,,,

Jacob Herbst (mwr148), Jonas Flach Jensen ()

Crypto in SME for FPGA

Advisor: Kenneth Skovhede

February 25, 2021

Contents

1 Introduction

2 Background

2.1 Field Programmable Gate Arrays

2.2 Synchronous Message Exchange

Synchronous Message Exchange (SME) is a programming model to enable FPGA development using high-level languages. SME is based on Communicating Sequential Processes (CSP) and at its core constructs as a strict subset of said process calculi, making use of the elements which has proven useful in hardware design[**?**]. Using the following concepts from the CSP model, SME can be derived:

- A program consists of a set of named processes.
- Each process runs on its own processor with no form of sharing with other processes.
- Concurrent processes can communicate using message passing with a `send(!)` and a `receive(?)` command. This message passing is Blocking and Non-buffered.

Without going into too much detail about the syntax and semantics of CSP[1], we can use the following syntax to describe a program.

`~x` PROCESS~, which assigns the PROCESS to the name x.

`x.in` is a compound name similar to an object field `in` of the object. Notice this abstraction makes the connection to SME and its C# implementation more obvious.

`x.out!y.in` This is the sending message passing. It will send y.in to x.out.

`x.out?y.in`. This is the receiving message passing. It will read x.out to y.in.

`x || y` will denote two concurrent processes, x and y.

Later we will show this can easily show abstractions of algorithms when using SME. SME has a similar notion of processes. There exist two types of SME processes, `simple process` and a `simulation process`. Of these, the simple process corresponds to a process in CSP as described above. Each simple process in SME will only share communication channels and constants with the other processes. For the communications channels, SME extends the concepts from CSP by using buses. Instead of using explicit naming for sources and destinations, each process will consist of a set of input and output busses that it can read and write to, respectively. Furthermore, these buses use broadcasting as means of synchronization instead of the blocking non-buffered approach. The broadcasting happens every clock-cycle on the internal clock. A bus is essentially just a collection of fields that can be read and written to depending on the process's access, merely a data transfer object. Here the syntax described above comes in hand. `message.text` would thus be the text field of the bus `message`. Corollary, we could define a very minimal process as such `[messageIn.text?messageOut.text]`, which would read the text field from the `messageIn` input bus and write it to the `messageOut` output bus. From these abstractions, one might be able to see how this effortlessly coincides with the hardware model.

2.3 A crypto library

2.3.1 Hashing

2.3.2 MD5

The Message-Digest algorithm MD5 is a reasonably simple one-way hashing function that produces a 128-bit digest specified in 1992 in RFC 1321[]. MD5 uses a Merkle-Damgård construction. The MD5 algorithm work by partition the input message into blocks of 512 bits. It is done by always padding the message with a single set bit followed by a series of 0's until the message = 448 mod 512. That is, even when the original message has a length of 448 mod 512, a 1' followed by 511 bits of 0's. Next, a 64-bit representation of the message length mod 2^{64}

is appended to the padded message. The digest will be calculated in a 32-bit 4-word buffer (A, B, C, D), with the initial value: A: 0x67542301 B: 0xefcdab89 C: 0x98badcfe D: 0x10325476 and we use the following functions corresponding to each of the four rounds: F(X, Y, Z) = XY $\wedge$ $\neg$ X Z It is worth noting that MD5 is not a very good hashing algorithm for cryptography, as collision attacks exist, but still show use for data integrity purposes and such.

3 Implementation

3.1 Approaches

3.1.1 naive

3.1.2 pipelined

3.2 MD5

3.2.1 naive

3.2.2 pipeline 1

4 Benchmarks

5 Discussion

6 Conclusion

References

[1] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666677, August 1978.