Introduction to the RISC-V ISA

Rishiyur S. Nikhil

December, 2022

These slides are available at: `https://github.com/rsnikhil/RISC-V_Intro`

Disclaimer: any errors/opinions herein should be attributed only to the author and not to RISC-V International or to Bluespec, Inc.

Contents

"RISC-V", per se, is *only* an ISA (Instruction Set Architecture)

- Only a specification, a document, describing:
  - "Architecturally-visible" state: registers (PC, integer, PC, floating point, CSRs)
  - Repertoire of instructions, and binary representation of each one
  - Semantics (meaning) of each instruction: how it accesses/changes architecturally visible state

  i.e., it's the abstract view of the machine targeted by general-purpose compilers.
- Not an actual CPU, processor, chip, core, ...
- *Not within the purview of the ISA*: micro-architectural choices in particular implementations, such as pipeline structure, instruction pre-fetch, branch-prediction and other speculation, bypassing, superscalarity, out-of-order execution, multithreading, caches, MHz, CPI, energy consumption, *etc.*

  There can be a variety of implementations varying widely on these dimensions, from IoT microcontrollers to server-class cores to supercomputing cores.

This Introduction is only *about the ISA, not any specific RISC-V implementation*

Primary References

Unprivileged and Privileged ISAs:

- *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA, Document Version 20191213*, Andrew Waterman and Krste Asanović (editors), December 13, 2019, 238 pp.
- *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture, Document Version 20211203*, Andrew Waterman, Krste Asanović1 and John Hauser (editors) December 4, 2021, 155 pp.

PDFs for all RISC-V specs can be found at:

`https://riscv.org/technical/specifications/`

Clean-slate design

- RISC-V was created circa. 2010, at University of California, Berkeley, USA, by a team with 4-5 decades of experience in ISA design and implementations.
  - Four previous RISC designs (hence the "V" in "RISC-V"), including the parent design of the **SPARC** ISA (commercialized by Sun Microsystems, Oracle, Fujitsu, ...).

Result:

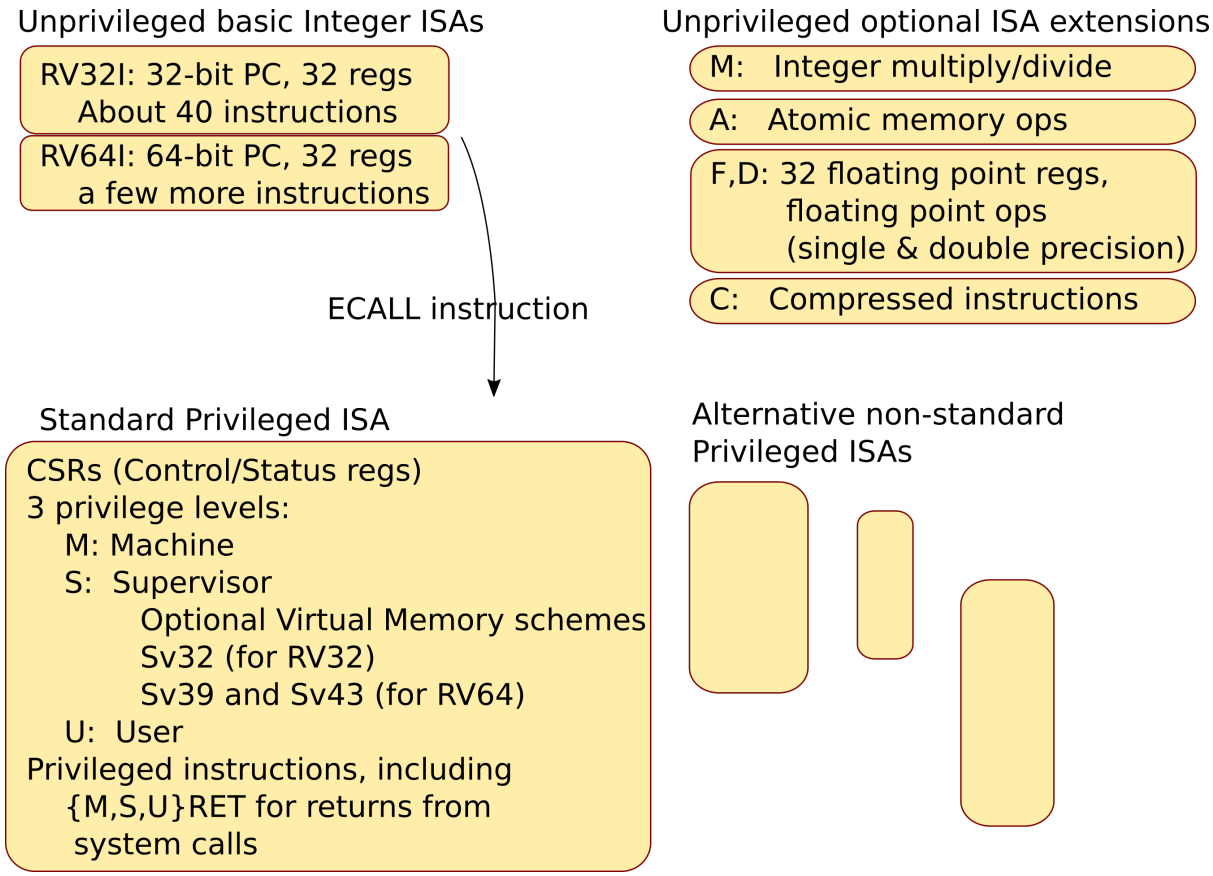
- *Very* clean, orthogonal, simple design.
- No legacy bloat (*e.g.,* branch-delay slots, ...)
- Compare: ARM's 64-bit architecture (ARM v8-A) spec is over 6000 pages long.

RISC-V is an *Open* ISA, unlike other ISAs

- There are many well-known, mature ISAs—x86, ARMv8, Power, Sparc, ...—but they are all *proprietary*, requiring licenses and fees if you want to implement them.
- RISC-V does not require any license fee to implement.[1]
  - This is central to all the current commercial interest in RISC-V.
  - This is central to *architecture researchers* who wish to innovate on state-of-the-art implementations, and indeed was the original motivation behind the creation of RISC-V by a team at Univ. of California, Berkeley.
- Because it is *open*, and already relatively mature, RISC-V seems well on its way to becoming one of the *three dominant ISAs* (along with x86 and ARM). Over time, it may also displace the numerous lesser-known ISAs used for microcontrollers.

[1] The name "RISC-V" is owned by RISC-V International, and you need their permission to use the name in commercial naming of products.

The RISC-V ISA has a *Modular* Organization



PDFs for all RISC-V specs can be found at: `https://riscv.org/technical/specifications/`

- Many other standard extensions exist: vector, crypto, bit-manipulation, ...
- Implementors can choose according to target, from small IoT/Embedded (*e.g.,* RV32IC "bare-metal") to server-class/HPC (e.g., RV64IMAFDC with vector, crypto, bit-manipulation and standard Privileged ISA)
- HW facilities allow SW to discover the configuration on which it is running.

Instruction formats

PDFs for RISC-V specs can be found at: `https://riscv.org/technical/specifications/`

- All instructions are 32-bits wide, for both RV32 and RV64.
- There are very few instruction formats (simplifies hardware-decoder in CPU pipelines).

130 *Volume I: RISC-V Unprivileged ISA V20191213*

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- In the optional "C" extension ("compressed"), instructions are 16-bits wide, for small-footprint systems (IoT/Embedded/Edge).
- Each 16-bit C instruction expands to a standard 32-bit RV32/64 instruction; so only needs hardware at the front-end of the CPU pipeline (16-bit fetch and expansion).

Base Unpriviliged Integer ISA (with 32 32-bit or 64-bit registers)

PDFs for RISC-V specs can be found at: `https://riscv.org/technical/specifications/`

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12|10:5] rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5] rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5] rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5] rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5] rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5] rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU
imm[11:5] rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5] rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5] rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000 shamt	rs1	001	rd	0010011	SLLI
0000000 shamt	rs1	101	rd	0010011	SRLI
0100000 shamt	rs1	101	rd	0010011	SRAI
0000000 rs2	rs1	000	rd	0110011	ADD
0100000 rs2	rs1	000	rd	0110011	SUB
0000000 rs2	rs1	001	rd	0110011	SLL
0000000 rs2	rs1	010	rd	0110011	SLT
0000000 rs2	rs1	011	rd	0110011	SLTU
0000000 rs2	rs1	100	rd	0110011	XOR
0000000 rs2	rs1	101	rd	0110011	SRL
0100000 rs2	rs1	101	rd	0110011	SRA
0000000 rs2	rs1	110	rd	0110011	OR
0000000 rs2	rs1	111	rd	0110011	AND
fm pred succ	rs1	000	rd	0001111	FENCE
000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK

- RV32I ISA (Unprivileged, 32 x 32-bit registers, integer only) has a mere 40 instructions (left)
- RV64I ISA (32 x 64-bit registers) adds 15 more instructions (below)

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]	rs1	110	rd	0000011	LWU
imm[11:0]	rs1	011	rd	0000011	LD
imm[11:5] rs2	rs1	011	imm[4:0]	0100011	SD
000000 shamt	rs1	001	rd	0010011	SLLI
000000 shamt	rs1	101	rd	0010011	SRLI
010000 shamt	rs1	101	rd	0010011	SRAI
imm[11:0]	rs1	000	rd	0011011	ADDIW
0000000 shamt	rs1	001	rd	0011011	SLLIW
0000000 shamt	rs1	101	rd	0011011	SRLIW
0100000 shamt	rs1	101	rd	0011011	SRAIW
0000000 rs2	rs1	000	rd	0111011	ADDW
0100000 rs2	rs1	000	rd	0111011	SUBW
0000000 rs2	rs1	001	rd	0111011	SLLW
0000000 rs2	rs1	101	rd	0111011	SRLW
0100000 rs2	rs1	101	rd	0111011	SRAW

- AUIPC (Add Upper Immediate to PC): enables position-independent code.
- Pure "Load-Store" ISA, *i.e.,* complete separation of memory access instructions from all other instructions (Lx, Sx).
- All I/O is via memory-mapped registers; no separate instructions.
- ECALL: "call-out" to Privileged ISA. Details are part of Privileged Spec.
- EBREAK: "call-out" to debugging environment (unspecified).

Possibly the cleanest, most orthogonal, industrial-strength ISA ever.

"M" extension: Integer Multiply and Divide instructions

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

- S: signed; U: Unsigned: enables signed and unsigned multiplications.
- H: upper half of double-width data: enables 64-bit multiplications on RV32, 128-bit multiplications on RV64.
- W: enables 32-bit multiplications on RV64.

"A" extension: Atomic memory operations

RV32A Standard Extension

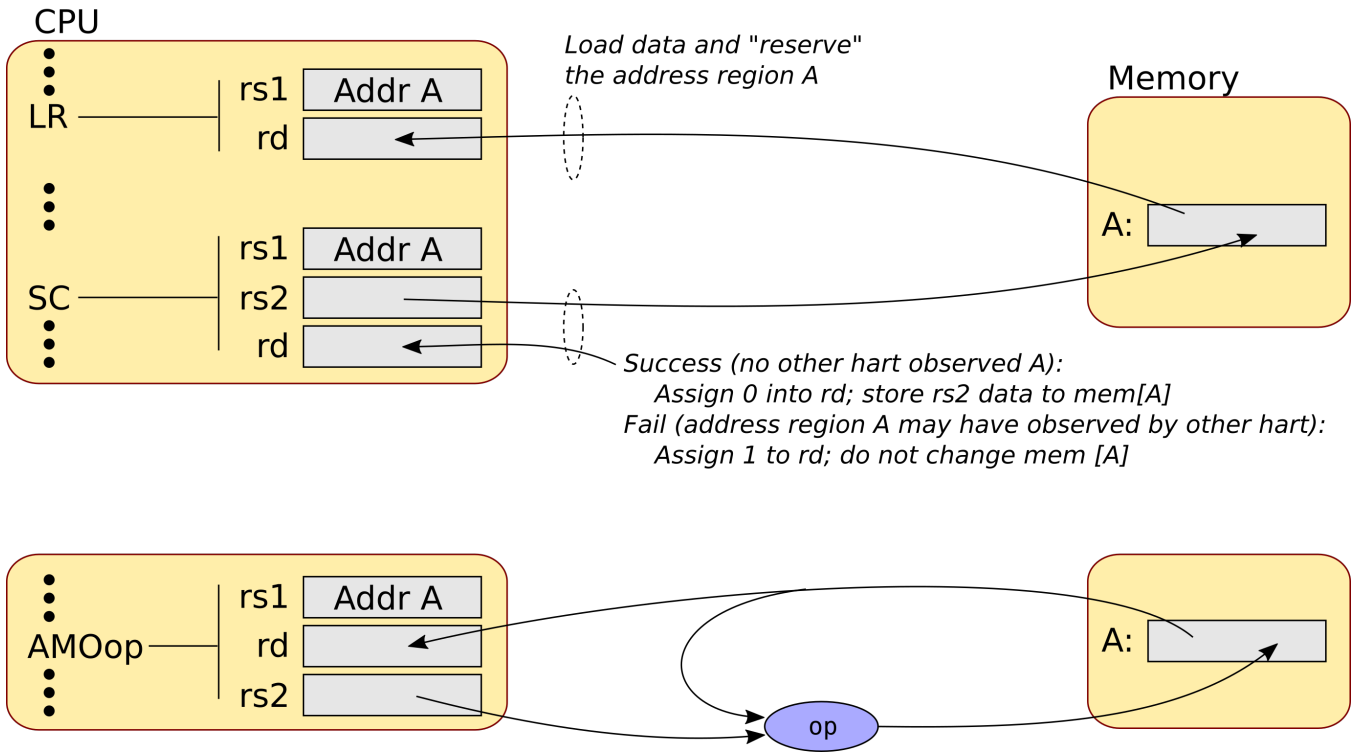
00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

RV64A Standard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	LR.D
00011	aq	rl	rs2	rs1	011	rd	0101111	SC.D
00001	aq	rl	rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	aq	rl	rs2	rs1	011	rd	0101111	AMOADD.D
00100	aq	rl	rs2	rs1	011	rd	0101111	AMOXOR.D
01100	aq	rl	rs2	rs1	011	rd	0101111	AMOAND.D
01000	aq	rl	rs2	rs1	011	rd	0101111	AMOOR.D
10000	aq	rl	rs2	rs1	011	rd	0101111	AMOMIN.D
10100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAX.D
11000	aq	rl	rs2	rs1	011	rd	0101111	AMOMINU.D
11100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAXU.D

- LR/SC: "Load Reserved/Store Conditional" on a memory location
- AMOxxx: atomic read-modify-writes on a memory location
- W ops operate on aligned 32-bit words
- D ops operate on aligned 64-bit words

AMO instructions (Atomic Memory Ops)



- LR/SC: Typically in a loop, until success
- LR/SC: Many nuances on "address region", "observed by another hart", number and type of instructions between the LR and SC, ...

* "hart" = "hardware thread", a single hardware thread, based on a single instruction-fetch unit

"F" extension: IEEE Single-precision Floating Point

State: 32 x 32-bit floating-point registers, plus these CSRs (Control/Status Regs)

Number	Privilege	Name	Description
Floating-Point Control and Status Registers			
`0x001`	Read/write	`fflags`	Floating-Point Accrued Exceptions.
`0x002`	Read/write	`frm`	Floating-Point Dynamic Rounding Mode.
`0x003`	Read/write	`fcsr`	Floating-Point Control and Status Register (`frm` + `fflags`).

Instructions:

RV32F Standard Extension

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S

(... more ... please consult spec)

RV64F Standard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	FCVT.L.S
1100000	00011	rs1	rm	rd	1010011	FCVT.LU.S
1101000	00010	rs1	rm	rd	1010011	FCVT.S.L
1101000	00011	rs1	rm	rd	1010011	FCVT.S.LU

"D" extension: IEEE Double-precision Floating Point

32 x 64-bit floating-point registers, plus these CSRs (Control/Status Regs)

Number	Privilege	Name	Description
Floating-Point Control and Status Registers			
0x001	Read/write	fflags	Floating-Point Accrued Exceptions.
0x002	Read/write	frm	Floating-Point Dynamic Rounding Mode.
0x003	Read/write	fcsr	Floating-Point Control and Status Register (frm + fflags).

Instructions:

RV32D Standard Extension

imm[11:0]			rs1	011	rd	0000111	FLD
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01	rs2	rs1	rm	rd	1000011	FMADD.D
rs3	01	rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	01	rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	01	rs2	rs1	rm	rd	1001111	FNMADD.D
0000001		rs2	rs1	rm	rd	1010011	FADD.D
0000101		rs2	rs1	rm	rd	1010011	FSUB.D
0001001		rs2	rs1	rm	rd	1010011	FMUL.D
0001101		rs2	rs1	rm	rd	1010011	FDIV.D
0101101		00000	rs1	rm	rd	1010011	FSQRT.D
0010001		rs2	rs1	000	rd	1010011	FSGNJ.D

(... more ... please consult spec)

RV64D Standard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	FCVT.L.D
1100001	00011	rs1	rm	rd	1010011	FCVT.LU.D
1110001	00000	rs1	000	rd	1010011	FMV.X.D
1101001	00010	rs1	rm	rd	1010011	FCVT.D.L
1101001	00011	rs1	rm	rd	1010011	FCVT.D.LU
1111001	00000	rs1	000	rd	1010011	FMV.D.X

"C" extension: Compressed Instructions for smaller footprint

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	*Illegal instruction*
000	nzuimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzuimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	uimm[5:4|8]	rs1′	uimm[7:6]	rd′	00	C.LQ *(RV128)*
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.FLW *(RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	uimm[5:4|8]	rs1′	uimm[7:6]	rs2′	00	C.SQ *(RV128)*
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.FSW *(RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.SD *(RV64/128)*

Table 16.5: Instruction listing for RVC, Quadrant 0.

(... more ... please consult spec)

- "C" instructions are 16-bits wide, can be packed two-to-a-32b-word.
- Not standalone; are mixed with standard 32-bit instructions.
- 3-bit register fields refer to the 8 "most popular" registers.
- Each 16-bit C instruction expands to a standard 32-bit RV32/64 instruction; so only needs hardware at the front-end of the CPU pipeline (16-bit fetch and expansion).
- Include M, F, D instructions.

Notes on Memory and I/O

- Pure "Load-Store" ISA (with register base-address + index), *i.e.,* total separation of memory instructions vs. non-memory instructions.
- Flat memory space (address-width depends on RV32 or RV64, and Virtual Memory Scheme).
- All I/O through memory-mapped device registers; no separate I/O instructions.
- Optional PMP CSRs (Physical Memory Protection) can impose base-and-bounds regions on memory with access permissions. This provides lightweight sandboxing without the overheads of full Virtual Memory (MMUs, page tables, address translation, ...).
- FENCE, FENCE.I, and SFENCE.VMA instructions for implementations that need to synchronize multicores, I-Caches and D-Caches, and MMUs with Caches.
- For multicores, RISC-V has an ARM-like "Weak Memory Model", enabling more reordering of memory traffic for higher performance. Also has a a TSO option, which is a stricter memory model (like x86).

Standard Privileged ISA

* Because of clean ECALL from Unprivileged ISA, easy to substitute non-standard Privileged ISAs, *e.g.,* containing just basic trap handling for embedded applications.

Standard Privileged ISA

Note: The separation between Unprivileged and Privileged ISAs is very clean. A CPU implementation can easily implement a non-standard Privileged ISA with the standard Unprivileged ISA, if desired.

Privilege Levels: enables clean *virtualization* (hypervisors, Xen, VMWare, ...)

Level	Code	Name	Comment
3	11	M (Machine)	Highest privilege; firmware, boot loaders, ...
2	10	*Reserved*	
1	01	S (Supervisor)	Operating systems (like Linux)
0	00	U (User/Application)	Applications

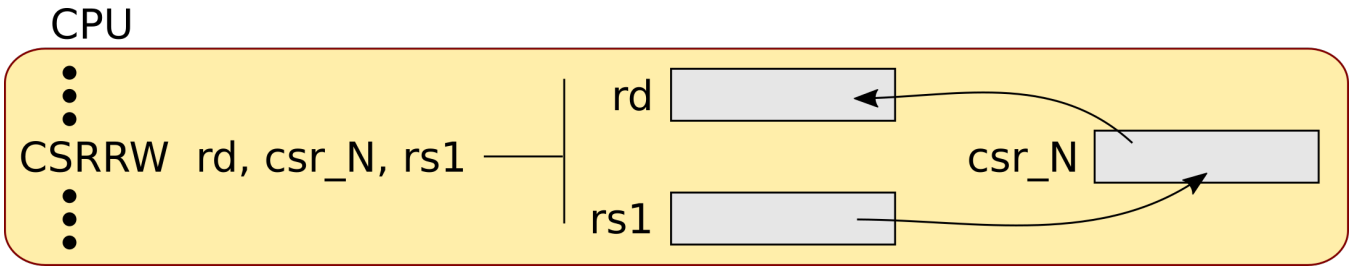
Implementations do not have to implement all privilege levels (implementation cost tradeoff):

Implemented Levels	Intended Usage
M	Simple embedded systems ("bare metal")
M,U	Secure embedded systems
M,S,U	Full OS + apps

Standard Privileged ISA: CSRs (Control and Status Registers)

- CSRs: additional set of registers in the CPU, identified by 12-bit CSR number.
- Up to 4096 possible CSRs, but most are optional; only a small handful are essential in an implementation.
- CSRs can be accessed programmatically with the CSR instructions shown below, each of which atomically swaps the contents of a CSR with standard integer registers, with some variations and nuances.
- CSR instructions can be used at all privilege levels, but the upper 4 bits of a CSR's number specifies which privilege levels can access that particular CSR, and with which ops (read, read/write).
- Some CSRs are "shadows" of other CSRs: only some bits visible, or read-only vs. read-write (*e.g.,* `cycle` is a shadow of `mcycle`).

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM



CSRs accessible from User level

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32 only.
0xC81	URO	timeh	Upper 32 bits of time, RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32 only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32 only.

Table 2.2: Currently allocated RISC-V unprivileged CSR addresses.

- CSRs for floating point
- CSRs for timing: real-time, cycle, instruction count
- CSRs for other performance counters (implementation-defined)

CSRs accessible from Supervisor level

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
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.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

- CSRs for handling exceptions (traps and interrupts) at Supervisor level (see later slides)
- `satp`: for Virtual memory (see later slides)

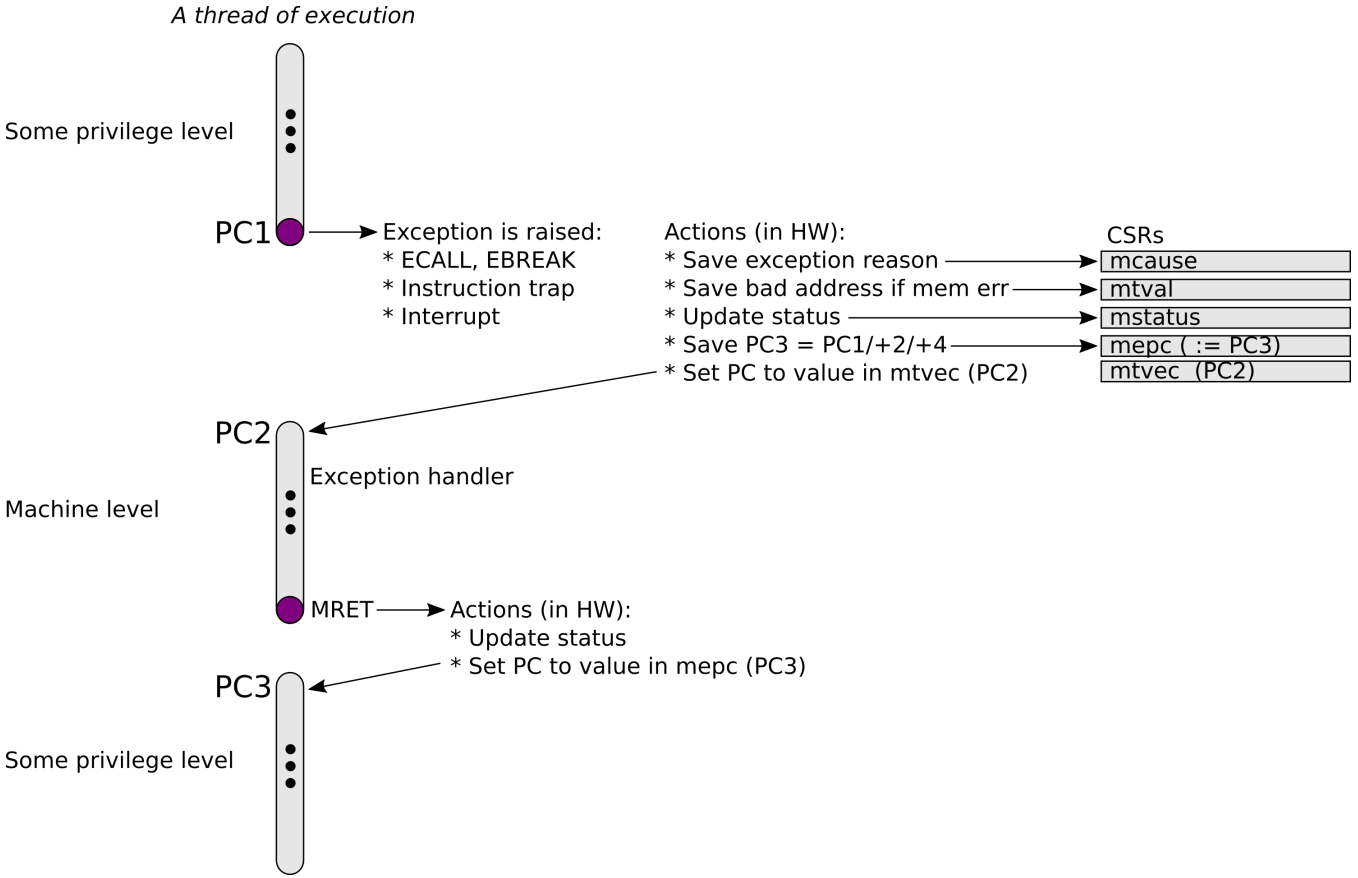
CSRs accessible from Machine level

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.
0xF15	MRO	mconfigptr	Pointer to configuration data structure.
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.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
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.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine bad guest physical address.

(... more ... please consult spec)

- CSRs for configuration discovery from software
- CSRs for handling exceptions (traps and interrupts) at Machine level (see later slides)

Exceptions (Traps and Interrupts)



- `mtvec` has been pre-loaded before this scenario (by boot-loader, OS, ...)
- Saved PC1/+2/+4 depends on whether the instruction needs to be retried, and if it is a compressed instruction.
- Exception handler can change `mtvec` to some other PC3, *e.g.,* to resume a different thread after the exception.
- Any other "arguments" and "results" are passed in registers, according to an ABI/calling convention.
- Can be recursive, *i.e.,* exception handler may itself encounter trap/interrupt.

CSR MStatus

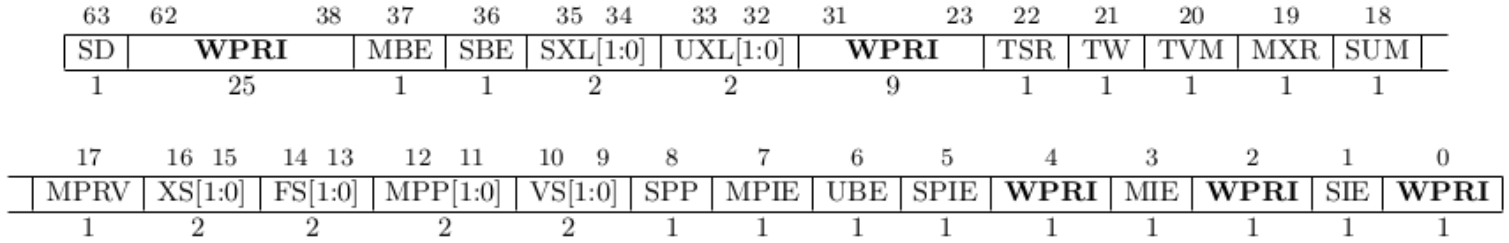


Figure 3.7: Machine-mode status register (`mstatus`) for RV64.

- `mstatus` is quite central and critical, and manipulation of its bits involves some complexity and subtlety; we recommend reading the spec very closely and carefully.
- Various bits represent a 2-level "stack" (indexed by privilege level) that are conceptually pushed on exception entry and popped on returns from exceptions (see "update status" on previous slide). These include interrupt enable bits (MIE/MPIE, SIE/SPIE), previous privilege level (MPP, SPP).
- The RV32 version of `mstatus` omits some of the bits.
- There is also an `sstatus` CSR at Supervisor level, which is similar to, but only a subset of `mstatus`.
- Exceptions can be delivered at Machine-level and Supervisor-level privileges.
- Exceptions delivered at Machine-level/Supervisor-level can be *delegated* to Supervisor-level/User-level (see `medeleg` and `mideleg` CSRs).

CSR MCause

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

- On any exception, the cause is recorded in CSR `mcause` during the transfer of control to the exception handler.
- Exception-handler code uses `mcause` to discover what kind of event caused this exception, using which it can branch to event-type-specific handlers.

CSRs MIP and MIE

15	12	11	10	9	8	7	6	5	4	3	2	1	0
0		MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
4		1	1	1	1	1	1	1	1	1	1	1	1

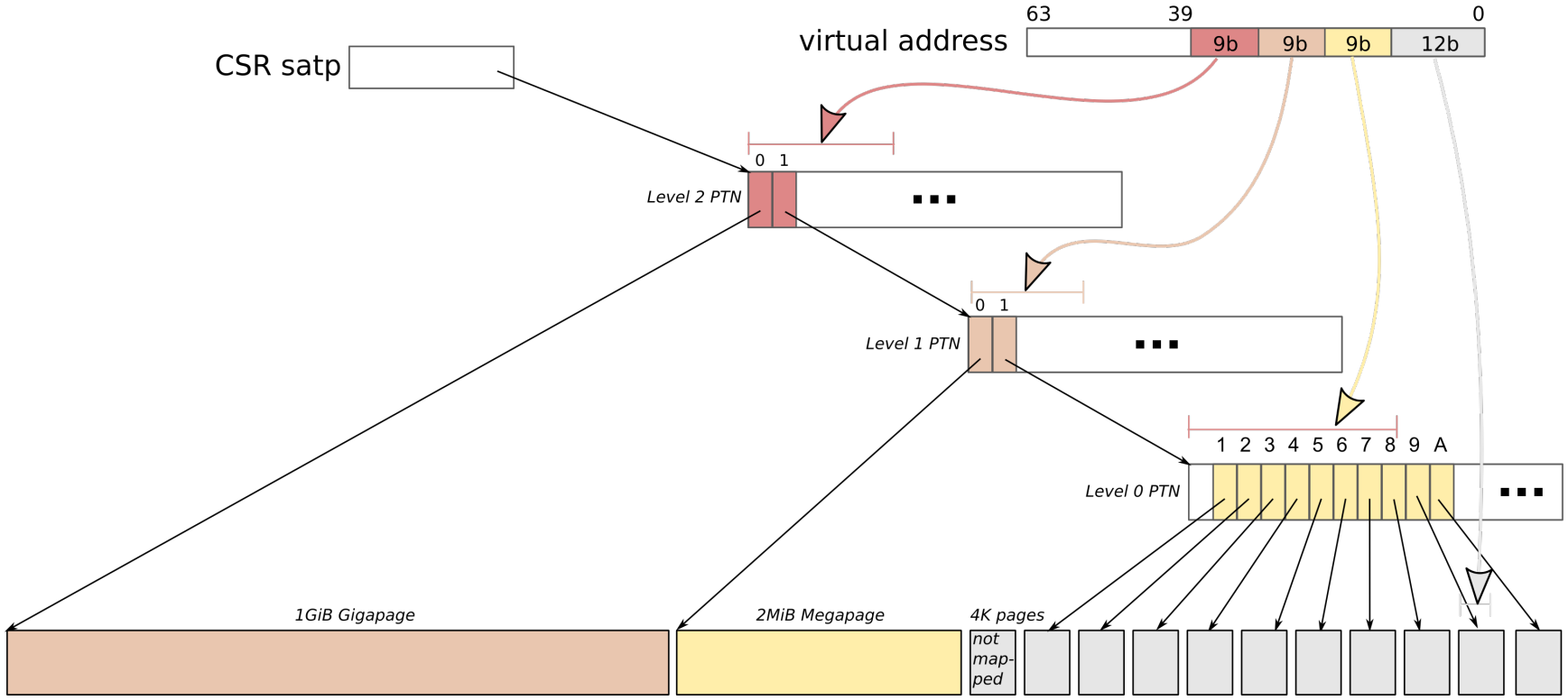
Figure 3.14: Standard portion (bits 15:0) of `mip`.

15	12	11	10	9	8	7	6	5	4	3	2	1	0
0		MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
4		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.15: Standard portion (bits 15:0) of `mie`.

- Interrupts arrive at a RISC-C hart only via the **`mip`** CSR.
- Both are full-width registers (32b in RV32, 64b in RV64) but only the bottom 16 bits have standard definitions.
- **`mip`**: Machine-level register for Interrupts-pending (there is also an **`sip`** for Supervisor level). The bits can be set by various sources of interrupts outside the CPU (see other slides on CLIC and PLIC for examples).
- Sources can be external devices ("E"), timers ("T") and inter-processor software interrupts ("S").
- **`mie`**: Machine-level register for Interrupts-enabled (there is also an **`sie`** for Supervisor level). The CPU can mask-out specific source of interrupts by writing 0 to the corresponding bit.

Virtual Memory Page Table for Sv39 using CSR `satp`

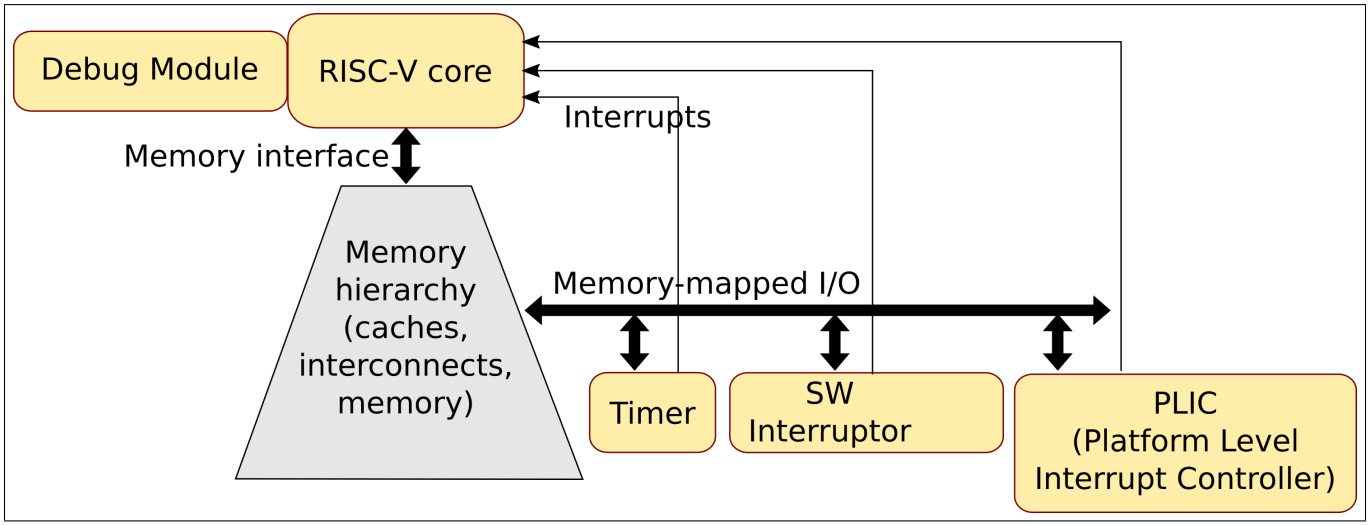


- When running in VM mode (indicated by `mstatus` bits, privilege level, *etc.*), the PC, and addresses in LD/ST instructions are VAs (Virtual Addresses).
- CSR `satp` contains a pointer to a 3-level tree. Each node in the tree is a *PTN* (Page Table Node), an aligned 4KiB block containing 512 *PTEs* (Page Table Entries, each 64b). Bits in the PTE indicate whether it is invalid, a leaf (pointing to a data page), or a pointer to the next-level node.
- Leaves at level 2 point at 1GiB naturally aligned "gigapages"; leaves at level 1 point at 2MiB naturally aligned "megapages"; leaves at level 0 point at 4KiB naturally aligned "pages";
- Sv39 addresses have 39 bits. 9-bit fields are used to index a PTE in a PTN, and the 12 LSBs to index a byte in a page.
- If a VA→PA translation fails (encounter invalid PTE, protection failure, ...) it raises a *page fault* exception or an *access fault* exception, which is handled in the usual way (see "Exceptions" slide earlier).

Standard Virtual Memory Schemes

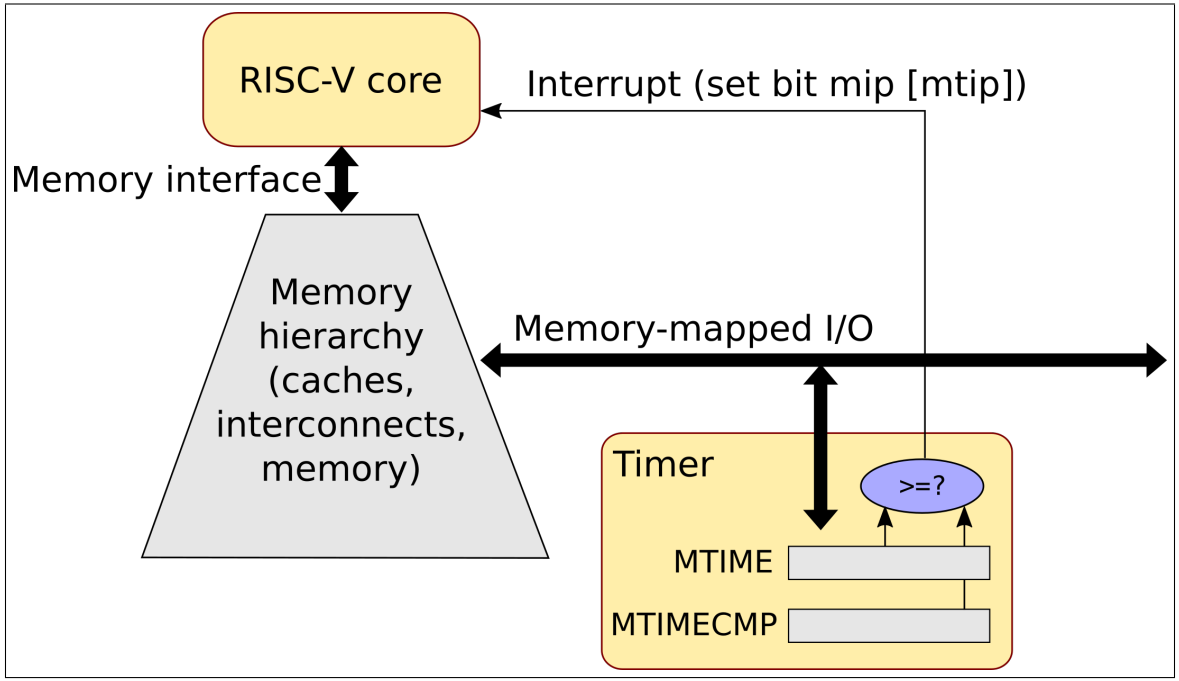
- The Privileged ISA for RV32 defines one standard Virtual Memory scheme: Sv32.
- The Privileged ISA for RV64 defines three standard Virtual Memory scheme: Sv39, Sv48 and Sv57.
- It is a choice for an implementation as to *how* VA→PA translation is implemented: in hardware or firmware or trap-handlers, whether it uses accelerators like TLBs (Translation Look-aside Buffers) *etc.* The SFENCE.VMA instruction is available to synchronize updates to address-translation hardware.

Common System Components accompanying RISC-V Cores



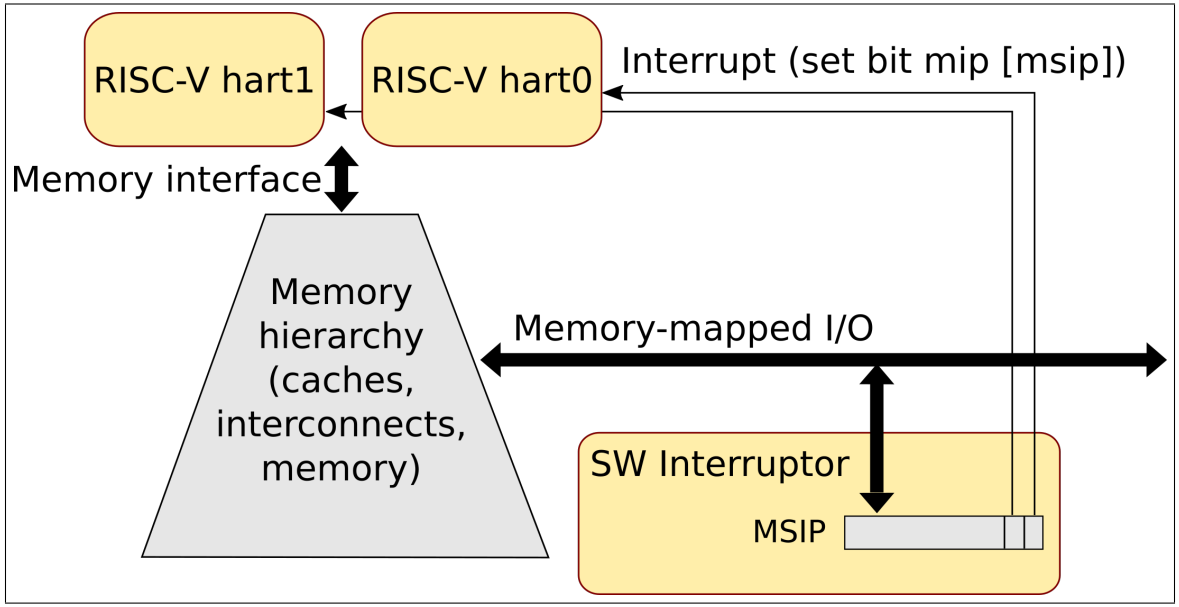
Each is described in more detail in the following slides.

Timer: Common System Component accompanying RISC-V Cores



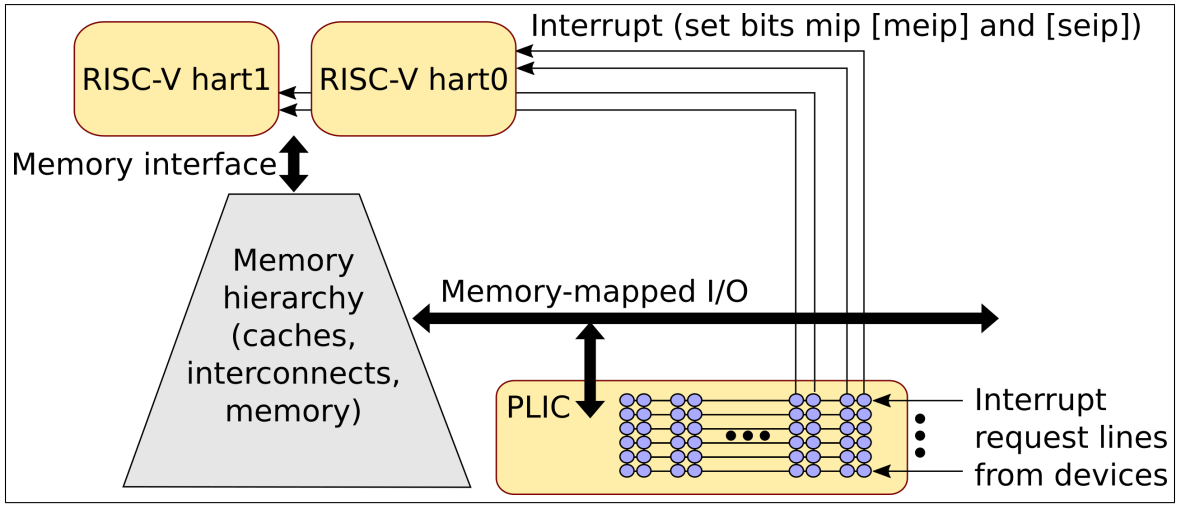
- For SW to measure real-time, and for real-time timer interupts
- Contains two memory-mapped registers, MTIME and MTIMECMP
- MTIME ticks upwards constantly.
- Timer Interrupts: typical usage:
  - CPU reads MTIME (let's call it t)
  - CPU writes $t + \delta$ in MTIMECMP
  - When MTIME ticks up by δ (so MTIME $\geq$ MTIMECMP), delivers an interrupt to CSR MIP at the MTIP bit position.

SW Interruptor: Common System Component accompanying RISC-V Cores



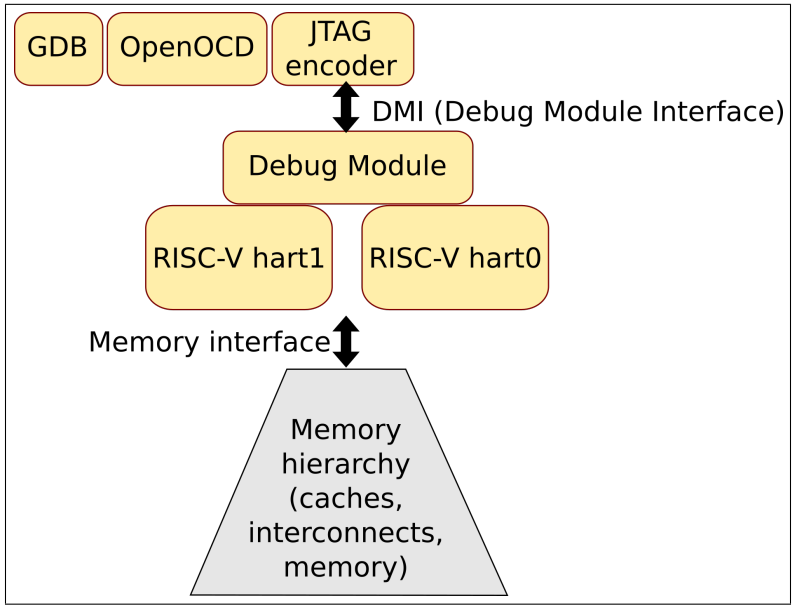
- For a hart to deliver a "sofware interrupt" to another (or same) hart.
- Illustration shows only two harts, but there could be more (multicore).
- Contains one memory-mapped register, MSIP
- SW Interruptor: typical usage:
  - Hart I writes MSIP $[J]$ to deliver a SW interrupt to hart J
  - MSIP $[J]$ is connected to the CSR MIP at the MSIP bit position (in core J).

PLIC: Common System Component accompanying RISC-V Cores



- Conceptually a matrix connecting an interrupt request line from a device to MIP[MEIP] or MIP[SEIP] of any of multiple harts.
- Illustration shows only two harts, but there could be more (multicore).
- Matrix nodes can be programmed by MMIO from cores to configure connectivity, priority, arbitration, interrupt masks, *etc.*
- A device interrupt can connect to multiple cores; PLIC has facilities for a core to "claim" an interrupt atomically.

Debug Module: Common System Component accompanying RISC-V Cores



- A RISC-V Debug Module is a HW module closely connected to multiple harts. Can control them individually and collectively.
- Functions: Reset hart, read/write registers and CSRs, read/write memory, stop/start/continue/single-step, set breakpoints, set watchpoints, ...
- Is controlled through a standard Debug Module Interface (DMI).
- Open-source code exists to JTAG-encode the DMI interface.
- Can connect from standard OpenOCD (Open On Chip Debugger) and GDB.

A Brief Note about the RISC-V Software Ecosystem

The following SW development tools have been available for many years and are mature (upstreamed into standard distributions):

- GNU toolchain (gcc, gdb, ...)
- Linux, FreeRTOS, seL4, ...

Hundreds/thousands of other porting efforts are underway (operating systems, programming language implementations, applications, tools), in hundreds/thousands of companies and universities worldwide; the ecosystem grows every day.

HW Verification of CPU and System Implementations

- Formal and Reference Models of RISC-V CPUs
- Standard ISA Tests
- Google open source "riscv-dv" instruction stream generator

Formal Specification of RISC-V ISA

RISC-V is the *only* major ISA with a public formal specification.*

Formal Specification (Unprivileged RV32/64 IMAFDC, Privileged):

- The formal spec itself: `https://github.com/riscv/sail-riscv`
- "A Tour of the RISC-V ISA Formal Specification": `https://github.com/rsnikhil/RISCV_ISA_Spec_Tour` by R.S.Nikhil, RISC-V Summit 2019. Video: `https://www.youtube.com/watch?v=k3NhEtk8TAs`

Written in **Sail**, a language (DSL) for ISA formal specifications from U.Cambridge, UK:

- Sail has also been used for a full formal spec for ARMv8 ISA, and for partial formal specs for Power, MIPs and x86.
- Sail website: `https://github.com/rems-project/sail`.
- Tutorial: "High Level Sail Overview", by Bill McSpadden, RISC-V Summit 2022.

The formal spec can be compiled into a simulator for RISC-V, which can be used as a golden reference model.

* We believe ARMv8 also has a formal specification, but only available internally inside ARM.

Golden Reference Models of the RISC-V ISA

Software simulators:

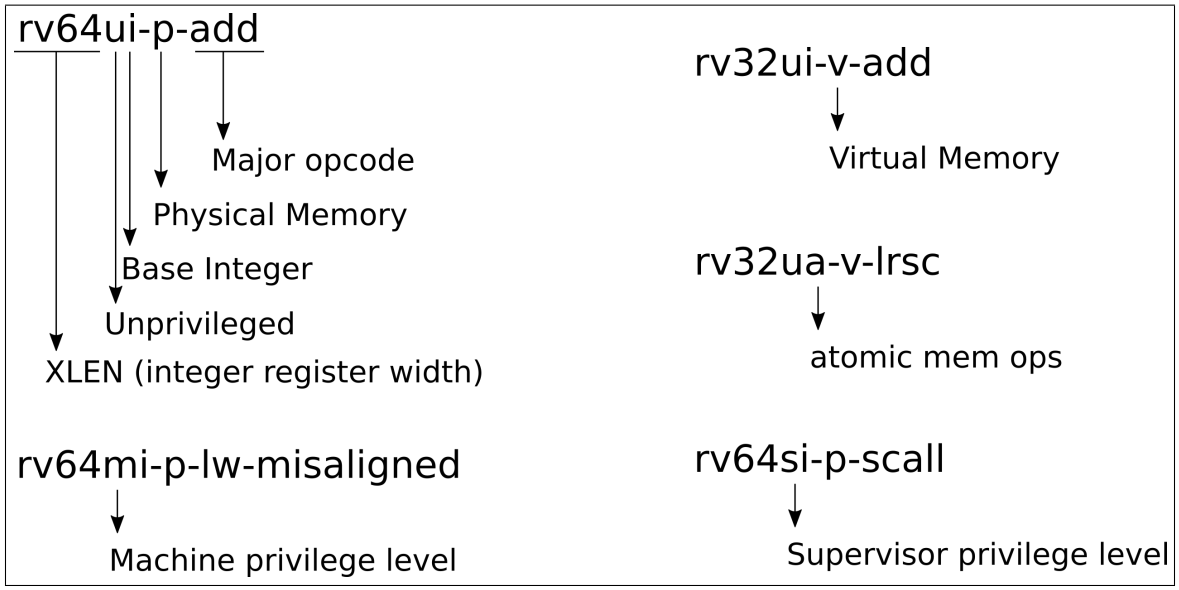
- These are pure simulators of the ISA, and do not simulate any micro-architectural details (pipelining, caches, MMUs, ...). They can produce an instruction trace that can be compared with the instruction trace from a hardware implementation.
- SPIKE (The "Golden Spike") is the principal simulator reference model written in C++ `https://github.com/riscv-software-src/riscv-isa-sim`
- Other:
  - Sail RISC-V formal spec, compiled to an executable.
  - In QEMU, TinyEMU, Imperas, ... (*many*)
  - Open-source formal spec written in Haskell: `https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec`

Hardware reference models:

- These are synthesizable HW implementations of RISC-V written for clarity (correctness), not performance, and can be run in HW (typically in FPGAs or HW simulation), adjacent to a real HW implementation, to compare ISA traces directly ("Tandem Verification").
- Examples:
  - RVBS, from U.Cambridge, UK: `https://github.com/CTSRD-CHERI/RVBS`
  - Magritte, from Bluespec, Inc.

Standard ISA Tests

- Reference: `https://github.com/riscv-software-src/riscv-tests`
- These are a set of several hundred programs, written in RISC-V Assembly language.
- Each program focuses on one particular opcode, and contains several test 1..N.
  - Each test ends with a postlude that either signals success, or failure with an indication of which of its N tests first failed.
  - The result signal is to store a value to a memory-mapped `tohost` location. Different implementations implement this in different ways.
- The test programs are named according to the ISA modular structure. Examples:



Other Verification Tests

Google open source "riscv-dv" instruction stream generator

- Reference: `https://github.com/google/riscv-dv`
  "RISCV-DV is a SV/UVM based open-source instruction generator for RISC-V processor verification."
- Typically run on reference golden model, and on DUT, and compare instruction traces.

Beyond ISA tests, HW verification is often performed by comparing behavior of reference golden model against HW implementation while booting an OS, typically Linux.

Would be useful: Test code whose execution time in simulation is between ISA tests (seconds) and Linux kernel (days), gradually adding virtual memory, interrupts, and process-switching:

- ISA tests' execution time: seconds (even in simulation).
- Linux kernel execution time: Many hours/days in simulation.

Perhaps a microkernel (seL4?) may serve this purpose.

End

Please direct questions/comments/errata to the author.

Possible topics to add to this introduction in future

- Verification under non-determinism (*e.g.*, with timer interrupts; multicore)
- Tandem Verification
- Verification of Weak Memory Model implementations
- Formal Verification