2011 Workshop on Lightweight Security & Privacy: Devices, Protocols, and Applications

Towards an Ultra Lightweight Crypto Processor

Begul Bilgin, Elif Bilge Kavun and Tolga Yalcin
Department of Cryptography
Institute of Applied Mathematics, METU
Ankara, Turkey
e-mail: e142589, e168522, tyalcin@metu.edu.tr

***Abstract*—In this paper, a lightweight processor suitable for lightweight cryptographic applications is presented. The processor instruction set is based on the stack-based ZPU architecture. In addition, a simple generic plug-in interface is implemented in order to allow integration of application specific coprocessors to the main processor core. In the current version of the processor, a simple direct memory access engine and a serialized Klein cipher coprocessor are implemented and connected to the processor core. Through these engines, it is possible to implement various lightweight security and authentication schemes in a code and area effective way. A simple assembler code is written and tested on the processor in order to verify the functionality of the processor core and coprocessors. The code implements a Davies-Meyer coding scheme and uses the Klein block cipher as a hash function. The GCC toolset originally written for the 32-bit ZPU is being adapted to work with the 8-bit processor core. The designed processor is synthesized using VeriSilicon GSMC 0.13um low-power process high-density standard cell library for a target operating frequency of 100 KHz, and the resultant gate count is 4.5K GE.**

Keywords-lightweight; cryptographic; ZPU; processor; Klein

I. INTRODUCTION

Security is a main issue on lightweight systems such as RFIDs, smart tags, etc. Such systems, just like their larger counter parts as in Internet protocol based communication systems; rely on a data packet with appropriate headers and trailers, as well as the supporting protocols for secure data encapsulation and authentication. However, their lightweight nature sets some challenging demands.

A lightweight crypto engine must be extremely low-cost and low-power. A custom hardware solution is the key to both of these targets. However, it lacks the flexibility and programmability offered by the software solutions. On the other hand, software-only solutions suffer from low performance compared to custom hardware.

A hybrid solution seems to be the answer and a compromise between cost and performance. A cryptographic processor can run time and power consuming security operations effectively, which can easily be configured via programming for the target application without time consuming redesign and high cost NRE issues.

Much work has been done on the cryptographic processor design and implementation related research, some even targeting lightweight systems. Most of such processors make use of cryptography specific instruction set extensions and usually specialize on effective execution of a single algorithm or application [1-4].

In this paper, a coprocessor based ultra-lightweight cryptographic processor is presented. It is composed of a core processor and application-specific cryptographic coprocessors connected to the main core via a memory I/O based plug-in interface.

The main processor instruction set is based on that of the ZPU, which is a zero-operand stack-based [5] minimalistic design. In order to cope with the lightweight design targets, the 32-bit datapath of the ZPU is modified to 8 bits. In accordance, several instructions intended for half and quarter word access are stripped from the design, further simplifying the design. In the rest of this paper, the resultant lightweight processor core is referred as the lightweight processing unit (LPU).

ZPU comes with its complete GCC toolchain. However, modifying the datapath in the original design results in the necessity of modification of the original toolchain, which is still under progress.

Two coprocessors are implemented in the current design. One is a simple DMA engine targeted towards fast transfer of memory arrays, which is one of the most common operations in the manipulation of data packages. Another common operation is the checksum calculation, whose functionality is embedded into the DMA engine.

The other coprocessor is a serialized version of the new lightweight Klein block cipher, which operates on 64-bit data blocks with various key sizes. In the current implementation, key size is chosen as 64 bits. The serial datapath is 8 bits, which also agrees with the LPU datapath width. The Klein module also imitates a DMA behavior in order to provide quick data processing.

There is no hash function embedded into the design. Instead, the Klein block cipher is used in Davies–Meyer mode in order to provide hashing functionality, as is done with the Present cipher in [6]. However, the flexible plug-in interface of the LPU allows incorporation of future lightweight hash functions in the design with minimal effort.

The rest of the paper is organized as follows: In Section II, the LPU core design is presented. Section III summarizes

978-0-7695-4340-6/11 $26.00 © 2011 IEEE
DOI 10.1109/LightSec.2011.17

the simple DMA engine, whereas Section IV outlines the implementation details of the serialized Klein core. In Section V, the hardware performance results are presented. Finally, the conclusions and future directions are summarized in Section VI.

II. MICROPROCESSOR CORE

The microprocessor core has to be suitable for lightweight applications in terms of area and power consumption. In literature, there exist many different architectures and freeware processor designs [7-8]. However, it is difficult to find an instruction set architecture (ISA) which is both compact, code-efficient and also has GCC support which provides C programming capability for the ease of use. One architecture that satisfies all target features is Zylin Processing Unit (ZPU) [9], which is an open source architecture that allows deployments to implement many versions without running into license problems. The most important strength of this architecture is that it is an extremely simple design and it is very easy to implement from scratch to suit specialized needs and optimizations. Therefore, a new lightweight design is created from scratch which is one-to-one instruction set compatible with the ZPU, namely Lightweight Processor Unit (LPU).

Furthermore, in our envisioned architecture, the main work horses the so-called "coprocessors", which are responsible for all the processing power demanding jobs (encryption, hashing, fast memory transfer, etc.). The main function of the microprocessor is limited to organization of the input/output data (packet handling) and initialization of coprocessors, and in doing so the microprocessor's performance is the least important issue. Therefore it is crucial that the microprocessor core design has to be kept as simple as possible in order to ensure minimal gate count and power consumption, with minimal regard for performance. All the existing microcontrollers with GCC support, except for the ZPU, are register based losing several gates for the register files. However, in the current architecture, use of registers is only required for data array start addresses and lengths, which are already built inside the coprocessors. ZPU, with its stack based architecture, removes all the registers from the design, providing a sufficient and yet effective platform for our target system.

The main difference of LPU comes from the datapath size and the use of memories. For lightweight implementation purposes, the datapath width should be minimized. Unlike the original ZPU core, where the datapath width is 32 bits, for many lightweight applications, an 8-bit datapath is more than enough. Also, the original ZPU core requires dual-port memories with both read and write support in the same cycle on both ports, which is a very demanding requirement. Most FPGA architectures and ASIC technologies do not offer such memories. On the other hand, the extremely simple instruction set architecture of the LPU can be easily implemented using only single-port memories.

A. Architecture Overview and Instruction Set

LPU is a stack-based processor. Unlike many instruction set architectures [7-8], it uses zero operand. The elements which are at the top of the stack are used as operands. Using this approach, instructions can fit in 8 bits, which results in a very compact processor architecture.

Let's explain the stack-based operation principle by means of a simple instruction: the AND instruction. The instruction is defined as (mem[sp+1] = mem[sp+1] + mem[sp]; sp = sp + 1) Basically; when an AND instruction comes, LPU takes the topmost two values of the stack (pops) and ANDs them. Then, it adds the result of the operation to the top of the stack (pushes). As the stack is physically RAM-based, data is never taken out from the stack. Instead; the first data, which is pointed by the stack pointer (let sp=10, mem[10]), is taken and stored temporarily. Then, stack pointer is incremented by 1 and next data (mem[11]) is read. The AND operation is performed on the stored data and the present value of the memory addressed by the stack pointer. After the AND operation, the result is stored onto the top of the stack, which is pointed by the last value of the stack pointer (mem[11]). The input values are lost, and can not be used for next operations. However, it is possible to store those using different instructions and temporary registers, if necessary. At the end of the operation, the program counter is incremented by 1 and the next instruction is fetched. Then, the new instruction is decoded and necessary steps are performed.

LPU architecture is also flexible in terms of instructions. As already mentioned, different implementations of ZPU may support different numbers of instructions. For LPU; there is a minimum set of 15 required instructions. It is theoretically possible to implement all other instructions via software emulation. However, this would not only decline overall performance of the processor, it would also enlarge the required program memory space enormously. Therefore, the emulation feature of the original ZPU architecture can not be used in LPU. Instead, all the supported instructions are implemented as hard-wired.

The ZPU instructions are completed in three cycles:

- **Fetch cycle:** To both decode the instruction and fetch the first operand from stack,
- **FetchNext cycle:** To store the first operand in the temporary register and fetch the second operand from stack,
- **Execute cycle:** To execute the target operation and store the result back into the stack.

Some of the instructions are completed in 2 cycles, but the number of cycles is fixed to 3 cycles in order to simplify the overall design. The extra unnecessary cycles is an acceptable compromise for the resultant simple architecture. Furthermore, there is only a single memory access (either read or write) per cycle, allowing the use of single-port RAMs. The only penalty is the necessity to use separate program and data memories. However, this, in practice, does not affect the overall resource utilization.

Figure 1 shows the top level block diagram of the LPU.

B. Instruction Decoder

Instruction decoder decodes the 8-bit instruction coming from the program memory according to the program counter and generates the control signals for individual processor

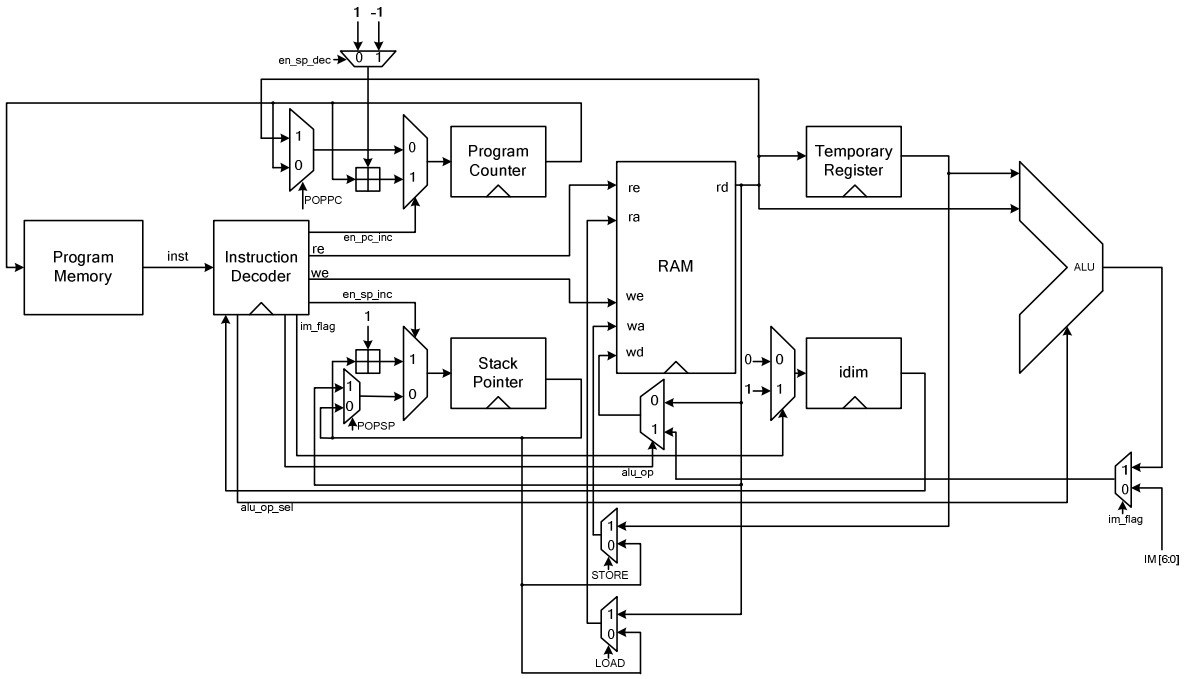

Figure 1. Block diagram of LPU core.

blocks (ALU, RAM, stack pointer, program counter, etc.). Every instruction is executed in 3 clock cycles; thus, instruction decoder also guarantees that the next program memory value is not taken until the last cycle using the active signal and the busy signal (from coprocessors).

The program counter is incremented by 1 for all instructions, except for pc branch instructions. These instructions take the value of the read data from memory as the program counter value. In addition, stack pointer "increment-by-1" and "decrement-by-1" enables are defined for corresponding cycles and instructions together with read and write enables of the RAM.

The most significant bit of the 8-bit instruction is the "immediate" instruction (IM) select, if instruction[7] is 1, then immediate value will be taken. Otherwise, the most significant three bits will be controlled first. If instruction[7:5] is "010" or "011"; then the instructions will be identified as "STORESP" or "LOADSP", respectively. If they are all zero, then the next bit is controlled. In case that the fourth most significant bit is 1, the ADDSP operation is performed. In this instruction, an ALU operation enable is also sent to carry out the addition. In other cases, instructions are identified according to their least significant nibble, and the ALU enable is produced if required.

C. Memory Organization

The memory organization of LPU, which covers the addresses of the coprocessors and the microprocessor, is quite simple. The 8-bit address space is used for memory organization. The most significant 4-bits (nibble) select the coprocessor and the remaining nibble addresses the specific location within the memory space of the selected coprocessor.

Table I shows this memory organization. Here, each input space (shown as 0xn0-0xnF) has 16 addresses. For example, the KLEIN core input is addressed as 0x00-0x1F, which means 2x16 = 32 bytes are available for KLEIN input and outputs (32x8 =256 bits). In general case, this scheme provides convenience for addressing and simplifies the decoder/encoder logic.

As can be seen from the table, the most significant nibble is used to select the coprocessor and the least significant nibble is used to select a specific I/O location and to address the words of the selected I/O inside that coprocessor. This way, each I/O can be 16-bytes = 128-bits long, which is sufficient if used as 0x00-0x1F for KLEIN algorithm.

D. Program Memory

With the proposed 8-bit instruction set, which also corresponds to 8-bit program memory addressing, the maximum addressable program space is only 256 lines. This, even with the coprocessor operation, which cuts back the program lengths tremendously, is a serious limitation. Therefore, paging is implemented for the program memory. After exclusion of half word and byte load and store, multiply, divide, modulus, and return from interrupt (popint) instructions, eight page instructions (page0 to page7) are implemented instead. Each page instruction sets the program counter to one of the eight 256-line (byte) pages in the program memory, resulting in a maximum of 2048 lines of program space.

The paging scheme is quite simple. In cases where the program counter has to jump to a subroutine on another

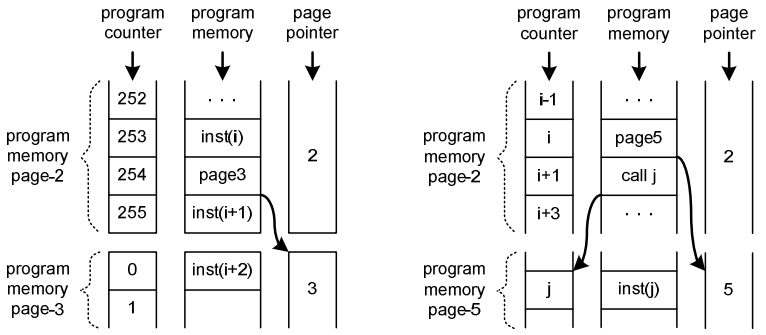

Figure 2. Program memory paging for regular flow (left) and subroutine jumps (right)

page, the corresponding page's instruction is executed before the jump instruction (call, poppc, callpcrel, pushpc, eqbranch, neqbranch). This necessitates that the previous page instruction has not been executed yet. It is handled by delaying the update of page pointer to the end of the next instruction. While this is a very simple working solution, it means that paging occurs by one instruction delay. Therefore, each page instruction has to be inserted one instruction before its intended effect, or has to be followed by a no operation instruction. In the present implementation, the former solution is implemented. For example, at the end of each page within the program, a page instruction in inserted to enable switching to another (possibly the next) page. While the program counter automatically returns to 0 after reaching 255, the page pointer is manually incremented by means of executing page*M* as the last line of code within page*N*, where *M* >= *N*+1. However, due to the delayed effect of the page instruction, it is placed into line 254 instead of line 255, which instead is populated with a regular instruction part of the overall program. Both schemes are illustrated in Figure 2.

There are two main disadvantages of paged operation: The first one is the overhead caused from the additional page instructions, which add to the total code length. It can be minimized via optimization of the compiler, which is in fact the second disadvantage. The original GCC compiler written for 32-bit operation has to be further modified in order to handle pages in an effective way. Currently, such optimizations are left to a future version of the compiler under development. It is anticipated that they will not be needed, taking into account the additional program space gained via paging.

E. Coprocessor Interface

The implemented coprocessor plug-in interface is quite simple, like the memory organization. In each coprocessor RAM, one of the 8-bit memory addresses (the address 0x-F) acts as the virtual command-status register (CSR) to provide an interface between the main processor and the coprocessors.

Firstly, the input memories of the chosen coprocessor are filled by the microprocessor. Then a write is issued to the corresponding CSR, instructing the coprocessor to start its operation. This raises the coprocessor active flag, which acts as a system-wide busy signal and halts the LPU core. Once the coprocessor is done, it pulls down the system-wide busy signal, allowing the processor to continue its program execution with the next instruction in line.

TABLE I. MEMORY ADDRESS ORGANIZATION

Address	Address Name	Description
0x00-0x1F	KLEIN	Klein coprocessor address space
0x20-0x3F	HASH	Reserved space for hash coprocessor
0x40-0x4F	DMA	DMA engine address space
0x50-0x5F	RSVD	Reserved space
0x60-0xEF	MSG	Message address space
0xF0-0xFF	STACK	Stack address space

F. Software Development Tools

LPU processor has GCC toolchain modification effort is still in progress. When completed, a full LPU GCC toolchain, including the GCC compiler, debugger and profiler, will be offered. The compiler, debugger and profile will be custom versions of the GCC tools re-compiled for LPU architecture.

In the present work, a simple Perl based parser is used to generate machine code from hand written assembler code. The parser supports all 34 hardware coded instructions.

A sample compiled code is given in Table II.

G. Security Considerations

Security leakage is a serious problem in cryptographic hardware implementations. Generally, all cryptographic algorithms are assumed to be vulnerable to these attacks if there are not special precautions in the implementation.

In our processor, in order to be protected against attacks, the instruction cycles can be modified so that no registers are left idle in any cycle of any instruction. Every register can be assigned a dummy operation in a fashion that the overall operation flow is not affected.

A second precaution can be continuously operating the coprocessors with random data. This can easily be achieved after simple modifications on the control circuitry. However, a true random number generator is required. It should be made impossible for an outside observer to detect when the coprocessors are run with real or random data.

The approximate area increase in the instruction modification scheme is below 10 percent for the processor core, while there is area increase for the coprocessors in either case. However, running the coprocessor cores continuously increases the overall power consumption considerably. This may be undesired for most embedded applications.

III. DIRECT MEMORY ACCESS ENGINE

The function of the simple DMA engine (coprocessor) is to perform fast transfer of memory arrays via hardware acceleration. In software, this functionality can be done within a while loop reading from one array and writing to another. However, in this software based scheme each read-write-repeat cycle takes 18 lines of code, which corresponds to 54 clock cycles in actual execution. On the other hand, with a pipelined hardware accelerator, this can be done within two cycles, one for read and one for write after the initial read-only cycle.

The DMA engine has three configuration registers in addition to its virtual CSR: source address pointer, destination address pointer and length register. The contents of these registers are set by the LPU software. This corresponds to a worst case of 14 lines of code and 42 clock cycles of execution. Once this is done, a dummy write is done onto the CSR, taking another 2 lines of code (6 clock cycles). The rest of the execution depends on the length of the transfer, which can be formulated as $2L+6$ cycles, where L is the length of the array to be transferred. Of the 6 extra cycles, 2 is the initial pipeline delay, and 4 is the CRC write delay.

CRC calculation and storage is an additional functionality embedded into the DMA engine. For each array being transferred, CRC checksum is calculated in parallel, and transferred as the trailer bytes of the destination array.

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
a_0											
a_1						a_0					
a_2						a_1	a_0				
a_3	a_2					a_0	a_1				
a_4	a_3	a_2				a_1	a_0	$2*a_2$			
a_5	a_4	a_3	a_2			a_0	a_1	$2*a_3$	$2*a_2+3*a_3$		
a_6	a_5	a_4	a_3	a_2		a_1	a_0	$2*a_4$	$2*a3+3*a_4$	$2*a_2+3*a3+a_4$	
a_7	a_6	a_5	a_4	a_3	a_2	a_0	a_1	$2*a_5$	$2*a_4+3*a_5$	$2*a_3+3*a_4+a_5$	$2*a_2+3*a_3+a_4+a_5 = b_0$
b_0	a_7	a_6	a_5	a_4	a_3	a_1	a_0	$2*a_6$	$2*a_5+3*a_2$	$2*a_4+3*a_5+a_2$	$2*a_3+3*a_4+a_5+a_2 = b_1$
b_1	a_0	a_7	a_6	a_5	a_4	b_0	a_1	$2*a_7$	$2*a_6+3*a_7$	$2*a_5+3*a_2+a_3$	$2*a_4+3*a_5+a_2+a_3 = b_2$
b_2	a_1	a_0	a_7	a_6	a_5	b_1	b_0	$2*a_0$	$2*a_7+3*a_0$	$2*a_6+3*a_7+a_0$	$2*a_5+3*a_2+a_3+a_4 = b_3$
b_3	b_2	a_1	a_0	a_7	a_6	b_0	b_1	$2*a_1$	$2*a_0+3*a_1$	$2*a_7+3*a_0+a_1$	$2*a_6+3*a_7+a_0+a_1 = b_4$
b_4	b_3	b_2	a_1	a_0	a_7	b_1	b_0	$2*b_2$	$2*a_1+3*a_6$	$2*a_0+3*a_1+a_6$	$2*a_7+3*a_0+a_1+a_6 = b_5$
b_5	b_4	b_3	b_2	a_1	a_0	b_0	b_1	$2*b_3$	$2*b_2+3*b_3$	$2*a_1+3*a_6+a_7$	$2*a_0+3*a_1+a_6+a_7 = b_6$
b_6	b_5	b_4	b_3	b_2	a_1	b_1	b_0	$2*b_4$	$2*b_3+3*b_4$	$2*b_2+3*b_3+b_4$	$2*a_1+3*a_6+a_7+a_0 = b_7$
b_7	b_6	b_5	b_4	b_3	b_2	b_0	b_1	$2*b_5$	$2*b_4+3*b_5$	$2*b_3+3*b_4+b_5$	$2*b_2+3*b_3+b_4+b_5 = c_0$
c_0	…										

Figure 3. Data flow within the serialized Klein datapath.

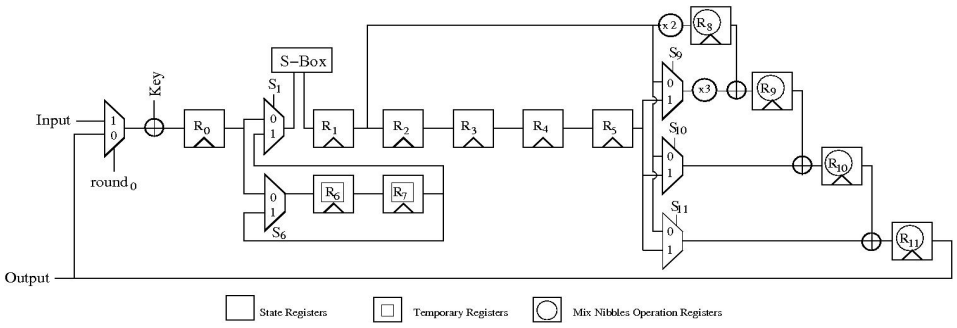

Figure 4. Block diagram of the serialized Klein datapath.

IV. KLEIN CIPHER COPROCESSOR

Klein [10] is a block cipher designed by Zheng Gong, Svetla Nikova and Yee-Wei Law to be used in RFID tags or similar systems that uses resource efficient cryptographic algorithms. It also has a good performance in software because of its wise permutation layer.

A. Structure of Klein

Klein is basically a Substitution-Permutation Network with 64 bits of message input. The length of the key may vary between 64, 80 and 96 bits and depending on them the round numbers are taken to be 12, 16 or 20 respectively.

In every round, after the modular addition of the round key, the state is divided into 16, 4 bits blocks and they go through the same 4x4 S-Box. The rotation used in Klein, namely RotateNibbles operation, is not bit oriented but byte oriented.

As the last step of the round, the state is divided into two 32 bits blocks and the MixNibbles operation, which is the same as the MixColumns operation of AES, is applied to the blocks.

B. Key Schedule

Unlike the round transformation, the key schedule in Klein is a Feistel like structure. The subkeys can be generated on-the-fly during each round transformation.

C. Serialized Implementation of Klein

Klein-64 block cipher is already a lightweight block cipher when implemented in parallel with area 1981GE. It can become even lighter with serial implementation similar to what was done to Keccak in [11] however, the throughput decreases, that is why this version can be used when the speed of the encryption is not important.

In this serialized architecture, the data is processed in bytes and all the operations are done in serial except the S-Box operations. The S-Boxes are thought as two 4-bit S-Boxes working in parallel. MixNibbles operation is also done in serialized fashion. Therefore a state S^i can be thought as $a^i_0 \| a^i_1 \| \ldots \| a^i_7$.

The RotateNibbles operation indicates that in every round a^i_0 and a^i_1 bytes will be used in MixNibbles operation as the last two bytes. Therefore they should be kept in the temporary registers which are shown as R6 and R7 in the data flow (Figure 3). Moreover, the MixNibbles operation uses 4 registers, R8 through R11 and so the total number of registers that should be used in the state is 12. The key xor of every byte is applied before they come to the first register R0, the substitution of every byte is done just before they are written into the register R2 and they are in registers R2 through R5 until they are not needed anymore and the output of R11 is the input of the next round. The overall output is also taken from the last register. As the data flow indicates,

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
k_0								
k_1	k_0							
k_2	k_1	k_0						
k_3	k_2	k_1	k_0					
k_4	k_3	k_2	k_1	k_0				
k_5	k_4	k_3	k_2	k_1	k_4	k_0+k_4		
k_6	k_5	k_4	k_3	k_2	k_5+k_1	k_4	k_0+k_4	
k_7	k_6	$k_5 = m_0$	k_4	k_3	k_6+k_2	k_5+k_1	k_4	k_0+k_4
m_0	k_7	$k_6 = m_1$	k_5	k_0+k_4	k_7+k_3	k_6+k_2	k_5+k_1	k_4
m_1	m_0	$k_7 = m_2$	k_6	k_0+k_4	k_7+k_3	k_6+k_2	k_5+k_1	k_4
m_2	m_1	m_0	k_7	k_0+k_4	k_7+k_3	k_6+k_2	k_5+k_1	$k_4 = m_3$
m_3	m_2	m_1	m_0	x	k_4	k_7+k_3	k_6+k_2	$k_5+k_1 = m_4$
m_4	m_3	m_2	m_1	m_0	x	k_4	k_7+k_3	$k_6+k_2 = m_5$
m_5	m_4	m_3	m_2	m_1	m_4	m_4+m_0	k_4	$k_7+k_3 = m_6$
m_6	…							

Figure 5. Data flow within the serialized Klein key schedule datapath.

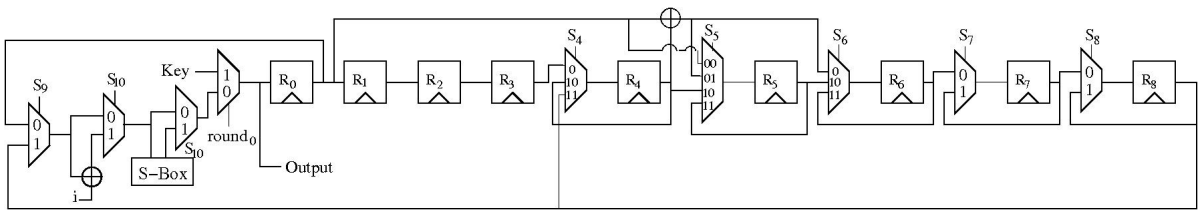

Figure 6. Block diagram of the serialized Klein key schedule datapath.

one round finishes in 8 clock-cycles in this implementation. The overall block diagram of the serialized Klein datapath can be seen in Figure 4.

The key schedule part of Klein is also implemented by using 8 bits data flow. There are no long operations that should be divided into sub-parts as in MixNibbles operation however, there are many temporary registers since the key schedule is a Feistel structure. That is why 9 registers are used. The data flow (Figure 5) for key schedule uses the same idea in state data flow, but this time the substitution operation is not applied to every byte and so the add counter operation therefore there are two other multiplexors for those operations.

It can be seen from the block diagrams (Figures 4 and 6) that one round of Klein finishes in 8 clock-cycles. Therefore the whole Klein encryption for one block is done in 96 clock-cycles. When compared with the parallel implementation, this application is almost 8 times slower, however it also requires less area. This serialized implementation requires 1365GE which is %32 less than the original implementation. Actually it is very close to the serial implementation of Present with 1075GE which was given in [4].

V. Implementation Results

The core processor (LPU), the DMA engine and the Klein cipher coprocessor have been described in Verilog-HDL language and synthesized with VeriSilicon GSMC 0.13um low-power process high-density standard cell library [12] for best area. The resultant gate counts are 1.8K, 0.6K and 1.7K, respectively. After the integration of the coprocessors into the LPU, the total area turned to be 4.5K, mainly because of the wrapper built around the LPU core for burst cipher operations.

Synthesized LPU power consumption is totally 153.4µW @100 KHz, while the total power consumption including the coprocessors is 347 µW at the same frequency. However, synthesis assumes 70% activity in the design, where the actual power consumption is a function of the program being executed. Therefore the actual power consumption is expected to be orders of magnitude lower than these figures.

VI. Conclusion

In this paper, the hardware design of a lightweight cryptographic processor is introduced. It is a base study towards a universal architecture complete with GCC toolkit. The system is composed of a core processor (LPU), which is based on the ZPU architecture, and a memory I/O based plug-in interface, which allows integration of any coprocessor to the main core. In its current state, two coprocessors, a DMA engine and the serialized version of the Klein block cipher, are implemented and embedded to the core. Several adaptations are applied to have an efficient LPU design. The functionality of the processor is verified by a hand-written assembler code (mapped to machine code via the Perl-based parser) which operates the Klein coprocessor in Davies-Meyer mode to provide secure hashing. When the GCC toolchain modification effort is completed, it will be possible to map C-codes directly to the processor program memory.

The current architecture is still open to further development. 8-bit addressing scheme both limits the program and data memory (including coprocessors and stack) to 256 bytes. While for most of the lightweight applications, 256 bytes of data memory might be sufficient, the same may not apply to the program memory. In cases, where the program memory might have to be larger than 256 bytes, a paging scheme has to be implemented.

Also in the current design, only a block cipher coprocessor (Klein) is implemented, and hashing is performed by running this block cipher in Davies-Meyer configuration. The choice of Klein is arbitrary, and aimed to demonstrate the open architecture of the proposed system. In a real application with solid specifications, it will be possible to define the system needs and plug-in the relevant coprocessors, such as an actual hash function module or a key management unit. The real strength of the system with its GCC support will then be utilized to its full extend.

Still, to our knowledge, this is the first lightweight cryptographic processor that offers this much flexibility and functionality with less than 5K GE.

References

[1] M. Hutter, M. Feldhofer, and T. Plos, "An ECDSA Processor for RFID Authentication," Proceedings of Workshop on RFID Security - RFIDSec'10, 6th Workshop, Istanbul, Turkey, June 7-9, 2010.

[2] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A Survey of Lightweight-Cryptography Implementations," IEEE Design and Test of Computers, November-December 2007 (vol. 24 no. 6), pp. 522-533.

[3] A. Ricci, M. Grisanti, I. De Munari, and P. Ciampolini, "Design of a 2uW RFID Baseband Processor Featuring an AES Cryptography Primitive," 15th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2008, pp. 376-379, 2008.

[4] C. Rolfes, A. Poschmann, G. Leander, and C. Paar, "Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents," Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, Volume 5189/2008, pp. 89-103, 2008, doi: 10.1007/978-3-540-85893-5_7

[5] J. L. Hennessy, D. A. Patterson, A. C. Arpaci-Dusseau, "Computer Architecture: A Quantitative Approach", Fourth edition, Morgan Kaufman, 2007.

[6] A. Bogdanov, G. Leander, C. Paar, A. Poschmann, B. Robshaw, Y. Seurin, "Hash Functions and RFID Tags : Mind The Gap," Workshop on Cryptographic Hardware and Embedded Systems - CHES 2008, Lecture Notes in Computer Science, Springer-Verlag, 2008.

[7] MIPS32 Architecture, MIPS Technologies.

[8] OpenRISC, http://opencores.org/project,or1k

[9] http://repo.or.cz/w/zpu.git?a=blob_plain;f=zpu/docs/zpu_arch.html

[10] Z. Gong, S. Nikova, and Y.W. Law, "KLEIN: A New Family of Lightweight Block Ciphers," University of Twente, CTIT, DIES Group, Internal Report, 2010.

[11] E. B., Kavun, T. Yalcin, "A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications," Proceedings of RFIDSec'10, Istanbul, 2010.

[12] VeriSilicon GSMC 0.13um Low-Power Process High-Density Standard Cell Library Databook, 2006 VeriSilicon Microelectronics (Shanghai) Co., Ltd.

TABLE II. LPU CODE EXAMPLE

C code	ASM code
`int main( void ) {`	
`volatile int *KLEIN_CSR ; KLEIN_CSR = (volatile int*)0x1F ;`	
`volatile int *KLEIN_LEN ; KLEIN_LEN = (volatile int*)0x1E ;`	
`volatile int *KLEIN_SRC ; KLEIN_SRC = (volatile int*)0x1E ;`	
`volatile int *KLEIN_DST ; KLEIN_DST = (volatile int*)0x1E ;`	
`volatile int *KLEIN_KEY ; KLEIN_KEY = (volatile int*)0x10 ;`	
`volatile int *MSG_INOUT ; MSG_INOUT = (volatile int*)0x60 ;`	
`volatile int *MSG_CSR ; MSG_CSR = (volatile int*)0xEF ;`	
`while (1) {`	

`*MSG_CSR = 0 ; // Message CSR is cleared by software`
`// To issue ready to external world`
	`im 0`
	`nop`
	`im 1`
	`im 111`
	`store`

`while ( *MSG_CSR == 0 ) ; // Wait until set externally`	`im 1`
	`im 111`
	`load`
	`storesp 4`

`// Run Klein in burst mode reading 16 bytes of`
`// plaintext from MSG_INOUT[1...16] and writing`
`// 16 bytes of ciphertext to MSG_INOUT[17...32]`
	`loadsp 0`
	`im 0`
	`eq`
	`im -8`
	`neqbranch`

`// Read message length from MSG_INOUT[0] into KLEIN_LEN`
`*KLEIN_LEN = MSG_INOUT[0] ;`
	`im 96`
	`load`
	`im 30`
	`store`

`// Set ciphertext start address to MSG_INOUT[1]`
`*KLEIN_SRC = MSG_INOUT + 1 ;`
	`im 97`
	`nop`
	`im 30`
	`store`

`// Set plaintext start address to MSG_INOUT[1] + LEN`
`*KLEIN_DST = *KLEIN_SRC + KLEIN_LEN ;`
	`im 30`
	`load`
	`loadsp 0`
	`im 2`
	`ashiftleft`
	`im 30`
	`add`

`// Send start to KLEIN engine (0x01 means OFB mode)`
`*KLEIN_CSR = 1 ;`
	`im 1`
	`nop`
	`im 31`
	`store`

`}`	`im -39` `poppcrel`
`}`	