A VHDL Design of a Reconfigurable Cache Memory System

Tiffany M. Brooks

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation
https://louis.uah.edu/honors-capstones/82

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
A VHDL Design of a Reconfigurable Cache Memory System

Tiffany Brooks
EE 412-02
May 8, 1995
ABSTRACT

The goal of this project was to model a reconfigurable cache using VHDL. A reconfigurable cache memory would allow users to dynamically choose the best cache mapping technique to suit their needs. After considering several different approaches to this problem, the best plan seemed to be to design the control for an existing VHDL model of a cache system. The cache model could be modified to work with the control. This paper gives background information on cache memory systems including the principle of locality, placement policies for mapping words into the cache, and the issue of cache coherency. It describes the design decisions that were made for the reconfigurable cache in the context of this background information. It also includes information on the design of the FSM and the VHDL code that was generated from the KISS description. Finally, it includes suggestions for further research into this area.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>The Basic Idea Behind a Cache Memory</td>
<td>1</td>
</tr>
<tr>
<td>Cache Hits and Misses and the Principle of Locality</td>
<td>2</td>
</tr>
<tr>
<td>Placement Policies for Data Within the Cache</td>
<td>3</td>
</tr>
<tr>
<td>Direct Mapping</td>
<td>4</td>
</tr>
<tr>
<td>Set Associative Mapping</td>
<td>5</td>
</tr>
<tr>
<td>Fully Associative Mapping</td>
<td>6</td>
</tr>
<tr>
<td>Mapping Blocks That Contain More Than One Word</td>
<td>7</td>
</tr>
<tr>
<td>Design Decisions Regarding Associativity</td>
<td>8</td>
</tr>
<tr>
<td>Write-Through, Write-Back, and Write-Around</td>
<td>9</td>
</tr>
<tr>
<td>Read Requests and Write Requests</td>
<td>10</td>
</tr>
<tr>
<td>Read Requests</td>
<td>10</td>
</tr>
<tr>
<td>Write Requests</td>
<td>11</td>
</tr>
<tr>
<td>Design of the FSM Control for Read Requests Write Requests</td>
<td>12</td>
</tr>
<tr>
<td>Conclusions</td>
<td>12</td>
</tr>
<tr>
<td>References</td>
<td>13</td>
</tr>
<tr>
<td>Bibliography</td>
<td>14</td>
</tr>
<tr>
<td>Appendix A: VHDL Code for the Cache from the Internet</td>
<td>A1</td>
</tr>
<tr>
<td>Appendix B: KISS Description and Notes on KISS Model</td>
<td>B1</td>
</tr>
<tr>
<td>Appendix C: VHDL Code Generated from KISS</td>
<td>C1</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: Example of Block Placement in a Direct-Mapped Cache 5
Figure 2: Example of Block Placement in a 2-Way Set-Associative Cache 6
Figure 3: Example of Block Placement in a Fully-Associative Cache 7
Figure 4: The Arrangement of Memory Blocks in the Reconfigurable Design 9
INTRODUCTION

The goal of this project was to model a reconfigurable cache memory using VHDL. This model could then be used to evaluate issues of performance in a bus environment in which multi-word transfers occur. After considering several different approaches to this problem, the most viable option seemed to be to design the cache control for an existing VHDL model of a cache system. The cache model could then be modified to work with the control. A VHDL model of a cache memory was found on the Internet, and this model is given in Appendix A. The control was designed as a Finite State Machine (FSM) using KISS, and the KISS model was then converted into VHDL.

VHDL is an acronym for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. VHDL was chosen to model the cache and the cache control because using a hardware description language is easier and less expensive than building physical hardware would be. It also models the performance of hardware more closely than software could. This is because VHDL has an aspect of concurrency that software cannot emulate.

There are basically three different placement techniques used to map words into the cache. These are direct, set-associative, and fully-associative mapping. The best placement technique for a particular application will depend on the access patterns of the program being executed. For example, if a user was running a program that executed a large number of loops, the best mapping technique might be a direct-mapped design. If, on the other hand, the user was running a program that dealt with a large number of arrays, a set-associative design might be the best. The advantage of having a reconfigurable cache is that the user can dynamically chose the best technique to suit his or her needs.

This paper gives background information on cache memory systems including the principle of locality, placement policies for mapping words into the cache, and the issue of cache coherency. It describes the design decisions that were made for the reconfigurable cache in this context. It also includes information on the design of the FSM and the VHDL code that was generated from the KISS description. Finally, it includes suggestions for further research into this area.

THE BASIC IDEA BEHIND A CACHE MEMORY

One of the main problems associated with the use of large main memories involves the fact that these memories are relatively slow when compared to the speed of the processor. The main memory is typically constructed of slower, less expensive IC devices, while the processor uses higher speed flip-flops and gates (Bartee, 362). If a computer system uses only the main memory to access data and instructions, then the speed of the processor will be limited by the main
One technique to reduce the access time associated with the use of main memory involves the addition of a cache memory to the processor, and this results in a more efficient memory system.

A cache memory is used to store the instructions and data that are the most frequently accessed by the processor. One of the best ways to illustrate why cache memories are useful is through the use of a library analogy (Patterson and Hennessy, 518). Suppose you were in the library working on a research paper about Eleanor of Aquitaine. You would be sitting at a desk and would have several books spread out in front of you that contain information about Eleanor's life. Suppose you are particularly interested in Eleanor's involvement with the Crusades, but none of the books in front of you contain information about this subject. In order to find more information about the Eleanor's connection with the Crusades, you would have to go back to the stacks to look for additional books. Once you have a good selection of books, most of the information you will need will be on the desk in front of you where it can be easily and quickly accessed. You will spend most of the time using the books in front of you without having to go back to the stacks.

In the library analogy, you act as the processor, the desk acts as the cache, and the stacks act as main memory. It is easy to see that if you (the processor) need a piece of information that is located on the desk (the cache), then this information can be accessed very quickly. If the information is not located on the desk, then you will have to get it from the stacks (main memory) and bring it back to the desk. Accessing books that are located in the stacks takes more time than accessing information on the desk. However, the stacks have the advantage of being able to hold more information than the desk can. There is a limit to the number of books the desk can hold.

CACHE HITS AND MISSES AND THE PRINCIPLE OF LOCALITY

Cache memory is based on the principle of locality, which states that future patterns of access will be similar to patterns that have occurred in the recent past. At any given time, programs will access only a small portion of their address space, just as in the library analogy, you would access only a small number of the books located in the library. There are two types of locality: temporal locality, or locality in time, and spatial locality, or locality in space. The principle of temporal locality states that if an item has been referenced in the recent past, it is likely that it will be accessed in the near future. One type of programming construct that displays temporal locality is a loop in which instructions are executed repeatedly (Baron and Higbie, 197). The principle of spatial locality states that if a certain piece of data or a certain instruction has been accessed in the recent past, it is likely that its neighbors will be accessed in the near future. Spatial locality is displayed by many computer programs, in which instructions are executed
sequentially, and by arrays, in which data is typically accessed sequentially (Baron and Higbie, 197).

In systems that use a memory system which includes a cache, the cache starts out empty, and a section of the program being executed is loaded into the cache. Memory requests are then first presented to the cache, and a cache hit occurs when the word being accessed is in the cache. The hit time is the time it takes to access the cache, including the time it takes to determine whether or not the data being accessed is located in the cache. The main memory is referenced only when the instruction or the data being accessed is not present in the cache. A cache miss occurs when the word is not located in the cache, and the main memory must be referenced. In the case of a cache miss, the word being accessed is retrieved from the main memory and copied into the cache. The miss penalty is the time it takes to copy a block from main memory into the cache, including the time it takes to provide the block to the processor. Because the block being referenced must be fetched from main memory in the case of a cache miss, the miss penalty is typically much larger than the hit time (Baron and Higbie, 197-198).

The hit ratio, $h$, is defined to be the probability that a reference to memory will be a hit, and the miss ratio, $m$, is the probability that the reference will be a miss. These metrics are usually experimentally determined. Since each cache reference is either a hit or a miss, the hit ratio can be found by the equation $h = (1 - m)$. If a computer system had a hit ratio of 1, then the cache would contain all of the instructions and data that needed to be referenced, and the computer would operate at the speed of the processor. If the hit ratio were 0, then the computer would operate at the speed of the main memory. Since the cache is typically 5 to 40 times faster than the main memory, it is good to have a high hit ratio (Bartee, 362).

Increasing the hit ratio is an important aspect of cache design, and the hit ratio can be increased by using a larger cache. However, there are definite costs, in both time and hardware, associated with raising the hit ratio by simply making the cache larger (Bartee, 362). Some computer systems use separate caches for instructions and data. Using a general purpose cache equal to the combined sizes of the instruction and data caches will usually result in a higher hit ratio. However, if a general purpose cache cannot supply both instructions and data in one clock cycle, and this can affect the overall performance of the system (Patterson and Hennessy, 468).

**PLACEMENT POLICIES FOR DATA WITHIN THE CACHE**

One of the most important issues in the design of cache memory systems involves the technique used to map blocks into the cache. There are basically three different placement methods used to map blocks to a cache addresses, and
these are direct, set associative, and fully associative mapping. Each method has advantages and disadvantages associated with it because the different mapping techniques involve tradeoffs between the hit ratio, the hit time, the miss penalty, and the cost of implementing the cache.

Direct Mapping

The easiest method of placing words into the cache is by using direct mapping. In a direct mapped cache, each memory address is mapped into exactly one location in the cache. In this type of design, multiple words from main memory can be mapped to the same cache location, although only one of these words is present in the cache at a given time. The minimum unit of information that is dealt with is a called a block, and blocks can contain one word or multiple words. To illustrate how a direct mapped cache works, assume that the words in memory are 64 bits long, memory addresses are 24 bits long, and that the cache contains 256 \((2^8)\) entries. In this example, one word blocks are mapped into the cache. The cache address is found by using the following equation:

\[
\text{block number} = (\text{block address}) \mod (\text{number of entries in the cache})
\]

If memory address \((28,561)_{10}\) is being mapped into the cache, the cache address is computed by \((28,561) \mod (256) = 145\). One of the advantages of using this type of mapping is that the cache address can be found by inspection if the cache size is a power of 2. The cache address can be determined by simply looking at the lower order \(\log_2 (\text{cache size in blocks})\) bits in the address (Patterson and Hennessy, 460). In this case, the address \((28,561)_{10}\) is \((0000 0000 01 10 11 11 1001 0001)_2\) and the cache address is \((1001 0001)_2\). The cache address is simply the lower 8 bits of the address. Figure 1 shows the contents of the cache after this word is placed into it.

When the processor needs to access a word in memory, the address presented to the cache will be broken down into two parts. The 8 least significant bits are referred to as the block number, and these bits are stored in the address register of the cache. Since multiple words from memory can be mapped into the same cache location, a tag is also needed to identify which word is located in the cache. The 16 most significant bits are the tag, and these bits are stored in a flip-flop register (Bartee, 364). In this case, each tag will have 16 \((24 - 8 = 16)\) bits. To determine whether a word is located within the cache, the cache control reads the word which is located at the address represented by the block number. It then determines whether or not the tag matches the 16 most significant bits of the address in main memory. If the tags match, a cache hit occurs, and the 64 bits of the word are transferred to the processor. If the tags are not the same, then a cache miss occurs, and the word must be obtained from the main memory.

In direct mapped cache, there is only one place that a particular address can be mapped. Therefore, there is only one cache address that needs to be
Figure 1: Example of Block Placement in a Direct-Mapped Cache

referenced to see whether a cache hit occurs. This results in a very simple replacement policy for a direct mapped cache. In a direct mapped implementation, the cache has the word that was most recently used by the processor (Patterson and Hennessy, 461).

Set Associative Mapping

The main problem with direct mapped cache memory occurs when two or more words that are frequently used are mapped to the same address in the cache. This is called a conflict miss. A conflict miss occurs in this example when the nine least significant bits of the words are the same, and it causes these words to be continually replaced with each other (Patterson and Hennessy, 513). The most common way of avoiding this problem is the use of a set associative cache, which has more than one tag at each location in the cache. In a set associative cache, each address is mapped into an unique set in the cache, but it can be placed in different locations within the set. An n-way associative cache has n different locations within the set that a particular address can be mapped to. Typically, there are two or four tags associated with each set. To illustrate how a set associative cache works, assume that the words in memory are 64 bits long, memory addresses are 24 bits long, the cache contains a total of 256 (2^8) entries, and that the address is being mapped into a 2-way set associative cache. In this example, one word blocks are mapped into the cache. The set number is found by the following equation:

\[
\text{set number} = (\text{block address}) \mod (\text{number of sets in the cache})
\]

If memory address (28,561) \text{,10} is being mapped into the 2-way set associative cache, the cache address is computed by (28,561) \mod (256/2) = 145. The number of set is 128 (256/2 = 128) because there are two locations within the set to which an address can be mapped. Again, if the address space in the cache is a
power of two, then the set number can be found by examining the lower order bits of the address that is being referenced. In this case, the address \((28,561)_{10}\) is \((0000 0000 0110 1111 1001 0001)_2\) and the cache address is \((001 0001)_2\). The cache address is simply the lower 7 bits of the address because these are \(2^7\) sets in the cache. The tag would be the upper 17 bits of the address. Figure 2 shows the contents of the cache after this word is placed into it.

The use of a set associative cache reduces the problems encountered when more than one word is mapped to the same address in the cache. As associativity increases, the miss rate for set associative caches decreases. However, there is a cost associated with increasing the associativity. Because there are multiple entries for each block in the cache, all of the tags that are associated with a particular cache address must be examined to see whether a cache hit occurs. This results in increased access time for the cache, as well as increased cost in hardware (van de Goor, 417).

**Fully Associative Mapping**

The third placement technique is fully associative mapping. In a fully associative cache, data can be mapped into any position within the cache. If the cache contains \(256 (2^8)\) entries, the address can be mapped into any cache address between 0 and 255. Figure 3 shows the possible placement of address \((28,561)_{10}\) in the cache. The increase in associativity associated with this type of mapping results in a fully associative cache having the highest hit rate of any of the placement policies. However, since each address in the cache must be examined to determine whether or not a hit has occurred, the fully associative cache has the highest cost in terms of both time and hardware. The cost of using a fully associative type of mapping may decrease in the future with the use of VLSI associative memories (van de Goor, 417).

---

<table>
<thead>
<tr>
<th>Set #</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0000</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>000 0001</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>001 0001</td>
<td>Y</td>
<td>0000 0000 0110 1111 1 Mem [6F9116]</td>
<td></td>
</tr>
<tr>
<td>111 1111</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Example of Block Placement in a 2-Way Set-Associative Cache
In addition to the cache address, the tag, and the data, each cache entry can also have a valid bit associated with it. The valid bit is used to determine whether or not the cache block contains valid information. In systems that use a cache, the cache starts out empty, and the data and tag fields will be meaningless. As information is loaded into the cache, the valid bit is set to 1 to show that the block contains valid data. If the bit is set to 1, then the cache entry is valid and the data can be accessed. If the valid bit is set to 0, then there cannot be a cache hit because the data will be invalid. The valid bit can also be used in associative memories to reduce the time associated with checking cache entries. For example, it would not be necessary to check all entries in a fully associative cache if some of the entries were not valid (Patterson and Hennessy, 460-461).

**Mapping Blocks That Contain More Than One Word**

In the previous examples, the placement techniques have taken advantage of temporal locality by mapping one word blocks into the cache. However, these examples have not been able to take advantage of spatial locality. Spatial locality can be exploited by increasing the number of words per block. To illustrate how this works, assume that the words in memory are 64 bits long, memory addresses are 24 bits long, and that the cache contains 256 ($2^8$) blocks. Also assume that the cache uses direct mapping with four word blocks. The cache address is found by using the following equation:

$$\text{block number} = (\text{block address}) \mod (\text{number of entries in the cache})$$

This equation is the same as for the previous examples, but the block address is different. The block address can be found by the following equation:

$$\text{block address} = \left\lfloor \frac{\text{word address}}{\text{number of words/block}} \right\rfloor$$
If memory address $(28,561)_{10}$ is being mapped into the cache, the block address is computed by the floor function of $(28,561) / (4)$, which is equal to $7140$. The block number is $(7140) \mod (256) = 228$. The cache address can be determined by looking at the lower order bits in the address if the word offset is taken into account. In this case, the word offset is two bits, because there are four words associated with each block ($4 = 2^2$). The cache address can be found by looking at bits 9 through 2, which are the lower eight bits in the cache address. Bits 1 and 0 are not included because they are used for the word offset. In this case, the address $(28,561)_{10}$ is $(0000\ 0000\ 0110\ 1111\ 1001\ 0001)_2$ and the cache address is $(11\ 1001\ 00)_2$. The tag is the upper 10 bits of the address. When memory address $(28,561)_{10}$ address is mapped into the cache, addresses $(28,560)_{10}$, $(28,562)_{10}$, and $(28,563)_{10}$ are mapped into the cache at the same time. Pulling multi-word blocks into the cache takes advantage of spatial locality, because the neighbors of the word being accessed are also mapped into the cache.

**DESIGN DECISIONS REGARDING ASSOCIATIVITY**

The best mapping technique for a particular application will depend on the access patterns of the program being executed. For example, if a user was running a program that executed a large number of loops, the best mapping technique might be a direct-mapped design. If, on the other hand, the user was running a program that dealt with a large number of arrays, a set-associative design might be the best. The advantage of having a reconfigurable cache is that the user can dynamically chose the best technique to suit his or her needs.

In the design of the control for the reconfigurable cache, several assumptions were made. One word was assumed to be four bytes, the total data capacity for the cache was assumed to be 2048 ($2^{11}$) bytes, and main memory was assumed to be contain 64K words ($64K = 2^{16}$), which means that the addresses were 16 bits long. Additionally, the maximum associativity for the cache was assumed to be four. Therefore, the design needed to be configurable as a 1-way (or direct-mapped), 2-way, or 4-way set-associative cache.

Because the cache needed to accommodate a 4-way set associative type of mapping, there needed to be four banks of memory located in the cache. Since the size of the cache would not change, this established the number of blocks that the cache would contain. The cache was assumed to contain 512 entries ($2^{11} / 2^2 = 2^9$). Since there are four ($2^2$) positions in each set, the total number of blocks was determined to be 128 ($2^9 / 2^2 = 2^7 = 128$). After the number of blocks was determined, the tag length was calculated. The tag length was fixed to be 7 bits ($16 - 7 - 2 = 9$). Since the memory was assumed to be byte addressable, the two least significant bits were used for the byte offset.

Since there were four banks of memory in the cache, the 2-way configuration should have two word blocks associated with it, and the 1-way
configuration should use four word blocks. However, since the 2-way configuration had one bit of word offset associated with it, the tags for this configuration would only be 6 bits long. Similarly, since the 1-way configuration had four words associated with it, there were two bits of word offset that had to be taken into account, so the associated tag would only be 5 bits long. However, the tags needed to be the same length for each configuration. This problem was resolved by letting the word offset become part of the tag. This gave a 7 bit tag for each of the three cases. Figure 4 shows the arrangement of the four memory banks.

WRITE-THROUGH, WRITE-BACK, AND WRITE-AROUND

At this point, it would be helpful to explain the importance of cache coherency. In any computer system, it is important that both the main memory and the cache contain the same data. The simplest way to ensure cache coherency is with write-through. When a write hit occurs in systems that use write-through, both the cache and main memory are written. However, write-through does not provide good performance, especially in cases where one word is written a number of times (Baron and Higbie, 198).

An alternative to write-through is write-back. In systems that use write-back, only the cache is written in the case of a cache hit. If the word in the cache is replaced and that word has been written, it will be copied back to main memory at this time. In systems that use write-back, a dirty bit is used to determine whether or not the word in the cache is the same as the word in main memory. When a word is placed in the cache, the dirty bit is set to 0. If the word is altered while it is in the cache, then the dirty bit is set to 1. Therefore, if a cache miss occurs, and the dirty bit of the least recently used word is equal to 0, it is not necessary to re-copy this word from the cache back into the main memory. The
least recently used word can simply be discarded in this case (Baron and Higbie, 198).

There is a third type of cache that uses a write-around policy. This type of cache is similar to the write-through cache, with the difference between the two occurring in the case of a write miss. If a write miss occurs, a write-around cache will simply write the data to memory without copying the data into the cache. The VHDL model that is given in Appendix A can use this type of policy or it can use write-back. However, the write-around cache is called write-through in the model, and in the following description of read and write requests, write-around is also called write-through.

**READ REQUESTS AND WRITE REQUESTS**

There are two types of requests that can be presented to the cache: read requests and write requests. For each type of request, either a cache hit or a cache miss will occur. In designing the control for the reconfigurable cache, it was necessary to examine the sequence of events that take place in each of these four cases. Also, since the original VHDL cache design includes an option for having either a write-through or write-back cache, this had to be taken into account in the design of the control.

**Read Requests**

Initially, the cache control is in an initial state, and in this state, the control receives a read request. If the cache is using write-through and a cache hit occurs, then the word is supplied to the processor and the cache control returns to its original state. If a cache miss occurs, the word is copied into the cache from main memory. The placement of the word in memory depends on whether the cache is configured as a 1-way, 2-way, or 4-way set-associative cache. If the cache is configured to be a 1-way set-associative cache, then there is no need to select a victim to replace. This is because there is only one position within the cache to which a block can be mapped. In the case of a cache miss, four words are written into the cache. The words will be mapped into the cache starting with position 0 and ending with position 3.

For the 2-way set-associative case, there are two locations within the set to which a word can be mapped. Since a word can be mapped into more than one spot, one of the locations within the set must be discarded. The selection of which data to discard is known as victim selection. In this particular design, the location within the set is chosen randomly. There are two possibilities that can be chosen; therefore a single bit may be used to determine the location of the block within the set. If the random number generator supplies a 0, then the two words are mapped into positions 0 and 1. If the random number generator supplies a 1, then the two words are mapped into positions 2 and 3 within the set.
Finally, for the 4-way set-associative case, there are four locations within the set to which a word can be mapped. In this case, a victim must be selected. There are four possibilities that may be chosen; therefore two bits should be used to determine the location of the block within the set. If the random number generator supplies a 00, then the word is mapped into position 0. If the random number generator supplies a 01, then the word is mapped into position 1. If the random number generator supplies a 10, then the word is mapped into position 2. Finally, if the random number generator supplies a 11, then the word is mapped into position 3.

After the word or words are mapped into the cache, the control will return to its original state. Since the read request has not yet been filled, the word must still be supplied to the processor. At this point, the process will start over. This time a cache hit will occur because the word will now be present within the cache. The word will then be supplied to the processor and the cache control will return to its original state.

If the cache receives a read request and is using the write-back policy, then the sequence of events that occur is slightly different than the write-through case. If a cache hit occurs, then the word is supplied to the processor and the cache control returns to its original state. If a cache miss occurs, the word is copied into the cache from main memory. Again, the placement of the word in memory depends on whether the cache is configured as a 1-way, 2-way, or 4-way set-associative cache. The same basic sequence of events occurs as in the write-through case with one exception. After the victim is selected in the 2-way or 4-way set-associative case, or after the cache address is computed in the 1-way set-associative case, then the word or words which are being replaced must be checked to see whether or not they are consistent with memory. If the dirty bit is set to 0, then the data may simply be discarded. If the dirty bit is set to 1, then the words must be copied back into memory. After the data is either copied back or discarded, the dirty bit is reset, and the word or words can be copied into the cache as in the write-through case. The control will then return to its original state. The word will then be supplied to the processor because the word will be present in the cache. Finally, the cache control will return to its original state.

**Write Requests**

If the cache control is in the initial state and it receives a write request, then either a write hit or a write miss will occur. Again, the sequence of events that takes place depends on whether a write-through or write-back policy is being used. If the cache is using write-through and a cache hit occurs, then the word is written into the cache and then written to memory. If a cache miss occurs, the word is written to memory only. If write-back is being used and a cache hit occurs, then the word is written to the cache only and the dirty bit is set to 1. If a cache miss occurs, then the word or words will be fetched from memory, with victim
selection occurring as described above, and the word will then be written into the cache. At this point, the cache control will return to its initial state.

Design of the FSM Control for Read and Write Requests

In designing the control for the reconfigurable cache, it was necessary to determine the sequence of events that take place in each of four cases for both a write-through and write-back implementation. After the steps in each case were determined, it was necessary to assign states to each of the events that occurred. Each state was a part of the FSM design of the cache control. In order to implement the control, KISS was used. The KISS description for the FSM and notes on this model are given in Appendix B. Appendix C contains the VHDL code that was generated from this description.

CONCLUSIONS

The addition of a cache memory to a computer system helps to reduce the access time associated with the use of the main memory. By using the principle of locality and storing the data and instructions that are used most frequently, the cache provides rapid access to these items, and this can greatly improve the performance of computer systems. The use of direct mapped cache memories, set associative cache memories, and fully associative cache memories result in more efficient memory systems. However, since the best placement technique for a particular application will depend on the access patterns of the program being executed, it would be useful for the user to dynamically choose the best technique to suit his or her needs. This is the advantage of having a reconfigurable cache.

The best approach to this problem seemed to be to design the control for an existing VHDL model of a cache system. The cache model could then be modified to work with the control. The control was designed as a Finite State Machine (FSM) using KISS, and the KISS model was then converted into VHDL. Unfortunately, the control was not added to the existing VHDL model of the cache because of time constraints. Further research into this area would include making this addition, and there are several other possible areas that could be explored with this model. These would include increasing the maximum associativity to include the option of having a fully-associative cache, allowing the maximum associativity to be variable, and writing a program or modeling control to choose which technique would provide the best performance based on the access patterns of the program being executed. Also, the completed model could be used in other research areas; for example, it could be used to evaluate performance issues in bus environments in which multi-word transfers occur.
REFERENCES


BIBLIOGRAPHY


Appendix A: VHDL Code for the Cache from the Internet
package cache_types is

  type strategy_type is (write_through, copy_back);

end cache_types;
Entity declaration for cache.

use work.dlx_types.all,
  work.mem_types.all,
  work.cache_types.all;

entity cache is
  generic (cache_size : positive;          -- in bytes, power of 2
    line_size : positive;                  -- in bytes, power of 2
    associativity : positive;             -- 1 = direct mapped
    write_strategy : strategy_type;       -- write_through or copy_back
    Tpd_clk_out : Time;                   -- clock to output propagation delay
    tag : string := "";                   --
    origin_x, origin_y : real := 0.0);     --

port (phil, phi2 : in bit;                -- 2-phase non-overlapping clocks
  reset : in bit;                         -- synchronous reset input
  -- connections to CPU
  cpu_enable : in bit;                    -- starts memory cycle
  cpu_width : in mem_width;               -- byte/halfword/word indicator
  cpu_write : in bit;                     -- selects read or write cycle
  cpu_ready : out bit;                    -- status from memory system
  cpu_a : in dlx_address;                 -- address bus output,
  cpu_d : inout dlx_word_bus bus;         -- bidirectional data bus
  -- connections to memory
  mem_enable : out bit;                   -- starts memory cycle
  mem_width : out mem_width;              -- byte/halfword/word indicator
  mem_write : out bit;                    -- selects read or write cycle
  mem_burst : out bit;                    -- tells memory to burst txfer
  mem_ready : in bit;                     -- status from memory system
  mem_a : out dlx_address;                -- address bus output
  mem_d : inout dlx_word_bus bus);        -- bidirectional data bus

end cache;
Behavioural architecture for cache.

use work.bv_arithmetic.bv_to_natural,
   work.bv_arithmetic.natural_to_bv;

architecture behaviour of cache is

begin  --  behaviour

cacheBehaviour : process

constant words_per_line : positive := line_size / 4;
constant number_of_sets : positive := cache_size / line_size / associativity;

subtype word_offset_range is natural range 0 to words_per_line-1;
subtype entry_index_range is natural range 0 to associativity-1;
subtype set_index_range is natural range 0 to number_of_sets-1;

type line is array (word_offset_range) of dlx_word;

type entry is record
   tag : natural;
   valid : boolean;
   dirty : boolean;
   data : line;
end record;

type store_array is array (set_index_range, entry_index_range) of entry;

variable store : store_array;
variable cpu_address : natural;
variable word_offset : word_offset_range;
variable set_index : set_index_range;
variable cpu_tag : natural;
variable entry_index : entry_index_range;
variable hit : boolean;
variable next_replacement_entry_index : entry_index_range := 0;

procedure do_read_hit is
begin
    cpu_d <= store(set_index, entry_index).data(word_offset);
    cpu_ready <= '1' after Tpd_clk_out;
    wait until phi2 = '0';
    cpu_d <= null after Tpd_clk_out;
    cpu_ready <= '0' after Tpd_clk_out;
end do_read_hit;

procedure do_write_through is
begin
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    mem_a <= cpu_a after Tpd_clk_out;
    mem_width <= cpu_width after Tpd_clk_out;
    mem_d <= cpu_d after Tpd_clk_out;
    mem_write <= '1' after Tpd_clk_out;
    mem_burst <= '0' after Tpd_clk_out;
    mem_enable <= '1' after Tpd_clk_out;
    wait until mem_ready = '1' or reset = '1';
    cpu_ready <= mem_ready after Tpd_clk_out;
    wait until phi2 = '0';
    mem_d <= null after Tpd_clk_out;
    mem_write <= '0' after Tpd_clk_out;
    mem_enable <= '0' after Tpd_clk_out;
    cpu_ready <= '0' after Tpd_clk_out;
end do_write_through;

procedure do_write_hit is
begin
    case cpu_width is
    when width_word =>
        store(set_index, entry_index).data(word_offset) := cpu_d;
    when width_halfword =>
        if cpu_a(1) = '0' then -- ms half word
            store(set_index, entry_index).data(word_offset)(0 to 15) := cpu_d(0 to 15);
        else -- ls half word
            store(set_index, entry_index).data(word_offset)(16 to 23) := cpu_d(16 to 23);
        end if;
    when width_byte =>
        if cpu_a(1) = '0' then -- ms half word
            if cpu_a(0) = '0' then -- byte 0
                store(set_index, entry_index).data(word_offset)(0 to 7) := cpu_d(0 to 7);
            else -- byte 1
                store(set_index, entry_index).data(word_offset)(8 to 15) := cpu_d(8 to 15);
            end if;
        else -- ls half word
            if cpu_a(0) = '0' then -- byte 2
                store(set_index, entry_index).data(word_offset)(16 to 23) := cpu_d(16 to 23);
            else -- byte 3
                store(set_index, entry_index).data(word_offset)(24 to 31) := cpu_d(24 to 31);
            end if;
        end if;
    end case;
    if write_strategy = copy_back then
        store(set_index, entry_index).dirty := true;
    end if;
    -- if write_through cache, also update main memory
if write_strategy = write_through then
    do_write_through;
else -- copy_back cache
    cpu_ready <= '1' after Tpd_clk_out;
    wait until phi2 = '0';
    cpu_ready <= '0' after Tpd_clk_out;
end if;
end do_write_hit;

procedure copy_back_line is
    variable next_address : natural;
    variable old_word_offset : natural;
begin
    next_address := (store(set_index, entry_index).tag * number_of_sets + set_index) * line_size;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    mem_width <= width_word after Tpd_clk_out;
    mem_write <= '1' after Tpd_clk_out;
    mem_enable <= '1' after Tpd_clk_out;
    mem_burst <= '1' after Tpd_clk_out;
    old_word_offset := 0;
    burst_loop : loop
        if old_word_offset = words_per_line-1 then
            mem_burst <= '0' after Tpd_clk_out;
        end if;
        mem_a <= natural_to_bv(next_address, mem_a'length) after Tpd_clk_out;
        mem_d <= store(set_index, entry_index).data(old_word_offset) after Tpd_clk_out;
        wait_loop : loop
            wait until phi2 = '0';
            exit burst_loop when reset = '1'
                or (mem_ready = '1' and old_word_offset = words_per_line-1);
            exit wait_loop when mem_ready = '1';
        end loop wait_loop;
        old_word_offset := old_word_offset + 1;
    next_address := next_address + 4;
end loop burst_loop;
    store(set_index, entry_index).dirty := false;
    mem_d <= null after Tpd_clk_out;
    mem_write <= '0' after Tpd_clk_out;
    mem_enable <= '0' after Tpd_clk_out;
end copy_back_line;

procedure fetch_line is
    variable next_address : natural;
    variable new_word_offset : natural;
begin
    next_address := (cpu_address / line_size) * line_size;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    mem_width <= width_word after Tpd_clk_out;
    mem_write <= '0' after Tpd_clk_out;
    mem_enable <= '1' after Tpd_clk_out;
    mem_burst <= '1' after Tpd_clk_out;
    new_word_offset := 0;
    burst_loop : loop
        if new_word_offset = words_per_line-1 then
            mem_burst <= '0' after Tpd_clk_out;
        end if;
        mem_a <= natural_to_bv(next_address, mem_a'length) after Tpd_clk_out;
wait_loop : loop
  wait until phi2 = '0';
  store(set_index, entry_index).data(new_word_offset) := mem_d;
  exit burst_loop when reset = '1'
    or (mem_ready = '1' and new_word_offset = words_per_line-1);
  exit wait_loop when mem_ready = '1';
  end loop wait_loop;
  new_word_offset := new_word_offset + 1;
  next_address := next_address + 4;
  end loop burst_loop;
  store(set_index, entry_index).valid := true;
  store(set_index, entry_index).tag := cpu_tag;
  store(set_index, entry_index).dirty := false;
  mem_enable <= '0' after Tpd_clk_out;
end fetch_line;

procedure replace_line is
begin
  -- first chose an entry using "random" number generator
  entry_index := nextReplacement_entry_index;
  nextReplacement_entry_index := (nextReplacement_entry_index + 1) mod associativity;
  if store(set_index, entry_index).dirty then
    copy_back_line;
  end if;
  fetch line;
end replace_line;

procedure do_read_miss is
begin
  replace_line;
  if reset = '1' then
    return;
  end if;
  do_read_hit;
end do_read_miss;

procedure do_write_miss is
begin
  -- if write_through cache, just update main memory
  if write_strategy = write_through then
    do_write_through;
  else -- copy_back cache
    replace_line;
    if reset = '1' then
      return;
    end if;
    do_write_hit;
  end if;
end do_write_miss;

begin -- process cache behaviour
  -- reset: initialize outputs and the cache store valid bits
  cpu_ready <= '0';
  cpu_d <= null;
  mem_enable <= '0';
  mem_width <= width_word;
  mem_write <= '0';
  mem_burst <= '0';
  mem_a <= x"00000000";
  mem_d <= null;
  for init_set_index in set_index_range loop
for init_entry_index in entry_index_range loop
    store(init_set_index, init_entry_index).valid := false;
    store(init_set_index, init_entry_index).dirty := false;
end loop; -- init_entry_index
end loop; -- init_set_index

-- wait for a cpu request
wait until phi2 = '1' and cpu_enable = '1';
-- decode address
cpu_address := bv_to_natural(cpu_a);
word_offset := (cpu_address mod line_size) / 4;
set_index := (cpu_address / line_size) mod number_of_sets;
cpu_tag := cpu_address / line_size / number_of_sets;
-- check for hit:
hit := false;
for lookup_entry_index in entry_index_range loop
    if store(set_index, lookup_entry_index).valid
        and store(set_index, lookup_entry_index).tag = cpu_tag then
        hit := true;
        entry_index := lookup_entry_index;
        exit;
    end if;
end loop; -- lookup_entry

if hit then
    if cpu_write = '1' then
        do_write_hit;
    else
        do_read_hit;
    end if;
else
    if cpu_write = '1' then
        do_write_miss;
    else
        do_read_miss;
    end if;
end if;
exit when reset = '1';
end loop;
-- loop exited on reset: wait until it goes inactive
-- then start again
wait until phi2 = '0' and reset = '0';
end process cacheBehaviour;

end behaviour;
Appendix B: KISS Description and Notes on KISS Model
state 0 → cache control is waiting for a request
state 1 → receives a read request, write through
   if hit = 1, then supply word = 1, back to state 0
   if hit = 0, then go to state 3
state 2 → receives a read request, write back
   if hit = 1 and dirty = 0, then supply word = 1, back to state 0
   if hit = 0, go to state 3
state 3 → miss has occurred
   if way - 2 or way - 4, go to state 4, out random_gen
   if way - 1 go to state 5
state 4 → have generated one random bit for victim selection
   if way - 2 go to state 5
   if way - 4 go to state 5, out random_gen
state 5 → if dirty = 0 go to state 7, 8, 9, or 10
   write data to cache
   if dirty = 1 go to state 6
   have to store data in memory
state 6 → have a cache miss w/ victim selected and dirty = 1
   write word to memory, go to state
state 7 → have a cache miss w/ victim selected and dirty = 0
If miss, go to SHAKE 3
  if hit = 1, write-cache, dirty = 1, go to SHAKE 0
  if hit = 0, then write-mem, go to SHAKE 2
  if hit = 1, then write-cache, to SHAKE 12
  if hit = 0, then write-mem = 1, back to SHAKE 0

If request, write-through
  go to SHAKE 0, cache control
  write 0 = cache command
  waiting for a request

write FSM

Write states for write
Appendix C: VHDL Code Generated from KISS
LIBRARY IEEE, ARITHMETIC;
   USE IEEE.std_logic_1164.all;
   USE ARITHMETIC.std_logic_arith.all;
--
-- Suggested state assignments
-- Using assignment: spectral
--
-- s0 0b0000
-- part0 0b0001
-- s11 0b1000
-- s12 0b1100
-- s13 0b1010
-- s3 0b1001
-- s4 0b1101
-- s5 0b1011
-- s7 0b0100
-- s8 0b1110
-- s9 0b0110
-- s10 0b0101
-- s6 0b1111
ENTITY min_tbrooks IS
   PORT(
      clock: IN std_logic;
      dirty: IN std_logic;
      hit: IN std_logic;
      random0: IN std_logic;
      random1: IN std_logic;
      read: IN std_logic;
      way_1: IN std_logic;
      way_2: IN std_logic;
      way_4: IN std_logic;
      write: IN std_logic;
      write_back: IN std_logic;
      dirty_en: OUT std_logic;
      random_gen1: OUT std_logic;
      random_gen2: OUT std_logic;
      read_cache: OUT std_logic;
      write_cache: OUT std_logic;
      write_en0: OUT std_logic;
      write_en1: OUT std_logic;
      write_en2: OUT std_logic;
      write_en3: OUT std_logic;
      write_mem: OUT std_logic
   );
END min_tbrooks;
ARCHITECTURE rtl OF min_tbrooks IS
   SUBTYPE IntState IS bit_vector (0 to 3);
   SIGNAL CurState, NextState: IntState;
BEGIN
-- State transition on clock
StateTransition: PROCESS (clock )
BEGIN
   IF (clock'EVENT AND clock = '1' AND clock'LAST_VALUE = '0') THEN
      CurState <= NextState;
   END IF;
END PROCESS;

-- combinational block
ComBlock: PROCESS (CurState,
dirty,
hit,
random0,
random1,
read,
way_1,
way_2,
way_4,
write,
write_back)
VARIABLE intern dirty_en: std_logic;
VARIABLE intern_random_gen1: std_logic;
VARIABLE intern_random_gen2: std_logic;
VARIABLE intern_read_cache: std_logic;
VARIABLE intern_write_cache: std_logic;
VARIABLE intern_write_en0: std_logic;
VARIABLE intern_write_en1: std_logic;
VARIABLE intern_write_en2: std_logic;
VARIABLE intern_write_en3: std_logic;
VARIABLE intern_write_mem: std_logic;
VARIABLE intern_NextState: InitState;
BEGIN
intern_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0000";
CASE CurState IS
WHEN "0000" =>
IF read = '0' AND
write = '1' AND
write_back = '0'
THEN
intern_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "1000";
ELSIF read = '0' AND
write = '1' AND
write_back = '1'
THEN
intern_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "1010";

ELSIF read = '1' AND write = '0' AND write_back = '0'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0001";

ELSIF read = '1' AND write = '0' AND write_back = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0000";
END IF;

WHEN "0001" =>
  IF hit = '0'
  THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "1001";
  ELSIF hit = '1'
  THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "0001";
  END IF;

WHEN "1000" =>

C4
IF hit = '0'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '1';
    intern_nextState := "0000";
ELSIF hit = '1'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '1';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_nextState := "1100";
END IF;

WHEN "1100" =>
    IF hit = '1'
    THEN
        intern_dirty_en := '0';
        intern_random_gen1 := '0';
        intern_random_gen2 := '0';
        intern_read_cache := '0';
        intern_write_cache := '0';
        intern_write_en0 := '0';
        intern_write_en1 := '0';
        intern_write_en2 := '0';
        intern_write_en3 := '0';
        intern_write_mem := '1';
        intern_nextState := "0000";
    END IF;

WHEN "1010" =>
    IF hit = '0'
    THEN
        intern_dirty_en := '0';
        intern_random_gen1 := '0';
        intern_random_gen2 := '0';
        intern_read_cache := '0';
        intern_write_cache := '0';
        intern_write_en0 := '0';
        intern_write_en1 := '0';
        intern_write_en2 := '0';
        intern_write_en3 := '0';
        intern_write_mem := '0';
        intern_nextState := "1001";
    ELSIF hit = '1'
    THEN
        intern_dirty_en := '1';
        intern_random_gen1 := '0';
        intern_random_gen2 := '0';
        intern_read_cache := '0';
        intern_write_cache := '1';
        intern_write_en0 := '0';
        intern_write_en1 := '0';
        intern_write_en2 := '0';
        intern_write_en3 := '0';
        intern_write_mem := '0';
        intern_nextState := "1001";
    END IF;

BEGIN IF;

WHEN "1001" =>
IF way_1 = '0' AND
way_2 = '0' AND
way_4 = '1'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '1';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "1101";
ELSIF way_1 = '0' AND
way_2 = '1' AND
way_4 = '0'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '1';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "1101";
ELSIF way_1 = '1' AND
way_2 = '0' AND
way_4 = '0'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "1011";
END IF;

WHEN "1101" =>
IF way_1 = '0' AND
way_2 = '0' AND
way_4 = '1'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "1011";
END IF;

END IF;
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "1011";

ELSIF way_1 = '0' AND way_2 = '1' AND way_4 = '0'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "1011";
END IF;

WHEN "1011" =>
IF dirty = '0' AND random0 = '0' AND random1 = '0' AND way_1 = '0' AND way_2 = '0' AND way_4 = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '1';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0100";
ELSIF dirty = '0' AND random0 = '0' AND random1 = '1' AND way_1 = '0' AND way_2 = '0' AND way_4 = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '1';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0110";
ELSIF dirty = '0' AND random0 = '1' AND random1 = '0' AND way_1 = '0' AND way_2 = '0' AND way_4 = '1'
THEN
intern_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '1';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0110";

ELSIF dirty = '0' AND
    random0 = '1' AND
    random1 = '1' AND
    way_1 = '0' AND
    way_2 = '0' AND
    way_4 = '1'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '1';
    intern_write_mem := '0';
    intern_NextState := "0101";

ELSIF dirty = '0' AND
    random1 = '0' AND
    way_1 = '0' AND
    way_2 = '1' AND
    way_4 = '0'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '1';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "0100";

ELSIF dirty = '0' AND
    random1 = '1' AND
    way_1 = '0' AND
    way_2 = '1' AND
    way_4 = '0'
THEN
    intern_dirty_en := '0';
    intern_random_gen1 := '0';
    intern_random_gen2 := '0';
    intern_read_cache := '0';
    intern_write_cache := '0';
    intern_write_en0 := '0';
    intern_write_en1 := '0';
    intern_write_en2 := '0';
    intern_write_en3 := '0';
    intern_write_mem := '0';
    intern_NextState := "0110";

ELSIF dirty = '0' AND
    way_1 = '1' AND
    way_2 = '0' AND
    way_2 = '0' AND
    way_4 = '0'
way_4 = '0'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '1';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_next_state := "0100";
ELSIF dirty = '1' AND
  way_1 = '0' AND
  way_2 = '0' AND
  way_4 = '1' AND
  write_back = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '1';
  intern_next_state := "1111";
ELSIF dirty = '1' AND
  way_1 = '0' AND
  way_2 = '1' AND
  way_4 = '0' AND
  write_back = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '1';
  intern_next_state := "1111";
ELSIF dirty = '1' AND
  way_1 = '1' AND
  way_2 = '0' AND
  way_4 = '0' AND
  write_back = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '1';
  intern_next_state := "1111";
END IF;
WHEN "0100" =>
IF random0 = '0' AND random1 = '1' AND
way_1 = '0' AND way_2 = '0' AND way_4 = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0101";
ELSIF random1 = '0' AND
way_1 = '0' AND way_2 = '1' AND way_4 = '0'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '1';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "0101";
ELSIF way_1 = '1' AND
way_2 = '0' AND way_4 = '0'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '1';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
  intern_NextState := "1110";
END IF;

WHEN "1110" =>
IF random0 = '1' AND random1 = '0' AND
way_1 = '0' AND way_2 = '0' AND way_4 = '1'
THEN
  intern_dirty_en := '0';
  intern_random_gen1 := '0';
  intern_random_gen2 := '0';
  intern_read_cache := '0';
  intern_write_cache := '0';
  intern_write_en0 := '0';
  intern_write_en1 := '0';
  intern_write_en2 := '0';
  intern_write_en3 := '0';
  intern_write_mem := '0';
END IF;
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0101";
ELSIF
random1 = '0' AND
way_1 = '0' AND
way_2 = '1' AND
way_4 = '0'
THEN
intem_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0101";
ELSIF way_1 = '1' AND
way_2 = '0' AND
way_4 = '0'
THEN
intem_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '1';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0110";
END IF;

WHEN "0110" =>
IF
random0 = '1' AND
random1 = '1' AND
way_1 = '0' AND
way_2 = '0' AND
way_4 = '1'
THEN
intem_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0110";
ELSIF
random1 = '1' AND
way_1 = '0' AND
way_2 = '1' AND
way_4 = '0'
THEN
intem_dirty_en := '0';
intern_random_gen1 := '0';
intern_random_gen2 := '0';
intern_read_cache := '0';
intern_write_cache := '0';
intern_write_en0 := '0';
intern_write_en1 := '0';
intern_write_en2 := '0';
intern_write_en3 := '0';
intern_write_mem := '0';
intern_NextState := "0101";
END IF;

C11
intern_write_en2 := '0';
intern_write_en3 := '1';
intern_write_mem := '0';
intern_NextState := "0101";
ELSIF way_1 = '1' AND
   way_2 = '0' AND
   way_4 = '0'
THEN
   intern_dirty_en := '0';
   intern_random_gen1 := '0';
   intern_random_gen2 := '0';
   intern_read_cache := '0';
   intern_write_cache := '0';
   intern_write_en0 := '0';
   intern_write_en1 := '0';
   intern_write_en2 := '0';
   intern_write_en3 := '1';
   intern_write_mem := '0';
   intern_NextState := "0101";END IF;

WHEN "0101" =>
IF TRUE -- true under any input condition
THEN
   intern_dirty_en := '0';
   intern_random_gen1 := '0';
   intern_random_gen2 := '0';
   intern_read_cache := '0';
   intern_write_cache := '0';
   intern_write_en0 := '0';
   intern_write_en1 := '0';
   intern_write_en2 := '0';
   intern_write_en3 := '0';
   intern_write_mem := '0';
   intern_NextState := "0000";END IF;

WHEN "1111" =>
IF TRUE -- true under any input condition
THEN
   intern_dirty_en := '1';
   intern_random_gen1 := '0';
   intern_random_gen2 := '0';
   intern_read_cache := '0';
   intern_write_cache := '0';
   intern_write_en0 := '0';
   intern_write_en1 := '0';
   intern_write_en2 := '0';
   intern_write_en3 := '0';
   intern_write_mem := '0';
   intern_NextState := "1011";END IF;

WHEN others =>
   intern_dirty_en := 'X';
   intern_random_gen1 := 'X';
   intern_random_gen2 := 'X';
   intern_read_cache := 'X';
   intern_write_cache := 'X';
   intern_write_en0 := 'X';
   intern_write_en1 := 'X';
   intern_write_en2 := 'X';
   intern_write_en3 := 'X';
   intern_write_mem := 'X';
END CASE;
dirty_en <= intern_dirty_en;
random_gen1 <= intern_random_gen1;
random_gen2 <= intern_random_gen2;
read_cache <= intern_read_cache;
write_cache <= intern_write_cache;
write_en0 <= intern_write_en0;
write_en1 <= intern_write_en1;
write_en2 <= intern_write_en2;
write_en3 <= intern_write_en3;
write_mem <= intern_write_mem;
NextState <= intern_NextState;
END PROCESS;
END rtl;
HONORS SENIOR PROJECT APPROVAL FORM

(To be submitted by the student to the Honors Program with a copy of the Honors Project suitable for binding. All signatures must be obtained.)

Name of Candidate: Tiffany M. Brooks

Department: Electrical Engineering

Degree: BSE

Full Title of Project: A VHDL Design of a Reconfigurable Cache Memory System

Approved by:

[Signatures and dates]

Honors Program Director for Honors Council