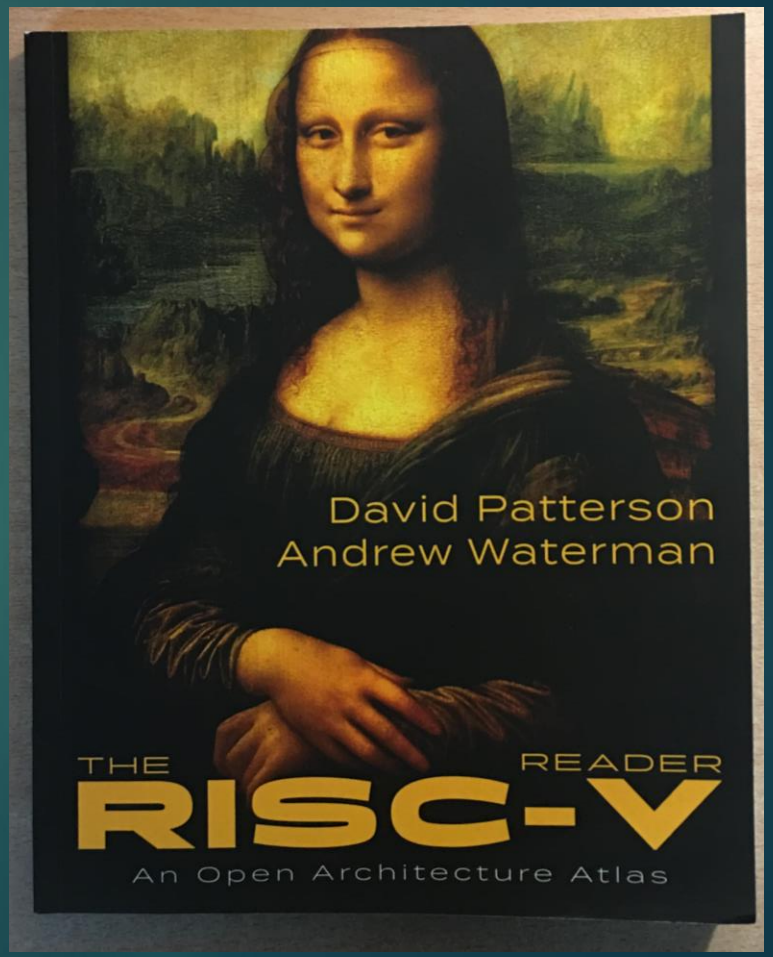
RISC-V

Simplicity is the ultimate sophistication.

—Leonardo da Vinci



自我介绍

- 个人主页：https://twd2.me
- GitHub：https://github.com/twd2

	Open Source	Closed Source
Compiler	GCC	ICC, …
OS	Linux	Windows, …
ISA	?	x86, ARM, …

- Why open source compilers and operating system but not ISAs?
- What if there were free and open ISAs we could use for everything?

RISC-V Origin Story

- UC Berkeley Research using x86 & ARM?
  - Impossible – too complex *and* IP issues
- 2010 started "3-month project" to develop own clean-slate ISA
  - Krste Asanovic, Andrew Waterman, Yunsup Lee, Dave Patterson
- 4 years later, released frozen base user spec

Why are outsiders complaining about changes of RISC-V in Berkeley classes?

中科院计算所面临的选择

- 研究人员想为系统资源加上标签，为进程区分服务。
- x86？过于复杂，IP核授权基本不可能。
- ARM？比较复杂，IP核授权很贵。
- MIPS？龙芯可以提供研究用源代码，很工程，基本改不动。
- Xilinx FPGA提供的IP核也由于某些原因不适合使用。

RISC-V (pronounced risk-five)

- 缩写为RV，是一个开放的精简指令集架构。注意，并不是实现。
- 32位的RISC-V简称RV32，64位和128位同理。

What's Different About RISC-V?

- Simple
  - Far smaller than proprietary ISAs
  - 2500 pages for x86, ARMv8 manual vs 200 for RISC-V manual
- Clean-slate design
  - 25 years later, so can learn from mistakes of predecessors
  - Avoids µarchitecture or technology-dependent features
- Modular
  - Small standard base ISA
  - Multiple standard extensions
- Supports specialization
  - Vast opcode space reserved
- Community designed
  - Base and standard extensions finished
  - Grow via optional extensions vs. incremental required features
- RISC-V Foundation extends ISA for technical reasons
  - vs. private corporation for internal (marketing) reasons

Specifications

- **Volume I: User-Level ISA 2.2 (frozen)**
- Volume II: Privileged Architecture 1.10 (draft)
- RISC-V External Debug Support 0.13.1

RISC-V Base Plus Standard Extensions

- A few base integer ISAs
  - RV32E, RV32I, RV64I
    - RV32E is 16-reg subset of RV32I
  - <50 hardware instructions in base (Similar to RISC-I!*)
- Standard extensions
  - M: Integer multiply/divide
  - A: Atomic memory operations
  - F/D: Single/Double-precision Fl-point
  - C: Compressed Instructions (<x86)
  - V: Vector Extension for DLP (>SIMD**)

G (general) = IMAFD

- Standard RISC encoding in fixed 32-bit instruction format
- Supported forever by RISC-V Foundation

* "How close is RISC-V to RISC-I?" David Patterson, 9/19/17, ASPIRE Blog

** "SIMD Instructions Considered Harmful," David Patterson and Andrew Waterman, 9/18/17

I: Base Integer ISA

- All instructions are 32 bits long?

Registers and Calling Convention

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Registers and Calling Convention (cont.)

```
entry_label:
  addi sp,sp,-framesize     # Allocate space for stack frame
                            # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)   # Save return address (ra register)
  # save other registers to stack if needed
  ... # body of the function

# restore registers from stack if needed
lw   ra,framesize-4(sp)   # Restore return address register
addi sp,sp, framesize     # De-allocate space for stack frame
ret                       # Return to calling point
```

I: Base Integer ISA (cont.)

RV32I

Integer Computation

add {– immediate}

subtract

{and / or / exclusive or} {– immediate}

{shift left logical / shift right arithmetic / shift right logical} {– immediate}

load upper immediate

add upper immediate to pc

set less than {– immediate} {– unsigned}

Control transfer

branch {equal / not equal}

branch {greater than or equal / less than} {– unsigned}

jump and link {– register}

Loads and Stores

{load / store} {byte / halfword / word}

load {byte / halfword} unsigned

Miscellaneous instructions

fence loads & stores

fence instruction & data

environment {break / call}

control status register {read & clear bit / read & set bit / read & write} {– immediate}

Integer Computation

- `add`, `sub`, `and`, `or`, `xor`, `sll`, `sra`, `srl`: these are trivial.
- `slt`, `sltu`: set if less than.

lui: Load Upper Immediate

- lui (load upper immediate) is used to build 32-bit constants.
- usually used with addi.
  - # t0 = 0xaaaaaaab

```
lui t0, 0xaaaab
addi t0, t0, 0xaab
```

auipc: Add Upper Immediate to PC

- auipc (add upper immediate to pc) is used to build pc-relative addresses.
- The program counter pc holds the address of the **current** instruction.
- usually used with addi, jalr.
  - # call printf, which is far away

```
auipc ra, %hi(printf)
jalr ra, (%lo(printf))(ra)
```

Loads and Stores

- RISC-V chose **little-endian** byte ordering.
- Misaligned accesses are supported but slow and not guaranteed to execute atomically.
- `lb, lh, lw, lbu, lhu, sb, sh, sw`

Conditional Branch

- No infamous branch delay slot.
- No condition codes.
- `beq, bne, blt, bge, bltu, bgeu`

Unconditional Jump

- `jal`: jump and link
  - `rd = pc + 4`
    `pc = pc + offset`
- `jalr`: jump and link (register)
  - `t = pc + 4`
    `pc = rs + offset`
    `rd = t`

Miscellaneous

- `fence` is used to order device I/O and memory.
- `fence.i` is used to synchronize the instruction and data streams.
- RISC-V uses memory mapped I/O.
- `ecall` is for system call.

M: **M**ultiply and Divide

- Two instructions for the high part and the low part of the product.
- Two instructions for the quotient and the remainder.
- Dividing by zero **does not** cause an exception.

RV32M

multiply

multiply **h**igh { – / **u**nsigned / **s**igned **u**nsigned }

{ **div**ide / **rem**ainder } { – / **u**nsigned }

A: Atomic

- Two types:
  - atomic memory operations, and
  - load reserved / store conditional
- Relaxed (aq, rl bit)

RV32A

atomic memory operation { add, and, or, swap, xor, maximum, maximum unsigned, minimum, minimum unsigned } .word

{ load reserved, store conditional } .word

A: Atomic (cont.)

- Sample code for compare-and-swap function using LR/SC.

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)          # Load original value.
    bne t0, a1, fail       # Doesn't match, so fail.
    sc.w a0, a2, (a0)      # Try to update.
    jr ra                  # Return.
fail:
    li a0, 1               # Set return to failure.
    jr ra                  # Return.
```

FD: Single / Double Floating Point

- FP Registers:
  - f0-f31
  - fcsr

RV32F and RV32D

Floating-Point Computation

float { add / subtract / multiply / divide / square root / minimum / maximum } { .single / .double }

float { – / negative } multiply { add / subtract } { .single / .double }

float move to .single from .x register

float move to .x register from .single

Comparison

compare float { equals / less than / less than or equals } { .single / .double }

Load and Store

float { load / store } { word / doubleword }

Conversion

float convert to { .single / .double } from .word { – / unsigned }

float convert to .word { – / unsigned } from { .single / .double }

float convert to .single from .double

float convert to .double from .single

Other instructions

float sign injection { – / negative / exclusive or } { .single / .double }

float classify { .single / .double }

C: Compressed Instructions

- RVC uses a simple compression scheme that offers shorter 16-bit instructions when:
  - the immediate or address offset is small, or
  - one of the registers is the zero register, the ABI link register, or the ABI stack pointer, or
  - the destination register and the first source register are identical, or
  - the registers used are the 8 most popular ones.

C: Compressed Instructions (cont.)

RV32C

Integer Computation

c.add { – / immediate }

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract

c. { shift left logical / shift right arithmetic / shift right logical } immediate

c.and { – / immediate }

c.or

c.move

c.exclusive or

c.load { – / upper } immediate

Loads and Stores

c. { – / float } { load / store } word { – / using stack pointer }

c.float { load / store } doubleword { – / using stack pointer }

Control transfer

c.branch { equal / not equal } to zero

c.jump { – / and link }

c.jump { – / and link } register

Other instructions

c.environment break

Other Extensions

- Q: Quad-Precision Floating-Point
- L: Decimal Floating-Point
- B: Bit Manipulation
- J: Dynamically Translated Languages
- T: Transactional Memory
- P: Packed-SIMD Instructions
- V: Vector Operations
- N: User-Level Interrupts
- ……

Hello, world!

```c
#include <stdio.h>
int main()
{
    printf("Hello, %s\n", "world");
    return 0;
}
```

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

RV64?

- 所有寄存器长度变为64位
- 原RV32中同名指令默认变为64位操作数
- 加入处理32位操作数的指令
- 加入处理64位操作数的访存指令

RV64

RV64C

Integer Computation

c.add {– / immediate} {– / word}

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract {– / word}

c. {shift left logical / shift right arithmetic / shift right logical} immediate

c.and {– / immediate}

c.or

c.move

c.exclusive or

c.load {– / upper} immediate

Loads and Stores

c. {– / ~~float~~} {load / store} {~~word~~ / doubleword} {– / using stack pointer}

c.float {load / store} doubleword {– / using stack pointer}

Control transfer

c.branch {equal / not equal} to zero

c.jump {– / ~~and link~~}

c.jump {– / and link} register

Other instructions

c.environment break

RV64?

- How to build arbitrary 64-bit constants?
- Hint:
  -
```
# t0 = 0xffffffffaaaaaaab
lui t0, 0xaaaab
addi t0, t0, 0xaab
```

```
➜  bin ./riscv64-unknown-elf-as
li t0, 0x1234567800000000
➜  bin ./riscv64-unknown-elf-objdump -d a.out

a.out:     文件格式 elf64-littleriscv


Disassembly of section .text:

0000000000000000 <.text>:
   0:   024692b7                lui     t0,0x2469
   4:   acf2829b                addiw   t0,t0,-1329
   8:   02329293                slli    t0,t0,0x23
➜  bin
```

```
➜  bin ./riscv64-unknown-elf-as
li t0, 0x1234567887654321
➜  bin ./riscv64-unknown-elf-objdump -d a.out

a.out:     文件格式 elf64-littleriscv


Disassembly of section .text:

0000000000000000 <.text>:
   0:   002472b7                lui     t0,0x247
   4:   8ad2829b                addiw   t0,t0,-1875
   8:   00c29293                slli    t0,t0,0xc
   c:   f1128293                addi    t0,t0,-239 # 0x246f11
  10:   00d29293                slli    t0,t0,0xd
  14:   d9528293                addi    t0,t0,-619
  18:   00e29293                slli    t0,t0,0xe
  1c:   32128293                addi    t0,t0,801
➜  bin
```

RV128?

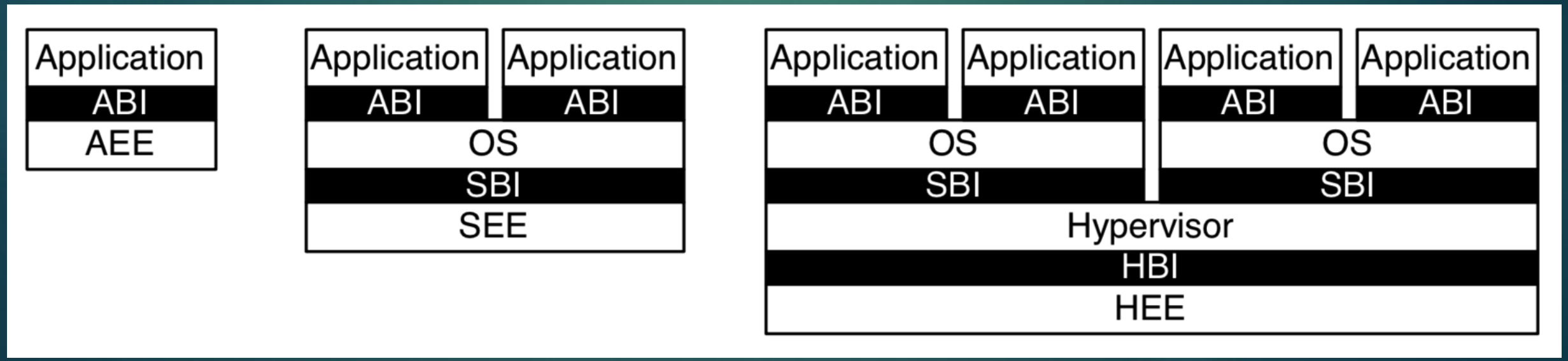
- 所有寄存器长度变为128位
- 原RV32中同名指令默认变为128位操作数
- 加入处理32位操作数的指令
- 加入处理64位操作数的普通指令和访存指令
- 加入处理128位操作数的访存指令

Specifications

- Volume I: User-Level ISA 2.2 (frozen)
- **Volume II: Privileged Architecture 1.10 (draft)**
- RISC-V External Debug Support 0.13.1

Privileged Architecture

- Different implementation stacks supporting various forms of privileged execution.



Privileged Architecture (cont.)

- RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	?
3	11	Machine	M

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Control and Status Registers

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.

Control and Status Registers (cont.)

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.

Control and Status Registers (cont.)

Machine Protection and Translation			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

CSR instructions

- `csrrw`: read and then write.
- `csrrs`: read and then set bit.
- `csrrc`: read and then clear bit.

sstatus: Status Register

- SIE: interrupt enable
- SPIE: previous interrupt-enable bit
- SPP: previous privilege mode
- SUM: can S-mode code access user memory?
- MXR: make executable as readable?
- UXL: U-mode ISA. RV32? RV64? RV128?

satp: Address Translation and Protection

31	30 … 22	21 … 0
MODE (**WARL**)	ASID (**WARL**)	PPN (**WARL**)
1	9	22

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

satp: Address Translation and Protection

63–60	59–44	43–0
MODE (**WARL**)	ASID (**WARL**)	PPN (**WARL**)
4	16	44

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

Virtual-Memory System

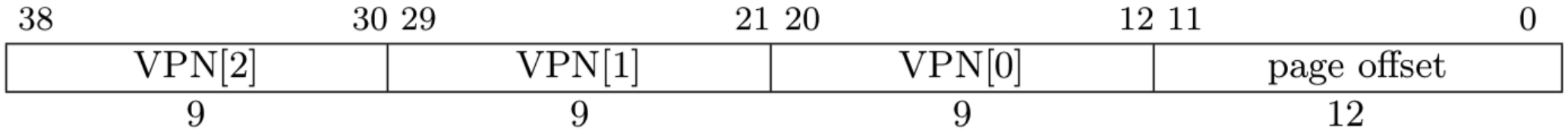


Figure 4.16: Sv39 virtual address.

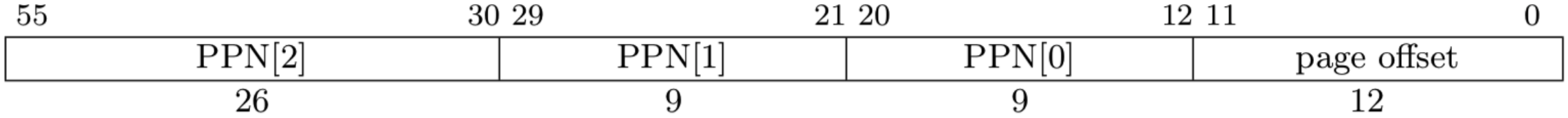


Figure 4.17: Sv39 physical address.

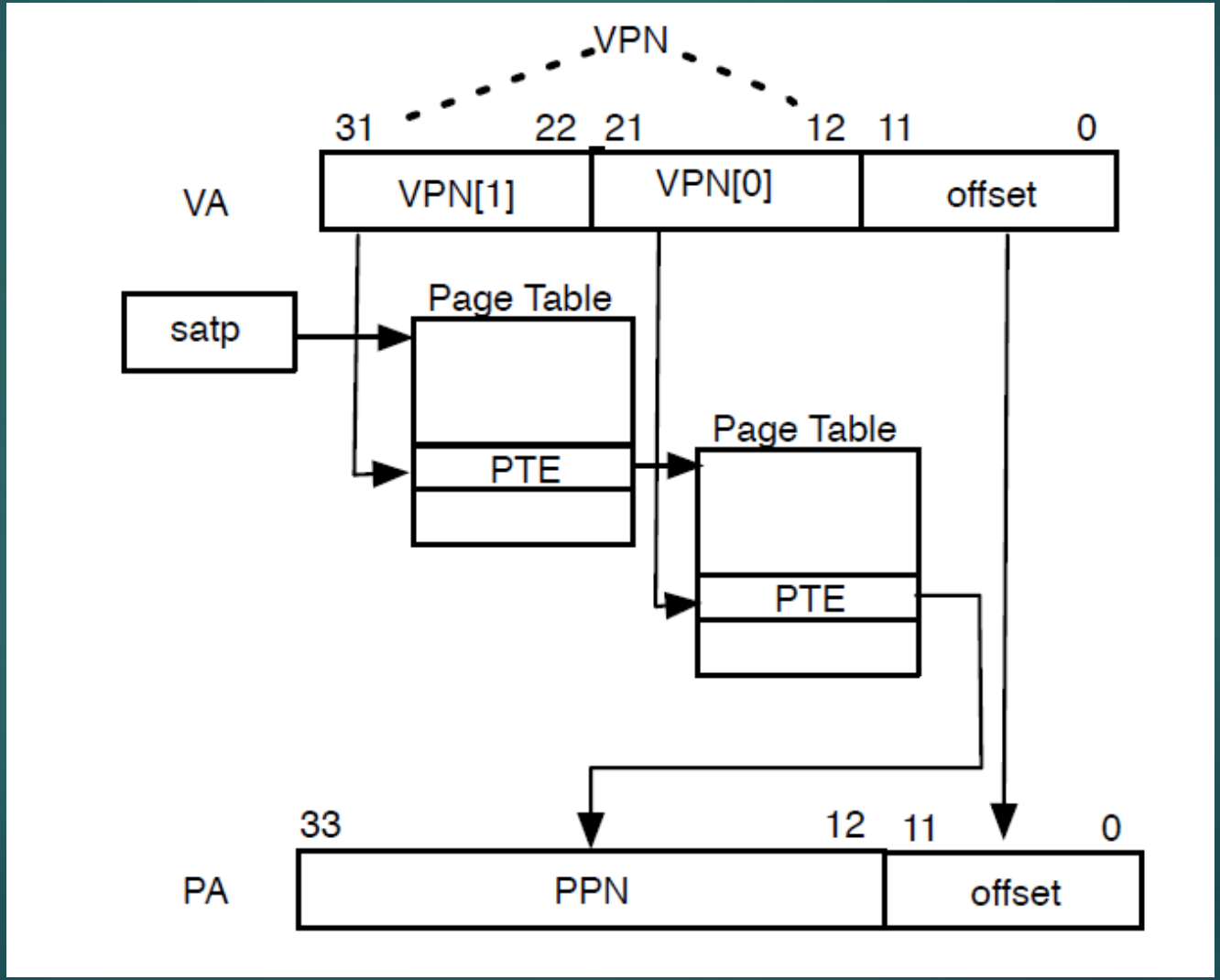
Virtual-Memory System (cont.)

63–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.18: Sv39 page table entry.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Virtual-Memory System (cont.)



Virtual-Memory System (cont.)

4.3.2 Virtual Address Translation Process

A virtual address va is translated into a physical address pa as follows:

1. Let a be `satp`$.ppn \times$ PAGESIZE, and let $i =$ LEVELS $- 1$. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)
2. Let pte be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE. (For Sv32, PTESIZE=4.) If accessing pte violates a PMA or PMP check, raise an access exception.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, $pte.x$, and $pte.u$ bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If $i > 0$ and $pa.ppn[i-1:0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, either raise a page-fault exception or:
   - Set $pte.a$ to 1 and, if the memory access is a store, also set $pte.d$ to 1.
   - If this access violates a PMA or PMP check, raise an access exception.
   - This update and the loading of pte in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.
8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i-1:0] = va.vpn[i-1:0]$.
   - $pa.ppn[\text{LEVELS}-1:i] = pte.ppn[\text{LEVELS}-1:i]$.

Kinds of Traps

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	$\geq$10	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	9–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	$\geq$16	*Reserved*

sip, sie

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
WIRI	SEIP	UEIP	**WIRI**	STIP	UTIP	**WIRI**	SSIP	USIP
XLEN-10	1	1	2	1	1	2	1	1

Figure 4.4: Supervisor interrupt-pending register (`sip`).

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
WPRI	SEIE	UEIE	**WPRI**	STIE	UTIE	**WPRI**	SSIE	USIE
XLEN-10	1	1	2	1	1	2	1	1

Figure 4.5: Supervisor interrupt-enable register (`sie`).

stvec: Trap Vector Base Address

XLEN-1 … 2	1 … 0
BASE[XLEN-1:2] (**WLRL**)	MODE (**WARL**)
XLEN-2	2

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

sepc, scause

- sepc: Exception Program Counter
- scause: Cause Register

`stval`: Trap Value Register

- faulting address, or
- faulting instruction bits

sscratch

medeleg, mideleg

- By default, all traps at any privilege level are handled in M-mode, though a M-mode handler can redirect traps back to the appropriate level with the `mret`.
- To increase performance, exceptions and interrupts can be processed directly by a lower privilege level.

Physical Memory Protection

- Physical memory is divided to several regions, and each region has its own permission.
- `pmpcfg, pmpaddr`

Privileged Architecture (cont.)

RV32/64 Privileged Instructions

{machine-mode / supervisor-mode} trap return

environment {break / call}

control status register {read & clear bit / read & set bit / read & write} {– / immediate}

supervisor-mode fence.virtual memory address

wait for interrupt

When a trap is taken

- `xepc = pc`
- `xstatus.xPP = privilege mode`
- `xstatus.xPIE = xstatus.xIE`
- `xstatus.xIE = 0`
- `pc = xtvec*`
- `privilege mode = x`
- where x is the trap handling privilege mode.

mret, sret: RETurn from trap

- pc = xepc
- privilege mode = xstatus.xPP
- xstatus.xIE = xstatus.xPIE
- where x = m or s

sfence.vma

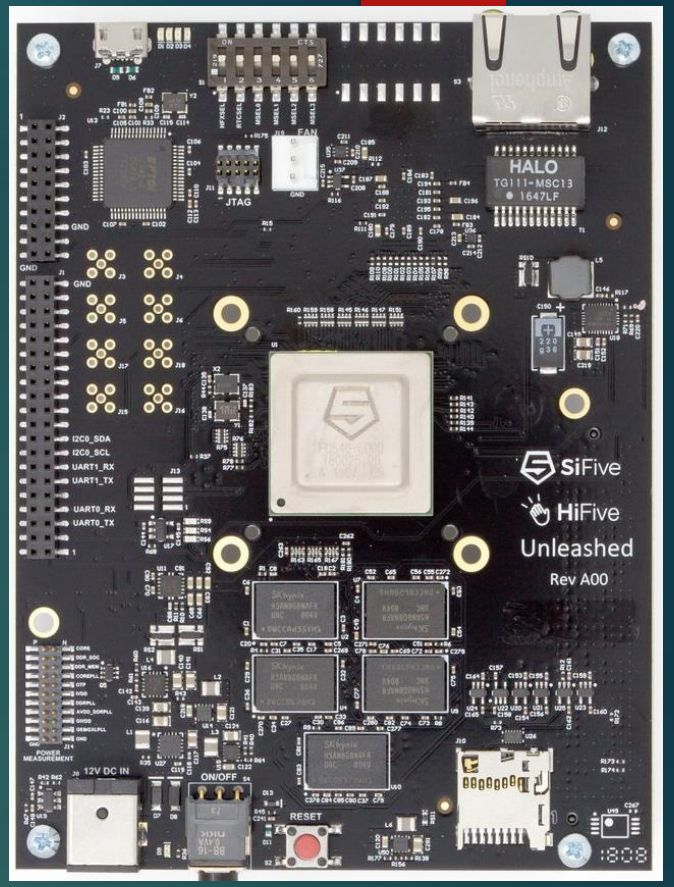
- sfence.vma is used to flush any **local** hardware caches related to address translation.
- How to do a TLB shootdown?
  - data fence, IPI, sfence.vma, IPI back

`wfi`: Wait For Interrupt

- The `wfi` instruction is just a hint, and a legal implementation is to implement `wfi` as a `nop`.

Implementations

- RTL Design
  - Rocket
  - BOOM
- SoC Board
  - HiFive1 (RV32IMAC, M-mode only)
  - HiFive Unleashed (RV64GC, M+S+U, 4 cores)
  - Kendryte K210 (RV64GC, M+S+U, AI, 2 cores)

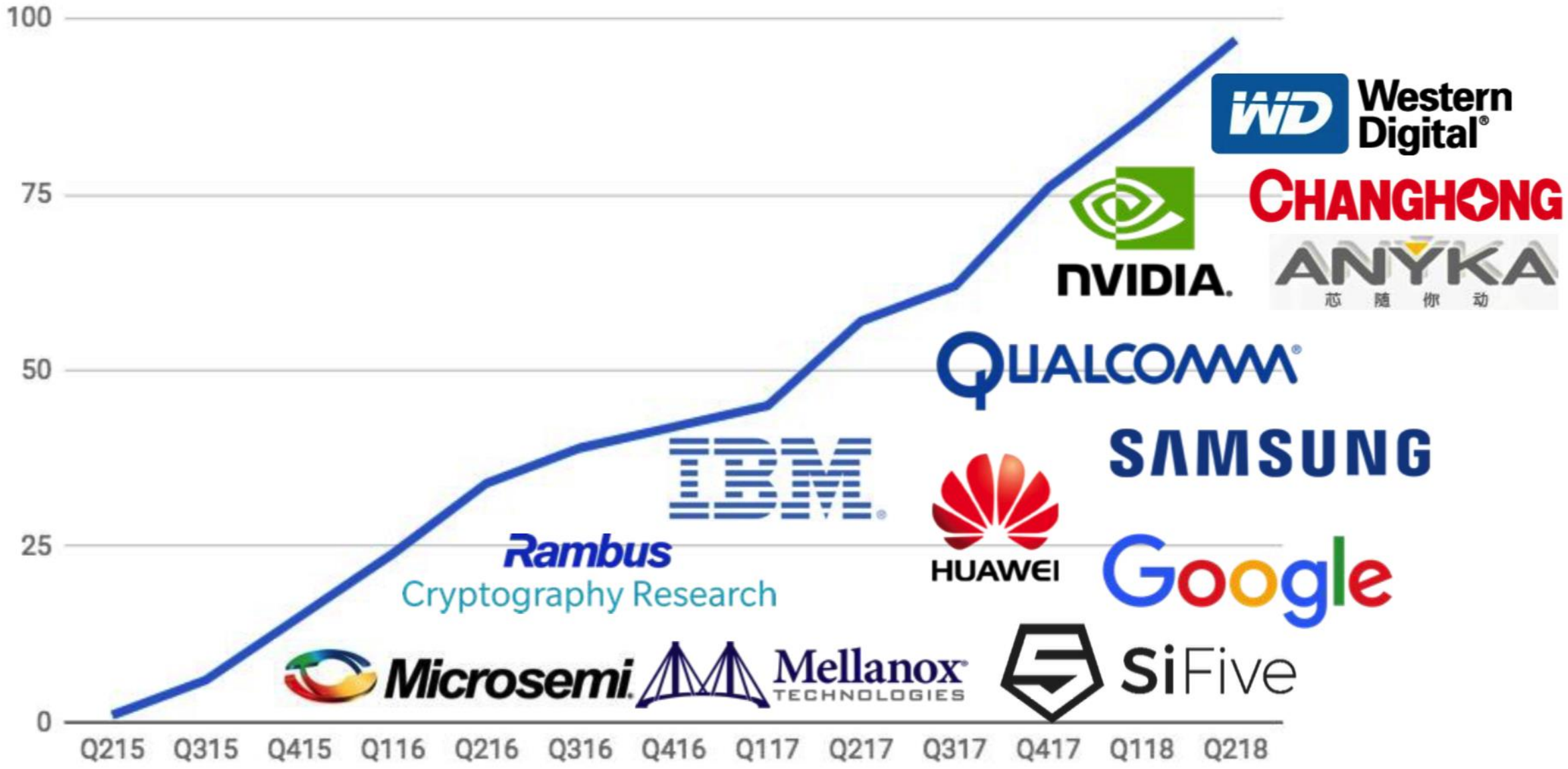


RISC-V Core	Z-scale	Rocket	BOOM
Description	32-bit 3-stage pipeline in-order 1-instruction issue L1 caches (≈ ARM Cortex-M0)	64-bit, FPU, MMU 5-stage pipeline in-order 1-instruction issue L1 & L2 caches (≈ ARM Cortex-A5)	64-bit, FPU, MMU 5-stage pipeline out-of-order 2-, 3-, or 4- instruction issue L1 &L2 caches (≈ ARM Cortex-A9)
Unique LOC	600 (40%)	1,400 (10%)	9,000 (45%)
LOC all 3 share	500 (30%)	500 (5%)	500 (5%)
LOC Z-scale & Rocket share	500 (30%)	500 (5%)	---
LOC Rocket & BOOM share	---	10,000 (80%)	10,000 (50%)
Total LOC	1,600	12,400	19,500

Research(es) on RISC-V

- TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V
  NDSS 2019

Foundation Members since 2015



Foundation Working Groups (partial list)

Bit Manipulation

AMD, Andes Technology, BAE Systems, Esperanto Technologies, ETH zürich, lowRISC, MediaTek, Microsemi, SiFive, Syntacore

Compliance

codasip, BAE Systems, bluespec, Microsemi, NVIDIA, SiFive, Syntacore, Berkeley Architecture Research

Debug

SiFive, Andes Technology, BAE Systems, bluespec, cortus, Google, NVIDIA, lowRISC, ROA Logic, Syntacore, VeriSilicon, ultraSOC

Memory Model

NVIDIA, IBM, Princeton University, VeriSilicon, lowRISC, MIT CSAIL, SiFive, Andes Technology, bluespec, BAE Systems, Western Digital, Syntacore

Privileged Spec

SiFive, Andes Technology, BAE Systems, Esperanto Technologies, bluespec, NVIDIA, Western Digital, lowRISC, Berkeley Architecture Research, MIT CSAIL, Syntacore

Vector

SiFive, Andes Technology, NVIDIA, Micron, Google, lowRISC, MediaTek, Microsemi, Esperanto Technologies, Syntacore, Berkeley Architecture Research, ETH zürich, Western Digital, BAE Systems

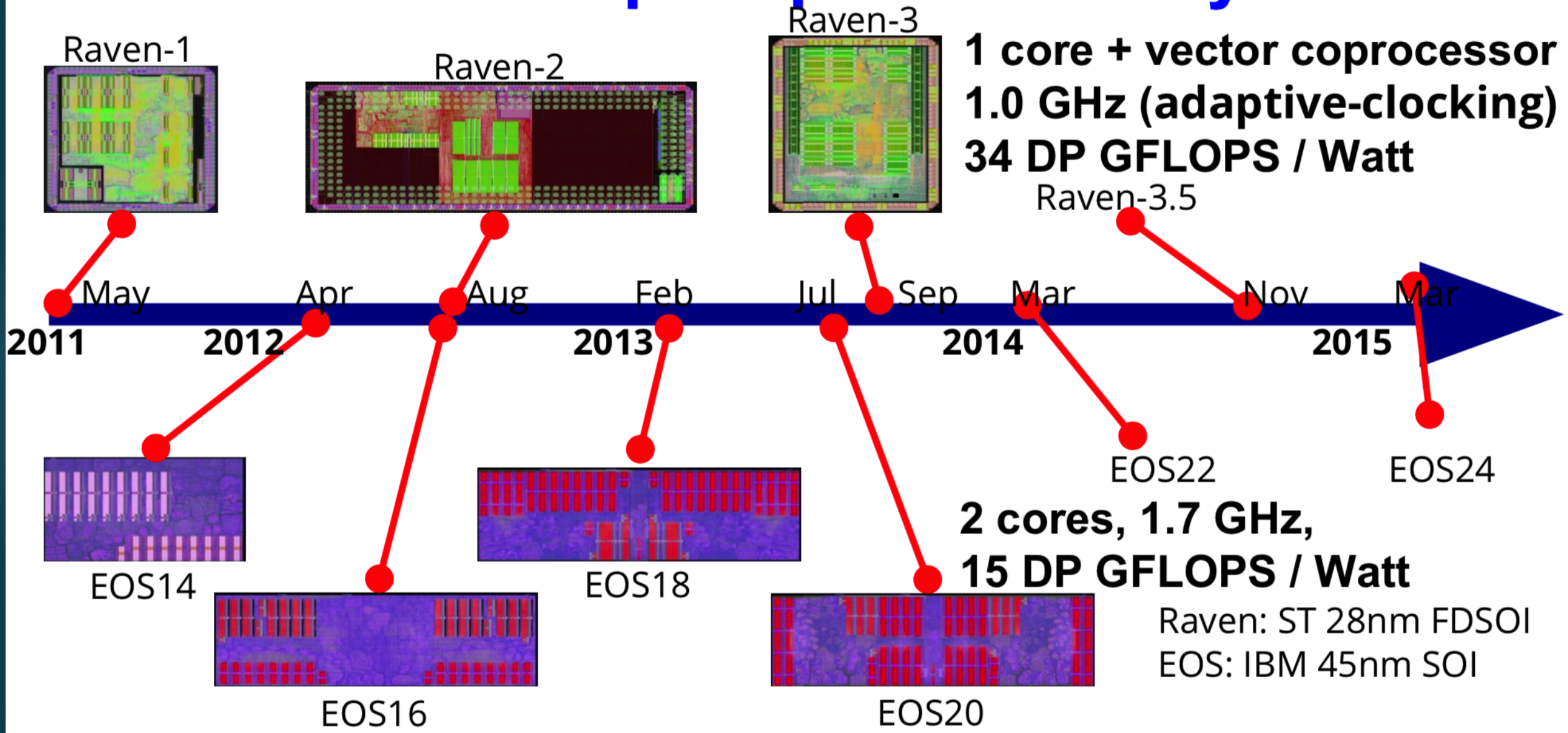
Security

NVIDIA, MediaTek, Microsemi, Huawei, Blockstream, SiFive, Esperanto Technologies, BAE Systems, Andes Technology, DRAPER, ultraSOC, AMD, NXP

Base ISA/Opcode

SiFive, Andes Technology, Esperanto Technologies, Western Digital, Syntacore, lowRISC, Berkeley Architecture Research, BAE Systems

Four 28nm & Six 45nm RISC-V Chips taped out in 5 years



Open Architecture Goal

Create industry-standard open ISAs for all computing devices

“Linux for processors”

相关资源

- 指令集规范：https://riscv.org/specifications/
- 芯片实现：https://riscv.org/risc-v-cores/
  - Rocket：https://github.com/freechipsproject/rocket-chip
  - BOOM：https://github.com/riscv-boom/riscv-boom
- 工具链：https://github.com/riscv/riscv-tools
- ABI：https://github.com/riscv/riscv-elf-psabi-doc

谢谢