ATLAS - ARX Transformation Logic Acceleration System

ECE 427 Project Proposal

Datis Hoghooghi
datismh2@illinois.edu
Shaun Lee
shl14@illinois.edu

Everitt Cheng
emcheng2@illinois.edu
Srijan Singh
srijans3@illinois.edu

Nikhil Bhalla
bhalla4@illinois.edu
Alex Chen
alex18@illinois.edu

Abstract – This work presents a unified hardware accelerator for custom data transformations. Instead of using dedicated datapaths per algorithm, this accelerator routes self-describing instructions through a shared pool of basic ARX functional units. The design includes a stretch goal of a mixed-signal TRNG for enhanced security purposes.

I. Problem Statement

Modern systems rely heavily on cryptographic and data transformation operations, which has increased the demand for efficient hardware acceleration. Most existing accelerators are designed for specific algorithms (such as AES or SHA-256), which makes them very efficient for those tasks but inflexible, wasting silicon area if the algorithm is not in use. Most of these algorithms share the same ARX operations (ADD, ROTATE, XOR, AND), and these commonalities can be exploited for a more efficient chip.

II. Architecture

We propose a hardware design that can act as a unified data transformation accelerator, with self routing dataflow headers. The goal is to efficiently alter or encode data according to multiple general data transformation algorithms while sharing functional units across them. Furthermore, unlike a microprocessor, we avoid the need for complex decodes, hazard management, and branch management. This proposal decomposes each algorithm into its needed functional units with heavy reuse in between, as well as new ARX algorithms being absorbed with near zero additional hardware cost.

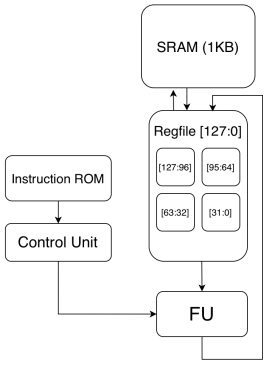

Fig. 1. Dataflow of single processing unit

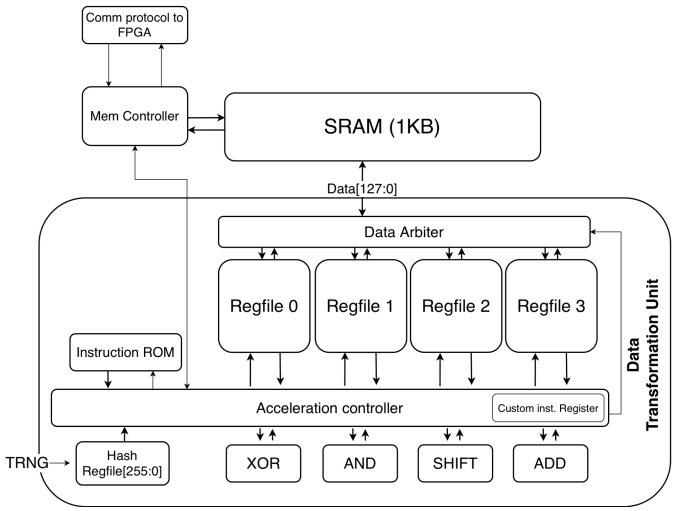

Fig 2. Block Diagram of Dataflow Architecture

To fit the design onto the chip, we have opted for an iterative architecture. It reuses the same hardware across successive cycles as dictated by the instruction stream. The Data Acceleration Controller sequences which unit each operand visits each cycle, effectively time-multiplexing a set of functional units to execute arbitrarily long transformation algorithms.

As a baseline, we plan on implementing ChaCha20, SHA-256, CRC-32, and HMAC (subset of SHA) natively on device, stored as micro instructions flashed onto on-chip ROM for area improvement over traditional SRAM. Custom algorithms will be dynamically uploaded by the user, and stored in SRAM.

The execution of these algorithms is computed through a shared pool of general purpose functional units that can do basic ARX computations based on needed parameters of the algorithm supplemented by special instructions. We also plan on allowing the user to input instructions stored in SRAM, giving way for them to accelerate custom data transformation

algorithms. Rather than a dedicated datapath per algorithm, we plan on routing operands through these shared units using self-describing dataflow instructions, removing per algorithm control logic.

As a stretch goal, we aim to strengthen security by integrating hash function acceleration alongside a True Random Number Generator (TRNG) in a mixed-signal design.

A. Data Acceleration Controller

This controller unit serves as the brain of the accelerator. Its primary responsibilities include:

- Reading and decoding instructions
- Tracking which architectural registers contain valid data
- Dynamic scheduling of which FU the data should be routed to, based on the instructions
- Requesting new custom instructions, or new data from Memory Controller
- Requesting to send finished data back to the memory controller
- Has hardcoded ranges of memory locations for user data and instructions.

Depending on the operation being done, the data acceleration controller will either read instructions out of the natively supported ROM, or read the custom user instructions out of the SRAM. The instruction format is as follows:

Instruction Format

We will support different instruction types for different operations. Within the register mapping (4 bits), the top two bits represent indexing into the 128 bit registers. The bottom 2 bits represent indexing into the 32-bit registers within the larger registers. Bit 3 represents if we would like to use 128-bit or 32-bit registers.

R-Type: Add, Rotate, XOR

RD[15:12]	RS1[11:8]	RS2[7:4]	128/32[3]	Opcode[2:0]

Opcodes: 000, 001, 010
000 - AND
001 - ADD
010 - XOR

S-Type: Shift

RD[15:12]	RS1[11:8]	DIR[7]	IMM[6:4]	128/32[3]	Opcode[2:0]

Opcodes: 100, 101
100 - Shift
101 - Circular Shift

shift directions encoded within IMM, top bit being left/right, bottom 3 are the amount (max 8 shift).

J-Type: Jump

Jamt[15:13]	RS1[11:8]	RS2[7:4]	128/32[3]	Opcode[2:0]

Opcodes: 110

T-Type: TRNG, Move, Commit

RD[15:12]	RS1[11:8]	OP[7:5]	Res.[4]	128/32[3]	Opcode[2:0]

Opcodes: 111
Operations [7] - Writeback RD to SRAM, KILL
Operations[6] - Writeback RD to SRAM
Operations[5] - RD = TRNG
Operations[5 & 6] - RD <= RS1, RS1 <= RD

K-Type: Repeat-K

Reserved[15:12]	N[11:8]	K[7:3]	Opcode[2:0]

Opcodes: 011
Repeat the next K instructions N times

B. Shared Functional Units & Registers

Rather than pipelining the entire algorithms, we use an iterative architecture with a shared pool of FUs:

- XOR
- ADD
- SHIFT
- AND

It should be noted that our shift unit will not be designed as a barrel shifter due to area constraints. Our design choice is to use a multi-cycle shifter, capable of shifting in either direction by powers of 2, up to 8 (1, 2, 4, 8). The user must consider formatting shifts greater than 8 into multiple instructions due to ISA limitations. If the user requests a shift that is not a power of 2, we will use an FSM to decompose this shift into multiple cycles.

If we have extra area, a potential feature with massive benefits would be a byte level shifter using muxes (to implement byte-level rotation). Unlike a full barrel shifter, we would split up our shift amounts to byte level shifts and then more granular shifts handled by the multi-cycle shifter. In rotation mode, bits that shift off one end re-enter at the opposite end, preserving all data. In logical shift mode, zeros fill the vacated positions instead, and the shifted-out bits are discarded. This is how we would support circular and regular shifts in order to support the SHA algorithm (and other encryption algorithms like it).

We will have four 128-bit registers, which are able to be partitioned into 32-bit registers as well. This allows flexible data transformations on either 32-bit or 128-bit data. The

routing of data to functional units will be done by the data acceleration control unit.

C. Memory System

For our design, we will use a dual-port 1KB SRAM with 128-bit read and write capabilities. We will logically split this SRAM by address to store 3 different parts:

- User programs (list of instructions, up to 4 user programs, each 128 to 512 bits)
- User data waiting to be operated on (128 to 512 bits)
- User data waiting to be sent back (128 to 512 bits)

We will have a complex memory controller that will control and arbitrate reads/writes to the SRAM. The memory controller will do the following:

- Handle data/instruction requests from the data acceleration controller and send data/instructions back
- Receive data writeback requests from the controller, and store the data back into SRAM
- Keep track of how many pieces of data are stored in the SRAM, where they are stored, and what set of operations must be done on them
- Handle read-in requests from the I/O protocol
- Drive write-out requests to the I/O protocol

D. I/O Subsystem

To handle I/O requests, we plan to use a "valid ready" handshaking protocol, along with a 16-bit read and write bus. For data_in requests, once the valid ready handshake happens, the first piece of data will be a 16-bit header, shown below, classifying what type of data is coming in (the memory controller will utilize this header to understand what to do with the data). After that, the real data/instructions will come into the system.

Header Packet For I/O

[0]	0 = Data_in, 1 = Instr_in
[2:1]	Size of data (Group of 128 bits)
[5:3]	Operation, bit 0-1 is preset, bit 2 is custom
[7:6]	Custom instruction slot (up to 4)
[9:8]	Max number of registers used at once by instrs
[15:8]	Reserved

Outputs will work similarly to inputs, with the accelerator device being the master, and the output device being the slave.

The rest of our pins will be dedicated to clock, power, ground, reset, and other standard pins. A list of the pin assignments are shown below:

Pin #	Assignment
[15:0]	Data_in bus
[31:16]	Data_out bus
[32]	Read_valid
[33]	Read_ready
[34]	Write_valid
[35]	Write_ready
[51:36]	Power, GND, RST, CLK, other standard/debug pins

E. TRNG Subsystem

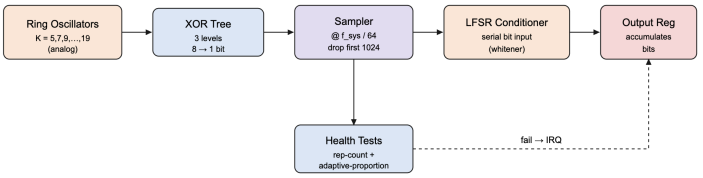

Fig 3. TRNG

We plan on referring to a paper [3] for the ring oscillator design.

The entropy source is the TRNG's analog block; everything downstream is standard cell digital. The entropy source comes from 8 ring oscillators with differing inverter counts from {5, 7, 9, 11, 13, 15, 17, 19}, in which we will have varying sizes to increase randomness. Per-sample entropy decreases as we increase the length of the oscillators, so they will be relatively short. The differing inverter counts are meant to discourage inter-ring locking, which is the synchronization of their oscillations as a result of a shared power supply and substrate, in turn reducing our randomness. Thus, the variable inverter sizing allows for oscillations of varying frequencies creating differing periods. This is important to our randomness as the relative phase between the rings is constantly shifting. Moreover, ≥20 µm physical separation between each oscillator to prevent oscillation coupling is needed.

We are designing ring oscillators with custom inverters so we can double the length to decrease transconductance, and in turn increase thermal noise for variability.

A 3-level custom layout XOR tree reduces the 8 outputs to one bit, sampled at the system clock/64 (through a counter), this is because the accumulated jitter between samples must be large relative to the RO period so that the sampled bits are effectively decorrelated.. We have a health test that counts the number of 1s and 0s as well as looking for bit strings that are continuous.

We configure clock domain crossing using a custom layout double flip-flop to deal with the metastability caused by the asynchronous output of the XOR tree. To ensure the randomness of the bits is scattered, we then feed this into a 16 bit serial LFSR conditioner to spread out and generate a random bit sequence. In its final stage, this is fed into the output register. The TRNG sub-system will work asynchronously to the data transformation accelerator.

F. *Parallelization and Throughput*

One way we intend on increasing the throughput of our current accelerator is by supporting the ability to run multiple algorithms at once. We do this by having multiple program counters that support instruction fetching of separate algorithms.

However, to ensure the data processing occurs without deadlock or excessive stalling when running multiple programs, each program being processed, whether on ROM or in SRAM, has a pre-determined max register count (for SRAM, this data will be passed in via the header packet). This count represents the maximum registers that the program will actively use at any one point. From this count, if the sum of register usage exceeds the amount of registers available (4), then the program that came first chronologically goes first. Otherwise, two programs will be able to run in parallel.

When two algorithms run in parallel, we will use a virtualization scheme for the registers. This means that if both programs call to register 0, they will be dynamically mapped to different registers.

An example of where this would work would be with running multiple CRC-32 algorithms in parallel, since CRC is a 32 bit accumulate style algorithm, it can occupy the same 128 bit register block while taking advantage of the multiple copies of the FUs that exist on the processor.

III. Roadmap

We will design and verify the functionality of our chip as follows:

A. Establishing Architecture & Tooling

This first stage is mainly focused on thoroughly establishing the architecture design and whiteboard for every edge case. Our plan is to find and fix any flaws in our architecture, work on area estimates, and create detailed microarchitecture specifications for our design.

After the architecture is fully specified, we plan to create a golden model that can be used for performance testing, and later on for verification of the design. This will be developed in C++ using gem5 (or equivalent).

Overall, our main hierarchy will be the shared ARX operations, as well as potentially further smaller, algorithmic specific units supplemented that may fall outside the ARX family. Additional units are a deliberate extension of the core, but larger possible units are out of the scope of this chip.

Here we also hope to establish much of the toolchain for RTL, setting up:

- Version control/collaboration with Git
- VCS/Verdi for simulation and linting
- Synopsys Design Compiler for synthesis
- Cadence Innovus for PNR and Calibre DRC & LVS

This should ideally be completed by the end of June 2026.

B. RTL Design & Verification

This stage is focused on developing and verifying the RTL for our chip. For the design aspect, we will split up the chip into logical blocks, and assign each person a block to design and validate basic functionality.

On the verification side, we will first test each unit with dedicated SV testbenches and directed testing, with outputs compared with the gold model. We will perform extensive testing to ensure that the instruction stream from memory to each functional unit is correct, as well as further edge cases in stalling for functional units, and any routing bugs. Eventually, full algorithms will be run end to end and compared with the gold model. We will also utilize Python-based instruction generation for further testing.

Functional coverage on our custom instruction encoding is essential to our verification process as well, covering custom opcodes and register operands. Post-synthesis verification is needed as well in ensuring proper timing, unintended latches, or any other issues not covered in simulation.

This phase should ideally be completed by mid-September.

C. Physical Design

This stage is focusing on translating the RTL to GDSII layout complying with the provided $1mm^2$ and 52 pin constraints. Thorough floorplanning will be done in regards to TRNG switching noise with proper spacing, as well as further factors such as SRAM/ROM placements for minimal bus length. PD will be split between Analog and Digital, with Digital focusing on standard cell P&R, SRAM/ROM. Analog will focus on the TRNG layout, especially custom inverters and custom ring oscillator layouts.

Clock domain crossing will also be laid out manually, as mentioned in the TRNG section with double flip-flops.

Verification of PD will be done through DRC and LVS per the TSMC 65nm rules.

This phase should ideally be completed by 1-2 weeks before the course deadline. We are leaving ourselves leniency for any issues which may arise in the future.

D. Software Bringup & Physical Interfacing

Before arrival, we will work on bringing up a PCB for our chip and an FPGA to interface with the chip. The PCB will be relatively simple, with just pin-out for the chip. The FPGA will interface directly with the chip, providing all of the signals to, and reading from the chip. These will run on the same clock domain.

Upon chip arrival, we must ensure that our own handshaking protocol works by sending test data to SRAM, and ensure a proper handshake can take place. From there, we must test our chips' individual functional units. However, to do this we must support customized assembly, which we plan on doing with Python-based instruction generation. We plan on utilizing custom Python scripts to generate specific instructions as to avoid designing our own compiler.

The idea would be to do a series of known answer tests on our four supported algorithms. We would have the same data inputted to a generic software implementation of our four supported FUs and our hardware implementation that would then be compared byte by byte to ensure full correctness.

Furthermore, to ensure correctness of the data transformation hardware, we must have a mechanism to bypass the TRNG. To do this, we expose a port that disconnects the TRNG from the registers it writes to.

Our group would then compare against a cycle-accurate model of our design in SystemVerilog for custom instructions and our supported algorithms.

Current Area Breakdown (extremely broad):

Very Big Area (≥25%), Big Area(≥10%), Medium Area(≥5%), Small Area(<5%), Very Small Area(<1%), **% relative to total utilization**

1KB Dual-port SRAM	Medium Area
ROM (max 1KB)	Small Area
Registers (4* 128b, 256 bits of R/W ports per reg)	Very Big Area
Functional Units (ADD/XOR/AND)	Big Area
Multi-Cycle Shifter	Medium Area
Memory Controller	Big Area
Control Units	Medium Area
Ring Oscillators	Small Area
XOR tree, LFSR, CDC, Health Test	Small Area
Routing, Clock Tree	Very Big area

* Note that the area can not be comprehensively calculated until synthesis, these are extremely broad estimates

Throughput Rough Estimate

A quarter round of ChaCha20 (done on 128-bits):

Operation	Cycles
Add	2
XOR	1
Shift << 8	1
Shift << 8	1
Add	2
Shift << 8	1
Shift << 4	1
Add	2
XOR	1
Shift << 8	1
Add	2
XOR	1
Shift << 7	3

Overall, the computation for a round of ChaCha20 would take approximately 19 cycles. A few more cycles would need to be added for overhead.

Scale Up & Scale Down Possibilities

While we have laid out lofty goals in this proposal, we understand these goals may not be fully possible due to area/time constraints.

Below, we have laid out possibilities for scaling up and scaling down this design:

<u>Scale Up:</u>

- Byte Level Shifter (allows us to support shifts greater than 8 bytes per cycle). Discussed in greater detail in our Architecture section. Best case scenario, we would do a barrel shifter, but we understand that this is an extreme area cost.
- Using an AES algorithm hash in our TRNG filtering - while an LFSR is a good start, we understand that it is not optimal for filtering our TRNG data. We would like to swap this to an AES algorithm if area allows for it.
- Adding more functional units, such as OR, NAND, etc.

- More algorithms such as compression (LZ4) which requires its own memory slice
- Parallelizing algorithms that do not overlap in their register use. Considerations we would need to have: scheduling functional units (incase of overlap), partitioning registers to prevent register overlap

Scale Down:

- If complexity/time constraints become an issue, we are very willing to drop the TRNG aspect
- Worst case, we will scale down our registers to 64 bits

IV. Team Composition & Division of Tasks

Division of Tasks:

Category	Leads
Architecture/RTL	Nikhil, Datis
*Analog Design	Alex, Srijan
Physical Design	Everitt, Shaun
Design Verification	Datis, Everitt
Scripting	Datis, Srijan
Hardware Tooling	Alex, Shaun
Bringup	Nikhil, Everitt

* Depending on Feasibility

While we expect every member to be touching each aspect of the project, we plan to assign 2 leads to each aspect of the project for redundancy and organizational purposes.

- **Nikhil**: Junior in Computer Engineering
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 391
  - *Experience:* FPGA Engineering @ Citadel Securities, FPGA Prototyping (SoC Design Team) @ Amazon, CPU/ASIC Design Verification @ Northrop Grumman
  - *Strengths:* RTL Design, Verification, FPGA & PCB Bringup
- **Datis** : Junior in Computer Engineering
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 408, ECE 391 CA
  - *Experience:* GPU Programming & FPGA @ KLA, Embedded Systems @ Surface Art Engineering, Project Lead at IEM
  - *Strengths*: Software, RTL Design, Verification, Scripting
- **Everitt** : Junior in Electrical Engineering
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 437, ECE 408
  - *Experience:* FPGA Engineering @ Leidos, LASSI Research Group Electronics Design
  - *Strengths:* RTL Design, Physical Design, FPGA Bringup
- **Shaun** : Junior in Computer Engineering
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 391
  - *Experience:* Hardware Validation @ Rivian
  - *Strengths:* RTL Design, Physical Design, FPGA Bringup
- **Srijan** : Junior in Electrical Engineering
  - *Relevant Coursework:* ECE 483, ECE 385, ECE 340, ECE 313 CA
  - *Experience*: Hardware Design Intern @ Astera Labs, LASSI Research Group Electronics Design
  - *Strengths:* Software/Scripting, Analog Design
- **Alex** : Junior in Electrical Engineering
  - *Relevant Coursework:* ECE 483, ECE 425, ECE 340
  - *Experience:* Transceiver Design @ Coherent Corp., Lead Software Developer @ Neovault
  - *Strengths:* Physical Design, Software/Scripting, Analog Design

Overall, our team brings an extremely strong and diverse set of skills. With the breadth of experience we have across different domains within this field, we believe that we will have the ability to handle the wide range of problems that will come up during this project.

References

[1] G. Sayilar and D. Chiou, “Cryptoraptor: High Throughput Reconfigurable Cryptographic Processor,” University of Texas at Austin, Austin, TX, USA.

[2] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, Recommendation for the Entropy Sources Used for Random Bit Generation, NIST Special Publication 800-90B, National Institute of Standards and Technology, Gaithersburg, MD, Jan. 2018. doi: 10.6028/NIST.SP.800-90B.

[3] S. Choi, Y. Shin, and H. Yoo, "Analysis of Ring-Oscillator-based True Random Number Generator on FPGAs," in 2021 International Conference on Electronics, Information, and Communication (ICEIC), Jeju, Republic of Korea, Jan. 31 – Feb. 3, 2021, pp. 1–3. doi: 10.1109/ICEIC51217.2021.9369714.