W4. Using structured procedures

Using *always* and *initial blocks* in Verilog language

1. Laboratory objective

Modeling synchronous sequencial designs in Verilog language

2. Theoretical Background

2.1 Introduction to *always* and *initial blocks*

Behavioral modeling is offered through two language structures:

- *always* blocks and
- *initial* blocks

The initial blocks are executed only once, at the beginning of the Verilog simulation but the support provided by the synthesis tools for their integration in the hardware is not consistent (especially for FPGA platforms).

The execution of an always block is triggered by any of the events specified in the block sensitivity list, the list is declared as follows: `always @ ( <sensitivity list> )`.

For a signal in the list, any transition triggers the execution of the always block. If the signal is preceded by a front, posedge, or negedge specifier, then its rising or falling front triggers the execution of the block. If a signal has an edge specifier, all signals in the list must be specified. In the list, multiple events are separated by the reserved word ***or***, or ***,*** (***comma***). If the initial or always block contains more than one statement, these are placed between ***begin*** and ***end***.

2.2 Procedural Assignments

These are assignments executed within an ***always*** or ***initial*** block and, unlike continuous assignments, are evaluated only at the end of the block execution. The left side of a procedural assignment can be:

o a signal declared with type reg,

o an integer variable,

o a real variable,

o a time variable,

o a bit or a part-select selection of the above cases.

Observation:
The signal used on the left side of the procedural assignment must be declared of type reg. If the right side of a procedural assignment has fewer bits than the left, it will be extended by bits of 0.

There are two types of procedural assignments:

- ***Blocking***, which use as an attribution symbol **=** and have the form **<left_hand_side> = <expression>,** respectively
- ***Non-blocking***, which use as an attribution symbol **<=** and have the form **<left_hand_side> <= <expression>**

Important: An initial or always block can contain either only locked or unblocked assignments, not a combination of the two **[Cumm00]**.

For blocking assignments, the evaluation of the right side and the updating of the left side are performed immediately, the execution continuing with the next instruction only later.

For the non-blocking assignment, the right parts of all assignments in the block are evaluated sequentially, but the corresponding right part updates are delayed at the end of the block execution.

2.2.1 Blocking Assignments versus Non-Blocking Assignments

The following code fragments illustrate the two types of assignments:

```
always @ (*) begin

        blk_b = blk_a ;

        blk_c = blk_b ;

        blk_d = blk_c ;

end
```

```
always @ (posedge clk) begin

     nonblk_b <= nonblk_a ;

     nonblk_c <= nonblk_b ;

     nonblk_d <= nonblk_c ;

end
```

Their equivalent physical attributions are presented below:

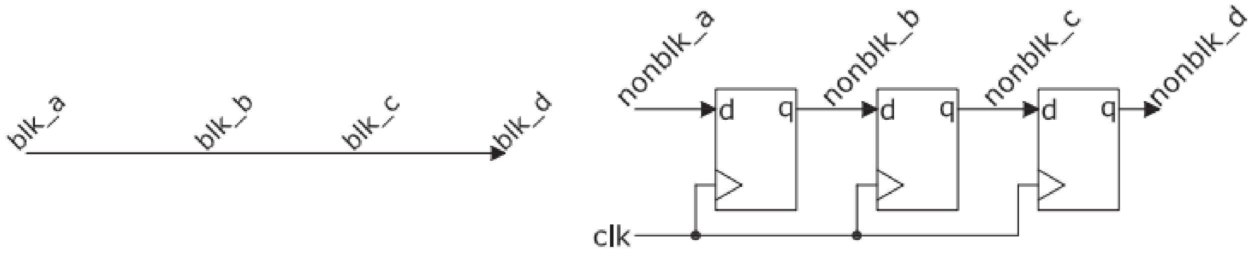

Fig. 1 - Blocking vs non-blocking elements

The left block has combinatorial attribution (using only connecting wires) while the one on the right has synchronous sequential attribution (a shift register).

2.2.2 Using procedural assignments

Synchronous sequential systems triggered on the front are modeled by always blocks having the clock signal in the sensitivity list and using only assignments without blocking. The clock signal will have the front specifier listed.

A potential reset synchronous input is added to the sensitivity list by negedge rst_b or posedge rst, for an active reset line at 0 and 1, respectively.

Note: In this laboratory, signals active at 0 are marked with the suffix _b.

The code fragment below describes a D-type flip-flop with the asynchronous reset input active at 0, rst_b:

```
always @ (posedge clk, negedge rst_b) begin

       if (!rst_b) q <= 1’d0 ;

        else

       q <= d ;

end
```

Sequential systems triggered by level are built with ***always blocks*** containing only non-blocking assignments and having the activation signal specified in the sensitivity list. No front specifier will be used. A possible asynchronous reset input is added to the sensitivity list, also without edge specifier.

The code fragment below describes a T-type latch with an asynchronous reset signal, active at 1, rst:

```
always @ (en, d, rst) begin

        if (rst_b) q <= 1’d0 ;

        else if (en) q <= d ^ q ;

end
```

Combinational structures can be modeled either using continuous assignments or by always blocks in which only locked assignments are used. All signals that appear on the right sides of the assignments in the block or the codings used in the always block will be added to the sensitivity list. Instead of adding all the necessary signals to the sensitivity list, Verilog allows the use of the * symbol as a sensitivity list.

The code fragment below describes a multiplexer with a selection line:

```
always @ (*) begin

        if (sel) o = d1 ;

        else o = d0 ;

end
```

Conditional instructions adopt the following format:

```
if (<condition>)

    <statement_true>;

else

    <statement_false>
```

The else branch is optional. The condition expression is evaluated and if it is different from 0 the statement_true statement is executed, otherwise statement_false is executed, if the branch is included.

If a branch contains several instructions, they will be bounded by the construction ***begin ... end***.

2.3 Study case

Consider an 8-bit parallel load register with asynchronous reset, active at 0 (left) and synchronous reset, active at 1 (right), respectively:

```
module reg8_async_rst_b (                        module reg8_sync_rst (

input clk,                                        input clk,

input rst_b,                                      input rst,

input [7:0] d,                                    input [7:0] d,

output reg [7:0] q );                             output reg [7:0] q );

always @ (posedge clk, negedge rst_b) begin       always @ (posedge clk)

        if (! rst_b) q <= 8'd0 ;                          if (rst)  q <= 8'd0 ;

        else q <= d ;                                     else  q <= d ;

endmodule                                         endmodule
```

On the left, rst_b affects the register regardless of when it is activated, so it is included in the sensitivity list. On the right, rst affects the register only when it is active on the rising edge of the clock cycle and is not included in the sensitivity list.

Important: Any synchronous register entry is treated similarly to the synchronous reset line by evaluating its value in the body of the ***always*** block without including it in the sensitivity list.

2.4 Case Instruction

Represents a multiple decision mechanism that verifies the fit of a selector expression concerning several branches, having the format:

```
case (<expression>)
    <value_1> : <statement>;
     ....
    <value_n> : <statement>;
    default : <statement_default>;
endcase
```

The expression selector is compared with the values by executing the instruction corresponding to the first match. Verification is done in order, starting from value_1. If no match was found and the default case is included, its instruction will be executed.

The casex and case instruction, with the same format, treats undefined bits, respectively, in high impedance, as don't cares bits, which does not influence the selector comparison. The symbol ***?*** can also be used to mark binary positions ***don't care***.

Exercise: Implement a 2-to-4 decoder with enable input and active outputs at 0.

Solution

```
module dec_2x4 (
input [1:0] s,
input e,
output reg [3:0] y
);
always @ ( * )
casez ( { e , s } )
3'b100 : y = 4'b1110 ;
3'b101 : y = 4'b1101 ;
3'b110 : y = 4'b1011 ;
3'b111 : y = 4'b0111 ;
3'b0?? : y = 4'b1111 ;
endcase
endmodule
```

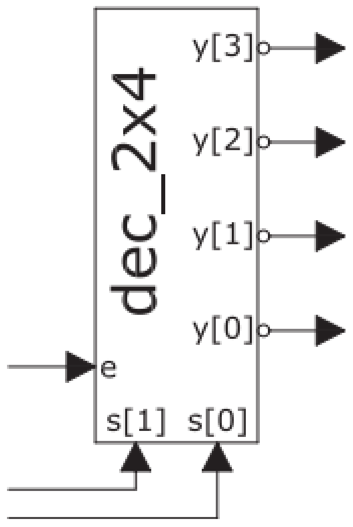

Fig. 2 - 2 to 4 Line Decoder

The last branch masks the s entry by don't care symbols.

2.5 Displaying Simulation Information

System task **$display()** prints data in the simulator console, having the format: **$display(expression_1, … expression_n)**;. Numerical arguments are printed in decimal. In string arguments (enclosed in quotation marks), a number of format specifiers are recognized:

- % b - binary value
- % c - ASCII character, on 8 bits
- % d - decimal value,% 0d specifies the minimum width

- % e,% f and% g - real values
- % h - hexadecimal value
- % m - hierarchical name of the module
- % o - octal value
- % s - string
- % t - simulation time provided by the $ time system call
- % u - unformatted data using 2 values (1 and 0)
- % z - unformatted data using 4 values (1, 0, z and x)

The command **&display()** recognizes the following sequences in string arguments:

- \ n - new line
- \ t - tabular
- \\ - backslash character
- \ ”- character quotes
- %% - percentage character

The command system **$monitor** with the same format as **$display**, prints formatted data whenever one of the arguments changes during the simulation. It will be called only once and its operation can be inhibited by the system ***$monitoroff*** call, respectively it can be reactivated by the ***$monitoron*** call.

2.6 Repetitive instructions

The Verilog language offers 4 repetitive constructions: ***forever***, ***repeat***, ***while,*** and ***for***. The construction forever has the format **forever statement,** and executes the statement statement indefinitely. In this lab, forever will only be used to generate the beat in testbench files. In the following fragment a clock signal with a filling factor of 50% and a period of 100ns is constructed:

```
reg clk ;

initial begin

clk = 1’d0 ;

forever #50 clk = ~clk ;

end
```

The repetitive construction is used in an ***initial block*** where the signal is first initialized and then continuously flipped every 50 ns.

The repeat construction is formatted **repeat (<number_of_times>) statement;** the repeat construct is formatted and executes the instruction a given number of times and is also used in testbenches. The following code prints all numbers between 60 and 63, including, in decimal and binary:

```
reg [5:0] n ;
initial begin
n = 6'd60 ;
repeat (4) begin
    $display ("%d(10)  = %b(2)" , n, n) ;
    n = n+1 ;
    end
end
```

The while construction has the format **while (condition) statement ;** and executes the instruction as long as the *condition* expression is true.

Similar the for construction with the format **for (loop_init ; loop_condition ; loop_update) statement;** executes the instruction as long as the repeat condition is true. This repetitive structure provides facilities for initialization and updating. The code fragment below prints all numbers between 90 and 99, including decimal and binary:

```
reg [6:0] n ;
initial begin
for ( n= 'd90 ; n < 100 ; n = n+1 )
#50 $display ("%d (10) = %b (20)", n, n);
end
```

References

[Stro05] L. Strozek. Verilog Tutorial - Edited for CS141. [Online]. Available: https://wiki.eecs.yorku.ca/course archive/2013-14/ F/3201/ media/verilog-tutorial harvard.pdf (Last accessed 20/07/2016).

[Cumm00] C. Cummings. Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2000SJ NBA.pdf (Last accessed 17/04/2016).