UNTRUSTED MEMORY AUTHENTICATION USING HASH TREE ENGINE

A Project
Presented to the faculty of the Department of Computer Engineering
California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

by
Andrew Ryan Tester

SPRING 2015
UNTRUSTED MEMORY AUTHENTICATION USING HASH TREE ENGINE

A Project

by

Andrew Ryan Tester

Approved by:

__________________________________, Committee Chair
Dr. Nikrouz Faroughi

__________________________________, Second Reader
Dr. Ted Krovetz

Date
Student:  Andrew Ryan Tester

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

__________________________, Graduate Coordinator  
Dr. Nikrouz Faroughi  

Department of Computer Engineering
Abstract

of

UNTRUSTED MEMORY AUTHENTICATION USING HASH TREE ENGINE

by

Andrew Ryan Tester

In this project a hardware Hash Tree Engine (HTE) embedded within a secure processor is modeled and simulated that verifies the authenticity of untrusted memory accessed by the processor. The HTE maintains a Merkle Hash Tree to attest the integrity of data accessed from a range of untrusted memory addresses. The project investigates the worst case time required to authenticate a cache block accessed from the untrusted memory. The HTE, which uses the SHA 256 hashing algorithm and a separate cache memory, is modeled and simulated using Verilog HDL. The nodes of the hash tree and data are assumed to be stored in the untrusted memory space. The root hash of the hash tree, however, resides inside the trusted boundary of the secure processor package.

_______________________, Committee Chair
Dr. Nikrouz Faroughi

_______________________
Date
# TABLE OF CONTENTS

| List of Tables | viii |
| List of Figures | ix  |

Chapter

1. INTRODUCTION ..............................................................................1
   1.1 Merkle Hash Tree .......................................................................2
   1.2 Untrusted RAM and the Trusted Boundary ......................................4

2. ARCHITECTURE ...........................................................................6
   2.1 Hash Tree Engine .........................................................................7
   2.2 SHA 256 .....................................................................................8
   2.3 HTE Hash Cache .........................................................................11

3. DESIGN VALIDATION .................................................................12
   3.1 SHA Module Validation ..............................................................12
   3.2 Hash Tree Engine Validation .......................................................15

4. RESULTS AND ANALYSIS .........................................................30
   4.1 Simulation Results .................................................................31

5. CONCLUSION ............................................................................32
   5.1 Completing the HTE .................................................................32
   5.2 Expanding to Larger Memory Spaces .........................................32

Appendix – Verilog Source Code .........................................................33
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hte.v</td>
<td>The Main Source File of the Hash Tree Engine</td>
<td>33</td>
</tr>
<tr>
<td>cache.v</td>
<td>The Fully Associative HTE Cache Module</td>
<td>43</td>
</tr>
<tr>
<td>sha.v</td>
<td>The SHA 256 Hashing Module</td>
<td>50</td>
</tr>
<tr>
<td>tb3.v</td>
<td>Test Bench File Used to Validate HTE</td>
<td>55</td>
</tr>
</tbody>
</table>

References ........................................................................................................... 65
LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1. Simulation results in terms of ns and the number of clock cycles</td>
<td>31</td>
</tr>
</tbody>
</table>

viii
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Example of a 4-ary Merkle Hash Tree</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Hash Tree Example Using XOR as the Hash Function</td>
<td>3</td>
</tr>
<tr>
<td>Figure 3</td>
<td>HTE Block Diagram</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Example of Loading the Second Chunk into the Dual-Port Block RAMs</td>
<td>9</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Computation of first filler word where ( i = 16 )</td>
<td>10</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Data Block without Preprocessing Data</td>
<td>13</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Testing of the SHA module</td>
<td>14</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Data Block Hash from Hash Utility</td>
<td>15</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Partial Hash Tree Representation – Part 1</td>
<td>18</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Partial Hash Tree Representation – Part 2</td>
<td>19</td>
</tr>
<tr>
<td>Figure 11</td>
<td>L2 Cache Begins Loading of Data Block from Untrusted RAM (Simulation Timing Diagram)</td>
<td>20</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Loading the Level 1 Hash Node from Hash RAM (Simulation Timing Diagram)</td>
<td>21</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Data Hash Compared to Stored Hash in Level 1 (Simulation Timing Diagram)</td>
<td>22</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Comparison of Level 1 Node's Hash to Stored Hash in Level 2 (Simulation Timing Diagram)</td>
<td>23</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Level 2 Node's Hash Compared to Stored Hash in Level 3 (Simulation Timing Diagram)</td>
<td>24</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Level 3 Node's Hash Compared to Stored Hash in Level 4 (Simulation Timing Diagram)</td>
<td>25</td>
</tr>
</tbody>
</table>
Figure 17. Level 4 Hash Compared To Root Hash (Simulation Timing Diagram)......... 26

Figure 18. Mismatch Detected in Level 4 Hash Node (Simulation Timing Diagram)..... 27

Figure 19. Hashing of the Second Data Block Begins (Simulation Timing Diagram)..... 28

Figure 20. Mismatch Detected in Second Data Block (Simulation Timing Diagram)..... 29
1. INTRODUCTION

In the current technological era, much concern is appropriately given to computer security that aims to protect people’s data in three ways, confidentiality, availability and integrity. Confidentiality ensures that a user’s private information is not accessible to unauthorized parties. Protecting availability means that a user should not be denied access due to a denial of service (DoS) attack. Integrity assures that data has not been altered either by malware over a network connection, or directly by attackers having physical access to the system. While it is not always possible to prevent data tampering, there are ways to detect attacks and generate an alert when tampering has occurred. This project addresses a means of memory authentication.

This report explores the feasibility of creating a hardware based hashing engine, independent of the processing core, in order to detect attacks on system RAM. A hash engine detects any unauthorized modification made to data stored in main memory by parsing a Merkle Hash Tree.
1.1 Merkle Hash Tree

A Merkel Hash Tree uses a tree of hashes to protect the integrity of memory data blocks. Figure 1 illustrates an example of a tree where the parent node has 4 "children," making it a "4-ary" (quaternary) tree (Gassend, 2003). A tree where each node has two children nodes is called a binary tree.

All data blocks in a protected section of untrusted memory are hashed and the resulting hash values form the lowest-level nodes of the hash tree. These nodes are themselves hashed and stored into a higher level parent node, up to the single, top-level Secure Root Hash (SRH). The SRH is stored within the processor package, the protected boundary of the system.

Each time a data block is validated, its hash value is calculated and checked against the hash stored in the corresponding parent node in the tree. The hash of the parent node is then computed and checked against the hash stored in its own parent node. This process continues until the hash of the root node is compared with the SRH. If the computed hash of the root node matches the SRH the block is considered valid.
Figure 1 - Example of a 4-ary Merkle Hash Tree

Figure 2 illustrates an example of a binary hash tree where an 8-bit bitwise XOR (exclusive OR) is used as the hash function. In the figure, each data block holds two bytes of data, except for the root hash.

Figure 2 - Hash Tree Example Using XOR as the Hash Function
For example, consider the validation of the block containing the 16-bit hex data "AAE2." To compute the 8-bit hash of the block, AA is hashed (XORed) with E2 to generate 48, which matches the hash stored in the Level 1 hash tree node. Next, the hash of the Level 1 node is checked against its parent hash by hashing its contents, 48 and 80, to produce C8. This matches the C8 stored in the Level 2 node. Next, the hash values 1F and C8 in the Level 2 node are hashed. The result is D7 which matches the value stored as the SRH. Because all of these nodes' computed hashes match their parent hashes stored in the hash tree, the data block is considered authenticated.

The entire main memory or a critical section of it could be protected by a hash tree. This enables the computer to execute security sensitive programs securely in a protected section of the memory, as the hash tree is used to assert that the programs have not been altered.

1.2 Untrusted RAM and the Trusted Boundary

A key concept in secure computing is a minimum set of secure hardware, firmware and software required for keeping the system secure, known as the Trusted Computing Base (Janssen). In this project, the CPU alone is considered trusted, and thus all data stored inside the CPU package is considered valid and within the trusted boundary. Furthermore, the chip is tamper proof because it would be destroyed if the CPU package were opened.
If the CPU is connected to discrete memory modules by exposed connecting wires, the memory is vulnerable to hardware based attacks. For example, it would be possible for an attacker who has physical access to the device to tap into these wires and monitor them for memory transactions issued by the CPU. The attacker could then substitute data from RAM with malicious data or instructions. Because of this vulnerability, the memory is considered outside of the trusted boundary.

Untrusted memory is also subject to software-based attacks such as buffer overflow attacks. In this case, malicious code "overflows" the allocated memory space for data into instruction space. This is possible when programs do not check that the data size does not exceed the size of the buffer. Malicious code can change, for example, the target address of a jump instruction. This allows the attacker to run malicious code with elevated privileges and obtain unauthorized access to restricted resources. Also, a buffer overflow may disrupt the proper execution of programs or make the system unavailable to the authorized users.
2. ARCHITECTURE

This chapter describes the design and structure of an integrated hardware implementation of a Merkle Tree called the Hash Tree Engine. This includes its major components, the modules used to implement the 256-bit Secure Hashing Algorithm (SHA-256) and the fully-associative HTE Hash Cache. The overall organization of the HTE is shown in Figure 3. The modules above the dashed line in the figure are not modeled in HDL; a test-bench, instead, is used to simulate input and output from the RAM. The HTE and L2 Cache are considered to be within the trusted boundary of the CPU package.

![Figure 3 - HTE Block Diagram](image-url)
2.1 Hash Tree Engine

The HTE design is broken into several cooperating Control Units (CUs). A data block can be captured from the data lines into a buffer while another block is being hashed and authenticated. Without this splitting of the HTE, CPU requests to memory would have to be stalled until the current block is authenticated.

The Data Queue CU monitors the data lines from the untrusted RAM for memory blocks requested by the CPU's L2 Cache. When a transfer is detected, the block is captured into the Data Queue and its address is stored in the Address Queue.

The Data SHA CU will activate the Data SHA when a full block has been loaded. The Data Queue's read enable signal is held high for 64 clock cycles, the number of cycles that a queue write takes.

The Cache Control Unit is initiated when an address is available in the Address Queue. It will request the stored parent hash value from the HTE Cache and check it against the computed hash from the data or hash SHA module. The Cache CU will validate the computed hash at each level of the hash tree. The hash of the highest level node is compared to the Secure Root Hash.

The Hash SHA Reset Control will latch the computed hash into a register when the hashing of a hash node is complete, and then resets the hash module. This makes the hash value available to the Cache CU, enabling another blocks to be hashed without causing a delay.
The Hash SHA Control Unit will initiate the run of the Hash SHA module, in similar fashion to the Data SHA CU. Then it will reset to its idle state when the Hash SHA is raises its done signal. The Hash SHA CU also controls the read enable signal of the Hash Data Queue.

2.2 SHA 256

The SHA 256 module is the first of two major subcomponents of the HTE design. The algorithm’s core hashing routine operates on one chunk of data at a time, which is 16 32-bit words in size. The HTE always sends data to the hash modules four chunks, or 64 words, at a time.

The SHA 256 specification calls for the data, called the message, to be appended with a one and as many zeros as needed to fill the final chunk, leaving the last 64 bits of the final chunk to store the message size. If 64 bits are not available in the final chunk, another chunk is added. The SHA hashing module, however, does not perform this preprocessing and assumes that each message is in the right format for hashing. The SHA algorithm also requires that data be provided in Big Endian format.

Here, SHA 256 module uses a 64-word single port RAM to store the incoming block to be hashed. It also has two dual-port RAMs for the 64-word processing array which the SHA core uses to process the current chunk. The SHA module makes two copies of the chunk from the single port RAM into the first 16 memory locations of these
dual port RAMs. These are written to each array unit simultaneously as shown in Figure 4, which illustrates the transfer for words 16-31 of the block, or Chunk 2.

![Diagram of chunk transfer]

**Figure 4 - Example of Loading the Second Chunk into the Dual-Port Block RAMs. Words 16 through 31 are copied to the first 16 addresses of the dual port RAMs.**

The SHA algorithm calls for the 16 words in the processing array to be extended into 64 words before hashing, using the array filling algorithm shown below. The use of two dual-port RAMs accelerates the computation of the additional 48 words by allowing the HTE to access the four words used by the filling algorithm in one clock cycle. Alternatively, one may use a single-port RAM, which would require four clock cycles to read these four words. Figure 5 shows an example of filling address 16 using the dual-port RAMs.
Figure 5 - Computation of first filler word where $i = 16$. Four previous values in the processing array are used, $w[0]$, $w[1]$, $w[9]$ and $w[14]$. The resulting $w[i]$ sum is stored in a latch until the next clock cycle when it is written back.

Pseudocode for array-filling algorithm (SHA-2 Wikipedia, the free encyclopedia)

```plaintext
for i from 16 to 63
    s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
    s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
    w[i] := w[i-16] + s0 + w[i-7] + s1
endfor
```
2.3 HTE Hash Cache

The second major component the HTE uses is a 32-way fully-associative cache to store recently used hashes. This reduces the need for retrieving hashes from RAM; thus reducing the time required to authenticate some data blocks. Available cache slots are assigned in order until all 32 slots are occupied, and then the least recently allocated block is evicted and made available.

The example below illustrates how the cache would handle the 22-bit memory word address, 0x0000e0. The upper sixteen bits of the RAM address are used by the cache as the tag value. The lower six bits indicate which word within the block has been requested.

Word Address in Hex: 0000e0
Binary Equivalent: 00 0000 0000 0000 1110 0000

Binary Tag: 0000 0000 0000 0011  Binary Address: 10 0000
Hex Tag: 0003  Hex Address: 20

The tag array has 32 16-bit registers to contain these tags. There are also two 32-bit arrays to contain dirty and present bits. To reduce search time, all tag locations are searched simultaneously with a 32-way comparator, so the tag is found in one clock cycle if present.
3. DESIGN VALIDATION

This chapter presents the validation of the Hash Tree Engine. Simulation waveforms are used to illustrate correct operation. The completed SHA 256 module was used to verify incremental results during the design and debugging of the HTE. Validation of the SHA module is presented first before the overall validation of the HTE.

3.1 SHA Module Validation

For the purposes of verifying the SHA 256 design, a 256-byte block of text, which was created using the following quote from Alice’s Adventures in Wonderland, was used as a test data block:

'I quite agree with you,' said the Duchess; 'and the moral of that is--"Be what you would seem to be"--or if you'd like it put more simply--"Never imagine yourself not to be otherwise than what it might appear to others that what you were or might have been was not otherwise than what you had been would have appeared to them to be otherwise."' (Carroll, 1886)

The selected text is repeated until the test data block is 247 bytes in size as shown in hex in Figure 6. This leaves nine bytes available to pad the block and also insert the data size information as required by the SHA algorithm. This block is also used in the verification of the overall HTE design functionality.
For the 247-byte data in Figure 6, the following padding and data size information is needed to create a 256-byte data block required by SHA:

1. 0x80, a 1 followed by 7 zeros as a one byte pad;
2. 0x7B8 (247 * 8 bits/byte) as a 64-bit data size.

Typically, a SHA utility would perform the necessary padding required for processing, but in this project, this padding procedure is not modeled in HDL, as stated earlier. Instead, the SHA test-bench provides the padding and data size information. The following output shows the last few bytes of the test data block generated by the test-bench with 0x80 and 0x7B8 inserted.

```
#4 data=32'h66206e6f;//56  | f no |
#4 data=32'h7420746f;     | t to |
#4 data=32'h20626520;     | be   |
#4 data=32'h6f746865;     | othe |
#4 data=32'h7776973;//60   | rwis |
#4 data=32'h65207480;     | e t  |
#4 data=32'h00000000;     |      |
#4 data=32'h00007b8;      |
```

**Figure 6 - Data Block without Preprocessing Data**

<table>
<thead>
<tr>
<th>Never Imagine.bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 4E 65 76 65 72 20 69 6D 61 67 69 6E 65 20 79 6F</td>
</tr>
<tr>
<td>00000100 75 72 73 74 65 6C 66 20 6F 66 66 66 20 6F 74 20 74</td>
</tr>
<tr>
<td>00000200 20 6F 74 68 65 72 77 69 73 65 20 79 6F 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000300 77 68 61 74 20 69 6D 74 20 6D 67 6E 65 74 20 61 70</td>
</tr>
<tr>
<td>00000400 70 65 61 72 20 74 68 65 64 20 6F 74 68 65 72 73 20 74</td>
</tr>
<tr>
<td>00000500 74 68 61 74 20 77 68 61 74 20 77 68 61 74 20 77 68 61 74</td>
</tr>
<tr>
<td>00000600 65 20 6F 72 20 6D 67 6E 65 20 69 6E 65 20 61 70 74 68 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000700 62 65 65 6E 65 2E 20 67 61 73 20 20 6D 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000800 65 72 77 69 73 65 20 74 68 65 67 6E 65 20 67 61 73 20 74 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000900 70 65 61 72 20 74 68 65 64 20 6F 74 68 65 72 73 20 74 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000A00 75 6E 64 20 68 69 6E 65 20 68 69 6E 65 20 67 61 73 20 74 68 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000B00 64 20 74 68 65 20 74 68 65 67 6E 65 20 67 61 73 20 74 68 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000D00 20 69 6D 67 6E 65 20 67 61 73 20 74 68 65 67 6E 65 20 67 61 73 20 74 68 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000E00 66 20 6E 6F 74 20 74 68 65 67 6E 65 20 67 61 73 20 74 68 65 67 6E 65 20 62 6F 6F</td>
</tr>
<tr>
<td>00000F00 72 77 69 73 65 20 74 68 65 20 67 61 73 20 74 68 65 67 6E 65 20 62 6F 6F</td>
</tr>
</tbody>
</table>
When this padded block is hashed by the SHA module, the result is shown in Figure 7 under the Value column. This is the same SHA 256 hash that the DigitalVolcano© Hash Tool produces for the unpadded block, shown in Figure 8 (DigitalVolcano, 2011).

![Figure 7 - Testing of the SHA module](image-url)
3.2 Hash Tree Engine Validation

In a system with memory authentication, the protected RAM content and hash tree values need to be initialized before the HTE begins authentication of cache blocks accessed from the untrusted memory. One option is to have the HTE compute the entire tree on startup. In this case, firmware securely embedded in the protected chip is invoked to disable the authentication function of the HTE while data blocks are being accessed for the first time. Another option would be for the firmware to initialize all data blocks to zero and also initialize the hash tree with known hash values.
In order to simplify the presentation of the HTE validation, it is assumed that all data blocks, except one, contain only zeros. The exception is the data block that contains the quote text.

The hash value for a zero data block is shown here:

\[
\text{B267799A CDCDFD12 8D459248 B4D32BD3 3FE931BA 4EDE20D7 674520EE BC38A21C}
\]

The hash for the quote data block is shown here:

\[
\text{FACD5137 2E361D67 065C45A6 D4713EE9 F3A998A1 A9D5F1AA 7648530C 3401FAB3}
\]

Therefore, Level 1 of the hash tree contains hash values of data blocks of zeros and the hash of a single data block which has the quote as illustrated in Figure 9. A hash node, which is the same size as a data block, contains a group of eight consecutive hash values.

Likewise, Level 2 hash nodes contain hash values computed for the Level 1 hash nodes as also illustrated in Figure 9. Specifically, the hashes of the Level 1 nodes of data blocks containing only zeros have the value starting with 98C4A6B2. The Level 2 hash of the Level 1 node which contains the hash of the quote data block has the value starting with 69BBC8B6.
The rest of the hash nodes are computed similarly in the tree hierarchy. The final hash (i.e. the Secure Root Hash), whose computed value starts with 049453B5 and is shown in Figure 10, would be stored in the SRH register inside the protected chip.

Figures 9 and 10 are a partial diagram of the hash tree showing only nodes along the path of validation of the quote data block. The full values of the hashes in these figures can be found in the testbench file in the appendix.
Figure 9 - Partial Hash Tree Representation – Part 1
<table>
<thead>
<tr>
<th>DB4E7DE4...</th>
<th>DB4E7DE4...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
<tr>
<td>DB4E7DE4...</td>
<td>DB4E7DE4...</td>
</tr>
</tbody>
</table>

**Secure Root Hash**
049453B5...

Figure 10 - Partial Hash Tree Representation – Part 2
Figure 11 is the simulation output illustrating the Data Queue Control Unit capturing the quote data block. The data starts with the hex value 4E657665, as also shown in Figure 6. The address of the block is latched into the Address Queue. The Cache Control Unit uses the address to request the Level 1 parent hash from the HTE Cache. This request is triggered by raising the r_req signal as shown in the figure. The data_fifo_count, also shown in the figure, indicates the number of words (0 so far) in the data queue.

Figure 11 - L2 Cache Begins Loading of Data Block from Untrusted RAM (Simulation Timing Diagram)
Figure 12 illustrates the HTE Cache retrieving the Level 1 hash node from the hash portion of the RAM. It is also loaded into the Hash Data Queue to be processed by the Hash SHA CU as shown in Figure 3. The counter for the Hash Data Queue is incremented by one as each word is loaded to the queue.

Figure 12 - Loading the Level 1 Hash Node from Hash RAM (Simulation Timing Diagram)
In Figure 13, the hash value computed by the Data SHA is shown to match the stored hash value from the Level 1 node. The figure illustrates the hash computation of the current block and also the comparison to the stored hash, which is read from the HTE Cache and stored in a register called hash_from_cache. The two hash values are then compared in order to authenticate the incoming block.

Figure 13 - Data Hash Compared to Stored Hash in Level 1 (Simulation Timing Diagram)
Figure 14 shows the hash of the Level 1 node being computed and latched in the register hh_reg. Then, the stored hash of the Level 1 node retrieved from the Level 2 node is latched in the register hash_from_cache. Next, as also shown in the figure, a bit in the L_good register is set when the two hash values match. In this case, the stored hash is compared with the hh_reg register. The computed 32 byte hash value, shown as eight four byte words labeled h0 through h7, is shown under the column labeled “Value.” Note that each time there is match between the hash values, a different bit in the L_good register is set.

Figure 14 - Comparison of Level 1 Node's Hash to Stored Hash in Level 2 (Simulation Timing Diagram)
Figure 15 shows the comparison of the Level 2 node’s computed hash to its stored hash. Note that the stored hash of Level 2 node had already been retrieved by the HTE cache when the Hash SHA computes the hash of the Level 1 node. The two hash values match, so the Cache CU marks the Level 2 hash valid. The data block authentication process continues with the HTE fetching the stored hash of the Level 3 node from memory.

Figure 15 - Level 2 Node's Hash Compared to Stored Hash in Level 3 (Simulation Timing Diagram)
Figure 16 illustrates the comparison of the computed hash of the Level 3 node with its stored hash in the Level 4 node. Again, the stored hash was already retrieved by the time the Hash SHA computes the hash of Level 3 node.

Figure 16 - Level 3 Node's Hash Compared to Stored Hash in Level 4
In the Figure 17 timing diagram, the Cache CU compares the computed hash of the Level 4 node to the Secure Root Hash (SRH) stored inside the secure processor. By the time the Cache CU parses the hash tree successfully all the way to the SRH, all the L_good register bits get set as shown in the figure.
Figure 18 illustrates the detection of an unauthorized modification of the hash tree in the untrusted RAM by the HTE. One of the bytes in the hash of the Level 4 node has been modified, causing its hash not to match the hash stored in the SRH register. The HTE raises the error signal, indicating that the CPU should halt the execution of the program.

![Simulation Timing Diagram](image-url)

**Figure 18 - Mismatch Detected in Level 4 Hash Node (Simulation Timing Diagram)**
Figure 19 illustrates the steps used to authenticate the next sequential data block accessed. In this case, because the HTE Cache contains the authenticated hash blocks (nodes), the authentication process is faster. The next block is assumed to contain all zeros, but in this case the second word in the block has been purposely altered in the RAM. The figure illustrates the loading of the incoming next block with the modified (nonzero) second word. As illustrated in the next figure, the block is detected as being invalid.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sdata_epid</td>
<td>0x3a00</td>
<td>8</td>
</tr>
<tr>
<td>data_fra_cour</td>
<td>0x00</td>
<td>12</td>
</tr>
<tr>
<td>sdata[35][0]</td>
<td>0x00000000</td>
<td>3</td>
</tr>
<tr>
<td>shrd_read</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>shrd_write</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sdata_fra[0]</td>
<td>0x00000000</td>
<td>9</td>
</tr>
</tbody>
</table>

**Figure 19 - Hashing of the Second Data Block Begins (Simulation Timing Diagram)**
Figure 20 illustrates the HTE detecting the invalid data block. The computed 32 byte hash indicated as h0 through h7 and shown under column “Value,” does not match the hash retrieved from the Level 1 node, shown as hash_from_cache in the figure. In this case, the HTE raises the error signal and thus halts the authentication process.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_done</td>
<td>0</td>
</tr>
<tr>
<td>dk</td>
<td>1</td>
</tr>
<tr>
<td>h0</td>
<td>0xc5936c53</td>
</tr>
<tr>
<td>h1</td>
<td>0xc33a6165</td>
</tr>
<tr>
<td>h2</td>
<td>0x25b6448</td>
</tr>
<tr>
<td>h3</td>
<td>0x1e4111</td>
</tr>
<tr>
<td>h4</td>
<td>0x1692448</td>
</tr>
<tr>
<td>h5</td>
<td>0xe97dc7f5</td>
</tr>
<tr>
<td>h6</td>
<td>72274172</td>
</tr>
<tr>
<td>h7</td>
<td>c1698e24</td>
</tr>
<tr>
<td>hash_from_cache</td>
<td>[3277...99a, d2f8f82, d6855248, 1b631d03f, 3f18c81a, 66ec3e574520c2c, 3390c1c]</td>
</tr>
<tr>
<td>error</td>
<td>1</td>
</tr>
<tr>
<td>hash_match</td>
<td>0x08000000</td>
</tr>
<tr>
<td>l_point</td>
<td>0x0800</td>
</tr>
<tr>
<td>hash_match_d</td>
<td>0x0800</td>
</tr>
<tr>
<td>hash_countl</td>
<td>0x00</td>
</tr>
<tr>
<td>hash_load</td>
<td>0</td>
</tr>
</tbody>
</table>

Mismatch Detected

Figure 20 - Mismatch Detected in Second Data Block (Simulation Timing Diagram)
4. RESULTS AND ANALYSIS

This chapter discusses the simulation results. The worst case time to authenticate a data block results when none of the hashes along the tree path from the data hash to the highest level hash node are available in the HTE Cache. On the other hand, if any of the hashes that are needed to authenticate a memory block have recently been used and therefore are loaded in the HTE cache, the authentication task will require less time than the worst case.
4.1 Simulation Results

Table 1 presents the simulation results of authenticating a memory data block in terms of clock cycles, which are based on the clock period of 4 nanoseconds used in the simulation of the design. The simulation results also give the times to complete each intermediate step of the authentication process. As indicated in the table, it would take 3318 clock cycles the worst case to authenticate a data block or detect an attack.

<table>
<thead>
<tr>
<th>Event</th>
<th>Figure</th>
<th>Time (in ns)</th>
<th>Time since data block is loaded (ns)</th>
<th>Time since data block loaded (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Block Load</td>
<td>11</td>
<td>160</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Data Block Hashed and Compared</td>
<td>13</td>
<td>3578</td>
<td>3418</td>
<td>855</td>
</tr>
<tr>
<td>Level 1 Node Hash Compared</td>
<td>14</td>
<td>3638</td>
<td>3478</td>
<td>870</td>
</tr>
<tr>
<td>Level 2 Node Hash Compared</td>
<td>15</td>
<td>7068</td>
<td>6908</td>
<td>1727</td>
</tr>
<tr>
<td>Level 3 Node Hash Compared</td>
<td>16</td>
<td>10236</td>
<td>10076</td>
<td>2519</td>
</tr>
<tr>
<td>Level 4 Node Hash Compared With SRH</td>
<td>17</td>
<td>13428</td>
<td>13268</td>
<td>3317</td>
</tr>
<tr>
<td>Level 4 Node Mismatch Detected</td>
<td>18</td>
<td>13430</td>
<td>13270</td>
<td>3318</td>
</tr>
</tbody>
</table>
5. CONCLUSION

The project is an investigation of an embedded system for hashing untrusted memory. The Hash Tree Engine, embedded within the secure processor package, accesses its own dedicated cache memory separate from the cache memories used by the processor.

5.1 Completing the HTE

In order for the design to implement all of the functions needed to perform the full operation of an HTE, additional steps are required. This design is capable of authenticating memory blocks accessed from an untrusted RAM. The next step would be to add the components necessary to update the hash tree when blocks are evicted from the lowest level cache (e.g., L2 cache). An outgoing data block would need to be captured and hashed in similar fashion before it leaves the secure boundary of the processor. A second set of queues, hashers and control units would comprise this other half of the HTE. These two halves of the HTE would run largely independently of each other, except there might be a need for arbitration for use of the HTE Cache.

5.2 Expanding to Larger Memory Spaces

The HTE is designed to monitor only one megabyte of untrusted memory space using an 8-ary hash tree. However, this would scale up quickly as adding four more levels of hashes would make it possible to monitor 4096 MB of untrusted memory, but a larger HTE cache would be needed.
A. Appendix – Verilog Source Code

hte.v – The Main Source File of the Hash Tree Engine

`timescale 1ns / 1ps
///////////////////////////////////////////////////////////////////////
///////////
// Company:
// Engineer: Andrew Tester
//
// Create Date: 12:50:48 11/25/2013
// Design Name:
// Module Name: hte
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 – File Created
// Additional Comments:
//
///////////////////////////////////////////////////////////////////////
///////////
module hte(
    input clk, rst,
    input [17:0] addr_reg,
    input [31:0] fromRAM,
    input ram_read,
    //output [31:0] h_out,
    //input [2:0] sel,
    input [31:0] from_hash_ram,
    output hash_read, error);

wire [17:0] L_addr[0:3];

//reg [17:0] addr_latch;
//reg [19:2] addr_reg;
reg [7:0] word_count_d;
reg [7:0] shad_count;
reg [7:0] shah_count;
reg [3:0] stateA;
wire [3:0] nextstateA;
reg [3:0] stateB;
wire [3:0] nextstateB;
reg [4:0] stateC;
wire [4:0]nextstateC;
reg [3:0]stateD;
wire [3:0]nextstateD;
reg [3:0]stateE;//Parallel to Section A
wire [3:0]nextstateE;
reg [3:0]stateF;//Parallel to Section B
wire [3:0]nextstateF;

reg addr_available;
wire data_fifo_write;
reg data_fifo_rd_en;
wire addr_empty, addr_full;
wire shad_rst, shah_rst;
wire [31:0]data_fifo_data;
wire [31:0]hash_fifo_data;
wire [11:0]data_fifo_count;
wire [11:0]hash_fifo_count;
wire shad_done, shad_rdy;
wire shah_done, shah_rdy;
wire [17:0]curr_addr;
wire [5:0]addr_fifo_count;
reg [1:0]shad_tick;
reg [1:0]shah_tick;
reg shad_load;
reg shah_load;
wire [31:0]hd[0:7];
wire [31:0]hh[0:7];
reg [31:0]hh_reg[0:7];
reg hh_set;
reg hash_avail;
//reg [3:0] hash_i;
reg addr_fifo_rd_en;
reg hash_fifo_rd_en;
reg [2:0]level;
reg r_req;
//wire hash_read;
reg [6:0]hash_word_count;
//wire [3:0]cache_state;
reg [2:0]word_count_c;
reg [6:0]word_count_h;
reg [31:0]hash_from_cache[0:7];
wire [31:0]from_HTE_cache;
wire [0:7]hash_match_d;
wire [0:7]hash_match_h;
wire [0:7]hash_match_r;
wire cache_hit;
reg [0:4]L_good;
reg hte_done;
reg shah_rst_, shad_rst_;
wire cache_done;
reg [31:0]secure_root_hash[0:7];

//instantiate all modules used
fifo_data
data_queue(.clk(clk),.rst(rst),.din(fromRAM),.wr_en(data_fifo_write),
    .empty(addr_empty),.full(addr_full),.dout(data_fifo_data),
    .rd_en(data_fifo_rd_en),.data_count(data_fifo_count));

fifo_addr
addr_queue(.clk(clk),.rst(rst),.wr_en(addr_available),.din(addr_req),
    .dout(curr_addr),.data_count(addr_fifo_count),
    .rd_en(addr_fifo_rd_en));

sha256
hasher_data(.clk(clk),.rst(shad_rst),.data(data_fifo_data),.ldl(1'b0),
    .ld4(shad_load),.done(shad_done),.h0(hd[0]),.h1(hd[1]),.h2(hd[2]),
    .h3(hd[3]),.h4(hd[4]),.h5(hd[5]),.h6(hd[6]),.h7(hd[7]),
    .sha_rdy(shad_rdy));

sha256
hasher_hash(.clk(clk),.rst(shah_rst),.ld4(shah_load),.ldl(1'b0),
    .data(hash_fifo_data),.sha_rdy(shah_rdy),.done(shah_done),
    .h0(hh[0]),.h1(hh[1]),.h2(hh[2]),.h3(hh[3]),.h4(hh[4]),.h5(hh[5]),
    .h6(hh[6]),.h7(hh[7]));

cache
HTE_cache(.rst(rst),.clk(clk),.addrreq(4'b0000,L_addr[level]/*,2'b00*/
    +word_count_c),.w_req(1'b0),.r_req(r_req),.fromram(from_hash_ram),
    .done(cache_done),.hit(cache_hit),.ram_read(hash_read),
    .cache_out(from_HTE_cache));

fifo_data hash_data_queue(.clk(clk),.rst(rst),.din(from_hash_ram),
    .wr_en(hash_fifo_write),.empty(/*addr_empty*/),
    .full(/*addr_full*/),.dout(hash_fifo_data),
    .rd_en(hash_fifo_rd_en),.data_count(hash_fifo_count));

//assign all addresses for hash values given a requested memory location
assign L_addr[3] = {curr_addr[17:15],3'b000}; // 3 bits for word address
assign L_addr[2] = {8 + curr_addr[17:12],3'b000};
assign L_addr[1] = {72 + curr_addr[17:9],3'b000};
assign L_addr[0] = {584 + curr_addr[17:6],3'b000};

assign error = (stateC == 5'h1f);
assign shad_rst = rst | shad_rst_; //SHAs can reset from 2 different signals
assign shah_rst = rst | shah_rst_;

always@(posedge clk)
begin
if(rst)
  begin

stateA <= 0;
stateB <= 0;
stateC <= 0;
stateD <= 0;
stateE <= 0;
stateF <= 0;
//setting secure root hash
secure_root_hash[0]=32'h049453b5;
secure_root_hash[1]=32'h2575052c;
secure_root_hash[2]=32'hf8a8c173;
secure_root_hash[3]=32'he16280c7;
secure_root_hash[4]=32'h1c6ae5f7;
secure_root_hash[5]=32'h9c420334;
secure_root_hash[6]=32'h41f2bc0c;
secure_root_hash[7]=32'ha74e05a4;
end
else
begin
//update state values for all Control Units
stateA <= nextstateA;
stateB <= nextstateB;
stateC <= nextstateC;
stateD <= nextstateD;
stateE <= nextstateE;
stateF <= nextstateF;
end
end
//Data Queue Control
always@(negedge clk)
begin
if(stateA == 0)
begin
//Initialize registers
addr_available <= 0;
word_count_d <= 0;
//data_fifo_rd_en <= 0;
end
else if(stateA == 1)
begin
//shad_tick <= 0;
//shad_load <= 0;
end
else if(stateA == 2)
begin
//addr_latch <= addr_req;
if(word_count_d == 0)
    addr_available <= 1; //Build square wave signal for
else /*if(word_count_d == 1)*/
    addr_available <= 0; // address queue write enable
if(word_count_d < 64) //Count 64 words of data
    word_count_d <= word_count_d + 1;
always@(posedge clk)
begin
  if(stateA == 0)
  begin //Reset State
    shad_count <= 0;
    data_fifo_rd_en <= 0;
  end
  /*else if(stateA == 1)
  begin
    sha_tick <= 0;
  end*/
  else if(stateA == 3)
  begin
    if(shad_tick == 1) //If data SHA has been activated
      data_fifo_rd_en <= 1; //Read enable of Data Queue
    else if(shad_tick == 3 && shad_count < 63) //SHA is running
      shad_count <= shad_count + 1;
    else if(shad_tick == 3 && shad_count == 63)
    begin
      //sha_tick <= 3;
      data_fifo_rd_en <= 0;
    end
  end
end
assign nextstateA = (stateA == 0) ? 1 :
  (stateA == 1) ? ((ram_read) ? 2 : 1) :
  (stateA == 2 && word_count_d == 64) ? 3:
  (stateA == 2 && word_count_d < 64) ? 2:
  //((stateA == 3) ? ((shad_done) ? 4 : 3):
  (stateA == 3) ? ((shad_count == 63) ? 0 : 3):
  //((stateA == 4) ? 0:
  4'bxxxx;

//Data SHA Control
always@(negedge clk)
begin
  if(stateB == 0)
  begin
    //hash_i <= 0;
    sha_tick <= 0;
    sha_load <= 0;
  end
  else if(stateB == 1)
  begin

if(shad_rdy)
begin
  if(shad_tick == 0)
  begin
    shad_load <= 0;
    shad_tick <= 1;
  end
  else if(shad_tick == 1)
  begin
    shad_load <= 1;
    shad_tick <= 2;
  end
  else if(shad_tick == 2)
  begin
    shad_load <= 0;
    shad_tick <= 3;
    //hash_i <= 0;
  end
end

assign nextstateB = (stateB == 0 && data_fifo_count < 64) ? 0 :
                   (stateB == 0 && data_fifo_count >= 64) ? 1 :
                   (stateB == 1 && !shad_done) ? 1 :
                   (stateB == 1 && shad_done) ? 2 :
                   (stateB == 2) ? 0 :
                   4'bxxxx;

//Cache Control Unit
always@(!negedge clk)
begin
  if(stateC==0)
  begin
    //Reset State
    addr_fifo_rd_en <= 0;
    level <= 0;
    r_req <= 0;
    word_count_c <= 0;
    L_good<=0;
    hte_done <= 0;
    //shad_rst_ <= 0;
  end
  else if(stateC==1)
  begin
    if(addr_fifo_count != 0) //If a block address is available
      addr_fifo_rd_en <= 1; //read enable high for one cycle
  end
  else if(stateC==2)
  begin
    addr_fifo_rd_en <= 0;
    r_req <= 1; //Request stored data hash from hash RAM
  end
  else if(stateC==3)
begin //Wait here until word is retrieved from cache
r_req <= 0;
end
else if(stateC==4)
begin //Data Hash from Hash RAM stored in local register
hash_from_cache[word_count_c] <= from_HTE_cache;
end
else if(stateC==5)
begin //0-7 counter incremented until rolls around to zero
word_count_c <= word_count_c + 1;
end
else if(stateC==6)
begin
if(shad_done)
begin
if(hash_match_d == 8'b11111111)
begin
//Check hash_from_cache against hash SHA's output
L_good[level] <= 1;
level<=level+1;
word_count_c <= 0;
shad_rst_ <= 1; //Reset the data SHA so it can hash another block
end
end
end
else if(stateC == 7)
begin
r_req <= 1;
shad_rst_ <= 0;
end
else if(stateC == 8)
begin //Wait here until hash cache retrieves word
r_req <= 0;
end
else if(stateC == 9)
begin
//copy word to array
hash_from_cache[word_count_c] <= from_HTE_cache;
end
else if(stateC == 10)
begin
word_count_c <= word_count_c + 1;
end
else if(stateC == 11)
begin
if(hh_set || level == 3'b011)
begin
if(hash_match_h == 8'b11111111 || hash_match_r == 8'b11111111)
begin
// Set 'good' bit for current level
L_good[level] <= 1;
end
else
begin
// Set 'good' bit for current level
L_good[level] <= 1;
end
end

begin
  //level <= level + 1;
  word_count_c <= 0;
end
//end
end

else if(stateC == 12)
begin
  if(hash_match_h == 8'b11111111)
    //If it was a match, go to next level
    level <= level + 1;
end

else if(stateC == 13)
begin
  shad_rst_ <= 0;
  //check computed hash against SRH
  if(hash_match_r == 8'b11111111)
    begin
      //Mark fifth position (binary 4) as matching
      L_good[3'b100]<=1;
      //Authentication has finished
      hte_done <= 1;
    end
  end
end
else if(stateC == 14)
begin
  hte_done <= 1;
end
end

assign nextstateC = (stateC == 0) ? 1 :
  (stateC == 1) ? ((addr_fifo_count > 0) ? 2 : 1) :
    (stateC == 2) ? 3 :
      //((stateC == 3) && (word_count_c == 0 && cache_hit)) ?
          14 :
            (stateC == 3) ? ((cache_done) ? 4 : 3) :
              (stateC == 4) ? 5 :
                (stateC == 5) ? (word_count_c==0?6:2):
                  //((stateC == 6) ? (shad_done && L_good[0] == 1 ? 7 :
                     6):
                       (stateC == 6) ? (shad_done ? (L_good[0] ? 7 : 5'h1F)
                            : 6):
                             (stateC == 7) ? 8 :
                               (stateC == 8) && (word_count_c == 0 && cache_hit) ?
14 :
                          (stateC == 8) ? ((cache_done) ? 9 : 8) :
                            (stateC == 9) ? 10 :
                              (stateC == 10) ?(word_count_c==0?11:7):
                                (stateC == 11) ? (hh_set ? ( (L_good[level] == 1) ?
12 :
                                      5'h1f) : 11):
                                      ...
// (stateC == 11) ? (shah_done) ? (L_good[level]) ? 12 :

5'h1f) : 11 :
(stateC == 12) ? (L_good == 5'bi11110 ? 13 : 7 ) :
(stateC == 13) ? (hh_set ? (hte_done ? 0 : 31 ) : 13):
(stateC == 14) ? 0 :
(stateC == 5'h1F) ? 5'h1F :
4'bxxxx;

// Hash SHA Queue Control
always@(negedge clk)
begin
    if(stateE == 0)
    begin
        shah_count <= 0;
        // Reset Hash SHA for current run
        shah_rst_ <= 1;
    end
    else if(stateE == 1)
    begin
        shah_rst_ <= 0;
    end
    else if(stateE == 2)
    begin
        if(shah_tick == 1)
        begin
            shah_count <= 1;
        end
        else if(shah_tick == 3)
        begin
            if(shah_count < 64)
                shah_count <= shah_count + 1;
        end
    end
    else if(stateE == 3)
    begin
        // Set h values for current block
        hh_reg[0] <= hh[0];
        hh_reg[1] <= hh[1];
        hh_reg[2] <= hh[2];
        hh_reg[3] <= hh[3];
        hh_reg[4] <= hh[4];
        hh_reg[5] <= hh[5];
        hh_reg[6] <= hh[6];
        hh_reg[7] <= hh[7];
    end
end

assign nextstateE = (stateE == 0) ? 1 :
(stateE == 1) ? ((shah_tick == 1)? 2 : 1):
(stateE == 2) ? ((shah_done)? 3 : 2) :
(stateE == 3) ? 0 :
4'bxxxx;
//Section E
//Hybrid Control - between CUs
always@(posedge clk)
begin
    if(rst)
        hh_set <= 0;
    else if(stateE == 3)
        //set register to indicate if computed hash was latched
        hh_set <= 1;
    else if(stateC == 12)
        hh_set <= 0;
end

//Hash SHA Control
always@(negedge clk)
begin
    if(stateF == 0)
        begin
            hash_fifo_rd_en <= 0;
            shah_load <= 0;
            shah_tick <= 0;
        end
    else if(stateF == 1)
        begin
            if(shah_rdy)
                if(shah_tick==0)
                    begin
                        shah_load <= 0;
                        shah_tick <= 1;
                    end
                else if(shah_tick==1)
                    begin
                        // Build square pulse sha_load to activate hash SHA
                        shah_load <= 1;
                        shah_tick <= 2;
                        hash_fifo_rd_en <= 1;
                    end
        end
    else if(stateF == 2)
        begin
            shah_tick <= 0;
        end
end
end

assign nextstateF = (stateF == 0) ? (hash_fifo_count >= 64 ? 64 : 1 : 0) :
  (stateF == 1) ? (shah_done ? 2 : 1) :
  (stateF == 2) ? (stateE == 0 ? 0 : 2) :
        4'bxxxx;

assign data_fifo_write = /*(state == 2) &&*/ ram_read;
assign hash_fifo_write = /*(stateD == 3) &&*/ hash_read;

genvar j;
genrate
  //generate arrays to compare stored hash values with computed
values 0-7
  for(j=0;j<8;j=j+1)
  begin:hash_match_assign
    assign hash_match_d[j] = (hash_from_cache[j] == hd[j]);
    assign hash_match_h[j] = (hash_from_cache[j] == hh_reg[j] ||
        hash_from_cache[j] == hh[j]);
    assign hash_match_r[j] = (secure_root_hash[j] == hh_reg[j]);
  end
endgenerate

dendmodule
module cache(
    input clk,
    input [31:0] datareq,
    output [31:0] toram,
    input [31:0] fromram,
    input [2[1:0] addrreq,
    input rst,
    //output [15:0]tagled, //dbg
    output [3[0]stateled, //dbg
    output reg found,
    output reg not_found,
    input r_req, w_req,
    output [31:0]latchout, //dbg
    //output reg trip_w, trip_r,
    output [2[0]ram_addr,
    output ramwrite,done,
    output reg ram_read,
    //output reg w_req_ram, r_req_ram
    output reg [31:0]cache_out,
    output reg hit
);
wire [31:0]dataw;
wire [3:0]line_w;
wire match_any;
reg [5:0]match_slot;
wire [31:0]tag_match;
//wire [31:0]tag_match_ar;
//reg half_clk;
//assign data = (rw == 0) ? 32'hzzzzzzzz : dataw;
reg [15:0]tag[0:31];
reg [31:0]tag_present;
reg [31:0]tag_dirty;
reg trip_w, trip_r;
//reg ram_read;
reg [3:0]addrcount;
//reg [10:0]addrlatch;
//reg [5:0]wordlatch;
reg [31:0]datalatch;
reg [2[0]addrreg;
reg [15:0]evict_addr_tag;
reg [3[0]state;
wire [3:0]nextstate;
reg [4:0]i;
reg [4:0]line;
reg [5:0]ram_addr_lsb;
reg [9:0]cache_addr;
reg [4:0] lru_head;
reg [4:0] lru_tail;
reg cache_ld_done;
reg ram_write_ready;
reg ram_write_done;

reg ram_read_ready;
reg [2:0] delay_ct;

//assign ramread = ram_read_ready && ram_addr_lsb != 6'h3F;//(state==7);
wire [31:0] cache_input;
//reg [3:0] slot; //AKA line

dist_mem_cache

cache0(.clk(clk),.a(cache_addr),.d(cache_input),.spo(dataw),.we(rw));

assign cache_input = (state==8) ? datalatch : fromram;
assign stateled = state; //dbg
assign latchout = datalatch; //dbg
assign ram_addr = (state != 5) ? {addrreg[21:6],ram_addr_lsb} :
{evict_addr_tag,ram_addr_lsb};
assign toram = dataw;
assign ramwrite = ram_write_ready;

always@(posedge clk)
begin
  if(rst)
    state<=0;
  else
    state <= nextstate;
end

always@(posedge clk)
begin
  if(state==3 && trip_r)
    cache_out <= dataw;
end

always@(negedge clk)
begin
  if(state==0)
    begin
      tag_present<={32'h00000000};
      tag_dirty <= {32'h00000000};
      i<=0;
      lru_head<=0;
      lru_tail<=0;
      //addrlatch<=0;
      //wordlatch<=0;
      addrreg <= 0;
      trip_r<=0;
      trip_w<=0;
      found<=0;
      not_found<=0;
      ram_read_ready<=0;
    end
end
ram_read <= 0;
delay_ct <= 0;
end
else if(state==1)
begin
hit <= 0;
ram_write_ready <= 0;
ram_read_ready <= 0;
ram_read <= 0;
if(trip_r|trip_w)
begin
trip_r<=0;
trip_w<=0;
end
else
begin
addreg <= addrreq;//How's that for ambiguity?
if(r_req)
begin
trip_r<=1;
end
else if(w_req)
begin
// addreg <= addrreq;
datalatch <= datareq;
trip_w<=1;
end
end
end
else if(state==2)
begin
if(match_any)
begin
line=match_slot[4:0];
found<=1;
hit <= 1;
end
else
//begin
// if(i<31)
// i<=i+1;
// else
// not_found<=1;
//end
end
else if(state==3)
begin // line set in state 2 or 6
cache_ld_done <= 0;
cache_addr <= {line,addrreg[5:0]};
found <=0 ;
end
else if(state==4)//Item not found
begin
//rm_latched<=0;
not_found <= 0;
end
else if(state==5)//No free slot
begin
if(ram_write_ready == 0)
begin
    cache_addr<={lru_tail,6'b000000};
    //ram_addr<={tag[lru_tail],6'b000000}; //not a reg
tag_present[lru_tail]<=0;
    ram_write_ready<=1;
    ram_addr_lsb<=0;
evict_addr_tag<=tag[lru_tail];
end
else //ram_write_ready == 1 writing is in progress
begin
    if(tag_dirty[lru_tail]==1)
    begin// Write evicted block back to RAM if modified
        if(ram_addr_lsb<63)
            begin
                ram_addr_lsb <= ram_addr_lsb+1;
                cache_addr <= cache_addr+1;
//writeback to ram
            end
        else
            begin
                ram_write_ready<=0;
                ram_write_done<=1;
                tag_dirty[lru_tail]<=0;
lru_tail<=lru_tail+1;
//lru_head<=lru_head+1;
            end
    end
else
begin
    ram_write_ready<=0;
    ram_write_done<=1;
    tag_dirty[lru_tail]<=0;
lru_tail<=lru_tail+1;
//lru_head<=lru_head+1;
end
end
end
else if(state==6)//A slot is available
begin
line<=lru_head;
lru_head<=lru_head+1;
i<=0;
ram_write_done <= 0;
cache_addr<={lru_head,6'b000000};
ram_addr_lsb <= 0;
//
ram_addr<=addrlatch;
cache_ld_done <= 0;
end
else if(state==7)//Read page from RAM
begin
  if(ram_read_ready == 0)
  begin
    if(delay_ct == 3'b111)
      begin
        ram_read_ready <= 1;
        ram_read <= 1;
        delay_ct <= 3'b000;
      end
    else
      begin
        delay_ct <= delay_ct + 1;
      end
    end
  end
else if(ram_addr_lsb<63)
begin
  //Increment word counter address of cache
  ram_addr_lsb<=ram_addr_lsb+1;
  cache_addr<=cache_addr+1;
end
else
begin
  //Register address of block and set present bit
  tag_present[line]<=1'b1;
  tag[line]<=addrreg[21:6];
  cache_ld_done<=1;
  i<=0;
  ram_read <= 0;
end
else if(state==8)
begin
  if(trip_w)
    begin
      //Set dirty bit if cache operation was a write
      tag_dirty[line]<=1;
    end
    //cache
    //trip_r<=0;
    //trip_w<=0;
end
else if(state==15) // Default error state
begin
  if(cache_addr>0)
    cache_addr<=cache_addr-4'b0001;
end
end

assign nextstate = (state == 0) ? 1 :
  (state == 1) ? ( (trip_r | trip_w) ? 2 : 1) :
  (state == 2 && found) ? 3 :
  (state == 2 && not_found) ? 4 :
  (state == 2) ? 2 :
  (state == 3) ? 8 :
  (state == 4 && !(lru_head+1==lru_tail)) ? 6 :
(state == 4 && (lru_head+1==lru_tail)) ? 5 : */
(state == 4 && !(tag_present==32'hffffffff)) ? 6 :
(state == 4 && (tag_present==32'hffffffff)) ? 5 :
(state == 5 && ram_write_done==0) ? 5 :
(state == 5 && ram_write_done==1) ? 6 :
(state == 6) ? 7 :
(state == 7) && (cache_ld_done==1) ? 3 :
(state == 7) && (cache_ld_done!=1) ? 7 :
(state == 8) ? 1 :
15;

assign tagled = tag_present;
assign rw = (state==8 && trip_w) || (state == 7 && ram_read_ready);
//assign ramwrite = (state == 5);
assign done = (state == 8);

genvar j;
  generate
    for(j=0;j<32; j=j+1)
      begin:tagmatchassign
        assign tag_match[j] = (tag_present[j] &&
tag[j]==addrreg[21:6]);
      end
  endgenerate

assign match_any = ( tag_match != 0);

always@(tag_match/*_ar*/)
begin
  case(tag_match/*_ar*/)
    32'h00000001:match_slot = 0;
    32'h00000002:match_slot = 1;
    32'h00000004:match_slot = 2;
    32'h00000008:match_slot = 3;
    32'h00000010:match_slot = 4;
    32'h00000020:match_slot = 5;
    32'h00000040:match_slot = 6;
    32'h00000080:match_slot = 7;
    32'h00000100:match_slot = 8;
    32'h00000200:match_slot = 9;
    32'h00000400:match_slot = 10;
    32'h00000800:match_slot = 11;
    32'h00001000:match_slot = 12;
    32'h00002000:match_slot = 13;
    32'h00004000:match_slot = 14;
    32'h00008000:match_slot = 15;
    32'h00010000:match_slot = 16;
    32'h00020000:match_slot = 17;
    32'h00040000:match_slot = 18;
    32'h00080000:match_slot = 19;
```verilog
endcase
end

module sha256(
    input clk, rst,
    input [31:0] data,
    input ld1, ld4,
    output reg [31:0] h0,
    output reg [31:0] h1,
    output reg [31:0] h2,
    output reg [31:0] h3,
    output reg [31:0] h4,
    output reg [31:0] h5,
)

// SHA 256 Hashing Module

32'h00100000:match_slot = 20;
32'h00200000:match_slot = 21;
32'h00400000:match_slot = 22;
32'h00800000:match_slot = 23;
32'h01000000:match_slot = 24;
32'h02000000:match_slot = 25;
32'h04000000:match_slot = 26;
32'h08000000:match_slot = 27;
32'h10000000:match_slot = 28;
32'h20000000:match_slot = 29;
32'h40000000:match_slot = 30;
32'h80000000:match_slot = 31;
endmodule
```

`timescale 1ns / 1ns

module sha256(
    input clk, rst,
    input [31:0] data,
    input ld1, ld4,
    output reg [31:0] h0,
    output reg [31:0] h1,
    output reg [31:0] h2,
    output reg [31:0] h3,
    output reg [31:0] h4,
    output reg [31:0] h5,
)

// SHA 256 Hashing Module

32'h00100000:match_slot = 20;
32'h00200000:match_slot = 21;
32'h00400000:match_slot = 22;
32'h00800000:match_slot = 23;
32'h01000000:match_slot = 24;
32'h02000000:match_slot = 25;
32'h04000000:match_slot = 26;
32'h08000000:match_slot = 27;
32'h10000000:match_slot = 28;
32'h20000000:match_slot = 29;
32'h40000000:match_slot = 30;
32'h80000000:match_slot = 31;
endcase
end

endmodule

sha.v – The SHA 256 Hashing Module
output reg [31:0] h6,  
output reg [31:0] h7,  
output reg done,  
output reg sha_rdy  
);

reg [3:0] state;  
reg [5:0] count;  
reg [2:0] runs; // Indicates times to run; 1 or 4  
reg [2:0] n;  // Run currently being processed  
reg [5:0] addr2k;  
reg [6:0] addrMain;

wire [31:0] out2k;  //Drives WE signals of 2kb ram  
wire load2k;  //Drives WE signals of the two dual rams  
wire loadMain;  //Drives WE signals of the two dual rams  
wire [6:0] addra;  
wire [6:0] addrb;  
wire [6:0] addrc;  
wire [6:0] addrd;  
wire [31:0] dataa;  
wire [31:0] datab;  
wire [31:0] datac;  
wire [31:0] datad;  
wire [31:0] mainOuta;  
wire [31:0] mainOutb;  
wire [31:0] mainOutc;  
wire [31:0] mainOutd;  
reg [6:0] i;  
reg [31:0] latch;  
reg [5:0] k_addr;  
wire [31:0] k;  

// wire [31:0] swapA;

reg [31:0] a;  
reg [31:0] b;  
reg [31:0] c;  
reg [31:0] d;  
reg [31:0] e;  
reg [31:0] f;  
reg [31:0] g;  
reg [31:0] h;  
wire [31:0] s0;  
wire [31:0] s1;  
wire [31:0] s2;  
wire [31:0] s3;  
wire [31:0] chuh;  
wire [31:0] madj;  
wire [31:0] temp1;  
//reg trip;

//Xilinx Core IP Modules
dist_mem_2k
ram2k(.clk(!clk),.we(load2k),.d(data),.a(addr2k),.spo(out2k));
blk_mem_dual
main1(.clka(!clk),.addra(addra),.dina(dataa),.douta(mainOuta),
    .wea(loadMain),.clkb(!clk),.addrb(addrb),.dinb(datab),
    .doutb(mainOutb),.web(1'b0));
blk_mem_dual
main2(.clka(!clk),.addrc(addrc),.dina(datac),.douta(mainOutc),
    .wea(loadMain),.clkb(!clk),.addrd(addrd),.dinb(datad),.doutb(mainOutd),
    .web(1'b0));
//Verilog coded lookup table of k values used by SHA 256
k_lookup k_lookup1(.addr(k_addr),.data(k));

assign addra = (state == 3) || (state == 5) || (state == 6) ? addrMain : addrMain-15;
assign addrb = (state == 3) || (state == 5) || (state == 6) ? addrMain : addrMain-2;
assign addrc = (state == 3) || (state == 5) || (state == 6) ? addrMain : addrMain-16;
assign addrd = (state == 3) || (state == 5) || (state == 6) ? addrMain : addrMain-7;

assign dataa = (state == 3) ? out2k : (state == 4) ? 32'h00000000 : latch;
assign datab = (state == 3) ? out2k : (state == 4) ? 32'h00000000 : latch;
assign datac = (state == 3) ? out2k : (state == 4) ? 32'h00000000 : latch;
assign datad = (state == 3) ? out2k : (state == 4) ? 32'h00000000 : latch;

//Filling array algorithm
assign s0 =
    {mainOuta[6:0],mainOuta[31:7]}^{mainOuta[17:0],mainOuta[31:18]}^  
    {3'b000,mainOuta[31:3]};
assign s1 =
    {mainOutb[16:0],mainOutb[31:17]}^{mainOutb[18:0],mainOutb[31:19]}^  
    {10'b0000000000,mainOutb[31:10]};
assign s2 = {a[1:0],a[31:2]}^{a[12:0],a[31:13]}^{a[21:0],a[31:22]};
assign s3 = {e[5:0],e[31:6]}^{e[10:0],e[31:11]}^{e[24:0],e[31:25]};
assign chuh = (e & f) ^ ((~e) & g);
assign madj = (a & b) ^ (a & c) ^ (b & c);
assign swapA = {mainOuta[7:0],mainOuta[15:8],mainOuta[23:16],  
    mainOuta[31:24]};
assign temp1 = h + s3 + chuh + k + mainOuta;

always@ (posedge clk)
begin
    if(rst)
    begin
        state<=0;
        //addr2k<=0;
begin
    state<=
begin
        state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
        //addr2k<=0;
begin
    state<=0;
end
else if(state==0)
begin
n<=0;
done <= 0;
sha_rdy <= 1;
addr2k<=0;
latch<=32'h55555555;
h0<=32'h6A09E667;
h1<=32'hBB67AE85;
h2<=32'h3C6EF372;
h3<=32'hA54FF53A;
h4<=32'h510E527F;
h5<=32'h9B05688C;
h6<=32'h1F83D9AB;
h7<=32'h5BE0CD19;
if(ld4)
begin
  count<=6'b111111;
  runs<=3'b100;
  state<=1;
  sha_rdy<=0;
end
else if(ld1)
begin
  count<=6'b001111;
  runs<=3'b001;
  state<=1;
  sha_rdy<=0;
end
end
else if(state==1)
begin
if(addr2k==count)
begin
  addr2k<=0;
  state<=2;
end
else
  addr2k<=addr2k+1;
end
else if(state==2)
begin
if(n<runs)
begin
  addrMain<=0;
  //addr2k<=0;
  //trip<=0;
a<=h0;
b<=h1;
c<=h2;
d<=h3;
e<=h4;
f<=h5;
g<=h6;
h<=h7;
state<=3;
end
else
begin
done <= 1;
end
end
else if(state==3)
begin
if(addrMain <= 15)
begin
addrMain<=addrMain+1;
addr2k<=addr2k+1;
end
else
begin
state<=4;
//addrMain<=16;
end
end
else if(state==4)
begin
if(addrMain<=63)
begin
if(trip==0)
trip<=1;
else
begin
//addrMain<=addrMain+1;
latch<=mainOutc+s0+mainOutd+s1; //w[i] = w[i-16]+s0+w[i-7]+s2;
state<=5;
//end
end
else
begin
addrMain<=0;
k_addr<=0;
state<=6;
end
end
else if(state==5)
begin
addrMain<=addrMain+1;
state<=4;
end
else if(state == 6)
begin
if(addrMain<=63)
begin
h <= g;
g <= f;
f <= e;
e <= d + temp1;
d <= c;
c <= b;
b <= a;
a <= temp1 + s2 + madj;
addrMain<=addrMain + 1;
    k_addr<=k_addr + 1;
    end
else
    state<=7;
end
else if(state == 7)
begin
    h0 <= h0 + a;
h1 <= h1 + b;
h2 <= h2 + c;
h3 <= h3 + d;
h4 <= h4 + e;
h5 <= h5 + f;
h6 <= h6 + g;
h7 <= h7 + h;
n <= n + 1;
state <= 2;
end
end
assign load2k = (state==1);
assign loadMain = (state==3)|(state==5);
endmodule

tb3.v – Test Bench File Used to Validate HTE

`timescale 1ns / 1ps

// Verilog Test Fixture created by ISE for module: hte
// Dependencies:
// Revision:
module tb3;

// Inputs
reg clk;
reg rst;
reg [17:0] addr_req;
// reg capture;
reg [31:0] data;
reg ram_read;
reg [31:0] hash_data;
// Outputs
wire hash_read;
//wire [3:0]cache_state;

// Instantiate the Unit Under Test (UUT)
hte uut (  
    .clk(clk),
    .rst(rst),
    .addr_req(addr_req),
    // .capture(capture),
    .fromRAM(data),
    .ram_read(ram_read),
    .from_hash_ram(hash_data),
    .hash_read(hash_read)
);

initial begin
    // Initialize Inputs
    clk = 0;
    rst = 1;
    // addr_req = 18'h24500; //byte address is meaningless
    addr_req = 18'h9140;  //word address equivalent
    // capture = 0;
    data = 32'hFFFFFFFF;
    hash_data = 32'habababab;
    ram_read = 0;

    // Wait 100 ns for global reset to finish
    #100;
    rst = 0;
    // Data block retrieved
    #60 ram_read = 1;
    #0 data=32'h4E657665;0
    //#2 1d4 = 0;
    #4 data=32'h7220696D;
    #4 data=32'h6167696E;
    #4 data=32'h75727365;4
#4 data=32'h67696e65;
#4 data=32'h20796f75;
#4 data=32'h7273656c;

#4 data=32'h66206e6f;//56
#4 data=32'h7420746f;
#4 data=32'h20626520;
#4 data=32'h6f746865;
#4 data=32'h72776973;//60
#4 data=32'h65207480;
#4 data=32'h00000000;
#4 data=32'h000007b8;
#4 ram_read = 0;
data = 32'hcccccccc;
//Second block of RAM
#3000 ram_read = 1;
addr_req = 18'h9180;
data = 32'h0;
#4 data = 32'h30001;// Modified zeros block
#4 data = 32'h0; // Modified zeros block
//#256 ram_read = 0; // Unmodified zeros block
#248 ram_read = 0; // Modified zeros block

end

initial begin
//wait(cache_state == 7)
wait(hash_read)
//No ram_read needed for hashes
//Level 1 node load
#0 hash_data=32'hb267799a;//1
#4 hash_data=32'hcdcdfd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hb267799a;//2
#4 hash_data=32'hcdcdfd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hb267799a;//3
#4 hash_data=32'hcdcdfd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hhb267799a;//4
#4 hash_data=32'hcdcdfffd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'hhb4d32bd3;
#4 hash_data=32'h3fe931ba;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hhb267799a;//5
#4 hash_data=32'hcdcdfffd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'hhb4d32bd3;
#4 hash_data=32'h3fe931ba;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hfacd5137;//6
#4 hash_data=32'h2e361d67;
#4 hash_data=32'h065c45a6;
#4 hash_data=32'h4713ee9;
#4 hash_data=32'hf3a998a1;
#4 hash_data=32'ha9d5f1aa;
#4 hash_data=32'h7648530c;
#4 hash_data=32'h3401fab3;

#4 hash_data=32'hhb267799a;//7
#4 hash_data=32'hcdcdfffd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'hhb4d32bd3;
#4 hash_data=32'h3fe931ba;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;

#4 hash_data=32'hhb267799a;//8
#4 hash_data=32'hcdcdfffd12;
#4 hash_data=32'h8d459248;
#4 hash_data=32'hhb4d32bd3;
#4 hash_data=32'h3fe931ba;
#4 hash_data=32'h4ede20d7;
#4 hash_data=32'h674520ee;
#4 hash_data=32'hbc38a21c;
#4 hash_data=32'h00000000;
#4 hash_data=32'hCCCCCCCC;
#40
wait(hash_read) //Level 2 Node
#0 hash_data = 32'h99bcb8b6;//1
#4 hash_data = 32'hc8323ebb;
#4 hash_data = 32'h756dc940;
#4 hash_data = 32'h08f891e5;
#4 hash_data = 32'h78f099ed;
#4 hash_data = 32'he2dd1476;
#4 hash_data = 32'hc8919682;
#4 hash_data = 32'hf73141fd;

#4 hash_data = 32'h98c4a6b2;//2
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//3
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//4
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//5
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//6
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//7
#4 hash_data = 32'h9bcb16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h98c4a6b2;//8
#4 hash_data = 32'h9bce16c6;
#4 hash_data = 32'hf38b71a7;
#4 hash_data = 32'h0f72cb42;
#4 hash_data = 32'hf52a8b5f;
#4 hash_data = 32'h34773482;
#4 hash_data = 32'h3dc503e1;
#4 hash_data = 32'h0c420adb;

#4 hash_data = 32'h00000000;
#40

//End of L2 Block load

//L3 Block Load

wait(hash_read)
#0 hash_data = 32'hdb4e7de4;//1
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'hd4ce3e79;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hcca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'h2e92586f;//2
#4 hash_data = 32'h8777a8dc;
#4 hash_data = 32'hf3884eb;
#4 hash_data = 32'h893c89c;
#4 hash_data = 32'h026243ff;
#4 hash_data = 32'h0e80b05;
#4 hash_data = 32'hacfc1e0eb;
#4 hash_data = 32'h23388b30;

#4 hash_data = 32'hdb4e7de4;//3
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h4ce3e79;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hcca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'hdb4e7de4;//4
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h4ce3e79;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hcca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'hdb4e7de4;//5
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hccca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'hdb4e7de4;//6
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hccca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'hdb4e7de4;//7
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hccca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'hdb4e7de4;//8
#4 hash_data = 32'h9331b66f;
#4 hash_data = 32'hca70e45c;
#4 hash_data = 32'h120c49e8;
#4 hash_data = 32'h39dea9d1;
#4 hash_data = 32'hccca6bffd;
#4 hash_data = 32'h924fd4e5;

#4 hash_data = 32'h00000000;
#40

//L4 Node

wait(hash_read)
#0 hash_data = 32'hdb92c02e;//1
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'h1cda801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hb7ba647e;//2
#4 hash_data = 32'h2fcf0721;
#4 hash_data = 32'h9a0ee0f4;
#4 hash_data = 32'h62699859;
#4 hash_data = 32'h0e2f0ad0;
#4 hash_data = 32'h1b5c8aff;
#4 hash_data = 32'h21486b72;
#4 hash_data = 32'hceb29c58b;
#4 hash_data = 32'hdb92c02e;//3
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hdb92c02e;//4
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hdb92c02e;//5
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hdb92c02e;//6
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hdb92c02e;//7
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'hdb92c02e;//8
#4 hash_data = 32'hba1097b0;
#4 hash_data = 32'h709ef989;
#4 hash_data = 32'hdb5eac3d;
#4 hash_data = 32'hd1d801ce;
#4 hash_data = 32'h338ae39a;
#4 hash_data = 32'h168dea3a;
#4 hash_data = 32'h117f31fc;
#4 hash_data = 32'h00000000;
end

always #2 clk = !clk;
endmodule
References


