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
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 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
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
### Base Integer Instructions: RV32I, RV64I, and RV128I

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>+RV64I(128I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>Load Byte</td>
<td>I</td>
<td>rd, rl, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Halfword</td>
<td>I</td>
<td>rl, rl, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Word</td>
<td>I</td>
<td>rl, rl, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Byte Unsigned</td>
<td>I</td>
<td>rl, rl, imm</td>
<td></td>
</tr>
<tr>
<td>Stores</td>
<td>Store Byte</td>
<td>S</td>
<td>r Snapdragon</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store Halfword</td>
<td>S</td>
<td>rl, rl, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store Word</td>
<td>S</td>
<td>rl, rl, imm</td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>Shift Left Immediate</td>
<td>SLL</td>
<td>rl, rl, rs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Immediate</td>
<td>SRL</td>
<td>rl, rl, rs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>SRA</td>
<td>rl, rl, rs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic Immediate</td>
<td>SRAI</td>
<td>rl, rl, rs, shamt</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD Immediate</td>
<td>ADDI</td>
<td>rd, rl, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD Subtract</td>
<td>SUB</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Upper Imm</td>
<td>U</td>
<td>rd, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add Upper Imm to PC</td>
<td>U</td>
<td>rd, imm</td>
<td></td>
</tr>
</tbody>
</table>

### RV Privileged Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>+RV64I(128I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR Access</td>
<td>Atomic R/W</td>
<td>CSRRW</td>
<td>rd, csr, rl</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Set Bit</td>
<td>CSRRS</td>
<td>rd, csr, rl</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Clear Bit</td>
<td>CSRRC</td>
<td>rd, csr, rl</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic R/W Imm</td>
<td>CSRRW</td>
<td>rd, csr, imm</td>
<td></td>
</tr>
<tr>
<td>Change Level</td>
<td>Env. Call</td>
<td>CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Environment Breakpoint</td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trap Redirect to Supervisor</td>
<td>MTBE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Redirect Trap to Hypervisor</td>
<td>MTRE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt Wait for Interrupt</td>
<td>MW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supervisor FENCE</td>
<td>SFENCE, vn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optional Compressed (16-bit) Instruction Extension: RVC

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RVI equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>Load Word</td>
<td>CL</td>
<td>LW rd', rs1', imm*4</td>
</tr>
<tr>
<td></td>
<td>Load Word SP</td>
<td>CI</td>
<td>LWP rd, imm</td>
</tr>
<tr>
<td></td>
<td>Load Double</td>
<td>CL</td>
<td>LD rd', rs1', imm*8</td>
</tr>
<tr>
<td></td>
<td>Load Double SP</td>
<td>CI</td>
<td>LDSP rd, imm*8</td>
</tr>
<tr>
<td></td>
<td>Load Quad</td>
<td>CL</td>
<td>LQ rd', rs1', imm*16</td>
</tr>
<tr>
<td>Stores</td>
<td>Store Word SP</td>
<td>CSS</td>
<td>SWP rs2, imm</td>
</tr>
<tr>
<td></td>
<td>Store Double</td>
<td>CS</td>
<td>SD rs1', rs2', imm*8</td>
</tr>
<tr>
<td></td>
<td>Store Double SP</td>
<td>CSSP</td>
<td>SDSP rs2, imm*8</td>
</tr>
<tr>
<td></td>
<td>Store Quad</td>
<td>CS</td>
<td>SQ rs1', rs2', imm*16</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD</td>
<td>CR</td>
<td>ADD rd, rs1, rs2</td>
</tr>
<tr>
<td></td>
<td>ADD Word</td>
<td>CR</td>
<td>ADDW rd, rd, imm</td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td>CI</td>
<td>ADDI rd, rd, imm</td>
</tr>
<tr>
<td></td>
<td>ADD Word Imm</td>
<td>CI</td>
<td>ADDIW rd, rd, imm</td>
</tr>
<tr>
<td></td>
<td>ADDP Imm* 16</td>
<td>CI</td>
<td>ADDP imm* 16</td>
</tr>
<tr>
<td></td>
<td>ADDP Imm*</td>
<td>CI</td>
<td>ADDP imm*</td>
</tr>
<tr>
<td></td>
<td>Load Immediate</td>
<td>CL</td>
<td>LI rd, imm</td>
</tr>
<tr>
<td></td>
<td>Load Upper Imm</td>
<td>CI</td>
<td>LUI rd, imm</td>
</tr>
<tr>
<td>Synch</td>
<td>Synch Thread</td>
<td>I</td>
<td>FENCE</td>
</tr>
<tr>
<td></td>
<td>Synch Inst &amp; Data</td>
<td>I</td>
<td>FENCE.i</td>
</tr>
<tr>
<td>System</td>
<td>System CALL</td>
<td>CR</td>
<td>Syscall</td>
</tr>
<tr>
<td></td>
<td>System BREAK</td>
<td>CR</td>
<td>Break</td>
</tr>
<tr>
<td>Counters</td>
<td>Read CYCLE</td>
<td>I</td>
<td>RCYCle rd</td>
</tr>
<tr>
<td></td>
<td>Read CYCLE upper Half</td>
<td>I</td>
<td>RCYCle rd</td>
</tr>
<tr>
<td></td>
<td>Read TIME</td>
<td>I</td>
<td>RTIME rd</td>
</tr>
<tr>
<td></td>
<td>Read TIME upper Half</td>
<td>I</td>
<td>RTIME upper Half</td>
</tr>
<tr>
<td></td>
<td>Read INSTR Retired</td>
<td>I</td>
<td>RINSTR rd</td>
</tr>
<tr>
<td></td>
<td>Read INSTR upper Half</td>
<td>I</td>
<td>RINSTR rd</td>
</tr>
</tbody>
</table>

### 32-bit Instruction Formats

### 16-bit (RVC) Instruction Formats

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 (s0=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.
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.

### RISC-V Calling Convention and Five Optional Extensions:
- 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 11 for RVA, and 6 each for RVF/D/Q.

#### Optional Multiply-Divide Instruction Extension: RVM

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32M (Multiply-Divide)</th>
<th>+RV(64,128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>MUL</td>
<td>rd,rs1,rs2</td>
<td>MUL(W</td>
<td>D)</td>
</tr>
<tr>
<td>Multiply upper Half</td>
<td>MULH</td>
<td>rd,rs1,rs2</td>
<td>MULH</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Multiply Half Sign/Uns</td>
<td>MULHSU</td>
<td>rd,rs1,rs2</td>
<td>MULHSU</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Multiply upper Half Uns</td>
<td>MULHU</td>
<td>rd,rs1,rs2</td>
<td>MULHU</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Divide</td>
<td>DIV</td>
<td>rd,rs1,rs2</td>
<td>DIV(W</td>
<td>D)</td>
</tr>
<tr>
<td>Divide Uns</td>
<td>DIVU</td>
<td>rd,rs1,rs2</td>
<td>DIVU</td>
<td>rd,rs1,rs2</td>
</tr>
</tbody>
</table>

#### Optional Atomic Instruction Extension: RVA

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32A (Atomic)</th>
<th>+RV(64,128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Load Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>Store Condition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td>SWAP</td>
<td>rd,rs1,rs2</td>
<td>AMOSWAP.(D</td>
<td>Q)</td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>rd,rs1,rs2</td>
<td>AMOADD.(D</td>
<td>Q)</td>
</tr>
<tr>
<td>Logical</td>
<td>XOR</td>
<td>rd,rs1,rs2</td>
<td>AMOXOR.(D</td>
<td>Q)</td>
</tr>
<tr>
<td>OR</td>
<td>AND</td>
<td>rd,rs1,rs2</td>
<td>AMOAND.(D</td>
<td>Q)</td>
</tr>
<tr>
<td>Min/Max</td>
<td>MINimum</td>
<td>rd,rs1,rs2</td>
<td>AMOINR.(D</td>
<td>Q)</td>
</tr>
<tr>
<td></td>
<td>MAXimum</td>
<td>rd,rs1,rs2</td>
<td>AMOMAX.(D</td>
<td>Q)</td>
</tr>
<tr>
<td>Min/Max Unsign</td>
<td>MINimum</td>
<td>rd,rs1,rs2</td>
<td>AMOMIN.(D</td>
<td>Q)</td>
</tr>
<tr>
<td></td>
<td>MAXimum Unsign</td>
<td>rd,rs1,rs2</td>
<td>AMOMAX.(D</td>
<td>Q)</td>
</tr>
</tbody>
</table>

#### Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

| Category     | Name                        | Fmt   | RV32(I|D|Q) | +RV(64,128) |
|--------------|-----------------------------|-------|----------|-------------|
| Move         | Move from Integer           | rd,s1 | FWM.(B|S)X | rd,s1      |
|             | Move to Integer             |       | FWM.(X|B) | rd,s1      |
| Convert      | Convert from Int            | rd,s1 | FCVT.(B|S)D(Q).W | rd,s1 |
|             | Convert from Int Uns        | rd,s1 | FCVT.(B|S)D(Q).W | rd,s1 |
|             | Convert to Int              | rd,s1 | FCVT.(B|S)D(Q).W | rd,s1 |
|             | Convert to Int Uns          | rd,s1 | FCVT.(B|S)D(Q).W | rd,s1 |

#### Optional Instruction Extensions: RVF, RVD, RVQ

| Category     | Name                        | Fmt   | RV32(I|D|Q) | +RV(64,128) |
|--------------|-----------------------------|-------|----------|-------------|
| Arithmetic   | ADD                         | rd,s1 | FADD.(B|S)D | rd,s1      |
|             | SUBtract                    | rd,s1 | FSUB.(B|S)D | rd,s1      |
|             | Multiply                    | rd,s1 | FMUL.(B|S)D | rd,s1      |
|             | Divide                      | rd,s1 | FSDIV.(B|S)D | rd,s1      |
|             | Square Root                 | rd,s1 | FSQRT.(B|S)D | rd,s1      |
| Mut-Add      | Multiply-Multiply           | rd,s1,rs2,rs2,rs3,rs2 | FMULX.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Multiply-Subtract           | rd,s1,rs2,rs2,rs3,rs2 | FMINUS.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Negative Multiply-Subtract  | rd,s1,rs2,rs2,rs3,rs2 | FMINUS.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Negative Multiply-Add       | rd,s1,rs2,rs2,rs3,rs2 | FMINUS.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
| Sign Inject  | SInjection Source           | rd,s1,rs2,rs2,rs3,rs2 | FSIGN.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Negative Sign Injection     | rd,s1,rs2,rs2,rs3,rs2 | FSIGN.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | xor Sign Injection          | rd,s1,rs2,rs2,rs3,rs2 | FXOR.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
| Min/Max      | Minimum                    | rd,s1,rs2,rs2,rs3,rs2 | FMAX.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Maximum                    | rd,s1,rs2,rs2,rs3,rs2 | FMAX.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
| Compare      | Compare Float <            | rd,s1,rs2,rs2,rs3,rs2 | FFSX.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
|             | Compare Float >            | rd,s1,rs2,rs2,rs3,rs2 | FFSX.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |
| Categorization | Classify Type               | rd,s1,rs2,rs2,rs3,rs2 | FCLASS.(B|S)D | rd,s1,rs2,rs2,rs3,rs2 |

### RISC-V Calling Convention

**Register ABI Name**
- **R**: callee
- **S**: caller
- **I**: caller
- **F**: callee
- **L**: caller

**Description**
- **Hard-wired zero**: Hard-wired zero
- **Return address**: Return address
- **Stack pointer**: Stack pointer
- **Function arguments/return values**: Function arguments/return values
- **Saved registers**: Saved registers
- **Temporaries**: Temporaries
- **Caller**: Caller
- **FP saved registers**: FP saved registers
- **FP arguments/return values**: FP arguments/return values
- **FP arguments**: FP arguments
- **FP temporaries**: FP temporaries
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](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](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
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`:

```c
(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

```c
_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`

```c
/* 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:
```

56
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 \texttt{mi\_startup()}

• \texttt{mi\_startup()} is first machine-independent code
Handling Exceptions

- Save context (\textit{save\_registers}) in \texttt{sys/riscv/exception.S}

- Call \texttt{do\_trap\_user} or \texttt{do\_trap\_supervisor} in \texttt{sys/riscv/riscv/trap.c}

  - Read the cause register (\textit{scause})

  - Jump to appropriate handler function (e.g., \textit{data\_abort})

- Restore context and return to previous mode (\texttt{load\_registers}, \texttt{eret}) in \texttt{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
- 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
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)!