Circuitos Digitais

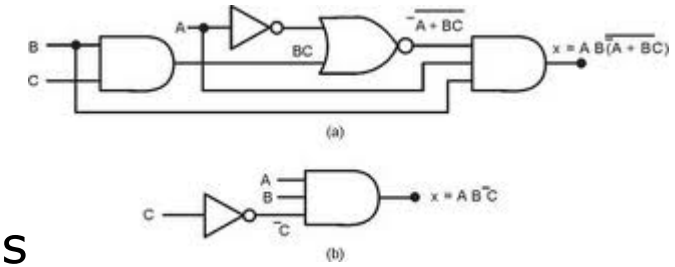
Tutorial Linguagem Verilog

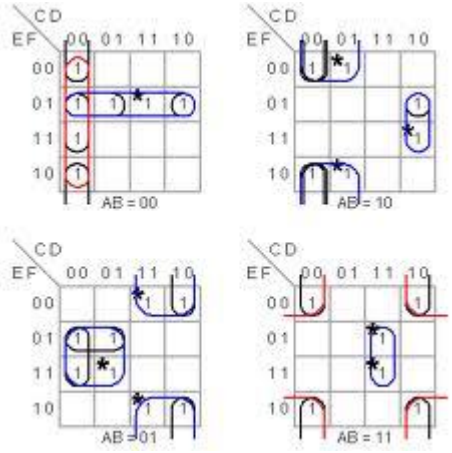
Baseado em:
http://doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

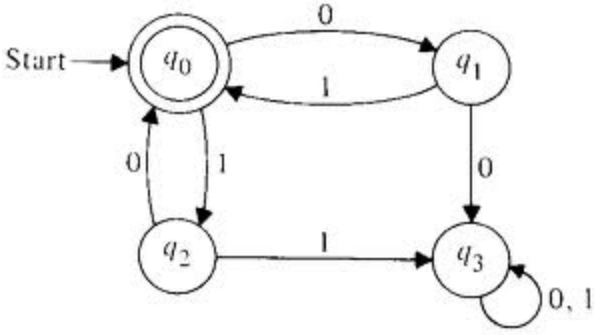
Universidade Federal Rural de Pernambuco
Professor: Abner Corrêa Barros
abnerbarros@gmail.com

Projetos sem HDL

- Especificações
- Modelagem
  - Máquina de Estados
  - Tabela Verdade/Tabela de transição
- Simplificação
  - Mapa de karnaugh
- Circuito Elétrico







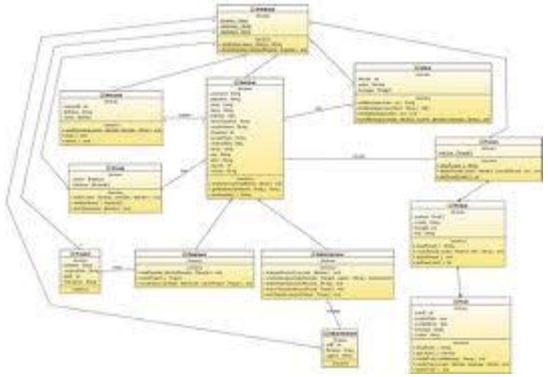
Projeto com HDL

- Especificação (specs)
- Modelagem
- Projeto de Alto Nível
- Codificação RTL
- Verificação
- Sintese.

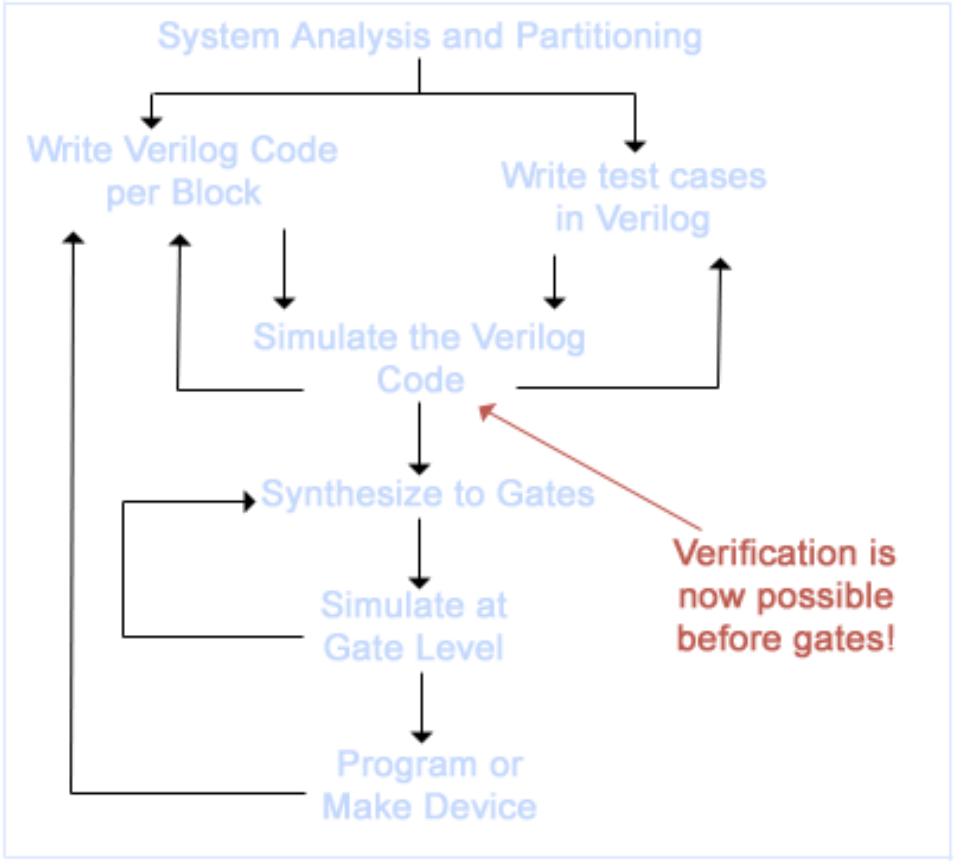
SPECIFICATION:

Type: Feathering:	3 Blades - Left or Right Handed Rotation
Diameter - Nominal:	15.50", 16", 16.50", 17" & 18.50" Diameter at the tips will be ~ ½" greater than nominal
Pitch Range Degrees:	18 to 24 degrees. Reverse always 24 degrees
Boss / Castings:	316 Stainless Steel ex Rod or Investment Cast
Blades:	DuPont Zytel Black ~ 40 % Glass
Nose Cone:	DuPont PETP (White) - SaliDrives GF PP - Glass Filled 30 % Polypropylene - Shafts
Spring and Cap Screws:	Stainless Steel 304 / 316
Blade Retaining Pins:	Titanium
Seals:	PETP Boss or Neoprene V Seals (Blades)
Shaft Diameters:	SAE 1/16 taper 7/8", 1", 1¼", 1½" ISO 1:10 taper 25 or 30 mm SaliDrives SAE 16/32 Spline on 28 mm
Nut Threads:	SAE - ¾" - 1" or 1¼" UNC for 1½" ISO - M16 or M20 x 1.5 mm SaliDrives M16 or M20 x 2.0 mm
Keyway:	SAE ¼" or 3/8" - parallel ISO 6 x 6 or 8 x 7 mm - parallel
Power Range:	> 55 hp @ > 1340 shaft rpm > ~15 hp @ < 1400 shaft rpm
Weight:	< 3.4 kg

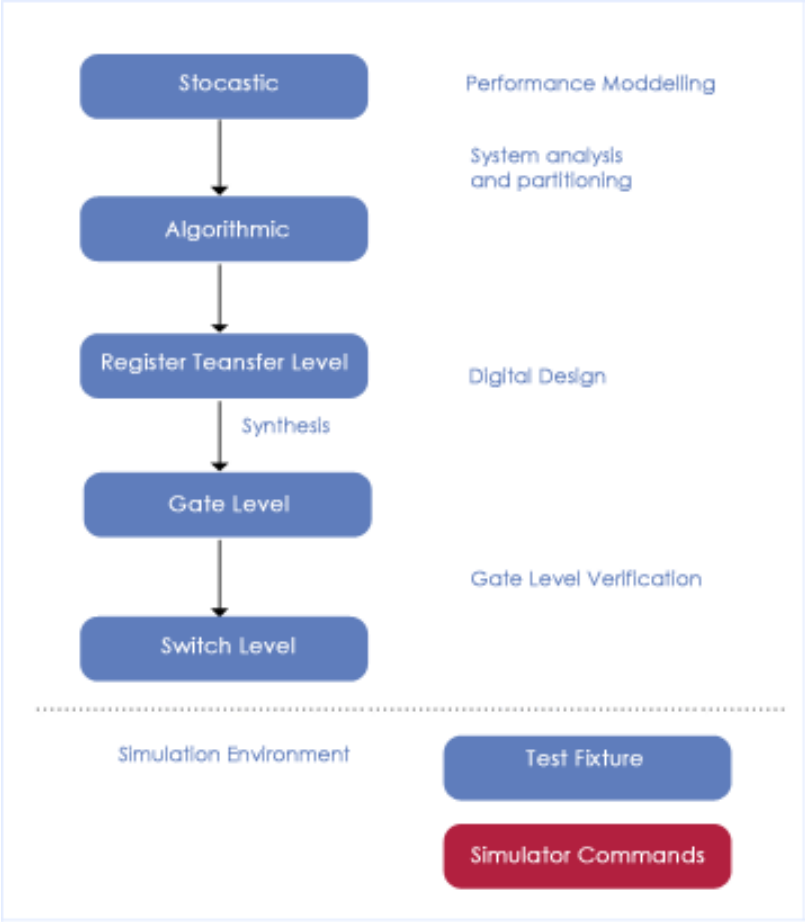
```
module mux(s,x0,x1,y);
    input s;
    input x0;
    input x1;
   output y;

   wire sel_0;
   wire sel_1;

   AND2 XLXI_1 (.I0(s),.I1(x1),.O(sel_1));
   AND2B1 XLXI_2 (.I0(s),.I1(x0),.O(sel_0));
   OR2 XLXI_3 (.I0(sel_1),.I1(sel_0),.O(y));
endmodule
```



Projeto com HDL - Verilog



Níveis de abstração em projetos com Verilog



Níveis de abstração em projetos com Verilog

No seu mais alto nível de abstração a linguagem Verilog contem **funções estocásticas** a fim de permitir a modelagem de desempenho do sistema.

Verilog permite também a **modelagem comportamental** do sistema e seus módulos, desta forma pode-se modelar a funcionalidade do sistema em um alto nível de abstração. Este é um recurso importante para a validação de conceitos e o particionamento do sistema.

Níveis de abstração em projetos com Verilog

Verilog permite a descrição a nivel RTL (Register Transfer Level), a qual é utilizada para permitir o projeto detalhado dos circuitos digitais que comporão o sistema. Ferramentas de síntese irão trasformar esta descrição em um arquivo de configuração e/ou de mapeamento a nível de portas lógicas.

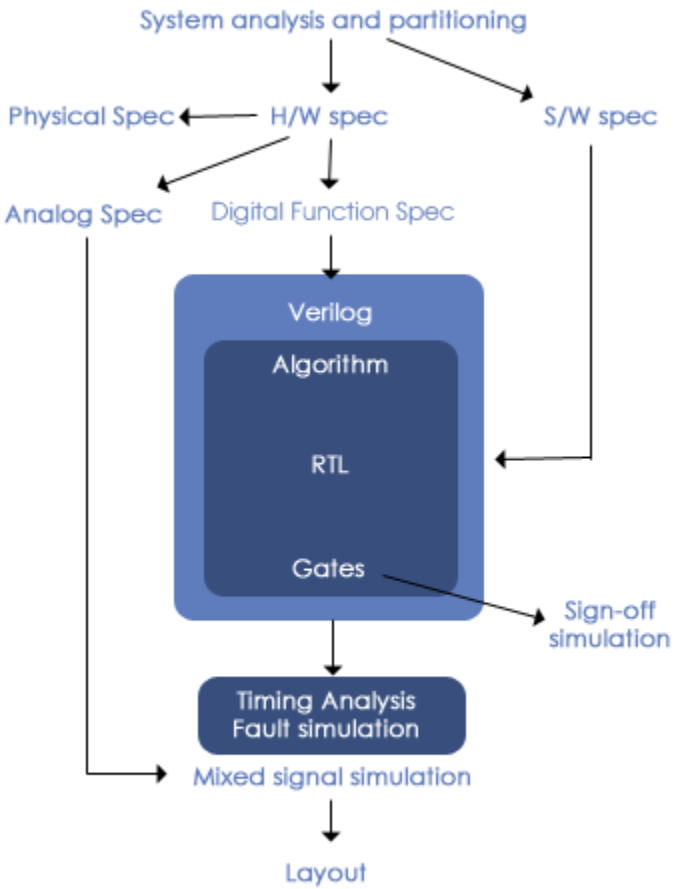
Verilog permite também uma descrição a nível de portas lógicas e de circuitos de chaveamento, a qual é utilizada para fins de verificação do projeto.

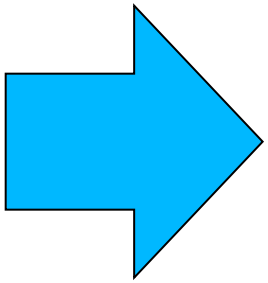
Níveis de abstração em projetos com Verilog

Verilog pode ser utilizado também para descrever ambientes de simulação, com vetores de teste, resultados esperados e circuitos de comparação e análise com os resultados obtidos.

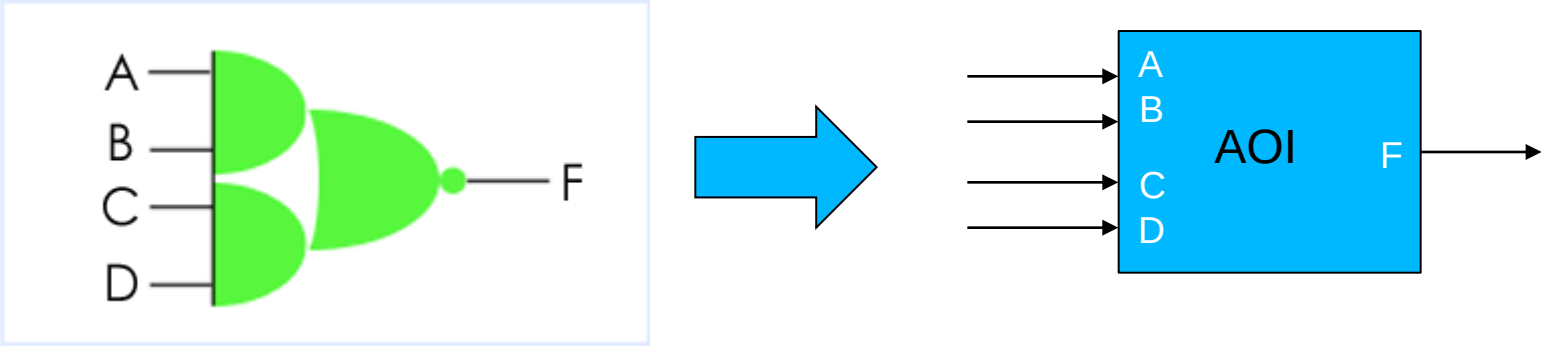
Algumas ferramentas de simulação permitem também que sejam introduzidos breakpoints e checkpoints a fim de facilitar o processo de verificação e debug dos módulos em desenvolvimento.

Design process





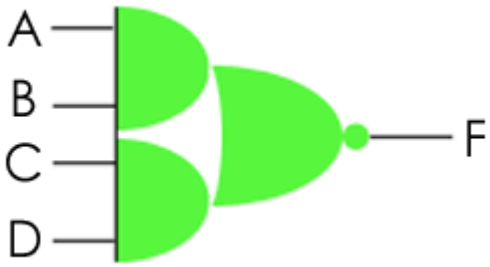
Simple Design




```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  ....
endmodule
// end of Verilog code
```

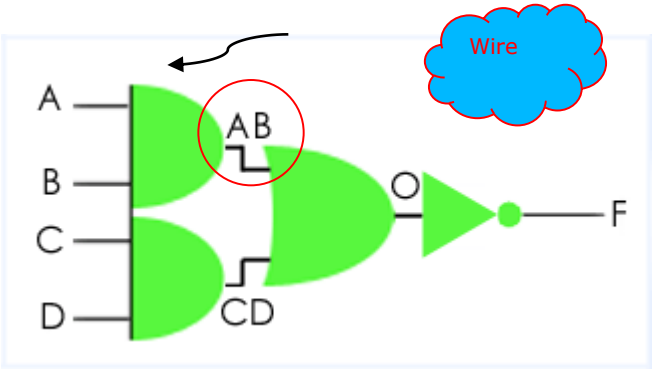
Simple Design

Module xxx (O_1 , O_2 , … I_1, I_2…);
    Input I_1, I_2, …., I_n;
    output O_1, O_2, …., O_n;
    assign O_1=….
    ….
    assign O_n=….
endmodule




```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
// end of Verilog code
```

Utilizando Wires



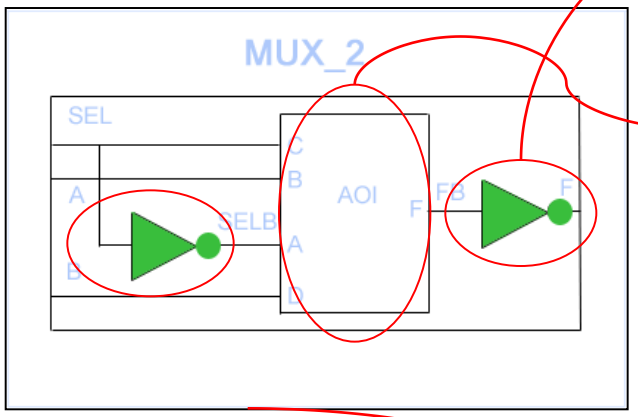

```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  wire F;  // the default
  wire AB, CD, O;  // necessary

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
endmodule
// end of Verilog code
```

Utilizando Wires

```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  /*  start of a block comment
  wire F;
  wire AB, CD, O;
  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
  end of a block comment */

  // Equivalent...
  wire AB = A & B;
  wire CD = C & D;
  wire O = AB | CD;
  wire F = ~O;
endmodule
// end of Verilog code
```

Comentário em Bloco

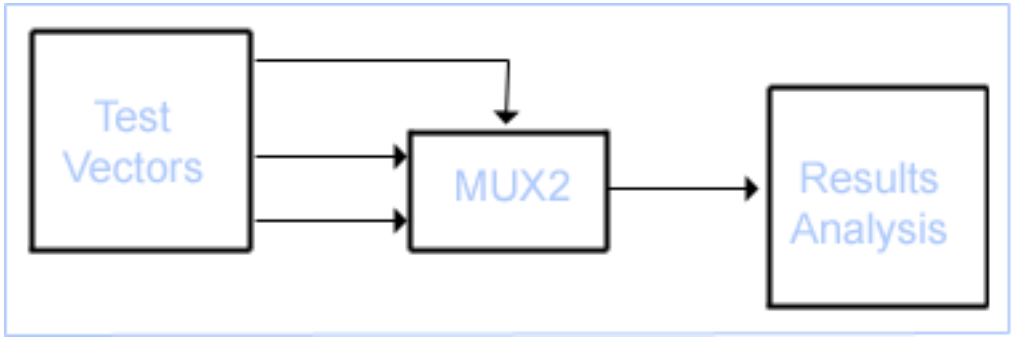
Projeto Hierarquico




```
// Verilog code for 2-input multiplexer
module INV (input A, output F);   // An inverter
  assign F = ~A;
endmodule

module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule

module MUX2 (input SEL, A, B, output F);   // 2:1
    multiplexer
  // wires SELB and FB are implicit
  // Module instances...
  INV G1 (SEL, SELB);
  AOI G2 (SELB, A, SEL, B, FB); // Position mapping
  INV G3 (.A(FB), .F(F));              // Named mapping
endmodule
// end of Verilog code
```

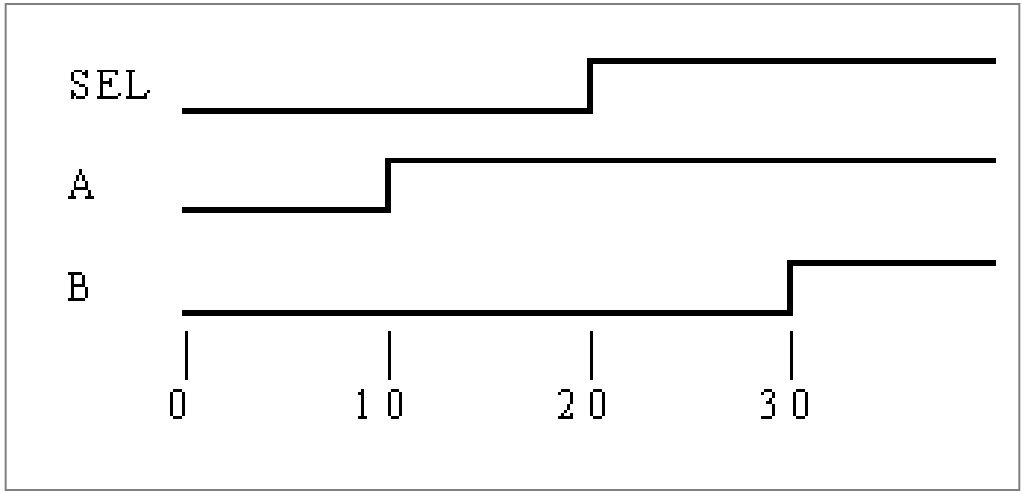
Testbench

```
module MUX2TEST;
   // No ports!
    ...
    initial // Stimulus
    ...
    MUX2 M (SEL, A, B, F);

    initial // Analysis
    ...
endmodule
```



Testbench - Stimulus

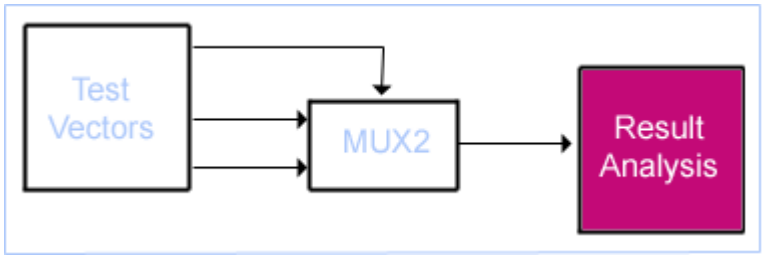
```
initial // Stimulus
begin
    SEL = 0; A = 0; B = 0;
    #10 A = 1;
    #10 SEL = 1;
    #10 B = 1;
end
```

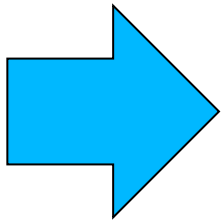


Testbench - **Analysis**

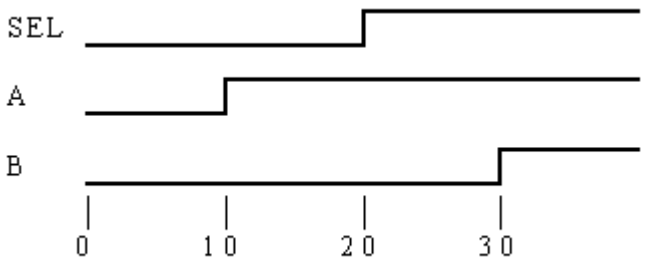
initial // Response

$monitor($time, , SEL, A, B, F);





```
0   0000
10 0101
20 1100
30 1111
```



Verilog RTL

```
module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
```

```
module MUX2 (input SEL, A, B, output F);
  input SEL, A, B;
  output F;
  INV G1 (SEL, SELB);
  AOI G2 (SELB, A, SEL, B, FB);
  INV G3 (.A(FB), .F(F));
endmodule
```

Bloco Always – Verilog Comportamental

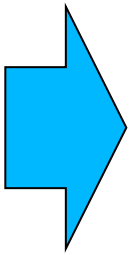
```
always @(sensitivity-list)
begin
  // statements
end
```

```
always @(sensitivity-list)
begin
  F = ~((a & b) | (c & d));
end
```

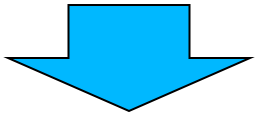
```
always @(a or b or c or d)
begin
  F = ~((a & b) | (c & d));
end
```

Módulo AOI utilizando Always

```
always @(sel)
begin
  selb = ~sel;
end

always @(a or sel or b or selb)
begin
  fb = ~((a & sel) | (b & selb));
end

always @(fb)
begin
  f = ~fb;
end
```



```
always @(sel or a or b)
begin
  selb = ~sel;
  fb = ~((a & sel) | (b & selb));
  f = ~fb;
end
```



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

Declaração de variáveis

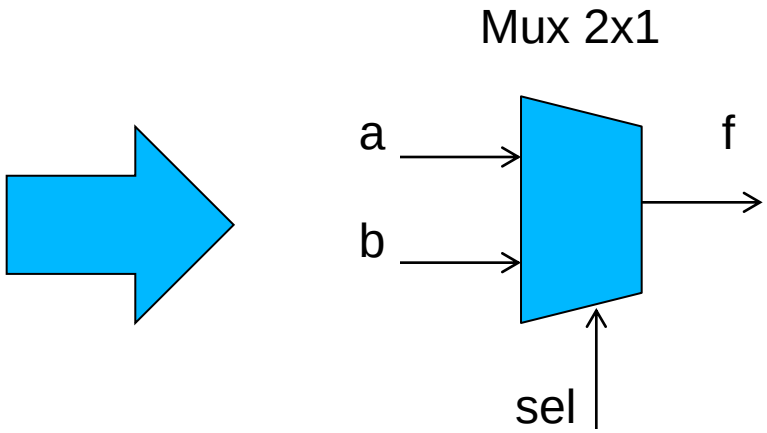
```
reg f;
always @(sel or a or b)
begin
  if (sel == 1)
    f = a;
  else
    f = b;
end
```

```
reg f; // must be declared before it is used in a statement
```

Lógica combinacional com always

Golden Rule 1: Para sintetizar lógica combinacional utilizando o always todos os sinais de entrada devem aparacer na lista de sensibilidade

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



Comando If

- O comando **if** é por **natureza sequencial**, ou seja, ele primeiro avalia a expressão de controle e em seguida executa o comando associado e, assim como na liguagem C, aceita apenas um comando aninhado em cada fluxo de execução.
- Sendo necessário aninhar mais de um comando deve se utilizar o comando de bloco **begin...end**
- Normalmente o comando if é sintetizado na forma de um **multiplexador**

```
always @(sel or a or b)
begin
  if (sel == 1)
    begin
      f = a;
     g = ~a;
    end
  else
    begin
      f = b;
      g = a & b;
    end
end
```

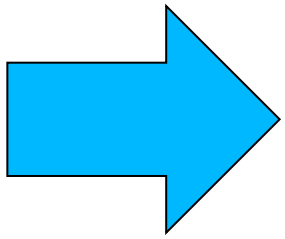
Comando if aninhado

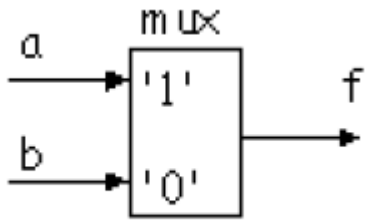
```
reg f, g;
always @(sel or sel_2 or a or b)
  if (sel == 1)
    begin
      f = a;
      if (sel_2 == 1)
        g = ~a;
      else
        g = ~b;
    end
  else
    begin
      f = b;
      if (sel_2 == 1)
        g = a & b;
      else
        g = a | b;
    end
```

Problemas com o comando if

```
reg sel, a, b;
```

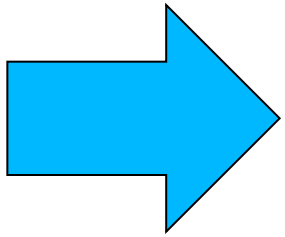
```
always @ (sel or a or b)
  begin : pure_if
    f = b;
    if (sel == 1)
      f = a;
  end
```

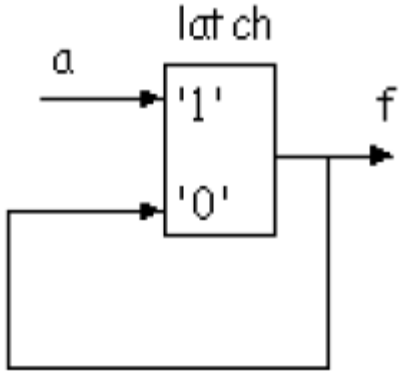





```
reg sel, a;
```

```
always @ (sel, a)
  begin : latching_if
    if (sel == 1)
      f = a;
  end
```





Problemas com o comando if

Golden Rule 2: Para sintetizar lógica combinacional utilizando o always todas as variáveis devem ser carregadadas em todas as condições

Resolvendo o problema com always combinacional

Se não for possível garantir que todas as variáveis sejam carregadas em todas as condições, carrege-as com valores default antes de executar o código condicional

```
always @ (sel or sel_2 or sel_3 or a or b)
  begin
    // default values assigned to f, g
    f = b;
    g = a & b;
    if (sel == 1)
      begin
        f = a;
        if (sel_2 == 1)
          g = ~ a;
       else
         begin
           g = ~ b;
           if (sel_3 == 1)
             g = a ^ b;
         end
      end
    else
      if (sel_2 == 1)
        g = a & b;
      else
        if (sel_3 == 1)
          g = ~(a & b);
  end
```

Comando Case

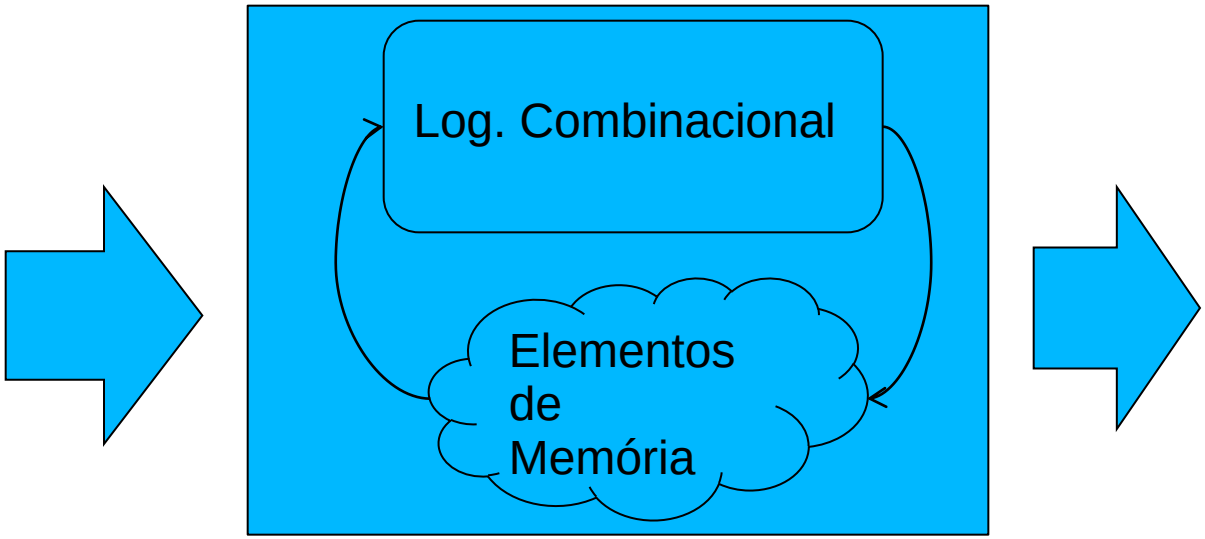
```
logic a,b,c;
…
case ({a, b, c})
        3'b000: f = 1'b0;
        3'b00l: f = 1'b1;
        3'b0l0: f = 1'b1;
        3'b011: f = 1'b1;
        3'b100: f = 1'b1;
        default: f = 1'b0;
endcase
```

```
logic [2:0] dado;
…
case (dado)
        3'b000: f = 1'b0;
        3'b00l: f = 1'b1;
        3'b0l0: f = 1'b1;
        3'b011: f = 1'b1;
        3'b100: f = 1'b1;
        default: f = 1'b0;
endcase
```

```
module SevSegCase(aIn, sOut);
  input [3:0]aIn;
  output [6:0]sOut;
  reg [6:0]sOut;
  always @(aIn)
    begin
      case (aIn)
        //                  abcdefg
        4'b0000:sOut = 7'b0000001; //0
        4'b0001:sOut = 7'b1001111; //1
        4'b0010:sOut = 7'b0010010; //2
        4'b0011:sOut = 7'b0000110; //3
        4'b0100:sOut = 7'b1001100; //4
        4'b0101:sOut = 7'b0100100; //5
        4'b0110:sOut = 7'b0100000; //6
        4'b0111:sOut = 7'b0001111; //7
        4'b1000:sOut = 7'b0000000; //8
        4'b1001:sOut = 7'b0001100; //9
        4'b1010:sOut = 7'b0001000; //A
        4'b1011:sOut = 7'b1000010; //B
        4'b1100:sOut = 7'b0000111; //C
        4'b1101:sOut = 7'b0000001; //D
        4'b1110:sOut = 7'b0110000; //E
        4'b1111:sOut = 7'b0000110; //F
      endcase
  end
endmodule
```

Projeto de módulos sequenciais

Módulos sequenciais diferem dos módulos combinacionais por disporem de elementos de memória que registram o estado do sistema de forma a alterar a sua resposta aos estímulos da entrada a partir do estado em que se encontra



Projeto de módulos sequenciais

Como elementos de memória normalmente utilizam-se Flip-Flops, os quias são implementados utilizando o comando always

- O comando always executa o que estiver em seu interior de maneira síncrona com os eventos monitorados em sua **lista de sensibilidade**
- Da mesma forma que os flip-flops ativam o armazenamento de informação sincronamente com o sinal de clock
- Enquanto o evento não ocorre a lógica interna do flip-flop não é ativada

Always - Lista de sensibilidade

```
always @(sensitivity-list)
begin
  // statements
end
```

sensitivity-list

```
always @(a, b, c, d)
always @(a or  b or c or d)
always @(*)
always @*
always @(posedge a or posedge b)
always @(negedge a or posedge b)
always @(negedge a)
```

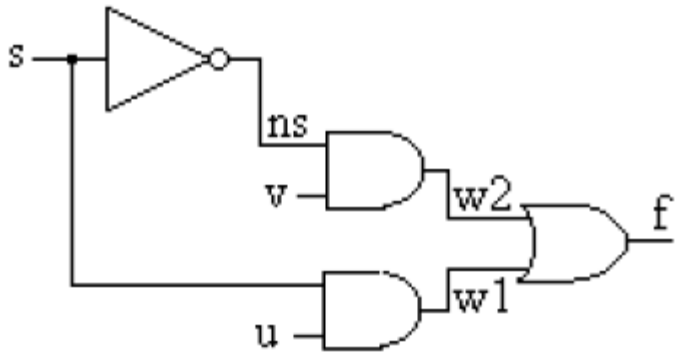
Always - observações

- Tipos de transições
  - posedge (transição de subida)
  - negedge (transição de descida)
  - sem polaridade (qualquer transição)

- Não é permitido misturar transições com e sem polaridade na lista de sensibilidade
  - ✓ always @(a or b or c or d)
  - ✓ always @(posedge a or negedge b)
  - X always @(a or posedge b)

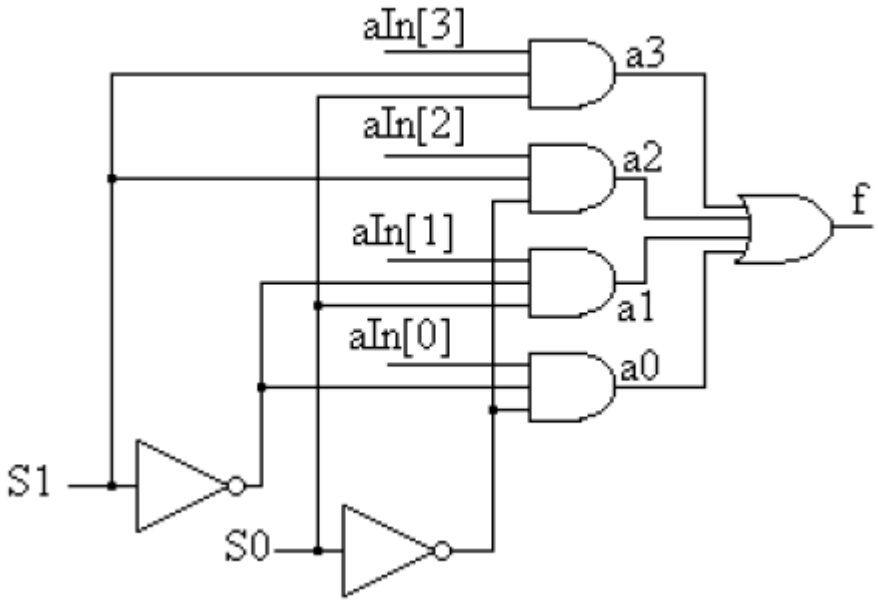
Exemplos de projetos
Lógica Combinacional - RTL

```
module AndOr(f, u, v, s);
  input s, u, v;
  output f;
  wire w1, w2;
  wire ns;
  and A1(w1, u, s);
  not N1(ns, s);
  and A2(w2, v, ns);
  or O1(f, w1, w2);
endmodule
```



Exemplos de projetos
Lógica Combinacional – RTL Hierárquico

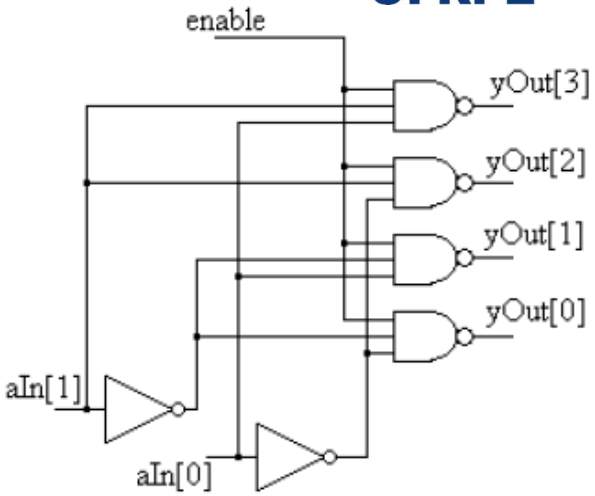
```
module Mux4To1(f, s0, s1, aIn);
  output f;
  input s0, s1;
  input [3:0]aIn;
  wire ns0, ns1;
  wire a0, a1, a2, a3;
  //
  not nots0(ns0, s0);
  not nots1(ns1, s1);
  //
  and and0(a0, ns0, ns1, aIn[0]);
  and and1(a1, s0, ns1, aIn[1]);
  and and2(a2, ns0, s1, aIn[2]);
  and and3(a3, s0, s1, aIn[3]);
  //
  or or1(f, a0, a1, a2, a3);
endmodule
```



Exemplos de projetos
Lógica Combinacional – Comportamental

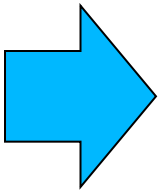
UFRPE

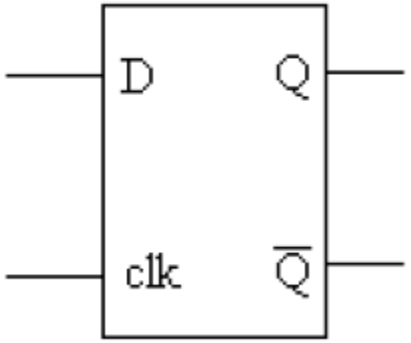
```
module Decode2To4(aIn, yOut, enable);
  input [1:0]aIn;
  input enable;
  output [3:0]yOut;
  reg [3:0] yOut;
  always@(aIn or enable)
    begin
      if(enable == 1)
        begin
          if(~aIn[1] && ~aIn[0]) yOut = 4'b0111;
          if(~aIn[1] && aIn[0]) yOut = 4'b1011;
          if(aIn[1] && ~aIn[0]) yOut = 4'b1101;
          if(aIn[1] && aIn[0]) yOut = 4'b1110;
        end
      else
        yOut = 4'b1111;
    end
endmodule
```



Exemplos de projetos
Lógica Sequencial– Comportamental

```
module classicD(D, clk, Q, Qn);
  input D, clk;
  output Q, Qn;
  reg Q, Qn;
  always@(D or clk)
    if(clk)
      begin
        Q <= D;
        Qn <= ~D;
      end
endmodule
```



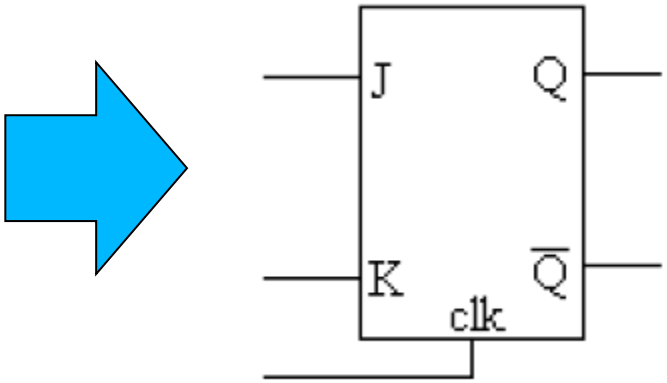


Exemplos de projetos
Lógica Sequencial– Comportamental

```
module DFFAsyncClr(D, clk, resetn, Q, presetn);
  input D, clk, resetn, presetn;
  output Q;
  reg Q;
  always@(posedge clk or negedge resetn or negedge presetn)
    if(!resetn)
      Q <= 0;
    else if(!presetn)
      Q <= 1;
    else
      Q <= D;
endmodule
```

Exemplos de projetos
Lógica Sequencial– Comportamental

```
module jkff(J, K, clk, Q);
  input J, K, clk;
  output Q;
  reg Q;
  reg Qm;
  always @(posedge clk)
    if(J == 1 && K == 0)
      Qm <= 1;
    else if(J == 0 && K == 1)
      Qm <= 0;
    else if(J == 1 && K == 1)
      Qm <= ~Qm;
  //
  always @(negedge clk)
    Q <= Qm;
endmodule
```



Lógicas sequencial x combinacional

- Na lógica sequencial as atualizações nas saidas ocorrem sincronamente com as transições indicadas para os elementos da lista de sensibilidade.
- Na lógica combinacional as saidas estão em constante atualização. Qualquer alteração no estado de qualquer das entradas força a atualização das saidas.

Review

Combinacional RTL

```
// continuous assignments
assign selb = ~sel;
assign fb = ~((a & sel) | (b & selb));
assign f = ~fb
```

Hierárquico

```
// a hierarchy of designs
INV G1 (SEL, SELB);
AOI G2 (SELB, A, SEL, B, FB);
INV G3 (.A(FB), .F(F));
```

Combinacional Comportamental

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

Combinacional Sequancial

```
// always block
always @(posedge clk)
Begin
   q=d;
end
```

Exercícios

Implemente e verifique a funcionalidade dos seguintes módulos:

a) Porta and com 5 entradas
b) Multiplexador 4 x 4
c) Contador módulo 4
d) Somador/subtrator completo de 4 bits no padrão de magnitude e sinal
e) Conversor BCD – 7 seguimentos
f) Flip-Flops JK, D e T

Comando Case

```
logic a,b,c;
…
case ({a, b, c})
      3'b000: f = 1'b0;
      3'b00l: f = 1'b1;
      3'b0l0: f = 1'b1;
      3'b011: f = 1'b1;
      3'b100: f = 1'b1;
      default: f = 1'b0;
endcase
```

```
logic [2:0] dado;
…
case (dado)
      3'b000: f = 1'b0;
      3'b00l: f = 1'b1;
      3'b0l0: f = 1'b1;
      3'b011: f = 1'b1;
      3'b100: f = 1'b1;
      default: f = 1'b0;
endcase
```

Projeto 1

Implemente uma calculadora de 4 bits com as seguintes operações:

- SOMA
- SUBTRAÇÃO
- MULTIPLICAÇÃO
- DIVISÃO

Projeto 2

Implemente uma máquina de vender refrigerantes com as seguintes funcionalidades:

- Aceita moedas de R$ 0,50 e R$ 1,0
- Vende refrigerantes de R$ 1.50 , R$ 2.50 e R$ 3.00
- Pode acumular até 6,0 em créditos
- Fornece quantos refrigerantes os créditos acumulados permitirem
- Fornece troco (tem uma tecla para pedir o troco)
- Possui um display para mostrar o valor acumulado e o troco
- Tem um estoque para até 3 latas de cada refrigerante
- Sinaliza quando tem refrigerante no estoque (LED)

Projeto 2