SystemVerilog Design

Prof. Rômulo Calado Pantaleão Camara

Carga Horária: 2h/60h

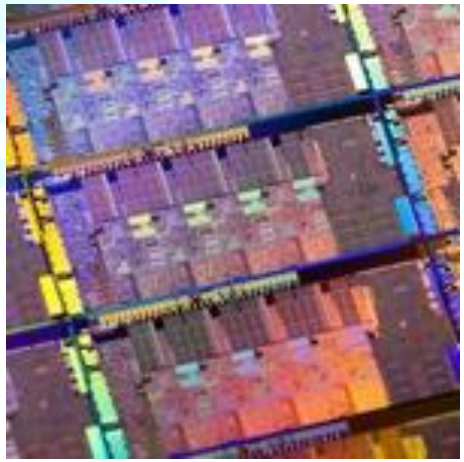
Universidade Federal do Vale do São Francisco - UNIVASF
Colegiado de Engenharia da Computação – CECOMP

Abstração do Mundo Real

✓A Maioria das pessoas quando utilizam um computador para jogar ou ver um filme, não sabe exatamente o que acontece dentro daquela máquina;

Barreira de abstração

✓Abstração é uma ferramenta no mundo real. Permite-nos usar muitas máquinas complexas, como automóveis e computadores com apenas algumas horas de treinamento.



Abstração

- ✓ SystemVerilog® também prover um nível de abstração para auxiliar na implementação de um projeto sem a necessidade de conhecimento prévio de todo o projeto;
- ✓ SystemVerilog® é compatível com qualquer tecnologia back-end;
- ✓ Permite flexibilidade em termos de escolha de tecnologia, metodologia de roteamento, e outras decisões de implementação back-end.

Objetivo

- ✓ Deixar as pessoas aptas a desenvolver uma verificação funcional em SystemVerilog utilizando a metodologia OVM_tpi.

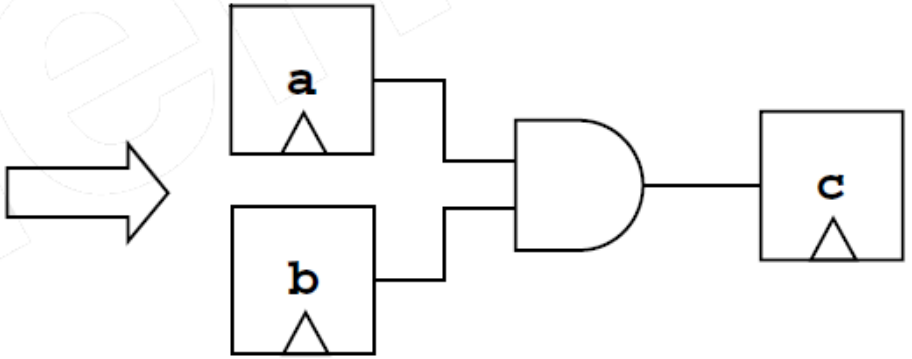
Agenda

- ✓ Definição de linguagem de descrição de hardware (HDL);
- ✓ Breve histórico sobre HDL;
- ✓ O que é SystemVerilog?
- ✓ Tipos de dados SystemVerilog e operadores utilizados para códigos sintetizáveis;
- ✓ Evento e Blocos Procedurais;
- ✓ Código sintetizável em SystemVerilog;
- ✓ Boas práticas de projeto;

HDL- Definição

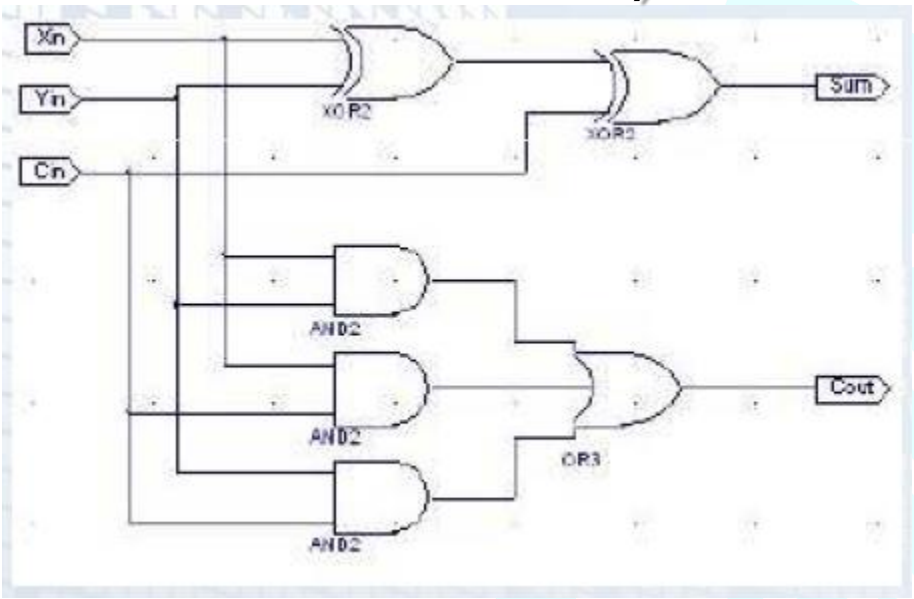
- ✓ Hardware description language (HDL) é uma linguagem de descrição utilizadas para descrever projetos de hardware;
- ✓ Diferentes das linguagens de softwares onde comando são executados um após o outro, HDL podem executar comandos em paralelo;

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



HDL- Definição

✓Antes das Linguagens de descrição de hardware (até meados dos anos 80):



- ✓ Desenho feito a mão, transistor por transistor, fio por fio.
- ✓ Imagine o custo de um único erro?
- ✓ Eram necessárias simulações SPICE.

HDL- Definição

✓Após as Linguagens de descrição de hardware (a primeira surgiu em 81):

```
// Simple 32-bit adder

module adder32 (a, b, sum);

input [31:0] a,b;
output [31:0] sum;

assign sum = a + b;

endmodule
```

- ✓ Modificada facilmente;
- ✓ A arquitetura não é definida no código, apenas na síntese;
- ✓ Pode ser simuladas com simuladores HDLs;
- ✓ Codifica-se geralmente a função do circuito;

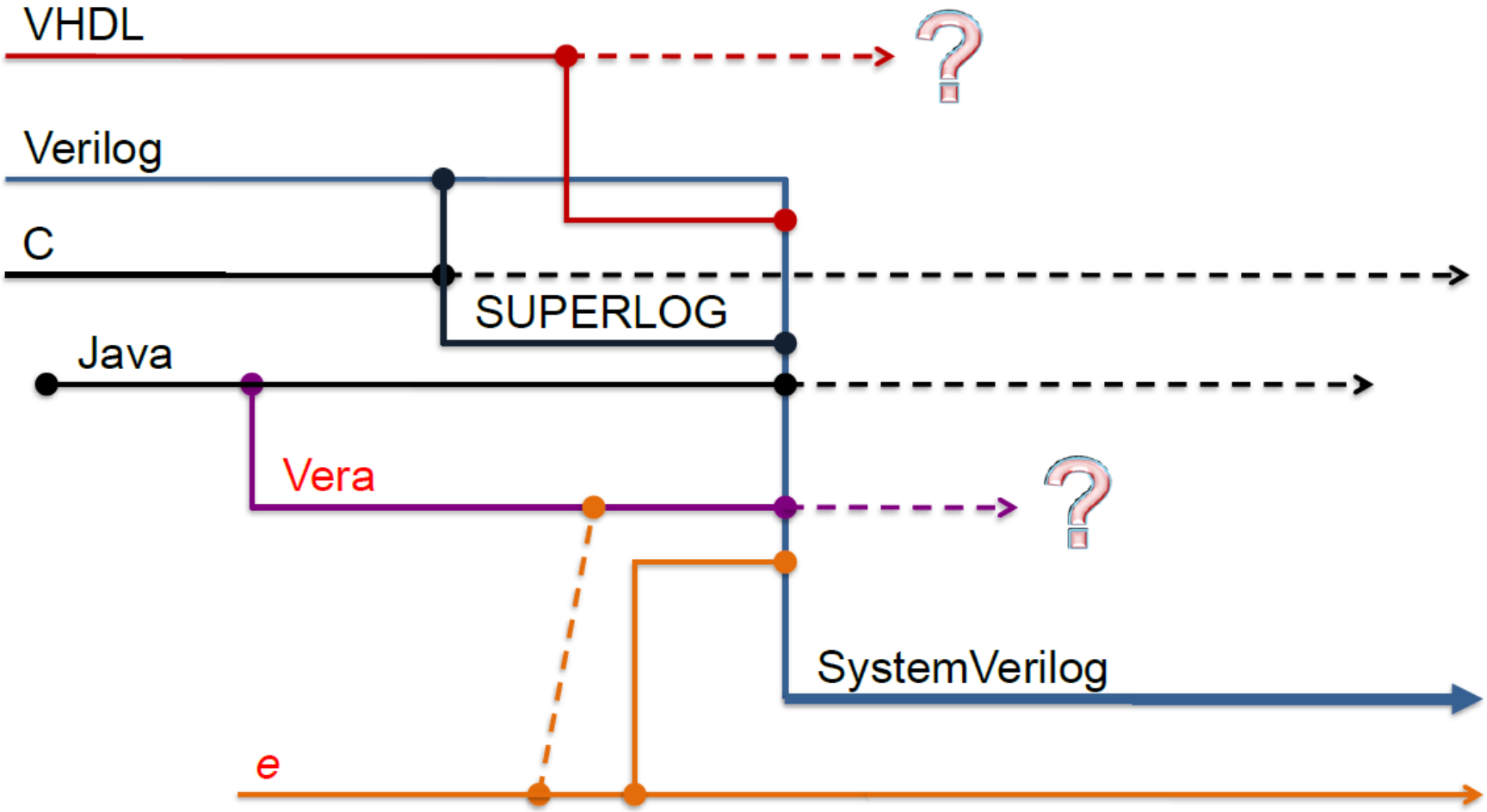
HDL- Definição

- ✓ Muito mais produtivo projetar o sistema pela sua funcionalidade.
- ✓ É possível projetar em qualquer nível de abstração (exceto transistores).
- ✓ Linguagens mais utilizadas:
  - Verilog HDL(1984)
    - Sintaxe mais parecida com C.
    - Mais utilizada no Vale do Silício.

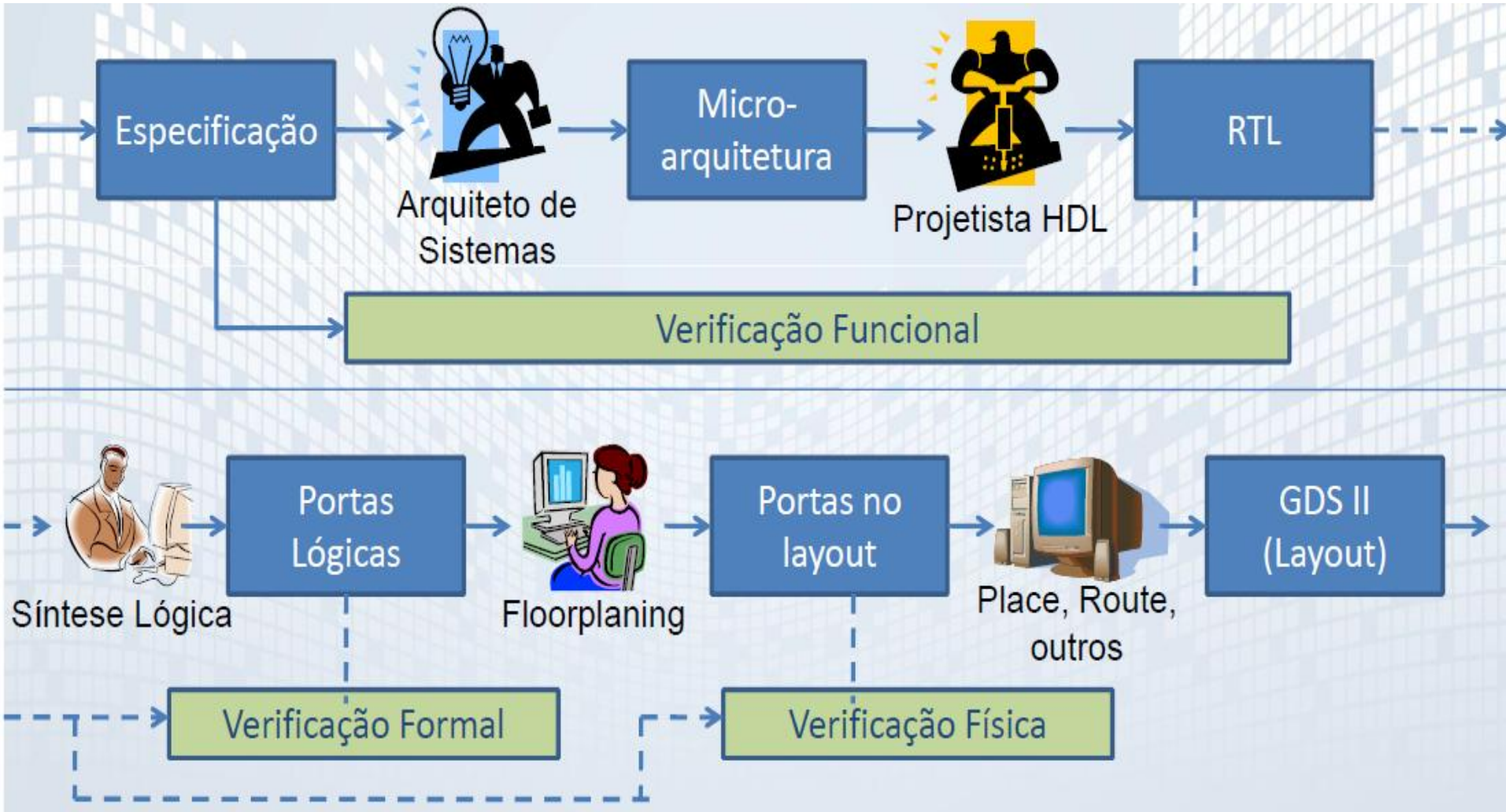
HDL- Definição

- ✓ Linguagens mais utilizadas:
  - VHDL – VHSIC Hardware Description Language (1981)
    - Criada pelo Departamento de Defesa Americano
    - Mais utilizado na Europa e nos órgão de governo americano.
  - System Verilog – Linguagem para Design Verificação que abrange o Verilog.

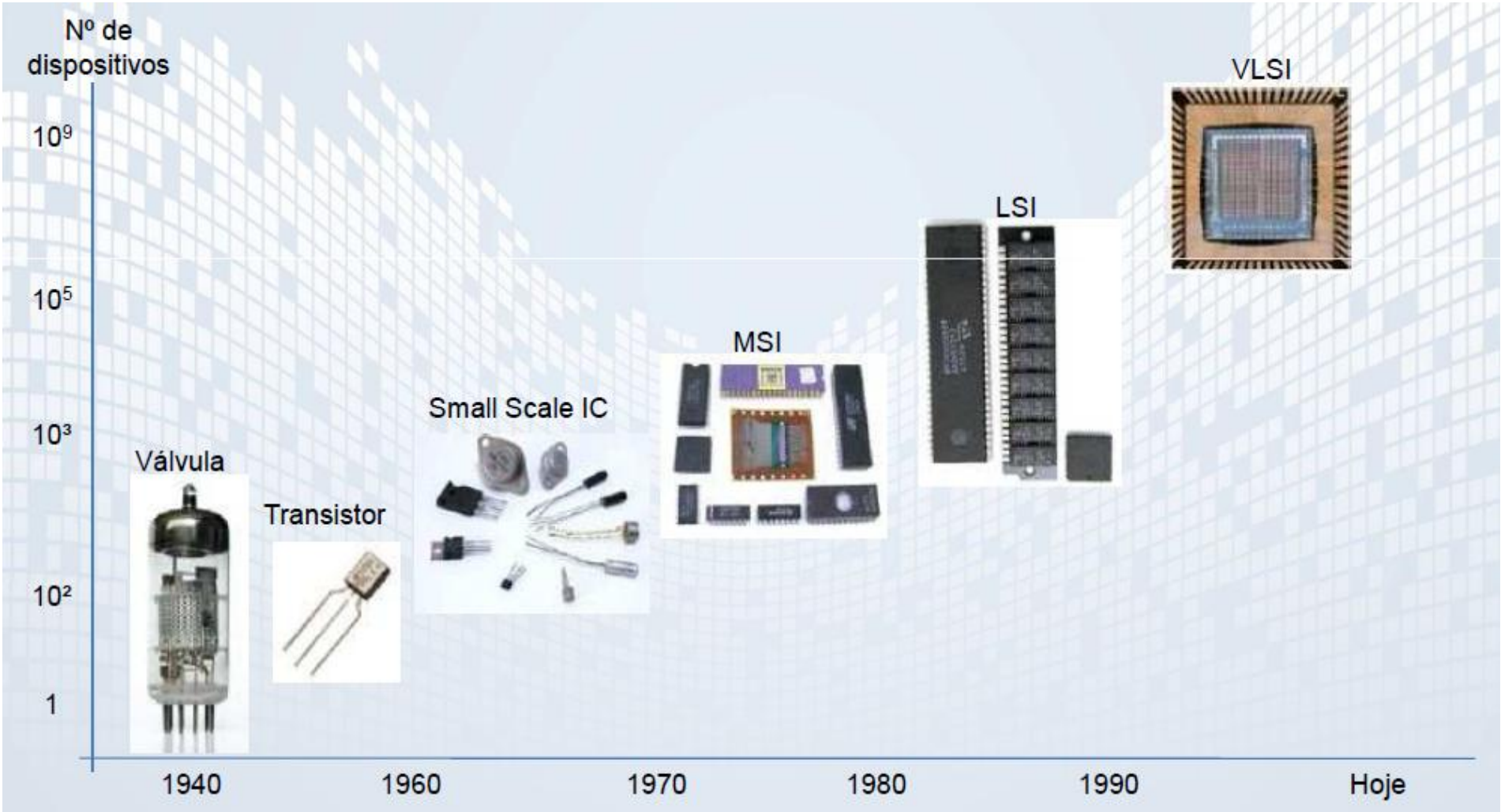
Histórico das Linguagens



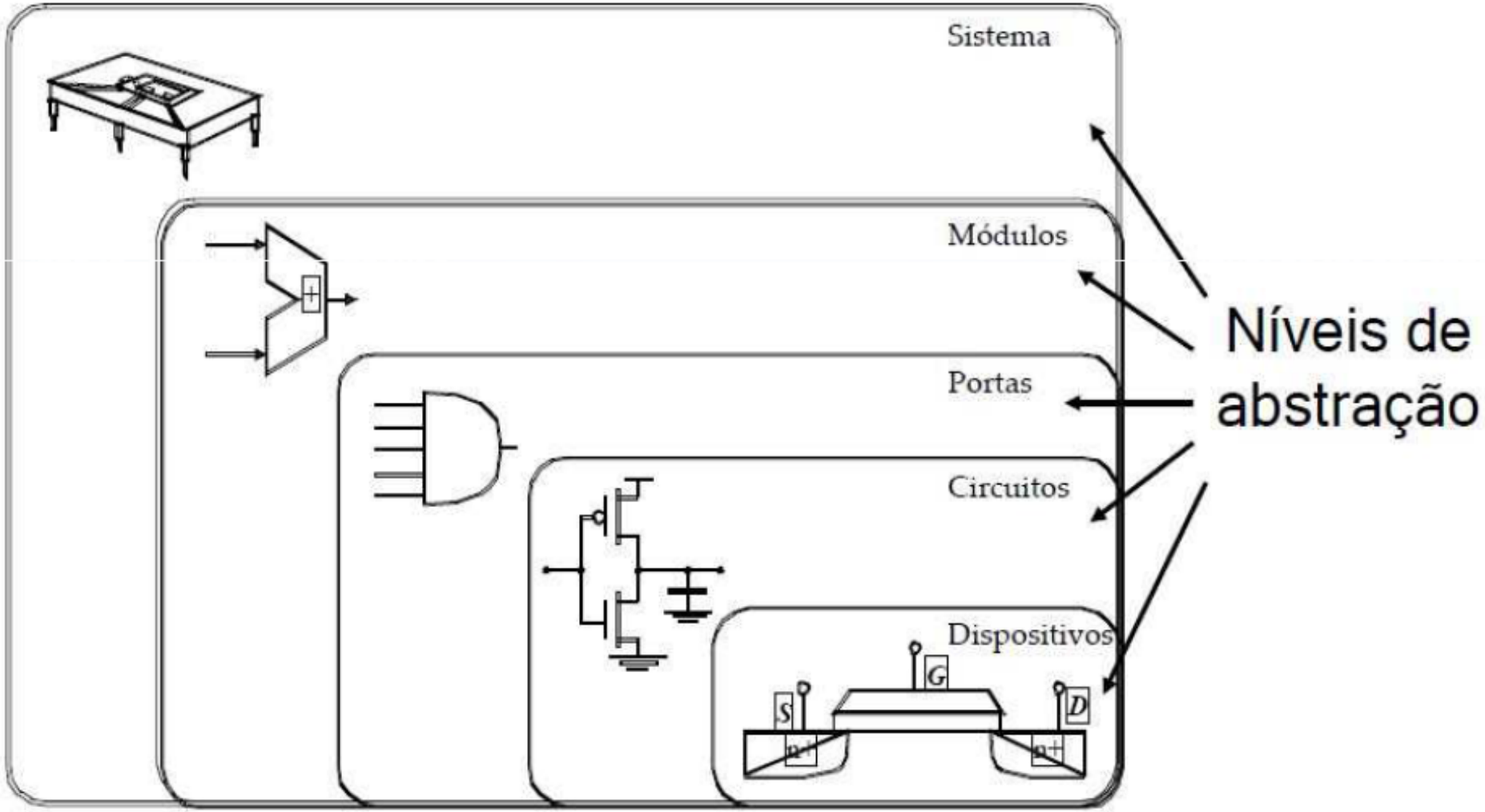
Fluxo de Projeto



Histórico dos CIs



Níveis Hierárquicos



O que é SystemVerilog?

- ✓ Linguagem de Descrição de Hardware dividida em:
  - Design.
  - Modelagem de Sistema.
  - Verificação:
    - Estática ou Formal.
    - Dinâmica ou Funcional.
    - Híbrida.

Vantagens do uso de SV para Design

- ✓ Sistemas com milhões de portas são complexos e levam muito tempo para ser projetados porta por porta;
- ✓ Sintaxe de SystemVerilog é fácil de aprender e entender;
- ✓ Prover reusabilidade de códigos por ser desenvolvido em forma de módulos;
- ✓ Prover fácil utilização de componentes já projetados dentro de outros módulos. Ex. Memórias, processadores.
- ✓ Permite desenvolvedores quebrar o código em blocos;

Design: Module - Definição

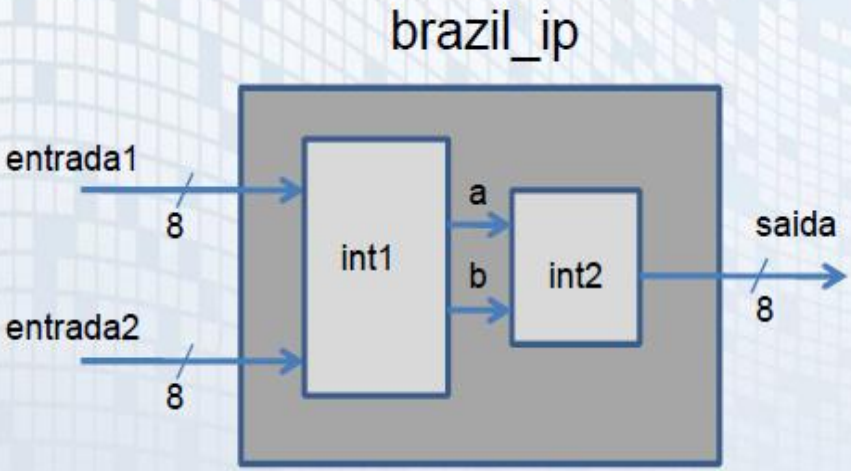
- Módulo: Estrutura Principal


```
module UNIVASF  (entrada1, entrada2,
saida);

input [7:0] entrada1, entrada2;
output [7:0] saida;

// comentario
/* Este código não faz nada, é uma
caixa vazia */

endmodule
```

Obs: Verilog é case sensitive

Design: Tipos de dados

- 1º Tipo de Dados: wire
- Hierarquia: instanciação de módulos.



Obs: a descrição dos módulos internos no próximo slide.

```
module brazil_ip (entrada1, entrada2, saida);
input [7:0] entrada1, entrada2;
output [7:0] saida;

wire [7:0] a, b;

/* Este codigo ainda não  faz nada, mas tem
duas caixas vazias dentro */
//instanciacao dos modulos internos

int1 X1 (.x(entrada1), .y(entrada2), .w(a), .z(b));
int2 X2 (.j(a), .k(b), .out(saida));

endmodule
```

Design: Tipos de dados

- ✓ Wire:
  1. Utilizado para conectar diferentes elementos;
  2. Podem ser tratado como um fio físico;
  3. Podem ser lidos ou atribuído;
  4. Nunca é guardado o valor;
  5. Necessitam ser atribuídos sempre por uma porta do módulo.
- ✓ Quando uma entrada ou saída é declarada no módulo, por ***default*** ela é uma wire;
- ✓ SystemVerilog ainda conta com mais dois tipos de dados: Reg e Logic;

Design: Tipos de dados

- ✓ Reg:
  1. Ao contrário do nome, regs não necessariamente corresponde a um registrador físico.
  2. Representa o elemento de armazenamento;
  3. Retém os valores até um próximo valor ser recebido;
  4. São sintetizados como flip-lops, latch ou circuitos sequenciais;

Design: Tipos de dados

- ✓ Logic:
  1. SystemVerilog inseriu esse tipo de dado para acabar com o problema do que deveria ser WIRE ou REG.
  2. A ideia é que o sintetizador se responsabilise pela escolha entre **wire** e **reg**.
  3. Não permite multiplos valores. No caso de wire e reg o valor de saída seria X.
  4. Pode ser wire em um local e reg no outro; like

```
logic a;
assign a = b ^ c; // wire style
always (c or d) a = c + d; // reg style
MyModule module(.out(a), .in(xyz)); // wire style
```

Exemplo Hierarquia

```
module multiplier (a, b, y);
input [3:0] a, b;
output [7:0] y;
wire [4:0] int;

//instantiating an adder module
adder u1 (.x1(a), .x2(b), .z(int));

//more contents of the module


endmodule
```

```
module adder (x1, x2, z);
input [3:0] x1, x2;
output [4:0] z;

//contents of the adder module

endmodule
```

Task - Definição

- ✓ Geralmente conhecidas como sub-rotinas ou ***procedures;***
- ✓ São definidas nos módulos onde serão utilizadas;
- ✓ Pode ser implementada em um módulo a parte, desde que seja instanciada no módulo que irá utilizar a diretiva de compilação ***include;***
- ✓ ***Tasks*** são temporizadas (pode incluir atrasos);
- ✓ Podem ter qualquer quantidade de entradas e saídas;
- ✓ Variáveis declaradas internas são locais;
- ✓ Não pode ser utilizado dentro de uma expressão;

Função - Definição

- ✓ Semelhante a ***Task***, com duas diferenças:
  - Só pode ter uma saída;
  - Não pode conter atrasos;
- ✓ Utilizado apenas para lógica combinacional;
- ✓ As entradas e saídas são definidas na descrição da função;

```
function integer clogb2;
    input [31:0] depth;
for (clogb2=0; depth>0; clogb2=clogb2+1)
    depth = depth >> 1; // divide by 2 until depth reaches 0
endfunction
```

Case, priority and unique case and if

- ✓ Case:
  - Checagem dos casos em ordem;
  - Uma condição pode sobrepor outra: ex:

```
case (prior)
0: op = a;
0,1,2 : op = b;
3: op = c;
endcase
```

  - Sintetiza de acordo com a prioridade do compilador;
  - default;

```
case (fullc)
        0,1: op = a;
        2: op = b;
        default: op = c;
endcase
```

Case, priority and unique case and if

- ✓ Priority and unique case:
  - Utilizado semelhante ao full_case, porém avisa ao designer fora do escopo de valores possíveis;
  - É permitido a sobreposição de valores;

```
Priority case (fullc)
        0,1: op = a;
        1,2: op = b;
        3 : op = c;
endcase
```

  - Unique case escreverá um *warning* sempre que executar o código acima.

Só pode haver um único caso para cada possibilidade existente.

Case, priority and unique case and if

- ✓ if-else:
  - Condicional semelhante a qualquer linguagem de programação, porém muito cuidado com latches!!
- ✓ Priority if:
  - Condições são analisadas em sequência e pelo menos uma deve ser verdadeira; pode existir mais de uma verdadeira; senão dará um *warning*;
- ✓ Unique if:
  - Utilizada para lógica paralela. As condições são checadas paralelamente. Apenas uma condição deve ser verdade.

```
priority if (en_a)
   op = a;
else if (en_b)
   op = b;
else if (en_c)
   op = c;
```

```
unique if (ctrl >= 3'b100)
   op = a;
else if (ctrl <= 3'b010)
   op = b;
else
   op = c;
```

Operadores: precedência

Symbol	Meaning	Precedence
~	NOT	Highest
*, /, %	MUL, DIV, MODULO	
+, -	PLUS, MINUS	
<<, >>,	Logical Left/Right Shift	
<<<, >>>	Arithmetic Left/Right Shift	
<, <=, >, >=	Relative Comparison	
==, !=	Equality Comparison	
&, ~&	AND, NAND	
^, ~^	XOR, XNOR	
|, ~|	OR, NOR	
?:	Conditional	Lowest

Tipos de dados e operadores

- ✓ Como qualquer outra linguagem, verilog tem um conjunto próprio de operadores:

Type	Symbol	Example	Comment
Arithmetic	`+ - * / %`	`c = a + b;`	Be careful when using * / or % as they synthesize to complicated logic.
Bit-wise	`~ & | ^ ~^`	`c = a | b;`	Performs a logical operation on the bits of the two operands.
Logical	`! && ||`	`if(flag1 && flag2)` `  c = a + b;`	Result produces 1'b0 or 1'b1
Reduction	`& | ^ ~& ~| ~^`	`c = &a;`	Performs a logical operation on all the bits of a single operand.

Tipos de dados e operadores

Type	Symbol	Example	Comment
Shift	`<< >>`	`c = a << 2;`	Shifts the bits of an operand. Use shift instead of multiply or divide, if possible.
Relational	`< > <= >= == !=`	`c = a <= b;`	Result produces 1'b0 or 1'b1. **Do not** confuse <= with non-blocking assignment (will be covered later).

Conditional	`?:`	`assign mux_out = sel ? a : b;`
		`/*Same as :`
		`if(sel)`
		`assign mux_out = a;`
		`else`
		`assign mux_out = b;*/`
			Shortcut for a if/else statement.

Concatenation	`{}`	`c = {1'b1, 2'b00, a};`	Concatenates bits together to make a bus.
Replication	`{{}}`	`c = {3{2'b10}};` `//c is now 6'b101010`	Replicates a group of bits.

Tipos de dados e operadores

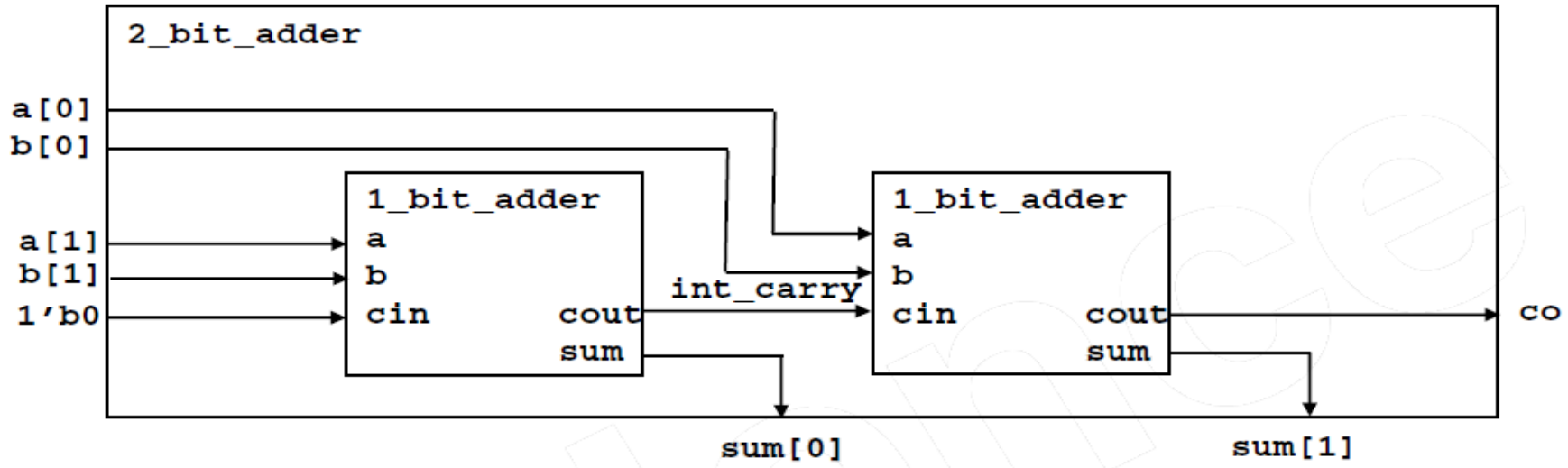
- ✓ Representação numérica

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	Binary	5	101
'b11	unsized	Binary	3	000000..00011
8'b11	8	Binary	3	00000011
8'b1010_1011	8	Binary	171	10101011
3'd6	3	Decimal	6	110
6'o42	6	Octal	34	100010
8'hAB	8	Hexadecimal	171	10101011
42	unsized	Decimal	42	0000...00101010

Tipos de dados

- ✓ SystemVerilog também suporta os seguintes tipos de dados:
  - Int;
  - Char;
  - String;
  - Outros;
- ✓ Todos para implementação a nível de sistema;
- ✓ Existem outro tipo de dado bastante utilizado: parâmetros:
  - Parâmetros que são utilizados pelos simuladores e sintetizadores. **Localparam ou parameter;**

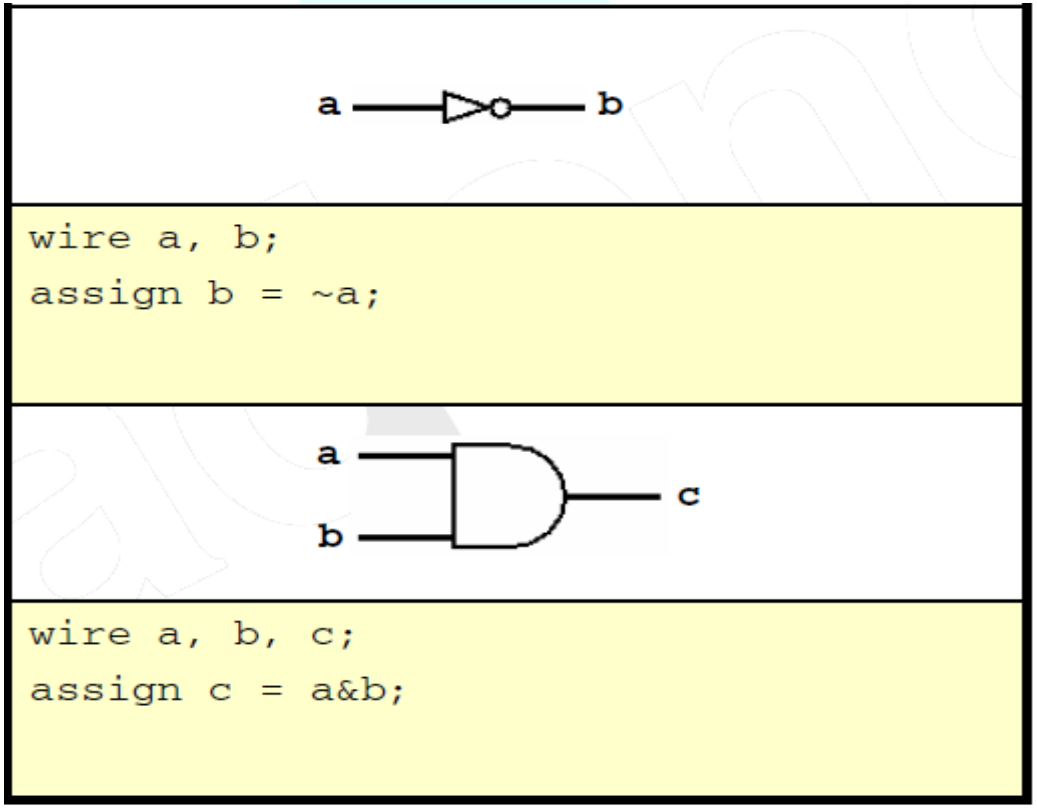
Exemplo




```
module 2_bit_adder (a, b, sum, co);
input [1:0] a, b;
output [1:0] sum;
output co;

wire int_carry;

//instantiating an adder module
1_bit_adder u1 (.a(a[0]), .b(b[0]), .cin(1'b0), .co(int_carry), .sum(sum[0]));
1_bit_adder u2 (.a(a[1]), .b(b[1]), .cin(int_carry), .co(co), .sum(sum[1]));

//more contents of the module
endmodule
```

Wires e Assign

- ✓ Wire e Assing são utilizados juntos para desenvolver uma lógica combinacional direta (on the fly);



Parâmentros

- ✓ São especificações de módulos que também existe em outras linguagem;
- ✓ Podem ser alteradas cada vez que o módulo for instanciado;
- ✓ São utilizados geralmente para definir tamanho, largura, nomes de estados e atrasos;

```
module adder (a, b, y);
parameter WIDTH = 4;
input [WIDTH-1:0] a, b;
output [WIDTH:0] y;

//more contents of the module
...
endmodule
```

```
module multiplier (a, b, y);
input [4:0] a, b;
output [9:0] y;
wire [5:0] int;

//instantiating an adder module
adder u1 (.x1(a), .x2(b), .z(int));
defparam u1.WIDTH = 5;
//more contents of the module
endmodule
```

Modelagem Comportamental

- ✓ Duas formas de descrever:
  - Continuous Assignment (*assign).*
  - Blocos procedurais.

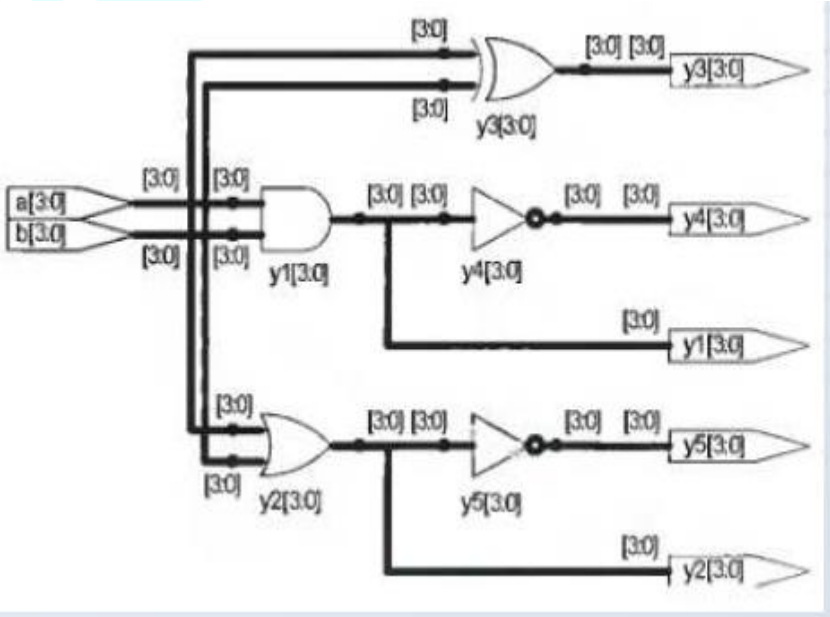
- ✓ *assign* descreve apenas lógica combinacional.

- ✓ Blocos procedurais descreve tanto lógica combinacional quanto sequencial.

Assign

- ✓ Qualquer mudança nos sinais do lado direito causa atualização imediata no lado esquerdo da igualdade.
- ✓ Descreve apenas circuitos combinacionais.

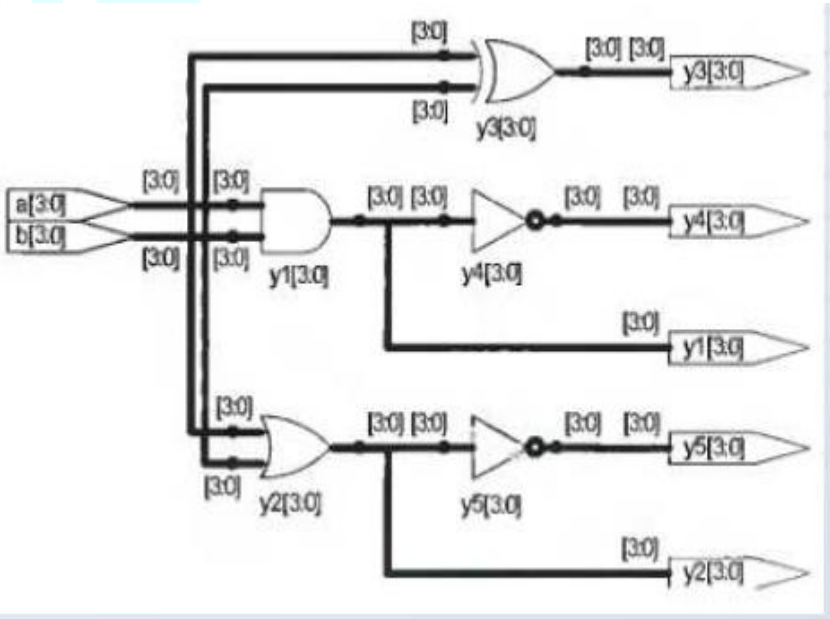
```
module gates (input [3:0] a, b,
output [3:0] y1, y2, y3, y4, y5);
/* 5 diferentes portas de 2 entradas
 e 4 bits */
    assign y1 = a & b; //AND
    assign y2 = a | b; //OR
    assign y3 = a ^ b; //XOR
    assign y4 = -( a & b ); //NAND
    assign y5 = - ( a | b ); //NOR
endmodule
```



Assign

- ✓ Operadores condicionais (select ? TRUE : FALSE).

```
module mux2 (input [3:0] d0, d1,
input s,
output [3:0] y);
assign y = s ? d1 : d0 ; //mux 2 inputs
endmodule
```



Blocos procedurais

- ✓ São seções contendo atribuições que são executadas linha por linha, semelhante a um software convencional;
- ✓ Multiplos blocos procedurais podem interagir concorrentemente;
- ✓ Existem dois tipos de blocos procedurais:
  - ***always***
    - Sempre executa um evento ou lista de eventos que exite no bloco;
    - Pode ser construído com lógica combinacional ou sequêncial;

Blocos procedurais

- ***Initial***
  - Utilizado apenas em simulação;
  - Não é sintetizável;
  - Utilizado para troca de sinais quando desenvolve um testbench;

Blocos Procedurais SV

- ✓ SystemVerilog inseriu outros blocos procedurais com a finalidade de auxiliar o projetista na escrita de descrições mais legíveis e organizadas:
  - **Always_ff**: Desenvolvimento de blocos sequenciais;
  - **Always_comb**: Descrição de blocos combinacionais;
  - **Always_latch**: Descrição de latches. Use apenas se realmente for desejado;

```
always_ff @(posedge clk or posedge rst)
  if (rst)
    op <= 1'b1;
  else
    op <= ip;
```

```
always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

```
always_latch
  if (gate == 1)
    op <= a;
```

Lista de Eventos

- ✓ Todo bloco ***always*** vem com uma lista de eventos;
- ✓ Só é executado o bloco quando todas as condições do evento são satisfeitas;
- ✓ O evento pode ser detalhadamente especificado(posedge, negedge);

Event list using the or operator

```
always @ (a or b or sel) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Event list using the * wildcard

```
always @ (*) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Lista de Eventos

- ✓ O evento pode ser detalhadamente especificado(posedge, negedge);
- ✓ O símbolo (*) adiciona todos os sinais do bloco na lista de eventos;
- ✓ **Iff** pode ser utilizado dentro de um evento;

Event list using the or operator

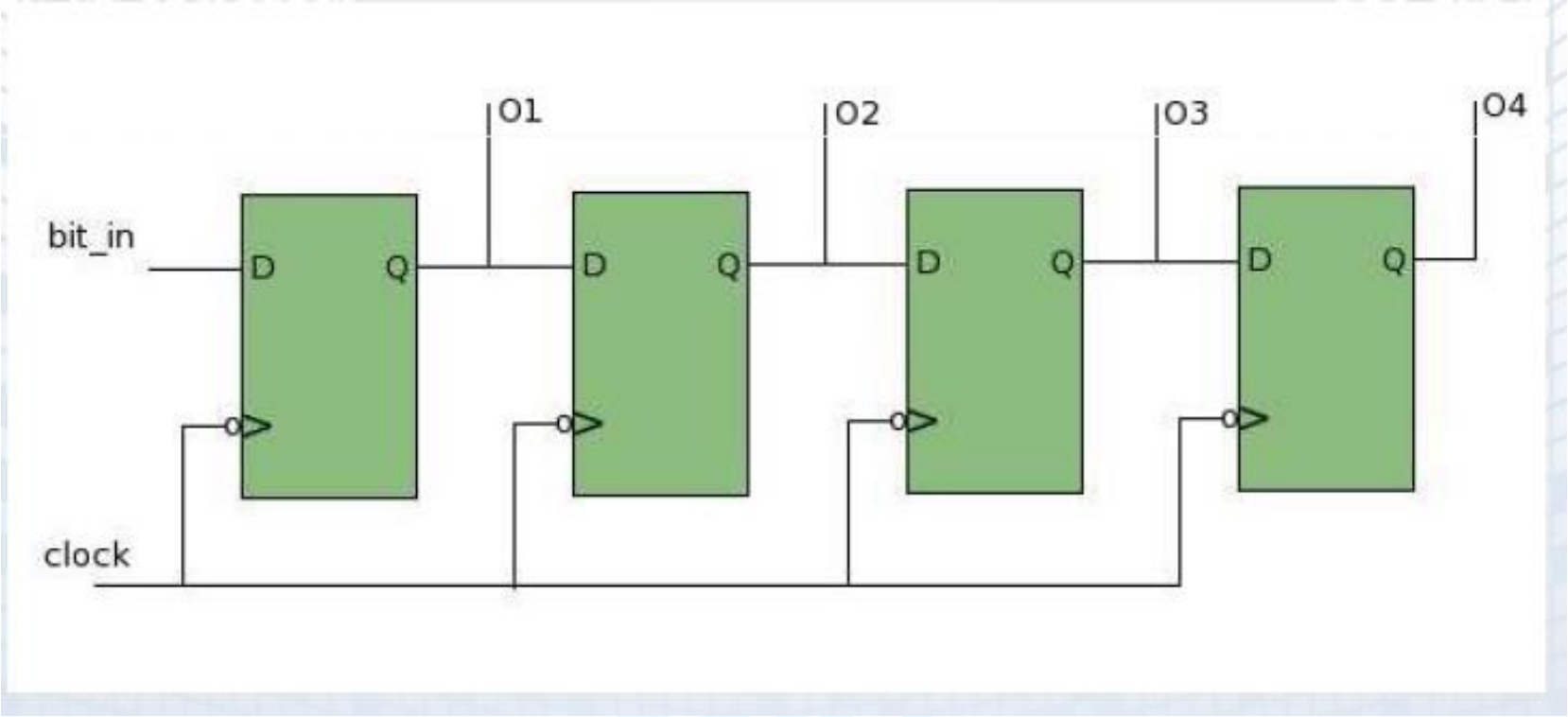
```
always @ (a or b or sel) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Event list using the * wildcard

```
always @ (*) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

blocking x non-blocking

- ✓ Exemplo: Registrador de deslocamento



blocking x non-blocking

- ✓ Atribuição blocking (=)

```
module shift_reg ( input clk, sin,
output reg [3:0] q);
always @ (posedge clk)
    begin
        q[0] = sin;
        q[1] = q[0];
        q[2] = q[1];
        q[3] = q[2];
    end
endmodule
```

- ✓ Atribuição non-blocking (<=)

```
module shift_reg ( input clk, sin,
output reg [3:0] q);
always @ (posedge clk)
    begin
        q[0] <= sin;
        q[1] <= q[0];
        q[2] <= q[1];
        q[3] <= q[2];
    end
endmodule
```

blocking x non-blocking

- ✓ Atribuição blocking (=)
- ✓ São avaliadas sequencialmente;
- ✓ Semelhante a linguagem de programação;
- ✓ USO NÃO RECOMENDADO PARA DESIGN, APENAS EM CÓDIGO PARA SIMULAÇÃO.
- ✓ Haverá casos de necessidades do uso.

- ✓ Atribuição non-blocking (<=)
- ✓ São avaliados em paralelo;
- ✓ Assemelha-se ao funcionamento do hardware;
- ✓ USAR SEMPRE QUE POSSÍVEL PARA CÓDIGOS SINTETIZÁVEIS.

Agenda

- ✓ O que é Verilog?
- ✓ Tipos de dados Verilog e operadores;
- ✓ Evento e lista de eventos;
- ✓ Código sintetizável Verilog;
- ✓ Boas práticas de projeto;

Escrevendo um código sintetizável

- ✓ A maioria dos comandos utilizados para simulação, não são sintetizáveis;
- ✓ As ferramentas de síntese conseguem abstrair alguns comandos de Verilog.

Comandos Verilog	Descrição
$display	Imprime a sentença no terminal;
$signed	Usado com o $display para imprimir números com sinal
$unsigned	Usado com o $display para imprimir números sem sinal
real	Tipo de registrador utilizado para guardar números reais;
initial	Só utilizados em simulações
delays	Geralmente utilizado em simulações;

Escrevendo um código sintetizável

- ✓ Construções suportadas para síntese:
  - module;
  - Portas (input, output e inout);
  - Nets (wire, wand, wor, supply0, supply1);
  - Register (reg, integer);
  - Parâmetros (parameter), limitados a inteiros. Redefinição de parâmetros não são suportados por algumas ferramentas;
  - Instanciação de módulos;
  - Portas primitivas (and, nand, or, nor, xor, buf, not, bufif1, bufif0, notif1, notif0);

Escrevendo um código sintetizável

- ✓ Construções suportadas para síntese:
  - module;
  - Portas (input, output e inout);
  - Nets (wire, wand, wor, supply0, supply1);
  - Register (reg, integer);
  - Parâmetros (parameter), limitados a inteiros. Redefinição de parâmetros não são suportados por algumas ferramentas;
  - Instanciação de módulos;
  - Portas primitivas (and, nand, or, nor, xor, buf, not, bufif1, bufif0, notif1, notif0);

Escrevendo um código sintetizável

- ✓ Construções suportadas para síntese:
  - assign;
  - function (Deve ser definida antes do uso);
  - task (Deve ser definida antes do uso);
  - always (Deve existir a lista de eventos);
  - begin-end;
  - disable;
  - = (blocking) e <= (non-blocking);
  - Valores inteiros;
  - if, if-else, case, casex, casez
  - for

Escrevendo um código sintetizável

- ✓ Operadores:
  - &, ~&, |, ~|, ^, ^~, ~^
  - ==, !=, <, >, <=, =>, !, &&, ||
  - <<, >>, {}, {{}}, ?:, +, -, *, /
- ✓ Vetores de bits: reg[3:0] a;

Boas Práticas de Projeto

- ✓ Seguindo boas praticas de descrição de Hardware o desenvolvedor:
  - Pode economizar várias horas de debug;
  - Irá desenvolver um código limpo;
  - Qualquer engenheiro entenderá sua descrição;
- ✓ Algumas práticas são sugeridas para serem seguidas:
  - Nomes de variáveis que tenha algum sentido;
  - Indentar o código;
  - Dividir o código em módulos e funções que façam sentido;
  - Usar parâmetros sempre que possível;
  - Comentar todo o código. Não seja preguiçoso!!!

Boas Práticas de Projeto

- ✓ O uso do ***always @ (*)*** requer que todo reg tenha um valor para todos os casos;
- ✓ Caso não aconteça isso, é criado um latch;
- ✓ Um caminho fácil é sempre colocar um valor ***default*** em qualquer ***case*** ou ***if-else*** que for desenvolver.

```
reg next_state, current_state;
always @ (*) begin
    if(current_state == STATE_INIT)
        next_state = STATE_1;
end
```

Wrong!

```
reg next_state, current_state;
always @ (*) begin
    next_state = current_state;
    if(current_state == STATE_INIT)
        next_state = STATE_1;
end
```

Correct!

Boas Práticas de Projeto

- ✓ Jamais misture blocos sequênciais com blocos combinacionais;

```
reg next_state, current_state;
always @ (posedge clk) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
    if(current_state == STATE_INIT)
        next_state = STATE_1;
end
```

Wrong!

```
reg next_state, current_state;
always @ (posedge clk) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
end

always @ (*) begin
    next_state = current_state;
    if(current_state == STATE_INIT)
        next_state = STATE_1;
end
```

Correct!

Boas Práticas de Projeto

- ✓ Alguns projetistas utilizam ***reset*** assíncrono, mas isso é muito arriscado;
- ✓ Violação de setup/hold;

```
reg next_state, current_state;
always @ (posedge clk or reset) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
end
```

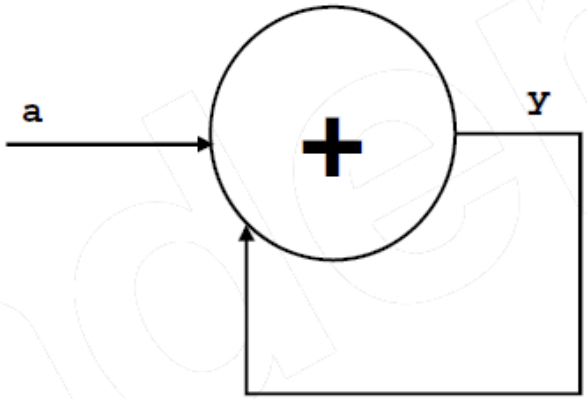
Asynchronous. Careful.

```
reg next_state, current_state;
always @ (posedge clk) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
end
```

Synchronous. Safe.

Boas Práticas de Projeto

- ✓ Nunca crie *loops* combinacionais;



`always @ (*) begin`
`  y = y + a`
`end`
	`always @ (posedge clk) begin`
	`  y <= y + a`
	`end`

Wrong!	Correct!

Boas Práticas de Projeto

- ✓ Nunca faça ***assign*** duas vezes na mesma variável no mesmo evento;

`wire y;`
`assign y = a&b;`
`assign y = a|b;`
	`wire y;`
	`if(c)`
	`  assign y = a&b;`
	`else`
	`  assign y = a|b;`

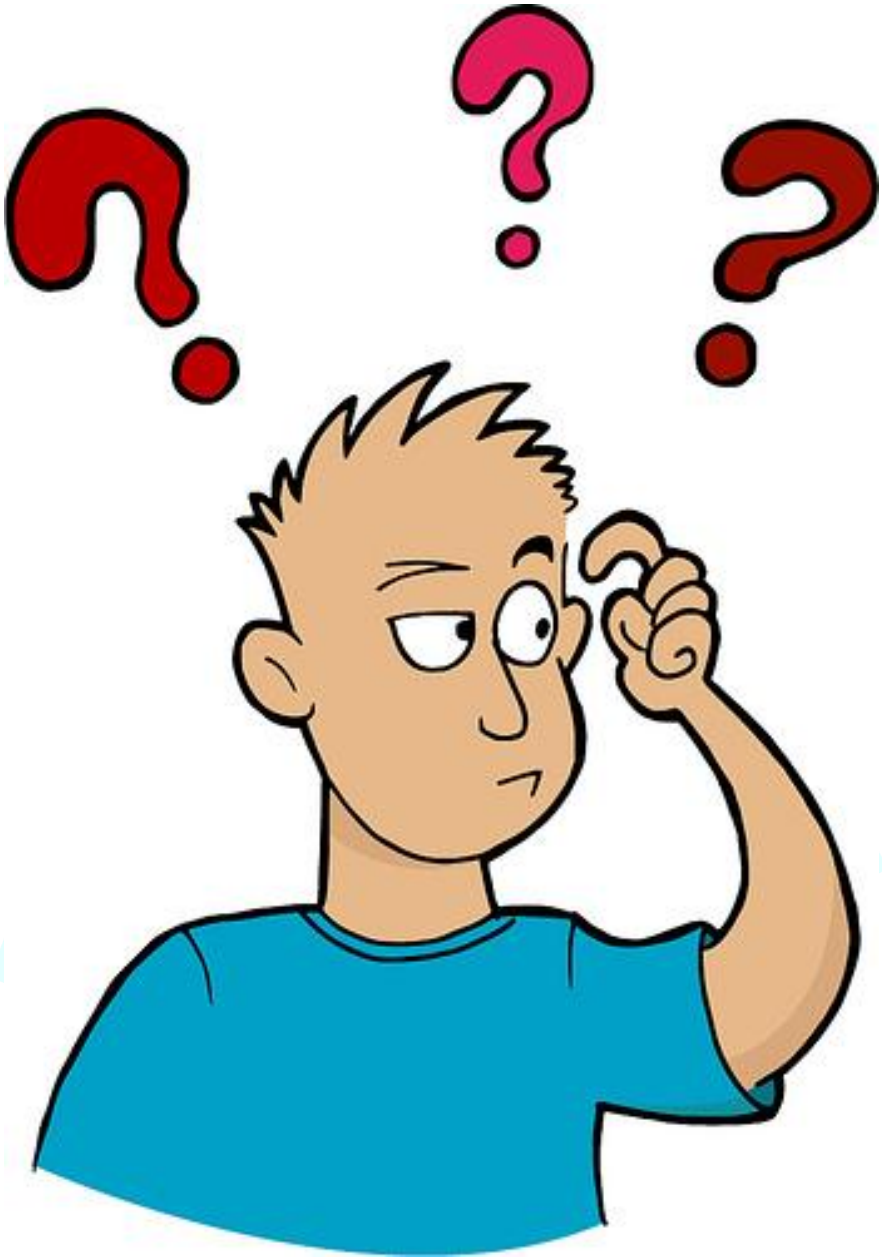
Wrong!	Correct!

Revisão

- ✓ HDL é uma linguagem de descrição de Hardware;
- ✓ HDL prover uma abstração muito importante para o desenvolvimento de um chip;
- ✓ Existem três tipos de dados em Verilog:
  - Wire – conexões físicas entre os blocos;
  - Registradores – "buferiza" elementos sequenciais;
  - Parâmetros – utilizados pelas ferramentas de sínteses para alocar espaços;
- ✓ Lógica combinacional executa linha por linha;
- ✓ Sequencial pode executar em paralelo;

Revisão

- ✓ Existem dois tipos de blocos:
  - Always
  - Initial
- ✓ blocking – usado na lógica combinacional; executa na hora;
- ✓ non-blocking – usado geralmente na lógica sequêncial; executa todos os comando em paralelo no evento desejado (posedge clk, negedge rst...);
- ✓ Comandos sintetizáveis;
- ✓ Boas praticas de descrição de Hardware;



Exercícios

- ✓ Descrever um MUX utilizando ***assign.***
- ✓ Descrever um MUX utilizando ***if-else.***
- ✓ Descrever um MUX utilizando ***case.***
- ✓ Use SystemVerilog procedural para desenvolver um simples registrador. Nesse exercício você terá a oportunidade de explorar tipos de dados como **always_ff**.
- ✓ Descreva um multiplexador de duas entradas de 8 bits e saída também de 8 bits. Aqui você terá que usar **always_comb** e **unique case**.

Exercícios

- ✓ Exercício: Escrever um código **sintetizável que descreva uma ALU** (unidade lógica aritmética) que realiza as funções de soma, subtração, negação e operações lógicas (AND, OR, NAND, NOR, XOR, XNOR).
- ✓ As entradas são de 4 bits.
- ✓ A saída só apresenta um resultado por vez, variando de acordo com uma entrada de seleção.
- ✓ Utilizar o máximo de conhecimentos possíveis.
- ✓ Duas versões, uma utilizando **always** e outra utilizando **assign**.
- ✓ Sejam eficientes mas não esqueçam dos detalhes.