POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master Degree in Electronic Engineering

Master Degree Thesis

Using HLS for design space exploration of LoadStore queue

Supervisor: Prof. Lavagno Luciano

Candidate: YUNQING ZHAO

March 2019

Table of contents

List of figures

List of tables

Abstract

Systemc is a set of C++ classes and macros which provide an event-driven simulation interface. These facilities enable a designer to simulate concurrent processes. It is applied to system-level modeling,architectural exploration,performance modeling,software development,functional verification,and high-level synthesis.

High-level synthesis (HLS) tools almost universally generate statically scheduled datapaths. Static scheduling implies that circuits out of HLS tools have a hard time exploiting parallelism in code with potential memory dependencies, with control-dependent dependencies in inner loops, or where performance is limited by long latency control decisions. In this work, we show that high-level synthesis of dynamically scheduled circuits is perfectly feasible by describing the implementation of a prototype synthesizer which generates a particular form of latency-insensitive synchronous circuits. So our design is depending on dynamically scheduled.

The ability to achieve maximal parallelism decides the efficiency of spatial computing. And it needs memory interfaces can correctly handle memory accesses that arrive in random order while still satisfying ensuring appropriate ordering and data dependency for semantic correctness. But a traditional memory interface for out-of-order processors could not meet the requirement like: a fundamental piece of information for correct execution.So, we need to find a approach order to get the requirements.The main idea is to dynamically allocate groups of memory accesses which are depending on the dynamic behavior of the application.Using high-level systhesis tool(Vivado HLS) to aquire the access order within the group.

Introduction

This chapter is a introduction of my thesis. I used systemC language to design a Load-Store Queue, and then, using Vivado HLS do synthesis, to do space exploration. I also introducted a dynamical scheduled method, came up with Lana Josipovic and her workmate, which can use our LSQ for design. For Load-Store queue, I used an approach with out-of-order exeution with a group allocator. And used Modelsim to do the simulation with the testbench by professor. Finally, check the results whether they were same.

Chapture1 was a simple introduction of SystemC language; Chapture2 summarized the main tools that I have used.; Chapture3 was a brief introduction of dynamical scheduled; Chapture4 detailed the load-store-queue structure and the implementation process; Chapture5 was the HLS results; while the last part shown the simulation results.

Acknowledgment

I would first like to thank my thesis supervisor Prof.Lavagno Luciano for providing this thesis, and for his motivation, patience, and immense knowledge. He teached me a lot in the whole period of thesis. And his guidance helped me in all the time of research and writing of this thesis.

Besides I thank my labmates and PhD students, Liang Ma, Junnan Shan for their help, advices and encouragement. I will always remember the laughter in the lab.

Finally, I must thanks to my parents for providing me with encouragements throughout my years of study in Italy and the process of projects, of courses and writing thesis.

Chapter 1

SystemC Language

1.1 What is SystemC

SystemC is a set of C++ classes and macros which provide an envent-driven simulation interface.These enable us to simulate concurrent processes, and we can use C++ syntax for describing each process. SystemC processes can communicate in a simulated read-time enviroment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined.

SystemC has semantic similarities to VHDL and Verilog, but may be said to have a syntactical overhead compared to these when used as a hardware description language.[1] On the other hand, it offers a greater range of expression, similar to object-oriented design partitioning and template classes.

SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level-synthesis.[1] And systemC is often associated with transaction-level modeling(TLM), and electronic system-level(ESL) design.

1.2 SystemC features

[1] SystemC includes some features, like modules, signals, processes, interfaces, ports and data types.

1. Modules are the basic blocks. A SystemC model usually includes of others modules which communicate by ports.

2. Ports allow communication from inside a module to the outside via channels.

3. Signals divided in two aspects, resolved signals and unresolved signals.

4. Processes are used to describe functionality. Processes are contained inside modules, and they are concurrent.

5. Interfaces are used to communicate between ports and channels.

6. SystemC introduces several data types which support the modeling of hardware. Like extended standard types, logic types, and fixed point types.

1.3 The advantage of SystemC

There are several advantages of using SystemC[2].

1. It inherits all the features of C++, a stable programming language accepted all over the world. It has got large language constructs, and makes easier to write the program with less efforts.

2. Rich in data types: along with the types supported by C++, systemC also supports the data types which are used by the hardware engineers.

3. It introduces the notion of time to C++, to simulate synchronous hardware designs. It is common in most of the HDL's.

4. All the 'processes' can excuted concurrently to simulate the concurrent behavior, irrespective of the order in which they are called.

5. It supports RTL design like most of the HDL's, while it also supports the design at an higher abstraction level.

Following picture shows the various design stages and the supported stages by systemC and HDL's:

21.PNG

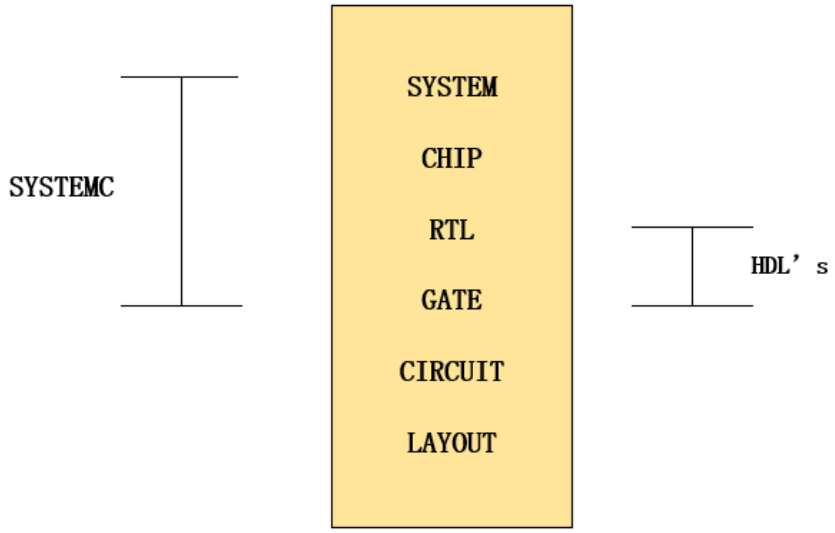


Figure 1.1: Various design stages and the supported stages

Chapter 2

Description of the tools

In this thesis, I used Vivado HLS to generate the RTL from my SystemC code and then used Modelsim to do the simulation with the testbench by professor.

2.1 High Level systhesis

High-level synthesis(HLS) is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. It raises the design abstraction level and allow rapid generaion of optimized RTL hardware for performance, area, and power requirements[3].

The goal of HLS is to let hardware designers verify hardware and efficiently build, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of abstraction while the tool does the RTL implementation[3]. Verification of the RTL is an important part of the process.

2.2 Vivado HLS

Vivado HLS is designed by Xilinx, which included as a no cost upgrade in all Vivado HLx Editions, accelerates IP creation by enabling C, C++ and SystemC soecifications to be directly targeted into Xilinx programmable devices without the need to manually creat RTL.

Supporting both the ISE and Vivado design environments Vivado HLS provides system and design architects alike with a fater path to IP creation.[3]

Vivado HLS supports older architectures specific to ISE Design Suite and installs automatically as part of the Vivado HLx Editions.

Chapter 3

Dynamically scheduled high level synthesis

This chapter describe a methodology to automatically generate dynamically scheduled circuits. The chapter is organized as follows: Section 1 is a example of one of the situations where dynamic extraction of operation-level parallelism proves essential to performance. Section 2 details the circuit generation methodology as implemented. Section 3 I compared the results of the technique with static HLS.

3.1 The reason of dynamic scheduling

The traditional statically scheduled HLS serves well that applications are regular. But, with FPGAs facing broader classes of applications and moving to datacenter, the capability of dynamic scheduling to automatically extract parallelism may be essential.

There is an example to prove the limitations of static scheduled HLS approaches, as shown in figure 3.1[4]. In this loop, there is a conditional statement(if) which depends on arrays A[] and arrays B[]. And the statement s = s + d replaces a dependency between iterations and delays the next iteration when the condition is true. A typical HLS tool needs to create a static schedule when we pipelining this loop. On the top of the figure 3.1(b), it is a possible schedule that the condition is true only for the second and third iteration. On the middle, it is an alternative schedule that could be to avoid pipelining the loop and creating a sequential finite-state machine. While the bottom one in the figure 3.1(b) is a better schedule, the operations of different iterations are overlapped as much as possible and the parallelism is reduced only

when the addition is executed. This behavior is beyond what can be achieved by statically scheduled HLS tool.

1.PNG

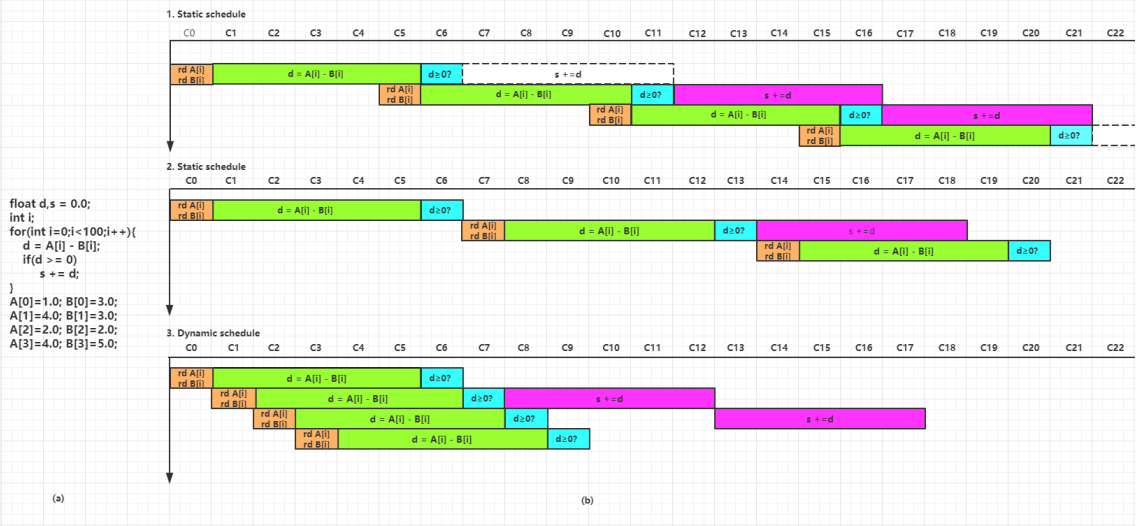


Figure 3.1: Limitations of static scheduling

From this simple example, there is a negative influence on performance when generating a schedule at synthesis time. Another problem is the conflict between read and write(a write in a previous iteration may address the same memory location as the read in a successive).However, they can be executed out of order if the two accesses match different locations. Of course, in recent years, many authors proposed approaches to some cases of potential dependencies through memory, but I choose dynamically scheduled circuits in my paper because I think it represent the most general solution.

3.1.1 Circuit of the example

Figure 3.2 shows a simplified version of an elastic circuit[5] implementing the example in figure 3.1(a).

22.PNG

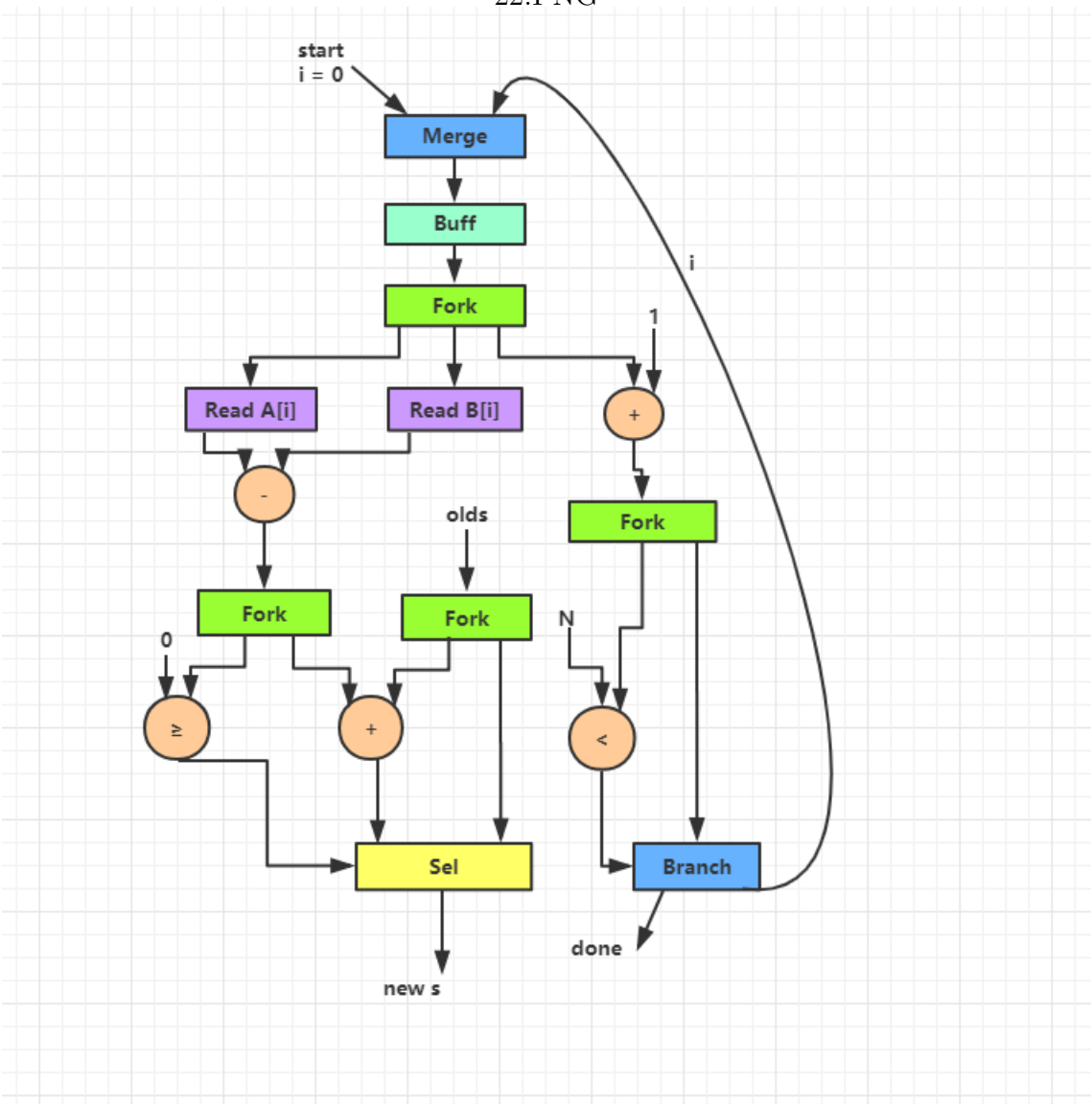


Figure 3.2: Elastic circuit

3.1.2 Comparison between static and dynamical scheduling

Compare with the static scheduling, dynamical scheduling out-of-order processors are capable of achieving good levels of paralellism on-the-fly and without extensive code preparation.

3.2 The circuit that I choose

In this section, I have chosen the elastic circuit[1] as the paradigm.

3.2.1 Circuit architecture

The architecture for the circuit that I used was shown in figure 3.3[4].

2.PNG

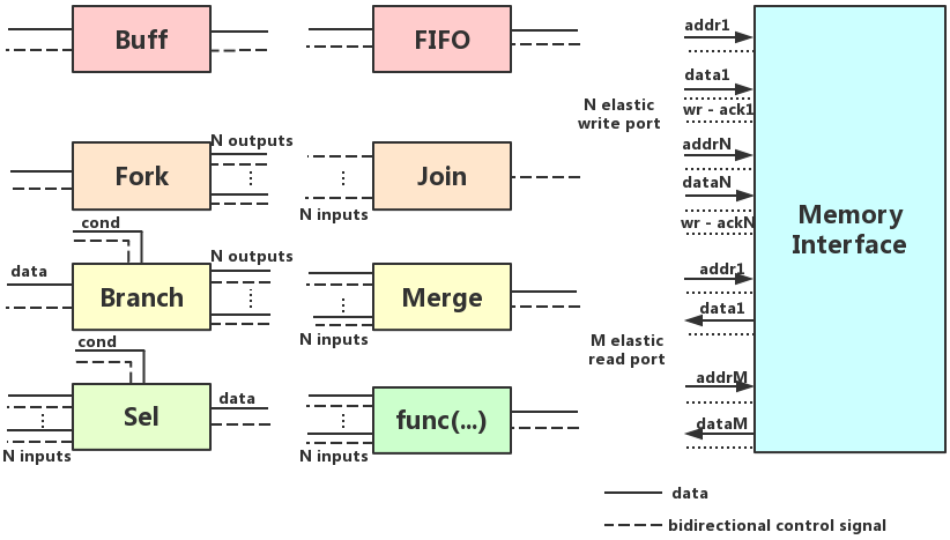


Figure 3.3: Circuit architecture

It outlines the elastic components all used.

(1) Buff is the elementary storage structure of the circuit and the equivalent of D flip-flops or registers in regular circuit.

(2) A fork put every token from input to multiple output; it outputs tokens to each successor, but only all successors have accepted the previous one, then it will update the new value.

(3) A branch executes control-flow statements by dispatching a token received at its single input to one of its outputs depending on a condition.

(4) A select used as a multiplexer, it waits for the required input to generate the output.

(5) FIFO is ordinary first-in first-out queues with the appropriate handshaking signals.

(6) A join is the reciprocal of a fork. Acts as a synchronizer which is waiting to receive a token on each and every one of its inputs before emitting a token at its outputs.

(7) A merge is the reciprocal of a branch. It propagates a token and data received on any input to its output.

(8) The func replaces any functional unit the code requires, such as integer and float point units.

Finally, the circuit interface with memory by elastic ports. From the picture we can see, the write port has two inputs(address and data) and a control-only signal from the memory interface demonstrate successful completion.

3.2.2 Connecting to load-store-queue

Figure 3.3 shows a memory component with elastic read and write ports. In general, every load or store operation would connect to a read or write port, but may be get a wrong result. So, access requests will arrive to the memory interface in random order and this may lead to against the dependency.

The solution to solve the problem is to use a load-store-queue which I will introduce in the next chapter similar to those present in dynamically scheduled processors. To get tokens following the actual order of execution of the basic block of the circuit is the important condition for the load-store-queue with correctly implementation. This ordering can make the load-store-queue as memory access arguments from basic block arrive out-of-order.

Figure 3.4 shows the connecting to a load-store-queue for correct out-of-order memory accesses. This program contains four basic blocks. Comparing with the figure 3.3, the difference is only increasing the elastic control signal like BB1 and BB2. BB1 sends a token to load-store-queue at the start of the program. We assuming that the execution of BB2 afterwards was determined by the control flow, then BB2 will sends a token to the load-store-queue next. Accesses of BB1 should be completed before those from BB2, so the order of these token can make the load-store-queue to suitably out-of-order memory accesses.

6.PNG

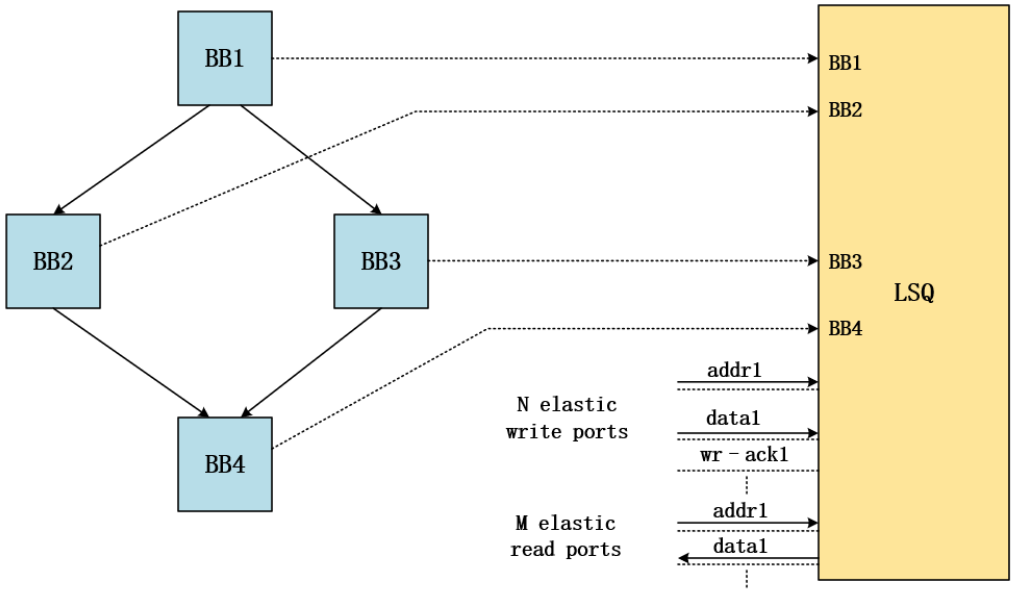


Figure 3.4: Connecting to load-store-queue

We found some examples to show the correct connections and incorrect connection, the details are in the picture 3.5[4].

7.PNG

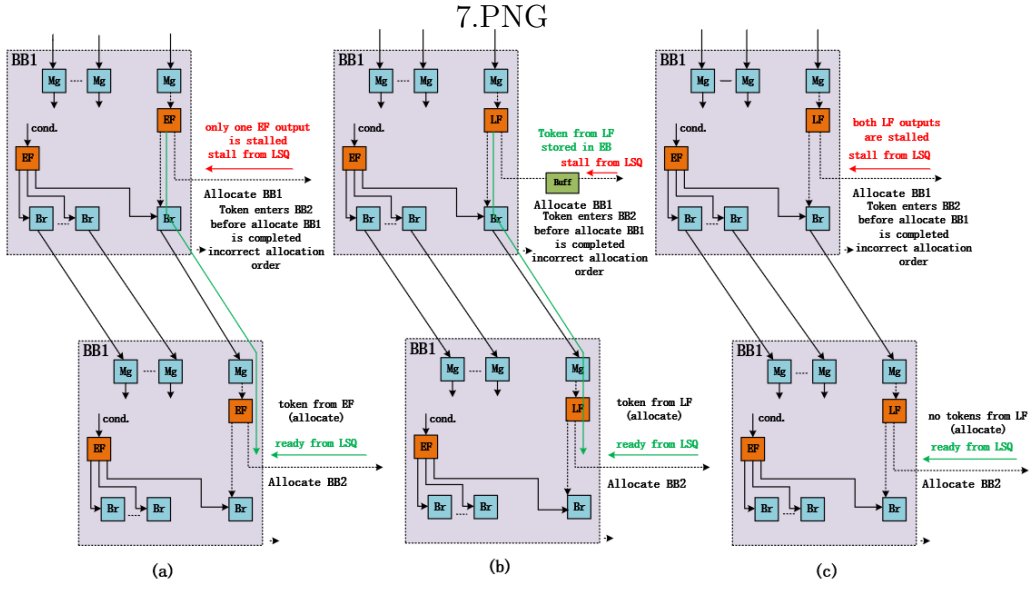


Figure 3.5: Connecting the circuit to the memory interface

Figure 3.5(a) and (b) were two examples of the incorrect connections. In the first example((a)) ,the eager fork may send an allocation to BB2 before the allocation of BB1 completes. The problem of the second example is that the allocation order may be reversed due to the storage element on the control line between the circuit and the load-store-queue. The last example in figure 3.5(c) shows the correct way to connect the load-store-queue: an allocation cannot occur unless all predecessor allocations have been completed.

3.3 Comparison the results

This section, we show the result of comparing with a commercial HLS tools and with other approaches that are closer to my chosen.

3.3.1 Comparison with Static HLS

The results of the comparison with static HLS of different kernals were shown in Figure 3.6[4].

31.PNG

	II_{avg}		CP(ns)		Execution time(us)		DSPs		FFs		Slices		LUTs	
Kernal	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN
Histogram	11	2.3	3.3	5.7	36.3	13.3	2	2	447	1734	130	1101	296	3632
Matrix power	16	4.2	3.4	6.0	20.7	9.6	5	5	790	2050	219	1465	500	4237
FIR	1	1	3.3	3.4	3.3	4.4	3	3	224	382	62	127	89	341
Loop with condition 1	9	1.3	2.8	4.8	25.3	6.2	2	4	525	984	161	289	391	767
Loop with condition 2	5	1.2	3.4	4.8	17.1	5.7	5	5	623	811	187	240	409	659

Figure 3.6: Compare the results between dynamically scheduled and statically scheduled

From the picture, we can see there are five different kernals to implement the program. And in the figure here has shown the timing cost and also the resource utilization. Kernals detailed as following:

(1) Histogram reads an array of features and increases the of value of corresponding histogram bins.

(2) Matrix power performs a series of matrix vector multiplication. Each iteration of a nested loop reads a column and a row coordinate and updates the corresponding matrix element.

(3) FIR filter is an ordinary FIR filter calculating the output based on the input.

(4) Loop with condition 1 is the kernal I have mentioned before.

(5) Loop with condition 2 is a variation of the previous kernal where replacing the conditional addition with a multiplication of the same variable.

3.3.2 Comparison with other closer approaches

Here we compared the approach that I chosen with other two similar approaches, which were proposed by Huang and Budiu respectively. The results were shown in figure 3.7[4].

32.PNG

	II_{avg}				CP(ns)				Execution time(us)				Slices			
Kernal	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN
Histogram	12	11	3.7	2.3	4.9	4.8	5.9	5.7	58.9	52.9	21.2	13.3	134	149	182+901	200+901
Matrix power	17	16	5.0	4.2	4.1	3.9	6.3	6.0	26.6	23.8	11.9	9.6	204	233	332+1113	352+1113

Sheet1 Sheet2

Figure 3.7: Compare the results between different dynamically approaches

Huang et al. generated elastic circuit from C code, to be mapped to a coarse-grain reconfigurable array[6].

Budiu et al. described a compiler for generating asynchronous circuit from C code[7, 8].

The designs of Huang et al. cannot achieve any pipelining, so the performances are not good even in static scheduled. While for the Budiu et al.'s design, it increasing the pipeline throughput compare with the design of Huang et al. But the performances are lower than that I have chosen.

Chapter 4

Load-store-queue Design

This chapter we described the details of our load-store-queue, by writing in SystemC code, then did the synthesis by Vivado HLS tools. Using modelsim to simulate the RTL that generated automatically by Vivado HLS ordering to verify the results. Section1 is an introduction of the work. Section2 indicated the allocating approach for out-of-order load-store-queue. Section3 shows the structure of load-store-queue and each function of every module.

4.1 Introduction

With the development of the society, we used more and more FPGAs in our designs. While modern computer processors are very complex. Their core task is to execute a series of instructions(we also called them programs) and save the results in memory. So, in order to execute instructions as much as possible, or to improve performance, the processor adds multiple execution cores, uses faster caching, and uses different techniques(like out-of-order execution, branch prediction, speculative execution, data prefetching, memory access reordering and memory disambiguation...). Here, we chosen out-of-order execution for our design.

Because a superscalar processor executes the instructions in an out-of-order fashion, the identification of data hazards created by the dependencies between the load and store instruction becomes a necessity. And, high-level synthesis tools could not satisfied the need because of the static scheduling itself. In many applications, they are impossible to disambiguate across the static scheduling, like data analytic, graph processing and others that not mentioned here. Besides,

high-throughput pipelined design can be generated with independent memory accesses by using static HLS tools, and also the performance is lower. So, a memory disambiguation technique is used.

The out-of-order execution have some advantages like: for instructions, an early execution was permitted, so do not have a dependency with younger instructions; the data that would be written to the I-cache memory by that store instruction can be forwarded to the dependent load instruction, thus, completing it beforehand.

Let us use a simple example to describe the situation. Figure 4.1[9] shows the a simple loop as an example.

11.PNG

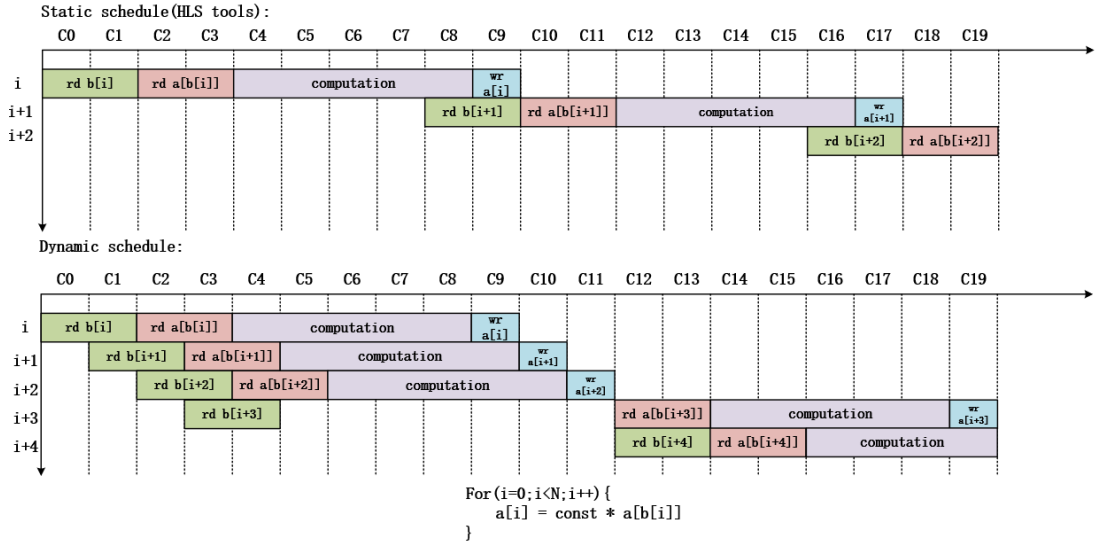


Figure 4.1: A loop example

On the top of the picture, it is a static schedule created by a high-level synthesis tool. We know that only write to a[i] finished, we can read a[b[i+1]] next, so it would generate a delay. While on the bottom of the picture where shows the dynamic schedule indicate that the accelerator could potentially start a new iteration every cycle.

In this work, we design an out-of-order load-store-queue by using SystemC code as an interface between memory and accelerator.

4.2 Load-store-queue architecture

This section introduce the traditional load-store-queue and the architecture of our load-store-queue.

4.2.1 Traditional load-store-queue

We use a typical load-store-queue[4] below to give a brief description. It contains two pointers and two entries, each entry consists of four parts, as the picture shows.

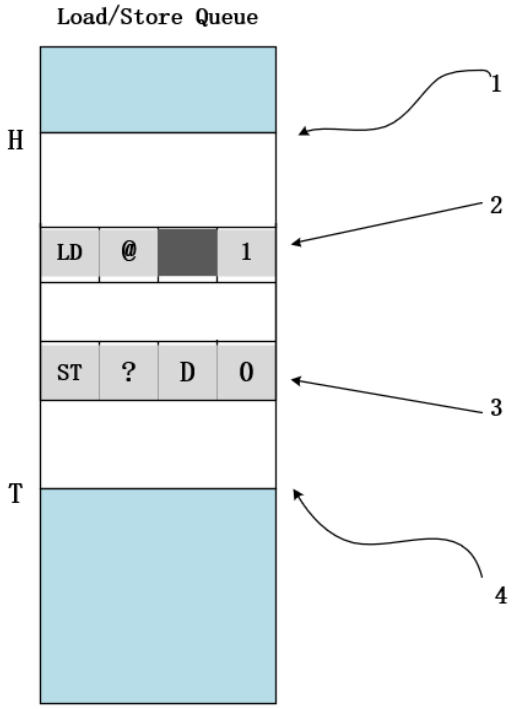


Figure 4.2: Typical load-store-queue

From the picture, 1 is the head of the queue, where entries are allocated when executed; 2 represent a typical load entry, with the target address "@" already available and set the executed flag. 3 us a typical store queue which the target address not yet available and the data to store "D" already available, and, not yet executed. 4 is the tail of the queue where entries are allocated. For each entry, the four parts are: (1) the operation type (load or store); (2) a memory address to access; (3) the data to be stored(only for store operation); (4) a flag to signal completion(put to 0 or 1).

4.2.2 A load-store-queue in an out-of-order processor

Here is an example of the basic operation of an load-store-queue in an out-of-order processor, as the figure 4.3 shown[9]. The execution process is divided into six parts.

13.png

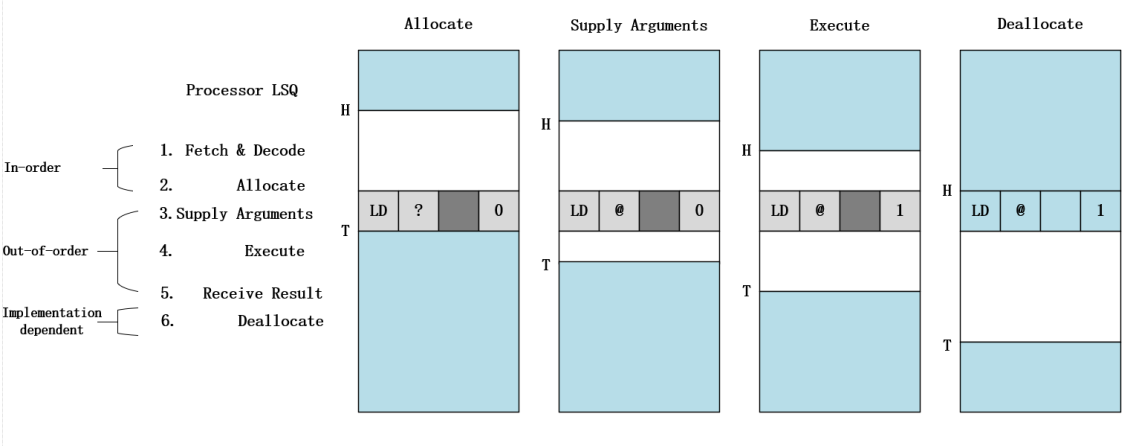


Figure 4.3: A load-store-queue with an out-of-order processor

- **Fetch and Decode. According to the requirements of various parameters of the configuration, the processor fetches the instruction from I-cache, adding it to the fetch queue. If we can not find an instruction from I-cache, and configuring an II-cache, we will find instruction from it, otherwise, we find the instruction from memory, and ,decode it. Of course , it is a load or store instruction and passes it to the load-store-queue**

- **Allocate. The load-store-queue allocates register update unit resource for the instruction which has been fetched from step1(means the instruction of the end of the load-store-queue) and connects the instruction with the new entry.**

- **Supply Arguments. Check that whether the instruction from step2 can be used. For store instruction, written the data value, and it finally determine the memory address. This information is provided to load-store-queue, which writes the actual address and data in the reserved entry.**

- **Execute. The load-store-queue executes the operation when satisfying the dependency and ready.**

- **Receive the result(write back). Write the result of the memory instruction from the load-store-queue back to processors.**

- **Deallocate. The load-store-queue deallocates the entry for the instruction.**

The first two steps need to be executed sequentially because this order need to respect the potential dependencies of the load-store-queue. The fifth step is to receive the result. Step 3 to 5 can occur out-of-order in the nowadays superscalar processors, but the load-store-queue ensures that dynamically generated memory dependencies are properly serialized.

In a spatial computing system, there is no fetch and decode. A typical spatial computing system is designed by transform a dataflow into a circuit. Like the figure 4.4 shows[9]. Solid line represent actual data dependencies while dotted line indicate the potential dependencies. Static scheduled maybe can transform or remove some of the edges(as shown in the figure 4.4(b)), but not all of them can be removed. So we need to find an approach to solve the problem.

14.png

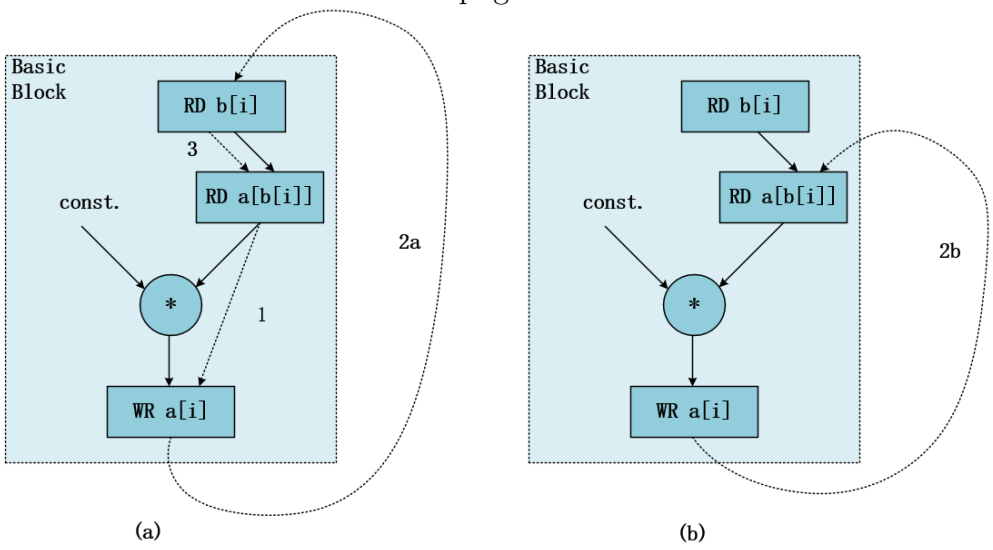


Figure 4.4: Typical spatial computing system

4.2.3 Comparison our method to others

For spatial exploration, we hope an out-of-order execution for a memory interface with satisfying the demand of dependencies. So we found an approach that allocating entries by group is most suitable. Comparing with other strategies which not detailed here, the advantages are two aspects. (1) Applicable to any program. (2) Fast out-of-order execution. Figure 4.5[7][10], Figure 4.6[11], and Figure 4.7[9] show the different allocating approaches and the last is what we used to design the load-store-queue.

16.png

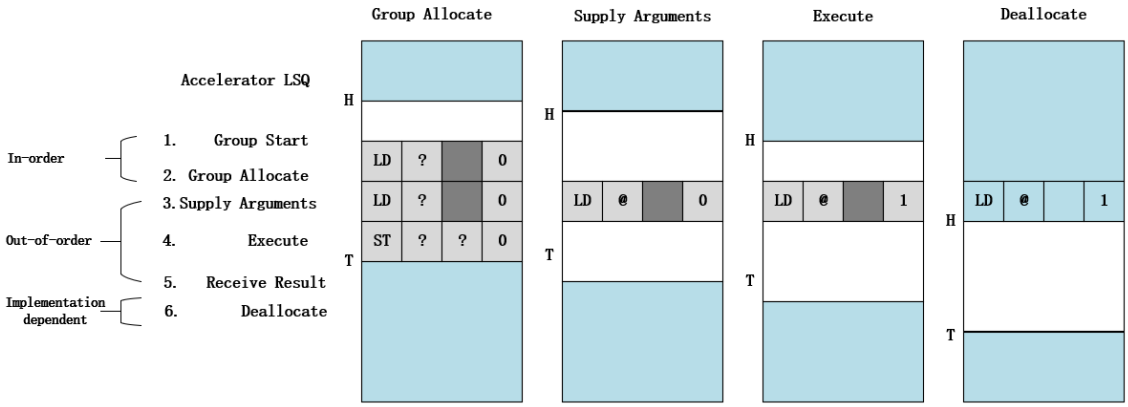


Figure 4.5: Allocating entries statically before execution

15.png

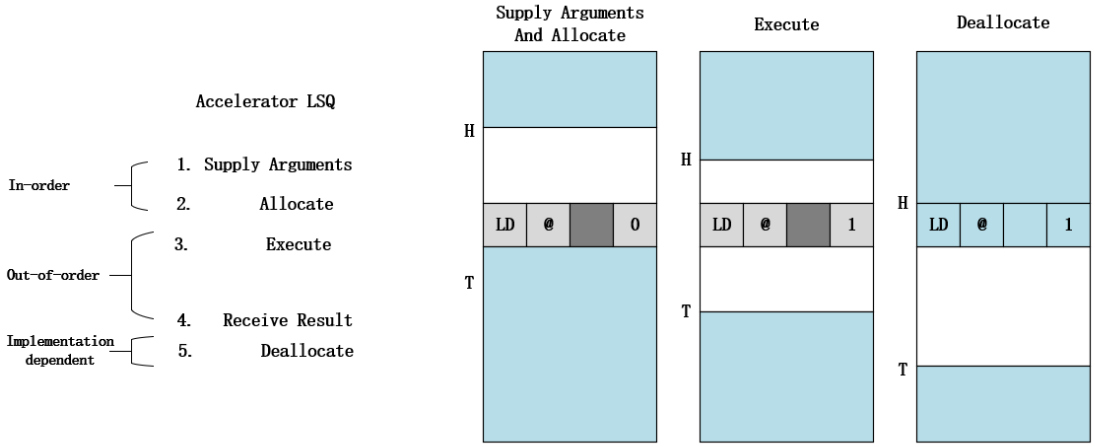


Figure 4.6: Allocating entries when the arguments are supplied to the load-store-queue

4.3 Load-store-queue structure

This section details our load-store-queue. Mainly divided into two aspects: Overview structure and Group allocator. When we use SystemC code to design the load-store-queue, we divided it into six module, they are store-address-port, store-data-port, load-port, store-queue, load-queue and bbstart, respectively.

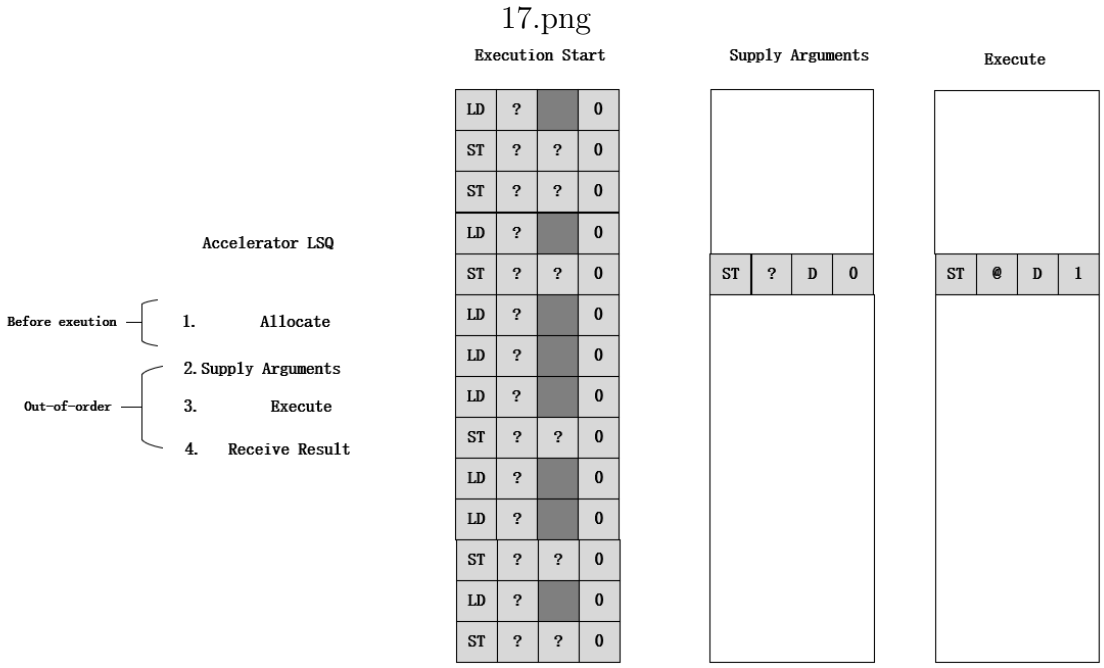


Figure 4.7: Allocating entries by group

4.3.1 Overview structure

Figure 4.8 is the overview structure of our load-store-queue. At the top of the picture, group allocator links entries that have been prepared in the queues to access ports. In the middle of the figure, are store queue and load queue; dispatchers connect the entries to the access ports; the issue logic is used to decides the safely entries to send to the memory and to check dependencies. Both load queue and store queue have a tail and a head register and contains entries.

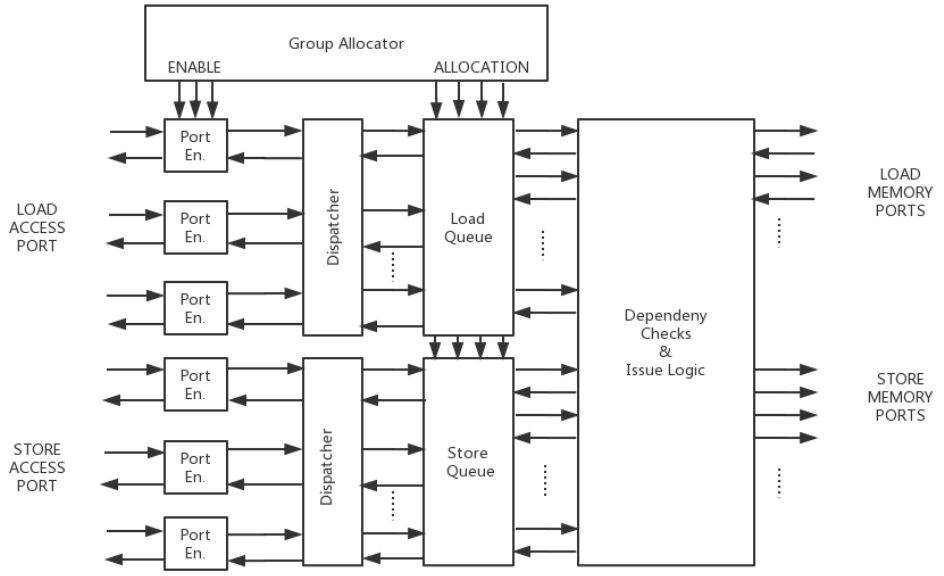


Figure 4.8: Load-store-queue structure

A counter is used for ports to determine the number of arguments it can accept, the counter is incremented at every time the corresponding group is allocated, and every time the port received an argument.

For bbstart module, mainly composed of eight functions.

- **Calculate load slots: to calculate empty slots in load queue.**
- **Calculate store slots: to calculate empty slots in store queue.**
- **Empty slot comparator1: to check if new allocation can be performed based on available empty slots in both queues.**
- **Load offsets: calculate tails for loadtailq.**
- **Store offsets: calculate tails for storetailq.**
- **bbsize: send basic block size to queue.**
- **bbports: send basic block ports to queue.**
- **bbready0: send elastic signal to circuit.**

Each entry of load queue consists from eight parts: (1) load address; (2) address valid flag,indicating the state of the entry; (3) data received

from memory; (4) data valid flag, representing whether the data has been received from memory;(5) preceding store, pointer to the last store entry in the store queue; (6) Port ID; (7) executed flag; (8) the flag of returned to port. The mainly implement functions as below shown:

- Tail update: update queue tail when allocating.
- Initialize values: initialize queue values or flags when allocating.
- Tail queue update: update tail queue.
- Port queue update: update port queue.
- Check bits: checks for dependencies between queue.
- Allocated bits: needed for empty flags.
- Empty: needed in bbstart to determine empty slots.
- Empty entries: determined empty entries.
- Pending register: load return from empty.
- Shifted load tails: shifted values based on tail queue entry.
- Entries to check: indicates which entries need to be checked for dependencies.
- Comparators: Find address conflicts.
- Decision: decide bypass or load request or nothing.
- Input priority port0: determine first (from head) position expecting input value from port0.
- Output priority port0: load single port output.
- Update address bits: flags for address queue.
- Update all done bits: load all done update.

Each entry of store queue consists from seven parts: (1) store address; (2) address valid flag,indicating the state of the entry; (3) store data; (4) data valid flag, representing whether the data has been received from memory;(5) preceding store, pointer to the last load entry in the load queue; (6) Port ID; (7) executed flag. The mainly

implementation functions are similar with load queue, so I will not detail it here.

4.3.2 Group allocator

The group allocator to our memory interface is different with others. When a statically-determined sequence of accesses begins, this unit allocates in parallel all the corresponding entries into each of the queues. In the picture 4.9[9], describe the group allocator.

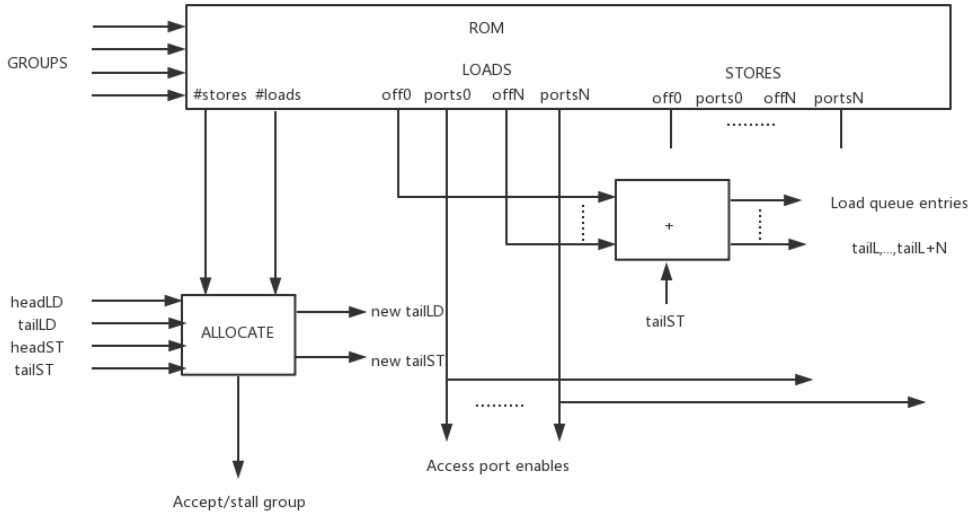


Figure 4.9: Group allocator

Only one new request can arrive at any moment in time because group

allocation requests are sequentially by definition. Each group allocation request a ROM to transform the data at the core of the group allocator. This ROM contains six parts: (1) number of stores in the group; (2) number of loads in the group; (3) Load offset, that means for each load in reference sequence, the number of stores preceding in the group; (4) The access port ID for load; (5) Store offset, similar to load offset, represent the number of loads preceding in the group for each store in reference sequence. (6) The access port ID for store.

Chapter 5

Synthesis using Vivado HLS tools

This chapter, we detailed the synthesis process by Vivado HLS of SystemC code, and transforming them into hardware structure. Generating the RTL for the next verification. We also discussed the synthesis rules in this chapter.

5.1 Constrains of the HLS tool

Several constrains should be considered when using Vivado HLS tool to do synthesis.
(1) Pointer limitation: Vivado HLS can not suppose pointer casting. It allows array pointer. Each pointer points to a scalar or an array of scalars, not to other pointer.
(2) Recursive functions: they could not be synthesized. They are functions that can form endless recursion, also tail recursion where there is a finite number of function calls
(3) System calls: these are actions related to some tasks of operating system. They are not synthesizable.
(4) Dynamic memory usage function: they can only used for checking the functionally of the top-level function, not for synthesis.
In order to do the synthesis for top-level function, all the constraints mentioned before must be taken into account.

5.2 Optimization solutions

We did the synthesis with different solutions of optimization, and compared the performance of them, then choose one RTL result for

verification.

5.2.1 Loop optimization

Loops are very well supported by synthesis. There are several methods to optimize the performance of the loop and the frequently adopted are pipelining, loop merging, loop inlining, loop flattening and loop unrolling. Here we used loop unrolling for optimization.

5.2.2 Array optimization

Array may result in some problems during synthesis. When we need a big array, it may run out of memory. Fortunately, Vivado HLS synthesizes the array interfaces to single port RAM by default, but it also can be a dual-port RAM if this can increase the performance of the top-level function. The frequently adopted directives for array are: array partition, array map and array reshape.

Here, we used array partition which is used to split array into multiple smaller arrays, each of them has its own interface. The synthesis results were shown in figure 5.1 and 5.2

33.png

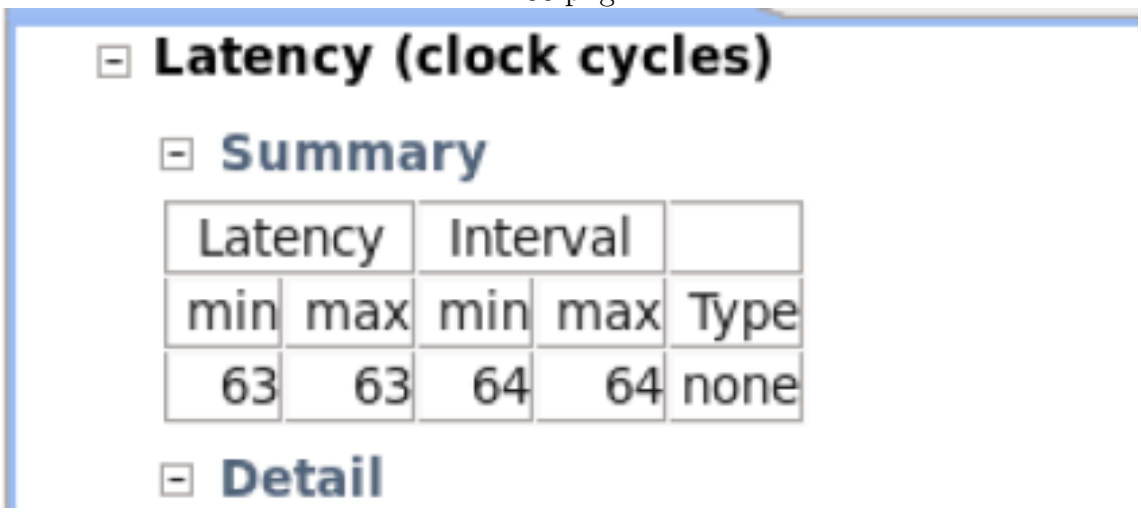
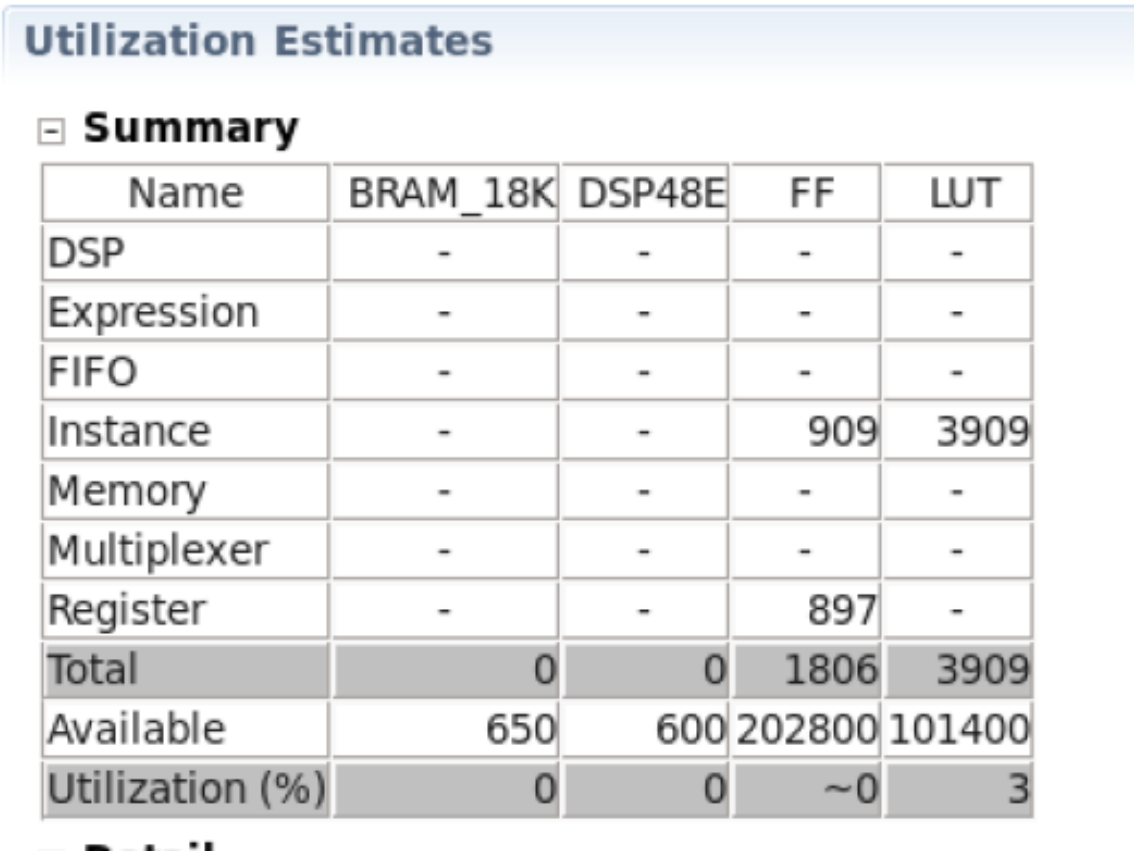


Figure 5.1: Latency for array partition

34.png



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	909	3909
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	897	-
Total	0	0	1806	3909
Available	650	600	202800	101400
Utilization (%)	0	0	~0	3

Figure 5.2: Utilization resource for array partition

Figure 5.3 and 5.4 is the interface after synthesising.

35.png

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
rst	in	1	ap_none	rst	pointer
clk	in	1	ap_ctrl_hs	load_queue::load_queue	return value
ap_rst	in	1	ap_ctrl_hs	load_queue::load_queue	return value
StoreEmpty	in	1	ap_none	StoreEmpty	pointer
BBStart	in	1	ap_none	BBStart	pointer
BBFirstLoad	in	1	ap_none	BBFirstLoad	pointer
Port0_Ready	in	1	ap_none	Port0_Ready	pointer
Port0_LoadWriteEn	in	1	ap_none	Port0_LoadWriteEn	pointer
StoreTail	in	4	ap_none	StoreTail	pointer
StoreHead	in	4	ap_none	StoreHead	pointer
BBLoadSize	in	4	ap_none	BBLoadSize	pointer
StoreAdrDone	in	4	ap_none	StoreAdrDone	pointer
StoreDataDone	in	4	ap_none	StoreDataDone	pointer
LoadChecks	in	4	ap_none	LoadChecks	pointer
MemoryLoadData	in	32	ap_none	MemoryLoadData	pointer
Port0_AddrIn	in	10	ap_none	Port0_AddrIn	pointer
StoreAddressQueue_0	in	10	ap_none	StoreAddressQueue_0	pointer
StoreAddressQueue_1	in	10	ap_none	StoreAddressQueue_1	pointer
StoreAddressQueue_2	in	10	ap_none	StoreAddressQueue_2	pointer
StoreAddressQueue_3	in	10	ap_none	StoreAddressQueue_3	pointer
StoreDataQueue_0	in	32	ap_none	StoreDataQueue_0	pointer
StoreDataQueue_1	in	32	ap_none	StoreDataQueue_1	pointer

Figure 5.3: Interface1 for array partition

36.png

StoreDataQueue_0	in	32	ap_none	StoreDataQueue_0	pointer
StoreDataQueue_1	in	32	ap_none	StoreDataQueue_1	pointer
StoreDataQueue_2	in	32	ap_none	StoreDataQueue_2	pointer
StoreDataQueue_3	in	32	ap_none	StoreDataQueue_3	pointer
BBLoadOffsets_0	in	2	ap_none	BBLoadOffsets_0	pointer
BBLoadOffsets_1	in	2	ap_none	BBLoadOffsets_1	pointer
BBLoadOffsets_2	in	2	ap_none	BBLoadOffsets_2	pointer
BBLoadOffsets_3	in	2	ap_none	BBLoadOffsets_3	pointer
BBLoadPorts_0	in	1	ap_none	BBLoadPorts_0	pointer
BBLoadPorts_1	in	1	ap_none	BBLoadPorts_1	pointer
BBLoadPorts_2	in	1	ap_none	BBLoadPorts_2	pointer
BBLoadPorts_3	in	1	ap_none	BBLoadPorts_3	pointer
LoadAddrQOut_0	out	10	ap_vld	LoadAddrQOut_0	pointer
LoadAddrQOut_1	out	10	ap_vld	LoadAddrQOut_1	pointer
LoadAddrQOut_2	out	10	ap_vld	LoadAddrQOut_2	pointer
LoadAddrQOut_3	out	10	ap_vld	LoadAddrQOut_3	pointer
LoadEmpty	out	1	ap_vld	LoadEmpty	pointer
MemoryLoadEnable	out	1	ap_vld	MemoryLoadEnable	pointer
Port0_Valid	out	1	ap_vld	Port0_Valid	pointer
LoadTailOut	out	4	ap_vld	LoadTailOut	pointer
LoadHeadOut	out	4	ap_vld	LoadHeadOut	pointer
LoadAdrDoneOut	out	4	ap_vld	LoadAdrDoneOut	pointer
LoadDataDoneOut	out	4	ap_vld	LoadDataDoneOut	pointer
Port0_DataOut	out	32	ap_vld	Port0_DataOut	pointer
MemoryLoadAddress	out	10	ap_vld	MemoryLoadAddress	pointer

Figure 5.4: Interface2 for array partition

Compared to the two solutions, we finally used the RTL generated by the second solution for verifying, which the result indicated in the next character.

We also did the synthesis with changing the number of depth and the number of ports. The utilization estimates results were seen in bellow.

From the pictures, we know that with the one variable depth changed, the utilization estimates with the significantly increased.

52.png

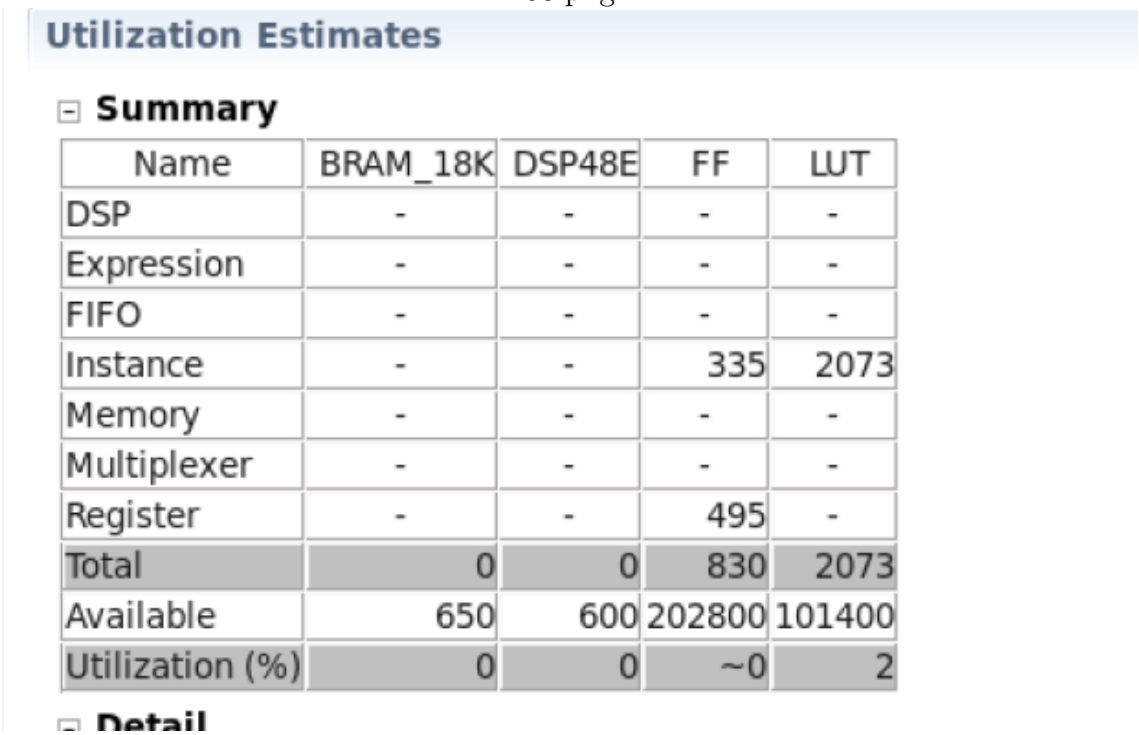
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	335	2073
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	489	-
Total	0	0	824	2073
Available	650	600	202800	101400
Utilization (%)	0	0	~0	2

Figure 5.5: Port-num: 1 or 2; Depth: 2

53.png



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	335	2073
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	495	-
Total	0	0	830	2073
Available	650	600	202800	101400
Utilization (%)	0	0	~0	2

Detail

Figure 5.6: Port-num: 4; Depth: 2

54.png

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	335	2075
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	499	-
Total	0	0	834	2075
Available	650	600	202800	101400
Utilization (%)	0	0	~0	2

Figure 5.7: Port-num: 6; Depth: 2

55.png

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	56	1257
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	288	-
Total	0	0	344	1257
Available	650	600	202800	101400
Utilization (%)	0	0	~0	1

Figure 5.8: Port-num: 1; Depth: 1

Chapter 6

Simulation results

In this chapter, we verified the RTL that generated by Vivado HLS and show the simulation results of waves for our load-store-queue by using Modelsim. Figure 6.1 is the store-address-port result.

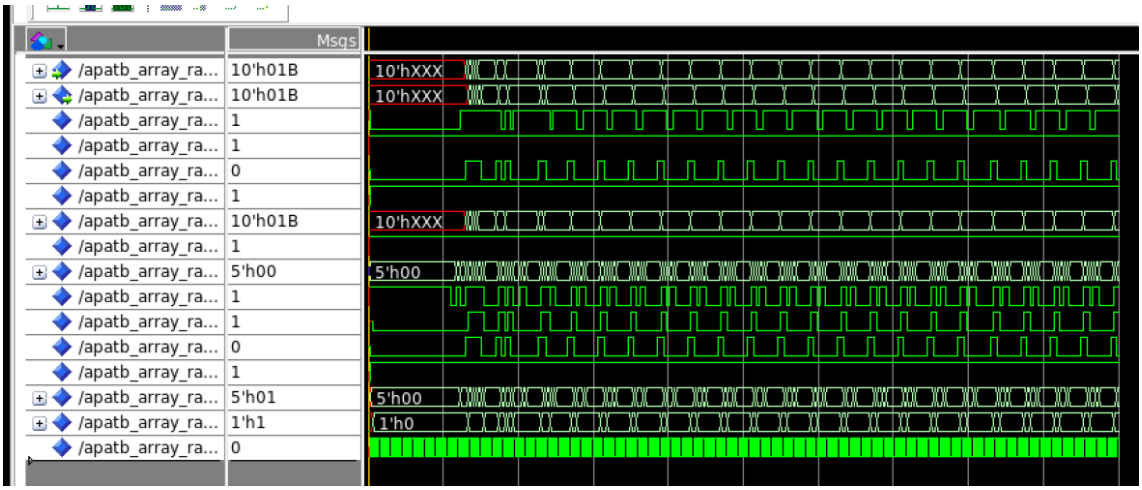


Figure 6.1: store-address-port

Figure 6.2 shows the store-data-port result.

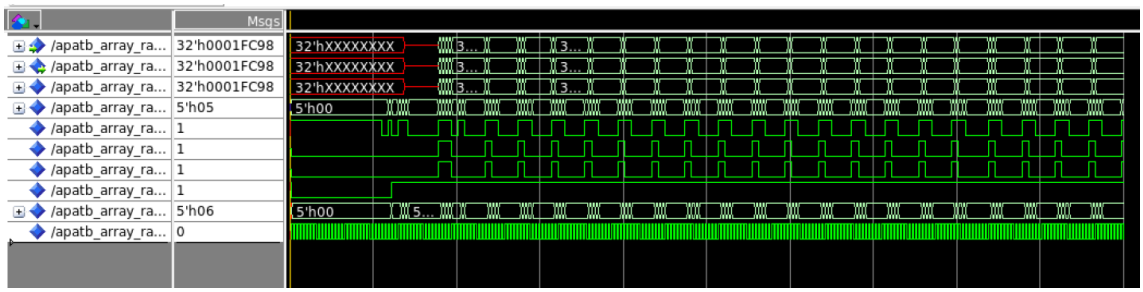


Figure 6.2: store-data-port

Figure 6.3 is the result of load-port.

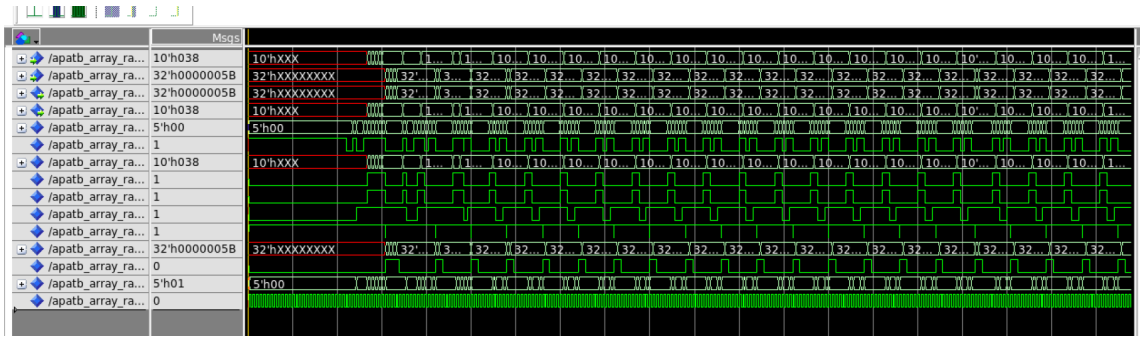


Figure 6.3: load-port

Figure 6.4 and 6.5 are the bbstart result.

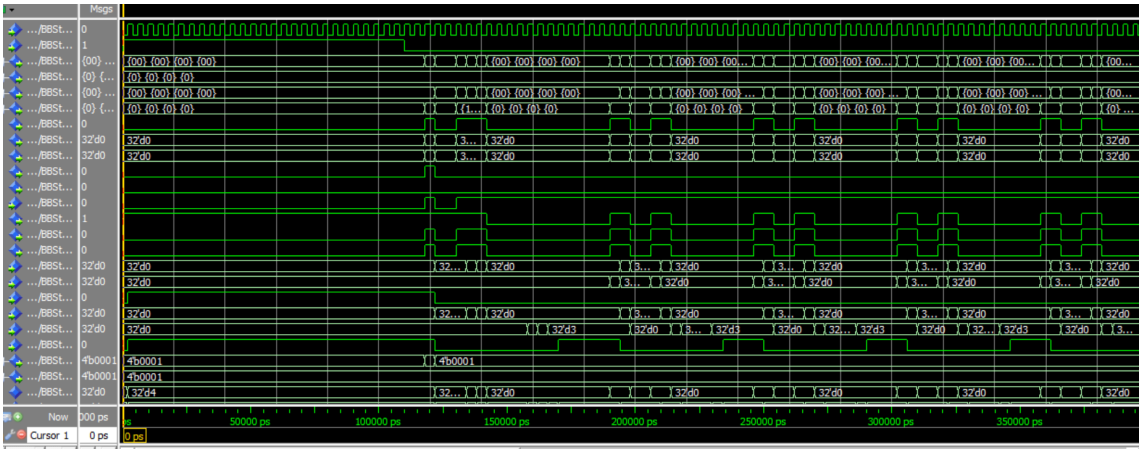


Figure 6.4: bbstart1

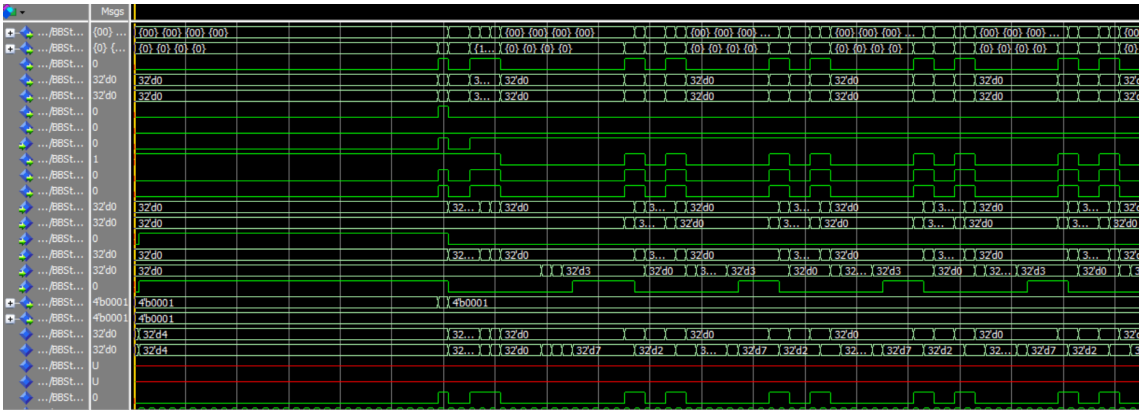


Figure 6.5: bbstart2

From figure 6.6 to 6.8, are shown the result of store-queue.

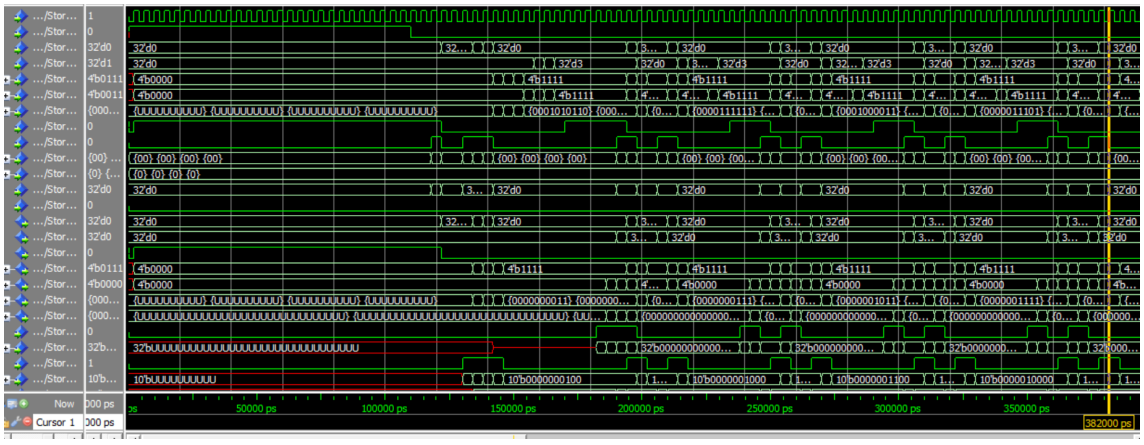


Figure 6.6: store-queue1

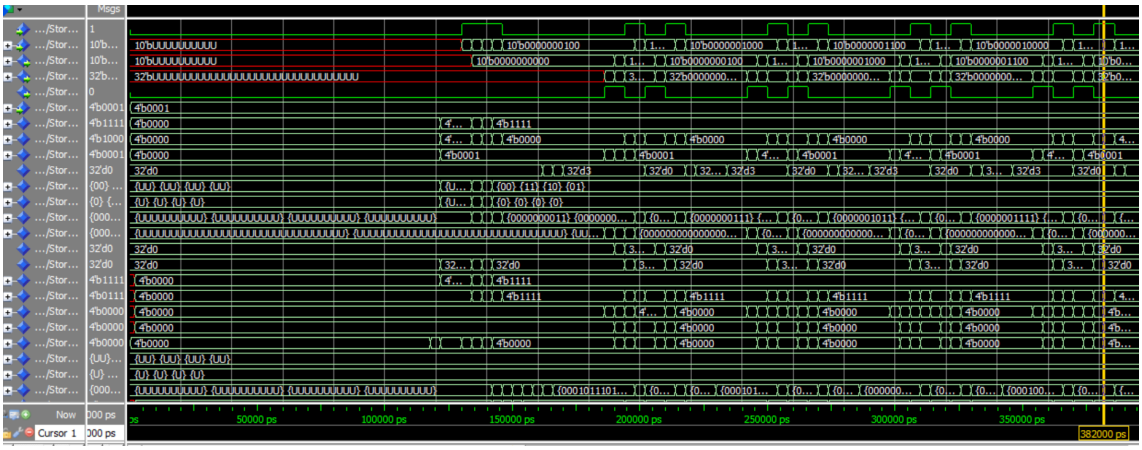


Figure 6.7: store-queue2

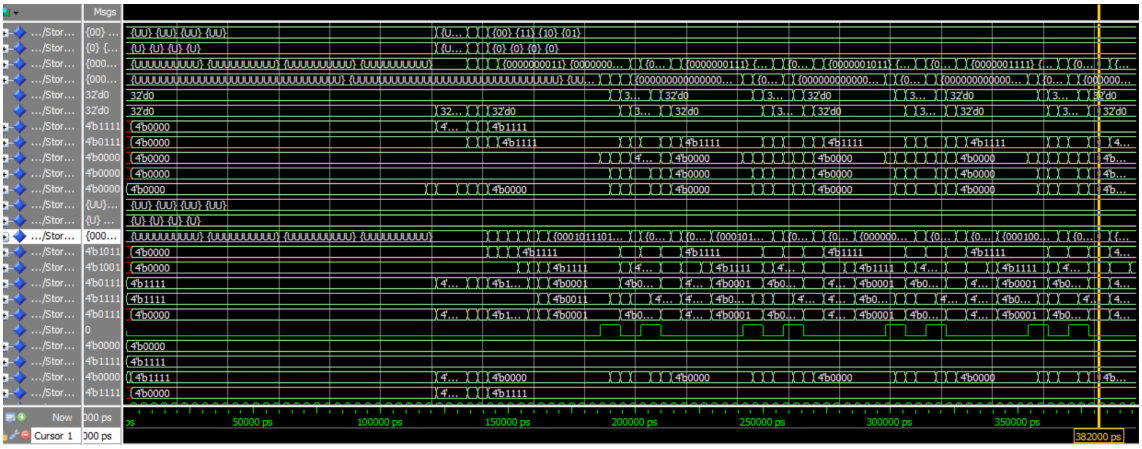


Figure 6.8: store-queue3

Obviously, remain figures represent for load-queue results.

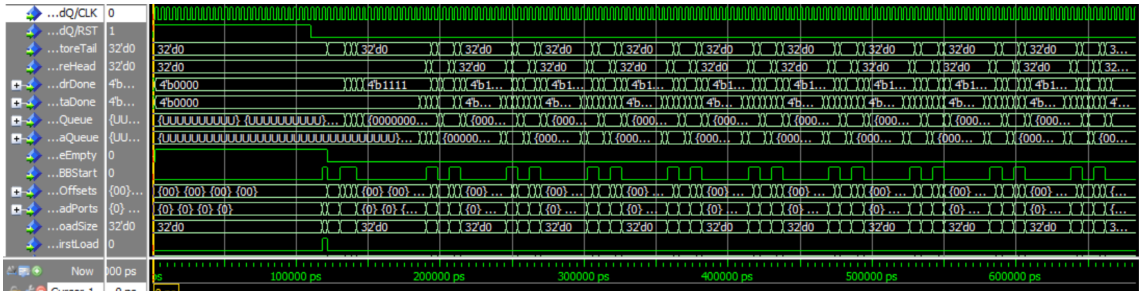


Figure 6.9: load-queue1

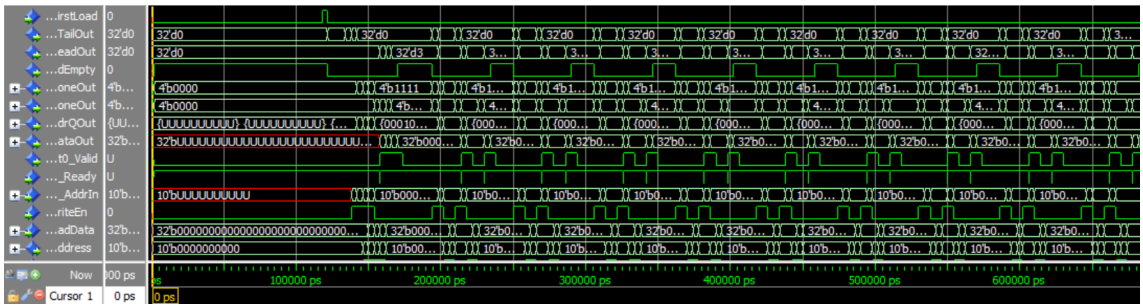


Figure 6.10: load-queue2

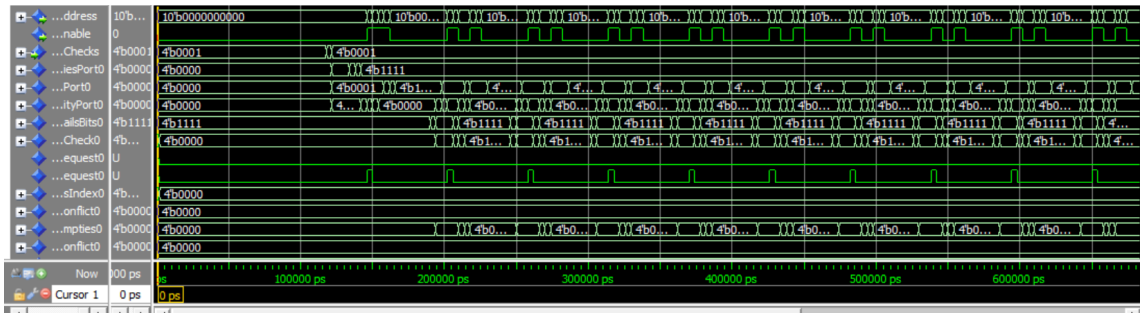


Figure 6.11: load-queue3

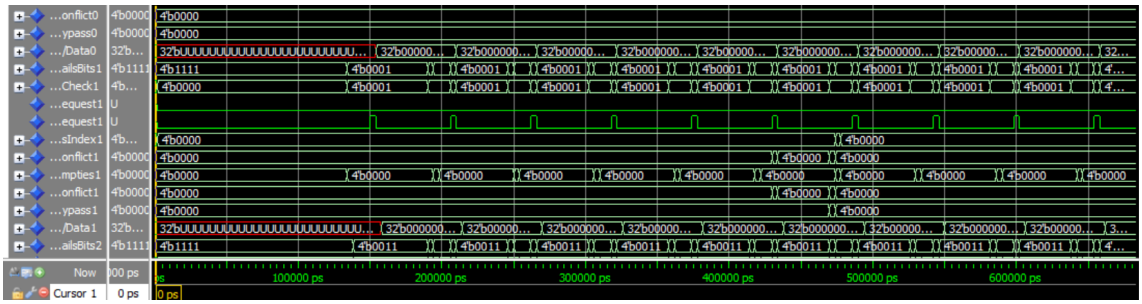


Figure 6.12: load-queue4

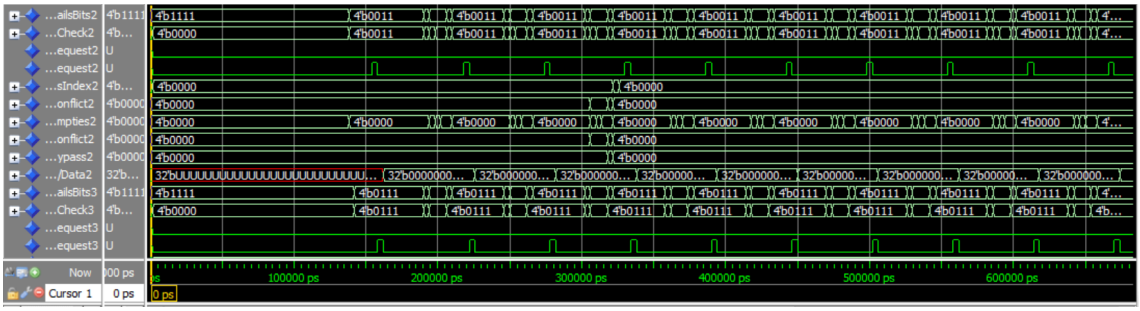


Figure 6.13: load-queue5

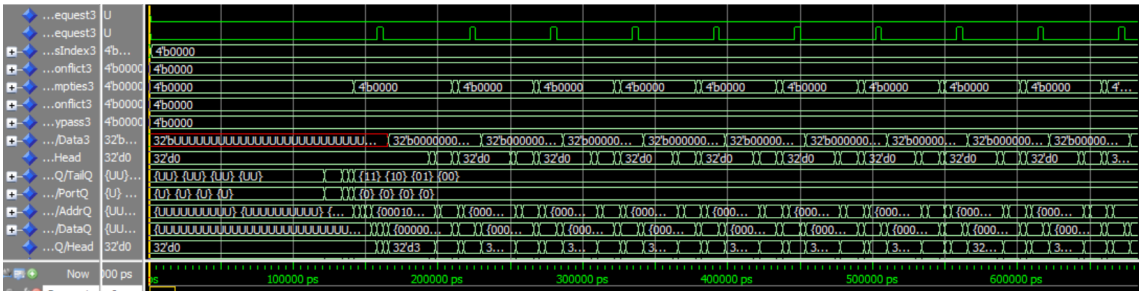


Figure 6.14: load-queue6

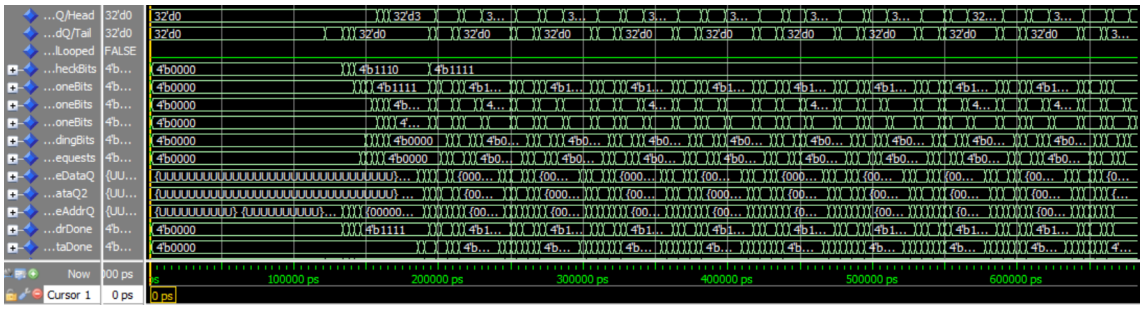


Figure 6.15: load-queue7

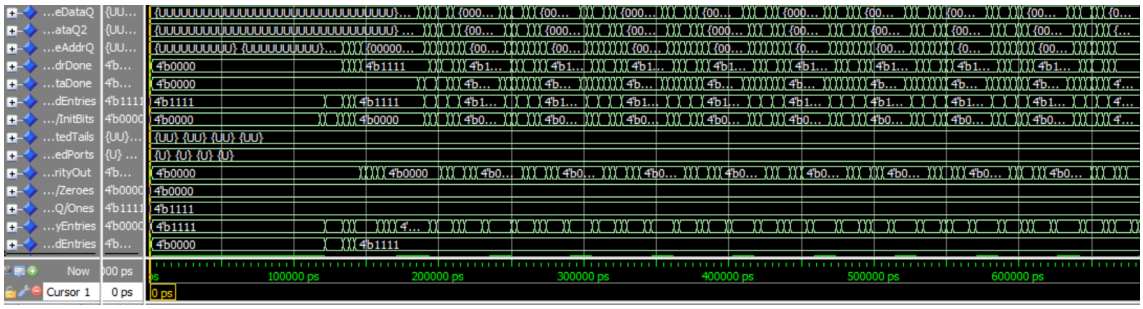


Figure 6.16: load-queue8

Chapter 7

Conclusion

This thesis works on Using high-level synthesis to design a load-store-queue for spatial computing through Vivado HLS tool. Since the design space can be explored more extensively and the optimum solution can be easily found.

From the contents we have discussed in the previous chapters, the performance of the top level function can be highly improve by adding some directives.

From the simulation result, we observed that our load-store-queue can match the expected, also it can be used for a dynamically scheduled design which described in chapter 3. But, to be honest, it is complicated for implementing. So what we should do next is to find an approach which is more simple than us with the same expected, and of course the higher performance the best.

Appendices

Listing 1: storeaddressport in SystemC

```
//file:store address port.h
#include "systemc.h"
const int FIFO_DEPTH = 4;
const int ADDRESS_SIZE = 10;
const int PORT_NUM = 1;
SC_MODULE(store_address_port){
sc_in<bool> CLK, pValid, port_enable, RST;
sc_out<bool> ReadyToPrev, send_address;
sc_in<sc_uint<ADDRESS_SIZE> > input_address;
sc_out<sc_uint<ADDRESS_SIZE> > address_to_mem;
sc_signal<sc_uint<FIFO_DEPTH+1> > counter1;
sc_signal<bool> send_address_internal;
void id_38();
void prc_Counter();
void id_58();
void id_59();
void id_60();

SC_CTOR(store_address_port){
    SC_METHOD(id_38);
    SC_METHOD(prc_Counter);
     sensitive_pos<<CLK;
     sensitive<<RST;
    SC_METHOD(id_58);
    SC_METHOD(id_59);
    SC_METHOD(id_60);

}
};
```

Listing 2: storeaddressport in SystemC

```
#include "storeaddressport.h"
void store_address_port :: id_38(){
        address_to_mem = input_address.read();
}
void store_address_port :: prc_Counter(){
        sc_uint<FIFO_DEPTH +1> count;
  if(RST.read()==1){
          count = 0;
    counter1 = count;
  }
   else if(port_enable.read() == 1)
        counter1 = counter1.read() + 1;
    else if(send_address_internal == 1)
        counter1 = counter1.read() - 1;
}
void store_address_port :: id_58(){
      sc_uint<FIFO_DEPTH+1> temp = counter1.read();
        if((temp > 0) & (pValid.read() == 1))
                send_address_internal = 1;
                else
                        send_address_internal = 0;
}
void store_address_port :: id_59(){
        sc_uint<FIFO_DEPTH+1> temp = counter1.read();
```

```
25                      if((temp > 0) & (pValid.read() == 1))
26                              send_address.write(1);
27                      else
28                              send_address.write(0);
29  }
30  void store_address_port :: id_60(){
31          sc_uint<FIFO_DEPTH+1> temp = counter1.read();
32          if(temp > 0)
33                              ReadyToPrev.write(1);
34                      else
35                              ReadyToPrev.write(0);
36  }
```

Listing 3: storeaddressport in SystemC

```
#include "systemc.h"
const int FIFO_DEPTH = 4 ;
const int DATA_SIZE = 32 ;
const int PORT_COUNT = 1 ;
SC_MODULE(store_data_port){
sc_in<bool> rst,clk,pValid1,port_enalbe1;
sc_out<bool> readytoprev1,send_data1;
sc_in<sc_uint<DATA_SIZE> > input_data1;
sc_out<sc_uint<DATA_SIZE> > data_to_mem1;
sc_signal< sc_uint <FIFO_DEPTH + 1> > counter2;
sc_signal<bool> send_data_internal1;
void id_104();
void prc_Counter();
void id_125();
void id_126();
void id_127();
SC_CTOR(store_data_port){
        SC_METHOD(id_104);
    SC_METHOD(prc_Counter);
    sensitive<<clk.pos();
    SC_METHOD(id_125);
    SC_METHOD(id_126);
    SC_METHOD(id_127);
}
};
```

Listing 4: storeaddressport in SystemC

```
#include "storedataport.h"
void store_data_port :: id_104(){
        data_to_mem1 = input_data1.read();
}
void store_data_port :: prc_Counter(){
        if(rst.read() == 1)
                counter2 = 0;
        else if(port_enalbe1 == 1)
         counter2 = counter2.read() + 1;
        else if(send_data_internal1 == 1)
        counter2 = counter2.read() - 1;
}
void store_data_port :: id_125(){
    sc_uint<FIFO_DEPTH + 1> temp = counter2.read();
    if((temp > 0)&(pValid1 == 1))
        send_data_internal1 = 1;
                else
                        send_data_internal1 = 0;
}
void store_data_port :: id_126(){
    sc_uint<FIFO_DEPTH + 1> temp = counter2.read();
        if((temp > 0)&(pValid1 == 1))
          send_data1 = 1;
                else
                        send_data1 = 0;
}
void store_data_port :: id_127(){
        sc_uint<FIFO_DEPTH + 1> temp = counter2.read();
```

```
29          if(temp > 0)
30                  readytoprev1 = 1;
31                  else
32                          readytoprev1 = 0;
33  }
```

Listing 5: storeaddressport in SystemC

```
#include "systemc.h"
const int FIFO_DEPTH = 4;
const int DATA_SIZE = 32;
const int PORT_NUM = 1;
const int ADDRESS_SIZE = 10 ;
SC_MODULE(load_port){
sc_in<bool> rst,clk,pValid2,port_enable2,nReady2,valid_from_mem2;
sc_out<bool> readytoprev2,send_address2,ValidToNext2,ready_to_mem2;
sc_in<sc_uint<ADDRESS_SIZE> > input_addr2;
sc_in<sc_uint<DATA_SIZE> > data_from_mem2;
sc_out<sc_uint<DATA_SIZE> > data_out2;
sc_out<sc_uint<ADDRESS_SIZE> > address_to_mem2;
sc_signal<sc_uint<FIFO_DEPTH+1> > counter3;
sc_signal<bool> send_address_internal2;
void prc_Counter();
void id_178();
void id_198();
void id_199();
void id_200();
void id_203();
void id_204();
void id_205();
SC_CTOR(load_port){
        SC_METHOD(id_178);
        SC_METHOD(prc_Counter);
        sensitive<<clk.pos();
        SC_METHOD(id_198);

        SC_METHOD(id_199);

        SC_METHOD(id_200);
        SC_METHOD(id_203);
        SC_METHOD(id_204);
        SC_METHOD(id_205);
}
};
```

Listing 6: storeaddressport in SystemC

```
#include "loadport.h"
void load_port :: id_178(){
        address_to_mem2 = input_addr2.read();
}
void load_port :: prc_Counter(){
        if(rst.read() == 1)
                counter3 = 0;
         else if(port_enable2 == 1)
        counter3 = counter3.read() + 1;
          else if(send_address_internal2 == 1)
        counter3 = counter3.read() - 1;

}
void load_port :: id_198(){
        sc_uint<FIFO_DEPTH+1> temp = counter3.read();
        if((temp > 0) & (pValid2 == 1))
                send_address_internal2 = 1;
```

```
18                  else
19                          send_address_internal2 = 0;
20  }
21  void load_port :: id_199(){
22          sc_uint<FIFO_DEPTH+1> temp = counter3.read();
23          if((temp > 0) & (pValid2 == 1))
24                  send_address2 = 1;
25                  else
26                          send_address2 = 0;
27  }
28  void load_port :: id_200(){
29          sc_uint<FIFO_DEPTH+1> temp = counter3.read();
30          if(temp > 0)
31                  readytoprev2 = 1;
32                  else
33                          readytoprev2 = 0;
34  }
35  void load_port :: id_203(){
36              ready_to_mem2 = nReady2.read();
37  }
38  void load_port :: id_204(){
39                  data_out2 = data_from_mem2.read();
40  }
41  void load_port :: id_205(){
42                  ValidToNext2 = valid_from_mem2.read();
43  }
```

Listing 7: storeaddressport in SystemC

```
#include "systemc.h"
const sc_uint<4> BB0StoreSize = 1;
const sc_uint<4> BB0LoadSize = 1 ;
const int FIFO_DEPTH = 4;
const int TAIL_SIZE = 2 ;
const int PORT_SIZE = 1;
const int STORE_MAX = 3;
const int LOAD_MAX = 3;
SC_MODULE(BB_START){
static const sc_biguint<TAIL_SIZE> BB0LoadOffsets[4];
static const sc_biguint<TAIL_SIZE> BB0StoreOffsets[4];
static const sc_biguint<PORT_SIZE> BB0StorePorts[4];
static const sc_biguint<PORT_SIZE> BB0LoadPorts[4];
sc_in<bool> clk,rst,BBStart0,LoadEmpty,StoreEmpty;
sc_in<sc_uint<FIFO_DEPTH> > StoreTail,StoreHead,LoadTail,LoadHead;
sc_out<sc_biguint<TAIL_SIZE> > BBLoadOffsets[4];
sc_out<sc_biguint<TAIL_SIZE> > BBStoreOffsets[4];
sc_out<sc_biguint<PORT_SIZE> > BBLoadPorts[4];
sc_out<sc_biguint<PORT_SIZE> > BBStorePorts[4];
sc_out<sc_uint<LOAD_MAX+1> > BBLoadSize;
sc_out<sc_uint<STORE_MAX+1> > BBStoreSize;
sc_out<sc_uint<FIFO_DEPTH> > LoadChecks,StoreChecks;
sc_out<bool> BBStart,BBFirstLoad,BBFirstStore,ReadyToPrev0,
    RdPort0_Enable,WrPort0_Enable;
sc_signal<bool> BBFirstLoadInternal,BBFirstStoreInternal,BB0Allocate;
sc_signal<sc_uint<FIFO_DEPTH> > EmptyStoreSlots,EmptyLoadSlots;
void prc_CalculateLoadSlots_proc();
void prc_CalculateStoreSlots_proc();
void prc_EmptySlotComparator1_proc();
void prc_LoadOffsets_proc();
void prc_StoreOffsets_proc();
void prc_BBSize_proc();
void prc_BBPorts_proc();
void id_397();
void prc_BBready0();
void id_419();
void id_420();
void id_426();
void id_427();
void id_429();
void id_430();
SC_CTOR(BB_START){
//#pragma HLS ARRAY_PARTITION variable=BB0LoadPorts complete dim=1
//#pragma HLS ARRAY_PARTITION variable=BB0StorePorts complete dim=1
//#pragma HLS ARRAY_PARTITION variable=BB0StoreOffsets complete dim=1
//#pragma HLS ARRAY_PARTITION variable=BB0LoadOffsets complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBStorePorts complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBStoreOffsets complete dim=1
//#pragma HLS ARRAY_PARTITION variable=BBLoadPorts complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBLoadOffsets complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBLoadPorts complete dim=1
                    SC_METHOD(prc_CalculateLoadSlots_proc);
                    sensitive<<clk.pos();
                    sensitive<<LoadHead<<LoadTail<<LoadEmpty;
                    SC_METHOD(prc_CalculateStoreSlots_proc);
                    sensitive<<clk.pos();
                    sensitive<<StoreHead<<StoreTail<<StoreEmpty;
```

```
57                  SC_METHOD(prc_EmptySlotComparator1_proc);
58                  sensitive<<clk.pos();
59                  sensitive<<BBStart0;
60                  SC_METHOD(prc_LoadOffsets_proc);
61                  sensitive<<clk.pos();
62                  sensitive<<StoreTail;
63                  SC_METHOD(prc_StoreOffsets_proc);
64                  sensitive<<clk.pos();
65                  sensitive<<StoreTail;
66                  SC_METHOD(prc_BBSize_proc);
67                  sensitive<<clk.pos();
68                  sensitive<<BB0Allocate;
69                  SC_METHOD(prc_BBPorts_proc);
70                  sensitive<<clk.pos();
71                  sensitive<<BB0Allocate;
72                  SC_METHOD(id_397);
73                  sensitive<<clk.pos();
74                  sensitive<<BB0Allocate;
75                  SC_METHOD(prc_BBready0);
76                  sensitive<<clk.pos();
77                  sensitive<<rst;
78                  SC_METHOD(id_419);
79                  sensitive<<clk.pos();
80                  sensitive<<BB0Allocate;
81                  SC_METHOD(id_420);
82                  sensitive<<clk.pos();
83                  sensitive<<BB0Allocate;
84                  SC_METHOD(id_426);
85                  sensitive<<clk.pos();
86                  sensitive<<StoreEmpty;
87                  SC_METHOD(id_427);
88                  sensitive<<clk.pos();
89                  SC_METHOD(id_429);
90                  sensitive<<clk.pos();
91                  sensitive<<BB0Allocate<<StoreEmpty;
92                  SC_METHOD(id_430);
93                  sensitive<<clk.pos();
94          }
95  };
96  //#pragma HLS ARRAY_PARTITION variable=BBLoadPorts complete dim=1
97  const sc_biguint<2> BB_START::BB0LoadOffsets[4] = {00,00,00,00};
98  const sc_biguint<2> BB_START::BB0StoreOffsets[4] = {00,00,00,01};
99  const sc_biguint<1> BB_START::BB0StorePorts[4] = {0,0,0,0};
100 const sc_biguint<1> BB_START::BB0LoadPorts[4] = {1,0,0,0};
```

Listing 8: storeaddressport in SystemC

```
1  #include "bbstart.h"
2  void BB_START::prc_CalculateLoadSlots_proc(){
3  //#pragma HLS DATAFLOW
4           LoadHead = 0;
5           sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
6           sc_uint<FIFO_DEPTH> loadtail = LoadTail.read();
7  if(rst ==1)
8                   EmptyLoadSlots = 4;
9   else if ((loadhead < loadtail) | (LoadEmpty == 1))
10     EmptyLoadSlots.write(loadhead-loadtail+FIFO_DEPTH);
11     else
12       EmptyLoadSlots.write(loadhead-loadtail) ;
13 }
14 void BB_START::prc_CalculateStoreSlots_proc(){
```

```
15 //#pragma HLS DATAFLOW
16          sc_uint<FIFO_DEPTH> storetail = StoreTail.read();
17          sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
18 if(rst == 1)
19                   EmptyStoreSlots = 4;
20  else if((storehead < storetail) | (StoreEmpty == 1))
21     EmptyStoreSlots.write(storehead-storetail+FIFO_DEPTH);
22     else
23           EmptyStoreSlots.write(storehead-storetail);
24 }
25 void BB_START ::prc_EmptySlotComparator1_proc(){
26 //#pragma HLS DATAFLOW
27 if(rst == 1)
28   BB0Allocate = 0;
29   if(BBStart0 == 1)
30     if((EmptyStoreSlots.read() == BB0StoreSize ) & (EmptyLoadSlots.read
         () == BB0LoadSize))
31       BB0Allocate = 1;
32      else
33       BB0Allocate = 0;
34 }
35 void BB_START ::prc_LoadOffsets_proc(){
36 //#pragma HLS DATAFLOW
37   sc_uint<4> current_load_offset;
38   sc_uint<4> new_load_tail;
39   int i;
40   sc_biguint<2> temp[4] ;
41   if(BB0Allocate == 1){
42     for(i=0;i<FIFO_DEPTH;i++)
43       if(i < LOAD_MAX){
44        sc_uint<2> r = BB0LoadOffsets[i].to_int();
45        current_load_offset = r;
46        new_load_tail = (StoreTail.read()+current_load_offset) %
           FIFO_DEPTH;
47        BBLoadOffsets[i].write(new_load_tail.to_int());
48       }
49       else
50           BBLoadOffsets[i] = "00";
51   }
52     else{
53            BBLoadOffsets[0] = "00";
54        BBLoadOffsets[1] = "00";
55        BBLoadOffsets[2] = "00";
56        BBLoadOffsets[3] = "00";
57     }
58 }
59 void BB_START ::prc_StoreOffsets_proc (){
60 //#pragma HLS DATAFLOW
61   int current_store_offset;
62   int new_store_tail;
63   sc_biguint<2> temp[4];
64   int i;
65   if(BB0Allocate == 1){
66     for(i=0;i<FIFO_DEPTH;i++)
67       if(i<STORE_MAX){
68         current_store_offset = BB0StoreOffsets[i].to_int();
69         new_store_tail = (StoreTail.read()+current_store_offset) %
            FIFO_DEPTH;
70         BBStoreOffsets[i] = new_store_tail;
71       }
72       else
73           BBStoreOffsets[i] = "00";
```

```
74    }
75    else{
76              BBStoreOffsets[0] = "00";
77          BBStoreOffsets[1] = "00";
78          BBStoreOffsets[2] = "00";
79          BBStoreOffsets[3] = "00";
80    }
81  }
82  void BB_START ::prc_BBSize_proc(){
83  //#pragma HLS DATAFLOW
84    if(BB0Allocate == 1){
85              BBStoreSize = BB0StoreSize;
86              BBLoadSize= BB0LoadSize;
87    }
88      else{
89      BBStoreSize = 0;
90      BBLoadSize = 0;
91      }
92  }
93  void BB_START ::prc_BBPorts_proc(){
94  //#pragma HLS DATAFLOW
95    if(BB0Allocate == 1){
96              BBLoadPorts[0] = BB0LoadPorts[0];
97              BBLoadPorts[1] = BB0LoadPorts[1];
98              BBLoadPorts[2] = BB0LoadPorts[2];
99              BBLoadPorts[3] = BB0LoadPorts[3];
100
101             BBStorePorts[0] = BB0StorePorts[0];
102             BBStorePorts[1] = BB0StorePorts[1];
103             BBStorePorts[2] = BB0StorePorts[2];
104             BBStorePorts[3] = BB0StorePorts[3];
105
106   }
107   else{
108             BBLoadPorts[0] = "0";
109             BBLoadPorts[1] = "0";
110             BBLoadPorts[2] = "0";
111             BBLoadPorts[3] = "0";
112
113         BBStorePorts[0] = "0";
114         BBStorePorts[1] = "0";
115         BBStorePorts[2] = "0";
116         BBStorePorts[3] = "0";
117
118   }
119 }
120 void BB_START ::id_397(){
121 //#pragma HLS DATAFLOW
122
123                   BBStart.write(BB0Allocate.read());
124
125 }
126 void BB_START ::prc_BBready0 (){
127 //#pragma HLS DATAFLOW
128   if(rst == 1){
129     ReadyToPrev0 = 1;
130   }
131     else if((EmptyStoreSlots.read() == BB0StoreSize)&(EmptyLoadSlots.
          read() == BB0LoadSize)){
132       ReadyToPrev0 = 1;
133     }
134     else
```

```
135         ReadyToPrev0 = 0;
136
137 }
138 void BB_START::id_419(){
139 //#pragma HLS DATAFLOW
140
141                     RdPort0_Enable.write(BB0Allocate.read());
142
143 }
144 void BB_START::id_420(){
145 //#pragma HLS DATAFLOW
146
147                     WrPort0_Enable.write(BB0Allocate.read());
148
149 }
150 void BB_START::id_426(){
151 //#pragma HLS DATAFLOW
152
153                     BBFirstLoad.write(BB0Allocate.read() & StoreEmpty.read()
                        );
154
155 }
156 void BB_START::id_427(){
157 //#pragma HLS DATAFLOW
158
159                     BBFirstStore = 0;
160
161 }
162 void BB_START::id_429(){
163 //#pragma HLS DATAFLOW
164           if((BB0Allocate == 1) & (StoreEmpty == 1))
165                     LoadChecks.write("0000");
166                     else
167                               LoadChecks.write("0001");
168
169 }
170 void BB_START::id_430(){
171 //#pragma HLS DATAFLOW
172                     StoreChecks.write("0001");
173 }
```

Listing 9: storeaddressport in SystemC

```
#include "systemc.h"
const int FIFO_DEPTH =4;
const int DATA_SIZE =32;
const int ADDRESS_SIZE = 10 ;
const int PORT_NUM = 1;
const int TAIL_NUM = 2;
const sc_uint<FIFO_DEPTH> IntCheckBits = "0000";
SC_MODULE(store_queue){
sc_in<bool> rst, clk, LoadEmpty, BBStart, Port0_DataWriteEn, BBFirstStore,
    Port0_AddrWriteEn;
sc_in<sc_uint<FIFO_DEPTH> > LoadTail, LoadHead, BBStoreSize;
sc_in<sc_uint<FIFO_DEPTH> > LoadAdrDone, LoadDataDone, StoreChecks;
sc_in<sc_uint<DATA_SIZE> > DataInPort0;
sc_in<sc_uint<ADDRESS_SIZE> > AddrInPort0;
sc_in<sc_biguint<ADDRESS_SIZE> > LoadAddressQueue[4];
sc_in<sc_biguint<TAIL_NUM> > BBStoreOffsets[4];
sc_in<sc_biguint<PORT_NUM> > BBStorePorts[4];
sc_out<sc_biguint<ADDRESS_SIZE> > StoreAddrQOut[4];
sc_out<sc_biguint<DATA_SIZE> > StoreDataQOut[4];
sc_out<bool> StoreEmpty, MemoryStoreEnable;
sc_out<sc_uint<FIFO_DEPTH> > StoreTailOut, StoreHeadOut;
sc_out<sc_uint<FIFO_DEPTH> > StoreAdrDoneOut, StoreDataDoneOut;
sc_out<sc_uint<DATA_SIZE> > MemoryDataOut;
sc_out<sc_uint<ADDRESS_SIZE> > MemoryAddressOut;
sc_signal<sc_uint<FIFO_DEPTH> > EntriesPort0, InputAddrPriorityPort0,
    InputDataPriorityPort0;
sc_signal<sc_uint<FIFO_DEPTH> > CheckBits, AdrDoneBits, DataDoneBits,
    AllDoneBits, InitBits;
sc_signal<sc_uint<FIFO_DEPTH> > ShiftedAdrDone, ShiftedDataDone;
sc_signal<sc_uint<FIFO_DEPTH> > ValidEntries, HeadTailsBits,
    HeadEntriesToCheck, Zeroes, Ones, EmptyEntries, AllocatedEntries;
sc_signal<sc_uint<FIFO_DEPTH> > PreviousLoadHead, Head, Tail;
sc_signal<sc_biguint<TAIL_NUM> > TailQ[4], ShiftedTails[4];
sc_signal<sc_biguint<PORT_NUM> > PortQ[4], ShiftedPorts[4];
sc_signal<sc_biguint<ADDRESS_SIZE> > ShiftedLoadAddrQ[4];
sc_signal<sc_uint<ADDRESS_SIZE> > AddrQ[4];
sc_signal<sc_uint<DATA_SIZE> > DataQ[4];
sc_signal<bool> StoreRequest;
void id_517();
void id_518();
void prc_TailUpdate_proc();
void prc_InitializeValues_proc();
void prc_TailQUpdate_proc();
void prc_PortQUpdate_proc();
void prc_AllocatedBits_proc();
void prc_EmptyEntries_proc ();
void prc_Empty_proc();
void prc_CheckBits_proc();
void id_672();
void id_673();
void id_674();
void id_675();
void prc_ShiftLoadQ();
void prc_HeadEntriesToCheck_proc();
void prc_ShiftHeadTails_proc();
void prc_HeadDependencyCheck_proc();
void id_782();
```

```
54  void id_783();
55  void id_784();
56  void prc_UpdateAllDoneBits_proc();
57  void prc_SearchPortQPort0_proc();
58  void prc_InputAddrPriorityPort0_proc();
59  void prc_InputDataPriorityPort0_proc();
60  void prc_UpdateAddressBits();
61  void prc_UpdateDataBits();
62  void id_904();
63  void id_905();
64  SC_CTOR(store_queue){
65  #pragma HLS ARRAY_PARTITION variable=BBStorePorts complete dim=1
66  #pragma HLS ARRAY_PARTITION variable=BBStoreOffsets complete dim=1
67  #pragma HLS ARRAY_PARTITION variable=LoadAddressQueue complete dim=1
68  #pragma HLS ARRAY_PARTITION variable=PortQ complete dim=1
69  #pragma HLS ARRAY_PARTITION variable=TailQ complete dim=1
70  #pragma HLS ARRAY_PARTITION variable=DataQ complete dim=1
71  #pragma HLS ARRAY_PARTITION variable=AddrQ complete dim=1
72  #pragma HLS ARRAY_PARTITION variable=ShiftedLoadAddrQ complete dim=1
73  #pragma HLS ARRAY_PARTITION variable=ShiftedPorts complete dim=1
74  #pragma HLS ARRAY_PARTITION variable=ShiftedTails complete dim=1
75  #pragma HLS ARRAY_PARTITION variable=StoreDataQOut complete dim=1
76  #pragma HLS ARRAY_PARTITION variable=StoreAddrQOut complete dim=1
77  #pragma HLS ARRAY_PARTITION variable=BBStorePorts complete dim=1
78  #pragma HLS ARRAY_PARTITION variable=BBStoreOffsets complete dim=1
79  #pragma HLS ARRAY_PARTITION variable=LoadAddressQueue complete dim=1
80
81
82  SC_METHOD(id_517);
83   sensitive<<clk.pos();
84  SC_METHOD(id_518);
85   sensitive<<clk.pos();
86  SC_METHOD(prc_TailUpdate_proc);
87   sensitive<<clk.pos();
88  SC_METHOD(prc_InitializeValues_proc);
89   sensitive<<clk.pos();
90   sensitive<<BBStart<<Tail<<BBStoreSize;
91  SC_METHOD(prc_TailQUpdate_proc);
92   sensitive<<clk.pos();
93  SC_METHOD(prc_PortQUpdate_proc);
94   sensitive<<clk.pos();
95  SC_METHOD(prc_AllocatedBits_proc);
96   sensitive<<clk.pos();
97  SC_METHOD(prc_EmptyEntries_proc);
98   sensitive<<clk.pos();
99   sensitive<<AllDoneBits<<AllocatedEntries<<AdrDoneBits<<AllocatedEntries;
100 SC_METHOD(prc_Empty_proc);
101  sensitive<<clk.pos();
102  sensitive<<EmptyEntries;
103 SC_METHOD(prc_CheckBits_proc);
104  sensitive<<clk.pos();
105 SC_METHOD(id_672);
106  sensitive<<clk.pos();
107 SC_METHOD(id_673);
108  sensitive<<clk.pos();
109 SC_METHOD(id_674);
110  sensitive<<clk.pos();
111 SC_METHOD(id_675);
112  sensitive<<clk.pos();
113 SC_METHOD(prc_ShiftLoadQ);
114  sensitive<<clk.pos();
115  sensitive<<LoadHead<<LoadTail<<LoadDataDone<<LoadAdrDone;
```

```
116 SC_METHOD(prc_HeadEntriesToCheck_proc);
117 sensitive<<clk.pos();
118 sensitive<<Head<<CheckBits<<ValidEntries<<HeadTailsBits;
119 SC_METHOD(prc_ShiftHeadTails_proc);
120 sensitive<<clk.pos();
121 sensitive<<LoadHead<<Head;
122 SC_METHOD(prc_HeadDependencyCheck_proc);
123 sensitive<<clk.pos();
124 sensitive<<Head<<ShiftedDataDone<<ShiftedAdrDone<<HeadEntriesToCheck<<
        AdrDoneBits<<DataDoneBits<<AllDoneBits;
125 SC_METHOD(id_782);
126 sensitive<<clk.pos();
127 SC_METHOD(id_783);
128 sensitive<<clk.pos();
129 SC_METHOD(id_784);
130 sensitive<<clk.pos();
131 SC_METHOD(prc_UpdateAllDoneBits_proc);
132 sensitive<<clk.pos();
133 SC_METHOD(prc_SearchPortQPort0_proc);
134 sensitive<<clk.pos();
135 SC_METHOD(prc_InputAddrPriorityPort0_proc);
136 sensitive<<clk.pos();
137 sensitive<<Head<<EntriesPort0<<AdrDoneBits;
138 SC_METHOD(prc_InputDataPriorityPort0_proc);
139 sensitive<<clk.pos();
140 sensitive<<Head<<EntriesPort0<<DataDoneBits;
141 SC_METHOD(prc_UpdateAddressBits);
142 sensitive<<clk.pos();
143 SC_METHOD(prc_UpdateDataBits);
144 sensitive<<clk.pos();
145 SC_METHOD(id_904);
146 sensitive<<clk.pos();
147 sensitive<<Tail;
148 SC_METHOD(id_905);
149 sensitive<<clk.pos();
150 sensitive<<Head;
151 }
152 };
```

Listing 10: storeaddressport in SystemC

```
1  #include "storequeue.h"
2  void store_queue :: id_517(){
3          sc_uint<FIFO_DEPTH> temp;
4                  temp[0] = 0;
5                  temp[1] = 0;
6                  temp[2] = 0;
7                  temp[3] = 0;
8          Zeroes = temp;
9  }
10 void store_queue :: id_518(){
11 //#pragma HLS DATAFLOW
12         sc_uint<FIFO_DEPTH> temp;
13         temp[0] = 1;
14         temp[1] = 1;
15         temp[2] = 1;
16         temp[3] = 1;
17         Ones = temp;
18 }
19 void store_queue :: prc_TailUpdate_proc(){
20 //#pragma HLS DATAFLOW
```

```
21 //#pragma HLS DATAFLOW
22         sc_uint<FIFO_DEPTH> tail ;
23         sc_uint <FIFO_DEPTH> bbstoresize = BBStoreSize.read();
24
25    if(rst == 1)
26       Tail = 0;
27     else if(BBStart == 1)
28       Tail = (tail + bbstoresize) % FIFO_DEPTH;
29 }
30 void store_queue ::prc_InitializeValues_proc(){
31 //#pragma HLS DATAFLOW
32 //#pragma HLS DATAFLOW
33   int i;
34   sc_uint<FIFO_DEPTH> temp ;
35  if(rst == 1){
36                   temp[0] = 0;
37                   temp[1] = 0;
38                   temp[2] = 0;
39                   temp[3] = 0;
40         InitBits = temp;
41  }
42   else if(BBStart == 1)
43     for(i=0;i<FIFO_DEPTH;i++)
44       if(i < BBStoreSize.read())
45           temp[(Tail.read() + i) % FIFO_DEPTH] = 1;
46         InitBits = temp;
47 }
48 void store_queue ::prc_TailQUpdate_proc(){
49 //#pragma HLS DATAFLOW
50 //#pragma HLS DATAFLOW
51          int i;
52          sc_uint<FIFO_DEPTH> tail = Tail.read();
53          sc_uint <FIFO_DEPTH> bbstoresize = BBStoreSize.read();
54     for(i=0;i<FIFO_DEPTH;i++)
55       if(i < bbstoresize)
56          TailQ[(tail + i) % FIFO_DEPTH] = BBStoreOffsets[i].read();
57 }
58 void store_queue ::prc_PortQUpdate_proc(){
59 //#pragma HLS DATAFLOW
60 //#pragma HLS DATAFLOW
61           int i;
62           sc_uint<FIFO_DEPTH> tail = Tail.read();
63           sc_uint <FIFO_DEPTH> bbstoresize = BBStoreSize.read();
64     for(i=0;i<FIFO_DEPTH;i++)
65       if(i < bbstoresize)
66         PortQ[(tail + i) % FIFO_DEPTH] = BBStorePorts[i].read();
67 }
68 void store_queue ::prc_AllocatedBits_proc(){
69 //#pragma HLS DATAFLOW
70 //#pragma HLS DATAFLOW
71          int j;
72          sc_uint<FIFO_DEPTH> r = InitBits.read();
73          sc_uint<FIFO_DEPTH> h;
74     if(rst == 1){
75         h[0] = 0;
76         h[1] = 0;
77         h[2] = 0;
78         h[3] = 0;
79      AllocatedEntries = h;
80     }
81      else
82        for(j=0;j<FIFO_DEPTH;j++)
```

```
83            if(r[j] == 1)
84              h[j] = 1;
85        AllocatedEntries = h;
86    }
87    void store_queue ::prc_EmptyEntries_proc(){
88    //#pragma HLS DATAFLOW
89    //#pragma HLS DATAFLOW
90              int i;
91              sc_uint<FIFO_DEPTH> temp1 = AllDoneBits.read();
92              sc_uint<FIFO_DEPTH> temp2 = AllocatedEntries.read();
93              sc_uint<FIFO_DEPTH> temp;
94    if(rst == 1){
95                      temp[0] = 1;
96                      temp[1] = 1;
97                      temp[2] = 1;
98                      temp[3] = 1;
99            EmptyEntries = temp;
100   }
101   else
102     for(i=0;i<FIFO_DEPTH;i++){
103       if((temp1[i] == 1) | (temp2[i] == 0))
104           temp[i] = 1;
105       else
106           temp[i] = 0;
107     }
108     EmptyEntries = temp;
109
110   }
111   void store_queue ::prc_Empty_proc(){
112   //#pragma HLS DATAFLOW
113
114     if( EmptyEntries.read() == Ones.read() )
115       StoreEmpty = 1;
116     else
117       StoreEmpty = 0;
118   }
119
120   void store_queue ::prc_CheckBits_proc(){
121   //#pragma HLS DATAFLOW
122   //#pragma HLS DATAFLOW
123     int i;
124     int j;
125     sc_uint<FIFO_DEPTH> current_tail;
126     sc_uint<FIFO_DEPTH> r ;
127     sc_uint<FIFO_DEPTH> temp;
128     sc_uint<FIFO_DEPTH> h = StoreChecks.read();
129     sc_uint<FIFO_DEPTH> previousloadhead ;
130     sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
131     sc_uint<FIFO_DEPTH> tail = Tail.read();
132     sc_uint<FIFO_DEPTH> bbstoresize = BBStoreSize.read();
133      if(rst == 1){
134               r[0] = 0;
135               r[1] = 0;
136               r[2] = 0;
137               r[3] = 0;
138       CheckBits = r;
139      }
140       else
141           PreviousLoadHead.write(loadhead);
142         for(j=0;j<FIFO_DEPTH;j++)
143           if((j < bbstoresize) & (BBStart == 1))
144                   r[(tail + j) % FIFO_DEPTH] = h[j];
```

```
145           else
146             current_tail = TailQ[j].read().to_int();
147             if((previousloadhead = (((current_tail-1) % FIFO_DEPTH) &
                    loadhead)) > current_tail)
148                   r[j] = 0;
149             else if((previousloadhead = (((current_tail-1) % FIFO_DEPTH) &
                     loadhead)) < PreviousLoadHead.read())
150                   r[j] = 0;
151         CheckBits = r;
152         PreviousLoadHead = previousloadhead;
153 }
154
155 void store_queue :: id_672(){
156 //#pragma HLS DATAFLOW
157         StoreAdrDoneOut = AdrDoneBits.read();
158 }
159 void store_queue :: id_673(){
160 //#pragma HLS DATAFLOW
161         StoreDataDoneOut = DataDoneBits.read();
162 }
163 void store_queue :: id_674(){
164 //#pragma HLS DATAFLOW
165  sc_biguint<ADDRESS_SIZE> addrq[4];
166     addrq[0] = AddrQ[0].read();
167     addrq[1] = AddrQ[1].read();
168     addrq[2] = AddrQ[2].read();
169     addrq[3] = AddrQ[3].read();
170         StoreAddrQOut[0] = addrq[0];
171         StoreAddrQOut[1] = addrq[1];
172         StoreAddrQOut[2] = addrq[2];
173         StoreAddrQOut[3] = addrq[3];
174 }
175 void store_queue :: id_675(){
176 //#pragma HLS DATAFLOW
177 sc_biguint<DATA_SIZE> dataq[4];
178         dataq[0] = DataQ[0].read();
179         dataq[1] = DataQ[1].read();
180         dataq[2] = DataQ[2].read();
181         dataq[3] = DataQ[3].read();
182         StoreDataQOut[0] = dataq[0];
183         StoreDataQOut[1] = dataq[1];
184         StoreDataQOut[2] = dataq[2];
185         StoreDataQOut[3] = dataq[3];
186 }
187 void store_queue ::prc_ShiftLoadQ(){
188 //#pragma HLS DATAFLOW
189 //#pragma HLS DATAFLOW
190   sc_uint<9> load_upper_bound;
191   int i;
192   sc_uint<FIFO_DEPTH> r;
193   sc_uint<FIFO_DEPTH> h = LoadAdrDone.read();
194   sc_uint<FIFO_DEPTH> b;
195   sc_uint<FIFO_DEPTH> a = LoadDataDone.read();
196   sc_uint<FIFO_DEPTH> v ;
197   sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
198   sc_uint<FIFO_DEPTH> loadtail = LoadTail.read();
199   sc_uint<FIFO_DEPTH> temp;
200   sc_uint<FIFO_DEPTH> temp1;
201   if(rst == 1){
202           v[0] = 1;
203           v[1] = 1;
204           v[2] = 1;
```

```
205             v[3] = 1;
206             ValidEntries = v;
207             r[0] = 0;
208             r[1] = 0;
209             r[2] = 0;
210             r[3] = 0;
211             ShiftedAdrDone = r;
212             b[0] = 0;
213             b[1] = 0;
214             b[2] = 0;
215             b[3] = 0;
216             ShiftedDataDone = b;
217    }
218   else if(loadtail == loadhead)
219      load_upper_bound = loadtail + FIFO_DEPTH;
220     else
221      load_upper_bound = loadtail;
222   for(i=0;i<FIFO_DEPTH;i++){
223         ShiftedLoadAddrQ[i] = LoadAddressQueue[(i+loadhead) % FIFO_DEPTH
                ].read() ;
224         r[i] = h[(i+loadhead) % FIFO_DEPTH];
225         b[i] = a[(i+loadhead) % FIFO_DEPTH];
226
227        if((i + loadhead) < load_upper_bound)
228           v[i] = 1;
229          else
230           v[i] = 0;
231     }
232      ValidEntries = v;
233      ShiftedAdrDone = r;
234      ShiftedDataDone = b;
235 }
236 void store_queue ::prc_HeadEntriesToCheck_proc(){
237 //#pragma HLS DATAFLOW
238 //#pragma HLS DATAFLOW
239    int i;
240    sc_uint<FIFO_DEPTH> v = ValidEntries.read();
241    sc_uint<FIFO_DEPTH> r ;
242    sc_uint<FIFO_DEPTH> h = HeadTailsBits.read();
243    sc_uint<FIFO_DEPTH> c = CheckBits.read();
244    sc_uint<FIFO_DEPTH> head = Head.read();
245 if(rst ==1){
246             r[0] = 0;
247             r[1] = 0;
248             r[2] = 0;
249             r[3] = 0;
250            HeadEntriesToCheck = r;
251 }
252 else
253           for(i=0;i<FIFO_DEPTH;i++)
254      r[i] = h[i] & v[i] & c[head.to_int()];
255    HeadEntriesToCheck = r;
256
257 }
258 void store_queue ::prc_ShiftHeadTails_proc(){
259 //#pragma HLS DATAFLOW
260 //#pragma HLS DATAFLOW
261    int current_tail;
262    sc_uint<9> load_upper_bound;
263    int i;
264    sc_uint<FIFO_DEPTH> h ;
265    sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
```

```
266   current_tail = TailQ[Head.read()].read().to_int();
267   if(rst == 1){
268           h[0] = 1;
269           h[1] = 1;
270           h[2] = 1;
271           h[3] = 1;
272           HeadTailsBits = h;
273   }
274   else if(((current_tail-1) % FIFO_DEPTH) == loadhead)
275     load_upper_bound = current_tail + FIFO_DEPTH;
276     else if((((current_tail-1) % FIFO_DEPTH) == loadhead) & (LoadEmpty
          == 1))
277       load_upper_bound = current_tail;
278       else
279       load_upper_bound = current_tail;
280   for(i=0;i<FIFO_DEPTH;i++)
281     if((i+loadhead) < load_upper_bound)
282       h[i] = 1;
283       else
284       h[i] = 0;
285   HeadTailsBits = h;
286 }
287 void store_queue ::prc_HeadDependencyCheck_proc(){
288 //#pragma HLS DATAFLOW
289 //#pragma HLS DATAFLOW
290   bool read_stall;
291   sc_uint<FIFO_DEPTH> r =  ShiftedAdrDone.read();
292   sc_uint<FIFO_DEPTH> h = HeadEntriesToCheck.read();
293   sc_uint<FIFO_DEPTH> d = DataDoneBits.read();
294   sc_uint<FIFO_DEPTH> s = ShiftedDataDone.read();
295   sc_uint<FIFO_DEPTH> a = AdrDoneBits.read();
296   sc_uint<FIFO_DEPTH> b = AllDoneBits.read();
297   sc_uint<FIFO_DEPTH> head = Head.read().to_int();
298   sc_uint<ADDRESS_SIZE> addrq[4] ;
299   sc_uint<ADDRESS_SIZE> shiftedloadaddrq[4];
300   int i;
301   if(rst == 1){
302     StoreRequest = 0;
303     }
304   else
305   if((((a[head] == 1) & (d[head] == 1)) & (b[head] == 0)))
306     read_stall = 0;
307     for(i=0;i<FIFO_DEPTH;i++){
308       if(h[i] == 1)
309         if((r[i] == 1) & (s[i] == 0))
310             addrq[0] = AddrQ[0].read();
311             shiftedloadaddrq[i] = ShiftedLoadAddrQ[i].read();
312           if(addrq[0] == shiftedloadaddrq[i])
313             read_stall = read_stall | 1;
314
315           if(r[i] == 0)
316            read_stall = read_stall | 1;
317     }
318     if(read_stall == 0)
319       StoreRequest = 1;
320 }
321 void store_queue :: id_782(){
322 //#pragma HLS DATAFLOW
323         MemoryStoreEnable = StoreRequest;
324 }
325 void store_queue :: id_783(){
326 //#pragma HLS DATAFLOW
```

```
327           sc_uint<FIFO_DEPTH> head = Head.read();
328           MemoryDataOut = DataQ[head].read();
329 }
330 void store_queue :: id_784(){
331 //#pragma HLS DATAFLOW
332           sc_uint<FIFO_DEPTH> head = Head.read();
333           MemoryAddressOut = AddrQ[head].read();
334 }
335 void store_queue ::prc_UpdateAllDoneBits_proc(){
336 //#pragma HLS DATAFLOW
337 //#pragma HLS DATAFLOW
338   int i;
339   sc_uint<FIFO_DEPTH> b ;
340   sc_uint<FIFO_DEPTH> r = InitBits.read();
341   sc_uint<FIFO_DEPTH> head = Head.read();
342   if(rst == 1){
343             AllDoneBits = "0000";
344             Head = 0;
345   }
346       else
347        for(i=0;i<FIFO_DEPTH;i++){
348          if(r[i] == 1)
349            b[i] = 0;
350          AllDoneBits = b;
351        }
352        if(StoreRequest == 1){
353          b[head] = 1;
354          AllDoneBits = b;
355          if(head == FIFO_DEPTH-1)
356            Head.write(0);
357            else
358            Head.write( head + 1);
359        }
360 }
361
362 void store_queue ::prc_SearchPortQPort0_proc(){
363 //#pragma HLS DATAFLOW
364   int i;
365   sc_uint<FIFO_DEPTH> r;
366   if(rst == 1)
367             EntriesPort0 = 0;
368   else
369   for(i=0;i<FIFO_DEPTH;i++){
370             sc_uint<PORT_NUM> h = PortQ[i].read();
371       if(h == 0)
372        r[i] = 1;
373        else
374        r[i] = 0;
375    EntriesPort0 = r;
376   }
377 }
378 void store_queue ::prc_InputAddrPriorityPort0_proc(){
379 //#pragma HLS DATAFLOW
380   bool blocked;
381   int i;
382   sc_uint<FIFO_DEPTH> a = AdrDoneBits.read();
383   sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
384   sc_uint<FIFO_DEPTH> h = InputAddrPriorityPort0.read();
385   sc_uint<FIFO_DEPTH> head = Head.read();
386   if(rst == 1){
387   blocked = 0;
388   InputAddrPriorityPort0 = "0000";
```

```
389     }
390     else
391     for(i=0;i<FIFO_DEPTH;i++)
392       if((r [(i + head) % FIFO_DEPTH] == 1) & (a[(i + head) % FIFO_DEPTH]
            == 0 )){
393         if(blocked == 0){
394           h[(i + head) % FIFO_DEPTH ] = 1;
395           InputAddrPriorityPort0 = h;
396           blocked = 1;
397         }
398       }
399
400 }
401 void store_queue ::prc_InputDataPriorityPort0_proc(){
402 //#pragma HLS DATAFLOW
403     int i;
404     bool blocked;
405     sc_uint<FIFO_DEPTH> d = DataDoneBits.read();
406     sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
407     sc_uint<FIFO_DEPTH> h = InputDataPriorityPort0.read();
408     sc_uint<FIFO_DEPTH> head = Head.read();
409     if(rst == 1){
410     blocked = 0;
411     InputDataPriorityPort0 = "0000";
412     }
413     else
414     for(i=0;i<FIFO_DEPTH;i++)
415       if((r[(i + head) % FIFO_DEPTH] == 1) & (d[(i + head) % FIFO_DEPTH]
            == 0 )){
416         if(blocked == 0){
417           h[(i+head) % FIFO_DEPTH ] = 1;
418           InputDataPriorityPort0 = h;
419           blocked = 1;
420         }
421       }
422 }
423 void store_queue ::prc_UpdateAddressBits(){
424 //#pragma HLS DATAFLOW
425             int i;
426             sc_uint<FIFO_DEPTH> a;
427             sc_uint<FIFO_DEPTH> r = InitBits.read();
428             sc_uint<FIFO_DEPTH> h = InputAddrPriorityPort0.read();
429             sc_uint<ADDRESS_SIZE> temp = AddrInPort0.read();
430       if(rst == 1)
431           AdrDoneBits = "0000";
432        else
433         for(i=0;i<FIFO_DEPTH;i++){
434           if(r[i] == 1)
435             a[i] = 0 ;
436             else if((h[i] == 1) & (a[i] == 0) & (Port0_AddrWriteEn == 1)){
437                   //   sc_uint<ADDRESS_SIZE> addrq = AddrQ[i].read();
438                     AddrQ[0].write(temp);
439                     AddrQ[i].write(temp);
440                 a[i] = 1;
441             }
442          AdrDoneBits = a;
443         }
444 }
445 void store_queue :: prc_UpdateDataBits(){
446 //#pragma HLS DATAFLOW
447           int i;
448           sc_uint<FIFO_DEPTH> d ;
```

```
449     sc_uint<DATA_SIZE> temp = DataInPort0.read();
450         sc_uint<FIFO_DEPTH> r = InitBits.read();
451         sc_uint<FIFO_DEPTH> h = InputAddrPriorityPort0.read();
452     if(rst == 1)
453         DataDoneBits = "0000";
454     else
455       for(i=0;i<FIFO_DEPTH;i++){
456         if(r[i] == 1)
457           d[i] = 0 ;
458           else if((h[i] == 1) & (d[i] == 0) & (Port0_DataWriteEn == 1)){
459            // sc_uint<DATA_SIZE> dataq = DataQ[i].read();
460                   DataQ[0].write(temp);
461                   DataQ[i].write(temp);
462               d[i] = 1 ;
463           }
464        DataDoneBits = d;
465       }
466 }
467 void store_queue :: id_904(){
468         StoreTailOut = Tail.read();
469 }
470 void store_queue :: id_905(){
471         StoreHeadOut = Head.read();
472 }
```

Listing 11: storeaddressport in SystemC

```
#include "systemc.h"
const int FIFO_DEPTH = 4;
const int DATA_SIZE = 32;
const int ADDRESS_SIZE = 10;
const sc_uint<FIFO_DEPTH> IntCheckBits = "0000";
SC_MODULE(load_queue){
sc_in<bool> rst,clk,StoreEmpty,BBStart,BBFirstLoad,Port0_Ready,
    Port0_LoadWriteEn;
sc_in<sc_uint<FIFO_DEPTH> > StoreTail,StoreHead,BBLoadSize;
sc_in<sc_uint<FIFO_DEPTH> > StoreAdrDone,StoreDataDone,LoadChecks;
sc_in<sc_uint<DATA_SIZE> > MemoryLoadData;
sc_in<sc_uint<ADDRESS_SIZE> > Port0_AddrIn;
sc_in<sc_biguint<ADDRESS_SIZE> > StoreAddressQueue[4];
sc_in<sc_biguint<DATA_SIZE> > StoreDataQueue[4];
sc_in<sc_biguint<2> > BBLoadOffsets[4];
sc_in<sc_biguint<1> > BBLoadPorts[4];
sc_out<sc_biguint<ADDRESS_SIZE> > LoadAddrQOut[4];
sc_out<bool> LoadEmpty,MemoryLoadEnable,Port0_Valid;
sc_out<sc_uint<FIFO_DEPTH> > LoadTailOut,LoadHeadOut;
sc_out<sc_uint<FIFO_DEPTH> > LoadAdrDoneOut,LoadDataDoneOut;
sc_out<sc_uint<DATA_SIZE> > Port0_DataOut;
sc_out<sc_biguint<ADDRESS_SIZE> > MemoryLoadAddress;
sc_signal<sc_uint<FIFO_DEPTH> > EntriesPort0,InputPriorityPort0,
    OutputPriorityPort0;
sc_signal<sc_uint<FIFO_DEPTH> > LoadTailsBits0,EntriesToCheck0,
    BypassIndex0,Conflict0,Empties0,LastConflict0,CanBypass0;
sc_signal<sc_uint<FIFO_DEPTH> > LoadTailsBits1,EntriesToCheck1,
    BypassIndex1,Conflict1,Empties1,LastConflict1,CanBypass1;
sc_signal<sc_uint<FIFO_DEPTH> > LoadTailsBits2,EntriesToCheck2,
    BypassIndex2,Conflict2,Empties2,LastConflict2,CanBypass2;
sc_signal<sc_uint<FIFO_DEPTH> > LoadTailsBits3,EntriesToCheck3,
    BypassIndex3,Conflict3,Empties3,LastConflict3,CanBypass3;
sc_signal<sc_uint<FIFO_DEPTH> > PreviousLoadHead,Head,Tail;
sc_signal<sc_uint<DATA_SIZE> > Data1,Data2,Data3;
sc_signal<sc_biguint<DATA_SIZE> > Data0;
sc_signal<bool> BypassRequest0,LoadRequest0,BypassRequest1,LoadRequest1,
    BypassRequest2,LoadRequest2,BypassRequest3,LoadRequest3;
sc_signal<sc_uint<FIFO_DEPTH> > PreviousStoreHead;
sc_signal<sc_uint<FIFO_DEPTH> > CheckBits,AdrDoneBits,DataDoneBits,
    AllDoneBits;
sc_signal<sc_uint<FIFO_DEPTH> > PendingBits,LoadRequests,InitBits,
    ShiftedAdrDone,ShiftedDataDone;
sc_signal<bool > LoadRequests0,LoadRequests1,LoadRequests2,LoadRequests3
    ;
sc_signal<sc_uint<FIFO_DEPTH> > ValidEntries,PriorityOut,Zeroes,Ones,
    EmptyEntries,AllocatedEntries;
sc_signal<sc_biguint<2> > ShiftedTails[4];
sc_signal<sc_uint<2> > TailQ[4];
sc_signal<sc_biguint<1> > ShiftedPorts[4];
sc_signal<sc_uint<1> > PortQ[4];
sc_signal<sc_biguint<ADDRESS_SIZE> > ShiftedStoreAddrQ[4];
sc_signal<sc_biguint<ADDRESS_SIZE> > AddrQ[4];
sc_signal<sc_biguint<DATA_SIZE> > ShiftedStoreDataQ[4],
    ShiftedStoreDataQ2[4];
sc_signal<sc_uint<DATA_SIZE> > DataQ[4];
void id_1044();
void id_1045();
```

```
46  void prc_TailUpdate_proc();
47  void prc_InitializeValues_proc();
48  void prc_TailQUpdate_proc();
49  void prc_PortQUpdate_proc();
50  void prc_Empty_proc();
51  void prc_EmptyEntries_proc ();
52  void prc_CheckBits_proc();
53  void prc_AllocatedBits_proc();
54  void prc_ShiftStoreQ();
55  void prc_PriorityFunction_proc();
56  void prc_PendingReg_proc();
57  void id_1198();
58  void id_1199();
59  void id_1200();
60  void id_1201();
61  void id_1202();
62  void prc_ShiftedDataQ_proc0();
63  void prc_ShiftLoadTails0_proc();
64  void prc_EntriesToCheck0_proc();
65  void prc_Comparators0();
66  void prc_FindLastConflict0();
67  void prc_Empties0_proc();
68  void prc_Decision0();
69  void id_1399();
70  void prc_ShiftLoadTails1_proc();
71  void prc_EntriesToCheck1_proc();
72  void prc_Comparators1();
73  void prc_FindLastConflict1();
74  void prc_Empties1_proc();
75  void prc_Decision1();
76  void id_1515();
77  void prc_ShiftLoadTails2_proc();
78  void prc_EntriesToCheck2_proc();
79  void prc_Comparators2();
80  void prc_FindLastConflict2();
81  void prc_Empties2_proc();
82  void prc_Decision2();
83  void id_1631();
84  void prc_ShiftLoadTails3_proc();
85  void prc_EntriesToCheck3_proc();
86  void prc_Comparators3();
87  void prc_FindLastConflict3();
88  void prc_Empties3_proc();
89  void prc_Decision3();
90  void id_1746();
91  void prc_ChooseMemRequest_proc();
92  void prc_DatadoneBits_proc();
93  void prc_DataQUpdate_proc0();
94  void id_1829();
95  void prc_DataQUpdate_proc1();
96  void id_1847();
97  void prc_DataQUpdate_proc2();
98  void id_1865();
99  void prc_DataQUpdate_proc3();
100 void id_1883();
101 void prc_SearchPortQPort0_proc();
102 void prc_InputPriorityPort0_proc();
103 void prc_OutputPriorityPort0_proc();
104 void prc_SendToPort0_proc();
105 void prc_UpdateAddressBits();
106 void prc_UpdateAllDoneBits_proc();
107 SC_CTOR(load_queue){
```

```
108 #pragma HLS ARRAY_PARTITION variable=AddrQ complete dim=1
109 #pragma HLS ARRAY_PARTITION variable=DataQ complete dim=1
110 //#pragma HLS ARRAY_PARTITION variable=TailQ complete dim=1
111 //#pragma HLS ARRAY_PARTITION variable=PortQ complete dim=1
112 //#pragma HLS ARRAY_PARTITION variable=ShiftedStoreDataQ2 complete dim=1
113 //#pragma HLS ARRAY_PARTITION variable=ShiftedStoreDataQ complete dim=1
114 //#pragma HLS ARRAY_PARTITION variable=ShiftedStoreAddrQ complete dim=1
115 //#pragma HLS ARRAY_PARTITION variable=ShiftedPorts complete dim=1
116 //#pragma HLS ARRAY_PARTITION variable=ShiftedTails complete dim=1
117 //#pragma HLS ARRAY_PARTITION variable=LoadAddrQOut complete dim=1
118 //#pragma HLS ARRAY_PARTITION variable=BBLoadPorts complete dim=1
119 //#pragma HLS ARRAY_PARTITION variable=BBLoadOffsets complete dim=1
120 //#pragma HLS ARRAY_PARTITION variable=StoreDataQueue complete dim=1
121 //#pragma HLS ARRAY_PARTITION variable=StoreAddressQueue complete dim=1
122 SC_METHOD(id_1044);
123  sensitive<<clk.pos();
124 SC_METHOD(id_1045);
125  sensitive<<clk.pos();
126 SC_METHOD(prc_TailUpdate_proc);
127  sensitive<<clk.pos();
128 SC_METHOD(prc_InitializeValues_proc);
129  sensitive<<clk.pos();
130  sensitive<<BBStart<<Tail<<BBLoadSize;
131 SC_METHOD(prc_TailQUpdate_proc);
132  sensitive<<clk.pos();
133 SC_METHOD(prc_PortQUpdate_proc);
134  sensitive<<clk.pos();
135 SC_METHOD(prc_CheckBits_proc);
136  sensitive<<clk.pos();
137 SC_METHOD(prc_AllocatedBits_proc);
138  sensitive<<clk.pos();
139 SC_METHOD(prc_Empty_proc);
140  sensitive<<clk.pos();
141  sensitive<<EmptyEntries;
142 SC_METHOD(prc_EmptyEntries_proc);
143  sensitive<<clk.pos();
144  sensitive<<AllDoneBits<<DataDoneBits<<AdrDoneBits<<AllocatedEntries;
145 SC_METHOD(prc_ShiftStoreQ);
146  sensitive<<clk.pos();
147  sensitive<<StoreHead<<StoreTail<<StoreAdrDone<<StoreDataDone;
148 SC_METHOD(prc_PriorityFunction_proc);
149  sensitive<<clk.pos();
150  sensitive<<LoadRequests<<Head;
151 SC_METHOD(prc_PendingReg_proc);
152  sensitive<<clk.pos();
153 SC_METHOD(id_1198);
154  sensitive<<clk.pos();
155 SC_METHOD(id_1199);
156  sensitive<<clk.pos();
157 SC_METHOD(id_1200);
158  sensitive<<clk.pos();
159 SC_METHOD(id_1201);
160  sensitive<<clk.pos();
161 SC_METHOD(id_1202);
162  sensitive<<clk.pos();
163 SC_METHOD(prc_ShiftedDataQ_proc0);
164  sensitive<<clk.pos();
165 SC_METHOD(prc_ShiftLoadTails0_proc);
166  sensitive<<clk.pos();
167  sensitive<<StoreHead;
168 SC_METHOD(prc_EntriesToCheck0_proc);
169  sensitive<<clk.pos();
```

```
170 sensitive<<LoadTailsBits0<<ValidEntries<<CheckBits;
171 SC_METHOD(prc_Comparators0);
172 sensitive<<clk.pos();
173 sensitive<<EntriesToCheck0<<ShiftedAdrDone;
174 SC_METHOD(prc_FindLastConflict0);
175 sensitive<<clk.pos();
176 sensitive<<Conflict0<<ShiftedDataDone;
177 SC_METHOD(prc_Empties0_proc);
178 sensitive<<clk.pos();
179 sensitive<<ShiftedAdrDone<<EntriesToCheck0;
180 SC_METHOD(prc_Decision0);
181 sensitive<<clk.pos();
182 SC_METHOD(id_1399);
183 sensitive<<clk.pos();
184 SC_METHOD(prc_ShiftLoadTails1_proc);
185 sensitive<<clk.pos();
186 sensitive<<StoreHead;
187 SC_METHOD(prc_EntriesToCheck1_proc);
188 sensitive<<clk.pos();
189 sensitive<<LoadTailsBits1<<ValidEntries<<CheckBits;
190 SC_METHOD(prc_Comparators1);
191 sensitive<<clk.pos();
192 sensitive<<EntriesToCheck1<<ShiftedAdrDone;
193 SC_METHOD(prc_FindLastConflict1);
194 sensitive<<clk.pos();
195 sensitive<<Conflict1<<ShiftedDataDone;
196 SC_METHOD(prc_Empties1_proc);
197 sensitive<<clk.pos();
198 sensitive<<ShiftedAdrDone<<EntriesToCheck1;
199 SC_METHOD(prc_Decision1);
200 sensitive<<clk.pos();
201 SC_METHOD(id_1515);
202 sensitive<<clk.pos();
203 SC_METHOD(prc_ShiftLoadTails2_proc);
204 sensitive<<clk.pos();
205 sensitive<<StoreHead;
206 SC_METHOD(prc_EntriesToCheck2_proc);
207 sensitive<<clk.pos();
208 sensitive<<LoadTailsBits2<<ValidEntries<<CheckBits;
209 SC_METHOD(prc_Comparators2);
210 sensitive<<clk.pos();
211 sensitive<<EntriesToCheck2<<ShiftedAdrDone;
212 SC_METHOD(prc_FindLastConflict2);
213 sensitive<<clk.pos();
214 sensitive<<Conflict2<<ShiftedDataDone;
215 SC_METHOD(prc_Empties2_proc);
216 sensitive<<clk.pos();
217 sensitive<<ShiftedAdrDone<<EntriesToCheck2;
218 SC_METHOD(prc_Decision2);
219 sensitive<<clk.pos();
220 SC_METHOD(id_1631);
221 sensitive<<clk.pos();
222 SC_METHOD(prc_ShiftLoadTails3_proc);
223 sensitive<<clk.pos();
224 sensitive<<StoreHead;
225 SC_METHOD(prc_EntriesToCheck3_proc);
226 sensitive<<clk.pos();
227 sensitive<<LoadTailsBits3<<ValidEntries<<CheckBits;
228 SC_METHOD(prc_Comparators3);
229 sensitive<<clk.pos();
230 sensitive<<EntriesToCheck3<<ShiftedAdrDone;
231 SC_METHOD(prc_FindLastConflict3);
```

```
232 sensitive<<clk.pos();
233 sensitive<<Conflict3<<ShiftedDataDone;
234 SC_METHOD(prc_Empties3_proc);
235 sensitive<<clk.pos();
236 sensitive<<ShiftedAdrDone<<EntriesToCheck3;
237 SC_METHOD(prc_Decision3);
238 sensitive<<clk.pos();
239 SC_METHOD(id_1746);
240 sensitive<<clk.pos();
241 SC_METHOD(prc_ChooseMemRequest_proc);
242 sensitive<<clk.pos();
243 sensitive<<PriorityOut;
244 SC_METHOD(prc_DatadoneBits_proc);
245 sensitive<<clk.pos();
246 SC_METHOD(prc_DataQUpdate_proc0);
247 sensitive<<clk.pos();
248 SC_METHOD(id_1829);
249 sensitive<<clk.pos();
250 SC_METHOD(prc_DataQUpdate_proc1);
251 sensitive<<clk.pos();
252 SC_METHOD(id_1847);
253 sensitive<<clk.pos();
254 SC_METHOD(prc_DataQUpdate_proc2);
255 sensitive<<clk.pos();
256 SC_METHOD(id_1865);
257 sensitive<<clk.pos();
258 SC_METHOD(prc_DataQUpdate_proc3);
259 sensitive<<clk.pos();
260 SC_METHOD(id_1883);
261 sensitive<<clk.pos();
262 SC_METHOD(prc_SearchPortQPort0_proc);
263 sensitive<<clk.pos();
264 SC_METHOD(prc_InputPriorityPort0_proc);
265 sensitive<<clk.pos();
266 sensitive<<Head<<EntriesPort0<<AdrDoneBits;
267 SC_METHOD(prc_OutputPriorityPort0_proc);
268 sensitive<<clk.pos();
269 sensitive<<Head<<EntriesPort0<<Port0_Ready;
270 SC_METHOD(prc_SendToPort0_proc);
271 sensitive<<clk.pos();
272 SC_METHOD(prc_UpdateAddressBits);
273 sensitive<<clk.pos();
274 SC_METHOD(prc_UpdateAllDoneBits_proc);
275 sensitive<<clk.pos();
276 }
277 };
```

Listing 12: storeaddressport in SystemC

```
1  #include "loadqueue.h"
2  void load_queue :: id_1044(){
3  if(rst == 1)
4   Zeroes = "0000";
5
6  }
7  void load_queue :: id_1045(){
8  //#pragma HLS DATAFLOW
9           sc_uint<FIFO_DEPTH> temp;
10          if(rst == 1){
11          temp[0] = 1;
12          temp[1] = 1;
```

```
13          temp[2] = 1;
14          temp[3] = 1;
15          Ones = temp;
16          }
17  }
18  void load_queue :: prc_TailUpdate_proc(){
19          sc_uint<FIFO_DEPTH> tail;
20          sc_uint<FIFO_DEPTH> bbloadsize = BBLoadSize.read();
21          sc_uint<FIFO_DEPTH> temp;
22      if(rst == 1)
23          Tail = 0;
24       else if(BBStart == 1)
25         temp = (tail + bbloadsize) % FIFO_DEPTH;
26      Tail = temp ;
27  }
28
29
30  void load_queue ::prc_InitializeValues_proc(){
31  //#pragma HLS DATAFLOW
32    int i;
33    sc_uint<FIFO_DEPTH> w = InitBits.read();
34    sc_uint<FIFO_DEPTH> bbloadsize = BBLoadSize.read();
35    sc_uint<FIFO_DEPTH> tail = Tail.read();
36    if(rst == 1)
37    InitBits = "0000";
38    else if(BBStart == 1)
39      for(i=0;i<FIFO_DEPTH;i++)
40        if(i < bbloadsize)
41            w[(tail + i) % FIFO_DEPTH] = 1;
42    InitBits = w;
43  }
44  void load_queue ::prc_TailQUpdate_proc(){
45  //#pragma HLS DATAFLOW
46      int i;
47      sc_uint<FIFO_DEPTH> bbloadsize = BBLoadSize.read();
48      sc_uint<FIFO_DEPTH> tail = Tail.read();
49      for(i=0;i<FIFO_DEPTH;i++){
50        if(i < bbloadsize){
51            sc_uint<2> bbloadoffsets = BBLoadOffsets[i].read();
52            sc_uint<2> r = TailQ[(tail + i) % FIFO_DEPTH].read();
53            TailQ[(tail + i) % FIFO_DEPTH] = bbloadoffsets;
54        }
55      }
56  }
57  void load_queue ::prc_PortQUpdate_proc(){
58  //#pragma HLS DATAFLOW
59    int i;
60      for(i=0;i<FIFO_DEPTH;i++){
61  //#pragma HLS PIPELINE II=1
62        if(i < BBLoadSize.read()){
63           sc_uint<1> r = PortQ[(Tail.read() + i) % FIFO_DEPTH].read();
64           r =  BBLoadPorts[i].read();
65           PortQ[(Tail.read() + i) % FIFO_DEPTH] = r;
66        }
67      }
68  }
69  void load_queue ::prc_Empty_proc(){
70  //#pragma HLS DATAFLOW
71          sc_uint<FIFO_DEPTH> r = Ones.read();
72          sc_uint<FIFO_DEPTH> h = EmptyEntries.read();
73    if( h == r )
74      LoadEmpty = 1;
```

```
75    else
76      LoadEmpty = 0;
77  }
78  void load_queue ::prc_EmptyEntries_proc(){
79  //#pragma HLS DATAFLOW
80              int i;
81              sc_uint<FIFO_DEPTH> a = AllocatedEntries.read();
82              sc_uint<FIFO_DEPTH> k = AllDoneBits.read();
83              sc_uint<FIFO_DEPTH> h;
84    for(i=0;i<FIFO_DEPTH;i++){
85  //#pragma HLS PIPELINE II=1
86      if((k[i] == 1 )|(a[i] == 0))
87        h[i] = 1;
88      else
89        h[i] = 0;
90      EmptyEntries = h;
91    }
92  }
93  void load_queue ::prc_CheckBits_proc(){
94  //#pragma HLS DATAFLOW
95            int j;
96      int current_tail;
97      sc_uint<FIFO_DEPTH> r;
98      sc_uint<FIFO_DEPTH> h = LoadChecks.read();
99      sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
100     sc_uint<FIFO_DEPTH> previousstorehead ;
101     if(rst == 1)
102         CheckBits = "0000";
103     else{
104       PreviousStoreHead = storehead;
105     }
106     r = CheckBits.read();
107       for(j=0;j<FIFO_DEPTH;j++){
108         if((j < BBLoadSize.read()) & (BBStart == 1))
109           r[(Tail.read() + j) % FIFO_DEPTH] = h[j];
110         else
111           current_tail = TailQ[j].read().to_int();
112           if((previousstorehead = (((current_tail-1) % FIFO_DEPTH) &
                storehead)) > current_tail)
113            r[j] = 0;
114           else if((previousstorehead = (((current_tail-1) % FIFO_DEPTH)
                & storehead)) < PreviousStoreHead.read())
115             r[j] = 0;
116    CheckBits = r;
117       }
118 }
119 void load_queue ::prc_AllocatedBits_proc(){
120 //#pragma HLS DATAFLOW
121  int j;
122  sc_uint<FIFO_DEPTH> w = InitBits.read();
123  sc_uint<FIFO_DEPTH> a;
124     if(rst == 1)
125         AllocatedEntries = "0000";
126       else
127       for(j=0;j<FIFO_DEPTH;j++){
128 //#pragma HLS PIPELINE II=1
129         if(w[j] == 1)
130           a[j] = 1;
131       AllocatedEntries = a;
132       }
133 }
134 void load_queue ::id_1198(){
```

```
135 //#pragma HLS DATAFLOW
136                     LoadAdrDoneOut.write(AdrDoneBits.read());
137 }
138 void load_queue ::id_1199(){
139 //#pragma HLS DATAFLOW
140                     LoadDataDoneOut.write(DataDoneBits.read());
141 }
142 void load_queue ::id_1200(){
143 //#pragma HLS DATAFLOW
144                     LoadAddrQOut[0].write(AddrQ[0].read());
145                     LoadAddrQOut[1].write(AddrQ[1].read());
146                     LoadAddrQOut[2].write(AddrQ[2].read());
147                     LoadAddrQOut[3].write(AddrQ[3].read());
148 }
149 void load_queue ::id_1201(){
150 //#pragma HLS DATAFLOW
151                     LoadTailOut.write(Tail.read());
152 }
153 void load_queue ::id_1202(){
154 //#pragma HLS DATAFLOW
155                     LoadHeadOut.write(Head.read());
156 }
157 void load_queue ::prc_ShiftStoreQ(){
158 //#pragma HLS DATAFLOW
159   int store_upper_bound;
160   int i;
161   sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
162   sc_uint<FIFO_DEPTH> storetail = StoreTail.read();
163   sc_uint<FIFO_DEPTH> w;
164   sc_uint<FIFO_DEPTH> q;
165   sc_uint<FIFO_DEPTH> c = StoreAdrDone.read();
166   sc_uint<FIFO_DEPTH> d = StoreDataDone.read();
167   sc_uint<FIFO_DEPTH> f;
168   sc_biguint<DATA_SIZE> shiftedstoredataq[4] ;
169   sc_biguint<ADDRESS_SIZE> shiftedstoreaddrq[4];
170   if(rst ==1)
171   ValidEntries = 0;
172   else if(storetail == storehead)
173     store_upper_bound =  storetail + FIFO_DEPTH;
174     else
175       store_upper_bound = storetail;
176   for(i=0;i<FIFO_DEPTH;i++){
177 //#pragma HLS PIPELINE II=1
178         sc_biguint<ADDRESS_SIZE> h = StoreAddressQueue[(i + storehead) %
                FIFO_DEPTH].read();
179     sc_biguint<DATA_SIZE> r = StoreDataQueue[(i + storehead) %
          FIFO_DEPTH].read();
180     shiftedstoredataq[i] = r;
181         shiftedstoreaddrq[i] = h;
182     ShiftedStoreDataQ[i] = shiftedstoredataq[i];
183     ShiftedStoreAddrQ[i] = shiftedstoreaddrq[i];
184     w[i] = c[(i + storehead) % FIFO_DEPTH];
185     q[i] = d[(i + storehead) % FIFO_DEPTH];
186     ShiftedAdrDone = w;
187     ShiftedDataDone = q;
188     if((i + storehead) < store_upper_bound)
189       f[i] = 1;
190     else
191       f[i] = 0;
192     ValidEntries = f;
193   }
194 }
```

```
195 void load_queue ::prc_PriorityFunction_proc(){
196 //#pragma HLS DATAFLOW
197   bool prio_req;
198   int i;
199   int j;
200   int head = Head.read();
201   sc_uint<FIFO_DEPTH> r = LoadRequests.read();
202   sc_uint<FIFO_DEPTH> h;
203   if(rst ==1)
204   PriorityOut = "0000";
205   else{
206   h[head] = r[head];
207   PriorityOut = h;
208   }
209   for(i=1;i<FIFO_DEPTH-1;i++){
210 //#pragma HLS PIPELINE II=1
211     prio_req = 0;
212     for(j=1;j<i-1;j++){
213 //#pragma HLS PIPELINE II=1
214       prio_req = r[(head +j-1) % FIFO_DEPTH].to_bool()|prio_req;
215     }
216     h[(head + i) % FIFO_DEPTH] = r[(head + i) % FIFO_DEPTH].to_bool() &
          (~prio_req);
217     PriorityOut = h;
218   }
219 }
220 void load_queue ::prc_PendingReg_proc(){
221 //#pragma HLS DATAFLOW
222   int i;
223   sc_uint<FIFO_DEPTH> r = PriorityOut.read();
224   sc_uint<FIFO_DEPTH> h = PendingBits.read();
225     if(rst == 1)
226       PendingBits = "0000";
227       else
228       for(i=0;i<FIFO_DEPTH;i++){
229 //#pragma HLS PIPELINE II=1
230         if(r[i] == 1)
231           h[i] = 1;
232           else
233           h[i] = 0;
234         PendingBits = h;
235       }
236 }
237 void load_queue ::prc_ShiftedDataQ_proc0(){
238 //#pragma HLS DATAFLOW
239           ShiftedStoreDataQ2[0].write( ShiftedStoreDataQ[0].read());
240           ShiftedStoreDataQ2[1].write( ShiftedStoreDataQ[1].read());
241           ShiftedStoreDataQ2[2].write( ShiftedStoreDataQ[2].read());
242           ShiftedStoreDataQ2[3].write( ShiftedStoreDataQ[3].read());
243 }
244
245 void load_queue ::prc_ShiftLoadTails0_proc(){
246 //#pragma HLS DATAFLOW
247   int current_tail;
248   int store_upper_bound;
249   int i;
250   sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
251   sc_uint<2> tailq = TailQ[0].read().to_int();
252   sc_uint<FIFO_DEPTH> r ;
253   if(rst ==1)
254           LoadTailsBits0 = 0;
255   else
```

```
256   current_tail = tailq;
257   if(((current_tail - 1) % FIFO_DEPTH) < storehead)
258     store_upper_bound = current_tail + FIFO_DEPTH;
259     else if((((current_tail -1) % FIFO_DEPTH) == storehead) & StoreEmpty)
260       store_upper_bound = current_tail;
261       else
262        store_upper_bound = current_tail;
263   for(i=0;i<FIFO_DEPTH;i++){
264 //#pragma HLS PIPELINE II=1
265     if((i + storehead) < store_upper_bound)
266       r[i] = 1;
267       else
268       r[i] = 0;
269     LoadTailsBits0 = r;
270   }
271 }
272 void load_queue ::prc_EntriesToCheck0_proc(){
273 //#pragma HLS DATAFLOW
274   int i;
275   sc_uint<FIFO_DEPTH> r;
276   sc_uint<FIFO_DEPTH> h = LoadTailsBits0.read();
277   sc_uint<FIFO_DEPTH> w = ValidEntries.read();
278   sc_uint<FIFO_DEPTH> a = CheckBits.read();
279   if(rst ==1)
280           EntriesToCheck0 = 0;
281   else
282    for(i=0;i<FIFO_DEPTH;i++){
283 //#pragma HLS PIPELINE II=1
284           r[0] = h[i] & w[i] & a[0];
285           EntriesToCheck0 = r;
286   }
287 }
288 void load_queue ::prc_Comparators0(){
289 //#pragma HLS DATAFLOW
290   int i;
291   sc_uint<FIFO_DEPTH> w = EntriesToCheck0.read();
292   sc_uint<FIFO_DEPTH> a = ShiftedAdrDone.read();
293   sc_uint<FIFO_DEPTH> s;
294   sc_biguint<ADDRESS_SIZE> addrq[4];
295   addrq[0]= AddrQ[0].read();
296   if(rst == 1)
297           Conflict0 = 0;
298   else
299    for(i=0;i<FIFO_DEPTH;i++){
300 //#pragma HLS PIPELINE II=1
301  sc_biguint<ADDRESS_SIZE> h = ShiftedStoreAddrQ[i].read();
302     if((addrq[0] == h) & (w[i] == 1) & (a[i] == 1))
303       s[i] = 1;
304       else
305       s[i] = 0;
306     Conflict0 = s;
307   }
308 }
309 void load_queue ::prc_FindLastConflict0(){
310 //#pragma HLS DATAFLOW
311   bool found;
312   int i;
313   sc_uint<FIFO_DEPTH> s = Conflict0.read();
314   sc_uint<FIFO_DEPTH> r;
315   sc_uint<FIFO_DEPTH> h = ShiftedDataDone.read();
316   sc_uint<FIFO_DEPTH> c;
317   if(rst == 1){
```

```
318     found = 0;
319     LastConflict0 = 0;
320     CanBypass0 = "1111";
321     }
322     else
323      for(i=0;i<FIFO_DEPTH;i++){
324       if((s[FIFO_DEPTH-1-i] == 1) & (found == 0)){
325         found = 1;
326         r[FIFO_DEPTH -1-i] = 1;
327         if(h[FIFO_DEPTH-1-i] == 1)
328           c[FIFO_DEPTH-1-i] = 1;
329          else
330           c[FIFO_DEPTH-1-i] = 0;
331       }
332       else
333         r[FIFO_DEPTH-1-i] = 0;
334     }
335         LastConflict0 = r;
336         CanBypass0 = c;
337
338 }
339 void load_queue :: prc_Empties0_proc(){
340     int i;
341     sc_uint<FIFO_DEPTH> s = ShiftedAdrDone.read();
342     sc_uint<FIFO_DEPTH> d = EntriesToCheck0.read();
343     sc_uint<FIFO_DEPTH> empties0;
344     if(rst == 1)
345             Empties0 = 0;
346     else
347      for(i=0;i<FIFO_DEPTH;i++){
348 //#pragma HLS PIPELINE II=1
349       if((s[i] == 0) & (d[i] == 1))
350           empties0[i] = 1;
351         else
352           empties0[i] = 0;
353       Empties0 = empties0;
354     }
355 }
356 void load_queue :: prc_Decision0(){
357           sc_uint<FIFO_DEPTH> s = AdrDoneBits.read();
358           sc_uint<FIFO_DEPTH> r = DataDoneBits.read();
359           sc_uint<FIFO_DEPTH> h = LastConflict0.read();
360           sc_uint<FIFO_DEPTH> k = CanBypass0.read();
361           sc_uint<FIFO_DEPTH> p = PendingBits.read();
362           sc_uint<FIFO_DEPTH> a = PriorityOut.read();
363           sc_uint<FIFO_DEPTH> b = Empties0.read();
364           sc_uint<FIFO_DEPTH> d = BypassIndex0.read();
365           if(rst == 1){
366           BypassIndex0 = "0000";
367       BypassRequest0 = 0;
368       LoadRequest0 = 0;
369           }
370
371       bool temp = BypassRequest0;
372
373       if((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (a[0] ==
              0)){
374         if((h == k) & (h.to_int() > 0)&(b.to_int() < h.to_int()))
375           BypassRequest0 = 1;
376           else
377           BypassRequest0 = 0;
378         if((b == 0)&(h == 0))
```

```
379           LoadRequest0 = 1;
380           else
381           LoadRequest0 = 0;
382       }
383       d = k;
384       BypassIndex0 = d;
385   }
386   void load_queue ::id_1399(){
387           sc_uint<FIFO_DEPTH> r ;
388           if(rst ==1)
389                   LoadRequests0 = 0;
390           else
391           r = LoadRequests.read();
392                       r[0] = LoadRequest0;
393                       LoadRequests0 = r[0];
394
395   }
396   void load_queue ::prc_ShiftLoadTails1_proc(){
397   //#pragma HLS DATAFLOW
398     int i;
399     int current_tail;
400     int store_upper_bound;
401     sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
402     sc_uint<2> tailq = TailQ[1].read();
403     sc_uint<FIFO_DEPTH> r;
404     if(rst ==1)
405             LoadTailsBits1 = 0;
406      else
407     current_tail = tailq.to_uint();
408     if(((current_tail - 1) % FIFO_DEPTH) < storehead)
409       store_upper_bound = current_tail + FIFO_DEPTH;
410       else if((((current_tail-1) % FIFO_DEPTH) == storehead) & StoreEmpty)
411         store_upper_bound = current_tail;
412         else
413         store_upper_bound = current_tail;
414     for(i=0;i<FIFO_DEPTH;i++){
415   //#pragma HLS PIPELINE II=1
416       if((i + storehead) < store_upper_bound)
417         r[i] = 1;
418         else
419         r[i] = 0;
420       LoadTailsBits1 = r;
421     }
422   }
423   void load_queue ::prc_EntriesToCheck1_proc(){
424   //#pragma HLS DATAFLOW
425     int i;
426     sc_uint<FIFO_DEPTH> r;
427     sc_uint<FIFO_DEPTH> h = LoadTailsBits1.read();
428     sc_uint<FIFO_DEPTH> w = ValidEntries.read();
429     sc_uint<FIFO_DEPTH> a = CheckBits.read();
430     if(rst ==1)
431             EntriesToCheck1 = 0;
432       else
433        for(i=0;i<FIFO_DEPTH;i++)
434         r[0] = h[i] & w[i] & a[0];
435         EntriesToCheck1 = r;
436
437   }
438   void load_queue ::prc_Comparators1(){
439   //#pragma HLS DATAFLOW
440     int i;
```

```
441   sc_uint<FIFO_DEPTH> w = EntriesToCheck1.read();
442   sc_uint<FIFO_DEPTH> a = ShiftedAdrDone.read();
443   sc_uint<FIFO_DEPTH> s;
444   sc_biguint<ADDRESS_SIZE> addrq1 = AddrQ[1].read();
445   if(rst == 1)
446           Conflict1 = 0;
447   else
448    for(i=0;i<FIFO_DEPTH;i++){
449           sc_biguint<FIFO_DEPTH> h = ShiftedStoreAddrQ[i].read();
450     if((addrq1 == h) & (w[i] == 1) & (a[i] == 1))
451        s[i] = 1;
452     else
453        s[i] = 0;
454     Conflict1 = s;
455   }
456 }
457 void load_queue ::prc_FindLastConflict1(){
458 //#pragma HLS DATAFLOW
459         bool found;
460           int i;
461           sc_uint<FIFO_DEPTH> s = Conflict1.read();
462           sc_uint<FIFO_DEPTH> r;
463           sc_uint<FIFO_DEPTH> h = ShiftedDataDone.read();
464           sc_uint<FIFO_DEPTH> c;
465           if(rst == 1){
466             found = 0;
467             LastConflict1 = 0;
468             CanBypass1 = "1111";
469             }
470             else
471             for(i=0;i<FIFO_DEPTH;i++){
472               if((s[FIFO_DEPTH-1-i] == 1) & (found == 0)){
473                 found = 1;
474                 r[FIFO_DEPTH -1-i] = 1;
475                 if(h[FIFO_DEPTH-1-i] == 1)
476                   c[FIFO_DEPTH-1-i] = 1;
477                 else
478                   c[FIFO_DEPTH-1-i] = 0;
479               }
480               else
481                 r[FIFO_DEPTH-1-i] = 0;
482               CanBypass1 = c;
483               LastConflict1 = r;
484   }
485 }
486 void load_queue :: prc_Empties1_proc(){
487 //#pragma HLS DATAFLOW
488           int i;
489           sc_uint<FIFO_DEPTH> s = ShiftedAdrDone.read();
490           sc_uint<FIFO_DEPTH> d = EntriesToCheck1.read();
491           sc_uint<FIFO_DEPTH> empties1 = Empties1.read();
492          if(rst == 1)
493                   Empties1 = 0;
494           else
495           for(i=0;i<FIFO_DEPTH;i++){
496 //#pragma HLS PIPELINE II=1
497             if((s[i] ==0) & (d[i] == 1))
498                 empties1[i] = 1;
499               else
500                 empties1[i] = 0;
501             Empties1 = empties1;
502   }
```

```
503 }
504 void load_queue :: prc_Decision1(){
505 //#pragma HLS DATAFLOW
506         sc_uint<FIFO_DEPTH> s = AdrDoneBits.read();
507         sc_uint<FIFO_DEPTH> r = DataDoneBits.read();
508         sc_uint<FIFO_DEPTH> h = LastConflict1.read();
509         sc_uint<FIFO_DEPTH> k = CanBypass1.read();
510         sc_uint<FIFO_DEPTH> p = PendingBits.read();
511         sc_uint<FIFO_DEPTH> a = PriorityOut.read();
512         sc_uint<FIFO_DEPTH> b = Empties1.read();
513         sc_uint<FIFO_DEPTH> d ;
514         if(rst == 1){
515                 BypassIndex1 = "0000";
516             BypassRequest1 = 0;
517             LoadRequest1 = 0;
518                 }
519             bool temp = BypassRequest1;
520             if((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (
                    a[0] == 0)){
521               if((h == k) & (h > 0) & (b < h))
522                 BypassRequest1 = 1;
523                 else
524                 BypassRequest1 = 0;
525               if((b == 0)&(h == 0))
526                 LoadRequest1 = 1;
527                 else
528                 LoadRequest1 = 0;
529             }
530             d = k;
531             BypassIndex1 = d;
532
533 }
534 void load_queue ::id_1515(){
535 //#pragma HLS DATAFLOW
536         sc_uint<FIFO_DEPTH> r ;
537 if(rst ==1)
538                             LoadRequests1 = 0;
539 else
540         r = LoadRequests.read();
541                     r[1] = LoadRequest1;
542                     LoadRequests1 = r[1];
543 }
544 void load_queue ::prc_ShiftLoadTails2_proc(){
545 //#pragma HLS DATAFLOW
546         int i;
547         int current_tail;
548         int store_upper_bound;
549         sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
550         sc_uint<2> tailq = TailQ[2].read();
551         sc_uint<FIFO_DEPTH> r;
552          if(rst ==1)
553                   LoadTailsBits2 = 0;
554           else
555           current_tail = tailq.to_uint();
556           if(((current_tail - 1) % FIFO_DEPTH) < storehead)
557             store_upper_bound = current_tail + FIFO_DEPTH;
558             else if((((current_tail-1) % FIFO_DEPTH) == storehead) &
                    StoreEmpty)
559               store_upper_bound = current_tail;
560               else
561               store_upper_bound = current_tail;
562           for(i=0;i<FIFO_DEPTH;i++){
```

```
563 //#pragma HLS PIPELINE II=1
564               if((i + storehead) < store_upper_bound)
565                 r[i] = 1;
566                 else
567                 r[i] = 0;
568               LoadTailsBits2 = r;
569   }
570 }
571 void load_queue ::prc_EntriesToCheck2_proc(){
572   int i;
573   sc_uint<FIFO_DEPTH> r;
574   sc_uint<FIFO_DEPTH> h = LoadTailsBits2.read();
575   sc_uint<FIFO_DEPTH> w = ValidEntries.read();
576   sc_uint<FIFO_DEPTH> a = CheckBits.read();
577   if(rst ==1)
578             EntriesToCheck2 = 0;
579     else
580      for(i=0;i<FIFO_DEPTH;i++){
581 //#pragma HLS PIPELINE II=1
582             r[0] = h[i] & w[i] & a[0];
583             EntriesToCheck2 = r;
584   }
585 }
586 void load_queue ::prc_Comparators2(){
587   int i;
588   sc_uint<FIFO_DEPTH> w = EntriesToCheck2.read();
589   sc_uint<FIFO_DEPTH> a = ShiftedAdrDone.read();
590   sc_uint<FIFO_DEPTH> s = Conflict2.read();
591   sc_biguint<ADDRESS_SIZE> addrq2 = AddrQ[2].read();
592   if(rst == 1)
593             Conflict2 = 0;
594    else
595     for(i=0;i<FIFO_DEPTH;i++){
596              sc_biguint<FIFO_DEPTH> h = ShiftedStoreAddrQ[i].read();
597      if((addrq2 == h) & (w[i] == 1) & (a[i] == 1))
598        s[i] = 1;
599        else
600        s[i] = 0;
601      Conflict2 = s;
602   }
603 }
604 void load_queue ::prc_FindLastConflict2(){
605 //#pragma HLS DATAFLOW
606         bool found;
607           int i;
608           sc_uint<FIFO_DEPTH> s = Conflict2.read();
609           sc_uint<FIFO_DEPTH> r;
610           sc_uint<FIFO_DEPTH> h = ShiftedDataDone.read();
611           sc_uint<FIFO_DEPTH> c;
612  if(rst == 1){
613             found = 0;
614             LastConflict2 = 0;
615             CanBypass2 = "1111";
616  }
617          else
618            for(i=0;i<FIFO_DEPTH;i++){
619 //#pragma HLS PIPELINE II=1
620              if((s[FIFO_DEPTH-1-i] == 1) & (found == 0)){
621                found = 1;
622                r[FIFO_DEPTH -1-i] = 1;
623                if(h[FIFO_DEPTH-1-i] == 1)
624                  c[FIFO_DEPTH-1-i] = 1;
```

```
625                 else
626                   c[FIFO_DEPTH-1-i] = 0;
627               }
628               else
629                 r[FIFO_DEPTH-1-i] = 0;
630               LastConflict2 = r;
631               CanBypass2 = c;
632   }
633 }
634 void load_queue :: prc_Empties2_proc(){
635 //#pragma HLS DATAFLOW
636           int i;
637           sc_uint<FIFO_DEPTH> s = ShiftedAdrDone.read();
638           sc_uint<FIFO_DEPTH> d = EntriesToCheck2.read();
639           sc_uint<FIFO_DEPTH> empties0 ;
640           if(rst == 1)
641                   Empties2 = 0;
642            else
643           for(i=0;i<FIFO_DEPTH;i++){
644             if((s[i] == 0) & (d[i] == 1))
645                 empties0[i] = 1;
646               else
647                 empties0[i] = 0;
648             Empties2 = empties0;
649   }
650 }
651 void load_queue :: prc_Decision2(){
652 //#pragma HLS DATAFLOW
653         sc_uint<FIFO_DEPTH> s = AdrDoneBits.read();
654         sc_uint<FIFO_DEPTH> r = DataDoneBits.read();
655         sc_uint<FIFO_DEPTH> h = LastConflict2.read();
656         sc_uint<FIFO_DEPTH> k = CanBypass2.read();
657         sc_uint<FIFO_DEPTH> p = PendingBits.read();
658         sc_uint<FIFO_DEPTH> a = PriorityOut.read();
659         sc_uint<FIFO_DEPTH> b = Empties2.read();
660         sc_uint<FIFO_DEPTH> d;
661         if(rst == 1){
662                 BypassIndex2 = "0000";
663             BypassRequest2 = 0;
664             LoadRequest2 = 0;
665                 }
666             bool temp = BypassRequest2;
667             if((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (
                    a[0] == 0)){
668               if((h == k) & (h > 0)&(b < h))
669                 BypassRequest2 = 1;
670                 else
671                 BypassRequest2 = 0;
672               if((b == 0)&(h == 0))
673                 LoadRequest2 = 1;
674                 else
675                 LoadRequest2 = 0;
676             }
677             d = k;
678             BypassIndex2 = d;
679 }
680 void load_queue ::id_1631(){
681 //#pragma HLS DATAFLOW
682         sc_uint<FIFO_DEPTH> r;
683 if(rst ==1)
684                         LoadRequests2 = 0;
685  else
```

```
686         r = LoadRequests.read();
687                     r[2] = LoadRequest2;
688                     LoadRequests2 = r[2];
689 }
690 void load_queue ::prc_ShiftLoadTails3_proc(){
691 //#pragma HLS DATAFLOW
692           int i;
693           int current_tail;
694           int store_upper_bound;
695           sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
696           sc_uint<2> tailq = TailQ[3].read();
697           sc_uint<FIFO_DEPTH> r = LoadTailsBits3.read();
698           if(rst ==1)
699                   LoadTailsBits3 = 0;
700            else
701           current_tail = tailq.to_uint();
702           if(((current_tail - 1) % FIFO_DEPTH) < storehead)
703             store_upper_bound = current_tail + FIFO_DEPTH;
704             else if((((current_tail-1) % FIFO_DEPTH) == storehead) &
                    StoreEmpty )
705               store_upper_bound = current_tail;
706               else
707               store_upper_bound = current_tail;
708           for(i=0;i<FIFO_DEPTH;i++){
709 //#pragma HLS PIPELINE II=1
710             if((i + storehead) < store_upper_bound)
711               r[i] = 1;
712               else
713                r[i] = 0;
714             LoadTailsBits3 = r;
715   }
716 }
717 void load_queue ::prc_EntriesToCheck3_proc(){
718 //#pragma HLS DATAFLOW
719   int i;
720   sc_uint<FIFO_DEPTH> r;
721   sc_uint<FIFO_DEPTH> h = LoadTailsBits3.read();
722   sc_uint<FIFO_DEPTH> w = ValidEntries.read();
723   sc_uint<FIFO_DEPTH> a = CheckBits.read();
724   if(rst ==1)
725           EntriesToCheck3 = 0;
726     else
727      for(i=0;i<FIFO_DEPTH;i++)
728           r[0] = h[i] & w[i] & a[0];
729           EntriesToCheck3 = r;
730
731 }
732 void load_queue ::prc_Comparators3(){
733 //#pragma HLS DATAFLOW
734   int i;
735   sc_uint<FIFO_DEPTH> w = EntriesToCheck3.read();
736   sc_uint<FIFO_DEPTH> a = ShiftedAdrDone.read();
737   sc_uint<FIFO_DEPTH> s;
738   sc_biguint<ADDRESS_SIZE> addrq3 = AddrQ[3].read();
739   if(rst == 1)
740           Conflict3 = 0;
741    else
742     for(i=0;i<FIFO_DEPTH;i++){
743 //#pragma HLS PIPELINE II=1
744            sc_biguint<FIFO_DEPTH> h = ShiftedStoreAddrQ[i].read();
745      if((addrq3 == h) & (w[i] == 1) & (a[i] == 1))
746        s[i] = 1;
```

```
747         else
748          s[i] = 0;
749       Conflict3 = s;
750
751   }
752 }
753 void load_queue ::prc_FindLastConflict3(){
754 //#pragma HLS DATAFLOW
755             bool found;
756             int i;
757             sc_uint<FIFO_DEPTH> s = Conflict3.read();
758             sc_uint<FIFO_DEPTH> r;
759             sc_uint<FIFO_DEPTH> h = ShiftedDataDone.read();
760             sc_uint<FIFO_DEPTH> c;
761             if(rst == 1){
762               found = 0;
763               LastConflict3 = 0;
764               CanBypass3 = "1111";
765               }
766             else
767              for(i=0;i<FIFO_DEPTH;i++){
768 //#pragma HLS PIPELINE II=1
769                if((s[FIFO_DEPTH-1-i] == 1) & (found == 0)){
770                  found = 1;
771                  r[FIFO_DEPTH -1-i] = 1;
772                  if(h[FIFO_DEPTH-1-i] == 1)
773                    c[FIFO_DEPTH-1-i] = 1;
774                  else
775                    c[FIFO_DEPTH-1-i] = 0;
776                }
777                else
778                  r[FIFO_DEPTH-1-i] = 0;
779                CanBypass3 = c;
780                LastConflict3 = r;
781   }
782 }
783 void load_queue :: prc_Empties3_proc(){
784 //#pragma HLS DATAFLOW
785             int i;
786             sc_uint<FIFO_DEPTH> s = ShiftedAdrDone.read();
787             sc_uint<FIFO_DEPTH> d = EntriesToCheck3.read();
788             sc_uint<FIFO_DEPTH> empties0 = Empties3.read();
789             if(rst == 1)
790                     Empties3 = 0;
791              else
792             for(i=0;i<FIFO_DEPTH;i++){
793 //#pragma HLS PIPELINE II=1
794               if((s[i] == 0) & (d[i] == 1))
795                    empties0[i] = 1;
796                 else
797                   empties0[i] = 0;
798               Empties3 = empties0;
799   }
800 }
801 void load_queue :: prc_Decision3(){
802               sc_uint<FIFO_DEPTH> s = AdrDoneBits.read();
803                    sc_uint<FIFO_DEPTH> r = DataDoneBits.read();
804                    sc_uint<FIFO_DEPTH> h = LastConflict3.read();
805                    sc_uint<FIFO_DEPTH> k = CanBypass3.read();
806                    sc_uint<FIFO_DEPTH> p = PendingBits.read();
807                    sc_uint<FIFO_DEPTH> a = PriorityOut.read();
808                    sc_uint<FIFO_DEPTH> b = Empties3.read();
```

```
809                     sc_uint<FIFO_DEPTH> d;
810                     if(rst == 1){
811                             BypassIndex3 = "0000";
812                         BypassRequest3 = 0;
813                         LoadRequest3 = 0;
814                             }
815                         bool temp = BypassRequest3;
816                         if((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp
                               == 0) & (a[0] == 0)){
817                           if((h == k) & (h > 0)&(b < h))
818                             BypassRequest3 = 1;
819                             else
820                             BypassRequest3 = 0;
821                           if((b == 0)&(h == 0))
822                             LoadRequest3 = 1;
823                             else
824                             LoadRequest3 = 0;
825                         }
826                         d = k;
827                         BypassIndex3 = d;
828
829 }
830 void load_queue ::id_1746(){
831 //#pragma HLS DATAFLOW
832         sc_uint<FIFO_DEPTH> r ;
833         if(rst ==1)
834                         LoadRequests3 = 0;
835         else
836                 r = LoadRequests.read();
837             r[3] = LoadRequest3;
838             LoadRequests3 = r[3];
839 }
840
841 void load_queue :: prc_ChooseMemRequest_proc(){
842 //#pragma HLS DATAFLOW
843         sc_uint<FIFO_DEPTH> r = PriorityOut.read();
844         sc_biguint<ADDRESS_SIZE> addrq1 = AddrQ[1].read();
845   MemoryLoadEnable = 0;
846   if(r[0] == 1){
847           MemoryLoadAddress = AddrQ[0].read();
848     MemoryLoadEnable = 1;
849   }
850    else if(r[1] == 1){
851            MemoryLoadAddress = addrq1;
852            MemoryLoadEnable = 1;
853    }
854       else if(r[2] == 1){
855           MemoryLoadAddress = AddrQ[2].read();
856           MemoryLoadEnable = 1;
857       }
858         else if(r[3] == 1){
859                 MemoryLoadAddress = AddrQ[3].read();
860                 MemoryLoadEnable = 1;
861         }
862           else
863                   MemoryLoadAddress = "0000000000";
864 }
865 void load_queue :: prc_DatadoneBits_proc(){
866 //#pragma HLS DATAFLOW
867         sc_uint<FIFO_DEPTH> r = InitBits.read();
868         sc_uint<FIFO_DEPTH> h;
869         sc_uint<FIFO_DEPTH> q = PendingBits.read();
```

```
870     if(rst == 1)
871         DataDoneBits = "0000";
872       else{
873        if(r[0] == 1)
874           h[0] = 0;
875           else if(q[0] == 1)
876             h[0] = 1;
877              else if(BypassRequest0 == 1)
878                  h[0] = 1;
879        if(r[1] == 1){
880                  h[1] = 0;
881          }
882            else if(q[1] == 1)
883                 h[1] = 1;
884              else if(BypassRequest1 == 1)
885                 h[1] = 1;
886          }
887        if(r[2] == 1){
888              h[2] = 0;
889         }
890            else if(q[2] == 1)
891                 h[2] = 1;
892              else if(BypassRequest2 == 1)
893                  h[2] = 1;
894        if(r[3] == 1){
895              h[3] = 0;
896        }
897            else if(q[3] == 1)
898                 h[3] = 1;
899             else if(BypassRequest3 == 1)
900                 h[3] = 1;
901        DataDoneBits = h;
902 }
903 void load_queue :: prc_DataQUpdate_proc0(){
904 //#pragma HLS DATAFLOW
905   int i;
906   sc_uint<FIFO_DEPTH> r = PriorityOut.read();
907   sc_uint<FIFO_DEPTH> d = BypassIndex0.read();
908   sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
909   sc_uint<FIFO_DEPTH> q = PendingBits.read();
910   sc_biguint<DATA_SIZE> data0 = Data0.read();
911 if(rst == 1)
912         Data0 = 0;
913
914 else if(BypassRequest0 == 1)
915       for(i=0;i<FIFO_DEPTH;i++){
916 //#pragma HLS PIPELINE II=1
917         if(d[i] == 1){
918                 sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
919           data0 = h;
920         }
921       }
922       else if(q[0] == 1)
923         data0 = MemoryLoadData.read();
924     Data0 = data0;
925
926 }
927 void load_queue::id_1829(){
928 //#pragma HLS DATAFLOW
929         if(rst ==1){
930                 DataQ[0] = 0;
931         }
```

```
932           else{
933            sc_biguint<DATA_SIZE> tmp = Data0.read();
934         DataQ[0] = tmp;
935           }
936 }
937 void load_queue :: prc_DataQUpdate_proc1(){
938 //#pragma HLS DATAFLOW
939   int i;
940   sc_uint<FIFO_DEPTH> r = PriorityOut.read();
941   sc_uint<FIFO_DEPTH> d = BypassIndex1.read();
942   sc_uint<FIFO_DEPTH> q = PendingBits.read();
943   sc_uint<DATA_SIZE> data1;
944   if(rst ==1)
945             Data1 = 0;
946   else   if(BypassRequest1 == 1)
947         for(i=0;i<FIFO_DEPTH;i++){
948 //#pragma HLS PIPELINE II=1
949           if(d[i] == 1){
950                   sc_uint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
951             data1 = h;
952           }
953         }
954         else if(q[1] == 1)
955           data1 = MemoryLoadData.read();
956       Data1 = data1;
957 }
958 void load_queue ::id_1847(){
959 //#pragma HLS DATAFLOW
960          sc_uint<DATA_SIZE> tmp = Data1.read();
961          if(rst ==1){
962                           DataQ[1] = 0;
963                   }
964                DataQ[1] = tmp;
965 }
966 void load_queue :: prc_DataQUpdate_proc2(){
967 //#pragma HLS DATAFLOW
968   int i;
969   sc_uint<FIFO_DEPTH> r = PriorityOut.read();
970   sc_uint<FIFO_DEPTH> d = BypassIndex2.read();
971   sc_uint<FIFO_DEPTH> q = PendingBits.read();
972   sc_uint<DATA_SIZE> data2;
973   if(rst == 1)
974     Data2 = 0;
975   else if(BypassRequest2 == 1)
976         for(i=0;i<FIFO_DEPTH;i++){
977 //#pragma HLS PIPELINE II=1
978           if(d[i] == 1){
979                   sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
980             data2 = h;
981           }
982         }
983         else if(q[2] == 1)
984           data2 = MemoryLoadData.read();
985       Data2 = data2;
986
987 }
988 void load_queue ::id_1865(){
989 //#pragma HLS DATAFLOW
990          sc_uint<DATA_SIZE> tmp = Data2.read();
991          if(rst ==1){
992                                   DataQ[2] = 0;
993                           }
```

```
994                 DataQ[2] = tmp;
995 }
996 void load_queue :: prc_DataQUpdate_proc3(){
997 //#pragma HLS DATAFLOW
998   int i;
999   sc_uint<FIFO_DEPTH> r = PriorityOut.read();
1000  sc_uint<FIFO_DEPTH> d = BypassIndex3.read();
1001  sc_uint<FIFO_DEPTH> q = PendingBits.read();
1002  sc_uint<DATA_SIZE> data3 ;
1003    if(rst == 1)
1004        Data3 =0;
1005    else if(BypassRequest3)
1006       for(i=0;i<FIFO_DEPTH;i++){
1007 //#pragma HLS PIPELINE II=1
1008         if(d[i] == 1){
1009                  sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
1010           data3 = h;
1011         }
1012       }
1013       else if(q[3] == 1)
1014         data3 = MemoryLoadData.read();
1015     Data3 = data3;
1016
1017 }
1018 void load_queue ::id_1883(){
1019 //#pragma HLS DATAFLOW
1020        sc_uint<DATA_SIZE> tmp = Data3.read();
1021        if(rst ==1){
1022                                  DataQ[3] = 0;
1023                           }
1024                 DataQ[3] = tmp;
1025 }
1026 void load_queue :: prc_SearchPortQPort0_proc(){
1027 //#pragma HLS DATAFLOW
1028        int i;
1029        sc_uint<FIFO_DEPTH> r;
1030        if(rst ==1)
1031                 EntriesPort0 = 0;
1032        else
1033        for(i=0;i < FIFO_DEPTH; i++){
1034         sc_uint<1> portq = PortQ[i].read();
1035                 if(portq == 0)
1036                          r[i] = 1;
1037                          else
1038                           r[i] = 0;
1039                 EntriesPort0 = r;
1040        }
1041 }
1042 void load_queue :: prc_InputPriorityPort0_proc(){
1043 //#pragma HLS DATAFLOW
1044   bool blocked;
1045   int i;
1046   sc_uint<FIFO_DEPTH> head = Head.read();
1047   sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
1048   sc_uint<FIFO_DEPTH> h;
1049   sc_uint<FIFO_DEPTH> b = AdrDoneBits.read();
1050   if(rst == 1){
1051   blocked = 0;
1052   InputPriorityPort0 = "0000";
1053   }
1054   else
1055   for(i=0;i<FIFO_DEPTH;i++){
```

```
1056 //#pragma HLS PIPELINE II=1
1057     if((r[(i + head) % FIFO_DEPTH] == 1) & (b[(i+head) & FIFO_DEPTH] ==
         0)){
1058       if(blocked == 0){
1059         h[(i+head) % FIFO_DEPTH] = 1;
1060         blocked =1;
1061          InputPriorityPort0 = h;
1062       }
1063     }
1064   }
1065 }
1066 void load_queue :: prc_OutputPriorityPort0_proc(){
1067 //#pragma HLS DATAFLOW
1068  bool blocked;
1069   int i;
1070   sc_uint<FIFO_DEPTH> head = Head.read();
1071   sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
1072   sc_uint<FIFO_DEPTH> h;
1073   sc_uint<FIFO_DEPTH> b = AdrDoneBits.read();
1074   if(rst == 1){
1075   blocked = 0;
1076   OutputPriorityPort0 = "0000";
1077   }
1078   for(i=0;i<FIFO_DEPTH;i++){
1079 //#pragma HLS PIPELINE II=1
1080     if((r[(i + head) % FIFO_DEPTH] == 1) & (b[(i+head) & FIFO_DEPTH] ==
         0)){
1081       if(blocked == 0){
1082         h[(i+head) % FIFO_DEPTH] = 1 ;
1083         blocked = 1;
1084          OutputPriorityPort0 = h;
1085       }
1086     }
1087   }
1088 }
1089 void load_queue :: prc_SendToPort0_proc(){
1090 //#pragma HLS DATAFLOW
1091   bool send0;
1092   int i;
1093   sc_uint<FIFO_DEPTH> h = OutputPriorityPort0.read();
1094   sc_uint<FIFO_DEPTH> datadonebits = DataDoneBits.read();
1095   sc_uint<FIFO_DEPTH> r = AllDoneBits.read();
1096   sc_uint<DATA_SIZE> dataq[4];
1097   bool temp = Port0_Ready.read();
1098     if(rst == 1)
1099       Port0_Valid = 0;
1100       else
1101        send0 = 0;
1102        for(i=0;i<FIFO_DEPTH;i++){
1103                dataq[i] = DataQ[i].read();
1104         if((h[i] == 1) & (datadonebits[i] == 1) & (r[i] == 0) & (temp ==
               1))
1105                 Port0_DataOut.write(dataq[i]);
1106             send0 = 1;
1107        }
1108       if(send0 == 1)
1109         Port0_Valid = 1;
1110         else if(Port0_Ready == 1)
1111           Port0_Valid = 1;
1112 }
1113 void load_queue :: prc_UpdateAddressBits(){
1114 //#pragma HLS DATAFLOW
```

```
1115    int i;
1116    sc_uint<FIFO_DEPTH> b;
1117    sc_uint<FIFO_DEPTH> h = InputPriorityPort0.read();
1118    sc_uint<FIFO_DEPTH> r = InitBits.read();
1119    sc_uint<ADDRESS_SIZE> port0_addrin = Port0_AddrIn.read();
1120    sc_biguint<ADDRESS_SIZE> addrq[4];
1121    if(rst == 1)
1122              AdrDoneBits = "0000";
1123         else
1124         for(i=0;i<FIFO_DEPTH;i++){
1125 //#pragma HLS PIPELINE II=1
1126           if(r[i] == 1)
1127             b[i] = 0;
1128             else if((h[i] == 1) & Port0_LoadWriteEn)
1129                     addrq[i] = port0_addrin;
1130                AddrQ[i] = addrq[i];
1131                b[i] = 0;
1132              AdrDoneBits = b;
1133      }
1134 }
1135 void load_queue :: prc_UpdateAllDoneBits_proc(){
1136 //#pragma HLS DATAFLOW
1137    int i;
1138    sc_uint<FIFO_DEPTH> r = InitBits.read();
1139    sc_uint<FIFO_DEPTH> h = OutputPriorityPort0.read();
1140    sc_uint<FIFO_DEPTH> a = AllDoneBits.read();
1141    sc_uint<FIFO_DEPTH> d = DataDoneBits.read();
1142    sc_uint<FIFO_DEPTH> sending;
1143    sc_uint<FIFO_DEPTH> head = Head.read();
1144    sc_uint<FIFO_DEPTH> tail = Tail.read();
1145    bool temp = Port0_Ready;
1146      sending = "0000";
1147      if(rst == 1){
1148       AllDoneBits = "0000";
1149       Head.write(0);
1150      }
1151        else
1152        for(i=0;i<FIFO_DEPTH;i++){
1153 //#pragma HLS PIPELINE II=1
1154           if(r[i] == 1)
1155             a[i] = 0;
1156             else if((h[i] == 1)&(d[i] == 1)&(r[i] == 0)&(temp == 1))
1157                      a[i] = 1;
1158           AllDoneBits = a;
1159           sending[i] = 1;
1160        if((r[head] == 1) | (sending[head] == 1)){
1161           if(((head + 1) % FIFO_DEPTH)!= tail){
1162             if(head == FIFO_DEPTH - 1){
1163               head = 0;
1164             }
1165               else{
1166               head = head + 1;
1167               }
1168           }
1169        }
1170      }
1171      Head.write(head);
1172 }
```

Bibliography

[1] en.wikipedia.org.

[2] www.electrosofts.com.

[3] www.xilinx.com.

[4] P. Ienne L. Josipovic, R. Ghosal. Dynamically scheduled high-level synthesis. *FPGA '18 Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136, 2018.

[5] M. Kishinevsky J. Cortadella and B. Grundmann. Synthesis of synchronous elastic architectures. *In Proceedings of the 43rd Design Automation Conference*, pages 657 – 62, 2006.

[6] O.Temam Y.Chen "Y.Huang, P.Ienne and C.Wu. Elastic cgras. *In Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays,*, pages 171 – 80, 2013.

[7] P.V.Artigas M.Budiu and S.C. Goldstein. Dataflow: A complement to superscalar. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177 – 86, 2005.

[8] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. *Technical Report CMU-CS-02-107*, 2002.

[9] P.Brisck L.Josipovic and P.Ienne. An out-of-order load-store-queue for spatial computing. *ACM Transactions on Embeded Computing Systems (TECS)*, pages 125:1 – 125:19, 2017.

[10] Y. Chen P. Ienne O. Temam J. Huang, Y. Huang and C. Wu. A low-cost memory interface for high-throughput accelerators. *In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 11:1–11:10, 2014.

[11] V. Betz H. Wong and J. Rose. Efficient methods for out-of-order load/store excution for high-performance soft processors. *In Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 442–445, 2013.