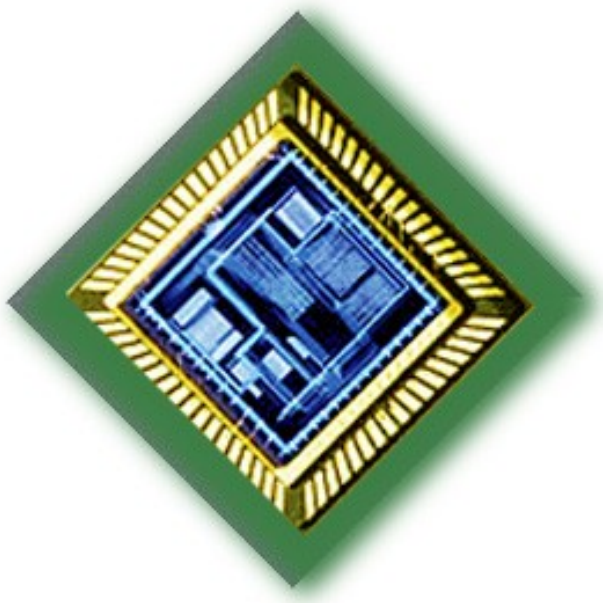
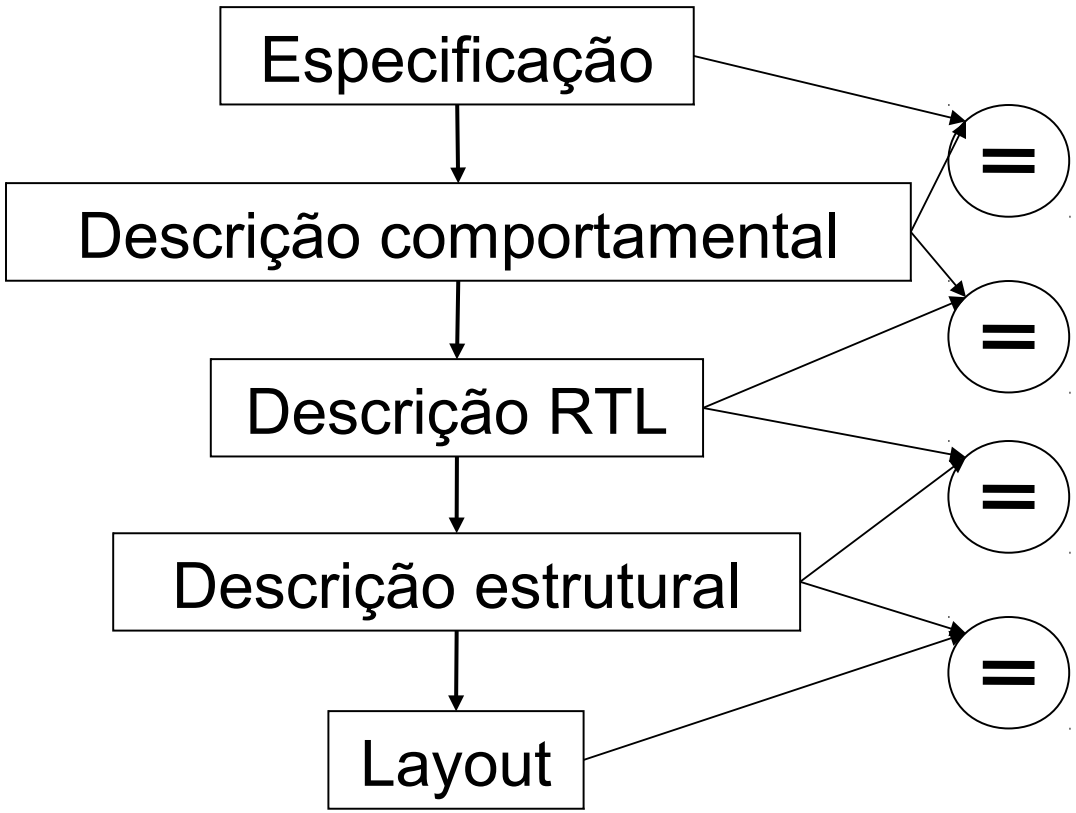


SystemVerilog para descrição RTL

Curso do Brazil-IP

Elmar Melcher
UFCG
elmar@dsc.ufcg.edu.br

Fluxo de projeto (simplificado)



Função

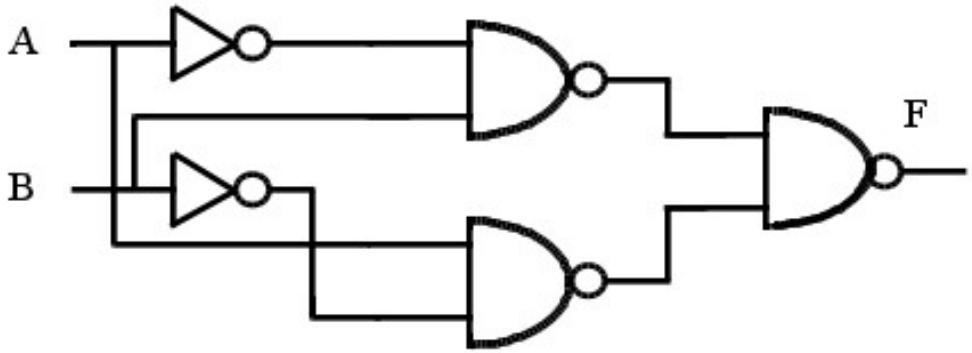
Função & Timing

Função & Timing

Função & Timing

consumo, área, etc.

Representação gráfica vs textual



+ intuitivo
+ hierarquia
– trabalhoso de fazer bem feito
– trabalhoso de modificar

```
if (function==1)
   y = a-b;
else
   y = a+b;
```

– aprender a linguagem
+ hierarquia
+ rápido de fazer, mas precisa de comentários
+ rápido de modificar
+ fácil de processar automaticamente

Representação gráfica vs textual

- Representação gráfica melhor para domínio estrutural.
- Representação textual melhor para domínio comportamental, atingindo um nível de abstração maior.

Propriedades desejáveis de uma HDL

- Expressar ações concorrentes
- Expressar tempo (atraso, clock)
- Permitir descrição comportamental, estrutural e física
- Permitir mesclar diferentes vistas de diferentes subsistemas
- Permitir simulação, síntese e verificação
- Ser fácil e seguro de usar

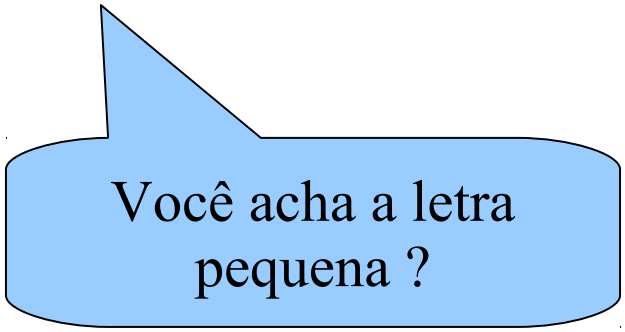
Exemplos de outras HDLs

- Verilog 1995, 2001, 2005
- SystemC
- VHDL (VLSI HDL - Very Large Scale Integration Hardware Description Language)
- Abel, Palasm, Cupl, OCCAM, Handle-C, ELLA

. . .

SystemVerilog vs. VHDL

```
module dpcm(input reset, clock,
            input signed [3:0] data_in,
            output logic signed [3:0] data_out);

   logic [3:0] prev;

   always_ff @(posedge clock)
     if(reset) begin
       data_out <= 0;
       prev <= 0;
     end
     else begin
       data_out <= data_in - prev;
       prev <= data_in;
     end

endmodule
```




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;

entity dpcm is
  port( reset: in std_logic;
        clock: in std_logic;
        data_in: in std_logic_vector(3 downto 0);
        data_out: out std_logic_vector(3 downto 0) );
end dpcm;

architecture behv of dpcm is

   signal prev: std_logic_vector(3 downto 0);

   begin
     process(clock)
     begin
       if (clock='1' and clock'event) then
         if (reset = '0') then
           data_out <= "0000";
           prev <= "0000";
         else
           data_out <= data_in - prev;
           prev <= data_in;
         end if;
       end if;
   end process;

end behv;
```

SystemVerilog vs. VHDL

```
module dpcm(input reset, clock,
            input signed [3:0] data_in,
            output logic signed [3:0] data_out);

   logic [3:0] prev;

   always_ff @(posedge clock)
     if(reset) begin
       data_out <= 0;
       prev <= 0;
     end
     else begin
       data_out <= data_in - prev;
       prev <= data_in;
     end

endmodule
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;

entity dpcm is
  port( reset: in std_logic;
        clock: in std_logic;
        data_in: in std_logic_vector(3 downto
        data_out: out std_logic_vector(3 down
end dpcm;

architecture behv of dpcm is

   signal prev: std_logic_vector(3 downto 0);

begin
  process(clock)
begin
        if (clock='1' and clock'event) then
```

O.K, mas agora tá faltando VHDL

SystemVerilog vs. VHDL

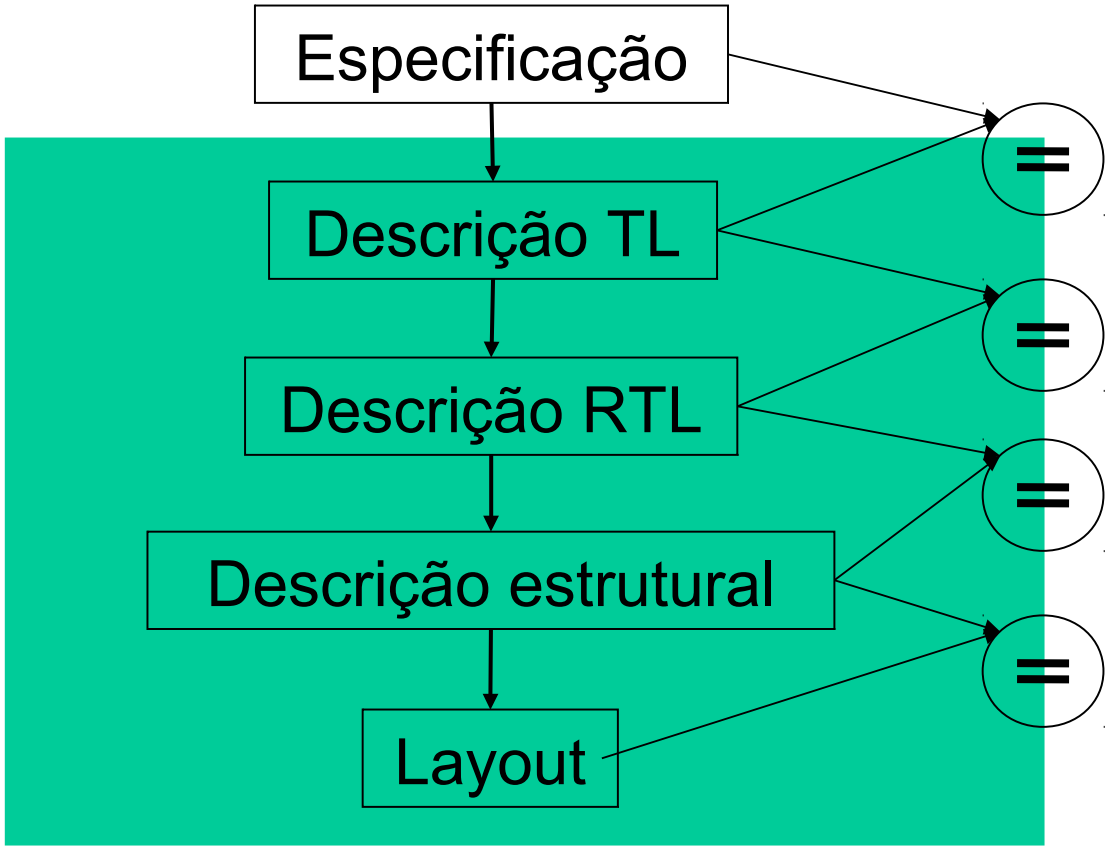
❑ linhas	17	31
❑ palavras	40	77
❑ letras	339	687

- ✓ Menos trabalho
- ✓ Menos erros
- ✓ Mais fácil de entender
- ✓ Mais espaço para comentários
- ➔ **Maior produtividade**

Propriedades de SystemVerilog

- Expressar ações concorrentes
- Expressar tempo (atraso, clock)
- Permitir descrição TL ✓, RTL ✓, estrutural ✓ e física ✓
- Permitir mesclar diferentes vistas de diferentes subsistemas
- Permitir simulação ✓, síntese ✓ e verificação ✓
- Ser fácil e seguro de usar

Fluxo de projeto usando SystemVerilog



Função

Função & Timing

Função & Timing

Função & Timing

consumo, área, etc.

Breve Histórico

- Verilog, 1981
- SystemVerilog, padrão IEEE em 2005
- OVM, Janeiro 2008

Código bonito

- ✔ 50% é comentário
- ✔ Todas as declarações tem comentário
- ✔ Um **module** ou **class** não ultrapassa 100 linhas (uma página na tela)
- ✔ Identificadores refletem o que é o objeto identificado
- ✔ Indentação consistente

- ➔ regras do SRS da Motorola
- ➔ regras do Brazil-IP Network
  http://lad.dsc.ufcg.edu.br/fenix/metodologia

Código bonito

Identificadores

- Evite nomes longos demais:

  fpop_rs1 no lugar de **floating_point_opcode_rs1**

- Evite confusão entre '**0**' e '**O**', '**1**' e '**l**'.

- Use capitalização consistente.
  Estilo C: **packet_addr, data_in**
  Estilo Pascal: **PacketAddr, DataIn**
  Estilo Modula: **packetAddr, dataIn**

Código bonito

Sufixo de identificadores

***_clk**	sinal de relógio
***_next**	sinal antes de registrá-lo
***_n**	sinal ativo nível baixo
***_xi**	entrada do circuito
***_xo**	saída do circuito

Comentários

**/* Comentario atravessando
varias linhas */**

// Comentario ateh o fim da linha

Cuidado com acentos á é ô à etc.

Convenções Léxicas

- **Números**

  Decimal, hexadecimal, binário com tamanho em bits:
  6'd33, 8'hA6, 4'b1101
  default: decimal positivo sem tamanho
  permitido usar _ + -
  8'b1001_0011

- **Cadeias de caracteres**

  "Delimite usando aspas numa mesma linha"
  Limitado a 1024 caracteres

Convenções Léxicas

- **Identificador**
  - A ... Z
  - a ... z
  - Underscore
- Primeiro caractere de um identificador não pode ser um dígito
- **SystemVerilog diferencia letras maiúsculas de minúsculas (case sensitive)**

Tipos de dados (sintetizáveis)

- Variável de 1 bit
  ```
  logic nome;
  ```
- Um vetor de bits
  ```
  logic [msb : lsb] nome;
  ```
- Enumeração
  ```
  enum logic [size-1 : 0] {A, B, C} nome;
  ```
- Exemplos
  ```
  logic [3:0] cabo;   //Um cabo de 4 fios
  enum logic [1:0] {red, yellow, green} lamp;
  ```

Tipos de dados (sintetizáveis)

- Memória

```
logic [msb : lsb] memory1 [upper : lower];
```

- Exemplo

```
logic [3:0] mem [63:0];
// An array of 64 4-bit registers

logic mem [4:0];
// An array of 5 1-bit registers
```

Outros Tipos de Dados

integer j; //32 bits incluindo 'z' e 'x' e sinal (compl. 2)

Tipos de Dados

Tipos sintetizáveis: default é sem sinal.

Tipos não sintetizáveis: default é com sinal.

Ex.: **logic [15:0] value;**
value = [0 : 65,536].

Ex.: **shortint value;**
value = [-32,768 : 32,767]

Operadores lógicos unários

- z é tratado como x

~	negação bit a bit
!	negação lógica
&	“e” bit a bit
|	“ou” bit a bit
^	“ou” exclusivo
-	negativo
+	positivo

Operadores aritméticos binários

- Se um dos bits envolvidos for x ou z, todo o resultado é x.

*	multiplicação
/	divisão
%	resto da divisão
+	soma
-	subtração

Operadores lógicos binários

- Deslocamento lógico (com preenchimento com ‘0’)
- Deslocamento negativo não pode

<< (esquerda) **>>** (direita)

- Deslocamento aritmético

<<< (esquerda) **>>>** (direita)

- Comparação

== **!=** **<** **<=** **>** **>=**

- Lógico

| **&** **^** **||** **&&**

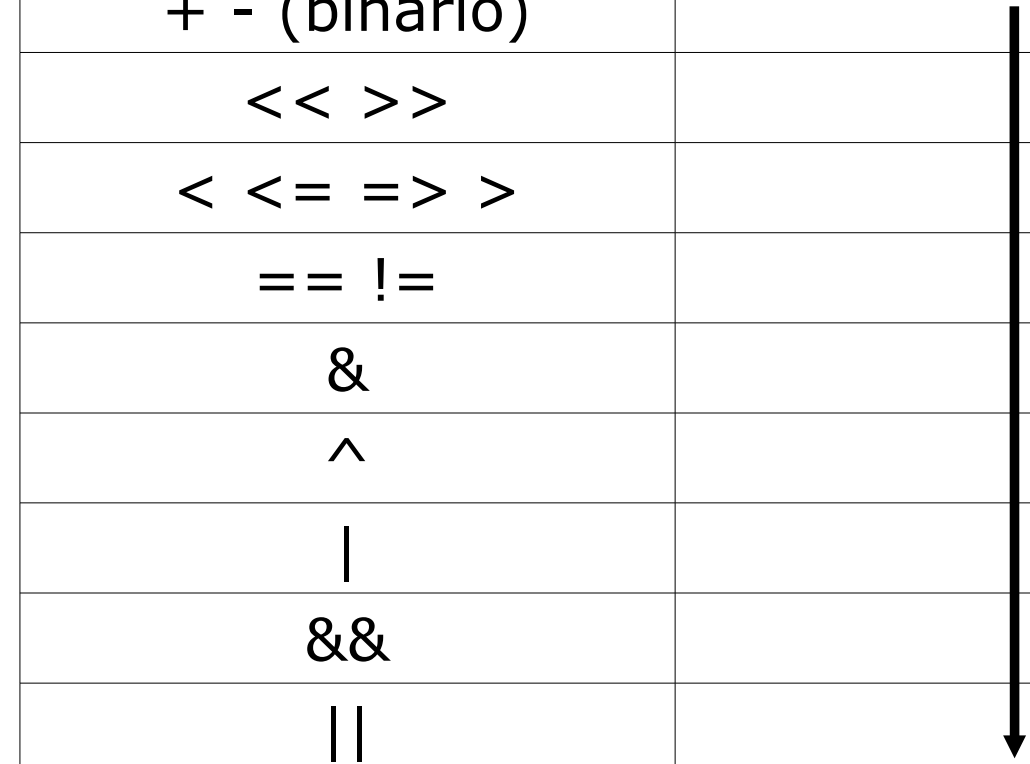
Operador condicional

<condição> ? <expressão_verdadeira> : <expressão_falsa>

- Para expressões simples é útil.
- Usá-lo aninhando é suicídio.
- Prefira estruturas **if ... else**.

Precedência de operadores

+ - ! ~ (unario)	mais
+ - (binario)	
<< >>	
< <= => >	
== !=	
&	
^	
|	
&&	
||	
?: (condicional)	menos



Diretivas de Compilação

- `define – (Similar a #define em C) usado para definir parâmetro global
- Exemplo:

```
`define BUS_WIDTH 16
logic [`BUS_WIDTH - 1 : 0] SystemBus;
```

- `include – usado para incluir outro arquivo
- Exemplo:

```
`include "./fulladder.svh"
```

Compiler Directives

- `parameter` – used to define global parameter; better than `` `define ``
- Example:

```
parameter BUS_WIDTH = 16;
 logic [ BUS_WIDTH - 1 : 0 ] System_Bus;
```

Module

Definição geral

```
module module_name (
    port_list );
    …
    variable declaration;
    …
    description of behavior;
endmodule
```

Exemplo

```
module HalfAdder (
  input A, B,
  output logic Sum, Carry);

  /* ta vazio */

endmodule
```

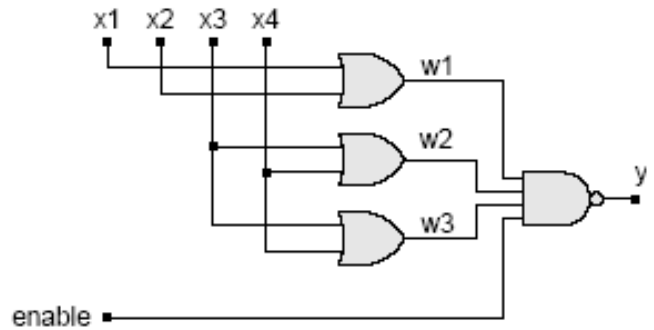
Instanciação de Primitivas

- Formato (portas lógicas primitivas):

```
and G2(Carry, A, B);
```

  - Primeiro parâmetro (Carry) – Output
  - Outros parâmetros (A, B) - Inputs

```
module or_nand_1 (
  input enable, x1, x2, x3, x4,
  output logic y);
  logic w1, w2, w3;
  or (w1, x1, x2);
  or (w2, x3, x4);
  or (w3, x3, x4);
  nand (y, w1, w2, w3, enable);
endmodule
```



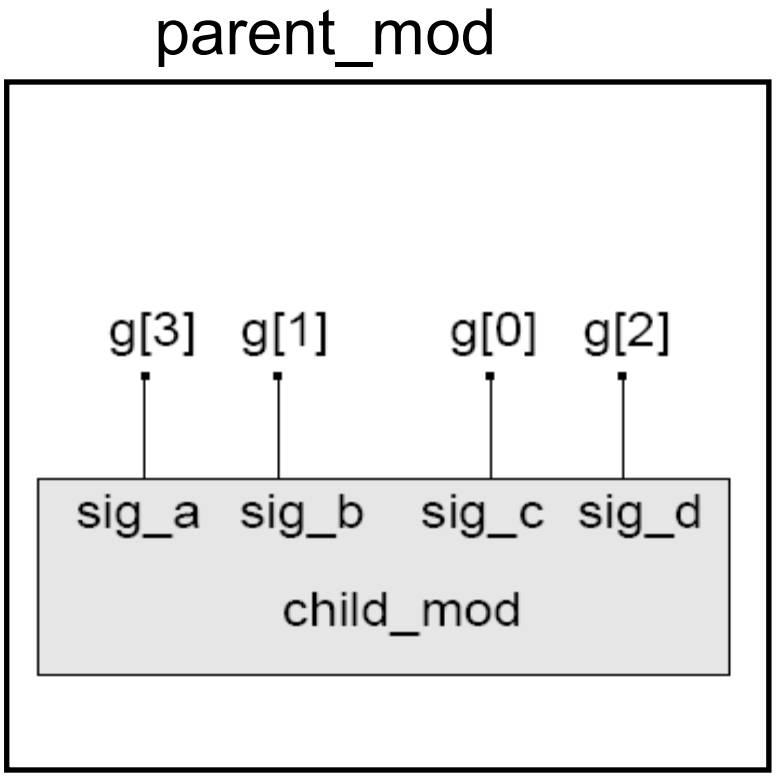
Pre-synthesis

Conexão de instâncias de *modules*

- Connexão por posição

```
module child_mod(
  input sig_a, sig_b,
  output logic sig_c, sig_d);
// descrição do module
endmodule
```

```
module parent_mod;
   logic [3:0] g;
    child_mod U1(g[3], g[1],
                 g[0], g[2]);
    // ordem é significativa

 endmodule
```

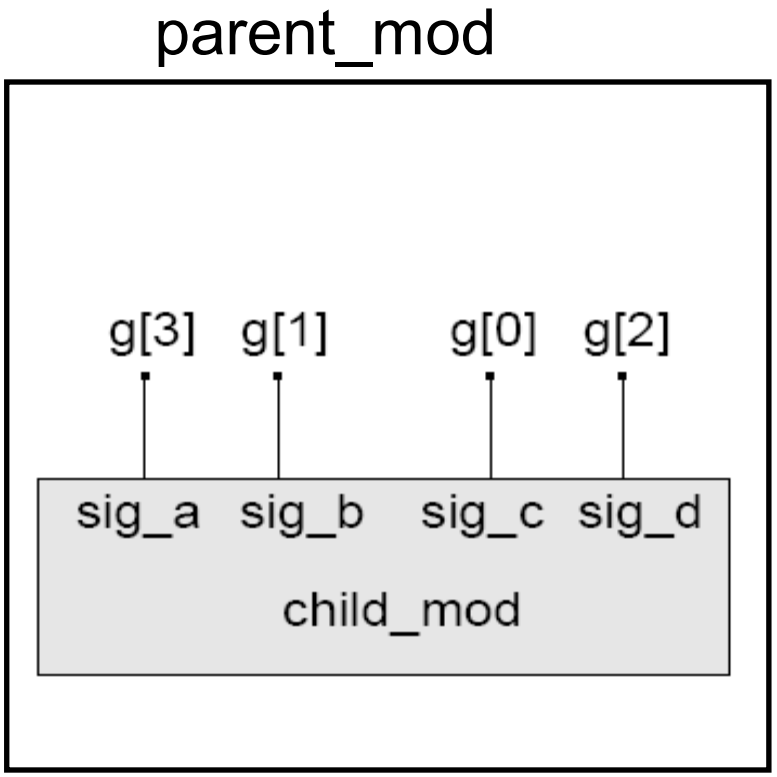


Conexão de instâncias de *modules*

- Connexão explícita

```
module child_mod(
  input sig_a, sig_b,
  output logic sig_c, sig_d);
// descrição do module
endmodule
```

```
module parent_mod;
   logic [3:0] g;
   child_mod U1(.sig_c(g[0]),
                .sig_b(g[1]),
                .sib_d(g[2]),
                .sig_a(g[3]));
   // ordem arbitrária
endmodule
```

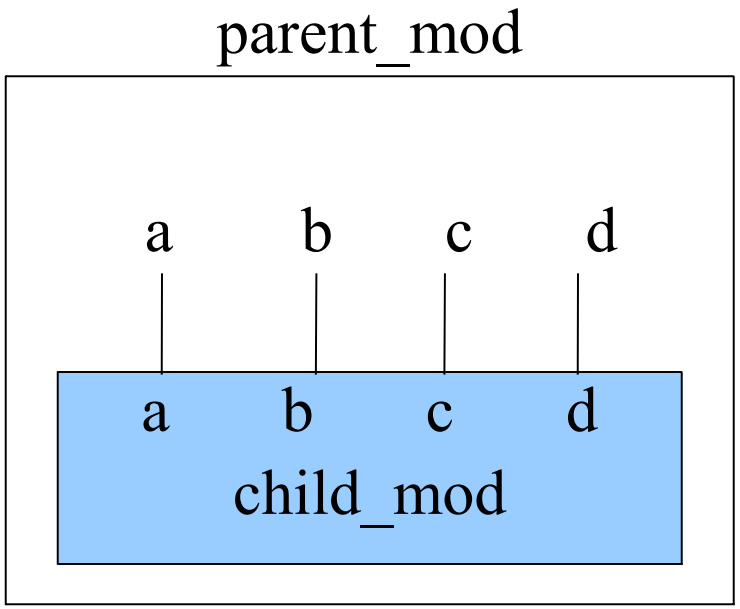


Conexão de instâncias de *modules*

- Connexão por casamento

```
module child_mod(
  input a, b,
  output logic c, d);
// descrição do module
endmodule
```

```
module parent_mod;
   logic a,b,c,d;
    child_mod U1( .* );

 endmodule
```



Parametrização de *modules*

- Na hora de declarar o module:

module dpcm #(parameter N=10) (input a, ..., output b, ...);

- Na hora de instanciá-lo:

dpcm dpcm_i #(.N(20)) (.a(x), ..., .b(y), ...);

Construções Procedurais

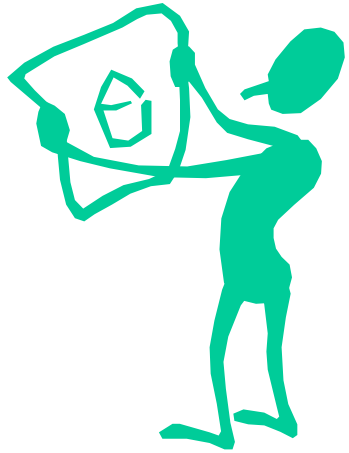
- Existem dois:
  - initial : Executa uma única vez no início da simulação, NÃO sintetizável
  - always_comb : Executa repetidamente, sintetizável
- Exemplo:

```
…
initial begin
  Sum <= 0;
  Carry <= 0;
end
…
```

```
…
always_comb begin
  Sum <= A ^ B;
  Carry <= A & B;
end
…
```

Descrição de Lógica Combinacional

- Atribuições
- Estruturas Condicionais
  - if
  - case
- Exemplo
  - Multiplexador

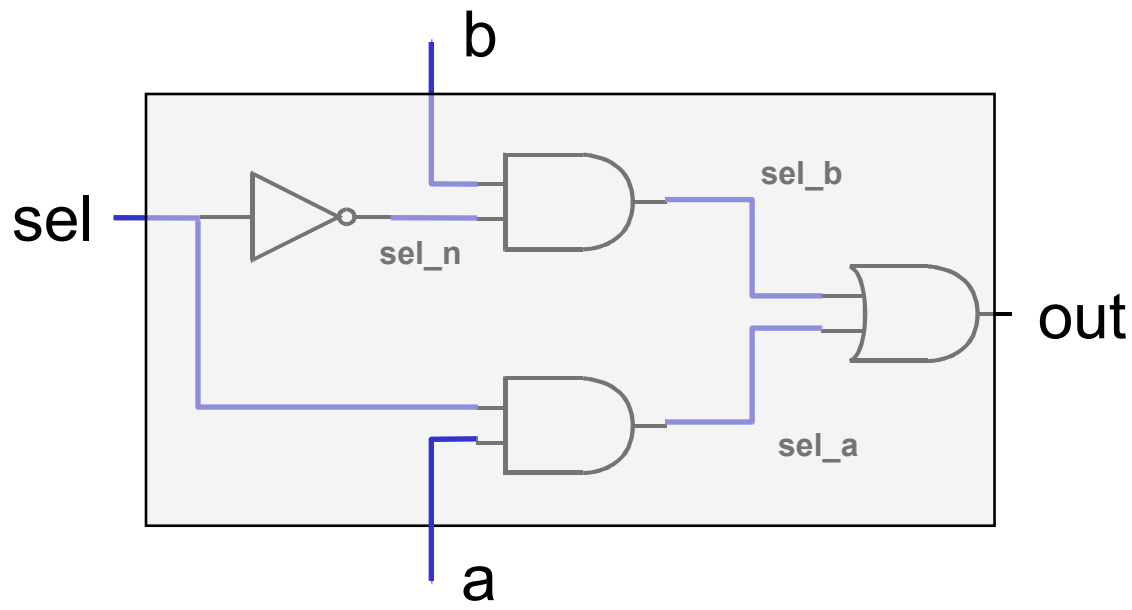


Atribuição

- **Fluxo de dados**: Representa sinais de saída em função de sinais de entrada

- Exemplo:

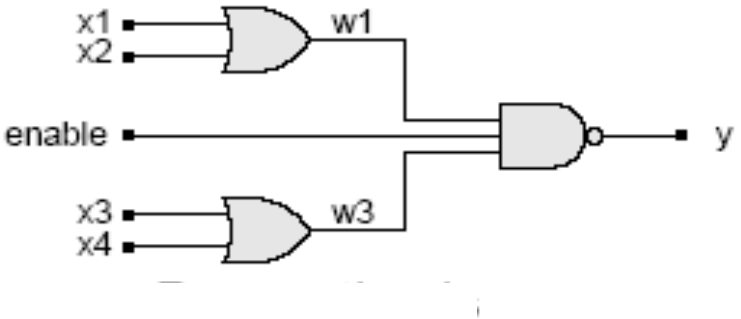
```
always_comb out <= (sel & a) | (~sel & b);
```



Atribuição

Exemplo:

```
module or_nand_2 (
  input enable, x1, x2, x3, x4,
  output logic y);
  always_comb y <= !(enable & (x1 | x2) & (x3 | x4));
endmodule
```



if

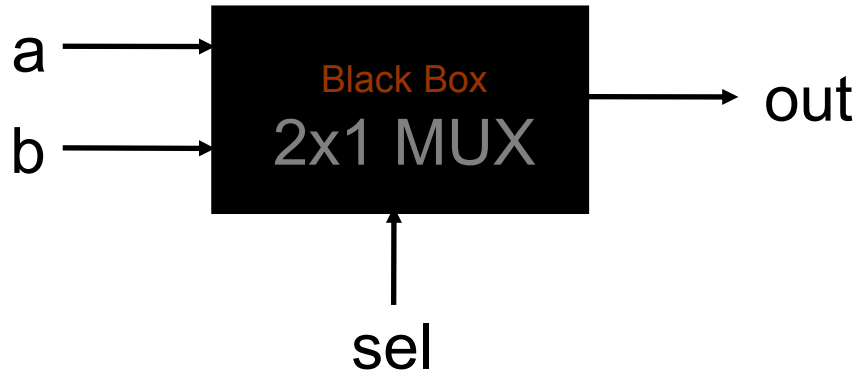
- Formato:

```
if (condition)
  procedural_statement
else if (condition)
  procedural_statement
else
   procedural_statement
```

if

- **Exemplo 1:**

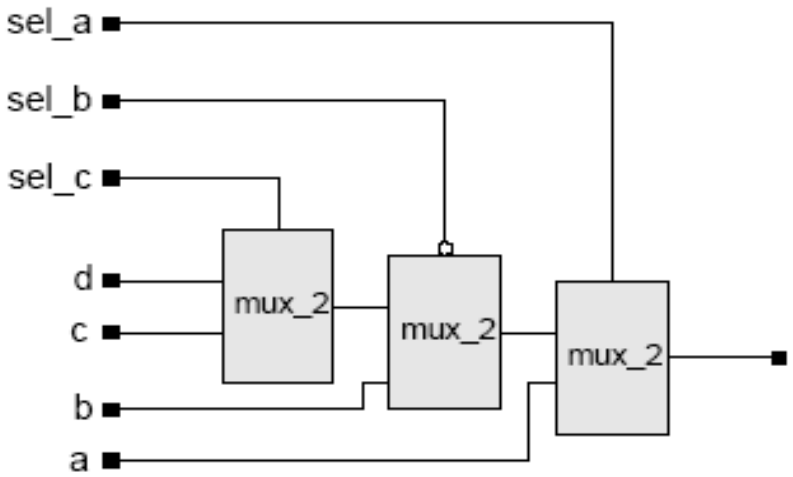
```
module mux_2x1(input a, b, sel,
               output logic out);
  always_comb
     if (sel == 1) out <= a;
      else out <= b;

endmodule
```



if

- Exemplo 2:

```
module mux_4pri (
   input a, b, c, d, sel_a, sel_b, sel_c,
   output logic y);

   always_comb
   begin
     if (sel_a == 1) y <= a; else
     if (sel_b == 0) y <= b; else
     if (sel_c == 1) y <= c; else
        y <= d;
   end
endmodule
```



case

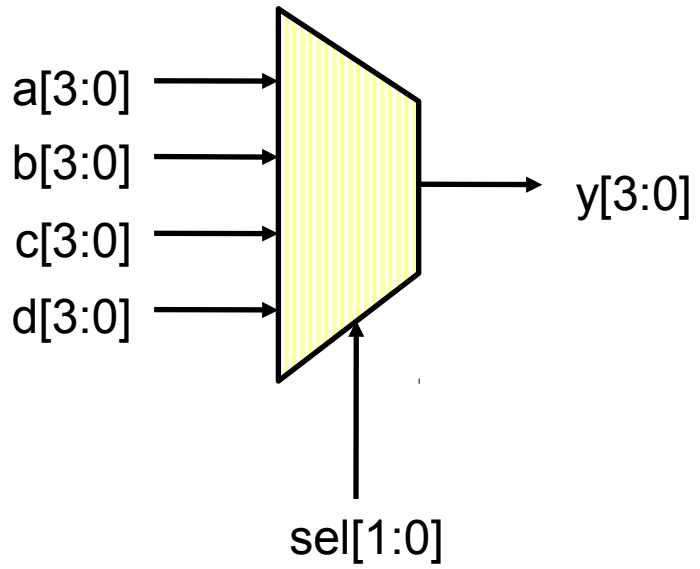
- Instrução Case
- Exemplo:

```
case (X)
   2'b00: Y <= A + B;
   2'b01: Y <= A – B;
   2'b10: Y <= A / B;
endcase
```

case

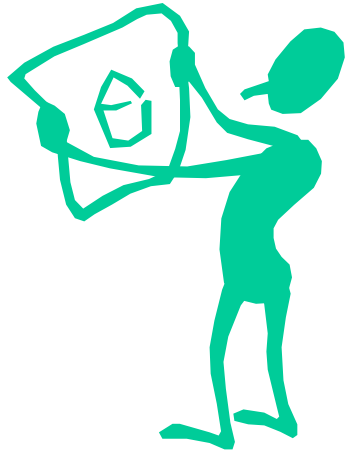
Exemplo

```
module mux_4bits (
  input [3:0] a, b, c, d,
  input [1:0] sel,
  output logic [3:0] y);

  always_comb
     case (sel)
      0: y <= a;
      1: y <= b;
      2: y <= c;
      default: y <= d;
    endcase
endmodule
```



Descrição RTL

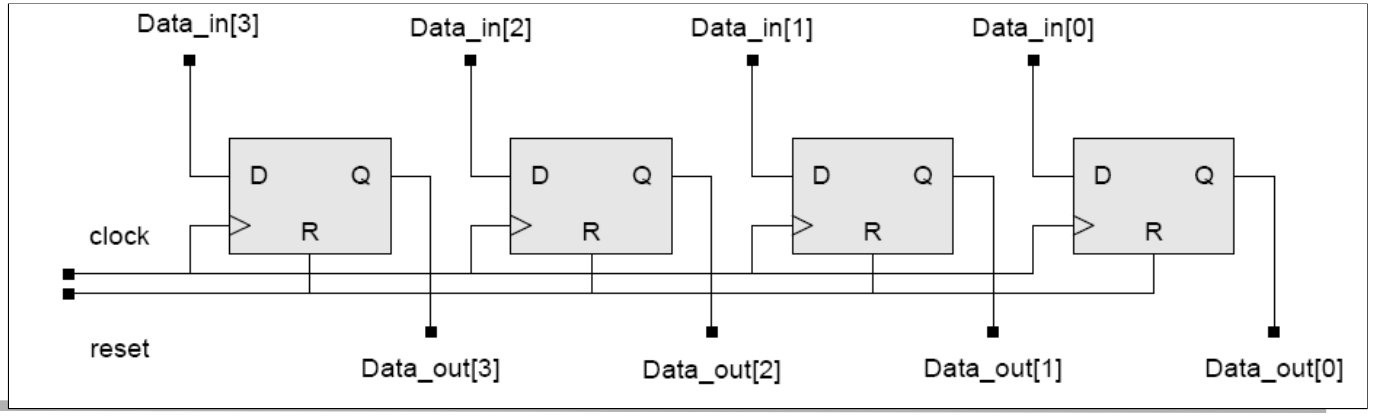
- Sistemas seqüenciais síncronos
  - reset síncrono e assíncrono
- Exemplos
  - contador
  - registro de deslocamento
- FSM



Reset assíncrono

- Regra Geral: Uma variável será sintetizada como flip-flop se o seu valor é atribuída só no momento da transição de um sinal.
- Exemplo:

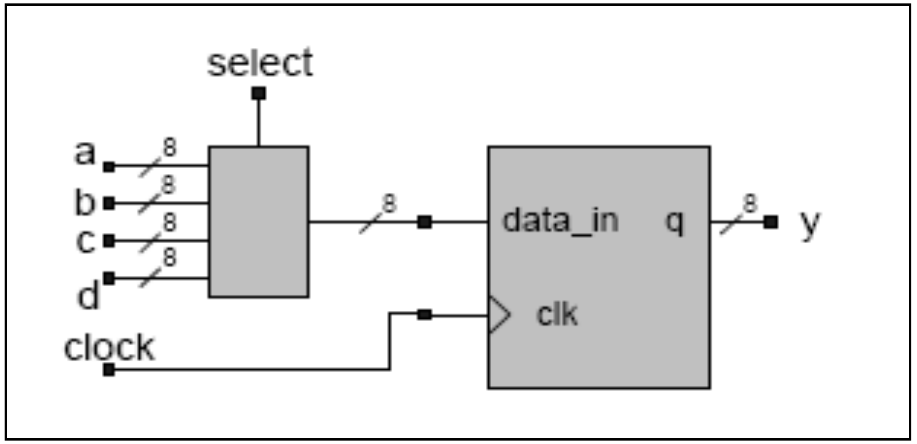
```
module D_reg4a (
  input [3:0] Data_in,
  input clock, reset,
  output logic [3:0] Data_out);

  always_ff @ (posedge reset or posedge clock)
     if (reset) Data_out <= 4'b0;
     else Data_out <= Data_in;
endmodule
```



Reset síncrono

- Combinational logic that is included in a synchronous behavior will be synthesized with registered output.
- Example:

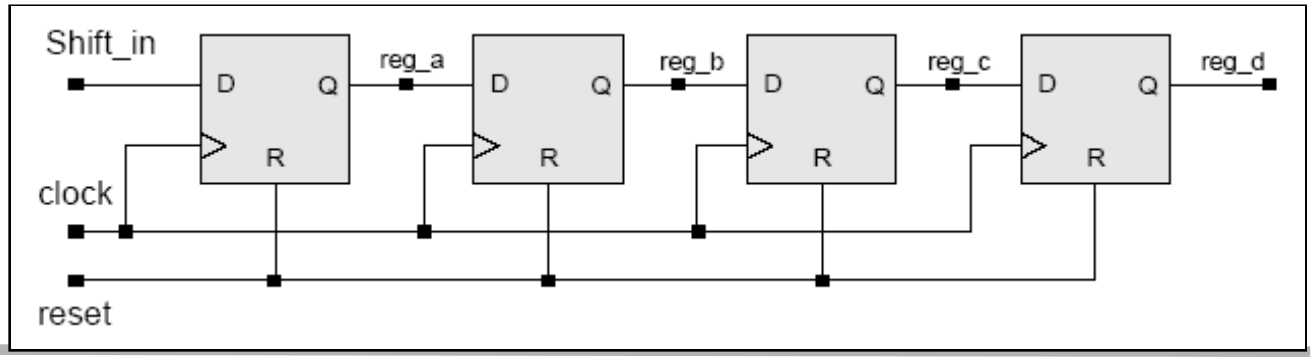
```
module mux_reg (
    input [7:0] a, b, c, d,
    output logic [7:0] y,
    input [1:0] select);

    always_ff @ (posedge clock)
      case (select)
        0: y <= a;
        1: y <= b;
        2: y <= c;
        default: y <= d;
     endcase
  endmodule
```



Registrador de deslocamento

- Shift register can be implemented knowing how the flip-flops are connected

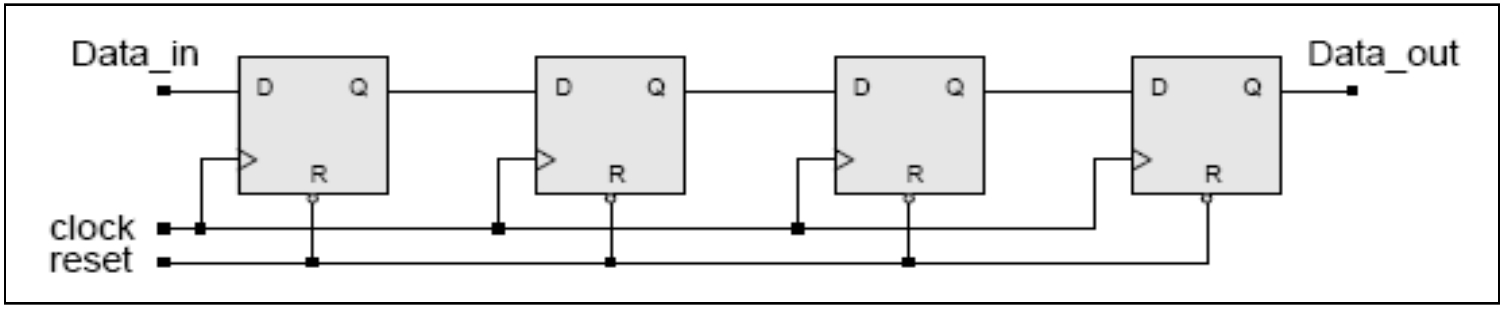
```
always_ff @ (posedge clock) begin
  if (reset)   begin
    reg_a <= 0;
    reg_b <= 0;
    reg_c <= 0;
    reg_d <= 0;
  end
  else begin
    reg_a <= Shift_in;
    reg_b <= reg_a;
    reg_c <= reg_b;
    reg_d <= reg_c;
  end
end
```



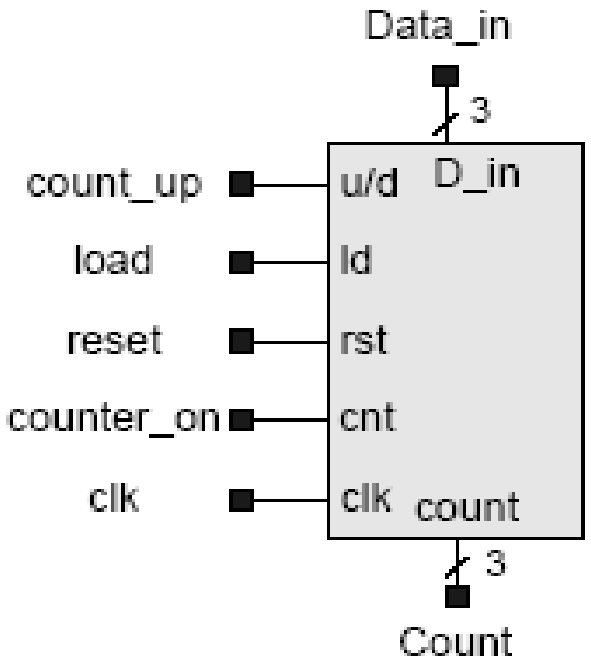
Registrador de deslocamento

- Shift register can be implemented using concatenation operation referencing the register outputs

```
module Shift_reg4 (
  input Data_in, clock, reset,
  output logic Data_out);
  logic [3:0] Data_reg;
  always_comb Data_out = Data_reg[0];
  always_ff @ (negedge reset or posedge clock) begin
    if (reset == 0) Data_reg <= 0;
    else Data_reg <= {Data_in, Data_reg[3:1]};
  end
endmodule
```



Contador



Functional Specs.

- Load counter with Data_in when load = 1
- Counter counts when counter_on = 1
  - counts-up when count_up = 1
  - Counts-down when count_up = 0

Contador

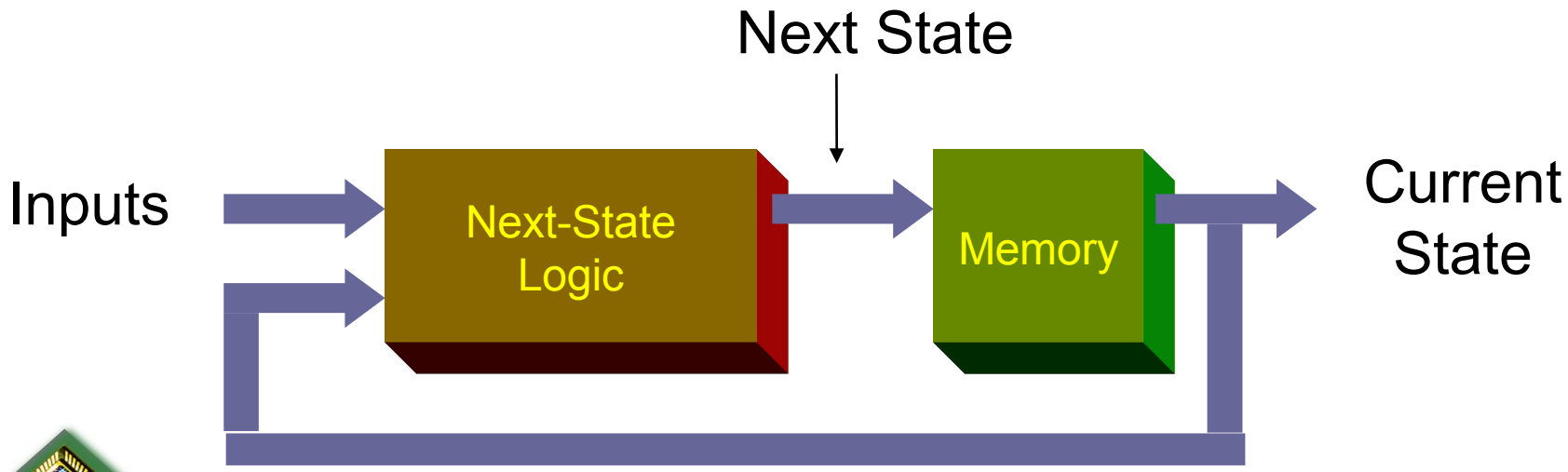
```
module up_down_counter (
    input clk, rst, ld, ud, cnt,
    input [2:0] D_in,
    output logic [2:0] count);

    always_ff @ (posedge rst or posedge clk)
        if (rst)
            count <= 0;
        else if (ld)
            count <= D_in;
        else if (cnt) begin
             if (ud)
                count <= count +1;
             else count <= count –1;
        end
endmodule
```

FSM

- Quando a sequência das ações no seu projeto dependem do estado de um elemento seqüencial, uma máquina de estados finitos (FSM) pode ser implementada.
- FSMs são amplamente usadas em aplicações que requerem uma atividade seqüencial
  - Exemplos:
    - Sequence Detector
    - Fancy counters
    - Traffic Light Controller
    - Data-path Controller
    - Device Interface Controller
    - etc.

FSM

- All state machines have the general feedback structure consisting of:
  - Combinational logic implements the next state logic
    - Next state (ns) of the machine is formed from the current state (cs) and the current inputs
  - State register holds the value of current state



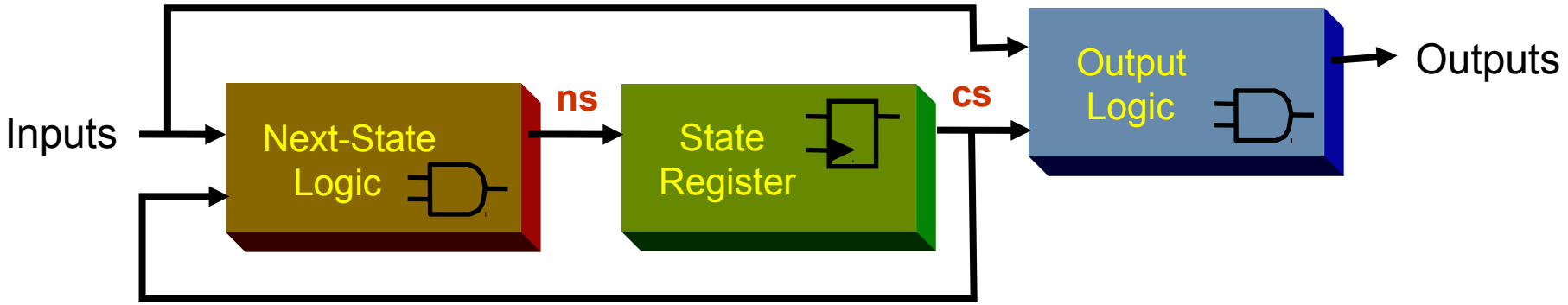
Tipos de FSM

Moore State Machine

- Next state depends on the current state and the inputs but the output depends only on the present state
  - next_state(t) = h(current_state(t), input(t))
  - output = g(current_state(t))

Tipos de FSM

Mealy State Machine

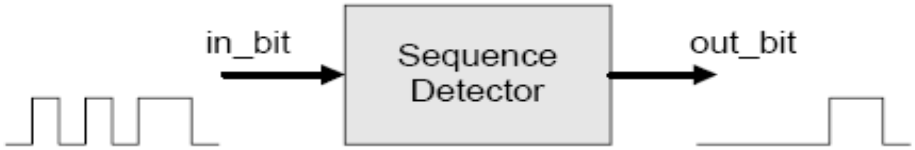


- Next state and the outputs depend on the current state and the inputs
  - next_state(t) = h(current_state(t), input(t))
  - output(t) = g(current_state(t), input(t))

Estrutura típica de uma FSM

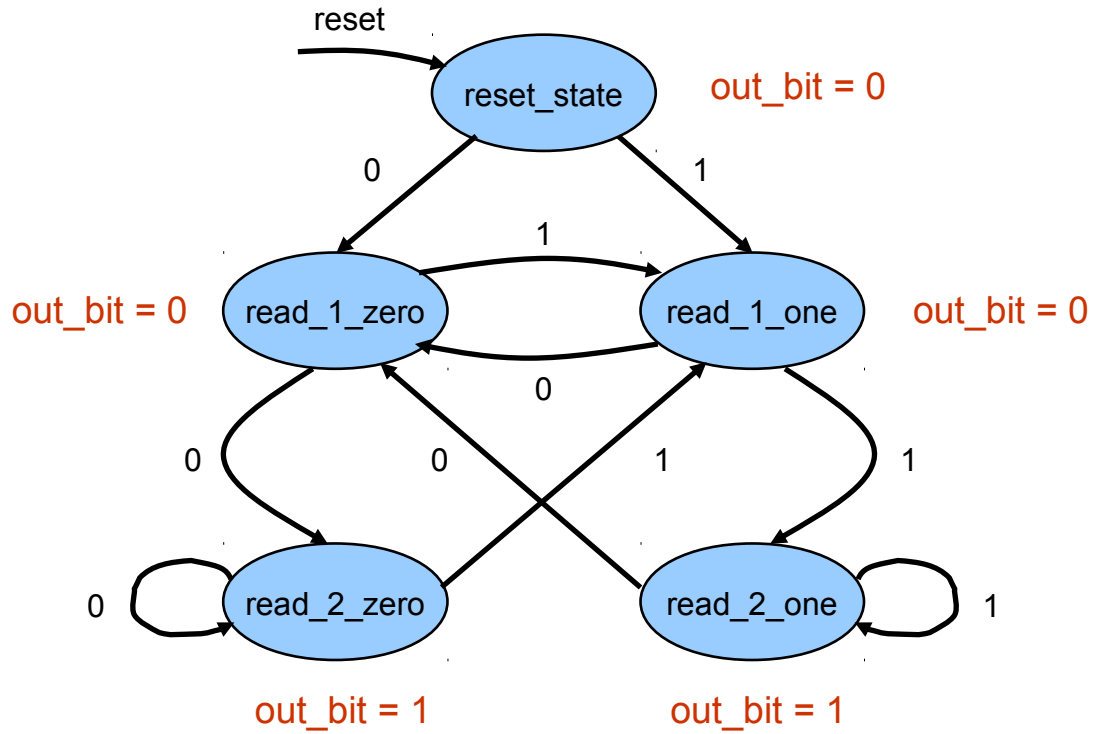
```
module mod_name (  input … ,  output … );
        parameter size = … ;
        enum logic [size-1: 0] {state_0, state_1, ... } state;

        always_ff @ (negedge reset or posedge clk)
                if (reset == 0) state <= state_0;
                else
                 case  (state)
                   state_0: state <= state_1;
                   state_1: ...
                   ...
                   default: state <= state_0;
                 endcase

       always_comb ...

endmodule
```

Detector de Seqüência



Functionality: Detect two successive 0s or 1s in the serial input bit stream

FSM Flow-Chart



Detector de Seqüência

```
module seq_detect (
  input clock, reset, in_bit,
  output logic out_bit);

  enum logic [2:0] { reset_state,
     read_1_zero, read_1_one,
     read_2_zero, read_2_one }  state;

  always_ff @ (posedge clock or posedge reset)
    if (reset)
      state <= reset_state;
    else
     case (state)
       reset_state:
            if (in_bit == 0)
                state <= read_1_zero;
            else
                state <= read_1_one;
       read_1_zero:
            if (in_bit == 0)
                state <= read_2_zero;
            else
                state <= read_1_one;
       read_2_zero:
            if (in_bit == 0)
                state <= read_2_zero;
            else
                state <= read_1_one;
       read_1_one:
          if (in_bit == 0)
              state <= read_1_zero;
          else
              state <= read_2_one;
```

Detector de Seqüência

```
      read_2_one:
         if (in_bit == 0)
            state <= read_1_zero;
         else
            state <= read_2_one;
      default: state <= reset_state;
  endcase

  always_comb out_bit <= (
      (state == read_2_zero) ||
      (state== read_2_one));
endmodule
```