Verilog modeling using the `always` and `initial` blocks

Oprițoiu Flavius
flavius.opritoiu@cs.upt.ro

September 18, 2023

Introduction

Objectives:

- Construct sequential synchronous designs using Verilog

Reading:

- Lukasz Strozek: "Verilog Tutorial - Edited for CS141", Laboratory notes, [Stro05]

Verilog behavioural modeling supported by two structured procedures:

- `always`, and
- `initial`

© 2023 Oprițoiu Flavius. All Rights Reserved.

always and initial blocks

The `initial` blocks are executed only once, at the start of the simulation.

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

Transition of any signal from the sensitivity list triggers block's re-execution. If a signal is proceeded by a `posedge` or `negedge` edge specifier, only the respective edge triggers block's re-execution. One cannot combine in the same sensitivity list signals with and without edge specifiers. Events in the list are separated by the or keyword or by comma. If the `always` or `initial` block contains several statements, those are enclosed between `begin` and `end`.

© 2023 Oprițoiu Flavius. All Rights Reserved.

Procedural assignments

Procedural assignments are issued inside `always` and `initial blocks` and, unlike continuous assignments, are evaluated only during execution of the block they resides in.

The left hand side of a procedural assignments can be:

- a signal declared with the `reg` type,
- an integer variable,
- a real variable,
- a time variable,
- a bit-select or *part-select* of the above, or
- a concatenation of the above

Important: A signal used as the left hand side of a procedural assignment needs to be declared of `reg` type.

If the right hand side of a procedural assignment has fewer bits than the left hand side, it will be extended with 0s in the msb.

© 2023 Oprițoiu Flavius. All Rights Reserved.

Procedural assignments (contd.)

Verilog uses two types of procedural assignments:

- *blocking assignments*, that use symbol =, having the following form <left_hand_side> = <expression> and
- *non-blocking assignments*, that use symbol <=, having the following form <left_hand_side> <= <expression>

Important: For combinational components, the always block uses only blocking assignments!

Important: For sequential synchronous components, the always block uses only non-blocking assignments!

© 2023 Oprițoiu Flavius. All Rights Reserved.

Procedural assignment use cases

Edge triggered sequential synchronous designs (flip-flop components) are modeled by `always` blocks using only non-blocking assignments. The sensitivity list includes the clock signal (*clk*), preceded by an edge specifier, and, possibly, an asynchronous reset signal, also preceded by an edge specifier.

Note: Throughout the practical activities of this class, the active low signals will be marked with suffix **_b**.

The code fragment bellow describes a *D*-type flip-flop with active low reset, *rst_b*:

```
1 always @ (posedge clk, negedge rst_b) begin
2   if (! rst_b) q <= 1'd0;
3   else q <= d;
4 end
```

© 2023 Oprițoiu Flavius. All Rights Reserved.

Procedural assignment use cases (contd.)

Level triggered sequential designs (latch) are constructed using `always` blocks containing only non-blocking assignments. The sensitivity list contains the enable signal and, possible a reset signal. No edge specifier is provided.

The code fragment bellow describes a T-type latch with active high asynchronous reset, *rst*:

```
1 always @ (en, d, rst) begin
2   if (rst) q <= 1'd0;
3   else if (en) q <= d ^ q;
4 end
```

© 2023 Oprițoiu Flavius. All Rights Reserved.

Procedural assignment use cases (contd.)

Combinational designs aside from the continuous assignments (`assign`) can also be modeled using `always` blocks having only blocking assignments. The sensitivity list include all signals whose modification requires re-executing the `always` block. Typically, these sensitivity list signals include all signals appearing in the right hand side of assignments or in condition-type expressions inside the block. Instead of adding all the required signals to the sensitivity list, Verilog permits using the * symbol as sensitivity list.

The code fragment bellow describes a 1-bit selection multiplexer:

```
1 always @ (*) begin
2   if (sel) o = d1;
3   else o = d0;
4 end
```

© 2023 Oprițoiu Flavius. All Rights Reserved.

Conditional statement

The Verilog *conditional* statement has the following format:

```
if (<condition>)
    <statement_true>;
else
    <statement_false>;
```

The `else` branch can be omitted. The `condition` is evaluated and if it is different than 0, *statement_true* is executed, otherwise *statement_else* is executed if the `else` branch is present.

If more than one statement is to be executed on a branch, a `begin` ... `end` construct will enclose those statements.

© 2023 Oprițoiu Flavius. All Rights Reserved.

Parallel load register with reset control line

Parallel load register on 8 bits with asynchronous active low reset (left side) and, respectively, with synchronous active high reset (right side):

```
1   module reg8_async_rst_b (
2     input clk,
3     input rst_b,
4     input [7:0] d,
5     output reg [7:0] q
6   );

8     always @ (posedge clk, negedge rst_b)
9       if (! rst_b) q <= 8'd0;
10      else q <= d;
11  endmodule
```

```
1   module reg8_sync_rst (
2     input clk,
3     input rst,
4     input [7:0] d,
5     output reg [7:0] q
6   );

8     always @ (posedge clk)
9       if (rst) q <= 8'd0;
10      else q <= d;
11  endmodule
```

Important: for sequential synchronous components, the synchronous inputs are not added to the sensitivity list, unlike the asynchronous inputs.

Therby, in the left side , the *rst_b* asynchronous input is included in the sensitivity list whereas the *rst* synchronous input on the right is not.

© 2023 Oprițoiu Flavius. All Rights Reserved.

The case statement

Multi-way decision mechanism matching a selector expression against several branches and having the following format:

```
case (<expression>)
    <case_value_1> : <statement_1>;
    ...
    <case_value_n> : <statement_n>;
    default : <statement_default>;
endcase
```

The *expression* is searched for and if a match is found, the corresponding statement is executed. The search is performed in order from *case_value_1* onwards. If the `default` clause is present and no match occurred its statement will be executed.

If required, binary positions can also be specified not to influence the matching process. These binary positions will use the `?` symbol in the binary expression of the respective *case_value*s.

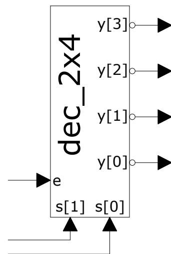
© 2023 Oprițoiu Flavius. All Rights Reserved.

Two-to-four-lines decoder

Exercise: Implement a two-to-four-lines decoder with enable line and active low outputs

Solution:

```
1  module dec_2x4 (
2    input [1:0] s,
3    input e,
4    output reg [3:0] y
5  );
6    always @ ( * )
7      casez ({e, s})
8        3'b100 : y = 4'b1110;
9        3'b101 : y = 4'b1101;
10       3'b110 : y = 4'b1011;
11       3'b111 : y = 4'b0111;
12       3'b0?? : y = 4'b1111;
13     endcase
14 endmodule
```



The last case branch tests input *e* masking the value of the selection lines by using don't care symbols for them (`3'b0??`).

© 2023 Oprițoiu Flavius. All Rights Reserved.

Displaying simulation information

`$display()` writes information to the console, having the syntax: `$display("format", expr_1, ... , expr_n);`. In the *format* string, the following format specifier are recognized:

- `%b` - binary values
- `%c` - characters, 8 bits per character
- `%d` - decimal values
- `%e`, `%f` and `%g` - real values
- `%h` - hexadecimal values
- `%m` - hierarchical module names
- `%o` - octal values
- `%s` - strings, 8 bits per character
- `%t` - simulation time provided by system task `$time`
- `%u` - unformatted data using two values (1 and 0)
- `%z` - unformatted data using four values (1, 0, `z` and `x`)

© 2023 Oprițoiu Flavius. All Rights Reserved.

Displaying simulation information (contd.)

`$display()` recognizes the following sequences in the *format* string:

- `\n` - for new line
- `\t` - for tab character
- `\\` - for backslash
- `\"` - for quote
- `%%` - for percent symbol

`$monitor`, with the same format as `$display`, prints formatted information by writing them whenever any of its arguments changed during simulation.

© 2023 Oprițoiu Flavius. All Rights Reserved.

Loop statements

Verilog provides 4 loop statements: `forever`, `repeat`, `while` and `for`.

The `forever` construct has the format `forever statement;` and executes the provided statement indefinitely. For this laboratory, the statement will be used in testbenches for generating clock signals, as in the code fragment bellow which constructs a 50% duty cycle signal with a period of 100ns:

```
reg clk;
initial begin
  clk = 1'd0;
  forever #50 clk = ~clk;
end
```

The statement is used in an `initial` procedural block, where the clock signal is first initialized and then continuously complemented every 50ns.

© 2023 Oprițoiu Flavius. All Rights Reserved.

Loop statements (contd.)

The `repeat` statement's format is `repeat (<number_of_times>) statement;` and it executes the statement for a fixed number of times. We will use the `repeat` statement in testbenches. The following code prints all number from 60 to 63 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
```

© 2023 Oprițoiu Flavius. All Rights Reserved.

Loop statements (contd.)

The `while` construct has the format `while (condition) statement;` and executes the provided statement for as long as *condition* evaluates to true.

Similarly, the `for` construct has the format `for (loop_init; loop_condition; loop_update) statement;` and executes the provided statement as long the loop condition remains true. The construct has additional loop condition initialization and loop update elements. The code fragment bellow prints all number from 90 to 99 in decimal and binary:

```
reg [6:0] n;
initial begin
  for (n = 'd90; n < 100; n = n+1)
    #50 $display("%d(10) = %b(2)", n, n);
end
```

© 2023 Oprițoiu Flavius. All Rights Reserved.

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).

© 2023 Oprițoiu Flavius. All Rights Reserved.