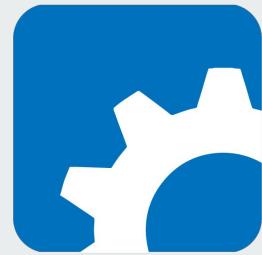
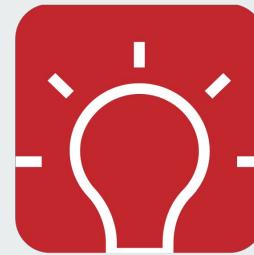
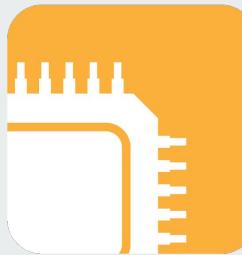


---

# Curso de Verilog



CdH

---

# Aula 3

---

# Conteúdo

- Níveis de Abstração: Behavioral
  - Always x Initial
  - Estruturas de controle de fluxo
    - if-else
    - for
    - case
  - Parameters
  - Lógica Combinacional x Sequencial
  - Blocking x Non-blocking statements
  - Tasks e Functions



---

## Always e Initial

- São as estruturas mais básicas em modelagem comportamental
- Todas as outras estruturas só podem aparecer dentro de uma dessas
- Não podem ser aninhadas

---

# Always

- **Informações Gerais:**

- Executa sempre
- Deve possuir um delay ou lista de sensibilidade associada a ele.
- Não faz atribuição para wires.
- Ao utilizar lista de sensibilidade a estrutura será:
  - `always@(<lista>)`
- Dica:
  - O comando `always@(*)` utiliza todos os sinais de que o bloco depende na lista de sensibilidade.

---

# Always x Initial

Executa sempre

Caso haja múltiplos blocos com always, eles são executados concorrentemente.

É sintetizável

Executado uma única vez no início da simulação

Caso haja múltiplos blocos com initial, eles são executados concorrentemente.

Geralmente não é sintetizável, com exceção de utilização para memórias.

# Exemplo initial

```
module stimulus;  
  
reg x,y, a,b, m;  
  
initial  
  
  m = 1'b0; //single statement; does not need to be grouped  
  
initial  
begin  
  #5 a = 1'b1; //multiple statements; need to be grouped  
  #25 b = 1'b0;  
end
```

```
initial  
begin  
  #10 x = 1'b0;  
  #25 y = 1'b1;  
end  
  
initial  
  #50 $finish;  
  
endmodule  
  
time      statement executed  
0         m = 1'b0;  
5         a = 1'b1;  
10        x = 1'b0;  
30        b = 1'b0;  
35        y = 1'b1;  
50        $finish;
```

---

## Exemplo Always: Multiplexador

```
always@(A or B or sel)
begin
    if(sel==0)
        C=A;
    else
        C=B;
end
endmodule
```

//Toggle clock every half-cycle (time period = 20)  
always #10 clock = ~clock;

---

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop



---

# Estruturas de Controle de Fluxo

- **Conditional**
- **Case**
- **Loop**
- **If e else** são as palavras utilizadas para estruturas condicionais
- 3 tipos de estrutura:
  - Sem else:
    - if(<expressão>) true\_statement;
  - Com 1 else
    - if(<expressão>) true\_statement;
    - else false\_statement;
  - Else-if aninhados
    - if(<expressão1>) true\_statement1;
    - else if (<expressão2>) true\_statement2;
    - else if (<expressão3>) true\_statement3;
    - else default\_statement;

---

# Estruturas de Controle de Fluxo

- Conditional
  - **Case**
  - Loop
- **Case, endcase e default** são as palavras chave utilizadas nessa expressão
  - Pode ser aninhado
  - Case compara valores **0, 1, z e x**
  - **default** é “opcional”
  - Exemplo:
    - `case(expressão)`
      - `alternativa1: statement1;`
      - `alternativa2: statement2;`
      - `...`
      - `default: default_statement;`
    - `endcase`

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop
- Exemplo de case com e sem default:

```
1 module mux (a,b,c,d,sel,y);
2 input a, b, c, d;
3 input [1:0] sel;
4 output y;
5
6 reg y;
7
8 always @ (a or b or c or d or sel)
9 case (sel)
10 0 : y = a;
11 1 : y = b;
12 2 : y = c;
13 3 : y = d;
14 default : $display("Error in SEL");
15 endcase
16
17 endmodule
```

```
1 module mux_without_default (a,b,c,d,sel,y);
2 input a, b, c, d;
3 input [1:0] sel;
4 output y;
5
6 reg y;
7
8 always @ (a or b or c or d or sel)
9 case (sel)
10 0 : y = a;
11 1 : y = b;
12 2 : y = c;
13 3 : y = d;
14 2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
15 2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display("Error in SEL");
16 endcase
17
18 endmodule
```



---

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop
- Possui 2 variações:
  - casez:
    - trata todos os valores z como don't care
  - casex:
    - trata todos os valores x e z como don't care



# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop

```
17 always @ (sel)
18 case (sel)
19 1'b0 : $display("Normal : Logic 0 on sel");
20 1'b1 : $display("Normal : Logic 1 on sel");
21 1'bx : $display("Normal : Logic x on sel");
22 1'bz : $display("Normal : Logic z on sel");
23 endcase
24
25 always @ (sel)
26 casex (sel)
27 1'b0 : $display("CASEX : Logic 0 on sel");
28 1'b1 : $display("CASEX : Logic 1 on sel");
29 1'bx : $display("CASEX : Logic x on sel");
30 1'bz : $display("CASEX : Logic z on sel");
31 endcase
32
33 always @ (sel)
34 casez (sel)
35 1'b0 : $display("CASEZ : Logic 0 on sel");
36 1'b1 : $display("CASEZ : Logic 1 on sel");
37 1'bx : $display("CASEZ : Logic x on sel");
38 1'bz : $display("CASEZ : Logic z on sel");
39 endcase
```

Driving 0  
Normal : Logic 0 on sel  
CASEX : Logic 0 on sel  
CASEZ : Logic 0 on sel

Driving 1  
Normal : Logic 1 on sel  
CASEX : Logic 1 on sel  
CASEZ : Logic 1 on sel

Driving x  
Normal : Logic x on sel  
CASEX : Logic 0 on sel  
CASEZ : Logic x on sel

Driving z  
Normal : Logic z on sel  
CASEX : Logic 0 on sel  
CASEZ : Logic 0 on sel



---

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop

- while
  - Exemplo:

```
initial
begin
    count = 0;

    while (count < 128) //Execute loop till count is 127.
                           //exit at count 128
    begin
        $display("Count = %d", count);
        count = count + 1;
    end
end
```

- for
- repeat
- forever



---

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop
- while
- for
- Exemplo:

```
integer count;  
  
initial  
  for ( count=0; count < 128; count = count + 1)  
    $display("Count = %d", count);
```

- repeat
- forever



---

# Estruturas de Controle de Fluxo

- Conditional
- Case
- Loop
  - while
  - for
  - repeat
    - O número de repetições deve ser uma constante, variável ou valor de um sinal.

```
initial
begin
    count = 0;
repeat(128)
begin
    $display("Count = %d", count);
    count = count + 1;
end
end
```



- forever

---

# Estruturas de Controle de Fluxo

- Conditional
  - Case
  - Loop
- while
  - for
  - repeat
  - forever
    - Executa até encontrar um **\$finish**
    - Usado com estruturas de controle de tempo
    - Exemplo:

```
reg clock;  
  
initial  
begin  
    clock = 1'b0;  
    forever #10 clock = ~clock;  
end
```



---

# Parameters

- **Parameter**
- **defparam**
- **Modificação na instanciação**
- Representa uma constante que pode ser customizada durante a compilação.

---

# Parameters

- Parameter
- **defparam**
- Modificação na instanciação
- Utilizado para sobrescrever valores de parâmetros

---

# Parameters

- Parameter
- **defparam**
  - **Exemplo:**
- Modificação na instanciação

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0

initial //display the module identification number
    $display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

//instantiate two hello_world modules
hello_world w1();
hello_world w2();

endmodule

Displaying hello_world id number = 1
Displaying hello_world id number = 2
```



---

# Parameters

- Parameter
- defparam
- Modificação na instanciação
- Forma alternativa de modificar parâmetros
- Estrutura:
  - <nome\_módulo> #(parâmetro) (<portas>)...
- Exemplo:

```
//define top-level module
module top;

    //instantiate two hello_world modules; pass new parameter values
    hello_world #(1) w1; //pass value 1 to module w1
    hello_world #(2) w2; //pass value 2 to module w2

endmodule
```

---

# Lógica Combinacional X Lógica Sequencial

Saída depende apenas dos valores de entrada atuais.

Não utiliza clock

Always Combinacional:  
always(\*)

Saída pode depender do valor atual e de valores anteriores.

Utiliza clock

Always Sequencial:  
always(posedge clk)  
Ou  
always(negedge clk)



---

# Blocking x Non-Blocking Assignment

Representado por '='

Avaliação e assign imediato

Usado em always Combinacional

Representado por '<='

Todas as expressões são avaliadas e depois o assign é feito.

Usado em always Sequencial



## Exemplo

```
module teste2(
    input[2:0] A,
    input clk,
    output reg[2:0] D
);
    reg[2:0] B,C;
    always@(posedge clk)
begin
    B=A;
    C=B;
    D=C;
end
endmodule
```

```
module teste(
    input[2:0] A,
    input clk,
    output reg[2:0] D
);
    reg[2:0] B,C;
    always@(posedge clk)
begin
    B<=A;
    C<=B;
    D<=C;
end
endmodule
```

```

module teste2(
    input[2:0] A,
    input clk,
    output reg[2:0] D
);
    reg[2:0] B,C;
    always@(posedge clk)
    begin
        B=A;
        C=B;
        D=C;
    end
endmodule

```



```

module teste(
    input[2:0] A,
    input clk,
    output reg[2:0] D
);
    reg[2:0] B,C;
    always@(posedge clk)
    begin
        B<=A;
        C<=B;
        D<=C;
    end
endmodule

```







# Tasks

- Estrutura:
  - task <nome>;
  - <declaração>
  - <statements>
  - endtask
- Utilizar se:
  - Houver delays, estruturas de controle de evento
  - Houver zero ou mais de um output
  - Não houver inputs.
- Peculiaridades:
  - Pode acionar outras tasks e functions
  - Pode possuir portas de input, output e inouts.

# Tasks

```
//Define a module called operation that contains the task bitwise_oper
module operation;
...
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @ (A or B) //whenever A or B changes in value
begin
    //invoke the task bitwise_oper. provide 2 input arguments A, B
    //Expect 3 output arguments AB_AND, AB_OR, AB_XOR
    //The arguments must be specified in the same order as they
    //appear in the task declaration.
    bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
...
...
//define task bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //outputs from the task
input [15:0] a, b; //inputs to the task
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end

```

```
//Define a module that contains the task asymmetric_sequence
module sequence;
...
...
reg clock;
...
initial
    init_sequence; //Invoke the task init_sequence
...
always
begin
    asymmetric_sequence; //Invoke the task asymmetric_sequence
end
...
...
//Initialization sequence
task init_sequence;
begin
    clock = 1'b0;
end
endtask

```

---

# Functions

- Estrutura:
  - `function <range><nome>;`
  - `<declarações>`
  - `<statements>`
  - `endtask`
- Utilizar se:
  - Não houver delays, timing e estruturas de controle de evento.
  - Houver somente um output
  - Houver pelo menos um input
- Peculiaridades:
  - Pode acionar outra function, mas não uma task.
  - Não há outputs pois um registrador é criado com o nome da function e o resultado é atribuído a ele.
  - o `<range>` na criação da function define o tamanho desse registrador

# Functions

```
//Define a module that contains the function calc_parity
module parity;
...
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @(addr)
begin
    parity = calc_parity(addr); //First invocation of calc_parity
    $display("Parity calculated = %b", calc_parity(addr));
                                //Second invocation of calc_parity
end
...
...
//define the parity calculation function
function calc_parity;
input [31:0] address;
begin
    //set the output value appropriately. Use the implicit
    //internal register calc_parity.
    calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
```

```
//Define a module that contains the function shift
module shifter;
...
//Left/right shifter
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

//Compute the right- and left-shifted values whenever
//a new address value appears
always @(addr)
begin
    //call the function defined below to do left and right shift.
    left_addr = shift(addr, `LEFT_SHIFT);
    right_addr = shift(addr, `RIGHT_SHIFT);
end
...
...
//define shift function. The output is a 32-bit value.
function [31:0] shift;
input [31:0] address;
input control;
begin
    //set the output value appropriately based on a control signal.
    shift = (control == `LEFT_SHIFT) ?(address << 1) : (address
>> 1);
end
endfunction
```

# Prática - Behavioral

