Xvisor: Embedded Hypervisor for RISC-V

Anup Patel <anup.patel@wdc.com>

Open Source Summit Europe 2019
Roadmap

• RISC-V H-Extension
• Xvisor Overview
• Xvisor RISC-V
• Status & Future Work
• Xvisor RISC-V Demo
• Questions
RISC-V H-Extension

The RISC-V Hypervisor Extension
What is RISC-V?

Free and Open Instruction Set Architecture (ISA)

- **Clean-slate and Extensible ISA**
- **XLEN (machine word length)** can be 32 (RV32), 64 (RV64), and 128 (RV128)
- **32 general purpose registers**
- **Variable instruction length** (instruction compression)
- **Three privilege modes**: Machine (M-mode), Supervisor (S-mode), and User (U-mode)
- **Control and Status Registers (CSR)** for each privilege mode

<table>
<thead>
<tr>
<th>S-mode CSRs (Used By Linux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstatus</td>
</tr>
<tr>
<td>sie</td>
</tr>
<tr>
<td>sip</td>
</tr>
<tr>
<td>stvec</td>
</tr>
<tr>
<td>sepc</td>
</tr>
<tr>
<td>scause</td>
</tr>
<tr>
<td>stval</td>
</tr>
<tr>
<td>satp</td>
</tr>
<tr>
<td>sscratch</td>
</tr>
</tbody>
</table>

NOTE: *sedeleg*, *sideleg*, and *scounteren* not used currently

**General Purpose Registers**

- **zero** Hardwired-zero register
- **ra** Return address register
- **sp** Stack pointer register
- **gp** Global pointer register
- **tp** Thread pointer register
- **a0-a7** Function argument registers
- **t0-t6** Caller saved registers
- **s0-s11** Callee saved registers
RISC-V H-Extension: Spec Status

H-Extension spec close to freeze state

• Designed to suit both Type-1 (Baremetal) and Type-2 (Hosted) hypervisor
• v0.4-draft was released on 16\textsuperscript{th} June 2019
• v0.5-draft will be released soon
• WDC’s initial QEMU, Xvisor and KVM ports were based on v0.3
• They have all been updated to the new v0.4 spec
  – There were limited software changes required between v0.3 and v0.4
    • QEMU required more changes
RISC-V H-Extension: Privilege Mode Changes

New execution modes for guest execution

• HS-mode = S-mode with hypervisor capabilities and new CSRs

• Two additional modes:
  – VS-mode = Virtualized S-mode
  – VU-mode = Virtualized U-mode
RISC-V H-Extension: CSR changes

More control registers for virtualising S-mode

• In HS-mode (V=0)
  – “s<xyz>” CSRs point to standard “s<xyz>” CSRs
  – “h<xyz>” CSRs for hypervisor capabilities
  – “vs<xyz>” CSRs contains VS-mode state

• In VS-mode (V=1)
  – “s<xyz>” CSRs point to virtual “vs<xyz>” CSRs

<table>
<thead>
<tr>
<th>HS-mode CSRs for hypervisor capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>hstatus</td>
</tr>
<tr>
<td>hideleg</td>
</tr>
<tr>
<td>hedeleg</td>
</tr>
<tr>
<td>htimedelta</td>
</tr>
<tr>
<td>hgatp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HS-mode CSRs for accessing Guest/VM state</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsstatus</td>
</tr>
<tr>
<td>vsie</td>
</tr>
<tr>
<td>vsip</td>
</tr>
<tr>
<td>vstvec</td>
</tr>
<tr>
<td>vsepc</td>
</tr>
<tr>
<td>vscause</td>
</tr>
<tr>
<td>vstval</td>
</tr>
<tr>
<td>vsatp</td>
</tr>
<tr>
<td>vsscratch</td>
</tr>
</tbody>
</table>
RISC-V H-Extension: Two-stage MMU

Hardware optimized guest memory management

• Two-Stage MMU for VS/VU-mode:
  – VS-mode page table (Stage1):
    • Translates Guest Virtual Address (GVA) to Guest Physical Address (GPA)
    • Programmed by Guest (same as before)
  – HS-mode guest page table (Stage2):
    • Translates Guest Physical Address (GPA) to Host Physical Address (HPA)
    • Programmed by Hypervisor

• In HS-mode, software can program two page tables:
  – HS-mode page table: Translate hypervisor Virtual Address (VA) to Host Physical Address (HPA)
  – HS-mode guest page table: Translate Guest Physical Address (GPA) to Host Physical Address (HPA)

• Format of VS-mode page table, HS-mode guest page table and HS-mode host page table is same (Sv32, Sv39, Sv48, ....)
RISC-V H-Extension: I/O & Interrupts

I/O and guest interrupts virtualization

• Virtual interrupts injected by updating VSIP CSR from HS-mode

• Software and Timer Interrupts:
  – Hypervisor will emulate SBI calls for Guest

• HS-mode guest page table can be used to trap-n-emulate MMIO accesses for:
  – Software emulated PLIC
  – VirtIO devices
  – Other software emulated peripherals
Xvisor Overview

The Extensible Versatile Hypervisor
What is Xvisor?

The Extensible Versatile Hypervisor

• **XVISOR** = eXtensible Versatile hypervisor

• Xvisor is an open-source GPLv2 Type-1 monolithic (i.e. Pure Type-1) hypervisor

• Community driven open source project
  (http://xhypervisor.org, xvisor-devel@googlegroups.com)

• 9+ years of development and hardening (since 2010)

• Supports variety of architectures: ARMv5, ARMv6, ARMv7, ARMv7ve, ARMv8, x86_64, and RISC-V (work-in-progress)

• Primarily designed and developed for embedded virtualization

• First paper in IEEE PDP 2015 titled “*Embedded Hypervisor Xvisor: A comparative analysis*”
Xvisor: Traditional Classification

How does Xvisor compare with other hypervisors?

**Type 1 Examples:** Xvisor, Xen, VMWare ESX Server, Microsoft HyperV, OKL4 Microvisor, etc

**Type 2 Examples:** Linux KVM, FreeBSD Bhyve, VMWare Workstation, Oracle VirtualBox, etc
Xvisor: Features

Lots of features

• Virtualization Infrastructure:
  • Device tree based configuration
  • Soft real-time pluggable scheduler
  • Hugepage for Guest and Host
  • Tickless and high-resolution timekeeping
  • Host device driver framework
  • Threading framework
  • Runtime loadable modules
  • Management terminal
  • Light-weight virtual filesystem (VFS)
  • White-box testing framework
  • ... Many More ...
Xvisor: Features (Contd.)

Lots of features

• Domain Isolation:
  • VCPU and Host Interrupt Affinity
  • Spatial and Temporal Memory Isolation

• Device Virtualization:
  • Pass-through device support (Not available for RISC-V due to lack of IOMMU)
  • Block device virtualization
  • Network device virtualization
  • Input device virtualization
  • Display device virtualization
  • VirtIO v0.9.5 for para-virtualization

• Domain Messaging:
  • Sharing On-chip Coprocessor
  • Zero-copy Inter-Guest Communication
Xvisor: Key Aspects

Everything is a VCPU in Xvisor

• Scheduling entity is VCPU

• Two types of VCPUs:
  1. Normal VCPU: A VCPU belonging to Guest/VM
  2. Orphan VCPU: A VCPU belonging to Hypervisor for background processing

• Orphan VCPUs are very light-weight compared to Normal VCPUs

• Scheduler supports pluggable scheduling policy, available policies:
  • Fixed priority round-robin
  • Fixed priority rate monotonic

• Scheduling policies are soft real-time
Xvisor: Key Aspects (Contd.)

Three Execution Context

• Linux kernel runs in two possible contexts: **process context** OR **interrupt context**

• Xvisor runs in three possible contexts:
  1. **Normal Context**: Handling trap for a Normal VCPU
  2. **Orphan Context**: Running an Orphan VCPU
  3. **Interrupt Context**: Processing Host interrupt

• **Allowed to sleep in Orphan Context only**

• This ensures:
  – Host interrupt handlers can never blocks
  – Trap handling (Stage2 traps, MMIO traps, etc) for Guest can never block

• **Xvisor provides predictable delay in MMIO emulation and Trap handling for Guest**
Xvisor RISC-V

The RISC-V port of Xvisor
Xvisor RISC-V

World’s first Type-1 RISC-V hypervisor

- Hypervisor Component
- M-mode Software
- HS-mode Software
- VS-mode Software
- VU-mode Software
- U-mode Software

- Guest User Space
- Guest Kernel
- VirtIO Frontends
- Guest0
- Guest1
- GuestN
- CPU Virtualization
- Management Terminal
- VirtIO Backends
- Orphan vCPUs

- Guest IO Emulation
- Device Drivers (Host HW Access)

- Xvisor Hypervisor

- Firmware (OpenSBI)

- VU-mode
- VS-mode
- HS-mode
- M-mode
# Xvisor RISC-V: VCPU Context

What things are saved/restored for a VCPU?

## struct arch_regs (Orphan and Normal VCPUs)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Save/Restore</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>Zero register</td>
<td>---</td>
</tr>
<tr>
<td>ra</td>
<td>Return address register</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>sp</td>
<td>Stack pointer register</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>gp</td>
<td>Global pointer register</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>tp</td>
<td>Thread pointer register</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>a0-a7</td>
<td>Function argument registers</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>t0-t6</td>
<td>Caller saved registers</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>s0-s11</td>
<td>Callee saved register</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>sepc</td>
<td>Program counter</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>sstatus</td>
<td>Shadow SSTATUS CSR</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>hstatus</td>
<td>Shadow HSTATUS CSR</td>
<td>Trap Entry/Exit</td>
</tr>
<tr>
<td>sp_exec</td>
<td>Stack pointer for traps</td>
<td>Trap Entry/Exit</td>
</tr>
</tbody>
</table>

## struct riscv_priv (Normal VCPUs Only)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Save/Restore</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlen</td>
<td>Register width</td>
<td>---</td>
</tr>
<tr>
<td>isa</td>
<td>Feature bitmap</td>
<td>---</td>
</tr>
<tr>
<td>vsstatus</td>
<td>SSTATUS CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vsie</td>
<td>SIE CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vstvec</td>
<td>STVEC CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vsscratch</td>
<td>SSCRATCH CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vsepc</td>
<td>SEPC CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vscause</td>
<td>SCAUSE CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vstval</td>
<td>STVAL CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vsip</td>
<td>SIP CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>vsatp</td>
<td>SATP CSR</td>
<td>Context Switch</td>
</tr>
<tr>
<td>fp</td>
<td>Floating-point Registers</td>
<td>Context Switch</td>
</tr>
</tbody>
</table>
Xvisor RISC-V: Host Interrupts

How are host interrupts handled?
Xvisor RISC-V: Context Switch

How is a Guest VCPU preempted?

Diagram showing the process of context switch with RISC-V Trap Entry, Host IRQ Subsystem, Timer Subsystem, Scheduler, RISC-V Context Switch, and Timer IRQ.
Xvisor RISC-V: Guest MMIO Emulation

How is a Guest VCPU MMIO trap handled?

Normal Context
(Predictable delay because we cannot sleep)

Guest

HS-mode
RISC-V Trap Entry

RISC-V Page Fault Handler

Device Emulation Framework

Device Emulator

RISC-V Trap Exit

VS/VU-mode

Guest

MMIO Trap

Time

Time
Xvisor RISC-V: Guest RAM handling

How is Guest RAM accessed from Xvisor?

• **Guest RAM is not pre-mapped in HS-mode page table** to:
  – Reduce memory consumed by HS-mode page table
  – Avoid cache aliasing by not having two different virtual addresses for Guest RAM pages

• **Guest RAM is accessed 4K (Page Size) at a time iteratively**, as follow:
  – Map 4K page of Guest RAM in HS-mode page table
  – Access 4K page of Guest RAM using hypervisor virtual address
  – Unmap 4K page of Guest RAM from HS-mode page table

• **RISC-V unprivileged load/store can be used** to improve Guest RAM accesses

• **Host hugepages** to make Xvisor memory access faster

• **Guest hugepages** to make Guest OS memory access faster

• **RISC-V hugepage size is**: 2M or 1G for RV64 and 4M for RV32
Xvisor RISC-V: SBI Interface

Syscall style interface between Host and Guest

- **SBI = Supervisor Binary interface**
- **SBI v0.1 in-use by Linux kernel**
  (Refer, [https://github.com/riscv/riscv-sbi-doc/blob/v0.1.0/riscv-sbi.md](https://github.com/riscv/riscv-sbi-doc/blob/v0.1.0/riscv-sbi.md))
- **SBI v0.2 in draft stage**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Function ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>sbi_set_timer</td>
<td>0</td>
</tr>
<tr>
<td>IPI</td>
<td>sbi_clear_ipi</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>sbi_send_ipi</td>
<td>4</td>
</tr>
<tr>
<td>Memory Model</td>
<td>sbi_remote_fence_i</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>sbi_remote_sfence_vma</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>sbi_remote_sfence_vma_asid</td>
<td>7</td>
</tr>
<tr>
<td>Console</td>
<td>sbi_console_putchar</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>sbi_console_getchar</td>
<td>2</td>
</tr>
<tr>
<td>Shutdown</td>
<td>sbi_shutdown</td>
<td>8</td>
</tr>
</tbody>
</table>
Xvisor RISC-V: Device tree based configuration

Using DT for both Host and Guest

Three types of device tree (DT):

1. **Host DT:**
   - Device tree which describes underlying host HW to Xvisor
   - Used by Xvisor at boot-time

2. **Guest Xvisor DT:**
   - Device tree which describes Guest virtual HW to Xvisor
   - Used by Xvisor to create Guest

3. **Guest OS DT:**
   - Device tree which describes Guest virtual HW to Guest OS
   - Used by Guest OS at boot-time
Xvisor RISC-V: Zero-copy Inter-Guest Transfer

Transferring data between Guests with minimum overhead

• Achieved using:
  1. **VirtIO RPMSP:**
     • Used for control messages
     • Name-service notifications
  2. **Shared Memory:**
     • Used for actual data transfers
     • Very fast zero-copy

• Linux applications can transfer data across Guests using **RPMSG character device**

• **VirtIO Network** can also be used in-place of VirtIO RPMSG
# Xvisor RISC-V: Code Size and Memory Usage

Lines of code, binary size and runtime memory usage

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>Comments</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch/riscv/</td>
<td>2099</td>
<td>6357</td>
</tr>
<tr>
<td>core/</td>
<td>9151</td>
<td>35989</td>
</tr>
<tr>
<td>commands/</td>
<td>1031</td>
<td>10409</td>
</tr>
<tr>
<td>daemons/</td>
<td>147</td>
<td>524</td>
</tr>
<tr>
<td>drivers/</td>
<td>19094*</td>
<td>62395*</td>
</tr>
<tr>
<td>emulators/</td>
<td>4117*</td>
<td>25375*</td>
</tr>
<tr>
<td>libs/</td>
<td>6023</td>
<td>18296</td>
</tr>
<tr>
<td>TOTAL</td>
<td>41662</td>
<td>159345</td>
</tr>
</tbody>
</table>

* Can be further decreased or increased based on compile-time configuration

<table>
<thead>
<tr>
<th>BLOB</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>580 KB*</td>
</tr>
<tr>
<td>.data</td>
<td>50 KB</td>
</tr>
<tr>
<td>.rodata</td>
<td>205 KB*</td>
</tr>
<tr>
<td>.bss</td>
<td>141 KB</td>
</tr>
<tr>
<td>vmm.bin</td>
<td>849 KB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime Memory</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text memory freed at boot-time</td>
<td>136 KB</td>
</tr>
<tr>
<td>Typical memory usage</td>
<td>22.6 MB*</td>
</tr>
<tr>
<td>Max VAPOOL limit</td>
<td>32 MB*</td>
</tr>
</tbody>
</table>

**NOTE:** Stats gathered from Xvisor-next on 21st October 2019 for RV64GC
Xvisor RISC-V: Ideal for Embedded Systems

Why is Xvisor RISC-V ideal for Embedded Systems?

- No dependency on any Guest OS for running management tools
- Single software providing complete virtualization solution
- Guest types described using device tree instead of fixed Guest types
- Soft real-time and pluggable scheduler
- Predictable delay in MMIO emulation and Trap handling for Guest
- Para-virtualization complying open-standards (such as VirtIO)
- Zero-copy inter-guest communication
- Low memory footprint with reasonable code size
- Playground for academic research
Status & Future Work

Where are we? and What next?
Xvisor RISC-V: Current State

Where are the patches?

• Initial Xvisor RISC-V port was released last year (21st October 2018) in Xvisor v0.2.11
  – Just capable of booting on QEMU without using any HS-mode CSRs
  – QEMU with H-Extension was not available at that time

• Currently, Xvisor RISC-V is in very good shape (comparable to Xvisor ARM64):
  – Able to boot Guests with multiple VCPUs on SMP Host
  – Able to boot Open Embedded and Fedora as Guest OS

• Xvisor v0.3.0 release will have a feature complete RISC-V port but it is delayed due to:
  – Few pending RISC-V spec changes
  – Few pending QEMU H-Extension fixes

• To play with Xvisor RISC-V on QEMU refer:
  – docs/riscv/riscv-virt-qemu.txt in latest Xvisor-next sources

• To follow Xvisor RISC-V development, please join xvisor-devel@googlegroups.com
Xvisor RISC-V: TODO List

What next?

• Get Xvisor RISC-V 32-bit working
• SBI v0.2 base and replacement extensions support
• SBI v0.2 para-virtualized time accounting
• Access Guest RAM using RISC-V unprivileged load/store
• Loadable module support
• Virtualize vector extensions
• Libvirt support
• Allow 32bit Guest on 64bit Host (Defined in RISC-V spec)
• Allow big-endian Guest on little-endian Host and vice-versa (Defined in RISC-V spec)
• ..... and more .....
Xvisor RISC-V Demo

Xvisor running on QEMU RISC-V
Questions  ???
Backup
RISC-V H-Extension: Compare ARM64

How is RISC-V H-Extension compared to ARM64 virtualization?

<table>
<thead>
<tr>
<th>RISC-V H-Extension v0.4 draft</th>
<th>ARM64 (ARMv8.x) Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No separate privilege mode for hypervisors.</strong> Extends S-mode with hypervisor capabilities (HS-mode) and Guest/VM run in virtualized S-mode/U-mode (VS-mode or VU-mode).</td>
<td><strong>Separate EL2 exception-level for hypervisors</strong> with its own &lt;xyz&gt;_EL2 MSRs. The Guest/VM will run in EL1/EL0 exception levels.</td>
</tr>
<tr>
<td><strong>Well suited for both Type-1 (baremetal) and Type-2 (hosted) hypervisors.</strong> The S&lt;xyz&gt; CSRs access from VS-mode map to special VS&lt;xyz&gt; CSRs which are only accessible to HS-mode and M-mode.</td>
<td><strong>Special ARMv8.1-VHE Virtualization Host Extension for better performance of Type-2 (hosted) hypervisor.</strong> Allows Host kernel (meant for EL1) to run in EL2 by mapping &lt;xyz&gt;_EL1 MSRs to &lt;abc&gt;_EL2 MSRs in Host mode.</td>
</tr>
<tr>
<td><strong>Virtual interrupts for Guest/VM injected using VSIP CSR.</strong> The hypervisor does not require any special save/restore but it will emulate entire PLIC in software.</td>
<td><strong>Virtual interrupts for Guest/VM injected using LR registers of GICv2/GICv3 with virtualization extension.</strong> The hypervisor will save/restore LR registers and emulate all GIC registers in software except GIC CPU registers.</td>
</tr>
</tbody>
</table>
# RISC-V H-Extension: Compare ARM64 (Contd.)

How is RISC-V H-Extension compared to ARM64 virtualization?

<table>
<thead>
<tr>
<th>RISC-V H-Extension v0.4 draft</th>
<th>ARM64 (ARMv8.x) Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual timer events for Guest/VM using SBI calls emulated by hypervisor. The SBI calls trap to hypervisor so save/restore of virtual timer state not required.</td>
<td>Virtual timer events for Guest/VM using ARM generic timers with virtualization support. The hypervisor will save/restore virtual timer state and manage virtual timer interrupts.</td>
</tr>
<tr>
<td>Virtual inter-processor interrupts for Guest/VM using SBI calls emulated by hypervisor. The hypervisor does not require any special save/restore.</td>
<td>Virtual inter-processor interrupts for Guest/VM by emulating ICC_SGI1R_EL1 (virtual GICv3) or GICD_SGIR (virtual GICv2). The save/restore will be handled as part of LR registers save/restore.</td>
</tr>
<tr>
<td>Nested virtualization supported using HSTATUS.TVM and HSTATUS.TSR bits. The hypervisor will trap-n-emulate Guest hypervisor capabilities.</td>
<td>Special ARMv8.3-NV for supporting nested virtualization on ARMv8. The hypervisor will trap-n-emulate Guest hypervisor capabilities. The ARMv8.4-NV further enhances nested virtualization support.</td>
</tr>
</tbody>
</table>
Linux KVM RISC-V

Process1 ... ProcessN

Host User Space

Guest User Space

Guest Kernel

VirtIO Frontends

QEMU/KVMTOOL
(Guest IO Emulation + VirtIO Backends)

Process (Guest0)

Device Drivers
(Host HW Access)

Host Linux Kernel

KVM Module
(CPU Virtualization)

Firmware (OpenSBI)

Hypervisor Component

M-mode Software

HS-mode Software

VS-mode Software

VU-mode Software

U-mode Software

VU-mode

VS-mode

U-mode

HS-mode

M-mode
Xen RISC-V (Work-in-progress)

- Dom0 (Guest0 - Control)
  - Dom0 User Space
  - Dom0 Kernel
    - Device Drivers (Host HW Access)
    - Xen PV Backends
  - Xen Toolstack (Management)

- Dom1 (Guest1)
  - DomU User Space
  - DomU Kernel
    - Xen PV Frontends

- DomN (GuestN)
  - DomU User Space
  - DomU Kernel
    - Xen PV Frontends

- CPU Virtualization
- Hypervisor
- Basic Host HW Access
- Firmware (OpenSBI)

- Hypervisor Component
- M-mode Software
- HS-mode Software
- VS-mode Software
- VU-mode Software
- U-mode Software