Using CλaSH for Cryptographic Hash Functions
Maikel Stortteler
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.g.j.stortteler@student.utwente.nl

ABSTRACT
In this research, CλaSH shall be put to the test and possibly improved. CλaSH is a new functional hardware description language, based on the functional programming language Haskell. CλaSH aims to provide a higher level of abstraction than traditional HDLs to the circuit designer. The test-cases will consist of specifying and compiling some popular cryptographic hash functions. There will also be a variant in which multiple hash functions share hardware components.

Keywords
CλaSH, CAES language for synchronous hardware, hardware description language, Haskell, high level abstraction, cryptographic hash function, MD5, SHA-1, SHA-2, WHIRLPOOL.

1. INTRODUCTION
Cryptographic hash functions are hash functions for which it is infeasible to create or alter an input so that it will generate a specific output (message digest). Also finding two different inputs that generate the same digest should be infeasible. These cryptographic hash functions have many applications in cryptography. They are used for authentication and digital signatures for example.

An application might be in network routers. Content centric networking is an alternative approach of how computer networks work. The idea is to concentrate on the data that should be transported and less on the exact location of where that data should come from. Different users might want to retrieve the same data. Network nodes could cache the data that passes them, so that when a request for data comes that has already been transmitted to someone else, the network node can return the data immediately instead of forwarding the request.

The goal is to minimize traffic on networks.

For content centric networking to work, data should be correctly identified. The user should know a description of the data he wants, along with the address it should originally come from. This is to ensure the correct data is transmitted. Cryptographic hash functions are the perfect solution for this description, as they describe the data in a highly efficient manner and provide security as well. They deliver a so called checksum. It would be infeasible for an attacker to provide malicious data with the same hash digest.

These hash functions are likely to be implemented in the hardware of the router or network node. They might be executed in a high frequency, so software implementations might not be sufficient.

These network nodes should be designed and built at some point, and as with all hardware, this can be done by describing the hardware in some hardware description language (HDL). Verilog HDL[7] and VHDL[2] are popular examples, a new language is CλaSH[3, 4, 9]. CλaSH is a functional HDL and uses the same syntax and semantics as the functional programming language[10] Haskell[11]. CλaSH is still in a developing stage and this research aims to aid the process.

Some applications of describing cryptographic hash functions in the HDL will be used as a form of black box testing. Cryptographic hash functions are actually functions, as the name suggests. Output can be described as a function on some input, with the note that inputs exceeding a certain length will be fed to the function in pieces. Functions lend themselves to be described in a functional language very well. Also specifically designed hardware implementations of cryptographic hash functions are available to compare with[8, 15]. There are multiple popular and assumable secure ones, which differ in algorithm, but also have a lot in common. Software implementations are widely available too.

2. BACKGROUND
This research focuses on CλaSH, with the involvement of cryptographic hash functions. A description of both will be given to help understanding the topic.

2.1 CλaSH
CAES language for synchronous hardware (CλaSH) is a functional hardware description language, being developed at the University of Twente. Its purpose is to provide a circuit designer a way to describe circuits in a more natural way than possible with language elements found in traditional hardware description languages. The traditional languages are good at describing detailed hardware properties, but generally lack the ability to work on a higher level of abstraction. A higher level abstraction is desired for the increasingly complex circuits.

CλaSH is based on the syntax and semantics of the functional programming language Haskell. Functional programming is a form of programming in which the outcome is described in a mathematical way. It avoids the concept of states and concentrates on the application of functions, as opposed to the conventional imperative programming style.

Haskell can solve complex tasks in far less lines of code than one would use in imperative languages. A great example is working on trees, like searching a tree. But also defining that tree would be very easy and straight-forward. Trees are widely used; take artificial intelligence for example, or compilers. With
that ease of use, there comes a downside. Programs are defined in a mathematical manner, but the algorithms would be generated by the compiler and the programmer would have to trust that compiler for finding an efficient solution. So in this perspective a positive point of imperative languages is that the programmer can exactly tell what the program should do.

Polymorphism is an important feature in CλaSH. Vectors can have any length and functions can thereby be performed on any width of data, but at compile time the width has to be known. Think of this like the 64-bit architecture of modern systems, at compile time CλaSH has to know that vectors have a length of 64.

The prototype CλaSH compiler can generate a VHDL specification of a given program. This program can easily be tested up front, as it is a viable Haskell program. The way CλaSH comes up with the synthesizable VHDL code is as follows. First the Glasgow Haskell Compiler is used. This is an open source Haskell compiler that provides the parsing and type checking. GHC compiles the code to Core, which is a small typed functional language and can be relatively easily processed. Then CλaSH transforms the Core code to a normal form, which directly corresponds to hardware. This normalizing includes variations of transformations typically found in lambda calculus and reduction systems. Higher order functions are reduced to first order functions and polymorphisms are concretized. The last step is a straightforward translation to a VHDL netlist.

Now hardware can be synthesized out of the VHDL netlist. A common way to prototype is by using field programmable gate arrays[15, 18]. FPGAs can be programmed to resemble a certain circuit, this is a quick and economical way to test the specified hardware.

2.2 Cryptographic Hash Functions

A cryptographic hash function is a hash function for which output changes unpredictable when input changes. It generates for a message of arbitrary length, in a deterministic procedure, a hash value of fixed size, also called message digest. An ideal cryptographic hash function makes it infeasible to conclude anything about an unknown message, given a hash value. The complexity of the hash function itself should be minimal in order to attain processing speed.

In security the functions are used in verifying mechanisms for the integrity of files and messages. Digital signatures embody hash values and password verification is usually bases on hash functions too, as passwords are not stored in cleartext.

The function should be able to withstand all types of cancrtanalytic attacks. The function must be:

- Preimage resistant: given \( h \), the equation \( h = \text{hash}(m) \) should be difficult to solve, i.e. difficult to find a \( m \).
- Second-preimage resistant: given \( m_1 \), it should be difficult to find a different \( m_2 \), such that \( \text{hash}(m_1) = \text{hash}(m_2) \).
- Collision resistant: finding two different messages \( m_1 \) and \( m_2 \) such that \( \text{hash}(m_1) = \text{hash}(m_2) \) should be difficult.

In the above, difficult should be read as impractical to solve within the time it could still be useful, years for example. Not solvable in asymptotic polynomial time is another way to express this.

Cryptographic hash functions consist typically out of a series of bitwise operations. AND, OR, XOR and bit round shifts are the operations performed on single bits and then’s addition.

Many functions have internal word-sizes of 32 bits. The functions work by taking a few words from the input message, perform operations on them, store a result and also use this result as a variable in a next round. These round cycles are performed until the whole message has gone through the function.

Addition can be seen as taking two words as integers and performing normal addition on them.

To assure that the compression function is one-way, the function is often based on block ciphers. This means that processed data is fed back into the function to add complexity. Diffusion is something to strive for, a small amount of the input should affect the whole output. Ideally a single bit-flip in the input causes half of the output bits to flip. That is, without some traceable pattern.

Popular hash functions are MD5[14], SHA-1[6, 13] and SHA-2[15]. The latter two are designed by the National Security Agency of the USA. MD5 is however considered cryptographically broken [17] and should not be used any longer for security applications. It is not collision resistant, collisions can be generated with a complexity of \( 2^{121} \).

A part of the SHA-1 scheme is displayed in figure 1. This shows one of the eighty rounds of the function. The nonlinear function \( F \) consists of logical operations.

Note that hash functions do not always have to be cryptographic. Hash that do not provide the security features

Figure 1. One round of the SHA-1 scheme[12].
exist and are typically faster to calculate. They find much usage in indexing data.

3. PROBLEM
The CλaSH compiler is still a prototype. It is not yet complete, recursion for example must still be added. Translations are not fully optimized, redundant hardware elements can be generated as an outcome. The possibility of not terminating is even a factor to take into account.

In most cases, software is slower than an equivalent designated hardware implementation. This is mainly because of the overhead of loading program rules to the processor, where a hardware implementation directly operates on the input data. So when considered cost efficient, hardware designs might include cryptographic hash functions in their circuits, think of routers for content centric networking. Therefore it will be useful to perform some tests on this topic and perhaps suggest some improvements to the compiler. The first research question of this research will therefore be:

- How well are cryptographic hash functions translated to hardware by CλaSH?

In order to be able to say something about this, results can be compared to known hardware implementations of the same hash functions. Points of interest will be the size of the hardware component and the speed that could be achieved. By size we relate to the actual size the hardware would take on some FPGA.

- How do the generated cryptographic hash function hardware specifications compare to specifically designed hardware implementations?

An important aspect of this research will be the feedback to CλaSH. Therefore:

- Should CλaSH be improved with respect to this research, and how?

The purpose of CλaSH is to provide the circuit designer with a higher level of abstraction. Popular HDLs could be straightforward to some degree in describing cryptographic hash functions. To add complexity and really test the higher level of abstraction, we will try to design a circuit that will perform a given hash function specified by some control input bits. The challenge is to share blocks of hardware, as many hash functions have internal operations in common.

- To what extend is it possible to combine hardware of different hash functions with CλaSH?

Sharing blocks reduces size, but extra overhead for controlling will increase size. It might happen that this application will be interesting for actual deployment, so:

- How space efficient is combining hardware for cryptographic hash functions?

3.1 Desired Results
Hardware description languages already exist. VHDL for example is widely supported and developed very well. The aim of CλaSH is to provide an alternative that offers a higher level of abstraction in a functional manner.

First of all, the CλaSH compiler should work properly and be optimized such that the most efficient hardware description is generated. The black box testing of this research aims to aid in this. Shortcomings in the compiler might be pointed out and improved.

Secondly, the abstraction level will be put to the test. It is imaginable that going from a single hash function to a circuit with multiple, while sharing components, will give overhead and complexity in the specification itself. That is, in traditional languages it will. CλaSH should provide a more natural way to achieve the sharing of hardware components. This shall be done by someone who has no experience in working with CλaSH or functional programming languages.

All together, the tests will be a good showcase for CλaSH.

4. RELATED WORK
Cryptographic hash functions are a popular topic, much literature can be found about them. Specific hardware descriptions of some functions can be found in [8, 15, 16]. CλaSH however is a new HDL and is still in a developing stage. Tests are already performed, but there is room for improvement. In the context of this research, there is no specific related work available.

5. APPROACH
The developers of CλaSH have released a virtual machine with CλaSH installed, in order to avoid version and dependency problems. This virtual machine is used for this research.

The hash functions used are: MD5, SHA-1, SHA-2 (the 256 bit version) and WHIRLPOOL[5]. These are the most common cryptographic hash functions, so there is plenty of background information to work with.

At the moment of writing, recursion in CλaSH is being worked on and cannot yet be used. CλaSH uses vectors instead of the basic lists in Haskell and their dimensions should be known at compile time.

For ease of use and a little more abstraction, programs were implemented in Haskell first. Recursion was avoided. Afterwards, the programs were adjusted to be valid CλaSH specifications.

Programs written are the four complete hash functions, but also just parts of them for evaluating purposes. Then an effort was made to combine some functions which would share hardware components.

VHDL is a language, so in theory, one could be able to read the VHDL specifications and draw conclusions. It is a low level language however, so this can become infeasible for bigger programs. Especially because the specifications are generated and not directly man-made, a VHDL-developing program was chosen to analyze our results.

Altera’s Quartus II[1] was the program of choice. This program can be used to compile the VHDL code and view the hardware in a block diagram. Simulation of the hardware can also be done, CλaSH provides a way to include some test inputs in the generated VHDL code.

A good way to test the generated VHDL is to write the actual hardware on a FPGA, such that the complete process of creating hardware with CλaSH will be complete. We will however see that due to time constraints this would have had no additional value. Cryptographic hash functions themselves could have been written to a FPGA, but there was no way of properly testing them.

6. RESULTS
The intent of this research was to test CλaSH and create hardware designs of cryptographic hash functions. One could expect technical findings about the two topics, but we’ll also include some experiences on working with CλaSH. Reason for
this is that we came across some unexpected points of notice and want to take these points into account on behalf of the research question:

Should CλaSH be improved with respect to this research, and how?

Therefore the results can be divided into two sections. The first is about our findings on using CλaSH, which might be somewhat subjective. The second part goes into more detail about the technical aspects.

6.1 Using CλaSH

First we discuss the programming of the functions, then their specific CλaSH implementation.

6.1.1 Programming

At the start of the actual programming part of this research, we were under the impression that CλaSH was nearly the same as Haskell, programming wise. We knew that recursion was not yet supported (this is under development at the moment of writing) and lists should be given a fixed length before compiling.

Beyond that, we’d see no limitations. CλaSH can handle higher order functions and also since programs would first be compiled to Core code by GHC, we assumed that it could handle imported modules too and would be able to translate all the Core code pieces derived from it.

The cryptographic hash functions were first implemented in Haskell and then translated to CλaSH by adding vector length definitions. We took into account that recursion should be avoided.

When the less complicated MD5, SHA-1 and SHA-2 were finished, WHIRLPOOL was implemented. All four functions were tested with a test input with known digest. The functions seemed to be correct.

To run the functions with actual input, a pre-processing piece of program had to be created. Standard input in Haskell for the hash functions would be in string format, these strings must be turned into 32 bit words. Here are some additional requirements. For MD5 for example, a single high bit has to be added to the end of the message and then low bits must be added until the message length is 448 modulo 512 bits. The last 64 bits contain the length of the message.

6.1.1.1 Combination of Hardware Blocks

MD5 and SHA-1 have the most common structure of the four functions. Parts of the programs are exactly the same. These parts could be shared and so the functions could be merged. Take this function for example, in which b, c and d are words:

\[
\begin{align*}
f &= (b \text{ and } c) \text{ or } ((\text{not } b) \text{ and } d)
\end{align*}
\]

Sharing of the parts was done by adding another input variable of type Bit to the functions. This bit would decide whether the one or the other function should be active. In an imperative language this can be done by a conditional statement, in Haskell guard conditions were used. Guard conditions are analogue to a case-statement.

The exact design of this merged function would depend on an interactive process of testing and trying different configurations, but a global outline was made and both functions turned out to be correct after testing.

6.1.2 CλaSH

Lists were used in the Haskell program for convenience, the functions on lists that were used should be one on one translatable to Haskell’s Vector. Vectors are required in CλaSH and should be defined with a fixed length. The vectors CλaSH uses are however different vectors than those originally in Haskell and it soon came clear that the original vectors couldn’t be used through an import. In fact, no imports at all were valid. All the functions and types that were used in the Haskell programs required a translation to CλaSH’s supported functions and types, in many cases completely altering the structure and operation of parts. There was no real one on one translation to the supported vectors.

These things together made it an unexpectedly difficult process to create the CλaSH equivalents of the already functioning Haskell functions.

As a result, we did not manage to create the full programs in CλaSH, only the core parts. The parts that are responsible of pre-processing an input message were not finished on time. The cryptographic hash function cores were complete however.

6.1.2.1 Observations

Instead of working with Haskell’s pre-defined 32 bit words, vectors of bits are used, with length 32. The Haskell variant had all used operations at its disposal, the CλaSH vectors required some additional work.

AND, OR and XOR functions are implemented by their bit-wise versions and CλaSH’s version of zipWith: vzipWith. The shift operation has to be defined by hand. Each bit in a vector had manually been placed in a different position in the resulting vector. For 32 bit words, this results in unclear code with many brackets. Code for a 4-bit word looks like this:

\[
\begin{align*}
\text{shift} &\colon \text{Word} \rightarrow \text{Word} \\
\text{shift} a &= (a(2) \leftrightarrow (a(1) \leftrightarrow ((a(0) \leftrightarrow ((a(3) \leftrightarrow \text{empty}))))
\end{align*}
\]

Loops and recursion could be solutions to this problem. One might even think of a script that will unfold loops specifying such pieces of code before compilation by the GHC. Code could look like this:

\[
\begin{align*}
\text{shift} &\colon \text{Int} \rightarrow \text{Word} \rightarrow \text{Word} \\
\text{shift} i a | i == 0 &= (a(3)) \leftrightarrow \text{empty} \\
\text{otherwise} &= (a(\lceil i/4\rceil)) \leftrightarrow (\text{shift} (i-1) a)
\end{align*}
\]

Then there is addition. Addition is not exactly a bit-wise operation, it is an operation on full words. Addition of two bits (XOR) can have a carry-out (AND) that is given to its neighbor as a carry-in. A zipWith does not work in this case and the first found solution was by pattern matching. A half adder was created, based on that a full adder. The full adder takes three
bits and returns a tuple of sum and carry-out. Now by specifying all equations of the full word, CλaSH finds the solution by pattern matching. Again loops and recursion would provide aid.

Later on in the process, we were told of two functions unknown to us. They were functions in CλaSH that converted a vector of bits to an unsigned integer and back. Addition was defined for unsigned integers, so this replaced the first found solution, providing a more readable and shorter code. The problem encountered here, was that these two functions were not in CλaSH’s reference manual:

```haskell
add2 :: Word -> Word -> Word
add2 a b = u2bv ((bv2u a) + (bv2u b))
```

Guard conditions in CλaSH work, but an otherwise-clause must always be added. The compiler will not decide to just let the signal die if no conditions are met.

Basic operations are now covered. Full functions are created by combining many of these operations. In general, programs look like:

```haskell
fold f init roundAdds
```

In this, f should be seen as a full round operation displayed in figure 1. The initial word values are contained in init and a vector with tuples of round constant and message word is provided through roundAdds.

### 6.2 Generated VHDL

CλaSH generates a separate VHDL-file for each function. These functions can be seen as components. This provides structure.

What immediately comes to notice is the naming of the variables / signals. The names start with something resembling the variable names used in the program, followed by a long incrementing number. This long number could have been replaced by numbers incrementing from zero.

Then there are the clock and reset signals. Every component receives these signals, they are always unused however. A clock signal would have been necessary for actual operation of the hardware, with messages bigger than 512 bits, so multiple cycles would be performed. For now, the signals are redundant. CλaSH assumes hardware with successive states, i.e. with clock cycles, is created. A hash function can be created out of pure combinatorial hardware however.

The overall VHDL-files look quite big. Beyond the clock and reset signals, we can’t really find redundant code. This may be because there is no, or because of our lack of VHDL experience.

The generated VHDL-files were compiled by Quartus II. Some pieces of code that only existed out of a single or a few components were made too. These parts of hardware description are easier to study in hardware block diagrams. These block diagrams are shown in Quartus II.

The generated block diagrams show, apart from the clock and reset signals, no redundancy. So either the Quartus II compiler optimizes the hardware, or no further redundancy is generated by CλaSH. Also unused functions in the module don’t show up in the diagrams. They are already discarded by CλaSH. This has been verified in the VHDL-files.

Functions AND, OR and XOR are created like they should be. They become an array of basic combinatorial hardware components. Also the shift function, which is just the rearrangement of wires on hardware level, comes out in the right form. For addition there are two possibilities. In case of converting to integer and performing addition, an addition block is shown without the display of internal components. This can be explained by the fact that VHDL contains such an addition as a basic block of hardware. Both cases deliver a valid and optimal result. Optimal as in there are no redundant hardware components, textbook implementations are created. See figure 2 for the generated half adder with a XOR-component and an AND-gate.

```
add2 a b = u2bv ((bv2u a) + (bv2u b))
```

Figure 2. A half adder.

This half adder is a small part of the full integer adder. The block diagram of the full adder is unfortunately too big to include. The diagram in which an integer addition included in VHDL is used is shown in figure 3.

```
add0 a b = add2 a b + out
```

Figure 3. Integer adder.

Complete cryptographic hash functions took a long time to compile and were hard to read in the block diagram. When generated just parts of the functions, they are generated in the right way. Block diagrams show correct and optimal hardware. The full functions are analogue to the smaller parts and seem correct.

About the combination of hardware: this cannot be done by just calling the same function multiple times. A different component is generated for each instance of a function call. Neither can guards be used. Each case will get its own hardware and at the end of the line, a switch is placed. This switch decides which case is used, depending on the guard conditions.

After a lot of testing, we decided that the sharing of components between functions cannot be done. Perhaps there is a far fetched way to accomplish this, but to the best of our knowledge we conclude that CλaSH cannot cope with this.

One might however wonder what the benefits would have been. Sharing of hardware blocks would mean the sharing of logical components in this case. If for every share a switch must be added, the result would be less efficient than the two originals together. A limitation of CλaSH has been found though.

Instead of running the hardware on a FPGA, a simulation can be done. Quartus II provides a way to simulate compiled VHDL and CλaSH can generate a testbench along with the hardware, which contains some input values. ModelSim is the program within Quartus II that does this simulation. The results are shown as an overview of all signals, i.e. all bits of the digest. The MD5 program was fitted with an input with known digest and this input was one block size width, so that the function would complete in one round. The result of the testbench matched the expected result for the hash function.
7. CONCLUSIONS

Cryptographic hash functions are translated perfectly to hardware by CλaSH. That is, the functions themselves. The pre-processing part of the input could not have been tested due to time constraints.

We cannot predict anything about the pre-processing hardware part either, as this significantly differs from the rest of the functions.

The functions themselves are very complex, as they are intended to be, but at a low level they are very much straightforward. Basic logical operations are performed on bits and the functions consist of a long and wide sequence of these operations.

CλaSH generates just these basic operations and connects them in the right sequence. No redundant hardware is generated. So the CλaSH generated hardware is in essence the same as existing hardware implementations.

A shortcoming was found when we tried to use hardware components for multiple functions. To make this clear: it is not possible to create hardware with CλaSH that has two different paths for the input of a component and splits the output of the component into two different paths again. CλaSH generates a new component for each signal path.

For cryptographic hash functions this turned out to be not much of a shortcoming. Components are very basic and the profits of adding a switch for every sharing of a component would not weigh up to the cost of it. One could however find situations in which the profits would weigh up to the cost.

A feature that might be useful is loops. A round shift operation is easily defined in a loop. Currently the programmer has to define each separate vector component for this kind of problems. Loops should however be unfolded by the compiler and not be translated to a multiple clock-cycle solution. The latter can always be accomplished manually. An argument against loops will be that loops are not encouraged in Haskell either.

Recursion is a feature that is under development, but we feel the urge to mention it here also. Programming in a functional language involves recursion. CλaSH should provide recursion too.

Clock and reset signals are always added to components. These signals are not always needed. When hardware with no clock cycles is created, these signals are redundant.

Then there is the part about user friendliness and documentation. CλaSH is the product of a small research group and this can be noticed by new users. CλaSH is based on Haskell, but actually programming in CλaSH is really different from programming in Haskell. Imported modules are not supported and the vectors CλaSH uses are not the vectors Haskell provides.

When these factors are taken into account, CλaSH works. But these points should be made clear to new users through documentation and this is not the case. Also the reference manual is incomplete, some useful functions are missing.

8. FUTURE WORK

Research questions are answered, but the research was not complete. The pre-processing parts of the cryptographic hash functions should still be implemented in order to get working hardware implementations. Other conclusions might be found in the process.

9. REFERENCES