SystemVerilog para Projetos

Curso Presencial IFPB/UFAL

Marcos Morais – Professor DEE
Projetos de Excelência em Microeletrônica – PEM
Centro de Engenharia Elétrica e Informática
Universidade Federal de Campina Grande

Sumário

- Níveis de abstração;
- Projetos bottom-up e top-down;
- Diagrama de blocos;
- Separação entre caminho de dados (datapath) e controle;
- Linguagens de descrição de hardware;
- Fluxos de projeto em microeletrônica e FPGA;
- Subconjunto sintetizável;
- Descrição comportamental e nível RTL;
- Módulos;
- Tipos de dados e valores literais;
- Operadores e expressões;
- Sentenças procedurais e de controle;
- Atribuição de variáveis;
- Procedimentos e processos;
- Tarefas e funções;
- Interfaces;
- Simulação e síntese lógica;
- Aplicações e exemplos;
- Laboratório - Projeto do flux-capacitor (vai e vem de LEDs);
- Laboratório - Projeto de máximo divisor comum

Sistemas Digitais Complexos com SystemVerilog HDL

Nível COMPORTAMENTAL e RTL

Projetos de Excelência em Microeletrônica – PEM
Centro de Engenharia Elétrica e Informática
Universidade Federal de Campina Grande

Descrevendo hardware de forma soft

"Hardware is called hardware because it's hard"

The Art of Electronics, Horowitz and Hill, 1980

Nível RTL

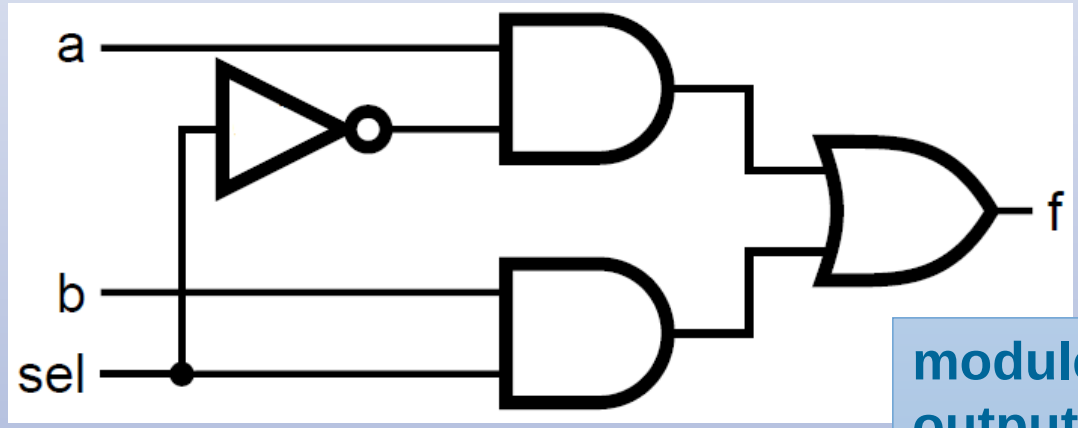
- A descrição em **RTL** (*Register Transfer Level*, nível de transferência de registradores) indica **como** e **quando** o conteúdo de registradores são **transformados** e armazenados em outros registradores (ou os mesmos de origem).
- Utilizado para descrever lógica seqüencial e lógica combinacional
- Na lógica combinacional as variáveis (logic) são **temporárias** e **desaparecem** no resultado final

Lógica Combinacional em RTL

Sentenças procedurais e lógica

Revisando descrição com assign

- Descreva o circuito abaixo em verilog usando assign:



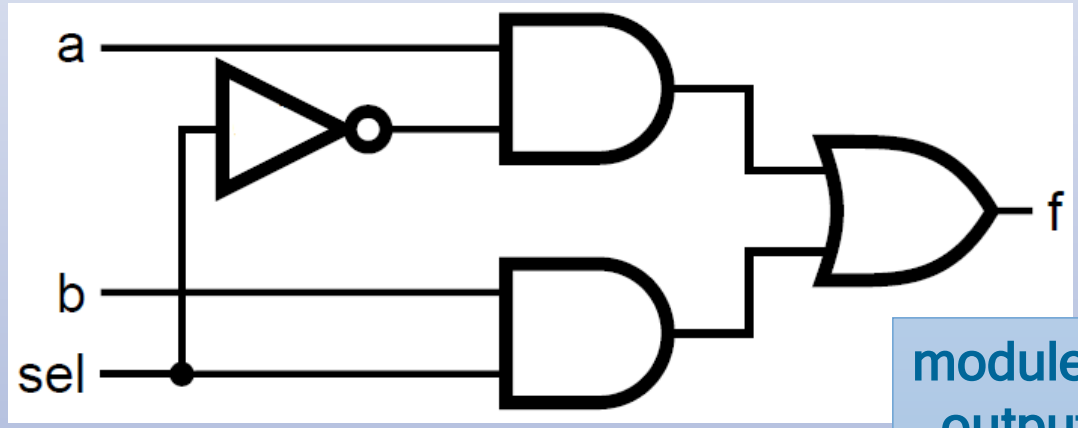

```
module mux(f, a, b, sel);
output logic f;
input logic a, b, sel;

assign f = ~sel ? a : b;

endmodule
```

Blocos always

- Para síntese, descrevem partes de hardware que "existem", "estão disponíveis":



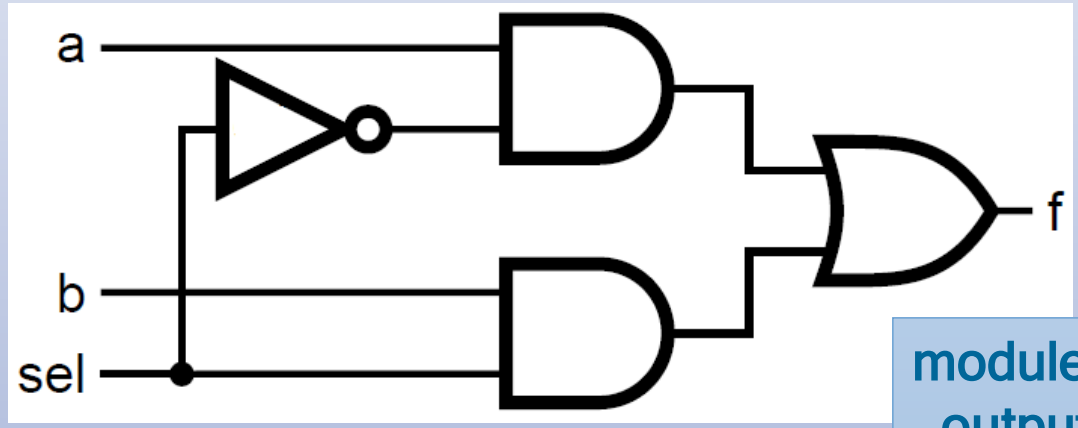

```
module mux2to1 (
  output logic f;
  input logic a, b, sel;

always @(a or b or sel)
   f= ~sel ? a : b;

endmodule
```

Blocos always

- Para síntese, descrevem partes de hardware que "existem", "estão disponíveis":



Lista de sensibilidade contém sinais cuja mudança faz o bloco funcionar

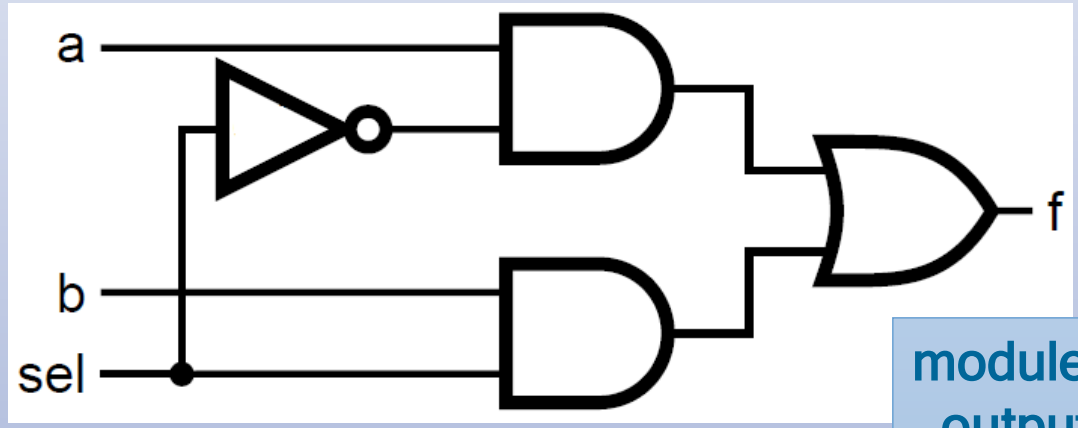
```
module mux2to1 (
  output logic f;
  input logic a, b, sel;

always @(a or b or sel)
   f= ~sel ? a : b;

endmodule
```

Blocos always

- Para síntese, descrevem partes de hardware que "existem", "estão disponíveis":



Lista de sensibilidade contém sinais cuja mudança faz o bloco funcionar

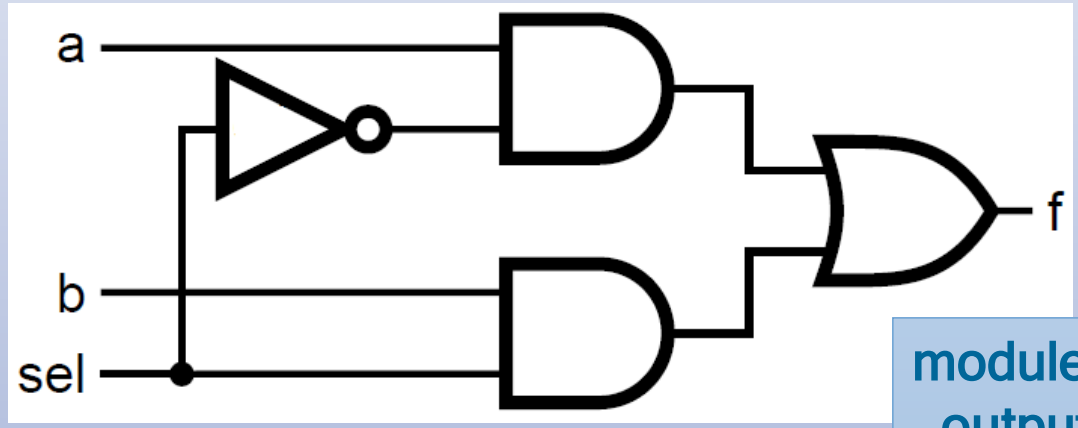
```
module mux2to1 (
  output logic f;
  input logic a, b, sel;

always @(*)
   f= ~sel ? a : b;

endmodule
```

Blocos always

- Para síntese, descrevem partes de hardware que "existem", "estão disponíveis":



SystemVerilog facilita especificar que se quer fazer circuito combinacional

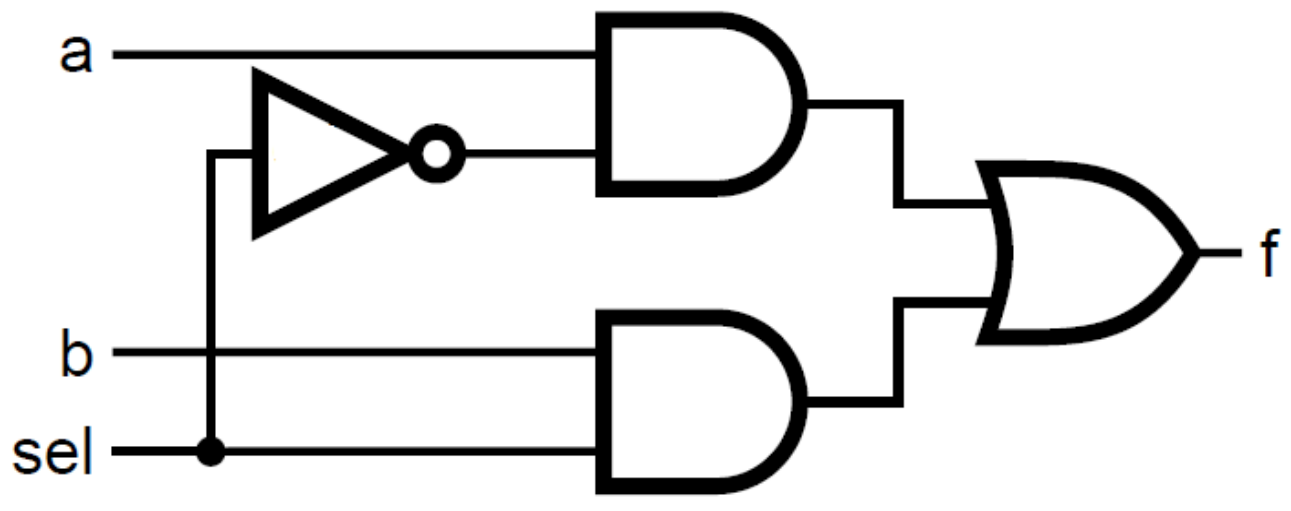
```
module mux2to1 (
  output logic f;
  input logic a, b, sel;

always_comb
   f= ~sel ? a : b;

endmodule
```

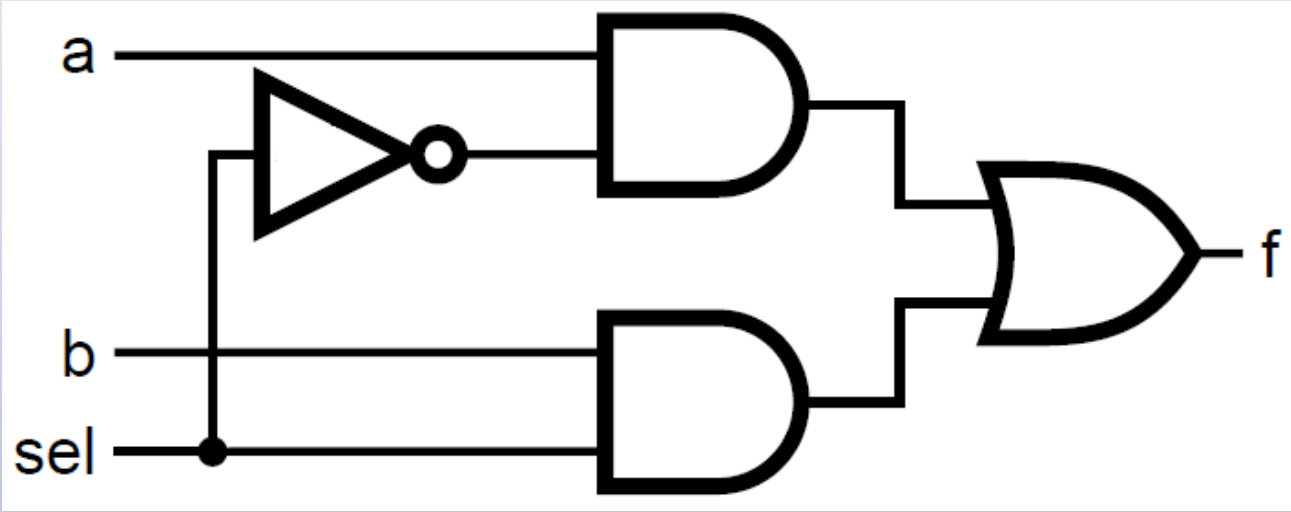
Sentenças procedurais

- Podemos utilizar sentenças procedurais com estilo de linguagem de alto nível (HLL) para implementar lógica. Considere o exemplo:



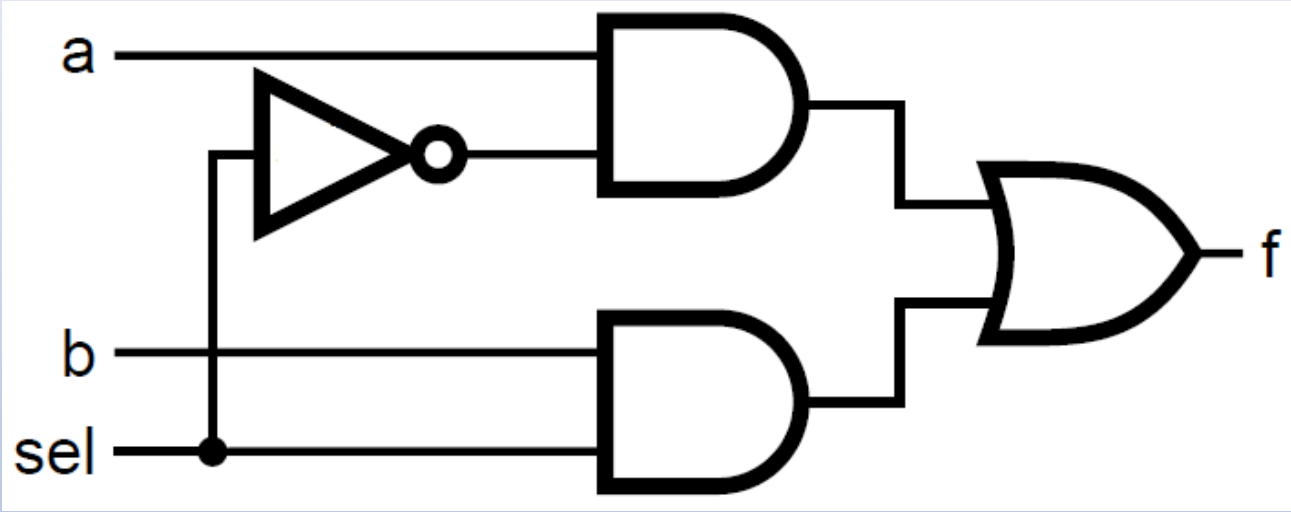
- Descreva esta função em palavras (comandos).
- Dica: se ... senão

Se... senão



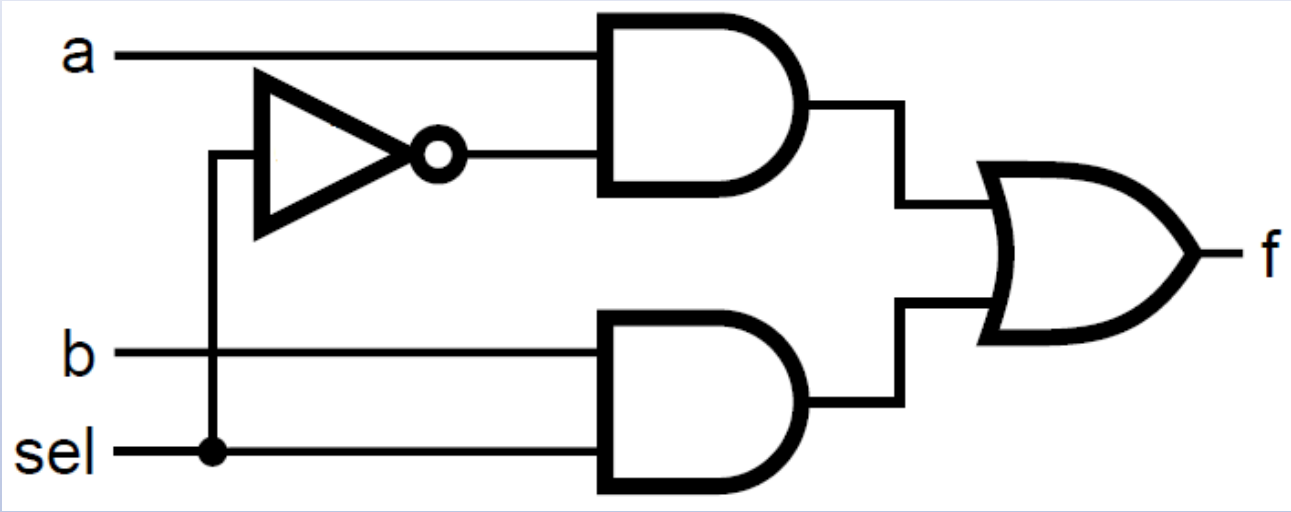

```
se (sel for 0)
  f recebe a;
senão
  f recebe b;
```

If...




```
if (sel == 0)
   f = a;
else
  f = b;
```

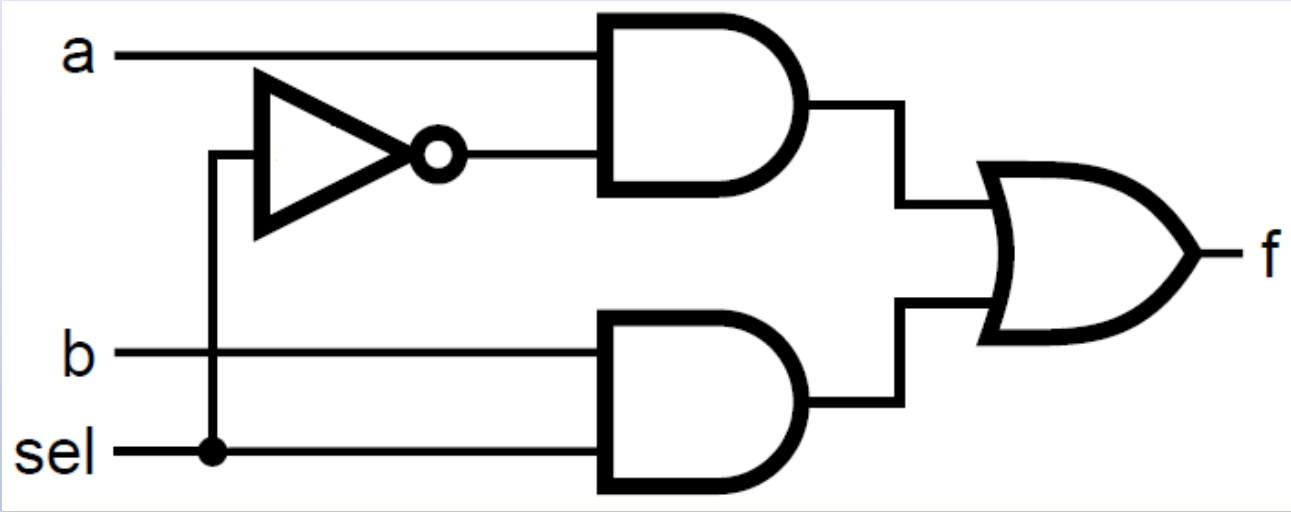
If...




```
if (sel == 0)
   f = a;
else
  f = b;
```

Constantes em SystemVerilog por default estão em decimal com 32 bits

If...




```
if (sel == 1'b0)
   f = a;
else
  f = b;
```

Multiplexador com always_comb

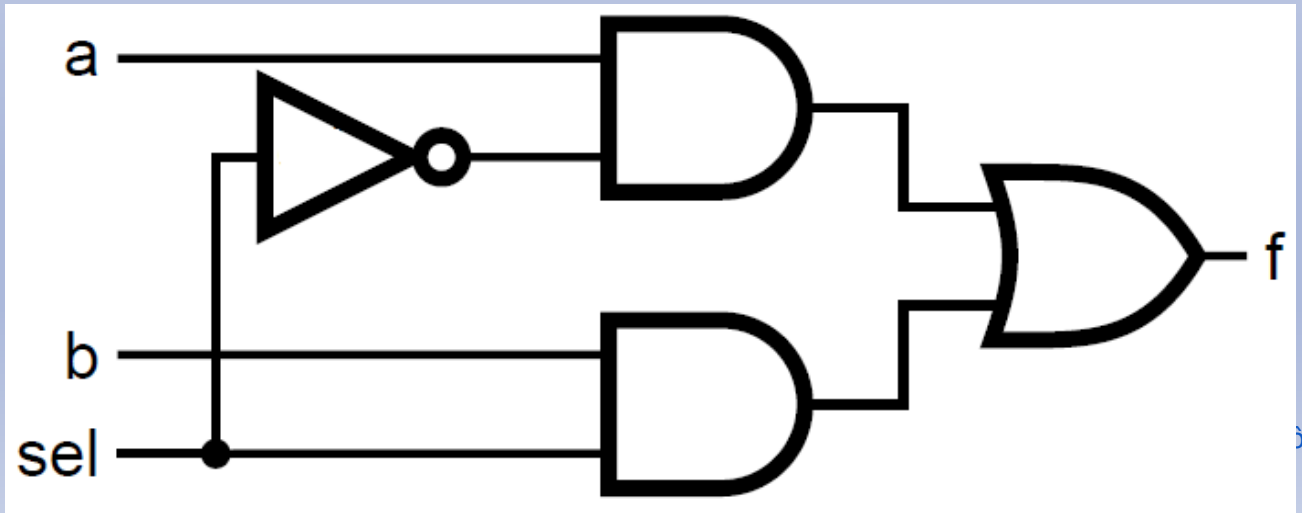
```
module mux(
  input logic a, b,sel,
  output logic f );

always_comb
 if (~sel)
  f = a;
else
  f = b;
endmodule
```

Módulos podem conter um ou mais blocos always

Circuitos combinacionais são ativados se qualquer dos sinais usados mudar

Testes de valores lógicos ou binários



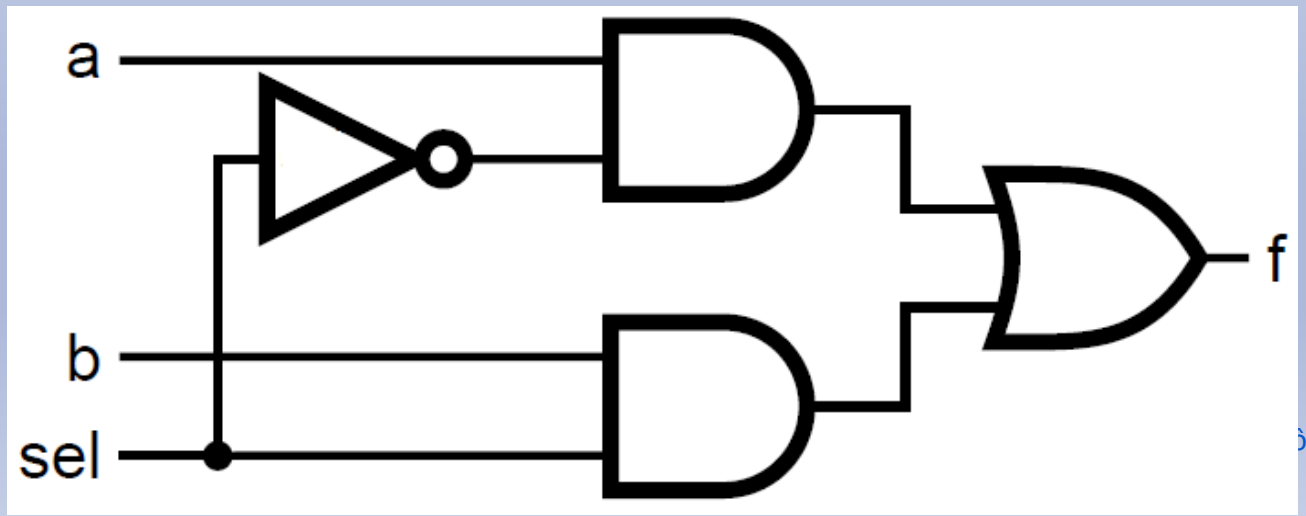
Multiplexador com always_comb

```
module mux(
  input logic a, b,sel,
  output logic f );

always_comb
 if (~sel)
  f = a;
else
  f = b;
endmodule
```

Um **`logic`** pode se comportar como memória: mantém o seu valor até que seja imperativamente atribuído

O corpo de um bloco always contém "código" imperativo tradicional



Dois tipos principais de dados: Nets

- Os ***nets*** (fiação) representam as conexões entre as coisas
- Não retêm os seus valores
- Obtêm os seus valores de uma saída como uma porta ou outro módulo
- Não devem ser atribuídos em um bloco *initial* ou *always*
- Em SystemVerilog, isso ficou mais flexível

Dois tipos principais de dados: logic

- **logic**s representam armazenamento de dados
- Comportam-se como uma memória
- Retém o seu valor até que seja explicitamente atribuído em um bloco **always** ou **initial**
- Pode ser usado para modelar latches, flip-flops, etc., mas não correspondem exatamente a eles

Nets e Registradores

Fios e registradores podem ser bits, vetores ou arrays

```
logic a; // fio simples
logic [-1:4] vec; // registrador de 6 bits
logic [31:0] dcache[0:63]; // Uma memória de 32 bits
```

Atribuição Procedural

- No interior de um bloco *initial* ou *always*

```
sum = a + b + cin;
```

- Tal como C: O lado direito é avaliado e atribuído ao lado esquerdo antes que a próxima sentença seja avaliada
- A sentença do lado direito pode conter fios (*wires*), *logic,* constantes, funções lógicas, etc
- A do lado esquerdo tem que ser um *logic*
- (apenas primitivas ou atribuições contínuas podem atribuir valores a fios)

Blocos always

```
always @(lista_sensibilidade)
  begin
       /* Este bloco de sentenças é ativado sempre que qualquer
       das variaveis lista_sensibilidade muda de valor */
  end
```

- As sentenças entre o **begin** e o **end** do bloco **always** é avaliado sequencialmente sempre que qualquer das variáveis **logic** mencionadas nos parênteses, conhecida como lista de sensibilidade, mudar seus valores.

Projeto assíncrono

```
always_comb
begin
  // sentenças aqui
end
```

```
always_latch
begin
  // sentenças aqui
end
```

- Usado para definir lógica combinacional e latches, como vimos.

Projeto assíncrono

```
always_comb
begin
  // sentenças aqui
end
```

~~always_latch~~
~~begin~~
~~// sentenças aqui~~
~~end~~

- Usado para definir lógica combinacional e latches, como vimos.

Projeto síncrono

Informamos o sinal de clock juntamente com borda.

Borda de subida

```
always_ff @(posedge clk)
```

Borda de descida

```
always_ff @(negedge clk)
```

Variáveis logic

- Apenas variáveis (logic) podem ser manipuladas em blocos always.
- Entradas, saídas e fios são nets. Estes não armazenam valores. Para armazenar valores é preciso defini-los como logic.

```
output logic saida_1;
```

if...else

```
always_comb
  begin
    if (condicao_1) acao_1;
    else if (condicao_2) acao_2;
    else acao_3;
  end
```

```
if (condicao)
begin
  logic_1=1'b1;
  logic_2=1'b1;
  logic_3=1'b1;
end
```

if...else

```
always_comb
  if (condicao_1) acao_1;
  else if (condicao_2) acao_2;
  else acao_3;
```

```
if (condicao)
begin
  logic_1=1'b1;
  logic_2=1'b1;
  logic_3=1'b1;
end
```

Case

```
always_comb
case(net)
  valor_1: acao_1;
  valor_2: acao_2;
// etc
endcase
```

```
case(net)
  valor_1:
    begin
      acao_1;
      acao_2;
      \\ etc
    end
  valor_2:
    begin
      acao_1;
      acao_2;
      \\ etc
    end
endcase
```

Exemplo: decodificador 7 segmentos

```
module decod7seg (
  input logic [3:0] entrada,
  output logic [0:6] saida);

always_comb @ (entrada)
begin
  case (entrada)
    4'd0: saida= 7'h7E;
    4'd1: saida= 7'h30;
        .
    4'd9: saida = 7'b1111011 ;

   endcase
endmodule
```

Exemplo: decodificador 7 segmentos

```
module decod7seg (
  input logic [3:0] entrada,
  output logic [0:6] saida);

always_comb @ (entrada)
begin
  case (entrada)
    4'd0: saida= 7'h7E;
    4'd1: saida= 7'h30;
        .
    4'd9: saida = 7'b1111011 ;
    default: saida = 7b‘0000001;
   endcase
endmodule
```

Exemplo: decodificador 7 segmentos

```
module decod7seg (
  input logic [3:0] entrada,
  output logic [0:6] saida);

always_comb
begin
  case (entrada)
    4'd0: saida= 7'h7E;
    4'd1: saida= 7'h30;
        .
    4'd9: saida = 7'b1111011;
    default: saida = 7b‘0000001;
   endcase
endmodule
```

Exemplos com sentenças imperativas

```
if (select == 1)
  y = a;
else
  y = b;

case (op)
  2'b00: y = a + b;
  2'b01: y = a - b;
  2'b10: y = a ^ b;
  default: y = 'hxxxx;
endcase
```

Operadores

Operadores aritméticos	
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo

Operadores booleanos	
&&	AND lógico
!	Not lógico
||	OR lógico
&	AND bit-a-bit
~	NOT bit-a-bit
|	OR bit-a-bit
^	XOR bit-a-bit
& (unário)	Redução AND
~(unário)	Redução NOT
|(unário)	Redução OR
^(unário)	Redução XOR

Operadores de redução

```
A = | ent_1; /* ALGUM bit 1? E.g. |0110=1 */
B = & ent_2; /* TODOS bits 1? E.g. &1111=1 */
C =~| ent_3; /* TODOS bits 0? E.g. ~|0000=1 */
```

Operadores relacionais	
<	Menor que
>	Maior que
<=	Menor que ou igual a
>=	Maior que ou igual a
==	Igual a
!=	Não igual a

Com ou sem sinal

- Verilog trata os valores numéricos como strings de bits sem sinal. O tratamento do sinal deve ser feito com lógica auxiliar.

Exemplo: contador

```
module (
  input direction, reset, clock,
  output logic [7:0] count_value
);

always @(posedge clock or posedge reset)
begin if (reset)
  count_value = 8'd0;
else
  count_value = count_value + 1'd1;
endmodule

// acrescente código para direcão e reset síncronos
```

Laços

- Laço for: sintaxe idêntica a C

```
for (atrib inicial; expressao; atrib. passo)
begin
  sentencas
end
```

- O número de vezes que o for executa tem que ser fixo para poder ser sintetizado.
- Não faz contador – mas sim, circuito combinacional

Exemplo:
soma os números de 0 a 254

```
module soma(input clock, output logic [15:0] op1);
logic [15:0] storage;
logic [7:0] i;

always_ff @(posedge clock)
begin
  for (i = 8'b00; i < 8'd255; i = i+8'd1)
    begin
      storage=storage+i;
    end
    op1=storage;
end

endmodule
```

Inferência de latches

```
// Código ruim– infere um latch!
always @ (b or c)
begin
  if (b)
    begin
     a = c;
  end
end
```

- Ocorre quando não são explicitados todos os valores que podem ser atribuídos a uma variável.
- Neste caso, o valor de **a** é retido. **a** só muda quando b é colocado em **1**. Conseqüentemente temos um latch.

Evitando latches

```
// Código bom! Dá o comportamento desejado.
always @ (b or c)
begin
a = 1'b0;
if (b)
  begin
    a = c;
  end
end
```

- Esta descrição gera o resultado correto e demonstra um mecanismo de evitar *latches*.
- **a** recebe sempre zero e somente se **b** e **c** forem ambos **1** que a recebe **1**.
- Outra opção é usar um **else**.

Evitando latches

```
// Código melhor! Informa o comportamento desejado.
always_comb
begin
a = 1'b0;
if (b)
  begin
    a = c;
  end
end
```

- Como seria com **else?**.
- Qual a função lógica?

O que vimos hoje

- Projetos digitais modernos (complexos) utilizam linguagens de descrição de hardware
- É possível expressar um sistema digital em diversos níveis de abstração
- O nível estrutural é utilizado para definir hierarquia e conectividade
- SystemVerilog é uma evolução de Verilog

Contato

Marcos Morais
Professor DEE
morais@dee.ufcg.edu.br