Accelerating Homomorphic Encryption

Accelerator Architecture

Noelle Crawford[*]
noellec3@illinois.edu
University of Illinois at Urbana-Champaign

Marcanthony Huang[*]
mg25@illinois.edu
University of Illinois at Urbana-Champaign

Gregory Jun[*]
hgjun2@illinois.edu
University of Illinois at Urbana-Champaign

Arpan Raja[*]
arpanr2@illinois.edu
University of Illinois at Urbana-Champaign

Wesley Wu[*]
wwu70@illinois.edu
University of Illinois at Urbana-Champaign

1 ARCHITECTURAL OVERVIEW

See pg.3 for our architectural diagram.

1.1 RISC-V CPU

We will be implementing and testing our accelerator to work with a simple 32-bit speculative execution RISC-V CPU. This processor will be used to run our baseline experiments.

1.2 Control Unit

The control unit is in charge of tracking state for our accelerator. To simplify the main control unit, we will implement the computation control unit as a separate component specifically for tracking the state of the systolic array computation and register routing. This is necessary because we must ensure we feed register values into the systolic array following the correct pattern, and route them correctly into the result register.

1.2.1 Load/Execute. During this phase we will first check to see if we are finished and make sure we are allowed to go through with this operation. If we reached the ed we will signal the process finished. Otherwise we will begin the process of loading in the blocks of data for A and B on which we will do our computation. Due to the nature of a systolic array we will be loading ad computing data as it comes in. Once we have loaded and executed on all the data needed (filling up the output regfile), we will store loads and write to thee writeback stage.

1.2.2 Writeback. After the output regfile has successfully bee loaded we will the begin the process of storing the data into the memory using the pointer stored in the dram controller. We will also increment the pointer to prepare the accelerator for the next phase.

1.3 DRAM

For simplicity, we will model the DRAM during simulation using a simplified (non-standard compliant) memory interface. In reality the CPU will handle all interfacing with memory which is already built into the existing cpu, and will be connected using the 'dram' interface depicted in thee diagram.

1.3.1 DRAM Controller. The DRAM controller will select the correct address to send to the DRAM interface, as well as toggling the R/W signals. The pointers will be stored where we begin the fetch data and will be incremented as we load data (A and B) and store data (C). We will calculate a upper limit for our destination pointer and will stop execution once we hit this limit.

1.3.2 Register File Selector and Selection Buffer. The register file selector routes the DRAM output to the register files to be used as input. It outputs four signals, the 64-bit buffered polynomial coefficient to be loaded, and a valid bit to be sent to each register file and signify which one should load.

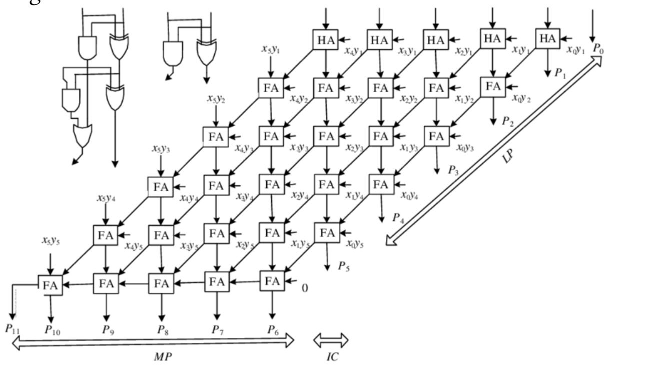
1.4 Register Files

Our design has 3 SRAM register files: A, B, and C such that we are computing $A \times B = C$. Register files A, B each store 16 64-bit coefficients for the duration of a computation, and register file C must store 32 such coefficients due to the nature of the computation.

1.5 Systolic Array

The major computational component of our algorithm will be routed using a systolic array. This prevents us from having to constantly write and retrieve data from our register files, while simplifying data movement. The base computation being performed in each cell is a 64-bit modular multiplication.

1.5.1 Bit Parallel Multipliers. We are currently planning to implement our cells as bit-parallel multipliers in order to maximize throughput, however, we are still in the process of testing alternative solutions. The diagram below shows the dataflow withing a single cell.



[*] All authors contributed equally to this work. (Alphabetical order)

2 PROGRAMMING MODEL

Our accelerator provides a single instruction specifying it's use:

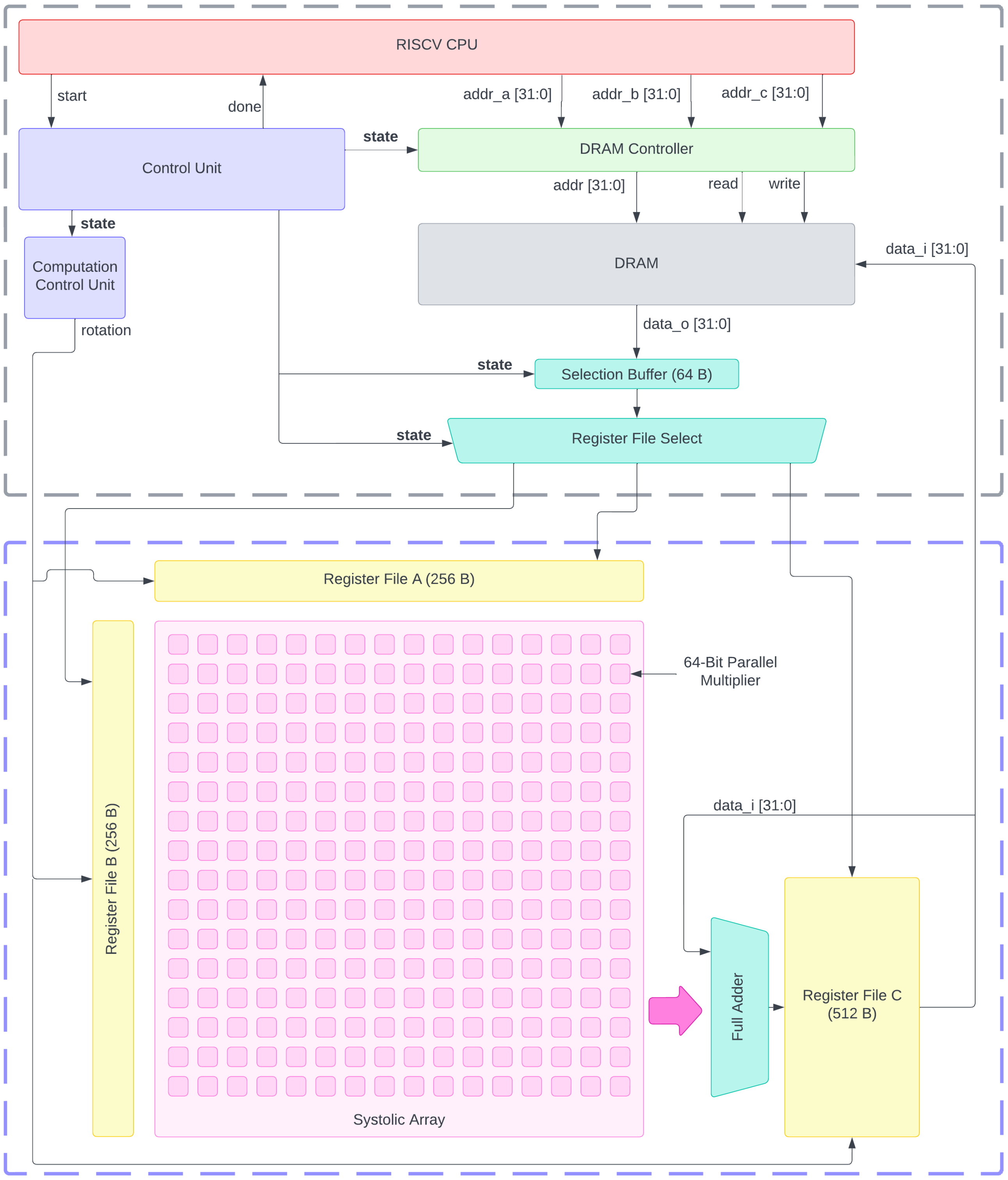
hmul rs1, rs2, rd, h_{imm}

Here rs1 and rs2 are to pointers for our 2 arrays (A and B). rd will correspond to the pointer for our final output (C). h_{imm} corresponds is a unsigned-10 bit value which is used to determine the size of the input. Here $2^{h_{imm}}$ is equal to the length of the polynomial.

ex. hmul r1, r2, r3, $1024

This instruction would tell the CPU to perform a polynomial multiplication using r1 and r2 as inputs. The order of the input polynomial will be 1024, and the output will placed into memory location r3.

REFERENCES


RISCV CPU
start
done
addr_a [31:0]
addr_b [31:0]
addr_c [31:0]
Control Unit
state
DRAM Controller
addr [31:0]
read
write
state
Computation Control Unit
DRAM
data_i [31:0]
rotation
data_o [31:0]
state
Selection Buffer (64 B)
state
Register File Select
Register File A (256 B)
64-Bit Parallel Multiplier
data_i [31:0]
Register File B (256 B)
Full Adder
Register File C (512 B)
Systolic Array