TOWER METHODOLOGY FOR VERIFICATION OF MULTI-CORE ARCHITECTURE – A CASE STUDY

by

Divya Parthasarathi

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical and Computer Engineering

Summer 2005

TOWER METHODOLOGY FOR VERIFICATION OF MULTI-CORE ARCHITECTURE – A CASE STUDY

by

Divya Parthasarathi

Approved: _____
Guang R.Gao, Ph.D.
Professor in charge of thesis

Approved: _____
Gonzalo R Arce, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: _____
Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic and International Programs

ACKNOWLEDGMENTS

"Every bit counts"

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible, whose constant guidance, motivation and encouragement aided me in the completion of the task.

I am extremely grateful to Dr.Guang R Gao for giving me an opportunity to work on such a great project and sparing is valuable time in guiding me. It has been thoroughly a great experience working with such a state of art project. He has not only given me guidance in my master thesis, but also has been my motivation.

I take this opportunity to express my deep sense of gratitude for the able guidance of Fei Chen, who admits his busy schedule was there to help me and shared his knowledge with me. He had also developed the code generator which was very useful in the verification process.

I would be failing in my duty if I fail to thank Dr. Dr. Monty Denneau at IBM and Ben Maron who are the designers of the Cyclops architecture. It was Ben's diagrams which were instrumental in the understanding of the project and these diagrams have been used through out this document. I would also like to thank all the CAPSL Hardware team for their valuable help. Finally to my parents and friends; to whom I owe more than what I can mention. I gratefully acknowledge the unending support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES vii
LIST OF TABLES viii
ABSTRACT x

Chapter

1 Introduction and Background 1

1.1 Introduction to Multi-Core Architecture 1
1.2 Problem Formulation 2
1.3 Problem solution 3

2 Description Of Cyclops 64 Architecture 6

2.1 C64 chip description 6
2.2 Inter-processor Communication module (A-Switch) 8
2.2.1 General Description 8
2.2.2 Packet Format of the Messages 9
2.2.3 Flow of Message in the network 12
2.2.4 Communication in 'A-Switch' 16
2.2.5 Address scheme used in A-Switch 20
2.2.6 Detailed description of the A-Switch 21
2.2.7 SWA module unit 25
2.2.8 SW_CS module unit 37
2.3 Verification needed for A-Switch 49

3 Functional Verification 50

3.1 Introduction 50
3.2 Tool flow for A-Switch 52

4. Software Emulation 55

4.1 Introduction 55
4.2 Tool flow for A-Switch 56

5 Conclusion 60

5.1 Conclusion 60
5.2 Future Work 60

Appendix

A Example of KSM file 61

B Example of a Shell file 63

C Exampe of KSF file 64

REFERENCES 65

LIST OF TABLES

Table 2.1 The port number of the neighbors in a 3D mesh configuration 11

Table 2.2 Channel program 17

Table 2.3 Header Format 17

Table 2.4 Tag Format 18

Table 2.5 Address and Tag Details 19

Table 2.6 Truth table of Decoder used in the Age Discriminator Unit 32

LIST OF FIGURES

Figure 1.1 Multi-Core processors 2

Figure 1.2 Block diagram of the Verification Methodology 3

Figure 2.1 Cyclops64 Supercomputer 7

Figure 2.2 Internal connection of the C64 Chip 8

Figure 2.3 Message Format in Cyclops 64 10

Figure 2.4 First double word 10

Figure 2.5 Subsequent Double words 10

Figure 2.6 Pictorial Representation of connection between the neighbors in C64 chip 11

Figure 2.7 Algorithm for changing the message 13

Figure 2.8 Example of Packet Routing 15

Figure 2.9 Address Format of A-Switch 21

Figure 2.10 Detailed view of the connection between crossbar and Switch 22

Figure 2.11 Block Diagram of the 'A-Switch' 23

Figure 2.12 Block Diagram of Make Spare Unit 24

Figure 2.13 Block Diagram of Extract Spare Unit 25

Figure 2.14 Block Diagram of SWA module 26

Figure 2.15 Block Diagram of Injection Queue/ Input port Unit 28

Figure 2.16 Block Diagram of Output Port module 31

Figure 2.17 Block Diagram of the Output Queue Unit 35

Figure 2.18 Block Diagram of SW_CS module 37
Figure 2.19 Block Diagram GlueA..... 38
Figure 2.20 Block Diagram of Glue A unit 39
Figure 2.21 Block Diagram of Glue B 40
Figure 2.22 Block Diagram of DMA module..... 41
Figure 2.23 Block Diagram of the MPG unit 41
Figure 2.24 Block Diagram of SWD_B unit 43
Figure 2.25 Block Diagram of SWD_A unit 46
Figure 3.1 Verification tool flow for functional verification..... 53
Figure 4.1 Verification tool flow for Software Emulation 57

ABSTRACT

The creation of a computer system has become a monumental task. Many designers, engineers, and scientists cooperate to create the computer system down to its most basic components. An extremely crucial phase of the design of the hardware sub-systems is the verification of the hardware paradigms and structures that will work in synch to create the new computer system. Therefore, the verification of a system level chip is quite a complex task. Moreover, the verification process in any design is considered a major bottleneck, but it is required to ensure that the number of errors in the hardware designs is minimized. It can be safely said that the complexity of verification increases exponentially with the increase in design complexity [1]. There is definitely a need to use more than one verification tool to test a system-level chip design. Various 'System on Chip' verification methodologies have been developed and are being used in the market. However, these methodologies require anywhere from a medium to a large amount of resources and complex verification structures. A two-level verification methodology has been proposed in this thesis for the multi-core architecture of Cyclops-64, which involves a significant amount of resources. Moreover, it has enough functionality to compete with system level verification methodologies that are available in the market. The Two-Level Verification method involves the classic functional verification and the software emulation. This thesis demonstrates the application of the two level verification methodologies to the inter-processor communication module of the Cyclops 64 architecture.

The bottom-up verification methodology proves to be a very efficient in term of reusability of the test-benches, groups of programs and/or data that is used for verifying the system. Thus, this methodology was a logical choice for the Functional

Verification part of the two-level verification process. Functional verification can be carried out with any hardware simulation tool available, like Modelsim. This type of verification helps in acquiring a detailed knowledge of the system components. At the same time, it makes it possible to perform extensive verification on each of these components. The complexity of a system level design calls for the use (or the creation) of a robust and automatized tool set. Usually, existent tools and a small set of “glue” programs (i.e. programs that will coordinate between different parts of the tool) form such tool sets. The methodology that is being proposed by this thesis will use the above formula. Software emulation provides a set of robust and automatized programs and tools. A typical software emulation tool has a code generator, which is used to convert the component’s code written in a hardware description language to a gate level instruction code in ‘C’; a logical processor, which emulates the component and an automatic test pattern generator; and an output checker to avoid any manual error in verification. This thesis demonstrates the

1. Functional verification for the inter-processor communication module (A-Switch) of the Cyclops 64 architecture.
2. Application of the second level verification methodology-software emulation to the A-Switch module.
3. Combination of the two levels for a full system level verification.
4. Preliminary verification of the A-Switch.

.

Chapter 1

INTRODUCTION

1.1 Cyclops 64 and Multi-Core Architecture:

During recent years, a plethora of new paradigms, and some old ones, has surfaced. Among them, the most accepted paradigm seems to be the multicore technologies. A prime example of this trend is the decision of Intel to go multi-core. The first product of this new line is already available, i.e. the Pentium D. On top of this wave is the Cyclops family of supercomputers. Cyclops 64, which is part of the Cyclops family; is a new generation technology that uses the multi-core architecture. Multi-core paradigm can be considered as a design in which a single physical processor contains the core logic of more than one processor [1]. This type of architecture packs several such processors into a single physical processor. Single-core processors have many disadvantages such as narrow data bandwidth, big gap between CPU speed and memory speed. In a single-core processor about 75% of CPU time is wasted in waiting for memory access results. Even though new technologies have been developed to subdue this problem, point in case Intel® Hyper Threaded technology, this still represents a great problem in today single-core computers. In general, the ratio of cost and performance is very bad for single-core processor architecture. The Multi-Core Architecture has come as a solution to these problems [2]. A multi-core architecture can be considered as a SMP implemented on a single VLSI integrated circuit. The goal of Multi-Core architecture is to allow greater

utilization of thread-level parallelism, especially for applications that lack sufficient instruction-level parallelism to make good use of superscalar processors. It can also be called as Chip-level multiprocessing (CMP) or Chip Multithreading [2]. This chip-level multiprocessing improves the throughput of the whole computer system but it has no benefits for single applications that cannot be parallelized. CMP has a better data locality than regular multi-processor architectures. Moreover, Better communication behavior between processing units saves space and energy.

The Multi-core architecture hence enables a system to run more tasks simultaneously and thereby achieve greater overall system performance. The pictorial representation of the multi-core architecture is as shown in figure 1.1. Each core in this design has its own resources to run without blocking any other core.

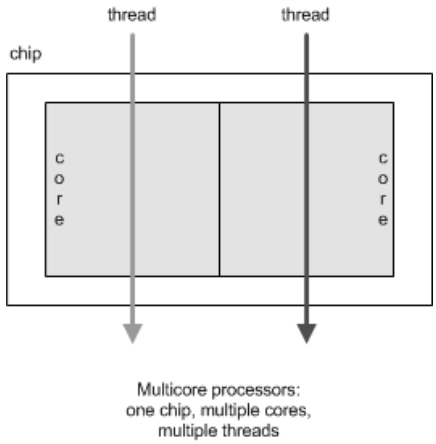


Figure 1.1. Multi-Core processors have multiple execution cores on a single chip
courtesy : Intel [1]

Cyclops-64 is a multi-core architecture which has 75 processors on a single chip. The main idea behind the multi core architecture is “divide and conquer” [1]. The computational work that is to be performed using a single microprocessor is

divided and spread over multiple execution cores. A multi-core processor can perform more work within a given clock cycle, hence delivering a better overall performance.

1.2 Problem Formulation :

A crucial phase of the hardware sub-systems design is the verification of the hardware paradigms and structures that will work in synch to create the new computer system. Verification process in any design is considered a major bottleneck, but it is required to ensure that the number of errors in hardware designs is minimized. Verification is a process used to demonstrate that the intent of a design is preserved in its implementation [7]. About 70% of the time to manufacture a product goes into functional verification. Functional verification attempts to determine if the design will operate as specified. As the design complexity increases the verification complexity also increases [3]. Design sizes is said to be increasing in proportion to Moore's Law. It has been shown that if a design block or a module in a larger system design has a verification complexity of one, then when these blocks are connected in parallel it is said to have verification complexity of two, i.e. it doubles. Similarly if the blocks are connected in parallel and the input of one affects the other then it is said to quadruple. In general, with the increase in the design sizes the verification complexity increases exponentially. An example of this rule is the Cyclops 64 architecture chip. System Level verification of the Cyclops 64 is a necessary evil to make sure that the end product is free of any bugs.

1.3 Proposed Solution:

System Level Verification is the biggest task for the verification industry. There have been various solutions proposed for system level verification by many companies. For example, Cadence has come up with the unified verification

methodology [7]. This verification methodology considers into consideration the digital logic verification to mixed signal simulation verification. For the verification of Cyclops 64, such a complex structure is not required. A two-level verification methodology has been proposed for a system level verification in this thesis, which applies a simpler verification structure. The Two-Level Verification method involves the classical functional verification and the software emulation.

The functional verification is for component level verification. This type of verification helps in acquiring a detailed knowledge of the system components and, at the same time, makes it possible to perform extensive verification on each of these components. The complexity of a system level design calls for the use (or the creation) of a robust and automatized tool set. In general, the automatized tool set consists of some existing tool for simulation along with a small set of "glue" programs (i.e. programs that will coordinate between different parts of the tool). The methodology that is being proposed by this thesis will use the above procedure. Software emulation provides a set of robust and automatized programs and tools. The inter-process communication unit of the C64 chip called the 'A-Switch' was tested with this verification methodology. The 'System Level verification' of the Cyclops 64 was carried out in two steps. Verification can be carried out in either bottom-up or top-down method. In a top-down approach the verification starts with the top-most level. The bottom-up verification technique starts from the low-level blocks verification followed by the verification of the integrated blocks. This is the most common verification methodology used. The bottom-up verification methodology also proves to be a very efficient in terms of reusability of the test-benches, groups of programs and/or data that are used for verifying the system. Thus, this methodology was a logical choice for the Functional Verification part of the two-level verification

process. The software emulation helps in the system level verification method. The block diagram of the verification procedure for both the verification level is as shown in figure 1.2. The main component of the verification process is the Execution unit. It is also called Design-unit Under Test (DUT). The Execution Unit consists of the simulation or emulation or co-simulation process. Verification is a strategy to make sure all aspects of the system meet the required specification and simulation is a tool to attain this. The test plans are generated to meet the specification requirements. A generator is developed depending on the specification. The input to the execution unit is the stimulus file generated by the generator. The results of the design unit is sent to the Response Unit and checked later. This is a generalized procedure of both the methods but the way in which each of the units was implemented varied. The description of the functional verification and software emulation has been explained in the later sections.

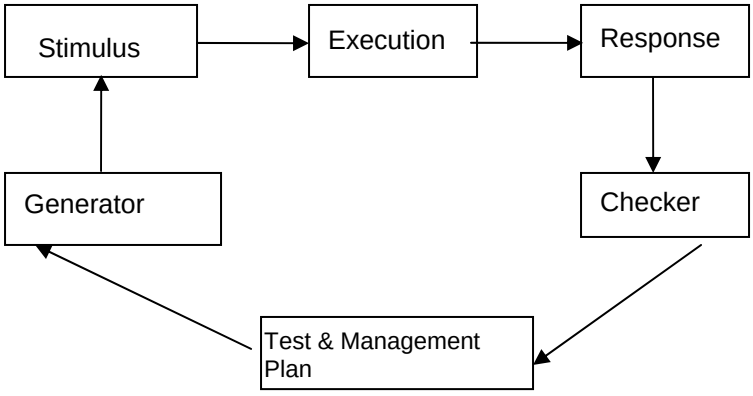


Figure 1.2: Block diagram of the Verification Methodology

Chapter 2

DESCRIPTION OF THE INTERPROCESS COMMUNICATION MODULE IS CYCLOPS 64

2.1 Cyclops 64 Description:

The Cyclops 64 Architecture is designed for high performance supercomputers, which have a performance of petaflop. A C64 consists of tens of thousands of C64 processing nodes arranged in a 3D-mesh network. These processing nodes consist of a C64 chip, external DRAM and some interface logic. Each of the C64 chip has eighty processors. Each processor in turn contains two thread units, two SRAM memory banks of 32KB each. The chip has no data cache, instead it use a part of the SRAM as a scratch pad for this purpose. Such a memory provides a fast temporary storage to exploit locality under software control [2]. Processors are connected to a crossbar network that enables intra-chip communication, i.e. access to other processor's on chip memory as well as off-chip DRAM and the inter-chip communication via input and output ports that connect each C64 chip to its nearest neighbors in the 3D mesh. The intra-chip network also facilitates access to special hardware devices such as the gigabit Ethernet port and the serial ATA disk drive attached to each C64 node. The Gigabit Ethernet links are used to connect the C64 supercomputer with the host interface and the ATA hard drives attached to each C64 node avoid disk bottlenecks and network congestion. There is a separate control network that connects the C64 system to the host system. This control network carries commands from the host system to each C64 node and is connected to the C64 node

via JTAG interface. Figure 2.1 illustrates an instance of the C64 supercomputer architecture with 24 X 24 X 24 logically arranged C64 nodes in the 3D-meshconfiguration.

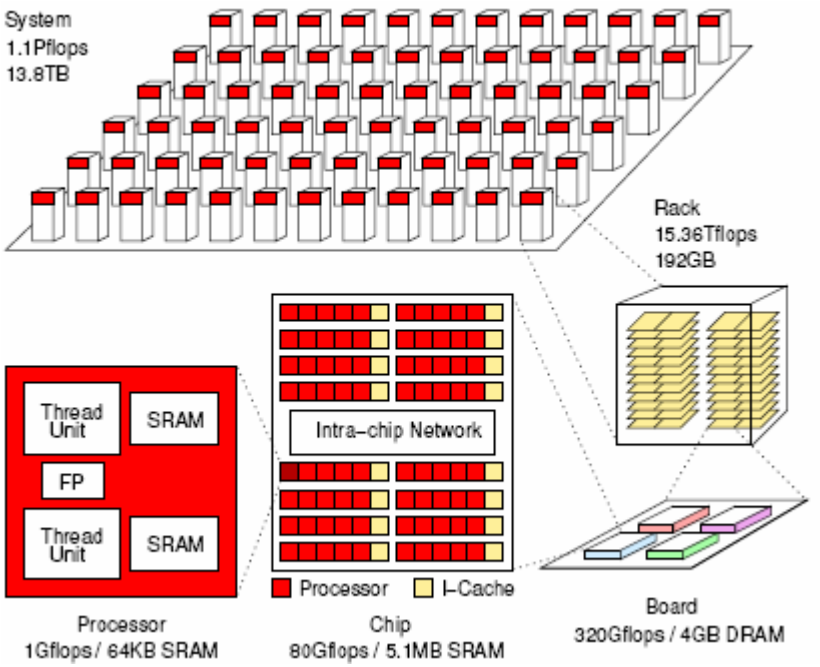


Figure 2.1: Cyclops64 Supercomputer

Essentially, a single C64 chip consists of

- Eighty processing units (or cores)
- One I/O Switch for inter chip communication, i.e. for communicating between two C64 nodes. This module is called the 'A-Switch'
- One 96-way pipelined crossbar switch for intra chip communication, i.e. for communicating with different modules in the chip
- Sixteen SRAM I-Caches, each shared by five processors.

Figure 2.2 gives the detailed connection of these modules in the C64 chip.

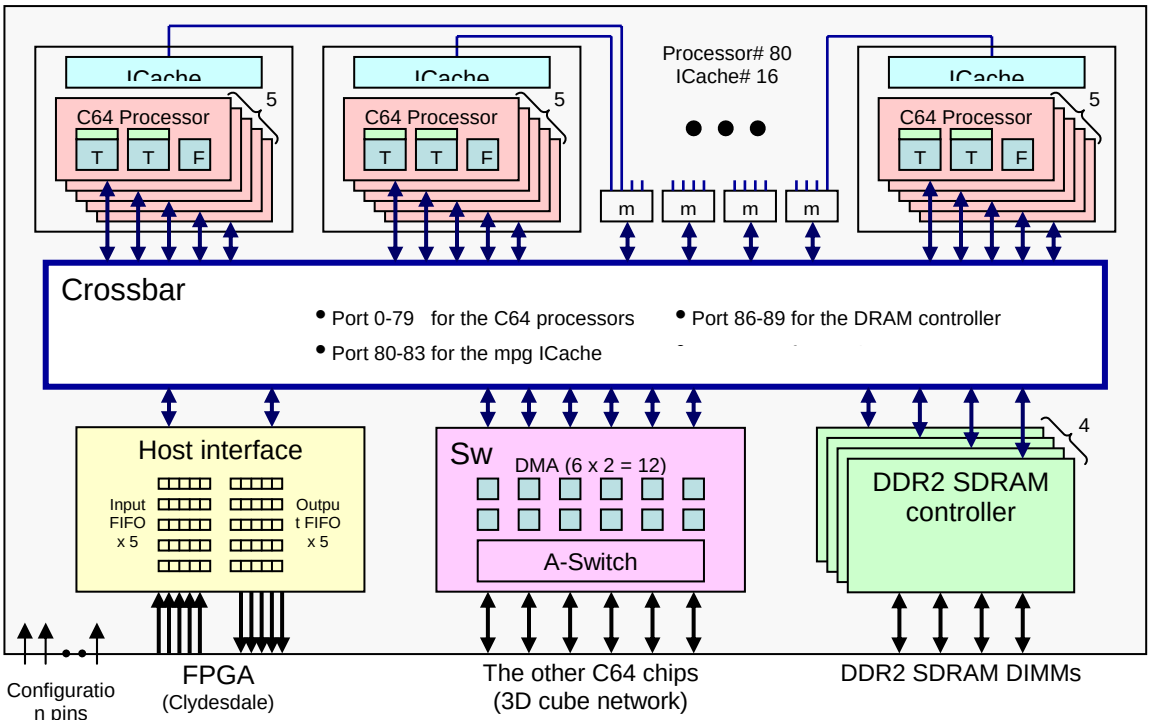


Figure 2.2: Internal connection of the C64 Chip

courtesy: Yuhei Hayashi

2.2 Inter-Process Communication Module(A-Switch)

2.2.1 General Description:

The Inter-chip/ Inter process communication of the Cyclops64 is accomplished with help of the A-Switch Module. The Cyclops64 chip has a point to point connection with its neighbors. A SerDes (Serializer Deserializer) is used to convert the parallel data from the A-Switch into serial data. A SerDes is an integrated circuit transceiver that converts parallel data to serial data and vice-versa. The transmitter section in the SerDes is a serial-to-parallel converter, and the receiver

section is a parallel-to-serial converter. SerDes chips facilitate the transmission of parallel data between two points over serial streams, reducing the number of data paths and thus the number of connecting pins or wires required. In the Cyclops 64 architecture a ‘24’ bit cable is used to connect the two Cyclops-64 chips. The C64 chips are arranged in a 3D configuration, i.e. each chip has six neighbors. Hence an A-Switch module has six incoming data lines from the adjacent chip and six outgoing data lines to adjacent chip. The communication between the adjacent chips is termed as the channel communication. Other than this the A-Switch has six incoming and six outgoing point to point connection with the crossbar. The crossbar forms a means to communicate with the other units of the chip. Since there are separate connections for each of the neighbors, the A-Switch is capable of transmitting six incoming and six outgoing packets simultaneously. The switch is also capable of routing the messages to and from the internal buffer memory. A DMA is used to control the data transfer between the switch’s internal buffer and the processors memory, hence increasing the efficiency of the processor.

2.2.2 Packet Format:

In the A-Switch, message transfer takes place in the form of packets. The software program is responsible for constructing these message packets. The format of this message packet is as shown in figure 2.3. The Packets consists of ‘header’ part and ‘message’ part. The message part may contain one complete message or part of a message. The header and the message are an integral part of the double word. The packet size can vary from two to two fifty five double words. The minimum packet

size is two since a message to be passed has to have one packet for the header and one for the message.

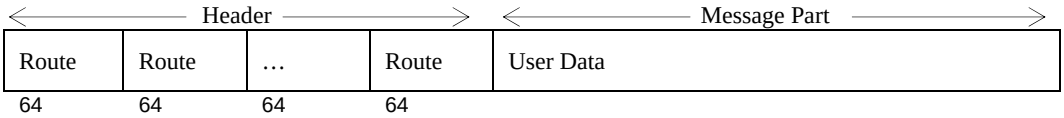


Figure 2.3: Message format in Cyclops 64

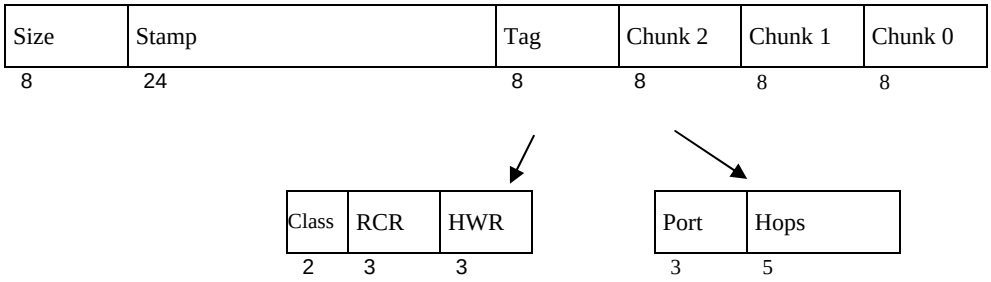


Figure 2.4: First double word:

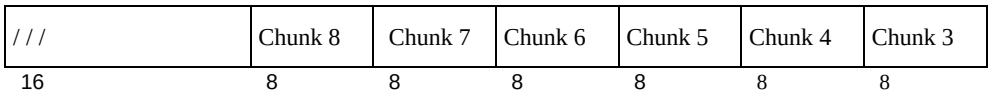


Figure 2.5: Subsequent double words

The header is a sequence of sixty-four bit “route” double word. The word route is used for the header, since it contains the direction in which the packet has to be routed. As mentioned earlier, the A-Switch has six neighbors and the packet can be routed in any of these six orthogonal directions. The direction the packet has to been sent is specified by the ‘chunk’ bits as shown in figure 2.4. The chunk is divided into two parts; the first five bits give the number of hops and the last three give the

direction. The direction can be any thing from 0-5. Each of the six neighbors are assigned a port number depending on the direction in which they are locate. The port numbers of the neighbors are as shown in Table 2.1. The pictorial representation of the port numbers of the chip and its neighbors is as shown in figure 2.6.

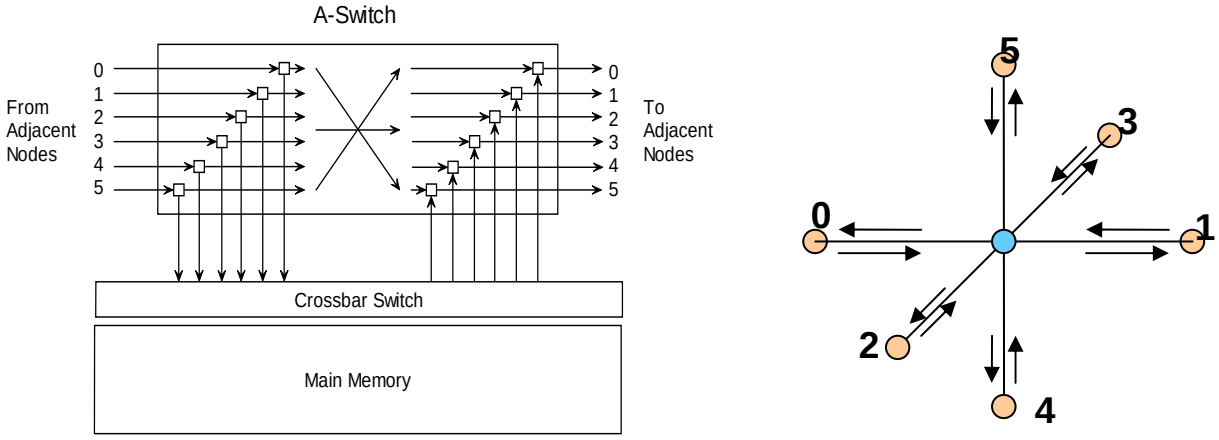


Figure 2.6: Pictorial Representation of connection between the neighbors in C64 chip

Table 2.1: The port number of the neighbors in a 3D mesh configuration

Port No.	In From	Out To
0	$X - 1$	$x + 1$
1	$X + 1$	$x - 1$
2	$Y - 1$	$y + 1$
3	$Y + 1$	$y - 1$
4	$Z - 1$	$z + 1$
5	$Z + 1$	$z - 1$

When a message passes from one chip to another, it is said to ‘hop’ from one chip to another. The ‘Hop’ field in the header gives the number of hops the

message needs to take in the direction given by the value in the direction field of the header. For example if the chunk reads as “010 00100”, it implies that the packet has to hop four times in the negative y direction. Port two represents the negative y-direction. The next field in the header is the ‘Tag’. The ‘Tag’ can be separated into three fields, namely HWR, RCR and Class. The class specifies in which direction the messages has to be passed. When a load store or load request instruction is carried out it is said to be of class ‘0’. In reference to ‘A-Switch’, it is called the forward direction, i.e. the data is being sent to from the processor to the A-Switch. Class ‘1’ is used for messages such as load return, which in term of the “A-Switch’ is referred to as ‘Reverse Direction’. Reverse direction as the name suggest is used for data are sent from the switch to processor.

Messages to be passed can have more than one header word. The number of header that is being passed in a set of messages is indicated by the HWR bit – called the Header Words Remaining is used. The RCR- Rename Chunk Remaining field is used to get around the bad nodes in a network. When the RCR bit is set to zero, it indicates no renaming. When the RCR bit is set to a non- zero value, it causes the chunks to be renamed after the given value. For example consider a direct path x->y->z between processor A and processor B. If there is a bad node between Processor A and processor B then direct path from A to B is blocked. The A-Switch gets around this problem by sending the message through the a different renamed path x1->y1->z1 -> x2->y2->z2. In this message the RCR bit would be one, to indicate the path is renamed after the first chuck. As the message moves from z1 to x2, i.e. after the message has passed through the first chunk, Class 0 changes to Class 2 and Class 1 Changes to Class 3. When the message arrives at Processor B, the original classes are restored.

The stamp field is self explanatory; this field gives the time it was created. The last field in the header is the size field which gives the total number of double words. The size can range from 2 to 255. The next section deals with the flow of these message packets.

2.2.3 Flow of Message in the network:

The software program arranges the message packet in the order of x, y and z direction, i.e. the number of hops in x-direction, if any followed by number of hops in y-direction, if any and finally the number of hops in the z-direction if any. As and when the packets pass through the C64 chip the A-Switch modifies the headers to indicate that the header is one step closer to the destination. A packet has minimum one header with three chunks. The chunks can be a zero value or a non zero value. The algorithm followed by A-Switch in routing the packet and modifying the packets is as shown below.

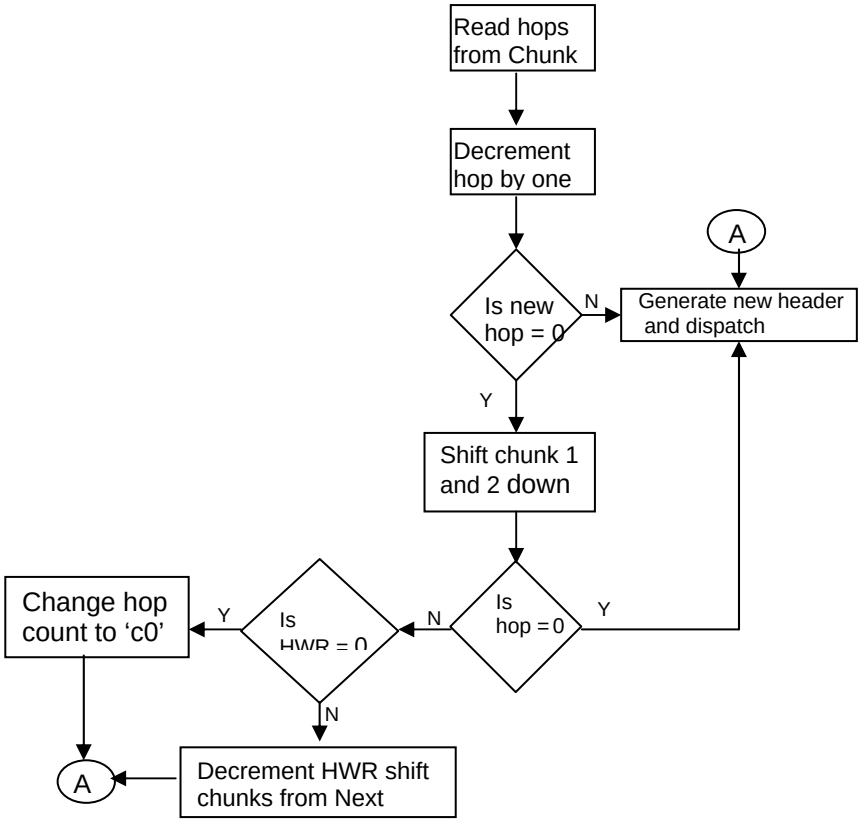


Figure 2.7: Algorithm for changing the message

First Chunk0 is read and the number of hops is checked. If the number of hops is zero, it indicates the packet has arrived at the final destination. If the number of hops is a non-zero value, the hop count is decremented. If the new value of hop is not a zero then concatenate the new value to the rest of the header and send it cross to the next chip. If the new value is zero, then chunk1 and chunk2 are shifted down, i.e. chuck0 is removed. Again the number of hops in the new chunk is checked and if that number is also a zero then it checks the HWR field. A zero in the HWR would mean that there are no more headers and the next chip (node) is the final destination. If the next node is the final destination then the hop count of the chunk is modified to '0C0'. The 'A-Switch' is designed to recognize hop count '0C0' and route the packet to its

processor. Since the Cyclops chip are to be arranged in 24 X 24 X24 grid the maximum hop count can be 24. So making the hop count ‘0C0’ is valid. On the other hand if the HWR is a non-zero value, there are more chunks, so the A-Switch shifts the three chunks from the next header (route double word). If this results in an empty route double word (i.e. the next header is empty), then the A-Switch decrements the HWR bit and sends it across to the next node. This process is repeated until the packet reaches the final destination. The Cyclops chips are arranged in a 24 X 24 X 24 grid. Three chunks are sufficient to route the packet to any one of the nodes in this grid. Provision has been made in case more than two chunks are needed. The flow chart for the algorithm is as shown in figure 2.7. An example of how the packet is routed is as shown in figure 2.8. In the 3D mesh, the packet has to be routed from node A to F and the path is from A -> B -> C -> D -> E -> F. The message has to be transferred to port ‘1’ in the node ‘A’, if the message has to be passed to node ‘B’. From ‘B’ to ‘C’, it is through port ‘1’, since it passes in the positive ‘x’ direction. The number of hops in the positive x direction is therefore two, from node A to B and then from node B to C. The direction for this transfer is specified in Chunk0. Once the message reaches node ‘C’, the message has to go through the positive ‘y’ direction which in through port ‘3’. Again the number of hops is ‘y’ direction is two, from C to D and D to E. This direction instruction is specified in Chunk1. The final chunk gives the details about the positive ‘z’-direction. Only header message has been shown in figure 2.6. The figure also gives a view of how the chunks vary with each hop.

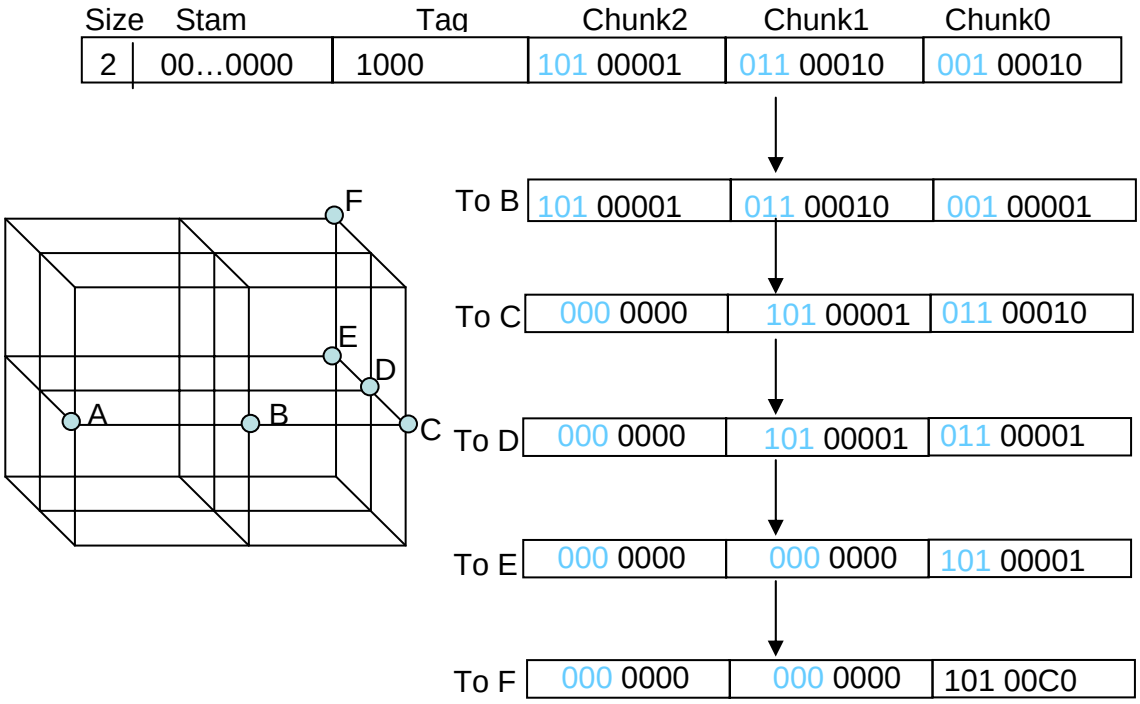


Figure 2.8: Example of packet routing

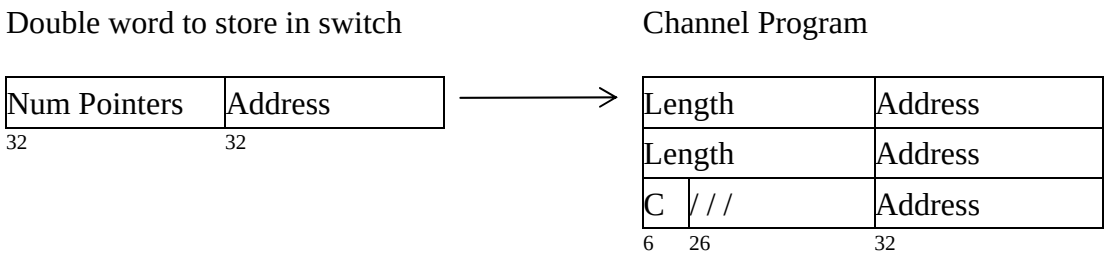
2.2.4 Communication in the A-Switch:

The communication between the processor and the switch consist of mainly the write and read operation. Write operation, also called the output message transfer is controlled by channel programs. In any system the message packets to be transmitted are never located at one place. Parts of the message packets reside in different parts of memory, for example the header can be in one area and the data in another. Usually before transmitting the messages, the headers and data messages are retrieved from their different location copied in a continuous temporary memory location and then transmitted. In Cyclops 64 this process is handled differently. The

channel program creates a pointer to where the message packets are located and controls the A-Switch message transfer with these set of pointer.

The channel program consists of sequence of pairs of length and address as shown in table 2.2. The length specifies the number of double words located in the main memory that have to be written. The location of the memory is given by the address in the address field. Since the message is segmented and located in different location, there will be a sequence of these lengths and address. A Root double word links all these sequences. The Root double word has a 'Num Pointer' field with indicates the sequence length and the address field give the location where the channel program is located. The last double word in the channel program has a special bit number –'C' to indicate the end of the sequence. Once the write operation is initiated the program checks for completion by testing the bit pointed by the last pointer in the channel program. The channel program and data may be in any part of the main memory (DRAM, interleaved, SRAM or scratch pad SRAM). There is no requirement for the channel program, packet segments and completion indicator to be in the same type of memory.

Table 2.2: Channel Program



In a read operation, also called the input data transfer, the message is transferred from the switch to the software defined circular buffer. The program must

first define a circular buffer for each of the input port. The header format of the cross bar unit is as shown in Table 2.3.

Table 2.3: Header Format

Position	Field	Full Name	Notes
101 (94 after Tar stripped)	V	Valid	1 for a valid packet
100 (93 after Tar stripped)	C	Class	0 for forward, 1 for reverse
99 (92 after Tar stripped)	BSE	Block Start/End	1 during the first and last packets of a block transfer
98:92	Tar[6:0]	Target	Identification of one of 96 targets
91:64	T[27:0]	Tag	Tag
63:0	D[63:0]	Data	Data or other info

The tag can be further broken down to various fields. The tag format is as shown in table 2.4. Depending on the tag, the bits zero to sixty three can be considered as data or address. The various interpretations of the data field depending on the tag are as shown in table 2.5.

Table 2.4: Tag Format

Position	Field	Occupant	Notes
R 27: 17			Variable Interpretation
16:10	PID[6:0]	Src Proc ID	Source PID
9:7	TID[2:0]	Src Thread ID	Source TID
6	S	Signed	1 : Signed
5:0	GPR[5:0]	GPR	Source GPR

Table 2.5: Address and Tag Details

Tag[27:0]	Operation	Notes
11 || Sz[1:0] || A[23:0]	Store with Short Address	Data in D[63:0]
10_0000_000 || Sz[1:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Store with Long Address, first packet	First packet carries Address Second packet carries Data PID and TID are for Return, so not used here, nor are Signed and GPR
10_0001|| N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Block Store, first packet, any Address	Address in D[63:0] PID,TID,Signed,GPR ignored Good for up to 32 values since we use N+1 for the Count
10_0010_000 || Sz[1:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Load	Address in D[63:0] PID and TID are for Return
10_0011 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Block Load, subsuming section of Load Multiple within one bank. This itself is not a block transfer.	Address in D[63:0] Signed is ignored Good for up to 32 values since we use N+1 for the Count
10_0100 || UU[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Atomic Load or Store, first packet	Address in D[63:0] Signed is ignored. Pure Store flagged by GPR = 0
10_0101 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Icache ReFill Request	
10_0110 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	DCache ReFill Request	First packet carries Address
10_0111 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	DCache CastOut	First packet carries Address and Mask if Address is short
01_0000_000 || Sz[1:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Load Return	First and subsequent packets carry data PID and Signed (since sign extension done at storage unit) ignored. Sz is needed.
01_0001 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	Block Load Return	PID and Signed (since all items are doublewords) ignored
01_0010 || N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	ICache ReFill Return	First packet carries reflected address. Subsequent packets carry data. Signed, TID and GPR ignored.
01_0011|| N[4:0] || PID[6:0] || TID[2:0] || Signed || GPR[5:0]	DCache ReFill Return	First packet carries reflected address. Subsequent packets carry data. Signed and GPR ignored

A detailed explanation of how these addresses are used will be given later with examples. The first in read or write operation is initialization of the circular buffer. During the software initialization time, the program gives the A-Switch the location and the size of this buffer. Executing a store long address instruction will perform this initialization. The circular buffer can be tested by executing a load double instruction. After processing with the packet, the program informs the A-Switch that that part of the buffer is free and can be used by another packet. Executing a store-double instruction will accomplish this task. The input packet in the circular buffer consists of a single header double word followed by the message part of the packet. Input packets may not be processed in the order of arrival. The channel program obtains the address and length of two or more packets process them in any order and informs the A-Switch about the memory area where the packet is located. The A-Switch keeps track of the memory areas and assures a in-order delivery of the message. The circular buffer can be in any part of the main memory, DRAM, interleaved SRAM, scratch pad SRAM. But SRAM proves to me more advantageous due to its high data handling rates. The read and write operation are later explained in details with an example in section 2.2.8.

2.2.5 Address Scheme used in A-Switch:

The Cyclops64 uses 32-bit addressing scheme. The address format for the A-Switch is as shown in figure 2.9. Here 7^{th} bit is set indicates the direction in which the address is being transfers. Forward direction is indicated by ‘0’ and reverse by ‘1’. The 8^{th} and the 9^{th} bits are used for the command. The different commands possible are

0- Initiate a write. This is used when the channel program is initiated.

1- Define the circular buffer location and size.

2- Get parameters defining and input packet

3- Write job index

The 25th to 31st bit gives the port address. The ports from 90 to 95 of the crossbar are used for this communication with the A-Switch.

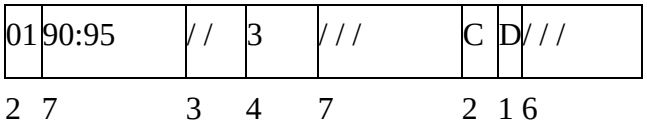


Figure 2.9: Address format of A-Switch

The communication of the A-Switch can be best explained with an example of how exactly the message is transferred. The next section concentrates entirely on the message transfer process, giving detailed description of each of the modules in the A-Switch.

2.2.6 Detailed description of A-Switch:

An overall picture of the C64 chip was shown earlier in figure 2.2. Figure-2.10 gives a detailed connection between the crossbar, the A-Switch and the channel. The crossbar has two latches connected to it, one for the input and the other for the output. The connections between the crossbar and the rest of the modules are carried out through these latches. The ports between ninety and ninety five are reserved for the A-Switch, the other port are shared between the I-Cache, DRAM and the host. The 'A-Switch' takes care of message transfer from and to the other channels. There are some configuration pins used by the A-Switch. These configuration pins are like the

initializing pins which define base and bound values for each of the modules in the A-Switch. An initialization example of the modules is in the appendix. Note the values of the base and bound can be changed. The pins DDPORT, give the location of the 4DRAM devices, which is used in relocation. The bit sizes of each of these signals are given in the figure 2.10.

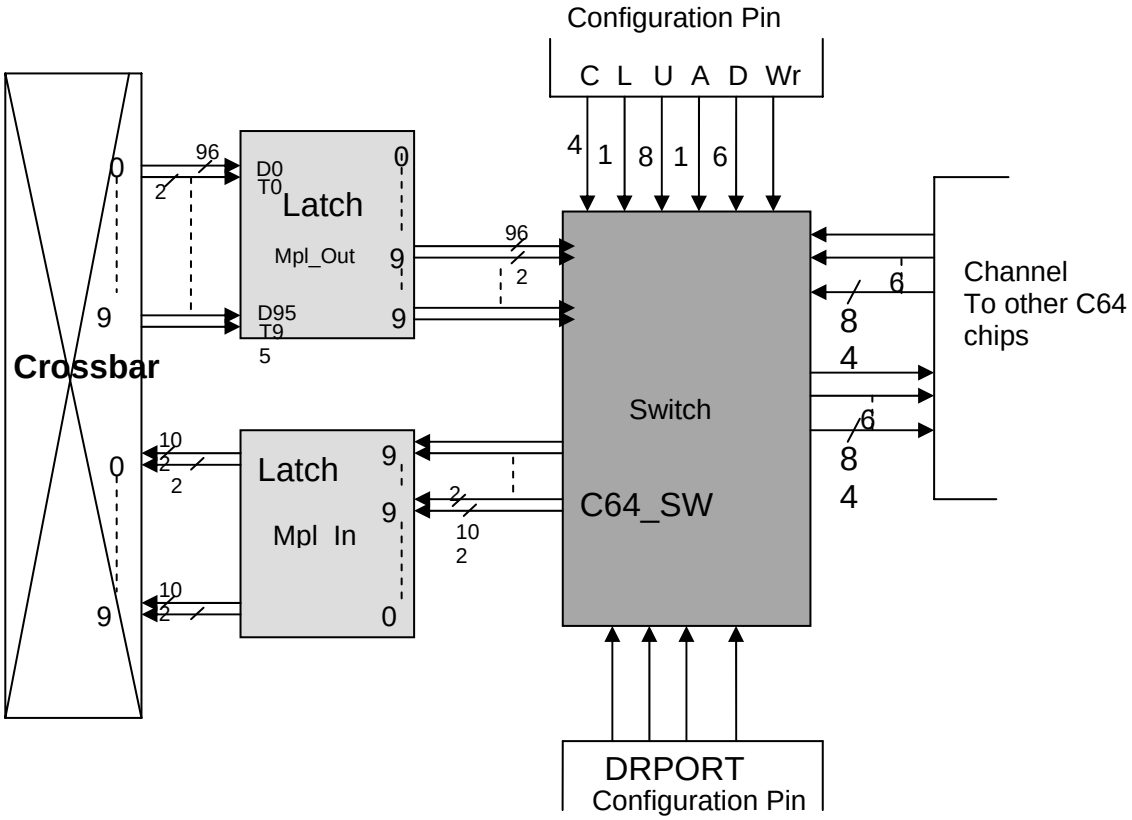


Figure 2.10: Detailed view of the connection between crossbar and Switch

A detailed picture of the A-Switch is as shown in figure 2.11. The entire switch is divided into two main modules; SWA and SW_CS. There are other modules such as the Error Correcting Code Generator (ECC_G) and Error Correcting Code Checker (ECC_C) , Make spare and extract spare.

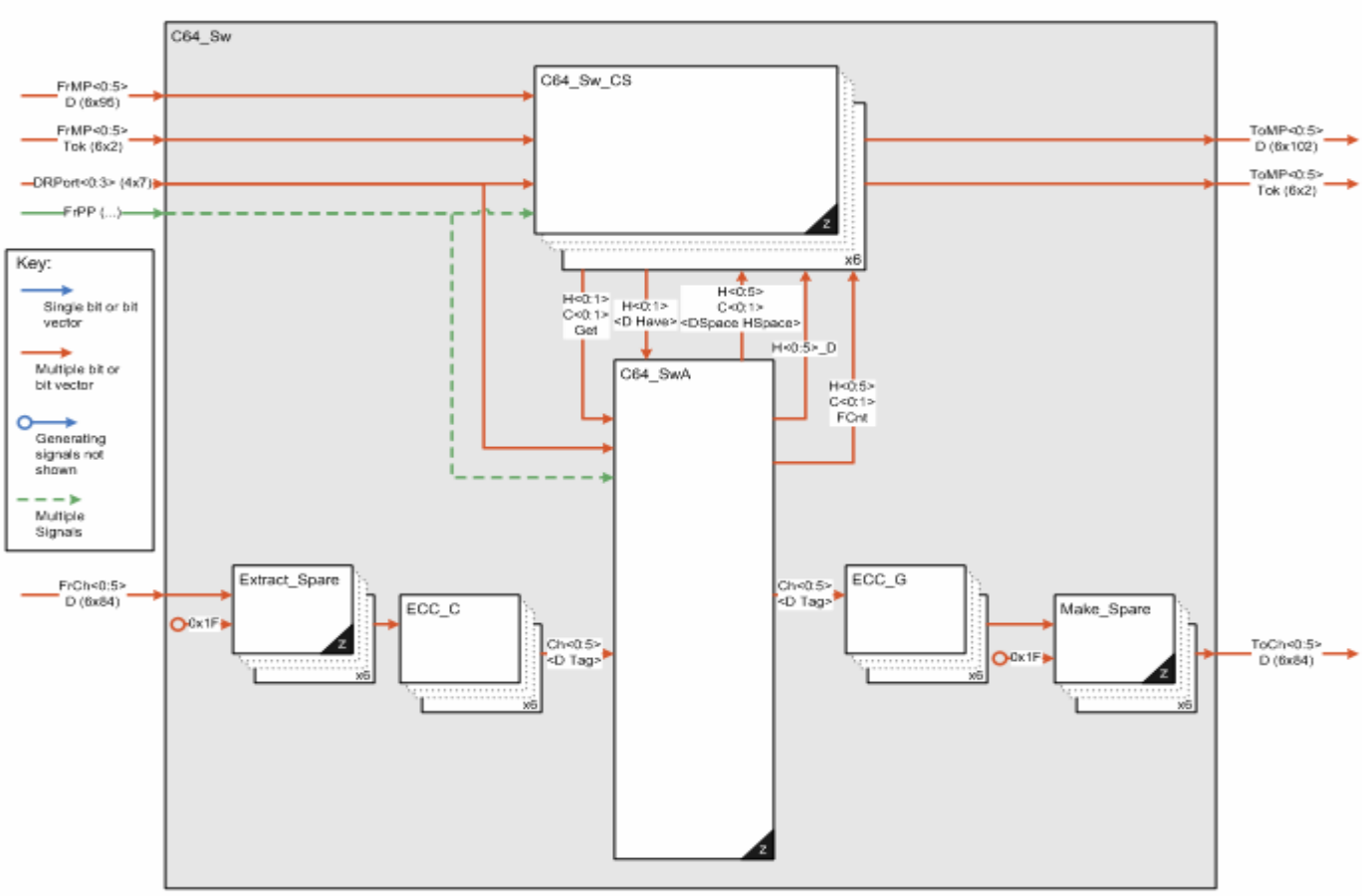


Figure 2.11: Block Diagram of the ‘A-Switch’

The ‘SWA’ transfers sixty four bits of data along with eight bit tag bits. These seventy bits are appended with eight bits generated by ‘ECC’ module, hence mounting to eighty bits in all. A 24 bit cable is used to connect two C64 chips. Of the 24 bits 3 are used for strobes and the remaining 21 bits are used for the message transfer. The SerDes is used to convert the 80 bit parallel message from the ‘Switch’ to serial data message. The SerDes transfers message, by converting four message bits into one high speed bit. Packs of this four bit (one nibble) are transferred in each of the 21 bits connection serially. Hence the total number of data required is for a 21 pin connection is 21X4 = 84 bits. Since the data is only 80 bits the ‘Make Spare’ module

is used to generate the extra four bits. Similarly when the data comes in, an ‘Extract Spare’ module is used to extract the four bits. Data is transferred in these outgoing channels at a speed of 2 Gbit/sec/signal. The logic with which these units create and extract the spare bits is relatively simple. The eighty four bit message is first divided into twenty one nibbles. The nibbles are then fed to a multiplexer with the select line ‘11111’. The spare bits are forced to zero. The block diagrams of the ‘Make Spare’ and ‘Extract Spare’ units are as shown in figure 2.12 and 2.13 respectively. The extracted bits are fed to the ECC_C unit which in turn is connected to the ‘SWA’ unit.

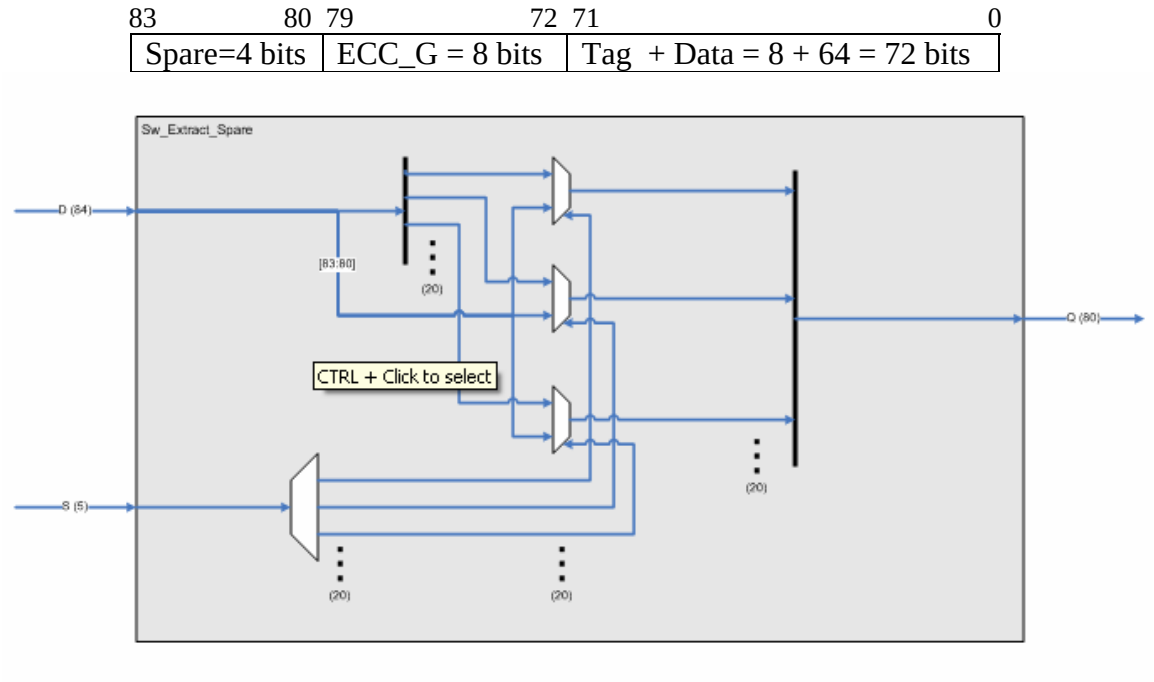


Figure 2.12: Block diagram of Extract Spare Unit

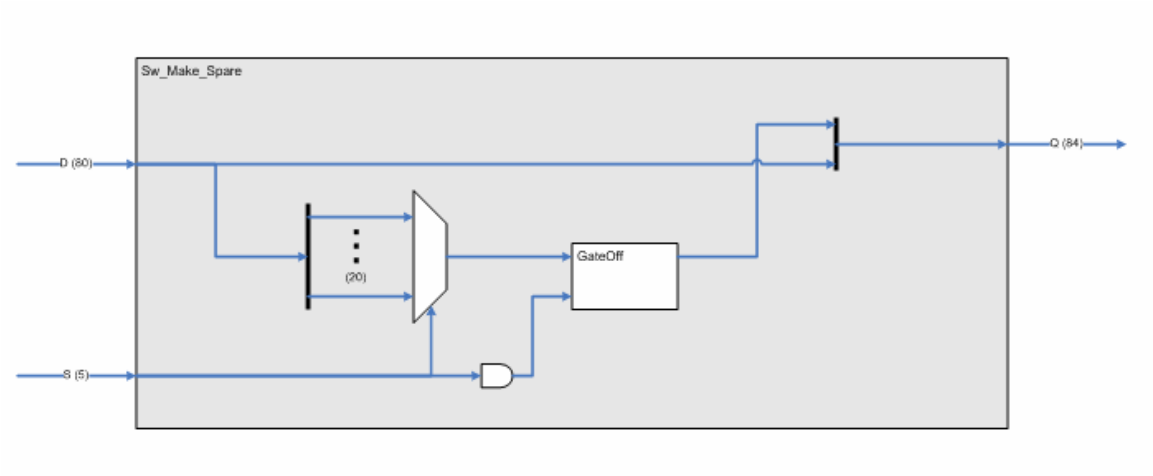


Figure 2.13: Block Diagram of Make Spare Unit

The SWA is the main module which handles the routing to the different nodes. A detailed description of the SWA module is given in section 2.2.7. The detailed description of the SW_CS module; which handles the communication between the switch and the crossbar is given in section 2.2.8.

2.2.7 SWA Module:

The SWA module is quite complex. It mainly consists of five modules as shown in figure 2.14. The inputs from the processor are denoted by FrH<0:5>. The signals are named in certain pattern. Consider the signal FrH<0:5>, FrH implies from header(i.e. processor) and the notation <0:5> implies that there are 6 copies of these. FrH<0:5> is equivalent to saying FrH0, FrH1, FrH2, …, FrH5. The input signals 'FrCh<0:5>' are the inputs from the channel. Similarly there are two sets of output; one for the processor and one for the channel given by ToH<0:5> and ToCh<0:5>.

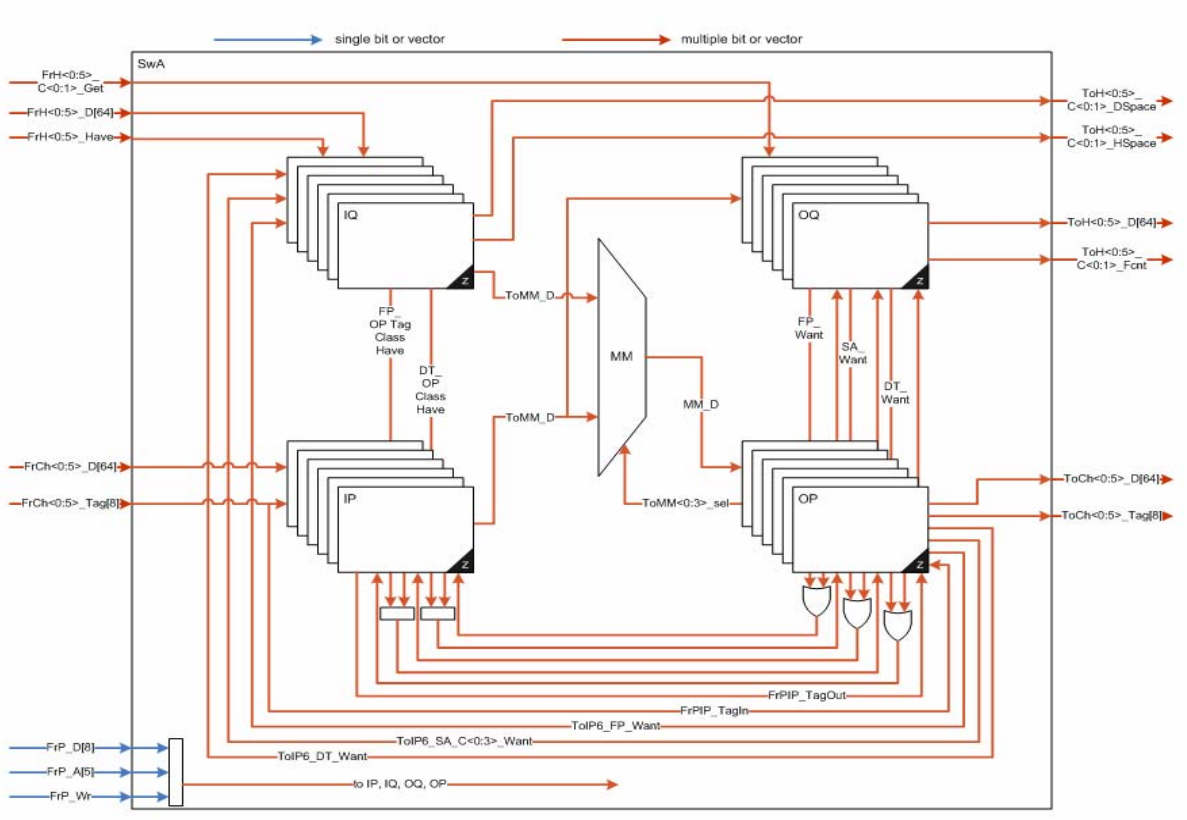


Figure 2.14: Block diagram of SWA module

Message Flow in SWA module:

The message flow in the SWA can be considered to be in two direction; one from the processor to the channel and the other from the channel to the processor. Let us first consider the data flow from the processor to the channel. The message to be transferred is fed to the switch through the inputs FrH<0:5>. The message is fed to any one of these input ports depending on which output post it has to be sent. For example if the data has to be sent through the port 3, then data has to be inserted to 'FrH3_D'. The ports are numbered from zero to five. There are also the tag bit coming from the crossbar, they are represented by 'FrH<0:5>_Tag'. The tag bit specifies if the

message coming in through the input port is valid or invalid. When the tag is '1000 0000', it implies that the data is valid. Please note that the examples to be explained hence forth will use the hexadecimal notation. Other than these there are some configuration pins to the A-Switch that initialize the different modules. When the message traverses from the processor to the channel, it goes through the modules IQ, MM and OP. When the message traverses from the channel to the processor it goes through the modules IP, MM and OQ and the OP module is also used in the selection process. To explain the detailed description of this traversal, an example has been chosen.

First consider the transfer of message from the processor (head) to the channel. The message packets to be transferred are generated by the program, as explained in section 2.2.2. The header that was created in section 2.2.2 has been copied below for convenience.

Size	Stamp	Tag	Chunk2	Chunk1	Chunk0
2	00…0000	1000 0000	101 00001	011 00010	001 00010

According to this header the data has to be transferred through port 1, hence this header and the message part are injected to the input port 1. This is fed to the IQ- Injection Queue module inside the switch.

There are six IQ modules; one for each input ports. The block diagram of the IQ modules is as shown in figure 2.15. The data is sent to the Data FIFO and the 'Sequence-A'. The 'Sequence-A' unit simply receives messages from the Channel, creates the first record for the Header FIFO, and activates the write signal for the DF so that the data which was injected can get into the data FIFO for the appropriate class. This sequencer breaks down the header.

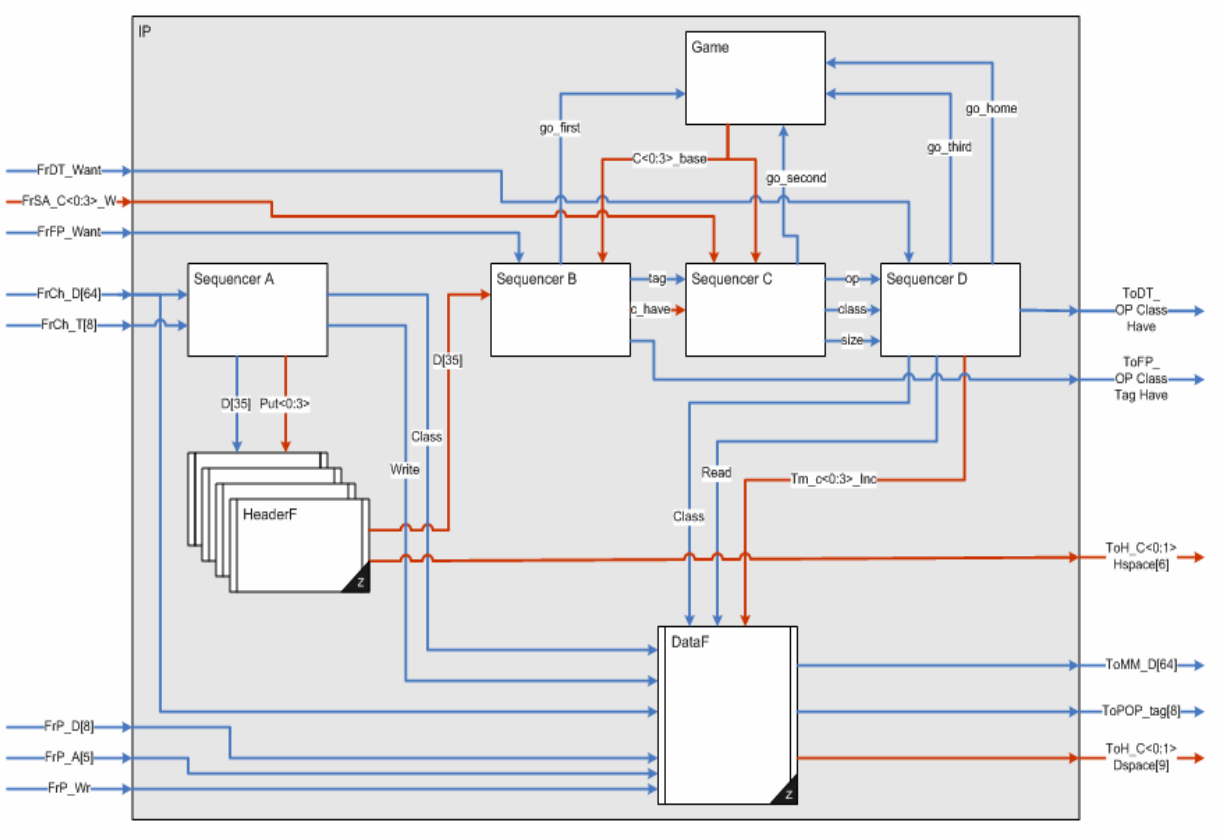


Figure 2.15: Block Diagram of Injection Queue/ Input port Unit

The details about the header have already been discussed. The header contains the size, stamp, tag and chunk field. Here is an example of another header.

Size(63:56)::Stamp(55:32)::Tag(31:24)::Chunk2(23:16):: Chunk1(15:8) ::Chunk(7:0)

04 000000 01 A3 63 43

The Tag can be further broken down into

Class :: RCR :: HWR

00 0000 01

The ‘Seq A’ unit concatenates the port address got from the chunk 0, the size and the stamp and sends this packet to the ‘Header FIFO’-HF unit. The packet sent for this example would be 404000000. The ‘Seq A’ also sends the class information got from the ‘Tag’ to the Data FIFO, which in this case is ‘0’. The ‘DF’ unit writes the message to its register, when the write signal from the ‘Seq-A’ is high. There are different set of registers for each of the class. When the write signal is high, depending on the class the corresponding register is updated. The messages are stored in the registers, so that it can be retrieved later when required, hence allowing pipe-ling, else all transfers would be blocked until the message is delivered to the port.

There are four ‘HF’ units; one for each class, so the ‘Seq_A’ also has a ‘put’ signal to indicate to which ‘HF’ unit the data belongs to. When the ‘put’ signal for the corresponding HF unit is high the packet from Seq-A is stored in the 32-deep FIFO. A record of the port, size and stamp are stored in these FIFOs. The message stored is later released, depending on the request from the Seq-B unit. The message transfer of a particular transfer is broken down into four steps and when the transfer is any one of these steps, it is said to be in that base. The different bases are

0^{th} base: Home Plate – The transfer process is said to be base zero when there is a no data available to be sent for the particular class.

First base: The transfer process is said to be first base when there is a message available to be sent and it waits for a nod from the OP unit to start sending the message. The OP unit has its own logic to determine when it is free to send data of that particular class in the particular port.

Second base: When permission to send the message is received from the OP unit, the transfer is said to be in the Second base and remains in this base until the blocks can be sent. There is a queue for the block transfer. As long as the message transfer is waiting in this queue it is said to be in Second base.

Third base: Once the block transfer starts after waiting in the queue, the class for this particular port moves to the third base. The class stays in this base as long as the block is transferred. Once the entire block is transferred, it moves back to the Home Plate.

The ‘Game’ unit monitors the change in the base and indicates all the other units about the current status of the base. The ‘Sequence-B’ unit checks if the particular class is in zero base. If the class is in base zero and there is a message in the FIFO for the corresponding class, the ‘Seq-B’ indicates the OP module that there is a message to be transferred and set the class to base one. The game unit also receives a notification about this change and announce to other units accordingly. When the OP unit indicates that it is free to receive message for that class, the ‘Sequence-C’ checks if the class is in the first base; if it is the first base, it changes the status of the class to second base and sends the required information to ‘Sequence-D’. The ‘Seq-D’ unit monitors the block transfer. ‘Seq-D’ receives the port address, class and size from ‘Seq-C’. With this information it waits for its turn in block transfer queue. Once the chance to send the message comes the ‘Sequence-D’ changes the class base to third base and sends a read signal to the Data FIFO. The ‘DF’ unit in turn transfers the messages to the MUX module. The ‘Seq-D’ maintains the class in third stages till the entire block is transferred; this is accomplished by decrementing the counter which is loaded with the ‘size’. Once the block is transferred, the class is changed back to Zero base. The changes in the bases status is always notified to the Game unit. It can be seen that the IQ module and OP module work in synchronization to make the transfer possible. The detailed diagram of the Output Port (OP) is as shown in figure 2.16.

Output Port Module:

As the name suggest the OP unit handles all the communication with the output ports, i.e. the channel ports. Each of the 6 IQ’s has its sister OP. So there is one

outgoing data and tag for each of the OP's. The data to be transferred to a particular port is sent to that OP unit. In all there are six OP units. It can be seen from the block diagram that the ports are numbered from <0:6>, instead of <0:5>. Of the seven data path six are dedicated to the six channels and the last data path is used by the IP- Input Port for message transfer for from the processor, i.e. the crossbar to the channel.

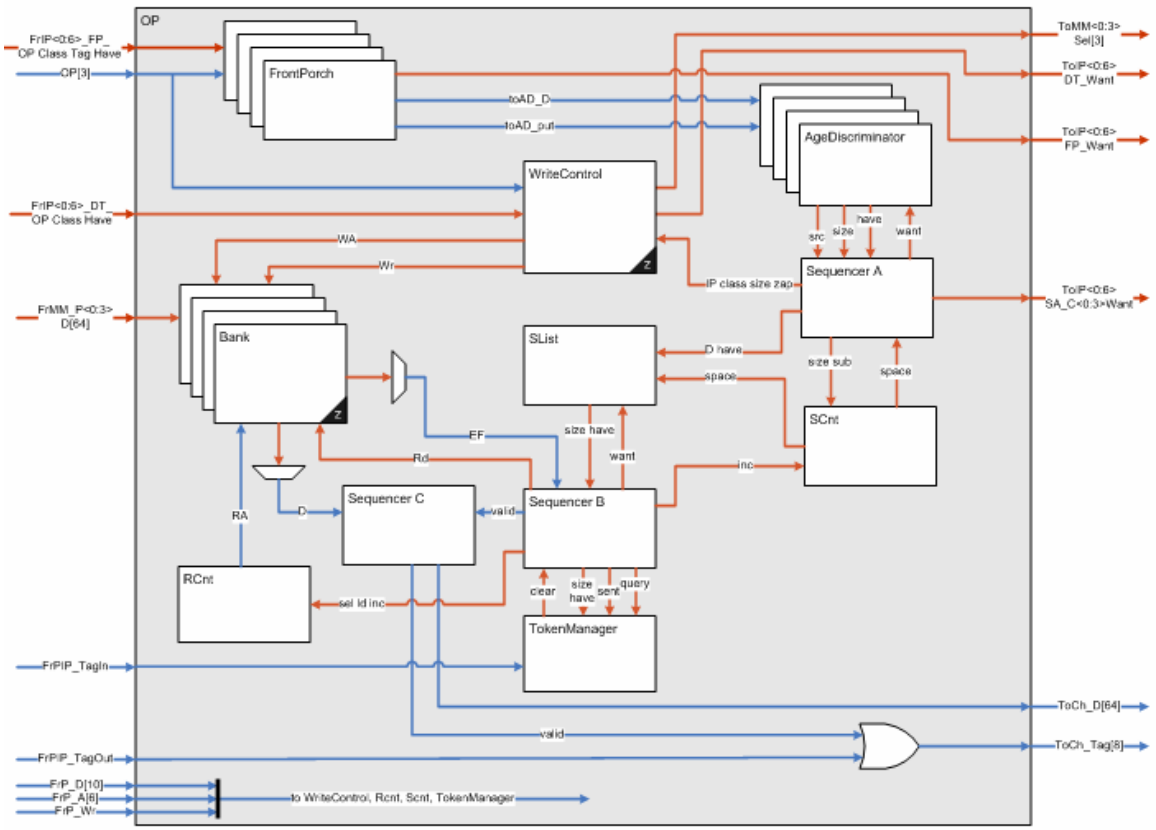


Figure 2.16: Block Diagram of Output Port module

For the data path; processor to channel, the information about the message, like its port number, class and tag are to the Front Porch (FP) unit. When the FP unit is free to take in the next data, it signals the IQ unit. The 'Seq-B' of the IQ unit acknowledges this signal by sending the port number along with the class and tag

to the front porch. There are four front porch unit; one for each class. The info is written to the corresponding front porch unit depending on the ‘have’ signal. This unit checks if the message received is for it’s output port. The FP unit also checks for the corresponding class, if there is a match then source address is decoded, appended with the tag from IQ and sent to the Age Discriminator Unit. The truth table for the address decoder is as shown in table 2.6.

Table 2.6: Truth table of Decoder used in the Age Discriminator Unit

Frip0_have	Frip1_have	Frip2_have	Frip3_have	Frip4_have	Frip5_have	Frip6_have	Source
0	0	0	0	0	0	1	000
0	0	0	0	0	1	0	001
0	0	0	0	1	0	0	010
0	0	0	1	0	0	0	011
0	0	1	0	0	0	0	100
0	1	0	0	0	0	0	101
1	0	0	0	0	0	0	110

As in this example, the message is coming from header so the source ‘110’. This source is appended with the size of the data and sent to the Age Discriminator (AD). The AD unit has seven buckets, so that it can hold one data from each source, so no source will ever use more than one bucket. The AD unit with the help of the value given in the time stamp determines which of the message needs to be taken care of first. The oldest message gets the highest priority. Once the message to be sent is selected, the size of the message is sent to the ‘Sequence-A’ unit of the OP module. The output signals from the four AD units are the ‘size’ of each of the messages, these are fed to one ‘Seq-A’ unit. The OP module has a bank unit where the messages are stored. The ‘Scnt’ unit keeps track of the space available in this bank

unit. When the size of the message is sent to the ‘Seq-A’, the sequencer checks with the ‘Scnt’ unit to see if there is space available for that particular the class. If there is space available, the ‘Seq-A’ commands the ‘Write Control’ unit to create an entry for these set of messages and reserve the space. It also signals the IQ units to send the data. This is when the class changes to second base.

The ‘SCnt’ unit after reserving the space decrements the amount reserved from the available space list. The SCnt unit also enters the new space availability in the ‘Slist’ unit. The SList is needed to determine if the next chip has enough room for the entire message. When an entry is taken from the head of the AD unit, the Size of the message is queued on the SList for the Class. Messages for a given Class will be read from the Banks in order. The Sequence-B determines if the data can be transferred for the given class. Once the message transfer queue is ready for a block transfer, the ‘Seq-B’ sends a ‘want’ signal to the ‘Slist’ unit. On receiving the ‘want’ signal from ‘Seq-B’, the ‘Slist’ gives the information about the size to the ‘Seq-B. ‘Seq-B’ is the actual unit which determines which of the class needs to be sent out of the chip. Sequencer B looks at the heads of the SList and the amount of Space available in the next Chip for each Class, and picks one guy for the next off chip transfer. The Sequence-B unit supervises the transfer. At this point the class is changed from the second base to third base.

The Write Control unit controls on each cycle which input port gets to write to which bank of the output port. The switch supports up to four simultaneous channels assigned to four input ports to write to the same bank. When a transfer from IQ to OP is created, it is assigned to any one of the four channel controllers, depending on which one is free. The channel controller determines on any given cycle

which of the four banks the IQ should be writing to. It maintains the address within the Banks and arbitrates among the other three channel controllers for write access to the bank on the cycle. When a channel wins access the write access the write control unit notifies the IQ as to which channel communication is going ready. The write control unit also counts the items to determine when the transfer is complete. The write address is sent to the Bank.

Another important unit of the OP modules is the 'Token Manager'-TM. The token manager keeps track of how much data space and header space are left for each class in the next chip. These values are used to calculate whether or not the Header FIFO in the next chip has space. Suppose there are 15 message items to be sent, the message are put one at a time to the Header Fifo of the next chip. When the next chip has returned 4 Tokens (16 data items) we know that it has completely processed the message and pulled the entry from the Header Fifo. With the help of the token from the token manger of the next chip, it can be determined if the messages have been successfully transmitted and if there is space to transfer more. Yet another important unit in the OP module is the 'Sequence-C' unit. Sequencer C handles header modification on the way out of the chip. The hop count in the lowest chunk is decremented; the algorithm for checking the header has been explained in section 2.2.3. The 'Seq-C' unit carries out this algorithm and changes the header as required. Finally the message to be sent to the channel is delivered to the ECC_G and make spare unit to get the final version of the message to be transmitted across the cable. So far message transfer between the Processor and the channel was described; now let us consider the data transfer from the channel to processor.

The message which has to be transferred from the channel to the processor will have the hop count as ‘C0’. As explained earlier, when the ‘Seq-C’ of the OP unit checks for the chunks and hops, if the chunks are empty and the new number of hops is zero; the OP unit modifies the hop count to ‘C0’ to indicate that the next switch is the final destination. The channel message is first extracted, i.e. the spare bits are removed and the then the data is checked for error. If there are any error, the ECC_C(error correcting code_ correction) unit retrieves the message and sends it to the SWA module. The SWA module directs these inputs to the Input Port-IP Unit. The Input port unit functions exactly like the Injection Queue unit-IQ. The IP unit sends the information of the OP, class and tag to the OQ unit. The OQ – Output Queue unit is similar to the OP unit but not the same. The detailed block diagram of the OQ unit is as shown in figure 2.17.

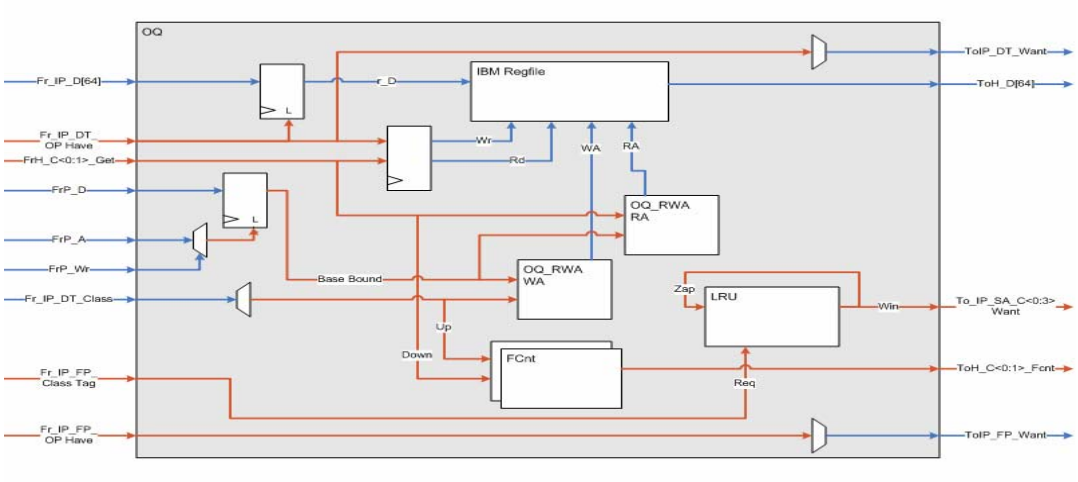


Figure 2.17: Block Diagram of the Output Queue Unit

As seen the OQ unit has relatively lesser number of modules than the OP unit. When the ‘class’, ‘op’ and ‘tag’ and ‘have’ information is received from the IP unit, the OQ unit latches up all these information. The messages are stored in the

register. The register it written into when the write signal goes high. The write signal is set high depending on the port number and the have signal. It has to be made sure that the port number is ‘C’. Once this match is confirmed the class in the IP unit is changed to second base. The IP unit waits for its block transfer to start. When the processor is free to receive data it signals the OQ unit, which in turn signals the IP unit to start the block transfer. The messages stored in the register are retrieved and sent across to the processor. There can be one more form of communication from the channel, i.e. channel to channel communication. When the data is routed from the initial node to the destination node, it is routed through many switches. The form of communication that occurs in these intermediate switches is - channel to channel. In the channel to channel communication, the data transfers from IP- MM – OP. This transfer is very similar to the processor to channel communication. The message from other channels come to the IP unit, which sends the ‘class’, ‘op’ and tag information to the OP unit. The handshake between the IP and OP has been already explained.

The MM units called the ‘Mother of all MUXs’ merely connects the Input message from the Input modules to the output modules. LRU- Least recently used arbitration algorithm is used to select which of the data should be addressed. A more detailed diagram on each of the sub modules can be seen in the appendix. Please note that all the input and output modules occur in pairs. Each input module has its corresponding output module. There are six of these modules; one for each port. This enables the chip to communicate with the six ports simultaneously. The class 2 and 3 are called the virtual class, which are used in case of renamed headers as explained in section 2.2.2.

2.2.8 SW_CS Module

In the previous section the description of the different forms of communication in the SWA modules was discussed. When it was said that the data comes from the processor or goes to the processor, it referred to the SW_CS module. The SW_CS module acts as the bridge between the SWA and the processor. Once Again the description of the SW_CS module will be given with examples. The block diagram of the SW_CS module is as shown in figure 2.18. The SW_CS consists of four main modules- MPG unit, DMA unit (SWD), GlueA and GlueB. The Glue A and Glue B unit are merely glue logic that connects the SWA to the DMA units. Please note the DMA unit is also called the SWD unit, and these two terms can be used in interchangeably.

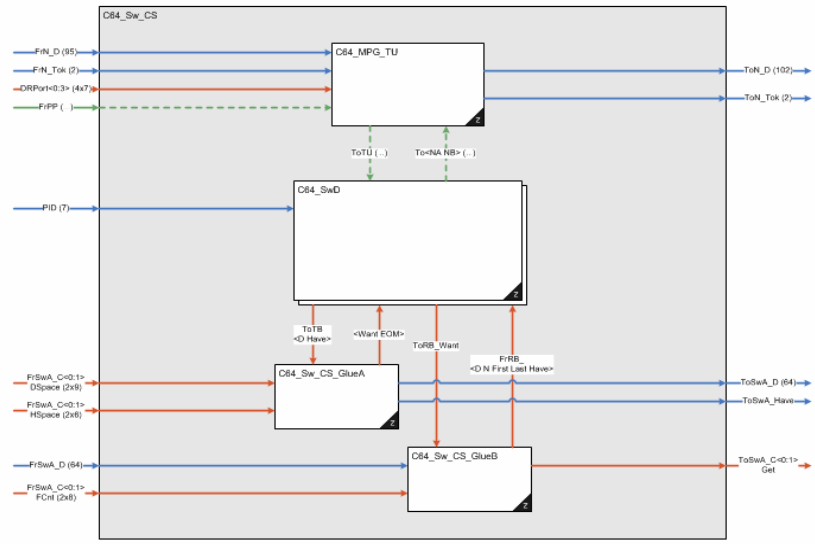


Figure 2.18: Block Diagram SW_CS module

The ‘GlueA’ unit is responsible for the outgoing messages, i.e. from the crossbar to the SWA and ‘GlueB’ for the incoming messages. Each of these units in turn have two internal units; one for each class. The inputs to the ‘GlueA’ are the input

messages from the DMA machines and the space available form the ‘SWA’ unit. The GlueA unit checks if the DMA unit has any message to be sent and checks if the ‘SWA’ needs any message for that class. If there is a match the GlueA unit then checks for the space available, if all the conditions are satisfied, the GlueA unit simply transfers the message to the switch. A blocks diagram of the main GlueA unit and its internal unit is as shown in figure 2.19 and 2.20 respectively. Once the message passing to the SWA starts for a particular class, the GlueA unit does not stop the transfer until it all the messages in a block transfer are sent. It avoids the risk of getting holes in the outgoing message due to interleaved DMA traffic.

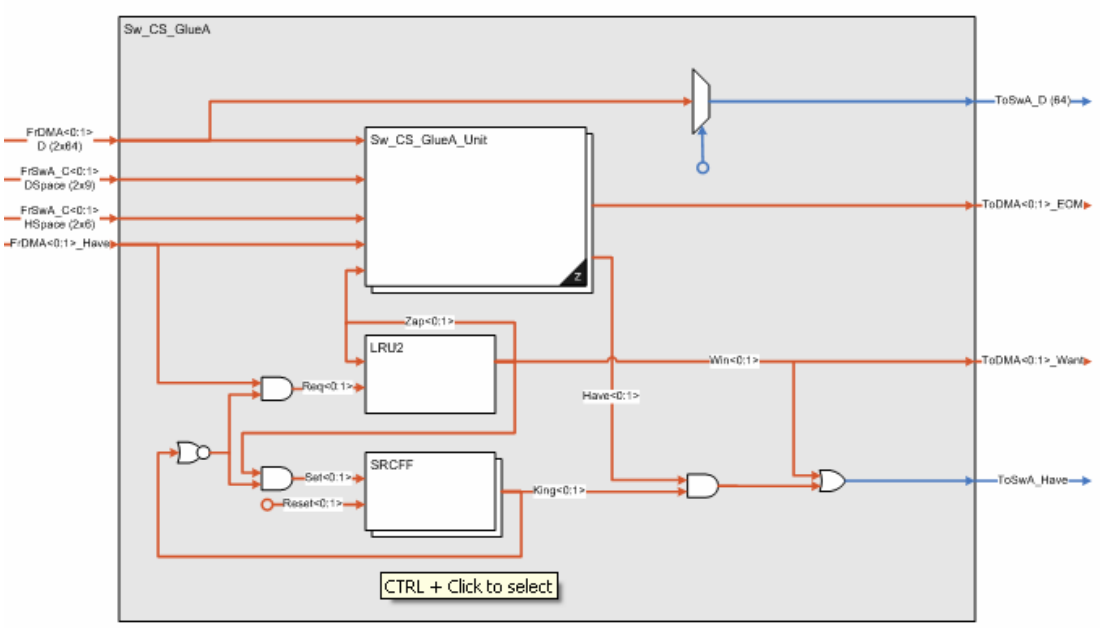


Figure 2.19: Block Diagram GlueA

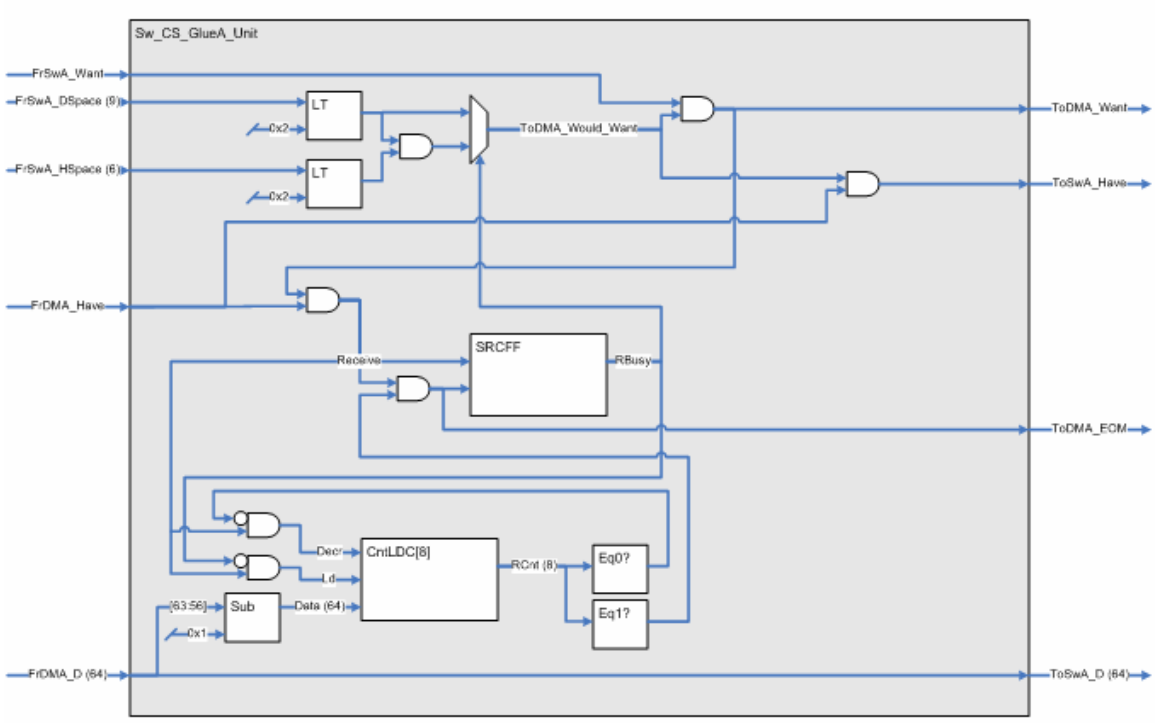
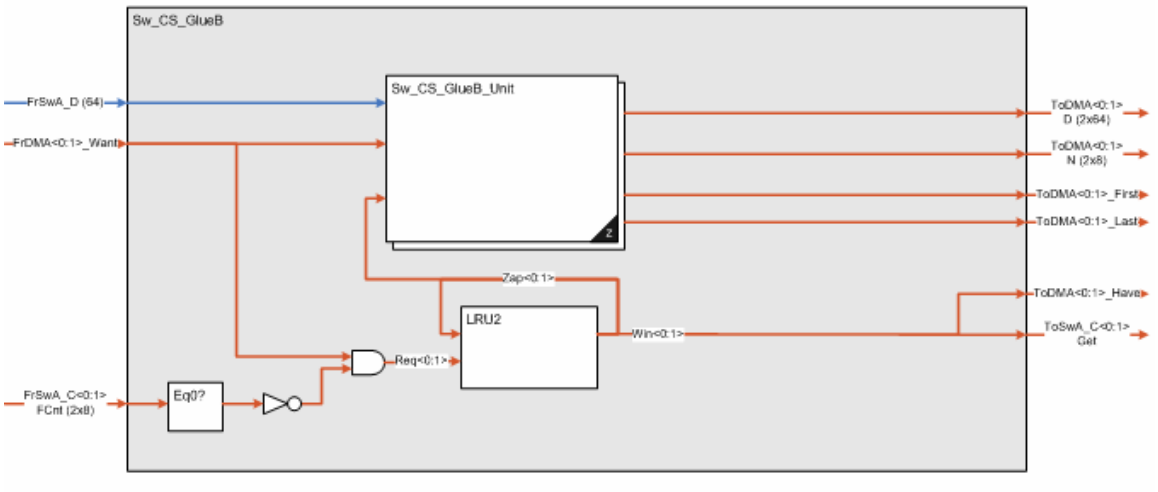


Figure 2.20: Block Diagram Glue A unit

The GlueB is similar to the GlueA unit. The inputs to the GlueB are the message and have from the ‘SWA’ and ‘want’ signal from the DMA machine. The block diagram of the Glue unit is as shown below .



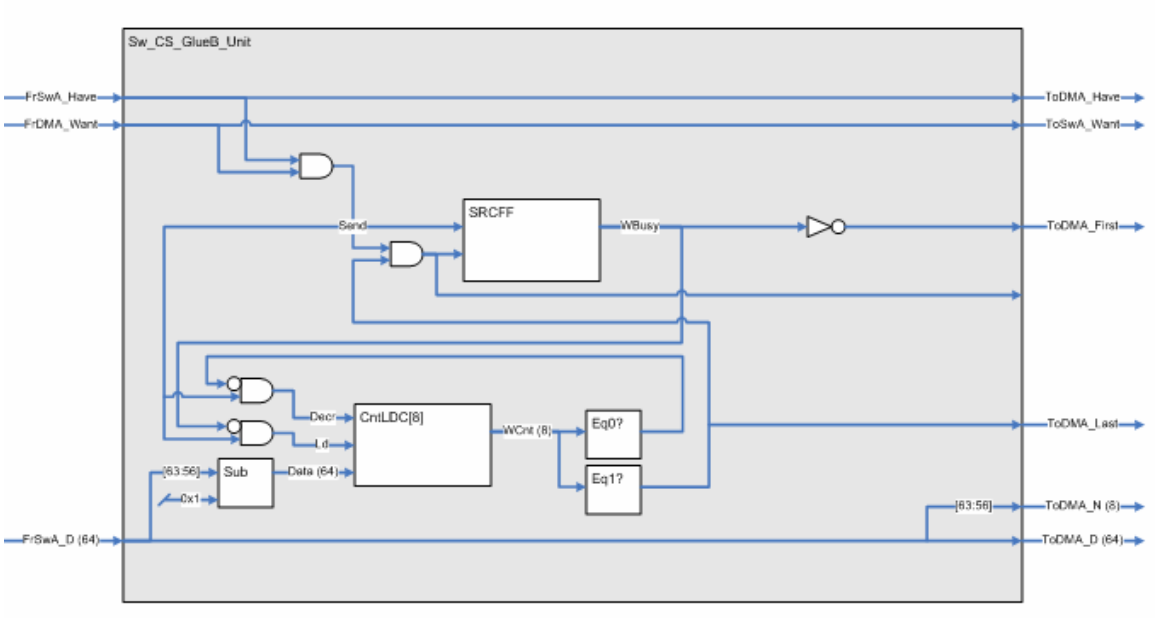


Figure 2.21: Block Diagram of Glue B unit

The message passes from the SWA to the DMA machine and then from the DMA machine to the ‘MPG’ unit. The MPG unit is the glue logic between the Crossbar and the SWD unit. To understand how exactly the message is transferred between the crossbar and the SWA, it is best to describe both the MPG and SWD units together with examples. The block diagrams of the MPG and SWD units area as shown 2.22 and 2.23 respectively.

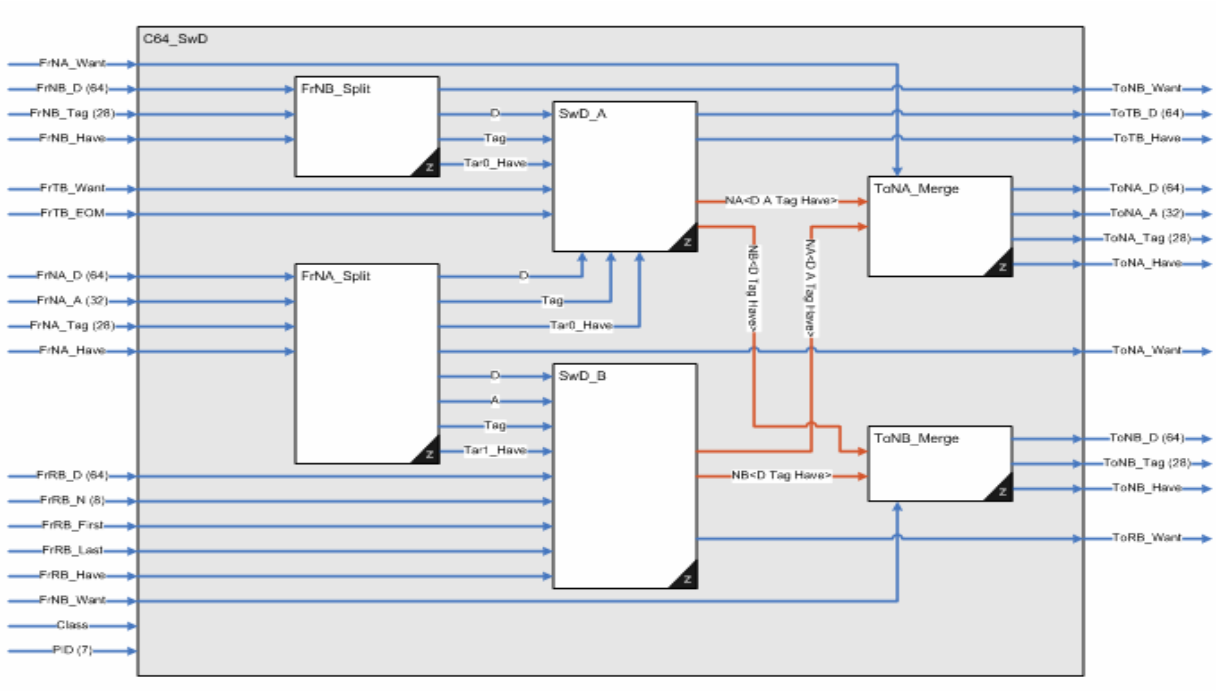


Figure 2.22: Block Diagram of the DMA unit

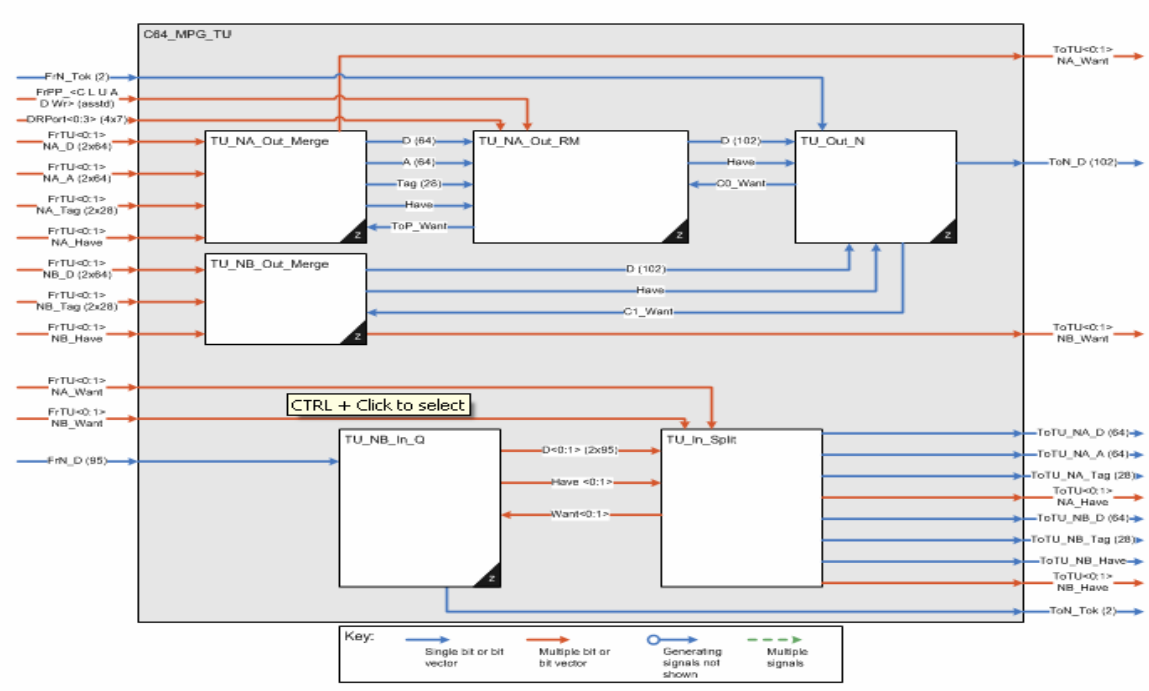


Figure 2.23: Block Diagram of the MPG unit.

The basic concept of message passing between the crossbar and the SWA; i.e. read and write operation was explained in earlier in section 2.2.3. The read operation can be explained in a sequence of operation. The first step in a read operation is defining the read buffer. An example of the header that does the operation of defining the read buffer is as shown below.

58060000_000000006D030080

This header can be decoded with the help of the Tables 2.4 and 2.5. The values of the for each of the value in the table is

Position	Field	Value	Meaning
94	Valid	1	It is a valid header
93	Class	0	The header is for class 0
92	BSE	1	Start block transfer
91:64	Tag	8060000	Explained below
63:0	Data	000000006D030080	Data / Address

The Tag is 8060000 in hexadecimal, which implies the 27 to 17 bits is - ‘100000000110’. From Table 2.5, this value can be interpreted and it represents store long address with the first packet giving the address and the second packet the data. Now when this header is given to the A-Switch from the crossbar, the message is first sent to the MPG unit. The messages which come into the MPG unit are queued in the TU_NB_IN_Q unit, inside the MPG. The IN_Q has two FIFO units; one for each class. The messages are stored in the corresponding FIFO units and stored in till the next unit is free to accept the message. This queue unit also checks if the header is a valid message. The next unit in the sequence is the split unit. The split unit separates the address from the header, decodes the header and sends the data to DMA0 or DMA1, depending upon the on the class.

The address format for the A-switch was explained earlier in section 2.2.5. The address in this header is ‘6D030080’ which is ‘0110 1101 0000 0011 0000 0000 1000 0000’ in binary. The port number is ‘1011010’ that is number ninety in decimal which implies that the data was sent from port ninety. The command bits are ‘01’, which implies ‘define the circular buffer’. The direction bit is ‘0’, hence it is forward direction. Since the header is to be sent in the forward direction the address is sent from the MPG to DMA0 unit, else it would have been passed to DMA1. DMA1 is for class ‘1’. An acknowledge token is sent back to the cross bar by the MPG unit. The DMA again has two sub units as shown in the figure 2.22; one for the outgoing messages and one for the incoming. The unit SWD_A deals with the outgoing message and the SWD_B deals with the incoming message. The header in this example is an incoming message; hence the message is passed to SWD_B. The internal block diagram of this unit is as shown in figure 2.24.

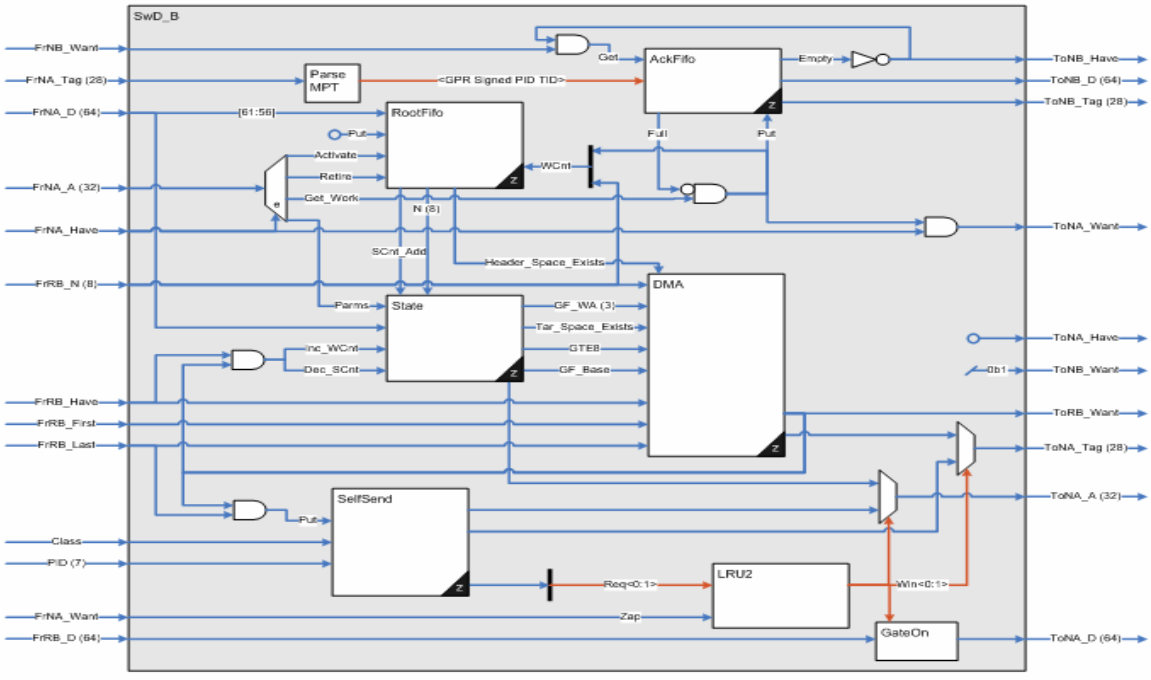


Figure 2.24: Block Diagram of SWD_B unit

When the message comes from the MPG unit is first passed to the FrNA_Split unit. It is this unit which is responsible for deciding if the header that arrived is an1 incoming or outgoing message. The FrNa_Split unit decides whether the message is an incoming or outgoing with the help of the command bits in the address. If the command is to initiate a write then it is said to outgoing message, if the command is for defining any buffers, it is said to be a incoming message. Since the purpose of this header is to define the circular buffer, the address is sent to SWD_B. Now the DMA is prepared to define the buffer. This header can be considered as a command to tell the DMA to be prepared for the defining the buffer. The second packet which is the data packet gives the definition of the buffer. Let the next message be `580600000000003041001000`.

This header again passes through the MPG unit. Since it is store long address, this message follows the same path as it predecessor, arrives at SWD_B. Now SWD_B knows that this is the definition of the circular buffer. The address that arrives to the SWD_B is ‘3041001000’. From this address the first two digits represent the size of the global FIFO and the next eight digits represent the base address. The base address is ‘41001000’. SWD_B decodes this address and finds that the thirty first bit is ‘0’ in this case. The thirty first bit represents whether to use the SRAM or the DRAM for this communication. In case the bit value is ‘0’, DRAM is used, else SRAM. SRAM is usually used for block transfer, but here it is not a block transfer, so DRAM is used. The size and the base are stored in the SWD_B_State unit. The circular buffer is defined with this process.

The circular buffer can be tested by passing a load and return header to the switch. The header of load and return for the above store is 48860408000000006D030100. The Tag in this case is 8860408 which is '10001000011000000100000001000' in binary. It can be decoded form the table 2.5, that this is load and return command. The value for the PID field is '1' and the GPR is '8'. This again follows the same path as the previous header and reaches SWD_B. The address in this case is 6D030100. The SWD_B decodes this and sends back a tag to the MPG unit. The tag to be sent is a formed by concatenating 010000000 with the PID(Processor ID), to indicate which processor, the TID (Target ID) and the GPR(General Purpose Register). The tag formed by this is '4060408'. When this tag reaches the MPG unit; the MPG unit adds few more bits and sends it across to the crossbar. First and foremost, it adds the valid bit, then the class value and lastly the target address. The class in this case would be '1', indicating return. This is basically how a read operation goes about.

The write operation is a little more complex. As mentioned earlier to send a message out of the chip, a processor assembles a channel program in memory. Starting at a Root address is a list of pointers. Each element of the channel program except the last points to an array of data and also contains the number of data items for that array. The last element points to a synchronization location. When everything is assembled, the processor writes the Root address and list size to the DMA engine. If there is room in the Root fifo, a 1 is returned to indicate that the write was successful. If not then a 0 is returned and the processor must try again. The SWD_A is responsible for this outgoing message. The SWD_A has basically six units which aid in this process. The block diagram of SWD_A is as shown in figure 2.25

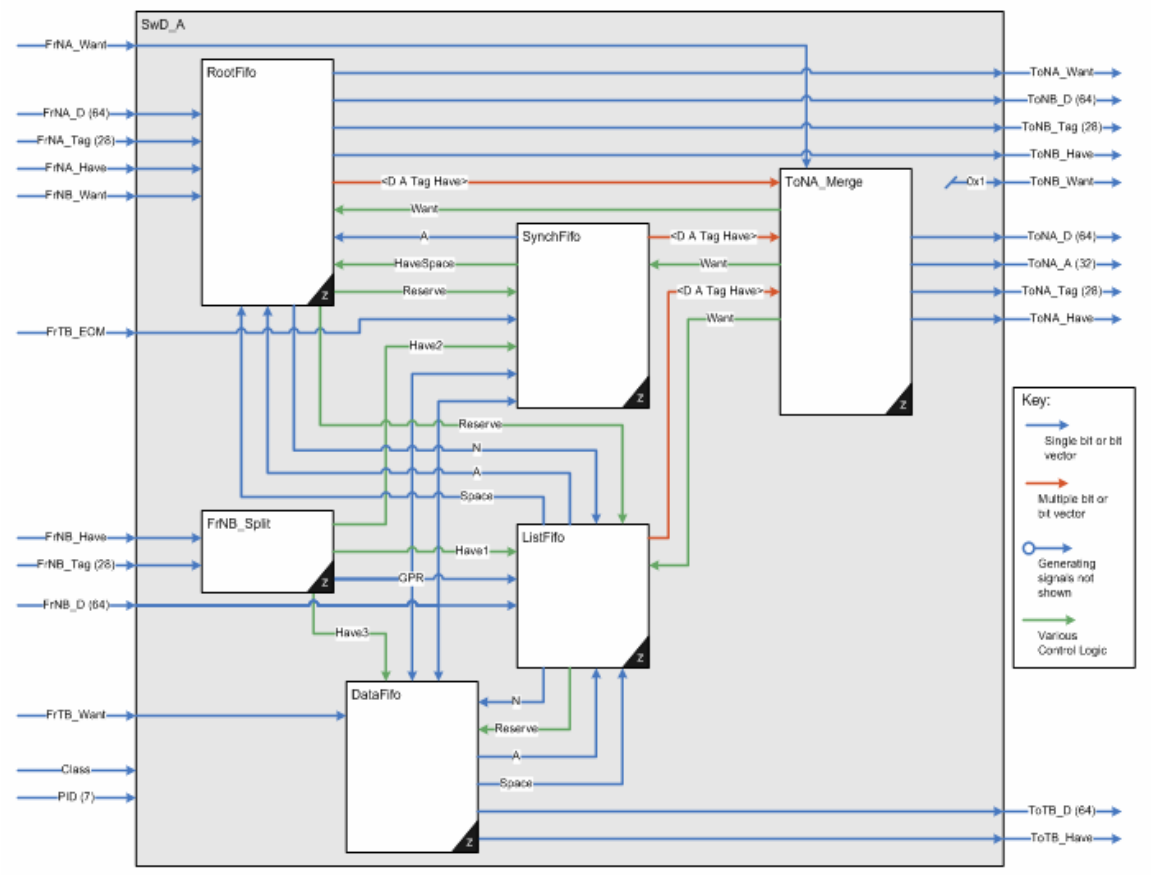


Figure 2.25: Block Diagram for SWD_A unit

SwD_A, first fetches all the list of pointers to the List Fifo except the last one. The last pointer is sent to the Synch Fifo. The List Fifo fetches the data pointed by its members and has the data sent to the Data Fifo. The Data Fifo sends data out of the unit in-order. Remember the data are located in different parts. The FIFOs makes sure that the data is re organized and sent in-order. When the last item has been sent, the Data Fifo notifies the Synch Fifo that it should do an Atomic Store that flips a synchronization bit in storage. The to and fro traffic between the crossbar and the switch in this case is best understood with an example. Consider a situation of sending two messages; one header and one data to the switch. The first step would again be

defining the buffer. The next step would be opening a channel program. This is done with the help of store long command. For example consider the two headers

5806240000000006D130000

5806240000000000348100700

In this case the processor ID (PID) is ‘1001’, TID is ‘0’ and GPR is ‘0’. As usual the first message consists of the address and the second consists of the data. In this case the data is ‘348100700’. The first two digits give the global buffer size, which is equal to thirty four. The remaining digits give the base address which is ‘8100700’. The location is ‘700’ This is still and incoming message, so it is sent to SWD_B. SWD_B decodes this and finds the thirty first bit of the base address to be ‘1’,hence stores it in the SRAM. SRAM is used in block transfers. The SWD_B sends back the Tag information to the MPG unit, and the SWD_A. The SWD_A stores this information to the Root FIFO. The Root FIFO controls all the operation in the SWD_A. With this handshake a channel program is opened. Now the pointers have to be loaded. The DMA make a request to the crossbar to send the list of pointers from location ‘700’, to be stored in the LIST FIFO. The Root FIFO unit takes an entry from the Fifo and issues one or more block loads to retrieve the elements of the list. By means of the return TID, the loads are directed to the particular registers in the List Fifo.

The crossbar accomplice to this request and sends the message to the particular TID specified by the Root FIFO. The message has to be sent to the List FIFO, the message would like

7443698000000000148100740

7443698000000000148100800

The TID indicates to which FIFO this message is targeted for. In this case the tag is ‘4436980’, of which the TID is 011 and this implies that this message is for the LIST FIFO. The last pointer is sent to the Synch FIFO.

64416A800000000348100820

Here the TID is 101, which indicates Synch FIFO. Once again all this is incoming messages, so the SWD_B handles the communication, get the tag and send it to the MPG and SWD_A. The DMA requests for data from location ‘740’ and ‘800’, these locations were got from the header sent to the LIST FIFO. The messages arrive to the data FIFO. The DMA indicates that data is ready and if the switch is ready to accept data, the message is sent to GlueA unit.

So far the communication between chip and its internal memory, chip and the channel has been discussed. Now let us consider the channel to chip communication. The channel to chip communication is relatively simpler. For any communication the circular buffer has to be defined. The circular buffer has to be defined only once. This is more like an initialization of the switch. This initialization defines the base address. This base address is later used in the channel to chip communication. When the message passes from SWA module to SW_CS, it goes through the GlueB first. The Glue_B unit first gets the header message from the SWA unit. Depending on the size given in the header, the Glue_B collects all the data for one set of messages. The Glue_B later send these messages to the SWD unit and indicates which of these messages are first and which one the last message. The message is sent to FrRB_N of the SWD_B unit. The data is stored in the SRAM, since it is a block data transfer. The header message is stored in the location given by the base address. The address is then incremented by eight locations and the consecutive

data or header message is stored in this location. This process repeats for all the messages. These addresses are sent to the MPG unit. The MPG unit retrieves the messages from these circular buffer, calculates the actual address with the help of the Re-Map unit and sends it across to the crossbar.

Channel to channel communication is the simplest of all. It does not involve any DMA unit. The entire communication is handled by the SWA alone. This kind of communication has already been explained in section 2.2.7. Hence a description of the types of communication that are possible in the A-Switch has been discussed.

2.3 Verification Required for A-Switch:

The A-Switch is said to be fully functional if it can transfer messages from crossbar to channel and vice versa and from crossbar to the internal buffers. The test-bench is generated in such a way as to test all the forms of communication. Functional verification helps in determining each modules of the A-Switch, hence giving an extensive insight of the sub modules in A-Switch. Each and every modules is tested for all the possible paths the modules can achieve. Functional verification hence provides a detailed analysis of the ‘A-Switch’. To test the ‘A-Switch’ along with other units the Software emulation process is used.

Chapter 3

FUNCTIONAL VERIFICATION

3.1 Introduction:

Functional verification is a verification methodology that focuses on the design and implementation of the system and components before they are built [6]. The goal of functional verification is to prove that a design will work as intended. The first step in functional verification is to determine what the intent is [6], i.e. to get a good understanding of the specification. Second step would be to determine what the design does and then compare this with specification to ensure they match. Functional verification requires that several elements like the stimulus the design unit and response checker and many more to be in place. It relies on the ability to simulate the design under test (DUT) with a specific input stimulus, observing the results of that stimulus on the design and deciding if the results are correct. The block diagram of the simulation environment is as shown in figure 1.3.

A simulation based verification environments are almost always built upon a set of structured elements. In a structure based simulation environment the design is partitioned into set of functions that allow the overall complexity to be broken into manageable parts. There are many advantages of this approach in a very complex project like Cyclops 64. One of the main advantage is that the interactions between the sub-modules can be easily managed and viewed. In a structured verification methodology the test benches and blocks that were used to verify the sub

modules can be reused to test the higher modules. Verifying from the lowest sub modules to the top most modules also helps in better understanding of the functionality of the each module. A structured simulation is equivalent to bottom-up verification methodology. There are various verification tools available. Simulators are the most common and familiar verification tool used. They are called simulators since their role is limited to approximating the reality [6].

Simulators are not static tools [7], i.e they cannot perform the verification without additional information/action from the user. The additional information required in this case are the stimulus generator and checker. A stimulus is required to provide the inputs, so that the simulator can emulate the design's responses based on its description. A checker is needed to validate the outputs of the simulator against the design intent.

The test cases for the verification of the Cyclops 64 have been designed so as to cover all the possible values. An ideal test case should make sure the every single statement of the design code is executed. This is not possible for a very complex code structure. Breaking down the complex code into different structural elements and then testing each module again proves to be advantageous. There are various simulation languages available such as VHDL and Verilog. Simulation language is different from verification language. Hardware Verification Languages (HVL) can automate verification. Some of the verification languages are OpenVera from Synopsys [16], RAVE from Forte [17], C++, Perl, TCL and many more. Verification languages are more useful when the entire system that includes both hardware and software of a system are to be verified. But for the verification of the Cyclops 64, which is still in design stage only hardware simulation is required; hence simulation languages are to

be used. The HVL languages are used in the software emulation method described in section 4.1.

3.2 Tool Flow of Functional Verification of A-Switch:

The simulation languages used in the verification of the A-Switch module are VHDL and 'KSM'. 'KSM' is a proprietary hardware description language by IBM. The entire design of the Cyclops 64 has been written in 'KSM'. The 'KSM' code can be simulated with 'Delphi'. Delphi is a powerful Integrated Development Environment (IDE) used primarily to build client/server applications for Microsoft Windows, with an emphasis on databases [7]. Delphi was developed by 'Borland International' and is based on Object Pascal. It applies the object-oriented concept and was designed to give developers the ability to build applications easily, with minimal coding required. A tool has been created by IBM under the Delphi environment which converts any code written in 'KSM' language into Pascal code, simulates it and generates the output in a text format. The output is stored in a text file with an extension - 'KSP'. The tool is also capable of converting the KSM code to VHDL. Though the KSM simulation is sufficient to verify the functionality of the modules, VHDL simulation was used due to its familiarity and ease in checking the output. The outputs in VHDL simulation are in the form of waveform and in case of KSM simulation it is a text file. Modelsim was the hardware simulation tool used for VHDL codes. The functional verification was carried out with the simulation languages, KSM and VHDL. The tool flow for these simulations is as shown in figure 3.1.

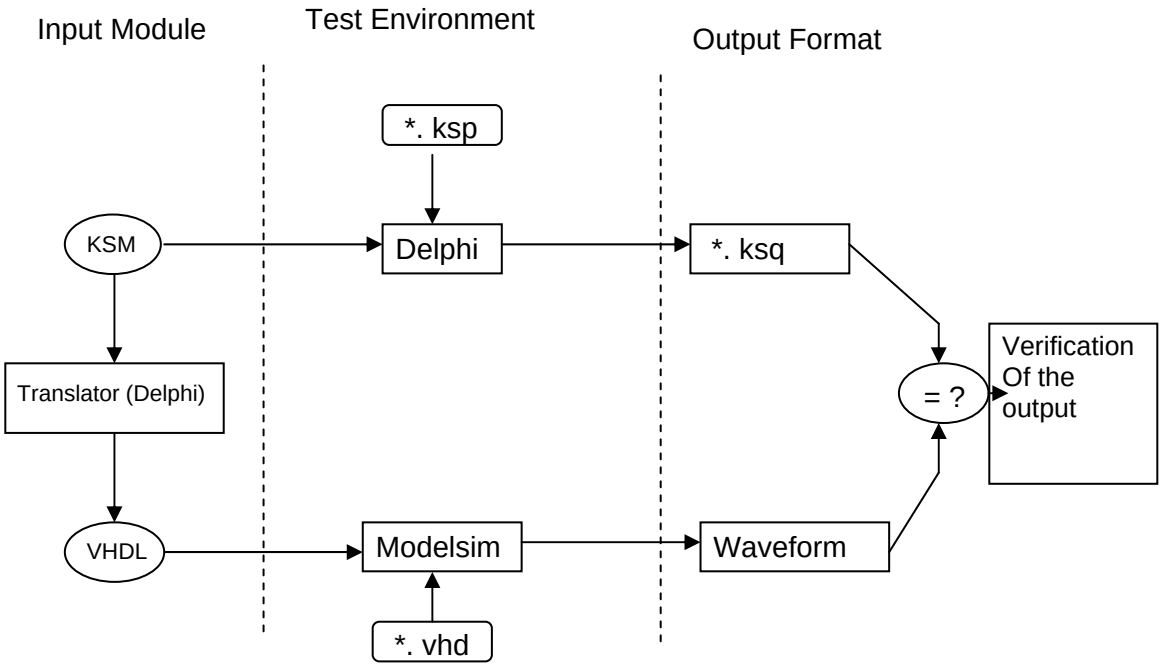


Figure 3.1: Verification tool flow for functional verification

Since the simulation is carried out in two different languages simultaneously it can be compared to co-simulation [7]. The outputs of the both the co-simulation where checked manually and they are found to be similar. The test bench for the 'KSM' tool is called a shell file and for the 'VHDL' tool it is just like another VHDL file. An example of the shell file is given in appendix B. The input pattern for the 'KSM' and 'VHDL' are text files. Using file input and output commands, the VHDL test-bench is capable of reading from the text file. The same set of inputs was given to both the simulation and the outputs were found to be the same. To make sure that the entire code is covered during verification of the inter-processor communication module -'A-Swtich', the 'A-Switch' had to be connected in a 3D mesh configuration. Connecting more than two 'A-Switch' modules and then test

them for inter process communication is not possible in the Modelsim simulator due to various limiting factors like memory. Where as using the Delphi Simulator tool, the switch modules can be connected in a 3D-mesh configuration and tested for inter process communication very easily . Since it is possible to test the entire module in 'KSM', the question may arise towards the need of the verification using VHDL; the answer to this 'ease to check the output'. In KSM it is very difficult to check the intermediate signals, where as in the VHDL simulation it very easy. Taking advantages of both these simulation the functional verification can be successfully completed.

Chapter 4

SOFTWARE EMULATION

4.1 Introduction:

Functional Verification is a necessary verification procedure to ensure the system is functioning as intended but is not sufficient. For an integrated circuit design such as the Cyclops 64 the functional verification of the entire chip is not possible with the simulation procedure. As seen in chapter 3, hardware simulation is not possible for more then two modules connected to each other, consider the entire C64 chip which contains eighty processor, i.e. 160 thread units, sixteen SRAM, one 96-way crossbar, a DRAM and the 'A-Switch'. It is almost impossible to test all these modules together using simulation verification methodology. Even with the 'KSM' simulation tool modules of this complexity is not possible to verify. There is definitely a need use a different verification methodology which can achieve the system level verification.

The most common 'System on Chip' verification methodologies available in the market are a combination of one or more tool set. Along the similar line a tool set was developed which is called 'Software Emulation Tool Set'. Emulation is commonly referred to as duplication (provide an **emulation** of) of the functions of one system with a different system, so that the second system appears to behave like the first system [7]. The basic principle involved in this software emulation tool set is to convert the 'KSM' code to basic binary instruction and then emulate these binary instructions. The tools used of this are the stack code generator and the logical

processor. The stack code generator creates the binary instruction and the logical processor emulates the instruction. The logical processor simulator has been coded in C hence gives an added advantage of flexibility and extendibility. A detailed description of the Software Simulation Tool Set is given in the next section. This tool set will also aid in the unified verification of the Cyclops 64, when the software and hardware are to be verified together.

4.2 Tools Set for Software Emulation:

The Software Simulation tool set is as shown in figure 4.1. The input to this tool is a file with the extension of 'KSF', which is given to the Stack code generator. The 'Delphi' tool flow for the KSM code was explained in section 3.2. The tool other then generating the VHDL and Pascal code, also generates the 'KSF' file. The KSF file contains the code in the form of instructions. An example of a KSF code is given in appendix C. The KSM language has some forty primitive instructions. The KSM design of the modules are converted into these primitive instructions and stored in the 'KSF' file. These instructions are later converted to Pascal code and executed in the Delphi simulation tool. In a similar fashion the 'Stack code Generator' in the Software Emulator developed converts these instructions into more basic instruction. For example a 2X1 MUX is considered as a primitive instruction, it would be better if these instructions can be further broken down to there most primitive more like 'AND' and 'OR' gates. The stack code generator converts the instruction into their most primitive instruction set. The Delphi tool set is capable of executing the instruction of variable width whereas the stack code generator converts the variable width instruction to a fixed width. The code generated by the stack code generator can

be used in both software simulation and hardware simulation on the FPGAs. This object file, given by the extension '.stk' is later fed to the logical processor simulator which is capable of emulating any logic. The output of the logical processor simulator is again a text file. The stimulus file is generated by the generator and the output file is checked using a checker. The generator and checker can be written any HVL like 'perl'.

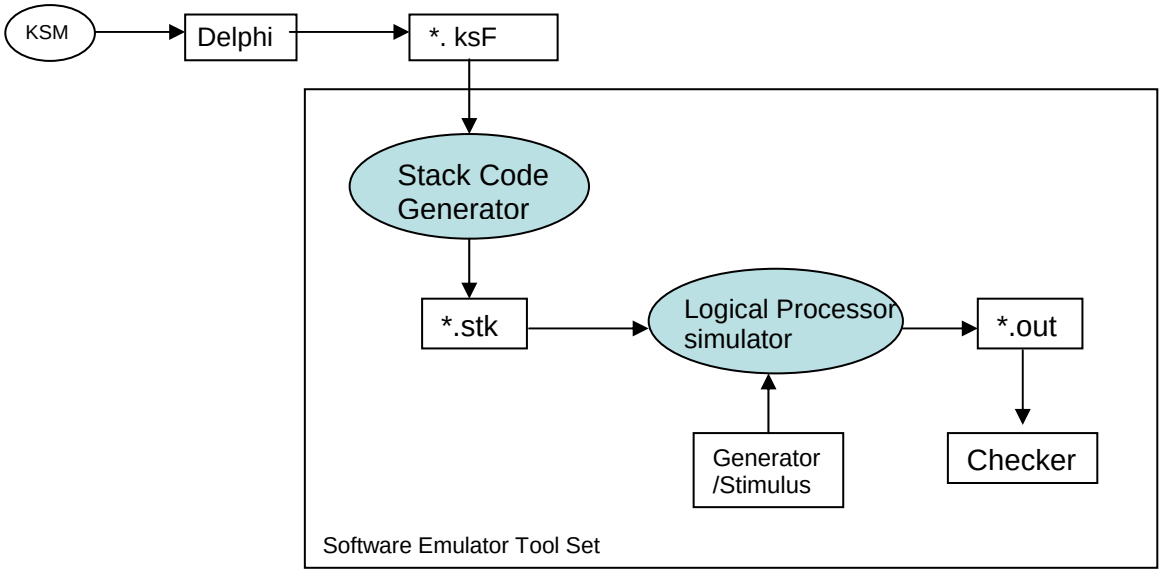


Figure 4.1: Verification tool flow for Software Emulation

An overview of the Delphi tool and Software emulator tool would look similar, both the tools are converting the KSF file to an object file which they are familiar with and execute them. The Delphi is familiar with Pascal code, hence it converts the KSM code to Pascal and the Software Emulator proposed coverts into C code. Then what is the need of two different versions? The difference lies in how the instructions are scheduled and executed.

Consider for example the 'A-Switch' in a 4X4X4 grid configuration. A shell file connects these sixty four modules together. These instructions are converted

into ‘KSF’ file where the entire ‘A-Switch’ module is a single instruction. Now when the Delphi tool executes this instruction it converts each of the ‘A-Switch’ instruction with its internal instruction. It is like a macro function which is later replaced by the function during compilation. Hence in the Delphi tool, there are essentially sixty four repetition of the code. When the entire chip has to be tested it will have eighty of the processors, sixteen SRAM and so on. This complex structure is not possible to verify using Delphi. In the software emulation tool, the ‘A-Switch’ module is more like a function. There is no repetition of the code leading to lesser memory occupation. With this capacity entire chip can be easily connected and verified.

The scheduling policy used in the software emulation tool is entirely different from that of Delphi too. In Delphi tool the sixty modules are connected and run some random number of times in a hope that the values will settle down to their final value. Remember the sixty four modules are interrelated and depend on each other. The execution has to continue till the intermediate values settle and the correct final value is reached. This approach of executing it for a random amount of time is very naive. The chance of the values settling and not settling down to their final value is equal. More over a lot of CPU time is wasted in doing some unwanted instruction. There is always likelihood that the value obtained is not right when the instructions are not executed for the right amount of time. For example suppose the instruction settles to its final value after n cycle, but the simulator has chosen the random time to be n-1 cycles, the output is definitely going to be wrong. This is not acceptable. The stack code generator offers a very clever solution of this.

The stack code generator analyzes the data dependencies between the modules with help of the combinatorial connection among the sub modules and within

the sub modules and generates a schedule such that minimum amount of CPU cycle is taken to get the correct value. The stack code generator is also capable of generating a schedule which occupies minimum memory. This is useful for the hardware emulation in the FPGA chip, since the memory is restricted in the FPGA. This memory allocation scheme need not be used in the software emulation, since the available memory is virtually unlimited. To derive the scheduling with the improved memory allocation by itself is time consuming and can limit the size of module that can be tested; hence it is better not to use the memory allocation scheme for the modules like ‘A-Switch’. In software emulation for modules like ‘A-Switch’ each variable allocated a unique memory location. This concept cannot be used for hardware emulation using FPGAs.

With the help of the stack code generator and the logical simulator it is possible to emulate very complex modules.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion:

With the increasing number of gates and design complexity, verification process has to be planned out very carefully and started at an early stage of designing. A single tool is no more enough to test these complex designs. A unified verification methodology is required to maximize the speed and efficiency. The two-level verification methodology proposed proves to be efficient for the verification of the multi-core architecture. The functional verification makes sure that the system works for intended design and the software emulation helps in verifying the integrated system. At an early stage of designing, functional verification can be carried out since the modules are small and it is very easy to debug with the help of waveforms. The software emulation makes it possible to verify the entire chip without the need of complex verification structure.

5.2 Future Work:

The two-level verification should be applied to the entire chip, to verify the full functionality of the C64 chip. The software emulation also generates object code for the FPGA, hence the hardware simulation can be tried. Only preliminary verification has been applied of the A-Switch, a more rigorous testing has to be done to measure the performance of the A-Switch.

Appendix A

EXAMPLE OF A KSM FILE

The KSM language is similar to any hardware description language. An entity in VHDL is called a 'Block' in KSM, which defines the input and output port. One of the advantages in KSM is that the intermediate signals do not have to be defined before using them. An example of the KSM file is given below. The syntax has been explained by comparing it with VHDL.

```
Block c64_Sw
  Comments
    You can add any comments over here

  Probes
  // Here the intermediate signals that have to
  viewed, have to be inserted.

  // For example in this consider the 'FrCh0'-input
signal to the SWA module. To view this signal, the
following line has to included in the probes.

  SwA.FrCh<0:5>_<D Tag>


  Inputs
    // The inputs of the block are defined here.

    FrMP<0:5>_<D[94:0] Tok[1:0]>

    FrCh<0:5>_D[83:0]
    FrPP_<C[3:0] L[11:0] U[7:0] A[15:0] D[63:0]
      Wr>
    DRPort<0:3>[6:0]
    ReSet
    Clk
  Outputs
    ToMP<0:5>_<D[101:0] Tok[1:0]>
    ToCh<0:5>_D[83:0]

  Conns
```

```
            // This is equivalent to architecture of VHDL

      MS#_S = Pad[5](0x_1F)  #:0:5

         //This syntax implies MS0_S = "111111"
         // i.e pad with hexadecimal 1F.
         // Repeat this for MS1_S to MS5_S

   // Component instantiation: Component name is c64_SWA

      . = c64_SwA(
            SwA FrCh<0:5> <D Tag>
             cs<0:5>.ToSwA <D Have>
             FrH<0:5>_<D[63:0] Have>
            cs<0:5>.ToSwA_C<0:1>_Get
            FrH<0:5>_C<0:1>_Get
            SwA FrP D
            FrP_D[9:0]
            SwA FrP A
            FrP_A[7:0]
            SwA FrP Wr
            FrP_Wr
            ReSet
            Clk
          )  SwA

        // Only the inputs signals have to be mapped to
the component. Outputs signals are automatically mapped.
Consider an output signal 'ToCh0_D from c64_SWA'. The output is
mapped to


   EndBlock
```

Appendix B

EXAMPLE OF A SHELL FILE

The shell file is the test-bench file is KSM. It can be considered as the top most module of any deign.

```
Block Shell_c64_Sw
  Comments
    Test shell featuring just Sw
  EndComments

  VFSROOT
  FixedWidth

  Inputs
    FrMP<0:5>_<D[94:0] Tok[1:0]>
    FrCh<0:5>_D[83:0]
    FrPP_<C[3:0] L[11:0] U[7:0] A[15:0] D[63:0]
    Wr>
    DRPort<0:3>[6:0]
    ReSet
    Clk

  Outputs
    ToMP<0:5>_<D[101:0]
    Tok[1:0]>
    ToCh<0:5>_D[83:0]

  Conns

    ToMP<0:5>_<D Tok>
    ToCh<0:5>_D
    = c64_Sw(
        FrMP<0:5>_<D Tok>
        FrCh<0:5>_D
        FrPP_<C L U A D Wr>
        DRPort<0:3>
        ReSet
        Clk
      )  sw
EndBlock
```

Appendix C

EXAMPLE OF KSF FILE

An Example of a KSF file is given below. The first statement gives the design unit name. The statements following that have an instruction number and name along with type of gate. At the end the inputs and outputs are listed.

```
Block C64_SIGB8 FirstReg 3 FirstComb 1
     1  ZTOCH_DD  OR  Inputs  I:1:0/0/W8 I:2:1/8/W8 I:3:2/16/W8
I:4:3/24/W8 I:5:4/32/W8 I:6:5/40/W8 I:7:6/48/W8 I:8:7/56/W8  Outputs
11/74/W8  Width 8  Rank 1  Next 4 RBPI
     2  ZTOCH_D.ZDD  GATEOFF  Inputs  C:1:1:11/74/W8 I:10:9/72/W1
Outputs  12/82/W8  Width 8  Rank 2  Next 0 RBPI
     3  ZTOCH_D.ZQ  RRREG  Inputs  C:2:1:12/82/W8 I:11:10/73/W1
Outputs  13/90/W8 VFSLoc 1 PO 2  Width 8  Rank 0  Next 5 RBPI RPO
     4  ZTOTU_D.ZDD  GATEOFF  Inputs  I:9:8/64/W8 I:10:9/72/W1
Outputs  14/98/W8  Width 8  Rank 1  Next 2 RBPI
     5  ZTOTU_D.ZQ  RRREG  Inputs  C:4:1:14/98/W8 I:11:10/73/W1
Outputs  15/106/W8 VFSLoc 2 PO 1  Width 8  Rank 0  Next 1 RBPI RPO

Inputs
     1 FRTU0_D[7:0]
     2 FRTU1_D[7:0]
     3 FRTU2_D[7:0]
     4 FRTU3_D[7:0]
     5 FRTU4_D[7:0]
     6 FRTU5_D[7:0]
     7 FRTU6_D[7:0]
     8 FRTU7_D[7:0]
     9 FRCH_D[7:0]
    10 RESET
    11 CLK

Outputs
     1 TOTU_D[7:0]    C:5:1
     2 TOCH_D[7:0]    C:3:1
```

REFERENCES

[1] **Brian Bailey, Mentor Graphics,** *Waking of the Sleeping Giant – Verification* DesignCon April 2002

[2] **Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao,** "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture"

[3] Multi-core architecture http://www.absoluteastronomy.com/encyclopedia/m/mu/multicore.htm

[4] Intel Article on Multi core architecture http://www.intel.com/cd/ids/developer/asmo-na/eng/211198.htm

[5] **Anick Bergeron,** *Writing Testbenches Functional Verification of HDL Models,* Qualis Design Corporation, Kluwer Academic Publishers 2003

[6] **Andreas S Meyer,** *Principles pf Functional Verification*

[7] Emulation : http://kb.iu.edu/data/aeve.html

[8] **Will Walker** *"Verification reuse ensures predictable design",* ISD Jan 2002

[9] **Brian Bailey**, Mentor Graphics, *"Co-Verification: from tool to methodology",* DesignCon Jan 2002

[10] **David Dempster and Michael Stuart** *"Verification Methodology Manual"* Teamwork International 2001

[11] **Rashinkar et al,** *"System-on-a-chip Verification"* Kluwer Academic Publishers 2001

[12] **Ashenden et al,** *"System-on-Chip Methodologies & Design Languages"* Kluwer Academic Publishers 2001

[13] **Cadence, White paper**, 'The unified Verfication Methodology'

[14] **Lavi Lev, Rahul Razdan, Chirstopher Tice**, *'It's About Time – Requirements for the functional verification of nanometer-scale ICS'*, Cadence

[15] **Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao,** *'TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture'*

[16] http://www.open-vera.com

[17] http://www.forteds.com/behavioralsynthesis/index.asp