Peter the Great St. Petersburg Polytechnic University

Institute of Computer Science and Cybersecurity

Graduate School of Computer Technologies and Information Systems

**Lecture: SystemVerilog**

Subject: Automation of discrete device design (in English)

Completed by student of group 5130901/10101 Surname X.X.

(signature)

Lecturer \_\_\_\_\_\_\_\_\_\_\_\_\_ Antonov A.P.

(signature)

Saint Petersburg

2023

**Оглавление**

[1. SystemVerilog with the Quartus II Software 4](#_Toc159621268)

[1.1. Objectives 4](#_Toc159621269)

[1.2. SystemVerilog 5](#_Toc159621270)

[2. Data Declaration and Data Types 6](#_Toc159621271)

[2.1. SystemVerilog Variables 6](#_Toc159621272)

[2.2. Data Type – Logic 7](#_Toc159621273)

[2.3. Casting 8](#_Toc159621274)

[2.4. User-Defined Types 9](#_Toc159621275)

[2.5. User-Defined Types – Structures 10](#_Toc159621276)

[2.6. Arrays 11](#_Toc159621277)

[2.7. Array Reduction Methods 12](#_Toc159621278)

[2.8. Un-sized Integer Literals 13](#_Toc159621279)

[2.9. Packages 14](#_Toc159621280)

[2.10. Importing Packages 15](#_Toc159621281)

[2.11. Unsupported Data Types/Features 16](#_Toc159621282)

[3. Procedural Blocks 17](#_Toc159621283)

[3.1. Extending Verilog Procedural Blocks 17](#_Toc159621284)

[3.2. always\_ff 18](#_Toc159621285)

[3.3. always\_comb 19](#_Toc159621286)

[3.4. always\_latch 20](#_Toc159621287)

[4. Procedural Blocks 21](#_Toc159621288)

[4.1. Increment and Decrement Operators 21](#_Toc159621289)

[4.2. Assignment Operators 22](#_Toc159621290)

[4.3. Wild Equality and Inequality Operators 23](#_Toc159621291)

[4.4. Jump Statements 24](#_Toc159621292)

[4.5. Block Names 25](#_Toc159621293)

[4.6. Enhanced Case Statements 26](#_Toc159621294)

[5. State Machine Design 27](#_Toc159621295)

[5.1. State Machine Guidelines 27](#_Toc159621296)

[5.2. State Machine Guidelines – Resets (1) 28](#_Toc159621297)

[5.3. State Machine Guidelines – Resets (2) 29](#_Toc159621298)

[5.4. State Encoding With Enumerated Types 30](#_Toc159621299)

[5.5. Setting the State Machine Encoding 31](#_Toc159621300)

[5.6. State Machine Coding Style 32](#_Toc159621301)

[6. Enhanced Port Connections 33](#_Toc159621302)

[6.1. Module Port Connections 33](#_Toc159621303)

[6.2. Implicit .name Port Connections 34](#_Toc159621304)

[6.3. Implicit .\* Port Connections 35](#_Toc159621305)

[6.4. Passing Data Types Through Module Ports 36](#_Toc159621306)

[6.5. Verilog Bus Module Connections 37](#_Toc159621307)

[6.6. SystemVerilog Bus Module Connections 38](#_Toc159621308)

[6.7. Interfaces 39](#_Toc159621309)

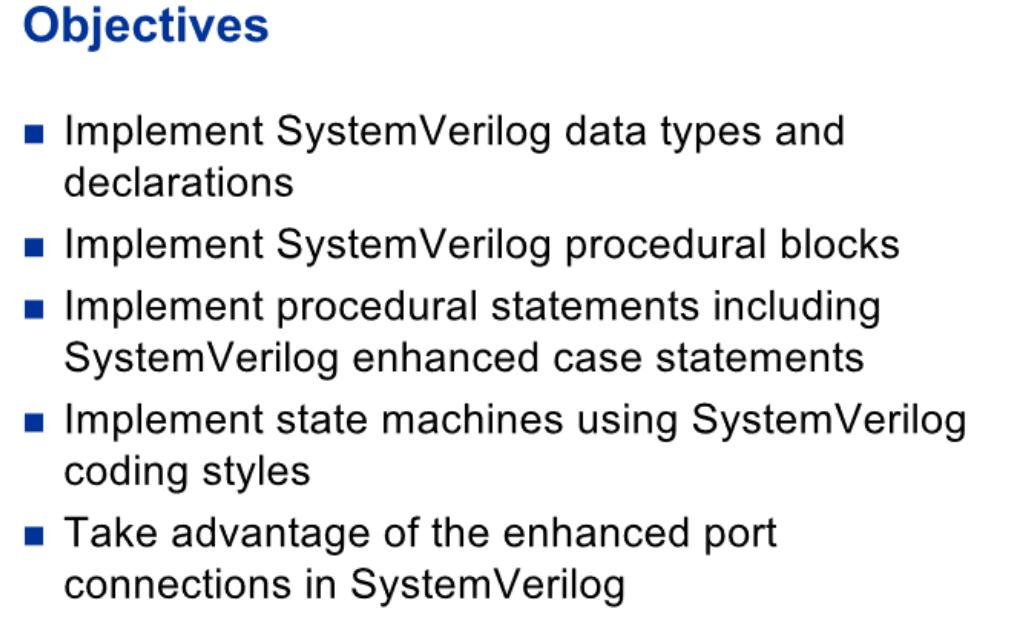
[6.8. Defining an Interface 40](#_Toc159621310)

[6.9. Assigning Interface Connection Views 41](#_Toc159621311)

[6.10. Instantiating an Interface 42](#_Toc159621312)

# SystemVerilog with the Quartus II Software

## Objectives



At the end of this course

You will be able to:

Implement SystemVerilog data types and declarations such as logic, type definitions and enumerated types

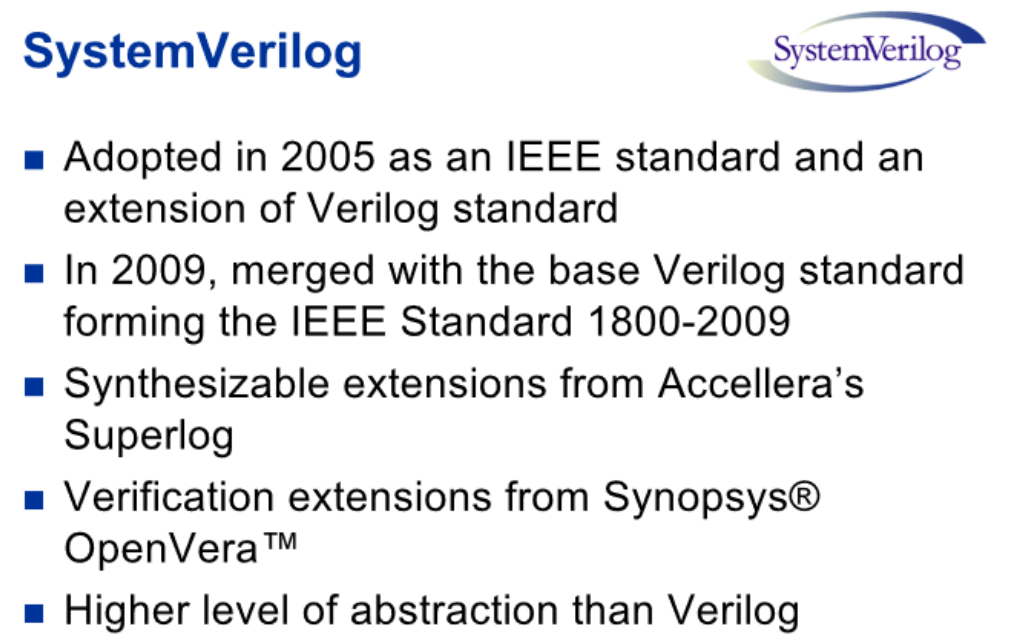
You will also be able to Implement SystemVerilog procedural blocks

Implement procedural statements including SystemVerilog enhanced case statements

Implement state machines using SystemVerilog coding styles.

And you’ll also be able to take advantage of the enhanced port connections in SystemVerilog

## SystemVerilog



SystemVerilog was originally adopted as an IEEE standard in 2005, as an extension of the Verilog language. It provides support for all Verilog constructs. In addition it combines synthesizable constructs from Accellera’s language Superlog and verification constructs from Synopsys’s OpenVera structure.

In 2009, SystemVerilog merged with the base Verilog standard, becoming one standard, the IEEE standard 1800-2009

SystemVerilog provides a higher level of abstraction for modeling and verification than verilog. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage driven verification, and assertion based verification

# Data Declaration and Data Types

This SystemVerilog on-line training covers the synthesizable constructs offered in SystemVerilog that are supported in Altera's Quartus II Software. A basic understanding of the verilog language will aid the student's learning of SystemVerilog. At the end of this training the student should have a good grasp of these SV constructs:

Data declarations and data types

Procedural blocks

Procedural statements

State machine design and

Enhanced port connections

Now let's take a look at Data declarations and date types supported in SV.

## SystemVerilog Variables

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Verilog variables include reg and integer types. All verilog variables use 4 state values. That is variables may take on values of 0, 1, x or z.

SV provides more flexibility in variable and net types by introducing additional data types. To the Verilog reg and integer types, SV adds bit, byte, int, logic and time.

Bit, byte along with 3 forms of int; shortint, int and longint are 2 state data types. They can take on the values of 0 or 1. The default value is 0.

2-state date types simulate faster and consume less memory than 4 state data types. However, 2-state data types in simulation may not accurately model hardware implementation.

The new SV logic and time types are 4 state data types; they can take on the values 0, 1, x, or z. The default value is x.

2-state and 4 state integer types use integer arithmetic and can be signed or unsigned values. Bit, logic reg and time default to unsigned values. All others default to signed.

## Data Type – Logic

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

As seen on the previous slide, SystemVerilog introduces a new data type named logic. The logic data type is similar to the signal data type in VHDL. This new data type is intended to remove confusion associated with the reg data type in Verilog. The reg data type is sometimes mistakenly understood that a register of the given variable name should be created in hardware. However, there is no correlation between using a reg variable and the hardware that is inferred in Verilog. The context of the reg variable determines whether combinatorial or sequential logic is inferred.

For this reason the keyword logic is recommended to be used in place of the keyword reg and wire in most cases.

As a general rule for synthesis, in SV, use net or wire where multiple drivers are required. For everything else use logic or bit.

In this example 2 variables data\_wire and data\_reg are defined as the data type logic. In the assignment of data\_wire, a hardware register is not inferred. In the assignment of data\_reg, a hardware register is inferred. The SV procedural block statements always\_comb and always\_ff will be covered later in this training module.

## Casting

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Verilog is loosely typed so no significant type checking is done at compile time.

Systemverilog has more complex data types so it needs to be stricter about type conversions. So system verilog provides the cast operator, the apostrophe. Casting allows assignment of values to variables that may not be otherwise valid.

The four examples here list various uses of the casting operator. The first example cast the expression x-2 to change its size to 10 bits. Here no truncation warning will be given.

The second expression casts the result of the multiplication of two real numbers to the type int.

The third expression converts the variable x to be a signed variable.

Finally the last expression converts the number 0 to the corresponding value in the enum fruit.

## User-Defined Types

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

SV also supports user defined types and enumerated types.

User defined types allow the user to define their own data type using the typedef syntax. In the top example 2 new data types are defined by the user. The first uint is defined as unsigned integer values. The second user type defined is main\_bus. This type is defined as a logic type of 16 elements.

Enumerated data types limit the variable assignments to a specific set of values. In the bottom example a new data type is defined as boolean. This is an enumerated data type with the possible values of true and false.

A type can be used before it is defined, provided it is first identified as a type by an empty typedef:

## User-Defined Types – Structures

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Structures are also supported in SV.

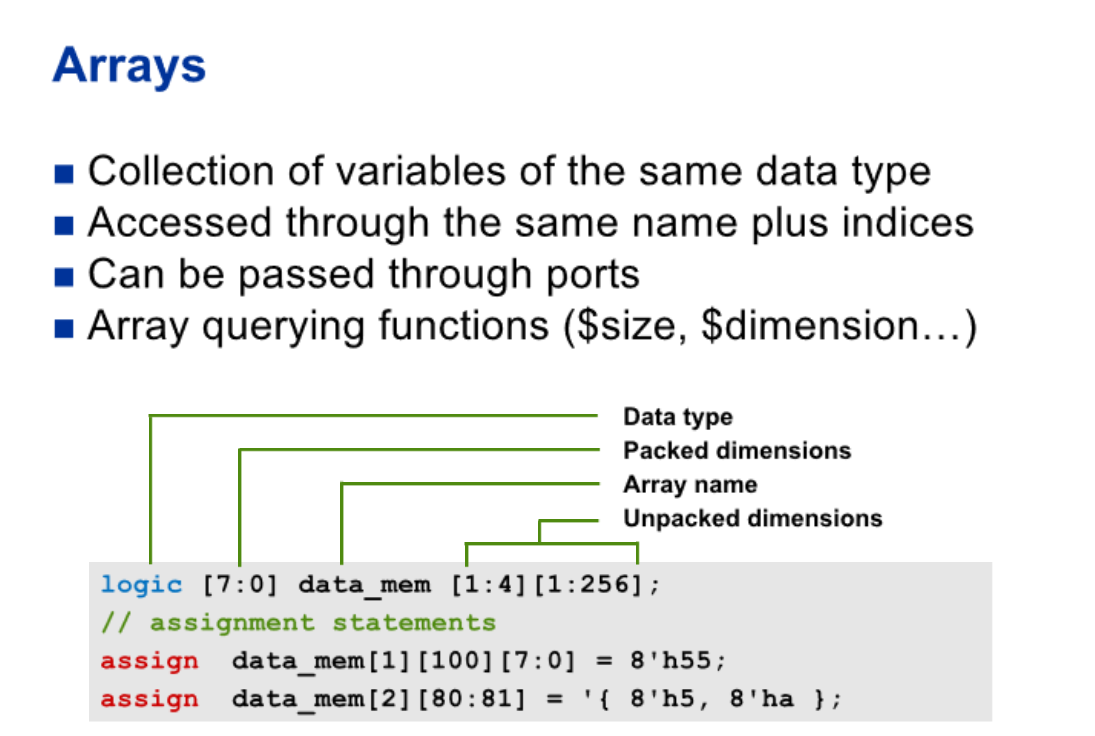
A structure is a collection of variables and/or constants defined as a single name. In this example a structure consisting of a logic data type called even and a logic data type consisting of 8 elements called parity is defined. The name assigned to the structure is par\_struct.

The next line of code shows two structures, par\_in, and par\_out which are defined as the type par\_struct.

Assignments to elements in the structure can be made by position as shown in the assignment to par\_in where the bit even of the structure is assigned a value of 0 and the byte parity is assigned a hex value of 80.

Also individual assignments to elements in the structure can be made as shown in the par\_out assignment statements.

## Arrays



Arrays are a collection of variables of all the same data type. All data types can be declared as arrays in SV. Arrays are accessed through the name given to the array plus indices into the array.

Arrays can be passed through module ports.

Arrays can be either packed or unpacked, packed arrays dimention appear to the left of the array name in the declaration. It can be multidimentional.

Unpacked array dimentions appear to the right of the array name.

In the example code shown an array called data\_mem is defined as a 2 dimensional array of bytes. The byte being a packed array of logic. The definition of the array describes 4 instances of a memory 256 entries deep; perhaps a 4 way cache memory.

After the array is defined in the example code, two assignment statements are made.

In the first assignment statement a single byte entry is updated in the array. Instance number 1 at line 100 is updated to the value of hex 55.

In the second assignment statement 2 adjacent bytes in the second instance of the memory array are updated at entries 80 and 81.

In this assignment the cast operator is used to distinguish the operation from a concatenation. That is the data values are not being concatenated to fit into the 8 bit entry.

Like Verilog memories, the dimensions following the data type, set the size of the packed array. The dimensions following the instance, set the unpacked size. As in Verilog-2001, a comma-separated list of array declarations can be made. All arrays in the list shall have the same data type and the same packed array dimensions.

SystemVerilog uses the term part select to refer to a selection of one or more contiguous bits of a single dimension packed array. This is consistent with the usage of the term part select in Verilog. SystemVerilog uses the term slice to refer to a selection of one or more contiguous elements of an array. Verilog only permits a single element of an array to be selected, and does not have a term for this selection.

SV provides new system functions that can be used to return information on arrays such as $size, $dimension… Please refer to the LRM for more details on these.

## Array Reduction Methods

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

System verilog supports array reduction methods that reduce any unpacked array to a single value.

The supported methods are sum, product, and, or, or xor.

In the example given here, I have an array with three elements in it, the first assignment operator assigns the sum of all elements to the results.

The second operation sets the result to the product of the three elemetents

While the third assign operation here will perform a bitwise and of all the elements.

## Un-sized Integer Literals

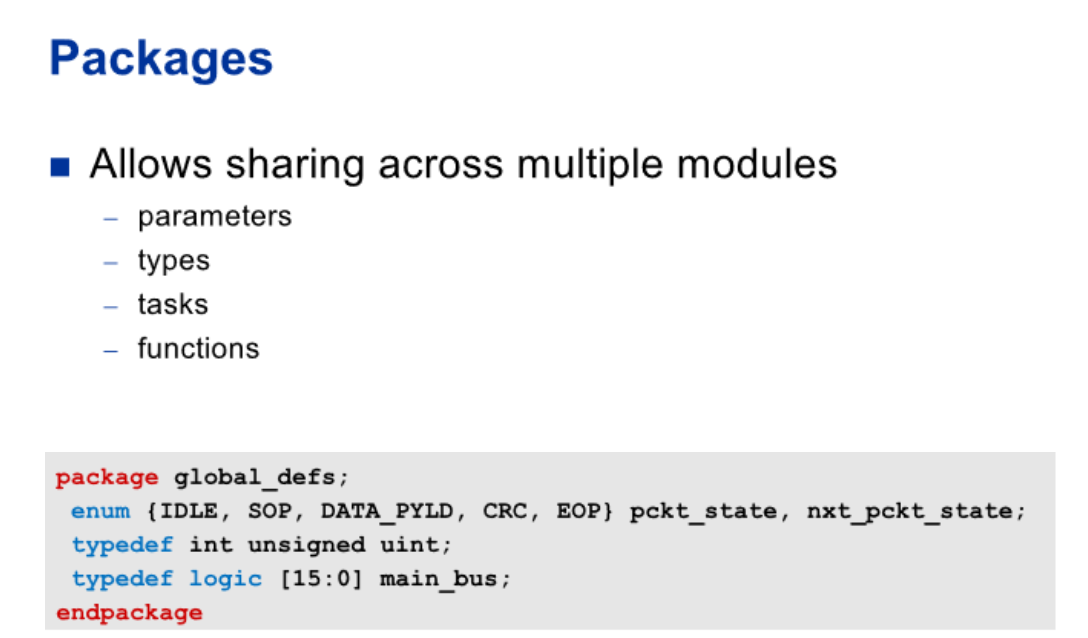
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

SystemVerilog adds the ability to specify un-sized literal single bit values with a preceding apostrophe (') without a base specifier. All bits of the un-sized value are set to the value of the specified bit.

In this example code a parameter is used to define the width of the data bus. At the set condition, all bits of data\_reg regardless of the size of the bus are set to 1 using the un-sized integer literal. The variable could also be assigned to all 0's x's or z's. Verilog did not provide a convenient method for filling an un-sized vector with all ones.

## Packages



Packages are an extension in SV.

Packages allow user defined types, parameters, tasks and functions to be shared across multiple design modules.

In this example a package called global\_defs has been defined. This package defines an enumerated data type and 2 other user defined types.

On the next slide we'll see how we can import and use the content from this package in other modules.

## Importing Packages

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Here are three ways we can use the contents from the package declared in the previous slide.

At the top, in the module definition of main\_ctl there is an input port in\_bus that is of a type main\_bus that is defined in the package.

Using the double colon scope resolution operator allows the type defined in the global\_defs packaged to be referenced.

In the middle example, we import everything in the package global\_defs, after which we can reference main\_bus directly.

Because we used the wild card to import, all the defines in the package can be available.

In the third example, we import in the module header. When importing this way, the global\_defs package is locally imported in to the module only.

## Unsupported Data Types/Features

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Here are some other data types SystemVerilog provides. These are not presently supported in the Quartus II software. For more information on events, unions, classes, and queues, please refer to the SystemVerilog Language Reference Manual.

# Procedural Blocks

We will now look at SV procedural blocks.

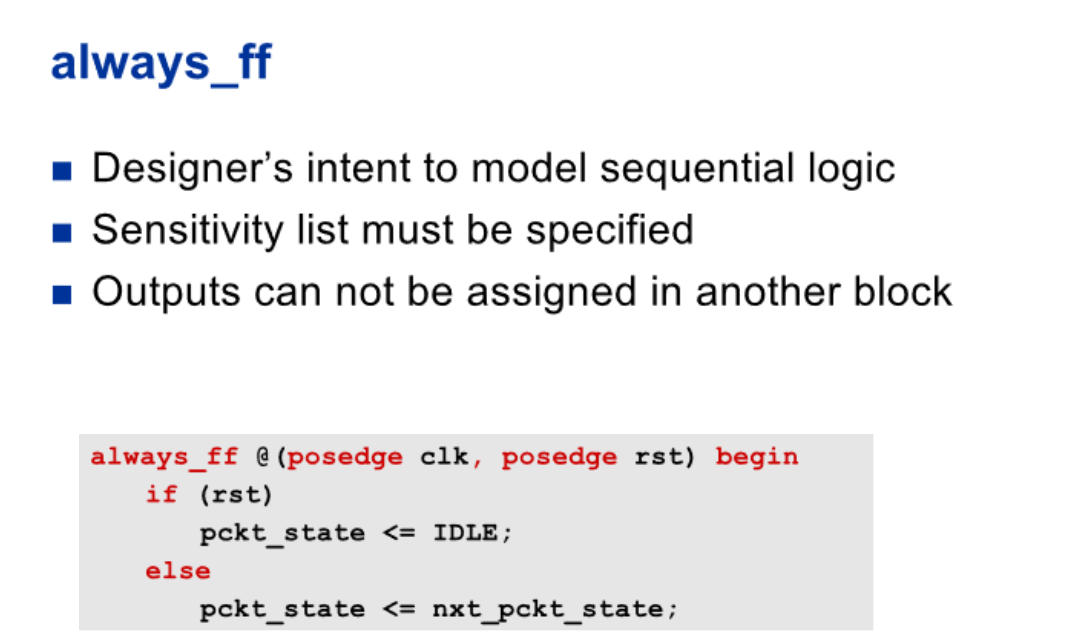
## Extending Verilog Procedural Blocks



Verilog provides a a general purpose procedural block called always. SV extends this general purpose block to indicate the designer’s intent. The always block in verilog can be used to create sequential logic, combinatorial logic or latched logic. SV introduces 3 extensions to the always block: always\_ff, always\_comb and always\_latch. Using these specialized procedural blocks reduces the ambiguity of the verilog general purpose always block. The intent of the designer is made clear.

EDA tool such as synthesizers, simulators, LINT checkers and verification checkers can now perform their task with more accuracy and provide consistency between tools when understanding the designers intent.

## always\_ff

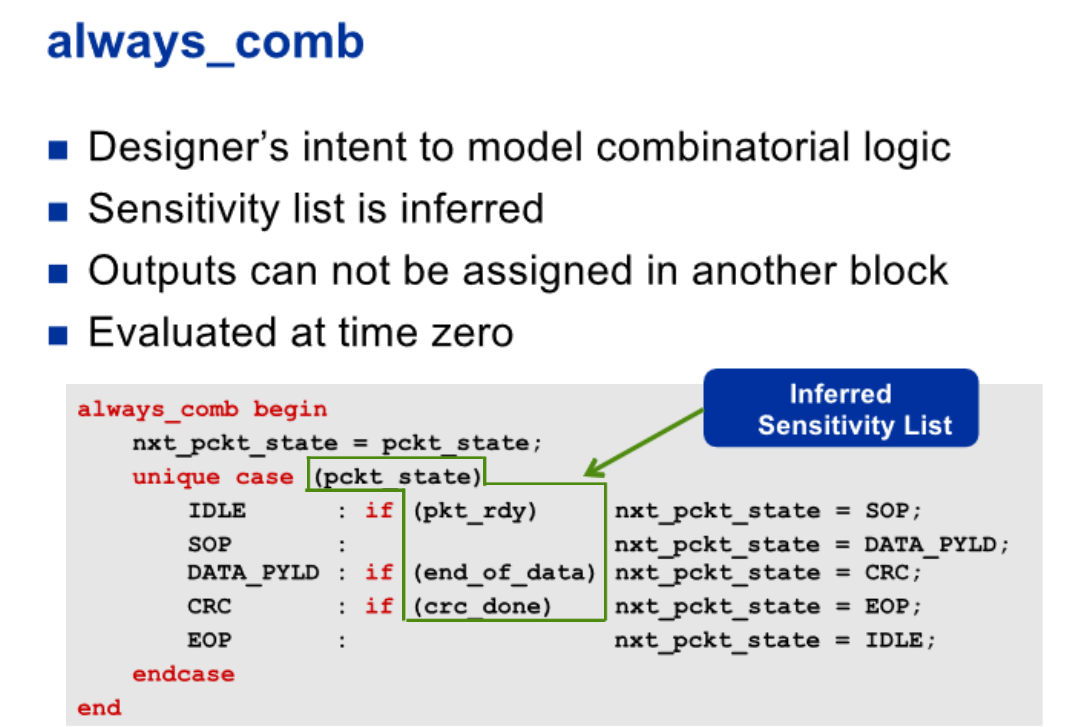


Using the SV always\_ff procedural block, the designer provides software tools with the intent that sequential logic will be modeled.

A sensitivity list is required for the always\_ff block. Each signal in the sensitivity list must be qualified by either a posedge or negedge keyword. This informs the software tools of the polarity of the clocking signal as well as the signal edge on which an asynchronous clear or set is performed.

Outputs from a SV always\_ff procedural block, can not also be assigned in another procedural block.

## always\_comb



Using the always\_comb procedural block the designer provides software tools with the intent that combinatorial logic will be modeled.

The sensitivity list is automatically inferred and includes all signals that are read by the procedural block. See the LRM for complete details on what signals are inferred.

Because of the standard way to infer the sensitivity list, all sw tools will infer the same sensitivity list. This eliminates any ambiguity left by a designer by listing an incorrect or incomplete sensitivity list. In this example pckt\_state, pkt\_rdy, end\_of\_data and crc\_done will be inferred in the sensitivity list. Any signals used in functions called by the always\_comb block will also be included in the sensitivity list.

Any variables on the left hand side of any equations inside the always\_comb procedural block can not be assigned a value in some other procedural block. This prevents usage of the logic in a non combinatorial way such as the creation of an unintentional latch. This also ensures that all sw tools enforce the same modeling rules.

The always\_comb block will automatically trigger at time zero in simulation after all initial blocks and general purpose always blocks have been activated . This ensures that the output logic is consistent with the input signals at time zero.

## always\_latch

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The always\_latch procedural block indicates that the designer's intent is to model latch based logic. This allows software tools to perform checks that would not be performed for combinatorial logic.

like the always\_comb procedural block the sensitivity is inferred. In this example the sensitivity list includes the data enable signals as well as the data input signals. And also like the always\_comb block the left hand assignments can not be assigned in another procedural block.

The always\_latch block is also evaluated in simulation at time zero.

# Procedural Blocks

Now we’ll look at SV statements that go inside procedural blocks.

## Increment and Decrement Operators

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

New in SV are increment and decrement operators. In this example we have an up/down counter. When the up signal is present the counter will increment by one and when the down signal is present, the counter will decrement by one.

These operators behave as blocking assignments. If used in sequential logic it is possible to create a race condition where a second sequential block reading the value of the counter may read the value of counter before or after the value is updated. Therefore to avoid race conditions, these operators should only be used in combinatorial blocks.

## Assignment Operators

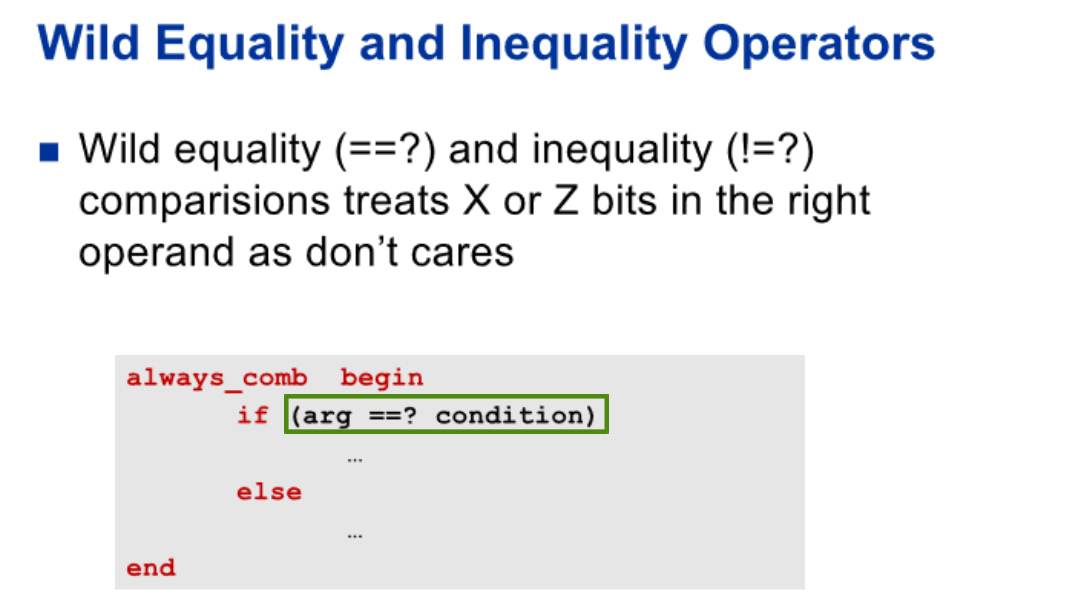
Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

In addition to the auto increment and auto decrement operators, SV also offers these assignment operators. With these assignment statements, a corresponding operation is also done.

For example; the += assignment operator is equivalent to the output signal equals the output signal plus some input signal. Assignment operators support addition, subtraction, multiplication, division, remainder from division, bit wise operations such as and, or, xor and logical and arithmetic shifts right and left.

## Wild Equality and Inequality Operators



on top of the comparison operators supported in verilog.

System verilog addes support for Wild equality and inequality operators that handles comparisons but treats X and Z values in the right operand as wildcards that matches 0,1,Z, or X.

X and Z values in the left operand are NOT treated as wildcards.

These operators compare operands bit for bit and returns a 1 bit result. 0 if false, 1 if true

## Jump Statements

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Jump statements are also supported in SV. These statements can make code more intuitive and concise. Jump statements include break, continue, and return. Break will terminate execution of a loop. The loops is not executed again unless the execution flow encounters the beginning of the loop again as a new statement.

Continue jumps to the end of a loop and executes the loop control, it is not necessary to add begin and end statements as required by the verilog disable statement.

Return can be executed at anytime in a task or function, the task or function will exit immediately.

## Block Names

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Complex code can have nested if then else constructs. Using block names enhances the readability of code. SV allows a matching block label after the end keyword. This allows matching up of begin and end blocks. The names used must match the corresponding begin label.

In this example, data labels are used to identify the begin and end of the cntr\_reg block of code and the ram\_val\_reg block of code.

Also shown in the example, a label write\_registers is used to identify the total block of code in the always\_comb procedural block.

Any non matching block labels will be detected by the Quartus II SW and cause a synthesis error.

## Enhanced Case Statements

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The Verilog standard defines that case statements are evaluated in the order in which they appear. This infers priority in the same way as the if then else construct.

SV provides special modifiers unique and priority to case, casex and casez statements.

The unique modifier allows designers to tell synthesis tools that no priority is implied in the case statement. Statements can be evaluated in parallel. This allows synthesis and simulation tools to optimize out any inferred priority.

The priority modifier indicates that the designer considers that is ok for more than one case selection expressions to be true at the same time and the first statement that is true should have priority.

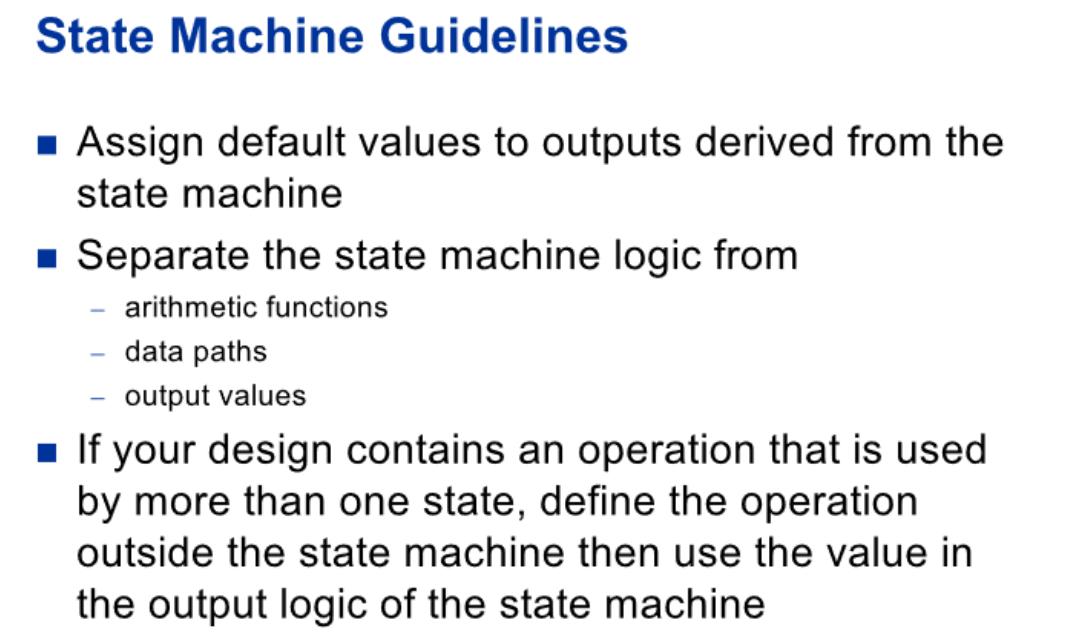
In the top example the unique modifier is used. In this case only one case select will match and no priority should be inferred by simulation or synthesis tools.

In the second example it is possible for more than 1 case selects to be true. The designer by using the priority modifier indicates that the order of the statements indicates which statement has priority.

# State Machine Design

Now we will look at state machine designs using SV constructs.

## State Machine Guidelines



Here are some guidelines that apply to all state machines regardless of the language used to code the state machine.

Assign default values to outputs derived from the state machines.

Separate the state machine logic from arithmetic functions, datapaths, and output values

If your design contains an operation that is used by more than one state, define the operation outside the state machine then use the value in the output logic of the state machine.

## State Machine Guidelines – Resets (1)

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

To ensure state machines come out of reset properly it is recommended that a reset circuit such as this one be used in FPGA designs. This circuit provides an asynchronously asserted and a synchronously deasserted reset signal to any system flip flops that are asynchronously cleared or set in the FPGA.

As shown when the active low async reset signal rst\_async\_ n goes low, this will cause the synchronizing flip flops to clear which will then cause System flip flops connected to the rst\_sync\_n to go to the clear state. Then when the async reset deasserts or goes high, the synchronizing flips flops will clock through the VCC tied to the 1st sync flip flop and remove the reset signal synchronously. This allows the Quartus II SW TimeQuest timing analysis tool to accurately measure the recovery and removal timing to the system flops.

## State Machine Guidelines – Resets (2)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The reset circuit shown on the previous slide can be implemented in SV as shown here. The code instantiates the 2 sync flip flops rst\_s1 and rst\_s2 which are async cleared by the incoming signal rst\_async\_n. Again when the incoming reset signal de-asserts, the sync flip flops will synchronously remove the reset rst\_sync\_n.

## State Encoding With Enumerated Types

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

State variables used in a state machine take advantage of the SV enumerated data type. On this slide there are 4 examples of how the state variables could be defined.

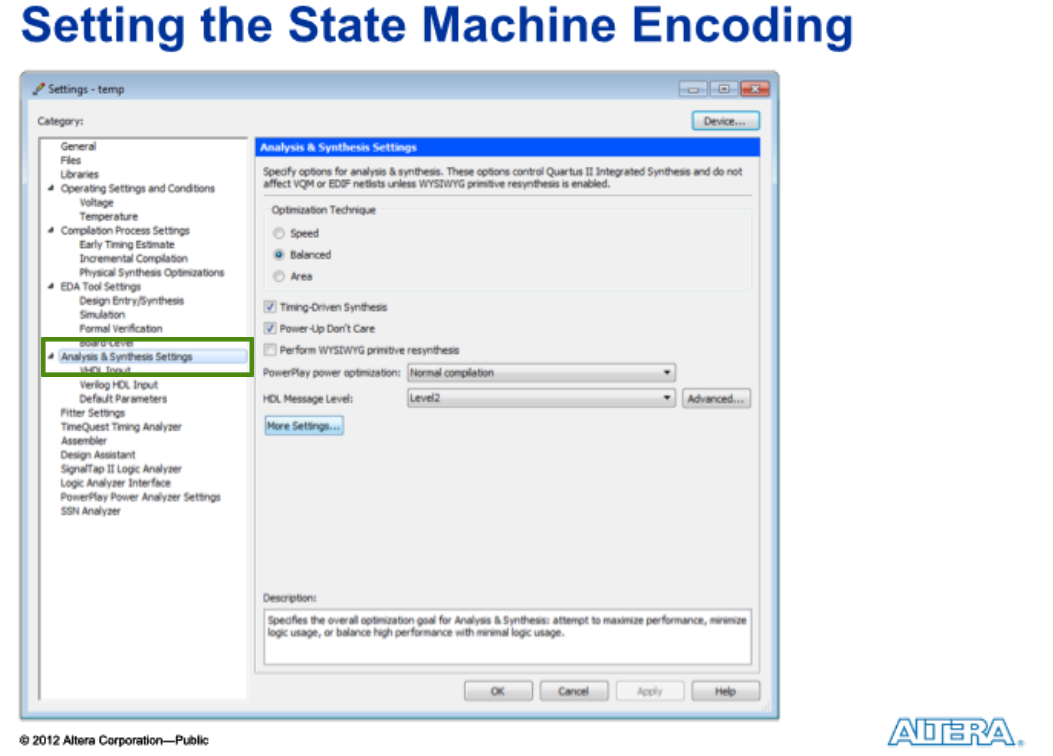
In the first example the default data type int is used. This results in defining a 32 bit signed int data type.

The second example refines this to an unsigned data type. Remember the int data type is a 2 state variable taking only the values of '0' and '1'. While the 2-state data types save on simulation memory and may help simulations to run faster, they may not reflect the hardware in simulation accurately and hide design problems.

In the third example a vectored data type of logic is defined to the state variables.

The fourth example illustrates defining the state machine variables encodings. However the Quartus II software defaults to auto encoding and will ignore these assignments unless the State Machine Processing setting is changed from Auto to User encoded or one of the other options.

## Setting the State Machine Encoding



From the Quartus II SW the state machine variable encoding can be controlled. Under the Assignments menu in the Quartus II software go to settings. Under the Analysis and Synthesis section click More Settings.

Scroll down to State Machine Processing. The settings for State Machine processing default to auto, the tool will decide the state encodings for the state machine variables. The other options are Gray, Johnson, Minimal bits, One-hot, Sequential and User encoding.

## State Machine Coding Style

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Here is an example of a state machine using SV constructs.

The enumerated data type is used to define the state machine variables. The variables pckt\_state and nxt\_pckt\_state may take on the values of Idle, SOP, DATA\_PYLD, CRC and EOP. The enumerated data type uses the data type of logic.

The clocked portion of the state machine uses the specialized SV procedural block always\_ff indicating to synthesis and simulation tools that the intent of the designer is to implement sequential logic.

The next state logic of the state machine uses the specialized procedural block always\_comb to indicate the intent to infer combinatorial logic.

Also note the unique keyword is used to enhance the case statement. Again this tells the synthesis and simulation tools that no priority of the case statement is inferred.

And finally the output assignments are made outside the clocked state and the combinatorial next state blocks.

# Enhanced Port Connections

In this last section of the SV training module we will look at making port connections simpler and less prone to error.

## Module Port Connections



The Verilog language provides two syntax styles for connecting module ports together: ordered port connections and named port connections.

Ordered port connections use the position of the port in the module declaration. The name is not required to be known, the connection is simply made by port position. The requirement to know the port position is a great disadvantage in making port connections. This method is prone to errors and also requires constant editing if changes to the port declaration are made. It is also difficult to understand the intent of the design.

Named port connections required the named port of the module declaration along with the net name that connects to the port. Using this method is less prone to connection errors. The position of the port is not required to be known, only the port name. The disadvantage of named port connections is that it is verbose. Both the port name and the port connection must be listed for each port.

SV introduces three enhancements that simplify netlists: the dot-name convention, the dot-star convention and interfaces.

## Implicit .name Port Connections

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

The SV dot-name convention provides a concise way of making connections independent from knowing the port order.

The dot-name conventions takes advantage that many port connections use the same name of the net as the port connection. The SV syntax is to list the port name and than a wire of the same name is inferred to make the connection. The port name and size must match for a proper connection. When the port name does not match the net name, use the port naming convention of verilog.

In this example the port names clk,rst,addr wr,rd and data\_in and data\_out match nets in name and size. Therefore the connections are inferred. the ports mem\_data1 and mem\_data2 the connections are to nets that are named ram\_data and flash\_data therefore the verilog port naming convention need to be used for those ports.

## Implicit .\* Port Connections

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

SV also provides a dot-star port convention to simplify connecting modules. Using this convention all port names defined on a module declaration that have identical wire names will automatically connect. Use the Verilog port naming convention if names are not identical.

As you can see from this example, making port connections is much simpler and less prone to error. All ports on the module mem\_port that have nets that match the name and size are automatically connected. In this case the ports mem\_data1 and memdata2 are explicitly connected to the ram\_data and flash\_data using the traditional verilog port naming convention.

## Passing Data Types Through Module Ports

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Verilog only permitted wire data type connections on the receiving side of ports. Only wire, reg and integer types could be on the transmitting side of ports. It was also illegal to pass unpacked arrays of any number of dimensions through module ports.

SystemVerilog extends support for additional data types to be passed through module ports. SystemVerilog allows an array of any dimension to be passed through a module port or to a task/function as an argument.

In this example a structure is created and named tlb. Then an output and input structure of type tlb is defined as tlb\_out and tlb\_in.

With SV the structure can be passed through the module port connections.

## Verilog Bus Module Connections

Изображение выглядит как текст, диаграмма, снимок экрана, План

Автоматически созданное описание

This example shows a block diagram with two bus masters and 4 slave interfaces connected through an interconnect fabric.

Using the standard verilog port naming convention the top level looks like this code snippet. This interface connection makes redundant connections on the other bus master as well all the slave interfaces. Using the SystemVerilog dot-name port convention makes this a little less tedious and less prone to error.

However SystemVerilog introduces a new port type: interfaces that will make these connections much easier and less prone to error.

## SystemVerilog Bus Module Connections

Изображение выглядит как текст, снимок экрана, линия, диаграмма

Автоматически созданное описание

With SV interfaces, module connections are greatly simplified. Wires are collected together into one connection between modules.

The connection between modules is defined as an interface and then using the SV dot-name port naming convention, the connections are simply made between modules.

## Interfaces

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Interfaces allows you to bundle signals and other functionalities together. This minimizes wiring code and errors associated with it.

With interfaces, although you can use any signal type you wish, it is highly recommended to use the type logic because it’s compatible with both procedural assignments and continuous assignments. For bidirectional signals and signals with multiple drivers, use the data type wire.

When specifying interfaces as ports of a module you have the option of declaring it as an array which further saves typing.

Lastly, you may wish to include tasks and functions associated with interface, this feature maybe useful for testbenches.

## Defining an Interface

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

SV interfaces are defined using interface declarations like the one shown here. In this example the bus interface signals wr, rd, sel, addr, data\_in, data\_out and bus error are defined as the interface my\_bus. Signals making up the interface can consist of any verilog or SV variable types, nets types or user defined types. Tho as stated in the previous slide, logic is the recommended type.

In this interface declaration two different views of the interface are defined; a master view and a slave view. Using the modport keyword in the interface declaration the direction of ports is defined in regards to whether the module connection will be a master or a slave.

In this example the master will drive the control signals such as read, write, select and address. It will also drive the data\_input bus. The slave interface will drive the data output bus as well as the bus error signal.

## Assigning Interface Connection Views

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

In this example the module fabric will be examined.

The fabric has 2 bus connections cpu\_bus and usb\_bus that use the slave view of my\_bus.

The fabric also has 4 bus connections sram, dram, flash and the usb controller that use the master view of My\_bus.

When referencing signals in the interface, use the interface name(dot) syntax shown here.

This code shows the sram\_mbus(dot)sel signal is an output of a master view of an interface. When the input select signal of the slave view of the cpu bus interface is active along with the corresponding address bits on this bus interface, the select signal is active on the master port to the SRAM.

## Instantiating an Interface

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Now lets look at how interfaces are instantiated and how they are connected to the fabric module.

In my top level module body here, I first declare and instantiate 6 instances of the my\_bus interface. That corresponds to the four slave and two master connections that go through the fabric.

Then when I instantiate the module fabric, for each of the instances, I would specify the port interface name along with the interface instance with the correct view..

Of course this example is not complete, we’d have to connect the other end of these 6 interfaces to the appropriate module for those components.