RISC-V: Berkeley Hardware for Your Berkeley Software (Distribution)

Arun Thomas (BAE Systems)

BSDCan 2016

RISC-V Goal: Become the industry-standard ISA for **all** computing devices

“Our modest goal is world domination”

Our Goal: Make BSD the standard OS for RISC-V

RISC-V

BSD Daemon, Courtesy of Marshall Kirk McKusick

Talk Overview

- Goal: Get you hacking RISC-V
  - RISC-V 101
  - Hardware and Software Ecosystem
  - FreeBSD/RISC-V

RISC-V 101

RISC-V is an open instruction set **specification**.

You can build open source or proprietary implementations. Your **choice**.

No licensing, No royalties, No **lawyers**.

RISC-V

- **Modest Goal**: “Become the standard ISA for all computing devices”
  - Microcontrollers to supercomputers
- Designed for
  - Research
  - Education
  - Commercial use

RISC-V Foundation

bluespec DRAPER Google

Hewlett Packard Labs IBM Mellanox TECHNOLOGIES

Microsemi NVIDIA ORACLE

Rambus Cryptography Research SiFive Western Digital

antmicro EMBEDDED SYSTEMS BAE SYSTEMS codasip ESPRESSIF

GRAY RESEARCH INTRINSIX IDT LATTICE SEMICONDUCTOR

lowRISC ROA LOGIC Rumble Development runtime.io

Syntacore™ Custom cores and tools Technolution VectorBlox embedded supercomputing

Origin of RISC-V

David Patterson and Krste Asanović

Origin of RISC-V

- Dave and Krste began searching for a common research ISA in 2010
    - x86 and ARM: too complex, IP issues
    - Decided to develop their own ISA (Summer 2010)
- Released frozen User Spec (v2.0) in May 2014

RISC-V ISA

- **Fifth** RISC ISA from Berkeley, so RISC-**V**
- **Modular** ISA: Simple base instruction set plus extensions
  - 32-bit, 64-bit, and 128-bit ISAs
  - <50 hardware instructions in the base ISA
- Designed for extension/customization

RISC-V ISA Overview

- Base integer ISAs
  - RV32I, RV64I, RV128I, RV32E
- Standard extensions
  - M: Integer multiply/divide
  - A: Atomic memory operations
  - F: Single-precision floating point
  - D: Double-precision floating point
  - G: IMAFD, "General purpose" ISA

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base		+RV{64,128}	
Loads	Load Byte	I	`LB`	`rd,rs1,imm`		
	Load Halfword	I	`LH`	`rd,rs1,imm`		
	Load Word	I	`LW`	`rd,rs1,imm`	`L{D|Q}`	`rd,rs1,imm`
	Load Byte Unsigned	I	`LBU`	`rd,rs1,imm`		
	Load Half Unsigned	I	`LHU`	`rd,rs1,imm`	`L{W|D}U`	`rd,rs1,imm`
Stores	Store Byte	S	`SB`	`rs1,rs2,imm`		
	Store Halfword	S	`SH`	`rs1,rs2,imm`		
	Store Word	S	`SW`	`rs1,rs2,imm`	`S{D|Q}`	`rs1,rs2,imm`
Shifts	Shift Left	R	`SLL`	`rd,rs1,rs2`	`SLL{W|D}`	`rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI`	`rd,rs1,shamt`	`SLLI{W|D}`	`rd,rs1,shamt`
	Shift Right	R	`SRL`	`rd,rs1,rs2`	`SRL{W|D}`	`rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI`	`rd,rs1,shamt`	`SRLI{W|D}`	`rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA`	`rd,rs1,rs2`	`SRA{W|D}`	`rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI`	`rd,rs1,shamt`	`SRAI{W|D}`	`rd,rs1,shamt`
Arithmetic	ADD	R	`ADD`	`rd,rs1,rs2`	`ADD{W|D}`	`rd,rs1,rs2`
	ADD Immediate	I	`ADDI`	`rd,rs1,imm`	`ADDI{W|D}`	`rd,rs1,imm`
	SUBtract	R	`SUB`	`rd,rs1,rs2`	`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	UJ	`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

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

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	`C.LW`	`rd',rs1',imm`	`LW`	`rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP`	`rd,imm`	`LW`	`rd,sp,imm*4`
	Load Double	CL	`C.LD`	`rd',rs1',imm`	`LD`	`rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP`	`rd,imm`	`LD`	`rd,sp,imm*8`
	Load Quad	CL	`C.LQ`	`rd',rs1',imm`	`LQ`	`rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP`	`rd,imm`	`LQ`	`rd,sp,imm*16`
Stores	Store Word	CS	`C.SW`	`rs1',rs2',imm`	`SW`	`rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP`	`rs2,imm`	`SW`	`rs2,sp,imm*4`
	Store Double	CS	`C.SD`	`rs1',rs2',imm`	`SD`	`rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP`	`rs2,imm`	`SD`	`rs2,sp,imm*8`
	Store Quad	CS	`C.SQ`	`rs1',rs2',imm`	`SQ`	`rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP`	`rs2,imm`	`SQ`	`rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD`	`rd,rs1`	`ADD`	`rd,rd,rs1`
	ADD Word	CR	`C.ADDW`	`rd,rs1`	`ADDW`	`rd,rd,imm`
	ADD Immediate	CI	`C.ADDI`	`rd,imm`	`ADDI`	`rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW`	`rd,imm`	`ADDIW`	`rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP`	`x0,imm`	`ADDI`	`sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN`	`rd',imm`	`ADDI`	`rd',sp,imm*4`
	Load Immediate	CI	`C.LI`	`rd,imm`	`ADDI`	`rd,x0,imm`
	Load Upper Imm	CI	`C.LUI`	`rd,imm`	`LUI`	`rd,imm`
	MoVe	CR	`C.MV`	`rd,rs1`	`ADD`	`rd,rs1,x0`
	SUB	CR	`C.SUB`	`rd,rs1`	`SUB`	`rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI`	`rd,imm`	`SLLI`	`rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ`	`rs1',imm`	`BEQ`	`rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ`	`rs1',imm`	`BNE`	`rs1',x0,imm`
Jump	Jump	CJ	`C.J`	`imm`	`JAL`	`x0,imm`
	Jump Register	CR	`C.JR`	`rd,rs1`	`JALR`	`x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL`	`imm`	`JAL`	`ra,imm`
	Jump & Link Register	CR	`C.JALR`	`rs1`	`JALR`	`ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`		`EBREAK`	

32-bit Instruction Formats

	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

16-bit (RVC) Instruction Formats

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

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM				
Category	**Name**	**Fmt**	**RV32M (Multiply-Divide)**	**+RV{64,128}**
Multiply	MULtiply	R	`MUL rd,rs1,rs2`	`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 rd,rs1,rs2`	`DIV{W|D} rd,rs1,rs2`
	DIVide Unsigned	R	`DIVU rd,rs1,rs2`	
Remainder	REMainder	R	`REM rd,rs1,rs2`	`REM{W|D} rd,rs1,rs2`
	REMainder Unsigned	R	`REMU rd,rs1,rs2`	`REMU{W|D} rd,rs1,rs2`

Optional Atomic Instruction Extension: RVA				
Category	**Name**	**Fmt**	**RV32A (Atomic)**	**+RV{64,128}**
Load	Load Reserved	R	`LR.W rd,rs1`	`LR.{D|Q} rd,rs1`
Store	Store Conditional	R	`SC.W rd,rs1,rs2`	`SC.{D|Q} rd,rs1,rs2`
Swap	SWAP	R	`AMOSWAP.W rd,rs1,rs2`	`AMOSWAP.{D|Q} rd,rs1,rs2`
Add	ADD	R	`AMOADD.W rd,rs1,rs2`	`AMOADD.{D|Q} rd,rs1,rs2`
Logical	XOR	R	`AMOXOR.W rd,rs1,rs2`	`AMOXOR.{D|Q} rd,rs1,rs2`
	AND	R	`AMOAND.W rd,rs1,rs2`	`AMOAND.{D|Q} rd,rs1,rs2`
	OR	R	`AMOOR.W rd,rs1,rs2`	`AMOOR.{D|Q} rd,rs1,rs2`
Min/Max	MINimum	R	`AMOMIN.W rd,rs1,rs2`	`AMOMIN.{D|Q} rd,rs1,rs2`
	MAXimum	R	`AMOMAX.W rd,rs1,rs2`	`AMOMAX.{D|Q} rd,rs1,rs2`
	MINimum Unsigned	R	`AMOMINU.W rd,rs1,rs2`	`AMOMINU.{D|Q} rd,rs1,rs2`
	MAXimum Unsigned	R	`AMOMAXU.W rd,rs1,rs2`	`AMOMAXU.{D|Q} rd,rs1,rs2`

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ				
Category	**Name**	**Fmt**	**RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)**	**+RV{64,128}**
Move	Move from Integer	R	`FMV.{H|S}.X rd,rs1`	`FMV.{D|Q}.X rd,rs1`
	Move to Integer	R	`FMV.X.{H|S} rd,rs1`	`FMV.X.{D|Q} rd,rs1`
Convert	Convert from Int	R	`FCVT.{H|S|D|Q}.W rd,rs1`	`FCVT.{H|S|D|Q}.{L|T} rd,rs1`
	Convert from Int Unsigned	R	`FCVT.{H|S|D|Q}.WU rd,rs1`	`FCVT.{H|S|D|Q}.{L|T}U rd,rs1`
	Convert to Int	R	`FCVT.W.{H|S|D|Q} rd,rs1`	`FCVT.{L|T}.{H|S|D|Q} rd,rs1`
	Convert to Int Unsigned	R	`FCVT.WU.{H|S|D|Q} rd,rs1`	`FCVT.{L|T}U.{H|S|D|Q} rd,rs1`
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	`FNMSUB.{S|D|Q} rd,rs1,rs2,rs3`	
	Negative Multiply-ADD	R	`FNMADD.{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`	
Categorization	Classify Type	R	`FCLASS.{S|D|Q} rd,rs1`	
Configuration	Read Status	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`	

RISC-V Calling Convention			
Register	ABI Name	Saver	Description
`x0`	`zero`	---	Hard-wired zero
`x1`	`ra`	Caller	Return address
`x2`	`sp`	Callee	Stack pointer
`x3`	`gp`	---	Global pointer
`x4`	`tp`	---	Thread pointer
`x5-7`	`t0-2`	Caller	Temporaries
`x8`	`s0/fp`	Callee	Saved register/frame pointer
`x9`	`s1`	Callee	Saved register
`x10-11`	`a0-1`	Caller	Function arguments/return values
`x12-17`	`a2-7`	Caller	Function arguments
`x18-27`	`s2-11`	Callee	Saved registers
`x28-31`	`t3-t6`	Caller	Temporaries
`f0-7`	`ft0-7`	Caller	FP temporaries
`f8-9`	`fs0-1`	Callee	FP saved registers
`f10-11`	`fa0-1`	Caller	FP arguments/return values
`f12-17`	`fa2-7`	Caller	FP arguments
`f18-27`	`fs2-11`	Callee	FP saved registers
`f28-31`	`ft8-11`	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

RISC Background

- Reduced Instruction Set Computer (RISC)
  - Smaller, less complex instruction sets
  - Load/store architecture
  - Easy to implement and efficient
- Berkeley RISC-I/II (Patterson) heavily influenced SPARC
- Stanford RISC (Hennessy) became MIPS
- ARM is the "Advanced RISC Machine”

RV64I registers

- 32 64-bit general-purpose registers (***x0***-***x31***)
    - ***x0*** is zero (***zero***) register
    - ***x1*** is return address (***ra***) register
    - ***x2*** is stack pointer (***sp***) register
    - ***x8*** is frame pointer (***fp***) register
- Program Counter (***pc***)

RISC-V Assembly

- Looks a lot like MIPS
- Many assembly macros and aliases
- MIPS resources are helpful

RISC-V: Data Operations

```
/* x1 = 1 */
li x1, 1
/* x2 = 2 */
li x2, 2
/* x3 = x1 + x2 */
add x3, x1, x2
```

RISC-V: Memory Operations

```
/* x0 = *x1 */
ld x0, (x1)
/* *x1 = x0 */
sd x0, (x1)
```

RISC-V: Control Flow

```
/* branch if x1 == x2 */
beq x1, x2, loop
/* call */
call func /* jal func */
/* return */
ret /* jr ra */
```

RISC-V Privilege Levels

- Level 0 - User (**U-mode**) - Applications
- Level 1 - Supervisor (**S-mode**) - BSD
- Level 2 - Hypervisor (**H-mode**) - Xen/bhyve
- Level 3 - Machine (**M-mode**) - Firmware
  - Only **required** level

Higher Privilege

Control and Status Registers (CSRs)

- Used for **low-level** programming
- Different registers for kernel (**S-mode**), hypervisor (**H-mode**), firmware (**M-mode**)
    - e.g., `sstatus`, `hstatus`, `mstatus`
- Used to configure:
    - System properties
    - Memory Management Unit (MMU)
    - Interrupts

Status Registers

- Machine-level Status Register (***mstatus***)
  - Current privilege mode
  - MMU mode
  - Interrupt enable
  - Past mode and interrupt status
- Supervisor-level Status Register (***sstatus***)
  - Restricted view of (***mstatus***) for S-mode

CSRing Status

```
/* Read sstatus */
csrr x1, sstatus
/* Write sstatus */
csrw sstatus, x1
```

Exception Types

- Synchronous Exceptions
  - Environment call (***ecall***) (formerly ***scall***)
  - Memory faults
  - Illegal instructions
  - Breakpoints
- Interrupts
  - Timer
  - Software
  - Devices

Exception Registers

- ***sepc*** - Supervisor exception program counter
  - Virtual address of instruction that encountered exception
- ***scause*** - Supervisor trap cause
  - Cause for exception
- ***sbadaddr*** - Supervisor Bad Address
  - Faulting address for memory faults

RISC-V Memory Modes (1/2)

- Set by "Virtualization Management" (***VM***) field in ***mstatus***
  - Determines virtual memory translation and protection scheme
- Supports simple schemes for microcontrollers
- ***Mbare*** - No translation or protection
  - For CPUs that only support **M-mode**
  - VM mode on reset
- ***Mbb*** and ***Mbbid*** - Base and bounds protection
  - For CPUs that support **U-mode**

RISC-V Memory Modes (2/2)

- Page-based schemes for CPUs that support **S-mode**
  - Hardware-managed TLBs - MMU does page table walk on TLB miss
  - Up to four levels of page tables
  - Various page size: 4 KB, 2 MB, 4 MB, 1 GB, 512 GB
  - ***sptbr*** - supervisor page table base register
- ***Sv32*** - 32-bit virtual addressing for RV32
- ***Sv39*** - 39-bit virtual addressing for RV64
- ***Sv48***- 48-bit virtual addressing for RV64

See RISC-V specs for more details

RISC-V Specs

- User-Level ISA Specification v2.1 (Jun 2016)
- Privileged ISA Specification v1.7 (May 2015)
    - v1.9 will be released soon
- Compressed ISA Specification v1.9 (Nov 2015)

RISC-V Hardware and Software Ecosystem

Development Platforms

- Software Emulation
  - Spike RISC-V ISA simulator (riscv-isa-sim)
  - QEMU/RISC-V
  - Angel JavaScript emulator
- FPGA emulation
  - Pick your poison (Xilinx, Altera, MicroSemi, Lattice)
  - Xilinx ZedBoard is a popular platform

RISC-V SoCs/Cores (1/3)

- Berkeley https://github.com/ucb-bar
  - Rocket - 5 stage pipeline, single-issue
  - BOOM - Out-of-order core
  - Zscale - Microcontroller core
  - Sodor - Educational cores (1-5 stage)
- LowRISC (Cambridge) https://github.com/lowrisc
  - "Raspberry Pi for grownups”
  - Tagged architecture and Minion cores

RISC-V SoCs/Cores (2/3)

- SHAKTI (IIT-Madras) https://bitbucket.org/casl/shakti_public
    - RISC-V is the "standard ISA" for India
    - IIT-Madras building 6 open-source cores, from microcontrollers to supercomputers
- YARVI https://github.com/tommythorn/yarvi
    - Used in Cambridge's computer architecture course

RISC-V SoCs/Cores (3/3)

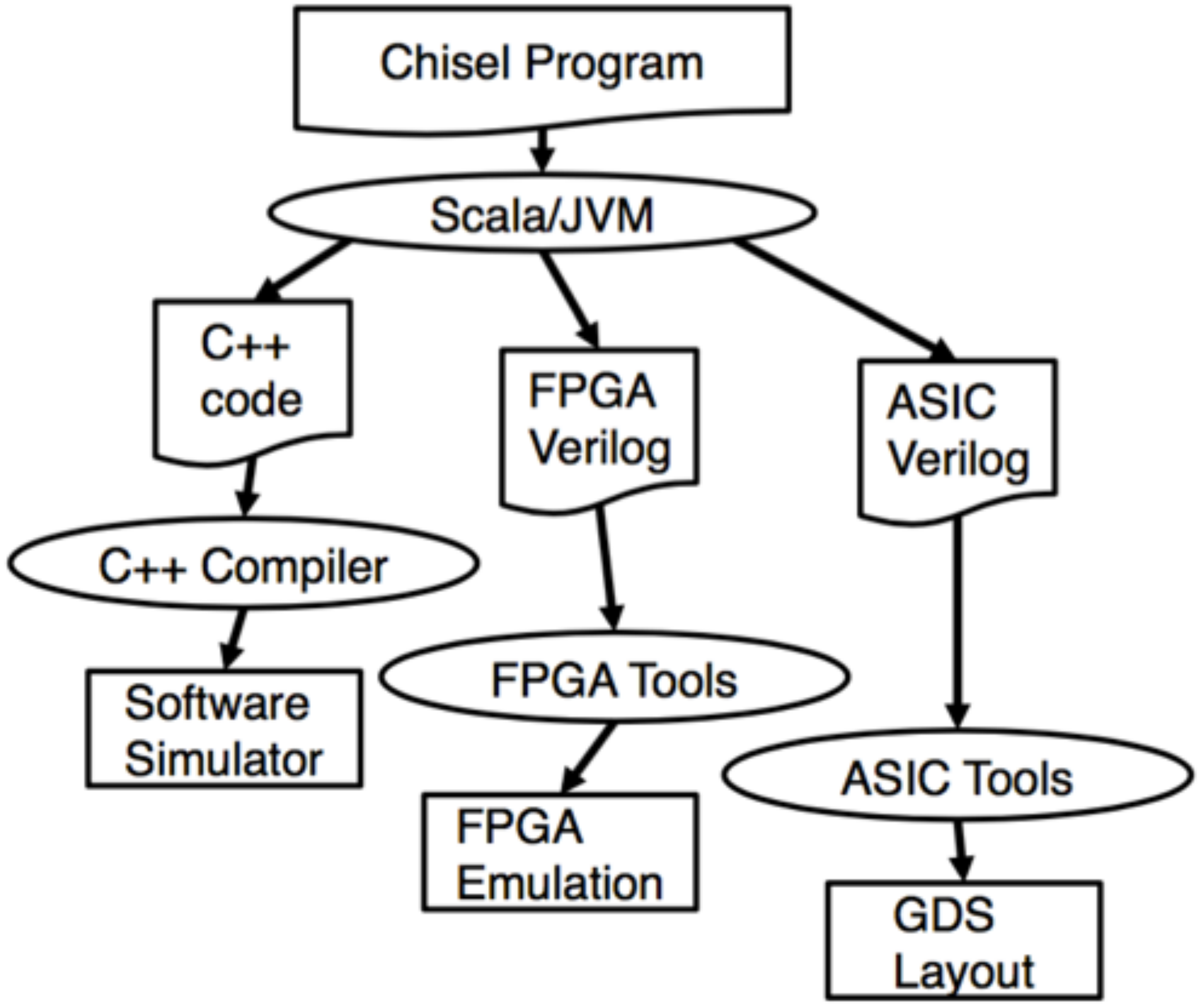
- PULPino (ETH Zurich) https://github.com/pulp-platform/pulpino
- PicoRV32 https://github.com/cliffordwolf/picorv32
- ORCA https://github.com/VectorBlox/orca
- BlueSpec, Inc has RISC-V Factory
- Many, many more commercial and open source RISC-V cores

Rocket Chip SoC Generator

- Parameterized RISC-V SoC Generator written in Chisel HDL

- Can use this as the basis for your own SoC

- Can target C++ software simulator, FPGA emulation, or ASIC tools



Making RISC-V Yours

- Modify the tunable parameters of an existing core
- Implement an accelerator using the Rocket Custom Coprocessor (RoCC) interface
- Implement your own RISC-V instruction set extension (Ch. 9, User Spec)
- Implement your own RISC-V core

Current Software Landscape

- Several OS ports in progress
  - Proxy kernel, Linux (Yocto/Poky, Gentoo, Debian), FreeBSD, NetBSD, seL4, Genode
- Support for primary open source toolchains
  - Binutils, GCC, clang/LLVM
- Multiple software simulators/emulators
  - Spike, QEMU, Angel

Common Software Options

- **Newlib + Proxy Kernel (pk)**
  - Single user application only
  - Proxies system calls to host system
- **Glibc + Linux Kernel**
  - Distributions: Busybox, **Yocto/Poky**, Gentoo
- **FreeBSD**
  - RISC-V support will appear in FreeBSD 11

NetBSD/RISC-V

- Matt Thomas has been working on the port
- Core kernel support has been merged
- Waiting on pmap changes to be merged and updated RISC-V toolchain

FreeBSD/RISC-V

FreeBSD/RISC-V Port

- Courtesy of Ruslan Bukin
- Merged to -CURRENT, Will be in FreeBSD 11.0
- Based on ARMv8 port
- Targets RV64G and Sv39
- Using GCC as toolchain (clang's not ready)

FreeBSD/RISC-V Code

- Key source directories:
  - *sys/riscv/include*
  - *sys/riscv/riscv*
- Key configuration files:
  - *sys/conf/files.riscv*
  - *sys/riscv/conf/DEFAULTS*
  - *sys/riscv/conf/GENERIC*

Typical Boot Process

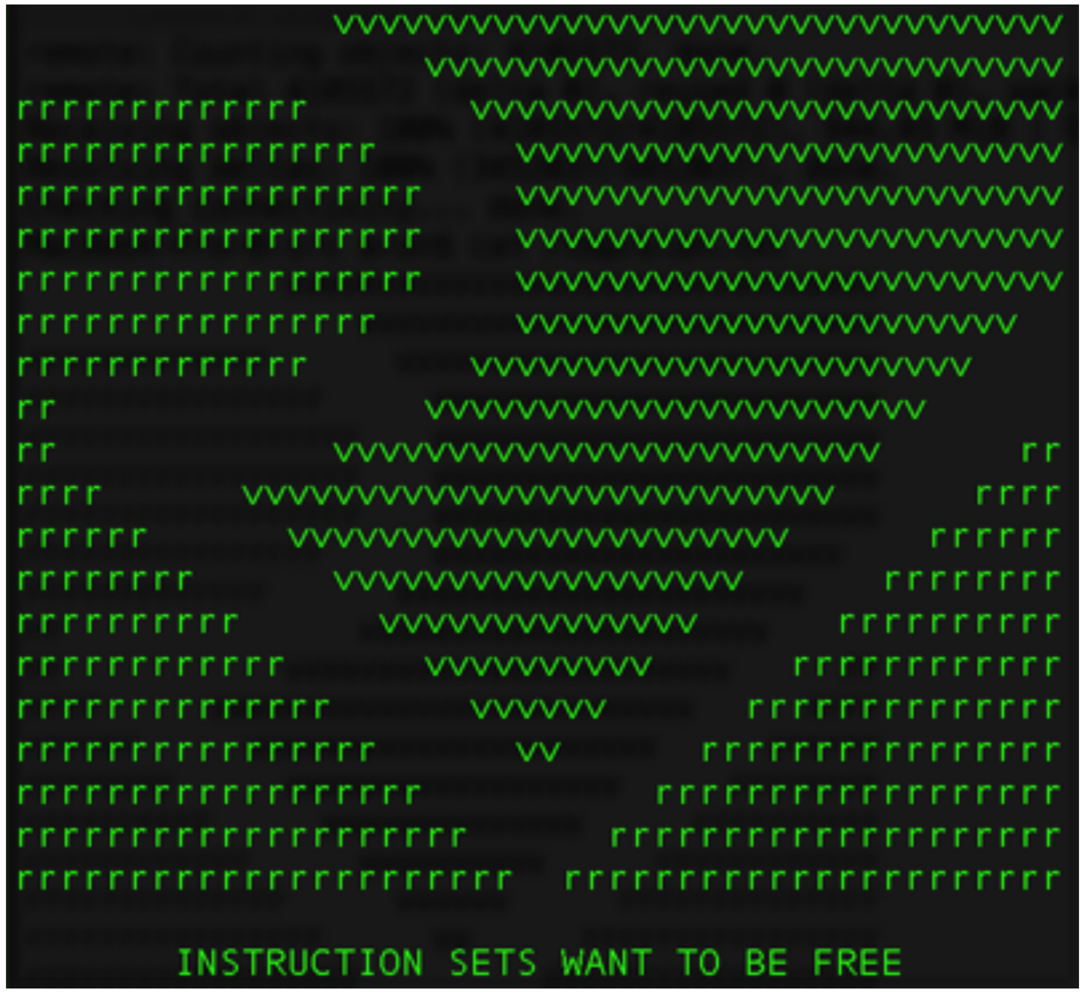
1. Firmware
2. Bootloader
3. Kernel

Booting Up

- Firmware/bootloader responsibilities:
  - Hardware initialization (e.g., DRAM, serial)
  - Passing boot parameters to kernel
  - Loading the kernel
- Typical options:
  - SoC ROM + U-Boot + loader(8)
  - UEFI + loader(8)

Booting up on RISC-V

- Berkeley Boot Loader (BBL) is firmware/loader



```
              vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                  vvvvvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrr       vvvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrr      vvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrrr    vvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrrr    vvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrrr    vvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrr      vvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrr       vvvvvvvvvvvvvvvvvvvvvv
rr                vvvvvvvvvvvvvvvvvvvvvv
rr            vvvvvvvvvvvvvvvvvvvvvvvv      rr
rrrr      vvvvvvvvvvvvvvvvvvvvvvvvvv      rrrr
rrrrrr      vvvvvvvvvvvvvvvvvvvvvv      rrrrrr
rrrrrrrr      vvvvvvvvvvvvvvvvvv      rrrrrrrr
rrrrrrrrrr      vvvvvvvvvvvvvv      rrrrrrrrrr
rrrrrrrrrrrr      vvvvvvvvvv      rrrrrrrrrrrr
rrrrrrrrrrrrrr      vvvvvv      rrrrrrrrrrrrrr
rrrrrrrrrrrrrrrr      vv      rrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrr          rrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrr      rrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrrr  rrrrrrrrrrrrrrrrrrrrrr

       INSTRUCTION SETS WANT TO BE FREE
```

Booting FreeBSD

- Not using BBL currently
  - Kernel reimplements some BBL functionality
  - For ease of development
- Long term: Follow forthcoming RISC-V boot spec
- Using DeviceTree currently; Priv Spec 1.9 specifies a simpler structure

Device Tree on RISC-V

- Used by Linux and FreeBSD on several architectures
- Data structure that describes hardware configuration
- In *sys/boot/fdt/dts/riscv/spike.dts*:

```
timer0: timer@0 {
    compatible = "riscv,timer";
    interrupts = < 1 >;
    interrupt-parent = < &pic0 >;
    clock-frequency = < 1000000 >;
};
```

Kernel Initialization (1/2)

- Early kernel initialization
  - Set initial page table and enable MMU
  - Set up exception vector table and handlers
- Initialize Devices
  - Serial
  - Timers (e.g., for clock tick)

Kernel Initialization (2/2)

- Machine-independent initialization
  - Initialize kernel subsystems
  - More device initialization
- Enable interrupts
  - Switch to User mode and run init

FreeBSD Kernel Startup: First Steps

- FreeBSD kernel's first instructions, sys/riscv/riscv/locore.S
    - Set up stack and initial page table
    - Switch to Supervisor mode and enable MMU

```
_start:
    ...
    /* Set page tables base register */
    la   s1, pagetable_l1
    csrw sptbr, s1
    ...
    /* Exit from machine mode */
    ...
    csrw mepc, t0
    eret
```

Exception Vector Table

- In *sys/riscv/riscv/locore.S*

```
	/* Trap entries */
mentry:
	/* User mode entry point (mtvec + 0x000) */
	j	user_trap
	/* Supervisor mode entry point (mtvec + 0x040) */
	j	supervisor_trap
	/* Hypervisor mode entry point (mtvec + 0x080) */
	j	bad_trap
	/* Machine mode entry point (mtvec + 0x0C0) */
	j	bad_trap
	/* Reset vector */
_start:
```

FreeBSD Kernel Startup: initriscv()

- ***start*** continues execution:
    - Sets up environment for C code
    - Calls first C function ***initriscv()***
- ***initriscv()*** continues RISC-V-specific init
- See *sys/riscv/riscv/machdep.c*
    - Maps devices and initializes console
    - Sets up real page table and switches to it

FreeBSD Kernel Startup: mi_startup()

- Finally, start calls ***mi_startup()***
- ***mi_startup()*** is first machine-independent code

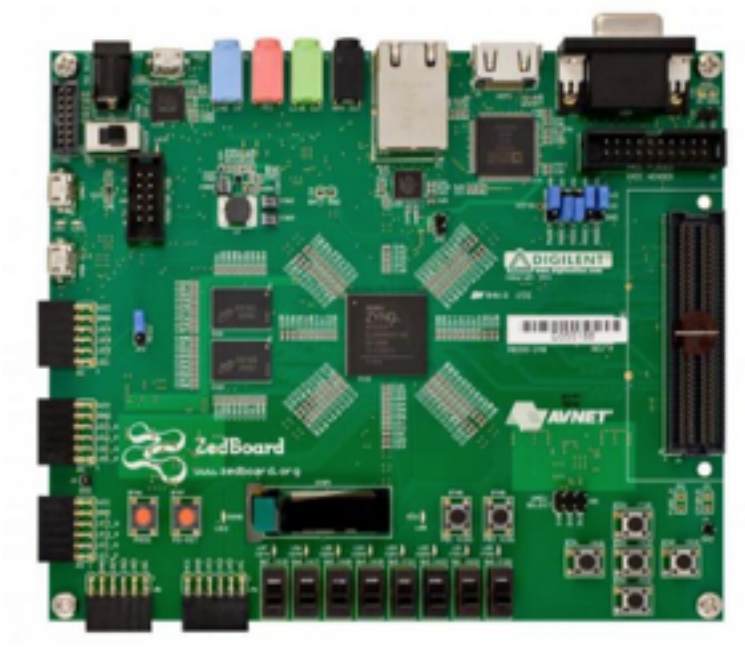
```
Copyright (c) 1992-2016 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
   The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
FreeBSD 11.0-CURRENT #0 95ccd77(HEAD)-dirty: Sat Jun  4 03:39:55 UTC 2016
```

Handling Exceptions

- Save context (***save_registers***) in *sys/riscv/exception.S*
- Call ***do_trap_user*** or ***do_trap_supervisor*** in *sys/riscv/riscv/trap.c*
  - Read the cause register (***scause***)
  - Jump to appropriate handler function (e.g., ***data_abort***)
- Restore context and return to previous mode (***load_registers***, ***eret***) in *sys/riscv/exception.S*

Developing FreeBSD/RISC-V

- See https://wiki.freebsd.org/riscv for full instructions
  - *make TARGET_ARCH=riscv64 buildworld*
  - *make TARGET_ARCH=riscv64 KERNCONF=SPIKE buildkernel*
  - *spike -m1024 -p2 +disk=root.img kernel*
- Target Platforms
  - Spike RISC-V ISA simulator
  - QEMU/RISC-V
  - Rocket on Xilinx ZedBoard



FreeBSD/RISC-V TODO

- Package up RISC-V simulators and toolchain
- clang/LLVM RISC-V backend work
- Update to new privileged ISA
- FreeBSD ports support
- QEMU user

RISC-V Resources

- RISC-V specs: http://riscv.org/specifications
- RISC-V Workshop Proceedings: http://riscv.org/category/workshops/proceedings
- HPCA Tutorial: http://riscv.org/2015/02/risc-v-tutorial-hpca-2015
- Mailing Lists: http://riscv.org/mailing-lists
- Stack Overflow: http://stackoverflow.com/questions/tagged/riscv
- "The Case for Open Instruction Sets", *Microprocessor Report*
- "RISC-V Offers Simple, Modular ISA", *Microprocessor Report*

FreeBSD/RISC-V Resources

- Wiki Page: wiki.freebsd.org/riscv
- IRC: #freebsd-riscv on EFnet
- Mailing List: freebsd-riscv@freebsd.org
- Ruslan's RISC-V Workshop talk (slides, video)
- FreeBSD/RISC-V in Action (video)

Fourth RISC-V Workshop

- **Save the date**: July 12-13, 2016 at MIT CSAIL / Stata Center in Cambridge, MA

Massachusetts Institute of Technology

RISC-V

- A new open instruction set specification
- An excellent platform for computer architecture research and education
- A solid foundation for open hardware efforts and commercial products
- Runs FreeBSD now. **You should try it.**

Questions?

- Contact: arun.thomas@acm.org
- See you at the 4th RISC-V workshop (July 12-13)!