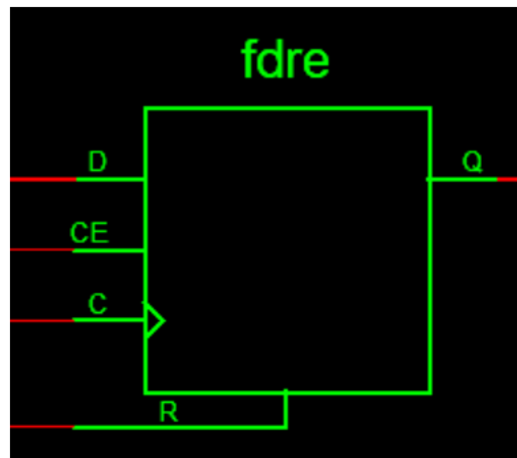

Figure 7.9: 32-bit Register Synthesized RTL

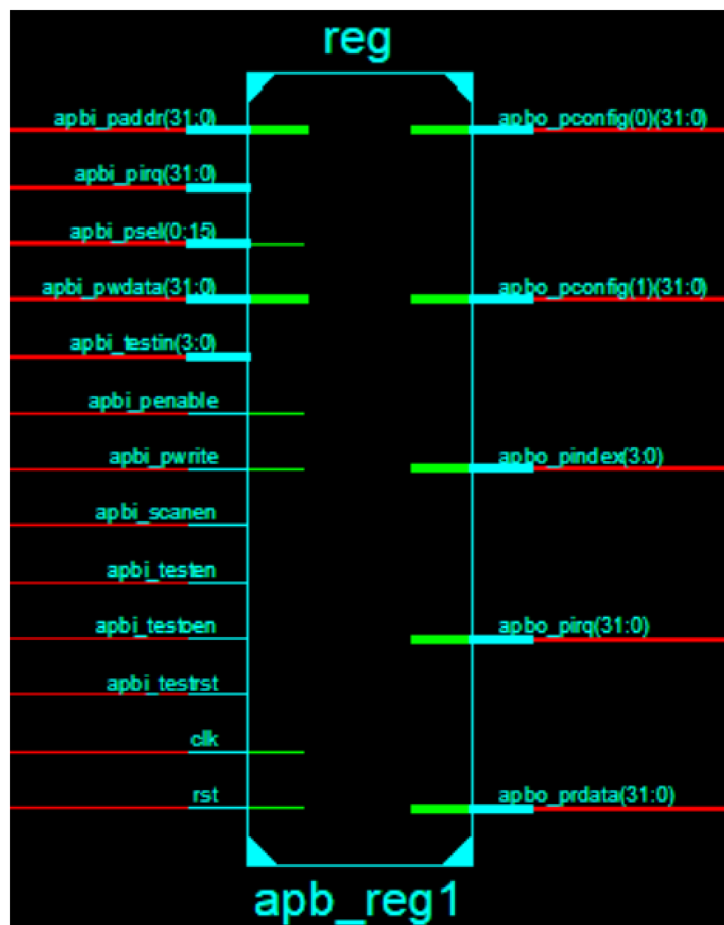

Figure 7.10: 32-bit Register Synthesized RTL APB Wrapper

```
#include <stdio.h>
main()
{
    int *baseaddr_p = (int *)0x80000800;

    printf("Register Test\n\r");

    // Write multiplier inputs to register 0
    *(baseaddr_p+0) = 0x00020003;
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+0));

    //*(baseaddr_p+1) = 0x00067611;
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+0));
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+1));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+2));
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+2));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+4));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+8));

    printf("End of test\n\n\r");
}
```

Figure 7.11: Register Test Program

```
# Register Test
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0x00000000
#
#
# Wrote: 0x80000808
#
#
# Wrote: 0x00000000
#
#
# Wrote: 0x80000810
#
#
# Wrote: 0x80000820
#
#
# End of test
#
```

Figure 7.12: Test Verification

Chapter 8

Multiplier and its APB Integration

8.1 Introduction

Multiplier is the main arithmetic unit of a processor. When we form the product A * B, the first operand (A) is called the multiplicand, and the second operand (B) is called the multiplier. As illustrated here, binary multiplication requires only shifting and adding. In the following example, we multiply 13 (1101 - 4 bit) by 11 (1011 – 4 bit) to give output of 143 (10001111 – 8 bit). 8.1

8.2 Shift and Add Multiplier

Shift-and-Add Multiplier forms the simplest multiplier (paper and pencil multiplication) to multiply two numbers. This method adds the multiplicand A to itself B times, where B denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.
Considering figure 8.2, partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. Instead of forming all the partial products first and then adding, each new partial product is added in as soon as it is formed, which eliminates the need for adding more than two binary numbers at a time. [21]
Multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, a 4-bit full adder, and an 8-bit register for the

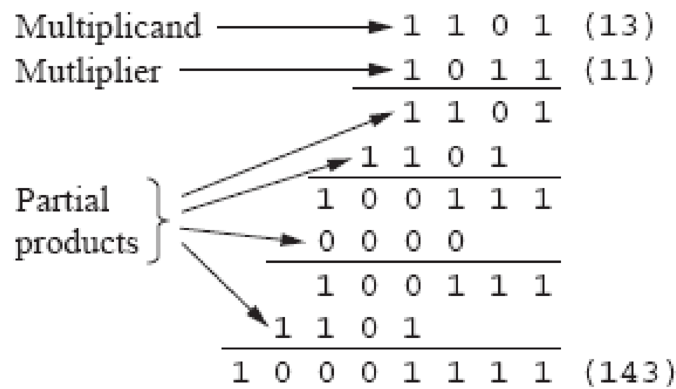

Figure 8.1: General Paper and Pencil Multiplication

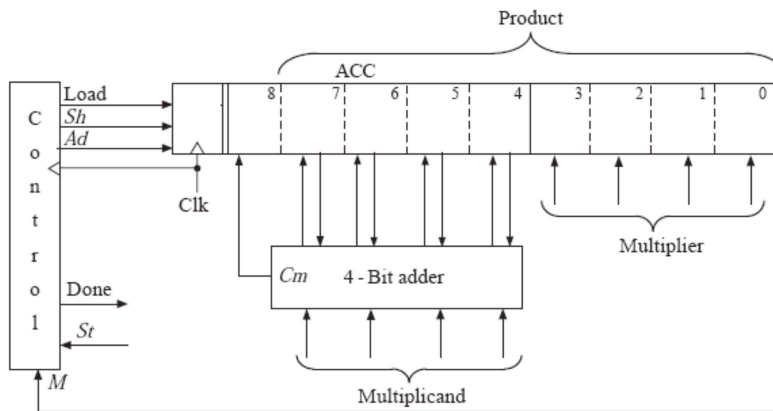

Figure 8.2: 4 by 4 multiplication with accumulator

product. The product register serves as an accumulator to accumulate the sum of the partial products.
This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs. The 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (Ad) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal (Sh) occurs, all 9 bits of ACC are shifted right by the next clock pulse.

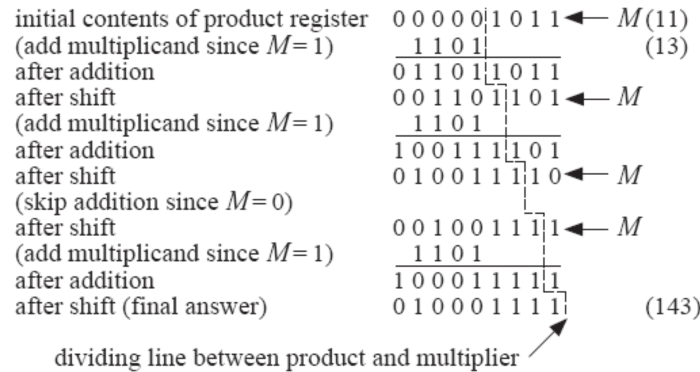

Figure 8.3: Shift and Add Multiplication Example

8.3

8.3 System Design and Behavioral Model with N Parametrization

Multiplier System is composed of: 8.4

1. Adder
2. Accumulator
3. Register
4. Controller

The original algorithm shifts the multiplicand left with zeros inserted in the new positions, so the least significant bits of the product cannot change after they are formed. Instead of shifting the multiplicand left, we can shift the product to the right. Therefore, the multiplicand is fixed relative to the product, and since we are adding only 4 bits, the adder needs to be only 4 bits wide. Only the left half of the 8-bit product register is changed during the addition.
Another observation is that the product register has an empty space with the size equal to that of the multiplier. As the empty space in the product register disappears, so do the bits of the multiplier. In consequence, the final version of the multiplier circuit combines the Accumulator with the multiplier. Since, n = 4, a 2-bit counter is needed to count the four shifts, and K = 1 when the counter is in state 3 (112). Figure 5 shows the operation of the multiplier when 1101 is multiplied by 1011. S0, S1, S2, and S3 represent

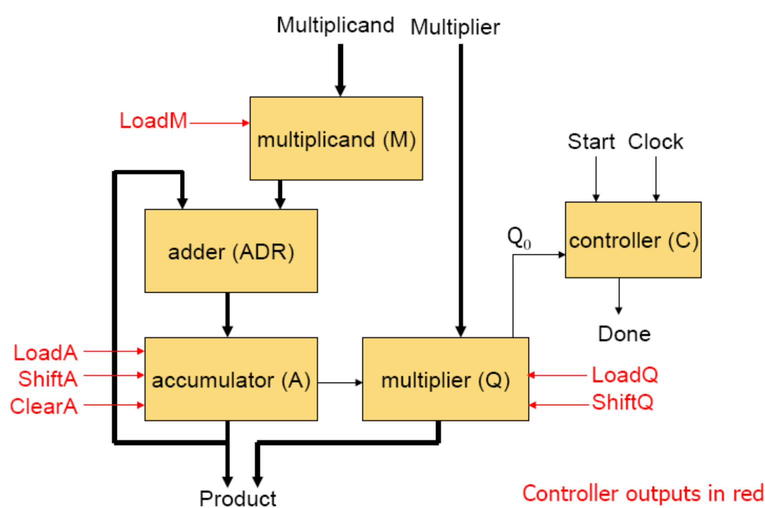

Figure 8.4: System Level Design

states of the control circuit.
At time t0, the control is reset and waiting for a start signal. At time t1, the start signal St is 1, and a Load signal is generated. At time t2, M = 1, so an Ad signal is generated. When the next clock occurs, the output of the adder is loaded into the accumulator and the control goes to S2. At t3, Shift signal is generated, so at the next clock shifting occurs and the counter is incremented. At t4, M= 1, so Adder = 1, and the adder output is loaded into the accumulator at the next clock. At t5 and t6, shifting and counting occur. At t7, three shifts have occurred and the counter state is 11, so K = 1. Since M = 1, addition occurs and control goes to S2. At t8, Sh = K = 1, so at the next clock the final shift occurs and the counter is incremented back to state 00.
At t9, a Done signal is generated. The multiplier design given here can easily be expanded to 8, 16, or more bits simply by increasing the register size and the number of bits in the counter. The add shift control would remain unchanged. 8.5
We start with 8-bit (N) Multiplier which takes 8-bit (N) Multiplier and Multiplicand. Start signal starts the counter (multiplication process) and done signal is generated when multiplication is finished with 16-bit (2N) Product. Top level RTL design on which we design and simulate this multiplier is shown in Fig 8.6.

Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

Figure 8.5: Operation using States Counter

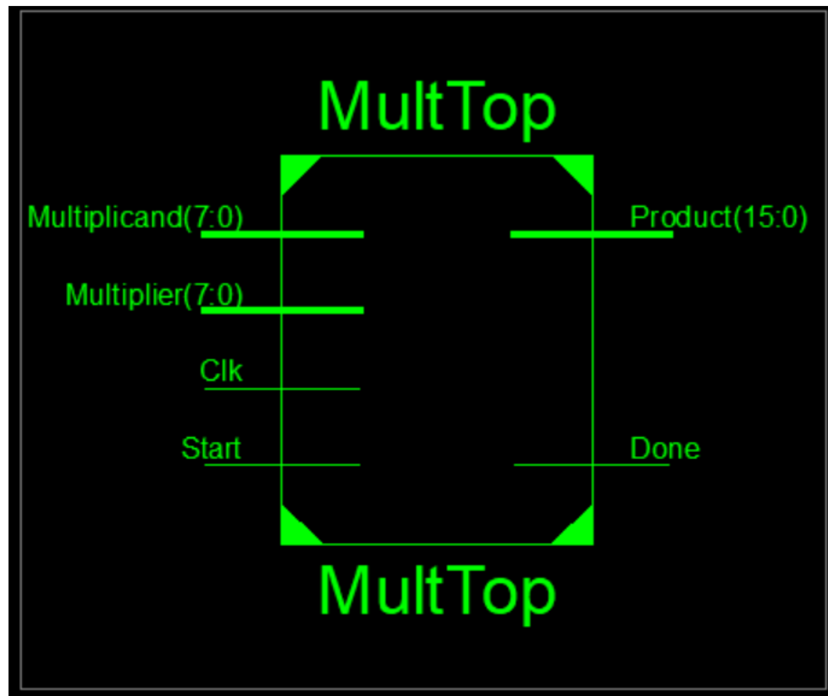

Figure 8.6: Top Level Design (RTL)

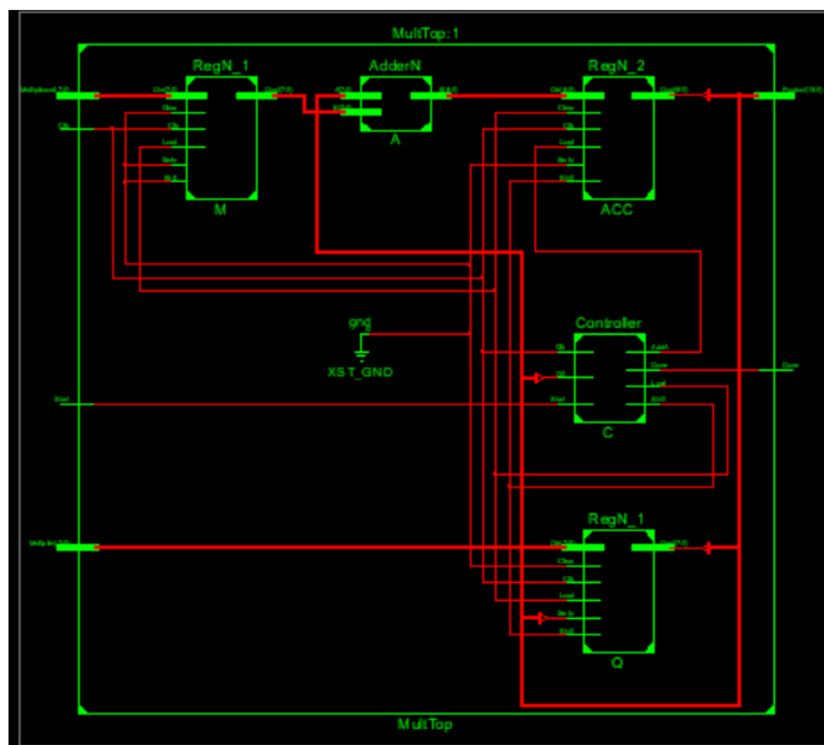

Figure 8.7: Synthesized Model (Xilinx)

8.4 Synthesis and Simulation in Xilinx

Multiplier core is written in VHDL, compiled and simulated in both Xilinx ISE and ModelSim. Top Level RTL in Xilinx ISE after synthesis for N=8 is shown in Fig 8.7. Since, the multiplier needs to be parametrized for N=4, 8, 16 and 32, a generic parameter was introduced. For N=8, the simulation results are shown as in Fig 8.8.

Here, Multiplier is 255 (11111111) and multiplicand is 127 (01111111) to produce 16-bits result 32385 (111111010000001). Delay calculated between Start and Done pulse with 10 ns clock is 500 ns. It is then tested for different number of bits and output delay (time it takes for Start = '1' to Done = '1') is recorded in following table 8.1:

8.5 APB Integration

APB Integration of Multiplier is inspired from OpenCores' Theora Hardware APB Integration in Fig 8.9. [8]

The APB is part of the AMBA hierarchy of buses and is optimized for mini-

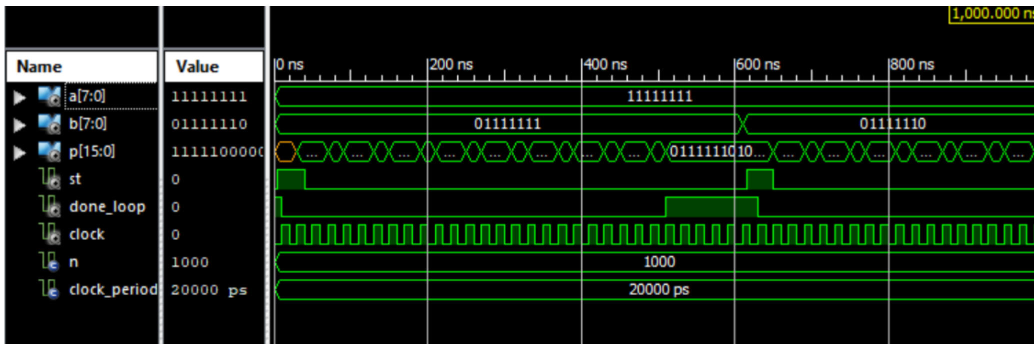

Figure 8.8: Simulation of 8-bit Multiplier

Table 8.1: 4,8,16 and 32-bit Multiplier Variables and Their Latency

N bits	Multiplier	Multiplicand	Product	Delay (ns)
4	4	F	3C	260
8	FF	7F	7E81	500
16	03E8	2710	989680	760
32	00002714	000186A1	000000003BA10B94	780

mal power consumption and reduced interface complexity. The AMBA APB appears as a local secondary bus that is encapsulated as a single AHB slave device. APB provides a low-power extension to the system bus which builds on AHB signals directly.
In order to fit communication protocol of AMBA APB, a wrapper for peripheral (APB Integration) is designed for N-bit Multiplier. APB takes two buses name APBI and APBO, Clock and Reset. [3] APBI and APBO are further distributed into different signals and vectors. Except the wrapping function, it also contains the configuration register. It is 'packaged' and reused as a component in wrapper. Brief diagram of how APB Peripheral will communicate with Leon3 System-on-Chip is shown in Fig 8.10.

8.6 Multiplier APB Integration

To include the IP Core (N bit Multiplier or 'mult') in Leon3, we need to copy it to known library (opencores in this case) and modify 'dir.txt' in Fig 8.11.

In 'mult' folder, we include its basic core files, APB interface and package. They are then synthesized accordingly in 'vhdlsyn.txt'. 'mult.vhd' includes the package for interface 'mult_amba_interface' shown in Fig 8.12.
Finally, we include it in 'devices.vhd' in AMBA Core. For its instantiation

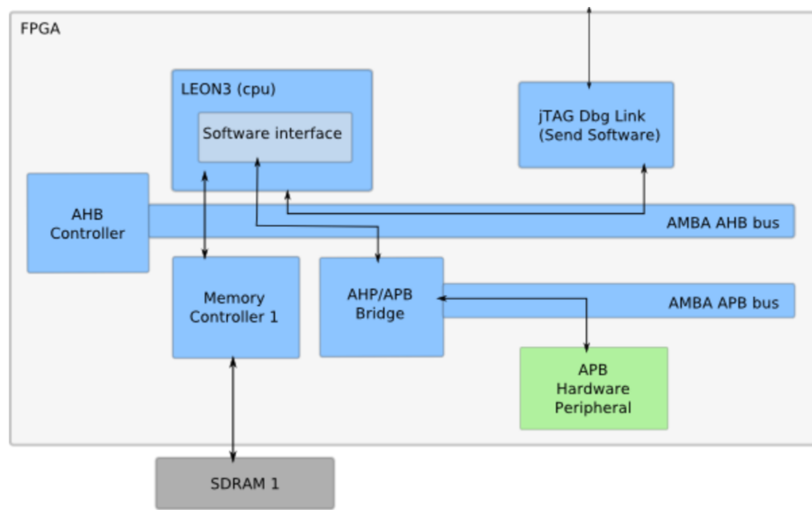

Figure 8.9: APB Integration (Theora Hardware)

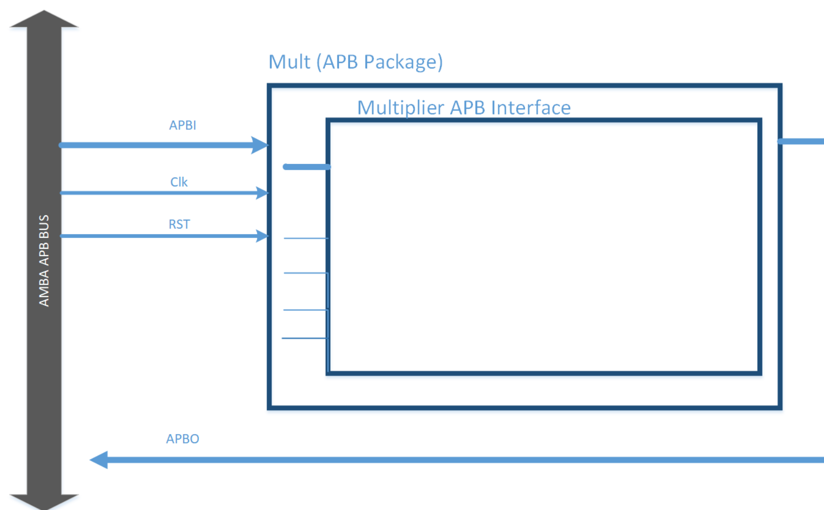

Figure 8.10: Multiplier Integration on AMBA APB Bus

dirs.txt - Notepad
File Edit Format View Help
```
can i2c ge_1000baseX mult
```

Figure 8.11: mult dir

```
mult.vhd
mult_amba_interface.vhd
Components.vhd
Mul_Top.vhd
Multiplier_Controller.vhd
Nbit_Adder.vhd
Nbit_Register.vhd
```

Figure 8.12: vhdlsyn.txt

```
multiplier1 : mult_amba_interface                          -- NBitCSMultiplier
  generic map (pindex  => 8, paddr => 8, pmask => 16#FFF#) --
  port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo(8));
```

Figure 8.13: Multiplier Component Leon3 Top

in Top Module, we include it in 'leon3mp.vhd' as shown in Fig 8.13.
APB slave vector given for 'pindex', 'paddr' and 'apbo' is unique for every peripheral associated. Here, value is 8 which gives us the starting address for this peripheral as 0x80000800.
As a start, we used 16-bit multiplier core as a base line to design multiplier interface. Top module of multiplier takes 16 bits of both multiplier and multiplicand. This value is given by 32-bit input bus 'pwdata'. Multiplier outputs 32-bit Product which is given by 32-bit output bus 'prdata'. Problem arises for control logic signals such as 'Start' and 'Done'. We need proper addressing (paddr) to integrate it with software. Integration system was designed initially for 16-bit system which can also be used for 8 and 4 bits with minor changes. It is given in Fig 8.14.
However, for N=32 (32-bit Multiplier and Multiplicand), situation is different. PWDATA (32-bit bus) takes either Multiplier or Multiplicand at a time. Also, Product is 64-bit while PRDATA is 32 causing splitting of Product and introduction of bit decoder before output at PRDATA as in Fig 8.15.

8.7 Software Integration

Addressing protocol for communication between APB Interface and Software is a bit tricky process. We know base pointer for Multiplier Peripheral is 0x80000800. [8] [20] Taking in view PADDR in Interface module, we define variables as 8.16:
Pointer at 0x80000800 is 32-bit Integer which takes 16-bit multiplier and multiplicand concatenated together. Software takes a while till Done is '1'

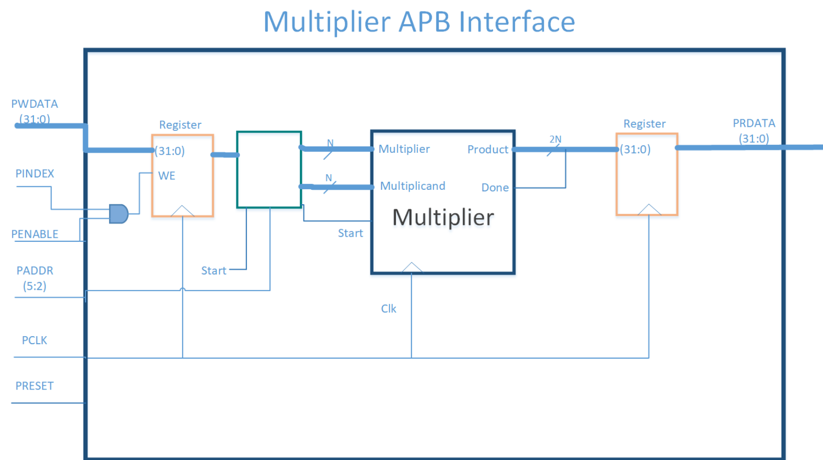

Figure 8.14: Multiplier APB Interface

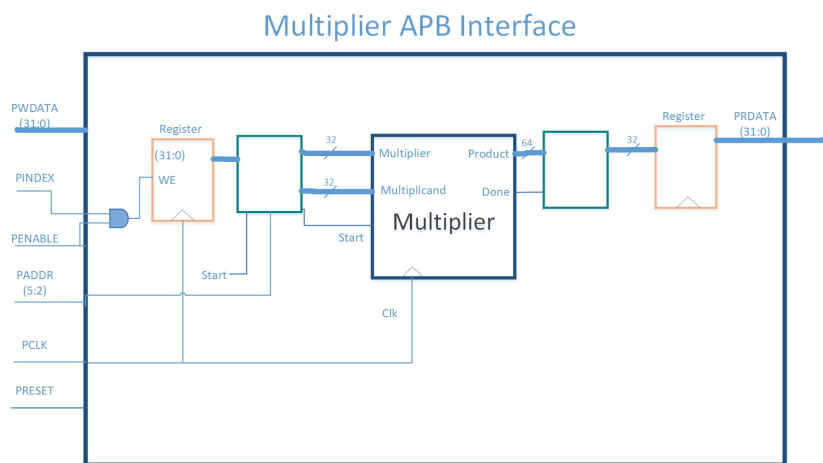

Figure 8.15: 32-bit Multiplier APB Interface

0x80000800	*Done
0x80000804	*Multiplier / *Multiplicand
0x80000808	*Product

Figure 8.16: Address Pointer Declaration for Variables

Table 8.2: Pointer Variables for Multiplier in C

PADDR(5:2)	Pointer Addr	Variable Name
0	0x80000800	Done
1	0x80000804	Multiplier / Multiplicand
10	0x80000808	Product

Figure 8.17: Address Pointer Declaration for Variables (32-bit)

to get product value in 32-bit. Link between PADDR and pointer values can be easily shown in table 8.2 (which can be extended for N=4 and N=8 as well).
Again, for 32-bit Multiplier, system is different. We have to create separate variables for multiplier and multiplicand. Also, product is 64-bit which needs to be declared as two integers concatenated or double (in embedded C) 8.17. [28]
Pointer at 0x80000800 is 32-bit multiplier. 0x80000804 takes 32-bit multiplicand. Software takes a while till Done is '1' to get product value in two 32 bit integers. Link between PADDR and pointer values can be easily shown in table 8.3.

Table 8.3: Pointer Variables for Multiplier in C (32-bit)

PADDR(5:2)	Pointer Addr	Variable Name
0	0x80000800	Done
1	0x80000804	Multiplier
10	0x80000808	Multiplicand
11	0x8000080C	Product1
100	0x80000810	Product2

```
#include <stdio.h>
main()
{
    int *Done = (int *)0x80000800;
    int *Data = (int *)0x80000804;
    int *Product = (int *)0x80000808;

    printf("Multiplier Test\n\r");

    *(Data) = 0x03E82710;

  while(Done)
  {
      printf("Done: 0x%08x \n\r", *(Done));
      printf("Product: 0x%08x \n\r", *(Product));
      break;
  }

    printf("End of test\n");
}
```

Figure 8.18: Example C Compiler Code of 16-bit

8.8 Hardware / Software Verification:

Taking in view variables declared in tables above, we create C file which is compiled for SPARC V8 Processor (Leon3). As an example multiplier code for 16-bit is shown in Fig 8.18.
For Hardware / Software Verification, we use MKPROM which simulates testbench in ModelSim and runs compiled programs. It is a utility program which converts a LEON RAM application image into a bootable ROM image. The resulting bootable ROM image contains system initialization code, an application loader and the RAM application itself. [11]
Its main advantage is that we can simulate and verify our IP Core before implementing it on FPGA. Result of compiled code is shown in figure 8.19. Red mark shows that Multiplier is ported at Slave 8 of APB Bus with starting address of 0x80000800. Yellow mark shows test results of multiplier: 03E8

```
# LEON3 MP Demonstration design
# GRLIB Version 1.5.0, build 4164
# Target technology: inferred  , memory library: inferred
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Cobham Gaisler          LEON3 SPARC V8 Processor
# ahbctrl: mst1: Cobham Gaisler          AHB Debug UART
# ahbctrl: slv0: European Space Agency   LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Cobham Gaisler          AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Cobham Gaisler          Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Cobham Gaisler          Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Cobham Gaisler          Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Cobham Gaisler          AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv8: OpenCores               N Bit Multiplier
# apbctrl:       I/O ports at 0x80000800, size 256 byte
# apbctrl: slv11: Cobham Gaisler          General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# MULTIPLIER V08: APB MULTIPLIER SLAVE module rev 0
# grgpio11: 8-bit GPIO Unit rev 3
# gptimer3: Timer Unit rev 1, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 4, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# ahbuart7: AHB Debug UART rev 0
# leon3_0: LEON3 SPARC V8 processor rev 3: iuft: 0, fpft: 0, cacheft: 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
#  moving .text from 0x00001620 to 0x40000000
#  moving .data from 0x0000c760 to 0x4000b140
# Multiplier Test
#
#
# Done: 0x00000000
#
#
# Product: 0x00989680
```

Figure 8.19: Compiler Code Verification 16-bit

```
multiplier1 : mult_amba_interface                      -- NBitCSMultiplier
  generic map (N => 8, pindex  => 8, paddr => 8, pmask => 16#FFF#) --
  port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo(8));
```

Figure 8.20: N bit in Leon3 Top

and multiplicand: 2710. Similarly, we can create separate compiler codes for 4, 8 and 32-bits.

8.9 N bit Parametrization for APB Interface

For each N value, we had to create separate Interface file for a given IP Core. To parameterize N value, we had to create introduce N generic value for interface file, APB Package and Leon3 Top Module. Example for latter is shown in Fig 8.20.

8.10 GUI Control

Each design has a simple graphical configuration interface that can be started by issuing 'make xconfig' in the template design directory. The tool presents

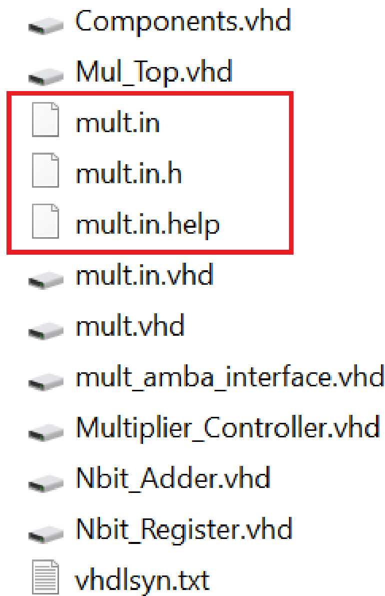

Figure 8.21: in files in Multiplier Core

```
-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := 1;
  constant CFG_GRGPIO_IMASK : integer := 16#0000#;
  constant CFG_GRGPIO_WIDTH : integer := (8);
-- N Bit Multiplier
  constant CFG_MULT_N : integer := (16);
-- GRLIB debugging
  constant CFG_DUART : integer := 1;
```

Figure 8.22: mult.in.vhd and mult.in.h

the user with configuration options and generates the file 'config.vhd' that contains configuration constants used in the design. [12]
Each core has a set of files that are used to generate the core's xconfig menu entries. The xconfig files are typically located in the same directory as the Multiplier HDL files in 8.21. 'mult.in' file defines the menu structure and options for the Multiplier core. 'mult.in.help' gives help function defined in GUI Menu. The two remaining files 'mult.in.h' and 'mult.in.vhd' are used when generating the 'config.vhd' file for a design which typically consists of a set of lines for each core where the first line decides if the core should be instantiated in the design and the following lines contain configuration options.
In GUI Sub Menu Entries, we add variables to 'config.vhd' defined in 'mul.in.vhd' and 'mult.in.h' as in Fig 8.22.
And also 'config.in' as in Fig 8.23.
These are then regenerate GUI to show added features and variables, we

```
mainmenu_option next_comment
comment 'N Bit Multiplier'
  source lib/opencores/mult/mult.in
endmenu
```

Figure 8.23: GUI Control config.in

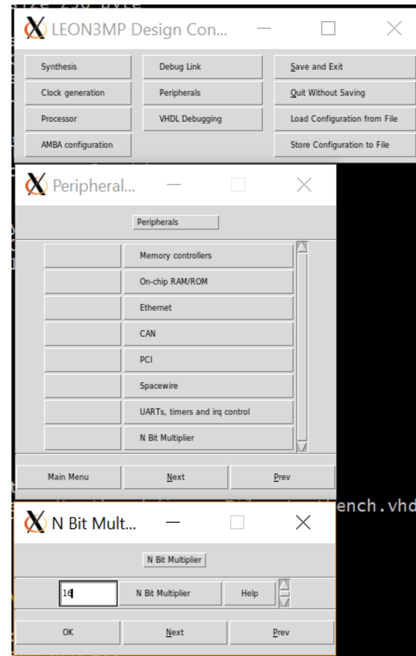

Figure 8.24: Multiplier Configuration in xconfig

defined for Multiplier. Here, we can change its N value from GUI Control without changing HDL files for Multiplier Core and Interface as in Fig 8.24.

Chapter 9

Memory-Mapped Interface (AMBA AHB)

9.1 Introduction

The Advanced High-Performance Bus (AHB) is part of the AMBA hierarchy of buses and is intended for high performance designs with multiple bus masters and high bandwidth operations. [26] AHB-Lite implements the features required for high-performance, high clock frequency systems including:

1. burst transfers
2. split transactions
3. single-cycle bus master handover
4. single-clock edge operation
5. non-tristate implementation
6. wider data bus configurations (64/128/256/512/1024 bits).

9.2 Advanced High-Performance Bus (AHB)

The most common AHB devices (or cores) are internal memory devices, external memory interfaces, and high bandwidth peripherals (AHB/APB Bridge).
Although low-bandwidth peripherals can be included as AHB interface cores, for better performance and less complexity, they are interface to AMBA Advanced Peripheral Bus (APB). [4]
AHB System can be broken down to:

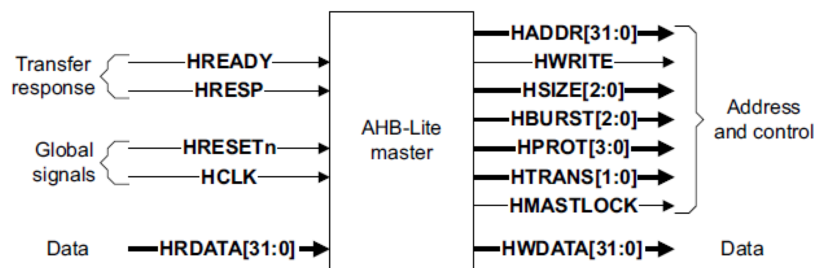

Figure 9.1: AHB Master

1. AHB Master
2. AHB Slave
3. AHB Decoder
4. AHB Multiplexor

9.2.1 AHB Master

It provides address and control information to initiate read and write operations. In our case, it Leon3 Processor and Memory Management Unit 9.1.

9.2.2 AHB Slave

It processes and responds to the signals initiated by the master. It is normally the IP Core interface with Master (Leon3) to get desired results. It uses 'HSELx' signal from decoder to control its response to bus transfer.
AHB interface can be as simple as a register which can be read and written through bus transfers on an on-chip bus. The register will be accessed when a given address address decoder ('HADDR'), or an address within a given range, appears on the bus. The memory address, and the related bus command, is analyzed by an address decoder. It works as a shared resource between software and hardware 9.2.

9.2.3 AHB Decoder

This component 9.3 decodes the address of each transfer and provides a select signal for the slave that is involved in the transfer. It also provides the control signal to the multiplexor.

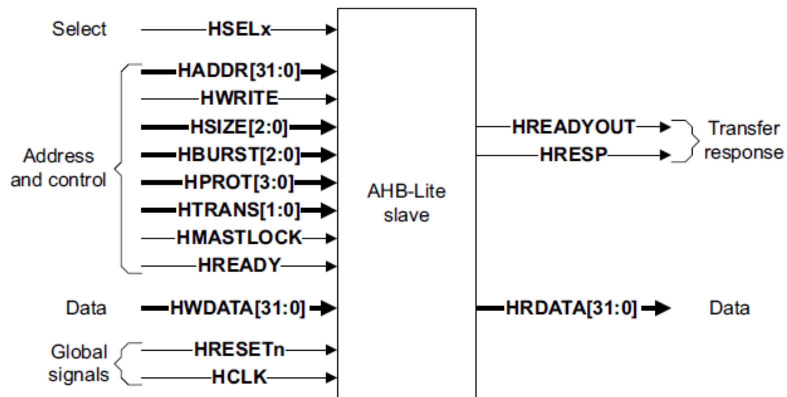

Figure 9.2: AHB Slave

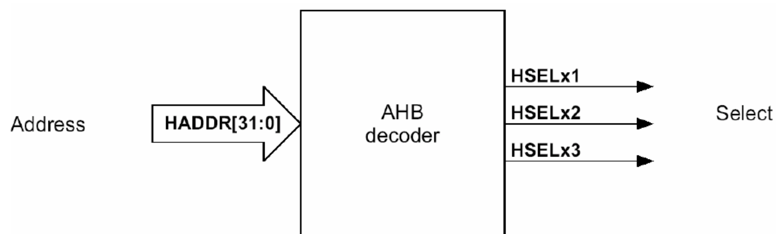

Figure 9.3: AHB Decoder

Table 9.1: AHB Signals

Sr #	Name	Description
1	HCLK	Bus clock
2	HRESETn	AHB reset
3	HADDR [31:0]	AHB Address Bus
4	HSELx	AHB Select
5	HREADY	AHB Strobe
6	HWRITE	AHB Write Enable
7	HSIZE [2:0]	Size of Transfer
8	HBURST [1:0]	Burst Type
9	HPROT [3:0]	Protection Control
10	HTRANS [1:0]	Indicates transfer type
11	HMASTLOCK	Signal is Locked or Not
12	HRDATA [31:0]	AHB Read Data bus
13	HWDATA [31:0]	AHB Write Data bus
14	HRESP	Transfer Acknowledgement

9.2.4 AHB Multiplexor

A slave-to-master multiplexor is required to read data bus and respond to each slave data and signals.

9.3 Co-Processor AHB Slave Interface

Following table 9.1 shows the signals used AHB Slave interface (AHBSI and AHBSO) [4].

9.3.1 AHB Slave Data Transfer with enhanced features

AHB Data Transfer works like APB bus for read and write with a few modifications and enhancements. Simplest write transfer data with no wait states are shown in Fig 9.4 [4].
Read transfer is shown in Fig 9.5.
Simple write transfer in AHB Slave (wrapper for an IP Core) is enabled when 'HSELX' and 'HWRITE' are high. It causes slave to write the data from 'HWDATA' to its internal memory address location given by the master or decoder at 'HADDR'. Once the data is written, then it issues the 'HREADY' and 'HRESP' for acknowledgment of data. In read operation, it will fetch the data from its internal memory for the given address location

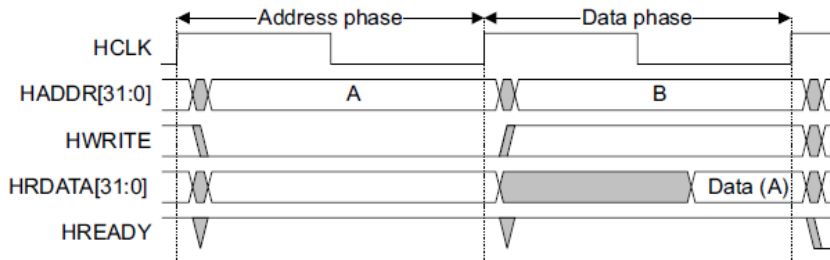

Figure 9.4: AHB Data Write

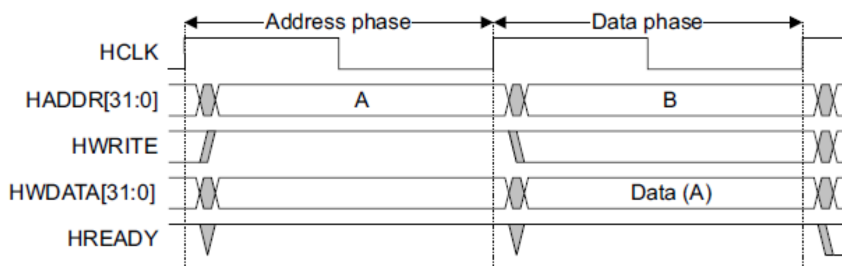

Figure 9.5: AHB Data Read

in 'HADDR' and is given out through 'HRDATA' signal 9.6.
Other enhancements include:

- Transfer Types HTRANS [1:0]: IDLE, BUSY, NONSEQ, SEQ
- Master Transfer Lock: HMASTLOCK
- Transfer Size HSIZE [2:0]: 8,16, 32, 64, 128, 256, 512, 1024
- Burst Operation
- Waited Transfers and Acknowledgement

9.4 Software Interface

In software, the representation of a register is easy to do using an initialized pointer. The base address of this pointer is determined by Slave bus index of the AHB Peripheral or Co-Processor. For example, bus index for Slave (HINDEX=8) will be 0xFF000000 with 1-Mbytes memory and Address (HADDR) at 16FF0. Following diagram 9.7 gives us example how to architect software interface code:

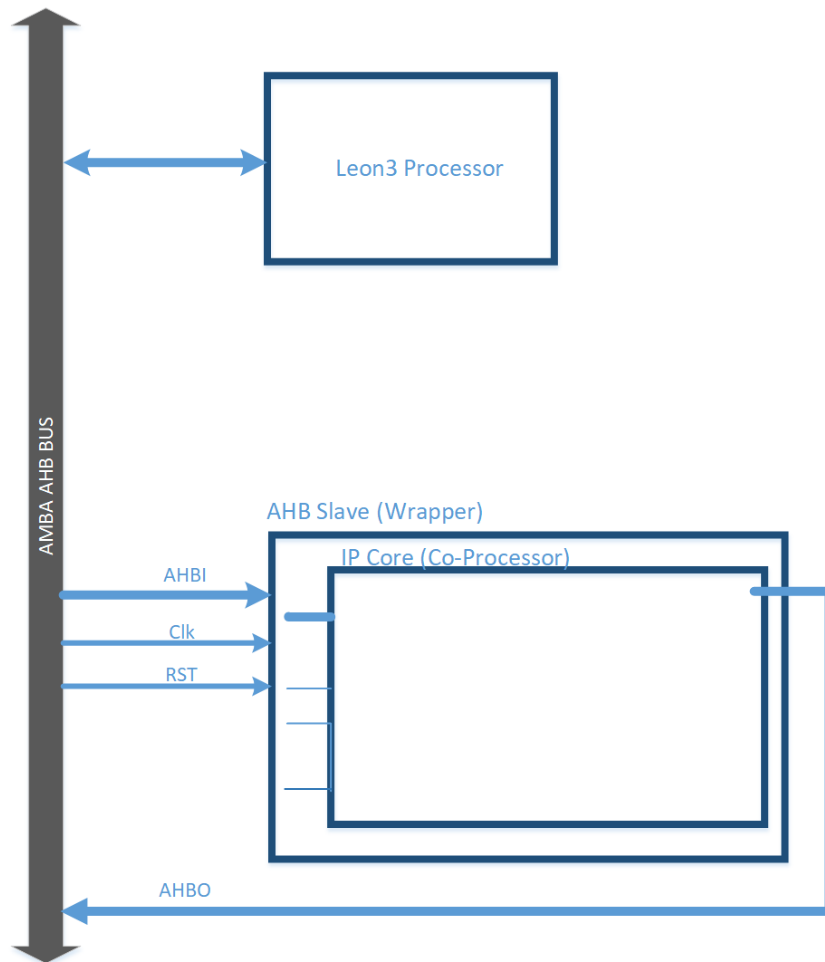

Figure 9.6: AHB Slave and Leon Interface

```
Int *MMRegister = (int*) 0xFF00000;  //Base Address of Co-Processor Wrapper
// write the value '0xFF' into the register
*MMRegister = 0xFF;
// read the register
int value = *MMRegister;
```

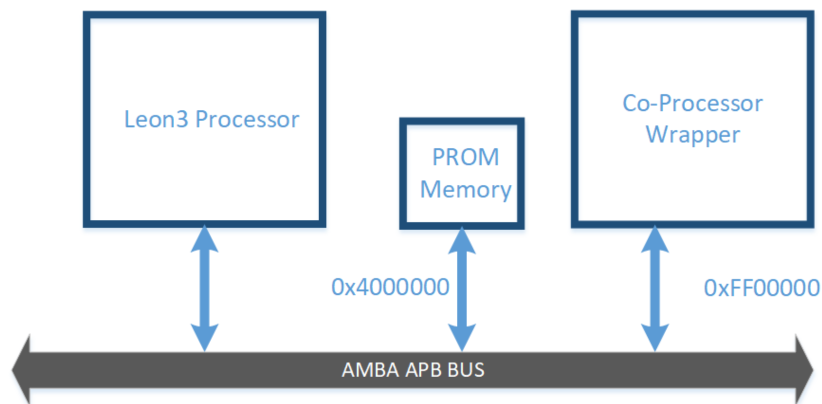

Figure 9.7: Software Memory Interfacing

9.5 Register Example

The IP core has one memory mapped 32-bit register that will be reset to zero. The register can be read or written from default HADDR. The core's bus index, base address and mask settings are configurable via VHDL generics (HINDEX, HADDR, HMASK). The HADDR and HMASK VHDL generics are propagated via the AHBSO.HCONFIG signal and the index is propagated via the AHBSO.HINDEX signal. These values are then used by the AHB bridge to generate the AHB address decode and slave select logic [12]. Its RTL view as well as synthesized model is similar to APB Slave RTL 7.8 in section 7.5.
Its software interface code is written as same model as APB Interface in Fig reffig:AHB_Software.
For Hardware / Software Verification we use MKPROM in similar pattern to test and verify this model. In Fig 9.9 we can see that 1 Mbyte Memory is initialized as AHB Slave 8 (HINDEX=8). After initialization, our example code is run to verify our IP Core, Interface and Software. 9.10

```c
#include <stdio.h>
main()
{
    int *baseaddr_p = (int *)0xFF000000;

    printf("Register Test\n\r");

    // Write multiplier inputs to register 0
    *(baseaddr_p+0) = 0x00020003;
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+0));

    //*(baseaddr_p+1) = 0x00067611;
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+0));
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+1));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+2));
    printf("Wrote: 0x%08x \n\r", *(baseaddr_p+2));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+4));
    printf("Wrote: 0x%08x \n\r", (baseaddr_p+8));

    printf("End of test\n\n\r");
}
```

Figure 9.8: Register Test Program

```
# LEON3 MP Demonstration design
# GRLIB Version 1.5.0, build 4164
# Target technology: inferred  , memory library: inferred
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 12
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Cobham Gaisler          LEON3 SPARC V8 Processor
# ahbctrl: mst1: Cobham Gaisler          AHB Debug UART
# ahbctrl: slv0: European Space Agency   LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Cobham Gaisler          AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv8: OpenCores               AHB Register
# ahbctrl:       memory at 0xff000000, size 1 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Cobham Gaisler          Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Cobham Gaisler          Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Cobham Gaisler          Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Cobham Gaisler          AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Cobham Gaisler          General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 3
# gptimer3: Timer Unit rev 1, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 4, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# ahb_register8: Example core rev 0
# ahbuart7: AHB Debug UART rev 0
# leon3_0: LEON3 SPARC V8 processor rev 3: iuft: 0, fpft: 0, cacheft: 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
```

Figure 9.9: Register initialization in Leon

```
# Register Test
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0xff000008
#
#
# Wrote: 0x00020003
#
#
# Wrote: 0xff000010
#
#
# Wrote: 0xff000020
#
#
# End of test
#
```

Figure 9.10: Test Verification

Chapter 10

AES-128 AHB Interface

10.1 Introduction

The Advanced Encryption Standard (AES) specifies a FIPS- approved cryptographic algorithm that can be used to protect electronic data. AES-128 pipelined cipher core which is downloaded as open source project from OpenCores, uses AES algorithm which is a symmetric block cipher to encrypt (encipher) information. Here the AES algorithm is capable of using cryptographic keys of 128-bit to do this conversion. It takes 128-bit Unciphered data and Key Data (symmetric block cipher) and outputs 128-bit ciphered data. This core is designed in Verilog and needs to be packaged in VHDL to interface it with AMBA bus and Leon Processor. [29]

10.2 AES-128 Synthesis and Simulation

Before interfacing the core with AHB and Leon Processor, we synthesize it in Xilinx and simulate it in Modelsim with given testbench with 284 input and output vectors for verification. Once 'Data Out Valid' is high, it outs 128-bit ciphertext every clock cycle which means its throughput is 1 clock cycle and latency of 42 clock cycles. Its synthesized top module can be shown as in Fig 10.1.
Similarly its simulation is shown as in Fig 10.2.

10.3 AHB Integration

In order to fit communication protocol of AMBA AHB, a wrapper for peripheral is designed for AES-128. APB takes two buses name AHBSI and

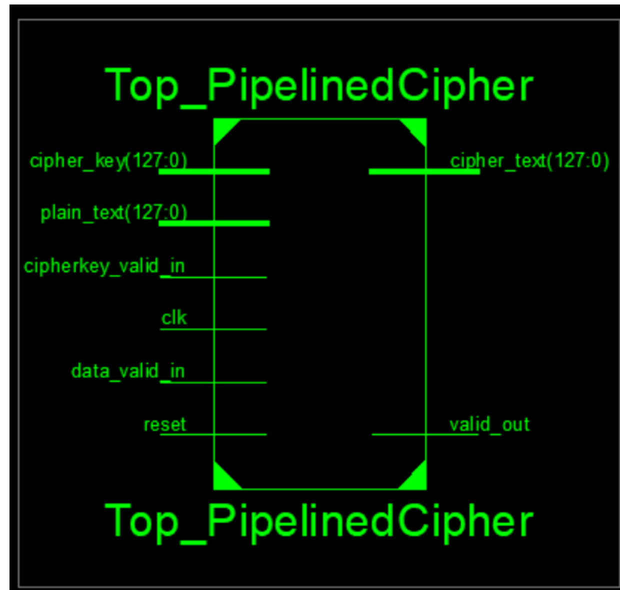

Figure 10.1: AES Top Module

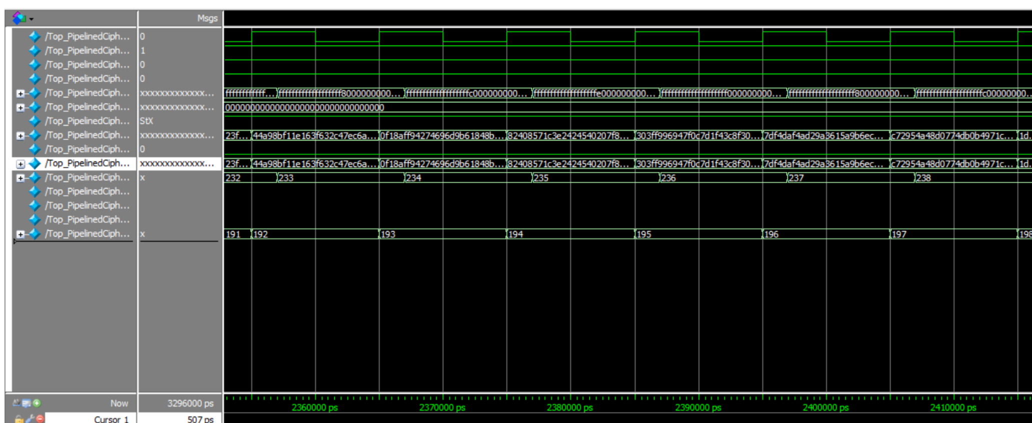

Figure 10.2: AES-128 Simulation in ModelSim

AHB I/O address offset	Register
0x00	Control Register
0x10	Data Input 0 Register
0x14	Data Input 1 Register
0x18	Data Input 2 Register
0x1C	Data Input 3 Register
0x20	Data Output 0 Register
0x24	Data Output 1 Register
0x28	Data Output 2 Register
0x2C	Data Output 3 Register
0x3C	Debug Register

Figure 10.3: GRAES Registers

AHBSO, Clock and Reset. AHBSI and AHBSO are further distributed into different signals and vectors. Except the wrapping function, it also contains the configuration register. Since, the core is written in Verilog, it has to be 'packaged' in VHDL and re-used as a component in wrapper (AES AMBA Interface). Its communication with Leon3 Master using AHB bus is similar to AHB Register interface shown in Fig 9.6.
AHB slave signals given for 'HINDEX', 'HADDR' and 'AHBSO' is unique for every peripheral associated. We take HINDEX=8 and HADDR=16FF0 in similar fashion as in 32-bit Register example 9. Here, the problem arises for bus width. AHB Read and Write both take 32-bit data while AES-128 works on 128-bit data for both input and output. Thus, a system needs to designed i.e. taking four 32-bit data wires to be concatenated to 128-bit Key and Data registers at input and split from 128-bit Read Data register at output of Wrapper designed. to find a solution we moved to GRAES, an encryption standard commercial IP Core designed by Gaisler (they also designed Leon3 Processor) [16]. Luckily, their signals and register configuration was available as shown in Fig 10.3.
Also, for integration of other signals used by the core such as 'Key_Valid', 'Data_Valid' and 'Data_Out_Valid', we designed hardware interface with proper addressing with HADDR taking 4-bits multiplexor selection. Data path RTL for AES Wrapper is shown in Fig 10.4.
The synthesized model for AES AMBA Interface in Xilinx can be shown in Fig 10.5.

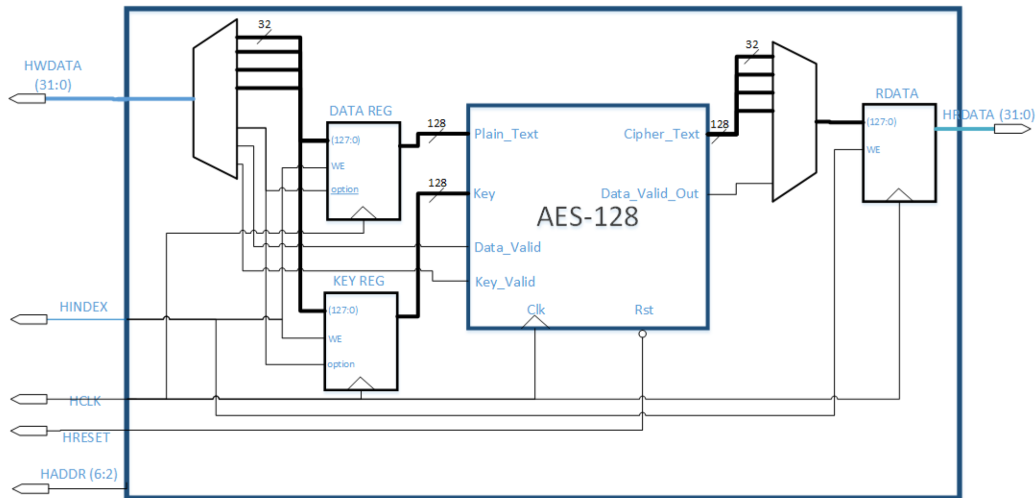

Figure 10.4: AES Wrapper (Interface) RTL

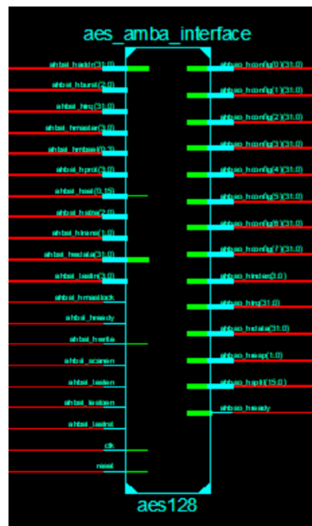

Figure 10.5: AES AHB Interface Synthesized Model

Table 10.1: AES-128 Registers with Pointer Addressing

HADDR	REG (INT*)	Signal (Variable)
0000	00	Option (Key/Data)
0001	04	Start Operation
0010	08	Valid Data Out
0100	10	Write Register 0
0101	14	Write Register 1
0110	18	Write Register 2
0111	1C	Write Register 3
1000	20	Read Register 0
1001	24	Read Register 1
1010	28	Read Register 2
1011	2C	Read Register 3

10.4 Software Integration

In software, the representation of a 128-register is used using 32-bit (Integer) pointer. The base address of this pointer is determined by Slave bus index of the AES Wrapper i.e. bus index for Slave (HINDEX=8) will be 0xFF000000 with 1-Mbytes memory and Address (HADDR) at 16FF0. Using GRAES Register example we architect pointer declaration for addressing as shown in Table 10.1. We use 4 32-bit Write and Read Registers. For Key and Plain Text values, HADDR(6:2) is decoded at '0000' as '0' and '1' respectively.
To interface software variables with pointer addressing decoded in HADDR, we need to coincide Reg(int*) in table 10.1 with 32-bit integer pointers declared in software code in C. The declared variables to be used in C code are shown in Fig .

10.5 Verification

For Hardware / Software Verification we use MKPROM in similar pattern to test and verify this model. In Fig 10.7 we can see that 1 Mbyte Memory is initialized as AHB Slave 8 (HINDEX=8).
Key data which is 128-bit is broken down into 4 32-bits as follows:

- Register 0 : [00000000]
- Register 1 : [00000000]
- Register 2 : [00000000]

Address	Pointer
0xFF000000	*Option
0xFF000004	*Start
0xFF000008	*aes_done
0xFF000010	*DataIn0
0xFF000014	*DataIn1
0xFF000018	*DataIn2
0xFF00001C	*DataIn3
0xFF000020	*DataOut0
0xFF000024	*DataOut1
0xFF000028	*DataOut2
0xFF00002C	*DataOut3

Figure 10.6: AES Pointer Variables in C

```
# LEON3 MP Demonstration design
# GRLIB Version 1.5.0, build 4164
# Target technology: inferred  , memory library: inferred
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 12
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Cobham Gaisler          LEON3 SPARC V8 Processor
# ahbctrl: mst1: Cobham Gaisler          AHB Debug UART
# ahbctrl: slv0: European Space Agency   LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Cobham Gaisler          AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv8: OpenCores               AES 128
# ahbctrl:       memory at 0xff000000, size 1 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Cobham Gaisler          Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Cobham Gaisler          Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Cobham Gaisler          Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Cobham Gaisler          AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Cobham Gaisler          General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 3
# gptimer3: Timer Unit rev 1, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 4, #cpu 1, eirq 0
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# AES 128 V08: AHB AES 128 SLAVE module rev 0
# ahbuart7: AHB Debug UART rev 0
# leon3_0: LEON3 SPARC V8 processor rev 3: iuft: 0, fpft: 0, cacheft: 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
```

Figure 10.7: AES Pointer Variables in C

```
#  moving .text from 0x00001620 to 0x40000000
#  moving .data from 0x0000c8e0 to 0x4000b2c0
# AES128 Test
#
#
# Read Cipher Text
#
# Read Register 0: 0x0336763e
#
# Read Register 1: 0x966d9259
#
# Read Register 2: 0x5a567cc9
#
# Read Register 3: 0xce537f5e
#
# End of test
#
```

Figure 10.8: AES Pointer Variables in C

- Register 3 : [00000000]

Similarly, for Plain Text data (Data In) is shown as:

- Register 0 : [f34481ec]
- Register 1 : [3cc627ba]
- Register 2 : [cd5dc3fb]
- Register 3 : [08f273e6]

The Cipher Text (Data Out) is verified with 4 32-bit Registers as shown in Fig 10.8.

Chapter 11

Conclusion and Future Work

In this thesis a simple guidelines and workflow to infer the required procedure for the integration of peripheral or coprocessor with complicated architecture of Leon3 processor using AMBA bus architecture in Memory-Mapped interface. Coprocessor interface can also be used but as told earlier it is not included in Leon3 documentation nor it is recommended by Gaisler. The proposed approach shows that bus architecture plays an important role in integration. Although Memory-Mapped interface has drawbacks of variable latency and throughput, its easy-to-use configuration and interface makes it a valuable tool in time-constraint implementation environment.

11.1 Integration Examples

Apart from arithmetic cores, latest in on-chip coprocessor integration is large graphic-processors being attached with general-purpose processors to form Accelerated Processing Unit (APU) microprocessors which are being used by AMD [35]. The Arm Machine Learning processor is another example of configuraing GPPs and embedding hardware acclerators to optimize processor for machine learning and data science tasks which has higher performance and low power consumption [2].

11.2 Future Work

Similarly, in future it may be desired in the case of AES cipher and decipher, if embedded with processor, in our case, entangled in pipeline of Leon3 to create custom instruction set that is automatically encrypted or decrypted per instruction so that we no longer Coprocessor for AES.

Bibliography

[1] Aeroflex Gaisler. *TSIM2 Simulator User's Manual*, 2012.

[2] ARM. Arm machine learning processor.

[3] ARM. *AMBA 3 APB Protocol*. ARM Inc, 2004.

[4] ARM. Amba 3 ahb-lite protocol specification v1.0, 2006.

[5] John Catsoulis. *Designing Embedded Hardware*. O'Rielly, 2005.

[6] Cauestr. Reconfigurable computing lab 04: Softcore processors and hardware acceleration, 2012.

[7] COBHAM. *GRMON User Manual*. Cobham Gaisler, 2017.

[8] André Luiz Nazareth da Costa. Hardware implementation of theora decoding.

[9] Paola Ceminari Ariel Arelovich Martín Di Federico. Aes block cipher implementations with amba-ahb interface. In *2017 1st Conference on IEEE PhD Research in Microelectronics and Electronics Latin America, PRIME-LA 2017*, 2017.

[10] franiodoro. Intel hex2bin. Sourceforge.net, 2013.

[11] Cabham Gaisler. Mkprom2.

[12] Cobham Gaisler. Grlib ip library user manual. Technical report, Gaisler, 2016.

[13] Cobham Gaisler. *Bare-C Compiler*. COBHAM, 2017.

[14] Cobham Gaisler. *LEON/GRLIB Guide Configuration and Development Guide*. Cobham, 2017.

[15] Cobham Gaisler. Leon3 processor, 2018.

[16] Gasiler. *GRLIB IP Core User's Manual*. Gaisler, 2016.

[17] Cygwin Authors Grp. Cygwin for windows, 2018.

[18] Hobbes. Why did the esa choose sparc for leon? stackexchage.com, 2013.

[19] I. Hodjat, a. Verbauwhede. Interfacing a high speed crypto accelerator to an embedded cpu. In *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.*, volume 1, pages 488–492, 2004.

[20] Jeff Johnson. Creating a custom ip block in vivado, 2014.

[21] Jr and Lizy Kurian John Charles H. Roth. *Digital Systems Design using VHDL*. Thomson Learning, 2008.

[22] Sven Åke Andersson. Leon3 soft processor: Guide, 2012.

[23] Jason G. Tong Ian D. L. Anderson Mohammed A. S. Khalid. Soft-core processors for embedded systems. *2006 International Conference on Microelectronics*, pages 170–173, 2006.

[24] Luis Azuara Pattara Kiatisevi. Design of an audio player as system-on-a-chip. Master's thesis, University of Stuttgart, 2002.

[25] Pedro Martos. Bin2coe. sourceforge.net, 2013.

[26] Tadikonda Nagarjuna. Implementation of different operations for data transfer for amba-advanced high performance bus. *International Journal of Computer Science and Mobile Computing*, 3:768–776, 2014.

[27] Shobana Padmanabhan. Automatic application-specific customization of softcore processor microarchitecture. Master's thesis, Washington University in St. Louis, 2006.

[28] Tutorials Point. C data types.

[29] Amr Salah. Aes-128 pipelined encryption, 2013.

[30] J. B. Nade Dr. R. V. Sarwadnya. The soft core processors: A review. *Ijireeice*, 3:197–203, 2015.

[31] Patrick R. Schaumont. *A Practical Introduction to HArdware/Software Co-Design*. Springer, 2010.

[32] Oregano Systems. 8051 ip core, 2006.

[33] Oregano Systems. Oregano 8051 manual, 2006.

[34] Sven Keller Tanya Vladimirova, David Eamey and Prof Sir Martin Sweeting. Floating-point mathematical co-processor for a single-chip on-board computer. *ResearchGate*, 2015.

[35] Techterms. Apu definition.

[36] Simon Teran. 8051, 2009.

[37] X. Guo J. Fan P. Schaumont I. Verbauwhede. *Programmable and Parallel ECC Coprocessor Architecture: Tradeoffs between Area, Speed and Security*. Springer, 2009.

[38] Wikipedia. Central processing unit. *Wikipedia*, 2018.

[39] Wikipedia. Intel mcs51, 2018.