# Implementing RISC-V Scalar Cryptography/Bitmanip extensions in Chisel

Hongren Zheng
zhenghr22@mails.tsinghua.edu.cn

2022-08-27

Section 1

# Background

# Motivation

- I was working on a project called OpenRigil
  - Open Source RISC-V cryptographic hardware token
  - Now public at `https://github.com/OpenRigil`
  - Target to be one alternative to Yubikey
  - Highlight: it is all in Chisel
- Decide to use Rocket Chip as the base system
- But, Rocket Chip lacked the support of Scalar Crypto extension (Zk)
  - To meet my requirement, I added it
- I also added the support of Bitmanip extension (Zb)
  - Zbk* and Zb* overlap a lot
  - So I added it with small efforts

# RISC-V Background

- RISC-V is an open standard ISA
  - First developed in Berkeley around 2010
  - Unlike proprietary/private standards like x86/ARM
  - Now it is widely adopted and has many ISA extensions
- In Autumn 2021, RISC-V ratified Scalar Crypto and Bitmanip Extension
  - Zk extension
  - Zb extension
  - One special note: there is no "K" extension or "B" extension

# RISC-V Zk

- Crypto Bitmanip
  - Zbkb: common bitmanip for crypto, e.g. rotation/byte-reverse
  - Zbkc: carry-less multiplication
  - Zbkx: crossbar (xbar) permutation
- Zkn: NIST cipher suite
  - Zkne/Zknd: AES enc/dec
  - Zknh: hash function SHA256/SHA512
- Zks: ShangMi cipher suite
  - Zksed: SM4 enc/dec
  - Zksh: hash function SM3
- Zkr: Entropy Source Extension
- Zkt: Data Independent Execution Latency

# RISC-V Zb

- Zbb: Basic bitmanip, similar to Zbkb
- Zbc: carry-less multiplication, Similar to Zbkc
- Zba: Useful arithmetic operations (no official name)
- Zbs: single bit instructions

# Chisel: the HDL

- Chisel is a modern language
  - Used many higher-order function in my design
  - Clearly describe the circuit
- Chisel is for parameterizable circuit
  - In our project, designs could be reused between RV32 and RV64
- Chisel util are useful
  - e.g. for rotation, I could just use `rotateRight`
  - instead of combining << and >>

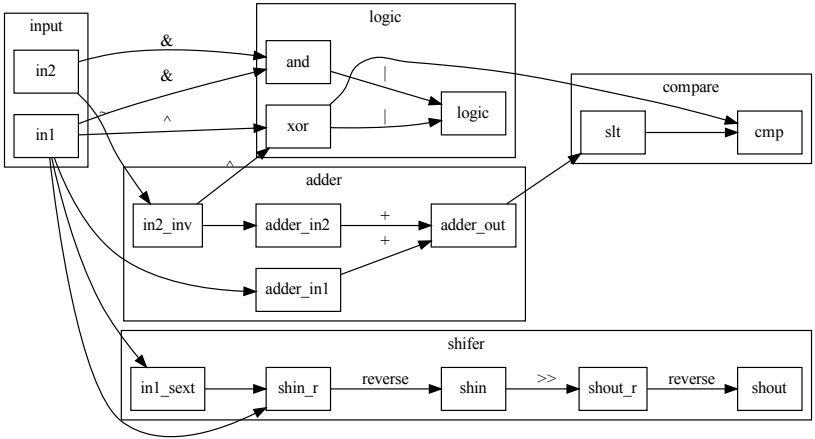
Section 2

# Designs

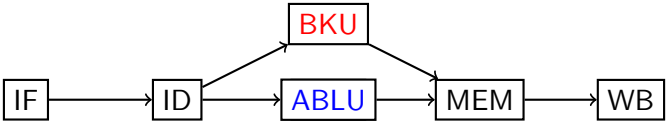
# Design goal

- Rocket Chip: Area
  - Its ALU is really small (see next slide)
  - Reuse as many datapaths as possible
- In comparison, XiangShan: Frequency
  - Big core
  - Focus on performance

# Rocket Chip ALU

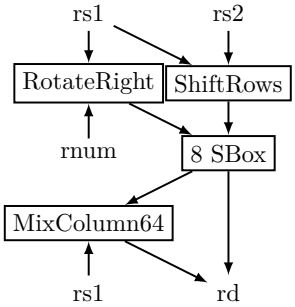


# Architecture Overview

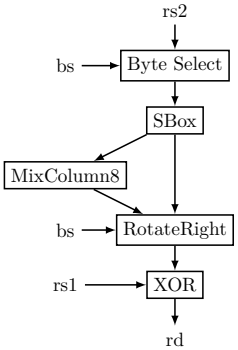



- Classical five stages: IF, ID, EXE, MEM, WB
- EXE means Execution, it often contains ALU (Arithmetic Logic Unit)
- My work: in EXE stage
  - Add BKU (Bitmanip Crypto Unit)
  - Replace ALU with ABLU (Arithmetic Bitmanip Logic Unit)

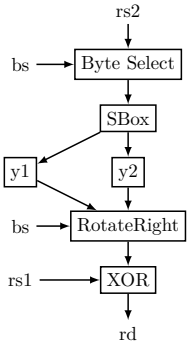
# K in BKU: Overview




(a) AES for RV64




(b) AES for RV32




(c) SM4

- Merged several instructions into one datapath
- Reuse common module between RV32 and RV64 for AES

# AES review

- Multi-round encryption
  - Input: 16 bytes, view as a 4x4 matrix
  - Row Shift: shift one row
  - SBox: substitute every byte
  - Mix Column: matrix multiplication
  - XOR with round key
- Key expansion
  - Generate round key with initial key via some SBox and XOR

# Reuse across RV32/RV64

```
class MixColumn8(enc: Boolean) {
  val out = if (enc) ... else ...
}

class MixColumn64(enc: Boolean) {
  VecInit(io.in.asBools.grouped(32).map(VecInit(_).asUInt).map({
    ...
    val m = Module(new MixColumn8(enc))
  }
}
```

- Mixcolumn
  - RV32, operate on 1 byte (`Mixcolumn8`)
  - RV64, operate on 16 bytes (`Mixcolumn64`)
  - Direction as parameter: encryption and decryption
  - Used `Mixcolumn8` to build `Mixcolumn64`
  - higher-order functions are helpful

# Reuse across RV32/RV64

```
class CryptoNIST(xLen: Int) {
  val aes = if (xLen == 32) { ... } else { ... }
}
```

- Top module: `xLen` as parameter
  - Generate different RTL based on `xLen`

# Another example: GFMul

```
class GFMul(y: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  io.out := VecInit(
    (if ((y & 0x1) != 0) Seq(   (io.in)) else Nil) ++
    (if ((y & 0x2) != 0) Seq( xt(io.in)) else Nil) ++
    (if ((y & 0x4) != 0) Seq(xt2(io.in)) else Nil) ++
    (if ((y & 0x8) != 0) Seq(xt3(io.in)) else Nil)
  ).reduce(_ ^ _)
}
```

- In AES we only need to multiply a constant $y$ in Galois field
- But several constants are needed, so how about a parameterized module
- Another level of meta: UInt from Chisel (circuit) and Int from Scala

# B in BKU: Zbc/Zbkc/Zbkx

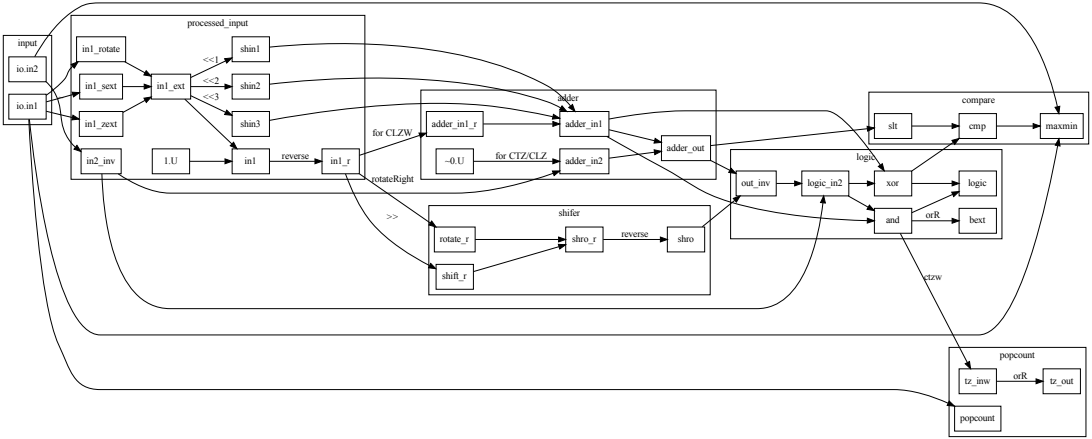
```
val clmul = clmul_rs2.asBools.zipWithIndex.map({
  case (b, i) => Mux(b, clmul_rs1 << i, 0.U)
}).reduce(_ ^ _)(xLen-1,0)

val xperm8 = VecInit(rs2_bytes.map(
  x => Mux(x(7,log2Ceil(xLen/8)).orR,
    0.U(8.W), rs1_bytes(x)) // return 0 when x overflow
).toSeq).asUInt
```

- This is much easier to understand
- Verilog example: see chipsalliance/rocket-chip#2906
  full of indices (or used generate)

# ABLU

- ABLU: merge common logic of bitmanip into ALU
  - Some logic could be reused, for example
  - $\sim$`b` (from substraction in adder)
  - `Reverse` (from shift left in shifter)
- ANDN
  - ANDN: `a & ~b`
  - Can just be implemented along side AND: `a & b`
  - Reuse $\sim$`b`
  - Result: `a & Mux(~b, b)`
  - Reusing 64 and gates
- ROR and ROL
  - Reuse `Reverse` in ALU for ROL
- CPOP
  - just use `PopCount` from `chisel3.util`!

# ABLU diagram

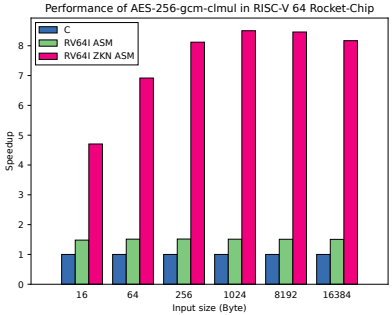


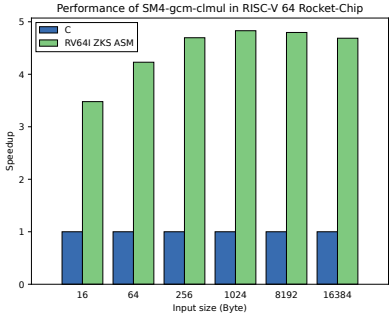
Section 3

# Evaluation

# Speedup

- Running in xc7k325tffg900-2 FPGA, 100 MHz
- Baseline: software-only OpenSSL
- Target: Hardware accelerated OpenSSL
- For RV64, up to **10X** for AES, 5X for SM4
- For RV32, up to 4X for AES, 3.7X for SM4



# Area

- ZKN and ZKS: the size of a multipler/divider

| Module | Area | Area, RV32 |
|---|---|---|
| Rocket | 67377 | 36346 |
| ALU | 1721 | 791 |
| ABLU | 4309 | 1953 |
| xperm/clmul | 7612 | 2008 |
| DIV | 8015 | 3107 |
| ZKN | 6804 | 1829 |
| ZKS | 709 | 707 |