An Automatic Cache Generator for Stratix FPGAs

by

Peter Yiannacouras

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE

DIVISION OF ENGINEERING SCIENCE

FACULTY OF APPLIED SCIENCE AND ENGINEERING
UNIVERSITY OF TORONTO

Supervisor: J. S. Rose

April 2003

Abstract

Caches have been used to successfully alleviate the degradation in performance caused by accessing slow storage components, and hence have become a prominent part of memory hierarchy. In this thesis, a cache generator is proposed which can produce a variety of different caches with different sizes. This allows designers to effortlessly create, alter, and examine different caches in order to best meet the needs of their target system. The target of these cache designs is for FPGA designs, specifically, Altera's Stratix FPGA. Analysis of the area and speed of the generated designs demonstrated that the designs can meet a wide range of design specifications and are in general fast and low-cost cache designs.

Acknowledgements

I would like to thank my supervisor Professor Jonathan Rose for his guidance and valuable input. I would also like to thank Sepehr Seyedi for his assistance with formatting, John Virk for his insight, and Aneal Harlal for providing access to necessary apparatus.

I would also like to thank my family and friends for their support throughout my undergraduate experience in the Engineering Science program.

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 Stratix and its device resources 3
2.2 CAD flow 4
2.3 Speed and fmax 4
2.4 Latency 4
2.5 Caches 5
2.5.1 Cache Line Size 5
2.5.2 Cache Depth 6
2.5.3 Associativity 6
2.5.4 Replacement Policy 7
2.5.5 Write Policy 7
2.6 Content Addressable Memories 8
2.6.1 Xilinx implementation of CAMs using Block RAM 9

3 Design of the Caches **12**

3.1 Fully Associative Cache 13
3.1.1 CAM 16
3.1.2 Encoder 17
3.1.3 Data store 17
3.1.4 Tag store 18
3.1.5 Counter Replacement 18
3.1.6 Optional Registers 19
3.2 Direct-Mapped Cache 19
3.3 Two-Way Set Associcative Cache 20
3.4 Design Verification 22
3.5 Design of the Generator 23

4 Results 24
4.1 Area 25
4.2 Speed 31
5 Conclusion 37
References 38
Appendix A – Data Gathered 39
Appendix B – Sample Waveform 41
Appendix C – Code 42

List of Figures

Figure 1: Representation of a CAM in RAM 9
Figure 2: Tree of possible cache variants 12
Figure 3: Topology of a cache read for the associative cache 14
Figure 4: Topology of counter-based replacement policy 18
Figure 5: Topology of a cache read for the direct-mapped cache 19
Figure 6: Topology of a cache read for the two way set associative cache 21
Figure 7: Graph of LE usage vs cache depth 26
Figure 8: Graph of LE usage vs address width 26
Figure 9: Graph of LE usage vs data width 26
Figure 10: Graph of frequency vs cache depth 32
Figure 11: Graph of frequency vs address width 32
Figure 12: Graph of frequency vs data width 33
Figure B.1: Waveform of sample cache test 41
Figure B.2: Continued waveform of sample cache test 41

List of Tables

Table 1: Characterstics of Each Cache Type 13
Table 2: Memories used for different cache depths 29
Table 3: Memories used for different address widths 29
Table 4: Memories used for different data widths 29
Table A.1: Measurements of LE usage for different cache depths 39
Table A.2: Measurements of LE usage for different address widths 39
Table A.3: Measurements of LE usage for different data widths 39
Table A.4: Measurements of fmax for different cache depths 39
Table A.5: Measurements of fmax for different address widths 40
Table A.6: Measurements of fmax for different data widths 40
Table A.7: Speeds of a minute cache 40

1 Introduction

Over the last couple of decades, memories have failed to keep up with the increase in speeds seen in microprocessors [6]. This is known as the *processor-memory performance gap*, which is a growing problem showing no signs of abating. The most common way to bridge this performance gap is through the use of caches. Caches were commercially introduced more than 30 years ago in the IBM 360/86 and have been an active area of research for even longer [3]. Several different types of caches were developed and were continuously improved upon, and as a result, significant performance gains can be achieved through their use. Since then, caches have become a salient element in today's memory hierarchy. They are consistently used in both general purpose and embedded computer systems, as well as in hard disks, web servers, internet browsers, and any other devices beset by slower storage components.

The purpose of this thesis is to propose a cache generator which, given a set of input parameters, will output an efficient cache implementation satisfying the given parameters. The generator is very versatile allowing for a number of different cache types to be generated, each with a number of configurable options. Users can select the appropriate size, latency, interface, and behaviour for their desired cache and use the proposed generator to produce an implementation of it. The implementations generated are targeted for FPGA designs, specifically, Altera's Stratix family of FPGAs. Altera's recent release of version 3.0 of its soft processor Nios includes a cache, indicating an evident need for caches in FPGA designs.

The remainder of this document is organized in the following sections in order of appearance: Section 2, the Background section, explains concepts fundamental to

understanding the designs and results in this report; section 3, the Design section, describes the various cache designs that can be generated; section 4, the Results section, examines the resource utilization and speed performance for caches of various types and sizes; section 5, the Conclusion section, includes concluding remarks. Supplementary information is contained in three appendices.

2 Background

2.1 Stratix and its device resources

Stratix is the name given to Altera's currently most advanced family of programmable logic devices. Using state of the art 0.13 micron technology, it contains an abundance of resources and permits operational speeds of up to 420 MHz [1]. Resources in an FPGA come in two basic forms: Logic elements and memory blocks. *Logic elements*, or *LEs*, are used to implement arbitrary logic functions and consist of a 4-input look up table (LUT) and a D flip flop. *Memory blocks* are used as storage components, and in Stratix, come in three different sizes: 512 bits, 4 kilobits, and 512 kilobits (not including parity bits). These memory blocks are named M512, M4K, and M-RAM (Mega RAM) according to their size. Respectively, they can operate at 318 MHz, 291 MHz, and 256 MHz. All memories are dual port memories with customizable address and data widths on each port. However, the M512 can only use one port for reading, and the other for writing, Altera refers to this as a "simple dual-port RAM" [1].

Logic Elements and memories are distributed throughout the chip and can be used to build arbitrary digital systems as long as the device has enough to do so. Thus, resource usage (or equivalently, area usage) is generally reported in terms of the number of logic elements (LEs), and individual memories used in the design. The size of a circuit is a crucial parameter in any digital design especially since it is directly proportional to cost. In this case, larger circuits call for larger and hence more expensive Stratix chips. Thus minimizing area is a high priority in all designs.

2.2 CAD flow

The generator outputs a Verilog description of the cache as well as a Quartus project file. Quartus is Altera's Computer Aided Design (CAD) software package. It can synthesize HDL code with its own native compiler and then place and route the design onto a selected Altera FPGA. Since the Quartus software package is fully capable of accomplishing all aspects of the implementation of the caches onto a Stratix FPGA, it was the only tool used in the CAD flow of this project. Specifically, the CAD software used was Quartus II version 2.1.

2.3 Speed and fmax

Quartus produces a report of resource utilization and timing analysis. One of the most important measurements made during timing analysis is the register to register maximum frequency (*fmax*) which is used to define the speed of a system. This metric represents the maximum speed the clock signal can have while ensuring data is correctly transmitted and received by all registers in the system. The two registers which are furthest apart, in terms of the time it takes for data from the source register to reach the destination register, define the fmax. The path taken by the slowest signal from this source to the destination is called the *critical path.*

2.4 Latency

Latency is defined as the number of cycles required to complete an operation not including the cycle in which it was issued. Hence, an operation with a latency of one will receive its request in one cycle and deliver its result in the next. Designers must choose how to allocate the time required to complete an operation. An operation can be

distributed over multiple clock cycles, or completed in one long clock cycle. This negotiation of latency and frequency is highly application specific.

2.5 Caches

A cache can only store a subset of the data available in the address space. If the currently addressed data is found in the cache, a *hit* is said to have occurred and the cache can satisfy the memory operation without involving the slower memory. If, however, the data is not found in the cache, then a *miss* is said to have occurred and the slower memory must be accessed. An effective cache is one which minimizes misses, hence having a high *hit rate*.

Data values in the cache are identified using a *tag*. A tag is the part of the address required to uniquely identify the data. Tags are each stored in a *tag store* alongside its corresponding data stored in a *data store*. To detect a hit, the cache must compare the tag of the currently addressed data, to all the tags in the tag store. These are some of the basic terminologies relating to caches. There are several adjustable attributes in a cache which may radically change its performance and cost. These attributes are described below.

2.5.1 Cache Line Size

The unit of data storage used in the cache is known as a *cache line*. The cache line size, usually measured in bits, depends on the memory device and can be as small as the smallest possible data that can be transferred to/from the memory device. Many memories will access a maximum of 32 bits of data in a single access, but still allow for individual 8 bit data values to be addressed. The question arises of whether to cache each

individual byte, associating a tag with each one, or to cache groups of bytes. By doing the latter, the circuit becomes more complex, but it can exploit the fact that after accessing one piece of data, it is highly likely that neighbouring data will be accessed soon after. This is known as *spatial locality* and is exhibited by many programs [6]. In addition, this method takes advantage of burst mode transmissions which make accessing a group of bytes from the RAM faster than accessing each byte individually.

2.5.2 Cache Depth

The maximum number of cache lines that can be stored in the cache is known as the *cache depth.*

2.5.3 Associativity

Ideally, new data can be added to the cache as long as the cache has a cache line available. This implies that any data can map to any cache line. Such a cache is known as a *fully associative cache*, or just associative cache. To find a match it must search through all entries in the cache and compare it to the tag given. Because of this, a fully associative cache is large, expensive, and potentially slow. To alleviate these problems more simplified caches such as the *direct-mapped* cache have been introduced. A direct-mapped cache maps data to only one cache line determined by the low order bits of its address; these bits are referred to as the *index* bits. With this mapping, the cache line being read from or written to is known immediately from the address. As a result, the circuit is much simpler and faster. In between these two extremes is the *set associative* cache. Instead of using the index bits do define a single cache line, they are used to define a small number of possible cache lines where the data can be placed. These select

cache lines define one of many sets. Since this cache needs only to search through the small set instead of the entire cache, it is also much simpler than the fully associative. Set associative caches generally provide the best compromise between circuit complexity and performance.

2.5.4 Replacement Policy

When new data is to be cached and all available cache lines are occupied, old data must be evicted from the cache to make space. The strategy used to choose which data to evict is known as the *replacement policy*. Since a direct-mapped cache maps data to only one location, it is not applicable to speak of replacement policies for such a cache. However, for associative and set-associative caches, a replacement policy is mandatory and can come in many varieties. The most common method is known as *LRU* (Least Recently Used). This strategy tracks how recently each piece of data was referenced, and selects the one which was used least recently to evict. This policy requires more circuitry than other more simple approaches; however, it accurately captures the principle of locality.

2.5.5 Write Policy

The manner with which a cache manages write operations is referred to as the cache's *write policy*. A write operation can potentially cause the cache and memory to become unsynchronized. If the cache contains a more recent value than the memory, and that value is evicted without being written to memory, this value would be lost. To prevent this problem, a cache can ensure that new values are flushed to memory before being evicted. A cache which behaves in this manner is using a *write-back policy*.

Another means of dealing with this problem is to ensure the synchronization of the cache and memory by always writing to both. This is known as a *write-through* policy and is much simpler than the write-back policy. However, using the write-back policy obviously results in more effective caches.

2.6 Content Addressable Memories

As discussed previously, a cache must search through a number of tags to find a match; this process can be done using a *CAM* (C*ontent Addressable Memory*). A CAM is the inverse of RAM. While RAM is given an address and outputs the data stored at that address, a CAM receives data (often called a *pattern*) and returns the address where it is stored, or indicates that the pattern is not currently in the CAM. This makes CAMs ideal for searching through tags and detecting cache hits. CAMs can come in different varieties. Some support *multiple-matching* and other more powerful search options, but these features are immaterial in the context of caches. In a cache, the pattern given to the CAM is the tag and the address returned by the CAM is the position of the data in the data store. Only one data word can be associated with each tag, and the tag itself is unique by definition. Therefore, only the basic function of the CAM is required, the aforementioned features would needlessly increase resource utilization. Consequently, the only relevant parameters are the CAM's size. A CAM's size is specified in the same manner as in a RAM. It has a depth and a width that are reported respectively. For example, a 32 x 8 CAM stores 32 words each 8 bits wide. The CAM used in this design was derived from an implementation described in a Xilinx Application Note [4]. The Xilinx implementation is summarized below.

2.6.1 Xilinx implementation of CAMs using Block RAM

This implementation was developed by Xilinx for use with their Virtex family of FPGAs. It uses a technique which manipulates RAM to be used as a CAM. The obvious approach of cycling through all words in the RAM testing each for a match would be a very slow implementation of a CAM. Instead, in this Xilinx implementation, the pattern is used as the address to the RAM, and the data in the RAM stores the positions in the CAM where the pattern can be found. To store the pattern with decimal value 6 in a CAM at address 2, its representation in the RAM would be as shown in Figure 1 for a CAM with 16 words and an 8 bit input pattern.

DATA \ Address in CAM	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
⋮																⋮
255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1: Representation of a CAM in RAM

In the RAM, there is an entry for every possible input pattern (as seen along the left most column of Figure 1), and each entry has one bit for every possible location in the CAM. Thus an m x n CAM requires a 2^n x m RAM. Xilinx uses RAM blocks with 4096 bits which can be configured as a 256-word x 16-bit RAM. This in turn can be used as a 16-word x 8-bit CAM in the manner shown in Figure 1.

Initially, this may seem like an extremely poor implementation since it requires a vast amount of RAM. In fact, the amount of RAM required for an m x n CAM grows exponentially with the data width as seen in equation 1, where m is the CAM depth and n the width.

$$\#bits = m \times 2^{n} \tag{1}$$

With 32-bit data, the CAM would require more than 4 gigabits of memory, far more than any FPGA can provide. However its elegance lies in the fact that these small CAM blocks can be cascaded to increase both the depth and width of the CAM with linear growth in each dimension. Hence a 32-word deep CAM can be built out of two 16 word CAMs. Similarly, a 16 bit wide CAM can be built out of two 8-bit wide CAMs. In general, using M x N CAM blocks, an m x n CAM will require the following number of memory bits.

$$\#bits = M \times 2^{N} \times (\frac{m}{M}) \times (\frac{n}{N}) \tag{2}$$

Simplifying,

$$\#bits = \frac{1}{N} 2^{N} \times m \times n \tag{3}$$

As seen in equation 3, the growth is no longer exponential with n, instead, it is exponential with the parameter N which is fixed and determines the CAM block width. The expression is also independent of M, meaning CAM blocks can be of any depth (less

than m) without affecting the amount of memory used overall. As a result, it remains only to choose a suitable N. The function $\frac{1}{N}2^N$ is monotonically increasing for integer values of N greater than zero, hence, the optimal N is the minimal N allowed. Because of constraints on the aspect ratio of the RAM blocks in Xilinx's Virtex, the smallest possible N, or shallowest memory, occurs for N=8. The CAM block depth M is then assigned a value such that it utilizes the entire 4096 bit RAM block, and hence the Xilinx implementation uses a 16 x 8 CAM block to build larger CAMs.

As shown above, there is certainly a recognizable advantage in cascading smaller CAM blocks to build larger sized CAMs. This scalability is provided by the CAM's decoded address. The output of the CAM block has one bit for each word in the CAM. Thus, to add more depth to the CAM, one need only increase the number of bits in the output. This can be done with some additional logic by simply using multiple CAM blocks in parallel. The additional logic is required to select the correct CAM block when write operations occur. The width of the CAM can also be extended by simply performing a logical AND of the output of multiple CAM blocks. Each CAM block receives a subsection of the input pattern; the full pattern is in the CAM if each CAM block experiences a match in the same address.

The major limitation in this design is that there is poor support for erasing values from the CAM. The CAM can easily remove a given pattern from a given address, but can not remove all patterns at a given address. Doing so requires erasing it from every possible entry in the CAM. Looking at Figure 1, this would require writing a zero to the correct column of each of the 256 rows in the RAM. Using 256 cycles to erase data is impractical, hence requiring designers to remedy or avoid this erasing problem.

3 Design of the Caches

The automatic cache generator can be used to produce a variety of different caches, each with a variety of different options. This flexibility allows designers to better meet their unique needs for area, speed, and cache effectiveness. The dominant factor in finding this balance is in choosing the associativity of the cache. With this generator, designers can choose between fully-associative, direct-mapped, or two-way set-associative caches. Each cache can then be configured to have different read and write latencies. Latency here is exclusive of any time required to access peripherals outside the cache, namely, a slower memory device. A tree diagram of the various cache designs and their configurable read/write latencies is shown in Figure 2.

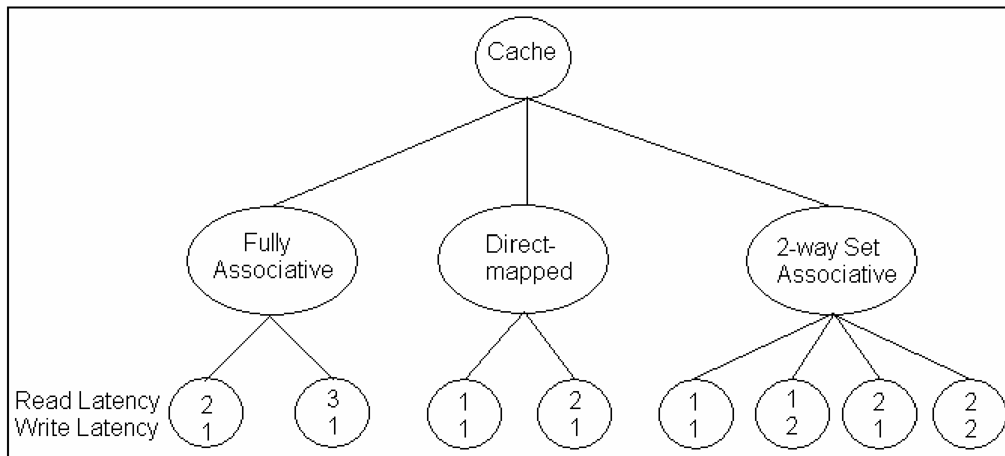

Figure 2: Tree of possible cache variants

Options also exist to provide an output port for indicating cache hits, and whether to propagate control signals (read/write requests) to the memory immediately or only after the cache discovers it has missed. These options can be used to meet certain

interface requirements. However, neither produces an appreciable change in the area, speed or functionality of the cache and hence will be ignored in this discussion.

Other characteristics of the cache are not configurable and are common to all the caches. Since the generator can ideally be used with any type of memory, and by any processor/device, these unconfigurable implementation decisions were often made in favour of faster performance and smaller area. Hence, all generated caches employ a write-through policy. For the same reason, none of the caches allow access to sub-sections of the cache line. Cache operations work only on whole cache lines and the cache line is equal to the size of the user-defined data word. Aside from these two stipulations, there are other options to allow designers ample flexibility in choosing a cache. A summary of the cache characteristics are shown in Table 1. Entries in italics are user-selectable. A more detailed description of the individual caches and their various configurations follow.

Table 1: Characteristics of Each Cache Type

Associativity	Fully Associative	Direct-mapped	Two-way Set Associative
Read Latency	*2/3*	*1/2*	*1/2*
Write Latency	1	1	*1/2*
Replacement Policy	Counter-based	N/A	LRU
Write Policy	Write-Through	Write-Through	Write-Through
Depth	*any*	*any*	*any*
Address Width	*any*	*any*	*any*
Data Width	*any*	*any*	*any*
Cache Line Size	=Data Width	=Data Width	=Data Width
Cache hit Output	*yes/no*	*yes/no*	*yes/no*
Propagate on Miss	*yes/no*	*yes/no*	*yes/no*

3.1 Fully Associative Cache

The design of the fully associative cache involves 5 main components: A CAM, an encoder, a data store, a tag store, and a counter. Before getting into the implementation details of each component, the functionality of the cache will be

described in terms of how these components help to achieve its function. A schematic of a how the CAM, encoder, and data store satisfy a read operation is shown in Figure 3.

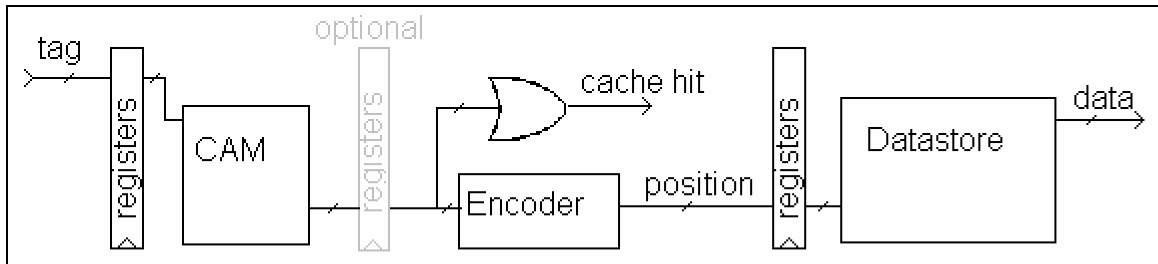

Figure 3: Topology of a cache read for the associative cache

After receiving a read request, the cache must determine which, if any, of the locations in the data store contain the data associated with the given tag. This location will be referred to as the data's *position*. The CAM provides a mapping from tags to positions, therefore the CAM will return where the data is located in the data store. Since the CAM has an output line for each possible position in the data store, the line that corresponds to the correct position will be set high. If none of the lines are high then a miss has occurred and the data is not in the cache. A simple logical OR of all the CAM outputs is used to detect this. However, while the decoded output of the CAM is ideal for detecting cache hits and cascading CAM blocks (as discussed previously), it is slightly inadequate here since the data store is a block of RAM which requires an encoded address. Hence an encoder must be used between the CAM and the data store as shown.

The optional registers allows for the choice in 2 or 3 cycle read latencies. To implement the 2 cycle read latency, the optional registers are taken out of the design leaving the CAM lookup, logical OR, and encoding to be done in one clock cycle. In the 3 cycle read latency case, the optional registers are employed to isolate the CAM lookup

in the first cycle. The logical OR and encoding is done in the second cycle, and the third cycle performs the data fetch.

When a cache miss occurs the difference is only in the last cycle. Instead of fetching from the data store, a read is issued to memory and the cache waits for the data to be retrieved. Upon receiving the data, it passes it to the processor (or other bus master device) and performs a write operation in order to cache the new data.

Write operations have a completely different structure than reads. A write requires two clock cycles though the first is done in the same clock cycle the write is requested. This first cycle merely deletes whatever cached value is located at the position targeted by the counter replacement algorithm. Performing this deletion requires overcoming the previously discussed erasing problem of the Xilinx CAM which is used in this design. To this end, a tag store is included with the associative cache, despite the fact that ideally the CAM should serve as the tag store. Since the Xilinx CAM implementation allows multiple matching, which also enables its cascading ability, the cache must explicitly enforce that only one tag can be associated with a cache location. The tag store is used for this purpose since it can only store one tag in each location. The CAM is a one-way mapping from tag to cache position, while the tag store is a one-way mapping from cache position to tag. When data is being evicted from a given position in the cache, the tag store is used to identify its tag so that it may be erased from the CAM. This lookup requires one clock cycle; however in the design, it is ensured that this is done before the end of any previous operation. Hence the tag to be evicted is available at the start of any subsequent operation, enabling the first cycle to be used to erase this tag, and

the following cycle used to write the data to the data store and the new tag into both the CAM and tag store.

The counter-based replacement policy allows for a simple and effective way to select data to evict from the cache. The counter points at the next value to be evicted in the cache and increments with each new value cached. Hence, values which were entered in the cache least recently are evicted first. The downfall of this method is that it does not take into account how many times a piece of data was accessed since it was initially cached. The advantage is that it requires much less resources and simpler control circuitry. This method was also used in the sample code for an associative cache using Altera CAMs [2]. Implementation details of the individual components follow.

3.1.1 CAM

The CAM implementation used in this design is identical to the Xilinx CAM; the only modification is in the chosen dimensions of the CAM blocks. The Xilinx 16 x 8 CAM blocks were implemented in 4096-bit RAM blocks since this is the only size of RAM blocks available on Xilinx's Virtex chips. From equation (3) it can be seen that the coefficient $\frac{1}{N}2^N$ is solely responsible for the inflation of memory bits required. This coefficient will be referred to as the *waste factor*. For N=8, as used in the Xilinx design, the waste factor is 32. On Stratix, the parameter N can be made as small as 5 if implemented in an M512 as a 16 x 5 CAM. This results in a waste factor of 6.4, which is one fifth of that of the Xilinx design. Consequently, the M512 is used for the CAM blocks in this design. This was done using Altera's altsyncram megafunction.

3.1.2 Encoder

The encoder is the largest block of logic in the cache and is hence often within the system's critical path. The size of the encoder is determined solely by the depth of the cache, thus increases in cache depth result in a longer critical path and smaller fmax. An efficient implementation of an encoder is required to mitigate this reduction in fmax. Standard implementations of encoders have outputs which are each a function of all the inputs. The implementation used here separates the outputs and makes each of them a wide logical OR of half the inputs. Since the output of the CAM is guaranteed to have at most only one bit high, each output of the encoder needs only to examine the inputs which can set it high and otherwise assume it is low. The zeroth bit, for instance, will be high if any odd-indexed input is high. Similar, patterns can be found with the other outputs. In addition to the reduction of the complexity of the function, further performance gains are achieved by the ability of Stratix to implement fast wide OR function using a direct connection to neighbouring LEs known as a carry chain.

3.1.3 Data store

The data store is simply implemented as RAM using the altsyncram megafunction. No specific RAM block is specified which allows the compiler to decide which of the three blocks, or combination of blocks, should be used to meet the size requirements of the data store. The data store has separate data input and data output ports. The data input port contains a multiplexer which chooses between the data from the processor or memory. Similarly, the data output port chooses between the cache's output and the memory's output. This is the case for all cache designs.

3.1.4 Tag store

The tag store is also implemented as a RAM block using the altsyncram megafunction. Again it is left to the compiler to decide which RAM blocks to use.

3.1.5 Counter-based Replacement

The counter is implemented using the built in lpm_counter megafunction. However the design is slightly more elaborate, as seen in Figure 4, in order to save a clock cycle as mentioned previously. It saves a clock cycle by ensuring that at the end of any memory operation, the tag to be evicted on the *next* memory operation is outputted from the tag store. Normally, this requires only that the counter value be used as the tag store's input, the exception is after an eviction. After an eviction occurs, the counter is signaled to increment during the one latent cycle available during a write operation. The counter will be updated in the next clock cycle meaning that the tag store won't be updated until the clock cycle after. The cycle spent incrementing the counter can be concealed with a purely combinational incrementer. The circuit chooses between the latched counter output and the combinational incremented output in a manner to ensure that the output of the mux is a value which increments immediately upon being signaled.

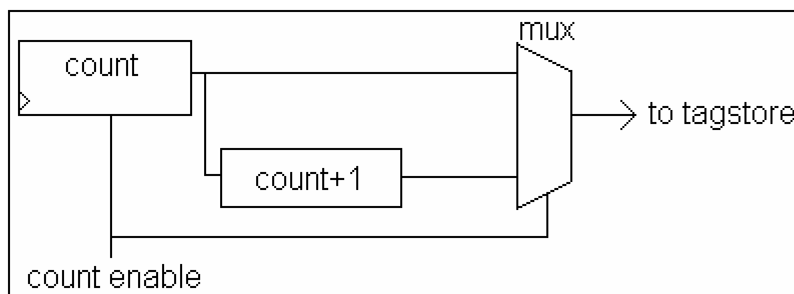

Figure 4: Topology of counter-based replacement policy

3.1.6 Optional Registers

The optional registers were implemented using the built-in flip flops of the memory blocks. The altsyncram function enables users to specify whether these registers should be used or not. If not they are bypassed. Doing this small optimization allows the optional registers to be added to the circuit without any increase in logic cells. Throughout the design, optional registers will always be placed adjacent to the output of memory blocks in order to capitalize on this savings.

3.2 Direct-Mapped Cache

The direct-mapped cache has a very trivial design. It requires only a tag store, data store, and comparator. Since only one tag in the tag store can match the tag input, that tag is the only one read, and the only one that need be compared. Therefore only one comparator is needed. A schematic of the read circuitry is shown in Figure 5.

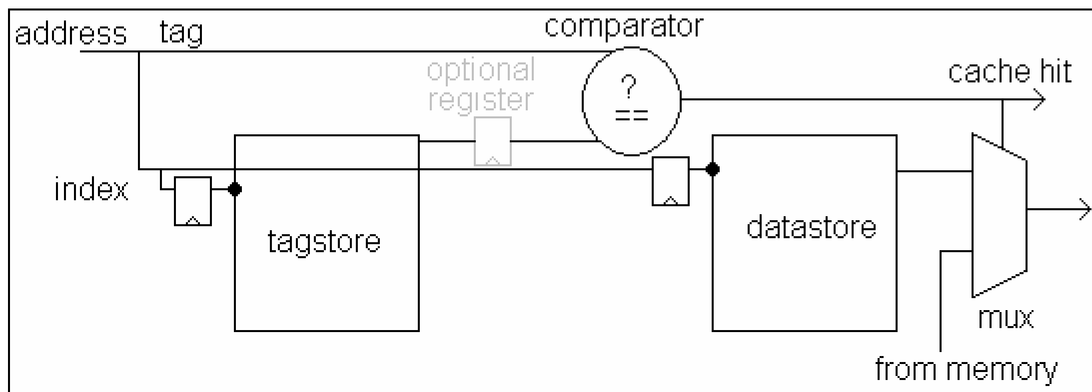

Figure 5: Topology of a cache read for the direct-mapped cache

The tag store and data store are in parallel in this configuration. Since the data store does not require any output values from the tag store, the data fetch can be done

concurrently with the tag lookup enabling single-cycle latencies. In the single cycle case, the data is fetched while the tag is fetched and compared. If the tags match than a hit has occurred and the mux is set to output the contents from the data store. Otherwise the cache waits for the memory to fetch the data and passes it through the mux. The insertion of the optional register can significantly increase the fmax of the cache by separating the memory access time from the time to do a comparison.

The individual components were implemented using the megafunctions available in Quartus. The tag store and data store both used the altsyncram megafunction while the comparator used the lpm_compare megafunction. The mux and any other small logic were implemented with standard Verilog constructs.

3.3 Two-Way Set Associative Cache

The two-way set associative cache is generally a slightly bigger and slower, but more effective cache than the direct-mapped. Its purpose is to serve as a compromise between the expensive fully associative cache and the oversimplified direct-mapped cache. Because of this, it is designed to be more configurable in order to provide more flexibility for designers. The two-way set associative cache can have either a one or two cycle read latency coupled with either a one or two cycle write latency. It is also made more effective by employing a true LRU replacement policy.

The design of the two-way set associative cache is very similar to that of the direct-mapped cache. In effect, the design is simply two direct-mapped caches arranged in parallel. This makes the design ideal for taking advantage of Stratix's bi-directional dual port memory in order to compress the two caches into single RAM blocks. During a read operation, the set is accessed in its entirety. Both tags, as well as both cached values

in the set are read simultaneously using the dual port feature. Each of the tags are compared to the input tag using two different comparators. If either tag matches, the corresponding data is returned. If neither tags match, the cache must wait for the fetch from memory to complete. A schematic of this operation is shown in Figure 6.

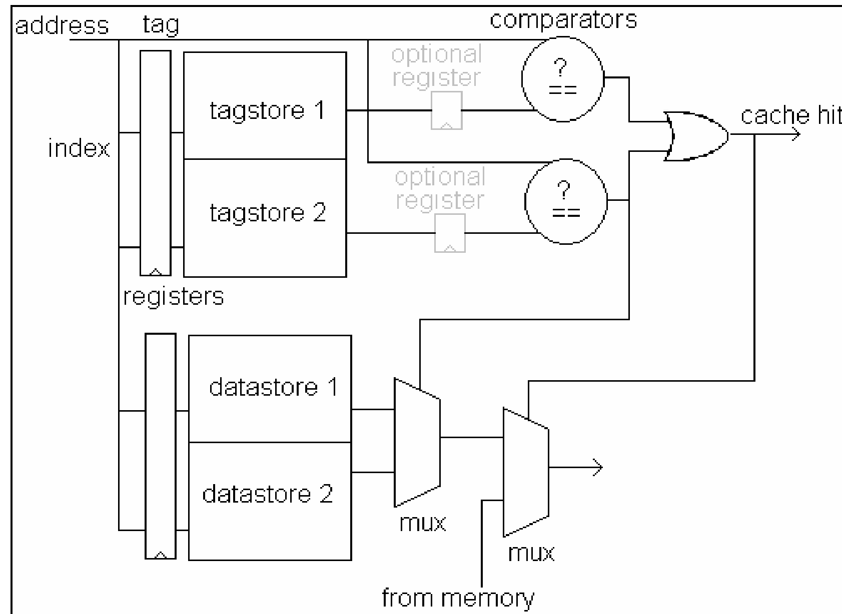

Figure 6: Topology of a cache read for the two way set associative cache

The data store and tag stores are each implemented using the altsyncram megafunction with the bidirectional dual port option set. The comparators are implemented using the lpm_compare megafunction. The optional registers allow for single or dual cycle read latencies in the same manner as in the direct-mapped cache.

Write operation are also identical to the direct-mapped cache except that the LRU circuitry must specify which of the two cache lines in a set are to be overwritten. The LRU circuitry must establish an ordering of all the values in each set. Since there are only two possible values in the sets of a two-way set associative cache, an ordering can be imposed using a single bit. The circuit needs only to toggle the bit when any change

is made to the ordering. The implementation of this circuit is a one bit wide RAM block with one bit for each set in the cache (in this case this is half the cache depth). The circuit toggles by inverting its output and feeding it to its own input data lines. The RAM can be used with optional output registers which allows the user to expand the write latency to two and potentially speed up the design. Doing so isolates the LRU read operations from logic that uses this result.

The LRU circuit can be implemented using LEs, which may be faster for small cache sizes. However, by using block RAMs, the speed of the circuit is fixed for any size cache. A cache with 4096 words would require 2048 LEs to implement a flip flop for each set. This would use an enormous amount of area on the chip and would also require a huge select circuit to choose which flip flop is currently being accessed. The larger this circuit becomes, the slower it will perform. Surely its performance would quickly become slower than the 3 ns (approximately) needed to access the one M4K block which can be used instead.

3.4 Design Verification

The designs of all caches were initially exposed to exhaustive testing to verify their correct functionality. This was done using the waveform simulation in Quartus. All read and write operations, the replacement policy, the assertion of the wait signal, and overall functionality of the cache were confirmed using simulation. After the overall structures of each cache variant were verified, the caches were further tweaked and altered. In order to ensure that these changes did not introduce new bugs, a small subset of the tests were performed on new versions of the caches. A sample of the waveforms used is shown in Appendix B.

3.5 Design of the Generator

The program which generates the cache designs is a small C program which reads input parameters from the command line and outputs the corresponding cache to a file. This program is relatively small and simple since many of the configurables are embedded in the Verilog output code using preprocessor directives. The program has a template of Verilog code for each associativity, each containing preprocessor directives which can be easily set to compile the desired cache. In fact, the direct-mapped and two-way set associative caches can be entirely configured within the Verilog code by simply defining or undefining the constants located at the top of the file. Additionally, the dimensions of the cache are parameters to the module which can be set when instantiating the cache. The C program simply returns this code with the correct parameters set or unset. On the other hand, generation of the associative cache requires significantly more effort. The cascading of the CAMs is a non-trivial task and is hence done in the C code. Similarly, the encoder is also generated wholly from within the C generator program.

4 Results

There are many factors to consider when choosing the best cache for a system. One such factor is the effectiveness of the cache in terms of its hit rate. There has been an abundance of studies on the hit rates of caches with different sizes, associativities, write policies, and replacement policies [6]. The statistics provided by these studies can be combined to evaluate the effectiveness of one of the generated caches over another. This thesis will not attempt to classify any generated cache with respect to hit rates or overall effectiveness. Instead the focus will be on measurements related to the cache's implementation onto Stratix. Specifically, the designs will be compared with respect to their speed, in terms of fmax, and area, in terms of both LEs and memories used, as reported by Quartus II version 2.1.

Factors such as latency, size, and cache associativity will be treated as independent variables. Each of these are parameters which designers can choose based on the results to follow. Measurements were made for each of the eight cache variants, that is, for each type of associativity and latency (as seen in Figure 1). Unfortunately, the size parameter allows for an enormous amount of different caches. Taking measurements for each one is clearly unfeasible; instead, the size of the cache is broken up into its three components (cache depth, address width, and data width) so that each may be examined independently. Thus, two of the components are held fixed while the third is varied in order to observe the effect it has on the cache's area and speed. A 32-word cache with a 32-bit address space and 32-bit data (32x32x32 respectively), is used as a common reference point. Each dimension is increased individually starting from this reference point.

4.1 Area

Area measurements were made by compiling the cache chip alone. If the cache was compiled within some system, there is a good chance that logic could be merged with logic from the system so that the effective area added to the system is less than that reported here. In addition, parts of the design may be reduced by the compiler if unused or aliased signals are found. Hence these measurements are an upper bound on the area of the cache.

Both LEs (logic elements) and memories (M512s, M4Ks, and Mega RAMs) will be measured and discussed in this section. The effect of increased cache depth, address width, and data width will be examined for all cache types. However, it was discovered that changes in latency have a negligible impact on the area of the circuits. This is mostly due to the savings attained by implementing the optional registers as part of the memory blocks. Otherwise, only small changes to the control logic were made, and hence, the area was scarcely affected. Because of this, the measurements will be made only for the different associativities. This data is shown in Figures 7 through 9.

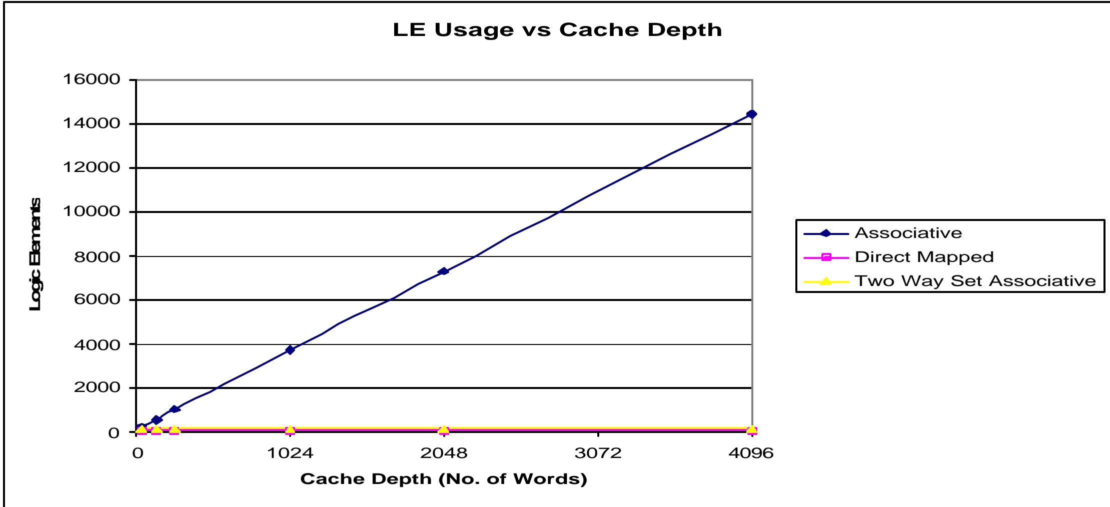

Figure 7: Graph of LE usage vs. cache depth

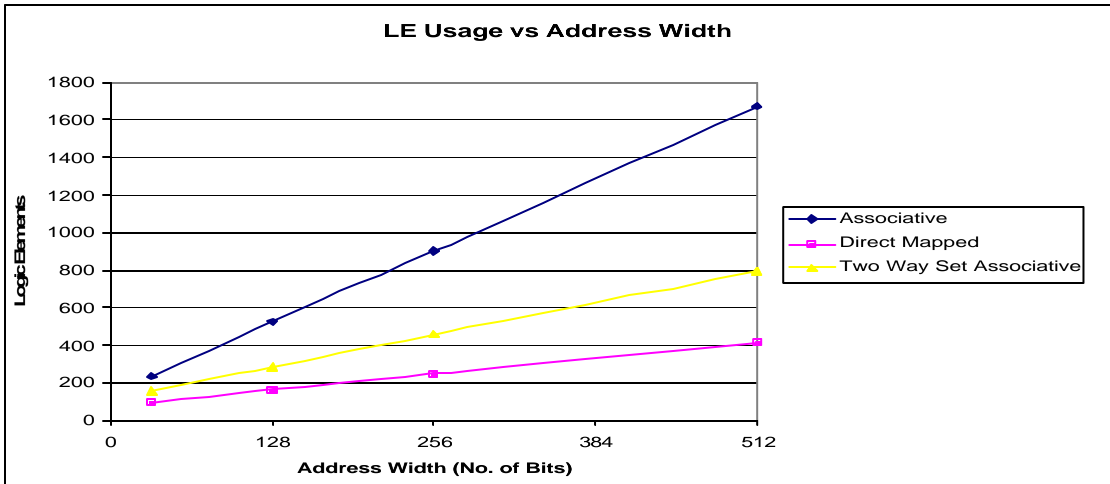

Figure 8: Graph of LE usage vs. address width

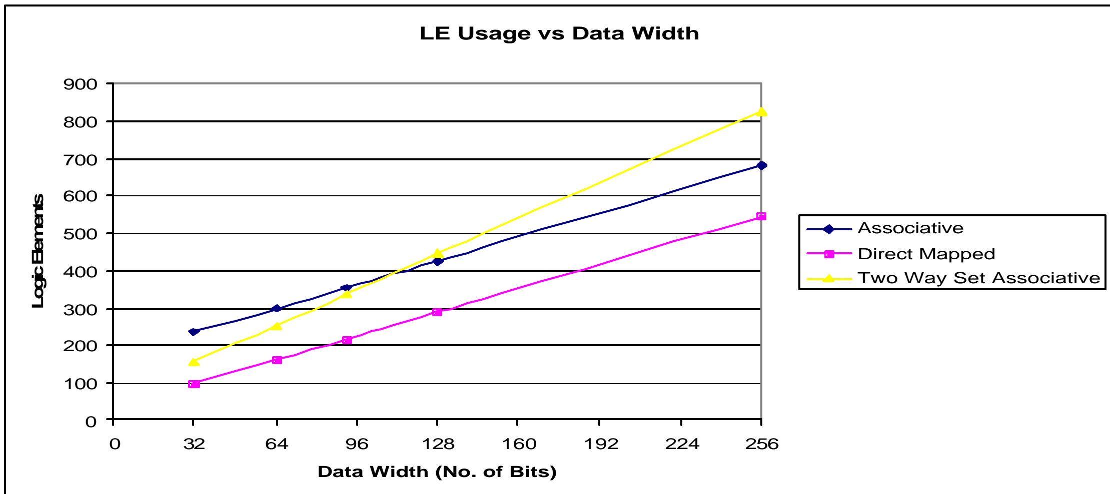

Figure 9: Graph of LE usage vs. data width

Figure 7 shows the logic elements used for implementing caches with depths ranging from 32 to 4096 words. One important feature in this graph is that the number of LEs used for both the direct-mapped and two-way set associative cache is independent of the cache's depth. They are fixed at 98 and 155 respectively (see Table A.1 in Appendix A). This is expected since the logic components of those designs include only the comparator, which depends on address width, and the output multiplexers, which depend only on the data width. The increased cache depth only affects the memories used.

The effect of cache depth is particularly important, since larger cache depths generally mean larger hit rates. The address width and data width are likely to be defined by the system but the cache depth can be independently adjusted to attain a desired hit rate. The results seem to suggest that in the direct-mapped and set associative caches, this can be done without increasing the number of logic elements. Moreover, the number of logic elements is seen to be very small. The smallest Stratix chip, the S10, contains 10 570 logic elements. The 98 or 155 LEs required by these caches will use up only a small fraction of these, making the designs very inexpensive with respect to LEs.

Conversely, the fully associative cache is quite expensive, as is expected. The change in LEs with cache depth appears to be linear with a slope of 3.53 LEs/word. The contributing factors are the CAM, the encoder, the LRU circuitry, and the wide logical OR. Still the design is reasonably small in terms of LEs. A 16KB (4096 words x 4 bytes/word) associative cache, which is twice the size of the Pentium 4's L1 data cache, can be implemented with 14 438 LEs. This can be programmed on the slightly bigger

S20 Stratix chip, and would need only a minute fraction of the LEs of the biggest Stratix which has 114 140.

In Figure 8, it is demonstrated that all cache types suffer increases in area with increases in address widths. The direct-mapped and two way set associative cache designs both exhibit increases because of the comparators which will now need to compare larger tags. For tags of width n, the comparators are a function of 2n inputs. Since this function is likely implemented using the fast carry-chains, one input of the LUTs is used up and hence each LUT has only 3 available inputs. Thus the number of LUTs (and hence LEs) needed for the function is 2n/3 or 0.667 LEs/bit. By no coincidence, the increase seen in Figure 8 for the direct-mapped and two-way set associative caches are linear with slopes of 0.667 LEs/bit and 1.33 LEs/bit respectively. The two way set associative cache increases at twice the rate of the direct mapped since it contains two comparators. The associative cache increases at a much higher rate of 3 LEs/bit. This figure is composed of the 1 LE/bit for the tag mux which inputs into the CAM, added to the 2LEs/bit for cascading CAM blocks. Ideally only 1 LE/ bit is required for cascading CAM blocks, but two CAM blocks are required to expand the CAM's depth to the reference value of 32, therefore 1 LE/bit is required for each. Still LE usage is reasonably small allowing for an absurdly large 512-bit address space to use only 1673 LEs for a fully associative cache.

The effect of increasing data width on the number of LEs used is shown in Figure 9. The only logic which depends on the data width are the two muxes on the data input and data output ports of the data store. Thus the growth is linear with slopes of 2 LEs/bit (1 LE/bit for each mux) for the associative cache and direct-mapped cache. The two-way

set associative cache contains an additional data multiplexer to choose between the two values in a give set. Thus its slope is 3 LEs/bit. Again these increases are rather moderate.

While all the designs had relatively small LE requirements, the opposite is seen in their memory requirements. The designs are quite memory intensive and this defines the upper bound on the size of the cache that can be implemented on a given Stratix chip. The number of M512, M4K, and MRAM (Mega RAMs) required for each cache design is shown in Tables 2, 3 and 4. Again the latencies are ignored since this did not impact the number of memories used.

Table 2: Memories used for different cache depths

	Associative			*Direct Mapped*			*Two Way Set Ass.*		
Depth	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*
32	14	2	0	0	2	0	1	4	0
128	56	2	0	0	2	0	1	4	0
256	112	4	0	0	4	0	1	4	0
1024	448	16	0	0	14	0	1	14	0
2048	NA	NA	NA	0	27	0	0	29	0
4096	NA	NA	NA	0	0	1	0	55	0

Table 3: Memories used for different address widths

Addr.	*Associative*			*Direct Mapped*			*Two Way Set Ass.*		
Width	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*
32	14	2	0	0	2	0	1	4	0
128	53	4	0	1	4	0	3	8	0
256	105	8	0	0	8	0	3	16	0
512	207	15	0	0	15	0	3	30	0

Table 4: Memories used for different data widths

Data	*Associative*			*Direct Mapped*			*Two Way Set Ass.*		
Width	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*	*M512*	*M4K*	*MRAM*
32	14	2	0	0	2	0	1	4	0
64	14	3	0	0	3	0	1	6	0
92	14	4	0	1	3	0	1	8	0
128	14	5	0	1	4	0	1	10	0
256	15	8	0	0	8	0	1	17	0

Most of these results are intuitive given the limitation in a memory block's depth and width. Memory blocks are cascaded appropriately to overcome this limitation and create the desired memory space. In the case of the 2048 and 4096 word associative caches, no Stratix chip has enough memory to do so, for this reason the values are not available. Other more important observations will be discussed below.

One important observation is that using a memory block in bidirectional dual port mode reduces the maximum allowed data width in half. Because of this, increases in address and data widths require twice as many memories to increase the tag store and data store in the two way set associative cache. The naïve designer might hence declare that two way set associative caches are hence twice as big in area than a direct-mapped cache but this is clearly not true. While the memory blocks needed are twice as many, they use only half the depth. Therefore corresponding increases in cache depth would eventually make the number of memory blocks used equal for both the direct-mapped and two-way set associative cache. This can clearly be seen in the data corresponding to increases in cache depth. The two-way set associative cache initially requires twice as many M4Ks, but as the depth reaches 256, the two equalize since 256 32-bit words fit in a single M4K.

Another important observation is the growth of the M512s used for the CAMs. In the measurements for the associative cache, the number of M512s used are almost entirely for the CAM. As seen in the tables, these measurements agree with the fact that the growth of the CAM is linear with the growth in each dimension. The slope of this linear growth is 1 M512 per word of depth, and 1 M512 per bit of address. Changes in the data width obviously have no effect on the CAM and this too can be seen in Table 4.

The most interesting observation occurs for the direct-mapped cache with a depth of 4096 words. The compiler apparently merged the tag store and data store into a single Mega RAM block, otherwise, the implementation would have required more than 50 M4K blocks as in the two way set associative cache. This is clearly a valid approach since the Mega RAM can support data widths of up to 144 bits wide, or two 72-bit wide data ports in dual port mode. The tag store requires only 27 bits ($32 - \log_2 32$), and the data store 32 bits. Since both values are less than 72 and since each are used in single port mode, they can each be assigned to one of the available ports of the Mega RAM and each use a subsection of the memory. Implementing the design this way has two main advantages. The first advantage is that it saves an enormous amount of memory blocks. Only 1 Mega RAM block and 98 LEs are needed to implement this 16 KB direct-mapped cache. This enables the design to fit on the smallest Stratix, the S10, while using less than 1% of its logic elements and none of its M512 or M4K memories. The second advantage is that it simplifies routing which can possibly speed up the circuit. Instead of cascading several M4K blocks which span the entire chip, the design can be confined to the small area near the Mega RAM. The disadvantage is that the Mega RAMs are reported as being up to 35 MHz slower than the M4K [1]. More details about the speed of all caches follow.

4.2 Speed

Finding the system's speed required taking measures to ensure the values were consistent and accurate. First, the speed of the caches had to be measured by placing the circuit in a test bed with registers on all inputs and outputs of the cache. Doing this ensures that the fmax reported by Quartus would be the maximum clock speed allowed

for any signal to travel from the inputs of the cache to the outputs of the cache. Another potential inconsistency is in the different speed grades of Stratix chips. To remedy this problem the compiler was instructed to use only the fastest speed grades. With both these solutions in place, data was gathered for all cache types including different latencies since latency plays a pivotal role in the system's fmax. Figures 10, 11, and 12 depict graphs generated from the measured data (see Appendix A) and plotted on a log base 2 scale.

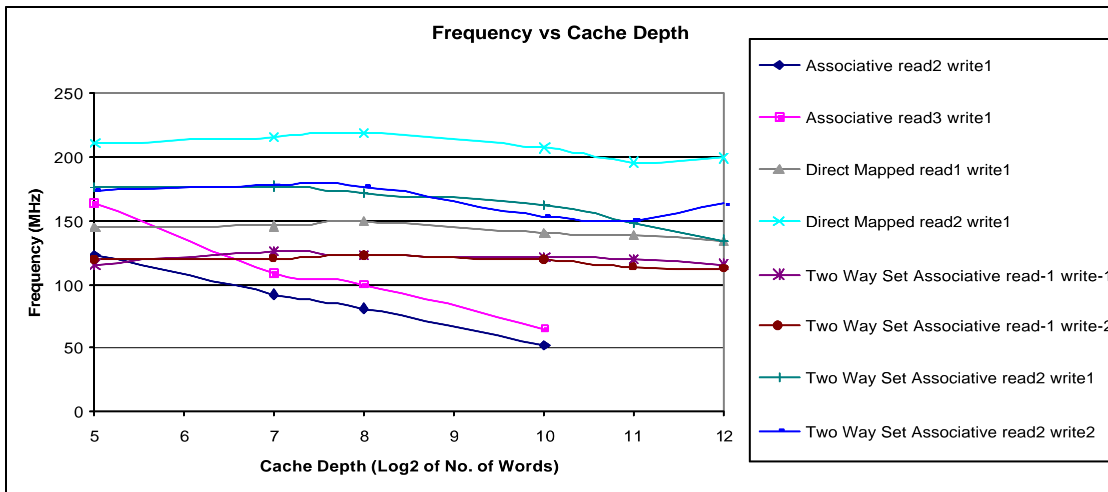

Figure 10: Graph of frequency vs. cache depth

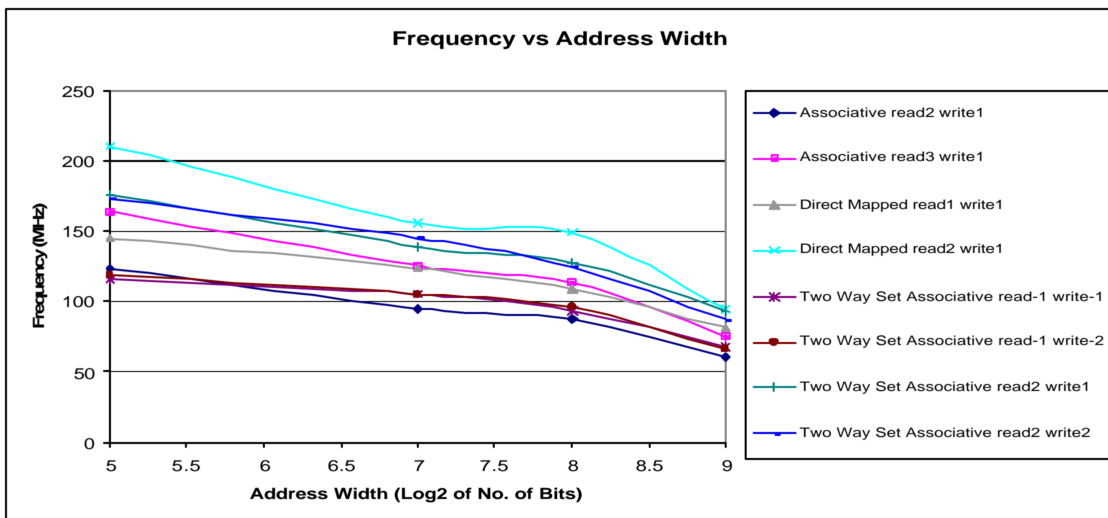

Figure 11: Graph of frequency vs. address width

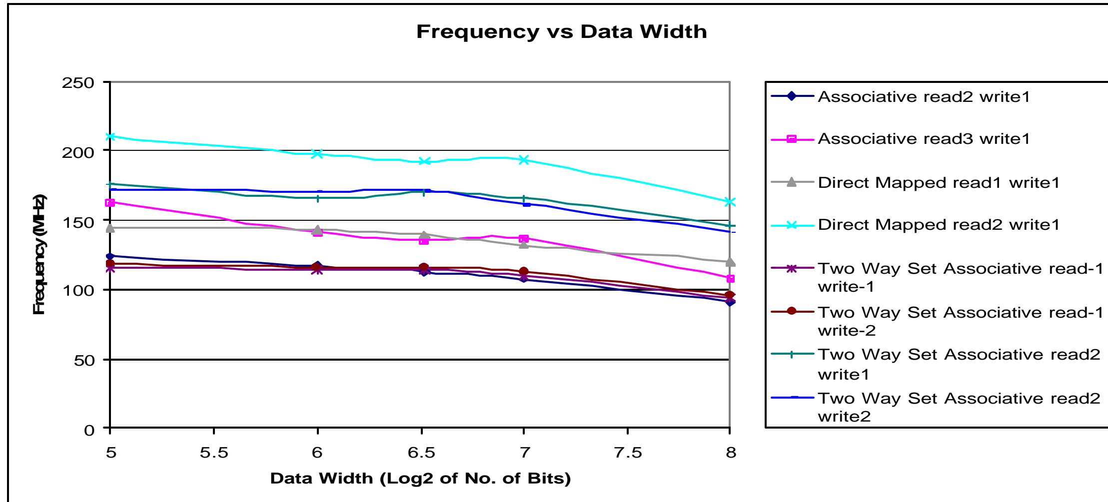

Figure 12: Graph of frequency vs. data width

In all cases the direct-mapped cache with a two cycle read latency is the fastest of the designs with a peak frequency of 218.91 MHz. This is definitely an expected result since this was the motivation behind the birth of direct-mapped caches. Another expected result is that the absolute slowest frequency occurred for the two cycle associative cache of maximum allowed depth and was 52.12 MHz. The price of cache effectiveness is hence a hefty one in these designs; though it is interesting to note that an associative cache with a three cycle read latency often outperforms a single cycle read latency direct-mapped cache. Single cycle reads are also an expensive commodity in these caches. In the 32x32x32 case, the fmax is reduced by 70 MHz for all caches which implement single cycle reads. On the other hand, the adjustable write latencies had a very small impact on the fmax. As seen in the graphs, caches which differ only in write latency exhibit almost identical performances. However, in a minute cache with dimensions 16x10x8, this option produced speed ups of up to 70 MHz since in this and some other caches, the LRU circuitry can become the critical path. This data can also be found in Appendix A.

Enlarging the cache depth has a similar affect on speed as it did in area. The direct-mapped and two-way set associative caches maintain a relatively consistent frequency while the associative caches suffer from significant speed degradation. Speeds reach as low as 52.12 MHz for a 1024 word associative cache. This reduction in speed was expected since the critical path is certainly in the encoder and wide logical OR circuits of the associative cache. Since both of these circuits increase with cache depth, it was expected that the fmax would fall. Unfortunately, the two different latencies produce converging frequencies, rendering this option ineffective for large depths. Initially, the associative cache with a 3 cycle read latency is 40 MHz faster for a 32 word cache. However it quickly decreases and converges until the two are within 12 MHz as seen in Figure 10. This is also expected since the optional registers only isolate lookups in individual CAM blocks, an unfortunate by-product of using the output registers of RAM blocks as the optional registers. As the depth grows, the time to perform an individual lookup does not increase though the encoding step does. The convergence arises since as the encoding stage becomes slower, the savings, which are fixed, become less significant relative to the time it takes to perform the encoding.

None of this is seen in either the direct-mapped or two-way set associative caches. The critical path for both these caches is the comparator, which has no dependence on cache depth. The fmax starts to decrease slightly as the design is dispersed over the chip, causing the inputs to travel greater distances, but otherwise remains relatively consistent. One important note is that the 4096 word direct-mapped cache implemented using the single Mega RAM outperforms the caches of the same type with depths as low as 1024.

Consequently, it seems the Mega RAM implementation improves the speed of the system despite the fact that the Mega RAM is in fact slower than the smaller memory blocks.

The effect of increasing address width has a devastating effect on all caches. It increases CAM size, as well as the comparators, hence directly affecting the critical path of all the cache designs. The aforementioned convergence also occurs here, but for all cache designs, resulting in maximum and minimum speeds of 95.06 MHz and 59.8 MHz for address widths 512 bits wide. Fortunately, widths this large are never used. In fact, the Pentium 4 has only a 36-bit address space [8]. However, the results do show that increases in the tag directly affect the fmax of the system.

Increasing the data width does not alter the critical path. The only circuitry affected are the muxes at the data ports of the data store. Hence, increases in data width do not adversely affect the speed of the system. As seen in Figure 12, the caches maintain a relatively steady fmax for data widths of up to 128 bits. However a sudden decrease is observed for all caches when the data width reaches 256 bits, which is the same width as the bus on a Pentium 4 to the L2 cache [8]. With this width, it was observed that the critical path had shifted and now included the data store and muxes. It was also observed that the delay resulted largely from logic elements being forced in locations far away from the RAM. This is a fundamental property of any digital system, namely that even parts of the system which are not in the critical path can expand enough that they become the critical path.

Overall, a significant amount of information can be drawn from these results which can aid designers in selecting appropriate cache parameters. First, the purpose behind the existence of direct-mapped and set associative caches, namely that they reduce

area and speed up the design, are clearly exemplified in the data presented. The direct-mapped and two way set associative caches are both smaller and faster, particularly when comparing caches with the same latencies. Evidently, designs which require small and effective caches can use associative caches, but when larger caches are needed, designers have little choice but to use set associative or direct-mapped caches. The decision between these latter two is less obvious. It is a matter of how eager a designer is for an effective cache. For example, Altera's newest soft processor, Nios version 3.0, uses a direct-mapped cache while the Pentium 4 uses an 8-way set associative L2 cache. These decisions are highly subjective and application specific.

The adjustable latencies allows for a finer grain of tuning. Read latencies can vary from 1 to 3 cycles while adding no additional area to the designs. Systems which can tolerate an additional wait state can use this to replace a single cycle direct-mapped cache for a more effective, and also much faster, two cycle two-way cache. Similar tradeoffs can be made between two way and fully associative caches, though at the expense of additional area.

With respect to choosing cache dimensions, cache depth can be expanded to the target chip's capacity without significant penalties in LE area and speed for non-fully associative caches. It was also seen that the most influential dimension of a cache is the address width. Any increase in tag size will directly increase area and decrease fmax. Growth in the data width will certainly increase area but in general doesn't change the speed of the system.

5 Conclusion

An automatic cache generator was crafted which emits Stratix specific cache designs in a Verilog output file. The input to the generator is a set of parameters describing the desired cache. Inputs include associativity, latency, cache depth, address width, and data width. Cache designs were generated and evaluated for a wide variety of input parameters. From this analysis, a number of trends were established concerning their area and speed. Overall, it can be concluded that the generator is very robust offering a wide variety of cache types, each able to satisfy a wide variety of size and speed constraints.

A few future modifications can be made to improve the generator. First is support for accessing subsections of a cache line using the byte enable lines of the memory blocks. A second improvement is support for the more popular 4-way set associative cache. Another modification is the possibility of pipelining cached reads. Finally, investigation into Mega RAM implementations for set associative caches may be of interest.

References

[1] Altera Corporation. “Altera Stratix FPGA Family Data Sheet”. December 2002, http://www.altera.com/literature/ds/ds_stx.pdf

[2] Altera Corporation, “AN 119: Implementing High-Speed Search Applications with Altera CAM,” in Altera Application Notes. July 2001, http://www.altera.com/literature/an/an119.pdf

[3] J.L. Baer, Department of Computer Science & Engineering, University of Washington. “2K papers on caches by Y2K: Do we need more?”. November 2000, http://www.irit.fr/ACTIVITES/EQ_APARA/HPCA6/BaerHpca6.PDF

[4] J.L. Brelet. “Using Block RAM for High Performance Read/Write CAMs” in Xilinx Application Note xapp204, May 2000.

[5] S. Brown, and Z. Vranesic. “Fundamentals of Digital Logic with Verilog Design”. McGraw-Hill, 2003.

[6] J. L. Hennessy and D. A. Patterson. “Computer Architecture: A Quantitative Approach, 3rd edition”. Morgan Kaufmann Publishers, 2003.

[7] C. Hamacher, Z. Vranesic, and S. Zaky. “Computer Organization”, 5th edition. McGraw-Hill, 2002.

[8] Intel Corporation. “Intel® Pentium® 4 Processor with 512-KB L2 Cache on 0.13 Micron Process at 2 GHz – 3.06 GHz, with Support for Hyper-Threading Technology1 at 3.06 GHz Datasheet”. January 2003, ftp://download.intel.com/design/Pentium4/datashts/29864307.pdf

Appendix A – Data Gathered

Note: R*x* W*y* refers to a cache with a read latency of x and write latency of y. Also, data for the memory usage was given in the body of the thesis and will not be repeated here.

Table A.1: Measurements of LE usage for different cache depths

Depth	*Associative*	*Direct Mapped*	*Two Way Set Associative*
32	235	97	156
128	575	97	156
256	1049	97	155
1024	3736	99	155
2048	7318	99	154
4096	14438	98	155
Slope	3.53	0.00	0.00

Table A.2: Measurements of LE usage for different address widths

Address Width	*Associative*	*Direct Mapped*	*Two Way Set Associative*
32	234	97	156
128	521	164	284
256	905	247	455
512	1673	417	796
Slope	3.00	0.667	1.33

Table A.3: Measurements of LE usage for different data widths

Data Width	*Associative*	*Direct Mapped*	*Two Way Set Associative*
32	234	97	156
64	298	161	252
92	354	217	336
128	426	289	444
256	682	545	828
Slope	2.00	2.00	3.00

Table A.4: Measurements of fmax for different cache depths

	Associative		*Direct Mapped*		*Two Way Set Associative*			
Depth	*R2 W1*	*R3 W1*	*R1 W1*	*R2 W1*	*R1 W1*	*R1 W2*	*R2 W1*	*R2 W2*
32	123.51	163.45	144.84	210.08	115.09	118.79	175.93	172.86
128	91.75	109.4	145.05	215.47	126.04	119.7	176.43	177.97
256	80.75	99.8	149.39	218.91	122.43	121.77	171.35	176.34
1024	52.12	65.23	140.19	206.78	121.4	118.37	162.53	152.93
2048	No fit	No fit	139.3	195.2	120.4	113.29	147.97	150.44
4096	No fit	No fit	133.28	199.44	116.02	111.74	134.66	162.95

Table A.5: Measurements of fmax for different address widths

	Associative		*Direct Mapped*		*Two Way Set Associative*			
Addr	*R2 W1*	*R3 W1*	*R1 W1*	*R2 W1*	*R1 W1*	*R1 W2*	*R2 W1*	*R2 W2*
32	123.51	163.45	144.84	210.08	115.09	118.79	175.93	172.86
128	94.69	125.36	123.78	155.86	104.67	105.23	138.77	145.24
256	86.87	113.51	108.31	149.39	93.33	95.76	127.58	124.69
512	59.8	75.49	81.32	95.06	67.7	66.18	93.61	86.91

Table A.6: Measurements of fmax for different data widths

	Associative		*Direct Mapped*		*Two Way Set Associative*			
Data	*R2 W1*	*R3 W1*	*R1 W1*	*R2 W1*	*R1 W1*	*R1 W2*	*R2 W1*	*R2 W2*
32	123.51	163.45	144.84	210.08	115.09	118.79	175.93	172.86
64	117.34	141.34	142.94	198.22	114.08	115.93	165.87	170.88
92	112.66	135.15	139.97	192.72	113.3	116.18	170.44	171.7
128	107.41	137.25	132.33	193.42	110.49	113.06	166	161.73
256	90.61	108.71	119.98	163.4	94.26	95.98	146.74	141.98

Table A.7: Speeds of a minute cache

	Associative		*Direct Mapped*		*Two Way Set Associative*			
Dimensions	*R2 W1*	*R3 W1*	*R1 W1*	*R2 W1*	*R1 W1*	*R1 W2*	*R2 W1*	*R2 W2*
16x10x8	162.47	212.92	215.33	328.41	172.21	167.01	215.61	271.12

Appendix B – Sample Waveform

Test of Associative Cache

1. Testing of basic read/write possibilities. Note that the read write operations need only be signaled for the first cycle of the operation. Since this is an R2 W1 cache, reads will complete two cycles later (if cached), and writes will complete one cycle later. Also, note that the cache_hit signal is valid only for the second cycle of an operation. Finally, note that M_ is prefixed to signals from the master (the processor), and S_ is prefixed to signals to the slave (the memory).

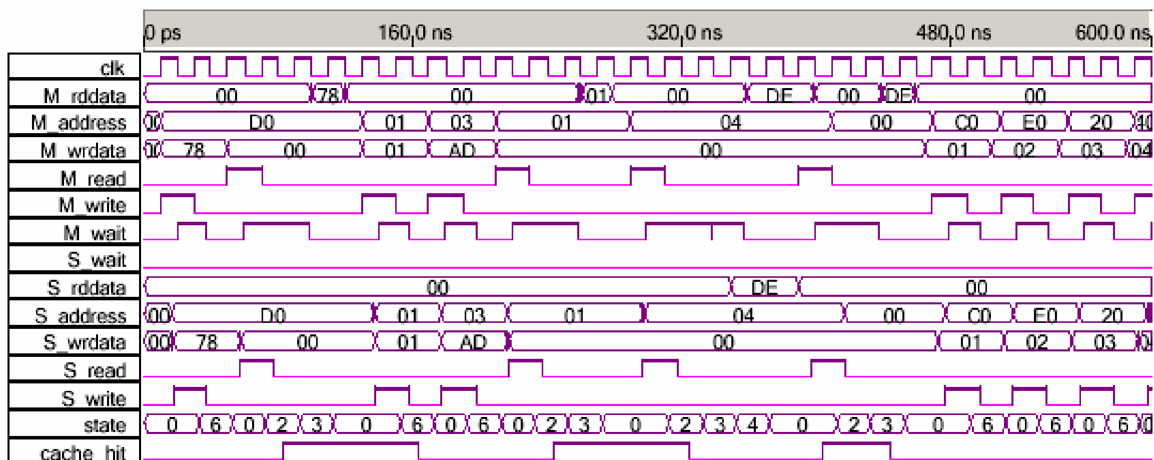

Figure B.1: Waveform of sample cache test

2. Continuing, the associative cache is filled and it is ensured old values were kicked out. Proof: Notice the last read operation results in a cache miss even though it was cached above.

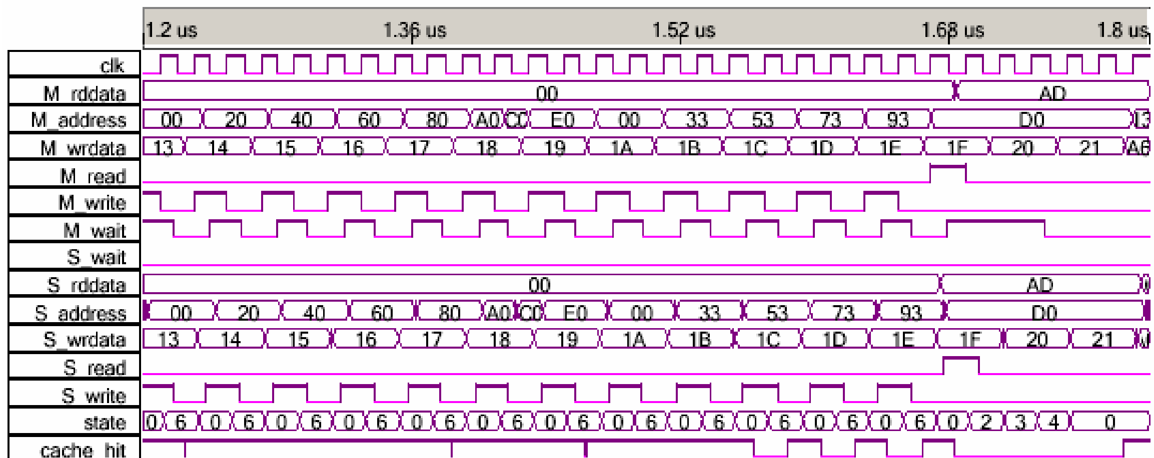

Figure B.2: Continued waveform of sample cache test

Appendix C – Code

All source code, scripts, and sample outputs are on the included CD-ROM. Also, all caches measured in the results section of this thesis can be found in the sample outputs.