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

Contents

1 Introduction **2**
1.1 Project Phases . . . 3

2 Network-on-Chip Structure **3**
2.1 Architecture Overview . . . 3
2.2 Configuration . . . 4

3 Reactive Coprocessor (AJS) **4**
3.1 Control Unit . . . 5
3.1.1 ISA Categories . . . 5
3.1.2 FSM Refinement . . . 5
3.2 Data Format . . . 7
3.3 Datapath . . . 8

4 Application Specific Processor **9**
4.1 Control Unit . . . 9
4.2 Datapath . . . 10
4.3 Operations . . . 11
4.3.1 Store Reset . . . 11
4.3.2 Store . . . 11
4.3.3 XOR . . . 11
4.3.4 MAC . . . 11
4.3.5 AVE . . . 12
4.4 Data Format . . . 12

5 ASP Network Interface **13**
5.1 Datapath . . . 13
5.2 Timing . . . 14
5.3 FPGA Testing . . . 14

6 JOP Network Interface **15**

7 ReCOP Network Interface **16**

8 Java Programmes **16**
8.1 ASP APIs . . . 16
8.2 Multi-JOP Programme . . . 17

9 Integration **17**
9.1 FPGA . . . 17
9.2 Performance Analysis . . . 18
9.3 SystemJ Program . . . 19

10 Future Work **19**
10.1 ReCOP . . . 19
10.2 ASP . . . 19
10.3 ANI . . . 19
10.4 Java API . . . 20

11 Conclusion **20**

12 Acknowledgements **20**

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 12, 2017

Abstract

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

Keywords: Processor design, Network-on-chip

1 Introduction

There has been an increased demand on computation requirements and functionality, and it has become troublesome to continue the advancements in the capabilities of a single-core processor chip. An alternative attempt to fulfil this demand is the development of a multi-core system. There has been a development of advanced multicore system-on-chip which provide a very powerful execution platform which can either be comprised of homogenous, comprised of many cores of the same type, or heterogeneous, by mixing different types of cores. As there is an increased number of processor cores, the importance and emphasis of parallelism and time predictability become important factors to consider when trying to make sure that the platform is effectively executed.

This project report will present a take on developing a System-on-Chip (SoC) in which is based on the notion of the Network-on-Chip (NoC) concept. This will feature the development of a prototype SoC which allows for computer systems designers to program software or hardware solutions for embedded systems. This system-on-chip is to be a heterogeneous execution platform with different types of cores connected together; the resulting product is the called the ADD-HSoC (which stands for Advanced Digital Design - Heterogenous System on Chip). The system will utilise the access distribution method of Time-Division Multiple Access – Multistage Interconnection Network (TDMA-MIN) to achieve correct execution, with high bandwidth and with guaranteed latency for the prevention of packet interleaving when communicating between multiple cores.

The project is to be broken into phases, whereby each phase is a milestone which marks great strides in the development of the ADD-HSoC project. These phases entail: the design of processors, Application Specific Processor and Reactive Coprocessor from scratch to be placed in the NoC, the integration of the processors so that it is capable of inter-communication and a SystemJ application to demonstrate the functionalities of the ADD-HSoC.

1.1 Project Phases

1. Phase One: General and Application Specific Processor Design
   - Design of ReCOP processor (AJS)
   - Design of ASP and its network interface
2. Phase Two: Network Interface modifications
   - Program development for JOPs
   - Adaptation of JOP Network Interface (JNI) to support communication with ASP
   - Integration of JOP and ASP using TDMA-MIN
   - Integration of AJS ReCOP and JOP using TDMA-MIN
3. Phase Three: ADD-HSoC Integration
   - Integration of all components into full blown ADD-HSoC
   - SystemJ program demonstrating ADD-HSoC execution

2 Network-on-Chip Structure

2.1 Architecture Overview

The ADD-HSoC is a network-on-chip that consists of three different types of processors interconnected to execute SystemJ. The three main processor cores types are ReCOP, JOP and ASP. Each of these processors is capable of stand-alone execution and has its own instruction set architecture each unique to its own. Each type processor is capable of communicating with each other, however, with the introduction of TDMA-MIN, the identity of each packet will not be sufficient to communicate in a multi-core system. As a result, either modification to the processor or an external module must be made to adapt to a specific port in the TDMA-MIN system. The choice is to make network interfaces unique to each core to make the communications possible between cores.

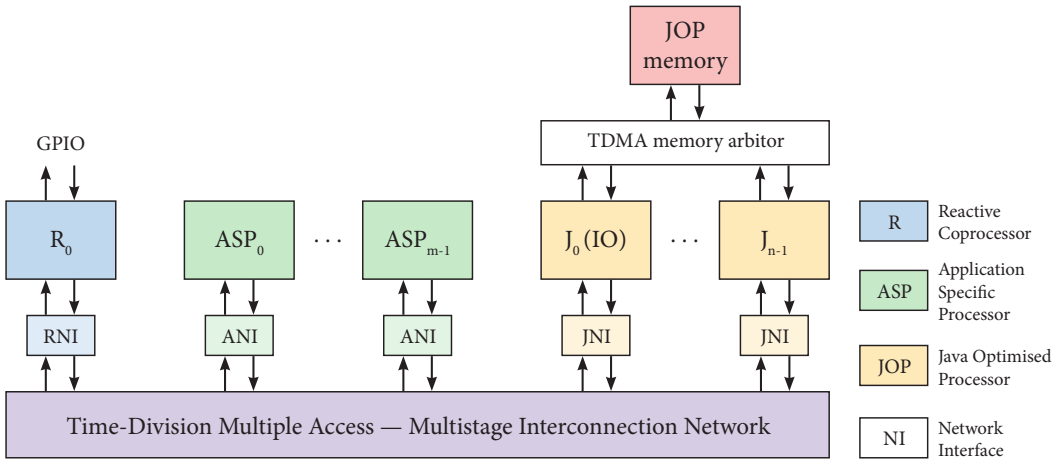


Figure 1: Top-level diagram of TDMA-MINoC

The overall structure for as seen in Fig. 1, is the protocol of interconnection for the multicore system. Each processor core connects to its own network interface and each network interface is connected to the TDMA-MIN. With this, the project initially started with a given JOP. JOP network interface, ReCOP and ReCOP network interface designs for a head start on the project, each one modified during the project to accommodate for the ADD-HSoC configuration.

2.2 Configuration

The configuration and function of the ADD-HSoC are for the system to feature multi-core intercommunication; this will ultimately be used to execute SystemJ applications. When SystemJ is compiled and programmed into the system, it is separated into their respective cores for computation. Each SystemJ application can be described with AGRC graphs, to capture the fact there are two different types of computation that can be simply described as a set of control and data computations [1]. This is where it can be specified that the ReCOP will execute control logic such as actions or decisions, and the JOP executes data computations like Java code functionality. This has already been developed where ReCOP Instruction Set Architecture (ISA) and Java have been made to be placed into their respective cores, the new aspect to the ADD-HSoC is to bring in an Application Specific Processor (ASP) to assist the JOP, much like an external hardware extension where the programmer is able to invoke the ASP if need be.

The execution of SystemJ can be extended to multiple ReCOPs, JOPs or ASPs to further catalyse computations for the execution platform and can be easily specified with the ADD-HSoC upon compilation. The top-level configuration file has been edited to accommodate for the addition of the new processor type in the system, the ASP, and a flag to choose between the ReCOP designs (v2 – provided, or AJS – designed). The number of ASPs can be up to 12, see Section 4.4 for details.

3 Reactive Coprocessor (AJS)

The ReCOP designed by AJS is inspired by the MIPS multicycle implementation. This is chosen partially due to how each distinct part of the processor can be easily taken apart to be debugged and tested, using simulations with tools such as ModelSim to verify the functional correctness. The priority would be to ensure the functionality of executing instructions over efficient or fast processing. With that being said, the multicycle implementation is still open to further developments to make fast execution of instructions, such as pipelining cycles.

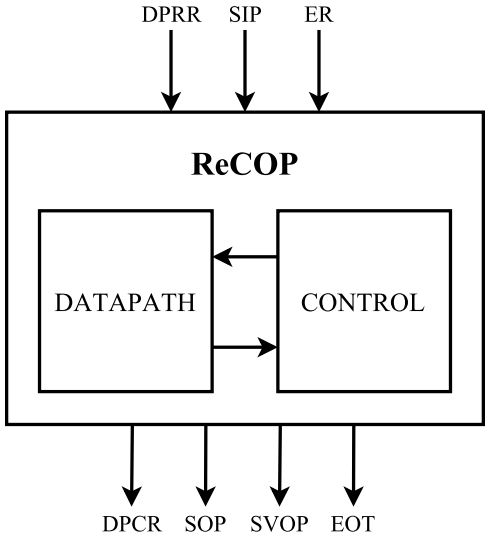


Figure 2: Top-level diagram of ReCOP_AJS

The design for the ReCOP is broken down into two main components, the datapath and the control unit as illustrated in Fig. 2. The datapath is as the name implies contains all data components where data transfers in registers and memory are done. The datapath will rely on the control unit to control the dataflow within the ReCOP. The control unit as usual with multicycle implementation is with the use of a finite state machine to compute specific instructions.

3.1 Control Unit

The instructions of the ReCOP is broken down into its specific cycles of execution to for the multicycle implementation. As such, the instructions have been categorised to make sure the control units finite state machine has a set of defined stages of control. The following is the categorisation of the ReCOP instruction set architecture.

3.1.1 ISA Categories

- C-Type (Computation)
  - `AND`, `OR`, `SUB`, `SUBV` and `MAX`
- L-Type (Load Register)
  - `LDR`, `LER`, `SSVOP`, `LSIP` and `SSOP`
- S-Type (Store Memory)
  - `STR` and `STRPC`
- JUMP Type
  - `JMP`, `PRESENT` and `SZ`
- D-Type (Data Call)
  - `DCALLBL` and `DCALLNB`
- F-Type (Flag Clear or Set)
  - `CLFZ`, `CER`, `CEOT` and `SEOT`

3.1.2 FSM Refinement

The ReCOP follows a simple categorisation of states, they are as follows: Instruction Fetch and Decode, Execution of Instruction and Data call Result service routine. These are illustrated in Fig. 3. Each section is further broken down to ensure the correctness of execution.

- *Instruction Fetch and Decode*

  The instruction fetch (`IF1`) is invoked and to set the PC to point to the next program memory address. This will fetch for the upper 16bit of the instruction, a stall state (`IF1S`) is in place to wait for synchronous register and memory to be stored. Then the instruction is decoded (`ID1`) and sent to the control unit to decipher if the lower 16 bits is needed to be used for the instruction (i.e. operand for immediate and direct addressing mode). While in this state, the operand is already prefetched in this state whether it is needed or not. If it is required, the control will set the PC to point to the next program memory address (`IF2` and `ID2`).

- *Execution of Instruction*

  The control unit will decide what the processor will do with the datapath based on what the instruction type is. The states are labelled with specific names to make it clear as to what sort of datapath controls it would require. The following is the breakdown of the execution states:
  - `EX`: If the instruction uses the ALU
  - `MA`: If memory access is needed to be loaded in the register
  - `LR`: If register is loaded
  - `SM`: If data is stored into memory
  - `JP`: If a change in PC with jump or conditional

  - `NOOP`: If no operation
  - `DC`, `DCS`, `DCC`: If data call is executed. `DC` invokes the data call, `DCS` stalls for the synchronous store of the `DPCR`. `DCC` clears the `DPCR` so the call is not up for more than one clock cycle and executed multiple times. The state is also the loop state for blocking datacalls

- *Data call Result service routine*

  This is a check on `DPRR` (`irq` bit of `DPRR` bit 1, shown in Fig. 6), for a result from the NoC. This will then run the store of the result into memory writeback address or into register zero depending on if the datacall was nonblocking or blocking respectively. If there is no result to be processed, the ReCOP will execute the next instruction, entering the Instruction Fetch and Decode (`IF1`) state.

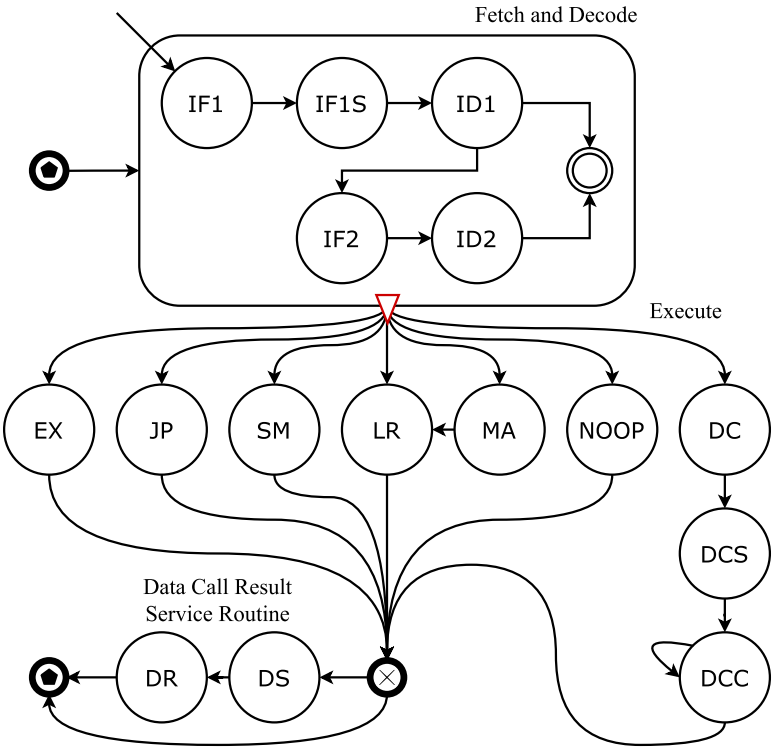


Figure 3: Control Unit FSM of ReCOP_AJS

3.2 Data Format

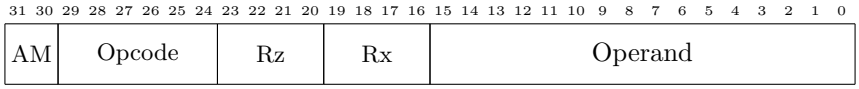


Figure 4: ReCOP_AJS Instruction Format

Instructions for the ReCOP are broken down into either 32 bits or 16 bits depending if the instruction will require an operand or not respectively (shown in Fig. 4). The addressing mode (AM) is what will decide if the operand is needed.

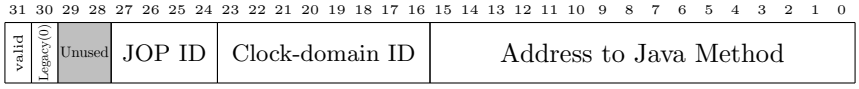


Figure 5: ReCOP_AJS Data-call Packet

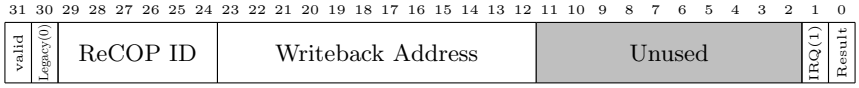


Figure 6: ReCOP_AJS Data-call Result Packet

Data calls (shown in Fig. 5 and Fig. 6) by the ReCOP to the JOP and vice versa in the ADD-HSoC leave the legacy bit to be set to always be 0. The packet also specifies the JOP or ReCOP IDs and not the port destination as the network interfaces will handle the aggregation of packets to their ports.

3.3 Datapath

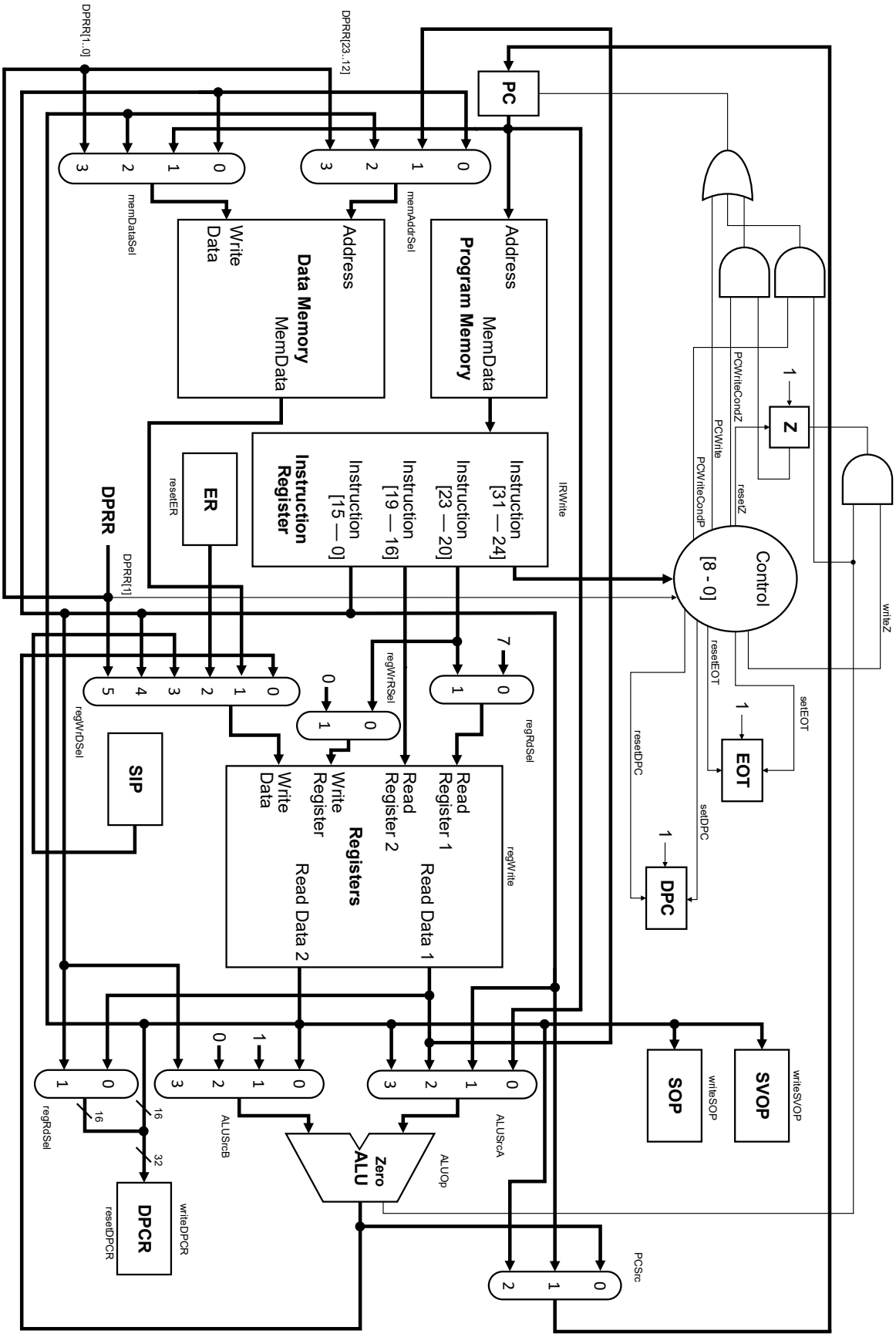


Figure 7: ReCOP_AJS Datapth.

4 Application Specific Processor

Application Specific Processors (ASP) are hardware accelerators that specialise on very few number operations to significantly reduce the number of cycles that a general processor would otherwise require to do.

An ASP was designed to accelerate a few arithmetic algorithms on arrays of data that would otherwise be performed on a JOP, including XOR, multiply and accumulate (MAC) functions, and average filter (AVE). The number of ASPs to be integrated into the network can be specified by changing the `num_asp` in the top-level configuration file.

As ASPs do not share memory with JOPs in the system, a JOP must send the data to the ASPs internal memory using STORE command through the ASP Network Interface (ANI) and TDMA-MIN (see section 5). The particular ASP designed has 2 N-element ($N = 512$ for this system) 16-bit vectors as data memory.

4.1 Control Unit

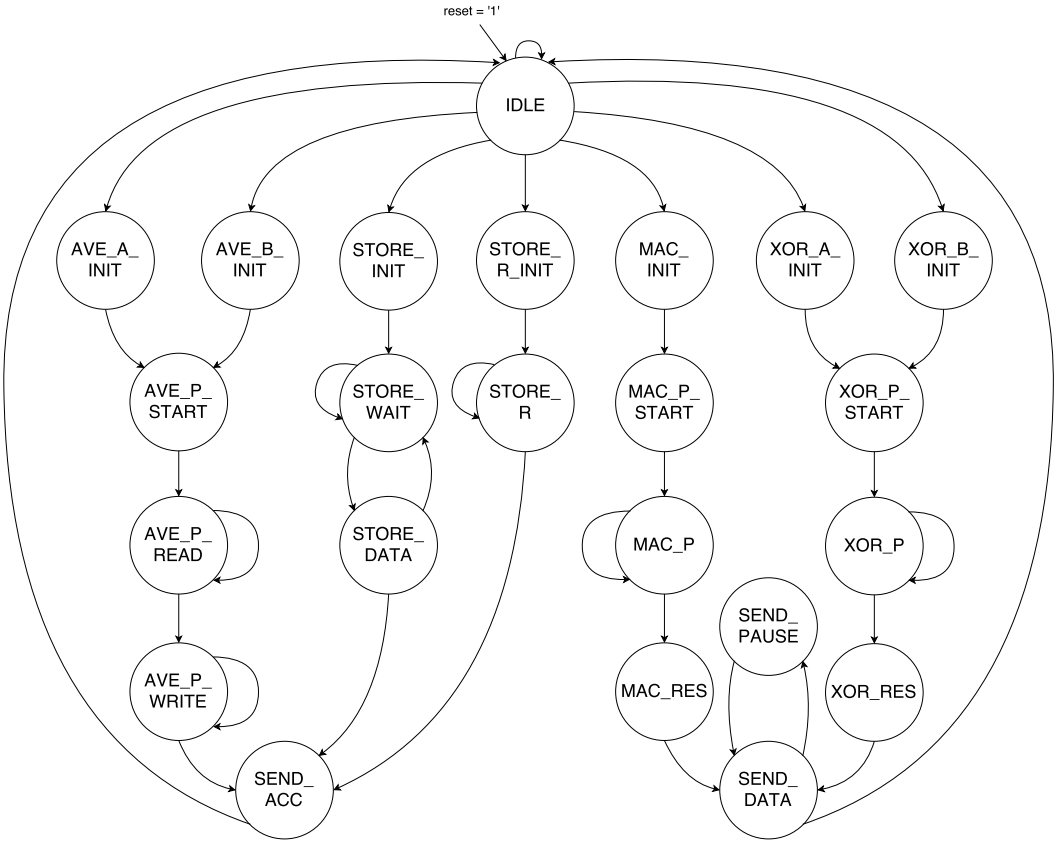


Figure 8: ASP Control Unit FSM

Control unit of the ASP uses the Moore FSM model to control the datapath in the system. ASP initialises in the `IDLE` state upon start up or if reset signal is high.

When the valid port is high, `d_to_asp` will be stored into ASP's internal register. If valid and legacy bits are 1's, a state transition depending on the op code will happen. All `INIT` states will clear result registers, initialises `rd_pointer` and `wr_pointer`.

Some operations require a stall cycle for the result registers to load before sending them, such as states `MAC_RES` and `XOR_RES`. There is also a one-cycle stall in the `SEND_PAUSE` between each packet sent.

`SEND_ACC` sends `Operation Complete` packet.

For complete state transition logic, see `ASP_state_transition_table.html`.

4.2 Datapath

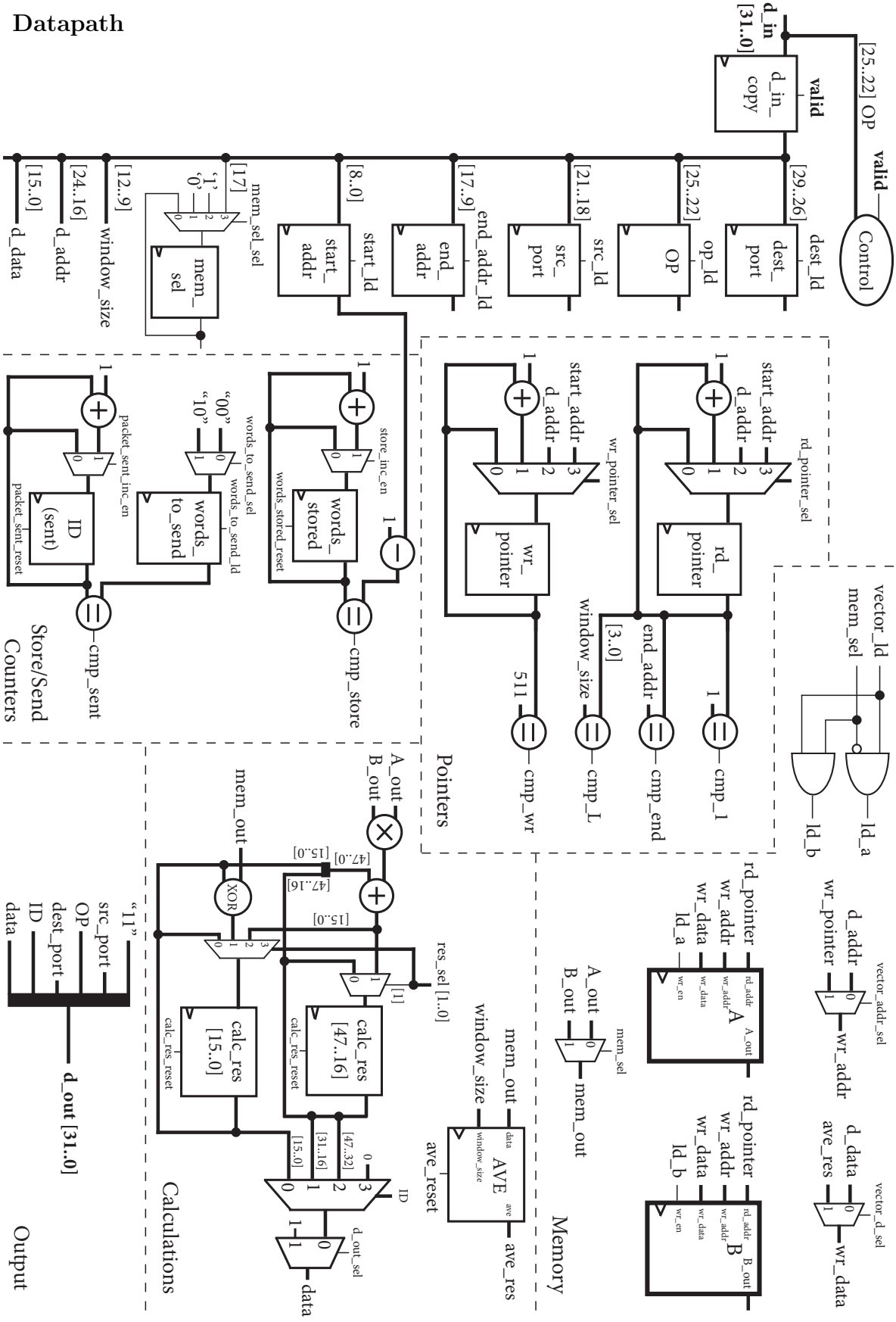


Figure 9: ASP datapath.

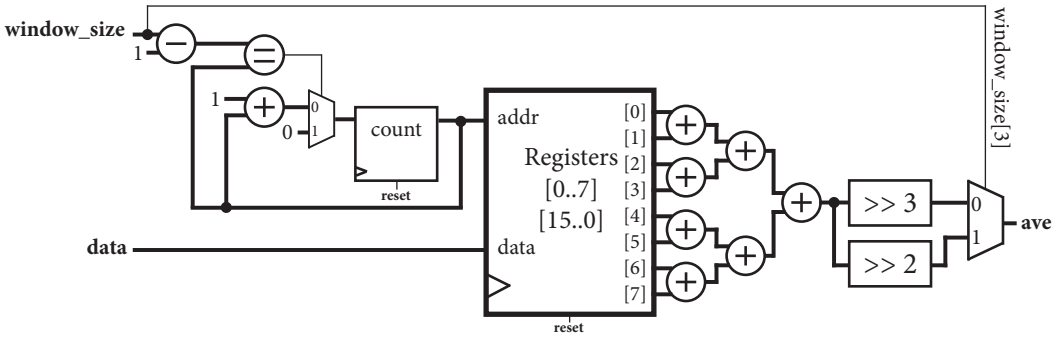


Figure 10: Average Filter datapath

4.3 Operations

4.3.1 Store Reset

Resets either vector A or vector B to 0 depending to the `mem_sel` bit. An `Operation Complete` (formerly known as `Access Granted` packet – 0x0001) will be sent back upon completion.

$$T_{execution} = N + 2 \text{ clock cycles}$$

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 contain 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. Returns `Operation Complete`.

$$T_{execution} = n_{words}(T_{communication} + 2) \text{ clock cycles}$$

4.3.3 XOR

This function will perform a bit-wise XOR operation to the specified vector from index `begin_address` (n) to `end_address` (m) inclusively. After the operation, ASP will send the result to the back.

$$T_{execution} = m - n + 3 \text{ clock cycles}$$

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 is the first packet and ID 3 being the last. It is up to the JOP programme to concatenate those 3 packets into a `long`.

$$T_{execution} = m - n + 9 \text{ clock cycles}$$

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. 10 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. Bit 3 of `window_size` is 1 when `window_size` is 8, and is 0 otherwise, therefore it can be used as the select line of the output multiplexer to select between the right shift of 2 or 3 (divide by 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 $N - L$ has been stored, which will then send `Operation Complete`.

$$T_{execution} = N + 3 \text{ clock cycles}$$

4.4 Data Format

Figure 11: Store invoke packet format to ASP

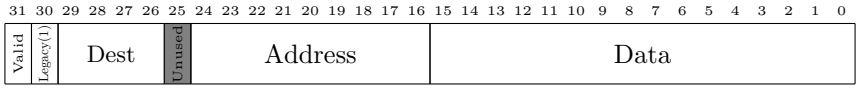


Figure 12: Data packet format to ASP

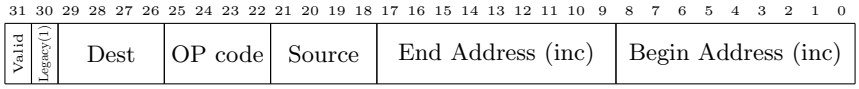


Figure 13: Command invoke packet format to ASP

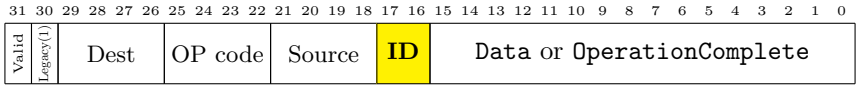


Figure 14: Result packet format from ASP

Due to the number of bits assigned to the destination port in ASP packet format – 4 bits, the maximum supported nodes the ASP can communicate to 16 (2^4). Each ReCOP and JOP takes a node in the network, therefore the maximum number of ASPs is 12, with 1 ReCOP and 3 JOPs.

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. It is also responsible for sending packets during the correct TDM Slot Counter.

5.1 Datapath

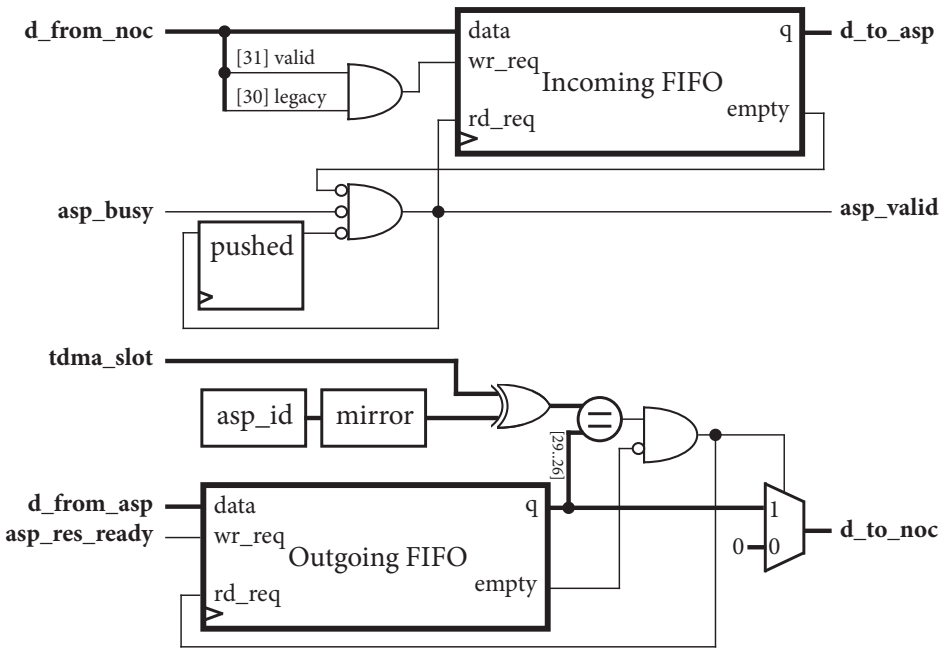


Figure 15: ANI datapath

ASP packets must have valid and legacy bits as 1's for pushing the packet from NoC into the Incoming FIFO. The head of the FIFO should only be popped to the ASP when it is not busy, not empty, and `pushed` as false. The `pushed` register makes sure that there is a one-cycle pause for the ASP to react to the previous packet before popping the next packet. `asp_valid` is set to high when popping to indicate that new packet is to be processed in the ASP.

When ASP has a packet ready to be sent to the NoC, it will have `asp_res_ready` high - this is used as the `wr_req` to push the data packet to the Outgoing FIFO. For ANI to send the packet at the head to the correct port, the ANI's own assigned `asp_id` is mirrored, XOR'ed and compared against the `dest_port`.

5.2 Timing

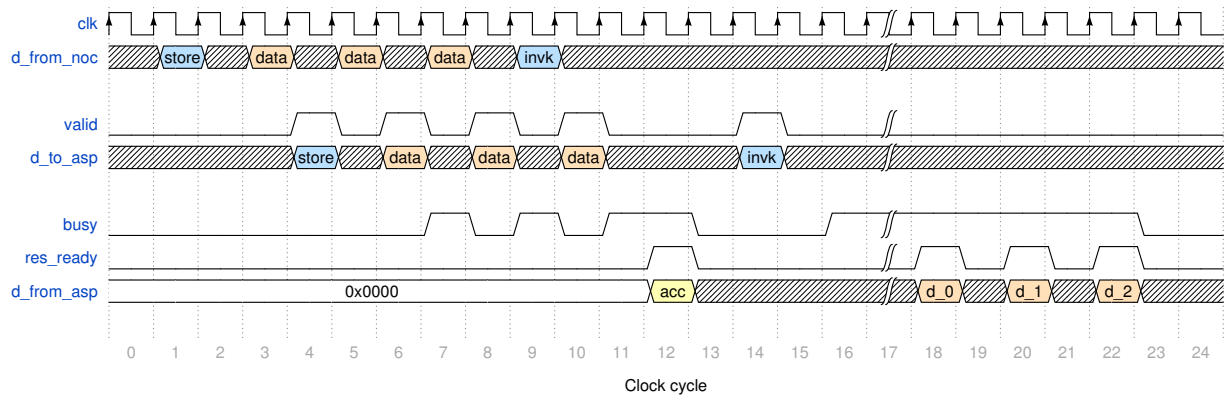


Figure 16: ANI timing with ASP

Fig. 16 shows the timings of packets from NoC and to NoC, `valid`, `busy` and `res_ready` for Store 3 packets, followed by a MAC invoke command.

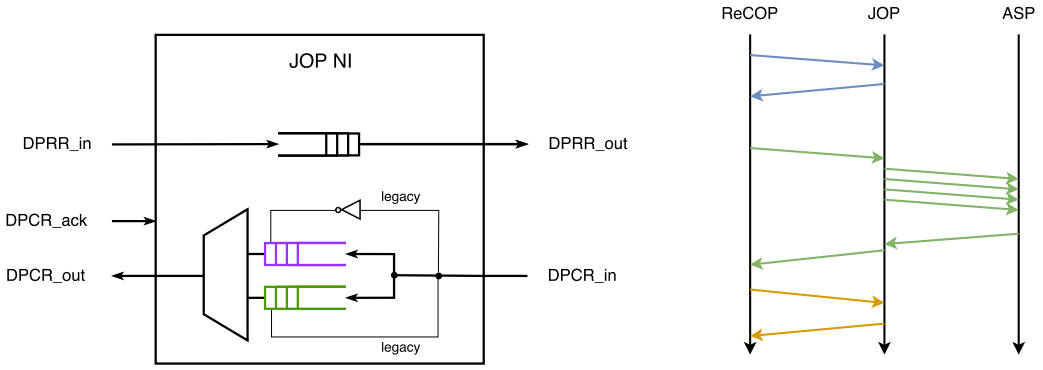
5.3 FPGA Testing

Both ASP and ANI were tested on the DE2-115 board during phase 1 of the project. The test bench consisted of:

- ASP
- ANI
- Fake JOP (emululates JOP ASP datacalls)
- Fake TDM Slot Counter
- BCD decoder for 7-segment display
- Switches and buttons from the environment

Upon a button press, Fake JOP reads the switches and send a packet or a series of packets hardcoded with ASP packets to the ANI's `d_from_noc` port. After command completion, the result packet from `d_to_noc` is transmitted to the BCD decoder and latched to be shown on the 7-segment display in hexadecimal format. This was when the ASP operation Store Reset was discovered to not be implemented – fixed during phase 2.

6 JOP Network Interface



(a) JNI Modifications

(b) ReCOP, JOP and ASP communication sequence

Figure 17

For the communication of the JOP with ReCOP and ASP, lies an underlying protocol of communication. Whenever the ReCOP talks with the JOP, it can send data calls multiple times and the JOP responds with corresponding packets back to the ReCOP. Servicing each ReCOP datacall by the JOP will be slower than the how fast the ReCOP can send, so a FIFO is in place to capture the calls so that there is no loss of data packets. Invoking JOP to execute data nodes will or will not invoke the ASP and as a result, may or may not expect data coming from the ASP. An example of this can be seen in Fig. 17b.

As there are multiple inter-communications between ReCOP, JOPs and ASP there lies chance that packets read are from ReCOP but are expected to be from the ASP. To prevent this interleaving of data, some adjustments have been made to the JNI; this can be seen in Fig. 17a. An additional FIFO will catch packets coming specifically from the ASP. The JNI will be able to aggregate the correct FIFO data to the JOP when there is an ASP invocation. How this is done is by checking any outgoing packets that are to be sent to the ASP, the data read by the JOP will now be switched to be reading from the ASP FIFO. Once the number of expected ASP packets have been received, the JOP will now be reading from the ReCOP FIFO.

To check for outgoing packets for whether or not the packet is intended to be for ASP or ReCOP is done by checking the legacy bit. As coincidently the bit for ASP communication packets are set to be always one and ReCOP specific packets to be zero. If there were to be any changes to this, then the JNI would be modified to check for port mapping done at compile time. This would check if at the TDM slot was connected to an ASP or ReCOP port.

Checking for the number of ASP packets to be expected is done by checking whether or not the outgoing packet is an ASP invoke packet. Depending on which instruction the ASP invoke is, will set the number of expected packets to be received, where all instructions receive one – except MAC which is three. The checking of expected packets is not done with a counter comparison as one would expect, but with a comparison of the ASP packet ID (seeing Fig. 14), making it not require an ALU.

7 ReCOP Network Interface

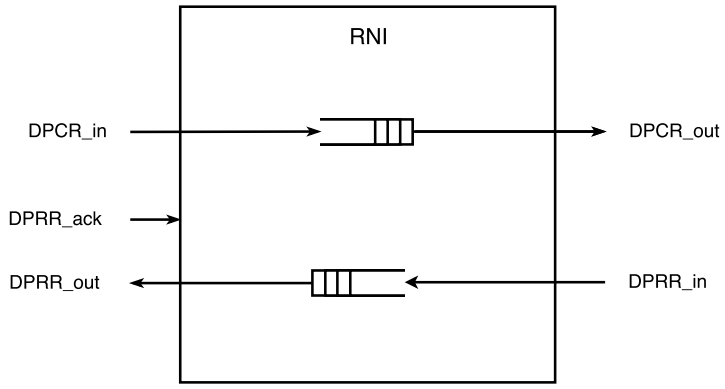


Figure 18: RNI modifications

The ReCOP_AJS should in the current case of the ADD-HSoC, runs datacall result service routines at the end of each currently executed instruction, and as such does not interrupt its current instruction when a result from JOP is done. This can result in the fact that some packets from JOP can be missed on each TDM slot. The issue becomes is very apparent when there are one ReCOP and multiple JOPs in the system, where results from many JOPs may all quickly arrive and be overwritten. A small modification has been made to alleviate this issue by placing a DPRR FIFO to capture all the datacall results.

8 Java Programmes

8.1 ASP APIs

Several 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 specified 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. Therefore this method cannot be used for sending packets to ReCOP directly.

2. `public static int pollASPResponse();`

   Polls the datacall register. If `valid` and `legacy` bits are 1, the packet is returned.

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

   Sends the reset invoke operation to ASP `ASPid` to reset vector indicated by `memSel`.This method returns the `Operation Complete` packet. Returns 1 upon completion.

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

   Sends a store instruction to ASP `ASPid` with an integer array. The instruction will store the array from `start` to `data.length` to vector specified by `memSel`. Returns 1 upon completion.

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

   Sends an XOR instruction to ASP `ASPid`. The addresses of XOR instruction is from `start` to `end`. Returns the XOR result.

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

   Sends a MAC instruction to ASP `ASPid`. The addresses of MAC instruction is from `start` to `end`. It will poll response until 3 packets have been received and concatenated before returning the result.

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

   Sends an AVE command to ASP `ASPid`. `windowSize` specifies the window size of the moving average filter – must be 4 or 8. Returns 1 upon completion.

All of the APIs listed above 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 10.

8.2 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.

9 Integration

In the final phase of the project, ReCOP_AJS was successfully integrated with the SystemJ programmes. Multi-ASP support was also implemented, allowing the developer to specify the ASPid in the API call. The number of ASPs is specified in the config file and generated during compilation.

9.1 FPGA

Entity	LE	M9K	DSP
1 ASP	484	2	2
1 ReCOP_AJS	1113	72	0
3 JOPs	16643	283	0
ANI	95	2	0
RNI	8017	0	0
JNI	67182	0	0
MINoC	413	0	0
Misc	7	0	0
Total	**93954**	**359**	**0**

Table 1: Resource consumption on FPGA

The complete system uses 82% of the total available logic elements (LE), 83% of M9K memory blocks on the DE2-115 board. Maximum operation frequencies were calculated by Quartus software using the Slow 1200mV 85^0C Model.

- $F_{\text{NoC\_MAX}} = 66.8MHz$
- $F_{\text{ReCOP\_MAX}} = 72.0MHz$
- $F_{\text{ASP\_MAX}} = 78.4MHz$

The critical path in the system is in the JOPs, therefore it is difficult to improve the current system clock frequency of 60MHz.

9.2 Performance Analysis

Operation (data size)	**JOP (us)**	**ASP (us)**	**Speedup factor (2 sf)**
XOR (64)	53	11	4.8
XOR (128)	101	12	8.4
XOR (256)	197	14	14
XOR (512)	543	18	30
MAC (64)	169	24	7.0
MAC (128)	332	24	14
MAC (256)	659	26	25
MAC (512)	1311	30	44
AVE (512, $L = 4$)	18145	18	1100
AVE (512, $L = 8$)	19706	18	1100

Table 2: Performance comparison between JOP and ASP, including communication time.

Execution times shown in Table 2 were recorded time differences between the start of the API call and after receiving the results averaged 3 times – therefore they include the time taken to write into the JOP datacall registers and communication time in the TDMA-MIN. Time stamps were retrieved from the microsecond program counter at memory location `IO_US_CNT`.

All supported ASP invoke operations were tested against a JOP using data sizes with increments of powers of 2, from 64 to 512 elements.

9.3 SystemJ Program

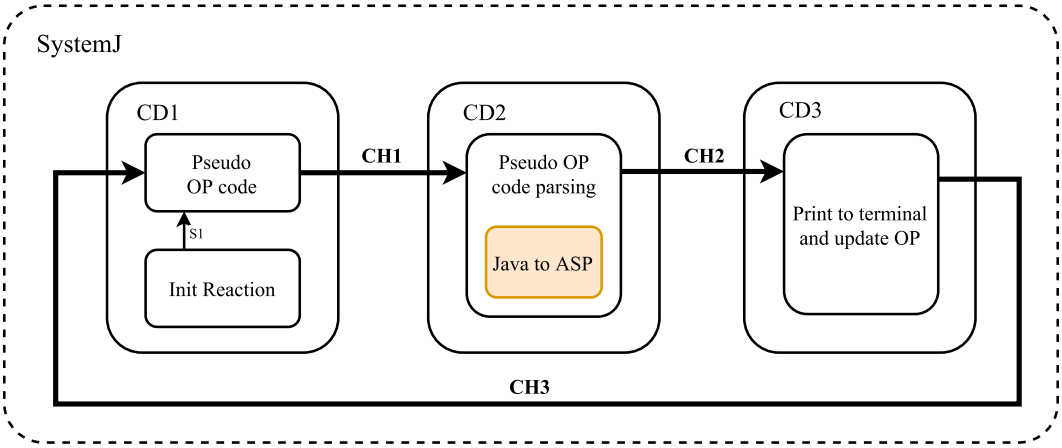


Figure 19: Clock-domain diagram for SystemJ program used

A SystemJ program was developed to showcase the ASP communication with AJS ReCOP in a multi-clock-domain (shown in Fig.19) scenario. The SystemJ program is similar to the Java program which is showcasing the communications between ReCOP_AJS, ASP and JOP. For SystemJ, one clock domain will be sending instructions to ASP after received an OP code from another clock domain. After the operation is completed on ASP and JOP receives the operation complete packet, this clock domain will send the opcode and result to the next clock domain to update the op code.

10 Future Work

Due to time constraints in the project, some aspects of the design were considered, but not implemented.

10.1 ReCOP

With the current nature of the multicycle implementation there lies possibility to expand the ReCOP functionality. The datapath has been laid out in a fashion where it is ready to be pipelined and it would be a simple process of sectioning each stage in the datapath into pipeline stages, as a result, the control unit will have to also accommodate for this future change. This will at least double the average number of instructions per cycle.

10.2 ASP

The memory blocks for vectors A and B can have two read ports. This can be used to parallelise some of the operations by having one read pointer going forward, starting at start address, and another read pointer going backwards, starting at end address until they rendezvous. This can reduce the execution time by up to half.

10.3 ANI

The current ANI design does not check whether the incoming packet has been delivered to the correct node in the network, which can be a security or data hazard. One potential fix is to compare the destination port within the packet against the ANI's assigned port number before pushing it into the incoming FIFO.

10.4 Java API

As mentioned in Section 8.1 the APIs are blocking. A possible fix is to create an ASPCommunication object to store the current progress of the datacall. When 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 from ASP, instead of blocking poll, call a separate method responsible for polling, checks the OP code in the response packet, concatenate the result if necessary then return it. Since ASP never return values larger than 2^{41}, any values larger can be treated as `result not ready`.

11 Conclusion

The project has undergone through some major breakthroughs in order to design all the components to construct a full-blown ADD-HSoC. The expected amount of progress for each phase were completed at scheduled times without any major obstacles.

The end prototype from the project showed promising results using SystemJ programs which show the communication between different processor types in the Heterogenous Multiprocessor System.

The Reactive Coprocessor designed by AJS featured in the ADD-HSoC is able to effectively take control of the SystemJ control logic. Approaches for the ADD-HSoC configuration and any future changes in the ADD-HSoC would result in a straightforward process.

Application Specific Processor that was designed during this project showed significant performance improvements compared to the existing Java Optimised Processors for specific computational operations.

A set of Java APIs for communication between JOP and ASP was developed for intuitive program development. As a result, the full overview of the ADD-HSoC project produced a fully functional platform for computer systems designers to easily develop SystemJ applications with impelling computation acceleration and efficiency.

12 Acknowledgements

The authors would like to thank Professor Zoran Salcic and Dr Morteza Biglari-Abhari for their lectures on RTL, processor, and system designs, Benjamin Tan, 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.

References

[1] A. Malik, Z. Salcic, and P. S. Roop, "Tandem virtual machine x2014; an efficient execution platform for gals language systemj," in *2008 13th Asia-Pacific Computer Systems Architecture Conference*, Aug 2008, pp. 1–8.