FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

T566 –SISTEMAS DIGITAIS AVANÇADOS

Aula 11- Verilog (Cont. 1)

Prof. Danilo Reis

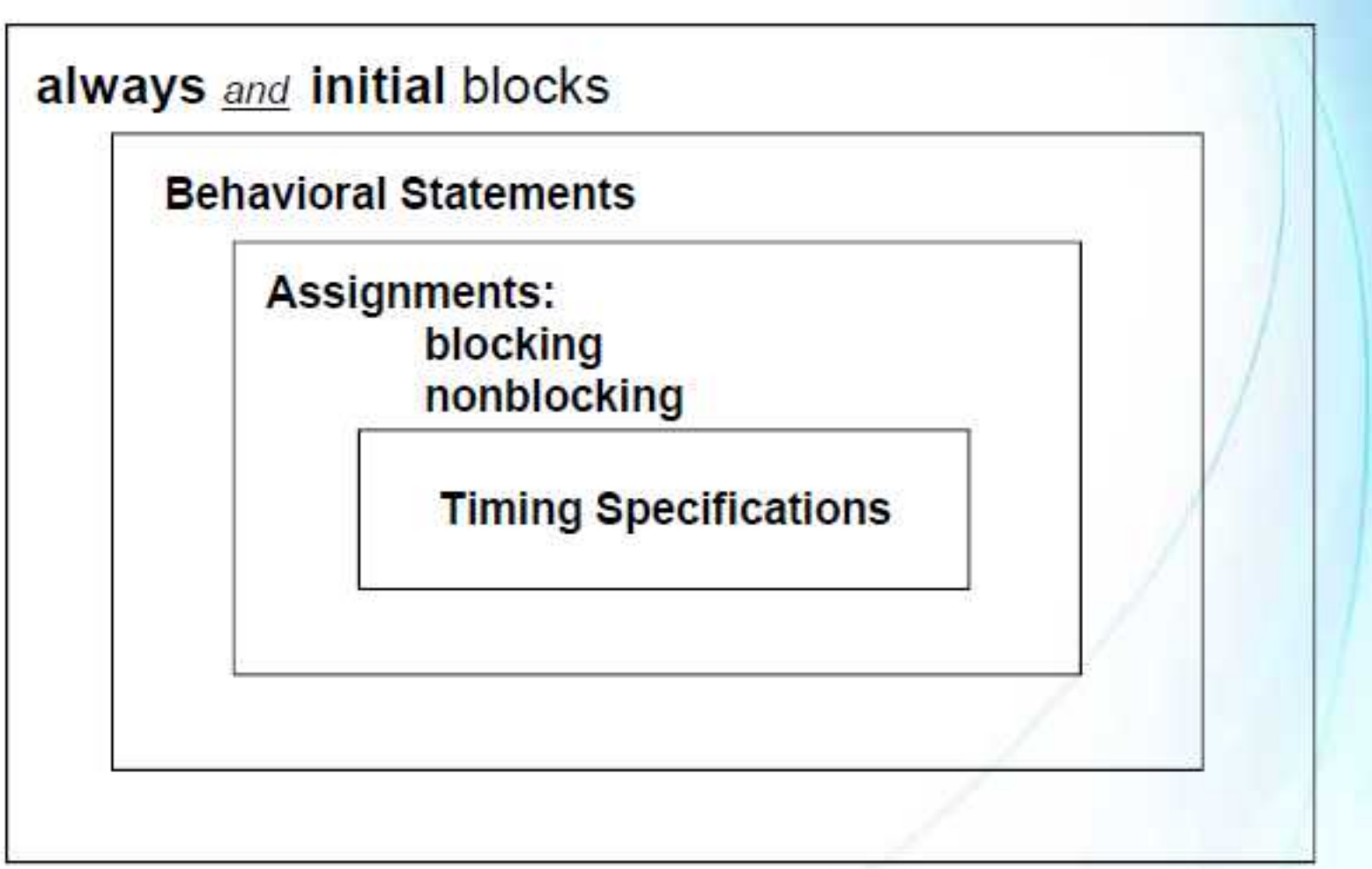
Blocos Procedurais

initial
- Usado para inicializar instruções de comportamento para simulação.

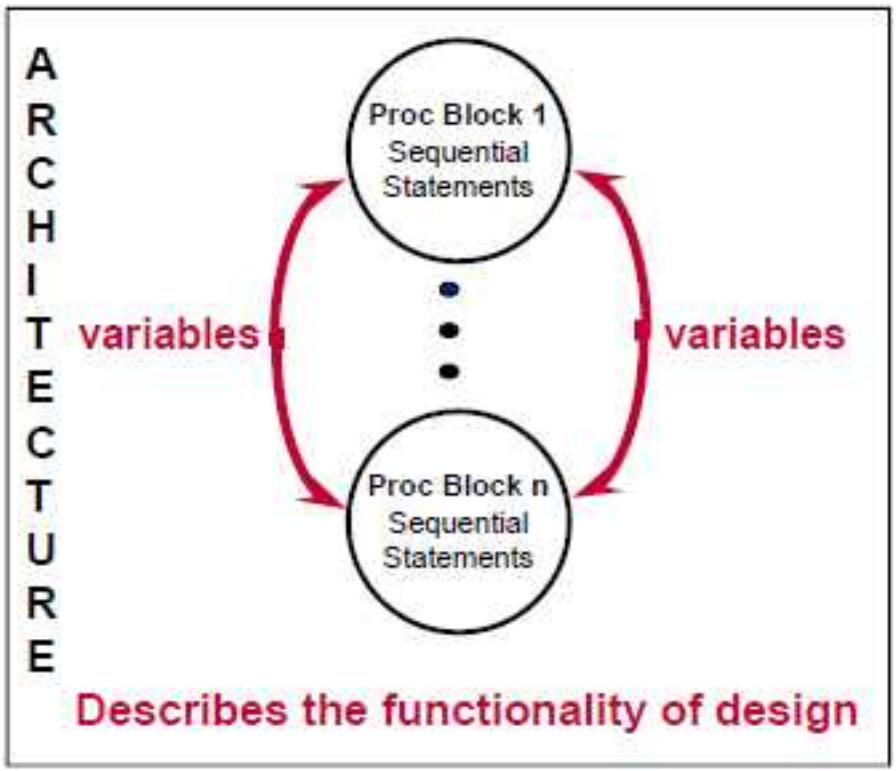
always
- Usaso para descrever a funcionalidade do circuito usando instruções comportamentais.

- Cada bloco always e initial representa um processo separado
- Processos executam em paralelo e começam no time 0 da simulação
- Instruções dentro de um processo são executadas sequencialmente.
- Blocos always e initial não podem ser aninhados.

Blocos Procedurais



Blocos Procedurais



- Cada bloco procedural executa em paralelo com os outros blocos procedurais.
  - A ordem dos blocos always/initial blocks não importa
- Dentro de um bloco procedural , as instruções são executadas sequencialmente.
  - A ordem das instruções dentro de um bloco always/initial impo rta.

Blocos Procedurais características

1. LHS deve ser uma variavel do tipo data type (e.g. reg,integer, real, time);
2. LHS pode ser um a bit-select ou part-select;
3. LHS pode ser uma concatenação de qualquer um dos tipos acima;
4. RHS pode ser uma expressão contendo variaveis net data type, ou function call (ou combinação destas)

Blocos Initial

- Consiste de instruções comportamentais;
- Cada bloco initial executa concorrentemente iniciando no tempo 0, executa apenas uma vez e para;
- Tem que usar as palavras chaves **begin** e **end** para agrupar o instruções de comportamento;
- Exemplos de usos:
  - *Inicialização*
  - *Monitoramento*
  - *Qualquer funcionalidade que necessita ser executada apenas uma vez;*

***Note** que o bloco **initial** executa apenas uma vez, a duraçào pode ser infinita (i.e. funcionalidade dentro pode continuar sem término determinado no modelo de execução)*

Exemplo Blocos Initial

```
module system;

    reg a, b, c, d;

    // single statement
    initial a = 1'b0;

    /* multiple statements:
       needs to be grouped */
    initial begin
        b = 1'b1;
        #5 c = 1'b0;
        #10 d = 1'b0;
    end

    initial #20 $finish;

endmodule
```

Time	Statement(s) Executed
0	a = 1'b0; b = 1'b1
5	c = 1'b0;
15	d = 1'b0;
20	$finish

Blocos always

- Consiste de instruções de comportamento;
- Cada bloco **always** executa concorrentemente começando no tempo 0 e executa continuamente em um loop;
- Deve-se usar as palavras chaves **begin** e **end** para agrupar instruções de comportamento quando o bloco **always** contem mais que uma instrução;
- Exemplos de uso:
  - Modelar um circuito digital;
  - Qualquer processo ou funcionalidade que necessite ser executada continuamente.

Exemplo blocos always

```
module clk_gen
    #(parameter period = 50)
    (
    output reg clk
);

    initial clk = 1'b0;

    always
        #(period/2) clk = ~clk;

    initial #100  $finish;

endmodule
```

Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	$finish;

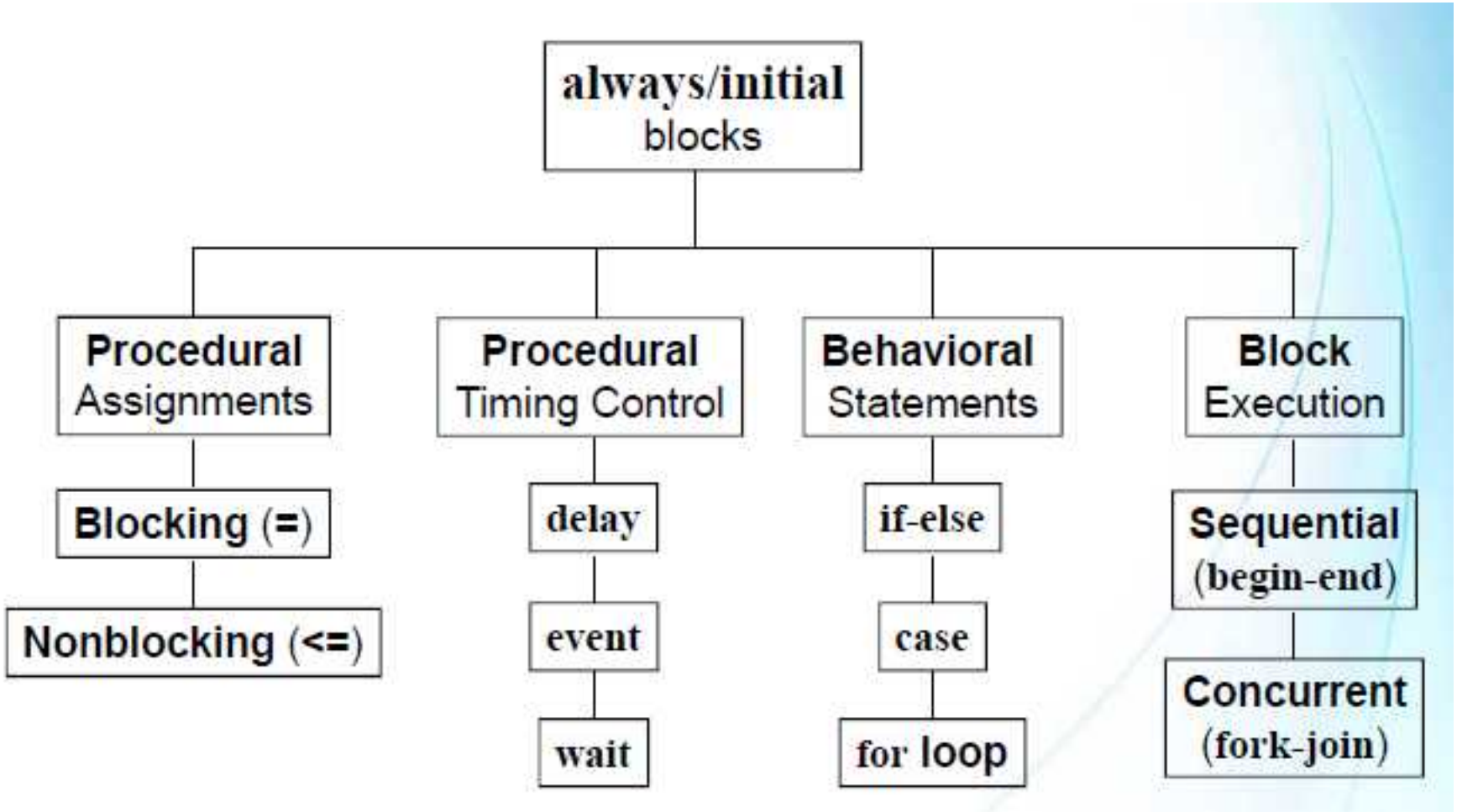
Nomeando Blocos

- Blocos procedurais podem ser nomeados adicionando : <name> depois do **begin;**
- Vantagens:
  - *Permitir que o bloco seja referenciado em outras partes do código pelo nome;*
  - *Permitir declaração de objetos locais ao bloco procedural;*
  - *Permitir monitoramento de blocos procedurais pelo nome na simulação;*

```
initial
begin : clock_init
    clk = 1'b0;
end

always
begin : clock_proc
    #(period/2) clk = ~clk;
end
```

Estrutura dos Blocos always/initial



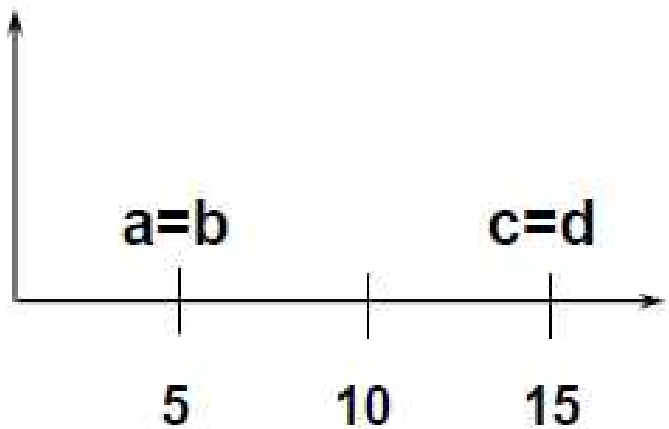
Nomeando Blocos

- Blocking (=) : Atualiza a atribuição do LHS bloqueando a execução de outras atribuições dentro do bloco procedural até que o mesmo seja finalizado
  - RHS (inputs) amostradas quando a instrução é executada;
  - LHS (outputs) atualizadas imediatamente ou depois de um delay definido;
  - Instruções seguintes devem esperar o término da instrução blocante
- Nonblocking (<=) : Agenda a atribuição do LHS sem bloquear a execução das demais instruções do bloco.
  - RHS (inputs) amostradas quando a instrução é executada;
  - LHS (outputs) agendadas para serem atualizadas no final da execução ou após um delay definido expirar;
  - As instruções seguintes não ficam bloqueadas;

Blocking x Nonblocking assignments

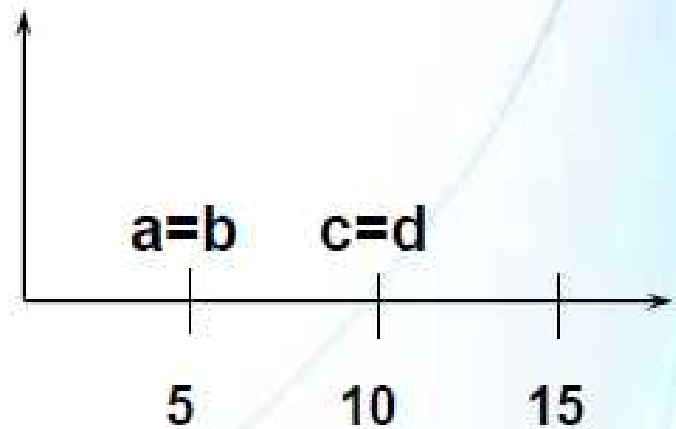
Blocking (=)

```
initial begin
    #5 a = b;
    #10 c = d;
end
```



Nonblocking (<=)

```
initial begin
    #5 a <= b;
    #10 c <= d;
end
```



Avaliando Blocking & Nonblocking

```
initial begin
   a = 1'b0;  //Assgnmnt0
   b = 1'b1;  //Assgnmnt1
   #5 a <= b; //Assgnmnt2
   #5 b <= a; //Assgnmnt3
end
```

Isto pode parecer confuso, mas é perfeitamente válido em Verilog e esta aqui para ilustrar o comportamento de instruções blocantes e não blocantes

1. Assignment 0 executa e atribui para **a** valor 0 bloqueando as demais instruções;
2. Assignment 1 executa e atribui a **b** o valor 1 bloqueando as demais instruções;
3. Assignment 2 executa e agenda para colocar o valor de **b** em **a** depois de 5 passos de tempo;
4. Assignment 3 executa e agenda para colocar o valor de **a** em **b** depois de 5 passos de tempo
5. Processo término 5 passos de tempo depois que **a** é atualizado para 0 e **b** para 1

Exemplos Blocking x Nonblocking

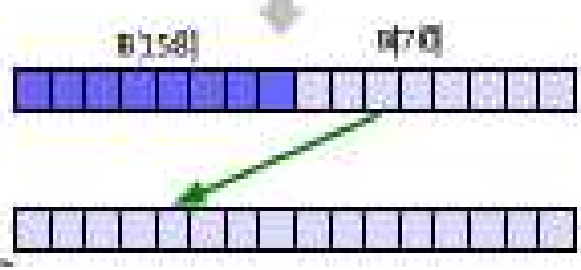
- Example: Swap bytes in words

- Which one is correct?

Blocking

Incorrect

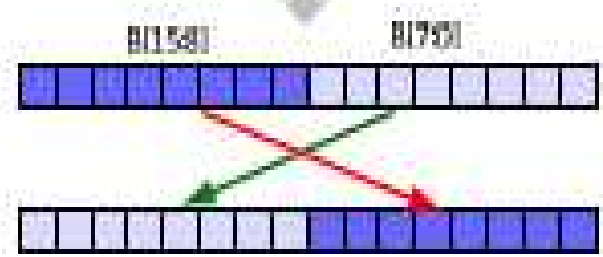
```
always @ (*)
    begin
        B[15:8] = B[7:0];
        B[7:0]  = B[15:8] ;
    end
```



Non-Blocking

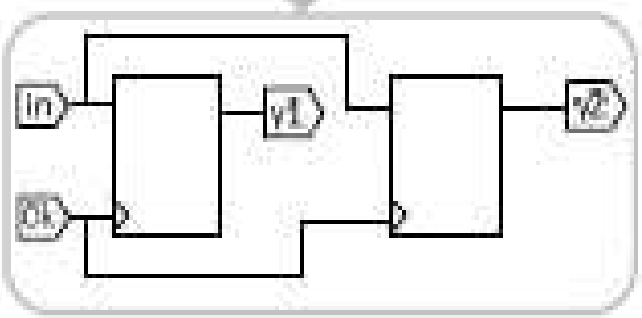
```
always @ (*)
    begin
        B[15:8] <= B[7:0];
        B[7:0]  <= B[15:8] ;
    end
```

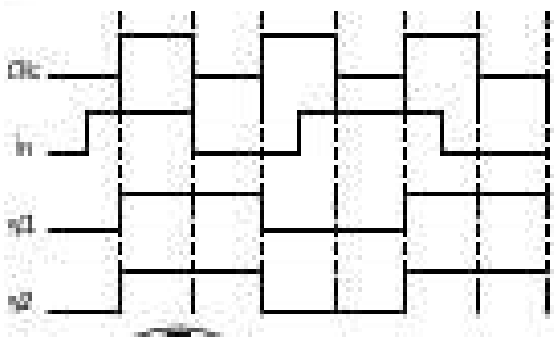
Correct



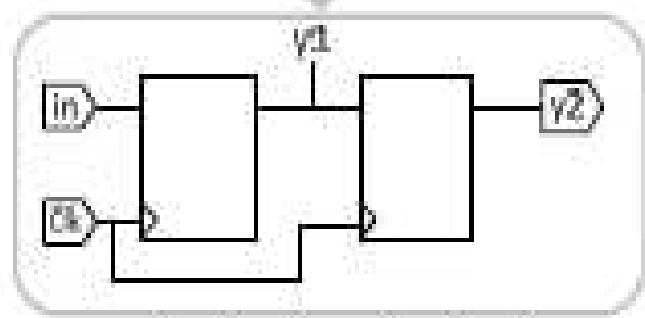
Exemplos Blocking x Nonblocking

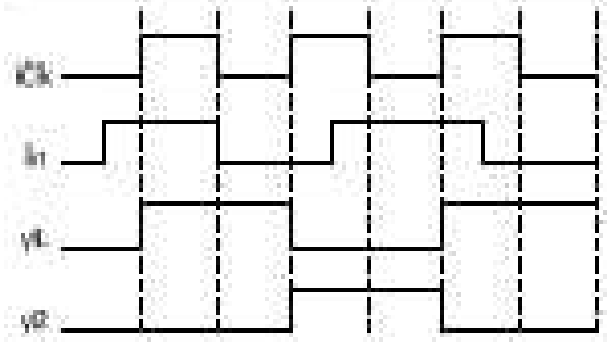
```
always @ (posedge Clk)
    begin
        y1 = in;
        y2 = y1 ;
    end
```



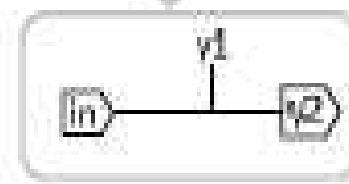



```
always @ (posedge Clk)
    begin
        y1 <= in;
        y2 <= y1 ;
    end
```

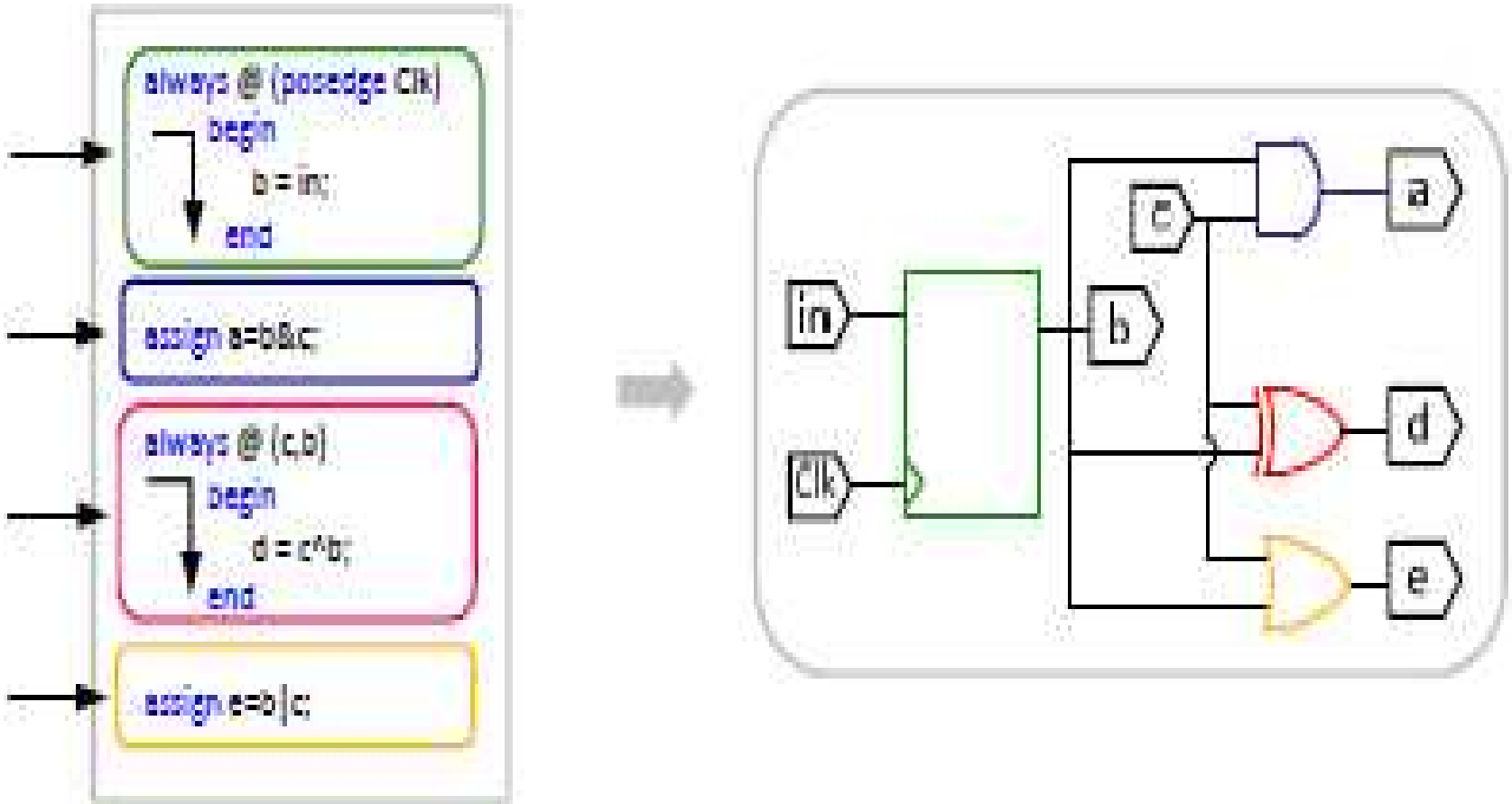




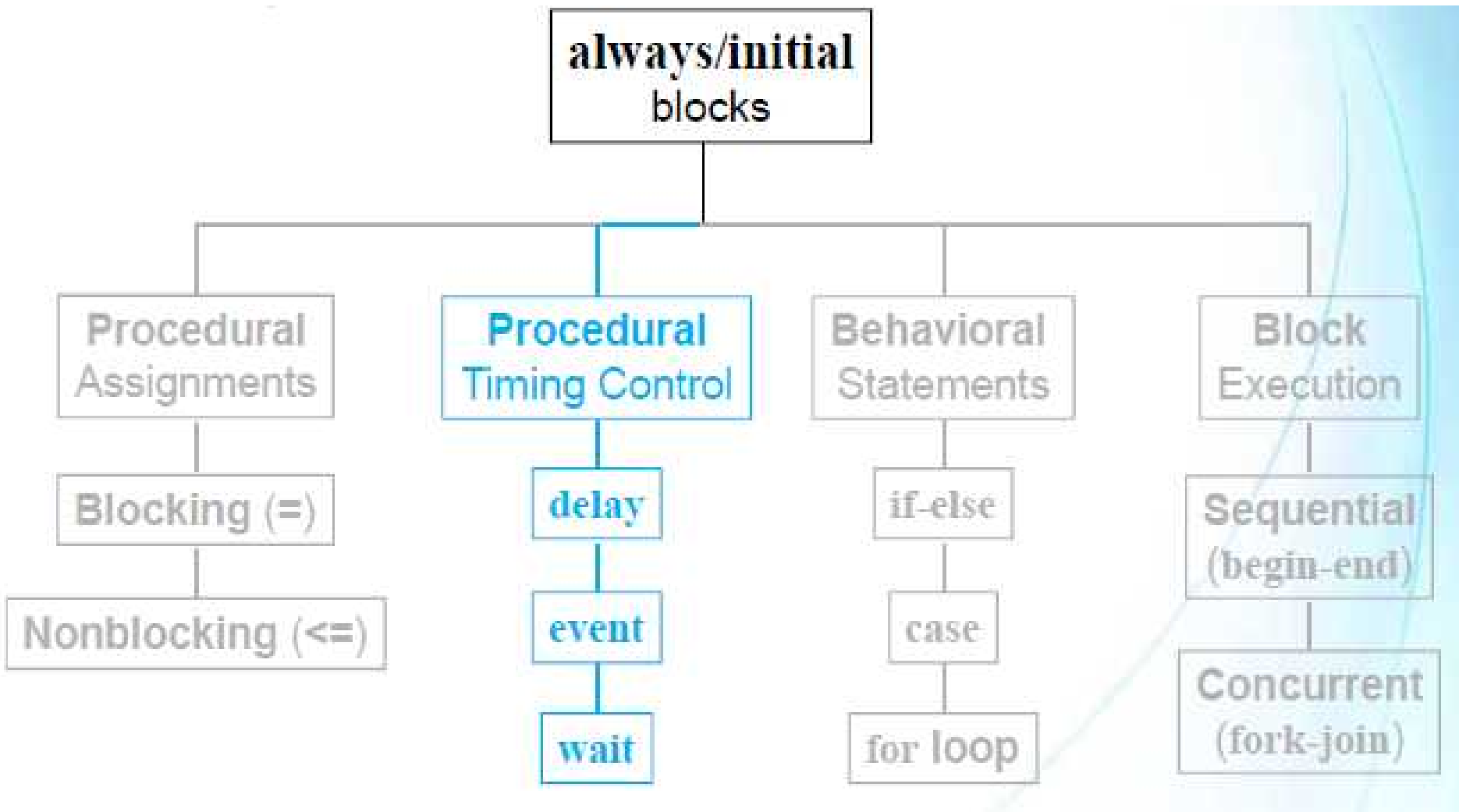

```
always @ (*)
    begin
        y1 = in;
        y2 = y1 ;
    end
```



Exemplos Blocking x Nonblocking



Blocos always/initial (Procedural Timing Control)



Procedural Timing Control

- Controle de **Delay**
- Controle de Eventos (**Event**)
- Instrução **Wait**
- Eventos nomeados(Named event)

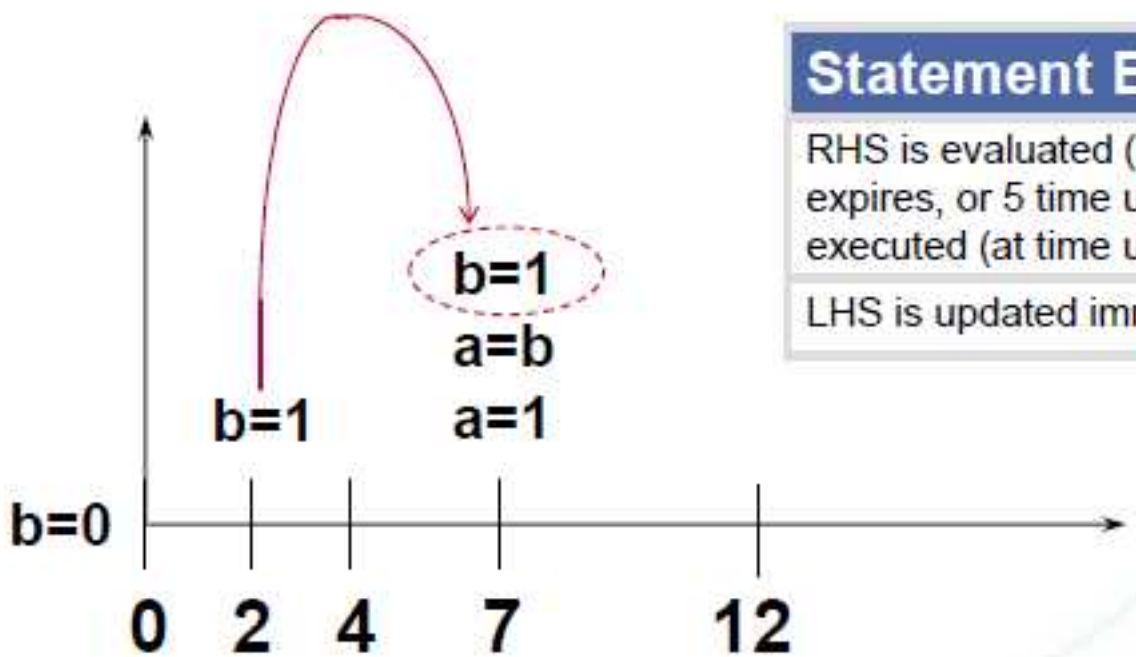
Delay Controls for Procedural Assignment

- Use **#<value>** notação para gerar um **delay** em procedimento de atribuição
- Regular (Inter-Assignment) Delay Control
- Intra-assignment Delay Control
- Zero Delay Control
- Ignorado pela sintese;

Regular (Inter-Assignment) Delay Control

```
#5 a = b;
```

- Ambos Delays lados do assignment read (RHS) e write (LHS) ;



Statement Execution (1)

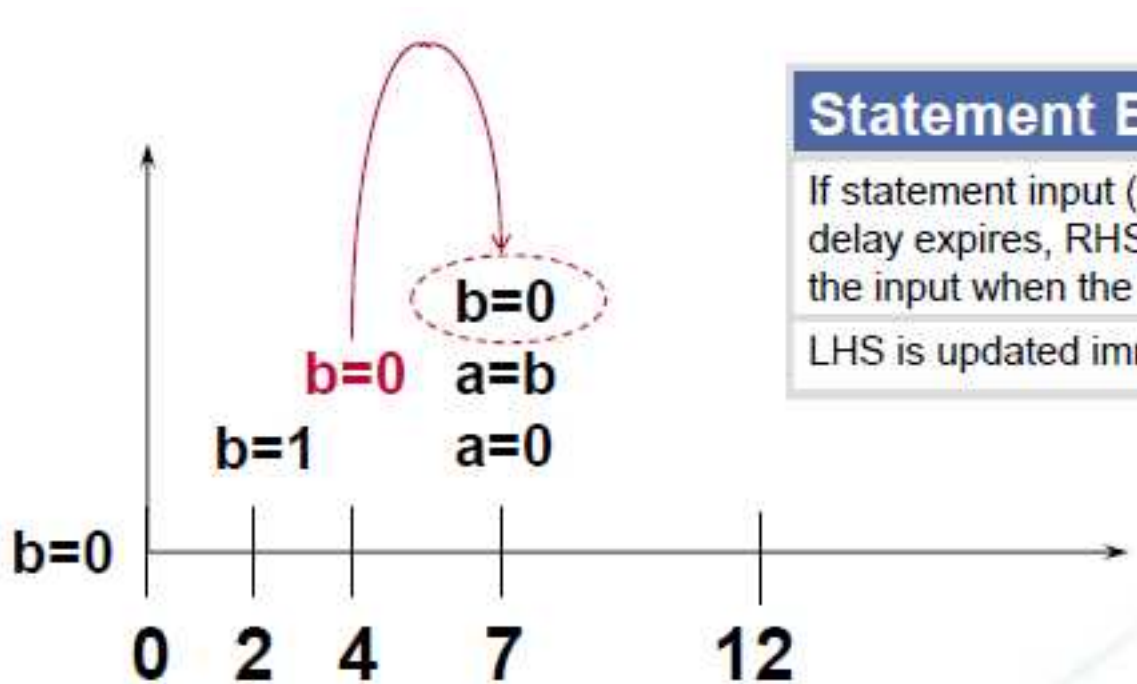
RHS is evaluated (i.e. **b** is read) after delay expires, or 5 time units after statement is executed (at time unit 7)

LHS is updated immediately

Regular (Inter-Assignment) Delay Control(cont)

```
#5  a = b;
```

- Ambos Delays lados do assignment read (RHS) e write (LHS) ;



Statement Execution (2)

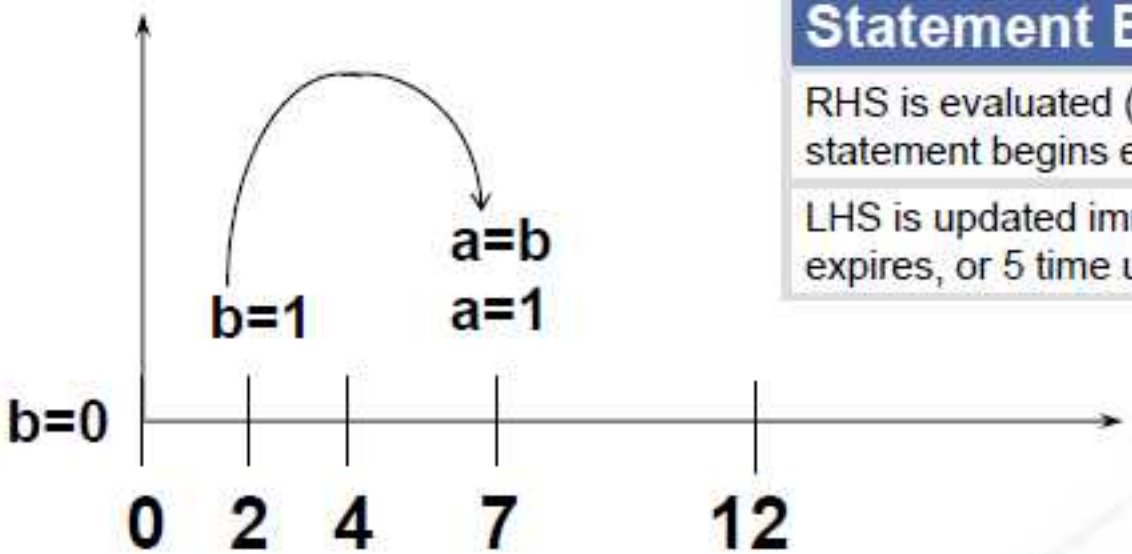
If statement input (RHS, or **b**) changes before delay expires, RHS picks up the new value of the input when the read actually occurs

LHS is updated immediately with new value

Intra-Assignment Delay Control

```
a = #5 b;
```

- Delays apenas do lado da escrita (write) (LHS) ;



Statement Execution (1)

RHS is evaluated (i.e. **b** is read) when statement begins execution

LHS is updated immediately after delay expires, or 5 time units later

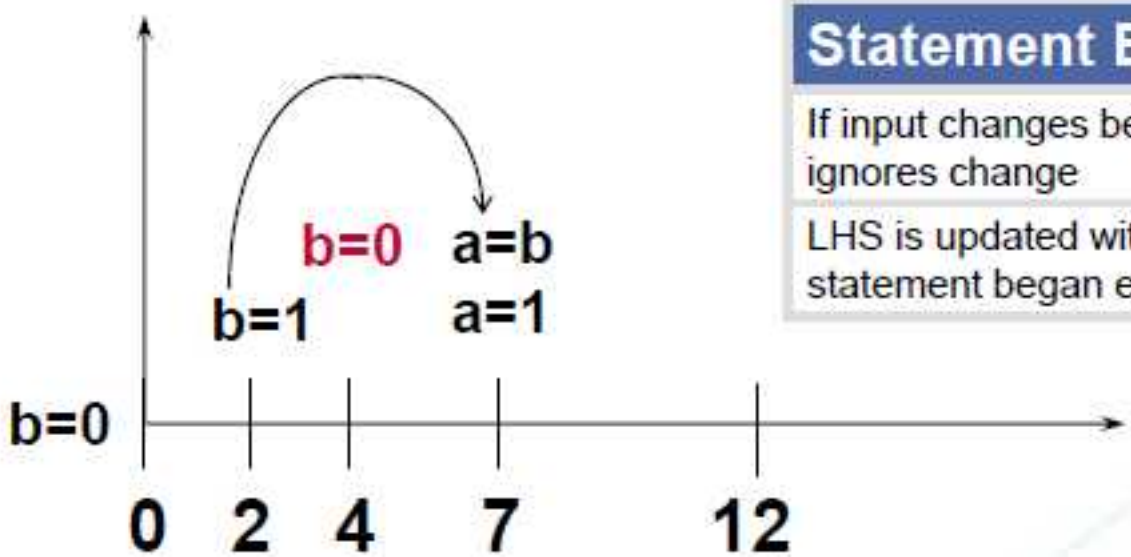
Intra-Assignment Delay Control (Cont.)

```
a = #5 b;
```

<=>

```
temp = b;
#5 a = temp;
```

- Delays apenas do lado da escrita (write) (LHS) ;
- Ignora trocas que acontecem no delay;



Statement Execution (2)

If input changes before delay expires, RHS ignores change

LHS is updated with value of RHS when statement began execution

Zero Delay Control

```
initial begin
    a = 0;
    b = 0;
end

initial begin
    #0 a = 1;
    #0 b = 1;
end
```

Statement Execution
All four statements will be executed at simulation time 0
Since #0 used for statements a = 1 and b = 1 have, they will be executed last.

- Prover um modo de controlar a ordem de execução no tempo 0;
- Não recomenda-se atribuir diferentes valores para uma variável ou em processos diferentes;

`timescale Compiler Directive Preview

Time unit for delays in module

Precision for delay values used by simulator

```
`timescale 1 ns / 10 ps

module mult_acc (
    input [7:0] ina, inb,
    input clk, clr,
    output [15:0] out
);
        ...
    assign #5 ...
    assign #25 ...
```

Controle de Eventos

- Prover um controle sensível a borda sensitiva simbolizado pelo simbolo @

@(expression)

- Pausa a execução do procedimento de instruções até que o evento ocorra (i.e. o valor da expressão muda)
- Para testar múltiplos eventos (lógica OR de uma lista de eventos)Vírgulas (,) (Verilog '2001 e posterior)

Examples*

```
initial begin
      // Rising edge control (inter-assignment)
      @ (posedge clk) r1 = r2;

      // Falling edge control (intra-assignment)
      r3 = @(negedge clk) r4;

      // Either edge control
      @ (a) r5 = r6;

      // Either edge control using more
      //    complex expression
      @ (a ^ b & c) r7 = r8

      // Logical OR of two events using comma
      @ (posedge clk, enable) r9 = r10;

      // Logical OR of two events events using
      //       "or" keyword
      r11 = @ (posedge clk or enable) r12;
end
```

Controle de Eventos

- Usa controle de eventos no início de um bloco **always** block para controlar quando o bloco deve iniciar;
  - Cada execução do bloco **always** requer que o evento tenha ocorrido;
  - Força o bloco **always** seja "sensível" aos itens no controle de eventos;
- Suportado pela ferramenta de síntese;

Format

```
always @(sensitivity_list) begin
    -- Statement_1
    -- ..............
    -- Statement_N
end
```

Example

```
// Process executes whenever
//    a, b, c or d changes in value
always @(a, b, c, d) begin
    #15 y = (a ^ b) & (c ~| d);
end
```

Instrução wait

- Prover controle de eventos sensível a nível;
- Usa palavra reservada **wait;**

wait (*expression*)

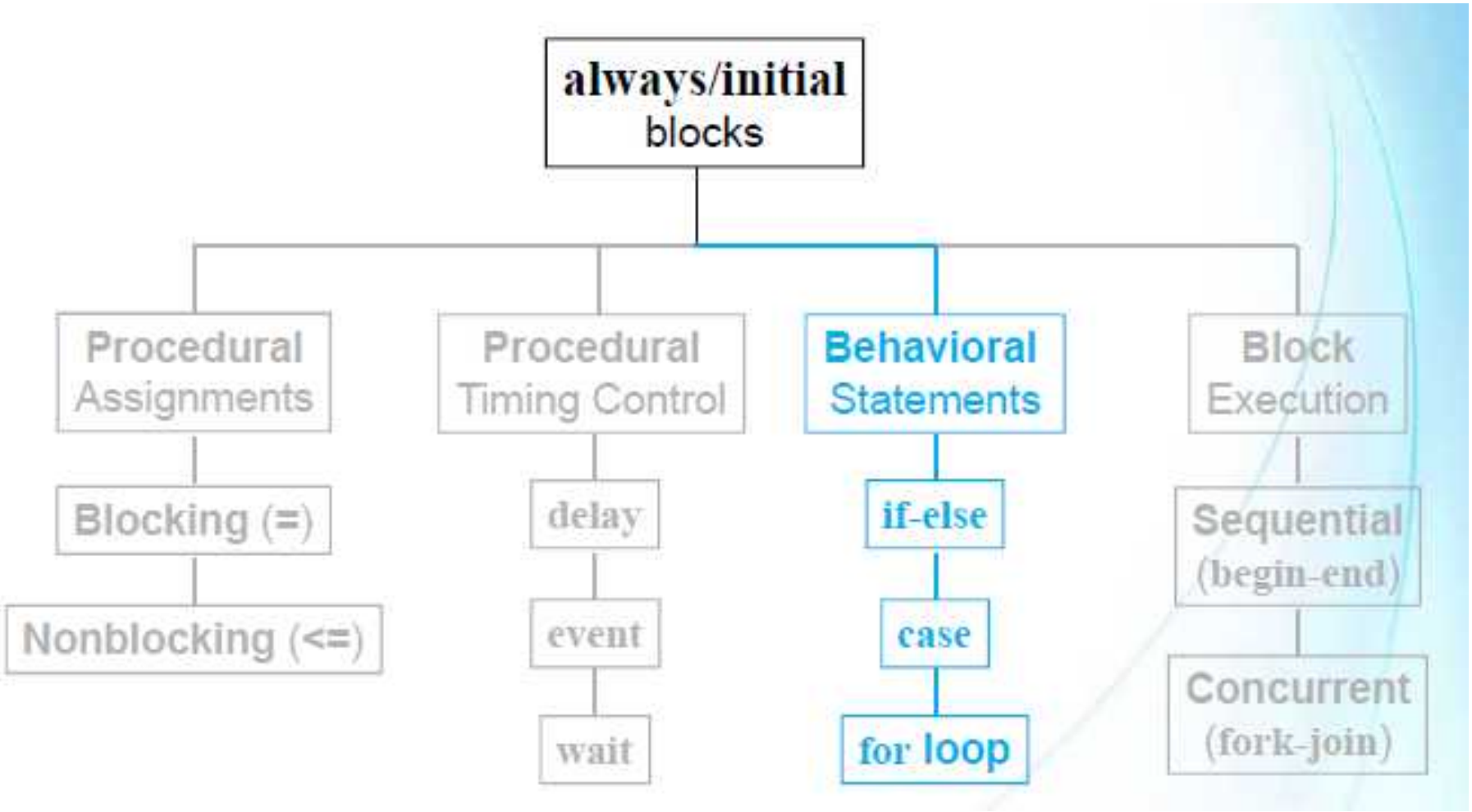
- Pausa a execução de um bloco de procedimento até a instrução **wait** seja satisfeita;
  - Se o nível não for satisfeito o bloco fica aguardando;
  - Se o nível for satisfeito, o bloco prossegue com a execução
- Não suportada pela síntese;

Example

```
initial begin
    ...
    wait (gate) r1 = r2;  //Assignment0
    wait (!gate) r3 = r4; //Assignment1
    ...
end
```

- *Assignment0 must pause until gate is true (1) before r1 takes on value of data. Statement does not pause if gate already true*
- *Assignment1 must pause until gate is false (0) before r1 takes on value of data. Statement does not pause if gate already false*

always/initial Blocks (Behavioral Statements)



Instruções de comportamento (Behavioral)

Descreve o comportamento;
Deve ser utilizado dentro de um bloco procedural
Instruções de comportamento(Behavioral)

- **if-else**
- **case**
- **Loop**

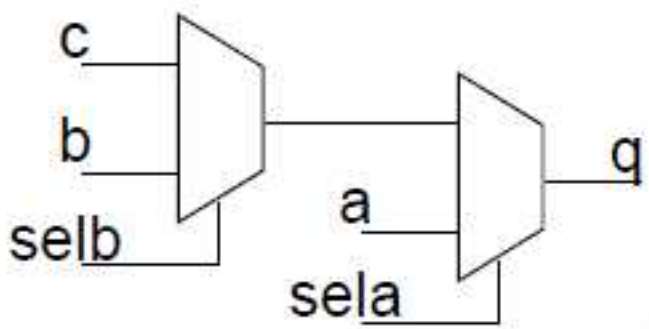
Instruções **if-else**

- Format:

```
if <condition1>
    {sequence of statement(s)}
else if <condition2>
    {sequence of statement(s)}
        ...
else
    {sequence of statement(s)}
```

- Example:

```
always @ (sela, selb, a, b, c) begin
    if (sela)
        q = a;
    else if (selb)
        q = b;
    else
        q = c;
```



Instruções **if-else**

Condições são avaliadas de cima para baixo
- Priorização

A primeira condição, que é **true**, causa a correspondente sequencia de instruções a serem executadas;
Se todas as condições ou alguma das condições são falsas, então a sequência de instruções associadas ao **else** são executadas

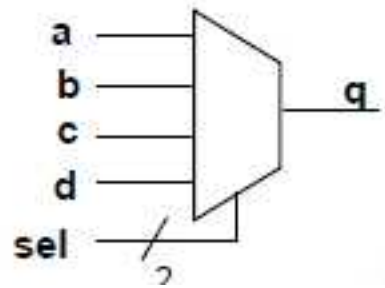
Instruções **case**

- Format:

```
case {expression}
    <condition1> :
        {sequence of statements}
    <condition2> :
        {sequence of statements}
    ...
    default : -- (optional)
        {sequence of statements}
endcase
```

- Example:

```
always @ (sel, a, b, c, d) begin
    case (sel)
        2'b00 : q = a;
        2'b01 : q = b;
        2'b10 : q = c;
        default : q = d;
    endcase
end
```



Instruções **case**

- As condições são avaliadas na ordem, o primeiro valor que for verdadeiro é o escolhido.
- Trata ambos **X** e **Z** como valores lógicos reais;
- Clausula default representa todos outros valores possíveis.
- Condições que não são especificamente previstas.
- **Verilog** não exige (no entanto,é recomendável) que:
  - Todas as condições sejam tratadas;
  - Todas as condições sejam únicas;

Instruções **case**

- **casez**
  - Trata ambos **Z** e **?** No caso de condições de *don't cares*
- **casex**
  - Trata ambos **X** e **Z** No caso de condições como *don't cares*, ao invés de valores lógicos

```
casez (encoder)
    4'b1??? : high_lvl = 3;
    4'b01?? : high_lvl = 2;
    4'b001? : high_lvl = 1;
    4'b0001 : high_lvl = 0;
    default : high_lvl = 0;
endcase
```

- if **encoder = 4'b1z0x**, then **high_lvl = 3**

```
casex (encoder)
    4'b1xxx : high_lvl = 3;
    4'b01xx : high_lvl = 2;
    4'b001x : high_lvl = 1;
    4'b0001: high_lvl = 0;
    default : high_lvl = 0;
endcase
```

- if **encoder = 4'b1z0x**, then **high_lvl = 3**

Instrução Loop

- **forever** loop - executa continuamente
- **repeat** loop - executa um número fixo de vezes;
- **while** loop - executa se a expressão é verdadeira;
- **for** loop - executa uma vez o inicio do loop e então continua executando se a expressão for verdadeira

  - Instrução Loop - usada para operações repetitivas.

Forever and repeat Loops

- **forever** loop - executa continuamente

```
initial begin
    clk = 0;
    forever  #25  clk = ~clk;
end
```

Clock with period of 50 time units

Not synthesizable!

- **repeat** loop - executa um número fixo de vezes

```
initial begin
    count = 0;
    while (count < 101) begin
        $display ("Count = %d", count);
        count = count + 1;
    end
end
```

Counts from 0 to 100
Exits loop at count 101

Not synthesizable!

While Loops

- **while loop** - executa se expressão é verdadeira

```
initial begin
    count = 0;
    while (count < 101) begin
        $display ("Count = %d", count);
        count = count + 1;
    end
end
```

Counts from 0 to 100
Exits loop at count 101

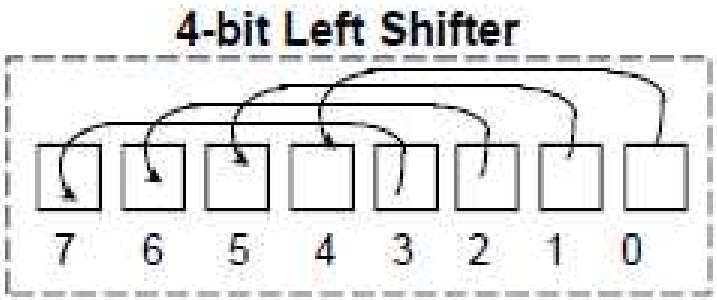
Not synthesizable!

for Loops

- **for** loop - executa uma vez no início e continua executando se a expressão for verdadeira.

```
// declare the index for the FOR LOOP
integer  i;

always @(inp, cnt) begin
    result[7:4] = 0;
    result[3:0] = inp;
    if (cnt == 1) begin
        for (i = 4; i <= 7; i = i + 1) begin
            result[i] = result[i-4];
        end
        result[3:0] = 0;
    end
end
```



Referências

- Curso oficial da Altera "Introduction to Verilog"
- Aula do curso "ASIC & FPGA Chip Design:HDL Coding" Dr. Mahdi Shabany