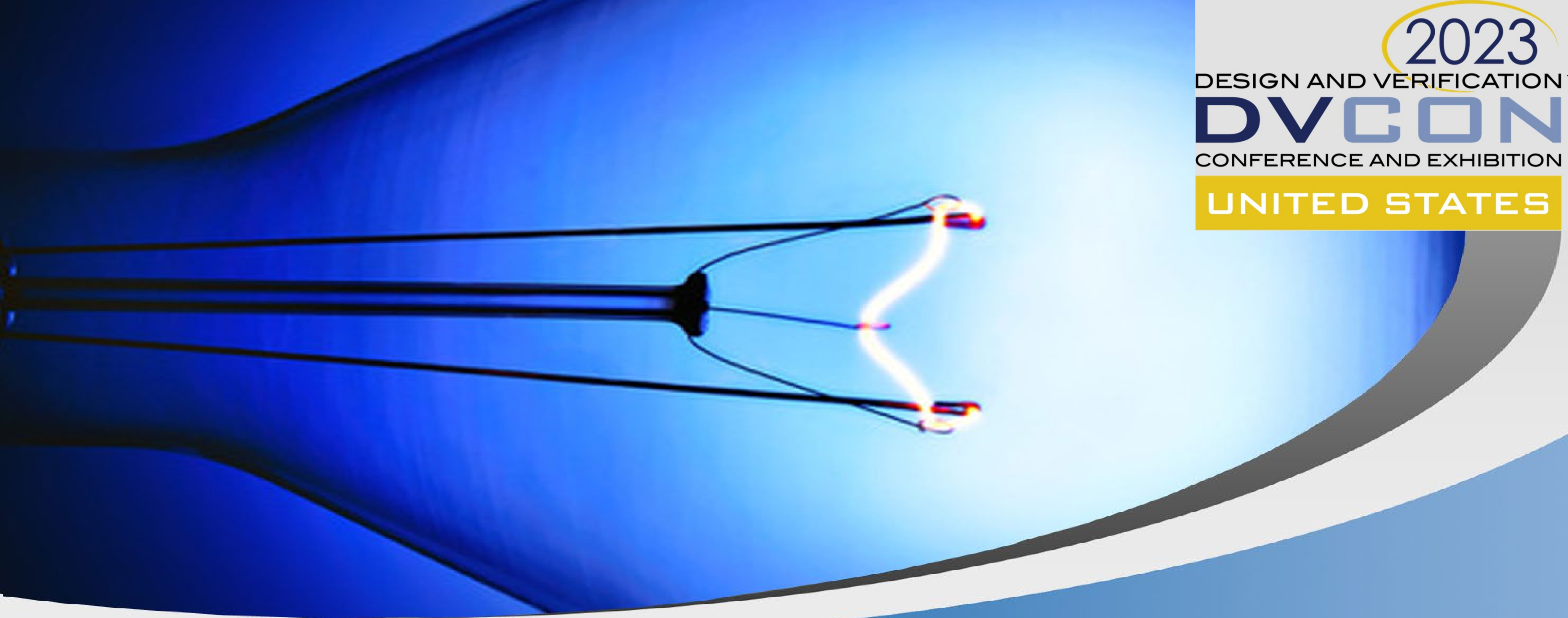


User Experiences with the Portable Stimulus Standard

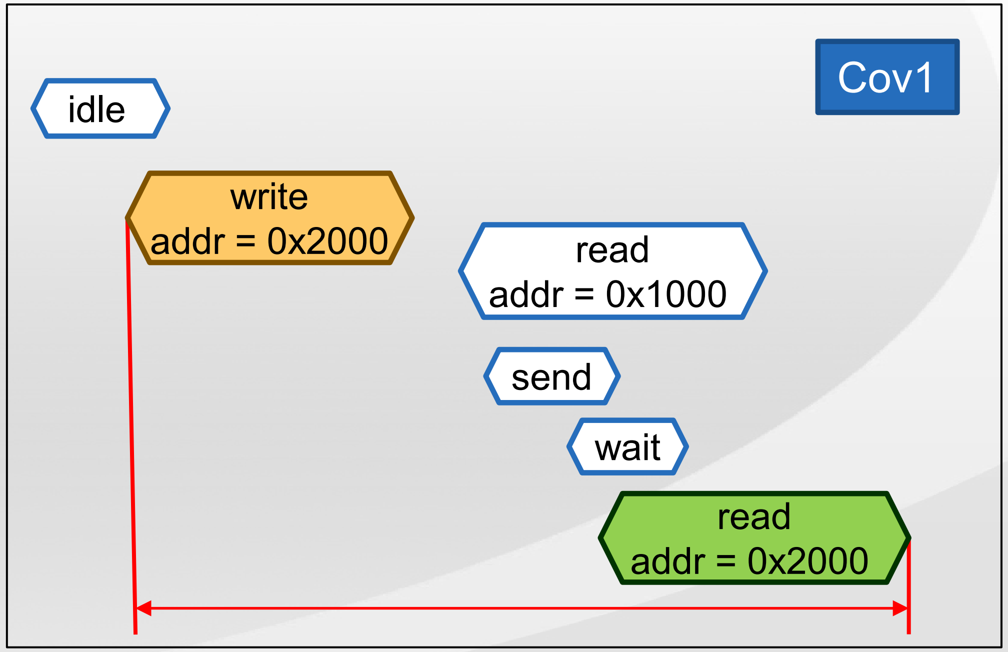
Tom Fitzpatrick, PSWG Vice Chair
Prabhat Gupta, AMD
Mike Chin, Intel

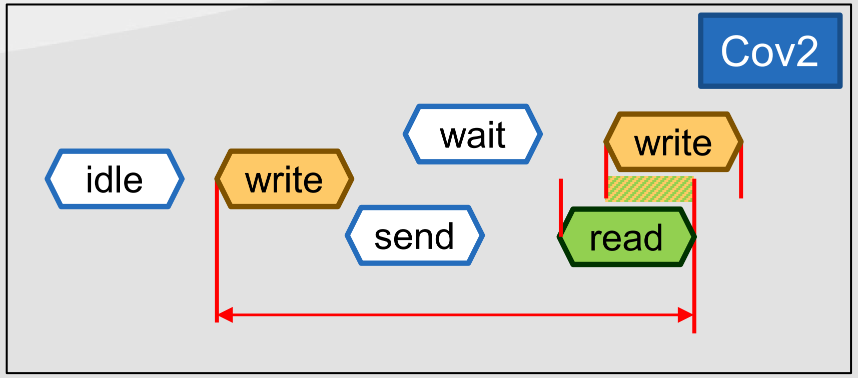
The Biggest Change

Introducing Behavioral Coverage (WIP)

- **Given a stream of action executions, find out whether a given temporal scenario (query) occurs in this stream**
- **The cover statement specifies the interesting scenario**
- **A monitor encapsulates behaviors to be covered**
  - A monitor may be implicit (in a cover statement) or explicit

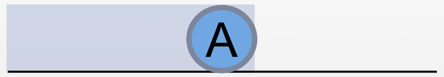
```
action write { rand bit [32] addr; }
action read { rand bit [32] addr; }

monitor read_after_write { write w; read r; activity { w; r; }}

cover Cov1 { read_after_write wr; activity { wr with r.addr == w.addr; }}
cover Cov2 { write w1, w2; read r; activity { w1; r overlaps (w2) ;}
```



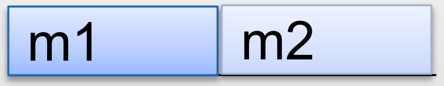


Behavioral Coverage Tracking

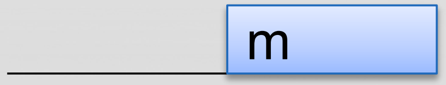
- **Action traversal (e.g., do A with x == 5):**
  - Match nearest action execution matching imposed constraints



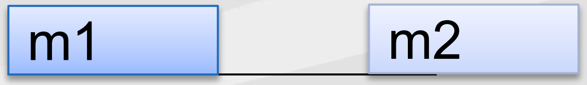
- **Concatenation: concat { m1; m2 }**
  - Match first m2 after m1
    - Match m1 from the current point; from every match point of m1 match m2



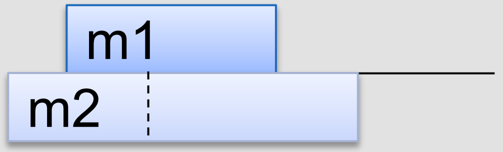
- **Eventually: eventually m**
  - Some time in the future
    - Match m from everywhere, starting from the current point



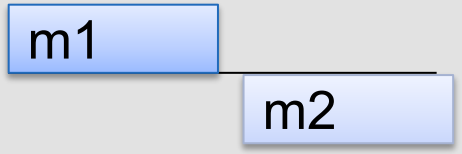
- **Sequence: sequence { m1; m2 }** or **{ m1; m2 }**
  - Match m2 after m1, not necessarily immediately
    - Match m1 from the current point; starting from the match of m1 match m2 from every point
    - Equivalent to concat { m1; eventually m2 }



- **Overlaps: {m1 overlaps m2}**
  - Match m1 if m2 executes at any point while m1 is executing



- **Select: select { m1; m2 }**
  - Match either m1 or m2



Solve/Runtime Messaging

- **New addition to the Core Library**
  - Solve time
    - format
    - print

```
solve function void print_foo(my_struct s) {
  print("The context of the struct is:\n");
  print("value = %d\nname = '%s'\n", s.value, s.name);
}
solve function string get_foo_context_string(my_struct s) {
  return format("value = %d\nname = '%s'\n", s.value, s.name);
}
```

  - Runtime
    - message

```
exec body {
  y = my_func();
  message(FULL, "The values of the variables x and y are: ");
  message(LOW, "%d, %d", x, y);
}
```

  - Either
    - error/fatal

```
package io_pkg { // may change to std_pkg or…
  function void error(string format, type... args);
  function void fatal( int status, string format, type... args);
}
```

returned to calling environment

Solve/Runtime Messaging

- **Solve-Time File I/O**
  - Via file handles

```
package io_pkg {
  typedef chandle file_handle_t;
  static const file_handle_t nullhandle = /* implementation-specific */;
  enum file_option_e {TRUNCATE, APPEND, READ};
  function file_handle_t file_open(string filename, file_option_e opt = TRUNCATE);
  function void file_close(file_handle_t file_handle);
  function bool file_exists(string filename);
  function void file_write(file_handle_t file_handle, string format, type... args);
  function string file_read(file_handle_t file_handle, int size = -1);
```

  - Single Functions

```
  function void file_write_lines(string filename, list<string> lines,
                                 file_option_e opt = TRUNCATE);
  function list<string> file_read_lines(string filename);}
```

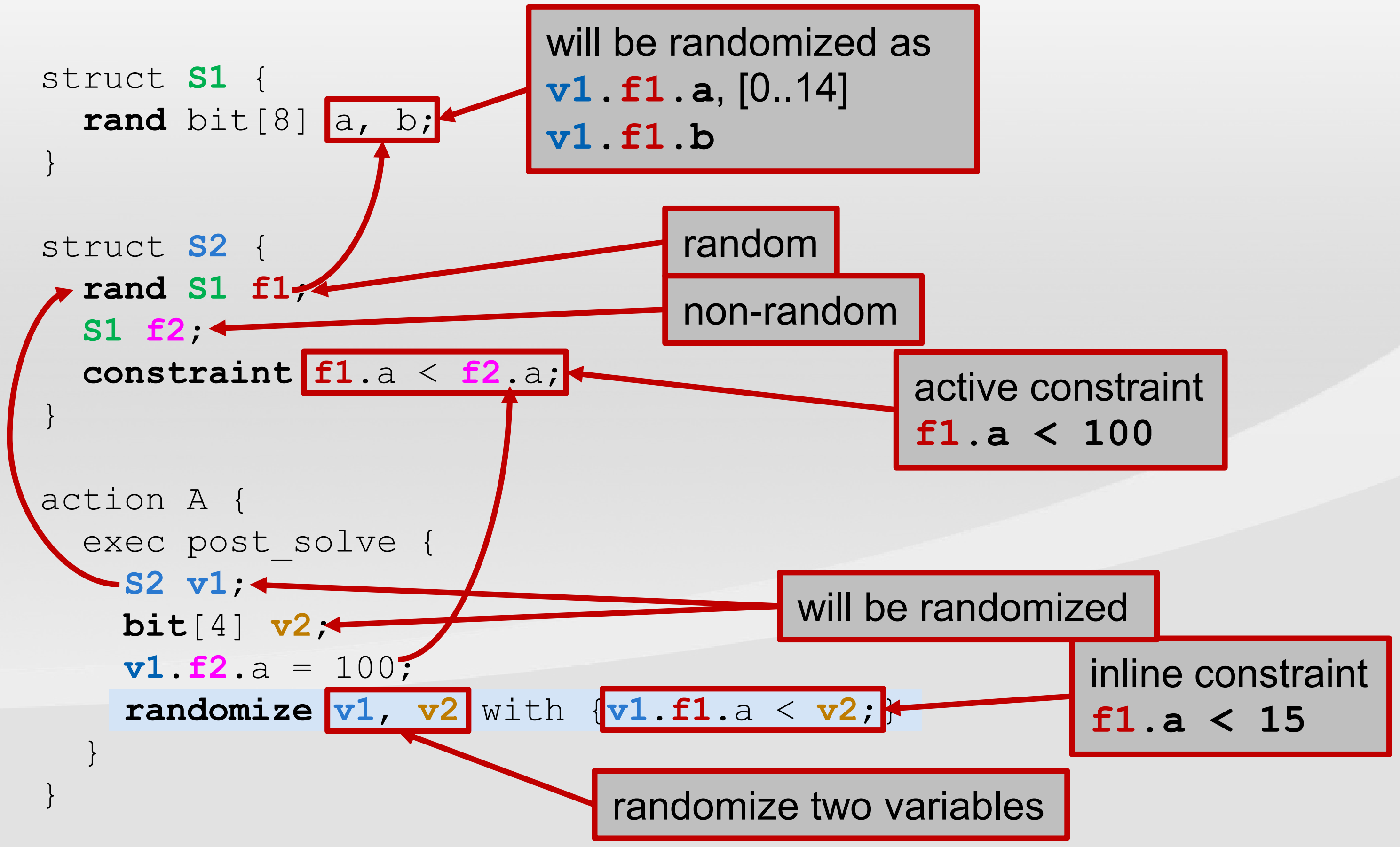
List randomization

- **Lists can now be declared *rand***
  - Randomized when its container is randomized
  - Just like other *rand* fields
- **The *size* is considered a state variable**
  - *size* cannot be constrained directly

```
struct S {
  rand list<bit[8]> lst;
  exec pre_solve { // Initialize the list
    repeat (100) {
      lst.push_back(0);
    }
  }
  constraint {lst.size() in [4..100]; // Error: illegal constraint on size
    foreach (lst[i]) {
      lst[i] == i+lst.size(); // OK: size is a state variable in foreach
    }
  }
}
```

Procedural randomization statement

- **Allowed in solve exec blocks**
- **Subset allowed in target exec**
  - No struct randomization
- **Also supports**
  - urandom
  - urandom_range



will be randomized as v1.f1.a, [0..14] v1.f1.b
random
non-random
active constraint f1.a < 100
will be randomized
inline constraint f1.a < 15
randomize two variables

```
struct S1 {
  rand bit[8] a, b;
}

struct S2 {
  rand S1 f1;
  S1 f2;
  constraint f1.a < f2.a;
}

action A {
  exec post_solve {
    S2 v1;
    bit[4] v2;
    v1.f2.a = 100;
    randomize v1, v2 with {v1.f1.a < v2;}
  }
}
```

Dist randomization constraint

- **Similar to SystemVerilog**
  - `:=` assigns a specific weight
  - `:/` distributes weight across a list
- **Subject to other constraints**
  - `dist` only biases values in the legal range

```
struct S {
  rand bit[32]      x;
  bit               y;
  constraint dist x in [100..102 := 1, 200 := 2, 300 := 5];
```

```
100:1, 101:1, 102:1, 200:2,    300:5
```

```
  constraint dist x in [100..102 :/ 1, 200 := 2, 300 := 5];
```

```
100:1/3
101:1/3,        200:2,     300:5
102:1/3
```

```
  constraint dist x in [100..102 := 1, 200 := 2, 300 := 5];
  constraint (y==1) -> x > 300;
}
```

Constraint causes **`dist`** to be ignored

Labels as action handles

- **Labels as action handles**
  - Create handle for anonymous action traversal
  - Can be referenced from above

```
action mem2mem_chain {
    activity {
      do mem_c::load_buff;
      repeat (10) {
        select {
          xfer: do dma_c::mem2mem_xfer;
          cpy:  do cpu_c::memcpy;
        }
      }
    }
  }

  action my_test {
    activity {
      do mem2mem_chain with { xfer.size > 10; };
    }
  }
```

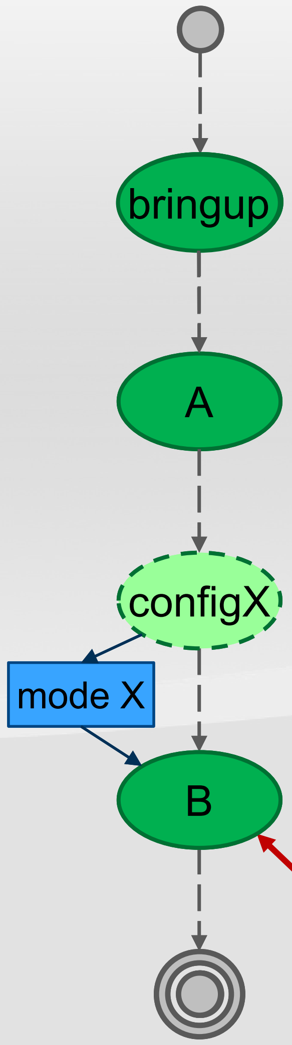
dma_c::mem2mem_xfer

Inference Issue

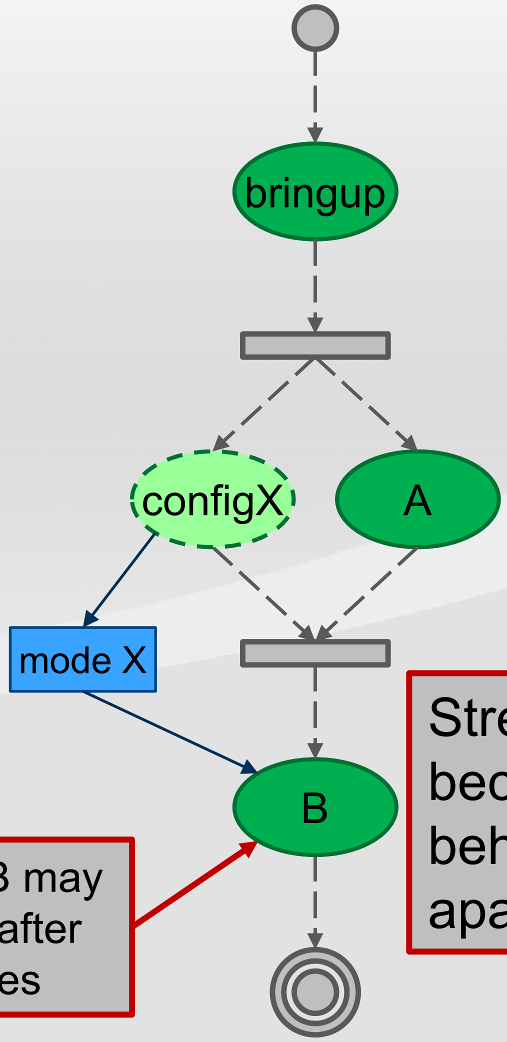
```
state config_s {
   rand mode_e mode;
}

action configX {
  output config_s out_cfg;
  constraint out_cfg == X;
}

action B {
  input config_s cfg;
  constraint cfg.mode == X;
}
```

```
action my_stress_seq {
  activity {
    do bringup;
    do A;
    do B;
  }
}
```

B should start immediately after A completes – it's a critical aspect of the intent



In reality B may start long after A completes



Stress is not achieved because relevant behavior is spaced apart or diluted

Inference For "Atomic" Block

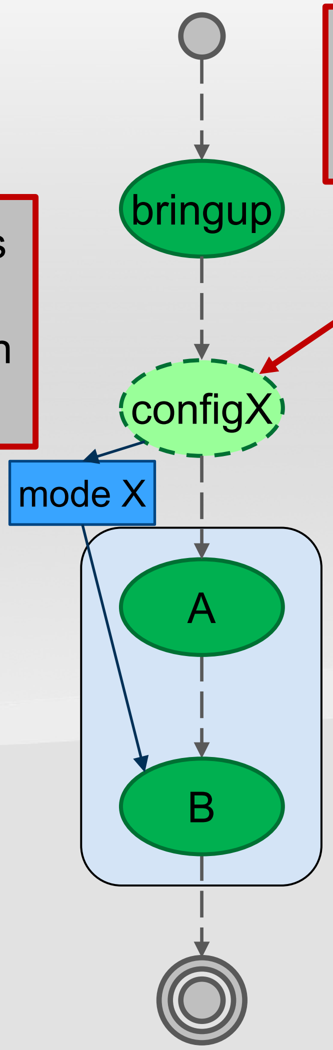
The **`atomic`** activity block guarantees that the invocation of each action does not wait for any action other than its predecessor in the sequence

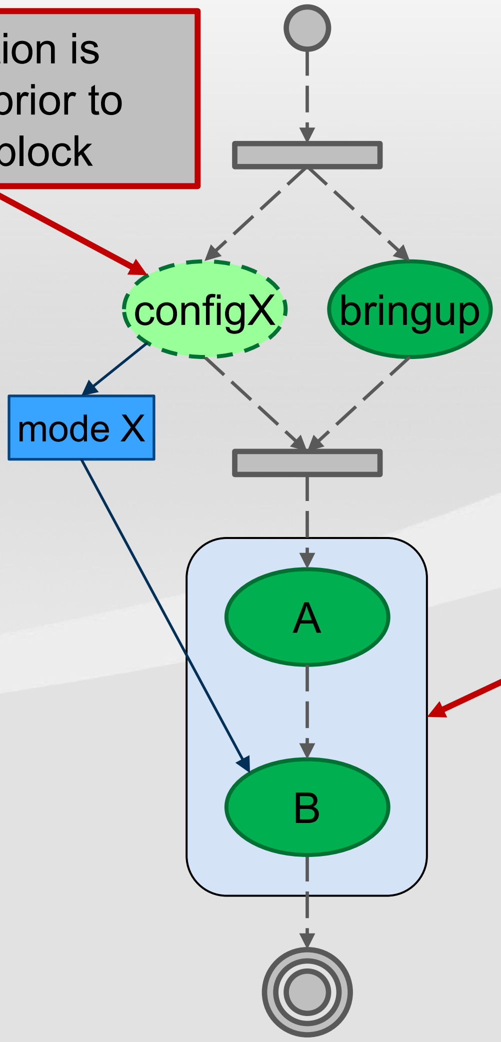
```
action my_stress_seq {
  activity {
    do bringup;
    atomic {
      do A;
      do B;
    }
  }
}
```

Any dependency of B is a dependency of the entire block

Inferred action is scheduled prior to the atomic block

Atomic block doesn't start until all its dependencies are met





Concise read-modify-write

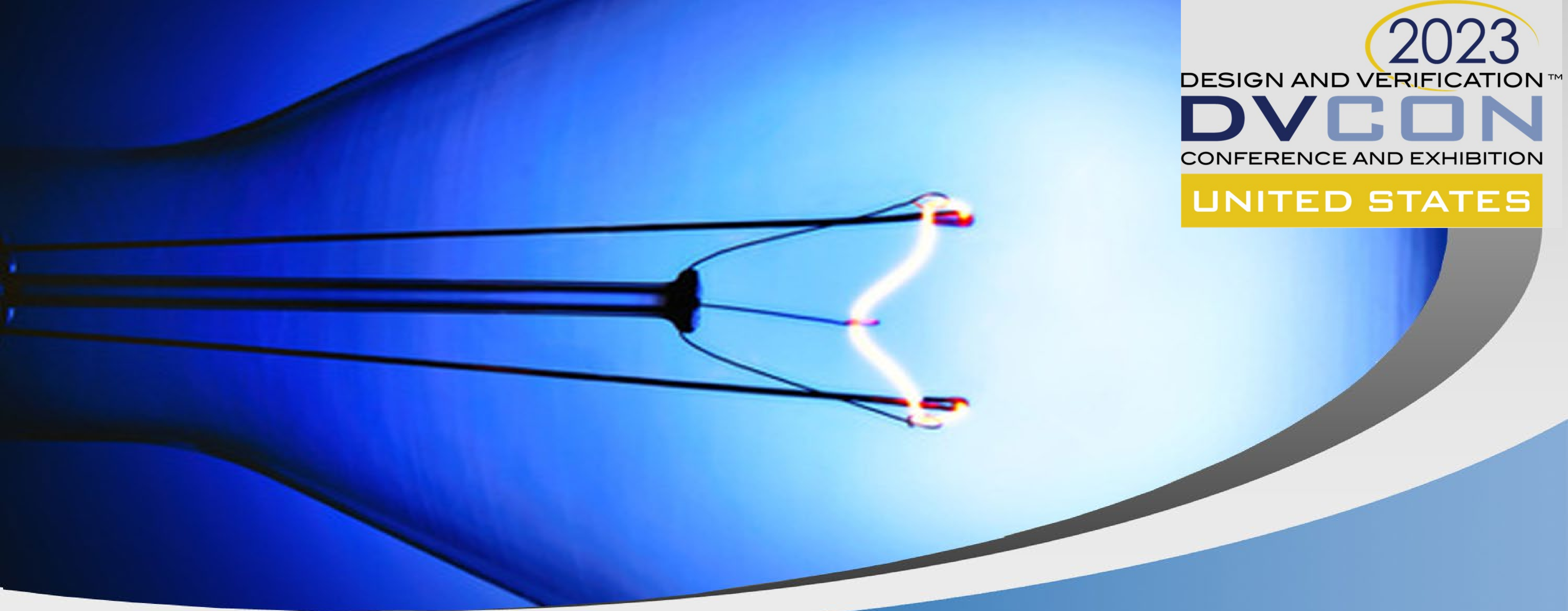
- **Added a way to read-modify-write in a single operation**

```
pure component reg_c < type R,
  reg_access ACC = READWRITE,
  int SZ = (8*sizeof_s<R>::nbytes)> {
    function R read();

    function void write(R r);

    function bit[SZ] read_val();

    function void write_val(bit[SZ] r);

    function void write_masked(R mask, R val);
    function void write_val_masked(bit[SZ] mask,
                                   bit[SZ] val);
    function void write_field(string name,
                              bit[SZ] val);
    function void write_fields(list<string> names,
                               list<bit[SZ]> vals);
}
```

```
struct CR : packed_s<> {
  bit         en;
  bit[11]     pad;
  bit[4]      mode;
  bit[16]     coeff;
}

component dut_c {
  dut_regs_c        regs;

  action cfg_a {
    rand bit[4]  mode;
    rand bit[16] coeff;
    exec body {
      comp.regs.cr.write_masked(
         {.mode=~0, .coeff=~0}, {.mode=mode, .coeff=coeff});

      comp.regs.cr.write_val_masked(0xFFFFF000,
                               (coeff << 16) | (mode << 12));

      comp.regs.cr.write_fields({"mode", "coeff"},
                               {mode, coeff});
    }
  }
}
```

Additional New Features

- **Static functions in components**
  - Function declaration associated with the component *type* (not *instance*)
    - Called via the "::" scope operator
- **Tag/Addr_value**
- **Enum base type**
- **Floating Point computation and storage types**



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

AI Engine Subsystem Verification With PSS

AMD together we advance_

Prabhat Gupta - AMD

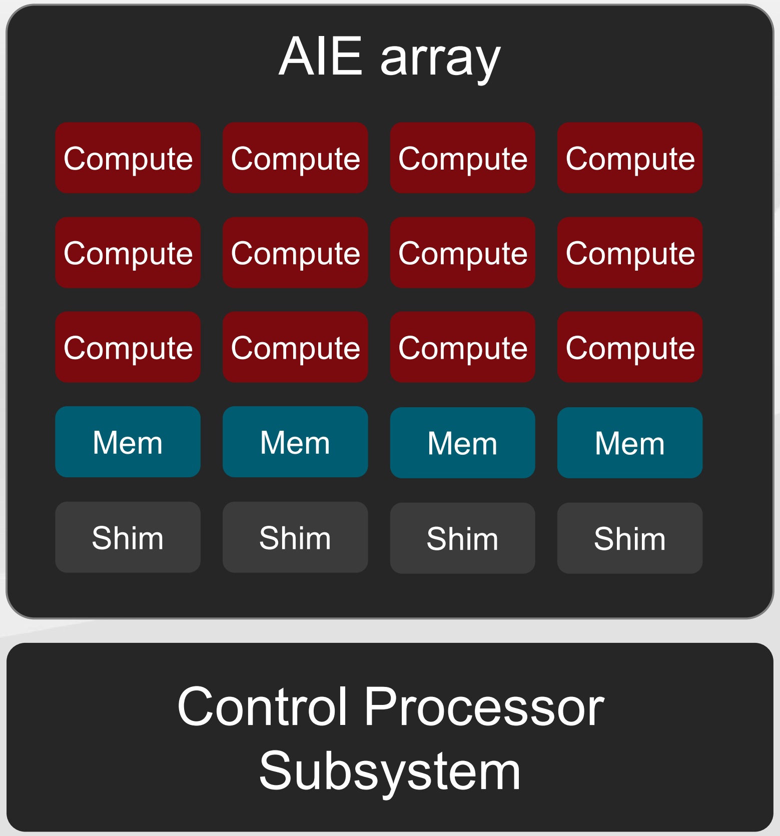
accellera SYSTEMS INITIATIVE

Agenda

- Introduction
- PSS initial approach – tackle portability
- First-class PSS model
- Constraints
- PSS Methodology
- Conclusion

Introduction

- **Verification of a grid of general-purpose AI Engines**
  - AI Engine information is available at https://www.xilinx.com/products/technology/ai-engine.html
- **Example use cases of AI Engine array**
  - Auto framing and eye gaze correction with Microsoft Teams
  - Low power background blur, audio noise reduction, etc.
- **Highly configurable network fabric**
- **Compute, Memory, and Shim engines**
- **Connected to SoC with a high-speed on-chip data network and pervasive control network**
- **Grid of engines and highly configurable network makes creating functional, performance, and post-silicon validation tests very challenging**

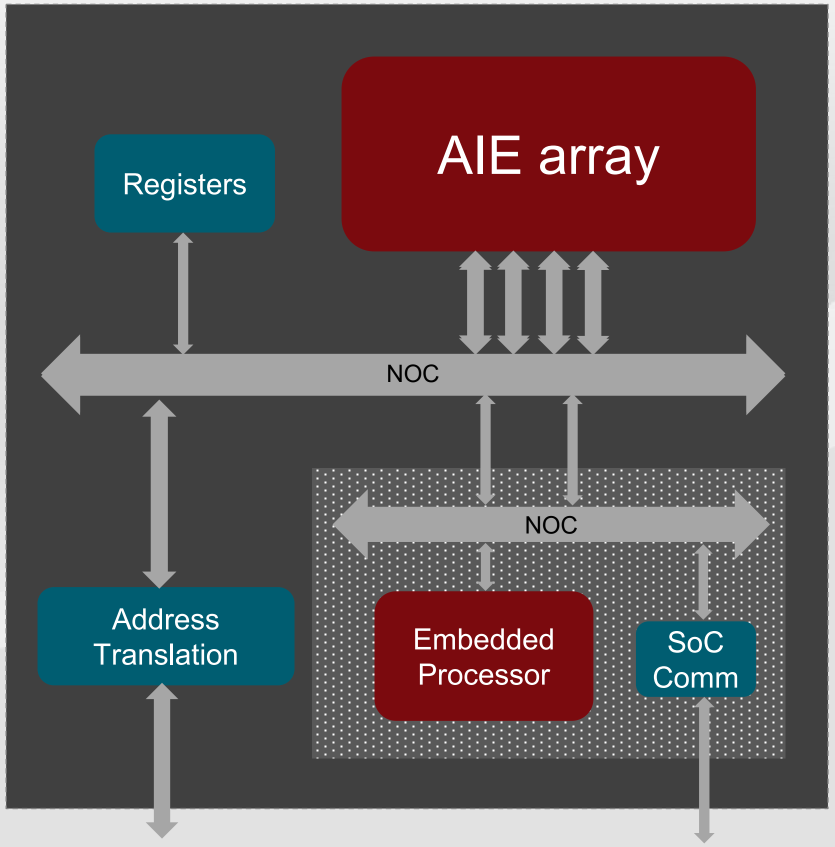

AIE array
Compute Compute Compute Compute
Compute Compute Compute Compute
Compute Compute Compute Compute
Mem Mem Mem Mem
Shim Shim Shim Shim
Control Processor Subsystem

Circuit-switched DMA

- More than one channel going east-west or north-south
- Multi-hop acyclic paths for DMAs
- Adjacent columns can be isolated as a group
- DMAs can’t cross the isolation boundary

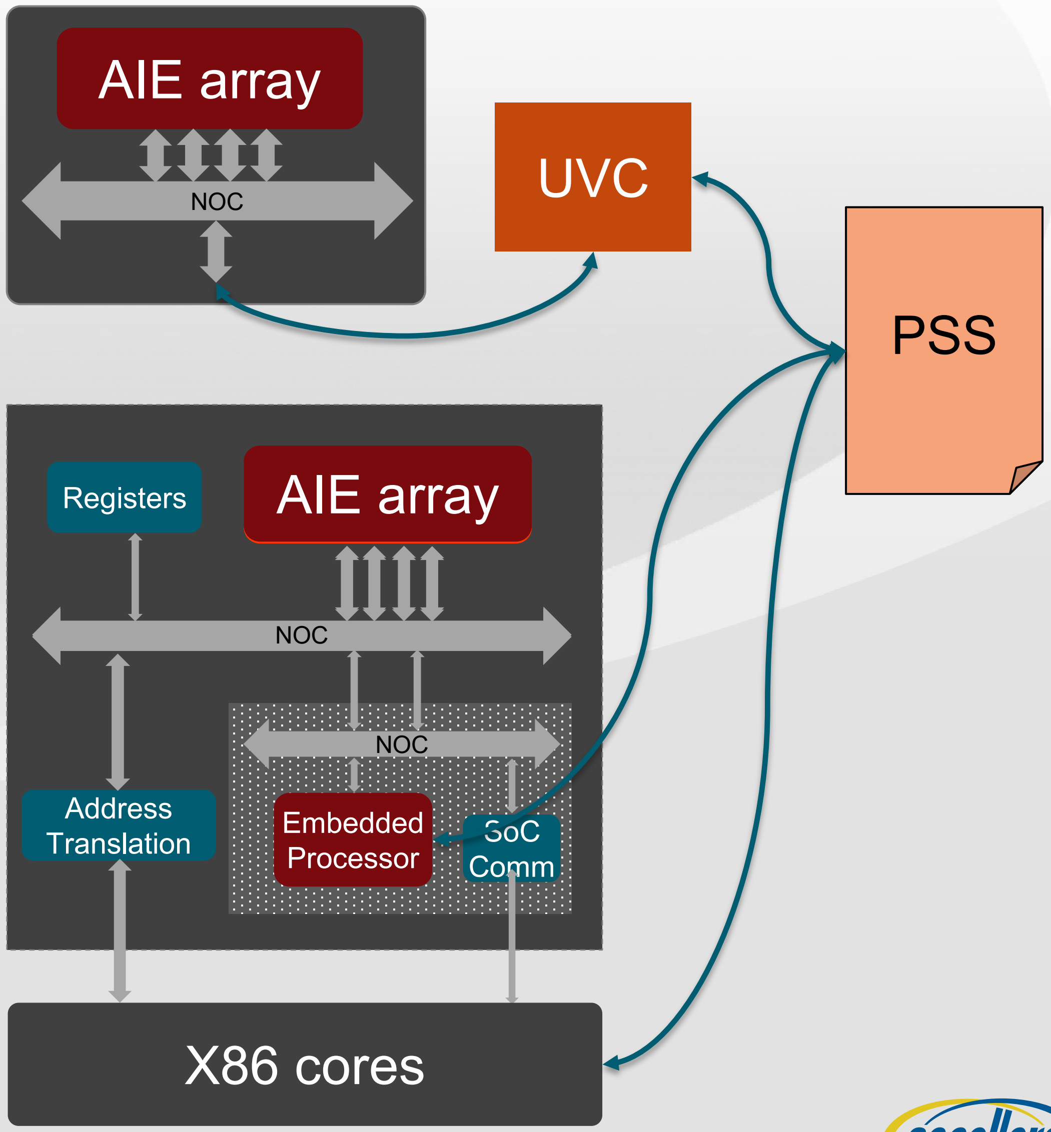
AI sub-system

- AI Array with embedded control processor
- AI array can be subdivided for isolated parallel work from different applications
- Host processor sends work to embedded processor
- Embedded processor sends compute kernels to AIE array



Testbench environments

- **IP only**
  - AIE array UVM/C++ environment
- **Subsystem**
  - AIE array with an embedded processor running C code with UVM/C++ environment
- **Full SoC**
  - X86 processors, AIE subsystem, and rest of chip
  - C code running on x86 and embedded processor
- **Post-silicon bring-up and validation**
  - C code running on the main and embedded processor



Verification Challenges

- **AIE Array**
  - Many possible paths through the AI engine array
  - Compute array can be configured into different partitions, which impacts routing
  - Need to exercise many combinations to verify routing/arbitration throughput/latency
    - Legal pseudo-random paths for routing
  - Challenging to account for possible parallel DMAs without a lot of procedural code in a complex test
  - Lots of boilerplate code for new tests
- **Subsystem**
  - Need to verify new interfaces
  - Most tests in UVM, take advantage of randomization/coverage with the embedded processor in bypass
  - A lot of boilerplate code for each test
  - Some bare-metal content for the embedded processor, but it’s a huge barrier, not usually done
- **SoC**
  - Need bare-metal test content, but only have UVM content
  - Must develop multi-core tests to exercise key cases → multicore is always a challenge
  - Synchronization across AIE and other IP are challenging

The value – portability

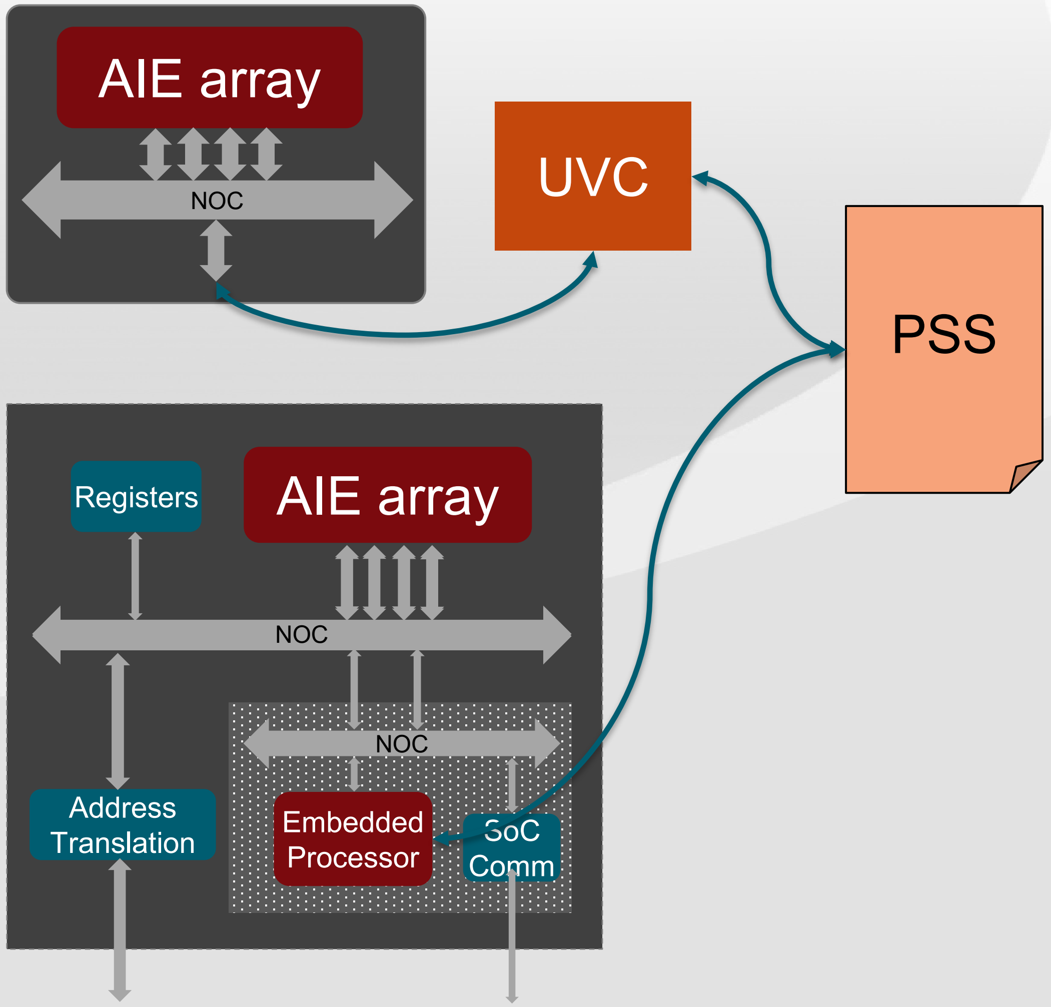
PSS FIRST STEPS

An AIE Test

- **Power up and bring the AIE array out of the reset with the UVM testbench**
- **Boot AIE array**
  - Initialization sequence in PSS
- **Configure AIE array isolation with PSS**
  - Adjacent columns isolated to work on different applications
  - Find a valid random isolation setup, that could be user-supplied, and do isolation programming
- **Configure acyclic circuit-switched DMA circuits with PSS**
  - Find N random valid circuits across all isolation units, program the circuits
- **Generate traffic**
  - Run M parallel DMAs and check the results
  - Select valid routes from the last step
- **Repeat with new isolation setup**

PSS first steps for AIE project

- **UVM to PSS – started at IP level**
  - Mostly procedural code converted from UVM to PSS
  - Fixed column isolation setup and DMA circuits
  - PSS address spaces and registers
- **Portability to the embedded control processor (CP)**
  - Address space adapted to CP address map and TLBs
- **Value**
  - NOC randomization
  - New tests running on the control processor
  - Test available for post-silicon bring up

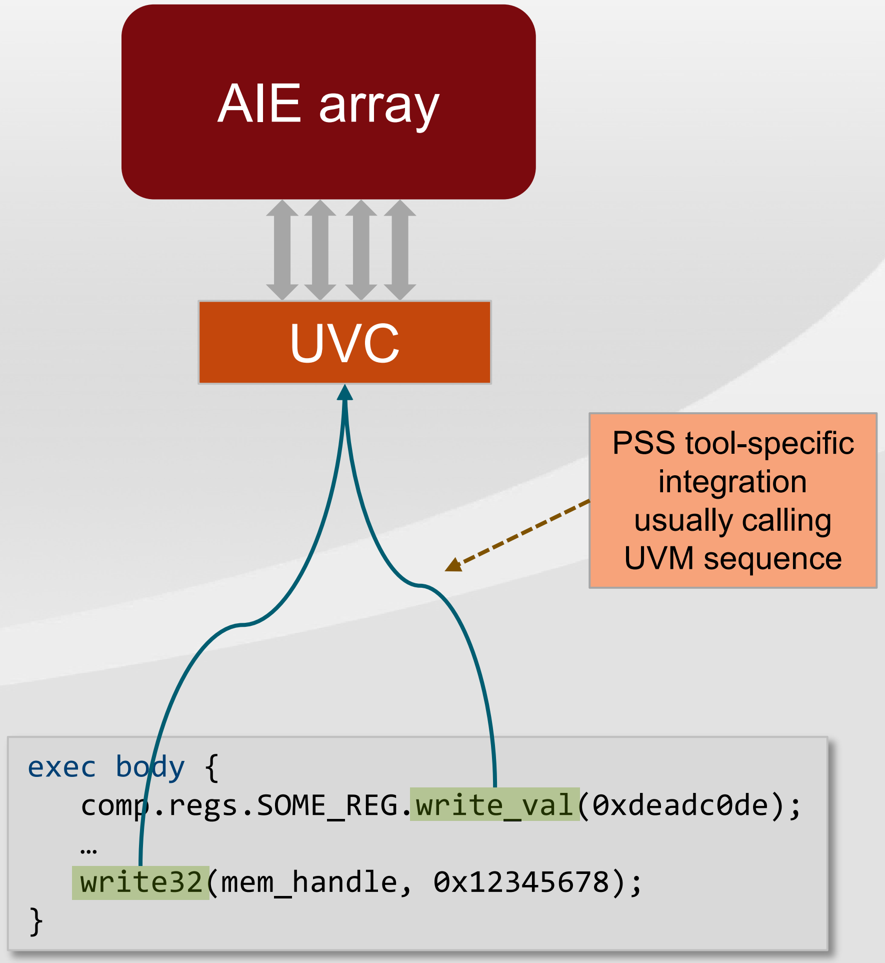


How does the PSS portability work?

PSS MODEL INTEGRATION

AIE IP PSS Integration

```
component pss_top {
  transparent_addr_space_c<aie_mem_trait_s> mem;
  transparent_addr_space_c<mem_trait_s>     sysmem;

  exec init_down {
    // Add PSS executors to map to UVCs
    // May have some tool-specific setup for integration
    // Call to add_executor
    ...

    // Memory setup
    repeat(col: COMPUTE_TILE_COLS) {
      repeat (row: COMPUTE_TILE_ROWS) {
        transparent_addr_region_s<aie_mem_trait_s> tile_region;

        aie_tile_region.size = COMPUTE_MEM_SIZE;
        aie_tile_region.addr = compute_tile_base(row, col);

        aie_tile_region.trait.mem_block = COMPUTE_TILE;
        aie_tile_region.trait.row = row + COMPUTE_TILE_START_ROW;
        aie_tile_region.trait.col = col;

        (void)mem.add_region(tile_region);
      };
    };

    transparent_addr_region_s<mem_trait_s> sysmem_region;
    (void)sysmem.add_region(sysmem_region);
    ...
```



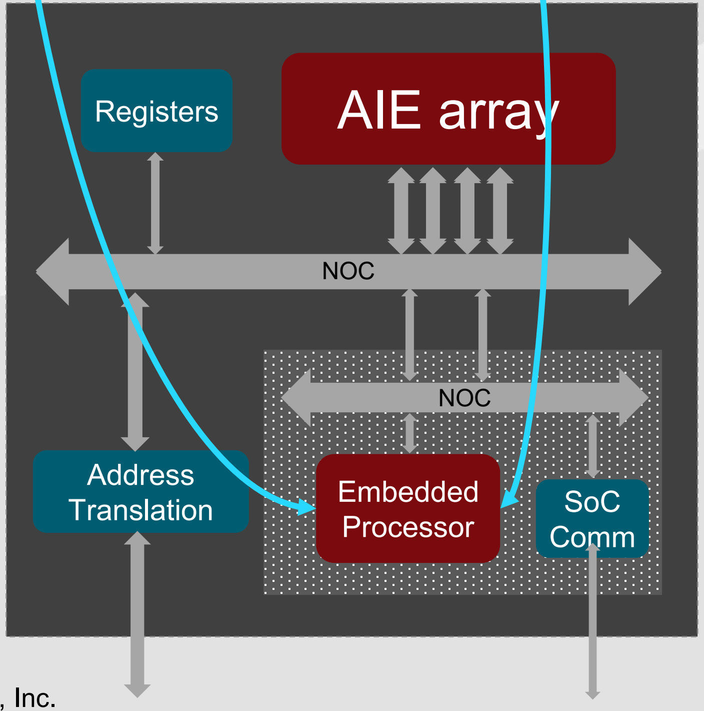
```
exec body {
   comp.regs.SOME_REG.write_val(0xdeadc0de);
   …
   write32(mem_handle, 0x12345678);
}
```

AIE Subsystem PSS integration

```
component pss_top {
  transparent_addr_space_c<aie_mem_trait_s> mem;
  transparent_addr_space_c<mem_trait_s>     sysmem;

  exec init_down {
    // Add PSS executors to map to Embedded Processor
    // May have some tool-specific setup for integration
    // Call to add_executor
    ...

    // Memory setup
     repeat(col: COMPUTE_TILE_COLS) {
      repeat (row: COMPUTE_TILE_ROWS) {
        transparent_addr_region_s<mem_trait_s> tile_region;

        aie_tile_region.size = COMPUTE_MEM_SIZE;
        aie_tile_region.addr = TLB(compute_tile_base(row, col));

        aie_tile_region.trait.mem_block = COMPUTE_TILE;
        aie_tile_region.trait.row = row + COMPUTE_TILE_START_ROW;
        aie_tile_region.trait.col = col;

        (void)mem.add_region(tile_region);
      };
    };
    transparent_addr_region_s<mem_trait_s> sysmem_region;
    (void)sysmem.add_region(sysmem_region);
    ...
  }
}
```

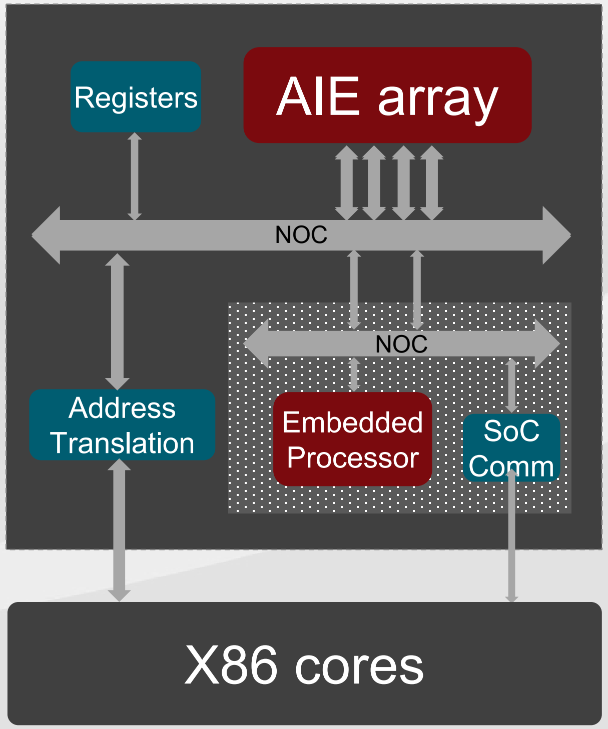
```
exec body {
   comp.regs.SOME_REG.write_val(0xdeadc0de);
   …
   write32(mem_handle, 0x12345678);
}
```

```
exec body {
   *(volatile uint32_t*) SOME_REG_ptr = 0xdeadc0de;
   *(volatile uint32_t*) mem_handle_ptr = 0x12345678;
}
```



SoC PSS integration

- **Multiple executors**
  - One embedded processor and a few x86 processors
- **Two test compilation units**
  - Tool-specific setup to generate test code for x86 and embedded control processor
- **PSS action synchronization across executors**
  - Tool-specific implementation, usually memory-based mailboxes
- **PSS Memory setup**
  - Shared system memory
  - Local AIE memories



How it should be

FIRST-CLASS PSS MODEL

Desired PSS model capabilities/example tests

PSS model capabilities

- Setup random isolation groups
- Setup multiple acyclic circuits in an isolation group
- Multiple parallel DMAs with data checks
- Enable inference for simple test writer interface
- A simple PSS Test API that allows random and directed tests

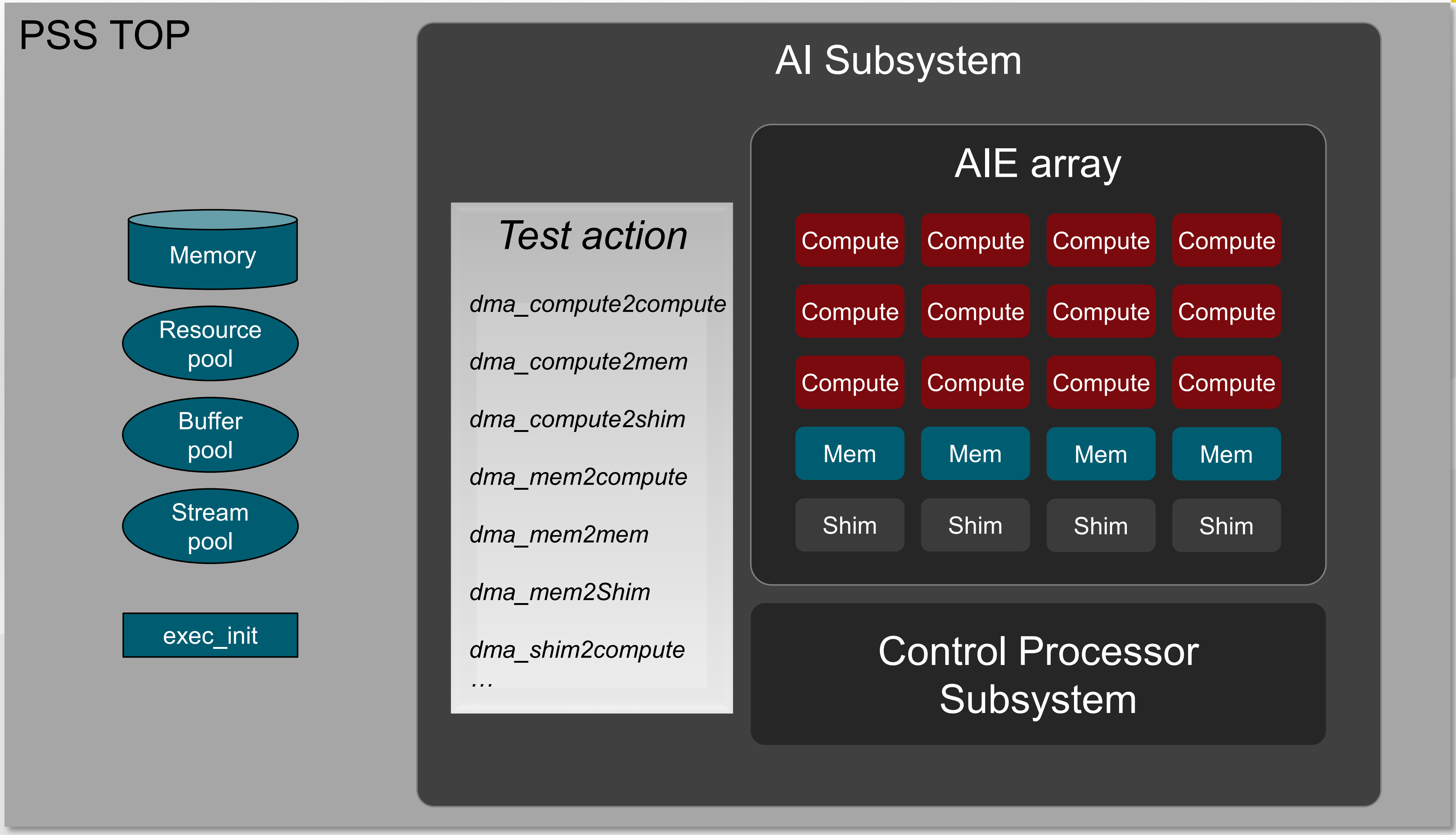
Example tests

- N parallel DMAs with random circuits in a random isolation setup
- N parallel DMAs with random circuits in an isolation setup with all columns as one group
- Maximum parallel DMAs to system memory
- Validate every point-to-point path in the AIE grid

```
action aie_dma_one_group {
  activity {

    do setup_isolation with {
      out_iso_state.num_iso_groups == 1;
    }

    parallel {
      replicate (N) { do aie_c::dma; }
    };
  };
};
```

```
action aie_dma {

  activity {
    parallel {
      replicate (N) { do aie_c::dma; }
    };
  };
};
```

PSS model


PSS TOP
Memory
Resource pool
Buffer pool
Stream pool
exec_init
AI Subsystem
Test action
dma_compute2compute
dma_compute2mem
dma_compute2shim
dma_mem2compute
dma_mem2mem
dma_mem2Shim
dma_shim2compute
…
AIE array
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Mem
Mem
Mem
Mem
Shim
Shim
Shim
Shim
Control Processor Subsystem

PSS model

```
component aie_tile <int NO_DMA,int NO_BD>
{
   pool [NO_DMA] dma_chan_s chan_pool;

   action configure_isolation {}

   action configure_switch {}

   action stream_to_mem {}

   action mem_to_stream {}
}
```

Abstract base actions in separate files. Allows project-specific action implementation while keeping high-level tests the same

```
component mem_tile <
            int MEM_SIZE, component Translation,
            int NO_DMA, int NO_BD>
  : aie_tile<NO_DMA, NO_BD>
{
   Translation translation;
}
```

```
 component tile_addr_translation_c {

   target function bit[64] translate(
             addr_handle_t hndl,
             bit[64] base_address)
   {
     bit[64] addr;
     ...
     return (addr);
   }
 }
```

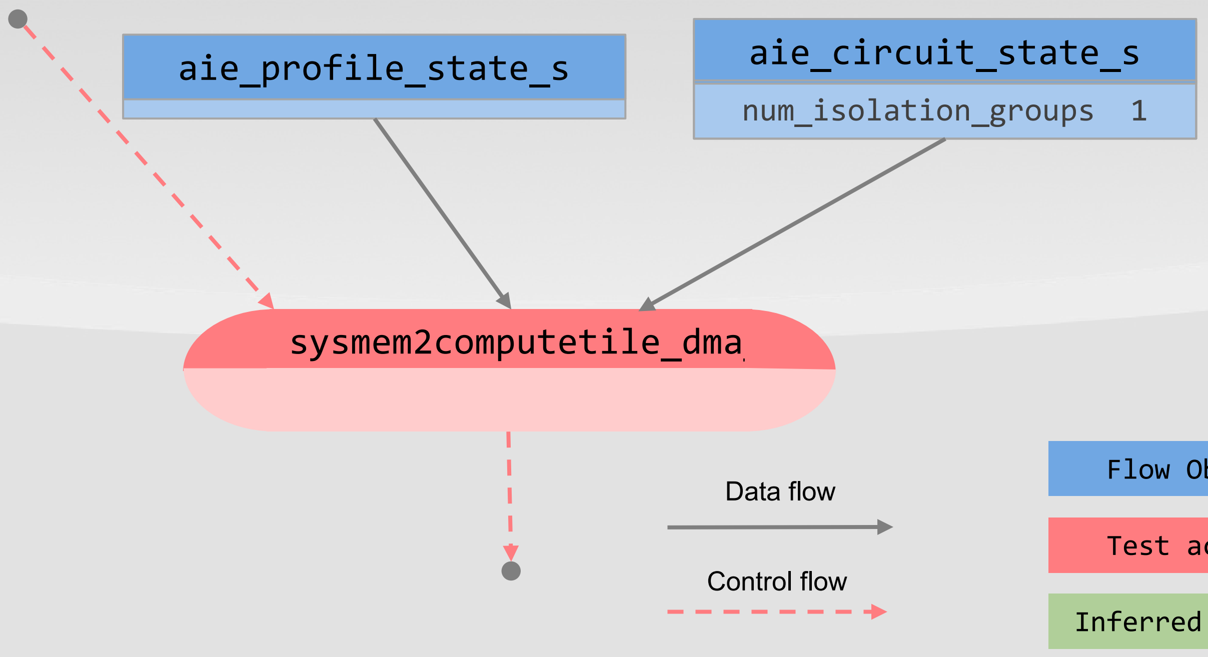
```
component compute_tile <
            int MEM_SIZE, component Translation,
            int NO_DMA, int NO_BD>
  : aie_tile<NO_DMA, NO_BD>
{
   Translation translation;
}
```

```
component shim_tile< int NO_DMA, int NO_BD >
   : aie_tile<NO_DMA, NO_BD>
{}
```

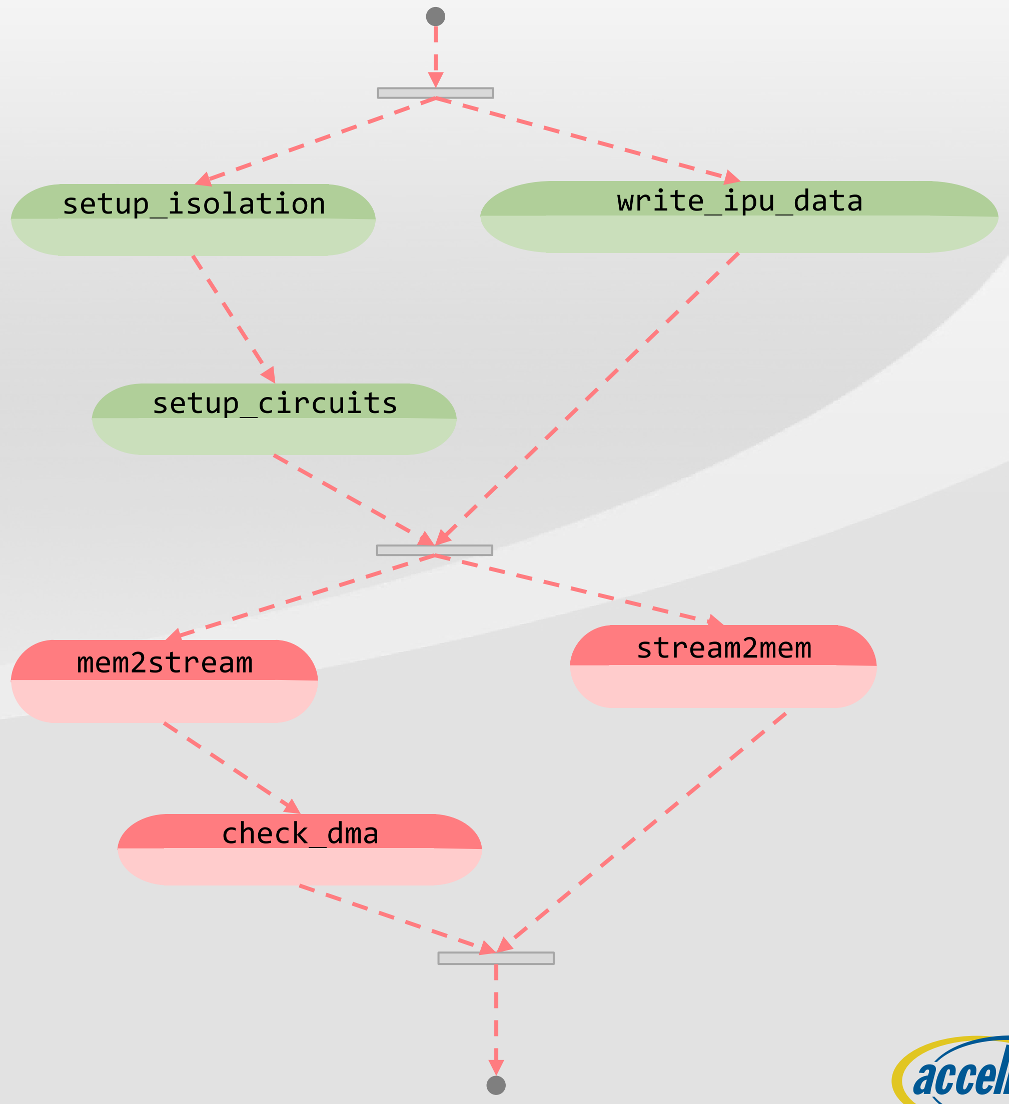
Test composition

Test

```
action sysmem2computedma {
    activity {
        sysmem2computetile_dma_parallel with {
            in_circuits_state.num_isolation_groups == 1;
        };
    };
};
```



Solved scenario



Test API design process

dma_compute2compute

- **State object to store current circuit state**
  - Max size circuits array in state object
  - Most rules about circuits encapsulated in state object
- **First try**
  - Action input current circuit state and output updated circuit state with new circuits
- **Problem**
  - Huge constraint space for doing two DMAs in series
  - Sparse solution space with constraints on input and output state objects

```
action dma_compute2compute_parallel {

  input  circuit_state_s circuits_in;
  output circuit_state_s circuits_out;

  // constraint rules to create output circuit state
  // from the input circuit state
  // ...

  rand array<circuit_node_s, MAX_CIRCUITS> src;
  rand array<circuit_node_s, MAX_CIRCUITS> dst;

  rand int in [1..MAX_CIRCUITS] parallel_count;

  activity {
    parallel {
      replicate(i: parallel_count) {
         do compute_tile::mem_to_stream with {node == src[i];}
         do compute_tile::stream_to_mem with {node == dst[i];}
      }
    }
  }
}
```

```
action two_dma {
   activity {
      do dma_compute2compute_parallel;
      do dma_compute2compute_parallel;
   }
}
```

Test API design process

dma_compute2compute

- **State object to store current circuit state**
  - Max circuits array
- **Preferred solution**
  - Constraints for isolation and circuits in the state object
  - DMA actions have an input state object but no output state
- **Tip**
  - A generic DMA action, that inputs the current state manipulates it, and then outputs it, is not always a good solution for high-level test space modeling with PSS
    - This generic action is very procedural that unnecessarily adds to constraint-solving complexity
    - Be cognizant of PSS global constraint-solving semantics

```
action dma_compute2compute_parallel {

  input  circuit_state_s circuits_in; // Only INPUT

  rand array<circuit_node_s, MAX_CIRCUITS> src;
  rand array<circuit_node_s, MAX_CIRCUITS> dst;

  rand int in [1..MAX_CIRCUITS] parallel_count;

  activity {

    parallel {
      replicate(i: parallel_count) {
         do compute_tile::mem_to_stream with {node == src[i];}
         do compute_tile::stream_to_mem with {node == dst[i];}
      }
    }
  }
}
```

Test API – building block actions

Test API design

- Constraint space and usability concerns affect the Test API design most
  - Quick iterations on API design is highly desirable

Three levels of test API – building block actions

- Level 1
  - Fully encapsulated sub-IP or IP model with registers, initialization, and programming sequences
  - E.g., compute, mem, and shim tile PSS components with mem_to_stream and stream_to_mem actions
  - Not directly used to create tests
- Level 2
  - Main test writers' interface
  - For example, *sysmem2computetile_dma_parallel* action from AIE component
- Level 3
  - Simplified high-level test interface used by the architect, SoC DV, and post-silicon
  - DMA action to do DMA from a given tile to another tile

They are powerful and dangerous

CONSTRAINTS

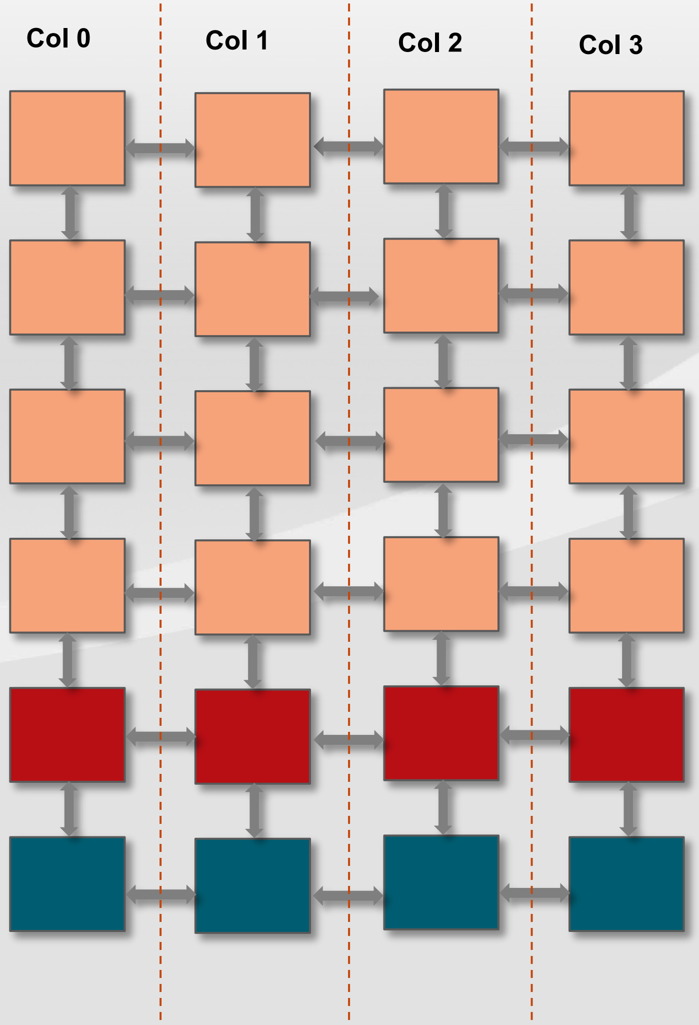
Constraint efficiency

- **Constraints are the backbone of the PSS model**
  - Allows for action inference, flow object binding inference, describe state and resource rules
- **PSS models formulate a global constraint-solving problem**
  - Local randomization in PSS 2.1 should contain the constraint-solving problem for data-only randomization
- ***Efficient constraints are extremely important for PSS models***
- **TIPS**
  - Only expose important attributes of test space as rand for test space exploration
    - E.g., random DMA channel allocation is important as a model attribute but not so much the data for DMA
  - Explicitly restrict the domain of a random attribute
    - E.g., DMA channel number shouldn't be '`rand int`' but '`rand int in [0..MAX_DMA-1]`'

Constraint efficiency – example problem

- Group adjacent columns as an isolation group
- Problem – create random legal isolation groups
- Isolation group examples

```
- One group    - [Col 0,    Col 1,    Col 2,    Col 3]
- Four groups  - [Col 0], [Col 1], [Col 2], [Col 3]
- Two groups   - [Col 0,    Col 1], [Col 2,    Col3]
- Two groups   - [Col 0,    Col 1,    Col 2], [Col3]
- Two groups   - [Col 0], [Col 1,    Col 2,    Col3]
- Three groups - [Col 0], [Col 1,    Col 2], [Col 3]
- Three groups - [Col 0,    Col 1], [Col 2], [Col 3]
- Three groups - [Col 0], [Col 1], [Col 2,    Col 3]
```



Constraint search space

- **Get N good mango baskets from M baskets**
  - Which basket is bad is not important
- **Constraint solver needs to find values of all rand attributes that satisfy the constraints**
- **The possibility space of the cross of rand attributes domain size is the constraint space**

```
action eat_good_mangoes {

  rand array<int in [1..MAX_MANGO], MAX_BASKET>               baskets;

  rand array<bool, MAX_BASKET>                            good_baskets;

  rand int in [0..MAX_BASKETS]                          num_good_baskets;
}
```

```
Size of constraint space = MAX_MANGO *
                           MAX_BASKET *
                           MAX_BASKET * 2
                           MAX_BASKET
```

```
action eat_good_mangoes {

  rand array<int in [1..MAX_MANGO], MAX_BASKET>            baskets;

  // Reduced constraint space
  // index from 0 to no_good_baskets-1 are good in the baskets array
  rand int in [0..MAX_BASKETS]                         num_good_baskets;
}
```

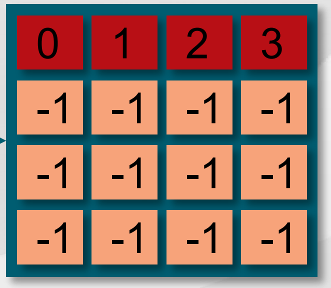
Constraint efficiency – a complex solution

```
action setup_isolation {

   // This array is one isolation profile that would use all columns
   rand array<array<int in [-1..NO_OF_COL-1], NO_OF_COL>,NO_OF_COL> isolation_profile;
   rand array<int in [-1..NO_OF_COL-1], NO_OF_COL*NO_OF_COL>        col_index;

   rand int in [1..NO_OF_COL]                            no_of_groups;
   rand array<int in [0..NO_OF_COL], NO_OF_COL>          group_sizes;

   rand int in [0..NO_OF_COL-1] columns[NO_OF_COL];
   constraint unique {columns};

   constraint group_sizes.sum() == NO_OF_COL;

   constraint foreach (group_sizes[i]) {
      if(i >= no_of_groups) {
         group_sizes[i] == 0;
      } else {
         group_sizes[i] >= 1;
      }
   }

   constraint col_index[0] == 0;

   constraint foreach (g:isolation_profile[i]) {
      if(group_sizes[i] > 0) {
         if(i < no_of_groups) {
            foreach (g[j]) {
               if(j < group_sizes[i]) {
                  if(i ==0 && j == 0) {
                     col_index[0] == 0;
                  }
                  else {
                     col_index[i*NO_OF_COL + j] == col_index[i*NO_OF_COL + j - 1] + 1;
                  }
               }
               else {
                  col_index[i*NO_OF_COL + j] == col_index[i*NO_OF_COL + j - 1];
               }
            }
         }
      }
   }
```

```
   constraint foreach (g:isolation_profile[i]) {
      if(i < no_of_groups) {
         foreach (g[j]) {
            if(j < group_sizes[i]) {
               g[j] == columns[col_index[i*NO_OF_COL + j]];
            }
            else {
               g[j] == -1;
            }
         }
      }
   }

   constraint foreach (g:isolation_profile[i]) {
      if(i >= no_of_groups) {
         foreach (g[j]) {
            g[j] == -1;
         }
      }
   }

   exec post_solve {

      printf("################ Isolation profile ##################\n");
      printf("No of isolated col groups: %d\n", no_of_groups);
      foreach (g:isolation_profile[i]) {
         if(group_sizes[i] > 0) {
            outf("(");

            foreach (g[j]) {
               if(j < group_sizes[i]) {
                  outf("%d,",g[j]);
               }
            }
         }
      }
   }
};
```

2D array of valid groups

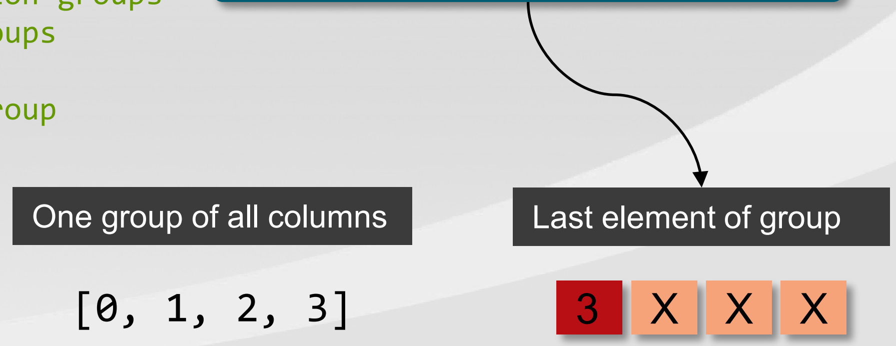
[0, 1, 2, 3]



Constraint efficiency: a simple efficient solution

```
// Stores the last column of an isolation group
rand array <int in [0..MAX_ISOLATION_GROUPS-1], MAX_ISOLATION_GROUPS> last_elem_of_group;

// Used for forcing DMAs into isolations so they aren't empty if we want
rand int in [1..MAX_ISOLATION_GROUPS] num_nonempty_groups;

// Ensure groups are continuous and don't skip over each other
constraint foreach(gb:last_elem_of_group[i]) {

  if(i < MAX_ISOLATION_GROUPS-1) {               // Only look at valid isolation groups
    if( i < num_nonempty_groups-1) {             // Only look at non-empty groups

      // last column of a group must be less than last column of the next group
      last_elem_of_group[i] < last_elem_of_group[i+1];
    }
    else {
      // group everything that is not valid
      last_elem_of_group[i] == last_elem_of_group[i+1];
    }
  }
}
// Last element of the last isolation group must be the last column
constraint last_elem_of_group[MAX_ISOLATION_GROUPS-1] == MAX_ISOLATION_GROUPS-1;
```

Compact representation of the solution



Constraint space comparison

2D array of valid groups

Last element of group

`[0, 1, 2, 3]`

0	1	2	3
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

3	X	X	X

`[0], [1], [2], [3]`

0	-1	-1	-1
1	-1	-1	-1
2	-1	-1	-1
3	-1	-1	-1

0	1	2	3

2D array of valid groups

Last element of group

`[0, 1], [2, 3]`

0	1	-1	-1
2	3	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

1	3	X	X

`[0], [1, 2], [3]`

0	-1	-1	-1
1	2	-1	-1
3	-1	-1	-1
-1	-1	-1	-1

0	2	3	X

The symmetry of the solution space presents a bigger search space for constraint solver

Constraint - conclusion

- **Design symmetry-free solution space when possible**
  - Only one possible representation of the solution in the constraint model
- **Minimize action inference for test writer interface actions**
  - If a compound action always causes inference of another action, add that action explicitly
- **Use explicit binds where possible**
  - Usually, a constraint solver is used for the possible binding of inputs and output
  - Explicit binding reduce constraint complexity
- **Avoid chaining of complex constraints from input to output**
  - For example, avoid, constraining all output flow object fields except a few equal to input flow object fields
- **Design PSS model to only expose important random attributes to the next integration level**
  - A poorly designed constraint model could easily overwhelm current and future constraint solvers
  - Only allow free random attributes at system-level that are important for system-level tests

A language can only do so much

PSS METHODOLOGY

PSS methodology - call to action

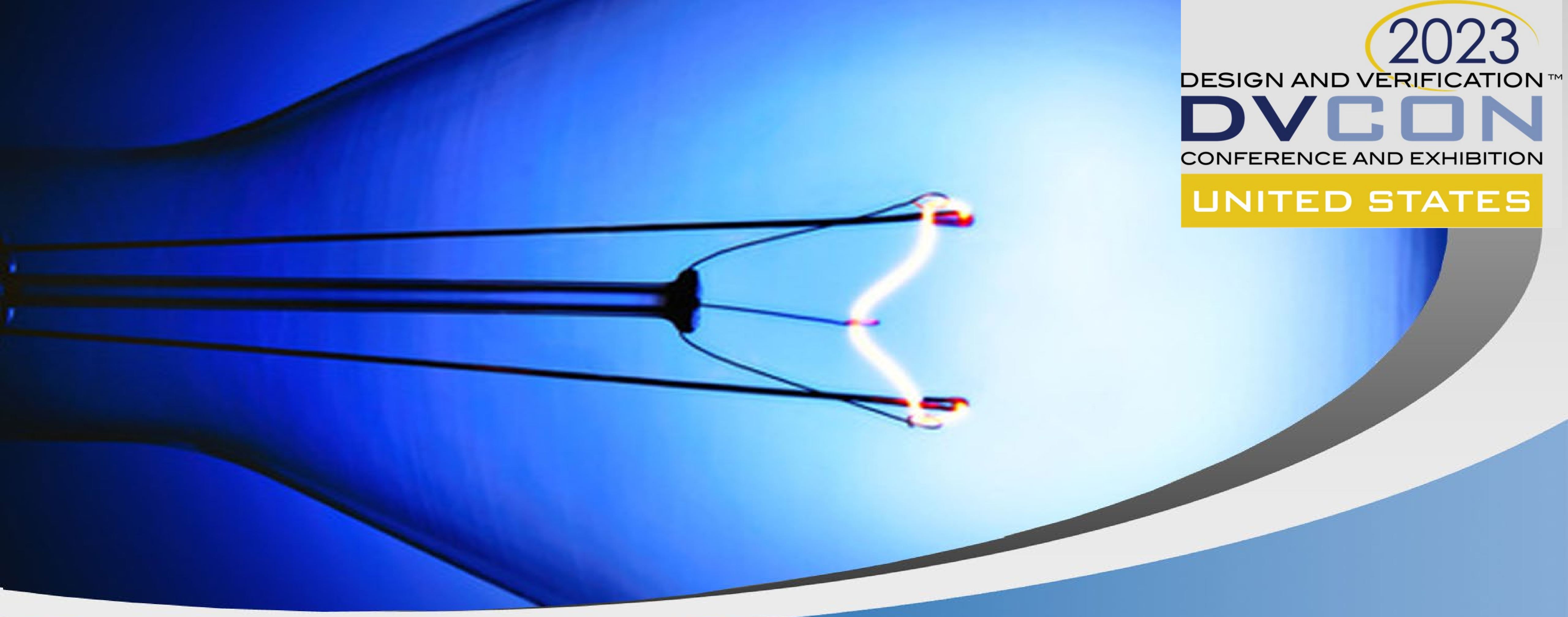
- **Create industry-wide PSS methodology library like UVM**
  - Common types to facilitate smooth multi-IP and third-party integration
  - Support virtualization and address translation to work across IPs including third part IPs
- **Create high-level open action libraries for the standard protocols that work across vendors**
  - PCIe, CXL, UCIe, etc.
- **Create a forum for community-maintained design patterns**
  - For example, Power state transitions with interleaved traffic
- **Publish best practice guidelines**
  - Constraint modeling
  - Code structuring
- **Improve PSS action export methodology for use in the production firmware**

PSS deployment strategies

- **Start small and provide immediate value**
  - Refactor initial design to create a first-class PSS model
- **Starting a PSS project at any integration level is good**
  - SoC with processors
    - It’s difficult to create random, multi-IP tests manually in C for the processors
    - Use PSS for random, multi-engine functional, stress, and power tests
  - Sub-system UVM and/or SystemC
    - New tests can be created quickly. Reduced boilerplate for tests
    - Scenario randomization as compared to data randomization
    - Focus on describing system rules and encapsulated functionality then use PSS automation to create new tests
  - IP
    - PSS allows better reasoning of resource modeling and config space for an IP (see DVCON US 2022 presentation)
    - PSS-based portable programming sequences help your friends at SoC and post-silicon (if they are friends)

Conclusion

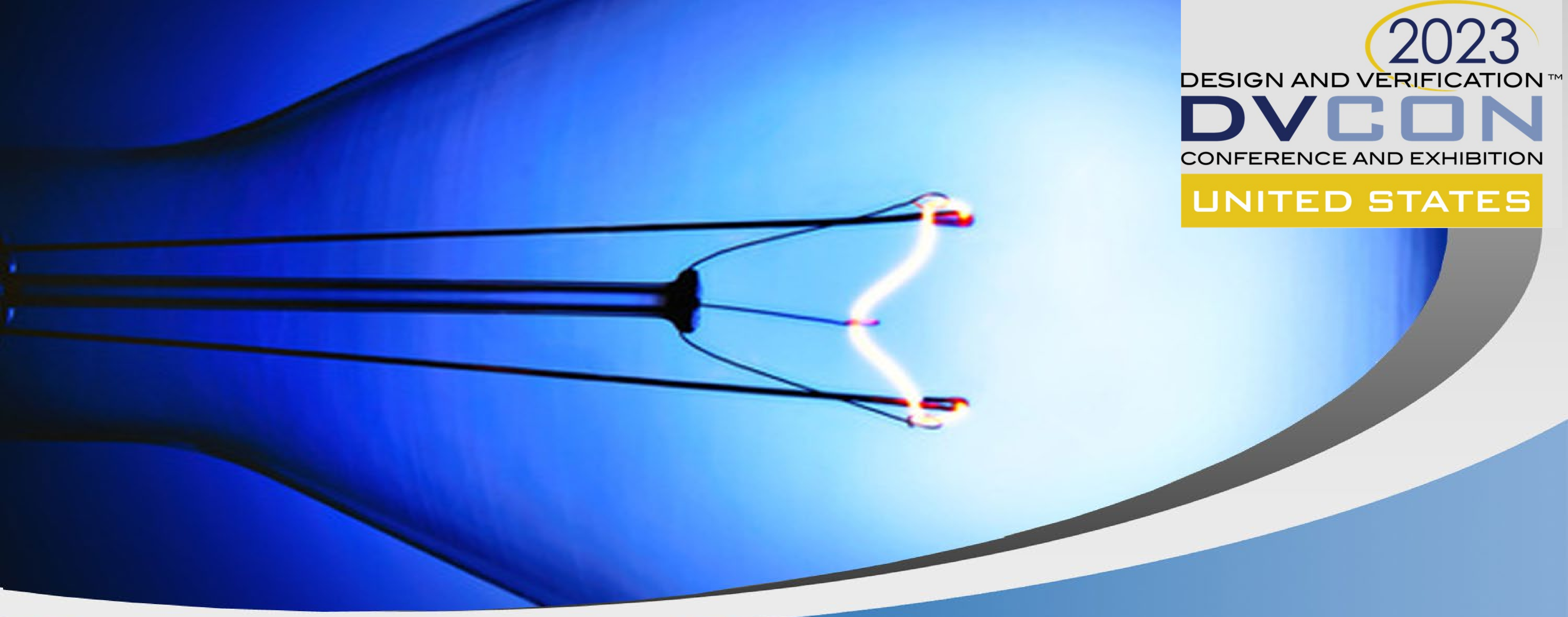
- **Reduced test development costs with portability**
  - IP programming sequence can be used in post-silicon bring-up, validation, and manufacturing tests
  - PSS enables the rapid creation of complex tests that identify functional, power, performance, and firmware bugs in heterogeneous multi-processor systems through randomized, multi-IP testing
- **PSS improves communication among Architecture, Firmware, DV, and Post-silicon teams through formal high-level language, leading to fewer bugs**
  - Describe and share SoC configuration, Inter-IP initialization, and power sequencing at high-level
  - The PSS language allows for effective communication through its semantics and constructs, serving as an executable specification
- **Better post-silicon bring-up and validation**
  - Generate a range of tests, from simple to complex, in thousands, for effective bring-up, validation, and optimization
  - Collect generation-time and runtime reports to demonstrate and analyze coverage
- **Creating new tests is fun with PSS**
  - Reduced friction for creating new tests, especially multi-IP tests
    - Understanding every detail of SoC is not required with PSS partial scenario specification
  - Improved productivity is personally satisfying for engineers



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Q & A

accellera SYSTEMS INITIATIVE



Approaches and Challenges to Scalable Modeling

DVCon 2023

Mike Chin, Principal Engineer, Intel Corporation
(michael.a.chin@intel.com)

Motivation/Need for Portable Stimulus

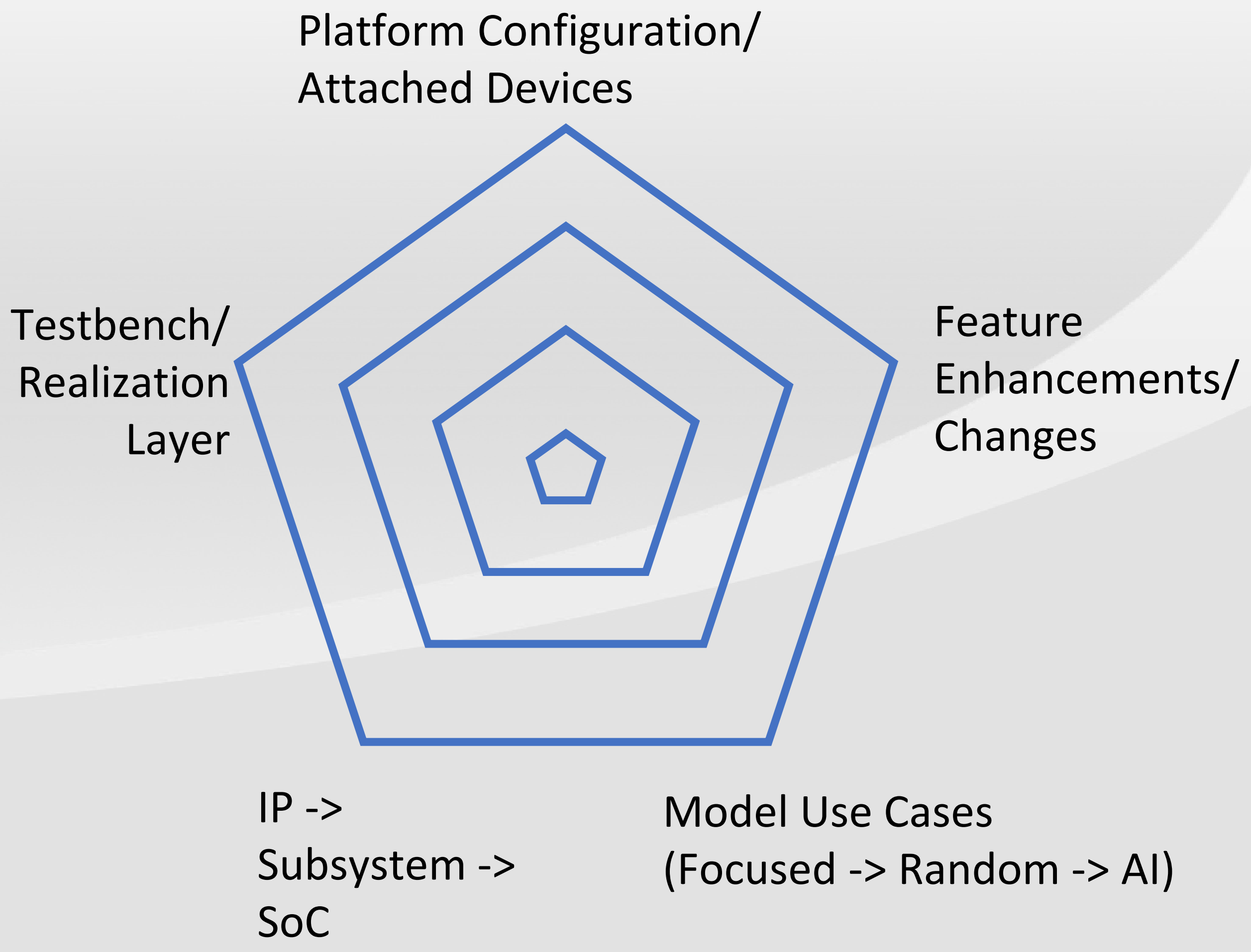
- **Lower ramp up cost for new validation engineers**
  - Simplify content creation
  - Hide complexity - realization layer
  - **Drive efficient productivity!**
- **Grow expertise from "user" to "power user"**
  - User – verification engineer
  - Power users – model developers, backend developers, debuggers
- **"Correct by construction" validation content**
  - Drive accurate modeling to ensure test content correctness through implicit actions

Portable Stimulus has the promise to accelerate verification through ease of creation and accuracy of verification content!

Challenges to Scalable Modeling

- Premise
- Pace of innovation, complexity of design continues to increase
- Verification resources (at best!) remain constant
- How do we effectively validate across the continuum?

How do we comprehend this problem? What methodologies can we use to maximize development ROI?


Platform Configuration/ Attached Devices
Feature Enhancements/ Changes
Model Use Cases (Focused -> Random -> AI)
IP -> Subsystem -> SoC
Testbench/ Realization Layer

Our Modeling Problem

- **Simple state machine IP**
  - States:
    - On
    - Ready
    - Off
  - Actions:
    - PowerUp
    - PowerDown
    - Initialize
    - Read
  - Support 2 independent FSM instances

How do we create scalability in our model?
How do we deal with the same model executing on different platforms?
Enhancements for a new project?

Base Model + Extend vs. Override

```
package dvcon2023 {
  enum fsm_state { On, Off, Ready };
  state state_t {
    fsm_state cacheState;
  };

  component BasicIP {
    pool state_t fsmState;
    action PowerUp {
      output state_t outState;  constraint outState == On;
    };
    action PowerDown {
      input state_t inState;    constraint inState == Ready;
      output state_t outState;  constraint outState == Off;
    };
    action Initialize {
      input state_t inState;    constraint inState == On;
      output state_t outState;  constraint outState == Ready;
    };
    action Read {
      input state_t inState;    constraint inState == Ready;
      output state_t outState;  constraint outState == Ready;
    };
  };
};
```

Extend vs. Override in modeling logic

- **“Base” model**
  - Reusable, core modeling logic consisting of:
    - Actions
    - Buffers, Streams, States
    - Resources, pools
  - Scalable models address different execution platforms, model configurations, and use cases
    - A scalable realization layer is challenging to implement

- **How do we build reusability and scalability into our models?**
  - Use SW best practices!
  - Extend models
    - Support new capabilities through “extend” language syntax
  - Override settings
    - Constrain model settings to reflect project/configuration values

Extend

- **Use to support capabilities not present in the base model**
  - Optional features/New enhancements
  - Platform-specific configuration support
  - SoC/Subsystem integration
- **Support for new logic capabilities**
  - Actions
  - Enum/struct members
  - Resources

```
package dvcon2023 {
  extend enum fsm_state [ Busy ];
  extend component BasicIP {
    action Write {
      input state_t inState;    constraint inState == Ready;
      output state_t outState;  constraint outState == Busy;
    };
    action PollCompletion {
      input state_t inState;    constraint inState == Busy;
      output state_t outState;  constraint outState == Ready;
    };
  };
};
```

Override

- **Configurable parameters in the base model that are overridden per project/configuration**
  - Number of subdevice instances
  - Agent count
  - Target types
- **Additional constraints to specify component variable value ranges or specific values**
  - Base models must be written to be configurable!

```
//  Platform 1 (Post-si)
package dvcon2023 {
  extend component BasicIP {
    function IPPowerDown();
  };
  extend action BasicIP::PowerDown {
    exec body { comp.IPPowerDown(); }
  };
};
```

```
//  Platform 2 (Pre-si testbench)
package dvcon2023 {
  extend component BasicIP {
    function SidebandShutOffClocks();
    function SidebandVerifyControllerPowerState();
    action PreSiPowerDown : PowerDown {
      exec body {
        comp.SidebandShutOffClocks();
        super;
        comp.SidebandVerifyControllerPowerState();
      }
    };
  };
};
```

Challenges to Extend vs. Override

- **Realization Layers**
  - Scalable support for all permutations:
    - Model scope (IP -> Subsystem -> SoC)
    - Platforms
      - Devices
      - Traffic generators
    - Use cases (focused tests, randomization, AI)
  - Extend/Override mechanisms need to be carefully managed!

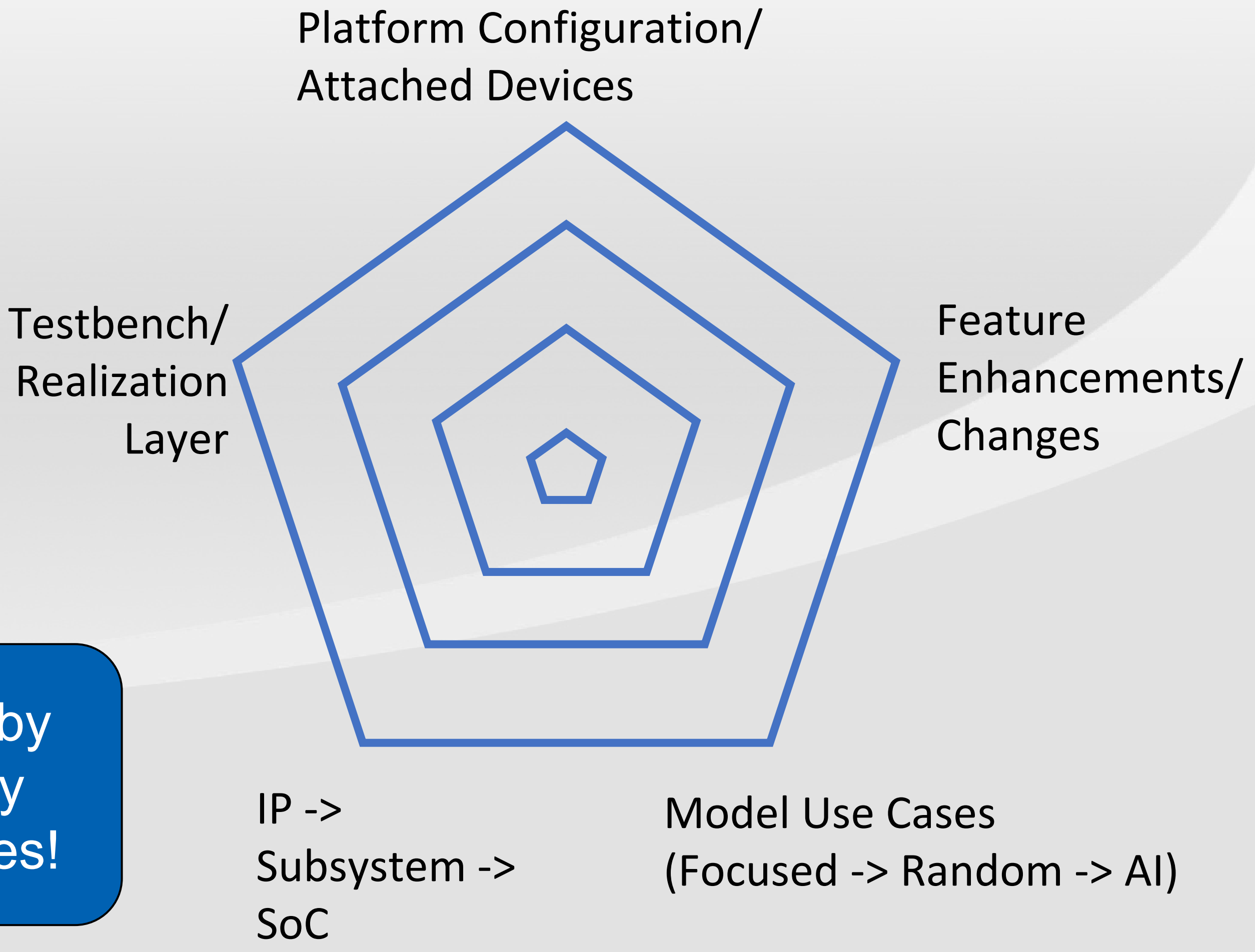
- **Base vs. Extend vs. Override**
  - Extend/Override are useful mechanisms for expanding model support
  - When do we make the decision to put something back into the base model?

Challenges to Scalable Modeling

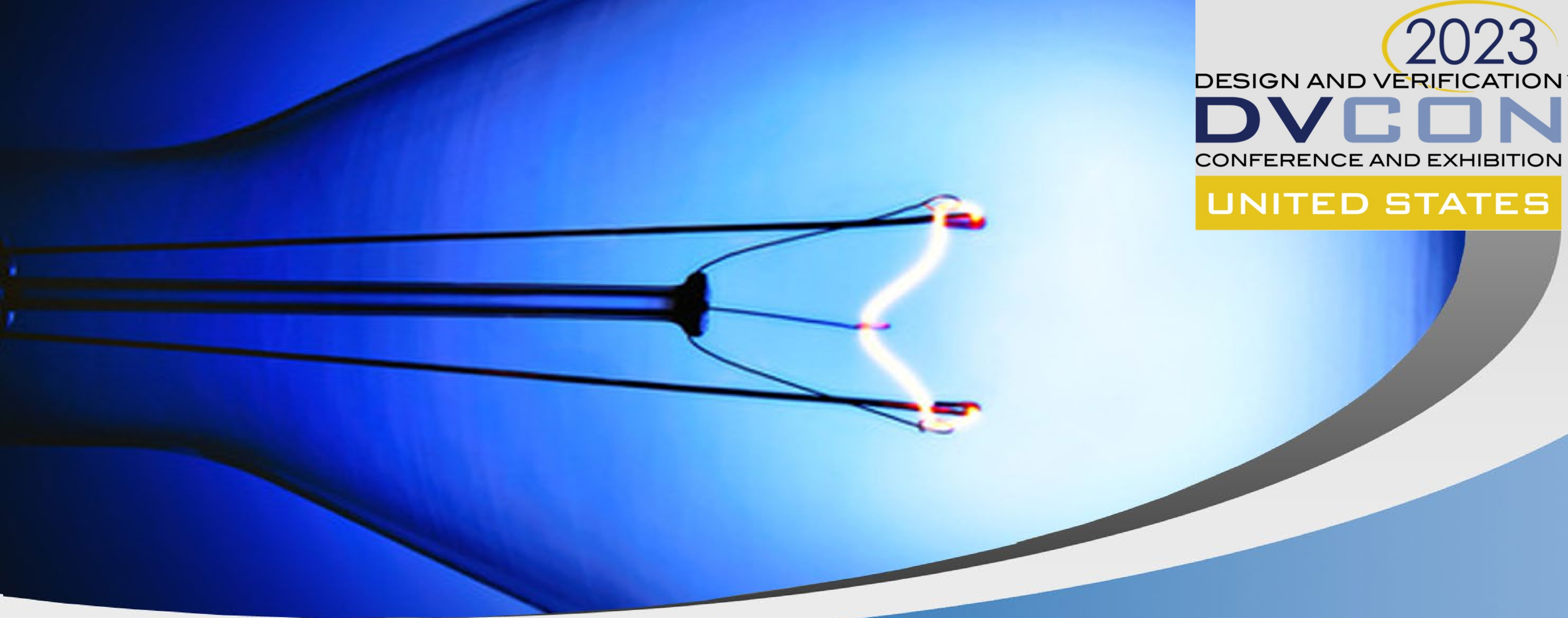
- **Modeling capabilities**
  - Dynamic hardware problems still pose challenges to scalable modeling
    - Dynamic bandwidth calculation/throttling
    - Reconfigurable transport layer modeling
- **Vendor/use case specialization**
  - Working in 2 languages is cumbersome
  - Language support for vendor tool capabilities
    - Randomization tools
    - AI workflows/tools
- **Scalable Subsystem/SoC support**
  - Intuitive model composition with multiple IP models
  - Layered support for subsystem/SoC features
- **Input collateral for model configuration**
  - Leveraging non-PSS config files
  - Convert to PSS is the only option

Summary - Challenges to Scalable Modeling

- **Premise**
  - Pace of innovation, complexity of design continues to increase
  - Verification resources (at best!) remain constant


Platform Configuration/ Attached Devices
Feature Enhancements/ Changes
Model Use Cases (Focused -> Random -> AI)
IP -> Subsystem -> SoC
Testbench/ Realization Layer

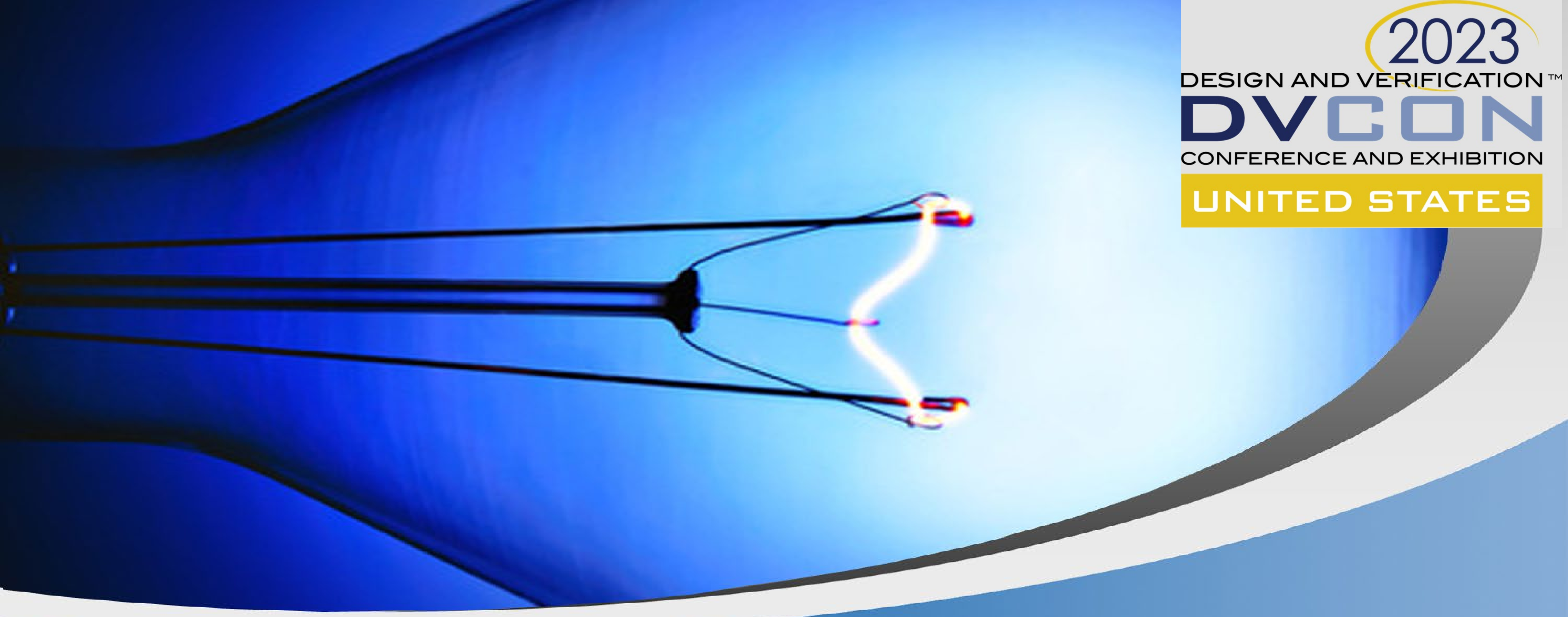
Modularity modeling approaches supported by PSS drive efficient reusability and scalability across a wide variety of verification challenges!



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Q & A

accellera SYSTEMS INITIATIVE



2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Panel Discussion

accellera SYSTEMS INITIATIVE