INSTRUCTION SET EXTENSIONS FOR ACCELERATING SNOW 3G ON A MULTI-THREADED SOFTWARE-DEFINED RADIO PLATFORM

Chris Jenkins[1], Michael Schulte[1,2], and John Glossner[3]
[1]University of Wisconsin, Department of ECE, Madison, Wis., USA
[2]AMD, Inc. Research and Advanced Development Labs, Austin, Tex., USA
[3]Sandbridge Technologies, Tarrytown, N.Y., USA
cdjenkins@wisc.edu, schulte@engr.wisc.edu, jglossner@sandbridgetech.com

ABSTRACT

Software-defined radio (SDR) is an emerging technology that facilitates having multiple wireless communication protocols on one device. New cellular standards, such as HSPA+, LTE, and LTE+, require speeds in excess of 40 Mbps. SNOW 3G is a new stream cipher approved for use in these systems. Our optimized software-only version of SNOW 3G achieves a throughput of 14 Mbps per thread for message confidentiality and 18 Mbps per thread for message integrity on our SDR platform. To have secure cellular communications in SDR platforms, the performance of security algorithms must be improved. This paper presents instruction set architecture (ISA) extensions and hardware designs for SNOW 3G processing. These ISA extensions and hardware designs are evaluated for the Sandbridge™ Sandblaster® 3011 (SB3011) SDR platform. These enhancements improve the performance of optimized software implementations by a factor of 2.5 for message confidentiality and 1.3 for message integrity.

1. INTRODUCTION

Software-defined radios (SDRs) use a combination of software and hardware to dynamically support multiple wireless communication standards. These devices have been widely recognized as one of the most important new technologies for wireless communication systems [1]. SDRs enable the efficient implementation of a diverse set of wireless communication systems. SDRs also provide the ability to change communication protocols and dynamically update communication systems through over-the-air software downloads [2].

Current cellular systems perform cryptographic operations, such as confidentiality (i.e., encryption and decryption) and integrity (i.e., message authentication), on programmable processors. The former prevents other users of the wireless medium from eavesdropping on the user's data transmission or voice call. The latter ensures a message or voice frame that a user sends has not been altered during transmission—either intentionally or inadvertently. As data rates over the air interface increase, cryptographic performance can become a bottleneck.

The 3rd Generation Partner Project (3GPP) has defined two standards for providing confidentiality and message integrity during transmission. The first standard uses a block cipher called Kasumi [3]. The second uses a stream cipher called SNOW 3G [4]. Both provide confidentiality and integrity, but have fundamentally different properties to prevent attacks on one cipher translating into attacks on the other.

Future DSP architectures will need to enable cryptographic processing at very high data rates. For example, fourth-generation radio access networks seek to provide over-the-air throughputs of up to 100 Mbps for mobile environments and up to 1 Gpbs in low-mobility/stationary environments. On our test SDR platform, our reference software implementation with SNOW 3G only achieves 14 Mbps per thread for confidentiality and 18 Mbps per thread for message integrity.

This paper presents hardware designs and ISA extensions for implementing SNOW 3G processing on a multi-threaded DSP SDR platform: the Sandblaster 3011. Our SNOW 3G design has the following important features: 1) the ability to operate efficiently in a multi-threaded micro-architecture, and 2) no hidden state is added to the programming model. Our ISA extensions and hardware designs should also be useful in other SDR architectures. The primary contribution of this paper is the presentation of a programmable approach to accelerating SNOW 3G that utilizes existing hardware features present in many SDR platforms. The paper also presents profile information for SNOW 3G to demonstrate which portions of the algorithm consume a majority of the execution time on an SDR platform. It also examines the performance benefits of accelerating different parts of the SNOW 3G algorithm.

The rest of this paper has the following organization. Section 2 describes the SNOW 3G algorithm and its use for confidentiality and message integrity. Section 3 details our platform architecture, simulation environment, testing methodology, and SNOW 3G performance profiling results. Section 4 presents our proposed functional units and ISA

extensions for accelerating SNOW 3G. Section 5 demonstrates the performance improvements and design characteristics of our solution. Section 6 presents conclusions.

2. BACKGROUND

SNOW 3G is a stream cipher approved for use in cellular communication systems to provide both integrity and confidentiality [5]. It uses a 128-bit key and a 128-bit initialization vector (IV). With these two entities, the cipher produces a key stream, 32 bits at a time. SNOW 3G is composed of two primary components: a finite state machine (FSM) and a linear feedback shift register (LFSR), as shown in Figure 1. The two components operate together to produce the key stream, *Z*. The FSM contains three 32-bit registers, *R1*, *R2*, and *R3*, plus two substitution boxes (or S-boxes), *S1* and *S2*. Each S-box maps one 32-bit value to another 32-bit value. The LFSR contains 16 32-bit registers, *S0* through *S15*, with taps at registers *S0*, *S2*, and *S11*. These values are combined using bitwise XOR operations to produce a new value, *v*, that is loaded into *S15*. Before the bitwise XOR operations, *S0* is processed by the function MUL_α and *S11* is processed by DIV_α. These functions are denoted α and α^{-1} in the figure. To define MUL_α and DIV_α, the SNOW 3G specification defines two additional functions, MULx(*V,c*) and MULxPOW(*V,i,c*) using C language constructs as:

MULx(*V,c*) = ((*V* & 0x80) = = 0x80) ?

$$(V << 1) \oplus c : V << 1$$

MULxPOW(*V,i,c*) = (i = = 0) ?

$$V : \text{MULx}(\text{MULxPOW}(V,i-1,c),c)$$

where *V* and *c* are 8-bit input values and *i* is an integer. With these two definitions, the SNOW 3G specification defines:

$MUL_\alpha(c)$ = (MULxPOW(*c*, 23, 0xA9) || MULxPOW(*c*, 245, 0xA9) || MULxPOW(*c*, 48, 0xA9) || MULxPOW(*c*, 239, 0xA9)).

$DIV_\alpha(c)$ = (MULxPOW(*c*, 16, 0xA9) || MULxPOW(*c*, 39, 0xA9) || MULxPOW(*c*, 6, 0xA9) || MULxPOW(*c*, 64, 0xA9))

The FSM registers are initialized to zero. The registers of the LFSR are initialized based on the *IV* and *key* as described by the standard. Figure 1 illustrates how the cipher runs in initialization mode, during which the LFSR and FSM are clocked 32 times. For each clock, the output of the FSM is a 32-bit word, *F,* which is an input to the LFSR. Next, the cipher enters key stream mode, as shown in Figure 2. In this mode, the FSM still produces *F*, but *F* is used to generate a 32-bit key stream word, *Z*. The LFSR then shifts and loads a new value into register S15. The process repeats until enough data bits are generated for the confidentiality or integrity algorithm.

For encryption, the key stream is XORed with the plaintext (data to be encrypted). For decryption, the ciphertext (encrypted data) is XORed with the key stream. The number of key bits generated matches the length of the data.

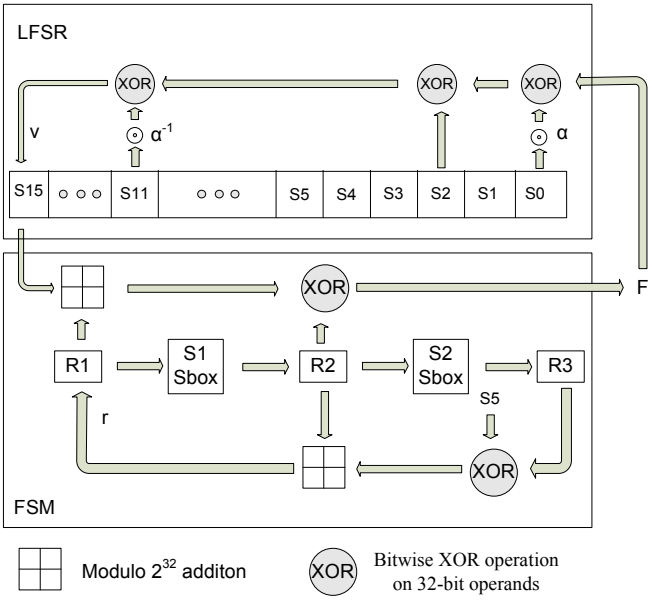

Figure 1 - SNOW 3G in Initialization Mode ([4])

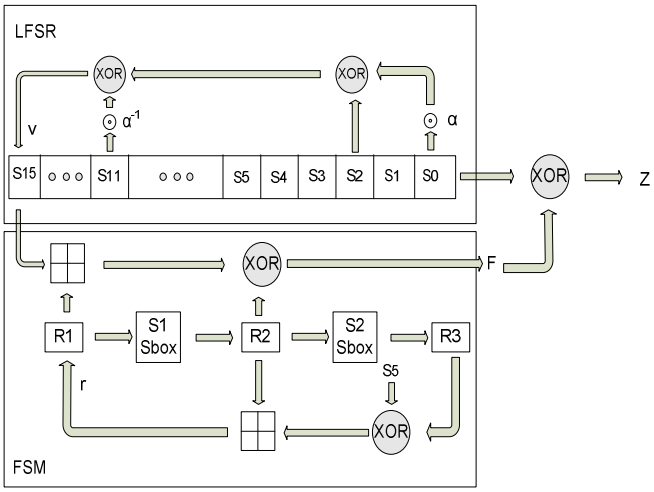

Figure 2 - SNOW 3G in Key Stream Mode ([4])

The integrity algorithm uses the SNOW 3G cipher in a different manner. It runs the cipher to produce five 32-bit words (*z1, z2, z3, z4, z5*). It pairs up the first four words into two 64-bit data values, called *P* (*z1* || *z2*) and *Q* (*z3*||*z4*), as shown in Figure 3. It also breaks up the message into 64-bit blocks and performs padding on the last 64-bit block if the entire message is not a multiple of 64 bits. An additional 64-bit block, which contains the length (*D*) of the message, is appended to the end of the message. To perform the

integrity algorithm, *D-1* blocks of the message are multiplied by the polynomial *P* inside the function *EVAL_M*. The multiplication is performed over a Galois field of 64 bits. The length field is combined (XORed) into the running data value and the result is multiplied by *Q* inside the function *MUL*, which is a single 64-bit Galois field multiply. The upper (left-most) 32 bits of the result are XORed with *OTP* (*z5*, the last 32-bit word). The upper (left-most) 32 bits of the resulting value are used as the message authentication code (MAC) for the message.

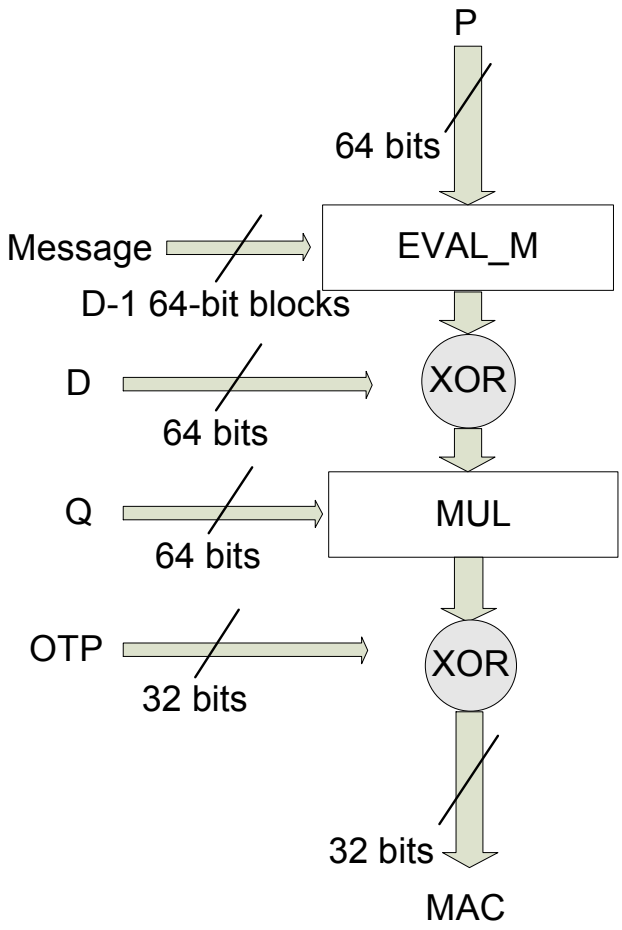

Figure 3 - Flow of the Integrity Algorithm ([6])

3. METHODOLOGY

3.1. Sandbridge Sandblaster 3011

Our test environment simulates the SB3011 SDR platform. The SB3011 platform contains four DSP cores, an ARM processor, and input and output peripherals found on many wireless handheld devices. Each core is multi-threaded and executes eight threads simultaneously [2]. Each core is partitioned into three main units: an instruction fetch and branch unit, an integer and load/store unit, and a single-instruction/multiple-data (SIMD) vector-processing unit (VPU). The SIMD VPU consists of four vector processing elements (VPEs), a shuffle unit, a reduction unit, and an accumulator register file. The VPU performs logic and arithmetic operations on 16-bit, 32-bit, and 40-bit fixed-point data types concurrently in each VPE. The VPU requires two load instructions to load 32-bit data into each VPE, while 16-bit data requires one load instruction. The instruction format allows for up to three source operands and one destination operand per instruction.

3.2. Simulation Environment

To analyze the impact of our instructions and hardware designs on performance, we use the Sandblaster toolchain [7]. This toolchain provides a full-system, cycle-accurate simulator and provides the ability for the compiler to recognize user-defined instructions supported at the ISA level. These instructions are mapped to user-defined functions inside the simulator that execute in an atomic fashion with respect to the architecture. Each user-defined instruction takes the same amount of time as a native instruction. We use the C code provided in the SNOW 3G standard, except we change recursive code to a loop-based implementation. This helps saves stack space and provides the ability to use zero-overhead loop counters and various compiler optimization techniques to improve performance.

3.3. SNOW 3G Performance Profile and Optimizations

To determine where to accelerate the SNOW 3G cipher, we profile the SNOW 3G reference implementation [4, 8] code using the Sandblaster toolchain and full-system, cycle-accurate simulator [7]. For both the confidentiality and integrity algorithms, we identify which parts of each algorithm consume a significant percentage of the execution time. SNOW 3G has some potential software optimizations to improve performance. We analyze the potential benefit from these optimizations prior to looking at hardware acceleration.

Ten 1-KB tables can be used to accelerate SNOW 3G; the MUL and DIV tables implement the MUL_α and DIV_α functionality in the LFSR, while the remaining eight tables implement the S1 and S2 S-boxes. Figure 4 shows the speed-up gained from using tables stored in level-1 (L1) data memory for the different operations. Each bar represents adding a table only for the specified function(s). Using all of the tables provides a total speed-up of almost 25x. This comes with a cost of 10 KB of static data in L1 memory. We also modify the LFSR code from the standard to use vector instructions to perform the shift of the LFSR. Even with all the tables present, plus this optimization (a 27x speed-up over the original code), the architecture only achieves a throughput of 14 Mbps per thread for confidentiality.

The integrity algorithm, called UIA2, specifies one additional method for improving message integrity performance. The optimization happens outside the main SNOW 3G processing and inside the *EVAL_M* computation. Similar to the confidentiality algorithm, called UEA2, pre-computed tables can be used to improve performance. Unlike UEA2, the tables are not static and must be generated for each set of *key* and *IV* values. To perform this optimization, an extra 16 KB of data must be generated and stored for these tables, resulting in a total of 26 KB in table data. The *pre_mul_p* function initially reads the value of *P* and sets up the eight tables. Each table has 256 64-bit entries. The new *MUL_P* function, which replaces the *EVAL_M* function, performs eight table lookups and XORs the table outputs to produce a single 64-bit quantity. Implementing this software optimization provides a speed-up of 60x for message integrity, which improves throughput to 18 Mbps per thread on our test platform. Like confidentiality, additional performance must be obtained to run SNOW 3G at next-generation cellular data rates.

SNOW 3G contains several primary functions that implement the algorithm. As discussed before, SNOW 3G consists of two modes: initialization mode and key stream mode. We extract the portions of the FSM that occur in each phase. The LFSR is implemented using two functions. One function implements the LFSR in the initialization mode, and the other function implements the LFSR in the key generation mode.

Table 1 and Table show the percentage execution time for different functions (parts) of the UEA2 and UIA2 algorithms on the SB3011 with all software optimizations utilized. The *ClockFSM* function implements the FSM. Table and Table show the execution time breakdown of the *ClockFSM* and *LFSRKeyStreamMode* functions, respectively. In the tables, "Other processing" corresponds to all processing outside the functions explicitly given in the table.

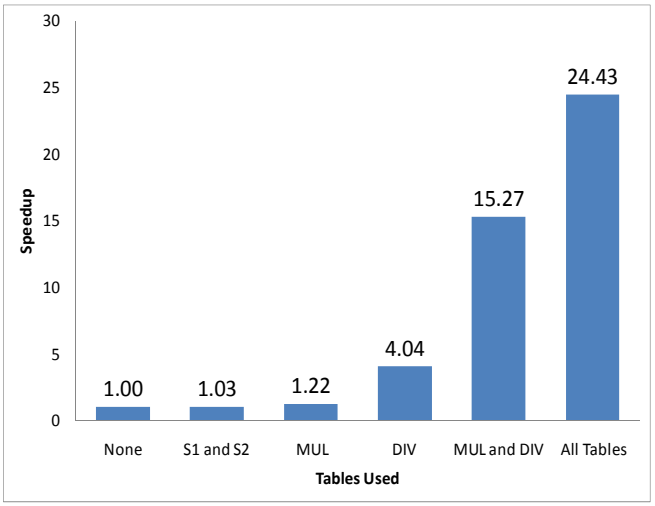

Figure 4 - UEA2 Speed-ups Due to Using Tables

Table 1 - UEA2 Percentage Execution Time Summary

Function Name	Percentage (%)
ClockFSM (KeyStream)	41.8
ClockFSM (Initialization)	11.0
LFSRInitializationMode	8.3
LFSRKeyStreamMode	30.3
Other processing	8.0

Table 2 - UIA2 Percentage Execution Time Summary

Function Name	**Percentage (%)**
MUL64	28.9
MUL_P	20.2
ClockFSM	3.8
ClockLFSRInitializationMode	2.4
ClockLFSRKeyStreamMode	0.4
pre_mul_p	38.8
Other processing	3.9

Table 3 - FSM Percentage Execution Time Summary

Function Name	Percentage (%)
S1	29.0
S2	29.0
Other processing	42.0

Table 4 - LFSR Percentage Execution Time Summary

Function Name	Percentage (%) (Initial/KeyStream)
MUL_{α}	9.6 / 10.0
DIV_{α}	9.6 / 10.0
Other processing	80.8 / 80.0

4. PROPOSED IMPLEMENTATION

4.1. New Proposed Instructions

To access the new SNOW 3G hardware, described in Section 4.2, the ISA was modified to include new SNOW 3G instructions. As stated before, our simulation environment allows for custom instructions to be supported at the ISA level using C code. For our work, we implement new VPU instructions to accelerate various aspects of the algorithm. These instructions take up to three source operands and a single destination operand.

We introduce new SNOW 3G instructions to implement the FSM and update the value *v*. Figure 5 illustrates the transformation of the register file in each VPE when implementing the SNOW 3G instructions. Here, *R1*, *R2*, and *R3* correspond to the FSM register values, while S0, S2, S5, S11, and S15 correspond to the LFSR register values. We model each 64-bit load (ld) as either loading 16 bits into each VPE or loading 32 bits into two consecutive VPEs, where ld_upper loads data into the first pair of VPEs (VPE0 and VPE1) and ld_lower loads data into the second pair of

VPEs (VPE2 and VPE3). The same is true for stores. We propose the following instructions to accelerate SNOW 3G:

- *snow3g_fsm(VRD, VRS1, VRS2, VRS3).* This instruction reads two 32-bit values from the first pair of VPE registers, specified by VRS1, and two 32-bit values from the second pair of VPE registers, specified by VRS2 and VRS3. It passes the values from VRS1 to the corresponding *S1* and *S2* transforms. It stores the resulting value in the appropriate registers specified by VRD in the first pair of VPEs. The second set of VPEs uses the values in VR2 and VR3 to implement the part of the FSM that generates a new *F* value and *r* value as $F = (S15 + R1) \oplus R2$ and $r = (R3 \oplus S5) + R2$. The resulting *F* and *r* values are stored in the appropriate registers specified by VRD in the second pair of VPEs.
- *snow3g_shuffle(VRD, VRS).* This instruction rearranges the 32-bit values in VRS and stores the result in VRD. The effective result is rotating a 128-bit register to the right by 32-bits.
- *snow3g_v(VRD, VRS1, VRS2, [1]VRS3).* This instruction produces the value $v=\alpha^{-1}(S11) \oplus S2 \oplus \alpha(S0)$, during initialization mode and $v=\alpha^{-1}(S11) \oplus S2 \oplus \alpha(S0 \oplus) F$, during key stream mode. It takes the source registers specified by VRS1 and VRS2 and extracts the appropriate values to compute *v*. When the instruction uses three operands, the last register of VRS3 in the second pair of VPEs is used as well. The instruction uses the MUL_α and DIV_α tables. The results from the table lookups and *S2* are combined to produce *v*, which is written into the first register of the first set of VPEs specified by VRD.
- *snow3g_clmul(VRD, VRS1, VRS2*). This instruction, which is not shown in Figure 5, is performed in all four VPEs in SIMD fashion. It performs a 16-bit-by-16-bit carry-less multiplication using VRS1 and VRS2 in each VPE and stores the resulting 31-bit output in VRD of the same VPE. This instruction is only used for the UIA2 algorithm.

Pseudo-code for implementing the FSM functionality and value *v* generation follows, where memory address are represented by the letters a = {R1, R2}, b= {R3, F}, c = {S15, X}, d = {S4, S5}, e = {v, S0}, and f = {S1, S2}, g = {S11, S12}. The notation {X, Y} denotes the concatenation of values X and Y, and '||' denotes an operation separator for the given VLIW instruction.

Initilizaiton Mode
ld_upper (vr1, a)
ld_lower (vr1, b)
FSM:
ld_lower (vr2, c)
ld_lower (vr3, d)
snow3g_fsm (vr1, vr1, vr2, vr3) || ld_upper (vr3, e)
snow3g_shuffle (vr1, vr1) || ld_lower (vr3, f)
ld_lower (vr2, g)
LFSR *v* generation:
snow3g_v (vr3, vr3, vr2, vr1) || st_lower (vr1)
st_lower (vr1)
goto FSM

KeyStream Mode
ld_upper (vr1, a)
ld_lower (vr1, b)
FSM:
ld_lower (vr2, c)
ld_lower (vr3, d)
snow3g_fsm (vr1, vr1, vr2, vr3) || ld_upper (vr3, e)
snow3g_shuffle (vr1, vr1) || ld_lower (vr3, f)
ld_lower (vr2, g)
LFSR *v* generation:
snow3g_v (vr3, vr3, vr2) || st_lower (vr1)
st_lower (vr1)
goto FSM

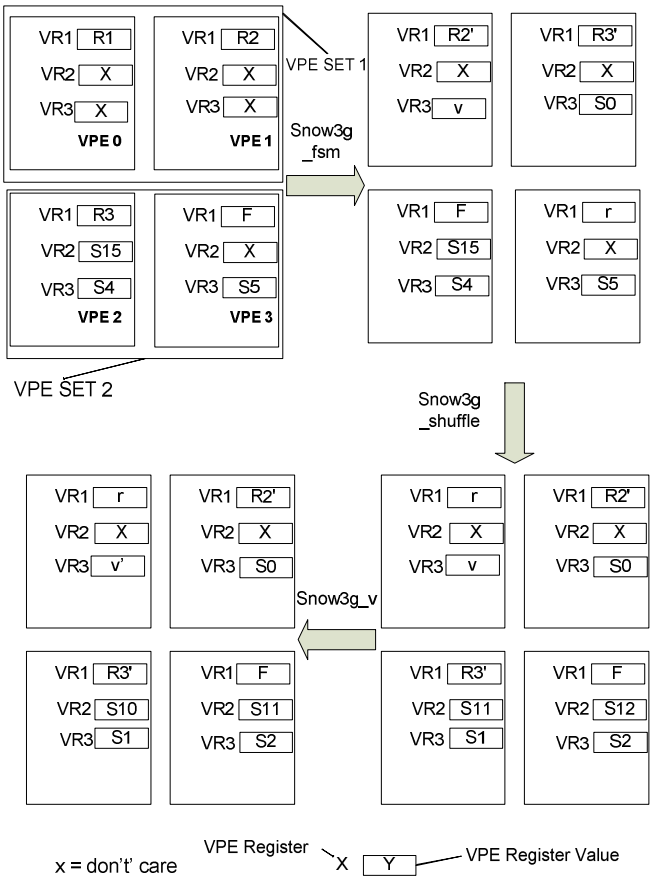

Figure 5 - FSM and *v* Generation

[1] VRS3 is optional

4.2. Execution Unit Description

Our proposed implementation is designed to reuse existing hardware. While ASIC solutions exist [9-11], a programmable solution potentially yields less area because bus interface logic, FIFOs, and additional registers -- which are present with an ASIC -- are not needed with ISA extensions. We chose to utilize the VPU to implement our new instructions. The SB3011 VPU has some benefits for placement of our new functional units. First, it has a larger load/store bandwidth than the integer unit—64 bits versus 32 bits. Second, the VPU can access more data for a given instruction—480 bits versus 64 bits. Third, the VPU can hold more state than the integer unit—1,280 bits per thread versus 256 bits per thread.

To implement the instructions, we add the hardware shown in Figure 6 to the different VPEs. The FSM uses the S1, S2, and fused addition-XOR units. The LFSR uses the MULα and DIVα tables. The MUL operation uses the 16-bit-by-16-bit carry-less multiplier (CLMUL).

5. EXPERIMENTAL RESULTS

We synthesized our proposed functional units to determine their impact on the processor hardware. The latency, area, and power of each proposed additional functional unit and the existing SB3011 vector multiply-accumulate (VMAC) unit are listed in Table . The VMAC unit performs a 16-bit-by-16-bit plus 40-bit multiply-accumulate operation. The SB3011 contains four VMAC units. Our proposed ISA extensions utilize four CLMUL units and one of each of the other units. Figure 6 depicts the throughput due to adding the ISA extensions. Compared to the optimized software with table lookups and vector shifting, the ISA extensions improve the performance of UEA2 and UEI2 by factors of roughly 4.2 and 1.5, respectively.

Table 5 - Synthesized modules at 600 MHz using a 65-nm TSMC standard cell library

Unit	Delay (ns)	Area (μm^2)	Power (μW)
S1	0.93	4,162	372.5
S2	1.00	4,068	373.9
16x16 CLMUL	1.50	2,100	570.2
Addition-XOR	1.58	453	75.2
MUL_α	0.40	126	18.5
DIV_α	0.33	123	17.0
SB3011 VMAC	1.65	6,864	1,543.1

Table 6 - Throughput of UEA2 and UIA2 algorithms

Version	Throughput (Mbps)
UEA2 (optimized software)	14.4
UEA2 (ISA extensions)	36.2
UIA2 (optimized software)	19.1
UIA2 (ISA extensions)	24.9

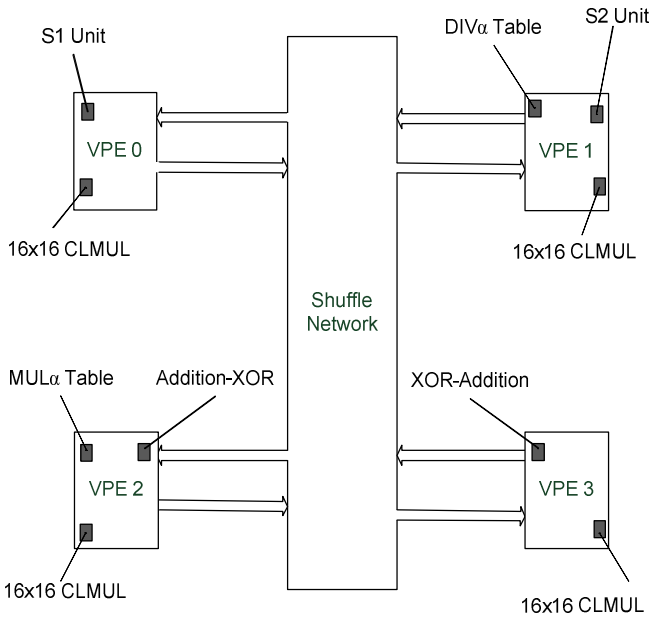

Figure 6 - New Execution Units

6. CONCLUSION AND FUTURE WORK

Our proposed ISA extensions seek to accelerate SNOW 3G by reusing existing hardware present in SDR platforms. This alleviates the area overhead of communication logic and storage elements needed by an ASIC. Our extensions use the VPU of the Sandbridge Sandblaster 3011 because it provides access to more data than the integer load/store unit in three areas: 1) loading/storing data, 2) specifying data, and 3) storage space to hold the data inside the processor. By merging the aforementioned features with new function units, our ISA extensions are portable to other SIMD-type architectures.

In conclusion, we have profiled and demonstrated a method to accelerate SNOW 3G on a SDR platform. We maintained the programming model and micro-architecture of our SDR platform in the process. To reach more than 50 Mbps per thread for confidentiality and integrity, additional instructions, assembly optimization, and hardware may be needed.

7. REFERENCES

[1] M. Mehta, N. Drew, G. Vardoulias, N. Greco, and C. Niedermeier, "Reconfigurable terminals: an overview of architectural solutions," IEEE Communications Magazine, vol. 39, pp. 82-89, 2001.

[2] S. Mamidi, E.R. Blem, M.J. Schulte, J. Glossner, D. Iancu, A. Iancu, M. Moudgill, and S. Jinturkar, "Instruction set extensions for software defined radio on a multithreaded processor," in Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2005, pg. 273.

[3] 3rd Generation Partnership Project. 3GPP confidentiality and integrity algorithms. 2010 (June/5).

[4] ETSI/SAGE (2006, September 6, 2006). Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 2: SNOW 3G specification. Available: http://www.gsmworld.com/documents/snow_3g_spec.pdf.

[5] ETSI/SAGE (2006, September 6, 2006). Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 5: Design and evaluation report.

[6] ETSI/SAGE (2006, September 6, 2006). Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 1: UEA2 and UIA2 specification.

[7] J. Glossner, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis. (2005, "Sandbridge software tools," in Embedded Computer Systems: Architectures, Modeling, and Simulation).

[8] ETSI/SAGE. (2006, September 6, 2006). Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 4: Design conformance test data. Available: http://www.gsmworld.com/documents/snow_3g_spec.pdf.

[9] Elliptics. CLP-41: SNOW 3G cipher core. 2010 (June/5), Available: http://www.elliptictech.com/products-clp-41.php.

[10] IP Cores. SNOW 3G LTE cipher. 2010 (June/5), Available: http://www.ipcores.com/Snow3G.htm.

[11] P. Kitsos, G. Selimis, and O. Koufopavlou. “A high performance ASIC implementation of the SNOW 3G stream cipher,” presented at 16th International Conference on very Large Scale Integration (VLSI-SoC 2008).