Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

Politechnika Śląska jako Centrum Nowoczesnego Kształcenia
opartego o badania i innowacje

POWR.03.05.00-IP.08-00-PZ1/17

Projekt współfinansowany przez Unię Europejską ze środków Europejskiego Funduszu Społecznego

Digital Circuits Design

Faculty of Automatic Control, Electronics and Computer Science, Informatics, Bachelor Degree

Lecture 14.

Hardware Description Language - simulation

Verilog

Contents

- Synthesizable constructions - continuation
- Simulation

Procedural assignment – `always`

- `always`
  - Each procedural block is executed concurrently
  - Procedural block is endless loop
  - Synthesis is performed with restrictions
  - Enables description of combinational and sequential circuits
  - `always` blocks cannot be nested

- Event control: `@(<sensitivity_list>)`
  - Any change of signal from the sensitivity list results in „an execution" of the always block

```
@(CLK) Q = D;
@(posedge CLK) Q = D; //0->1, x->1, 0->x
@(negedge CLK) Q = D; //1->0, x->0, 1->x
```

Procedural assignment – `always`

- Example 1

```
always @(posedge CLK or negedge CLR)
begin  //block; { and } are preserved for concatenation
  if(!CLR)
    Q <= 1'b0;
  else
    Q <= D;
end
```

- Example 2

```
always @(a or b or sel)
  if(sel)
    y = a;
  else
    y = b;
```

Procedural assignment – `always`

- Example 1

```
always @(posedge CLK or negedge CLR)
begin
  if(!CLR) //…… reset
    Q <= 1'b0;
  else
    Q <= D;
end
```

- Example 3

```
always @(posedge CLK)
  if(!CLR) //…… reset
    Q <= 1'b0;
  else
    Q <= D;
```

Procedural assignment

- Blocking assignment (=)
  - The expression is evaluated and then assigned to the variable immediately, before execution of the next statement (so „block” the execution of other statements)
- Nonblocking (<=)
  - Expressions are executed concurrently
  - The order does not matter
  - The expression does not „block” the execution of statements in another procedural blocks
- Do not mix blocking and nonblocking assignments
- Use **blocking** assignments for a **combinational** circuit
- Use **nonblocking** assignments for a **sequential** circuit

Procedural assignment

- Blocking (=)

```
always @(posedge CLK)
begin
  if(CLR)
    begin
      Q1 = 4'b0;
      Q2 = 4'b0;
    end
  else
    begin
      Q1 = IN1; //Q2 = Q1;
      Q2 = Q1;  //Q1 = IN1;
    end
end
```

- The order matters!!!

- Nonblocking (<=)

```
always @(posedge CLK)
begin
  if(CLR)
    begin
      Q1 <= 4'b0;
      Q2 <= 4'b0;
    end
  else
    begin
      Q1 <= IN1; //Q2 <= Q1;
      Q2 <= Q1;  //Q1 <= IN1;
    end
end
```

- The order does not matter!!!

Procedural assignment

- Blocking (=)

```
always @(posedge CLK or posedge rst)
      if(rst)
                Q1 = 1'b0;
      else
                Q1 = Q2;
always @(posedge CLK or posedge rst)
      if(rst)
                Q2 = 1'b1;
      else
                Q2 = Q1;
```

  - **Race condition!!!**

- Nonblocking (<=)

```
always @(posedge CLK or posedge rst)
      if(rst)
                Q1 <= 1'b0;
      else
                Q1 <= Q2;
always @(posedge CLK or posedge rst)
      if(rst)
                Q2 <= 1'b1;
      else
                Q2 <= Q1;
```

if-else

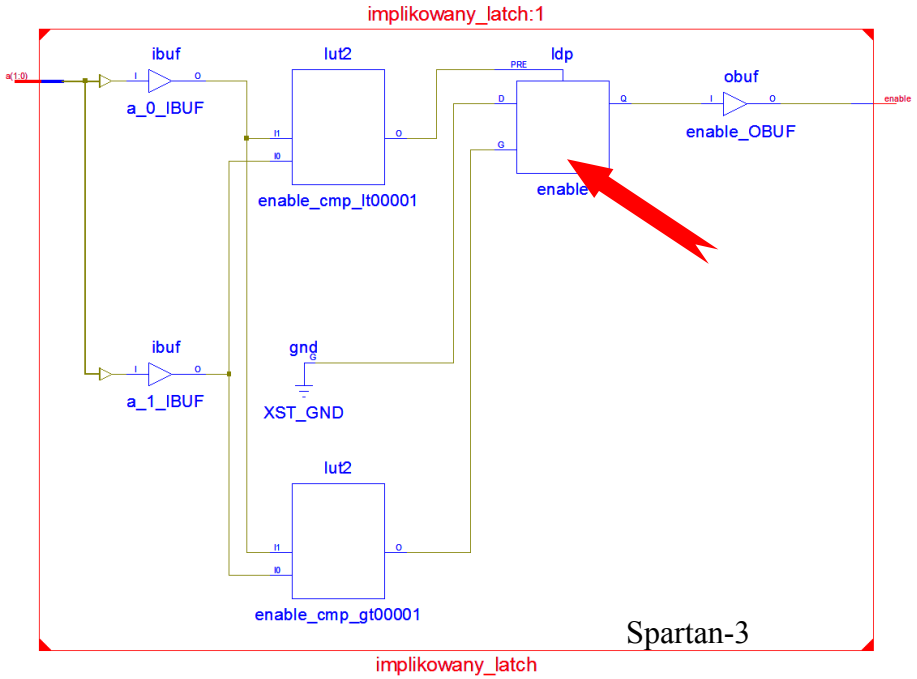
- `if (<condition>) instr_true;`
  `[else instr_false;]`
- `if-else` can be nested
- Use `if-else` for combinational logic
  `else` is not necessary for sequential logic

- Example

```
if(EN) Q = A;

if(Q < MAX) Q = Q + 1;
else Q = 0;

if(CTRL == 0)
  S = A + B;
else if(CTRL == 1)
       S = A - B;
     else if(CTRL == 2)
            S = A * B;
          else S = A;
```

Implicated „latch”

```
module impl_latch(a, enable);
input [1:0] a;
output reg enable;

always @(a)
 if(a<1)
    enable=1'b1;
 else
    if(a>2)
     enable=1'b0;

endmodule
```



- Other examples

```
if(a>10)
    enable=1'b1;
```

```
if(a>10)
    enable=1'b1;
else
    driver=1'b0
```

```
always @(in_a or in_b or sel)
begin
    case (sel)
          2'b00: out_a = in_a;
          2'b01: out_a = in_b;
    endcase
end
```

case

- Enables comparison the `expression` with `item` expressions and execute particular statement/statements

```
case(<expression>)
  item_1: instr_1;
  ...
  item_n: instr_n;
  default: instr_def;
endcase
```

- Example

```
case(CTRL)
  2'd0: S = A + B;
  2'd1: S = A - B;
  2'd2: S = A * B;
  default: S = A;
endcase
```

`for` loop

- `for(<e>,<c>,<i>)` is consisted of:
  - `<e>` initialization
  - `<c>` condition
  - `<i>` increment/decrement

- Example

```
parameter w = 8;
input [w-1:0] D;
reg [w-1:0] Q;
integer i;

always @(posedge CLK)
begin
  for(i=0; i<w; i=i+1)
    Q[i] <= D[i];
end
```

`for` loop

- rd84 Example

```
input [7:0] D;
reg [3:0] Y;
integer i, cnt;

always @(posedge clk)
begin
  cnt= 0;
  for(i=0; i<8; i=i+1)
    if(D[i])
        cnt= cnt +1;
  y<= cnt;
end
```

Verilog

- The HDL is a description of the operation and/or **construction** of the electronic circuit
- The source is called a **description** or **model**

- **Designed to simulate**

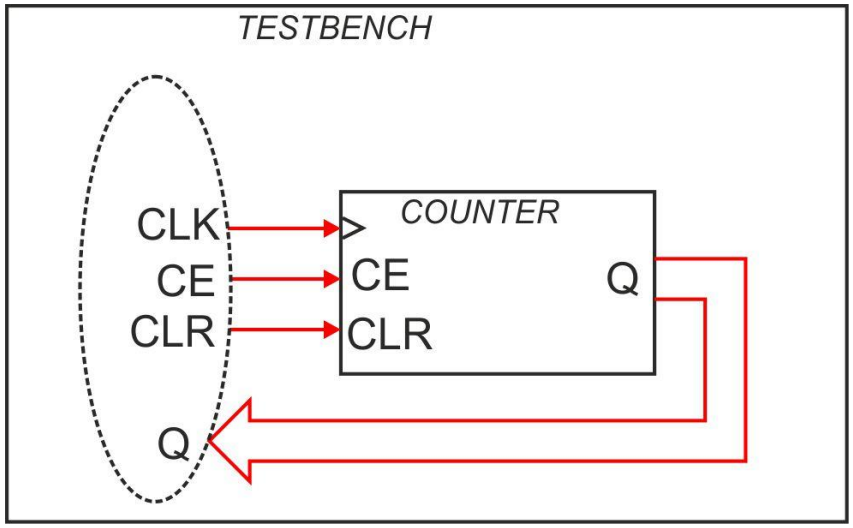
Module/instance – simulation

```
module COUNTER(Q,CLK,CLR);
input CLK,CLR;
output [2:0] Q;

//instances of the TFF module
TFF tff0(Q[0],CLK ,CLR);
TFF tff1(Q[1],Q[0],CLR);
TFF tff2(Q[2],Q[1],CLR);

endmodule
```

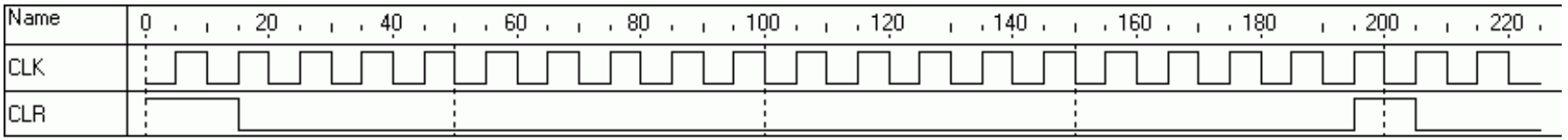
```
module TFF(...
...
endmodule
```

Instance – simulation

- Test module



- Example tests



Procedural blocks – `always` and `initial`

- Every procedural block is „executed” concurrently
- Simulation of every block begins in so called zero simulation time
- `initial` and `always` cannot be nested

`always` block in simulation

- „Execution” (simulation) of the procedural block begins at 0 time
- Block is „executed” as endless loop
- Initialization of signals from `always` block must be performed by using `initial` block
- Define the end of simulation because `always` is endless loop - otherwise the simulation will not end!

`Initial` block

- „Execution” of the procedural block begins at 0 time
- `initial` block is „executed” once during simulation
- It is not synthesizable!

```
module BEHAV;
reg CLK;

initial
begin
      CLK = 1'b0;
      forever #5 CLK = ~CLK;
end

initial
      $monitor("%t: CLK: %b",$time,CLK);
endmodule
```

Delays

```
initial
begin
  b = 1'b0;
  #10 a = b;
  //wait 10 time units; take b and assign to a
end
```

```
initial
begin
  x = 1'b0; y = 1'b1;
  z = #10 x + y;
  //count x + y; wait 10 time units; assign to z
end
```

Instances and simulation

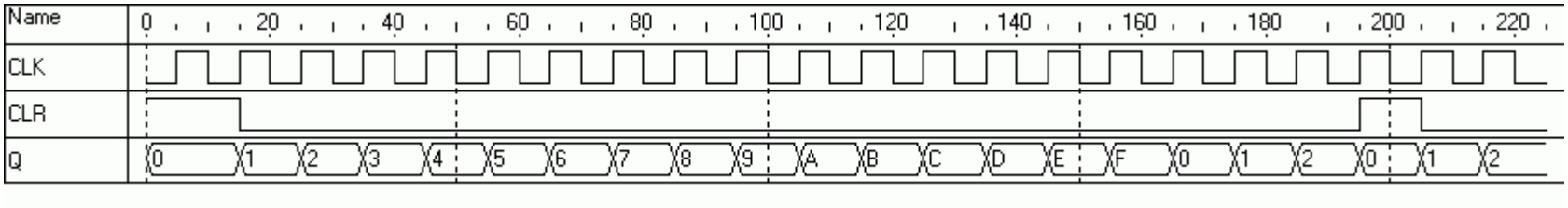
```
module Test;
reg CLK,CLR;
wire [3:0] Q;

CNT C1(Q,CLK ,CLR);

//Clock generator
initial
begin
   //initial value
   CLK = 1'b0;
   //generator
   forever
        #5 CLK = ~CLK;
end
...
```

Instances and simulation

```
//CLR control
initial
begin
   CLR = 1'b1;       //initial value t = 0 CLR = 1
   #15  CLR = 1'b0; // t = t + 15  = 15,  CLR = 0
   #180 CLR = 1'b1; // t = t + 180 = 195, CLR = 1
   #10  CLR = 1'b0;
   #20 $finish;      //the end
end

//monitor function
initial
   $monitor("%t: %b %b : %d",$time,CLK,CLR,Q);

endmodule
```

Simulation

- Stimulation test



- `$monitor`

```
 0: 0 1 :  0
 5: 1 1 :  0
10: 0 1 :  0
15: 1 0 :  1
20: 0 0 :  1
25: 1 0 :  2
30: 0 0 :  2
35: 1 0 :  3
40: 0 0 :  3
45: 1 0 :  4
50: 0 0 :  4
55: 1 0 :  5
```

```
 60: 0 0 :  5
 65: 1 0 :  6
 70: 0 0 :  6
 75: 1 0 :  7
 80: 0 0 :  7
 85: 1 0 :  8
 90: 0 0 :  8
 95: 1 0 :  9
100: 0 0 :  9
105: 1 0 : 10
110: 0 0 : 10
115: 1 0 : 11
```

Structural procedures

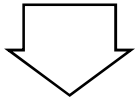
```
module stimulus;
reg a, b;

initial
  b = 1'b0;

initial begin
  #10 a = 1'b0;
  #15 a = 1'b1;
  #5  b = 1'b1;
end

initial
  #50 $finish;

endmodule
```

```
module CLKGEN;
reg clock;
parameter hper = 10;

initial
  clock = 1'b0;

always
   #hper clock = ~clock;

initial
  #1000 $finish;

endmodule
```

Procedural assignment

Blocking (=)

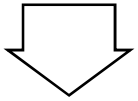
```
initial
begin
  b = 1'b0;
  #10 a = 1'b0;
  #15 a = 1'b1;
  #5  b = 1'b1;
end
```



time	Event
0	b = 0
10	a = 0
25	a = 1
30	b = 1

Nonblocking (<=)

```
initial
begin
  b = 1'b0;
  a <= #10 1'b0;
  a <= #15 1'b1;
  b <= #5 1'b1;
end
```



Time	Event
0	b = 0
5	b = 1
10	a = 0
15	a = 1

Directives

- Time

```
`timescale <ref_time_unit>/<time_precision>
`timescale 1ns / 1ps
```

Based on:

Robert Czerwiński „Digital Circuits Design” lecture