University of Kentucky
UKnowledge

Theses and Dissertations--Electrical and Computer Engineering

Electrical and Computer Engineering

2007

AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER

Vijai Raghunathan
University of Kentucky

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Raghunathan, Vijai, "AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER" (2007). *Theses and Dissertations--Electrical and Computer Engineering*. 8.
https://uknowledge.uky.edu/ece_etds/8

This Master's Thesis is brought to you for free and open access by the Electrical and Computer Engineering at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Electrical and Computer Engineering by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained and attached hereto needed written permission statements(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine).

I hereby grant to The University of Kentucky and its agents the non-exclusive license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless a preapproved embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's dissertation including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Vijai Raghunathan, Student

Dr. William Dieter, Major Professor

Dr. Yuming Zhang, Director of Graduate Studies

ABSTRACT OF THESIS

AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER

Designing hardware to output pixels for light field displays or multi-projector systems is challenging owing to the memory bandwidth and speed of the application. A new technique of hardware that implements 'anywhere pixel routing' was designed earlier at the University of Kentucky. This technique uses hardware to route pixels from input to output based upon a Look up Table (LUT). The initial design suffered from high memory latency due to random accesses to the DDR SDRAM input buffer. This thesis presents a cache design that alleviates the memory latency issue by reducing the number of random SDRAM accesses.

The cache is implemented in the block RAM of a field programmable gate array (FPGA). A number of simulations are conducted to find an efficient cache. It is found that the cache takes only a few kilobits, about 7% of the block RAM and on an average speeds up the memory accesses by 20-30%.

Keywords: Pixel router, LUT, Memory latencies, Block RAM, Cache.

Vijai Raghunathan
(Author's signature)
____10/18/2007____
(Date)

AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER

By

Vijai Raghunathan

Dr. William Dieter
(Director of Thesis)

Dr. Ruigang Yang
(Co-Director of Thesis)

Dr.Yuming Zhang
(Director of Graduate Studies)

10/18/2007
(Date)

RULES FOR THE USE OF THESIS

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

_____

_____

THESIS

Vijai Raghunathan

The Graduate School

University of Kentucky

2007

AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in the
College of Engineering
at the University of Kentucky

By,
Vijai Raghunathan

Lexington, Kentucky

Director: Dr.William Dieter, Assistant Professor of Electrical & Computer Engineering

Lexington, Kentucky

2007

Copyright © Vijai Raghunathan 2007

Dedicated to
My friend, philosopher and guide unhsivaham

ACKNOWLEDGEMENTS

I would like to thank Dr. Ruigang Yang for all his help. Without his constant motivation and advice this thesis idea and research would not have materialized. I would like to thank Dr. Bill Dieter for all his advice and guidance. He was in constant touch with the happenings of the project and was a great advisor for my research. I would also like to thank Dr. Robert Heath for putting away some of his valuable time and agreeing to be a part of my thesis committee.

I am extremely thankful to my parents and family members for all their moral support. Finally, I would like to thank all my friends who have been very helpful and supportive during my research days at University of Kentucky.

Table of Contents

ACKNOWLEDGEMENTS iii
Table of Contents iv
List of Tables vi
List of Figures vii
Chapter 1: Introduction 1
1.1) Motivation 4
1.2) Choosing a Cache for the Design: 6
1.3) Parallel Execution 6
1.4) Basic Concepts of Cache Used in the Project 7
1.5) Calibration 7
1.6) Linear Interpolation 9
1.7) The LUT 10
Chapter 2: Previous Work 11
2.1) Cache in General Purpose Processors 11
2.2) Graphics Related Work 12
Chapter 3: SDRAM Performance 15
3.1) Design Values and Simulation Numbers 15
3.2) Simulation Results (RAM only) 16
Chapter 4: The Cache Design 19
4.1) Cache Parameters 19
4.2) Hardware Design 20
4.3) Simulation Parameters 20
4.4) Caching Function 22
4.5) Simulation Results 23
Chapter 5: Concept of Memory Blocks 25
5.1) Memory Blocks 27
5.2) Loading Block Sequences 31
5.3) Cache Simulations with Blocks 34
5.4) SDRAM with Blocks 43
Chapter 6: Set Associative Caches 45
6.1) Set Associative Caches 45
6.2) Caching Function 45
6.3) Simulation Results 46
Chapter 7: Bilinear Interpolation 50
7.1) Advantages of Bilinear Interpolation 51
7.2) SDRAM simulation with Bilinear Interpolation 56
Chapter 8: Finding the Input Access Pattern 60
8.1) Determining the Access Pattern 60
8.2) Algorithm 62
Chapter 9: SDRAM vs Cache Comparison 64
9.1) Test LUTs 64
9.2) Simulation Results 64
9.3) Bilinear Interpolation 65
Chapter 10: Conclusion 67

Appendix A: Simulation Results 68
Appendix B: Simulation Code 82
B.1) SDRAM Simulation..... 82
B.2) Cache with Blocks 93
References.....109
VITA 111

List of Tables

Table 1.1: Address Bits split up 10

List of Figures

Figure 1.1: Pixel Compositor [2] 2
Figure 1.2: Reverse Mapping Process 3
Figure 1.3: VGA Timing [3] 4
Figure 1.4: Projector Output [3] 8
Figure 1.5: Sample Triangle 9
Figure 3.1: SDRAM Access Times 17
Figure 3.2: MER 18
Figure 4.1: Inclusion of Cache 20
Figure 4.2: Cache Logic Flow 21
Figure 4.3: Caching Function 22
Figure 4.4: Access Time Vs Cache Size (45 degrees) 23
Figure 4.5: Cache Size Vs Access Times (0 degrees) 24
Figure 5.1: LUT in memory 25
Figure 5.2: Input Frame Access 26
Figure 5.3: Division into Memory Blocks 28
Figure 5.4: LUT access using blocks 29
Figure 5.5: Input frame access in case of Blocks 30
Figure 5.6: Blocks Labeling 31
Figure 5.7: Hit rate for an access block size of 8x8 pixels as a function of a cache size for a cache with l lines and w words per line. 35
Figure 5.8: Access Times for a block Size 8x8 pixels as a function of a cache size for a cache with l lines and w words per line. 35
Figure 5.9: Hit rate for an access block size of 16x16 pixels as a function of a cache size for a cache with l lines and w words per line. 36
Figure 5.10: Access Times for a block Size 16x16 pixels as a function of a cache size for a cache with l lines and w words per line. 37
Figure 5.11: Hit rate for an access block size of 32x32 pixels as a function of a cache size for a cache with l lines and w words per line. 38
Figure 5.12: Access Times for a block Size 32x32 pixels as a function of a cache size for a cache with l lines and w words per line. 38
Figure 5.13: Hit rate for an access block size of 64x64 pixels as a function of a cache size for a cache with l lines and w words per line. 39
Figure 5.14: Access Times for a block Size 64x64 pixels as a function of a cache size for a cache with l lines and w words per line. 39
Figure 5.15: Hit rate for an access block size of 128x128 pixels as a function of a cache size for a cache with l lines and w words per line. 40
Figure 5.16: Access Times for a block Size 128x128 pixels as a function of a cache size for a cache with l lines and w words per line. 40
Figure 5.17: SDRAM vs Cache with Blocks 41
Figure 5.18: Cache 64x32, Block 64 Hit Rate 42
Figure 5.19: SDRAM vs SDRAM with Blocks vs Cache with Blocks 43
Figure 6.1: Hit Rate Comparison Direct vs Set Associative 46
Figure 6.2: Access time Comparison Direct vs Set Associative 47
Figure 7.1: Example for Bilinear 50

Figure 7.2: Nearest Neighbor Method 51
Figure 7.3: Bilinear Interpolation Method 52
Figure 7.4: Bilinear Interpolation with 1 bit after radix point 53
Figure 7.5: Bilinear Interpolation with 2 bits after radix point 54
Figure 7.6: Bilinear Interpolation with 3 bits after radix point 55
Figure 7.7: Explanation of Bilinear 57
Figure 7.8: Plot SDRAM vs Cache (Bilinear) 58
Figure 8.1: Sample Input access 61
Figure 8.2: Possible LUT functions 61
Figure 8.3: Input Access Algorithm 63
Figure 9.1: SDRAM vs Cache (Overall) 65
Figure 9.2: SDRAM vs Cache Bilinear 66

Chapter 1: Introduction

Light field displays can render 3 dimensional (3D) images without the use of 3D glasses. The light field display project's design uses a method of rendering through a cluster of projectors. These projectors are first calibrated and then they project onto a projection screen that has many micro-lenses. The projection of light rays onto these micro lenses creates a light field such that a 3 dimensional view can be established for viewers without any complex head tracking system or by using other mechanical devices [1].

The projection of light rays from different projectors means that these projectors have to be calibrated in order to ensure that the overlapping region between projector outputs is smooth. The calibration of these projectors is done using existing techniques [1]. Essentially, there needs to be a system that counters the distortions caused by multi-projector systems, and this action is performed by doing a warping on the input images [2]. The warping performed is dependent on the calibration results. Traditionally, the warping and blending of pixels is done in software, but software has its own limitations. Software depends on the graphics card used and the lower level device driver details are not provided by the manufacturer. Graphics cards are limited by the number of input signals. If there is a 16 input, 16 output projector system, the rendering of all the projectors cannot be handled by a single graphics card. In this case, multiple graphics cards are required to scale the system to higher number of inputs or outputs.

Dedicated hardware can do this warping and blending and routing the output pixels onto the projection screen without the need to develop custom software for every kind of graphics card. Thus, the "Anywhere Pixel router" was designed. The first hardware modules were designed for VGA (Video Graphics Array) outputs and the processors used were older versions of FPGA (Field Programmable Gate Array) like Virtex2 or Spartan 3 [3]. An FPGA is a reprogrammable logic chip comprising of many digital gates and some memory. In the current version of hardware, designed prior to the work presented here and shown in Figure 1.1, the VGA signals are replaced with HDMI (High Definition Multimedia Interface)/DVI (Digital Visual Interface) signals and a more advanced

Virtex-4 FPGA is used. Though the hardware is newer, the basic concept of warping and routing remains the same.

The basic idea of an “Anywhere Pixel Router” is to have series of input HDMI/DVI signals from the graphics cards of different computers, and a series of output HDMI/DVI signals connected to different projectors. An FPGA processor connects the transforms the inputs into the warped outputs using a Look-Up Table (LUT) that is calculated offline. The Anywhere Pixel Router uses memory for storing the input frames, the LUT and the output frames. The block RAM (Random Access Memory) present in the FPGA is not sufficient. For example, a 4-input and a 4-output multi-projector system needs 4 input frames, 4 LUTs and 4 output frames. Assuming a 1024x768 display system and each pixel having 32 bits of data then the amount of memory needed is 288 Megabits (Mb), which is 17000 times larger than the memory available in the block RAM. Hence there is a need for an external SDRAM (Synchronous Dynamic Random Access Memory) to store and retrieve all the data [3].

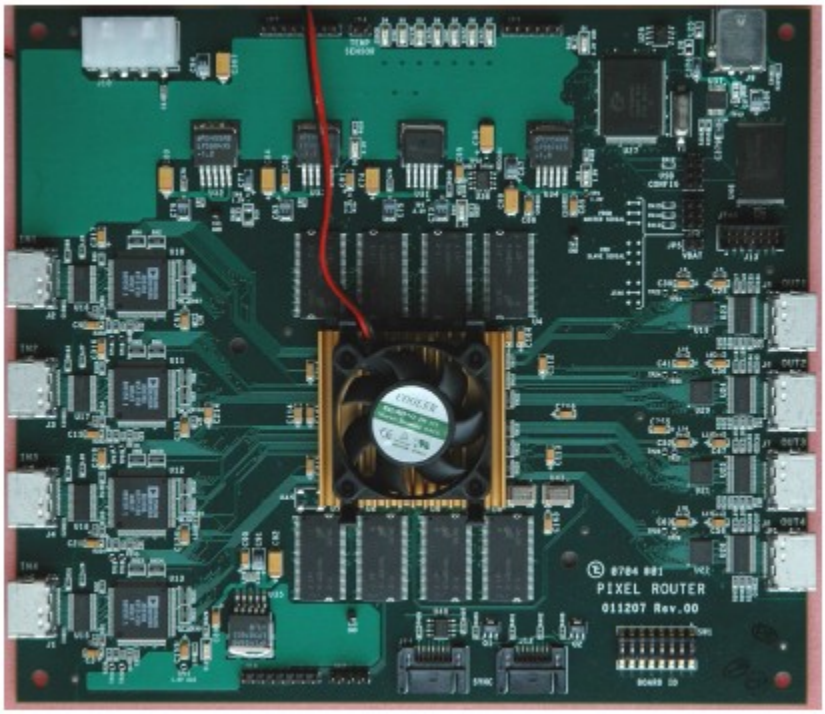
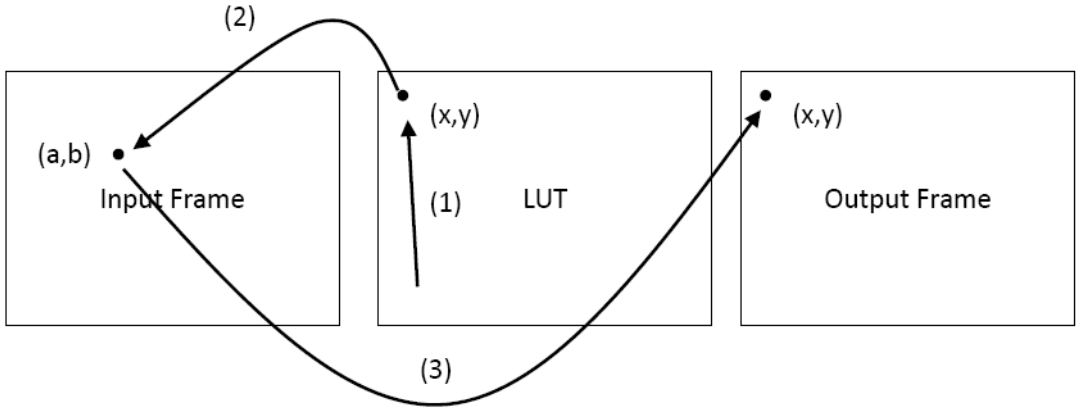


Figure 1.1: Pixel Compositor [2]

The basic function of the memory controller present in the FPGA is shown in Figure 1.2. The LUT is determined offline and loaded directly into the SDRAM by the FPGA before the routing starts. An input frame from one of the input channels is loaded to the SDRAM. The routing is done based on the technique of reverse mapping. The data in that LUT pixel tells which address or pixel to look for in the input frame so that it can be routed to the output frame. For example, if the controller is accessing pixel at location (x, y) in the LUT and data present in that location (x, y) of the LUT is (a, b), then the value in input frame pixel location (a, b) is routed to location (x, y) of the output frame.

Both the LUT and the output frame are accessed sequentially. Sequential access in the SDRAM has the least possible access time—the cause of the performance problem in the Anywhere Pixel Router (APR) is random access done on the input frame. The random nature of the accesses depends on the LUT. The LUT might contain a simple rotation about the center, might contain some geometric functions on different pixels. After the routing is done, the output frame has to be transferred to the corresponding projector for display.



(1) LUT value at (x,y) is looked up and address is decoded.

(2) Based on the address at (x,y) input frame value at (a,b) is fetched.

(3) This input value is placed in output frame at location (x,y) .

Figure 1.2: Reverse Mapping Process

Daniel Rudolf, the hardware consultant for APR has successfully designed the memory controller that does the warping and blending of input pixels to output for a 1024x768 system working at a frame rate of 60 Hz. The memory controller works for a 4 input, 4 output system which is shown in the Figure 1.1. Even though the memory controller functions correctly, it is not fast enough to produce outputs at 60 Hz.

1.1) Motivation

The APR processes input pixels from a source, like a graphics card and outputs the result onto a projector. The hardware uses the VGA timing (timing remains the same for DVI/HDMI systems) shown in Figure 1.3. The pixel clock is the clock that is used to time the horizontal and vertical scan times in a display system.

Resolution	Pixel Clk, in MHz
640x480, 60Hz	25.175
800x600, 60Hz	40.000
1024x768, 60Hz	65.000
1280x1024, 60Hz	108.000

Figure 1.3: VGA Timing [3]

A frame of data has to be read from the input, processed, and sent to the output by the hardware at 60 Hz which is approximately 16.6 ms for every frame. An important issue is timing factor of the entire process. The volume of data that is to be handled is large. Consider a 1-in, 1-out system with a resolution of 1024x768 pixels. The input frame requires 24 Mb assuming each memory location has 32 bits of data. Similarly, the LUT and the output each have 24 Mb. The memory controller in the FPGA has to fetch, look up and load a total of 72 Mb of data within 16.6 ms in order to have no lag in the output. For a 1-in1-out system, the throughput needed is approximately 4 Gb/s.

Another factor limiting the output is the external SDRAM. The SDRAM is a memory storage unit capable of handling data up to several gigabits in size. A DRAM module is an array of cells with each cell having a capacitor and a transistor to store 1 bit of data. An array of cells is a memory row. To read a value from any cell in the DRAM module,

the row of that cell is first selected. Sense lines connect cells in a memory row to a latch, after which a column address is used to multiplex the data to the output. The RAS (Row Address Strobe) latency, $t_{ras,}$ is the time taken to open a row after a request is sent to the memory module to access values in that row. The RAS latency for the APR's memory module running at 133 MHz is 7 cycles. The CAS (Column Address Strobe) latency, $t_{cas,}$ is the time taken to receive a value stored in a memory column at the output after a request is sent to the memory module to access that value. The CAS latency for the APR's memory module running at 133 MHz is 2 cycles.

For an SDRAM operating at 133 MHz, the overhead due to CAS and RAS delays may not seem to add significant delay. The APR requires a throughput of several Gb/second and a delay of several clock cycles per access add substantial delay. Even with a fast DDRRAM (Double Data Rate RAM), it is unlikely that the target of 16.6ms for one frame could be achieved.

There is a possibility of having more memory modules interfaced to the FPGA to increase the memory bandwidth, but the FPGA has a limited number of I/O (Input/Output) pins. Having more than one FPGA might increase the possibility of having more memory modules but such an implementation is beyond the scope of this thesis.

There is a need for achieving faster frame rates given the requirements of SDRAM, the FPGA. One of the simplest and most feasible methods is to design a cache for the memory. Since the input frame pixels are the ones that are accessed randomly, the cache is designed to store and retrieve the input pixels. The design of a cache depends on many parameters like the number of cache lines, the number of pixels stored in each cache line, and the replacement strategy if there is a cache miss. The cache represents only a small portion of the memory space when compared to the memory present in SDRAM. A cache can be designed to fit easily within the memory space present in the block RAM of the FPGA. When implemented in the block RAM, the cache is present in the FPGA itself so cache access takes only 1 cycle at the most. If there is a cache hit, a lot of cycles are

saved when compared to the normal SDRAM access of the input frame, especially if the pixels are accessed in random order.

1.2) Choosing a Cache for the Design:

A stream of input pixel values cannot be loaded directly into any buffer from the SDRAM. This is because during run time, it is not possible to predict what the LUT has in store for the routing. Every value in the LUT must be looked up; the corresponding value in input frame is fetched and placed in the output frame. The entire process works by going through each pixel in the LUT. Even with a fast memory module such as a DDR SDRAM, the full memory bandwidth cannot be utilized because of non-sequential memory accesses. Double data rate ensures that on every clock cycle, two adjacent memory location values are transferred [4]. The second value it transfers from the LUT could correspond to a completely different row in the input frame. If this is the case, then the process is slowed down. To fetch this value, one has to go through the process of closing the existing row, opening a new one and then reading the value. The LUT access and the output frame accesses (writing) are sequential. So the double data rate feature can be used while reading from the LUT and writing to the output frame. Reading from input frame is the biggest limiting factor in the application because input frame access order is unknown. It could be random or it could be totally sequential. It is for this access that a cache is necessary to speed up the process.

1.3) Parallel Execution

For this application, memory accesses can happen in parallel. For example, the input frames, the LUTs, and the output frames could all be stored in different memory chips, but connected to the same master processor. The processor can access values from the LUT, the input frame and the output frame at the same time. Accessing memories in parallel will lead to speedup as the three different accesses could be made to run in parallel. For instance, when the LUT value for pixel i is fetched, the input value of pixel i-1 can be read from the input, and the value of pixel i-2 can be written to the output. If the LUT, the inputs and the outputs are in the same memory unit, then each access must wait for previous accesses to complete before doing the next access. Otherwise, the current access will be interrupted.

1.4) Basic Concepts of Cache Used in the Project

There are different types of caches. Finding a suitable cache for this specific application is the main goal of this research. There are different parameters for a cache, like the number of cache lines, the number of locations per each cache line. Also, the cache could be direct mapped cache or a set-associative cache. In a direct mapped cache the ingredients needed to map a cache location to that of a memory address are the tag, the offset and the index. Similarly, in a set associative cache, it is the block address and the offset. Further in a set associative cache, a replacement policy is required to replace an existing block in the cache. There could be different replacement strategies like Least Recently Used (LRU), First In First Out (FIFO) [5].

All these ideas will be explained in detail in the later sections along with discussion of how the parameters are chosen and how the cache locations are mapped to the memory addresses.

1.5) Calibration

Since the APR involves multi-projector systems, there will be overlap between projector outputs on the projection screen. This means that the projectors have to be calibrated to remove all the distortions caused by overlapping and also due to curved surfaces. The outputs have to be blended properly. A typical four projector system has outputs like the one shown in Figure 1.4. The process of calibration seems independent of this project but the LUT that is fed into the hardware is created by using the results of the calibration. Though LUT design is not part of this project, knowledge of how the LUT is arranged does affect the cache design and parameters that will be used in the application.

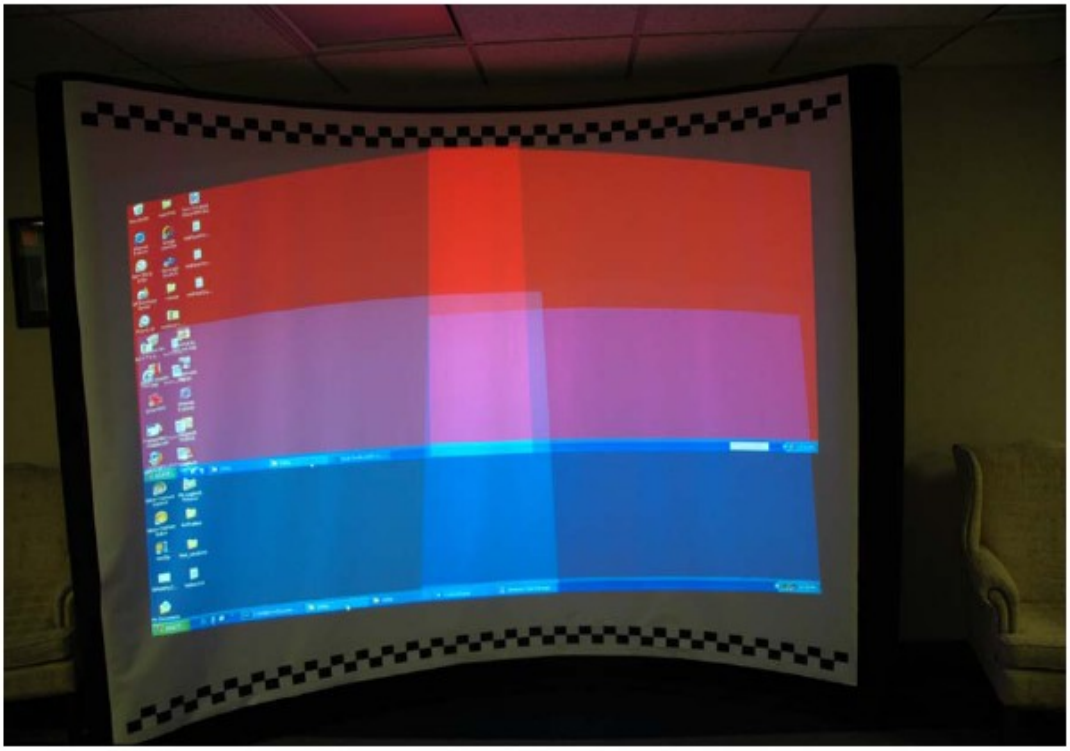


Figure 1.4: Projector Output [3]

The calibration results contain one output file each projector used, and one file for each projector's blending values. For example a four projector system has four different output files and four files containing the alpha blending values. The calibration results could be different for different calibration algorithms, but the one used for this project had outputs like described above. It is from these calibration results that the LUTs are created.

The calibration technique used for this work essentially created output files which contained a number of triangle vertices and the values of the respective vertices. The values at each triangle vertices basically were the row and column of the input frame pixel the output frame pixel is being mapped to. Thus the relationship between the output frame and the input frame was that of a reverse mapping [3]. The mapping function is given by the LUT that is calculated offline from the calibration results.

But the calibration results contain only a limited set of triangle vertices with their corresponding values. To obtain the entire set of values for creating the LUT, some kind of interpolation has to done.

1.6) Linear Interpolation

The calibration results contain several triangle vertices. Based on the coordinate values the triangle needs to be completely filled. Once all the triangles are filled, then the desired LUT is obtained. Each LUT value contains other information and not just the row and column value of the input frame. The LUT values and their corresponding components will be discussed in Subsection 1.7.

There are a number of algorithms to fill in triangles. Methods could be obtained from existing scan line conversion techniques, incremental algorithms, or midpoint algorithms. Each of these techniques has its own advantages and disadvantages [7]. Most of these methods adopt the basic linear interpolation with small variations in their approach.

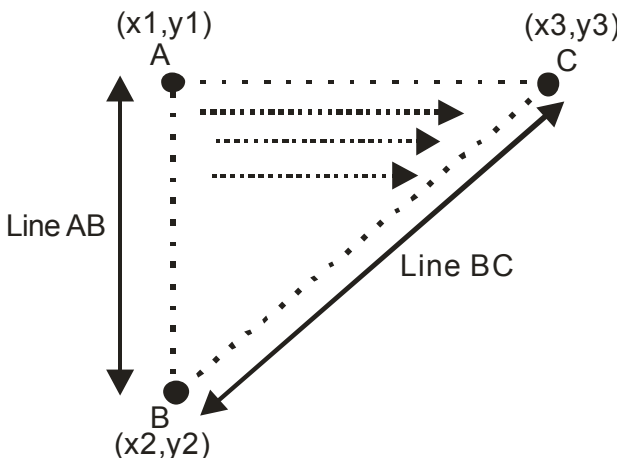


Figure 1.5: Sample Triangle

In Figure 1.5, a sample triangle with vertices A, B and C is given and the corresponding coordinates at A, B and C are (x1, y1), (x2, y2) and (x3, y3). First using linear interpolation [8] the values along Line AB are obtained. Similarly values along Line BC are obtained. Then, by interpolating between AB and BC the entire triangle can be filled. The first few test LUTs in this research were all computed using simple linear interpolation.

1.7) The LUT

As mentioned earlier, the LUT does not contain only row and column values. Based on the calibration results, each LUT value also contains vital information regarding the alpha blending associated with that pixel. In a multi-projector system, pixels from one input frame could be routed to more than one output frame. Assume there are four input frames x0, x1, x2 and x3 and four output frames y0, y1, y2 and y3. Frame y0 may have inputs from any of x0, x1 ,x2 and x3. So having just row and column value in the LUT is not enough, a channel ID (IDentifier) is required to identify the source of that pixel. Sometimes certain pixels in the output could be blank and it will be a waste of time trying to decode the address of these pixels. In that case, there is a flag indicating whether the pixels are valid or can be ignored.

Thus depending upon the application and requirements a lot of parameters can be added to the LUT value. After going through various hardware constraints and the requirements of the application, the LUT used for this research had values which were 32 bits wide.

Table 1.1: Address Bits split up

BITS	NAME
31	IGNORE
30-23	ALPHA
22-21	Channel ID
20-0	Memory Address (10-0 Column) (20-11 Row)

Table 1.1 shows the exact layout of each LUT value. The IGNORE bit tells if the pixel is ignored or not. The alpha value is an eight-bit value that is multiplied with the resulting output pixel value once its value is fetched from the input frame through reverse mapping. The channel ID tells the source of that pixel, whether the pixel has to be fetched from input frame ‘x0’ or ‘x1’ or ‘x2’ and so on. The row and column values that are obtained from the linear interpolation form the lower twenty one bits. Currently there is provision for a maximum resolution of 2048x1024 (11 column bits and 10 row bits or vice versa). More bits can be added to the LUT values to contain more information.

Chapter 2: Previous Work

A description of some concepts involved in the caches of general purpose processors and also on cache designs used in texture mapping applications is given in this section.

2.1) Cache in General Purpose Processors

Memory latencies have long been a problem in computer systems. Even though the memory sizes have recently increased and the size of the chips has been reducing, there is still the problem of latencies involved in accessing data from memory.

Often, a lot of CPU (Central Processing Unit) cycles are wasted while waiting for data from memory modules. To alleviate this problem, the concept of cache was introduced. A cache is basically a small memory which is accessed by the CPU first before searching in other memory modules like RAM or storage devices like disks. For this reason the cache is sometimes referred to as the first level of memory hierarchies in a computer system [5]. A cache is small in size when compared to the second level of memory hierarchies, like RAM. Owing to its small size, a cache can reside where the processor takes fewer cycles to access data from it. A cache primarily tries to exploit the principle of locality; that a memory location accessed will be accessed soon or memory locations neighboring the current memory location being accessed will be accessed in the near future.

Primarily, there are three kinds of cache designs, direct mapped, set associative, and fully associative caches. In a direct mapped cache, a memory address is strictly mapped onto a single location in the cache. If an address is not present in the cache, the old address is immediately replaced by the new one.

In a fully associative cache, a memory address can be mapped to any location in the cache. A cache location can be chosen from all the various possible locations to store a value with a given memory address.

In a set associative cache, a memory address is mapped onto a particular cache set. Within that set, the address can be mapped to any cache location. There is associativity within a set.

Direct mapped caches are the easiest to implement while fully associative are the most difficult. A set associative cache design is in between the two in terms of advantages and disadvantages.

In set associative caches, one of the blocks within a set has to be replaced when there is a cache miss. This calls for a replacement strategy, which decides which block has to be replaced. Some of the replacement schemes are Least Recently Used, Most Recently Used, and First in first out.

AMD and Intel's computer processor datasheets provide information about their cache architectures and also give details on their timing diagrams and state machines [12, 13].

Przybylski, Horowitz and Hennessy discuss the various trade offs one must consider while designing caches [14]. They discuss how speed of the process varies with change in size of the cache, how the number of sets in a set associative cache influences the miss rate, how the size of the tag (part of the memory block address) influences the miss rate of a cache. The concept of multi-level caches is also discussed. Their work simulates several test benches for all the above mentioned criteria to get a proper understanding of the cache design process.

2.2) Graphics Related Work

Hakura and Gupta have proposed cache architecture for texture mapping [10]. In computer graphics, mip-mapping is the process of adding pre-calculated collection of bitmap images to a main texture to increase rendering speed. Hakura and Gupta start their design by considering existing mip-mapping ideas and provide two kinds of implementations, one a "base non-blocked representation" and the other a "blocked representation". The base non-blocked representation stores pixel values of RGB (Red

Green Blue) in contiguous memory locations so that addressing calculations are minimal. The blocked representation is a technique where textures within a specific block of square area are all placed in consecutive memory location and accessed sequentially. The addressing schemes for the blocked representation are complicated and may result in more than one step to map the memory address to a cache location. They try to find a good design by varying block sizes and cache sizes and examining the miss rate.

The APR accesses memory in a way that is different from texture mapping, but the calibration results (discussed in previous sections) yield a similar data set to work with, with the concept of reverse mapping of pixels. Hakura and Gupta discuss a cache design for real time rendering, but one of the ideas behind the APR's cache design is the knowledge of the way the output pixels are mapped to the input pixels offline. Also, Hakura and Gupta focus on both temporal and spatial localities, but a close examination of the APR only exhibits spatial locality. Each pixel is accessed only once per frame. A cache large enough to take advantage of temporal locality would have to hold the entire input image. A unique feature of the cache design in the APR is that the cache architecture is very simple. The order in which pixels are accessed is never modified, unlike the memory address representations discussed by Hakura and Gupta.

Igehy, Eldridge and Proudfoot discuss an interesting prefetching technique for texture caches [11]. They have extended the idea of Hakura and Gupta by adding the prefetching feature to the texture cache. First, a robust texture prefetching architecture is proposed (block diagrams see [11]). Then the textures are stored in "super sized" blocks in memory so that memory addressing is easier and yields higher cache hits. Some of their test benches are interesting. They have also discussed the cache efficiency for various cases.

The design of Igehy, Eldridge, and Proudfoot is different from the APR design. There is no implementation of a separate prefetching architecture or storage of textures in specific memory blocks in the APR, but the previous work discussed above created many ideas for this research.

Krishnan describes the performance of the memory controller using just SDRAM on an older version of the Anywhere Pixel Router hardware in an earlier work [3]. Memory performance on the current hardware provides a baseline against which to measure cache performance.

Chapter 3: SDRAM Performance

The SDRAM is a memory storage unit capable of handling data up to several gigabits in size. SDRAM modules have high bandwidth with the introduction of DDR SDRAM modules. SDRAM access speeds of several hundred megahertz are possible. A SDRAM module is usually limited by factors such as CAS latency, RAS latency, and latency caused during periodical refreshing of SDRAM cells. These latencies make the accesses to random locations slow. Even if a fast microprocessor is used for a particular application, the microprocessor has to wait until it receives data from the SDRAM. There is a big problem when dealing with time critical applications where a delay of a few processor cycles could result in bad outputs.

The APR is time critical. If a frame cannot be processed within the desired time, that frame has to be discarded. If the delay spans a few frames then the output is slow and jerky. For example, in a system that outputs a resolution of 1024x768 at 60 Hz, 1024x768 pixels worth of data must be processed within 16.6 ms. If this is not done, then the frame is discarded which is not desirable. The main motivation for this research is overcoming the meager efficiency of the system when only the SDRAM module was used in the memory controller [3].

3.1) Design Values and Simulation Numbers

Before the results of the simulation are studied, it is necessary to understand the conditions under which the simulation is run. The Anywhere Pixel Router is implemented using a Xilinx Virtex 4 model XC4VLX40 running at 133 MHz. It is connected to eight 16 bit wide 512 Mb MT46V32M16 modules running at 133 MHz.

For increasing the speed of the application by means of parallelism, the LUT, input frame, and output frame are all stored in different RAM modules connected to the FPGA. This ensures that values in the LUT and sometimes even in the input frame can be pipelined to achieve speedup.

The MT46V32M16 RAM has only 16 bits of storage per every location, but the LUT and the input/output frame have 32 bits of data currently. So two DDRRAM modules are used for each LUT, input frame, and output frame by the FPGA so that 32 bits of data can be stored and read. The LUT and output frame are accessed sequentially.

The access time is calculated based on the number of cycles taken to process one frame of the image. The access time is the number of cycles divided by the frequency at which the process is running. Thanks to parallel execution explained in Subsection 1.3, the time taken to access a value in a column of an open row is 1 cycle, which is denoted as $t_{sr.}$ The time taken to access a value in a column of a row that is not open involves opening the new row and then accessing the desired column. This time is denoted as t_{cr}, which involves one RAS delay and one CAS delay, a total of 8 cycles.
If the pixel is a IGNORE pixel, then it takes one cycle to process it. The simulations were conducted for frame resolution 1024x768 pixels outputted at 60 Hz. The multi-projector system used was a four input, four output system, and the controller processes one frame after the other in succession. All simulation results are in reference to outputting a single frame.

3.2) Simulation Results (RAM only)

The simulations are run in C taking into account the number of cycles mentioned above. In Section 2, the nature of LUT values could be predicted. In a multi-projector system, the projectors could be aligned in any random fashion by the user. They could be straight aligned exactly adjacent to each other, or perpendicular to each other or in an angle to one another, so the LUT mapping function could be any rotation. The output can also be skewed if the projectors are rotated around the vertical and horizontal axis. The anywhere pixel router memory subsystem should be able to deliver output at the required rate no matter what the LUT function is.

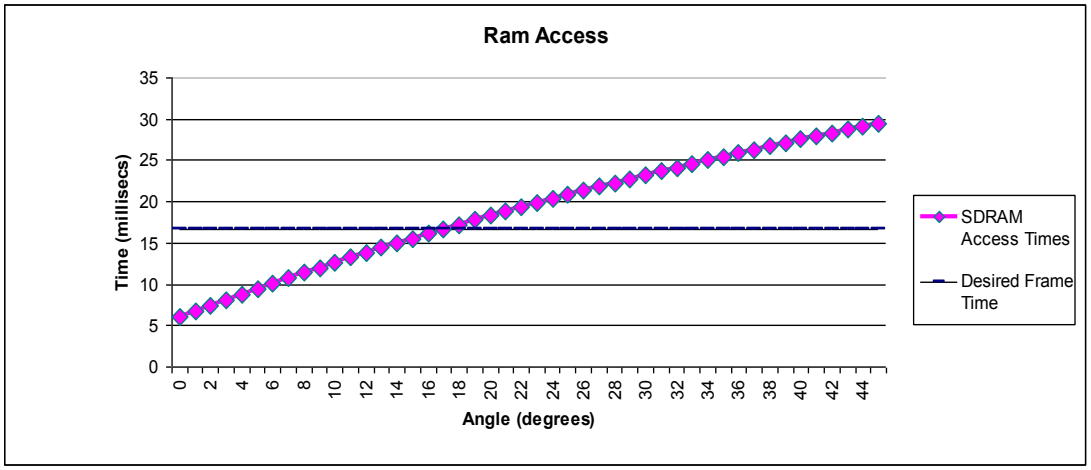


Figure 3.1: SDRAM Access Times

Figure 3.1 shows the performance of the specified SDRAM unit for different angles in the LUT ranging from 0-45 degrees. The access time in each case corresponds to the average time the process takes to completely transfer one frame from the input to the output based on the LUT. One frame corresponds to a single display frame of 1024x768 pixels. The actual LUT's function might not be just an angle of rotation, but in fact might be a more complex function. These simulations are carried out to get an idea of how fast the memory units are without cache.

In the above simulation, the process having the LUT for angle 0 degree (identity transform) is the fastest, as the input frame access is completely sequential. As the angle of rotation decreases, the number of sequential accesses to the current open row of the input frames increases, and the time taken decreases. Figure 6 shows a plot of average access time as a function of rotation angle. The behavior is in accordance with the prediction. LUTs having higher angles take more time than the time allocated for one frame which is 16.6 milliseconds. As soon as the angle is slightly greater than 14 degrees, the access time becomes too long to support the required frame rate. Table A.1 in Appendix A has the simulation results and numbers for this simulation. Figure 3.2 shows the memory efficiency rate (MER) for the various angles. The MER is the ratio of the number of pixels in the image to the total number of memory cycles needed to transfer all

the pixels from input to output, multiplied by 100%. The MER for angle 0 is close to 100% and the efficiency decreases as the angle increases.

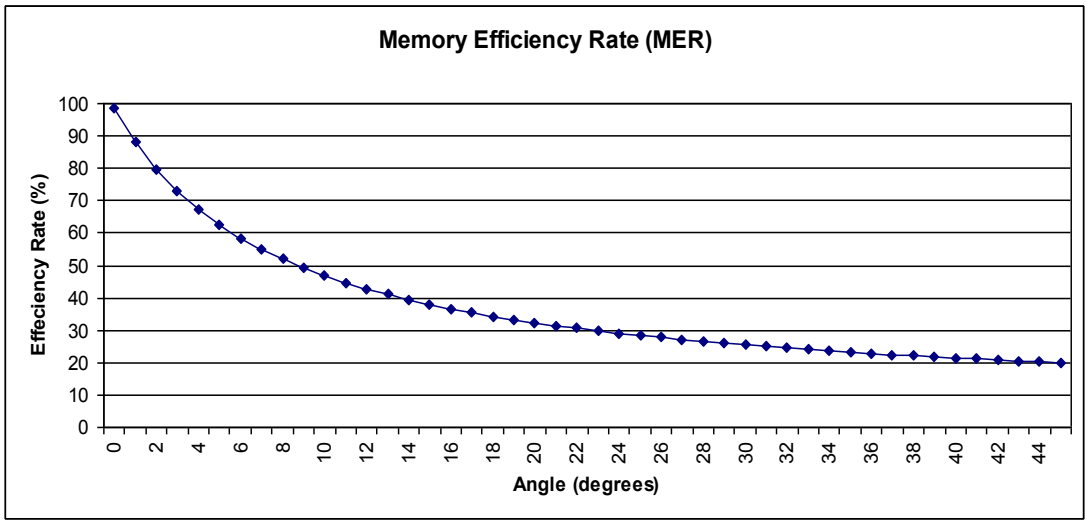


Figure 3.2: MER

Chapter 4: The Cache Design

Even for relatively small angles the SDRAM alone cannot keep up with the desired frame rate of 60 fps (frames per second). Some form of cache can enhance performance enough to make the Anywhere Pixel Router run in real time. This section discusses the usage of a simple direct mapped cache and checks its efficiency. Later the set associative cache design will be discussed in Section 6.

4.1) Cache Parameters

A cache is a small memory unit that comes in between the processor and the main memory. When it wants to access memory, the processor first checks the cache for the value. The processor uses the value stored in the cache if it is found. Otherwise, it must fetch the data from main memory. In the APR, the cache is present in the block RAM of the FPGA. The main parameters of the cache are:

1. '*l*' – number of cache lines.
2. '*w*' – width of each cache line in 32-bit words.

Together '*l*' and '*w*' determine the size of the cache. The capacity of the cache is "*l* x *w* x 32" bits. A larger cache will improve the hit rate compared to a smaller cache for the simple reason that it is able to store more SDRAM contents than the smaller cache.

In practice, the size of the cache is limited. For instance, in the APR the FPGA has a limited amount of block RAM. Only a small portion of this block RAM can be used for the cache because the FPGA is running a number of other controllers that portions of the block RAM already. The size of cache in APR is restricted to a maximum of 4K memory locations due to hardware resources constraints. Larger caches mean better performance, but usage of more block RAM space. A cache having 4K memory locations uses 128Kb (Kilobits) (4096 x 32) of block RAM. In the FPGA (Xilinx XC4VLX40) that is being used, there is 1728 Kb of block RAM [6]. A 128 Kb cache would correspond to about 7.5% of the block RAM.

4.2) Hardware Design

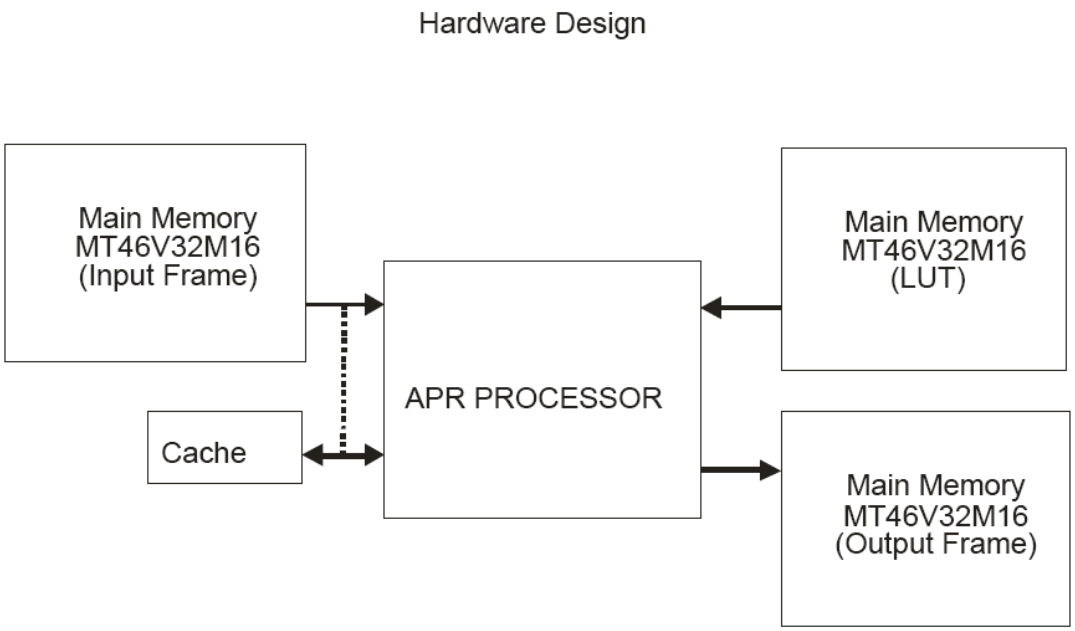


Figure 4.1: Inclusion of Cache

Figure 4.1 shows the memory hardware structure for the APR. The FPGA reads from the LUT, writes to the output frame, and these two are sequential. The FPGA reads from the cache first to check if the desired value is present. If the value is found in the cache, it means a cache hit. If not, the APR memory controller reads from the input frame to get the desired value and also fill the entire row in the cache from the input frame (the dotted line in figure).

4.3) Simulation Parameters

The simulation parameters used for the cache are all the same as those described in Section 3 for the SDRAM. The only difference is the introduction of cache hit cycles and cache miss cycles instead of same row access and change in row access. Figure 4.2 shows the logic flow when incorporating a cache.

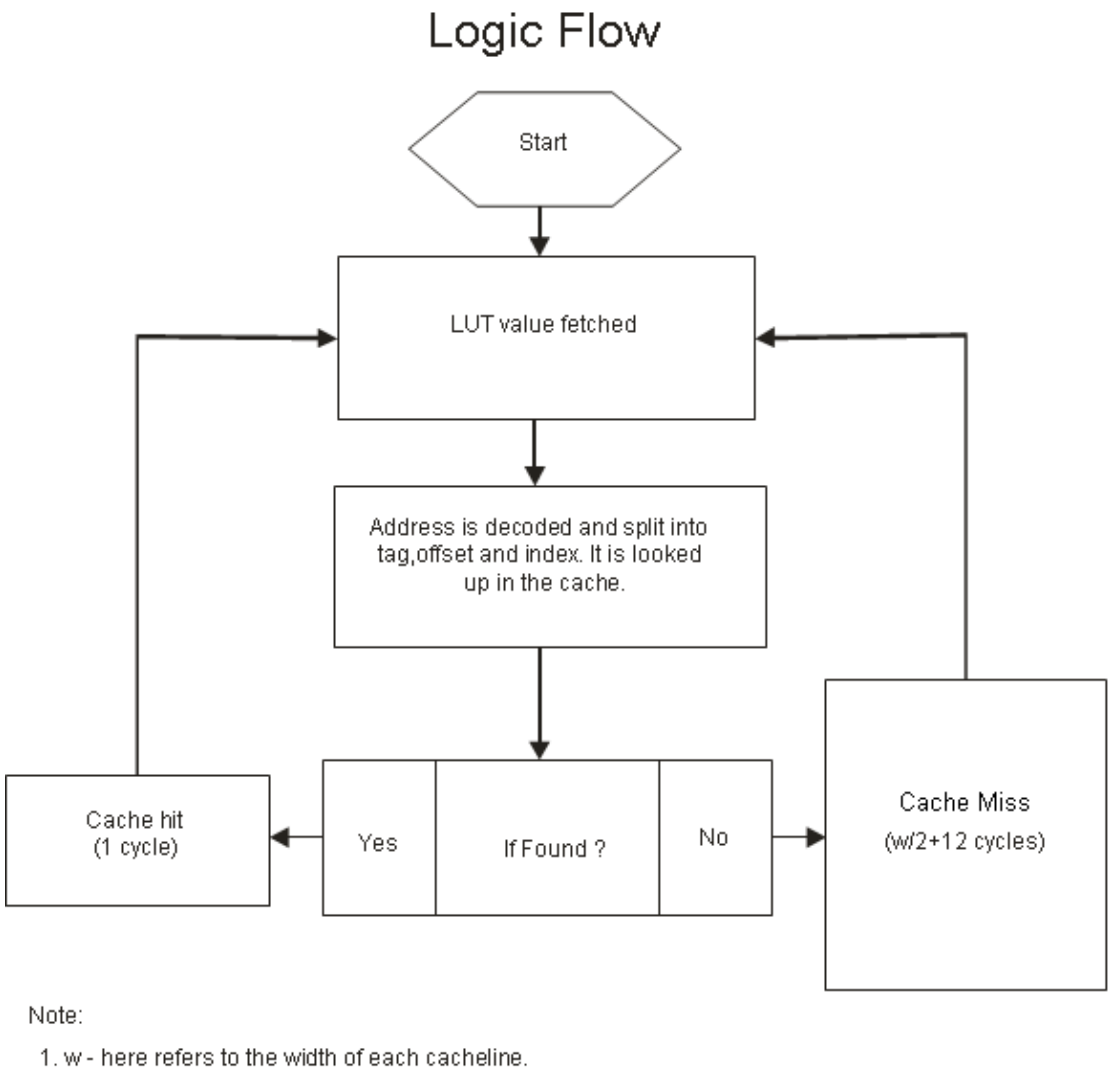


Figure 4.2: Cache Logic Flow

Note that the logic flow shown above corresponds to a direct mapped cache. The logic flow is the same for a set associative cache except that the address bits are decoded into offset and block only, not tag, offset and index like direct mapped.

A cache hit means only one cycle, while cache miss means $w/2+t_{ras}+t_{cas}+3$ cycles, where 'w' is the width of each cache line. For the APR, t_{ras} is 7 cycles, and t_{cas} is 2 cycles. The additional 3 cycle latency is the overhead for the memory controller in the FPGA to fill a cache line. Basically, the cache miss means fetching the missed value, along with its

entire cache line from the main memory. Since the RAM used here is a DDR, it takes only w/2 clock cycles for fetching the cache line.

4.4) Caching Function

A good caching function should match the structure of the LUT. This means that a proper division of the address bits in the LUT into tag, offset and index is required. Assume a cache with size "l x w", where "l" is the number of cache lines and "w" is the width of each cache line in words. The LUT/ input /frame and output frame have 20-bit addresses. The upper 10 bits correspond to the memory row and lower 10 bits correspond to memory column.

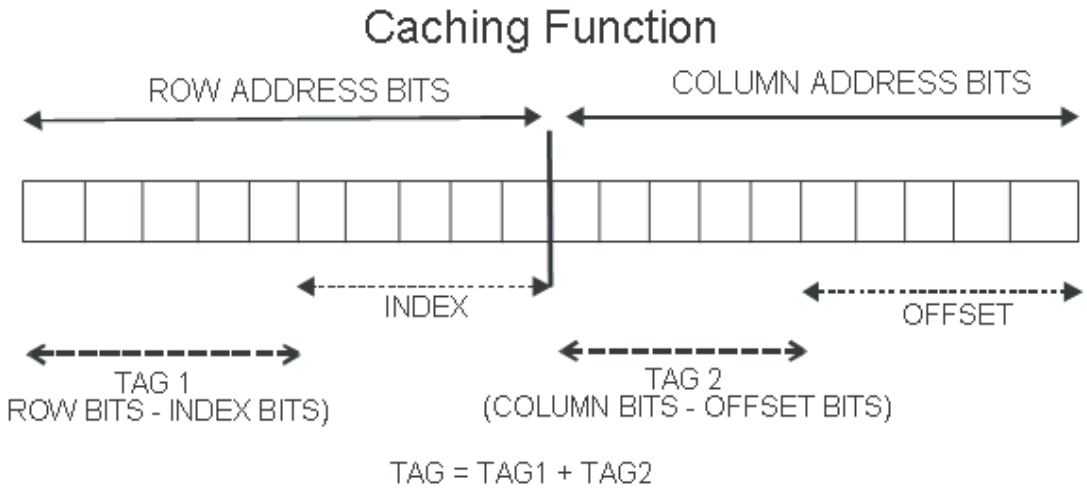


Figure 4.3: Caching Function

Figure 4.3 shows how address bits are allocated in a LUT address. From the nature of the LUT, it is evident that adjacent RAM rows should map to adjacent cache lines so that hits can be increased. A small increase in a RAM column should correspond to a small offset in the cache. Hence the lower row bits of the address are chosen as index values, the lower column bits of address are chosen as offset values and the remaining address is chosen as the tag. This mapping guarantees that a pixel will never map to the same cache index as the pixel with in the same image column, but one row above or below in the image. The caching function is the same for a set associative cache, except that as the number of sets in an associative cache increases, the number of index bits reduces and the tag bits increase.

4.5) Simulation Results

The simulation was conducted for a sample LUT whose function was a 45 degree rotation. The number of cache lines and width of each cache line were varied to show how access times vary for different cache sizes with the same LUT function.

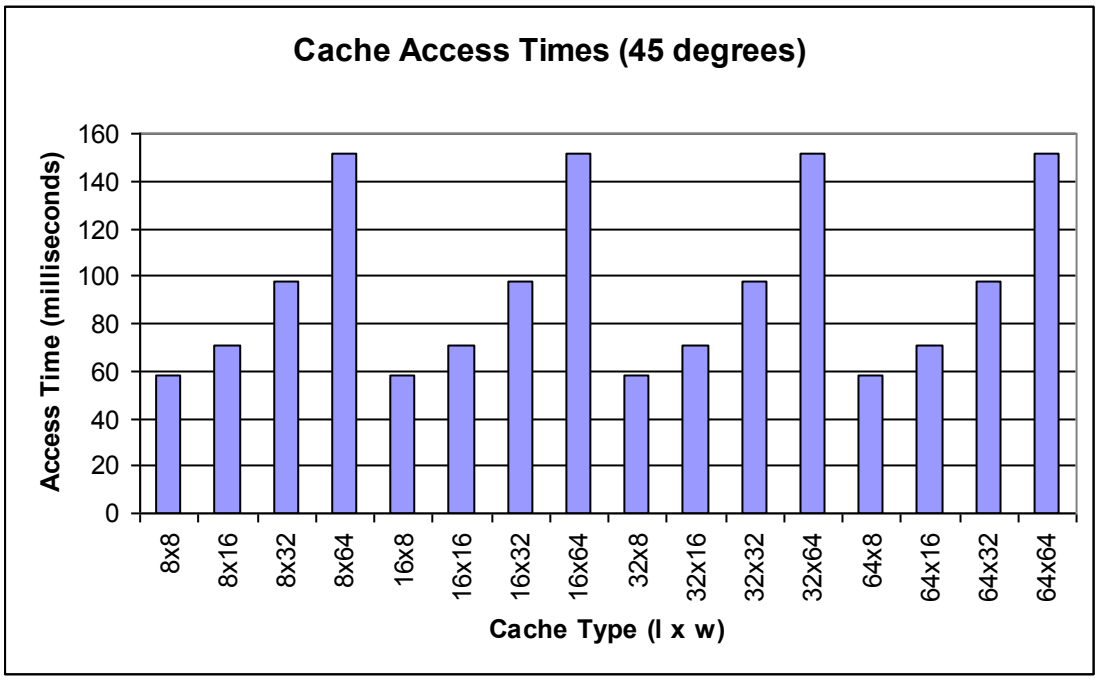


Figure 4.4: Access Time Vs Cache Size (45 degrees)

The minimum cache time obtained is around 58 milliseconds, which is far higher than the access time of a system with no cache and just a SDRAM. This is not at all desirable as it would be a waste designing the cache if the system is slower than the existing one without cache. This problem is fixed in later sections. For a 45 degree angle of rotation, every adjacent LUT value corresponds to a different row value in the input frame. Even with cache, the APR will have cache misses on every cycle and the cache line will have to be filled. Filling a cache line is a big penalty because of the number of cycles used there. The main goal is to come up with some algorithm that increases the cache hit rate. Complete data are included in Table A.2 in Appendix A.

In Figure 4.4, the access time increases as width of each cache line increases for a cache with the same number of cache lines. This is in accordance with the fact that bigger width means bigger penalty while filling up the cache line during a cache miss.

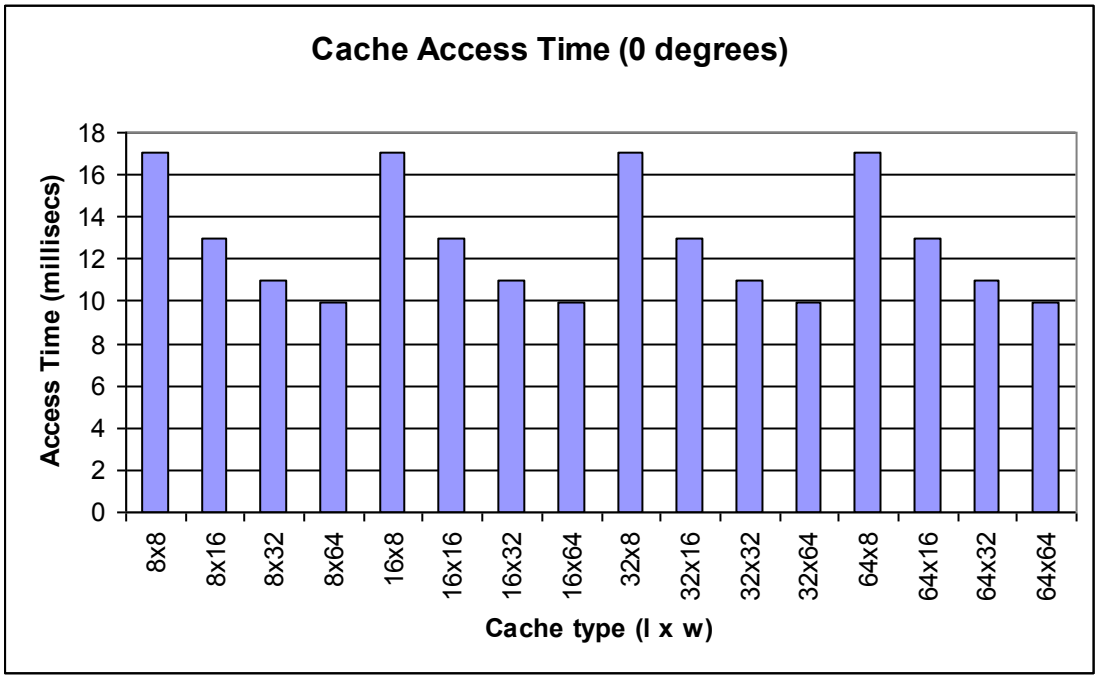


Figure 4.5: Cache Size Vs Access Times (0 degrees)

Figure 4.5 shows how the cache performs with a 0 degree angle of rotation. Caches with wider cache lines are better. Access times for the 0 degree rotation are better than those for a 45 degree rotation. The design with no cache is better than the design with cache in the case of 0 degree rotation. The 0 degree rotation is a special case, as most pixel accesses in the input frame are sequential. As the angle increases the cache becomes less efficient. The values plotted in Figure 4.5 are given in Table A.3 in Appendix A.

Chapter 5: Concept of Memory Blocks

In the previous section, the simulation results showed that the introduction of a cache to the existing design does not improve the speed of the application by itself. In fact, the memory system slows down due to overhead caused by fetching cache line data that are never used. This section analyzes the problem and provides a solution.

The hit rate of the cache has to be improved in order to achieve faster access times. The nature of the LUT is heavily dependent of the projector setup and this was seen in Section 1. For example, in case of a 45 degree rotation in the LUT, every adjacent LUT location points to a different memory row in the input frame. In the case of a 25 degree rotation, on average every third or fourth LUT location points to a different row in the input frame.

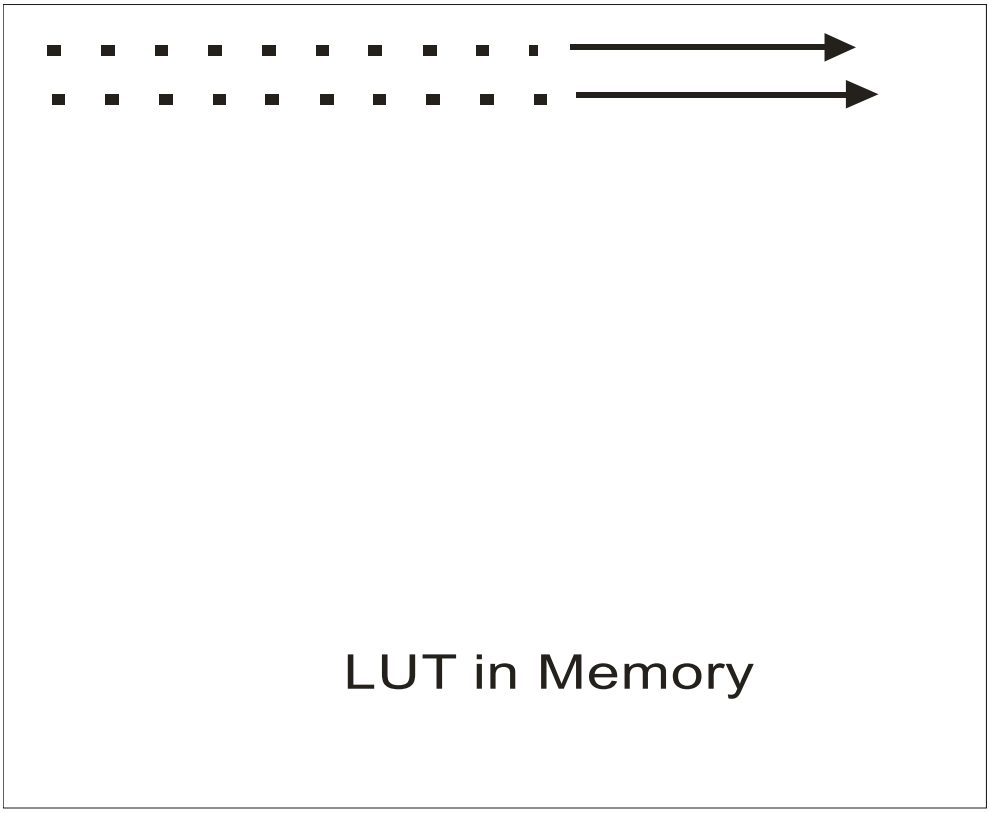


Figure 5.1: LUT in memory

Figure 5.1 shows the way the LUT values are accessed from the SDRAM. It can be seen that the process is sequential. This implies that reading LUT values is not a constraint. The LUT maps input pixels to output pixels through reverse mapping. When the input pixels are fetched from the input frame, the process may not be sequential. It will be sequential only if the LUT function is an identity transformation (0 degree rotation). Even for small angles of rotation like 15 degrees, the input frame access will not be sequential.

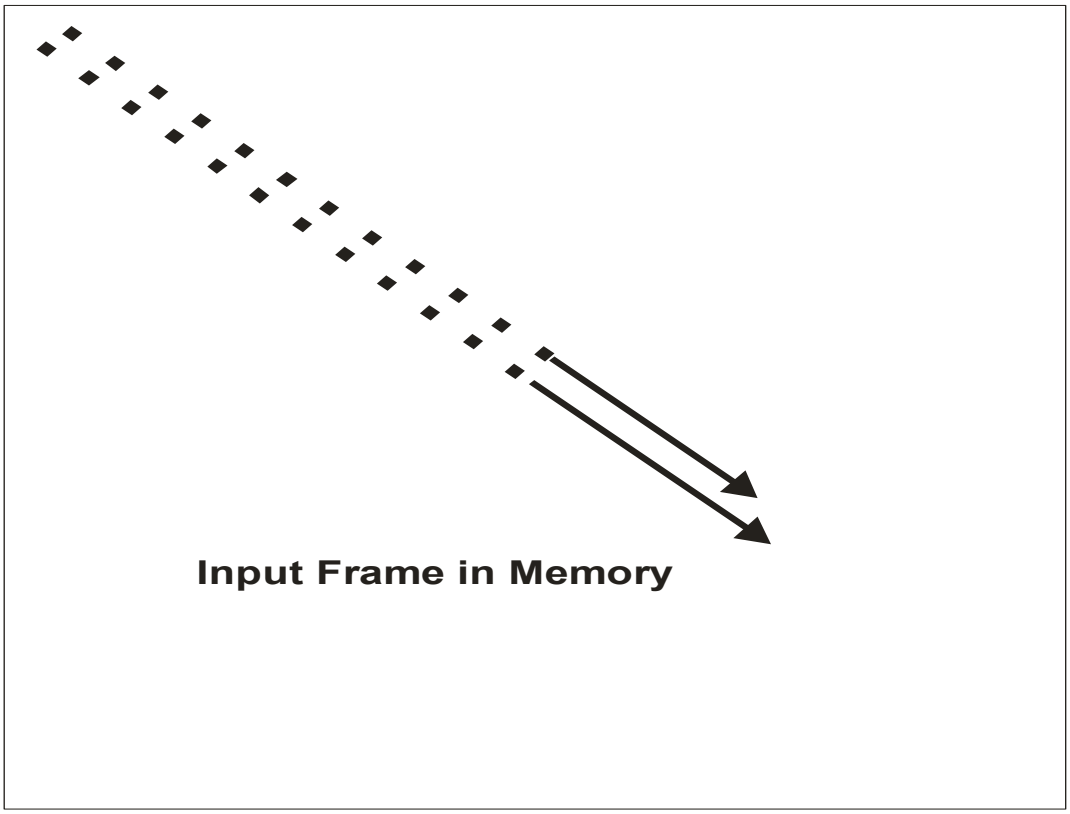


Figure 5.2: Input Frame Access

Figure 5.2 shows how the input frame would be accessed in case the angle is 35 to 45 degrees. Each spot in the figure represents a memory location.

It can be seen that access is not at all sequential. In case of just using the SDRAM module, this kind of access would result in a large access time, due to many row changes. When a cache is present then every adjacent LUT access results in a cache miss and a heavy penalty to fill the cache line. In the APR, the cache has to be designed keeping in mind that there is a high chance of accessing memory locations neighboring the current

memory location being accessed; that is in the previous memory row or the next memory row. The same memory location is never accessed twice in one frame unless there are some rounding errors and approximation errors while interpolating the calibration values, so the concept of temporal locality is not of much importance.

Even though the input frame access is not sequential, there is a pattern which describes the access. This pattern is very useful for the analysis. Even if the LUT function is just not an angle of rotation, it is still possible to find a particular pattern in which the input frame is accessed for every LUT. This is because the LUT is obtained from calibration results and the calibration results correspond to the way the projectors are positioned.

The entire LUT is calculated offline and then loaded into the memory. The LUT is not processed during run time. This means that the pattern in which input frame accesses are made can be predicted offline from the LUT values. This pattern could be useful for the cache design. If LUT values are accessed in a different order then the input frame accesses can be made more sequential. A proper balance between the LUT and input frame accesses gives a faster access time.

5.1) Memory Blocks

Consider the case of a LUT having 45 degree rotation function and the pattern is predicted offline before the LUT is loaded into memory. Now the entire LUT memory space could be divided into small squares or memory blocks. As an example, consider a 64x64 pixel LUT, input frame and output frame. A block of 64x64 pixels would easily fit inside a single bank of the memory currently being used.

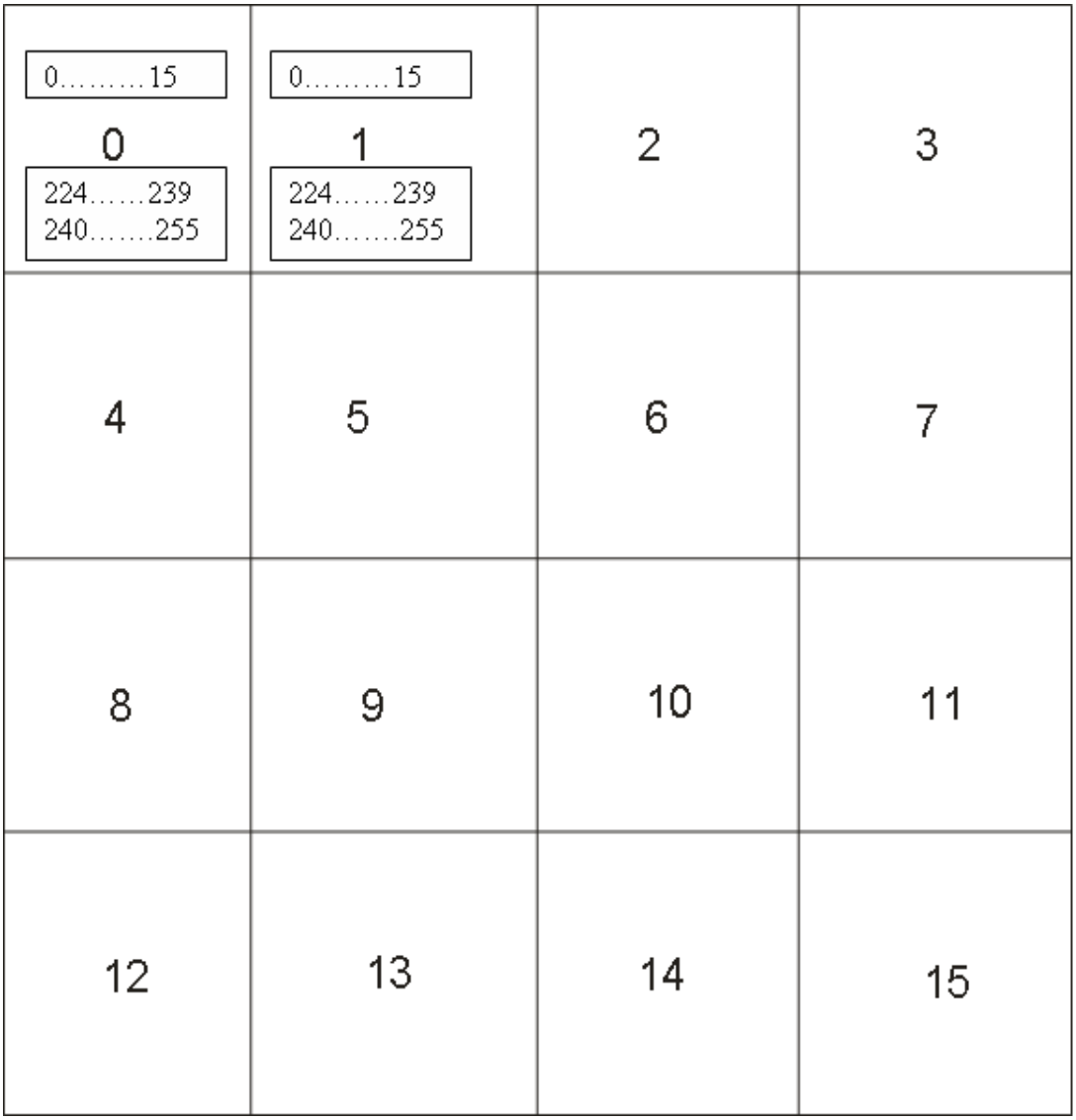


Figure 5.3: Division into Memory Blocks

Assume that the 64x64 pixel LUT is divided into 16x16 pixel memory blocks. Figure 5.3 shows the division of the memory space into smaller blocks. The blocks are arbitrarily labeled to demonstrate the concept behind these blocks. Now assume that this 64x64 LUT has a 45 degree rotation function in it.

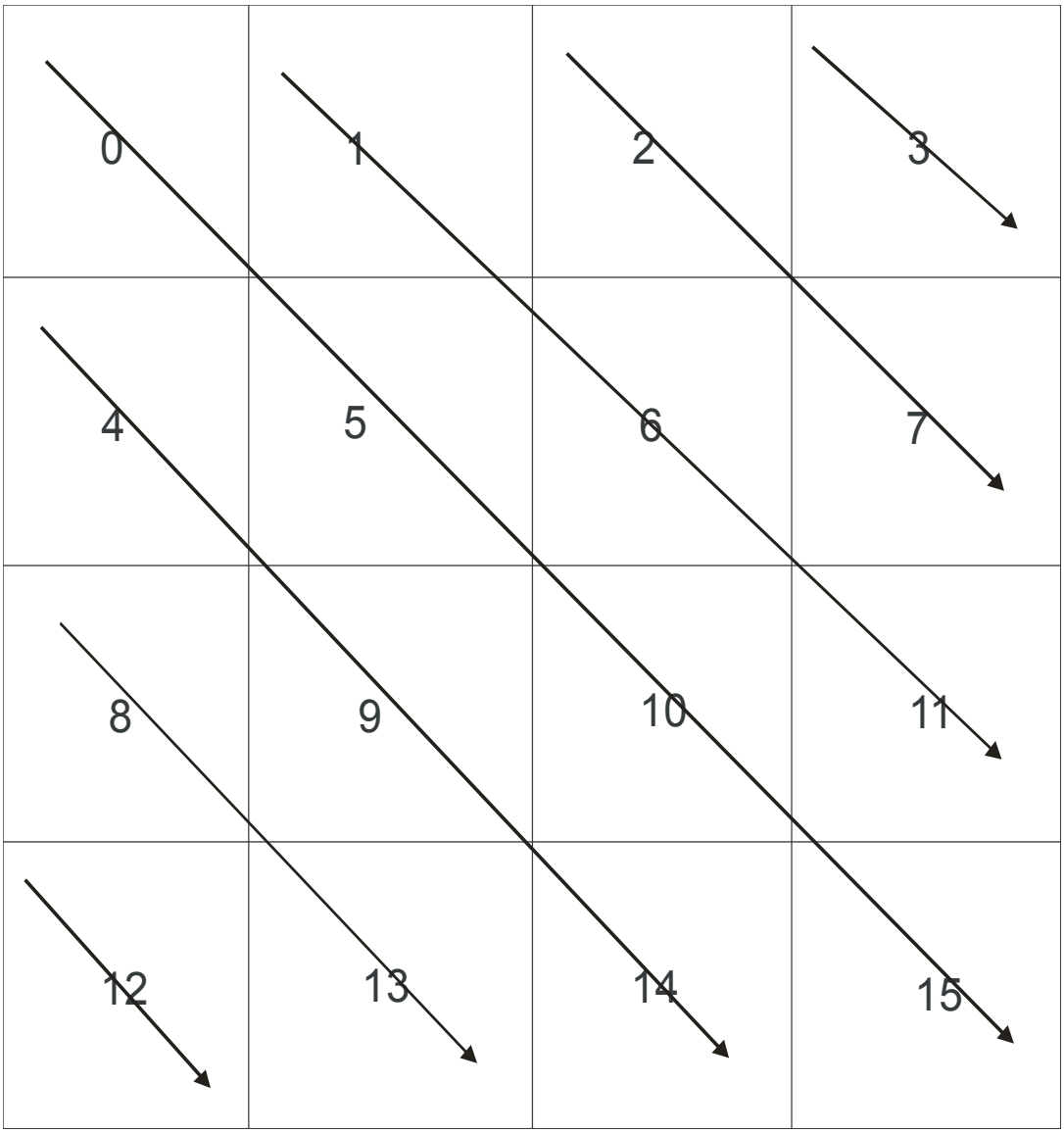


Figure 5.4: LUT access using blocks

As shown in the Figure 5.4, assume that the LUT is accessed in blocks (3, 2, 7…….12). A visible pattern is noticed in the access which is very simple to guess, it is following an angle of rotation close to 45 degrees. If the LUT is accessed in blocks like this, the input frame access would be something like in the Figure 5.5.

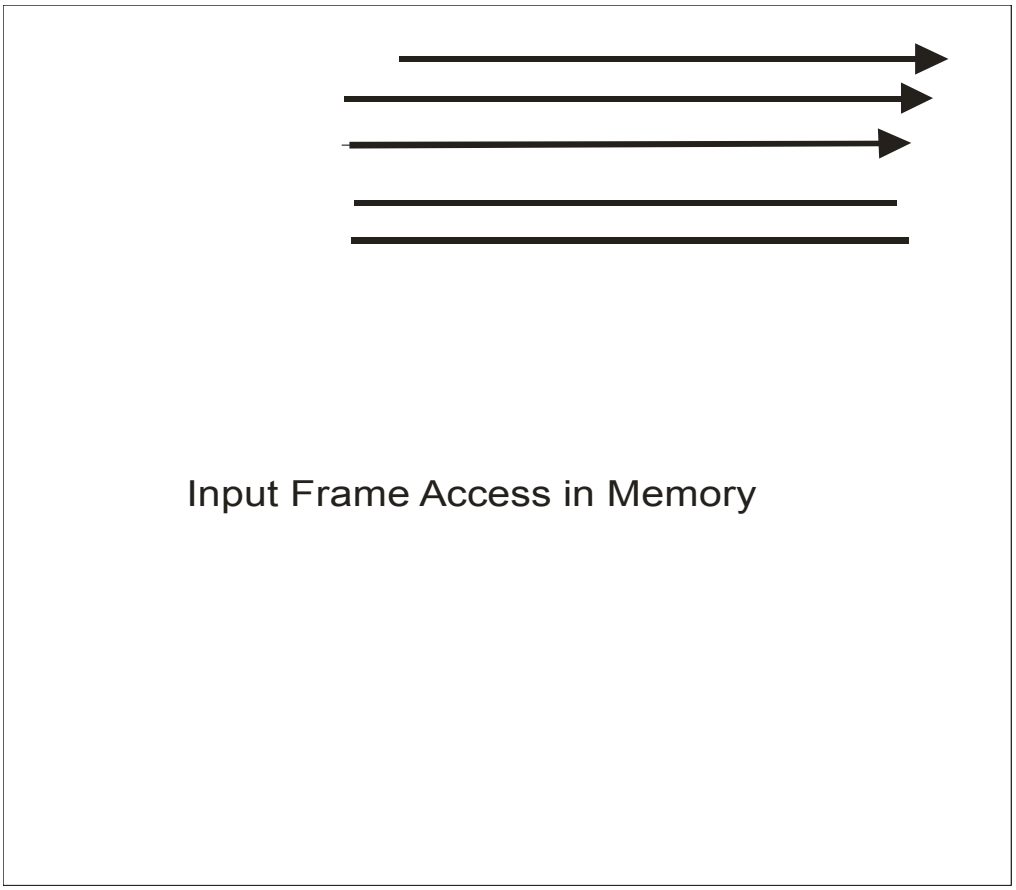


Figure 5.5: Input frame access in case of Blocks

A group of input frame rows are referred to repeatedly; that is, accesses will be sequential in nature.

It can be seen that a set of 5-10 lines in the memory space where the input frame is stored is continuously accessed if the LUT is accessed as shown in Figure 5.5. It is also easier to store 5-10 lines of memory (or just part of the 5-10 lines) in the cache without the need to replace them for a long time. The important part in this design is to predict the LUT characteristics offline and then decide the sequence of blocks that have to be accessed.

When using blocks, the LUT and output frame accesses are not totally sequential, but within a block it is almost sequential. Within a block, LUT values can be pre-fetched and stored in some array for reducing the access time.

5.2) Loading Block Sequences

If the LUT has to be accessed in blocks like described above, then the FPGA should be instructed to do so before the process begins executing. The sequence of blocks can be stored in a file and loaded when the LUTs are loaded in the following manner. Consider the example of a 45 degree rotation on a 64x64 pixels image. Divide the image into 16 16x16 pixels blocks.

b_j \ b_i	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Figure 5.6: Blocks Labeling

Any memory block can be identified if the 'b_i' and 'b_j' values (shown in Figure 5.6) are given. For example if b_i=2 and b_j=3, then it refers to block number 14. The order in which the blocks should be accessed can be calculated offline and stored in a file. This file of sequence numbers can then be downloaded into the FPGA or SDRAM from where the memory controller can access the information. This file is a few Kilobytes in size. The block sequence can be obtained from this file and stored in the block RAM of the FPGA. For an image occupying 1024 columns and 768 rows in memory, the sequencing numbers for a 64x64 block system require 8 bits. The entire list occupies around 5 Kb of block RAM which is a very small percentage of the block RAM present. Again, these numbers are in reference to the simulations that were conducted and by studying the FPGA datasheets. The format for a file containing the block sequence for LUT0 (projector 0 in a multi-projector system) is shown below.

```
Blocks0.txt
<channel id>  <Block size>
<bi0> <bj0>
..
.
.
<bin><bjn>
********End of file*******
```

Figure 16's block sequencing would be:

```
Blocks0.txt
0 16
3 0
2 0
3 1
1 0
2 1
```

3 2

0 0

1 1

2 2

3 3

0 1

1 2

2 3

0 2

1 3

0 3

********End of file*******

Please note that the memory blocks always correspond to the division of the memory space into blocks, not the actual image. The actual image could be 100,000x100,000 pixels, but it is stored only in a specific way in the SDRAM, depending on the number of columns and rows present in the SDRAM. Whenever blocks are mentioned, it refers to the breaking up of the memory space into blocks and not the actual image into blocks. A 45 degree rotation on the actual image could correspond to some other angle of rotation in the actual SDRAM memory space. The numerical value of the angle is of less importance than the memory pattern during input access, so that higher cache hits can be achieved.

In the above example, since the image is only 64x64 pixels it will easily fit well inside any modern SDRAM units. Assume a 16x16 block access. Given any b_i, b_j value, the corresponding block in memory can be accessed as follows:

```
For (j=bj*16;j<((bj*16)+16);j++) {
        For (i=bi*16;i<((bi*16)+16);i++) {
                J=row number of memory location,
                I=column number of memory location;
```

```
        End
End
```

Of course, j and i might have to be multiplied by some scaling factor or an offset needs to be added depending upon how the LUT is stored in the RAM. Block after block would be accessed in the LUT, and loaded into the output buffer after retrieving pixels from the input frame.

The example above deals with a very small image (64x64 pixels). The actual image could be large (maybe 800x600 or 1024x768).

5.3) Cache Simulations with Blocks

A series of simulations were carried out to estimate cache performance for a variety of inputs. The conditions and numerical values used for these cache simulations are all the same as used in Section 4, except that the LUT access is arranged as memory blocks and the angle of rotation is assumed to be 45 degrees. The system could behave differently for a different LUT function. In Section 9, an actual LUT that was created from calibration results and some sample LUTs will be simulated for results.

The simulation shows how the access times vary with block size and the size of the cache.

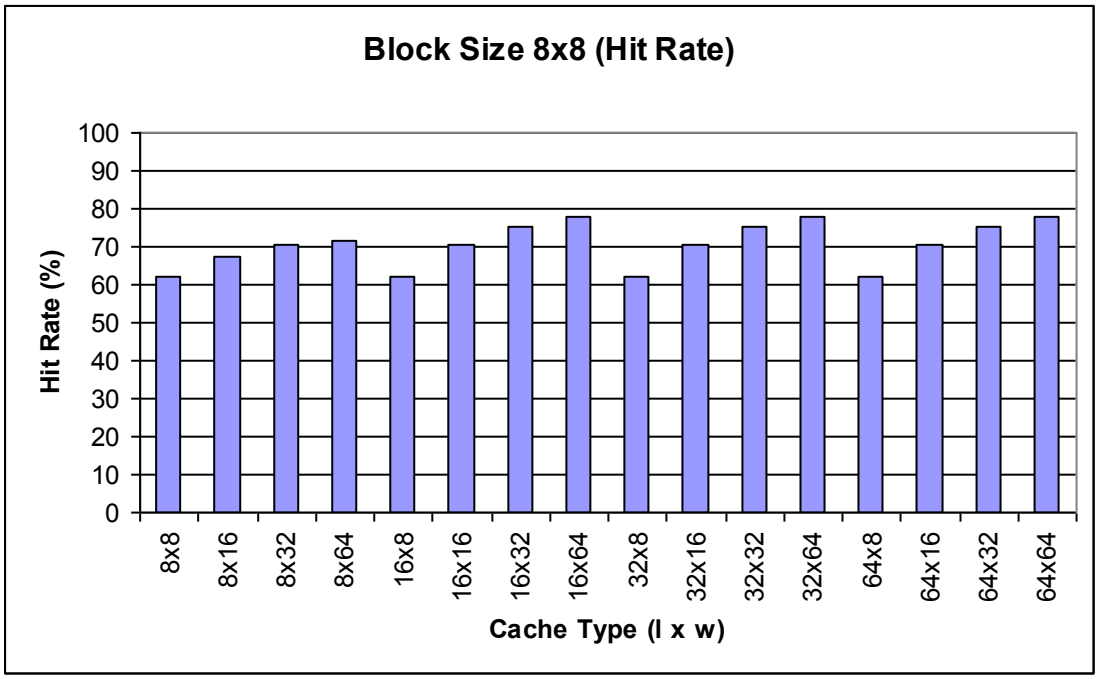


Figure 5.7: Hit rate for an access block size of 8x8 pixels as a function of a cache size for a cache with l lines and w words per line.

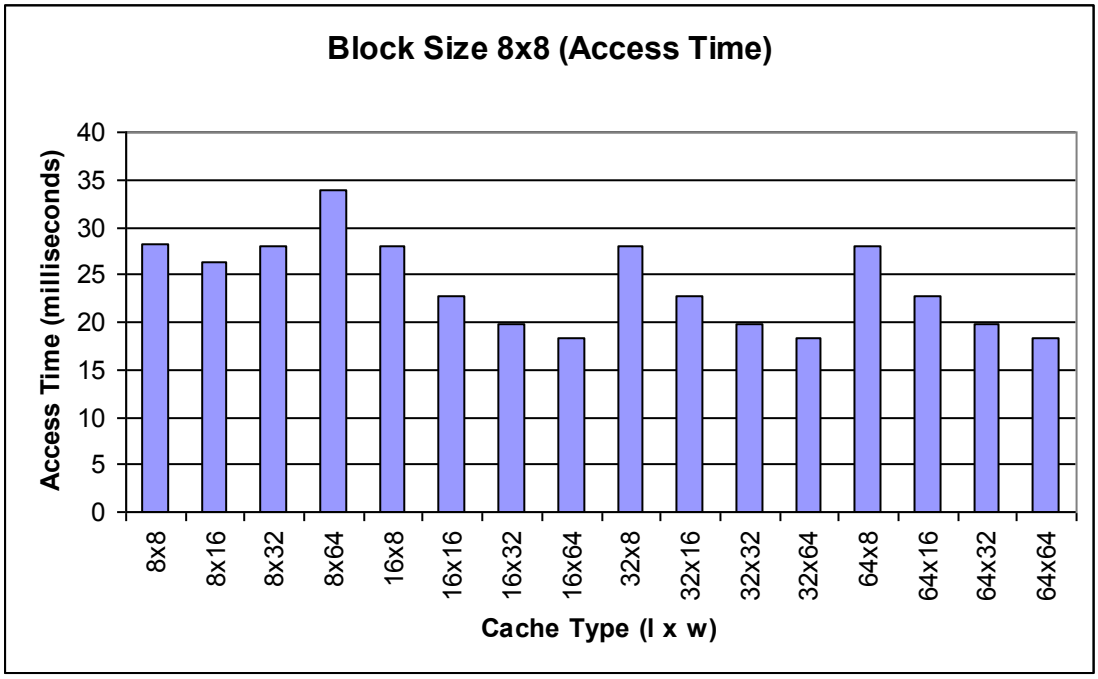


Figure 5.8: Access Times for a block Size 8x8 pixels as a function of a cache size for a cache with l lines and w words per line.

Figure 5.7 shows the hit rate for various cache sizes, and Figure 5.8 shows the access times for various cache sizes, when the LUT is accessed in 8x8 blocks. After the introduction of memory blocks, the access times have considerably reduced (almost one third the old access times). The access times decrease as the number of cache lines increases. This is due to the fact that more cache lines mean storage of more adjacent input frame rows. In Figure 5.7, the access times also decrease with increasing 'w'. This decrease is because there are more cache hits with greater 'w'. Even though filling up the cache line in case of a miss causes longer miss penalties, the block access ensures a higher hit rate. Table A.4 in Appendix A contains detailed simulation results for this experiment.

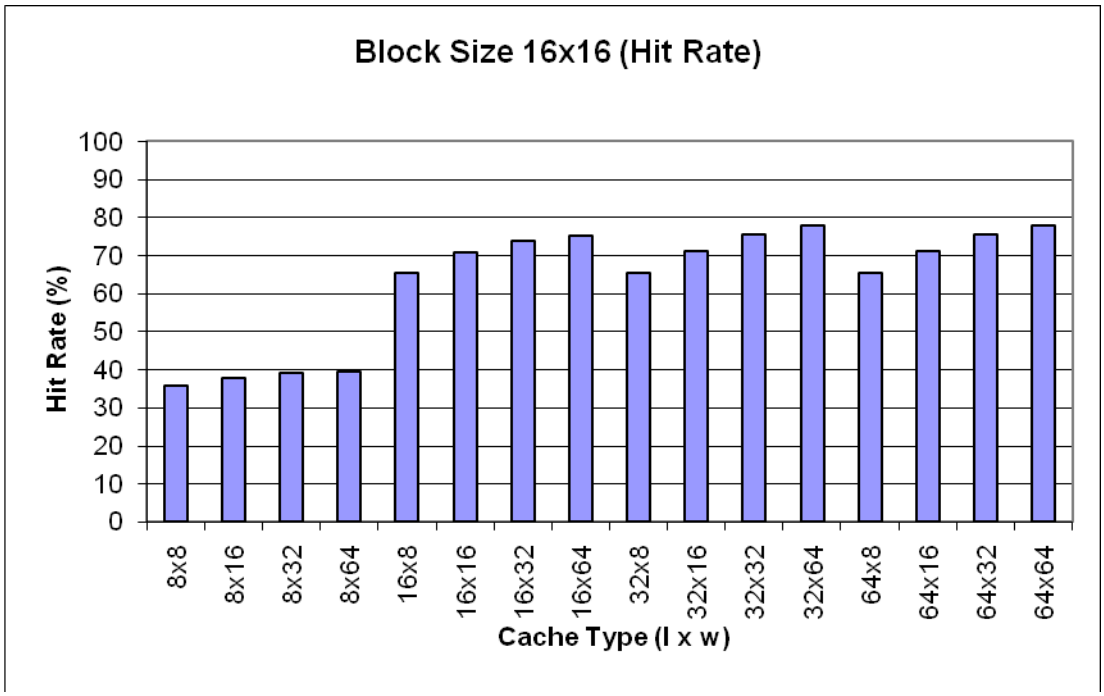


Figure 5.9: Hit rate for an access block size of 16x16 pixels as a function of a cache size for a cache with l lines and w words per line.

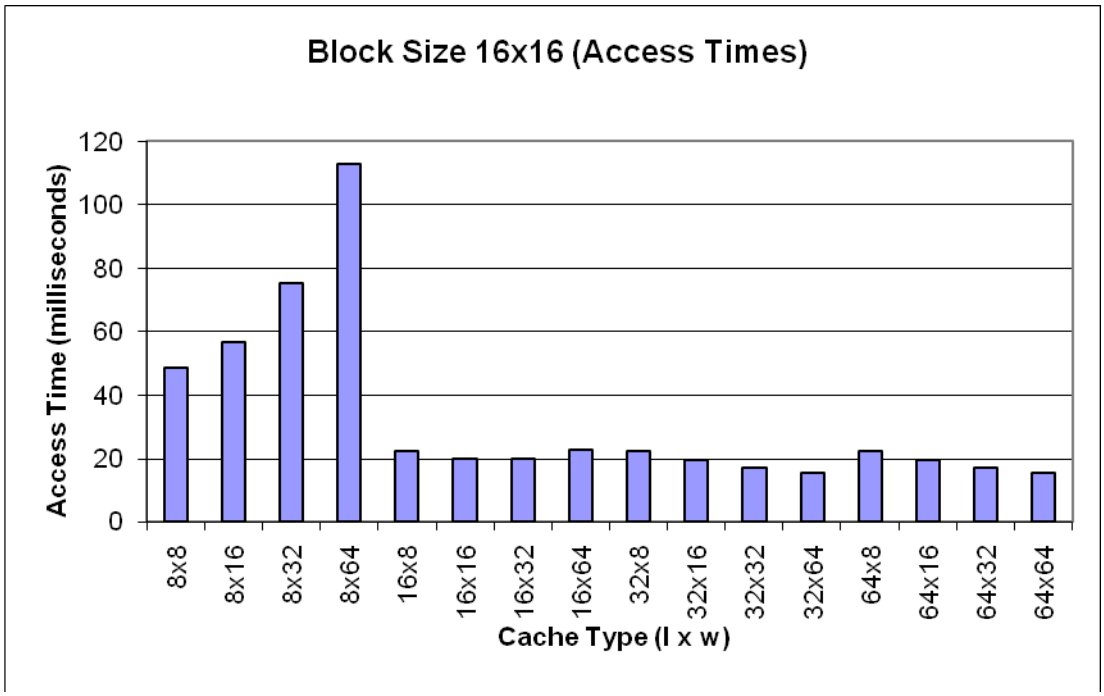


Figure 5.10: Access Times for a block Size 16x16 pixels as a function of a cache size for a cache with l lines and w words per line.

The simulation results shown in the Figures 5.9 and 5.10 are for block size 16x16. All the results discussed above for 8x8 blocks are found even in this case. When 'l' is less than the block size, the access times are higher than desirable. This is because the cache does not have enough cache lines to store all the input frame rows being accessed. This case is similar to accessing the LUT sequentially using cache but without blocks.

As with the 8x8 block simulations, simulations were run for block sizes 32, 64 and 128. The results are all similar to the ones discussed above. All the corresponding plots are shown in figures 5.11-5.16.

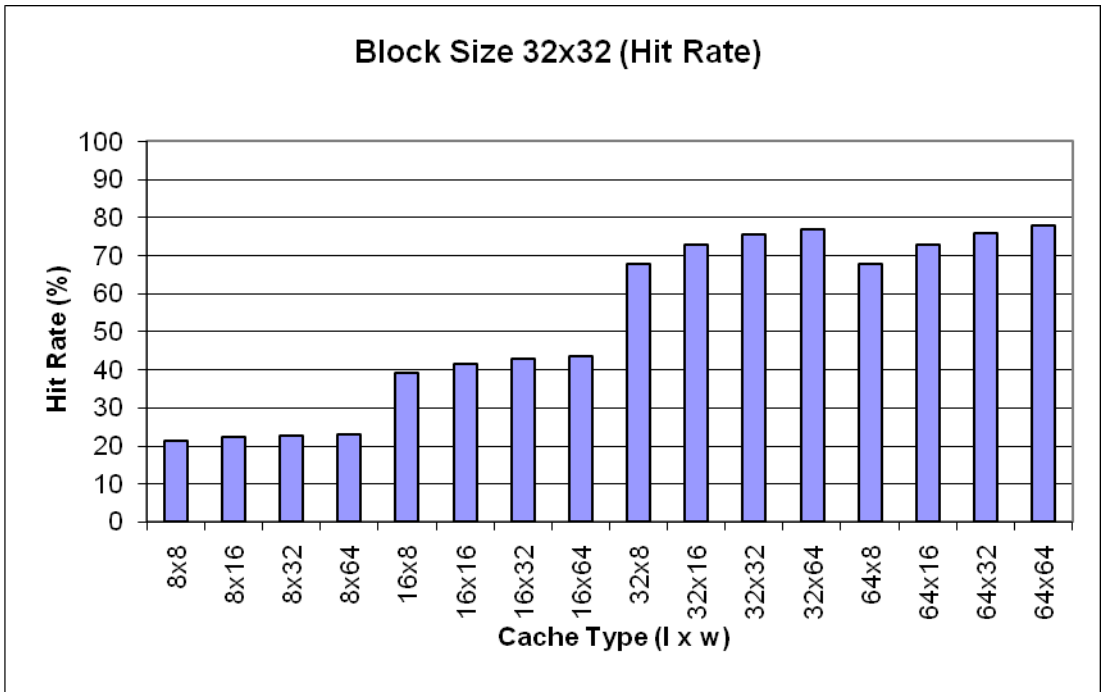


Figure 5.11: Hit rate for an access block size of 32x32 pixels as a function of a cache size for a cache with l lines and w words per line.

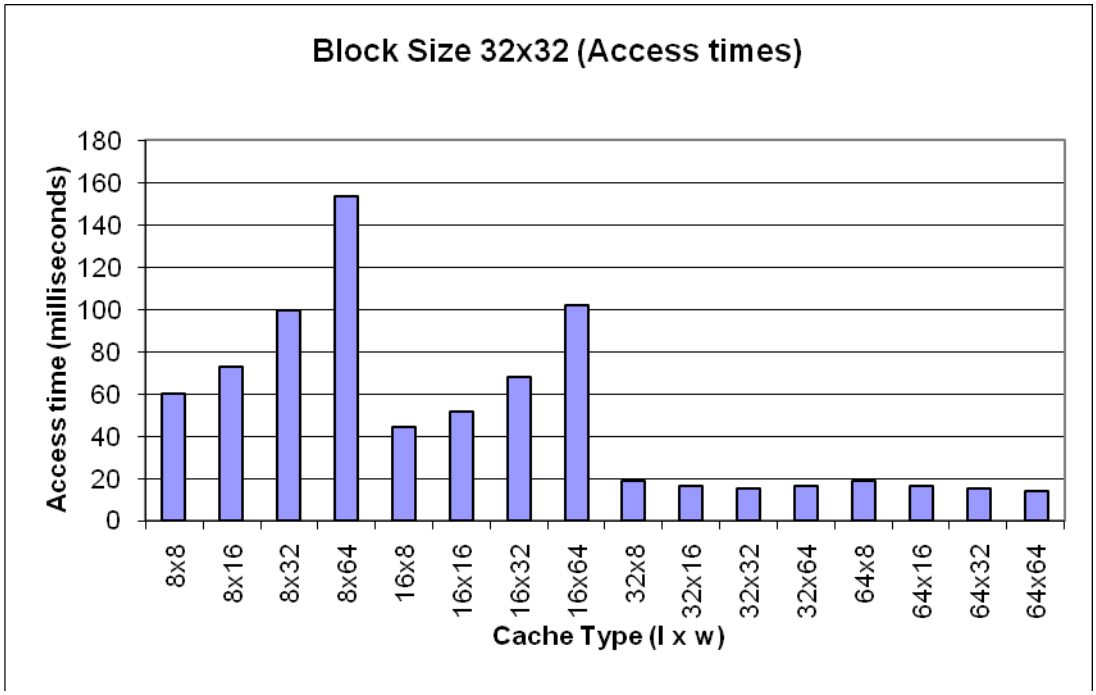


Figure 5.12: Access Times for a block Size 32x32 pixels as a function of a cache size for a cache with l lines and w words per line.

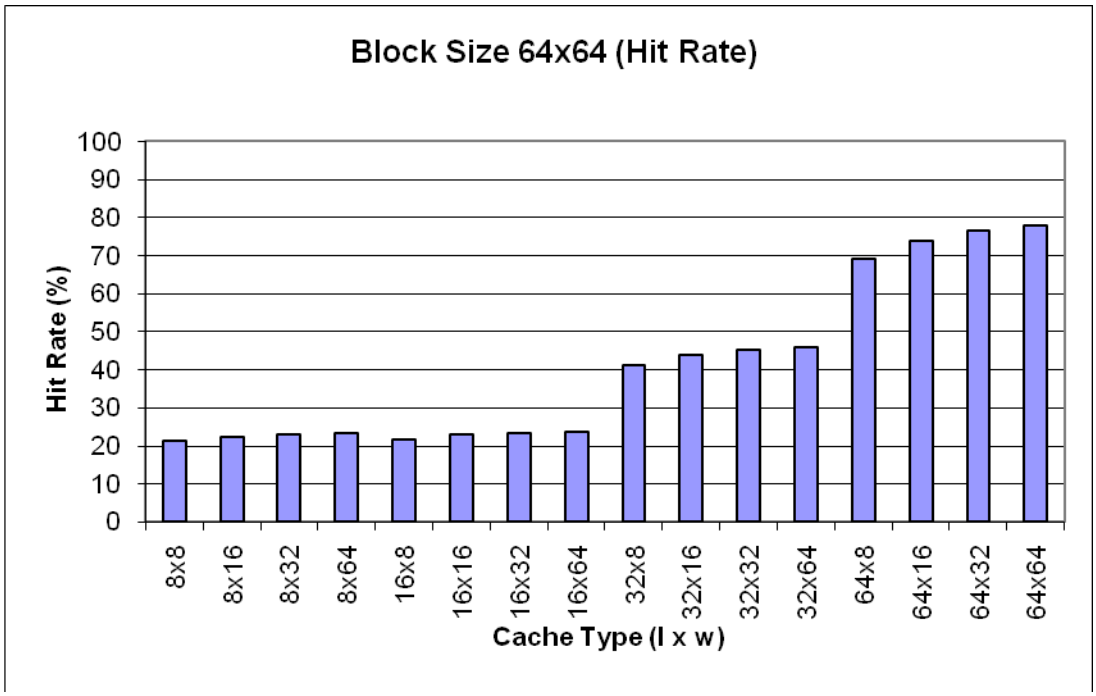


Figure 5.13: Hit rate for an access block size of 64x64 pixels as a function of a cache size for a cache with l lines and w words per line.

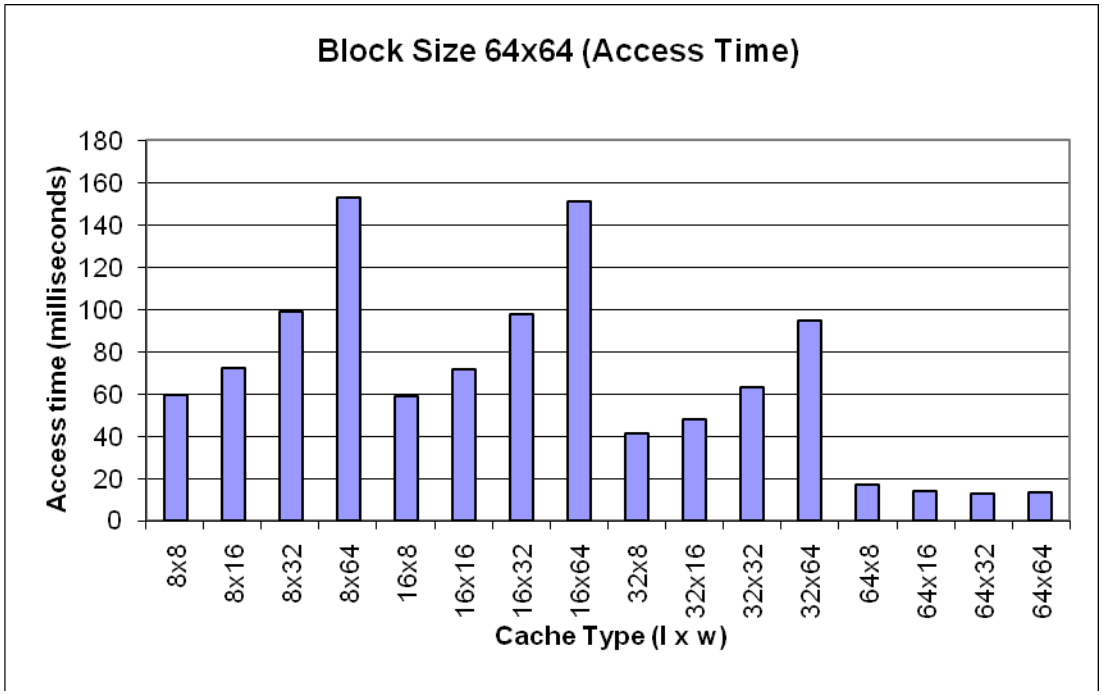


Figure 5.14: Access Times for a block Size 64x64 pixels as a function of a cache size for a cache with l lines and w words per line.

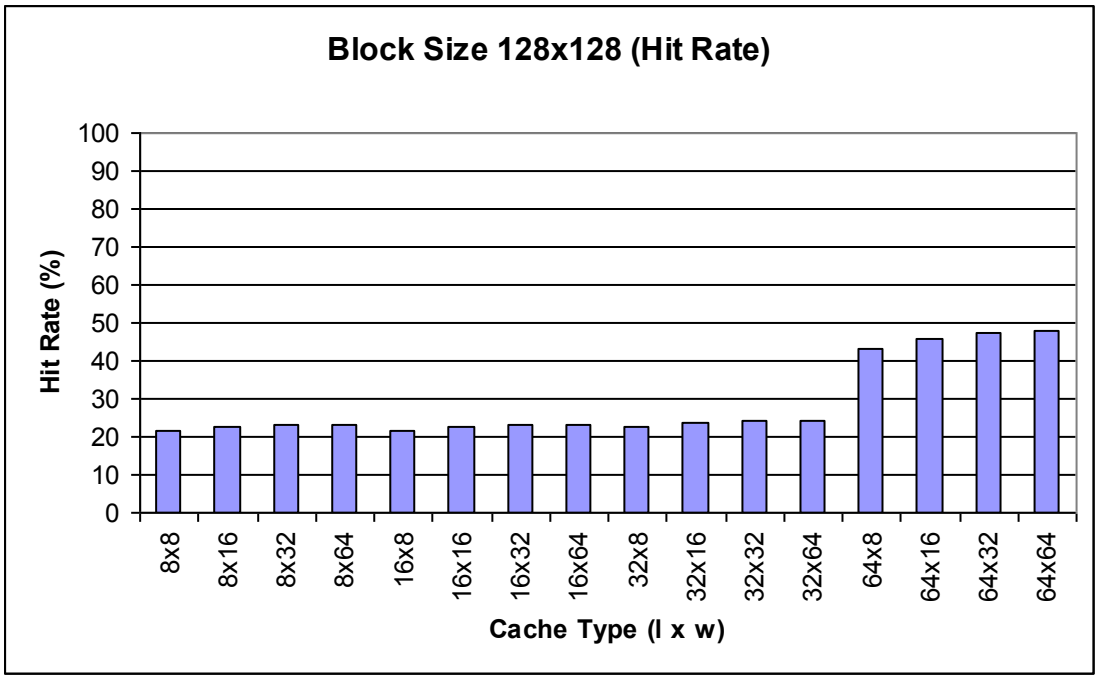


Figure 5.15: Hit rate for an access block size of 128x128 pixels as a function of a cache size for a cache with l lines and w words per line.

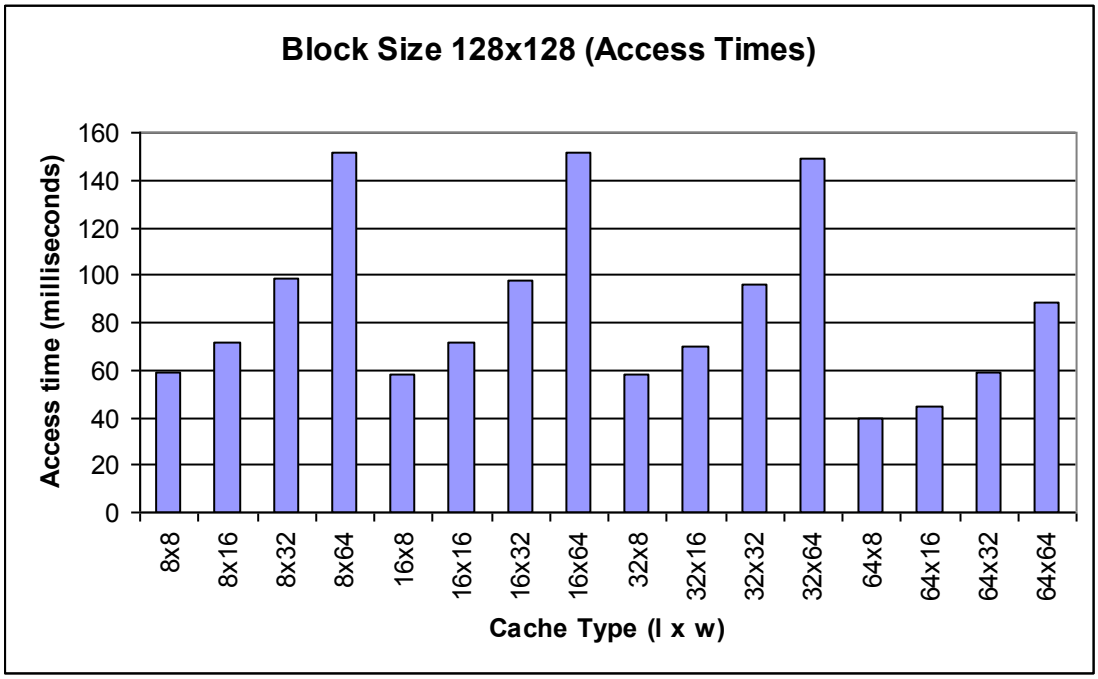


Figure 5.16: Access Times for a block Size 128x128 pixels as a function of a cache size for a cache with l lines and w words per line.

The fastest access time is in the range of 13 milliseconds for a LUT with function 45 degrees. This is 2x times faster than the existing system which does not have a cache and just accesses the SDRAM directly. The cache size for this speed is only 64x32, meaning 64x32x32 bits. This corresponds to 64 Kb of block RAM usage, which is small.

The design with only the SDRAM and no cache performed badly as was seen in Section 3. Consider the fastest cache 64x32 with block size 64x64 locations. This cache design should be able to perform well with all the angles. The following simulation takes this particular cache and blocks size and tests it against the various LUT functions ranging from 0 degrees to 45 degrees.

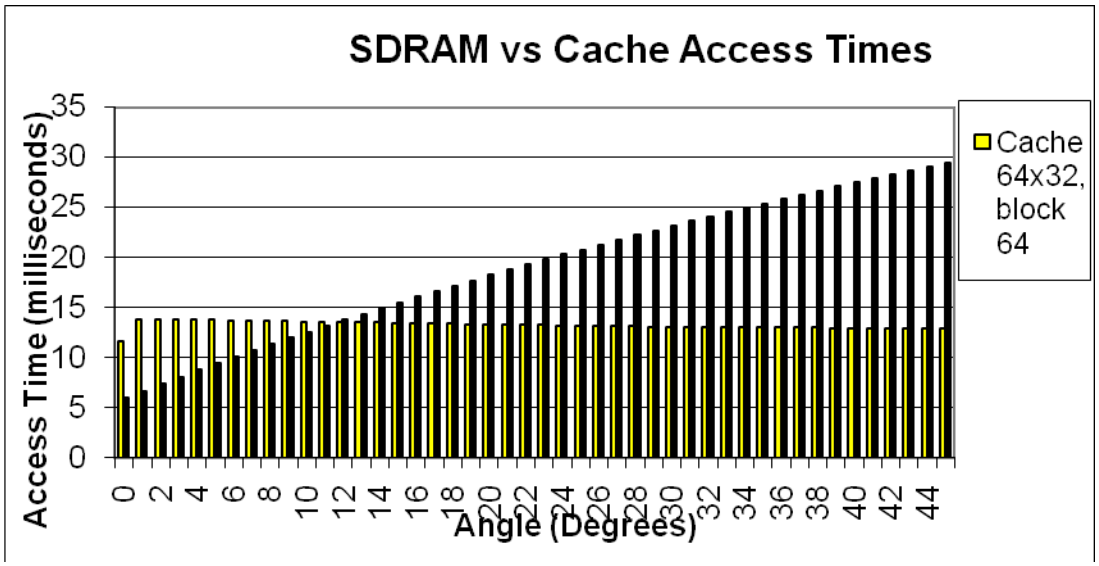


Figure 5.17: SDRAM vs Cache with Blocks

Figure 5.17 shows how the access times vary with change in LUT function angle for the cache under test. The fastest time is for the identity transformation, but the average access times for most of the LUT functions is around 13-14 milliseconds. No matter how much the value of the angle changes, the access time varies up and down only by a very small margin within 1 millisecond. All LUT functions seem to give an access time near a particular value (in this case around 11-14 milliseconds) for a particular cache size and block size. Accessing the LUT according to the blocks makes every angle of rotation

similar in terms of the access time. Table A.5 in the appendix shows all the simulation results for the cache that was the fastest when simulating for a 45 degree LUT function.

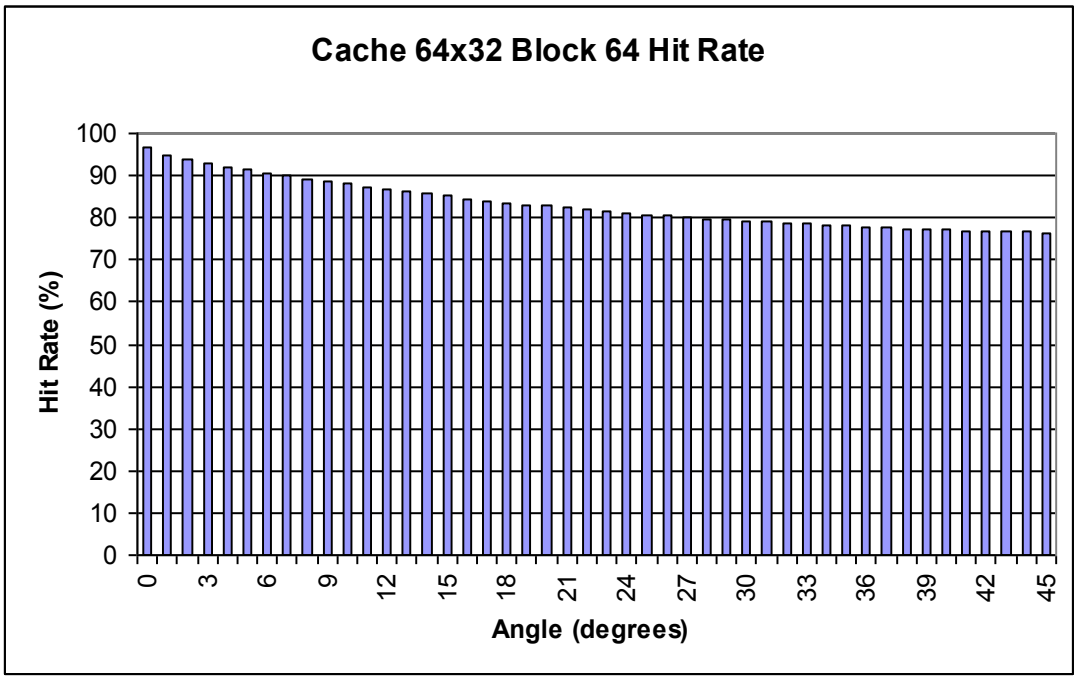


Figure 5.18: Cache 64x32, Block 64 Hit Rate

Figure 5.18 shows the hit rate of the cache (64x32, block 64) for the various LUT functions. The plot is in accordance with the theory that as the LUT function becomes more complicated, the hit rate decreases. Thanks to the memory blocks and the knowledge of the input frame access pattern, the hit rate does not decrease by a large margin. Rather, it decreases a little bit for the first 10 degrees and then smoothes out.

Figure 5.17 shows the comparison between the access times of the design with only the SDRAM and the design that has the cache along with memory blocks. For comparison purposes, the cache 64x32 with block size 64 is chosen and compared against the SDRAM. It is observed that the cache access times are slightly slower than the SDRAM access times for 0 degrees through 11 degrees. This is true as an identity transformation essentially needs no cache. As the angle starts increasing the SDRAM access times increase significantly.

Clearly, the cache design improves access times substantially on average for a rotated input image when compared with direct SDRAM access. Section 9 analyzes the benefits of cache on other LUT functions.

5.4) SDRAM with Blocks

With the introduction of the blocks and the cache module, access times of the system improved by a large margin. One of the design features of the APR's cache is that it divides the memory space into blocks and then accesses these blocks in the LUT so that reading values from the input frame is more sequential. We performed several simulations to see if accessing SDRAM in a blocked fashion without the cache performs comparably to a system with a cache.

The simulation was conducted for a 32x32 block size and for four different angles namely 0, 5, 35 and 45 degrees of rotation. All the simulating conditions are exactly same as in Section 3 except that the LUT is accessed according to the order of the blocks.

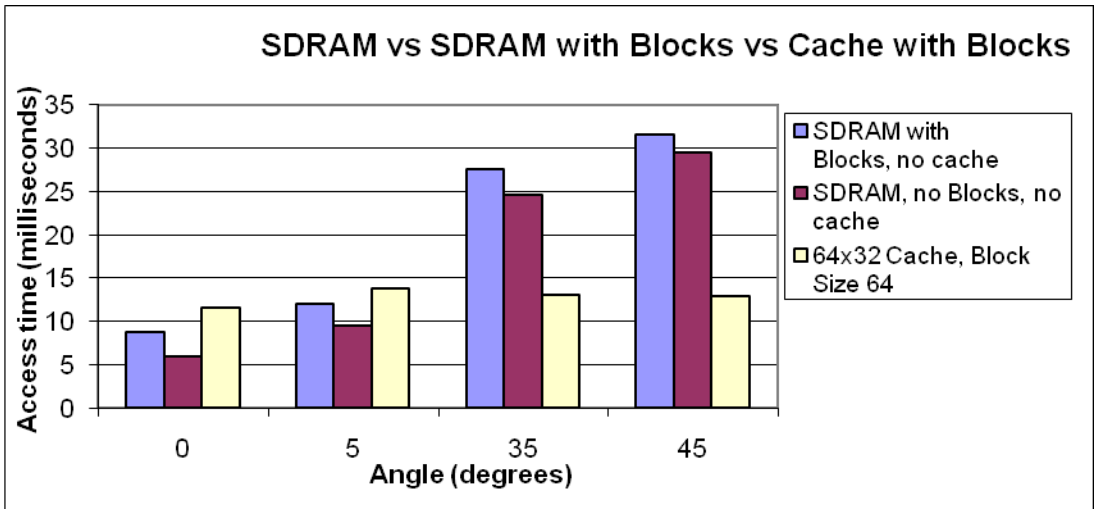


Figure 5.19: SDRAM vs SDRAM with Blocks vs Cache with Blocks

From Figure 5.19, it is observed that the access times are at least 5 ms slower than the access times with just the SDRAM. Even if the results are compared to the cache using

block size 64x64 and angle of rotation 45 degrees, the blocked SDRAM access times are slower.

Block access relies on the fact that the input frame access is faster for accesses with high spatial locality in order to gain an overall speed up. Without a cache, the input frame will take a large number of cycles even within a block. When the LUT values are read in blocks, there is an attempt to change to a different row every time the end of each row in a block is reached. For example, if the LUT is read in blocks of 32x32 memory locations then once the 32 locations of the first line are read, a row change is initiated. The reason why the average frame access time with blocks, but no cache is slightly higher than the model with just the SDRAM is that changing rows introduces extra cycles. If a cache is used for the input frame then the input frame access time is greatly reduced. This leads to an overall speed up.

Chapter 6: Set Associative Caches

As we have seen, a direct mapped cache with blocked memory access patterns can greatly improve the average access for the Pixel Anywhere Router. In microprocessors, increased cache associativity often leads to better hit rates and faster access times. Likewise, associativity can improve the average access time for the Pixel Anywhere Router.

6.1) Set Associative Caches

In a direct mapped cache, each memory address maps to a single cache block. If there are existing values in that block, it is immediately replaced by the new block. In a set associative cache, the cache is broken down into sets and each set has a few blocks. A particular memory address is mapped to a particular set in the cache, but within that set the value could be in any block. If each set has two blocks, then it is a 2-way set associative cache. If a memory address can map to any block in the cache, it is fully associative cache. A direct mapped cache is basically a 1-way set associative cache [5].

Since there is a choice of which block to replace in set associative caches, some method must be used to decide which block to replace. Two commonly used techniques are LRU and FIFO. In LRU, the recently used values are kept while those that are not recently used are replaced. In FIFO, the first block to enter the cache is replaced. In this section, the simulation uses LRU while dealing with set associative caches.

6.2) Caching Function

The caching function is the same as that shown in Section 4, except that the number of index bits decreases as the associativity increases, and correspondingly, the number of tag bits increase. For the same cache size, a 2-way set associative cache has 1 bit fewer in its index field and 1 bit more in its tag field compared with a direct mapped cache. A 4 way set associative cache has 2 bits fewer in its index field, and so on.

For comparing the results with a direct mapped cache, the number of cycles during a cache hit is assumed to be one for a 2-way set associative cache. In practice, the need to

test all the tags in a set may cause a set associative cache to require more clock cycles per hit.

6.3) Simulation Results

The following simulation was conducted for a 2-way set associative cache with LRU as its replacement policy.

For comparison, a block size with 32x32 locations is chosen and size is varied for both direct mapped and 2-way set associative caches in the Figures 6.1-6.2. First the hit rate and then the access times are compared.

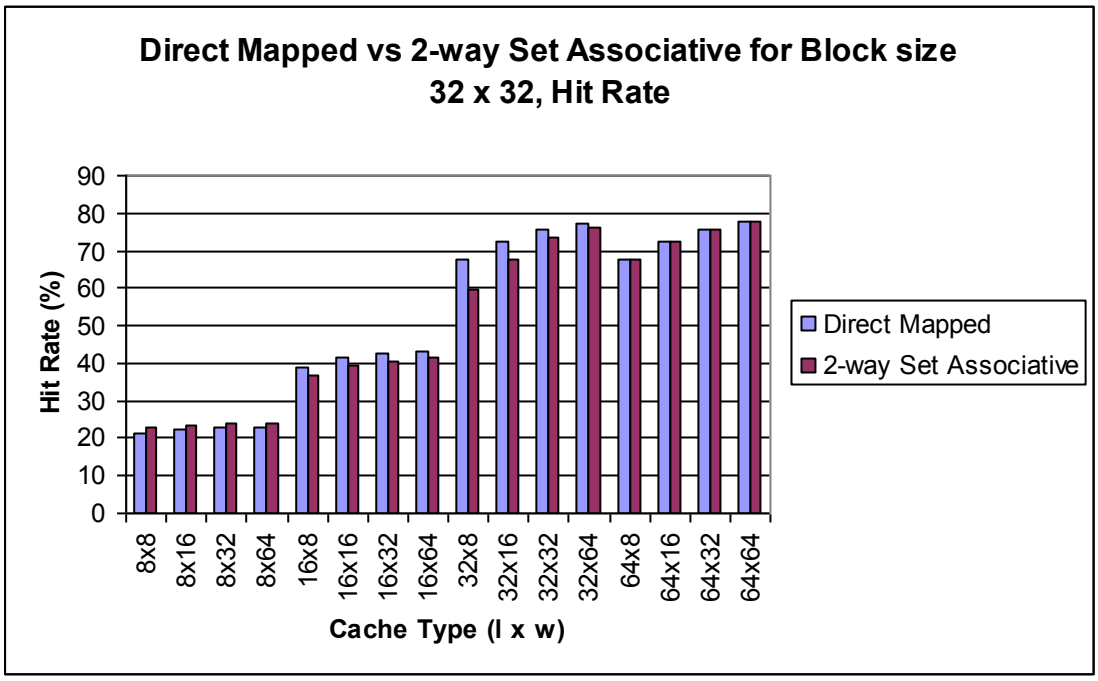


Figure 6.1: Hit Rate Comparison Direct vs Set Associative

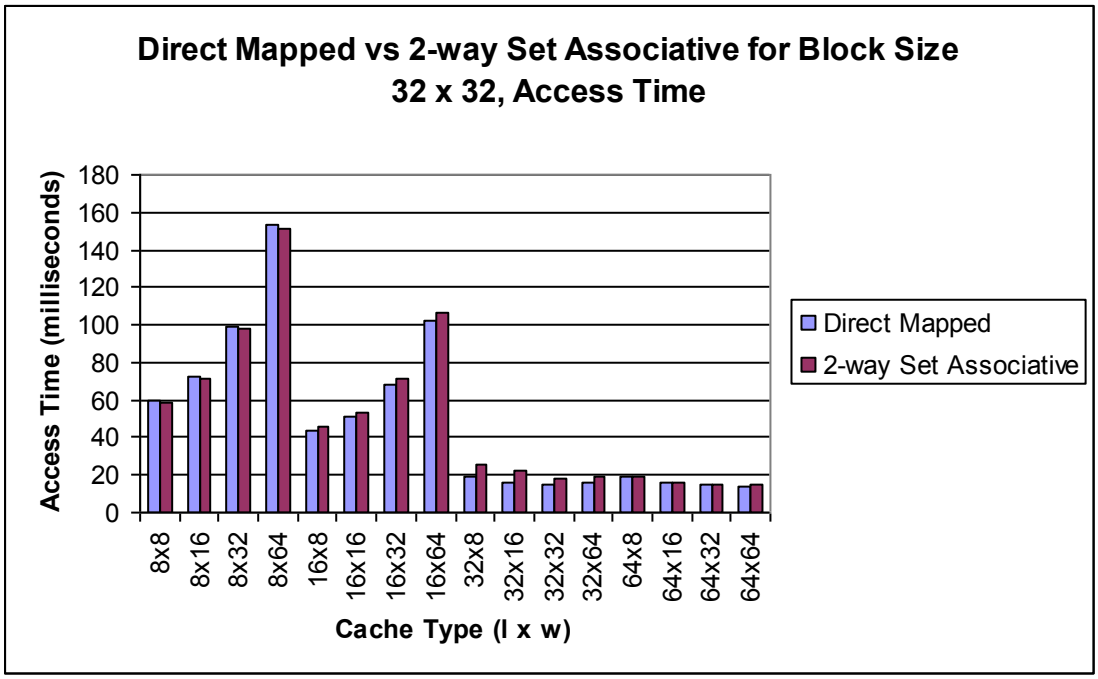


Figure 6.2: Access time Comparison Direct vs Set Associative

The consistent fact in the Figures 6.1 and 6.2 is that if hit rate is higher, the access time is lower for a particular cache size. There is not a significant difference in the access times between the direct mapped and the 2-way set associative cache. For some cache sizes, the direct mapped is better while for some cache sizes the set associative is slightly better.

The 2-way set associative cache is better in cases where the number of cache lines is very small. For example, with eight cache lines, the 2-way set associative cache is slightly better than the direct mapped cache. For a larger number of cache lines like 32 or 64, the direct mapped is better. The 2-way set associative cache is a good choice if the space in the block RAM present can store only a small number of cache lines.

Set associative caches perform better in many cases when compared to a direct mapped cache. As an example, many personal computers like the AMD Athlon [8] have a set associative cache between their main memory and processor. In the APR, there is no great difference in access time between the direct mapped and 2-way set associative cache, because the access pattern of the input frame is known before the application starts

to execute. With this pattern the block sequencing is determined and the LUT is accessed according to this pattern. There is no significant difference in access times between the two types of caches, because the values accessed in the input frame can be predicted before run time. Due to rounding errors in the interpolation, the same pixel might be used twice or thrice but usually not more than that. On the other hand, the likelihood of accessing the neighboring pixels is very high. That is, the application depends more on spatial locality than temporal locality. The access pattern of the input frame, computed offline, ensures that there are no conflicts in the cache and hence the access times of a direct mapped cache and 2-way set associative cache are not very different. For a 45 degree angle of rotation, every pixel accessed is a new line in the input frame and invariably existing cache lines have to be replaced. Other than small changes in the access times values, the 2-way set associative cache is not significantly more efficient than the direct mapped cache.

A 2-way set associative cache requires more logic than a direct mapped cache. The additional logic might increase the number of cycles required for a hit. The APR is highly time critical and also is limited by hardware resources. The identity transformation using only the SDRAM takes about 6 milliseconds and even with the best cache design, this time cannot be beaten as every adjacent pixel in the LUT corresponds to the same line in the input frame. The direct mapped cache for a 45 degree rotation is close to this value and can be preferred over the 2-way set associative cache. The fastest access time given a maximum cache size of 4K locations is produced by the direct mapped cache. The fastest 2-way set associative cache is obtained for block size 32x32 and the fastest direct mapped cache is obtained for block size 64x64.

Simulations for 4-way associative caches were not conducted because the application is using an FPGA with limited number of resources, and a 4-way associative cache would demand more hardware logic and complexity.

The minimum time obtained for the fastest direct mapped cache is 11 ms, which occurs for a rotation of zero degrees. The remaining times are close to 13 ms, which is still

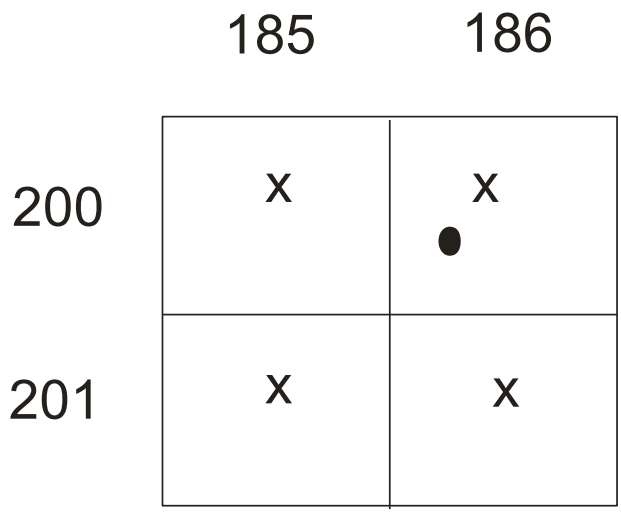
below the minimum of 16.6 ms per frame. The access time without the cache and blocks grew linearly from 6 ms to 30 ms as the angle goes from 0 degrees to 45 degrees. To conclude, the usage of simple direct mapped cache with proper block sequencing is recommended owing to its simple design and effective results.

Chapter 7: Bilinear Interpolation

The current APR implements reverse mapping by using the nearest neighbor method. Bilinear interpolation yields more visually pleasing graphics than nearest neighbor method, but requires more computational power. This section presents the concept of bilinear interpolation, and how the hardware is affected. It also compares the performance of an Anywhere Pixel Router with only SDRAM and the one with a cache, while dealing with bilinear interpolation.

During the process of magnification in images, usually nearest neighboring pixel values are used to determine the value of the current pixel. This technique can result in blocky appearances. Bilinear interpolation reduces this effect by considering the values of all the neighboring pixels when determining an output pixel's value [9].

In the future, bilinear interpolation will be performed by the hardware for the Anywhere Pixel Router. In that case, all the LUT values will be fixed point and not rounded off to the nearest integer value.



Consider Row Value, Column value as (200.1,185.8)

Figure 7.1: Example for Bilinear

As an example, consider Figure 7.1. The LUT will contain a value like (200.1, 185.8). The black spot in the figure shows the approximate location of that pixel. Now instead of just rounding off the value to (200,186), the hardware will now take the values of the four neighboring pixels shown by 'x' in the figure to determine the value of that pixel.

7.1) Advantages of Bilinear Interpolation

The current hardware implements the reverse mapping of pixels using nearest neighbor method. The output is distorted such that the fonts in the screen are very blurred as shown in Figure 7.2.

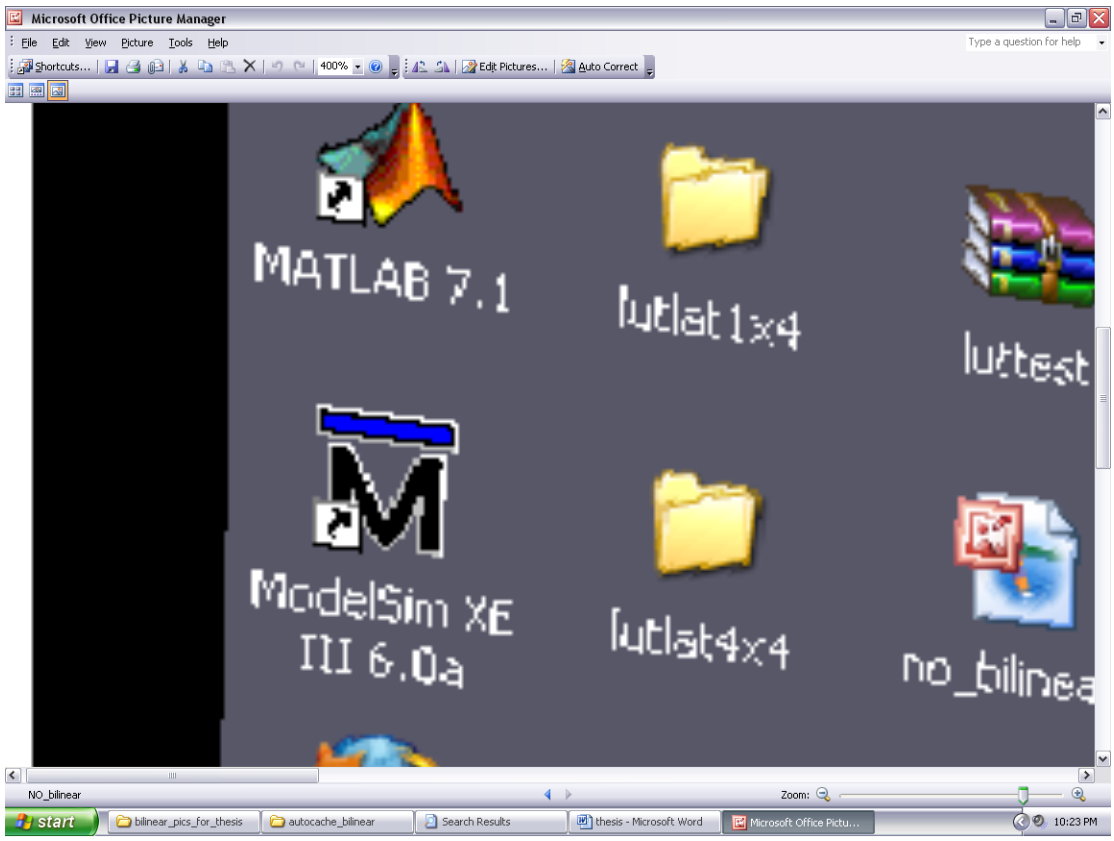


Figure 7.2: Nearest Neighbor Method

The output with bilinear interpolation is much better, as shown in Figure 7.3.

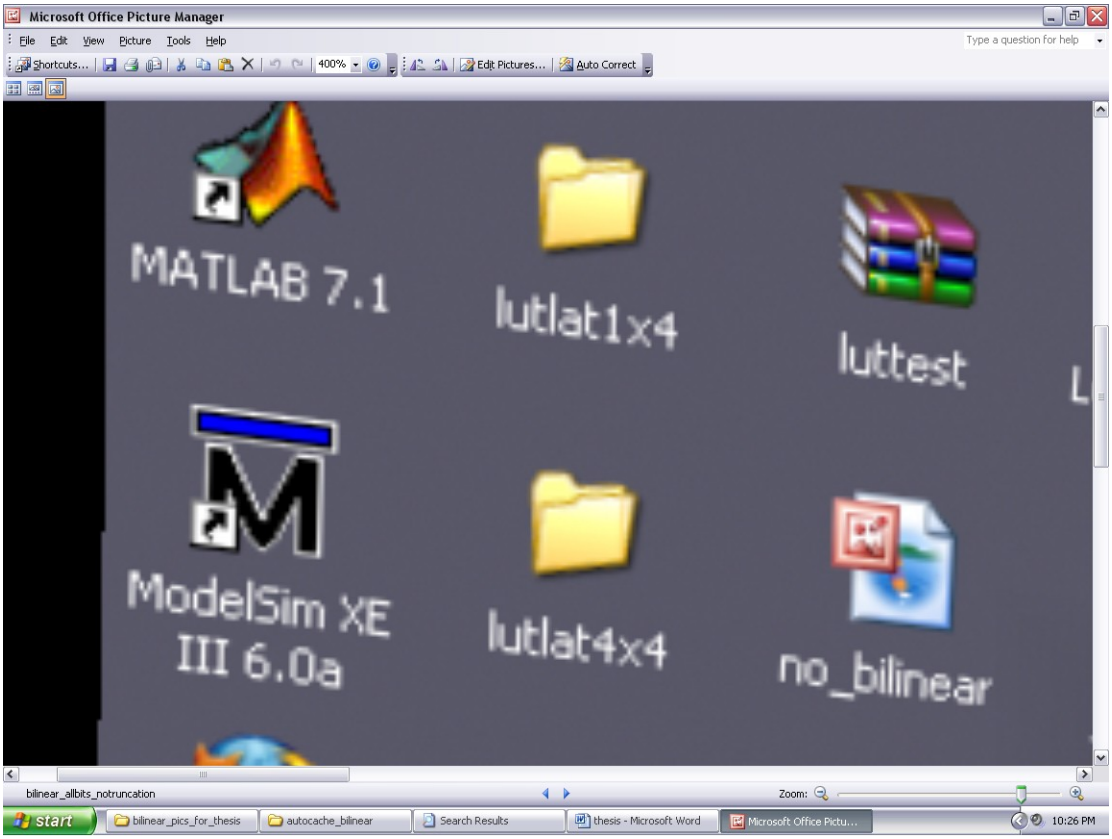


Figure 7.3: Bilinear Interpolation Method

The current hardware design supports only integer values for input frame row and column. But in bilinear interpolation method, the row and column values are fractional. It must be decided how many bits after the radix point to allocate for the row and column values. Figures 7.4, 7.5, and 7.6 show the output images for bilinear interpolation with 1 bit, 2 bits, and 3 bits after the radix point.

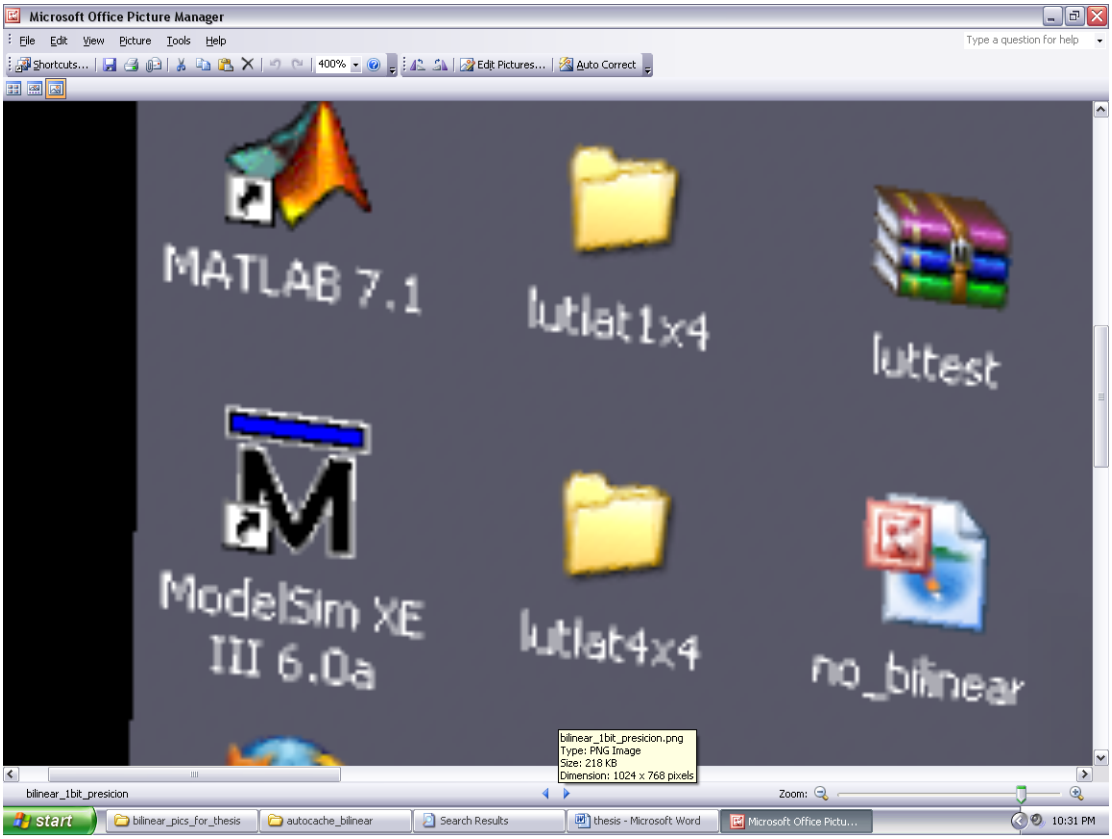


Figure 7.4: Bilinear Interpolation with 1 bit after radix point

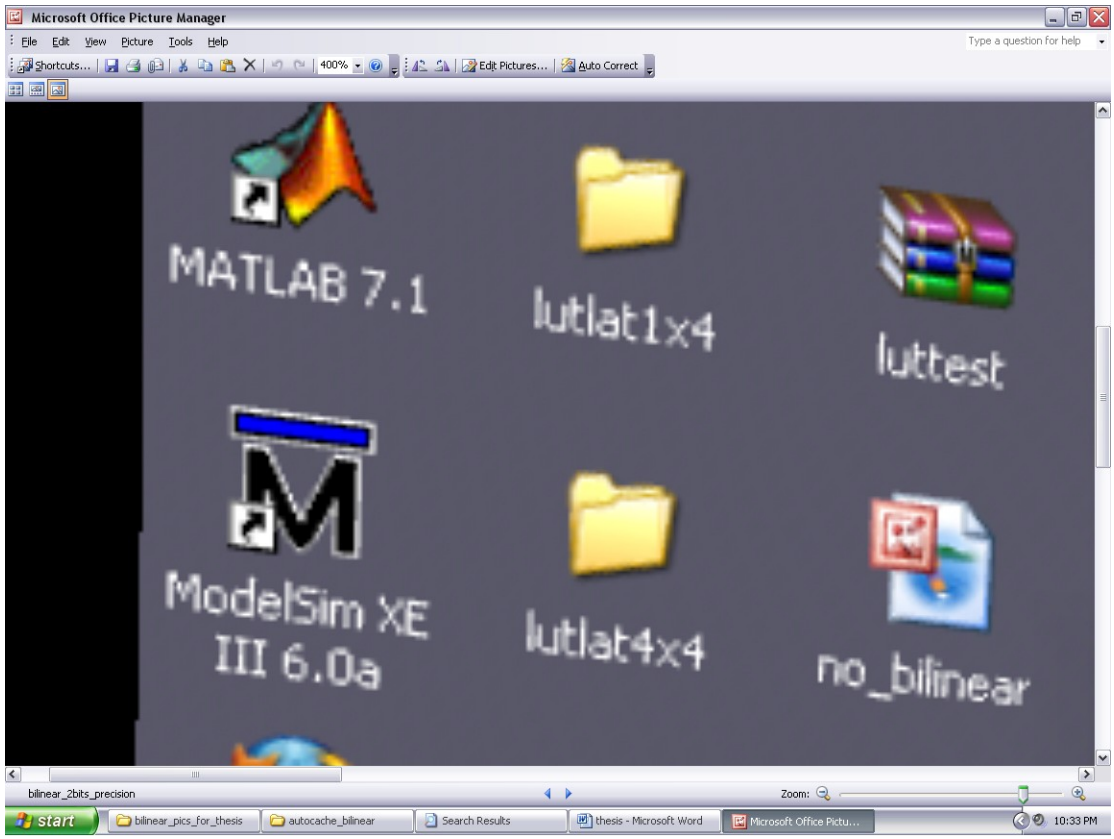


Figure 7.5: Bilinear Interpolation with 2 bits after radix point

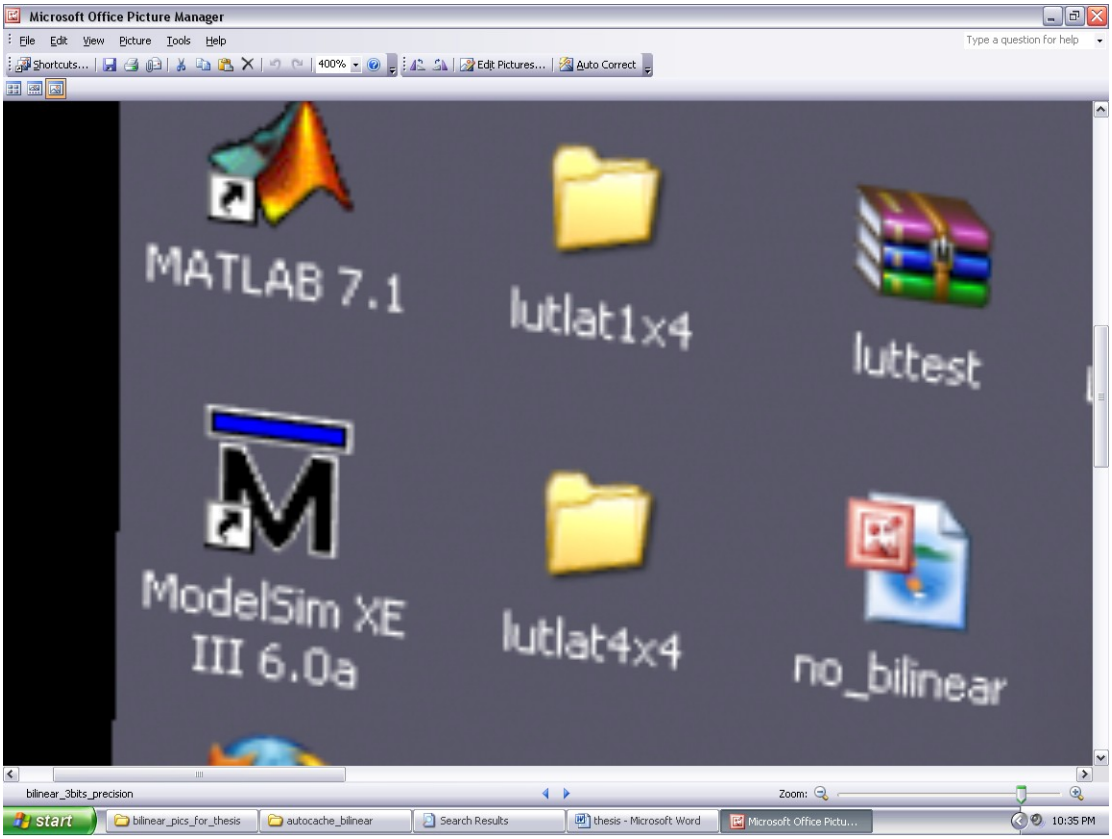


Figure 7.6: Bilinear Interpolation with 3 bits after radix point

The bilinear interpolation was also done in software for 4 bits after the radix point, but there was no significant change in the visual appearance of the output when compared to the design that used 3 bits after the radix point. The idea is to have fewer bits and achieve good image quality, as more bits take up more memory space. It was decided that the system will have the 3 bits after the radix point, meaning that the row and column value will each need 3 extra bits for their representation.

Every value in the LUT now accesses two distinct rows in the image. These distinct rows in the image map to two distinct rows in the input frame memory unless the SDRAM has an exceptionally large row length, which is not practical. So every LUT access now would correspond to a definite change in row, adding more cycles to the process. With a cache that pre-fetches data, the number of cycles should be reduced in case of bilinear interpolation. The simulation in the following section estimates how performance

changes when LUT values are kept in fised point format. The results with and without the direct mapped cache are compared.

7.2) SDRAM simulation with Bilinear Interpolation

The following simulation deals with accessing input frame in the SDRAM without using any cache for the bilinear interpolation. All hardware and memory modules used are same as in earlier sections. During bilinear interpolation, if the row in the input frame is the same as that of the row in the LUT value, then the time taken is $t_{cas} + t_{ras} + t_{cas}$, which is 11 cycles. It is only one t_{cas} for accessing two values of the same row, because the memory module used is DDR and two adjacent values of the same row are used in bilinear interpolation. If the row in the input frame is not the same as that of the LUT value, then it is $t_{ras} + t_{cas} + t_{ras} + t_{cas}$, which is 18 cycles. All the above mentioned values are for the particular hardware and memory module used in the APR and might vary if different modules are used.

Figure 7.8 shows the various access times in case bilinear interpolation is performed by the hardware for various LUT function angles. Of course, the LUT would not be just some angle of rotation, the input pattern might have in fact two or three distinct angles. But this set of simulations gives an idea of how slow the system gets in case of bilinear interpolation.

It can be observed that in contrary to expectation the 45 degree rotation function takes slightly less time than the identity transformation.

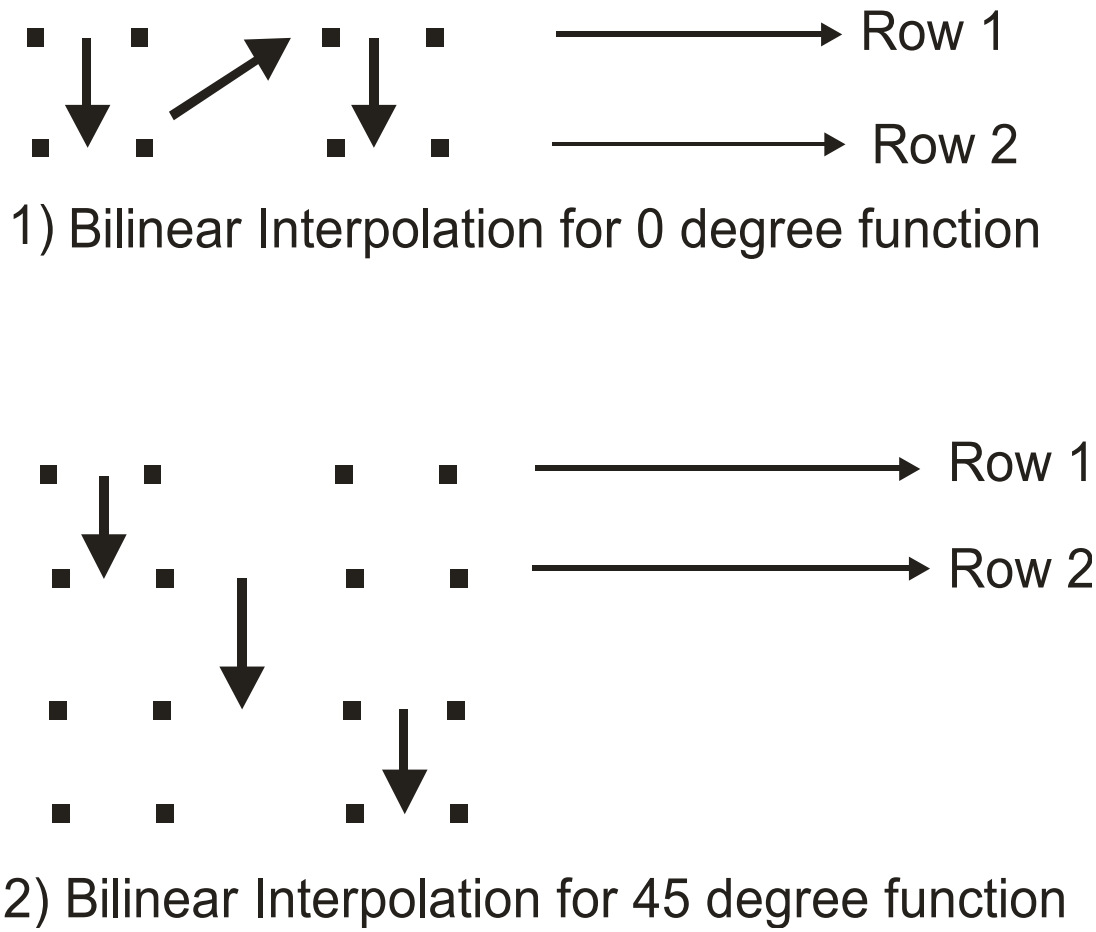


Figure 7.7: Explanation of Bilinear

Figure 7.7 shows access patterns for 0 degree and 45 degree rotations. For some function, like the identity transformation, bilinear interpolation takes about at least 3 memory row changes in the input frame for adjacent LUT value access. On the other hand, a 45 degree function with bilinear interpolation takes only at the most 2 memory row changes. That explains why the 45 degree is slightly faster than the identity transformation.

For the given hardware space, the direct mapped cache producing the fastest access time was 64x32 cache with block size being 64x64 locations. Simulations are conducted to see how the system with that cache performs in case of bilinear interpolation.

The access times for different angles are plotted to compare the access times of the RAM and the design with the 64x32 cache for bilinear interpolation in Figure 7.8. Block access

knowing the input access pattern makes sure that the access time for any LUT function for a given cache and block size is nearly constant. The average access time for any LUT function would be somewhere between 35-40 milliseconds. The cache access times are much faster compared to SDRAM access times.

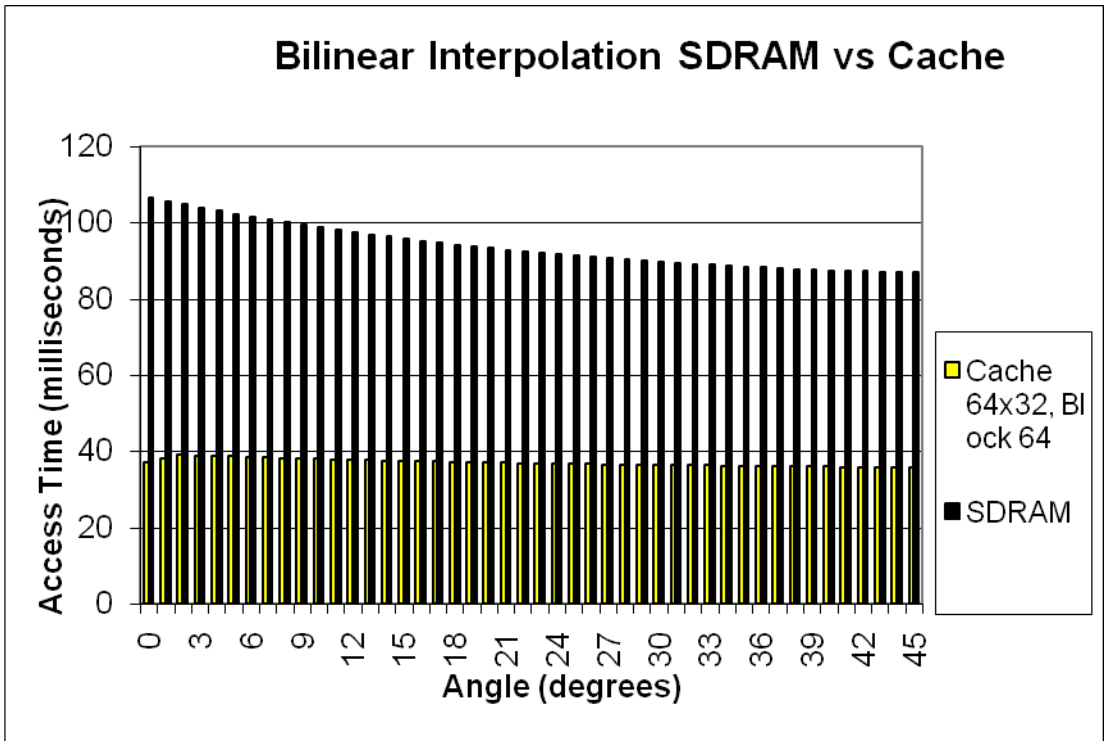


Figure 7.8: Plot SDRAM vs Cache (Bilinear)

For bilinear interpolation by hardware, even the cache with blocks does not meet the requirement of 16.6 milliseconds for a frame. The average is around 35-40 ms which is at least 20 ms greater than the target time. In the simulation, one output pixel location depends on four memory locations, so the cache is accessed 4 times to check for a match, meaning 4 cycles if all the cache accesses are hits. In most cases, two adjacent values in bilinear interpolation would correspond to two adjacent locations in a cache line. The only exceptions are at the boundaries of offset bits. For example, a 20-bit address of row 500 and column 768 and another address of row 500 and column 767 would have different tag bits and hence would not map to the same cache line. In cases where two pixels are adjacent in the cache line, the two values can be fetched from the same cache line simultaneously in the same clock cycle. The structure of the FPGA allows parallel

computations in the same clock cycle. If this done, the access times will further reduce. Table A.8 in Appendix A shows the various access times for bilinear interpolation when the fastest cache is used and Table A.7 in Appendix A shows access times when only SDRAM is used.

One more possibility in bilinear interpolation is there could a few lines of buffer for memory access instead of a cache. For example, there could be a two row buffer for memory access. If a new memory row has to replace an existing row in the buffer, LRU could be used. Even in that case, the LUT memory will have to be accessed in blocks in order to have a good hit rate. For large angles of rotation, the two row buffer system would not be as efficient as the cache because the cache can store more lines than the buffer. For smaller angles of rotation, the buffer system would be beneficial, as the same two rows in memory might be accessed repeatedly.

Chapter 8: Finding the Input Access Pattern

Block access plays a major role in the overall speed of the design, so it is essential that the order in which blocks are accessed is good enough to attain reasonable speeds. If a proper access pattern is deciphered, then it is easy to order the way the memory blocks are accessed in the LUT thereby leading to the overall increase in speed of the system when using a cache. The access pattern is computed offline and depends on the LUT. The order in which blocks are accessed is different for a 45 degree angle when compared to a 25 degree angle. It is essential to first find out the function in the LUT.

In all the simulations in previous sections, the LUT was generated synthetically by rotating an identity LUT by a known amount. All the LUTs were generated just to test the efficiency of the design. In a real implementation the LUT is created from the results of the calibration. This section describes how to determine the access pattern for which the best ordering is unknown.

8.1) Determining the Access Pattern

The LUT function is a result of the projector system, how the projectors are aligned, and how the projector areas overlap one another. For a multi-projector system, the different LUTs could contain different angles of rotation. It is also possible that one section of a single LUT contains partly one angle of rotation and another section another angle of rotation. An efficient algorithm to compute the nature of LUT is needed so that an appropriate block access pattern can be passed to the hardware.

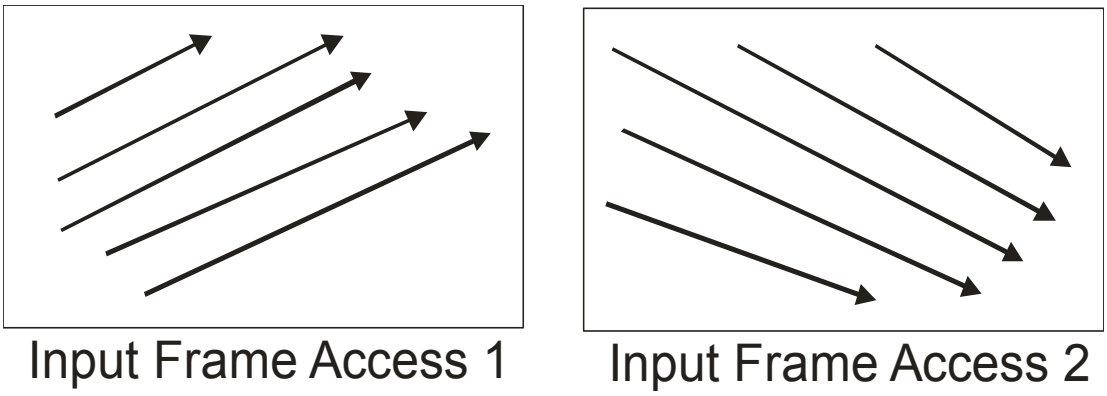


Figure 8.1: Sample Input access

Figure 8.1 shows the access patterns of normal LUTs that are created from calibration results. The projectors could be tilted or kept perpendicular. The LUT's function is unknown and it should be determined by examining the contents of the LUT.

Figure 8.2 shows some possible LUT functions. Figure 8.2(a) is a LUT in which one half of the LUT accesses input frame pixels at 5 degrees while the other half accesses at 10 degrees. The function in figure 8.2(b) is a possible 4 function LUT split as shown. It is evident that the orders in which blocks are accessed in the two cases are totally different. A simple algorithm to decide the order in which the LUT memory blocks are accessed is needed.

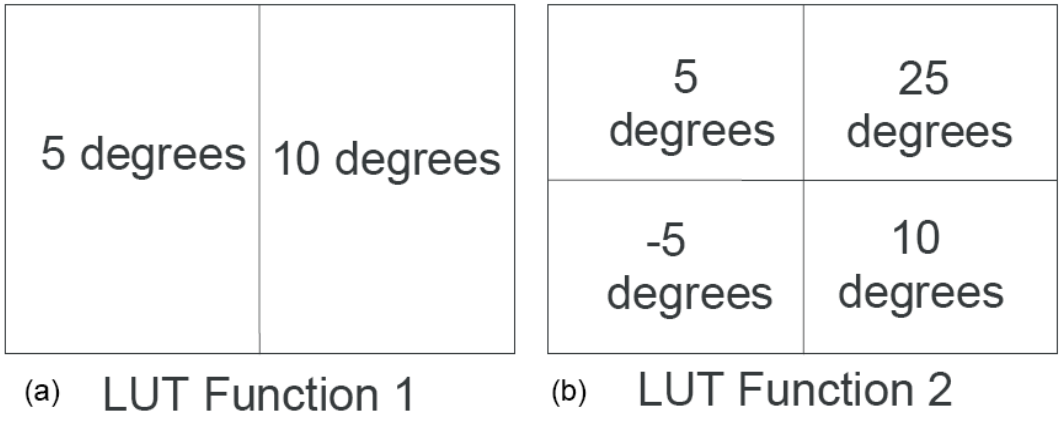


Figure 8.2: Possible LUT functions

8.2) Algorithm

The flow chart in Figure 8.3 shows a method to determine the access pattern of the input frame. This technique was adopted to find the access pattern and then create a block order so that the cache design is effective. The algorithm was tested on some synthetically generated LUTs and also on LUTs obtained from calibration results. The algorithm was effective in creating a block order for the cache design.

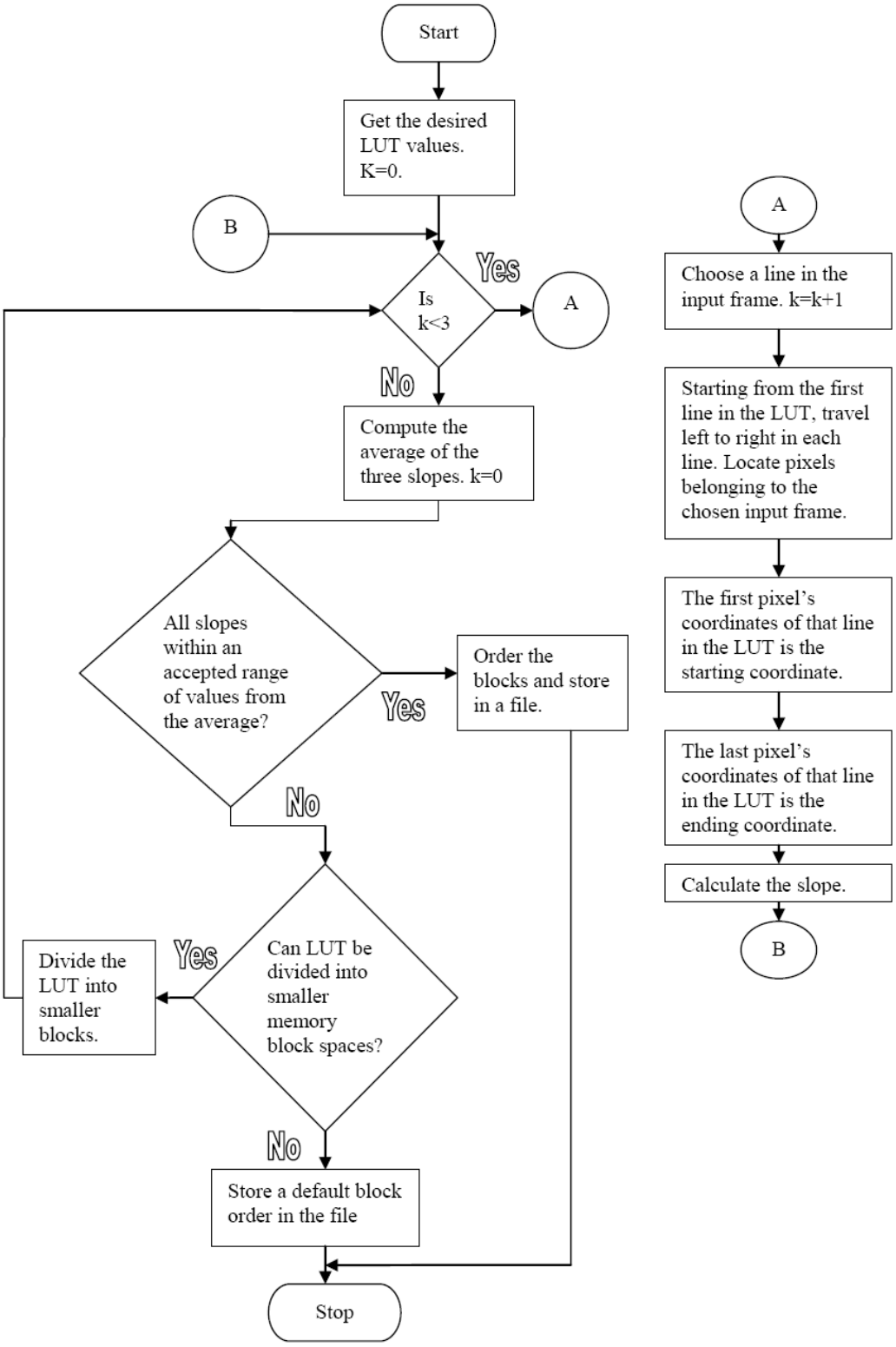


Figure 8.3: Input Access Algorithm

Chapter 9: SDRAM vs Cache Comparison

A few sample LUTs are taken as test inputs and simulations are run to compare the design without cache to the design with the cache and blocks. The design with only SDRAM is a failure in many cases. The cache design with block access pattern overcomes this problem in all cases tested. The previous simulations were all based on simple LUTs. Most of the cache simulations were restricted to a particular LUT function. It is necessary to see how the design with the cache performs with measured LUTs. The block ordering for the following simulations were obtained from the algorithm in Section 8. The fastest cache, 64x32 with block size 64x64 memory locations is taken and its results are compared to the design with only the SDRAM module.

9.1) Test LUTs

Three test LUTs are taken for the comparison. The first is a LUT with a 90 degrees rotation function. Often, it is possible for users to place the projectors perpendicular to the surface and project onto the screen. This results in a 90 degree rotation in the input access. This LUT is labeled as I. The second LUT is from a real calibration result whose pattern is somewhat close to 5 to 6 degrees of rotation. This LUT is labeled II in the simulation. The third LUT is also from a real calibration result whose pattern is slightly more complicated than LUT II. The pattern in which the input frame pixels are accessed is close to angle 10 degrees. This test LUT is labeled III. The calibration in the above cases was done for a 4 projector system, and so 4 LUTs were generated. One of the four LUTs is taken for test purposes in each case.

9.2) Simulation Results

Figure 9.1 shows the direct comparison in performance between the design with only the SDRAM and the cache design with blocks while handling the test LUTs.

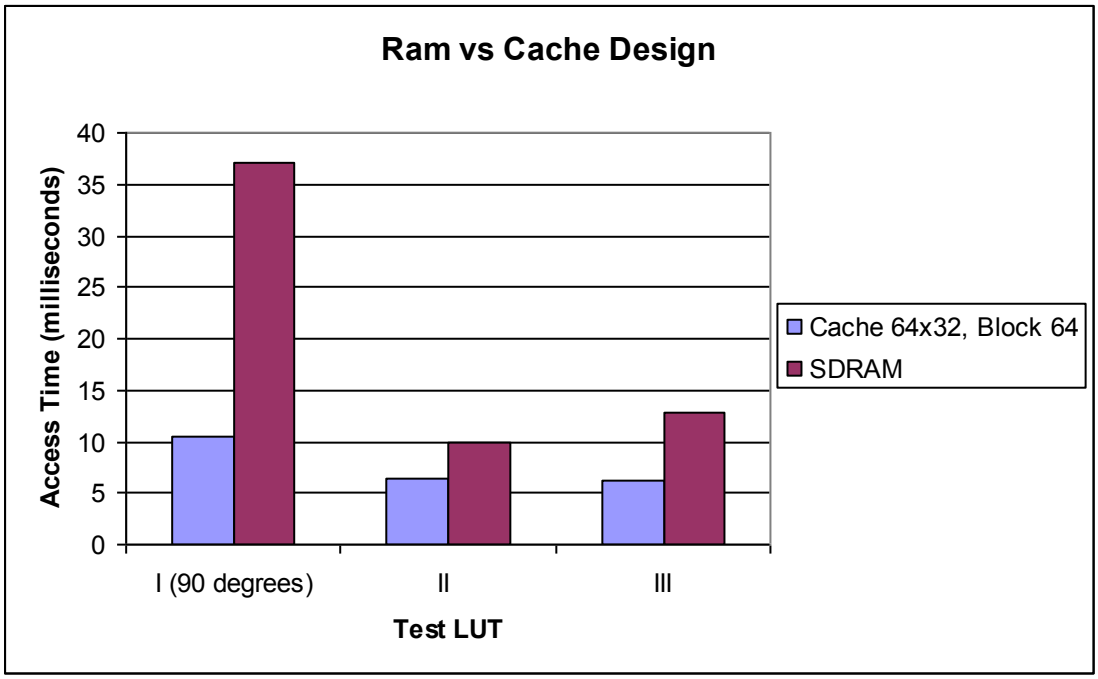


Figure 9.1: SDRAM vs Cache (Overall)

For 90 degrees LUT function the SDRAM is definitely slow as every LUT access corresponds to a change in row in the input frame memory. The LUT II was obtained from calibration when the projectors were subjected to slight tilt. Hence the design with SDRAM is not too slow but the cache is still faster. The LUT III is obtained by exposing the projectors to more inclination than case II and the cache is faster than SDRAM by at least a factor of 40%. There are a number of possible ways in which projectors could be set up by users. Whenever the LUT function involves more rotation, the design with the cache and memory blocks reduces access time. Even for less complicated functions, the cache design is still faster than the SDRAM design. Table A.10 in Appendix A shows details of the simulation results when the design with cache and blocks was made to handle LUT I, II and III. Table A.11 in Appendix A shows the simulation results when the design with only RAM is made to handle the test LUT I, II and III.

9.3) Bilinear Interpolation

One of the future goals of this research work is to implement bilinear interpolation in hardware. The same LUTs I, II and III are taken and tested for bilinear interpolation.

Figure 9.2 shows the head-to-head comparison of cache design and the design with only SDRAM with of bilinear interpolation.

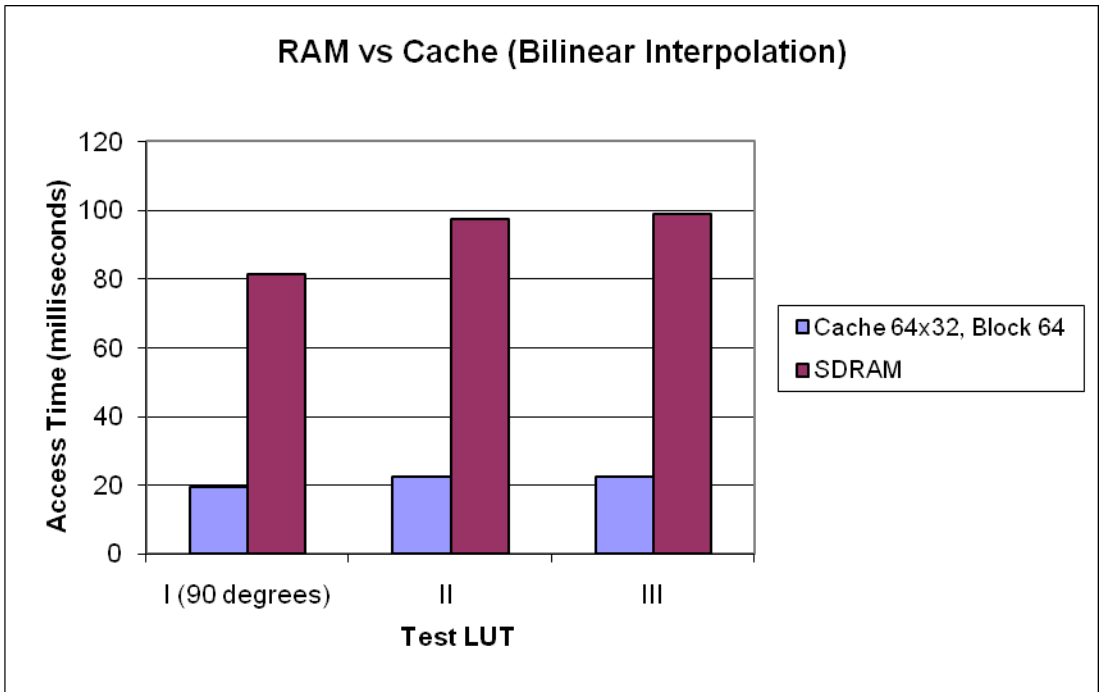


Figure 9.2: SDRAM vs Cache Bilinear

As predicted the cache design with the memory blocks is much faster than the design with SDRAM when bilinear interpolation is done in hardware. The access time with the cache is at least 4 times faster than the access time with the SDRAM. Table A.13 in Appendix A shows the results of the design with cache and blocks when doing bilinear interpolation. Table A.14 in Appendix A shows the simulation results of the design with only RAM when dealing with bilinear interpolation.

Chapter 10: Conclusion

The APR uses a memory controller that directly accesses the SDRAM. That design is not fast enough to process video frames at the desired rate, and under test conditions it was observed that many output frames were skipped when the LUT function had angles of rotation greater than 14 degrees. A new robust design was required that can increase the frame rate for all LUT functions, without requiring additional hardware. A blocked access pattern calculated offline from the LUT, is key to efficient cache access. The new cache algorithm improves performance for a wide variety of rotations, skewing angles, and other forms of distortion.

The most important performance criterion for the APR is that it must process 60 input frames per second. The design with cache and memory blocks processed the input frame within the desired the frame time of 16.6 ms. The cache is faster than the SDRAM by 20% to 30% on an average for synthetic LUTs and 2 times faster for LUTs resulting from actual projector calibration. Moreover, the cache occupies about 7% of the block RAM memory space. Thus an effective cache was designed for the APR.

Bilinear interpolation is not being implemented currently, but it is a definite feature of the APR in the near future Even though the cache is faster than the SDRAM design for bilinear interpolation, the average access times produced by the simulation are above the desired time allocated for every frame. Right now, it is assumed that the cache is invoked 4 times for every LUT value access, which can be reduced to fewer cycles considering the adjacent access of memory locations during bilinear interpolation. If the number of cycles is reduced, then the results will be significantly better.

Appendix A: Simulation Results

Table A.1: SDRAM Simulation

Angle	Pixels	Cycles	Same Row (%)	Change in Row (%)	Ignore (%)	Time(millisecs)
0	786432	797945	99.902473	0.097529	0	5.999587
1	786432	894090	97.317505	1.844025	0.83847	6.722481
2	786432	989094	94.726814	3.569794	1.70339	7.436797
3	786432	1078820	92.260361	5.199687	2.539953	8.111428
4	786432	1168287	89.826584	6.824875	3.348541	8.784113
5	786432	1257222	87.429298	8.440399	4.1303	9.452797
6	786432	1344379	85.090385	10.023625	4.885992	10.108112
7	786432	1429905	82.806648	11.577225	5.616125	10.751165
8	786432	1514367	80.567039	13.111496	6.321462	11.386218
9	786432	1597506	78.374992	14.621735	7.003275	12.011323
10	786432	1679259	76.232277	16.106796	7.660929	12.626007
11	786432	1759710	74.134575	17.568207	8.297221	13.230902
12	786432	1838894	72.083282	19.006601	8.910115	13.826271
13	786432	1916930	70.074463	20.424143	9.501393	14.413008
14	786432	1993748	68.110786	21.819559	10.06966	14.990587
15	786432	2069467	66.182961	23.195013	10.62203	15.559902
16	786432	2143996	64.299011	24.548849	11.15214	16.12027
17	786432	2217552	62.45219	25.88501	11.6628	16.673323
18	786432	2290065	60.643387	27.202225	12.15439	17.218534
19	786432	2361451	58.873749	28.498968	12.62728	17.75527
20	786432	2431948	57.138317	29.779562	13.08212	18.285323
21	786432	2501451	55.43874	31.042099	13.51916	18.807903
22	786432	2569988	53.773754	32.28709	13.93916	19.323219
23	786432	2637552	52.144367	33.514404	14.34123	19.831218
24	786432	2704360	50.544483	34.727985	14.72753	20.333534
25	786432	2770195	48.978043	35.923893	15.09806	20.828534
26	786432	2835176	47.444279	37.10429	15.45143	21.317113
27	786432	2899317	45.940525	38.269424	15.79005	21.799376
28	786432	2962709	44.465893	39.420956	16.11315	22.276007
29	786432	3025177	43.0233	40.555698	16.421	22.745691
30	786432	3086959	41.60741	41.677982	16.71461	23.210218
31	786432	3147908	40.220898	42.785137	16.99397	23.668481

32	786432	3208003	38.862484	43.876774	17.26074	24.120323
33	786432	3267426	37.533569	44.956207	17.51022	24.567112
34	786432	3326114	36.229706	46.022289	17.74801	25.008377
35	786432	3383997	34.95369	47.073746	17.97257	25.443586
36	786432	3441411	33.699036	48.116684	18.18428	25.87527
37	786432	3497516	32.480621	49.135845	18.38354	26.297113
38	786432	3553474	31.277466	50.152332	18.5702	26.717849
39	786432	3608711	30.099487	51.155727	18.74479	27.133165
40	786432	3663094	28.949102	52.143604	18.90729	27.54206
41	786432	3716826	27.821985	53.119659	19.05836	27.946061
42	786432	3769830	26.719412	54.082489	19.1981	28.344586
43	786432	3822155	25.639853	55.032986	19.32716	28.738007
44	786432	3873766	24.584326	55.970509	19.44517	29.126059
45	786432	3924740	23.563513	56.896465	19.54002	29.509323

Table A.1 shows the simulation results while running the anywhere pixel router with just the DDRRAM module. The table has details about same row accesses, change in row accesses, and the number of IGNORE pixels present. This simulation does not include results for any blocks or cache designs.

Table A.2: Cache Access times (45 degrees)

Angle	Cachetype	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Time(millsec)
45	8x8	786432	7748976	21.48984	19.54002	58.26298
45	8x16	786432	9448710	22.52922	19.54002	71.04293
45	8x32	786432	12982833	23.04993	19.54002	97.61529
45	8x64	786432	20118410	23.31098	19.54002	151.2662
45	16x8	786432	7748976	21.48984	19.54002	58.26298
45	16x16	786432	9448710	22.52922	19.54002	71.04293
45	16x32	786432	12982833	23.04993	19.54002	97.61529
45	16x64	786432	20118410	23.31098	19.54002	151.2662
45	32x8	786432	7748976	21.48984	19.54002	58.26298
45	32x16	786432	9448710	22.52922	19.54002	71.04293
45	32x32	786432	12982833	23.04993	19.54002	97.61529
45	32x64	786432	20118410	23.31098	19.54002	151.2662
45	64x8	786432	7748976	21.48984	19.54002	58.26298
45	64x16	786432	9448710	22.52922	19.54002	71.04293
45	64x32	786432	12982833	23.04993	19.54002	97.61529

45	64x64	786432	20118410	23.31098	19.54002	151.2662

Table A.3: Cache Access Times (0 degrees)

Angle	Cachetype	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Time(millsec)
0	8x8	786432	2267136	87.5	0	17.04614
0	8x16	786432	1726464	93.75	0	12.98093
0	8x32	786432	1456128	96.875	0	10.94833
0	8x64	786432	1320960	98.4375	0	9.93203
0	16x8	786432	2267136	87.5	0	17.04614
0	16x16	786432	1726464	93.75	0	12.98093
0	16x32	786432	1456128	96.875	0	10.94833
0	16x64	786432	1320960	98.4375	0	9.93203
0	32x8	786432	2267136	87.5	0	17.04614
0	32x16	786432	1726464	93.75	0	12.98093
0	32x32	786432	1456128	96.875	0	10.94833
0	32x64	786432	1320960	98.4375	0	9.93203
0	64x8	786432	2267136	87.5	0	17.04614
0	64x16	786432	1726464	93.75	0	12.98093
0	64x32	786432	1456128	96.875	0	10.94833
0	64x64	786432	1320960	98.4375	0	9.93203

Table A.4: Cache with Blocks Simulation

Angle	Blocksize	Cachetype	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Time (milli-seconds)
45	64	64x32	786432	1720845	76.52232	19.54002	12.93868
45	64	64x64	786432	1734158	77.94813	19.54002	13.03878
45	32	64x64	786432	1852801	77.88798	19.54002	13.93084
45	64	64x16	786432	1861184	73.92515	19.54002	13.99387
45	32	64x32	786432	1997781	75.68105	19.54002	15.02091
45	32	32x32	786432	2031720	75.52122	19.54002	15.27609
45	16	32x64	786432	2049237	77.88849	19.54002	15.4078
45	16	64x64	786432	2049237	77.88849	19.54002	15.4078
45	32	32x16	786432	2145650	72.67927	19.54002	16.13271

45	32	64x16	786432	2145650	72.67927	19.54002	16.13271
45	32	32x64	786432	2166056	76.96165	19.54002	16.28614
45	64	64x8	786432	2242686	68.94849	19.54002	16.8623
45	16	32x32	786432	2244312	75.44594	19.54002	16.87453
45	16	64x32	786432	2244312	75.44594	19.54002	16.87453
45	8	16x64	786432	2429553	77.92664	19.54002	18.26732
45	8	32x64	786432	2429553	77.92664	19.54002	18.26732
45	8	64x64	786432	2429553	77.92664	19.54002	18.26732
45	32	32x8	786432	2498910	67.60979	19.54002	18.7888
45	32	64x8	786432	2498910	67.60979	19.54002	18.7888
45	16	32x16	786432	2581582	71.0776	19.54002	19.41039
45	16	64x16	786432	2581582	71.0776	19.54002	19.41039
45	16	16x32	786432	2616939	73.69105	19.54002	19.67623
45	16	16x16	786432	2618233	70.83231	19.54002	19.68596
45	8	16x32	786432	2622489	75.51676	19.54002	19.71796
45	8	32x32	786432	2622489	75.51676	19.54002	19.71796
45	8	64x32	786432	2622489	75.51676	19.54002	19.71796
45	16	16x8	786432	2963853	65.33508	19.54002	22.28461
45	16	32x8	786432	2963853	65.33508	19.54002	22.28461
45	16	64x8	786432	2963853	65.33508	19.54002	22.28461
45	16	16x64	786432	2993775	75.09537	19.54002	22.50959
45	8	16x16	786432	3030430	70.70528	19.54002	22.78519
45	8	32x16	786432	3030430	70.70528	19.54002	22.78519
45	8	64x16	786432	3030430	70.70528	19.54002	22.78519
45	8	8x16	786432	3492282	67.61437	19.54002	26.25776
45	8	16x8	786432	3711024	62.33457	19.54002	27.90244
45	8	32x8	786432	3711024	62.33457	19.54002	27.90244
45	8	64x8	786432	3711024	62.33457	19.54002	27.90244
45	8	8x32	786432	3712938	70.38129	19.54002	27.91683
45	8	8x8	786432	3740979	62.08064	19.54002	28.12766
45	8	8x64	786432	4511441	71.77023	19.54002	33.92061
45	128	64x8	786432	5235924	43.15783	19.54002	39.36785
45	64	32x8	786432	5520786	41.1597	19.54002	41.50967
45	32	16x8	786432	5878440	38.96116	19.54002	44.1988
45	128	64x16	786432	5990626	45.96011	19.54002	45.0423
45	64	32x16	786432	6360422	43.81421	19.54002	47.82272
45	16	8x8	786432	6464598	35.6589	19.54002	48.606

45	32	16x16	786432	6812620	41.4458	19.54002	51.22271
45	16	8x16	786432	7541855	37.88121	19.54002	56.70568
45	128	32x8	786432	7674234	22.48802	19.54002	57.70101
45	128	16x8	786432	7782174	21.573	19.54002	58.51259
45	128	8x8	786432	7803594	21.39142	19.54002	58.67364
45	64	16x8	786432	7811091	21.74454	19.54002	58.73001
45	128	64x32	786432	7862874	47.36493	19.54002	59.11936
45	64	8x8	786432	7863141	21.30331	19.54002	59.12136
45	32	8x8	786432	7976400	21.17653	19.54002	59.97293
45	64	32x32	786432	8381340	45.1547	19.54002	63.01759
45	32	16x32	786432	9002823	42.69078	19.54002	67.6904
45	128	32x16	786432	9329895	23.61221	19.54002	70.14959
45	128	16x16	786432	9477829	22.62217	19.54002	71.26188
45	64	16x16	786432	9498196	22.81481	19.54002	71.41501
45	128	8x16	786432	9507336	22.4247	19.54002	71.48373
45	64	8x16	786432	9569750	22.33594	19.54002	71.95301
45	32	8x16	786432	9686655	22.21146	19.54002	72.832
45	16	8x32	786432	9982485	39.00299	19.54002	75.05628
45	128	64x64	786432	11789834	48.06684	19.54002	88.64537
45	64	32x64	786432	12600043	45.8163	19.54002	94.73717
45	128	32x32	786432	12786378	24.17768	19.54002	96.13818
45	128	16x32	786432	13005591	23.14529	19.54002	97.7864
45	64	16x32	786432	13012434	23.34455	19.54002	97.83785
45	128	8x32	786432	13048359	22.94388	19.54002	98.10796
45	64	8x32	786432	13116654	22.85372	19.54002	98.62146
45	32	8x32	786432	13240446	22.73369	19.54002	99.55223
45	32	16x64	786432	13543426	43.31729	19.54002	101.8303
45	16	8x64	786432	15011802	39.5565	19.54002	112.8707
45	128	32x64	786432	19774547	24.45501	19.54002	148.6808
45	64	16x64	786432	20109090	23.61107	19.54002	151.1962
45	128	16x64	786432	20129770	23.40457	19.54002	151.3517
45	128	8x64	786432	20198097	23.20252	19.54002	151.8654
45	64	8x64	786432	20277435	23.11325	19.54002	152.4619
45	32	8x64	786432	20416417	22.99296	19.54002	153.5069

Table A.5: 64x32 Cache with Block 64

Angle	Blocksize	Cachetype	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Time (milli-seconds)
0	64	64x32	786432	1548288	96.875	0	11.64126
1	64	64x32	786432	1834245	94.68981	0.83847	13.79132
2	64	64x32	786432	1843425	93.78166	1.70339	13.86034
3	64	64x32	786432	1844532	92.93989	2.539953	13.86866
4	64	64x32	786432	1843749	92.13499	3.348541	13.86278
5	64	64x32	786432	1834380	91.39735	4.1303	13.79233
6	64	64x32	786432	1829952	90.66251	4.885992	13.75904
7	64	64x32	786432	1826955	89.9465	5.616125	13.7365
8	64	64x32	786432	1823445	89.25768	6.321462	13.71011
9	64	64x32	786432	1821474	88.58515	7.003275	13.69529
10	64	64x32	786432	1815507	87.95561	7.660929	13.65043
11	64	64x32	786432	1811403	87.33864	8.297221	13.61957
12	64	64x32	786432	1806516	86.74876	8.910115	13.58283
13	64	64x32	786432	1802169	86.17796	9.501393	13.55014
14	64	64x32	786432	1799064	85.62431	10.06966	13.5268
15	64	64x32	786432	1793745	85.09699	10.62203	13.4868
16	64	64x32	786432	1791369	84.57807	11.15214	13.46894
17	64	64x32	786432	1785942	84.09296	11.6628	13.42814
18	64	64x32	786432	1782432	83.61791	12.15439	13.40174
19	64	64x32	786432	1779408	83.15926	12.62728	13.37901
20	64	64x32	786432	1775277	82.72387	13.08212	13.34795
21	64	64x32	786432	1772442	82.30019	13.51916	13.32663
22	64	64x32	786432	1769877	81.89227	13.93916	13.30735
23	64	64x32	786432	1766502	81.5061	14.34123	13.28197
24	64	64x32	786432	1763721	81.13289	14.72753	13.26106
25	64	64x32	786432	1760319	80.77837	15.09806	13.23548
26	64	64x32	786432	1757457	80.43848	15.45143	13.21396
27	64	64x32	786432	1753542	80.11831	15.79005	13.18453
28	64	64x32	786432	1750356	79.8102	16.11315	13.16057
29	64	64x32	786432	1747170	79.51737	16.421	13.13662
30	64	64x32	786432	1746873	79.22516	16.71461	13.13438
31	64	64x32	786432	1746981	78.94529	16.99397	13.1352
32	64	64x32	786432	1744416	78.69059	17.26074	13.11591
33	64	64x32	786432	1741176	78.45637	17.51022	13.09155

34	64	64x32	786432	1737828	78.23435	17.74801	13.06638
35	64	64x32	786432	1736208	78.01743	17.97257	13.0542
36	64	64x32	786432	1735344	77.80978	18.18428	13.0477
37	64	64x32	786432	1733562	77.61892	18.38354	13.0343
38	64	64x32	786432	1731132	77.4437	18.5702	13.01603
39	64	64x32	786432	1729836	77.27522	18.74479	13.00629
40	64	64x32	786432	1728918	77.11703	18.90729	12.99938
41	64	64x32	786432	1727082	76.97462	19.05836	12.98558
42	64	64x32	786432	1724814	76.84555	19.1981	12.96853
43	64	64x32	786432	1723680	76.72183	19.32716	12.96
44	64	64x32	786432	1722897	76.60751	19.44517	12.95411
45	64	64x32	786432	1720845	76.52232	19.54002	12.93868

Table A.6: 2-way Set Associative Cache

Angle	Blocksize	Cachetype	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Time (milli -seconds)
45	32	64x8	786432	1550295	75.6513	19.54002	11.65635
45	32	64x16	786432	1701563	75.6513	19.54002	12.79371
45	16	64x8	786432	1771098	75.44619	19.54002	13.31653
45	16	64x16	786432	1928818	75.44619	19.54002	14.50239
45	32	64x32	786432	2004099	75.6513	19.54002	15.06841
45	64	64x8	786432	2007096	70.94561	19.54002	15.09095
45	8	64x8	786432	2155869	75.51778	19.54002	16.20954
45	16	64x32	786432	2244258	75.44619	19.54002	16.87412
45	16	32x8	786432	2297523	70.98364	19.54002	17.27461
45	64	64x16	786432	2306392	70.94561	19.54002	17.34129
45	8	64x16	786432	2311337	75.51778	19.54002	17.37847
45	32	64x64	786432	2409350	76.2422	19.54002	18.11541
45	32	32x8	786432	2500230	67.5986	19.54002	18.79872
45	16	32x16	786432	2595623	70.98364	19.54002	19.51596
45	8	64x32	786432	2622273	75.51778	19.54002	19.71634
45	8	32x8	786432	2723529	70.70567	19.54002	20.47766
45	32	32x32	786432	2828355	71.76946	19.54002	21.26583
45	16	64x64	786432	2875138	75.44619	19.54002	21.61758
45	64	64x64	786432	2889482	74.53169	19.54002	21.72543
45	32	32x16	786432	2904814	67.5986	19.54002	21.84071
45	64	64x32	786432	2904984	70.94561	19.54002	21.84198

45	16	32x32	786432	2958057	72.08456	19.54002	22.24103
45	8	32x16	786432	3030373	70.70567	19.54002	22.78476
45	32	32x64	786432	3229489	73.81693	19.54002	24.28187
45	8	64x64	786432	3244145	75.51778	19.54002	24.39207
45	16	16x16	786432	3605359	64.22602	19.54002	27.10796
45	8	32x32	786432	3644061	70.70567	19.54002	27.39896
45	8	16x8	786432	3734259	62.1376	19.54002	28.07714
45	16	16x32	786432	3825108	68.00118	19.54002	28.76021
45	16	32x64	786432	3825825	72.63489	19.54002	28.7656
45	16	16x8	786432	4027503	56.31841	19.54002	30.28198
45	8	16x16	786432	4043719	63.9239	19.54002	30.4039
45	16	16x64	786432	4737468	69.93904	19.54002	35.62006
45	8	32x64	786432	4871437	70.70567	19.54002	36.62735
45	8	16x32	786432	4892325	64.82697	19.54002	36.7844
45	8	8x8	786432	5086599	50.67368	19.54002	38.2451
45	8	8x16	786432	5159152	56.45892	19.54002	38.79062
45	128	64x8	786432	5299359	42.62009	19.54002	39.8448
45	64	32x8	786432	5464656	41.63551	19.54002	41.08764
45	8	8x32	786432	6047304	59.38759	19.54002	45.46845
45	32	16x8	786432	6161655	36.56031	19.54002	46.32823
45	128	64x16	786432	6489699	42.62009	19.54002	48.79473
45	16	8x8	786432	6517983	35.20635	19.54002	49.00739
45	64	32x16	786432	6685968	41.63551	19.54002	50.27043
45	8	16x64	786432	6706892	65.27799	19.54002	50.42776
45	32	16x16	786432	7042216	39.90924	19.54002	52.94899
45	16	8x16	786432	7338422	39.24268	19.54002	55.17611
45	128	32x8	786432	7459269	24.3103	19.54002	56.08473
45	64	16x8	786432	7588446	23.63192	19.54002	57.05598
45	32	8x8	786432	7722450	23.32929	19.54002	58.06353
45	128	16x8	786432	7733709	21.98385	19.54002	58.14819
45	128	8x8	786432	7758909	21.77022	19.54002	58.33766
45	64	8x8	786432	7824651	21.62959	19.54002	58.83196
45	8	8x64	786432	8201056	60.85955	19.54002	61.66207
45	64	32x32	786432	8681283	43.74212	19.54002	65.27281
45	128	64x32	786432	8870379	42.62009	19.54002	66.69458
45	32	16x32	786432	9170844	41.89949	19.54002	68.95372
45	64	16x16	786432	9211182	24.73564	19.54002	69.25701

45	128	32x16	786432	9225585	24.3103	19.54002	69.3653
45	32	8x16	786432	9421491	23.98605	19.54002	70.83828
45	128	8x16	786432	9424021	22.98228	19.54002	70.8573
45	128	16x16	786432	9464966	22.70826	19.54002	71.16516
45	64	8x16	786432	9500818	22.79727	19.54002	71.43472
45	16	8x32	786432	9536715	41.10235	19.54002	71.70463
45	128	64x64	786432	12563662	45.77853	19.54002	94.46362
45	128	32x32	786432	12586902	25.11711	19.54002	94.63836
45	64	16x32	786432	12621717	25.18463	19.54002	94.90013
45	32	8x32	786432	12891930	24.37503	19.54002	96.93181
45	128	8x32	786432	12911874	23.58666	19.54002	97.08176
45	64	32x64	786432	12973154	44.71296	19.54002	97.54251
45	128	16x32	786432	12995466	23.19298	19.54002	97.71027
45	64	8x32	786432	13005927	23.37519	19.54002	97.78892
45	32	16x64	786432	13726950	42.77458	19.54002	103.2102
45	16	8x64	786432	14163584	42.06479	19.54002	106.4931
45	128	32x64	786432	19459443	25.38681	19.54002	146.3116
45	64	16x64	786432	19505499	25.39597	19.54002	146.6579
45	32	8x64	786432	19882400	24.57212	19.54002	149.4917
45	128	8x64	786432	19961296	23.90277	19.54002	150.0849
45	64	8x64	786432	20086601	23.67757	19.54002	151.0271
45	128	16x64	786432	20118977	23.43648	19.54002	151.2705

Table A.6 lists all the access times and hit rates for various cache sizes and blocks sizes. The table is arranged in the ascending order of access times. So the first value in the table is the fastest access time produced by the 2-way set associative cache.

Table A.7: Bilinear Interpolation with SDRAM

Angle	Pixels	Cycles	Ignore (%)	Time(milliseconds)
0	786432	14149632	0	106.388211
1	786432	14042036	0.896708	105.57922
2	786432	13926589	1.760229	104.711197
3	786432	13814780	2.596537	103.870526
4	786432	13706830	3.403981	103.058875
5	786432	13602365	4.185359	102.273419
6	786432	13501419	4.940414	101.514429

7	786432	13403822	5.67042	100.780614
8	786432	13309574	6.375376	100.071982
9	786432	13218505	7.056554	99.387258
10	786432	13130581	7.714208	98.726176
11	786432	13045683	8.349228	98.08784
12	786432	12963692	8.962504	97.471371
13	786432	12884693	9.553401	96.877389
14	786432	12808805	10.121028	96.306801
15	786432	12734991	10.673141	95.751815
16	786432	12663999	11.204147	95.21804
17	786432	12595863	11.713791	94.705738
18	786432	12530226	12.204742	94.212227
19	786432	12466986	12.677765	93.736738
20	786432	12406211	13.13235	93.279779
21	786432	12347884	13.568624	92.84123
22	786432	12291784	13.98824	92.419431
23	786432	12237911	14.3912	92.014365
24	786432	12186401	14.776484	91.627076
25	786432	12136982	15.146129	91.255501
26	786432	12089586	15.500641	90.89914
27	786432	12044451	15.838242	90.559781
28	786432	12001254	16.161346	90.234995
29	786432	11960097	16.469193	89.925542
30	786432	11920844	16.762796	89.630403
31	786432	11883529	17.041906	89.349844
32	786432	11848424	17.304483	89.085892
33	786432	11814509	17.558161	88.830896
34	786432	11782719	17.795944	88.591874
35	786432	11752697	18.020502	88.366143
36	786432	11724324	18.232727	88.152811
37	786432	11697787	18.431219	87.953284
38	786432	11672865	18.61763	87.765902
39	786432	11649592	18.791708	87.590918
40	786432	11627849	18.95434	87.427437
41	786432	11607602	19.105783	87.2752
42	786432	11588936	19.245401	87.13486
43	786432	11571647	19.37472	87.004863

44	786432	11555973	19.491959	86.887017
45	786432	11544498	19.577789	86.800739

Table A.8: Cache 64x32, Block 64 Bilinear

Angle (degrees)	Block size	Cache Type (l x w)	Pixels	Cycles	Hit Rate(%)	Access Time (milliseconds)
0	64	64x32	786432	4928211	98.01709	37.05422
1	64	64x32	786432	5062248	96.99751	38.06201
2	64	64x32	786432	5193501	95.95404	39.04888
3	64	64x32	786432	5177490	95.11308	38.9285
4	64	64x32	786432	5164083	94.29782	38.82769
5	64	64x32	786432	5147151	93.51428	38.70038
6	64	64x32	786432	5124570	92.76419	38.5306
7	64	64x32	786432	5105346	92.03641	38.38606
8	64	64x32	786432	5088327	91.33151	38.2581
9	64	64x32	786432	5072106	90.64986	38.13614
10	64	64x32	786432	5056401	89.99243	38.01805
11	64	64x32	786432	5042064	89.35534	37.91026
12	64	64x32	786432	5025255	88.74521	37.78387
13	64	64x32	786432	5010657	88.15469	37.67411
14	64	64x32	786432	4996197	87.58768	37.56539
15	64	64x32	786432	4982004	87.03667	37.45868
16	64	64x32	786432	4970064	86.50589	37.3689
17	64	64x32	786432	4959663	85.99329	37.2907
18	64	64x32	786432	4947363	85.50253	37.19822
19	64	64x32	786432	4934748	85.03135	37.10337
20	64	64x32	786432	4923504	84.57712	37.01883
21	64	64x32	786432	4911222	84.1424	36.92648
22	64	64x32	786432	4903581	83.71973	36.86903
23	64	64x32	786432	4894014	83.31776	36.7971
24	64	64x32	786432	4884468	82.93196	36.72532
25	64	64x32	786432	4877049	82.55988	36.66954
26	64	64x32	786432	4866849	82.20869	36.59285
27	64	64x32	786432	4858482	81.87052	36.52994

28	64	64x32	786432	4852209	81.54583	36.48277
29	64	64x32	786432	4844460	81.23856	36.42451
30	64	64x32	786432	4838613	80.94368	36.38055
31	64	64x32	786432	4833777	80.66225	36.34419
32	64	64x32	786432	4823028	80.40072	36.26337
33	64	64x32	786432	4822029	80.14549	36.25586
34	64	64x32	786432	4816095	79.90808	36.21124
35	64	64x32	786432	4811769	79.68238	36.17872
36	64	64x32	786432	4807935	79.4693	36.14989
37	64	64x32	786432	4801803	79.27173	36.10378
38	64	64x32	786432	4792647	79.09066	36.03494
39	64	64x32	786432	4785072	78.92014	35.97799
40	64	64x32	786432	4779753	78.75938	35.93799
41	64	64x32	786432	4772544	78.61261	35.88379
42	64	64x32	786432	4765737	78.47701	35.83261
43	64	64x32	786432	4759344	78.35188	35.78454
44	64	64x32	786432	4755615	78.23499	35.7565
45	64	64x32	786432	4734099	78.16283	35.59473

Table A.9: SDRAM with blocks (comparison)

Angle	Pixels	Cycles	SDRAM with Blocks(milliseconds)	SDRAM, no blocks,no cache (milliseconds)	64x32 Cache, BlockSize 64 (milliseconds)
0	786432	1155065	8.684699	5.999587	11.64126
5	786432	1595890	11.99917	9.452797	13.79233
35	786432	3667848	27.5778	24.56711	13.0542
45	786432	4199407	31.57449	29.50932	12.93868

Table A.10: Cache 64x32, Block 64 (Sample LUTs)

Test LUT	Pixels	Cycles	Hit Rate (%)	Ignore (%)	Access Time (milliseconds)
I (90	786432	1382400	72.705208	24.951044	10.39399

degrees)					
II	786432	863401	89.52726	7.719294	6.491737
III	786432	830897	85.397339	7.719294	6.247346

Table A.11: SDRAM (Sample LUTs)

Test LUT	Pixels	Cycles	Same Row Access (%)	Change in Row (%)	Ignore (%)	Access Time (milli-seconds)
I (90 degrees)	786432	4921344	0.048955	75	24.95104	37.00259
II	786432	1320726	82.686745	9.593964	7.719294	9.930271
III	786432	1696220	75.865807	16.414896	7.719294	12.75353

Table A.12: SDRAM Vs Cache (Sample LUTs)

Test LUT	RAM Access Time (milliseconds)	Cache Design Access time (milliseconds)
I (90 degrees)	37.00259	10.39399
II	9.930271	6.491737
III	12.75353	6.247346

Table A.13: Cache 64x32, Block 64 (Sample LUTs, Bilinear Interpolation)

Test LUT	Pixels	Cycles	Hit Rate (%)	Access Time (milliseconds)
I (90 degrees)	786432	2582695	72.738899	19.41876
II	786432	2977696	89.603584	22.38869
III	786432	2953412	88.831619	22.20611

Table A.14: SDRAM (Sample LUTs, Bilinear Interpolation)

Test LUT	Pixels	Cycles	Access Time (milliseconds)
I (90 degrees)	786432	10826129	81.3994638
II	786432	12916800	97.1187958
III	786432	13127255	98.701164

Table A.15: SDRAM Vs Cache (Sample LUTs, Bilinear Interpolation)

Test LUT	RAM Access Time (milliseconds)	Cache Design Access time (milliseconds)
I (90 degrees)	81.39946	19.41876
II	97.1188	22.38869
III	98.70116	22.20611

Appendix B: Simulation Code

B.1) SDRAM Simulation

The following C code was written to simulate what happens while accessing a test LUT in case bilinear interpolation is performed. The *ramaccess* function is the main function in the code and its logic is used by all simulations involving the SDRAM.

```
#include"stdafx.h"
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>

//GETLUT FUNCTION DECLARATIONS
void getlut(void);
unsigned int convert(char *);
#define pi 3.1457
double * ptr;
int w=1024,l=768,*chid,*lineid,hexerr;

//RAM ACCESS
#define clk 133000000.0
void ramaccess(void);

int main(void) {
      printf("\nReading LUT...............");
      getlut();
      printf("Finished!");
      ramaccess();
      getch();
      return(0);
}

void getlut(void) {
      FILE *f1;
      char a[30];
```

```
char b[6],*p;
p=&b[0];
int i,j;
unsigned int u,v,value;
ptr=new double [1024*768];
chid=new int[1024*768];
lineid=new int[8];
f1=fopen("lut0.txt","r+");
fscanf(f1,"%s",&a[0]);
fscanf(f1,"%s",&a[0]);
fscanf(f1,"%s",&a[0]);
for (j=0;j<l;j++) {
      for (i=0;i<w;i++) {
            fscanf(f1,"%s",&a[0]);
            b[0]=a[3];
            b[1]=a[4];
            b[2]=a[5];
            b[3]=a[6];
            b[4]=a[7];
            b[5]=NULL;
            if(a[0]=='0' || a[0]=='1' || a[0]=='2' || a[0]=='3'
|| a[0]=='4' || a[0]=='5' || a[0]=='6' || a[0]=='7') {
                  value=convert(p);
                  ptr[j*1024+i]=value;
            }
            else {
                  ptr[j*1024+i]=-1;
            }
      }
}
fclose(f1);
f1=fopen("lut0.txt","r+");
fscanf(f1,"%s",&a[0]);
fscanf(f1,"%s",&a[0]);
fscanf(f1,"%s",&a[0]);
for (j=0;j<768;j++) {
      for (i=0;i<1024;i++) {
            fscanf(f1,"%s",&a[0]);
```

```
                b[0]='0';b[1]='0';b[2]='0';b[3]='0';
                b[4]=a[2];
                b[5]=NULL;
                if(a[0]=='0' || a[0]=='1' || a[0]=='2' || a[0]=='3'
|| a[0]=='4' || a[0]=='5' || a[0]=='6' || a[0]=='7') {
                        value=convert(p);
                        chid[j*1024+i]=(value&0x6)>>1;
                }
                else {
                        chid[j*1024+i]=-1;
                }
            }
        }
        fclose(f1);
}

unsigned int convert(char * ppp) {
        unsigned int ret=0,i;
        double j=0,tt;
        unsigned int ttt;
        char dum[21];
        // Hexadecimal 1
        if(ppp[0]=='0') {
            dum[0]='0';dum[1]='0';dum[2]='0';dum[3]='0';
        }
        else if (ppp[0]=='1') {
            dum[0]='0';dum[1]='0';dum[2]='0';dum[3]='1';
        }
        else if (ppp[0]=='2') {
            dum[0]='0';dum[1]='0';dum[2]='1';dum[3]='0';
        }
        else if (ppp[0]=='3') {
            dum[0]='0';dum[1]='0';dum[2]='1';dum[3]='1';
        }
        else if (ppp[0]=='4') {
            dum[0]='0';dum[1]='1';dum[2]='0';dum[3]='0';
        }
        else if (ppp[0]=='5') {
```

```
        dum[0]='0';dum[1]='1';dum[2]='0';dum[3]='1';
}
else if (ppp[0]=='6') {
        dum[0]='0';dum[1]='1';dum[2]='1';dum[3]='0';
}
else if (ppp[0]=='7') {
        dum[0]='0';dum[1]='1';dum[2]='1';dum[3]='1';
}
else if (ppp[0]=='8') {
        dum[0]='1';dum[1]='0';dum[2]='0';dum[3]='0';
}
else if (ppp[0]=='9') {
        dum[0]='1';dum[1]='0';dum[2]='0';dum[3]='1';
}
else if (ppp[0]=='A' || ppp[0]=='a') {
        dum[0]='1';dum[1]='0';dum[2]='1';dum[3]='0';
}
else if (ppp[0]=='B' || ppp[0]=='b') {
        dum[0]='1';dum[1]='0';dum[2]='1';dum[3]='1';
}
else if (ppp[0]=='C' || ppp[0]=='c') {
        dum[0]='1';dum[1]='1';dum[2]='0';dum[3]='0';
}
else if (ppp[0]=='D' || ppp[0]=='d') {
        dum[0]='1';dum[1]='1';dum[2]='0';dum[3]='1';
}
else if (ppp[0]=='E' || ppp[0]=='e') {
        dum[0]='1';dum[1]='1';dum[2]='1';dum[3]='0';
}
else if (ppp[0]=='F' || ppp[0]=='f') {
        dum[0]='1';dum[1]='1';dum[2]='1';dum[3]='1';
}
else {
        hexerr=-1;
}
//hexadecimal 2
if(ppp[1]=='0') {
        dum[4]='0';dum[5]='0';dum[6]='0';dum[7]='0';
```

```
}
else if (ppp[1]=='1') {
      dum[4]='0';dum[5]='0';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='2') {
      dum[4]='0';dum[5]='0';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='3') {
      dum[4]='0';dum[5]='0';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='4') {
      dum[4]='0';dum[5]='1';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='5') {
      dum[4]='0';dum[5]='1';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='6') {
      dum[4]='0';dum[5]='1';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='7') {
      dum[4]='0';dum[5]='1';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='8') {
      dum[4]='1';dum[5]='0';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='9') {
      dum[4]='1';dum[5]='0';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='A' || ppp[1]=='a') {
      dum[4]='1';dum[5]='0';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='B' || ppp[1]=='b') {
      dum[4]='1';dum[5]='0';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='C' || ppp[1]=='c') {
      dum[4]='1';dum[5]='1';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='D' || ppp[1]=='d') {
```

```
        dum[4]='1';dum[5]='1';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='E' || ppp[1]=='e') {
        dum[4]='1';dum[5]='1';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='F' || ppp[1]=='f') {
        dum[4]='1';dum[5]='1';dum[6]='1';dum[7]='1';
}
else {
        hexerr=-1;
}
//hexadecimal 3
if(ppp[2]=='0') {
        dum[8]='0';dum[9]='0';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='1') {
        dum[8]='0';dum[9]='0';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='2') {
        dum[8]='0';dum[9]='0';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='3') {
        dum[8]='0';dum[9]='0';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='4') {
        dum[8]='0';dum[9]='1';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='5') {
        dum[8]='0';dum[9]='1';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='6') {
        dum[8]='0';dum[9]='1';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='7') {
        dum[8]='0';dum[9]='1';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='8') {
        dum[8]='1';dum[9]='0';dum[10]='0';dum[11]='0';
```

```
}
else if (ppp[2]=='9') {
      dum[8]='1';dum[9]='0';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='A' || ppp[2]=='a') {
      dum[8]='1';dum[9]='0';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='B' || ppp[2]=='b') {
      dum[8]='1';dum[9]='0';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='C' || ppp[2]=='c') {
      dum[8]='1';dum[9]='1';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='D' || ppp[2]=='d') {
      dum[8]='1';dum[9]='1';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='E' || ppp[2]=='e') {
      dum[8]='1';dum[9]='1';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='F' || ppp[2]=='f') {
      dum[8]='1';dum[9]='1';dum[10]='1';dum[11]='1';
}
else {
      hexerr=-1;
}
//hexadecimal 4
if(ppp[3]=='0') {
      dum[12]='0';dum[13]='0';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='1') {
      dum[12]='0';dum[13]='0';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='2') {
      dum[12]='0';dum[13]='0';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='3') {
      dum[12]='0';dum[13]='0';dum[14]='1';dum[15]='1';
}
```

```
else if (ppp[3]=='4') {
      dum[12]='0';dum[13]='1';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='5') {
      dum[12]='0';dum[13]='1';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='6') {
      dum[12]='0';dum[13]='1';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='7') {
      dum[12]='0';dum[13]='1';dum[14]='1';dum[15]='1';
}
else if (ppp[3]=='8') {
      dum[12]='1';dum[13]='0';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='9') {
      dum[12]='1';dum[13]='0';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='A' || ppp[3]=='a') {
      dum[12]='1';dum[13]='0';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='B' || ppp[3]=='b') {
      dum[12]='1';dum[13]='0';dum[14]='1';dum[15]='1';
}
else if (ppp[3]=='C' || ppp[3]=='c') {
      dum[12]='1';dum[13]='1';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='D' || ppp[3]=='d') {
      dum[12]='1';dum[13]='1';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='E' || ppp[3]=='e') {
      dum[12]='1';dum[13]='1';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='F' || ppp[3]=='f') {
      dum[12]='1';dum[13]='1';dum[14]='1';dum[15]='1';
}
else {
      hexerr=-1;
```

```
}

//hexadecimal 5
if(ppp[4]=='0') {
      dum[16]='0';dum[17]='0';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='1') {
      dum[16]='0';dum[17]='0';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='2') {
      dum[16]='0';dum[17]='0';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='3') {
      dum[16]='0';dum[17]='0';dum[18]='1';dum[19]='1';
}
else if (ppp[4]=='4') {
      dum[16]='0';dum[17]='1';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='5') {
      dum[16]='0';dum[17]='1';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='6') {
      dum[16]='0';dum[17]='1';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='7') {
      dum[16]='0';dum[17]='1';dum[18]='1';dum[19]='1';
}
else if (ppp[4]=='8') {
      dum[16]='1';dum[17]='0';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='9') {
      dum[16]='1';dum[17]='0';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='A' || ppp[4]=='a') {
      dum[16]='1';dum[17]='0';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='B' || ppp[4]=='b') {
      dum[16]='1';dum[17]='0';dum[18]='1';dum[19]='1';
```

```
        }
        else if (ppp[4]=='C' || ppp[4]=='c') {
                dum[16]='1';dum[17]='1';dum[18]='0';dum[19]='0';
        }
        else if (ppp[3]=='D' || ppp[4]=='d') {
                dum[16]='1';dum[17]='1';dum[18]='0';dum[19]='1';
        }
        else if (ppp[4]=='E' || ppp[4]=='e') {
                dum[16]='1';dum[17]='1';dum[18]='1';dum[19]='0';
        }
        else if (ppp[4]=='F' || ppp[4]=='f') {
                dum[16]='1';dum[17]='1';dum[18]='1';dum[19]='1';
        }
        else {
                hexerr=-1;
        }
        dum[20]=NULL;
        j=19.0;
        for(i=0;i<20;i++) {
                tt=pow(2.0,j);
                ttt=(unsigned int)tt;
                if(dum[i]=='0') {
                        ret=ret+(0*ttt);
                }
                else {
                        ret=ret+(1*ttt);
                }
                j=j-1.0;
        }
        return(ret);
}



void ramaccess(void) {
        FILE *f2;
        f2=fopen("ramaccessbilinear.csv","a+");
        int i,j,currow=0;
```

```
	long int cycles=0,pixels=0;
	float timee,p_cir,p_sr,same_row=0.0,change_in_row=0.0,ignore=0.0;
	int x1,x2,y1,y2;
	for(j=0;j<l;j++) {
		for(i=0;i<w;i++) {
			pixels=pixels+1;
			if(ptr[j*w+i]<0) {
				cycles=cycles+1;
				ignore=ignore+4.0;
			}
			else {
				y1=floor((ptr[j*w+i]/w));
				y2=ceil((ptr[j*w+i]/w));
				if(currow==y1) {
					cycles=cycles+11;
					same_row=same_row+3;
					change_in_row=change_in_row+1;
					currow=y2;
				}
				else {
					cycles=cycles+18;
					change_in_row=change_in_row+2;
					same_row=same_row+2;
					currow=y2;
				}
			}
		}
		cycles=cycles+8;
	}
	timee=cycles/clk;
	p_sr=(same_row*100)/(w*l*4);
	p_cir=(change_in_row*100)/(w*l*4);
	ignore=(ignore*100)/(w*l*4);
	printf("\n%d\t%d\t%4.6f\t%4.6f\t%4.6f\t%4.6f\n",pixels,cycles,p_s
r,p_cir,ignore,timee);
	fprintf(f2,"%d,%d,%4.6f,%4.6f,%4.6f,%4.6f",pixels,cycles,p_sr,p_c
ir,ignore,timee*1000);
	fclose(f2);
```

```
}
```

B.2) Cache with Blocks

The following code was written to simulate what happens when there is cache with blocks while accessing test LUTs. The *cache* and *caloit* functions contain the main logic for the cache simulations. All the simulations involving the cache (even without blocks) use these two functions.

```
#include<stdafx.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>

//GETLUT FUNCTION DECLARATIONS
void getlut(void);
unsigned int convert(char *);
#define pi 3.1457
int * ptr;
int w=1024,l=768,*chid,*lineid,hexerr;

//GET BLOCKS DECLARATIONS
void getblocks(void);
int *ii,*jj,bwid=-1,blen=-1;

//CACHE ALGORITHM DECLARATIONS
#define clk 133000000.0
void cache(void);
int * caloit(int,int,int,int);
int l1=64,w1=32; // CACHE SIZE
struct cache {
      int jtag;
} cachevar[512];

int main(void) {
      int i;
      for(i=0;i<512;i++) {
```

```
            cachevar[i].jtag=-1;
        }
        printf("\nGetting LUT Values.........");
        getlut();
        printf("Finished!");
        printf("\nGetting Blocks............");
        getblocks();
        printf(". Finished!");
        cache();
        printf("\nFinished Process!");
        getch();
        delete(ptr);
        delete(chid);
        delete(lineid);
        delete(ii);
        delete(jj);
        return(0);
}

void getlut(void) {
        FILE *f1;
        char a[30];
        char b[6],*p;
        p=&b[0];
        int i,j;
        unsigned int u,v,value;
        ptr=new int [w*l];
        chid=new int[w*l];
        lineid=new int[8];
        f1=fopen("lut0.txt","r+");
        fscanf(f1,"%s",&a[0]);
        fscanf(f1,"%s",&a[0]);
        fscanf(f1,"%s",&a[0]);
        for (j=0;j<l;j++) {
            for (i=0;i<w;i++) {
                fscanf(f1,"%s",&a[0]);
                b[0]=a[3];
                b[1]=a[4];
```

```
                    b[2]=a[5];
                    b[3]=a[6];
                    b[4]=a[7];
                    b[5]=NULL;
                    if(a[0]=='0' || a[0]=='1' || a[0]=='2' || a[0]=='3'
|| a[0]=='4' || a[0]=='5' || a[0]=='6' || a[0]=='7') {
                          value=convert(p);
                          ptr[j*w+i]=value;
                    }
                    else {
                          ptr[j*w+i]=-1;
                    }
              }
        }
        fclose(f1);
        f1=fopen("lut0.txt","r+");
        fscanf(f1,"%s",&a[0]);
        fscanf(f1,"%s",&a[0]);
        fscanf(f1,"%s",&a[0]);
        for (j=0;j<l;j++) {
              for (i=0;i<w;i++) {
                    fscanf(f1,"%s",&a[0]);
                    b[0]='0';b[1]='0';b[2]='0';b[3]='0';
                    b[4]=a[2];
                    b[5]=NULL;
                    if(a[0]=='0' || a[0]=='1' || a[0]=='2' || a[0]=='3'
|| a[0]=='4' || a[0]=='5' || a[0]=='6' || a[0]=='7') {
                          value=convert(p);
                          chid[j*w+i]=(value&0x6)>>1;
                    }
                    else {
                          chid[j*w+i]=-1;
                    }
              }
        }
        fclose(f1);
}
```

```
unsigned int convert(char * ppp) {
	unsigned int ret=0,i;
	double j=0,tt;
	unsigned int ttt;
	char dum[21];
	// Hexadecimal 1
	if(ppp[0]=='0') {
		dum[0]='0';dum[1]='0';dum[2]='0';dum[3]='0';
	}
	else if (ppp[0]=='1') {
		dum[0]='0';dum[1]='0';dum[2]='0';dum[3]='1';
	}
	else if (ppp[0]=='2') {
		dum[0]='0';dum[1]='0';dum[2]='1';dum[3]='0';
	}
	else if (ppp[0]=='3') {
		dum[0]='0';dum[1]='0';dum[2]='1';dum[3]='1';
	}
	else if (ppp[0]=='4') {
		dum[0]='0';dum[1]='1';dum[2]='0';dum[3]='0';
	}
	else if (ppp[0]=='5') {
		dum[0]='0';dum[1]='1';dum[2]='0';dum[3]='1';
	}
	else if (ppp[0]=='6') {
		dum[0]='0';dum[1]='1';dum[2]='1';dum[3]='0';
	}
	else if (ppp[0]=='7') {
		dum[0]='0';dum[1]='1';dum[2]='1';dum[3]='1';
	}
	else if (ppp[0]=='8') {
		dum[0]='1';dum[1]='0';dum[2]='0';dum[3]='0';
	}
	else if (ppp[0]=='9') {
		dum[0]='1';dum[1]='0';dum[2]='0';dum[3]='1';
	}
	else if (ppp[0]=='A' || ppp[0]=='a') {
		dum[0]='1';dum[1]='0';dum[2]='1';dum[3]='0';
```

```
}
else if (ppp[0]=='B' || ppp[0]=='b') {
      dum[0]='1';dum[1]='0';dum[2]='1';dum[3]='1';
}
else if (ppp[0]=='C' || ppp[0]=='c') {
      dum[0]='1';dum[1]='1';dum[2]='0';dum[3]='0';
}
else if (ppp[0]=='D' || ppp[0]=='d') {
      dum[0]='1';dum[1]='1';dum[2]='0';dum[3]='1';
}
else if (ppp[0]=='E' || ppp[0]=='e') {
      dum[0]='1';dum[1]='1';dum[2]='1';dum[3]='0';
}
else if (ppp[0]=='F' || ppp[0]=='f') {
      dum[0]='1';dum[1]='1';dum[2]='1';dum[3]='1';
}
else {
      hexerr=-1;
}
//hexadecimal 2
if(ppp[1]=='0') {
      dum[4]='0';dum[5]='0';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='1') {
      dum[4]='0';dum[5]='0';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='2') {
      dum[4]='0';dum[5]='0';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='3') {
      dum[4]='0';dum[5]='0';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='4') {
      dum[4]='0';dum[5]='1';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='5') {
      dum[4]='0';dum[5]='1';dum[6]='0';dum[7]='1';
}
```

```
else if (ppp[1]=='6') {
      dum[4]='0';dum[5]='1';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='7') {
      dum[4]='0';dum[5]='1';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='8') {
      dum[4]='1';dum[5]='0';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='9') {
      dum[4]='1';dum[5]='0';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='A' || ppp[1]=='a') {
      dum[4]='1';dum[5]='0';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='B' || ppp[1]=='b') {
      dum[4]='1';dum[5]='0';dum[6]='1';dum[7]='1';
}
else if (ppp[1]=='C' || ppp[1]=='c') {
      dum[4]='1';dum[5]='1';dum[6]='0';dum[7]='0';
}
else if (ppp[1]=='D' || ppp[1]=='d') {
      dum[4]='1';dum[5]='1';dum[6]='0';dum[7]='1';
}
else if (ppp[1]=='E' || ppp[1]=='e') {
      dum[4]='1';dum[5]='1';dum[6]='1';dum[7]='0';
}
else if (ppp[1]=='F' || ppp[1]=='f') {
      dum[4]='1';dum[5]='1';dum[6]='1';dum[7]='1';
}
else {
      hexerr=-1;
}
//hexadecimal 3
if(ppp[2]=='0') {
      dum[8]='0';dum[9]='0';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='1') {
```

```
        dum[8]='0';dum[9]='0';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='2') {
        dum[8]='0';dum[9]='0';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='3') {
        dum[8]='0';dum[9]='0';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='4') {
        dum[8]='0';dum[9]='1';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='5') {
        dum[8]='0';dum[9]='1';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='6') {
        dum[8]='0';dum[9]='1';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='7') {
        dum[8]='0';dum[9]='1';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='8') {
        dum[8]='1';dum[9]='0';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='9') {
        dum[8]='1';dum[9]='0';dum[10]='0';dum[11]='1';
}
else if (ppp[2]=='A' || ppp[2]=='a') {
        dum[8]='1';dum[9]='0';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='B' || ppp[2]=='b') {
        dum[8]='1';dum[9]='0';dum[10]='1';dum[11]='1';
}
else if (ppp[2]=='C' || ppp[2]=='c') {
        dum[8]='1';dum[9]='1';dum[10]='0';dum[11]='0';
}
else if (ppp[2]=='D' || ppp[2]=='d') {
        dum[8]='1';dum[9]='1';dum[10]='0';dum[11]='1';
}
```

```
else if (ppp[2]=='E' || ppp[2]=='e') {
      dum[8]='1';dum[9]='1';dum[10]='1';dum[11]='0';
}
else if (ppp[2]=='F' || ppp[2]=='f') {
      dum[8]='1';dum[9]='1';dum[10]='1';dum[11]='1';
}
else {
      hexerr=-1;
}
//hexadecimal 4
if(ppp[3]=='0') {
      dum[12]='0';dum[13]='0';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='1') {
      dum[12]='0';dum[13]='0';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='2') {
      dum[12]='0';dum[13]='0';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='3') {
      dum[12]='0';dum[13]='0';dum[14]='1';dum[15]='1';
}
else if (ppp[3]=='4') {
      dum[12]='0';dum[13]='1';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='5') {
      dum[12]='0';dum[13]='1';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='6') {
      dum[12]='0';dum[13]='1';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='7') {
      dum[12]='0';dum[13]='1';dum[14]='1';dum[15]='1';
}
else if (ppp[3]=='8') {
      dum[12]='1';dum[13]='0';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='9') {
```

```
        dum[12]='1';dum[13]='0';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='A' || ppp[3]=='a') {
        dum[12]='1';dum[13]='0';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='B' || ppp[3]=='b') {
        dum[12]='1';dum[13]='0';dum[14]='1';dum[15]='1';
}
else if (ppp[3]=='C' || ppp[3]=='c') {
        dum[12]='1';dum[13]='1';dum[14]='0';dum[15]='0';
}
else if (ppp[3]=='D' || ppp[3]=='d') {
        dum[12]='1';dum[13]='1';dum[14]='0';dum[15]='1';
}
else if (ppp[3]=='E' || ppp[3]=='e') {
        dum[12]='1';dum[13]='1';dum[14]='1';dum[15]='0';
}
else if (ppp[3]=='F' || ppp[3]=='f') {
        dum[12]='1';dum[13]='1';dum[14]='1';dum[15]='1';
}
else {
        hexerr=-1;
}

//hexadecimal 5
if(ppp[4]=='0') {
        dum[16]='0';dum[17]='0';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='1') {
        dum[16]='0';dum[17]='0';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='2') {
        dum[16]='0';dum[17]='0';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='3') {
        dum[16]='0';dum[17]='0';dum[18]='1';dum[19]='1';
}
else if (ppp[4]=='4') {
```

```
        dum[16]='0';dum[17]='1';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='5') {
        dum[16]='0';dum[17]='1';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='6') {
        dum[16]='0';dum[17]='1';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='7') {
        dum[16]='0';dum[17]='1';dum[18]='1';dum[19]='1';
}
else if (ppp[4]=='8') {
        dum[16]='1';dum[17]='0';dum[18]='0';dum[19]='0';
}
else if (ppp[4]=='9') {
        dum[16]='1';dum[17]='0';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='A' || ppp[4]=='a') {
        dum[16]='1';dum[17]='0';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='B' || ppp[4]=='b') {
        dum[16]='1';dum[17]='0';dum[18]='1';dum[19]='1';
}
else if (ppp[4]=='C' || ppp[4]=='c') {
        dum[16]='1';dum[17]='1';dum[18]='0';dum[19]='0';
}
else if (ppp[3]=='D' || ppp[4]=='d') {
        dum[16]='1';dum[17]='1';dum[18]='0';dum[19]='1';
}
else if (ppp[4]=='E' || ppp[4]=='e') {
        dum[16]='1';dum[17]='1';dum[18]='1';dum[19]='0';
}
else if (ppp[4]=='F' || ppp[4]=='f') {
        dum[16]='1';dum[17]='1';dum[18]='1';dum[19]='1';
}
else {
        hexerr=-1;
}
```

```
	dum[20]=NULL;
	j=19.0;
	for(i=0;i<20;i++) {
		tt=pow(2.0,j);
		ttt=(unsigned int)tt;
		if(dum[i]=='0') {
			ret=ret+(0*ttt);
		}
		else {
			ret=ret+(1*ttt);
		}
		j=j-1.0;
	}
	return(ret);
}

void getblocks(void) {
	FILE *f1;
	char a[30];
	int no_blocks,i;
	f1=fopen("blocks0.txt","r+");
	fscanf(f1,"%s",&a[0]);
	fscanf(f1,"%s",&a[0]);
	bwid=blen=atoi(a);
	no_blocks=(w*l)/(bwid*blen);
	printf("\nNum of blocks....%d",no_blocks);
	ii=new int[no_blocks];
	jj=new int[no_blocks];
	for(i=0;i<no_blocks;i++) {
		fscanf(f1,"%s",&a[0]);
		ii[i]=atoi(a);
		fscanf(f1,"%s",&a[0]);
		jj[i]=atoi(a);
	}
	fclose(f1);
//	printf("\nLast i:%d Last j:%d",ii[no_blocks-1],jj[no_blocks-1]);
}
```

```
int * caloit (int tl,int tw, int tcr,int tcc) {
	int *oitt, oit[3];
	int lbits,wbits;
	int col_index_bits,row_index_bits,row_tag_bits,col_tag_bits;
	int
ampt_row_no=0,ampi_col_no=0,ampi_tag_no=0,degree_ct,degree_ci,degree_rt
,tcib,trtb,tctb;
	oit[0]=0,oit[1]=0,oit[2]=0;

	lbits=(int)( log(tl*1.0)/log(2.0) );
	wbits=(int)( log(tw*1.0)/log(2.0) );
	row_tag_bits=10-lbits;
	col_tag_bits=10-wbits;
	oit[0]=tcc&(((int)pow(2.0,wbits*1.0))-1);
	oit[1]=tcr&(((int)pow(2.0,lbits*1.0))-1);
	oit[2]=(((tcr>>lbits)&(((int)pow(2.0,row_tag_bits))-
1))<<col_tag_bits)|((tcc>>wbits)&(((int)pow(2.0,col_tag_bits))-1));
//	printf("\n%d",oit[0]);
//	getch();
/*	if((lbits+wbits)<=10) {
		row_index_bits=0;
		col_tag_bits=10-(wbits+lbits);
		col_index_bits=lbits;
		row_tag_bits=10-row_index_bits;
		degree_ct=9;
		degree_rt=9;
		degree_ci=lbits+wbits-1;
	}
	else {
		col_tag_bits=0;
		col_index_bits=10-wbits;
		row_index_bits=lbits-col_index_bits;
		row_tag_bits=10-row_index_bits;
		degree_ct=0;
		degree_rt=9;
		degree_ci=9;
	}
	tcib=col_index_bits;trtb=row_tag_bits;tctb=col_tag_bits;
```

```
	while(tctb>0) {
		ampi_tag_no=ampi_tag_no+ (int) ( pow(2.0,(degree_ct*1.0)));
		tctb=tctb-1;
		degree_ct=degree_ct-1;
	}

	while(tcib>0) {
		ampi_col_no=ampi_col_no+ (int) ( pow(2.0,(degree_ci*1.0)));
		tcib=tcib-1;
		degree_ci=degree_ci-1;
	}

	while(trtb>0) {
		ampt_row_no=ampt_row_no+ (int) ( pow(2.0,(degree_rt*1.0)));
		trtb=trtb-1;
		degree_rt=degree_rt-1;
	}

	oit[0]=tcc& ( (int)(pow(2.0,(wbits*1.0))-1) );
	oit[1]= ( (tcr& ( (int)(pow(2.0,(row_index_bits*1.0))-
1)))<<col_index_bits ) | ( (tcc&ampi_col_no)>>wbits);
	if((wbits+lbits)<=10) {
		oit[2]=( (tcr&ampt_row_no)<<(10-(wbits+lbits))) | (
(tcc&ampi_tag_no)>>(10-(wbits+lbits)) );
	}
	else {
		oit[2]=( (tcr&ampt_row_no)>>row_index_bits);
	}*/

	oitt=oit;
	return(oitt);
}
void cache(void) {
	FILE *f2;
	f2=fopen("cachebilinear.csv","a+");
	long int cycles=0,pixels=0;
	int k,j1,i1,currow,curcol,cachelookup;
	int tag=0,offset=0,index=0;
```

```
int x1,x2,y1,y2;
float timee,hits=0.0,ignore=0.0;
int *tttt,bilinear=0;
for(k=0;k<((w*l)/(blen*bwid));k++) {
for(j1=(jj[k]*bwid);j1<((jj[k]*bwid)+bwid);j1++) {
      for(i1=(ii[k]*blen);i1<((ii[k]*blen)+blen);i1++){
                  pixels=pixels+1;
                  if(ptr[j1*1024+i1]>=0) {
                        bilinear=0;
                        y1=ptr[j1*1024+i1]/1024;
                        x1=ptr[j1*1024+i1]-y1*1024;
                        y2=y1+1;
                        x2=x1+1;
                  //
printf("\nx1:%d,x2:%d,y1:%d,y2:%d",x1,x2,y1,y2);
                  //    printf("\n%d",cachevar[16].jtag);
                        currow=y1;curcol=x1;
      //                while(bilinear<4){   // INCLUDE THIS LINE
FOR BILINEAR INTERPOLATION
                        tttt=caloit(l1,w1,currow,curcol);
                        offset=tttt[0];
                        index=tttt[1];
                        tag=tttt[2];
                  //    printf("\n%d",cachevar[index].jtag);
                        if(tag==cachevar[index].jtag) {
                              cycles=cycles+1;
                              hits=hits+1;
                  //
printf("\n%d,%d,%d,%d,%d,Hit",currow,curcol,index,tag,offset);
                  //          getch();
                        }
                        else {
                              cachevar[index].jtag=tag;
                  //
printf("\n%d,%d,%d,%d,%d,Miss",currow,curcol,index,tag,offset);
                  //          getch();
                              if(w==8) {
                                    cycles=cycles+16;
```

```
                        }
                        if(w==16) {
                              cycles=cycles+20;
                        }
                        if(w==32) {
                              cycles=cycles+28;
                        }
                        if(w==64) {
                              cycles=cycles+44;
                        }
                        if(w==128) {
                              cycles=cycles+76;
                        }
                        if(w==256) {
                              cycles=cycles+140;
                        }
                        if(w==512) {
                              cycles=cycles+266;
                        }
                  }
//                bilinear=bilinear+1;
                  offset=0;index=0;tag=0;
                  if(currow==y1 && curcol==x1) {
                        currow=y1;curcol=x2;
                  }
                  else if (currow==y1 && curcol==x2) {
                        currow=y2,curcol=x1;
                  }
                  else {
                        currow=y2;curcol=x2;
                  }
//                }
            }
            else {
            //    printf("\nIgnore");
                  cycles=cycles+1;
                  ignore=ignore+4;
            }
```

```
                }
                cycles=cycles+8;
        }
        }
        timee=cycles/clk;
        hits=(hits*100)/(w*l*4);
        ignore=(ignore*100)/(w*l*4);
        printf("\n%d\t%dx%d\t%d\t%d\t%4.6f\t%4.6f\t%4.6f\n",bwid,l,w,pixe
ls,cycles,hits,ignore,timee);
        fprintf(f2,"%d,%dx%d,%d,%d,%4.6f,%4.6f,%4.6f",bwid,l,w,pixels,cyc
les,hits,ignore,timee*1000);
        fclose(f2);
}
```

References

[1] Ruigang Yang, Xinyu Huang, Subhasri Krishnan, Christopher Jaynes, "Toward the Light Field Display: Autostereoscopic Rendering via a Cluster of Projectors", Eurographics 2006. (For abstract see at:
http://vis.uky.edu/~gravity/Research/lddisplay/LightFieldDisplay.html
For paper see at:
http://www.vis.uky.edu/~xhuan4/LFDisplay/LFdisplay-short-sub.pdf)

[2] Ruigang Yang, Daniel R. Rudolf, Vijai Raghunathan, "Flexible Pixel Compositor for Plug-and-Play Multi-Projector Displays", PROCAMS 2007, Minneapolis, Minnesota .

[3] Subhasri Krishnan, "A Control Mechanism To The Anywhere Pixel Router", University of Kentucky 2007. (See at: http://lib.uky.edu/ETD/ukyelen2007t00594/Subhasri_Thesis.pdf)

[4] DDR-RAM datasheet from Micron Devices. (See at: http://download.micron.com/pdf/datasheets/dram/ddr/512MBDDRx4x8x16.pdf)

[5] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach", third edition [p390-p448].

[6] Virtex-4 FPGA datasheet from Xilinx. (See at: http://direct.xilinx.com/bvdocs/publications/ds112.pdf)

[7] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, "Computer Graphics Principles and Practice", Second Edition in C.

[8] Wikipedia Reference:
Interpolation: http://en.wikipedia.org/wiki/Interpolation
CPU cache: http://en.wikipedia.org/wiki/CPU_cache

[9] Tomas Akenin-Moller, Eric Haines, "Real Time Rendering", Second Edition [Chapter 5 on Texturing].

[10] Ziyad S. Hakura, Anoop Gupta, "The Design and Analysis of a Cache Architecture for Texture Mapping". [24th International Symposium on Computer Architecture 1997] (See at:
http://graphics.stanford.edu/papers/texture_cache/)

[11] Homan Igehy, Matthew Eldridge, Kekoa Proudfoot, "Prefetching in a Texture Cache Architecture". [Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware] (See at: http://graphics.stanford.edu/papers/texture_prefetch/)

[12] AMD Athlon 64 X2 Dual Processor Datasheet. (See at: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33425.pdf)

[13] Intel Pentium 4 Datasheets (See at: http://download.intel.com/design/Pentium4/datashts/31030802.pdf)

[14] Steven Przybylski, Mark Horowitz, John Hennessy, "Performance Tradeoffs in Cache Design". Proceedings of the 15th Annual International Symposium on Computer architecture (See at: http://delivery.acm.org/10.1145/60000/52433/p290-przybylski.pdf?key1=52433&key2=0187849811&coll=portal&dl=ACM&CFID=34656754&CFTOKEN=82692607)

[15] Steven Przybylski, "Cache Design: A Performance-Directed Approach", Morgan Kaufmann Publications 1990.

VITA

Vijai Raghunathan was born on June 1st 1984 in Madras, India. He received his Bachelor of Technology from SASTRA University in Tanjore, India. He was in the Dean's Merit list and received scholarship on all four years of his undergraduate study from 2001-2005. He has completed many projects successfully as an intern at TCS (Tata Consultancy Services, India), BSNL Telephones, India and NASA (Goddard Space Flight Center, Greenbelt MD). He joined the University of Kentucky on August 2005.