2024
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES

SAN JOSE, CA, USA
MARCH 4-7, 2024

RISC-V Core Verification: A New Normal in Verification Techniques

Adnan Hamid, Breker Verification Systems
John Sotiropoulos, Breker Verification Systems

BREKER™

accellera SYSTEMS INITIATIVE

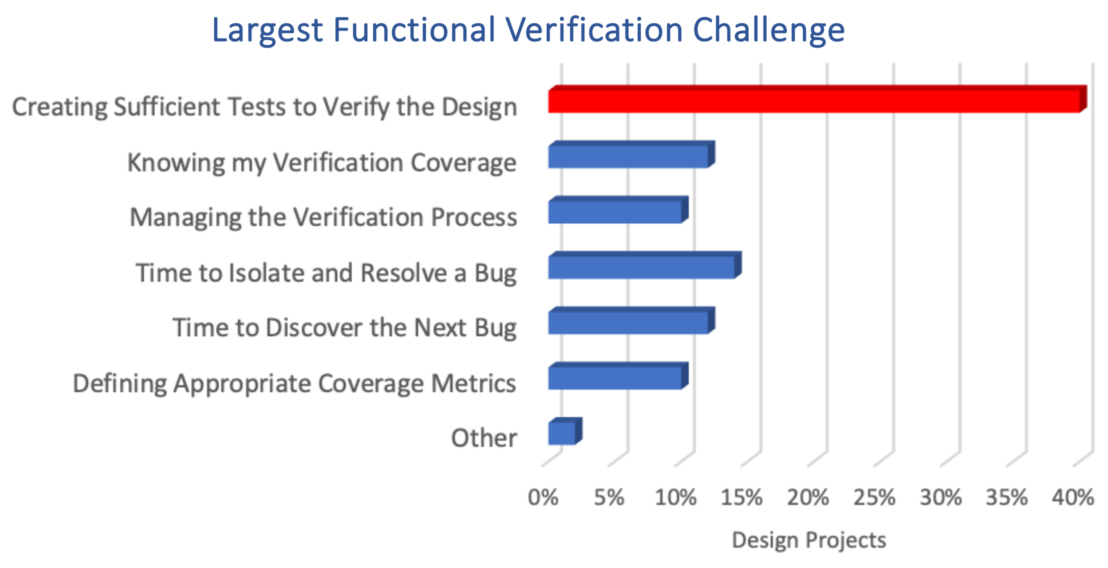
Agenda

- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

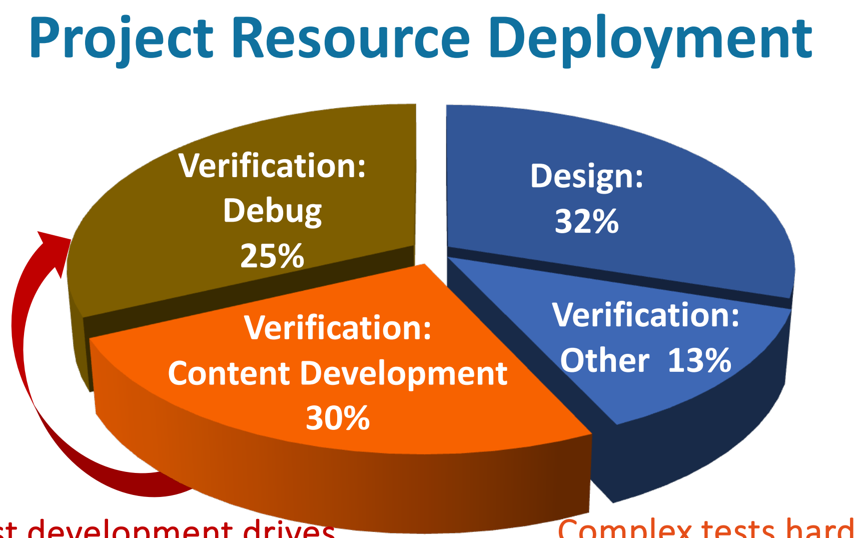
Agenda

- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

The High Cost of Developing Test Content



Source: Wilson Research 2020



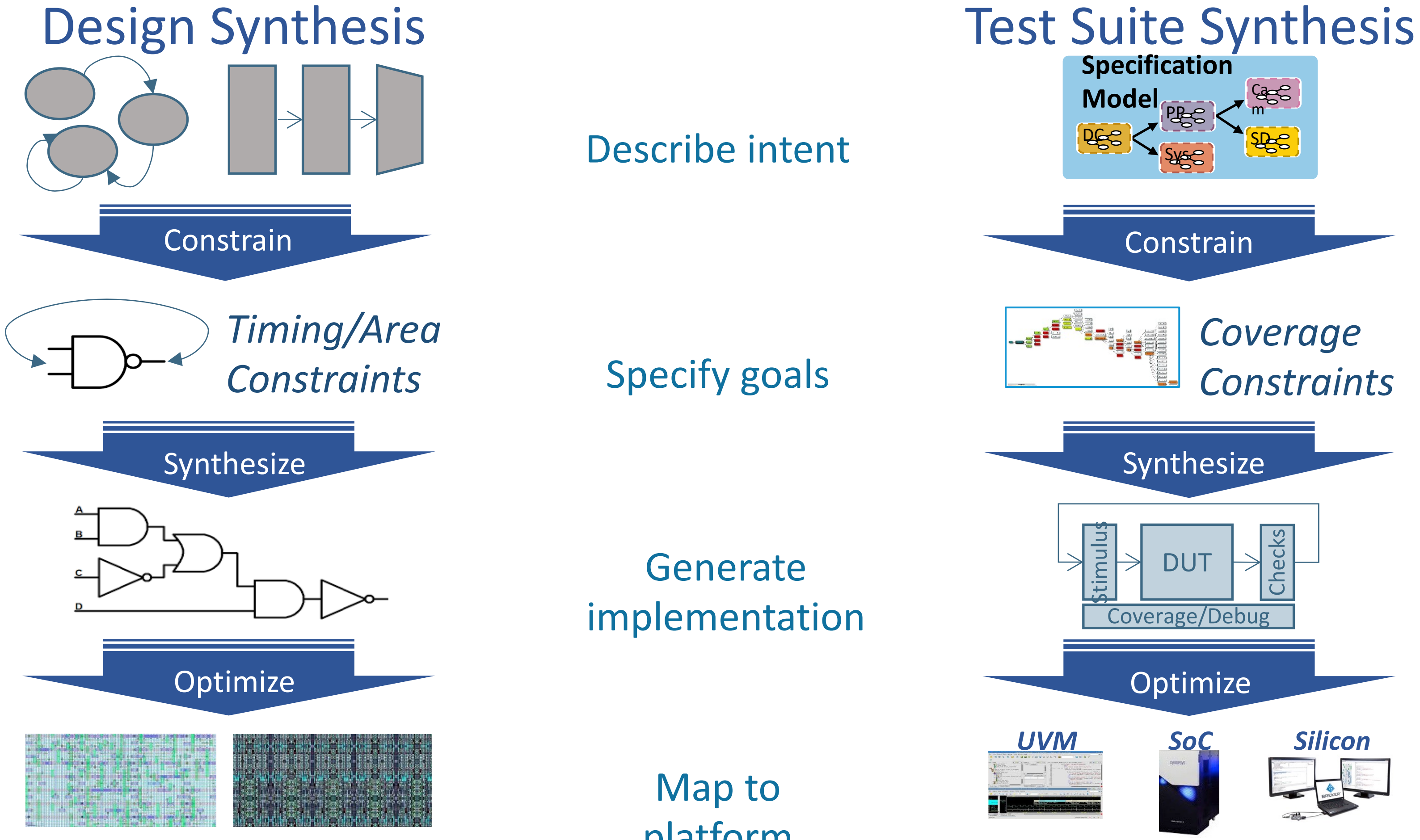
Test development drives debug

Complex tests hard to get right

A Look At RISC-V

- Open Instruction Set Architecture (ISA) creating a discontinuity in the market
- Appears to be gaining significant traction in multiple applications
- Significant verification challenges
  - Arm spends $150M per year on 10^{15} verification cycles per core
  - Hard for RISC-V development group to achieve this same quality
  - Lots of applications expands verification requirements
  - Requires automation, reuse and other new thinking

Test Suite Synthesis… Analogous to Logic Synthesis



Describe intent

Specify goals

Generate implementation

Map to platform

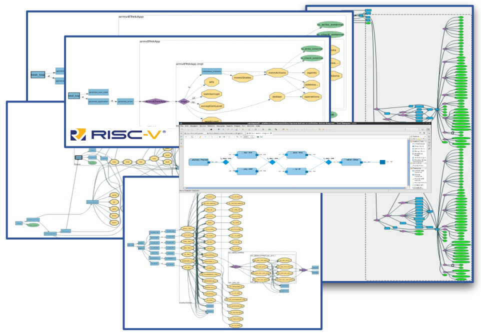
Breker Core Technology

- AI Planning Algorithms
- 3D Coverage Closure
- Synthesizable VerificationOS

AI Planning Algorithms

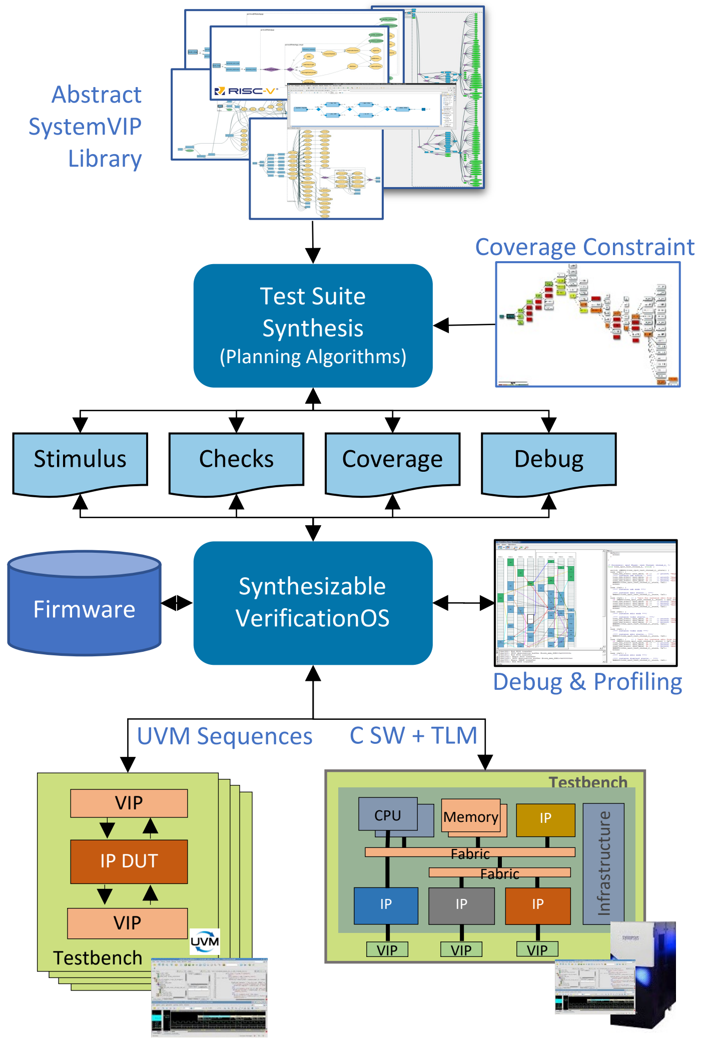
Breker Background: Test Suite Synthesis for RISC-V Cores & SoCs

- Breker is a key, longstanding part of the verification ecosystem for processors and SoCs based on x86 and Arm architectures
- Breker has become part of the verification ecosystem for processors and SoCs based on RISC-V architectures
  - Working with multiple RISC-V developers and users/integrators
- RISC-V has room to grow if we solve the verification barrier
  - We are experienced in x86 and Arm verification, allowing us to share this experience with RISC-V teams through automated tests

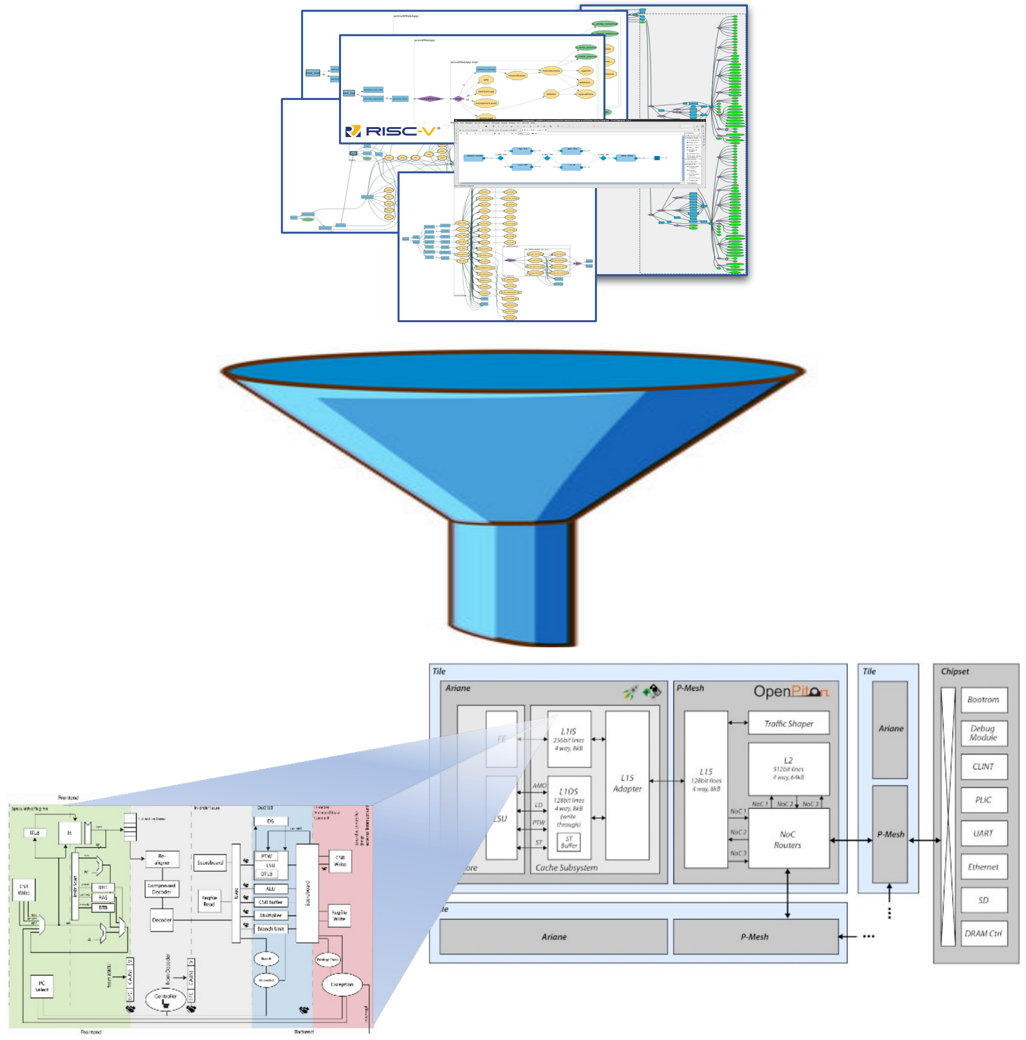


The Breker SystemVIP Library

- *Core Integrity FastApps*
- *RISC-V System Integrity TrekApp*
- *ARM System Integrity TrekApp*
- *Cache Coherency TrekApp 2.0*
- *Firmware-First TrekApp*
- *Power Management TrekApp*
- *Security TrekApp*
- *Networking TrekApp*



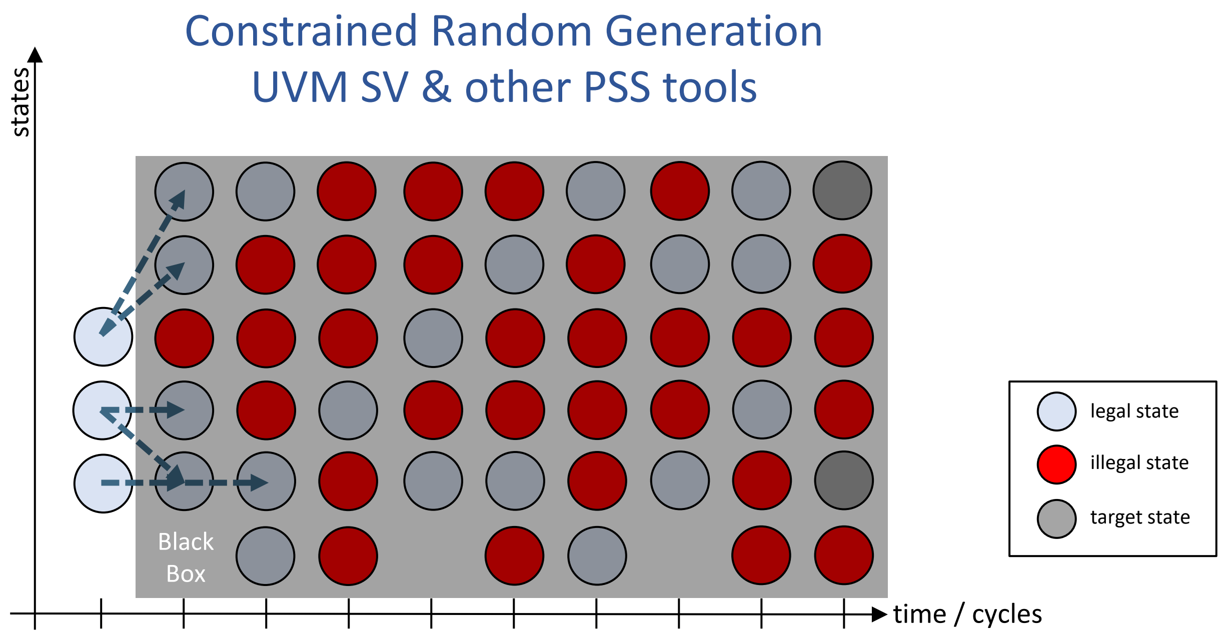
Breker SystemVIP Library

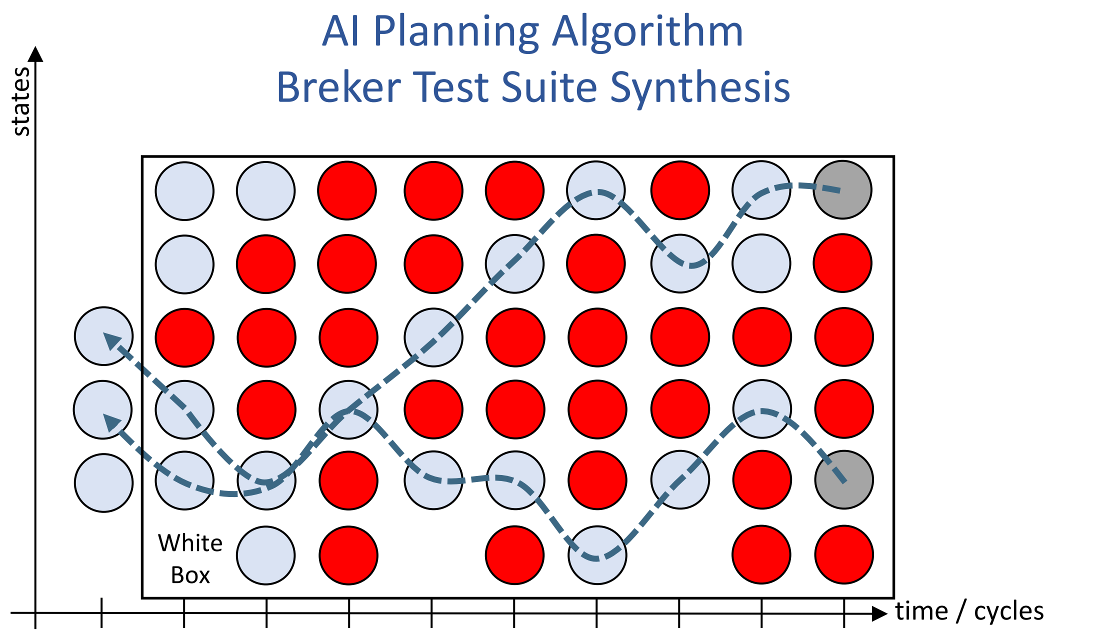


SoC SystemVIP Library

- The ***RISC-V Core TrekApp*** provides fast, pre-packaged tests for RISC-V Core and SoC integrity issues
- The ***Coherency TrekApp*** verifies cache and system-level coherency in a multiprocessor SoC
- The ***End-to-end IP TrekApp*** IP test sets ported from UVM to SoC
- The ***Power Management TrekApp*** automates power domain switching verification
- The ***Security TrekApp*** automates testing of hardware access rules for HRoT fabrics
- The ***Networking & Interface TrekApp*** automates packet generation, CXL, UCIe interface tests

Constrained Random vs AI Planning Algorithm Synthesis





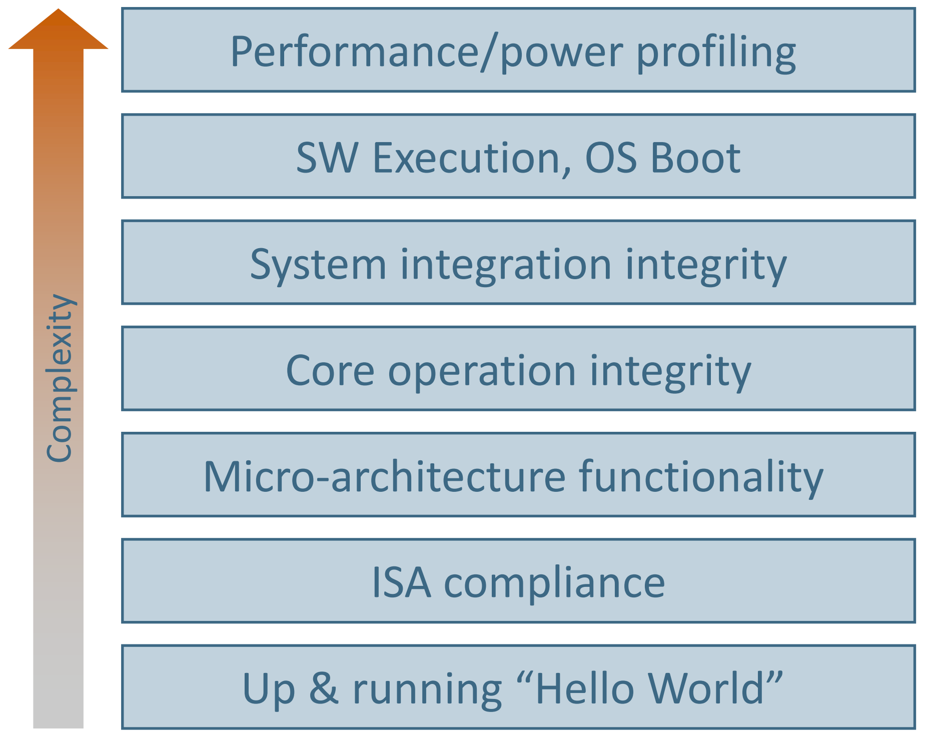
Design black box, shotgun tests to search for key state
Low probability of finding complex bug

Starts with key state and intelligently works backward through space
Deep sequential, optimized test discovers complex corner-cases

RISC-V Verification Challenges

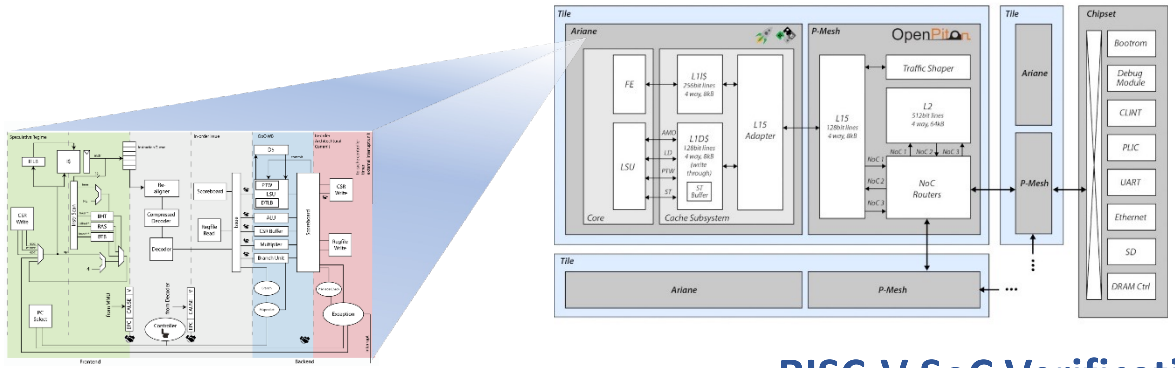
- Processors are hard to verify
  - Consider Arm and Intel verification investments
- Automation is the answer
  - Number of diversified test generators, etc.
- RISC-V special requirements
  - Custom instruction verification
  - Compliance assurance
  - Broad range of architectures
- Different processors have different needs
  - Embedded cores
  - Processor clusters
  - Application processors

Suggested RISC-V verification "stack"



Performance/power profiling
SW Execution, OS Boot
System integration integrity
Core operation integrity
Micro-architecture functionality
ISA compliance
Up & running "Hello World"
Complexity

Different Challenges for Core vs SoC Verification



RISC-V Core Verification Challenges

Random Instructions	Do instructions yield correct results
Register/Register Hazards	Pipeline perturbations dues to register conflicts
Load/Store Integrity	Memory conflict patterns
Conditionals and Branches	Pipeline perturbations from synchronous PC change
Exceptions	Jumping to and returning from ISR
Asynchronous Interrupts	Pipeline perturbations from asynchronous PC change
Privilege Level Switching	Context switching
Core Security	Register and Memory protection by privilege level
Core Paging/MMU	Memory virtualization and TLB operation
Sleep/Wakeup	State retention across WFI
Voltage/Freq Scaling	Operation at different clock ratios
Core Coherency	Caches, evictions and snoops

RISC-V SoC Verification Challenges

System Coherency	Cover all cache transitions, evictions, snoops
System Paging/IOMMU	System memory virtualization
System Security	Register and Memory protection across system
Power Management	System wide sleep/wakeup and voltage/freq scaling
Packet Generation	Generating networking packets for I/O testing
Interface Testing	Analyzing coherent interfaces including CXL & UCIe
Random Memory Tests	Test Cores/Fabrics/Memory controllers across DDR, OCRAM, FLASH etc
Random Register Tests	Read/write test to all uncore registers
System Interrupts	Randomized interrupts through CLINT
Multi-core Execution	Concurrent operations on fabric and memory
Memory Ordering	For weakly order memory protocols
Atomic Operation	Across all memory types

RISC-V Verification & Validation Tasks

Core Integrity

Complexity

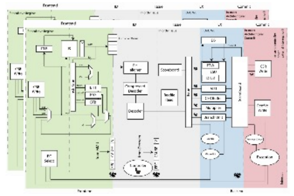
- Interrupts/Paging/Memory Order
- Micro-architecture functionality
- ISA compliance
- First Instruction Completion

IP Integrity

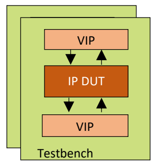
Complexity

- End-to-End Use Cases
- Concurrency Testing
- IP Configuration

CPU Cores



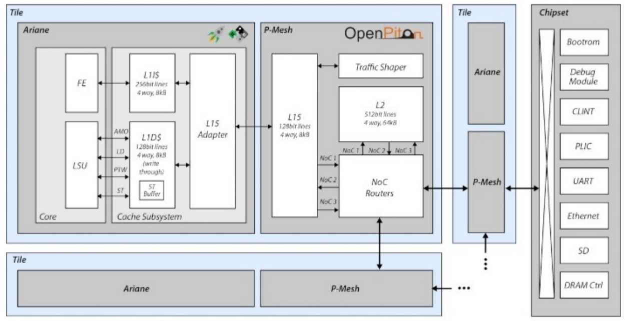
Uncore IPs



SoC Integrity

Complexity

- Power/Performance Profiling
- Security
- Power Management
- Interrupts/Paging/Memory Order
- Cache Coherency



Firmware

Firmware Integrity

Complexity

- End-to-End Use Cases
- Concurrency Testing
- HW/Firmware Compatibility

Single Source of Truth for all stages of Verification & Validation

SVIPs for IP Integrity

- Mem2Mem (dma)
- IO Offload (PCIE/Eth)
- WQ Servicing
- …

SVIPs for Core Integrity

- Register Hazards
- Load/Store
- Core Cache Coherency
- Core Interrupts
- …

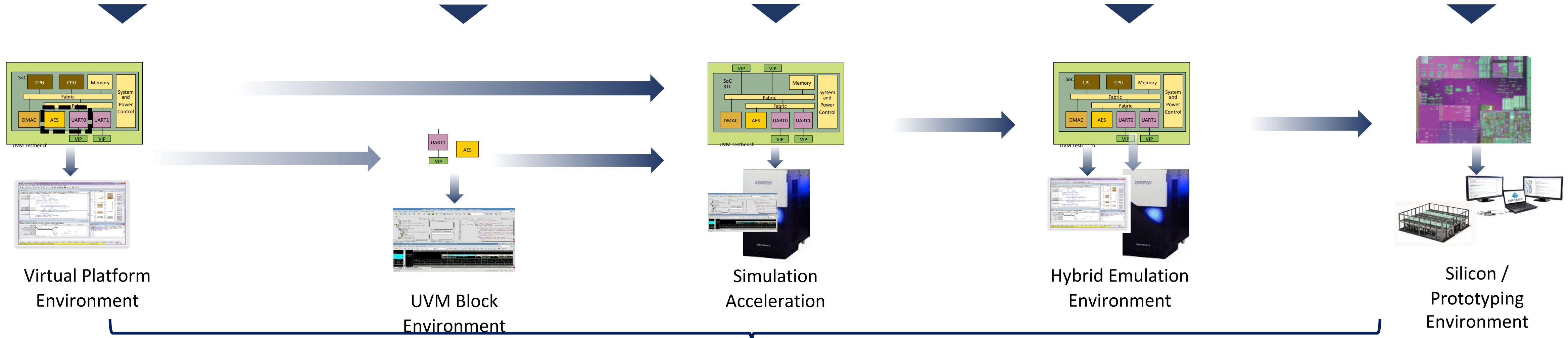
SVIPs for SoC Integrity

- SoC Cache Coherency
- Memory Ordering
- Power Management
- System Interrupts
- …

SVIPs for FW Integrity

- Mem2Mem (dma)
- IO Offload (PCIE/Eth)
- WQ Servicing
- …

Test Suite Synthesis



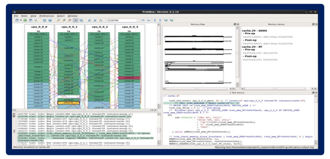
Virtual Platform Environment

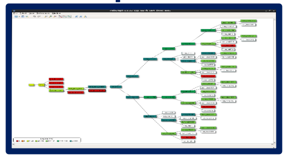
UVM Block Environment

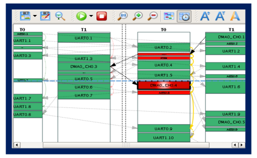
Simulation Acceleration

Hybrid Emulation Environment

Silicon / Prototyping Environment


High Level Debug


Coverage Analysis


Performance Profiling

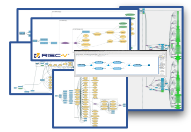
Agenda

- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

Core-Integrity Challenges

Random Instructions	Do instructions yield correct results
Register/Register Hazards	Pipeline perturbations dues to register conflicts
Load/Store Integrity	Memory conflict patterns
Conditionals and Branches	Pipeline perturbations from synchronous PC change
Exceptions	Jumping to and returning from ISR
Asynchronous Interrupts	Pipeline perturbations from asynchronous PC change
Privilege Level Switching	Context switching
Core Security	Register and Memory protection by privilege level
Core Paging/MMU	Memory virtualization and TLB operation
Sleep/Wakeup	State retention across WFI
Voltage/Freq Scaling	Operation at different clock ratios
Core Coherency	Caches, evictions and snoops

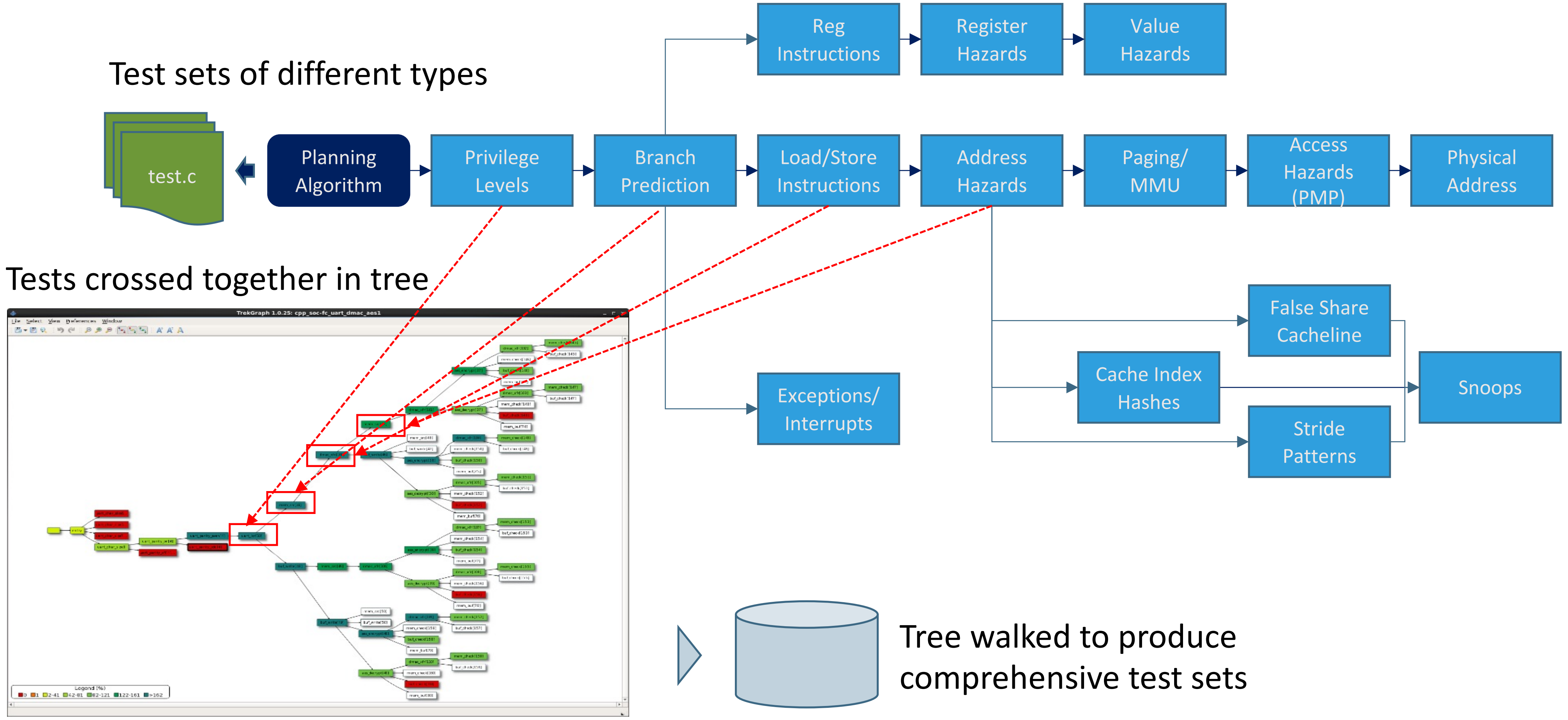
Breker
RISC-V Core-Integrity
FASTApps



Crossing RISC-V Core Verification Components

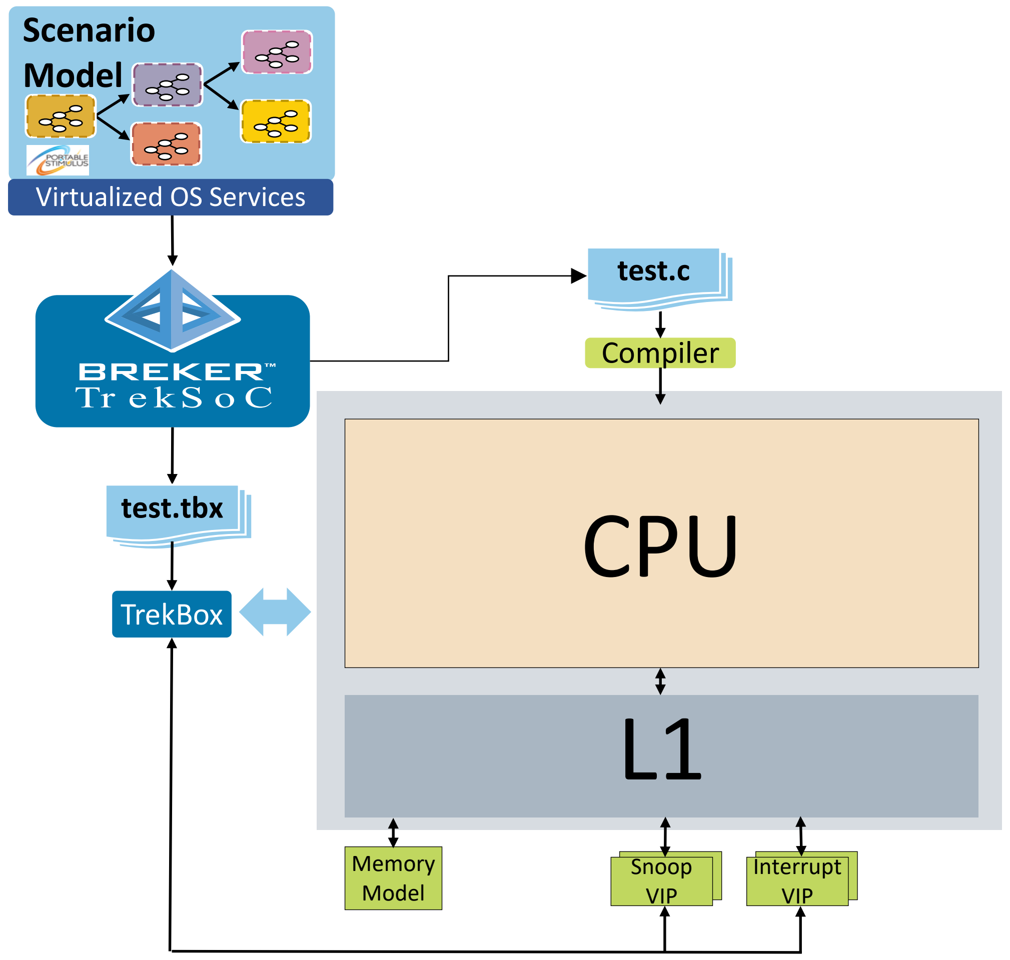
Test sets of different types

Tests crossed together in tree

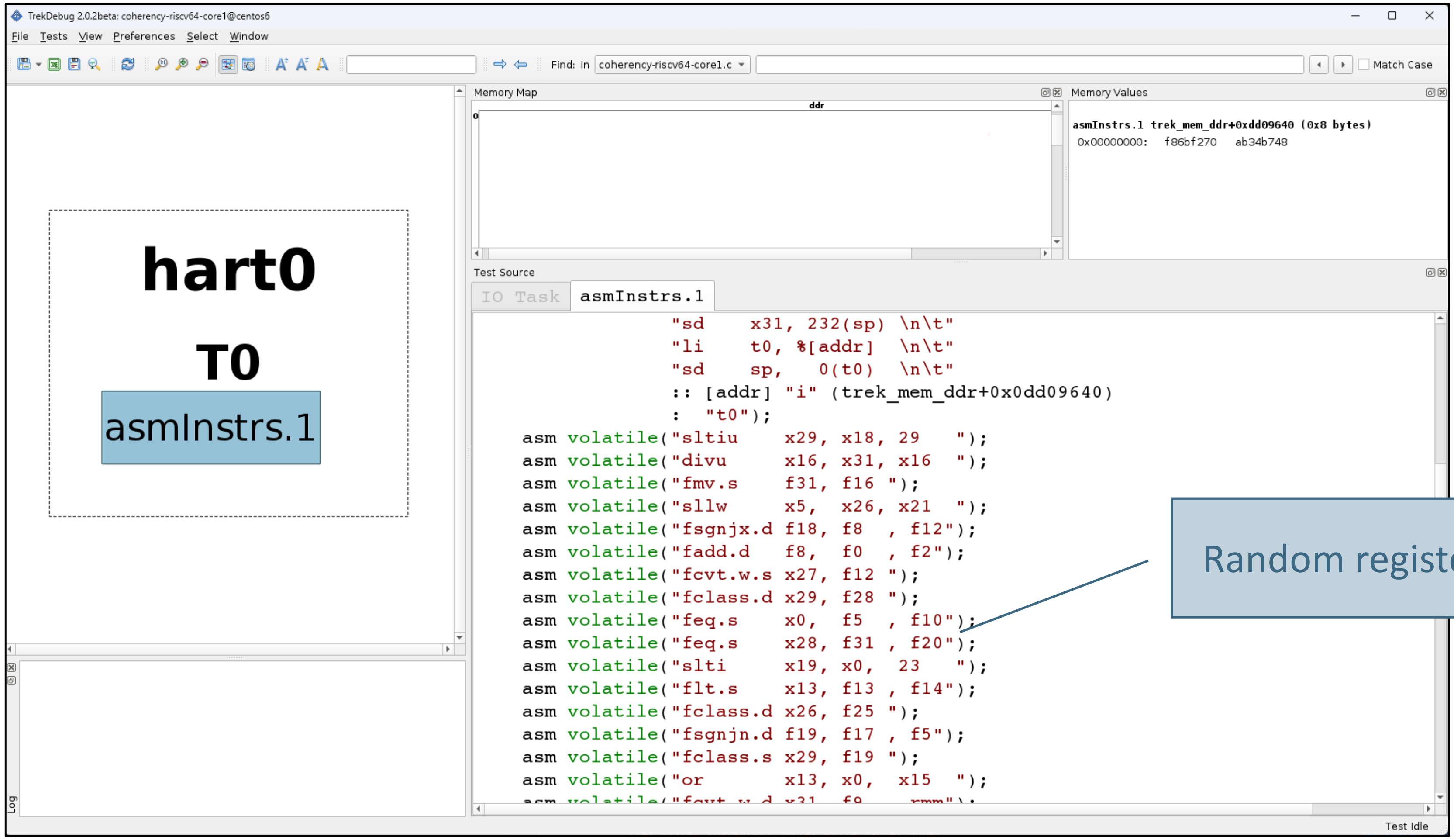


Tree walked to produce comprehensive test sets

RISC-V Core Testbench Integration



RV64 Core Instruction Generation

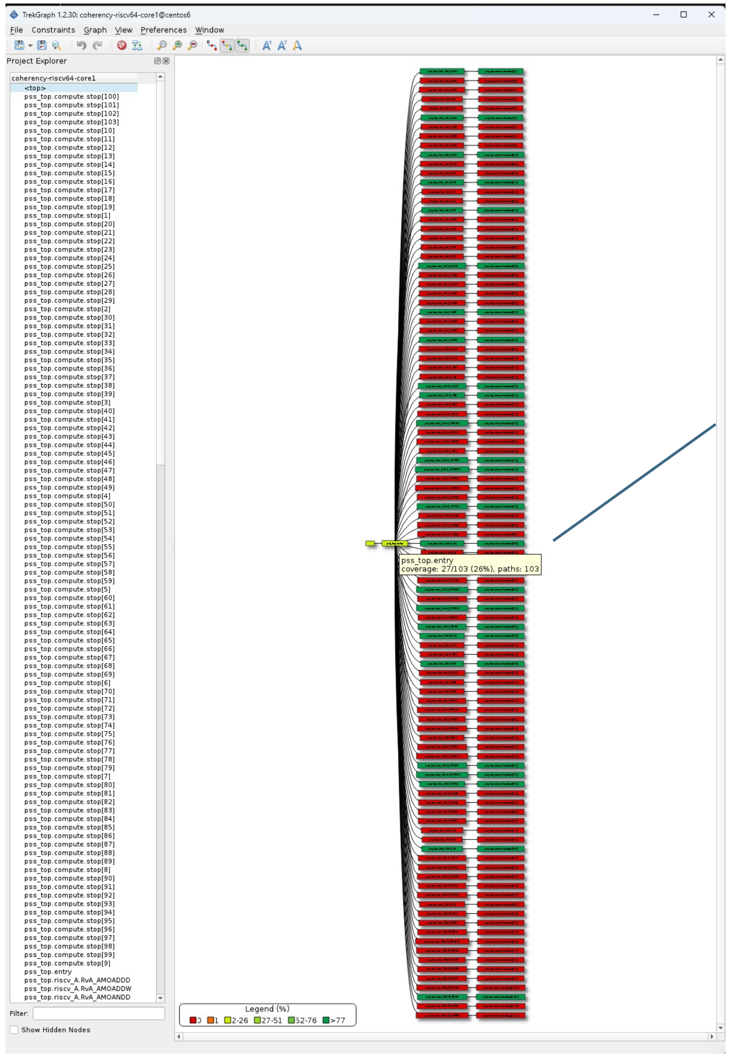

TrekDebug 2.0.2beta: coherency-riscv64-core1@centos6
File Tests View Preferences Select Window
Find: in coherency-riscv64-core1.c
Match Case
Memory Map
ddr
Memory Values
hart0
T0
asmInstrs.1
Test Source
IO Task
asmInstrs.1
Log
Test Idle

```
asmInstrs.1 trek_mem_ddr+0xdd09640 (0x8 bytes)
 0x00000000:  f86bf270   ab34b748
```

```
                   "sd    x31, 232(sp) \n\t"
                   "li    t0, %[addr]  \n\t"
                   "sd    sp,   0(t0)  \n\t"
                   :: [addr] "i" (trek_mem_ddr+0x0dd09640)
                   :  "t0");
    asm volatile("sltiu    x29, x18, 29   ");
    asm volatile("divu     x16, x31, x16  ");
    asm volatile("fmv.s    f31, f16 ");
    asm volatile("sllw     x5,  x26, x21  ");
    asm volatile("fsgnjx.d f18, f8  , f12");
    asm volatile("fadd.d   f8,  f0  , f2");
    asm volatile("fcvt.w.s x27, f12 ");
    asm volatile("fclass.d x29, f28 ");
    asm volatile("feq.s    x0,  f5  , f10");
    asm volatile("feq.s    x28, f31 , f20");
    asm volatile("slti     x19, x0,  23   ");
    asm volatile("flt.s    x13, f13 , f14");
    asm volatile("fclass.d x26, f25 ");
    asm volatile("fsgnjn.d f19, f17 , f5");
    asm volatile("fclass.s x29, f19 ");
    asm volatile("or       x13, x0,  x15  ");
```

Random register instructions

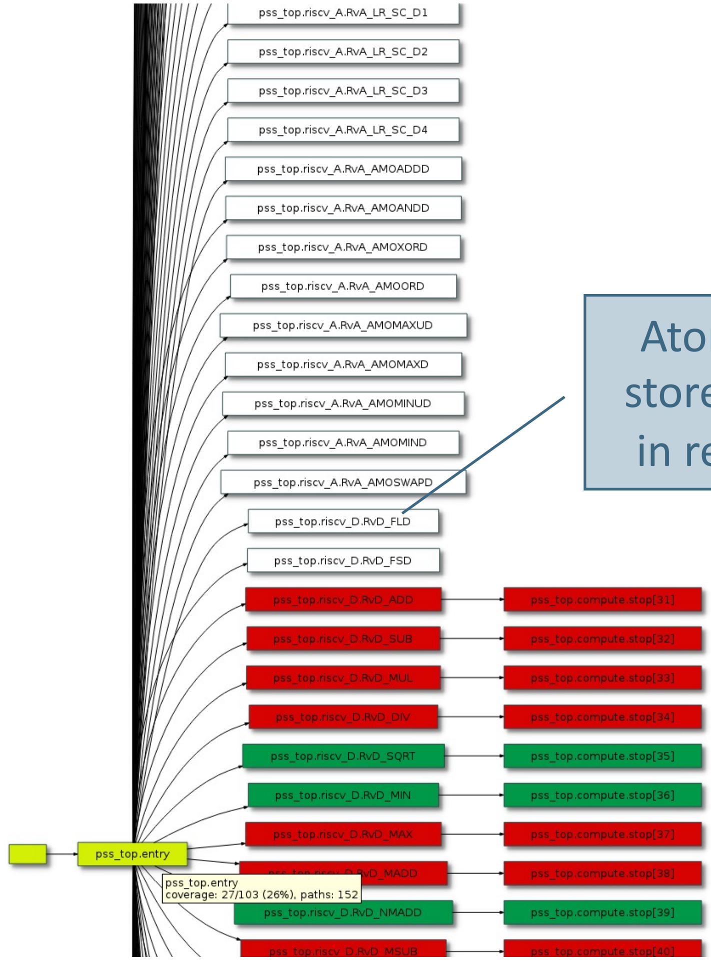
Instruction Coverage Analysis



```
coherency-riscv64-core1
    <top>
    pss_top.compute.stop[100]
    pss_top.compute.stop[101]
    pss_top.compute.stop[102]
    pss_top.compute.stop[103]
    pss_top.compute.stop[10]
    pss_top.compute.stop[11]
    pss_top.compute.stop[12]
    pss_top.compute.stop[13]
    pss_top.compute.stop[14]
    pss_top.compute.stop[15]
    pss_top.compute.stop[16]
    pss_top.compute.stop[17]
    pss_top.compute.stop[18]
    pss_top.compute.stop[19]
    pss_top.compute.stop[1]
    pss_top.compute.stop[20]
    pss_top.compute.stop[21]
    pss_top.compute.stop[22]
    pss_top.compute.stop[23]
    pss_top.compute.stop[24]
    pss_top.compute.stop[25]
    pss_top.compute.stop[26]
    pss_top.compute.stop[27]
    pss_top.compute.stop[28]
    pss_top.compute.stop[29]
    pss_top.compute.stop[2]
    pss_top.compute.stop[30]
    pss_top.compute.stop[31]
    pss_top.compute.stop[32]
    pss_top.compute.stop[33]
    pss_top.compute.stop[34]
    pss_top.compute.stop[35]
    pss_top.compute.stop[36]
    pss_top.compute.stop[37]
    pss_top.compute.stop[38]
    pss_top.compute.stop[39]
    pss_top.compute.stop[3]
    pss_top.compute.stop[40]
    pss_top.compute.stop[41]
    pss_top.compute.stop[42]
    pss_top.compute.stop[43]
    pss_top.compute.stop[44]
    pss_top.compute.stop[45]
    pss_top.compute.stop[46]
    pss_top.compute.stop[47]
    pss_top.compute.stop[48]
    pss_top.compute.stop[49]
    pss_top.compute.stop[4]
    pss_top.compute.stop[50]
    pss_top.compute.stop[51]
    pss_top.compute.stop[52]
    pss_top.compute.stop[53]
    pss_top.compute.stop[54]
    pss_top.compute.stop[55]
    pss_top.compute.stop[56]
    pss_top.compute.stop[57]
    pss_top.compute.stop[58]
    pss_top.compute.stop[59]
    pss_top.compute.stop[5]
    pss_top.compute.stop[60]
    pss_top.compute.stop[61]
    pss_top.compute.stop[62]
    pss_top.compute.stop[63]
    pss_top.compute.stop[64]
    pss_top.compute.stop[65]
    pss_top.compute.stop[66]
    pss_top.compute.stop[67]
    pss_top.compute.stop[68]
    pss_top.compute.stop[69]
    pss_top.compute.stop[6]
    pss_top.compute.stop[70]
    pss_top.compute.stop[71]
    pss_top.compute.stop[72]
    pss_top.compute.stop[73]
    pss_top.compute.stop[74]
    pss_top.compute.stop[75]
    pss_top.compute.stop[76]
    pss_top.compute.stop[77]
    pss_top.compute.stop[78]
    pss_top.compute.stop[79]
    pss_top.compute.stop[7]
    pss_top.compute.stop[80]
    pss_top.compute.stop[81]
    pss_top.compute.stop[82]
    pss_top.compute.stop[83]
    pss_top.compute.stop[84]
    pss_top.compute.stop[85]
    pss_top.compute.stop[86]
    pss_top.compute.stop[87]
    pss_top.compute.stop[88]
    pss_top.compute.stop[89]
    pss_top.compute.stop[8]
    pss_top.compute.stop[90]
    pss_top.compute.stop[91]
    pss_top.compute.stop[92]
    pss_top.compute.stop[93]
    pss_top.compute.stop[94]
    pss_top.compute.stop[95]
    pss_top.compute.stop[96]
    pss_top.compute.stop[97]
    pss_top.compute.stop[98]
    pss_top.compute.stop[99]
    pss_top.compute.stop[9]
    pss_top.entry
    pss_top.riscv_A.RvA_AMOADDD
    pss_top.riscv_A.RvA_AMOADDW
    pss_top.riscv_A.RvA_AMOANDD
```

pss_top.entry
coverage: 27/103 (26%), paths: 103

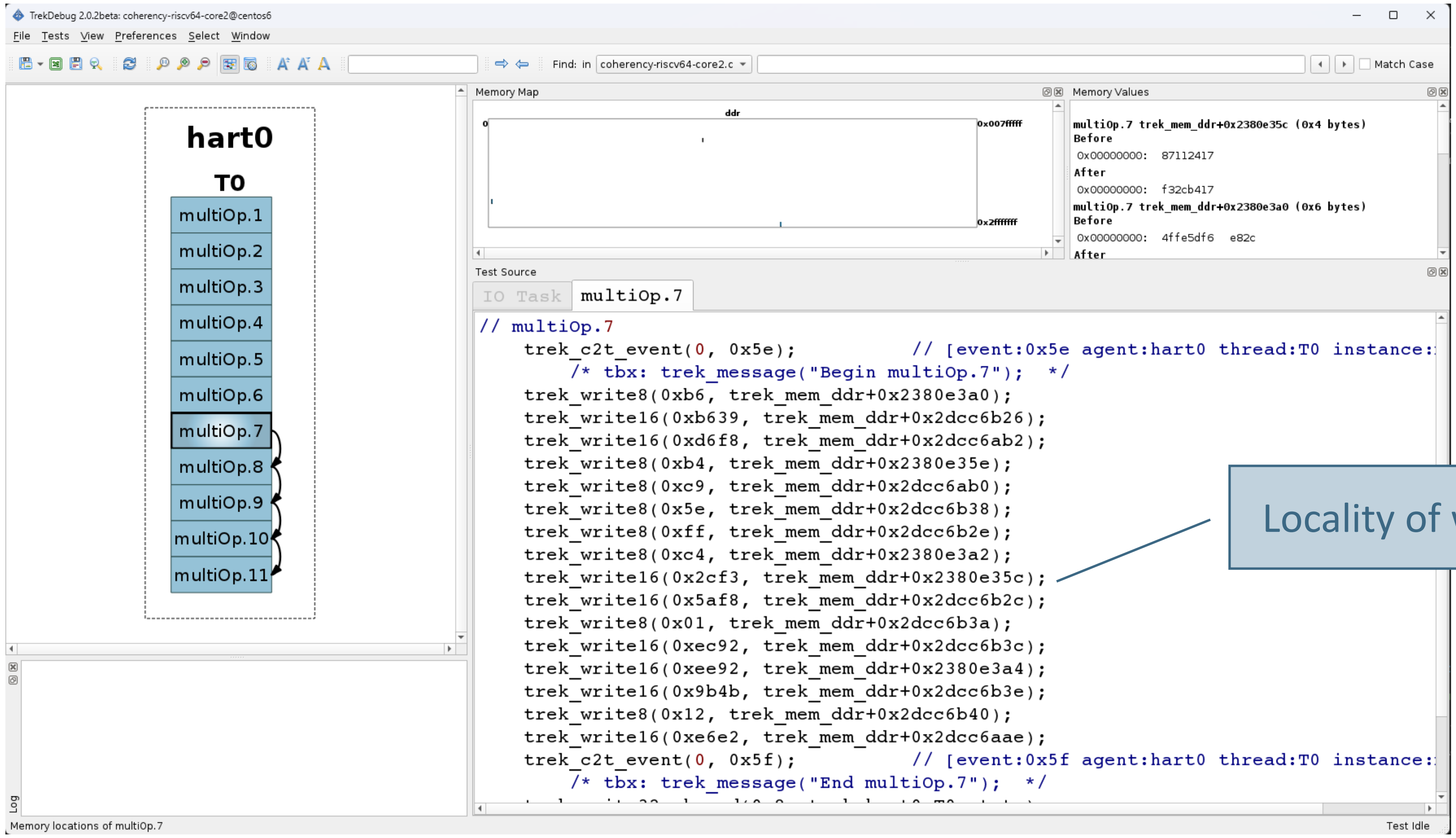
27/103 reachable opcode have been exercised



pss_top.entry
coverage: 27/103 (26%), paths: 152

Atomics, loads and stores not reachable in register only test

RV64 Core Load/Store



Memory Values

```
multiOp.7 trek_mem_ddr+0x2380e35c (0x4 bytes)
Before
 0x00000000:  87112417
After
 0x00000000:  f32cb417
multiOp.7 trek_mem_ddr+0x2380e3a0 (0x6 bytes)
Before
 0x00000000:  4ffe5df6  e82c
After
```

```
// multiOp.7
    trek_c2t_event(0, 0x5e);          // [event:0x5e agent:hart0 thread:T0 instance:
        /* tbx: trek_message("Begin multiOp.7");  */
    trek_write8(0xb6, trek_mem_ddr+0x2380e3a0);
    trek_write16(0xb639, trek_mem_ddr+0x2dcc6b26);
    trek_write16(0xd6f8, trek_mem_ddr+0x2dcc6ab2);
    trek_write8(0xb4, trek_mem_ddr+0x2380e35e);
    trek_write8(0xc9, trek_mem_ddr+0x2dcc6ab0);
    trek_write8(0x5e, trek_mem_ddr+0x2dcc6b38);
    trek_write8(0xff, trek_mem_ddr+0x2dcc6b2e);
    trek_write8(0xc4, trek_mem_ddr+0x2380e3a2);
    trek_write16(0x2cf3, trek_mem_ddr+0x2380e35c);
    trek_write16(0x5af8, trek_mem_ddr+0x2dcc6b2c);
    trek_write8(0x01, trek_mem_ddr+0x2dcc6b3a);
    trek_write16(0xec92, trek_mem_ddr+0x2dcc6b3c);
    trek_write16(0xee92, trek_mem_ddr+0x2380e3a4);
    trek_write16(0x9b4b, trek_mem_ddr+0x2dcc6b3e);
    trek_write8(0x12, trek_mem_ddr+0x2dcc6b40);
    trek_write16(0xe6e2, trek_mem_ddr+0x2dcc6aae);
    trek_c2t_event(0, 0x5f);          // [event:0x5f agent:hart0 thread:T0 instance:
        /* tbx: trek_message("End multiOp.7");  */
```

Locality of write addrs

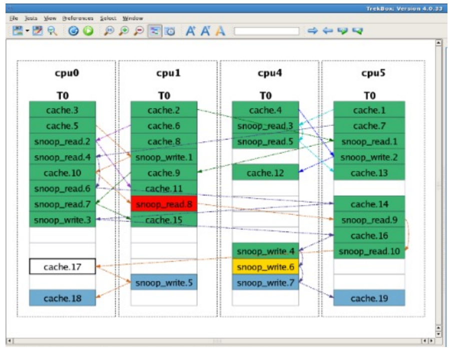
Example Address Allocation Patterns

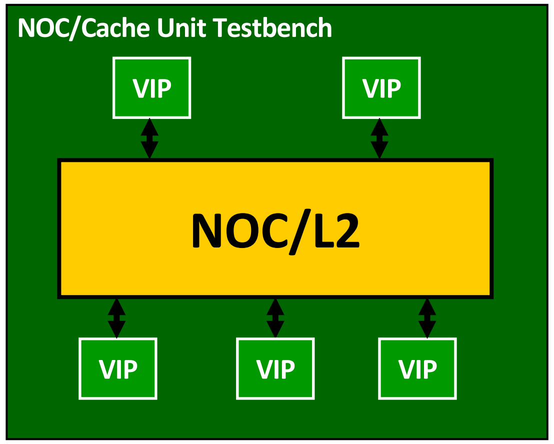
```
// memAllocAddrSlice allocated setId:0x1 of 0x4 blocks
// memAllocAddrRand size:0x8 addr: trek_mem_ddr+0x08b810c8
// memAllocAddrRand size:0x8 addr: trek_mem_ddr+0x2380e378
// memAllocAddrRand size:0x8 addr: trek_mem_ddr+0x2380e380
// memAllocAddrRand size:0x8 addr: trek_mem_ddr+0x2380e370
```

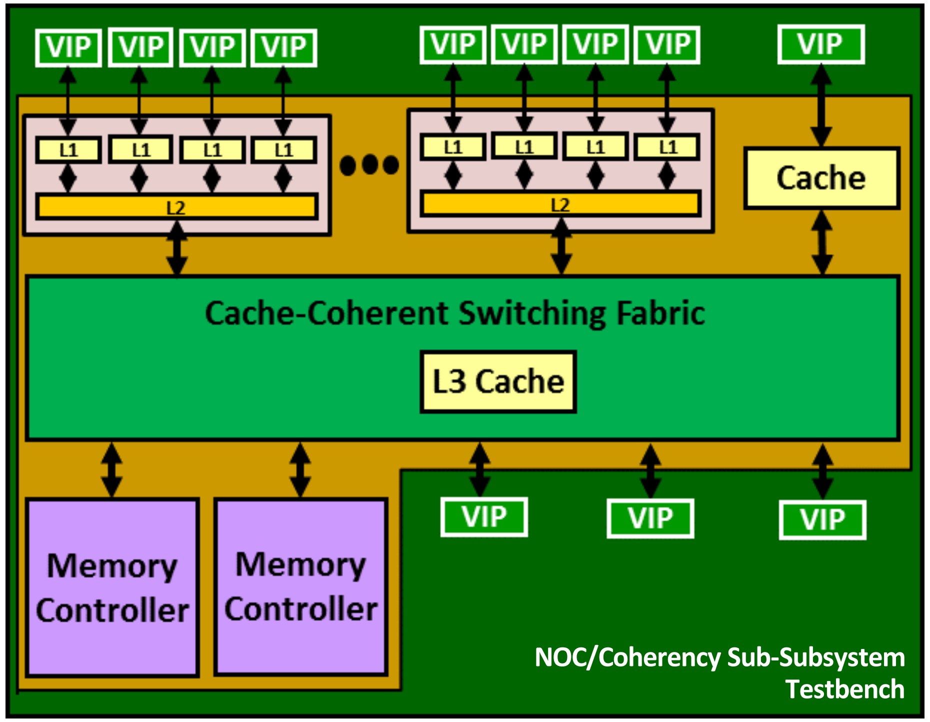
```
// memAllocAddrSlice allocated setId:0x1 of 0x4 blocks
// memAllocAddrStride stride_len:0x2000 size:0x8 addr: trek_mem_ddr+0x08b830c8
// memAllocAddrStride stride_len:0x2000 size:0x8 addr: trek_mem_ddr+0x08b850c8
// memAllocAddrStride stride_len:0x2000 size:0x8 addr: trek_mem_ddr+0x08b870c8
// memAllocAddrStride stride_len:0x2000 size:0x8 addr: trek_mem_ddr+0x08b890c8
```

```
// memAllocAddrSlice allocated setId:0x1 of 0x4 blocks
// memAllocAddrHash hash:0x44 size:0x100 addr: trek_mem_ddr+0x08bc1100
// memAllocAddrHash hash:0x44 size:0x100 addr: trek_mem_ddr+0x08c01100
// memAllocAddrHash hash:0x44 size:0x100 addr: trek_mem_ddr+0x08c41100
// memAllocAddrHash hash:0x44 size:0x100 addr: trek_mem_ddr+0x08c81100
```

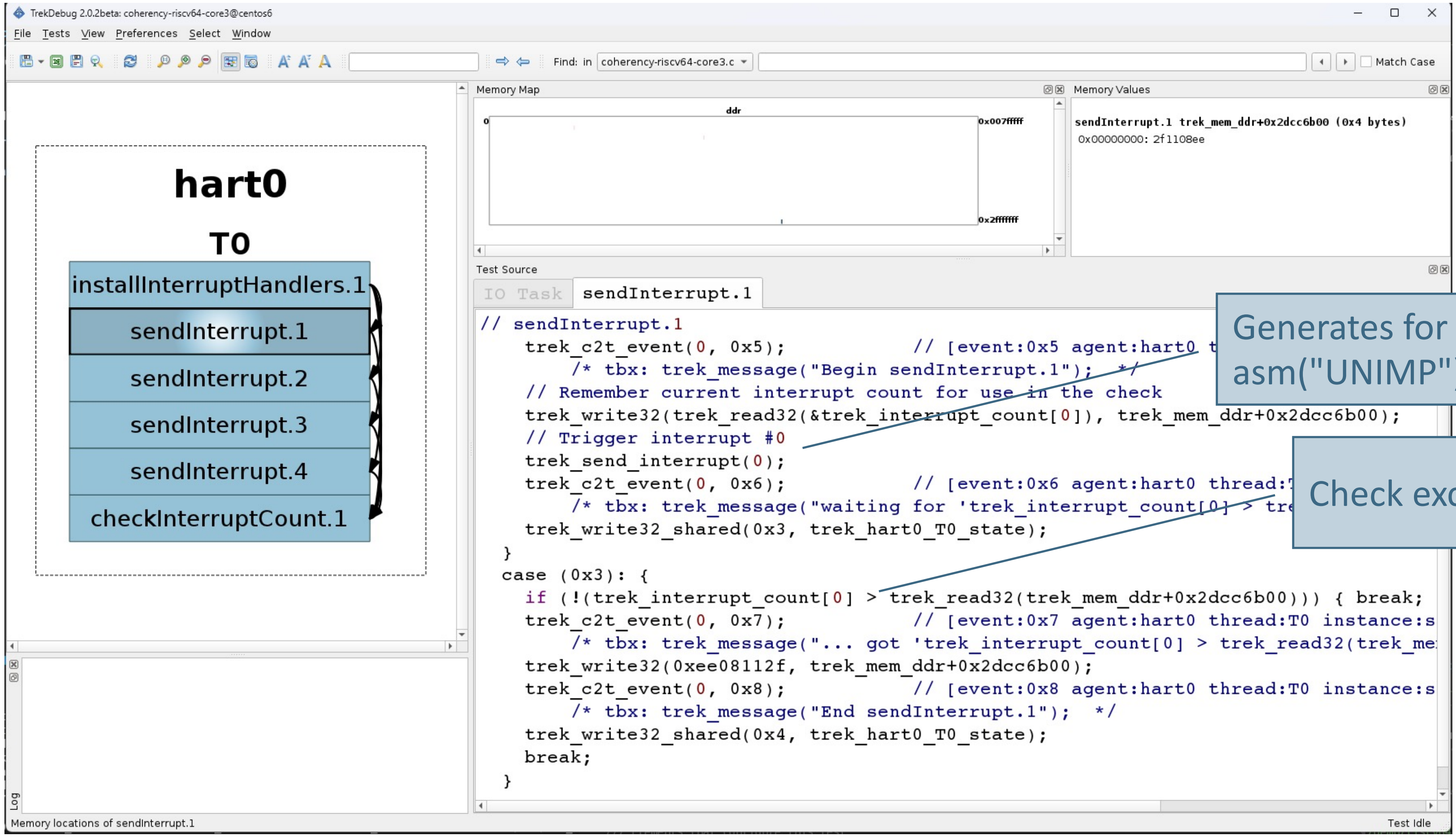
Application to Unit Bench and Sub-System Bench







RV64 Core Exception Testing



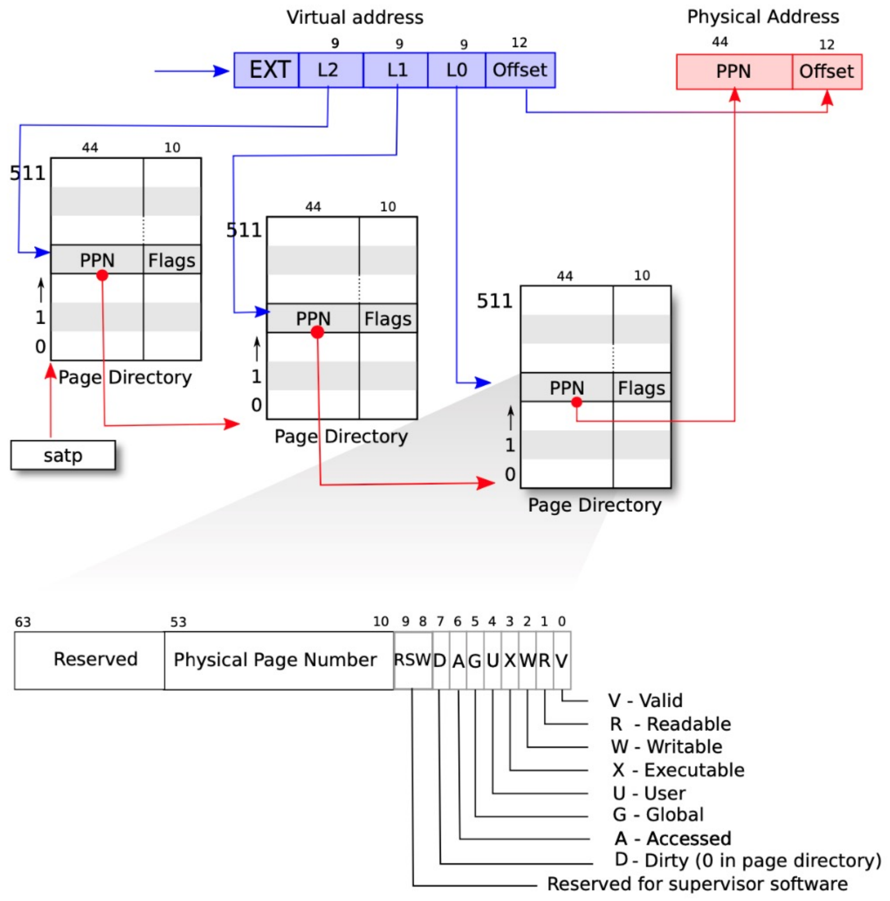
```
sendInterrupt.1 trek_mem_ddr+0x2dcc6b00 (0x4 bytes)
 0x00000000: 2f1108ee
```

```
// sendInterrupt.1
    trek_c2t_event(0, 0x5);              // [event:0x5 agent:hart0 t
        /* tbx: trek_message("Begin sendInterrupt.1");  */
    // Remember current interrupt count for use in the check
    trek_write32(trek_read32(&trek_interrupt_count[0]), trek_mem_ddr+0x2dcc6b00);
    // Trigger interrupt #0
    trek_send_interrupt(0);
    trek_c2t_event(0, 0x6);              // [event:0x6 agent:hart0 thread:T
        /* tbx: trek_message("waiting for 'trek_interrupt_count[0] > tre
    trek_write32_shared(0x3, trek_hart0_T0_state);
  }
  case (0x3): {
    if (!(trek_interrupt_count[0] > trek_read32(trek_mem_ddr+0x2dcc6b00))) { break;
    trek_c2t_event(0, 0x7);              // [event:0x7 agent:hart0 thread:T0 instance:s
        /* tbx: trek_message("... got 'trek_interrupt_count[0] > trek_read32(trek_me
    trek_write32(0xee08112f, trek_mem_ddr+0x2dcc6b00);
    trek_c2t_event(0, 0x8);              // [event:0x8 agent:hart0 thread:T0 instance:s
        /* tbx: trek_message("End sendInterrupt.1");  */
    trek_write32_shared(0x4, trek_hart0_T0_state);
    break;
  }
```

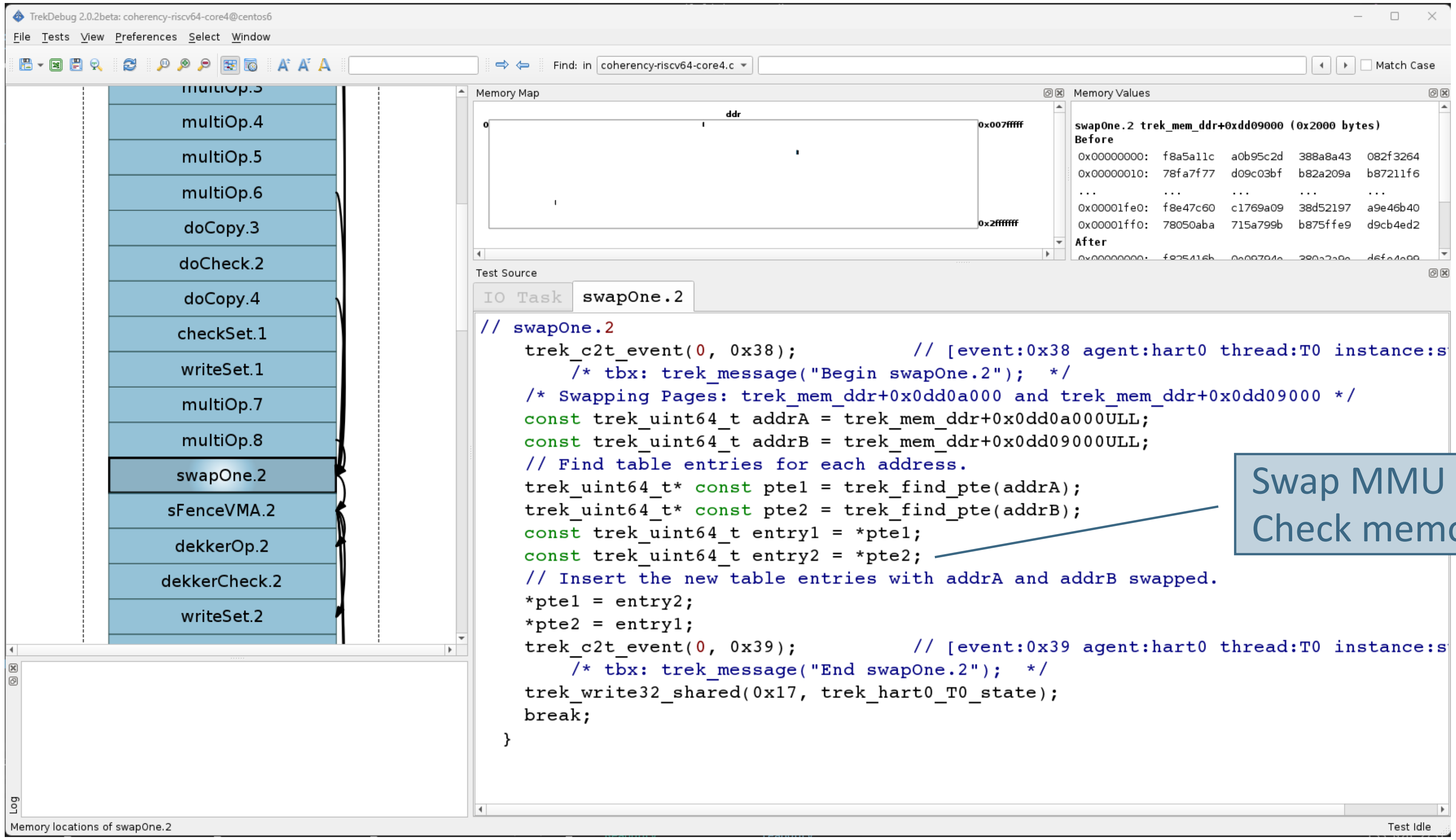
Generates for example, asm("UNIMP");

Check exception counts

Page Based Virtual Memory Tests



RV64 Core Page Based MMU Tests

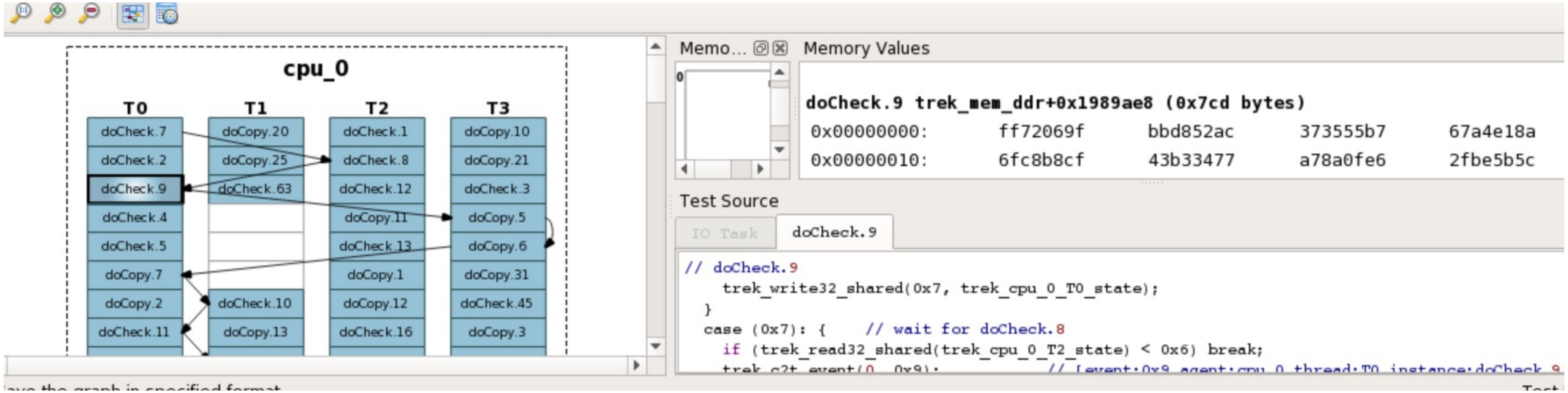


```
swapOne.2 trek_mem_ddr+0xdd09000 (0x2000 bytes)
Before
 0x00000000:  f8a5a11c  a0b95c2d  388a8a43  082f3264
 0x00000010:  78fa7f77  d09c03bf  b82a209a  b87211f6
 ...          ...       ...       ...       ...
 0x00001fe0:  f8e47c60  c1769a09  38d52197  a9e46b40
 0x00001ff0:  78050aba  715a799b  b875ffe9  d9cb4ed2
After
```

```
// swapOne.2
    trek_c2t_event(0, 0x38);           // [event:0x38 agent:hart0 thread:T0 instance:s
        /* tbx: trek_message("Begin swapOne.2");  */
    /* Swapping Pages: trek_mem_ddr+0x0dd0a000 and trek_mem_ddr+0x0dd09000 */
    const trek_uint64_t addrA = trek_mem_ddr+0x0dd0a000ULL;
    const trek_uint64_t addrB = trek_mem_ddr+0x0dd09000ULL;
    // Find table entries for each address.
    trek_uint64_t* const pte1 = trek_find_pte(addrA);
    trek_uint64_t* const pte2 = trek_find_pte(addrB);
    const trek_uint64_t entry1 = *pte1;
    const trek_uint64_t entry2 = *pte2;
    // Insert the new table entries with addrA and addrB swapped.
    *pte1 = entry2;
    *pte2 = entry1;
    trek_c2t_event(0, 0x39);           // [event:0x39 agent:hart0 thread:T0 instance:s
        /* tbx: trek_message("End swapOne.2");  */
    trek_write32_shared(0x17, trek_hart0_T0_state);
    break;
  }
```

Swap MMU PTE's and Check memory access

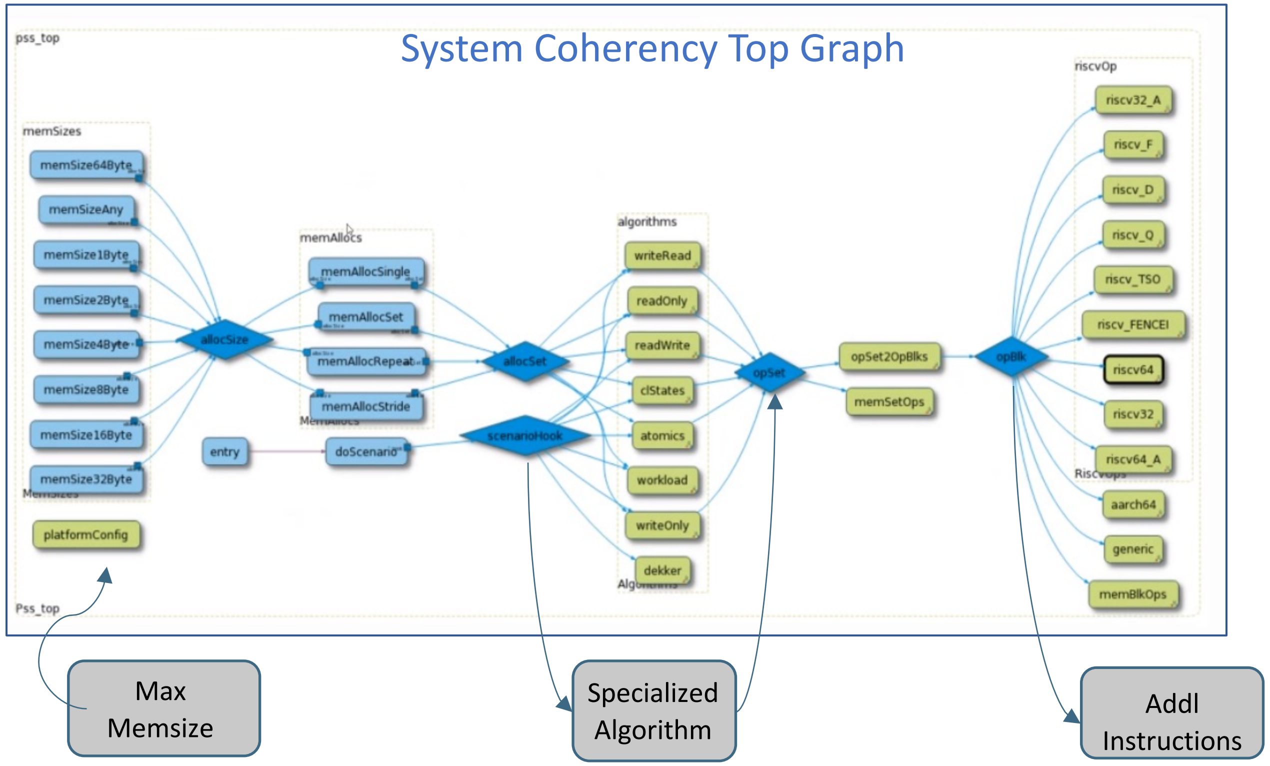
Core-Integrity: Single Core, 4 Threads



```
doCheck.9 trek_mem_ddr+0x1989ae8 (0x7cd bytes)
0x00000000:      ff72069f      bbd852ac      373555b7      67a4e18a
0x00000010:      6fc8b8cf      43b33477      a78a0fe6      2fbe5b5c
```

```
// doCheck.9
    trek_write32_shared(0x7, trek_cpu_0_T0_state);
  }
  case (0x7): {     // wait for doCheck.8
    if (trek_read32_shared(trek_cpu_0_T2_state) < 0x6) break;
    trek_c2t_event(0, 0x9);          // [event:0x9 agent:cpu_0 thread:T0 instance:doCheck.9
```

Modular, Configurable and Extendable Building Blocks



Example Customizations

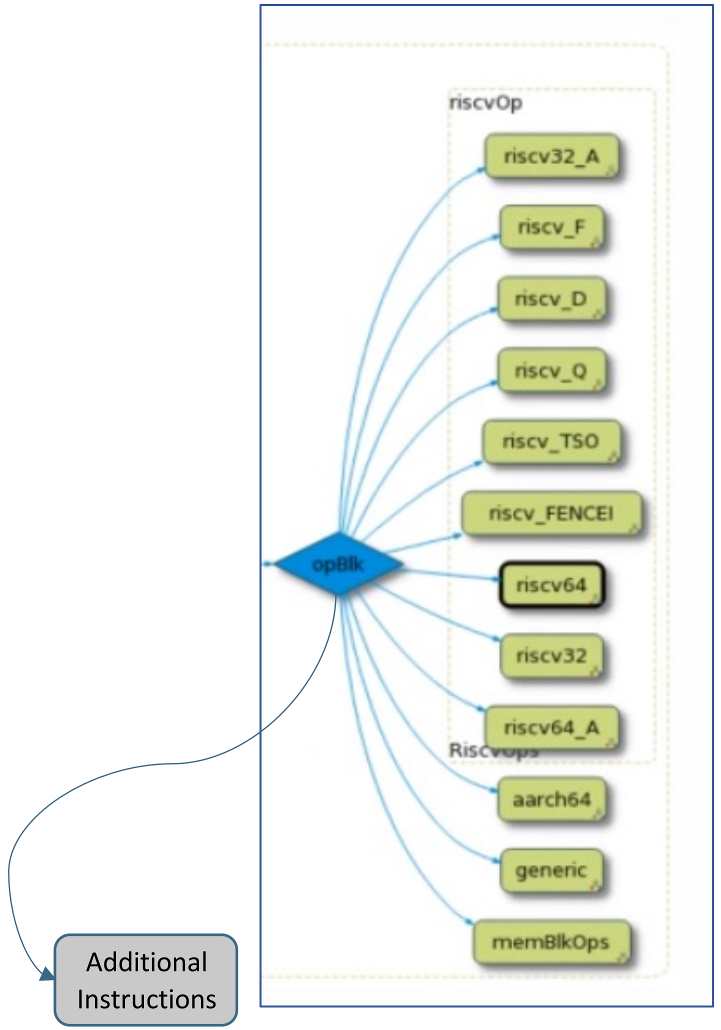
Specific Component Characteristic

Special Coherency Test Algorithm

Extra Processor Instruction

Testing a Custom Instruction

- RISC-V ISA custom instructions pose a particularly difficult verification challenge
- Custom instructions need to be tested with the processor tests, not as an afterthought
- Breker solution allows custom instruction tests to be easily added into test graph
- Breker synthesis combines these tests with the app to ensure full custom processor testing



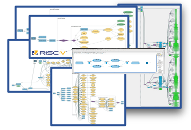
Agenda

- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

SoC-Integrity Challenges

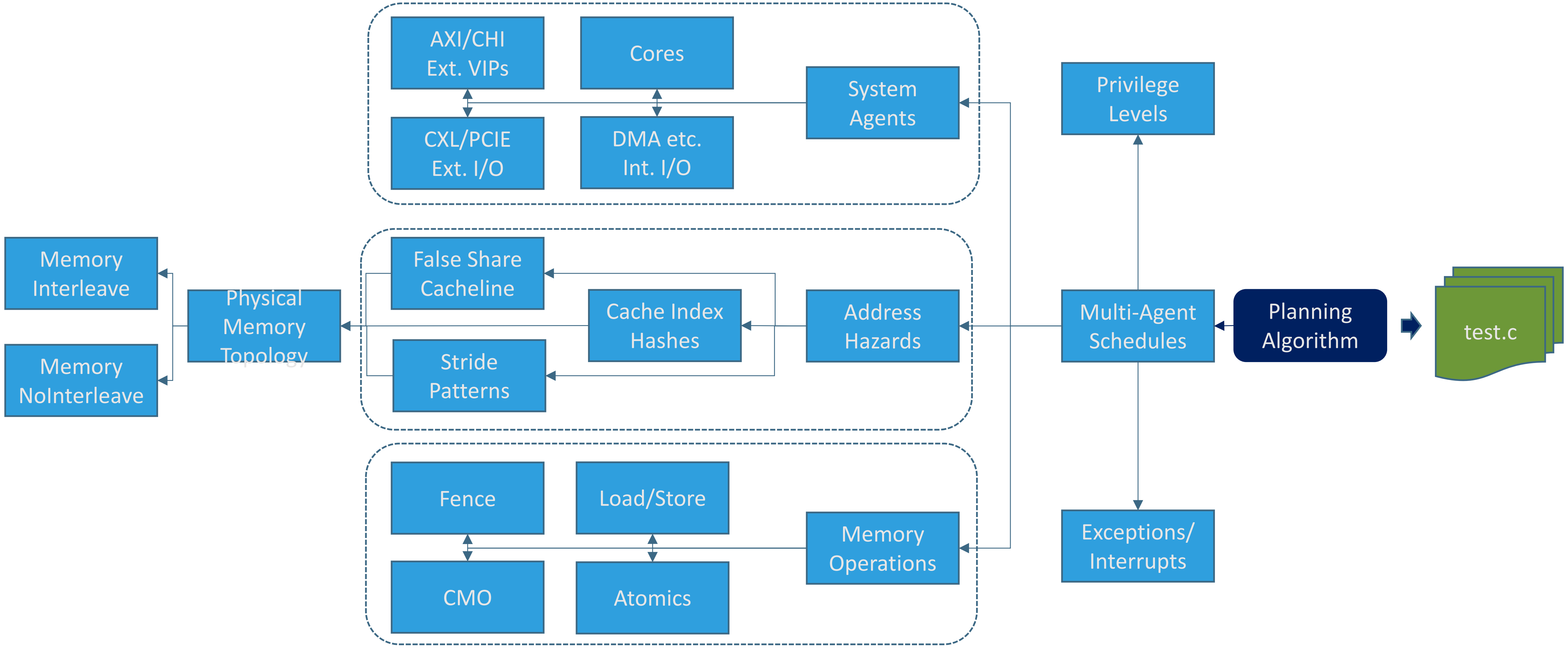
Random Memory Tests	Test Cores/Fabrics/Memory controllers across DDR, OCRAM, FLASH etc
Random Register Tests	Read/write test to all uncore registers
System Interrupts	Randomized interrupts through CLINT
Multi-core execution	Concurrent operations on fabric and memory
Memory ordering	For weakly order memory protocols
Atomic operation	Across all memory types
System Coherency	Cover all cache transitions, evictions, snoops
System Paging/IOMMU	System memory virtualization
System Security	Register and Memory protection across system
Power Management	System wide sleep/wakeup and voltage/freq scaling

Breker
RISC-V SoC-Integrity
SystemVIP

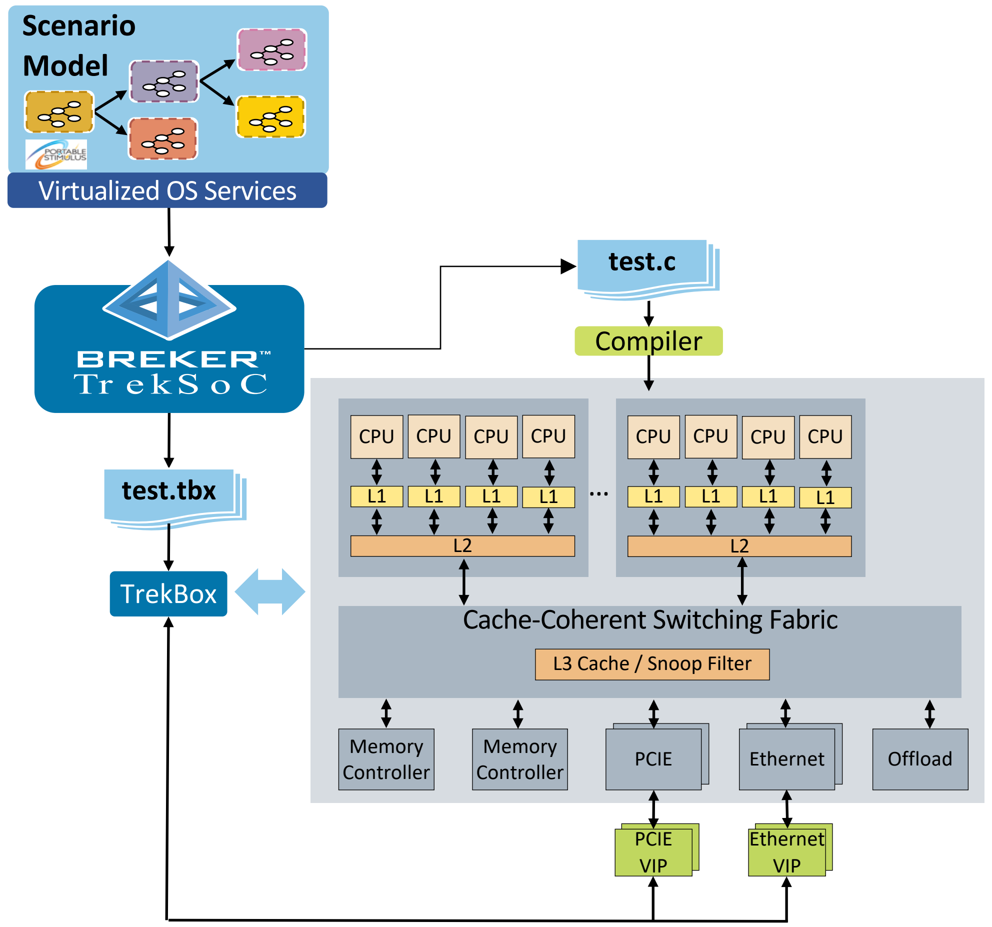


- **End-to-End use cases**
- **Early Firmware Testing**
- **Performance-Power Profiling**

RISC-V SoC Integrity TrekApp

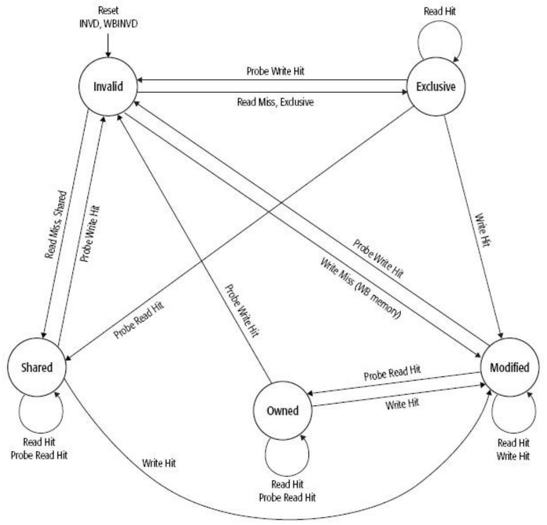


RISC-V SoC Testbench Integration



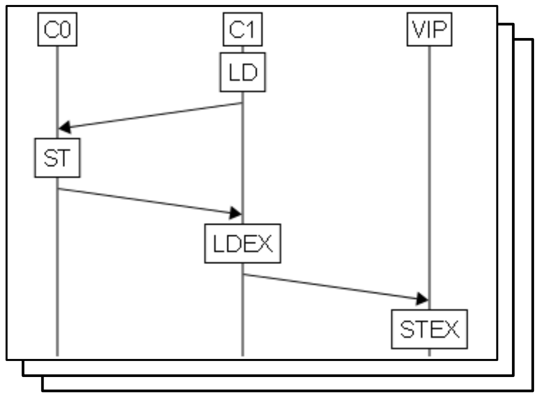
Multi-Agent Scheduling Plans: Overview

- True Sharing within scenario
- False Sharing across scenarios

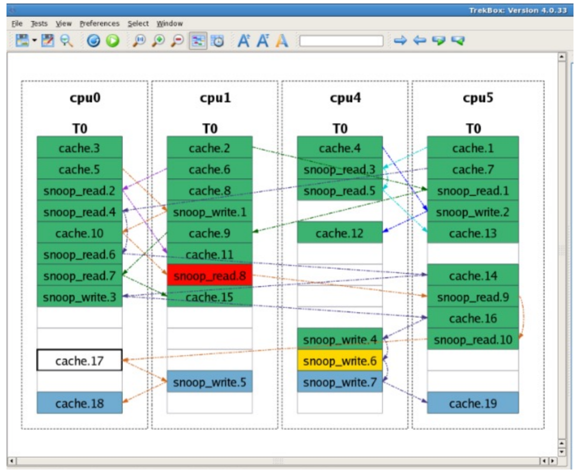


N **Transition Sequences**

N Transition Scenarios

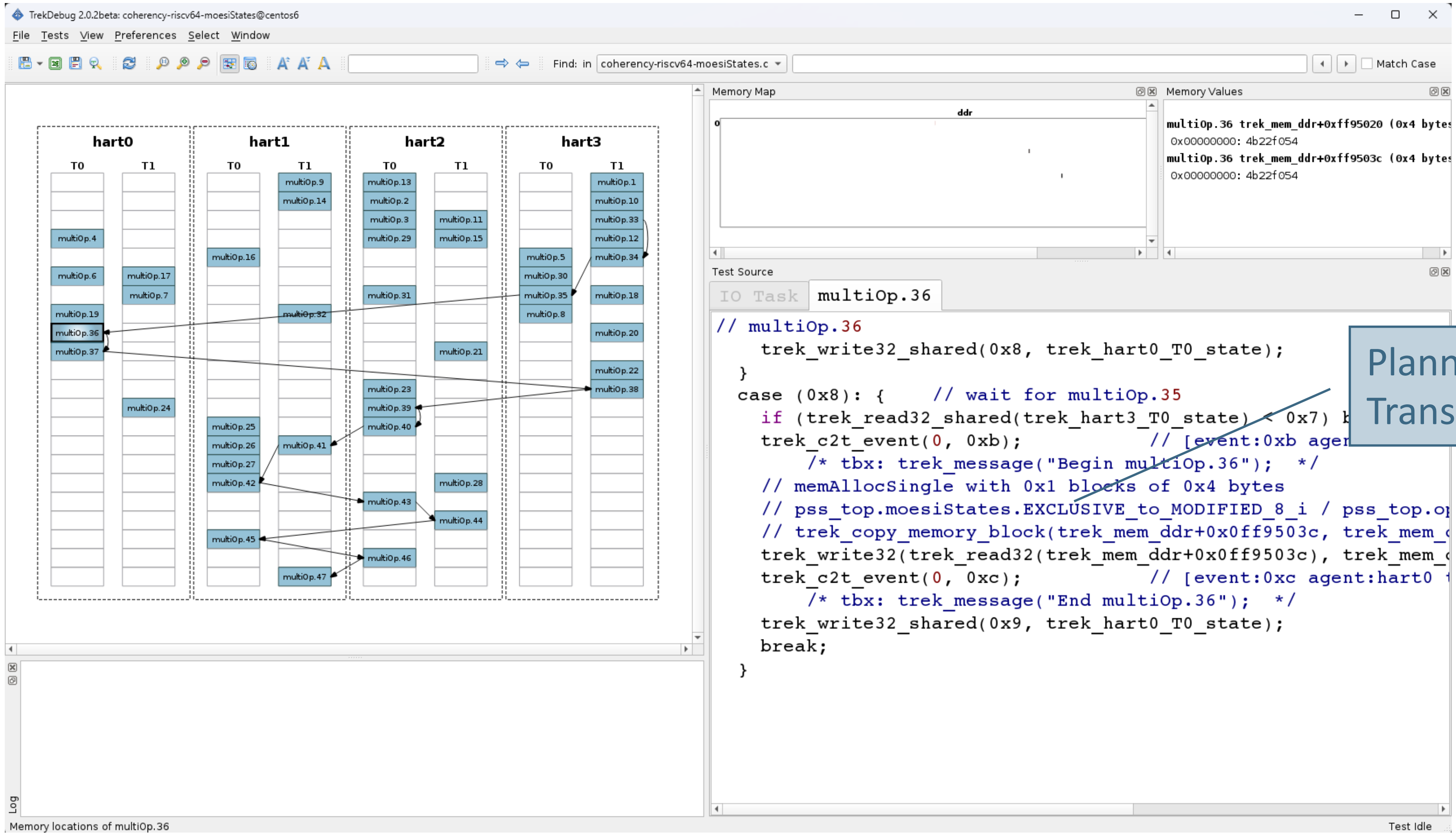


Concurrent Scenario Test Case



Schedule Memory
Interleave & Pack
Resolve Dependencies

RV64 MultiCore MoesiStates


TrekDebug 2.0.2beta: coherency-riscv64-moesiStates@centos6
File Tests View Preferences Select Window
Find: in coherency-riscv64-moesiStates.c
Match Case
hart0
hart1
hart2
hart3
T0
T1
Memory Map
ddr
Memory Values
Test Source
IO Task
multiOp.36
Log
Memory locations of multiOp.36
Test Idle

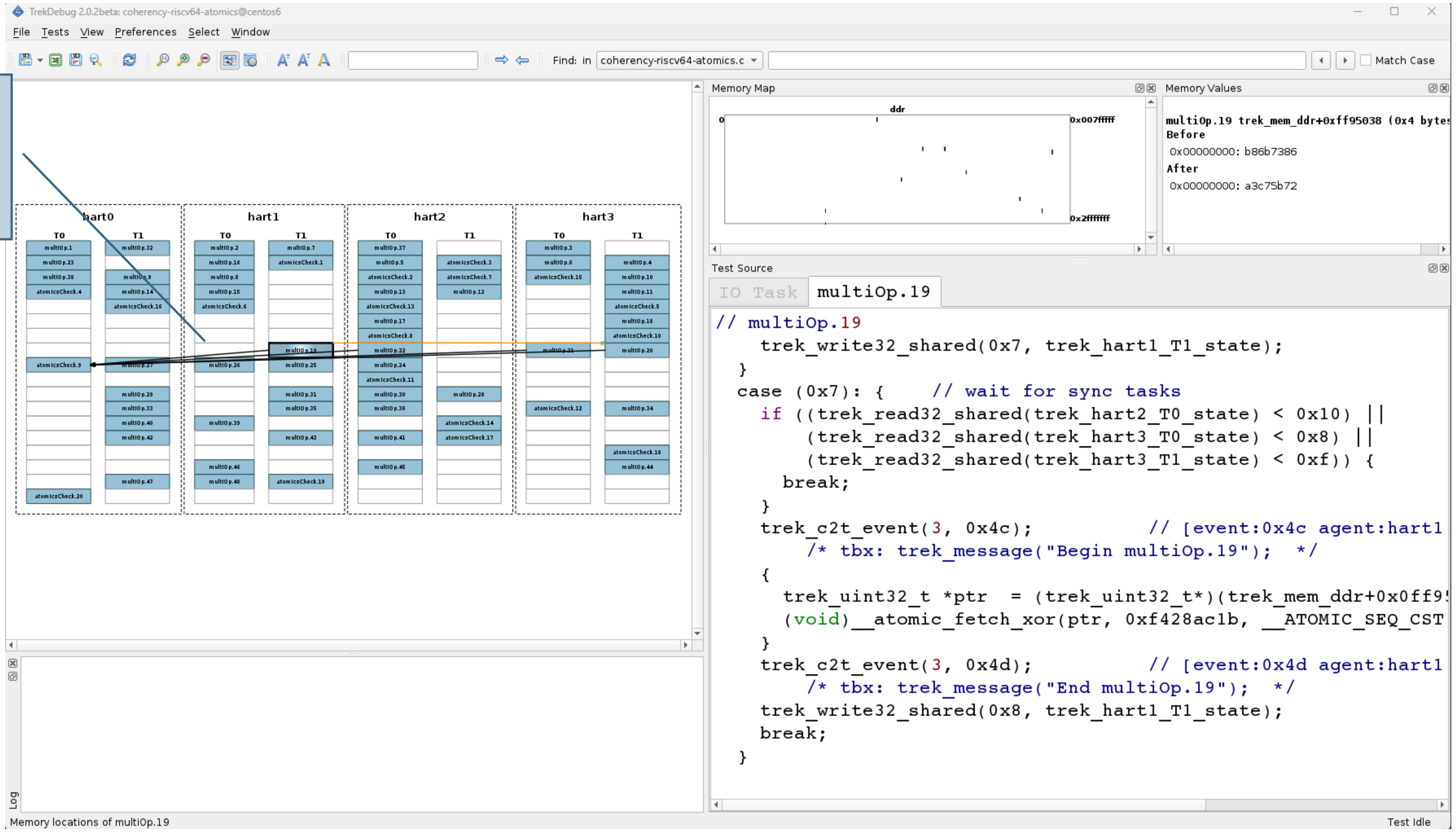
```
multiOp.36 trek_mem_ddr+0xff95020 (0x4 bytes
 0x00000000: 4b22f054
multiOp.36 trek_mem_ddr+0xff9503c (0x4 bytes
 0x00000000: 4b22f054
```

```
// multiOp.36
    trek_write32_shared(0x8, trek_hart0_T0_state);
  }
  case (0x8): {    // wait for multiOp.35
    if (trek_read32_shared(trek_hart3_T0_state) < 0x7) b
    trek_c2t_event(0, 0xb);                // [event:0xb agen
        /* tbx: trek_message("Begin multiOp.36");  */
    // memAllocSingle with 0x1 blocks of 0x4 bytes
    // pss_top.moesiStates.EXCLUSIVE_to_MODIFIED_8_i / pss_top.op
    // trek_copy_memory_block(trek_mem_ddr+0x0ff9503c, trek_mem_c
    trek_write32(trek_read32(trek_mem_ddr+0x0ff9503c), trek_mem_c
    trek_c2t_event(0, 0xc);                // [event:0xc agent:hart0 t
        /* tbx: trek_message("End multiOp.36");  */
    trek_write32_shared(0x9, trek_hart0_T0_state);
    break;
  }
```

Planned Cache State Transitions

Atomics Testing

Check result is aggregate of synchronized atomic operations



```
multiOp.19 trek_mem_ddr+0xff95038 (0x4 bytes
Before
 0x00000000: b86b7386
After
 0x00000000: a3c75b72
```

```
// multiOp.19
    trek_write32_shared(0x7, trek_hart1_T1_state);
  }
  case (0x7): {     // wait for sync tasks
    if ((trek_read32_shared(trek_hart2_T0_state) < 0x10) ||
        (trek_read32_shared(trek_hart3_T0_state) < 0x8) ||
        (trek_read32_shared(trek_hart3_T1_state) < 0xf)) {
      break;
    }
    trek_c2t_event(3, 0x4c);            // [event:0x4c agent:hart1
        /* tbx: trek_message("Begin multiOp.19");  */
    {
      trek_uint32_t *ptr  = (trek_uint32_t*)(trek_mem_ddr+0x0ff9
      (void)__atomic_fetch_xor(ptr, 0xf428ac1b, __ATOMIC_SEQ_CST
    }
    trek_c2t_event(3, 0x4d);            // [event:0x4d agent:hart1
        /* tbx: trek_message("End multiOp.19");  */
    trek_write32_shared(0x8, trek_hart1_T1_state);
    break;
  }
```

Memory locations of multiOp.19

RISC-V SoC Memory Ordering: Dekker Algorithm

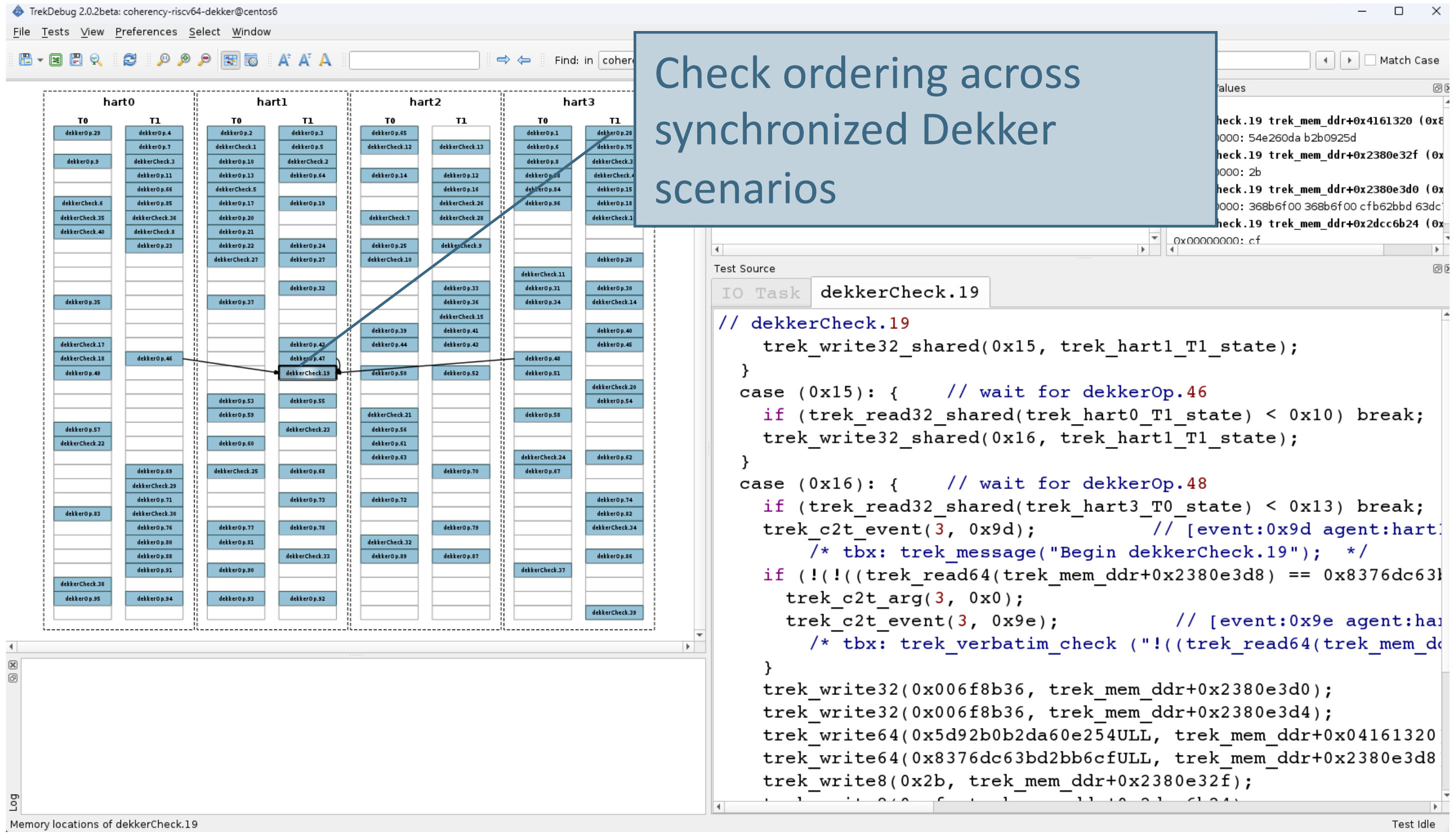
- Assume initial state A=0 , B=0

- The Dekker Algorithm States

```
core 0: ST A, 1; MEM_BARRIER; LD B
core 1: ST B, 1; MEM_BARRIER; LD A
error iff ( A == 0 && B == 0 )
```

- This is a test for a weakly ordered memory system
  - Such a system must preserve the property that a LD may not reorder ahead of a previous ST from the same agent

Dekker Memory Ordering



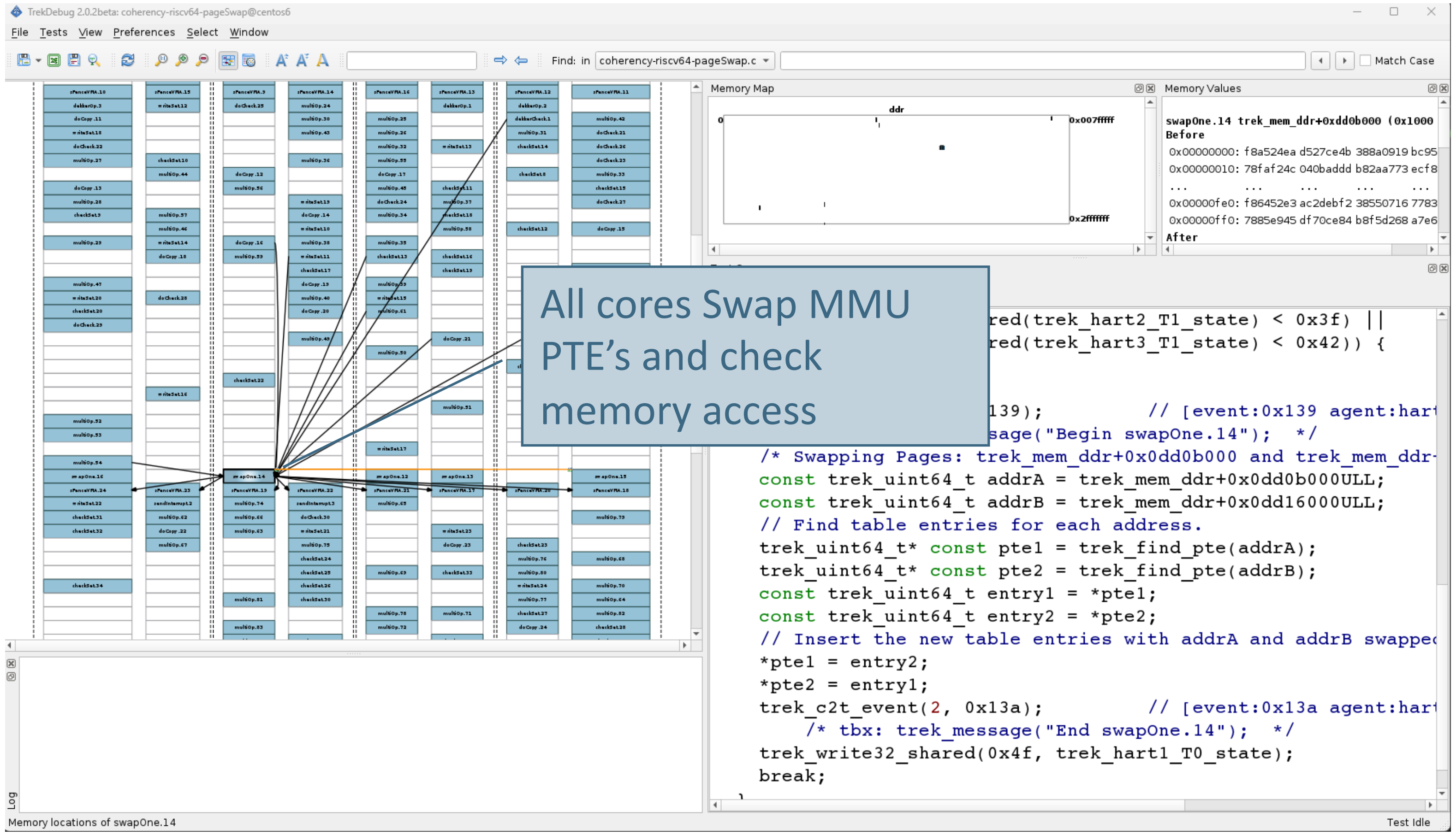
TrekDebug 2.0.2beta: coherency-riscv64-dekker@centos6
File Tests View Preferences Select Window
Find: in coher
Match Case
hart0
hart1
hart2
hart3
T0
T1
Values
Test Source
IO Task
dekkerCheck.19
Log
Memory locations of dekkerCheck.19
Test Idle

Check ordering across synchronized Dekker scenarios

```
heck.19 trek_mem_ddr+0x4161320 (0x8
0000: 54e260da b2b0925d
heck.19 trek_mem_ddr+0x2380e32f (0x
0000: 2b
heck.19 trek_mem_ddr+0x2380e3d0 (0x
0000: 368b6f00 368b6f00 cfb62bbd 63dc
heck.19 trek_mem_ddr+0x2dcc6b24 (0x
0x00000000: cf
```

```
// dekkerCheck.19
    trek_write32_shared(0x15, trek_hart1_T1_state);
  }
  case (0x15): {    // wait for dekkerOp.46
    if (trek_read32_shared(trek_hart0_T1_state) < 0x10) break;
    trek_write32_shared(0x16, trek_hart1_T1_state);
  }
  case (0x16): {    // wait for dekkerOp.48
    if (trek_read32_shared(trek_hart3_T0_state) < 0x13) break;
    trek_c2t_event(3, 0x9d);           // [event:0x9d agent:hart
      /* tbx: trek_message("Begin dekkerCheck.19");  */
    if (!(!((trek_read64(trek_mem_ddr+0x2380e3d8) == 0x8376dc63
      trek_c2t_arg(3, 0x0);
      trek_c2t_event(3, 0x9e);           // [event:0x9e agent:ha
      /* tbx: trek_verbatim_check ("!((trek_read64(trek_mem_d
    }
    trek_write32(0x006f8b36, trek_mem_ddr+0x2380e3d0);
    trek_write32(0x006f8b36, trek_mem_ddr+0x2380e3d4);
    trek_write64(0x5d92b0b2da60e254ULL, trek_mem_ddr+0x04161320
    trek_write64(0x8376dc63bd2bb6cfULL, trek_mem_ddr+0x2380e3d8
    trek_write8(0x2b, trek_mem_ddr+0x2380e32f);
```

MultiCore MMU Tests


TrekDebug 2.0.2beta: coherency-riscv64-pageSwap@centos6
File Tests View Preferences Select Window
Find: in coherency-riscv64-pageSwap.c
Match Case
Memory Map
ddr
0
0x007fffff
0x2fffffff
Memory Values
Log
Memory locations of swapOne.14
Test Idle

```
swapOne.14 trek_mem_ddr+0xdd0b000 (0x1000
Before
 0x00000000: f8a524ea d527ce4b 388a0919 bc95
 0x00000010: 78faf24c 040baddd b82aa773 ecf8
 ...         ...      ...      ...      ...
 0x00000fe0: f86452e3 ac2debf2 38550716 7783
 0x00000ff0: 7885e945 df70ce84 b8f5d268 a7e6
After
```

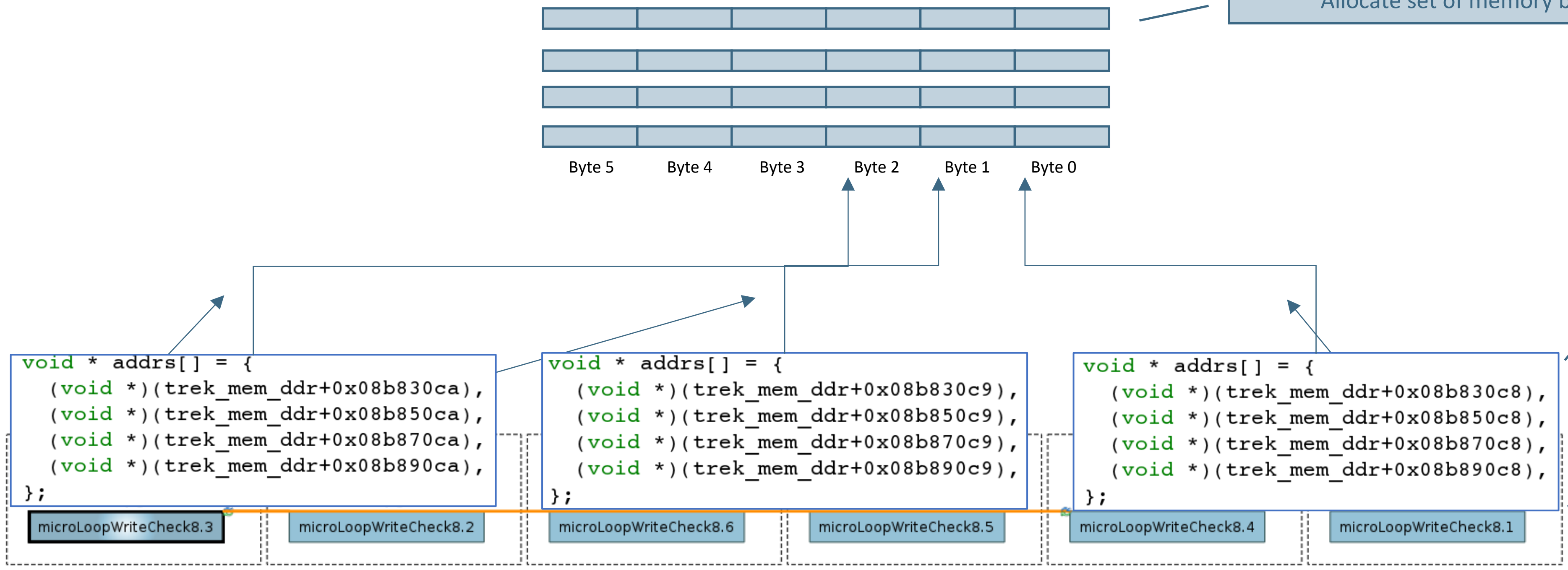
```
red(trek_hart2_T1_state) < 0x3f) ||
red(trek_hart3_T1_state) < 0x42)) {

139);            // [event:0x139 agent:hart
sage("Begin swapOne.14");  */
    /* Swapping Pages: trek_mem_ddr+0x0dd0b000 and trek_mem_ddr
    const trek_uint64_t addrA = trek_mem_ddr+0x0dd0b000ULL;
    const trek_uint64_t addrB = trek_mem_ddr+0x0dd16000ULL;
    // Find table entries for each address.
    trek_uint64_t* const pte1 = trek_find_pte(addrA);
    trek_uint64_t* const pte2 = trek_find_pte(addrB);
    const trek_uint64_t entry1 = *pte1;
    const trek_uint64_t entry2 = *pte2;
    // Insert the new table entries with addrA and addrB swappe
    *pte1 = entry2;
    *pte2 = entry1;
    trek_c2t_event(2, 0x13a);            // [event:0x13a agent:hart
        /* tbx: trek_message("End swapOne.14");  */
    trek_write32_shared(0x4f, trek_hart1_T0_state);
    break;
```

All cores Swap MMU PTE's and check memory access

False-Share Memory Stress Tests

Allocate set of memory blocks



```
void * addrs[] = {
   (void *)(trek_mem_ddr+0x08b830ca),
   (void *)(trek_mem_ddr+0x08b850ca),
   (void *)(trek_mem_ddr+0x08b870ca),
   (void *)(trek_mem_ddr+0x08b890ca),
};
```

```
void * addrs[] = {
   (void *)(trek_mem_ddr+0x08b830c9),
   (void *)(trek_mem_ddr+0x08b850c9),
   (void *)(trek_mem_ddr+0x08b870c9),
   (void *)(trek_mem_ddr+0x08b890c9),
};
```

```
void * addrs[] = {
   (void *)(trek_mem_ddr+0x08b830c8),
   (void *)(trek_mem_ddr+0x08b850c8),
   (void *)(trek_mem_ddr+0x08b870c8),
   (void *)(trek_mem_ddr+0x08b890c8),
};
```

Each core operates on a “slice” of memory

Random cores with synchronized start

```
int trek_microloop_write_check8( void * addrs[], int count, trek_uint8_t pattern){
  int errorCount = 0;
  int ii;
  for ( ii = 0; ii < count; ++ii){
    trek_write8(pattern, addrs[ii]);
  }
  for ( ii = 0; ii < count; ++ii){
    if (trek_read8(addrs[ii]) != pattern) {
      ++errorCount;
      trek_runtime_error("trek_microloop_write_check8", addrs[ii], pattern, trek_read8(addrs[ii]));
    };
  }
  return errorCount;
}
```

```
for ( int ii = 0; ii < 1000; ++ii ){
  errorCount += trek_microloop_write_check8(addrs, 4, 184);
}
```

Each core has free running loop

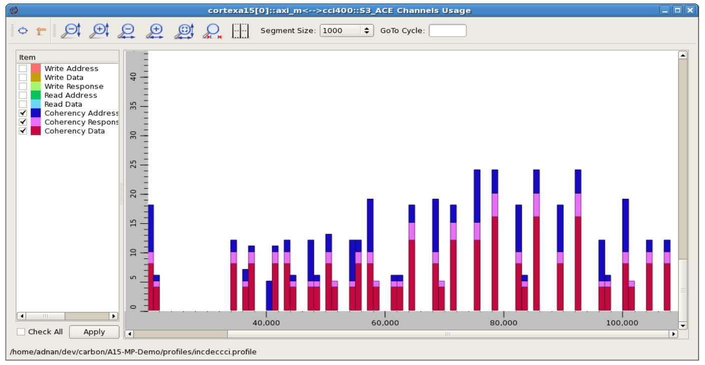
High Coverage and Bug Hunting

Recent examples of bugs discovered in real designs

- RISC-V spec misunderstanding between core vendor and user
- Coherent Mesh Network (CMN) programming issues
- Misconfigured ARM CMN pin to enable coherent traffic
- DDR model unable to handle AXI "wrap" transactions.
- Common cache line access reveals deadlock
- Custom instruction bugs discovered by stress tests
- Results mismatch with ultrawide address strides
- Incorrect exception for guest virtual address[63:38] = 0x1ffffff
- Bad mcause value for guest physical address[63:31] != 0x0

SystemVIP Test Suite Synthesis Coverage Comparison

Typical directed coherency coverage

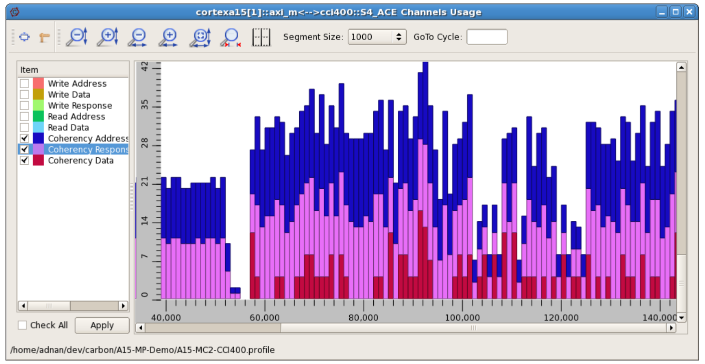

cortexa15[0]::axi_m<-->cci400::S3_ACE Channels Usage

Segment Size: 1000

GoTo Cycle:

/home/adnan/dev/carbon/A15-MP-Demo/profiles/incdeccci.profile

… vs. Breker automated coherency tests


cortexa15[1]::axi_m<-->cci400::S4_ACE Channels Usage

Segment Size: 1000

GoTo Cycle:

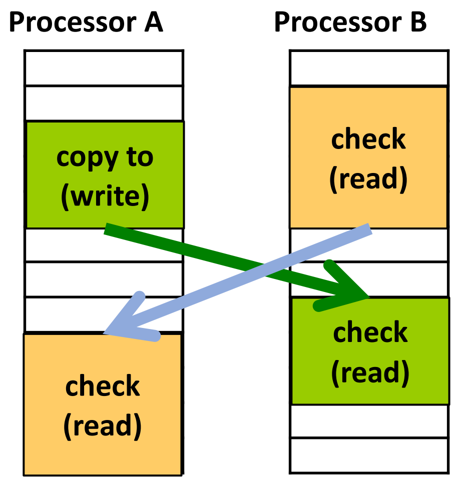
/home/adnan/dev/carbon/A15-MP-Demo/A15-MC2-CCI400.profile

Bug Example: RISC-V spec mis-interpretation

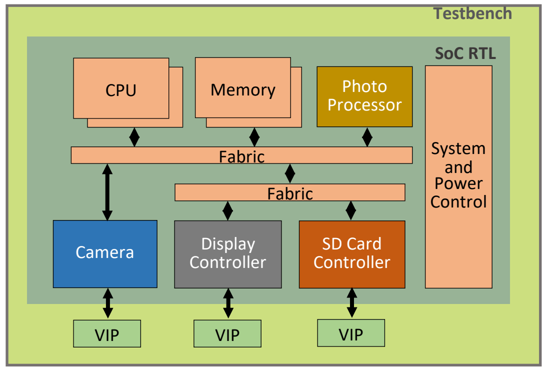
- Design: Customer SoC using a third-party RISC-V processor
- Breker SystemVIP: RISC-V SoC & Coherency TrekApp
- Bug: Weakly ordered memory read-write mismatch on complex load-store
- Test: Combined RISC-V Load Store and Dekker Algorithm
- Reason: Misunderstanding in RISC-V Fence instruction execution
- Resolution: Bug agreed by processor vendor, processor core reissued

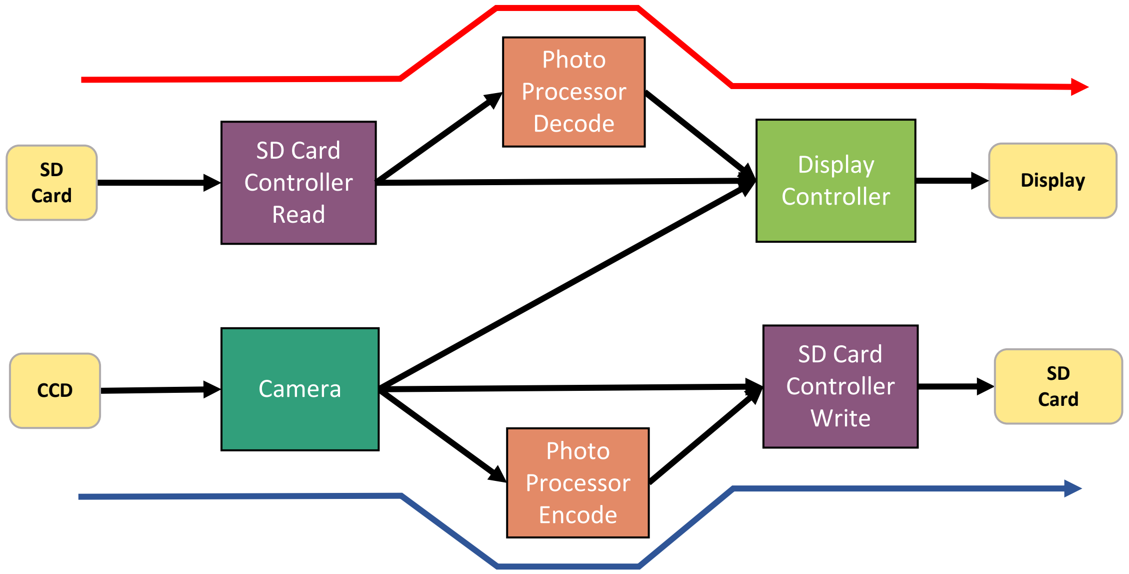
The Dekker Algorithm States

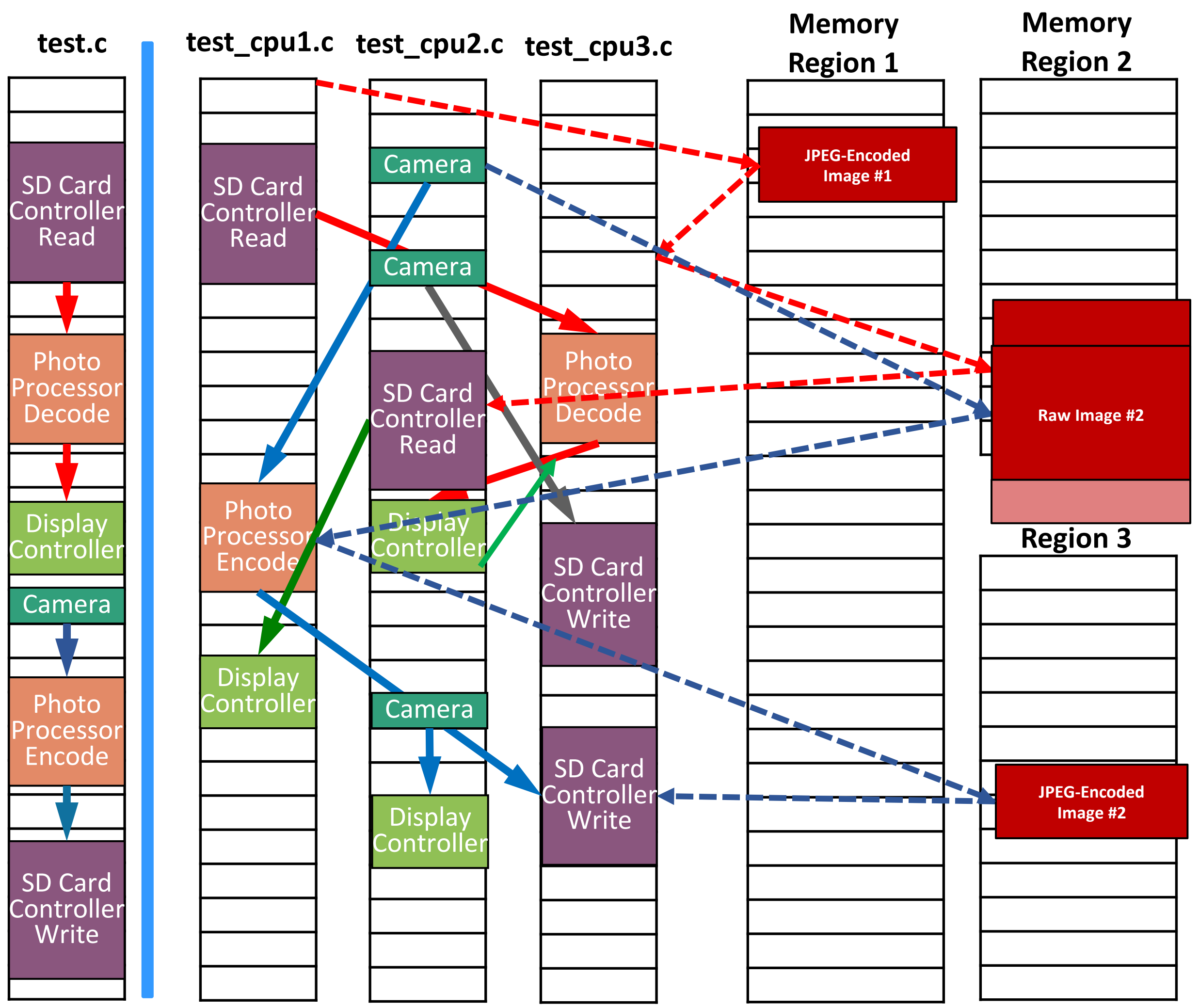
```
    core 0: ST A, 1; MEM_BARRIER; LD B
    core 1: ST B, 1; MEM_BARRIER; LD A
    error iff ( A == 0 && B == 0 )
```



Concurrent Test Execution



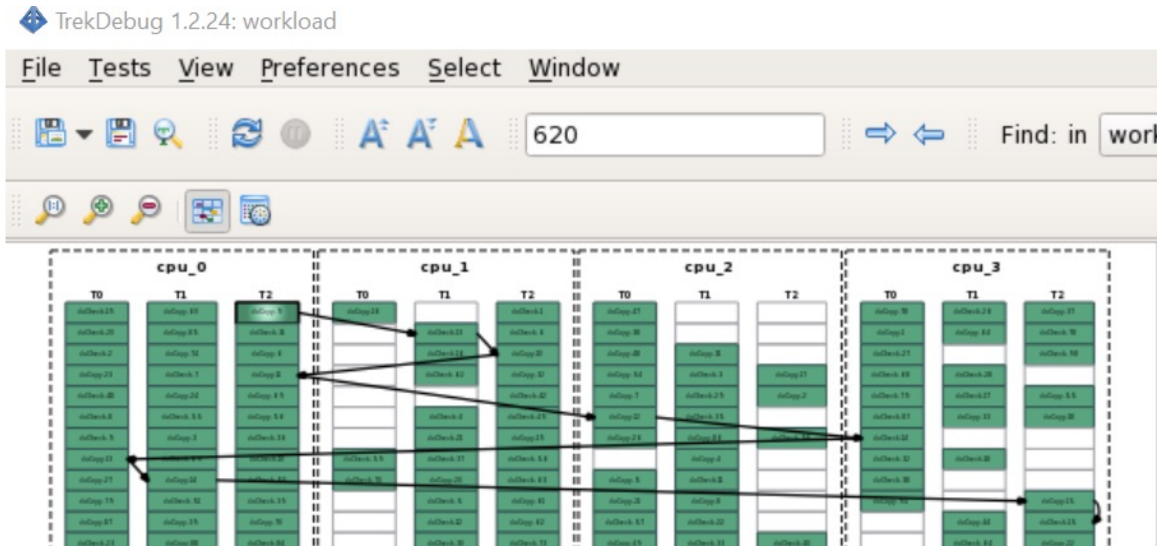




Core-Integrity Example: Multi-Hart (x4), 3 Threads Each

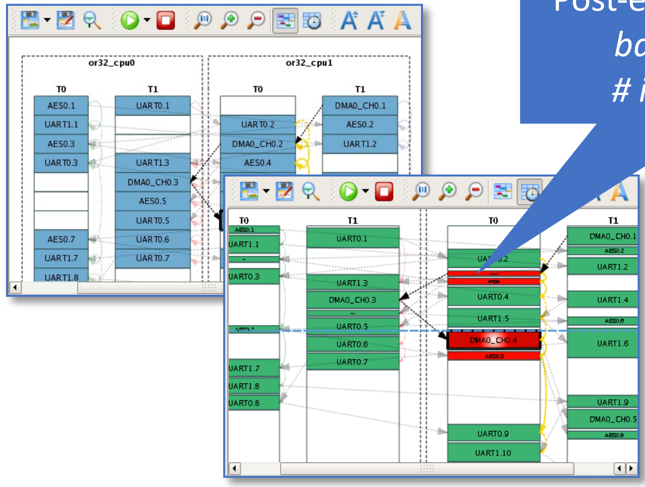
Breker Concurrent Scheduling Stress Tests the Processor/SoC

Advanced, Abstract Debug



Quickly observe concurrent multi-test progress and DUT reaction

Execution Profiling

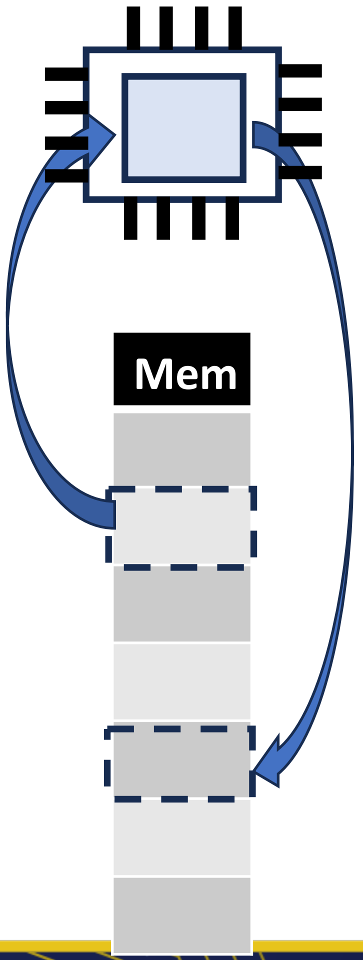


Post-execution test length *based on # clocks, # instructions, etc.*

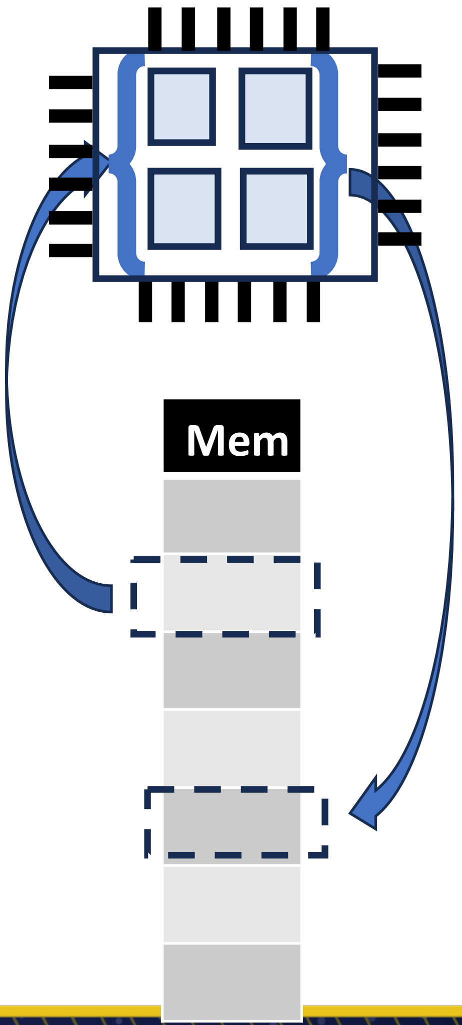
Post-run analysis of design performance/power bottlenecks

Scalability – Going from One to Many Cores

Re-running your test(s) in multi-core designs



Basic Workload Test,
Single Core

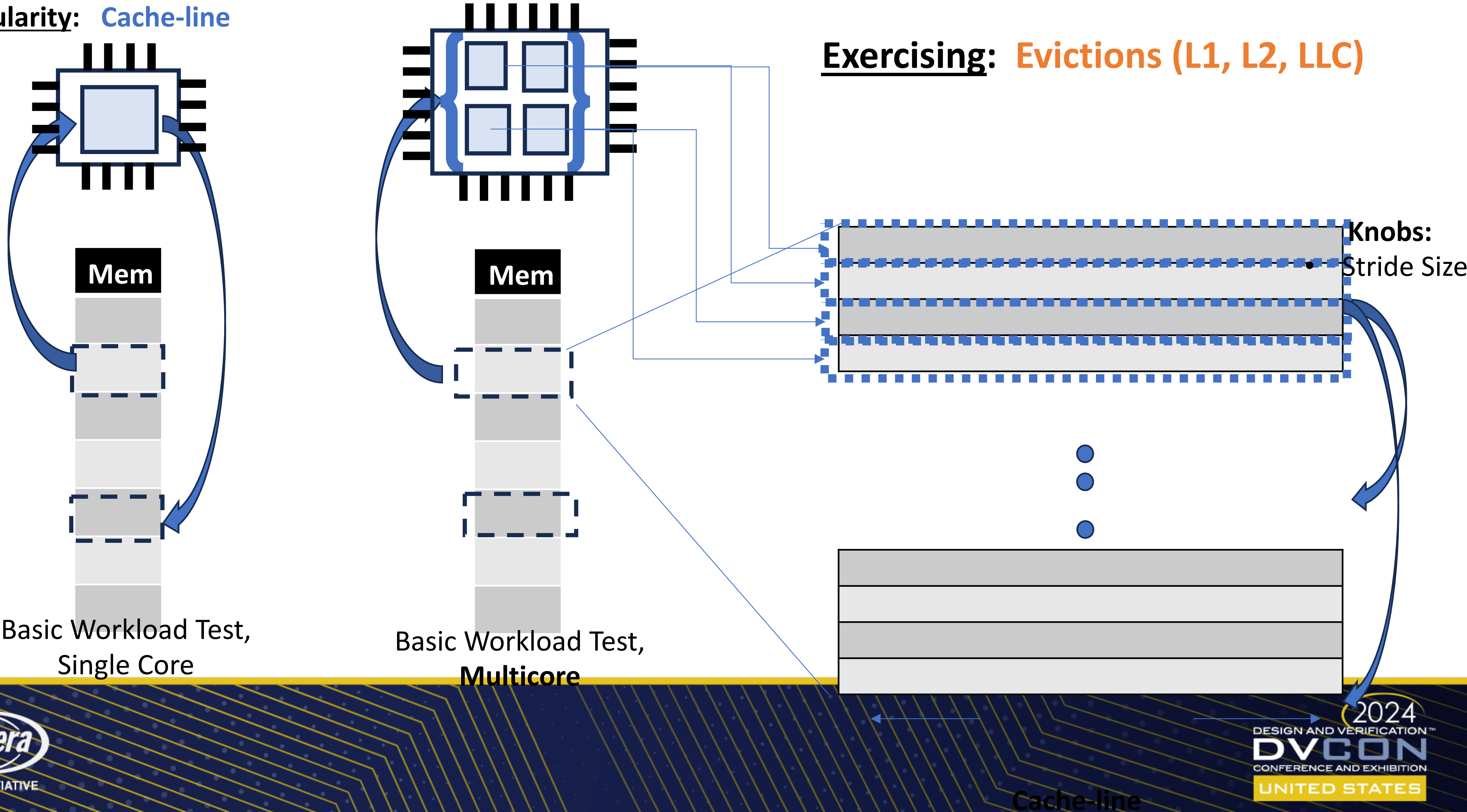


Basic Workload Test,
Multicore

Scalability –What Are the Additional "Knobs" For Multi-Core?

Granularity: Cache-line

Exercising: Evictions (L1, L2, LLC)



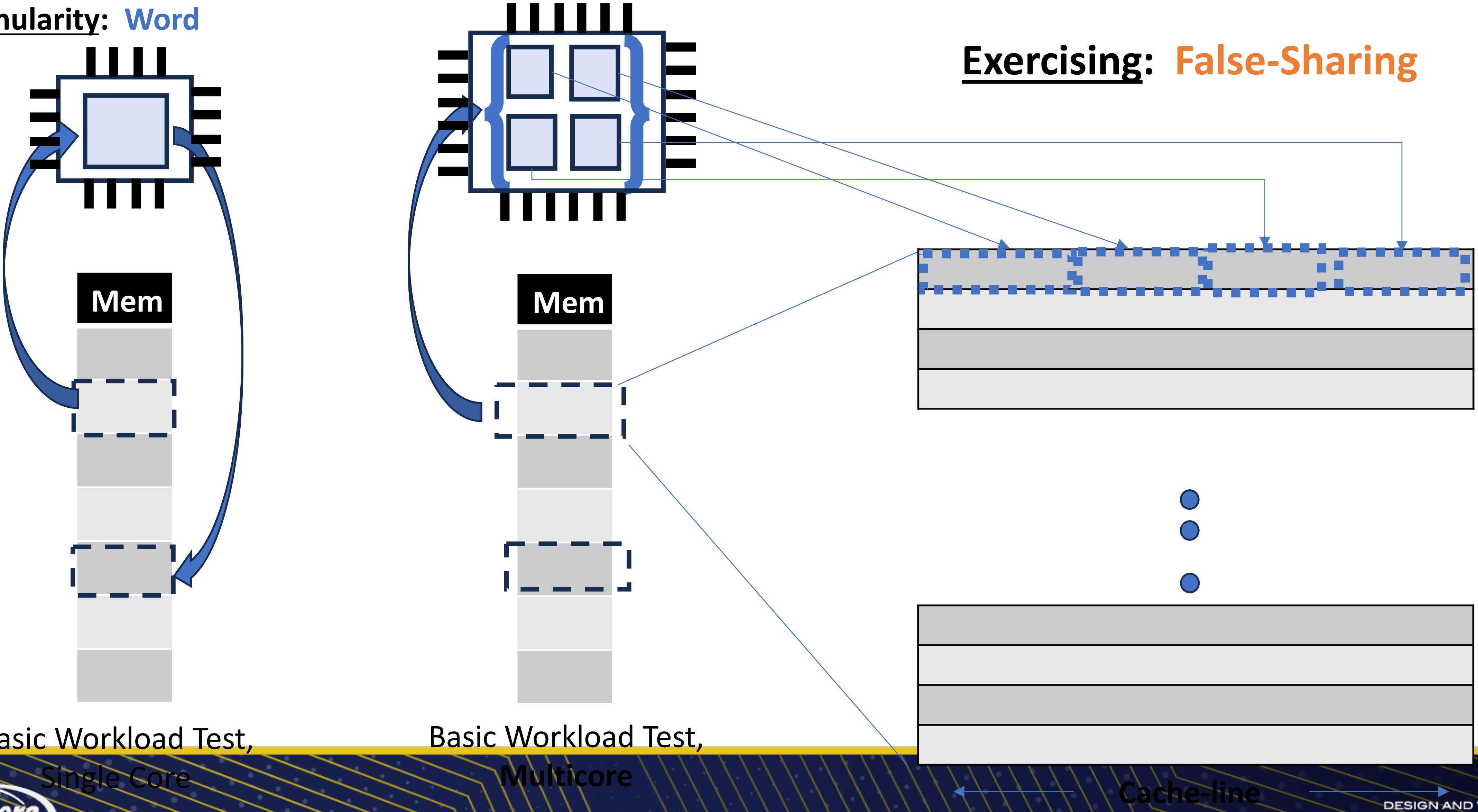
Basic Workload Test, Single Core

Basic Workload Test, **Multicore**

Scalability –What Are the Additional "Knobs" For MultiCore?

Granularity: Word

Exercising: False-Sharing



Basic Workload Test, Single Core

Basic Workload Test, **Multicore**

Thanks for Listening!
Any Questions?

www.brekersystems.com