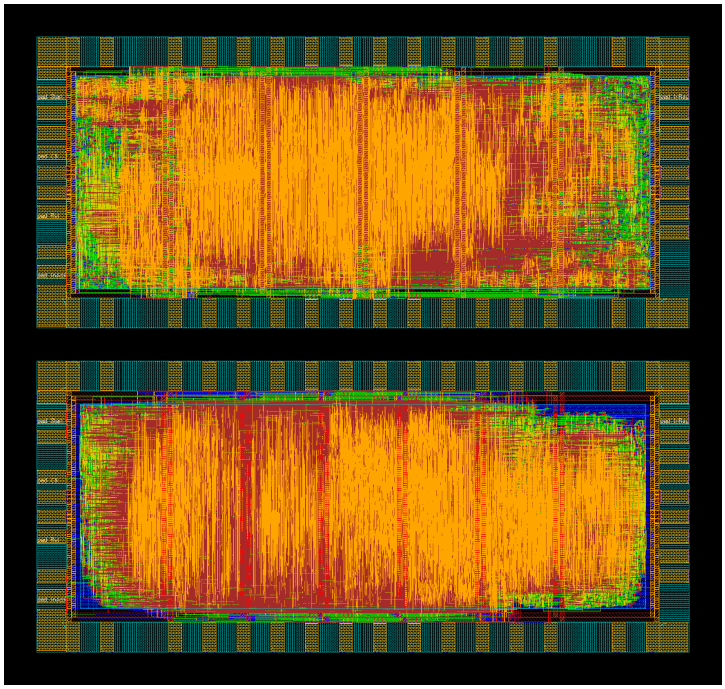
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Silicon Implementation of Second-Round SHA-3 Candidates

Patrice Guillet Enrico Pargaetzi Martin Zoller

February 2010



Final layout of our two ASICs

Advisors:	Luca Henzen, henzen@iis.ee.ethz.ch
	Frank K. Gürkaynak, kgf@ee.ethz.ch
Supervisors:	Hubert Kaeslin, kaeslin@iis.ee.ethz.ch
	Norbert Felber, felber@iis.ee.ethz.ch
Professor:	Wolfgang Fichtner, fichtner@iis.ee.ethz.ch

Handout:	September 2009
Due:	February 2010

Contents

1 Introduction

1.1 What is a hash function?

A hash function transforms a message m of arbitrary length to a hash value of fixed length (e.g. 256 bits). Hash functions are widely used in information security applications, e.g. to detect transmission errors when sending large files over a network, or to make sure that files are authentic and have not been tampered with on their way to the receiver. In order to achieve this, a hash function must be resistant against three kinds of attacks:

- **Preimage Attack:** Given a hash value, find a message for which the hash function will return this value.
- **Second Preimage Attack:** Given a message and its hash value, find a different message which has the same hash value.
- **Collision Attack:** Find any two messages which are mapped to the same hash value.

Examples for the attacks are given in fig. 1.

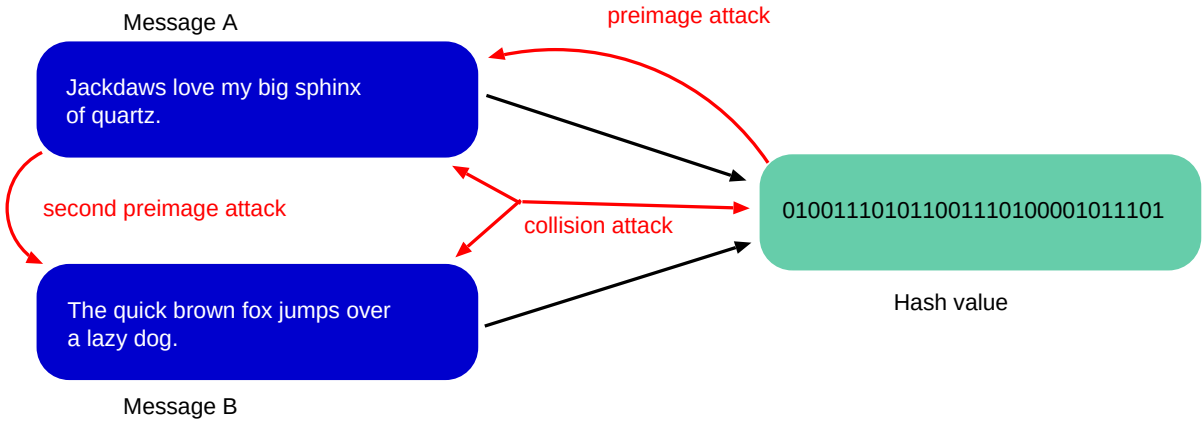


Figure 1: Example of a hash function and possible attacks to it.

1.2 SHA Hash Functions

The Secure Hash Algorithm (SHA) series of hash functions consists of several algorithms which have been selected and standardized by NIST. Currently the series consists of **SHA-0**, **SHA-1** and the **SHA-2** family. The SHA-0 standard was withdrawn shortly after publication. SHA-1 is considered broken, but still in use for some applications. The SHA-2 family of functions suffer from similar flaws, but are still computationally secure due to longer hash values. A competition to elect a successor algorithm (SHA-3) is currently in progress.

1.3 The SHA-3 Competition

In November 2007, NIST started a competition to find a successor to the SHA-2 family of hash functions which will be called **SHA-3**. Submissions were accepted until October 31, 2008. There were 51

submissions fulfilling the basic requirements. In July 2009, the first round of the competition was completed and 14 second-round candidates were announced. A conference on these remaining algorithms is scheduled for August 2010.

1.4 VLSI Implementations of Candidate Algorithms

It is hard to compare hardware implementations which are based on different EDA tools, standard cell libraries, or ASIC manufacturing processes. We only know of one benchmark where all 14 second-round SHA-3 candidates were compared using a common toolchain: The work by S. Tillich et al. [3]. In this benchmark, the gate-level netlists of all algorithms were compared.
We're going one step further by comparing the performance of six algorithms on ASICs ready for manufacturing. This way, the additional area and delays introduced by placement and routing - which can vary greatly between algorithms - are also considered. The algorithms analysed in this thesis are Blue Midnight Wish, Fugue, Groestl, JH, SHAvite-3, and Skein. Pietro Gendotti analysed six more algorithms in his Master thesis [9]: BLAKE, CubeHash, Hamsi, Keccak, Luffa, and Shabal. Unfortunately the two benchmarks are not comparable since we designed two ASICs for the UMC 180nm process, while Pietro made a single 90nm chip. This is due to the fact that an important goal of our semester thesis was to get an ASIC manufactured, but there was no suitable tape-out date for 90 nm. However, we may get the chance to create a 90 nm chip with our algorithms at a later date.

2 Toolchain

2.1 Software

The Microelectronics Design Center of ETH (DZ) provided us with a complete kit of Electronic Design Automation (EDA) tools. A small tool called "Cockpit" allowed us to start all the necessary programs in a convenient way. The entire toolchain runs on Linux.

VHDL Coding and Schematic Diagrams We wrote all our code in *GNU Emacs*, using its VHDL mode. It includes syntax highlighting, auto-completion, automatic indentation and many more features which make writing VHDL code a pleasurable experience. For creating schematic diagrams we used *tgif* and a symbol library provided by DZ. Both programs are free software.

Behavioral and Gate-level simulation To compile and simulate our circuits, we used *ModelSim* by Mentor Graphics. The behavioral simulation is a very important and time-consuming step in the design flow since it often shows bugs and design flaws of the VHDL code which don't trigger any compile errors.

Synthesis The synthesis of VHDL code to gate-level netlists, including optimization of timing and area, clock tree generation and scan chain insertion, was done with *Synopsys DC*. We used scripts to automatically do synthesis runs for different clock periods. The results were then used to determine the optimum clock period for each hash algorithm. To obtain a netlist for an entire chip, we synthesized its components (algorithms and interface), froze their netlists and included them in a final synthesis run.

Test Coverage We used *Tetramax* by Synopsys to determine the fault coverage of our designs and to generate test patterns.

Placement and Routing We did the floorplanning, placement and routing of our ASICs with *SoC Encounter* by Cadence Design Systems. The same software was used to analyse power dissipation and IR drop of the final chips.

DRC and LVS A final Design Rule Check and Layout vs. Schematic Analysis was done using *Calibre* by Mentor Graphics.

2.2 IC Manufacturing Process

Our ASICs were manufactured using the UMCL180 1P6M technology. They were processed by *Europractice IC Service* as double mini@asics, which are 3.24×1.525 mm large and thus have an area of 4.94 mm^2.

3 Interface

To handle the different hash functions on one chip and the input and output bits there is need for an interface. Because of limited pin availability we had to limit ourselves to 8-bit serial input and output. To store the 512-bit input blocks and 256-bit outputs we used flipflop registers. The interface was developed by Pietro Gendotti [9] and adapted for our chips. It also contains a Finite State Machine (FSM) that controls the I/O operations and determines which algorithm is running, etc. Furthermore, there is a clock gating module, so only one algorithm at a time is clocked. This permits effective power dissipation measurement during testing. For the same reason I/O operations during the runtime of an algorithm are not allowed. There also exists a Blow-Up mode to fill the buffers with a predefined pattern. The file *interfacepkg.vhd* contains most of the parameters of the single algorithms, so the whole framework is easy to adapt for a new algorithm, by just plugging in the specific numbers in the package file. The chip entity also includes a reset synchronizer to avoid reset hazard issues. See fig. 2 for a schematic diagram of the interface.

Adapting existing code is not always easy; a good understanding of the code is needed first. Luckily Pietro's code was pretty well commented, which simplified the tasked massively. Nevertheless, without additional explications by our advisor Luca, it would have been nearly impossible to get a proper overview and adapt the code to our needs.

4 Algorithms

4.1 Blue Midnight Wish

Blue Midnight Wish (BMW) is the SHA-3 proposition of the Norwegian University of Science and Technology in Trondheim. The implemented version of BMW takes blocks of 512 bits as input and generates a message digest of 256 bits. The tweaked submission of BMW was used.

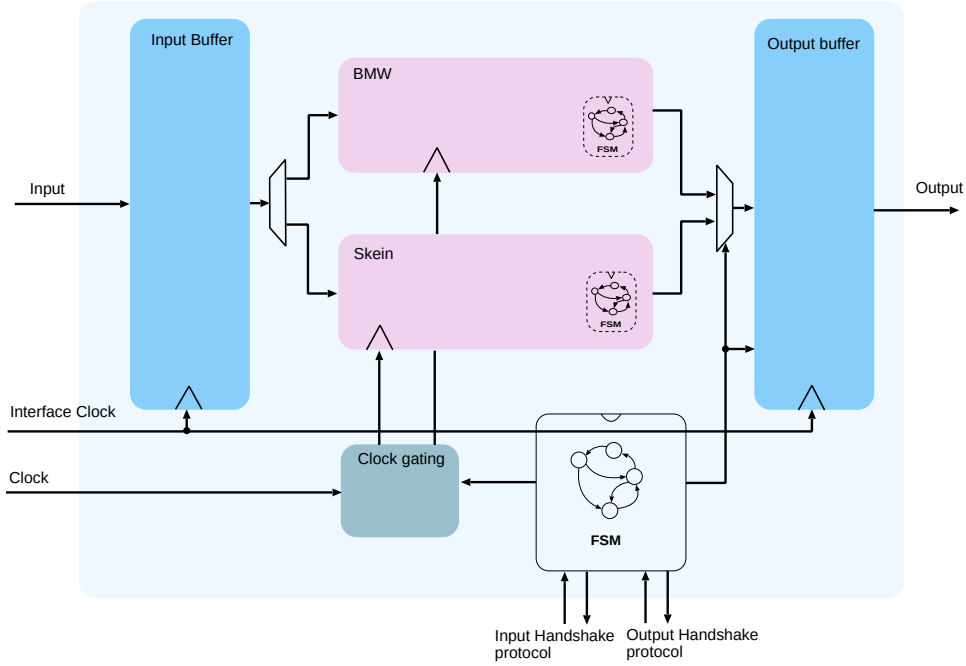


Figure 2: Schematic diagram of the interface.

Algorithm Description The state of BMW is stored in the value H, called *double pipe*. H is set to an initial value at the start of the algorithm. The input block M is then fed, toghether with H, to a function f_0 which outputs the first half Q_a of the *quadruple pipe* Q. A second function, called here $expand$, takes as inputs M, H and Q_a in order to generate the second half Q_b of Q. The third and last function f_3 folds M, Q_a and Q_b into the new double pipe H. With this new value the next round starts. After the last block of the input message has been processed a finalization round starts: the double pipe value is used as the message block and is replaced by a finalisation constant. The 256 least-significant bits of H at the end of this round are the message digest. A round of BMW is illustrated in Figure 3 while the width of the signals is resumed in Table 1.

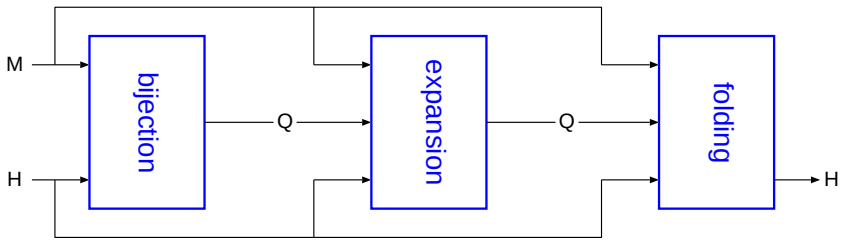


Figure 3: A round of BMW

M	H	Q_a	Q_b	message digest	word width
512	512	512	512	256	32

Table 1: Widths of the values used in BMW-256 (in bits)

The 3 functions are described shortly below.

- f_0: This function is split into two subfunctions. Both of them are bijective transformations. The first one is made up of 16 expressions (one for each word, $i \in [0, 15]$) of this form[1]:

$$W_i = \sum_{k=1}^{5} (-1)^{\alpha} \left(M_{\beta_{k,i}} \oplus H_{\beta_{k,i}} \right)$$

  The second one returns the first half of the quadruple pipe:

$$Q_i = s_{i \bmod 5}(W_i) + H_{i+1 \bmod 15}$$

  The $s_{0...5}$ functions do one or more logical shifts of the argument and return the result by xoring them.

- $expand$: The $expand$ function needs 16 rounds in order to compute the second part Q_b of the quadruple pipe Q. In each round 17 terms are added to a new word. The last term is again an addition of 4 words of the input message with a constant, the whole xored with a word of the double pipe. The remaining 16 terms are already computed words of Q after a bit-shift or a bit-rotation.

- f_2: The last function needs 2 intermediate values: XL and XH are obtained by xoring chosen words of the double pipe. The new value of H is then computed word by word adding and xoring values from M, H, Q, XL and XH after they have been bit-shifted or bit-rotated.

Golden Model A Golden Model was coded in Matlab and tested against the KAT values provided with the submission package. The debugging of the golden model with the known intermediate values helped pointing out some little inaccurancies of the description. The length of the vectors was randomized within a settable interval. The model performed surprisingly well: generating 6000 vectors with an average length of 5 blocks took about 30 minutes.

HDL Implementation The three functions are kept distinct in the HDL-code too. The implementation of f_0 was pretty straightforward: the description found in the documentation was simply translated into VHDL. The $expand$ function is a little more involved and can be implemented in two ways. One can implement the 16 steps in a cascaded way such that the input of the previous slices feeds the current one. At the end of the computation all words are available. This leads to a very long propagation delay (see [3] for such an implementation). The other way to do this is to let the hardware compute just one round at a time and to store it in a register. The output of the register is then fed back to the function. Although the hardware needed does not shrink significantly (the same slices must be implemented), this alternative was chosen. It's higher operating frequency allows us to put the algorithm on a chip toghether with other functions without the need for a separate slower clock. f_2 was, as f_0, directly implemented from the algorithm specifications. In Figure 4 the structure of the chosen implementation is showed.

The dataflow is controlled by a 37-states Moore-FSM. In order to store H, M, Q_a and Q_b four registers with 512 bits each are needed. As stated before the $expand$ function takes 16 computation cycles to complete. The other two parts of the algorithm are single-cycle operations. The latency of the algorithm, hencem is 18 cycles. The control signal `ExpControlxS` selects the slice to be computed and the word where the result has to be stored into.

[1] for the values of the coefficients please refer to [13]

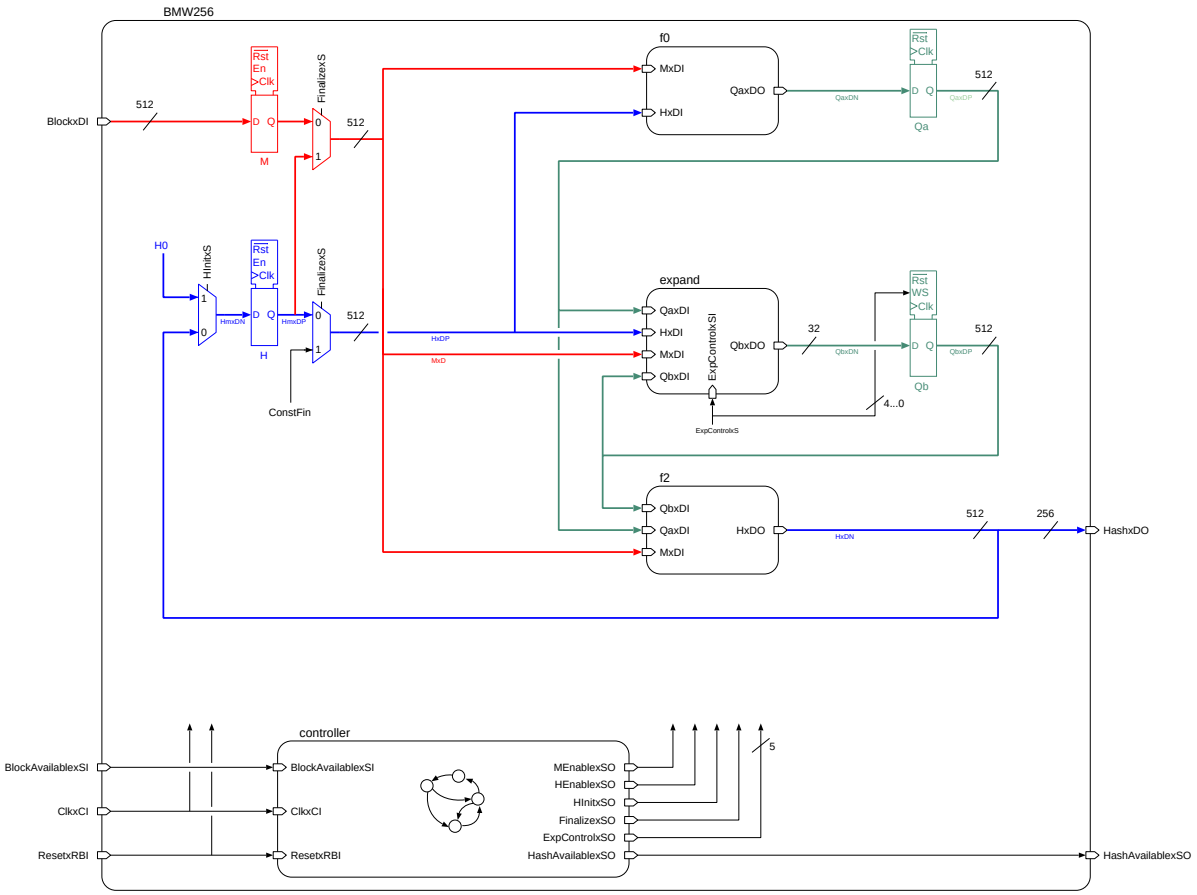


Figure 4: The block diagram of the chosen implementation for BMW-256

Synthesis Take a look at the specification of the algorithm: in the three functions one can find 126 32-bit-additions. We expected the synthesized circuit to be huge. A run with very lax timing constraints (10 MHz clock) gave an estimated area of 71 kGE. This value is almost constant up to 55 MHz. Here, the throughput of the algorithm is 1.6 Gbit/s. At its maximum synthesizable clock speed (185 MHz) the area is given as 117 kGE and the throughput is 5.3 Gbit/s. The output of Synopsys confirmed the expected number of flip-flops. No latches were instantiated.

Test Coverage With the circuit tested against the golden model and synthesized we were ready for the generation of the test vectors. TetraMax gave a test coverage of 99.18%.

4.2 Skein

Skein is the submission of Niels Ferguson, Bruce Schneier et al. The algorithm can be used to do tree-hashing, as PRNG and as MAC, to name just a few possibilities. We focused on hashing only. The implemented version (Skein-256-256) takes 256 bits as input blocks and outputs a digest of 256 bits.

Algorithm Description Skein-256-256 is based on the Threefish-256 function, a tweakable block cipher whose inputs and outputs are four words of 64 bits. 4 out of the 72 rounds needed to compute Threefish-256 are shown in Figure 5.

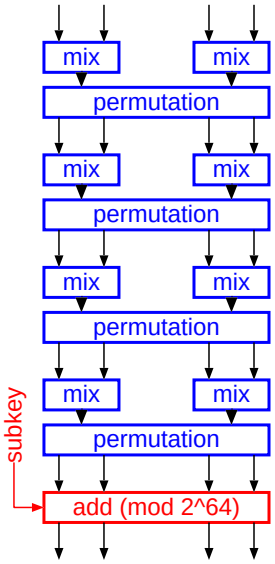


Figure 5: 4 rounds of the Threefish-256 cipher

The mix-boxes carry out the following calculations:

$$M_{out\,0} = (M_{in\,0} + M_{in\,1}) \bmod 2^{64}$$
$$M_{out\,1} = (M_{in\,1} \lll \gamma) \oplus M_{in\,0}$$

The permutation is defined as $M_{0,1,2,3} \leftarrow M_{0,3,2,1}$. After 4 rounds a *subkey* is added. The computation of the subkey is done by xoring chosen words of a shift register which has been loaded with the main key and a *tweak*. For details about this computation please refer to [18].

The developers of Skein call the operating mode of their algorithm *UBI* (Unique Block Iteration). In figure 6 the use of this operation mode for hashing is shown.

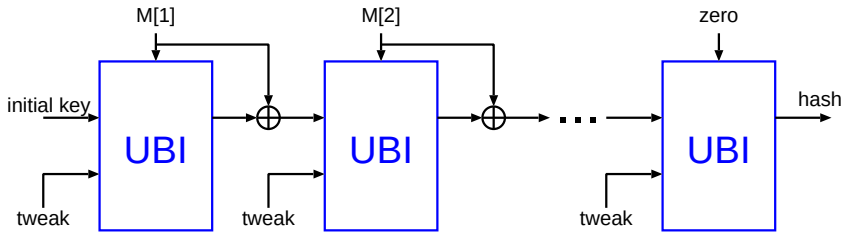


Figure 6: The UBI structure of Skein used for hashing

In each UBI block there is a Threefish-256 cipher feeded by a message block, the tweak (containing configuration information and a counter which keeps track of the number of bits already processed) and the key. Notice that the key of an UBI (except for the first one) is given by the output of the previous stage. The rightmost block is the finalization block.

Golden Model Since Skein makes use of 64 bits for each word the golden model was written in C using the datatype *uint64_t*. Its main advantages are the optimisation for the new 64 bits computer processors and the well defined overflow behavior. Because of the variety of possible uses, the documentation of Skein is very extensive and tangled... like a skein. This, the somehow poorly chosen intermediate values and the use of proprietary libraries in the reference implementation made the debugging of the golden model a hard job. Hard but necessary in order to write the subsequent HDL code.

HDL Implementation The rotation coefficients of Threefish-256 are periodic: $\gamma_i = \gamma_{i+8}$. The natural choice for the implementation of such a structure is an iterative decomposition. Again we had two possibilities: 4 or 8 rounds per computational cycle. The execution of 4 rounds calls for a subkey while 8 rounds need two subkeys. On the other hand, with 4 rounds one needs a control circuitry to switch between the first and the second set of coefficients. We took the control overhead into account and decided for the shorter propagation delay of the first alternative. An implementation with 8 rounds per cycle can be found in [10].

The block diagram of the chosen implementation is depicted in figure 7. The entity *permix* performs 4 rounds of the Threefish-256 function. The `PerselectxS` bit selects the coefficients for the rotations. The *controller* is a Mealy-FSM of 58 states and the latency of the algorithm is 19 cycles. The output of the counter `BnxS` keeps track of the number of bytes already encoded. This value is part of the tweak, together with two flags which indicate if the input block being processed is the first and/or the last one. The latter information has to be provided to the algorithm (an internal way of detecting the last block is feasible but would introduce a large amount of latency).

Synthesis The area of the synthesized circuit with a clock of 10 MHz was 20 kGE. At 40 MHz the area needed is still quite modest with 22 kGE. The throughput is 1.1 Gbit/s. The maximum achieved clock speed is 142 MHz, with an area of 42 kGE and a throughput of 4.1 Gbit/s. Again, the number of instantiated flip-flops was the one we expected and there were no latches.

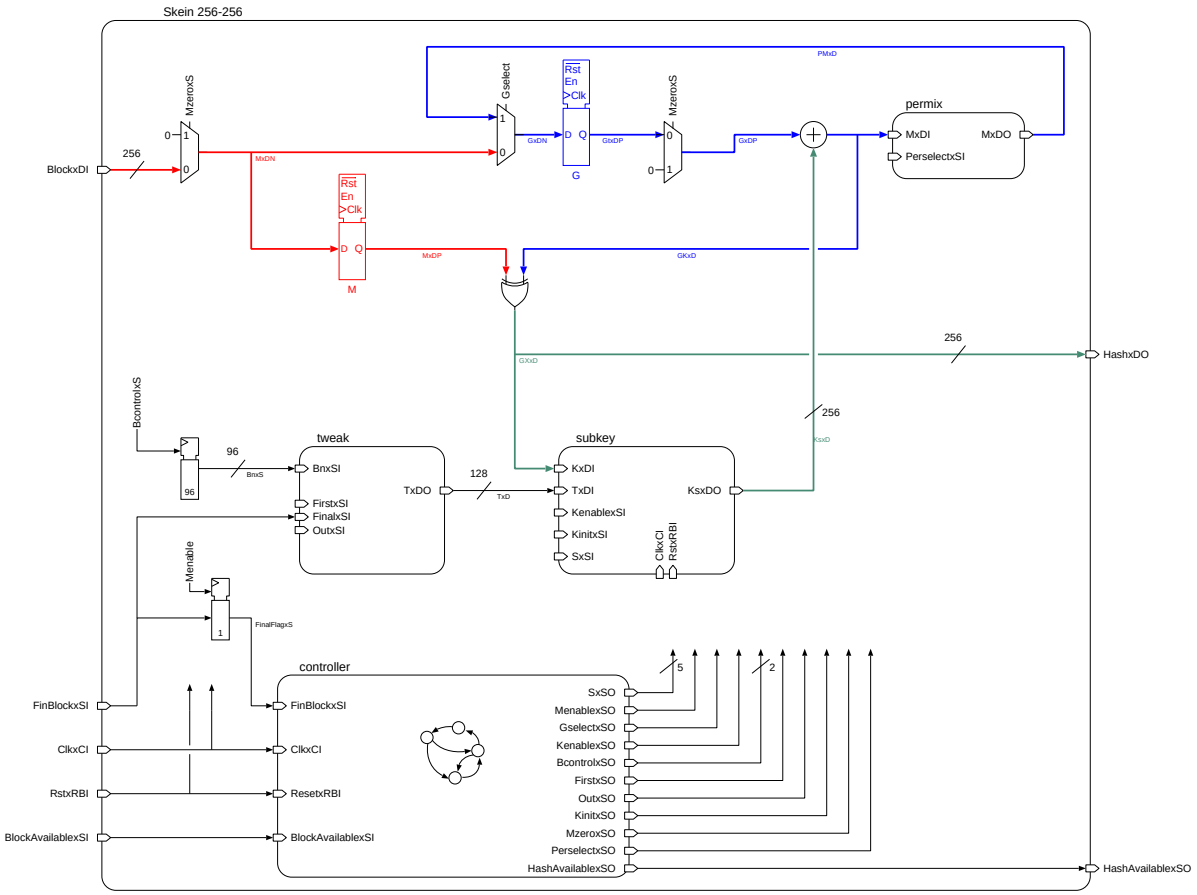


Figure 7: The block diagram of the chosen implementation for Skein-256-256

Test Coverage The obtained netlist was explored using TetraMax, wich reported a reassuring test coverage of 99.98%.

4.3 Fugue

Algorithm Description The Fugue algorithm was developed at IBM's Watson research center in Hawthorne, New York. It works without a compression function - the hash function is directly able to process inputs of variable length. The round function of Fugue is somewhat similar to that of AES. It uses the same SubBytes transformation, but the MixColumns transformation used in AES is replaced by a more complex operation called SMIX. The security of Fugue against current attack methods has been proven.

The algorithm works with input blocks of 32 bits, which is also its word with. It has a state matrix of 30 words which is initialized with a constant value. For each input block, the round function is applied to the state matrix. After processing the message, a final round of calculations follows. The hash value is a subset of the final state with a length up to 512 bits. For our chip we considered only the 256-bit version.

Golden Model The Golden Model for Fugue was written in Java. It uses a class with functions for matrix manipulations in $GF(2^8)$ which we wrote for this model. Initially the S-Box was implemented using *log* and *exp* functions in $GF(2^8)$, as proposed in [2]. In the end a simpler and less error-prone approach using bitshifts was used.

The algorithm specification for Fugue [14] is very good as all its basic assumptions are clearly stated. Furthermore, the developers provide a file with intermediate results, which simplified debugging of the model. Still, it took us about a week to find all the errors. It was important to generate useful debugging output and compare it to expected intermediate results in order to locate problems.

HDL Implementation The Fugue algorithm was implemented in VHDL using two computation cycles per input block, as in [3]. This is a good compromise between hardware efficiency and speed since most of the combinational circuitry is used in both cycles. As shown in fig. 8, the final round requires 37 cycles, which was necessary to avoid the need for more than one instance of the SMIX circuitry.

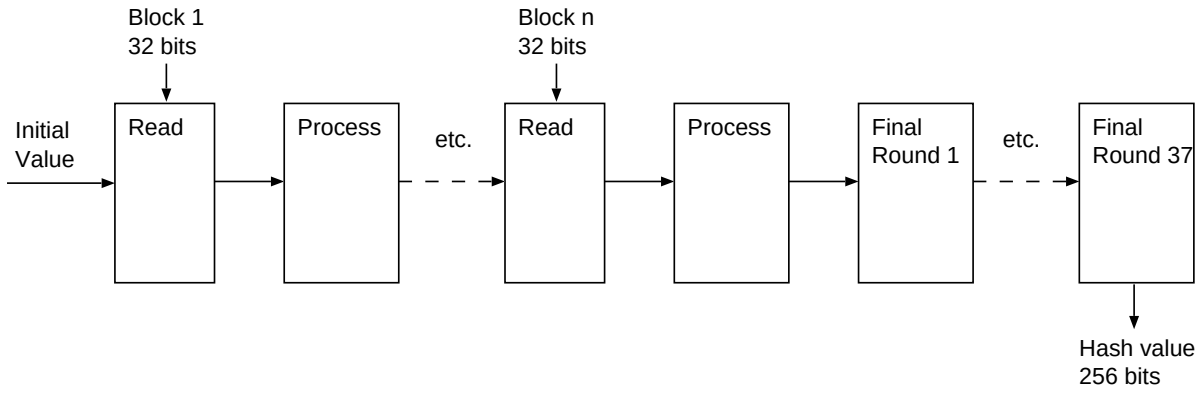


Figure 8: Process diagram of the Fugue algorithm.

A schematic diagram of the hardware implementation is given in fig. 9. The SubBytes operation was first implemented using look-up tables, but it was later replaced by a combinational implementation. With a minor loss in throughput, this allowed us to reduce the circuit area. The SMIX column permutation can be described as a matrix multiplication in the $GF(2^8)$ Galois Field: The 16 output bytes of the S-Box are written as a column vector and multiplied with a 16×16 matrix. This operation was realized as a hard-coded combinational circuit. The multiplications are done by adding up bit-shifted versions of the 8-bit signals; each row of the resulting vector is then obtained by adding some of these signals. Note that additions are simple XOR operations in $GF(2^8)$. The Fugue algorithm also requires XOR operations between entire columns of the state matrix. These are called CXOR and their implementation as combinational circuits is straightforward. Furthermore, there are transformations called ROR3, ROR14, and ROR15, which rotate the state matrix to the right by 3, 14 and 15 columns respectively. They are implemented as simple bit shifts.

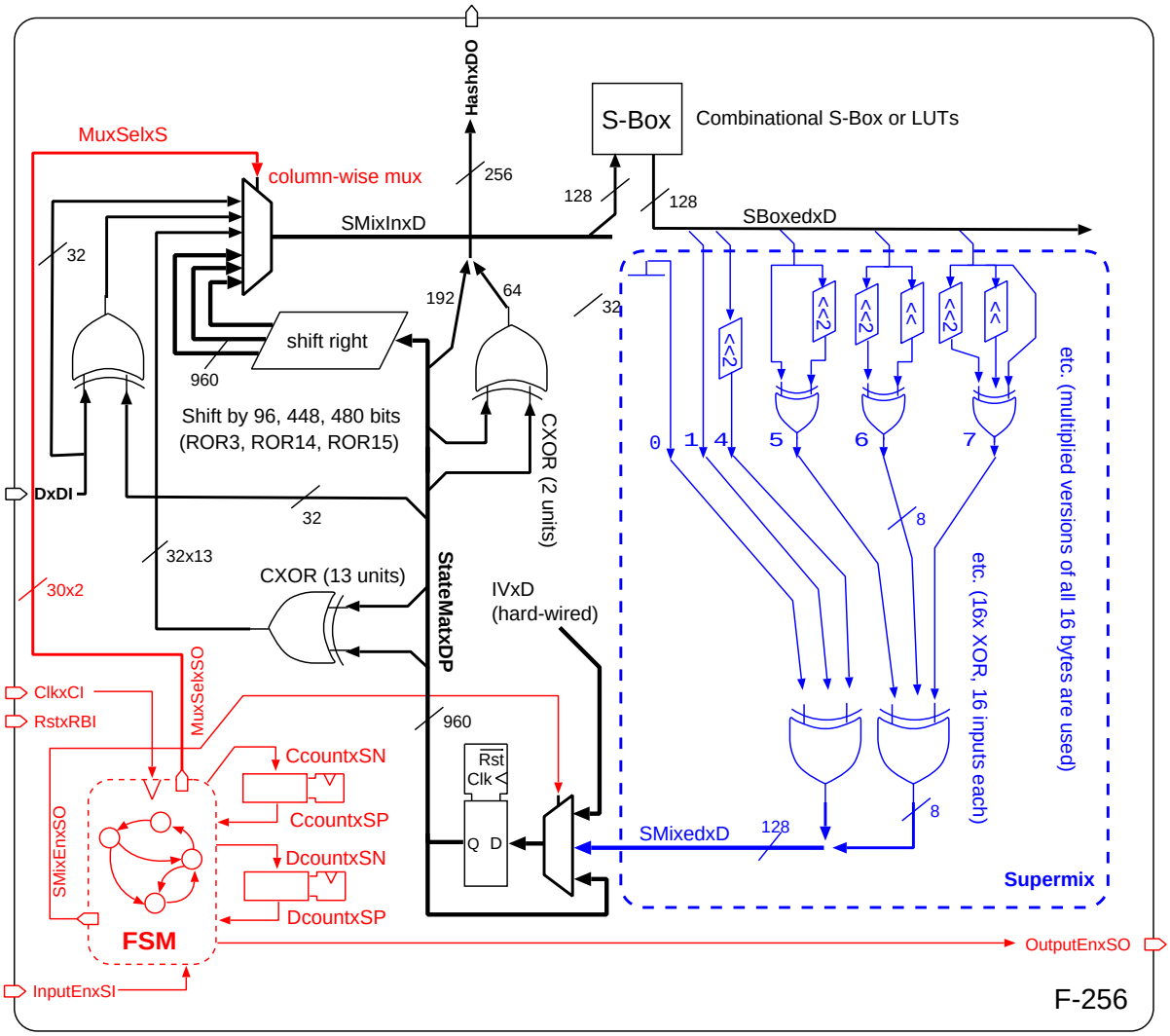


Figure 9: Schematic diagram of the Fugue algorithm as implemented in hardware.

An FSM controls the datapath of the algorithm. Its state diagram is given in fig. 10. Two counter registers are used.

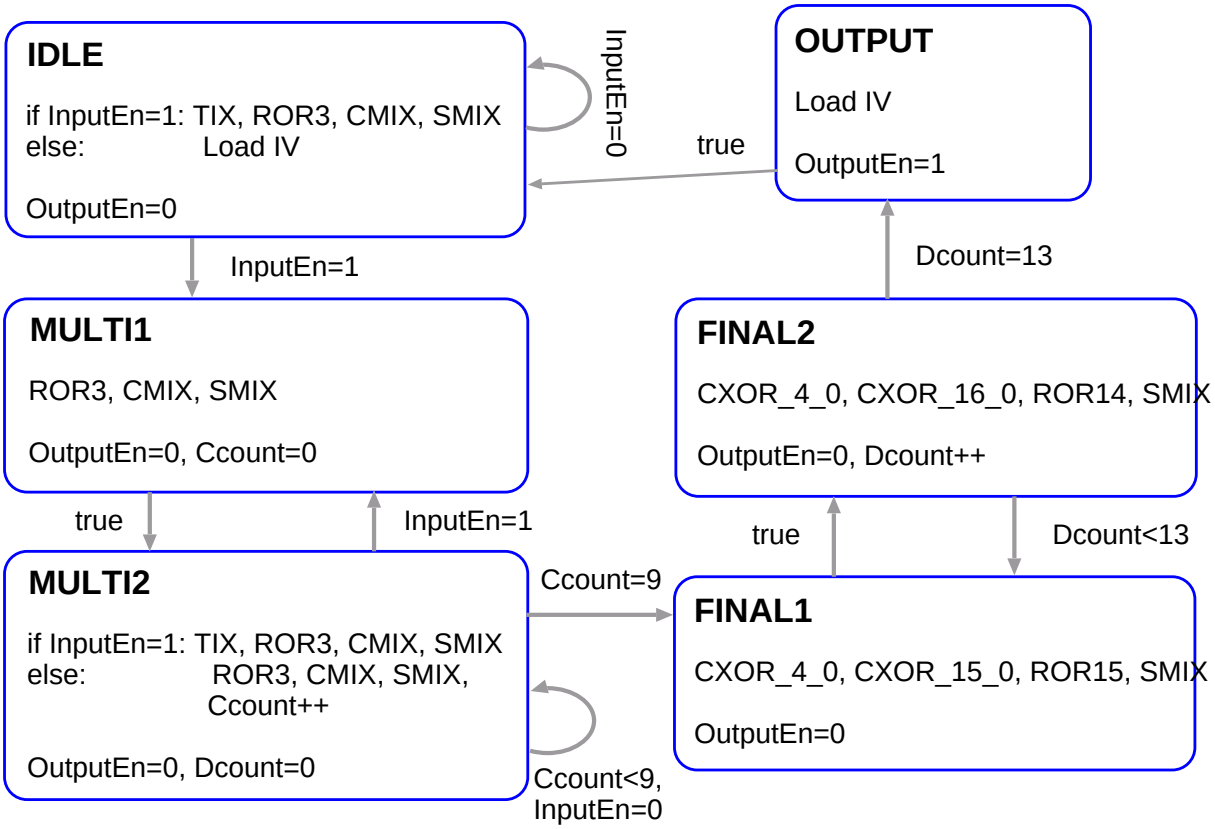


Figure 10: FSM state diagram of our implementation of the Fugue algorithm.

Synthesis There were no problems during synthesis and we quickly obtained gate-level netlists for different clock periods. As opposed to the other algorithms we implemented, Fugue's required area doesn't increase very much when maximizing the clock frequency: The minimum area implementation uses 24.8 kGE and can be clocked up to 100 MHz, while the maximum speed implementation has a size of 38.6 kGE and achieves up to 345 MHz.

Test Coverage After adding a scan chain, analysis in Tetramax returned a fault coverage of 98.7%.

4.4 Groestl

Algorithm Description The Groestl Algorithm is based on the AES algorithm. Its basic structure is shown in fig. 11. The version we implemented takes inputs of 512 bit block length and outputs a 256 bit hash value. For the first block a constant Initialization Vector is used, for further blocks the state is used for processing.

The Groestl compression function is based on two similar round functions P and Q. They consist of the 4 parts "AddRoundConstant", "SubBytes", "ShiftBytes" and "MixBytes". The state matrix of Groestl is a 8×8 matrix with entries of 1 byte.

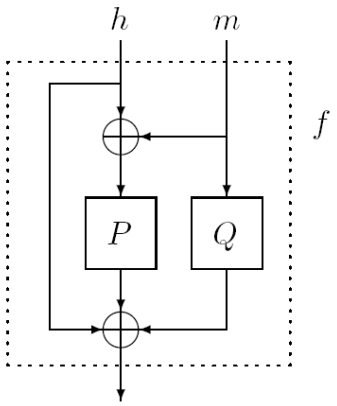


Figure 11: Structure of the Groestl compression function

The only difference between P and Q is in the AddRoundConstant part where the round constant is added at a different place and XORed with "FF".

SubBytes substitutes the matrix entries. It uses the Rijndael S-Box as AES does.
ShiftBytes is circular shift operation operated on the rows of the state matrix.
MixBytes is a complicated matrix multiplication with a circular matrix. It is carried out on a finite field defined by the Rijndael polynomial $x^8 + x^4 + x^3 + x + 1$.
We used Groestl with the recommended 10 rounds per block and 10 additional rounds output transformation.

Golden Model We began our exploration of the algorithm by programming a golden model in MATLAB. For the Groestl algorithm we used the Fixed Point Toolbox. Understanding the algorithm and programming in MATLAB went hand in hand.

The hardest part was the MixBytes operation, which uses multiplication over a finite galois field. We implemented this multiplication first by using the algorithm described in [2].

The verification of the golden model was done by using the intermediate values files provided by the Groestl Team. The availability of intermediate values massively simplified verification of partial code.

The golden model is able to generate random messages of different length which were used to verify the VHDL code. This golden model is quite slow; generating 20 test vectors took about one hour.

Block Diagram In order to make a structured approach to the VHDL coding, a block diagram was developed first. The diagram was drawn in TGIF and was continuously updated as the VHDL code was

refined at later stages. 3 registers with 512 flipflops each were required.

VHDL The implementation of the VHDL code was pretty straightforward from the golden model. A lot of the code was rewritten avoiding processes to detect a zero-latency loop in the code. The SubBytes and the ShiftBytes operations were combined in a single statement for simplification. The most complicated part is again the MixBytes step. As the matrix we multiplicate with is circular and consists of numbers from 2 to 7, we have hardcoded the galois multiplication, what lead to somewhat unreadable code, prone to errors. Still this promised to be the most efficient solution. Debugging was possible because of the mix of the columns and the availability of intermediate values, so you could estimate where the error should be. A Finite State Machine (FSM) was written to control the different stages of the algorithm. Simulation with Modelsim and the randomly generated vectors by the golden model was used to debug the VHDL code and verify its correctness. The testbench was provided by Luca Henzen and adapted for our algorithm.

Implementations The first implementation had two separate round functions P and Q. We reached a very high throughput, but needed a lot of area. The second implementation featured one P/Q round split after the combined SubShiftBytes function where we inserted a pipeline register. The advantage of this architecture is that we can calculate P and Q alternatively at no additional cost, because we would have to store the values calculated by either P or Q anyway. This pipeline register allows us to cut the longest path almost in half. The latency is only doubled and not quadrupled because of the alternating calculation of the rounds. This leads to a similar performance as the parallel implementation with about half the area. See fig. 12 for a schematic diagram. Both implementations used a Look-Up Table (LUT) for the Rijndael S-Box. To further cut down on size we used the Wolkenstorfer Implementation [12] of a combinational S-Box. Although we lost a bit on performance, the gain in area was definitely worth it. A comparison of the three implementations is given in fig. 13.

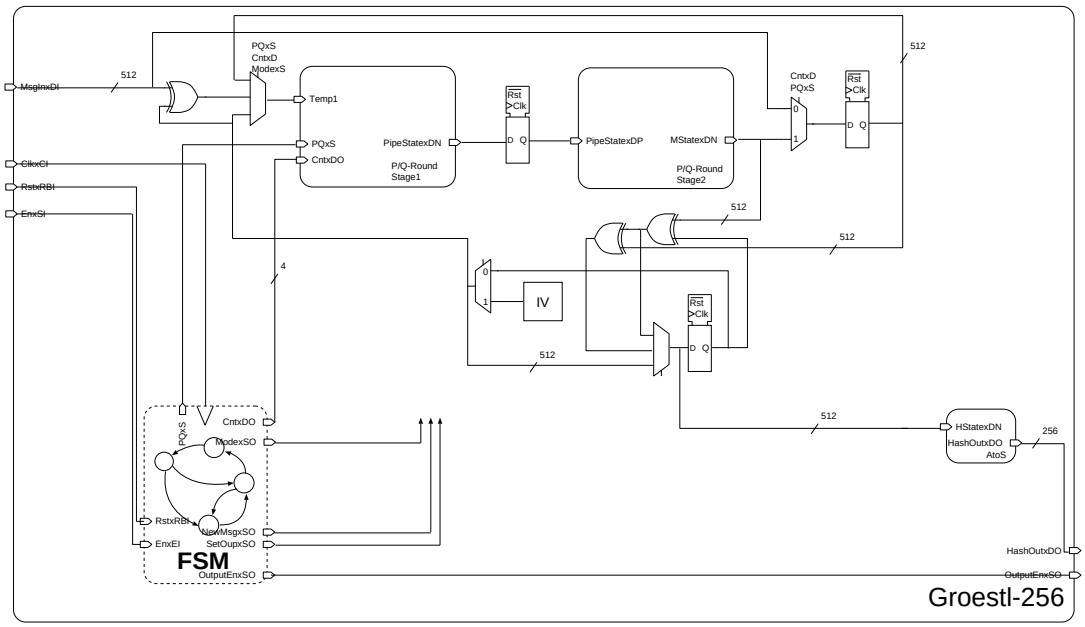


Figure 12: Block diagram of the small Groestl implementation

Synthesis The synthesis detected a zero-latency loop in the code, which lead to a considerable amount of code rewriting and debugging to find the mistake. Synthesis was first carried out without any timing constraint targeting minimal area. The timing constraints were then increased and refined when the synthesizer reached the feasibility limit. The highest throughput reached with Synopsis for Groestl was 10.66 Gbit/sec with the parallel implementation at a clock frequency of 208 MHz and a latency of 10 clock cycles. The respective size was 165 kGE. The small implementation reached a throughput of 9.38 Gbit/sec at a clock frequency of 384.6 MHz and a latency of 21 clock cycles. It occupied 131 kGE of area. The combinational implementation, which we used in the end, reached a maximum throughput of 7.32 Gbit/sec at a clock frequency of 300 MHz and latency of 21 clock cycles, using 72 kGE.

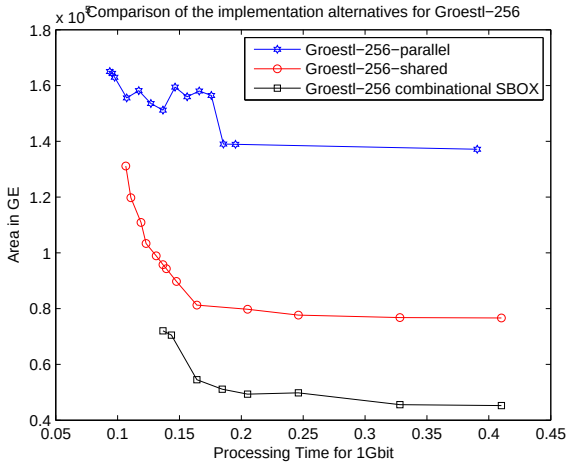


Figure 13: Area vs. Throughput comparison of the three Groestl implementations

Test Coverage Test coverage was explored with TetraMax and showed 99.58% coverage.

4.5 JH

Algorithm Description The JH algorithm combines some features from AES and Serpent. Its basic structure is depicted in fig. 14. The version we implemented takes a block input of 512 bits and gives out a hash value of 256 bits (JH-256). The internal state consists of 1024 bit ordered in a 16x16 matrix with 4-bit entries. A constant initialization vector (IV) is used for the first block of a message, for further blocks the internal state vector is used. The incoming message is XORed with the first half of the IV or state vector respectively, before entering the round function. After this, the obtained state vector is grouped into a 16x16 4-bit state matrix. JH executes 36 rounds per message block. The rounds consist of a 4 bit S-Box substitution, where one of two S-Boxes is chosen based on a constant vector, a linear operation and some permutations. There are 36 different 256-bit constant vectors, one for each round. Each bit in this vector represents which S-Box is used for the respective 4-bit entry in the state matrix. The state matrix has to be degrouped again to regain a state vector. The second half of this vector is then XORed with the original incoming message. The 256 last bits of the last state vector represent the hash value.

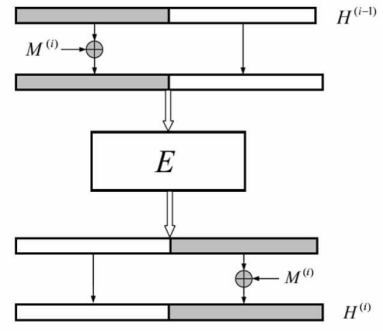


Figure 14: Structure of the JH compression function

Golden Model To understand the algorithm and get test vectors, we programmed a golden model in MATLAB. Unfortunately intermediate values were not provided by the developer. We had to obtain intermediate values by manually executing the reference implementation in C on the Known Answer Test (KAT) files. The conversion of the different formats, e.g. from a bitstring to 4 bit elements and back, caused some trouble in MATLAB. The constant vectors can be generated with a reduced dimension function of the round. Only one S-Box is used there. The golden model was then used to obtain test vectors.

Block Diagram Before starting with the VHDL code, a block diagram was again required. Because the message needs to be stored for the final XOR operation, a total of 1536 flipflops is required. The block diagram of the final implementation of JH is given in fig. 15.

VHDL The VHDL was written in the same style as for the Groestl algorithm, avoiding processes. The implementation of the round function was rather easy. The substitution was simliar to the one used in Groestl. The linear operation and the permutations were straightforward to implement. As we used only one message digest size (256 bit) we could store the initialization vector in a LUT instead of calculating it with a specific message in the beginning. Again we managed to include a zero-latency-loop, but with the experience gained already, this problem was resolved quickly. The golden model was used for the debugging, generating intermediate values where necessary. A simple Finite State Machine was written as control element. The testbench from the Groestl algorithm was adapted and used for debugging and functional verification of the code in Modelsim. Random test vectors with corresponding answers could be obtained from the MATLAB golden model.

Implementations The first implementation used Look-Up Tables for the two S-Boxes and calculated the constants for every round in parallel. A second implementation stored the 36 constants of size 256 bit each in a LUT as well which led to a small gain in terms of area. The third implementation used combinational S-Boxes, which led to worse results and was therefore not pursued. So the final implementation used on the chip stores the constants instead of calculating them. The latency for all the implementations is 36 clock cycles.

Synthesis The synthesizer detected a zero-latency loop, which was quickly detected and removed. Synthesis was done incrementally, targeting minimum area first, then increasing the clock frequency

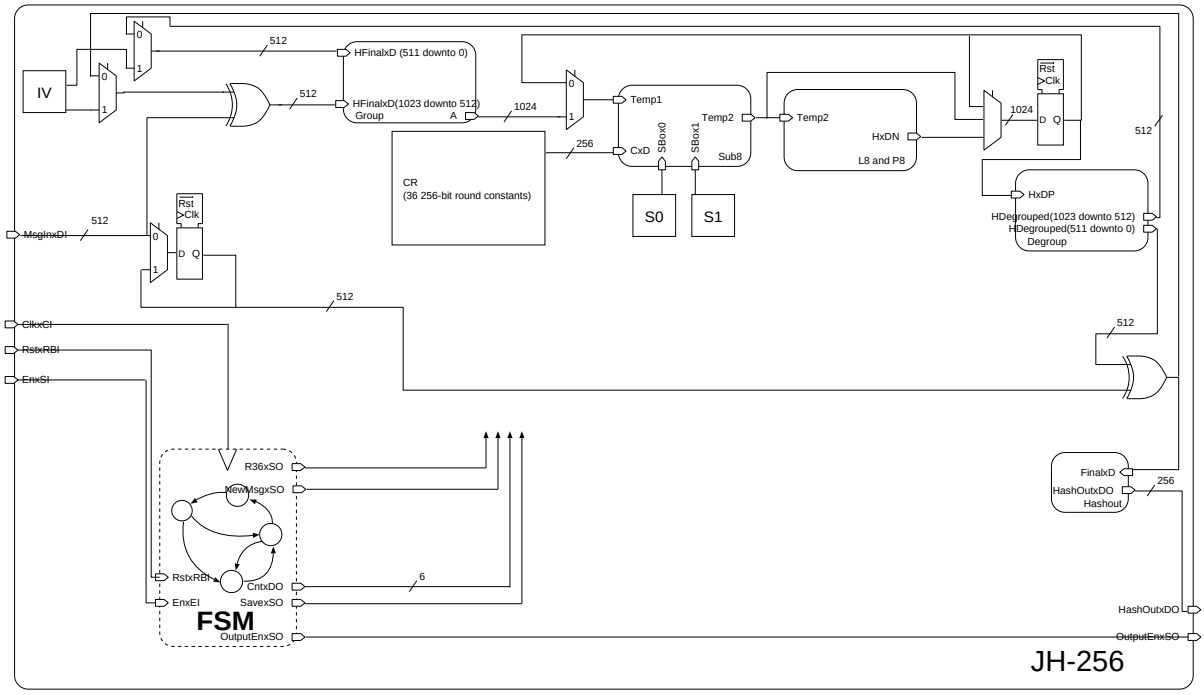


Figure 15: Block diagram of the implemented JH algorithm with constants stored

until the synthesizer couldn't find any feasible netlist anymore. The highest throughput achieved for both implementations of JH was 6.46 Gbit/sec at a clock frequency of 454.5 MHz. As shown in the plot in fig. 16, the first implementation used 67.6 kGE to achieve this throughput while the second one used only 65.6 kGE.

Test Coverage TetraMax showed a test coverage of 99.12% for our JH implementation.

4.6 SHAvite-3

Algorithm Description The SHAvite-3 algorithm was developed by Orr Dunkelman and Eli Biham at Technion in Haifa, Israel. Its message expansion function processes input blocks of 512 bits and generates 36 128-bit subkeys from each of them. During these calculations, the AES round function is called 16 times per block. Optionally, a 256-bit salt can be used. The subkeys are then passed to a block cipher which is applied to the 256-bit state. Thereby the AES round function is called 36 times, once for each subkey. The hash value is equal to the state matrix after processing all message blocks.

Golden Model The Golden Model for SHAvite-3 was written in Java. It is based on the same $GF(2^8)$ matrix manipulation class used for Fugue.
The AES round function was written from scratch and debugged using examples from the AES specification. Understanding and implementing the SHAvite-3 algorithm was more difficult because its specification is not very clear. Also, there was an error which had been fixed in the reference implementation and example output, but was still present in the algorithm specification. Later, it was corrected without our intervention, though.

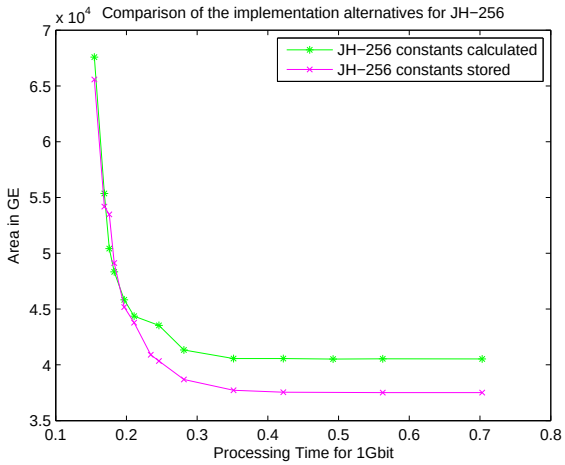


Figure 16: Area vs. Throughput plot of the two JH implementations

HDL Implementation A simple diagram of the compression function of SHAvite-3 is given in fig. 17. It was not easy to come up with an efficient hardware implementation of the algorithm that is still reasonably small. We first implemented SHAvite-3_{256} using a total of 9 AES units - six for the block cipher and three for the message expansion function. In this case, a message block can be processed in six clock cycles, and the average utilization of the AES units is about 96%. This implementation was too large to fit on our ASIC, though - we estimate circuit size to be around 130 kGE.

The second implementation contained just two AES units: One for the message expansion and one for the block cipher. A message block can be processed in 36 clock cycles. The average utilization of the AES units is only 72%, but the area is reduced to about 25 kGE.

Due to the dependency of each subkey on all previous subkeys, our implementation of the expansion function uses a 512-bit state matrix. The block cipher's state matrices (L_i, R_i) need to be stored twice to allow for the feedforward operation after each message block, thus the total size of the state registers is 1024 bits.

Synthesis We synthesized the circuit in Synopsys with different target speeds. The minimum clock period necessary to meet timing conditions is 3.7 ns, i.e. the maximum speed is 270 MHz. For the silicon implementation on "Roesti" the circuit is operated at 200 MHz.

Test Coverage The fault coverage of the final implementation was determined to be 98.81%.

5 Back-end design

5.1 Introduction

After the synthesis we had an overview of the key figures of the final implementations. The algorithms were easily split into two groups: the slow ones and the fast ones. In order to accomodate each group

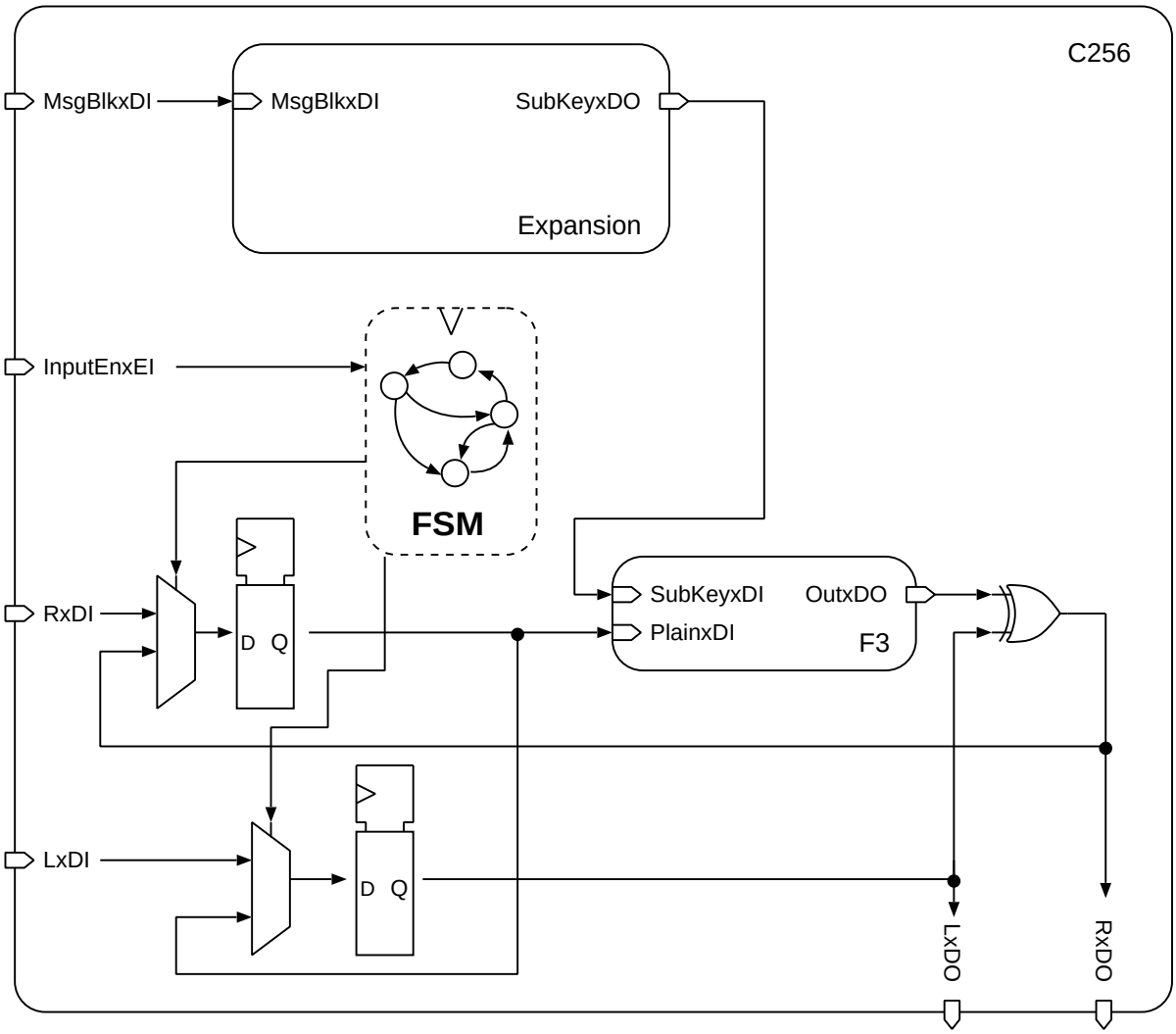


Figure 17: Simplified schematic diagram of SHAvite's compression function C-256.

Polenta:

	Area [kGE]	Frequency [MHz]	Input size [bits]	Output size [bits]	Latency	Throughput [Gbit/s]
BMW-256	95	125	512	256	18	3.56
Skein-256-256	27	125	256	256	19	1.68
Interface	9	125	8	8		
	130					

Roesti:

	Area [kGE]	Frequency [MHz]	Input size [bits]	Output size [bits]	Latency	Throughput [Gbit/s]
Fugue	26	200	32	256	2	3.2
Groestl	65	200	512	256	21	4.88
JH	44	200	512	256	36	2.84
SHAvite-3	48	200	512	256	36	2.84
Interface	12	200	8	8		
	195					

Table 2: Overview of key figures for all implemented algorithms.

of algorithms on a single ASIC, the team of the DZ came up with a new design, the "double mini@sic". The key figures of the final implementations are given in table 2[2]. Please note that the latency does not include the finalization rounds.

5.2 Padframe and Bonding Diagram

Our designs both have the same padframe and use the same supply pins. They have 60 pads each, but the QFN56 package in which the sample ICs will be delivered has only 56 pins, so four of the power pins are double-bonded. The DZ staff prepared the padframe and bonding diagram for us. It is shown in fig. 18.

5.3 Chip "Polenta"

Placing and routing A netlist of the whole circuit was synthesized giving a clock period of 7 ns in order to have 8 ns after back-end design. Placing was not a problem with a density of about 68%. In Figure 19 you can see how the chip area is shared between the functions. Once the cells were placed, the clock tree was inserted and the signals routed. After each step a timing analysis was run and the optimization routines of Encounter were used until no timing violations were found. This alone would not have sufficed. We had to re-synthesize BMW with a clock of 5.5 ns and sink the timing constraints in Encounter to 7.5 ns, too. Doing so, at the end of the process, we obtained a design wich gave no timing violations at 7.5 ns. The fabricated chip should, hence, meet the goal of 8 ns.

Power distribution Since the space needed by BMW, Skein and the interface sums up to 130 kGE, we had enough room to build a robust power distribution system. Two power rings run near the padframe and supply the cells via 6 power strips. Furthermore, a margin of 18 μm was left between the innermost

[2]You may be asking yourself what the names of the chips mean. *Polenta* and *Roesti* are specialties from the cantons Ticino and Bern, where the authors come from.

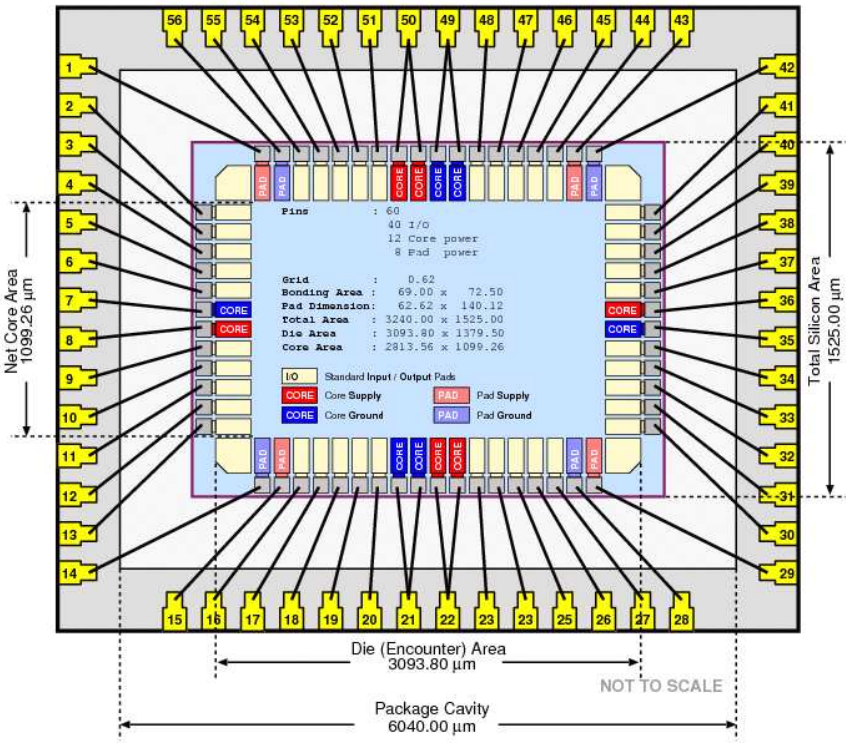


Figure 18: Bonding diagram of the UMC 180nm double mini@asic with QFN56 package, as used in our project. [11]

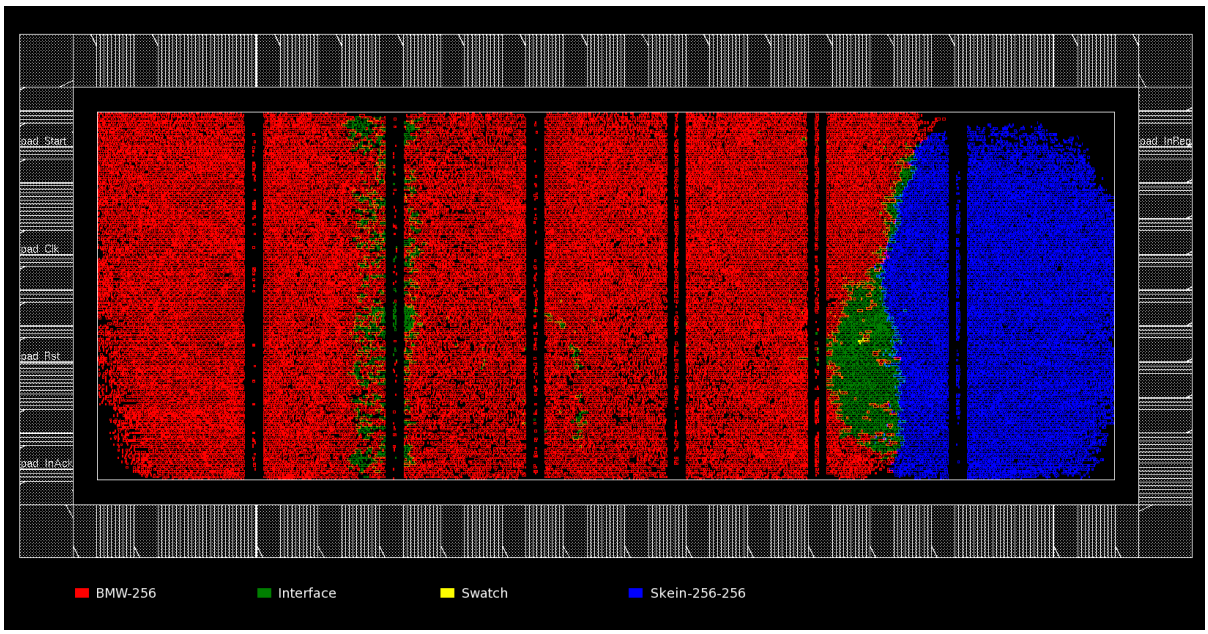


Figure 19: The floorplan of “Polenta” with the functions highlighted

ring and the cell area to allow for routing. With the routing done, a power analysis was performed. Since the two algorithms will never work at the same time, two separate analyses had to be run. Two VCD-files were built using the actual stimuli of the algorithms. The results were highly satisfying and are summarized in Table 3.

	BMW	Skein
Internal Power [mW]	20.37	11.82
Switching Power [mW]	25.43	11.44
Leakage Power [μW]	6.4	6.4
Total Power [mW]	45.8	23.27
Worst IR-drop [mV]	5.6	3.2

Table 3: Results of the power analysis

The worst IR-drop is barely 0.3% of the supply voltage and it is not likely to affect the speed of the circuit in a noticeable way.

5.4 Chip "Roesti"

For the back-end design we followed the tool chain set up by the DZ and trained in the VLSI II exercises. We used SoC Encounter for most tasks. The pin allocation for the double module was done manually. The power pins were allocated by the DZ experts, who did a double bonding (2 pads on 1 pin) on the power pads. The clock and reset pins were allocated close to the center in order to achieve a more equated path. We used the same pin layout for both chips as far as possible. For power distribution we used 5 strips to minimize IR Drop.

The Backend Design showed some differences in delay between Synopsys and Encounter. We had to recompile several Synopsys netlists in order to obtain our goal of 200 MHz frequency for Roesti. The key was to write scripts in order to automate the backend design process as far as possible, because several reiterations were necessary. The final floorplan showed that Groestl occupied the most area, which is because 200 MHz is still pretty fast for Groestl. SHAvite-3 got split into parts and distributed. The arrangement of the four functions is depicted in fig. 20. On the final "Roesti" chip the total cell area usage is about 71%.

6 Design for testability

Testability is crucial in microelectronics to reduce to a minimum the number of defective devices brought to the market. In this project we used scan-chains in order to detect faults. Each algorithm is provided with its own scan chain, inserted during synthesis with Synopsys. The test mode is enabled using the signal `ScanEnxT`. TetraMax was used for the generation of the test patterns and gave us the test coverage. We reported this value for all of the algorithms because a low coverage would be a drawback for that implementation. Luckily, there were no low values.

The structure described above was kept in the chip as well: a scan chain for each algorithm. The scan-in signals are derived from the data-in pins via a multiplexer. In a similar way the scan-outs are fed to the data-out pins. The test coverages of the chips are 98.88% for "Polenta" and 99.00% for "Roesti".

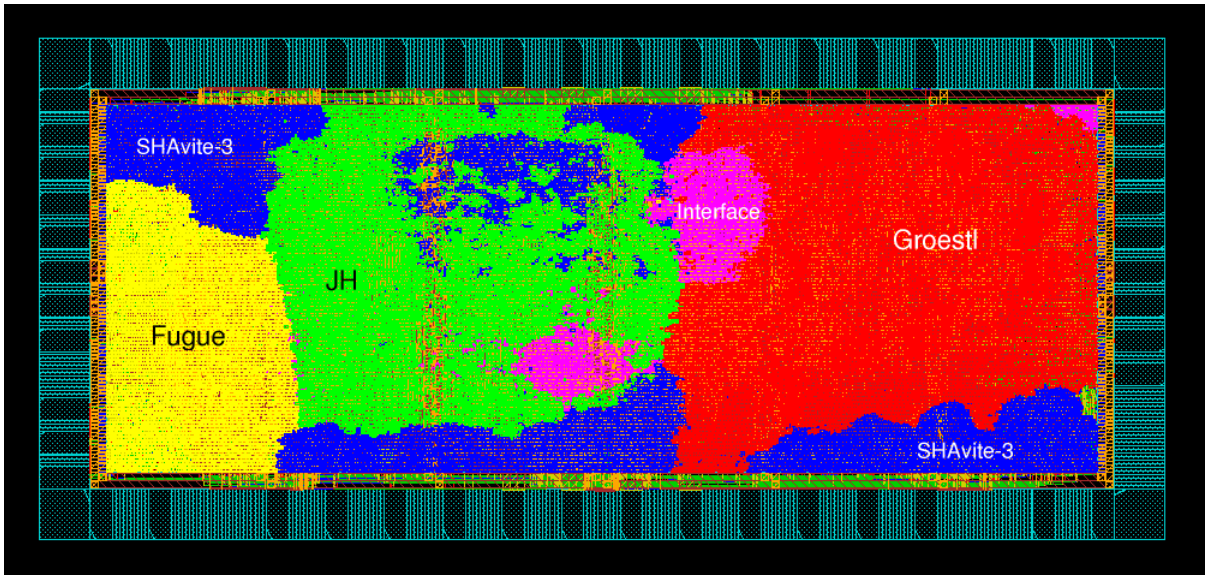


Figure 20: Distribution of the four hash functions on the “Roesti” chip.

7 Performance Evaluation

7.1 Comparison of circuit area and throughput

Figure 21 shows the required cell area in kGE plotted against the processing time for 1 Gbit of all algorithms implemented on our ASICs. For each algorithm, the selected point of operation is marked with a star.

The smallest of our algorithm implementations is Fugue, requiring only 26 kGE at a theoretical throughput of 3.2 Gbit/s. Fugue is also the algorithm with the best throughput/area ratio. Skein is similar in size, but with a considerably lower throughput. The worst algorithm in terms of hardware efficiency is Blue Midnight Wish - in our case it requires exorbitant 95 kGE to achieve an acceptable throughput. The remaining three algorithms have approximately the same throughput/area ratio as Skein and thus represent the “middle-class”.

7.2 Power efficiency

The power consumption is an important issue for portable devices wich have to run on a battery. The results of the power analysis tell us how much power the chip will draw during operation with a given algorithm. But different algorithms have different throughputs. Hence, the power consumptions are not suitable for a comparison. In Table 4 a more meaningful variable is shown: the energy necessary in oder to hash 1 Gbit of data.

8 Conclusions

Results The efficiency (throughput/area ratio) of our implementations after gate-level synthesis is generally in the same range as the results of Stefan Tillich et al. [3]. An overview is given in table 5 The

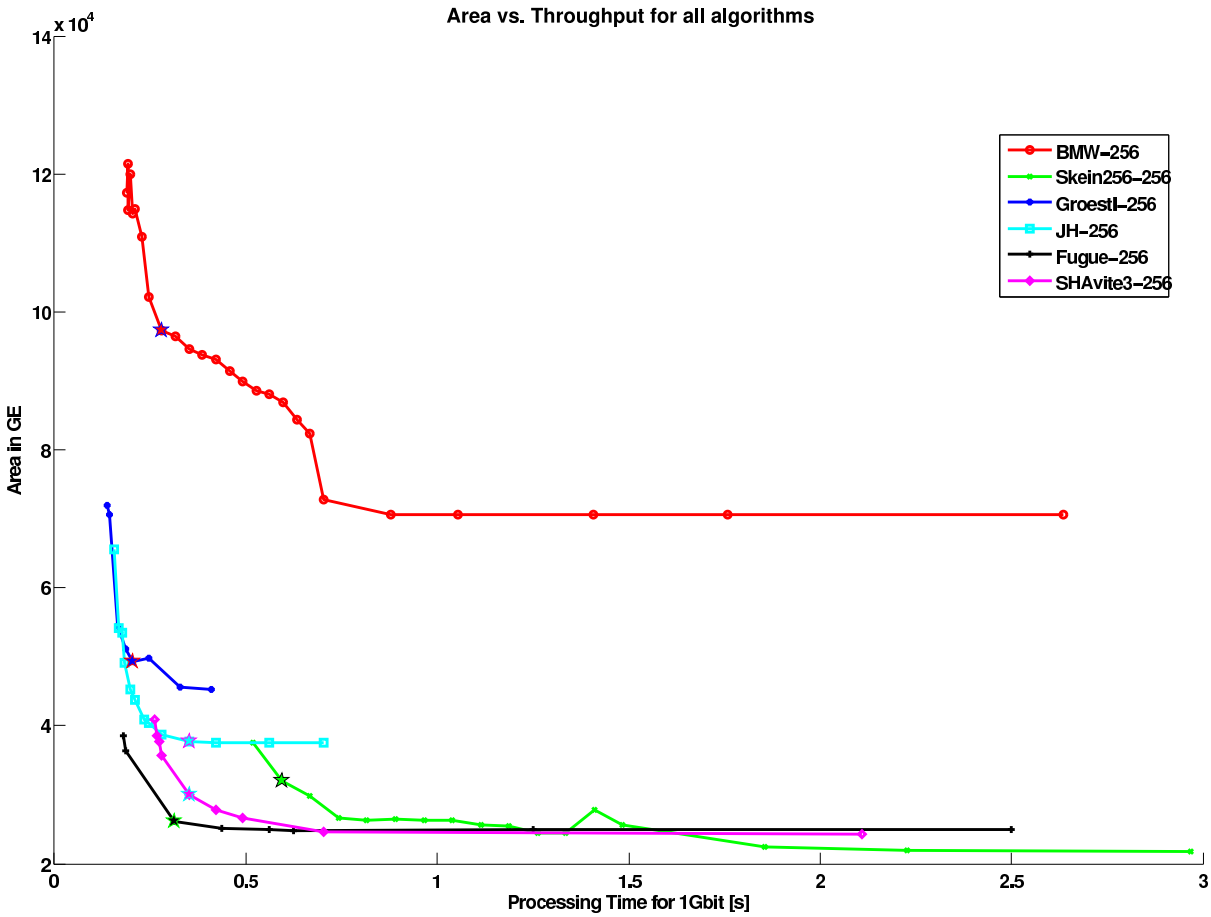


Figure 21: Different options for cell area and performance of our six hash algorithms.

Algorithm	$Power/Throughput$ $[\frac{mJ}{Gbit}]$
BMW	12.9
Skein	13.9
Fugue	29.1
Groestl	19.1
JH	23.2
SHAvite-3	23.2

Table 4: Power efficiency of the implemented algorithms

	Area Performance [$\frac{Mbit/s}{kGE}$]
BMW-256	37.5
Skein-256-256	62.2
Fugue	123
Groestl	75.1
JH	64.5
SHAvite-3	59.2

Table 5: The throughput per unit area of the implemented algorithms

algorithms BMW, Fugue, SHAvite-3 and Skein are a little more efficient in our case, even after being placed and routed on the chip. Grøstl and JH are slightly more efficient up to the synthesized netlist, but in the final chip they are operated at a suboptimal frequency. In any case, the performance differences with respect to [3] are rather small and the causes for them were not investigated in detail.

As discussed in chapter 7.1, there are quite important differences in area and throughput between the algorithms. The Fugue algorithm offers the best throughput/area ratio of the six candidates we considered. It is also best suited for a minimum area implementation. The fact that the message blocks are only 32 bits long allows to input a block every other cycle, which leads to a high utilization of the hardware. The Skein algorithm is also remarkable: Its area requirements are almost as modest as Fugue's, even though it accepts 256-bit input blocks. However, it achieves a considerably lower throughput. Blue Midnight Wish is the algorithm with the lowest efficiency: In our implementation, it requires more than three times as much area as Fugue, but provides only about half the throughput. As we don't know about any special benefit of BMW over the other candidates, we definitely can't recommend it from the hardware point of view.

Lessons learned Even though the algorithms are not very complex, the implementation of the Golden Models required more time than expected. It is non-trivial to implement a hash algorithm in accordance with a written specification, especially if it is unclear or even erroneous. Locating an error in the code can be difficult, especially if the intermediate results are not known. On the other hand, creating an own golden model for each algorithm is a good way of getting to know and understanding the algorithms.

Writing a VHDL model of an algorithm is not too difficult if a schematic diagram is drawn beforehand. However, the functional verification can be very time-consuming. Here the intermediate results are known or can easily be determined from the Golden Model, but the data flow is different in every clock cycle, and all values must arrive at the right register in the right cycle.

The maximum clock frequency determined by Synopsys DC is considerably higher than the achievable frequency in the final chip. In our case this wasn't a big problem, but if a chip is to be operated at a given target frequency, a margin must be included when estimating circuit size and speed from gate-level synthesis results.

Bibliography

References

[1] Sam Trenholme: *Rijndael's S-Box*, URL: http://samiam.org/s-box.html, Archived at http://www.webcitation.org/5nRMwoY0J on Feb 10, 2010

[2] Sam Trenholme: *AES' Galois Field*, URL: http://samiam.org/galois.html, Archived at http://www.webcitation.org/5nRN0iKuh on Feb 10, 2010

[3] Stefan Tillich, Martin Feldhofer, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Alexander Szekely: *High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein*, URL: http://eprint.iacr.org/2009/510.pdf, Archived at http://www.webcitation.org/5nRMp43nP on Feb 10, 2010

[4] Federal Information Processing Standards Publication 197: *Specification for the Advanced Encryption Standard (AES)*, URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Archived at http://www.webcitation.org/5nRN4qP4z on Feb 10, 2010

[5] Ewan Fleischmann, Christian Forler, and Michael Gorski: *Classification of the SHA-3 Candidates*, URL: http://eprint.iacr.org/2008/511.pdf, Archived at http://www.webcitation.org/5nRN7XMls on Feb 10, 2010

[6] IST Programme of the European Commission: *The SHA-3 Zoo*, URL: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo, Archived at http://www.webcitation.org/5nRNFQ3W4 on Feb 10, 2010

[7] National Institute of Standards and Technology: *Cryptographic Hash Algorithm Competition*, URL: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, Archived at http://www.webcitation.org/5nRNITQZg on Feb 10, 2010

[8] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu: *Finding Collision in the Full SHA-1*, URL: http://cryptome.org/wang_sha1_v2.zi, Archived at http://www.webcitation.org/5nRNRAVKw on Feb 10, 2010

[9] Pietro Gendotti: *Silicon Implementation of Non-AES-Based SHA-3 Second Round Candidates*. Master thesis at the Integrated Systems Laboratory, ETH Zurich, October 2009.

[10] Stefan Tillich: *Hardware Implementation of the SHA-3 Candidate Skein*, URL: http://eprint.iacr.org/2009/159.pdf, Archived at http://www.webcitation.org/5nRNsVAKQ on Feb 10, 2010

[11] ETH Zurich Microelectronics Design Center: *UMC180 Mini ASIC Setup*, URL: http://www.dz.ee.ethz.ch/en/information/ic-technologies/umc/180/mini-asic-setup.html, Archived at http://webcitation.org/5nRNujsDE on Feb 10, 2010

[12] Johannes Wolkerstorfer, Elisabeth Oswald and Mario Lamberger: *An ASIC Implementation of the AES SBoxes*, Topics in Cryptology - CT-RSA 2002, ISBN 978-3-540-43224-1, Vol. 2271/2002 pages 29-52.

[13] Danilo Gligoroski et al.: *Cryptographic Hash Function "Blue Midnight Wish"*, URL: http://people.item.ntnu.no/ danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf, Archived at http://www.webcitation.org/5nRNy7WJv on Feb 10, 2010

[14] Shai Halevi, William E. Hall, and Charanjit S. Jutla: *The Hash Function "Fugue"*, URL: http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue. index.html/$FILE/fugue_09.pdf, Archived at http://www.webcitation.org/5nRO4qvXN on Feb 10, 2010

[15] Lars R. Knudsen et al.: *Grøstl - a SHA-3 candidate*, URL: http://www.groestl.info/Groestl.pdf, Archived at http://www.webcitation.org/5nRO8BW6M on Feb 10, 2010

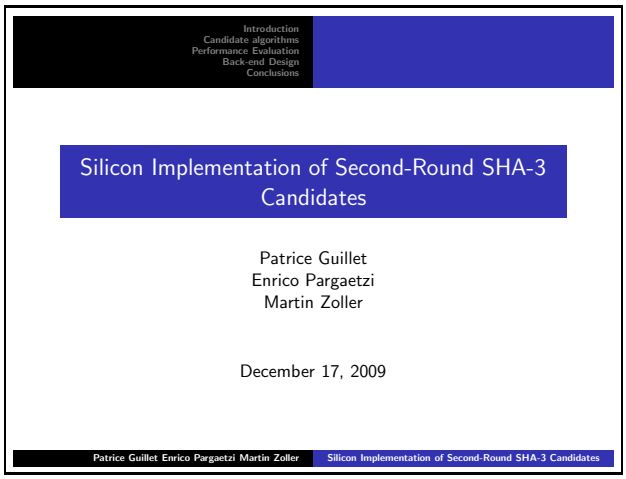
[16] Hongjun Wu: *The Hash Function JH*, URL: http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf, Archived at http://www.webcitation.org/5nROBimsp on Feb 10, 2010

[17] Eli Biham, and Orr Dunkelman: *The SHAvite-3 Hash Function - Tweaked Version*, http://www.cs.technion.ac.il/ orrd/SHAvite-3/Spec.23.11.09.pdf, Archived at http://www.webcitation.org/5nROFsIUJ on Feb 10, 2010

[18] Bruce Schneier et al.: *The Skein Hash Function Family*, URL: http://www.skein-hash.info/sites/default/files/skein1.1.pdf, Archived at http://www.webcitation.org/5nROIi248 on Feb 10, 2010

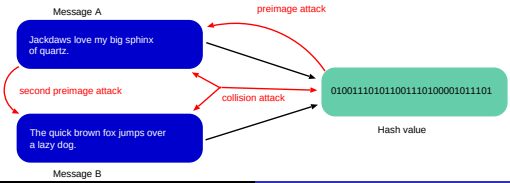
[19] Tobias Oetiker et al.: *The Not So Short Introduction to LaTeX2e*, URL: http://tobi.oetiker.ch/lshort/lshort.pdf, Archived at http://www.webcitation.org/5nROO2La6

A Presentation Slides



What is a hash function?

- A hash function transforms a message m of arbitrary length to a hash value of fixed length (e.g. 256 bits).
- Widely used in information security applications
- Goal: Resistance against preimage, second preimage, and collision attacks



SHA Hash Functions

- SHA (**S**ecure **H**ash **A**lgorithm) series: Hash algorithms as standardized by NIST
  - SHA-0: withdrawn shortly after publication
  - SHA-1: broken
  - SHA-2 family: SHA-224, SHA-256, SHA-384, and SHA-512
  - SHA-3: Competition in progress
- The existing SHA hash functions are insecure by design, even though SHA-2 is still computationally secure

The SHA-3 Competition

- **Goal:**
  - Secure
  - More efficient in Hardware/Software than SHA-2 family
- Currently Second Round running: 14 algorithms left
- August 2010: Conference on Second-round Algorithms
- **Wanted:** Comparable VLSI implementations of all algorithms
- **Algorithms implemented in hardware (UMC L180 process) in this thesis:**
  BlueMidnightWish, Fugue, Groestl, JH, SHAvite-3, Skein

Task Description

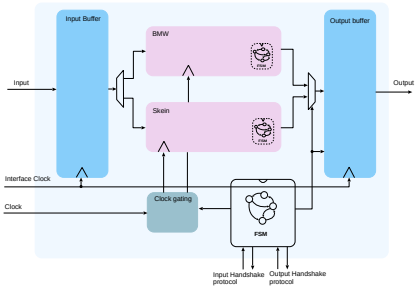
Silicon implementation of six candidate algorithms

- Golden Model based on specification
- VHDL implementation (trade-off between throughput and required cell area)
- Functional verification
- Synthesis and code optimization
- Back-end design (including test structures)
- Next semester: Testing

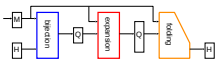
Interface

- Interface between I/O pins and algorithms
- Serial read-in/write-out of data (8 pins each)
- Uses a 512-bit input buffer and 256-bit output buffer
- Clock gating used to allow for power dissipation measurements
- "BlowUp" for fast data generation
- Developed by Pietro Gendotti, adapted

Hardware Implementation

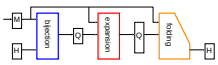


Blue Midnight Wish



- Input block length: 512 bits
- Word length: 32 bits
- Internal state: 16 words
- A bijective transform, a non-linear expansion and a folding function are applied in CBC-Modus to the input message

Blue Midnight Wish



- Input block length: 512 bits
- Word length: 32 bits
- Internal state: 16 words
- A bijective transform, a non-linear expansion and a folding function are applied in CBC-Modus to the input message

One CBC-Round

$$W_i \leftarrow \sum_{k=1}^{5} (-1)^{p_i} \left(M_{q_i} \oplus H_{q_i}\right) \qquad Q_i \leftarrow \bigoplus_{k=1}^{r} (W_i \lll s_k)$$

for $i = 16 \ldots 17$ **do**

$$Q_i \leftarrow M_{i-16} + M_{i-13} - M_{i-6} + K_i + \\ + \sum_{t=0}^{15} \bigoplus_{k=1}^{r} \left(Q_{i-(16-t)} \lll s_k\right)$$

end for

for $i = 18 \ldots 31$ **do**

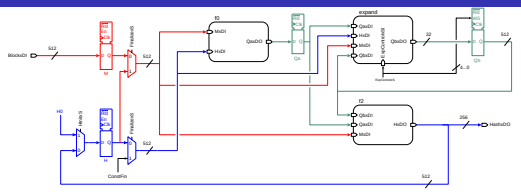
$$Q_i \leftarrow M_{i-16} + M_{i-13} - M_{i-6} + K_i + \\ + \sum_{t=0}^{13} \left(Q_{i-(16-t)} \lll s_t\right) + \\ + \sum_{t=14}^{15} \left(Q_{i-(16-t)} \lll s_t \oplus Q_{i-(16-t)}\right)$$

end for

$$XL \leftarrow \bigoplus_{i=16}^{23} Q_i \qquad XH \leftarrow XL \oplus \bigoplus_{i=24}^{31} Q_i$$

$$H_i \leftarrow H_{s_i} \lll \alpha_i + \left(XH \lll \beta_i \oplus Q_{b_i} \lll \gamma_i \oplus M_{c_i}\right) \\ + \left(XL \lll \delta_i \oplus Q_{d_i} \oplus Q_i\right)$$

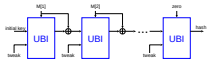
Hardware Implementation of BMW256



Basic structure

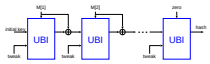
- 4 registers with 512 bits each
- controlled by a 37-states Moore machine
- Latency: 18 cycles per block
- Final round: 18 cycles

Skein



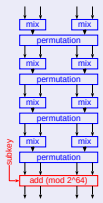
- Input block length: 256 bits
- Word length: 64 bits
- Internal state: 4 words
- The *Unique Block Iteration* applied to an input message block returns the key for the next round

Skein



- Input block length: 256 bits
- Word length: 64 bits
- Internal state: 4 words
- The *Unique Block Iteration* applied to an input message block returns the key for the next round

Inside UBI: 4 of 72 rounds



mix:

$$M_{out_0} \leftarrow \left(M_{in_0} + M_{in_1}\right)$$

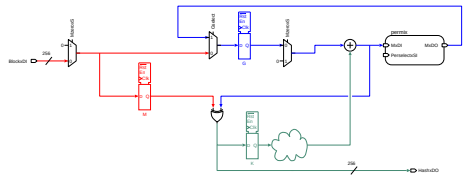
$$M_{out_1} \leftarrow \left(M_{in_1} \lll r\right) \oplus M_{out_0}$$

permute:

$$M_{0,1,2,3} \leftarrow M_{0,3,2,1}$$

- The subkey is computed in a simple key-schedule from the tweak and the key
- The rotation coefficients r are periodic: $r_i = r_{i+8}$

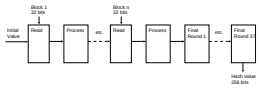
Hardware Implementation of Skein-256-256



Basic structure

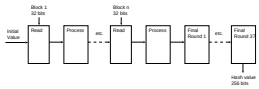
- 3x256 bits in data-registers and 103 bits in 3 registers for the control-path
- controlled by a 58-states Mealy machine
- Latency: 19 cycles per block
- Final round: 19 cycles

Fugue



- Input block length: 32 bits
- Word length: 32 bits
- Internal state: 30 words
- Structure similar to AES, but adapted to hash function requirements (e.g. using a more complex column permutation function)

Fugue

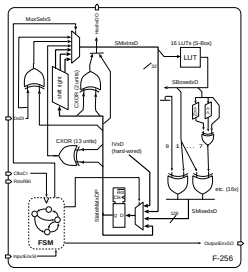


- Input block length: 32 bits
- Word length: 32 bits
- Internal state: 30 words
- Structure similar to AES, but adapted to hash function requirements (e.g. using a more complex column permutation function)

Round

S: State matrix (30×4 bytes)
M: Message block (1×4 bytes)
W: Column vector (16 bytes)
N: Constant 16×16 matrix
$S[10]$ += $S[10]$
for $i = 0, \ldots, 3$ **do**
  $S[i] \leftarrow M[i]$
end for
$S[8]$ += $S[0]$; $S[1]$ += $S[24]$
for $h = 1, \ldots, 2$ **do**
  $S \leftarrow ROR3(S)$
  $S \leftarrow CMIX(S)$
  for $i = 0, \ldots, 3$ **do**
    for $j = 0, \ldots, 3$ **do**
      $W[4 \cdot i + j] \leftarrow SBox(S[i][j])$
    end for
  end for
  $W \leftarrow N \cdot W$
end for

Hardware Implementation



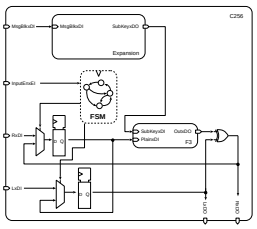
Basic structure

- registers storing the state matrices
- S-Box (8 bit to 8 bit mapping) implemented combinationally
- SMIX operation: Bitshifts and XORs
- Data flow controlled by Finite State Machine
- Latency: 2 cycles per 32-bit block
- Final round: 37 cycles

SHAvite-3

- Input block length: 512 bits
- Internal state: 1024 bits (5 matrices)
- 36 Rounds executed per Message Block
- Rounds include Message expansion (using AES) and encryption of the state matrix (AES) with the generated keys

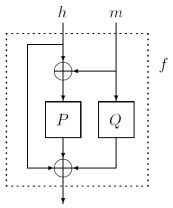
Hardware Implementation



- 2 implementations tested: One with 2 and one with 9 AES units
- Using 9 units is more efficient, but requires too much area
- One AES unit used for message expansion, the other one for the round function F3

Groestl

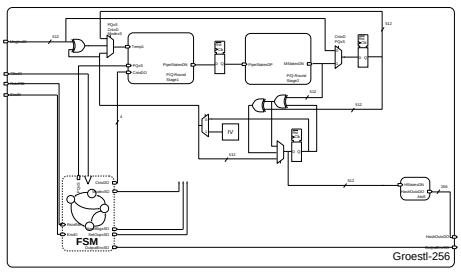
- Input block length: 512 bits
- Internal state: 8x8 8bit matrix
- Structure similar to AES, two similar rounds P,Q
- Rounds include operations AddRoundConstant, SubBytes, ShiftBytes, MixBytes
- Output Transformation is 10 P rounds



Hardware Implementation

- 3 implementations tested: parallel, LUT, comb.S-Box
- No big effect regarding throughput, but regarding size
- Final implementation computes P and Q alternating with one pipeline stage and a combinational S-Box
- Uses 3x512 FF for saving states

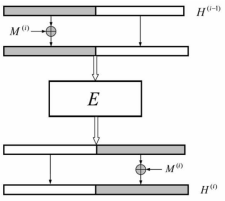
Hardware Implementation II



Groestl-256

JH

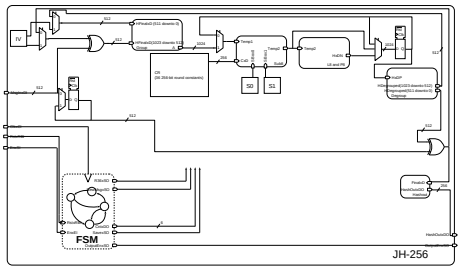
- Input block length: 512 bits
- Internal state: 16x16 4bit matrix, 1024 bits
- 36 Rounds executed per Message Block
- Rounds include one of two 4 bit S-Box substitutions, linear operation and permutations
- Theoretical Preimage Attack shown, computationally infeasible



Hardware Implementation

- 3 implementations tested: Constants calculated, constants stored, combinational S-Boxes
- No effects regarding throughput, small effect regarding size
- Final implementation stores S-Boxes and constants
- Uses 1024 + 512 FF for saving state and message

Hardware Implementation II



Figures of merit

Polenta:

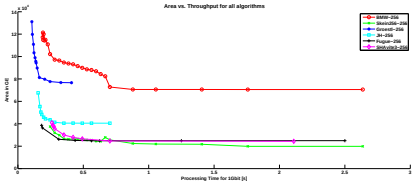
	Area [kGE]	Frequency [MHz]	Input size [bits]	Output size [bits]	Latency[1]	Throughput [Gbit/s]
BMW-256	95	125	512	256	18	3.56
Skein-256-256	27	125	256	256	19	1.68
Interface	9	125	8	8		
	130					

Roesti:

	Area [kGE]	Frequency [MHz]	Input size [bits]	Output size [bits]	Latency[1]	Throughput [Gbit/s]
Fugue	26	200	32	256	2	3.2
Groestl	65	200	512	256	21	4.88
JH	44	200	512	256	36	2.84
SHAvite-3	48	200	512	256	36	2.84
Interface	12	200	8	8		
	195					

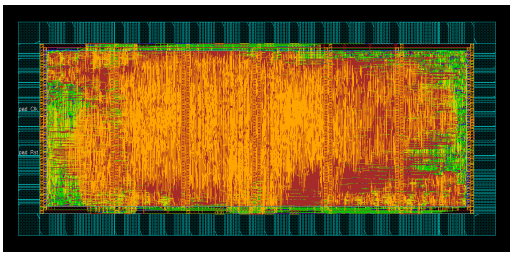
[1] without finalization

Performance Comparison



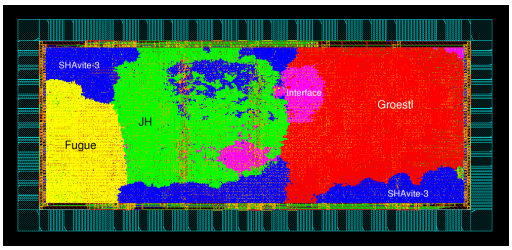
Back-end design

- View of Roesti in SoC encounter ($\sim$ 71% cell area usage):



Back-end design II

- Arrangement of the components on the chip "Roesti"



Conclusions

- Considerable loss in the achievable speed between Synopsys synthesis and SOC Encounter output
- Algorithms generally easy to implement, but some of the specifications were rather bad
- Functional verification takes more time than expected
- Creation of an own golden model is crucial to understanding of algorithms
- Performance: Better or equal to results of Stefan Tillich et al.
- Outlook: Completion of backend design, tape-out by Jan 18, 2010

B Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Semester Project at the Departement of
Information Technology and Electrical Engineering

Autumn Term 2009

Patrice Guillet
Enrico Pargaetzi
Martin Zoller

Silicon Implementation of Second Round SHA-3 Candidates

Advisors: Luca Henzen, ETZ J71.2, henzen@iis.ee.ethz.ch
Frank K. Gürkaynak, ETZ J60, kgf@ee.ethz.ch

Handout: September 2009
Due: December 2009

Two copies of the written report are to be turned in. Both copies remain property of the Integrated Systems Laboratory

1 Introduction

In recent years there have been a series of serious and alarming cryptanalytic attacks on several commonly-used hash functions, such as MD4, MD5, SHA-0, and SHA- 1 [1, 2]. These culminated with the demonstration of relatively efficient methods for finding collisions in the SHA-1 hash function [3]. Although there are several cryptographic hash functions, such as the SHA-2 family, that have not yet succumbed to such attacks, the U.S. National Institute of Standards and Technology (NIST) put out a call in 2007 for candidate proposals for a new cryptographic hash function family, to be dubbed SHA-3 [4].

Hash functions are algorithms for converting an arbitrarily large input into a fixed- length message digest. They typically consist of two main components: a compression function that operates on fixed-length pieces of the input, and a mode of operation that governs how apply the compression function repeatedly on the pieces in order to allow for arbitrary-length inputs. Cryptographic hash functions are furthermore required to have several important and stringent security properties including (but not limited to) first-preimage resistance, second-preimage resistance, collision resistance, pseudorandomness, and unpredictability.

2 Project Description

The submission document of all hash algorithms, which have been accepted at the first phase of the competition, provides an exhaustive specification of the internal components of the compression function and the modes of operation. It lists its tunable parameters and describe the resistance against several known attacks. Furthermore, the performances of software implementations of the function are also reported. However the hardware performances on various platforms, like ASIC or FPGA devices, is seldom given. While an average performance on high-end processors is generally not critical, implementability and flexibility in hardware is crucial, because the new standard will be implemented in a variety of lightweight devices. During this Semester Project the students are, thus, asked to compensate this lack in the algorithm specification.

3 Goals

The goals of this thesis are the implementation and evaluation of second round SHA-3 candidates. The following tasks should be accomplished during this project:

- Getting some insight into the field of cryptography and in particular into the field of hash functions.
- Learning how to design a digital VLSI circuit using VHDL and professional design tools.
- Understanding the basics of a strategic implementation technique.
- Fabrication of a test chip.

4 Milestones

The following milestones should be achieved during this semester thesis. Some milestones can be added or skipped, depending on the status of the project. A possible calendar in Fig. 1 shows all milestones.

1. Establish a project plan.
2. Select interesting candidate algorithms
3. Study relevant literature in order to understand the task of each functional block of the selected function. It should become clear that there are different implementation options for each of the operations in the compression function.
4. Write a model in Matlab or C code to understand the algorithm.
5. Write synthesizable VHDL code and test it with an appropriate testbench.
6. Synthesize the VHDL code in order to get estimates on the performance and area requirements of the design.
7. Back-end design. This includes insertion of scan chain and test structures, floor-planning, placement and routing, final verifications such as Design Rule Check (DRC), Layout Versus Schematic (LVS) and postlayout simulations.
8. Prepare the final presentation and document the entire project.

5 General Recommendations

Finally, some recommendations for this semester thesis:

- While coding VHDL, use the IIS standard coding style [5] documented by the Design Zentrum (DZ) website [6].
- VHDL coding is greatly simplified and accelerated using the Emacs editor and its famous and widely adopted VHDL mode. This Emacs installation at the institute supports among other powerful features VHDL syntax highlighting, signal and component declaration and instantiation, code beautifying, and automated sensitivity list updates based on the VHDL standard. Since most assistants at the IIS are quite familiar with this editor, they can read and evaluate your VHDL code (and help to solve problems) much faster. Please consult the corresponding FAQ under the following link:

  `http://www.dz.ee.ethz.ch/en/information/hdl-help/emacsvhdl-mode.html`

If the design will be fabricated using the 90 nm technology by UMC, the students should be aware of the following technological limitations during this project:

- The total dimensions of the chip including the seal ring and bonding pads can not exceed 1.925 mm x 1.925 mm. After the area required for the seal ring and the pads is subtracted the remaining core area is approximately 2.022 mm^2.

- The available active area corresponds to the area of approximately 450 kGE. At the end of a synthesis run the Synopsys Design Compiler will report the total cell area. Note that this number does not include area that you need to reserve for routing. Although the UMC technology enables high routing efficiency, you should still reserve about 10-20% of area for routing (especially for power routing).
- Further important information are listed under the DZ website:

  `http://www.dz.ee.ethz.ch/en/information/ic-technologies.html`

6 Project Realization

6.1 Meetings

Weekly meetings will be held between the students and the assistants. The exact weekly meeting time and location will be determined to fit the schedule of the students and the assistants. These meetings will be used to evaluate the status and progress of the project. The students are highly encouraged to provide up-to-date block diagrams, to take notes, and to maintain a "todo"-list, i.e., bring along paper and pencil.

6.2 Reports

Documentation is an important and often overlooked aspect of engineering. One short intermediate report and one final report (the semester thesis) are to be completed within this study. Note that the intermediate report should be designed to be part of the final report.

The common language of engineering is de facto English. Therefore, the intermediate and final report of the work is preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif (for block diagrams) is strongly encouraged by the IIS staff.

Short Intermediate Report The short intermediate report should be written in such a way to become part of your final report and should already contain the final outline. Furthermore, general information about the topic, a description of the problem, explanations of related terminology, and descriptions of similar approaches in literature (with corresponding references to books, papers etc.) should be provided.

Final Report The final report has to be presented at the end of this project and three copies need to be handed out and remain property of the IIS. This report is only accepted when the keys for the ETZ building have been properly returned. Note that this task description is part of your Thesis and has to be attached to the end of your final report. A data disc (e.g., CD or DVD) containing all essential files of your project should also be added to the final report.

6.3 Presentation and Demonstration

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date has to be determined.

7 Calendar (Tentative)

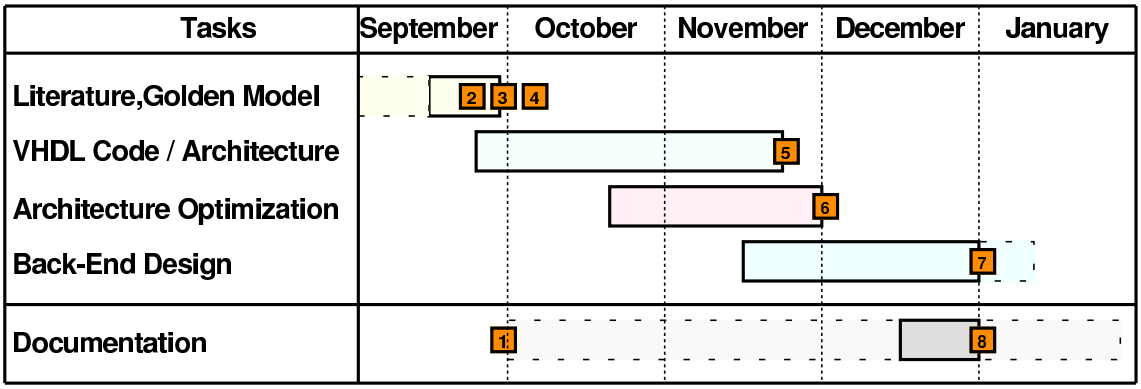


Figure 1: Tentative Calendar

References

[1] Florent Chabaud and Antoine Joux, *Differential Collisions in SHA-0*. In Hugo Krawczyk, editor, CRYPTO, volume 1462 of Lecture Notes in Computer Science, pages 56-71. Springer, 1998.

[2] Xiaoyun Wang and Hongbo Yu, *How to Break MD5 and Other Hash Functions*. In Ronald Cramer, editor, EUROCRYPT, volume 3494 of Lecture Notes in Computer Science, pages 19-35. Springer, 2005.

[3] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, *Finding Collisions in the Full SHA-1*. In Victor Shoup, editor, CRYPTO, volume 3621 of Lecture Notes in Computer Science, pages 17-36. Springer, 2005.

[4] NIST, *Call for a New Cryptographic Hash Algorithm (SHA-3) Family*. Federal Register, Vol.72, No.212, `http://www.nist.gov/hash-competition`, 2007.

[5] H. Kaeslin, *From VLSI Architectures to CMOS Fabrication*, Cambridge University Press, 2008.

[6] Design Zentrum website: `http://www.dz.ee.ethz.ch` and VHDL naming conventions: `http://www.dz.ee.ethz.ch/en/information/hdl-help.html`

Zurich, September 21, 2009 Prof. Dr. Wolfgang Fichtner

The thesis will not be accepted without returning the keys!