Master Degree Thesis
Using HLS for design space exploration of LoadStore queue

Supervisor: Prof. Lavagno Luciano

Candidate: YUNQING ZHAO

March 2019
Table of contents

1 SystemC Language 7
   1.1 What is SystemC ................................. 7
   1.2 SystemC features ................................ 8
   1.3 The advantage of SystemC .......................... 8

2 Description of the tools 11
   2.1 High Level synthesis ............................. 11
   2.2 Vivado HLS ........................................ 12

3 Dynamically scheduled high level synthesis 13
   3.1 The reason of dynamic scheduling ................. 13
       3.1.1 Circuit of the example ...................... 14
       3.1.2 Comparison between static and dynamical scheduling 15
   3.2 The circuit that I choose ........................ 16
       3.2.1 Circuit architecture ......................... 16
       3.2.2 Connecting to load-store-queue .............. 17
   3.3 Comparison the results ............................ 20
       3.3.1 Comparison with Static HLS ................ 20
       3.3.2 Comparison with other closer approaches .... 21

4 Load-store-queue Design 23
   4.1 Introduction ....................................... 23
   4.2 Load-store-queue architecture .................... 25
       4.2.1 Traditional load-store-queue ................ 25
       4.2.2 A load-store-queue in an out-of-order processor 26
       4.2.3 Comparison our method to others ............ 28
   4.3 Load-store-queue structure ....................... 29
       4.3.1 Overview structure ............................. 30
       4.3.2 Group allocator ............................... 33
5 Synthesis using Vivado HLS tools 35
  5.1 Constrains of the HLS tool 35
  5.2 Optimization solutions 35
    5.2.1 Loop optimization 36
    5.2.2 Array optimization 36

6 Simulation results 45

7 Conclusion 53

Appendices 57

Bibliography 111
List of figures

1.1 Various design stages and the supported stages .................. 9
3.1 Limitations of static scheduling ................................. 14
3.2 Elastic circuit ......................................................... 15
3.3 Circuit architecture .................................................. 16
3.4 Connecting to load-store-queue ..................................... 18
3.5 Connecting the circuit to the memory interface .................... 18
3.6 Compare the results between dynamically scheduled and
    statically scheduled .................................................. 20
3.7 Compare the results between different dynamically
    approaches ............................................................... 21
4.1 A loop example ......................................................... 24
4.2 Typical load-store-queue .............................................. 25
4.3 A load-store-queue with an out-of-order processor ................. 26
4.4 Typical spatial computing system .................................... 28
4.5 Allocating entries statically before execution ..................... 29
4.6 Allocating entries when the arguments are supplied to the
    load-store-queue ....................................................... 29
4.7 Allocating entries by group .......................................... 30
4.8 Load-store-queue structure .......................................... 31
4.9 Group allocator ........................................................ 33
5.1 Latency for array partition .......................................... 36
5.2 Utilization resource for array partition ............................ 37
5.3 Interface1 for array partition ....................................... 38
5.4 Interface2 for array partition ....................................... 39
5.5 Port-num: 1 or 2; Depth: 2 ........................................... 40
5.6 Port-num: 4; Depth: 2 ................................................ 41
5.7 Port-num: 6; Depth: 2 ................................................ 42
5.8 Port-num: 1; Depth: 1 ................................................ 43
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>store-address-port</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>store-data-port</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>load-port</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>bbstart1</td>
<td>47</td>
</tr>
<tr>
<td>6.5</td>
<td>bbstart2</td>
<td>47</td>
</tr>
<tr>
<td>6.6</td>
<td>store-queue1</td>
<td>48</td>
</tr>
<tr>
<td>6.7</td>
<td>store-queue2</td>
<td>49</td>
</tr>
<tr>
<td>6.8</td>
<td>store-queue3</td>
<td>49</td>
</tr>
<tr>
<td>6.9</td>
<td>load-queue1</td>
<td>49</td>
</tr>
<tr>
<td>6.10</td>
<td>load-queue2</td>
<td>50</td>
</tr>
<tr>
<td>6.11</td>
<td>load-queue3</td>
<td>50</td>
</tr>
<tr>
<td>6.12</td>
<td>load-queue4</td>
<td>51</td>
</tr>
<tr>
<td>6.13</td>
<td>load-queue5</td>
<td>51</td>
</tr>
<tr>
<td>6.14</td>
<td>load-queue6</td>
<td>51</td>
</tr>
<tr>
<td>6.15</td>
<td>load-queue7</td>
<td>52</td>
</tr>
<tr>
<td>6.16</td>
<td>load-queue8</td>
<td>52</td>
</tr>
</tbody>
</table>
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 an 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 synthesis tool (Vivado HLS) to acquire 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 execution 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 taught 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 event-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 environment, 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. 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. 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 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 executed 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:

![Figure 1.1: Various design stages and the supported stages](21.png)
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 synthesis

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 generation 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 specifications to be directly targeted into Xilinx programmable devices without the need to manually create RTL.

Supporting both the ISE and Vivado design environments Vivado HLS provides system and design architects alike with a faster 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 describes a methodology to automatically generate dynamically scheduled circuits. The chapter is organized as follows: Section 1 is an 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 compares 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 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.

3.1.1 Circuit of the example

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 represents the most general solution.
3.1 – The reason of dynamic scheduling

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 parallelism on-the-fly and without extensive code preparation.
3.2 The circuit that I choose

In this section, I have chosen the elastic circuit\cite{1} as the paradigm.

3.2.1 Circuit architecture

The architecture for the circuit that I used was shown in figure 3.3\cite{4}.

![Circuit architecture](2.png)

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

<table>
<thead>
<tr>
<th>Kernal</th>
<th>L1_{avg}</th>
<th>CP(ns)</th>
<th>Execution time(us)</th>
<th>DSPs</th>
<th>FFs</th>
<th>Slices</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STAT</td>
<td>DYN</td>
<td>STAT</td>
<td>DYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram</td>
<td>11</td>
<td>2.3</td>
<td>3.3</td>
<td>5.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STAT</td>
<td>DYN</td>
<td>STAT</td>
<td>DYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix power</td>
<td>16</td>
<td>4.2</td>
<td>3.4</td>
<td>6.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STAT</td>
<td>DYN</td>
<td>STAT</td>
<td>DYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>1</td>
<td>1</td>
<td>3.3</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STAT</td>
<td>DYN</td>
<td>STAT</td>
<td>DYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop with condition 1</td>
<td>9</td>
<td>1.3</td>
<td>2.8</td>
<td>4.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STAT</td>
<td>DYN</td>
<td>STAT</td>
<td>DYN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop with condition 2</td>
<td>5</td>
<td>1.2</td>
<td>3.4</td>
<td>4.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. Kernels 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 kernel I have mentioned before.

5. Loop with condition 2 is a variation of the previous kernel where replacing the conditional addition with a multiplication of the same variable.
3.3 – Comparison the results

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].

![Figure 3.7: Compare the results between different dynamically approaches](image.png)

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.
3 – Dynamically scheduled high level synthesis
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 satisfy 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.

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 introduces 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.

![Typical load-store-queue](image)

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 is 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.
4.2 – Load-store-queue architecture

- 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 step 1 (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 step 2 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.
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.
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.
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.
4.3 – 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
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
4.3 – Load-store-queue structure

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.

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 Constraints 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

![Figure 5.1: Latency for array partition](33.png)
## Utilization Estimates

### Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>-</td>
<td>-</td>
<td>909</td>
<td>3909</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>897</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>0</td>
<td>1806</td>
<td>3909</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>650</td>
<td>600</td>
<td>202800</td>
<td>101400</td>
</tr>
<tr>
<td><strong>Utilization (%)</strong></td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5.2: Utilization resource for array partition
Figure 5.3 and 5.4 is the interface after synthesising.

![Image](35.png)

**Interface**

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

Figure 5.3: Interface 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.

![Figure 5.4: Interface2 for array partition](image.png)
### Utilization Estimates

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>-</td>
<td>-</td>
<td>335</td>
<td>2073</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>489</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>0</td>
<td>824</td>
<td>2073</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>650</td>
<td>600</td>
<td>202800</td>
<td>101400</td>
</tr>
<tr>
<td><strong>Utilization (%)</strong></td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.5: Port-num: 1 or 2; Depth: 2
Figure 5.6: Port-num: 4; Depth: 2
### Utilization Estimates

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>-</td>
<td>-</td>
<td>335</td>
<td>2075</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>499</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>0</td>
<td>834</td>
<td>2075</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>650</td>
<td>600</td>
<td>202800</td>
<td>101400</td>
</tr>
<tr>
<td><strong>Utilization (%)</strong></td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.7: Port-num: 6; Depth: 2
### Utilization Estimates

#### Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>-</td>
<td>-</td>
<td>56</td>
<td>1257</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>288</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>0</td>
<td>344</td>
<td>1257</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>650</td>
<td>600</td>
<td>202800</td>
<td>101400</td>
</tr>
<tr>
<td><strong>Utilization (%)</strong></td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>1</td>
</tr>
</tbody>
</table>

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
Obviously, remain figures represent for load-queue results.
6 – Simulation results

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

```c
// 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_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();
    SCCTOR(store_address_port){
        SCMETHOD(id_38);
        SCMETHOD(prc.Counter);
        sensitive pos << CLK;
        sensitive << RST;
        SCMETHOD(id_58);
        SCMETHOD(id_59);
        SCMETHOD(id_60);
    }
};
```

Listing 2: storeaddressport in SystemC

```c
#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();
}
```
if ((temp > 0) & (pValid.read() == 1))
    send_address.write(1);
else
    send_address.write(0);
}

void store_address_port::id_60()
{
    sc_uint<FIFO_DEPTH+1> temp = counter1.read();
    if (temp > 0)
        ReadyToPrev.write(1);
    else
        ReadyToPrev.write(0);
}
Listing 3: storeaddressport in SystemC

```c
#include "systemc.h"
const int FIFODEPTH = 4;
const int DATASIZE = 32;
const int PORTCOUNT = 1;
SC_MODULE(store_data_port){
    SC_IN<sc_cbool> rst, clk, pValid1, port_enalbe1;
    SC_OUT<sc_cbool> readytoprev1, senddata1;
    SC_IN<sc_cuint<DATA_SIZE>> inputdata1;
    SC_OUT<sc_cuint<DATA_SIZE>> data_to_mem1;
    SC_SIGNAL<sc_cuint<FIFODEPTH + 1>> counter2;
    SC_SIGNAL<sc_cbool> send_data_internal1;
    void id104();
    void prec Counter();
    void id125();
    void id126();
    void id127();
    SCCTOR(store_data_port){
        SC_METHOD(id104);
        SC_METHOD(prec.Counter);
        sensitive <<clk.pos();
        SC_METHOD(id125);
        SC_METHOD(id126);
        SC_METHOD(id127);
    }
};
```

Listing 4: storeaddressport in SystemC

```c
#include "storedataport.h"
void store_data_port :: id104(){
    data_to_mem1 = input_data1.read();
}
void store_data_port :: prec.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 :: id125(){
    sc_uint<FIFODEPTH + 1> temp = counter2.read();
    if((temp > 0)&(pValid1 == 1))
        send_data_internal1 = 1;
    else
        send_data_internal1 = 0;
}
void store_data_port :: id126(){
    sc_uint<FIFODEPTH + 1> temp = counter2.read();
    if((temp > 0)&(pValid1 == 1))
        senddata1 = 1;
    else
        senddata1 = 0;
}
void store_data_port :: id127(){
    sc_uint<FIFODEPTH + 1> temp = counter2.read();
}```
if (temp > 0) {
    readytoprev1 = 1;
} else {
    readytoprev1 = 0;
}
Listing 5: storeaddressport in SystemC

```c
#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_out<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;
    void prec.Counter();
    void id_178();
    void id_198();
    void id_200();
    void id_203();
    void id_204();
    void id_205();
    SCCTOR(load_port){
        SC_METHOD(id_178);
        SC_METHOD(prec.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

```c
#include "loadport.h"
void load_port::id_178(){
    address_to_mem2 = input_addr2.read();
}
void load_port::prec.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;
```
else
send_address_internal2 = 0;

void load_port :: id_199()
{
    sc_uint<_FIFODEPTH+1> temp = counter3.read();
    if((temp > 0) & (pValid2 == 1))
        send_address2 = 1;
    else
        send_address2 = 0;
}

void load_port :: id_200()
{
    sc_uint<_FIFODEPTH+1> temp = counter3.read();
    if(temp > 0)
        readytoprev2 = 1;
    else
        readytoprev2 = 0;
}

void load_port :: id_203()
{
    ready_to_mem2 = nReady2.read();
}

void load_port :: id_204()
{
    data_out2 = data_from_mem2.read();
}

void load_port :: id_205()
{
    ValidToNext2 = valid_from_mem2.read();
}
Listing 7: storeaddressport in SystemC

```c
#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_bignint<TAIL_SIZE> BB0LoadOffsets[4];
    static const sc_bignint<TAIL_SIZE> BB0StoreOffsets[4];
    static const sc_bignint<PORT_SIZE> BB0StorePorts[4];
    static const sc_bignint<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_bignint<TAIL_SIZE>> BBLoadOffsets[4];
    sc_out<sc_bignint<TAIL_SIZE>> BBStoreOffsets[4];
    sc_out<sc_bignint<PORT_SIZE>> BBStorePorts[4];
    sc_out<sc_bignint<PORT_SIZE>> BBLoadPorts[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;
    void pre_CalculateLoadSlots_proc();
    void pre_CalculateStoreSlots_proc();
    void pre_EmptySlotComparator1_proc();
    void pre_LoadOffsets_proc();
    void pre_StoreOffsets_proc();
    void pre_BBPorts_proc();

    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

        SC_METHOD(pre_CalculateLoadSlots_proc);
            sensitive << clk.pos();
            sensitive << LoadHead << LoadTail << LoadEmpty;
        SC_METHOD(pre_CalculateStoreSlots_proc);
            sensitive << clk.pos();
            sensitive << StoreHead << StoreTail << StoreEmpty;
    }
}
```

69
Listing 8: storeaddressport in SystemC

```c
#include "bbstart.h"

void BB_START :: prc_CalculateLoadSlots_proc()
{
    //pragma HLS DATAFLOW
    LoadHead = 0;
    sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
    sc_uint<FIFO_DEPTH> loadtail = LoadTail.read();
    if (rst ==1)
        EmptyLoadSlots = 4;
    else if ((loadhead < loadtail) | (LoadEmpty == 1))
        EmptyLoadSlots.write(loadhead - loadtail + FIFO_DEPTH);
    else
        EmptyLoadSlots.write(loadhead - loadtail);
}

void BB_START :: prc_CalculateStoreSlots_proc()
{
```

```c
const sc_biguint<2> BB_START::BB0LoadOffsets [4] = {00,00,00,00};
const sc_biguint<2> BB_START::BB0StoreOffsets [4] = {00,00,00,01};
const sc_biguint<1> BB_START::BB0StorePorts [4] = {0,0,0,0};
const sc_biguint<1> BB_START::BB0LoadPorts [4] = {1,0,0,0};
```
```c
#pragma HLS DATAFLOW
sc_uint<2> currtail = StoreTail.read();
sc_uint<2> currhead = StoreHead.read();
if (rst == 1) EmptyStoreSlots = 4;
else if ((storehead < currtail) | (StoreEmpty == 1))
    EmptyStoreSlots.write(storehead - currtail);
else
    EmptyStoreSlots.write(storehead - currtail);
}
void BB_START::proc_EmptySlotComparator1_proc() {
    //pragma HLS DATAFLOW
    if (rst == 1) BB0Allocate = 0;
    if (BBStart0 == 1) {
        if (EmptyStoreSlots.read() == BB0StoreSize) & (EmptyLoadSlots.read() == BB0LoadSize)
            BB0Allocate = 1;
        else
            BB0Allocate = 0;
    }
    void BB_START::proc_LoadOffsets_proc() {
        //pragma HLS DATAFLOW
        sc_uint<4> current_load_offset;
        sc_uint<4> new_load_tail;
        int i;
        sc_biguint<2> temp[4];
        if (BB0Allocate == 1) {
            for (i = 0; i < FIFODEPTH; i++)
                if (i < LOADMAX) {
                    r = BB0LoadOffsets[i].to_int();
                    current_load_offset = r;
                    new_load_tail = (StoreTail.read() + current_load_offset) % FIFODEPTH;
                    BBLoadOffsets[i].write(new_load_tail.to_int());
                } else
                    BBLoadOffsets[i] = "00";
        } else{
            BBLoadOffsets[0] = "00";
            BBLoadOffsets[1] = "00";
            BBLoadOffsets[2] = "00";
            BBLoadOffsets[3] = "00";
        }
    }
    void BB_START::proc_StoreOffsets_proc() {
        //pragma HLS DATAFLOW
        int current_store_offset;
        int new_store_tail;
        sc_biguint<2> temp[4];
        if (BB0Allocate == 1) {
            for (i = 0; i < FIFODEPTH; i++)
                if (i < STOREMAX) {
                    current_store_offset = BB0StoreOffsets[i].to_int();
                    new_store_tail = (StoreTail.read() + current_store_offset) % FIFODEPTH;
                    BBStoreOffsets[i] = new_store_tail;
                } else
                    BBStoreOffsets[i] = "00";
    }
```
74 } else {
  BBStoreOffsets[0] = "00";
  BBStoreOffsets[1] = "00";
  BBStoreOffsets[2] = "00";
  BBStoreOffsets[3] = "00";
}

78 }
    }

80 }
81 }
82 void BB_START::proc_BBSize_proc(){
  //pragma HLS DATAFLOW
  if (BB0Allocate == 1){
    BBStoreSize = BB0StoreSize;
    BBLoadSize = BB0LoadSize;
  } else{
    BBStoreSize = 0;
    BBLoadSize = 0;
  }
83 }
84 void BB_START::proc_BBPorts_proc(){
  //pragma HLS DATAFLOW
  if (BB0Allocate == 1){
    BBLoadPorts[0] = BB0LoadPorts[0];
    BBLoadPorts[1] = BB0LoadPorts[1];
    BBLoadPorts[2] = BB0LoadPorts[2];
    BBLoadPorts[3] = BB0LoadPorts[3];
    BBStorePorts[0] = BB0StorePorts[0];
    BBStorePorts[1] = BB0StorePorts[1];
    BBStorePorts[2] = BB0StorePorts[2];
    BBStorePorts[3] = BB0StorePorts[3];
  } else{
    BBLoadPorts[0] = "0";
    BBLoadPorts[1] = "0";
    BBLoadPorts[2] = "0";
    BBLoadPorts[3] = "0";
    BBStorePorts[0] = "0";
    BBStorePorts[1] = "0";
    BBStorePorts[2] = "0";
    BBStorePorts[3] = "0";
  }
93 }
94 void BB_START::id_397(){
  //pragma HLS DATAFLOW
  BBStart.write(BB0Allocate.read());
99 }
100 void BB_START::proc_BBready0(){
  //pragma HLS DATAFLOW
  if (rst == 1){
    ReadyToPrev0 = 1;
  } else if ((EmptyStoreSlots.read() == BB0StoreSize) && (EmptyLoadSlots.read() == BB0LoadSize)){
    ReadyToPrev0 = 1;
  } else

ReadyToPrev0 = 0;

void BB_START :: id_419 (){
    //pragma HLS DATAFLOW
    RdPort0_Enable.write ( BB0Allocate.read () );
}

void BB_START :: id_420 (){
    //pragma HLS DATAFLOW
    WrPort0_Enable.write ( BB0Allocate.read () );
}

void BB_START :: id_426 (){
    //pragma HLS DATAFLOW
    BBFirstLoad.write ( BB0Allocate.read () & StoreEmpty.read () );
}

void BB_START :: id_427 (){
    //pragma HLS DATAFLOW
    BBFirstStore = 0;
}

void BB_START :: id_430 (){
    //pragma HLS DATAFLOW
    if ((BB0Allocate == 1) & (StoreEmpty == 1))
        LoadChecks.write ("0000");
    else
        LoadChecks.write ("0001");
}

void BB_START :: id_431 (){
    //pragma HLS DATAFLOW
    StoreChecks.write ("0001");
}
Listing 9: storeaddressport in SystemC

```c
#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, BBSave, Port0_DataWriteEn, BBFirstStore, Port0_AddrWriteEn;
    sc_in<sc_uint<FIFO_DEPTH>> LoadTail, LoadHead, BBStoreSize;
    sc_in<sc_uint<DATA_SIZE>> AddrInPort0;
    sc_in<sc_uint<ADDRESS_SIZE>> LoadAddressQueue[4];
    sc_in<sc_uint<TAIL_NUM>> BBStoreOffsets[4];
    sc_in<sc_uint<PORT_NUM>> BBStorePorts[4];
    sc_out<sc_uint<ADDRESS_SIZE>> StoreAddrQOut[4];
    sc_out<sc_uint<DATA_SIZE>> StoreHeadOut;
    sc_out<sc_uint<DATA_SIZE>> StoreTailOut;
    sc_out<sc_uint<PORT_NUM>> PortQ[4];
    sc_out<sc_uint<ADDRESS_SIZE>> DataQ[4];
    sc_out<sc_uint<DATA_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>> ShiftedAddrDone, ShiftedDataDone;
    sc_signal<sc_uint<FIFO_DEPTH>> ValidEntries, HeadTailBits, HeadEntriesToCheck, Zeros, Ones, EmptyEntries, AllocatedEntries;
    sc_signal<sc_uint<FIFO_DEPTH>> PreviousLoadHead, Head, Tail;
    sc_signal<sc_uint<TAIL_NUM>> TailQ[4], ShiftedTails[4];
    sc_signal<sc_uint<PORT_NUM>> PortQ[4], ShiftedPorts[4];
    sc_signal<sc_uint<ADDRESS_SIZE>> ShiftedLoadAddrQ[4];
    sc_signal<sc_uint<DATA_SIZE>> AddrQ[4];
    sc_signal<sc_uint<ADDRESS_SIZE>> AddrQ[4];
    sc_signal<sc_uint<DATA_SIZE>> DataQ[4];
    void TailUpdate proc();
    void HeadEntriesToCheck proc();
    void ShiftHeadTails proc();
    void HeadDependencyCheck proc();
    void StoreRequest();
}```
void id_783();
void id_784();
void prc_UpdateAllDoneBits_proc();
void prc_SearchPortQPort0_proc();
void prc_InputAddrPriorityPort0_proc();
void prc_UpdateAddressBits();
void prc_UpdateDataBits();
void id_904();
void id_905();

SCCTOR(store_queue){
#pragma HLS ARRAY_PARTITION variable=BBStorePorts complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBStoreOffsets complete dim=1
#pragma HLS ARRAY_PARTITION variable=LoadAddressQueue complete dim=1
#pragma HLS ARRAY_PARTITION variable=PortQ complete dim=1
#pragma HLS ARRAY_PARTITION variable=TailQ complete dim=1
#pragma HLS ARRAY_PARTITION variable=AddrQ complete dim=1
#pragma HLS ARRAY_PARTITION variable=ShiftedPorts complete dim=1
#pragma HLS ARRAY_PARTITION variable=ShiftedTails complete dim=1
#pragma HLS ARRAY_PARTITION variable=StoreDataQOut complete dim=1
#pragma HLS ARRAY_PARTITION variable=StoreAddrQOut complete dim=1
#pragma HLS ARRAY_PARTITION variable=BBStorePorts complete dim=1
#pragma HLS ARRAY_PARTITION variable=LoadAddressQueue complete dim=1

SCMETHOD(id_517);
sensitive << clk.pos();
SCMETHOD(id_518);
sensitive << clk.pos();
SCMETHOD(prc_TailUpdate_proc);
sensitive << clk.pos();
SCMETHOD(prc_InitializeValues_proc);
sensitive << clk.pos();
SCMETHOD(prc_TailQUpdate_proc);
sensitive << clk.pos();
SCMETHOD(prc_PortQUpdate_proc);
sensitive << clk.pos();
SCMETHOD(prc_AllocatedBits_proc);
sensitive << clk.pos();
SCMETHOD(prc_EmptyEntries_proc);
sensitive << clk.pos();
sensitive << AllDoneBits << AllocatedEntries << AddrDoneBits << AllocatedEntries;
SCMETHOD(prc_Empty_proc);
sensitive << clk.pos();
sensitive << EmptyEntries;
SCMETHOD(prc_CheckBits_proc);
sensitive << clk.pos();
SCMETHOD(id_672);
sensitive << clk.pos();
SCMETHOD(id_673);
sensitive << clk.pos();
SCMETHOD(id_674);
sensitive << clk.pos();
SCMETHOD(id_675);
sensitive << clk.pos();
SCMETHOD(prc_ShiftLoadQ);
sensitive << clk.pos();
sensitive << LoadHead << LoadTail << LoadDataDone << LoadAddrDone;
Listing 10: storeaddressport in SystemC

```c
#include "storequeue.h"

void store_queue :: id_517 (){
    sc_uint <FIFO_DEPTH> temp;
    temp[0] = 0;
    temp[1] = 0;
    temp[2] = 0;
    temp[3] = 0;
    Zeroes = temp;
}

void store_queue :: id_518 (){
    //pragma HLS DATAFLOW
    sc_uint <FIFO_DEPTH> temp;
    temp[0] = 1;
    temp[1] = 1;
    temp[2] = 1;
    temp[3] = 1;
    Ones = temp;
}
```
```c
#pragma HLS DATAFLOW
sc_uint<_FIFODEPTH> tail;
sc_uint<_FIFODEPTH> bbstoresize = BBStoreSize.read();

if(rst == 1)
    Tail = 0;
else if(BBStart == 1)
    Tail = (tail + bbstoresize) % FIFODEPTH;
}
void store_queue::prc_InitilizeValues_proc()
#pragma HLS DATAFLOW
#pragma hpragma HLS DATAFLOW
int i;
sc_uint<_FIFODEPTH> temp;
if(rst == 1){
    temp[0] = 0;
    temp[1] = 0;
    temp[2] = 0;
    temp[3] = 0;
    InitBits = temp;
}
else if(BBStart == 1)
    for(i=0;i<_FIFODEPTH;i++)
        if(i < BBStoreSize.read())
            temp[(Tail.read() + i) % FIFODEPTH] = 1;
    InitBits = temp;
}
void store_queue::prc_TailQUpdate_proc(){
#pragma HLS DATAFLOW
#pragma hpragma HLS DATAFLOW
int i;
sc_uint<_FIFODEPTH> tail = Tail.read();
sc_uint<_FIFODEPTH> bbstoresize = BBStoreSize.read();
for(i=0;i<_FIFODEPTH;i++)
    if(i < bbstoresize)
        TailQ[(tail + i) % FIFODEPTH] = BBStoreOffsets[i].read();
}
void store_queue::prc_PortQUpdate_proc(){
#pragma HLS DATAFLOW
#pragma hpragma HLS DATAFLOW
int i;
sc_uint<_FIFODEPTH> tail = Tail.read();
sc_uint<_FIFODEPTH> bbstoresize = BBStoreSize.read();
for(i=0;i<_FIFODEPTH;i++)
    if(i < bbstoresize)
        PortQ[(tail + i) % FIFODEPTH] = BBStorePorts[i].read();
}
void store_queue::prc_AllocatedBits_proc(){
#pragma HLS DATAFLOW
#pragma hpragma HLS DATAFLOW
int j;
sc_uint<_FIFODEPTH> r = InitBits.read();
sc_uint<_FIFODEPTH> h;
if(rst == 1){
    h[0] = 0;
    h[1] = 0;
    h[2] = 0;
    h[3] = 0;
    AllocatedEntries = h;
}
else
    for(j=0;j<_FIFODEPTH;j++)
```
if (r[j] == 1)
    h[j] = 1;
AllocatedEntries = h;
}
void store_queue :: prc_EmptyEntries_proc()
{
    //pragma HLS DATAFLOW
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> temp1 = AllDoneBits.read();
    sc_uint<FIFO_DEPTH> temp2 = AllocatedEntries.read();
    sc_uint<FIFO_DEPTH> temp;
    if (rst == 1){
        temp[0] = 1;
        temp[1] = 1;
        temp[2] = 1;
        temp[3] = 1;
        EmptyEntries = temp;
    }
    else
    for (i = 0; i < FIFO_DEPTH; i++){
        if (temp1[i] == 1) | (temp2[i] == 0)
            temp[i] = 1;
        else
            temp[i] = 0;
    }
    EmptyEntries = temp;
}
void store_queue :: prc_Empty_proc()
{
    //pragma HLS DATAFLOW
    if (EmptyEntries.read() == Ones.read())
        StoreEmpty = 1;
    else
        StoreEmpty = 0;
}
void store_queue :: prc_ChekBits_proc()
{
    //pragma HLS DATAFLOW
    //pragma HLS DATAFLOW
    int i;
    int j;
    sc_uint<FIFO_DEPTH> current_tail;
    sc_uint<FIFO_DEPTH> r ;
    sc_uint<FIFO_DEPTH> temp;
    sc_uint<FIFO_DEPTH> h = StoreChecks.read();
    sc_uint<FIFO_DEPTH> previousloadhead ;
    sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
    sc_uint<FIFO_DEPTH> tail = Tail.read();
    sc_uint<FIFO_DEPTH> bbstoresize = BBStoreSize.read();
    if (rst == 1){
        r[0] = 0;
        r[1] = 0;
        r[2] = 0;
        r[3] = 0;
        CheckBits = r;
    }
    else
        PreviousLoadHead.write(loadhead);
    for (j = 0; j < FIFO_DEPTH; j++)
        if ((j < bbstoresize) & (BBStart == 1))
            r[(tail + j) % FIFO_DEPTH] = h[j];
```c
else
    current_tail = TailQ[j].read().to_int();
if (((previous_loadhead = (((current_tail - 1) % FIFO_DEPTH) &
                      loadhead)) > current_tail)
    r[j] = 0;
else if (((previous_loadhead = (((current_tail - 1) % FIFO_DEPTH) &
                      loadhead)) < PreviousLoadHead.read())
    r[j] = 0;
    CheckBits = r;
    PreviousLoadHead = previous_loadhead;
}

void store_queue::id672()
{
    StoreAddrDoneOut = AdrDoneBits.read();
}
void store_queue::id673()
{
    StoreDataDoneOut = DataDoneBits.read();
}
void store_queue::id674()
{
    //pragma HLS DATAFLOW
    sc_biguint<ADDRESS_SIZE> addrq[4];
    addrq[0] = AddrQ[0].read();
    addrq[1] = AddrQ[1].read();
    StoreAddrQOut[0] = addrq[0];
    StoreAddrQOut[1] = addrq[1];
    StoreAddrQOut[3] = addrq[3];
}
void store_queue::id675()
{
    //pragma HLS DATAFLOW
    sc_biguint<DATA_SIZE> dataq[4];
    dataq[0] = DataQ[0].read();
    dataq[1] = DataQ[1].read();
    StoreDataQOut[0] = dataq[0];
    StoreDataQOut[1] = dataq[1];
    StoreDataQOut[2] = dataq[2];
    StoreDataQOut[3] = dataq[3];
}
void store_queue::id676()
{
    //pragma HLS DATAFLOW
    sc_uint<9> load_upper_bound;
    sc_uint<FIFO_DEPTH> r;
    sc_uint<FIFO_DEPTH> h = LoadAdrDone.read();
    sc_uint<FIFO_DEPTH> b;
    sc_uint<FIFO_DEPTH> a = LoadDataDone.read();
    sc_uint<FIFO_DEPTH> v;
    sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
    sc_uint<FIFO_DEPTH> loadtail = LoadTail.read();
    sc_uint<FIFO_DEPTH> temp;
    sc_uint<FIFO_DEPTH> temp1;
    if (rst == 1){
        v[0] = 1;
        v[1] = 1;
        v[2] = 1;
```
v[3] = 1;
ValidEntries = v;
r[0] = 0;
r[1] = 0;
r[2] = 0;
r[3] = 0;
ShiftedAdrDone = r;
b[0] = 0;
b[1] = 0;
b[2] = 0;
b[3] = 0;
ShiftedDataDone = b;
}
else if (loadtail == loadhead)
load_upper_bound = loadtail + FIFO_DEPTH;
else
load_upper_bound = loadtail;
for (i=0;i<FIFO_DEPTH;i++){
ShiftedLoadAddrQ[i] = LoadAddressQueue[(i+loadhead) % FIFO_DEPTH].read();
r[i] = h[(i+loadhead) % FIFO_DEPTH];
b[i] = a[(i+loadhead) % FIFO_DEPTH];
if ((i + loadhead) < load_upper_bound)
v[i] = 1;
else
v[i] = 0;
ValidEntries = v;
ShiftedAdrDone = r;
ShiftedDataDone = b;
}
void store_queue :: prc_HeadEntriesToCheck_proc()
{
#pragma HLS DATAFLOW
#pragma HLS DATAFLOW
int i;
sc_uint<FIFO_DEPTH> v = ValidEntries.read();
sc_uint<FIFO_DEPTH> r ;
sc_uint<FIFO_DEPTH> h = HeadTailsBits.read();
sc_uint<FIFO_DEPTH> c = CheckBits.read();
sc_uint<FIFO_DEPTH> head = Head.read();
if (rst ==1){
r[0] = 0;
r[1] = 0;
r[2] = 0;
r[3] = 0;
HeadEntriesToCheck = r;
}
else
for (i=0;i<FIFO_DEPTH;i++)
r[i] = h[i] & v[i] & c[head.to_int()];
HeadEntriesToCheck = r;
}
void store_queue :: prc_ShiftHeadTails_proc()
{
#pragma HLS DATAFLOW
#pragma HLS DATAFLOW
int current_tail;
sc_uint<9> load_upper_bound;
int i;
sc_uint<FIFO_DEPTH> h ;
sc_uint<FIFO_DEPTH> loadhead = LoadHead.read();
81
266 current_tail = TailQ[Head().read().to_int();
if (rst == 1){
    h[0] = 1;
    h[1] = 1;
    h[2] = 1;
    h[3] = 1;
    HeadTailsBits = h;
}
else if(((current_tail - 1) % FIFODEPTH) == loadhead)
    load_upper_bound = current_tail + FIFODEPTH;
else if(((current_tail - 1) % FIFODEPTH) == loadhead) & (LoadEmpty
    == 1))
    load_upper_bound = current_tail;
else
    load_upper_bound = current_tail;
for (i = 0; i < FIFODEPTH; i++)
if ((i + loadhead) < load_upper_bound)
    h[i] = 1;
else
    h[i] = 0;
HeadTailsBits = h;
}

82 void store_queue::prc_HeadDependencyCheck_proc(){
    //pragma HLS DATAFLOW
    //pragma HLS DATAFLOW
    bool read_stall;
    sc_uint<FIFODEPTH> r = ShiftedAddrDone.read();
    sc_uint<FIFODEPTH> h = HeadEntriesToCheck.read();
    sc_uint<FIFODEPTH> d = DataDoneBits.read();
    sc_uint<FIFODEPTH> s = ShiftedDataDone.read();
    sc_uint<FIFODEPTH> a = AddrDoneBits.read();
    sc_uint<FIFODEPTH> b = AllDoneBits.read();
    sc_uint<FIFODEPTH> head = Head().to_int();
    sc_uint<ADDRESS_SIZE> addrq[4];
    sc_uint<ADDRESS_SIZE> shiftedloadaddrq[4];
    int i;
    if (rst == 1){
        StoreRequest = 0;
    }
    else if(((a[head] == 1) & (d[head] == 1)) & (b[head] == 0))
    read_stall = 0;
for (i = 0; i < FIFODEPTH; i++)
if (h[i] == 1)
    if ((r[i] == 1) & (s[i] == 0))
    addrq[0] = AddrQ[0].read();
    shiftedloadaddrq[i] = ShiftedLoadAddrQ[i].read();
    if (addrq[0] == shiftedloadaddrq[i])
    read_stall = read_stall | 1;
for (i = 0;
    read_stall = read_stall | 1;
    if (read_stall == 0)
    StoreRequest = 1;
}
void store_queue::id_782(){
    //pragma HLS DATAFLOW
    MemoryStoreEnable = StoreRequest;
}
void store_queue::id_783(){
    //pragma HLS DATAFLOW
```cpp
sc_uint<FIFO_DEPTH> head = Head.read();
MemoryDataOut = DataQ[head].read();
}
void store_queue::id_T84() {
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> head = Head.read();
    MemoryAddressOut = AddrQ[head].read();
}
void store_queue::prc_UpdateAllDoneBits_proc() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> b;
    sc_uint<FIFO_DEPTH> r = InitBits.read();
    sc_uint<FIFO_DEPTH> head = Head.read();
    if(rst == 1) {
        AllDoneBits = "0000";
        Head = 0;
    } else {
        for (i=0;i<FIFO_DEPTH;i++) {
            if (r[i] == 1)
                b[i] = 0;
        AllDoneBits = b;
    }
    if (StoreRequest == 1) {
        b[head] = 1;
        AllDoneBits = b;
        if (head == FIFO_DEPTH-1)
            Head.write(0);
        else
            Head.write(head + 1);
    }
}
void store_queue::prc_SearchPortQPort0_proc() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> r;
    if(rst == 1)
        EntriesPort0 = 0;
    else
        for (i=0;i<FIFO_DEPTH;i++) {
            sc_uint<PORT_NUM> h = PortQ[i].read();
            if (h == 0)
                r[i] = 1;
            else
                r[i] = 0;
        EntriesPort0 = r;
    }
}
void store_queue::prc_InputAddrPriorityPort0_proc() {
    //pragma HLS DATAFLOW
    bool blocked;
    int i;
    sc_uint<FIFO_DEPTH> a = AdrDoneBits.read();
    sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
    sc_uint<FIFO_DEPTH> h = InputAddrPriorityPort0.read();
    sc_uint<FIFO_DEPTH> head = Head.read();
    if (rst == 1) {
        blocked = 0;
        InputAddrPriorityPort0 = "0000";
    }
```
} else
for (i = 0; i < FIFO_DEPTH; i++)
if (((r[(i + head) % FIFO_DEPTH] == 1) && (a[(i + head) % FIFO_DEPTH] == 0)) {
    if (blocked == 0) {
        h[(i + head) % FIFO_DEPTH] = 1;
        InputAddrPriorityPort0 = h;
        blocked = 1;
    }
}

} void store_queue :: prec_UpdateAddressBits()
{//pragma HLS DATAFLOW
    int i;
    sc_uint<ADDRESS_SIZE> a;
    sc_uint<ADDRESS_SIZE> r = InputAddrPriorityPort0.read();
    sc_uint<ADDRESS_SIZE> h = Head.read();
    if (rst == 1)
        AdrDoneBits = "0000";
    else
        for (i = 0; i < FIFO_DEPTH; i++)
            if ((r[i] == 1) && (h[i] == 1) && a[i] == 0) {
                // sc_uint<ADDRESS_SIZE> addrq = AddrQ[i].read();
                AddrQ[0].write(temp);
                AddrQ[i].write(temp);
                a[i] = 1;
            }
    AdrDoneBits = a;
} void store_queue :: prec_UpdateDataBits()
{//pragma HLS DATAFLOW
    int i;
    sc_uint<ADDRESS_SIZE> d;

sc_uint<DATA_SIZE> temp = DataInPort0.read();
sc_uint<FIFO_DEPTH> r = InitBits.read();
sc_uint<FIFO_DEPTH> h = InputAddrPriorityPort0.read();

if (rst == 1)
    DataDoneBits = "0000";
else
    for (i=0; i<FIFODEPTH; i++){
        if (r[i] == 1)
            d[i] = 0;
        else if((h[i] == 1) & (d[i] == 0) & (Port0_DataWriteEn == 1)){
            // sc_uint<DATA_SIZE> dataq = DataQ[i].read();
            DataQ[0].write(temp);
            DataQ[i].write(temp);
            d[i] = 1;
        }
    }
    DataDoneBits = d;
}

void store_queue::id_904(){
    StoreTailOut = Tail.read();
}

void store_queue::id_905(){
    StoreHeadOut = Head.read();
}
Listing 11: storeaddressport in SystemC

```cpp
#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<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>> LoadAddrOut[4];
    sc_out<sc_uint<FIFO_DEPTH>> LoadTailOut, LoadHeadOut;
    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>> Data0;
    sc_signal<sc_biguint<DATA_SIZE>> Data0;
    sc_signal<sc_uint<FIFO_DEPTH>> CheckBits, AdrDoneBits, DataDoneBits,
    AllDoneBits;
    sc_signal<sc_uint<FIFO_DEPTH>> PendingBits, InitBits,
    ShiftedAdrDone, ShiftedDataDone;
    sc_signal<sc_uint<FIFO_DEPTH>> 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_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();
}
```

87
void prc_TailUpdate_proc();
void prc_InitializeValues_proc();
void prc_TailQUpdate_proc();
void prc_PortQUpdate_proc();
void prc_Empty_proc();
void prc_EmptyEntries_proc();
void prc_CheckBits_proc();
void prc_AllocatedBits_proc();
void prc_ShiftStoreQ();
void prc_PriorityFunction_proc();
void prc_PendingReg_proc();
void id_1198();
void id_1199();
void id_1200();
void id_1201();
void id_1202();
void prc_ShiftedDataQ_proc0();
void prc_ShiftLoadTails0_proc();
void prc_EntriesToCheck0_proc();
void prc_Comparators0();
void prc_FindLastConflict0();
void prc_Empties0_proc();
void prc_Decision0();
void id_1399();
void prc_ShiftLoadTails1_proc();
void prc_EntriesToCheck1_proc();
void prc_Comparators1();
void prc_FindLastConflict1();
void prc_Empties1_proc();
void prc_Decision1();
void id_1515();
void prc_ShiftLoadTails2_proc();
void prc_EntriesToCheck2_proc();
void prc_Comparators2();
void prc_FindLastConflict2();
void prc_Empties2_proc();
void prc_Decision2();
void id_1631();
void prc_ShiftLoadTails3_proc();
void prc_EntriesToCheck3_proc();
void prc_Comparators3();
void prc_FindLastConflict3();
void prc_Empties3_proc();
void prc_Decision3();
void id_1746();
void prc_ChooseMemRequest_proc();
void prc_DatadoneBits_proc();
void prc_DataQUpdate_proc0();
void id_1829();
void prc_DataQUpdate_proc1();
void id_1847();
void prc_DataQUpdate_proc2();
void id_1865();
void prc_DataQUpdate_proc3();
void id_1883();
void prc_SearchPortQPort0_proc();
void prc_InputPriorityPort0_proc();
void prc_OutputPriorityPort0_proc();
void prc_SendToPort0_proc();
void prc_UpdateAddressBits();
void prc_UpdateAllDoneBits_proc();
SCCTOR{ load_queue}{
#pragma HLS ARRAY_PARTITION variable=AddrQ complete dim=1
#pragma HLS ARRAY_PARTITION variable=DataQ complete dim=1
//#pragma HLS ARRAY_PARTITION variable=PortQ complete dim=1
//#pragma HLS ARRAY_PARTITION variable=ShiftedStoreDataQ complete dim=1
//#pragma HLS ARRAY_PARTITION variable=ShiftedStoreAddrQ complete dim=1
//#pragma HLS ARRAY_PARTITION variable=ShiftedPorts complete dim=1
//#pragma HLS ARRAY_PARTITION variable=ShiftedTails complete dim=1
//#pragma HLS ARRAY_PARTITION variable=LoadAddrQOut 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=StoreDataQueue complete dim=1
//#pragma HLS ARRAY_PARTITION variable=StoreAddressQueue complete dim=1

SC_METHOD(id_1044);
sensitive << clk.pos();
SC_METHOD(id_1045);
sensitive << clk.pos();
SC_METHOD(proc_TailUpdate_proc);
sensitive << clk.pos();
SC_METHOD(proc_InitializeValues_proc);
sensitive << clk.pos();
SC_METHOD(proc_BBStart<尾>>BBLoadSize);
SC_METHOD(proc_TailQUpdate_proc);
sensitive << clk.pos();
SC_METHOD(proc_PortQUpdate_proc);
sensitive << clk.pos();
SC_METHOD(proc_CheckBits_proc);
sensitive << clk.pos();
SC_METHOD(proc_AliveBits_proc);
sensitive << clk.pos();
SC_METHOD(proc_Empty_proc);
sensitive << clk.pos();
SC_METHOD(proc_LoadRequests<头>);
SC_METHOD(proc_PendingReg_proc);
sensitive << clk.pos();
SC_METHOD(id_1198);
sensitive << clk.pos();
SC_METHOD(id_1199);
sensitive << clk.pos();
SC_METHOD(id_1200);
sensitive << clk.pos();
SC_METHOD(id_1201);
sensitive << clk.pos();
SC_METHOD(id_1202);
sensitive << clk.pos();
SC_METHOD(proc_ShiftedDataQ_proc0);
sensitive << clk.pos();
SC_METHOD(proc_ShaftLoadTails0_proc);
sensitive << clk.pos();
SC_METHOD(proc_EntriesToCheck0_proc);
sensitive << clk.pos();

89
170 sensitive <<LoadTailsBits0<<ValidEntries<<CheckBits;
171 SC_METHOD(pre_Comparators0);
172 sensitive <<clk.pos();
173 sensitive <<EntriesToCheck0<<ShiftedAdrDone;
174 SC_METHOD(pre_FindLastConflict0);
175 sensitive <<clk.pos();
176 sensitive <<Conflict0<<ShiftedDataDone;
177 SC_METHOD(pre_Empties0_proc);
178 sensitive <<clk.pos();
179 sensitive <<ShiftedAdrDone<<EntriesToCheck0;
180 SC_METHOD(pre_Decision0);
181 sensitive <<clk.pos();
182 SC_METHOD(id_1399);
183 sensitive <<clk.pos();
184 SC_METHOD(pre_ShiftLoadTails1_proc);
185 sensitive <<clk.pos();
186 sensitive <<StoreHead;
187 SC_METHOD(pre_EntriesToCheck1_proc);
188 sensitive <<clk.pos();
189 sensitive <<LoadTailsBits1<<ValidEntries<<CheckBits;
190 SC_METHOD(pre_Comparators1);
191 sensitive <<clk.pos();
192 sensitive <<EntriesToCheck1<<ShiftedAdrDone;
193 SC_METHOD(pre_FindLastConflict1);
194 sensitive <<clk.pos();
195 sensitive <<Conflict1<<ShiftedDataDone;
196 SC_METHOD(pre_Empties1_proc);
197 sensitive <<clk.pos();
198 sensitive <<ShiftedAdrDone<<EntriesToCheck1;
199 SC_METHOD(pre_Decision1);
200 sensitive <<clk.pos();
201 SC_METHOD(id_1515);
202 sensitive <<clk.pos();
203 SC_METHOD(pre_ShiftLoadTails2_proc);
204 sensitive <<clk.pos();
205 sensitive <<StoreHead;
206 SC_METHOD(pre_EntriesToCheck2_proc);
207 sensitive <<clk.pos();
208 sensitive <<LoadTailsBits2<<ValidEntries<<CheckBits;
209 SC_METHOD(pre_Comparators2);
210 sensitive <<clk.pos();
211 sensitive <<EntriesToCheck2<<ShiftedAdrDone;
212 SC_METHOD(pre_FindLastConflict2);
213 sensitive <<clk.pos();
214 sensitive <<Conflict2<<ShiftedDataDone;
215 SC_METHOD(pre_Empties2_proc);
216 sensitive <<clk.pos();
217 sensitive <<ShiftedAdrDone<<EntriesToCheck2;
218 SC_METHOD(pre_Decision2);
219 sensitive <<clk.pos();
220 SC_METHOD(id_1631);
221 sensitive <<clk.pos();
222 SC_METHOD(pre_ShiftLoadTails3_proc);
223 sensitive <<clk.pos();
224 sensitive <<StoreHead;
225 SC_METHOD(pre_EntriesToCheck3_proc);
226 sensitive <<clk.pos();
227 sensitive <<LoadTailsBits3<<ValidEntries<<CheckBits;
228 SC_METHOD(pre_Comparators3);
229 sensitive <<clk.pos();
230 sensitive <<EntriesToCheck3<<ShiftedAdrDone;
231 SC_METHOD(pre_FindLastConflict3);
sensitive << clk.pos();
sensitive << Conflict3 << ShiftedDataDone;
SC_METHOD(prc_Empties3_proc);
sensitive << clk.pos();
sensitive << ShiftedAdrDone << EntriesToCheck3;
SC_METHOD(prc_Decision3);
sensitive << clk.pos();
SC_METHOD(id_1746);
sensitive << clk.pos();
SC_METHOD(id_1829);
sensitive << clk.pos();
SC_METHOD(id_1847);
sensitive << clk.pos();
SC_METHOD(id_1865);
sensitive << clk.pos();
SC_METHOD(id_1883);
sensitive << clk.pos();
SC_METHOD(id_1901);
sensitive << clk.pos();
SC_METHOD(id_1919);
sensitive << clk.pos();
SC_METHOD(id_1937);
sensitive << clk.pos();
SC_METHOD(id_1955);
sensitive << clk.pos();
SC_METHOD(id_1973);
sensitive << clk.pos();
SC_METHOD(id_1991);
sensitive << clk.pos();
SC_METHOD(id_2011);
sensitive << clk.pos();
SC_METHOD(id_2029);
sensitive << clk.pos();
SC_METHOD(id_2047);
sensitive << clk.pos();
SC_METHOD(id_2065);
sensitive << clk.pos();
SC_METHOD(id_2083);
sensitive << clk.pos();
SC_METHOD(id_2101);
sensitive << clk.pos();
SC_METHOD(id_2119);
sensitive << clk.pos();
SC_METHOD(id_2137);
sensitive << clk.pos();
SC_METHOD(id_2155);
sensitive << clk.pos();
SC_METHOD(id_2173);
sensitive << clk.pos();
SC_METHOD(id_2191);
sensitive << clk.pos();
SC_METHOD(id_2211);
sensitive << clk.pos();
SC_METHOD(id_2229);
sensitive << clk.pos();
SC_METHOD(id_2247);
sensitive << clk.pos();
SC_METHOD(id_2265);
sensitive << clk.pos();
SC_METHOD(id_2283);
sensitive << clk.pos();
SC_METHOD(id_2301);
sensitive << clk.pos();
SC_METHOD(id_2319);
sensitive << clk.pos();
SC_METHOD(id_2337);
sensitive << clk.pos();
SC_METHOD(id_2355);
sensitive << clk.pos();
SC_METHOD(id_2373);
sensitive << clk.pos();
SC_METHOD(id_2391);
sensitive << clk.pos();
SC_METHOD(id_2409);
sensitive << clk.pos();
SC_METHOD(id_2427);
sensitive << clk.pos();
SC_METHOD(id_2445);
sensitive << clk.pos();
SC_METHOD(id_2463);
sensitive << clk.pos();
SC_METHOD(id_2481);
sensitive << clk.pos();
SC_METHOD(id_2501);
sensitive << clk.pos();
SC_METHOD(id_2519);
sensitive << clk.pos();
SC_METHOD(id_2537);
sensitive << clk.pos();
SC_METHOD(id_2555);
sensitive << clk.pos();
SC_METHOD(id_2573);
sensitive << clk.pos();
SC_METHOD(id_2591);
sensitive << clk.pos();
SC_METHOD(id_2609);
sensitive << clk.pos();
SC_METHOD(id_2627);
sensitive << clk.pos();
SC_METHOD(id_2645);
sensitive << clk.pos();
SC_METHOD(id_2663);
sensitive << clk.pos();
SC_METHOD(id_2681);
sensitive << clk.pos();
SC_METHOD(id_2701);
sensitive << clk.pos();
SC_METHOD(id_2719);
sensitive << clk.pos();
SC_METHOD(id_2737);
sensitive << clk.pos();
SC_METHOD(id_2755);
sensitive << clk.pos();
SC_METHOD(id_2773);
sensitive << clk.pos();
SC_METHOD(id_2791);
sensitive << clk.pos();
SC_METHOD(id_2809);
sensitive << clk.pos();
SC_METHOD(id_2827);
sensitive << clk.pos();
SC_METHOD(id_2845);
sensitive << clk.pos();
SC_METHOD(id_2863);
sensitive << clk.pos();
SC_METHOD(id_2881);
sensitive << clk.pos();
SC_METHOD(id_2901);
sensitive << clk.pos();
SC_METHOD(id_2919);
}
};

Listing 12: storeaddressport in SystemC

#include "loadqueue.h"
void load_queue :: id_1044()
if (rst == 1)
Zeroes = "0000";
}
void load_queue :: id_1045()
#pragma HLS DATAFLOW
sc_uint<FIFO_DEPTH> temp;
if (rst == 1) {
temp[0] = 1;
temp[1] = 1;
```c
} else if (BBStart == 1)
    temp = (tail + bblocksize) % FIFO_DEPTH;
    Tail = temp;
}

void load_queue :: prc_InitValues_proc()
{
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> w = InitBits.read();
    sc_uint<FIFO_DEPTH> bblocksize = BB_loadSize.read();
    sc_uint<FIFO_DEPTH> tail = Tail.read();
    if (rst == 1)
        InitBits = "0000";
    else if (BBStart == 1)
        for (i = 0; i < FIFO_DEPTH; i++)
            if (i < bblocksize)
                InitBits = w[(tail + i) % FIFO_DEPTH] = 1;
    }

void load_queue :: prc_TailUpdate_proc()
{
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> bblocksize = BB_loadSize.read();
    sc_uint<FIFO_DEPTH> tail = Tail.read();
    if (i == 0)
        BB_loadOffsets = BB_loadOffsets[i].read();
    for (i = 0; i < FIFO_DEPTH; i++)
        TailQ[(tail + i) % FIFO_DEPTH] = bblocksize;
}

void load_queue :: prc_ProtUpdate_proc()
{
    //pragma HLS DATAFLOW
    int i;
    //pragma HLS PIPELINE II=1
    if (i == BB_loadSize.read())
    {
        sc_uint<1> r = PortQ[(Tail.read() + i) % FIFO_DEPTH].read();
        r = BB_loadPorts[i].read();
        PortQ[(Tail.read() + i) % FIFO_DEPTH] = r;
    }
}

void load_queue :: prc_Empty_proc()
{
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> r = Ones.read();
    sc_uint<FIFO_DEPTH> h = EmptyEntries.read();
    if (h == r)
        LoadEmpty = 1;
```
else
    LoadEmpty = 0;
}

void load_queue :: prc_EmptyEntries_proc()
{//#pragma HLS DATAFLOW
    int i;
    sc_uint< FIFO_DEPTH > a = AllocatedEntries.read();
    sc_uint< FIFO_DEPTH > k = AllDoneBits.read();
    sc_uint< FIFO_DEPTH > h;

    for (i = 0; i < FIFO_DEPTH; i++)
        if ((k[i] == 1) || (a[i] == 0))
            h[i] = 1;
        else
            h[i] = 0;
    EmptyEntries = h;
}

void load_queue :: prc_CheckBits_proc()
{//#pragma HLS DATAFLOW
    int j;
    int current_tail;
    sc_uint< FIFO_DEPTH > r;
    sc_uint< FIFO_DEPTH > h = LoadChecks.read();
    sc_uint< FIFO_DEPTH > storehead = StoreHead.read();
    sc_uint< FIFO_DEPTH > previousstorehead;

    if (rst == 1)
        CheckBits = "0000";
    else {
        PreviousStoreHead = storehead;
        r = CheckBits.read();
        for (j = 0; j < FIFO_DEPTH; j++)
            if ((j < BBLoadSize.read()) && (BBStart == 1))
                r[(TailQ[j].read() + j) % FIFO_DEPTH] = h[j];
            else
                current_tail = TailQ[j].read().to_int();
                if ((previousstorehead == (((current_tail - 1) % FIFO_DEPTH) &
                    storehead)) > current_tail)
                    r[j] = 0;
                else if ((previousstorehead == (((current_tail - 1) % FIFO_DEPTH)
                    & storehead)) < PreviousStoreHead.read())
                    r[j] = 0;
        CheckBits = r;
    }

    void load_queue :: prc_AllocatedBits_proc()
{//#pragma HLS DATAFLOW
        int j;
        sc_uint< FIFO_DEPTH > w = InitBits.read();
        sc_uint< FIFO_DEPTH > a;

        if (rst == 1)
            AllocatedEntries = "0000";
        else
            for (j = 0; j < FIFO_DEPTH; j++)
                if (w[j] == 1)
                    a[j] = 1;
        AllocatedEntries = a;
    }

    void load_queue :: id_1198()
```cpp
#pragma HLS DATAFLOW
LoadAdrDoneOut.write(AdrDoneBits.read());
}

void load_queue::id_1199()
{
#pragma HLS DATAFLOW
LoadDataDoneOut.write(DataDoneBits.read());
}

void load_queue::id_1200()
{
#pragma HLS DATAFLOW
LoadAddrQOut[0].write(AddrQ[0].read());
LoadAddrQOut[1].write(AddrQ[1].read());
LoadAddrQOut[2].write(AddrQ[2].read());
LoadAddrQOut[3].write(AddrQ[3].read());
}

void load_queue::id_1201()
{
#pragma HLS DATAFLOW
LoadTailOut.write(Tail.read());
}

void load_queue::id_1202()
{
#pragma HLS DATAFLOW
LoadHeadOut.write(Head.read());
}

void load_queue::pre_ShiftStoreQ()
{
int store_upper_bound;
int i;
sc_uint<FIFODEPTH> storehead = StoreHead.read();
sc_uint<FIFODEPTH> storetail = StoreTail.read();
sc_uint<FIFODEPTH> w;
sc_uint<FIFODEPTH> q;
sc_uint<FIFODEPTH> c = StoreAdrDone.read();
sc_uint<FIFODEPTH> d = StoreDataDone.read();
sc_uint<FIFODEPTH> f;
sc_biguint<DATA_SIZE> shiftedstoredataq[4];
sc_biguint<ADDRESS_SIZE> shiftedstoreaddrq[4];
if (rst ==1)
{
    ValidEntries = 0;
    else if (storetail == storehead)
    
        store_upper_bound = storetail + FIFODEPTH;
    else
        store_upper_bound = storetail;
    
    for (i=0;i<FIFODEPTH;i++)
    {
#pragma HLS PIPELINE II=1
        sc_biguint<ADDRESS_SIZE> h = StoreAddressQueue[(i + storehead) % FIFODEPTH].read();
        sc_biguint<DATA_SIZE> r = StoreDataQueue[(i + storehead) % FIFODEPTH].read();
        shiftedstoredataq[i] = r;
        shiftedstoreaddrq[i] = h;
        ShiftedStoreDataQ[i] = shiftedstoredataq[i];
        ShiftedStoreAddrQ[i] = shiftedstoreaddrq[i];
        w[i] = c[(i + storehead) % FIFODEPTH];
        q[i] = d[(i + storehead) % FIFODEPTH];
        ShiftedAdrDone = w;
        ShiftedDataDone = q;
        if ((i + storehead) < store_upper_bound)
            f[i] = 1;
        else
            f[i] = 0;
        ValidEntries = f;
    }
}
```
```cpp
void load_queue :: prc_PriorityFunction_proc()
{
    //pragma HLS DATAFLOW
    bool prio_req;
    int i;
    int j;
    int head = Head.read();
    sc_uint<_FIFO_DEPTH> r = LoadRequests.read();
    sc_uint<_FIFO_DEPTH> h;
    if (rst == 1)
        PriorityOut = "0000";
    else
    {
        h[head] = r[head];
        PriorityOut = h;
    }
    for (i = 1; i < _FIFO_DEPTH - 1; i++)
    {
        //pragma HLS PIPELINE II=1
        prio_req = 0;
        for (j = 1; j < i - 1; j++)
        {
            //pragma HLS PIPELINE II=1
            h[(head + j - 1) % _FIFO_DEPTH] = r[(head + j - 1) % _FIFO_DEPTH].to_bool() & ~prio_req;
        }
        PriorityOut = h;
    }
}

void load_queue :: prc_PendingReg_proc()
{
    //pragma HLS DATAFLOW
    int i;
    sc_uint<_FIFO_DEPTH> r = PriorityOut.read();
    sc_uint<_FIFO_DEPTH> h = PendingBits.read();
    if (rst == 1)
        PendingBits = "0000";
    else
        for (i = 0; i < _FIFO_DEPTH; i++)
        {
            //pragma HLS PIPELINE II=1
            if (r[i] == 1)
                h[i] = 1;
            else
                h[i] = 0;
        }
    PendingBits = h;
}

void load_queue :: prc_ShiftedDataQ_proc0()
{
    //pragma HLS DATAFLOW
    ShiftedStoreDataQ2[0].write(ShiftedStoreDataQ[0].read());
    ShiftedStoreDataQ2[1].write(ShiftedStoreDataQ[1].read());
    ShiftedStoreDataQ2[2].write(ShiftedStoreDataQ[2].read());
    ShiftedStoreDataQ2[3].write(ShiftedStoreDataQ[3].read());
}

void load_queue :: prc_ShiftLoadTails0_proc()
{
    //pragma HLS DATAFLOW
    int current_tail;
    int store_upper_bound;
    int i;
    sc_uint<_FIFO_DEPTH> storehead = StoreHead.read();
    sc_uint<2> tailq = TailQ[0].read().to_int();
    sc_uint<_FIFO_DEPTH> r;
    if (rst == 1)
        LoadTailsBits0 = 0;
    else
```
```c
current_tail = tailq;
if(((current_tail - 1) % FIFODEPTH) < storehead)
    store_upper_bound = current_tail + FIFODEPTH;
else if(((current_tail - 1) % FIFODEPTH) == storehead) & StoreEmpty
    store_upper_bound = current_tail;
else
    store_upper_bound = current_tail;
for (i = 0; i < FIFODEPTH; i++)
for (i = 0; i < FIFODEPTH; i++){
    //pragma HLS PIPELINE II=1
    r[i] = 1;
    else
        r[i] = 0;
LoadTailsBits0 = r;
}
}
void load_queue::prc_EntriesToCheck0_proc() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFODEPTH> r;
    sc_uint<FIFODEPTH> h = LoadTailsBits0.read();
    sc_uint<FIFODEPTH> w = ValidEntries.read();
    sc_uint<FIFODEPTH> a = CheckBits.read();
    if(rst == 1)
        EntriesToCheck0 = 0;
    else
        for (i = 0; i < FIFODEPTH; i++)
        for (i = 0; i < FIFODEPTH; i++){
            //pragma HLS PIPELINE II=1
            r[0] = h[i] & w[i] & a[0];
            EntriesToCheck0 = r;
        }
}
void load_queue::prc_Comparators0() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFODEPTH> w = EntriesToCheck0.read();
    sc_uint<FIFODEPTH> a = ShiftedAdrDone.read();
    sc_uint<FIFODEPTH> s;
    sc_bigtUInt<ADDRESS_SIZE> addrq[4];
    addrq[0] = AddrQ[0].read();
    if(rst == 1)
        Conflict0 = 0;
    else
        for (i = 0; i < FIFODEPTH; i++)
        for (i = 0; i < FIFODEPTH; i++)
            //pragma HLS PIPELINE II=1
            if((addrq[0] == h) & (w[i] == 1) & (a[i] == 1))
                s[1] = 1;
            else
                s[1] = 0;
            Conflict0 = s;
}
void load_queue::prc_FindLastConflict0() {
    //pragma HLS DATAFLOW
    bool found;
    int i;
    sc_uint<FIFODEPTH> s = Conflict0.read();
    sc_uint<FIFODEPTH> r;
    sc_uint<FIFODEPTH> h = ShiftedDataDone.read();
    sc_uint<FIFODEPTH> c;
    if(rst == 1)
```
found = 0;
LastConflict0 = 0;
CanBypass0 = "1111";
}
else
for (i = 0; i < FIFO_DEPTH; i++){
  if ((s[FIFO_DEPTH−1−i] == 1) & (found == 0)){
    found = 1;
    r[FIFO_DEPTH−1−i] = 1;
    if (h[FIFO_DEPTH−1−i] == 1)
      c[FIFO_DEPTH−1−i] = 1;
    else
      c[FIFODEPTH−1−i] = 0;
  }
else
  r[FIFODEPTH−1−i] = 0;
}
LastConflict0 = r;
CanBypass0 = c;
}
}

void load_queue :: prec_Empties0_proc()
{
  int i;
  sc_uint<FIFODEPTH> s = ShiftedAdrDone.read();
  sc_uint<FIFODEPTH> d = EntriesToCheck0.read();
  sc_uint<FIFODEPTH> empties0;
  if (rst == 1)
    Empties0 = 0;
  else
    for (i = 0; i < FIFODEPTH; i++){
      //pragma HLS PIPELINE II=1
      if ((s[i] == 0) & (d[i] == 1))
        empties0[i] = 1;
      else
        empties0[i] = 0;
    }
  Empties0 = empties0;
}

void load_queue :: prec_Decision0()
{
  sc_uint<FIFODEPTH> s = AdrDoneBits.read();
  sc_uint<FIFODEPTH> r = DataDoneBits.read();
  sc_uint<FIFODEPTH> h = LastConflict0.read();
  sc_uint<FIFODEPTH> p = PendingBits.read();
  sc_uint<FIFODEPTH> a = PriorityOut.read();
  sc_uint<FIFODEPTH> b = Empties0.read();
  sc_uint<FIFODEPTH> d = BypassIndex0.read();
  if (rst == 1){
    BypassIndex0 = "0000";
    BypassRequest0 = 0;
    LoadRequest0 = 0;
  }
  bool temp = BypassRequest0;
  if ((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (a[0] == 0)) {
    if ((h == k) & (h.to_int() > 0) & (b.to_int() < h.to_int())
      BypassRequest0 = 1;
    else
      BypassRequest0 = 0;
    if ((b == 0) & (h == 0))
LoadRequest0 = 1;
else
    LoadRequest0 = 0;
}
d = k;
BypassIndex0 = d;
}

void load_queue :: id_1399()
{
    sc_uint<FIFO_DEPTH> r;
    if (rst == 1)
        LoadRequests0 = 0;
    else
        r = LoadRequests.read();
    r[0] = LoadRequest0;
    LoadRequests0 = r[0];
}

void load_queue :: prc_ShiftLoadTails1_proc()
{
    #pragma HLS DATAFLOW
    int i;
    int current_tail;
    int store_upper_bound;
    sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
    sc_uint<2> tailq = TailQ[1].read();
    sc_uint<FIFO_DEPTH> r;
    if (rst == 1)
        LoadTailsBits1 = 0;
    else
        current_tail = tailq.to_uint();
    if (((current_tail - 1) % FIFO_DEPTH) < storehead)
        store_upper_bound = current_tail + FIFO_DEPTH;
    else if (((current_tail - 1) % FIFO_DEPTH) == storehead) & StoreEmpty
        store_upper_bound = current_tail;
    else
        store_upper_bound = current_tail;
    for (i = 0; i < FIFO_DEPTH; i++)
        if ((i + storehead) < store_upper_bound)
            r[0] = 1;
        else
            r[0] = 0;
    LoadTailsBits1 = r;
}

void load_queue :: prc_EntriesToCheck1_proc()
{
    #pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> r;
    sc_uint<FIFO_DEPTH> h = LoadTailsBits1.read();
    sc_uint<FIFO_DEPTH> w = ValidEntries.read();
    sc_uint<FIFO_DEPTH> a = CheckBits.read();
    if (rst == 1)
        EntriesToCheck1 = 0;
    else
        for (i = 0; i < FIFO_DEPTH; i++)
            r[0] = h[i] & w[i] & a[0];
    EntriesToCheck1 = r;
}

void load_queue :: prc_Comparators1()
{
    #pragma HLS DATAFLOW
    int i;
}
sc_uint<\texttt{FIFO\_DEPTH}> w = EntriesToCheck1.read();
sc_uint<\texttt{FIFO\_DEPTH}> a = ShiftedAdrDone.read();
sc_uint<\texttt{FIFO\_DEPTH}> s;
sc_biguint<\texttt{ADDRESS\_SIZE}> addrq1 = AddrQ[1].read();
if (\texttt{rst} == 1)
  Conflict1 = 0;
else
  for (\texttt{i=0};\texttt{i<\texttt{FIFO\_DEPTH}};\texttt{i++}){
    sc_biguint<\texttt{FIFO\_DEPTH}> h = ShiftedStoreAddrQ[\texttt{i}].read();
    if ((addrq1 == h) & (w[\texttt{i}] == 1) & (a[\texttt{i}] == 1))
      s[\texttt{i}] = 1;
    else
      s[\texttt{i}] = 0;
    Conflict1 = s;
  }
void load_queue::prc_FindLastConflict1(){
  //\texttt{pragma HLS\_DATAFLOW}\n  \texttt{bool found;}\n  \texttt{int i;}\n  sc_uint<\texttt{FIFO\_DEPTH}> s = Conflict1.read();
  sc_uint<\texttt{FIFO\_DEPTH}> r;
  sc_uint<\texttt{FIFO\_DEPTH}> h = ShiftedDataDone.read();
  sc_uint<\texttt{FIFO\_DEPTH}> c;
  if (\texttt{rst} == 1){
    \texttt{found} = 0;
    \texttt{LastConflict1} = 0;
    \texttt{CanBypass1} = "1111";
  }\n  else
    for (\texttt{i=0};\texttt{i<\texttt{FIFO\_DEPTH}};\texttt{i++}){
      \texttt{if} ((s[\texttt{FIFO\_DEPTH}−1−\texttt{i}] == 1) & (\texttt{found} == 0)){
        \texttt{found} = 1;
        \texttt{r}[\texttt{FIFO\_DEPTH}−1−\texttt{i}] = 1;
        \texttt{if} (\texttt{h}[\texttt{FIFO\_DEPTH}−1−\texttt{i}] == 1)
          \texttt{c}[\texttt{FIFO\_DEPTH}−1−\texttt{i}] = 1;
        else
          \texttt{c}[\texttt{FIFO\_DEPTH}−1−\texttt{i}] = 0;
      }
    }\n    else
      \texttt{r}[\texttt{FIFO\_DEPTH}−1−\texttt{i}] = 0;
    CanBypass1 = \texttt{c};
    \texttt{LastConflict1} = \texttt{r};
  }
void load_queue::prc_Empties1_proc(){
  //\texttt{pragma HLS\_DATAFLOW}\n  \texttt{int i;}\n  sc_uint<\texttt{FIFO\_DEPTH}> s = ShiftedAdrDone.read();
  sc_uint<\texttt{FIFO\_DEPTH}> d = EntriesToCheck1.read();
  sc_uint<\texttt{FIFO\_DEPTH}> empties1 = Empties1.read();
  if (\texttt{rst} == 1)
    Empties1 = 0;
  else
    for (\texttt{i=0};\texttt{i<\texttt{FIFO\_DEPTH}};\texttt{i++}){
      //\texttt{pragma HLS\_PIPELINE II=1}\n      \texttt{if} ((\texttt{s[i]} == 0) & (\texttt{d[i]} == 1))
        empties1[\texttt{i}] = 1;
      else
        empties1[\texttt{i}] = 0;
    }\n    Empties1 = empties1;
```cpp
void load_queue::prc_Deco1()
{
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> s = AdrDoneBits.read();
    sc_uint<FIFODEPTH> r = DataDoneBits.read();
    sc_uint<FIFODEPTH> h = LastConflict1.read();
    sc_uint<FIFODEPTH> k = CanBypass1.read();
    sc_uint<FIFODEPTH> p = PendingBits.read();
    sc_uint<FIFODEPTH> a = PriorityOut.read();
    sc_uint<FIFODEPTH> b = Empties1.read();
    sc_uint<FIFODEPTH> d;
    if (rst == 1)
    {
        BypassIndex1 = "0000";
        BypassRequest1 = 0;
        LoadRequest1 = 0;
    }
    bool temp = BypassRequest1;
    if ((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (a[0] == 0))
    {
        if ((h == k) & (h > 0) & (b < h))
            BypassRequest1 = 1;
        else
            BypassRequest1 = 0;
        if ((b == 0)&(h == 0))
            LoadRequest1 = 1;
        else
            LoadRequest1 = 0;
    }
    d = k;
    BypassIndex1 = d;
}

void load_queue::id_1515()
{
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> r ;
    if (rst ==1)
        LoadRequests1 = 0;
    else
    
        r = LoadRequests.read();
        r[1] = LoadRequest1;
        LoadRequests1 = r[1];
}

void load_queue::prc_ShiftLoadTails2_proc()
{
    //pragma HLS DATAFLOW
    int i;
    int current_tail;
    int store_upper_bound;
    sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
    sc_uint<2> tailq = TailQ[2].read();
    sc_uint<FIFO_DEPTH> r;
    if (rst ==1)
        LoadTailsBits2 = 0;
    else
        current_tail = tailq.to_uint();
    if (((current_tail - 1) % FIFODEPTH) < storehead)
        store_upper_bound = current_tail + FIFODEPTH;
    else if (((current_tail - 1) % FIFODEPTH) == storehead) & StoreEmpty
        store_upper_bound = current_tail;
    else
        store_upper_bound = current_tail;
    for (i=0;i<FIFODEPTH;i++)
    {
    }
```

```c
//#pragma HLS PIPELINE II=1
if ((i + storehead) < store_upper_bound)
  r[i] = 1;
else
  r[i] = 0;
LoadTailsBits2 = r;
}

void load_queue::prc_EntriesToCheck2_proc()
{
  int i;
  sc_uint<FIFODEPTH> r;
  sc_uint<FIFODEPTH> h = LoadTailsBits2.read();
  sc_uint<FIFODEPTH> w = ValidEntries.read();
  sc_uint<FIFODEPTH> a = CheckBits.read();
  if (rst == 1)
    EntriesToCheck2 = 0;
  else
    for (i=0; i<FIFODEPTH; i++)
      r[i] = h[i] & w[i] & a[0];
  EntriesToCheck2 = r;
}

void load_queue::prc_Comparators2()
{
  int i;
  sc_uint<FIFODEPTH> w = EntriesToCheck2.read();
  sc_uint<FIFODEPTH> a = ShiftedAddrDone.read();
  sc_uint<FIFODEPTH> s = Conflict2.read();
  sc_biguint<ADDRESS_SIZE> addrq2 = AddrQ[2].read();
  if (rst == 1)
    Conflict2 = 0;
  else
    for (i=0; i<FIFODEPTH; i++)
      h = ShiftedStoreAddrQ[i].read();
      if (addrq2 == h) & (w[i] == 1) & (a[i] == 1))
        s[i] = 1;
      else
        s[i] = 0;
    Conflict2 = s;
}

void load_queue::prc_FindLastConflict2()
{  
  //pragma HLS DATAFLOW
  bool found;
  int i;
  sc_uint<FIFODEPTH> s = Conflict2.read();
  sc_uint<FIFODEPTH> r;
  sc_uint<FIFODEPTH> h = ShiftedDataDone.read();
  sc_uint<FIFODEPTH> c;
  if (rst == 1)
    found = 0;
  LastConflict2 = 0;
  CanBypass2 = "1111";
  } 
  else
    for (i=0; i<FIFODEPTH; i++)
      if ((s[FIFODEPTH−1−i] == 1) & (found == 0))
        found = 1;
      r[FIFODEPTH−1−i] = 1;
      if (h[FIFODEPTH−1−i] == 1)
        c[FIFODEPTH−1−i] = 1;
    } 
```
```c
if (!rst)
    { else
        c[FIFODEPTH−1−i] = 0;
    }
else
    r[FIFODEPTH−1−i] = 0;

if (!rst)
    { LastConflict2 = r;
        CanBypass2 = c;
    }
else
    { FIFODEPTH−1−i] = 0;
    }

void load_queue :: prc_Empties2_proc()
{//#pragma HLS DATAFLOW
int i;
sc_uint<FIFODEPTH> s = ShiftedAadrDone.read();
sc_uint<FIFODEPTH> d = EntriesToCheck2.read();
sc_uint<FIFODEPTH> empties0;
if (rst == 1)
    { Empties2 = 0;
    }
else
    { for (i=0;i<FIFODEPTH;i++)
        { if ((s[i] == 0) & (d[i] == 1))
            empties0[i] = 1;
        }
    }
Empties2 = empties0;
}
}

void load_queue :: prc_Decision2()
{//#pragma HLS DATAFLOW
sc_uint<FIFODEPTH> s = AdrDoneBits.read();
sc_uint<FIFODEPTH> r = DataDoneBits.read();
sc_uint<FIFODEPTH> h = LastConflict2.read();
sc_uint<FIFODEPTH> k = CanBypass2.read();
sc_uint<FIFODEPTH> p = PendingBits.read();
sc_uint<FIFODEPTH> a = PriorityOut.read();
sc_uint<FIFODEPTH> b = Empties2.read();
sc_uint<FIFODEPTH> d;
if (rst == 1)
    { BypassIndex2 = "0000";
    BypassRequest2 = 0;
    LoadRequest2 = 0;
    }
bool temp = BypassRequest2;
if ((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (a[0] == 0))
    { if ((h == k) & (h > 0) & (b < h))
        BypassRequest2 = 1;
    else
        BypassRequest2 = 0;
    if ((b == 0) & (h == 0))
        LoadRequest2 = 1;
    else
        LoadRequest2 = 0;
    }
    d = k;
    BypassIndex2 = d;
}
void load_queue :: id_1631()
{//#pragma HLS DATAFLOW
sc_uint<FIFODEPTH> r;
if (rst == 1)
    { LoadRequests2 = 0;
    }
else
```
```c
r = LoadRequests.read();
r[2] = LoadRequest2;
LoadRequests2 = r[2];
}
void load_queue::proc_ShiftLoadTails3()
{
    //pragma HLS DATAFLOW
    int i;
    int current_tail;
    int store_upper_bound;
    sc_uint<FIFO_DEPTH> storehead = StoreHead.read();
    sc_uint<2> tailq = TailQ[3].read();
    sc_uint<FIFO_DEPTH> r = LoadTailsBits3.read();
    if(rst == 1)
        LoadTailsBits3 = 0;
    else
        current_tail = tailq.to_uint();
    if(((current_tail - 1) % FIFO_DEPTH) < storehead)
        store_upper_bound = current_tail + FIFO_DEPTH;
    else if(((current_tail - 1) % FIFO_DEPTH) == storehead) &
        StoreEmpty)
        store_upper_bound = current_tail;
    else
        store_upper_bound = current_tail;
    for(i=0;i<FIFO_DEPTH;i++)
        //pragma HLS PIPELINE II=1
        if(((i + storehead) < store_upper_bound)
            r[i] = 1;
        else
            r[i] = 0;
        LoadTailsBits3 = r;
}
void load_queue::proc_EntriesToCheck3()
{
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> r;
    sc_uint<FIFO_DEPTH> h = LoadTailsBits3.read();
    sc_uint<FIFO_DEPTH> w = ValidEntries.read();
    sc_uint<FIFO_DEPTH> a = CheckBits.read();
    if(rst == 1)
        EntriesToCheck3 = 0;
    else
        for(i=0;i<FIFO_DEPTH;i++)
            r[0] = h[i] & w[i] & a[0];
        EntriesToCheck3 = r;
}
void load_queue::proc_Comparators3()
{
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> w = EntriesToCheck3.read();
    sc_uint<FIFO_DEPTH> a = ShiftedAddrDone.read();
    sc_uint<FIFO_DEPTH> s;
    sc_biguint<ADDRESS_SIZE> addrq3 = AddrQ[3].read();
    if(rst == 1)
        Conflict3 = 0;
    else
        for(i=0;i<FIFO_DEPTH;i++)
            //pragma HLS PIPELINE II=1
            sc_biguint<FIFO_DEPTH> h = ShiftedStoreAddrQ[i].read();
            if((addrq3 == h) & (w[i] == 1) & (a[i] == 1))
                s[i] = 1;
}
```cpp
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<FIFODEPTH> s = Conflict3.read();
758   sc_uint<FIFODEPTH> r;
759   sc_uint<FIFODEPTH> h = ShiftedDataDone.read();
760   sc_uint<FIFODEPTH> c;
761   if (rst == 1) {
762     found = 0;
763     LastConflict3 = 0;
764     CanBypass3 = "1111";
765   } 
766   else
767     for (i = 0; i < FIFODEPTH; i++) {
768       //#pragma HLS PIPELINE II=1
769       if ((s[FIFODEPTH-1-i] == 1) & (found == 0)) {
770         found = 1;
771         r[FIFODEPTH-1-i] = 1;
772         if (h[FIFODEPTH-1-i] == 1)
773           c[FIFODEPTH-1-i] = 1;
774         else
775           c[FIFODEPTH-1-i] = 0;
776       } 
777       else
778         r[FIFODEPTH-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<FIFODEPTH> s = ShiftedAdrDone.read();
787   sc_uint<FIFODEPTH> d = EntriesToCheck3.read();
788   sc_uint<FIFODEPTH> empties0 = Empties3.read();
789   if (rst == 1) 
790     Empties3 = 0;
791   else
792     for (i = 0; i < FIFODEPTH; 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 {//#pragma HLS DATAFLOW
803   sc_uint<FIFODEPTH> s = AdrDoneBits.read();
804   sc_uint<FIFODEPTH> r = DataDoneBits.read();
805   sc_uint<FIFODEPTH> h = LastConflict3.read();
806   sc_uint<FIFODEPTH> k = CanBypass3.read();
807   sc_uint<FIFODEPTH> p = PendingBits.read();
808   sc_uint<FIFODEPTH> a = PriorityOut.read();
809   sc_uint<FIFODEPTH> b = Empties3.read();
```
```cpp
sc_uint<FIFO_DEPTH> d;
if (rst == 1) {
    BypassIndex3 = "0000";
    BypassRequest3 = 0;
    LoadRequest3 = 0;
}

bool temp = BypassRequest3;
if ((s[0] == 1) & (r[0] == 0) & (p[0] == 0) & (temp == 0) & (a[0] == 0)) {
    if ((h == k) & (h > 0) & (b < h))
        BypassRequest3 = 1;
    else
        BypassRequest3 = 0;
    if ((b == 0) & (h == 0))
        LoadRequest3 = 1;
    else
        LoadRequest3 = 0;
}

d = k;
BypassIndex3 = d;
}

void load_queue::id_1746() {
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> r;
    if (rst == 1)
        LoadRequests3 = 0;
    else
        r = LoadRequests.read();
    r[3] = LoadRequest3;
    LoadRequests3 = r[3];
}

void load_queue::proc_ChOOSE_MEM_REQUEST_PRC() {
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> r = PriorityOut.read();
    sc_biguint<ADDRESS_SIZE> addrq1 = AddrQ[1].read();
    MemoryLoadEnable = 0;
    if (r[0] == 1)
        MemoryLoadAddress = AddrQ[0].read();
    MemoryLoadEnable = 1;
}
else if (r[1] == 1) {
    MemoryLoadAddress = addrq1;
    MemoryLoadEnable = 1;
}
else if (r[2] == 1) {
    MemoryLoadAddress = AddrQ[2].read();
    MemoryLoadEnable = 1;
}
else if (r[3] == 1) {
    MemoryLoadAddress = AddrQ[3].read();
    MemoryLoadEnable = 1;
}
else
    MemoryLoadAddress = "0000000000";
}

void load_queue::proc_DATADATA_DONE_PRC() {
    //pragma HLS DATAFLOW
    sc_uint<FIFO_DEPTH> r = InitBits.read();
    sc_uint<FIFO_DEPTH> q = PendingBits.read();
```
if (rst == 1)
   DataDoneBits = "0000";
else{
   if (\[0\] == 1)
      h[0] = 0;
   else if (q[0] == 1)
      h[0] = 1;
   else if (BypassRequest0 == 1)
      h[0] = 1;
   if (r[1] == 1){
      h[1] = 0;
   } else if (q[1] == 1)
      h[1] = 1;
   else if (BypassRequest1 == 1)
      h[1] = 1;
   if (r[2] == 1){
      h[2] = 0;
   } else if (q[2] == 1)
      h[2] = 1;
   else if (BypassRequest2 == 1)
      h[2] = 1;
   if (r[3] == 1){
      h[3] = 0;
   } else if (q[3] == 1)
      h[3] = 1;
   else if (BypassRequest3 == 1)
      h[3] = 1;
   DataDoneBits = h;
}
void load_queue::proc_DataQUpdate_proc0(){
   //pragma HLS DATAFLOW
   int i;
   sc_uint<\text{FIFO\_DEPTH}> r = PriorityOut.read();
   sc_uint<\text{FIFO\_DEPTH}> d = BypassIndex0.read();
   sc_biguint<\text{DATA\_SIZE}> h = ShiftedStoreDataQ2[i].read();
   sc_biguint<\text{DATA\_SIZE}> q = PendingBits.read();
   sc_biguint<\text{DATA\_SIZE}> data0 = Data0.read();
   if (rst == 1)
      Data0 = 0;
   else if (BypassRequest0 == 1)
      for (i = 0; i < \text{FIFO\_DEPTH}; i++){
         //pragma HLS PIPELINE II=1
         if (d[i] == 1){
            sc_biguint<\text{DATA\_SIZE}> h = ShiftedStoreDataQ2[i].read();
            data0 = h;
         }
      }
   else if (q[0] == 1)
      data0 = MemoryLoadData.read();
   Data0 = data0;
}
void load_queue::id_1829(){
   //pragma HLS DATAFLOW
   if (rst == 1){
      DataQ[0] = 0;
   }
}
```c
else {
    sc_biguint<DATA_SIZE> tmp = Data0.read();
    DataQ[0] = tmp;
}
}

void load_queue::pre_DataQUpdate_proc1() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> r = PriorityOut.read();
    sc_uint<FIFODEPTH> d = BypassIndex1.read();
    sc_uint<FIFODEPTH> q = PendingBits.read();
    sc_uint<DATA_SIZE> data1;
    if(rst == 1)
        Data1 = 0;
    else if(BypassRequest1 == 1)
        for( i=0; i<FIFODEPTH; i++ ){
            if(d[i] == 1){
                sc_uint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
                data1 = h;
            }
        } else if(q[1] == 1)
        data1 = MemoryLoadData.read();
    Data1 = data1;
}

void load_queue::id1847() {
    //pragma HLS DATAFLOW
    sc_uint<DATA_SIZE> tmp = Data1.read();
    if(rst == 1){
        DataQ[1] = 0;
    }
    DataQ[1] = tmp;
}

void load_queue::pre_DataQUpdate_proc2() {
    //pragma HLS DATAFLOW
    int i;
    sc_uint<FIFO_DEPTH> r = PriorityOut.read();
    sc_uint<FIFODEPTH> d = BypassIndex2.read();
    sc_uint<FIFODEPTH> q = PendingBits.read();
    sc_uint<DATA_SIZE> data2;
    if(rst == 1)
        Data2 = 0;
    else if(BypassRequest2 == 1)
        for( i=0; i<FIFODEPTH; i++ ){
            if(d[i] == 1){
                sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
                data2 = h;
            }
        } else if(q[2] == 1)
        data2 = MemoryLoadData.read();
    Data2 = data2;
}

void load_queue::id1865() {
    //pragma HLS DATAFLOW
    sc_uint<DATA_SIZE> tmp = Data2.read();
    if(rst == 1){
        DataQ[2] = 0;
    }
}
```
```c

void load_queue :: proc_DataQUpdate_proc3()
{//pragma HLS DATAFLOW
  int 1;
  sc_uint<_FIFODEPTH> r = PriorityOut.read();
  sc_uint<_FIFODEPTH> d = BypassIndex3.read();
  sc_uint<_FIFODEPTH> q = PendingBits.read();
  sc_uint<DATA_SIZE> data3;
  if(rst == 1)
    Data3 = 0;
  else if(BypassRequest3)
    for (i = 0; i < FIFO_DEPTH; i++)
      //pragma HLS PIPELINE II=1
      if(d[i] == 1)
        sc_biguint<DATA_SIZE> h = ShiftedStoreDataQ2[i].read();
      data3 = h;
  else if(q[3] == 1)
    data3 = MemoryLoadData.read();
  Data3 = data3;
}

void load_queue :: id_1883()
{//pragma HLS DATAFLOW
  sc_uint<DATA_SIZE> tmp = Data3.read();
  if(rst == 1)
    DataQ[3] = 0;
}

void load_queue :: proc_SearchPortQP0_proc()
{//pragma HLS DATAFLOW
  int 1;
  sc_uint<_FIFODEPTH> r;
  if(rst == 1)
    EntriesPort0 = 0;
  else
    for (i = 0; i < FIFO_DEPTH; i++)
      sc_uint<1> portq = PortQ[i].read();
      if(portq == 0)
        r[i] = 1;
      else
        r[i] = 0;
    EntriesPort0 = r;
}

void load_queue :: proc_InputPriorityP0_proc()
{//pragma HLS DATAFLOW
  bool blocked;
  int 1;
  sc_uint<_FIFODEPTH> head = Head.read();
  sc_uint<_FIFODEPTH> r = EntriesPort0.read();
  sc_uint<_FIFODEPTH> b = AdrDoneBits.read();
  if(rst == 1)
```c
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 void load_queue::prc_OutputPriorityPort0_proc(){
1066  bool blocked;
1067  int i;
1068  sc_uint<FIFO_DEPTH> head = Head.read();
1069  sc_uint<FIFO_DEPTH> r = EntriesPort0.read();
1070  sc_uint<FIFO_DEPTH> h;
1071  sc_uint<FIFO_DEPTH> b = AdrDoneBits.read();
1072  if (rst == 1){
1073    blocked = 0;
1074    OutputPriorityPort0 = "0000";
1075  }
1076  for (i = 0; i < FIFO_DEPTH; i++){
1077    //pragma HLS PIPELINE II=1
1078    if ((r[(i + head) % FIFO_DEPTH] == 1) & (b[(i+head) & FIFO_DEPTH] == 0)){
1079      if (blocked == 0){
1080        h[(i+head) % FIFO_DEPTH] = 1;
1081        blocked = 1;
1082        OutputPriorityPort0 = h;
1083      }
1084    }
1085  }
1086 void load_queue::prc_SendToPort0_proc(){
1087  bool send0;
1088  int i;
1089  sc_uint<FIFO_DEPTH> h = OutputPriorityPort0.read();
1090  sc_uint<FIFO_DEPTH> datadonebits = DataDoneBits.read();
1091  sc_uint<FIFO_DEPTH> r = AllDoneBits.read();
1092  sc_uint<DATA_SIZE> dataq[4];
1093  bool temp = Port0_Ready.read();
1094  if (rst == 1)
1095    Port0_Valid = 0;
1096  else
1097    send0 = 0;
1098  for (i = 0; i < FIFO_DEPTH; i++){
1099    dataq[i] = DataQ[i].read();
1100    if ((h[i] == 1) & (datadonebits[i] == 1) & (r[i] == 0) & (temp == 1))
1101      Port0_DataOut.write(dataq[i]);
1102    send0 = 1;
1103  }
1104  if (send0 == 1)
1105    Port0_Valid = 1;
1106  else if (Port0_Ready == 1)
1107    Port0_Valid = 1;
1108 }
1109 void load_queue::prc_UpdateAddressBits(){
1110  //pragma HLS DATAFLOW
1111 ```
int i;
sc_uint<FIFO_DEPTH> b;
sc_uint<FIFO_DEPTH> h = InputPriorityPort0.read();
sc_uint<FIFO_DEPTH> r = InitBits.read();
sc_uint<ADDRESS_SIZE> port0_addrin = Port0_AddrIn.read();
sc_bigruint<ADDRESS_SIZE> addrq[4];
if(rst == 1)
  AdrDoneBits = "0000";
else
  for (i=0;i<FIFO_DEPTH;i++)
    // #pragma HLS PIPELINE II=1
    if (r[i] == 1)
      b[i] = 0;
    else if ((h[i] == 1) & Port0_LoadWriteEn)
      addrq[i] = port0_addrin;
      AddrQ[i] = addrq[i];
      b[i] = 0;
      AdrDoneBits = b;
}
void load_queue:: prec_UpdateAllDoneBits_proc() {
  // #pragma HLS DATAFLOW
  int i;
  sc_uint<FIFO_DEPTH> r = InitBits.read();
  sc_uint<FIFO_DEPTH> h = OutputPriorityPort0.read();
  sc_uint<FIFO_DEPTH> a = AllDoneBits.read();
  sc_uint<FIFO_DEPTH> d = DataDoneBits.read();
  sc_uint<FIFO_DEPTH> sending;
  sc_uint<FIFO_DEPTH> head = DataIn.read();
  sc_uint<FIFO_DEPTH> tail = Tail.read();
  bool temp = Port0_Ready;
  sending = "0000";
  if (rst == 1){
    AllDoneBits = "0000";
    Head.write(0);
  } else
  for (i=0;i<FIFO_DEPTH;i++)
    // #pragma HLS PIPELINE II=1
    if (r[i] == 1)
      a[i] = 0;
    else if ((h[i] == 1)&(d[i] == 1)&(r[i] == 0)&(temp == 1))
      AllDoneBits = a;
      sending[i] = 1;
    if (((head == 1) | (sending[head] == 1))
      if ((((head + 1) % FIFO_DEPTH)! = tail){
        if (head == FIFO_DEPTH - 1){
          head = 0;
        }
      } else{
        head = head + 1;
      }
    }
  }
  Head.write(head);
Bibliography