Section 1

Zk Chisel

Subsection 1

Background

Hardware crypto acceleration

- Crypto operations is pervasive for modern apps
  - HTTPS: when you are browsing websites
  - Bitlocker: encrypt you disk
- Imagine you are watching a 4K video from encrypted network/disk
  - Software-only can not handle this load
  - need specialized hardware!

Different specialized hardware

- Accelerator: NN accelerator, Crypto co-processor
  - over network: cloud
  - over PCIe: GPU, SmartNIC, FPGA
  - inside SoC: NPU co-processor, SGX, ORAM controller
  - In core: special ALU, crypto unit
- Note the latency and throughput
- In-core is quite RISC, using peripheral as a black box is CISC-alike
  - Hardware is all about State machine
  - RISC: move state machine to software
  - Problem: frequency, code size (decoding, icache)
  - Maybe that's why we need Vector extension
- Programming interface from CPU perspective
  - MMIO(PCIe, SoC, etc): normal load/store
  - ISA extension: special instructions or even special regs
- Why NN needs dedicated hardware but crypto needs special instructions

RISC-V: the ISA

- RISC-V is an open standard ISA
  - First developed in Berkeley around 2010
  - Unlike proprietary/private standards like x86/ARM
  - Detailed in my seminar talk
- RISC-V recently ratified the scalar cryptography extension
  - In Autumn 2021
  - Covers block cipher/hash function: AES/SHA/SM4/SM3
  - Not covering RSA/ECC/ZKP (second part)

RISC-V Zk

- Crypto Bitmanip
  - Zbkb: rotation
  - Zbkc: carry-less multiplication
  - Zbkx: xperm4, xperm8 (detailed in next slide)
- Zkn: NIST cipher suite
  - Zkne/Zknd: AES enc/dec
  - Zknh: special operation in SHA256/SHA512
- Zks: ShangMi cipher suite
  - Zksed: SM4 enc/dec
  - Zksh: special operation in SM3

xperm8

- xperm8 rd, rs1, rs2
- The xperm8 instruction operates on bytes.
- The rs1 register contains a vector of XLEN/8 8-bit elements.
- The rs2 register contains a vector of XLEN/8 8-bit indexes.
- The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds

No Timing side-channel SBox

```
uint64_t sbox_bitmanip_xperm(uint64_t v) {
  uint64_t r = 0, m = 0;
  for (int i = 0; i < 32; i++) {
    r |= _rv64_xperm_b(sbox_packed[i], v ^ m);
    m += 0x0808080808080808LL;
  }
  return r;
}
```

- From claire[1]
- Suppose `v = 0xff01`, and the SBox is 256 byte long
- The first byte of `r` would be set in first round
- The second byte of `r` would be set in the last round

[1] `http://svn.clairexen.net/handicraft/2020/lut4perm/demo02.cc`

Chisel: the HDL

- To implement hardware, we need a hardware description language (HDL)
- Chisel is an HDL based on Scala
- Commonly used by RISC-V community
- Its syntax is more flexible compared with Verilog
  - Parameterized modules (code once, instantiate everywhere)
  - higher-order function like `map`/`reduce`/`filter`

Code snippet (xperm)

```
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 more easy to understand
- Verilog example: see chipsalliance/rocket-chip#2906
  full of index (or use generate)

Subsection 2

Designs

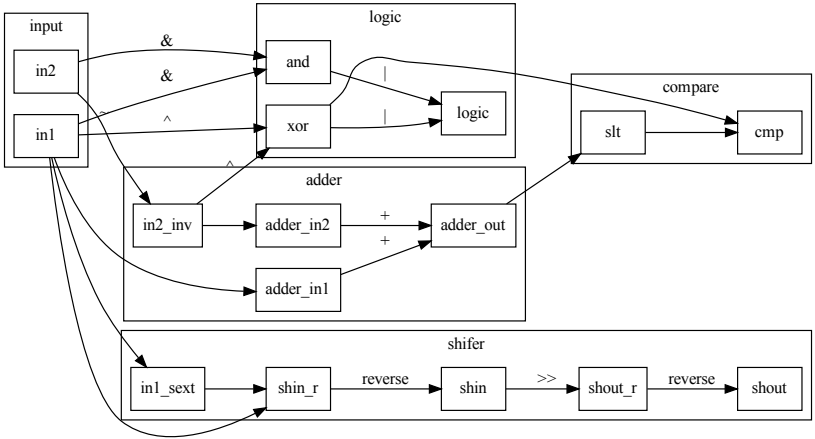
Circuit Design based on Rocket Chip

- To implement an extension, we must have a core
- Rocket Chip as the core
  - Open source RISC-V core, 2.2k stars
  - Widely used by research/industry
  - Taped out by Sifive
- Upstreamed my design to rocket chip
  - Attended several rocket chip working group meetings to discuss this PR
  - Got feedback from Andrew Waterman, the designer of RISC-V
  - Heated discussion on this (64 comments as shown below)

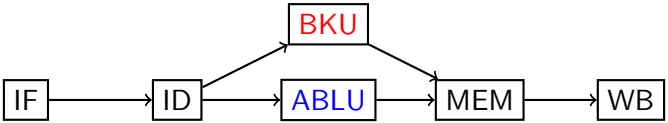
Design goal

- XiangShan: Frequency
  - XiangShan
  - ALU example
  - BKU example
- Rocket Chip: Area
  - Its ALU is really small
  - Addition/Subtraction
  - Left/Right Shift
  - Comparison
  - Reuse many datapaths

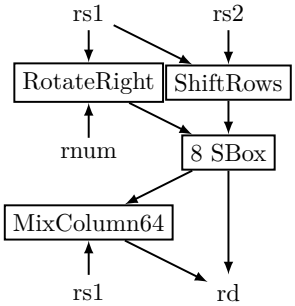
Rocket Chip ALU



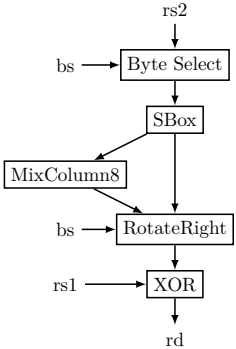
Five stage pipeline



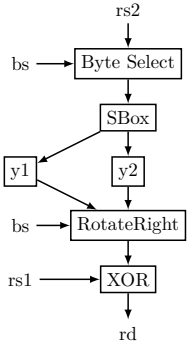
- Classical five stages: IF, ID, EXE, MEM, WB
- EXE means Execution, it often contains ALU (Arithmetic Logic Unit)
- My work: in EXE stage
  - Add BKU (Bitmanip Crypto Unit)
  - Replace ALU with ABLU (Arithmetic Bitmanip Logic Unit)
  - Why single cycle (After evaluation)



(a) AES for RV64



(b) AES for RV32



(c) SM4

- Merged several instructions into one datapath
- Reuse common module between architecture (thanks to Chisel)

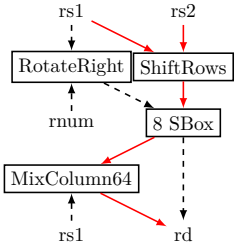
AES review

- Multi-round encryption
  - Input: 16 bytes, view as a 4x4 matrix
  - Row Shift: shift one row
  - SBox: substitute every element
  - Mix Column: matrix multiplication
  - XOR with round key
- Decryption: should be the reverse process
  - Row Shift and SBox can change order
  - XOR and Mix Column can change order
  - But with Mix Column on the round key
  - Thus still the same procedures
- Key expansion
  - Generate round key with initial key via some SBox and XOR
  - For dec, need Mix Column

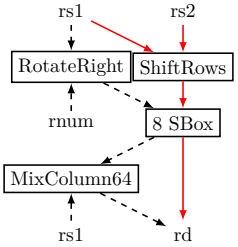
Key ISA design in AES

- There are multiple ways to design ISA
  - One instruction for one step: waste of instruction encoding space
  - One instruction for all steps: latency? CISC?
  - One instruction for some steps
- RISC-V
  - One instruction for Row Shift, SBox and Mix Column
  - Reuse ordinary XOR
  - Reuse ordinary LD/ST
  - Reuse ordinary regs
- Unlike accelerator design!

AES64



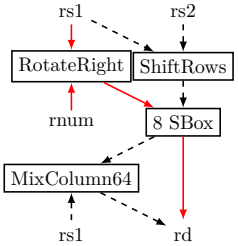
(a) aes64esm, aes64dsm



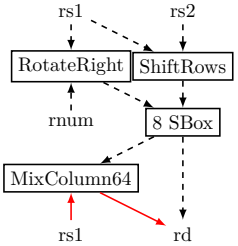
(b) aes64es, aes64ds

Enc/Dec in the same datapath

AES64 (cont'd)

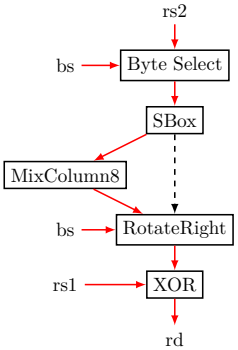


(a) aes64ks1i

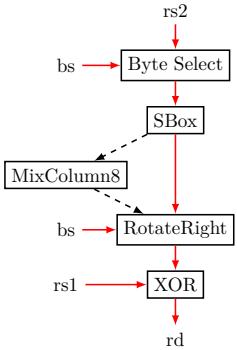


(b) aes64im

AES32

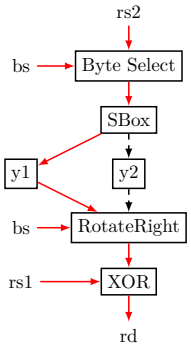


(a) aes32esmi, aes32dsmi

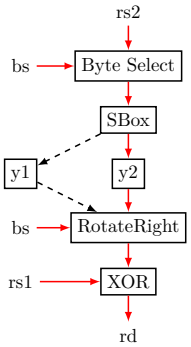


(b) aes32esi, aes32dsi

SM4



(a) sm4ed



(b) sm4ks

- SM4 and AES32 can merge datapath, decrease area for RV32
- They even share SBox[2]

[2] `https://github.com/mjosaarinen/sm4ni`

GFMul

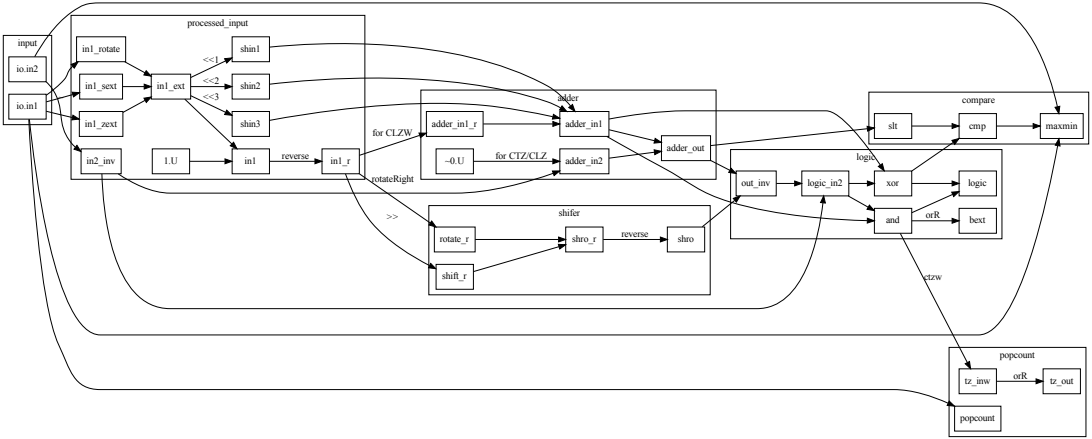
```
class GFMul(y: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  require(y != 0)
  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(_ ^ _)
}
```

- Another level of meta: UInt for Chisel (circuit) and Int for Scala
- Similar to perlasm below

ABLU

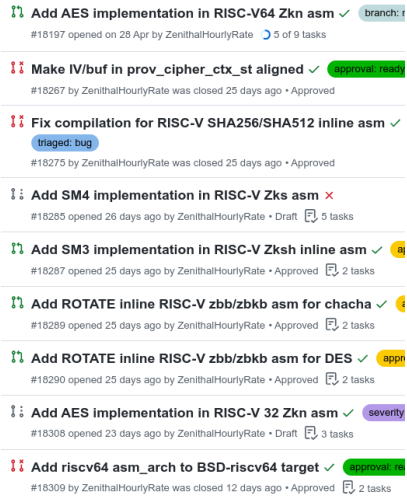
- ABLU: merge common logic of bitmanip into ALU
- ANDN
  - ANDN: `a & ~b`
  - Can just be implemented along side AND: `a & b`
  - Result: `a & Mux(~b, b)`
  - Reusing 64 and gates
- ROR and ROL (reuse the shift logic above)
- CLZ/CTZ

New ALU


input
io.in2
io.in1
processed_input
in1_rotate
in1_sext
in1_zext
in2_inv
in1_ext
<<1
<<2
<<3
shin1
shin2
shin3
1.U
in1
reverse
in1_r
for CLZW
rotateRight
>>
adder
adder_in1_r
adder_in1
~0.U
for CTZ/CLZ
adder_in2
adder_out
shifer
rotate_r
shift_r
shro_r
reverse
shro
compare
slt
cmp
maxmin
logic
out_inv
logic_in2
xor
and
orR
logic
bext
clzw
popcount
tz_inw
orR
tz_out
popcount

Software Design based on OpenSSL

- To evaluate my circuit design, I programmed corresponding software
- In **Assembly** language
  - performant crypto needs hand-written asm
- Exploit crypto ISA extension
  - Details and examples in my thesis
- Based on OpenSSL
  - widely-used crypto software lib
- Upstreamed my design to OpenSSL
  - 9 PRs, 3 merged



AES64

```
# Q0 and Q1 store a state
aes64esm $Q2,$Q0,$Q1
# change the order of source reg
aes64esm $Q3,$Q1,$Q0
# use ordinary instructions
ld       $T0,0($KEYP)
ld       $T1,8($KEYP)
xor      $Q0,$Q2,$T0
xor      $Q1,$Q3,$T1
# Q0 and Q1 now store a new state
```

AES64 key scheduling

```
# T0 and T1 store a round key
# ks1i for SBox and RCON XOR
aes64ks1i    $T2,$T1,$rnum
# ks2 for XOR
aes64ks2     $T0,$T2,$T0
aes64ks2     $T1,$T0,$T1
# T0 and T1 now a new state

# if we need round key for decryption
# inverse mixcolumn from T0, T1 and store them
# T0 and T1 are used in next round
aes64im      $T2,$T0
sd           $T2,0($KEYP)
aes64im      $T2,$T1
sd           $T2,8($KEYP)
```

AES32

```
# aes32esmi4 $key,$s0,$s1,$s2,$s3 is a shorthand all four inst below
# outer product and implicit row shift
aes32esmi $key,$key,$s0,0
aes32esmi $key,$key,$s1,1
aes32esmi $key,$key,$s2,2
aes32esmi $key,$key,$s3,3

# Q0~Q3 stores a whole
# note the difference between 64bit and 32bit
aes32esmi4 $T0,$Q0,$Q1,$Q2,$Q3
```

AES32 key scheduling

```
# invm4 $rd,$tmp,$rs is a shorthand for al 6 inst below
# aes32esi4 SBox on Input, aes32dsmi reverse SBox and mix column
# the result is only mix column
li          $tmp,0
li          $rd,0
aes32esi4   $tmp,$rs
aes32dsmi  $rd,$rd,$tmp,0
aes32dsmi  $rd,$rd,$tmp,1
aes32dsmi  $rd,$rd,$tmp,2
aes32dsmi  $rd,$rd,$tmp,3
```

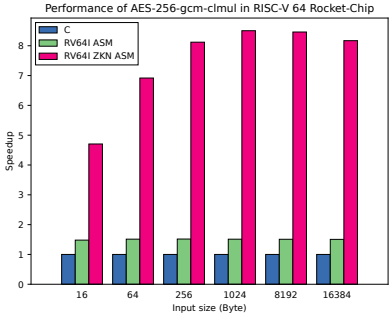
Why bother? Becuase RV32 does not need that much performance

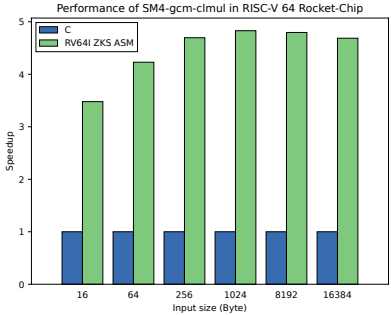
Subsection 3

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

Thoughts on this project

- The design of this ISA extension is SOTA
  - Algorithm itself, software, hardware
- ISA study is not easy...do not do it!
  - Only useful in industry, not much methodology...
  - Take a long time to be implemented (every can design one, but then?)
- Infra
  - UCB: rocket chip, Chisel, BOOM, etc...
  - ETH: pulp-platform
  - ICT: XiangShan
  - No our own infra: yes it is hard to maintain