SystemVerilog for RTL

A collection of useful add-ons with a lot of examples
SystemVerilog for RTL

This document describes how to use SystemVerilog for RTL description. SystemVerilog was mainly developed to be used as a verification language similar to E (from Cadence), but it has also some advantages for RTL coding. Verilog 2001/2005 is very old and limited in terms of features, that’s why I want to share some knowledge and experience using SystemVerilog for RTL.

You might think that some parts in the book are pure Verilog 2001, which is correct, because SystemVerilog was developed based on Verilog and some parts are the same.

In order to get a complete overview I will describe everything what you need to write your own RTL code.

**Be careful: Some constructs can only be used for verification and not for RTL description.** There might be updates in the future in order to enable also some additional features, which are already available in VHDL for example.
Agenda

1. Basic SystemVerilog
   a. FlipFlop description
   b. Combination description
Basic System Verilog for RTL

There are 3 new always statements, which differentiate between flipflop, latch and combinational logic. *The advantage of using these constructs is that EDA tools can check the intention of the RTL coding based on these keywords*. That was not possible using in Verilog the always keyword independent of what you want to describe.

**Flipflop description**

A flipflop can be described as usual with an alway @ (posedge clk) ... and so on statement.

There is a predefined statement in SystemVerilog, which checks for correct syntax. Here comes an example with async reset (low active)

Example:

```verilog
always_ff @ (posedge clk,negedge rst_n)
begin
  if (!rst_n) begin
    q <= 1'b0;
    q1 <= 'h0;
  end else begin
    q <= d;
    q1 <= {q1[0+:7],d1};
  end
end
```

This is a simple example for a flipflop. Some things are special. First I used the **logical operator !** for evaluating the reset condition. The result of this logical operation is true or false, which is correct in this context. We are not inverting the reset in the "if" statement.

The next special thing is the description of that **shift register using the concatenation operator**. That saves lines of code. Furthermore you can initialize a vector with 'h0, because the tools are able to fill the whole vector with zeros. If you want to initialize a vector with different value you have to use different descriptions.
Latch description

A latch description is very simple, but sometimes it happens per accident. In this case we use the key word in order to describe a latch.

Example:

```
always_latch begin
  if (enable) begin
    q <= d;
  end
end
```

As you might notice, there is no sensitivity list needed. This is a simple latch, which is transparent when the enable is 1.
**Combinational description**

There are 2 different possibilities to describe combinational logic in Verilog or SystemVerilog. One possibility is to use assign statements with different kind of logical expressions.

e.g.

```verilog
assign tst_s = (count == 4) ? a_s : b_s;
```

The second possibility is to use an always statement. Now in SystemVerilog there is no need to take care anymore about the sensitivity list. The only required key word is `always_comb`.

This key word shows that you want to describe combinational logic

Example:

```verilog
always_comb begin
  vector_s = 'h0;
  if (tst_s) begin
    vector_s[10] = a_s;
  end
end
```

One additional thing you can notice is the way of describing the logic. The first step assigns all zeros to the whole vector. Only if `tst_s` is one bit number 10 is the same as `a_s`. This kind of construct is very common in VHDL and can be used in Verilog or SystemVerilog as well, because they evaluated sequentially. **But be careful; we use blocking assignments, which means the result is evaluated with every = sign. It could happen that you describe an unintended behavior.**
New System Verilog type definitions for RTL

Enumeration Type:

One of the most important enhancements is the enumeration type, which can be used for Finite State Machines. Here is an example how an enumeration type can look like:

```plaintext
typedef enum { idle, write, read, stall, done } fsm_t;
```

This simple example can be enhanced defining values for an enumeration type, which makes it easier, if you want to cast the FSM state to a pre-defined integer value. The following example show, how we can define values for a specific definitions of the enum type. Afterwards the casting to a bit-vector is shown as well.

```plaintext
typedef with values:
typedef enum logic [2:0] { idle=3’d0, write=3’d1, read=3’d2, stall=3’d3, done=3’d4 } fsm_t;
```

There is also a default counting:

```plaintext
typedef enum logic [1:0] { idle, write, read, done } fsm_t;
default counting: idle = 0 ; write = 1; read = 2; done = 3
```

Enumeration methods:

- `first()`: returns first element
- `last()`: returns last element
- `next()`: returns next element
- `prev()`: returns previous element
- `num()`: returns number of elements
- `names()`: returns the value of the variable as string

E.g.

```plaintext
assign fsm_state_one_s = fsm_state_s.first();
assign number_s = fsm_state_s.num();
```

Casting can be done in the following way:

```plaintext
logic [2:0] a;
fsm_t fsm_state_s;

assign a = 3'(fsm_state_s);
assign fsm_state_s = fsm_t'(a);
```
This enumeration type can be used for ports as well, which means you can transport enumeration signals to different modules. This is a straightforward coding style.

typedef enum {idle, write, read, stall, done} fsm_t;

module abc (
    input clk_i,
    input rst_n_i,
    input fsm_t fsm_state_i,
    output fsm_t fsm_outstate_o
);

//-- functional description----

dendmodule
**Struct Type:**

There is an additional type which can be used and shortens the code. Everybody who is used to different software programming languages knows these constructs and likes to use them.

This construct combines different signals in one definition, which has 2 advantages:

1. You can keep logical belonging signals together
2. The module interfaces are much smaller, which makes top-level connections easier

A simple struct looks like the following example:

```verilog
typedef struct {
    logic    rd_wr;
    logic [7:0] data;
    logic [31:0] addr;
} mem_ctrl_st;
```

To access a field of that struct you have to use the hierarchical separator:

```verilog
mem_ctrl_s     mem_signal_s;
logic         rd_wr_signal_s;

assign mem_signal_s.rd_wr = 1'b1;
assign rd_wr_signal_s = mem_signal_s.rd_wr;
```

With this approach you can combine signals in one type.

By the way: this is the same approach as you have it in VHDL using records.

There is a possibility to make it even more convenient to use if you are using the word packed. This gives you the opportunity to treat this struct as one vector, with all the different options to use or manipulate a vector. Here is an example:

```verilog
typedef struct packed {
    logic    rd_wr;
    logic [7:0] data;
    logic [31:0] addr;
} mem_ctrl_st;

mem_ctrl_st  mem_signal_s;
logic [1:0] lsb_addr_s;
logic [7:0]data_s;

initial begin
    mem_signal_s = 'h0; // this sets all signals within struct to zero
end

assign lsb_addr_s = mem_signal_s[0:2];
assign data_s = mem_signal_s.data;
```
**Union Type:**

There is in addition the possibility to combine multiple elements in one memory, which could be used to combine signals even further. Very important to notice is that different signals in a union must have the same ‘type’ or bitwidth. You are not allowed to mix them in one union.

If you want to combine different struct’s, you should use an additional struct. One example is that you combine different struct’s:

```verilog
typedef struct packed {
    logic rd_wr;
    logic [7:0] data;
    logic [31:0] addr;
} mem_ctrl_st;

typedef struct packed {
    logic [7:0] x;
    logic [32:0] y;
} coord_st;

typedef union {
    mem_ctrl_st mem_s;
    logic [40:0] addr_new_s;
    coord_st coord_s;
} comb_ut;

To access value x of union comb_ut you have to use the hierarchical structure:

```verilog
comb_ut union_s;
logic [7:0] x_s;
assign x_s = union_s.mem_s.data;
```

Packed structs are treated as a vector, which means all members in the union comb_ut are of the same size and type.
There is the option to define the union as packed the same way we did for a struct. This gives a similar advantage. The union will be treated as one vector, which enables all vector based manipulations. A packed structure or union stores all members contiguously, with the first member being the left-most (most significant bits) of the storage.

```verilog
typedef struct packed {
    logic [7:0]     x;
    logic [7:0]     y;
} coord_st;

typedef union packed {
    coord_st     coord_a_s;
    coord_st     coord_b_s;
} comb_ut;

comb_ut     coordinate_s;
logic       select_s;

initial begin
    coordinate_s  =  'h0; // this sets all signals within union to zero
end
assign select_s  =  coordinate_s[32];
```
**Keyword: Automatic**

This keyword is similar to the keyword auto in C. It means “re-entrant”.

As soon as a task is declared as automatic, the memory will be allocated, when it is used and de-allocated when it is done.

That is very useful if you want to use functions or tasks in a recursive manner:

```verilog
task automatic do_things;
begin
end
endtask;
```

You could define automatic variables in dynamic or static tasks. Outside these tasks the automatic keyword cannot be used.

```verilog
automatic int count;
```

In general it is not too bad to use these dynamic procedures or variables, because the memory will be freed up again as soon as the task, function or variable is not needed anymore.

It is synthesizable, so don’t be afraid that it might cause some unknown behavior.
Different ways to model delays (not needed for RTL)

There are in principle 2 different possibilities to model wire delays. One is a kind of blocking delay and the other one is a transport delay. Usually delays on wires are working as a transport delay.

Let’s look at the two options to describe a delay:

1. Use an assign statement (blocking delay)
2. Use an always process (transport delay)

Option 1:
Here is a simple example:

    assign #(50ps) delay_sig_s = input_i; // SystemVerilog

The timing unit (e.g. ps) is only allowed in SystemVerilog. In pure Verilog you have relay on the timescale definition and only type the appropriate number.

    `timescale 1ns / 1ps
    assign #(0.05) delay_sig_s = input_i; // Verilog

This assign statement also delays the input_i signal by 50ps.

Now I want to describe the effect of using this delay approach:

If input_i change its state delay_sig_s will change 50ps later. During this 50ps any change on the input_i will not propagate to the delay_sig_s signal. This delay definition acts like a blocking assignment. It acts like a low pass filter where you filter spikes smaller than 50ps.

Option 2:
Here is a simple example:

    // SystemVerilog
    always_comb begin
    delay_sig_s <= #(50ps) input_i;
    end

The Verilog example follows the same rules as described above:

    // Verilog
    `timescale 1ns / 1ps
    always @* begin
    delay_sig_s <= #(0.05) input_i;
    end

Due to the fact that this is a non-blocking assignment the delay will be evaluated every time input_i changes. Every edge will be delayed by the delay even if there more than one input_i change within 50ps. This is a real transport delay.

This version should be used if you want to describe wire or circuit delays, where you know that filter functionality is not realistic.
System Verilog Packages

Packages are well known from VHDL and are now available in SystemVerilog as well. Those packages can be used to define different things:

1. Different types : typedef
2. Local constants : localparam
3. Functions

A package can be imported into a SystemVerilog module very easily. In principle there is also the possibility to import a package into another package, which might be useful. In addition there are no include files and include directories required anymore.

A package description can look like the following example:

```verilog
// ------------------------------------------ Package Definition ------------------------------------------
package test_pack

// ------------------------------------------ Type Definition ------------------------------------------
typedef enum { idle, run, done } fsm_t; // fsm type

// ------------------------------------------ Parameter Definition ------------------------------------------
llocalparam xyz_c = 4'd8; // special constant

// ------------------------------------------ Function Definition ------------------------------------------
function [3:0] func_f;
    input           in_i;
    logic [ 3:0]    reg_s;
    begin
        reg_s   = in_i + 1;
        return reg_s;
    end
endfunction

//====================================================
endpackage
```

This example describes a simple package, where an fsm type, a local parameter and a function is defined. The package usage in a module or another package is defined via the keyword import. There are 2 options to import the package content:

1. Import everything defined in a package
   ```verilog
   import test_pack::*; // the wildcard * stands for everything
   ```
2. Import only a specific part of the package
   ```verilog
   import test_pack::fsm_t; // only the fsm_t type can be used now
   ```
Here is an overall example, which shows the usage in a module.

```systemverilog
//------------------------------- import package global -----------------------------
// is needed, because fsm_t is used in input definition
import test_pack::*;   // import everything from the package

//------------------------------- Module Definition -------------------------------
module test_module

//------------------------------- In- & Output definition -------------------------

( // input clock and reset signals
  input fsm_t state_i,       // state input
  ...
  // outputs
  output ready_o             // ready for change
);

//------------------------------- import package local ---------------------------------
// if not needed in module in/out definition, this import can be used in the module as well
import test_pack::xyz_c;   // only use the locaparam from package

....
endmodule
```

As soon as you work with packages you have to take care about the compile order. First the package must be compiled before you can use it in a module or another package. That means that it is import to define a compile order for all files otherwise your compile might crash.

This is different compared to Verilog and include files. Here it is not required to define a certain order, because all include directories are previously defined and every compile will first build the final source file (include file together with source file) before compiling it.

As you might know an include statement is nothing than putting the code written in an include file into the source file.
System Verilog Arrays

The simplest array is a bit or logic vector. This vector can be used in the same way as in Verilog 2001. There is one change. You can initialize or fill a vector with ‘1’, ‘0’, ‘x’ or ‘z’ using a very simple command.

```
logic [7:0] vector_s;
assign vector_s = '1; // SystemVerilog or '0, 'x, 'z
```

This is very helpful, because especially during initialization there is no need to think about width of the vector. You can just type this simple fill command and every bit will be set accordingly.

There are packed and unpacked arrays defined in SystemVerilog.

Example:

```
logic [7:0] [3:0] array_s [2:0] [13:0];
```

The main difference is that it is guaranteed that the packed dimensions are laid out continuously in the memory. The unpacked dimensions can be placed everywhere in the memory.

The packed dimension:
- can be copied to another packed object
- can be sliced
- are restricted to bit types (bit, logic, int,...)

Unpacked arrays can be handled very efficient from the simulator because the simulator can choose where to put it.

First we look at the order of these packed and unpacked arrays

Using the previous example:

```
logic [7:0] [3:0] array_s [2:0] [13:0];
Order: 3 4 1 2
```

Packed arrays must be used, if you want to describe functions, where you want to return multidimensional arrays. Due to the nature of having these arrays as a kind of vector in a memory you can return it. If the dimensions between packed and unpacked arrays are the same you can copy them easily. If this is not the case you have to use casting methods.

```
logic [9:0] [31:0] array_packed_s;
logic [31:0] unpacked_array_s [9:0];
assign array_packed_s = 320'(unpacked_array_s);
```
In general I would recommend using a loop to copy between different array types. In this case you are sure what you are doing and not relying on a casting method, which might do something, which you don’t expect. Here is the loop example:

```systemverilog
always_comb begin
    for (int i = 0; i < 10; i++) begin
        array_packed_s[i] = unpacked_array_s[i];
    end
end
```

The next very important thing is how to initialize arrays. There are different ways and I am showing here the most elegant way how to do it.

The simplest way using a “for loop” and initialize every single bit or vector individually.

```systemverilog
always_ff @ (posedge clk, negedge rst_n) begin
    if (!rst_n) begin
        for (int i = 0; i < 10; i++) begin
            // try foreach , might be better
            array_unpacked_s[i] = ‘h0;
        end
        end else begin
            ...
        end
    end
end
```

The same functionality can be realized using a “default” statement, like shown in the following example.

```systemverilog
always_ff @ (posedge clk, negedge rst_n) begin
    if (!rst_n) begin
        array_unpacked_s = '{default : ‘h0};
        end else begin
            ...
            end
    end
end
```

This works only if you have a 2 dimensional array. If you have more than 2 dimensions you should think of using loops are directly define the values. By the way : the most common way to initialize an array is:

```systemverilog
array_unpacked_s = '{'h1,'h2, ‘h1,’h2, ‘h1,’h2, ‘h1,’h2, ‘h0,’h4};
```
Now, let’s look at a “foreach loop”. Using this approach is maybe even more appropriate than using a simple for loop. With that approach you loop through the different dimensions, depending on the order.

```
logic [31:0] unpacked_array_s [9:0];

always_comb begin
    // loops 9 downto 0
    foreach (unpacked_array_s[i]) begin
        unpacked_array_s[i] = 'h0;
    end
end
```

Another example:

```
logic [9:0][3:0] packed_array_s;

always_comb begin
    // i loops from 9 downto 0 and j from 3 downto 0
    foreach (packed_array_s[i][j]) begin
        packed_array_s[i][j] = 1'b0;
    end
end
```
**System Verilog: priority & unique statement**

In SystemVerilog you find new key words called **priority** and **unique**.

If either key word is used you will get a run time warning if no condition matches. Example:

```verilog
logic [2:0] a;
unique if (a == 0) begin
    $display("0");
end else if (a == 1) begin
    $display("1");
end else if (a == 2) begin
    $display("2");
end

// values 3,4,5,6,7 will cause a warning

priority if (a == 0) begin
    $display("0");
end else if ((a == 0) || (a == 1)) begin
    $display("1");
end else begin
    $display("all others");
end

// all possibilities are covered, so no warning
```

A “unique if” indicates that there should be no overlap in a series of if else if conditions. This allows the expression to be evaluated in parallel. A “priority if” indicates that a series of if else if conditions are evaluated in ordered list. If a is 0 the priority statement ensures that only the first if statement fires.

The same behavior is true for case, casez or casex statements. A unique keyword defines that there should be no overlap in the case statements and every value must be reachable.

```verilog
logic [2:0] a;
unique case (a)
    0 : begin $display("0"); end
    1 : begin $display("1"); end
    2 : begin $display("2"); end
endcase
// values 3,4,5,6,7 will cause a warning

priority casez (a)
    2'b00? : begin $display("0 or 1"); end
    2'b0?? : begin $display("2 or 3"); end
    default : begin $display("all others"); end
endcase
// all possibilities are covered, so no warning
// if a is 0 or 1 only the first statement is used due to priority
```
The synthesis should follow also the simulation tool. It is absolutely required to run a formal verification tool in order to check RTL code against synthesis result (equivalence check) in order to ensure that the functionality is not changed. Both keywords are comparable to the previous compile directive " // synopsys full_case" or " // synopsys parallel_case".

unique => // synopsys parallel_case
priority => // synopsys full_case

The main difference is that the synopsys statements are only compiler directives and therefore not seen in the simulation. This might be very dangerous because the synthesis result might be different than the RTL code in terms of behavior. The new introduced SystemVerilog keywords are seen by every tool and fire warnings if a violation occurs. In addition this can be also used at if statements, which is not the case for the pragmas. By the way; it was also a good idea in former Verilog designs to avoid using these pragmas because of the possible difference RTL compared to schematic.
In this section a description of some useful special functions are is done. Those functions can be used mainly between “pragma” statements, because most of them are not synthesizable.

Example for switch of synthesis translation:

```verilog
// synopsys translate_off
always_comb begin
    $display(" %m : This is an real instance at \%d", $time);
end
// synopsys translate_on
```

The $display function gives you the possibility to print statements during your simulation. If you want to get the instance name, you should use the %m statement, which prints the instance. $time gives you the actual simulation time as an integer number.

If you want to stop the simulation, if something is happening, which should not happen, you should use assertions. An assertion works the following way:

```verilog
assert ("here comes the statement ,which must be fulfilled") else "do something else"
```

Here is an example for an assert statement:

```verilog
// synopsys translate_off
always_comb begin
    assert (test > 2) else
        $error(" %m : test is smaller or equal 2 at time : \%d", $time);
end
// synopsys translate_on
```

If the assertion fires, the simulation stops and I get the print out:

```
tb.dut.target_inst : test is smaller or equal 2 at time : 2500
```

The $error function can only be used in assert statements. It is not allowed to use it stand alone in the code.
There is the possibility to get the simulation time and use it for calculation. In order to get the simulation time, you should use the function called `$realtime`. It gives back a real number.

```verilog
`timescale 1ns/1ps // timescale “unit”/”resolution”
// synopsys translate_off

real simtime_s;
real new_result_s;

always_ff @ (posedge clk) begin
  if (test == 1) begin
    simtime_s = $realtime;
  end else if (test == 3) begin
    new_result_s = $realtime - simtime_s;
    assert (new_result_s >= 4.0) else $error("%m : the sim time difference is smaller 4.0ns at : %d", $time);
    // the unit of 4.0 is defined in the timescale!!!
  end
end

// synopsys translate_on
```

Let’s see what comes next.
Verilog Primitives:

There are different primitives available, whom are very useful to generate for example scan pattern on top-level. Some tools cannot use HDL descriptions to perform properly. Here are the predefined Verilog primitives:

**Logic Gates:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code></td>
<td>N-Input and gate</td>
<td><code>and(out,in1,in2)</code></td>
</tr>
<tr>
<td><code>nand</code></td>
<td>N-Input nand gate</td>
<td><code>nand(out,in1,in2)</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td>N-Input or gate</td>
<td><code>or(out,in1,in2)</code></td>
</tr>
<tr>
<td><code>nor</code></td>
<td>N-Input nor gate</td>
<td><code>nor(out,in1,in2)</code></td>
</tr>
<tr>
<td><code>xor</code></td>
<td>N-Input xor gate</td>
<td><code>xor(out,in1,in2)</code></td>
</tr>
<tr>
<td><code>xnor</code></td>
<td>N-Input xnor gate</td>
<td><code>xnor(out,in1,in2)</code></td>
</tr>
</tbody>
</table>

**Transmission Gates:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not</code></td>
<td>Inverter</td>
<td><code>not(out,in)</code></td>
</tr>
<tr>
<td><code>buf</code></td>
<td>Buffer</td>
<td><code>buf(out,in)</code></td>
</tr>
<tr>
<td><code>bufif0</code></td>
<td>Tristate buffer (active low enable)</td>
<td><code>bufif0(out,in,cntrl)</code></td>
</tr>
<tr>
<td><code>bufif1</code></td>
<td>Tristate buffer (active high enable)</td>
<td><code>bufif1(out,in,cntrl)</code></td>
</tr>
<tr>
<td><code>notif0</code></td>
<td>Tristate inverter (active low enable)</td>
<td><code>notif0(out,in,cntrl)</code></td>
</tr>
<tr>
<td><code>notif1</code></td>
<td>Tristate inverter (active high enable)</td>
<td><code>notif1(out,in,cntrl)</code></td>
</tr>
</tbody>
</table>

**Switch primitives:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pmos</code></td>
<td>Uni-directional pmos switch</td>
<td><code>pmos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rpmos</code></td>
<td>Resistive pmos switch</td>
<td><code>rpmos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>nmnos</code></td>
<td>Uni-directional nmnos switch</td>
<td><code>nmnos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rnmos</code></td>
<td>Resistive nmnos switch</td>
<td><code>rnmos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>cmos</code></td>
<td>Uni-directional cmos switch</td>
<td><code>cmos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rcmos</code></td>
<td>Resistive cmos switch</td>
<td><code>rcmos(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>tranif0</code></td>
<td>Bi-directional transistor (low)</td>
<td><code>tranif0(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>tranif1</code></td>
<td>Bi-directional transistor (high)</td>
<td><code>tranif1(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rtranif0</code></td>
<td>Resistive transistor (low)</td>
<td><code>rtranif0(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rtranif1</code></td>
<td>Resistive transistor (high)</td>
<td><code>rtranif1(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>tran</code></td>
<td>Bi-directional pass transistor</td>
<td><code>tran(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>rtran</code></td>
<td>Resistive pass transistor</td>
<td><code>rtran(drain,source,gate)</code></td>
</tr>
<tr>
<td><code>pullup</code></td>
<td>Pullup resistor</td>
<td><code>pullup(out)</code></td>
</tr>
<tr>
<td><code>pulldown</code></td>
<td>Pulldown resistor</td>
<td><code>pulldown(out)</code></td>
</tr>
</tbody>
</table>

Some of those are not really needed. Mainly the pullup and pulldown primitives are used for modelling purposes.
Verilog Strength:

Sometimes it might be important to change the strength of a driven signal. Therefore it is required to know the different drive strengths in Verilog.

**Strength Table in the right order:**

<table>
<thead>
<tr>
<th></th>
<th>Inverter</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Buffer</td>
<td>buf(out,in)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Tristate buffer</td>
<td>bufif0(out,in,cntrl)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Tristate buffer</td>
<td>bufif1(out,in,cntrl)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Tristate inverter</td>
<td>notif0(out,in,cntrl)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>notif1(out,in,cntrl)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>