Department of Electrical and Computer Engineering

COMPSYS 701 – Advanced Digital Systems Design

Final Report

Heterogeneous Multiprocessor System on Network-on-Chip Design

Prepared by Andrew Lai, John Zhang, Sean Wu

Group 8

12th June 2017

Heterogeneous Multiprocessor System on Network-on-Chip Design

Group 8 (AJS) - Andrew Lai (`klai054`), John Zhang (`szha215`), Shiyang Wu (`swu145`)
Department of Electrical and Computer Engineering, University of Auckland, New Zealand

June 11, 2017

Abstract

With Moores Law becoming harder to prove, multi-core or multiprocessor systems are becoming more popular ways to improve performance rather than focusing on architecture size or maximum frequency. This report presents a multiprocessor system design based on the Network-on-Chip implemented in System-on-Chip, including designing a new hardware acceleration processor, network interfaces and integrating them alongside existing components using the SystemJ programming language which can run Java programmes on hardware-accelerated Jave Optimised Processors (JOP).

Keywords: Processor design, Network-on-chip

1 Introduction

2 Network-on-Chip Structure

2.1 Basic Architecture

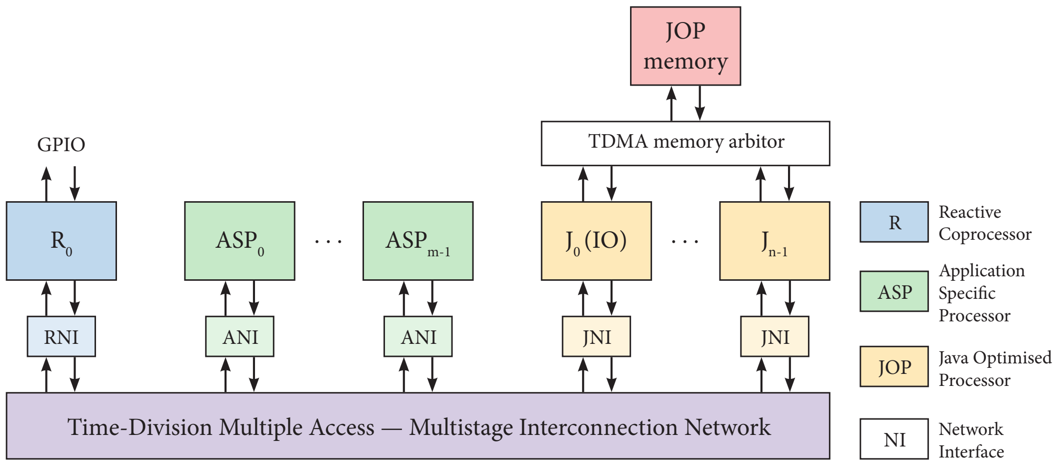

Figure 1: Top-level diagram of TDMA-MINoC

2.2 Configuration

3 Reactive Coprocessor (AJS)

3.1 Control Unit

3.2 Datapath

3.3 Data Format

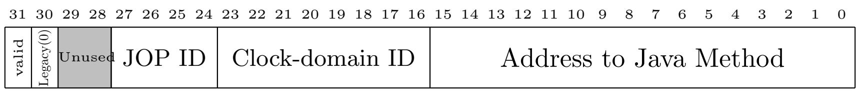

Figure 2: ReCOP_AJS Data-call Packet

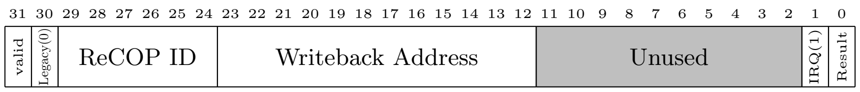

Figure 3: ReCOP_AJS Data-call Result Packet

3.4 Simulation

4 Application Specific Processor

4.1 Control Unit

4.2 Datapath

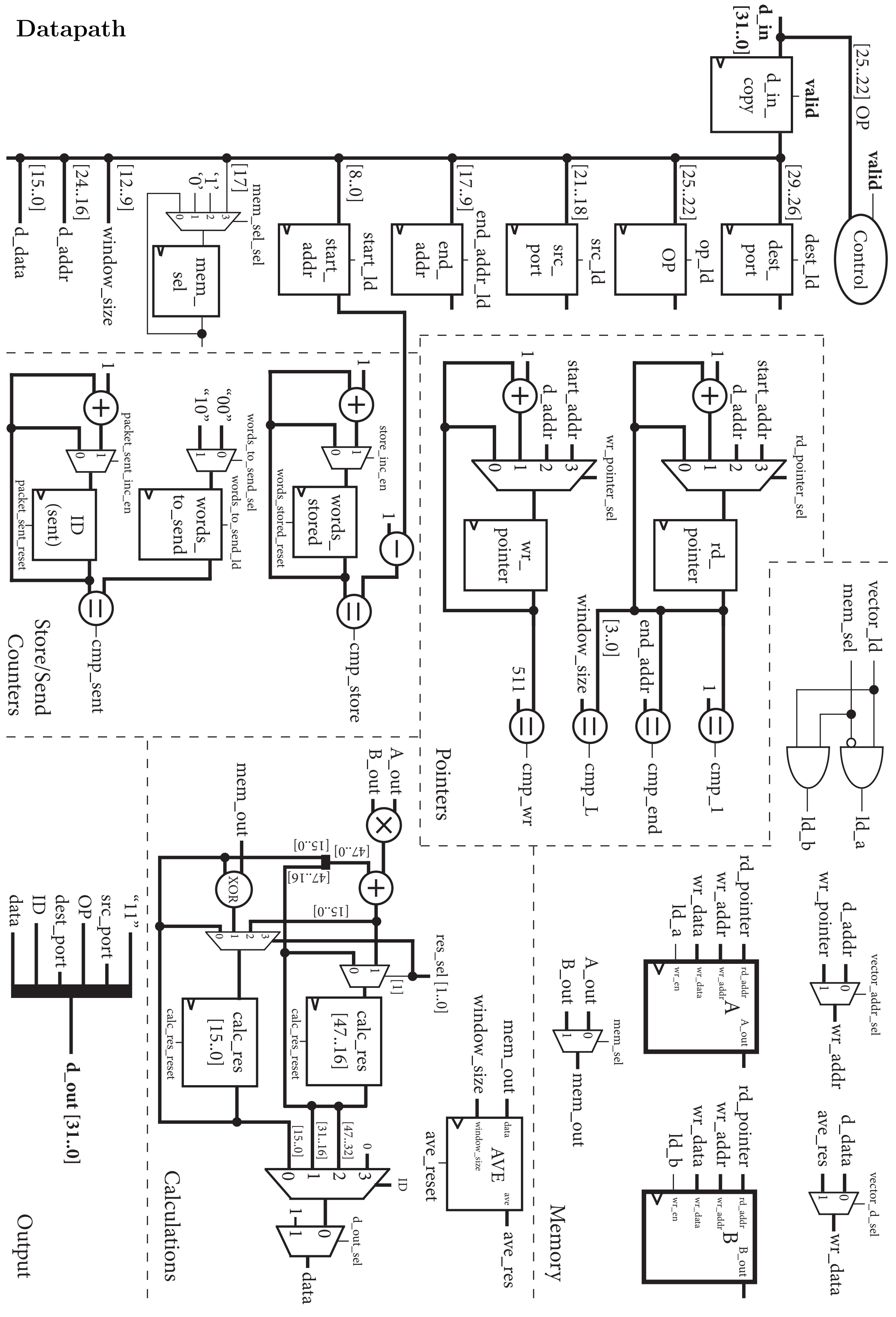

Figure 4: ASP datapath.

Fig. 4 shows the datapath of the ASP.

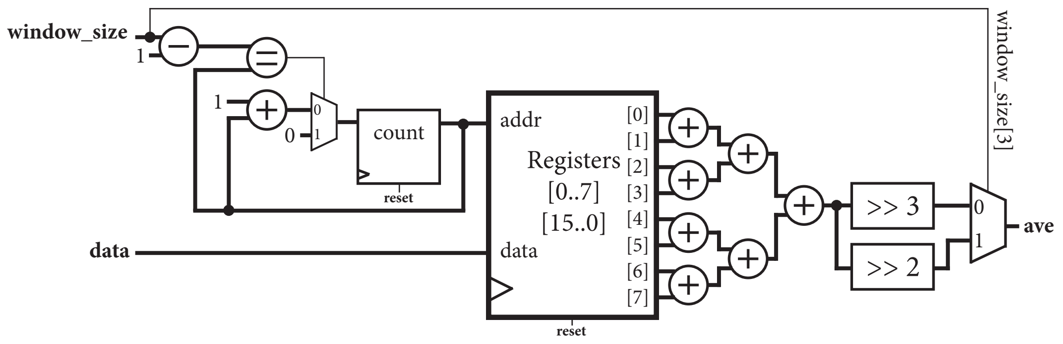

Figure 5: Average Filter datapath

4.3 Operations

4.3.1 Store Reset

Resets either vector A or vector B to 0 depending to the `mem_sel` bit. Returns `Operation Complete` (formerly known as `Access Granted` packet – 0x0001).

4.3.2 Store

This operation will store a fixed number of words to the ASP internal memory. It will send two types of packets – a Store Invoke packet, followed by a series of Data packets (see section 4.4). The Store Invoke packet contains information on the number of words and which vector to be stored in. Data packets contains the 16-bit data and 9-bit address indicating the memory location to be stored at. The number of data packets must match the store invoke packet's specified number of words. An

4.3.3 XOR

This function will perform a bit-wise XOR operation to the specified vector from index `begin address` to `end address` inclusively. After the operation, ASP will send the result to the ANI.

4.3.4 MAC

Multiply and Accumulate will perform a dot product opertion between index `m` to `n` for vector A and B.

$$S = \sum_{i=m}^{n} A(i)B(i) \tag{1}$$

There is a high probability that the MAC result will overflow a 16-bit data structure, with possible maximum being 41 bits.

$$(2^{16})^2 \times 512 = 2^{41} \tag{2}$$

41-bit data can be sent as 3 packets of 16 bits. To stay consistent, the ASP will always send 3 packets consecutively. In order to differentiate between the packets, a 2-bit packet ID is inserted into each packet, from 0 to 2. ID 0 being the first packet and ID 3 being the last. It is up to the JOP programme to concatenate those 3 packets into a `long`.

4.3.5 AVE

Average filters vector A or B with a run-time window size (L) of 4 or 8 using the algorithm described in (3). The datapath of the AVE block in Fig. 5 shows that the 8 internal registers are always summed and right shifted by 2 or 3 depending on the window size, and data is always stored at data registers at index `count`. The `count` is an up counter that resets after its value is `window size -` 1, as the register indexing is 0 to 7, while the `window size` can be 4 or 8.

$$X(i) = \sum_{k=i}^{i+L-1} \frac{X(k)}{L} \quad \text{where } i \in [0, N-L) \tag{3}$$

This allows pipelining, since it is always averaging a stream of data coming in, replacing the oldest. Upon initialisation, the registers are reset, `rd_pointer` points at 0 and starts to increment by 1, `wr_pointer` is set to point at 0, but not incrementing. When `rd_pointer` is equal to the `window size`, the `AVE` registers will be filled, outputting the first average value, and `cmp_L` will be high. The `cmp_L` signal triggers a state transition, and `wr_pointer` starts incrementing, and `vector_ld` is set high in order to write into memory. This continues until the `rd_pointer` overflows and equal to 1 – when the element

4.4 Data Format

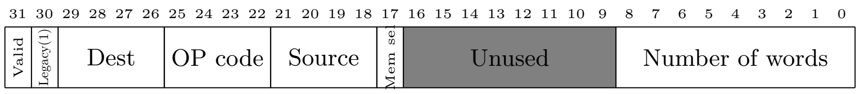

Figure 6: Store invoke packet format to ASP

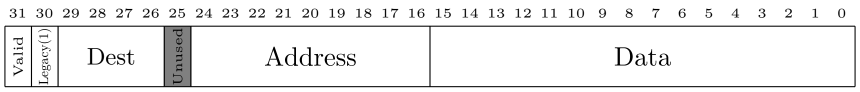

Figure 7: Data packet format to ASP

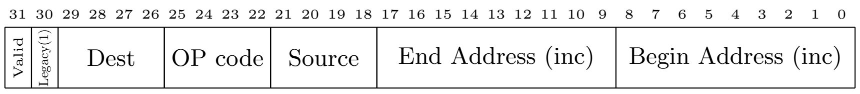

Figure 8: Command invoke packet format to ASP

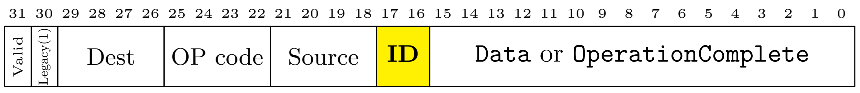

Figure 9: Result packet format from ASP

The `last bit` in the original design was not used.

5 ASP Network Interface

In order for the ASP to communicate with the rest of the system, a network interface, ANI has been designed. The ANI contains two Altera Megafunction First-in-first-out (FIFO) queues to ensure all packets are sent and received.

5.1 Datapath

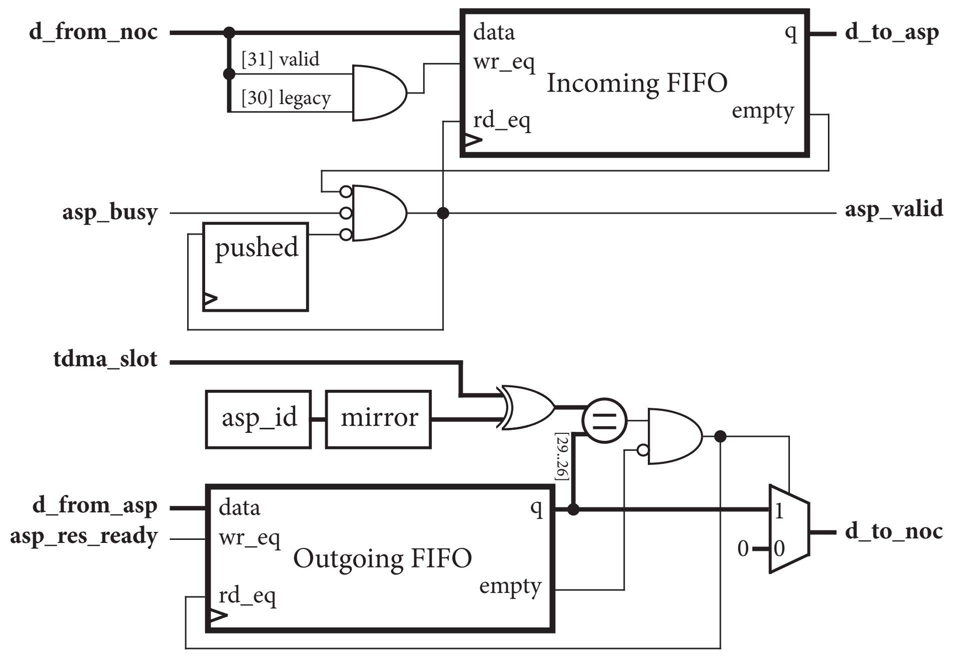

Figure 10: ANI datapath

5.2 Interaction with TDMA-MIN

5.3 Timing

5.4 Simulation

5.5 Sythesis

5.6 FPGA Testing

5.7 Performance Analysis

6 JOP Network Interface

7 Java Programmes

7.1 ASP APIs

Serveral APIs are provided in the `ASPCommunication` class. The developer has the responsibility of calling them using the correct `ASPid`, which specifies the ASP this packet is intended for. JNI will handle the correct ports, for the particular ASP ID. The number of ASPs is specificed in the top-level configuration file. The list of APIs implemented is:

1. `private static void sendPacket(int packet);`

   Writes the packet to the data call result register, which will be sent to the JNI and out to the NoC. Used by the other public methods.

2. `public static int pollASPResponse();`

Polls the datacall register, checks *valid* bit.

3. `public static int storeReset(int ASPid, int memSel);`

   Sends the reset instruction to ASP with `ASPid` and reset its vector depending on the value of `memSel`.

4. `public static int store(int ASPid, int[] data, int start, int memSel);`

   Sends a store instruction to ASP of `ASPid` with an Integer array. The instruction will command ASP to store the array from `start` to `data.length` to vector specified by `memSel`.

5. `public static int xor(int ASPid, int memSel, int start, int end);`

   Sends an xor instruction to ASP with `ASPid`. The addresses of xor instruction is from `start` to `end`.

6. `public static long mac(int ASPid, int start, int end);`

   Sends a mac instruction to ASP with `ASPid`. The addresses of mac instruction is from `start` to `end`. ASP will return 3 packets of data and this API will return the cacatenated data with a long type.

7. `public static int ave(int ASPid, int windowSize, int memSel);`

   Sends an ave command to ASP with `ASPid`. `windowSize` specifies the window size of the moving average filter.

All of the above listed APIs are blocking - the programme will freeze until the ASP has responded with the expected number of packets. However, this can cause time predictability issues and greatly increase the tick duration, which is undesirable. A potential solution has been provided in Section 9.

7.2 Multi-JOP Programme

A simple multi-JOP Java program was developed to perform a simple matrix multiplication:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} \tag{4}$$

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & 5 & 2 & 3 \\ 3 & 2 & 3 & 4 & 1 \\ 2 & 3 & 1 & 2 & 3 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 6 & 5 \\ 1 & 2 \\ 2 & 2 \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} 46 & 48 \\ 31 & 37 \\ 22 & 28 \end{pmatrix} \tag{5}$$

For the purpose of multi-JOP computation, each JOP will perform a row of matrix multiplication. Therefore for $JOP_i, i \in [0, 2]$, the calculations it will perform are:

$$\sum_{j=0}^{L} \mathbf{C}_{ij} = \sum_{k=0}^{N} \mathbf{A}_{ki} \times \mathbf{B}_{jk} \tag{6}$$

Where L is the number of columns of $\mathbf{C}$, and N is the number of columns of $\mathbf{A}$. Notice the number of rows of $\mathbf{B}$ is equal to N as well.

8 Integration

8.0.1 SystemJ Program

9 Future Work

Due to time constraints in the project, several parts of the design can be further improved.

9.1 ReCOP

Pipeline Ready TM

9.2 ASP

9.3 ANI

9.4 Java API

As mentioned in section JAVA APIS, the APIs are blocking. A possible fix is to create an object of the ASPCommunication class, storing the the current progress of the datacall. For storing data, the complete array can be divided into smaller arrays by keeping track of the indices and only send one smaller block per tick. While waiting for a response, instead of blocking poll, create a separate method responsible for polling and checking the OP code in the response packet

10 Conclusion

11 Acknowledgements

The authors would like to thank Professor Zoran Salcic and Dr Morteza Biglari-Abhari for their lectures on RTL, processor, and system designs, (soon-to-be Dr) Benjamin Tan, (soon-to-be Dr) Nicholas Harvey for their continuous guidance and support on this project. The authors would also like to thank Dr Heejong Park for his guidance and support for existing portions of the project.

12 References

Appendix