

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



PSS Data Types

 Session 04

What You Will Learn

- ▶ Scalar Data Types in PSS
- ▶ Enums
- ▶ Strings
- ▶ User-Defined Data Types (typedef)
- ▶ Structs
- ▶ Collections (Arrays, Lists, Maps, Sets)
- ▶ Template Types

Scalar Data Types

- ▶ Scalar data types describe a single data item
- ▶ Only 2-state values supported
 - ▶ X,Z treated as 0

PSS	
<code>rand bit[8]</code>	<code>power_budget_pct;</code>
<code>rand int[11]</code>	<code>frequency_offset;</code>
<code>bool</code>	<code>low_power_mode_enabled;</code>
<code>string</code>	<code>build_banner;</code>
<code>chandle</code>	<code>dpi_bridge;</code>

Bit and bit-vector are unsigned

Int is sized and signed

true(=1) or false(=0); Defaults to false

String defaults to empty ("")

Opaque handle to a foreign language pointer
Similar to SystemVerilog

Scalar Data Types

- ▶ Scalar data types describe a single data item
- ▶ Only 2-state values supported
 - ▶ X,Z treated as 0

PSS	SystemVerilog
<pre>rand bit[8] in [0..100] power_budget_pct; rand int[11] in [-1000..1000] frequency_offset; bool low_power_mode_enabled; string build_banner; chandle dpi_bridge;</pre>	<pre>class type_showcase_c; rand bit [7:0] power_budget_pct; rand logic signed [10:0] frequency_offset; bit low_power_mode_enabled; string build_banner; chandle dpi_bridge; constraint c_ranges { frequency_offset inside {[-1000:1000]}; power_budget_pct inside {[0:100]}; build_banner inside {"AA", "BB"}; } endclass</pre>

Value Range Lists

- [0..100]
- [..10]
- [90..]

PSS gives greater visibility to “built-in” constraints

Scalar Types: Enums

- ▶ Similar to SystemVerilog enums
 - ▶ List of constant named values
- ▶ enum is a type declaration

```
enum_declaration ::= enum enum_identifier [ : data_type ] { [identifier [ = constant_expression ]  
                    { , identifier [ = constant_expression ] } ] } ;
```

```
enum soc_stage_e:bit[8] {SPEC, RTL = 4, SIGNOFF};
```

```
soc_stage_e stage;
```

Defaults to previous value+1

Can assign arbitrary *unique* value

Default value = 0

Optional type to limit possible values

Object of type `soc_stage_e`

Scalar Types: Strings

► Similar to SystemVerilog strings

```
string_type ::= string [ in [ string_literal { , string_literal } ] ] identifier [ array_dim ] [ = const_expr ]  
{ , identifier [ array_dim ] [ = const_expr ] } ;
```

```
string in ["Alpha", "Beta", "Gamma"] build_banner;
```

Default value is first element of list (or "")

Optional list to limit possible values

► Supports several built-in operators and methods

```
build_banner = "Gamma";  
build_banner.size() == 5;  
build_banner[1..3] == "amm";  
build_banner[..2] == "Gam";  
build_banner[2..] == "mma";  
build_banner.lower() == "gamma";  
build_banner.upper() == "GAMMA";  
build_banner.find("a") == 1;  
build_banner.find("a", 2) == 4;  
build_banner.find_last("a") == 4;  
build_banner.find_all("a") == {1, 4};  
build_banner.split("a") == {"G", "mm", ""};
```

of characters in the string

Substring: characters m-to-n

Substring: first n-1 characters

Substring: characters n to (size-1)

Change case

Position of first occurrence of substring

Position of first occurrence after n

Position of last occurrence

Position of all occurrences or substring

Split at sep (like Python)

Typedef

- ▶ User-defined type definition

`typedef_declaration ::= typedef data_type identifier ;`

- ▶ Allows for more descriptive type names

```
typedef bit[32] in [1..1024] block_id_t;  
block_id_t block_id;
```

Structs

- An aggregate of data items

```
struct_declaration ::= struct_kind struct_identifier [ template_param_decl_list ][: type_identifier] {  
    { struct_body_item } }
```

- Each struct_body_item must be a scalar or an aggregate of scalars

PSS	SystemVerilog
<pre>struct ip_manifest_s { bit[16] base_addr; int[8] in [1..64] instance_count; bool is_secure; string ip_name; }</pre>	<pre>typedef struct { bit [15:0] base_addr; logic signed [7:0] instance_count; bit is_secure; string ip_name; } ip_manifest_s;</pre>

Arrays

► Fixed-sized arrays of plain-data types

```
array < data_type , constant_expression > [ identifier [ = constant_expression ]  
                                           { , identifier [ = constant_expression ] } ] ;  
data_type identifier [ constant_expression ] ;
```

```
array<int, 4> timing_checkpoints;
```

Array size

Array element type

```
timing_checkpoints = { 1, 2, 3, 4 };  
timing_checkpoints[0] = 10;  
timing_checkpoints.size() == 4;  
timing_checkpoints.sum() == 19;  
low_power_mode_enabled = (3 in timing_checkpoints);
```

Array initialization

Array literal expression

Array index operator: []

Built-in methods

Set membership operator
(returns bool)

::: Lists

► Variable-sized ordered list of elements

```
list < data_type > [ identifier [ = constant_expression ]  
                  { , identifier [ = constant_expression ] } ] ;
```

► Similar to a SystemVerilog queue

```
list<roles_e> engineer_roles; // SystemVerilog: roles_e engineer_roles[$];
```

List element type

```
engineer_roles = {Design, Verification, Synthesis, Validation};  
engineer_roles.push_front(Architect);  
engineer_roles.push_back(Software);  
engineer_roles.insert(2, PhysicalDesign);  
engineer_roles[0] = SoCArchitect;  
blame = engineer_roles.delete(3);  
blame = engineer_roles.pop_back();  
numroles = engineer_roles.size(); // == 5
```

0:Design, 1:Verification, 2:Synthesis, 3:Validation
0:Architect, 1:Design, 2:Verification, 3:Synthesis, 4:Validation
0:Architect, 1:Design, 2:Verification, 3:Synthesis, 4:Validation, 5:Software
0:Architect, 1:Design, 2:PhysicalDesign, 3:Verification, 4:Synthesis, 5:Validation, 6:Software
0:SoCArchitect, 1:Design, 2:PhysicalDesign, 3:Verification, 4:Synthesis, 5:Validation, 6:Software
0:SoCArchitect, 1:Design, 2:PhysicalDesign, 3:Synthesis, 4:Validation, 5:Software
0:SoCArchitect, 1:Design, 2:PhysicalDesign, 3:Synthesis, 4:Validation

blame==Synthesis

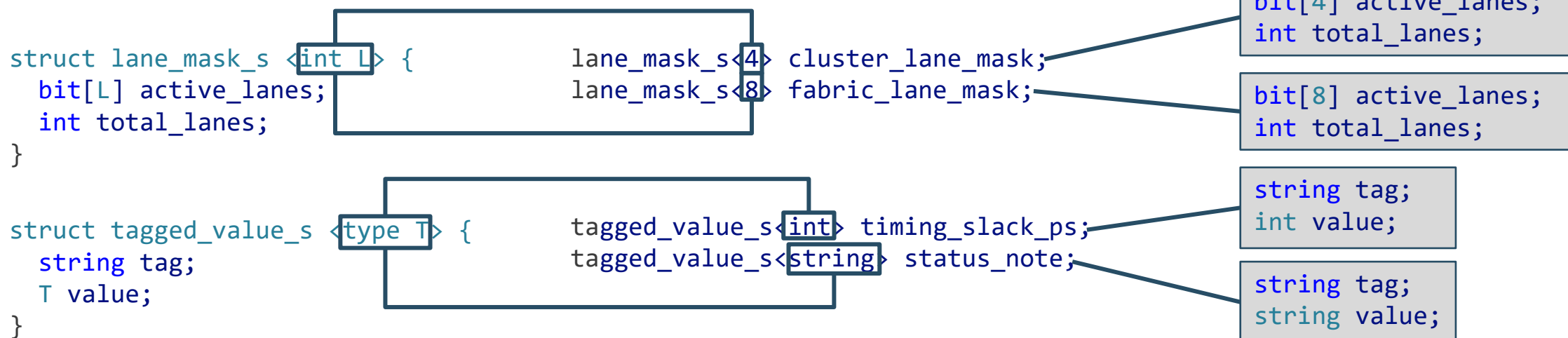
blame==Software

Template Types

- ▶ Define a generic parameterized type
- ▶ Template parameters can be value or type

```
value_param_decl ::= < data_type [ identifier [ = constant_expression ] >  
                  { , identifier [ = constant_expression ] } ] ;
```

```
type_param_decl ::= < type [ identifier [ = type_identifier ] >  
                  { , identifier [ = type_identifier ] } ] ;
```





Thank You

