Cryptographic Library for FPGAs using SME

A Bachelor Project Defense

Jacob Herbst (mrw148) and
Jonas Flach-Jensen (sjm233)
Institute of Computer Science (DIKU)

Agenda

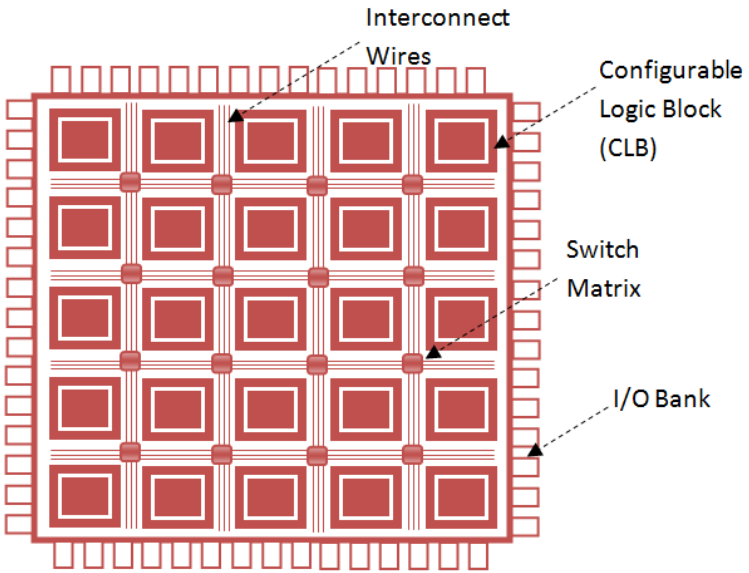
- Why is this project interesting?
- Presentation of MD5 and AES
- Results
- suggestions for future work

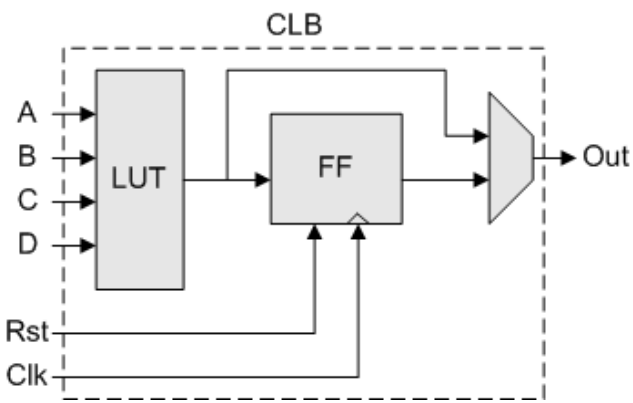
Motivation - Cryptography

- Is ubiquitous
- Often Hardware-focused
- This project has focused on implementing a variety of algorithms instead of nitpicking one algorithm.

Motivation - Why use FPGAs?

- Architecture
- Configurable not only in computation but also in interface
- Often fast as the overhead from generality is ommitted
- Low power consumption (fixed precision)
- Difficult to program (Xilinx Vivado 😩)





MD5

- Cryptograpic hash function
- Merkle-Damgaard construction
- Four different compression functions, 64 rounds

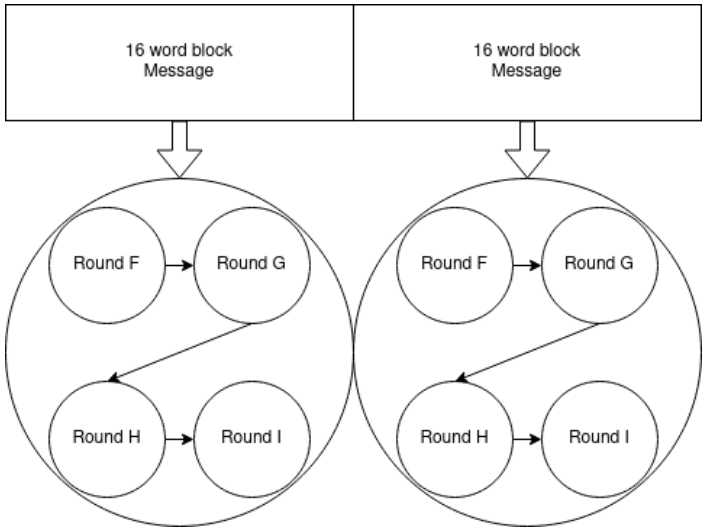


Figure: MD5 round

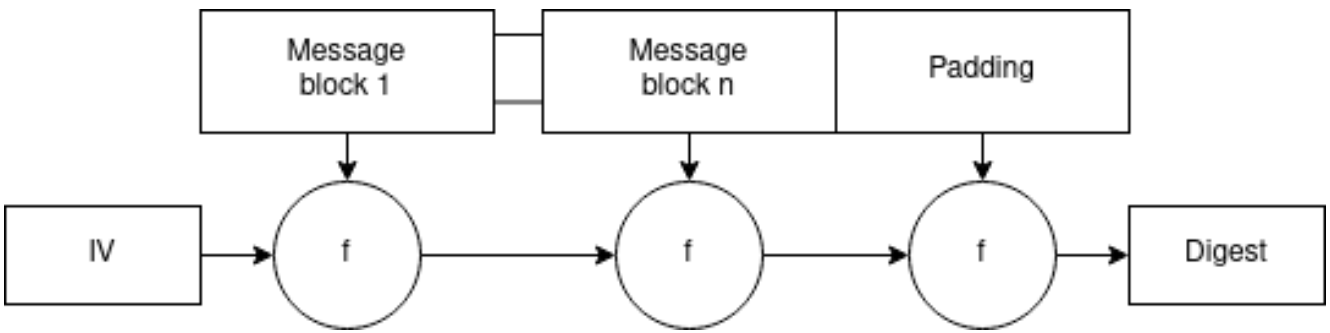


Figure: Merkle-Damgaard construction

MD5 - Optimizations

- Naive; 1 simple process, 4 busses
- Pipelined version; clocked process for preprocessing and each compression stage
- Same idea for SHA-2

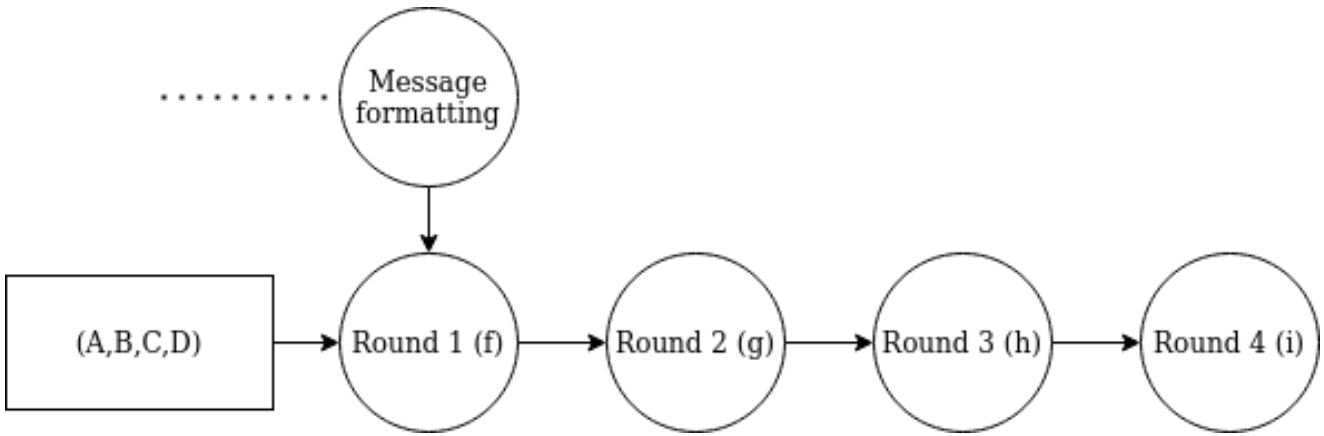


Figure: MD5 pipeline

MD5 - Pipeline

- Stalls on large messages due to data dependency
- Solution is enabling multiple inputs

Independent message blocks									
clock	0	1	2	3	4	5	6	7	8
	P_1	M_1	F_1	G_1	H_1	I_1	C_1		
		P_2	M_2	F_2	G_2	H_2	I_2	C_2	

Dependent message blocks												
clock	0	1	2	3	4	5	6	7	8	9	10	11
	P_1	M_1	F_1	G_1	H_1	I_1	C_1					
		P_2	M_2	-	-	-	-	F_2	G_2	H_2	I_2	C_2

AES

- The algorithm is Rijndael. It is a Block Cipher and a Substitution-permutation (SP) network.
- Four steps, or lookups:

$$W_i \begin{cases} K_i & \text{if } i < N \\ W_{i-N} \oplus \text{SubWords}(W_{i-1} \lll 8) \oplus \text{rcon}_{i/N} & \text{if } i \geq N \text{ and } i \equiv 0 (\text{mod } N) \\ W_{i-N} \oplus \text{SubWords}(W_{i-1}) & \text{if } i \geq N,\ N > 6, \text{ and } i \equiv 4 \text{ (mod } N) \\ W_{i-N} \oplus W_{i-1} & \text{otherwise} \end{cases}$$

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02_{16} \\ S[a] \\ S[a] \\ S[a] \cdot 03_{16} \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 03_{16} \\ S[a] \cdot 02_{16} \end{bmatrix}$$

$$e_j = T_0[a_{0,3}] \oplus T_1[a_{1,2}] \oplus T_2[a_{2,1}] \oplus T_3[a_{3,0}] \oplus k_j$$

AES - Optimization

- Fast naive version
- Pipelined by splitting up rounds
- no data dependecy

Results from pipelining

- $hi(x) = x + 2 \cdot blocks$
- $lo(x) = 2 + x \cdot blocks$
- $C(x) = x + 2 \cdot blocks$
- Zedboard bus: 2132 MBps

MD5: Is easily optimised beyond the memory limit.

SHA: Worst hard to optimize because of dependecies.

AES: Reached limit with current approach? Potentially other approaches can reach higher.

ChaCha: Easily reaches limit.

MD5

Version	f_{max} (Mhz)	$clocks_{hi}$	$TP(MBps)_{hi}$	$clocks_{lo}$	$TP(MBps)_{lo}$	LUT	FF
Naive	2.38	b	152	b	152	11607	2304
$Proc_4$	9.50	hi(6)	266	lo(6)	101	10247	5226
$Proc_8$	19.00	hi(10)	532	lo(10)	122	10087	7538
$Proc_{16}$	33.50	hi(18)	937	lo(18)	119	10206	12162
$Proc_{32}$	65.00	hi(34)	1817	lo(34)	123	10149	21347
$Proc_{64}$	**115.00**	**hi(66)**	**3209**	**lo(66)**	**112**	**10350**	**39718**

SHA

Version	f_{max} (Mhz)	$clocks_{hi}$	$TP(MBps)_{hi}$	$clocks_{lo}$	$TP(MBps)_{lo}$	LUT	FF
Naive	2.1	b	134.4	b	134.4	24330	2560
$Proc_4$	**8.0**	**hi(6)**	**223.9**	**lo(6)**	**85.3**	**24466**	**8938**
$Proc_8$	8.0	hi(10)	223.8	lo(10)	51.2	24756	14066

AES

Version	f_{max} (Mhz)	clocks	TP(MBps)	LUT	FF	BRAM
Naive	22	b	352	10612	3195	0
TBox	25	b	400	16458	3195	0
$Proc_4$	68	C(3)	544	16474	2817	0
$Proc_{11}$	**208**	**C(12)**	**1663**	**15659**	**4383**	**0**
$Proc_{22}$	217	C(24)	1662	15454	7401	0
$BRAM_{11}$	195	C(31)	1556	10012	10398	72

ChaCha

Version	f_{max} (Mhz)	clocks	TP(MBps)	LUT	FF
Naive	1.25	b	80	14670	3457
$Proc_{11}$	40.00	C(9)	1279	14736	16898
$Proc_{22}$	**82.00**	**C(20)**	**2557**	**17565**	**32420**

Results compared to CPU

Comparing to CPU over GPU

- GPU is the standard approach for hardware acceleration.
- CPU is closer in TDP.
- CPU is more approachable and making a GPU version would require higher development time.
- already reached some board limitations.

MD5: 4.5 times faster than any comparable CPU version.

AES: Proximity of the C# version, but cannot compete AES-NI.

SHA: Only half the speed of i5, but faster than ARM processor. Potential for improvement.

ChaCha: Reaches the bandwidth limit of the Zynq board. Not quite i5 speed. Doubling processes: 2714 MBps. Only 150 more than the previous version.

MD5

	Naive	Proc_{64}	C#	C	OpenSLL_{low}	OpenSLL_{high}
Pi	152	3210	287	256	42	293
i5	152	3210	604	622	81	691

AES

	Naive	Proc_{11}	C#	C	OpenSLL_{low}	OpenSLL_{high}
Pi	400	1963	70	198	72	89
i5	400	1699	1963	340	847	5722

SHA

	Naive	Proc_{4}	C#	OpenSLL_{low}	OpenSLL_{high}
Pi	134	224	163	42	165
i5	134	224	438	61	461

ChaCha

	Naive	Proc	OpenSLL_{low}	OpenSLL_{high}
Pi	80	2557	84	307
i5	80	2557	388	3092

Power usage

Why use TDP?

- Selling point for FPGAs
- Only possibility in the current stage.
- Time consuming to do actual tests.

All of our versions is significantly more power efficient than the CPUs:

FPGA	1.765W
i5	65.000W
Pi	7.500W

Future work

Critical work

- The Dependency routing needs to be fixed in the hashing functions.
- Make hashes able to switch between messages to circumvent the stalling.
- Make a useful interface to expose our implementations.

Optimizing work

- Investigate SHA hopefully getting the performance to reasonable levels
- Test if different approaches of AES approaches could yield better performance. Sugestions:
  1. Naive but pipelined
  2. Stateful BRAM.

Comparing work

- Compare to other research papers results which often are written in HDL to see if SME can actually provide comparable results. This would require better a better FPGA.
- Test against a GPU.

Questions?

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE

HEHE