RISC-V

Luke Hopkins

Instruction Set Architecture

- An instruction set architecture (ISA) is a contract between hardware and software that specifies the instructions that can be executed by a processor.
  - It is the vocabulary that software uses to talk to the hardware.
  - There are many possible ways of realizing the ISA in an implementation
    - Scalar, superscalar, in-order, out-of-order.
    - Different proportions of microcode versus hardware.
  - It permits the development of a software ecosystem by allowing developers to create tools like compilers, linkers, assemblers, and utilities that will work with any implementation that complies with the ISA. Protection of investment in software is the primary motivation for an ISA.
  - The first instruction set architecture was defined by IBM for the 360.
    - Defined to be machine independent.
      - Same ISA used on small and large machines.
      - Previously the instruction set was tied to a given machine. You couldn't run a program written for one machine on another.
    - Smaller less expensive machines had more microcode, larger machines had more hardware.
    - IBM has the longest-lived ISA: programs written for the IBM 360 can still be run on the latest mainframes unmodified, assuming you can emulate the I/O devices.

RISC History

- The concept of reduced instruction set computing grew out of optimizing compiler research conducted by IBM in the Hudson Valley.
- John Cocke led this research in the 1970s which culminated in the development of the IBM 801, widely recognized as the first RISC processor.
- David Patterson coined the term RISC in 1980.
- RISC was successful for several reasons
  - Instruction frequency in typical programs favors primitive instructions like loads, stores, branches, comparisons and simple arithmetic and logical operations.
  - Support for more complex instructions require additional levels of logic that complicate stages of the instruction processing pipeline resulting in longer delays.
  - The compiler knows details about the program that the CPU doesn’t and can optimize things that internal processor microcode can’t. The microcode must take a general approach to solving problems that doesn’t take into consideration what’s known at compile time.
  - Instead of having fixed routines of vertical microcode in the CPU, have the compiler generate the instructions tailored to the program and provide a good I-Cache.

RISC-V History

- David Patterson, also at UC Berkeley, did research on RISC computing during the 1980s.
  - RISC-I (1981)
  - RISC-II (1983)
  - RISC-III (1984) SOAR
  - RISC-IV (1988) SPUR
- 2010 Krste Asanovic, et al, at UC Berkeley, were doing research on the architecture of custom accelerators and needed an ISA to base their work on.
  - x86 couldn’t be licensed, ARM was encumbered by restrictive agreements that would have made it impossible to share their work.
  - Accelerator development is particularly important now that Moore’s law is in decline.
- Asanovic and his team ultimately decided to build on Patterson’s earlier work, and RISC-V was born. The first core was taped out in 2011.

RISC-V Fundamentals

- RISC-V is not an implementation it’s an open standard instruction set architecture.
  - There is no Verilog, VHDL, or microarchitectural detail in the specification.
  - It’s not a core, it just specifies the Hardware / Software contract.
  - If you build your core according to the standard, you get the software ecosystem for free.
  - Modular. Ratified parts don’t change.
- Because it is royalty free and licensed through Creative Commons Attribution it permits both closed source commercial implementations and freely distributed open-source implementations (license is up to the core implementor).
  - It’s not just the ISA, or RISC vs CISC, it’s the business model.
- Sanction free, not subject to export restrictions.
  - Chinese manufactures fear losing access to x86 or ARM.
- There is no single company that controls the ISA, and therefore it should have longevity.
  - SPARC, DEC VAX and PDP, Intel i960, etc.
- Are there patent issues?
  - AFAIK, nothing so far, but when you manufacture hardware, there’s significant money involved.
  - ISA will probably never be challenged, but an open-source core might be.

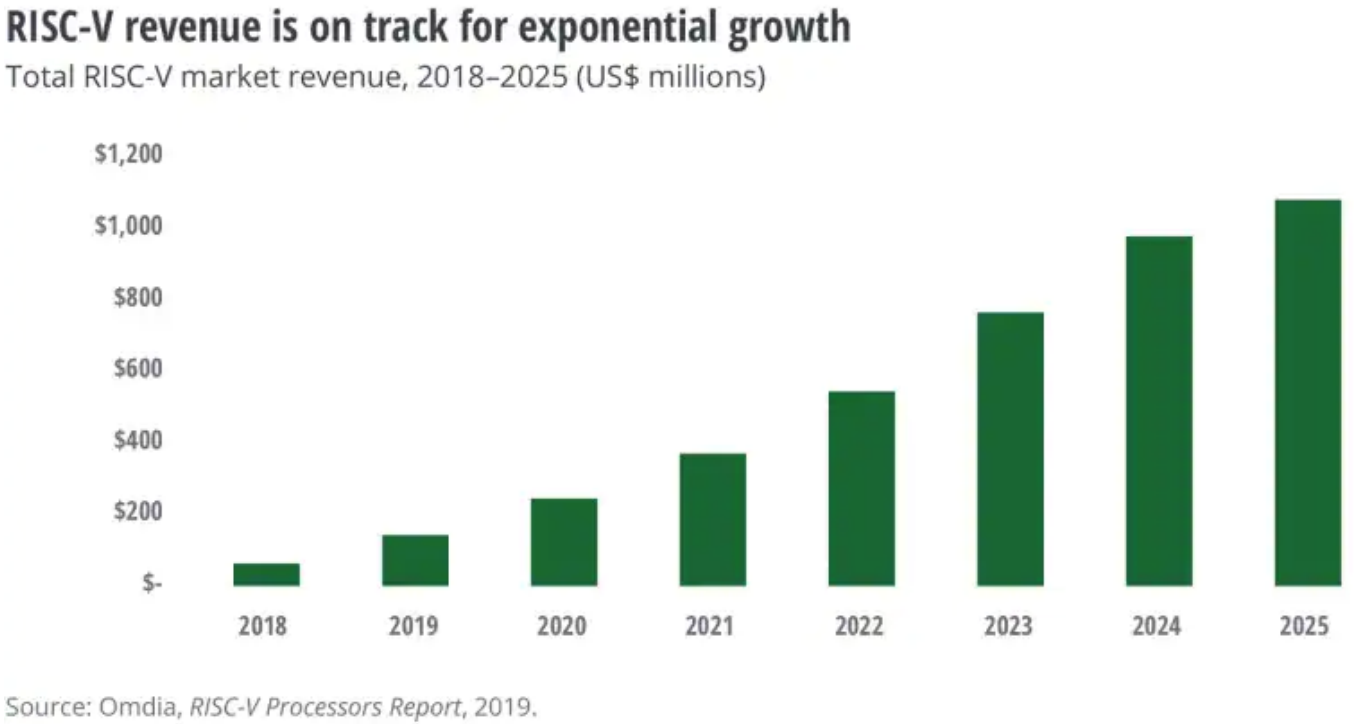
RISC-V International

- Originally called the RISC-V Foundation (started in 2015)
  - Headquarters moved from Delaware to Switzerland in 2019 to promote international political neutrality.
- Shepard RISC-V ISA standards, promote usage in industry through training and advocacy. Provide structure and stability. Compliance suites.
- Community driven organization
  - Contributions belong to RISC-V International. They maintain the copyright and the ISA is released under a Creative Commons Attribution International license.
  - Members may participate in the working groups, but archives are publicly visible to everyone.
- Funding comes from corporate members who pay a tiered membership fee.
  - $250,000 Annual Fee
    - Board seat and membership in the technical steering committee.
  - $100,000 Annual Fee
    - Membership in the technical steering committee.
  - Lower cost tiers with less privileges, that are based on company size.
    - $2000 - $35,000.
- Individuals may join for free as community members or can join if they are part of a corporate body that is a member.
- There are currently more than 2000 members, including 18 members paying at least $100,000 per year.
- Some board members are elected, others are based on membership fees.
  - Board members include representatives from: Intel, Seagate, Alibaba, Andes, SiFive (Asanovic), Google (Patterson), and others.
- Calista Redmond is currently CEO of RISC-V international. Formerly head of IBM z ecosystem development.

RISC-V Applications

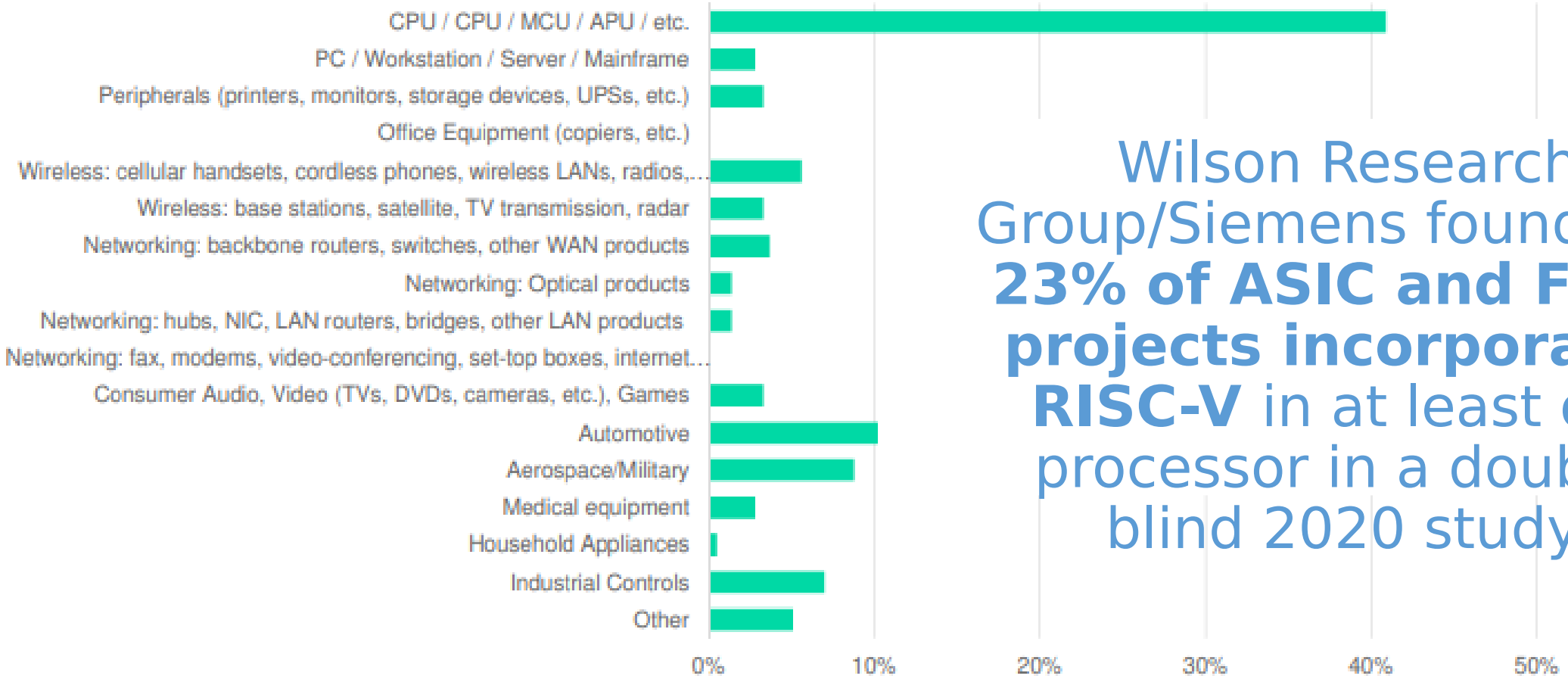
- Initial applications of RISC-V have been in the embedded and SoC area.
  - Machine learning accelerators
  - Embedded controllers
  - IoT
  - Automotive
  - Education (Hennesy and Patterson)
  - Research (including accelerators, security, and low power applications)
- Deeply embedded SoCs vs Server.
  - As of 2 to 3 years ago, lack of robust server implementations based on RISC-V meant that most Linux distributions were not ready for prime time.
    - Chicken and egg problem.
    - This is resolving. Alibaba is now starting to use RISC-V based cores in their Apsara cloud.
    - More capable evaluation boards are being produced now to support Linux porting and implementation.
- You won’t see RISC-V in personal computers anytime soon.
- The choice of an ISA or core depends on several factors.
  - Non-recurring engineering and royalty expense
  - If you’re not implementing the ISA from scratch, the choice of available cores is important.
  - Availability of hardware IP and standard interconnection logic.
  - Impediments from unfavorable licensing agreements
    - You may not want someone else’s logo on your chip.
    - You may want to reuse the work in another product.
  - Longevity and control
  - Software ecosystem

RISC-V Revenue Projections



Source: Omdia and Deloitte (Dec. 2021). Original report 2019.

Nearly a quarter of designs incorporate RISC-V



Wilson Research Group/Siemens found that **23% of ASIC and FPGA projects incorporated RISC-V** in at least one processor in a double-blind 2020 study.

RISC-V®

Slide Credit: RISC-V Foundation.

RISC-V Commercial Implementations

- Too many to list (below is a sample).
- Western Digital – SweRV
  - Written in system Verilog and open-source.
  - Flash Drive Controller (1 billion RISC-V cores shipped per year).
- Every Nvidia GPU now has a RISC-V controller embedded in it.
- SiFive
  - Started by original RISC-V team from UC Berkeley. Asonovic, et al.
  - Series of configurable core IP with varying performance characteristics.
- Esperanto Technologies
  - RISC-V based inference SoC with over 1000 RISC-V based cores.
- Alibaba
  - Building cores for their cloud and edge computing platform (Yitian 710).
  - As of late last year, they have done 11 RISC-V implementations.
  - They are porting Android OS.
- Andes Technology
  - High performance, low-power 32-bit RISC-V cores for embedded SoC applications.
- Greenwaves Technologies
  - Low power RISC-V cores that also are adapted for signal processing and use in IoT sensor applications.
- StarFive
  - RISC-V AI visual processing (Jinghong 7110)
- There are currently more commercial vendors of the RISC-V ISA than any other ISA.

University RISC-V Open Source

- Rocket (UC Berkeley)
  - Generated Verilog using a Scala based DSL called Chisel. In order.
- Boom (UC Berkeley)
  - Out of order
- Ariane (ETH Zurich)
  - Linux Capable
- RI5CY (ETH Zurich)
  - DSP enhancements
- PULP (ETH Zurich)
  - Parallel Ultra Low Power
- RISCY (MIT)
- Ibex (ETH Zurich, University of Bologna, now lowRISC – Cambridge University)
- Klessydra (University of Rome)

Software Ecosystem

- Compilers, Libraries, Tools (Dates shown are when upstreamed)
  - GCC and Binutils (May 2017)
  - Newlib (March 2018)
  - glibc (February 2018)
  - GDB (March 2018)
  - LLVM – Experimental support in version 8.0, full support in 9.0 (Nov. 2019)
- Linux Kernel (2017)
  - Debian and Fedora distros have almost all packages ported.
- FreeBSD (2017)
- QEMU (2017)
- Embedded OS/RTOS
  - FreeRTOS (AWS), LiteOS, SylixOS, RTEMS, ZephyrOS, …
- Prior to upstreaming you had to download and build parts of the toolchain.
- Commercial Software Ecosystem (Debuggers, Compilers, Tools)
  - IAR, Ashling, Lauterbach, Mirium, Segger, etc.

ISA Details

- Implementation independent (no microarchitectural assumptions)
- Modular
  - Base ISA
  - Enhanced by adding standardized extensions.
- Extensible
  - There are separate instruction encoding spaces reserved for standards use and custom application use.
- Base ISAs
  - RV32I, RV64I (47 instructions), both ratified (most implementations based on these)
  - RV128I – not ratified and under discussion. May find use in large data centers where persistent objects from disk are mapped into process address spaces.
  - RV32E - Embedded, Subset of RV32I, 16 registers not ratified.
- RV (RISC-V), 32-, 64-, or 128-bit address space size, I-Integer

Standard Extensions

- M – Integer multiplication and division
- A – Atomic memory operations including Load Reserve and Store Conditional
- F – Single precision floating point (32-bit FP)
- D – Double precision floating point (64-bit FP)
- Q – Quad precision floating point (128-bit FP)
- G – Used to signify general purpose combination of IMAFD.
- C – Compressed Instructions
- All of the above have been ratified and therefore will not change.
- The base ISAs and all of the extensions, except for C (compressed) use a 32-bit instruction encoding. The C extension uses a 16 bit format.
- Linux implementations assume RV64IMAFDC.

Standard Extensions Continued

- L – Decimal Floating Point (open - not fully defined)
- B – Bit manipulation (frozen - unlikely to change but not ratified)
- J – Support for dynamically translated languages (open)
- T – Transactional Memory (open)
- P – Packed SIMD instructions (open)
- V – Vector operations (ratified)
- K – Scalar Cryptography (ratified)
- H – Hypervisor support (ratified)
- N – User Level Interrupts (open)
- S – Supervisor Level Instructions (ratified)

Architecture

- The architecture is divided into two parts
  - Privileged
    - Things that are relevant to operating systems
    - Memory Protection
    - Virtual Address Translation
    - Exceptions
    - Interrupts
    - Privilege Levels
      - User
      - Supervisor
      - Machine
    - Hypervisor Support
      - Virtualized versions of User and Supervisor mode
    - Supervisor Binary Interface
  - Unprivileged
    - Things that are relevant to compilers, assemblers and linkers.
    - Almost all instructions are defined here
    - Things that can be executed in user mode (as opposed to supervisor or machine mode)
- The Privileged and Unprivileged Architectures are for the most part completely independent.
- Privileged specification is somewhat more pliable than the unprivileged specification

Base Integer ISA Detail - Registers

Register	ABI Convention
x0/zero	Not applicable
x1	ra – return address
x2	sp – stack pointer
x3	gp – global pointer
x4	tp – thread pointer
x5	t0 – temporary – caller saved
x6	t1 – temporary – caller saved
x7	t2- temporary – caller saved
x8	s0 / fp – callee saved, can be used as a frame pointer.
x9	s1 – callee saved
x10	a0 – function argument, return value
x11	a1 – function argument, return value
x12	a2 – function argument
x13	a3 – function argument
x14	a4 – function argument
x15	a5 – function argument

Register	ABI Convention
x16	A6 – function argument
x17	A7 – function argument
x18	s2 – callee saved
x19	s3 – callee saved
x20	s4 – callee saved
x21	s5 – callee saved
x22	s6 – callee saved
x23	s7 – callee saved
x24	s8 – callee saved
x25	s9 – callee saved
x26	s10 – callee saved
x27	s11 - callee saved
x28	t3 – temporary – caller saved
x29	t4 – temporary – caller saved
x30	t5 – temporary – caller saved
x31	t6 – temporary – caller saved

- In terms of hardware all registers are equivalent except for x0, which is hardwired to 0. The ABI determines their usage in the software stack.
- Registers are 32 bits wide in RV32I and 64 bits wide in RV64I. RV32

Base Integer ISA - Instructions

- Load / Store Architecture – loads, stores and atomic memory operations are the only instructions that refer to memory. All other operations act on registers only.

Instruction Category	Varieties	Assembly Examples
Loads	Byte, Half Word, Word, Double For Sizes less than the full register width there are two variants signed, and unsigned. In the signed the result is sign extended. In the unsigned case the result is zero extended.	ld t0, 32(s1) - Loads the double at offset 32 from s1 lbu t1, 34(s2) – Loads the byte at offset 34 from s2 and zero extends the result to fill the rest of the register.
Stores	Byte, Half Word, Word, Double	sd t0, 32(s1) – stores t0 to a location that is at offset 32 from s1. sb t0, 34(s2) – stores the least significant byte of t0 to an offset which is 34 bytes from s2.

Base Integer ISA- Instructions (Cont.)

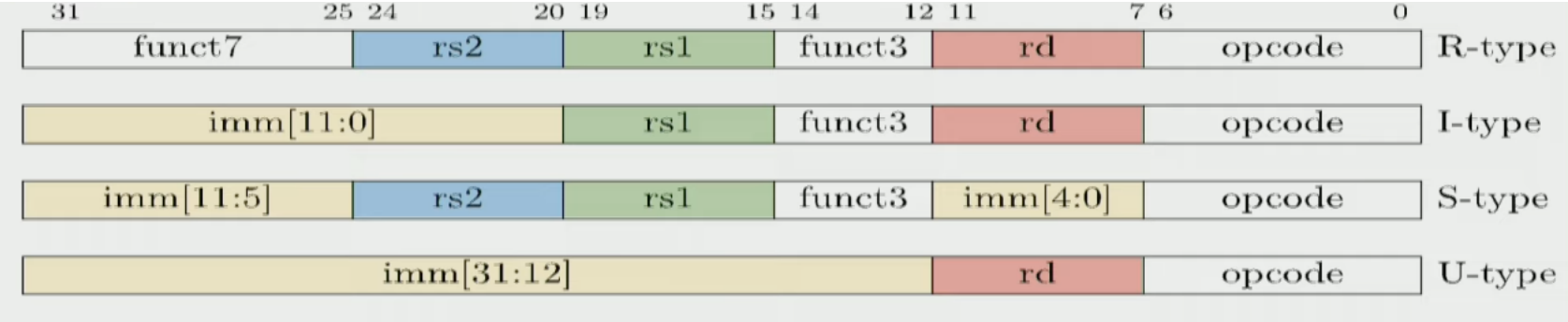
Instruction Category	Varieties	Assembly Examples
Shifts	Left Logical, Right Logical, Right Arithmetic. Logical shifts shift in 0. Arithmetic shifts shift in the MSB. The shift may be specified as an immediate parameter or in a register. Additional word long shifts in RV64I.	srai t0, t1, 3 – Shifts t1 right by 3 bits. Shifts in copies of the MSB of t1.
Arithmetic	Add, add immediate (12 bits – sign extended), subtract, load upper immediate, add upper immediate (20 bits shifted by 12) to program counter.	addi t0, t1, 12 – adds 12 to t1 puts result in t0.
Logical	XOR, OR, AND. With both registers and immediate.	or t0, t1, t2 – OR of t1 and t2, result in t0.
Compare	Compares a value to a register or an immediate and sets the destination to 1 if the comparison is true. Unsigned and signed versions.	slt t0, t1, t2 – Sets t0 to 1 if t1 is smaller treating t1 and t2 as twos complement numbers. sltu – same as above but t1 and t2 are treated as unsigned
Branches	Branch if =, !=, <, >=. Also unsigned compare for < and >=. Compares two registers.	bge t0, t1, 16 - If t0 >= 16 then pc to the current pc + 16.

Sequence below loads t1 from a location 0x532020 from the program counter. Commonly generated by compiler for accessing global variables:
auipc t0, 1330
ld t1, 32(t0)

Base Integer ISA – Instructions (Cont.)

Instruction Category	Varieties/Description	Assembly Examples
Subroutine Call	Jump and Link	jalr ra, 32(t0) Sets the program counter to t0 + 32 and puts pc+4 into ra.
ECALL	System call that results in a trap to a higher privilege level. Typically, machine mode unless otherwise delegated.	Ecall – conventional to identifiy the systcall using a7, and a0-a5 as parameters.
Fence	fence instruction orders preceding reads, writes, or I/O with succeeding reads, writes or I/O. fence.i orders stores to instruction memory with respect to subsequent fetches from instruction memory	fence w, r – make sure that all previous writes complete and are observable to all subsequent reads. fence.i – makes sure that subsequent instruction execution observes all previous writes to instruction memory.

Instruction Encodings



- Operands are always in the same place in the instruction.
- If the immediate is sign extended, the sign bit is always in bit 31 of the instruction.
- The low order 2 bits of the opcode will always be b’11’ for a 32-bit or longer instruction. Otherwise, the instruction is compressed.
- Instructions are stored in memory in little endian byte order.
- There is also an encoding space that has been defined that is reserved and will never interfere with the encodings used in the standards.
  - You can safely extend the instruction space without breaking things built on the existing software ecosystem.
  - There are also reserved ranges for things like custom core level interrupts, and exception encodings.

Notes on Common Extensions

- A floating-point extension (F,D, or Q)
  - Adds 32 registers and a status register.
  - The register set can have a width of 32, 64 or 128 bits for single, double and quad-precision respectively.
  - Adds instructions for floating-point register load/store, floating-point arithmetics, compare, and classify.
  - If the base integer address space width is less than the floating-point register width, then the floating-point load and store operations don’t execute atomically.
- Atomic Memory Operation Extension (A)
  - Atomically update a value in memory by performing an operation between the value in memory and a register, also place the initial value from memory in another register.
    - Operations include: add, and, max, min, or, xor, swap
  - Also includes load reserve and store conditional.
    - If the load successfully placed a reservation on the address of the store, the store will complete, otherwise an error is returned in the destination register specified in the store.
    - Allows the detection of intervening access between two instructions. Can be used to build more complex atomic primitives like compare and swap.
  - Acquire and release flags for precise synchronization.

Notes on Common Extensions (Cont.)

- Integer Multiplication and Division Extension (M)
  - Signed and unsigned division with separate instructions to compute the quotient and remainder.
  - Signed and unsigned multiplication with separate instructions to compute the high and low order parts of the product.
  - Some implementations will fuse the upper and lower multiply in the microarchitecture allowing a single hardware multiply.
- Compressed Instructions (C)
  - ISA supports variable length instructions. Not a separate mode. Can have a mix. Decoding the low order bits of the instruction allows determination of the length.
  - Two-byte format for the most frequently used instructions.
    - Load, store, add, subtract, logical, jump, jump and link, etc.
    - Adds 40 instructions
  - Smaller memory footprint also improves I-cache efficiency.
  - Instruction length is determined by least significant bits (starting at the lowest byte in memory). Longer encodings are also defined, 48, 64, etc.
  - 8 new instruction encodings for 16-bit instructions to allow specification of registers and immediate values.

Notes on Extensions (Cont.)

- Bit Manipulation Extension (B)
  - Adds 42 instructions
  - Any way you can think of to permute, extract, or count bits.
    - Generalized Reverse, Shuffle, etc.
  - Min/Max
  - CRC calculation
  - Bit Matrix operations
    - 64-bit value as an 8x8 matrix, Boolean AND as multiply, and either XOR or OR as the addition operator.
    - Also used to provide arbitrary permutations of bits in bytes.

Notes on Extensions (Cont.)

- Scalar Cryptography (K)
  - Divided into sub-extensions.
  - Most instructions operate on 64 or 32 bits at a time (depending on RV32I or RV64I), hence scalar.
  - Subset of bit manipulation instructions, that are useful for acceleration of cryptographic functions.
  - Acceleration of AES encryption and decryption.
  - Hash computations, SHA (256, 512), SM3, SHA2.
  - Block cipher encrypt/decrypt SM4.
  - Entropy source conditioning for seeding random bit generators.
  - There is a vector cryptography extension that is being developed.
  - Implementation note on all instructions that execution time must not depend on data (preventing side-channel attacks).

Other Extensions

- Vector (187 additional instructions - Ratified) and Packed SIMD
  - Useful for signal processing and machine learning (matrix multiplication).
- Dynamically Translated Languages
- User Level Interrupts
- Transactional Memory
- Hypervisor
- Decimal Floating Point

Privileged Architecture

- Platform / OS related Architecture
- Control and Status Registers
  - Space defined for up to 4096
  - Some ranges are reserved for custom implementations
  - Normally only accessible in privileged modes (typically machine mode, sometimes supervisor).
  - Interrupt enables, trap vectors, trap cause, interrupt pending bits, address protection, performance counters, debug and trace.
  - Six instructions are defined for accessing CSRs.

RISC-V Traps

- Exceptions are synchronous and are associated with the execution of a specific instruction (e.g., invalid load or store address, ecall, etc.).
- Interrupts are caused by asynchronous external events (e.g., timer ticks, external interrupt lines, software interrupts, etc.).
- Collectively interrupts and exceptions are referred to as traps, and by default all are handled in machine mode.
- The processing of traps for interrupts and exceptions is identical. This may be to a single location or, depending on the implementation, to a vectored location.
- Exceptions and interrupt handling may, in some cases, be delegated to a lower privilege mode (assuming the trap was caused while executing at the lower privilege level), otherwise redirection can be performed.
- Machine level registers related to traps
  - mcause – what caused the trap
  - mepc – the instruction that caused the exception, or where we will return on an interrupt
  - mtval – additional information about an exception, such as the location of an invalid instruction.
  - mstatus – global interrupt enable, and information about the previous state of the core prior to entering the interrupt handler.
  - mip, mie – Interrupt pending and interrupt enable registers.
  - mtvec – the location of the trap handler. In some implementations you can vector to an offset based on mcause.
  - mtval – additional information on an exception (like the address of an instruction that was invalid)
  - mideleg, and medelg – Interrupt and exception delegation registers.
  - mscratch – scratch register, typically used in the trap handler.

RISC-V Traps

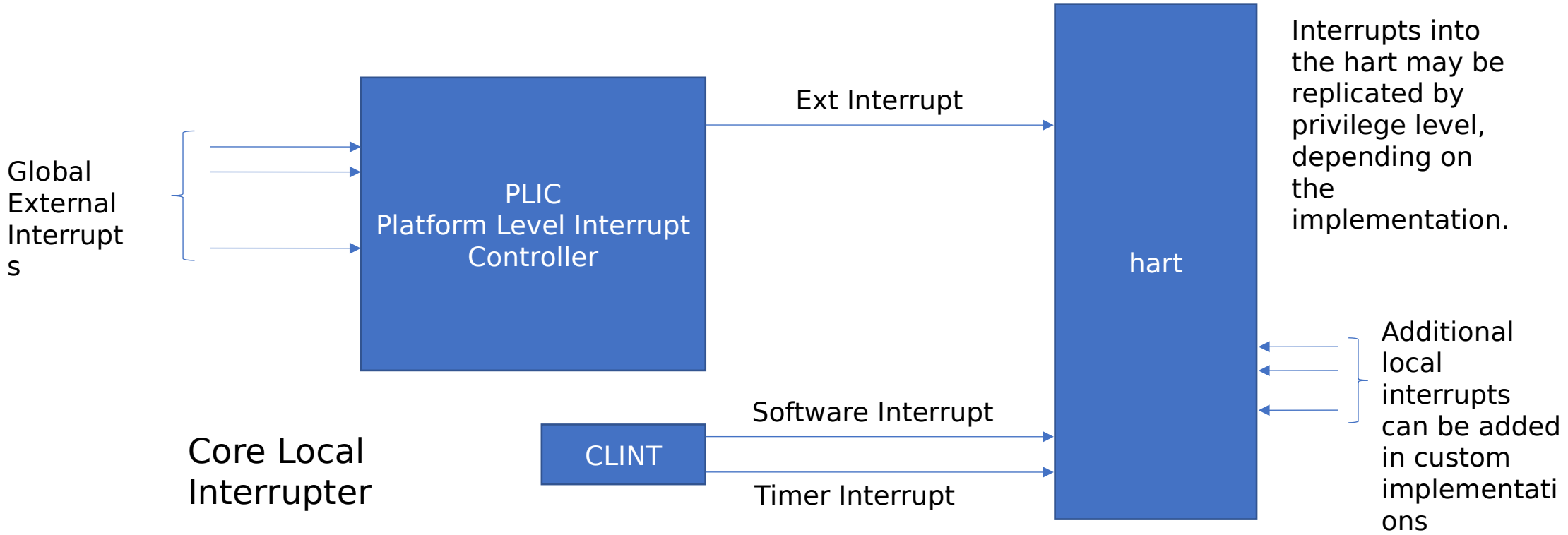
Exception Code (MSB = 1, not shown)	Description
1	Supervisor software interrupt
3	Machine software interrupt
5	Supervisor timer interrupt
7	Machine timer interrupt
9	Supervisor external interrupt
11	Machine external interrupt
>= 16	Reserved for custom use

Exception Code (MSB = 0)	Description
0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store address misaligned
7	Store access fault
8	Environment call from U-mode
9	Environment call from S-mode
11	Environment call from M-mode
12	Instruction Page Fault
13	Load Page Fault
15	Store Page Fault

Interrupts

- Architecturally, there are 3 interrupts that are supported by a RISC-V core
  - Timer, software, and external.
  - Can be enabled and disabled independently.
  - 3 pending and enable bits at the machine level (timer, software and external interrupts), also 3 corresponding pending and enable bits at the supervisor level.
  - A core (or hardware thread), can trigger a software interrupt in another core (or hardware thread) by writing to a memory mapped address.
  - There is a user level interrupt extension that has been proposed.

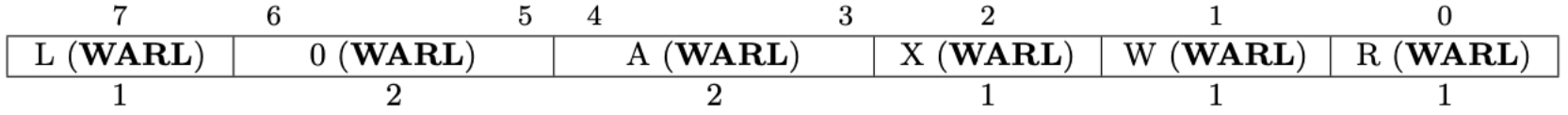
Interrupt Structure



- The PLIC and CLINT assert interrupts to the hardware thread (hart).
- Memory mapped registers in the CLINT
  - msip – For interprocessor communication – one core can raise an interrupt to another.
  - mtime, mtimecmp – interrupt presented when mtime >= mtimecmp.
- Memory mapped areas in the PLIC
  - Priority settings, Interrupt enables, Interrupt Pending Bits
  - claim/complete register, claim atomically returns highest priority interrupt and masks the interrupt, complete unmasks the interrupt.

Physical Memory Protection

- Supervisor and User mode have no memory access permissions by default.
- You must enable access by setting PMP registers that provide an aperture through which to access memory, otherwise you'll take at access fault.
- Two ways to configure a PMP region
  - Naturally aligned power of 2 or base and bounds (top of range)
- Up to 16 regions can be configured (for read, write, execute).
- PMP registers normally don't affect M-mode access, unless they are locked, in which case they can't be changed until the core is reset. MPRV bit can modify this in a machine mode interrupt handler.
- PMP access is tested after virtual to physical address translation, but a page fault will be raised prior to a PMP check.

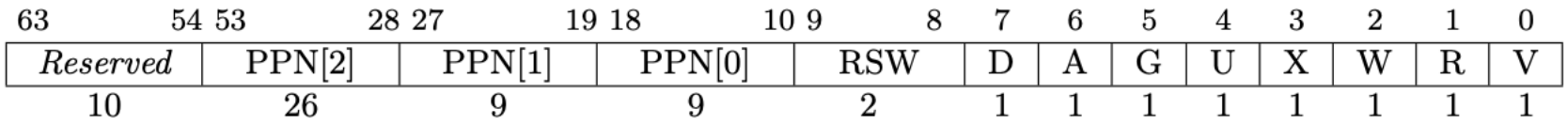


7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Virtual Address Translation

- If the supervisor extension is implemented, it adds virtual memory support.
- Multiple address space sizes are supported, since not all applications require large address spaces.
  - The larger the address space the greater the number of page table levels, the longer the latency to translate an address, and the larger the footprint required in the TLB.
- Supported address space sizes
  - 32-bit – Sv32, 2 Levels: 2 MB and 4 KB pages.
  - 39-bit – Sv39, 3 Levels: 1 GB, 2MB and 4KB pages.
  - 48-bit - Sv48, 4 Levels: 512 GB, 1GB, 2MB and 4KB pages
  - 57- and 64-bit address spaces have also been proposed.
- You can stop at any level to create leaf super-page references in the page table.
- RV64I allows for the specification of a 56-bit physical address in the page table entry, but the specification notes that a given implementation may not support the full 56 bits of addressability that’s provided in the page table entry.

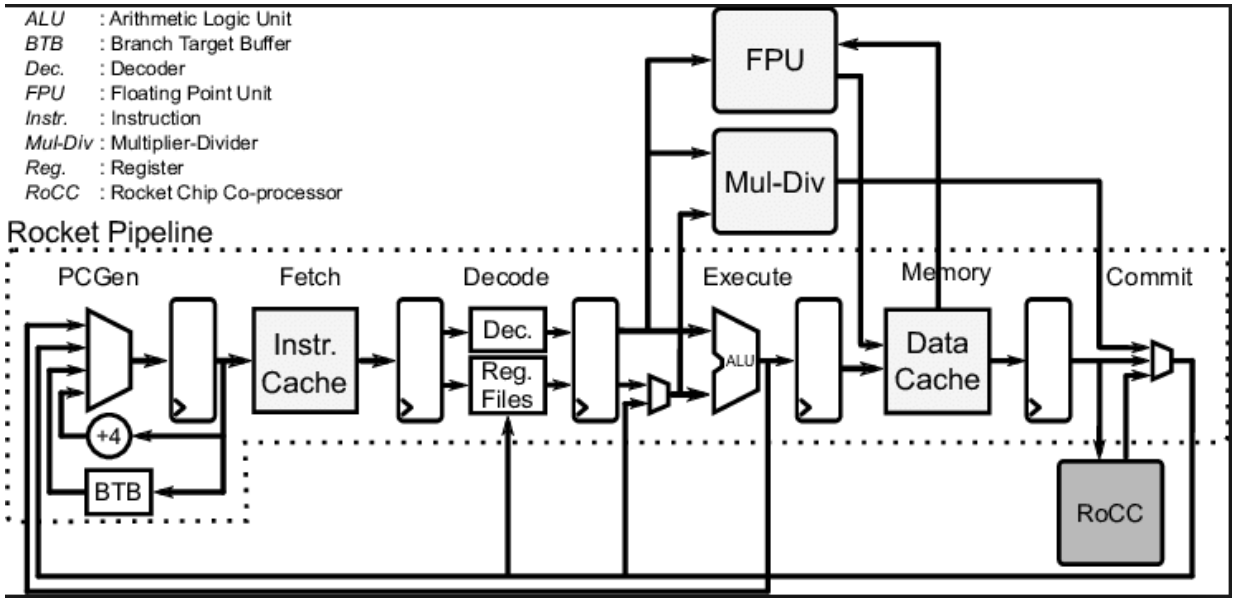
Sv39 Page Table Entry



- Sv39 Page Table Entry (from Privileged Spec) shown above.
  - PPNs – physical page numbers (collectively the bits of the physical address above the least significant 12).
    - May refer to a physical page, if it is a leaf entry, or the next level page table.
  - RSW reserved for use by supervisor software.
  - D – Dirty
  - A – Accessed
  - G – Global (mapping that exists in all address spaces).
  - U– Give user mode access to the page.
  - X,W,R – execute, write, read, permissions (0,0,0 means non-leaf entry).
  - V – Valid (if V is 0 all other bits are ignored)
- satp register contains the physical address of root of the page table (4KB aligned) an address space identifier, and mode bits, where mode is Sv32, Sv39, Sv48, or no translation.
- Supervisor mode can’t access user mode pages (U = 1). SUM bit in sStatus CSR that allows access.
- Supervisor mode can’t execute user mode pages.
- In other architectures you may find more controls settings in the page table entries, some of these are in RISC-V’s physical memory attributes (e.g., cacheability, whether or not atomicity is allowed, whether speculative fetching is permitted, etc.).

An Open-Source Implementation

- RISC-V Rocket
  - Implemented in Chisel
  - Parameterized with “knobs”.
    - L1 I-Cache size, # ways, L1 D-Cache size, etc.
    - Floating point or no floating point, etc.
  - Single Issue, in-order, cpi >= 1 (Follow on Boom is superscalar).



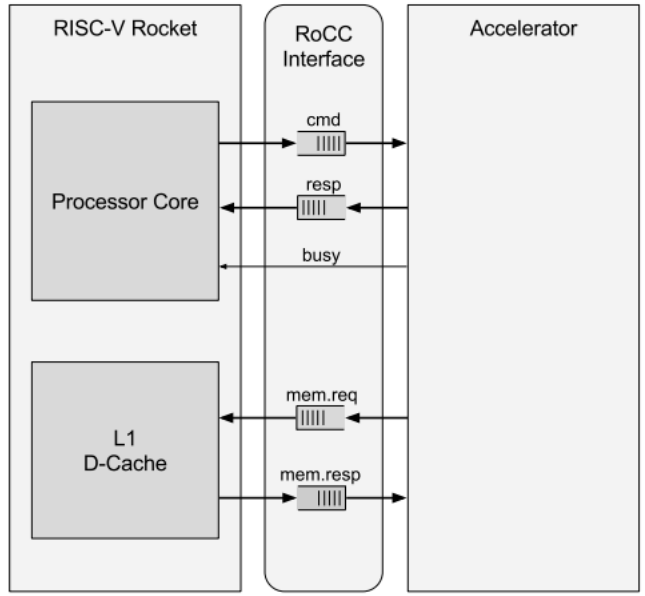
- PC gen determines the next program counter, including speculation on branch target address.
- Instruction Fetch takes metadata (predicted/taken flags) and target address from PC Gen. Also drives request to MMU for address translation (if implemented).
- Decode stage, decodes instructions and handles compressed instructions.
- Functional units are in the execute stage like the ALU, FPU and Multiplier-Divider.
- Memory stage handles movement of data to-and-from D-Cache.
- Commit stage takes the incoming instruction and updates the architectural state, including writing CSR and

Chisel

- Chisel
  - Hardware description language
  - Used for the Rocket and Boom open-source cores.
  - Developed at UC Berkeley.
  - Goal is to improve the productivity of designers relative to Verilog or VHDL.
  - Useful for generating families of designs through parameterization.
  - Everything that can be defined with Verilog or VHDL can be defined with Chisel, but there are additional constructs.
  - Hardware types were added to the Scala programming language
  - Methods are used to build interconnected hardware from the hardware types.
  - Also includes a library of standard components like Arbiters, queues, interfaces, etc.
  - Not c-to-gates, Chisel produces output to an intermediate language. A back-end program then produces final output, which is typically Verilog, but could be VHDL, etc.
  - For more info: https://www.chisel-lang.org/

Rocket Custom Coprocessor (RoCC) Interface

- RoCC is not part of the ISA and hasn't been standardized, but it demonstrates one way of extending the instruction set.
- You can refer to the custom functions from a high-level language using an assembly language macro
  - The compiler will assign registers automatically
  - Code looks like rc = custom_inst(FUNC_1, parm1, parm2);
  - No tool chain modifications needed.



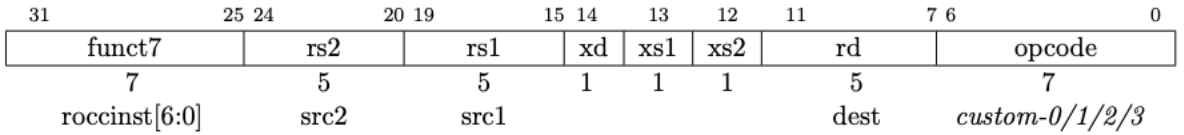


*Diagram Courtesy of James Martin

TileLink Interconnect

- If you are building SoCs you need a way to interconnect things.
- SiFive defined an open-source interconnect called TileLink
  - Probably the predominant interconnect used in RISC-V projects.
  - There are bridges to other popular interconnects like AXI or AHB.
- Worker/Controller point-to-point protocol
- Message based
- Supports both simple non-coherent and coherent interfaces.
- RISC-V Rocket uses TileLink
- Three Protocol Levels
  - TL-UL – TileLink Uncached Lightweight
    - Simple Read write operations
  - TL-UH – TileLink Uncached Heavyweight
    - Adds to TL-UH: multibeat messages, atomic operations, and prefetch
  - TL-C – TileLink Cached
    - Supports cache block transfers in snoop and directory-based cache coherency protocols.
- TL-UL and TL-UH are commonly used for memory mapped hardware.

Hardware Ecosystem Example

- OpenTitan
  - Announced in 2019
  - Part of the lowRISC project which originated at Cambridge University, now a not for profit, with partner organizations.
  - Primary motivation is to develop RISC-V open-source silicon components to support Root of Trust implementations.
  - As of now 29 hardware IP blocks have been developed that can be incorporated into SoC designs including USB 2.0, UARTs, SRAM Controller, SPI device interface, PWM, GPIO, Flash Controller, Crypto IP, Analog to Digital Converter Control, etc.
  - Ibex CPU core
  - Apache 2 license (permissive, no copyleft requirement)
  - C source code in the form of device interface functions are associated with each hardware device and represent APIs that define how to interact with the IP block.
  - Most components have TileLink interfaces.
  - opentitan.org

What if I want to try it out?

- Building the program
  - There are risc-v specific GCC command line options.
    - Example: -march=rv64imac, also specify ABI –mabi=lp64
- Simulation options for your programs
  - Spike
    - Supports all base ISAs and most extensions including bit manipulation, crypto, vector and packed SIMD.
    - Instructions on how to build spike and run a program are on the github repo page.
    - Can be used with GDB.
    - github.com/riscv-software-src/riscv-isa-sim
    - Allows you to model caches
  - QEMU
    - Supports most extensions.
    - qemu.org/download
    - Use bin/qemu-system-riscv64 with –machine virt –cpu rv64 –m <memory_size> -smp <num harts> -nographic -bios none –kernel <executable> -s –S
      - for basic bare metal
      - -s QEMU listens for GDB, -S QEMU waits for start from GDB
    - The default GDB port exposed from QEMU is 1234
- Evaluation Boards
  - If you prefer real hardware.
  - See riscv.org/exchange/ for a list boards

Customizing the Hardware

- Custom instructions can be added to Spike and QEMU for software simulation.
- If you really want to customize the hardware you can start with an open-source core and modify it.
  - Each core is implemented with a specific tool chain.
    - You will want to simulate your hardware changes.
    - There are free and open-source tools available.
    - Most of the tool chains have code to convert the generated output to a C or C++ program that can be compiled and run to perform the simulation.
    - There are tools to produce waveform output, like GTKWave.
      - http://gtkwave.sourceforge.net/
  - Rocket and Boom use the Chisel tool chain
    - You can use the Chisel compiler to generate C++ simulation models.
  - OpenTitan and the Ibex core are programmed in SystemVerilog.
    - Verilator can be used to create the simulation model.
    - See opentitan.org/doc/getting_started/ and veripool.org/verilator/
  - I have not tried any of these tools and so cannot offer a recommendation.
  - To realize your design in physical hardware you'll need an FPGA board.

Useful Links

- RISC-V international website - riscv.org
- Online learning – riscv.org/risc-v-learn-online
- ISA Manuals - riscv.org/technical/specifications/
- Chisel - chisel-lang.org
- Spike Simulator - github.com/riscv-software-src/riscv-isa-sim
- QEMU - qemu.org/download
- OpenTitan - opentitan.org
- Boards - riscv.org/exchange/
- Me: lmhopkin@us.ibm.com