RandomX Hash Accelerator Documentation

Executive Summary

This is the documentation for the RandomX Hash accelerator project. The aim of this project was to accelerate the VM Loop Execution stage of the RandomX cryptocurrency hashing algorithm. The remaining part of the hashing algorithm is ran by an external CPU processor. This project is in an incomplete state currently, and was simplified from the original proposal which turned out to just be a boondoggle of complexity that was too difficult to implement in the term.

Table of Contents

Background

Documentation for the RandomX algorithm can be found here: https://github.com/tevador/RandomX/blob/master/doc/specs.md. A copy of this in PDF form can also be found in the docs/ folder, at this link: https://github.com/wwerst/randx_fpga/blob/main/docs/RandomX_specs.pdf
This RandomX document will sometimes be referred to as the "specs document."

What follows in this section is a background on the RandomX hashing algorithm. This project will implement the critical, slow subpart of the algorithm on an FPGA. The remainder of the algorithm will still run on a CPU over USB (down-the-road possibly PCIe).

The RandomX hashing algorithm is a recently developed hashing algorithm designed for cryptocurrency proof-of-work with cpu-only miners. The algorithm is designed to reverse engineer an algorithm whose optimal ASIC implementation is a modern cpu. It does this by generating a random program that maps nicely to x86 instruction set, and takes advantage of several modern cpu features such as tiered data caches, branch prediction/speculative execution, and out-of-order superscalar execution. The algorithm generates instructions for a hypothetical RandomX CPU, which has a special instruction set where all random 8-byte words are valid instructions. The RandomX cpu block diagram is below:

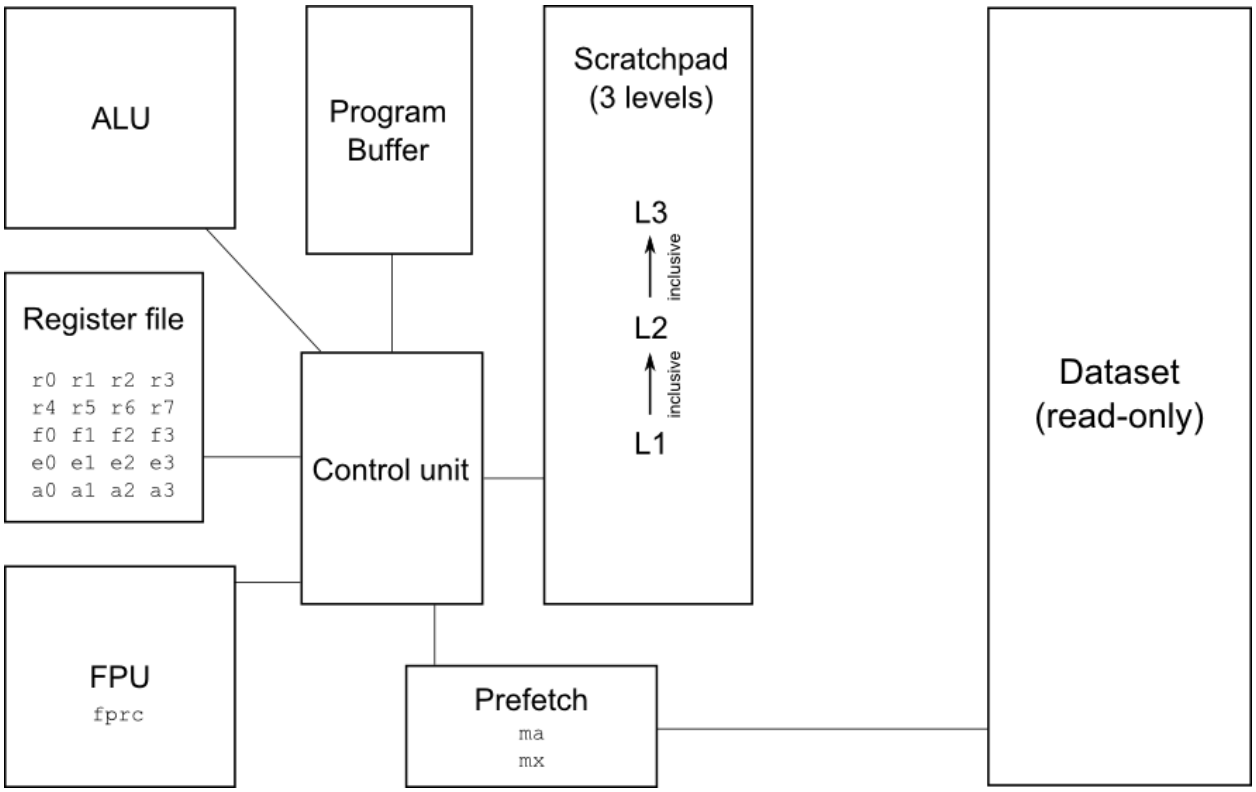

Figure 1. This is an architecture diagram for the RandomX cpu from the specs document.

The hashing algorithm uses this RandomX CPU as the slow part of hashing function, but there are many other parts that are necessary as well. This project will only target the slow part (elaborated more later), but for completeness, the full algorithm is captured below. There are two algorithms, the “Main Algorithm” and the “VM Loop”.

2. Algorithm description

The RandomX algorithm accepts two input values:

- String `K` with a size of 0-60 bytes (key)
- String `H` of arbitrary length (the value to be hashed)

and outputs a 256-bit result `R`.

The algorithm consists of the following steps:

1. The Dataset is initialized using the key value `K` (described in chapter 7).
2. 64-byte seed `S` is calculated as `S = Hash512(H)`.
3. Let `gen1 = AesGenerator1R(S)`.
4. The Scratchpad is filled with `RANDOMX_SCRATCHPAD_L3` random bytes using generator `gen1`.
5. Let `gen4 = AesGenerator4R(gen1.state)` (use the final state of `gen1`).
6. The value of the VM register `fprc` is set to 0 (default rounding mode - chapter 4.3).
7. The VM is programmed using `128 + 8 * RANDOMX_PROGRAM_SIZE` random bytes using generator `gen4` (chapter 4.5).
8. The VM is executed (chapter 4.6).
9. A new 64-byte seed is calculated as `S = Hash512(RegisterFile)`.
10. Set `gen4.state = S` (modify the state of the generator).
11. Steps 7-10 are performed a total of `RANDOMX_PROGRAM_COUNT` times. The last iteration skips steps 9 and 10.
12. Scratchpad fingerprint is calculated as `A = AesHash1R(Scratchpad)`.
13. Bytes 192-255 of the Register File are set to the value of `A`.
14. Result is calculated as `R = Hash256(RegisterFile)`.

Figure 2. The “Main Algorithm” from the specs document.

4.6 VM execution

During VM execution, 3 additional temporary registers are used: `ic`, `spAddr0` and `spAddr1`. Program execution consists of initialization and loop execution.

4.6.1 Initialization

1. `ic` register is set to `RANDOMX_PROGRAM_ITERATIONS`.
2. `spAddr0` is set to the value of `mx`.
3. `spAddr1` is set to the value of `ma`.
4. The values of all integer registers `r0` - `r7` are set to zero.

4.6.2 Loop execution

The loop described below is repeated until the value of the `ic` register reaches zero.

1. XOR of registers `readReg0` and `readReg1` (see Table 4.5.3) is calculated and `spAddr0` is XORed with the low 32 bits of the result and `spAddr1` with the high 32 bits.
2. `spAddr0` is used to perform a 64-byte aligned read from Scratchpad level 3 (using mask from Table 4.2.1). The 64 bytes are XORed with all integer registers in order `r0` - `r7`.
3. `spAddr1` is used to perform a 64-byte aligned read from Scratchpad level 3 (using mask from Table 4.2.1). Each floating point register `f0` - `f3` and `e0` - `e3` is initialized using an 8-byte value according to the conversion rules from chapters 4.3.1 and 4.3.2.
4. The 256 instructions stored in the Program Buffer are executed.
5. The `mx` register is XORed with the low 32 bits of registers `readReg2` and `readReg3` (see Table 4.5.3).
6. A 64-byte Dataset item at address `datasetOffset + mx % RANDOMX_DATASET_BASE_SIZE` is prefetched from the Dataset (it will be used during the next iteration).
7. A 64-byte Dataset item at address `datasetOffset + ma % RANDOMX_DATASET_BASE_SIZE` is loaded from the Dataset. The 64 bytes are XORed with all integer registers in order `r0` - `r7`.
8. The values of registers `mx` and `ma` are swapped.
9. The values of all integer registers `r0` - `r7` are written to the Scratchpad (L3) at address `spAddr1` (64-byte aligned).
10. Register `f0` is XORed with register `e0` and the result is stored in register `f0`. Register `f1` is XORed with register `e1` and the result is stored in register `f1`. Register `f2` is XORed with register `e2` and the result is stored in register `f2`. Register `f3` is XORed with register `e3` and the result is stored in register `f3`.
11. The values of registers `f0` - `f3` are written to the Scratchpad (L3) at address `spAddr0` (64-byte aligned).
12. `spAddr0` and `spAddr1` are both set to zero.
13. `ic` is decreased by 1.

Figure 3. The “VM Loop” from the RandomX specs document. This is step 8 in the “Main Algorithm”.

Functional Specification

The VM execution stage in the above background section on the RandomX algorithm is the critical code in the hash function. Depending on the specifics of the CPU that the hash is being ran on, this VM execution stage takes up about 90-95% of the time to run a hash. The remaining time is taken up by other parts of the algorithm, such as an AES generator step to generate random data to seed the program, and also a final hashing The VM execution is a series of 8 programs that are randomly generated for a special instruction set for RandomX, where all randomly generated 8-byte instructions are valid instructions. The design of the algorithm is reverse-engineered in a way that is designed so that a modern cpu is the optimal ASIC design for the hashing algorithm. It is designed to benefit from use of L1, L2, and L3 cache, branch prediction/speculative executions, and out-of-order execution. To date, the RandomX algorithm has held up against ASIC or GPU acceleration because of this design. In this project, we aim to build a RandomX-capable cpu in the FPGA.

Algorithm with Hash Example

In the following, the term Main Algorithm refers to the algorithm steps in section 2 of the specs.md file.. Also in the following, VM loop refers to section 4.6 in the same file. CPU refers to the host computer's cpu, which the FPGA board is connected to via USB. All CPU-based code will be adapted from existing working implementations (XMRig or RandomX repo), with the new FPGA work used as an accelerator for the critical parts of the current algorithm.

Algorithm Steps:

1) On the CPU, Algorithm steps 1-6 are executed (the dataset in step 1 is typically common to recent hashes and changes infrequently). The computed scratchpad is saved in RAM on the CPU.
2) The CPU executes Main Algorithm step 7, generating the VM program.
3) The CPU traces dependencies and converts the generated VM program into the dataflow format.
4) The CPU sends a new hash initialization packet to the FPGA control unit. The control unit commands the Scratch to load from USB via the USB Load En signal.
5) The CPU then streams in the generated scratchpad to the scratchpad unit, as well as the dataflow program instructions to the control unit.
6) The control unit runs the VM loop the prescribed number of times (2048 currently). Then, the CPU reads out the register file from the control unit (which retrieves this from the end of the result tables in appropriate dataflow units).
7) The CPU computes the next program, and informs the control unit of the next program to load. If no next program, jump to step 9.
8) The control unit loads the next program over USB, and then runs step 6 and 7 above.
9) The CPU commands the control unit to end the hash computation and store the scratchpad to the USB bus. The CPU finishes the hash computation.

Block Diagrams

Loop Engine

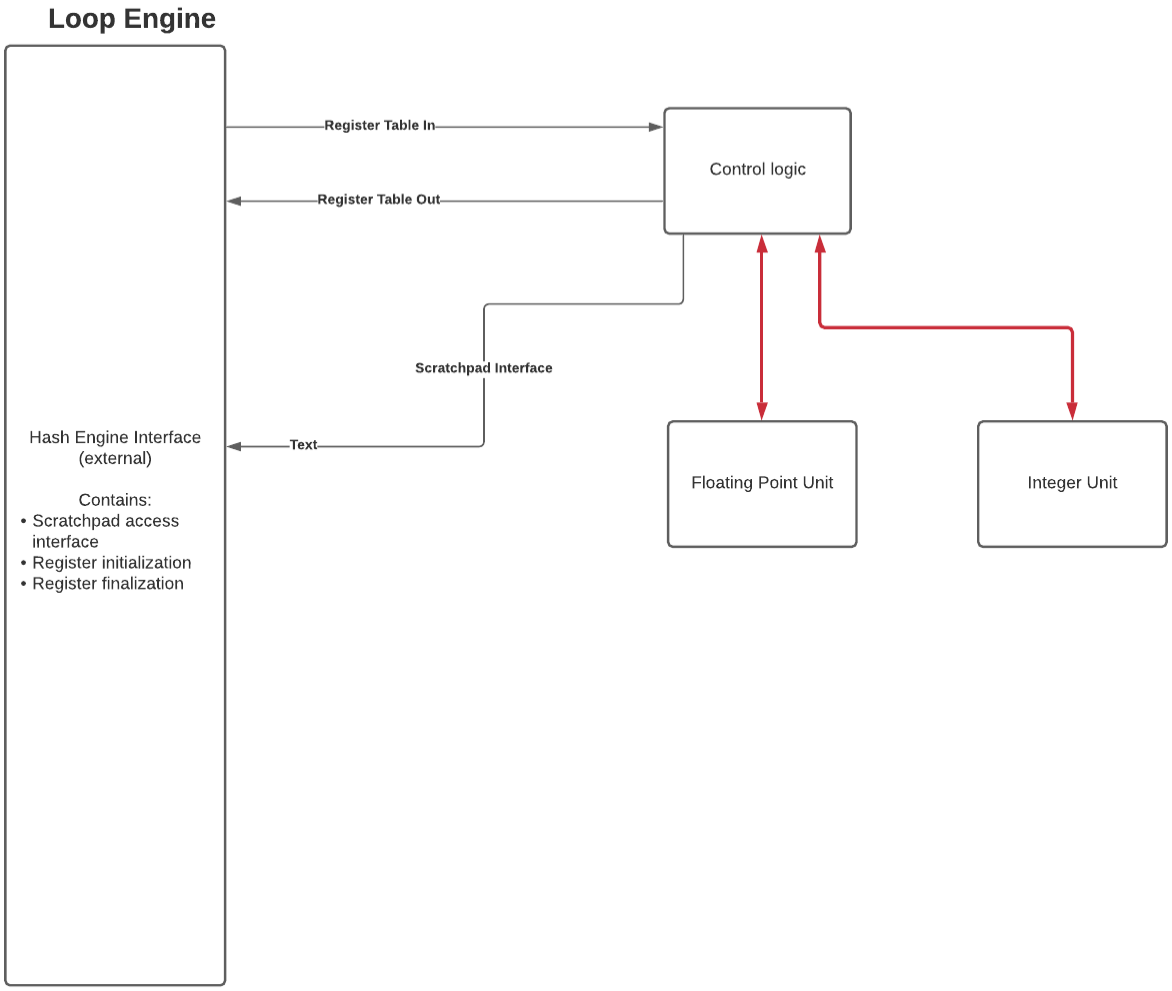

Figure 1. Block diagram of the Loop Engine.

The loop engine runs the main program execution. The program is pre-loaded into the loop engine's program memory by the hash engine. When the command is given, the register file is loaded from the hash engine and then one iteration of the loop is ran. The register file is then presented back to the hash engine. This is repeated for the 2048 times that the RandomX algorithm requires.

Hash Engine

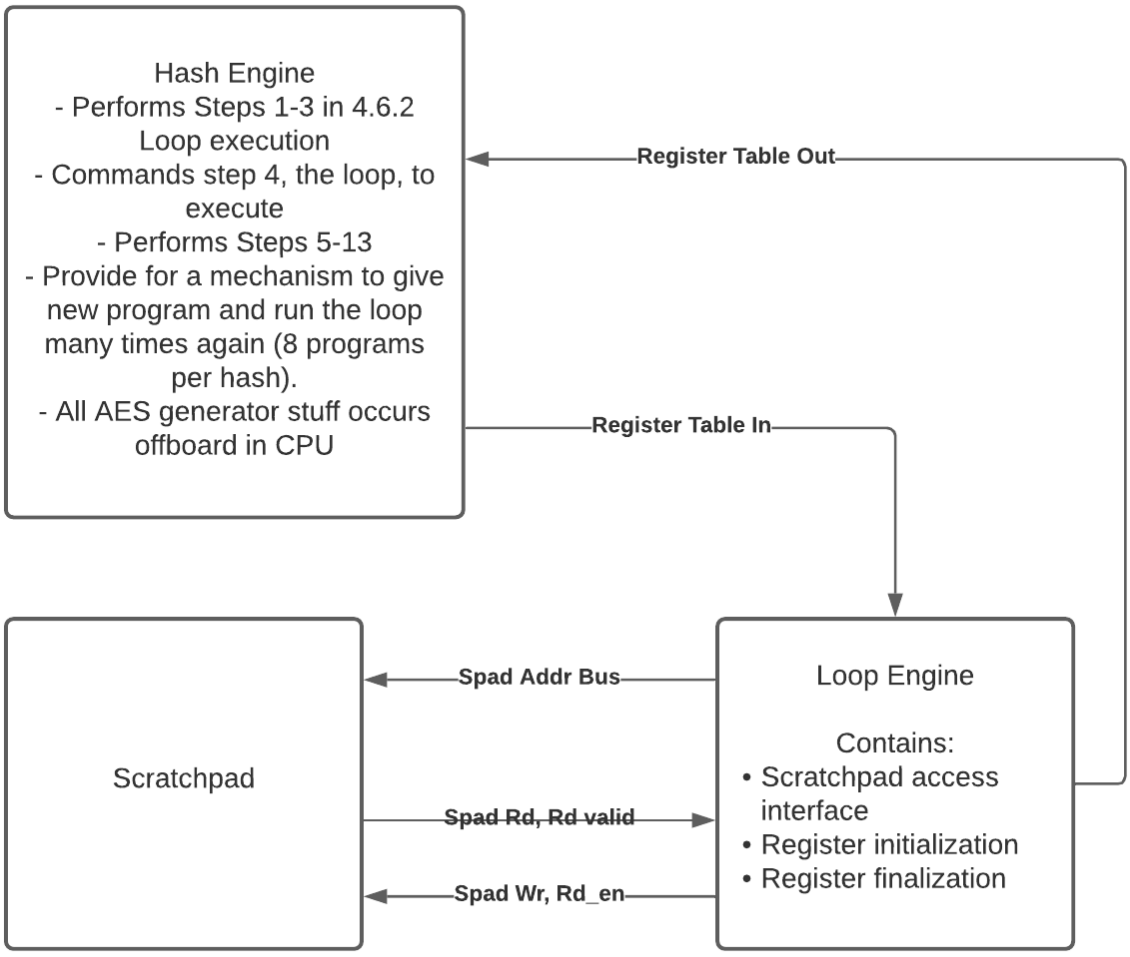

Figure 2. Hash Engine.

The hash engine orchestrates the whole operation. It performs the rest of the hashing operation besides the 4.6.2 loop execution, which is performed by the loop engine. The hash engine interfaces with the host cpu over USB 3.0