

RISCrypto: Custom Crypto Processor

Abrianna Zhang

Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
abrianna@mit.edu

Kristie Sue

Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
kxue@mit.edu

Abstract—We present RISCrypto, a custom cryptographic processor implemented on an FPGA that supports the full RISC-V instruction set while integrating specialized cryptographic instructions. The processor accelerates modular arithmetic and small-scale RSA decryption, targeting RSA-256. Custom modules for modular addition/subtraction, modular exponentiation, and greatest common divisor (GCD) operations enable high-performance cryptography on FPGA.

Index Terms—RISC-V, FPGA, processor, cryptography, RSA, Montgomery multiplication, modular arithmetic

I. INTRODUCTION

RISCrypto is a custom FPGA-based processor designed to accelerate cryptographic operations. The system integrates the full RISC-V instruction set while providing specialized instructions and hardware modules for modular arithmetic, modular exponentiation (MODEXP), and greatest common divisor (GCD) computations. The processor communicates with a host PC via UART, allowing program loading and output streaming without requiring hardware recompilation.

II. SYSTEM OVERVIEW

The system consists of three main components: the I/O interface, the 5-stage RISC-V pipeline, and custom cryptographic modules.

A. Input/Output Interface

The processor communicates with an external host over a single UART port, which multiplexes program loading and result output via a switch. During program loading, bytes received over UART are written sequentially into the instruction BRAM. In execution mode, writes to memory-mapped UART addresses are buffered and sent to the host via the same UART line. When the end of a program is detected, the register values are dumped back onto the external host via UART.

B. 5-Stage RISC-V Pipeline

The core processor follows a conventional 5-stage pipeline: Fetch, Decode, Execute, Memory, and Write-back, although the Memory stage is actually pipelined into two serial parts for the sake of convenient memory access. Standard RISC-V instructions operate as usual, while custom crypto instructions are executed through specialized modules. Long-running operations execute asynchronously, with a register scoreboard managing data hazards and FIFOs managing pipeline transitions to maintain CPU throughput.

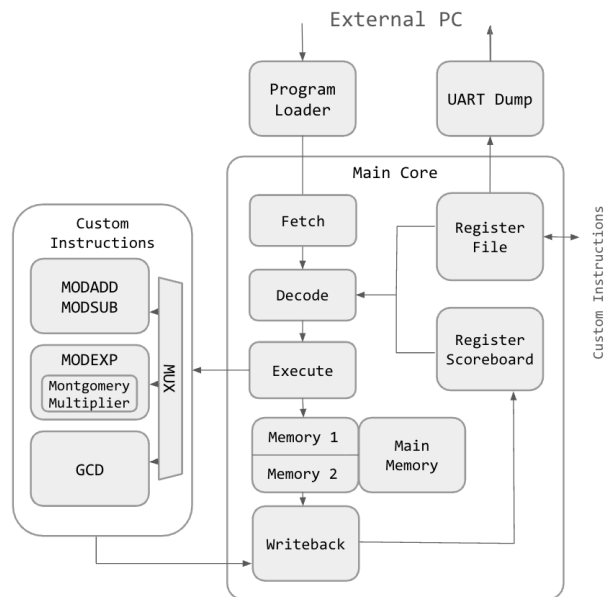


Fig. 1. Top-level block diagram of the RISCrypto system.

C. Custom Cryptographic Instruction Modules

The custom cryptographic instruction modules utilize internal 256-bit registers for operands and results. Long-running modules execute asynchronously, allowing the CPU to continue executing other instructions.

The following dedicated modules implement accelerated cryptographic instructions:

- 1) **MODADD/MODSUB**: Modular addition and subtraction for 256-bit operands.
- 2) **MODEXP**: Modular exponentiation for RSA decryption or key recovery.
- 3) **GCD**: Computes the greatest common divisor.
- 4) **RWR/WWR**: Reads and writes to special 256-bit registers.

They use the following helper modules to perform their computations:

- 1) **Montgomery Multiplication Step:** Performs inner loop modular multiplication, used as a helper module for modular exponentiation.

- 2) **Wide Multiplication:** Performs multiplication of two wide operands (targeted at 256 bits) in 64 bit chunks
- 3) **Wide Addition:** Performs addition of two wide operands (targeted at 512 bits) in 128 bit chunks
- 4) **Wide Comparison:** Performs comparison of two wide operands (targeted at 512 bits) by comparing the top and bottom halves in different pipeline stages
- 5) **Negative Modular Inverse Calculation:** Given the Montgomery modulus p , computes $-p^{-1} \pmod{R}$, where R is targeted at 2^{256} . This result is used in the Montgomery multiplication module.
- 6) **Montgomery Domain Conversion:** Converts in input a to its Montgomery domain counterpart, $\tilde{a} = aR \pmod{p}$ where R is a power of 2 (targeted at 2^{256}) and p is the Montgomery modulus.

III. MAIN CORE

A. Fetch

The fetch module queries the program data memory module (a dedicated BRAM block) upon each cycle, incrementing the internal pc register based on external signals (stalls and redirects). It also returns the current instruction received from the query two cycles ago, checks if there was a redirect to discard to data, and sends out the appropriate valid bit and instruction data back to the main core.

B. Decode and Execute

The decode module extracts instruction fields, determines execution paths (ALU, branch, memory, crypto), and outputs register values. The execute module performs standard ALU/memory operations. Both modules are combinational, and thus are pipelined within the main core module.

C. Memory and Registers

The register file provides fast combinational access to 64-bit registers. Only the writeback stage has access to the write line, while Fetch, along with some external debugging wires, can read from the registers. Program data is stored in a main memory BRAM, which supports 2-cycle reads and writes. We separated the 'Memory' stage into two pipelined parts, Memory 1 and Memory 2, as BRAM read/writes take 2 cycles.

D. Data Hazards

Data hazards are managed via a 32-bit array of 1-bit 'busy' values, the register scoreboard. Instructions stall in decode if any of the source registers are detected to be busy, and registers are freed in writeback (by clearing their corresponding bit in the scoreboard) once they have been updated.

Redirects and stalls are both managed by wires directly connecting modules later in the pipeline (Decode and Memory 1) to modules earlier in the pipeline (Fetch and Decode).

IV. CUSTOM INSTRUCTIONS

The custom instruction module takes in long-running commands and stores them in a FIFO, then processes them one by one. It stores the sources in the special `src1` and `src2` registers, the modulus in the special `mod` register, and the result of the operation in the `res` register. These long-running commands include all the crypto operations as well as reads from the result register. Other reads and writes return after one cycle. Since we also store a special register for the negative inverse of the modulus, this has to be updated every time the modulus is written to, which is done asynchronously. While this is happening, the negative inverse is marked as invalid. The status of the custom instructions is recorded in a status register - either `IDLE`, `BUSY`, or `DONE`. Commands are popped from the FIFO when the status is not `BUSY`. In addition, `MODEXP` must also make sure that the negative inverse of the modulus is also valid before being popped from the FIFO. Detailed explanations of the individual custom instructions, as well as the more complex helper modules, are given below.

1) **MODADD/MODSUB:** `Modaddsub` is a simple module which, depending on the custom instruction, computes either $(src1 + src2) \% mod$ or $(src1 - src2) \% mod$ and writes the result to `res`. It splits the computation into two cycles in order to avoid performing operations on three large inputs in one cycle. For addition, it computes $src1 + src2$ on the first cycle and subtracts the mod if necessary on the second cycle. For subtraction, it computes $sum = src1 + mod$ and $src1 \geq src2$ on the first cycle and uses these results to choose between $sum - src2$ and $src1 - src2$ on the second cycle.

2) **GCD:** The GCD module uses the Euclidean algorithm to compute the greatest common divisor, given inputs a and b . It uses an internal FSM with the following states:

- `GIDLE`, where the module is not performing any computations
- `GCOMP`, where the module compares a and b to see which is should reduce on the next step
- `GSUB`, where the module subtracts $\min(a, b)$ from $\max(a, b)$

It also keeps track of its internal status - `IDLE`, `BUSY`, or `DONE` - so that it can signal it to the main custom instruction module.

3) **MODEXP:** `Modexp` converts the base to the Montgomery domain, does all the multiplication in the Montgomery domain, and then reconverts out of the Montgomery domain. Since it is most useful for repeated multiplications, instead of being a separate instruction as in our original design, `montstep` is only a helper module for `modexp` and is currently not supported as a separate instruction. The following pseudocode illustrates the algorithm used by `modexp`:

```
def modexp(src1, src2, mod):
    pow = to_mont_dom(src1)
    res = to_mont_dom(1)
    e = src2
    while e:
        if (e & 1):
```

```

    res = montstep(res, pow, mod)
    pow = montstep(pow, pow, mod)
    e = e >> 1
    return montstep(res, 1, mod)

```

This algorithm is based on the traditional algorithm that repeatedly squares the base and multiplies them onto a result accumulator based on the bits of the exponent, except that it does all computations in the Montgomery domain to avoid reducing by the mod after every multiplication. The last step can be shown to convert `res` out of the Montgomery domain; the proof will be omitted as it easily follows from the calculations presented in the next section.

The `modexp` module uses an FSM with the following states:

- `MODEXP_IDLE`, where the module is not performing any computations
- `TO_MONT_DOM`, where the module converts its input into the Montgomery domain
- `MUL_POW`, where the module squares the power
- `MUL_RES`, where the module multiplies the result by the power, assigning this to the result accumulator based on the exponent
- `FROM_MONT_DOM`, where the module converts the result out of the Montgomery domain

To transition between states, we check whether the output valid signals of the Montgomery converter and Montgomery step helper modules are asserted, and update our registers using the outputted data. Like `gcd`, `modexp` also keeps track of its internal status to signal to the custom instruction module.

4) *Montstep*: Montgomery multiplication works as follows: Suppose the inputs are a and b and that the modulus is a prime p . Let r be a power of 2 greater than all of a, b , and p . Let $\tilde{a} = ar \bmod p$ and $\tilde{b} = br \bmod p$. Then $\tilde{a}$ and $\tilde{b}$ are a and b converted to the Montgomery domain. We will compute $\tilde{a}\tilde{b}r^{-1} \pmod{p} = abr \pmod{p}$, which is ab in the Montgomery domain, using the following steps.

- 1) Compute $t = \tilde{a} \cdot \tilde{b}$.
- 2) Compute the magic correction factor $m = tp' \pmod{r}$, where $p' = -p^{-1} \pmod{r}$.
- 3) Compute $t' = t + mp$.
- 4) Use a shift to compute $x = t'/r$.
- 5) If $x \geq p$, our final result is $x - p$. Otherwise, our final result is just x .

It can be shown that this computes $\tilde{a}\tilde{b}r^{-1} \pmod{p}$; the proof will be omitted as it is not that insightful. We can repeat this process to do many repeated multiplications. Afterwards, we convert out of the Montgomery domain by doing another Montgomery multiplication with our result $abr \pmod{p}$ and 1 as the arguments.

The `montstep` module uses an FSM with the following states:

- `MIDDLE`, where no computations are being performed
- `MMUL_ARGS`, where the module computes $t = \tilde{a} \cdot \tilde{b}$
- `MMUL_NINV`, where the module computes $m = tp' \pmod{r}$
- `MFLOOR`, where the module computes $\lfloor \frac{t}{r} \rfloor + \lfloor \frac{mp}{r} \rfloor$

- `MADJUST`, where the module possibly adjust the result from the above by 1 to get $x = \lfloor \frac{t+mp}{r} \rfloor$
- `MREDUCE`, where the module decides between x or $x - p$ for the final result

5) *Wide Multiplication*: The `big_mul` module multiplies two wide operands in 64 bit chunks. This module is necessary because large multiplication needs to be pipelined in order to meet timing, and there are not enough DSPs on the board to support 256×256 multiplication the usual way. We describe the algorithm we implemented for big multiplication works below, where the inputs are x and y . We assume x and y are 256 bits wide for illustration, though our implementation is parameterized and works for any widths above 64.

- 1) Split x and y into 64 bit chunks. Let them be x_0, x_1, x_2, x_3 and y_0, y_1, y_2, y_3 from least to most significant.
 - 2) Calculate the products $p_{ij} = x_i \cdot y_j$
 - 3) Calculate $\sum_{j=0}^3 p_{ij} \cdot 2^{64*i+64*j}$ for each i
 - 4) Add the results from above together to get the final result
- The module uses an FSM with the following states:

- `MUL_IDLE`, where no computation happens
- `XYSEL`, where the module selects the next chunks of x and y to compute the partial products of
- `PARTIAL`, where the module multiplies the selected 64 bit chunks of x and y
- `PARTIAL_PIPE`, where the module waits for the above multiplication for 4 cycles. This is so that the multiplication can be pipelined in the DSP block
- `BIG_PARTIAL`, where the module adds the next shifts and adds each partial product to the big partial product accumulator one by one
- `FINAL`, where the module performs the final addition of big partial products

Overall, this module takes 330 cycles for a single 256×256 bit multiplication.

6) *Conversion to Montgomery Domain*: The `to_mont_domain` module converts its input to the Montgomery domain. As described in previous sections, this means that given input a , it computes $aR \pmod{p}$ where p is the Montgomery modulus and R is a power of 2. It does this by shifting a to get aR , and then repeatedly subtracts $2^i p$, where i is decreasing starting at 256, until the final result is less than p . It uses an FSM with the following states:

- `TMD_IDLE`, where the module does not perform any computations
- `TMD_COMP_CALC`, where the module performs the comparison of ar and $2^i p$
- `TMD_COMP`, where the module uses the results of the comparison to decide whether to subtract from ar
- `TMD_SUB`, where the module performs the computation $1 + \sim(2^i p)$, which is $2^i p \pmod{R}$
- `TMD_ADD`, where the module adds the result from the subtraction step to ar , effectively subtracting $2^i p$

7) *Negative Modular Inverse Calculation*: As described before, the `calc_mod_inv` module computes $-p^{-1} \pmod{R}$,

where R is a power of 2, targeted at 2^{256} . This is done by computing $p^{R/2-1} \pmod{R}$ and negating the result, which works by Euler's theorem (since R is always a power of 2). The implementation is similar to that of modexp.

V. TESTING

A. Simulation

We have simulation test benches using cocotb for all our modules. For the main core, we simulate the program loader using a `MemoryModel` class that returns instructions after 2 cycles. These instructions are read from `.bin` files assembled from RISC-V files, taking advantage of the 'insn' directive for custom instructions. To make sure that our custom instructions are encoded correctly in our assembly and C programs, we have a python script that simulates our crypto processor on the binary files that we can check the output and register values from - basically a small RISC-V interpreter in Python. The tests also use this to check the register values from the cocotb simulation so that we don't have to hardcode these values every time we change a test program. Our testing strategy covers arithmetic, custom instructions, control flow, data hazards, and loads and stores to memory, using programs compiled from both RISC-V and C code.

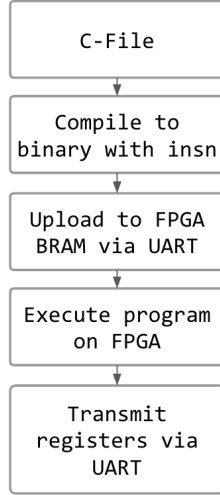


Fig. 2. Flow of toolchain.

B. Hardware

In addition to regular execution mode for the programs, we have supported a 'step mode' which allows users to step through the program one instruction at a time via a debounced button. To deal with memory-access timing, we artificially delay all BRAM requests by two cycles.

In 'step mode,' the lower two bytes of the current PC and lower 6 bytes of the current instruction are displayed on the seven-segment controller of the FPGA. In addition, a full register dump via UART to the receiving external PC occurs

after each step. A demonstration of this can be seen in our video submission.

The UART dump cycles through the 8 bytes of each of the 32 register indexes, extracts their values, and sends the appropriate byte to the host PC in a finite state machine cycle. The cycles are IDLE, WAIT, and TRANSMIT, with the FSM sitting in WAIT until the UART line is detected to be no longer busy.

VI. EVALUATION

A. Benchmarking

We benchmarked our processor in simulation, running it with and without our custom crypto instructions. Although our processor is written to be targeted at RSA-256, we benchmarked with much smaller moduli in the interest of time. While our hardware implementation of modexp uses Montgomery multiplication to avoid reducing by the modulus every time we perform a step of multiplication, our software implementation of modexp uses normal modular multiplication since Montgomery multiplication would require implementing 256 bit integers in the barebones C that we are using, which is quite cumbersome. As a result, when we tested it with width 256 for the crypto operations, we did not see a speedup from our custom instructions. This is because in order to perform Montgomery multiplication, our hardware implementation does computations with 256 bit integers, while our software implementation does not, making the hardware implementation a lot slower. However, when we decreased the crypto instruction width to 128 bits, we did see a significant improvement from using our custom instructions. Here are some benchmarking results based on our simulations with 128 bit crypto instructions, where the performance is measured in cycles.

Modulus	Without Custom Instructions	With Custom Instructions
143	117,643	55,085
589	683,899	64,355
745	738,678	73,565
851	2,855,141	84,585
1,763	3,028,627	147,701
1,841	3,223,687	162,941
1,843	3,062,050	249,139
2,201	3,119,726	114,490
3,737	6,365,627	386,002
5,671	8,246,090	388,352
7,081	12,461,130	247,410
8,383	13,624,438	759,784

TABLE I
BENCHMARKING RESULTS

From the data, the average speedup from our custom instructions was 91%, with a median of 95%.

B. Timing

Our design meets timing requirements for 256 bit crypto operations, though we ended up testing with 128 bit crypto operations as described above. For 256 bit crypto operations, our WNS was around 0.125 ns, while for 128 bit crypto operations our WNS was 0.523 ns. The timing requirement

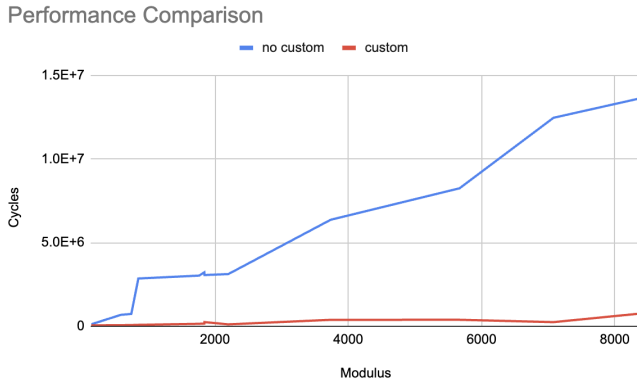


Fig. 3. Plot of Benchmarking Results

was achieved using the wide multiplication, addition, and comparison modules which pipeline these operations. Overall, our design is high latency and high throughput, as the custom instruction module can accept a new instruction on every cycle, but it may take many cycles until that instruction has finished processing.

C. Resource Usage

Our design uses 32 DSPs, since we perform 64×64 bit multiplication in our wide multiplication module. This could likely be reduced by doing multiplication in chunks of 32 bits instead, though this would significantly increase the number of cycles needed for wide multiplication and might affect performance. We use 3 BRAM blocks, two for the program itself and one for the program data. Our longest programs are 272 instructions or 8.7 kilobits long, which means we could have fit the entire program into one BRAM. However, if we actually implemented 256 bit operations in C, we may have needed the extra BRAM, as the programs would have become much longer. We use one 32×771 DRAM for the custom instruction FIFO. This could probably also be reduced since in practice there is only one long-running modexp or gcd waiting at once.

D. Implementation Insights

Although the initial implementations of the crypto instructions were relatively simple, we quickly realized that because these instructions worked with wide operands of 256 bits, these implementations far exceeded the resources on our FPGAs and drastically failed to meet timing with a WNS of under -10 ns. For example, our first version of our processor, which worked in simulation, could not build because it utilized more DSPs than the 120 available on the board. As a result, most of the work on the crypto modules went to reducing resource usage and meeting the WNS requirement. All of the three custom instruction modules underwent pipelining with the help of the wide addition, multiplication, and comparison modules.

The initial implementation of the fetch module also underwent changes. When testing with compiled C programs, we realized that the stalling logic we implemented initially did

not work for arbitrarily many stalls in a row. We ended up simplifying the logic so that it stalled for an extra 2 cycles for every data hazard. Although this increased the number of wasted cycles, it worked for all the complex C programs that we tested, for arbitrarily many data hazards in a row.

VII. CODE

Our code can be found at <https://github.mit.edu/6205F25/fa25-6205-team05>

VIII. CHECKLIST

We achieved all the items on our checklist from the commitment and goal sections. Our final processor is able to communicate with our computers through UART, run programs with arithmetic, custom instructions, control flow, data hazards, and loads and stores to program memory. In addition, as shown in the Evaluation section, we were able to achieve a significant speedup using our custom instructions. For the stretch section, we did not end up implementing a cache for program data. However, as shown in our video submission, our processor is noticeably faster at breaking RSA than Python running on a Mac CPU.

IX. CONTRIBUTIONS

Abrianna wrote and tested the code for the main core, the UART communication, and did the hardware testing. Kristie wrote and tested the code for the crypto operations, integrated custom instructions into the main core, debugged the fetch module, and did the benchmarking in simulation.

Abrianna wrote sections I and III of the report and created the block diagram. Kristie wrote sections IV, VI, VII, and VIII and created the performance table and graph. We collaboratively wrote all the other sections.

REFERENCES

- [1] RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA," 2019. [Online]. Available: <https://riscv.org/technical/specifications/>