Implementation of a SHA2-256 hashing core in the synchronous language Quartz

Ammar Ben Khadra

March 5, 2013

Abstract

We present a high throughput implementation of the SHA2 hashing algorithm implemented in the synchronous language Quartz. We start with a basic implementation of the SHA2 standard for 256 bit width and then apply optimizations systematically. The final implementation can hash a block of 512 bits in 66 clock cycles giving a throughput of 1.24 Gbps on Xilinx Virtex-5 FPGA technology running at 160 MHz.

1 Introduction

Information security is a major concern in our "online" society. In order to take full advantage of what technology has to offer, we need to make sure that our transactions are conducted in a secure way. Cryptography lays at the heart of information security providing essential security mechanisms such as encryption and hashing. As opposed to encryption, hashing is a fixed size one-way function i.e. given a message of variable length, the hash function maps that message to a fixed size "hash". A key property of the hash is that the original message is computationally expensive to trace. Ideally, it's guessed only by brute-force trial of the message space.

The Secure Hash Algorithm version 2 (SHA-2) has been standardized by the American National Institute of Science and Technology in [6] as a successor to SHA-1 which has well known flaws. SHA-2 is currently widely used in many applications and protocols such as IPv6 and IPSec. High throughput implementations of SHA2 are much needed to handle the increased bandwidth demand. To this end, hardware implementation in the form of ASICs and FPGAs are viable options to process traffic at *wire-speed* in networking equipments. We present an implementation done in the synchronous language *Quartz* which is synthesized to *Verilog* and mapped to a Xilinx Virtex-5 FPGA.

We start by implementing the basic SHA-2 algorithm described in [6]. We apply then a set of optimizations in order to reach an implementation that can handle 1.24 Gbps. To the best of our knowledge, this throughput matches the performance of the second best SHA-2 commercial implementation available [1]. Additionally, we discuss some inefficiencies in Quartz to Verilog synthesis process and provide some suggestions for improvements.

2 SHA-2 standard overview

We will highlight the important steps in the standard here. Interested readers should refer to the standard [6] for full details. The algorithm consists of two main stages discussed below.

2.1 Pre-processing:

This stage consists of the following steps:

1. Padding of message to ensure that the total message size is multiple of block size which is 512 bits.

2. Parse the message into N blocks. Note that each block consists of 16 words M_i each of 32 bits width.

3. Calculate the schedule values of each block W_i where $16 \leq i \leq 63$. Note that the first 16 schedule values W_i are equal to corresponding words M_i.

2.2 Hash computation:

Hash computation is a serial process done block by block. The result of hashing a block is an intermediate hash H_i where $0 \leq i \leq 7$ that is used by the next block. Hashing of the last block in the message results in eight hash words H_i that are concatenated to provide the final hash of 256 bit width.

3 Implementation of standard

We have implemented the standard in a Quartz module called `UserModule`. A wrapper module `TestModule` has been used to provide the input stimuli and check the resulting hash. SHA2-256 algorithm can handle messages of a bit length up to 2^{64}. We do not check that in our `UserModule` implementation and assume that input is correct. The pseudo code of the algorithm is given in Listing 1.

```
initialize_constants(Ki)
always
begin
    Initialize(Hi);
    while (!msg_end)
    begin

           If read_words_num < 16
                  Wi[read_words_num] := read_new_word();
                  read_words_num := read_words_num + 1;
           else
                  HashBlock(Wi, Ki);
    end
    //Message ended, now start padding
    If !fit_msg_length
           Pad block with zeros only
           HashBlock(Wi, Ki);
    Pad a new block with zeros except last 2 words.
    Append total msg length at at last 2 words.
    HashBlock(Wi, Ki);
    Send final Hi to output
end
```

Listing 1: Basic UserModule

Note that the basic `UserModule` implementation make use of the submodule `HashBlock` which is provided in Listing 2. This implementation maps the standard without using the standard's temporary variables `temp1` and `temp2` which is more efficient as one inner loop iteration can be done now in one clock cycle (one `pause` statement is used). An inner loop iteration would have needed 4 clock cycles otherwise. Also, some details have been omitted for simplicity. The implementation uses the Quartz constructs `next` and `pause` to better illustrate inefficiencies later.

```
1 Copy intermediate hashes Hi to temporary hashes Ti,
2  i in {a,b,c,d,e,f,g,h}.
3 // Calculate schedules, first 16 are equal to msg words Mi
4 for (i in 16 ..63)
5     next(Wi[i]):= sigma0(Wi[i-15]) + Wi[i-7] + sigma1(Wi[i-2])
   + Wi[t-16];
6     pause;
7 // Do inner iteration
8 for(i in 0 .. 63)
9 begin
10    next(Th) := Tg;
11    next(Tg) := Tf;
12    next(Tf) := Te;
13    next(Te) := Td + sum1(Te)+ Ch(Te,Tf,Tg) + Th + Ki[i] +
   Wi[i];
14    next(Td) := Tc;
15    next(Tc) := Tb;
16    next(Tb) := Ta;
17    next(Ta) := sum0(Ta) + Maj(Ta, Tb, Tc) + sum1(Te)+
   Ch(Te,Tf,Tg) + Th + Ki[i] + Wi[i];
18    pause;
19 end
20 Calculate next Hi given Ti and current Hi.
21 pause;
```

Listing 2: Basic HashBlock

Note that the functions `sigma0`, `sigma1`, `sum0`, and `sum1` are functions $\sigma_0, \sigma_1, \Sigma_0$, and Σ_1 respectively defined in the standard. Inspecting the code we can see that (1) T_A calculation in line 17 represents our critical path. In addition, (2) by doing a simple calculation it can be noted that we need $(48+64+1) = 113$ clock cycles to hash a block which is 512 bit length. Our goal in the optimization section should be working on those issues to reduce both the critical path length and the number of clock cycles needed to hash a block.

4 Optimization

4.1 Instantiate a module only once

We have identified in the previous section the two main issues to work on in the optimization process. We will deal with these issues in the next two subsections. Now, the issue of module Instantiation is discussed. If we have a second look at `UserModule` in Listing 1. We can see that `HashBlock` has been called three time in total in lines 12, 17, and 20; Which means that module `HashBlock` is instantiated three times i.e. three hardware instances are synthesized by Quartz2Verilog synthesizer. Actually, module `UserModule` didn't fit into our Virtex-5 technology due to the big hardware area requirement in our first synthesis attempts.

To reduce the hardware requirements we need to instantiate `HashBlock` only once. That can be done by running `HashBlock` in parallel with `UserModule`. Also, synchronization signals have to be used to make sure that `HashBlock` starts only when needed and that `UserModule` will wait until `HashBlock` is done before proceeding. Note that we needed to modify the model in order to obtain better hardware synthesis results. This brings up the issue of the right balance between abstraction and detail in a model-based design language which will be discussed later in section 5.

4.2 Optimize schedule calculation

By inspecting module `HashBlock` in Listing 2 we can have the following observations:

1. To generate a schedule value $Wi[i]$, we only need the last 16 schedule values. That is, a data structure of 16 words can be used instead of 64 words. Remember that each word is 32 bit width.

2. Calculating the schedule is independent from the inner-loop iteration i.e. we can reschedule schedule calculation to be right before when the schedule Wi is used in T_A and T_E calculation.

That being said, we refer to [5] where the design and schematic of this efficient schedule expander circuit is discussed. By rescheduling Wi calculation to be done in parallel inside the inner-loop iteration we can save 48 clock cycles. Effectively reducing the clock cycles needed in `HashBlock` to 66 clock cycles. Note that one clock cycle is needed to initialize `Ti` values at the beginning.

Core	Frequency	Slices	Throughput
Helion	221 MHz	319	1.71 Gbps
Cast	160 MHz	418	1.24 Gbps
Quartz	160 MHz	1056	1.24 Gbps

Table 1: Performance comparison on Virtex-5

4.3 Reducing critical path

The calculation of T_A represents the critical path of the inner-loop iteration of module `HashBlock`. Actually, T_A and T_E are the only elements calculated in each iteration, all other temporary elements are directly assigned. Calculation of T_A and T_E can be written as follows:

$$T_A^{i+1} := T_H^i + K^i + W^i + \Sigma_1(T_E^i) + Ch(T_E^i, T_F^i, T_G^i) + \Sigma_0(T_A^i) + Maj(T_A^i, T_B^i, T_C^i)$$

$$T_E^{i+1} := T_H^i + K^i + W^i + \Sigma_1(T_E^i) + Ch(T_E^i, T_F^i, T_G^i) + T_D^i$$

Note that the operation $T_H^i + K^i + W^i$ represents the maximum common term that can be rescheduled. That is, this term, let's call it α, can be calculated in the previous clock cycle, which effectively removes two addition operations from the critical path. Note also that $T_H^i = T_G^{i-1}$.

5 Quartz synthesis improvements

5.1 Efficient data-path synthesis

We have merged module `HashBlock` in `UserModule` and applied the aforementioned optimizations. The performance of the Quartz core has been compared to the leading commercial cores available, namely, Helion[2] and Cast[1]. The performance comparison can be seen in Table 1. It's worth to mention that Quartz core figures represents the result of synthesizing the wrapper module `TestModule` together with `UserModule` which adds area overhead. Also, `TestModule` uses 1 BROM in verifying the results which may have prevented us from effectively using a BROM for the constants K_i. Generally, Quartz core matches in throughput Cast core but lacks in area as it occupies more than double the area.

As noted before, SHA2 algorithm is a serial hashing process where each block depends on the intermediate hash done by the previous block. This removes the possibility that the increased throughput is actually due to more hardware running in parallel. Actually, we think that the area overhead is largely due to inefficient Quartz to Verilog synthesis. In that regard we can make the following observations:

1. The method by which guarded actions are synthesized has some inefficiencies. Hardware is not being efficiently re-used as at different stages of the algorithm.
2. Synthesis of data-path elements, carry-out adders in SHA2, is hard to control and has some unexpected results.

Discussion of the first issue is out of scope of this report. We will elaborate a bit on the second issue; The inner-loop iteration of SHA2-256 uses addition operation modulo 2^{32}. That can be implemented in hardware by means of a 32 bit carry-out adder which adds two (or more) 32 bit numbers and discards the last carry to keep the result to 32 bit width. To implement that in Quartz, while satisfying the strong typing system, we need to add macros as depicted in listing 3. Note that `word_size` in SHA2-256 is 32 bit.

```
// Two input carry-out adder
macro Add(a,b) = (nat2bv(bv2nat(a) + bv2nat(b), word_size);

// Three input carry-out adder.
macro Add3(a,b,c) = (nat2bv(bv2nat(a) + bv2nat(b) + bv2nat(c),
    word_size));
```

Listing 3: Carry-out adder in Quartz

The result of synthesizing Add3 has one 34 bit adder. To reduce that we have implemented 3 input addition by means of `Add(a, Add(b,c))` which resulted in 33 bit adders being synthesized. Note that the calculation of T_A, and T_E needs 5 and 4 input addition respectively. Also, note that α is calculated in the previous clock cycle and needs 3 input addition. The end result is that 4 34-bit adders, 11 33-bit adders, and 8 32-bit adders are being synthesized which is inefficient.

Additionally, inefficient data-path synthesis limits our ability to use various architectural techniques like pipelining and loop unrolling. Pipelining implementation of various stages can be found in [5] and [7]. Unrolling implementations e.g. 2X and 4X can be found in [4]. Keeping in mind that:

$$Throughput = \frac{Frequency * Block\,Size}{Number\,of\,Clock\,Cycles\,per\,Block}$$

Pipelining allows for higher frequency (and more clock cycles per block). On the other hand, unrolling reduces the highest possible frequency but a block can be hashed in less cycles (more iterations per cycle). Unrolling comes with high area overhead and bigger data-path. Quartz synthesis can be highly inefficient with unrolling.

5.2 Automatic parallelization

We have discussed in section 4.1 the importance of instantiating a module only once to avoid hardware redundancy in synthesis. One time instantiation can be done by running a submodule in parallel. Additionally, input and output arguments may have to be copied (registered input and/or output respectively) and signals `enable` and `done` have to be setup.

In practice, this procedure is not only inefficient as registers for inputs and/or outputs has to be added, it simply doesn't scale. Running few submodules in parallel can easily become complex and unmanageable. Also, changing the model that much for better hardware synthesis reduces the advantages of model-based design in the first place. We propose to automate the parallelization process as much as possible, while keeping the model structured. For example, multiple serial calls to `HashBlock` as in listing 1 should be automatically synthesized to one submodule. Multiple calls can be dealt with using multiplexers at input and/or outputs as needed. Nonetheless, calling the same submodule in parallel code can be more tricky to resolve and requires further investigation to check whether it's totally resolvable at compile time.

To this end, we propose adding meta-data in the form of *annotations* to submodule calls. It's a non-intrusive mechanism that helps the synthesis tool to e.g. identify instances of sub-modules. It can also help in better synthesizing the data-path such as specifying the adder's bit-width and type (carry-propagate, carry-lookahead, ... etc). Annotation syntax similar to the one available in Behavoir-Interaction-Priority model-based design language [3] can be very useful in improving synthesis results.

6 Conclusion

A high throughput implementation of the SHA2-256 standard has been presented. The standard has been implemented using the synchronous language Quartz targeting Xilinx Virtex-5 technology. The end throughput matches that of commercial cores which demonstrates the viability of Quartz and model-based design in general. Nonetheless, there is still alot of work to be done and many issues to be tackled.

References

[1] CAST Inc, Secure Hash Algorithm Core. *http://www.cast-inc.com/ip-cores/encryption/sha-256/index.html*. Accessed: 27/02/2013.

[2] Helion Technology, Fast Hashing Cores. *http://www.heliontech.com/fast_hash.htm*. Accessed: 27/02/2013.

[3] Verimag Research Center, Behavior Interaction Priority (BIP) component framework. *http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=*. Accessed: 04/03/2013.

[4] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. Improving sha-2 hardware implementations. In *In Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006*, 2006.

[5] L. Dadda, M. Macchetti, and J. Owen. The design of a high speed asic unit for the hash function sha-256 (384, 512). In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 70 – 75 Vol.3, feb. 2004.

[6] NIST. Federal Information Processing Standards publication 180-4. *http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf*, (March), 2012.

[7] Hoang Anh Tuan, K. Yamazaki, and S. Oyanagi. Three-stage pipeline implementation for sha2 using data forwarding. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 29 –34, sept. 2008.