I, Bhargav Bappudi, hereby submit this original work as part of the requirements for the degree of Master of Science in Electrical Engineering.

It is entitled:
Example modules for hardware-software co-design

Student's name: Bhargav Bappudi

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.

Committee member: Wen-Ben Jone, Ph.D.

Committee member: George Purdy, Ph.D.
Example Modules for Hardware-Software Co-Design

A thesis submitted to the Graduate School

of

The University of Cincinnati

In partial fulfilment of the requirements for the degree of

Master of Science

In the

Department of Electrical Engineering and Computing Systems

of the

College of Engineering and Applied Sciences

July 2016

by

Bhargav Bappudi

B.E. Electrical and Electronics Engineering

Anna University, 2013

Committee Chair: Dr. Carla Purdy
Abstract

Embedded systems have found applications in many fields including consumer electronics, medical devices, defense technology systems, and telecommunications. With advances in technology, newer electronic devices require high speed and durability and modern embedded systems need to be designed to meet these demanding requirements. Recent changes in embedded system architectures have incorporated both hardware and software components. Traditionally, designs for hardware and software were developed separately in the early stages of the co-design process. Because of recent developments in the hardware software co-design approach, SoC designs can now implement hardware dependent software and software dependent hardware.

The major objective of this thesis is to provide an example of hardware and software implementations of a specific module, a priority queue, to enable students of embedded system design to compare different approaches to developing a hardware/software design of this commonly used data structure and to customize priority queue designs for specific applications. A second objective is to discuss and compare the different verification methods currently available for hardware and software modules and to clarify the advantages of formal verification, which is becoming a standard verification method for hardware modules.

*Keywords:* Hardware software co-design, hardware/software partitioning, SoC Design, embedded system, priority queue.
Table of Contents

1 INTRODUCTION...........................................................................................................................................1

1.1 Problem...............................................................................................................................................1

1.2 Objective............................................................................................................................................3

1.3 Approach............................................................................................................................................3

1.4 Organization......................................................................................................................................4

2 BACKGROUND AND RELATED WORK.................................................................................................5

2.1 Embedded systems implemented with modern SoCs.........................................................................5

2.2 Need for hardware software co-design methodology.......................................................................6

2.3 Tasks involved in hardware software co-design methodology.........................................................8

2.4 Related work....................................................................................................................................9

2.5 Hardware testing/verification methods vs software testing/verification methods..............................15

   2.5.1 Hardware testing/verification methods.......................................................................................15

   2.5.2 Example of hardware design verification method: Formal verification.................................16

   2.5.3 Software testing methods.............................................................................................................18

   2.5.4 Example of a white box testing method: basis path method....................................................21

3 APPROACH............................................................................................................................................23

3.1 Basic concepts..................................................................................................................................23

3.2 Hardware implementation of the priority queue..............................................................................23

3.3 Software implementation of the priority queue................................................................................25

3.4 Operation of the co-design controller module..................................................................................28

   3.4.1 Pseudocode for the co-design decision controller.................................................................30
4 RESULTS AND ANALYSIS ......................................................................................................................32

4.1 Hardware implementation results ..................................................................................................32

4.1.1 Hardware implementation of priority queue ...........................................................................34

4.2 Software implementation results ..................................................................................................42

4.2.1 Software implementation of priority queue ...........................................................................42

4.3 Hardware-software co-design .....................................................................................................48

4.4 Co-design controller module .......................................................................................................49

4.5 Computation time for the hardware and software modules .........................................................55

5 CONCLUSION AND FUTURE WORK ............................................................................................56

5.1 Summary of work .........................................................................................................................56

5.2 Future work ....................................................................................................................................57

APPENDIX – Hardware verification of four-bit adder ........................................................................58

REFERENCES .........................................................................................................................................64
List of Figures

Figure 2.1: Basic structure of an embedded system [6] .................................................................5

Figure 2.2: (a) Existing interface design between hardware and software modules (b) hardware software module interaction in the newer methodology of co-design [4]........................................7

Figure 2.3: Basic flow of the partitioning process [3] .................................................................9

Figure 2.4: A high level block diagram of the hardware-based priority queue [10]............................10

Figure 2.5: Pseudocode of hybrid priority queue’s enqueue operation [10].....................................11

Figure 2.6: Pseudocode of hybrid priority queue’s dequeue operation [10].....................................11

Figure 2.7: Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue [10] .................................................12

Figure 2.8: Binary heap structure of priority queue in software implementation [11]....................13

Figure 2.9: Hardware interface [11] .............................................................................................13

Figure 2.10: Hardware realization of priority queue [11] ............................................................14

Figure 2.11: Hardware-software interface [11] ..........................................................................14

Figure 2.12: Formal verification in ASIC design flow [11] ..........................................................17

Figure 2.13: Types of testing in software [15] ..........................................................................19

Figure 3.1: Hardware queue [11] ............................................................................................24

Figure 3.2: Hardware realization [11] .......................................................................................25

Figure 3.3: Example implementation of software priority queue, (a) Element 1 inserted in the bottom child node (b) Element 1 swapped with its parent node element 8 (c) Element 1 swapped with its parent node element 4 (d) Element 1 swapped with element 2 to maintain balanced tree........................................27

Figure 3.4 Co-design controller module....................................................................................29

Figure 4.1: Output of the priority queue hardware implementation............................................41

Figure 4.2: Output of the priority queue hardware implementation............................................47

Figure 4.3: Output of the manager module..............................................................................52
## List of Tables

Table 4.1: Distribution of insertions into segments (%) [11] .................................................................33

Table 4.2: Insertion rate into first five positions [11] ..................................................................................33

Table 4.3: Priority queue operations in software [11] ..................................................................................42

Table 4.4: Successive insert and remove operations (%) [11] .................................................................48

Table 4.5: Different operations performed on the priority queues and computation time required ... 54
1. INTRODUCTION

Electronic devices have become an indispensable part of our everyday life. With the rise of virtual reality and 3D devices, it is now hard to imagine a life without electronic devices. These new devices demand high computing power, high efficiency, low power consumption, more memory and more reliability, thereby making designs more complex [1]. Many such devices are being used in embedded systems as they find applications in numerous fields from safety critical systems to the most advanced designs in the technological world.

An embedded system is an electronic system that has a programming module which is embedded in computer hardware [2]. Hardware and software are designed and combined on a processor to implement specific functions. System-on-a-Chip (SoC) consists of the critical elements in an electronic system on a single microchip [2]. It is widely used across the embedded industry due to its small-form-factor, computational excellence and low power consumption. In order to achieve a faster computational device, a new design approach – hardware-software co-design – has been introduced in modern embedded systems.

1.1 PROBLEM

Traditionally in embedded systems design, the design of hardware and software was done separately in the early stages. Hardware systems generally are robust in design and have a better performance but designing complex systems can be challenging. This is because, as the technology node keeps shrinking, it is becoming more and more difficult to build complex designs in hardware. Software, on the other hand, provides flexibility in terms of developing complex designs as compared to hardware but
producing higher quality code, fixing errors and optimizing the data structures is very essential. Hardware software co-design approach uses the advantages of hardware, resulting in having a faster device, and also the advantages of software in enabling easier modifications for the complex devices.

However, achieving higher performance and maintaining reliability simultaneously is difficult. In a hardware software co-design, devices are flexible, but insuring security in the hardware and software components is of high concern, especially since hardware attacks are very common with SOC. These devices must also be robust and have high durability under various working conditions. As a result, a lot of emphasis has been placed on the verification and testing of these components before they are deployed in the market. To maximize efficiency of the hardware development process, it is necessary to detect most of the errors in the design as soon as possible. However, systems are increasingly dependent on software, and maintaining reliability in software intensive systems is very difficult due to the lack of standardized testing methods for software [3].

In addition to the existing hardware verification methods, formal methods of verification have become widely available in recent design systems, thus ensuring the reliability of the design and robustness of the system. Hence in the co-design methodology critical decisions are more safely applied in the hardware design.
1.2 OBJECTIVE

The principal objective of this thesis is to provide a priority queue module that could be implemented as a hardware software design in Altera/Xilinx boards and to provide an overview of the advantages of having a hardware software co-design methodology. Co-design methodology deals with the problem of designing complex embedded systems, where hardware/software partitioning is one key challenge. The choices of hardware design and software type depend on the user requirements, and the hardware software co-design needs to be developed accordingly. A second objective is to discuss and compare the different verification methods currently available for hardware and software modules and to discuss the advantages of formal verification, which is becoming a standard verification method for hardware modules.

1.3 APPROACH

The approach for implementing the priority queue module was to design the hardware as well as the software module which could be implemented as is on Altera/Xilinx boards (both software as well as hardware modules). For the software design, a priority queue was built using C and test bench cases were written to verify its functionality. For the hardware system, we implemented a priority queue module in a Hardware Description Language (Verilog) and verified the functional correctness and robustness using standardized hardware verification methods.

To provide an example to demonstrate equivalence checking for hardware design, a four-bit adder was implemented using RTL Verilog and was tested with test bench cases for its functionality. After verifying the correctness of the functionality, it was designated “my original/golden/reference design”. The RTL
Verilog was modified with a simplified Boolean expression for the four-bit adder, albeit maintaining the functionality. This design was then verified against the golden design for logic equivalence check. The Synopsys Formality tool [12] created compare points and logic cones were formed for both reference and implemented design and were proved to be mapping correctly against each other. This was a test for the RTL – RTL design verification.

The RTL design was then synthesized using 40nm technology node and a gate level netlist was generated. This gate level netlist was verified against the golden RTL design. The Formality tool created compare points and logic cones to map both the designs for design equivalence. This was a test for the RTL – gate level design verification. More details of this work are included in an Appendix.

1.4 ORGANIZATION

This thesis has been organized in the following way: chapter 2 discusses the background and related work; chapter 3 discusses the approach; chapter 4 gives the results and analysis; then, chapter 5 discusses future work and the conclusion.
2. BACKGROUND AND RELATED WORK

This chapter provides the information and concepts to understand this work and related work. Section 2.1 discusses the basic concepts of embedded systems implemented with modern SoCs. Section 2.2 describes the need for hardware-software co-design methodology. Section 2.3 explains the tasks involved in the co-design process. Section 2.4 describes related work pertaining to this thesis. Section 2.5 elaborates on the various types of testing and verification methods used in hardware and software design methods.

2.1 Embedded systems implemented with modern SoCs

An embedded computing system is an application-specific electronic subsystem that is used in a larger system such as a consumer appliance, medical device, or automobile [4]. An embedded system consists of a processor, memory, peripherals and software [5]. Performance of an embedded system depends on the type of processor being used in it and the choice of processor depends on user requirements, speed, cost and several other factors. The memory of the embedded system entirely depends on the application for which the system is developed [5]. Typically, all the intermediate results and status flags are stored in the memory apart from the software. Figure 2.1 shows the basic structure of an embedded system.

![Figure 2.1: Basic structure of an embedded system](image)

<table>
<thead>
<tr>
<th>Application Software Layer (Optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Software Layer (Optional)</td>
</tr>
<tr>
<td>Hardware Layer (Required)</td>
</tr>
</tbody>
</table>
As the silicon technology node size keeps shrinking, it is becoming easier to integrate complex designs of a computer or other electronic system into a Single System on Chip (SoC) [4]. System-on-a-Chip consists of the critical elements in an electronic system on a single microchip [2]. SOC design is not limited to a single task. In addition to specific hardware subsystems, a modern SoC also can include one or several CPU subsystems to execute software [4]. Multimedia platforms such as Nomadik and Nexperia are examples of multiprocessor SoCs that use digital signal processors, microcontrollers, and other kinds of programmable processors [7].

Providing, SoCs consisting of an assembly of processors executing tasks concurrently will require design methodologies to focus on selecting and using either programmable or dedicated processors in place of the gates and arithmetic logic units that current methods use.

2.2 Need for hardware software co-design methodology

In the current SoC designs, software modules can be designed only after the hardware platform design is completed. This often leads to poor hardware designs because problems caught during software development cannot be fixed in the platform [4]. It also means that the design process takes far too long [4]. Hardware software co-design requires both hardware and software models to interact together during the entire design process. The designer needs to develop the hardware and software modules differently because the abstractions needed to describe the interconnections between the software and hardware components are totally different from the existing abstractions of wires between hardware components and function calls that describe software connections [4]. Figure 2.2 describes the existing SoC design methodology where the software module interacts with hardware after the hardware part is designed completely and the newer methodology of co-design where both
hardware and software work modules interact with each other from the beginning of the design process.

![Diagram](image)

**Figure 2.2:** (a) Existing interface design between hardware and software modules (b) hardware software module interaction in the newer methodology of co-design [4].

The semiconductor industry is moving toward a universal chip which can compute all sorts of applications required by the user. If this solution is developed using a standard platform, it is likely to become the preferred option because it would eliminate the cost and effort required to build specific HW/SW platforms. However, due to many factors, modern SoCs require multiple platforms to work on for some specific applications. For example, although AVC/H.264 is a common standard for video compression, different types of video compression systems require different platforms [4]. The video camera of a cellphone requires less computation and fewer frame rates when compared to a high definition video camera, and its architecture is very different as compared to the HD video camera, since it requires low power consumption and needs to be very compact. Hence we could see that this one application needs different platforms to work on and that has been a major motivation for a hardware-software co-design interface [4].
Modern SoC design requires complex software to run on the SoC to meet the user requirements of complex functionality and also to have a short time-to-market window. Developing robust complex software with thousands of lines of code without any bugs is a challenging task. And at the same time scaling down the current ASIC design to achieve a highly parallel multiprocessor SoC is difficult [4]. These constraints demand a new methodology such as hardware-software co-design where the flexibility of software and the higher performance of hardware can be used together to design complex systems.

2.3 Tasks involved in hardware software co-design methodology

Co-design methodology involves five subtasks: modelling, partitioning, scheduling, validation and implementation [8,9].

- Modelling is the step where the system specifications are defined [3]. Better modelling results in the implementation of complex embedded systems with better performance and faster implementation of the circuits.

- In hardware software co-design, partitioning is the most crucial step, as the entire performance and cost of the embedded system depends on this. Depending on the user requirements, the tasks need to be assigned to the software modules or hardware cores. These decisions are carried out in the partitioning process. This step is shown in Figure 2.3.

- After the tasks are partitioned, depending on their priority, the tasks are organized and scheduled to meet their deadlines. However, in order to achieve a better performance for the embedded system, the partitioning and scheduling steps must be done together [3].
The design of the embedded system is validated to prove that the functionality of the system as designed matches what was intended.

The implementation process involves the physical implementation of the hardware cores by synthesizing the blocks and of the software modules by compilation.

2.4 Related work

There have been many research papers published about the advantages of hardware-software co-design implementation on SoCs and embedded systems. One good example is a hardware-software architecture based priority queue that was implemented in real-time applications by Kumar et al. [10]. The software priority queue in this paper was implemented using a binary heap data structure which supports enqueue and dequeue operations in \( O(\log n) \) time. A binary heap has the heap property, i.e., the priority of the root node is always higher than or equal to that of its child nodes [10]. The heap is stored as a linear array in which the first element stored in the array points towards the root node having the highest priority of the queue. Given an index \( i \) of an element, the indices of its left child and right child nodes are \( 2i \) and \( 2i+1 \) respectively [10].
When the size of the queue is less than that of the hardware limit, the priority queue operates in hardware mode and when it exceeds the limit, excess elements are stored in the priority queue in, main memory of the system and the queue is then managed by both hardware and software. Elements of the priority queue that are managed by hardware are memory mapped, providing software with direct access to these elements that are stored in priority-queue-structured on-chip memory [10]. The architecture of this priority queue is shown in Figure 2.4.

![Figure 2.4: A high level block diagram of the hardware-based priority queue [10].](image)

When the priority queue is managed in the hardware mode, enqueue operations take $O(1)$ time and dequeue operations take $O(\log n)$ time [10]. Though the dequeue operations take more time, the element having the highest priority is returned immediately to the first location in the queue and the user does not have to wait till the dequeue operation is be completed to have a sorted queue in order to get the element of the highest priority. Software accesses the hardware priority queue elements via a memory mapped interface as if they resided in main memory [10]. Figure 2.5 gives the pseudocode of the hybrid priority queue’s enqueue operation and Figure 2.6 gives the pseudocode of the dequeue operation.
Figure 2.5: Pseudocode of hybrid priority queue’s enqueue operation [10].

```
1: procedure HYBRID_PQ_ENQUEUE(queue, elem)
2:   if Queue = Full then
3:     throw exception
4: end if
5: Hardware_pq_enqueue(elem)
6: queue.size ++
7: if queue.size > queue.hw_limit then
8:   index = queue.size
9:   Copy overflow hardware element to the end of software queue.
10: queue.data[index] = overflow_cell
11: while index > queue.hw_limit do
12:   if queue.data[index] < queue.data[parent(index)] then
13:     swap_queue_data(queue, index, parent(index))
14:   index = parent(index)
15: end if
16: end while
17: end if
18: end procedure
```

Figure 2.6: Pseudocode of hybrid priority queue’s dequeue operation [10].

```
1: procedure HYBRID_PQ_DEQUEUE(queue)
2:   if Queue = Empty then
3:     throw exception
4: end if
5: result = queue.top;
6: if queue.size < queue.hw_limit then
7:   hardware_pq_dequeue()
8: else
9:   Replace root with last element of heap array.
10: queue.data[0] = queue.data[size]
11: Execute hardware dequeue and return position of newly inserted element.
12: new_index = hardware_pq_dequeue()
13: Continue heap restoration in software from the position returned.
14: Restore_hw_heap(new_index)
15: end if
16: queue.size --;
17: end procedure
```
The hardware priority queue design was implemented on an Altera Cyclone III (EP3C25) FPGA. When the queue was fully managed with the hardware queue size of 255 or less, the results showed that the queue was 6 times faster when compared to the same queue implemented in software [10]. When the hardware limits were reached, the queue was extended to software as well and the highest length of the queue implemented was 8192. When the co-design implemented queue was compared to that of the software queue, it was noted that the co-design implementation queue was found to be 30% better in terms of performance as shown in Figure 2.7 [10].

Figure 2.7: Performance comparison between the software and hybrid implementation of a priority queue, (a) software priority queue (b) hybrid priority queue [10].

The priority queue which we implemented is based on the hardware software priority queue that was implemented by Flemming Høeg et al. [11]. The hardware software co-design implemented in this paper is different from the priority queue implemented in [10]. In this application, the main goal is to speed up the design at the additional cost of having a hardware component, since the hardware implementation works faster than a software implementation of a binary heap priority queue [11].
The software priority queue in [11] is also implemented in the tradition way, as a heap stored in a binary tree and indexed by an array as shown in Figure 2.8.

![Figure 2.8: Binary heap structure of priority queue in software implementation [11].](image)

However, the major difference between the implementation in [10] and that in [11] is the hardware implementation. The priority in hardware in [11] was realized using parallel computation to achieve faster operation. The interface to the priority queue consists of two registers: an input register, in, and an output register, out [11]. This is shown in Figure 2.9.

![Figure 2.9: Hardware interface [11].](image)

An array of parallel registers is implemented, forming the queue where the elements of larger value are shifted backwards in the registers and smaller elements are kept in the front using comparators for the register pairs. The advantage of this implementation is that, at any given instance, the user has the element with the highest priority available at the first output register and need not wait for the entire
queue to be sorted to extract the element having the highest priority [11]. Figure 2.10 shows the realization of the hardware priority queue done in [11].

![Figure 2.10: Hardware realization of priority queue [11].](image)

Once the hardware limit of the priority queue is reached, the remaining elements that are being added to the queue would be sent to and managed by the software module that has an interface with the hardware module. Figure 2.11 shows the co-design interface for the hardware-software modules. The relative sizes of the hardware and software components can be determined by an analysis of the type of the data being stored, thus enabling an efficient priority queue implementation for data in a given domain.

![Figure 2.11: Hardware-software interface [11].](image)
2.5 Hardware testing/verification methods vs software testing/verification methods

In developing the hardware and software modules, emphasis is given to the testing and verification of these modules, before and after they are integrated together for an SoC. A lot of emphasis has been given to the verification and testing of these components before they are deployed in the market. To maximize efficiency of the development process, it is necessary to detect most of errors in the design as soon as possible. There have been many testing and verification methods in both software and hardware that have been used in the past and are also currently used. These are discussed briefly in the following sections.

2.5.1 Hardware testing/verification methods

A hardware system undergoes multiple verification cycles before the chip is sent for fabrication. Once the physical chip is fabricated, it is tested rigorously for its functionality and robustness. From the microarchitecture specification to the physical layout of the chip, there are various verification steps in each phase of the design cycle such as vector simulation based verification of behavioral and RTL design and static timing analysis checks performed at the gate level netlist to check for proper data flow in the signal path [12]. Layout verification is performed to verify that the design meets the functionality and the design parameters. It includes Design Rule Check (DRC), Layout Verses Schematic (LVS), Electrical Rule Check (ERC), Antenna checks, Electro migration Check (EM), Design for Manufacturability (DFM) check etc. [1]. After the chip is fabricated, it is tested by using the test vectors generated by an Automatic Test Pattern Generation (ATPG) and the internal data and outputs are viewed using the probes and oscilloscope. If a bug is detected after the system or chip is fabricated, then it becomes difficult to rectify it and this is not economically profitable. It is essential to try to ensure that the design is robust enough and without any faults during the initial stages of design. Due to the increasing
complexity of design, it is not always possible to detect the bugs with vector simulation based verification methods. Formal methods of verification have gained importance in the hardware design cycle since these provide an additional layer of verification for increased robustness and fault tolerance of the system in addition to classic simulation and testing methods. Formal verification does not detect the bug in the system, rather it ensures the absence of a bug in the system [13]. Formal verification fits into different stages of the hardware design cycle. It could be used to compare and verify two RTL designs for correctness of functionality at the Initial design phase and in the comparison of an RTL design to a gate level design.

2.5.2 Example of hardware design verification method: Formal verification

Formal methods can be used to prove or disprove the functional equivalence of two designs. In hardware design, this method of verification can be used to compare gate level netlist against its register level transfer source, source RTL to modified RTL, original gate level netlist to modified gate level netlist, Verilog to database and vice versa. The design should use synchronous elements and not combinational logic using state holding loops [13].

RTL – Gate Level design comparison

Register transfer level design is synthesized to its gate level netlist using EDA tools like Synopsys Design Compiler and Cadence Encounter [1]. The source RTL code is synthesized to its corresponding technology gate level netlist. It is always essential to verify that the synthesized netlist is logically equivalent to its source RTL code. Formal verification methods have been found very useful during this phase in the hardware design cycle [13]. The hardware design cycle has many layers of verification during each phase of the cycle. It is always a safer approach to detect and rectify an error/bug early in
the design stages rather than in the later stages which involve many ECOs (Engineering Change Order) which are very costly [1].

Similarly, in the formal verification method, two RTL designs or two gate level designs can be verified against each other for functional equivalence. Figure 2.12 shows the use of formal verification methods in different stages of ASIC design flow.

![Figure 2.12: Formal verification in ASIC design flow [12].](image-url)
2.5.3 Software testing methods

A typical software design cycle in embedded systems is based on the waterfall design methodology and includes the following steps:

Analysis -> Specification/Design -> Coding -> Unit testing -> Integration -> System testing -> Maintenance [14].

Similar to a hardware design cycle, software goes through numerous testing methods to make sure that it is robust, bug free and functionally verified. The software code is tested in either static analysis or dynamic analysis. Static analysis checks for syntax errors, use of individual statements and the structure of code. Most of these static analysis checks are automated. Dynamic analysis checks of the code involve simulating the system under controlled conditions with some specific results that are expected after the code is run [15].

Broadly, dynamic testing is classified into two methods: blackbox or unit testing and whitebox testing.

Blackbox testing

Black box testing ignores the internal mechanism of the system and focuses on the output generated against any particular set of inputs. Blackbox testing is also called functional testing [15]. Black box testing is often used for validation.

Whitebox testing

White box testing is a testing technique that takes into account the internal structure of a system. It is also called structural testing [15]. White box testing is often used for verification. Figure 2.13 shows the different types of testing in a software cycle.
Figure 2.13: Types of testing in software [15].
The testing activities in software can be further classified as follows:

- Unit testing
- Integration testing
- Functional testing
- System testing
- Performance testing
- Acceptance testing

**Unit Testing**
Unit testing is the testing of an individual unit or group of related units. It falls under the class of black box testing. It is often done by the programmer to test that the unit they have implemented is producing expected output against given input [15].

**Integration Testing**
Integration testing is testing in which a group of components are combined to produce output. Also, the interaction between software and hardware is tested in integration testing if software and hardware components have any relation. It may fall under both white box testing and black box testing [15].

**Functional Testing**
Functional testing is testing to ensure that the specified functionality required in the system requirements works. It falls under the class of black box testing [15].
**System Testing**
System testing is testing to ensure that by putting the software in different environments (e.g., operating systems) it still works. System testing is done with full system implementation and environment. It falls under the class of black box testing [15].

**Performance Testing**
Performance testing is testing to assess the speed and effectiveness of the system and to make sure it is generating results within a specified time as in performance requirements. It falls under the class of black box testing [15].

**Acceptance Testing**
Acceptance testing is often done by the customer to ensure that the delivered product meets the requirements and works as the customer expected. It falls under the class of black box testing [15].

**2.5.4 Example of a white box testing method: basis path method**
What are the challenges faced while testing a software system?

To develop a robust and bug free software system, the software needs to be thoroughly tested using test vectors that provide adequate test coverage and also making use of the limited resources available. While developing the test bench with test case vectors, emphasis is on eliminating redundant testing [16]. Software can have many paths between its entry and exit points. Providing complete path coverage is difficult, as every case statement multiplies the number of potential paths and every decision doubles the number of potential paths [16]. The challenge for the test engineer is to select a set of test cases that is most likely to identify as many different potential defects as possible within the limited number of resources [16]. Avoiding redundancy in the test cases is a major criterion and the basis path testing method aims to provide test cases with maximum fault coverage and to avoid redundancy in the test cases.
In software engineering, basis path testing, or structured testing is a white box method for designing test cases [17]. Basis path testing analyzes the control flow graph of a program to find a set of linearly independent paths of execution. The method normally uses McCabe's cyclomatic complexity to determine the number of linearly independent paths and then generates test cases for each path thus obtained [17]. Basis path testing guarantees complete branch coverage (all control flow graph edges), but achieves that without covering all possible control flow graph paths—the latter is usually too costly. Basis path testing has been widely used and studied [17].

Formal verification of software would provide a higher level of confidence but formal methods for software are not well developed and at present are only used in systems with severe safety and security requirements such as aerospace systems [18].
3. APPROACH

This chapter provides the detailed approach to developing the hardware and software modules of the priority queue implementation. Section 3.1 discusses the basic concepts involved in the approach. Section 3.2 describes the hardware implementation of the priority queue. Section 3.3 explains the software implementation of priority queue. Section 3.4 describes the co-design controller module.

3.1 Basic concepts

The modules for the hardware priority queue and software priority queue are built for students to gain experience in designing hardware-software systems. The priority queue is a good example for that. The priority queue implemented here is similar to the design implemented by Flemming Høeg et al. [11]. Hardware and software modules implemented here could be used as is by the students on an FPGA board. A priority queue is a data structure that holds a set of elements in descending (or ascending) order of priority levels [11]. The priority queue uses two basic operations:

- Push (Insert): Inserting new elements into the queue unless the queue is full.
- Pop (Remove): Removing the element having the highest priority from the queue.

3.2 Hardware implementation of the priority queue

A priority queue of size 16 was built in hardware design using RTL Verilog. This is a parameterized design, and the code can be modified to make a queue of a different length. The students can modify the priority queue to any size by changing the Q_MAX_SIZE value in the priority.v file. The input to this queue is a priority, implemented here as a 32-bit positive integer. The hardware realization of the priority queue is made efficient by having two registers, in and out, for each queue position. Elements
in each position can be compared and switched if necessary in parallel to speed up the operation time.

The first in and out registers are shown in in Figure 3.1

![Figure 3.1: Hardware queue [11].](image)

A flag is set for each register to show if the register is empty or full. Before an element can be added to the queue, the status of the flag for the ‘in’ register is checked to see if the register is empty. An element can be sent to the register only if the register is empty and similarly, during the remove operation, an element can be popped out of the out register only when that register is full. In practice, the in location can be kept empty for each input cycle.

The hardware realization consists an array of parallel registers as shown in Fig 3.2 where the left most input register in the array gets the data in and the left most output register has the element of highest (or lowest) priority available for the user [11]. There is an array of in registers and a parallel array of out registers. At every positive clock edge in a clock cycle, an element can be moved into the first input register. At the negative clock edge of the same clock cycle, the elements in each input register are compared to the corresponding elements in the parallel output registers and if an element in an input register is smaller in value (higher priority) than the element in the output register, they are swapped. Hence, at every clock cycle, an element can be inserted into the queue and in the same clock cycle, the
element of highest priority is available in the output port. At the next positive clock edge, a new element can be inserted in the first input register and the elements in the queue can be shifted to the next empty register. At the negative clock edge, comparison and swapping takes place if necessary and this cycle repeats till the limit of hardware queue size is reached. Hence, with this hardware realization a constant operation of insert and delete times $O(1)$ (worst case time) is achieved, which is far better compared to the $O(\log n)$ software realization of a priority queue [11].

![Figure 3.2: Hardware realization [11.](image)](image)

### 3.3 Software implementation of the priority queue

Software realization of the priority queue was done in C. The advantage of implementing the software module is that the size of the priority queue could be changed dynamically. Implementing a complex algorithm is easier in software than in hardware, as software enables the developer to work at a higher level of abstraction and it is cost effective as well. A binary heap priority queue was implemented in the software module. The advantage of having a binary heap implementation is that it does not require additional space for pointers and could be stored compactly as compared to a binomial heap or Fibonacci heap [10]. As discussed above, there are three basic operations performed in a priority queue. Insertion of a new element is always performed at the bottom of the heap (child node). According to the property of the queue, the minimum element (highest priority) needs to be on top of the queue. Hence, this new element inserted is compared with its parent node and if the parent node
is found to be a larger element, their locations in the queue are swapped. This new node would check its parent and keep going till the top of the heap, if necessary, ensuring that the property of the queue is maintained. While inserting a new element, in a queue of \( n \) elements, the operation of finding the proper place in order to keep the list sorted requires \( \log(n) \) operation time (worst case). When the remove operation is requested by the user, the element in the top of the queue (highest priority element) is removed and the element in the extreme child node is copied to the top root node location. The queue is again sorted to have the minimum element on top of the node. The worst case operation time for this requires \( \mathcal{O}(\log(n)) \). It is not necessary to keep all the elements of the queue in a sorted manner, rather, only the minimum element is needed to kept on top of the queue [11]. But each node will have the heap property, i.e., its contents will be (greater/less) than the contents of its children.

The maximum size of the priority queue implemented in the software is 1024, whereas the students while using this software module could change the maximum size of the queue as per their requirements.
Figure 3.3: Example implementation of software priority queue, (a) Element 1 inserted in the bottom child node (b) Element 1 swapped with its parent node element 8 (c) Element 1 swapped with its parent node element 4 (d) Element 1 swapped with element 2 to maintain balanced tree.
Figure 3.3 shows an example implementation of a software priority queue. In this example, insertion of a new element is done and element 1 is inserted in the bottom child node. The child node is compared with its parent node to check if the value of the new element in the child node is less than the parent node, if it is, then the elements are swapped. Here element 1 has higher priority than that of element 8 and their locations are swapped. Now element 1 has its parent node as element 4 and, since 1 has higher priority than 4, its location is again swapped. Again a comparison is made between the child and parent nodes and here the parent node of element 1 is element 2 and swapping is again done. As mentioned earlier, it is not necessary to keep all the elements of the queue in a sorted manner, rather, only the minimum element is needed to kept on top of the queue. And each parent node must have a priority higher than the priority of its children.

3.4 Operation of the co-design controller module

The controller module in the co-design methodology checks the conditions for the flow of data every time the user performs an enqueue or dequeue operation on the priority queue. The controller module controls the flow of data between the hardware module and the software module. While partitioning the entire queue into hardware and software queues, the size of the hardware queue should be chosen to maximize performance for the given application. In the hardware queue, a temporary pair of in and out parallel registers are implemented. During the enqueue operation, data from the hardware to software queue is passed through the temporary in register when the hardware limit is reached. However, the software queue can accept a new element from the temp in register only when the software queue is not in use (when push or pop operation in the child node is not being done). If the software queue is busy, the temp in register would have a wait time of one or more computational cycles to fill in the child node in the bottom of the priority queue tree in software. Similarly, during a
dequeu operation, when the element from the out register is popped, software pops the root element having the highest priority in the software queue and sends the element to the temp out register. This operation is performed during the positive half clock cycle and during the negative half clock cycle, the elements in the temp in and temp out registered are compared with each other and swapped if necessary, which is similar to the operation carried out in the other register pairs in the hardware queue. During the next positive clock cycle, the element from the temp out register is shifted to the empty last out register in the hardware queue. However, the worst case time for sending the highest priority element to the temp out register by the software queue could be $O(\log(n))$. The following is pseudocode which explains the conditions checked by the co-design controller module. Figure 3.4 shows the block diagram of the operation of this module.

![Block Diagram](image)

**Figure 3.4: Co-design manager module**
3.4.1 Pseudocode for the co-design decision controller

1: procedure co-design controller

2: if enqueue =1 & dequeue =0 then

3: check if hardware queue full or empty

4: if hardware queue not full then

5: push element into in register of hardware

6: else if hardware queue is full then

7: check if software queue is idle

8: if software queue is ready then

9: new element is pushed into in register and last element of in register is sent to software queue through temp in register

10: else if software queue is not ready then

11: signal hardware queue to wait until software queue is ready to accept new element

12: end if

13: end if

14: end if

15: if enqueue =0 & dequeue =1 then

16: if hardware is queue is full & software queue is ready then

17: first element is popped out of the hardware register, and root node element from software queue is sent to temp out register, Software queue is updated

18: end if

19: if hardware is queue is full & software queue is not ready then
hardware queue waits for software queue to be ready, then step 17 is executed

end if

if enqueue =1 & dequeue =1 then

error is generated; push and pop operations cannot be done simultaneously

end if

if enqueue =0 & dequeue =0 then

both priority queues are idle

end if

end co-design controller
4. RESULTS AND ANALYSIS

The implementations of the hardware and the software priority queues were based on the hardware-software co-design priority queue implemented by Flemming Høeg et al. [11]. This chapter throws light on the results obtained by the co-designed priority queue [11]. Section 4.1 discusses the hardware implementation results from the paper. Section 4.2 discusses the software implementation results. Section 4.3 describes the result of co-design implementation and advantages of this design approach. Section 4.4 discusses the computation time for the hardware and software modules.

4.1 Hardware implementation results

The queue was optimized for a particular dataset. Initially, the entire design when built in hardware was split into different segments where each segment had different elements of the queue. The maximum total length of queue during the simulation was 453 and the average length of the queue was 182 [11]. Table 4.1 shows the results of distribution of insertion of elements into different segments. This tables shows the distribution of priorities of new elements during the insertion operation as compared to the priorities of elements which are existing in the queue already. The queue here is divided into ten segments where each segment is of equal size. Column A represents the absolute distribution obtained by dividing the entire queue into equal segments and column B represents the relative distribution obtained by dynamically dividing the portion of queue currently in use into segments [11].
As seen from the above table, most of the insertions are done in the first three segments of the hardware queue. The additional hardware segments were not involved in most of the insertions during the operation of the priority queue. As a result, hardware registers were using more space and the implementation was not cost effective.

Another observation made was that, when the size of the priority queue was reduced to 5, 43.6% of the insertions were performed on the first register [11]. Table 4.2 shows the insertion rates for the first five positions.

<table>
<thead>
<tr>
<th>Position number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion rate</td>
<td>43.6</td>
<td>7.0</td>
<td>0.6</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 4.2: Insertion rate into first five positions [11].
4.1.1 Hardware implementation of priority queue

The following is a qslice module of the priority queue. The size of the element and the different input and output ports are initialized here. Comparing the elements based on priority and swapping them, if needed, is done in the module. Priority queue can arrange the priority values either in the highest or lowest priority as per the requirements. Students can run this Verilog code to generate a priority queue having the highest priority element in the first node by assigning value 0 or 1 to the parameter, pq in this file.

Name of the module: qslice (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 06/21/16

/**********************************************************************************
Variables description:
clk- clock signal is high when set 1 and low when set to 0;
rst- reset signal to reset the value of register to 0 when signal set low and reset value to 1 when signal set high;
shunt - first input register in the slice register;
shuntin - array of shunt (input) registers for the size of the hardware;
q - first output register parallel to the shunt register;
qin - array of output registers for the size of the hardware;
deq_req- dequeue request signal when goes high, it enables queue to remove an element from queue;
pq - pq value decides if the sorting is done based on the highest/lowest element in terms of priority. When pq=1, largest element has highest priority and when pq=0 smallest element has highest priority;
**********************************************************************************/

Module description: Qslice module of the priority queue. Input, output ports are initialized here. Comparing the elements based on priority and swapping them, if needed, is done in the module.

***********************************************************************************/

module qslice(clk,rst,shunt,shuntin,q,qin,deq_req,pq);

output [31:0]shunt,q;
input [31:0]shuntin,qin;
input clk,rst,deq_req,pq;
reg [31:0] shunt, q;
wire [31:0] shuntin, qin;
wire clk, rst, deq_req, pq;

//Initializing the slice registers to their natural values
initial
begin
  shunt = 0;
  q = 0;
end

//Re-initializing the slice registers to their natural values
always @(rst)
begin
  shunt = 0;
  q = 0;
end

//Comparing values in shunt and q and swap if necessary
always @(negedge clk)
begin
  if (shunt != 0 && pq == 0 && q != 0 && q > shunt)
    begin
      {q[31:0], shunt[31:0]} = {shunt[31:0], q[31:0]};
    end
  if (shunt != 0 && pq == 0)
    begin
      {q[31:0], shunt[31:0]} = {shunt[31:0], q[31:0]};
    end
  if (shunt != 0 && pq == 1 && q != 0 && q < shunt)
    begin
      {q[31:0], shunt[31:0]} = {shunt[31:0], q[31:0]};
    end
  if (shunt != 0 && q == 0)
    begin
      {q[31:0], shunt[31:0]} = {shunt[31:0], q[31:0]};
    end
end

//Forwarding shunt and q values
always @(posedge clk)
begin
  if (deq_req == 0)
    begin
      shunt = shuntin;
    end
  if (deq_req == 1)
    begin
      q = qin;
    end
end
endmodule
The following priority queue module instantiates the slice modules and stitches them together based on the size of the priority queue. The enqueue and dequeue operations are performed in this module.

Currently, the size of the priority queue has been set to 16. However, students can change the size of the queue as per their requirement by changing the value of Q_MAX_SIZE.

Name of the module: priorityq (implemented in Verilog)

// Author: Bhargav Bappudi
// Date modified: 06/25/16

/************************************************************
Variables description:
clk- clock signal is high when set 1 and low when set to 0;
rst- reset signal to reset the value of register to 0 when signal set low and reset value to 1 when signal set high;
enq_req- enqueue request signal when set high (1), enables queue to insert elements into the queue;
q_empty - When q_empty flag is set to 1, it determines that hardware queue is empty;
q_full- When q_full flag is set to 1, it determines that hardware queue is full;
deq_data - When deq_data is set to 1, the elements in the queue sorted according to their priorities are popped out from the queue;
enq_data- When enq_data is set to 1, the elements are sent into the queue to be sorted according to their priorities;
pq - pq value decides if the sorting is done based on the highest/lowest element in terms of priority. When pq=1, largest element has highest priority and when pq=0 smallest element has highest priority);
***************************************************************************/

/************************************************************
Module description: The following priority queue module instantiates the slice modules and stitches them together based on the size of the priority queue. The enqueue and dequeue operations are performed in this module.
***************************************************************************/

module priorityq(clk,rst,enq_req,q_empty,q_full,deq_data,enq_data,pq);

output q_empty,q_full;
output [31:0]deq_data;
input clk,rst;
input [31:0]enq_data;
input enq_req,deq_req,pq;

reg q_empty,q_full;
reg [31:0]deq_data;
wire clk,rst,pq;
wire [31:0]enq_data;
wire enq_req,deq_req;

parameter Q_MAX_SIZE=16;
integer q_size;
wire [31:0] shuntin[0:Q_MAX_SIZE-1];
wire [31:0] qin[0:Q_MAX_SIZE-1];
wire [31:0] deq_data_store;

//Instantiating all the qslice in the priorityq and stitching them together

genvar i;
gen
for (i=0; i<=0; i=i+1)
begin
qslice
slice(shuntin[i+1],deq_data_store,enq_req?enq_data:0,qin[i],clk,rst,deq_req,pq);
end
for (i=1; i<=Q_MAX_SIZE-2; i=i+1)
begin
qslice slice(shuntin[i+1],qin[i-1],shuntin[i],qin[i],clk,rst,deq_req,pq);
end
for (i=Q_MAX_SIZE-1; i<=Q_MAX_SIZE-1; i=i+1)
begin
qslice slice(qin[i-1],shuntin[i],0,clk,rst,deq_req,pq);
end
endgenerate

//Initializing all other registers to their natural values
initial
begin
q_empty=1;
qu_full=1;
deq_data=0;
q_size=0;
end

//Re-initializing all other registers to their natural values
always @(rst)
begin
q_empty=1;
qu_full=1;
deq_data=0;
q_size=0;
end

//Update q_empty and q_full based on the values of q_size
always @(q_size)
begin
if (q_size==0)
begin
q_empty=1;
qu_full=0;
end
else if (q_size==Q_MAX_SIZE)
begin
  q_empty=0;
  q_full=1;
end
else
begin
  q_empty=0;
  q_full=0;
end
end

// Enqueue operation
always @(posedge clk)
begin
  if (enq_req==1 && deq_req==0 && q_size<Q_MAX_SIZE && enq_data!=0)
  begin
    q_size=q_size+1;
  end
end
// Dequeue operation
always @(posedge clk)
begin
  if (enq_req==0 && deq_req==1 && q_size>0)
  begin
    deq_data=deq_data_store;
    q_size=q_size-1;
  end
end
endmodule
A sample test bench module has been shown below. Students are encouraged to test this priority queue with a different set of inputs as per the size of the queue that is being implemented and run the queue at different clock speeds.

Name of the module: priorityq_tb (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 07/10/16

/**********************************************************************************
Variables description:
Declaring all the variables from priorityq module;
qu_empty – When q_empty flag is set to 1, it determines that hardware queue is empty;
qu_full– When q_full flag is set to 1, it determines that hardware queue is full;
deq_data – When deq_data is set to 1, the elements in the queue sorted according to their priorities are popped out from the queue;
clk– clock signal is high when set 1 and low when set to 0;
rst– reset signal to reset the value of register to 0 when signal set low and reset value to 1 when signal set high;
enq_data– When enq_data is set to 1, the elements are sent into the queue to be sorted according to their priorities;
enq_req– enqueue request signal when set high (1), enables queue to insert elements into the queue;
deq_req– dequeue request signal when goes high, it enables queue to remove an element from queue;
pq – pq value decides if the sorting is done based on the highest/lowest element in terms of priority. When pq=1, largest element has highest priority and when pq=0 smallest element has highest priority);
***********************************************************************************/

/**********************************************************************************
Module description: Sample test bench module to verify the priority queue implemented;
***********************************************************************************/

module priorityq_tb();

wire q_empty,q_full;
reg clk,rst,enq_req,deq_req,pq;
reg [31:0]enq_data;
wire [31:0]deq_data;

// initializing the module from priorityq
priorityq q1(q_empty,q_full,deq_data,clk,rst,enq_data,enq_req,deq_req,pq);

// resetting the registers to initial value 0.
initial
begin
    clk = 1'b0;
rst=1'b0;
enq_req=1'b0;
deq_req=1'b0;
enq_data=0;
forever
  begin
    pq=0;
    #5 clk =~clk;
  end
#800 $finish;
end

// Sending the test data inputs.
initial
  begin
    #5 enq_req=1'b1;
    #10 enq_data=15;
    #10 enq_data=14;
    #10 enq_data=13;
    #10 enq_data=12;
    #10 enq_data=11;
    #5 enq_req=1'b0;
    #5 deq_req=1'b1;
    #55 deq_req=1'b0;
    #5 enq_req=1'b1;
    #10 enq_data=10;
    #10 enq_data=9;
    #10 enq_data=8;
    #10 enq_data=7;
    #10 enq_data=6;
    #10 enq_data=5;
    #10 enq_data=4;
    #10 enq_data=3;
    #10 enq_data=2;
    #10 enq_data=1;
    #10 enq_req=1'b0;
    #10 deq_req=1'b1;
  end
endmodule
Here, the clock cycle for this operation was 10ns with a 50% duty cycle. When the enqueue request was set to 1, the data were sent in to the enqueue registers. After 50ns, the enqueue request was set to 0 and dequeue request was set to 1. At this point of time, the smaller elements having higher priorities were available for the user in the priority queue as can be seen in Figure 4.1. The initial set of inputs to the queue were 15, 14, 13, 12, 11 and the output of higher priority elements were 11, 12, 13, 14 and 15 till the 100th ns of operation.

Figure 4.1: Output of the priority queue hardware implementation
4.2 Software implementation results

The queue was also implemented in a pure software solution and it was found that the total execution time for insertion and remove operations was pretty high compared to the hardware solution, since the worst case delay of the software based priority queue is $O(\log(n))$ as compared to hardware which is $O(1)$. However, it is easier to implement a larger priority queue in software as compared to hardware since it does not take much memory space and can be easily managed too. Table 4.3 shows the priority queue operations in the software module (no values for execution time were given in [11]).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of Operations</th>
<th>% of execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>2805132</td>
<td>10%</td>
</tr>
<tr>
<td>Remove</td>
<td>2805132</td>
<td>20%</td>
</tr>
<tr>
<td>Minimum</td>
<td>5301218</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 4.3: Priority queue operations in software [11].

4.2.1 Software implementation of priority queue

The following modules were built to implement priority queue in C. PriorityQueueC.h is a header file which defines the size of the priority queue. Here the size was defined as 1024. However, students can modify the size of the queue by defining the value of MAXLEN as per their requirement. The other basic functions of priority queue such as push, pop and rebase data to maintain priority are defined in this module.
Name of the header file: PriorityQueueC.h (implemented in C)

//Author: Bhargav Bappudi
//Date modified: 06/08/16

/*********************************************************************************
Variables description:
MAXLEN- defines the size of the software priority queue, here it is defined as 1024, the size of software queue;
push - performs push operation of sending a new element to the queue;
pop - performs pop operation to removing an element from the queue;
*********************************************************************************/

/**************************************************************/
Module description: The following modules were built to implement priority queue in C. PriorityQueueC.h is a header file which defines the size of the priority queue. Here the size was defined as 1024. The other basic functions of priority queue such as push, pop and rebase data to maintain priority are defined in this module.
**************************************************************/

#include <stdio.h>
#include <stdlib.h>

// Defining the maximum length of the priority queue of software module.
#define MAXLEN 1024

// Push an element with value 'd' onto Queue
int push(int *arr, int *len, int d);

// Pop top of the priority queue
int pop(int *arr, int *len, int *d);

// Function to rebase down data to maintain priority when element is removed.
int rebase_down(int *arr, int *len);

// Function to rebase down data to maintain priority when element is added.
int rebase_up(int *arr, int *len);
The following module in PriorityQueueC.c performs the operation of priority queue by sorting and swapping the elements based on the incoming priorities.

Name of the module: PriorityQueueC.c (implemented in C)

//Author: Bhargav Bappudi
//Date modified: 06/08/16

/*
 **________________________________________________________________________
 Variables description:
 int *arr - pointer to the array of queue;
 len- current size of queue;
 parent- parent node in the priority queue;
 child - child node in the priority queue;
 **________________________________________________________________________
 */

/*
 **________________________________________________________________________
 Module description: The following module in PriorityQueueC.c performs the operation
 of priority queue by sorting and swapping the elements based on the incoming
 priorities. It initializes the header file which defined above. Incoming elements
 are pushed into the queue and operation of popping the highest priority element out
 of the queue is performed here.
 **________________________________________________________________________
 */

#include <stdio.h>
#include <stdlib.h>
#include "PriorityQueueC.h"

// Inserting is always at end of the queue and traversed up to maintain priority

int push(int *arr, int *len, int d)
{
    if (*len > MAXLEN){
        printf("Max length reached \n");
        return (0);
    }
    arr[*len] = d;
    *len = *len + 1;
    rebase_up(arr, len);
    return (1);
}

// After push to last node, child node is compared with its parent.
// Elements are swapped if necessary to maintain priority.
// This process is repeated until priority is achieved.

int rebase_up(int *arr, int *len)
{
    int parent, temp;
    int point = *len - 1;
    if (point%2==0)
parent = point / 2 - 1;
else
parent = point / 2;

while (!(arr[parent] < arr[point]) && (parent != point)) {
    if (((parent >= 0) && (arr[parent] > arr[point]))) {
        temp = arr[parent];
        arr[parent] = arr[point];
        arr[point] = temp;
    }
    point = parent;
    if (point % 2 == 0)
        parent = point / 2 - 1;
    else
        parent = point / 2;
}
return 1;
}

// Pop is always the root node, then root node is replaced with last node
// Then traversed down to maintain priority.

int pop(int *arr, int *len, int *d) {
    if ((len <= 0)) {
        printf("Please set a Size for the Queue from 0 to %d ", MAXLEN);
        return 0;
    }
    *d = arr[0];
    arr[0] = arr[*len - 1];
    *len = *len - 1;
    rebase_down(arr, len);
    return(1);
}

int rebase_down(int *arr, int *len) {
    int Lchild, Rchild, child, temp;
    int point = 0;
    Lchild = point * 2 + 1;
    Rchild = point * 2 + 2;

    if (arr[Lchild] < arr[Rchild])
        child = Lchild;
    else
        child = Rchild;

    while (!(arr[child] > arr[point]) && (child <= *len)) {
        if (((child < *len) && (arr[child] < arr[point]))) {
            temp = arr[child];
            arr[child] = arr[point];
            arr[point] = temp;
        }
        point = child;
        Lchild = point * 2 + 1;
        Rchild = point * 2 + 2;
if (arr[Lchild] < arr[Rchild])
    child = Lchild;
else
    child = Rchild;
}
return 1;
}

A sample test bench module has been shown below. Students are encouraged to test this priority queue with a different set of inputs as per the size of the queue that is being implemented and compare the performance against the outputs of the software based priority queue implemented in the paper [11]. Here qs is the number of inputs being sent to the priority queue. Students can modify the limit based on their requirements.

Name of the test bench: main.c (implemented in C)

//Author: Bhargav Bappudi
//Date modified: 06/14/16

***************************************************************************/

Variables description:
Initializing the modules described in header file and PriorityQueueC.c file;
qs - determines the number of elements that user is pushing into the queue;
qs should be < or = MAXLEN, here the value of qs is 90;
*******************************************************************************/

_Module description: Test bench with test cases are written to verify the
implementation of the priority queue. 90 random elements between 1 to 100 are
pushed into the queue and based on their priority the priority queue is build.
*******************************************************************************/

#include <stdio.h>
#include <math.h>
#include "PriorityQueueC.h"

void main()
{

    // qs denotes the number of elements being sent to the queue.
    int i, d = 0, len = 0, arr[MAXLEN], qs=90, prev_pop;
len = 0;

// pushing elements to the queue.

for (i=0; i<qs; i++)
    push(arr, &len, rand()%100);

printf("Random Array elements in Priority Queue are \n");
for (i=0; i<qs; i++)
    printf(" %d ", arr[i]);

// printing elements in sorted priority order.

prev_pop = -104;
printf("\n Popping the Array elements in the priority order \n");
for (i = 0; i < qs; i++)
{
    pop(arr, &len, &d);
    printf(" %d ", d);
    if (prev_pop > d)
        printf("\n WRONG \n");
}


Output:

Figure 4.2: Output of the priority queue software implementation
Figure 4.2 shows the output of the software implementation of the priority queue. Random elements between 1 to 100 are sent to the queue of size 90 here. After an execution time of 0.049 seconds, the queue returns with the smallest element having highest priority and the queue is sorted according to the decreasing order of their priorities, i.e., 0 has the highest priority in this queue.

### 4.3 Hardware-software co-design

Based on the above results, a hardware-software co-design approach was considered where the speed advantage of the hardware implementation was combined with the flexibility of the software module for managing a larger priority queue size without using much memory space. Since most of the insert and remove operations were done in the first few segments of the hardware module, a smaller hardware priority queue size was built which would interact with the software module when the limit of the hardware size was reached. Table 4.4 shows the successive insert and remove operations performed for a sample size of 100 on the dynamic priority queue. For example, 67% of insert operations before a remove consisted of only 1 insertion, while 69% of remove operations before an insertion consisted of only operation.

<table>
<thead>
<tr>
<th>Sequence Length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>67</td>
<td>33</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Remove</td>
<td>69</td>
<td>23</td>
<td>6</td>
<td>1</td>
<td>&lt;1</td>
<td>&lt;1</td>
</tr>
</tbody>
</table>

Table 4.4: Successive insert and remove operations (%) [11].

As seen from the above table, the common case is one insert operation followed by one remove operation where 83% (=67+33/2) of the insert operations are immediately followed by a remove [11].
This shows that most of the activity happens in the front of the queue, i.e., the hardware part, and very few operations are done in the software queue. Hence, it is safe to say that applications like the priority queue which implements both hardware and software parts, can be done in the co-design methodology to save additional space that hardware parts require to build larger queues and also with the flexibility of software modules larger queues can be built without using much memory space. A priority queue for a given application can be customized depending on the insert/remove behavior for that application.

4.4 Co-design controller module

Name of the module: manager.c (implemented in C)

//Author: Bhargav Bappudi
//Date modified: 07/15/16

/****************************************************
Variables description:
deq_req- dequeue request signal when set to 1, enables queue to remove an element from queue;
enq_req- enqueue request signal when set to 1, enables queue to insert elements into the queue;
qlastin - flag status of last register in hardware to say if hardware queue is full or empty if value is 1 and empty if value is 0;
sqinuse- flag to return status if software queue is idle if value is 0 and if it is in operation then value is 1;
tempin - temporary in register in the hardware as an interface between hardware to software module;
tempout - temporary out register in the hardware as an interface between hardware to software module;
****************************************************

Module description: The controller module in the co-design methodology checks the conditions for the flow of data every time the user performs an enqueue or dequeue operation on the priority queue. Controller module controls the flow of data between the hardware module and the software module.

**********************************************************************
```
#include <stdio.h>
#include <stdlib.h>

int manager(int deq_req, int enq_req, int qlastin, int sqinuse, int tempin, int tempout)
{
    //user requesting for enqueue operation.
    if (enq_req==1 & deq_req==0)
    {
        printf("n User is requesting to enter a new element to the queue\n");
        //checking if the hardware queue is empty or if the hardware queue is full and software queue is not in use
        if ((qlastin==0) || (qlastin==1 & sqinuse==0))
        {
            printf("n enter the element\n");
            return 0;
        } else {
            printf("n software queue in use, wait time of one operational cycle in software to move element from tempin to sw queue\n");
            return 1;
        }
    }

    //when the hardware queue is full and user sends in new element to the queue, last element of the in register in hardware moves to temp in reg.
    if (qlastin==1 & enq_req==1 & deq_req==0)
    {
        (qlastin==tempin);
        printf("n when the hardware queue is full and user is sending a new element\n");
        printf("n element in last in register of priority queue is sent to the tempin register\n");
    }

    //user requesting for dequeue operation.
    if (enq_req==0 & deq_req==1)
    {
        printf("n The requested queue element of highest priority from hardware is available for the user\n");
        printf("n Element from the tempout register is sent to last out register in hardware\n");
        printf("n Root node of software queue having highest priority is sent to tempout register\n");
        return 0;
    } else {
        printf("n user cannot perform both enqueue and dequeue operations at the same time\n");
    }
}
```
return 0;
}

if (enq_req==0&&deq_req==0)
{
    printf("\n queue is idle \n");
    return 0;
}
}

Name of the test bench: main.c (implemented in C)

//Author: Bhargav Bappudi
//Date modified: 07/15/16

/******************************************************************************
Variables description: Initializing the modules described in manager.c file;
i - number of test cases to be generated, here i=5;
enq_req[i] - different cases for different values of enqueue request;
deq_req[i] - different cases for different values of dequeue request;
qlastin[i] - different cases for different values of qlastin register;
sqinuse[i] - different cases for different values of sqinuse register;
tempin - temporary in register in the hardware as an interface between hardware to software module;
tempout - temporary out register in the hardware as an interface between hardware to software module;
*******************************************************************************/

/**********************************************************************************
Module description: Test bench with test cases are written to verify the implementation of the manager module. 5 random cases were written with different conditions of hardware and software queue cases of operation.
***********************************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    // initializing the elements for the test bench

    int i;
    int enq_req[5]={1,0,1,0,1};
    int deq_req[5]={0,1,0,0,1};
    int qlastin[5]={0,0,1,1,1};
    int sqinuse[5]={0,0,1,1,0};
    int tempin, tempout;

    for (i=0; i<5; i++)
    {
        printf("\n%d th iteration\n",i);
        manager(enq_req[i], deq_req[i], main_tempin[i], sqinuse[i], tempin, tempout);
    }
}
Figure 4.3 shows the output of the manager module with different sequences of operations performed by the hardware and software queue. Test case 1 shows that when the enqueue request was invoked by the user, dequeue request was 0 and hardware queue was not full which was known from the value of the flag of the last input register in the hardware queue. Hence the new element was sent into the hardware input register. This operation was done in one cycle in the hardware module. Test case 2 shows that when dequeue request was invoked by the user, enqueue request was 1, hardware limit was reached and software queue was not in use, so the element of highest priority from the hardware out register was popped out. This operation was performed in two operation cycles in hardware. Also, the element from the root node of the software queue was moved to the temp out register which happens in two operation cycles in the software queue. In the next operation cycle in the hardware,
the element from temp out moves to the last out register in the hardware. Test case 3 shows that enqueue request was invoked by the user, dequeue request was 0, hardware limit was reached and the software queue was in use. The element is sent into the first hardware in register and the element from the last in register in the hardware queue is moved to the temp in register. This operation is performed in one hardware cycle. Since the software queue is busy, the element in temp in register needs to wait for one operation cycle in hardware to move to the bottom of the software priority queue tree to the child node. This entire operation needs two operation cycles in the software queue. Test case 4 shows that both enqueue and dequeue operation is 0, which means that user is not performing any push or pop action. And similarly, both enqueue and dequeue operation cannot happen together which is shown in test case 5. Table 4.5 shows the different operations on each test case.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Request by user</th>
<th>Hardware queue limit</th>
<th>Software queue in use?</th>
<th>Operation performed</th>
<th>Computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enqueue = 1</td>
<td>No</td>
<td>No</td>
<td>New element inserted into the hardware queue.</td>
<td>1 hardware cycle</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enqueue = 0</td>
<td>Yes</td>
<td>No</td>
<td>Element from hardware out register is popped out. Element from top root node is moved to temp out register and sent to the last vacant hardware out register.</td>
<td>2 hardware cycles; 2 software cycles</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Enqueue = 1</td>
<td>Yes</td>
<td>Yes</td>
<td>New element sent to hardware in reg. Last element in hardware in reg is shifted to temp in reg. Wait time of 1 s.w operation cycle for element to move from temp in to last child node in s.w binary heap tree.</td>
<td>1 hardware cycle; 2 software cycles</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Enqueue = 0</td>
<td>Yes</td>
<td>Yes</td>
<td>Element from hardware out register is popped out. Software queue is in progress, wait time of $O(\log(n))$ to pop root node element from s.w queue to temp out reg and then sent to the last vacant hardware out register.</td>
<td>2 hardware cycles; $2 + O(\log(n))$ software cycles</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Enqueue = 0</td>
<td>No</td>
<td>No</td>
<td>Element from hardware out register is popped out.</td>
<td>1 hardware cycle</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Enqueue = 1</td>
<td>Yes</td>
<td>No</td>
<td>New element inserted into the hardware queue. Last element in hardware in reg is shifted to temp in reg., ready to move from temp in to last child node in s.w binary heap tree.</td>
<td>1 hardware cycle; 1 software cycle</td>
</tr>
<tr>
<td></td>
<td>Dequeue= 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Different operations performed on the priority queues and the computation time required
4.5 Computation time for the hardware and software modules

The worst case time complexity in a hardware based priority queue is $O(1)$, i.e., it takes 1 clock cycle for the element of the highest priority from the queue to be ready for the user. However, in a software based priority queue, probably several cycles might be required for the computation. The larger number of operations, $O(\log(n))$, would make the software queue slower. In addition, the software queue will be larger. When data is moved between the hardware and software queues, the hardware queue must wait for the software queue to be ready to accept or provide an item. This may cause an enqueue or dequeue operation in the entire queue to take time which is $O(\log(n))$. So in practice the time taken for queue operations on a set of data will be dependent on the actual data itself.
5. CONCLUSION AND FUTURE WORK

This chapter summarizes the work done and describes how these hardware and software systems implemented here can be further developed in future work.

5.1 Summary of work

The main purpose of this thesis was to provide modules to use in building hardware software systems for students and the priority queue is considered a good example for that. The hardware module for the priority queue was implemented using RTL Verilog. The hardware based priority queue was implemented with a size of 16. However, students are given the flexibility to change the size and simulate it with different sizes to compare and understand the difference in the performance and time complexity compared to the software based priority queue. The software based priority queue was implemented in C. While developing the module initially, the size of the priority queue was set to 1024. However, similar to the hardware design, the students are encouraged to change the size of the queue and compare its performance against the hardware implementation. Students can use these modules to create a hardware-software priority queue with optimized performance for a given data set. Students can run these hardware and software modules to compare their results against the results mentioned [11].
5.2 Future work

The hardware and software systems for the priority queue which were implemented can be extended into a co-design implementation. The first important extension that needs to be done to these systems is to create an interface between the hardware and software modules to enable them to communicate with each other and work together. The major challenge here for the user is to define the hardware limit and create the interface between the hardware and software designs so that they can communicate with each other. A user manual/tutorial can also be provided which explains the operation of the priority queue and how to choose the type of implementation needed based upon the user requirements. These extensions could be implemented as future work.
Appendix – Hardware verification of four-bit adder

For the hardware design, a four-bit adder was implemented using RTL Verilog and was tested with test bench cases for its functionality. After verifying the correctness of the functionality, it was made as my original/golden/reference design. The RTL Verilog was modified with a simplified Boolean expression for the four-bit adder albeit maintaining the functionality. This design was now verified against the golden design for logic equivalence check. Formality tool created compare points and logic cones were formed for both reference and implemented design and were proved to be mapping correctly against each other. This was a test for the RTL – RTL design verification.

The RTL design was then synthesized using 40nm technology node and a gate level netlist was generated. This gate level netlist was verified against the golden RTL design. Tool created compare points and logic cones to map both the designs for design equivalence. This was a test for the RTL – gate level design verification.

The following modules are the Verilog code for four-bit adder with both reference design and modified design.
Name of the module: full adder (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 05/28/16

*******************************************************************************/
Variables description:
a and b are the two integer inputs to the adder;
Sum is the sum of the inputs;
carry is carry of the sum;
*******************************************************************************/

*******************************************************************************/
Module description: Input and output ports are defined. Sum operation and carry
operation are performed in this module. This is the golden design in the formal
verification tool.
*******************************************************************************/

// initializing the variables
module full_adder ( a , b , c , sum , carry );

// defining the output port
output sum ;

// defining the output port
output carry ;

// defining the input ports
input a , b , c ;

// sum operation of three integers is performed.
assign sum = ((~a) & (~b) & (c)) | ((~a) & (b) & (~c)) | ((a) & (~b) & (~c)) | (a) & (b) & (c);

// carry operation
assign carry = ((a) ^ (b)) & (c) | ((a) & (b));
endmodule

module adder_4bit ( a , b , sum , carry );

output [3:0] sum ;
output carry ;
input [3:0] a ;
input [3:0] b ;
wire [2:0] s;

// performing the 4 bit addition on 4 registers.
full_adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
full_adder u1 (a[1],b[1],s[0],sum[1],s[1]);
full_adder u2 (a[2],b[2],s[1],sum[2],s[2]);
full_adder u3 (a[3],b[3],s[2],sum[3],carry);
Name of the test bench: adder_4bit_tb (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 05/28/16

Variables description: Variables defined in adder_4bit are initialized here;
*******************************************************************************/
Module description: Test bench with test cases to verify the functionality of the
four-bit adder golden design.
*******************************************************************************/

// test bench module
module adder_4bit_tb ();

// defining the register ports
reg [3:0] a,b;
wire [3:0] sum;
wire carry;

// initializing the variables from adder_4bit module
adder_4bit uut (a,b,sum,carry);

initial
begin

// sending in the input test vectors.

a= 4'h0; b=4'h0;
#10 a= 4'h1; b=4'h1;
#10 a= 4'h2; b=4'h2;
#10 a= 4'h3; b=4'h3;
#10 a= 4'h4; b=4'h4;
#10 a= 4'h5; b=4'h5;
#10 a= 4'h6; b=4'h6;
#10 a= 4'h7; b=4'h7;
#10 a= 4'h8; b=4'h8;
#10 a= 4'h9; b=4'h9;
#10 a= 4'h3; b=4'h3;

end

endmodule
The above figure shows the output of the 4-bit adder which was implemented as the reference design. Here the inputs are being sent to the register a and b after every 10 ns. Sum port shows the sum of the inputs in a and b register and carry signal turns high if there is any carry for the sum. This output verifies the functionality of the four-bit adder and hence, used as our reference design.
Name of the module: full adder (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 05/28/16

/*******************
Variables description:
a and b are the two integer inputs to the adder;
Sum is the sum of the inputs;
carry is carry of the sum;
*******************/

/*******************
Module description: Input and output ports are defined. Sum operation and carry operation are performed in this module. This is the modified design in the formal verification tool. The carry function has been modified with a simplified Boolean expression as compared to the golden design.
*******************/

// initializing the variables

module full_adder ( a , b , c , sum , carry );

// defining the output port
output sum ;

// defining the output port
output carry ;

// defining the input ports
input a , b , c ;

// sum operation of three integers is performed.
assign sum = ((~a)&(~b)&(~c)) | (a)&(~b)&(~c) | ((a)&(~b)&(~c)) | (a)&(b)&(c);

// carry operation
assign carry = ((a)^b)&c;

endmodule

module adder_4bit ( a , b , sum , carry );

output [3:0] sum ;

output carry ;

input [3:0] a ;

input [3:0] b ;

wire [2:0] s;

// performing the 4 bit addition on 4 registers.

full_adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
full_adder u1 (a[1],b[1],s[0],sum[1],s[1]);
full_adder u2 (a[2],b[2],s[1],sum[2],s[2]);
full_adder u3 (a[3],b[3],s[2],sum[3],carry);

endmodule
Name of the test bench: adder_4bit_tb (implemented in Verilog)

//Author: Bhargav Bappudi
//Date modified: 05/28/16

FLICTECTIONEY: Variables defined in adder_4bit are initialized here;
FLICTECTIONEY: Variables defined in adder_4bit are initialized here;

Module description: Test bench with test cased to verify the functionality of the

/ test bench module
module adder_4bit_tb ();

// defining the register ports
reg [3:0] a,b;
wire [3:0] sum;
wire carry;

// initializing the variables from adder_4bit module
adder_4bit uut ( a,b ,sum ,carry );

initial
begin
// sending in the input test vectors.

a= 4'h0; b=4'h0;
#10 a= 4'h1; b=4'h1;
#10 a= 4'h2; b=4'h2;
#10 a= 4'h3; b=4'h3;

#10 a= 4'h4; b=4'h4;
#10 a= 4'h5; b=4'h5;
#10 a= 4'h6; b=4'h6;

#10 a= 4'h7; b=4'h7;
#10 a= 4'h8; b=4'h8;
#10 a= 4'h9; b=4'h9;

#10 a= 4'h3; b=4'h3;

end
endmodule

This modified design is verified using formality tool against the reference design and it proved that in
spite of changes in the RTL design of Boolean expressions, the design is functionally equivalent to the
reference design.
References


Accessed online on June 9th 2016.


Accessed online on May 25th 2016.


Accessed online July 17, 2016.