SystemRDL to PSS
Basic to Pro

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

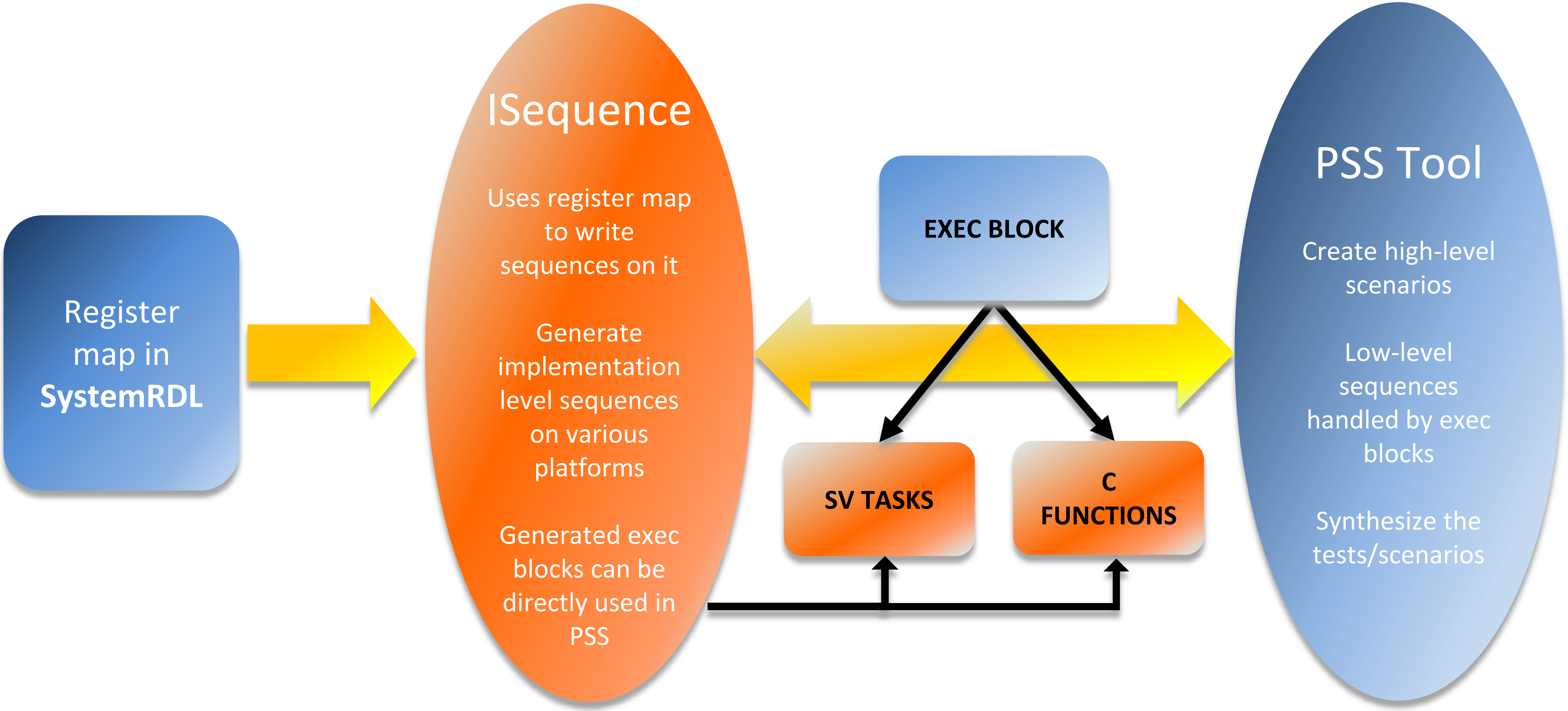
Amanjyot Kaur
(Host)

Nikita Gulliya
(Presenter)

Agenda

- Introduction
- Components
  - Field
  - Register
  - Register File
  - Address Map
  - Memory
- Signals
- Address Allocation
- Enumerations
- Parameters
- Structures
- Property Assignment
- Special Register
- Verification Constructs
  - HDL PATH
  - Constraint
  - Structural Testing
- Perl preprocessor
- SoC HW/SW Interface Layer (HSI)
- Example Sequence with HSI
- Introduction to Sequences
- Problems faced with sequences
- Proposed Solution: ISequenceSpec
- Introduction to Portable Standard Stimulus (PSS)
- ISS+PSS tool flow example: Industrial Washer
- Conclusion

SystemRDL to PSS



Register map in **SystemRDL**

ISequence

Uses register map to write sequences on it

Generate implementation level sequences on various platforms

Generated exec blocks can be directly used in PSS

PSS Tool

Create high-level scenarios

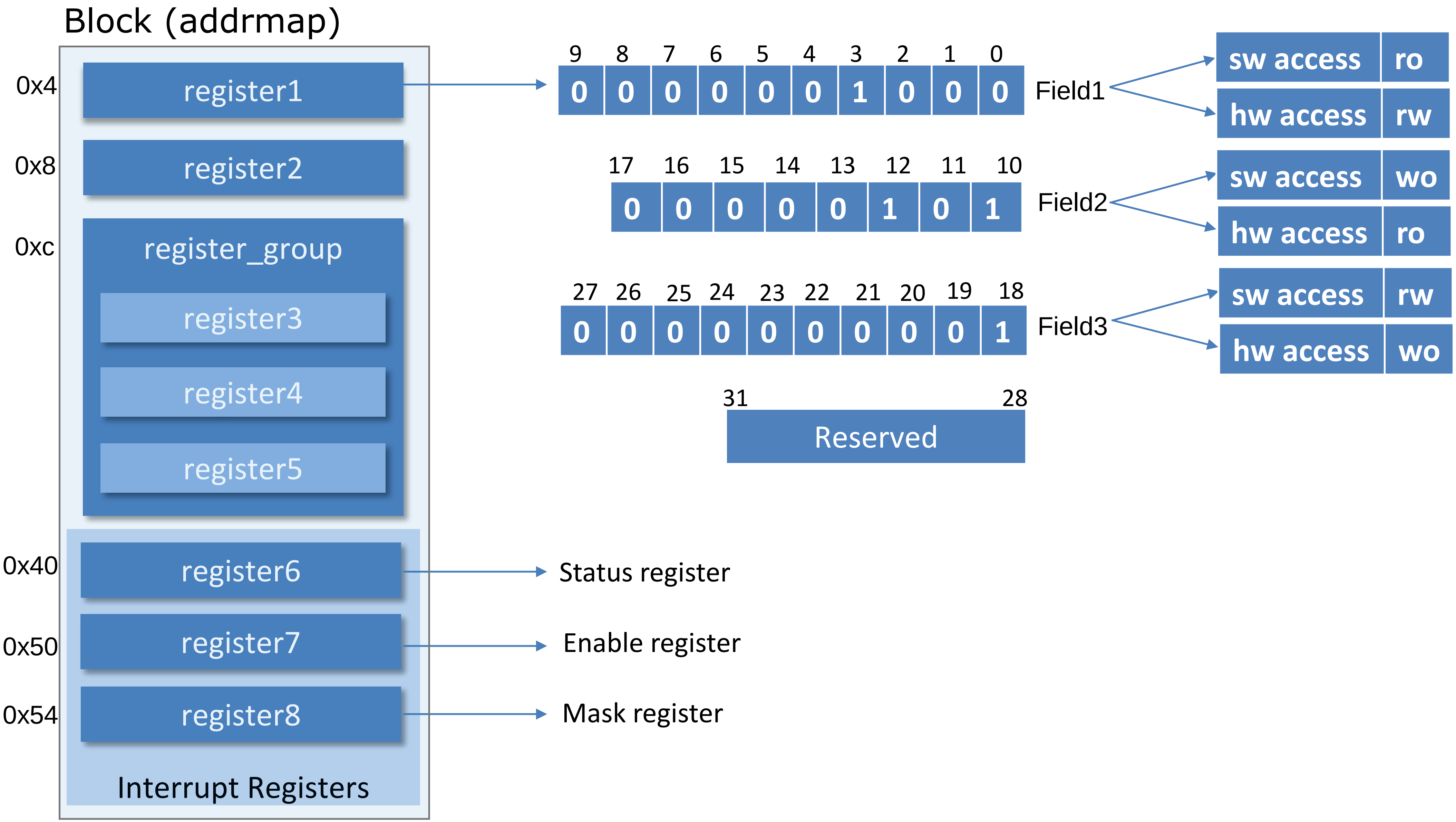
Low-level sequences handled by exec blocks

Synthesize the tests/scenarios

SystemRDL Importance and History

- An embedded system consists of Hardware and Software components.
- SystemRDL is a textual representation of Hardware-Software interface consisting of addressable registers, interrupts, counters etc.
- History
  - Created at Cisco, released as Accellera 1.0 standard.
  - Version 2.0 released in Jan 2018
    - Added Verification constructs, parameterization, data types etc.
    - Reference: https://www.accellera.org/images/downloads/standards/systemrdl/SystemRDL_2.0_Jan2018.pdf
- Support specification centric flow, automatically generate
  - RTL bus interface
  - Verification model
  - C header and API
  - Documentation

Example



Block (addrmap)

0x4	register1
0x8	register2
0xc	register_group: register3, register4, register5
0x40	register6
0x50	register7
0x54	register8

Interrupt Registers

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	1	0	0	0	Field1

17	16	15	14	13	12	11	10	
0	0	0	0	0	1	0	1	Field2

27	26	25	24	23	22	21	20	19	18	
0	0	0	0	0	0	0	0	0	1	Field3

31	28
Reserved	

Field1	sw access	ro
	hw access	rw
Field2	sw access	wo
	hw access	ro
Field3	sw access	rw
	hw access	wo

register6 → Status register

register7 → Enable register

register8 → Mask register

Defining Components

Definitive definition :

In definitive definition we instantiate the component in a separate statement. It is suitable for reuse.

```
addrmap top {
  regfile reggrp1 {
   reg r1 {
     regwidth = 32;
     field f1 {
       hw = rw;
       sw = rw;
     };
     f1 field1[31:0] = 31’b0;
   };
   r1 reg1 @0x100;
  };
  reggrp1 reggrp1;
};
```

regwidth

Bit information

Default value

Offset value

Anonymous definition:

In Anonymous definition we instantiate the component in the same statement. It is suitable for components that are used once.

```
addrmap top{
    regfile {
           reg {
            desc=“Specify the register”;
            field {} field1;
           } reg1;
      } reggrp1;
  };
```

Describes the component’s purpose.

Default regwidth = 32, fieldwidth=1, offset values = 0, default value =0 sw=rw, hw=rw, taken

Field

The **field** component is the lowest-level structural component, it stores the bit information of a register .

Definitive field definition:

field [**#(**field_parameter_instance [**,** field_parameter_instance]*)] field_instance_element [**,** field_instance_element]***;**

e.g. field f { };
f f1;

Anonymous field definition:

field {field_body} field_instance_element [,field_instance_element]*;

e.g. field { } f1;

e.g.

field { } singlebitfield; // 1 bit wide, not explicit about position
field { } somefield[4]; // 4 bits wide, not explicit about position
field { } fieldindices[3:0]; // a 4 bits field with explicit indices

Field ordering in registers

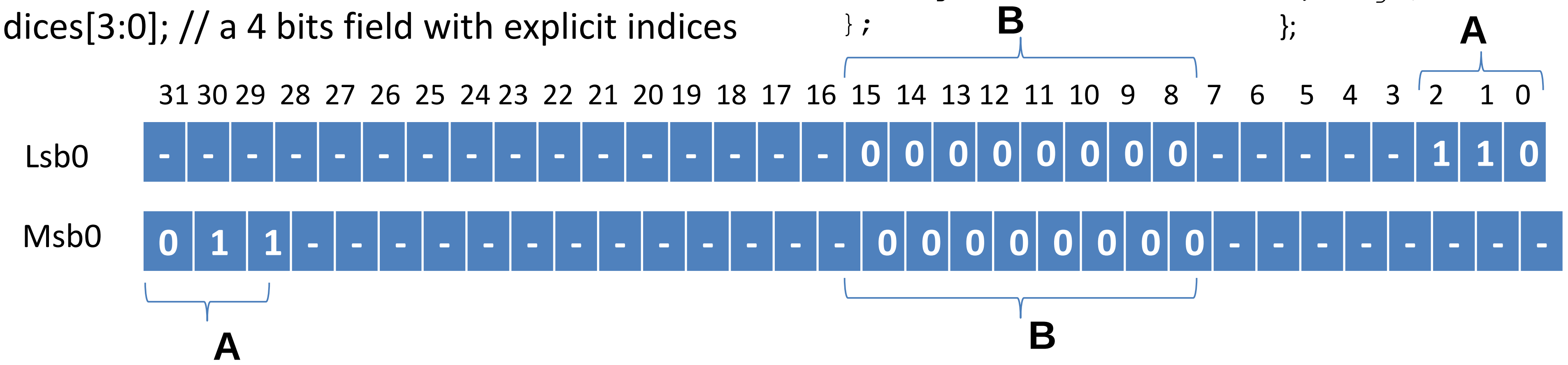
Field ordering in registers	Syntax
lsb0	field_type field_instance [high:low]
msb0	field_type field_instance [low:high]

```
addrmap top{
 lsb0;
 reg{
  field {} A[3]= 3'b110;
  field {} B[15:8];
 } regA;
};
```

3 bits from 2 down to 0

```
addrmap top{
 msb0;
 reg{
  field {} A[3]= 3'b110;
  field {} B[8:15];
 } regA;
};
```

3 bits from 29 down to 31



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Lsb0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	-	-	-	-	-	1	1	0
Msb0	0	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-

Software Access Properties

Properties	Description	Dynamic
rclr	Clear on read	Yes
rset	Set on read	Yes
onread	Read side-effect	Yes
woset	Write one to set	Yes
woclr	Write one to clear	Yes
onwrite	Write function	Yes
swwe	Software write-enable active high	Yes
swwel	Software write-enable active low	Yes
swmod	Assert when field is modified by software (written or read with a set or clear side effect)	Yes
swacc	Assert when field is software accessed	Yes
singlepulse	The field asserts for one cycle when written 1 and then clears back to 0 on the next cycle. This creates a single-cycle pulse on the hardware interface	Yes

```
reg transmit{
  field {} data,ack;
  field {
   hw = rw;
   sw = rw;
   onread  = rclr;
   onwrite = woset;
   swacc;
  } s_dat;
   field {
   hw = r;
   sw = w;
   singlepulse;
  } tot_sz;

};
addrmap myAmap{
 transmit tx1;
 tx1.data -> swwel = true;
 tx1.ack -> swmod = true;
};
```

Hardware Access Properties

Property	Description	Dynamic
we	Write-enable (active high)	Yes
wel	Write-enable (active low)	Yes
anded	Logical AND of all bits in field	Yes
ored	Logical OR of all bits in field	Yes
xored	Logical XOR of all bits in field	Yes
fieldwidth	Determines the width of all instances of the field. This number shall be a numeric. The default value of fieldwidth is undefined	Yes
hwclr	Hardware clear. This field need not be declared as hardware-writable	Yes
hwset	Hardware set. This field need not be declared as hardware-writable	Yes
hwenable	Determines which bits may be updated after any write enables. Bits that are set to 1 will be updated	Yes
hwmask	Determines which bits may be updated after any write enables. Bits that are set to 1 will not be updated	Yes

```
reg transmit{
  field {
    fieldwidth = 5;
  }src,dst,data;
  field {}nack;
  field {}ack;

};
addrmap myAmap{
 transmit tx1,tx2;
 tx1.src -> we  = true;
 tx1.dst -> wel = true;
 tx1.nack -> anded = true;
 tx2.src -> hwenable = tx1.src;
};
```

Register

A register is defined as a set of one or more SystemRDL field instances that are atomically accessible by software at a given address.

Definitive register definition

[**external**] *reg_name* [**#(** *parameter_instance* [**,** *parameter_instance*]*)]
reg_instance_element [**,** *reg_instance_element*]* **;**

Anonymous register definition

reg {[*reg_body*]**}**
[**external**] *reg_instance_element* [**,** *reg_instance_element*]***;**

Register Instantiation into three forms:

Register Instantiation forms	Description
internal	all register logic is created by the SystemRDL compiler for the instantiation (the default form)
external	the register/memory is implemented by the designer and the interface is inferred from instantiation
alias	Alias registers are used where designers want to allow alternate software access to registers. SystemRDL allows designers to specify alias registers for internal or external registers

```
 reg transmit {
  field {
   hw=w;
   sw=rw;
  } data;
 };
 reg some_intr {
  field {
   hw=w;
   sw=rw;
   onwrite = woclr;
  } intr_fld;
 };
addrmap foo {
 some_intr  event1;
 external transmit transmit;
 alias event1 some_intr
event1_for_dv;
};
```

Register Properties

Properties	Description	Dynamic
regwidth	Specifies the bit-width of the register (power of two)	No
accesswidth	Specifies the minimum software access width (power of two) operation that may be performed on the register	Yes
errextbus	The associated external register has error input	No
intr	Represents the inclusive OR of all the interrupt bits in a register after any field enable and/or field mask logic has been applied	No
shared	Defines a register as being shared in different address maps	No

RDL:

```
addrmap top {
  reg transmit{
    errextbus = true;
    regwidth = 32;
    field {
      hw = rw;
      sw = rw;
    } data;
  };
 external transmit transmit @0x0;
};
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Memory Component

A *memory* is an array of storage consisting of a number of entries of a given bit width. The physical memory implementation is technology dependent and memories shall be **external**.

Definitive memory definition

external mem_name [#(parameter_instance [, parameter_instance]*)]
mem_instance_element [, mem_instance_element]* ;

Anonymous memory definition

mem {[mem_body]} **external** mem_instance_element [,
mem_instance_element]* ;

Properties	Description	Dynamic
alignment	Specifies alignment of all instantiated components in the associated register file	No
sharedextbus	Forces all external registers to share a common bus	No
errextbus	For an external regfile, the associated regfile has an error input	No

RDL

```
mem fixed_mem #(longint unsigned
word_size = 32, longint unsigned
memory_size = word_size * 4096) {
  mementries = memory_size/word_size ;
  memwidth = word_size ;
} ;
```

Register File Components

- A *register file* is as a logical grouping of one or more register and register file instances.
- The only difference between the register file component (**regfile**) and the **addrmap** component is an **addrmap** defines an RTL implementation boundary where the **regfile** does not.

Definitive Register File Definition

[**external** | **internal**] *regfile_name* [**#(***parameter_instance* [**,** *parameter_instance*]***)**]
regfile_instance_element [**,** *regfile_instance_element*]* **;**

Anonymous Register File Definition

regfile {[*regfile_body*]**}**
[**external** | **internal**] *regfile_instance_element* [**,** *regfile_instance_element*]* **;**

Properties	Description	Dynamic
alignment	Specifies alignment of all instantiated components in the associated register file	No
sharedextbus	Forces all external registers to share a common bus	No
errextbus	For an external regfile, the associated regfile has an error input	No

```
regfile fifo_rfile {
   reg {field {} a;} a;
   reg {field {} a;} b;
};
regfile top_regfile {
  external fifo_rfile fifo_a;
  external fifo_rfile fifo_b[64];
  sharedextbus;
};

addrmap top{
   top_regfile top_regfile;
};
```

Addrmap

- An address component map (**addrmap**) contains registers, register files, memories, and/or other address maps and assigns a virtual address or final addresses.
- Specifies RTL module boundary

Definitive Definition

component new_component_name [**#(***parameter_definition* [**,** *parameter_definition*]***)**]
{[*component_body*]**}** [*instance_element* [**,** *instance_element*]*]**;**

Anonymous Definition

component **{**[*component_body*]**}** *instance_element* [**,** *instance_element*]***;**

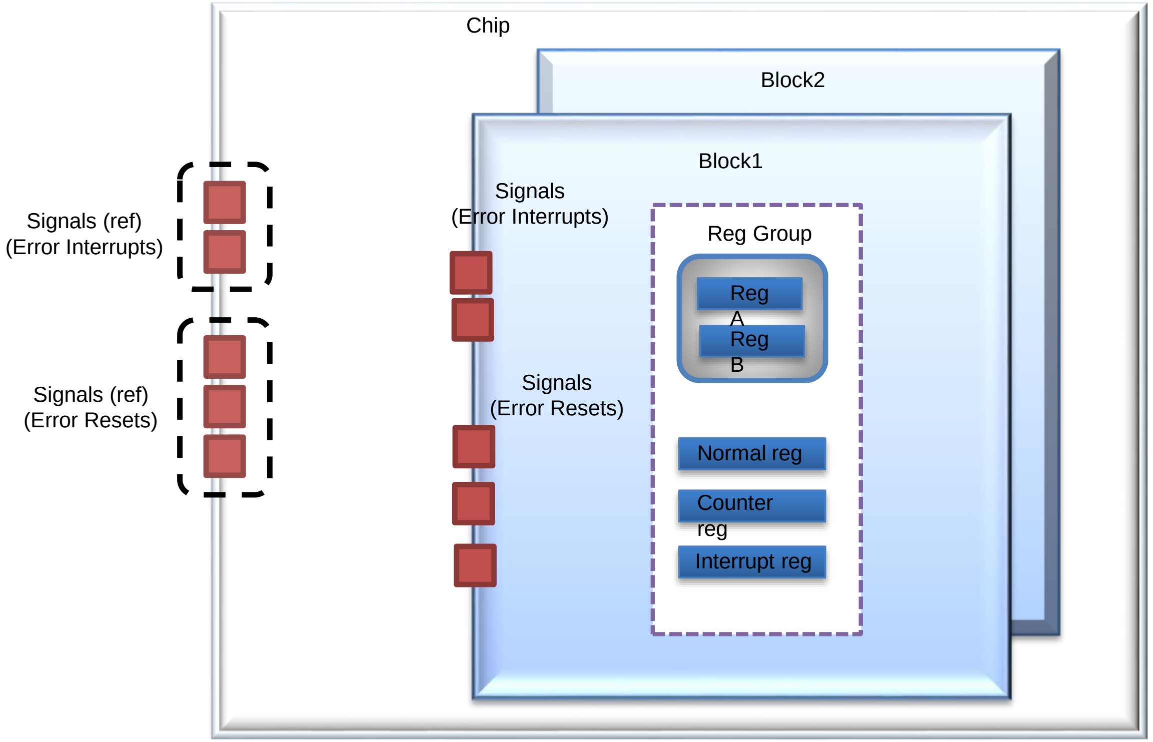
Properties	Description	Dynamic
alignment	Alignment of all instantiated components in the address map	No
sharedextbus	Forces all external registers to share a common bus	No
errextbus	The associated addrmap instance has an error input	No
littleendian	Uses little-endian architecture in the address map	Yes
addressing	Controls how addresses are computed in an address map	No
rsvdset	The read value of all fields not explicitly defined is set to 1 if rsvdset is True; otherwise, it is set to 0	No
rsvdsetx	The read value of all fields not explicitly defined is unknown if rsvd-setX is True	No
msb0	Specifies register bit-fields in an address map are defined as 0:N versus N:0	No
lsb0	Specifies register bit-fields in an address map are defined as N:0 versus N:0	No

Addrmap - Contd..

```
addrmap top{
  errextbus;
  rsvdset;
  reg reg1 {
    field {
    } fld1[31:20];
    field {
    } fld2[7:5];
  };
  reg reg2 {
    field {
    } fld1[32];
  };
reg1 reg1 @0x0;
external reg2 reg2 @0x4;
};
```

Signals

- “Signals” creates ports, at the block or chip level, and connect certain internal design signals to the external world.
- User can choose what gets connected to these signals and where these signals are used in the generated RTL using properties



Keyword	Description	Dynamic
signalwidth	Width of the signal	No
sync	Synchronous to the clock of the component	Yes
async	Asynchronous to the clock of the component	Yes
cpuif_reset	Default signal to use for resetting the software interface logic. This parameter only controls the CPU interface of a generated slave	Yes
field_reset	Default signal to use for resetting field implementations	Yes
active low	Signal is active low (state of 0 means ON)	Yes
active high	Signal is active high (state of 1 means ON)	Yes
resetsignal	Reference to the signal used to reset the field	Yes

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Signals - Contd..

```
addrmap top {
signal{activelow;async;field_reset;} pci_soft_reset;
signal{async;activelow;cpuif_reset;} pci_hard_reset;

  reg PCIE_REG_BIST {
    regwidth = 8;
      field {
        hw = rw;
        sw = r;
        fieldwidth = 4;
      } cplCode [3:0];
      field {
        hw = rw;
        sw = rw;
        fieldwidth = 1;
        resetsignal = pci_hard_reset;
      } capable [7:7]=0;
  };
 PCIE_REG_BIST PCIE_REG_BIST @0x0;
};
```

Instance address allocation

Instance Alignment

Address allocation operators

a) ***@*** *expression :* Specifies the address for the component instance.

b) **+=** *expression :* Specifies the address stride when instantiating an array of components (controls the spacing of the components).

c) **%=** *expression :* Specifies the alignment of the next address when instantiating a component (controls the alignment of the components).

Addressing Modes

Addressing Modes:

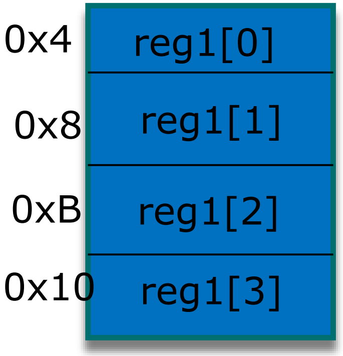
a) ***Compact*** *:* Specifies the components are packed tightly together while still being aligned to the accesswidth parameter

b) ***Regalign*** *:* Specifies the components are packed so each component's start address is a multiple of its size

c) ***fullalign*** *:* The assigning of addresses is similar regalign, except for arrays.

Offset (@)

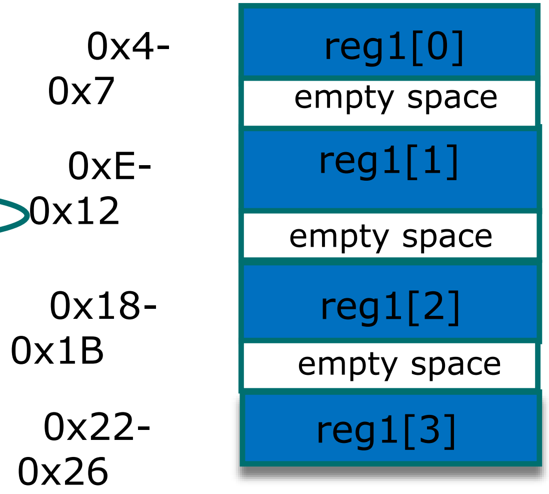
```
addrmap top {
  reg r1 {
   field { } f1[3:0];
  };
  r1 reg1[4] @0x4;
};
```



0x4	reg1[0]
0x8	reg1[1]
0xB	reg1[2]
0x10	reg1[3]

Stride (+=)

```
addrmap top {
  reg r1 {
   field { } f1[3:0];
  };
  r1 reg1[4] @0x4 += 10 ;
};
```

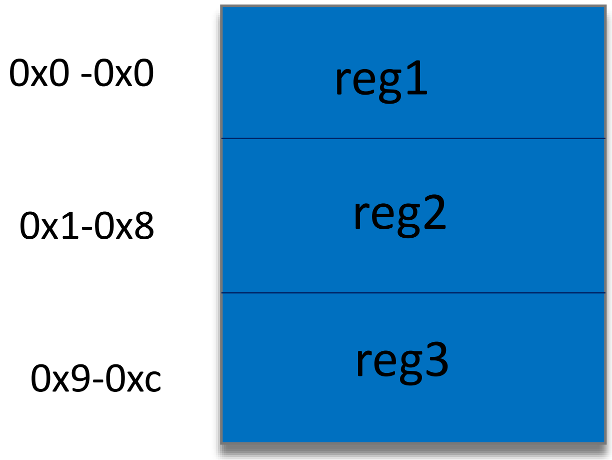


0x4- 0x7	reg1[0]
	empty space
0xE- 0x12	reg1[1]
	empty space
0x18- 0x1B	reg1[2]
	empty space
0x22- 0x26	reg1[3]

Compact

- It specifies the components are packed tightly together.

```
addrmap b1{
  addressing = compact;
  reg {
    regwidth = 8;
    field {
    } fld[7:0];
  } reg1;
  reg {
    regwidth=64;
    field {
    } fld1[63:0];
  } reg2;
  reg {
    regwidth = 32;
    field {
    } fld2[31:0];
  } reg3[20];
};
```



0x0 -0x0 reg1

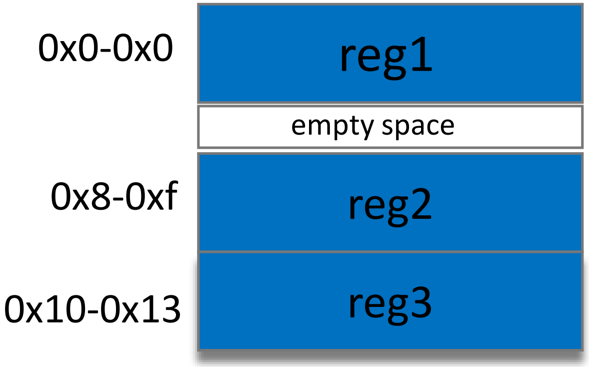
0x1-0x8 reg2

0x9-0xc reg3

Regalign

- It specifies the components are packed so each component's start address is a multiple of its size

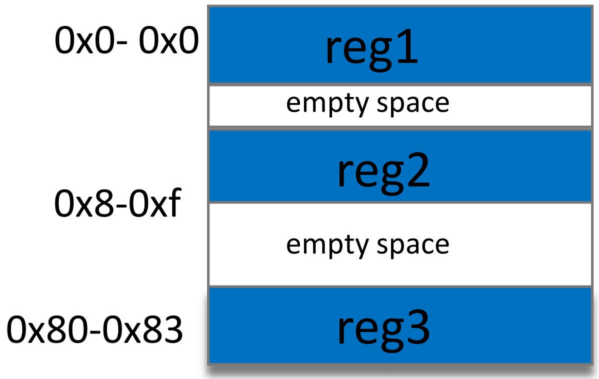
```
addrmap b1{
  addressing = regalign;
  reg {
    regwidth = 8;
    field {
    } fld[7:0];
  } reg1;
  reg {
    regwidth=64;
    field {
    } fld1[63:0];
  } reg2;
  reg {
    regwidth = 32;
    field {
    } fld2[31:0];
  } reg3;
};
```



Fullalign

- The assigning of addresses is similar regalign, except for arrays.
- The alignment value for the first element in an array is the size in bytes of the whole array (i.e., the size of an array element multiplied by the number of elements), rounded up to nearest power of two.

```
addrmap b1{
  addressing = fullalign;
  reg {
    regwidth = 8;
    field {
    } fld[7:0];
  } reg1;
  reg {
    regwidth=64;
    field {
    } fld1[63:0];
  } reg2;
  reg {
    regwidth = 32;
    field {
    } fld2[31:0];
  } reg3[20];
};
```



Enumerations

- It encloses a set of constant named integral values into the enumeration’s scope

Syntax: An *enum component definition* appears as follows.

enum *enum_name* **{** *encoding*; [*encoding*;]* **};**

Enumerator references shall be prefixed with their enumerated type name and two colons (::), e.g., MyEnumeration::MyValue.

Keyword	Description	Dynamic
enum	It encloses a set of constant named integral values into the enumeration’s scope	no
encode	Binds an enumeration to a field.	Yes

```
enum Enum1 {
 VAL1 = 3'h0 ;
 VAL2 = 3'h1 ;
} ;
enum Enum2 {
 VAL11 = 3'h0 ;
 VAL22 = 3'h1 ;
 VAL33 = 3'h2 ;
} ;
property MyUDP { component = addrmap ; type = Enum1;};
addrmap top {
  reg some_reg { field {} a[3] ; } ;
    addrmap {
      MyUDP = Enum1::VAL1 ; // Allowed
      some_reg regA ;
      regA.a -> reset = Enum1::VAL2 + Enum2::VAL33;
    } submap1 ;
    addrmap {
      reg {
       field {
        hwclr=longint'(Enum1::VAL1) ==
longint'(Enum2::VAL11);
       } b;
      } other_shared_reg ;
    } submap2 ;
};
```

Defining component parameters

- All definitive component types, except enumerations and constraints, may be parameterized using Verilog-style parameters.

```
 reg myReg #(longint unsigned SIZE =32){
  regwidth = SIZE;
  field {
  } data[SIZE - 1];
};
addrmap myAmap {
  myReg reg32;
  myReg reg32_arr[8];
  myReg #(.SIZE(16)) reg16;
  myReg #(.SIZE(8)) reg8;
 };
```

Parameter used

Parameter override during instantiation

Struct

Syntax: A **struct** definition appears as follows.

[**abstract**] **struct** *struct_name* [**:** *base_struct_name*]

{{*member_type member_name*;}*****};**

Deriving structures

A **struct** declaration may *derive* from another **struct** by specifying the base **struct**'s name after a colon (:),

```
struct base_struct {
 bit foo ;
} ;

struct derived_struct : base_struct {
 longint unsigned bar ;
} ;

struct final_struct : derived_struct {
 // final_struct's members are foo, bar, and baz.
 string baz ;
} ;
```

```
struct configIP {
 boolean Reg1_is_present;
 boolean Reg2_is_present;
};
struct configTop {
 configIP IP1;
 configIP IP2;
};
addrmap ip #(configTop t){
 reg r1 {
  ispresent = t.IP1.Reg1_is_present;
  field {}f1;
 };
 reg r2{
  ispresent = t.IP2.Reg2_is_present;
  field {}f1;
 };
 r1 r1;
 r2 r2;
};
addrmap top {
 ip #(.t(configTop'{IP1:configIP'{Reg1_is_present:true},
           IP2:configIP'{Reg2_is_present:false} } ) ) ip1;
 ip #(.t( configTop'{IP1:configIP'{Reg1_is_present:false},
            IP2:configIP'{Reg2_is_present:true} } ) ) ip2;
};
```

Property Assignment

1

Dynamic Assignment

- When a property is assigned after the component is instantiated, the assignment itself is referred to as a *dynamic assignment*.

Syntax:

instance_name -> property_name [= value];

```
reg {
 default name =”def
 name”;
 field f_type {
  name = “other name”;
 };
 f_type f1;
 f1->name = “Dynamic
Assignment”;
} some_reg;
```

2

Property Assignment

- A specific property shall only be set once per scope.

Syntax:

property_name[=expression];

```
reg {
 default name =”def
 name”;
 field f_type {
  name = “other name”;
 };
 f_type f1;
 f1->name = “Dynamic
Assignment”;
} some_reg;
```

3

Default Property Assignment

- A specific property **default** value shall only be set once per scope.

Syntax

default property_name [= value];

```
reg {
 default name =”def
 name”;
 field f_type {
  name = “other name”;
 };
 f_type f1;
 f1->name = “Dynamic
Assignment”;
} some_reg;
```

4

SystemRDL Default Value for Property type

- Property takes its default value

```
reg {
 default name =”def
 name”;
 field f_type {
  name = “other name”;
  we;
 };
 f_type f1;
 f1->name = “Dynamic
Assignment”;
} some_reg;
```

Interrupt

- Interrupt is a signal generated and sent to the processor by hardware or software indicating an event that needs attention

Keyword	Description
intr	Interrupt, part of interrupt logic for a register
posedge	Interrupt when next goes from low to high
negedge	Interrupt when next goes from high to low
bothedge	Interrupt when next changes value
level	Interrupt while the next value is asserted and maintained (the default)
nonsticky	Defines a non-sticky (hierarchical) interrupt (not locked)
enable	Defines an interrupt enable; i.e., which bits in an interrupt field are used to assert an interrupt
mask	Defines an interrupt mask ; i.e., which bits in an interrupt field are not used to assert an interrupt
haltenable	Defines a halt enable (the inverse of haltmask); i.e., which bits in an interrupt field are set to de-assert the halt out.
haltmask	Defines a halt mask (the inverse of haltenable); i.e., which bits in an interrupt field are set to assert the halt out
sticky	Defines the entire field as sticky; i.e., the value of the associated interrupt field shall be locked until cleared by software (write or clear on read)

```
addrmap block_name {
    reg Status1 {
       regwidth = 32;
        field {
          hw = rw;
          sw = rw;
          onread = r;
          onwrite = woclr;
          intr;
         } Fld[31:0] = 32'h0;
    };
    reg Status2 {
       regwidth = 32;
        field {
          hw = rw;
          sw = rw;
          onread = r;
          onwrite = woclr;
          intr;
         } Fld[31:0] = 32'h0;
    };
     reg Enable1 {
       regwidth = 32;
        field {
          hw = rw;
          sw = rw;
          onread = r;
          onwrite = w;
         } Fld[31:0] = 32'h0;
   };
```

```
reg Mask1 {
       regwidth = 32;
        field {
          hw = rw;
          sw = rw;
          onread = r;
          onwrite = w;
         } Fld[31:0] = 32'h0;
  };
  Status1 Status1 @0x0000;
  Status2 Status2 @0x0004;
  Enable1 Enable1 @0x0008;
  Mask1   Mask1   @0x000C;
  Status1.Fld -> enable = Enable1.Fld;
  Status2.Fld -> mask = Mask1.Fld;
};
```

Counter

- A *counter* is a special purpose field which can be incremented or decremented by constants or dynamically specified values.

Keyword	Description
counter	Field implemented as a counter.
incrvalue	Increment counter by specified value.
decrvalue	Decrement counter by specified value.
incrsaturate	Indicates the counter saturates in the incrementing direction.
decrsaturate	Indicates the counter saturates in the decrementing direction.
Incrthreshold	Indicates the counter has a threshold in the incrementing direction.
decrthreshold	Indicates the counter has a threshold in the decrementing direction.
decrwidth	Width of the interface to hardware to control decrementing the counter externally.
incrwidth	Width of the interface to hardware to control incrementing the counter externally.
threshold	This is an alias of incrthreshold.
saturate	This is an alias of incrsaturate.
underflow	Underflow signal asserted when counter underflows or wraps.
overflow	Overflow signal asserted when counter overflows or wraps.
incr	The counter increment is controlled by another component or signal (active high).
decr	The counter decrement is controlled by another component or signal (active high).

```
addrmap block_name {
  reg incr_reg {
     regwidth = 32;
      field {
       hw = na;
       sw = rw;
       counter;
       incrvalue = 2;
       incrsaturate = 15;
       incrthreshold = 10;
     } Fld[31:0] = 32'h0;
  };
 reg decr_reg {
     regwidth = 32;
      field {
       hw = na;
       sw = rw;
       counter;
       decrvalue = 2;
       decrthreshold = 10;
       decrsaturate = 5;
     } Fld[31:0] = 32'h0;
  };
 incr_reg incr_reg @0x0000;
 decr_reg decr_reg @0x0004;
};
```

HDL PATH

- By specifying an HDL path, the verification environment can have direct access to memory, register, and field implementation nets in a Design Under Test (DUT).

An **hdl_path_slice** or **hdl_path_gate_slice** can be put on a **field** or **mem** component. It can be used when the corresponding RTL or gate-level netlist is not contiguous.

Syntax:

hdl_path = "*path*";

hdl_path_gate = "*path*";

hdl_path_slice = ‘{"*path*" [, "*path*"]*};

hdl_path_gate_slice = ‘{"*path*" [, "*path*"]*};

Property	Description	Dynamic
hdl_path	Assigns the RTL hdl_path for an addrmap, reg, or regfile	Yes
hdl_path_slice	Assigns a list of RTL hdl_path for a field or mem	Yes
hdl_path_gate	Assigns the gate-level hdl_path for an addrmap, reg, or regfile	Yes
hdl_path_gate_slice	Assigns a list of gate-level hdl_path for a field or mem	Yes

```
addrmap blk_def #(string ext_hdl_path = "ext_block"){
   hdl_path = "int_block" ;
   reg {
     hdl_path = { ext_hdl_path, ".externl_reg" } ;
       field {
         hdl_path_slice = '{ "field1" } ;
       } f1 ;
   } external external_reg ;
   reg {
     hdl_path = "int_reg" ;
       field {
         hdl_path_slice = '{ "field1" } ;
       } f1 ;
   } internal_reg ;
} ;
addrmap top {
  hdl_path = "TOP" ;
  blk_def #( .ext_hdl_path("ext_block0")) int_block0 ;
  int_block0 -> hdl_path = "int0" ;
  blk_def #( .ext_hdl_path("ext_block1")) int_block1 ;
  int_block1 -> hdl_path = "int1" ;
};
```

Constraint

- A *constraint* is a value-based condition on one or more components; e.g., constraint-driven test generation allows users to automatically generate tests for functional verification.

Definitive definition

constraint *constraint_component_name*
{[*constraint_body*]**};**
*constraint_component_name constraint_inst***;**

Anonymous definition

constraint {[*constraint_body*]**}**
*constraint_component_name***;**

Property	Description	Dynamic
constraint_disable	Specifies whether to disable (true) or enable (false) constraints	Yes

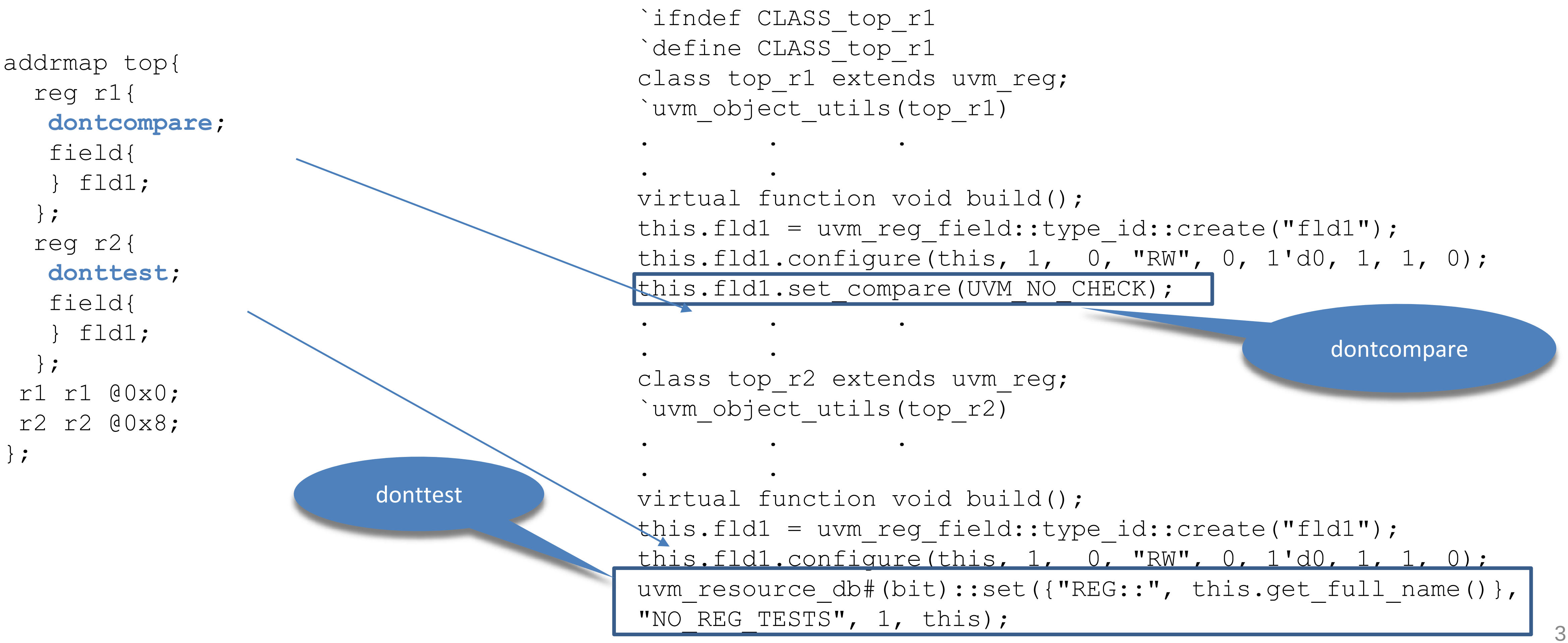
```
constraint max_value { this < 256; };
enum color {
 red = 0 { desc = " color red ";};
 green = 1 { desc = " color green ";};
};
reg register1 {
  field {
  } limit[0:2]= 0;
  field {
    max_value max1;
  } f1[3:9]= 3;
  field {
    encode=color;
    constraint{this inside{color::red,color::green};}rg1;
  } f2[10:31];
};
addrmap constraint_component_example {
  register1 reg1;
  register1 reg2;
  reg2.f2.rg1->constraint_disable = true;
};
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Structural Testing

1) dontcompare : This is testing property indicates the components read data shall be discarded and not compared against expected results.

2) donttest : This testing property indicates the component is not included in structural testing.

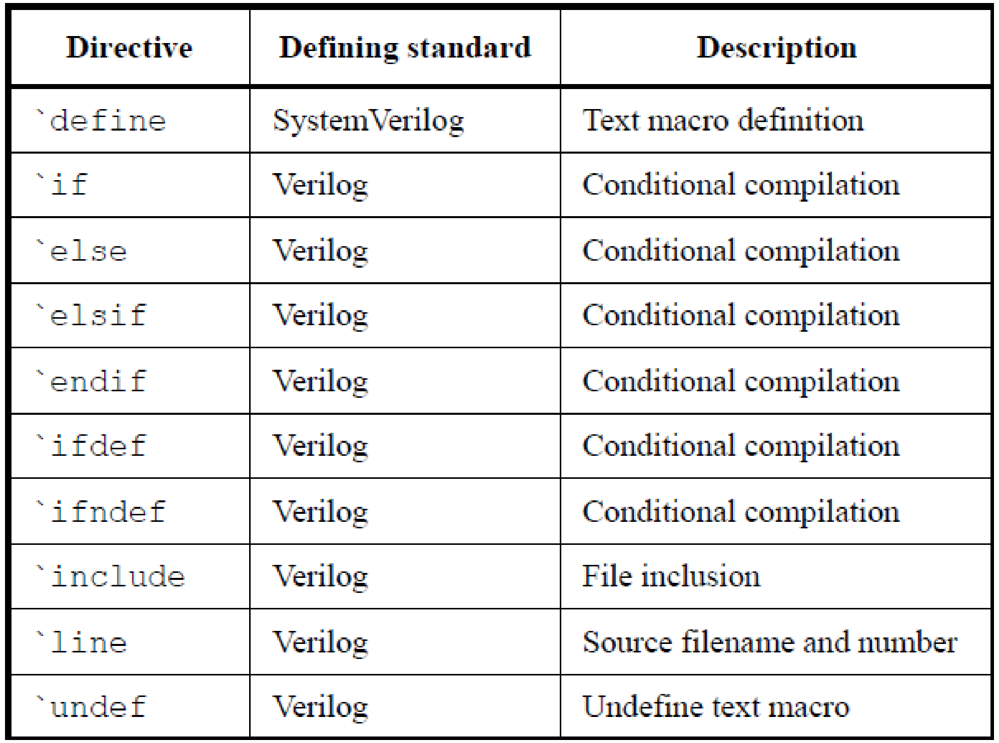


```
addrmap top{
  reg r1{
   dontcompare;
   field{
   } fld1;
  };
  reg r2{
   donttest;
   field{
   } fld1;
  };
 r1 r1 @0x0;
 r2 r2 @0x8;
};
```

```
`ifndef CLASS_top_r1
`define CLASS_top_r1
class top_r1 extends uvm_reg;
`uvm_object_utils(top_r1)
.        .        .
.        .
virtual function void build();
this.fld1 = uvm_reg_field::type_id::create("fld1");
this.fld1.configure(this, 1,  0, "RW", 0, 1'd0, 1, 1, 0);
this.fld1.set_compare(UVM_NO_CHECK);
.        .        .
.        .
class top_r2 extends uvm_reg;
`uvm_object_utils(top_r2)
.        .        .
.        .
virtual function void build();
this.fld1 = uvm_reg_field::type_id::create("fld1");
this.fld1.configure(this, 1,  0, "RW", 0, 1'd0, 1, 1, 0);
uvm_resource_db#(bit)::set({"REG::", this.get_full_name()},
"NO_REG_TESTS", 1, this);
```

SystemRDL with Embedded Perl

- Perl snippets shall begin with **<%** and be terminated by **%>**; between these markers any valid Perl syntax may be used.
- Any SystemRDL code outside of the Perl snippet markers is equivalent to the Perl print 'RDL code' and the resulting code is printed directly to the post-processed output.
- **<%=$VARIABLE%>** (no whitespace is allowed) is equivalent to the Perl print $VARIABLE.
- The resulting Perl code is interpreted, and the result is sent to the traditional Verilog-style preprocessor.



Directive	Defining standard	Description
`define	SystemVerilog	Text macro definition
`if	Verilog	Conditional compilation
`else	Verilog	Conditional compilation
`elsif	Verilog	Conditional compilation
`endif	Verilog	Conditional compilation
`ifdef	Verilog	Conditional compilation
`ifndef	Verilog	Conditional compilation
`include	Verilog	File inclusion
`line	Verilog	Source filename and number
`undef	Verilog	Undefine text macro

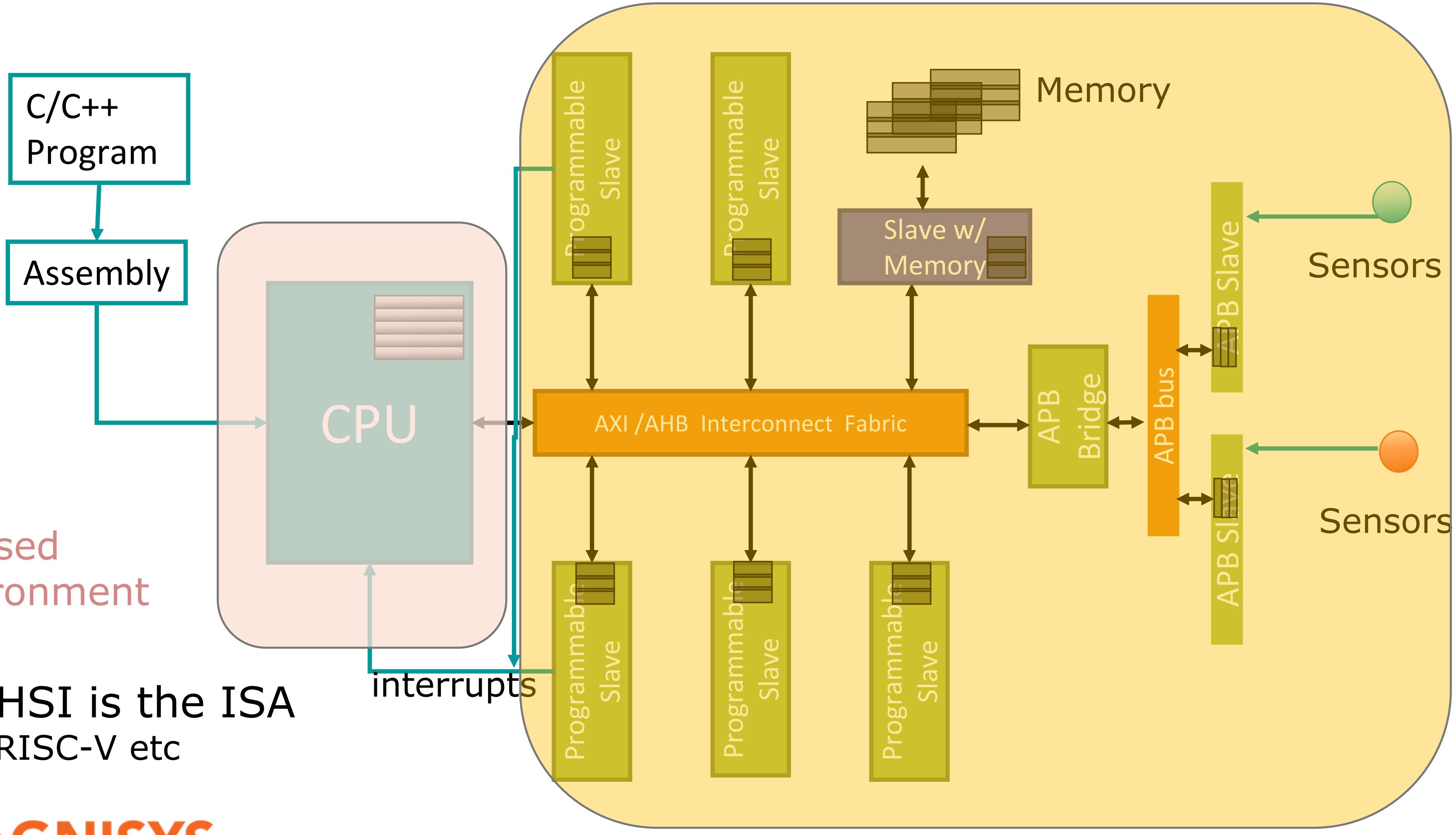
```
reg myReg { <% for( $i = 0; $i < 6; $i += 2 ) {
%> myField data<%=$i%> [<%=$i+1%>:<%=$i%>]; <% } %>
};
```

→

```
reg myReg {
 myField data0 [1:0];
 myField data2 [3:2];
 myField data4 [5:4];
};
```

SoC HW/SW Interface Layer

The slaves are programmed by reading/writing to the embedded register



SV-UVM environment

C based environment

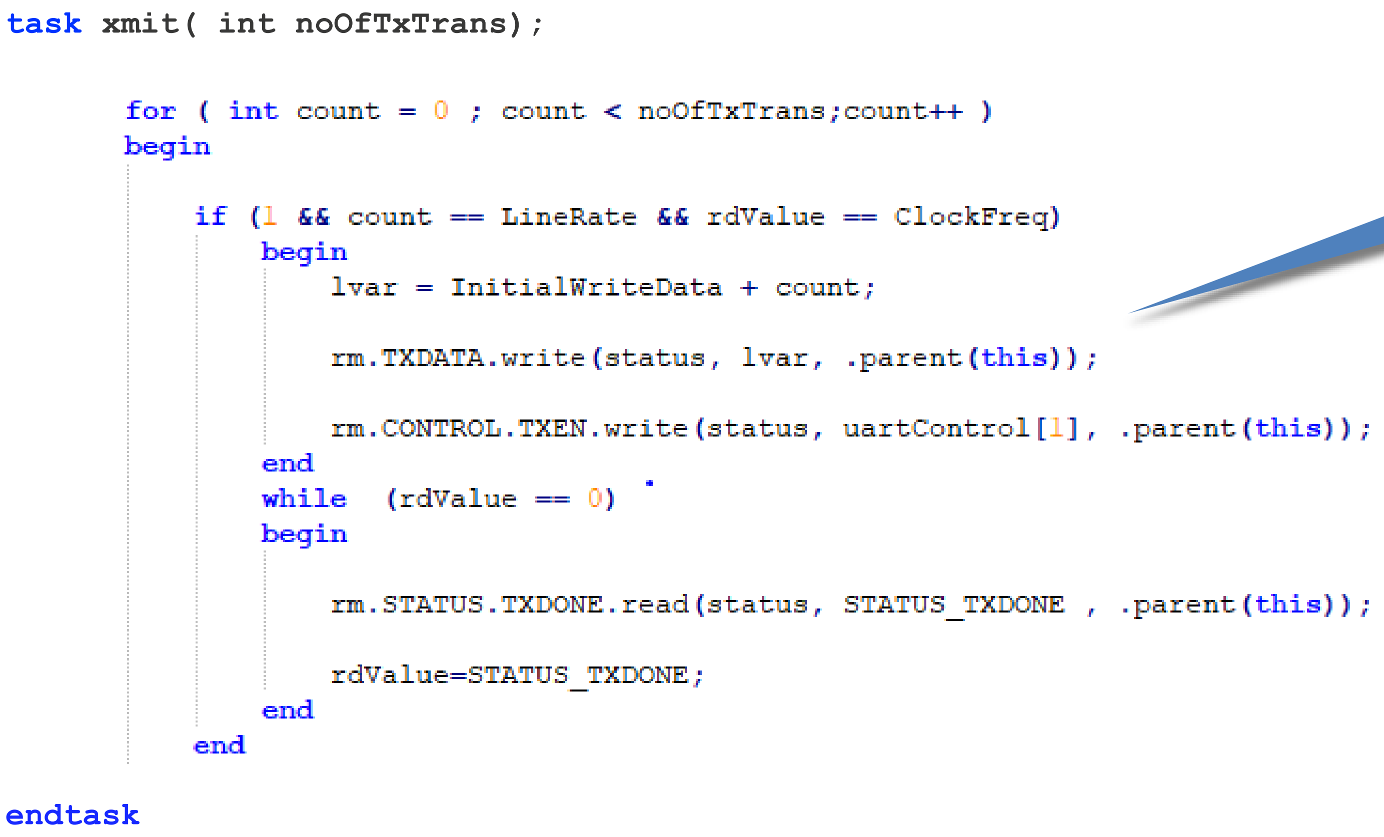
CPU HSI is the ISA

ARM, RISC-V etc

IP (slave) HSI are the registers, interrupts

Example Sequence with HSI

As an example, the code below is a SV task that is manually coded by the user. It shows that HSI is a critical part of a sequence to achieve a certain behavior in the target device.



```
task xmit( int noOfTxTrans);

      for ( int count = 0 ; count < noOfTxTrans;count++ )
      begin

          if (1 && count == LineRate && rdValue == ClockFreq)
              begin
                  lvar = InitialWriteData + count;

                  rm.TXDATA.write(status, lvar, .parent(this));

                  rm.CONTROL.TXEN.write(status, uartControl[1], .parent(this));
              end
              while  (rdValue == 0)
              begin

                  rm.STATUS.TXDONE.read(status, STATUS_TXDONE , .parent(this));

                  rdValue=STATUS_TXDONE;
              end
          end

endtask
```

Writing a Register

Writing a Field

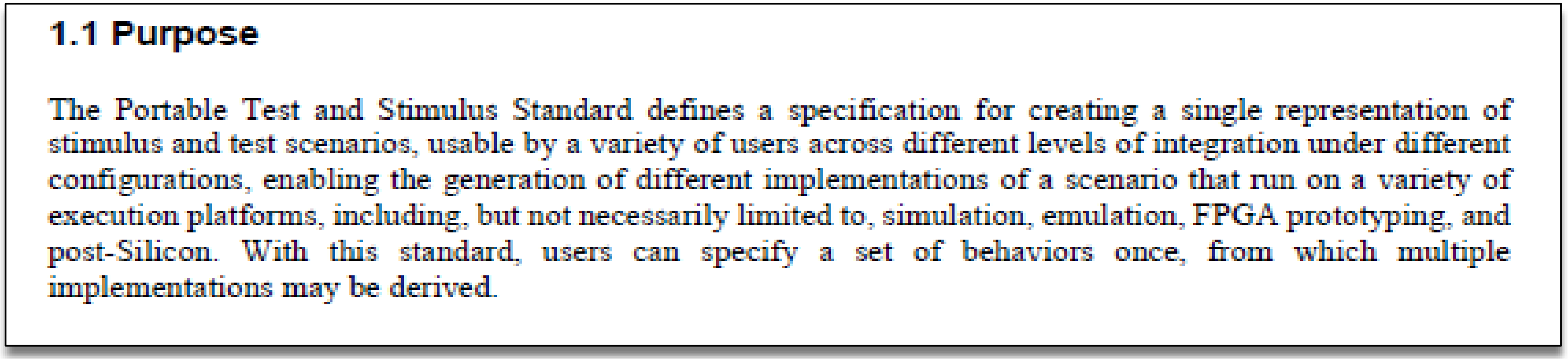
Reading a Field

Introduction to Sequences

- Sequences are built on registers, memories, pins
- Sequences contain
  - Register / Field Writes
  - Register / Field Reads
  - Pin Manipulation Commands
  - Wait / Function calls, sub sequence calls
- Information about Registers/Memories can be in any format
  - IP-XACT
  - SystemRDL
  - Word / Excel
  - Text files

Portable Stimulus Standard

- PSS 1.0 Standard was released in June 2018



1.1 Purpose

The Portable Test and Stimulus Standard defines a specification for creating a single representation of stimulus and test scenarios, usable by a variety of users across different levels of integration under different configurations, enabling the generation of different implementations of a scenario that run on a variety of execution platforms, including, but not necessarily limited to, simulation, emulation, FPGA prototyping, and post-Silicon. With this standard, users can specify a set of behaviors once, from which multiple implementations may be derived.

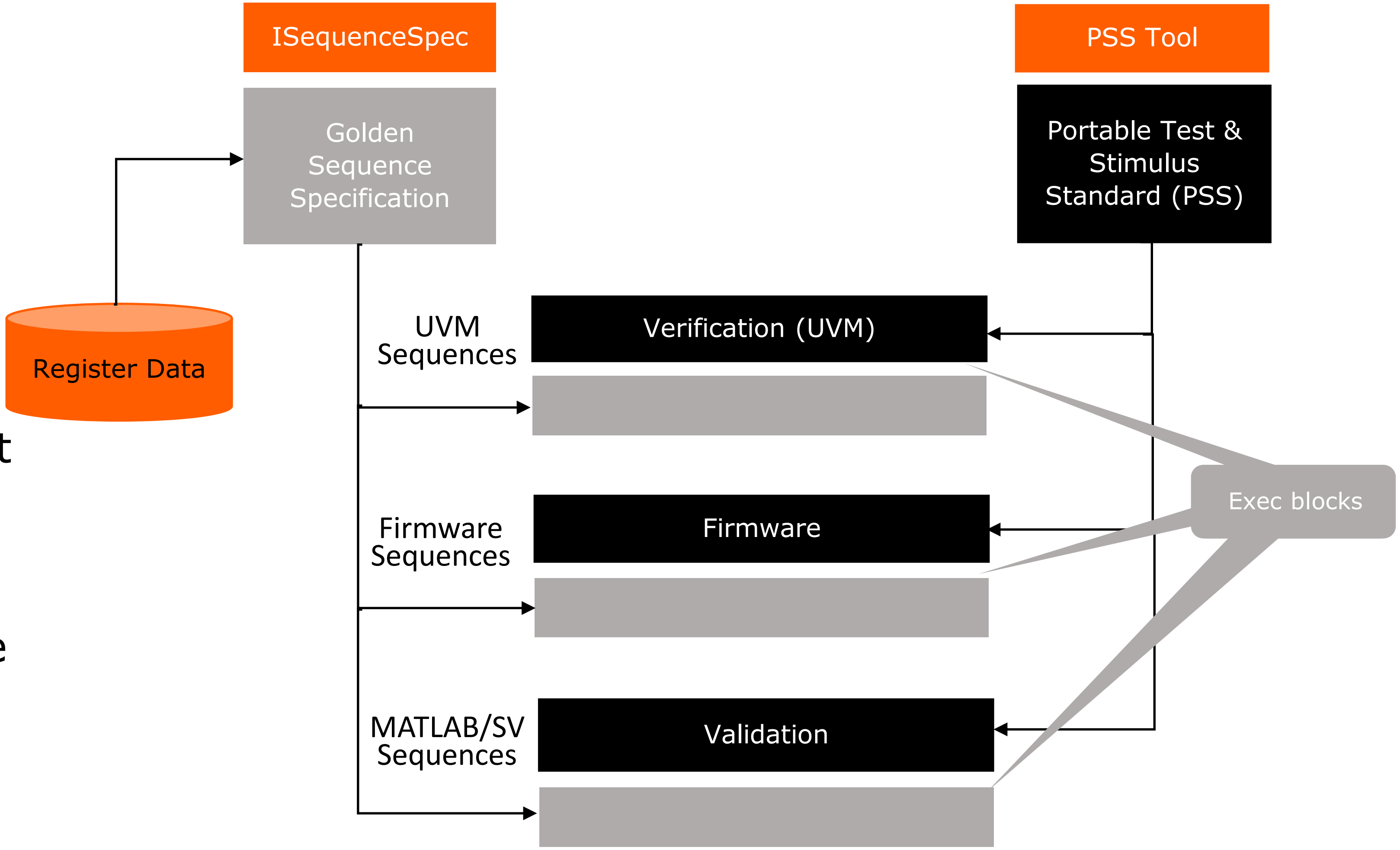
- Powerful concepts of PSS: Abstraction and Reuse
- PSS is useful for high-level test scenario creation
  - Modeling Data flow
  - Modeling Behavior
  - Constraints, Randomization, Coverage
- Actions are a key abstraction unit – can model the scenarios and include exec blocks
- The implementation-level tests are handled by “exec blocks”

Portable Stimulus Standard – contd.

- PSS helps automate the testing process, thereby reducing the time to generate complex use-case scenarios
- It can generate tests, 10x faster than hand coding
- Portability from IP to sub-system to SoC level, including hardware- software can be achieved
- However, low level implementation sequences need to be created manually

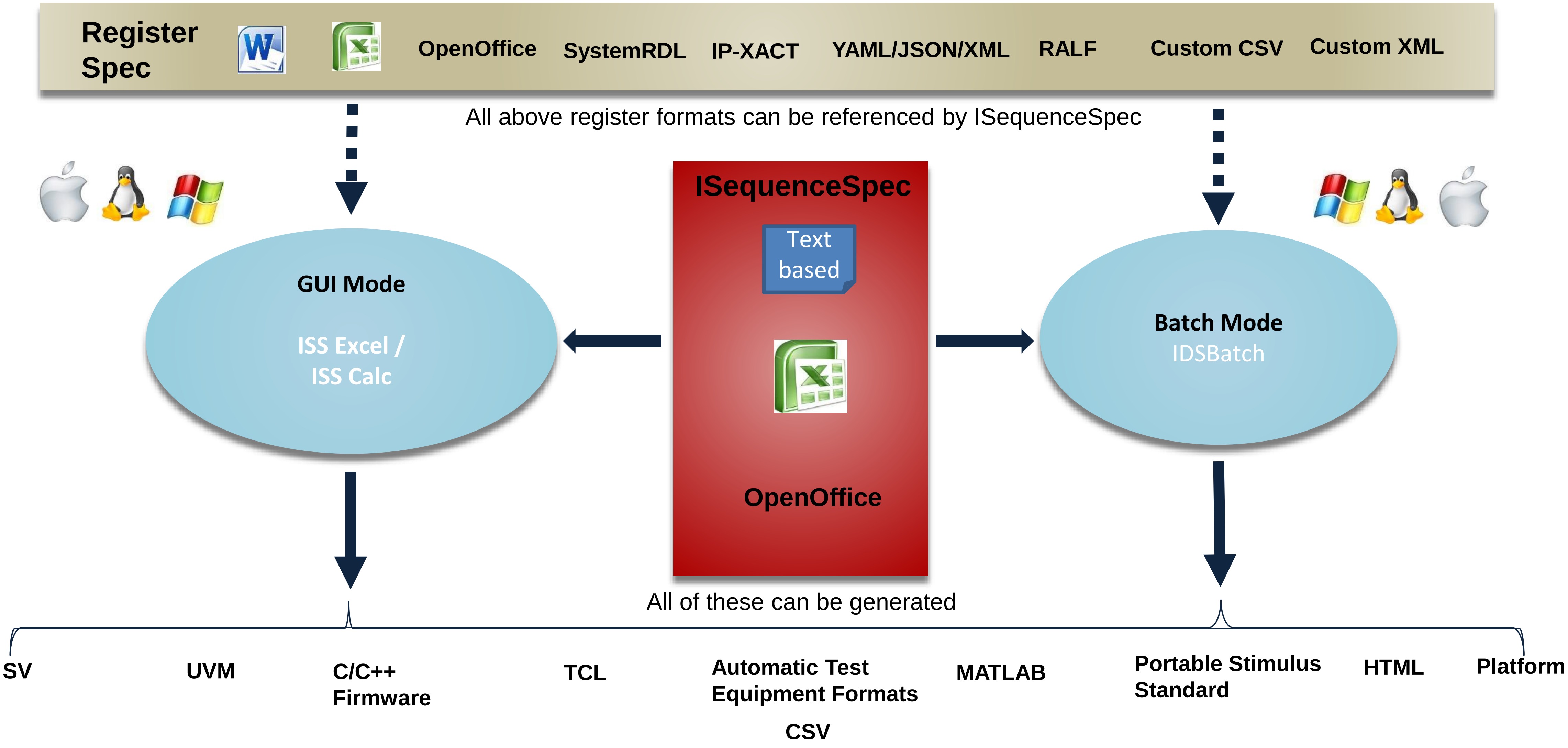
ISequenceSpec™ + PSS Proposed Tool Flow

- Capture sequences in pseudo-code in the golden spec (spreadsheet or text)
- Generate sequences in multiple formats (C, System Verilog, UVM)
- PSS tool user creates the test scenarios and calls the exec blocks generated by ISS
- PSS tool user synthesizes the tests/scenarios and generates the required files for the target platform



ISequenceSpec™ Suite

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY



Register Spec | OpenOffice | SystemRDL | IP-XACT | YAML/JSON/XML | RALF | Custom CSV | Custom XML

All above register formats can be referenced by ISequenceSpec

ISequenceSpec

Text based

OpenOffice

GUI Mode

ISS Excel / ISS Calc

Batch Mode

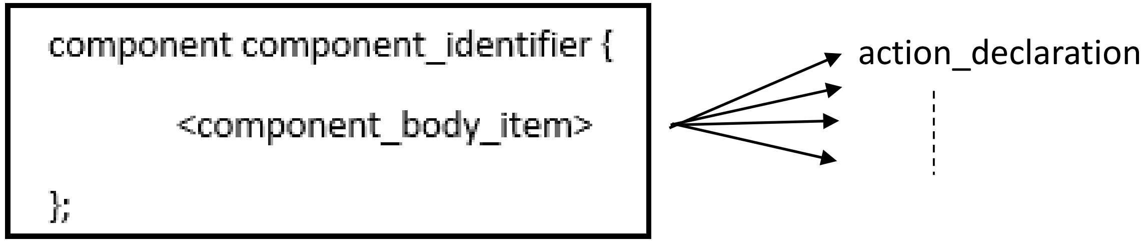
IDSBatch

All of these can be generated

SV | UVM | C/C++ Firmware | TCL | Automatic Test Equipment Formats | CSV | MATLAB | Portable Stimulus Standard | HTML | Platform

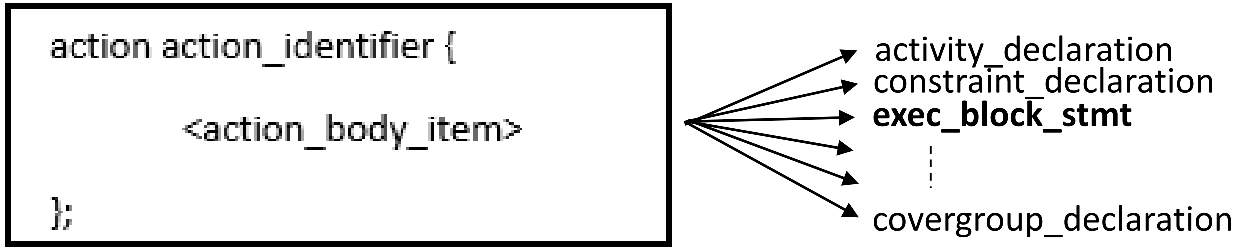
PSS Language Constructs

- **Component**: A structural entity, defined per type and instantiated under other components.



```
component component_identifier {
        <component_body_item>
};
```

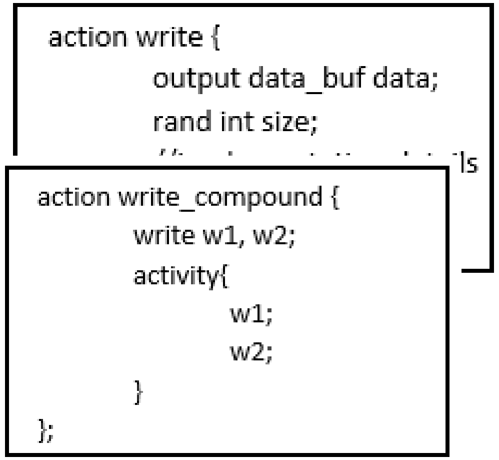
- **Action**: An element of behavior.



```
action action_identifier {
        <action_body_item>
};
```

  - <u>**Atomic action**</u>: An action that corresponds directly to operations of the underlying system under test (SUT) and test environment.

  - <u>**Compound action**</u>: An action which is defined in terms of one or more sub-actions.



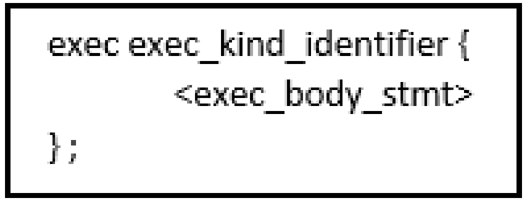
```
action write {
        output data_buf data;
        rand int size;
        //                          ls
```

```
action write_compound {
        write w1, w2;
        activity{
                w1;
                w2;
        }
};
```

- **Activity**: An abstract, partial specification of a scenario that is used in a compound action to determine the high-level intent and leaves all other details open.

PSS Language Constructs – contd.

- **Exec block**: Specifies the mapping of PSS scenario entities to its non-PSS implementation.



```
exec exec_kind_identifier {
        <exec_body_stmt>
};
```

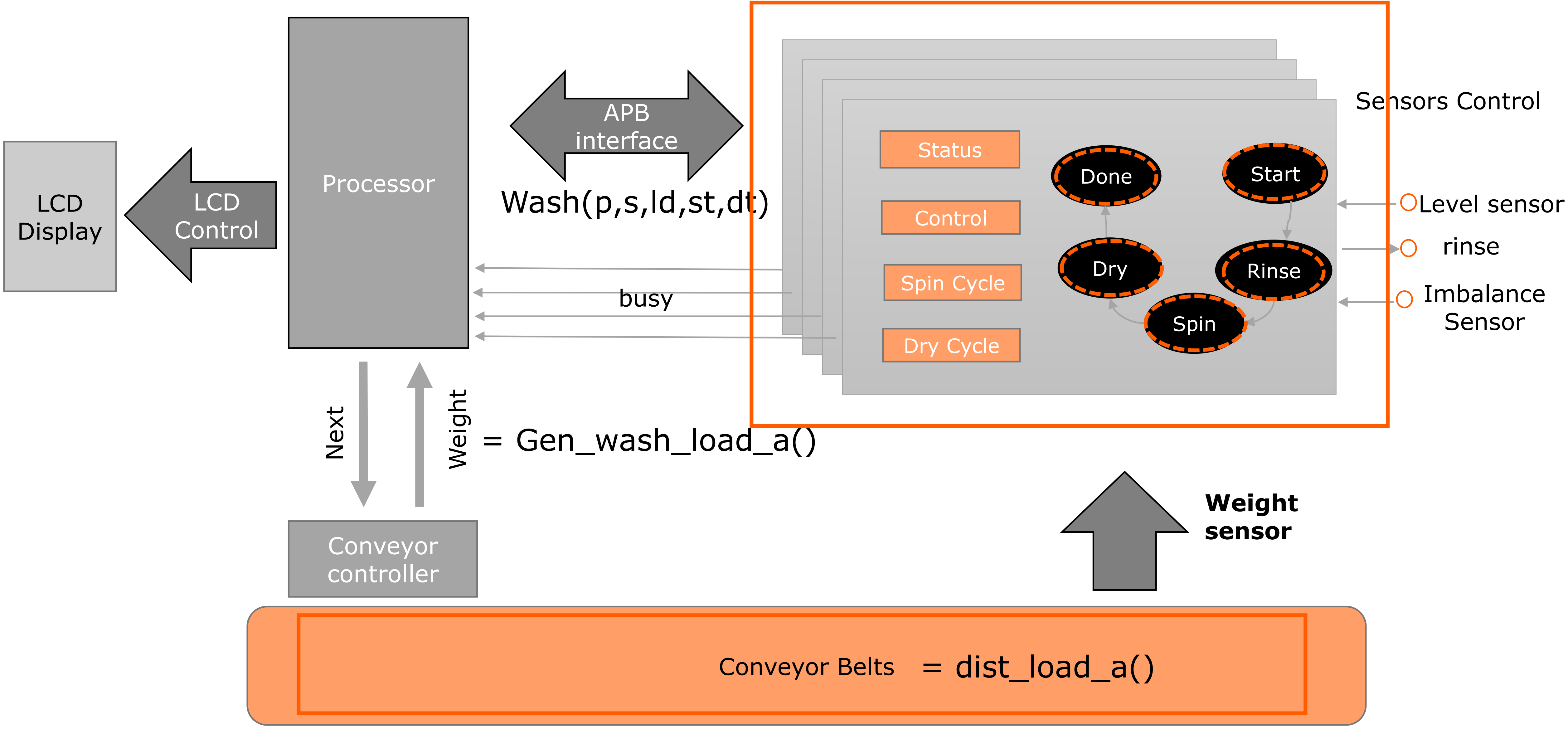
exec_kind_identifier: pre_solve | post_solve | body | header | declaration | run_start | run_end | init

- Exec block connects the PSS code to low level implementation tasks/functions/sequences.

DUT : An Industrial Washer Control System

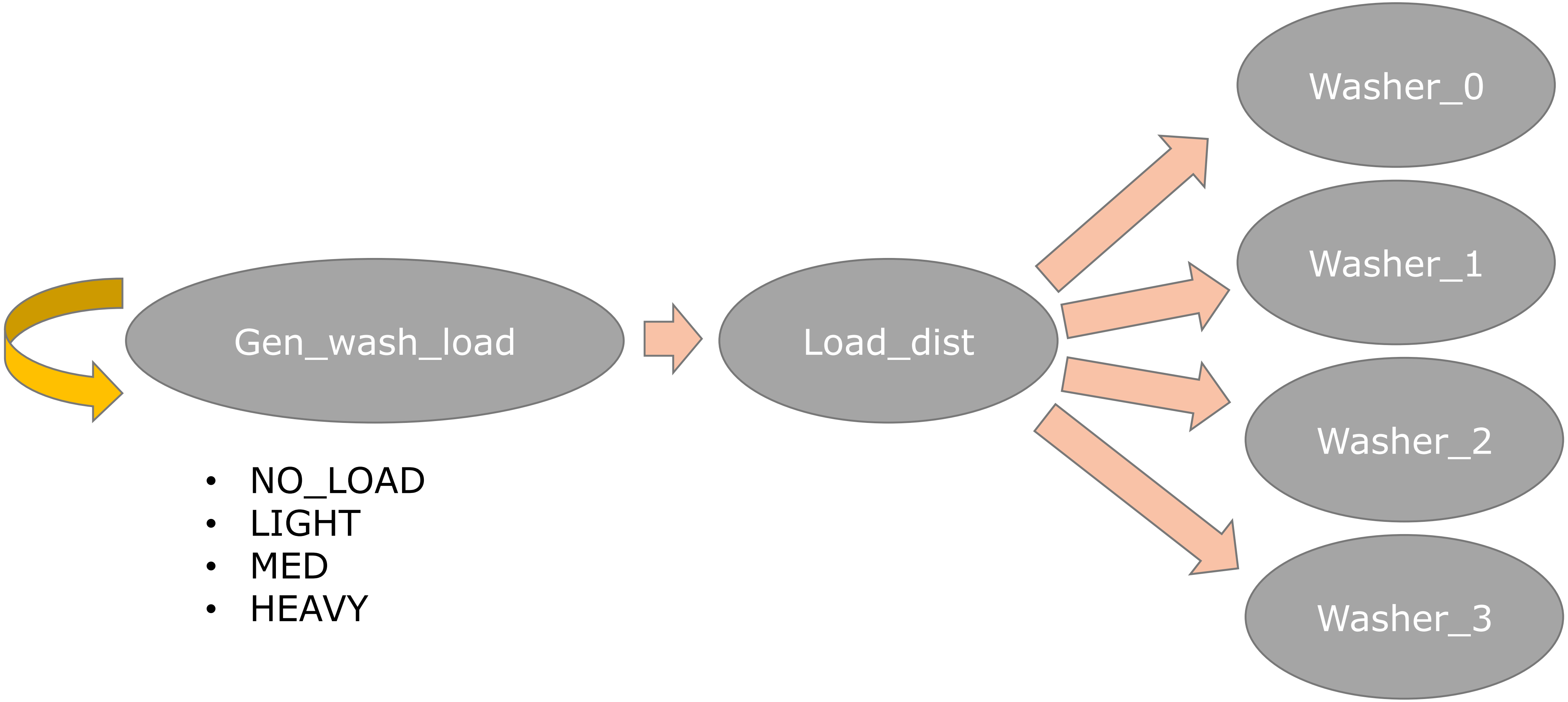
- Microprocessor based system with Reg bus
- Four settable registers
  - Status
  - Control
  - Spin time
  - Dry time
- The weight of the payload determines the Spin and Dry time
  - Using two distinct lookup tables
- The Imbalance sensor output is stored in the error field of the status register.

The Design



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

PSS Verification Intent



Gen_wash_load

Load_dist

Washer_0

Washer_1

Washer_2

Washer_3

- NO_LOAD
- LIGHT
- MED
- HEAVY

Washer_0:

- BUSY
- IDLE

Washer_1:

- BUSY
- IDLE

Washer_2:

- BUSY
- IDLE

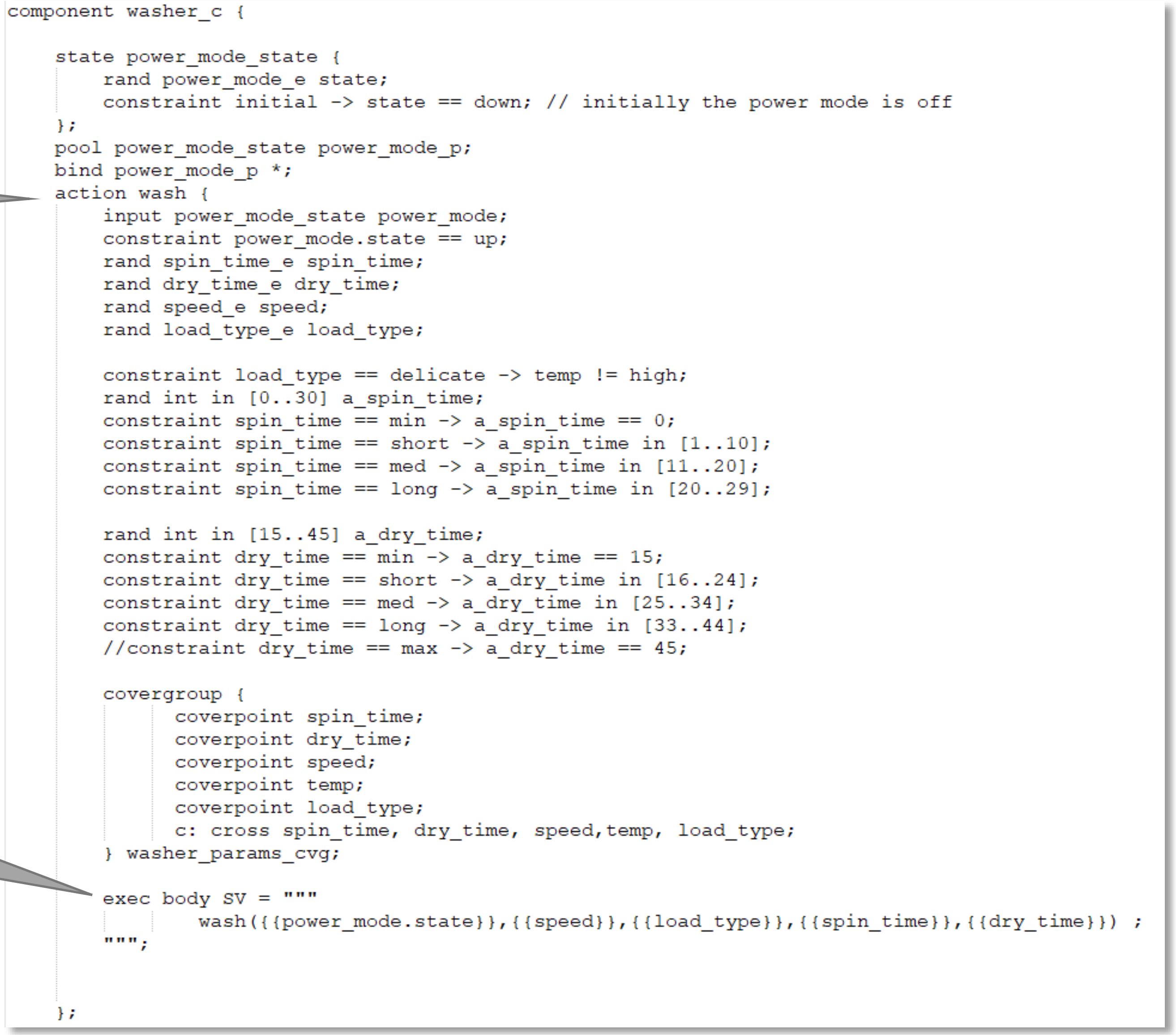
Washer_3:

- BUSY
- IDLE

PSS Code

PSS action blocks containing the Exec Block

"exec" blocks generated by ISequenceSpec containing SV tasks.

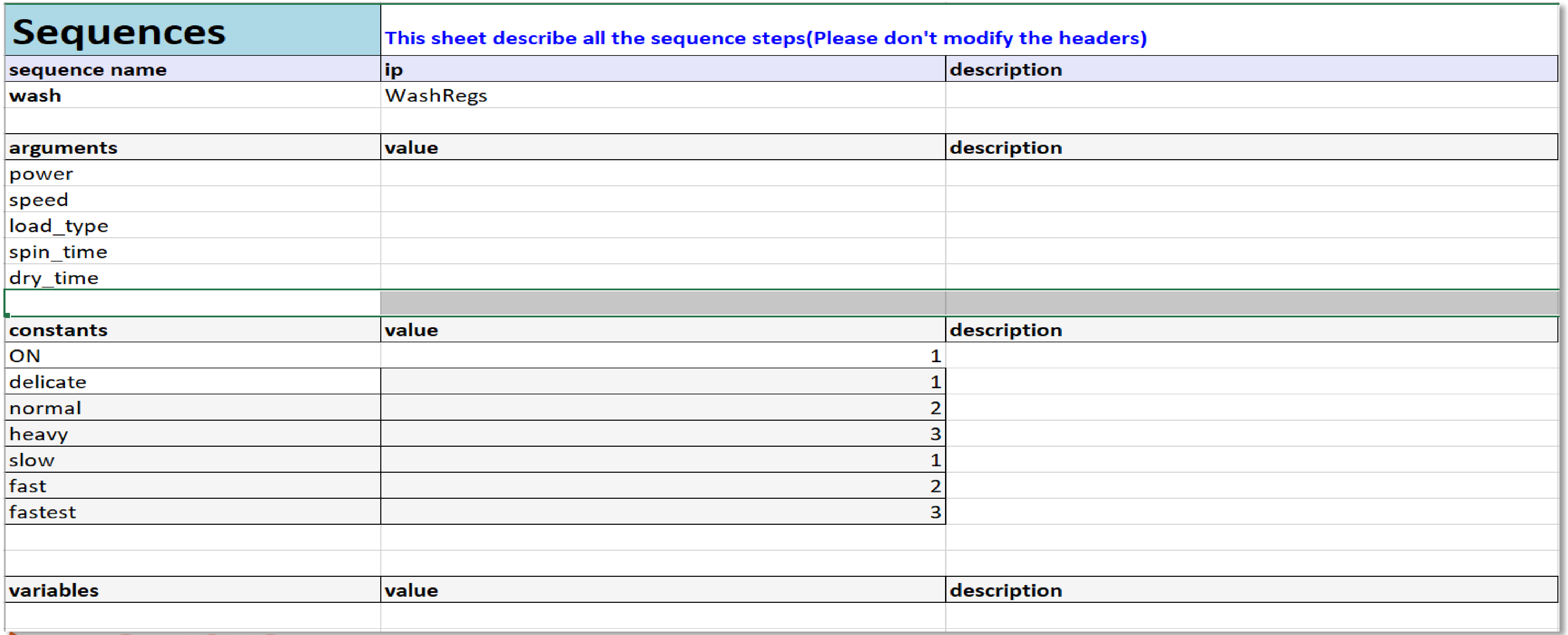


```
component washer_c {

    state power_mode_state {
        rand power_mode_e state;
        constraint initial -> state == down; // initially the power mode is off
    };
    pool power_mode_state power_mode_p;
    bind power_mode_p *;
    action wash {
        input power_mode_state power_mode;
        constraint power_mode.state == up;
        rand spin_time_e spin_time;
        rand dry_time_e dry_time;
        rand speed_e speed;
        rand load_type_e load_type;

        constraint load_type == delicate -> temp != high;
        rand int in [0..30] a_spin_time;
        constraint spin_time == min -> a_spin_time == 0;
        constraint spin_time == short -> a_spin_time in [1..10];
        constraint spin_time == med -> a_spin_time in [11..20];
        constraint spin_time == long -> a_spin_time in [20..29];

        rand int in [15..45] a_dry_time;
        constraint dry_time == min -> a_dry_time == 15;
        constraint dry_time == short -> a_dry_time in [16..24];
        constraint dry_time == med -> a_dry_time in [25..34];
        constraint dry_time == long -> a_dry_time in [33..44];
        //constraint dry_time == max -> a_dry_time == 45;

        covergroup {
              coverpoint spin_time;
              coverpoint dry_time;
              coverpoint speed;
              coverpoint temp;
              coverpoint load_type;
              c: cross spin_time, dry_time, speed,temp, load_type;
        } washer_params_cvg;

        exec body SV = """
                wash({{power_mode.state}},{{speed}},{{load_type}},{{spin_time}},{{dry_time}}) ;
        """;

    };
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

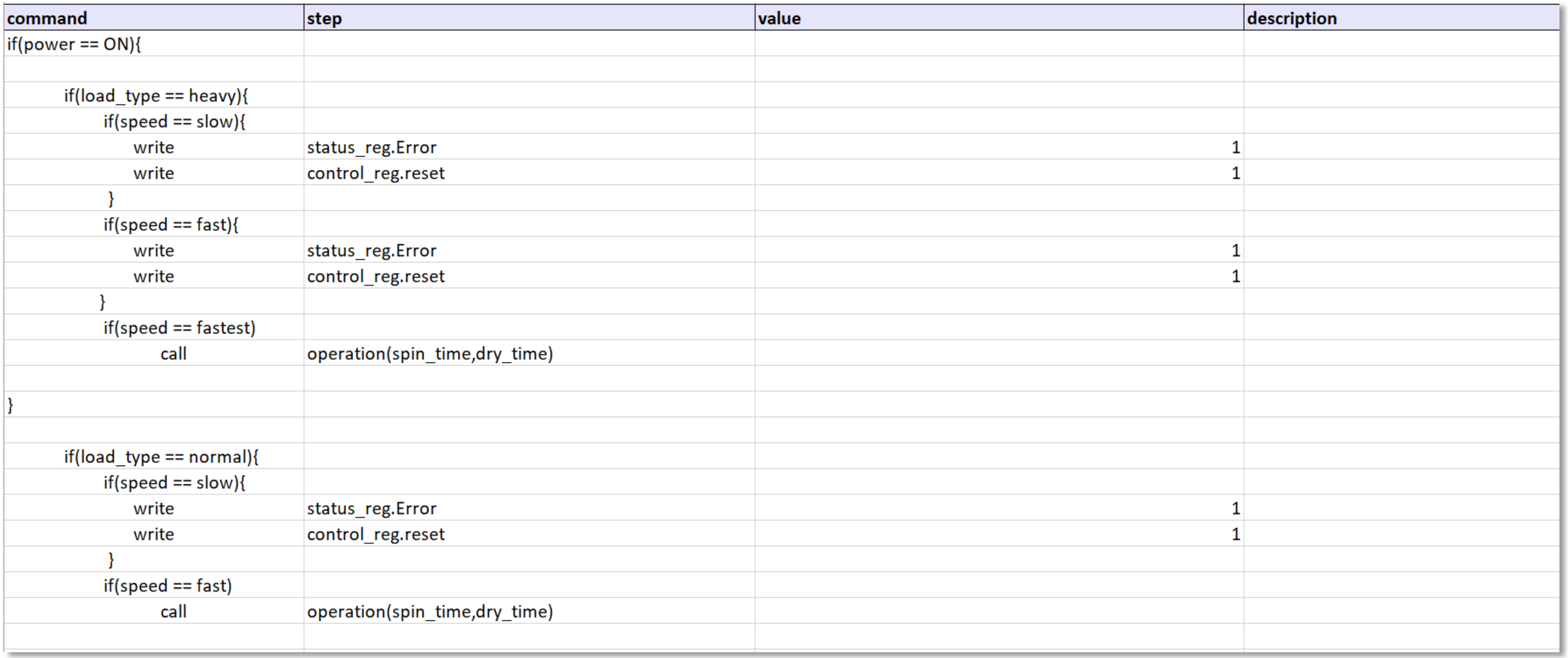
Capture Sequences in Golden Spec : Wash

Capture sequence in a pseudo-code



Sequences	This sheet describe all the sequence steps(Please don't modify the headers)	
sequence name	**ip**	**description**
wash	WashRegs	
arguments	**value**	**description**
power		
speed		
load_type		
spin_time		
dry_time		
constants	**value**	**description**
ON	1	
delicate	1	
normal	2	
heavy	3	
slow	1	
fast	2	
fastest	3	
variables	**value**	**description**

Capture Sequences in Golden Spec - Continued



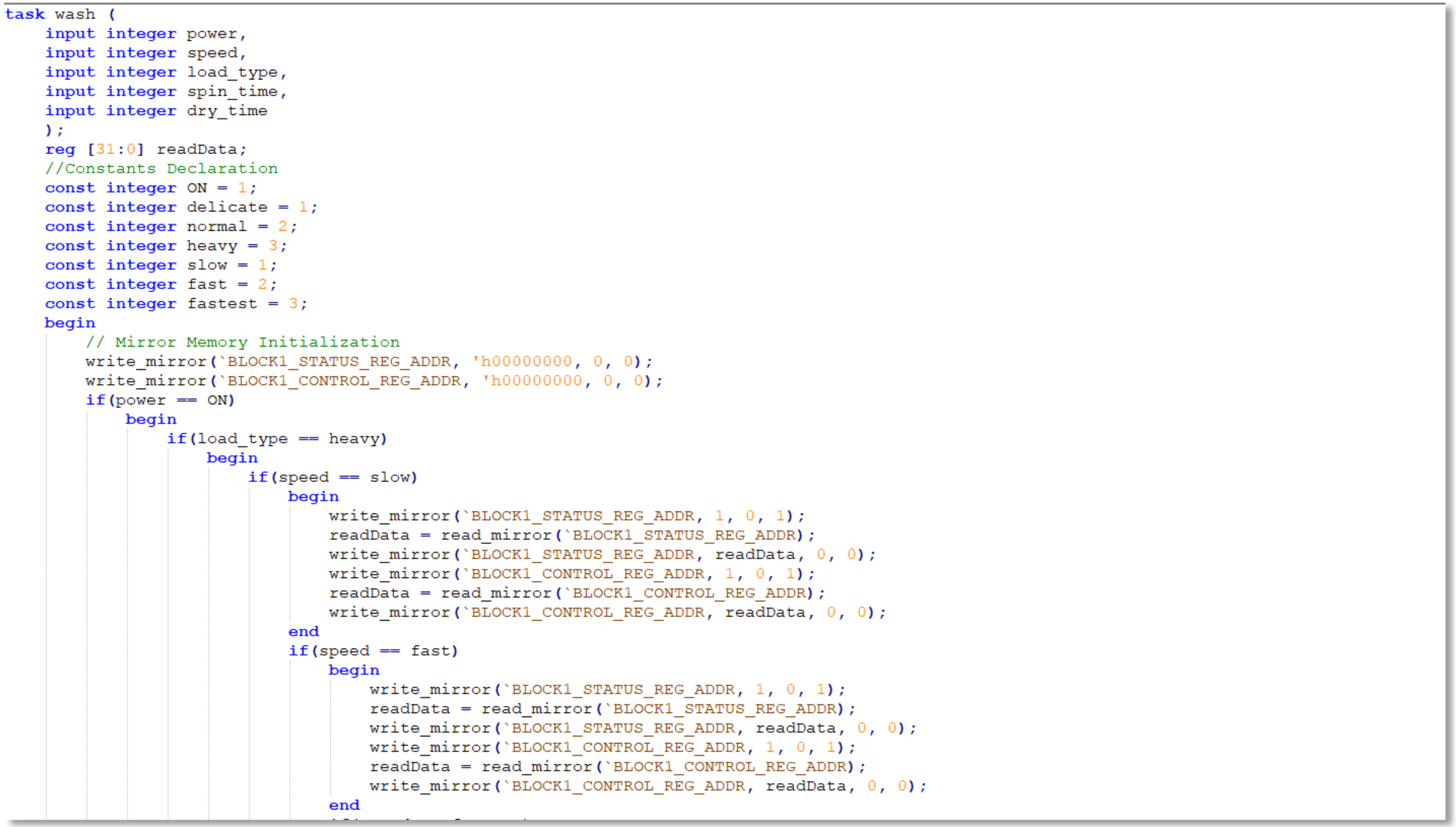
command	step	value	description
if(power == ON){			
if(load_type == heavy){			
if(speed == slow){			
write	status_reg.Error	1	
write	control_reg.reset	1	
}			
if(speed == fast){			
write	status_reg.Error	1	
write	control_reg.reset	1	
}			
if(speed == fastest)			
call	operation(spin_time,dry_time)		
}			
if(load_type == normal){			
if(speed == slow){			
write	status_reg.Error	1	
write	control_reg.reset	1	
}			
if(speed == fast)			
call	operation(spin_time,dry_time)		

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Generated Sequences in the Target Format

Generated SV Tasks for Verification

SV generated sequences

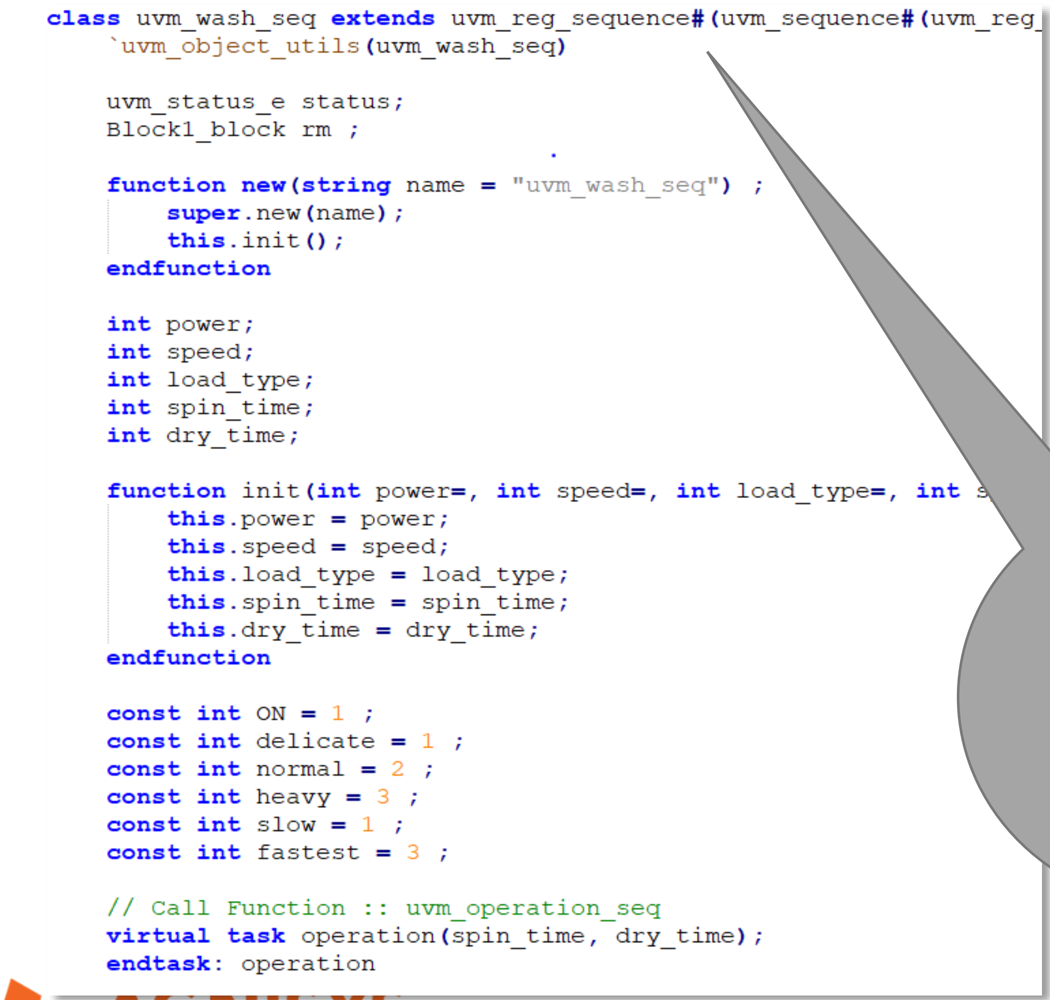


```
task wash (
    input integer power,
    input integer speed,
    input integer load_type,
    input integer spin_time,
    input integer dry_time
    );
    reg [31:0] readData;
    //Constants Declaration
    const integer ON = 1;
    const integer delicate = 1;
    const integer normal = 2;
    const integer heavy = 3;
    const integer slow = 1;
    const integer fast = 2;
    const integer fastest = 3;
    begin
        // Mirror Memory Initialization
        write_mirror(`BLOCK1_STATUS_REG_ADDR, 'h00000000, 0, 0);
        write_mirror(`BLOCK1_CONTROL_REG_ADDR, 'h00000000, 0, 0);
        if(power == ON)
            begin
                if(load_type == heavy)
                    begin
                        if(speed == slow)
                            begin
                                write_mirror(`BLOCK1_STATUS_REG_ADDR, 1, 0, 1);
                                readData = read_mirror(`BLOCK1_STATUS_REG_ADDR);
                                write_mirror(`BLOCK1_STATUS_REG_ADDR, readData, 0, 0);
                                write_mirror(`BLOCK1_CONTROL_REG_ADDR, 1, 0, 1);
                                readData = read_mirror(`BLOCK1_CONTROL_REG_ADDR);
                                write_mirror(`BLOCK1_CONTROL_REG_ADDR, readData, 0, 0);
                            end
                            if(speed == fast)
                                begin
                                    write_mirror(`BLOCK1_STATUS_REG_ADDR, 1, 0, 1);
                                    readData = read_mirror(`BLOCK1_STATUS_REG_ADDR);
                                    write_mirror(`BLOCK1_STATUS_REG_ADDR, readData, 0, 0);
                                    write_mirror(`BLOCK1_CONTROL_REG_ADDR, 1, 0, 1);
                                    readData = read_mirror(`BLOCK1_CONTROL_REG_ADDR);
                                    write_mirror(`BLOCK1_CONTROL_REG_ADDR, readData, 0, 0);
                                end
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

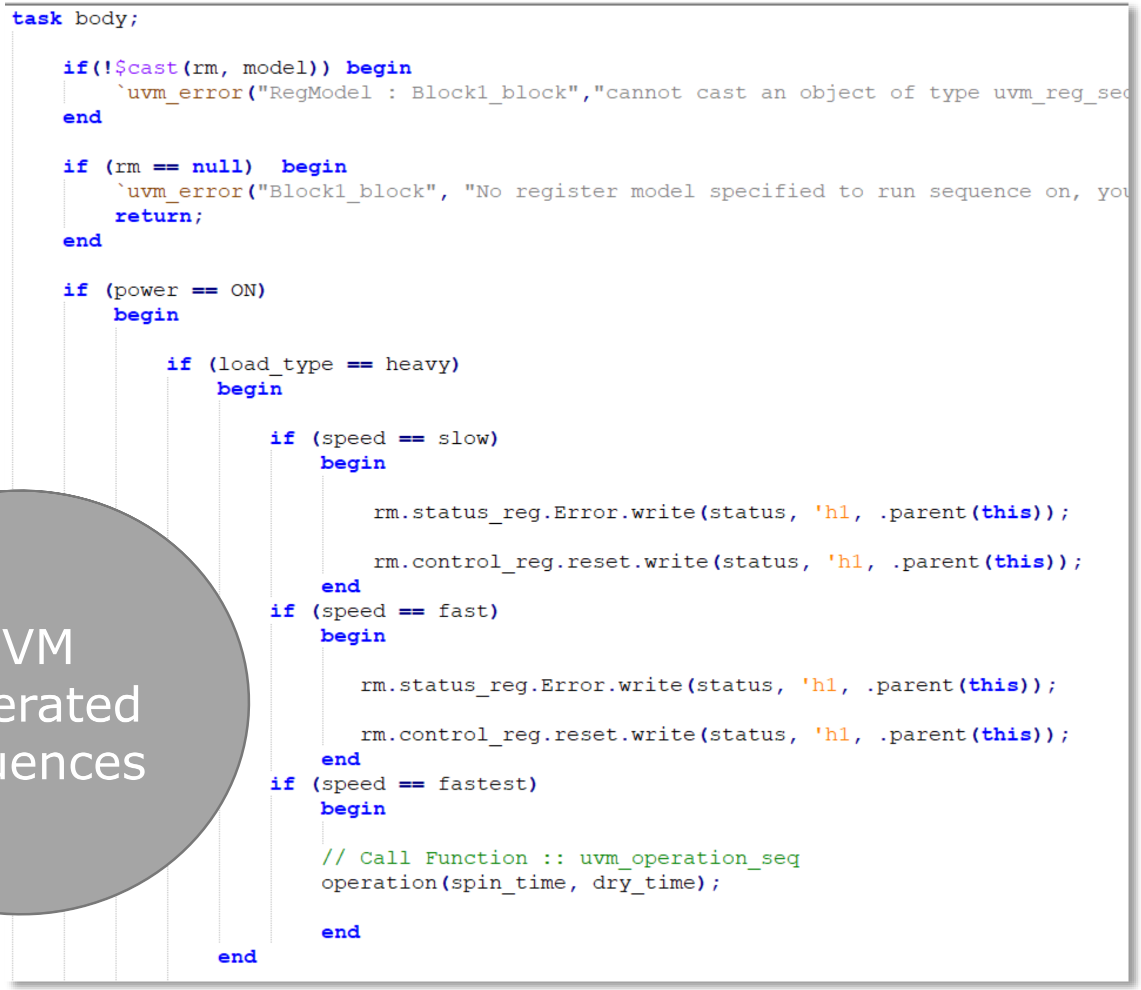
Generated Sequences in the Target Format

Generated UVM sequence output



```
class uvm_wash_seq extends uvm_reg_sequence#(uvm_sequence#(uvm_reg_
    `uvm_object_utils(uvm_wash_seq)

    uvm_status_e status;
    Block1_block rm ;

    function new(string name = "uvm_wash_seq") ;
        super.new(name);
        this.init();
    endfunction

    int power;
    int speed;
    int load_type;
    int spin_time;
    int dry_time;

    function init(int power=, int speed=, int load_type=, int s
        this.power = power;
        this.speed = speed;
        this.load_type = load_type;
        this.spin_time = spin_time;
        this.dry_time = dry_time;
    endfunction

    const int ON = 1 ;
    const int delicate = 1 ;
    const int normal = 2 ;
    const int heavy = 3 ;
    const int slow = 1 ;
    const int fastest = 3 ;

    // Call Function :: uvm_operation_seq
    virtual task operation(spin_time, dry_time);
    endtask: operation
```

UVM generated sequences



```
task body;

    if(!$cast(rm, model)) begin
        `uvm_error("RegModel : Block1_block","cannot cast an object of type uvm_reg_se
    end

    if (rm == null)  begin
        `uvm_error("Block1_block", "No register model specified to run sequence on, yo
        return;
    end

    if (power == ON)
        begin

            if (load_type == heavy)
                begin

                    if (speed == slow)
                        begin

                            rm.status_reg.Error.write(status, 'h1, .parent(this));

                            rm.control_reg.reset.write(status, 'h1, .parent(this));
                        end
                    if (speed == fast)
                        begin

                            rm.status_reg.Error.write(status, 'h1, .parent(this));

                            rm.control_reg.reset.write(status, 'h1, .parent(this));
                        end
                    if (speed == fastest)
                        begin

                        // Call Function :: uvm_operation_seq
                        operation(spin_time, dry_time);

                        end
                end
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

PSS Tool value for UVM testbenches

- Automating UVM virtual sequence logic
  - Smart quality tests to reduce the manual effort while improving the regression quality and thoroughness
  - Lightweight solution to complement and further leverage the existing UVM assets
- Systematic coverage and verification goals filling (coverage maximization)
  - Better aiming at the hard to achieve remaining coverage goals
  - Optimized solution with controlled repetitions
- Portability
  - Allow applying the same scenarios on VIP and AVIP or embedded SW
- Ease of scenario creation via either text or GUI
  - Removes the adoption barrier for non-verification engineer users that typically need to learn both UVM and specific VIP implementations
- Speed-up of constraint driven testbenches
  - Enables combining gen-time solving with run-time repetition for fast platforms like post-silicon

Conclusion

- Currently, users need to manually write long sequences that deal with the registers and pin manipulation commands
- Every scenario can not be covered by manual sequences which can have low coverage.
- This limitation is removed by PSS where all scenarios are covered which finally provide the maximum coverage.
- Also ISequenceSpec™ augments PSS tools and includes:
- Capturing sequences in a golden spec
- Generate implementation-level SV/UVM/C sequences that enable register R/W and pin manipulation commands
- PSS tools are useful for SoC high-level test scenario creation; the IP level details are currently handled using “exec blocks”

About Agnisys

- The EDA leader in solving complex design and verification problems associated with HW/SW interface
- Formed in 2007
- Privately held and Profitable
- Headquarters in Boston, MA
  - ~1000 users worldwide
  - ~50 customer companies
- Customer retention rate ~90%
- R&D centers (US and India)
- Support centers - Committed to ensure comprehensive support
  - Email : support@agnisys.com
  - Phone : 1-855-VERIFYY
  - Response time within one day; within hours in many cases
  - Time Zones (Boston MA, San Jose CA and Noida India)

IDESIGNSPEC™ (IDS) EXECUTABLE REGISTER SPECIFICATION

Automatically generate UVM models, Verilog/VHDL models, Coverage Model, Software model etc.

AUTOMATIC REGISTER VERIFICATION (ARV)

ARV-Sim™ : Create UVM Test Environment, Sequences, Verification Plans and instantly know the status of the verification project.

ARV-Formal™ : Create Formal Properties and Assertions, and Coverage Model from the specification.

ISEQUENCESPEC™ (ISS)

Create UVM sequences and Firmware routines from the specification.

DVinsight™ (DVi)

Smart Editor for SystemVerilog and UVM projects.

IDS – Next Generation™ (IDS-NG)

Comprehensive SoC/IP Spec Creation and Code Generation Tool

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY