FPGA Implementation of Whirlpool and FSB hash algorithms in Bluespec

Kyle Fritz
Jingwen Ouyang
Jeff Simpson

May 15, 2009

Contents

1 Introduction

The Cryptographic Hash Algorithm Competition is a public competition held by the National Institute of Standards and Technology to develop a new cryptographic hash algorithm in response to a vulnerability found in SHA-1 and to serve as an eventual successor to the SHA-2 family of algorithms. The new hash algorithm will be called "SHA-3". NIST will accept public comments on the candidates before selecting finalists.

The SHA-3 Zoo is the collection of algorithms submitted to the competition. The purpose of the zoo project is to help evaluate the algorithms publically and provide an overview of the results. The evaluations will include checking for accuracy of the algorithm, the ability of the algorithm to withstand known attacks, and the speed and resource usage of hardware implementations of the algorithm.

In this project, we will implement the Fast Syndrome-Based (FSB) hashing algorithm, which is one of the candidate SHA-3 hashes in Bluespec for use on an FPGA. Additionally, we will implement the Whirlpool hashing algorithm, which is used by the FSB algorithm.

The goal of this project is to use the process and results of the hash algorithm implementation to analyze and make recommendations about the FSB hash algorithm to the community.

2 Test Harness

The test harness is responsible for providing a number of functions to the overall design. It provides a simplified standard interface to the hash function, an abstraction from the memory interface, a simple and fast testing mechanism, and FPGA interfacing.

2.1 Simplified Standard Interface

The test harness is a wrapper for the hash function, providing it with a simplified standard interface and testing mechanisms. The interface for a hash function consists of the following methods:

Methods	
Put Length	Provide the message length (in bits) to the hash function
Put Word	Provide the next sequential input word to the hash function
Get Load Request	Handle a lookup table load request from the hash function
Put Load Response	Return the lookup table response to the hash function
Get Hash Result	Retrieve the Hash Digest results from the hash function

The hash function will expect that once it is given a message length, it will then be provided with enough words to cover that amount of data. The hash function is also able to make requests to the lookup table by address and receive responses which preempt data loading. When the hash has completed, the test harness will get the hash result from the hash function.

This simplified hash interface allows multiple hash implementations to use the same test harness and allows the harness to change independanrly of the hash functions without requiring any changes to the hash functions.

This common hash interface allows the hash functions to be replaced by stand-in stub functions to allow testing of the test harness itself. A fake hash which returns a constant hash or the result of a lookup table request was developed to verify that the harness was working properly before development of the full hash algorithms had completed.

2.2 Memory Organization

In the FPGA hardware, we used four segments of memory: a 32KB segment of RAM to store the input word and size, generated from the test case data, a 32KB segment of RAM to be used by the test harness to communicate with the NIOS, a 4KB segment of RAM to be used by NIOS for the stack, and an 8MB segment of flash ROM to be used by the lookup tables as needed. The memory map is shown here:

Address	Size	Data
`0400000:040105F`	4KB	NIOS Stack
`0410000:0417FFF`	32KB	Input Message, loaded from VMH or HEX file
`0440000:0447FFF`	32KB	Test Harness and NIOS communications (CBus status and command registers)
`1000000:17FFFFF`	8MB	Lookup Table

2.3 Memory Interface

The test harness will start by requesting from a fixed memory location the size of the message, and the location to find the data and store the result. Using the message size as a counter, sequential addresses starting at the data memory address are requested, and the responses are passed into the hash function. When the hash has completed, the output digest is loaded into the result address in the same fashion.

In the FPGA, the different memory segments are wired to different sections in hardware to correspond to the RAM and Flash. In simulation, only one memory space is needed, so the lookup table and input memory are loaded into the same memory, at the same addresses used in the FPGA.

2.4 Testing

One of the most important features that the test harness must provide is a simple and fast testing mechanism. Comparing the hash results to a reference hash result is the most frequent operation in the development of a hash implementation, and it is important that it be quick and exhaustive, to make sure that a design changes does not effect the correctness.

To achieve this, every stage of the hash generation was scripted, from the generation of input verilog vmh files to the generation of the reference hash and the comparison. Using a single command allowed us to test the hash function against a full suite of test cases including a zero-length message, word-aligned messages, multiple-word messages, and messages whose length does not fall on an even byte boundary.

2.5 FPGA Interface

The test harness interfaces with the FPGA to load the input message, load the lookup table, start the hash, and get the results.

The input message is loaded into the FPGA as an Intel HEX file loaded into the firmware (SOF) image. This file is generated from a Verilog VMH file containing the message size, the message, the data address and the result address. These are loaded into a RAM segment in the FPGA which can be addressed by the test harness.

The lookup table is generated as a binary file containing exactly the memory contents needed by the hash algorithm. This is converted to an Altera flash image using a utility provided with the NIOS IDE and loaded into the flash prior to running the hash program.

The NIOS processor handles FPGA system controls of starting the hash and retrieving the results. CBus command registers are used for the status and command registers, allowing NIOS to poll the status of the system and command the system to start processing. When the hash has completed, the data stored in the result address is requested by NIOS and sent over the serial UART to a connected PC, which is recording it into a log file.

2.6 Support Utilities

A number of support utilities were written to assist with the test harness data generation and translation.

2.6.1 data2vml.pl

This utility converts a raw binary file into a Verilog VMH file that can be read in by Bluespec simulation code and addressed in a manner that is identical to the way memory is interfaced in the FPGA. The VMH files have a particular format that is specific to this project, but are easily generalizable.

Address	Data
`0002:0003`	Message size (in bits) is stored in two 32 bit words (total of a single 64 bit size).
`0004`	Address of the data start location is stored. The default value for this is 0x16 (the address directly following the result location)
`0005`	Address of the resulting hash location is stored. the default value for this is 0x06 (the address directly following this location)
`0006:0015`	Intentionally unused, as they will be used to store the resulting hash, by default
`0016:7FFF`	Message data, broken up into 32 bit words, in ASCII hex digits
`400000:end`	Optional look-up table. This location translates to the location of the flash ROM in the FPGA, and allows the simulation to provide the correct results.

2.6.2 hex2bin.pl

This utility converts a text file containing ascii hex digits (0-9, a-f, A-F) into a raw binary data file for use as an input as a test-case. This is helpful for easily defining the raw data to be hashed without having to use a standalone hex editor. Whitespace and non-hex digits are ignored.

2.6.3 truncate.pl

This utility takes an ascii text file and removes the end of file marker (0x0A). This is used for generating data files can be run through the reference implementationa and simulation to match the published hashed strings. This was especially important for Whirlpool which provided a series of reference hash strings (empty string, single letter “a”, the letters “abc”, the letters “acd”, one million letter “a”s, among others). The additional 0x0A at the end of the text file would produce faulty results.

2.6.4 vmh2intel.pl

This utility converts a Verilog VMH file into an Intel HEX file. Intel HEX files are the format needed for loading the input message data into the FPGA. Addressing and checksums are properly handled for all addresses up to 4 hex digits. This utility should function in a general case for any Verilog VMH file.

2.7 Architecture Details

2.7.1 mkCoreFPGATH

The main function of the FPGA Test Harness module is to handle emulating the FPGA for simulation purposes. This module uses an AvalonRegisterFile to simulate the memory on the FPGA and modifies a CBus register to signal the hash function to start processing in the absense of a NIOS controller.

2.7.2 mkCore

The core module handles all the memory transfers between the FPGA (or FPGA Test harness) and the hash function. It also handles communication with NIOS processor for status and commands. It is implemented as a state machine which will cycle through the following states:

1. Wait For Start Command: In this state, the processing is stalled while waiting for either NIOS or the FPGA Test Harness to signal the processing to start. In a complete system, this would signal that data has been loaded and is ready to be processed.
2. Request/Get Size: In this state, the memory is queried to retrieve the size of the input message (in two 32-bit words, for a total of a single 64 bit message length). This length

is provided to the hash function and used for a counter to signal when the data has been fully imported.

3. Request/Get Data Address: In this state, the data address is retrieved from memory. The location to find the data address is hard-coded, but the address stored in that location can be any memory location addressable by the FPGA. This allows the system a lot of flexibility in message handling.

4. Request/Get Result Address: In this state, the result address is retrieved from memory. The location to find the result address is hard-coded, but the address in that location can be any memory location addressable by the FPGA. This address can be the same as the data address, if the desired functionality is to overwrite the input message with the resulting hash to save memory.

5. Request/Get Memory: In this state, the message is retrieved from memory one word at a time until either the message has been completely retrieved, the end of available memory has been reached, or a table lookup has been requested.

   (a) Initiate Table Lookup: While in the Request/Get Memory state, a table lookup request can be issued by the hash function. If this occurs, the Request Table Lookup rules will fire instead of requesting normal message input data.

   (b) Request Table Lookup: These rules will request memory from the lookup table instead of the message input memory. An offset value is hard-coded into the mkCore module to allow the hash function to index from zero, rather than having to know the location of the lookup table in memory.

6. Wait For More Data: In this state, the processing is stalled with the status register set to a "waiting for more data" condition. In a complete system, this would signal to the source of data that more data is needed. It should respond by refilling the same data locations which were already used, then setting the command register to a "continue" command, to indicate that more data is now available. The processing will continue at this point.

7. Write Hash: When the hash function has completed and returned a digest to the core module, it will write the hash into the result address memory, as well as display it to the screen to be logged into the simulation log file. Once it has been saved to memory, the status register will reflect that the data is available, and NIOS will be able to retrieve that data and display it to the UART.

2.7.3 NIOS

The NIOS processor handles the task of waiting for the hash function to be ready, commanding the hash to start, waiting for the hash to complete, and retrieving the results. Additionally, the NIOS provides a message input generator functionality for testing messages larger than the available memory size of 32KB. This is accomplished by reloading the memory space with every request for more data made by the hashing function.

Normal Operation:

1. Wait for the hash function status register to indicate that it is "ready"
2. Set the hash function command register to "start"
3. Wait for the hash function status register to indicate that it is "done"
4. Retrieve from memory the result address
5. Retrieve the hash digest from the result address and send it as ASCII text through the serial UART, to be captured into a text file log for testing.

Message Input Generator Operation:

1. Wait for the hash function status register to indicate that it is "ready"
2. Load the data memory with the generated data.
3. Store the start location to the "data address" location
4. Store the total message size (in bits) to the "message size" location
5. Set the hash function command register to "start"
6. While the hash function status register is not "done"
   (a) If the hash function status register indicates "more data requested", reload the memory space with more data (or allow it to re-use the same data), and set the command register to "start"
7. Retrieve the hash digest from the result address and send it as ASCII text through the serial UART, to be captured into a text file log for testing.

3 Typical Hash Algorithms

A typical hash algorithm can be conceptually broken up into three stages: the Pre-Processor, Compression, and Finalization.

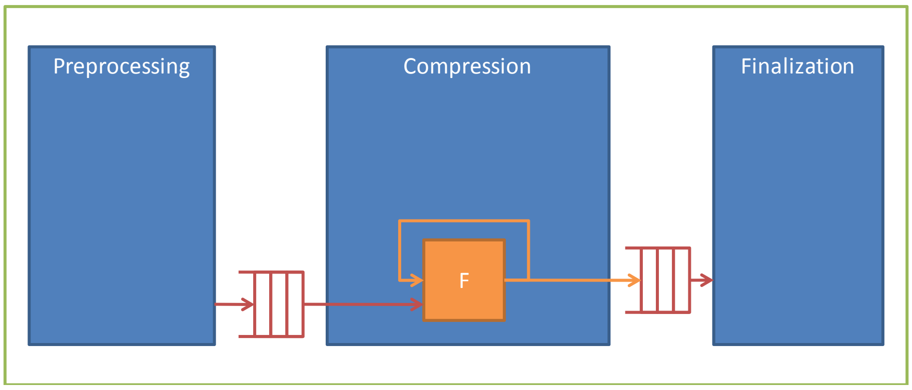

Figure 1: Block diagram for a typical hash function

In the Pre-Processor, the *Input Message* to be hashed is used to generate a *Message Block*. Typically, the input message can have any arbitrary length and will be delivered in small equally-sized pieces we refer to as *Input Words*. The *Message Block* is usually a fixed length. As the *Input Words* arrive, they are combined and/or split to create the fixed-length *Message Blocks*. When the *Input Message* has been completely assigned to blocks, the final un-filled block is usually padded to the proper length using 0 or 1 bits, the message length, or some other data determined by the algorithm.

In the Compressor, a function is applied to the *Message Block* from the Pre-Processor and the current state or running hash inside the Compressor. This function is typically the most complicated portion of the hashing algorithm. This function will continue to loop over itself, using the output as the running hash until the final Message Block is processed.

In the Finalization stage, the output of the Compressor has some final function performed on it to create the *Hash Digest*. In some hash functions, the finalization stage does not contain any function at all, or can contain a simple truncation operation.

4 Whirlpool Hash

Whirlpool is a cryptographic hash function designed by Vincent Rijmen (co-creator of the Advanced Encryption Standard) and Paulo Barreto in 2000. Since that time, this hash has undergone several changes to improve its security. The final version was adopted by the International Organization for Standardization (ISO) in the ISO/IEC 10118-3:2004 standard.[1] Whirlpool has been around for couple of years and has been analyzed thoroughly. Reference implementations (Java, Matlab, and C, among others) are available in the public domain free of charge. There is not currently a Bluespec version. Although Whirlpool is not a SHA-3 candidate, it is still beneficial to develop a Bluespec implementation. Additionally, the other hash function, FSB, uses Whirlpool at its final step. Implementing Whirlpool ourselves can also avoid the complication of interfacing unfamiliar implementations.

This section provides a brief description of the Whirlpool algorithm. This algorithms Bluespec implementation is also described. Finally, the hardware implementation strategy and some design trade-offs are discussed.

4.1 The algorithm

Whirlpool operates on messages less than 2^{256} bits in length, and produces a message digest of 512 bits. It uses Merkle-Damgård strengthening and the Miyaguchi-Preneel hashing scheme with a dedicated 512-bit block cipher called W. This consists of the following. First, the message to be hashed is padded with a single 1 bit and some zeros to make the total length to be $512 \times N + 256$ bits, where N is an integer. Then the original input message length is appended to the end forming a bit string of size $512(N + 1)$. This bit string is divided into a sequence of 512-bit message blocks M_1, M_2, $\ldots M_t$, which is then used to generate a sequence of intermediate hash values H_0, H_1, $H_2 \ldots H_t$. (By definition, H_0 is a string of 512 "0"-bits). An internal state S_i is also generated by the cipher W. This S_i is then XORed with M_i and H_{i-1} to compute H_i.[1]

The heart of the Whirlpool hash function is the cipher W. In its original design, it used a randomly generated selection box to compute S_i. As a result, it lacks internal structure, and is difficult to implement in hardware. In the final version of Whirlpool, a recursive structure is used. This new 8×8 substitution box (S-box) consists of a few smaller 4×4 "mini-boxes": the exponential E-box, its inverse, and the pseudo-randomly generated R-box, shown in figure 2. The efficient number of rounds of iterations over this S-box (generally set to 10) is carefully calculated for security. For more mathematical details, please refer to Whirlpool documentation.[2]

[1]http://www.larc.usp.br/ pbarreto/WhirlpoolPage.html

[2]The Whirlpool Hashing Function $\leftarrow$ reference doc

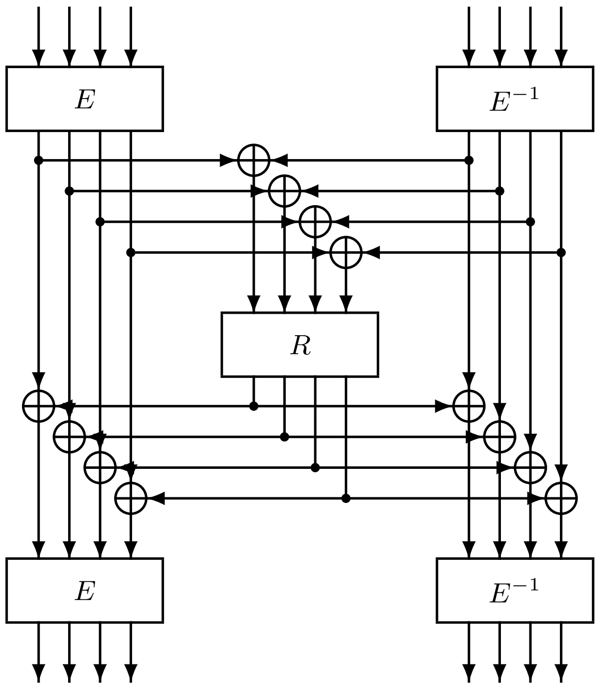

Figure 2: Structure of the Whirlpool S-box.

4.2 Implementation

As mentioned earlier, a typical hash implementation design consists of three parts: preprocessing, compression, and finalization.

4.2.1 Pre-Processing

The preprocessor's function is to take in a message split into arbitrarily sized input words and split or combine the words into arbitrarily-sized message blocks. These message blocks are forwarded to the next stage as they are filled. When the end of the input message is reached, the data in the final block has a single 1 bit appended to it, and the end of that word is padded with zeros and the message length, occupying a specified number of bits.

The difficulty in implementing a preprocessor function is handling the tradeoff between complexity and speed. As the message size, message block size, and input word size are not necessarily lined up evenly, one cannot simply shift each new input word into the message block until the input words have been exhausted - there will be a number of points where only a portion of the input word will fit into the current message block, while the remainder will need to be saved for the following block.

The simplest method of designing the preprocessor is to have it handle one bit at a time, constantly checking the remaining message length, remaining block length, and remaining word length. This function will take 1 cycle per message bit minimally, which makes it the slowest option for preprocessing, but the logic required to handle this design is very minimal.

Following is a set of simple rules which describe the operation of the Preprocessor function, used by both the Whirlpool and FSB hash functions:

1. If there are remaining bits of the input message, remaining bits of the input word, and remaining space in the message block, shift one bit of the input word into the message block.
2. If there are no remaining bits in the input word, and there are remaining bits in the input message, retrieve the next word.
3. If there are no remaining bits in the current block, send the block.
4. If there are no remaining bits in the current message:
    (a) Append a 1 to the message
    (b) Pad on the right with zeroes until only the space needed for the message length remains
    (c) Append the message length

This function was designed to minimize area by only using a single 1-bit shifter for loading message bits. It uses a total of 346 combinational functions and 1,423 registers. This is very small compared to our original design, which used 13,756 combinational functions and 6,499 registers.

The trade-off of speed for size was necessary to fit the Whirlpool hash into the FPGA as a part of the FSB algorithm, however, the preprocessing happens in parallel with the compression stage of the algorithm, which will take longer to process a message, thus minimizing any performance loss seen by degrading the preprocessing algorithm.

4.2.2 Compression

As it mentioned in section 3, the compression module uses a feed-back structure. There are three main operations for each incoming message block, shown in figure 3. First, the init stage takes in the message block, converts the message block into an 8 bit $\times$ 64 vector M_i, and resets internal state S_i to 0. The process buffer stage computes the internal state S_i using the S-box in figure 2. Finally, the finalize stage XORs M_i, S_i, and H_{i-1} to get a new intermediate hash H_i. Depending on whether this message block is the last one or not, H_i is either sent out as a final result or is looped back for computation of H_{i+1}.

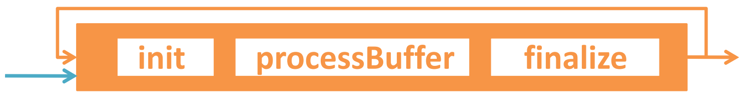

Figure 3: Structure of compression module

The most complicated part of this module is to implement the S-box in figure 2. Instead of figuring out its operation from the reference documentation, we decided to convert the reference C implementation into Bluespec code faithfully, which guarantees its correctness. This sounds relatively simple and easy to do. The simulation result of the converted code comes out correct without too much debugging. However, the C implementation is highly efficient in timing. It does a lot of operations all at once in one single cycle, which requires a lot of logic elements and wouldnt fit onto the Altera DE2-70 FPGA, which contains only about 68000 logic elements. To reduce the hardware usage, a few strategies were used. First, instead of using a for-loop to iterate the S-box 10 times in a single cycle, a counter is used do one round of S-box operation at a time. This way it takes 10 times longer to finish it, but reusing hardware saves a lot of logic and shortens the critical path. After that, the S-box operation is split into multiple cycles, so that only one XOR-branch operates at the same time, shown in figure 4. Also, S-box uses a 64 bit by 256 bit by 8 bit (total of 16 KB) table to look up values of the E-box. In order to conserve logic elements, this table is stored in SRAM, which allows only one table lookup per cycle. After these changes, Whirlpool was able to fit onto FPGA. However, it now takes 1280 cycles to complete the original for-loop. This 1280 cycles is comprised of roughly 10 (number of rounds of S-box iteration) $\times$ 16 (number of XOR branches for each S-box) $\times$ 8 (number of table lookups for each XOR branches), not including a few extra cycles for state machine transitions. This is a huge timing sacrifice for the sake of a reduced area design.

To reduce Whirlpool down to a size where it will fit with FSB in the FPGA, a few additional modifications were required. First, all the necessary FIFOs are reduced a size of one, instead of the default size of two. The use of logic elements in a FIFO is proportional to that FIFOs size. This change cut the logic usage nearly to half of the previous version. Secondly, any FIFOs that are used as an intermediate storage between modules are replaced by registers and ready bits, so that the functionality is preserved while using many fewer logic elements.

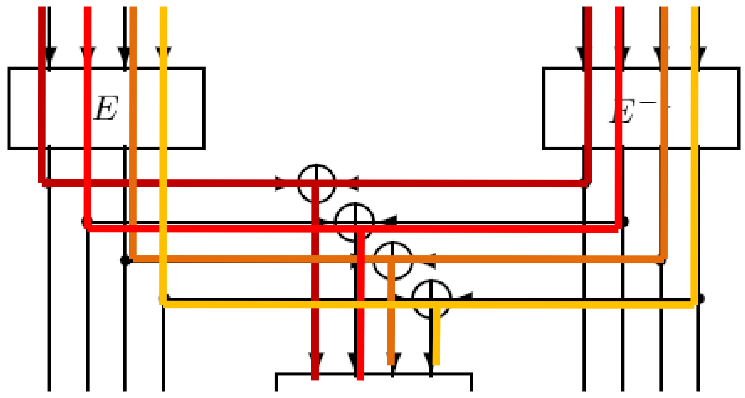

Figure 4: Only one S-box Xor-branch operates at the same time

One more modification was to avoid using multi-layer multiplexers. For example, Figure 5 (left) shows how to select the second 2nd element of the C_2 vector using double layered MUXes; Figure 5 (right) shows how to select the same element using only one MUX after the C_n vectors are concatenated into one large C vector. It may seem like a small change, but when each of the C_n vectors are 256 elements, with each element being 64 bits, it makes a big difference.

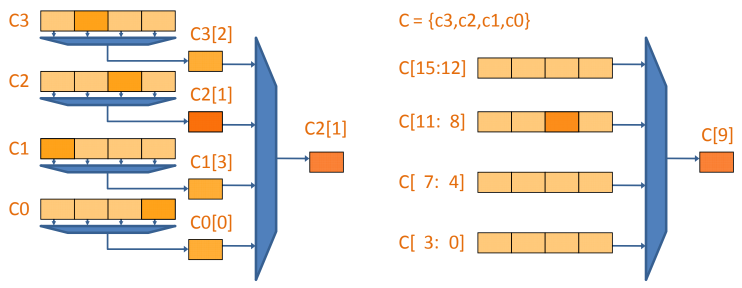

Figure 5: Two ways of indexing: Double layer MUX and Single MUX over concatenated vector

4.2.3 Finalization

Whirlpools finalization part is very simple. It does only one thing: unwrap the intermediate hash from its 8bit by 64 vector form into a 512 bit string as result. This is easily done at the end of the compression module using the following Bluespec code:

```
return unpack(truncate(pack(intermediate hash)));
```

4.3 Conclusion

The Bluespec implementation of Whirlpool was successfully simulated and verified in Bluespec compiler. It was also successfully put into the FPGA with correct results. There are many noticeable design trade-offs between speed and area. Efficiency in area is chosen over speed, as space is limited when FSB is including Whirlpool as its finalization module. To fit both hash functions on the FPGA at the same time was not trivial. More discussion of fit will be discussed in FSB section.

Table 1 lists the hardware usage of the finalized standalone Whirlpool implementation.

Component	Usage / Total (Percentage)
Total logic elements	10,723 / 68,416 (16 %)
Total combinational functions	8,252 / 68,416 (12 %)
Dedicated logic registers	6,940 / 68,416 (10 %)
Total registers	6990
Total pins	45 / 622 (7 %)
Total virtual pins	0
Total memory bits	435,968 / 1,152,000 (38 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

Table 1: Hardware usage for Standalone Whirlpool implementation

5 FSB Hash

The Fast Syndrome-Based (FSB) hash function was written by Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stphane Manuel and Nicolas Sendrier. It is a new algorithm, designed in 2008 for the NIST SHA-3 competition. At the time of this writing, no easy attacks existed, and the authors provide proofs of reduction to known hard problems. As it is still very new, there was no prior hardware implementation of this hash function. Our implementation in Bluespec should prove useful to the judges of the NIST competition.

Like many other hash functions, it uses a Merkle-Damgård domain extender. Because of this, and the use of primitive operations such as XOR and shift, FSB is touted as a simple algorithm to understand and implement.

This section provides a description of the FSB algorithm, as well as our Bluespec implementation. We also discuss the challenges FSB presented when implementing for an FPGA and the design tradeoffs we made.

5.1 The algorithm

The FSB variant we implemented produces message digests of 512 bits. It uses a Merkle-Damgård domain extender and a large-state compression function. Additionally, it uses Whirlpool as a final compression function.

Input messages are first broken into blocks of fixed size; in the case of the 512 bit variant of FSB, these blocks are 1240 bits long. The end of the message is padded with a single one bit, many zeroes, and the message length. This stream of blocks serves as input into FSBs compression function. The compression function also accepts the result of the compression function on the previous message block; this is 1984 bits long in the case of the 512 bit version.

The compression function combines the next message block with the previous compression function result, creating a smaller digest of bits. The inputs are broken up into very small pieces. For the 512 bit version of FSB, 248 pieces are created. Each pair, i, of pieces is then used to calculate a number, W_i, using some easy multiplications and additions to constants. W_i encodes two useful numbers: when divided by a constant, r, it serves as an index into a table; W_i modulo r serves as a shift amount. In the 512 bit version of FSB, a table of 1987 elements of 1024 parity bits of the digits of pi is used. The value obtained from the table is cyclically shifted by the shift amount, and is then truncated to the length of the compression function result. Each of these values is XORed to produce the output of the compression function for this iteration.

Once all the message blocks have been processed, the compression function output is passed to Whirlpool for one final compression.

5.2 Implementation

Our Bluespec implementation of FSB follows the same, three part structure as our Whirlpool implementation; there are preprocessing, compression, and finalization stages.

5.2.1 Pre-Processing

The preprocessing stage for FSB is almost identical to Whirlpools. It also needs to accept arbitrarily sized words. It only differs in the number of bits grouped in a message block and the number of bits reserved at the end of the last block for message length encoding.

5.2.2 Compression

The implementation of this stage follows closely from the specification. For each of the steps listed above, we have written a rule. FIFOs connect the results of these rules, allowing them to fire concurrently and pipeline the process. Because there is such a large internal state, there are many pieces to process for each message block, and this pipelining is essential for performance. The rules in this stage are:

1. dequeueBlocks
2. splitBlocks
3. dequeueIntermediates
4. splitIntermediates
5. wRule
6. memoryRequestRule
7. memoryResponseRule
8. shiftRule
9. truncateShiftedResultRule
10. xorRule

The dequeuing rules take the next message block and the previous compression result and store them in a register. The split rules extract bits from that register and form the pieces. The w rule computes the value of W_i for each piece. The memory rules issue and receive requests for the parity digits of pi, using the quotient of W_i and r. The shift rule cyclically shifts the memory response value by W_i modulo r. The truncate rule makes the shifted result the appropriate size, and the xor rule aggregates the results from all the pieces. This process is shown in Figure 6. The most noticeable aspect to this stage is the need for a memory interface. Our original design had no such interface. We planned on encoding the

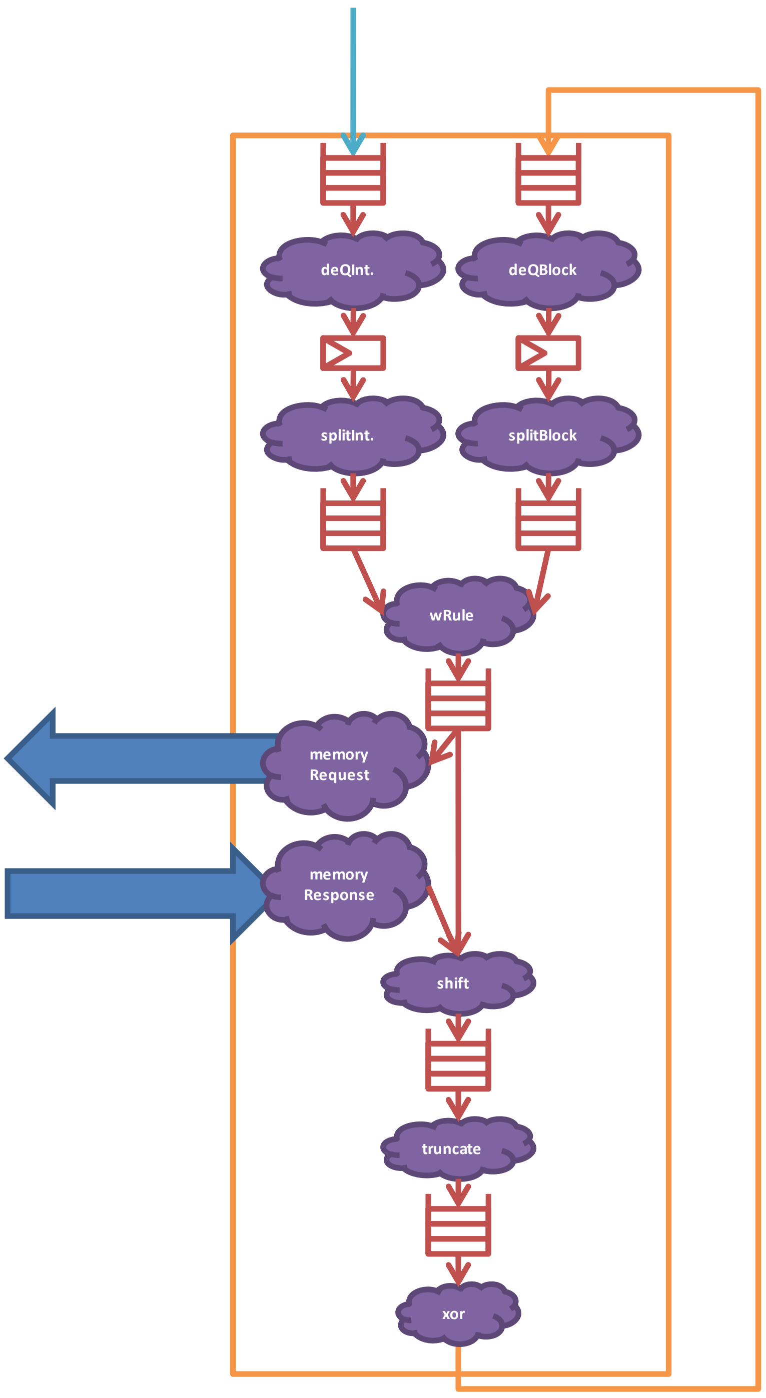

Figure 6: The organization of rules in FSBs compression function. The next message block and the previous compression function result are enqueued at the top of the diagram. The result exits the bottom.

parity digits we needed into a single Bluespec Vector# type. The Vector# type acts much like an array of bit strings. The 512 bit version of FSB requires 1987 elements of 1024 bits. We hoped that we could use this Vector# type to store all those elements, and that it would synthesize to make use of the ROM on the FPGA. While this approach produced correct values in simulation, the 249KB of data far exceeded the amount of ROM on chip. Therefore, we memory mapped the FPGAs 8MB flash memory to the address space and created an IMemHash interface that allowed the hash function to forward memory requests to the test harness. As the hash function has no knowledge of the memory organization, it issues requests starting at address 0, and the test harness offsets those requests before issuing them to the memory. The need for memory access slows down the compression function considerably. The latency of the memory is at least several cycles per request. For 1024 bits of data, grouped into 32 bit words, 32 memory requests are necessary to obtain the entire value from pi for each piece we need. There are 248 pieces per memory block in the 512 bit version of FSB, requiring nearly 8000 memory requests per message block. We also found that a large amount of our hash functions combinational area was used by this stage. Because FSB uses such a large internal state, with bit strings from 1000 to 2000 bits long, even very simple operations can be expensive in area. Because of area limitations, we were forced to avoid single-cycle barrel shifters and use an asynchronous multi-cycle shifter module. This module shifts by a constant amount every cycle, synthesizing as simpler hardware. This tradeoff, however, lessens our performance. We were also prepared to replace our division and modulus operators with multi-cycle modules, but were able to fit our design before that step became necessary

5.2.3 Finalization

The finalization stage for FSB is very simple to understand, but is also a very sizable portion of our area. This is because it is a complete hash function itself: Whirlpool.

Our initial approach was to leverage the polymorphic properties of our hash interface. We wanted to allow input words into Whirlpool of the size that FSBs compression function outputs. This would have made FSBs finalization stage very simple, inputting one word and waiting for the result. Although the polymorphism allowed this and produced a correct result in simulation, this caused Whirlpools preprocessing stage to be immense in size.

Therefore, to keep Whirlpools area low, FSBs finalization stage breaks up the compression function result into 32 bit words and inputs them one at a time into Whirlpool. The extra cycles required by this process is, fortunately, hidden to some extent; while FSBs finalization stage is continuing to input words, Whirlpool can start its compression on an earlier message block.

The result of Whirlpool is then returned to the test harness as FSBs result.

5.3 Conclusion

As stated earlier, our first approaches to implementing FSB failed to fit on the FPGA. We had problems with both too much on-chip memory usage and too much combinational logic. The tables below lists our resource usage before and after we made the area reducing changes.

Component	First Version	Final Version
ROM Bits	2,477,824	435,968
Combinational Logic	158,518	60,327

6 Building Instructions

6.1 Checking Out Code

The code and build tools for the Whirlpool and FSB hash algorithms are located in the 6.375 CVS tree (/mit/6.375/cvsroot).

To check out the whirlpool implementation code, use the following command:

```
> cvs checkout -r wp_final 2009s/groups/group2
```

To check out the fsb implementation code, use this command:

```
> cvs checkout -r fsb_final 2009s/groups/group2
```

Note: The only difference between the whirlpool and fsb code tags is the contents of the file "hashToUse.bsv" which will set up which hash to use. It is also possible to switch between the hash functions by editing that file directly and rebuilding.

6.2 Building/Synthesizing

To build the simulation, type `make`. This will generate the mkCoreFPGATH.exe simulation binary.

To synthesize, type `make bitfile`. This will synthesize the hardware and place and route for the FPGA device.

6.3 Testing in Simulation

To run the full simulation test-suite, run one of the following make targets:

`make check-wp` for Whirlpool, `make check-fsb` for FSB

This target will perform the following actions:

1. Build the reference hash implementation
2. For each test-case:
    (a) Generate reference hash
    (b) Generate data file from test-case source (binary, hex, or text)
    (c) Generate Verilog VMH file including the message size, data, and default data and result addresses. For FSB this will also include the lookup table
    (d) Run the simulation, generating a testname.out log file.
    (e) Search the log file for the output hash, saving it to testname.hash
3. Compare all the simulated hash results to the reference hashes

6.4 Testing on FPGA

To run the full FPGA test-suite for the FSB hash algorithm, the flash lookup table must first be loaded into the device. You may safely skip this step if you are testing whirlpool.

`make load-fpga`

This command will generate the Altera Flash image from the binary FSB lookup table and load it into the flash of the FPGA. nt To run the test-suite, run one of the following make targets:

`make check-wp-fpga` for Whirlpool, `make check-fsb-fpga` for FSB.

This target will perform the following actions:

1. Build the reference hash implementation
2. For each test-case:
    (a) Copy the NIOS code hex files into the place and route build directory.

   (b) Generate Verilog VMH file including the message size, data, and default data and result addresses.
   (c) Generate Intel HEX files from the Verilog VMH Files
   (d) Generate SOF files including the FPGA firmware, Intel HEX file and NIOS code.
   (e) Run the SOF file on the FPGA, collecting the serial output into testname.fpga.out.
   (f) Search the log file for the output hash, saving it to testname.fpga.hash

3. Compare all the simulated hash results to the reference hashes.

6.5 Additional Procedures

In addition to building the simulation, synthesizing the hardware, and running the full test-suite in simulation and on the FPGA, there are other procedures which can be used in this build environment.

Additional targets:

cleanhash: This will remove all the data, vmh, out and hash files, so that re-running `make check-wp` or `make check-wp-fpga` will generate fresh results.

testname.hash: This will generate the hash using the simulation for a particular test case

testname.fpga.hash: This will generate the hash using the FPGA for a particular test case

testname.wp.ref.hash: This will generate the Whirlpool reference hash for a particular test case

testname.fsb.ref.hash: This will generate the FSB reference hash for a particular test case

erase-flash: This will erase the flash device

load-flash: This will generate a flash image of the lookup table and load it into the flash device.

cleandocs: This will remove the automatically generated DVI and PDF documentation files

docs: This will regenerate the DVI and PDF documentation files from the LaTeX input