SiFive

An Introduction to the RISC-V Architecture

Drew Barbier - Sr. Product Marketing Manager

Date May, 2019

About This Presentation

This presentation is targeted at embedded developers who want to learn more about RISC-V

At the end of this presentation you should have a basic understanding of RISC-V fundamentals and know where to find more information

3-Part Webinar Series

May 7th

An Introduction to the RISC-V Architecture

May 15th

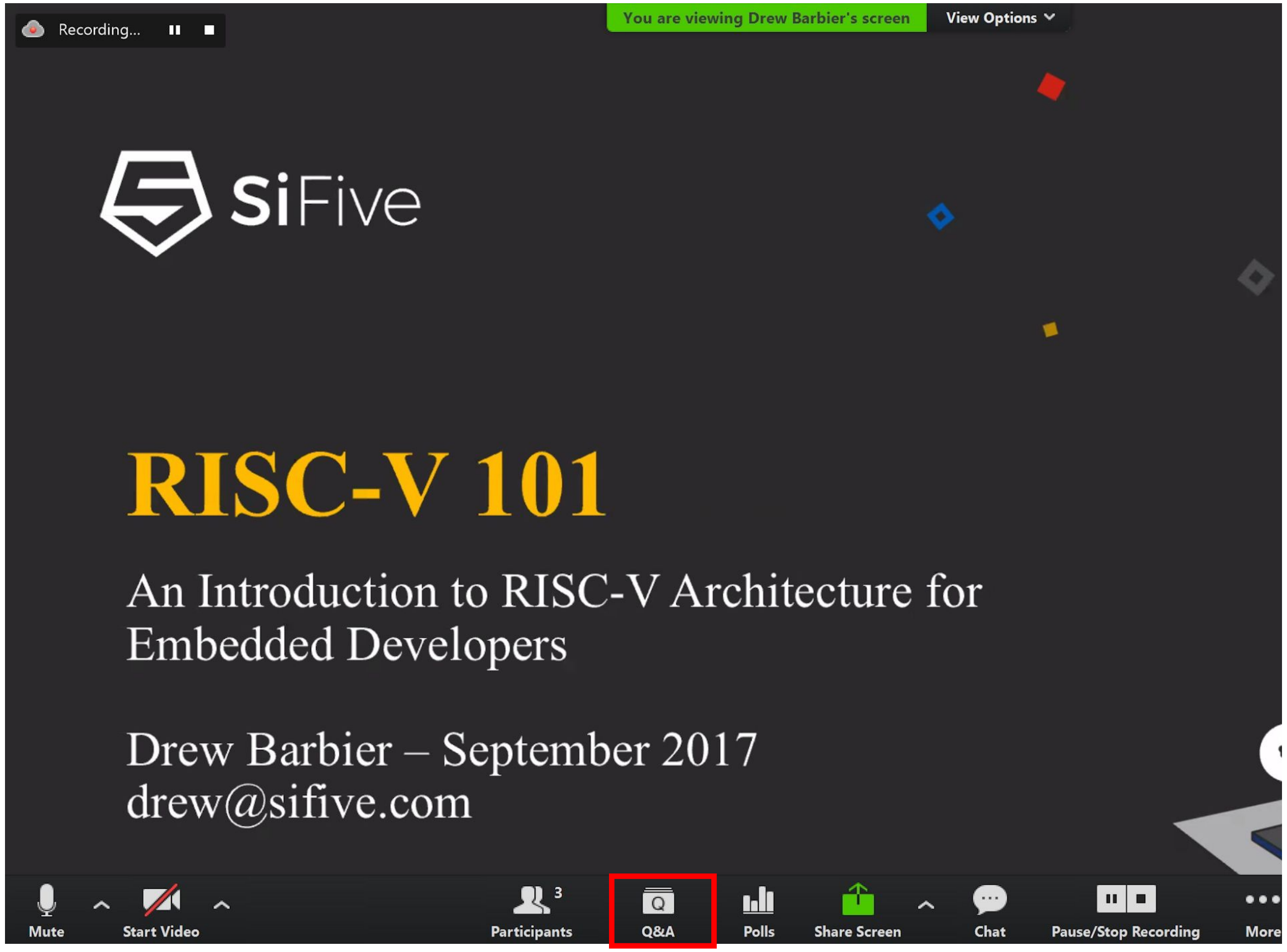
SiFive’s 2 Series Core IP

May 23rd

From a Custom 2 Series Core to Hello World in 30 Minutes

https://info.sifive.com/risc-v-second-webinar-series

How To Ask Questions



Recording...

You are viewing Drew Barbier's screen

SiFive

RISC-V 101

An Introduction to RISC-V Architecture for Embedded Developers

Drew Barbier – September 2017
drew@sifive.com

SiFive

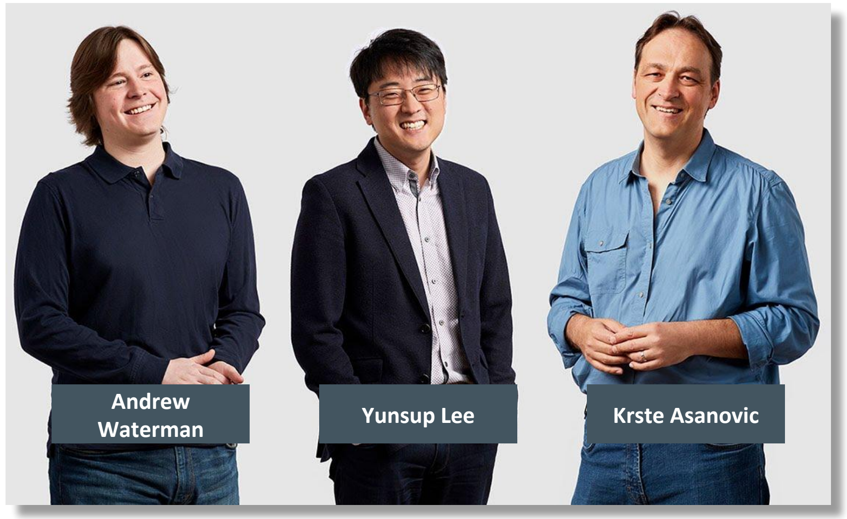
RISC-V Introduction

RISC-V Origin Story (pronounced: risk five)

- **Started as a "3-month project" in 2010 at UC Berkeley**
  - Required a simple ISA which could be extended
  - Commercial ISAs were too complex and presented IP legal issues

What is RISC-V?

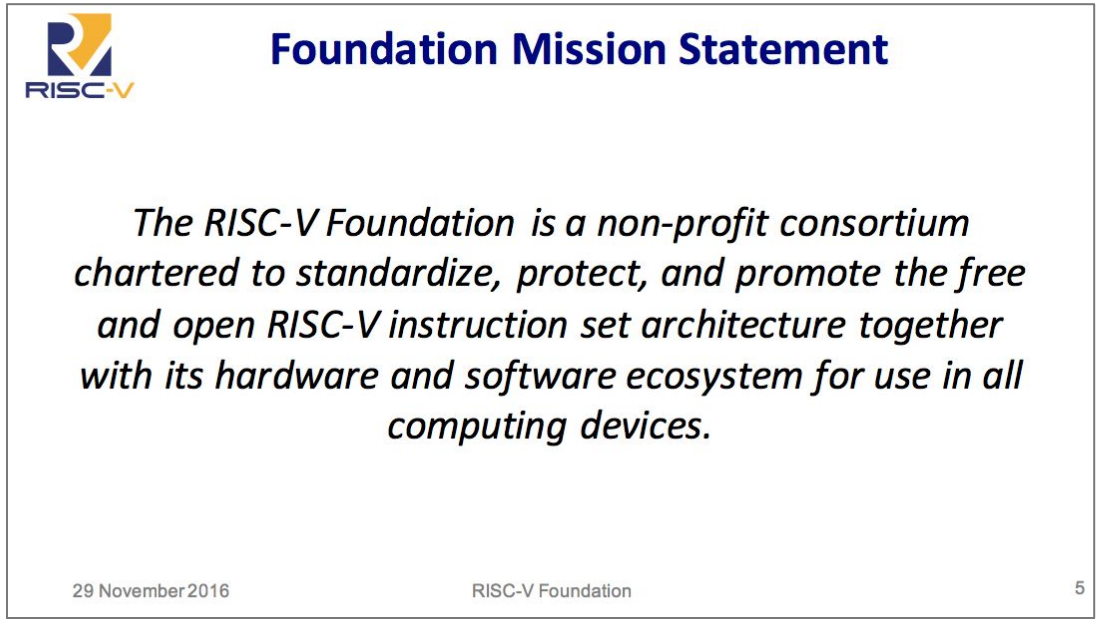
- **A high-quality, license-free, royalty-free RISC ISA**
- **Standard maintained by the non-profit RISC-V Foundation**
- **Suitable for all types of computing systems**
  - From Microcontrollers to Supercomputers
- **RISC-V is available freely under a permissive license**
- **RISC-V is not…**
  - A Company
  - A CPU implementation



Inventors of RISC-V

RISC-V Foundation – riscv.org

- **RISC-V Foundation is a non-profit organization formed in August 2015 to publicly govern the ISA**
- **Foundation Functions**
  - Directs future development of ISA
  - Compliance tests
  - Promotion of the ISA
- **>230 members representing a wide range of markets**



Foundation Mission Statement

The RISC-V Foundation is a non-profit consortium chartered to standardize, protect, and promote the free and open RISC-V instruction set architecture together with its hardware and software ecosystem for use in all computing devices.

29 November 2016 RISC-V Foundation 5

QUALCOMM Google NVIDIA IBM Rambus ALLWINNER MEDIATEK SAMSUNG Alibaba Group 阿里巴巴集团 NORTHROP GRUMMAN

NXP Mellanox TECHNOLOGIES PRINCETON UNIVERSITY SEAGATE BAE SYSTEMS Microsemi HUAWEI Micron ANDES TECHNOLOGY Blockstream codasip

MARVELL XtremeEDA MOSIS ORION 猎户星空 MINRES TECHNOLOGIES CEVA

C-SKY WD Western Digital SANECHIPS 中兴微电子 aselsan DATA 61 CSIRO CMC MICROSYSTEMS TESLA LAUTERBACH DEVELOPMENT TOOLS NUS National University of Singapore ultraSoC

THALES PANGO inside secure OpenSingularity BITMAIN HiGO 海高 DOLPHIN INTEGRATION GOWIN csem HITACHI Inspire the Next astc

tsmc Mentor A Siemens Business NUCLEI 芯来科技 GLORYSPACE 谨崍电子科技 WWW.GLORYSPACE.CN SiFive 君正 Ingenic Tortuga Logic SH CONSULTING SECURE-RF Securing the Internet of Things PEZY Computing

RISC-V foundation now > 230 members.

RISC-V

Free, open, extensible ISA for all computing devices

SoundAI DRAPER SEGGER SILEX INSIGHT TechanaLye TRINAMIC FORTH Institute of Computer Science QuickLogic SK hynix REM VeriSilicon SIEMENS

IAR SYSTEMS HORTONWORKS imperas imt. making ideas work INTRINSIC ID cadence BERKELEY LAB NETRONOME NSI-TEXE

CRYPTAPE 秘猿科技 DOVER MICROSYSTEMS dxcorr EMBECOSM EMDALO TECHNOLOGIES Embedded Intelligent Systems Hewlett Packard Enterprise 0x5 HEX-Five galois Berkeley Architecture Research

Technolution VectorBlox embedded supercomputing MIT CSAIL oculus SILICON LABS सी-डैक CDAC TANGRAM 致象尔微电子科技 BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación Canaan Centipede Simply Parallel CloudBEAR

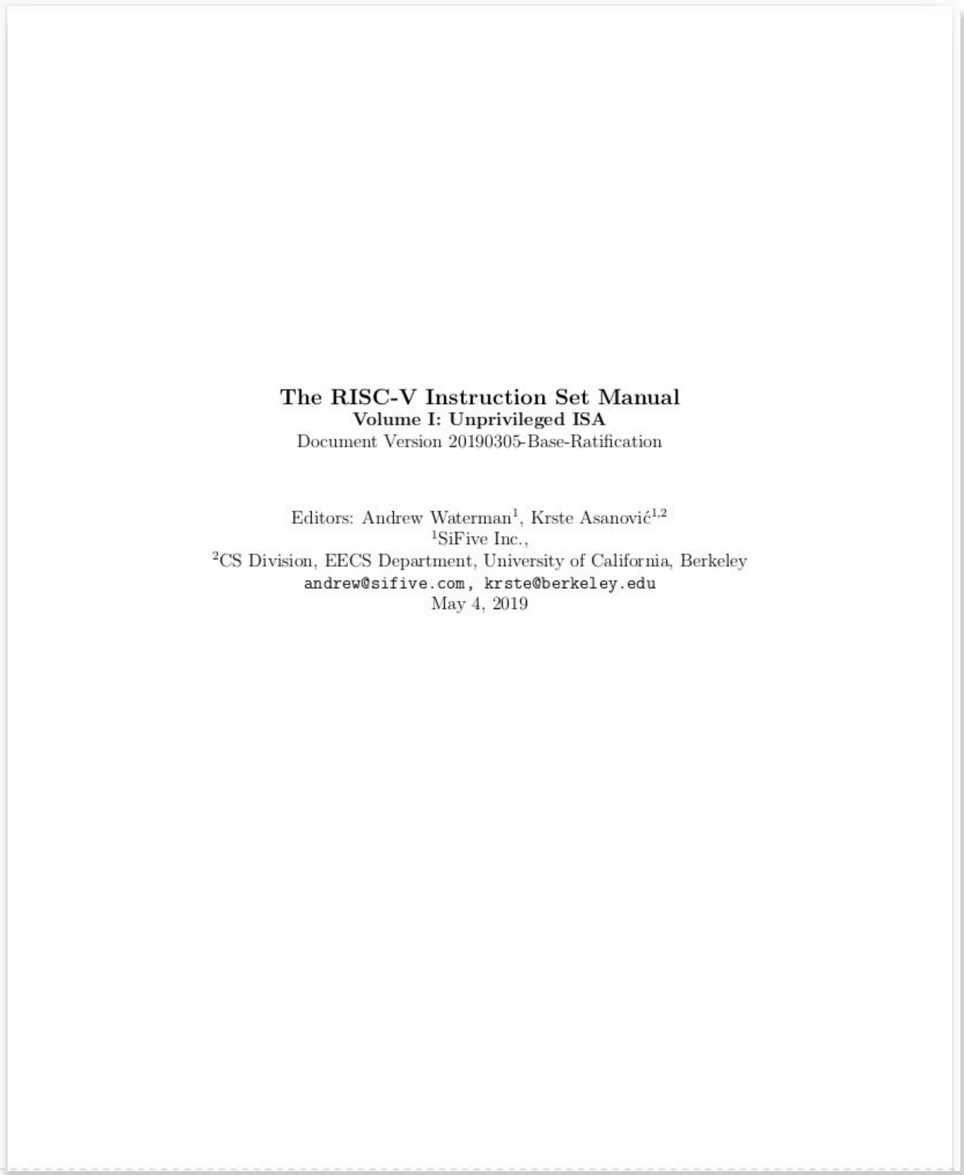
GRAY RESEARCH GREENWAVES TECHNOLOGIES INTRINSIX LATTICE SEMICONDUCTOR lowRISC Minima Roa Logic conductor investment 厦门半导体 Syntacore Custom cores and tools ICT 中科院计算所 INSTITUTE OF COMPUTING TECHNOLOGY, CAS

UBILITE IDT ASHLING expresslogic Symbiotic EDA Rumble Development

antmicro surecore PerfXLab 优矽科技 UC TECH IP bluespec cortus Esperanto TECHNOLOGIES ESPRESSIF ETH zürich

Status of the RISC-V Specifications

- **User Mode - version 2.2 Ratified**
  - Frozen in 2014 at version 2.0
  - Updates since 2.0:
    - CSR and FENCE.I instructions moved out of base extension “I”
    - Memory model clarifications
- **Privilege Mode - version 1.11 Ratified**
  - Version 1.11 ratified May 2019
- **Debug Spec - version 0.13 Ratified**
- **Specifications in Progress**
  - Hypervisor Extension - version 0.3 Draft
  - Vector Extension - version 0.7 Draft
  - And many more
- **Participate - https://riscv.org**
  - Join the mailing list
  - Become a member


The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20190305-Base-Ratification

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
May 4, 2019

SiFive

RISC-V Basics

RISC-V Instruction Set Architectures

- **RISC-V uses a standard naming convention to describe the ISAs supported in a given implementation**

- **ISA Name format: RV[###][abc…..xyz]**
  - RV – Indicates a RISC-V architecture
  - [###] - {32, 64, 128} indicate the width of the integer register file and the size of the user address space
  - [abc…xyz] – Used to indicate the set of extensions supported by an implementation.

Hello
my Architecture is
RV64GC

The Standard Extensions

- **Extensions define instructions**
  - "I" for Integer is the only required extension in a RISC-V implementation and defines 40 instructions
- **The RISC-V Specification defines a number of "Standard Extensions"**
  - Standard Extensions are defined by the RISC-V Foundation and are optional
- **RISC-V allows for custom, "Non-Standard", extensions in an implementation**
- **Putting it all together (examples)**
  - RV32I – The most basic RISC-V implementation
  - RV32IMAC – Integer + Multiply + Atomic + Compressed
  - RV64GC – 64bit IMAFDC
  - RV64GCXext – IMAFDC + a non-standard extension

Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions
Non-Standard User-Level Extensions	
Xext	Non-standard extension "ext"

Common RISC-V Standard Extensions
***Not a complete list**

SiFive

Register File

- **RV32I/64I have 32 Integer Registers**
  - Optional 32 FP registers with the F and D extensions
  - RV32E reduces the register file to 16 integer registers for area constrained embedded devices
- **Width of Registers is determined by ISA**
- **RISC-V Application Binary Interface (ABI) defines standard functions for registers**
  - Allows for software interoperability
- **Development tools usually use ABI names for simplicity**

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	-
x5-7	t0-2	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

RISC-V Modes

- **RISC-V Privileged Specification defines 3 levels of privilege, called Modes**
- **Machine mode is the highest privileged mode and the only required mode**
  - Flexibility allows for a range of targeted implementations from simple MCUs to high-performance Application Processors
- **Machine, Hypervisor, Supervisor modes each have Control and Status Registers (CSRs)**
  - More on these later

RISC-V Modes		
Level	Name	Abbr.
0	User/Application	U
1	Supervisor	S
2	Hypervisor	HS
3	Machine	M

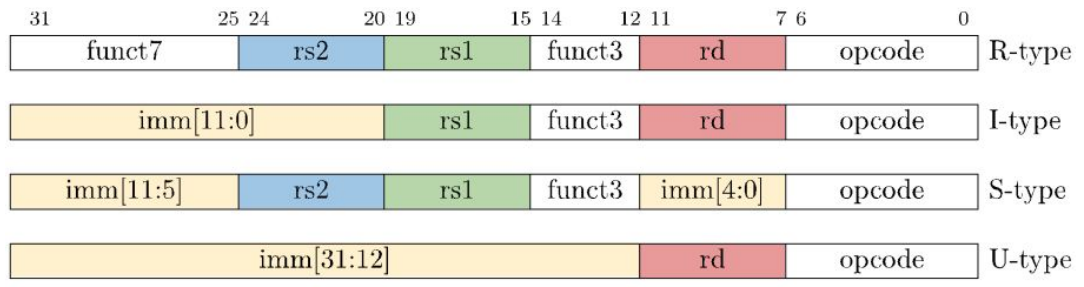
Supported Combinations of Modes	
Supported Levels	Modes
1	M
2	M, U
3	M, S, U
4	M, HS, S, U

SiFive

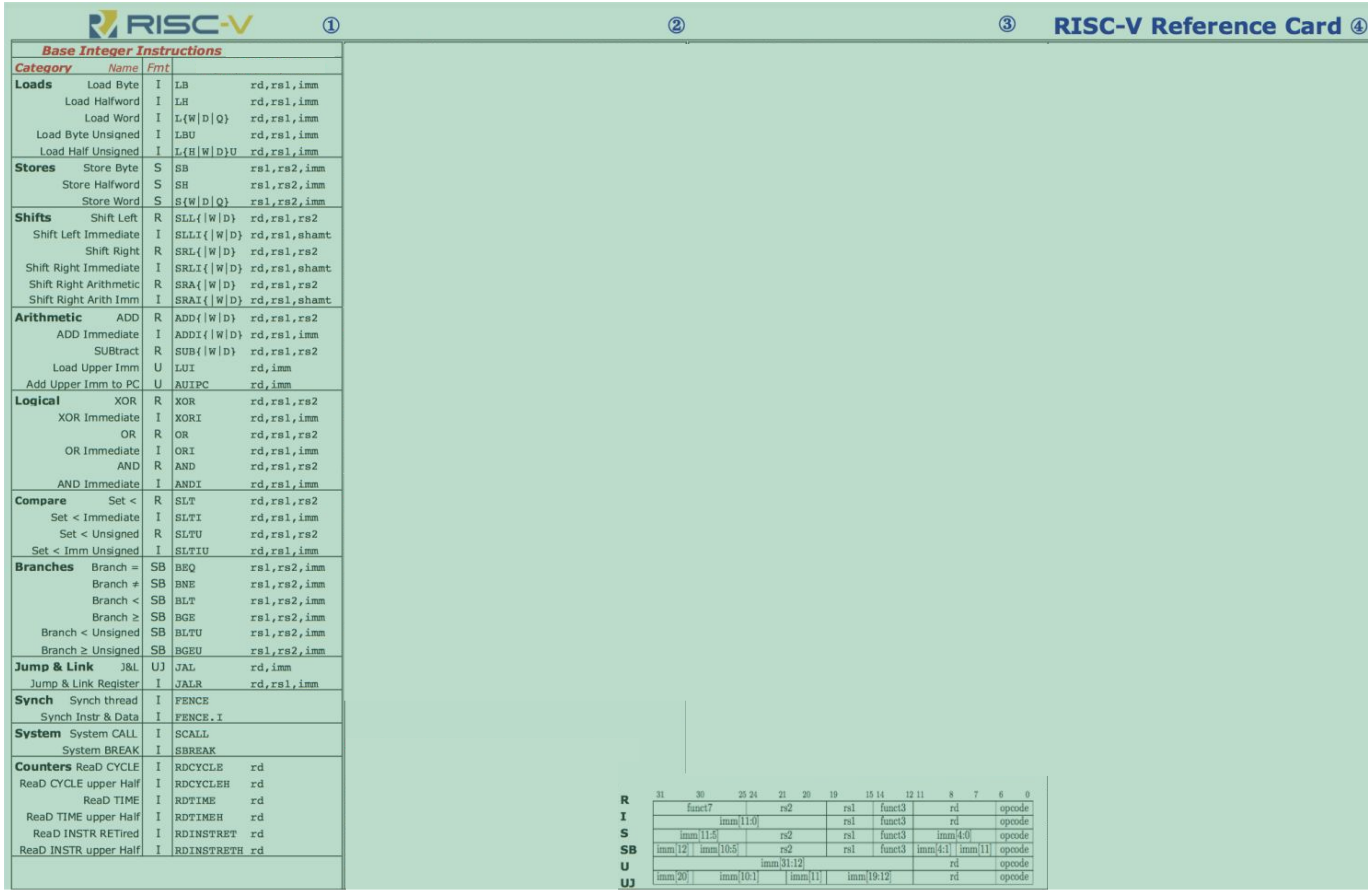
RISC-V Instructions

Base Integer ISA Encoding

- **32-bit fixed-width, naturally aligned instructions**
- **rd/rs1/rs2 in fixed location, no implicit registers**
- **Immediate field (instr[31]) always sign-extended**
- **Instruction Encoding Types**
  - R-type – Register
  - I-type – Immediate
  - S-type – Stores
  - U-Type – Loads with immediate
- **Reserved opcode space for custom instructions**
  - This opcode space will not be used by future standard extensions
  - instr[6:0] = 0b0001011 and 0b0101011



RISC-V Reference Card



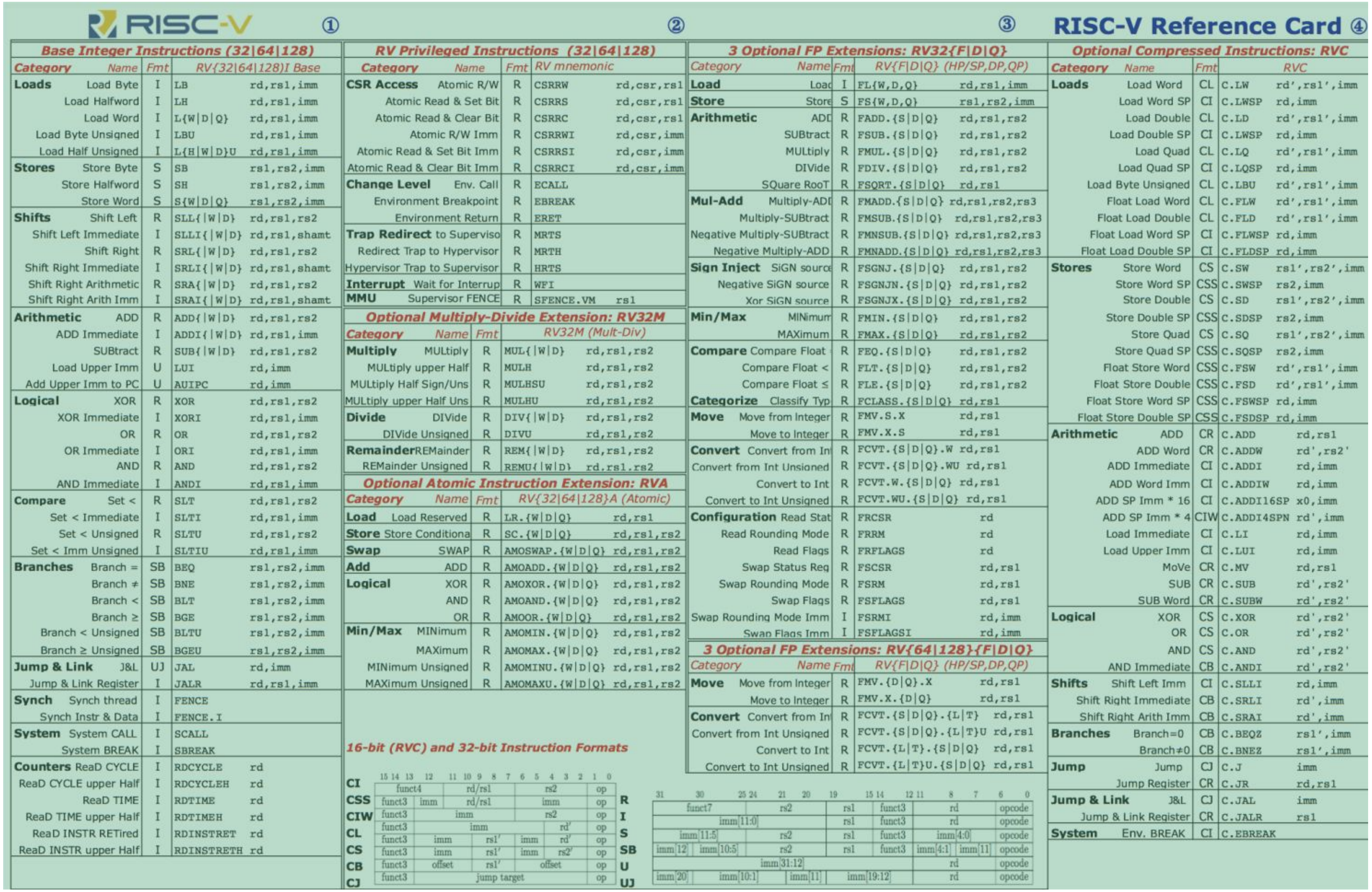
RISC-V ① ② ③ **RISC-V Reference Card** ④

Base Integer Instructions

Category	Name	Fmt		
Loads	Load Byte	I	`LB`	`rd,rs1,imm`
	Load Halfword	I	`LH`	`rd,rs1,imm`
	Load Word	I	`L{W|D|Q}`	`rd,rs1,imm`
	Load Byte Unsigned	I	`LBU`	`rd,rs1,imm`
	Load Half Unsigned	I	`L{H|W|D}U`	`rd,rs1,imm`
Stores	Store Byte	S	`SB`	`rs1,rs2,imm`
	Store Halfword	S	`SH`	`rs1,rs2,imm`
	Store Word	S	`S{W|D|Q}`	`rs1,rs2,imm`
Shifts	Shift Left	R	`SLL{|W|D}`	`rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI{|W|D}`	`rd,rs1,shamt`
	Shift Right	R	`SRL{|W|D}`	`rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI{|W|D}`	`rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA{|W|D}`	`rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI{|W|D}`	`rd,rs1,shamt`
Arithmetic	ADD	R	`ADD{|W|D}`	`rd,rs1,rs2`
	ADD Immediate	I	`ADDI{|W|D}`	`rd,rs1,imm`
	SUBtract	R	`SUB{|W|D}`	`rd,rs1,rs2`
	Load Upper Imm	U	`LUI`	`rd,imm`
	Add Upper Imm to PC	U	`AUIPC`	`rd,imm`
Logical	XOR	R	`XOR`	`rd,rs1,rs2`
	XOR Immediate	I	`XORI`	`rd,rs1,imm`
	OR	R	`OR`	`rd,rs1,rs2`
	OR Immediate	I	`ORI`	`rd,rs1,imm`
	AND	R	`AND`	`rd,rs1,rs2`
	AND Immediate	I	`ANDI`	`rd,rs1,imm`
Compare	Set <	R	`SLT`	`rd,rs1,rs2`
	Set < Immediate	I	`SLTI`	`rd,rs1,imm`
	Set < Unsigned	R	`SLTU`	`rd,rs1,rs2`
	Set < Imm Unsigned	I	`SLTIU`	`rd,rs1,imm`
Branches	Branch =	SB	`BEQ`	`rs1,rs2,imm`
	Branch ≠	SB	`BNE`	`rs1,rs2,imm`
	Branch <	SB	`BLT`	`rs1,rs2,imm`
	Branch ≥	SB	`BGE`	`rs1,rs2,imm`
	Branch < Unsigned	SB	`BLTU`	`rs1,rs2,imm`
	Branch ≥ Unsigned	SB	`BGEU`	`rs1,rs2,imm`
Jump & Link	J&L	UJ	`JAL`	`rd,imm`
	Jump & Link Register	I	`JALR`	`rd,rs1,imm`
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE`	`rd`
	ReaD CYCLE upper Half	I	`RDCYCLEH`	`rd`
	ReaD TIME	I	`RDTIME`	`rd`
	ReaD TIME upper Half	I	`RDTIMEH`	`rd`
	ReaD INSTR RETired	I	`RDINSTRET`	`rd`
	ReaD INSTR upper Half	I	`RDINSTRETH`	`rd`

	31	30 – 25	24 – 21	20	19 – 15	14 – 12	11 – 8	7	6 – 0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

RISC-V Reference Card



RISC-V ① ② ③ **RISC-V Reference Card** ④

Base Integer Instructions (32|64|128)

Category	Name	Fmt	RV{32|64|128}I Base	
Loads	Load Byte	I	`LB`	`rd,rs1,imm`
	Load Halfword	I	`LH`	`rd,rs1,imm`
	Load Word	I	`L{W|D|Q}`	`rd,rs1,imm`
	Load Byte Unsigned	I	`LBU`	`rd,rs1,imm`
	Load Half Unsigned	I	`L{H|W|D}U`	`rd,rs1,imm`
Stores	Store Byte	S	`SB`	`rs1,rs2,imm`
	Store Halfword	S	`SH`	`rs1,rs2,imm`
	Store Word	S	`S{W|D|Q}`	`rs1,rs2,imm`
Shifts	Shift Left	R	`SLL{|W|D}`	`rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI{|W|D}`	`rd,rs1,shamt`
	Shift Right	R	`SRL{|W|D}`	`rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI{|W|D}`	`rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA{|W|D}`	`rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI{|W|D}`	`rd,rs1,shamt`
Arithmetic	ADD	R	`ADD{|W|D}`	`rd,rs1,rs2`
	ADD Immediate	I	`ADDI{|W|D}`	`rd,rs1,imm`
	SUBtract	R	`SUB{|W|D}`	`rd,rs1,rs2`
	Load Upper Imm	U	`LUI`	`rd,imm`
	Add Upper Imm to PC	U	`AUIPC`	`rd,imm`
Logical	XOR	R	`XOR`	`rd,rs1,rs2`
	XOR Immediate	I	`XORI`	`rd,rs1,imm`
	OR	R	`OR`	`rd,rs1,rs2`
	OR Immediate	I	`ORI`	`rd,rs1,imm`
	AND	R	`AND`	`rd,rs1,rs2`
	AND Immediate	I	`ANDI`	`rd,rs1,imm`
Compare	Set <	R	`SLT`	`rd,rs1,rs2`
	Set < Immediate	I	`SLTI`	`rd,rs1,imm`
	Set < Unsigned	R	`SLTU`	`rd,rs1,rs2`
	Set < Imm Unsigned	I	`SLTIU`	`rd,rs1,imm`
Branches	Branch =	SB	`BEQ`	`rs1,rs2,imm`
	Branch ≠	SB	`BNE`	`rs1,rs2,imm`
	Branch <	SB	`BLT`	`rs1,rs2,imm`
	Branch ≥	SB	`BGE`	`rs1,rs2,imm`
	Branch < Unsigned	SB	`BLTU`	`rs1,rs2,imm`
	Branch ≥ Unsigned	SB	`BGEU`	`rs1,rs2,imm`
Jump & Link	J&L	UJ	`JAL`	`rd,imm`
	Jump & Link Register	I	`JALR`	`rd,rs1,imm`
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE`	`rd`
	ReaD CYCLE upper Half	I	`RDCYCLEH`	`rd`
	ReaD TIME	I	`RDTIME`	`rd`
	ReaD TIME upper Half	I	`RDTIMEH`	`rd`
	ReaD INSTR RETired	I	`RDINSTRET`	`rd`
	ReaD INSTR upper Half	I	`RDINSTRETH`	`rd`

RV Privileged Instructions (32|64|128)

Category	Name	Fmt	RV mnemonic	
CSR Access	Atomic R/W	R	`CSRRW`	`rd,csr,rs1`
	Atomic Read & Set Bit	R	`CSRRS`	`rd,csr,rs1`
	Atomic Read & Clear Bit	R	`CSRRC`	`rd,csr,rs1`
	Atomic R/W Imm	R	`CSRRWI`	`rd,csr,imm`
	Atomic Read & Set Bit Imm	R	`CSRRSI`	`rd,csr,imm`
	Atomic Read & Clear Bit Imm	R	`CSRRCI`	`rd,csr,imm`
Change Level	Env. Call	R	`ECALL`	
	Environment Breakpoint	R	`EBREAK`	
	Environment Return	R	`ERET`	
Trap Redirect	to Superviso	R	`MRTS`	
	Redirect Trap to Hypervisor	R	`MRTH`	
	Hypervisor Trap to Supervisor	R	`HRTS`	
Interrupt	Wait for Interrup	R	`WFI`	
MMU	Supervisor FENCE	R	`SFENCE.VM`	`rs1`

Optional Multiply-Divide Extension: RV32M

Category	Name	Fmt	RV32M (Mult-Div)	
Multiply	MULtiply	R	`MUL{|W|D}`	`rd,rs1,rs2`
	MULtiply upper Half	R	`MULH`	`rd,rs1,rs2`
	MULtiply Half Sign/Uns	R	`MULHSU`	`rd,rs1,rs2`
	MULtiply upper Half Uns	R	`MULHU`	`rd,rs1,rs2`
Divide	DIVide	R	`DIV{|W|D}`	`rd,rs1,rs2`
	DIVide Unsigned	R	`DIVU`	`rd,rs1,rs2`
Remainder	REMainder	R	`REM{|W|D}`	`rd,rs1,rs2`
	REMainder Unsigned	R	`REMU{|W|D}`	`rd,rs1,rs2`

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV{32|64|128}A (Atomic)	
Load	Load Reserved	R	`LR.{W|D|Q}`	`rd,rs1`
Store	Store Conditiona	R	`SC.{W|D|Q}`	`rd,rs1,rs2`
Swap	SWAP	R	`AMOSWAP.{W|D|Q}`	`rd,rs1,rs2`
Add	ADD	R	`AMOADD.{W|D|Q}`	`rd,rs1,rs2`
Logical	XOR	R	`AMOXOR.{W|D|Q}`	`rd,rs1,rs2`
	AND	R	`AMOAND.{W|D|Q}`	`rd,rs1,rs2`
	OR	R	`AMOOR.{W|D|Q}`	`rd,rs1,rs2`
Min/Max	MINimum	R	`AMOMIN.{W|D|Q}`	`rd,rs1,rs2`
	MAXimum	R	`AMOMAX.{W|D|Q}`	`rd,rs1,rs2`
	MINimum Unsigned	R	`AMOMINU.{W|D|Q}`	`rd,rs1,rs2`
	MAXimum Unsigned	R	`AMOMAXU.{W|D|Q}`	`rd,rs1,rs2`

3 Optional FP Extensions: RV32{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)	
Load	Load	I	`FL{W,D,Q}`	`rd,rs1,imm`
Store	Store	S	`FS{W,D,Q}`	`rs1,rs2,imm`
Arithmetic	ADD	R	`FADD.{S|D|Q}`	`rd,rs1,rs2`
	SUBtract	R	`FSUB.{S|D|Q}`	`rd,rs1,rs2`
	MULtiply	R	`FMUL.{S|D|Q}`	`rd,rs1,rs2`
	DIVide	R	`FDIV.{S|D|Q}`	`rd,rs1,rs2`
	SQuare RooT	R	`FSQRT.{S|D|Q}`	`rd,rs1`
Mul-Add	Multiply-ADD	R	`FMADD.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Multiply-SUBtract	R	`FMSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Negative Multiply-SUBtract	R	`FMNSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Negative Multiply-ADD	R	`FMNADD.{S|D|Q}`	`rd,rs1,rs2,rs3`
Sign Inject	SiGN source	R	`FSGNJ.{S|D|Q}`	`rd,rs1,rs2`
	Negative SiGN source	R	`FSGNJN.{S|D|Q}`	`rd,rs1,rs2`
	Xor SiGN source	R	`FSGNJX.{S|D|Q}`	`rd,rs1,rs2`
Min/Max	MINimum	R	`FMIN.{S|D|Q}`	`rd,rs1,rs2`
	MAXimum	R	`FMAX.{S|D|Q}`	`rd,rs1,rs2`
Compare	Compare Float =	R	`FEQ.{S|D|Q}`	`rd,rs1,rs2`
	Compare Float <	R	`FLT.{S|D|Q}`	`rd,rs1,rs2`
	Compare Float ≤	R	`FLE.{S|D|Q}`	`rd,rs1,rs2`
Categorize	Classify Typ	R	`FCLASS.{S|D|Q}`	`rd,rs1`
Move	Move from Integer	R	`FMV.S.X`	`rd,rs1`
	Move to Integer	R	`FMV.X.S`	`rd,rs1`
Convert	Convert from Int	R	`FCVT.{S|D|Q}.W`	`rd,rs1`
	Convert from Int Unsigned	R	`FCVT.{S|D|Q}.WU`	`rd,rs1`
	Convert to Int	R	`FCVT.W.{S|D|Q}`	`rd,rs1`
	Convert to Int Unsigned	R	`FCVT.WU.{S|D|Q}`	`rd,rs1`
Configuration	Read Stat	R	`FRCSR`	`rd`
	Read Rounding Mode	R	`FRRM`	`rd`
	Read Flags	R	`FRFLAGS`	`rd`
	Swap Status Reg	R	`FSCSR`	`rd,rs1`
	Swap Rounding Mode	R	`FSRM`	`rd,rs1`
	Swap Flags	R	`FSFLAGS`	`rd,rs1`
	Swap Rounding Mode Imm	I	`FSRMI`	`rd,imm`
	Swap Flags Imm	I	`FSFLAGSI`	`rd,imm`

3 Optional FP Extensions: RV{64|128}{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)	
Move	Move from Integer	R	`FMV.{D|Q}.X`	`rd,rs1`
	Move to Integer	R	`FMV.X.{D|Q}`	`rd,rs1`
Convert	Convert from Int	R	`FCVT.{S|D|Q}.{L|T}`	`rd,rs1`
	Convert from Int Unsigned	R	`FCVT.{S|D|Q}.{L|T}U`	`rd,rs1`
	Convert to Int	R	`FCVT.{L|T}.{S|D|Q}`	`rd,rs1`
	Convert to Int Unsigned	R	`FCVT.{L|T}U.{S|D|Q}`	`rd,rs1`

Optional Compressed Instructions: RVC

Category	Name	Fmt	RVC	
Loads	Load Word	CL	`C.LW`	`rd',rs1',imm`
	Load Word SP	CI	`C.LWSP`	`rd,imm`
	Load Double	CL	`C.LD`	`rd',rs1',imm`
	Load Double SP	CI	`C.LWSP`	`rd,imm`
	Load Quad	CL	`C.LQ`	`rd',rs1',imm`
	Load Quad SP	CI	`C.LQSP`	`rd,imm`
	Load Byte Unsigned	CL	`C.LBU`	`rd',rs1',imm`
	Float Load Word	CL	`C.FLW`	`rd',rs1',imm`
	Float Load Double	CL	`C.FLD`	`rd',rs1',imm`
	Float Load Word SP	CI	`C.FLWSP`	`rd,imm`
	Float Load Double SP	CI	`C.FLDSP`	`rd,imm`
Stores	Store Word	CS	`C.SW`	`rs1',rs2',imm`
	Store Word SP	CSS	`C.SWSP`	`rs2,imm`
	Store Double	CS	`C.SD`	`rs1',rs2',imm`
	Store Double SP	CSS	`C.SDSP`	`rs2,imm`
	Store Quad	CS	`C.SQ`	`rs1',rs2',imm`
	Store Quad SP	CSS	`C.SQSP`	`rs2,imm`
	Float Store Word	CSS	`C.FSW`	`rd',rs1',imm`
	Float Store Double	CSS	`C.FSD`	`rd',rs1',imm`
	Float Store Word SP	CSS	`C.FSWSP`	`rd,imm`
	Float Store Double SP	CSS	`C.FSDSP`	`rd,imm`
Arithmetic	ADD	CR	`C.ADD`	`rd,rs1`
	ADD Word	CR	`C.ADDW`	`rd',rs2'`
	ADD Immediate	CI	`C.ADDI`	`rd,imm`
	ADD Word Imm	CI	`C.ADDIW`	`rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP`	`x0,imm`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN`	`rd',imm`
	Load Immediate	CI	`C.LI`	`rd,imm`
	Load Upper Imm	CI	`C.LUI`	`rd,imm`
	MoVe	CR	`C.MV`	`rd,rs1`
	SUB	CR	`C.SUB`	`rd',rs2'`
	SUB Word	CR	`C.SUBW`	`rd',rs2'`
Logical	XOR	CS	`C.XOR`	`rd',rs2'`
	OR	CS	`C.OR`	`rd',rs2'`
	AND	CS	`C.AND`	`rd',rs2'`
	AND Immediate	CB	`C.ANDI`	`rd',rs2'`
Shifts	Shift Left Imm	CI	`C.SLLI`	`rd,imm`
	Shift Right Immediate	CB	`C.SRLI`	`rd',imm`
	Shift Right Arith Imm	CB	`C.SRAI`	`rd',imm`
Branches	Branch=0	CB	`C.BEQZ`	`rs1',imm`
	Branch≠0	CB	`C.BNEZ`	`rs1',imm`
Jump	Jump	CJ	`C.J`	`imm`
	Jump Register	CR	`C.JR`	`rd,rs1`
Jump & Link	J&L	CJ	`C.JAL`	`imm`
	Jump & Link Register	CR	`C.JALR`	`rs1`
System	Env. BREAK	CI	`C.EBREAK`	

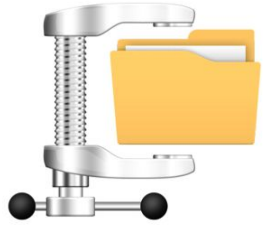
16-bit (RVC) and 32-bit Instruction Formats

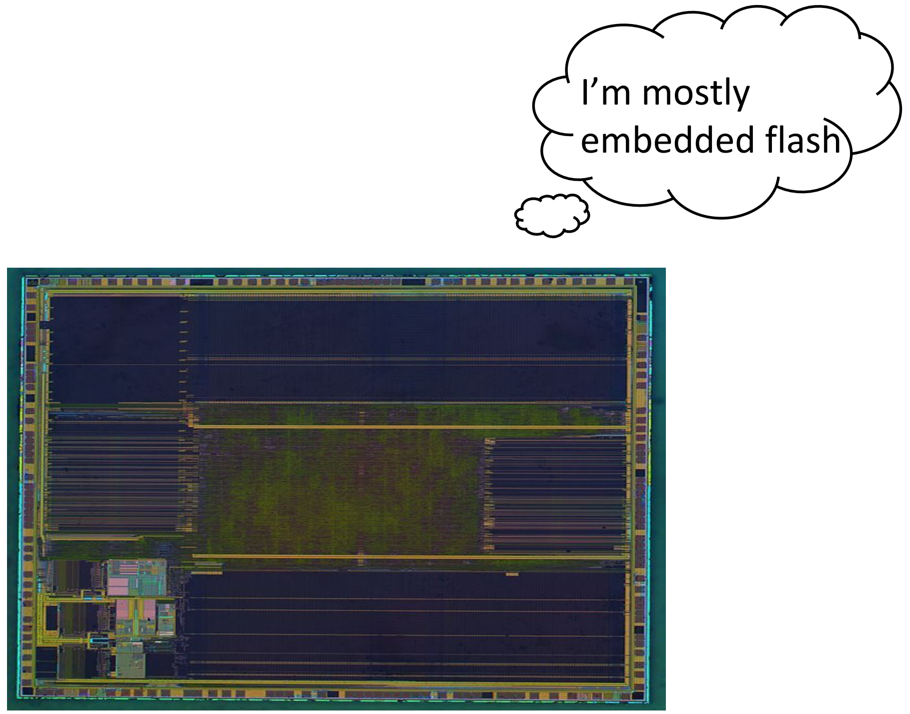
	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CI	funct4		rd/rs1		rs2		op
CSS	funct3	imm	rd/rs1		imm		op
CIW	funct3	imm			rs2		op
CL	funct3	imm				rd′	op
	funct3	imm		rs1′	imm	rd′	op
CS	funct3	imm		rs1′	imm	rs2′	op
CB	funct3	offset		rs1′	offset		op
CJ	funct3	jump target					op

	31	30 25	24 21	20	19 15	14 12	11 8	7	6 0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

Compressed Instructions (C Extension)

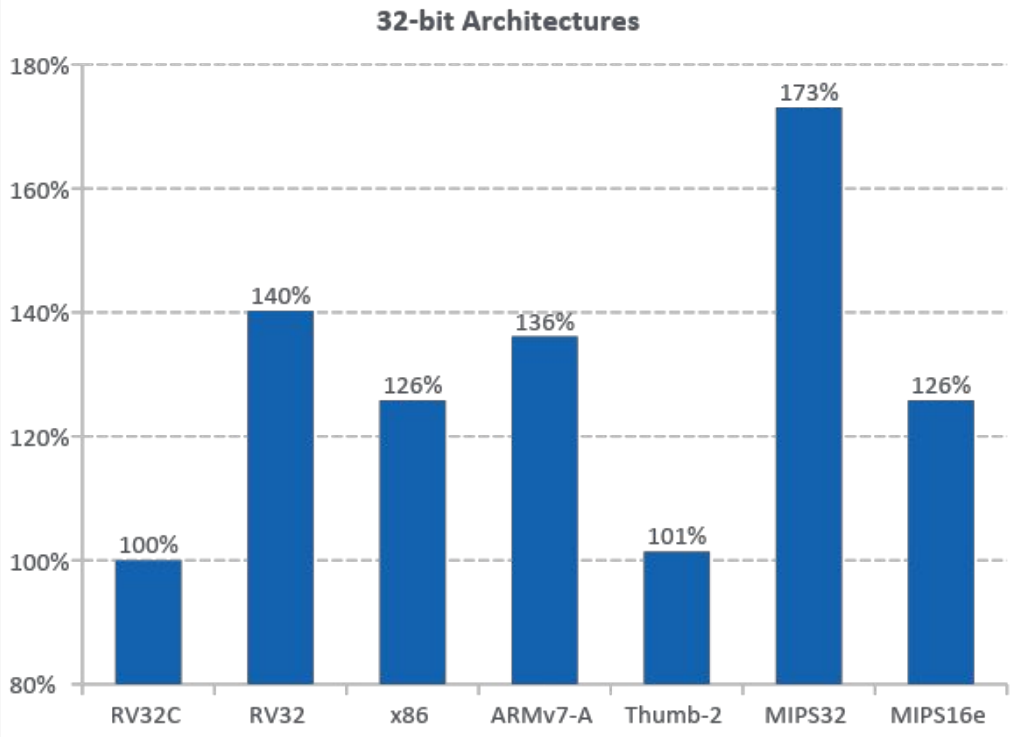
- **Most base integer instructions "Compress" to 16-bit equivalents**
  - 1:1 mapping of compressed instructions to standard instructions
- **Smaller code size can reduce cost in embedded systems**
  - Directly resulting in smaller Flash/ROM/RAM
- **Smaller code size can increase performance and reduce power**
  - Better utilization of Cache RAMs
  - Fewer transactions across high power interfaces (DRAM, Flash, etc…)
- **RV64 can also use the C Extension**

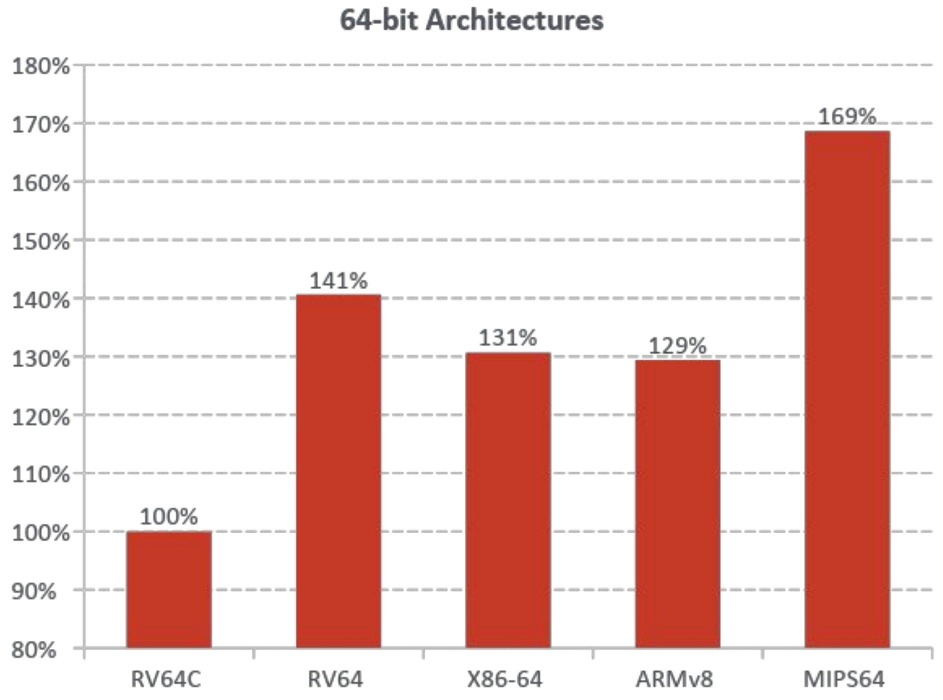




A Microcontroller

Code Size Comparison - SpecInt 2006





- **RISC-V is smallest ISA for 32- and 64-bit processors in SpecInt2k6**
- **All results with same GCC compiler and options**

Atomics (A Extension)

- **Atomic memory operations (AMO) perform Read-Modify-Write operations in a single Atomic instruction**
  - Logical, Arithmetic, Swap
  - Acquire (aq) and Release (rl) bits for release consistency

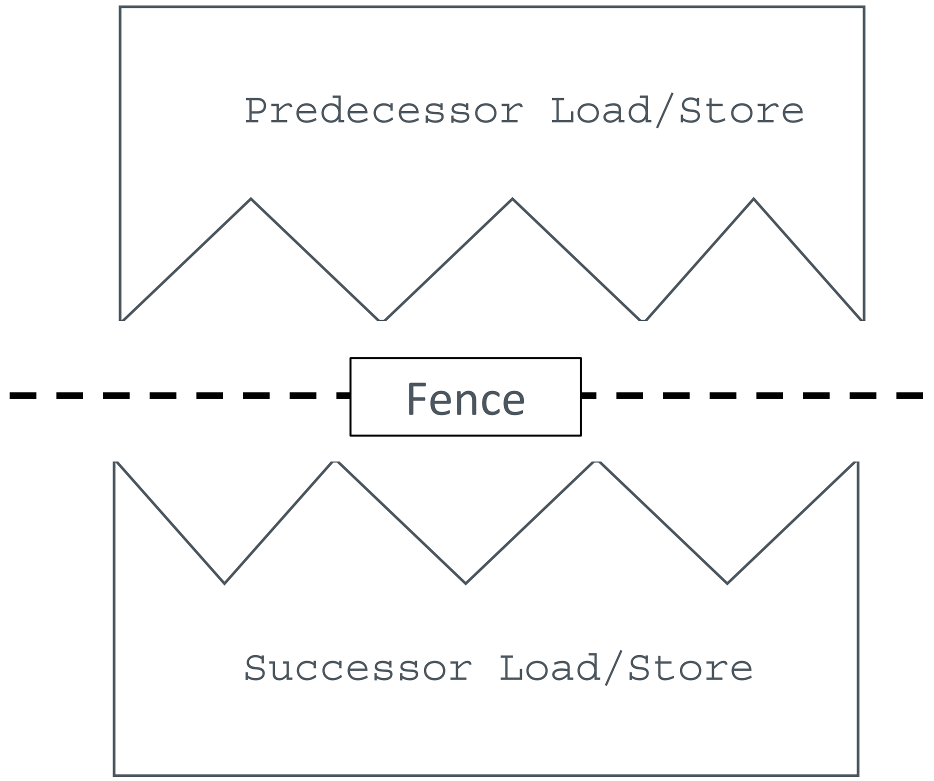
- **Load-Reserved/Store-Conditional pairs**
  - Guaranteed forward progress for short sequences

```
   li t0, 1 # Initialize swap value.
again:
   amoswap.w.aq t0, t0, (a0) # Attempt to acquire
lock.
   bnez t0, again # Retry if held.
   # ...
   # Critical section.
   # ...
   amoswap.w.rl x0, x0, (a0) # Release lock by
storing 0.
```

Example RISC-V Spinlock

Fence Instructions

- **Fences are used to enforce program order on device I/O and memory accesses**
- **FENCE instruction format**
  - FENCE predecessor, successor
  - Predecessor/successor can be
    - R,W,I,O
  - FENCE RWIO, RWIO – full barrier



CSR and ECALL Instructions

- **Control and Status Registers (CSRs) have their own dedicated instructions :**
  - Read/Write
  - Read and Set bit
  - Read and Clear bit
- **Environment Call instruction used to transfer control to the execution environment and a higher privileged mode**
  - Triggers a synchronous Interrupt (discussed later)
  - Example: User mode program can use an ECALL to transfer control to a Machine mode OS kernel, aka System Call

SiFive

RISC-V Control and Status Registers (CSR)

What are Control and Status Registers (CSRs)

- **CSRs are Registers which contain the working state of a RISC-V machine**

- **CSRs are specific to a Mode**
  - Machine Mode has ~17 CSRs (not including performance monitor CSRs)
  - Supervisor Mode has a similar number, though most are subsets of their equivalent Machine Mode CSRs
    - Machine Mode can also access Supervisor CSRs

- **CSRs are defined in the RISC-V privileged specification**
  - We will cover a few key CSRs here

Identification CSRs

- ***misa* – Machine ISA Register**
  - Reports the ISA supported by the hart (i.e. RV32IMAC)
  -
- ***mhartid* – Machine hart ID**
  - Integer ID of the Hardware Thread
  -
- ***mvendorid* – Machine Vendor ID**
  - JEDEC Vendor ID
  -
- ***marchid* – Machine Architecture ID**
  - Used along with *mvendorid* to identify a implementation. No format specified
  -
- ***mimpid* - Machine Implementation ID**
  - Implementation defined format

Machine Status (*mstatus)* - The Most Important CSR

Control and track the hart's current operating state

Bits	Field Name	Description
0	UIE	User Interrupt Enable
1	SIE	Supervisor Interrupt Enable
2	Reserved	
3	MIE	Machine Interrupt Enable
4	UPIE	User Previous Interrupt Enable
5	SPIE	Supervisor Previous Interrupt Enable
6	Reserved	
7	MPIE	Machine Previous Interrupt Enabler
8	SPP	Supervisor Previous Privilege
[10:9]	Reserved	
[12:11]	MPP	Machine Previous Privilege

Bits	Field Name	Description
[14:13]	FS	Floating Point State
[16:15]	XS	User Mode Extension State
17	MPRIV	Modify Privilege (access memory as MPP)
18	SUM	Permit Supervisor User Memory Access
19	MXR	Make Executable Readable
20	TVM	Trap Virtual memory
21	TW	Timeout Wait (traps S-Mode wfi)
22	TSR	Trap SRET
[23:30]	Reserved	
[31]	SD	State Dirty (FS and XS summary bit)

RV32 *mstatus* CSR

Timer CSRs

- ***mtime***
  - RISC-V defines a requirement for a counter exposed as a memory mapped register
  - There is no frequency requirement on the timer, but
    - It must run at a constant frequency
    - The platform must expose frequency

- ***mtimecmp***
  - RISC-V defines a memory mapped timer compare register
  - Triggers an interrupt when *mtime* is greater than or equal to *mtimecmp*

Bits	Field Name	Description
[63:0]	mtime	Machine Time Register

mtime CSR

Bits	Field Name	Description
[63:0]	mtimecmp	Machine Time Compare Register

mtimecmp CSR

Supervisor CSRs

- **Most of the Machine mode CSRs have Supervisor mode equivalents**
  - Supervisor mode CSRs can be used to control the state of Supervisor and User Modes.
  - Most equivalent Supervisor CSRs have the same mapping as Machine mode without Machine mode control bits
  - *sstatus, stvec, sip, sie, sepc, scause, satp*, and more
- ***satp*** **- Supervisor Address Translation and Protection Register**
  - Used to control Supervisor mode address translation and protection

Bits	Field Name	Description
[21:0]	PPN	Physical Page Number of the root page table
[30:22]	ASID	Address Space Identifier
31	MODE	MODE=1 uses Sv32 Address Translation

RV32 satp CSR

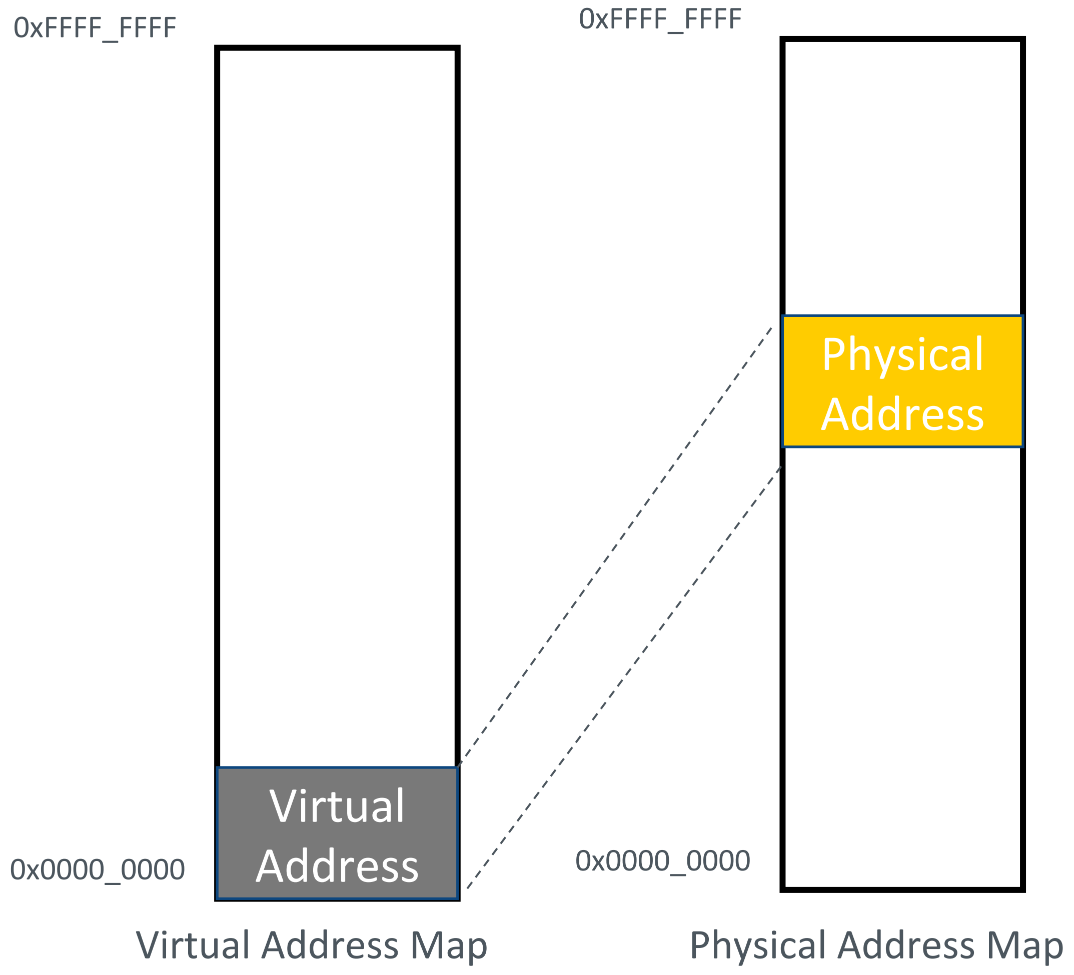
Bits	Field Name	Description
[43:0]	PPN	Physical Page Number of the root page table
[59:44]	ASID	Address Space Identifier
[63:60]	MODE	Encodings for Sv32, Sv39, Sv48

RV64 satp CSR

Virtual Memory

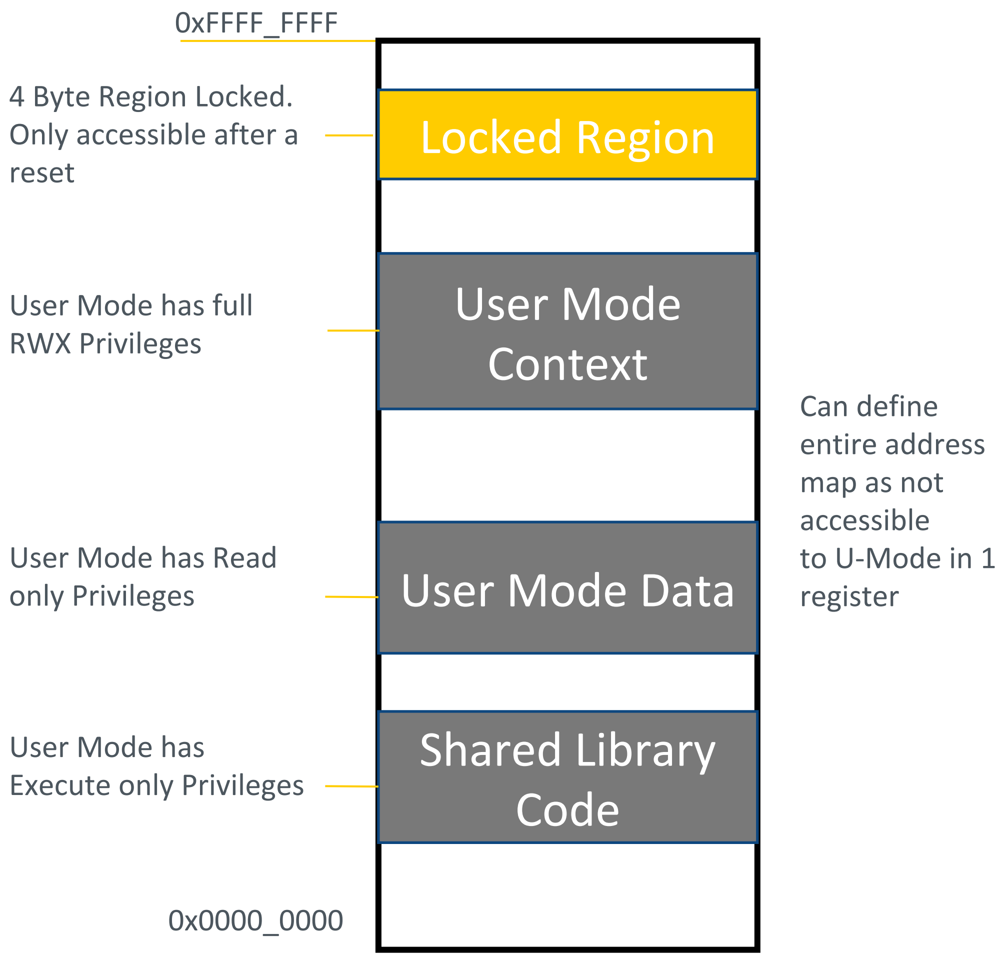
- **RISC-V has support for Virtual Memory allowing for sophisticated memory management and OS support (Linux)**

- **Requires an S-Mode implementation**
- **Sv32**
  - 32bit Virtual Address
  - 4KiB, 4MiB page tables (2 Levels)
- **Sv39 (requires an RV64 implementation)**
  - 39bit Virtual Address
  - 4KiB, 2MiB, 1GiB page tables (3 Levels)
- **Sv48 (requires an RV64 implementation)**
  - 48bit Virtual Address
  - 4KiB, 2MiB, 1 GiB, 512GB page tables (4 Levels)
- **Page Tables also contain access permission attributes**



Physical Memory Protection (PMP)

- **Can be used to enforce access restrictions on less privileged modes**
  - Prevent Supervisor and User Mode software from accessing unwanted memory
- **Up to 16 regions with a minimum region size of 4 bytes**
- **Ability to Lock a region**
  - A locked region enforces permissions on all accesses, including M-Mode
  - Only way to unlock a region is a Reset


0xFFFF_FFFF
4 Byte Region Locked. Only accessible after a reset
Locked Region
User Mode has full RWX Privileges
User Mode Context
Can define entire address map as not accessible to U-Mode in 1 register
User Mode has Read only Privileges
User Mode Data
User Mode has Execute only Privileges
Shared Library Code
0x0000_0000
Example PMP Memory Map

SiFive

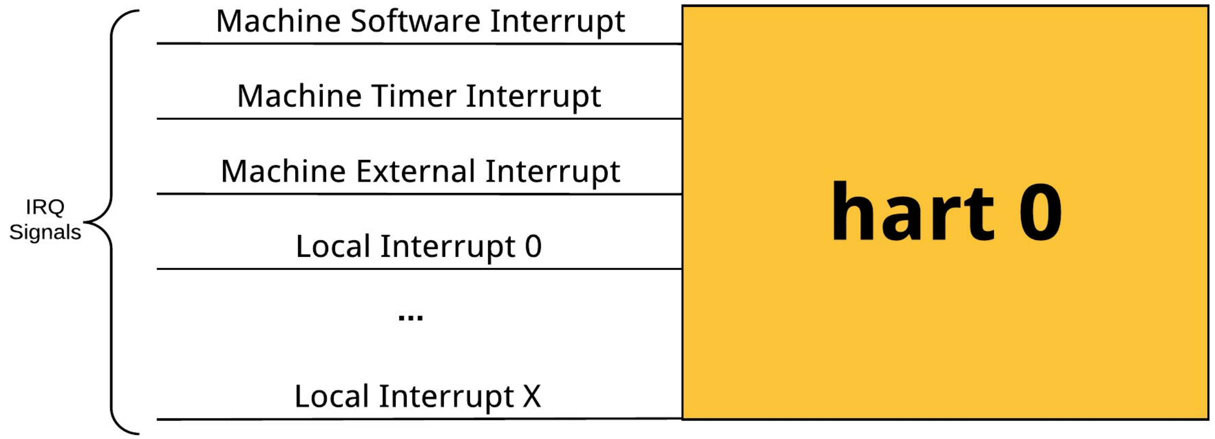
RISC-V Interrupts

RISC-V Interrupts

- **RISC-V defines the following interrupts per Hart**
  - Software – architecturally defined software interrupt
  - Timer – architecturally defined timer interrupt
  - External – Peripheral Interrupts
  - Local - Hart specific Peripheral Interrupts

- **Optionally per privilege level**
  - Can have Supervisor Software/Timer/Machine Interrupts
  - Can have User Software/Timer/Machine

- **Local interrupts are optional and implementation specific**
  - Can be used for hart-specific peripheral interrupts
  - Useful for latency-sensitive embedded systems or small embedded systems with a small number of interrupts



Machine Status (*mstatus*) – As it relates to Interrupts

Bits	Field Name	Description
0	**UIE**	**User Interrupt Enable**
1	**SIE**	**Supervisor Interrupt Enable**
2	Reserved	
3	**MIE**	**Machine Interrupt Enable**
4	**UPIE**	**User Previous Interrupt Enable**
5	**SPIE**	**Supervisor Previous Interrupt Enable**
6	Reserved	
7	**MPIE**	**Machine Previous Interrupt Enabler**
8	**SPP**	**Supervisor Previous Privilege**
[10:9]	Reserved	
[12:11]	**MPP**	**Machine Previous Privilege**

Bits	Field Name	Description
[14:13]	FS	Floating Point State
[16:15]	XS	User Mode Extension State
17	MPRIV	Modify Privilege (access memory as MPP)
18	SUM	Permit Supervisor User Memory Access
19	MXR	Make Executable Readable
20	TVM	Trap Virtual memory
21	TW	Timeout Wait (traps S-Mode wfi)
22	TSR	Trap SRET
[23:30]	Reserved	
[31]	SD	State Dirty (FS and XS summary bit)

RV32 *mstatus* CSR

- **M/S/U IE – Global Interrupt Enables for Modes which supports interrupts**
- **M/S/U PIE – Encodes the state of interrupt enables prior to an interrupt.**
  - These bits can also be written to in order to enable interrupts when returning to lower privilege modes
- **M/S PP – Encodes the privilege level prior to the previous interrupt**
  - These bits can also be written to in order to enter a lower privilege mode when executing MRET or SRET instructions

Machine Interrupt Cause CSR (*mcause)*

- **Interrupts are identified by reading the *mcause* CSR**
- **The *interrupt* field determines if a trap was caused by an interrupt or an exception**

Bits	Field Name	Description
XLEN-1	Interrupt	Identifies if an interrupt was synchronous or asynchronous
[XLEN-2:0]	Exception Code	Identifies the exception

mcause CSR

Interrupt = 1 (interrupt)	
Exception Code	Description
0	User Software Interrupt
1	Supervisor Software Interrupt
2	Reserved
3	Machine Software Interrupt
4	User Timer Interrupt
5	Supervisor Timer Interrupt
6	*Reserved*
7	Machine Timer Interrupt
8	User External Interrupt
9	Supervisor External Interrupt
10	*Reserved*
11	Machine External Interrupt
12 - 15	*Reserved*
≥16	Local Interrupt X

Interrupt = 0 (exception)	
Exception Code	Description
0	Instruction Address Misaligned
1	Instruction Access Fault
2	Illegal Instruction
3	Breakpoint
4	Load Address Misaligned
5	Load Access Fault
6	Store/AMO Address Misaligned
7	Store/AMO Access Fault
8	Environment Call from U-mode
9	Environment Call from S-mode
10	Reserved
11	Environment Call from M-mode
12	Instruction Page Fault
13	Load Page Fault
14	Reserved
15	Store/AMO Page Fault
≥16	Reserved

Machine Interrupt-Enable and Pending CSRs (*mie, mip)*

- ***mie*** **used to enable/disable a given interrupt**
- ***mip*** **indicates which interrupts are currently pending**
  - Can be used for polling
- **Lesser-privilege bits in *mip* are writeable**
  - i.e. Machine-mode software can be used to generate a supervisor interrupt by setting the STIP bit
- ***mip*** **has the same mapping as *mie***

Bits	Field Name	Description
0	USIE	User Software Interrupt Enable
1	SSIE	Supervisor Software Interrupt Enable
2	*Reserved*	
3	MSIE	Machine Software Interrupt Enable
4	UTIE	User Timer Interrupt Enable
5	STIE	Supervisor Timer Interrupt Enable
6	*Reserved*	
7	MTIE	Machine Timer Interrupt Enable
8	UEIE	User External Interrupt Enable
9	SEIE	Supervisor External Interrupt Enable
10	*Reserved*	
11	MEIE	Machine External Interrupt Enable
12-15	*Reserved*	
≥16	LIE	Local Interrupt Enable

mie CSR

Machine Trap Vector CSR (*mtvec)*

mtvec sets the Base interrupt vector and the interrupt Mode

Bits	Field Name	Description
[XLEN-1:6]	Base	Machine Trap Vector Base Address. 64-byte Alignment
[1:0]	Mode	MODE Sets the interrupt processing mode.

mtvec CSR

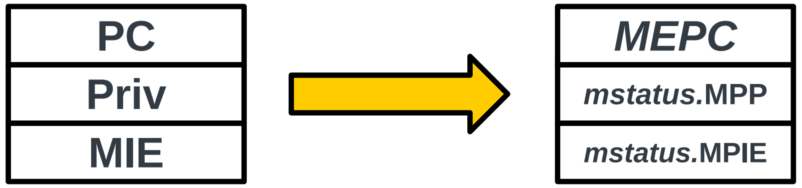
mtvec Modes		
Value	Name	Description
0x0	Direct	All Exceptions set PC to mtvec.BASE Requires 4-Byte alignment
0x1	Vectored	Asynchronous interrupts set pc to mtvec.BASE + (4×mcause.EXCCODE) Requires 4-Byte alignment
> 0x01		Reserved

- ***mtvec.*Mode = Direct**
  - All Interrupts trap to the address *mtvec.*Base
  - Software must read the *mcause* CSR and react accordingly
- ***mtvec*.Vectored**
  - Interrupts trap to the address *mtvec.*Base + (4**mcause.ExCode)*
  - Eliminates the need to read mcause for asynchronous exceptions

Trap Handler – Entry and Exit

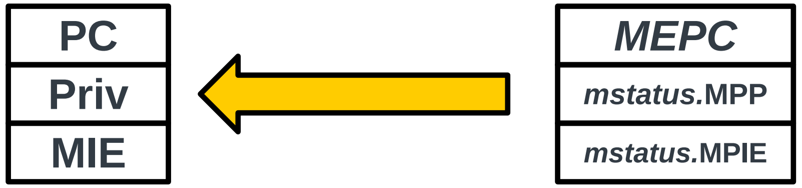
mtevc.MODE = Direct

- **On entry, the RISC-V hart will**
  - Save the current state



  - Then set PC = *mtvec, mstatus*.MIE = 0

- **MRET instruction restores state**



- Typical trap handler software will

```
Push Registers
…
interrupt = mcause.msb
if interrupt
  branch isr_handler[mcause.code]
else
  branch exception_handler[mcause.code]
…
Pop Registers
MRET
```

Interrupt handler pseudo code

Interrupt Handler Code

RISC-V Assembly interrupt handler to Push and Pop register file

```
  .align 2
  .global trap_entry
trap_entry:
  addi sp, sp, -16*REGBYTES

  //store ABI Caller Registers
  STORE x1,   0*REGBYTES(sp)
  STORE x5,   2*REGBYTES(sp)
      …
  STORE x30, 14*REGBYTES(sp)
  STORE x31, 15*REGBYTES(sp)

  //call C Code Handler
  call handle_trap

  //restore ABI Caller Registers
  LOAD x1,   0*REGBYTES(sp)
  LOAD x5,   2*REGBYTES(sp)
      …
  LOAD x30, 14*REGBYTES(sp)
  LOAD x31, 15*REGBYTES(sp)


  addi sp, sp, 16*REGBYTES
  mret
```

C Code Handler determines interrupt cause and branches to the appropriate function

```
void handle_trap()
{
    unsigned long mcause = read_csr(mcause);
    if (mcause & MCAUSE_INT) {
            //mask interrupt bit and branch to handler
            isr_handler[mcause & MCAUSE_CAUSE] ();
    } else {
            //branch to handler
            exception_handler[mcause]();
    }
}


//write trap_entry address to mtvec
write_csr(mtvec, ((unsigned long)&trap_entry));
```

Compiler Interrupt Attribute

- **Pushing and Popping Registers in Assembly is a pain**

- **The *interrupt* attribute was added to GCC to facilitate interrupt handlers written entirely in C**
  - Interrupt functions only saves/restores necessary registers onto the stack
  - Align function on an 8-byte boundary
  - Calles MRET after popping register file back off the stack

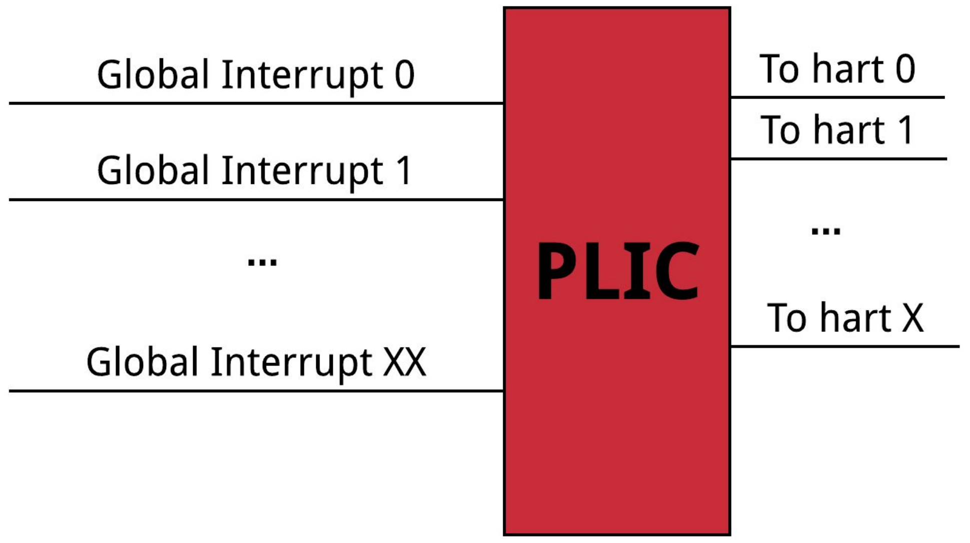
Interrupt handler with *interrupt* attribute.
No assembly Code necessary

```
void handle_trap(void) __attribute((interrupt));
 void handle_trap()
 {
     unsigned long mcause = read_csr(mcause);
     if (mcause & MCAUSE_INT) {
             //mask interrupt bit and branch to handler
             isr_handler[mcause & MCAUSE_CAUSE] ();
     } else {
             //synchronous exception, branch to handler
             exception_handler[mcause & MCAUSE_CAUSE]();
     }
 }


 //write handle_trap address to mtvec
 write_csr(mtvec, ((unsigned long)&handle_trap));
```

RISC-V Global Interrupts

- RISC-V defines Global Interrupts as a Interrupt which can be routed to any hart in a system
- Global Interrupts are prioritized and distributed by the Platform Level Interrupt Controller (PLIC)
- The PLIC is connected to the External Interrupt signal for 1 or more harts in an implementation



PLIC Interrupt Code Example

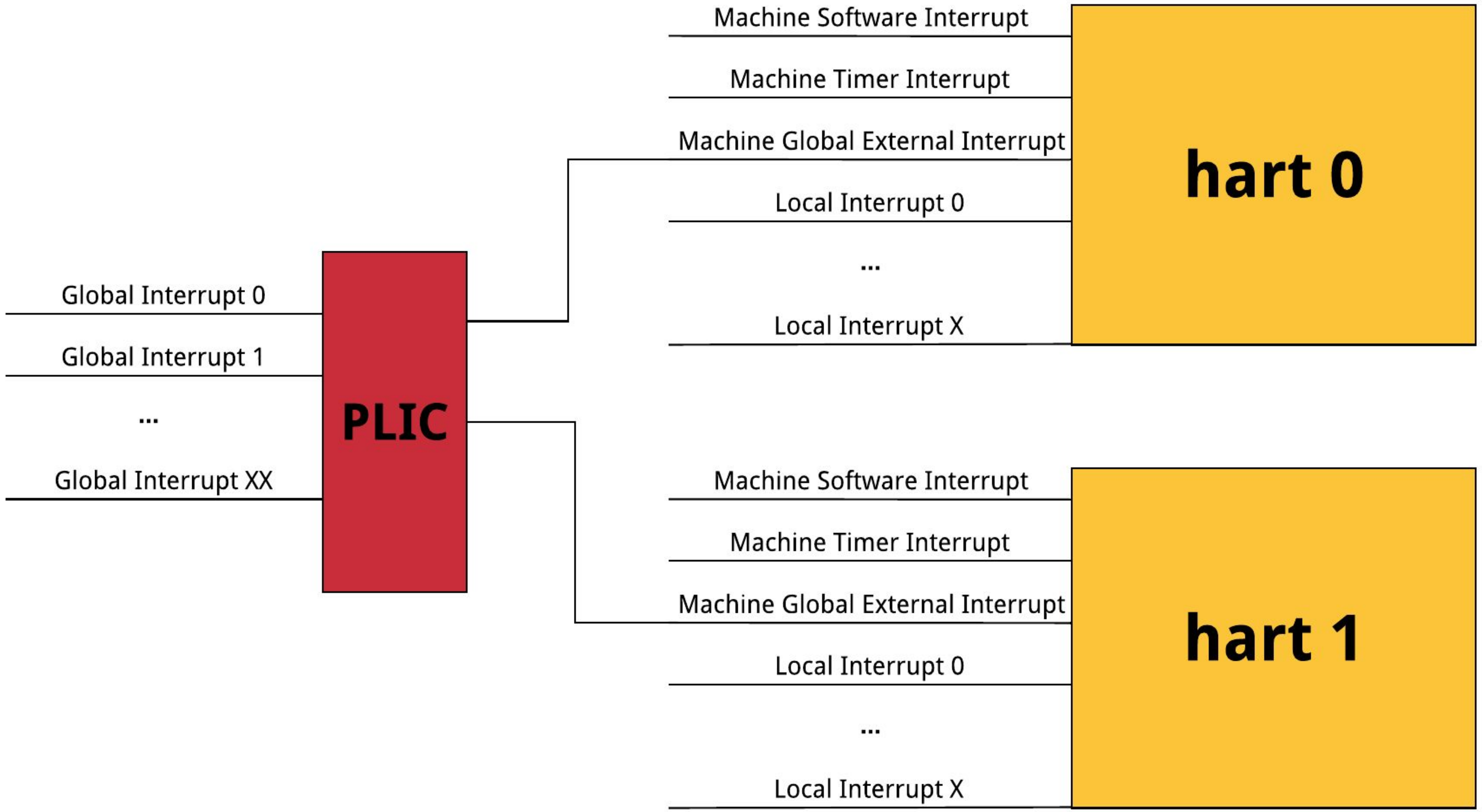
- **In this example an interrupt is presented to the PLIC**
- **The PLIC signals an interrupt to a hart using the Machine External Interrupt (interrupt 11)**
- **The interrupt handler (handle_trap) branches to the defined function to handle the Machine External Interrupt**
  - C Code placed the address of machine_external_interrupt function in location 11 of the async_handler vector table
- **The machine_external_interrupt handler does the following:**
  - Reads the PLIC's claim/complete register to determine highest priority pending interrupt
  - Uses another vector table to branch to the interrupt's specific handler
  - Completes the interrupt by writing the interrupt number back to the PLIC's claim/complete

```
void handle_trap(void) __attribute((interrupt));
void handle_trap()
{
    unsigned long mcause = read_csr(mcause);
    if (mcause & MCAUSE_INT) {
        //mask interrupt bit and branch to handler
        isr_handler[mcause & MCAUSE_CAUSE] ();
    } else {
        //synchronous exception, branch to    handler
        exception_handler[mcause & MCAUSE_CAUSE]();
    }
}

 //install PLIC handler at MEIP Location
isr_handler[11] = machine_external_interrupt;
//write trap_entry address to mtvec
write_csr(mtvec, ((unsigned long)&handle_trap));
```

```
void machine_external_interrupt()
{
   //get the highest priority pending PLIC interrupt
   uint32_t int_num = plic.claim_comlete;
   //branch to handler
   plic_handler[int_num]();
   //complete interrupt by writing interrupt number
   back to PLIC
   plic.claim_complete = int_num;
}
```

RISC-V Interrupt System Architecture (M-mode only example)

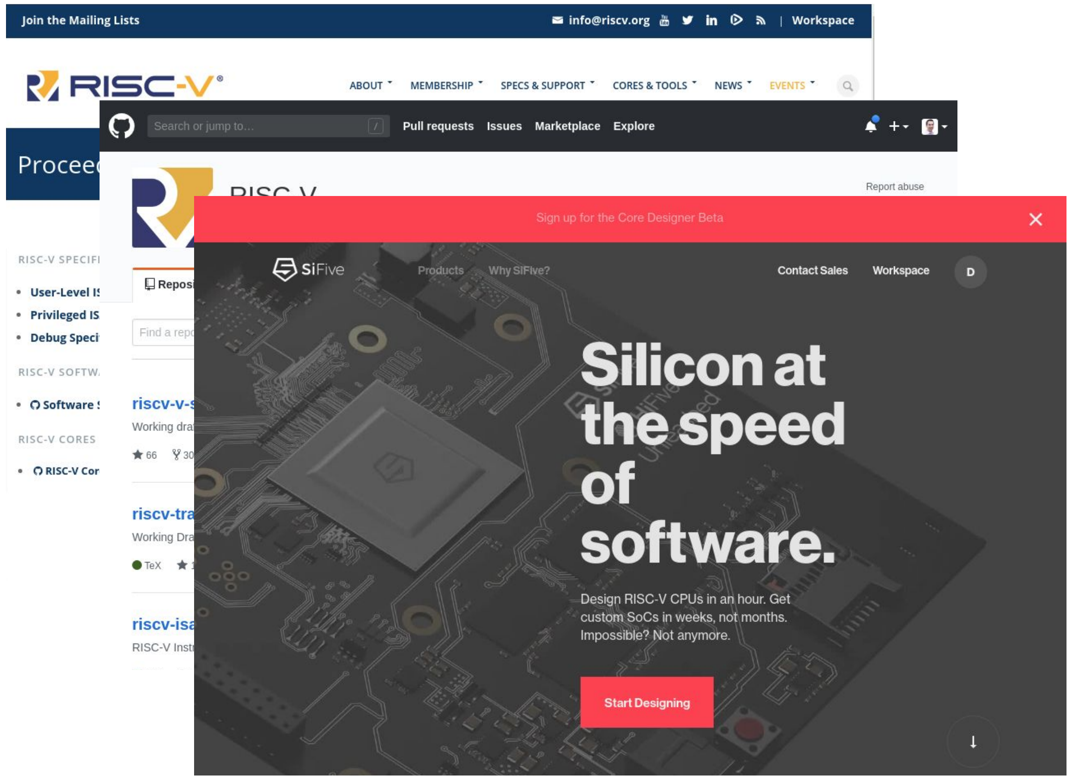


SiFive

More Information

Resources

- **https://riscv.org/**
  - RISC-V Specifications
  - Links to the RISC-V mailing lists
  - Workshop proceedings
- **GitHub**
  - https://github.com/sifive/
  - https://github.com/riscv
- [**https://www.sifive.com/**](https://www.sifive.com/)
  - RISC-V IP and Development Boards
  - RISC-V Tools
  - Forums


Join the Mailing Lists
info@riscv.org | Workspace
RISC-V
ABOUT MEMBERSHIP SPECS & SUPPORT CORES & TOOLS NEWS EVENTS
Search or jump to... Pull requests Issues Marketplace Explore
Report abuse
Sign up for the Core Designer Beta
SiFive Products Why SiFive? Contact Sales Workspace
Silicon at the speed of software.
Design RISC-V CPUs in an hour. Get custom SoCs in weeks, not months. Impossible? Not anymore.
Start Designing

SiFive

3-Part Webinar Series

May 7th

An Introduction to the RISC-V Architecture

May 15th

SiFive’s 2 Series Core IP

May 23rd

From a Custom 2 Series Core to Hello World in 30 Minutes

https://info.sifive.com/risc-v-second-webinar-series

SiFive

Questions?