Algorithm to Architecture Mapping of Compression and Hashing techniques

ES6126: Algorithms to Architectures Coursework Project

Deepshikha
G1601338G, M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
deep0023@e.ntu.edu.sg

Swarna Kamakshi Jayaraman
G1601351B, M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
s160015@e.ntu.edu.sg

Agatheswaran Rajashree Sundaram
G1601327A, M.Sc., Embedded Systems
School of Computer Science and Engineering
Nanyang Technological University, Singapore
rajashre01@e.ntu.edu.sg

***Abstract*— Heterogeneous architectures promise an improved system design leveraging the benefits of high-speed programmable computing elements that seamlessly work together and offer unparalleled scalability, performance and power economics [1]. Unlike software programming, hardware programming involves definitive bit-widths for variables, timing constraints and clear distinction between procedural and concurrent blocks. This paper focusses on translating the algorithmic description into gate level netlist via Verilog Hardware Description Language and mapping the same to the given Spartan 6 FPGA architecture. The correctness of the design shall be verified and the overall utilization and latency studied, to decide on which performance metrics can be improved.**

Keywords—Huffman; compression; hash; digest; MD5; Synthesis; HDL; Verilog;

I. INTRODUCTION

Various encryption algorithms have been optimized and benchmarked in software. However, quest for better performance in terms of throughput, i.e. number of outputs produced per unit time or reduced latency / execution cycles never ceases to exist. Although there may be many possible algorithms to accomplish an application, the challenge is to identify the right one for the right architecture. In industries, it is a common practice to breakdown the requirements and identify the right architecture to implement it on to achieve the required performance goals. After selection of architecture, the algorithm that best suits the architecture is selected and implemented. Although the high-level languages are well-optimized for most applications, they are on par with the performance that can be accomplished with hardware. This paper deals with implementation of two popular algorithms, Huffman encoding for data compression and MD5 hashing for data security and integrity checks. The algorithms are written in Verilog HDL, synthesized to get gate-level netlist and implemented in hardware.

II. IMPLEMENTATION

A. *Huffman Encoder and Decoder*

i. Background

Huffman Encoder is a lossless data compression algorithm, which separates unique symbols from a string, and encodes the symbols in variable length codes, based on their frequency of occurrence [4]. The result is a symbol table which has a different code for each of the different characters. The advantage of Huffman over fixed length codes is that we can save a large number of bits, since the minimum number of bits in a code is 1.

ii. Algorithm

Input String: “woowworldwoowwllloo”

The Distinct symbols, in sorted order of frequency are:

Symbol	Frequency
o	7
w	6
L	4
R	1
d	1

[a.] TABLE 1. Symbol vs. Frequency Table

Huffman algorithm starts with adding the least two frequencies and generating the sum. The resulting symbol corresponding to this addition is marked as '*'. The frequencies are updated. And the same steps are repeated till a single value remains. A Tree is formed, wherein the symbols are in the leaf nodes and '*' is the parent node for the 2 symbols which are combined. From the last value, we start to back-trace. For every sum performed, a '0' is appended to the code of the symbol with least frequency and a '1' is appended in to code of the symbol with second least frequency.

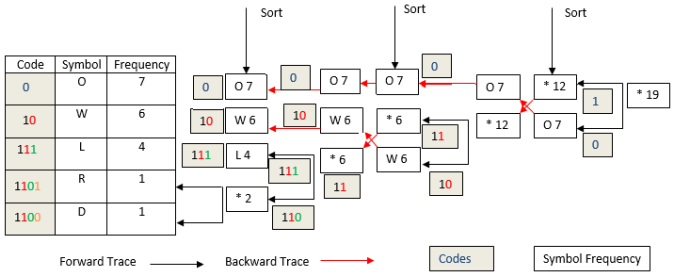

Fig. 1. Huffman Encoding and Decoding Logic

iii. Implementation

The symbols are sorted based on their frequency of occurrence. The least two frequencies are added and the sum is updated in the frequency table. The symbol corresponding to this summed frequency value is stored as '*'. A Huffman tree is built and stored in a linear array form wherein the new symbol '*' is stored at index (i), the symbol corresponding to the least frequency is stored at index (i+1), signifying the left child and the symbol corresponding to the second least frequency is stored at index (i+2), signifying the right child and the symbol table is updated. The above step is performed until all the entries in the symbol table are processed.

*	R	D	*	*	L	*	W	*	*	O	*

Fig. 2. Array equivalent of constructed Huffman Tree for the above example

Encoding

Start processing the tree from reverse direction. If the left or right child is not a valid symbol, update a temporary code, with '0' or '1' corresponding to left or right child. If the left child or right child is a valid symbol, copy the temporary code to the symbol code and append a '0' or a '1' at the end, based on whether it is a left or right child. Update the code length accordingly for the symbol.

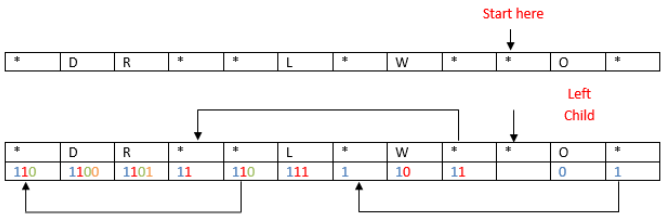

Fig. 3. Encoding from the tree (stored in array)

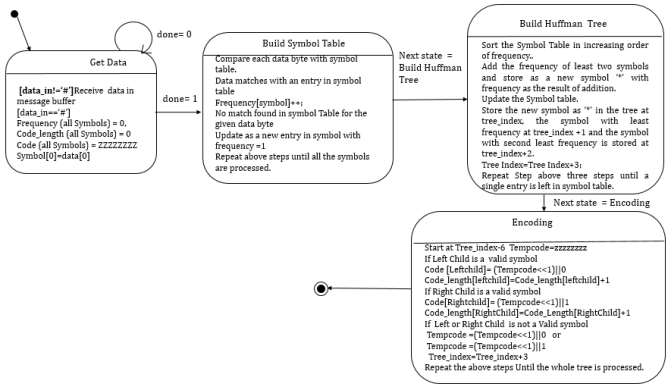

Fig. 4. FSM illustrating Encoder Logic

Decoder

The tree and code are passed as arguments and are stored in buffer and is processed in the reverse manner starting from the last parent.

The length of code is computed as n and the tree is back traversed till the nth parent is reached i.e. with each bit encountered the tree index is reduced by three.If the last bit of code is '0', the symbol is the left child of the tree which is the value at current tree_index+1, or else if the last bit of code is '1',the symbol is the right child of the tree, which is the value at current tree_index+2.

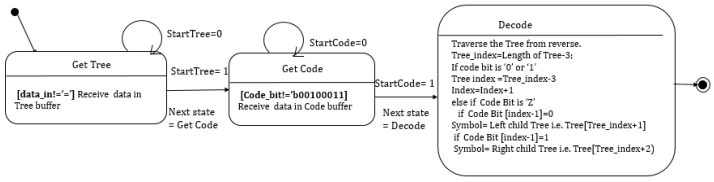

Fig. 5. FSM illustrating Decoder Logic

Example: For a code 111, the length is 3, the tree traversal is shown in Figure [6].

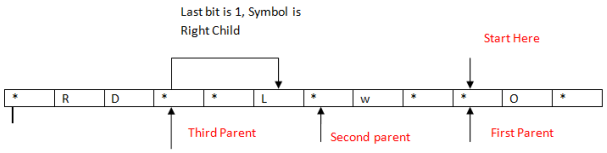

Fig. 6. Illustration of tree traversal for Decoding

B. *MD5 Hashing*

i. Background

A **cryptographic hash function** [2] is defined as a mathematical algorithm which takes in an arbitrary block of data as input and returns a fixed size hash value.

One way to retrieve the input from a hashed output is by Brute Force Attacks, wherein all possible candidates for inputs are tried out exhaustively to see if they produce a match. Another method to crack password hashes by means of Rainbow tables with pre-computed hash values, used as lookups.

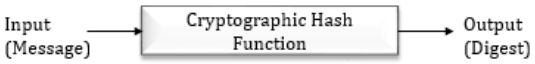

Fig. 7. Cryptographic Hash Function

A cryptographic hash function should satisfy the following properties:

- It should be **deterministic**. The same message should generate the same hash result, in all rounds.
- It should be quick in computing the hash value.
- It should be **irreversible** (one-way).
- A slight change in message should extensively modify the hash value, so that the old hash and the new one are completely uncorrelated.
- Hash value should be **unique** for a message. No two messages shall have the same hash output.

They are widely used as checksums to identify corruption of data or duplicate files, fingerprinting applications and in message authentication codes.

Message Digest 5 (MD5) is a popular cryptographic hashing function developed by the US Cryptographer Ronald Rivest in 1991. The algorithm was intended to take an arbitrary block of data and return a fixed size hash value of 128 bits.

MD5 is popularly used for verifying the integrity of downloaded files, by comparing the pre-computed MD5 checksum from file server to that of the downloaded file. Another interesting application of MD5 is storage of passwords. As it is not recommended to store passwords as plain texts, they are converted to hash values. Whenever a user inputs the password, the equivalent hash value is computed and checked against the stored hash value.

ii. Algorithm

A variable length message of say, b-bits is taken as input, where b is a non-negative integer and $b \geq 0$. The bits are ordered as illustrated in Figure [9].

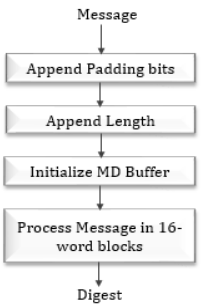

Fig. 8. Steps involved in MD5 Hashing

Step 1: Append Padding Bits

Message padding is done to ensure that total length after padding is congruent to 448 modulo 512, which is equal to 448. Intuitively, this can be observed as the length being 64 bits short of being a multiple of 512. Padding procedure involves appending a bit 1 to the message, followed by as many 0s as required for the length to be a multiple of 448.

Step 2: Append Message Length

The output of the previous step is appended with another 64 bits, representing the length of the message 'b' (before padding bits were added). We know that message lengths up to 2^64 can be represented by the lower 64 bits. Should there be a case where message length is greater than 64, only the lower 64 bits of the length value are used.
Each 512-bit block of the resultant message can be interpreted as 16 32-bit words.

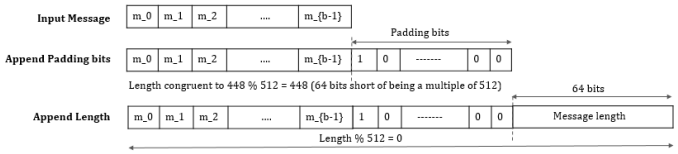

Fig. 9. Ordering of bits in Steps 1 and 2

Step 3: Initialize MD Buffer

MD5 hash is computed using a 4-word buffer (h0, h1, h2, h3). These words are **salts**, which in cryptography refer to random data additionally inputted to the one-way hashing functions to guard against dictionary attacks and rainbow table attacks. h0, h1, h2 and h3 are 32-bit long registers and initialized with the following values during the start of the algorithm in hexadecimal radix:

```
word h0: 01 23 45 67
word h1: 89 ab cd ef
word h2: fe dc ba 98
word h3: 76 54 32 10
```

Step 4: Process message in 16 32-bit (word) blocks

Four auxiliary functions are defined, each taking 3 32-bit words as input and outputting one 32-bit value:

```
F(X, Y ,Z ) = (X & Y) | (~X & Y)
G(X, Y ,Z ) = (X & Z) | (Y & ~Z)
H(X, Y ,Z ) =  X ^ Y ^ Z
I(X, Y ,Z ) =  Y ^ (X | ~Z)
```

Function F acts as a conditional function on input X. i.e. If X is true, then output Y, else output Z. Functions G, H and I also act in a bitwise parallel fashion as F, as defined by the above equations. The key to generating uncorrelated hashes is that as long as inputs X, Y and Z are independent and unbiased, the outputs of functions F, G, H and I are also independent and unbiased [3].

The next step is to construct a table of 64 elements given by the expression,

```
T[i] = floor(2^32 × abs(sin(i + 1))); i=0 to 63
```

For each 16-word block ‘i’ (i.e. 16 x 32 bits = 512-bit block),

- Set X[j] = M [i ×16 + j]; where j runs from 0 to 15 (representing each word in the 512-bit block. (Note: 1 word = 32 bits).

  Thus, each index of X holds one 32-bit block of the 512-bit message.

- Copy salt values in 32-bit registers, A, B, C and D as follows:

```
A = h0
B = h1
C = h2
D = h3
```

- MD5 Hashing involves 4 rounds, with each round having 16 different T[i] based on 16 phases. The Figure [10] depicts the actual sequence in which the computations are to be done for each round and phase. This sequence has been taken from Rivest’s original description of MD5 Algorithm in April 1991.

```
/* Round 1. */
/* Let [abcd k s i] denote the operation
     a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
[ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
[ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
[ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]

/* Round 2. */
/* Let [abcd k s i] denote the operation
     a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
[ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
[ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
[ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]

/* Round 3. */
/* Let [abcd k s t] denote the operation
     a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
[ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
[ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
[ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]

/* Round 4. */
/* Let [abcd k s t] denote the operation
     a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
[ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
[ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
[ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
```

Fig. 10. Computations involved in each Round of MD5 (Source: [3])

In Figure [10], “<<< s” denotes circular shift of the 32-bit value by ‘s’ positions. For each round and phase, result goes through a predefined number of circular shifts ‘s’, specified by the lookup table in Figure [11].

```
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }
```

Fig. 11. Number of circular shifts for each round and phase (Source: [2])

The result of each i-th message block of 512-bit length [ABCD] is then added to the previous salt value to get the output of that iteration. i.e. Each step has a unique additive constant.

```
A = A + h0
B = B + h1
C = C + h2
D = D + h3
```

The output of MD5 is given by A, B, C and D after all iterations of i, ‘A’ being the least significant byte and ‘D’ being the most significant byte.

iii. Realization of MD5 in Verilog HDL

As the MD5 input is to be received through UART Interface which sends 8 bits at a time, the input port of the MD5 module is configured to be 8-bit long. We know that MD5 digest is 128 bits long. Hence, the output port is configured to be 128 bits long. The module starts reading input data when start pin is high. When the final digest is computed, the valid pin is set to 1, to signal that the output is ready to read.

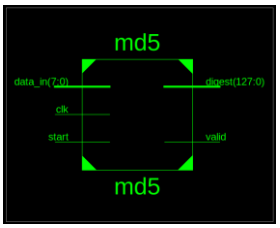

Fig. 12. RTL Schematic of MD5 Module

As Verilog HDL specifies the functionality in the Register transfer level abstraction, it is necessary to decide on the bit-widths of the registers beforehand, based on the application. As MD5 involves 4 rounds and 16 phases for each round, 2-bit ‘round’ register and 4-bit ‘phase’ register are defined.

The message buffer is an array of 64 8-bit values used to store incoming data. To interpret the message as 16 blocks of 32 bits each, a ‘group’ variable is defined as an array of 16 32-bit values. As a state-machine based design has been implemented, two registers are defined to track the present and next state of the state machine.

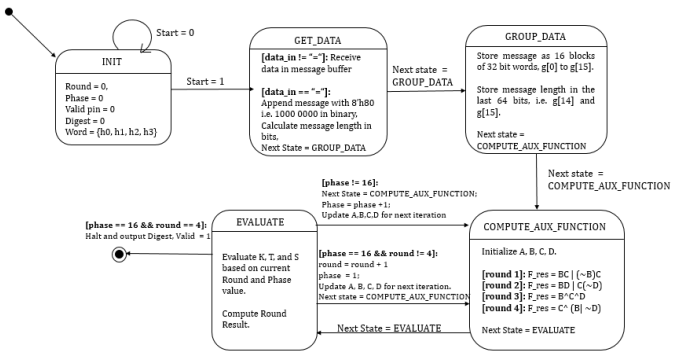

Fig. 13. State Machine Representation of Implemented Verilog Design

The Figure [13] illustrates the sequence of operations performed to compute the final digest.

III. RESULTS

A. *Huffman Encoder and Decoder Results*

- *Comparison of C and RTL Utilization (Encoder)*

Encoder		
Slice Logic Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of Slice Registers	12653	1378
Number of Slice LUTs	19490	1642
Number used as Logic	19372	1587
Number used as Memory	0	3

Slice Logic Distribution	Verilog Implementation	Vivado HLS (implementation)
Number of LUT Flip Flop pairs used	20712	1896
Number with an unused Flip Flop	8473	660
Number with an unused LUT	1222	254
Number of fully used LUT-FF pairs	11017	982
Number of unique control sets	667	57

IO Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of IOs	20	183
Number of bonded IOBs	20	183

Specific Feature Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of BUFG/BUFGCTRLs	1	1

Timing Summary for Huffman Encoder	Verilog Implementation	Vivado HLS Implementation
Speed Grade:	-3	-3
Minimum period:	25.476 ns	3.914ns
Maximum Frequency:	39.252MHz	255.522MHz
Minimum input arrival time before clock	8.912ns	4.733ns
Maximum output required time after clock	4.453ns	6.817ns
Maximum combinational path delay	No path found	5.260 ns

[b.] TABLE 2. Utiization Estimates for Encoder in C and RTL

Throughput Calculation

Encoder:

Number of cycles (HLS) = 1922372 cycles

Throughput = Number of bytes x Max Frequency / No. of cycles

= (64 x 255.522MHz) / 1922372 cycles = 8506 bytes/s

Decoder:

Number of cycles (HLS) = 391 cycles

Throughput = Number of bytes x Max Frequency / No. of cycles

= (64 x 255.522MHz) / 391 cycles = 41.93 Mbytes/s

- *Comparison of C and RTL Utilization (Decoder)*

Decoder		
Slice Logic Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of Slice Registers	0	69
Number of Slice LUTs	0	93
Number used as Logic	0	93
Number used as Memory	0	0

Slice Logic Distribution	Verilog Implementation	Vivado HLS (implementation)
Number of LUT Flip Flop pairs used	0	105
Number with an unused Flip Flop	0	39
Number with an unused LUT	0	9
Number of fully used LUT-FF pairs	0	57
Number of unique control sets	0	7

IO Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of IOs	84	46
Number of bonded IOBs	0	46

Specific Feature Utilization	Verilog Implementation	Vivado HLS (implementation)
Number of BUFG/BUFGCTRLs	1	1

Timing Summary for Huffman Encoder	Verilog Implementation	Vivado HLS Implementation
Speed Grade:	-3	-3
Minimum period:	No path found	3.914ns
Maximum Frequency:	No path found	255.522MHz
Minimum input arrival time before clock	No path found	4.733ns
Maximum output required time after clock	No path found	6.817ns
Maximum combinational path delay	No path found	5.260 ns

[c.] TABLE 3. Utiization Estimates for Decoder in C and RTL

B. *MD5 Results*

Target Board: Spartan 6 xc6slx45-2-csg484

- Resource Utilization Summary

Slice Logic Utilization		
Number of Slice Registers	1920 out of 54576	3%
Number of Slice LUTs	1725 out of 27288	6%
Number used as Logic	1677 out of 27288	6%
Number used as Memory	48 out of 6408	0%
Number used as RAM	48	

Slice Logic Distribution		
Number of LUT Flip Flop pairs used	2501	
Number with an unused Flip Flop	581 out of 2501	23%
Number with an unused LUT	776 out of 2501	31%
Number of fully used LUT-FF pairs	1144 out of 2501	45%
Number of unique control sets	18	

IO Utilization		
Number of IOs	139	
Number of bonded IOBs	139 out of 320	43%

Specific Feature Utilization		
Number of BUFG/BUFGCTRLs	1 out of 16	6%

[d.] TABLE 4. Utiization Estimates for MD5

- Timing

Timing Summary
Speed Grade: -2
Minimum period: **13.222ns** (Maximum Frequency: **75.629MHz**)
Minimum input arrival time before clock : 10.575ns
Maximum output required time after clock : 4.118ns
Maximum combinational path delay : No path found

TABLE 5. Timing Summary for MD5

Throughput Calculation

Start Time – Time when input is supplied to MD5 module

End Time – Time when output is ready

Number of cycles = End time - Start Time/ Clock period

= (3505 ns - 10 ns)/20 ns = 3495 cycles

Throughput = Number of bytes x Max Frequency / No. of cycles

= (5 x 75.629MHz) / 3495 cycles = 108195.99 bytes/s

ACKNOWLEDGMENT

We extend our sincere gratitude to Professor Anupam Chattopadhyay and Professor Vivek Chaturvedi, for giving us the opportunity to work on such intuitive real-world problems and learn a lot in the process. We would also like to thank the Teaching Assistant, Debjyoti Bhattacharjee for his professional guidance, continuous support, constructive criticism and timely help, throughout this project, without which we couldn't have completed the project on time. The contents that were taught during the lectures and lab sessions of the course "Algorithms to Architectures", coupled with the numerous reference materials shared by the professors gave us a lot of insight and helped us understand the rudiments of the subject with great ease.

REFERENCES

[1] http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

[2] https://en.wikipedia.org/wiki/Cryptographic_hash_function

[3] https://tools.ietf.org/html/rfc1321

[4] https://en.wikipedia.org/wiki/Huffman_coding