CryptoChip: A Modern RISCV Processor with a Crypto Accelerator

Kylee Krzanich
Dept. of Electrical Engineering
Stanford University
krzanich@stanford.edu

Sam Xu
Dept. of Electrical Engineering
Stanford University
samx@stanford.edu

Abstract—**In a world where security has become more and more important, our project focused on developing a crypto-accelerator for a modern RISC-V processor. We showed any-where between a 17-66x increase in performance for all cryptography transactions including but not limited to random number generation and encrypting. Our project is completely open source and the details can be found on our github https://github.com/krsandwich/EE272B.**

I. Introduction

The rapid expansion of crytographic operations and encryption services show no sign of stopping. Crytography has become an essential part of any consumer or enterprise system for both personal and business data protection. While the vast majority of consumer computers are capable of providing crytographic acceleration for the data of a single user through the use a general purpose processor, this comes at a cost of speed and power efficiency, and the speed is wholely insufficient for systems servicing many consumers at once.
The purpose of our cryptographic accelerator is to integrate specifically designed hardware that can improve the speed and efficiency of software solutions. We design accelerators for true random number generation and the Advanced Encryption Standard. Finally, to test our accelerators in a real world work environment, we attach them as tighly-coupled co-processors to a general purpose, RISCV processor generated by Chipyard.

II. Related Works

Crypto co-processors are not a new concept. Soliman et al., describes a high-throughput AES co-processor implemented on an FPGA. They were able to achieve up to 222 Gb/s throughput when running four accelerators in parallel with a 128-bit wide L2 cache [5]. We took a similar approach to Dur-e-Shahwar et al. in choosing to include an AES accelerator, hashing accelerator, as well as a random number generator [7].

Random number generation in general purpose processors is also not a new concept. Chip manufacturing companies such as Intel have included random number generators in their chips [8]. The Intel TRNG uses unpredictable analog noise such as junction or thermal noise. In their design, the chip samples the thermal noise of undriven resistors by amplifying the voltage across it; however, this technique are associated with other factors such as temperature and power supply fluctuations. The intel design minimize this by using two free-running oscillators such that the thermal noise source is used to modulate the frequency of the fast clock of the smaller oscillator. The drift between the two measurements becomes the source of random binary numbers.
As chip makers continue to make transistors fit in a smaller area, it has become more desirable to make a digitally based true random number generator. The analog components in Intel's design is undesirable as it consumes a lot of power and area.

III. Chipyard

To reduce development time, we use a framework called Chipyard, a series of open-sourced tools and RTL generators to develop heterogeneous SoC designs and provide a simulation framework. Chipyard helps alleviate many of the challenges that exist of using independent and uncoordinated open-source tools and designs, and provide a SoC framework to which we could attach our custom designed accelerators.

A. Rocket Chip

The main front-end of the chipyard framework is the Rocket Chip SoC generator. Rocket Chip uses a Chisel-based parameterized hardware generator. Rocket-chip along with Chipyard also alllow IP blocks written in Verilog to be inculded via a Chisel-wrapper, which we utilizes to attach our accelerator. To generate the final RTL from the Scala and Chisel based parameters, Chipyard generates a intermediate-representation of the RTL, before it outputs the generated source RTL which we export to external tools, such as Mflowgen and Caravel. Furthermore, Rocket-chip also allows the integration of existing IP-blocks at various levels including the memory mapped IO peripheries as tightly integrated accelerators. We use this to attach several serial communication IPs such as UART and GPIO to our device for debug post-tapeout. Additional peripheries includes: SPI-flash, I2C, GPIO, SPI, and PWM,

B. Configuration

Our final Rocket-chip configuration consists of a single Rocketchip-Core with a L1 data and instruction cache, floating-point capabilities, and scratch space. The L1 cache consists of an 4-kilobyte L1 data and 512-byte instruction cache. Each block-size is 64-bytes. The L1-data consists of 32 sets and 2 ways while the L1-instruction consists of 8 sets and 1 ways. Finally, due to a issue with the memory density, we could only afford up to 4-KB of scratch space.

Attached to the chip are three accelerators using the tightly-coupled RoCC interface - Random Number generator, AES Accelerator, and SHA3 accelerator.

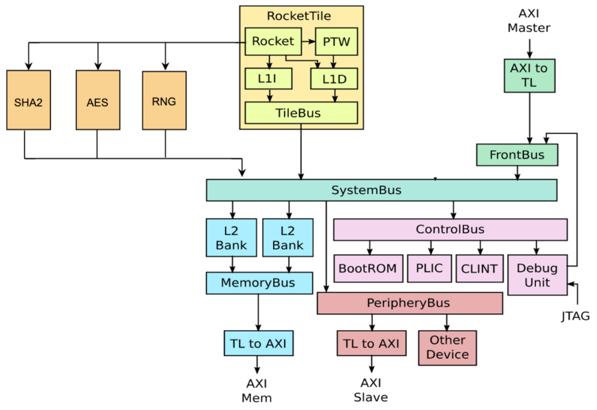

Fig. 1. Block diagram of our Rocket-chip (chiptop)

C. Simulation

Simulation of the Rocket-chip and the accelerators is done using VCS and Risc-V binaries generated using cross compilation tools and C-code. Because of memory constraints, we are forced to use a exceedingly small cache and scratch pad space, as a result, to conserve memory so we could run programs, we needed to write a custom linker script for the compiler tools such that any programs we load for simulation are loaded through the read-only memory-mapped SPI-flash module, and any data written by the programs, such as the heap and the stack, and loaded into the scratch space, which is present in a different memory address.
In other words, the process of compiling a program for simulation (and testing after tapeout) consists of the follows:

1) Generate risc-v relocatable object files from c-code using RISC-v preprocesser, parser, assembler and assembler.
2) Link the relocatable object files into the appropriate memory addresses for our processor
3) Generate a binary dump of the machine code and load it as a SPI-flash memory file

Additionally, to read the output of our programs, we cannot utilize the printf capabilities of the standard libraries. Instead, we need to write each character out one at a time to UART in order to generate a message.

IV. True Random Number Generator

True random number generators (TRNG) are a important security primitive that can be used for various essential tasks including the generation of secret and/or public keys, initializing the seed and state for cryptographic operations, nounces, games and simulations. TRNGs are such a important primitive, a brute-force attack only on the RNG could break any cryptographic system. Additionally, applications that demand a constant high-speed and quality generations of keys, such as secure web servers and encryption servers, an algorithmic approach of pseudo-random number generation is often insufficient. Therefore, we attached a TRNG module in our Cryptographic accelerator.

A TRNG extract randomness from some underlying physial phenomena that exhibits uncertainty or unpredictability. Examples of this includes atomic decay, thermal noise, and secondary effects such as clock-jitter and metastability in circuits. Our design will attempt at TRNG using only digital logic and does not rely on external analog components.

A. Ring Oscillators

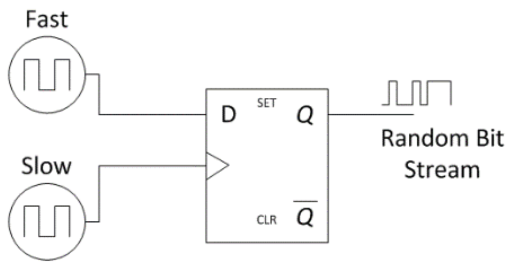

Fig. 2. Example of sampling for metastability

While pure digital circuits cannot derive randomness from noise by their very nature, it is possible to extract randomness from digital circuits in the time domain if we use asynchronous logic circuits. One simple technique to do so is by building a ring oscillator. A odd number of inverters connected in a loop will oscillate with a varied frequency. The frequency of the output of the ring oscillator is determined by number of inverters: n, the gate propagation delay and routing delays: τ.

$$f = \frac{1}{2n\tau}$$

The most common methods to extract randomness from a ring oscillator is sampling it asynchronously using an flip-flop. Whereas clock jitter and metastability is undesired to traditional digital design, it is desired here as the source of randomness. The clock jitter property of a free running ring oscillator that has been running for a long time is considered to be a good source of randomness [9]. Similarly, if the ring oscillator is running at a much higher frequency than the clock, a flip-flop sampling the ring at the rising edge of the clock will take place in a narrow time window of a signal transition. A race condition occurs during the setup/hold time. The probability of the bit settling on a 0 or a 1 is modeled by the gaussian:

$$Q(x) = \int_x^{\inf} \exp(\frac{-u^2}{2})du$$

where $x = \frac{\Delta}{\sigma}$ where σ is proportional to the width of the setup/hold time window, and Δ is the delay in the difference of the arriving signals. Ideally we would want $\text{Prob}(0) = \text{Prob}(1) = 0.5$, however for this project, we will not spent too much analysis on this aspect since we do not have a fine control over the technology process.

B. Linear Feedback Shift Register

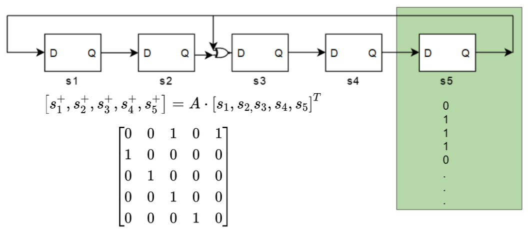
D Q
s1
s2
s3
s4
s5
$[s_1^+, s_2^+, s_3^+, s_4^+, s_5^+] = A \cdot [s_1, s_2, s_3, s_4, s_5]^T$
$\begin{bmatrix}0&0&1&0&1\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}$
0 1 1 1 0 . . .

Fig. 3. Examples of a Linear-Feedback Shift-regitsre and its transition matrix

A linear feedback shift register is a shift register where each bit is a linear function of some previous bits. Commonly the XOR function is often used for random number generation; as a result, a LFSR is often a combination of XOR functions of some bits over the shift register. LFSRs are often used to make random number generators in hardware as they are simple to build, and are mathematically robust enough to build complex logic with simple parts.

In figure 3, we see a example of a basic 5 bit LFSR with a single XOR gate as it's linear function. Often some bit over a LFSR is chosen as the output stream. Every clock, the shift register updates, and the output bit will produce a seemingly random stream of bits. However, this stream is finite in length, in other words, after some time, the output stream will repeat in a cycle. The length of the cycle will depend on the size of the shift register and the choice of linear functions. However, it can be mathmatically shown that the maximum possible cycle length of a size n shift register is $2^n - 1$. We will refer to a shift register that outputs a maximum length cycle of bits a maximal shift register.
We can design a maximal shift register by controlling our linear function. By using only XORs as our linear function, we can see that our LFSR updates equivalent to a matrix-multiplication with some transition matrix as shown in figure 3. By finding the characteristic polynomial of this transition matrix, we can determine if the resulting LFSR will be maximal. For example, the resulting characteristic polynomi of the transition matrix in figure 3 is

$$x^5 + x^3 + 1$$

Which is irreducible/unfactorable, which means that this LFSR has a maximal length. If the polynomial is factorable/reducible, then the LFSR will have some length from 1 to $2^n - 2$. Ideally, we want to design a LFSR with a maximal cycle length.
One disadvantage of using a LFSR is that it only generate one bit at a time. To generate a random 32 bit number, it will be unideal to simply replicate the LFSR 32 times as that is both expensive in terms of area, but also power as the ring oscillators tend to run at a high frequency. Furthermore, a LFSR is deterministic, as the initial state determines the all the values that will appear in the cycle.

C. Linear Hybrid Cellular Automata

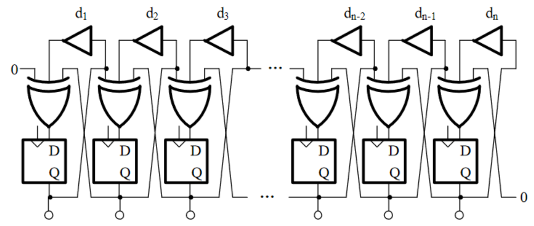

Fig. 4. Our Linear-Hybrid Cellular Automata Design

To address the first problem, instead of simply using a single bit of the shift register as a output, we output the entire shift register or portions of the shift register. Secondly, the state will continously be seeded by the ring oscillators using a XOR functions. We design a generalized LFSR called a Linear-Hybrid Cellular Automata (LHCA). Where each bit is a linear function (XOR) restricted to its neighbors. Using only two types of update rules based on the neighbors of each cell, we can create a maximal LFSR as follows:

$$\text{Rule 90:} s_i^+ = s_{i-1} + s_{i+1}$$

$$\text{Rule 150:} s_i^+ = s_{i-1} + s_i + s_{i+1}$$

Using the LHCA, we come up with the design shown in figure 4. This final design gives us a compact, and efficient TRNG that can generate 32 bits of random numbers per clock cycle, and can give uniform and uncorrelated results thanks to the ring oscillators.

D. Simulation and Statistical Testing

To test our accelerator, we attached it as a tightly-coupled RoCC processor to our Rocket-chip. This means that our TRNG can write a newly randomly generated 32 bit number to the return register every clock by using a custom assembly instruction that can be invoked through a custom syscall that we wrote. To test that our accelerator was actually outputting random values, we ran several statistical tests to test that the values were uniform and uncorrelated. This included Chi-Square and Kolmogrov-Smirnov.

V. Encryption

In modern processors, the ability to quickly encrypt and decrypt means that more applications can afford to implement security protocols without sacrificing performance. At a high level this includes everything from confidentially storing and transmitting data, user authentication, and integrity verification for received data.
One example of this is the IBM crypto co-processor which is used for more commerce related applications. The main

```
#define ROCC_INSTRUCTION_D(X, rd, funct) \
        ROCC_INSTRUCTION_R_I_I(X, rd, 0, 0, funct,
↪   10)

#define ROCC_INSTRUCTION_R_I_I(X, rd, rs1, rs2,
↪   funct, rd_n) {              \
    register uint64_t rd_  asm ("x" # rd_n);
↪   \
    asm volatile (
↪   \
        ".word " STR(CUSTOMX(X, 1, 0, 0, rd_n, rs1,
↪   rs2, funct)) "\n\t"  \
        : "=r" (rd_));
↪   \
    rd = rd_;
↪   \
  }

static inline unsigned long rng_get(){
        unsigned long value;
    ROCC_INSTRUCTION_D(1, value, 0);
    return value;
}
```

Fig. 5. Assembly instruction for random number generation

objective was to free up the CPU because crypto transactions during secure socket layer sessions were computationally intensive. It also has all 4 of the main encryption algorithms even though DES and 3DES were rendered virtually obsolete [2]. The second example is the Apple Secure Enclave which is a co-processor built into the iPhone SOC and provides memory encryption and has a hardware random number generator both of which we included in our own accelerator [3].
We decided on the advanced encryption standard because it is one of the most rigorous cryptography algorithms in widespread use today. It is the successor to the Data Encryption Standard (DES) and it is FIPS complaint, meaning that it is one of the few algorithms approved for government use [6]. Within bitcoin, AES is used for wallet encryption on a randomly generated master key.

A. Accelerator Integration

In the case of AES, we only really needed to provide 3 instructions: loadkey, encrypt and decrypt. Fig. 7 shows a c code version of how to call the encrypt command. Fig. 8 shows how a the data is stored into memory before the call to load the key. The loadkey command will perform four memory requests, each 64-bits wide, to get the key from memory. Then it will run the key expansion routine and store all 15 generated keys to be used for as many encrypt or decrypt operations as possible. When the encrypt command is called, the data that needs to be encrypted is stored in the read memory address and the encrypted output is written to write address. The processor sends the command to the accelerator via the tilelink bus. We have the accelerator then do two memory reads to retrieve the data and then input that data into our encryption algorithm, and once it finishes tilelink performs another two memory requests

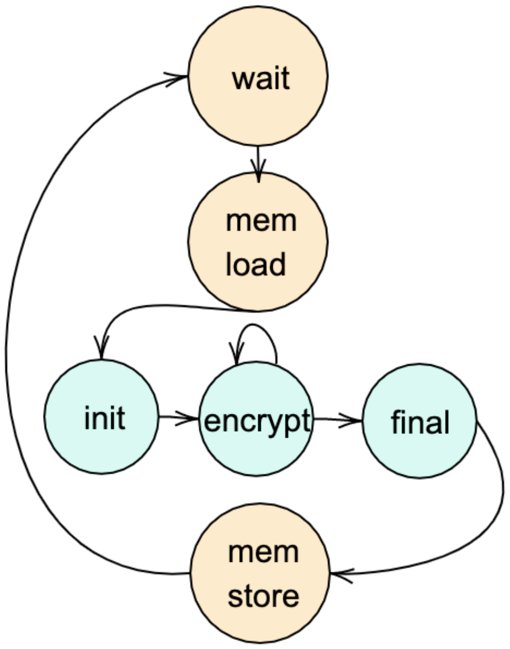

Fig. 6. FSM for the AES Encrypt command

to store the encrypted data before signaling to the CPU that the instruction has finished.

```
static inline void aes_encrypt(void *read_addr, void
↪   *write_addr)
{
        asm volatile ("fence");
        ROCC_INSTRUCTION_SS(0, (uintptr_t)read_addr,
        ↪   (uintptr_t)write_addr, 2)
}
```

Fig. 7. Function for calling custom accelerator ’encrypt’ command

```
uint8_t data [128] __attribute__ ((aligned (8)));
uint8_t out [128] __attribute__ ((aligned (8)));
for (i = 0; i < 8; i++) {
   *data++ = 0x12;
}
aes_encrypt(data, out);
```

Fig. 8. Example code for using the encrypt command

B. AES Design

AES is implemented using the Rijndael algorithm which uses a symmetric block cipher and operates on 128 bits of data at a time. In Fig. 9, the highlighted sections labeled encrypt and decrypt are configurable so for AES-256 we would need to use an 8 word key and have 14 encryption and decryption rounds.
We used the FIPS-197 publication as a mathematical reference for how the algorithm works. Designing the RTL around the algorithm was relatively straightforward. For example, the AddRoundKey stage is simply an XOR operation between one of the 15 generated keys and the current data [6]. Due to

timing issues with our memories, we were limited to a 20ns clock period which meant that it only made sense to pipeline each rounds instead of of each stage.
Our accelerator assumes correct usage of the memory systems. This means that we rely on the TileLink bus to fault in cases where a user tries to give the accelerator a restricted memory address

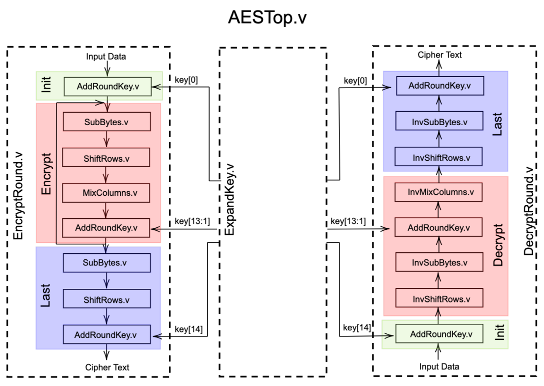

Fig. 9. file and dataflow structure for AES algorithm

C. Performance

To analyze the performance of our AES accelerator, we used TinyAES which is a very lightweight, portable c-program that we used as our baseline gold model. The tests were designed to average the cycle time over 100 consecutive runs, each with different inputs. We were trying to assess both the relative performance as well as get an idea of how the accelerator performs with a constant load. Even with the overhead of the memory transactions, we were able to show that our accelerator provided a 66x increase in performance over TinyAES on the basis of cycle count. After we finish working through some of our physical design issues, we plan on doing a power consumption comparison.

One future design optimization that we think would greatly increase performance is to provide a command that allows for batch encryption and decryption. As of now a user can only operate on 128 bits of data at a time. As of now, memory transactions take about 12 cycles and the AES algorithm takes about 17 cycles which gives us a total of 30 cycles of data processing. You can see from Fig. 11 that writing the data to memory as well communicating with the accelerator take a total of 170 cycles meaning that 85% of the time is spent outside of the accelerator. We believe that we can amortize that cost by doing more bulk transactions.

D. Security

Although our accelerator's job is to encrypt data for security purposes, we also had to ensure that the accelerator itself was secure. Many applications these days are vulnerable to timing attacks. Timing attacks are side-channel attacks in which the attacker can infer privileged information from the timing of the program. Within AES, there's a step that involved substituting certain bytes within the data from a fixed size array. One of the easiest side-channel attacks on this step is using the array look up time to deduce the index which would then tell the attacker what the original data was [4]. We wanted to be incredibly careful in ensuring that all aspects of our accelerator including the memory requests and other data transactions were completely immune from timing attacks. We started by ensuring that each command finished in a constant number of cycles regardless of the input. Below, Fig. 10 shows the timing of TinyAES and as you can see that as we sweep across a range of different inputs, the total cycle time could vary by hundreds of cycles. Fig. 11 shows the cycle times of our AES accelerator across a range of inputs. You can see there's a flat cycle time for everything but the first command. Looking at the waveforms we realized that the initial increase in cycles was just from the overhead of initializing the bus and memory subsystems.

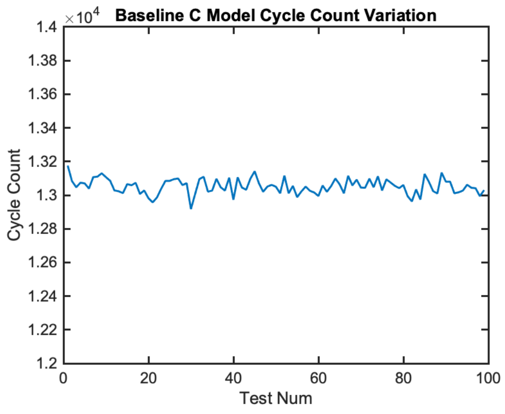

Fig. 10. file and dataflow structure for AES algorithm

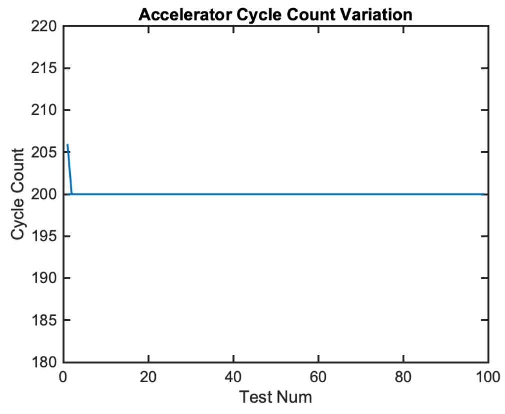

Fig. 11. file and dataflow structure for AES algorithm

VI. SHA3

The secure hashing algorithm is another important cryptographic function. Currently sha3 is already implemented in Chipyard as an example accelerator. We attach this to our

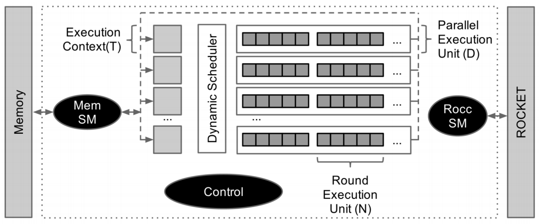

Fig. 12. Block diagram of Chipyard Sha3 Module

design for completeness sake since it doesn't take up much resources during physical design.
More specifically, SHA3 belongs to a class of hashing algorithms that is easy to compute and implement in hardware, inability to generate the message from the hash , inability to change the message and not the hash, and inability to find two messages with the same hash. Similar to our AES module, the SHA3 module has the ability to transfer two 64bit requests, the return value and a funct field for the specific function requested. The accelerator receives these requests using a ready/valid interface. Once the operation is completed, it is written back to memory using a simple state machine.

VII. Physical Design

Originally, the project was started assuming a much higher memory density. However, after starting the openlane flow, we realized the memory density of the technology was lower than expected at around only 9000 bits/mm^2. Given a total area of $10mm^2$, this only gives us no more than 4KB of cache, tlb, and scratch space memory given the rest of our chip. Thankfully, we were informed by people from efabless that they had been working on a memory compiler with pre-generated macros for memory cells with a much higher density at around 26,000 bits/mm^2. We show am example of a memory macro used in our design below in figure 13. From the compiled macro files, we were able to generate db files to also include in our flow.

We used mflowgen, a modular flow generator for ASIC/FPGA design-space exploration, to run our designs through the physical flow. As of now we have successfully run a previous configuration through the timing signoff stage with no violations. We are in the midst of resolving drc violations within the memory macros but we expect to resolve these quickly with the help of the efablesss team. 14 shows our current memory macro placement. You can also see from the diagram that our design is low density. Each of our accelerators are less than 0.4mm^2 in total area which met our target area for the total chip. As previously mentioned, we had to use a clock period of 20ns because of limitations the memory macros but we expect that, as the memory technology improves, we will be able to decrease the clock period considerably to achieve our desired 200MHz frequency.

VIII. Conclusion and Future Work

In this paper, we presented a Cypto co-processor for a RISC-V CPU that gives 17-66x speedup on all desired applications. The proof of concept design is built on top of Chipyard which is the best configurable, open-source SOC on the market. We believe that more and more programs will choose to encrypt information to protect the user's data. Our accelerators were designed to free the CPU from these intensive encryption algorithms. We would love to continue the design space exploration after tapeout. This would include trying out different techniques for our random number generator and adding bulk transactions to our AES module as described in section V, subsection C. For now we will focus on pushing our entire design through the rest of the physical flow including LVS and resolving any remaining DRC issues.

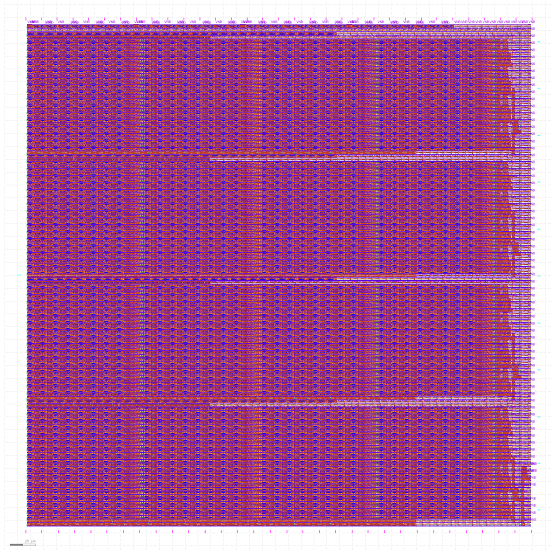

Fig. 13. Layout of the 4kB DFFRAM Macro we used

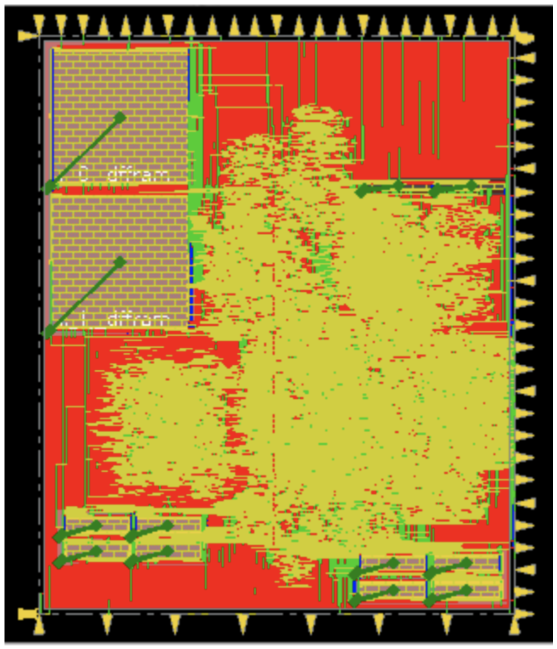

Fig. 14. Current layout

ACKNOWLEDGMENT

We would like to give a special thank you to our amazing TA Kartik. He has provided us with so much guidance and support and we are forever grateful for his expertise as well as his dedication to helping us get to the finish line. We would also like to thank the engineers at Apple for sponsoring this class so that we could participate in a tapeout.

REFERENCES

[1] Rijmen, Vincent, and Joan Daemen. ”Advanced encryption standard.” Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology (2001): 19-22.

[2] Arnold, Todd W., and Leendert P. Van Doorn. ”The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer.” IBM Journal of Research and Development 48.3.4 (2004): 475-487.

[3] Mandt, Tarjei, Mathew Solnik, and David Wang. ”Demystifying the secure enclave processor.” Black Hat Las Vegas (2016).

[4] Bernstein, Daniel J. ”Cache-timing attacks on AES.” (2005): 3.

[5] Soliman, Mostafa I., and Ghada Y. Abozaid. ”FPGA implementation and performance evaluation of a high throughput crypto coprocessor.” Journal of Parallel and Distributed Computing 71.8 (2011): 1075-1084.

[6] https://github.com/kokke/tiny-AES-c

[7] Khalid, Ayesha, et al. ”Resource-Shared Crypto-Coprocessor of AES Enc/Dec With SHA-3.” IEEE Transactions on Circuits and Systems I: Regular Papers 67.12 (2020): 4869-4882.

[8] Benjamin Jun and Paul Kocher, “The Intel Random Number Generator”, Cryptography Research, Inc. White Paper Prepared for Intel Corporation, April 22, 1999

[9] Mitchum, S and R. H. Klenke, “FPGA Based Digital True Random Number Generator”, 2009