SHA256-Efficient CPU Based on RISC-V Architecture*

Tsang-Yung Wu[1**] and Ming-Kuang Lin[1**] and Wen-Kuan Chang[1**]
Advisor: Prof. Tzi-Dar Chiueh

***Abstract*— During undergraduate research, we combed through lots of application papers, and felt that we were doing Computer Vision research instead of IC design. As a result, we decided to focus our mind on circuit design and we found Chisel and RISC-V foundation. Their purpose is to reduce the cost and risk of designing custom silicon and open-sourced RISC-V ISA standard to cut down the IP fee of Arm ISA. So, based on Chisel framework and RISC-V ISA, we developed a novel architecture with customized instructions to boost up the efficiency of SHA256, which is an important technology for Bitcoin and Ethereum, with efficiency outperforming original RISC-V ISA by 13%. In the end, We compared our enhanced CPU with original RISC-V CPU and other CPUs on instruction counts, area, and power to demonstrate performance of our architecture.**

I. INTRODUCTION

This paper will help readers go through the design process of the proposed CPU architecture and explain on why we do that. In section two, we will discuss about the flow of designing a CPU and our SHA256-efficient architecture. From section three to section five, we'll elaborate on the details of our works including software design and hardware design. In section five, we compared our performance with other CPU architecture on several aspects, including software and hardware emulation results. And for the last section, we'll conclude this paper and address on the future of our work and provide some visions.

II. DESIGN CONCEPTS AND ANALYSIS

Basically, to solve problems existing in an application, that said accelerating an algorithm, one cannot directly design a chip from hardware stage since it is not flexible enough to try and error. And especially for computer architecture, there are lots of software simulation tools to utilize, like Gem5. Therefore, based on this concept, we developed our architecture from software design, examining the algorithm and conducting experiments. After that, we started designing our hardware and compared the results with our software simulation results to justify our assumptions. In this paper, we went through this procedures step by step and complete our research.

To begin with, we wanted to accelerate operations efficiency of SHA256. This is one of the most important encryption technologies now and is utilized in Blockchain application. If the world in the future would like to adopt Blockchain, we thought every computer should have much more efficient operations on SHA256. As a result, we analyzed on the C code pattern of SHA256 and conducted several experiments on it. Finally, we designed an instruction based on our analysis to accelerate SHA256.

A. SHA256 Analysis

We used C program of SHA256 as our research materials and tested on it. To lift the performance significantly, we wanted to find out the most regular and consuming part in SHA256 and reduced cost on it. Therefore, we examined the code pattern of SHA256 and broke them down. We found that `SHA256_F1` to `SHA256_F4` functions used a lot, and there are some operations shared among them. Then, we compiled C code with RISC-V compiler, and dumped out assembly code. We saw some regularity among them, shown as below. These lines of assembly code popped up regularly.

```
lw -> srliw -> slliw -> or -> sext
```

Later, we used computer architecture approach to analyze our problems with Gem5 tool. We found that if we deleted those functions, instructions counts are about 52,000. And if we added them back, the instructions counts increased to about 81,000. Therefore, we believed that they are critical parts of SHA256 operations, and if we could reduce costs on them, we would be able to improve SHA256 efficiency significantly.

B. Instruction Design

To solve this problem, we designed an instruction capable of executing the previous assembly code at once, called `CURL`. This instructions is straight-forward enough. It would load data of *rs* in and do both shift left and shift right. Then, *OR* both of them together to get the result output. Basically, we expected this instructions would boost up the performance for 2x.

imm[11:0]	rs1	110	rd	0011011	CURL

Fig. 1. CURL Instruction Code

In the following sections, we'll address the details of how we added a customized instruction in RISC-V ISA. More importantly, we utilized different tools and frameworks to facilitate our works, including Gem5 tool to do software simulation, RISC-V open source project and Chisel framework to develop hardware description code much faster.

*This work was not supported by any organization
**They have equal contributions to this paper
[1]These three are students from National Taiwan University and major in electrical engineering. They are all undergraduate researcher under Micro-System Lab advised by Prof. Tzi-Dar Chiueh.

III. Software Design

A. Modify RISC-V Compiler

The compiler we used is built from riscv-tools, which houses a set of RISC-V simulators, compilers, and other tools. The source code is available at *https://github.com/riscv/riscv-tools*. we added a new instruction curl , belonging to RV64I Base Integer Instruction Set. Its semantics are given below:

```
curl rd, rs, imm
R[rd]=(R[rs]>>imm)|(R[rs]<<(32-imm)
```

- Open the file riscv-opcodes/opcodes, here you would be able to see the various opcodes and instruction bits assigned to various instructions. Assigned an unused instruction to `curl` as the following line:

```
curl rd rs1 imm12 14..12=6 6..2=0x06 1..0=3
```

- Running the following command:

```
cat opcodes-pseudo opcodes opcodes-rvc
opcodes-rvc-pseudo opcodes-custom |
./parse-opcodes -c >  /temp.h
```

- You would find the following two lines in temp.h. `MASK_CURL` is the mask used to isolate the opcode from the other part of the instruction, `MATCH_CURL` is compared against the instruction that have been masked. Add these two lines in *riscv-gnu-toolchain/riscv-binutils-gdb/include/opcode/riscv-opc.h*

```
#define MATCH_CURL 0x601b
#define MASK_CURL 0x707f
```

- Edit the file *riscv-opc.c* under *riscv-gnu-toolchain/riscv-binutils-gdb/opcodes* and add this line in the assignment of `const struct riscv_opcode riscv_opcodes[]`:

```
{"curl", "64I", "d,s,j", MATCH_CURL,
MASK_CURL, match_opcode, 0}
```

- Compile the *riscv-gnu-toolchain* again and you would have the customized compiler.

B. C Program API

Additionally, we need to create a new API for C code in order to fully utilized the new added instruction. We created a new header file called sha256_API.h and redefine the SHA256_F1, SHA256_F2, SHA256_F3, SHA256_F4 functions using `CURL` instruction. An example of SHA256_F1 is shown below:

```
#define SHA256_F1 (a,x) \
asm volatile("curl t1, %1, 2\n\t" \
             "curl t2, %1, 13\n\t" \
             "curl t3, %1, 22\n\t" \
             "xor t1, t1, t2\n\t" \
             "xor %0, t1, t3\n\t" \
             : "=r" (a) \
             : "r" (x));
```

C. Simulator

After we modified our compiler and our code, we also need to modify simulators to define and recognize the functionality of new instructions; therefore we can check the correctness of simulation results. The tools we used are Spike and Gem5.

1) Spike: Spike is a RISC-V ISA Simulator which implements a functional model of one or more RISC-V processors. It can simulate a compiled program and also supports gdb debug mode to help you debug your program. The tool is under the *riscv-tools* folder, and we have to modify some files under the *riscv-isa-ism* folder. We simply use it to check our result.

- Go to *riscv-isa-sim/riscv/encoding.h*, and add the following lines:

```
#define MATCH_CURL 0x601b
#define MASK_CURL 0x707f
DECLARE_INSN(curl, MATCH_CURL, MASK_CURL)
```

- Create a file *riscv-isa-sim/riscv/insns/curl.h* and add the functionality of curl in it.
- Go to the file *riscv.mk.in* under *riscv-isa-sim/riscv*, find `riscv_insn_list` then add "curl" into the list.
- Go to *riscv-isa-sim/spike_main/disasm.cc* and add the following line:

```
DEFINE_ITYPE(curl);
```

- Rebuild the simulator, The `CURL` instruction has been added to spike simulator.

2) Gem5: Gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor micro-architecture. Currently, Gem5 supports most of the common ISAs (ARM, ALPHA, MIPS, x86 and RISC-V). It also supports multiple CPU models and different simulation mode such as SE (*Syscall Emulation*) mode and FS (*Full System*) mode for analyzing the instruction counts and cycle counts of our program.

First of all, check the requirements and follow the steps in *http://learning.gem5.org/book/part1/building.html* to setup gem5 in your computer. After that, modify the file *decoder.isa* under *arch/riscv* by defining the opcodes and functionality of `CURL` mentioned above. Rebuild Gem5 then we can start simulating our program.

IV. Hardware Design

After we finished software design and simulation, our architecture can accelerate the most critical part of SHA256 operations. Therefore, we used Chisel framework and RISC-V ISA to leverage our architecture. We used open source project RISCV-SODOR and modified its architecture to build our own SHA256-efficient RISC-V CPU.

A. RISCV-SODOR

RISCV-SODOR is a open source RISC-V project for educational purpose designed by Berkeley. They designed it to enable broad range of developers to quickly learn and utilize RISC-V ISA. It contains 5 32-bit RISC-V CPU to demonstrate simple RISC-V ISA written in Chisel, which are RV32-1-stage, RV32-2-stage, RV32-3-stage, RV32-4-stage, and RV32-5-stage. Additionally, each core has their features, for example RV32-2-stage is mainly used to demonstrate pipe-lining in Chisel. In our proposed architecture, we used RV32-1-stage, essentially an ISA simulator, to demonstrate our works. Basically, it enabled us to quickly modify its codes, and designed our architecture. Open source RISCV-SODOR projects code is available at *https://github.com/ucb-bar/riscv-sodor*.

B. Architecture Design

To enhance RV32-1-stage core, we need to modify control path and data path to add additional hardware resource. All related Chisel codes are located at *riscv-sodor/src/common* and *riscv-sodor/src/rv32_1stage*. The former directory stored common codes for all types of cores, and the latter stored specific configuration of RV32-1-stage core. Finally, We'll have to modify some codes in both of them.

- Go to *common* directory and open *instructions.scala*. Find an object called `Instructions` and add below code line to add customized instruction to let CPU know.

```
def CURL=BitPat("b??????????????????110????0011011")
```

- Go to *common* directory and open *consts.scala*, find *ALU Operation Signal* section and add below code line to create new ALU operation for SHA256.

```
ALU_SHA = 12.asUInt(4.W)
```

- Go to *rv32_1stage* directory and open *cpath.scala*, find `csingals` and copy `ADDI` signal, replace its name with `CURL` and change `ALU_ADD` to `ALU_SHA`.
- Go to *rv32_1stage* directory and open *dpath.scala*. Add extra operation in ALU to build new ALU supporting CURL instructions based on Fig. 2.

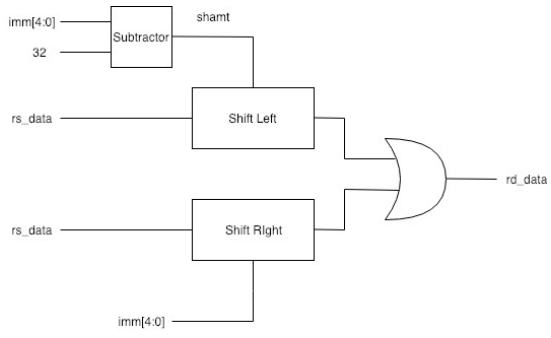

Fig. 2. CURL Block Diagram

C. Hardware Generation

We have finished our enhanced architecture in the previous section. Now, we have to generate Verilog code since Chisel code is non-synthesizable. Chisel uses **firrtl**, an intermediate representation to generate Verilog code, and **Verilator**, an open source tool to convert Verilog code to cycle-accurate behavioral model in C++ or SystemC, to further generate Verilog and C/C++ model. In this paper, we should go to *riscv-sodor* directory and type `make install` to generate them. The makefile configuration is specified in *riscv-sodor/Makefile*. You can take a look for further understanding.

D. Hardware Emulation

To test functionalities of our Chisel code, we need to do emulation. Basically, RISCV-SODOR provides C emulator to do C model emulation. It provides you a much faster way to test your code. On the other hand, using firrtl generated Verilog code, we can test it using self-written test-bench.

- To do C emulation, we generated SODOR emulator mentioned above located at *riscv-sodor/emulator/rv32_1stage*. First, we run pre-compiled benchmarks to check the correctness of our CPU design; then, we write our c model in order to compile a binary .riscv file, which is meant for running our custom benchmarks. The emulator will generate the emulation result with the whole process and tracer, a tool that can indicate some statistical data such as CPI, number of cycles, and number of arithmetic instructions, etc.

```
Cyc=         70 Op1=[0x80001000] Op2=[0
Mem[0: R:0x00000000 W:0x00000001] PC= 0
Cyc=         71 Op1=[0x00000000] Op2=[0
Mem[2: R:0x00000093 W:0x00000000] PC= 0
*** PASSED ***
#----------Stats--------------
#
#   CPI    : 1.00
#   IPC    : 1.00
#   cycles : 72
#
#   Bubbles      : 0.000 %
#   Nop instr    : 0.000 %
#   Arith instr  : 59.722 %
#   Ld/St instr  : 18.056 %
#   branch instr : 2.778 %
#   misc instr   : 19.444 %
```

Fig. 3. Emulator and Tracer

- To do Verilog emulation, we took out the main part of our design, `Core`, declared in Verilog code. This module executed most functionalities of CPU, including instructions decoding, execution, etc. What we had to do is to write machine code as testing patterns and feed into `Core` module to see results.

V. Simulation & Emulation Results

A. Software Results

In software simulation, we used Gem5 to simulate the whole SHA256 program for different iterations to figure out the improvement after modifying RISC-V CPU. Below are the simulation results of three different CPUs.

TABLE I
GEM5 X86 CPU SIMULATION RESULT

Iterations	Cycle Counts	Inst. Counts
0	4,231,898	2,048,910
1	4,325,664	2,093,489
10	4,961,577	2,398,580
100	11,320,707	5,449,490

TABLE II
GEM5 ORIGINAL RV CPU SIMULATION RESULT

Iterations	Cycle Counts	Inst. Counts
0	65,931	52,099
1	95,739	81,718
10	419,683	348,614
100	3,901,424	3,017,249

TABLE III
GEM5 MODIFIED RV CPU SIMULATION RESULT

Iterations	Cycle Counts	Inst. Counts
0	59,794	52,099
1	100,608	77,971
10	412,832	310,883
100	3,158,335	2,639,678

From Table I, Table II and Table III, we found that there still had 51,917 instructions for RISC-V and 2,048,910 instructions for x86 when running 0 iteration. It means that a fixed amount of instructions probably consists of system calls or linking associate files that would exist in the whole program. So we focused on the increments per iteration, described in the following table.

TABLE IV
INSTRUCTIONS INCREMENT PER ITERATION

Architecture	x86	Original RV	**Modified RV**
Insts. per Iteration	33,899	29,651	**25,875**

According to Table IV, the results show that the original RISC-V CPU outperform x86 CPU on the instruction counts per iteration by 12.5%. Furthermore, our Modified RISC-V CPU reduce instruction counts by 12.7% over the original one. In conclusion, RISC-V CPU is more efficient than X86 CPU, and additionally, our modified RISC-V CPU can lower the number of instructions and boost the efficiency by simple steps mentioned above.

B. Hardware Results

In hardware, we only emulated differences between original RISC-V CPU and modified RISC-V CPU. We used both C and Verilog emulation mentioned above to test. Moreover, we also compared the area, power performance between them to see more details.

In C emulation, we first ran provided pre-compiled benchmarks and basic functionality of CPU is proved to be correct. Then, we ran the C model of `SHA256_F1` several iterations with original CPU design and modified one to compare the number of instructions. The results is shown in Table V.

TABLE V
C EMULATION RESULT

Iterations	Original RV Inst. Counts	**Modified RV Inst. Counts**
1	106	100
10	358	296
100	3,412	**2,718**

In Verilog emulation, we ran `SHA256_F1`, which is one of the most important operations in SHA256, for several iterations to see the instruction counts difference between. Table VI shows the performance. The modified RISC-V processor has significant gain over the original one, which demonstrates the efficiency of our customized instruction, `CURL`.

TABLE VI
VERILOG SHA256_F1 EMULATION RESULT

Iterations	Original RV Inst. Counts	**Modified RV Inst. Counts**
1	14	8
10	113	53
100	1,103	**503**

Also, we used Design Compiler to synthesize our Verilog code with **TSMC13 technology**. Unsurprisingly, according to Table VII modified RISC-V processor has more area than the original one. And it is obvious since we have added additional computing resource in ALU. And the power consumption stay approximately the same. However, comparing performance gain with the area and power overhead, modified RISC-V processor is much more energy and area efficient since the instruction counts are low enough.

TABLE VII
AREA & POWER METRICS

Architecture	Original RV	Modified RV
Area (um^2)	208,486	209,880
Power (mW)	189.67	190.46

C. Overall Results

We used `SHA256_F1` as benchmark to compare instruction counts on software and hardware results and analyzed the difference of performance gains. Therefore, we executed 100 iterations of `SHA256_F1`, and the results is shown in Table VIII. For Gem5 simulation, we had subtracted the unused part of instructions counts to make comparison reasonable. However, We can found that instructions counts of software simulation are higher than low level simulation. We believe the reason is that since in lower level tool, we could more control the execution flow of the program, and obtain more accurate results. Therefore, from Verilog

emulation, we could learn that our modified CPU is 2x more efficient than the original RISC-V CPU on `SHA256_F1` function.

TABLE VIII
SHA256_F1 SOFTWARE SIMULATION & HARDWARE EMULATION RESULTS ON INSTRUCTION COUNTS FOR 100 ITERATIONS

Architecture	x86	Original RV	Modified RV
Gem5 Simulation	2,500	2,900	1,500
C Emulation	x	3,412	2,718
Verilog Emulation	x	1,103	503

Therefore, from software simulation, we learnt that our instructions could help us boost up the performance on SHA256. And, as Table VIII shown, the results are expected, and the hardware emulation results justified the efficiency of our architecture. In the end, we learnt that the more we analyzed from software level, the more performance gains we would get. And this is an important concept when designing a chip or CPU, especially when we want to be faster to get great results.

VI. CONCLUSION AND FUTURE WORK

In this paper, we investigated customized RISC-V ISA for SHA256 algorithm, motivated by increasingly important role of Blockchain applications. We analyzed SHA256 programs and accelerated it by reducing the instruction counts. Moreover, to help us develop hardware architecture faster, we utilized open source RISC-V project, *riscv-sodor*, and Chisel framework, aiming to reduce time costs to develop a chip. Within 3 months, we developed a SHA256-efficient RISC-V CPU outperforming original RISC-V CPU and x86 by 13% on the critical part of SHA256 algorithm. Yet our the complete procedure of setting up required environment and our works are avaialbe at *https://github.com/kuanforml/riscv-sodor/tree/cs152-sp18*, which we forked from original *riscv-sodor* project to help one quickly utilize this great open source project.

In the future, We plan to investigate more in encryption technologies used by Blockchain application to support future Internet infrastructure. We believe that encryption technologies will become more and more important and reduce the power consumption of these technologies is crucial as well. Based on this SHA256-efficient RISC-V CPU, we will add more instructions in it and leverage our architecture to further level.

REFERENCES

[1] Cheng, Kwang-Ting, and Yi-Chu Wang. "Using mobile GPU for general-purpose computinga case study of face recognition on smartphones." VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on. IEEE, 2011.

[2] Bachrach, Jonathan, et al. "Chisel: constructing hardware in a scala embedded language." Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE. IEEE, 2012.

[3] Waterman, Andrew, et al. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[4] Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer." Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014.

[5] Waterman, Andrew, et al. The RISC-V Instruction Set Manual Volume 2: Privileged Architecture Version 1.7. No. UCB-EECS-2015-49. University of California at Berkeley Berkeley United States, 2015.

[6] Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks." ACM SIGARCH Computer Architecture News. Vol. 44. No. 3. IEEE Press, 2016.

[7] Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." IEEE Journal of Solid-State Circuits 52.1 (2017): 127-138.

[8] Ergnay, Selman, and Yusuf Leblebici. "Hardware Implementation of a Smart Camera with Keypoint Detection and Description." Circuits and Systems (ISCAS), 2018 IEEE International Symposium on. IEEE, 2018.

[9] Rider, A. "IMPORTANT: YOU MUST READ THE FOLLOWING DISCLAIMER IN FULL BEFORE CONTINUING." (2018).