Design and Development of Network Interface controller for AJIT processor based SoC

Master's Thesis

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
Integrated Circuits and Systems

by

Mr. Harshad Bhausaheb Ugale
Roll No. 20307R008

under the supervision of
Prof. Madhav P. Desai

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
June 2023

Dissertation Approval

This dissertation entitled

Design and Development of Network Interface controller for AJIT processor based SoC

by

Mr. Harshad Bhausaheb Ugale
Roll No. 20307R008

is approved for the degree of
Master of Technology in Electrical Engineering

.....	.....
Prof. Madhav P. Desai	Prof. D. Manjunath
(Supervisor)	(Examiner)
.....	.....
Prof.	Prof. Kavi Arya
(Chairman)	(Examiner)

Date: 2023-06-29
Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/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.

.....
Mr. Harshad Bhausaheb Ugale
Roll No. 20307R008

Date : 2023-06-29

Acknowledgment

I want to express my heartfelt gratitude towards **Prof. Madhav P. Desai** for allowing to work on this project and providing her valuable guidance throughout. His suggestions have helped me in gaining a better understanding of this research topic.

I would also like to thank members of **VLSI Lab** for providing all the necessary support.

Lastly, I am perpetually thankful to my family and friends for their unwavering encouragement and support.

Abstract

The rapid advancement of network technologies has created a growing demand for efficient and reliable network communication solutions. As a result, the integration of a Network Interface Controller (NIC) into an indigenous AJIT processor-based System-on-Chip (SoC) has become increasingly crucial. This thesis aims to address this need by designing and developing a NIC that enables seamless network connectivity and communication capabilities for the AJIT processor. Additionally, it highlights the growing reliance of AI and ML technologies on network-based applications.

The outcomes of this research demonstrate the successful integration of the NIC into the AJIT SoC prototype. The NIC design and verification are carried out using AHIR-V2 tools, which provide an algorithmic approach to digital design. The developed NIC empowers the AJIT processor to efficiently send and receive data packets over the network, facilitating seamless connectivity with external devices and systems. Moreover, an AI/ML inference engine is employed to showcase the potential of leveraging Ethernet interface for advanced computational tasks.

The implications of this research extend beyond the immediate context of the AI/ML inference engine. The designed version of the NIC is flexible and can be modified to incorporate additional complex functionalities, such as self-routing and NIC-to-NIC communication. This opens up possibilities for the development of routers and other network devices with multiple NICs, offering enhanced network capabilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid advancement of network technologies has revolutionized various aspects of our lives, ranging from communication to data processing and beyond. In this era of interconnected systems, the need for efficient and reliable network communication solutions has become paramount. The integration of a Network Interface Controller (NIC) into an indigenous AJIT processor-based System-on-Chip (SoC) serves as a crucial step towards achieving seamless network connectivity and communication capabilities.

This chapter provides an introduction to the design and development of a Network Interface Controller for the AJIT processor-based SoC. The AJIT processor, being an indigenous processor offers unique opportunities for customization and optimization to meet specific requirements. By integrating a NIC, the AJIT SoC can effectively send and receive data packets over the network, enabling seamless connectivity with external devices and systems. Furthermore, this chapter discusses the goals and challenges, associated with the design and development of the Network Interface Controller. The goals include integrating the Ethernet interface into the AJIT SoC prototype, demonstrating the use of an AI/ML acceleration application, and generalizing the design for network appliance SoCs such as routers. By achieving these goals, the research contributes to the broader field of network communication solutions.

Overall, this chapter sets the stage for the subsequent chapters, which delve into the design, implementation, and evaluation of the Network Interface Controller. It establishes the importance of network connectivity in the context of the AJIT processor-based SoC and provides a roadmap for the rest of the thesis.

1.1 Objectives

1.1.1 Integration of Ethernet into AJIT System-on-Chip (SoC) Prototype

The first objective is to successfully integrate an Ethernet interface into the AJIT SoC prototype. This involves designing and implementing the necessary hardware components(Network Interface Controller) and firmware to enable Ethernet connectivity.

1.1.2 Network-Accelerated AI/ML Application using Ethernet Interface

The second objective is to utilize the integrated Ethernet interface of the AJIT processor to enable network-accelerated execution of AI/ML acceleration applications. This demonstration highlights the synergy between the processor's advanced computing capabilities and enhanced connectivity, showcasing the seamless integration of the Ethernet interface to support efficient processing and communication in networked AI/ML workloads.

1.1.3 Generalization to Network Appliance SoC (Router)

The third objective aims to generalize the network interface design to a broader context by extending it to a network appliance SoC, specifically a router. This involves evaluating the feasibility of incorporating the network interface into a router SoC, considering factors such as performance, power consumption, and compatibility with networking protocols.

By achieving these goals, this research aims to enhance the AJIT processor's capabilities and explore its potential in networked computing systems, paving the way for future advancements in indigenous processor development.

1.2 Challenges

1.2.1 Network Interface Controller Design and Validation

One of the key challenges in this research is the design and validation of the network interface controller. This involves designing a controller that can efficiently handle

the transmission and reception of Ethernet packets, ensuring proper synchronization, error handling, and protocol compliance. The validation process includes rigorous testing to verify the functionality, performance, and reliability of the controller under different network conditions and workloads.

1.2.2 Integration of Xilinx's MAC IP

Another challenge is the integration and configuration of Xilinx's MAC (Media Access Controller) IP into the network interface design. This requires understanding the specifications and functionality of the MAC IP and ensuring its seamless integration with the AJIT processor's architecture. The challenge includes addressing any compatibility issues, optimizing configurations, and verifying the correct operation of the MAC IP within the overall network interface design.

1.2.3 Software Design for Reliable Exchange of Information over Ethernet

Implementing reliable communication over Ethernet presents a challenge that requires careful software design. This involves developing protocols and algorithms to ensure the reliable exchange of information between the AJIT processor and external devices connected via Ethernet. The software design should handle packet loss, retransmissions, flow control, and other mechanisms to guarantee the integrity and timeliness of data transmission in the networked environment.

1.2.4 Software Design for SOC Firmware in AI/ML Acceleration Application

In the context of the AI/ML acceleration application, a significant challenge lies in the software design of the system-on-chip (SoC) firmware. This involves developing firmware that efficiently interacts with the network interface, handles data transmission and reception, and interfaces with the AI/ML acceleration modules. The firmware should be optimized for performance, ensuring seamless integration of the AI/ML application with the Ethernet interface and leveraging the network acceleration capabilities of the AJIT processor.

Chapter 2

Design and Validation of Network Interface Controller

This chapter focuses on the design and validation of the Network Interface Controller (NIC) for the AJIT processor-based System-on-Chip (SoC). The NIC plays a critical role in enabling network connectivity and communication capabilities within the SoC.

The design of the NIC involves several components and considerations, including network protocol support, data packet handling, memory management, and interface with the AJIT processor. To ensure a robust and reliable design, the NIC undergoes a rigorous validation process to verify its functionality, performance, and compatibility with the AJIT SoC architecture.

The primary goal of this chapter is to provide a detailed overview of the design process, highlighting the key components and their interconnections. It explores the challenges encountered during the design phase and discusses the solutions and design choices made to overcome them. Additionally, the chapter delves into the validation methodologies employed to ensure the correctness and efficiency of the NIC design.

By describing the design and validation of the NIC, this chapter aims to provide insights into implementing a network interface solution for the AJIT processor-based SoC. It serves as a foundation for subsequent chapters, which will explore specific aspects of the NIC design, such as the data packet handling mechanisms, and memory management schemes.

2.1 Design decisions

Before directly jumping on NIC design let's take a look at the necessary design decisions made. The NIC will receive packet data from MAC which will be stored in memory. The processor will need to allocate this memory and provide that information to NIC. This overall needs to 3 main interfaces to NIC. Figure 2.1 shows all the interfaces. Let's see each interface in detail.

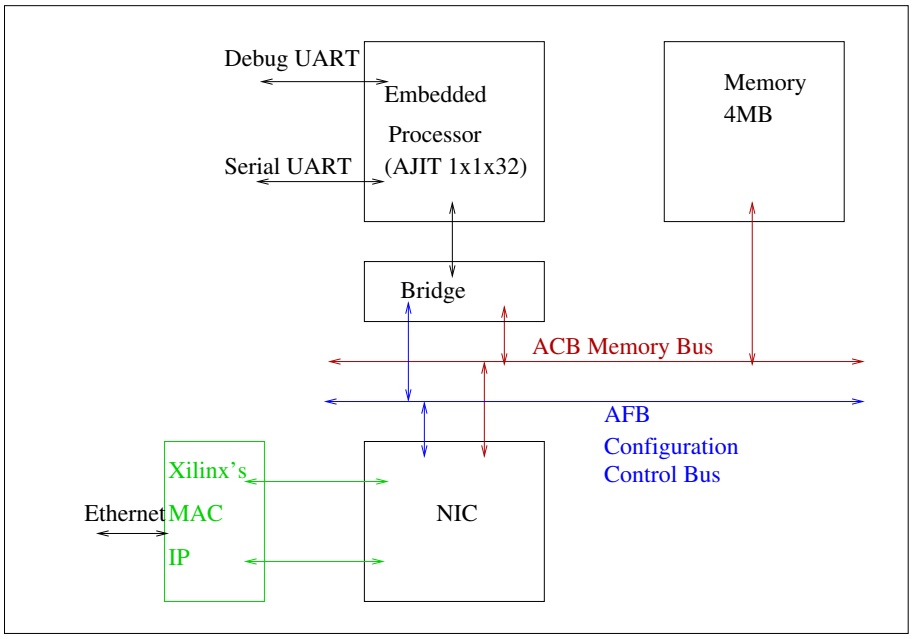

Figure 2.1: Top level with Interfaces

2.1.1 NIC-MAC interfce

NIC to MAC interface will used by NIC for receiving and transmitting packets from MAC. The memory which will be used is provided 8 bytes per request. so this interface is kept 73(64 bit data + 9 bit control) bits. The bit mapping is shown in table 2.1. The same interface will be used for both reception and transmission of packets.

2.1.2 NIC-Memory interface

The NIC-Memory interface is required for storeing and loading packets to and from memory. Already developed ACB(AJIT Core Bus) protocol will be used for this. The protocol consists of two interfaces,

Signal Name	Location	Signal Description
tlast	[72:72]	*tlast* becomes '1' if the 64 bit chunk is last chunk of packet.
tdata	[71: 8]	*tdata* is actual packet data chunk
tkeep	[7: 0]	*tkeep* is 8 bit field, each bit is mapped to 8 bytes of data. if any bit is '1' then corresponding byte in data is valid else not.

Table 2.1: NIC-MAC interface description

1. ACB Requeuest : Requests from NIC to store and load the packet will be sent through this interface. see 2.2 for bit mapping.
2. ACB Response : Response generated by memory to the request will be sent back to NIC on this interface. see 2.3 for bit mapping.

2.1.3 NIC-Processor interface

This will be more of a control interface. Processor will allocate the memory space for packet storage and provide thaat info to NIC using this interface. NIC will have registers inside which will written by processor using this interface. For further information see 2.3.1. An already developed AFB(AJIT FIFO Bus) protocol will be used for this. This AFB protocol also has two interfaces like ACB protocol only the address width is half.

1. AFB Requeuest : Requests from Processor to write or read the NIC reg will be sent through this interface. see 2.4 for bit mapping.
2. AFB Response : Response generated by NIC to the request will be sent back to Processor on this interface. see 2.5 for bit mapping.

Signal Name	Location	cSignal Description
lock	[109:109]	lock bit, if set to '1' by a master then other master's don't get access to Mmory.
read/write_bar	[108:108]	if '1', the request is read request, if '0', the request is write request.
byte_mask	[107: 100]	*byte_mask* is 8 bit field, each bit is mapped to 8 bytes of data. if any bit is '1' then corresponding byte in data is valid else not.
address	[99:64]	addres(byte) where read/write should be performed.
write_data	[63: 0]	Data to be written.

Table 2.2: NIC - Memory interface description

Signal Name	Location	Signal Description
err	[64:64]	Value '1' indicates errored response.
data	[63: 0]	Contains read data if the req. was read req.

Table 2.3: Memory - NIC interface description

Signal Name	Location	cSignal Description
lock	[73:73]	lock bit, if set to '1' by a master then other master's don't get access to Mmory.
read/write_bar	[72:72]	if '1', the request is read request, if '0', the request is write request.
byte_mask	[71: 68]	*byte_mask* is 4 bit field, each bit is mapped to 4 bytes of data. if any bit is '1' then corresponding byte in data is valid else not.
address	[67:32]	addres(byte) where read/write should be performed.
write_data	[31: 0]	Data to be written.

Table 2.4: Processor - NIC interface description

Signal Name	Location	Signal Description
err	[32:32]	Value '1' indicates errored response.
data	[31: 0]	Contains read data if the req. was read req.

Table 2.5: NIC - Procesor interface description

2.2 Interfaces data structures

The interface data structures used in the NIC design consist of three queues: the free_queue, rx_queue, and tx_queue.

- **free_queue**: This queue holds the addresses of free buffers that are available for storing packets. The processor initially assigns a set of buffers and pushes their addresses into the free_queue. These buffers do not have any active packets and are ready to be utilized for storing incoming packets. Both the processor and the NIC can push and pop from the free_queue.

- **rx_queue**: The rx_queue is pushed by the NIC and popped by the processor. It holds the addresses of buffers that currently contain active packets. When the NIC receives a packet, it stores the packet in a buffer and pushes the address of that buffer into the rx_queue. This allows the processor to identify the buffers with active packets that are ready for processing.

- **tx_queue**: The tx_queue is pushed by the processor and popped by the NIC. Once the processor has finished processing a packet, it pushes the address of the processed packet buffer into the tx_queue. The NIC monitors the tx_queue and retrieves the buffer addresses from it to send the packets out over the network.

These queues enable efficient coordination and communication between the processor and the NIC, ensuring the proper handling and processing of packets.The queue header format is shown below,

```
typedef struct _CortosQueueHeader {
        uint32_t totalMsgs; // current total messages
        uint32_t readIndex;
        uint32_t writeIndex;
        uint32_t length;
        uint32_t msgSizeInBytes;
        uint8_t *lock;
        uint8_t *bget_addr;
        // if misc == 1, then assume single writer
        // and single reader and don't use locks
        uint32_t misc;
} CortosQueueHeader;
```

To ensure synchronization and prevent conflicts during access to these queues, a locking mechanism is implemented. The locking mechanism utilizes atomic operations, which guarantee thread-safe access and modifications to the queues. This ensures that only one entity can perform push and pop operations on the queues at a given time, preventing simultaneous modifications and preserving the integrity of the queue data.

At startup, the processor initializes the queues by allocating memory for them and configuring their initial state. The addresses of these queues are then communicated

to the NIC by writing to specific NIC registers. This process allows the NIC to access and manipulate the queues effectively during runtime.

These are all interface decisions taken lets now move on to NIC design. Let's see the NIC registers now.

2.3 Network Interface Controller

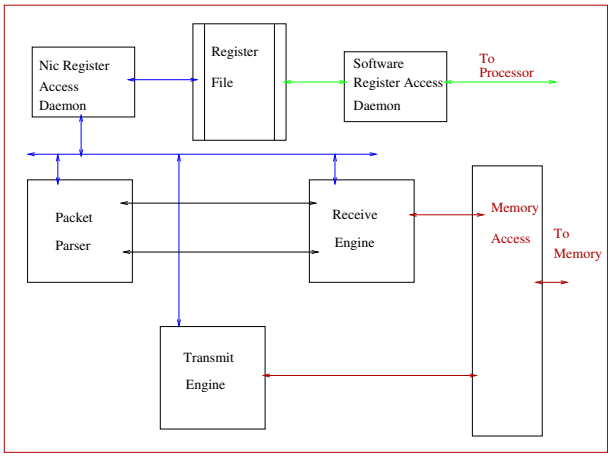

Figure 2.2: Architecture of Network Interface Controller

2.3.1 Register File Map

The NIC (Network Interface Controller) registers are specific memory locations within the NIC that are used for configuration, control, and status monitoring purposes. These registers allow communication between the processor and the NIC, enabling the processor to control and monitor NIC. The NIC registers provide a standardized interface for the processor to interact with the NIC and perform tasks such as enabling or disabling the NIC, setting queue address and monitoring the status of data transmission and reception.

2.3.2 Packet Storage Format

2.3.3 Parser

The parser daemon receives data from the MAC (Media Access Control) through a pipe and extracts relevant information from Ethernet packets. It utilizes a state machine to

Reg. ID	Address offset	Description
0	0x00	Control reg
1	0x04	Number of Servers
2	0x08	Address of rx_queue
10	0x28	Address of tx_queue
18	0x48	Address of free queue

Table 2.6: NIC registers map

process chunks of data and identifies essential details such as source and destination MAC addresses, type/length field, and packet data. The parsed information is then forwarded to receive engine daemon.

```
loop1 :
if(enable_by_processor)
       loop2 :
       -> read from MAC
       -> if(Header)
                -> send to header & packet pipe
                -> goto loop2
       -> else
                -> send to packet pipe
                -> goto loop2
else
       -> goto loop1
```

2.3.4 Receive Enigne

The receive engine daemon is responsible for the storage of packets coming from the parser. It receives the parsed packet information from the parser daemon and interacts with the processor to ensure the proper handling of received packets. The receive engine daemon uses the free_queue, to get an empty buffer address to store the active packets in buffers. Then uses rx_queue to provide their(buffer's) addresses to the processor for processing. The algorithm of receive engine is shown below,

```
loop1 :
if(enable_by_processor)
       loop2 :
       -> count = 0;
       -> buf_addr = pop from free queue
       loop2.1:
       -> Read from header_pipe and write to buff_addr
       -> count++
       -> if(!header_end)
                -> goto loop2.1
       loop2.2:
       -> Read from packet pipe and write to buf_addr
       -> count++
       -> if(last_chunk)
                -> write count and last bytemast to buf_addr[0].
                -> push buf_addr to rx_queue.
                -> goto loop2
       -> else
                -> goto loop2.2
else
       -> goto loop1
```

2.3.5 Transmit Engine

The transmit engine daemon focuses on transmitting processed packets from the processor to the external network. It receives the addresses of processed packet buffers from the processor via the tx_queue and sends the corresponding packets out through the Ethernet interface. The transmit engine daemon monitors the tx_queue and retrieves the buffer addresses to facilitate efficient packet transmission. The daemon also pushes free_queue with the address of buffers which is sent out. This allows the reuse of buffers. The algorithm of transmit engine is shown below,

```
loop1 :
if(enable_by_processor)
        loop2:
        -> buf_addr = try to pop tx_queue
        -> if(pop successful)
```

```
                -> Rx = read control data(buf_addr[0])
                -> count = extractCountFromRx(Rx)
                loop3:
                -> read packet from buf_addr
                -> send out to MAC
                -> count--
                -> if(count == 0)
                        -> push buf_addr to free_queue.
                        -> goto loop2
                -> else
                        -> goto loop3
        -> else
                        -> goto loop2
->else
        goto loop1
```

2.3.6 Software register Access

The software register access daemon enables the processor to access and modify the NIC's software registers. These registers contain various control and configuration parameters, allowing the processor to configure and manage the behavior of the NIC. The software register access daemon handles the communication between the processor and the NIC registers, ensuring reliable and secure access. The processor uses AFB protocol(see **??**) for register access.

2.3.7 Nic register Access

The NIC register access daemon provides the necessary interface for the NIC to read from and write to its internal registers. These registers store critical information for the proper functioning of the NIC, including configuration settings, status flags, and other control parameters. The NIC register access daemon ensures that the processor can interact with these registers and modify them as needed to configure and manage the NIC's behavior.

Signal Name	Location	cSignal Description
read/write_bar	[42:42]	if '1', the request is read request, if '0', the request is write request.
byte_mask	[41: 38]	*byte_mask* is 4 bit field, each bit is mapped to 4 bytes of data. if any bit is '1' then corresponding byte in data is valid else not.
reg_index	[37:32]	index of register to which read/write should be performed.
write_data	[31: 0]	Data to be written.

Table 2.7: NIC to Ref file interface description

Signal Name	Location	Signal Description
err	[32:32]	Value '1' indicates errored response.
data	[31: 0]	Contains read data if the req. was read req.

Table 2.8: Reg file to NIC interface description

2.4 Helper Modules

1. **memoryAccess**: This module can be called from other modules and used to read and write from and to memory.
2. **pusiIntoQueue**: This module is used to push buffer pointer to provided queue.
3. **popFromQueue**: This module is used to pop buffer pointer form provided queue.
4. **acquireLock**: This module is used to lock the queue for push and pop operation. It first reads the queue lock by setting memory lock to '1' and then acquires queue lock if its available and then releases memory lock.
5. **releaseLock**: This module is used to release the acquired queue lock.

2.5 Test Setup

2.6 Validation and Performance

The data rate found using 1x1x32 (single core single-threaded) AJIT processor are as shown in table 2.9

Only NIC with Memory we are able to achieve 433.2847 Mbps for 1750 packets of size 146 Bytes.

Table 2.9: Data rate achieved for differnet number of packets & packet sizes

	Packet Size(in Bytes)		
No. of Packets	**48**	**136**	**236**
	Data Rate(in Mbps)		
1	11.3316	31.7085	51.2869
10	18.8431	57.2255	102.4069
100	22.4589	64.1140	110.9845
1000	23.0982	64.0665	114.4293
10000	23.0892	64.3163	114.7907
50000	23.0983	64.4662	114.8123

Chapter 3

SoC Design and Validation

We validate an characterize the performance of the accelerator by integrating it into a System-on-chip (SoC) using AJIT Processor, which provides the accelerator with the commands to operate, and reads back the data. The whole system is also connected to a Network Interface Card, which provides high-speed IO for the quick loading of images into the memory. The architecture of the system is shown in Fig 3.1.

The system consists of a 32-bit wide AJIT processor at its core. The processor has an ACB interface through which it interacts with memory and other modules. The AFB interface linked to the Network Interface Controller (NIC) and the 8-engine accelerator cluster is used to transfer configuration information from the processor and relay back status flags to it. The modules are also provided with an ACB interface through which they can access the memory. The memory subsystem consists of a DRAM controller connected to an ACB to UI conversion protocol, which translates memory requests in the ACB bus to DRAM requests, and vice versa for the responses. In addition to the above, the processor has a couple of UART interfaces with which it can communicate with the outside world. Also, the NIC has an interface to connect with the Xilinx MAC IP, using which it can communicate to the host machine via Ethernet, currently configured to run at 100Mbps.

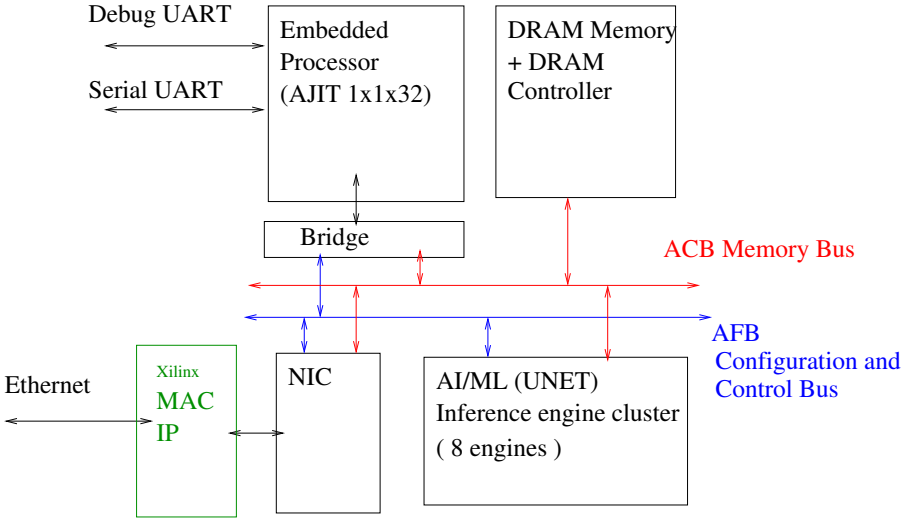

Figure 3.1: NIC, Processor and accelerator Soc Design

The pseudocode for the processors code is as follows:

```
[numbers=left]
initialiseMemorySpaces()
initialiseNicQueues()
fetchKernelsThroughEthernet()

while(1) do
fetchInputsThroughEthernet()

initliaseAcceleratorStates()

while (all accelerators not done) do
for engine in list_of_engines do
convolve(engine,stage)
updateState(engine,stage)
end for
end while
sendOutputsThroughEthernet()
end while
```

The above pseudocode explains the working of the processor code. It first initialise memory spaces required by all modules. These include the queues required by NIC, the kernel addresses for the accelerators and the memory spaces for receiving and storing tensors during computation by the engines. After the memory spaces are initialised, all the queues for NIC are configured and initialised, and the Network Interface Controller starts executing. The kernels are then fetched in through Ethernet, which is then followed by the input tensors for all the engines. With the setup ready, NIC is turned off and the engines are allowed to execute on the data stage by stage. When all the acclerators are done with their computation, the output data is sent to the host PC through Ethernet, where the output is verified for correctness.

The above is a simplified driver to demonstrate the functionality of using multiple accelerator engines inside the system using polling. The processor and the accelerators both support the use of interrupts. The Interrupt Service Routine corresponding to the accelerator interrupts can be suitably configured to allow the processor to be used for other processes, while the engines work on the compute-intensive AI/ML tasks, the

completion of which can be signaled to the processor, at which the processor interrupts and schedules the next task to the accelerator before resuming its execution.

3.1 Processor NIC Interface

The processor initializes the queues for the NIC and starts the NIC. When NIC receives data from MAC it pops the free_queue which has empty packet buffers address stored in it and writes the packet to that buffer. After successfully writing the packet NIC pushes the address of the buffer to rx_queue, any buffer address in rx_queue indicates that this is the received packet and needs some action by the processor. While this thing is going on processor keeps polling the rx_queue, as soon as it gets any data over there it reads it and takes the decision. In this setup, the processor is expected to store the packet at some memory location which is shared with the accelerator through its registers. After this packet storage is complete processor pushes this address to tx_queue which works as acknowledgment for the host, after receiving it (the host) sends a new packet. This is the overall flow for storing files in the memory.

Now to send the files out on the ethernet interface processor breaks a file into a number of packets, pops free_queue for empty buffer address, and stores the packet at that address. After storing the packet successfully processor pushes the buffer address to tx_queue. The transmit engine running inside NIC which is polling this tx_queue reads that buffer and sends it out. One by one all the packets are sent out.

3.2 Processor Accelerator Interface

The processor begins by feeding the data to registers 1 to 12. After that, it sets the bits 0 and 2 of register 0, which signals the accelerator to start working. The processor can then continue with other tasks or poll on register 0 as it waits for the accelerator to complete, which is indicated by setting bit 4 of register 0 as high. Then, the processor updates the registers to the parameters for the next stage or image.

In addition, the accelerator has a control daemon that resets the registers, reads the data from the AFB_ACCELERATOR_REQUEST, writes them onto the accelerator registers, and sends back the response to the AFB_ACCELERATOR_RESPONSE. The

control daemon runs infinitely waiting on AFB_ACCELERATOR_REQUEST requests from the processor.

The accelerator also has a worker daemon, which waits for the appropriate bits to be set in the r0 register. When the processor sends the request to execute through the registers, it calls the core function with the parameters stored by the processor in other registers. When the execution is completed, it writes back 0 into bit 4 of r0, thereby signaling the processor that the computation is completed.

The accelerator also has an interrupt daemon that waits on bit 4 of r0, and sets the signal ACCELERATOR_INTERRUPT_8 corresponding to the above bit. The interrupt signal is not currently used but can be utilized to prevent the processor from polling on r0 while the accelerator performs the computation.

3.3 Timing Result and Performance

Chapter 4

Summary & Conclusion

4.1 Conclusion

4.2 Future work

Appendix I

```
module sum(
  input a,b;
  output sum, cout
  );

  assign {cout, sum} = a + b;

endmodule
```

Listing I.1: tmp code template

Algorithm 1 Tmp alg template to use.

Require: $n \geq 0$
Ensure: $y = x^n$
  $y \leftarrow 1$
  $X \leftarrow x$
  $N \leftarrow n$
  while $N \neq 0$ **do**
    if N is even **then**
      $X \leftarrow X \times X$
      $N \leftarrow \frac{N}{2}$ ▷ This is a comment
    else if N is odd **then**
      $y \leftarrow y \times X$
      $N \leftarrow N - 1$
    end if
  end while

Bibliography