Louisiana State University
LSU Digital Commons

LSU Master's Theses | Graduate School

2016

A Study of FPGA Resource Utilization for Pipelined Windowed Image Computations

Aswin Vijaya Varma
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

 Part of the Electrical and Computer Engineering Commons

Recommended Citation

Vijaya Varma, Aswin, "A Study of FPGA Resource Utilization for Pipelined Windowed Image Computations" (2016). *LSU Master's Theses*. 3086.
https://digitalcommons.lsu.edu/gradschool_theses/3086

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

A STUDY OF FPGA RESOURCE UTILIZATION FOR PIPELINED WINDOWED IMAGE COMPUTATIONS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The School of Electrical Engineering and Computer Science

by
Aswin Vijaya Varma
B.Tech, Mahatma Gandhi University, 2010
August 2016

Acknowledgments

I would like to express my most sincere gratitude to my advisor Dr. Ramachandran Vaidyanathan for his exemplary support, patience and excellent guidance throughout my thesis implementation and writing. Without his valuable suggestions and constructive directions, this thesis would not have been successful. I also thank Dr. Suresh Rai and Dr. Lu Peng for agreeing to be on my thesis committee and giving their valuable suggestions to improve my thesis. I express my sincere thanks to Mohammed Nashid Hassan, the ex-IT Manager of LSU Continuing Education for providing me the financial stability by offering Graduate Assistantship in his department throughout my time at LSU.

I thank my loving parents Vijaya Varma and Jayasree Varma for being a constant source of inspiration and motivation as being role models to me throughout my life. Without their support, I wouldn't have been what I am today, both as a person and as a professional. I would like to thank my friend Tom, for his patience of explaining me the concepts of Image Processing and being there as a constant knowledge source throughout my thesis. I would like to express my sincere gratitude and love towards my extended family in the USA, my friends, Padmapriya, Nikhil, and Hari. I would like to extend a special thanks to my beloved friend, late Dr. Anton Joe for showing how to motivate oneself to be a graduate research student by constantly asking questions and finding answers to them. Last, but never the least, I would like to thank my fiancé Parvathy Varma, for her endless encouragement and motivation, to help me complete writing this thesis.

Table of Contents

List of Tables

List of Figures

Abstract

In image processing operations, each pixel is often treated independently and operated upon by using values of other pixels in the neighborhood. These operations are often called windowed image computations (or neighborhood operations).

In this thesis, we examine the implementation of a windowed computation pipeline in an FPGA-based environment. Typically, the image is generated outside the FPGA environment (such as through a camera) and the result of the windowed computation is consumed outside the FPGA environment (for example, on a screen for display or an engine for higher level analysis). The image is typically large (over a million pixels 1000×1000 image) and the FPGA input-output (I/O) infrastructure is quite modest in comparison (typically a few hundred pins). Consequently, the image is brought into the chip a small piece (tile) at a time.

We define a handshaking scheme that allows us to construct an FPGA architecture without making large assumptions about component speeds and synchronization. We define a pipeline architecture for windowed computations, including details of a stage to accommodate FPGA pin-limitation and bounded storage. We implement a design to better suit FPGAs where it ensures a smoother (stall-resistant) flow of the computation in the pipeline. Based on the architecture proposed, we have analytically predicted resource usage in the FPGA. In particular, we have shown that for an $N \times N$ image processed as $n \times n$ tiles on a z-stage windowed computation with parameter w, $\Theta(n^2 + \log N + \log z)$ pins are used and $\Theta(n^2 z)$ memory is used. We ran simulations that validated these predictions on two FPGAs (Artix-7 and Kintex-7) with different resources. As we had predicted, the pins and distributed memory are the most used resources. Our simulations also show that the operating clock speed of the design is relatively independent of the

number of stages in the pipeline; this is in line with what is expected with the handshaking scheme that isolates the timing of communicating modules.

This work, although aimed at FPGAs, could also be applied to any I/O pin-limited devices and memory limited environments.

Chapter 1
Introduction

In image processing operations, each pixel is often treated independently and operated upon by using values of other pixels in the neighborhood. The neighborhood pixels whose values determine a given pixel's transformation is a window. A windowed operation (or a neighborhood operation) slides the window through the image and manipulates each pixel. Informally for any pixel p, a windowed computation of size w uses the $(2w + 1) \times (2w + 1)$ pixels around pixel p to compute a new value for p. A complete image processing algorithm applies several such window functions (in sequence) to an image to get the desired result. These windowed computations, typically form a z-stage pipeline (for some integer $z \geq 1$) with stage i receiving its input from stage $i - 1$ and producing an output for stage $i + 1$.

In this thesis, we examine the implementation of a windowed computation pipeline in an FPGA-based environment. Typically, the image is generated outside the FPGA environment (such as through a camera) and the result of the windowed computation is consumed outside the FPGA environment (for example, in a screen for display or an engine for higher level analysis). The image is typically large (over a million pixels 1000×1000 image) and the FPGA input-output (I/O) infrastructure is quite modest in comparison (typically a few hundred pins). Consequently, the image is brought into the chip a small piece (tile) at a time.

We discuss the implementation of the windowed computation pipeline technique discussed in Vaidyanathan *et al.* [9], modified for suitability to an FPGA environment. An $N \times N$ image is divided into tiles of size $n \times n$ and processed as a series of z windowed computations, each of

"size w." In general, the size of the windowed computation may differ across the stages of the pipeline; we, however, assume a fixed w (maximum of the window sizes) for simplicity of discussion and to account for a key component in our design that depends on the maximum value. An $n \times n$ tile denotes the smallest unit of image that can be moved between each of the z stages. This is reflective of the I/O bandwidth (pins) that the FPGA affords.

We implement a general handshaking scheme that allows for the stages and other modules (both within and outside the stage) to operate relatively independently. This allows for a general study of our implementation of the windowed computation pipeline across FPGA types and even across multiple FPGA platforms.

The problem of dealing with windowed computation of images is as old as image processing as a field itself. For example, the paper titled "Computer Architectures for Pictorial Information Systems" talks about how image parallelism brings about cost effectiveness into image processing [8]. There are many technical papers which discuss different types of windowed computation techniques. In fact, each image processing algorithm is based on a different windowed computation technique.

In previous work related to this paper, pipelined scheduling has been studied by [1]. However, an analytical framework has been laid in the field of windowed image computation only by Vaidyanathan *et al.* [9]. Reuse of hardware components by reconfiguration could be seen as an effective way to reduce hardware costs of the design. Pipeline reconfiguration is discussed by Goldsten *et al.* [13]. Benoit *et al.* gives a survey of pipelined workflow scheduling, discussing various models and algorithms on pipelined workflow scheduling [1]. Others discuss the implementation of image processing on specific platforms [2], [5]. S. Asano *et al.* compare the performance of FPGA, GPU and CPU using three image processing applications and draws a

conclusion on which one is faster under which conditions [12]. Bishop *et al.* [21] and Srivastava *et al.* [22] have also studied image algorithms on reconfigurable platforms.

Our aim is to analytically study windowed computation and implement it on different FPGAs and vary the design parameters to see the effect on the pipeline and the utilization of FPGA resources. The contribution of this work are the following:

- We define a handshaking scheme that allows us to construct an FPGA architecture without making large assumptions about component speeds and synchronization. This also extends the relevance of this work to other FPGAs and I/O bandwidth limited modules.
- We define a pipeline architecture for windowed computations, including details of a stage to accommodate FPGA pin-limitation and bounded storage. This architecture is also generic with the potential for extension to other environments.
- We modify the method of Vaidyanathan *et al.* [9] to better suit FPGAs where it ensures a smoother (stall-resistant) flow of the computation in the pipeline.
- Based on the architecture proposed, we analytically predict resource usage in the FPGA. In particular, we show that for an $N \times N$ image processed as $n \times n$ tiles on a z-stage windowed computation with parameter w:
    - $\Theta(n^2 + \log N + \log z)$ pins are used.
    - $\Theta(n^2 z)$ memory is used.
- We run simulations that validate these predictions on two FPGAs (Artix-7 and Kintex-7) with different resources. As predicted, the pins and distributed memory turns out to be the most used resources. Our simulations also show that the operating clock speed of the design is relatively independent of the number of stages in the pipeline; this is in line with what

would be expected with the handshaking scheme that isolates the timing of communicating modules.

In the next Chapter, we discuss general structure of an FPGA, explaining various FPGA resources that are important to this work. We also discuss the Xilinx Vivado design tool that was used to implement the design in Chapter 5. Chapter 3 discusses windowed image computations that are the main focus of this work. Another central idea, handshaking, used to synchronize the various modules that are working independently is discussed in Chapter 4. The main chapter of this thesis, Chapter 5, goes on to discuss about the structure of windowed computation pipeline and its implementation. The simulation results obtained from the Xilinx Vivado tool is tabulated and expressed in plots, in Chapter 6. Finally, in Chapter 7, we summarize on results and make some concluding remarks.

Chapter 2
Preliminaries

In this chapter, we describe ideas central to the work in the thesis. Specifically, in Section 2.1, we describe a generic structure of an FPGA, focusing mainly on portions important to this work. In Section 2.2 we describe the Xilinx Vivado design tool that we used for this work.

2.1 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) [3, 4] are integrated circuits with logic elements that can be programmed to fit the designer's needs. Specifically, the hardware functionality and interconnects can be programmed to build a custom circuit suited for a particular purpose. Depending on the type and size of the design, the utilization of FPGA resources varies. The study in this thesis deals with the utilization of these FPGA resources. Hence, it is important to know details of these resources. In describing them, we broadly use terminology from Xilinx FPGAs [16]. In this thesis, we have used the Xilinx Artix-7 and Kintex-7 [16] FPGA devices. However, the ideas described apply to other FPGAs as well.

The general structure of an FPGA is shown in Figure 2.1. Broadly it contains the following elements:

- Configurable Logic Blocks (CLBs) are the basic hardware elements that can be configured to construct modules for simple logic functions. They comprise of Look-up tables (LUTs), flip-flops, registers, multiplexers and other logic resources.

- The interconnection fabric consists of a set of wires and configurable switches that connect the CLBs and I/O pins. By suitably configuring the interconnection fabric and the CLBs, the FPGA forms a circuit suited to the problem at hand.
- Switches as noted earlier, connect CLBs to the interconnection fabric. Typically, they are reconfigurable.
- Input/output (I/O) pads are interfaces that allow movement of data in and out of the FPGA.

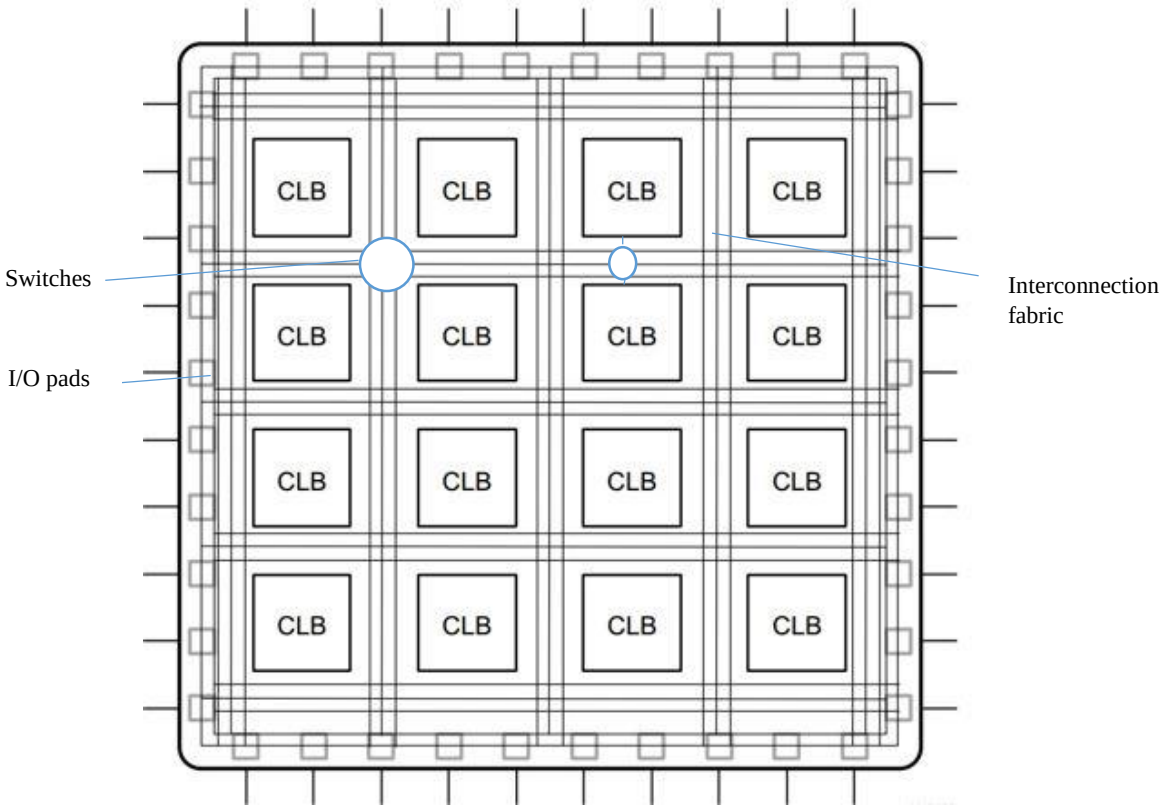


Figure 2.1: Basic FPGA Architecture. This figure has been adapted from the Xilinx 7 series FPGAs configurable logic block user guide [15].

Apart from these basic elements, an FPGA could have additional elements to increase the efficiency and throughput of the design. These elements (see Figure 2.2) include the following:

- Multiply-accumulate blocks (useful for dot products, and logic functions in DSP operations).
- Block RAM (BRAM) that are memory banks (within the chip) used typically for storing on-chip data.
- High speed serial transceivers that provide the interface needed (for example SerDes) to rapidly communicate across a pin-limited interface.
- Phase-locked loops (PLLs) to drive the FPGA at different clock speeds.
- Memory controllers for use with external memory, through address and data buses and control signals.

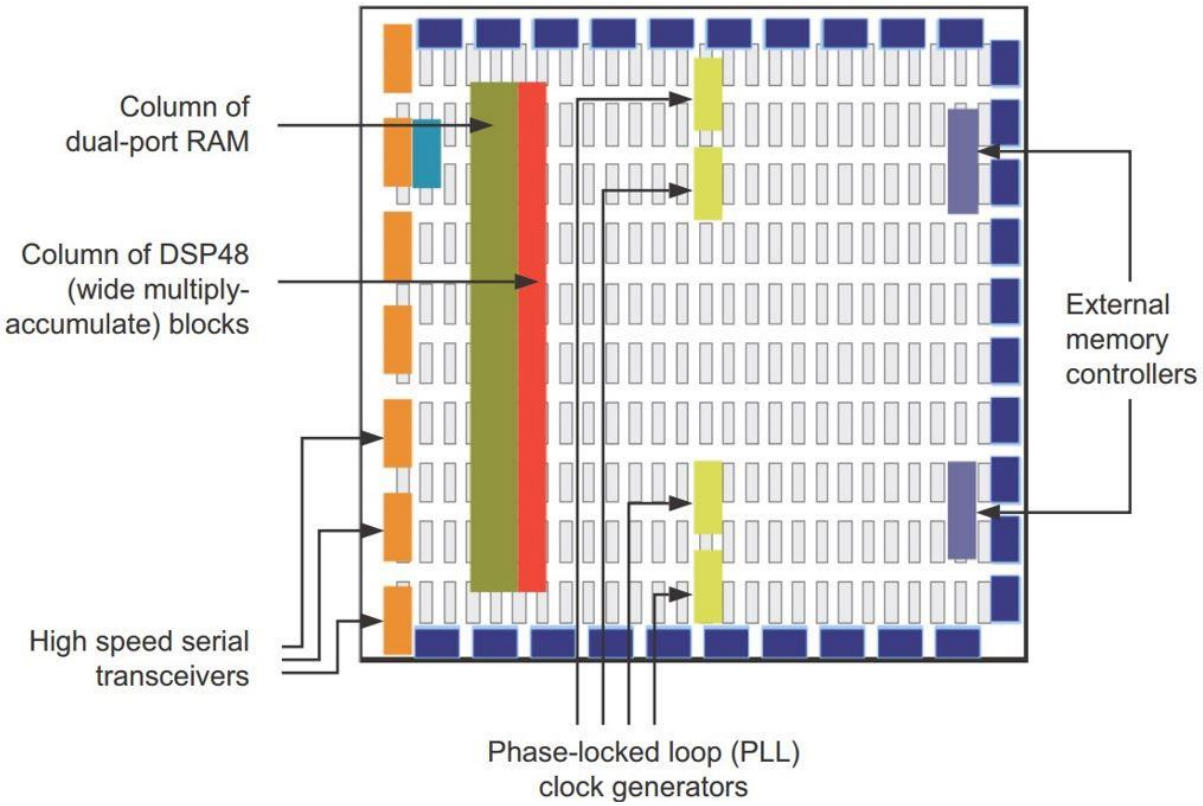


Figure 2.2: Contemporary FPGA Architecture. This figure has been taken from the Xilinx 7 series FPGAs configurable logic block user guide [15].

The focus of this thesis is to study the dependence of windowed image computations on FPGA resources. The image is typically too large to bring into the FPGA in its entirety as the number of pins available and the on-chip memory is restricted. The processing of the image across a multistage pipeline requires storage of a large fraction of the image. This too imposes significant restrictions on BRAM use within the FPGA. The image pipeline stages themselves require FPGA real estate (CLBs and interconnects). Thus the three main resources that we consider are CLBs, in particular, LUTs used as "distributed memory", I/O pins, and BRAMs. We now elaborate on these resources.

CLBs

The logic elements, LUTs and Flip-flops are a part of a configurable logic block (CLB) in the FPGA. Each CLB contains two "slices" connected to a switch matrix to provide routes to other FPGA resources (see Figure 2.3). Figure 2.4 shows a detailed view of a slice. The slice contains an input stage of flip-flops (that may be bypassed). These, broadly speaking, lead to a stage of LUTs whose inputs can come from various sources. The LUTs can be configured to perform various logic functions and their outputs exit the slice directly or through another stage of LUTs (functions).

A carry chain (CIN, COUT in Figure 2.4) allows for communication between a column of slices. Other interconnect resources can also be used to connect CLBs.

The flip-flops can also be configured as latches. Figure 2.4 illustrates, what is known as, a SLICEM. Here the LUT can also be used for storage of data (not just for the truth table of a logic function). This collection of storage spread across SLICEMs is called *distributed memory*. They provide a potentially faster alternative to the BRAM. Only one-third of the total slices available

are of type SLICEM described above. The remaining slices are of type SLICEL whose LUTs cannot be used as distributed memory.

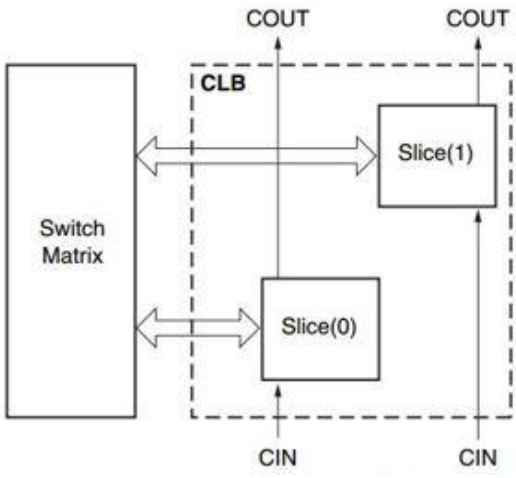


Figure 2.3: Basic CLB structure. This figure has been taken from Xilinx 7 Series FPGAs Configurable Logic Block guide. [15]

Slice resources

The CLBs labeled CLB_LL have only slices that can be used for logic. The CLBs labeled CLB_LM have both types of slices that can be used as logic and memory. CLB_LM accounts for one quarter of the CLBs in the Xilinx Kintex-7.

Block RAM

A Xilinx 7-series FPGA device contains several block RAM memory elements that can be configured as general-purpose 36kB or 18kB RAM/ROM memories. These single-port (or dual-port), block RAM memories offer fast and flexible storage of large amounts of on-chip data. On-chip memory is a scarce resource.

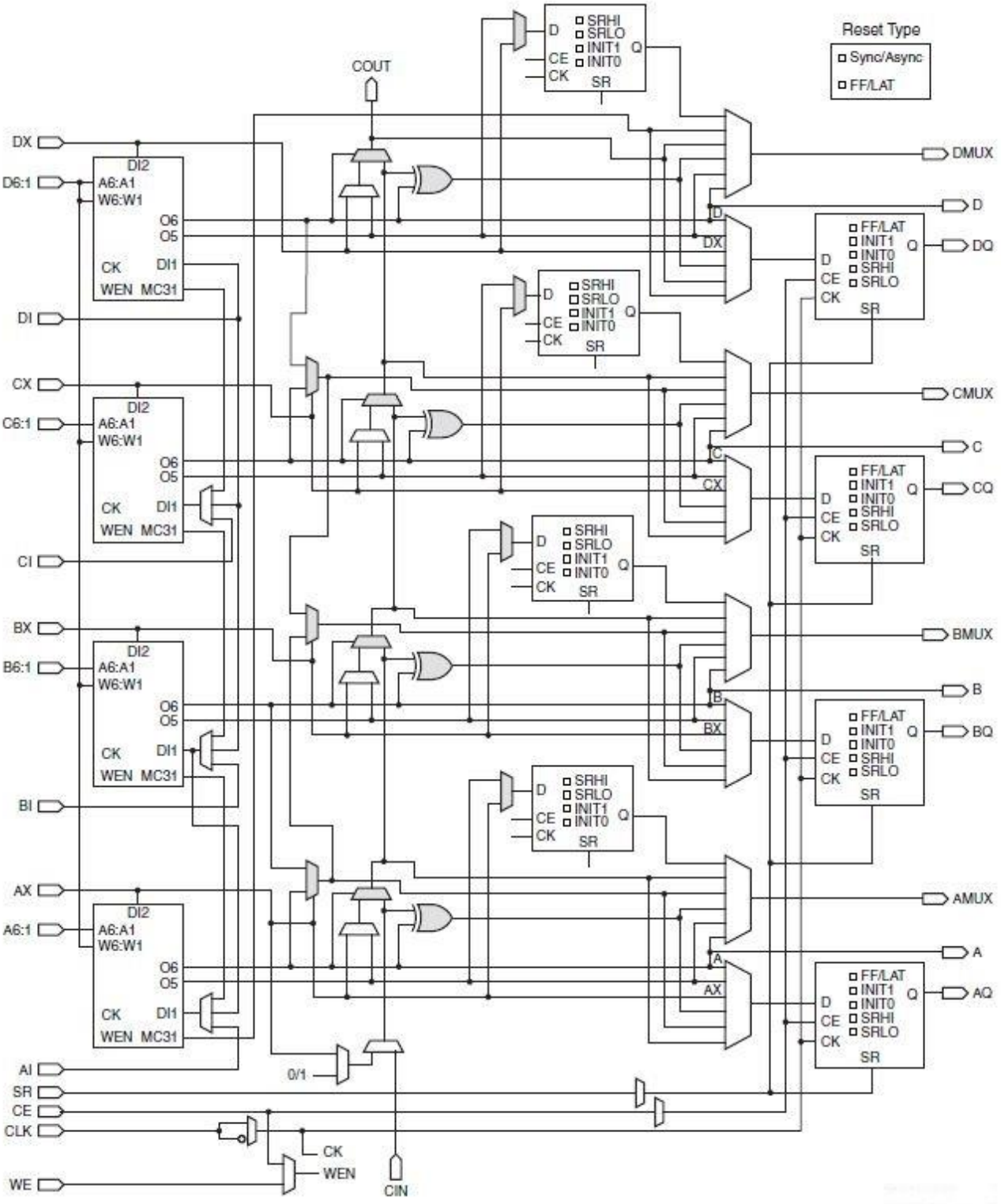


Figure 2.4: Structure of SLICEM. This figure has been taken from Xilinx 7 Series FPGAs Configurable Logic Block guide. [15]

2.2 Vivado Design Tool

The implementation, testing and resource utilization study of the windowed computation pipeline was on the Xilinx Vivado HL Webpack design tool. This section talks about the steps involved in specifying a design starting from design enetry, through RTL design all the way to the place and route step in the tool design flow. We had access to two 7-series FPGAs namely, Artix-7 and Kintex-7, available with the WebPack version of the design tool.

The basic steps involved in our design implementation are as follows (see Figure 2.5):

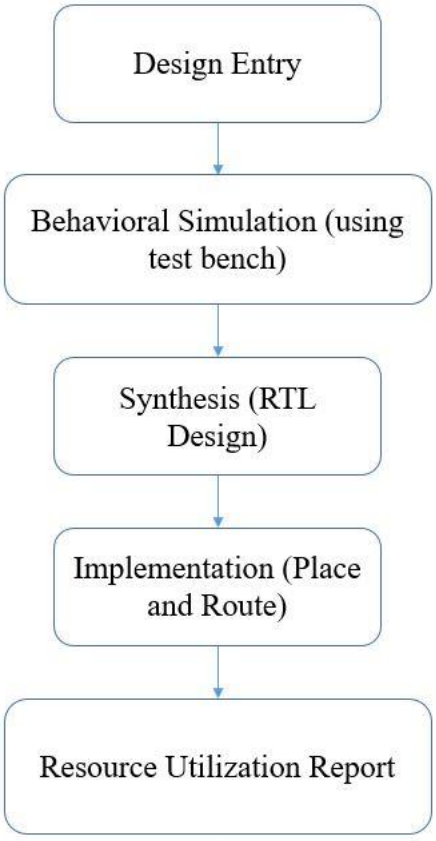


Figure 2.5: Basic design flow [18]

Design Entry: This step first creates a project with a desired name and specifies the output directory location. The design tool provides the flexibility to choose from a variety of project

options, such as RTL project, post-synthesis project, I/O planning project, imported project, or an example project (using a pre-defined template). For Verilog code, Vivado synthesis supports a part of the Verilog constructs specified in the IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005) [6]. In the next step, sources, Intellectual Properties (IPs) and constraints are added and the user can specify existing configurable IP provided by the tool to be used in the design or add physical and timing constraints to the project. Finally, a target device is selected (FPGA or Evaluation board). For our implementation, we have selected the Kintex-7 (XC7K70T) and Artix-7 (XC7A200) as the target devices. [16]

Behavioral Simulation: This step in the design flow make sure that the design does what it is intended to do. From the simulation settings, one can change the target simulator, the target simulation language and specify the top module of the design. After the initial compilation of the source files, the elaboration phase creates a hierarchical representation of the modules involved in the design going into the in-depth details to specify the design in terms of the most basic primitive elements, typically gates. Finally, a simulation phase executes the elaboration phase snapshot. This phase also includes waveform debug, where a test bench provides inputs for the design and generates outputs from it. These input-output combinations can be used for the formal verification of the design.

Synthesis: This step transforms the user's HDL code into gate-level primitives. The Vivado synthesis tool uses the selected target FPGA architecture to map the HDL logic into the various components inside the device.

The synthesis tool recognizes various logic elements such as flip-flops and registers, that can be implemented using the Configurable Logic Blocks (CLBs) in Xilinx FPGAs [16]. The synthesis tool also interprets various styles in which a RAM or ROM can be coded. Based on the

type of memory that the user wants to implement, the synthesis tool interprets various memories as distributed RAM or block RAMs. Our implementation makes extensive use of this distinction between the two memory types. Usually, for larger memories, it is a safe practice to make the tool infer block RAMs. Trying to make the synthesis tool infer distributed RAM for larger memories will lead to excessive CLB utilization with a potential performance slowdown. The dedicated Block RAM is identified with synchronous writes and synchronous reads, whereas, the distributed RAM has synchronous writes and asynchronous reads.

Place and Route: After the synthesis step, the Vivado tool outputs a netlist file, which is the description of a gate-level circuit. The place and route phase takes the netlist and places it on the FPGA device and provides necessary routing between the components. This process produces an output known as Native Generic Database (NGD) file. The map process in the implementation step maps the NGD file into FPGA logic components such as CLBs and IOBs [15]. The output of this process is called a Native Circuit Description (NCD) file which gives the physical representation of the design mapped to the FPGA components. The NCD file is used as input for bitstream generation by the Place and Route process. The Bitstream file is used to configure a physical target device (FPGA or evaluation board). We have not created a bitstream file for our study.

Resource Utilization Report: Once the implementation step is completed, Vivado produces multiple reports, of which the reports that are of significance to us are as follows.

- Vivado Synthesis Report gives a brief description of which code construct is mapped to what logic element in the FPGA.

- Utilization Report is produced right after the synthesis step which gives an estimation of how many resources are required for the design to be implemented on the target device. The resources of interest to us include Slice LUTs, BRAMs and IOBs.
- Post-implementation Utilization report is produced after the implementation phase. During the implementation phase, the tool optimizes the design so as to save resources. Generally, this utilization report gives a better representation of how many logical elements are used versus how many are available in the target device.
- Timing summary report gives a timing summary, if the design met all of the user-defined timing constraints, like clock period, input and output delays.

Chapter 3
Windowed Image Computation

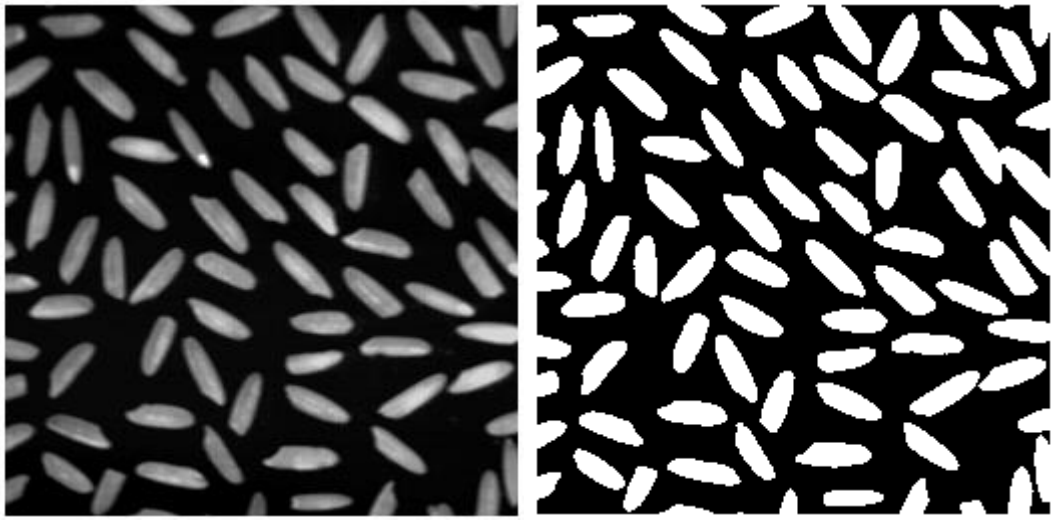
Often, an image is transformed into another image, typically to improve the quality of the image or to implement a more complex process defined in computer vision function. An image processing algorithm takes an image as its input and gives a transformed image as its output. A raw image, for example from a camera, must be processed through a series of steps (such as noise filtering and edge detection) before it can be used in an application. This series of transformations is performed through an image pipeline [11]. As noted in Chapter 2, we consider a pipeline of windowed image computations (or neighborhood operations) in this thesis. Based on the type of transformation, a windowed computation can be classified as point, local or global.

In each of these transformations, an input image I is transformed into an output image J. Let the pixels at position (x, y) be $I(x, y)$ and $J(x, y)$. Each output pixel $J(x, y)$ is a function of a subset of pixels of I. In a point algorithm, each output pixel $J(x, y)$ is dependent only on input pixel $I(x, y)$. For example, in image thresholding, each output pixel is based on the following equation:

$$J(x,y) = \begin{cases} 1, & \text{if } I(x,y) \geq T \\ 0, & \text{if } I(x,y) < T \end{cases}$$

Here, T is the threshold which is the basis for deciding whether the output pixel is a 1 or a 0. Consider the images in Figure 3.1. Figure 3.1(b) is a binary image with values, either 1 or 0 depending on the threshold level of the input image in Figure 3.1(a). These algorithms are a special case of windowed computations.

In a local algorithm, the output pixel depends on a small portion of the input image in the vicinity of the input pixel. Many algorithms, including commonly used filters are a part of this category. This is a widely used algorithm type in many image processing techniques.



(a) (b)

Figure 3.1: Example of image thresholding. Figure taken from [20]

The figures 3.2(a) and 3.2(b) shows the input and output images of a median filter application implemented in Matlab for a software validation of our implementation (see Chapter 5).

Input Output Input Output

(a) 16x16 Image (b) 1024x1024 Image

Figure 3.2 Median Filter Example

At the other extreme, in global algorithms each output pixel depends on a large portion of input image or in the worst case, the complete image. An example in this category is compressing an image using JPEG 2000 [7].

All of the above algorithms (point, local and global) can be expressed as special case of a general framework called *windowed computations*.

3.1 Windowed Computation

The processing of an individual pixel using the values of other pixels in the neighborhood is known as a *windowed operation* (or a neighborhood operation). The size of the "window" depends on the number of neighborhood pixels required for that particular operation. Applying a separate windowed operation on every pixel of the image is known as *windowed computation*.

Let $w \geq 0$ be an integer; we will call this the *window parameter*. Let W be a $(2w + 1) \times (2w + 1)$ matrix called the *window function*; the quantity $(2w + 1)$ will be called the *window size*. The elements of W are denoted by $W(u, v)$, when $-w \leq u, v \leq w$. In general, a windowed operation f performed on a pixel $I(x, y)$ produces an output pixel $J(x, y)$, where

$$J(x, y) = f(W; \{I(x + u, y + v) : -w \leq u, v \leq w\})$$

That is $J(x, y)$ is a function of the window function and the pixel within maximum distance w along the x or y axes from point (i, j). Applying the above function to f to every pixel $I(x, y)$ of the entire image is a windowed computation. Observe that the point algorithm is a windowed computation with $w = 0$ (uses a 1×1 matrix). As another example, if $w = 1, \text{then } J(x, y)$ depends on $(2w+1)^2 = 9$ pixels of the input image I; here the window is a 3×3 matrix and $J(x, y)$ depends on the 9 pixels of the input image I with co-ordinates $(x - 1, y - 1), (x - 1, y), (x - 1, y + 1), (x, y - 1), (x, y), (x, y + 1), (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)$. This is

expressed in Figure 3.3, where the dark colored pixel is subject to transformation depending on the 8 neighboring pixels and the dark colored pixel itself. Often, the function f is a sum of pixels weighted by elements of the window; here the window elements are real numbers. In this case, we have

$$J(x,y) = \sum_{u=-w}^{w} \sum_{v=-w}^{w} (W(u,v)) \cdot (I(x+u, y+v))$$

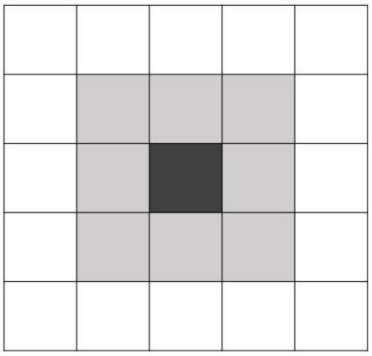


Figure 3.3: Windowed operation of size 3; the dark colored pixel is transformed using its own value and the 8 neighbors (shown shaded)

As an example of this type of computation, consider w =1 (resulting in a 3x3 window) and let each element of the window be 1/9. Then Equation becomes

$$J(x,y) = \frac{1}{9} \sum_{u=-3}^{3} \sum_{v=-3}^{3} I(x+u, y+v)$$

which is simply the average of all 9 pixels in the window centered around $I(x,y)$. This filter is called a median filter (see Figure 3.2).

Neighboring pixel values for border pixels (that may be outside the image) can be considered constant or a simple function of the image within the window. We will ignore them in

this thesis as this static computation will not take up significant FPGA resources. The value of the window parameter w defines the type of computation. For example, $w = 0$ implies a local computation with a 1×1 window matrix and $J(x, y)$ depends only on $I(x, y)$. If w is very large, then we have a global algorithm. Typically, w is small ranging from 0 to 5 for local algorithms. This means that the window is of size 1×1, 3×3, 5×5 and so on. Moreover, image processing is often performed on a pipeline of windowed operations [11].

3.2 Tiles and Windows

Since FPGAs are pin and memory limited (see Chapter 2), an entire image cannot be brought into the FPGA. Consequently, only a small part of the image is processed at a time.

Let the image I be an $N \times N$ array of pixels. Let a small part of the image represented by an $n \times n$ "tile" be brought into the FPGA at a time; here $n \ll N$. Consider a case where the FPGA has z pipeline stages (numbered 0,1,…., z-1) and each stage (for ease of explanation) using a windowed computation with parameter w.

Vaidyanathan *et al.* [9] have provided an analytical framework for scheduling tiles of the image through the pipeline. This approach delays certain elements to ensure a smooth movement of the image through the pipeline and ignores the computational overhead needed to manage this delay. The following example illustrates this.

Figure 3.4 shows the tile sizes (unaltered) for stage 0, 1, 2 of the pipeline. The input to stage 0 is a set of $n \times n$ tiles (25 tiles in the example, each of size $n \times n$, indicated by dotted lines); all tiles are of the same size. Tile 0 produces an output of size $(n - w) \times (n - w)$ (indicated in Figure 3.4 by dashed lines) as the neighboring pixels, for example for $(n - 1, n - 1)$, are not yet available. Thus tile 0 of stage 1 is of size $(n - w) \times (n - w)$. For the same reason, tile 0 of stage

2 is of size $(n - 2w) \times (n - 2w)$. Tiles 1,2,3 are of size $(n - iw) \times n$ for stage i and tile 4 is of size $(n + iw) \times n$.

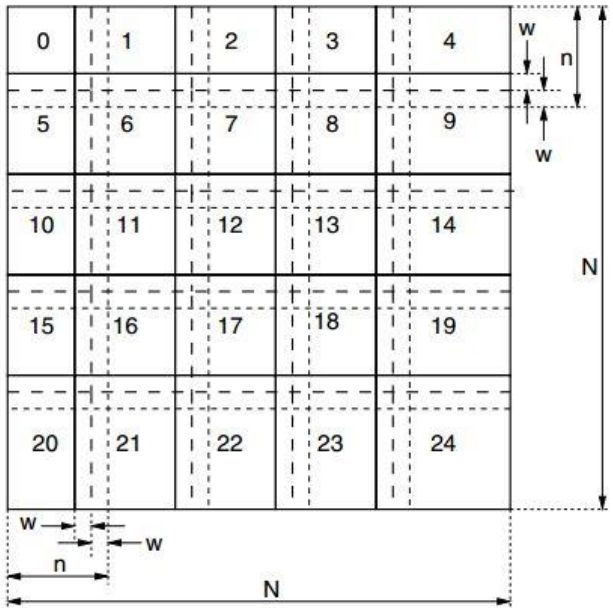


Figure 3.4: Varying output tile sizes (taken from Vaidyanathan *et al.* [9])

Figure 3.5 shows the tile sizes input to the various stages.

	$n - iw$	n	n	n	$n + iw$
$n - iw$	0	1	2	3	4
n	5	6	7	8	9
n	10	11	12	13	14
n	15	16	17	18	19
$n + iw$	20	21	22	23	24

Figure 3.5: Input tile sizes for i^{th} stage

Specifically, at stage $i; 0 \leq i < z$

- tile 0 is of size $(n - iw) \times (n - iw)$
- tiles 1 to 3 are of size $(n - iw) \times n$
- tile 4 is of size $(n - iw) \times (n + iw)$
- tiles $5, 10, 15$ are of size $n \times (n - iw)$
- tiles $9, 14, 19$ are of size $n \times (n + iw)$
- tile 20 is of size $(n + iw) \times (n - iw)$
- tiles $21, 22, 23$ are of size $(n + iw) \times n$
- tile 24 is of size $(n + iw) \times (n + iw)$
- all other tiles $(6, 7, 8, 11, 12, 13, 16, 17, 18)$are of size $n \times n$

Thus, most tiles are of size $n \times n$, but the existence of larger and smaller tiles around the perimeter can stall the pipeline. For example, tile 9, say in stage 1, has size $n \times (n + w)$, whereas tile 10 in stage 1 has size $n \times (n - w)$. So by delaying the processing of nw elements of tile 9 to be with tile 10, both computations can be of "size" $n \times n$, allowing for a smoother pipeline operation.

Consider stages 0 and 1. All tiles of stage 0 are of size $n \times n$ and require cn^2 time (say) to process, where c is some constant. Consider stage 1 and tiles 9 and 10. These tiles are of size $n \times (n + w)$ and $n \times (n - w)$ which requires $c(n^2 + nw)$ and $c(n^2 - nw)$ times, where stage 0 is at tile 10, stage 1 is at tile 9. Stage 0 require cn^2 time whereas stage 1 require $c(n^2 + nw)$. So stage 0 has to wait before it can pass its tile to stage 1. Similarly, when stage 0 is on tile 11 again requiring cn^2 time, stage 1 is on tile 10 requiring only $c(n^2 - nw)$ time and having to wait on stage 0 for cnw time. If stage 1 could defer cnw tiles of the work of tile 9 to be done with tile 10, the pipeline could flow more smoothly. This is one of the main ideas in Vaidyanathan *et al.* [9].

However, this ignores the storage of the nw elements and the reorganization of data required for a static computational fabric that the FPGA provides. To allow for fixed sized tiles, we alter the algorithm of Vaidyanathan *et al.* [9] as follows. The FPGA has an $(n + 2w) \times (n + 2w)$ array to perform a windowed computation of size w on an $n \times n$ tile. The output of this computation corresponds to the $n \times n$ area shown shaded in Figure 3.6.

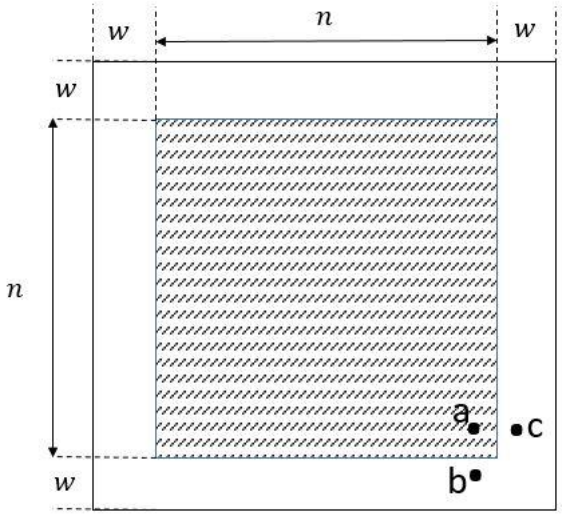
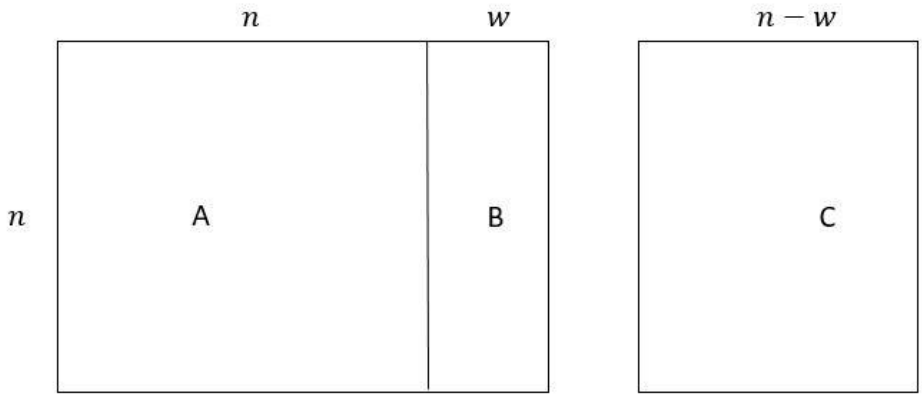


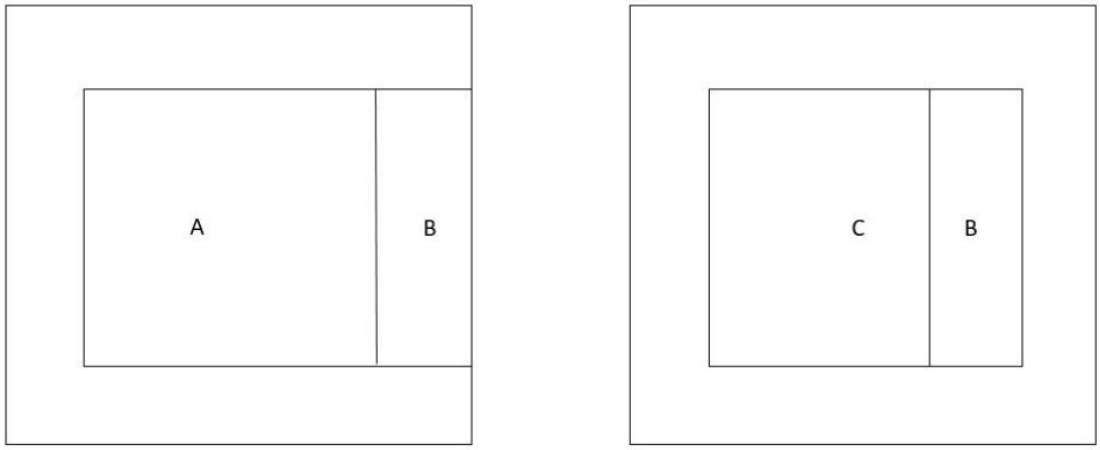
Figure 3.6: The $n \times n$ output from a $(n + 2w) \times (n + 2w)$ array for the windowed computation of a tile.

This array represents a fixed hardware fabric in the FPGA. So before this hardware is triggered, one has to ensure that correct values are "loaded" into the matrix. For example, the pixels at position *b* and *c* must be the neighbors of the pixel at position *a* in the original image.

Now consider tiles 9 and 10 in stage 1 (of size $n \times (n + w)$ and $n \times (n - w)$), respectively with the assumption that an $n \times w$ "chunk" of tile 9 will be used with tile 10.



Tile 9 (a) Tile 10 (b)



(c) (d)

Figure 3.7: Compute Fabric

Figures 3.7 (a) and (b) illustrate the layout of tiles 9 and 10. Processing square A as shown in Figure 37 (c) is fine. However, rectangle B is not a right neighbor of rectangle C and processing cannot be done as in Figure 3.7 (d). Hence additional effort must be used to realign the data. To simplify this, we use an image that is larger than the original $N \times N$ size (padded with dummy elements) to produce tiles of constant size $n \times n$.

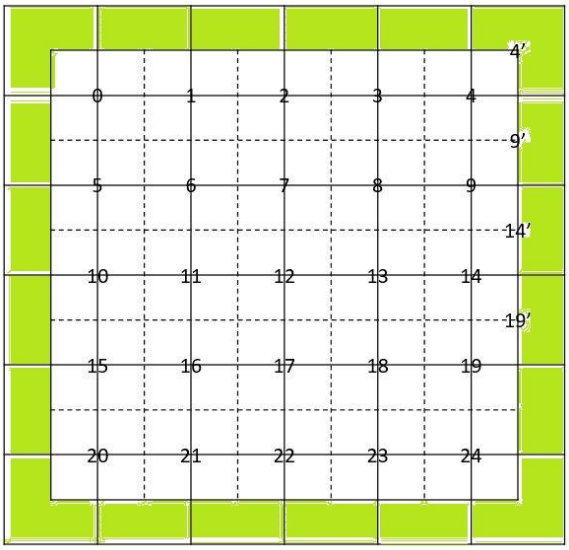


Figure 3.8: Expanded Image of size $Q \times Q$ (shown in solid lines)

As shown in Figure 3.8, we overlay a grid of $n \times n$ tiles on the image to include the dummy values (shown shaded) that need not be received. So the extra work of tile 9 is deferred to a new tile 9' (also of size $n \times n$ but possibly containing some irrelevant values). This maintains a constant tile size, albeit with a larger number of tiles.

3.3 Effect of Expanding the Image

Specifically, we expand the given $N \times N$ image to a $Q \times Q$ image, where $Q = n\left\lceil \frac{N+n}{n} \right\rceil$. This ensures that Q is divisible by n.

Observe that,

$$\frac{Q}{N} = \frac{n}{N}\left\lceil \frac{N+n}{n} \right\rceil \leq \frac{n}{N}\left(\frac{N+n}{n} + 1\right) = \frac{n}{N}\left(\frac{N}{n} + 2\right)$$

$$= 1 + 2\frac{n}{N} \cong 1 \; (as \; n \ll N)$$

Thus, the expansion of the image is not very large. In terms of time, the total number of tiles handled is expanded by

$$\left(\frac{Q}{N}\right)^2 \leq 1 + 4\frac{n}{N} + 4\frac{n^2}{N^2} \cong 1$$

Since z stages with T tiles in an ideal pipeline requires $T + z - 1$ units of time, the extra time needed here is

$$\frac{Q^2}{n^2} + z - 1 - \left(\frac{N^2}{n^2} + z - 1\right)$$

$$= \frac{Q^2 - N^2}{n^2} = \frac{N^2}{n^2}\left(\frac{Q^2}{N^2} - 1\right)$$

$$\leq \frac{N^2}{n^2}\left(4\frac{n}{N} + 4\frac{N^2}{N^2}\right) = 4\frac{N}{n} + 4$$

This quantity could be large as $n \ll N$. However, this analysis assumes no overhead for aligning data in the computation fabric; this, as illustrated in Figure 3.7, is not the case. Therefore, in the FPGA environment, our approach is quite competitive with that of Vaidyanathan *et al.* [9]. A more detailed analysis appears below.

Let $T1$ denote the time to run a z-stage windowed computation of size w on an $N \times N$ image. Vaidyanathan *et al.* [9] show that

$$T1 = \frac{N^2}{n^2}(1 + (\frac{n}{N} + \frac{6n^3}{wN^2} + \frac{zw}{N} + \frac{zn^2}{N^2}))$$

Let T2 be the corresponding time for the enlarged image of size $Q \times Q$, where $Q = n\left\lceil\frac{N+n}{n}\right\rceil$.

$$T2 = z + \left\lceil\frac{(N+n)^2}{n^2}\right\rceil - 1 \leq z - 1 + \left(\frac{N+n}{n} + 1\right)^2$$

We can further simplify this equation as follows:

$$T2 = z - 1 + \left(\frac{N}{n} + 2\right)^2$$

$$= z - 1 + \frac{N^2}{n^2} + 4\frac{N}{n} + 4$$

$$= \frac{N^2}{n^2}\left(z\frac{n^2}{N^2} + 1 + 4\frac{n}{N} + 3\frac{n^2}{N^2}\right)$$

The extra time that we spent in processing the padded tiles can be calculated by subtracting T1 from T2 as follows:

$$\Delta T = T2 - T1$$

$$= \frac{N^2}{n^2}\left(3\frac{n}{N} + 3\frac{n^2}{N^2} - \frac{6n^3}{wN^2} - \frac{zw}{N}\right)$$

$$= 3\frac{N}{n} + 3 - 6n - \frac{zNw}{n^2}$$

While the highest growing term is $3\frac{N}{n}$ (which is only a bit smaller than $4\frac{N}{n}$), in practical case the negative term will further reduce the overhead for our approach.

Chapter 4
Handshaking

In this chapter, we explain the idea of a handshaking protocol and how it is applied in this thesis. Handshaking, generally speaking, is the set of signals and rules used for communication between a sender and a receiver. Handshaking protocols are used in many applications, for example, TCP (3-way handshake) [12], Simple Mail Transfer Protocol (SMTP) [23], TLS handshake [24], WPA2 wireless (4-way handshake) [25] and dial-up access modems.

The role of the handshaking protocol is to ensure proper communication within the given context. For example, the 3-way TCP handshake involves a request from the potential sender, an acknowledgment (ack) from the receiver and an acknowledgment of the acknowledgment. This ensures that (in an environment in which packets could be lost) the sender and receiver are both on the same page. Other protocol details such as timeouts ensure a reliable practical performance.

We now explain handshaking in the context of this work.

4.1 Handshaking in this thesis

In our implementation, we deal with multiple modules in the architecture (see Chapter 5), all of which operate independently and simultaneously. Each module behaves in an event-driven manner. That is, particular processes (such as send, receive or compute) are triggered by particular events. This ensures that all modules can operate independently (and asynchronously). The handshaking protocol serves to synchronize the interaction between them.

Unlike the networking case, our environment is assumed to be free of errors, so acks are not used. Our handshaking is in the nature of “ready” and “done;” that is a sending process indicates that it

is ready to send to a receiver and a receiving process indicates that it is done with receiving. This ensures that a sender will not overwrite receiver data and that a receiver will not consume stale or un-updated data. The data path itself is static, the handshaking simply enables the data to be loaded into a register or latch. Figure 4.1 shows the signals between a sender and a receiver.

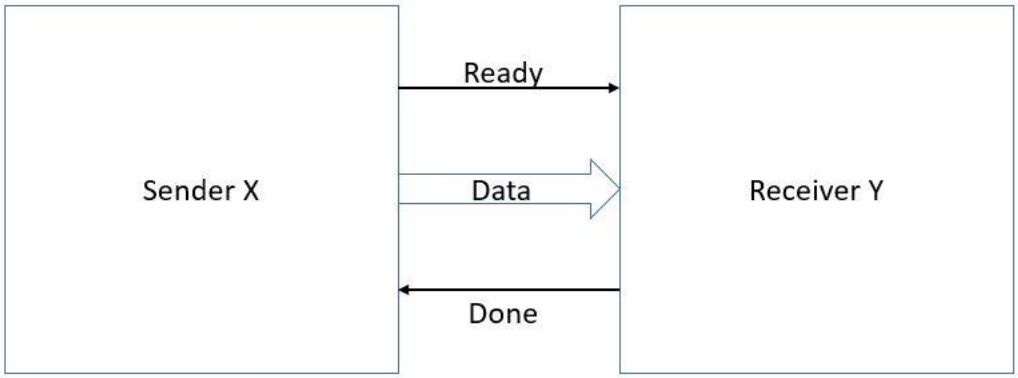


Figure 4.1: Simple Handshaking

In this thesis, we will refer to our 2-way handshaking as the *simple handshaking* protocol. This protocol has been studied before [26]. Other protocols also exist for pipeline control in FPGAs, for example [27]. The simple protocol we use is not particular to pipelines; it can be used in any situation requiring asynchronous communication between modules.

This thesis uses modules configured in the following way:

a) Single sender to single receiver (1-to-1)

b) Single sender to multiple receivers (1-to-many); at most 3 receivers in our case.

c) Multiple senders to a single receiver (many-to-1); at most 3 senders in our case.

These cases require slightly different handling. We now explain these cases with the most detail on the first. Others follow on similar lines.

4.2 Single Sender to Single Receiver (1-to-1 system)

Consider a module X shown in Figure 4.2, that receives data A from module X_A and sends data B to module X_B. Module X acts as both receiver (from X_A) and sender to X_B. The handshaking signals $ready_A$, $done_A$, $ready_B$, $done_B$ are shown in the figure. Signal $done_A$ issued by X indicates to X_A that it is ready to receive the next data from X_A. Signal $done_B$ is similarly generated by X_B. Signal $ready_B$ issued by X inidicate X_B that X is ready with a new data that is available for X_B. Signal $ready_A$ is similarly issued by X_A.

We now examine the action of module X relative to this interaction with module X_A and X_B.

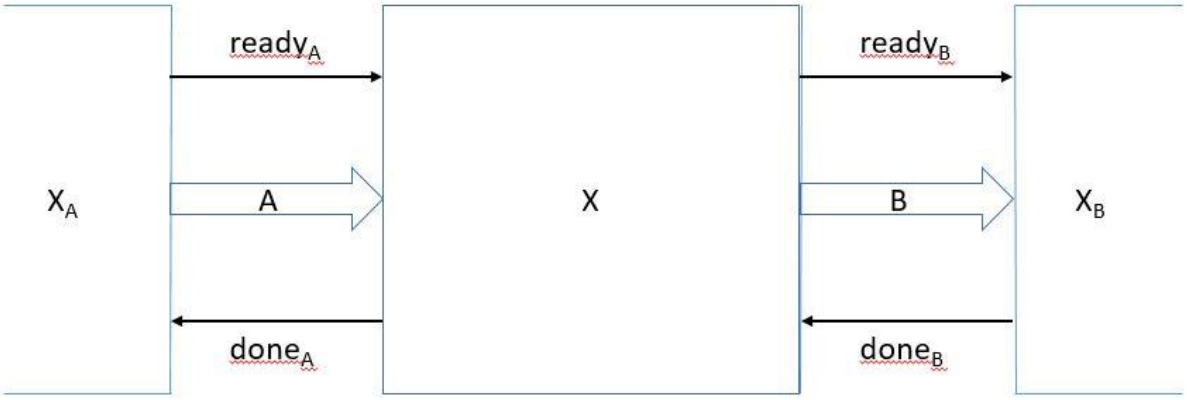


Figure 4.2: Module X, receiving from X_A and sending to X_B.

Each module (including X) goes through a cycle of three phases: receive data, compute, send result. Depending on the module in question, these tasks could take different amounts of time. In some cases, a particular phase may even be "empty"; for instance, a register does not transform the data in any way, so its compute part is empty. The image generator can be thought of as a module with an empty receive phase and the consumer of the image has an empty send phase. Figure 4.3 shows a pseudo-code of the three phases in module X.

Observe that if module X executes Receive, then after exiting Algorithm Receive, it sets Ready_to_Compute to 1. This ensures that it executes Algorithm Compute next. At the end of this algorithm, it sets $ready_B$ to 1, which causes it to execute Algorithm Send. Finally, Algorithm Send sets $done_A$ to 1 which takes the execution back to Algorithm Receive. Also verify that the condition for only one of the algorithms is met at any time. Thus each module operates in a three "state" cycle shown in Figure 4.4.

```
Algorithm Receive
{Execute if done_A = 1}
Wait until ready_A = 1
done_A ← 0
Receive Data  /*enable loading data*/
Ready_to_Compute ← 1
end

Algorithm Compute
{Execute if Ready_to_Compute = 1}
Initiate module's Computation
Wait until module has computed its function
Ready_to_Compute ← 0
ready_B ← 1
end

Algorithm Send
{Execute if ready_B = ready_C = ready_D =1}
Send Data /*enable output data*/
Wait until done_B = done_C = done_D =1
ready_B ← 0, done_A ← 1
end
```

Figure 4.3: Pseudo-code for receive, compute and send phases (1-to-1 system)

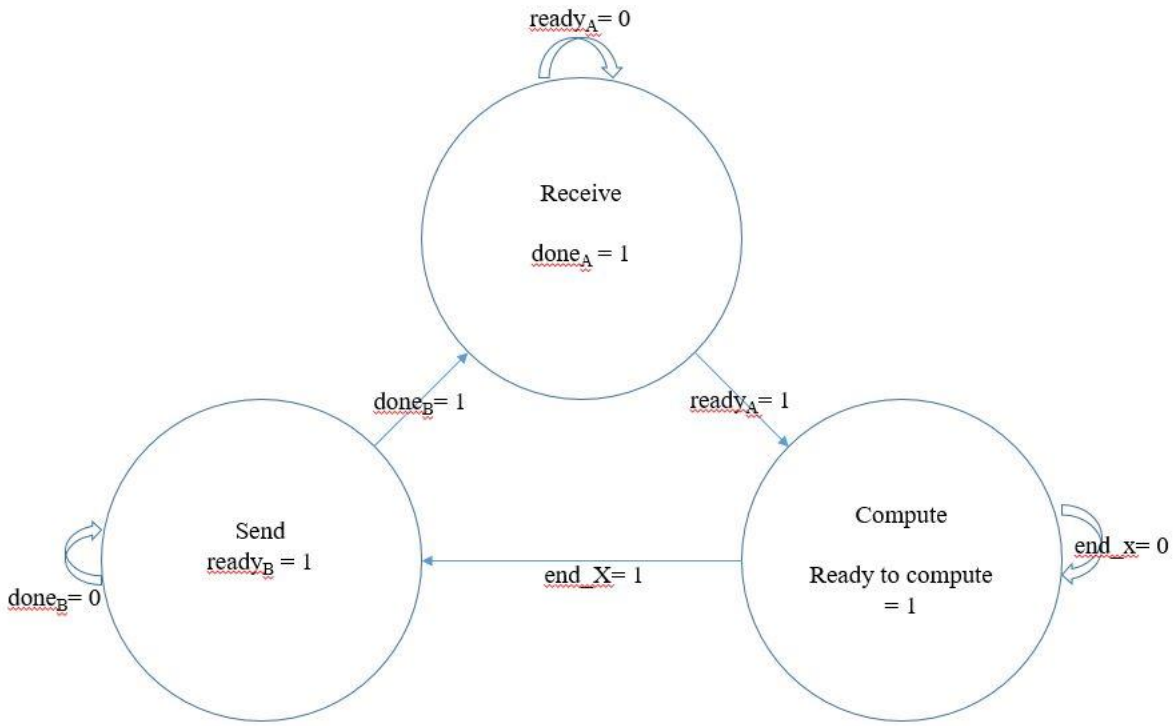


Figure 4.4: Three state cycle

We now argue that the simple handshaking ensures correct operation. Specifically, the following three considerations must be addressed.

a) A receiver does not consume data from a sender until the sender has the latest value on its data line.
b) A sender's data is not overwritten by newly received data until it has been consumed by a receiver.
c) There is no deadlock.

The initialization of the various modules ensures that the safety (a, b) and liveness (c) conditions are met. Specifically, we will assume that modules complete their tasks in finite time including supplying the image to the pipeline and consuming its output.

The 3-state diagram of Figure 4.4 ensures that conditions (a) and (b) are met. To see this, $done_A$ = 0 until module X has sent its data to module X_B. This ensure that any new data from X_A does not arrive before the current data is consumed. Similarly, $ready_B$ = 0 until module X_B has consumed the data sent by module X. This ensure that module X will not request for data to its sender before X_B is done receiving.

At this point, we have established that the simple handshaking works in the 1-to-1 case. We now consider the other cases.

4.3 Single Sender, Multiple Receivers (1-to-many)

The main difference here is that the single sender must wait until all of its receivers have received the data. Let module X send to modules X_B, X_C and X_D with corresponding signal $ready_B$, $done_B$, $ready_C$, $done_C$, $ready_D$, and $done_D$.

Figure 4.5 shows the altered code for send and compute; algorithm receive remains unchanged.

```
Algorithm Receive
{Execute if done_A = 1}
Wait until ready_A = 1
done_A ← 0
Receive Data /*enable loading data*/
Ready_to_Compute ← 1
end

Algorithm Send
{Execute if ready_B = ready_C = ready_D =1}
Wait until done_B = done_C = done_D =1
Receive Data /*enable loading data*/
ready_B ← ready_C ← ready_D ← 0, done_A ← 1
end

Algorithm Compute
{Execute if Ready_to_Compute = 1}
Wait until module has computed its function
Ready_to_Compute ← 0
ready_B ← ready_C ← ready_D ← 1
end
```

Figure 4.5: Pseudo-code for receive, compute and send phases (1-to-many system)

The correctness follows along the same lines as in Section 4.2.

4.4 Multiple Senders and Single Receiver (many-to-1)

Let module X receive data from modules X_{A1}, X_{A2}, and X_{A3}, with corresponding handshaking signals $ready_{A1}$, $done_{A1}$, $ready_{A2}$, $done_{A2}$, $done_{A3}$, and $ready_{A3}$. Here, each sender behaves as before (either sending to a single receiver or multiple receivers). The receiver should ensure that it has received everything before a compute is initiated.

In the multiple receiver case, each receiver receives data at about same time and is instructed by the sender (possibly independently) to use this data. We do not have a situation when a receiver receives the data but another receiver's data is not ready.

The situation with multiple (independent) senders is different. If sender X_{A1} has finished sending its data to X and X_{A2} or X_{A3} have not yet, then X cannot proceed to the compute phase. However, it should not hold X_{A1} from receiving and processing its next data.

To account for this, we will have independent receive algorithms for each sender and all three (in this case) must have been executed before we proceed to the compute phase.

Algorithm Receive i / $i \in \{1, 2, 3\}$ for our example*/*
{Execute if $done_{Ai} = 1$*}*
Wait until $ready_{Ai} = 1$
$done_{Ai} \leftarrow 0$
Receive Data from X_{Ai} */*enable loading data*/*
$A_{iOK} \leftarrow 1$
end

Algorithm Send / $i \in \{1, 2, 3\}$*/*
{Execute if $ready_B = 1$*}*
Wait until $done_B = 1$
*Send Data /*enable output data*/*
$ready_B \leftarrow 0$, $done_{Ai} \leftarrow 1$
end

Algorithm Compute / $i \in \{1, 2, 3\}$*/*
{Execute if $A_{iOK} = 1 \; \forall \; i$*}*
Wait until module has computed its function
$A_{iOK} \leftarrow 0 \; \forall \; i$
$ready_B \leftarrow 1$
end

Figure 4.6: Pseudo-code for receive, compute and send phases (many-to-1 system)

Notice here, that Receive *i* (for multiple values of *i*) could be executing concurrently. The order in which X executes each of them is immaterial. Again, as in Section 4.2, this method works correctly.

Chapter 5
The Pipeline Architecture

In this chapter, we discuss the structure and the components of the windowed computation pipeline that we have implemented for this thesis. The pipeline consists of a series of z stages each representing a windowed computation of size $(2\mathrm{w}+1)$. As noted in Section 3.2, the windowed computation is performed over a padded image of size $Q \times Q$. This image is brought into the FPGA as tiles each of size $n \times n$ in a row major order. Each stage of the pipeline accepts input and output tiles of size $n \times n$.

We begin with the overall architecture needed for the windowed computation (Section 5.1). This includes the FPGA and all external entities it interacts with. Next, in Section 5.2, we will describe the pipeline and a key interfacing module that are all internal to the FPGA. In Section 5.3, we will expand each pipeline stage and describe the functionality of its components. Section 5.4 is devoted to a memory interface module needed to co-ordinate the interaction between pipeline stage (in FPGA) and external memory.

Recall that independent modules communicate using the simple handshaking protocol (Chapter 4). The overall structure of the windowed computation pipeline architecture is as shown in Figure 5.1.

As shown in Figure 5.1, the system receives an input image, typically from a camera or some other image source, into the FPGA (a tile at a time) and the processed final image (again, a tile at a time) is consumed by an image consumer such as a monitor or a TV. Generally, the image source is an online source like a camera and its output is available only once. If portions of the

image are required for later use (as is the case here), we need to store the image for subsequent processing. The external memory in Figure 5.1 stores parts of the image needed between iterations of our algorithm which is brought in correctly when requested. A separate Memory Management Unit (MMU) associated with external memory is assumed. While we do not implement the external memory or the MMU, its functionality is discussed in Section 5.4.

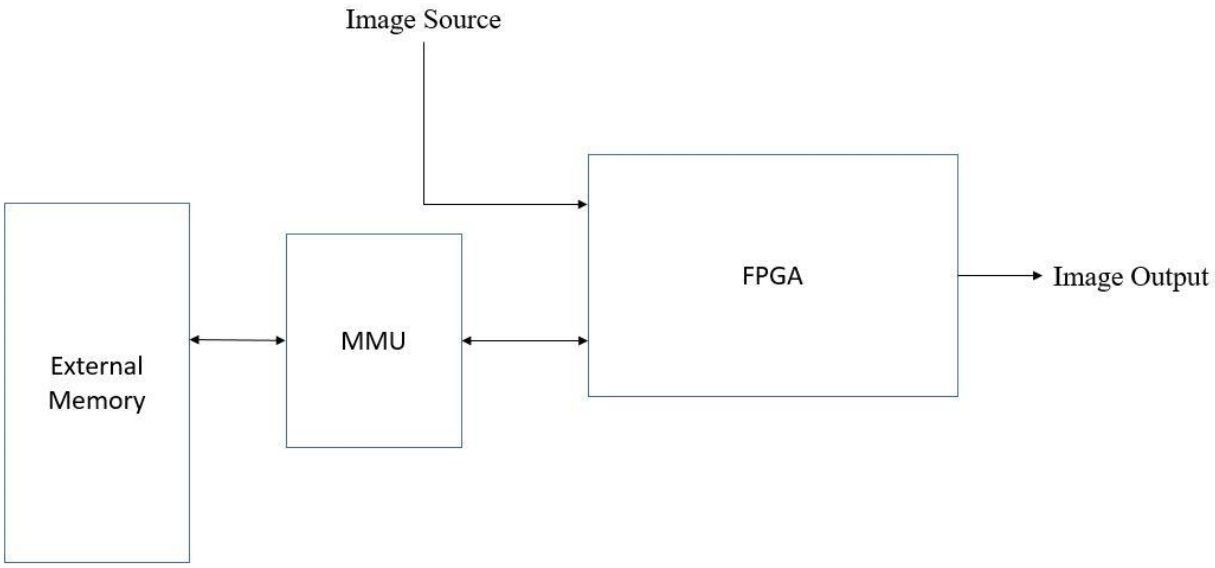


Figure 5.1: Basic Architecture

We now describe the need for such an external memory.

5.1 Memory Requirement and Overall Architecture

Recall that the modified image is of size $Q = n\left\lceil\frac{N+n}{n}\right\rceil$ (see Section 3.3) and that each tile is an $n \times n$ array; w is the window parameter, resulting in a $(2w+1) \times (2w+1)$ window around each pixel. Thus the image has $\left(\frac{Q}{N}\right)^2$ tiles denoted by $\tau_{i,\,j}$ (where $0 \leq i, j < \frac{Q}{n}$) and numbered in

row major order from 0 to $\left(\frac{Q}{n}\right)^2 - 1$; that is, $\tau_{i,j}$ is tile $i\left(\frac{Q}{n}\right) + j$. Recall also that the tile numbering reflects the order in which the tiles are input and output across the pipeline.

Thus when tile u is received, all tiles $0, 1, 2, \ldots, n-1$ have also been received. Put differently, if $\tau_{i,j}$ is received when $\tau_{u,v}$ have been received for all $0 \leq u \leq i$ and $0 \leq v \leq j$. In terms of a tile, not all elements of a currently received tile can be processed as some of the pixels near the border require knowledge of subsequent pixels (that will be received later).

Consider Figure 5.2. The figure shows the currently received tile $\tau_{i,j}$ hatched from south west to north east. However, the part that can be processed is shown hatched from north west to south east.

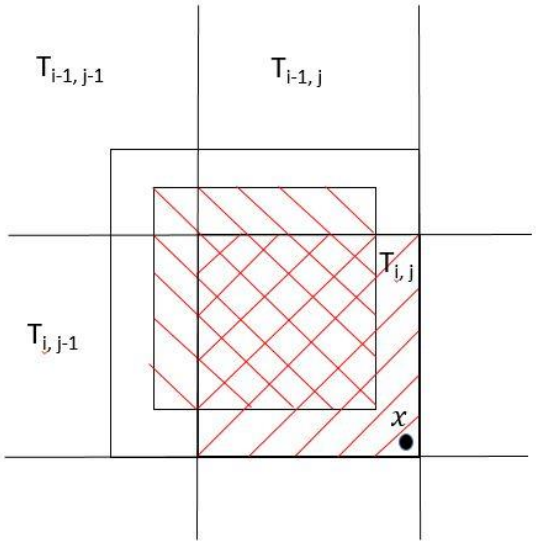


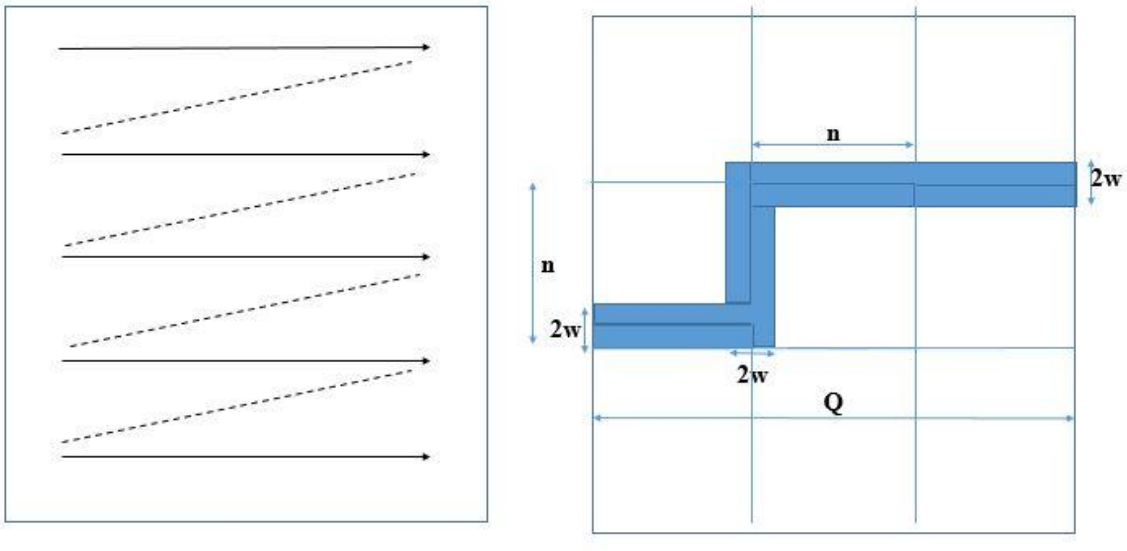
Figure 5.2: Received tile ($\tau_{i,j}$) shown hatched from south west to north east; processed tile shown hatched from north west to south east.

This is because, for example, the bottom rightmost pixel x of $\tau_{i,j}$ needs pixels from $\tau_{i,j-1}$, $\tau_{i-1,j}$, $\tau_{i-1,j-1}$ before it can be processed (assuming $w \geq 1$). Thus the portion of the image that needs to be

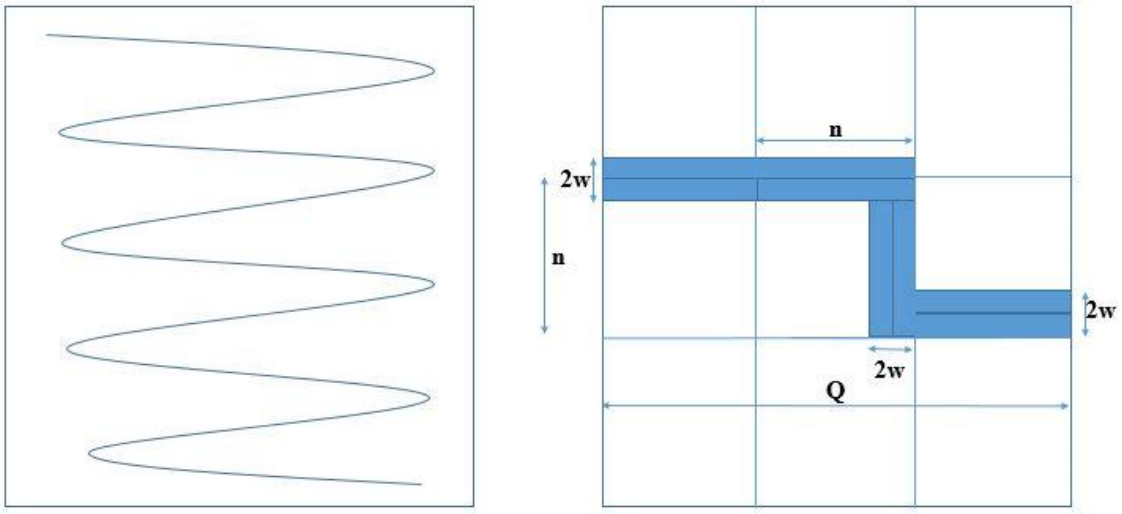
stored for future use is as shown in Figure 5.3; this has also been elaborated upon in Vaidyanathan *et al.* [4]. The memory requirement is $2Nw + 2wn$. Given that $N \gg n$, it is unreasonable to find enough storage within the FPGA. This is the reason for the external memory. In fact, of the two components of this storage, the $2Nw$ part is stored in the external "horizontal" storage and the $2wn$ part ("vertical storage") is stored within the FPGA in its Block RAM.

The next question is whether a different order of inputing tiles would reduce the storage requirement. We conjecture that it would not as is evidenced in the example below.

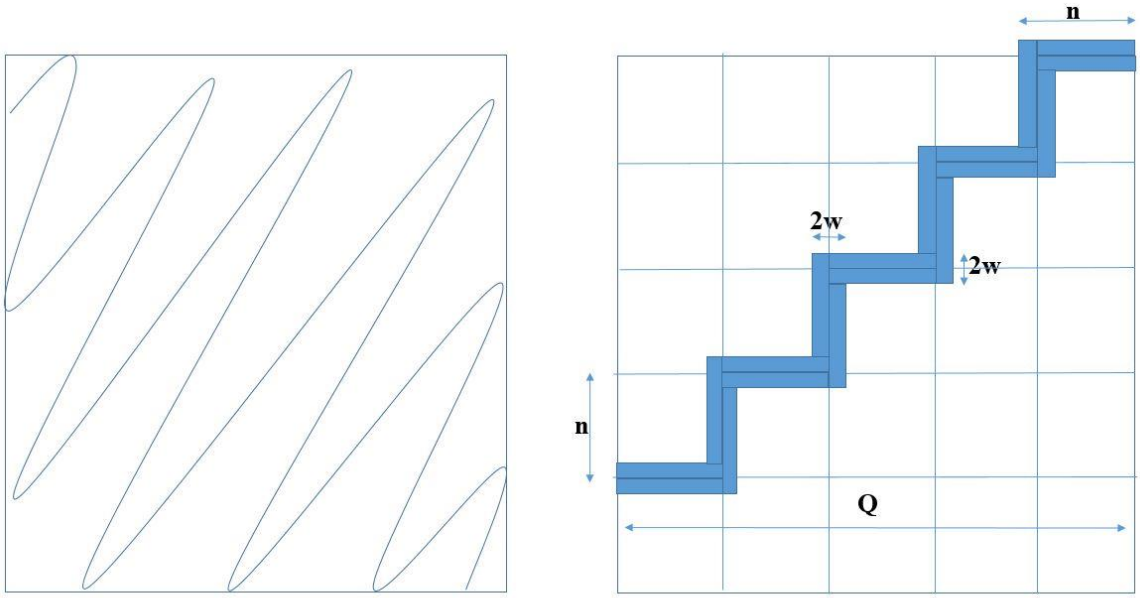
In snake-like row major (see Figure 5.3 (a)), the tiles are proximate to each other (no jump from last tile of a row to first tile of next). However, here too, the memory requirement is proportional to Qw (see Figure 5.3 (b)). Other tile ordering such as the diagonal snake-like row-major ordering or other proximity ordering such as the Z-ordering [10] will also require $\theta(Qw)$ memory (see Figures 5.3 (c), (d)).



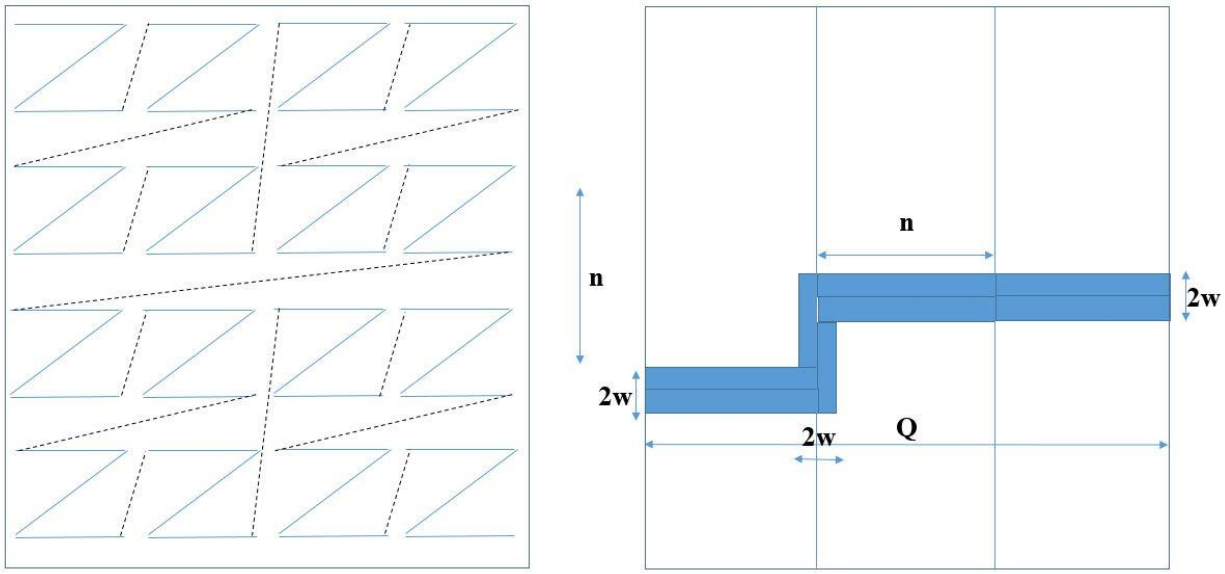
(a) Row-major ordering of tiles



(b) Snake-like row-major ordering of tiles



(c) Diagonal snake-like row-major ordering of tiles



(d) Z-ordering of tiles

Figure 5.3: Tile Ordering

Thus, as seen from the above examples, an external memory is required. We now detail the components of the architecture.

The nature of a windowed computation is such that, computing the new value of a pixel requires the value of the surrounding pixels within the $(2w + 1) \times (2w + 1)$ sized window. The image pipeline, as expected, strings the z stages (Stage 0 to Stage $z - 1$) together as shown in the top half of Figure 5.6. As discussed previously in this chapter, processing tile $\tau_{i,\,j}$ requires information from tiles $\tau_{i,\,j-1}$, $\tau_{i-1,\,j}$, $\tau_{i-1,\,j-1}$. Some of this information ($\tau_{i,\,j-1}$) is available locally within the FPGA (described further in Section 5.3). The remainder, $\tau_{i-1,\,j}$ and $\tau_{i-1,\,j-1}$ is in the external storage. Likewise, parts of the current tile $\tau_{i,\,j}$ must be saved in the external storage for later use. This applies to every tile of every stage. Within a stage, previous tiles are requested in succession, but across stages, these requests could be simultaneous. The memory interface acts as the go

between among the stages and the external memory. To conserve pins (a scarce resource), the memory interface transacts data for only one stage at a time.

Before we proceed, let us see what happens when tile $\tau_{i,j}$ is received. To process elements of this tile, portions of tiles $\tau_{i,j-1}$, $\tau_{i-1,j-1}$ and $\tau_{i-1,j}$ are required. In addition, a portion of $\tau_{i,j}$ itself must be stored for future use. Of the three tiles other than $\tau_{i,j}$ needed for current computation, call the data from $\tau_{i,j-1}$ as the *vertical data* and the remaining data from tiles $\tau_{i-1,j-1}$ and $\tau_{i-1,j}$ as *horizontal data* (see Figure 5.4). Similarly, $\tau_{i,j}$, has a horizontal and vertical part (not necessarily disjoint) that needs to be stored for the future (shown as in Figure 5.4). We will store horizontal and vertical data in horizontal and vertical memories that are outside and inside, respectively of the FPGA.

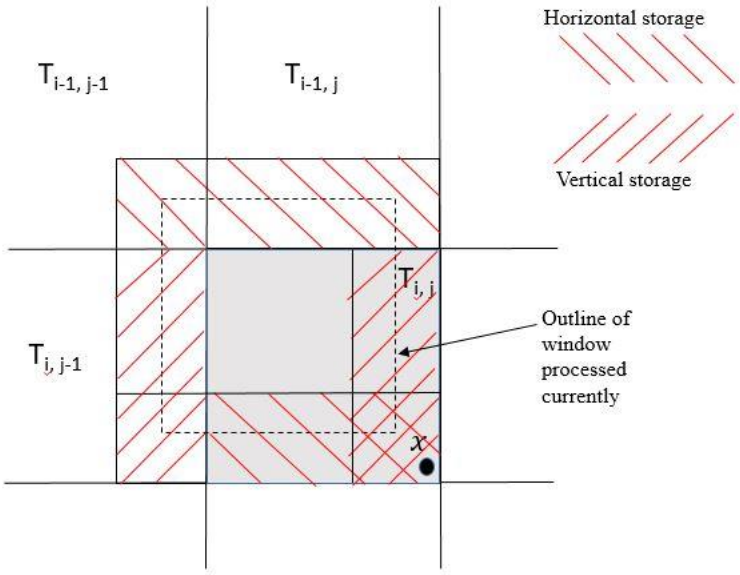


Figure 5.4: Pictorial representation of the horizontal and vertical data inside the compute fabric.

We now describe the overall functionality in some detail (see Figure 5.5). The figure shows (in red) the handshaking signals between the module; these are in Chapter 4 and we do not discuss them here. The data signals are in black and additional (non-data, non-handshaking) information is conveyed through lines in blue.

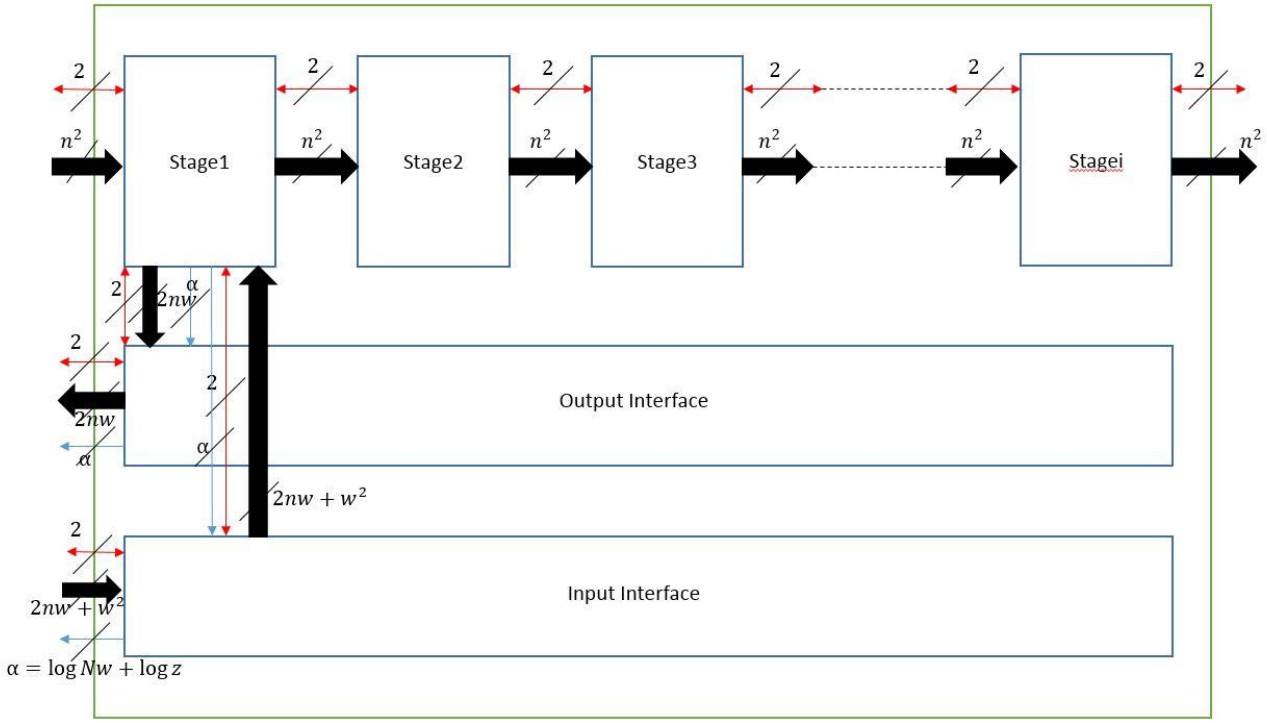


Figure 5.5: A schematic of the memory interface. Handshaking signals are red.

Each stage receives an $n \times n$ tile from the previous stage (or image source). It outputs an $n \times n$ processed tile to the next stage (or image consumer). A stage also interacts with the external memory through the memory interface (which we have segmented into the input and output interfaces in Figure 5.5). When a stage wishes to obtain horizontal data, say of tile $\tau_{i-1,\,j-1}$ and $\tau_{i-1,\,j}$, it sends a request (using handshaking) to the input interface. This request includes the tile number (i, j) and the stage number and is of length $\log(z) + \log(2Q) = \log(z) + \log Q + 1 = \alpha$ (say).

The input interface picks up the requests (one at a time) and transmits them to the MMU, which translates each request to a memory address and accesses the data. This data is sent back to the input interface along with the α-bit information that came with the request. The return data has size $2nw + w^2$ bits. The input interface passes this on to the correct stage (using the α-bit information). Notice that, the α-bit information is passed through bidirectional pins.

Similarly, when horizontal data for $\tau_{i,\,j}$ is exported to the memory, the stage sends α bits of identifying information and the data to the output interface, which passes it on to the MMU.

5.2 Structure within the FPGA

The Figure 5.6 gives more details on how individual stages are arranged inside the FPGA. It also illustrates a memory interface that allows stages to access the external memory.

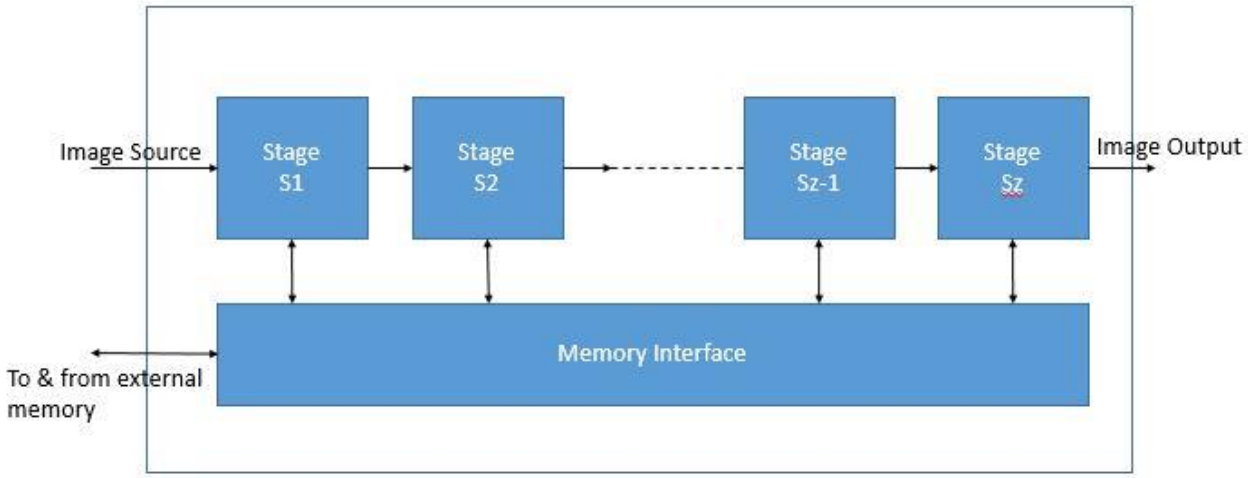


Figure 5.6: Windowed Computation Pipeline

An image processing algorithm usually is implemented through a number of stages where the size and value of the window function could change between stages according to the nature of the algorithm. Figure 5.7 shows the structure of a single stage with just the modules that deal with data (without control modules).

From Figure 5.7, we estimate the FPGA resources that are required to implement the design in any FPGA. To begin with, we estimate the number of I/O pins that are required to bring the input tile into the FPGA, the processed tile to be taken out of the FPGA and the additional pins that are needed for each stage to send their storage data out through the memory interface.

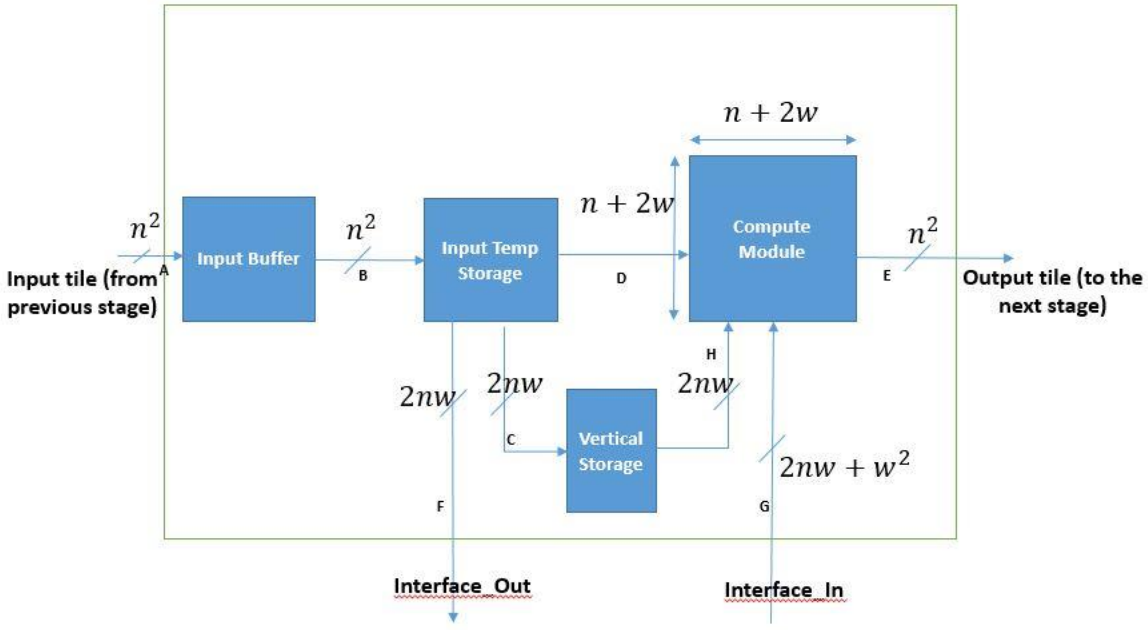


Figure 5.7: Windowed Computation Stage

Total number of pins required, $P =$ Number of input pins + number of output pins + Number of address and data pins needed for the memory interface.

$$P = n^2 + n^2 + \left((2nw + 2nw + w^2) + (2\log Nw) + (2\log z)\right)$$

$$= 2n^2 + 4nw + w^2 + 2\log N + 2\log w + 2\log z$$

In the above equation, the term $(2\log Nw)$ represents the address for the interface and $2\log z$ represents the stage number which is requesting for priority to send its storage data out. From the equation, it is evident that the total number of pins that are required for the windowed computation pipeline to be implemented in an FPGA is proportional to n^2 (square of the tile size).

We also try to estimate the amount of distributed memory required to implement our design. The two modules inside a stage which will be implemented using a distributed memory are the temporary input storage and the storage in the compute module. In each stage, the amount of distributed memory needed will be the sum of the sizes of both the modules. Assuming that these two modules are the main part which requires distributed memory, we estimate the distributed memory utilization to be:

$$DM = \text{Number of stages} * (\text{Memory required for input temp storage} + \text{memory required for compute module})$$

$$= z * (n^2 + (n + 2w)^2)$$

Finally, block RAM is another resource whose utilization has to be estimated. From Figure 6.4, the modules Input Buffer and Vertical Storage will be implemented using a dedicated block RAM (BRAM). In each stage, the amount of block RAM that is required to implement the design will be the sum of the size of both the modules. The estimate of utilization of BRAM inside the FPGA is:

$$BM = \text{Number of stages} * (\text{Memory required for input buffer} + \text{memory required for vertical storage})$$
$$= z * (n^2 + 2nw)$$

We now get into the implementation details of the modules in each stage (Section 5.3) and the memory interface for the pipeline (Section 5.4).

5.3 Pipeline Stage

Figure 5.8 shows the modules within each stage of the pipeline. The functionality of the modules shown in the figure are as follows:

a) Input Buffer module is an $n \times n$ memory which is used to store the data coming from the previous stage. In the case of the first stage in the pipeline, this data comes from outside the FPGA; in fact, this is the main image data entering the FPGA and the main contributor to input pin usage. The input buffer module is implemented to infer to the FPGA's block RAM resource.

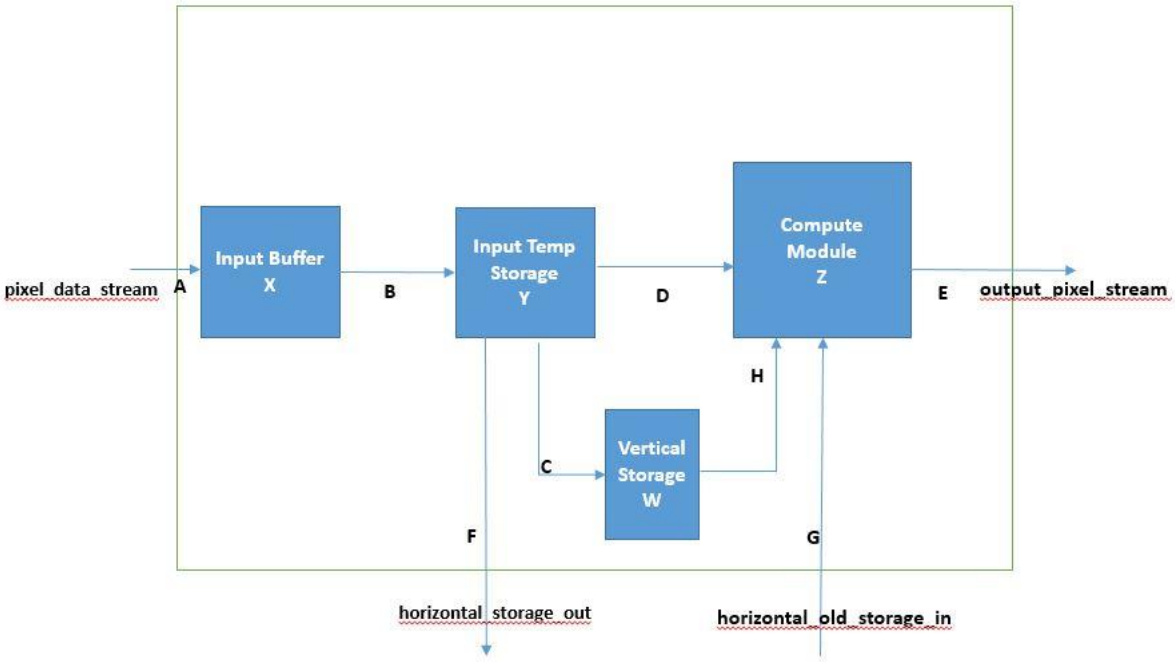


Figure 5.8: Basic Stage

(b) Temp Storage module is created to route parts of the input tile data into different modules (vertical storage, compute module and output interface). It is implemented in the distributed RAM.

(c) Vertical Storage module stores the vertical data from the present tile for use in the next iteration. This is implemented in the BRAM.

(d) Compute module – This is the heart of the windowed computation in which the window is applied to tile of pixels (see Equation 3.1). We assume a median filter (only for illustration purposes); a different operation can also be implemented, as our simulations (Chapter 6) indicate a lot of spare CLB resources.

Figure 5.9 shows the structure of the Compute unit for $w = 1$.

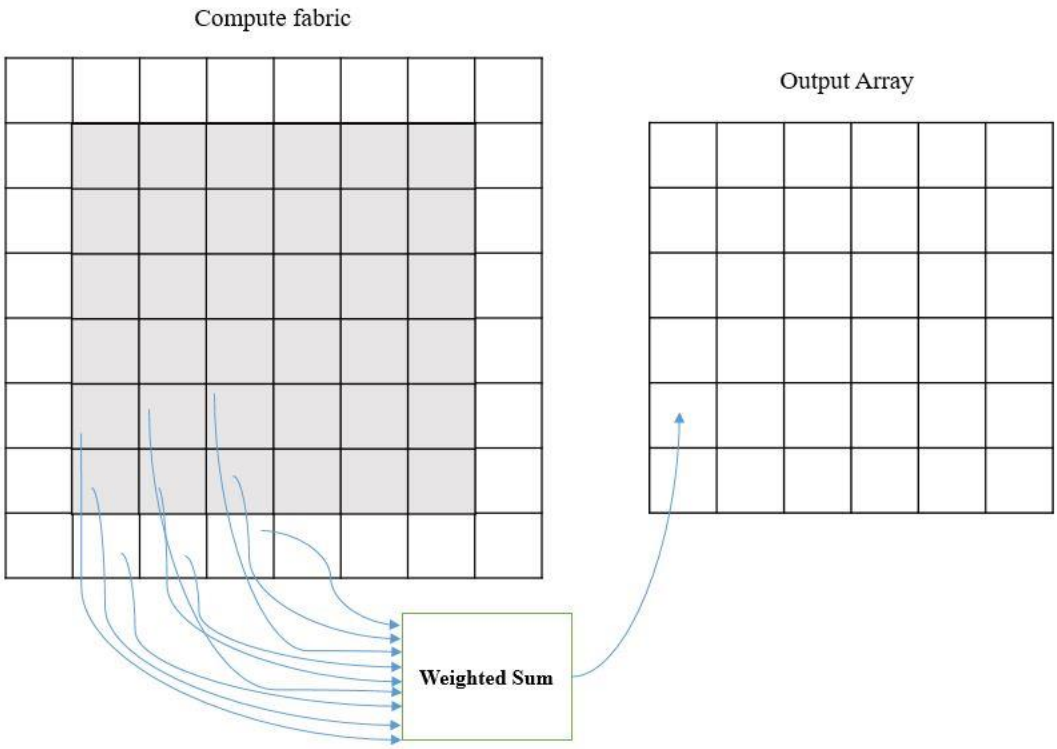


Figure 5.9: Compute fabric structure for $w = 1$

The memory is in the distributed RAM of the FPGA and the computing hardware (weight same in our case) is implemented in the CLBs.

5.4 Memory Interface

Each stage in the pipeline outputs horizontal data to and inputs old horizontal data from the external memory. Due to a limitation in the I/O pin count for the FPGA, we cannot let each stage output/input its horizontal data at the same time. We use a priority encoder to prioritize which stage gets to transact its horizontal data in and out of the device.

The idea is to prioritize request from earlier stages. This is because, without activity in the earlier stage, the later stages cannot proceed. However, our simple handshaking scheme ensures that an earlier stage will not proceed until its output has been consumed by the next stage. Thus this

approach is self-regulating and prevents earlier stages (with higher priority) from monopolizing the memory interface bandwidth.

Figure 5.10 shows the truth table for a simple priority encoder with z stages.

Request from stage s						Granted Stage	Valid
0	**1**	**2**	**3**	.....	z	0	1
1	X	X	X	.....	X	0	1
0	1	X	X	.....	X	1	1
0	0	1	X	.....	X	2	1
0	0	0	1	.....	X	3	1
.....	.....	.....	.....	.....	.....	.....	.....
0	0	0	0	.....	1	z	1
0	0	0	0	.....	0	X	0

Figure 5.10: Truth table for simple priority encoder

A valid bit indicates that whether there is a current request.

Memory Management Unit

As noted earlier, we do not implement the external memory or the memory management unit (MMU). Here we indicate how the MMU would work.

The input to the MMU can either be from the input or output interface. An input from the input interface includes a request for data from tiles $\tau_{i-1, j-1}$ and $\tau_{i-1, j}$ for stage s. This is conveyed as number i, j and s. The MMU translates this information to retrieve $2nw + w^2$ pixels and sends it back to the input interface.

Likewise, an input from the output interface has i, j, v and $2nw$ pixels of data then it needs to store in a location corresponding to tile $\tau_{i, j}$. We now explain how this could be done in the MMU.

When i, j, s is sent, it saves the duplicate part and the rest in clearly identified places. Let these go to locations $hor(i, j, s)$ and $dup(i, j, s)$. For example, the MMU could compute or store in a table, the starting address $hor(i, j, s)$ and the block size $(n - 2w) * 2w$ for this case. When a request i, j, s arrives, the MMU sends data from $hor(i - 1, j, s)$ (of size $(n - 2w) * 2w$) and from $dup(i - 1, j - 1, s)$ (of size $2nw$).

Chapter 6
Simulation Results

So far we have designed a broad approach to running a z-stage windowed computation pipeline in an FPGA (or any other pin-limited environment). To measure the effectiveness of our approach and to obtain better insight into it, we implemented our architecture on a commercially available FPGA (through CAD tools) and observed the amount of resources used for various problem sizes. This chapter reports the results of these simulations.

The resources studied are the following:

a) I/O Banks (IOBs) or Pins.

b) Distributed memory (LUTs as memory).

c) LUTs (a reflection of CLB usage).

d) B-RAMs

e) Operating clock frequencies.

As noted in Chapter 2, the first four are the main resources that constrain FPGA-based design. The operating clock frequency is a good indication of the speed of the pipeline.

Design Methodology

The parameters that affect the performance of the pipeline include:

(a) N = image parameter

(b) n = tile parameter

(c) w = window parameter

(d) z = number of stages

(e) FPGA type.

For this study, we used two FPGA types (Artix-7 and Kintex-7); these were the largest two accessible through the free distribution of the Vivado HL Webpack software.

In general, we use an image with $N \cong 1024$ and kept this fixed for all our experiments. As noted earlier (Chapter 5), the value of N only affects the number of IOBs used, that too logarithmically. Therefore, not much insight would be gained by varying N.

The following methodology was used to run several simulations.

For a fixed FPGA, $N \cong 1024$ and a fixed value of w:

1. The largest value of n, with z = 1 for which a feasible implementation was possible was determined. This n is called $n_{\max}$, the largest tile parameter possible (due to pin limitations).
2. Next, the value of z was increased from 1 to $z_{\max}$, the largest number of stages that could be accommodated; $n = n_{\max}$ for all these values of z.
3. Finally, z was increased beyond $z_{\max}$ and n reduced from $n_{\max}$ to accommodate the increase in z. This was continued until $n \geq 2w$.
4. Each of these was implemented for optimized clock. We examined two situations, one with z = 1 and the other with $z = z_{\max}$. In both cases, $n = n_{\max}$. The maximum attainable clock frequency for $z = z_{\max}$ and z = 1 are denoted by $f_{\min}$ and $f_{\max}$, respectively.

For all these runs, the resource utilization report provided by Vivado tool was used to observe the resource usage. Table 6.1 shows the different values of window parameter (w) against, the maximum values of n, z possible for that particular w. It also shows $f_{\max}$ and $f_{\min}$, the maximum frequency of the clock and minimum frequency of clock possible for a particular window

parameter. Each of these parameters, $n, z, f_{\max}$ and $f_{\min}$ is plotted against the corresponding w as shown in Figures 6.1, 6.2 and 6.3.

w	nmax	zmax	fmax (MHz)	fmin (Mhz)
0	15	6	20.00	18.87
1	14	5	18.51	17.86
2	13	5	16.67	16.13
3	11	5	15.87	15.63
4	11	5	15.38	15.15

Table 6.1: Limiting values for Artix-7

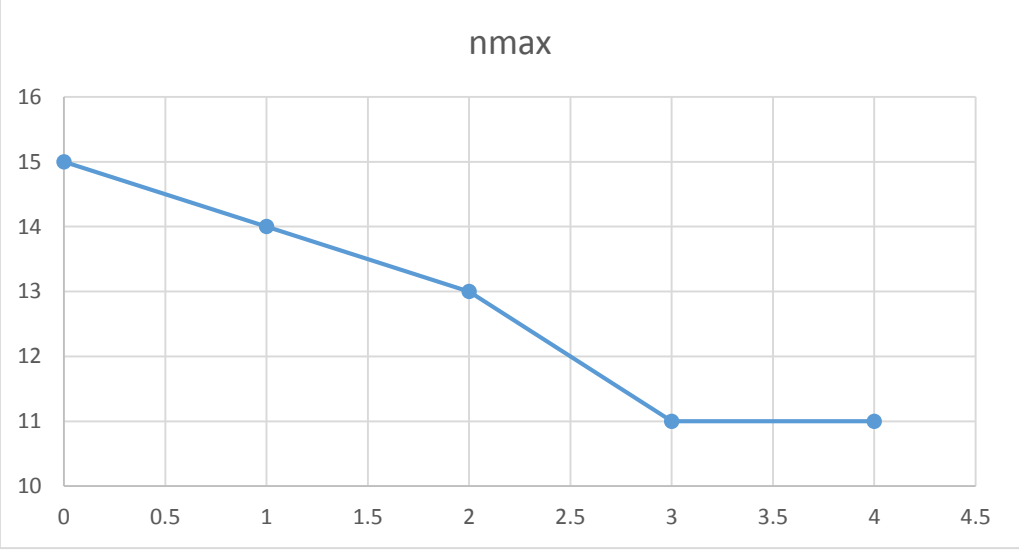


Figure 6.1: Window Parameter (w) in x-axis vs maximum Tile Size ($n_{\max}$) in y-axis for Artix-7

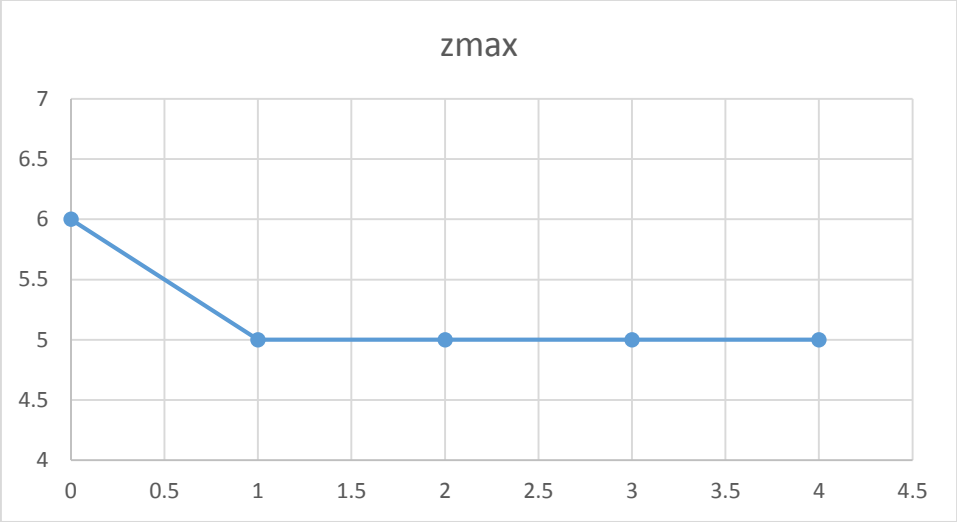


Figure 6.2: Window Parameter (w) in x-axis vs maximum no. of stages (z_{max}) in y-axis for Artix-7

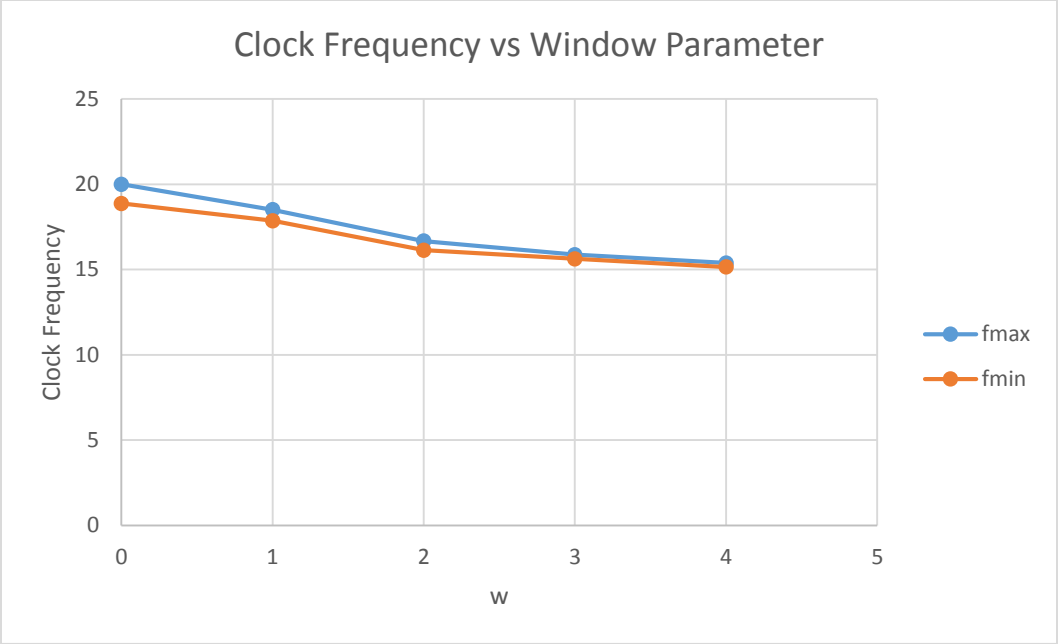


Figure 6.3: Window Parameter (w) in x-axis vs clock frequency (f_{max} and f_{min}) in y-axis for Artix-7

Table 6.1 (and Figures 6.1 – 6.3) show the following. The tile parameter $n_{\max}$ reduces quite markedly from $w = 0$ to $w = 4$. However, the maximum number of pipeline stages, $z_{\max}$, does not vary much. This indicates a higher sensitivity of the design to I/O resources (as opposed to resources within the FPGA); this observation is further in the data presented next.

There is little difference between $f_{\max}$ and $f_{\min}$, indicating that the current "path length" does not depend much on the number of stages. Thus any improvement in the design of a stage would not have a multiplication effect on a large number of stages (other than the parallelism due to a deep pipeline).

Each of the modules of Figure 5.8 operates in one clock cycle. The flow of information between modules is defined by the handshaking signals. Normally, the following sequence of actions could happen.

- At time $t = 0$, data arrives at the input buffer.
- At $t = 1$, it is available to the temporary storage.
- At $t = 2$, it is available at the vertical storage and the compute module. It is also sent to the external memory.
- At $t \leq 2$, horizontal data is available to the compute module.
- At $t = 3$, the compute module produces the output.
- At $t = 3$, the new vertical storage data is.

Thus stage produces an output every 3 clock cycles. With z stages, the time to produce the tiles of an $N \times N$ image is $3*(\frac{Q^2}{n^2} + z - 1)$ clock cycles. Even with a safety factory of 5, we have $15*\left(\frac{Q^2}{n^2} + z - 1\right)^2 \cong 15*\left(\frac{N}{n}\right)^2$ clock cycles to process the image.

With $N \cong 1000$, $n \cong 10$ and $f \cong 15$ Mhz, the image can be processed in

$$15 \cdot \left(\frac{1000}{10}\right)^2 \cdot \frac{1}{15\times10^6} = \frac{1}{100} \text{ seconds, or}$$

At the rate of 100 picture frames/sec where each picture is of size 1000×1000 pixels. This is a reasonable rate for most real time processing scenarios.

Tables 6.2 – 6.6 shows the absolute, percentage and normalized resource utilization for various window parameters, ranging from 0 to 4 for the Artix-7 FPGA. The absolute resource utilization number is the exact number as provided by the Vivado tool, whereas percentage utilization is the percentage of resource utilized from the maximum that is available for the particular FPGA, the normalized utilization is the ratio of absolute value to the maximum of all absolute values among the cases considered.

		LUTs (Total = 133800)			Distributed Memory (Total = 46200)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	15	10920	8.16	0.155	7594	16.43	0.165
2	14	22394	16.73	0.318	14874	32.19	0.324
3	14	34004	25.41	0.483	22800	49.35	0.497
4	14	45326	33.87	0.644	30004	64.94	0.654
5	14	55842	41.73	0.793	38120	82.51	0.831
6	14	70384	52.6	1.000	45829	99.19	1.000

		BRAMs (Total = 365)			IOBs (Total = 500)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	15	4	1.09	0.210	500	100.0	1.000
2	14	7	1.91	0.368	444	88.8	0.888
3	14	11	3.01	0.579	446	89.2	0.892
4	14	14.5	3.97	0.763	446	89.2	0.892
5	14	16	4.38	0.842	448	89.6	0.896
6	14	19	5.20	1.000	448	89.6	0.896

Table 6.2: Table of resource utilization for window parameter w=0 for Artix-7; the last row in each table corresponds to the largest implementable value of z.

		LUTs (Total = 133800)			Distributed Memory (Total = 46200)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	14	11850	8.85	0.207	7704	16.67	0.198
2	14	23439	17.51	0.410	15408	33.35	0.396
3	14	35028	26.18	0.613	23112	50.02	0.594
4	14	46764	34.95	0.819	30816	66.70	0.791
5	13	57052	42.64	1.000	38920	84.24	1.000

		BRAMs (Total = 365)			IOBs (Total = 500)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	14	4.5	1.09	0.272	496	99.2	0.992
2	14	8	1.91	0.484	498	99.6	0.996
3	14	11.5	3.01	0.696	500	100.0	1.000
4	14	15	3.97	0.909	500	100.0	1.000
5	13	16.5	4.38	1.000	446	89.2	0.892

Table 6.3: Table of resource utilization for window parameter w=1 for Artix-7.

		LUTs (Total = 133800)			Distributed Memory (Total = 46200)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	13	12402	9.27	0.204	7896	17.09	0.193
2	13	24286	18.15	0.399	15792	34.18	0.386
3	13	36164	27.03	0.595	23688	51.27	0.580
4	13	48132	35.97	0.792	31584	68.36	0.773
5	13	60090	44.91	0.989	39480	85.45	0.967
6	12	60764	45.41	1.000	40824	88.36	1.000

		BRAMs (Total = 365)			IOBs (Total = 500)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	13	5	1.37	0.263	492	98.4	0.988
2	13	8.5	2.33	0.447	494	98.8	0.992
3	13	12	3.29	0.631	496	99.2	0.996
4	13	15.5	4.25	0.816	496	99.2	0.996
5	13	19	5.20	1.000	498	99.6	1.000
6	12	19	5.20	1.000	442	88.4	0.887

Table 6.4: Table of resource utilization for window parameter w=2 for Artix-7.

		LUTs (Total = 133800)			Distributed Memory (Total = 46200)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	12158	9.08	0.176	7740	16.75	0.172
2	11	23862	17.83	0.346	15480	33.50	0.344
3	11	35572	26.58	0.516	23220	50.26	0.516
4	11	47276	35.33	0.686	30960	67.01	0.689
5	11	58974	43.94	0.856	38700	83.76	0.861
6	9	68860	51.46	1.000	44928	97.24	1.000

		BRAMs (Total = 365)			IOBs (Total = 500)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	8	2.19	0.267	428	85.6	0.986
2	11	13.5	3.69	0.450	430	86.0	0.991
3	11	19	5.20	0.633	432	86.4	0.995
4	11	24.5	6.71	0.817	432	86.4	0.995
5	11	30	8.22	1.000	434	86.8	1.000
6	9	26.5	7.26	0.883	330	66.0	0.760

Table 6.5: Table of resource utilization for window parameter w=3 for Artix-7.

		LUTs (Total = 133800)			Distributed Memory (Total = 46200)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	12304	9.19	0.208	7884	17.06	0.200
2	11	24012	17.95	0.406	15768	34.13	0.400
3	11	37518	28.04	0.634	23652	51.19	0.600
4	11	47428	35.45	0.802	31536	68.26	0.800
5	11	59133	44.19	1.000	39420	85.32	1.000

		BRAMs (Total = 365)			IOBs (Total = 500)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	9	2.46	0.290	472	94.4	0.987
2	11	14.5	3.97	0.468	474	94.8	0.992
3	11	20	5.48	0.645	476	95.2	0.996
4	11	25.5	6.98	0.822	476	95.2	0.996
5	11	31	8.49	1.000	478	95.6	1.000

Table 6.6: Table of resource utilization for window parameter w=4 for Artix-7.

Figures below present this data as a plot.

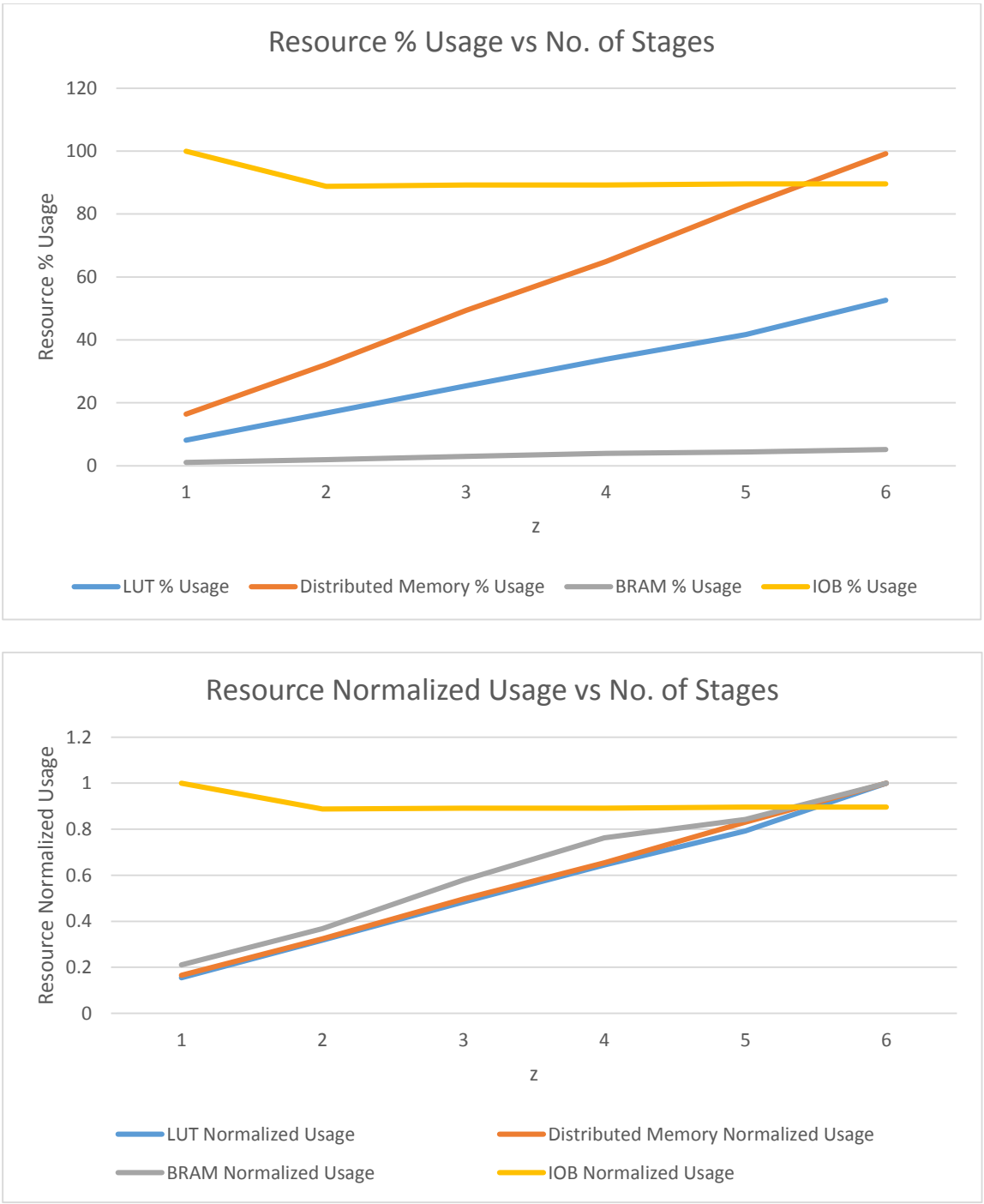


Figure 6.4: Plot of Resource Usage vs the Number of Stages for w=0 for the Artix-7 FPGA

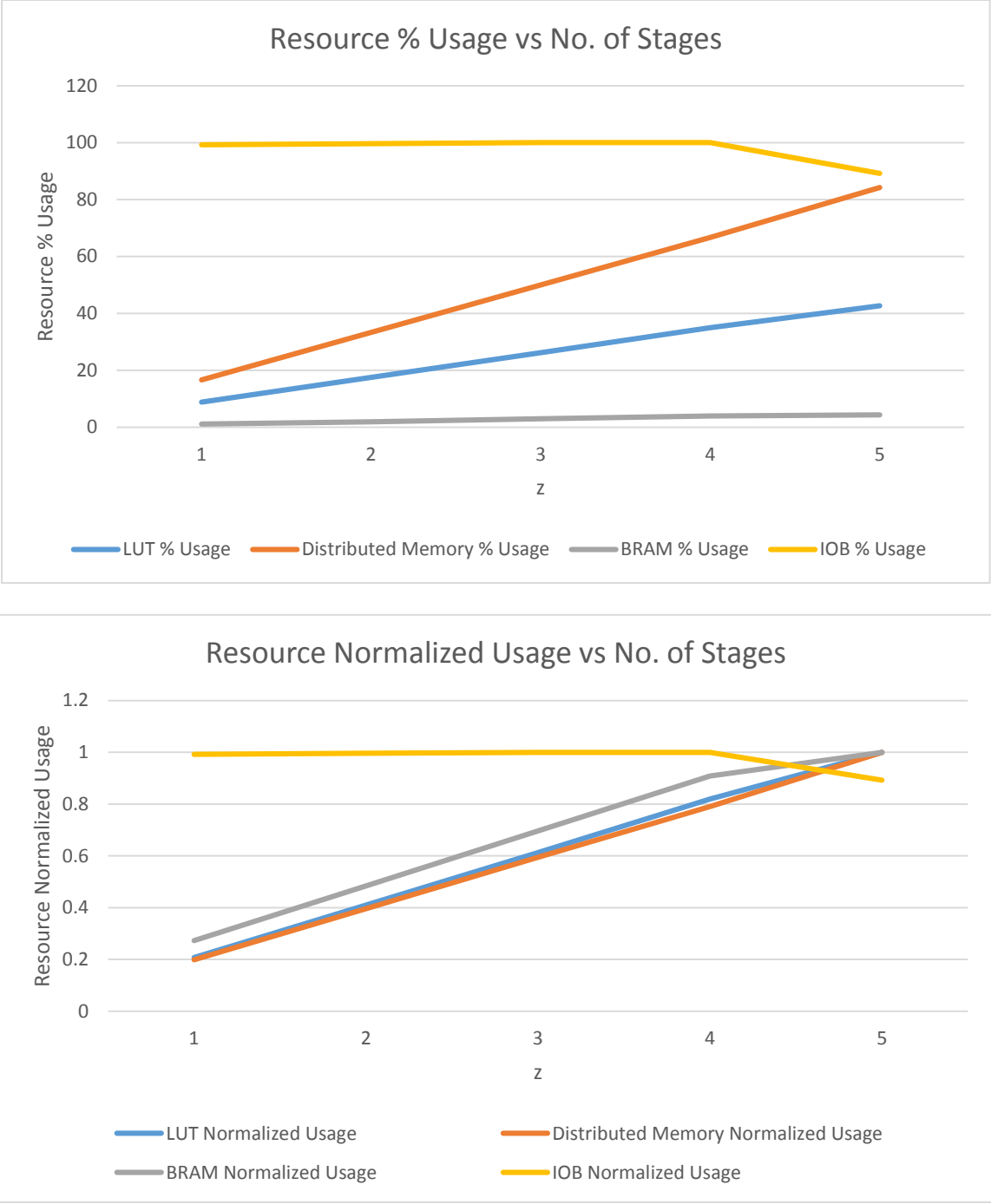
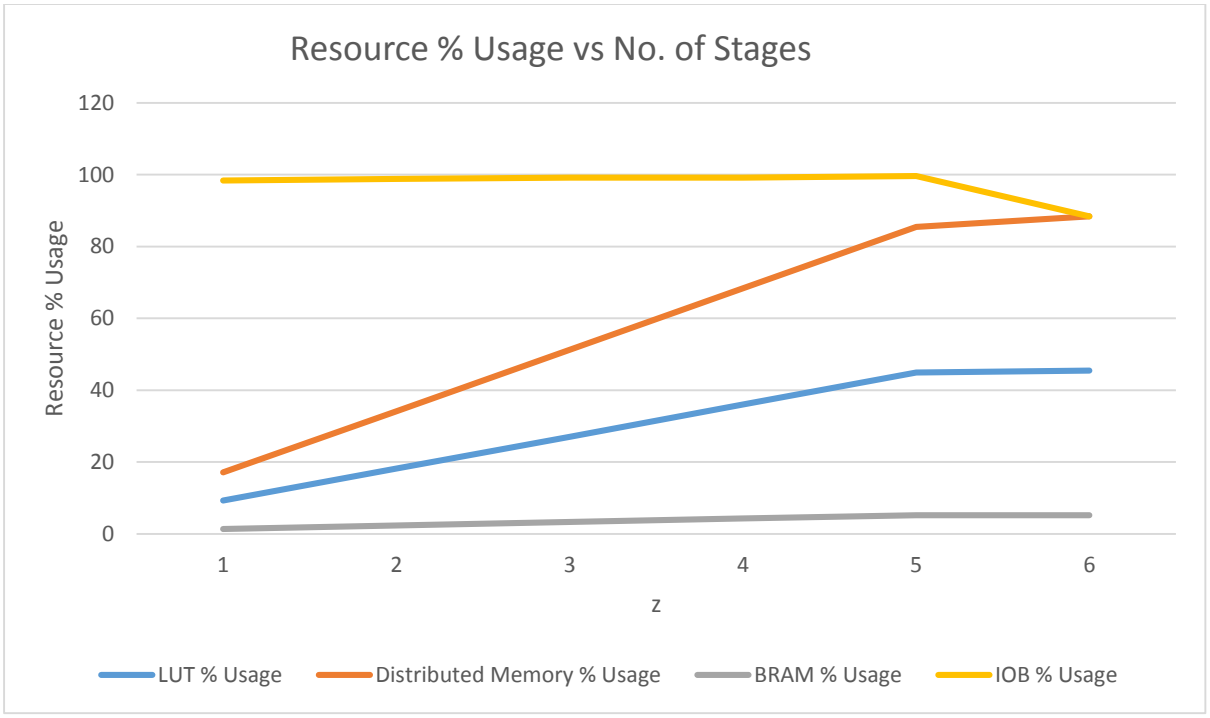


Figure 6.5: Plot of Resource Usage vs the Number of Stages for w=1 for the Artix-7 FPGA



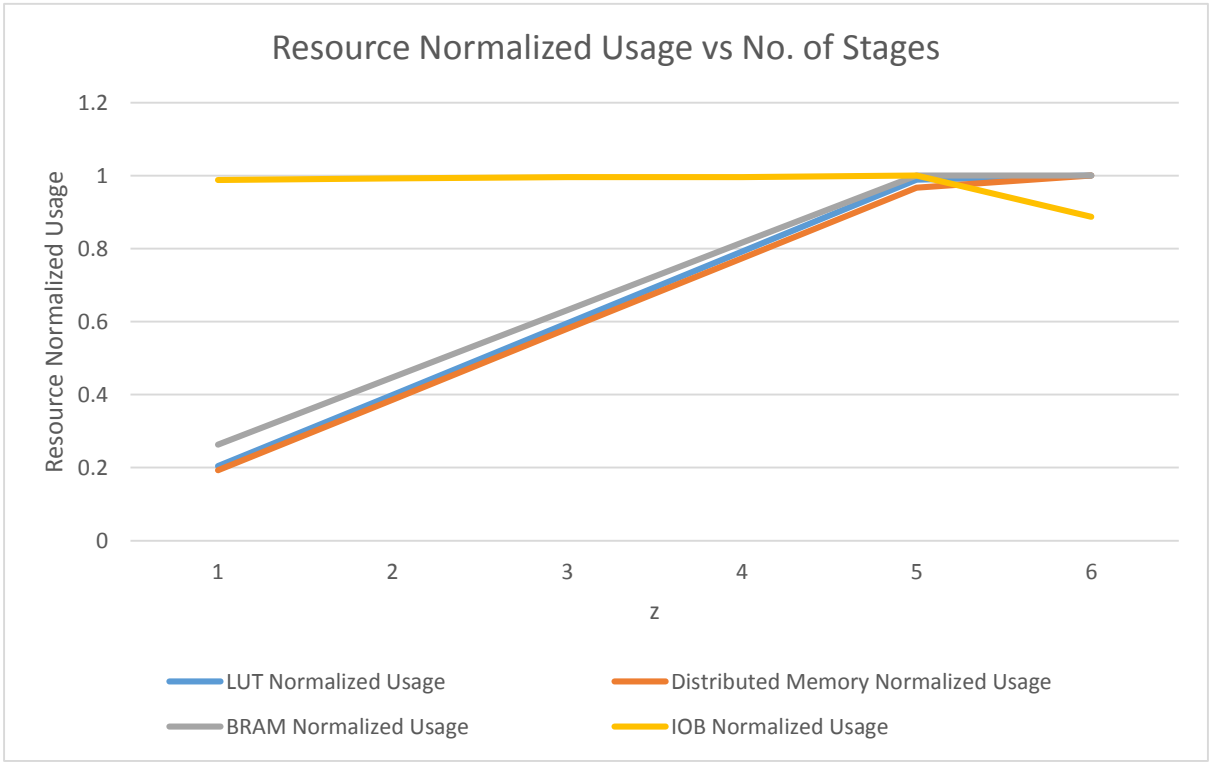


Figure 6.6: Plot of Resource Usage vs the Number of Stages for w=2 for the Artix-7 FPGA

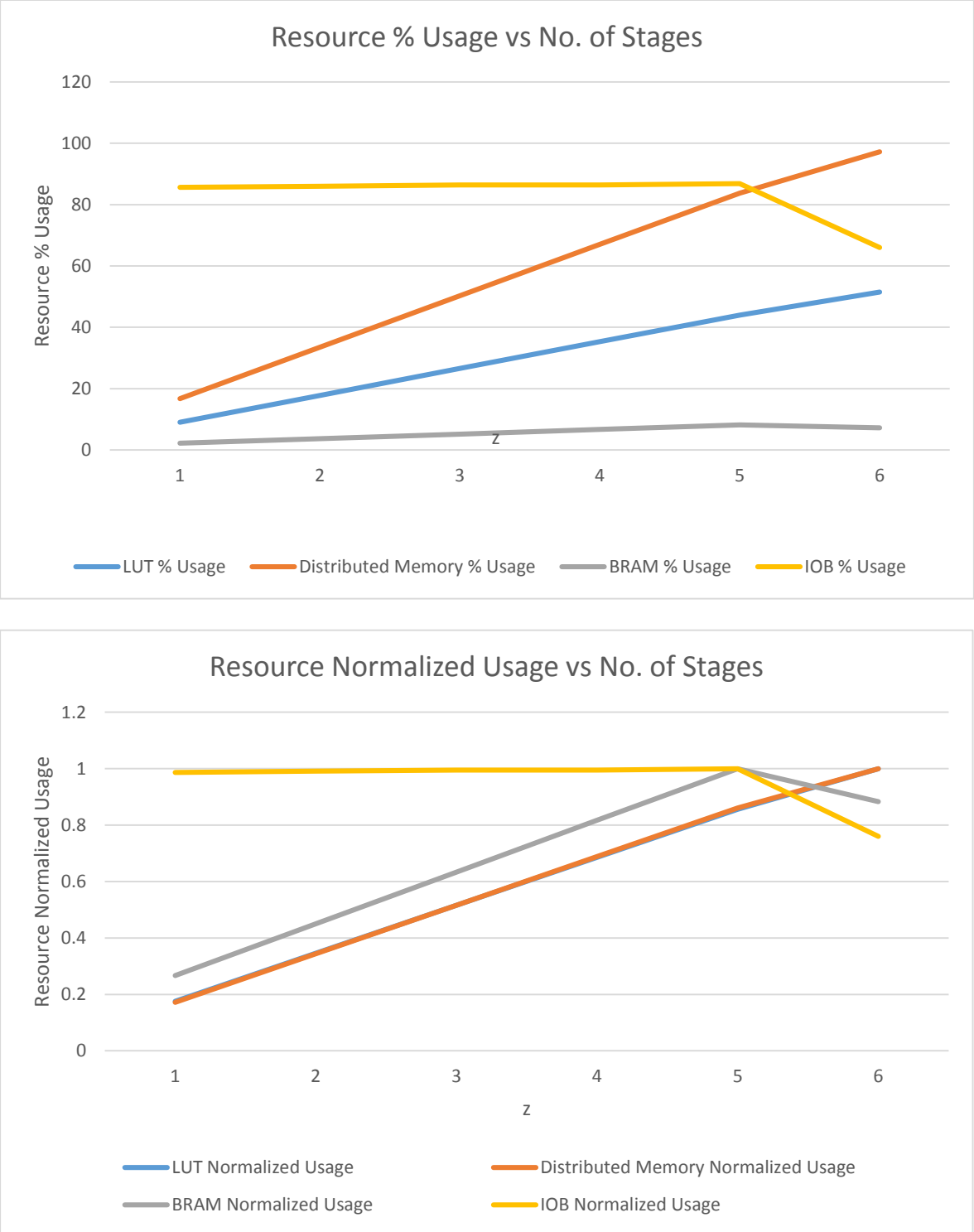
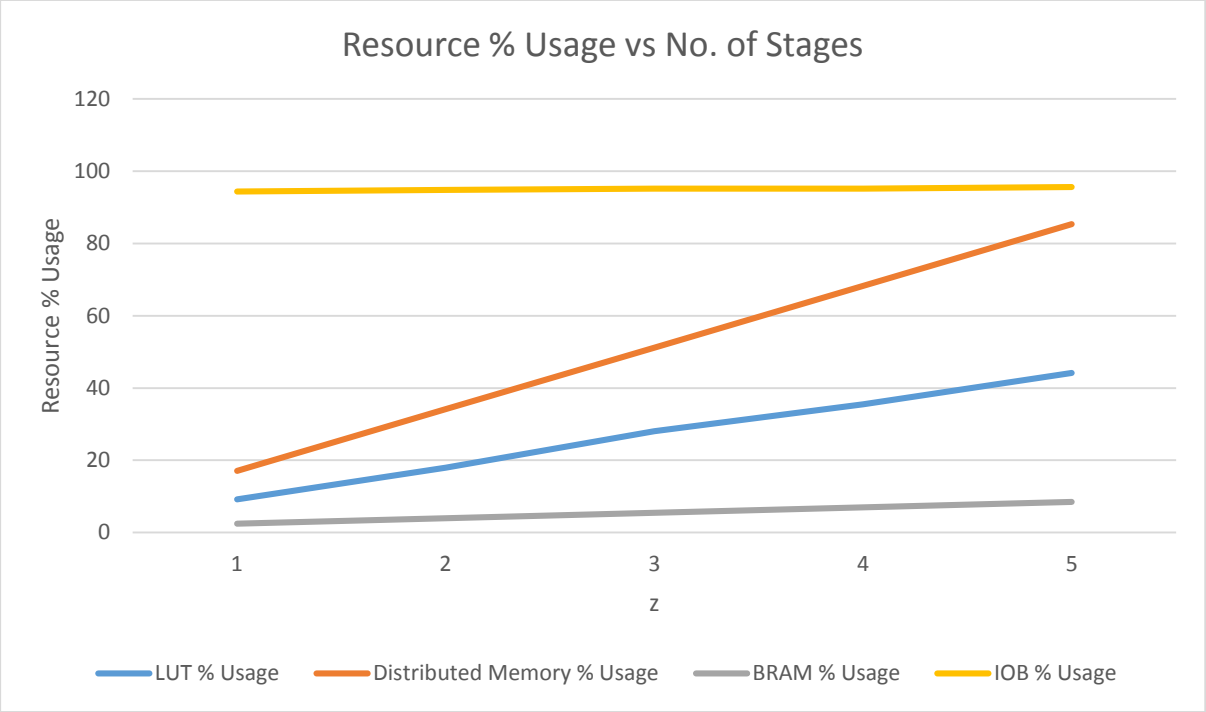


Figure 6.7: Plot of Resource Usage vs the Number of Stages for w=3 for the Artix-7 FPGA



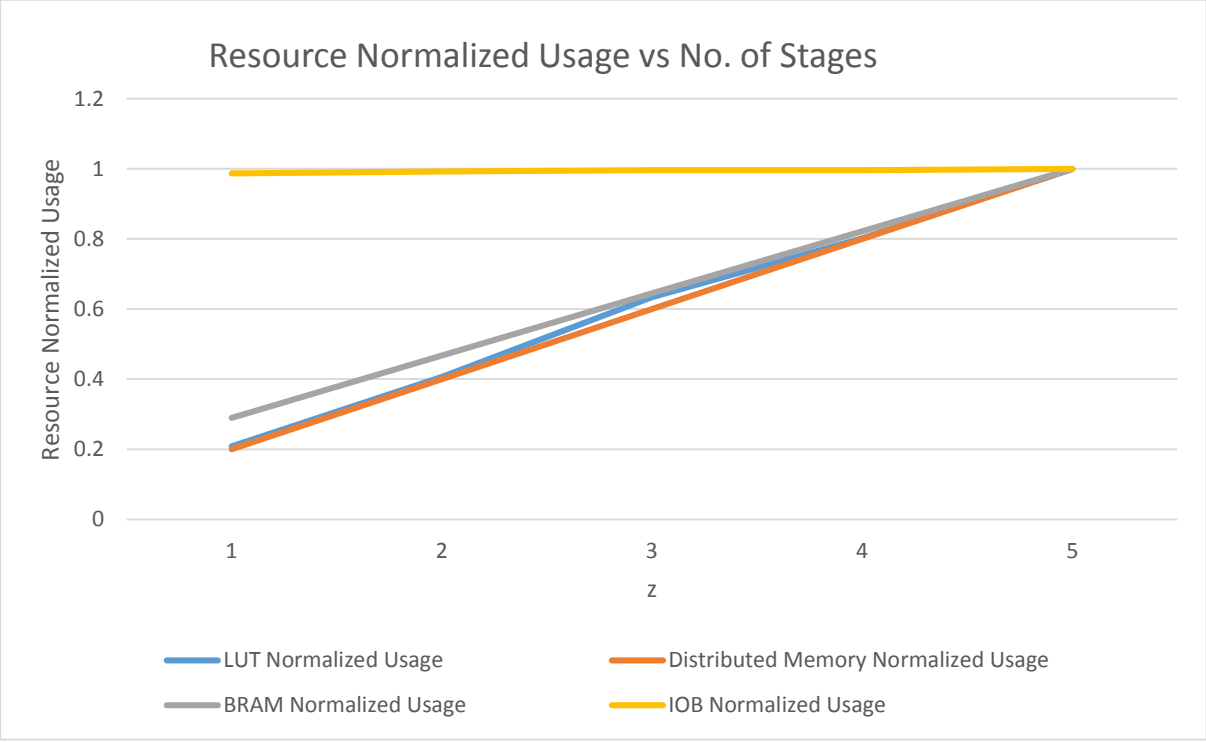


Figure 6.8: Plot of Resource Usage vs the Number of Stages for w=4 for the Artix-7 FPGA

Now we present the data for Kintex-7 which has 400 I/O pins, 35000 LUTs that can be configured as distributed memory and 325 BRAMs. Tables 6.7, 6.8 – 6.12 show similar data for the Kintex-7 FPGA (that has fewer pins and internal resources than the Artix-7 FPGA).

w	nmax	zmax	fmax	fmin
0	13	5	43.47	41.67
1	12	5	40.00	35.71
2	11	4	33.33	31.25
3	10	4	30.30	29.41
4	9	4	28.57	27.77

Table 6.7: Limiting values for Kintex-7

Tables 6.8 – 6.12 shows the absolute, percentage and normalized resource utilization for various window parameters, ranging from 0 to 4 for the Kintex-7 FPGA. The absolute resource utilization number is the exact number as provided by the Vivado tool, whereas percentage utilization is the percentage of resource utilized from the maximum that is available for the particular FPGA, the normalized utilization is the ratio of absolute value to the maximum of all absolute values among the cases considered.

		LUTs (Total = 101400)			Distributed Memory (Total = 35000)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	13	8764	8.64	0.183	5346	15.27	0.170
2	13	18213	17.96	0.381	11788	33.68	0.376
3	13	28012	27.62	0.587	18134	51.81	0.578
4	13	37827	37.30	0.792	24784	70.81	0.790
5	13	47724	47.06	1.000	31340	89.54	1.000

		BRAMs (Total = 325)			IOBs (Total = 400)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	13	2.5	0.77	0.200	388	97.0	0.984
2	13	5	1.54	0.400	390	97.5	0.989
3	13	7.5	2.30	0.600	392	98.0	0.995
4	13	10	3.07	0.800	392	98.0	0.995
5	13	12.5	3.84	1.000	394	98.5	1.000

Table 6.8: Table of resource utilization for window parameter w=0 for Kintex-7; the last row in each table corresponds to the largest implementable value of z.

		LUTs (Total = 101400)			Distributed Memory (Total = 35000)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	12	9952	9.81	0.202	6524	18.64	0.200
2	12	19694	19.42	0.401	13048	37.28	0.400
3	12	29411	29.00	0.599	19572	55.92	0.600
4	12	39278	38.73	0.801	26096	74.56	0.800
5	12	49029	48.35	1.000	32620	93.20	1.000

		BRAMs (Total = 325)			IOBs (Total = 400)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	12	3.5	1.07	0.260	386	96.5	0.984
2	12	6	1.84	0.440	388	97.0	0.989
3	12	8.5	2.61	0.630	390	97.5	0.995
4	12	11	3.38	0.810	390	97.5	0.995
5	12	13.5	4.15	1.000	392	98.0	1.000

Table 6.9: Table of resource utilization for window parameter w=1 for Kintex-7.

		LUTs (Total = 101400)			Distributed Memory (Total = 35000)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	11220	11.06	0.256	7344	20.98	0.250
2	11	22060	21.75	0.504	14688	41.96	0.500
3	11	32904	32.44	0.752	22032	62.94	0.750
4	11	43746	43.14	1.000	29376	83.93	1.000

w=2

		BRAMs (Total = 325)			IOBs (Total = 400)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	11	6.5	2.00	0.302	382	95.5	0.989
2	11	11.5	3.53	0.534	384	96.0	0.995
3	11	16.5	5.07	0.767	386	96.5	1.000
4	11	21.5	6.61	1.000	386	96.5	1.000

Table 6.10: Table of resource utilization for window parameter w=2 for Kintex-7.

		LUTs (Total = 101400)			Distributed Memory (Total = 35000)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	10	12372	12.20	0.256	7920	22.62	0.250
2	10	24353	24.01	0.503	15840	45.25	0.500
3	10	36337	35.83	0.752	23760	67.88	0.750
4	10	48320	47.65	1.000	31680	90.51	1.000

		BRAMs (Total = 325)			IOBs (Total = 400)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	10	7	2.15	0.318	374	93.5	0.989
2	10	12	3.69	0.545	376	94.0	0.995
3	10	17	5.23	0.772	378	94.5	1.000
4	10	22	6.77	1.000	378	94.5	1.000

Table 6.11: Table of resource utilization for window parameter w=3 for Kintex-7.

		LUTs (Total = 101400)			Distributed Memory (Total = 35000)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	9	11902	11.73	0.256	7488	21.39	0.250
2	9	23414	23.09	0.504	14976	42.78	0.500
3	9	34909	34.42	0.752	22464	64.18	0.750
4	9	46411	45.77	1.000	29952	85.57	1.000

		BRAMs (Total = 325)			IOBs (Total = 400)		
z	n	Absolute Usage	% Usage	Normalized Usage	Absolute Usage	% Usage	Normalized Usage
1	9	6	1.84	0.363	360	90.0	0.989
2	9	9.5	2.92	0.575	362	90.5	0.995
3	9	13	4.00	0.787	364	91.0	1.000
4	9	16.5	5.07	1.000	364	91.0	1.000

Table 6.12: Table of resource utilization for window parameter w=4 for Kintex-7.

On the whole, we observe that the design bottleneck (as evidenced from Table 6.1 as well) is the I/O resources. They typically saturate close to 100% of the % usage in Tables 6.2 – 6.6. The few case where they are less than 100% are due to the use of $n < n_{\max}$ and discretization effect due to integer value of n.

The next most important resource is the distributed memory, particularly for a large number of pipeline stages. This is used for the temporary storage and the compute module. By carefully designing the stage to eliminate or reduce the duplication due to the input buffer and temporary storage, the distributed memory usage could be cut by almost 30% (from $2n^2 + (n + 2w)^2$ to $n^2 + (n + 2w)^2$). This could also reduce the number of cycles needed at each stage to 2 clock cycles (rather than 3).

Chapter 7
Concluding Remarks

In this thesis, we have considered the running of a windowed computation on an FPGA. Specifically, we have addressed the following; we have defined a handshaking scheme that allowed us to construct an FPGA architecture without making large assumptions about component speeds and synchronization. The relevance of this work extends to other FPGAs and I/O bandwidth limited modules. We defined a pipeline architecture for windowed computations, including details of a stage to accommodate FPGA pin-limitation and bounded storage. This architecture is also generic with the potential for extension to other environments. We have modified the method of Vaidyanathan *et al.* [9] to better suit FPGAs where it ensures a smoother (stall-resistant) flow of the computation in the pipeline. Based on the architecture proposed, we have analytically predicted resource usage in the FPGA. In particular, we have shown that for an $N \times N$ image processed as $n \times n$ tiles on a z-stage windowed computation with parameter w; $\Theta(n^2 + \log N + \log z)$ pins are used and $\Theta(n^2 z)$ memory is used. We ran simulations that validated these predictions on two FPGAs (Artix-7 and Kintex-7) with different resources. As we had predicted, the pins and distributed memory turned out to be the most used resources. Our simulations have also shown that the operating clock speed of the design is relatively independent of the number of stages in the pipeline; this is in line with what was expected with the handshaking scheme that isolates the timing of communicating modules.

This work opens up many directions for future work. Can the temporary input buffer be removed? This would greatly reduce the amount of distributed memory used. Would the predicted trends of resource use translate to other FPGAs (we expect it would)? Would this translate to other modules, where the ratio of I/O resources to other resources may be greater? It would also be

interesting to run the algorithm to account for pipeline timing. This would address the efficiency of the memory interface and the effect of expanding the compute fabric to perform more complex operations.

Bibliography

1. A. Benoit, U. V. Catalyurek, Y. Robert and E. Saule, “A Survey of Pipelined Workflow Scheduling: Models and Algorithms,” LIP Research Report RR-LIP-2010-28, Ecole Normale Sup´erieure ´ de Lyon, France, 2010.
2. B. A. Draper, J. R. Beveridge, A. P. W. B¨ohm and M. Chawathe, “Accelerated Image Processing on FPGAs,” IEEE Trans. Image Processing, vol. 12, no. 12, Dec. 2003, pp. 1543–1551.
3. Stephen D. Brown, Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic. *Field-programmable gate arrays*. Vol. 180. Springer Science & Business Media, 2012.
4. Ebeling, William HC, and Gaetano Borriello. "Field programmable gate array." U.S. Patent No. 5,208,491. 4 May 1993.
5. I. K. Park, N. Singhal, M. H. Lee, S. Cho and C. W. Kim, “Design and Performance Evaluation of Image Processing Algorithms on GPUs,” IEEE Trans. Parallel and Distributed Systems, vol. 22, issue 1, pp. 91–104.
6. IEEE Standard for Verilog Hardware Description Language," in *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* , vol., no., pp.0_1-560, 2006
7. Marcellin, Michael W., et al. "An overview of JPEG-2000." *Data Compression Conference, 2000. Proceedings. DCC 2000*. IEEE, 2000.
8. P. E. Danielsson and S. Levialdi, "Computer Architectures for Pictorial Information Systems," in *Computer*, vol. 14, no. 11, pp. 53-67, Nov. 1981. doi: 10.1109/C-M.1981.220251, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1667158&isnumber=34899
9. R. Vaidyanathan and P. Vinukonda, "On Running Windowed Image Computations on a Pipeline," *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, Shanghai, 2012, pp. 813-820.

10. R. Vaidyanathan and J. L. Trahan, "Dynamic Reconfiguration: Architectures and Algorithms," 2003, Kluwer Academic/ Plenum Publishers, New York.
11. Ramanath, Rajeev, Wesley E. Snyder, Youngjun Yoo, and Mark S. Drew. "Color image processing pipeline." *IEEE Signal Processing Magazine* 22.1 (2005): 34-43.
12. S. Asano, T. Maruyama and Y. Yamaguchi, "Performance Comparison of FPGA, GPU and CPU in Image Processing," Proc. Field Programmable Logic and Applications (FPL), 2009, pp. 126–131.
13. S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," in *Computer*, vol. 33, no. 4, pp. 70-77, Apr 2000. doi: 10.1109/2.839324,
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=839324&isnumber=18132
14. Xilinx Inc., *7 Series CLB Architecture.* 2016. *http://www.xilinx.com/video/fpga/7-series-clb-architecture.html.* 16 May 2016
15. Xilinx Inc., *7 Series FPGAs Configurable Logic Block*. 2016. *http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf*. 17 June 2016
16. Xilinx Inc., *7 Series FPGAs Overview*. 2016. *http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_implement_fpga_design.htm*. 17 June 2016.
17. Xilinx Inc., *Implementation*. 2016. *http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_implement_fpga_design.htm*. 16 May 2016.
18. Xilinx Inc., *Logic Simulation.* 2016. *http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug900-vivado-logic-simulation.pdf*. 16 May 2016.

19. Xilinx Inc., *Synthesis*. 2016. *http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug901-vivado-synthesis.pdf*. 16 May 2016.
20. Keinert, Joachim and Jürgen Teich. *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. New York: Springer, 2011. Print.
21. S. L. Bishop, S. Rai, B. Gunturk, J. L. Trahan and R. Vaidyanathan, "Reconfigurable Implementation of Wavelet Integer Lifting Transforms for Image Compression," Proc. IEEE International Conference on Reconfigurable Computing and FPGAs, 2006, pp 20-22.
22. N. Srivastava, J. L. Trahan, R. Vaidyanathan and S. Rai, "Adaptive Image Filtering using Run-Time Reconfiguration," Proc. Reconfigurable Architectures Workshop, 2003. p. 180.
23. J. B. Postel, "SIMPLE MAIL TRANSFER PROTOCOL," RFC 821 (August 1982).
24. P. Morrissey, N. P. Smart, and B. Warinschi. 2010. The TLS Handshake Protocol: A Modular Analysis. *J. Cryptol.* 23, 2 (April 2010), 187-223. DOI=http://dx.doi.org/10.1007/s00145-009-9052-3.
25. IEEE Draft Amendment to IEEE Std 802.11, 1999 Edition (Reaff 2003) Amendment 7: 4.9 GHz5 GHz Operation in Japan (As Amended by IEEE Stds 802.11a-1999, 802.11b-1999, 802.11b-1999/Cor 1-2001, 802.11d-2001, 802.11g-2003, 802.11h-2003), and 802.11i-2004)," in *IEEE Std P802.11j/D1.6* , vol., no., pp., 2004 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4040936&isnumber=4040935
26. B. Lloyd, W. Simpson, "PPP Authentication Protocols," RFC 1334 (October 1992).
27. D. L. Oliveira, K. Garcia and R. d'Amore, "Using FPGAs to implement asynchronous pipelines," *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on*, Santiago,2014, pp.1-4. doi:10.1109/LASCAS.2014.6820272. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6820272&isnumber=6820243

Vita

Aswin Vijaya Varma was born in April of 1989, in Alleppey, India to P. R. Vijaya Varma and Jayasree S. Varma. In June 2010, he received his Bachelor's degree in Electronics and Communication Engineering from Mahatma Gandhi University, Kottayam, India. In January 2013, he joined the Department of Electrical and Computer Engineering at Louisiana State University, Baton Rouge, Louisiana. He worked as a graduate assistant at the LSU Continuing Education, while pursuing Masters thesis research under the supervision of Dr. R. Vaidyanathan. He will receive his Master of Science degree in Electrical Engineering in the Summer of 2016.