Curso básico de Verilog

Focado em código sintetizável



Objetivos do curso

- **Entender as diferenças de sintetizar um hardware e programar um software**
- **Sintetizar um projeto de hardware em Verilog.**
- **Boas práticas para sintetizar um hardware confiável.**
- **Interpretar resultados num ambiente de simulação.**
  - Num primeiro momento usaremos o Quartus II em todas as etapas do projeto, no entanto, ele não se mostra conveniente para simulações funcionais, nestes casos é interessante usar o software Modelsim, que fará parte da próxima etapa do curso.

Programar software e Sintetizar um hardware

- Programar software

Entendemos por **programar um software**:

A elaboração de um conjunto de instruções que serão compiladas a um nível de máquina que irá parametrizar um processador **genérico** e obter as saídas esperadas.

Note que este processador é um hardware construído para processar instruções de base (soma, funções lógicas e operações diversas entre bits).

Não há elaboração alguma de hardware,sim e somente, de software.

Programar software e Sintetizar um hardware

- Sintetizar um hardware

Entendemos por **sintetizar um hardware:**

A elaboração de um conjunto de instruções que serão interpretadas a fim de construir um hardware **dedicado** que responda aos estímulos de forma esperada.

Já neste caso este hardware dedicado pode operar qualquer tarefa projetada, após a definição desta tarefa, o hardware responderá da maneira que ele foi elaborado.

No fim podemos dizer que foi elaborado um hardware e ele possuirá características físicas após a confecção de um protótipo.

Programar software e Sintetizar um hardware



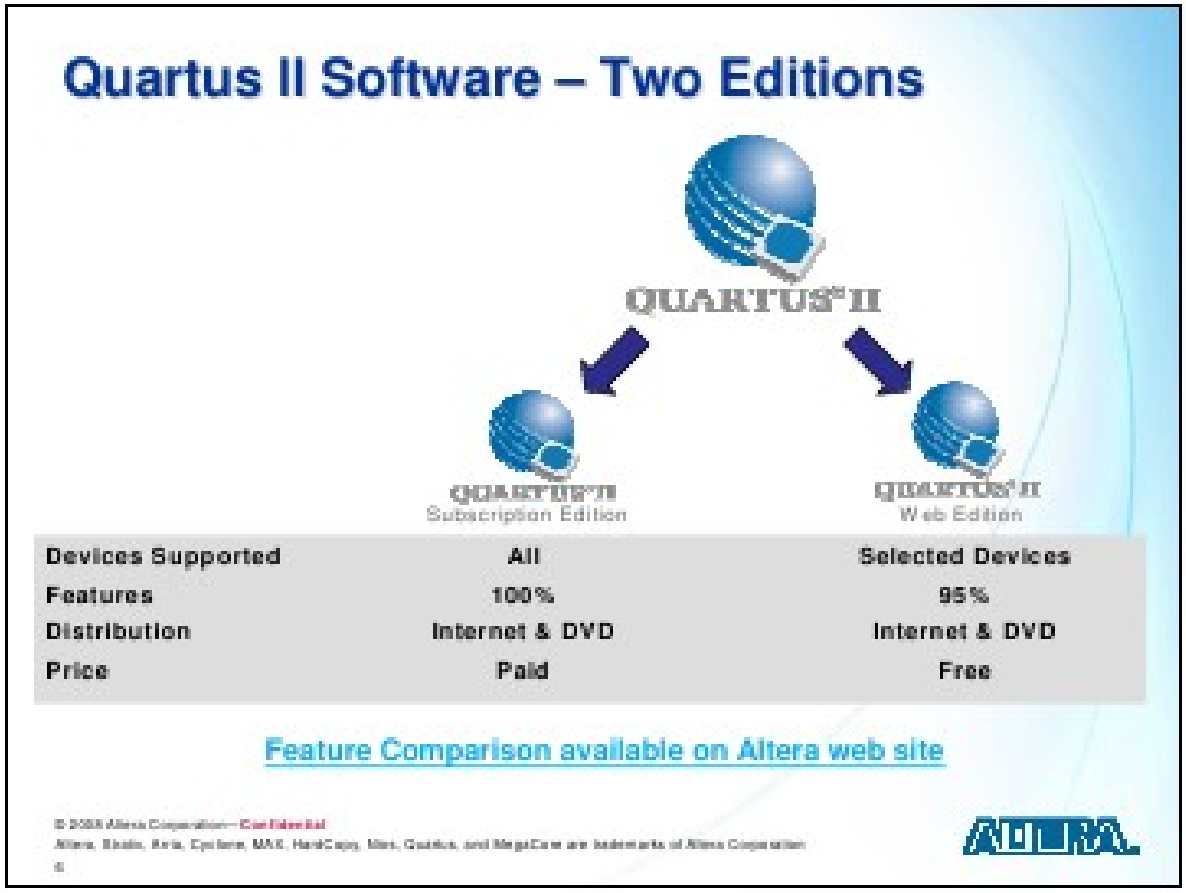
Anexo Altera Devices & Design Software



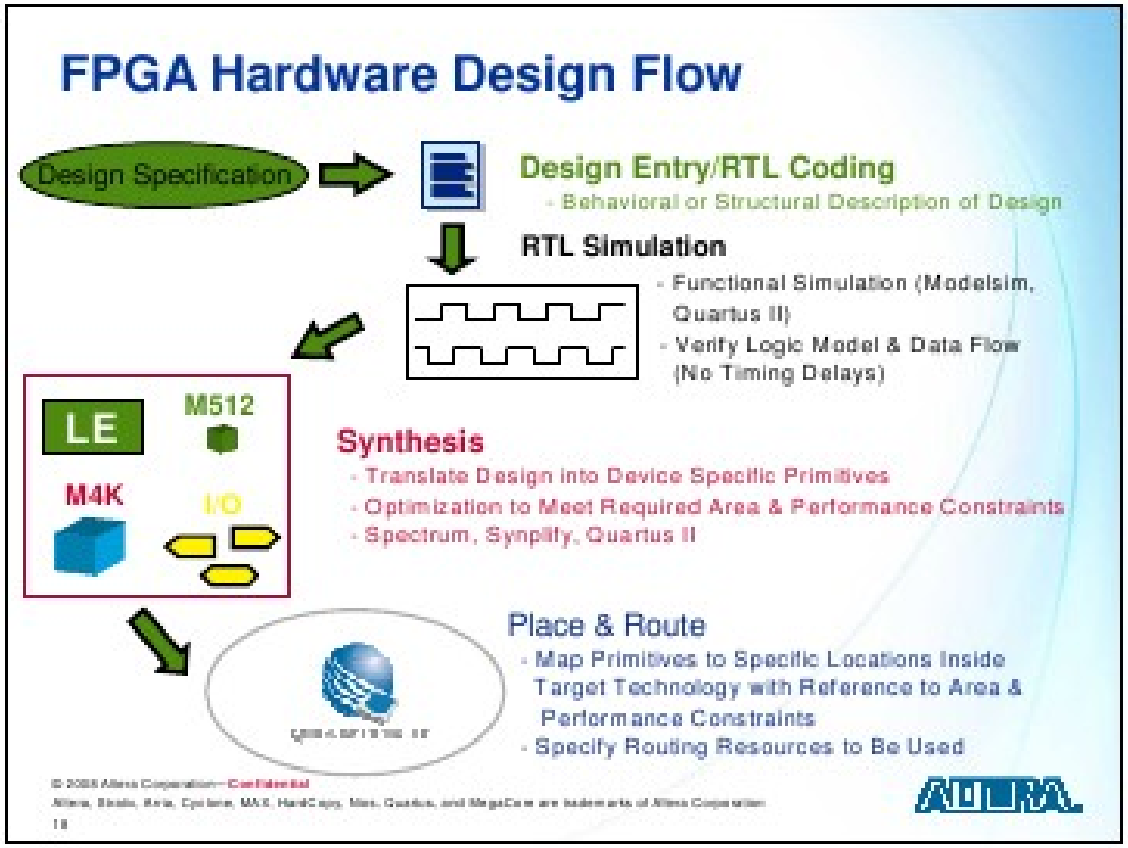
Anexo Altera Devices & Design Software



Anexo Altera Devices & Design Software



Anexo FPGA Hardware Design Flow



Verilog - Introdução e primeiros passos

O que é Verilog?
IEEE industry standard Hardware Description Language (HDL) – usada para descrever um sistema digital, para síntese e simulação.

Verilog - Introdução e primeiros passos

Histórico do Verilog:

Iniciado em **1984** com a Gateway Design Automation. Em **1989** Cadence comprou a Gateway (Verilog-XL simulator).

1990 a versão Cadence Verilog foi liberada para o público e Open Verilog International (OVI) foi criado para controlar as especificações da linguagem.

1993 OVI lançou a versão 2.0 e em seguida o IEEE adotou OVI Verilog como um padrão, chamado de Verilog 1364

Verilog - Introdução e primeiros passos

Principais caracterísiticas da sintaxe:

- CASE-sensitve
- Todas as palavras chaves são formatadas em minúsculo
- Espaços são utilizados para legibilidade do código
- Ponto e virgula (;) é utilizado para término de instrução
- Comentário de uma linha: //
- Comentário de diversas linhas: /* */

Verilog - Introdução e primeiros passos

Corpo do Projeto:

```
module nome (lista_portas);
// especificação de portas
// especificação de tipos de dados

// funcionalidade do circuito

endmodule
```

Verilog - Introdução e primeiros passos (Exemplo)

```
module body (in01,in02,out01);

input in01,in02;
output out01;

endmodule
```



Verilog - Introdução e primeiros passos

Portas:

Tipos:

input --> porta de entrada

output --> porta de saída

inout --> porta biredicional

Declaração de portas:

<tipo_de_porta> <nome_da_porta>;

- Exemplo:

```
input [7:0] ina, inb;
input clk, clr;
output [15:0] out;
```

Verilog - Introdução e primeiros passos

Tipos de dados:

Os “dados” num projeto de hardware se dividem em dois tipos básicos:

- **Net** : representa uma conexão física.
- **Register** : representa uma varíavel que será guardada temporariamente.

Verilog - Introdução e primeiros passos

Tipo de dado Net:

- wire --> representa uma interconexão simples
- tri --> representa uma interconexão tri-state (portas do top-level)

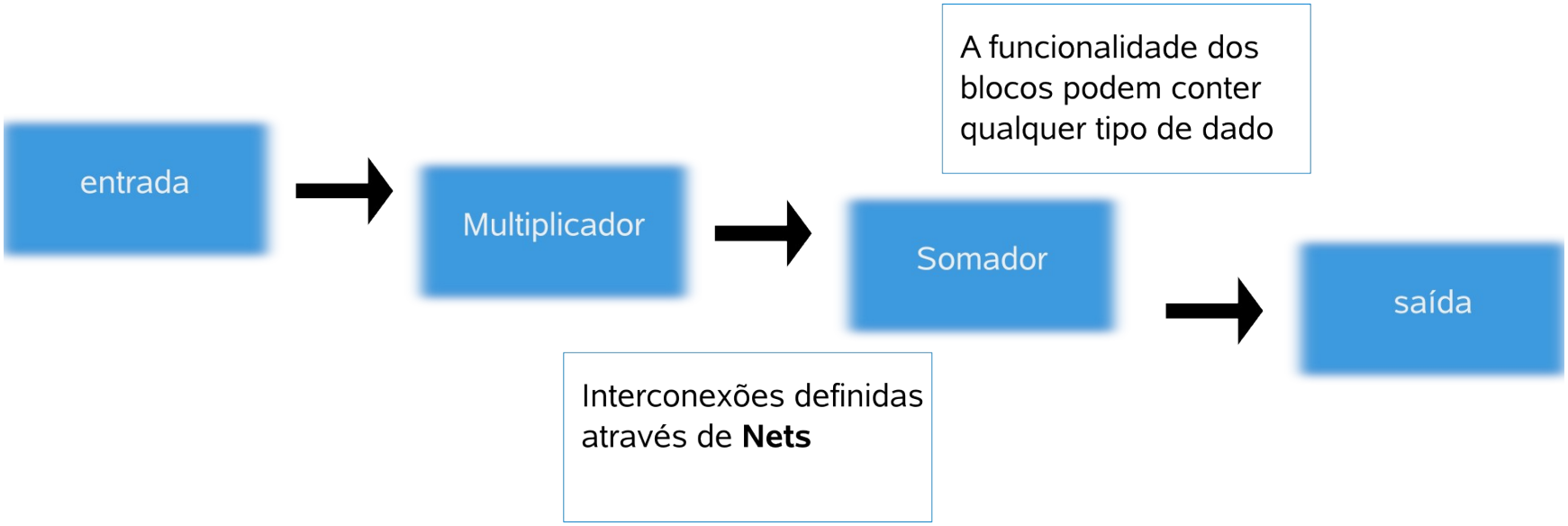
Declaração de barramentos:

- <tipo_de_dado> [MSB : LSB] <nome_do_sinal> ;
- <tipo_de_dado> [LSB : MSB] <nome_do_sinal> ;

Exemplos:

- wire <nome_do_sinal> ;
- wire [15:0] multiplicador_out, somador_out;

Verilog - Introdução e primeiros passos



Verilog - Introdução e primeiros passos

Tipo de dado Register:

- reg - variável com agrupamento de bits de qualquer largura
- integer - variável com um agrupamento de bits com sinal e com largura geralmente de 32 bits .

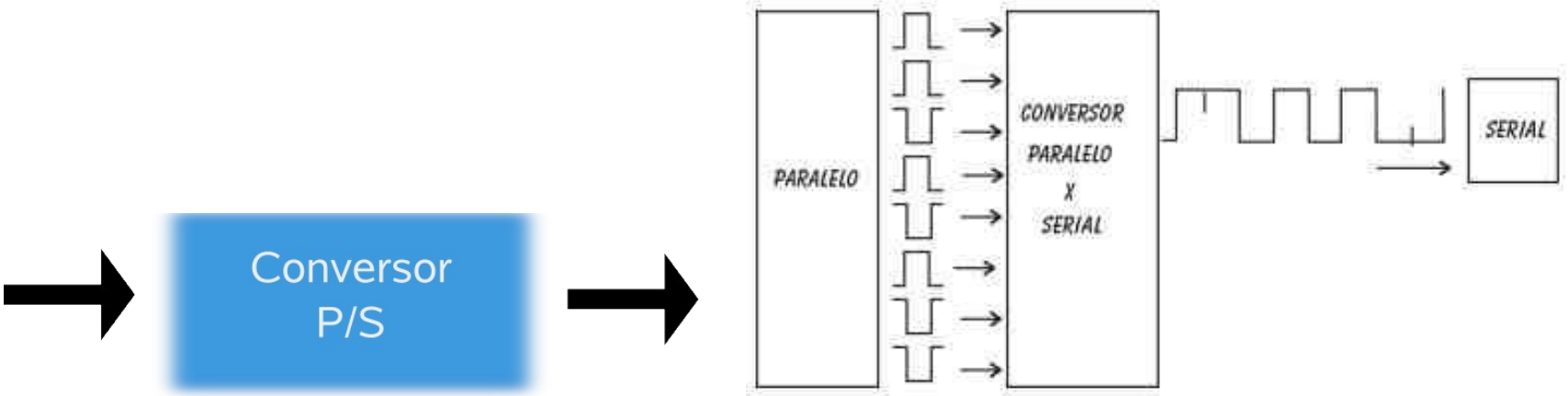
Declarações de barramentos:

- <tipo_de_dado> [MSB : LSB] <nome_do_sinal> ;
- <tipo_de_dado> [LSB : MSB] <nome_do_sinal> ;

Exemplos:

- reg <signal name> ;
- reg [7 : 0] out ;

Verilog - Introdução e primeiros passos



No interior do processo P/S deve-se guardar os bits de entrada e em seguida entregá-los serialmente na saída

Verilog - Introdução e primeiros passos

Memória:

Tipos de dado Register podem ser agrupados a fim de formar uma memória.

Exemplo:

```
reg [31:0] mem[0:1023]; // 1Kx32
reg [31:0] instr;
instr = mem[2];
```

Verilog - Introdução e primeiros passos

Parâmetro:

Atribuição de um valor a um símbolo

Exemplo:

```
parameter size = 8;
reg [size-1:0] a, b;
```

Atribuindo valores - números

- Especificação do número de bits do número:

<nº de bits>'<base ><número>

- Bases possíveis: decimal ('d ou 'D), binária ('b ou 'B), hexadecimal ('h ou 'H), octal ('o ou 'O).
- Ex.: 3'b010 = número binário de 3 bits

b010 = 32'b010 → número binário de 32 bits (32 é *default*)

10 = 32'd10 → número decimal de 32 bits (decimal é *default*)

32'd2 = 32'b10 = 2;

2'd7 = 2'd3 pois: 7 → 1 1 1 (o último bit é truncado!!)

Números

- Números negativos: especificados através do sinal menos (-) antes do < nº de bits >:
  - **CERTO:** -8'd3 = número negativo de 8-bits, armazenado como "complemento de 2" de 3.
  - **ERRADO:** 4'd**-**2 = ERRO!!
- Caracteres especiais para números:
  - '_' (*underscore*): usado para legibilidade.

Exemplo: 32'h21_65_bc_fe = 32'h2165bcfe

  - x' ou 'X' (valor desconhecido)

Exemplo: 12'h12**x** = número hexadecimal de 12 bits com o LSB (bit menos significativo) desconhecido.

  - 'z' ou 'Z' (valor de alta impedância)

Exemplo: 1'b**z** = número de alta impedância de 1 bit.

Números

- Se o MSB (bit mais significativo) for 0, x ou z, o número é preenchido com 0, x ou z, respectivamente.
  - Exemplo: 3'b01 = 3'b001,

    3'bx1 = 3'bxx1,

    3'bz = 3'bzzz.
- Se o MSB for 1, o número é preenchido com 0.
  - Exemplo: 3'b1 = 3'b001.

DESAFIO

- Qual é o resultado da seguinte atribuição a = 4’d017 na base binária?

DESAFIO

- Qual é o resultado da seguinte atribuição a = 4’d017 na base binária?

- Resposta:

4’b0001, pois o MSB é truncado (10001).

Operadores Aritméticos

Símbolo	Função	Exemplo (a = 5; b = 10, c = 2'b01)
+	adição	b + c = 11
-	subtração, negativo	b – c = 9; -b = -10
*	multiplicação	a * b = 50
/	divisão	b / a = 2
%	modulus	b % a = 0

- Trata vetores como um valor inteiro;
- Se algum operando é Z ou X, o resultado é desconhecido.

- Exemplo: a + 2'b0Z = desconhecido;

a + 2'd0X = desconhecido.

Operadores bit a bit

Símbolo	Função	Exemplo (a = 3'b101; b = 3'b110; c = 3'b01X)
~	inverte cada bit	~a = 3'b010
&	*and* cada bit	a & b é 3'b100; b & c é 3'b010
|	*or* cada bit	a | b é 3'b111
^	*Xor* cada bit	a ^ b é 3'b011
^~ ou ~^	*Xnor* cada bit	a ^~ b é 3'b100

- Opera em cada bit do operando;
- Resulta no número de bits do maior operando (completa com 0 para a esquerda).

- Exemplo: 3'b100 | 1'b1 é 3'b101

Operadores de Redução

Símbolo	Função	Exemplo (a = 5'b10101, b = 4'b0011 c = 3'bZ00, d = 4'bX011)
&	*and* todos os bits	&a = 1'b0 ; &d = 1'b0
~&	*nand* todos os bits	~&a = 1'b1
|	*or* todos os bits	|a = 1'b1; |c = 1'bX
~|	*nor* todos os bits	~|a = 1'b0
^	*Xor* todos os bits	^a = 1'b1
~^ ou ^~	*Xnor* todos os bits	~^a = 1'b0

- Reduz o vetor a um único bit;
- X e Z são considerados desconhecidos mas podem resultar em valores conhecidos.

-Exemplo: &d = 1'b0

Operadores Relacionais

Símbolo	Função	Exemplo (a = 3'b010, b = 3'b100, c = 3'b111 d = 3'b01z, e = 3'b01x)
>	Maior que	a > b resulta falso (1'b0)
<	Menor que	a < b resulta verdade (1'b1)
>=	Maior que ou igual	a >= d resulta desconhecido (1'bX)
<=	Menor que ou igual	a <= e resulta desconhecido (1'bX)

- Usado para comparar valores;
- Retorna 1 bit escalar : 0 para falso e 1 para verdade;
- Se algum operando é X ou Z, então o resultado é desconhecido.

Operadores de Igualdade

Símbolo	Função	Exemplo (a = 3'b010, b = 3'b100, c = 3'b111 d = 3'b01z, e = 3'b01x)
==	Igualdade	a == c resulta falso (1'b0)
!=	Desigualdade	e != e resulta desconhecido (1'bX)

- Usado para comparar valores;
- Retorna 1 bit escalar : 0 para falso e 1 para verdade;
- Se algum operando é X ou Z, então o resultado é desconhecido.
- Caso igualdade e caso desigualdade incluem X e Z.

Operadores Lógicos

Símbolo	Função	Exemplo (a = 3'b101, b = 3'b000)
!	Não verdade	!a é falso (1'b0)
&&	Duas expressões verdade	a && b resulta falso (1'b0)
||	Uma ou duas expressões verdade	a || b resulta verdade (1'b1)

- Usado para comparar valores;
- Retorna 1 bit escalar : 0 para falso e 1 para verdade;
- Se algum operando é X ou Z, então o resultado é desconhecido.

Operadores de Shift

Símbolo	Função	Exemplo (a = 4'b1010, b = 4'b10X0)
>>	Shift para direita	b >> 1; resulta em 4'b010X
<<	Shift para esquerda	a << 2; resulta em 4'b1000

- Desloca o vetor para direita ou para a esquerda, de acordo com o número de bits especificado;
- Completa com zeros;
- Bits deslocados são perdidos.

Operadores Diversos

Símbolo	Função	Exemplo
?:	Condicional	(condição) ? true_val : false_val; assign saida = (v ==2'b01) ? A : B;
{ }	Concatenação	a = 3'b010, b = 4'b1100 {a,b} resulta 7'b0101100
{{ }}	Replicação	{3{2'b10}} resulta 6'b101010

Ordem de precedência

+ , - , ! , ~

<< , >>

< , > , <= , >=

== , !=

&

^ , ^~ or ~^

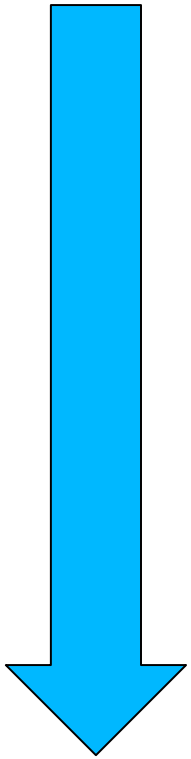
|

&&

||

?:

Prioridade máxima



Obs.: () podem ser usados para interferir na ordem default.

Exemplo

```
module exemplo();


wire [4:0] a = 5'b10110;
wire [4:0] b, c, d, e;

wire x = 0, y = 1;
wire [1:0] z, w;



assign b = ~a;

assign c = a - b;

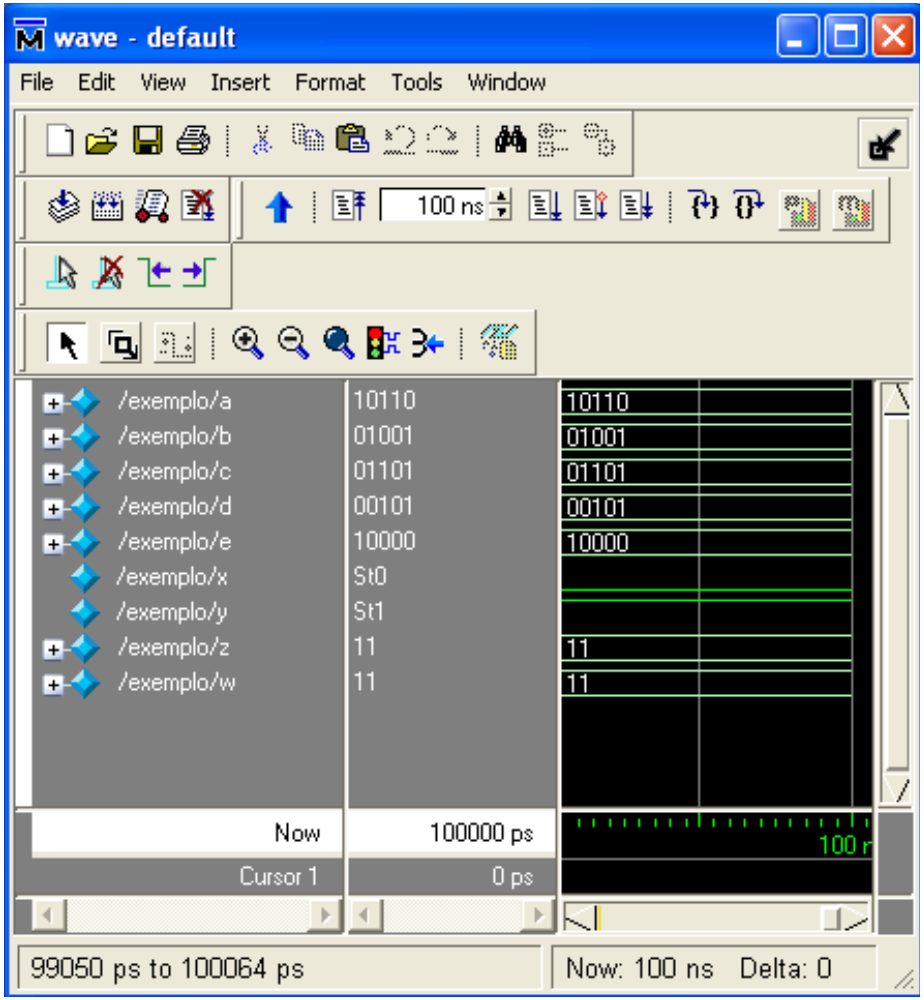
assign d = a>>2;

assign e = a<<3;

assign z = (x == 0)? 2'b11:2'b00;

assign w = ((x == 0) && (y == 1))? 2'b11:2'b00;

endmodule
```



Modelando a funcionalidade

Continous assignment e Procedural blocks

Continous assigment

Verilog – Modelando a funcionalidade

Existem duas vertentes para descrição da funcionalidade do circuito em Verilog:

- Continous assignment : Atribuição contínua, a funcionalidade não varia, não possui um comportamento. Ex.: um ganho para um determinado sinal.

- Procedural blocks: Blocos de processos, cria-se um processo que definirá uma ação, em que a mesma, pode possuir um comportamento variável mediante entrada, saídas e condições internas.

Verilog – Continous assignment

É utilizado para combinações entre sinais, estas combinações não variam com o tempo, isto é, contínuas.

Podem ser declaradas no momento da declaração do tipo de dado:

```
wire [15:0] adder_out = mult_out + out;
```

ou no decorrer do corpo do código, declarando um assigment:

```
assign adder_out = mult_out + out;
```

Verilog – Continous assignment

Principais características:

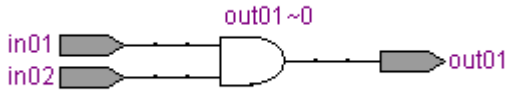
1) Lado esquerdo da atribuição precisa ser do tipo de dado **Net**
2) A atribuição é continuamente atualizada, ou seja, se qualquer ítem do lado direito da atribuição for modificado, instantâneamente o lado esquerdo é atualizado.
3) O lado direito pode ser de qualquer tipo (**Net** ou **Register**)

Verilog – Continous assignment (Exemplos)

Um exemplo do que foi aprendido até então:

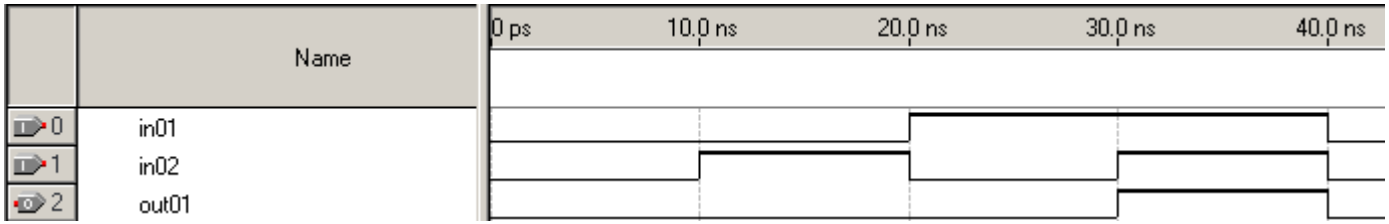
module **logicalassign** (**in01,in02,out01**);

input **in01,in02**;
output **out01**;



assign **out01** = **in01 & in02**;

endmodule



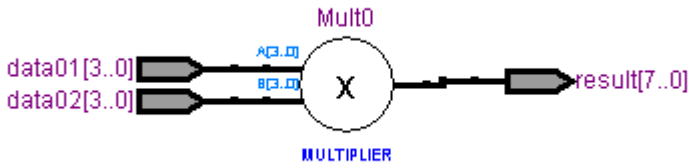
Verilog – Continous assignment (Exemplos)

```
module multiplicator (data01,data02,result);

input signed [3:0] data01,data02;
output signed [7:0] result;

assign result = data01 * data02;

endmodule
```



Verilog – Continous assignment (Instâncias)

Com o intuito de organizar o código e divisão de tarefas, podemos separar partes do projeto em blocos e instancia-los no maior nível de hierarquia do código.

Exemplo:

```
module soma(in_a,in_b,out);
input [3:0] in_a,in_b;
output [4:0] out;
wire out = in_a + in_b;
endmodule
```

```
module mult(in_a,in_b,out);
input [1:0] in_a,in_b;
output [3:0] out;
wire out = in_a * in_b;
endmodule
```

Verilog – Continous assignment (Instâncias)

Podemos instanciar estes componentes quantas vezes forem necessárias para a formação da tarefa desejada.

```
module toplevel (dado01,dado02,resultado);
input [1:0] dado1,dado2;
output[3:0] resultado;
wire ent01 = dado01;
wire ent02 = dado02;
mult multiplicador (.in_a(ent01), .in_b(ent02), .out(saida));
wire saida = resultado;
endmodule
```

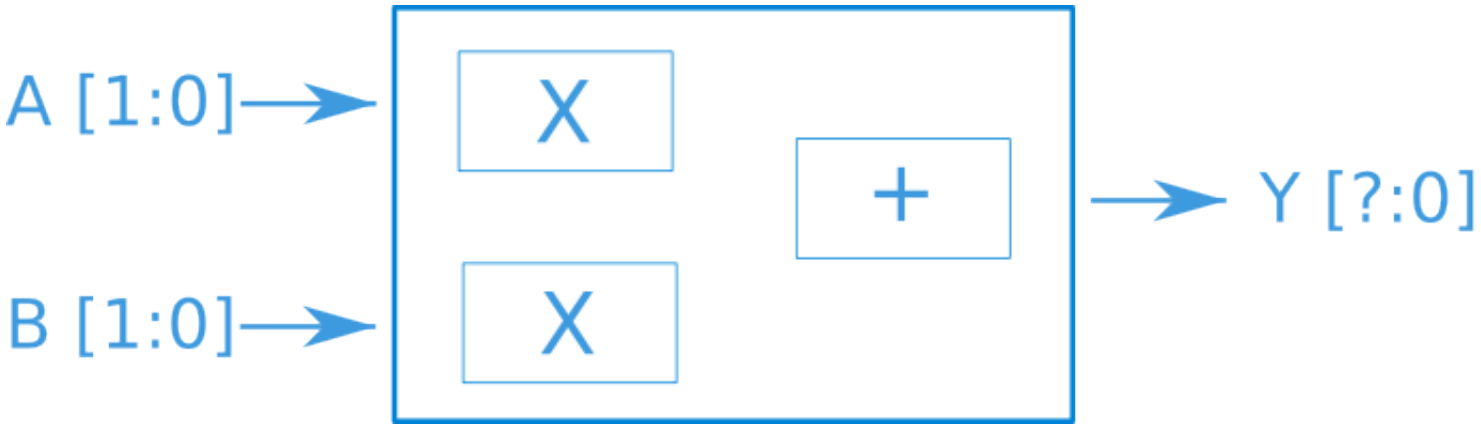
Exercício de número 1

Instâncias de somadores e multiplicadores a fim de gerar uma saida: $Y = A^2 + B^2$

Verilog – Exercício de número 01

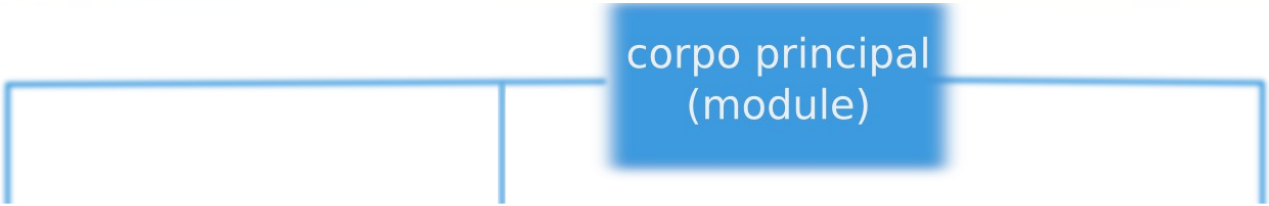
O objetivo do exercício é internalizar o conceito de instâncias e constatar instantaneidade da resposta no modo de descrição de funcionalidade continous assigment.

As principais características podem ser inferidas a partir da figura a seguir:

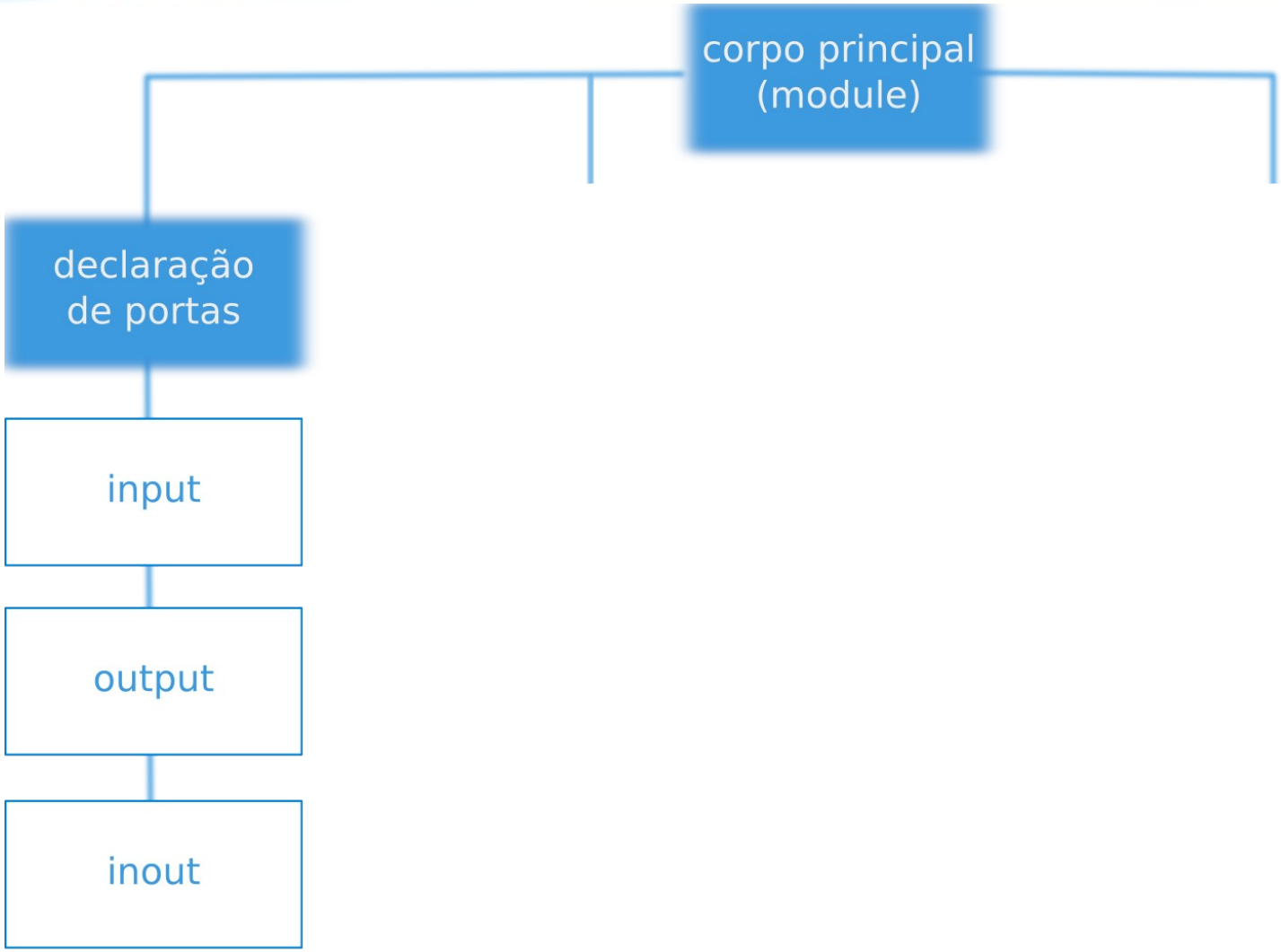


Revisão do que já foi abordado

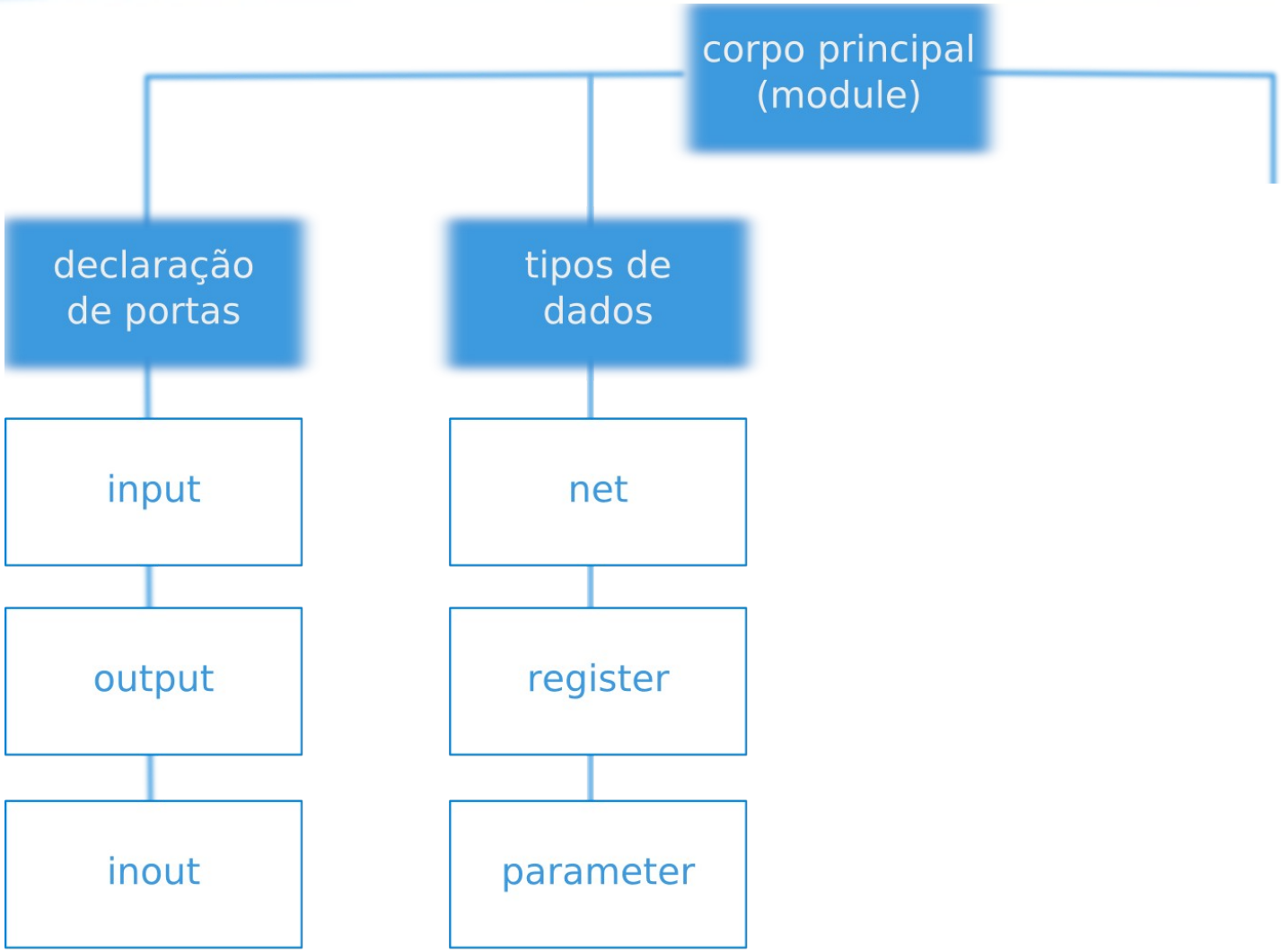
Verilog – Revisão



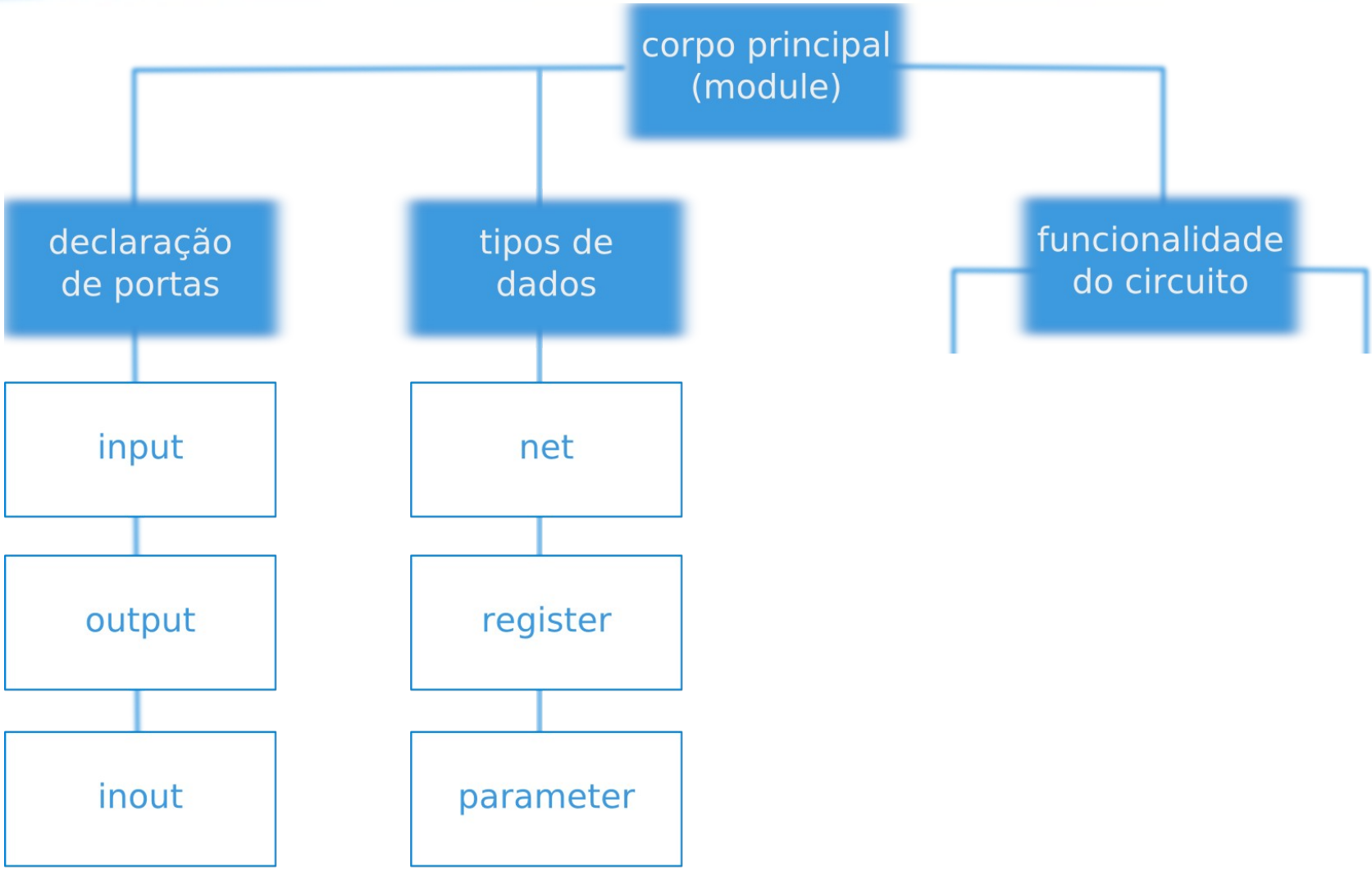
Verilog – Revisão



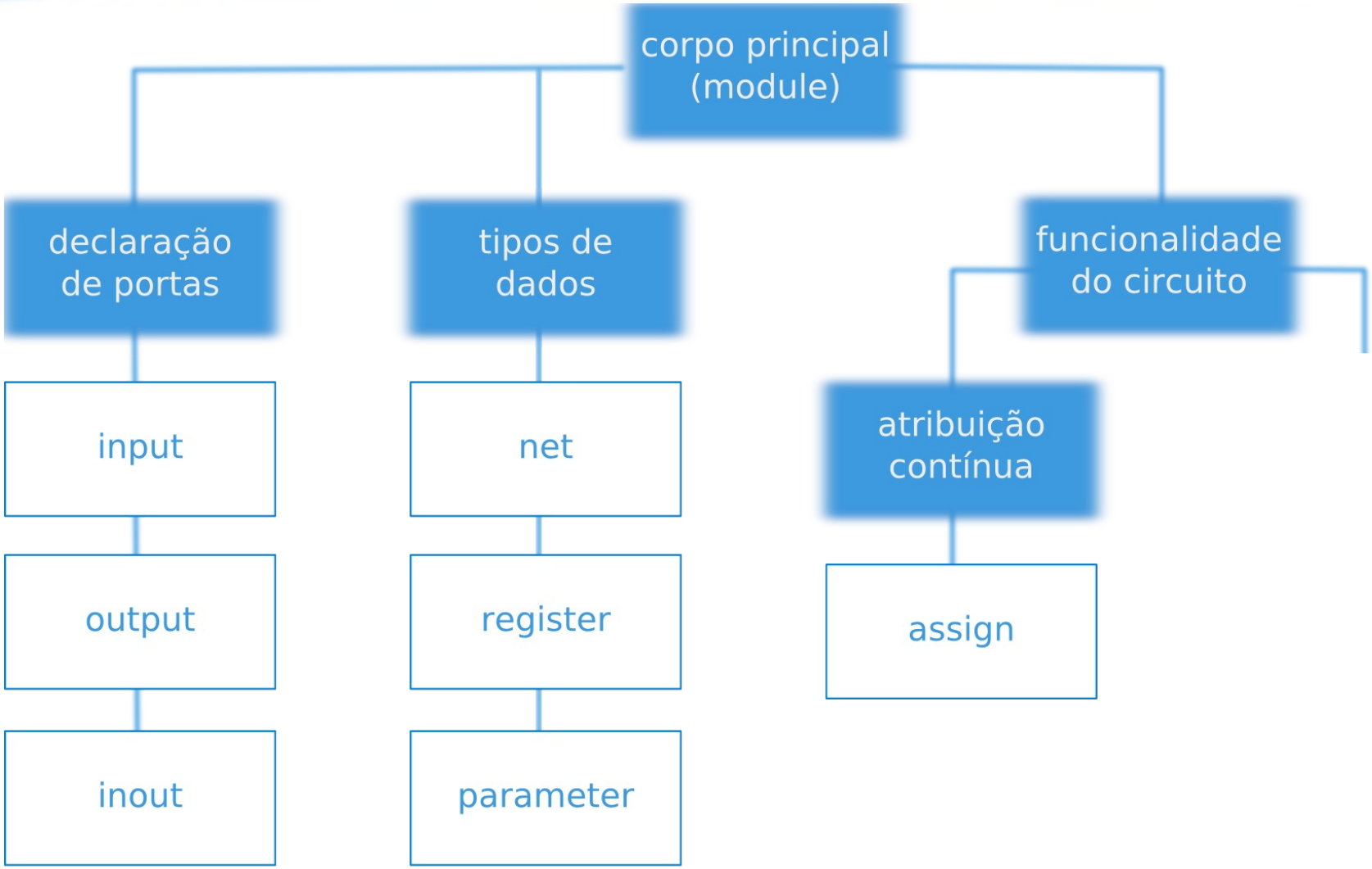
Verilog – Revisão



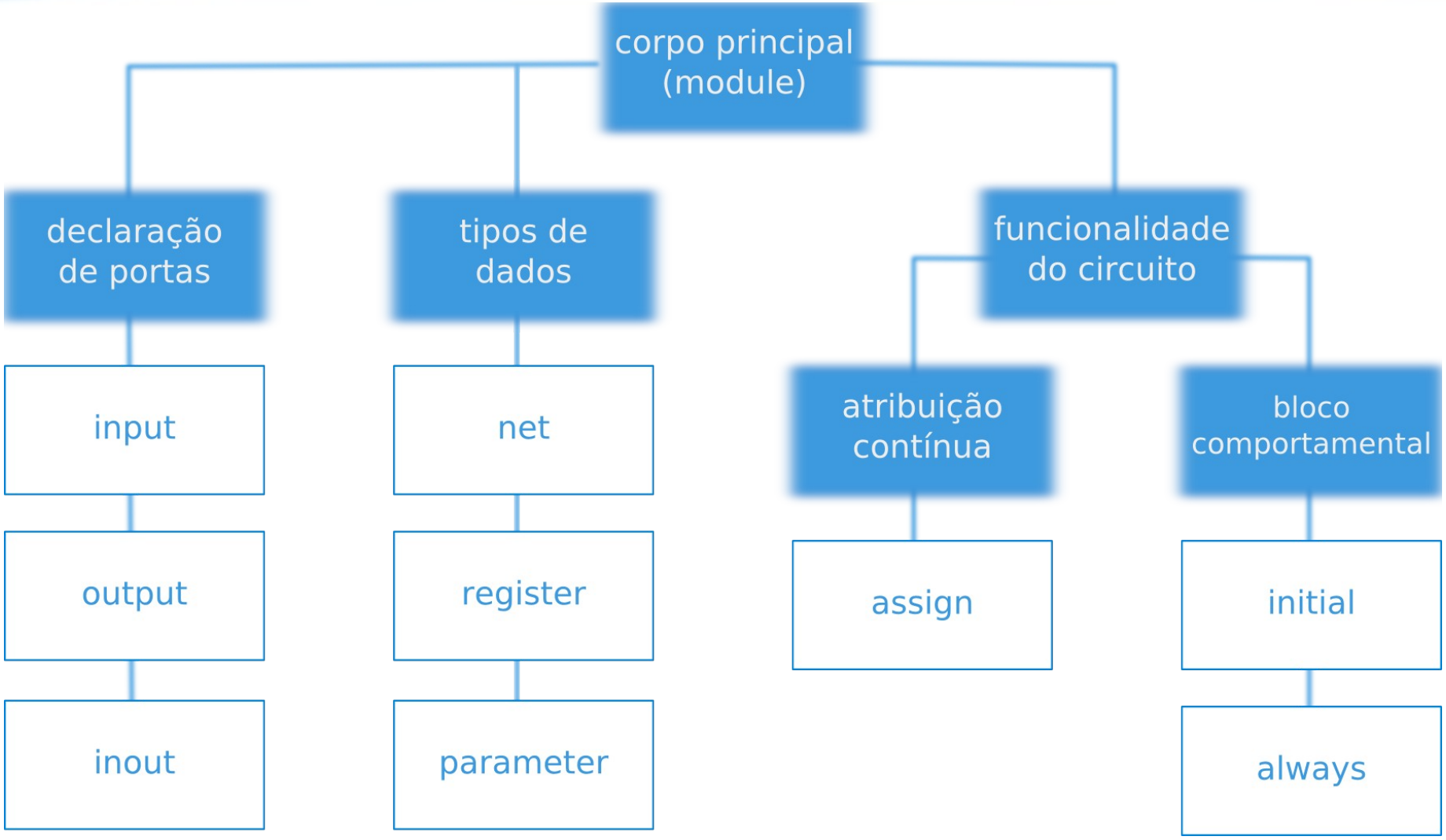
Verilog – Revisão



Verilog – Revisão



Verilog – Revisão



Procedural Blocks

Blocos de processos, que possuem comportamento.

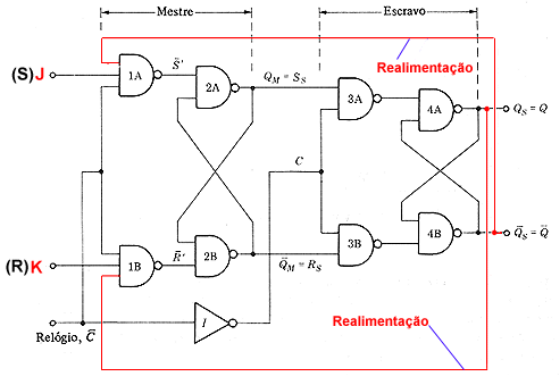
Blocos comportamentais.

Verilog – Procedural Blocks

Conforme dito anteriormente, Procedural blocks é utilizado para ações que possuam um comportamento definido.
Ex.: Blocos que dependam da variação de um determinado sinal para a atualização da saída, blocos sincronizados com a borda de um outro determinado sinal (clk).



- Bloco initial (não sintetizável, utilizado para simulação)
- Bloco always (sintetizável, utilizado para especificação de funcionalidade



Verilog – Initial blocks

O bloco do tipo initial é utilizado para definir condições iniciais, já que todos estes blocos iniciam no tempo 0.

Blocos do tipo initial não são sintetizáveis, conferindo a eles uso exclusivo para simulação.

É importamente notar que ele não possui lista de sensibilidade, isto se deve ao fato dele não repetir em nenhum momento.

Verilog – Always blocks

De uma forma geral, listaremos no bloco Always as regras de comportamento do nosso bloco funcional.

Cada always criado no código representa a descrição de um comportamento que se repete com o tempo, resultando em vários processos que rodam em paralelo.

Todos os processos começam ao mesmo tempo, no entanto, o desenrolar de cada um dependerá da lógica inferida no código.

Verilog – Always blocks

Principais características:

1) Lado esquerdo da atribuição precisa ser do tipo Register (reg, integer ou memory).
2) O lado direito pode ser somente uma parte do sinal, isto é, uma parte do barramento.
3) O lado direito da atribuição pode ser de qualquer tipo.

Verilog – Always blocks

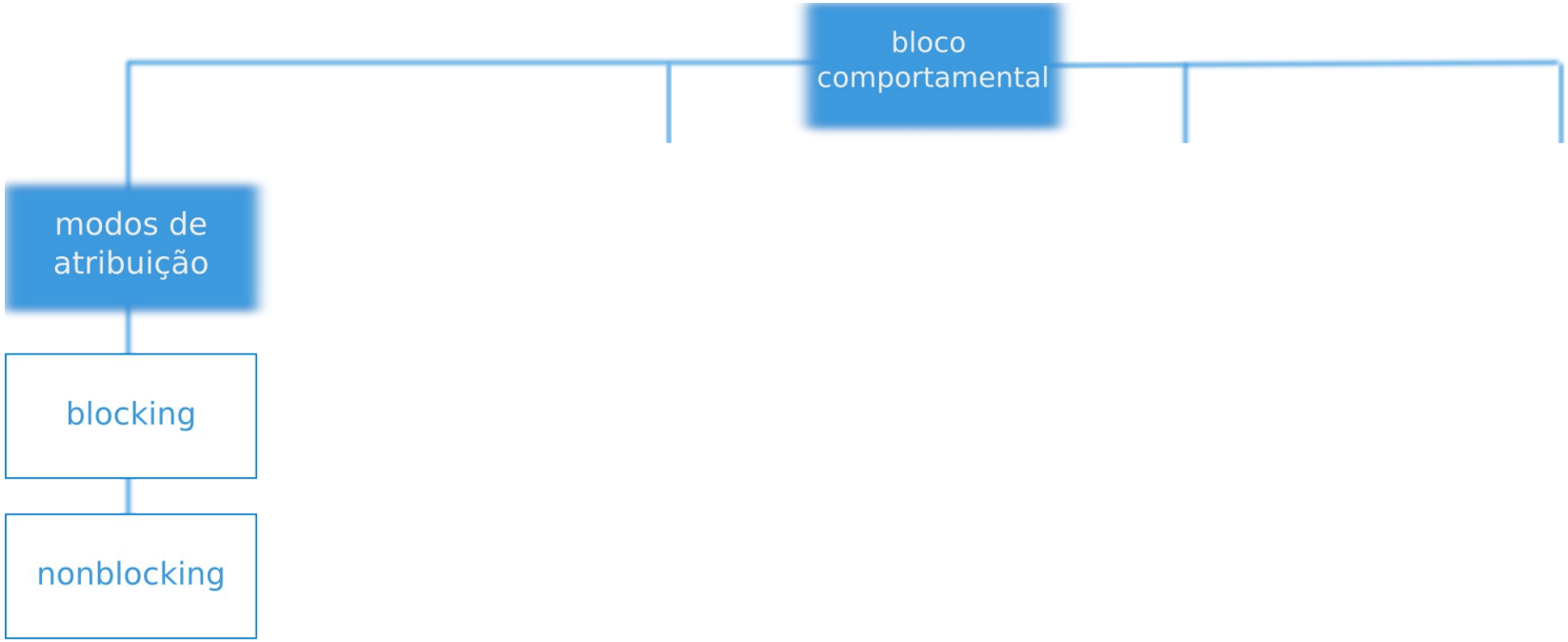
Dentro de Procedural blocks temos ainda quatro seções que devemos estudar:

- Modos de atribuição
- Lista de sensibilidade do processo
- Declarações de comportamento
- Modo de execução do bloco

Verilog – Blocos comportamentais (processos)



Verilog – Modos de atribuição



Verilog – Modos de atribuição

A atribuição pode ser feita de duas maneiras:

- Blocking (=)
- Nonblocking (<=)

Lembrando do início do curso, nosso código representa um conjunto de instruções para a criação do hardware. Os dois modos de atribuição influenciam somente na maneira que o compilador irá interpretar estas instruções.

Verilog – Blocking

Usando atribuição Blocking, como o compilador irá interpretar nosso código?

O compilador irá procurar a primeira atribuição, neste momento ele bloqueia a interpretação do código, resolve o lado direito da atribuição e em seguida carrega o lado esquerdo com a solução. No término desta sequencia ele pula para a próxima atribuição.

Blocking, se refere a este bloqueio que existe entre uma linha e outra de atribuição.

Verilog – Nonblocking

Usando atribuição nonblocking, como o compilador irá interpretar nosso código?

O compilador primeiramente irá conferir o lado direito de cada atribuição, e em cada uma delas, irá resolver a expressão associada. Depois de ele resolver todas estas pendencias e possuir a descrição destes trechos de funcionalidade, ele fará que todos os lados esquerdos de atribuição recebam suas respecivas soluções ao mesmo tempo (em paralelo)

nonblocking, se refere a ausência do bloqueio entre uma linha e outra de atribuição.

Verilog – Blocking X Nonblocking

- Blocking

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

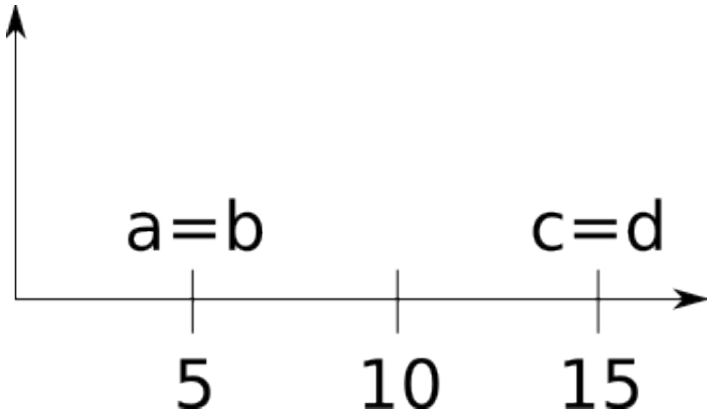
- Nonblocking

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

Verilog – Blocking X Nonblocking

- Blocking

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



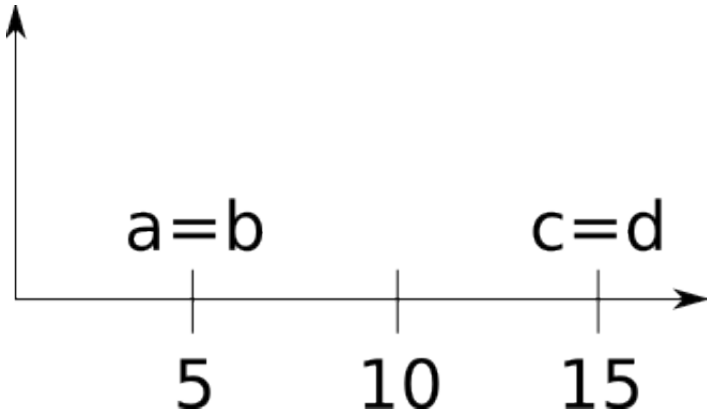
- Nonblocking

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

Verilog – Blocking X Nonblocking

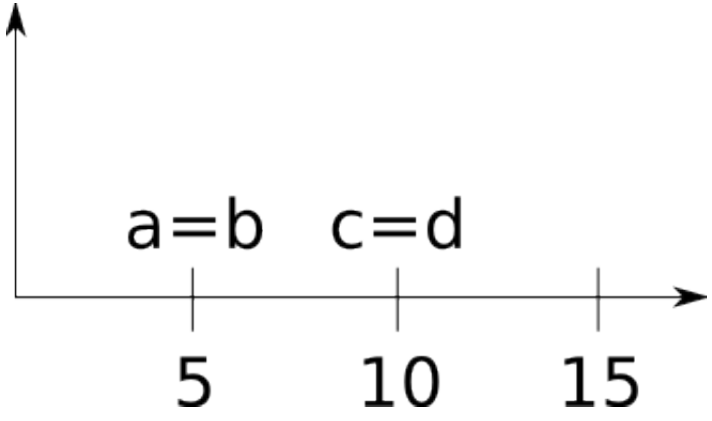
- Blocking

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



- Nonblocking

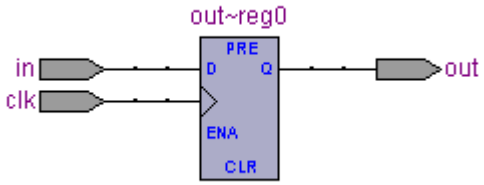
```
initial
  begin
       #5  a <= b;
       #10 c <= d;
   end
```



Verilog – Efeito do modo de atribuição

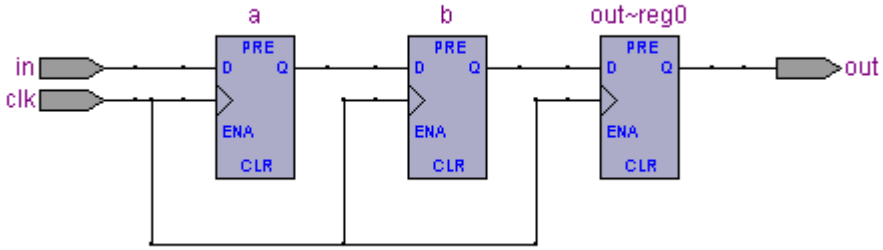
Blocking

```
always @ (posedge clk)
begin
    a = in;
    b = a;
    out = b;
end
```



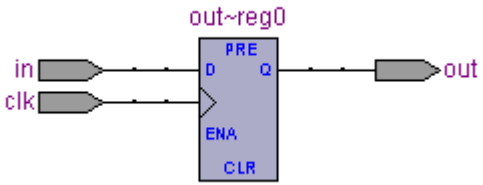
Nonblocking

```
always @ (posedge clk)
begin
    a <= in;
    b <= a;
    out <= b;
end
```

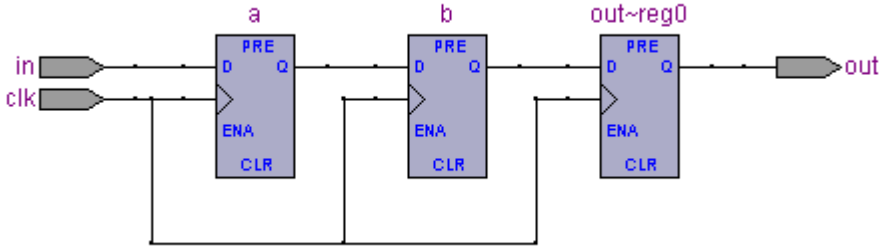


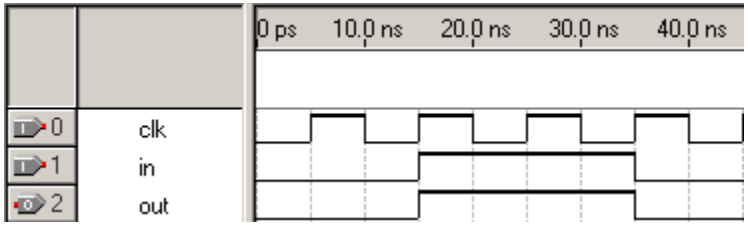
Verilog – Efeito do modo de atribuição

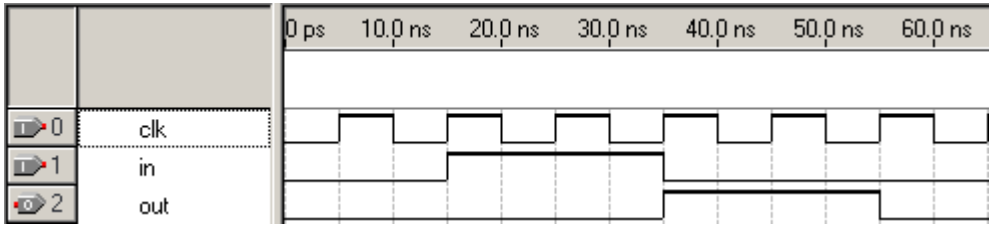
Blocking



Nonblocking







Verilog – A idéia do Fruto

A idéia é:

“ O que é fruto de um evento não pode interferir no mesmo”

Vejamos um exemplo:

Se uma maçã é fruto de uma determinada semente, será que a maçã já formada pode interferir na semente que a criou?

No entanto, depois da formação da maçã, será que ela pode gerar outras sementes resultando em outras maçãs?

Verilog – A idéia do Fruto

Na idéia da maçã, podemos chamar o desenvolvimento de uma maçã a partir de uma semente, de EVENTO.

Em seguida podemos chamar a maçã já formada de FRUTO, neste sentido fazemos a seguinte análise:

Num primeiro evento a maçã 1 foi fruto do desenvolvimento da semente1, e por isso mesmo a maçã 1 não pode interferir no primeiro evento.

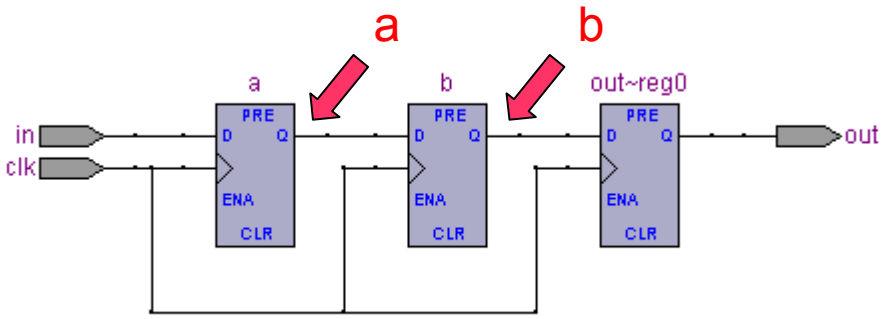
Embora depois de já formada ela gerou sementes, que num determinado segundo evento a maçã 1 não é mais fruto do segundo evento e consequentemente pode interferir no mesmo, como resultado visível na natureza, podemos observar que a semente gerada por essa maçã cria de fato uma segunda maçã.

Concluindo que esta maçã não pode interferir no evento que a criou, no entanto pode interferir num evento seguinte, em que, ela já é “adulta”.

Verilog – A idéia do Fruto

Saindo do pomar...

Vejamos o caso de três atribuições seguidas de registadores:



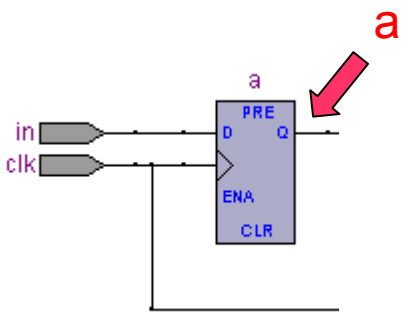
Em que, in e out, são entradas e saídas externas ao chip e o sinal denominado “a” está na saída do registrador a, de forma semelhante occorre com o sinal b.

Outro detalhe interessante é notar que o sinal de clock chega simultaneamente nos três registradores.

Verilog – A idéia do Fruto

Vamos para análise deste caso:

Isolando a parte que nos interessa, isolamos os sinais de in,a e clk.



Primeiramente devemos nos perguntar, que é o FRUTO?

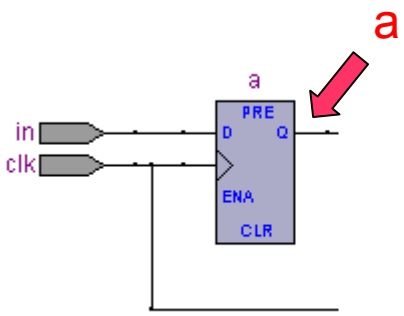
Neste contexto entendemos por fruto o nível lógico dos sinais, resultantes, em cada instante do tempo, nesta análise será o sinal “a”.

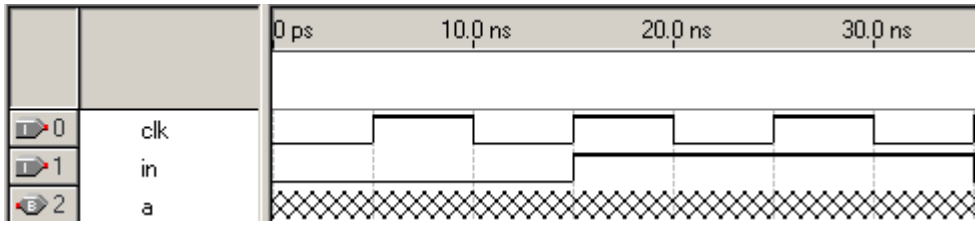
Em seguida devemos nos perguntar, o que é o EVENTO? Entendemos como evento aquele que gera o fruto, neste sentido devemos nos perguntar quem modifica o sinal “a”?

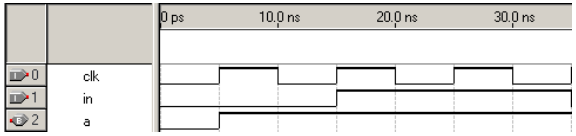
Verilog – A idéia do Fruto

Neste tipo de circuito sabemos que o flip-flop altera o valor da saída nas bordas de transição de nível do clock, neste exemplo foi adotado as bordas de subida do clock. Desta forma as bordas postivas é o EVENTO.

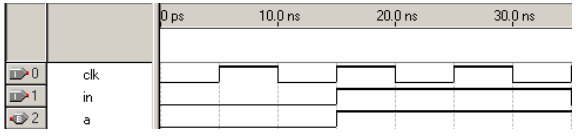
A partir dos sinais a seguir o que podemos concluir, sobre o sinal "a" ?



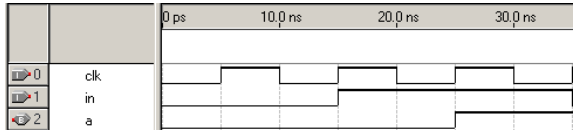




(a)



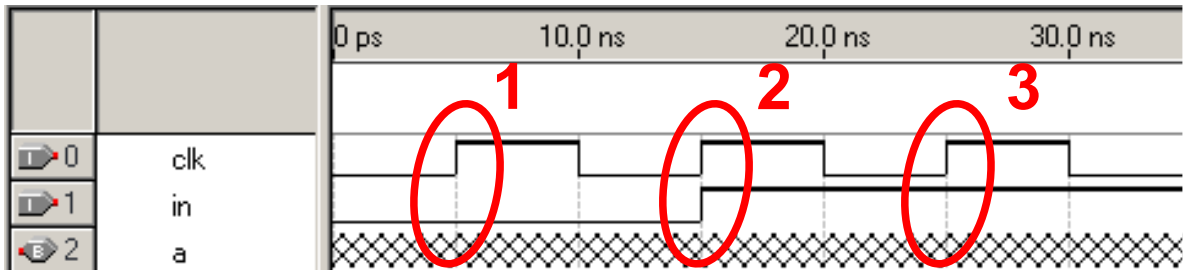
(b)



(c)

Verilog – A idéia do Fruto

De posse do conhecimento do EVENTO e do FRUTO podemos analisar a simulação.

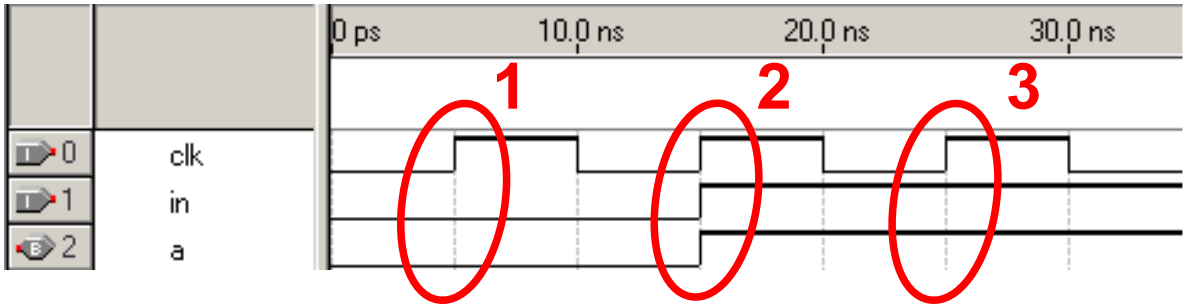


Devemos nos perguntar o sinal “in”, após a transição N é fruto da mesma transição. A resposta é NÃO, já que o sinal de entrada é externo e independe do clock. Se ele não é fruto da transição N, ele pode interferir na mesma, desta forma concluímos que a transição de nível da entrada chega instantaneamente na saída do registrador.

Essa transição de nível da entrada somente ocorre na transição 2 do clock, consecutindo somente alteração do nível de “a” nesta mesma transição.

Verilog – A idéia do Fruto

Segue o resultado correto da simulação:

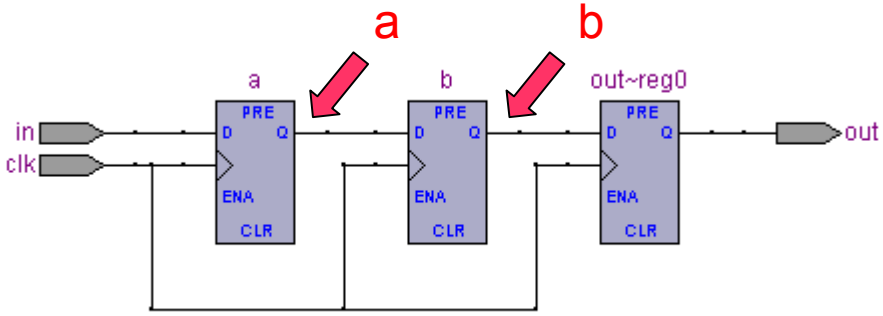


Notamos que o resultado condiz com o que era esperado, e somente na transição 2 do clock ocorreu mudança no sinal “a” e deve-se notar que apesar da existência do registrador, o sinal de saída foi atualizado na mesma transição do clock que o sinal de entrada foi alterado.

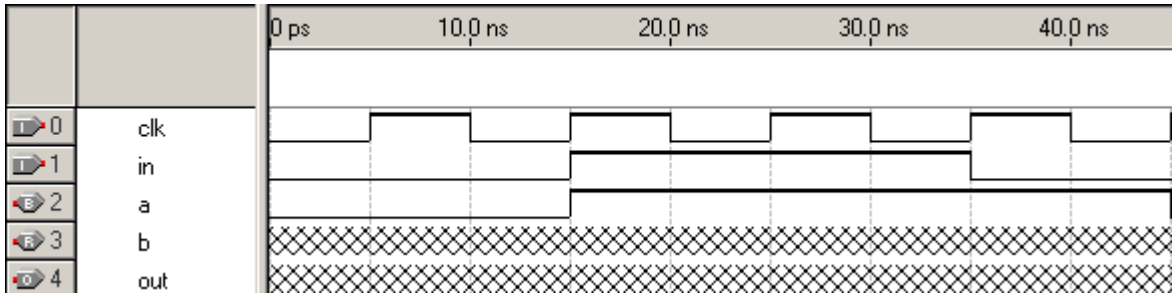
Isto é concluído pelo simulador pelo fato da entrada não depender do clock, e consequentemete esta transição ocorre num pequeno tempo (dt) antes do término da transição do mesmo.

Verilog – A idéia do Fruto

Neste momento podemos voltar para o circuito completo, o mesmo analisado na seção Modos de atribuição.

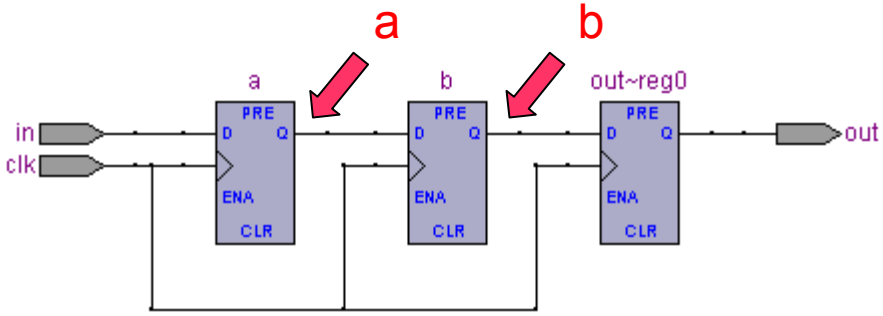


Quais serão os resultados esperados para “b” e “out”, deve-se salientar que agora possuem sinais de entrada, dependentes do clock, para os flip-flops “b” e “out~reg0”. Para a mesma entrada, como se comportará as saídas “b” e “out”?



Verilog – A idéia do Fruto

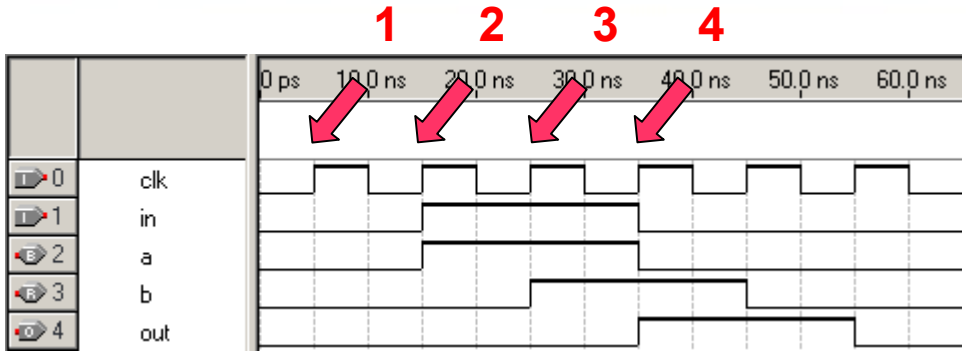
Já nesta “altura” não podemos dizer que os sinais de entrada dos flip-flops em questão são independentes do EVENTO, pelo contrário, eles surgem graças ao EVENTO, desta forma podemos chamá-los de FRUTOS.



Recorrendo a frase apresentada no ínicio deste conteúdo, podemos afirmar que as transições dos sinais, cujo são frutos de uma determinada transição do clock não podem interferir na mesma.

Desta forma, a respeito dos sinais “b” e “out” podemos concluir que:

Verilog – A idéia do Fruto

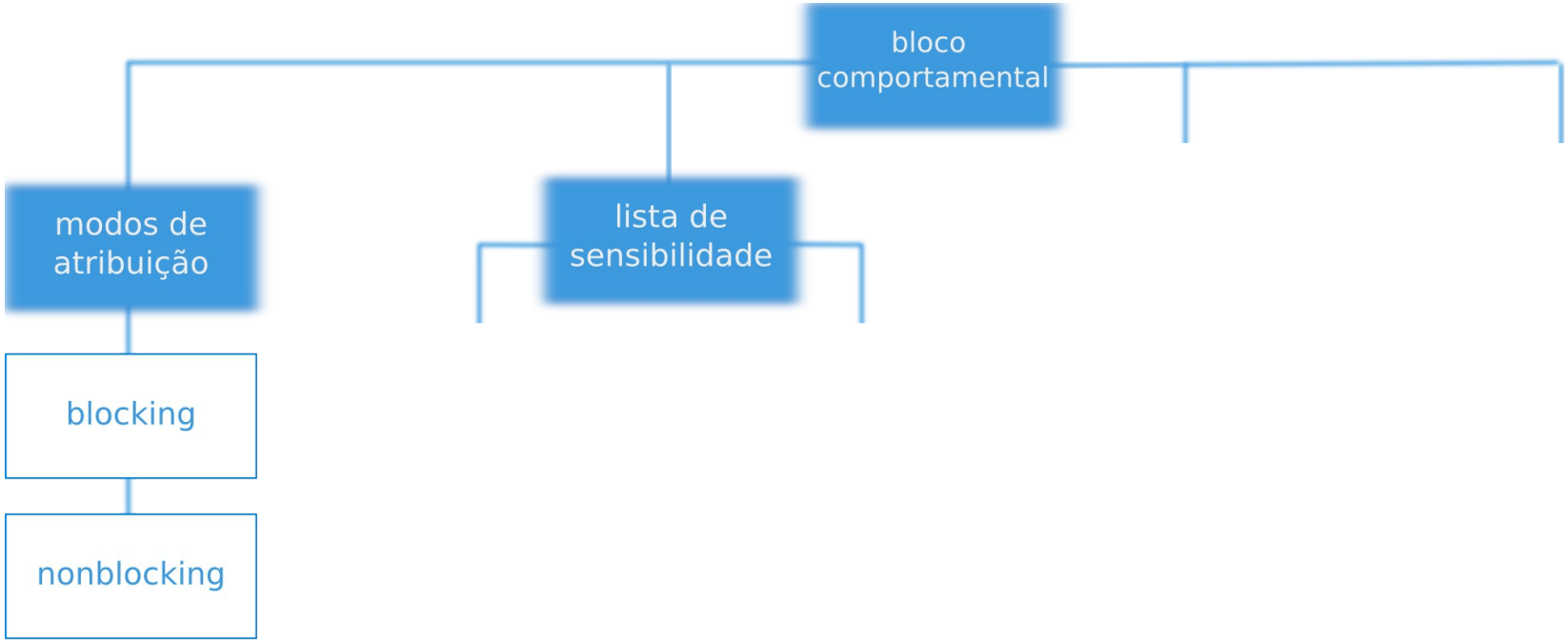


Neste momento devemos analisar as transições 3 e 4. Na transição 3 o sinal de “a” já está em alto e não é consequência da transição 3, ou melhor não é fruto da transição do clock, consequentemente ele pode interferir nesta mesma transição, diferente da transição do sinal “b” que é fruto sim da transição do clock.

Desta forma a mudança do sinal “a” é caregado em “b” neste instante.

De forma semelhanete acontece na transição 4, o que resulta no total num atraso de 3 ciclos de clock para a concatenação de 3 flip-flops em série, caso o primeiro sinal independer do clock.

Verilog – Lista de sensibilidade

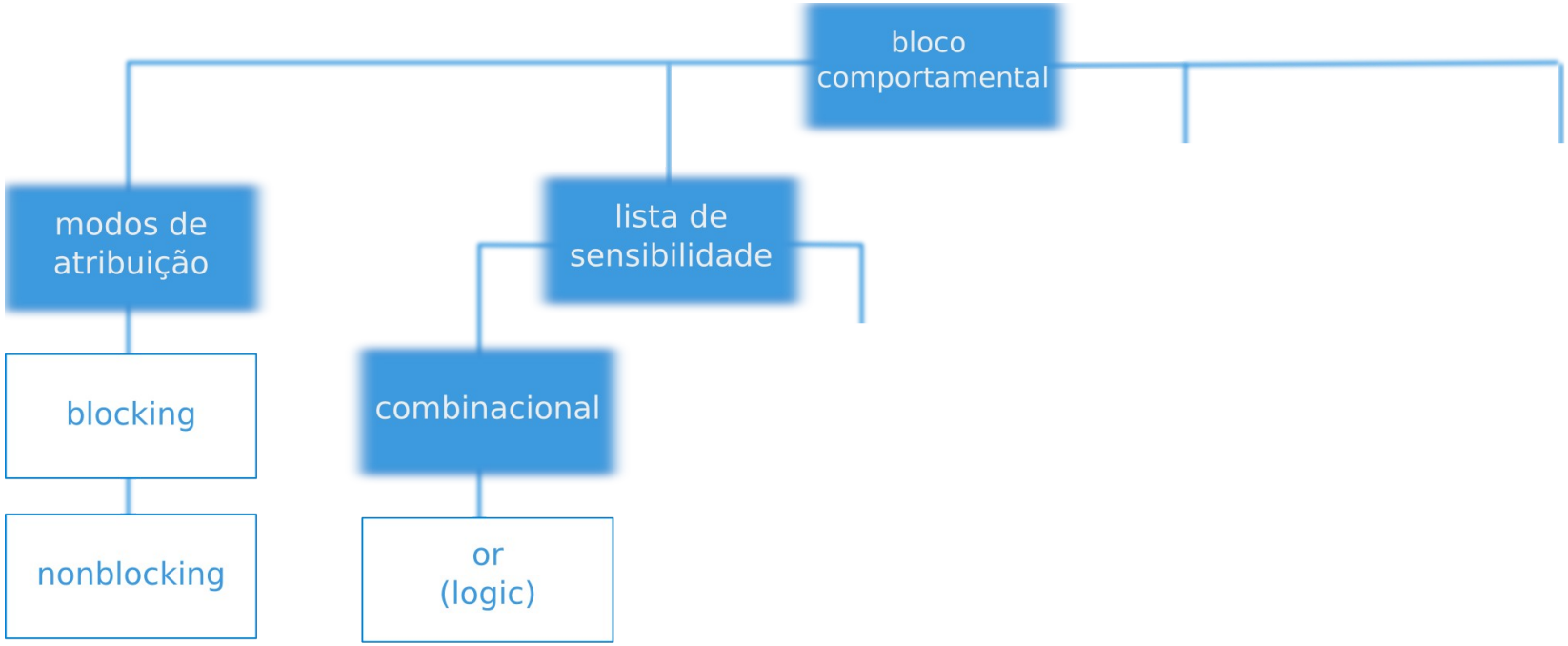


Verilog – Lista de sensibilidade

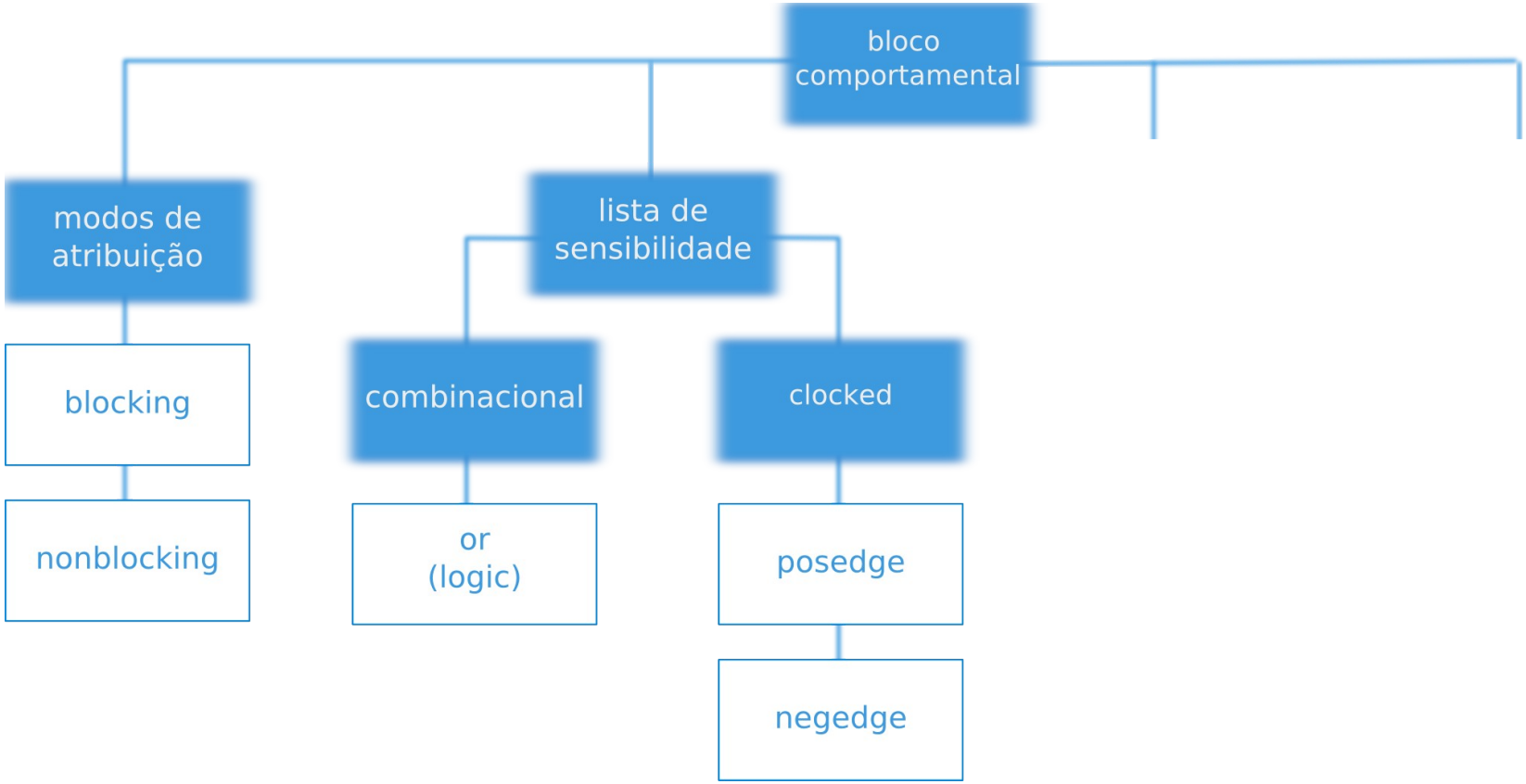
A lista de sensibilidade irá definir como o bloco de um determinado comportamento irá repetir, quando a condição estabelcida na lista for renovada, o bloco repetirá seu comportamento declarado.

```
always @(lista_de_sensibilidade)
    begin
        -- declaração #1
        -- ……..
        -- declaração #N
    end
```

Verilog – Lista de sensibilidade



Verilog – Lista de sensibilidade



Verilog – Lista de sensibilidade

Existem duas vertentes para a definição de lista de sensibilidade:

- Combinatorial process:
  A repetição do processo ocorre se qualquer sinal da lista for modificado
  Geralmente é definido a partir dos sinais de entrada
- Clocked process:
  A repetição é definida em cada borda do sinal da lista, sendo esta borda postiva ou negativa, no entanto, nunca as duas simultaneamente para um mesmo sinal.

Verilog – Lista de sensibilidade

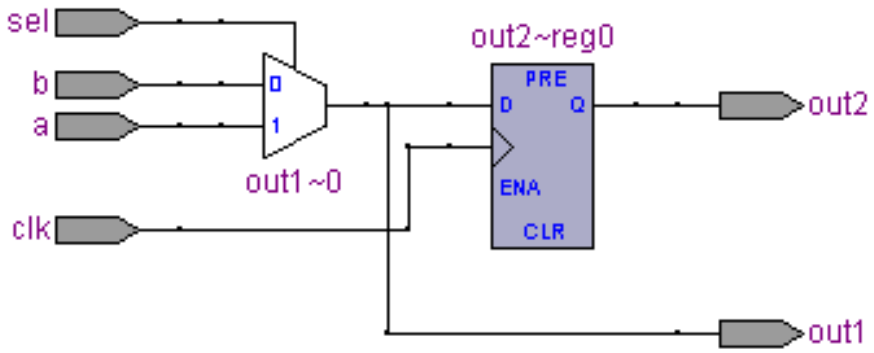
Exemplos:

- Combinatorial Process

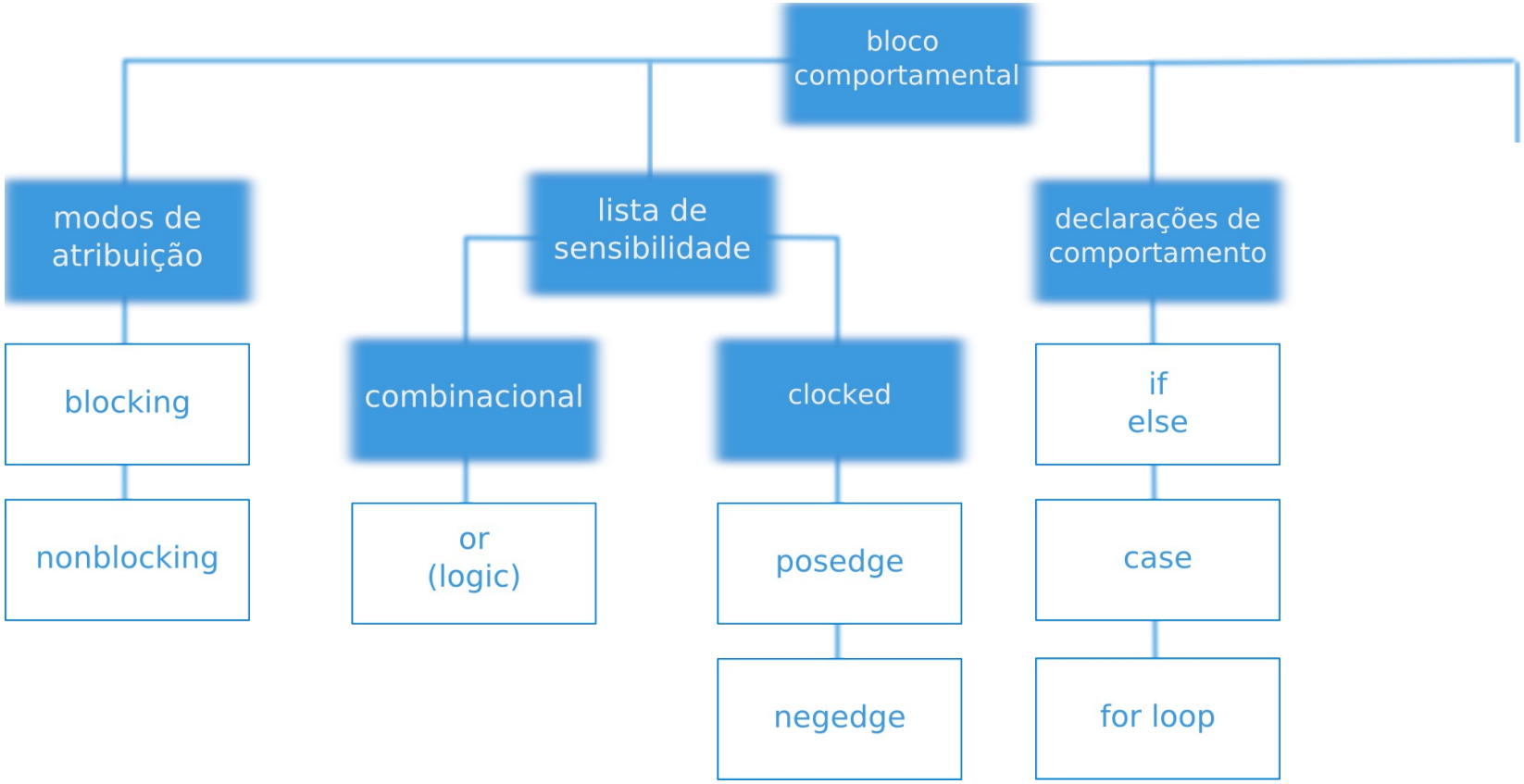
  always @(a or b or sel)

- Clocked Process

  always @(posedge clk or negedge clr_n)



Verilog – Declarações de comportamento



Verilog – Declarações de comportamento

Declarações de comportamento ou Behavioral Statements, são as maneiras de descrever o comportamento dentro de um procedural block.

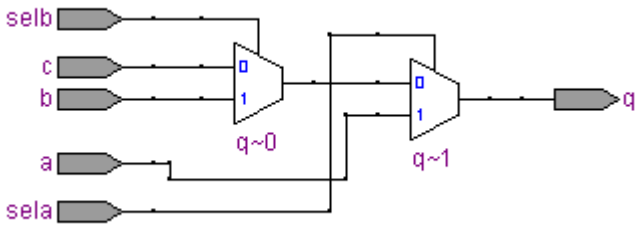
Estas declarações podem ser definidas a partir de três estruturas:

- **if – else**
- **case**
- **for loop** (não sintetizável)

Verilog – If – else statements

- Formato:

```
if (<condição1>)
        declaração_a
else
 if (<condição2>)
        declaração_b
else
        declaração_c
```



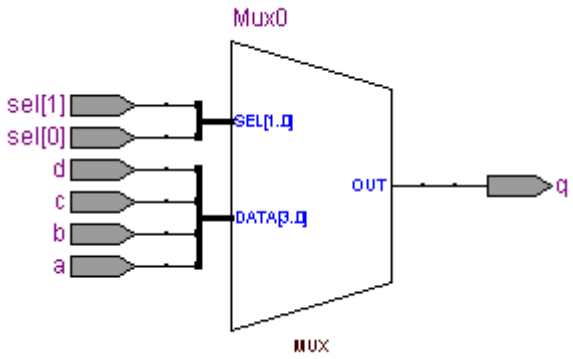
- Exemplo

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

Verilog – Case Statement

- Formato:

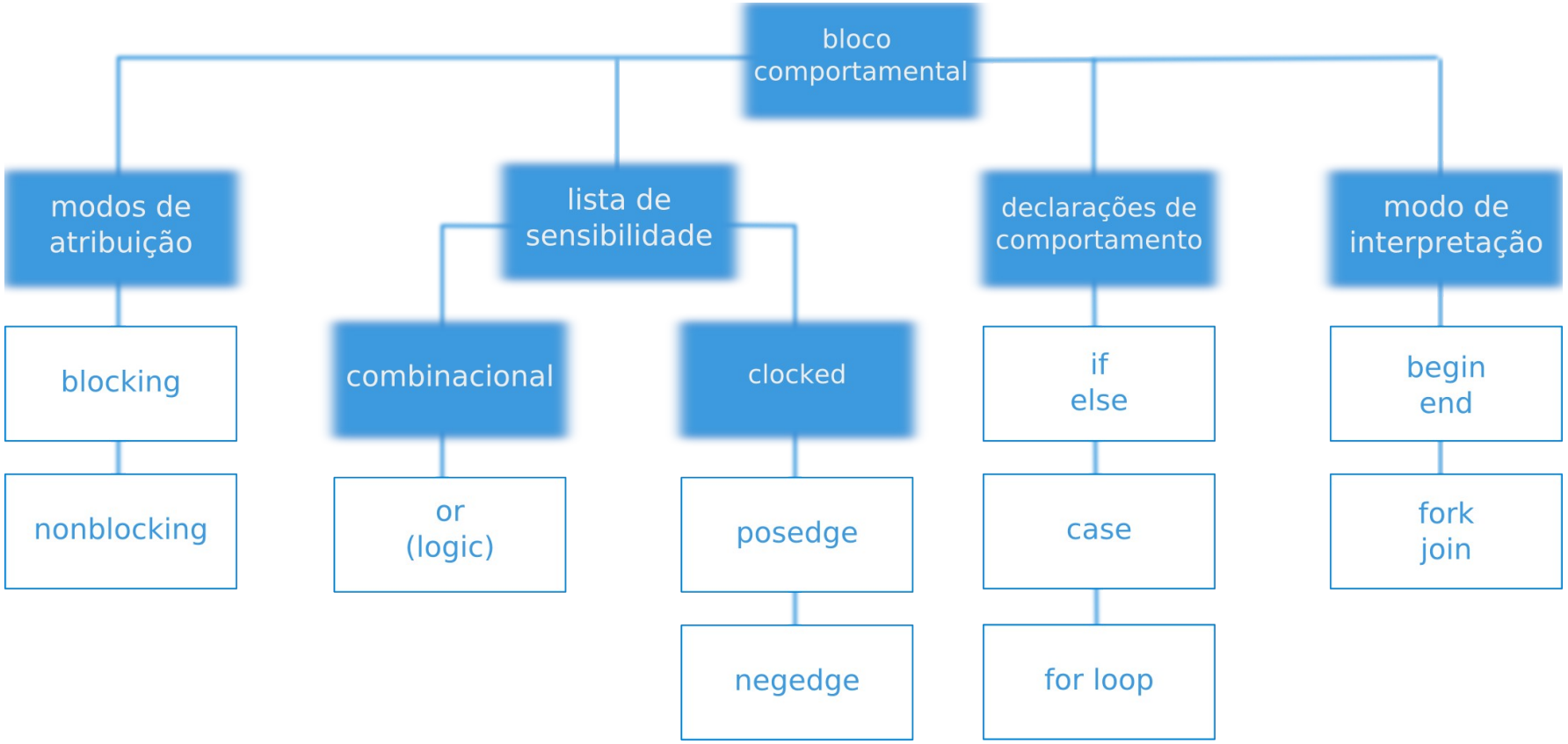
```
case (expressão)
   <condition1> :
         declaração_a
   <condition2> :
         declaração_b
    default :
         declaração_v
 endcase
```



- Exemplo

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

Verilog – Modos de interpretação



Verilog – Modos de interpretação

Os blocos initial e always podem ser interpretados de duas maneiras:

- Sequencial
- Paralelo

Quando o desenvolvedor deseja que as atribuições sejam interpretadas sequencialmente, deve-se usar begin e end para limitar o bloco.

```
always @ (posedge clk)
    begin
        a = b;
        c = d;
    end
```

Verilog – Modos de interpretação

Outrora o desenvolvedor precise que o compilador interprete as atribuições de forma paralela, ele pode limitar o bloco por fork e join

```
always @ (posedge clk)
    fork
        a = b;
        c = a;
    join
```

O mesmo se aplica para blocos do tipo initial.

```
initial
    begin
        a = b;
        c = d;
    end
```

```
initial
    fork
        a = b;
        c = a;
    join
```

Exercício de número 2

Projetar um contador módulo 64.

Verilog – Boas práticas (inicialização de variáveis)

É recomendável utilizar uma entrada de clear para inicializar/reinicializar os níveis lógicos dos registradores do projeto.

Na tecnolgia da Altera é padrão todos os registradores inicializarem com 0, no entanto mesmo neste caso é recomendável fazer uso desta recomendação para viabilizar a reinicialização dos registradores e garantir o resultado esperado em outros simuladores, como o Modelsim que não admite o estado desconhecido na inicialização dos registradores.

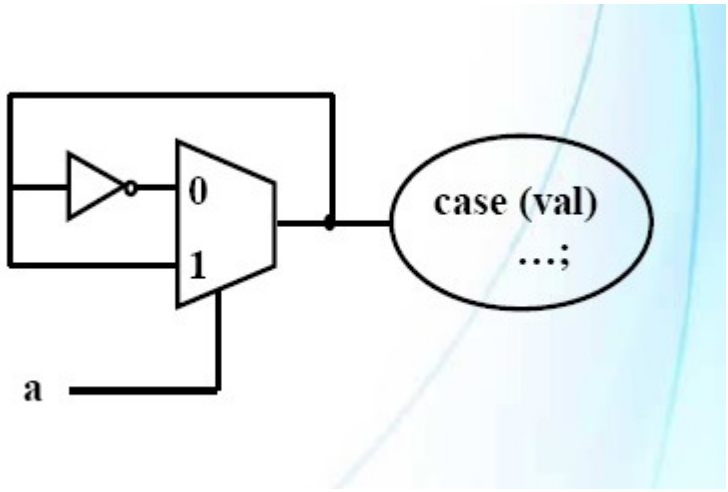
```
If (clr_n)
    begin
        // funcionalidade
    end
```

Verilog – Boas práticas (inicialização de variáveis)

Quando não se inicializa um valor de uma variável e determina a realimentação da mesma, se infere um latch.

Resumo: Não pode ler o valor de uma varíavel antes de definir o valor da mesma.

```
always @ (i or a)
        begin
        if (a == 1) val = val;
        else val = val + 1;
                case (val)
                1'b0 : q =
i[0];
                1'b1 : q =
i[1];
                endcase
        end
```



Verilog – Boas práticas (inicialização de variáveis)

Quando não se define uma ação numa determinada condição dentro de um processo com lista de sensibilidade combinacional, se infere um latch.

Resumo: Não pode deixar de definir todas as condições para todos os conjuntos if-else e case.

```
always @ (ina or inb)
begin
        if (ina == 1)
                begin
                if (inb == 1) out = 1;
                else out = 0;
                end
        else out = 0;
end
```

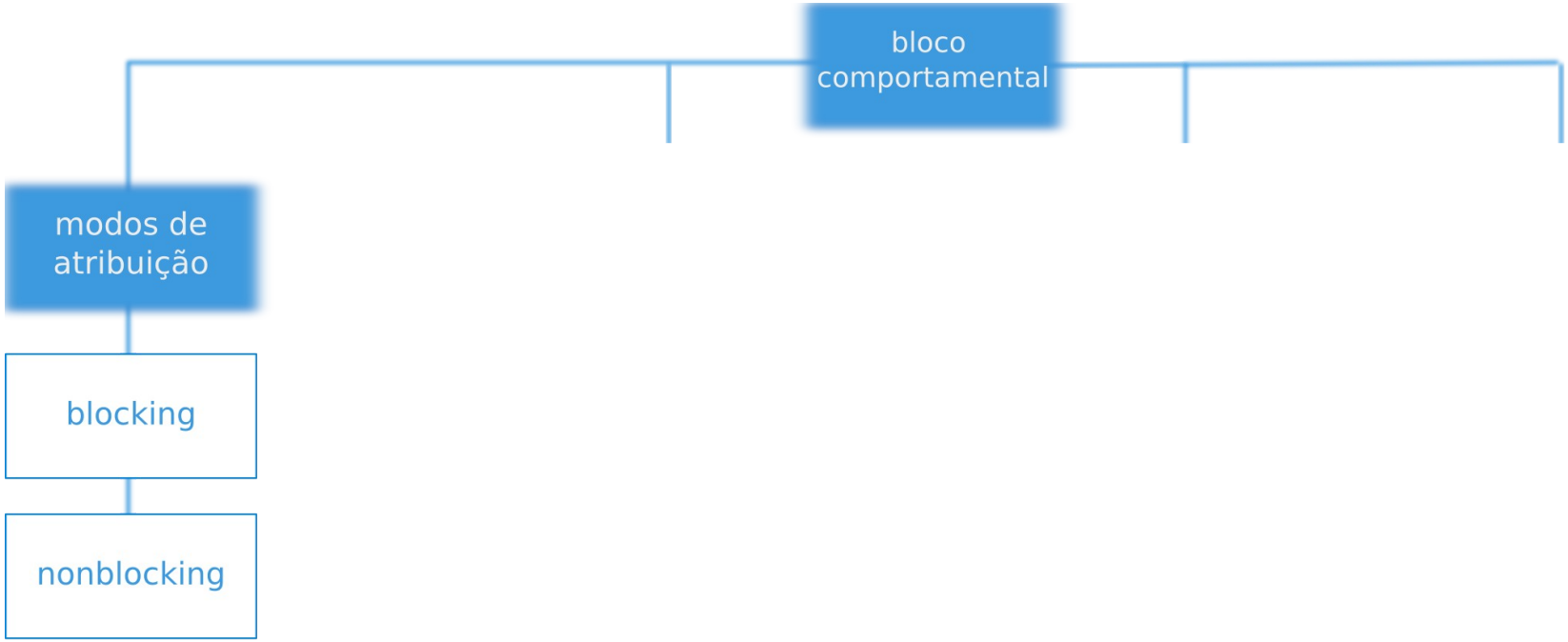
Exercício de número 3

Projetar um contador módulo 64, com entrada de clear e com quatro modos de operação, conta para cima, para baixo, carrega valor por entrada paralela e por último mantém o estado atual.

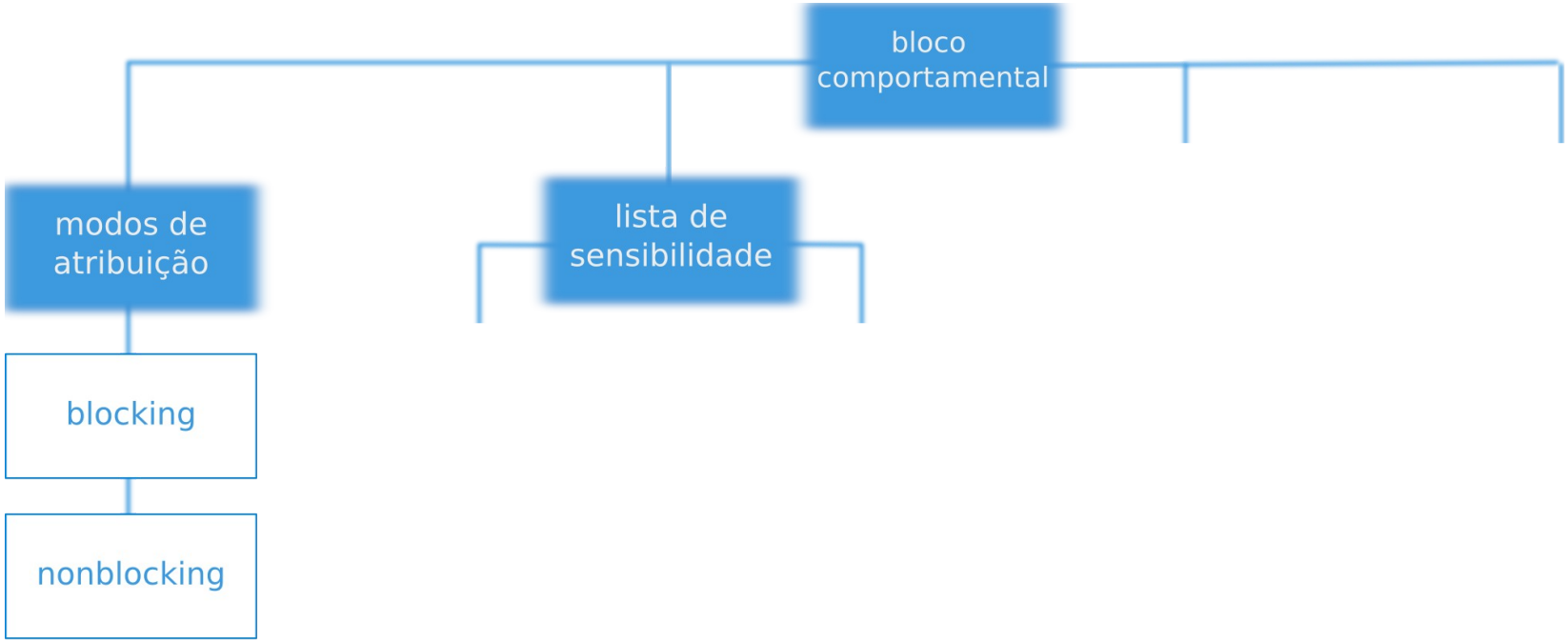
Verilog – Revisão parte 2



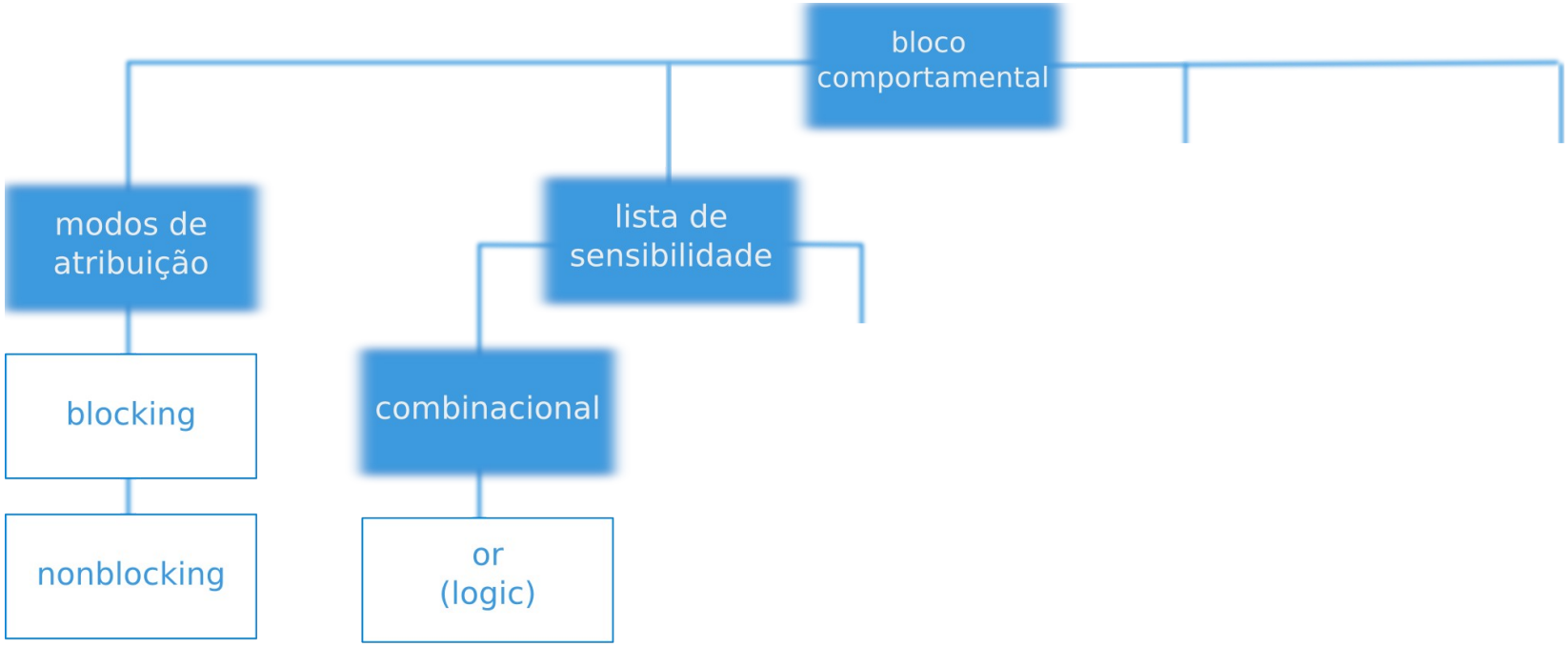
Verilog – Revisão parte 2



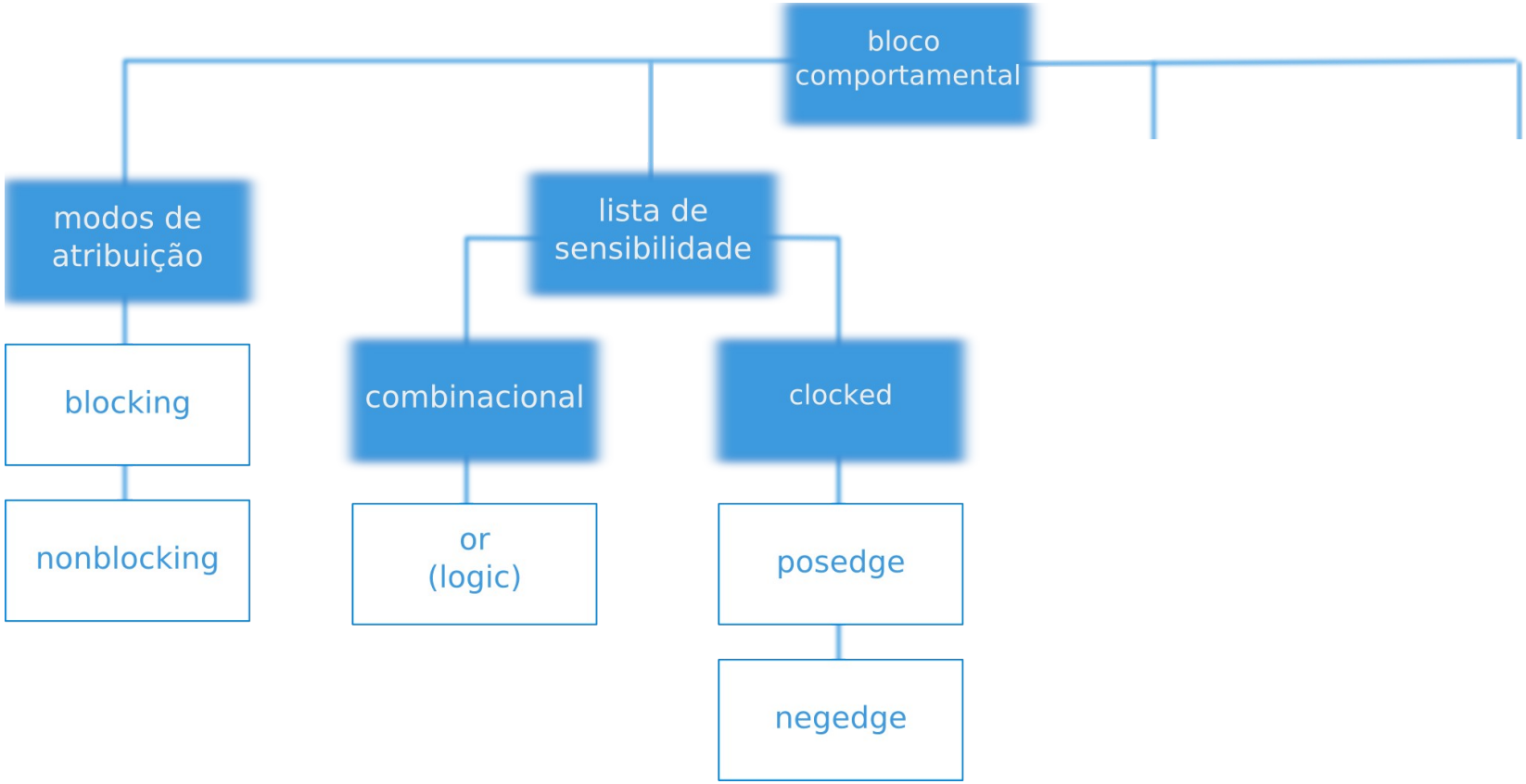
Verilog – Revisão parte 2



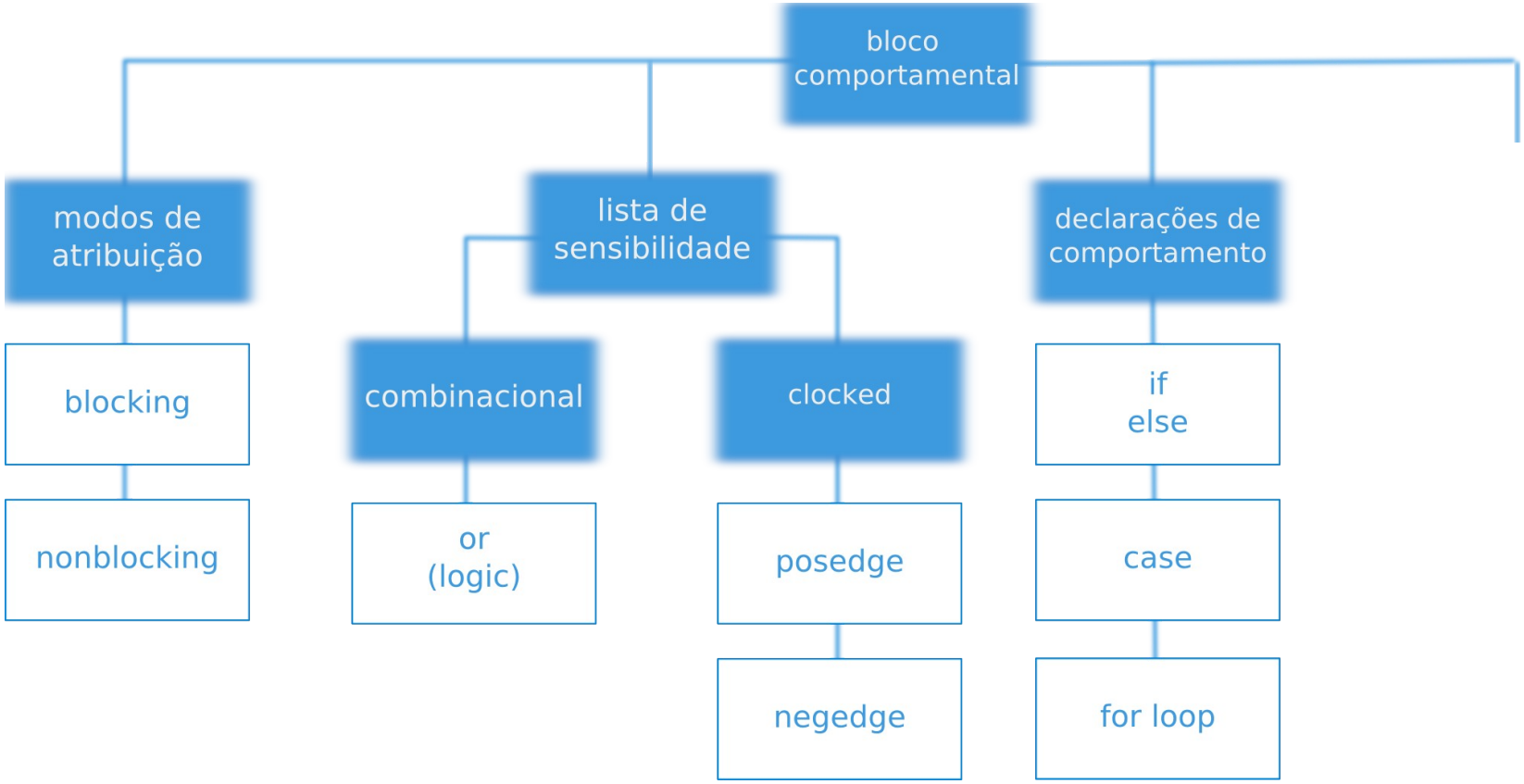
Verilog – Revisão parte 2



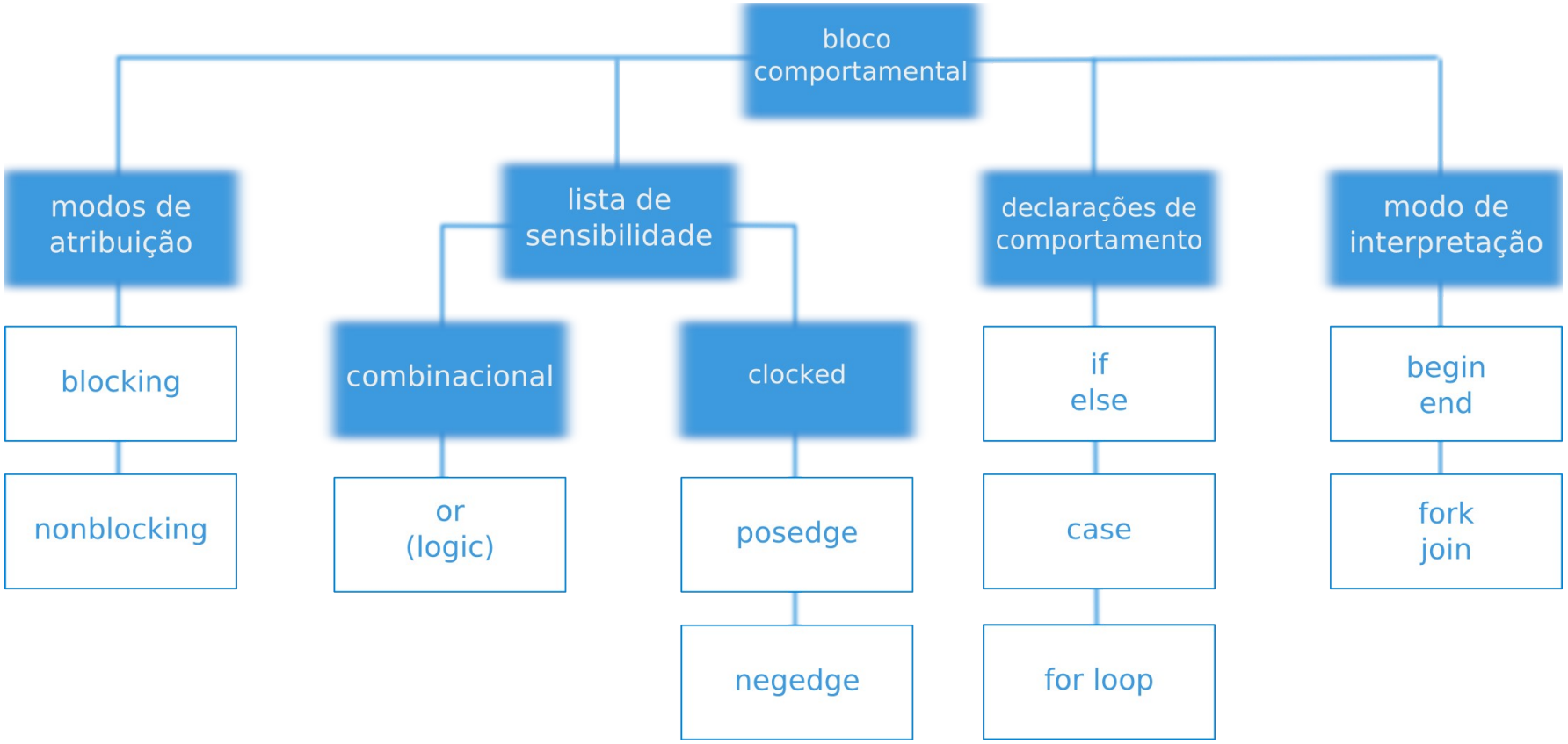
Verilog – Revisão parte 2



Verilog – Revisão parte 2



Verilog – Revisão parte 2



Testbenchs

Estudo da linguagem focado em códigos não sintetizáveis.

Verilog – Testbenchs

Tão importante quanto descrever a funcionalidade da nossa aplicação é validá-la corretamente.

O que seria validar nossa descrição de hardware ?

Validar é simular condições reais ao entorno de sua descrição de hardware a fim de garantir sua funcionalidade num ambiente real.

No término dessa avaliação, chamamos nosso projeto de validado e ele está pronto para seguir as etapas restantes do fluxo de projeto.

O que temos disponível para trabalhar com testbenchs ?

Verilog – Testbenchs

Uma boa notícia é que tudo que estudamos para descrever códigos sintetizáveis é permitido usar com testbenchs.

No entanto é possível descrever sistemas reais fazendo uso de sintaxes mais inteligíveis para o homem, como condições iniciais, repetições, atrasos de tempo e outras mais.

É importante expor uma lista dos principais recursos não sintetizáveis em Verilog:

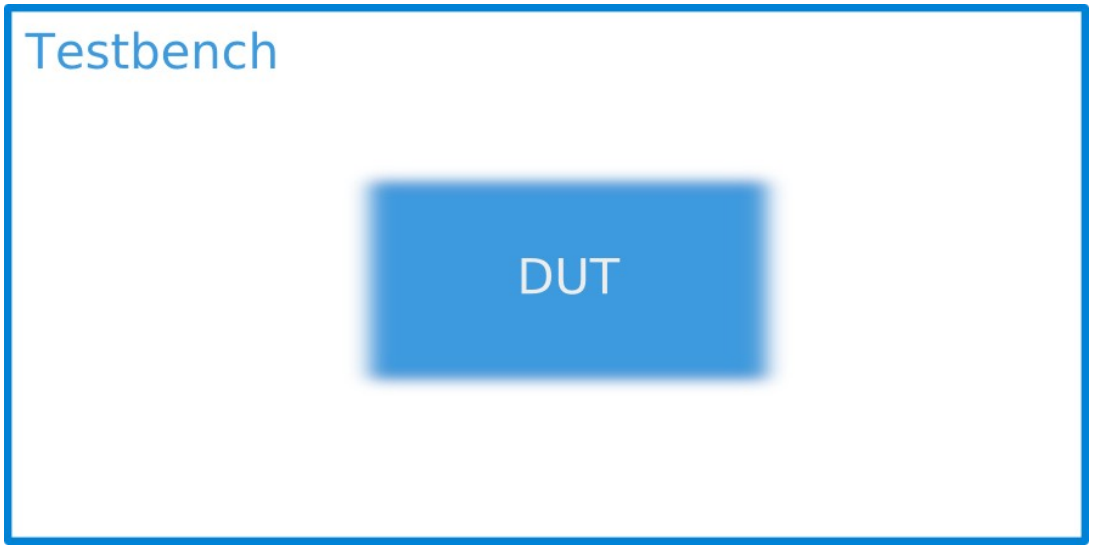
Verilog – Testbenchs (construções não sintetizáveis)

cmos,nmos,rcmos,rnmos,pmos,rpmos
assign and deassign
defparam*
event
force and release
fork,join
forever,while,repeat
initial
primitives (only gate level primitives supported)
pullup,pulldown
real data
time values
rtran,tran,tranif0
tranif1 rtranif0,rtranif1
user-defined primitives*

* Suportado pelo Quartus II

Verilog – Testbenchs

Chamamos de DUT (Device Under Test), a funcionalidade que desejamos validá-la, e de Testbench o código de apoio que irá aplicar estímulos e colher respostas para a avaliação do projeto, que pode ser feita até mesmo pelo desenvolvedor do código, ou ainda melhor, por uma outra equipe.



Verilog – Testbenchs (construções)

Podemos misturar várias construções no intuito de descrever a simulação, e a seguir veremos as principais:

```
always
#5 clk = ~clk;
```

always sem lista de sensibilidade roda sempre

#5 significa um retardo na atribuição de 5 ns.

```
initial
begin
        repeat (2000)
       begin
       #10 ena_in = 1'b0;
       #50 begin
       ena_in = 1'b1;
       in = $random;
       end
end
end
```

```
// Bloco initial, roda no tempo 0, sem repetir
// Modo de interpretação sequencial
// Construção não sintetizável que repete 'n' vezes
// Modo de interpretação sequencial
// atribuição retardada em 10 unidades de tempo
// atribuição da interpretação em 50 unidades após a atribuição anterior
// atribuição blocking
// atribuição blocking efetuada simultaneamente com a anterior
// fim do terceiro bloco de interpretação (#50)
// fim da declaração de comportamento repeat
// fim do bloco initial
```

Obs.:

Verilog – Testbenchs (construções)

Para alterações excepcionais de alguns sinais, é conveniente utilizar a construção initial com modo de interpretação paralelo, conforme a seguir:

```
initial
        fork
        #30 reset_n = ~reset_n;
        #61470 ifft_source_ready = 1'b1;
        #122940 fft_source_ready = 1'b1;
         join
```

Usamos um bloco do tipo initial com modo de interpretação paralelo.

Ele tem atribuições lidas paralelamente, no entanto, são retardadas em tempos diferentes, resultando em atribuições que acontecem no tempo que está escrito em cada linha da construção.

Verilog – Testbenchs

Podemos destacar outras tasks e functions:

$stop; - para a simulação

$finish; - sai da simulação

$monitor(…); - exibe um determinado valor

$time; - função que responde o tempo atual de simulação

Maiores informações:

http://www.ece.umd.edu/class/enee408c.F2001/Verilog-XL/system_tasks.html

Verilog – Modos de interpretação X de atribuição

Aprendemos anteriormente que no modo de atribuição nonblocking, todas elas eram carregadas simultaneamente.

Caso utilizarmos do modo de interpretação fork join (paralelo), todas as atribuições blocking passariam a se comportam como nonblocking, confome a seguir:

O mesmo se aplica para blocos do tipo initial.

```
initial
    begin
        a = b;
        c = d;
    end
```

```
initial
    fork
        a = b;
        c = a;
    join
```

Exercício de número 5

Descrever um testbench em verilog de um conversor P/S de 6 bits, usando a ferramenta Modelsim.

Verilog – Exercício de número 05

O exercício se propõe somente a descrição do Testbench, um código amador de um conversor P/S é descrito a seguir:

```
module paraleloserie (clk,clr_n,in,out,ena_in,ena_out);

input clk;
input clr_n;
input ena_in;
input[5:0] in;

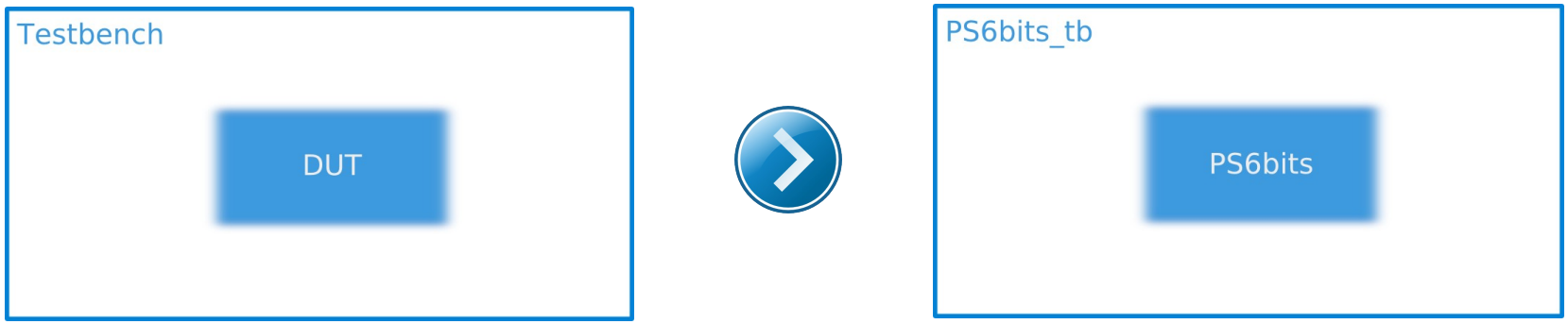
output out,ena_out;

reg out;
reg[2:0] cnt=3'b0;
reg start=1'b0;
reg ena_out=1'b0;
```

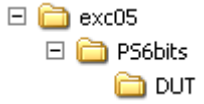
```
always@(posedge clk)
begin
if (~clr_n)
begin
        start <= 1'b0;
end
else
begin
        start <= (ena_in == 1'b1)? 1'b1 : start;
        if (start | ena_in) cnt <= (cnt == 3'd5)? 1'd0 : cnt + 3'd1;
        out <= in[cnt];
        ena_out <= ena_in;
end
end
endmodule
```

Verilog – Exercício de número 05

O objetivo do exercício é semelhante a um Reality Show, em que é proposto ao leitor que valide ou não o projeto, ou melhor, você deve dizer se o conversor P/S de 6 bits funciona apropriadamente.



É recomendável utilizar uma estrutura de diretório semelhante a esta que está no canto direito da projeção.



Dentro da pasta PS6bits colocamos os códigos não sintetizáveis (testbench) , e dentro da pasta DUT, colocamos os códigos sintetizáveis.

Agradecimentos



LAPTEL

Laboratório de Processamento Digital de Sinais e Telecomunicações



Contato

Marcos Oliveira

mvsoliveira@gmail.com
http:// www.laptel.ufjf.br

