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 Nepomnyashchiy M.T.

(signature)

Lecturer Antonov A.P.

(signature)

Saint Petersburg

2023

**Оглавление**

[1. SystemVerilog with the Quartus II Software 4](#_Toc159671711)

[2. Data Declaration and Data Types 5](#_Toc159671712)

[2.1. SystemVerilog Variables 5](#_Toc159671713)

[2.2. Data Type – Logic 6](#_Toc159671714)

[2.3. Casting 7](#_Toc159671715)

[2.4. User-Defined Types 8](#_Toc159671716)

[2.5. User-Defined Types – Structures 9](#_Toc159671717)

[2.6. Arrays 10](#_Toc159671718)

[2.7. Array Reduction Methods 11](#_Toc159671719)

[2.8. Un-sized Integer Literals 12](#_Toc159671720)

[2.9. Packages 13](#_Toc159671721)

[2.10. Importing Packages 14](#_Toc159671722)

[3. Procedural Blocks 15](#_Toc159671723)

[3.1. Extending Verilog Procedural Blocks 15](#_Toc159671724)

[3.2. always\_ff 16](#_Toc159671725)

[3.3. always\_comb 17](#_Toc159671726)

[3.4. always\_latch 18](#_Toc159671727)

[4. Procedural Blocks 19](#_Toc159671728)

[4.1. Increment and Decrement Operators 19](#_Toc159671729)

[4.2. Assignment Operators 20](#_Toc159671730)

[4.3. Wild Equality and Inequality Operators 21](#_Toc159671731)

[4.4. Jump Statements 22](#_Toc159671732)

[4.5. Block Names 23](#_Toc159671733)

[4.6. Enhanced Case Statements 24](#_Toc159671734)

[5. State Machine Design 25](#_Toc159671735)

[5.1. State Machine Guidelines 25](#_Toc159671736)

[5.2. State Machine Guidelines – Resets (1) 26](#_Toc159671737)

[5.3. State Machine Guidelines – Resets (2) 27](#_Toc159671738)

[5.4. State Encoding With Enumerated Types 28](#_Toc159671739)

[5.5. Setting the State Machine Encoding 29](#_Toc159671740)

[5.6. State Machine Coding Style 30](#_Toc159671741)

[6. Enhanced Port Connections 31](#_Toc159671742)

[6.1. Module Port Connections 31](#_Toc159671743)

[6.2. Implicit .name Port Connections 32](#_Toc159671744)

[6.3. Implicit .\* Port Connections 33](#_Toc159671745)

[6.4. Passing Data Types Through Module Ports 34](#_Toc159671746)

[6.5. Verilog Bus Module Connections 35](#_Toc159671747)

[6.6. SystemVerilog Bus Module Connections 36](#_Toc159671748)

[6.7. Interfaces 37](#_Toc159671749)

[6.8. Defining an Interface 38](#_Toc159671750)

[6.9. Assigning Interface Connection Views 39](#_Toc159671751)

[6.10. Instantiating an Interface 40](#_Toc159671752)

# SystemVerilog with the Quartus II Software

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**SystemVerilog**, an extension of the **Verilog** language, was adopted as an IEEE standard in **2005**. It combines synthesizable constructs from **Accellera’s Superlog** and verification constructs from **Synopsys’s OpenVera** structure. In **2009**, SystemVerilog merged with the base Verilog standard, resulting in the unified **IEEE 1800-2009** standard. Compared to Verilog, SystemVerilog offers a higher level of abstraction for modeling and verification, facilitating concise hardware descriptions and supporting directed and constrained-random testbench development, coverage-driven verification, and assertion-based verification.

# Data Declaration and Data Types

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 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.

## SystemVerilog Variables

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Verilog Variables: Verilog includes two primary variable types: reg and integer. All Verilog variables operate with four-state values: 0, 1, x (unknown), or z (high-impedance).

SystemVerilog Enhancements: SystemVerilog (SV) introduces additional flexibility by expanding variable and net types. In addition to Verilog’s reg and integer, SV adds the following data types:

* bit
* byte
* int (in three forms: shortint, int, and longint)

Two-State Data Types: These data types are two-state and can only take on values of 0 or 1. The default value is 0. They simulate faster and consume less memory than four-state data types. However, in simulation, they may not precisely model hardware behavior.

New SV Logic and Time Types: SV introduces logic and time as four-state data types. They can take on the values 0, 1, x, or z. The default value for these types is x.

Integer Types: Both two-state and four-state integer types support integer arithmetic and can be either signed or unsigned. bit, logic, and time default to unsigned values, while all others default to signed.

## Data Type – Logic

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

L**ogic** data type bears similarities to the **signal** data type in VHDL. Its purpose is to address the confusion often associated with the **reg** data type in Verilog.

In Verilog, the **reg** type can be misleading. Some mistakenly assume that it implies the creation of a hardware register with the given variable name. However, there is no direct link between using a **reg** variable and the inferred hardware in Verilog. The context of the **reg** variable determines whether it represents combinatorial or sequential logic.

To solve this, **SystemVerilog** introduces the **logic** keyword. In most cases, it is recommended to use **logic** instead of **reg** and **wire**. As a synthesis guideline, opt for **net** or **wire** when multiple drivers are necessary. For other scenarios, choose **logic** or **bit**.

Consider this example: We define two variables, **data\_wire** and **data\_reg**, both as **logic** data types. When assigning a value to **data\_wire**, no hardware register is inferred. However, when assigning to **data\_reg**, a hardware register is indeed implied. Later in this training module, we’ll delve into SystemVerilog procedural block statements like **always\_comb** and **always\_ff**.

## Casting

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

In Verilog, type checking during compilation is minimal due to its loose typing system. However, SystemVerilog takes a more rigorous approach, especially when dealing with complex data types. As a result, it enforces stricter rules for type conversions.

To address this, SystemVerilog introduces the cast operator, represented by the apostrophe ('). This operator allows the assignment of values to variables that might not otherwise be valid.

Here are four examples illustrating the use of the casting operator:

1. Changing Bit Width:
   * We cast the expression x - 2 to change its size to 10 bits. No truncation warning is issued.
2. Real to Integer Conversion:
   * The result of multiplying two real numbers is cast to the integer type.
3. Sign Conversion:
   * The variable x is converted to a signed variable.
4. Enum Mapping:
   * The number 0 is cast to the corresponding value in the enum fruit.

## User-Defined Types

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. User-Defined Types (typedef):
   * SystemVerilog (SV) allows users to define their own custom data types using the typedef syntax.
   * In the top example, two new data types are defined:
     + The first one, uint, represents unsigned integer values.
     + The second user-defined type is named main\_bus, defined as a logic type with 16 elements.
   * These user-defined types enhance code readability and maintainability.
2. Enumerated Data Types:
   * Enumerated types restrict variable assignments to a specific set of values.
   * In the bottom example, a new data type is defined as boolean. It is an enumerated data type with possible values of true and false.
   * Enumerated types help enforce clear semantics and prevent unintended assignments.

\*Type can be used before it is defined, as long as it is first identified as a type by an empty typedef declaration.

## User-Defined Types – Structures

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

In SystemVerilog (SV), structures provide a way to group variables and/or constants under a single name. Here’s a concise explanation:

* Structure Basics:
  + A structure is a composite data type that combines multiple variables or constants.
  + In the example, we define a structure named par\_struct.
  + This structure consists of two components:
    - A logic data type called even.
    - A logic data type with 8 elements named parity.
* Structure Instances:
  + We create two instances of the par\_struct structure: par\_in and par\_out.
  + These instances are defined as the type par\_struct.
* Assignments:
  + Assignments within the structure can be made by position.
  + For instance:
    - In the assignment to par\_in, the bit even of the structure is assigned a value of 0, and the byte parity is assigned a hexadecimal value of 80.
    - Individual assignments to elements in the structure are also possible, as shown in the par\_out assignment statements.

## Arrays

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**What are Arrays?**:

* + Think of arrays as organized shelves for similar items.
  + They hold multiple variables of the same data type.
  + Arrays can be **packed** (with dimensions to the left) or **unpacked** (with dimensions to the right).

**Memory Array**:

* + Imagine a 2D array called data\_mem.
  + It represents a 4-way cache memory with 256 entries.
  + Each entry is like a slot in the memory.

**Assignments**:

* + You can update specific entries in the array.
  + For instance:
    1. Set instance 1 at line 100 to the value **0x55**.
    2. Update adjacent bytes in the second instance at entries 80 and 81.

**Terminology**:

* + **Part Select**: Choosing contiguous bits from a packed array.
  + **Slice**: Selecting contiguous elements from an array.

\*Arrays help us model complex structures in hardware design. SV also provides system functions like $size and $dimension for querying array properties.

## Array Reduction Methods

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**What Are Array Reduction Methods?**:

* + These methods operate on an **unpacked array** to reduce it to a single value.
  + Supported reduction methods include:
    - sum(): Returns the sum of all array elements.
    - product(): Returns the product of all array elements.
    - and(): Performs bitwise AND (&) on all array elements.
    - or(): Performs bitwise OR (|) on all array elements.
    - xor(): Performs logical XOR (^) on all array elements.

## Un-sized Integer Literals

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**What Are Un-Sized Literal Single Bit Values?**:

* + In SV, you can specify single-bit values without explicitly stating their size.
  + These values are represented with a preceding apostrophe (').
  + All bits of the un-sized value are set to the specified bit.

**Example**:

* + Imagine a parameter defining the width of a data bus.
  + At the set condition, regardless of the bus size, all bits of data\_reg are set to **1** using an un-sized integer literal.
  + You could also assign all **0’s**, **x’s**, or **z’s** to the variable.

## Packages

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**Packages in SystemVerilog (SV)**:

* **What Are Packages?**:
  + Packages are like shared toolboxes for the code.
  + They hold user-defined types, parameters, tasks, and functions.
  + We can use them across different parts of your design.
* **Example:** global\_defs **Package**:
  + Imagine a package called global\_defs.
  + It defines an **enumerated data type** and 2 other custom types.

## Importing Packages

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

**Using Package Contents in SystemVerilog (SV)**:

1. **Module Definition**:
   * In the main\_ctl module, we define an input port named in\_bus.
   * This port is of type main\_bus, which is defined in the global\_defs package.
   * We use the **double colon** (::) to reference the type from the package.
2. **Import Everything**:
   * In the middle example, we import **everything** from the global\_defs package.
   * Now we can directly reference main\_bus.
   * The **wildcard** import makes all package defines available.
3. **Local Import**:
   * In the third example, we import the global\_defs package **locally** within the module header.
   * This way, the package is only accessible within this specific module.

# Procedural Blocks

We will now look at SV procedural blocks.

## Extending Verilog Procedural Blocks

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Verilog features a versatile procedural block known as "always". SystemVerilog (SV) builds upon this by enhancing the block to better express the designer's intent. In Verilog, the always block is flexible, supporting sequential, combinatorial, or latched logic. SV introduces three extensions to the always block: always\_ff, always\_comb, and always\_latch. These specialized procedural blocks minimize the ambiguity found in the general-purpose always block of Verilog, making the designer's intention clearer.

This improvement has a significant impact on Electronic Design Automation (EDA) tools, including synthesizers, simulators, LINT checkers, and verification checkers. With these enhancements, these tools can now execute their tasks with greater precision, ensuring consistency between tools in interpreting the designer's intent.

## always\_ff

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The always\_ff block has a few key features:

* **Sensitivity list:** The always\_ff block must have a sensitivity list. This list specifies the signals that the block is sensitive to. When any of the signals in the list change, the block will be re-evaluated.
* **Positive edge triggered:** By default, the always\_ff block is triggered on the positive edge of the clock signal. This means that the block will only be re-evaluated when the clock signal goes from low to high.
* **Outputs cannot be assigned in another block:** The outputs of an always\_ff block cannot be assigned in another procedural block. This means that the output of the always\_ff block must be determined solely by the logic within the block.

The image also shows an example of an always\_ff block that is used to model a state machine. The state machine has two states: IDLE and DATA. The output of the state machine (pckt\_state) is assigned the value of IDLE if the rst signal is high, otherwise it is assigned the value of nxt\_pckt\_state.

Here are some additional things to keep in mind about always\_ff blocks:

* They are used to model flip-flops and other sequential circuits.
* They are a good way to ensure that your design is synthesizable.
* They can be used to model both synchronous and asynchronous circuits.

## always\_comb

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Key features always\_comb block:

* **Inferred sensitivity list:** It’s not needed to specify the signals it's sensitive to. The tool will infer it based on the signals you use inside the block. In the example, these are pckt\_state, pkt\_rdy, end\_of\_data, and crc\_done.
* **Evaluated at time zero:** It's automatically triggered after all initializations to ensure the output logic aligns with the inputs at the beginning.
* **Outputs fixed within the block:** Assignments to the outputs can only happen within the always\_comb block itself, preventing unintended latches from arising.

The example shows an always\_comb block modeling a packet state machine. It takes the current state (pckt\_state) and transitions to the next state (nxt\_pckt\_state) based on the input conditions like pkt\_rdy, end\_of\_data, and crc\_done.

## always\_latch

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The always\_comb block is typically used to describe the behavior of logic circuits that can be implemented using simple logic gates, such as AND, OR, NOT, and XOR gates. The block consists of a sensitivity list and a body of statements. The sensitivity list specifies the signals that the block is sensitive to. Whenever any of the signals in the sensitivity list changes, the body of the block is re-evaluated.

The body of the always\_comb block typically consists of assignments to other signals. These assignments specify how the outputs of the block are calculated based on the values of the inputs.

In the example above, the always\_comb block is sensitive to the signals input1 and input2. Whenever either of these signals changes, the body of the block is re-evaluated, and the value of output\_signal is updated accordingly.

The always\_latch block is similar to the always\_comb block, but it is used to model **latches**, which are circuits that have memory. Latches store the value of their inputs at a specific point in time, and they continue to hold that value until they are reset.

# 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

always\_comb block is used to implement an **up/down counter**. The counter increments by one when the up signal is high, and decrements by one when the down signal is high. The cntr variable stores the current value of the counter.

Here's a breakdown of how the always\_comb block works:

1. cntr = cntr\_value: This line assigns the current value of cntr\_value to cntr. This line is essentially a placeholder and doesn't affect the functionality of the counter.
2. if (up) cntr++: If the up signal is high, this line increments the value of cntr by one using the increment operator (++).
3. else if (down) cntr--: If the up signal is low and the down signal is high, this line decrements the value of cntr by one using the decrement operator (--).

It's important to note that the increment and decrement operators (++ and --) are **blocking assignments** in Verilog. This means that the entire always\_comb block will be re-evaluated whenever the up or down signals change, even if the values of other inputs haven't changed. This can sometimes lead to race conditions, where the value of cntr might be read before or after it's been updated.

To avoid race conditions, it's generally recommended to use non-blocking assignments (<=) inside always\_comb blocks.

## Assignment Operators

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Sure, the image you sent is a Verilog code that describes a combinational logic circuit and its corresponding **always\_comb** block.

The always\_comb block implements an **up/down counter**, which means it increases or decreases a counter value based on two input signals, up and down. If up is high, the counter increments by one. If down is high, the counter decrements by one. The current value of the counter is stored in the cntr variable.

Overall, the always\_comb block defines the relationship between the input signals (up and down) and the output (cntr), ensuring the counter updates immediately whenever the inputs change.

## Wild Equality and Inequality Operators

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The image above has a code snippet that defines wild equality and inequality operators. These operators are used to compare two operands, but they treat X and Z bits in the right operand as wildcards that can match any value (0, 1, Z, or X).

The code snippet also states that X and Z values in the left operand are not treated as wildcards. The operators compare the operands bit by bit and return a 1-bit result, where 0 indicates false and 1 indicates true.

In essence, wild equality and inequality operators provide a way to perform more flexible comparisons in Verilog code. For example, you could use a wild equality operator to check if a certain bit in a register is set to 1, regardless of the values of the other bits.

## Jump Statements

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**Jump Statements**:

* + These are used in programming to control the flow of loops.
  + Three types of jump statements are explained:
    1. **Break**: Terminates a loop and prevents further iterations.
    2. **Continue**: Jumps to the next iteration of the loop.
    3. **Return**: Exits a function or method and returns a value.

**SystemVerilog (SV) s**upports these jump statements and makes code more intuitive and concise.

## Block Names

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Complex code often contains nested if-then-else constructs. To make such code more readable, we can use **block names**. In SV, you can label blocks using matching names after the end keyword.

* **Example**:
  + Code snippet with two blocks: cntr\_reg and ram\_val\_reg.
  + These labels help identify the beginning and end of each block.
  + Additionally, a label like write\_registers identifies a total block of code within an always\_comb procedural block.
* **Synthesis Error Warning**:
  + If block labels don’t match, the Quartus II will detect this and cause a **synthesis error**.

## Enhanced Case Statements

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

In Verilog, the standard behavior of case statements implies priority based on their order. SystemVerilog introduces unique and priority modifiers:

1. Unique Modifier:

* unique indicates no implied priority.
* Allows parallel evaluation for optimization.

2. Priority Modifier:

* priority allows multiple true cases.
* Specifies priority based on statement order.

So, unique avoids implied priority, enabling parallel evaluation, while priority manages cases with simultaneous truth by giving precedence to the first true case.

# State Machine Design

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

## State Machine Guidelines

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Some **state machine uidelines**:

1. **Default Values**: Assign default values to outputs derived from state machines. This helps avoid undefined states and ensures more robust system behavior.
2. **Separate Logic**: Isolate the state machine logic from arithmetic functions, data paths, and output values. This separation simplifies code comprehension and maintenance.
3. **Shared Operations**: If design includes an operation used by multiple states, define that operation outside the state machine. Then use its value in the output logic of the state machine. This modular approach reduces code duplication and promotes better organization.

## State Machine Guidelines – Resets (1)

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

To ensure proper initialization of state machines, it is recommended to use a reset circuit like the one described in **FPGA** designs. This circuit provides an asynchronously asserted and synchronously deasserted reset signal to system flip-flops that are asynchronously cleared or set in the FPGA.

Here’s how it works:

1. When the active low asynchronous reset signal (rst\_async\_n) goes low, it triggers the clearing of synchronizing flip-flops. Consequently, system flip-flops connected to rst\_sync\_n enter the reset state.
2. Subsequently, when the asynchronous reset deasserts (goes high), the synchronizing flip-flops propagate through the VCC tied to the first synchronizing flip-flop, synchronously removing the reset signal.
3. This approach allows the Quartus II SW TimeQuest timing analysis tool to accurately measure recovery and removal timings for the system flip-flops.

## State Machine Guidelines – Resets (2)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

The picture above shows a Verilog code module that generates a reset signal. Here's a how it works:

* **Module Inputs:**
* clk: Clock signal.
* rst\_async\_n: Asynchronous reset signal (active low).
* **Module Outputs:**
* rst\_sync\_n: Synchronous reset signal (active low).
* **Functionality:**
* The module uses two flip-flops (rst\_s1 and rst\_s2) to synchronize the asynchronous reset signal (rst\_async\_n) with the clock (clk).
* When rst\_async\_n is low (active), the flip-flops are cleared to 0, driving the rst\_sync\_n output low (active reset).
* When rst\_async\_n is high (inactive) and the clock (clk) triggers a positive edge, the flip-flops are set to 1, driving the rst\_sync\_n output high (reset inactive).

In essence, this module provides a synchronized reset signal that can be used in other parts of a digital circuit to ensure a clean and stable reset operation.

## State Encoding With Enumerated Types

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**State Variables and Their Definitions:**

* **Default Data Type (Example 1)**: The default data type used for state variables is int, resulting in a 32-bit signed integer. However, this choice may not accurately reflect hardware behavior during simulation.
* **Unsigned Data Type (Example 2)**: To refine this, consider using an unsigned data type. While int is a 2-state variable (only ‘0’ and ‘1’), unsigned types save memory and may speed up simulations. But be cautious, as they might hide design issues.
* **Vectored Data Type (Example 3)**: For more flexibility, define state variables using a vectored data type of logic. This allows handling multiple bits and simplifies code readability.
* **State Machine Variable Encodings (Example 4)**: If you need specific encodings for state machine variables, you can assign custom values.

## Setting the State Machine Encoding

Изображение выглядит как текст, программное обеспечение, веб-страница, Веб-сайт

Автоматически созданное описание

1. Open **Quartus II** software.
2. Navigate to the **Assignments** menu.
3. Click on **Settings**.
4. Under the **Analysis and Synthesis** section, locate and select **More Settings**.
5. Scroll down until you find **State Machine Processing**.
6. By default, the settings are on **Auto**, which means the tool decides the state encodings for state machine variables.
7. However, you can choose from other options such as **Gray**, **Johnson**, **Minimal bits**, **One-hot**, **Sequential**, or **User encoding**.

## State Machine Coding Style

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Here is how the code above works:

1. **State Machine Variables**:
   * The state machine uses an **enumerated data type** to define variables.
   * Two variables, pckt\_state and nxt\_pckt\_state, represent states such as Idle, SOP, DATA\_PYLD, CRC, and EOP.
   * The enumerated data type is based on the logic data type.
2. **Clocked Logic (Sequential)**:
   * The clocked portion of the state machine is implemented using the **specialized SV procedural block always\_ff**.
   * This block indicates to synthesis and simulation tools that the intent is to create sequential logic.
3. **Next State Logic (Combinatorial)**:
   * The next state logic of the state machine uses the **procedural block always\_comb**.
   * This block indicates the intent to infer combinatorial logic.
4. **Unique Keyword and Case Statement**:
   * The **unique** keyword enhances the **case statement**.
   * It informs synthesis and simulation tools that no priority is inferred for the case statement.
5. **Output Assignments**:
   * Output assignments are made **outside** both 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

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. **Ordered Port Connections:**
   * These connections rely on the **position** of the port within the module declaration.
   * The specific **name** of the port isn’t necessary; connections are made based on order.
   * However, this method has drawbacks:
     + It’s **error-prone**, especially when port declarations change.
     + Constant **editing** is required.
     + Design **intent** may not be clear.
2. **Named Port Connections:**
   * Requires both the **named port** from the module declaration and the **net name** connecting to that port.
   * Less prone to errors since you only need to know the **port names**.
   * Disadvantage: It’s **verbose**; both port name and connection must be listed.
3. **SystemVerilog Enhancements:**
   * **Dot-Name Convention**: Simplifies netlists by allowing implicit connections based on matching names.
   * **Dot-Star Convention**: Uses wildcards for matching names.
   * **Interfaces**: A structured way to group related signals.

## Implicit .name Port Connections

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

The code shows how to connect a memory module (mem\_port) to two other modules. The memory module has six ports: clk, rst, addr, data\_in, data\_out, and WI. The other two modules each have a port called mem\_data.

Implicit connections are made using the [...] notation. This notation allows you to specify a list of ports to connect. The compiler automatically connects ports based on their names.

In the example, the mem\_port module is instantiated with two implicit connections. The first connects the data\_out port of the memory module to the mem\_data port of the first module. The second connects the same data\_out port to the mem\_data2 port of the second module.

Here are some benefits of using implicit name port connections:

**Conciseness:** Less code is needed to make connections.

**Order independence:** The order you specify ports in the [...] notation doesn't matter.

**Self-documenting:** Port names show how they are connected.

However, there are also potential drawbacks:

**Readability:** They can be confusing for those unfamiliar with the concept.

**Maintainability:** Changing connections might be trickier.

## Implicit .\* Port Connections

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Implicit name port connections in SystemVerilog let you connect modules without explicitly listing every port. This makes code shorter and clearer.

Imagine connecting a memory module with ports like data\_out to other modules with ports named mem\_data. Instead of writing each connection, you use [...] notation. The compiler automatically matches ports with matching names, like data\_out to mem\_data.

**Benefits:**

* **Concise:** Less code to write.
* **Order-independent:** Doesn't matter which port you list first.
* **Self-documenting:** Port names show how they're connected.

**Drawbacks:**

* **Readability:** Can be confusing for those unfamiliar with the concept.
* **Maintainability:** Changing connections might be trickier.

## Passing Data Types Through Module Ports

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

In Verilog, only wire, reg, and integer types were allowed on the receiving side of module ports. Transmitting side allowed only wire, reg, and integer types as well, and passing unpacked arrays through ports was not allowed.

SystemVerilog expands this capability by allowing a broader range of data types to be transmitted through module ports. Any-dimensional arrays and structures can be passed as arguments to tasks/functions or through module ports.

In the provided example, a structure named tlb is created. Output and input structures, tlb\_out, and tlb\_in, are defined. With SystemVerilog, it becomes possible to pass the structure tlb through module port connections, offering more flexibility in data type support.

## Verilog Bus Module Connections

Изображение выглядит как текст, диаграмма, снимок экрана, План

Автоматически созданное описание

In a block diagram, there are two bus masters and four slave interfaces connected through an interconnect fabric. The Verilog port naming convention for the top-level results in redundant connections to the other bus master and all slave interfaces. The SystemVerilog dot-name port convention helps simplify and reduce errors in these connections.

Moreover, SystemVerilog introduces a new port type called interfaces. This feature significantly streamlines and minimizes errors in making connections compared to traditional Verilog approaches, offering a more efficient and error-resistant way to manage complex interconnections.

## SystemVerilog Bus Module Connections

Изображение выглядит как текст, снимок экрана, линия, диаграмма

Автоматически созданное описание

SystemVerilog interfaces simplify module connections by grouping wires into a single interface between modules. By defining the connection as an interface and using the SystemVerilog dot-name port naming convention, the process of making connections between modules becomes straightforward and more efficient. This approach improves code organization and readability.

## Interfaces

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Interfaces help group signals and functionalities, making wiring code simpler and reducing errors.

It's recommended to use the logic type in interfaces because it works well with both procedural and continuous assignments. For bidirectional signals or those with multiple drivers, the wire data type is suitable.

When using interfaces as module ports, you can declare them as arrays, saving on typing effort.

Additionally, interfaces can include tasks and functions, which can be useful, especially in testbenches.

## Defining an Interface

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

SystemVerilog interfaces are created using interface declarations. In the provided example, the my\_bus interface includes signals like wr, rd, sel, addr, data\_in, data\_out and bus\_error. These signals can be of various Verilog or SystemVerilog variable types, net types, or user-defined types, with logic being the recommended type.

Within the interface declaration, two different views are defined: a master view and a slave view. The modport keyword is used to specify the direction of ports, indicating whether the module connection will act as a master or a slave.

In this specific case, the master view is responsible for driving control signals (read, write, select, address) and the data\_input bus. On the other hand, the slave view drives the data\_output bus and the bus\_error signal. This structuring helps define and organize how interfaces interact in a module connection.

## Assigning Interface Connection Views

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

In this example, we are looking at a module called fabric. This fabric has two bus connections, cpu\_bus and usb\_bus, both using the slave view of the my\_bus interface. Additionally, there are four other bus connections, sram, dram, flash, and the usb controller, using the master view of my\_bus.

When you want to refer to signals within the interface, use the interface name followed by a dot. For instance, in the code, sram\_mbus.sel indicates that the sel signal is an output of the master view of an interface. If the select signal from the slave view of the CPU bus interface is active, along with the corresponding address bits on this bus interface, the select signal becomes active on the master port leading to the SRAM.

## Instantiating an Interface

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Let's see how interfaces are created and linked to the fabric module.

**1. Interface Instances:**

Six instances of the my\_bus interface are created. These act as connection points for data exchange between various parts of the system.

Each instance represents a specific connection, either as a slave (receiving data) or a master (sending data) through the fabric module.

**2. Connecting to the Fabric Module:**

When creating the fabric module, each interface instance is linked with its corresponding port. This creates a pathway for data to flow between the specific interface and the fabric module, enabling communication within the system.