PDVL v0.1
Programming, Design and Verification Language

Tobias Strauch
Munich, Bavaria
tobias at cloudx dot cc

PDVL is an aspect-oriented, translation-level Programming, Design and Verification Language (PDVL). This specification version 0.1 covers the Design (D) section of PDVL.

PDVL was first introduced at the IEEE Euromicro DSD 2017 conference [1].

If you want to reference it, please use the following BibTeX template:

```
@INPROCEEDINGS{8049764,
author={Strauch, Tobias},
booktitle={2017 Euromicro Conference on Digital System Design (DSD)},
title={An Aspect and Transaction Oriented Programming, Design and Verification Language (PDVL)},
year={2017},
volume={},
number={},
pages={30-39},
doi={10.1109/DSD.2017.66}}
```

Note: Examples in this document are given in red.

References

[1] T. Strauch, ”An Aspect and Transaction Oriented Programming, Design and Verification Language”, IEEE Euromicro DSD 2017, 30 Aug. - 1 Sep., Vienna, Austria, pp. 30 - 39

Contents

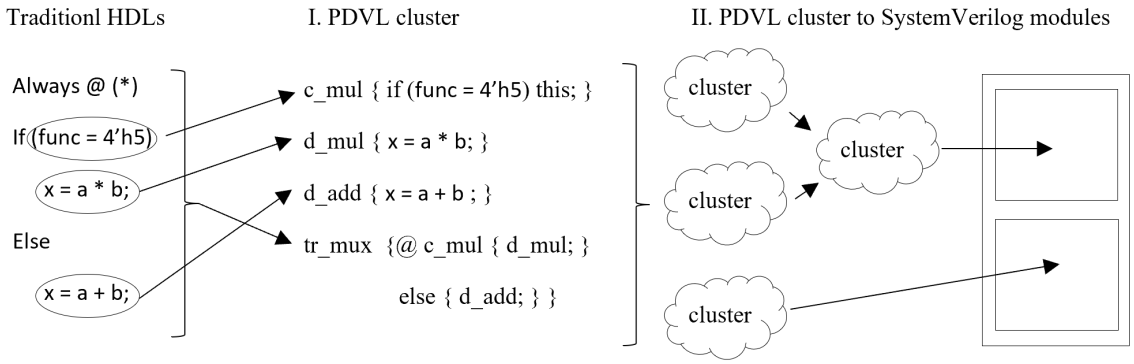

Figure 1: Traditional HDL processes, PDVL clusters and transformation process to System Verilog modules.

1. Introduction to basic building blocks (informative)

1.1. Top-level perspective

A very brief overview of PDVL is shown in figure 1. Traditional HDLs typically use always (or process) blocks to define logic. In PDVL we encapsulate conditions and assignments and define them individually as conditions and datapaths.

The term transaction has different meanings in the HDLs. In PDVL, a transaction basically defines the remainder of an always block that has not been defined by conditions and data paths. Transactions define under what conditions individual datapath assignments are valid.

PDVL combines signal definitions, conditions, datapaths and transactions to clusters “Figure 1, I. PDVL cluster”.

An essential step in the aspect-oriented PDVL concept is that clusters can be merged. When converting a PDVL design to a SystemVerilog design, the clusters can be added to a previously generated module hierarchy “Figure 1, II. PDVL cluster to SystemVerilog modules”.

1.2. Items, registers, datapaths, conditions, and transactions

We outline PDVL by using individual examples. The basic elements of PDVL are items, datapaths, conditions, events and transactions. These elements are grouped to clusters, which we will show later in this section. Assuming we want to assign the value of the item ”u” to the item ”y”. In PDVL, this would be defined as:

```
item u;
item y;
data andName { y = u; } // alternatively: d_anyName  y = u;
```

Items (keyword ”item”) can be of any type (vector, record, ...). A data construct (keyword ”data” or prefix ”d_”) describes, how items are modified. They are valid or unused at a given time and represent individual datapath assignments. A PDVL to Verilog compiler should use the most simplistic implementation and a simple assignment will be generated:

Verilog:

```
wire u;
wire y = u;
```

We now define that the assignment ”y = u;” is only valid, when its encapsulating data construct (here: ”d_yu”) is active:

```
item u, y;
d_yu { y = u; }
```

Now let’s assume that we want to assign ”u” to ”y” by default and - at a given point of time - assign ”v” to ”y”. The following code will give a compile error:

```
item u, v, y;
d_yu { y = u; }
d_yv { y = v; }
```

The compiler will complain that we have two assignments to the same element (here ”y”) at the same time that are not exclusive. Items can be assigned by different data constructs, but only one assignment to an item (or item bit) is allowed at a time. That is why we introduce conditions and transactions in PDVL. We expand the given example to:

```
item u, v, y;
d_yu { y = u; }
d_yv { y = v; }
cond sel_u;
trans mux { @sel_u d_yu;
            else d_yv; }
```

A transaction (Tr, keyword ”trans” or prefix ”t_”) defines which data construct is valid at a given point in time. This is done by evaluating the status of the given condition (keyword ”cond” or prefix ”c_”). The compiler generates the following multiplexer in Verilog for the PDVL code given above:

Verilog:

```
wire u, v, sel_u;
wire y = sel_u ? u : v;
```

As previously mentioned, a Tr defines which data construct is valid at a given point in time. So far we’ve only used asynchronous Tr, which produce strictly combinatorial logic. We introduce events and use a counter as an example of a sequential Tr:

```
reg [7:0] counterValue;
d_count { counterValue = counterValue + 1; }
d_reset { counterValue = 0; }
```

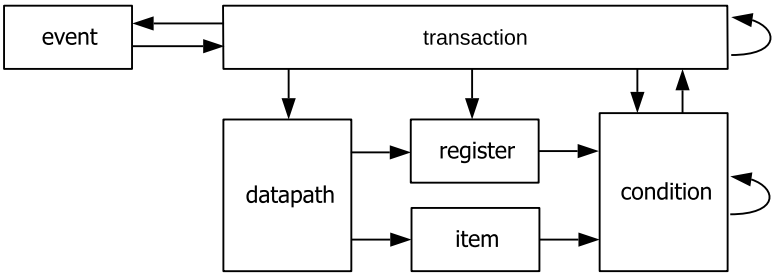

Figure 2: Transactions, datapath, items, conditions and events in PDVL.

```
c_overflow { if (counterValue == 8'hff) this; }
event clkr posedge clk;
tr_count {
   @clkr {
      d_count;
      @c_overflow d_reset; } }
```

A reg construct defines a sequential element (register, latch, etc.). We also use the event called "clkr". An attribute (e.g. "posedge clk") must be added to an event declaration body to precisely define a particular clock or latch enable function.

The Tr "tr_count" is a sequential Tr because it contains the event named "clkr" with an attribute "posedge clk". All elements declared as "reg" with a data path assignment within the body of the "clkr" event are sequential elements.

Verilog:

```
reg [7:0] counterValue;
always @ (posedge clk) begin
   counterValue <= counterValue + 1;
   if (counterValue == 8'hff) counterValue <= 0;
end
```

Figure 2 shows how Tr control the data and condition flow. Tr define, which data construct is active. An active data constructs assigns a relevant logic constructs to a reg or an item. Conditions can be set valid based on conditional checks of registers or items, by other conditions or directly by Tr. Tr can set the values of state registers of a FSM directly. Events can trigger Tr and can be emitted by Tr as it is used in clock generators for instance.

A Tr can also include other Tr.

```
tr_a {
   tr_b;
   tr_c; }
```

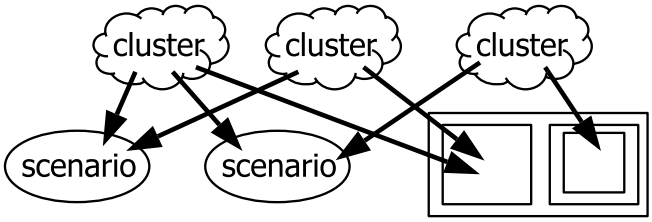

Figure 3: Building scenarios and synthesizable, hierarchical designs.

What has been said so far can be summarized as follows:

> **A Transaction defines when data path assignments are valid.**
> **To do this, it uses events and conditions.**

In PDVL, finite state machines (FSM) can be generated by using Tr. The code below shows an example FSM with two states (”wait” and ”wait_ack”). The state register name must be defined by the user. The entries of the state enumeration list are derived automatically.

```
tr_handshake_transmit {
   finite handshake {
      wait:  @c_tr { c_tr_hs; wait_ack; }
      wait_ack:  @c_ack { c_tr_done; wait; }
               else c_tr_hs; } }
```

More on FSM in the next section.

1.3. Clusters, joining and generating synthesizable, hierarchical designs

The basic elements of PDVL are items, datapaths, events and transactions. These elements are grouped to clusters (optional prefix ”c_”) or subclusters. Subclusters are clusters within a cluster.

```
cluster someDesign {
   items ...
   data ...
   event ...
   transactions ...
   cluster master {
      items ...
      data ...
      event ...
      transactions ... }
   cluster slave { ... } }
```

Once clusters are defined, commands will then generate scenarios and synthesizable, hierarchical designs. This is shown in Figure 3.

For synthesizable designs, a hierarchy is generated in a first step. Then clusters are placed in the individual design modules. The logic is joined within the modules. A PDVL compiler then does the signal routing between the modules.

```
place someDesign.master          // (sub-)cluster name
   sub.peripheral_0;             // hierarchical path
```

Clusters can also be merged to new clusters, before the temporarily generated clusters are placed in modules or used otherwise.

As an alternative concept to a predefined module hierarchy, PDVL allows the joining of Tr to individual scenarios. Scenarios can combine Tr of multiple abstraction levels and timing accuracies. They are usually tailored towards a specific need, such as modeling and verification. This is indicated in Figure 3.

2. PDVL design specification (D)

2.1. PDVL code framing

Within a document, PDVL code must be framed using '<"' and '">' symbols, whereas multiple sets are possible:

... documentation ...
<" ... PDVL code ... ">
... documentation ...
<" ... PDVL code ... ">
... documentation ...

The framing constructs (<" and ">) are omitted for examples in this specification.

2.2. PDVL declarations

2.2.1. Clusters

Any PDVL code is organized in clusters. Subclusters are embedded in clusters or other subclusters. Cluster or subclusters can be defined using the keyword cluster.

```
cluster a  [cluster_body] ;
```

PDVL allows the usage of a predefined prefix to define the construct as a cluster definition. The following statement is identical to the above:

```
cl_a  [cluster_body] ;
```

except, that the prefix remains part of the cluster identifier name (here "cl_a").

A cluster's body can list PDVL declarations and PDVL commands, whereas subclusters only support PDVL declarations.

```
cluster a {
... PDVL declarations ... ... PDVL commands ... }
cl_b {
... PDVL declarations ...
... PDVL commands ...
cluster c {
... PDVL declarations ... }
cl_d {
... PDVL declarations ... } }
```

We distinguish between cluster which only contain declarations and commands clusters. A clusters body can list PDVL declarations and PDVL commands, whereas subclusters only support PDVL declarations.

2.2.2. Parameters

Parameters can be defined using the following construct:

```
parameter PC_LEN = 32;
```

Parameter passing will be explained in the next section.

2.2.3. Items

Items are used for signals which result from assignments or combinatorial logic. They are non-sequential elements.

Items can be overwritten by latch or register definitions.

```
item someSignal;
```

Also, a list of item definitions is possible:

```
item someSignal, someOtherSignal;
```

Items can be directly assigned during definition:

```
item someSignal = someOtherSignal;
```

An item definition can be overwritten by a latch or register definition if the item does not have a direct assignment and the types of the item and the latch/register are identical.

Items also can have an attribute defined. Let's say the following code is joined in one module after the join-logic step:

```
(* AXADDR=DRIVER *) item write_add [31:0];
(* AXADDR *) item axi_mst_wr_addr [31:0];
```

If two items in the same module have the same attribute after joining, they are connected, whereas the attribute with the "driver" assignment is consider as the driving item. The keyword driver is non-case sensitive. The above example results in the following System Verilog code:

```
assign axi_mst_wr_addr = write_add;
```

The signal axi_mst_wr_addr can then be used for routing, for instance.

2.2.4. Latches

Elements defined as latches in PDVL will be defined in the SystemVerilog output as latches using the SystemVerilog “always_latch” construct.

```
latch someLatch;
```

Also, a list of latch definition is possible:

```
latch someLatch, someOtherLatch;
```

The chapter “Latch enable, register clocking, and reset” defines further latch related aspects.

Latches can have an attribute defined.

```
(* someAttribute=DRIVER *) latch someLatch [31:0];
```

They are only relevant when defined as driver. If not defined as driver, an error is given: ERR.CONNECT.NONDRIVER.

2.2.5. Registers

Elements defined as registers in PDVL will be defined in the SystemVerilog output as register using the SystemVerilog “always_ff” construct.

```
reg someReg;
```

Also, a list of register definition is possible:

```
latch someReg, someOtherReg;
```

Registers can have an attribute defined.

```
(* someAttribute=DRIVER *) latch someReg [31:0];
```

They are only relevant when defined as driver. If not defined as driver, an error is given:

ERR.CONNECT.NONDRIVER.

The chapter “Latch enable, register clocking, and reset” defines further register related aspects.

2.2.6. Types

PDVL supports SystemVerilog like single- and multidimensional vector types, enumeration types as well struct and unions. Types can be defined using the typedef keyword:

```
typedef logic [2:0][7:0] type_arr;
typedef enum logic [2:0] {WAIT, LOAD, DONE} type_enum;

typedef struct {
   logic [7:0] data;
   logic [7:0] addr;
} type_struct;
typedef union packed {
   struct packed {
      logic [7:0] data;
      logic [7:0] address;
   } data_packet;
   struct packed {
      logic [7:0] data;
      logic [7:0] instr;
   } instr_packet;
} type_union;
```

Predefined types can be used when items, latches or registers are defined. The keyword “item”, “latch” or “reg” is placed at the beginning of the definition:

```
item type_arr vecarr_reg;
latch type_enum enum_reg;
reg type_struct struct_reg;
reg type_union type_union_reg;
```

Single- or multidimentional types as well as enumeration types can be defined “inline”:

```
reg enum [2:0] {WAIT, LOAD, DONE} enum_reg;
reg [7:0] vecarr_reg [3:0][2];
```

In SystemVerilog, the vectors before the register (or item/latch) list define the packed dimension, and the vectors following the register list define the unpacked dimension. SystenVerilog allows C-style vector definition for unpacked dimensions. In the above example, the C-style array

dimension [2] is identical to [1:0]. PDVL allows C-style vector definition for packed dimensions as well.

reg [PC_LEN] pc;

The C-style vector definition by can also be used during item definition and assignments.

item [7] funct7i = instr[31:25];

In the current version of PDVL, items (or latches/registers) with an enumeration type cannot be routed. In case of an attempt, the following error is thrown:

ERR.AUTOROUTE.ENUMARATION_TYPES_NOT_SUPPORTED

2.2.7. Datapaths

3.2.7.1 Datapath definition

Datapaths can be defined using the keyword datapath.

datapath a { [datapath_body] };

PDVL allows the usage of a predefined prefix to define the construct as a datapath definition. The following statement is identical to the above:

d_a { [datapath_body] };

except, that the prefix remains part of the cluster datapath name (here “d_a”).

A datapath lists a set of signal assignments:

d_addi dp_out = rs1_dato + instr[31:20];

Assignments to any item, latch or register is valid.

Only blocking assignments are allowed.
Transactions define, when datapath assignments are valid.
Datapaths do not support attribute definitions.

3.2.7.2 Datapath port list

A datapath can have a port list:

```
d_a ([1:0] b, [1:0] c, [1:0] d) { a = b + c + d; };
```

In the example above, the datapath can be set valid within a transaction using a calling port list:

```
d_a(e + 2, f, g);
```

The available port types are identical to the ones of a SystemVerilog function. They are kept there for completeness. The mapping process can be based on regular expression replacement. The resulting assignment in the above example would be:

```
a = e + 2 + f + g;
```

3.2.7.3 Datapath and signal ordering

When multiple assignments to the same signal are done, then the order remains unchanged throughout the conversion process from PDVL to SystemVerilog.

3.2.7.4 Datapath definition auto-generate transaction

Datapaths can also be defined using a predefined prefix to define the construct as a datapath definition associated with a transaction call (transactions will be declared later). The following statement:

```
tr_d_a { [datapath_body] };
```

is split into the following two statements:

```
d_a { [datapath_body] };
tr_d_a { d_a; }
```

2.2.8. Conditions

3.2.8.1 Overview

Conditions can be defined in 3 different ways:

1. Conditions can be set true or false using transaction trees. The must also be declared. If the declaration contains the keyword “reg” and the transaction has the proper clock event definition, then the condition works as a register. Otherwise, the condition is asynchronous. The condition is true or falls depending on the logic cone resulting defined in the transaction tree.

2. Conditions can be defined using a definition body. In this case, they are asynchronous and always check if equations based on items, latches or registers result in true or falls.

3. A condition can be defined as a function using a port list.

Conditions can be used in 3 different ways:

1. Conditions can be used in a transaction tree using the keyword @.

2. Conditions defined as a function can be used in a transaction tree using the parameters.

3. In the current version of PDVL, it is not prohibited to use conditions as part of an equation. Nevertheless, using conditions in equations will limit verification capabilities.

3.2.8.2 Condition emitted in a transaction tree

A condition can be set true or false using transaction trees. The condition still needs to be defined:

```
cond a;
```

PDVL allows the usage of a predefined prefix to define the construct as a condition definition.

The following statement is identical to the above:

```
c_a;
```

except, that the prefix remains part of the condition name (here “c_a”).

To set a condition to valid is by using transactions:

```
tr_a { @c_b { c_a; } };
```

In this trivial example above, the condition c_a is valid whenever c_b is valid.

When the condition should be implemented as a register, the following declaration is valid:

```
c_a reg;
```

3.2.8.3 Condition with a condition body

A condition can be defined using a definition body.

```
cond a { [condition_body] };
```

The following statement is identical to the above:

```
c_a { [condition_body] };
```

The condition body defines, when a condition is valid, indicated by the keyword “this”:

```
c_instr_i_lui { if (opcode_i == 7'b0110111) this;}
```

A condition is invalid by default. It continuously and independently checks the defined equations. If any of the defined equations is valid, then the condition becomes valid as well.

Conditions with a condition body are always non-sequential.

3.2.8.4 Condition port list

A condition can have a port list:

```
c_instr_i_1_0 ([2:0] x) { if (instr_i[1:0] == x) this; };
```

In the example above, the condition can be checked within a transaction using a calling port list:

```
@c_instr_i_1_0(3'b010)
```

The available port types are identical to the ones of a SystemVerilog function.

A condition with a port list can be converted to a function. It can therefore not be used as a routing signal.

A condition with a port list cannot have attributes.

3.2.8.5 Conditions with defined level and signal name

Conditions can have an active level and a signal name defined. Only a pairwise definition is allowed.

```
c_rstn low rstn;
c_rst high rst { if (. . . ) this; };
```

In this case, no port list is allowed.

3.2.8.6 Condition attributes

Conditions can have an attribute defined:

(* AXVALID *) c_axi_mst_valid;
(* AXVALID = driver *) c_valid { if (. . .) this; };

Attribute definitions are not allowed for conditions, which have an active level (and a signal name) defined.

not allowed: (* FIFOWR *) c_store high fifo_write;

Conditions can only be connected to other conditions, but not to items, latches, and registers, or vice-versa.

When attributes are used, no port list is allowed.

2.2.9. Events

An event is a signal with a positive or negative edge attribute. The event needs to be defined:

event e_clk posedge clk;

PDVL allows the usage of a predefined prefix to define the construct as an event definition.

The following statement is identical to the above:

e_clk posedge clk;

except, that the prefix remains part of the event name (here “e_clk”).

Events do not support additional attribute definitions.

Events are used by transactions, as defined in the next section.

2.2.10. Transactions

3.2.10.1 Transaction definition

Transactions can be defined using the keyword transaction.

transaction a { [transaction_body] };

PDVL allows the usage of a predefined prefix to define the construct as a transaction defini-

tion. The following statement is identical to the above:

```
tr_a { [transaction_body] };
```

except, that the prefix remains part of the cluster transaction name (here “tr_a”).

Transactions do not support attribute definitions.

3.2.10.2 Transactions generate combinatorial or sequential logic

A transaction defines signal cones for registers and combinational signals. This is an example of a transaction which defines a combinatorial logic cone:

```
tr_decode {
   unique propagate {
      @c_instr_dec_op {
      @c_funct3_0 {
         @c_funct7_0 { c_instr_i_add; }
         @c_funct7_64 { c_instr_i_sub; } } } }
```

Transactions also define registers:

```
tr_register { @c_sys_rst { d_pc_reset; }
              else { @e_clk { d_pc_assign; } } }
```

A transaction can define signal cones for registers and combinational signals at the same time. The reset and edge part of the transaction are only relevant for registers. The following Table gives an overview of how the conversion process of transactions handles items and registers, using the following code:

```
c_rst low rstn;
e_clk posedge clk;
tr_s { @c_rst { d_rst; }
       else { @e_clk { d_cone; } } }
```

3.2.10.3 Transaction calling other transactions

Transactions can also be called by other transactions:

```
tr_register { @c_sys_rst { d_pc_reset; }
              else { @e_clk { tr_rv32i; tr_rv32m; } } }
```

Table 1: PDVL code and SystemVerilog output

PDVL	**SystemVerilog**
reg s_reg_rst; d_rst { s_item_rst = 1; } d_cone { s_reg_rst = s_reg_rst_next; }	always_ff @ (posedge clk or negedge rstn) begin | if (!rstn) begin | | s_reg_rst <= 0; | end | else begin | | s_reg_rst <= s_reg_rst_next; | end end
reg s_reg d_rst { } d_cone { y_reg = s_reg_next; }	always_ff @ (posedge clk) begin | if (!(!rstn)) begin | | s_reg <= s_reg_next; | end end
item s_item_rst; d_rst { s_item_rst = 1; } d_cone { s_item_rst = s_item_rst_logic; }	always_comb begin | if (!rstn) begin | | s_item_rst = 1; | end | else begin | | s_item_rst = s_item_rst_logic; | end end
item s_item; d_cone { s_item = s_item_logic; }	assign s_item = s_item_logic;

When the transaction body contains a list of transactions calls, then these transaction calls are replaced by the individual transaction bodies of the called transactions. This is just syntactical replacement. It is important to notice, that the order of the called transactions bodies remains intact.

2.2.11. Finite state machines (FSM)

Transactions also support specific finite state machine (FSM) constructs. If the transaction body starts with the keyword "finite" and a name-string, then FSM logic is generated. The construct body must then have a list of state-body-pairs, like a case statement:

```
tr_fsm { finite prot {
   wait : { d_sig2_set;
            start; }
   start : { d_sig2_clear;
            wait; } } }

tr_clocking { @e_clk { tr_fsm; } } ;
```

Transactions which generate FSMs must have register framing, which means they must have an event condition in the merged transaction tree. In the example above the name "prot" defines the name of the FSM registers.

Multiple transactions with the finite keyword and the same name-string are merged. When the keyword "one_hot" follows the finite keyword in at least one of the merged transaction bodies, then the register decoding is done in a one-hot encoding fashion:

```
tr_fsm_e { finite one_hot prot {
   start : { d_sig2_clear;
            end; }
   end : { d_sig2_clear;
            wait; } } }

tr_clocking { @c_arst { d_reset; }
                 else { @e_clk { tr_fsm; tr_fsm_e; } } } ;
```

When transactions with the finite keyword are merged, then the state bodies are merged according to their state names. The order of the logic within the state bodies is identical to the order when the transactions are called. In the example above, the logic of a state in the tr_fsm is listed before the logic of the same state of the tr_fsm_e transaction.

This mechanism also automatically defines new parameters and registers. Therefore, the FSM state register and the associated states don't need to be defined separately. They are also accessible by other PDVL constructs. The states are stored in a register with the FSM's name. The FSM states themselves will be based on the FSM's name, the keyword "state" and the individual state's name,

all separated by an underscore and everything in upper letters.

Therefore, in order to reset the FSM with the FSM state register prot can be set to a certain state (here PROT_STATE_WAIT) by using the following datapath:

```
d_reset { prot = PROT_STATE_WAIT; }
```

2.2.12. Interleaved pipelining of transactions

Transactions which generate sequential logic can also be pipelined. There are three ways to do this. Either a constant number or a range of pipeline stated is defined, or the interleaved stages are defined by name.

3.2.12.1 Pipelining defined by a constant number

A transaction with the body entry “pipe”, followed by a constant number and a body in curled brackets defines, that the logic cones for the sequential elements defined in the context, where this transaction is used, should be auto-pipelined. The constant number defined the number of stages.

```
tr_pipe { pipe 4 { . . .  } }
```

3.2.12.2 Pipelining defied by a range

The number of pipeline stages can also vary. The next example shows how a range of pipeline stages can be defined:

```
tr_pipe { pipe [MAX_STAGE : MIN_STAGE] { . . .  } }
```

3.2.12.3 Defining interleaved stages by name

The pipeline stages can also be defined by name, whereas all interleaved defined pipeline stages with the same name are merged. The list of all possible pipeline stages must be defined at least once completely. Not all interleaved pipelining definitions must have all stages listed.

```
tr_add { inter RV {
   fetch: { . . .  } // defining only once possible
   decode: { . . .  }
   execute: { . . .  } } }
```

2.2.13. Register context switching

Transactions also support context switching. This is only possible for sequential signals with one-dimensional type (bit, bus). An additional dimension is automatically added.

The transaction body must have the following entries. The keyword context, a constant number, which defines the context switching depth, the write address signal, and the read address signal followed by a new body definition in curly brackets. Here is an example:

```
tr_context { context 16 addIn addOut { ... } }
```

2.2.14. Functions

PDVL supports SystemVerilog functions in cluster definitions. The ANSI-C style can be used:

```
function [7 : 0] f_xyz ([7 : 0] x);
  f_xyz = x;
endfunction
```

Also functions using declarations and directions are supported:

```
function [7 : 0] f_b2i;
input [7 : 0] f_x;
  f_b2i = f_x;
endfunction
```

Assuming a function is used but not defined in a cluster. Once the cluster is placed in a module, the closed definition of the relevant function is copied over. The same method is implemented for parameters.

2.2.15. Macros

PDVL supports macro(s) to simplify code.

There is one macro defined so far. It is used to define registers (or latches) and its associated datapaths and transactions automatically.

Assuming the following code for a register is given:

```
reg m0_reg @c_rst { 0; } @e_clk { m0_reg_next; };
```

The register “m0_reg” is 1’b0 when the “c_rst” condition is true, otherwise the signal “m0_reg_next”

is assigned when the “c_clk” condition is true. Then the following internal code is automatically derived:

```
reg m_reg;
d_m0_reg_c_rst { m0_reg = 1’b0; }
d_m0_reg_e_clk { m0_reg = m0_reg_next; }
tr_m0_reg { @c_rst { d_m0_reg_c_rst; }
            else { @e_clk { d_m0_reg_e_clk; } } }
```

Alternatively, the macro also allows the usage of full assignments. Assuming the following code for a register is given:

```
reg [1:0] m1_reg @c_rst { m1_reg = 2’b00; }
                 @e_clk { m1_reg[0] = m1_reg_next[0]; m1_reg[1] = m1_reg_next[1]; };
```

Then the following internal code is automatically derived:

```
reg m1_reg;
d_m1_reg_c_rst { m1_reg = 2’b00 ; }
d_m1_reg_e_clk { m1_reg [ 0 ] = m1_reg_next [ 0 ] ;
                 m1_reg [ 1 ] = m1_reg_next [ 1 ] ; }
tr_m1_reg { @c_rst { d_m1_reg_c_rst; }
            else {@e_clk { d_m1_reg_e_clk; } } }
```

The macro also works for latches:

```
latch m_latch @c_en { m_latch_next; };
```

The resulting internal code is:

```
latch m_latch;
d_m_latch_c_en { m_latch = m_latch_next; }
tr_m_latch { @c_en { d_m_latch_c_en; } }
```

2.3. Build commands

Commands are used to generate a module hierarchy and to join clusters or logic inside the modules. These commands are listed within a build construct body.

The build command simply generates an empty module, such as SoC, FPGA_TOP or testbench for example. The build command also defines the current reference module. All relevant commands following a build command refer to this reference module.

The following example generates a module with the name SoC:

```
build SoC { [build_commands_list] }
```

The commands do not need to be in a particular order. If a command depends on the execution of another command, its execution is delayed respectively.

2.3.1. Uniquify command

The uniquify command copies a specified cluster and adds a given prefix to its name and all its members.

In the following example, the cluster “axi” is copied to a cluster now accessible as “ic0_axi”. The new cluster members then also have the prefix “ic0_”.

```
uniquify axi ic0_;
```

2.3.2. Place command

The place command generates an instantiation. The following example generates an instantiation “i_rv” of the module RV32_IMC.

```
place RV32_IMC i_rv;
```

It is possible to use a hierarchical path. This example generates the instantiation “i_rf” of the module RF inside the RV32_IMC module of the example above.

```
place RF i_rv.i_rf;
```

2.3.3. Join cluster command

The join command followed by a cluster identifier joins the cluster in the module, where the join command is executed:

```
build RV {
join cl_rv32imc; }
```

A cluster can also be joined in other target clusters:

```
join cl_rv32i cl_rv32imc;
```

or submodules:

```
join cl_rv32i i_rv;
```

It is possible to use a hierarchical path for source and target clusters as well as module hierarchies. In the following example, the cluster “cl_rv32imc” is joined in the module with the hierarchical path “i_sys.i_rv” relative to the module “SoC”.

```
build SoC { join cl_rv32imc i_sys.i_rv; }
```

When subclusters of one individual cluster are joined in different modules, it might be important to keep track of the aspect, that the subclusters are related to each other. This is particularly of interest when tristate signals need to be routed (see Tristate signal routing chapter). For this case, a join command with subcommands is introduced.

```
join tri { a i_0; b i_1; }
```

The above example joins the subclusters tri.a (tri.b) in the submodule i_0 (i_1).

2.3.4. Join logic command

The join command (directly) followed by a cluster body joins the cluster in the module, where the join command is executed:

```
build RV {
  join { tr_register { @c_sys_rst { d_pc_reset; }
                       else { @e_clk { tr_rv32i; tr_rv32m; } } } } }
```

A cluster body can also be joined in other target clusters:

```
join { tr_register { [transaction_body] } } cl_rv32imc;
```

or submodules:

join { [cluster_body] } i_rv;

It is possible to use a hierarchical path for target clusters as well as module hierarchies.

2.3.5. Remove command

Once all join commands are executed, transactions can be removed again from modules. The remove command allows the user to delete transactions.

remove tr_decode, tr_exit i_a;

In the example above the transactions tr_decode and tr_exit are removed from the submodule with the hierarchical path i_a. They can be replaced by an alternative transaction with a different name.

The remove command is executed before move command.

2.3.6. Move command

The move command is executed within a fully joined module. The command defines a list of conditions, items, latches or registers and their associated signal cones. When executed, the group is moved to a given submodule, defined by a hierarchical module path.

In the following example, the signals “rs1_dato, rs2_dato, rf” and their associated logic is moved into the module with the instantiation path “i_rf”.

move rs1_dato, rs2_dato, rf i_rf;

2.3.7. Route command

In PDVL, signals are automatically routed, as defined in the next section. Nevertheless, the user can use the route command to route signals individually. These signals are then connected and are therefore not considered during the automatic routing process anymore.

route i_a.i_b.sig_a i_c.sig_b;

In this example, the signal “sig_a” from the instance with the hierarchical path “i_a.i_b” is taken and connected to the signal “sig_b” in the instance with the hierarchical path “i_c”. In this case, the routed signal throughout the module hierarchies has the same name as the source signal, “sig_a”. The routed signal can also have a given name:

```
route i_a.i_b.sig_a i_c.sig_b sig_c;
```

The routed signal has always the same type as the source signal. Assuming the relevant signal “sig_a” in the above example is of type vector in the above example, then the routed signal “sig_c” is of type vector of the same range.

The hierarchical paths are relative to the module, which is created by the build cluster of the place command. When the initial section of the 2 hierarchical paths is identical, then the signal is routed on the lowest possible level only. In the following example, the signal sig_c will be routed within the module with the instantiation name “i_a”.

```
route i_a.i_b.sig_a i_a.i_c.sig_b sig_c;
```

Events are not allowed to be used by the routing command.

Conditions with defined attributes are not allowed to be used by the routing command.

```
join {
   e_clk posedge clk;
   c_en high en; };

not allowed: route i_3.clk clk clk_routed;
not allowed: route i_4.en en en_routed;
```

2.3.8. Generate commands (if, case, for, foreach)

The conditional-generate and the loop-generate constructs known from SystemVerilog are supported. The SystemVerilog statement “genvar” is not needed in PDVL. The optional generate block statement in SystemVerilog is not supported in PDVL.

3.3.8.1 Conditional-generate

The conditional-generate commands use either known if-constructs or case constructs. The following conditional-generate commands are supported:

```
if ( a < 5 ) { }
```

```
case ( a )
   0, 1: { }
   any_parameter : { }
   default: { }
endcase
```

3.3.8.2 Loop-generate

The loop-generate commands use either known for-constructs or foreach-constructs. They loop a specified loop identifier over a given range and execute all commands in the construct's body using the individual identifier value. The following loop-generate commands are supported:

```
for (i = 0; i < any_parameter; i = i + 1) { }
foreach i in 3 { }
foreach i in (master, slave_0, slave_1) { }
```

The foreach statement including a list of strings uses the individual strings as identifier replacement, not their numbered position in the list.

The main difference to SV loop-generate statement is, that not only individual identifiers of the loop index are updated, but also individual string can be changed, if one of the following rules applies (here "i" is the loop index and its value is 0):

1. If the string ends with "_i", the string ending is changed to "_0".
2. It the string contains one or multiple "_i_" sections, the last string section is changed to "_0_".

The following example takes the identifier "n" and loops twice (range 0 to 1) over the foreach body. By doing that, it places the submodules "GPIO_0" and "GPIO_1" with its instantiations "i_gpio_0" and "i_gpio_1" respectively.

```
foreach n in 2 { place GPIO_n i_gpio_n; }
```

3.3.8.3 Generate statements in build clusters and standard clusters

Generate commands are not only elaborated in build clusters, but also in standard clusters. When parameters are used in the conditional-generate statement, then they must be defined in the relevant clusters, as the following example shows.

In this example, the system bus cluster axi is used.

Uniquify system bus into new cluster:

uniquify axi ic0_;

Place parameters into new cluster:

join { parameter ic0_axi_masters = 1;
parameter ic0_axi_slaves = 3; } ic0_axi;

Before the new cluster and its subclusters are placed, the generate statements are elaborated.

Place new cluster and/or its subclusters into individual modules:

join ic0_axi.ic0_axi_slv_wr_0 i_gpio_0;

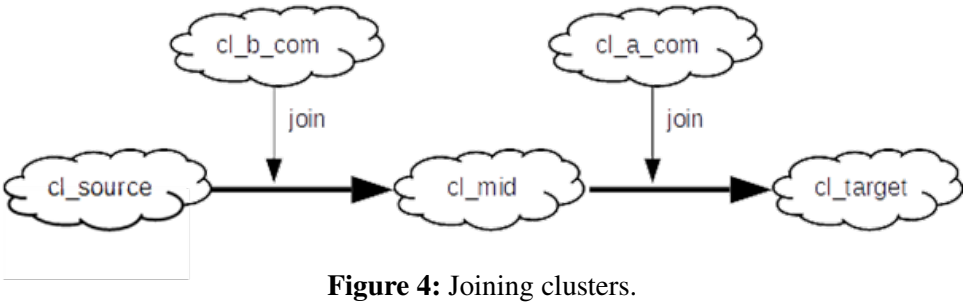

Figure 4: Joining clusters.

2.4. PDVL to RTL converter strategy

This section specifies the rules, how PDVL designs are converted into SystemVerilog designs. The conversion process is done in five steps:

1. Step I: Cluster elaboration (Clusters generating clusters)
2. Step II: Structural composition (Clusters generating modules and hierarchy)
3. Step III: Logic joining
4. Step IV: Logic relocation
5. Step V: Signal routing

The compiled database can then be written in SystemVerilog format.

2.4.1. Step I: Cluster elaboration

In Step I only commands that do affect other clusters are executed. In the Figure 4 example, the commands in cluster cl_b_com are elaborated first, before cl_a_com commands are executed. The key commands in step I are:

1. Join body/cluster into cluster
2. Uniquify cluster
3. Unrolling of foreach commands

The order of cluster elaboration is not specified. The exception is, that a cluster is not elaborated, if it is a target cluster of another command (join, uniquify, etc.).

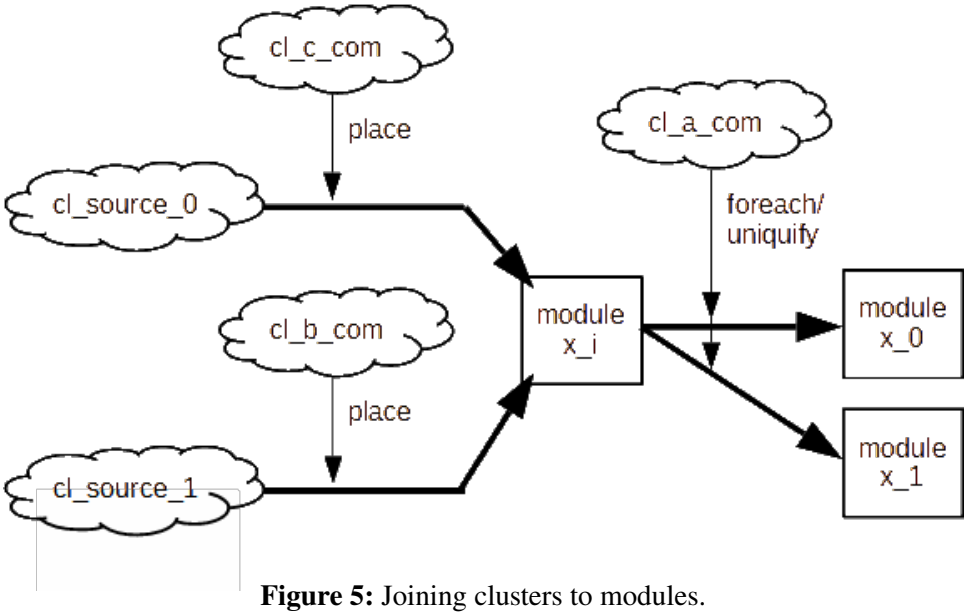

Figure 5: Joining clusters to modules.

2.4.2. Step II: Structural composition

In step II of the PDVL to SystemVerilog conversion process, the structural composition of the module hierarchy is achieved. Clusters and additional logic is joined in the modules, once the module hierarchy is defined.

In the Figure 5 example, the clusters “cl_source_0” and “cl_source_1” are elaborated first, before being joined in module “x_i”. Once the module “x_i” will not change anymore, commands with uniquify the module to “x_0” and “x_1” are executed.

The key commands in step II are:

1. Place command to generate hierarchy
2. Join body/cluster into modules
3. Uniquify modules
4. Unrolling of foreach commands

The order of structural composition step is not specified. The exception is, that a module is not uniquified, if it is a target module of another command (join, uniquify, etc.).

2.4.3. Step III: Logic joining and connecting signals by attribute

This section defines, how the logic cones of individual signals (combinatorial, latch or register) are extracted, after logic has been joined to individual SystemVerilog modules.

Items can have default assignments. Conditions can have conditions bodies, which define, when a condition has a certain value. Nevertheless, most logic is defined in transaction.

Transactions define when an assignment to a signal is valid. Transactions can also call other transactions. This is a pure syntactical merge of transactions bodies. The order is preserved. Therefore, the resulting tree of a transaction merge is predictive.

Once transaction trees are merged, the datapath valid assignments are replaced by the individual signal assignments. Again, the order is preserved, and the resulting tree of a transaction and datapath merge is predictive.

In a next step, the logic cone for each individual signal is extracted from the merged transactions. PDVL only allows non-blocking signal assignments in datapaths. The extracted logic cone does therefore not depend on intermediate values which typically result from sequential blocking assignments.

Thus, the logic joining process is predictive.

The final task during the logic joining process is to connect the signals (resulting from items, latches, registers, or conditions) by attribute.

3.4.3.1 Standard transaction merging

When the transaction body contains a list of transactions calls without any additional keywords, then these transaction calls are replaced by the individual transaction bodies of the called transactions. This is the default case. The transaction bodies of the called transactions are listed consecutively.

Example:

```
tr_a { tr_b;
       tr_c }
tr_b { @c_v { ... }
       else @c_x { ... } }
tr_c { @c_y { ... }
        d_fritz;
        @c_z { ... } }
```

results in:

```
tr_a {  @c_v { ... }
        else @c_x { ... }
        @c_y { ... }
        d_fritz;
        @c_z { ... } }
```

3.4.3.2 Transaction merging using attribute “unique”

The keyword “unique” at the beginning of the transaction call list joins the transaction bodies of the called transactions tagged with the unique attribute.

Example:

```
tr_a { unique tr_b;
```

```
        tr_c }
tr_b { unique @c_x { . . .  }
        @c_y { . . .  } }
tr_c { unique @c_y { . . .  }
        @c_z { . . .  } }
```

results in:

```
tr_a { unique @c_x { . . .  }
        @c_y { . . .  }
        @c_z { . . .  } }
```

3.4.3.3 Transaction merging using attribute “priority”

The keyword “priority” at the beginning of the list of transaction calls joins the transaction bodies of the called transactions using priority decoding.

Example:

```
tr_a { priority tr_b;
        tr_c }
tr_b { priority @c_v { . . .  }
        @c_x { . . .  } }
tr_c { priority @c_y { . . .  }
        @c_z { . . .  } }
```

results in:

```
tr_a { priority @c_v { . . .  }
        @c_x { . . .  }
        @c_y { . . .  }
        @c_z { . . .  } }
```

3.4.3.4 Case statements are handled like if-else statements

Assuming a transaction body contains a list of transaction calls with case or multiple case statements. When priority or unique is used, the following applies. A case statement is converted into the analog if-else representation. Then the if-else rules are applied.

During the write section, it is tried to reconvert the if-else statement into a case statement for better readability.

3.4.3.5 Called transactions

The usage of the priority or the unique attribute requires, that the transaction bodies of the called

transactions use one or multiple conditional statements. Therefore, a called transaction body must have either
one conditional statement or
a list of conditional statements with a priority (default) or unique attribute.

An exception to this rule is when the priority attribute is used. Then the last statements is considered as default when no conditional statement is used.

3.4.3.6 Mixture of attributes not allowed as of now

We argue at this moment of time, that a mixture of priority and unique attributes is not possible to be perfectly reflected in HDL. Future versions might allow corner cases where applicable. The user can always use the body of an if or if else statement to define if-else sub-statements with a different attribute. Therefor only priority (default) or unique are merged, otherwise error.

3.4.3.7 Identical conditions, unique attribute: no rules

3.4.3.8 Identical conditions, priority attribute:

After an initial merge, multiple identical conditional statements can appear in the if-else statement list. At this moment of time we argue that a merging algorithm is mandatory. Therefore, a conditional statement and its relevant sub-tree is merged to the first appearance of the conditional statement. The sub-tree is tree joined, as it is explained later.

The result must be compliant with all initial transactions. Otherwise, an error is given. For the unique attribute, no limitations are known. Simple, not post-synthesis comparison. No bit-wise check, rule must apply to LHS. No equivalence checking for conditions.

Example:

```
tr_a { priority tr_b;
       tr_c }
tr_b { priority @c_x { ... }
       @c_y { ... }
       @c_z { ... } }
tr_c { priority @c_x { ... }
       @c_z { ... } }
```

violation: tr_c cannot be merged.

In the above example, transaction tr_c cannot be merged, because the result would be identical to tr_b, which implies, that condition c_y is checked before c_z, which is not compatible to tr_c.

3.4.3.9 Connecting signals by attribute

In the final task during the logic joining process signals (resulting from items, latches, registers, or conditions) are automatically connected by attribute when applicable. They are always connected within a module only. This is fundamentally different to the signal routing step (step V), which connects signals with the same name throughout module boundaries.

The idea behind this concept is, that elements with the same name can be placed in two different modules. In one module, it is part of the relevant logic, whereas in the other module, it is connected by attribute with an element, which has a different name. The placed element is then routed by name.

For instance, a system bus can place elements at masters, slaves, or arbiters. In an arbiter, the elements can relate to the relevant logic. In a master or slave module, the elements are connected by attribute to the relevant signals with the same attribute, whereas the complete system bus is then automatically routed by name.

The following example shows two item definitions with the same attribute “AXADDR”.

```
(* AXADDR=DRIVER *) item write_add [31:0];
(* AXADDR *) item axi_mst_wr_addr [31:0];
```

When two signals within the same module have the same attribute after joining, they are automatically connected. The above example results in the following System Verilog code. (The item write_add becomes the driver (or right-hand-side (RHS)) of the assignment, and the item axi_mst_wr_add becomes the assigned signal (or left-hand-side (LHS)):))

```
assign axi_mst_wr_addr = write_add;
```

Items, latches, and registers can only be connected among each other but not to conditions.

Conditions can only be connected to other conditions, but not to items, latches, and registers.

Signals can only be assigned to items and conditions can only be connected to other conditions.

Signals can never be assigned to latches or registers because their logic cone would be overwritten.

The signals having an attribute with the “driver” assignment are consider as the driving signal in the assignment. Exactly one signal of all signals with the same attribute must have the attribute with a “driver” assignment, otherwise, no connection is generated. If more than one driver assignments of the same attribute are found, the following error is given:

ERR.CONNECT.MULTIPLE_DRIVERS

Multiple assignments to different signals by the same driving signal is possible.

2.4.4. Step IV: Logic relocate

In this step the move command is executed. The algorithm passes all clusters again and detects the build command to identify the relevant reverence design. All move commands following a build command are then executed relative to the reference module.

At this point of time, all wires and registers are defined. They all have their associative logic cone defined, which drive the wire/register. The move command now picks the wire/register and its associated logic cone and places it in the module, which is specified in the move command using a hierarchical name.
So far it is only possible to move logic within the sub hierarchy of the reference module. Future PDVL versions might allow the moving of logic above the reference module.

2.4.5. Step V: Signal route

In step V of the PDVL to SystemVerilog conversion process, the signals are routed. This is done by executing the explicit routing commands first, followed by an automatic routing process. The automatic routing process includes:

1. signals, which cross module hierarchy (inputs/outputs/inouts) are routed,
2. parameters are passed throughout module hierarchy, and
3. signals are connected by attribute within a module.

3.4.5.1 Explicit routing commands

PDVL supports explicit routing commands between 2 signals. These commands are executed before the automatic routing process starts. Signals driven by a routed signal are not considered undriven anymore and are therefore not automatically routed.

3.4.5.2 Cross module hierarchy signal routing

The automatic routing algorithm is based on exact name matching. A signal which is driven by logic within a module is called a driver signal. A signal which is not driven by any logic within a module (undriven signal) is connected to a driver signal of the same name from a different module.

If multiple driver signals with the same name exist, the undriven signal is connected to the driver signal with the shortest possible path within the SystemVerilog module hierarchy. When multiple driver signals with the same possible shortest hierarchical distance from the unconnected signal exist, then the conversion process should end with an error.

There is only one distinctive driver per routed signal. The routed signal has the same type as the driver signal. The number of signals to be connected is not limited.

An exception to these rules is the usage of tristate signals, which is explained in chapter Handling tristate routing.

3.4.5.3 Parameter passing throughout module hierarchy

The parameter passing algorithm is based on exact name matching. Parameters are defined in clusters. Parameters can also result from the cluster joining step (e.g., parameters related to finite state machines).

A procedure needs to take care of parameters, which are not defined within a particular module, but defined elsewhere. It is optional to pass parameter values from module to submodule, or to re-define the parameter in the relevant module.

If multiple defined parameters with the same name exist, the undefined parameter takes over the value of the defined parameter with the shortest possible path within the SystemVerilog module hierarchy. When multiple defined parameters (with different assignment values and) with the same possible shortest hierarchical distance from the undefined parameter exist, then the conversion process should end with an error.

2.5. PDVL to SystemVerilog conversion rules

When converting PDVL into synthesizable SystemVerilog, certain conversion rules must be considered.

2.5.1. Latch enable, register clocking, and reset handling

The following PDVL constructs are needed to define latches and registers in the SystemVerilog output files.

3.5.1.1 Latch enable

A latch must be defined as such. The latch enable signal is the result of a condition. If defined, the signal name and the high-active (or low-active) attribute of the condition is used.

Table 2: PDVL code and SystemVerilog output

PDVL	**SystemVerilog**

latch a;
c_en low en; // default
d_a { a <= . . . ; };
transaction { @c_en { d_a; } }
	always_latch (en)
	if (!en)
	a <= . . . ;

The condition c_en only needs to be defined once in the design. If multiple c_en conditions exist, the one on the closest hierarchical path is used.

3.5.1.2 Register w/o reset

A register must be defined as such. The clock signal is the result of an event. If defined, the signal name and the positive-edge (or negative-edge) attribute of the event is used.

The event e_clk only needs to be defined once in the design. If multiple e_clk events exit, the one on the closest hierarchical path is used.

3.5.1.3 Register with reset

A register can have a reset enable signal. The reset enable signal is the result of a condition

Table 3: PDVL code and SystemVerilog output

PDVL	SystemVerilog

reg a;
e_clk posedge clk; // default
d_a { a <= . . . ; };
transaction { @e_clk { } }
	always_ff (posedge clk)
	a <= . . . ;

If defined, the signal name and the high-active (or low-active) attribute of the condition is used.

Table 4: PDVL code and SystemVerilog output

PDVL	SystemVerilog

reg a;
e_clk posedge clk; // default
c_rst low rstn; // no default
d_a { a <= . . . ; };
transaction { @e_clk { } }
	always_ff (posedge clk or negedge rstn)
	if (!rstn)
	a <= . . . ;

The condition c_rst only needs to be defined once in the design. If multiple c_rst conditions exist, the one on the closest hierarchical path is used.

2.5.2. Blocking vs non-blocking assignments

PDVL only supports blocking assignments.They are used in datapaths and during item definition:

```
item a;
d_a { a = 1'b1; }
item b = 1'b0;
reg [1:0] c;
d_c { c = 2'b10; }
```

When the SystemVerilog code is generated, the following rules are recommended:

When a signal with combinatorial logic is written inside an always_comb block, blocking assignments should be used.

```
Always_comb begin
. . .
```

```
a = 1'b1;
...
end
```

When a signal with sequential logic is used (e.g. always_ff block), then non-blocking assignments should be used.

```
always_ff (...) begin
...
c <= 2'b10;
...
end
```

2.5.3. Signal dependencies within datapaths

PDVL defines, that each logic cone must be extracted stand alone. Example:

```
item a, b;
reg c;
d_abc { a = 1'b1; b = a; c = b;}
```

This will result in the following SystemVerilog code:

```
always_comb begin ... a = 1'b1; ... end
always_comb begin ... b = a; ... end
always_ff (...) begin ... c <= b; ... end
```

2.5.4. Signal assignment order

Transactions determine, when a datapath is valid. Example:

```
item [3:0] a;
d_a0 { a = 4'b1; }; d_a1 { a = a + 5; }
tr_a0 { d_a0; }; tr_a1 { d_a1; }
```

In the example above, both transactions (tr_a0 and tr_a1) are not merged into another transaction. The resulting signal assignment order is not deterministic:

```
always_comb begin a = 4'b1; a = a + 5; end
```

always_comb begin a = a + 5; a = 4'b1; end

An additional transaction can define the signal assignment order, by calling them with an explicit order:
tr_a { tr_a0; tr_a1; }

Which results in

always_comb begin . . . a = 4'b1; a = a + 5; . . . end

2.5.5. Parameter usage

There are 3 kinds of parameter usage:

1. parameters used during build commands (e.g. generate build commands like foreach)
2. parameters used during cluster elaboration (e.g. generate commands like foreach) and
3. parameters used in the logic itself (e.g. FSM machine state definitions).

The following code gives an example:

```
build testbench_tri_1 {
   parameter blocks = 3; // parameter members only known to this body
   join { parameter members = 3; } tri; // parameter member only known to     cluster tri
   // might be needed for subcluster generating
   foreach n in members { // parameter usage
         place TRI_n i_n;
}
join { parameter bw = 8; } i_0;
```

The parameter blocks is only known to this body (of testbench_tri).

The parameter members is joined in the cluster tri, which might need it for subcluster generation.

The parameter bw is joined in the submodule TRI_0 with the instantiation i_0. The parameter bw is then reused in the submodules TRI_1 and TRI_2 as well because it is relevant there, too. This parameter reuse technique is not applied on parameter blocks and members.

2.5.6. Routing tristate signals

Tristate signals are handled by clusters with multiple subclusters, whereas each subcluster has signals of the same name, which are drivers and which are used by other signal within each subclusters.

```
cluster tri {

foreach n in tri_buffers { // generates “tri_buffers” subclusters tbuf_n
   cluster tbuf_n {
   item data_io ]bw]; // tristate logic for signal data_io
   item data_tri ]bw];
   item data_out ]bw];
   item data_in ]bw];
   d_tri {
      // input
      for (b = 0; b < bw; b = b + 1) begin
            if (!data_io[b]) // data_io is used
                  data_in[b] = 1’b0;
            else
                  data_in[b] = 1’b1;
            end
      // tristate output
      for (b = 0; b < bw; b = b + 1) begin
            if (data_tri[b])
                  data_io[b] = 1’bZ; // data_io is driven
            else
                  data_io[b] = data_out[b];
            end
      }
      tr_tri { d_tri; }
      }
}
```

These subclusters are joined using a specific join command:

```
join tri { tbuf_0 i_0;
tbuf_1 i_1;
tbuf_2 i_2; }
```

The above example joins the subclusters tri.tbuf_0 (tri.tbuf_1, tri.tbuf_2) in the submodule i_0 (i_1, i_2). The signal “data_io”, which is a driver and which is used in other subclusters is connected via tristate routing signals (inouts). Using individual join commands will not result in a routed data_io signal:

```
// data_io is not routed.
```

```
join tri { tbuf_0 i_0; }
join tri { tbuf_1 i_1; }
join tri { tbuf_2 i_2; }
```

3. PDVL verification specification (V)

To be published.

4. PDVL programming specification (P)

To be published.

5. PDVL grammar

```
pdvl_grammar
   : '<"' pdvl_declaration* '">'
   ;

pdvl_declaration
   : pdvl_build_command
   | pdvl_cluster_declaration
   ;
```

5.1. Cluster declarations

5.1.1. Clusters

```
pdvl_cluster_declaration
   : pdvl_cluster_identifier '{' pdvl_cluster_declaration_body* '}'
   ;

pdvl_cluster_identifier
   : pdvl_cluster_cl_si
   | pdvl_cluster_si
   ;

pdvl_cluster_cl_si
   : [c] [l] [_] [a-zA-Z0-9_]*
   ;

pdvl_cluster_si
   : 'cluster' si
   ;

pdvl_cluster_declaration_body
   : pdvl_sub_cluster_declaration
   | sv_parameter_declaration
   | sv_type_declaration
   | pdvl_cond_declaration
   | pdvl_event_declaration
   | pdvl_item_declaration
   | pdvl_reg_declaration
   | pdvl_latch_declaration
   | pdvl_data_declaration
   | pdvl_if_declaration
   | pdvl_case_declaration
```

```
  | pdvl_foreach_declaration
  | pdvl_for_declaration
  | pdvl_transaction_declaration
  | sv_function_declaration
  | pdvl_graph_declaration
  | pdvl_theorem_declaration
  | pdvl_proof_declaration
  ;

pdvl_sub_cluster_declaration
  : pdvl_cluster_identifier '{' pdvl_cluster_declaration_body* '}'
  ;
```

5.1.2. Parameters

PDVL parameter declarations are identical to SystemVerilog parameter declarations.

5.1.3. Itemes, latches and registers

```
pdvl_item_declaration
  : 'item' sv_attribute_instance? pdvl_identifier_type sv_blocking_assign_expression? ';'
  ;

pdvl_reg_declaration
  : 'reg' sv_attribute_instance? pdvl_identifier_type pdvl_transaction_macro_list? ';'
  ;

pdvl_latch_declaration
  : 'latch' sv_attribute_instance? pdvl_identifier_type pdvl_transaction_macro_list? ';'
  ;

pdvl_identifier_type
  : pdvl_packed_dimension_list pdvl_identifier_list pdvl_unpacked_dimension_list
  | pdvl_packed_dimension_list pdvl_identifier_list
  | pdvl_enum_data_type pdvl_identifier_list
  | pdvl_identifier_list pdvl_unpacked_dimension_list
  | pdvl_struct_or_union_data_type pdvl_identifier_list
  | pdvl_type_identifier_data_type pdvl_identifier_list
  | pdvl_identifier_list
  ;

pdvl_identifier_list
  : si list_of_si*
```

```
    ;

pdvl_packed_dimension_list
    : pdvl_packed_or_c_style pdvl_packed_dimension_list?
    ;


pdvl_packed_or_c_style
    : sv_packed_dimension
    | pdvl_c_style_dimension
    ;


pdvl_c_style_dimension
    : '[' sv_constant_expression ']'
    ;


pdvl_unpacked_dimension_list
    : sv_unpacked_dimension pdvl_unpacked_dimension_list?
    ;


pdvl_enum_data_type
    : 'enum' sv_packed_dimension_list? '' sv_enum_name_declaration_list ''
    ;


pdvl_struct_or_union_data_type
    : sv_struct_union sv_packed_keyword? '' sv_struct_union_member* ''
    ;


pdvl_type_identifier_data_type
    : sv_class_scope? sv_type_identifier
    ;


pdvl_transaction_macro_list
    : pdvl_at_condition '{' sv_expression ';' '}' pdvl_transaction_macro_list?
    | pdvl_at_condition '{' pdvl_data_declaration_body* '}' pdvl_transaction_macro_list?
    ;
```

5.1.4. Datapaths

```
pdvl_data_declaration
   : pdvl_data_identifier pdvl_tf_port_list_brackets? '{' pdvl_data_declaration_body* '}'
   ;


pdvl_data_identifier
   : pdvl_data_identifier_by_name
   | pdvl_transaction_and_data_identifier_by_name
   | pdvl_data_si
   ;


pdvl_data_identifier_by_name
   : [d] [_] [a-zA-Z0-9_]*
   ;


pdvl_transaction_and_data_identifier_by_name
   : [t] [r] [_] [d] [_] [a-zA-Z0-9_]*
   ;


pdvl_data_si
   : 'data' si
   ;


pdvl_data_declaration_body
   : sv_statement_item_blocking_assignment
   | sv_case_statement
   | sv_loop_statement
   ;
```

5.1.5. Conditions

```
pdvl_cond_declaration
   : pdvl_cond_declaration_level_body
   | pdvl_cond_declaration_body_port
   | pdvl_cond_declaration_body
   | pdvl_cond_declaration_level
```

```
    | pdvl_cond_declaration_reg
    | pdvl_cond_declaration_simple
    ;


pdvl_cond_declaration_level_body
    : pdvl_cond_identifier pdvl_level_identifier pdvl_cond_signal_name '{' pdvl_cond_conditional_statement*
'}'
    ;


pdvl_cond_declaration_body_port
    : pdvl_cond_identifier pdvl_tf_port_list_brackets? '{' pdvl_cond_conditional_statement* '}'
    ;


pdvl_cond_declaration_body
    : sv_attribute_instance? pdvl_cond_identifier '{' pdvl_cond_conditional_statement* '}'
    ;


pdvl_tf_port_list_brackets
    : '(' sv_function_port_list? ')'
    ;


pdvl_cond_declaration_level
    : pdvl_cond_identifier pdvl_level_identifier pdvl_cond_signal_name ';'
    ;


pdvl_level_identifier
    : pdvl_high_level_identifier
    | pdvl_low_level_identifier
    ;


pdvl_high_level_identifier
    : 'high'
    ;


pdvl_low_level_identifier
    : 'low'
    ;
```

```
pdvl_cond_signal_name
   : si
   ;


pdvl_cond_declaration_reg
   : pdvl_list_of_cond_identifiers 'reg' ';'
   ;


pdvl_cond_declaration_simple
   : sv_attribute_instance? pdvl_list_of_cond_identifiers ';'
   ;


pdvl_list_of_cond_identifiers
   : pdvl_list_of_cond_identifiers_si
   | pdvl_list_of_cond_identifiers_by_name
   ;


pdvl_list_of_cond_identifiers_si
   : 'cond' si list_of_si*
   ;


pdvl_list_of_cond_identifiers_by_name
   : pdvl_cond_identifier_by_name pdvl_list_of_cond_identifiers_by_name_extension*
   ;


pdvl_list_of_cond_identifiers_by_name_extension
   : ',' pdvl_cond_identifier_by_name
   ;


pdvl_cond_identifier
   : pdvl_cond_identifier_by_name
   | pdvl_cond_si
   ;


pdvl_cond_identifier_by_name
```

```
   : [c] [_] [a-zA-Z0-9_]*
   ;


pdvl_cond_si
   : 'cond' si
   ;


pdvl_cond_conditional_statement
   : 'if' '(' sv_expression ')' pdvl_implicit_cond_handle ';'
   ;


pdvl_implicit_cond_handle
   : 'this'
   ;

   // not yet used:
pdvl_cond_macro_list
   : 'initial' '{' pdvl_data_declaration_body* '}'
   ;
```

5.1.6. Events

```
pdvl_event_declaration
   : pdvl_list_of_event_identifiers sv_edge_identifier pdvl_event_signal_name ';'
   ;


pdvl_list_of_event_identifiers
   : pdvl_list_of_event_identifiers_si
   | pdvl_list_of_event_identifiers_by_name
   ;


pdvl_list_of_event_identifiers_si
   : 'event' si (',' si)*
   ;


pdvl_list_of_event_identifiers_by_name
   : pdvl_event_identifier_by_name pdvl_list_of_event_identifiers_by_name_extension*
```

```
    ;


pdvl_list_of_event_identifiers_by_name_extension
    : ',' pdvl_event_identifier_by_name
    ;


pdvl_event_identifier
    : pdvl_event_identifier_by_name
    | pdvl_event_si
    ;


pdvl_event_identifier_by_name
    : [e] [_] [a-zA-Z0-9_]*
    ;

pdvl_event_si
    : 'event' si
    ;

pdvl_event_signal_name
    : si
    ;
```

5.1.7. Transactions

```
pdvl_transaction_declaration
    : pdvl_transaction_declaration_body
    ;


pdvl_transaction_declaration_body
    : pdvl_transaction_identifier pdvl_transaction_attribute? '{' pdvl_transaction_statement* '}'
    ;


pdvl_transaction_attribute
    : si
    ;
```

```
pdvl_transaction_identifier
   : pdvl_transaction_identifier_by_name
   | pdvl_transaction_si
   ;


pdvl_transaction_identifier_by_name
   : [t] [r] [_] [a-zA-Z0-9_]*
   ;


pdvl_transaction_si
   : 'transaction' si
   ;


pdvl_transaction_statement
   : pdvl_at_transaction_statement
   | pdvl_finite_transaction_statement
   | pdvl_pipe_transaction_statement
   | pdvl_inter_transaction_statement
   | pdvl_context_transaction_statement
   | pdvl_transaction_priority_list
   | pdvl_transaction_unique_single
   | pdvl_transaction_unique_list
   | pdvl_transaction_unique0_single
   | pdvl_transaction_unique0_list
   | pdvl_case_transaction_statement
   | pdvl_next_state_statement
   | pdvl_transaction_state
   | pdvl_emit_delayed_event
   | pdvl_do_not_emit
   | pdvl_emit
   ;


pdvl_transaction_body
   : '{' pdvl_transaction_statement* '}'
   ;


pdvl_transaction_state
   : pdvl_transaction_state_body
   | pdvl_transaction_state_statement
   ;
```

```
pdvl_priority_unique_propagate
   : 'propagate'
   ;


pdvl_transaction_priority_list
   : 'priority' pdvl_priority_unique_propagate? '{' pdvl_transaction_priority_unique_list_entry*
'}'
   ;


pdvl_transaction_priority_unique_list_entry
   : pdvl_at_transaction_single_statement
   ;


pdvl_transaction_unique_single
   : 'unique' pdvl_priority_unique_propagate? pdvl_transaction_priority_unique_list_entry
   ;


pdvl_transaction_unique_list
   : 'unique' pdvl_priority_unique_propagate? '{' pdvl_transaction_priority_unique_list_entry*
'}'
   ;


pdvl_transaction_unique0_single
   : 'unique0' pdvl_priority_unique_propagate? pdvl_transaction_priority_unique_list_entry
   ;


pdvl_transaction_unique0_list
   : 'unique0' pdvl_priority_unique_propagate? '{' pdvl_transaction_priority_unique_list_entry*
'}'
   ;


pdvl_transaction_state_body
   : si ':' pdvl_transaction_body semikolon?
   ;


pdvl_transaction_state_statement
```

```
  : si ':' pdvl_transaction_statement
  ;
```

5.1.8. If generate declaration

```
pdvl_if_declaration
  : 'if' '(' sv_constant_expression ')' '{' pdvl_cluster_declaration_body* '}' pdvl_else_if_declaration*
pdvl_else_declaration?
  ;


pdvl_else_if_declaration
  : 'else' 'if' '(' sv_constant_expression ')' '{' pdvl_cluster_declaration_body* '}'
  ;


pdvl_else_declaration
  : 'else' '{' pdvl_cluster_declaration_body* '}'
  ;
```

5.1.9. Case generate declaration

```
pdvl_case_declaration
  : 'case' '(' sv_constant_expression ')' pdvl_case_declaration_item* 'endcase'
  ;


pdvl_case_declaration_item
  : pdvl_case_declaration_default
  | pdvl_case_declaration_constant_expression
  ;


pdvl_case_declaration_constant_expression
  : pdvl_list_of_constant_expression ':' '{' pdvl_cluster_declaration_body* '}'
  ;


pdvl_case_declaration_default
  : 'default' ':' '{' pdvl_cluster_declaration_body* '}'
```

;

5.1.10. For generate declaration

pdvl_for_declaration
: 'for' '(' 'int' sv_genvar_initialization ';' sv_genvar_expression ';' sv_genvar_iteration ')' '{' pdvl_cluster_declaration_body* '}'
;

5.1.11. Foreach generate declaration

pdvl_foreach_declaration
: 'foreach' si 'in' pdvl_foreach_list_def '{' pdvl_cluster_declaration_body* '}'
;

5.1.12. General constructs

pdvl_at_condition
: pdvl_condition_control pdvl_condition_expression
;

pdvl_condition_control
: pdvl_at_false_condition
| pdvl_at_true_condition
;

pdvl_at_false_condition
: '@' '!'
;

pdvl_at_true_condition
: '@'
;

pdvl_condition_expression

```
    : si pdvl_expression_list_brackets?
    | '(' sv_expression ')'
    ;


pdvl_expression_list_brackets
    : '(' sv_expression sv_expression_list* ')'
    ;


pdvl_at_keyword_prio
    : 'prioxity'
    ;


pdvl_at_keyword
    : pdvl_at_keyword_prio
    ;


pdvl_at_transaction_statement
    : pdvl_at_transaction_single_else_statement
    | pdvl_at_transaction_single_statement
    ;


pdvl_at_transaction_single_statement
    : pdvl_at_condition pdvl_transaction_body semikolon?
    ;


pdvl_at_transaction_single_else_statement
    : pdvl_at_condition pdvl_transaction_body 'else' pdvl_transaction_body semikolon?
    | pdvl_at_condition pdvl_transaction_body 'else' pdvl_at_transaction_single_else_statement semikolon?
    | pdvl_at_condition pdvl_transaction_body 'else' pdvl_at_transaction_single_statement semikolon?
    ;


pdvl_finite_transaction_statement
    : pdvl_finite_transaction_keyword pdvl_finite_transaction_one_hot_keyword* pdvl_finite_transaction_ident
'' pdvl_case_transaction_item* ''
    ;


pdvl_finite_transaction_keyword
```

: 'finite'
;

pdvl_finite_transaction_one_hot_keyword
: 'one_hot'
;

pdvl_finite_transaction_identifier
: si
;

pdvl_pipe_transaction_statement
: pdvl_pipe_transaction_keyword sv_number pdvl_transaction_body
| pdvl_pipe_transaction_keyword sv_packed_dimension pdvl_transaction_body
;

pdvl_pipe_transaction_keyword
: 'pipe'
;

pdvl_inter_transaction_statement
: pdvl_inter_transaction_keyword pdvl_inter_transaction_identifier '' pdvl_case_transaction_item*
''
;

pdvl_inter_transaction_keyword
: 'inter'
;

pdvl_inter_transaction_identifier
: si
;

pdvl_context_transaction_statement
: pdvl_context_transaction_keyword sv_number pdvl_context_transaction_port_identifier '' pdvl_case_trans
''
;

```
pdvl_context_transaction_keyword
   : 'context'
   ;


pdvl_context_transaction_port_identifier
   : si
   ;


pdvl_case_transaction_statement
   : sv_unique_priority? pdvl_case_transaction_keyword '(' pdvl_case_transaction_expression ')'
pdvl_case_transaction_item* 'endcase'
   ;


pdvl_case_transaction_keyword
   : 'case'
   ;


pdvl_case_transaction_expression
   : sv_expression
   ;


pdvl_case_transaction_item
   : pdvl_case_transaction_item_expression ':' pdvl_transaction_body
   ;


pdvl_case_transaction_item_expression
   : sv_expression
   ;


pdvl_next_state_statement
   : '#' si ';'
   ;


pdvl_emit_delayed_event
   : sv_delay_control sv_expression ';'
```

```
    ;


pdvl_emit
    : si pdvl_expression_list_brackets? ';'
    ;


pdvl_do_not_emit
    : '!' si ';'
    ;
```

5.2. Build design hierarchy and connectivity

5.2.1. Build command

```
pdvl_build_command
    : 'build' sv_module_identifier '{' pdvl_build_command_body* '}'
    ;

pdvl_build_command_body
    : sv_parameter_declaration
    | pdvl_place_command
    | pdvl_uniquify_command
    | pdvl_join_body_command
    | pdvl_join_cluster_command
    | pdvl_remove_command
    | pdvl_move_command
    | pdvl_route_command
    | pdvl_if_command
    | pdvl_case_command
    | pdvl_for_command
    | pdvl_foreach_command
    ;


pdvl_parameter_declaration
    : sv_parameter_declaration
    ;
```

5.2.2. Uniquify command

```
pdvl_uniquify_command
   : 'uniquify' si si ';'
   ;
```

5.2.3. Join command

```
pdvl_join_command
   : pdvl_join_body_command
   | pdvl_join_cluster_command
   ;

pdvl_join_body_command
   : 'join' pdvl_join_body hierarchical_name ';'
   | 'join' pdvl_join_body semikolon?
   ;

pdvl_join_body
   : '{' pdvl_cluster_declaration_body* '}'
   ;

pdvl_join_cluster_command
   : pdvl_join_cluster_body_command
   | pdvl_join_cluster_single_command
   ;

pdvl_join_cluster_body_command
   : 'join' si '' pdvl_join_cluster_body* ''
   ;

pdvl_join_cluster_body
   : hierarchical_name hierarchical_name? ';'
   ;

pdvl_join_cluster_single_command
   : 'join' hierarchical_name hierarchical_name? ';'
   ;
```

5.2.4. Remove command

pdvl_remove_command
: 'remove' si list_of_si* hierarchical_name ';'
;

5.2.5. Place command

pdvl_place_command
: 'place' sv_module_identifier hierarchical_name ';'
;

5.2.6. Move command

pdvl_move_command
: 'move' si list_of_si* hierarchical_name ';'
;

5.2.7. Routing command

pdvl_route_command
: 'route' hierarchical_name hierarchical_name pdvl_route_signal_identifier* ';'
;

pdvl_route_signal_identifier
: si
;

5.2.8. If command

pdvl_if_command
: 'if' '(' sv_constant_expression ')' '{' pdvl_build_command_body* '}' pdvl_else_if_command* pdvl_else_command?
;

```
pdvl_else_if_command
   : 'else' 'if' '(' sv_constant_expression ')' '{' pdvl_build_command_body* '}'
   ;


pdvl_else_command
   : 'else' '{' pdvl_build_command_body* '}'
   ;
```

5.2.9. Case command

```
pdvl_case_command
   : 'case' '(' sv_constant_expression ')' pdvl_case_command_item* 'endcase'
   ;


pdvl_case_command_item
   : pdvl_case_command_default
   | pdvl_case_command_constant_expression
   ;


pdvl_case_command_constant_expression
   : pdvl_list_of_constant_expression ':' '{' pdvl_build_command_body* '}'
   ;


pdvl_list_of_constant_expression
   : sv_constant_expression (',' sv_constant_expression)*
   ;


pdvl_case_command_default
   : 'default' ':' '{' pdvl_build_command_body* '}'
   ;
```

5.2.10. For command

```
pdvl_for_command
   : 'for' '(' 'int' sv_genvar_initialization ';' sv_genvar_expression ';' sv_genvar_iteration ')' '{'
```

```
pdvl_build_command_body* '}'
   ;
```

5.2.11. Foreach command

```
pdvl_foreach_command
   : 'foreach' si 'in' pdvl_foreach_list_def '{'pdvl_build_command_body* '}'
   ;


pdvl_foreach_list_def
   : pdvl_foreach_list
   | pdvl_foreach_constant_expression
   ;


pdvl_foreach_list
   : '(' usi list_of_usi* ')'
   ;


pdvl_foreach_constant_expression
   : sv_constant_expression
   ;
```

6. SystemVerilog extensions

6.0.1. A.1 Source text

6.0.2. A.1.1 Library source text

```
sv_library_text
   : sv_library_description
   ;


sv_library_description
   : sv_include_statement
   ;


sv_include_statement
   : ’‘’ ’include’ sv_file_path_spec
   ;


sv_file_path_spec
   : ’"’ si (’.’ si)* ’"’
   ;


sv_precompiler
   : sv_precompiler_directives*
   ;


sv_precompiler_directives
   : sv_define_directive
   | sv_ifdef_directive
   | sv_ifndef_directive
   | sv_elsif_directive
   | sv_else_directive
   | sv_endif_directive
   | sv_undefineall_directive
   | sv_undef_directive
   | sv_include_directive
   | sv_default_nettype_directive
   | sv_pragma_directive
   | sv_timescale_directive
   | sv_function_call_directive
```

```
    | sv_macro_directive
    | sv_text
    ;


sv_text
    : [ -_a-~]*
    ;


sv_ifdef_directive
    : '`' 'ifdef' si
    ;


sv_ifndef_directive
    : '`' 'ifndef' si
    ;


sv_elsif_directive
    : '`' 'elsif' si
    ;


sv_else_directive
    : '`' 'else'
    ;


sv_endif_directive
    : '`' 'endif'
    ;


sv_function_call_directive
    : '`' sv_tf_call
    ;


sv_macro_directive
    : '`' si
    ;
```

```
sv_define_directive
   : '`' 'define' sv_define_macro
   ;


sv_define_macro
   : sv_define_macro_tf_call
   | sv_define_macro_simple
   ;


sv_define_macro_tf_call
   : sv_tf_call sv_text_EOL
   ;


sv_define_macro_simple
   : si (sv_text_EOL)?
   ;


sv_text_EOL
   : sv_text EOL
   ;


EOL
   : [
n]
   ;


sv_default_nettype_directive
   : '`' 'default' us_digit 'nettype' sv_text_EOL
   ;


sv_pragma_directive
   : '`' 'pragma' sv_text_EOL
   ;


sv_timescale_directive
   : '`' 'timescale' sv_text_EOL
   ;
```

```
sv_include_directive
   : '`' 'include' sv_text_EOL
   ;


sv_undefineall_directive
   : '`' 'undefineall'
   ;


sv_undef_directive
   : '`' 'undef' si
   ;
```

6.0.3. A.1.2 SystemVerilog source text

```
sv_description
   : sv_module_declaration
   | sv_library_text
   ;


sv_module_ansi_header
   : sv_module_keyword sv_module_identifier '(' sv_list_of_port_declarations? ')' ';'
   ;


sv_module_declaration
   : sv_module_ansi_header sv_non_port_module_item* 'endmodule'
   ;


sv_module_keyword
   : 'module'
   ;
```

6.0.4. A.1.3 Module parameters and ports

modified:

```
sv_list_of_ports
   : '(' sv_port (',' sv_port)* ')'
   ;


sv_list_of_port_declarations
   : sv_ansi_port_declaration (',' sv_ansi_port_declaration)*
   ;


sv_port_declaration
   : sv_inout_declaration
   | sv_input_declaration
   | sv_output_declaration
   ;


sv_port
   : sv_port_expression
   | '.' sv_port_identifier '(' sv_port_expression ')'
   ;


sv_port_expression
   : sv_port_reference
   | '' sv_port_reference (',' sv_port_reference)* ''
   ;


sv_port_reference
   : sv_port_identifier sv_constant_select
   ;


sv_port_direction
   : sv_port_input_direction
   | sv_port_output_direction
   | sv_port_inout_direction
   ;


sv_port_input_direction
   : 'input'
   ;
```

```
sv_port_output_direction
   : 'output'
   ;


sv_port_inout_direction
   : 'inout'
   ;


sv_ansi_port_declaration
   : sv_port_identifier
   ;
```

6.0.5. A.1.4 Module items

```
sv_module_common_item
   : sv_module_or_generate_item_declaration
   | sv_continuous_assign
   | sv_always_construct
   | sv_loop_generate_construct
   | sv_conditional_generate_construct
   ;


sv_module_item
   : sv_port_declaration
   | sv_non_port_module_item
   ;


sv_module_or_generate_item
   : sv_parameter_override
   | sv_module_param_instantiation
   | sv_module_instantiation
   | sv_module_common_item
   ;


sv_module_or_generate_item_declaration
   : sv_package_or_generate_item_declaration
```

;

Enable backward compatibility to Verilog by allowing port declarations here. Also include statement added here.

sv_non_port_module_item
: sv_port_declaration
| sv_generate_region
| sv_module_or_generate_item
| sv_include_statement
;

sv_parameter_override
: 'defparam' sv_list_of_defparam_assignments ';'
;

6.0.6. A.1.5 Configuration source text

6.0.7. A.1.6 Interface items

6.0.8. A.1.7 Program items

6.0.9. A.1.8 Checker items

6.0.10. A.1.9 Class items

6.0.11. A.1.10 Constraints

6.0.12. A.1.11 Package items

sv_package_or_generate_item_declaration
: sv_task_declaration
| sv_function_declaration
| sv_data_declaration
| sv_net_declaration
| sv_parameter_declaration
;

6.0.13. A.2 Declarations

6.0.14. A.2.1 Declaration types

6.0.15. A.2.1.1 Module parameter declarations

```
sv_parameter_declaration
   : 'parameter' sv_list_of_param_assignments ';'
   ;
```

6.0.16. A.2.1.2 Port declarations

```
sv_inout_declaration
   : sv_inout_type_declaration
   | sv_inout_implicit_declaration
   ;


sv_inout_type_declaration
   : 'inout' sv_net_port_type sv_list_of_port_identifiers ';'
   ;


sv_inout_implicit_declaration
   : 'inout' sv_list_of_port_identifiers ';'
   ;


sv_input_declaration
   : sv_input_type_declaration
   | sv_input_implicit_declaration
   ;


sv_input_type_declaration
   : 'input' sv_net_port_type sv_list_of_port_identifiers ';'
   ;


sv_input_implicit_declaration
   : 'input' sv_list_of_port_identifiers ';'
   ;
```

```
sv_output_declaration
   : sv_output_type_declaration
   | sv_output_implicit_declaration
   ;


sv_output_type_declaration
   : 'output' sv_net_port_type sv_list_of_port_identifiers ';'
   ;


sv_output_implicit_declaration
   : 'output' sv_list_of_port_identifiers ';'
   ;
```

6.0.17. A.2.1.3 Type declarations

```
sv_data_declaration
   : sv_data_type_or_implicit sv_list_of_variable_decl_assignments ';'
   ;


sv_net_declaration
   : sv_net_type_declaration_split_type
   | sv_net_declaration_split_implicit
   ;


sv_net_type_declaration_split_type
   : sv_net_type sv_data_type_or_implicit sv_list_of_net_decl_assignments ';'
   ;


sv_net_declaration_split_implicit
   : sv_net_type sv_list_of_net_decl_assignments ';'
   ;


sv_type_declaration
   : 'typedef' sv_data_type sv_type_identifier sv_variable_dimension* ';'
   ;
```

6.0.18. A.2.2 Declaration data types

6.0.19. A.2.2.1 Net and variable types

```
sv_data_type
   : sv_integer_vector_type sv_packed_dimension_list?
   | sv_integer_struct_union_type
   | sv_integer_enum_type
   | sv_integer_atom_type
   | sv_integer_type_identifier_type
   ;
 sv_packed_keyword? bug:

sv_integer_struct_union_type
   : sv_struct_union sv_packed_keyword? '' sv_struct_union_member* ''
   ;


sv_packed_keyword
   : 'packed'
   ;


sv_integer_enum_type
   : 'enum' sv_enum_base_type? '' sv_enum_name_declaration_list ''
   ;


sv_integer_type_identifier_type
   : sv_class_scope? sv_type_identifier
   ;


sv_data_type_or_implicit
   : sv_data_type
   | sv_implicit_data_type
   ;


sv_implicit_data_type
   : sv_packed_dimension_list
   ;


sv_enum_base_type
```

```
    : sv_integer_vector_type sv_packed_dimension_list?
    ;


sv_enum_name_declaration_list
    : si ('=' sv_constant_expression)? (',' sv_enum_name_declaration_list)?
    ;


sv_class_scope
    : sv_class_type '::'
    ;


sv_class_type
    : sv_class_identifier
    ;


sv_integer_atom_type
    : 'integer'
    ;


sv_integer_vector_type
    : sv_integer_vector_type_bit
    | sv_integer_vector_type_logic
    | sv_integer_vector_type_reg
    ;


sv_integer_vector_type_bit
    : 'bit'
    ;


sv_integer_vector_type_logic
    : 'logic'
    ;


sv_integer_vector_type_reg
    : 'reg'
    ;
```

```
sv_net_type
   : sv_net_type_split_wire
   ;

sv_net_type_split_wire
   : 'wire'
   ;

sv_net_port_type
   : sv_data_type_or_implicit
   ;

sv_struct_union_member
   : sv_data_type si ';'
   ;

sv_data_type_or_void
   : sv_data_type
   | 'void'
   ;

sv_struct_union
   : sv_struct
   | sv_union
   ;

sv_struct
   : 'struct'
   ;

sv_union
   : 'union'
   ;
```

6.0.20. A.2.2.2 Strengths

6.0.21. A.2.2.3 Delays

```
sv_delay_value
   : sv_unsigned_number
   ;
```

6.0.22. A.2.3 Declaration lists

sv_list_of_defparam_assignments
  : sv_defparam_assignment
  ;

sv_list_of_net_decl_assignments
  : sv_net_decl_assignment (',' sv_net_decl_assignment)*
  ;

sv_list_of_param_assignments
  : sv_param_assignment
  ;

sv_list_of_port_identifiers
  : sv_port_identifier sv_unpacked_dimension* (',' sv_list_of_port_identifiers)*
  ;

sv_list_of_tf_variable_identifiers
  : sv_port_identifier (',' sv_port_identifier)*
  ;

sv_list_of_variable_decl_assignments
  : sv_variable_decl_assignment (',' sv_variable_decl_assignment)*
  ;

6.0.23. A.2.4 Declaration assignments

sv_defparam_assignment
  : sv_hierarchical_parameter_identifier '=' sv_constant_param_expression
  ;

sv_net_decl_assignment
  : sv_net_identifier sv_unpacked_dimension* sv_blocking_assign_expression?
  ;

sv_param_assignment
  : si '=' sv_constant_param_expression
  ;

sv_variable_decl_assignment
  : sv_variable_identifier sv_variable_dimension*
  ;

6.0.24. A.2.5 Declaration ranges

```
sv_unpacked_dimension
   : sv_unpacked_dimension_constant_range
   | sv_unpacked_dimension_constant_expression
   ;

sv_unpacked_dimension_constant_range
   : '[' sv_constant_range ']'
   ;

sv_unpacked_dimension_constant_expression
   : '[' sv_constant_expression ']'
   ;

sv_packed_dimension_list
   : sv_packed_dimension sv_packed_dimension_list?
   ;

sv_packed_dimension
   : '[' sv_constant_range ']'
   ;

sv_variable_dimension
   : sv_unpacked_dimension
   ;
```

6.0.25. A.2.6 Function declarations

```
sv_function_data_type_or_implicit
   : sv_data_type_or_void
   | sv_implicit_data_type
   ;

sv_function_declaration
   : 'function' sv_function_body_declaration
   ;

sv_function_body_declaration
   : sv_function_body_declaration_item_implicit
   | sv_function_body_declaration_item
   | sv_function_body_declaration_port_implicit
   | sv_function_body_declaration_port
   ;
```

```
sv_function_body_declaration_item_implicit
   : sv_tf_identifier ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endfunction'
   ;

sv_function_body_declaration_item
   : sv_function_data_type_or_implicit
   sv_tf_identifier ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endfunction'
   ;

sv_function_body_declaration_port_implicit
   : sv_tf_identifier
   '(' sv_function_port_item (',' sv_function_port_list)* ')' ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endfunction'
   ;

sv_function_body_declaration_port
   : sv_function_data_type_or_implicit
   sv_tf_identifier
   '(' sv_function_port_item (',' sv_function_port_list)* ')' ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endfunction'
   ;
```

6.0.26. A.2.7 Task declarations

```
sv_task_declaration
   : 'task' sv_task_body_declaration
   ;

sv_task_body_declaration
   : sv_task_body_declaration_item
   | sv_task_body_declaration_port
   ;
```

```
sv_task_body_declaration_item
   : sv_tf_identifier ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endtask'
   ;

sv_task_body_declaration_port
   : sv_tf_identifier
   '(' sv_task_port_item (',' sv_task_port_list)* ')' ';'
   sv_tf_item_declaration*
   sv_statement_or_null
   'endtask'
   ;

sv_tf_item_declaration
   : sv_tf_port_declaration
   | sv_parameter_declaration
   | sv_data_declaration
   ;

sv_function_port_list
   : sv_function_port_item
   | si
   ;

sv_task_port_list
   : sv_task_port_item
   | si
   ;

sv_function_port_item
   : 'input' sv_data_type_or_implicit sv_port_identifier
   | 'input' sv_port_identifier
   ;

sv_task_port_item
   : sv_port_direction sv_data_type_or_implicit sv_port_identifier
   | sv_port_direction sv_port_identifier
   ;

sv_tf_port_direction
   : sv_port_direction
   ;
```

```
sv_tf_port_declaration
   : sv_tf_port_direction sv_data_type_or_implicit sv_list_of_tf_variable_identifiers ';'
   | sv_tf_port_direction sv_list_of_tf_variable_identifiers ';'
   ;
```

6.0.27. A.2.8 Block item declarations

6.0.28. A.2.9 Interface declarations

6.0.29. A.2.10 Assertion declarations

6.0.30. A.2.11 Covergroup declarations

6.0.31. A.2.12 Let declarations

6.0.32. A.3 Primitive instances

6.0.33. A.3.1 Primitive instantiation and instances

6.0.34. A.3.2 Primitive strengths

6.0.35. A.3.3 Primitive terminals

6.0.36. A.3.4 Primitive gate and switch types

6.0.37. A.4 Instantiations

6.0.38. A.4.1 Instantiation

6.0.39. A.4.1.1 Module instantiation

```
sv_module_instantiation
   : sv_module_identifier sv_hierarchical_instance (',' sv_hierarchical_instance )* ';'
   ;

sv_module_param_instantiation
   : sv_module_identifier sv_parameter_value_assignment sv_hierarchical_instance (',' sv_hierarchical_instanc
)* ';'
   ;

sv_parameter_value_assignment
   : '#' '(' sv_list_of_parameter_assignments? ')'
   ;

sv_list_of_parameter_assignments
   : sv_named_list_of_parameter_assignments
   | sv_ordered_list_of_parameter_assignments
   ;
```

```
sv_named_list_of_parameter_assignments
   : sv_named_parameter_assignment (',' sv_named_list_of_parameter_assignments)?
   ;

sv_ordered_list_of_parameter_assignments
   : sv_ordered_parameter_assignment (',' sv_ordered_list_of_parameter_assignments)?
   ;

sv_named_parameter_assignment
   : '.' si '(' sv_expression? ')'
   ;

sv_ordered_parameter_assignment
   : sv_expression
   ;

sv_hierarchical_instance
   : sv_name_of_instance '(' sv_list_of_port_connections? ')'
   | sv_name_of_instance '(' ')'
   ;

sv_name_of_instance
   : sv_instance_identifier
   ;

sv_list_of_port_connections
   : sv_ordered_port_connection (',' sv_ordered_port_connection )*
   | sv_named_port_connection (',' sv_named_port_connection )*
   ;

sv_ordered_port_connection
   : sv_expression
   ;

sv_named_port_connection
   : '.' sv_port_identifier '(' sv_expression? ')'
   ;
```

6.0.40. A.4.1.2 Interface instantiation

6.0.41. A.4.1.3 Program instantiation

6.0.42. A.4.1.4 Checker instantiation

6.0.43. A.4.2 Generated instantiation

```
sv_generate_region
   : 'generate' sv_generate_item* 'endgenerate'
   ;


sv_loop_generate_construct
   : 'for' '(' sv_genvar_initialization ';' sv_genvar_expression ';' sv_genvar_iteration ')' sv_generate_block
   ;


sv_genvar_initialization
   : sv_genvar_identifier '=' sv_constant_expression
   ;


sv_genvar_iteration
   : sv_genvar_iteration_assignment
   | sv_genvar_iteration_inc_or_dec
   ;


sv_genvar_iteration_assignment
   : sv_genvar_identifier sv_assignment_operator sv_genvar_expression
   ;


sv_genvar_iteration_inc_or_dec
   : sv_genvar_identifier inc_or_dec_operator
   ;


sv_conditional_generate_construct
   : sv_if_generate_construct
   | sv_case_generate_construct
   ;
```

```
sv_if_generate_construct
   : 'if' '(' sv_constant_expression ')' sv_generate_block 'else' sv_if_generate_construct
   | 'if' '(' sv_constant_expression ')' sv_generate_block 'else' sv_generate_block
   | 'if' '(' sv_constant_expression ')' sv_generate_block
   ;


sv_case_generate_construct
   : 'case' '(' sv_constant_expression ')' sv_case_generate_item* 'endcase'
   ;


sv_case_generate_item
   : sv_case_generate_item_default_null
   | sv_case_generate_item_default
   | sv_case_generate_item_specific
   ;


sv_case_generate_item_specific
   : sv_constant_expression ':' sv_generate_block
   ;


sv_case_generate_item_default_null
   : 'default' ':' ';'
   ;


sv_case_generate_item_default
   : 'default' ':' sv_generate_block
   ;


sv_generate_block
   : sv_generate_block_seq
   | sv_generate_block_single
   ;


sv_generate_block_seq
   : 'begin' sv_generate_item* 'end'
   ;
```

```
sv_generate_block_single
   : sv_generate_item
   ;


sv_generate_item
   : sv_module_or_generate_item
   ;
```

6.0.44. A.5 UDP declaration and instantiation

6.0.45. A.5.1 UDP declaration

6.0.46. A.5.2 UDP ports

6.0.47. A.5.3 UDP body

6.0.48. A.5.4 UDP instantiation

6.0.49. A.6 Behavioral statements

6.0.50. A.6.1 Continuous assignment and net alias statements

```
sv_continuous_assign
   : sv_assign_list_of_variable_assignments
   ;

sv_assign_list_of_variable_assignments
   : 'assign' sv_list_of_variable_assignments ';'
   ;

sv_list_of_variable_assignments
   : sv_variable_assignment (',' sv_variable_assignment)*
   ;
```

6.0.51. A.6.2 Procedural blocks and assignments

```
sv_always_construct
   : sv_always_keyword sv_statement
   ;

sv_always_keyword
   : sv_always_comb_identifier
   | sv_always_latch_identifier
   | sv_always_ff_identifier
   | sv_always_identifier
```

```
    ;

sv_always_identifier
    : 'always'
    ;

sv_always_comb_identifier
    : 'always_comb'
    ;

sv_always_latch_identifier
    : 'always_latch'
    ;

sv_always_ff_identifier
    : 'always_ff'
    ;

    operator_assignment should be moved to statement_item_operator_assignment !!!

sv_blocking_assignment
    : sv_variable_lvalue sv_blocking_assign_expression
    ;

sv_blocking_assign_expression
    : '=' ('((' si '))')? sv_expression
    ;

sv_operator_assignment
    : sv_variable_lvalue sv_assignment_operator sv_expression
    ;

sv_assignment_operator
    : sv_assignment_operator_assign
    | sv_assignment_operator_plus_assign
    | sv_assignment_operator_minus_assign
    | sv_assignment_operator_mul_assign
    | sv_assignment_operator_not_assign
    | sv_assignment_operator_mod_assign
    | sv_assignment_operator_and_assign
    | sv_assignment_operator_or_assign
    | sv_assignment_operator_xor_assign
    | sv_assignment_operator_shr2_assign
    | sv_assignment_operator_shr3_assign
    | sv_assignment_operator_shl2_assign
```

```
    | sv_assignment_operator_shl3_assign
    ;

sv_assignment_operator_assign
    : '='
    ;

sv_assignment_operator_plus_assign
    : '+='
    ;

sv_assignment_operator_minus_assign
    : '-='
    ;

sv_assignment_operator_mul_assign
    : '*='
    ;

sv_assignment_operator_not_assign
    : '/='
    ;

sv_assignment_operator_mod_assign
    : '%='
    ;

sv_assignment_operator_and_assign
    : '&='
    ;

sv_assignment_operator_or_assign
    : '|='
    ;

sv_assignment_operator_xor_assign
    : '^ '='
    ;

sv_assignment_operator_shr2_assign
    : '«='
    ;

sv_assignment_operator_shr3_assign
    : '«<='
```

```
    ;

sv_assignment_operator_shl2_assign
    : '»='
    ;

sv_assignment_operator_shl3_assign
    : '»>='
    ;

sv_nonblocking_assignment
    : sv_variable_lvalue sv_nonblocking_assign_expression
    ;

sv_nonblocking_assign_expression
    : '<=' sv_expression
    ;

sv_variable_assignment
    : sv_variable_lvalue sv_blocking_assign_expression
    ;
```

6.0.52. A.6.3 Parallel and sequential blocks

```
sv_seq_block
    : 'begin' (':' si)? sv_statement_or_null* 'end' (':' si)?
    ;
```

6.0.53. A.6.4 Statements

```
sv_statement_or_null
    : sv_statement
    | sv_null_statement
    ;

sv_null_statement
    : ';'
    ;

sv_statement
    : sv_statement_item
    ;
```

```
sv_statement_item
   : sv_statement_item_blocking_assignment
   | sv_statement_item_nonblocking_assignment
   | sv_case_statement
   | sv_conditional_statement
   | sv_subroutine_call_statement
   | sv_loop_statement
   | sv_procedural_timing_control_statement
   | sv_seq_block
   ;

sv_statement_item_blocking_assignment
   : sv_blocking_assignment ';'
   ;

sv_statement_item_nonblocking_assignment
   : sv_nonblocking_assignment ';'
   ;

sv_function_statement
   : sv_statement
   ;

sv_function_statement_or_null
   : sv_function_statement
   ;
```

6.0.54. A.6.5 Timing control statements

```
sv_procedural_timing_control_statement
   : sv_procedural_timing_control sv_statement_or_null
   ;

sv_delay_control
   : sv_si_delay_control
   ;

sv_si_delay_control
   : '#' sv_delay_value
   ;

sv_event_control
   : sv_event_control_event_expression
   | sv_event_control_asterisk
```

```
    ;

sv_event_control_event_expression
    : '@' '(' sv_event_expression ')'
    ;

sv_event_control_asterisk
    : '@' '(' '*' ')'
    ;

sv_event_expression
    : sv_edge_identifier sv_expression 'or' sv_event_expression
    | sv_edge_identifier sv_expression
    | '(' sv_event_expression ')'
    | sv_expression 'or' sv_event_expression
    | sv_expression (',' sv_expression)*
    ;


sv_edge_identifier
    : sv_posedge_identifier
    | sv_negedge_identifier
    ;


sv_posedge_identifier
    : 'posedge'
    ;


sv_negedge_identifier
    : 'negedge'
    ;

sv_procedural_timing_control
    : sv_delay_control
    | sv_event_control
    ;
```

6.0.55. A.6.6 Conditional statements

```
sv_conditional_statement
    : sv_priority_if_else_conditional_statement
    | sv_priority_if_conditional_statement
```

```
    | sv_unique_if_else_conditional_statement
    | sv_unique_if_conditional_statement
    : sv_if_else_conditional_statement
    | sv_if_conditional_statement
    ;

sv_priority_if_else_conditional_statement
    : 'priority' 'if' sv_cond_predicate sv_statement_or_null 'else' sv_statement_or_null
    ;

sv_priority_if_conditional_statement
    : 'priority' 'if' sv_cond_predicate sv_statement_or_null
    ;

sv_unique_if_else_conditional_statement
    : 'unique' 'if' sv_cond_predicate sv_statement_or_null 'else' sv_statement_or_null
    ;

sv_unique_if_conditional_statement
    : 'unique' 'if' sv_cond_predicate sv_statement_or_null
    ;

sv_if_else_conditional_statement
    : sv_unique_priority? 'if' sv_cond_predicate sv_statement_or_null 'else' sv_statement_or_null
    ;

sv_if_conditional_statement
    : sv_unique_priority? 'if' sv_cond_predicate sv_statement_or_null
    ;

sv_unique_priority
    : sv_unique_priority_unique
    | sv_unique_priority_unique0
    | sv_unique_priority_priority
    ;

sv_unique_priority_unique
    : 'unique'
    ;

sv_unique_priority_unique0
    : 'unique0'
    ;

sv_unique_priority_priority
```

: ’priority’
;

modified:

sv_cond_predicate
: sv_bracket_expression
| sv_unary_primary
| sv_primary
;

6.0.56. A.6.7 Case statements

sv_case_statement
: sv_unique_priority? sv_case_keyword ’(’ sv_case_expression ’)’ sv_case_item* ’endcase’
;

sv_case_keyword
: sv_case_keyword_casez
| sv_case_keyword_casex
| sv_case_keyword_case
;

sv_case_keyword_case
: ’case’
;

sv_case_keyword_casez
: ’casez’
;

sv_case_keyword_casex
: ’casex’
;

sv_case_expression
: sv_expression
;

sv_case_item
: sv_case_default_item
| sv_case_expression_item
;

```
sv_case_default_item
   : 'default' ':' sv_statement_or_null
   ;

sv_case_expression_item
   : sv_case_item_expression ':' sv_statement_or_null
   ;

sv_case_item_expression
   : sv_expression
   ;
```

6.0.57. A.6.7.1 Patterns

6.0.58. A.6.8 Looping statements

```
sv_loop_statement
   : sv_loop_repeat_statement
   | sv_loop_while_statement
   | sv_loop_do_statement
   | sv_loop_foreach_statement
   | sv_loop_for_statement
   ;

sv_loop_repeat_statement
   : 'repeat' '(' sv_expression ')' sv_statement_or_null
   ;

sv_loop_while_statement
   : 'while' '(' sv_expression ')' sv_statement_or_null
   ;

sv_loop_do_statement
   : 'do' sv_statement_or_null 'while' '(' sv_expression ')' ';'
   ;

sv_loop_foreach_statement
   : 'foreach' '(' sv_ps_or_hierarchical_array_identifier '[' sv_loop_variables ']' ')' sv_statement_or_null
   ;

sv_loop_for_statement
   : 'for' '(' sv_for_initialization ';' sv_expression ';' sv_for_step ')' sv_statement_or_null
   ;
```

```
sv_for_initialization
   : sv_list_of_variable_assignments
   | sv_list_of_variable_declarations
   ;

sv_list_of_variable_declarations
   : sv_for_variable_declaration (',' sv_for_variable_declaration)*
   ;

sv_for_variable_declaration
   : 'int' sv_variable_identifier '=' sv_expression
   ;

sv_for_step
   : sv_for_step_assignment
   ;

sv_for_step_assignment
   : sv_list_of_variable_assignments
   ;

sv_loop_variables
   : sv_index_variable_identifier (',' sv_index_variable_identifier)*
   ;
```

6.0.59. A.6.9 Subroutine call statements

```
sv_subroutine_call_statement
   : sv_tf_call ';'
   ;
```

6.0.60. A.6.10 Assertion statements

6.0.61. A.6.11 Clocking block

6.0.62. A.6.12 Randsequence

6.0.63. A.7 Specify section

6.0.64. A.7.1 Specify block declaration

6.0.65. A.7.2 Specify path declarations

6.0.66. A.7.3 Specify block terminals

6.0.67. A.7.4 Specify path delays

6.0.68. A.7.5 System timing checks

6.0.69. A.7.5.1 System timing check commands

6.0.70. A.7.5.2 System timing check command arguments

6.0.71. A.7.5.3 System timing check event definitions

6.0.72. A.8 Expressions

6.0.73. A.8.1 Concatenations

```
sv_concatenation
   : '{' sv_expression sv_expression_list* '}'
   ;

sv_expression_list
   : ',' sv_expression
   ;

sv_constant_concatenation
   : '{' sv_constant_expression sv_constant_expression_list* '}'
   ;

sv_constant_expression_list
   : ',' sv_constant_expression
   ;

sv_multiple_concatenation
   : '' sv_expression sv_concatenation ''
   ;
```

6.0.74. A.8.2 Subroutine calls

```
sv_constant_function_call
   : sv_function_subroutine_call
   ;

sv_tf_call
   : si '(' sv_list_of_arguments ')'
   ;

sv_subroutine_call
   : sv_tf_call
   ;

sv_function_subroutine_call
   : sv_subroutine_call
   ;

sv_list_of_arguments
   : sv_expression sv_expression_list*
   ;
```

6.0.75. A.8.3 Expressions

```
sv_inc_or_dec_expression
   : inc_or_dec_operator sv_variable_lvalue
   | sv_variable_lvalue inc_or_dec_operator
   ;

sv_conditional_expression
   : sv_cond_predicate '?' sv_expression ':' sv_expression
   ;

sv_constant_expression
   : sv_constant_expression_binary
   | sv_constant_trinary_expression
   | sv_unary_constant_primary
   | sv_constant_primary
   ;

sv_constant_expression_binary
   : sv_constant_expression_binary_primary
   | sv_constant_expression_binary_unary
```

```
    ;

sv_constant_expression_binary_primary
    : sv_constant_primary sv_binary_operator sv_constant_expression
    ;

sv_constant_expression_binary_unary
    : sv_unary_constant_primary sv_binary_operator sv_constant_expression
    ;

sv_unary_constant_primary
    : sv_unary_operator sv_constant_primary
    ;

sv_constant_trinary_expression
    : sv_constant_primary sv_trinary_operator sv_constant_expression ':' sv_constant_expression
    ;

sv_trinary_operator
    : sv_operator_conditional
    ;

sv_operator_conditional
    : '?'
    ;

sv_constant_bracket_mintypmax_expression
    : '(' sv_constant_mintypmax_expression ')'
    ;

sv_constant_mintypmax_expression
    : sv_constant_expression
    ;

sv_constant_param_expression
    : sv_constant_expression
    ;

sv_constant_part_select_range
    : sv_constant_range
    ;

sv_constant_range
    : sv_constant_expression ':' sv_constant_expression
    ;
```

```
sv_expression
   : sv_conditional_expression
   | sv_expression_binary
   | sv_unary_primary
   | sv_inc_or_dec_expression
   | sv_primary
   ;

sv_bracket_expression
   : '(' sv_expression ')'
   ;

sv_expression_binary
   : sv_expression_binary_primary
   | sv_expression_binary_unary
   ;

sv_expression_binary_primary
   : sv_primary sv_binary_operator sv_expression
   ;

sv_expression_binary_unary
   : sv_unary_primary sv_binary_operator sv_expression
   ;

sv_unary_primary
   : sv_unary_operator sv_primary
   ;

sv_mintypmax_expression
   : sv_expression
   ;

sv_part_select_range
   : sv_constant_range
   | sv_indexed_range
   ;

sv_indexed_range
   : sv_constant_primary
   ;


sv_genvar_expression
```

```
  : sv_constant_expression
  ;
```

6.0.76. A.8.4 Primaries

```
sv_constant_primary
  : sv_constant_concatenation
  | sv_constant_function_call
  | sv_primary_literal
  | sv_genvar_identifier
  | sv_constant_bracket_mintypmax_expression
  ;

sv_primary
  : sv_concatenation
  | sv_multiple_concatenation
  | sv_function_subroutine_call
  | sv_bracket_mintypmax_expression
  | sv_hierarchical_identifier_select
  | sv_primary_literal
  ;

sv_bracket_mintypmax_expression
  : '(' sv_mintypmax_expression ')'
  ;

sv_hierarchical_identifier_select
  : sv_hierarchical_identifier sv_select*
  ;

sv_primary_literal
  : sv_number
  ;

sv_bit_select
  : '[' sv_expression ']'
  ;

sv_select
  : sv_select_range
  | sv_bit_select
  ;

sv_select_range
```

```
  : '[' sv_part_select_range ']'
  ;

sv_constant_bit_select_list
  : sv_constant_bit_select sv_constant_bit_select_list?
  ;

sv_constant_bit_select
  : '[' sv_constant_expression ']'
  ;

sv_constant_select
  : sv_constant_bit_select ('[' sv_constant_part_select_range ']')*
  ;
```

6.0.77. A.8.5 Expression left-side values

```
sv_variable_lvalue
  : sv_variable_single_lvalue
  | sv_variable_concat_lvalue
  ;

sv_variable_single_lvalue
  : sv_hierarchical_variable_identifier sv_select*
  ;

sv_variable_concat_lvalue
  : '{' sv_variable_lvalue sv_variable_lvalue_list* '}'
  ;

sv_variable_lvalue_list
  : ',' sv_variable_lvalue
  ;
```

6.0.78. A.8.6 Operators

```
sv_unary_operator
  : operator_plus
  | operator_minus
  | operator_nand
  | operator_nor
  | operator_nxor
  | operator_exclamation
```

```
    | operator_tilde
    | operator_and
    | operator_or
    | operator_xor
    ;

operator_exclamation
    : '!'
    ;

operator_tilde
    : ' '
    ;

sv_binary_operator
    : operator_plus
    | operator_minus
    | operator_pow
    | operator_mul
    | operator_asr
    | operator_asl
    | operator_lsr
    | operator_lsl
    | operator_div
    | operator_mod
    | operator_case_equal
    | operator_case_inequal
    | operator_wildcard_equal
    | operator_wildcard_inequal
    | operator_logical_equal
    | operator_logical_inequal
    | operator_lesser_equal
    | operator_greater_equal
    | operator_land
    | operator_and
    | operator_lor
    | operator_or
    | operator_nxor
    | operator_xor
    | operator_lesser
    | operator_greater
    ;

operator_plus
    : '+'
```

```
    ;

operator_minus
    : '-'
    ;

operator_pow
    : '**'
    ;

operator_mul
    : '*'
    ;

operator_div
    : '/'
    ;

operator_mod
    : '%'
    ;

operator_logical_equal
    : '=='
    ;

operator_logical_inequal
    : '!='
    ;

operator_case_equal
    : '==='
    ;

operator_case_inequal
    : '!=='
    ;

operator_wildcard_equal
    : '==?'
    | '=?='
    ;

operator_wildcard_inequal
    : '!=?'
```

```
    | '!?='
    ;

operator_lesser_equal
    : '<='
    ;

operator_greater_equal
    : '>='
    ;

operator_land
    : '&&'
    ;

operator_and
    : '&'
    ;

operator_nand
    : ' &'
    ;

operator_lor
    : '||'
    ;

operator_or
    : '|'
    ;

operator_nor
    : ' |'
    ;

operator_xor
    : '^'
    ;

operator_nxor
    : ' ^'
    | '^ '
    ;

operator_lsr
```

```
    : '»'
    ;

operator_lsl
    : '«'
    ;

operator_asr
    : '»>'
    ;

operator_asl
    : '«<'
    ;

operator_lesser
    : '<'
    ;

operator_greater
    : '>'
    ;

inc_or_dec_operator
    : inc_operator
    | dec_operator
    ;

inc_operator
    : '++'
    ;

dec_operator
    : '–'
    ;
```

6.0.79. A.8.7 Numbers

```
sv_number
    : sv_integral_number
    ;

    cpp_gx:
```

```
sv_integral_number
   : sv_binary_number
   | sv_octal_number
   | sv_hex_number
   | sv_decimal_number
   ;

sv_decimal_number
   : cpp_unsigned_number
   | sv_unsigned_number sv_decimal_base sv_unsigned_number
   | sv_unsigned_number sv_decimal_base x_digit us_digit*
   | sv_unsigned_number sv_decimal_base z_digit us_digit*
   | sv_decimal_base sv_unsigned_number
   | sv_decimal_base x_digit us_digit*
   | sv_decimal_base z_digit us_digit*
   | sv_unsigned_number
   ;

sv_binary_number
   : sv_unsigned_number sv_binary_base sv_binary_value
   | sv_binary_base sv_binary_value
   ;

sv_octal_number
   : sv_unsigned_number sv_octal_base sv_octal_value
   | sv_octal_base sv_octal_value
   ;

sv_hex_number
   : sv_unsigned_number sv_hex_base sv_hex_value
   | sv_hex_base sv_hex_value
   ;

cpp_unsigned_number_old
   : '0x' sv_hex_value 'U'
   ;

sv_unsigned_number
   : decimal_digit sv_unsigned_number?
   | us_digit sv_unsigned_number?
   ;

sv_binary_value_digit
   : binary_digit sv_binary_value_digit?
   | us_digit sv_binary_value_digit?
```

```
    ;

sv_binary_value
    : [01xXzZ?_] [01xXzZ?_]*
    ;

sv_octal_value_digit
    : octal_digit sv_octal_value_digit?
    | us_digit sv_octal_value_digit?
    ;

sv_octal_value
    : [0-7xXzZ?_] [0-7xXzZ?_]*
    ;

sv_hex_value_digit
    : hex_digit sv_hex_value_digit?
    | us_digit sv_hex_value_digit?
    ;

sv_hex_value
    : [0-9a-fA-FxXzZ?_] [0-9a-fA-FxXzZ?_]*
    ;

sv_decimal_base
    : sv_decimal_unsigned_base
    | sv_decimal_signed_base
    ;

sv_decimal_unsigned_base
    : ”’ ’d’
    | ”’ ’D’
    ;

sv_decimal_signed_base
    : ”’ ’s’ ’d’
    | ”’ ’S’ ’d’
    | ”’ ’s’ ’D’
    | ”’ ’S’ ’D’
    ;

sv_binary_base
    : sv_binary_unsigned_base
    | sv_binary_signed_base
    ;
```

```
sv_binary_unsigned_base
   : ”’ ’b’
   | ”’ ’B’
   ;

sv_binary_signed_base
   : ”’ ’s’ ’b’
   | ”’ ’S’ ’b’
   | ”’ ’s’ ’B’
   | ”’ ’S’ ’B’
   ;

sv_octal_base
   : sv_octal_unsigned_base
   | sv_octal_signed_base
   ;

sv_octal_unsigned_base
   : ”’ ’o’
   | ”’ ’O’
   ;

sv_octal_signed_base
   : ”’ ’s’ ’o’
   | ”’ ’S’ ’o’
   | ”’ ’s’ ’O’
   | ”’ ’S’ ’O’
   ;

sv_hex_base
   : sv_hex_unsigned_base
   | sv_hex_signed_base
   ;

sv_hex_unsigned_base
   : ”’ ’h’
   | ”’ ’H’
   ;

sv_hex_signed_base
   : ”’ ’s’ ’h’
   | ”’ ’S’ ’h’
   | ”’ ’s’ ’H’
   | ”’ ’S’ ’H’
```

```
    ;

decimal_digit
    : 0_digit
    | 1_digit
    | 2_digit
    | 3_digit
    | 4_digit
    | 5_digit
    | 6_digit
    | 7_digit
    | 8_digit
    | 9_digit
    ;

binary_digit
    : x_digit
    | z_digit
    | 0_digit
    | 1_digit
    ;

octal_digit
    : x_digit
    | z_digit
    | 0_digit
    | 1_digit
    | 2_digit
    | 3_digit
    | 4_digit
    | 5_digit
    | 6_digit
    | 7_digit
    ;

hex_digit
    : x_digit
    | z_digit
    | 0_digit
    | 1_digit
    | 2_digit
    | 3_digit
    | 4_digit
    | 5_digit
    | 6_digit
```

```
    | 7_digit
    | 8_digit
    | 9_digit
    | a_digit
    | b_digit
    | c_digit
    | d_digit
    | e_digit
    | f_digit
    ;

x_digit
    : 'x'
    | 'X'
    ;

z_digit
    : 'z'
    | 'Z'
    | '?'
    ;

0_digit
    : '0'
    ;

1_digit
    : '1'
    ;

2_digit
    : '2'
    ;

3_digit
    : '3'
    ;

4_digit
    : '4'
    ;

5_digit
    : '5'
    ;
```

```
6_digit
   : '6'
   ;

7_digit
   : '7'
   ;

8_digit
   : '8'
   ;

9_digit
   : '9'
   ;

a_digit
   : 'a'
   | 'A'
   ;

b_digit
   : 'b'
   | 'B'
   ;

c_digit
   : 'c'
   | 'C'
   ;

d_digit
   : 'd'
   | 'D'
   ;

e_digit
   : 'e'
   | 'E'
   ;

f_digit
   : 'f'
   | 'F'
```

;

us_digit
: '_'
;

6.0.80. A.8.8 Strings

6.0.81. A.9 General

6.0.82. A.9.1 Attributes

sv_attribute_instance
: '(*' sv_attr_spec sv_list_attr_spec* '*)'
;

sv_attr_spec
: si ('=' sv_constant_expression)*
;

sv_list_attr_spec
: ',' sv_attr_spec
;

6.0.83. A.9.2 Comments

6.0.84. A.9.3 Identifiers

sv_class_identifier
: si
;

sv_tf_identifier
: si
;

sv_genvar_identifier
: si
;

sv_hierarchical_array_identifier
: sv_hierarchical_identifier

```
    ;

sv_hierarchical_identifier
    : si sv_hierarchical_identifier_list*
    ;

sv_hierarchical_identifier_list
    : sv_constant_bit_select_list? '.' si
    ;

sv_hierarchical_parameter_identifier
    : sv_hierarchical_identifier ('.' sv_hierarchical_parameter_identifier)?
    ;

sv_hierarchical_variable_identifier
    : sv_hierarchical_identifier
    ;

sv_index_variable_identifier
    : si
    ;

sv_instance_identifier
    : si
    ;

sv_module_identifier
    : si
    ;

sv_net_identifier
    : si
    ;

sv_port_identifier
    : si
    ;

sv_ps_or_hierarchical_array_identifier
    : sv_hierarchical_array_identifier
    ;

sv_type_identifier
    : si
    ;
```

```
sv_variable_identifier
   : si
   ;
```

6.0.85. A.9.4 White space

6.0.86. A.10 Footnotes (normative)

```
hierarchical_name
   : si ('.' si)*
   ;

si
   : [a-zA-Z_] [a-zA-Z0-9_]*
   ;

list_of_si
   : ',' si
   ;

simple_identifier
   : [a-zA-Z_] [a-zA-Z0-9_]*
   ;

usi
   : [a-zA-Z0-9_] [a-zA-Z0-9_]*
   ;

list_of_usi
   : ',' usi
   ;

semikolon
   : ';'
   ;
```

7. VHDL extensions

To be published.

8. C extensions

To be published.

9. License

PDVL is licensed under the Apache License, Version 2.0 (the “License”).