Interfacing a Convolution Engine with AJIT Processor FPGA System

Prathu Baronia

Supervisor : Prof. Madhav Desai

Department of Electrical Engineering
Indian Institute of Technology, Bombay

This thesis is submitted as part of

Dual Degree Project

June 26, 2019

I would like to dedicate this thesis to my family and friends, who supported me through both good and bad times.

Approval Sheet

The thesis/dissertation/report entitled "Interfacing a Convolution Engine with AJIT Processor FPGA System" by Prathu Baronia is approved for the degree of Master's in Technology in Microelectronics.

Examiner

_____

Supervisor

_____

Chairperson

_____

Date: _____

Place: _____

Declaration

I declare that this written submission represents my own ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misinterpreted or fabricated or falsified any idea/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

_____

Signature

_____

Name of Student

_____

Roll No.

Date: _____

Acknowledgements

I would like to acknowledge my supervisor, Prof. Madhav Desai for his guidance and mentor-ship during the period I worked with him, which helped inculcate the pursuit of excellence in both in my work and my conduct. I would like to acknowledge Mr. Mandar Datar a Phd candidate under Prof. Sachin Patkar for guiding me in my initial days and offering help at crucial moments in this journey.

Abstract

We head out to develop a FPGA system around AJIT cores and integrating HPC peripherals designed though inhouse and industry standard HLS tools into this system to increase performance of computationally expensive operations such as convolution of images. The goal of this project is to build a FPGA system capable of booting a custom built OS on the AJIT core. In the first part of the document we focus on the development cycle of the FPGA system where we first describe the processor itself, detailed need for such a system, our setup choices, a telescopic system view, individual block designs and respective design choices made, and the test procedures for the system.

In second part of the document we start the discussion on development of a Convolution core using the inhouse HLS tools. Here we discuss the the need of such peripherals which we later on justify through performance metrics also. Later on we discuss how we began designing the peripheral, the design of the internal architecture and subsystems, the choice of design tools and also our test setup. Finally we compare the hardware resource usages and timing results from different variants of the generated convolution engine.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Part I

AJIT FPGA Model

1 Introduction

1.1 Introduction to AJIT

AJIT is an indigenous processor which has been designed at IIT Bombay and is currently in its second design and manufacturing iteration. AJIT is based on the open SPARC-v8 ISA and is currently a 32-bit processor. The current design runs at 100 MHz and has been manufactured at SCL, Chandigarh using the 180nm technological node.

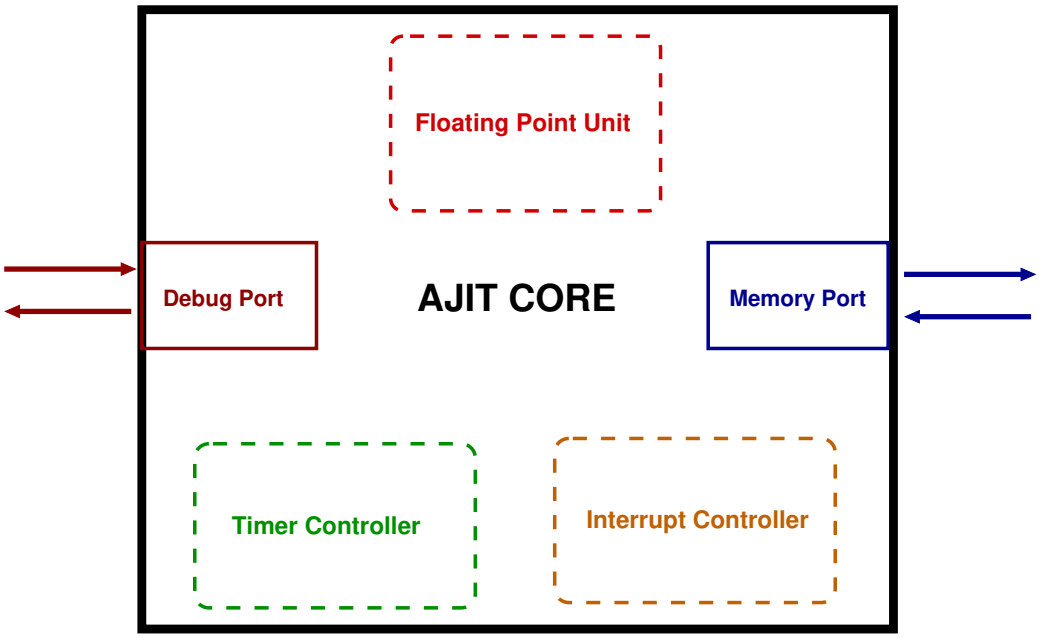


Figure 1.1: Overview of AJIT

1.1.1 Support for AJIT

AJIT has a functionally correct and verified C model which is used to verify new additions to the processor and user space applications. AJIT also has a VHDL model which we

incorporate into the later stages of this project and the same model later on loads different programs (before testing loading of an OS) directly from the DRAM.

1.2 Introduction to the Project

The main aim of this project is to provide an alternative to an earlier FPGA model designed around AJIT processor which had limited on-board memory of 4MB and hence could borderline load a Linux system since the minimum memory requirement for a Linux operating system to boot is around 4MB. The operating system thus loaded/booted had very limited kernel modules and userspace drivers to interface with hardware elements.

This project is focussed at leveraging high DRAM storage capabilities and the high read and write speed of PCIe bus of the Xilinx VC709 board in order to make it capable to boot a heavily loaded embedded OS(quite possibly an RTOS) with all the basic kernel modules and userspace drivers to interface with hardware elements such as networking.

This process of booting an embedded OS on a FPGA platform is really advantageous as compared to booting OS on development boards based around ASICs as we outline later. This project aims to integrate this FPGA system with a `C++` based application to implement a router system down the line.

We aim to incorporate soft core peripherals such as a Ethernet controller, USB controller etc to this FPGA system to ease the access to the system through `ssh` connections or through `serial` connections. These kinds of easily integrable peripherals are expected to make the development of custom applications really swift.

This project as we demonstrate later can also be used as a test-bed for HPC codecs development and testing. These peripherals can be distributed along with the AJIT processor as part of its HPC peripheral library.

2 | Need for a FPGA system

The FPGA system developed around AJIT Core consists of the necessary peripherals which enable the user to boot a decently packed embedded OS on it. A flexible FPGA system is preferred where the integration of new peripherals is as smooth as possible by keeping the disturbance caused to the current configuration as minimal as possible. Here in this system adding a new processor core is as smooth as adding a new peripheral which is a huge advantage over traditional systems based around a single core, this makes the transition to a multiprocessor model really easy and convenient.
The above system acts as a test-bed for various purposes such as:-

- Easy custom OS booting cycles : The system can be easily loaded with custom operating systems compiled for AJIT from scratch or through embedded OS build systems such as Buildroot or Yocto.

- Debugging of Bootloader related bugs : The FPGA system provides the boot log over serial(UART) to the host machine which is really helpful in debugging bugs related to initialization of hardware, loading of modules and is also helpful in providing information necessary in order to optimize to boot time for the system to the maximum extent.

- Debugging of Driver related bugs : Loading and unloading of kernel modules, memory inspection from host machine directly without going through the on-board processor channel.

- Peripheral testing and performance measurement : Any AXI peripheral can be tested directly from the host machine without interfering with the normal working

of the system with proper care for isolation obviously.

- Faster bootstrapping for custom projects : The FPGA system is highly configurable and is designed to offer basic building blocks and their control interfaces and respective drivers for their easy integration and development of minimum working example systems.

3 | Introduction to FPGA System

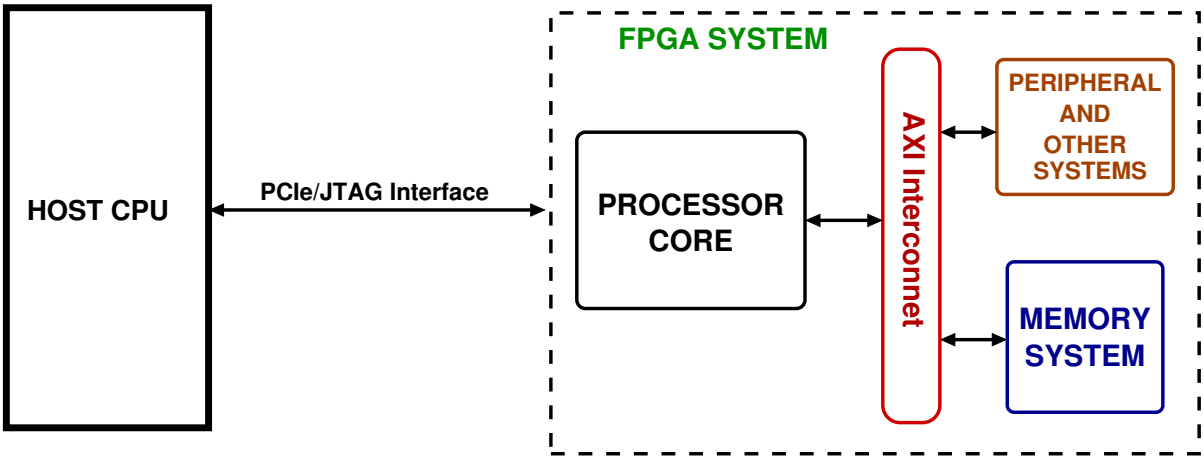


Figure 3.1: Introduction to the FPGA System

The previous block diagram showed a telescopic view of the mentioned FPGA system. Now we expand a bit on the FPGA System block mentioned in the previous diagram. Here we introduce a generic interface architecture for building a system. As shown in the figure above the generic interface diagram of the fpga model. Every subsystem including the processor core hanging from the interconnect appear just like a memory mapped peripheral to it. This kind of architecture as explained at last in future work would provide a relatively easy extension to a Multiprocessor SMP model as compared to a processor centric model. Some peripherals would act as AXI Slaves and some as AXI Masters depending on their functionality. Another advantage to this architecture is that since each peripheral is memory mapped and JTAG has a direct access to the Interconnect one can probe and read-write from-to any peripheral independent of the processor. As shown later after interfacing the PCIe to AXI we get another possible data flow channel other

than the standard UART and JTAG. The number of peripherals that can be interfaced in this fashion is only limited by the number of bits used for addressing on the interconnect.

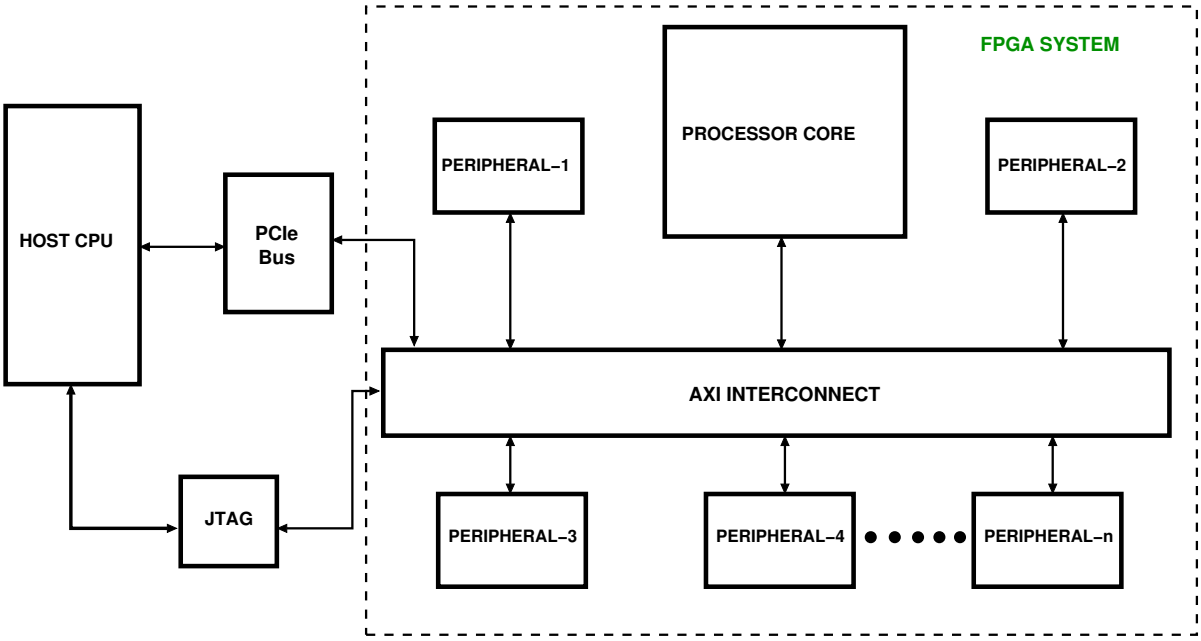


Figure 3.2: FPGA System Expanded

4 System Level Description

4.1 Master-Slave Pairing

The whole FPGA system development circles around this fundamental idea of Masters and slave peripherals. The slave peripheral needs to be in the memory map of the Master peripheral so that the master can access it by making memory requests on the interconnect. The below figure shows a 1 Master - 1 Slave configuration with an interconnect IP interfacing them. As is visible the interconnect needs both the clocks on which the peripherals operate in order to interface them by employing dual clocking FIFOs for each AXI channel.

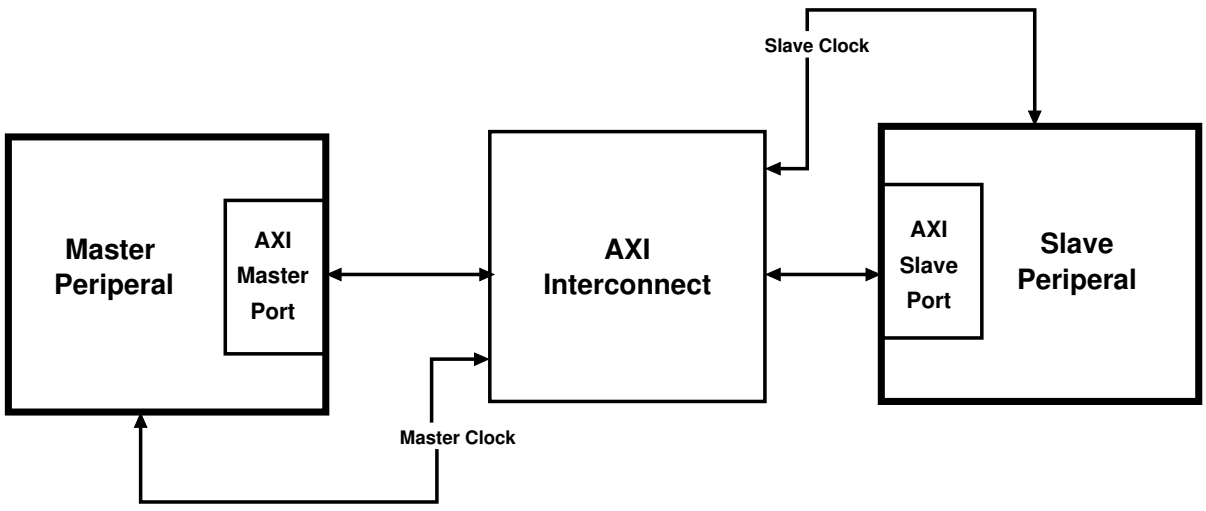


Figure 4.1: 1 AXI Master - 1 AXI Slave Configuration

The AXI Interconnect which bind the whole system is a NoC (Network on Chip) implementation of the communication subsystem and hence brings notable improvements over conventional bus and crossbar communication architectures. Networks-on-chip improve

the scalability of systems-on-chip and the power efficiency of complex SoCs compared to other communication subsystem designs.

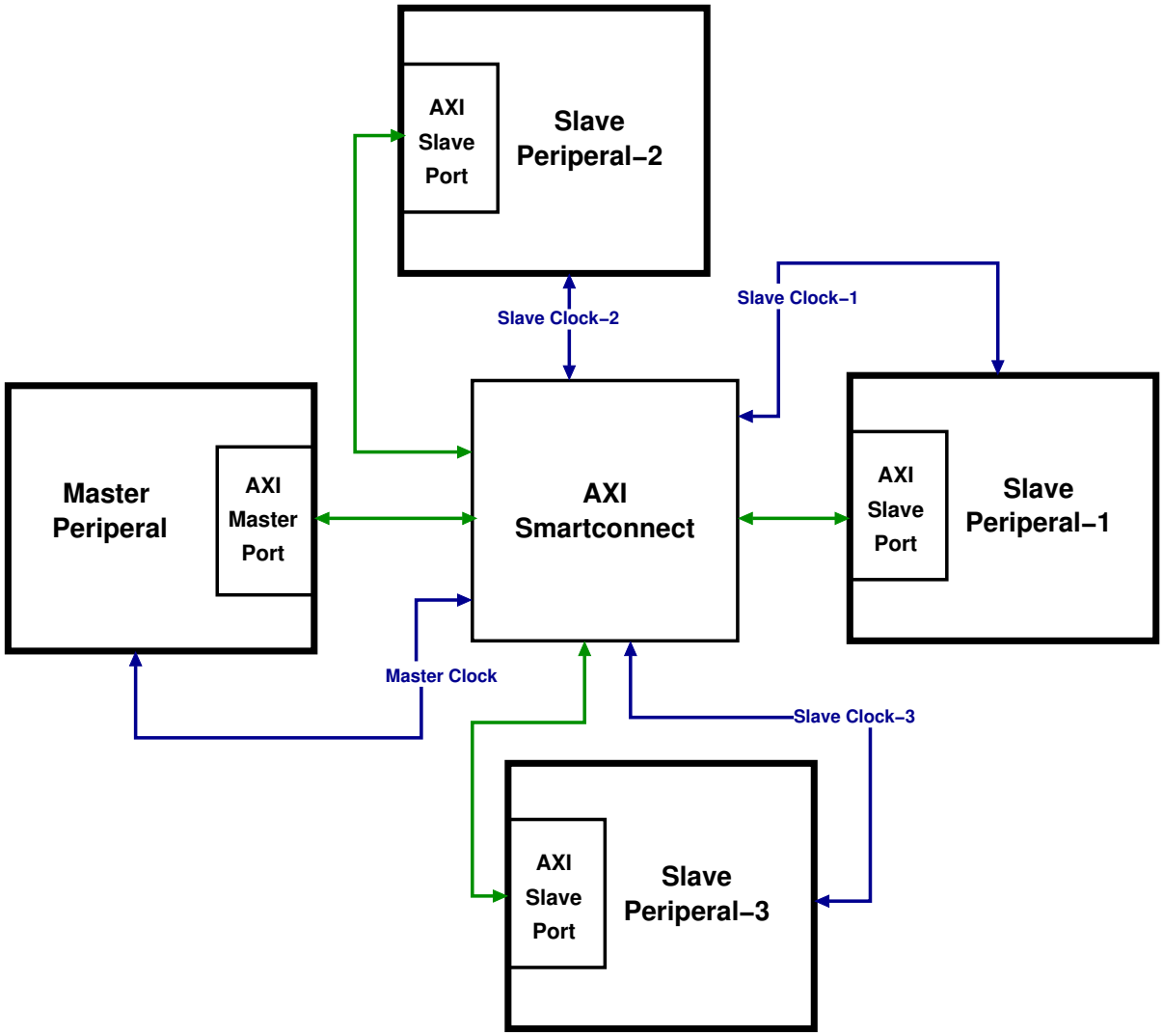


Figure 4.2: 1 AXI Master - 3 AXI Slave Configuration

4.2 Dual Clocking FIFOs in Interconnects

Dual clock FIFOs are designed for two circuits operating in different clock frequencies to communicate with each other. There is a read side and write side where data is stored into the internal memory of the FIFO using the write side clock and then read from the internal memory using the read side clock. AXI Smartconnect IPs from Xilinx employ dual clocking FIFOs to interface Master and slaves which have operate at different clock frequencies for example AJIT core operates at 100MHz but the PCIe-AXI Translation IP

operates at 250MHz thus we need something like an AXI Smartconnect which can handle this interfacing situation.

4.3 Memory mapped devices

This age old approach is a regular practice on desktop, portable and embedded devices. This makes a device registers appear like a memory location which can be manipulated with userspace applications.This makes writing to AXI peripherals through AJIT really convenient, the reading and writing just becomes writing to the device file `\dev\mem` on the AJIT Linux OS, this will initiate the appropriate memory request from the AJIT core to the AXI interconnect through the AFB-AXI Bridge to the on board DRAM.

4.4 Peripheral Control Interface

The AXI Slave interface for each peripheral on the AXI Interconnect has some data registers and some control registers which include start idle, and done bits. The start bit is written high after writing the data to the appropriate data registers and then the status bit i.e. the done bit can be polled continuously until the peripheral is found to be done with the task. This is obviously a cumbersome process but works out just fine for less number of peripherals, as the system matures we would need the peripherals to raise an interrupt to the processor core whenever it is finished with the assigned task or if some error has occurred while execution of the task.

4.5 Processor as a peripheral

The AJIT processor wrapper has been designed so to have 2 AXI slave interfaces for debugging and 1 AXI Master interface for memory reads and writes which are done indirectly through the AFB-AXI bridge as explained later. The below figure shows the wrapping done around in order to make it in raw sense AXI interfaceable. As you can see inside the wrapper, AJIT resides in the center and right adjacent to it on the right is

the AFB-AXI Bridge which has been explained in detail later but for time being can be understood just as a bridge between the custom data bus of AJIT i.e. AFB and the AXI interface. To the left of AJIT core are 2 64 bit FIFOs which are supposed to carry the 64 bit number sent from the host which contains debug commands. To even further left of these FIFOs are their controllers which provide an AXI interface control of these FIFOs to the masters on interconnect which includes the host machine(more on this later). These in combination provide AJIT processor an interface ability to the AXI interface and makes AJIT core just another AXI Master peripheral on the AXI interconnect.

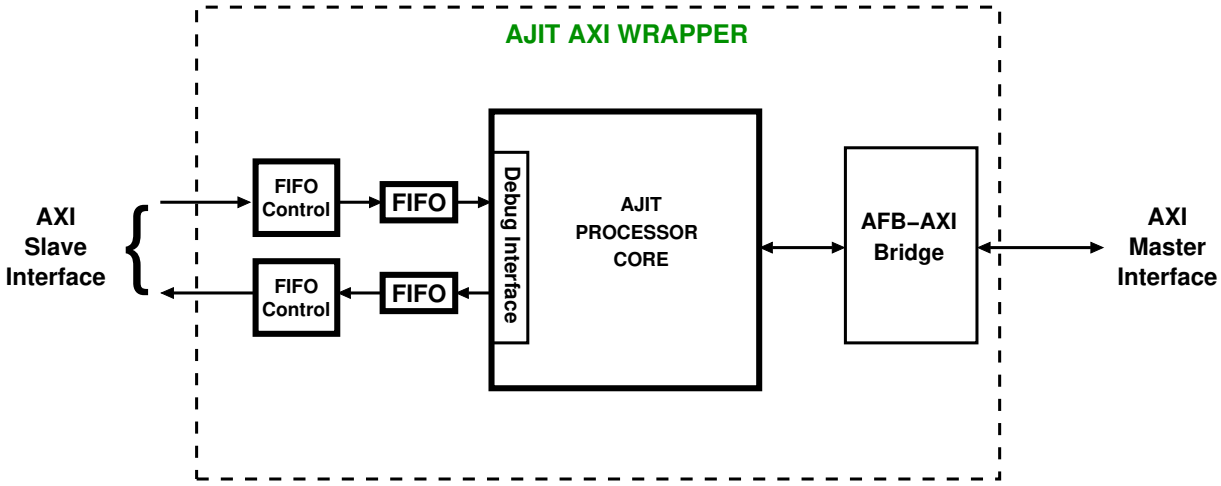
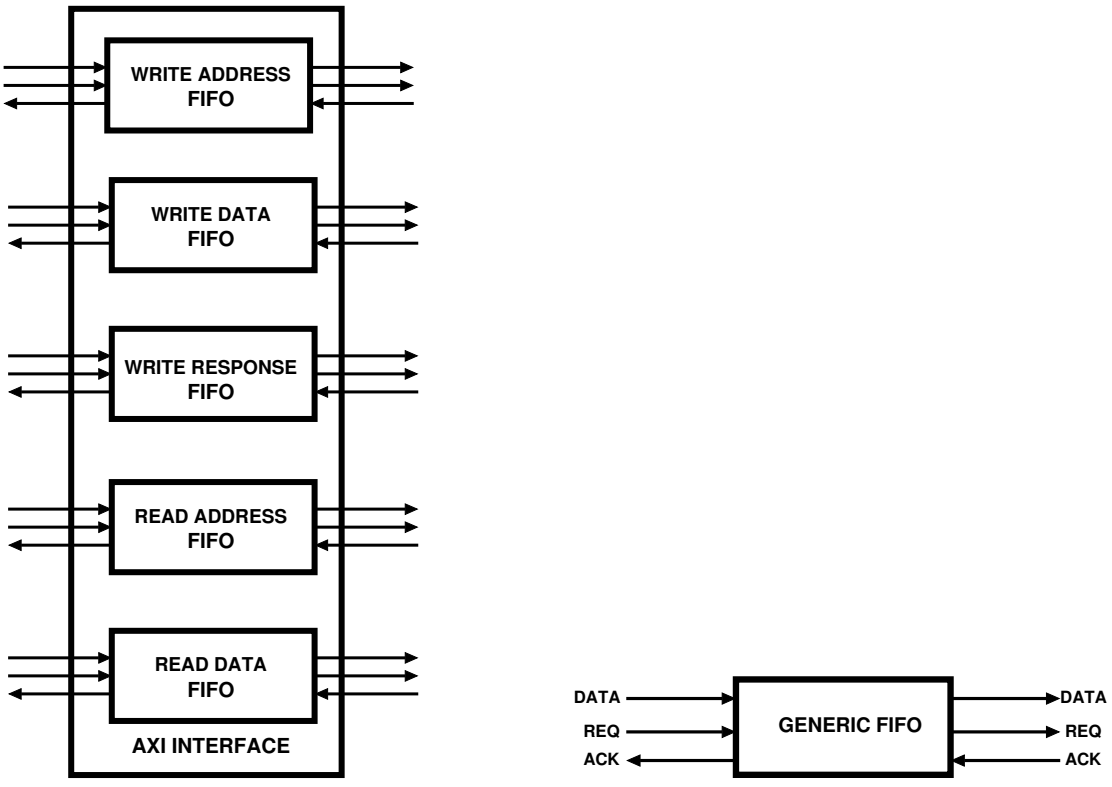


Figure 4.3: AJIT AXI Wrapper

5 Interfaces

5.1 AXI-Lite Interface

This interface operates with 5 FIFOs with different widths namely for Write Address, Write Data, Write Response, Read Address and Read data channel.



(a) AXI-Lite FIFOs (b) FIFO Interface

Figure 5.1: AXI-Lite Interface

5.2 AFB Interface

The master interface for the custom AFB interface is the AFB Request Interface and the corresponding slave interface is the AFB Response Interface.

5.2.1 AFB Response Interface

Port Name	Port Description	Direction	VHDL Data Type
AFB_RESP_DATA	Data Signal	in	std_logic_vector(32 downto 0)
AFB_RESP_READY	Ready Signal	in	std_logic_vector(0 downto 0)
AFB_RESP_ACCEPT	Acceptance Signal	out	std_logic_vector(0 downto 0)

Table 5.1: AFB Response Interface Signals

5.2.2 AFB Request Interface

Port Name	Port Description	Direction	VHDL Data Type
AFB_REQ_DATA	Data Signal	out	std_logic_vector(73 downto 0)
AFB_REQ_READY	Ready Signal	in	std_logic_vector(0 downto 0)
AFB_REQ_ACCEPT	Acceptance Signal	out	std_logic_vector(0 downto 0)

Table 5.2: AFB Response Interface Signals

5.3 Req and Ack Interface Protocol

Req and Ack protocol is a custom protocol defined by Prof.Madhav Desai wherein the environment sends out a `req` signal whenever it has data to send out and the slave responds with an `ack` signal which translates to a green flag to the transaction and the data is exchanged. The `req` and `ack` signals are interchangeable and thus can be asserted by the environment or by the slave. Figure 5.2 shows the timing diagram for the req-ack protocol:-

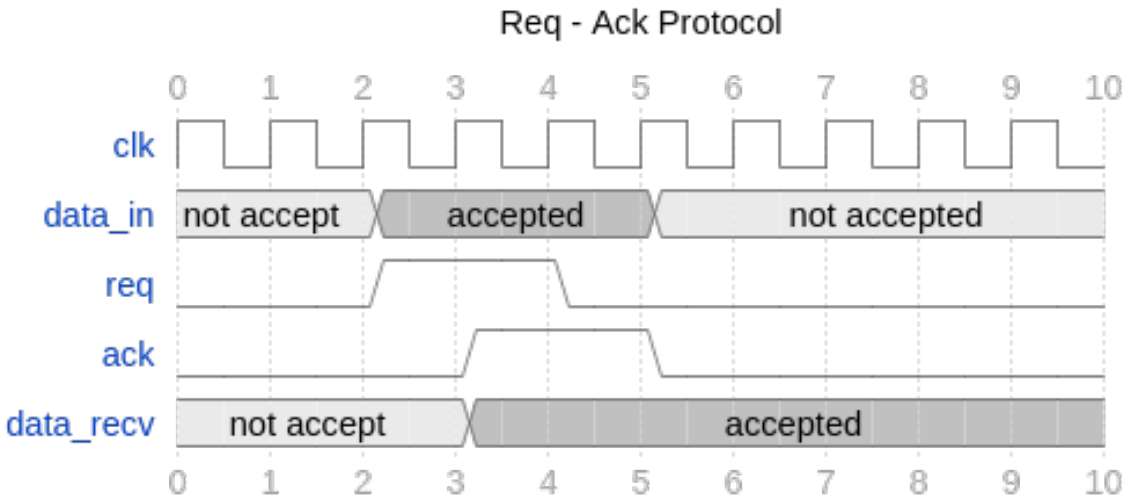


Figure 5.2: Timing diagram of Req-Ack protocol

5.4 HLS Stream Interface

This is a trivial fifo like interface which can be generated through Vivado HLS. It uses `ready` and `accept` signals instead of `req` and `ack`. Figure **??** shows the timing diagram for the ready-accept protocol:-

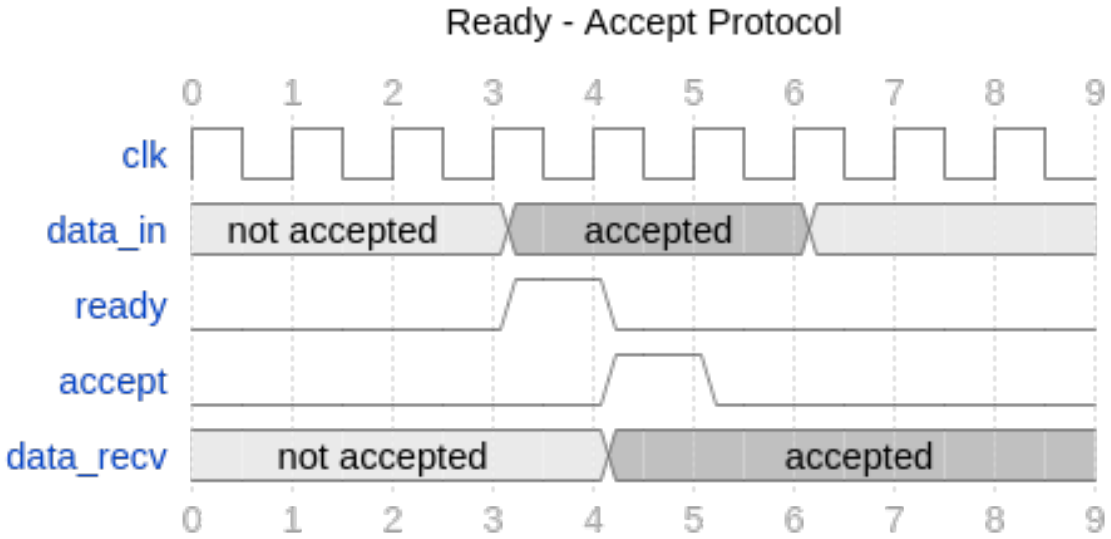


Figure 5.3: Timing diagram of HLS stream interface

6 | Blocks Description

6.1 DRAM Controller

6.1.1 Need for a DRAM interface

In the previous iteration of OS booting on AJIT processor the OS utilized the block RAM of the processor which was of size 4 MB and thus took awhile to boot and only a small scale OS without much driver support(like Network drivers) could be booted, thus a need for a larger DRAM support. Here we head out to utilize the onboard DRAM of the VC709 board based on the ideas of our generic peripheral interface discussed before.

6.1.2 Our implementation

The figure 6.1 below shows our DRAM interface implementation as part of a generic AXI peripheral system. DRAM interfaces with the AXI interconnect with the aid of a Memory Controller specifically employed for DRAMs. Memory controller was generated using the MIG (Memory Interface Generator) utility of Vivado 2017.1.
The support for DRAM size here goes up to 4 GB per slot and since VC709 has two slots we can have a combined support up to 8GB RAM.

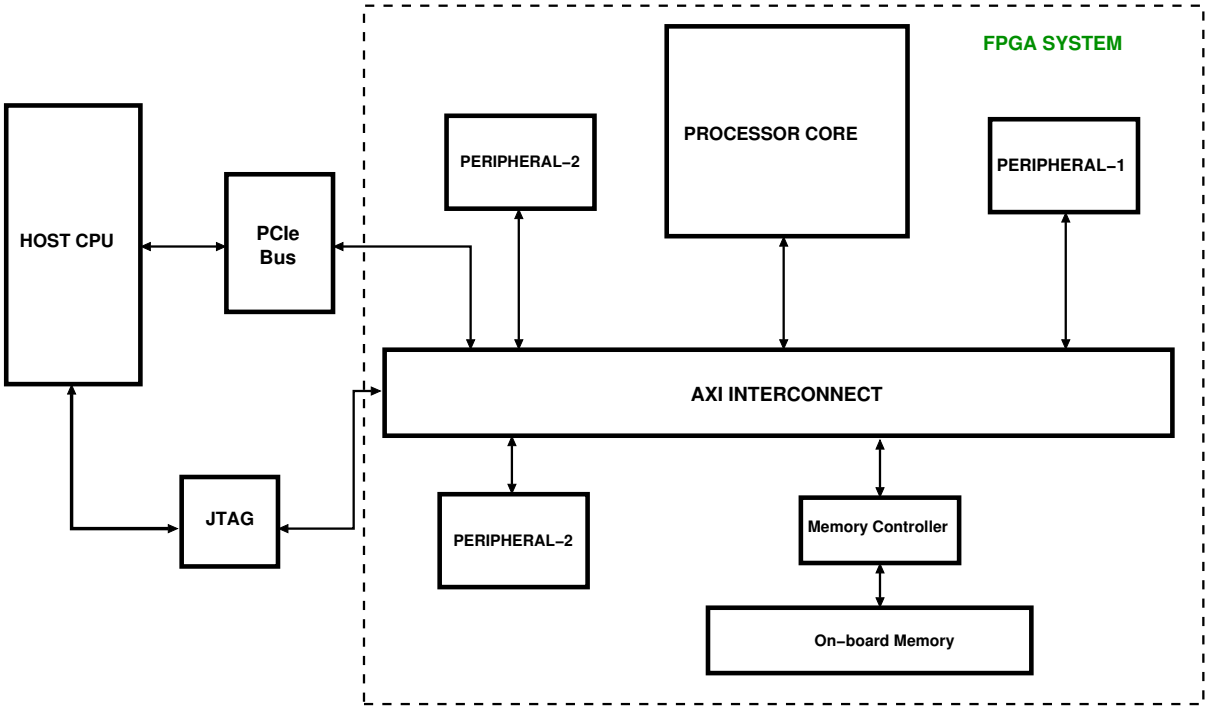


Figure 6.1: DRAM Controller integrated flow

6.1.3 MIG Vs EMC Core

The two cores provided in Vivado are for different types of memory. The following information can be found in the respective user guides which summarise their use cases:

Core	Memory Support
MIG	DDR2, DDR3, QDR-II, RLDRAM
EMC	SRAM, ZBT, and other similar SRAM like interfaces

Table 6.1: Comparison of Memory Cores

6.1.4 Testing of Interface

We tested the interface using the XSCT command line utility which employs the JTAG interface and gives a direct access to all AXI peripherals. Later on we also tested this with a PCI express interface from the Host CPU using the pcimem utility.

6.2 PCIe-AXI IP

6.2.1 Need for a PCIe interface

In the previous iteration of OS booting on AJIT processor the OS utilized the block RAM of the processor which was of size 4 MB and the size of the kernel was just 2 MB and now as we increase the RAM size to accommodate a heavy duty OS we need a faster way to write the image to the storage instead of JTAG and we are just in luck since VC709 has a PCIe interface and can be plugged directly into the PCIe slot of the host CPU motherboard. Right now we lack a flash interface in our system which would have allowed us to have a nonvolatile memory block which would not require to be written at every power on-off cycle of the board. Here we utilize the PCIe interface of the VC709 board.

6.2.2 Basics of PCIe interface

Before diving into the implementation details we need to revisit some essential basics of PCI express peripherals that would prove to be useful here. On a **Linux** machine one can type `lspci -vv` in the console and get a detailed output of the PCIe devices connected to the machine. Its a long list so I will explain with the example of the Xilinx's VC709 board. As can be seen from listing 6.1 the `lspci` log gives out the details about the connected VC709 board's pci interface. Table 6.2 explains the relevant parameters in the `lspci -vv` log.

Listing 6.1: lspci log

```
$ lspci -vv
...
...
0a:00.0 Memory controller: Xilinx Corporation Device 7038
        Subsystem: Xilinx Corporation Device 0007
        Physical Slot: 3
        Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr+ Steppi......
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <M......
        Interrupt: pin A routed to IRQ 16
        Region 0: Memory at f0000000 (32-bit, non-prefetchable) [size=128M]
        Region 1: Memory at e8000000 (32-bit, non-prefetchable) [size=128M]
        Region 2: Memory at e0000000 (32-bit, non-prefetchable) [size=128M]
        Region 3: Memory at d8000000 (32-bit, non-prefetchable) [size=128M]
        Region 4: Memory at d0000000 (32-bit, non-prefetchable) [size=128M]
        Region 5: Memory at c8000000 (32-bit, non-prefetchable) [size=128M]
        Capabilities: <access denied>
        Kernel modules: riffa
```

Parameter	Value
PCIe slot name	0a:00.0
Class	Memory Controller
Device id	7038
Physical slot	3
ProgIf	Region X
Kernel modules option	riffa

Table 6.2: lspci parameter description

X is an integer from 0 to 5

ProgIf : Programming Interface

Kernel Modules: Kernel module reporting that it is capable of handling the device (optional, Linux only) for example here riffa

Device id : This comes in handy when there are mulitple boards connected to the host

6.2.3 BARs and address translation

Whenever reading or writing we proceed with a method of providing a base address and the offset along with it which would amount to the actual address. We operate here with what is knows as BAR(Base Address Register), it is the register which would hold the base/start address of your Memory block which is DRAM here. One can make multiple BARs with different base addresses to access different parts of the memory. Each BAR is set with a base address and an address range which it can access. Each bar created by the user gives rise to a new resource file in the /sys/ directory of the

host machine which as explained later in driver development can be used to access the respective memory locations. The listing 6.2 shows the created resource files inside the `/sys/bus/pci/devices/0000:0a:00.0/` directory on the host machine in this case.

Listing 6.2: Resource files

```
$ ls -l /sys/bus/pci/devices/0000:0a:00.0/
...
...
-r--r--r-- 1 root root      4096 Jun 18 15:30 irq
-rw------- 1 root root 134217728 Jun 18 16:17 resource0
-rw------- 1 root root 134217728 Jun 18 16:17 resource1
-rw------- 1 root root 134217728 Jun 18 16:17 resource2
-rw------- 1 root root 134217728 Jun 18 16:17 resource3
-rw------- 1 root root 134217728 Jun 18 16:17 resource4
-rw------- 1 root root 134217728 Jun 18 16:17 resource5
...
...
```

134217728 *bytes* = 128MB

The `irq` file provides memory mapped support for the interrupts received from the PCIe-AXI Translation IP on the FPGA system. This could be later on used as an alternate to the current polling strategy of the custom PCIe driver as explained later.
The figure 6.2 shows our PCIe-AXI interface implementation as part of a generic AXI peripheral system.

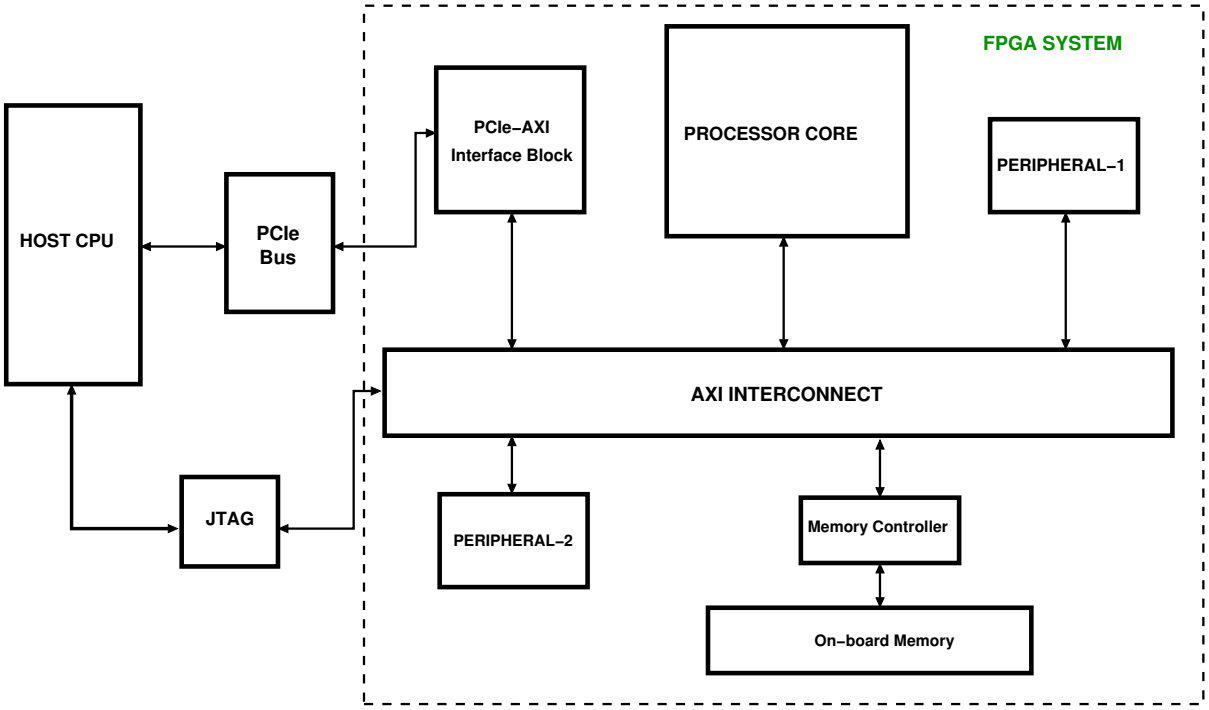


Figure 6.2: PCIe-AXI interface and DRAM integrated design

6.2.4 Testing

We tested this with a PCI express interface from the Host CPU using the pcimem utility. Later on we developed a C driver for creating an API interface which provides helper functions for sending and receiving information through the PCIe-AXI interface to the FPGA model.

6.3 AFB-AXI Bridge

6.3.1 Need for a Custom IP for AFB-AXI Interface

AFB(AJIT FIFO Bus) is an in-house term coined to represent the data bus of AJIT which allows it to talk to other AXI peripherals. AFB is not directly AXI compatible and thus needs a bridge, we developed it from scratch in Verilog keeping in mind the provided AFB specifications.

6.3.2 Our implementation

We generated a custom IP bridge by writing a Verilog file and then packaging an IP through Vivado's IP Manager to make it available in the user repository in Vivado IP repository. The bridge supports AXI-Lite on one side and AFB on the other. As can be seen in figure 6.3 the input port on the left directly interfaces with the AJIT core and then passes on the input AJIT request to the AFB parser block which then splits the request according to the AFB specs as mentioned before and packages it as AXI Master packets and forwards the packets to the corresponding AXI bridge's FIFOs i.e. write address, write data and read address. It also reads the response FIFO and the read data FIFO and packages the AXI FIFO's data back into the AFB response format and sends it back to AJIT core.

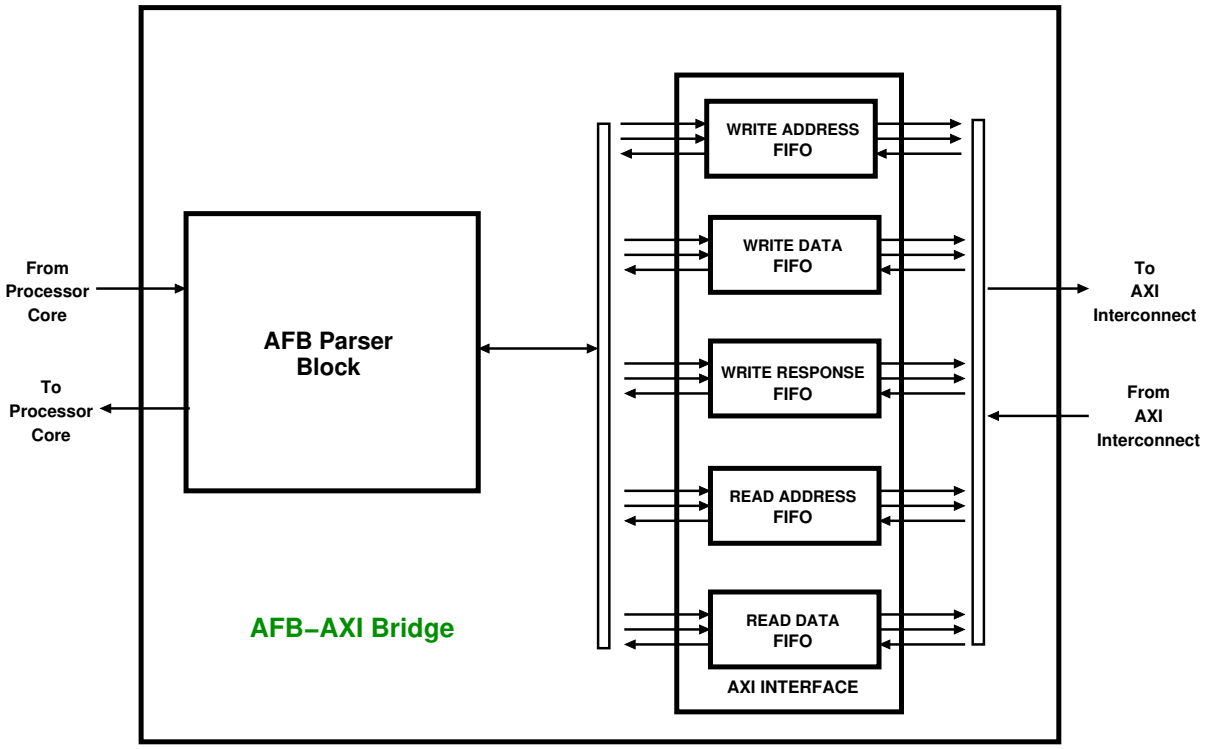


Figure 6.3: AFB-AXI Bridge

6.3.3 Full Flow

The figure below shows the full flow model integrated with the AFB-AXI bridge.

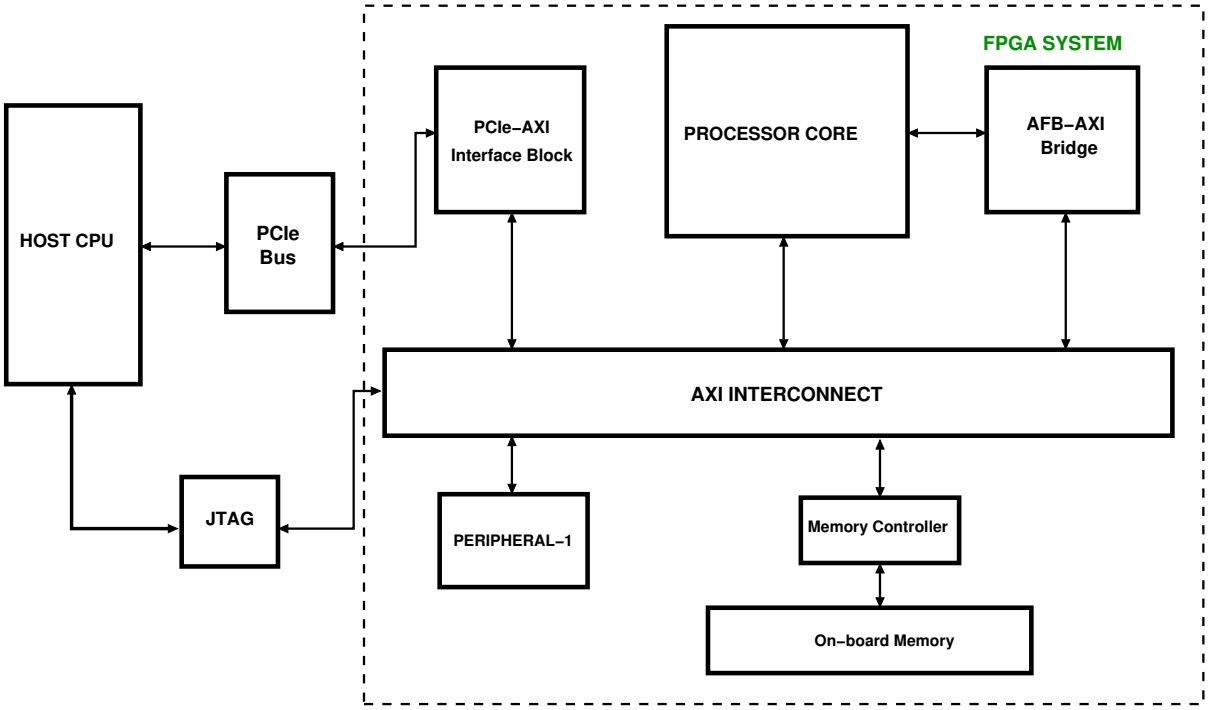


Figure 6.4: Flow with AFB-AXI Bridge

6.3.4 Testing of Interface

We conducted manual memory read-write tests on the interface using the XSCT command line utility which employs the JTAG interface and gives a direct access to all AXI peripherals. Later on we also tested this with a PCI express interface from the Host CPU using the pcimem utility.

6.4 FIFOs

Figure 6.5 shows the generic FIFO interface generated by AHIR which has been used at almost all interfaces in the FPGA system design. The AXI side interface `hls_stream` though uses a similar FIFO but with READY and ACCEPT signals instead of REQ and ACK. The timing diagram for the same has been shown in figure.

Figure 6.5: Generic FIFO Interface

6.5 FIFO Controllers

6.5.1 Need for a Memory Mapped FIFO

The Debug interface to the processor requires two 64-bit FIFOs compatible with the generic AXI peripheral interface. Here by compatibility we mean they need to have a Slave AXI interface in order to establish a bidirectional data flow through the PCIe-AXI interface.

The problem is solved by dividing it into two parts and both of them handling simpler generic tasks. The first block provides the Slave AXI interface as an input and a control interface on the output side to the other block which acts as a data FIFO. A similar block is created to perform the reverse operation as well.

6.5.2 Our implementation

We employ **Vivado HLS**(High Level Synthesis) to generate the FIFO controllers. This enables us to write short code snippets as shown in listing 6.3 to generate the Slave AXI data interface and control signals for the FIFO controller which would read AJIT's response from the FIFO and write it in it's slave register. The cpp template `hls_stream` generates the FIFO compatible interface so that the controller can en-queue and de-queue the debug FIFO. Table 6.3 provides explanation for the important `pragma` statements.

Listing 6.3: FIFO Controller HLS

```
void fifo_to_axi_slave (ap_uint<32> data_out, ..., hls::stream<ap_uint<32> > &in_fifo) {
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE s_axilite port=data_out
    #pragma HLS INTERFACE s_axilite port=data_valid
    if (!in_fifo.empty ()) {
        data_out   = in_fifo.read ();
        data_valid = true;
    }
    else {
        data_valid = false;
    }
}
```

Pragma Statement	Description
HLS INTERFACE s_axilite port return	Generates the control signals (start, done etc.)
HLS INTERFACE s_axilite port data_out	Memory mapped register for read data
HLS INTERFACE s_axilite port data_valid	Memory mapped register for data valid signal

Table 6.3: Pragma Statement Description

6.5.3 Full Flow

The figure 6.6 shows our complete FPGA implementation as a generic AXI peripheral system. AJIT core interfaces with the DRAM controller indirectly through the AFB-AXI bridge which translates its requests to the AXI format, AJIT interfaces to the host machine through the debug interface which is also AXI compatible and is controlled by the host through the PCI-AXI translation bridge which is used to translate PCIe requests to AXI requests but the interesting thing to note is that it can also be controlled by on-board master peripherals for example the processor. The figure 6.7 shows the crucial data flows in the FPGA system.

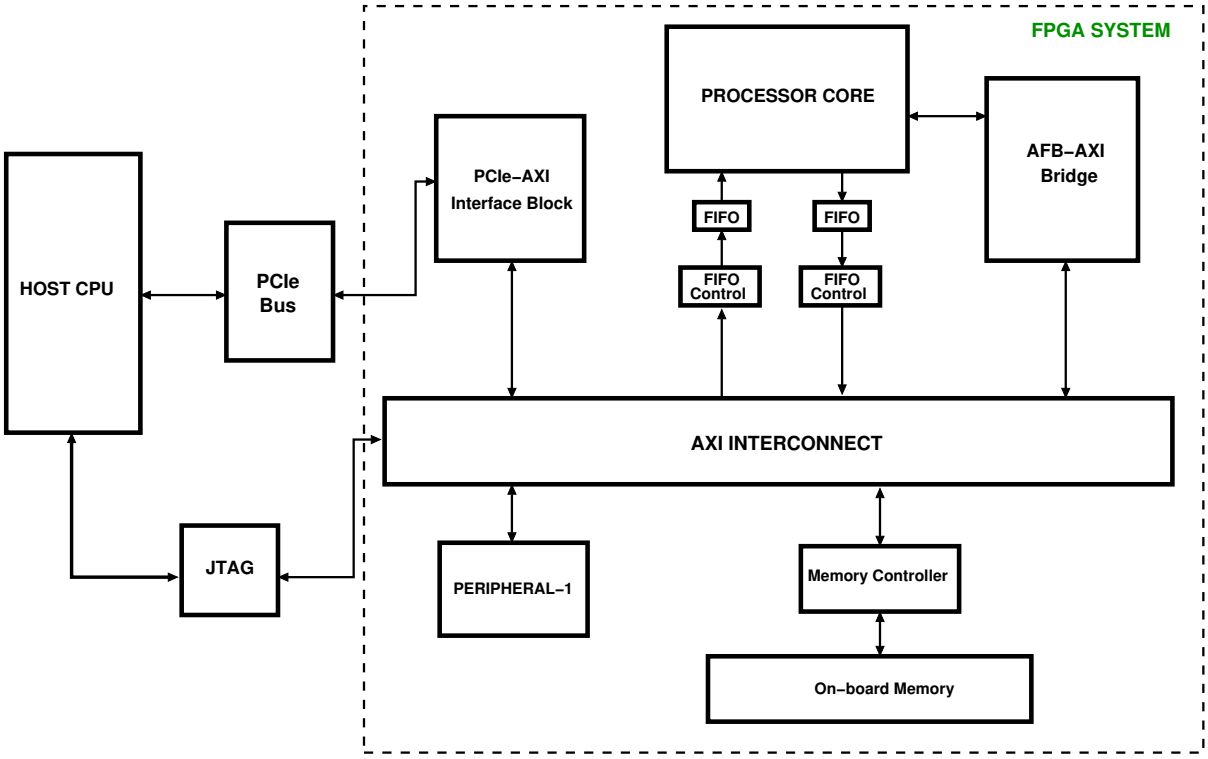


Figure 6.6: Complete FPGA system

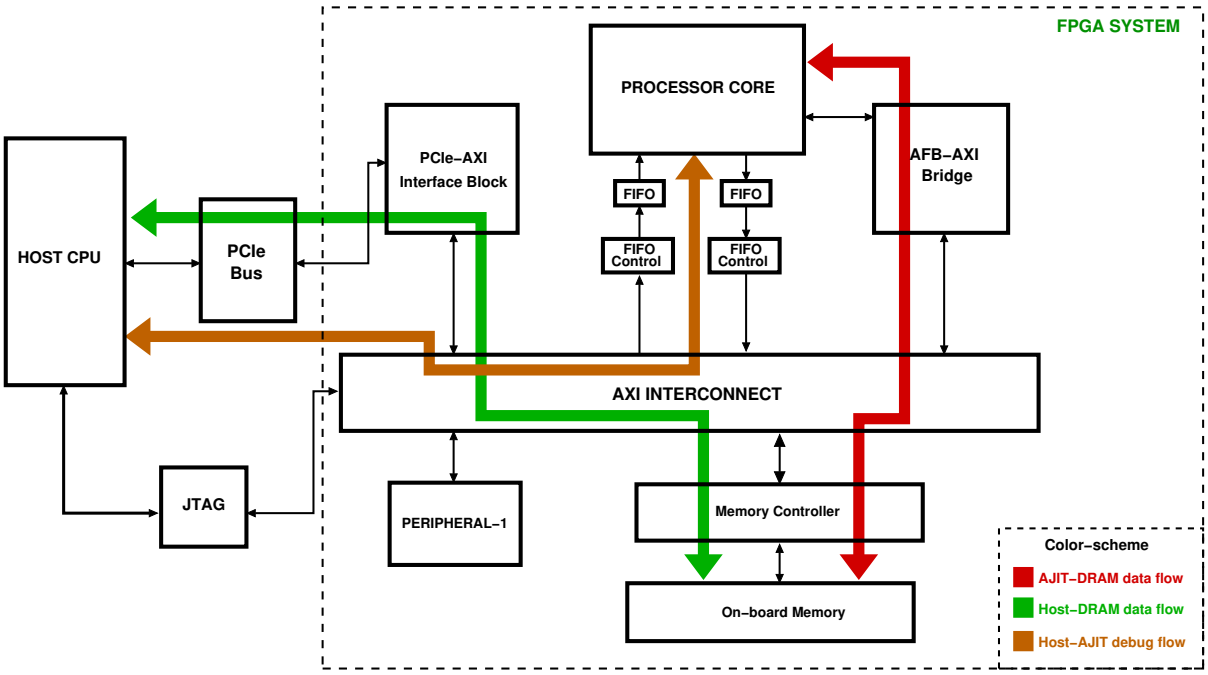


Figure 6.7: Complete FPGA system with data flow

6.5.4 Testing of Interface

We tested the interface using the XSCT command line utility which employs the JTAG interface and gives a direct access to all AXI peripherals, an example of the same is shown in listing 6.4.

Listing 6.4: xsct in action

```
$ xsct
xsct% connect
attempting to launch hw_server

****** Xilinx hw_server v2017.1
  **** Build date : Apr 14 2017-19:01:52
    ** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application

INFO: To connect to this hw_server instance use url: TCP:127.0.0.1:3121

tcfchan#0
xsct% target 3
xsct% mwr 0x80000000 0x01
xsct% mrd 0x80000000
      0x01
```

We also tested this with a PCI express interface from the Host CPU using the pcimem utility, an example of the same is shown in listing 6.5.

Listing 6.5: pcimem in action

```
$ sudo ./pcimem /sys/bus/pci/devices/0000 0a 00.0/resource0 0 w
  /sys/bus/pci/devices/0000\:0a\:00.0/resource0 opened.
  Target offset is 0x0, page size is 4096
  mmap(0, 134217728, 0x3, 0x1, 3, 0x0)
  PCI Memory mapped to address 0x4801f000.
  Value at offset 0x0 (0x4801f000): 0xC0BE0100
```

7 | Software Interface

7.1 Driver Interface

7.1.1 Introduction

The debug interface to the processor is a crucial part in development of applications as well as the development of the processor itself. Here we attempt to provide a seamless debug interface which is easily extendible and integrable to the FPGA system and new processor cores. The way the host machine communicates with the processor is by periodically sending and receiving packets of length 64-bits over the PCIe bus of the host machine to the AXI interconnect on the FPGA system. The following figure shows the debug interface of the processor:-

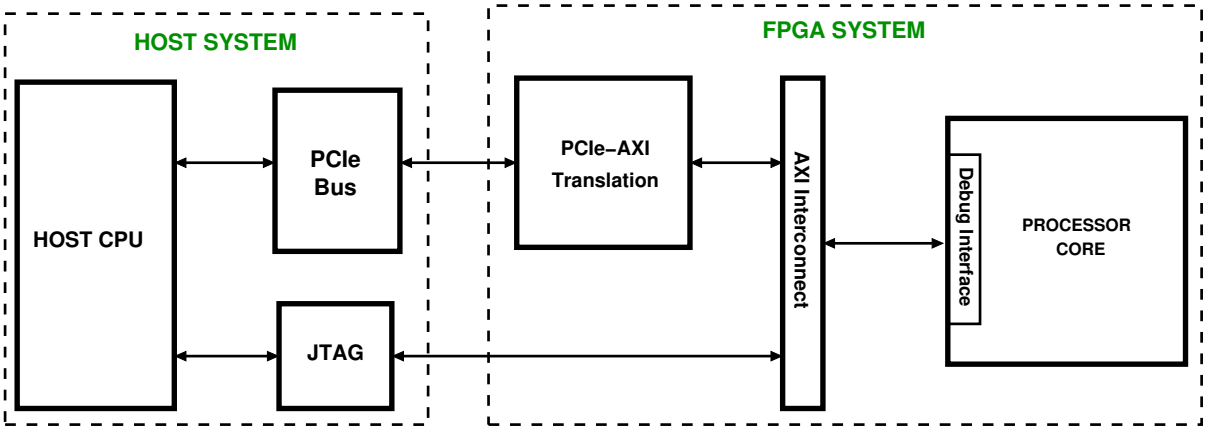


Figure 7.1: Debug channel from host to AJIT

7.1.2 Testbench for the Driver Interface

In order to test the FPGA system we run a software testbench on the host machine which controls the processor's mode and reads and writes the on board memory. The driver API provides lower level function calls to the testbench in order to execute fundamental tasks on the FPGA system. The testbench builds on this driver and contains a multi-threaded environment with several threads controlling their respective software pipes which are then collected by the main thread to produce a 64-bit packet which contains the relevant Debug instruction for the processor to execute. The driver API also aids development of userspace applications which need access to lower level FPGA system routines.

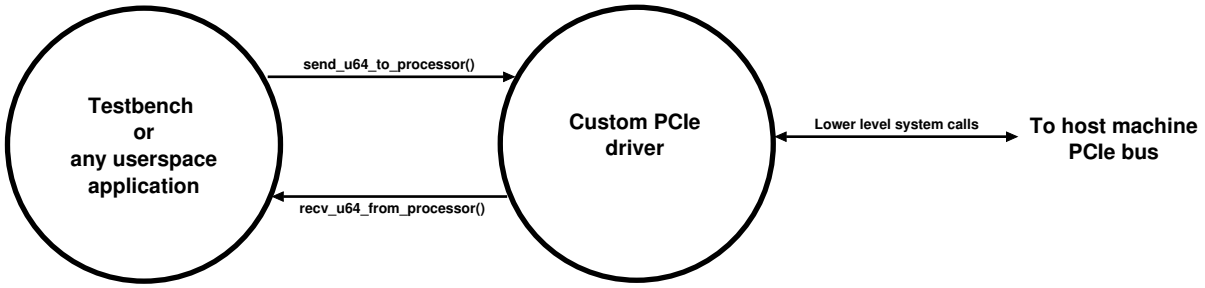


Figure 7.2: Driver API in action

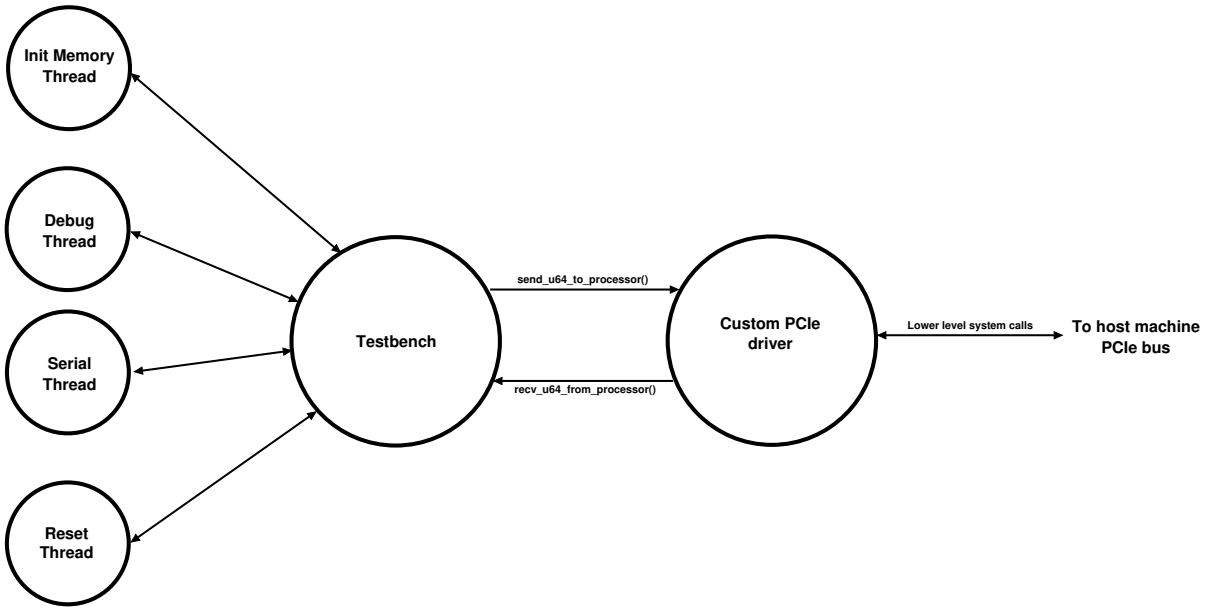


Figure 7.3: Software Testbench threads

7.2 Driver Interface discussion

For the previous iteration of already developed software programs for AJIT to be compatible with the new FPGA model we need to keep the identical call names to the API calls and need to have a Driver API interface which provides the same functions but with non-trivial and different inherent implementations which is the multiple bars of 128MB one.

7.3 Driver API

The four black boxed functions that the API provides are:

Listing 7.1: Driver API

```
1  void* initialize_axi_link(char* name){
2  ...
3  }
4
5  int close_axi_link(void* fpga_device){
6  ...
7  }
8
9  int write_to_axi_dram(void* fpga_device, uint32_t address, uint32_t write_word){
10 ...
11 }
12
13 int read_from_axi_dram(void* fpga_device, uint32_t address, uint32_t *read_word){
14 ...
15 }
16
17 int recv_u64_from_processor(uint64_t* r_word){
18 ...
19 }
20
21 int send_u64_to_processor(uint64_t* s_word){
22 ...
23 }
```

7.4 Driver development discussion

Driver infrastructure can be broken down in four simple parts:

`initialize_axi_link(...)`

As would be evident by the name this function, it initialize the axi link to a fpga device and returns the status of the opened device and prints error message if it fails to open the device. It initializes an internal data structure specifically created to manipulate fpga devices, allocates memory and returns a pointer to the created FPGA device.

`close_axi_link(...)`

As would be evident by the name this function closes an already open fpga device and returns the status of the device and prints error message if it fails to close the device. It frees the allocated memory for the internal data structure specifically created to manipulate fpga devices, it returns a `0` on successful freeing and hence closing of the connection FPGA.

`write_to_axi_dram(...)`

This function provides the user with the functionality to write 32-bit words directly to the on-board DRAM by specifying the actual physical DRAM address and the respective 32-bit word to be written. This will automatically find out the appropriate bar to be mapped to the host memory and would write the word to the mmapped file and returns `0` on a successful write.

`read_from_axi_dram(...)`

This function provides the user with the functionality to read 32-bit words directly from the on-board DRAM by specifying the actual physical DRAM address and the respective 32-bit word pointer to be stored with the read value. This will automatically find out the appropriate bar to be mapped to the host memory and would read the word from the mmapped file and returns `0` on a successful read.

`send_u64_to_processor(...)`

This function is used to send a 64-bit word to the processor core. The approach here is to memory map the base address of the FIFO controllers and providing the required offset to the data registers. For memory mapping we use `mmap` with `PROT_READ` and `PROT_WRITE` flags set so that we have the read-write access. We in effect get a pointer to the memory location of the data registers which we can dereference to write our data which has been split into nibbles. This function after writing the data registers, polls the success bit in the control register and as soon as it is set, the function returns the success status i.e. a `0`.

`recv_u64_from_processor(...)`

This function is used to receive a 64-bit word from the processor core. The approach again here is to memory map the base address of the FIFO controllers and providing the required offset to the data registers. For memory mapping we use `mmap` with `PROT_READ` and `PROT_WRITE` flags set so that we have the read-write access. This function before reading the data registers, polls the valid bit in the control register and as soon as it is set the function reads the data in two chunks by reading the higher nibble and lower nibble separately and then combines them to form the read 64-bit value.

7.5 Explaining polling method

The way in we perform a transfer of a 64-bit number is that we split it into two 32-bit numbers and write it out to 32 -bit memory mapped registers inside the FIFO controllers from where the data is sent to the Debug FIFOs. For simplicity the driver currently continuously polls the control register of the FIFO controllers and checks for the status bit to set and finishes the API call as soon it sets and returns the received data/status

dependent upon the API call made.

7.6 Need for interrupt based support in future

To avoid the redundant polling approach we are going to introduce an interrupt based driver API which would would contain ISR routines responding to the PCIe-AXI interface generated interrupts. For this functionality we most probably would need to have a Kernel Space Module which would have direct access to Kernel ISR routines and would allow us to respond to specifically VC709 generated interrupts on the PCIe bus in an efficient manner but this can in principle also be executed through user domain applications by registering the IRQ base address as the interrupt base and writing an interrupt handler function and register it as the handler to this particular interrupt

7.7 Testing of the driver

Preliminary tests were conducted using pcimem utility with successful results and extensive tests on the driver have been conducted using a multithreaded software testbench which periodically sends and receives 64-bit packets from AJIT core.

> A strong reference for this driver was a github project by the developer `billfarrow` where he has provided the basis to read & and write to pci devices from a userspace domain.

8 | Performance of the System

Timing Summary

In the following table we report the timing summary of the synthesized and implemented design of the FPGA System on VC709 operating at 100MHz.

Parameter	Description	Tolerance	Value Obtained
WNS	Worst Negative Slack	1-2 percent of 10ns	-0.1ns
CP	Critical Path	N/A	Exists within Xilinx Smartconnect

Table 8.1: Timing Summary

Utilization Summary

In table 8.2 we report the utilization summary of the synthesized and implemented design of different variants of the FPGA System on VC709.

Design	Hardware Usage
AJIT FPGA System	116024
AJIT and Microblaze	124398

Table 8.2: Utilization Summary

Hardware Usage : Number of Flip Flops in the design.

Microblaze : Used for testing the system peripherals without AJIT.

Host to FPGA system

Writing speed of PCIe

Writing speed of PCIe with 6 blocks of 128MB each after preliminary tests was found to be around 400 MB/s which is significantly higher than the required speed in the current situation and also when compared to the past FPGA system. This speed is really helpful when we need to write huge amount of data to memory for example to write the image of the custom compiled operating system for AJIT.

Reading speed of PCIe

Reading speed of PCIe with 6 blocks of 128MB each after preliminary tests was found to be around 10 MB/s which is significantly higher than the required speed in the current situation and also when compared to the past FPGA system.

9 | Tests conducted

9.1 Indirect filling of on-board DRAM

To check access of the host to the full 4GB on board DRAM Memory through a Memory Address Translator Block which Translates a read/write request in form of two AXI slave register writes to an AXI Master request on the onboard AXI interconnect. It employs 3 registers to accommodate address, data and type of request respectively. Figure 9.1 shows the flow for this indirect access to the on-board DRAM through the Memory Address Translation block. More details have been provided in section 10.2 regarding this block.

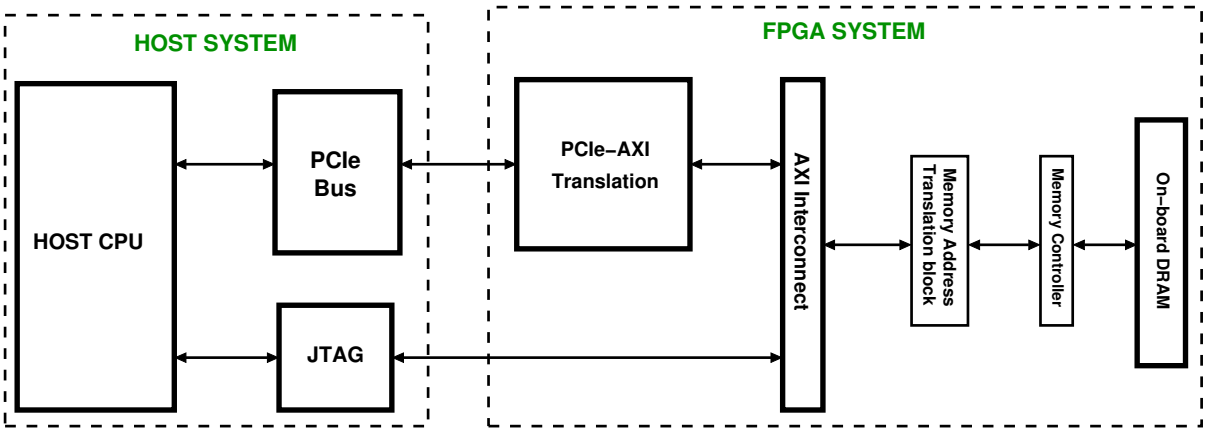


Figure 9.1: Indirect memory channel from host to AJIT

9.1.1 Results

In order to write the whole 4GB memory through this block took around 45 minutes since in order to perform a single write to the on board DRAM through this peripheral we need to perform 3 writes to this block including write address, write data and the

type of request as write into the AXI Slave registers of this block.

9.2 March test on memory

9.2.1 What is March Test?

A march test consists of a finite sequence of march elements. A march element is A finite sequence of Read and/or Write operations applied to every cell in memory in either increasing address order (cell 0 to cell n-1) or decreasing address order (cell n-1 to cell 0). All operations of a march element are done before proceeding to the next address. The march tests are a preferred method for RAM testing-Linear complexity, regularity, and symmetry.

9.2.2 Fault Models

Fault Models	Description
Stuck-At Fault	The logic value of a line is always **0 or 1**
Transition Fault	A cell or a line that fails to undergo a $0 \rightarrow 1$ or a $1 \rightarrow 0$ transition
Coupling Fault	A write operation to one cell changes the content of a second cell other cells in the memory
NPS Fault	The content of a cell, or the ability to change its content, is influenced by the contents of some
Address Decodes Fault	With a certain address no cell will be accessed, a certain cell is never accessed, with a certain address multiple cells are accessed simultaneously, and a certain cell can be accessed by multiple addresses

Table 9.1: Fault Models

NPS: Neighbourhood Pattern Sensitive

9.2.3 Our Implementation

Here we perform the standard March test on the on board DRAM memory by accessing it from the host using the 6 PCIe BARS created as PCIe resources on the /sys path of

Linux. We implement the **March X** algorithm because it is an $O(n)$ time algorithm and covers most of the basic faults.

Algorithm 1 March X algorithm

1: Step 1: **write** 0 with up addressing order;
2: Step 2: **read** 0 and **write** 1 with up addressing order;
3: Step 3: **read** 1 and **write** 0 with down addressing order;
4: Step 4: **read** 0 with down addressing order;

9.2.4 Faults covered

The above algorithm covers Address Decoder Faults, Stuck at Faults, Transition Faults and some coupling faults.

9.2.5 Timing Results

The below table consists of the time taken by the host to conduct the March test on each bar of the on-board memory.

BAR ID	Time Taken(in sec)
BAR0	59.658342534
BAR1	59.640203597
BAR2	59.607970220
BAR3	59.611384216
BAR4	59.690874289
BAR5	59.722889924

Table 9.2: Timing Results

10 | Alternate Designs

10.1 Single bar of 4GB

The first approach in an attempt to interface the full range of on-board DRAM i.e. 4GB to the host machine was made in a straight forward fashion by keeping a single bar of size 4GB the PCIe interface and it was expected that then by memory mapping by the thus generated `/sys/resourceX` files(where `X` is an integer) generated by the PCIe-AXI IP would give the whole access to whole DRAM. But even on repeated attempts of sucessful hardware desgins we could not make the PCIe interface to generate a resource file of size 4GB. At first this was suspected as an operating system issue but after further digging we found that the usual devices on PCIe bus of this machine had memory regions of the scale 16MB, 32MB going as far as 64MB. Hence we brought down our bar sizes to 128MB after some trial and error we found by experimentation that the motherboard had a limit on the BAR sizes of 128MB.We tried multiple BARS of smaller size i.e. 4K, 32MB etc until we arrived at 128MB limit. This happened out of necessity and lack of open documentation over the internet for the host motherboard.

10.2 Memory Address Translator block

To check access of the host to the full 4GB on board DRAM Memory through a Memory Address Translator Block which Translates a read/write request in form of two AXI slave register writes to an AXI Master request on the onboard AXI interconnect. It employs 3 registers to accommodate address, data and type of request respectively. The listing 10.1 provides the HLS design code for this block and employs similar pragma statements as explained before.

Listing 10.1: Memory Address Translator HLS

```
#include "ap_int.h"
#include "hls_stream.h"

void Memory_Address_Translator(int data_in,int address,int* translated_base,...,int *data_out)
{
        #pragma HLS INTERFACE m_axi port=translated_base offset=none
        #pragma HLS INTERFACE s_axilite port=data_in
        #pragma HLS INTERFACE s_axilite port=address
        #pragma HLS INTERFACE s_axilite port=read_write_in
        #pragma HLS INTERFACE s_axilite port=data_out
        #pragma HLS INTERFACE s_axilite port=return

        // read_write_in is 1 for write and 0 for read

        if(read_write_in)
        {
                translated_base[address/4] = data_in;
        }
        else
        {
                *data_out = translated_base[address/4];
        }
}
```

10.3 Memory Copier Block

This block serves as a small DMA, it will copy a block of 128MB of data from one address in memory into another. The 32-bit source and destination addresses and the read/write type request can be written into AXI Slave registers through the host machine over PCIe interface. It translates the required addresses and the type of request to an AXI Master request on the interconnect. This block was created as an alternative way to access the whole on-board DRAM from the host machine when the host motherboard was found to have a limit of 128MB on the size of `/sys/resourceX` files(where `X` is an integer)

generated by the PCIe-AXI IP. The listing 10.2 provides the HLS design code for this block and employs similar pragma statements as explained before.

Listing 10.2: Memory Copier Block HLS

```
#include <ap_int.h>
#include <hls_stream.h>
#include <string.h>

void hls_dma (char *pSrc, char *pDest) {
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE m_axi port=pSrc offset=slave
    #pragma HLS INTERFACE m_axi port=pDest offset=slave
    for (int i = 0; i < 1024*1024*128; i++) {
        pDest[i] = pSrc[i];
    }
}
```

11 | Current and expected problems

11.1 Expected Problems

11.1.1 Read speed over PCIe

The current data reading speed offered over PCIe by the test setup is around 10MB/s which is more than enough required by the debugging interface of AJIT processor which is around 40 KB/s. But this is considerably less when compared to the write speed offered over the same bus which is around 400 MB/s. This could be a bottleneck in the future if we higher speeds are required over the debug interface of AJIT processor.

11.2 Addressed Problems

11.2.1 Motherboard memory space limit for PCIe bars

The first approach in an attempt to interface the full range of on-board DRAM i.e. 4GB to the host machine was made in a straight forward fashion by keeping a single bar of size 4GB the PCIe interface and it was expected that then by memory mapping by the thus generated `/sys/resourceX` files(where `X` is an integer) generated by the PCIe-AXI IP would give the whole access to whole DRAM. But even on repeated attempts of successful hardware designs we could not make the PCIe interface to generate a resource file of size 4GB. At first this was suspected as an operating system issue but after further digging we found that the usual devices on PCIe bus of this machine had memory regions of the scale 16MB, 32MB going as far as 64MB. Hence we brought down our bar sizes

to 128MB after some trial and error we found by experimentation that the motherboard had a limit on the BAR sizes of 128MB.We tried multiple BARS of smaller size i.e. 4K, 32MB etc until we arrived at 128MB limit. This happened out of necessity and lack of open documentation over the internet. For example on two different systems in EE department in IIT Bombay we found first machine's motherboard has a limit of 128MB bar size whereas another machine has a limit of 256MB bar size. This makes the whole FPGA design scalable in terms of memory size and we can have the access to the whole 4GB memory on a recent versioned motherboard.

11.2.1.1 Vivado's random errors and resolving it by redundant actions

Updation command offered by Vivado when run on project sources messes up sometimes to update the rebuilt custom IPs and shows that those IPs have been locked and sometimes not found. This is mostly solved by checking permissions for the IP files or by repackaging the custom IPs or by reloading the whole project.

Part II

Convolution Engine

12 | Introduction

Background

Now we move on to development of a standalone Convolution Core which was be designed using industry standard Vivado HLS tools and also AHIR HLS tools which is a set of in-house tools developed at IIT Bombay by Prof. Madhav Desai. The goal associated with the development of this core was to speed up the computationally expensive task of convolution of two matrices. It divides the incoming computation among the internal coprocessors and fires them up in parallel to make the computation faster.
This core has a FIFO like interface and is capable of pumping out the resultant matrix elements as soon as they have been calculated so that the write back can begin.

Need for a HPC peripheral

These family of cores are useful in various contexts specially Machine Learning and Image Processing algorithms. Development boards which have a FPGA interfaced with a processor can utilize these kinds of peripherals and allow the programmer to have a simple interface to offload high computation tasks to codecs on the adjacent FPGA.

13 | Design of the Engine

Specifications

Following is the specification for the current storage of image and kernel data:-

- 8 bits per pixel of image data and also kernel data.
- Image would be of size 1024 x 1024 pixels (Basically 1022x1022 with zero padding).
- Kernel would be of size 3 x 3.

Following is the specification for the current convolution core:-

- An AXI Slave interface as the input interface for the addresses for the fetcher.
- The number of coprocessors are kept variable to compare performance and hardware resource consumption.
- A FIFO like interface for the fetcher to interface with the convolution core.
- An AXI Master port for the fetcher for performing write backs to the memory.

13.1 Storage design for image data

The data is stored row wise as shown in the figure below. This strategy was opted to make the fetch and write back protocol really straight-forward. The core gets the address of the first element of the first row of the image matrix, and of the kernel matrix and also the physical address of the write back location in the DRAM.

13.2 Fetch policy for the engine

The fetch policy for the core is includes fetching the kernel data first and sending it out to all the coprocessors inside the core. After this the fetching of image data begins and here we play a little smart and fetch 4 bytes of data in every memory read call and split the data within the core to segregate the individual pixel data.

13.3 Number of coprocessors

Since the image is chosen to be of size 1024x1024 it seems really convenient to have a number of coprocessors which is a factor of 1024. To keep it symmetric we have kept the number of coprocessors to be 1, 2, 4, 8, 16 and 32 so that a single model of coprocessor can be replicated in an convenient manner. Each coprocessor is fed with x rows of image data and the coprocessors are started in parallel, here x is dependent on the number of operational coprocessors in the design.

13.4 Flow design

Figure 13.1 represents the internal flow of the generic convolution engine which employs a variable number of cores since we will utilize this to employ this design to test the performance and hardware consumption on a range of number of coprocessors. As can be seen the core has a fetch unit which is responsible for fetching the incoming data from the input pipe to the core and distributing the work between all the operational cores one by one and then collecting back the results from each of their output pipes and writing it back to the output pipe of the core. Each coprocessor is designed to convolve the incoming image data with an initially stored kernel and write back the result to its respective output pipe. The coprocessor model is discussed later on and we also dive down into the internal design of the coprocessor when we discuss about improving its design efficiency and throughput.

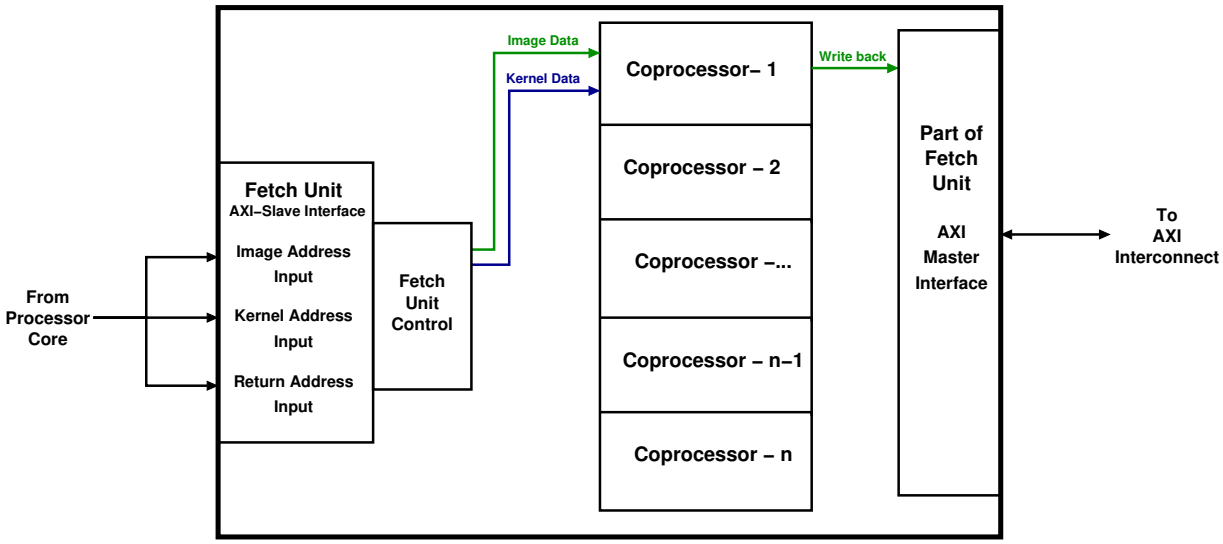


Figure 13.1: Core Design

Figure 13.2 shows a sample system which could be built to integrate the convolution engine with the AJIT FPGA system. The convolution engine can be controlled by AJIT and can also be directly tinkered with from the host machine through the PCIe-AXI block. The engine can be assigned a task to perform a convolution from a software application by first storing the relevant data in DRAM by `write_to_axi_dram` call from the driver API and then the results can be read back using the `read_from_axi_dram` call.

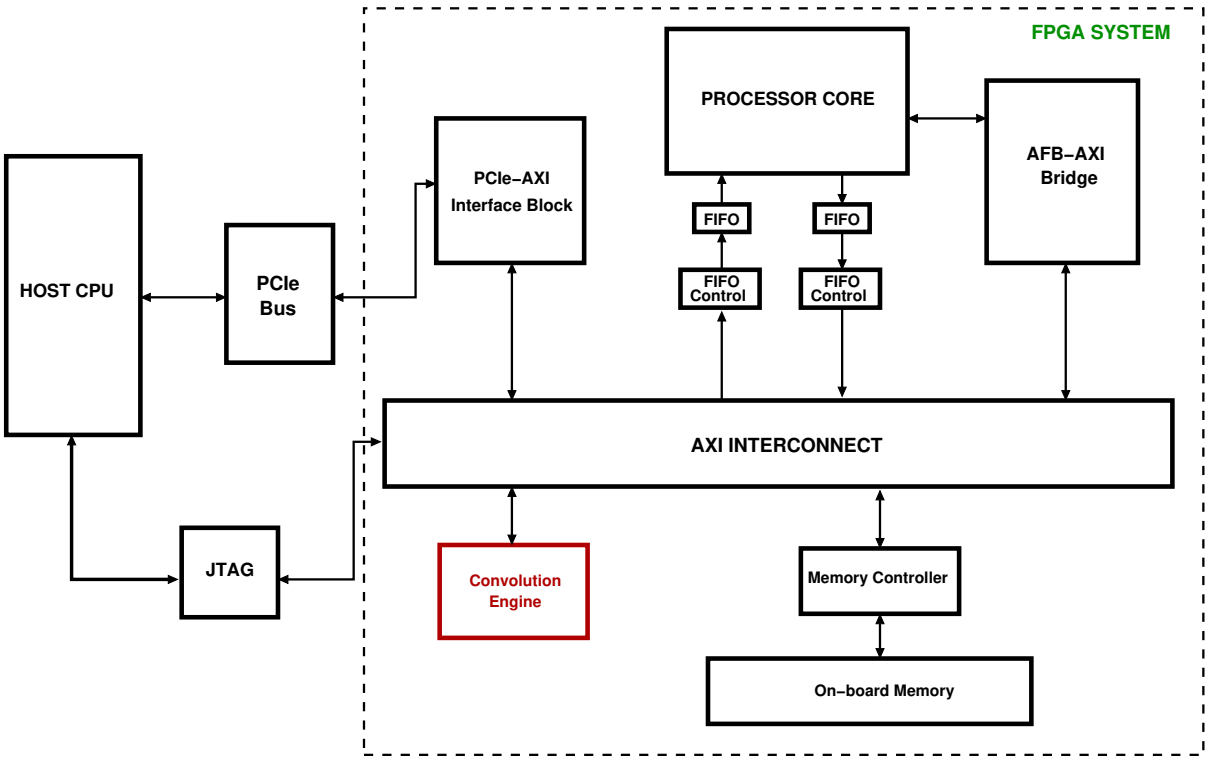


Figure 13.2: Complete FPGA system with data flow

14 | Implementation

14.1 Vivado HLS

Listing 14.1 shows the convolution engine design which would convolve a 1024×1024 image with a 3×3 kernel. This is a functionally correct but an unoptimized design for the convolution engine of this size.

Listing 14.1: Core Vivado HLS

```
#include "ap_int.h"
#include "hls_stream.h"

void core(char* init_row, char* kernel_data_row, char* row_dest) {
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE m_axi port=init_row offset=slave
    #pragma HLS INTERFACE m_axi port=kernel_data_row offset=slave
    #pragma HLS INTERFACE m_axi port=row_dest offset=slave

    // assume an image size of 3x3 for now and the image is already padded with zeros.
    // assuming a kernel of size 3x3 which is symmetric or already flipped stored at
    // the kernel storage location.

    // init_row points to a 5x5 image matrix in hardware.

    char i,j;
    char index=0;
    char acc=0;
    char rows = 3;
    char cols = 3;

    for(i=0;i<rows;i+=rows)
    {
        for(j=0;j<cols;j++)
        {
                acc+= kernel_data_row[0]*(*(init_row+i+j));
                acc+= kernel_data_row[1]*(*(init_row+i+j+1));
                acc+= kernel_data_row[2]*(*(init_row+i+j+2));
                acc+= kernel_data_row[3]*(*(init_row+i+rows+j));
                acc+= kernel_data_row[4]*(*(init_row+i+rows+j+1));
                acc+= kernel_data_row[5]*(*(init_row+i+rows+j+2));
                acc+= kernel_data_row[6]*(*(init_row+i+2*rows+j));
                acc+= kernel_data_row[7]*(*(init_row+i+2*rows+j+1));
                acc+= kernel_data_row[8]*(*(init_row+i+2*rows+j+2));
                row_dest[index] = acc;
                index++;
        }
    }
}
```

14.2 AHIR HLS

14.2.1 C to VHDL

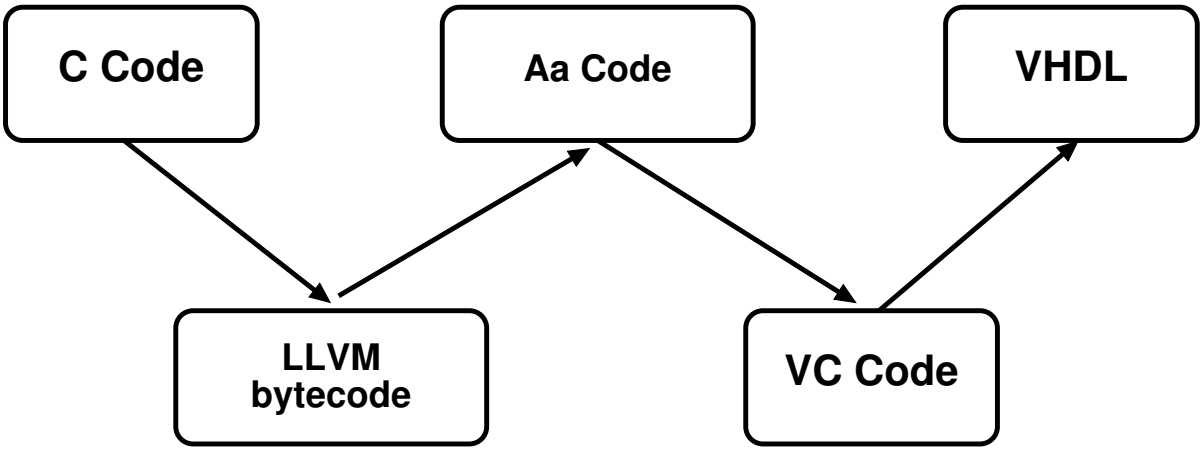


Figure 14.1: C to VHDL design flow

VC : Virtual circuit format

Aa : Aa language format(explained below)

14.2.2 Aa to VHDL

Aa is a programming language for the description of algorithms, with the ultimate aim being the conversion of these algorithms into a hardware implementation. The control-flow in an Aa program corresponds to a petri net of a specific class. The data-flow is constructed using static single-assignment variables (which can be assigned to only once), storage objects, messaging pipes and constants. Pointers to storage objects can be created and manipulated (pointer arithmetic, dereferencing etc.) in the usual way. A program in Aa can also be viewed as a description of a system which reacts with its environment through input and output ports. Thus, an Aa program can either be executed on a computer, or be mapped to a logic circuit.

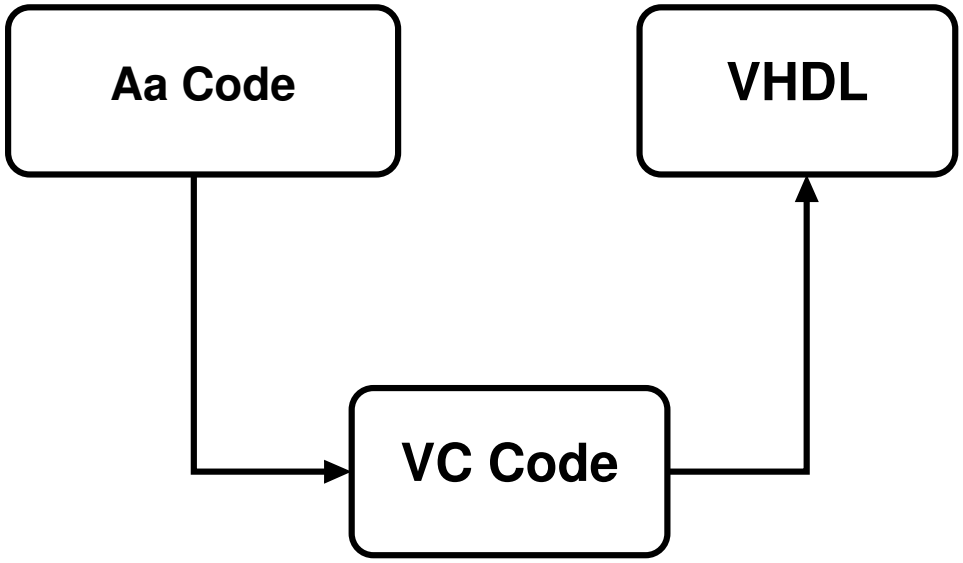


Figure 14.2: Aa to VHDL design flow

14.3 Testing Setup

14.3.1 AHIR Testbenches

14.3.1.1 Software Test

This method of testing the design works only for the C to VHDL design flow. We have a common C testbench file between software and hardware testing. This testbench when compiled for software testing runs different system units as multiple threads which communicate through software pipes.

14.3.1.2 Hardware Test

This method of testing the design works for the C to VHDL design flow as well as Aa to VHDL design flow. We have a common C testbench file between software and hardware testing. This testbench when compiled for hardware testing generates to two executables named `testbench_hw` and `ahir_system_test_bench` which can be run in different shells one after another to start a ghdl simulation enabled by communicating through sockets.

15 | Hardware Usage comparison

15.1 Background

Here we compare the hardware resources required by the HDL design generated by Vivado HLS and Aa HLS implementation of the same core. Also within the Aa HLS implementation we compare the hardware resources required by the Pipelined version of the core with the Non-pipelined version of the core by varying the number of coprocessors utilized in development of the internal architecture of the core. The default pipeline depth is currently kept at 7 to produce the following values and plots.

15.2 AHIR HLS Implementation

Here we produce the results from the synthesis and implementation of different variants of our convolution engine where we consider the number of Flip Flops in the implemented design as our metric to measure the hardware usage of each variant. We produce pipelined and non-pipelined variants for a given number of coprocessors through two methods namely C HLS and Aa HLS(described in previous chapter). Through this process we aim to show the advantage of generating a hardware system through Aa HLS instead of C HLS.The board chosen for this process was Xilinx's VC709.

15.2.1 C to VHDL

Non-pipelined Version

Number of Cores	FFs
1	2219
2	4643
4	9117
8	18210
16	36781
32	74897

Table 15.1: Hardware Resources

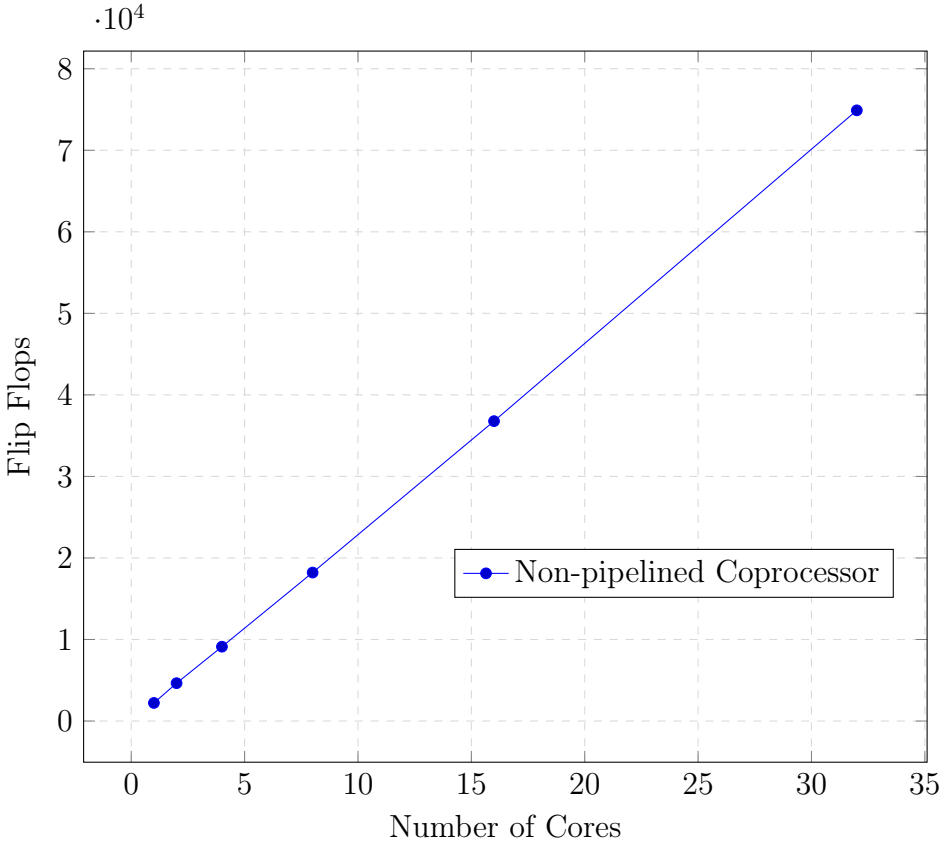


Figure 15.1: Number of Cores Vs Hardware resources

Pipelined Version

Number of Cores	FFs
1	2294
2	4801
4	9453
8	18898
16	38195
32	77705

Table 15.2: Hardware Resources

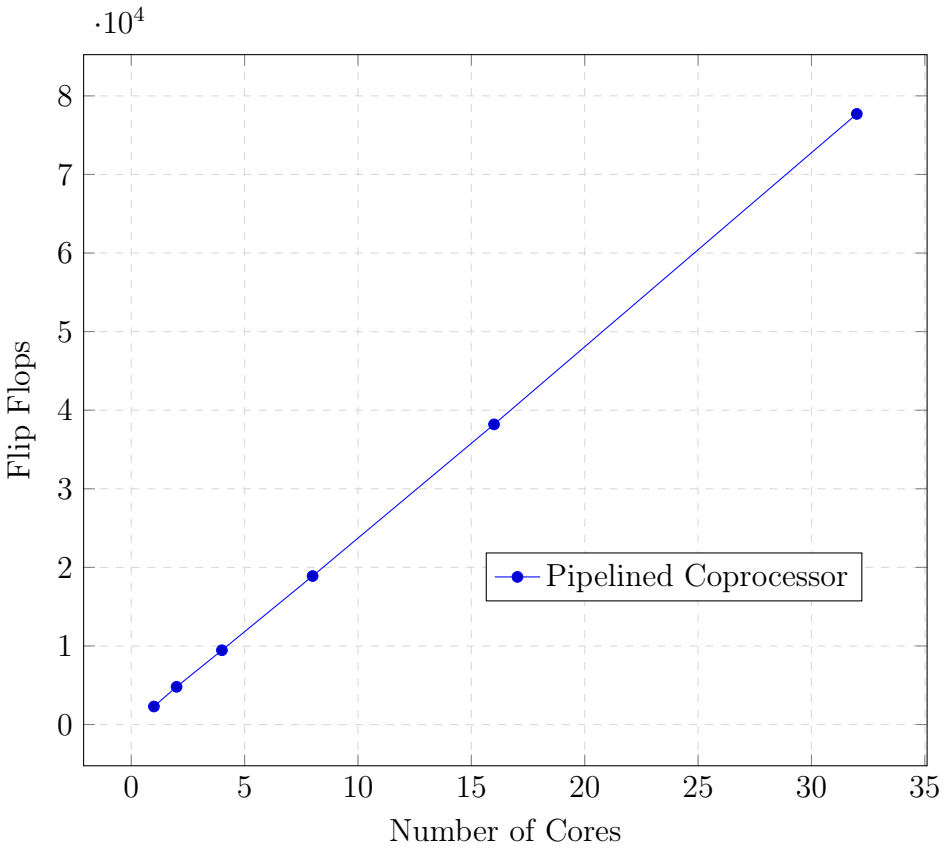


Figure 15.2: Number of Cores Vs Hardware resources

15.2.2 Aa to VHDL

Non-pipelined Version

Number of Cores	FFs
1	794
2	1635
4	3063
8	6053
16	11952
32	23885

Table 15.3: Hardware Resources

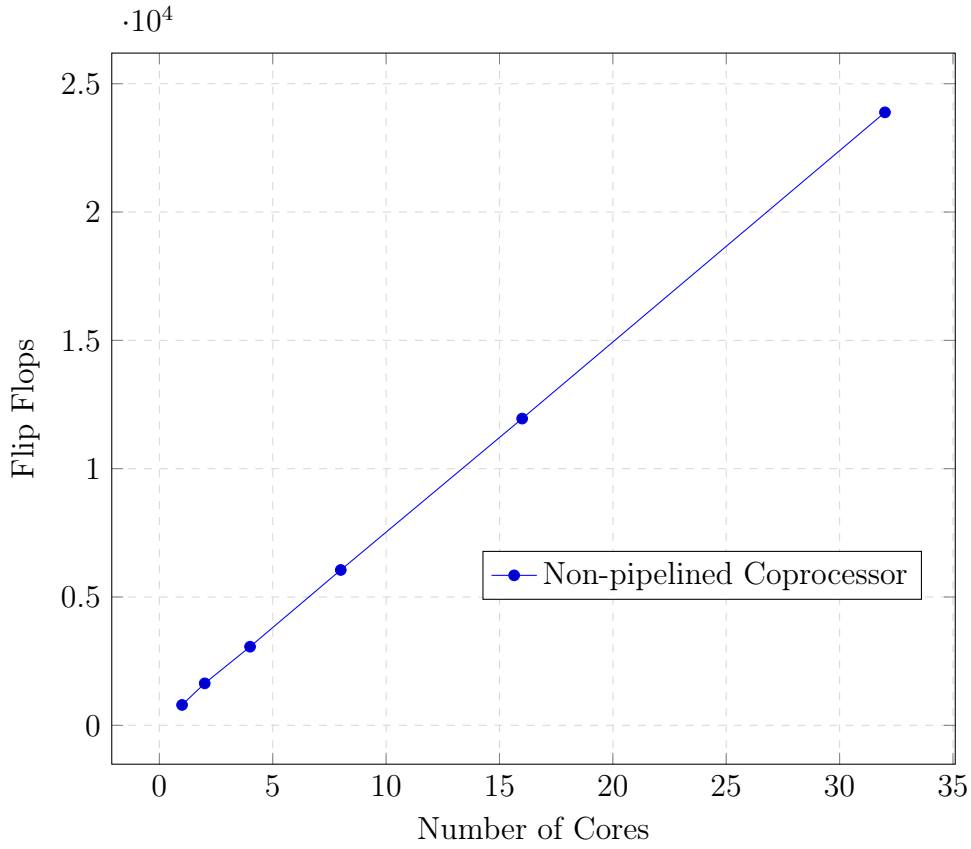


Figure 15.3: Number of Cores Vs Hardware resources

Pipelined Version

Number of Cores	FFs
1	871
2	1935
4	3639
8	7113
16	14090

Table 15.4: Hardware Resources

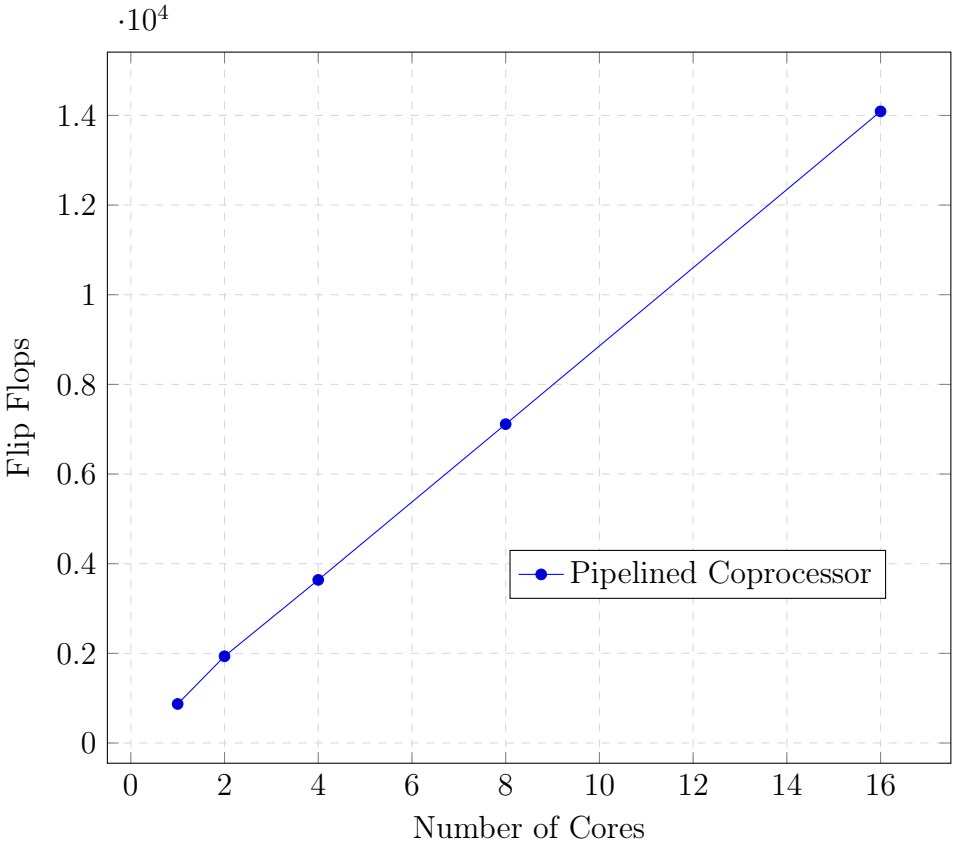


Figure 15.4: Number of Cores Vs Hardware resources

A | Abbreviations Used

Abbreviation	Full Form
PCIe	Peripheral Component Interconnect Express
AXI	Auxiliary Express Interface
AFB	AJIT FIFO Bus
AHIR	A Hardware Intermediate Representation
VC	Virtual Circuit

References

[1] Jeff Johnson's FPGA blog

[2] User manuals for 7 Series Memory Support

[3] Multi-port Memory Controller

[4] AXI4-Lite Manual

[5] March Test Reference

[6] RIFFA github homepage

[7] AaLRM : Aa Language Reference Manual

[8] NoC Wiki

[9] Dual Clocked FIFO reference article