

Learning the Portable Stimulus Standard

Tom Fitzpatrick

CEO Big Fish EDA Consulting

On behalf of the Accellera Systems Initiative

and the Accellera Portable Stimulus Working Group



Memory Access API and Executors

▶▶▶ Session 13

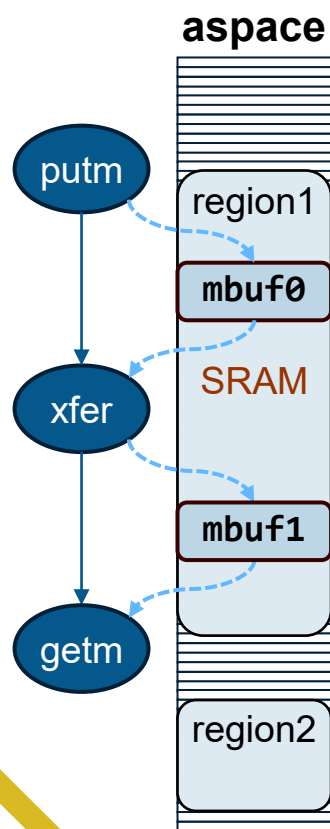


... What You Will Learn

- ▶ Abstract addresses using handles
- ▶ Memory access API
- ▶ Defining and claiming executors
- ▶ Executor-based API customization

Address Claims and Address Handles

- ▶ An address claim gets mapped to an actual address during test execution
- ▶ An `addr_handle_t` is used to represent the address during solve time



```
package addr_reg_pkg {
  typedefchandle  addr_handle_t;
  function  addr_handle_t  make_handle_from_claim(addr_claim_base_s claim, bit[64] offset = 0);
  function  addr_handle_t  make_handle_from_handle(addr_handle_t handle, bit[64] offset);

  target function bit[64] addr_value(addr_handle_t hnd1);

  action put_a {
    output mbuf_b obuf;

    exec body {
      addr_handle_t h0 = make_handle_from_claim(obuf.mClaim);
      bit[64] addr = addr_value(h0);
      repeat(i:obuf.size) {
        write8(make_handle_from_handle(h0,i), obuf.data[i]);
        message(NONE,"Wrote %2x to %0x",obuf.data[i],addr+i);
      }
    }
  }
}
```

An address handle can be obtained from a claim

or another handle

offset allows access to different addresses in a claim

Allow target to access concrete address

Built-in access API takes a handle as an argument

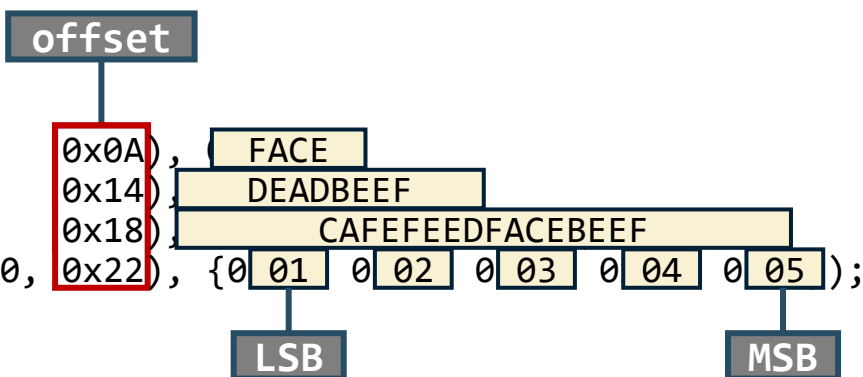
Memory Access API

- ▶ Standard set of functions to read/write byte-addressable address spaces
- ▶ All API methods use address handles

```
component pss_top {  
  transparent_addr_space_c<MemRegionTrait_s> aspace;  
  transparent_addr_region_s<MemRegionTrait_s> region1;  
  addr_handle_t a_h;  
  exec init_up {  
    region1.addr = 0x0000 ;  
    region1.size = 0x40;  
    a_h = aspace.add_region(region1);  
  }
```

```
  action entry_a {  
    addr_handle_t h0;  
    list<bit[8]> data;  
    exec body {  
      h0 = this.comp.a_h;  
      write8(h0, 0xA5);  
      write16(make_handle_from_handle(h0,  
      write32(make_handle_from_handle(h0,  
      write64(make_handle_from_handle(h0,  
      write_bytes(make_handle_from_handle(h0,  
    }  
  }
```

07	06	05	04	03	02	01	00	
								38
								30
								28
		05	04	03	02	01		20
		CAFEFEEDFACEBEEF						18
		DEADBEEF						10
					FACE			08
							A5	00



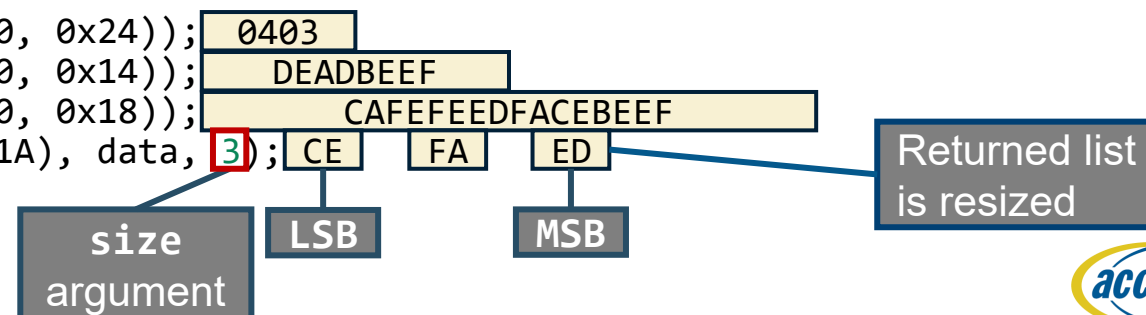
Memory Access API

- ▶ Standard set of functions to read/write byte-addressable address spaces
- ▶ All API methods use address handles

```
component pss_top {  
  transparent_addr_space_c<MemRegionTrait_s> aspace;  
  transparent_addr_region_s<MemRegionTrait_s> region1;  
  addr_handle_t a_h;  
  exec init_up {  
    region1.addr = 0x0000 ;  
    region1.size = 0x40;  
    a_h = aspace.add_region(region1);  
  }
```

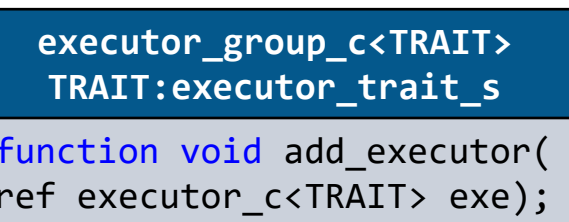
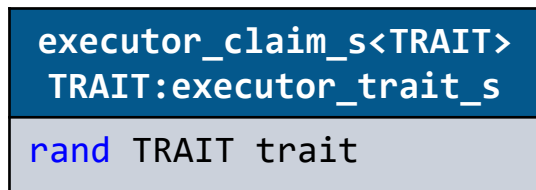
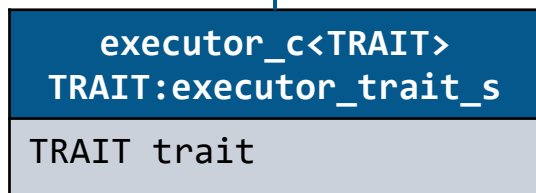
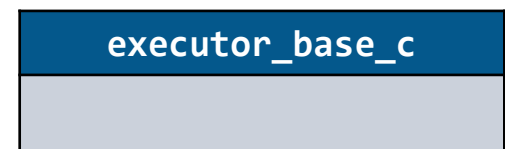
```
  action entry_a {  
    addr_handle_t h0;  
    list<bit[8]> data;  
    exec body {  
      h0 = this.comp.a_h;  
      data8 = read8(h0);  
      data16 = read16(make_handle_from_handle(h0, 0x24));  
      data32 = read32(make_handle_from_handle(h0, 0x14));  
      data64 = read64(make_handle_from_handle(h0, 0x18));  
      read_bytes(make_handle_from_handle(h0, 0x1A), data, 3);  
    }  
  }
```

07	06	05	04	03	02	01	00	
								38
								30
								28
		05	04	03	02	01		20
		CAFE		ED	FA	CE		18
		DEADBEEF						10
					FACE			08
							A5	00



Introducing Executors

- An *executor* represents a runtime agent that executes an element of the model realization (e.g., embedded processor, BFM, transactor, etc.)



```

package executor_pkg {
  struct executor_trait_s {};
  struct empty_executor_trait_s : executor_trait_s {};
  component executor_base_c {};
  component executor_c
    <struct TRAIT : executor_trait_s = empty_executor_trait_s> : executor_base_c {
      TRAIT trait;
    }

  struct executor_claim_s <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
    rand TRAIT trait;
  };

  component executor_group_c <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
    solve function void add_executor(ref executor_c<TRAIT> exe);
  }
}
  
```

executor TRAIT base type

default TRAIT type

TRAIT must be derived from executor_trait_s

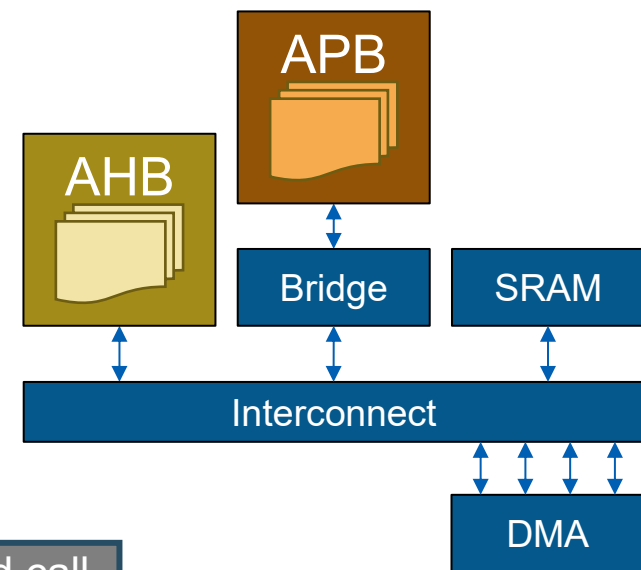
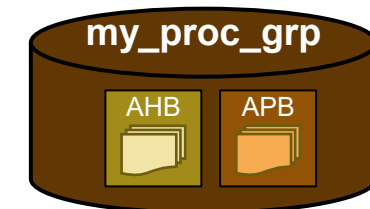
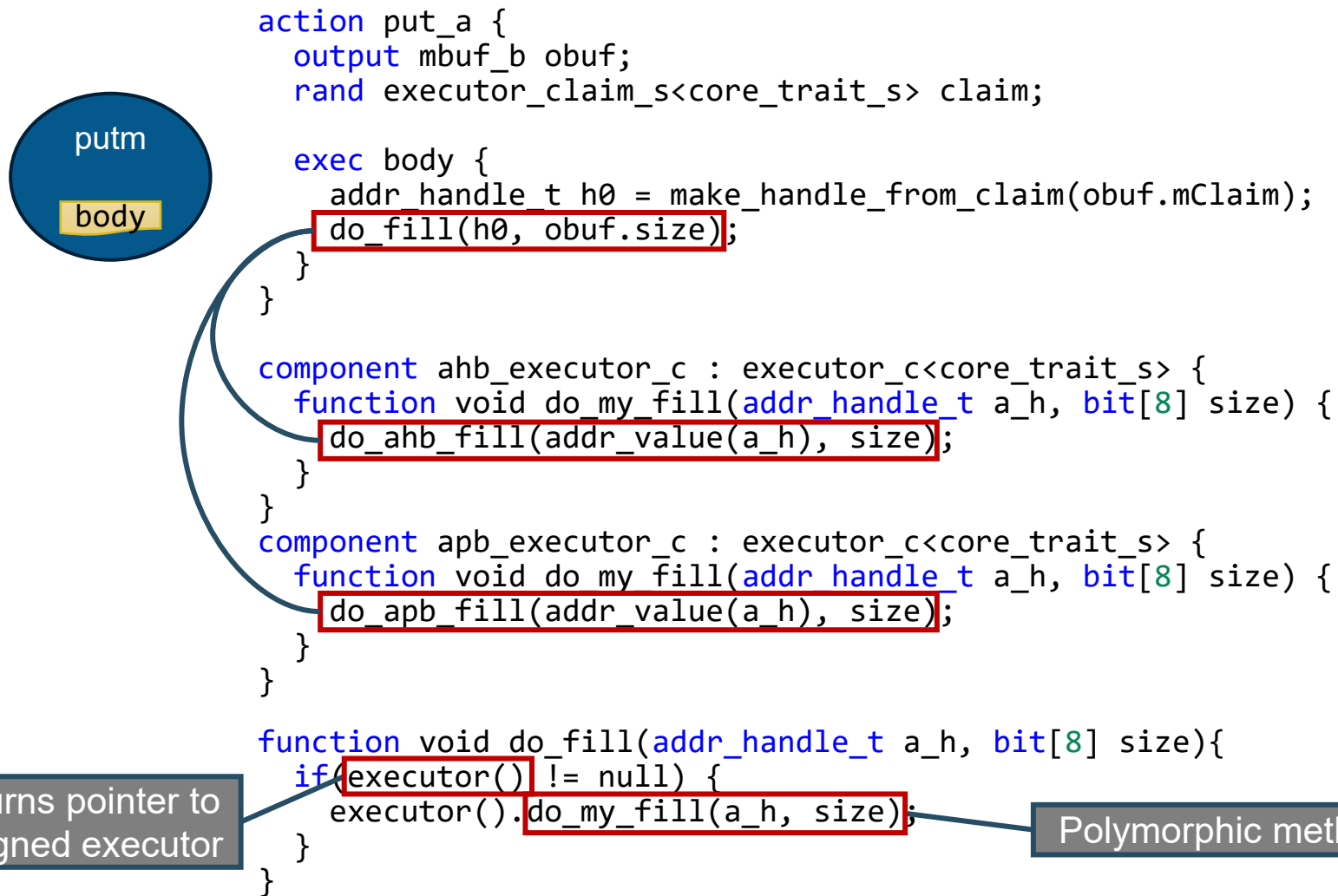
Default to empty trait

Claim is matched to an executor that satisfies the trait constraint

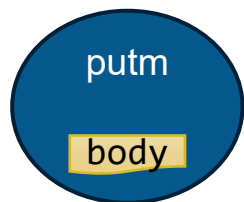
executor_c, group and claim must have the same TRAIT type

Executor claims can be made from actions or flow/resource objects

Executor-based API Customization



Executor Resources

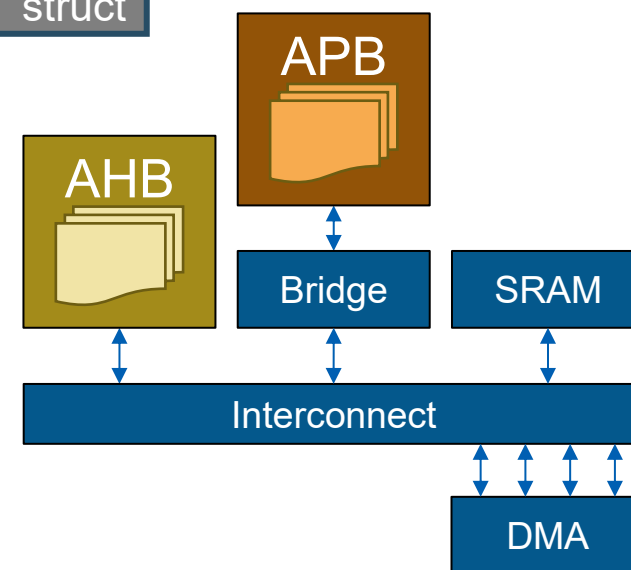
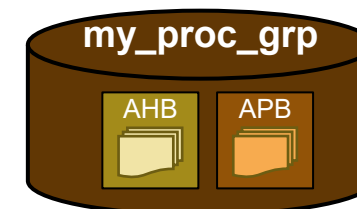


```
action put_a {  
  output mbuf b obuf;  
  lock my_core_r;  
  ...  
}
```

Exclusive access to executor
for duration of the action

```
resource my_core_r : executor_claim_s<core_trait_s> {  
  constraint trait.id == instance_id;  
};
```

Resource derived from
executor_claim_s struct





Thank You

