INTRODUÇÃO À LINGUAGEM VERILOG

Objetivos

- Apresentar a descrição da linguagem Verilog;
- Apresentar as estruturas básicas de controle em Verilog;
- Apresentar a forma de codificação em linguagem Verilog;
- Apresentar padrões de mapeamento para a linguagem Verilog.

Histórico

- 1984 – primeira versão desenvolvida pela Gateway Design Automation Inc. a partir de HiLo;
- 1985 – primeira versão do simulador Verilog;
- 1987 – expansão substancial da linguagem;
- 1990 – é incorporada pela Cadence Design System;
- 1991 – torna-se uma linguagem aberta;
- 1995 – padronização pelo IEEE 1364-1985;
- 2000 – novo padrão IEEE 1364-2000;
- 2005 – novo padrão IEEE 1364-2005.

Descrição da linguagem

- Alfabeto

  Um programa em Verilog poderá conter os seguintes caracteres:

  - as vinte e seis (26) letras do alfabeto inglês:
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    a b c d e f g h i j k l m n o p q r s t u v w x y z

  - os dez (10) algarismos:
    0 1 2 3 4 5 6 7 8 9

  - os símbolos:

<	menor	( )	parênteses
>	maior	[ ]	colchete
.	ponto	{ }	chaves
,	vírgula	+	soma
:	dois pontos	-	subtração
;	ponto-e-vírgula	*	asterisco
=	igualdade	/	barra
!	exclamação	#	sustenido
?	interrogação	“	aspas
&	ampersete ("e" comercial)	‘	apóstrofo
^	circunflexo	%	porcento
|	barra em pé	~	til

- Pontuação
  - Ponto-e-vírgula separa comandos, a menos que outro separador seja empregado;
  - Em alguns casos de operadores, convém o uso de espaços em branco antes, e depois.

- Observação:
  Em Verilog utilizam-se, ***obrigatoriamente***, as letras minúsculas para os comandos próprios da linguagem.

Representações de dados

- Constantes
  - Constante inteira

    Valores inteiros podem ser expressos nas seguintes bases de numeração:
    - binário (**b** ou **B**)
    - octal (**o** ou **O**)
    - decimal (**d** ou **D**)
    - hexadecimal (**h** ou **H**)

    Formato:
    <sinal><tamanho>'<base><valor> - descrição completa
    <sinal><base><valor> - padrão dependente da máquina (usual 32 bits)
    <sinal><valor> - valor decimal (padrão)

    Observações:
    - O sinal (_) pode ser usado no meio do valor para melhorar a legibilidade.
    - O indicador especial **x** representa um valor indeterminado.
    - O indicador especial **z** (ou **?**)representa um valor com alta impedância.

    Exemplos:
    1'b0, 1'B1, 1'o0, 1'O1, 0, 1, 1'h0, 1'H1, 'h0, 'h1
    8'b1010_0011, 8'hA1, -8'd3

    4'b11 - equivalente a 0011
    -4'b11 - em complemento de dois equivale a 1101
    4'b11x0 - segundo bit menos significativo indeterminado
    4'b011z - último bit menos significativo em alta impedância
    10'bz - decimal de 10 bits em alta impedância
    10'b? - decimal de 10 bits em alta impedância (*don't care*)
    8'h4x - hexadecimal com os 4 bits menos significativos indeterminados

  - Constate real

    Valores reais podem ser expressos em decimal ou em notação científica.

    Observações:
    - A vírgula decimal é representada por ponto decimal.
    - Deve haver pelo menos um dígito antes e depois do ponto decimal.

    Exemplos:
    10.465 -5.61 0.0 0.731 0.37e2 -3.oE-1

  - Constante literal

    Exemplos:
    Cadeia: "BRANCO", "CEU AZUL"

    Observações:
    O tamanho da cadeia é limitado e não deve ultrapassar uma linha.

- Caracteres predefinidos:

\n	passa para a próxima linha
\t	passa para a próxima coluna de tabulação (9,17, ...)
\\	barra invertida
\"	apóstrofo
%%	mostrar um %

- Portas lógicas

  Valores definidos: **buf**, **not**, **and**, **or**, **nand**, **nor**, **xor**, **xnor**

  Observações:
  - As portas **buf** e **not** têm uma entrada e podem ter duas ou mais saídas:
    buf B (saída1, saída2, entrada)
    not N1 (saída, entrada)

  - As portas **and** e **or** podem ter duas ou mais entradas:
    and A1 (saída, entrada1, entrada2)
    or O1 (saída, entrada1, entrada2, entrada3, entrada4)

- Tipos de dados

  - Redes (*net*)
    Redes representam saídas contínuas em relação às entradas.
    Se as entradas mudarem seus valores, as saídas serão automaticamente alteradas.

    Valores definidos: **wire**, **supply0**, **supply1, tri, tri0, tri1, triand, trior, trireg, wand, wor**
    Valor padrão: **z**
    Tamanho padrão: 1 bit

    Exemplo:
    ```
    reg  A;
    wire B;
    A = 1;
    not N1 (B, A);
    ```

  - Registradores (*reg*)
    Registradores representam dados cujos valores devem ser atribuídos explicitamente.
    Os registradores poderão representar valores negativos em complemento de dois.

    Valor definido: **reg**
    Valor padrão: **z**
    Tamanho padrão: 1 bit

    Exemplo:
    ```
    reg  A;
    wire B;
    A = 1;
    not N1 (B, A);
    ```

    Observação:
    Registradores (**reg**) guardam valor. As redes (**wire**) apenas transferem dados, se existirem; senão, assumirão o valor padrão (**x**).

- Vetor
  Redes e registradores podem representar mais de um bit, se declarados como vetor.
  O primeiro elemento da definição definirá o bit mais significativo.

  Exemplos:
  ```
  reg [7:0] A;          // registrador de 8 bits
  wire [16:0] B;        // dado em 16 bits

  A = 8'b1100_0101; // atribuição completa
  B [7:0] = A;          // atribuição parcial
  ```

- Números

  - Indicadores de tempo
    Números inteiros positivos serverm representar medidas de tempo (**$time**).

    Definição: **time**
    Tamanho padrão: 64 bits (no mínimo)

    Exemplo:
    ```
    time  inicio, fim;
    ```

  - Inteiros
    Números inteiros podem representar valores positivos ou negativos.

    Definição: **integer**
    Valor padrão: **x**
    Tamanho padrão: 32 bits (no mínimo)

    Exemplo:
    ```
    integer  K;

    K = 1;
    ```

  - Reais
    Números reais podem representar valores em decimal ou em notação científica.
    A vírgula decimal é representada por ponto decimal.
    Deve haver pelo menos um dígito antes e depois do ponto decimal.
    Serão convertidos para o inteiro mais próximo por arredondamento.

    Definição: **real**
    Valor padrão: **x**
    Tamanho padrão: 64 bits (no mínimo)

    Exemplo:
    ```
    real  PI;

    PI = 3.1415;
    ```

- Arranjo
  Registradores, inteiros e indicadores de tempo podem ser declarados como arranjos.

  Formato:
  <tipo> <tamanho do dado> <variável> <tamanho do arranjo>

  Exemplos:

```
reg A [7:0];                          // 8 registradores de 1 bit cada
reg [15:0] mem16_1024 [1023:0];       // memória de 1K por 16 bits
integer [7:0] B [16:0];               // 16 dados de 8 bits cada

A [0] = 1'b1;                         // atribuição ao último registrador
```

- *Tri-state*
  Drivers tri-state podem receber três tipos de valores: 0, 1 ou nenhum deles.
  Se dois valores forem atribuídos simultaneamente a uma rede (*net*) o resultado será indeterminado (**x**); entretanto, poderá ser um valor for de alta impedância, se um dos valores assim o for.

  Exemplos:

```
module triDriver (bus, drive, value);
  inout [7:0] bus;
  input       drive;
  input [7:0] value;

  assign #2 bus = (drive == 1) ? value : 8'bz; // se alto, recebe valor;
                                               // senão ficará em alta impedância
                                               // e poderá receber qualquer outro valor

endmodule // triDriver
```

- Tipos de operadores
  - Aritméticos

Algoritmo	Verilog
* / ***mod***	* / %
+ -	+ -

Observações:
- A divisão de valores inteiros terá resultado inteiro.
- Se um dos operandos for indeterminado (**x**), o resultado também o será.

  - Relacionais

Algoritmo	Verilog
< ≤ > ≥	< <= > >=
= ≠	== === != !===

Observação:
- O resultado de uma comparação poderá ser 0 (falso), 1 (verdadeiro), indeterminado ou de alta impedância.
- Os operadores (=== e !===) comparam todos os tipos de valores inclusive os indeterminados ou os de alta impedância.

  - Conectivos lógicos

Algoritmo	Verilog
não	!
e	&&
ou	||

Observação:
O resultado de uma operação lógica poderá ser 0, 1 ou indeterminado (**x**).

  - Lógicos (bit a bit)

Algoritmo	Verilog
complemento de um	~
e	& ~&
ou-exclusivo (XOR) /	^
XNOR	~^ ^~
ou	| ~|
deslocamento	<< >>

Observações:
- O resultado de uma operação lógica entre dois operandos é um valor cujos bits são operados, um a um, de acordo com a álgebra de proposições.
- O resultado de uma operação lógica aplicada sobre um só operando é chamado de "redução", e forma-se ao operar os dois bits mais a esquerda e continuar com os demais.
  As operações **xor** (^) e **xnor** (~^, ^~) são úteis em verificações de paridade.

Exemplos:

```
module bitoperators;
  reg [3:0] A, B;

  initial begin
    A = 4'b0101;
    B = 4'b1011;

                              // operações binárias
    $displayb ( A & B ); // o mesmo que (0001)
    $displayb ( A  |  B ); // o mesmo que (1111)
    $displayb ( A  ^  B );// o mesmo que (1110)
    $displayb ( A << 1 );// o mesmo que (1010)
    $displayb ( A >> 1 );// o mesmo que (0010)

                              // operações unárias
    $displayb ( & A );    // o mesmo que (0 & 1 & 0 & 1) = 0 (redução)
    $displayb (  | A );    // o mesmo que (0  |  1 |  0  |  1) = 1 (redução)
    $displayb ( ^ A );    // o mesmo que (0  ^  1 ^ 0 ^  1) = 0 (redução)
  end

endmodule // bitoperators
```

- Replicação ({ { } })

A replicação poderá ocorrer quantas vezer for necessário.

Exemplos:

```
module concatenate;
  reg        A;
  reg [7:0] C;

  initial begin
   A = 1'b1;
   C = { 4{A} };

   $displayb ( { 4{a} } );          // mostrará 1111
   $displayb ( c );                 // mostrará 00001111
  end

endmodule // concatenate
```

- Concatenação ({ , })

  O operador de concatenação (,) pode juntar constantes, redes, registradores, bits ou partes.

  Exemplos:

```
module concatenate;
  reg        A;
  reg [2:0] B;
  reg [4:0] C;

  initial begin
    A = 1'b1;
    B = 3'b000;
    C = 5'b10101;

    $displayb ( {a, b} );          // 4-bits iguais a 4'b1000
    $displayb ( {c[5:3], a} );     // 4-bits iguais a 4'b1011

  end

endmodule // concatenate
```

- Prioridade de operadores

Operador	Prioridade
+ - (unários) ! ~	mais alta
* / %	
+ - (binários)	
<< >>	
< <= >= >	
== != === !===	
& ~&	
^ ~^	
| ~|	
&&	
||	
?:	mais baixa

- Expressões
  - Aritmética

    Exemplos:

Algoritmo	Verilog
10 + 15	10 + 15
543.15/3	543.15/3
(x + y + z)*a/z	((x + y + z) * a)/z

  - Lógica

    Exemplos:

Algoritmo	Verilog
A = 0	A == 0
$a \neq 1$	a != 1
$(A \geq 0)$ & $(a \leq 1)$	(A >= 0) && (a <= 1)

    Observação:
    Para efeito de clareza, ou para mudar a precedência de operadores, pode-se separar as proposições por parênteses.

- Funções

  Funções podem retornar valor, mas

  - devem ser executadas em uma unidade de tempo de simulação;
  - não podem ter controles de tempo, diferente das tarefas;
  - não podem invocar tarefas, ao contrário destas que podem chamar outras do seu tipo.

  Formas gerais:

```
function <intervalo ou tipo> <nome>;     // sem parâmetros          (modelo antigo)
   <declaracoes>
   <portas>
   <acoes>
endfunction

function <intervalo ou tipo> <nome> ( <tipo> <parâmetro(s)>);   // (modelo atual)
   <declaracoes>
   <portas>
   <acoes>
endfunction
```

Exemplo:

```
module function1;

  function [1:1] add1;    // definir funcao
    input e1, e2;         // duas entradas
    reg    A;

    begin
      A = 1;
      if ( e1 == e2 )
        add1 = 1 & A;
      else
        add1 = 0;
    end
  endfunction

  initial begin
    reg B;
    B = add1(1, 0);       // chamada de funcao com dois argumentos
    $display( "Saida = %b", B );
  end

endmodule // function1
```

- Tarefas

  Tarefas são como procedimentos, podem ter zero ou vários parâmetros, e não retornam valor.

  Formas gerais:

```
task <nome>;              // sem parâmetros        (modelo antigo)
   <declaracoes>
   <portas>
   <acoes>
endtask

task <nome> ( <tipo(s)> <parâmetro(s)>);        // (modelo atual)
   <declaracoess>
   <portas>
   <acoes>
endtask
```

  Para a chamada de uma tarefa:

```
   <nome da tarefa> (<portas>);
```

Exemplo:

```
module tasks;

  task add;                  // definir a tarefa
    output saida;            // uma saída
    input   e1, e2;          // duas entradas
    reg     A;

    begin
      A = 1;
      if ( e1 == e2 )
        saida = 1 & A;
      else
        saida = 0;
    end
  endtask

  initial begin
    reg B;
    add( B, 1, 0 );          // chamada com tres argumentos
    $display( "Saida = %b", B );
  end

endmodule
```

Combinações de funções e tarefas

Em um mesmo módulo poderão ser combinadas definições e chamadas de funções e/ou tarefas.

Exemplo:

```
module test_methods;

 function [1:1] add1 ( input e1, e2 ); // definir funcao
   reg A;
   begin
     A = 1;
     if ( e1 == e2 )
        add1 = 1 & A;
     else
        add1 = 0;
   end
 endfunction

 task add ( output saida, input e1, e2); // definir tarefa
   reg A;
   begin
     A = 1;
     if ( e1 == e2 )
        saida = 1 & A;
     else
        saida = 0;
   end
 endtask

 reg B;

 initial
  begin: test_function
   B = add1(1, 0); // invocacao da funcao com dois argumentos
   $display( "Function output = %b", B );
   B = add1(1, 1); // invocacao da funcao com dois argumentos
   $display( "Function output = %b", B );
  end

 initial
  begin: test_task
   add( B, 1, 0 ); // invocacao da tarefa com tres argumentos
   $display( "Task output     = %b", B );
   add( B, 1, 1 ); // invocacao da tarefa com tres argumentos
   $display( "Task output     = %b", B );
  end

endmodule // test_methods
```

- Tarefas de sistema
  - Saídas padrões

Tarefas	Padrões
$display	decimal
$displayb	binário
$displayh	hexadecimal
$displayo	octal
$write	decimal
$writeb	binário
$writeh	hexadecimal
$writeo	octal

Observação:
A diferença entre $display e $write é que a primeira muda de linha automaticamente.

Exemplos:

```
$display ( 1'b1 );
$write    ( "\n", 1'b0 );
```

- Formatos:

Formatos	Padrões
%d ou %D	decimal
%b ou %B	binário
%h ou %H	hexadecimal
%o ou %O	octal
%m ou %M	hierárquico
%t ou %T	tempo
%e ou %E	real (notação científica)
%f ou %F	real (decimal)
%g ou %G	real (o menor entre os acima)

- Encerramento de simulação
  A ação **$finish** pára uma simulação e passa o controle ao sistema operacional.
  A ação **$stop** suspende uma simulação e passa ao modo interativo.

  Exemplo:

```
initial begin
  clock = 1'b0;
  . . .              // ações a serem simuladas
  #200 $stop    // suspende a simulação e passa ao modo interativo

  #500 $finish   // encerra todas as ações.
end
```

- Monitoramento
  A ação **$monitor** é semelhante à **$display** e será executada sempre que houver mudança no(s) valor(es) observado(s). As ações **$monitoron** e **$monitoroff** podem ligar ou desligar o monitoramento. Recomenda-se indicar o monitoramento no início de uma simulação.

  Exemplo:

```
module teste;
  integer A, B, C;

  initial begin
    A = 0;   B = 4;   C = 5;
    forever begin
    #5 A = A + B;
    #5 C = C  - 1;
    end // forever
  end

   initial #40 $finish;

   initial begin
     $monitor ($time, " A = %d, B = %d, C = %d ", A, B, C);
   end

endmodule // teste
```

- Controle de tempo

  Existem vários tipos de controle de tempo: atrasos, eventos, esperas e mudanças de nível. Só haverá variação no tempo simulado se acontecer:

  - um atraso em porta ou ligação (**wire**);
  - um atraso controlado (#);
  - um evento controlado (@);
  - uma ação dependente de espera (**wait**).

  - Atrasos (#)

    Exemplos:

```
initial begin
        A = 0;          // início da simulação no tempo = 0
   #10 B = 2;           // 10 unidade   de tempo à frente
   #15 C = A;           // 15 unidades de tempo depois, ou 25 após o início
   #B  C = 4;           // 02 unidades de tempo depois, ou 27 após o início
        B = 5           // 27 unidades após o início
end
                         // atraso associado a uma porta
and # (5) AND1 (saida, entrada1, entrada2)
```

  - Eventos (@)
    Eventos ocorrerão quando houver mudanças em um sinal predefinido ou variável.

    Exemplos:

```
@ (clock)   A = B;              // atribuir valor na mudança do contador de tempo

@ (negedge clock) A = B;        // atribuir valor quando clock for para 0

A = @ (posedge clock) B;        // quando clock for para 1, avaliar e atribuir
```

  - Espera (**wait**)
    Uma ação (ou bloco) deve aguardar até que uma condição seja verdadeira.

    Exemplo:

```
  wait ( A == 1 )
    begin
      A = B & C;
    end
```

- *Triggers*
  Determinações arbitrárias de um ou mais sinais.

  Exemplos:

```
event data_in;                    // evento definido pelo usuário

always @ (posgedge clock)         // sempre que o clock for para 1,
  if ( data [8] ==1 ) -> data_in; // determinar ocorrência do evento

always @ (data_in)                // sempre que houver determinação do evento
  mem[0:1] = buf;                 // receber e guardar dado na memória

                                  // quando ocorrer qualquer um dos eventos
always @ (negedge clock or data_in)
  A = 1'b0;
```

- *Threads* **(fork-join)**
  Construções que se executam concorrentemente. Uma vez iniciadas, somente após terem sido todas completadas, haverá prosseguimento na execução seqüencial.

  Forma geral:

```
fork:   // divisão em vários conjuntos de ações
        begin
          // código para o primeiro conjunto
        end

        begin
          // código para o segundo conjunto
        end

. . .
        begin
          // código para o último conjunto
        end

join    // a partir de onde prosseguirão as ações
        // quando todas as anteriores tiverem sido completadas
```

- Estrutura de programa
  - Blocos
    - Inicial
      Um bloco inicial pode ter um ou mais ações contidas por uma estrutura **begin** ... **end** .
      Se houver mais de um bloco, eles serão executados de forma independente e concorrente. É empregado para dar valores iniciais às variáveis, monitoramento, gerar formas de onda e processos.

      Exemplo:

```
initial
  clock = 1'b0;          // dar valor à variável

initial
  begin                  // gerar forma de onda
       wave = 0;
  #10 wave = 1;
  #20 wave = 0;
  #5   wave = 1;
  #7   wave = 0;
  #10 wave = 1;
  #20 wave = 0;
  end;
```

    - Em repetição infinita
      Um bloco em repetição infinita é precedido pela palavra **always** , e somete poderá ser interrompido por um comando **$finish** ou **$stop** .

      Exemplo:

```
module pulso;

  reg clock;

  initial          clock = 1'b0;      // iniciar o clock em 0
  always #10   clock = ~clock;   // trocar o sinal a cada 10 unidades de tempo
  initial    #500 $finish            // terminar após 500 unidades de tempo

endmodule
```

- Módulos

Módulos servem para encapsular definições (especificações estruturais) ou ações (especificações comportamentais), e podem ser instanciados.

As declarações podem incluir objetos de dados como registradores, memórias e ligacões (**wire**) bem como ações do tipo função (**function**) ou tarefas (**task**).

Os itens podem ser instâncias de outros módulos ou construções do tipo **initial** ou **always**, ou atribuições contínuas. Construções do tipo **initial** ou **always** são usadas para se definir *circuitos seqüenciais*; enquanto atribuições contínuas servem para *circuitos combinatórios*.

Forma geral:

```
module <nome> ( <portas> );
    <declarações>
    <itens>
endmodule
```

Exemplo:

```
                        // definir valores
module teste;
  integer A, B, C;

  initial begin
    A = 0;
    B = 4;
    C = 5;
  end
```

Se houver associações de portas, elas deverão ser definidas como **input**, **output** ou **inout**.

Saídas (**output**) em módulos internos podem ser declaradas como redes (*net*) ou registradores (*reg*), em módulos externos devem ser redes (*net*). Por convenção, as saídas são colocadas no início de listas de portas.

Entradas (**input**) em módulos internos devem ser declaradas como redes (*net*), em módulos externos podem ser redes (*net*) ou registradores (*reg*).

Entradas e saídas (**inout**) devem ser sempre declaradas como redes (*net*).

Ao serem usados, o número e seqüência das portas devem ser rigorosamente observados.

Exemplo:

```
                        // definir componente
module FFD (q, dado, clock);
  output q;
  input  dado, clock;
  reg    q;

  always @(posedge clock)
    q = dado;
endmodule // FFD
```

Instanciação

Forma geral:

<nome do módulo> <parâmetros> <nome da instância> (<portas>);

Exemplo:

```
                              // definir comportamento
module FFD_Teste;
  reg  data, clock;
  wire q;
                              // instanciação
  FFD D1 (q, data, clock);

                              // blocos concorrentes
                              // estímulos
  initial begin
    clock = 1'b0;
    forever  clock = #5 ~clock;
  end

  initial begin
        data = 1'b1;
  #10 data = 1'b1;
  #20 data = 1'b0;
  #30 data = 1'b1;
  #10 data = 1'b0;
  #10 data = 1'b1;
  #10 data = 1'b0;
  #20 $finish;
  end

  initial begin
    $display ($time, "   data,  q ");
    $display ($time, "   %d     %d", data, q);
    forever #10 $display($time, "   %d     %d", data, q);
  end

endmodule // FFD_Teste
```

- Comentários

  Comentários são precedidos pelos sinais // , ou /* */ envolvendo o texto.

  Exemplo:

```
// Esta linha nao faz nada - comentario

/*
   o que também pode ser colocado assim
*/
```

- Atribuição

  - Atribuição bloqueante

    A atribuição bloqueante espera até o fim do bloco para realizar a operação. Ou seja, como se as operações fossem seqüenciais.

    Forma geral:

```
<variável> = <expressão>;
```

    Exemplo:

```
A    =  B;
C    =  A;   // logo C = B
```

  - Atribuição não bloqueante

    A atribuição não bloqueante não espera a conclusão de uma operação para a realização de outra. Ou seja, as operações podem ser realizadas simultaneamente e não há garantias de que um mesmo valor seja atribuído por transitividade.

    Forma geral:

```
<variável> <= <expressão>;
```

    Exemplo:

```
A    <=  B;
C    <=  A;  // C pode ser igual ao valor anterior em A, antes de ser trocado por B
```

Outro exemplo:

```
module blocking;
  reg [7:0]  A, B;

  initial begin:
        A = 3;
   #1 A = A + 1;        // atribuição bloqueante
        B = A + 1;
        $display( "Atribuição bloqueante:  A= %b B= %b", A, B );

         A = 3;
     #1 A <= A + 1;     // atribuição não bloqueante
         B <= A + 1;
     #1 $display( "Atribuição não bloqueante: A= %b B= %b", A, B );
  end

endmodule
```

A saída deverá ser:

```
Atribuição bloqueante:       A= 00000100 B= 00000101
Atribuição não bloqueante: A= 00000100 B= 00000100
```

- Atribuição contínua

A atribuição contínua observa as variáveis do lado direito (expressão) e, caso alguma delas venha a mudar, o valor será reavaliado e atribuído á variável.

A atribuição contínua é recomendada para a definição de circuitos combinatórios.

Forma geral:

assign <variável> = <expressão>;

Exemplo:

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

  output saida;
  input   entrada1, entrada2;
                        // atribuição contínua
  assign saida = ~( entrada1 & entrada2 );

endmodule
```

- Atribuição condicional

Forma geral:

<variável> = <teste> ? <expressão 1>: <expressão 2>;

Exemplo:

X = (A < B) ? A : B;

- Descrição de entrada e saída
  - Saída formatada (padrão):

    Forma geral:

    $monitor (<formato>, <valores>);
    $display (<formato>, <valores>); // e suas variações
    $write (<formato>, <valores>); // e suas variações

  - Entrada/saída formatada em arquivo:
    - Abertura de arquivo

      Forma geral:

      <variável inteira> = **$fopen** (<nome externo do arquivo>);

      Exemplo:

      integer file1, file2; // pode ter até 31 arquivos abertos simultaneamente

      initial begin
        file1 = $fopen("dados1.txt"); // file1 = 0000_0000_0000_0000_0000_0000_0000_0010
        file2 = $fopen("dados2.txt"); // file2 = 0000_0000_0000_0000_0000_0000_0000_0100
      end

    - Fechamento de arquivo

      Forma geral:

      $fclose (<variável inteira>);

      Exemplo:

      $fclose (file1);
      $fclose (file2);

    - Saída formatada em arquivo:

      Forma geral:

      $fmonitor (<formato>, <valores>);
      $fdisplay (<formato>, <valores>);
      $fwrite (<formato>, <valores>);

- Entrada por arquivo:

  Forma geral:

  $readmemb (<nome do arquivo>, <memória>);
  $readmemh (<nome do arquivo>, <memória>);

  $readmemb (<nome do arquivo>, <memória>, <endereço inicial>);
  $readmemh (<nome do arquivo>, <memória>, <endereço inicial>);

  $readmemb (<nome do arquivo>, <memória>, <endereço inicial>, <endereço final>);
  $readmemh (<nome do arquivo>, <memória>, <endereço inicial>, <endereço final>);

  Formato dos dados em arquivo:

  ```
  @ <endereço 1>
  <dado_1>
  . . .
  <dado_m>
  @ <endereço 2>
  <dado_n>
  . . .
  <dado_p>
  ```

  Observação:
  Se os dados forem contíguos pode ser dispensada a indicação do endereço.

  Exemplo:

  ```
  module readmemory;
    reg [7:0] memory [3:0];
    integer   index;

    initial begin
      $readmemb( "dados.txt", memory);

      for( index = 0; index < 4; index = index + 1 )
        $display( "memory[%d] = %b", index, memory[index] );
    end
  endmodule // readmemory
  ```

  No arquivo “dados.txt”:

  ```
  00000000
  00000001
  00000010
  00000011
  ```

- Estruturas de controle
  - Seqüência simples

    Forma geral:

Algoritmo	Verilog
<comando>	<comando> ;
<comando>	<comando> ;

    Observação:
    Em Verilog todos os comandos são separados por ponto-e-vírgula.

  - Estrutura alternativa
    - Alternativa simples

      Forma geral:

Algoritmo	Verilog
se <condição>	**if** (<condição>)
então	**begin**
<comandos>	<comandos> ;
fim se	**end**

      Observação:
      Se houver apenas um comando, os indicadores de bloco podem ser omitidos.

    - Alternativa dupla

      Forma geral:

Algoritmo	Verilog
se <condição> então	**if** (<condição>) **begin**
<comandos 1>	<comandos 1> ;
senão	**end else begin**
<comandos 2>	<comandos 2> ;
fim se	**end**

      Observação:
      Se houver apenas um comando, os indicadores de bloco podem ser omitidos.

- Alternativa múltipla

Forma geral:

Algoritmo	Verilog
escolher <lista de valores>	**case** (<lista de valores>)
<opção 1>: <comando 1>	<expressão_1>: <comando 1> ;
<opção 2>: <comando 2>	<expressão_2>: <comando 2> ;
. . .	. . .
<opção n-1>: <comando N-1>	<expressão_n-1>:<comando N-1> ;
senão <comandos N>	**default**: <comando N>;
fim escolher	**endcase**

Observações:
- Recomenda-se usar uma lista de valores pequena (em torno de 4 valores, no máximo).
- As expressões deverão ter como resultados valores escalares válidos.
- A indicação ***default*** é opcional.
- São permtidas as seguintes variações do comando:

case : para verificar se conferem todos os valores 0, 1, **x** e **z** (inclusive)
casex : para verificar se conferem todos os valores, exceto **x** e **z**, irrelevantes (**?**)
casez : para verificar se conferem todos os valores, exceto **z**, irrelevante (**?**)

Exemplos:

```
. . .
case (sinal)
  1'bz:    $display("Sinal ainda indefinido");
  1'bx:    $display("Sinal desconhecido");
  default: $display("Sigal = %b", sinal);
endcase
. . .
```

```
module multiplexor4_1 (saida, e1, e2, e3, e4, ctrl1, ctrl2);
  output saida;
  input  ctrl1, ctrl2, e1, e2, e3, e4;
  reg    saida;

  always @( ctrl1 | ctrl2 | e1 | e2 | e3 | e4 )
  case ( {ctrl2, ctrl1} )              // concatenação permitida
    2'b00 :  saida = e1;
    2'b01 :  saida = e2;
    2'b10 :  saida = e3;
    2'b11 :  saida = e4;
    default : $display( "ERO: Verificar os bits de controle" );
  endcase
endmodule // multiplexor4_1
```

- Estrutura repetitiva

  - Repetição com teste no início

    Forma geral:

Algoritmo	Verilog
repetir enquanto <condição>	**while** (<condição>) **begin**
<comandos>	<comandos> ;
fim repetir	**end**

    Observação:
    A condição para execução é ser sempre verdadeira.

    Exemplo:

```
integer count;

count =0;
while ( count < 10 )
  begin
    $display("%d", count);
      count = count + 1;
  end
```

  - Repetição com teste no início e variação

    Forma geral:

Algoritmo	Verilog
repetir para <variável> = <valor inicial>	**for**(<valor inicial> ;
: <fim>	<teste de fim> ;
: <variação>	<variação>) **begin**
<comandos>	<comandos> ;
fim repetir	**end**

    Observações:
    A condição para execução é ser sempre verdadeira.
    Qualquer um dos elementos, ou mesmo todos, podem ser omitidos. Entretanto, se tal for necessário, recomenda-se o uso de outra estrutura mais apropriada.

    Exemplo:

```
integer count;

for ( count =0; count < 10; count = count + 1 )
  begin
    $display("%d", count);
  end
```

- Repetição por um número determinado de vezes

Forma geral:

Algoritmo	Verilog
repetir <vezes>	**repeat** (<valor>) **begin**
<comandos>	<comandos> ;
	<alteração do valor>
fim repetir	**end**

Observação:
O controle do número de vezes é feito pelo valor numérico de uma constante, variável ou sinal, apenas *uma vez* antes de executar o bloco, mesmo que o valor varie na execução.

Exemplo:

```
module countdown;
  integer count;

  initial begin
    count = 128;
    repeat (count) begin
      $display("%d time units left", count);
      count = count - 1;
    end
  end

endmodule
```

- Repetição infinita

Forma geral:

Algoritmo	Verilog
repetir	**forever** <comando>;
<comando>	
fim repetir	

Observação:
- A repetição só poderá ser encerrada por um comando **$finish** .
- Recomenda-se usar controle de tempo para limitar a execução.

Exemplo:

```
  reg clock;

  initial begin
    clock = 1'b0;
    forever #10 clock = ~clock;        // mudar o clock a cada 10 unidades de tempo
  end

  initial #200 $finish;                // parar a simulação após 200 unidades de tempo
```