2007

AN EFFECTIVE CACHE FOR THE ANYWHERE PIXEL ROUTER

Vijai Raghunathan  
*University of Kentucky*

---

**Recommended Citation**  
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 1.5: 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. ........................................... 36
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........................................... 37
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........................................... 38
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.

![Diagram of Reverse Mapping Process](image)

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.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Pixel Clk, in MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>640x480, 60Hz</td>
<td>25.175</td>
</tr>
<tr>
<td>800x600, 60Hz</td>
<td>40.000</td>
</tr>
<tr>
<td>1024x768, 60Hz</td>
<td>66.000</td>
</tr>
<tr>
<td>1280x1024, 60Hz</td>
<td>108.000</td>
</tr>
</tbody>
</table>

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.
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.

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 \((x_1, y_1)\), \((x_2, y_2)\) and \((x_3, y_3)\). 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

<table>
<thead>
<tr>
<th>BITS</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>IGNORE</td>
</tr>
<tr>
<td>30-23</td>
<td>ALPHA</td>
</tr>
<tr>
<td>22-21</td>
<td>Channel ID</td>
</tr>
<tr>
<td>20-0</td>
<td>Memory Address</td>
</tr>
<tr>
<td></td>
<td>(10-0 Column)</td>
</tr>
<tr>
<td></td>
<td>(20-11 Row)</td>
</tr>
</tbody>
</table>

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

![Diagram of Memory Hardware Structure for APR](image)

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 \( w/2 + t_{\text{ras}} + t_{\text{cas}} + 3 \) cycles, where ‘\( w \)’ is the width of each cache line. For the APR, \( t_{\text{ras}} \) is 7 cycles, and \( t_{\text{cas}} \) 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 “$l \times w$”, 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.

![Caching Function Diagram](image)

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.

![Cache Access Times (45 degrees)](image)

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.

![Input Frame in Memory](image)

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.
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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>224.....239</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>240.....255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 5.3: Division into Memory 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.
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.

Figure 5.6: Blocks Labeling
Any memory block can be identified if the ‘bi’ and ‘bj’ values (shown in Figure 5.6) are given. For example if bi=2 and bj=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
```
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:

```c
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;
```
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.

![Block Size 16x16 (Hit Rate)](image)

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.

![SDRAM vs Cache Access Times](image)

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.

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](image)

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
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.
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
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 fixed 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 \( t_{\text{cas}} + t_{\text{ras}} + t_{\text{cas}} \), which is 11 cycles. It is only one \( t_{\text{cas}} \) 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 \( t_{\text{ras}} + t_{\text{cas}} + t_{\text{ras}} + t_{\text{cas}} \), 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 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.

![Bilinear Interpolation SDRAM vs Cache](image)

**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

Start

Get the desired LUT values. K=0.

B

Is k<3

Yes

A

Choose a line in the input frame. k=k+1

A

Starting from the first line in the LUT, travel left to right in each line. Locate pixels belonging to the chosen input frame.

Yes

Order the blocks and store in a file.

No

All slopes within an accepted range of values from the average?

Yes

Order the blocks and store in a file.

No

Divide the LUT into smaller blocks.

Can LUT be divided into smaller memory block spaces?

Yes

No

Store a default block order in the file

Stop
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.
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 bilinear interpolation.

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

<table>
<thead>
<tr>
<th>Angle</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Same Row (%)</th>
<th>Change in Row (%)</th>
<th>Ignore (%)</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>786432</td>
<td>797945</td>
<td>99.902473</td>
<td>0.097529</td>
<td>0</td>
<td>5.999587</td>
</tr>
<tr>
<td>1</td>
<td>786432</td>
<td>894090</td>
<td>97.317505</td>
<td>1.844025</td>
<td>0.83847</td>
<td>6.722481</td>
</tr>
<tr>
<td>2</td>
<td>786432</td>
<td>989094</td>
<td>94.726814</td>
<td>3.569794</td>
<td>1.70339</td>
<td>7.436797</td>
</tr>
<tr>
<td>3</td>
<td>786432</td>
<td>1078820</td>
<td>92.260361</td>
<td>5.199687</td>
<td>2.539953</td>
<td>8.111428</td>
</tr>
<tr>
<td>4</td>
<td>786432</td>
<td>1168287</td>
<td>89.826584</td>
<td>6.824875</td>
<td>3.34851</td>
<td>8.784113</td>
</tr>
<tr>
<td>5</td>
<td>786432</td>
<td>1257222</td>
<td>87.429298</td>
<td>8.440399</td>
<td>4.1303</td>
<td>9.452797</td>
</tr>
<tr>
<td>6</td>
<td>786432</td>
<td>1344379</td>
<td>85.090385</td>
<td>10.023625</td>
<td>4.885992</td>
<td>10.108112</td>
</tr>
<tr>
<td>7</td>
<td>786432</td>
<td>1429905</td>
<td>82.806648</td>
<td>11.577225</td>
<td>5.616125</td>
<td>10.751165</td>
</tr>
<tr>
<td>8</td>
<td>786432</td>
<td>1514367</td>
<td>80.567039</td>
<td>13.111496</td>
<td>6.321462</td>
<td>11.386218</td>
</tr>
<tr>
<td>9</td>
<td>786432</td>
<td>1597506</td>
<td>78.374992</td>
<td>14.621735</td>
<td>7.003275</td>
<td>12.011323</td>
</tr>
<tr>
<td>10</td>
<td>786432</td>
<td>1679259</td>
<td>76.232277</td>
<td>16.106796</td>
<td>7.660929</td>
<td>12.626007</td>
</tr>
<tr>
<td>11</td>
<td>786432</td>
<td>1759710</td>
<td>74.134575</td>
<td>17.568207</td>
<td>8.297221</td>
<td>13.230902</td>
</tr>
<tr>
<td>12</td>
<td>786432</td>
<td>1838989</td>
<td>72.083282</td>
<td>19.006601</td>
<td>9.006601</td>
<td>13.826271</td>
</tr>
<tr>
<td>13</td>
<td>786432</td>
<td>1916930</td>
<td>69.757039</td>
<td>20.424143</td>
<td>9.501393</td>
<td>14.413008</td>
</tr>
<tr>
<td>14</td>
<td>786432</td>
<td>1993748</td>
<td>66.110786</td>
<td>21.819559</td>
<td>10.06966</td>
<td>14.990587</td>
</tr>
<tr>
<td>15</td>
<td>786432</td>
<td>2069467</td>
<td>62.45219</td>
<td>23.19501</td>
<td>10.62039</td>
<td>15.559902</td>
</tr>
<tr>
<td>16</td>
<td>786432</td>
<td>2143996</td>
<td>60.643387</td>
<td>24.548849</td>
<td>11.15214</td>
<td>16.12027</td>
</tr>
<tr>
<td>17</td>
<td>786432</td>
<td>2217552</td>
<td>58.873749</td>
<td>25.88501</td>
<td>11.6628</td>
<td>16.673233</td>
</tr>
<tr>
<td>18</td>
<td>786432</td>
<td>2290065</td>
<td>56.643387</td>
<td>27.202225</td>
<td>12.15439</td>
<td>17.218534</td>
</tr>
<tr>
<td>19</td>
<td>786432</td>
<td>2361451</td>
<td>54.873749</td>
<td>28.498968</td>
<td>12.62728</td>
<td>17.75527</td>
</tr>
<tr>
<td>20</td>
<td>786432</td>
<td>2431948</td>
<td>52.138317</td>
<td>29.779562</td>
<td>13.08212</td>
<td>18.285323</td>
</tr>
<tr>
<td>21</td>
<td>786432</td>
<td>2501451</td>
<td>49.43874</td>
<td>31.042099</td>
<td>13.51916</td>
<td>18.807903</td>
</tr>
<tr>
<td>22</td>
<td>786432</td>
<td>2569988</td>
<td>46.773754</td>
<td>32.28709</td>
<td>13.93916</td>
<td>19.323219</td>
</tr>
<tr>
<td>23</td>
<td>786432</td>
<td>2637552</td>
<td>44.144367</td>
<td>33.514404</td>
<td>14.34123</td>
<td>19.831218</td>
</tr>
<tr>
<td>24</td>
<td>786432</td>
<td>2704360</td>
<td>41.544458</td>
<td>34.727985</td>
<td>14.72753</td>
<td>20.333534</td>
</tr>
<tr>
<td>25</td>
<td>786432</td>
<td>2770195</td>
<td>39.978043</td>
<td>35.923893</td>
<td>15.09806</td>
<td>20.82534</td>
</tr>
<tr>
<td>26</td>
<td>786432</td>
<td>2835176</td>
<td>38.444279</td>
<td>37.10429</td>
<td>15.45143</td>
<td>21.317113</td>
</tr>
<tr>
<td>27</td>
<td>786432</td>
<td>2899317</td>
<td>36.940525</td>
<td>38.269424</td>
<td>15.79005</td>
<td>21.799376</td>
</tr>
<tr>
<td>28</td>
<td>786432</td>
<td>2962709</td>
<td>35.465893</td>
<td>39.420956</td>
<td>16.11315</td>
<td>22.276007</td>
</tr>
<tr>
<td>29</td>
<td>786432</td>
<td>3025177</td>
<td>34.0233</td>
<td>40.555698</td>
<td>16.421</td>
<td>22.745691</td>
</tr>
<tr>
<td>30</td>
<td>786432</td>
<td>3086959</td>
<td>41.60741</td>
<td>41.677982</td>
<td>16.71461</td>
<td>23.210218</td>
</tr>
<tr>
<td>31</td>
<td>786432</td>
<td>3147908</td>
<td>40.220898</td>
<td>42.785137</td>
<td>16.99397</td>
<td>23.668481</td>
</tr>
</tbody>
</table>
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
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)

<table>
<thead>
<tr>
<th>Angle</th>
<th>Cachetype</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate</th>
<th>Ignore</th>
<th>Time(millisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>8x8</td>
<td>786432</td>
<td>7748976</td>
<td>21.48984</td>
<td>19.54002</td>
<td>58.26298</td>
</tr>
<tr>
<td>45</td>
<td>8x16</td>
<td>786432</td>
<td>9448710</td>
<td>22.52922</td>
<td>19.54002</td>
<td>71.04293</td>
</tr>
<tr>
<td>45</td>
<td>8x32</td>
<td>786432</td>
<td>12982833</td>
<td>23.04993</td>
<td>19.54002</td>
<td>97.61529</td>
</tr>
<tr>
<td>45</td>
<td>8x64</td>
<td>786432</td>
<td>20118410</td>
<td>23.31098</td>
<td>19.54002</td>
<td>151.2662</td>
</tr>
<tr>
<td>45</td>
<td>16x8</td>
<td>786432</td>
<td>7748976</td>
<td>21.48984</td>
<td>19.54002</td>
<td>58.26298</td>
</tr>
<tr>
<td>45</td>
<td>16x16</td>
<td>786432</td>
<td>9448710</td>
<td>22.52922</td>
<td>19.54002</td>
<td>71.04293</td>
</tr>
<tr>
<td>45</td>
<td>16x32</td>
<td>786432</td>
<td>12982833</td>
<td>23.04993</td>
<td>19.54002</td>
<td>97.61529</td>
</tr>
<tr>
<td>45</td>
<td>16x64</td>
<td>786432</td>
<td>20118410</td>
<td>23.31098</td>
<td>19.54002</td>
<td>151.2662</td>
</tr>
<tr>
<td>45</td>
<td>32x8</td>
<td>786432</td>
<td>7748976</td>
<td>21.48984</td>
<td>19.54002</td>
<td>58.26298</td>
</tr>
<tr>
<td>45</td>
<td>32x16</td>
<td>786432</td>
<td>9448710</td>
<td>22.52922</td>
<td>19.54002</td>
<td>71.04293</td>
</tr>
<tr>
<td>45</td>
<td>32x32</td>
<td>786432</td>
<td>12982833</td>
<td>23.04993</td>
<td>19.54002</td>
<td>97.61529</td>
</tr>
<tr>
<td>45</td>
<td>32x64</td>
<td>786432</td>
<td>20118410</td>
<td>23.31098</td>
<td>19.54002</td>
<td>151.2662</td>
</tr>
<tr>
<td>45</td>
<td>64x8</td>
<td>786432</td>
<td>7748976</td>
<td>21.48984</td>
<td>19.54002</td>
<td>58.26298</td>
</tr>
<tr>
<td>45</td>
<td>64x16</td>
<td>786432</td>
<td>9448710</td>
<td>22.52922</td>
<td>19.54002</td>
<td>71.04293</td>
</tr>
<tr>
<td>45</td>
<td>64x32</td>
<td>786432</td>
<td>12982833</td>
<td>23.04993</td>
<td>19.54002</td>
<td>97.61529</td>
</tr>
</tbody>
</table>
Table A.3: Cache Access Times (0 degrees)

<table>
<thead>
<tr>
<th>Angle</th>
<th>Cachetype</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate (%)</th>
<th>Ignore (%)</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8x8</td>
<td>786432</td>
<td>2267136</td>
<td>87.5</td>
<td>0</td>
<td>17.04614</td>
</tr>
<tr>
<td>0</td>
<td>8x16</td>
<td>786432</td>
<td>1726464</td>
<td>93.75</td>
<td>0</td>
<td>12.98093</td>
</tr>
<tr>
<td>0</td>
<td>8x32</td>
<td>786432</td>
<td>1456128</td>
<td>96.875</td>
<td>0</td>
<td>10.94833</td>
</tr>
<tr>
<td>0</td>
<td>8x64</td>
<td>786432</td>
<td>1320960</td>
<td>98.4375</td>
<td>0</td>
<td>9.93203</td>
</tr>
<tr>
<td>0</td>
<td>16x8</td>
<td>786432</td>
<td>2267136</td>
<td>87.5</td>
<td>0</td>
<td>17.04614</td>
</tr>
<tr>
<td>0</td>
<td>16x16</td>
<td>786432</td>
<td>1726464</td>
<td>93.75</td>
<td>0</td>
<td>12.98093</td>
</tr>
<tr>
<td>0</td>
<td>16x32</td>
<td>786432</td>
<td>1456128</td>
<td>96.875</td>
<td>0</td>
<td>10.94833</td>
</tr>
<tr>
<td>0</td>
<td>16x64</td>
<td>786432</td>
<td>1320960</td>
<td>98.4375</td>
<td>0</td>
<td>9.93203</td>
</tr>
<tr>
<td>0</td>
<td>32x8</td>
<td>786432</td>
<td>2267136</td>
<td>87.5</td>
<td>0</td>
<td>17.04614</td>
</tr>
<tr>
<td>0</td>
<td>32x16</td>
<td>786432</td>
<td>1726464</td>
<td>93.75</td>
<td>0</td>
<td>12.98093</td>
</tr>
<tr>
<td>0</td>
<td>32x32</td>
<td>786432</td>
<td>1456128</td>
<td>96.875</td>
<td>0</td>
<td>10.94833</td>
</tr>
<tr>
<td>0</td>
<td>32x64</td>
<td>786432</td>
<td>1320960</td>
<td>98.4375</td>
<td>0</td>
<td>9.93203</td>
</tr>
<tr>
<td>0</td>
<td>64x8</td>
<td>786432</td>
<td>2267136</td>
<td>87.5</td>
<td>0</td>
<td>17.04614</td>
</tr>
<tr>
<td>0</td>
<td>64x16</td>
<td>786432</td>
<td>1726464</td>
<td>93.75</td>
<td>0</td>
<td>12.98093</td>
</tr>
<tr>
<td>0</td>
<td>64x32</td>
<td>786432</td>
<td>1456128</td>
<td>96.875</td>
<td>0</td>
<td>10.94833</td>
</tr>
<tr>
<td>0</td>
<td>64x64</td>
<td>786432</td>
<td>1320960</td>
<td>98.4375</td>
<td>0</td>
<td>9.93203</td>
</tr>
</tbody>
</table>

Table A.4: Cache with Blocks Simulation

<table>
<thead>
<tr>
<th>Angle</th>
<th>Blocksize</th>
<th>Cachetype</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate (%)</th>
<th>Ignore (%)</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1720845</td>
<td>76.52232</td>
<td>19.54002</td>
<td>12.93868</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x64</td>
<td>786432</td>
<td>1734158</td>
<td>77.94813</td>
<td>19.54002</td>
<td>13.03878</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x64</td>
<td>786432</td>
<td>1852801</td>
<td>77.88798</td>
<td>19.54002</td>
<td>13.93084</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x16</td>
<td>786432</td>
<td>1861184</td>
<td>73.92515</td>
<td>19.54002</td>
<td>13.99387</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x32</td>
<td>786432</td>
<td>1997781</td>
<td>75.68105</td>
<td>19.54002</td>
<td>15.02091</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x32</td>
<td>786432</td>
<td>2031720</td>
<td>75.52122</td>
<td>19.54002</td>
<td>15.27609</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x64</td>
<td>786432</td>
<td>2049237</td>
<td>77.88849</td>
<td>19.54002</td>
<td>15.4078</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x64</td>
<td>786432</td>
<td>2049237</td>
<td>77.88849</td>
<td>19.54002</td>
<td>15.4078</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x16</td>
<td>786432</td>
<td>2145650</td>
<td>72.67927</td>
<td>19.54002</td>
<td>16.13271</td>
</tr>
</tbody>
</table>

70
<table>
<thead>
<tr>
<th>45</th>
<th>32</th>
<th>64x16</th>
<th>786432</th>
<th>2145650</th>
<th>72.67927</th>
<th>19.54002</th>
<th>16.13271</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>32</td>
<td>32x64</td>
<td>786432</td>
<td>2166056</td>
<td>76.96165</td>
<td>19.54002</td>
<td>16.28614</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x8</td>
<td>786432</td>
<td>2242686</td>
<td>68.94849</td>
<td>19.54002</td>
<td>16.8623</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x32</td>
<td>786432</td>
<td>2244312</td>
<td>75.44594</td>
<td>19.54002</td>
<td>16.87453</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x32</td>
<td>786432</td>
<td>2244312</td>
<td>75.44594</td>
<td>19.54002</td>
<td>16.87453</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>16x64</td>
<td>786432</td>
<td>2429553</td>
<td>77.92664</td>
<td>19.54002</td>
<td>18.26732</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>32x64</td>
<td>786432</td>
<td>2429553</td>
<td>77.92664</td>
<td>19.54002</td>
<td>18.26732</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x64</td>
<td>786432</td>
<td>2429553</td>
<td>77.92664</td>
<td>19.54002</td>
<td>18.26732</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x8</td>
<td>786432</td>
<td>2498910</td>
<td>67.60979</td>
<td>19.54002</td>
<td>18.7888</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x8</td>
<td>786432</td>
<td>2498910</td>
<td>67.60979</td>
<td>19.54002</td>
<td>18.7888</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x16</td>
<td>786432</td>
<td>2581582</td>
<td>71.0776</td>
<td>19.54002</td>
<td>19.41039</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x16</td>
<td>786432</td>
<td>2581582</td>
<td>71.0776</td>
<td>19.54002</td>
<td>19.41039</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>16x32</td>
<td>786432</td>
<td>2616939</td>
<td>73.69105</td>
<td>19.54002</td>
<td>19.67623</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>16x16</td>
<td>786432</td>
<td>2618233</td>
<td>70.83231</td>
<td>19.54002</td>
<td>19.68596</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>16x32</td>
<td>786432</td>
<td>2622489</td>
<td>75.51676</td>
<td>19.54002</td>
<td>19.71796</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>32x32</td>
<td>786432</td>
<td>2622489</td>
<td>75.51676</td>
<td>19.54002</td>
<td>19.71796</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x32</td>
<td>786432</td>
<td>2622489</td>
<td>75.51676</td>
<td>19.54002</td>
<td>19.71796</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>16x8</td>
<td>786432</td>
<td>2963853</td>
<td>65.33508</td>
<td>19.54002</td>
<td>22.28461</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x8</td>
<td>786432</td>
<td>2963853</td>
<td>65.33508</td>
<td>19.54002</td>
<td>22.28461</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x8</td>
<td>786432</td>
<td>2963853</td>
<td>65.33508</td>
<td>19.54002</td>
<td>22.28461</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>16x64</td>
<td>786432</td>
<td>2993775</td>
<td>75.09537</td>
<td>19.54002</td>
<td>22.50959</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>16x16</td>
<td>786432</td>
<td>3030430</td>
<td>70.70528</td>
<td>19.54002</td>
<td>22.78519</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>32x16</td>
<td>786432</td>
<td>3030430</td>
<td>70.70528</td>
<td>19.54002</td>
<td>22.78519</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x16</td>
<td>786432</td>
<td>3030430</td>
<td>70.70528</td>
<td>19.54002</td>
<td>22.78519</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>8x16</td>
<td>786432</td>
<td>3492282</td>
<td>67.61437</td>
<td>19.54002</td>
<td>26.25776</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>16x8</td>
<td>786432</td>
<td>3711024</td>
<td>62.33457</td>
<td>19.54002</td>
<td>27.90244</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>32x8</td>
<td>786432</td>
<td>3711024</td>
<td>62.33457</td>
<td>19.54002</td>
<td>27.90244</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x8</td>
<td>786432</td>
<td>3711024</td>
<td>62.33457</td>
<td>19.54002</td>
<td>27.90244</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>8x32</td>
<td>786432</td>
<td>3712938</td>
<td>70.38129</td>
<td>19.54002</td>
<td>27.91683</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>8x8</td>
<td>786432</td>
<td>3740979</td>
<td>62.08064</td>
<td>19.54002</td>
<td>28.12766</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>8x64</td>
<td>786432</td>
<td>4511441</td>
<td>71.77023</td>
<td>19.54002</td>
<td>33.92061</td>
</tr>
<tr>
<td>45</td>
<td>128</td>
<td>64x8</td>
<td>786432</td>
<td>5235924</td>
<td>43.15783</td>
<td>19.54002</td>
<td>39.36785</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>32x8</td>
<td>786432</td>
<td>5520786</td>
<td>41.1597</td>
<td>19.54002</td>
<td>41.50967</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>16x8</td>
<td>786432</td>
<td>5878440</td>
<td>38.96116</td>
<td>19.54002</td>
<td>44.1988</td>
</tr>
<tr>
<td>45</td>
<td>128</td>
<td>64x16</td>
<td>786432</td>
<td>5990626</td>
<td>45.96011</td>
<td>19.54002</td>
<td>45.0423</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>32x16</td>
<td>786432</td>
<td>6360422</td>
<td>43.81421</td>
<td>19.54002</td>
<td>47.82272</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>8x8</td>
<td>786432</td>
<td>6464598</td>
<td>35.6589</td>
<td>19.54002</td>
<td>48.606</td>
</tr>
</tbody>
</table>

71
<table>
<thead>
<tr>
<th>Width</th>
<th>Height</th>
<th>Block Size</th>
<th>Cache Size</th>
<th>Access Time</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x16</td>
<td>16x8</td>
<td>786432</td>
<td>6812620</td>
<td>41.4458</td>
<td>19.54002</td>
<td>51.22271</td>
</tr>
<tr>
<td>16x16</td>
<td>8x16</td>
<td>786432</td>
<td>7541855</td>
<td>37.88121</td>
<td>19.54002</td>
<td>56.70558</td>
</tr>
<tr>
<td>32x8</td>
<td>32x8</td>
<td>786432</td>
<td>7674234</td>
<td>22.48802</td>
<td>19.54002</td>
<td>57.70101</td>
</tr>
<tr>
<td>16x8</td>
<td>16x8</td>
<td>786432</td>
<td>7782174</td>
<td>21.573</td>
<td>19.54002</td>
<td>58.51259</td>
</tr>
<tr>
<td>8x8</td>
<td>8x8</td>
<td>786432</td>
<td>7803589</td>
<td>21.39142</td>
<td>19.54002</td>
<td>59.12136</td>
</tr>
<tr>
<td>16x8</td>
<td>16x8</td>
<td>786432</td>
<td>7976400</td>
<td>21.17653</td>
<td>19.54002</td>
<td>59.97293</td>
</tr>
<tr>
<td>128x32</td>
<td>128x32</td>
<td>786432</td>
<td>8381340</td>
<td>45.1547</td>
<td>19.54002</td>
<td>63.01759</td>
</tr>
<tr>
<td>64x32</td>
<td>64x32</td>
<td>786432</td>
<td>9002823</td>
<td>42.69078</td>
<td>19.54002</td>
<td>67.6904</td>
</tr>
<tr>
<td>32x8</td>
<td>32x8</td>
<td>786432</td>
<td>9498196</td>
<td>23.34455</td>
<td>19.54002</td>
<td>71.95301</td>
</tr>
<tr>
<td>8x8</td>
<td>8x8</td>
<td>786432</td>
<td>9686655</td>
<td>22.21146</td>
<td>19.54002</td>
<td>72.832</td>
</tr>
<tr>
<td>16x16</td>
<td>16x16</td>
<td>786432</td>
<td>9982485</td>
<td>39.000299</td>
<td>19.54002</td>
<td>75.05628</td>
</tr>
<tr>
<td>64x64</td>
<td>64x64</td>
<td>786432</td>
<td>10789834</td>
<td>48.06684</td>
<td>19.54002</td>
<td>86.4537</td>
</tr>
<tr>
<td>32x64</td>
<td>32x64</td>
<td>786432</td>
<td>12600043</td>
<td>45.8163</td>
<td>19.54002</td>
<td>94.73717</td>
</tr>
<tr>
<td>128x64</td>
<td>128x64</td>
<td>786432</td>
<td>12786378</td>
<td>24.17768</td>
<td>19.54002</td>
<td>96.13818</td>
</tr>
<tr>
<td>16x32</td>
<td>16x32</td>
<td>786432</td>
<td>13005591</td>
<td>23.14529</td>
<td>19.54002</td>
<td>97.7864</td>
</tr>
<tr>
<td>64x32</td>
<td>16x32</td>
<td>786432</td>
<td>13012434</td>
<td>23.34455</td>
<td>19.54002</td>
<td>97.83785</td>
</tr>
<tr>
<td>128x32</td>
<td>128x32</td>
<td>786432</td>
<td>13048359</td>
<td>22.9438</td>
<td>19.54002</td>
<td>98.10796</td>
</tr>
<tr>
<td>64x64</td>
<td>32x64</td>
<td>786432</td>
<td>13116654</td>
<td>22.85372</td>
<td>19.54002</td>
<td>98.62146</td>
</tr>
<tr>
<td>32x64</td>
<td>32x64</td>
<td>786432</td>
<td>13240446</td>
<td>22.73369</td>
<td>19.54002</td>
<td>99.55223</td>
</tr>
<tr>
<td>32x64</td>
<td>16x64</td>
<td>786432</td>
<td>13543426</td>
<td>43.31729</td>
<td>19.54002</td>
<td>101.8303</td>
</tr>
<tr>
<td>16x64</td>
<td>16x64</td>
<td>786432</td>
<td>15011802</td>
<td>39.5565</td>
<td>19.54002</td>
<td>112.8707</td>
</tr>
<tr>
<td>128x64</td>
<td>128x64</td>
<td>786432</td>
<td>15011802</td>
<td>39.5565</td>
<td>19.5402</td>
<td>112.8707</td>
</tr>
<tr>
<td>64x64</td>
<td>32x64</td>
<td>786432</td>
<td>19774547</td>
<td>24.45501</td>
<td>19.54002</td>
<td>148.6808</td>
</tr>
<tr>
<td>128x64</td>
<td>128x64</td>
<td>786432</td>
<td>20109090</td>
<td>23.61107</td>
<td>19.54002</td>
<td>151.1962</td>
</tr>
<tr>
<td>64x64</td>
<td>16x64</td>
<td>786432</td>
<td>20129770</td>
<td>23.40457</td>
<td>19.54002</td>
<td>151.3517</td>
</tr>
<tr>
<td>128x64</td>
<td>128x64</td>
<td>786432</td>
<td>20198097</td>
<td>23.20252</td>
<td>19.54002</td>
<td>151.8654</td>
</tr>
<tr>
<td>64x64</td>
<td>8x64</td>
<td>786432</td>
<td>20277435</td>
<td>23.11325</td>
<td>19.54002</td>
<td>152.4619</td>
</tr>
<tr>
<td>32x64</td>
<td>8x64</td>
<td>786432</td>
<td>20416417</td>
<td>22.99296</td>
<td>19.54002</td>
<td>153.5069</td>
</tr>
</tbody>
</table>

Table A.5: 64x32 Cache with Block 64
<table>
<thead>
<tr>
<th>Angle</th>
<th>Blocksize</th>
<th>Cachetype</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate (%)</th>
<th>Ignore (%)</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1548288</td>
<td>96.875</td>
<td>0</td>
<td>11.64126</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1834245</td>
<td>94.68981</td>
<td>0.83847</td>
<td>13.79132</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1843425</td>
<td>93.78166</td>
<td>1.70339</td>
<td>13.86034</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1844532</td>
<td>92.93989</td>
<td>2.539953</td>
<td>13.86866</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1834749</td>
<td>92.13499</td>
<td>3.348541</td>
<td>13.86278</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1834380</td>
<td>91.39735</td>
<td>4.1303</td>
<td>13.79233</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1829952</td>
<td>90.66251</td>
<td>4.885992</td>
<td>13.75904</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1826955</td>
<td>89.9465</td>
<td>5.616125</td>
<td>13.7365</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1823445</td>
<td>89.25768</td>
<td>6.321462</td>
<td>13.71011</td>
</tr>
<tr>
<td>9</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1843749</td>
<td>92.34999</td>
<td>3.348541</td>
<td>13.86278</td>
</tr>
<tr>
<td>10</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1815507</td>
<td>87.95561</td>
<td>7.660929</td>
<td>13.65043</td>
</tr>
<tr>
<td>11</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1811403</td>
<td>87.33864</td>
<td>8.297221</td>
<td>13.61957</td>
</tr>
<tr>
<td>12</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1806516</td>
<td>86.74876</td>
<td>8.910115</td>
<td>13.58283</td>
</tr>
<tr>
<td>13</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1802169</td>
<td>86.17796</td>
<td>9.501393</td>
<td>13.55014</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1799064</td>
<td>85.62431</td>
<td>10.06966</td>
<td>13.5268</td>
</tr>
<tr>
<td>15</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1793745</td>
<td>85.09699</td>
<td>10.62203</td>
<td>13.4868</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1791369</td>
<td>84.57807</td>
<td>11.15216</td>
<td>13.46894</td>
</tr>
<tr>
<td>17</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1789904</td>
<td>84.33864</td>
<td>11.66096</td>
<td>13.42143</td>
</tr>
<tr>
<td>18</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1782432</td>
<td>83.61791</td>
<td>12.15439</td>
<td>13.40174</td>
</tr>
<tr>
<td>19</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1779408</td>
<td>83.15926</td>
<td>12.62728</td>
<td>13.37901</td>
</tr>
<tr>
<td>20</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1775277</td>
<td>82.72387</td>
<td>13.08212</td>
<td>13.34795</td>
</tr>
<tr>
<td>21</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1772442</td>
<td>82.30019</td>
<td>13.51916</td>
<td>13.32663</td>
</tr>
<tr>
<td>22</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1769877</td>
<td>81.89227</td>
<td>13.93916</td>
<td>13.30735</td>
</tr>
<tr>
<td>23</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1766502</td>
<td>81.5061</td>
<td>14.34123</td>
<td>13.28197</td>
</tr>
<tr>
<td>24</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1763721</td>
<td>81.13289</td>
<td>14.72753</td>
<td>13.26106</td>
</tr>
<tr>
<td>25</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1760319</td>
<td>80.77837</td>
<td>15.09806</td>
<td>13.23548</td>
</tr>
<tr>
<td>26</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1757457</td>
<td>80.43848</td>
<td>15.45143</td>
<td>13.21396</td>
</tr>
<tr>
<td>27</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1753542</td>
<td>80.11831</td>
<td>15.79005</td>
<td>13.18453</td>
</tr>
<tr>
<td>28</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1750356</td>
<td>79.8102</td>
<td>16.11315</td>
<td>13.16057</td>
</tr>
<tr>
<td>29</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1747170</td>
<td>79.51737</td>
<td>16.421</td>
<td>13.13662</td>
</tr>
<tr>
<td>30</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1746873</td>
<td>79.22516</td>
<td>16.71461</td>
<td>13.13438</td>
</tr>
<tr>
<td>31</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1746981</td>
<td>78.94529</td>
<td>16.99397</td>
<td>13.1352</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1744416</td>
<td>78.69059</td>
<td>17.26074</td>
<td>13.11591</td>
</tr>
<tr>
<td>33</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>1741176</td>
<td>78.45637</td>
<td>17.51022</td>
<td>13.09155</td>
</tr>
<tr>
<td>Angle</td>
<td>Blocksize</td>
<td>Cachetype</td>
<td>Pixels</td>
<td>Cycles</td>
<td>Hit Rate (%)</td>
<td>Ignore (%)</td>
<td>Time (milli-seconds)</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>--------------</td>
<td>------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x8</td>
<td>786432</td>
<td>1550295</td>
<td>75.6513</td>
<td>19.54002</td>
<td>11.65635</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x16</td>
<td>786432</td>
<td>1701563</td>
<td>75.6513</td>
<td>19.54002</td>
<td>12.79371</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x8</td>
<td>786432</td>
<td>1771098</td>
<td>75.44619</td>
<td>19.54002</td>
<td>13.31653</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x16</td>
<td>786432</td>
<td>1928818</td>
<td>75.44619</td>
<td>19.54002</td>
<td>14.50239</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x32</td>
<td>786432</td>
<td>2004099</td>
<td>75.6513</td>
<td>19.54002</td>
<td>15.06814</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x8</td>
<td>786432</td>
<td>2007096</td>
<td>70.94561</td>
<td>19.54002</td>
<td>15.09095</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x8</td>
<td>786432</td>
<td>2155869</td>
<td>75.51778</td>
<td>19.54002</td>
<td>16.20954</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x32</td>
<td>786432</td>
<td>2244258</td>
<td>75.44619</td>
<td>19.54002</td>
<td>16.87412</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x8</td>
<td>786432</td>
<td>2297523</td>
<td>70.98364</td>
<td>19.54002</td>
<td>17.27461</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x16</td>
<td>786432</td>
<td>2306392</td>
<td>70.94561</td>
<td>19.54002</td>
<td>17.34129</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x16</td>
<td>786432</td>
<td>2311337</td>
<td>75.51778</td>
<td>19.54002</td>
<td>17.37847</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>64x64</td>
<td>786432</td>
<td>2309350</td>
<td>76.2422</td>
<td>19.54002</td>
<td>18.11541</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x8</td>
<td>786432</td>
<td>2500230</td>
<td>67.5986</td>
<td>19.54002</td>
<td>18.79872</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>32x16</td>
<td>786432</td>
<td>2595623</td>
<td>70.98364</td>
<td>19.54002</td>
<td>19.51596</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>64x32</td>
<td>786432</td>
<td>2622273</td>
<td>75.51778</td>
<td>19.54002</td>
<td>19.71634</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>32x8</td>
<td>786432</td>
<td>2723529</td>
<td>70.70567</td>
<td>19.54002</td>
<td>20.47766</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x32</td>
<td>786432</td>
<td>2828355</td>
<td>71.76946</td>
<td>19.54002</td>
<td>21.26583</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>64x64</td>
<td>786432</td>
<td>2875138</td>
<td>75.44619</td>
<td>19.54002</td>
<td>21.61758</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x64</td>
<td>786432</td>
<td>2889482</td>
<td>74.53169</td>
<td>19.54002</td>
<td>21.72543</td>
</tr>
<tr>
<td>45</td>
<td>32</td>
<td>32x16</td>
<td>786432</td>
<td>2904814</td>
<td>67.5986</td>
<td>19.54002</td>
<td>21.84071</td>
</tr>
<tr>
<td>45</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>2904984</td>
<td>70.94561</td>
<td>19.54002</td>
<td>21.84198</td>
</tr>
</tbody>
</table>

Table A.6: 2-way Set Associative Cache
| 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 | 21.62956 | 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 |
Table A.6 lists all the access times and hit rates for various cache sizes and block 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

<table>
<thead>
<tr>
<th>Angle</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Ignore (%)</th>
<th>Time(milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>786432</td>
<td>14149632</td>
<td>0</td>
<td>106.388211</td>
</tr>
<tr>
<td>1</td>
<td>786432</td>
<td>14042036</td>
<td>0.896708</td>
<td>105.57922</td>
</tr>
<tr>
<td>2</td>
<td>786432</td>
<td>13926589</td>
<td>1.760229</td>
<td>104.711197</td>
</tr>
<tr>
<td>3</td>
<td>786432</td>
<td>13814780</td>
<td>2.596537</td>
<td>103.870526</td>
</tr>
<tr>
<td>4</td>
<td>786432</td>
<td>13706830</td>
<td>3.403981</td>
<td>103.058875</td>
</tr>
<tr>
<td>5</td>
<td>786432</td>
<td>13602365</td>
<td>4.185359</td>
<td>102.273419</td>
</tr>
<tr>
<td>6</td>
<td>786432</td>
<td>13501419</td>
<td>4.940414</td>
<td>101.514429</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>------------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>7</td>
<td>786432</td>
<td>13403822</td>
<td>5.67042</td>
<td>100.780614</td>
</tr>
<tr>
<td>8</td>
<td>786432</td>
<td>13309574</td>
<td>6.375376</td>
<td>100.071982</td>
</tr>
<tr>
<td>9</td>
<td>786432</td>
<td>13218505</td>
<td>7.056554</td>
<td>99.387258</td>
</tr>
<tr>
<td>10</td>
<td>786432</td>
<td>13130581</td>
<td>7.714208</td>
<td>98.726176</td>
</tr>
<tr>
<td>11</td>
<td>786432</td>
<td>13045683</td>
<td>8.349228</td>
<td>98.08784</td>
</tr>
<tr>
<td>12</td>
<td>786432</td>
<td>12963692</td>
<td>9.553401</td>
<td>97.471371</td>
</tr>
<tr>
<td>13</td>
<td>786432</td>
<td>12884693</td>
<td>10.121028</td>
<td>96.877389</td>
</tr>
<tr>
<td>14</td>
<td>786432</td>
<td>12808805</td>
<td>10.673141</td>
<td>96.306801</td>
</tr>
<tr>
<td>15</td>
<td>786432</td>
<td>12734991</td>
<td>11.204147</td>
<td>95.751815</td>
</tr>
<tr>
<td>16</td>
<td>786432</td>
<td>12663999</td>
<td>11.713791</td>
<td>95.21804</td>
</tr>
<tr>
<td>17</td>
<td>786432</td>
<td>12595863</td>
<td>12.204742</td>
<td>94.705738</td>
</tr>
<tr>
<td>18</td>
<td>786432</td>
<td>12530226</td>
<td>12.677765</td>
<td>94.212227</td>
</tr>
<tr>
<td>19</td>
<td>786432</td>
<td>12466986</td>
<td>13.13235</td>
<td>93.736738</td>
</tr>
<tr>
<td>20</td>
<td>786432</td>
<td>12406211</td>
<td>13.568624</td>
<td>93.279779</td>
</tr>
<tr>
<td>21</td>
<td>786432</td>
<td>12347884</td>
<td>14.477648</td>
<td>92.84123</td>
</tr>
<tr>
<td>22</td>
<td>786432</td>
<td>12291784</td>
<td>14.98824</td>
<td>92.414365</td>
</tr>
<tr>
<td>23</td>
<td>786432</td>
<td>12237911</td>
<td>15.46129</td>
<td>91.255501</td>
</tr>
<tr>
<td>24</td>
<td>786432</td>
<td>12136982</td>
<td>15.950641</td>
<td>90.89914</td>
</tr>
<tr>
<td>25</td>
<td>786432</td>
<td>12089586</td>
<td>16.469193</td>
<td>90.234995</td>
</tr>
<tr>
<td>26</td>
<td>786432</td>
<td>11960097</td>
<td>16.962707</td>
<td>89.925542</td>
</tr>
<tr>
<td>27</td>
<td>786432</td>
<td>11920844</td>
<td>17.462796</td>
<td>89.630403</td>
</tr>
<tr>
<td>28</td>
<td>786432</td>
<td>11883529</td>
<td>17.941906</td>
<td>89.349844</td>
</tr>
<tr>
<td>29</td>
<td>786432</td>
<td>11848424</td>
<td>18.404483</td>
<td>89.058892</td>
</tr>
<tr>
<td>30</td>
<td>786432</td>
<td>11814509</td>
<td>18.858161</td>
<td>88.830896</td>
</tr>
<tr>
<td>31</td>
<td>786432</td>
<td>11782719</td>
<td>19.309544</td>
<td>88.591874</td>
</tr>
<tr>
<td>32</td>
<td>786432</td>
<td>11752697</td>
<td>19.80502</td>
<td>88.366143</td>
</tr>
<tr>
<td>33</td>
<td>786432</td>
<td>11724324</td>
<td>20.232727</td>
<td>88.152811</td>
</tr>
<tr>
<td>34</td>
<td>786432</td>
<td>11697787</td>
<td>20.631219</td>
<td>87.953284</td>
</tr>
<tr>
<td>35</td>
<td>786432</td>
<td>11672665</td>
<td>21.01763</td>
<td>87.765902</td>
</tr>
<tr>
<td>36</td>
<td>786432</td>
<td>11649592</td>
<td>22.071708</td>
<td>87.590918</td>
</tr>
<tr>
<td>37</td>
<td>786432</td>
<td>11627849</td>
<td>22.534348</td>
<td>87.427437</td>
</tr>
<tr>
<td>38</td>
<td>786432</td>
<td>11607602</td>
<td>23.105783</td>
<td>87.2752</td>
</tr>
<tr>
<td>39</td>
<td>786432</td>
<td>11588936</td>
<td>23.645401</td>
<td>87.13486</td>
</tr>
<tr>
<td>40</td>
<td>786432</td>
<td>11571647</td>
<td>24.37472</td>
<td>87.004863</td>
</tr>
</tbody>
</table>

77
<table>
<thead>
<tr>
<th>Angle (degrees)</th>
<th>Block size</th>
<th>Block Type (l x w)</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate(%)</th>
<th>Access Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4928211</td>
<td>98.01709</td>
<td>37.05422</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5062248</td>
<td>96.99751</td>
<td>38.06201</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5193501</td>
<td>95.95404</td>
<td>39.04888</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5177490</td>
<td>95.11308</td>
<td>38.9285</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5164083</td>
<td>94.29782</td>
<td>38.82769</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5147151</td>
<td>93.51428</td>
<td>38.70038</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5124570</td>
<td>92.76419</td>
<td>38.5306</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5105346</td>
<td>92.03641</td>
<td>38.38606</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5088327</td>
<td>91.33151</td>
<td>38.2581</td>
</tr>
<tr>
<td>9</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5072106</td>
<td>90.64986</td>
<td>38.13614</td>
</tr>
<tr>
<td>10</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5056401</td>
<td>89.99243</td>
<td>38.01805</td>
</tr>
<tr>
<td>11</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5042064</td>
<td>89.35534</td>
<td>37.91026</td>
</tr>
<tr>
<td>12</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5025255</td>
<td>88.74521</td>
<td>37.78387</td>
</tr>
<tr>
<td>13</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>5010657</td>
<td>88.15469</td>
<td>37.67411</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4996197</td>
<td>87.58768</td>
<td>37.56539</td>
</tr>
<tr>
<td>15</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4982004</td>
<td>87.03667</td>
<td>37.45868</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4970064</td>
<td>86.50589</td>
<td>37.3689</td>
</tr>
<tr>
<td>17</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4959663</td>
<td>85.99329</td>
<td>37.2907</td>
</tr>
<tr>
<td>18</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4947363</td>
<td>85.50253</td>
<td>37.19822</td>
</tr>
<tr>
<td>19</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4934748</td>
<td>85.03135</td>
<td>37.10337</td>
</tr>
<tr>
<td>20</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4923504</td>
<td>84.57712</td>
<td>37.01883</td>
</tr>
<tr>
<td>21</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4911222</td>
<td>84.1424</td>
<td>36.92648</td>
</tr>
<tr>
<td>22</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4903581</td>
<td>83.71973</td>
<td>36.86903</td>
</tr>
<tr>
<td>23</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4894014</td>
<td>83.31776</td>
<td>36.7971</td>
</tr>
<tr>
<td>24</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4884468</td>
<td>82.93196</td>
<td>36.72532</td>
</tr>
<tr>
<td>25</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4877049</td>
<td>82.55988</td>
<td>36.66954</td>
</tr>
<tr>
<td>26</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4866849</td>
<td>82.20869</td>
<td>36.59285</td>
</tr>
<tr>
<td>27</td>
<td>64</td>
<td>64x32</td>
<td>786432</td>
<td>4858482</td>
<td>81.87052</td>
<td>36.52994</td>
</tr>
<tr>
<td>Angle</td>
<td>Pixels</td>
<td>Cycles</td>
<td>Blocks(milliseconds)</td>
<td>SDRAM, no blocks,no cache (milliseconds)</td>
<td>SDRAM, no cache, BlockSize 64 (milliseconds)</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>---------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>786432</td>
<td>1155065</td>
<td>8.684699</td>
<td>5.999587</td>
<td>11.64126</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>786432</td>
<td>1595890</td>
<td>11.99917</td>
<td>9.452797</td>
<td>13.79233</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>786432</td>
<td>3667848</td>
<td>27.5778</td>
<td>24.56711</td>
<td>13.0542</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>786432</td>
<td>4199407</td>
<td>31.57449</td>
<td>29.50932</td>
<td>12.93868</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate (%)</th>
<th>Ignore (%)</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>786432</td>
<td>1382400</td>
<td>72.705208</td>
<td>24.951044</td>
<td>10.39399</td>
</tr>
</tbody>
</table>

79
### Table A.11: SDRAM (Sample LUTs)

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Same Row Access (%)</th>
<th>Change In Row (%)</th>
<th>Ignore (%)</th>
<th>Access Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (90 degrees)</td>
<td>786432</td>
<td>4921344</td>
<td>0.048955</td>
<td>75</td>
<td>24.95104</td>
<td>37.00259</td>
</tr>
<tr>
<td>II</td>
<td>786432</td>
<td>1320726</td>
<td>82.686745</td>
<td>9.593964</td>
<td>7.719294</td>
<td>9.930271</td>
</tr>
<tr>
<td>III</td>
<td>786432</td>
<td>1696220</td>
<td>75.865807</td>
<td>16.414896</td>
<td>7.719294</td>
<td>12.75353</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>RAM Access Time (milliseconds)</th>
<th>Cache Design Access time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (90 degrees)</td>
<td>37.00259</td>
<td>10.39399</td>
</tr>
<tr>
<td>II</td>
<td>9.930271</td>
<td>6.491737</td>
</tr>
<tr>
<td>III</td>
<td>12.75353</td>
<td>6.247346</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Hit Rate (%)</th>
<th>Access Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (90 degrees)</td>
<td>786432</td>
<td>2582695</td>
<td>72.738899</td>
<td>19.41876</td>
</tr>
<tr>
<td>II</td>
<td>786432</td>
<td>2977696</td>
<td>89.603584</td>
<td>22.38869</td>
</tr>
<tr>
<td>III</td>
<td>786432</td>
<td>2953412</td>
<td>88.831619</td>
<td>22.20611</td>
</tr>
</tbody>
</table>

80
Table A.14: SDRAM (Sample LUTs, Bilinear Interpolation)

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>Pixels</th>
<th>Cycles</th>
<th>Access Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (90 degrees)</td>
<td>786432</td>
<td>10826129</td>
<td>81.3994638</td>
</tr>
<tr>
<td>II</td>
<td>786432</td>
<td>12916800</td>
<td>97.1187958</td>
</tr>
<tr>
<td>III</td>
<td>786432</td>
<td>13127255</td>
<td>98.701164</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Test LUT</th>
<th>RAM Access Time (milliseconds)</th>
<th>Cache Access Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (90 degrees)</td>
<td>81.39946</td>
<td>19.41876</td>
</tr>
<tr>
<td>II</td>
<td>97.1188</td>
<td>22.38869</td>
</tr>
<tr>
<td>III</td>
<td>98.70116</td>
<td>22.20611</td>
</tr>
</tbody>
</table>
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.

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

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

// hexadecimal 3
if (ppp[2] == '0') {
}
else if (ppp[2] == '1') {
}
else if (ppp[2] == '2') {
}
else if (ppp[2] == '3') {
}
else if (ppp[2] == '4') {
}
else if (ppp[2] == '5') {
}
else if (ppp[2] == '6') {
}
else if (ppp[2] == '7') {
}
else if (ppp[2] == '8') {
}
else if (ppp[2]=='9') {
}
}
}
}
}
}
}
else {
    hexerr = -1;
}

// hexadecimal 4
if (ppp[3]=='0') {
}
else if (ppp[3]=='1') {
}
else if (ppp[3]=='2') {
}
else if (ppp[3]=='3') {
}
else if (ppp[3]=='4') {
}
else if (ppp[3]=='5') {
}
else if (ppp[3]=='6') {
}
else if (ppp[3]=='7') {
}
else if (ppp[3]=='8') {
}
else if (ppp[3]=='9') {
}
}
}
}
}
}
}
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') {
}
else if (ppp[4]=='2') {
}
else if (ppp[4]=='3') {
}
else if (ppp[4]=='4') {
}
else if (ppp[4]=='5') {
}
else if (ppp[4]=='6') {
}
else if (ppp[4]=='7') {
}
else if (ppp[4]=='8') {
}
else if (ppp[4]=='9') {
}
}
}
```c
}  
}
}
}
}
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*1*4);
p_cir=(change_in_row*100)/(w*1*4);
ignore=(ignore*100)/(w*1*4);
printf("%d,%d,%4.6f,%4.6f,%4.6f,%4.6f\n",pixels,cycles,p_sr,p_cir,ignore,timee);
fprintf(f2,"%d,%d,%4.6f,%4.6f,%4.6f,%4.6f",pixels,cycles,p_sr,p_cir,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 \textit{cache} and \textit{caloit} functions contain the main logic for the cache simulations. All the simulations involving the cache (even without blocks) use these two functions.

```c
#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++) {

```

93
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') {
    }
    else if (ppp[0]=='1') {
    }
    else if (ppp[0]=='2') {
    }
    else if (ppp[0]=='3') {
    }
    else if (ppp[0]=='4') {
    }
    else if (ppp[0]=='5') {
    }
    else if (ppp[0]=='6') {
    }
    else if (ppp[0]=='7') {
    }
    else if (ppp[0]=='8') {
    }
    else if (ppp[0]=='9') {
    }
    else if (ppp[0]=='A' || ppp[0]=='a') {
    }
else if (ppp[0] == 'B' || ppp[0] == 'b') {
}
else if (ppp[0] == 'C' || ppp[0] == 'c') {
}
else if (ppp[0] == 'D' || ppp[0] == 'd') {
}
else if (ppp[0] == 'E' || ppp[0] == 'e') {
}
else if (ppp[0] == 'F' || ppp[0] == 'f') {
}
else {
    hexerr = -1;
}

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

// hexadecimal 3
if(ppp[2]=='0') {
}
else if (ppp[2]=='1') {

}
else if (ppp[2] == '2') {
}
else if (ppp[2] == '3') {
}
else if (ppp[2] == '4') {
}
else if (ppp[2] == '5') {
}
else if (ppp[2] == '6') {
}
else if (ppp[2] == '7') {
}
else if (ppp[2] == '8') {
}
else if (ppp[2] == '9') {
}
}
}
}
}
}
}
else {
    hexerr=-1;
}

//hexadecimal 4
if (ppp[3]=='0') {
}
else if (ppp[3]=='1') {
}
else if (ppp[3]=='2') {
}
else if (ppp[3]=='3') {
}
else if (ppp[3]=='4') {
}
else if (ppp[3]=='5') {
}
else if (ppp[3]=='6') {
}
else if (ppp[3]=='7') {
}
else if (ppp[3]=='8') {
}
else if (ppp[3]=='9') {
}
}
}
}
}
}
}
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') {
}
else if (ppp[4] == '2') {
}
else if (ppp[4] == '3') {
}
else if (ppp[4] == '4') {

}
else if (ppp[4] == '5') {
}
else if (ppp[4] == '6') {
}
else if (ppp[4] == '7') {
}
else if (ppp[4] == '8') {
}
else if (ppp[4] == '9') {
}
}
}
}
}
}
}
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]);
}
```c
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 *ttttt, 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\n",x1,x2,y1,y2);
                // printf("\n%d",cachevar[16].jtag);
                currow=y1; curcol=x1;
                //
                while(bilinear<4) { // INCLUDE THIS LINE
                    tttt=caloit(ll, 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;
                    }
                    else {
                        cachevar[index].jtag=tag;
                    }
                    //
                    printf("\n%d,%d,%d,%d,%d,Hit\n", currow, curcol, index, tag, offset);
                    // getch();
                }
            }
        }
    }
    //
    printf("\n%d,%d,%d,%d,%d,Miss\n", 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("%d,%dx%d,%d,%d,%4.6f,%4.6f,%4.6f\n",bwid,l,w,pixels,cycles,hits,ignore,timee);
fprintf(f2,"%d,%dx%d,%d,%d,%4.6f,%4.6f,%4.6f",bwid,l,w,pixels,cycles,hits,ignore,timee*1000);
fclose(f2);
}
References

For paper see at: http://www.vis.uky.edu/~xhuan4/LFDisplay/LFdisplay-short-sub.pdf)


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


(See at: http://graphics.stanford.edu/papers/texture_cache/ )


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.