



BREKER™

**Complex Verification Example:
RISC-V MMU Verification of Virtualization and
Hypervisor Operation for CPU and SOC platforms**

**Verification Futures Reading 2025
David Kelf, CEO, Breker Verification Systems**

Agenda

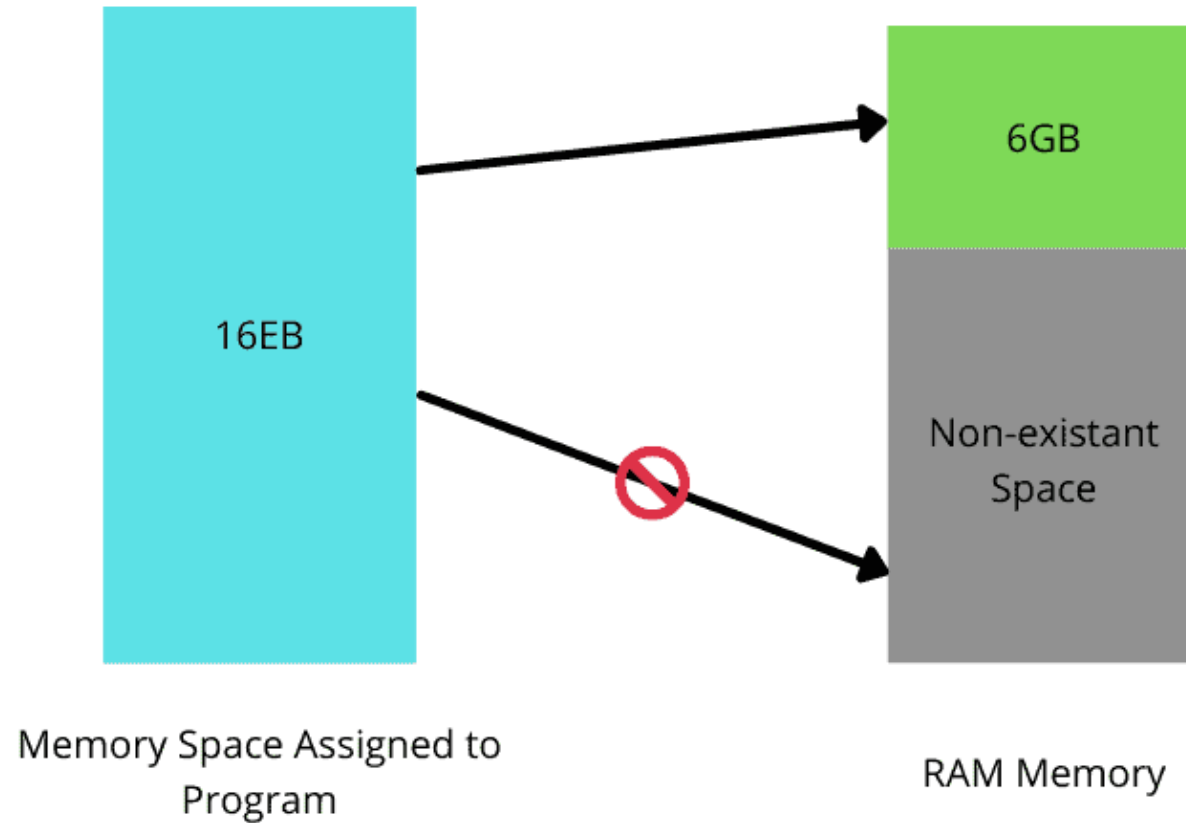
- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

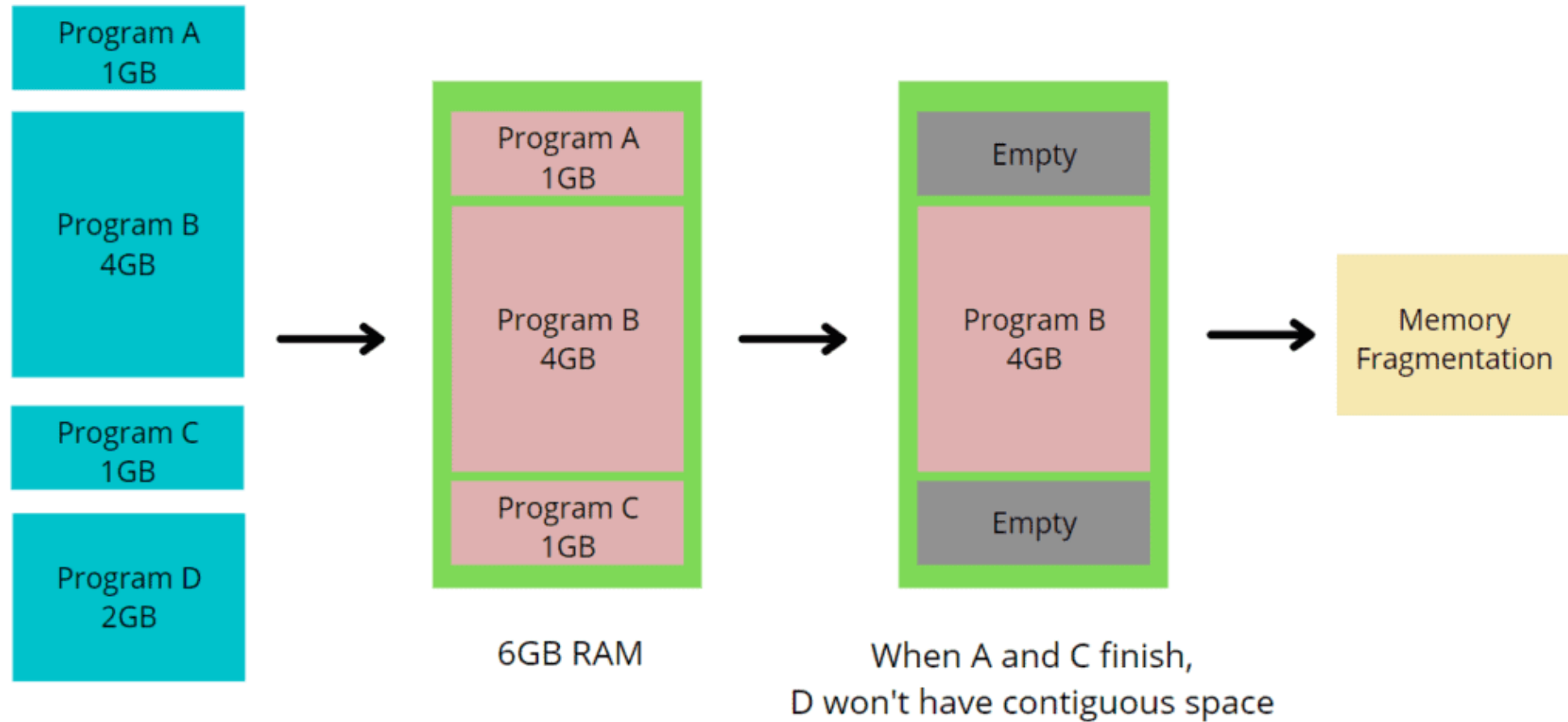
Motivation for MMU (1/3)

Allow software to use more memory than physically available



Motivation for MMU (2/3)

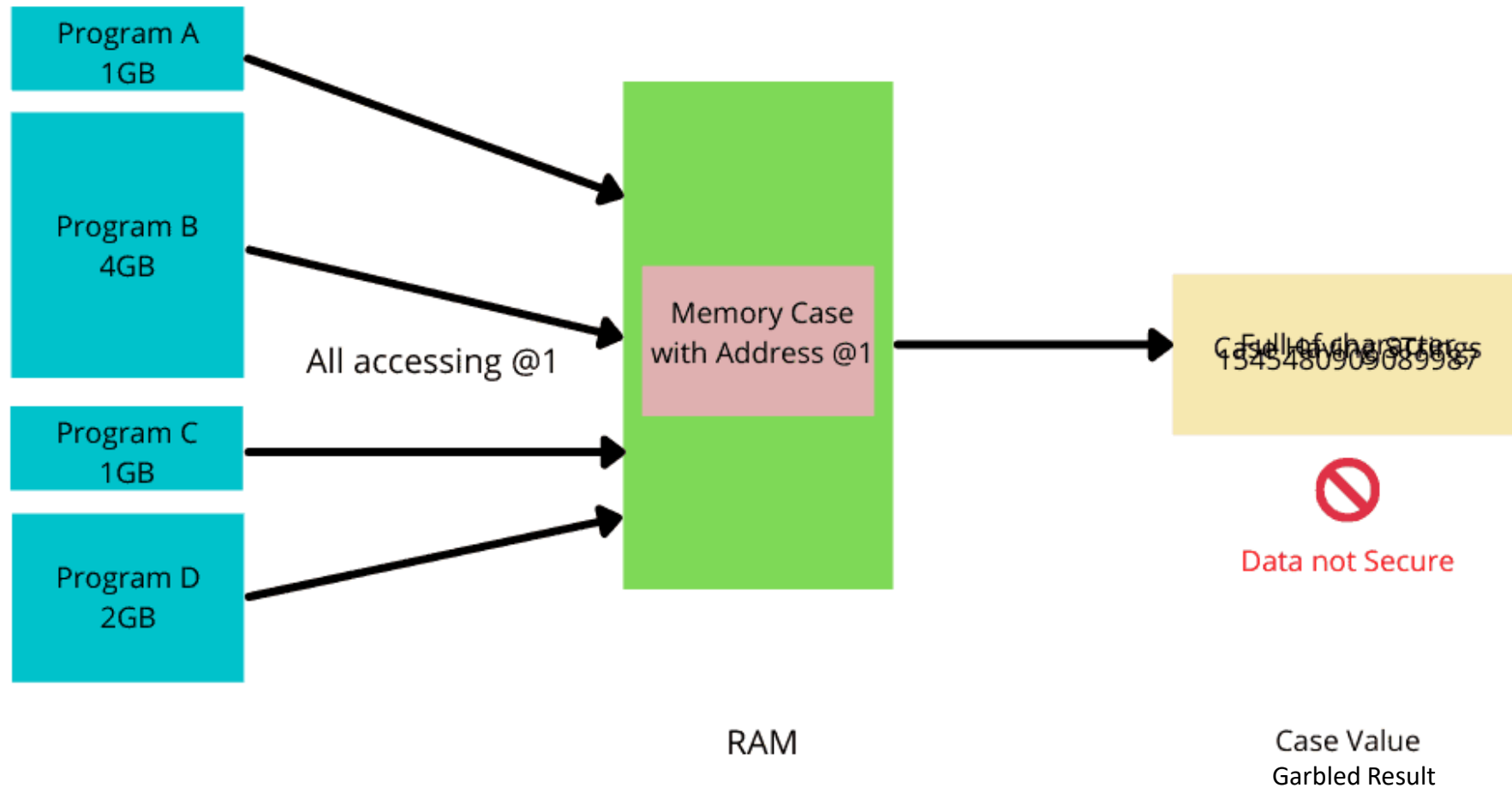
Handle Memory Fragmentation



Attribution: <https://www.baeldung.com/cs/virtual-memory-why>

Motivation for MMU (3/3)

Isolate memory accesses across processes

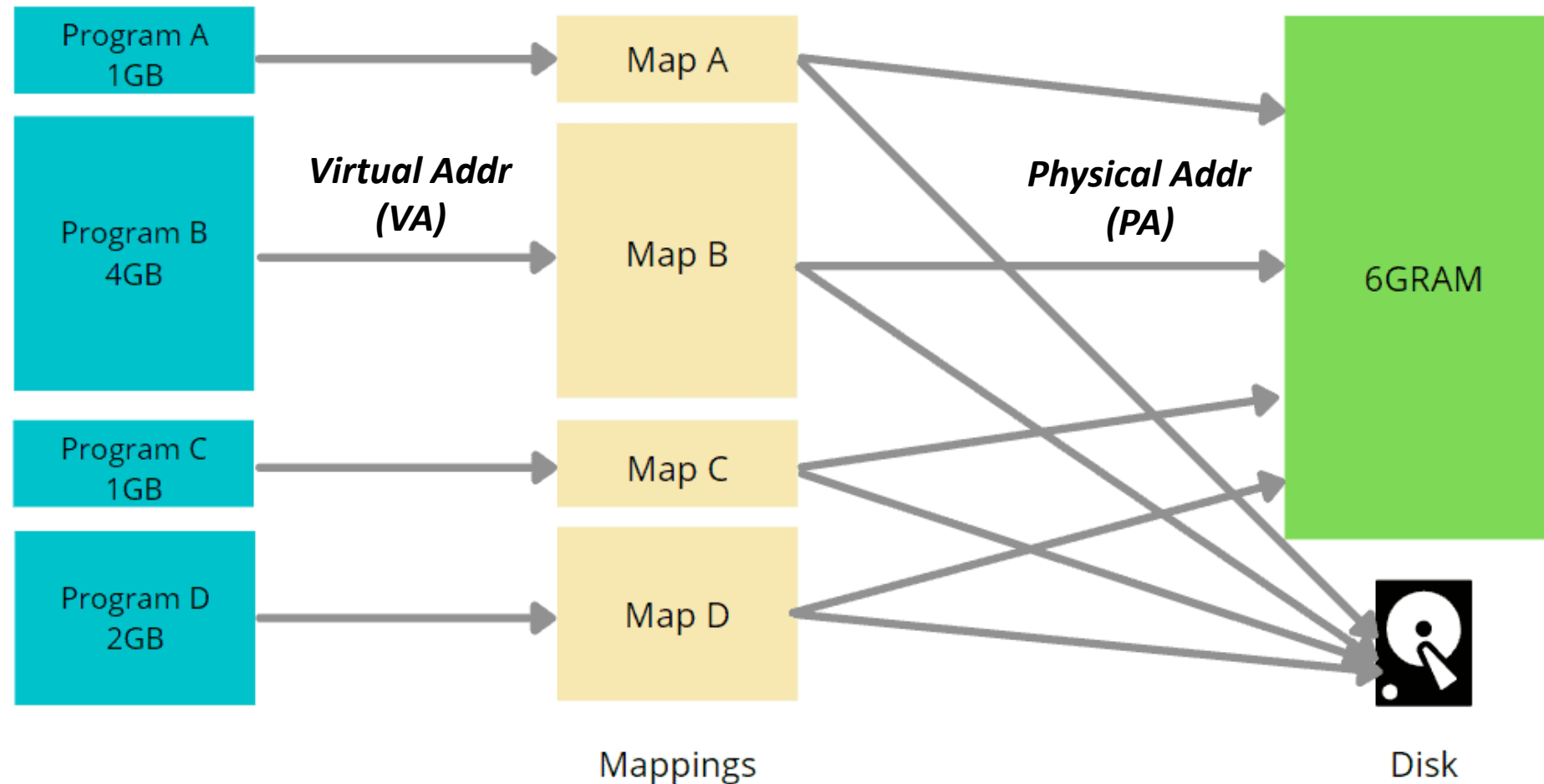


Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

Virtual Memory Mapping

Each program sees a *virtual address (VA)* that is *mapped* to a *physical address (PA)*

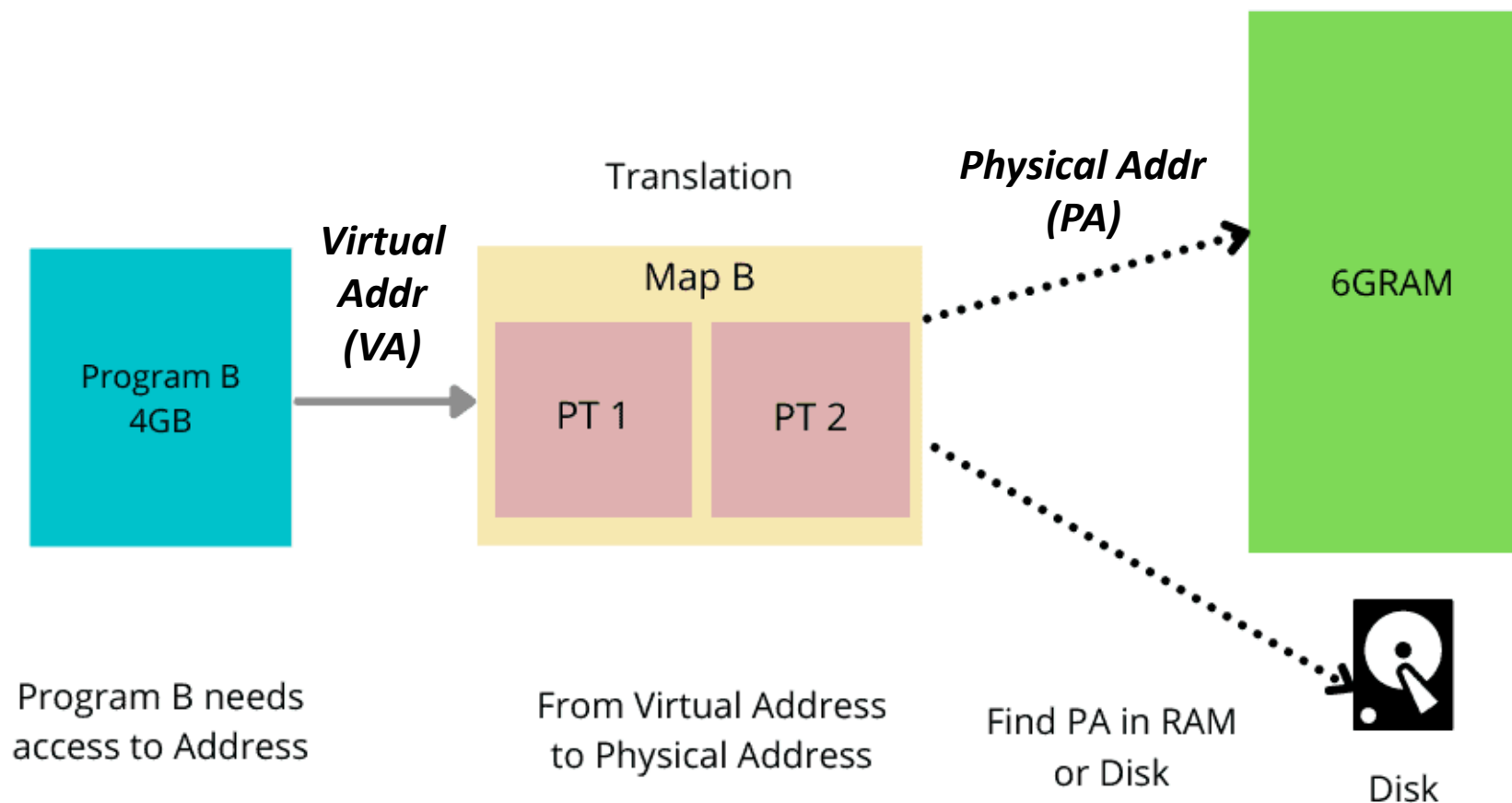


Attribution: <https://www.baeldung.com/cs/virtual-memory-why>

Virtual to Physical mapping via Page Tables

Each program gets a multi-stage page table lookup

- Having multiple levels of page tables reduces overall memory usage

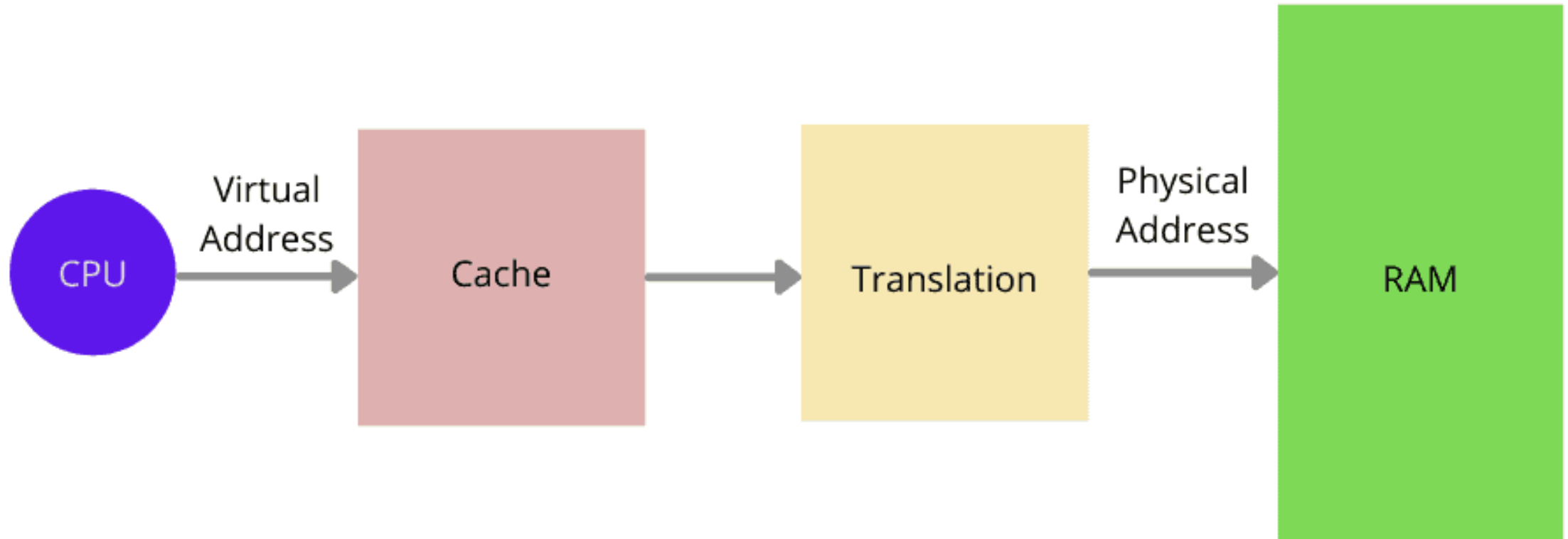




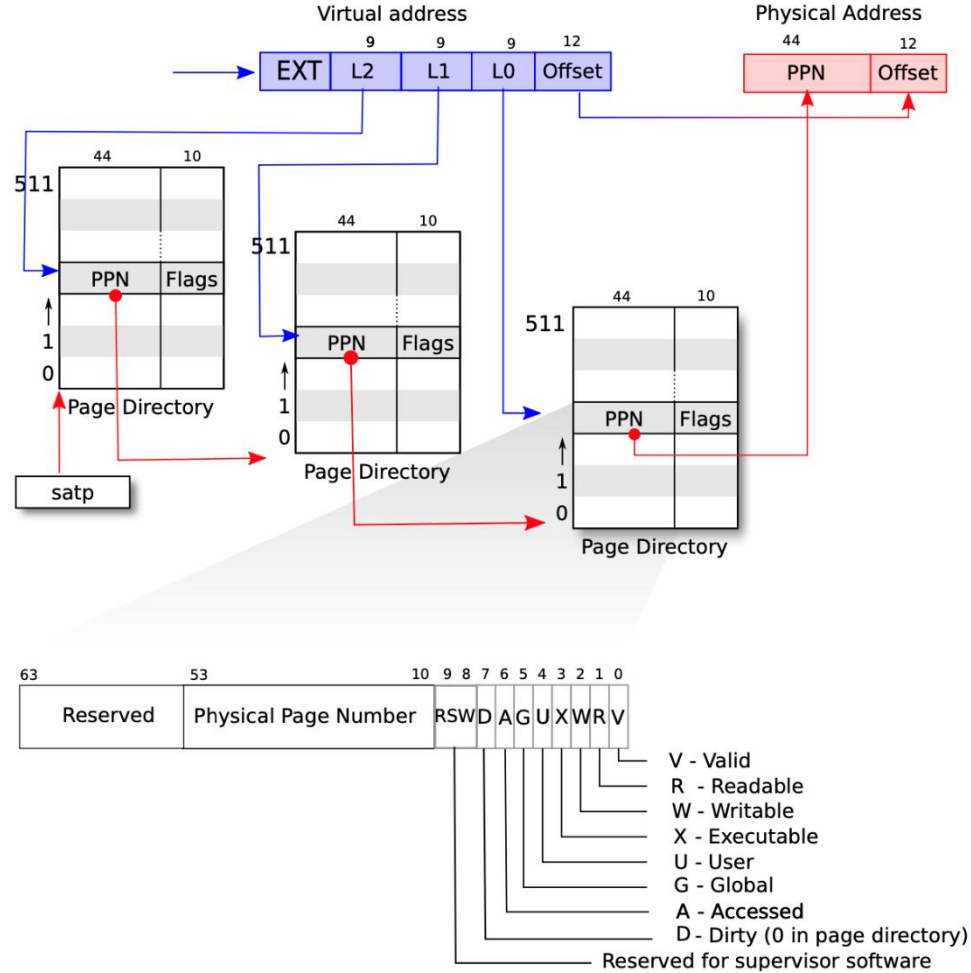
Virtual to Physical mapping via Page Tables

TLB Cache store a copy of previous table walks to improve performance

- **Software must flush appropriate entries from TLB when page mapping is changed**



One Stage Address Translation Page Table Walks

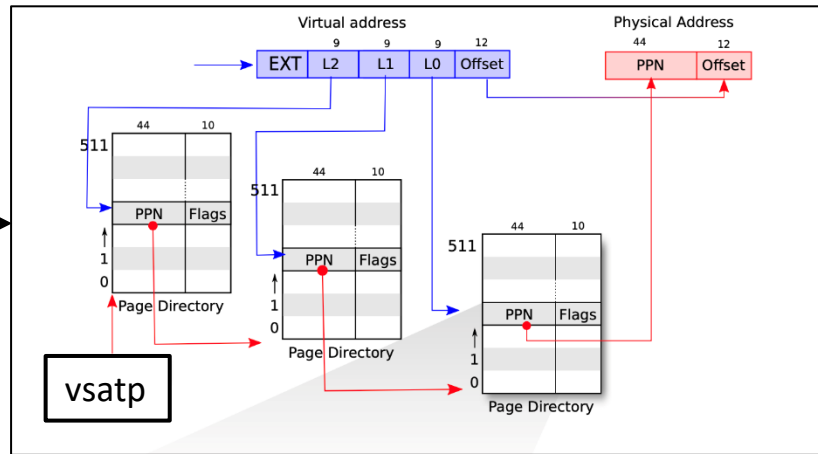


- Build up sparse tree of page directories
- S/U mode

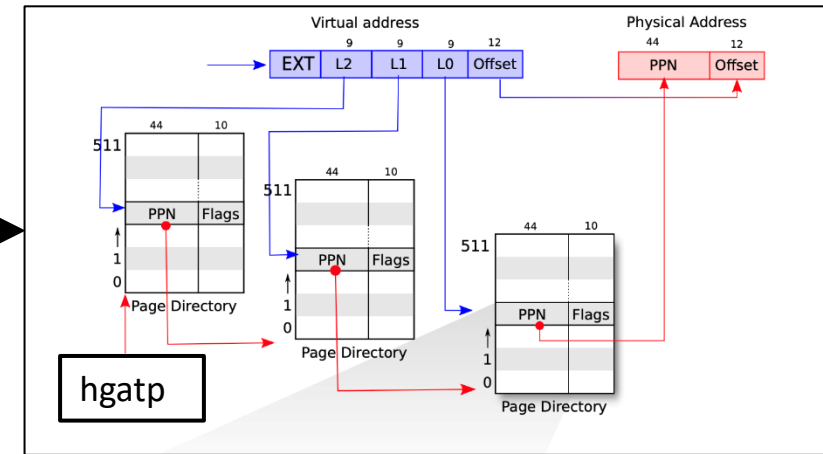
Figure 3.2: RISC-V address translation details.

Hypervisor and Two-Stage Address Translation

Guest
Virtual
Address
(GVA)



Guest
Physical
Address
(GPA)



Physical
Address
(PA)

virtual-VS/VU-mode

Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

Vignettes from the RISC-V ISA: satp CSR

3.1.11. Supervisor Address Translation and Protection (satp) Register

The `satp` CSR is an `SXLEN`-bit read/write register, formatted as shown in Supervisor address translation and protection (`satp`) register when `SXLEN=32`. for `SXLEN=32` and Supervisor address translation and protection (`satp`) register when `SXLEN=64`, for `MODE` values Bare, Sv39, Sv48, and Sv57. for `SXLEN=64`, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the `MODE` field, which selects the current address-translation scheme. Further details on the access to this register are described in [virt-control].

MODE

Fig

Encoding of `satp` `MODE` field. shows the encodings of the `MODE` field when `SXLEN=32` and `SXLEN=64`. When `MODE=Bare`, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in [pmp]. To select `MODE=Bare`, software must write zero to the remaining fields of `satp` (bits 30–0 when `SXLEN=32`, or bits 59–0 when `SXLEN=64`). Attempting to select `MODE=Bare` with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on address translation and protection behavior.

When `SXLEN=64`, three paged virtual-memory schemes are defined: Sv39, Sv48, and Sv57, described in Sv39: Page-Based 39-bit Virtual-Memory System, Sv48: Page-Based 48-bit Virtual-Memory System, and Sv57: Page-Based 57-bit Virtual-Memory System, respectively. One additional scheme, Sv64, will be defined in a later version of this specification. The remaining `MODE` settings are reserved for future use and may define

The `satp` CSR is considered *active* when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of `satp` when `satp` is active.

Vignettes from the RISC-V ISA: sfence.vma instruction

3.2.1. Supervisor Memory-Management Fence Instruction



The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *rs1* and *rs2*, as

described below. For the common case that the translation data structures have only been modified for a single address associated with a mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation.

• If *rs1* ≠ x0 and *rs2* = x0, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1* for all address spaces. The fence also invalidates all address-translation entries for that virtual address in *rs1*.

An implicit read of the memory-management data structures may return any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. The ordering implied by SFENCE.VMA does not place implicit reads and writes to the memory-management data structures into the global memory order in a way that interacts cleanly with the standard RVWMO ordering rules. In particular, even though an SFENCE.VMA orders prior explicit accesses before subsequent implicit accesses, and those implicit accesses are ordered before their associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.

Vignettes from the RISC-V ISA: Page Table Entries

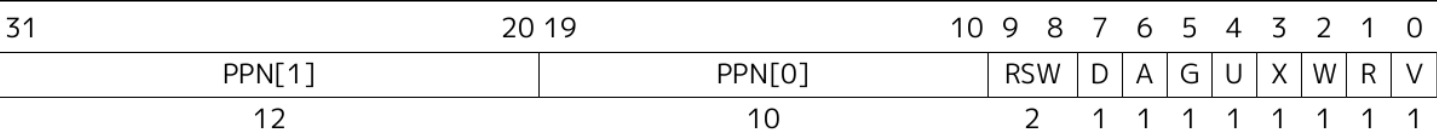


Figure 19. Sv32 page table entry.

The PTE for a given virtual address is fetched from the page table. If the PTE is not valid, a page-fault exception is raised. If the PTE is valid, the permissions are checked. Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception. Irrespective of SUM, the supervisor may not execute code on pages with U=1.



The U bit indicates the page when access pages with



The G bit designates global pages. For non-leaf PTEs, the G bit is clear, whereas marking a page as global in a different mapping being used.

An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the SUM field.

The RSW field is reserved for use by supervisor software; the implementation shall ignore this field.

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are defined:

- The Svade extension: when a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.

Vignettes from the RISC-V ISA: Virtual Address Translation Process

3.3.2. Virtual Address Translation Process

A virtual address va is translated into a physical address pa as follows:

1. Let $satp$ be the value of the `satp` register.
2. Let acc be the access type, and $orig$ be the original virtual address.
3. If acc is a store and $satp$ is not active, stop and raise a page-fault exception.
4. Otherwise, let $next$ be the next level of the page table.
5. A leaf PTE has been reached. If $i > 0$ and $pte.ppn[i-1:0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
6. Determine if the requested memory access is allowed by the $pte.u$ bit, given the current privilege mode and the access type. If not, stop and raise a page-fault exception.
7. Determine if the requested memory access is allowed by the $pte.a$ bit, given the current privilege mode and the access type. If not, stop and raise a page-fault exception.
8. Determine if the requested memory access is allowed by the $pte.d$ bit, given the current privilege mode and the access type. If not, stop and raise a page-fault exception.
9. If $pte.a=0$, or if the original memory access is a store and $pte.d=0$:
 - If the Svade extension is implemented, stop and raise a page-fault exception corresponding to the original access type.
 - If a read-only access is requested, stop and raise a page-fault exception.
 - Perform the address translation:
 - Calculate the physical address pa by adding the offset $va.pgoff$ to the physical address $pte.ppn$.
 - If the Svade extension is implemented, the address translation must be performed in memory directly.
 - If the Svade extension is not implemented, the address translation must be performed in memory directly.
10. The translation is successful.
 - $pa.pgoff = va.pgoff$.

The results of implicit address-translation reads in step 2 may be held in a read-only, incoherent *address-translation cache* but not shared with other harts. The address-translation cache may hold an arbitrary number of entries, including an arbitrary number of entries for the same address and ASID.

Entries in the address-translation cache may then satisfy subsequent step 2 reads if the ASID

associated with the entry matches the ASID of the virtual address. Implementations may also execute the address-translation algorithm speculatively at any time, for any virtual address, as long as `satp` is active (as defined in Supervisor Address Translation and Protection (satp) Register). Such speculative executions have the effect of pre-populating the address-translation cache.

The address-translation algorithm must be performed in memory directly.

Speculative executions of the address-translation algorithm behave as non-speculative executions of the algorithm do, except that they must not set the dirty bit for a PTE, they must not trigger an exception, and they must not create address-translation cache entries if those entries would have been invalidated by any SFENCE.VMA instruction executed by the hart since the speculative execution of the algorithm began.

Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

MMU & Hypervisor Test Plan: Select Privilege Level

1.1. MMU & Hypervisor Page Translation

Tests Plan for Paging and Hypervisor functionality.

Start: [Select privilege level](#)

1.1.1. Select privilege level

IsaRef: [link](#)

CvgRef: [gotoPrv](#)

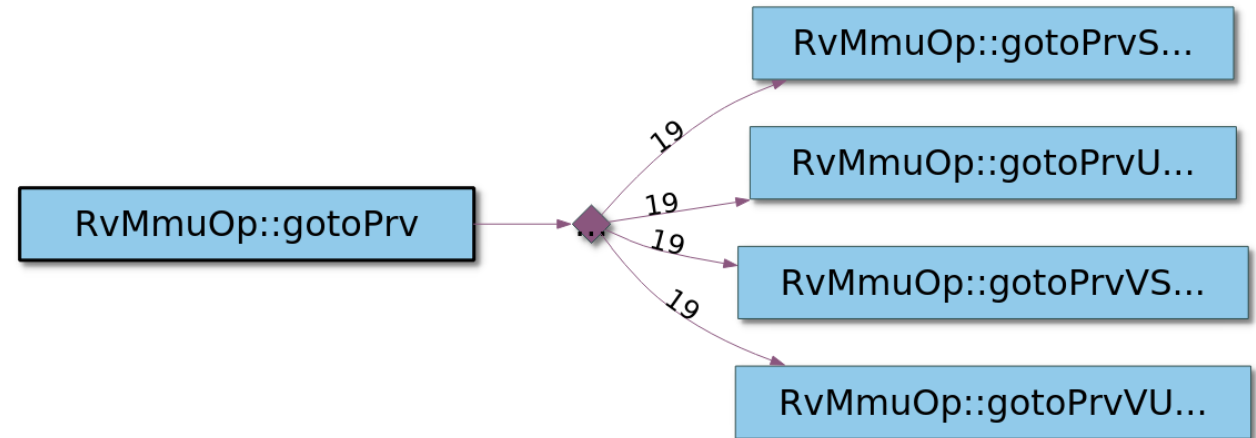
Pick a random privilege level to use for the scenario.

Select one of:

- [User mode](#)
- [Hypervisor-extended supervisor mode](#)
- [Virtual user mode](#)
- [Virtual supervisor mode](#)

NOTE: cannot run MMU operations in Machine mode.

TODO: support disabling of `misa.H` for S and U mode even if H extension is supported



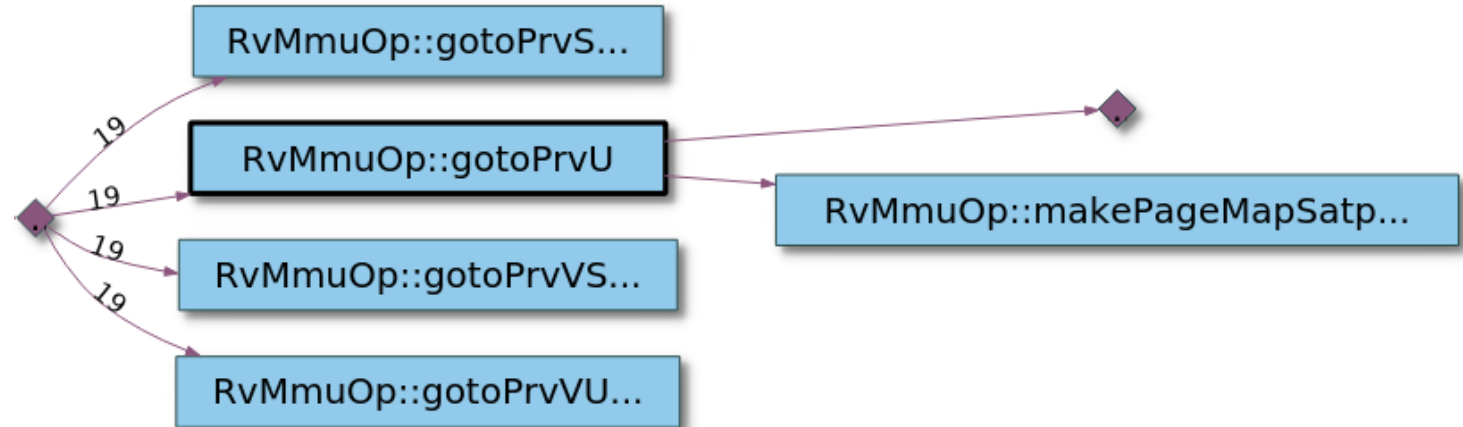
MMU & Hypervisor Test Plan: User Mode

1.1.1.1. User mode

CvgRef: gotoPrvU

Do

- require PTE.U flag
- Setup one-stage address translation
- Switch to U-mode
- Select operation



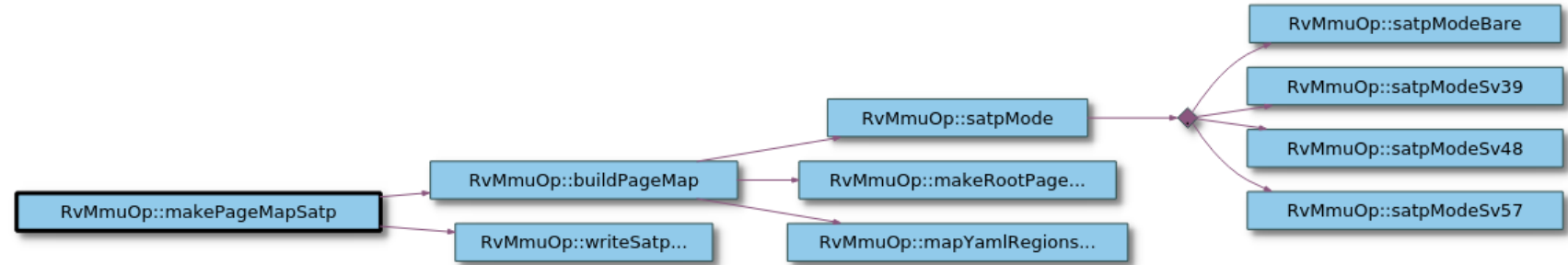
MMU & Hypervisor Test Plan: Setup one-stage address translation

1.1.3.1. Setup one-stage address translation

IsaRef: [link](#)

CvgRef: [makePageMapSatp](#)

Do



- Select one-stage paging mode
- Allocate 4KB naturally aligned root page
- Page map code stack and code addresses
- Write satp with root page table address and mode

TODO: randomize AISD mapping

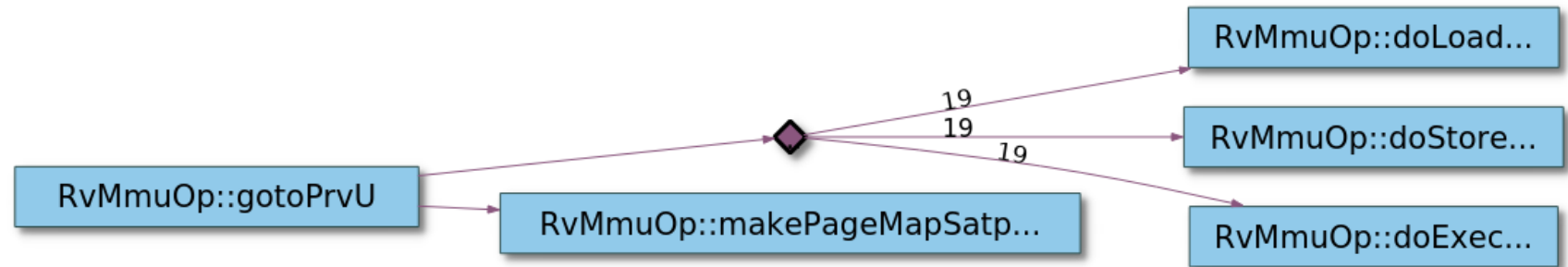
MMU & Hypervisor Test Plan: Select operation

1.1.2. Select operation

Pick an operation to exercise page translation

Select one of

- Do load operation
- Do store operation
- Do execute operation



TODO: need to add LR, SC and AMO operations.

MMU & Hypervisor Test Plan: Do store operation

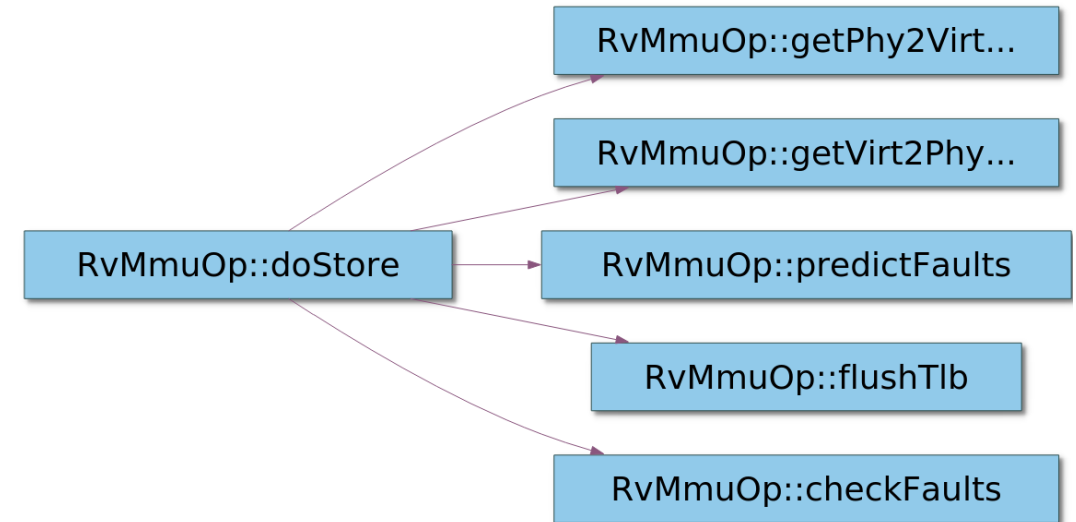
1.1.2.2. Do store operation

IsaRef: [link](#)

CvgRef: [doStore](#)

- [Allocate virtual address](#)
 - require PTE flags D, A, W, R and V set
 - if (U or VU mode) require PTE flag U set
- Store to allocated virtual address and update memory scoreboard
- [Check expected page faults](#) for mcause CAUSE_STORE_[GUEST_]PAGE_FAULT

NOTE: need both PTE.W and PTE.R — no write-only PTE flag configuration



1.1.4. Allocate virtual address

- allocate physical memory
- Page map physical address to virtual address
- predict expected page faults
- **if** (two-stage translation)
 - Generate hypervisor memory-management fence instructions
- Generate supervisor memory-management fence instruction

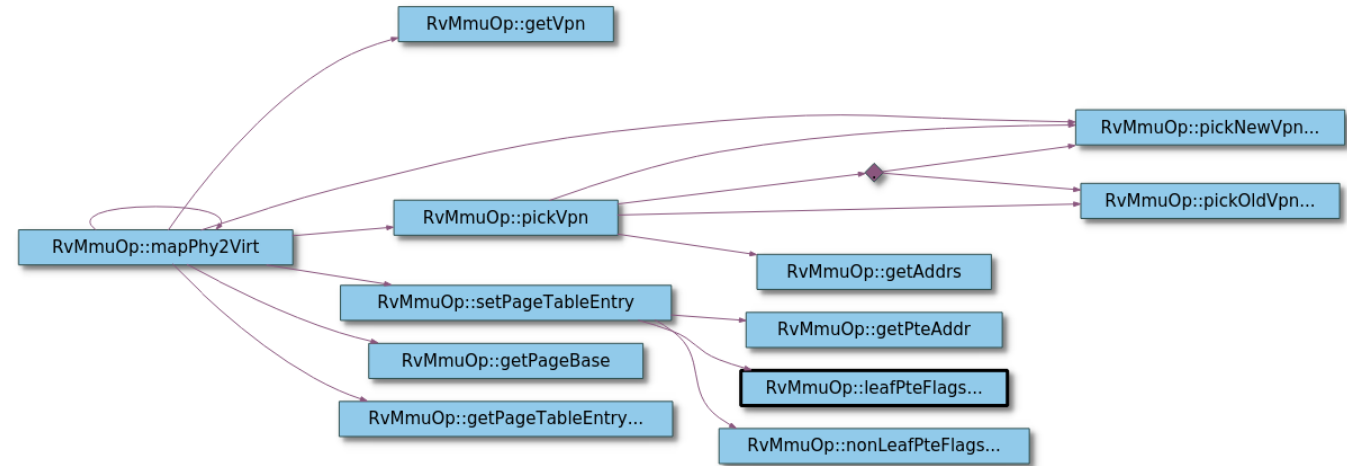
MMU & Hypervisor Test Plan: Page Map physical to virtual address

1.1.6. Page map physical address to virtual address

IsaRef: [link](#)

CvgRef: [getPhy2Virt](#)

- **if** (two-state translation)
 - [Page map physical address to virtual address](#) to get gpa
- **for each** PTE level
 - [Pick VPN for PTE](#)
 - [Pick PTE flags](#)



TODO: clarify if sign extension (e.g. must have bits 63–39 all equal to bit 38) is used

TODO: clarify usage of *megapages*, *gigapages*, *terapages* and *pentapages*

MMU & Hypervisor Test Plan: Pick PTE Flags

1.1.8. Pick PTE flags

1.1.9. Pick leaf PTE flags

IsaRef: [link](#)

CvgRef: [leafPteFlags](#)

Do

- Pick leaf PTE.V
- Pick leaf PTE.R
- Pick leaf PTE.W
- Pick leaf PTE.X
- Pick leaf PTE.U
- Pick leaf PTE.G
- Pick leaf PTE.A
- Pick leaf PTE.D
- Pick leaf PTE.RSW

NOTE: assumes *Svade* exte

TODO: randomize reserved

TODO: implement PBMT bits

TODO: implement *n* bit

1.1.9.3. Pick leaf PTE.W

CvgRef: [leafPteW](#)

- **if** (require PTE.W)
 - **Select** one of
 - PTE.W set CvgRef: [leafPteW1Set](#)
 - PTE.W clear CvgRef: [leafPteW1ClrErr](#)
 - predict page fault
- **else**
 - **Select** one of
 - PTE.W set CvgRef: [leafPteW0Set](#)
 - PTE.W clear CvgRef: [leafPteW0Clr](#)



MMU & Hypervisor Test Plan: Complete Scenario Model

```
int RvMmuOp::gotoPrv() {  
    select(  
        gotoPrvS(),  
        gotoPrvU(),  
        gotoPrvVS(),  
        gotoPrvVU()  
    );  
    return 0;  
}
```

Call graph of C/C++ program
with *select()* operator

S/U/VS/VU Priv
levels

2.23e+138 possible
Planning paths!

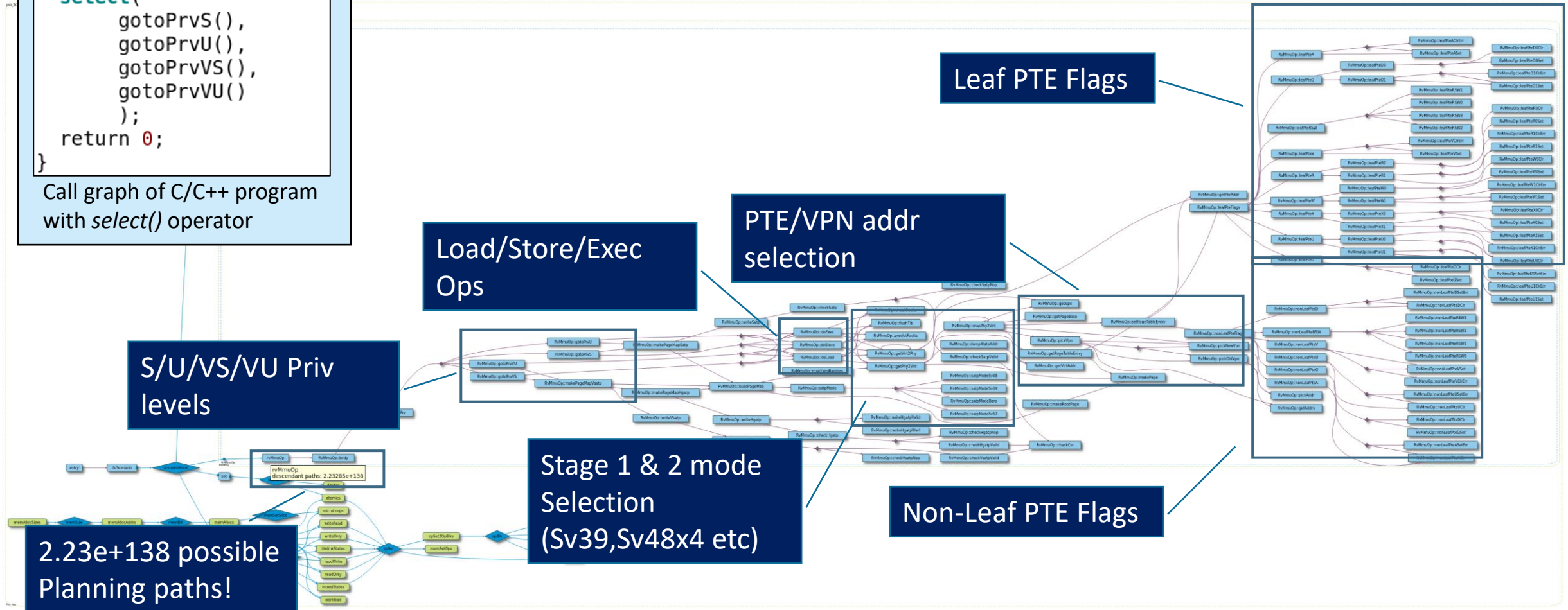
Load/Store/Exec
Ops

PTE/VPN addr
selection

Leaf PTE Flags

Stage 1 & 2 mode
Selection
(Sv39, Sv48x4 etc)

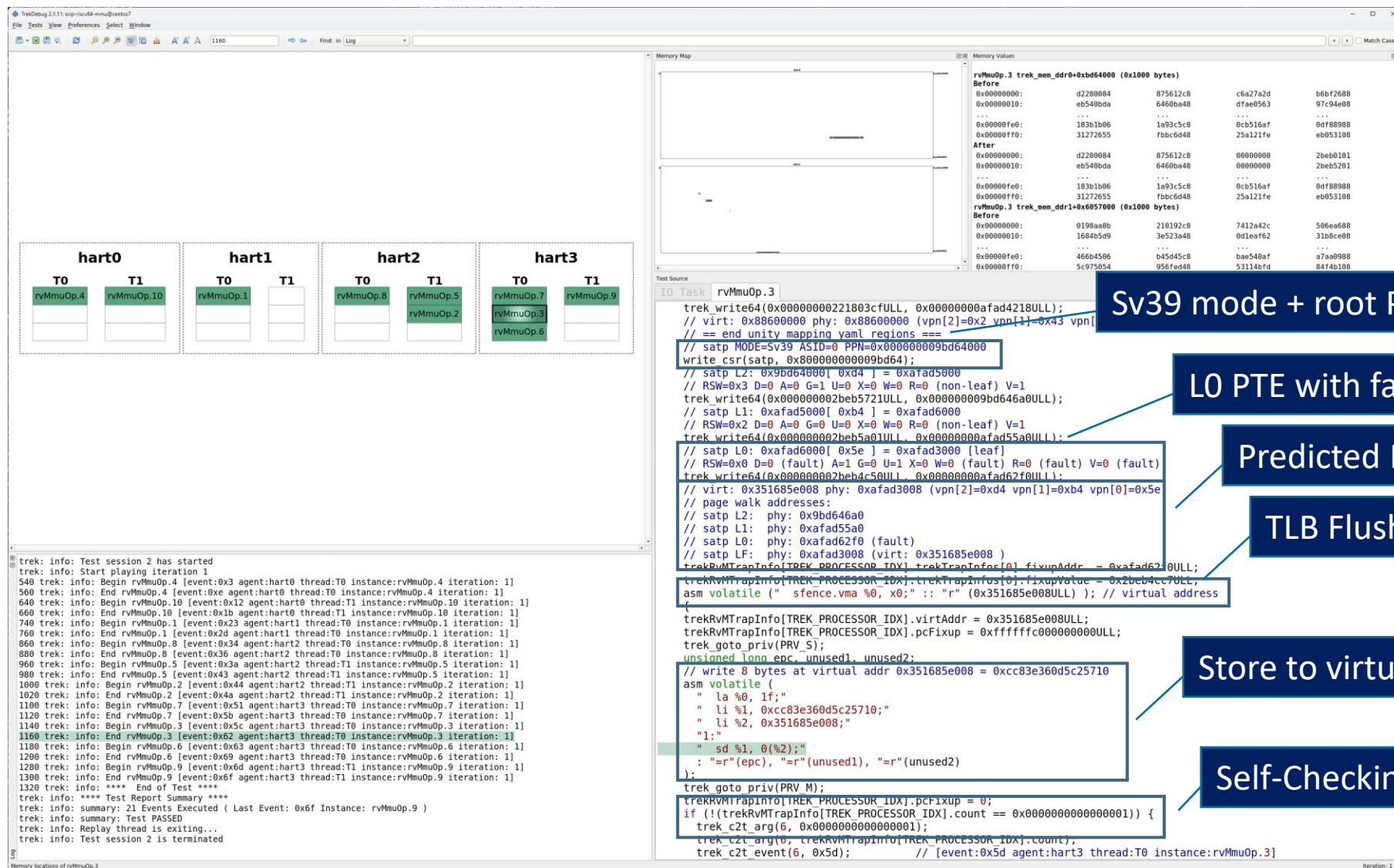
Non-Leaf PTE Flags



Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

MMU One-Stage Address Translation Example



harto hart1 hart2 hart3

TO T1

rvMmuOp.4 rvMmuOp.10

rvMmuOp.1 rvMmuOp.5

rvMmuOp.8 rvMmuOp.2

rvMmuOp.3 rvMmuOp.9

Memory Map

Memory Values

Test Source

Sv39 mode + root PPN

L0 PTE with fault

Predicted PTE Traversal

TLB Flush instruction

Store to virtual address

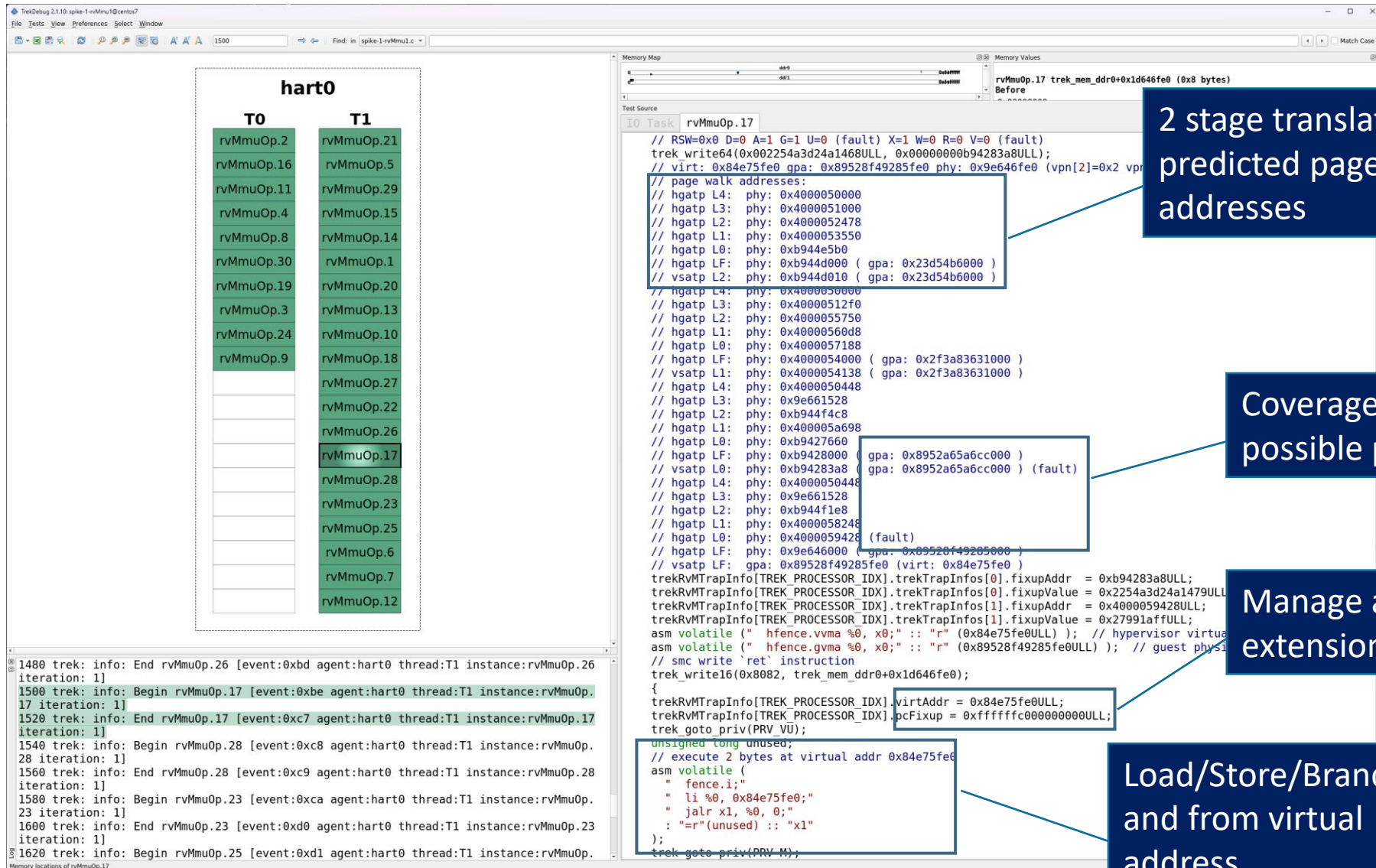
Self-Checking Test

```

trek write64(0x00000000221803cfULL, 0x00000000afad4218ULL);
// virt: 0x88600000 phy: 0x88600000 (vpn[2]=0x2 vpn[1]=0x43 vpn[0]=0x5e)
// == end unity mapping vaml regions ==
// satp MODE=Sv39 ASID=0 PPN=0x000000009bd64000
write csr(satp, 0x80000000009bd640);
// satp L2: 0x9bd64000[ 0xd4 ] = 0xafad5000
// RSW=0x3 D=0 A=0 G=1 U=0 X=0 W=0 R=0 (non-leaf) V=1
trek write64(0x000000002beb5721ULL, 0x000000009bd646a0ULL);
// satp L1: 0xafad5000[ 0xb4 ] = 0xafad6000
// RSW=0x2 D=0 A=0 G=0 U=0 X=0 W=0 R=0 (non-leaf) V=1
trek write64(0x000000002beb5a01ULL, 0x00000000afad5a01ULL);
// satp L0: 0xafad6000[ 0x5e ] = 0xafad3000 [leaf]
// RSW=0x0 D=0 (fault) A=1 G=0 U=1 X=0 W=0 (fault) R=0 (fault) V=0 (fault)
trek write64(0x000000002beb4c50ULL, 0x00000000afad62f0ULL);
// virt: 0x351685e008 phy: 0xafad3008 (vpn[2]=0xd4 vpn[1]=0xb4 vpn[0]=0x5e)
// page walk addresses:
// satp L2: phy: 0x9bd646a0
// satp L1: phy: 0xafad55a0
// satp L0: phy: 0xafad62f0 (fault)
// satp LF: phy: 0xafad3008 (virt: 0x351685e008)
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfo[0].fixupAddr = 0xafad62f0ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfo[0].fixupValue = 0x2beb4c50ULL;
asm volatile (" sfence.vma %0, x0; " :: "r" (0x351685e008ULL) ); // virtual address

trekRvMTrapInfo[TREK_PROCESSOR_IDX].virtAddr = 0x351685e008ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].pcFixup = 0xffffffff00000000ULL;
trek goto priv(PRIV_S);
unsigned long epc_unused1, unused2;
// write 8 bytes at virtual addr 0x351685e008 = 0xcc83e360d5c25710
asm volatile ("
    la %0, lf;
    li %1, 0xcc83e360d5c25710;
    li %2, 0x351685e008;
    sd %1, 0(%2);
    "=r"(epc_unused1), "=r"(unused2)
);
trek goto priv(PRIV_M);
trekRvMTrapInfo[TREK_PROCESSOR_IDX].pcFixup = 0;
if (!(trekRvMTrapInfo[TREK_PROCESSOR_IDX].count == 0x0000000000000001)) {
    trek c2t_arg(6, 0x0000000000000001);
    trek c2t_arg(0, trekRvMTrapInfo[TREK_PROCESSOR_IDX].count);
    trek c2t_event(6, 0x5d); // [event:0x5d agent:hart3 thread:T0 instance:rvMmuOp.3]
    
```


MMU Hypervisor Two-Stage Address Translation Example



hart0

T0	T1
rvMmuOp.2	rvMmuOp.21
rvMmuOp.16	rvMmuOp.5
rvMmuOp.11	rvMmuOp.29
rvMmuOp.4	rvMmuOp.15
rvMmuOp.8	rvMmuOp.14
rvMmuOp.30	rvMmuOp.1
rvMmuOp.19	rvMmuOp.20
rvMmuOp.3	rvMmuOp.13
rvMmuOp.24	rvMmuOp.10
rvMmuOp.9	rvMmuOp.18
	rvMmuOp.27
	rvMmuOp.22
	rvMmuOp.26
	rvMmuOp.17
	rvMmuOp.28
	rvMmuOp.23
	rvMmuOp.25
	rvMmuOp.6
	rvMmuOp.7
	rvMmuOp.12

Test Source

```
// RSW=0x0 D=0 A=1 G=1 U=0 (fault) X=1 W=0 R=0 V=0 (fault)
trek_write64(0x002254a3d24a1468ULL, 0x00000000b94283a8ULL);
// virt: 0x84e75fe0 gpa: 0x89528f49285fe0 phy: 0x9e646fe0 (vpn[2]=0x2 vpr=0)
// page walk addresses:
// hgap L4: phy: 0x4000050000
// hgap L3: phy: 0x4000051000
// hgap L2: phy: 0x4000052478
// hgap L1: phy: 0x4000053550
// hgap L0: phy: 0xb944e5b0
// hgap LF: phy: 0xb944d000 ( gpa: 0x23d54b6000 )
// vsatp L2: phy: 0xb944d010 ( gpa: 0x23d54b6000 )
// hgap L4: phy: 0x4000050000
// hgap L3: phy: 0x40000512f0
// hgap L2: phy: 0x4000055750
// hgap L1: phy: 0x40000560d8
// hgap L0: phy: 0x4000057188
// hgap LF: phy: 0x4000054000 ( gpa: 0x2f3a83631000 )
// vsatp L1: phy: 0x4000054138 ( gpa: 0x2f3a83631000 )
// hgap L4: phy: 0x4000050448
// hgap L3: phy: 0x9e661528
// hgap L2: phy: 0xb944f4c8
// hgap L1: phy: 0x400005a698
// hgap L0: phy: 0xb9427660
// hgap LF: phy: 0xb9428000 ( gpa: 0x8952a65a6cc000 )
// vsatp L0: phy: 0xb94283a8 ( gpa: 0x8952a65a6cc000 ) (fault)
// hgap L4: phy: 0x4000050448
// hgap L3: phy: 0x9e661528
// hgap L2: phy: 0xb944f1e8
// hgap L1: phy: 0x4000058248
// hgap L0: phy: 0x4000059428 (fault)
// hgap LF: phy: 0x9e646000 ( gpa: 0x89528f49285fe0 )
// vsatp LF: gpa: 0x89528f49285fe0 (virt: 0x84e75fe0)
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfos[0].fixupAddr = 0xb94283a8ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfos[0].fixupValue = 0x2254a3d24a1479ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfos[1].fixupAddr = 0x4000059428ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].trekTrapInfos[1].fixupValue = 0x27991affULL;
asm volatile (" hfence.vma %0, x0;" :: "r" (0x84e75fe0ULL) ); // hypervisor virtual
asm volatile (" hfence.gvma %0, x0;" :: "r" (0x89528f49285fe0ULL) ); // guest physical
// smc write `ret` instruction
trek_write16(0x8082, trek_mem_ddr0+0x1d646fe0);
{
trekRvMTrapInfo[TREK_PROCESSOR_IDX].virtAddr = 0x84e75fe0ULL;
trekRvMTrapInfo[TREK_PROCESSOR_IDX].pcFixup = 0xffffffff00000000ULL;
trek_goto_priv(PRV_VU);
}
// execute 2 bytes at virtual addr 0x84e75fe0
asm volatile (
" fence.i;"
" li %0, 0x84e75fe0;"
" jalr x1, %0, 0;"
: "=r"(unused) :: "x1"
);
trek_goto_priv(PRV_M);
```

Memory Map

Memory Values

rvMmuOp.17 trek_mem_ddr0+0x1d646fe0 (0x8 bytes)

Before

2 stage translation predicted page walk addresses

Coverage of all possible page faults

Manage address sign extension

Load/Store/Branch to and from virtual address

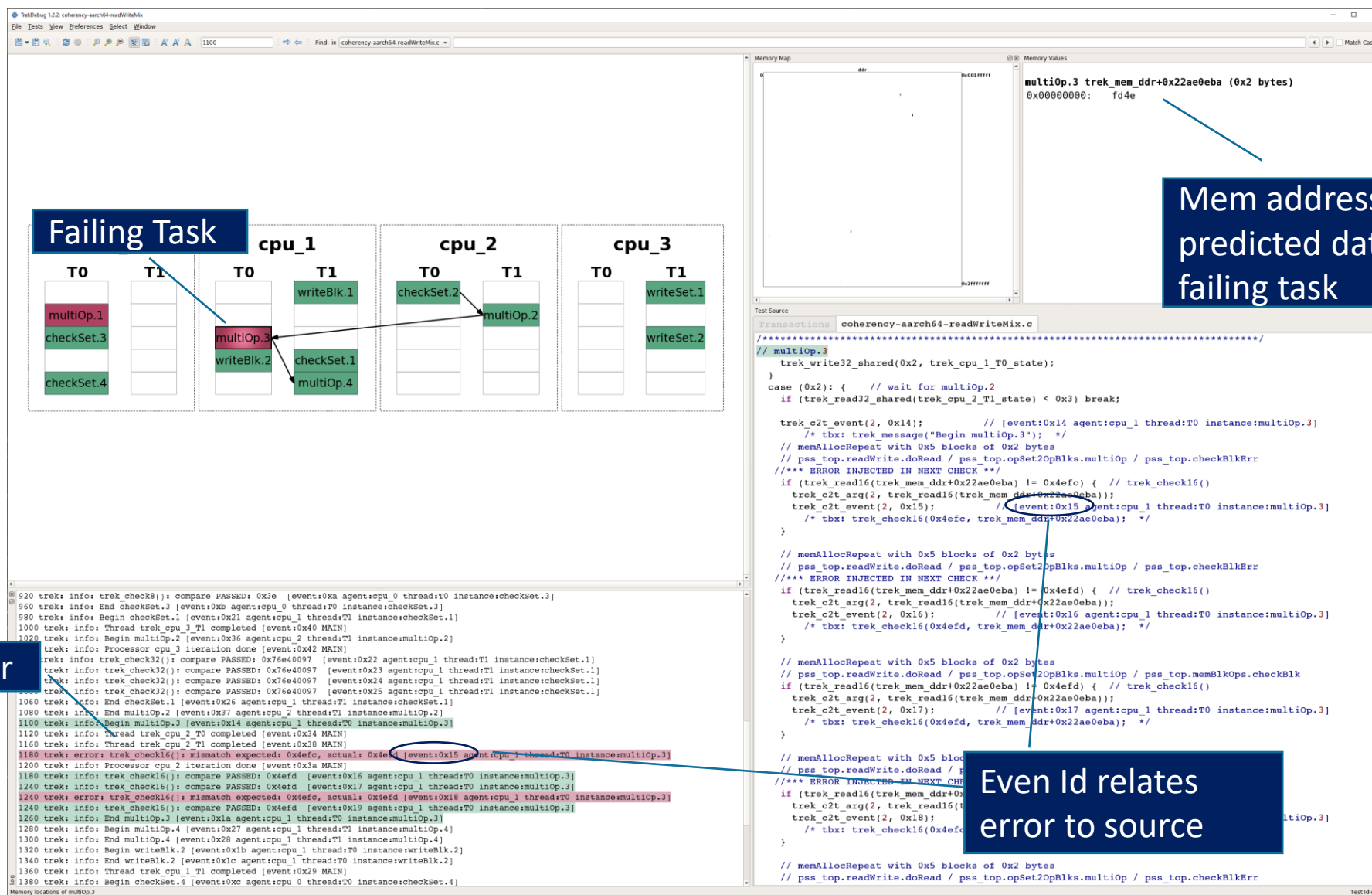
Example Customer Bugs and ISA Misinterpretations

- Guest Page Fault vs (Host) Page Fault **mcause** for Hypervisor
- Code Prefetch past end of mapped page
- Request to support Sv39 for G-stage translation
- Failing to take page fault for bad PTE Flags

Agenda

- Why do we need MMUs ?
- Virtual Memory and Page Table Walks
- RISC-V ISA MMU & Hypervisor Specification
- RISC-V MMU & Hypervisor Test Plan
- Example MMU Test Cases
- Debug, Coverage and Deployment

Debug Failing Test



Failing Task

The task diagram shows three CPUs (cpu_1, cpu_2, cpu_3) with tasks T0 and T1. In cpu_1, T0 is multiOp.1 and T1 is multiOp.3. In cpu_2, T0 is checkSet.2 and T1 is multiOp.2. In cpu_3, T0 is writeSet.1 and T1 is writeSet.2. Arrows indicate dependencies between tasks across CPUs.

Failing Error

The transaction log shows a mismatch error at line 1180: `trek_check16(): mismatch expected: 0x4efc, actual: 0x4ef4 [event:0x15 agent:cpu_1 thread:T0 instance:multiOp.3]`. The error is highlighted in red.

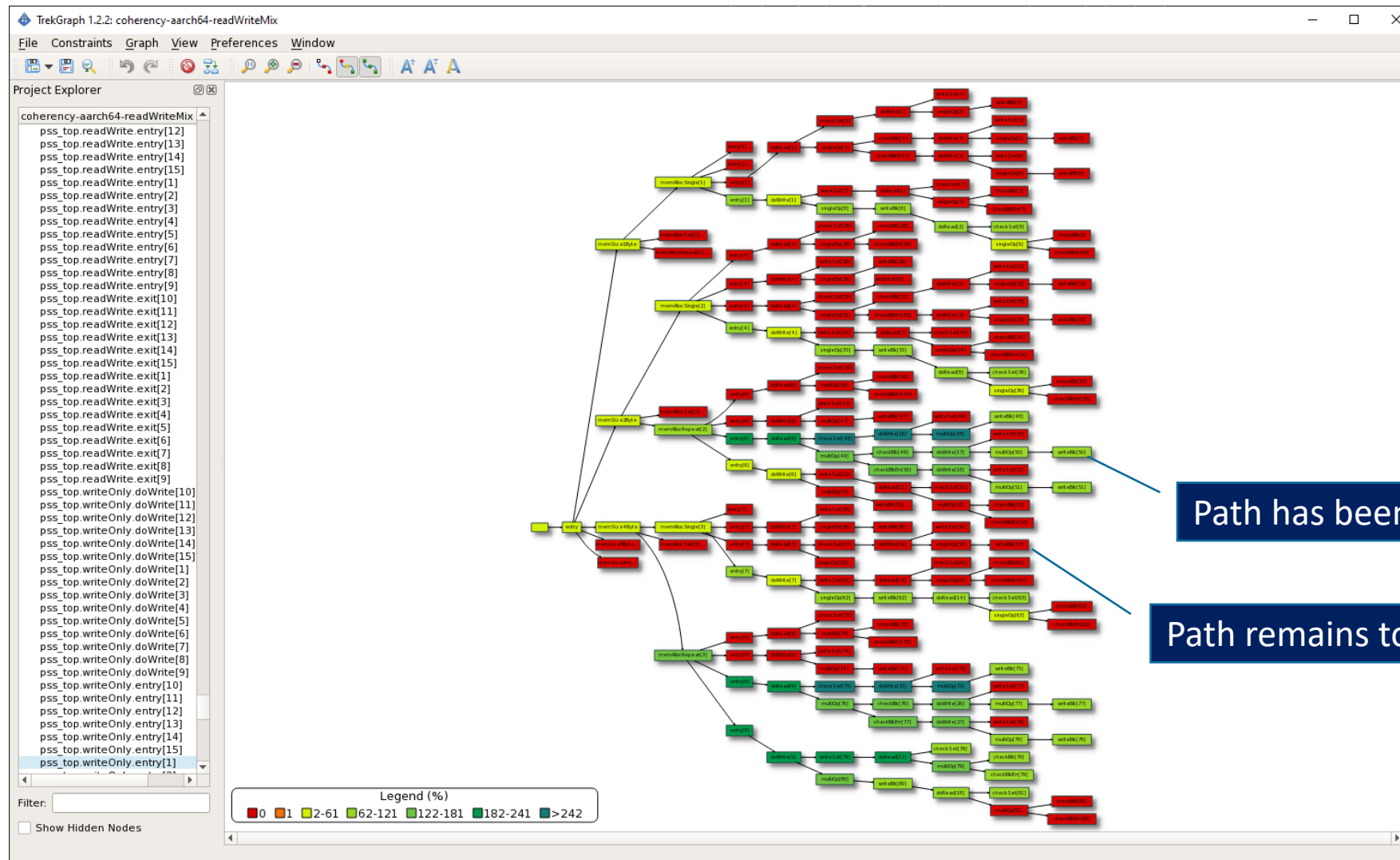
Mem address and predicted data used by failing task

The memory map shows the address `multiOp.3 trek_mem_ddr+0x22ae0eba (0x2 bytes)` with a predicted value of `0x00000000: fd4e`.

Even Id relates error to source

The transaction log shows the source code for the failing task, multiOp.3, which includes a memory allocation and a check for a specific value (0x4efc) at the memory address `trek_mem_ddr+0x22ae0eba`.

Analyze & Close Scenario Model Coverage



ISA Specs, Test Plans and Coverage

3.1. Supervisor CSRs | Page 57

sufficiently inexpensive to implement that we consider it worth supporting even if only rarely enabled.

3.2. Supervisor Instructions | Page 69

provided.

3.2.1. Sv32: Page-Based 32-bit Virtual-Memory Systems | Page 64

31 22 21 12 11 0

VPN[1] VPN[0] page offset

10 10 12

Figure 17. Sv32 virtual address.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the satp register.

33 22 21 12 11 0

PPN[1] PPN[0] page offset

12 10 12

Figure 18. Sv32 physical address.

31 20 19 10 9 8 7 6 5 4 3 2 1 0

PPN[1] PPN[0] RSW D A G U X W R V

12 10 2 1 1 1 1 1 1 1 1

Figure 19. Sv32 page table entry.

The PTE format for Sv32 is shown in Sv32 page table entry. The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. Encoding of PTE R/W/X fields. summarizes the encoding of the permission bits.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	Reserved for future use.
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	Reserved for future use.
1	1	1	Read-write-execute page.

Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception.

i AMOs never raise load page-fault exceptions. Since any unreadable page is also unwritable, attempting to perform an AMO on an unreadable page always raises a store page-fault exception.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the sstatus register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which

The RISC-V Instruction Set Manual: Volume II | © RISC-V International

ISA Specification

1.1. MMU & Hypervisor Page Translation | Page 2

Chapter 1. RISC-V ISA Verification Plan

1.1. MMU & Hypervisor Page Translation | Page 3

• Select operation

1.1.1. MMU & Hypervisor Page Translation | Page 4

• Check expected page faults for mcause CAUSE_LOAD[_GUEST_]PAGE_FAULT

1.1.2. Do store operation

IsaRef: link

CvgRef: doStore

• Allocate virtual address

- require PTE flags D, A, W, R and V set
- if (U or VU mode) require PTE flag U set

• Store to allocated virtual address and update memory scoreboard

• Check expected page faults for mcause CAUSE_STORE[_GUEST_]PAGE_FAULT

NOTE: need both PTE.W and PTE.R — no write-only PTE flag configuration

1.1.3. Do execute operation

IsaRef: link

CvgRef: doExec

• Allocate virtual address

- with PTE flags A, X and V set
- if (U or VU mode) require PTE flag U set

• Jump to allocated virtual address

• Check expected page faults for mcause CAUSE_FETCH[_GUEST_]PAGE_FAULT

1.1.3. Setup address translation

1.1.3.1. Setup one-stage address translation

IsaRef: link

CvgRef: makePageMapSatp

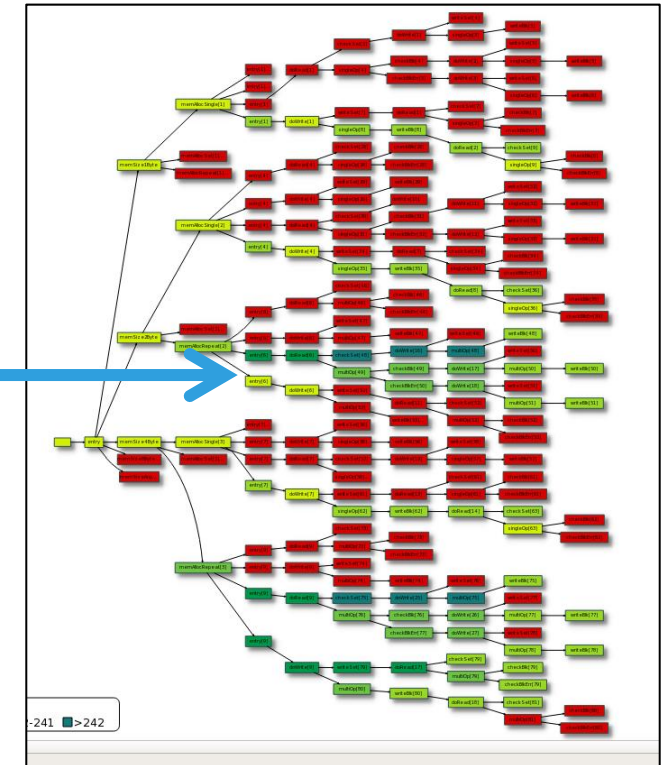
Do

- Select one-stage paging mode
- Allocate 4KB naturally aligned root page
- Page map code stack and code addresses
- Write satp with root page table address and mode

TODO: randomize A150 mapping

The RISC-V Instruction Set Manual: Volume II | © RISC-V International

RISCV Test Plan



Coverage Reports

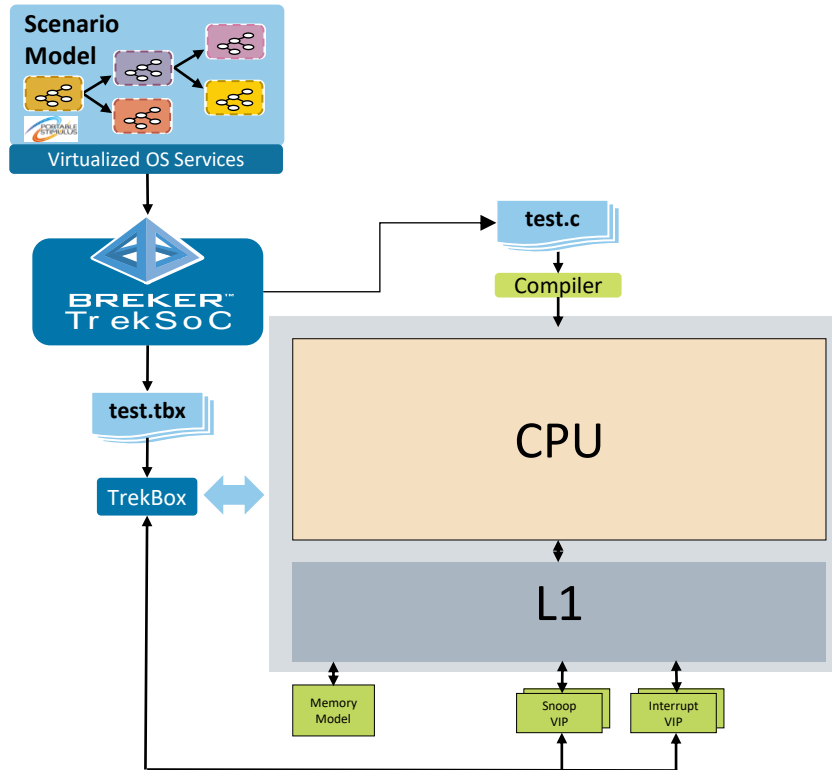
RVA23 Profile Coverage Roadmap

Extension	Description	Available
I	RV64I is the mandatory base ISA for RVA23U64.	Yes
M	Integer multiplication and division.	Yes
A	Atomic instructions.	Yes
F	Single-precision floating-point instructions.	4Q24
D	Double-precision floating-point instructions.	4Q24
C	Compressed instructions.	Yes
Zicsr	CSR instructions.	Yes
Zicntr	Base counters and timers.	1Q25
Zihpm	Hardware performance counters.	3Q25
Ziccif	Coherence PMAs must support instruction fetch.	Yes
Ziccrse	Coherence PMAs must support RsrVEventual.	4Q24
Ziccamoa	Coherence PMAs must support all atomics.	Yes
Zicclsm	Misaligned loads and stores.	Yes

Extension	Description	Available
Za64rs	Reservation sets 64B, contiguous, aligned.	4Q24
Zihintpause	Pause hint.	1Q25
Zba	Address generation.	Yes
Zbb	Basic bit-manipulation.	Yes
Zbs	Single-bit instructions.	Yes
Zic64b	Cache blocks must be 64 bytes.	Yes
Zicbom	Cache-block management instructions.	Yes
Zicbop	Cache-block prefetch instructions.	1Q25
Zicboz	Cache-Block Zero Instructions.	1Q25
Zfhmin	Half-precision floating-point.	4Q24
Zkt	Data-independent execution latency.	3Q25
V	Vector extension.	4Q24
Zvfhmin	Vector minimal half-precision floating-point.	4Q24
Zvbb	Vector basic bit-manipulation instructions.	4Q24
Zvkt	Vector data-independent execution latency.	4Q24
Zihintntl	Non-temporal locality hints.	1Q25
Zicond	Integer conditional operations.	4Q24
Zimop	may-be-operations.	2Q25
Zcmop	Compressed may-be-operations.	2Q25
Zcb	Additional compressed instructions.	Yes
Zfa	Additional floating-Point instructions.	4Q24
Zawrs	Wait-on-reservation-set instructions.	3Q25
Supm	Pointer masking.	3Q25
Zvkng	Vector crypto NIST algorithms with GCM.	3Q25
Zvksg	Vector crypto ShangMi algorithms with GCM.	3Q25
Zabha	Byte and halfword atomic memory operations.	2Q25
Zacas	Compare-and-Swap instructions.	2Q25
Ziccamoc	Coherence PMAs must provide AMOCASQ.	2Q25
Zvbc	Vector carryless multiplication.	2Q25
Zama16b	Misaligned loads, stores, and AMOs are atomic.	Yes
Zfh	Scalar half-precision floating-point.	2Q25
Zbc	Scalar carryless multiply.	Yes
Zicfilp	Landing Pads.	3Q25
Zicfiss	Shadow Stack.	3Q25
Zvfh	Vector half-precision floating-point.	3Q25
Zfbfmin	Scalar BF16 converts.	3Q25
Zvfbfmin	Vector BF16 converts.	3Q25
Zvfbfwma	Vector BF16 widening mul-add.	3Q25
Zifencei	Instruction-Fetch Fence.	Yes
Ss1p13	Supervisor architecture version 1.13.	Yes
Svbare	The satp mode Bare must be supported.	Yes

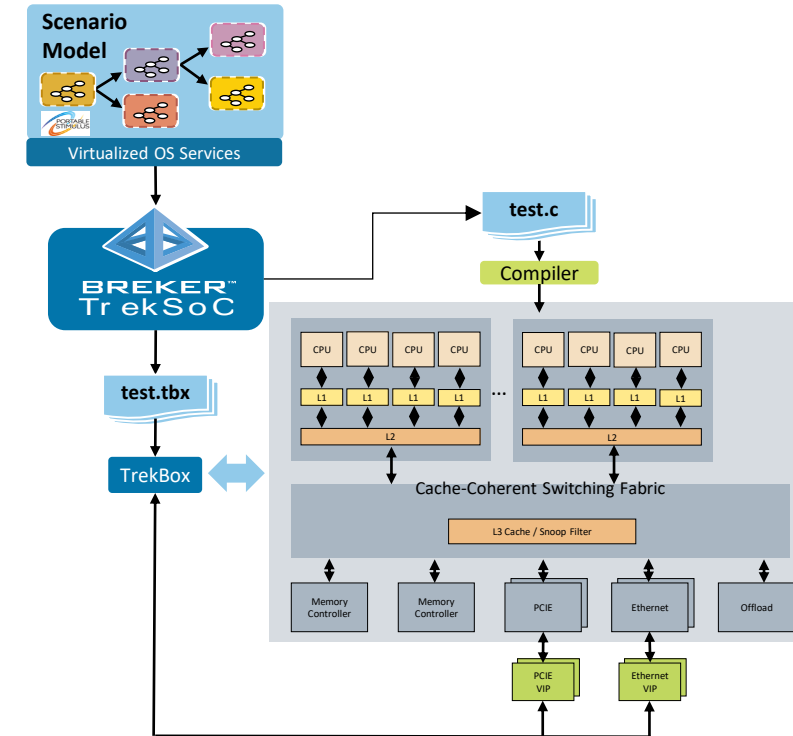
Extension	Description	Available
Sv39	Page-based 39-bit virtual-Memory system.	Yes
Svade	Page-fault exceptions for A and D bits.	Yes
Ssccptr	Coherence PMAs must support page-tables.	Yes
Sstvecd	stvec.MODE=Direct is supported.	1Q25
Sstvala	stval gets faulting virtual address.	1Q25
Sscounterenw	hpmcounter and scounteren.	3Q25
Svpbmt	Page-based memory types.	1Q25
Svinval	Fine-grained TLB invalidation.	1Q25
Svnapot	NAPOT translation contiguity.	2Q25
Sstc	supervisor-mode timer interrupts.	1Q25
Sscofpmf	count overflow and mode-based filtering.	3Q25
Ssnpm	Pointer masking for senvcfg.PME, henvcfg.PME.	3Q25
Ssu64xl	sstatus.UXL UXLEN=64.	Yes
H	The hypervisor extension.	Yes
Ssstateen	Supervisor-mode view of the state-enable extension.	3Q25
Shcounterenw	hpmcounter and hcounteren.	3Q25
Shvstvala	vstval gets faulting virtual address.	1Q25
Shtvala	htval gets faulting guest physical address.	1Q25
Shvstvecd	vstvec.MODE=Direct is supported.	1Q25
Shvsatpa	All translation modes in vsatp.	Yes
Shgatpa	hgatp SvNNx4 modes supported.	Yes
Sv48	Page-based 48-bit virtual-memory system.	Yes
Sv57	Page-based 57-bit virtual-memory system.	Yes
Zkr	Entropy CSR.	3Q25
Svadu	Hardware A/D bit updates.	1Q25
Sdtrig	Debug triggers.	3Q25
Ssstrict	No non-conforming extensions.	3Q25
Svvptc	Invalid to valid PTEs w/o fence.	1Q25
Sspm	Supervisor-mode pointer masking.	3Q25

Single Hart vs Multi-Hart + SoC



Testbench control needed for

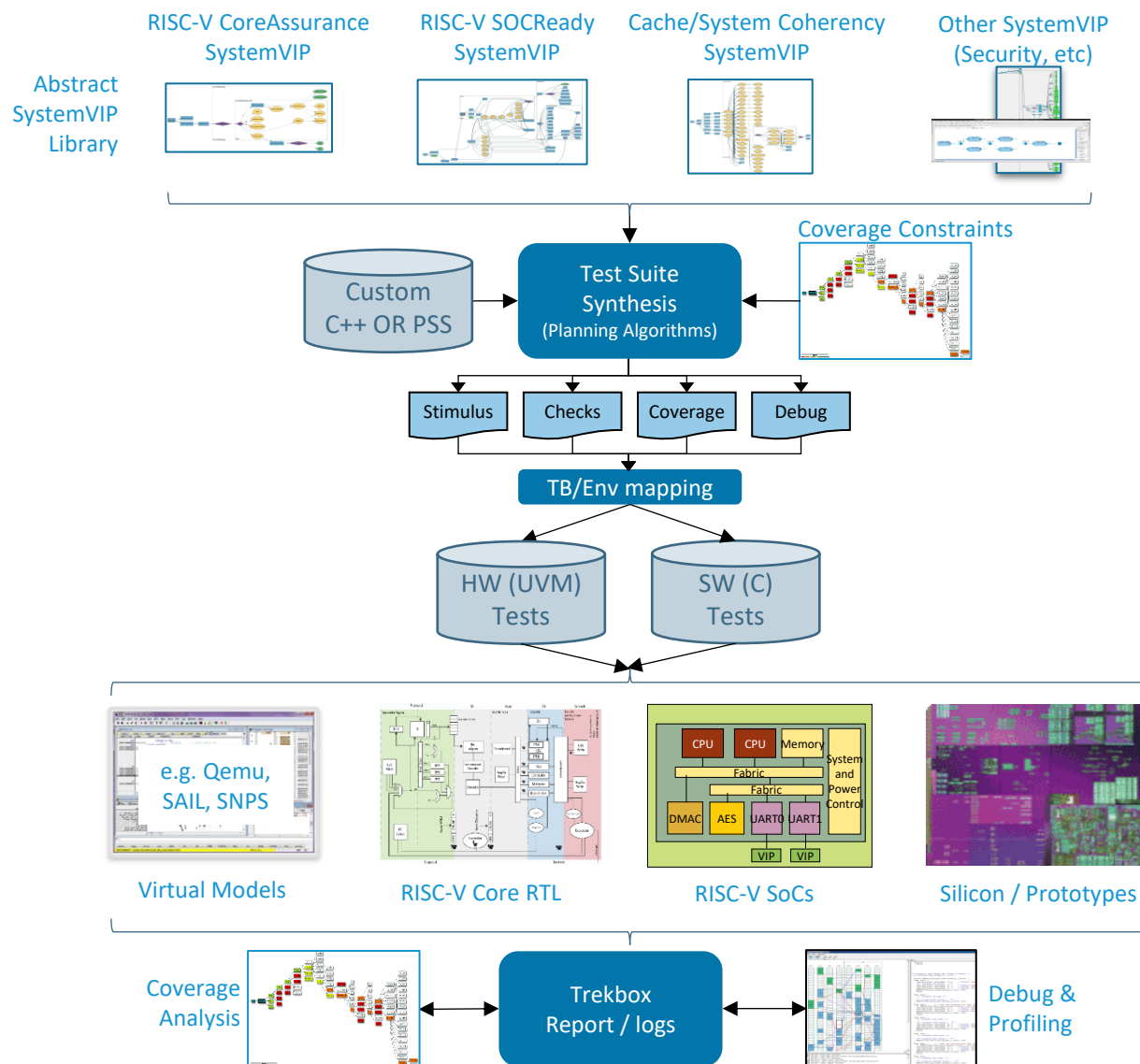
- External Interrupts,
- IOMMU,
- Debug Extension



Multi-core tests needed for

- AMO/atomics
- CMO/cache flush, invalidate
- RVWMO Memory Ordering

Breker Technology Overview for RISC-V



- EDA Test Generation Tool company
- RISC-V Core and SoC System VIP products
- 17+ Commercial RISC-V Deployments
 - 4+ open-source RISC-V deployments
- Support organization of RISC-V SME's
 - Help customers understand spec
 - Issue tracking, escalation, etc.
- Track Record:
 - 100% compliance + verification of x86_64
 - 10+ years on ARM SoC verification

Thanks for Listening!
Any Questions?
