SystemVerilog: Direto ao Ponto

Prof. Lucas Vinicius Hartmann

1 Comentários

Comentários são trechos do código que não tem função lógica, sendo completamente ignorados pelo compilador. Sua função é apenas ajudar pessoas lendo o código fonte a entender o que está sendo feito. Em SystemVerilog existem duas formas de criar comentários:

```
// Comentário só até o fim da linha começa com barra-barra.
/* Ou um comentário de várias linhas começa com barra-asterisco
e termina com asterisco-barra */
```

Como o texto da descrição é armazenado em formato ASCII é conveniente evitar o uso de caracteres acentuados e cedilhas, pois estes caracteres não fazem parte do código ASCII e podem causar problemas de compatibilidade entre diferentes computadores.

2 Variáveis, Constantes e Tipos de Dados

SystemVerilog é uma linguagem de descrição de circuitos lógicos, por consequência todos os valores da linguagem são representados em forma binária. Desta forma apenas um único tipo é suficiente para a grande maioria das descrições mais básicas: `logic`. O tipo `logic` representa uma informação de 1 bit, e pode assumir nível lógico baixo (`0`), nível lógico alto (`1`), alta impedância (`z`), e “não importa” ou “desconhecido” (`x`).

2.1 Nomes de Variáveis

Em SystemVerilog os nomes podem ser compostos por letras, números e traços de sublinhado, mas não podem começar com um número. Deve-se prestar atenção que maiúsculas e minúsculas são diferentes, ou seja `Variavel` é diferente de `variavel`.

2.2 Variáveis de 1 Bit

Para criar uma variável de um bit com nome `SinalX` se usa a seguinte notação:

```
logic SinalX;
```

Para acessar o valor desta variável posteriormente é utilizado apenas o nome da variável, ou seja, `SinalX`.

2.3 Variáveis Maiores que 1 Bit

Para representar informações numéricas inteiras com mais de um bit se usam palavras. Para declarar uma palavra de 8 bits chamada `PalavraY` se usa a seguinte notação:

```
logic [7:0] PalavraY;
```

Quando se deseja acessar a palavra completa utiliza-se simplesmente o nome da variável, por exemplo `PalavraY`. Quando se deseja utilizar apenas um bit específico se utiliza a notação de colchetes, por exemplo `PalavraY[0]` ou `PalavraY[7]` para os bits menos e mais significativos. Quando se deseja acessar uma faixa de bits utiliza-se o sinal de dois pontos para representar a faixa, por exemplo `PalavraY[3:0]` para os 4 bits menos significativos.

2.4 Memórias

Uma memória um agrupamento de palavras de mesmo tamanho. Para criar uma memória chamada Dados com 128 palavras de 8 bits cada utiliza-se a seguinte notação:

```
logic [7:0] Dados [0:127];
```

Para acessar uma palavra específica da memória é utilizada novamente a notação de colchetes, por exemplo `Dados[0]` e `Dados[127]` para a primeira e última palavras da memória. Um segunda coordenada pode ser adicionada para acessar bits específicos de uma palavra, como `Dados[13][5:3]` para acessar os bits 5 a 3 da palavra na posição 13 da memória.

2.5 Constantes Numéricas

Conforme comentado anteriormente todos os valores em SystemVerilog são fundamentalmente representados em forma binária, no entanto isto não significa que o usuário da linguagem precisa escrever apenas zeros e uns. Para representar o número

3456 decimal como uma constante de 16 bits é utilizada a seguinte notação:

```
16'd3456
```

Onde `16'` informa que trata-se de um número de 16 bits, `d` informa que o número foi escrito em base decimal, e `3456` é o número na base selecionada. As bases disponíveis são binário (b), octal (o), decimal (d) e hexadecimal (h). Traços de sublinhado podem ser utilizados para ajudar a tornar números extensos mais legíveis, por exemplo o número de pessoas no mundo é aproximadamente 6 bilhões e pode ser expresso como uma grandeza de 64 bits utilizando `64'd6_000_000_000`.

Números decimais também podem ser escritos diretamente sem a especificação do número de bits ou da base, sendo utilizados tantos bits quanto necessário para descrever o valor. Por exemplo o valor `16'd3456` também poderia ser escrito apenas como `3456`, porém neste caso seria representado com apenas 12 bits pois são suficientes para escrever até 4095.

Nas bases binária, octal e hexadecimal também se pode utilizar os símbolos x e z como parte do número. A notação `4'b0zz1` representa um número de 4 bits onde os bits 1 e 2 estão em alta impedância, e a notação `5'b101xx` representa um número de 5 bits onde os bits mais altos são `101` e os mais baixos não importam. Quando usada a notação octal cada símbolo representa um grupo de 3 bits, portanto a notação `9'o77z` representa um número de 9 bits onde os bits 8 a 3 estão em nível alto e os bits 2 a 0 estão em alta impedância. Da mesma forma na base hexadecimal cada simbolo representa um grupo de 4 bits, portanto `16'xx00` representa um número de 16 bits onde os 8 bits menos significativos estão em nível lógico baixo e os 8 bits mais significativos não importam.

3 Expressões e Operadores

Um circuito lógico consiste basicamente de um conjunto de sinais lógicos que armazenam ou transportam valores digitais, e um conjunto de operações lógicas que descrevem as relações entre estes sinais. Em SystemVerilog as relações lógicas entre sinais podem ser representadas por expressões lógicas, as são escritas como um

conjunto de operações terminadas por ponto-e-vírgula (`;`).

3.1 Operadores de Atribuição = e <=

Em SystemVerilog existem dois operadores de atribuição, a diferença entre estes será discutida no Capítulo 5.3. Para atribuir ou copiar o valor de uma variável `B` para uma variável `A` com o mesmo número de bits é utilizado um dos operadores de atribuição, igual ou menor-igual, como mostrado abaixo. Lembre que ponto e vírgula representa o fim da expressão.

```
A = B;
A <= B;
```

Para atribuir apenas partes da palavra pode-se utilizar a notação de colchetes. Por exemplo, para copiar os bits 7 e 6 de `Entrada` pra os bits 2 e 1 de `Saida` é utilizada a notação:

```
Saida[2:1] <= Entrada[7:6];
```

3.2 Operadores Lógicos

As operações lógicas básicas são representadas pelos caracteres “e comercial” (`&`) para AND, barra vertical (`|`) para OR, acento circunflexo (`^`) para XOR, e til (`~`) ou ponto de exclamação (`!`) para NOT. Considerando sinais de apenas 1 bit estas operações seriam suficientes para descrever circuitos. No entanto tratar bits individualmente seria extremamente trabalhoso em palavras grandes. SystemVerilog define então um conjunto de operadores que permite estender as operações a palavras inteiras.

3.2.1 Operadores por Bit

Os operadores por bit aplicam uma mesma operação lógica a cada um dos bits das palavras individualmente, retornando uma palavra de resposta com o mesmo tamanho das entradas. Por exemplo se `A`, `B` e `C` são palavras de 4 bits então:

```
C <= A & B;
```

É o mesmo que:

```
C[0] <= A[0] & B[0];
C[1] <= A[1] & B[1];
C[2] <= A[2] & B[2];
C[3] <= A[3] & B[3];
```

Operação	Código	A	B	C
AND	C <= A & B;	4'b0011	4'b0101	4'b0001
OR	C <= A | B;	4'b0011	4'b0101	4'b0111
XOR	C <= A ^ B;	4'b0011	4'b0101	4'b0110
NOT	C <= ~A;	4'b0011		4'b1100

3.2.2 Operadores de Redução

Os operadores de redução aplicam uma operação lógica a todos os bits de uma palavra retornando uma saída de apenas um bit representando o resultado. Por exemplo se A é uma palavra de 4 bits e B é um único bit então:

```
B <= &A;
```

É o mesmo que:

```
B <= A[0] & A[1] & A[2] & A[3];
```

Operação	Código	A	B
AND	B <= &A;	2'b00	1'b0
		2'b01	1'b0
		2'b10	1'b0
		2'b11	1'b1
OR	B <= |A;	2'b00	1'b0
		2'b01	1'b1
		2'b10	1'b1
		2'b11	1'b1
XOR	B <= ^A;	2'b00	1'b0
		2'b01	1'b1
		2'b10	1'b1
		2'b11	1'b0

3.2.3 Operadores por Palavra

Os operadores por palavra tratam como informação lógica a palavra completa, e não um único bit. Uma palavra com valor igual a zero é considerada como nível lógico baixo, enquanto qualquer palavra diferente de zero (por exemplo 3) representa o nível lógico alto. O resultado é retornado como um valor de um único bit. Este efeito pode ser descrito como se cada palavra de entrada fosse reduzida pela operação OR antes de ser avaliada pelo operação selecionada.

Operação	Código	A	B	C
AND	`C <= A && B;`	`4'b0000`	`4'b0000`	`1'b0`
		`4'b0000`	`4'b0101`	`1'b0`
		`4'b0011`	`4'b0000`	`1'b0`
		`4'b1000`	`4'b0010`	`1'b1`
OR	`C <= A || B;`	`4'b0000`	`4'b0000`	`1'b0`
		`4'b0000`	`4'b0101`	`1'b1`
		`4'b0011`	`4'b0000`	`1'b1`
		`4'b1000`	`4'b0010`	`1'b1`
NOT	`C <= !A;`	`4'b0000`		`1'b1`
		`4'b0110`		`1'b0`

3.2.4 Operadores de Deslocamento Lógico

É possível deslocar os bits de uma palavra para a esquerda ou direita utilizando os operadores menor-menor (<<) ou maior-maior (>>) respectivamente. Bits deslocados para fora da palavra são descartados, e bits que “vem de fora” são sempre zeros. Por exemplo para palavras A e B com 4 bits cada escrever:

```
B <= A >> 1;
```

é o mesmo que escrever:

```
B[0] <= A[1];
B[1] <= A[2];
B[2] <= A[3];
B[3] <= 0;
```

Operação	Código	A	B
Rotacionar Direita	`B <= A >> 1;`	`4'b1011`	`4'b0101`
	`B <= A >> 2;`	`4'b1011`	`4'b0010`
Rotacionar Esquerda	`B <= A << 1;`	`4'b1101`	`4'b1010`
	`B <= A << 2;`	`4'b1101`	`4'b0100`

3.3 Operadores Aritméticos

Em SystemVerilog são oferecidos também operadores aritméticos os quais realizam as quatro operações básicas de soma (+), subtração (-), multiplicação (*), divisão (/), resto de divisão (%) e potenciação (**) sobre valores numéricos armazenados em palavras. Note que estes operadores são traduzidos para um conjunto de operações lógicas com os bits das palavras envolvidas, então uso dos operadores mais complexos `* / % e **` podem facilmente gerar circuitos demasiadamente extensos para serem implementados em circuito.

Operação	Código	A	B	C
Soma	`C <= A + B;`	`32'd100`	`32'd3`	`32'd103`
Subtração	`C <= A - B;`	`32'd100`	`32'd3`	`32'd97`
Multiplicação	`C <= A * B;`	`32'd100`	`32'd3`	`32'd300`
Divisão	`C <= A / B;`	`32'd100`	`32'd3`	`32'd33`
Resto de divisão	`C <= A % B;`	`32'd100`	`32'd3`	`32'd1`
Potenciação	`C <= A ** B;`	`32'd100`	`32'd3`	`32'd1000000`

3.3.1 Operadores de Deslocamento Aritmético

Os operadores de deslocamento aritmético >>> e <<< trabalham de forma muito semelhante aos operadores de deslocamento lógico, porém consideram que o valor contido na palavra pode representar um número negativo em complemento de 2. Para manter a informação de sinal do número a operação de rotação para a direita irá

preservar o valor do bit de sinal, ou seja, do bit mais significativo da palavra. Por exemplo para palavras A e B com 4 bits cada escrever:

```
B <= A >>> 1;
```

é o mesmo que escrever:

```
B[0] <= A[1];
B[1] <= A[2];
B[2] <= A[3];
B[3] <= A[3];
```

Operação	Código	A	B
Rotacionar Direita	`B <= A >>> 1;`	`4'b1011`	`4'b1101`
	`B <= A >>> 2;`	`4'b1011`	`4'b1110`
Rotacionar Esquerda	`B <= A <<< 1;`	`4'b1101`	`4'b1010`
	`B <= A <<< 2;`	`4'b1101`	`4'b0100`

3.4 Operadores de Comparação

Operadores de comparação são utilizados para a obtenção de uma informação lógica a partir da comparação de valores de valores armazenados em palavras.

Operação	Código	A	B	C
Maior	`C <= A > B;`	`32'd100`	`32'd3`	`1'b1`
Maior ou igual	`C <= A >= B;`	`32'd100`	`32'd3`	`1'b1`
Igual	`C <= A == B;`	`32'd100`	`32'd3`	`1'b0`
Menor ou igual	`C <= A <= B;`	`32'd100`	`32'd3`	`1'b0`
Menor	`C <= A < B;`	`32'd100`	`32'd3`	`1'b0`
Diferente	`C <= A != B;`	`32'd100`	`32'd3`	`1'b1`

3.5 Operador de Concatenação e Replicação

Para agrupar diversos bits ou palavras formando uma nova palavra de maior dimensão é utilizado o operador de concatenação `{ }`. Também é possível expressar quantas vezes um dado fragmento deve ser repetido na nova palavra.

Operação	Código	A	B	C
Concatenação	`C <= {A, B};`	`8'hAB`	`8'hCD`	`16'hABCD`
	`C <= {A[7:4], B};`	`8'hAB`	`8'hCD`	`12'hACD`
	`C <= {A[2], B[1]};`	`8'b011`	`8'b10`	`2'b01`
	`C <= {A, B, B, A};`	`8'hAB`	`8'hCD`	`16'hABCDCDAB`
Concatenação e Replicação	`C <= {2{A}};`	`8'hAB`		`16'hABAB`
	`C <= {4{A[7:4]}};`	`8'hAB`		`16'hAAAA`
	`C <= {2{A[2]}, B};`	`3'b100`	`2'b00`	`4'b1100`

3.6 Operador Condicional

O operador condicional trabalha de forma semelhante a um multiplexador, possuindo duas entradas de sinal e um seletor, e é escrito na seguinte notação:

```
Y <= SEL ? XV : XF;
```

Se o sinal seletor SEL for verdadeiro Y recebe o valor de XV, se o seletor for falso Y recebe o valor de XF. Tanto o seletor quanto as entradas para verdadeiro e falso podem ser substituídos por expressões. Por exemplo para obter o valor absoluto de um número X pode-se utilizar o seguinte código:

```
Y <= X>=0 ? X : -X;
```

4 Estruturas de Controle

Frequentemente um problema exige soluções que seriam demasiadamente complexas de descrever apenas com as operações lógicas básicas. Nestes casos podem ser utilizadas estruturas de controle condicionais para simplificar o processo.

4.1 Estrutura Condicional IF-ELSE

Quando se deseja executar dois conjuntos distintos de comandos dependendo de uma condição ser atendida ou não, então utiliza-se a estrutura condicional `if-else` como mostrado na listagem a seguir.

```
if (condição)
begin
    // Atribuições ou outras estruturas de controle.
    // Executam apenas quando a condição for verdadeira.
end
else
begin
    // Atribuições ou outras estruturas de controle.
    // Executam apenas quando a condição for falsa.
end
```

Também é possível concatenar uma sequência de testes na seguinte forma:

```
if (condição1)
begin
    // Atribuições ou outras estruturas de controle.
    // Executam apenas quando a condição1 for verdadeira.
end
else if (condição2)
begin
    // Atribuições ou outras estruturas de controle.
    // Executam apenas quando a condição1 for falsa e a condição2
    // verdadeira.
end
else
begin
    // Atribuições ou outras estruturas de controle.
    // Executam apenas quando ambas as condições forem falsas.
end
```

Se for necessário apenas um único comando dentro de um bloco `if` ou `else` então `begin` e `end` podem ser omitidos. Também espaços e quebras de linha são ignorados pelo compilador, então pode-se construir uma estrutura `if-else` simples de forma mais compacta como listado abaixo.

```
if (somar == 0) y = A - B;
else            y = A + B;
```

4.2 Estrutura Condicional CASE

Quando se deseja usar uma variável para selecionar uma de diversas opções pode-se utilizar o condicional `case`. Este condicional compara o valor de uma variável (SELETOR) com uma lista de constantes, e executa apenas um conjunto de comandos

selecionado pela variável.

```
case (SELETOR)
4'b0100:
    Y <= A + B; // Executa apenas quando SELETOR == 4
4'b0010:
    Y <= A - B; // Executa apenas quando SELETOR == 2
4'b0101:
    Y <= A & B; // Executa apenas quando SELETOR == 5
4'b1100:
    Y <= A | B; // Executa apenas quando SELETOR == 12
default:
    Y <= 0;     // Executa quando o SELETOR não aparece na lista.
endcase
```

4.3 Estrutura Condicional CASEX

O condicional `casex` opera de maneira muito semelhante ao condicional `case`, mas permite também o uso de bits 'não-importa' (`x`) na lista de constantes. Isto é útil na simplificação de descrições, pois evita a repetição desnecessária de código.

```
casex (SELETOR)
4'bx100:
    Y <= A + B; // Executa quando o SELETOR é 4 ou 12.
4'b001x:
    Y <= A - B; // Executa quando o SELETOR é 2 ou 3.
4'b0x01:
    Y <= A & B; // Executa quando o SELETOR é 1 ou 5.
4'b10x0:
    Y <= A | B; // Executa quando o SELETOR é 8 ou 10.
default:
    Y <= 0;     // Executa quando o SELETOR não aparece na lista.
endcase
```

5 Descrição de Circuitos Lógicos

Circuitos lógicos de um modo geral podem ser divididos em dois grandes grupos quanto à presença de um sinal de relógio. Os circuitos combinacionais implementam expressões lógicas que dependem exclusivamente dos valores atuais das entradas, e tem sua saída alterada assim que qualquer uma das entradas for alterada. Por outro lado os circuitos sequenciais dependem também de estados anteriores das entradas e saídas, utilizam-se de flip-flops, e geralmente alteram o estado de suas saídas apenas quando um

sinal de relógio é acionado. Ambos estes circuitos podem ser descritos utilizando SystemVerilog.

5.1 Circuitos Combinacionais

Para a implementação de circuitos combinacionais é utilizado o bloco `always_comb`, em uma notação como a mostrada abaixo. Note que o operador de atribuição para circuitos combinacionais é apenas o sinal de igual.

```
always_comb
begin
    // Atribuições combinacionais, como por exemplo
    X = A & B;
    Y = X;
    // Também podem ser utilizadas as estruturas de controle.
end
```

Entre as linhas de `begin` e `end` podem ser utilizadas atribuições de lógica combinacional, e as saídas `X` e `Y` serão atualizadas assim que `A` ou `B` mudarem de valor. Também podem ser utilizadas as estruturas de controle `if-else`, `case` e `casex` na descrição da lógica combinacional, mas deve-se observar que sejam atribuições completas. Por atribuição completa entenda que se uma variável `Y` receber uma atribuição em uma das condições, então ela deve receber atribuição em todas as condições. Ou seja, se `Y` recebe uma atribuição no bloco do `if` então ela deve obrigatoriamente receber uma atribuição no bloco do `else`, e da mesma forma se `Y` recebe uma atribuição numa das condições do `case` ou `casex` então ele deve receber atribuições em todos os demais.

```
always_comb
begin
    if (SELETOR == 0)
    begin
        Y = X0;
    end
    else
    begin
        Y = X1;
    end
end
```

5.2 Circuitos Sequenciais

Para a implementação de circuitos síncronos é utilizado um bloco `always_ff`, e o operador de atribuição <= (menor igual) como mostrado abaixo:

```
always_ff @(posedge CLK)
begin
    // Atribuições sequenciais, como por exemplo
    X <= A & B;
    Y <= X;
    // Também podem ser utilizadas as estruturas de controle.
end
```

Note que as expressões lógicas utilizadas são semelhantes às do exemplo anterior, porém por se tratar de um circuito sequencial as saídas são flip-flops acionados pela borda de subida do sinal CLK, e somente terão seu valor alterado quando este sinal variar de acordo.

Deve-se notar que as atribuições de lógica sequencial são executadas todas simultaneamente e de forma paralela. Isto implica que, no exemplo anterior, Y receberá o valor que X possuía logo antes de borda de subida do relógio.

Estado Anterior			CLK	Estado Final	
A	B	X		X	Y
4'b0011	4'b0101	4'b0000	↑	4'b0001	4'b0000
4'b1100	4'b1010	4'b0001	↑	4'b1000	4'b0001
4'b1111	4'b1111	4'b1000	↑	4'b1111	4'b1000
4'b0101	4'b1010	4'b1111	↑	4'b0000	4'b1111

Caso seja necessário também é possível utilizar a borda de descida em vez da borda de subida, bastando para isto substituir `posedge` por `negedge` como mostrado abaixo.

```
always_ff @(negedge SINAL)
```

É possível criar circuitos sensíveis a ambas as bordas de um sinal utilizando:

```
always_ff @(posedge SINAL or negedge SINAL)
```

E da mesma forma pode-se criar circuitos sensíveis a bordas de sinais distintos utilizando:

```
always_ff @(posedge SINAL1 or posedge SINAL2)
```

Pode-se também utilizar as estruturas de controle `if-else`, `case` e `casex` em um bloco de lógica sequencial. Diferente dos circuitos combinacionais, no entanto, as atribuições não precisam ser completas. Se uma variável Y recebe uma atribuição em um bloco `if` mas não em um `else` então assume-se que no `else` Y deve manter seu valor anterior. Desta forma os blocos `else` (de `if`) e `default` (de `case` e `casex`) se tornam opcionais.

5.3 Revisitando os Operadores de Atribuição

O operador de atribuição = (igual) pode ser utilizado em lógica sequencial para emular o comportamento de linguagens de programação. No exemplo abaixo se fosse utilizado sempre o operador <= então A, B e X receberiam respectivamente os valores de X, Y, e X imediatamente anteriores à borda de subida do clock. No entanto, como o operador = emula o comportamento de liguagens de programação, a variável A receberá o valor antigo de X, mas a variável B receberá o novo valor de X. Note que não existe diferença entre os operadores quando a variável que recebe atribuição não é utilizada em outras expressões dentro do mesmo bloco `always_ff`, como é o caso das variáveis A e B.

```
always_ff @(posedge clk)
begin
    A <= X; // Usa o valor antigo de X
    X  = Y; // A partir deste ponto X vale Y
    B <= X; // Usa o valor novo de X, ou seja, Y
end
```

Deve-se ressaltar que se as expressões de X e B não fossem triviais então o atraso de propagação para a expressão de B seria acumulado ao da expressão de X. Este acúmulo de atrasos de propagação tende a limitar a máxima frequência de operação do circuito e, portanto, deve ser utilizado com cautela.

6 Módulos

Módulos são unidades da descrição que contem entradas e saídas, e uma descrição das relações entre estas. É possível construir circuitos lógicos a partir de uma hierarquia em árvore onde um módulo raiz contém n submódulos, e cada submódulo mais m outros módulos. Desta forma pode-se organizar uma descrição de circuito lógico complexa fracionando o problema total em pequenas unidades com problemas mais simples, de maior facilidade de implementação e de verificação de erros.

6.1 Definição de Módulos

A definição de módulos é composta por 3 partes fundamentais: identificação do módulo, lista de portas e descrição da funcionalidade. No exemplo abaixo, um módulo chamado `minha_and` possui duas entradas lógicas `A` e `B`, e uma saída `Y` dada pela operação AND. É recomendado que cada módulo seja descrito em um arquivo fonte distinto (texto puro ASCII), de preferência com o mesmo nome do módulo mais o sufixo da linguagem. Para a linguagem SystemVerilog a extensão utilizada é `.sv`, portanto no caso do módulo `minha_and` o nome do arquivo recomendado é `minha_and.sv`.

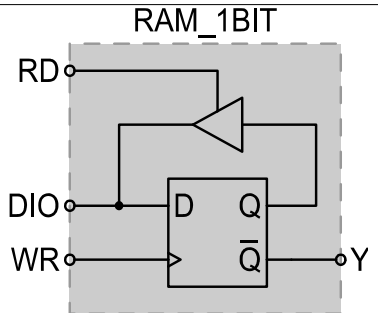
```
// Identificação do módulo
module minha_and(
    // Lista de portas
    input logic A, B,
    output logic Y
);

    // Descrição da funcionalidade
    always_comb
    begin
        Y = A && B;
    end

endmodule
```

A lista de portas contém a descrição dos sinais do módulo que fazem interface com o restante do circuito. As portas são declaradas de forma semelhante às variáveis lógicas e podem conter um ou mais bits e uma ou mais palavras, mas diferem pois devem conter também uma definição de direcionalidade. As definições de direcionalidade especificam para cada porta do módulo se o sinal representa uma entrada, uma saída, ou um sinal bidirecional. Todos os tipos de portas podem ser utilizados como valores em expressões lógicas, mas diferem quando à aceitação de atribuições. Portas de entrada são definidas com a denominação `input` e nunca devem receber atribuições dentro do módulo. Portas de saída são definidas com a denominação `output` e devem obrigatoriamente receber atribuições no módulo. Portas bidirecionais são definidas com a denominação `inout`, e podem ou não receber atribuições. A direcionalidade de uma porta `inout` é controlada pelo valor atribuído a ela no módulo, operando como entrada enquanto a porta recebe uma atribuição de alta impedância (`z`) e como saída em caso contrário.

Um exemplo de aplicação para portas `inout` é na descrição de barramentos de dados de uma memória, como na imagem a seguir. O módulo `RAM_1BIT` possui dois sinais de entrada `RD` e `WR`, um sinal de saída `Y`, e um sinal `DIO` que pode servir tanto como entrada quanto como saída. Nesta memória o valor `Q` do *flip-flop* é acessado por meio do barramento de dados DIO. A gravação de um valor na memória é uma operação de lógica sequencial, e portanto descrita com `always_ff`. O valor desejado deve ser aplicado ao sinal `DIO` e então acionado o sinal `WR`, cuja transição de subida ativa o *flip-*

flop fazendo com que este armazene o valor de `DIO`. A operação de leitura é controlada pelo *buffer tristate*, logo é uma operação de lógica combinacional e deve ser descrita com `always_comb`. Quando o sinal `RD` estiver ativo o o *buffer tristate* comunica o valor de `Q` para o barramento de dados `DIO` por meio da atribuição `DIO <= Q` e, de forma complementar, quando o sinal `RD` estiver inativo o *buffer tristate* deve manter o sinal `DIO` em alta impedância por meio da atribuição `DIO <= 1'bz`.

```
module RAM_1BIT(
    input  logic RD, WR,
    inout  logic DIO,
    output logic Y
);

    // Sinal interno, estado do flip-flop
    logic Q;

    // Operação de escrita
    always_ff @(posedge WR)
        Q <= DIO;

    // Operação de leitura
    always_comb
        if (RD) DIO = Q;    // DIO = saída
        else    DIO = 1'bz; // DIO = entrada em alta impedância

    // Sinal de saída simples
    always_comb
        Y = !Q;

endmodule
```



6.2 *Instanciação de Módulos*

Quando um módulo utiliza em sua construção um bloco lógico definido, geralmente em um arquivo distinto, por outro módulo diz-se que o primeiro está instanciando o segundo. Este mecanismo é extremamente útil na representação de blocos repetitivos da descrição, ou quando se deseja gerenciar a complexidade de um grande problema dividindo-o em diversos problemas menores e de solução mais simples. Considere, por exemplo, que um engenheiro descreva um decodificador de display de 7 segmentos utilizando SystemVerilog. Para tal é definido um módulo `dec7seg` com uma entrada X de 4 bits que informa em base binária o dígito hexadecimal a ser exibido, e uma saída Y de 7 bits contendo a informação de liga/desliga para cada segmento do *display*. Se posteriormente o engenheiro decidir criar um relógio ele não precisará reescrever a lógica de decodificação de *display* para cada dígito, bastará instanciar 4 cópias do módulo `dec7seg` conectando-as aos sinais de interesse.

Quando se instancia um módulo cria-se uma relação de parentesco. Um módulo que instancia outros é chamado de módulo pai, e um módulo que é instanciado por outro é chamado módulo filho. Cada módulo pode instanciar tantos outros módulos quantos forem necessários, criando uma estrutura de árvore. Deve existir também um único módulo que não é instanciado por outros, este módulo é denominado módulo raiz, principal ou *top-level design entity*, e suas portas descrevem as conexões do circuito para o mundo externo.

A instanciação de um módulo é feita de forma semelhante a declaração de variáveis, especificando-se o tipo do módulo filho e o nome da instância, porém é necessário especificar como serão conectadas suas portas.

```
nome_modulo nome_instancia(lista,de,conexoes);
```

Em SystemVerilog a conexão de portas pode ser especificada de três formas distintas: Por posição, por nome ou por casamento. Note, porém, que o módulo pai somente possui acesso às portas do módulo filho, e não a seus sinais lógicos internos.

Na conexão por posição o módulo pai especifica, na lista de conexões, os nomes dos sinais do módulo pais que devem ser conectados às portas do filho na mesma ordem em que foram declarados no módulo filho. Esta á forma mais simples para conectar sinais a portas com nomes distintos, mas é de difícil manutenção. Imagine que o módulo filho sofra uma edição um nova porta venha a ser adicionada, então todas as instanciações devem ser corrigidas para incorporar a nova porta. Devido ao risco de erros esta notação geralmente é utilizada apensa quando o módulo filho possui poucos sinais, e é pouco provável que ele venha a ser modificado.

```
// Conecta sinal1, sinal2 e sinal3 do módulo pai
// com as portas do módulo, na mesma ordem em que foram
// declaradas na criação do módulo
nome_modulo nome_instancia(sinal1, sinal2, sinal3);
```

Na conexão por nomes especifica-se individualmente a qual porta devem ser conectados os sinais do módulo pai. Para tal, na lista de conexões usa-se a notação `.porta(sinal)`, que conecta o sinal de nome `sinal` do módulo pai à porta de nome `porta` do módulo filho. Com esta notação é possível especificar as conexões em

qualquer ordem, ou deixar portas desconectadas caso desejado. Também as conexões permanecem coerentes mesmo que o módulo filho venha a ser modificado, reduzindo problemas durante o desenvolvimento da descrição. A única desvantagem desta forma é que, para módulos filhos com centenas de sinais, a descrição pode se tornar tediosa.

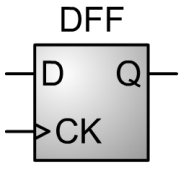
```
// Conecta porta2 do módulo filho com sinal2 do módulo pai e
// e       porta1 do módulo filho com sinal1 do módulo pai.
// Outras portas ficam desconectadas, se existirem.
nome_modulo nome_instancia(.porta2(sinal2), .porta1(sinal1));
```

Finalmente a descrição por casamento é realizada especificando .* como o último argumento da lista de conexões. Este marcador afeta todas as portas do módulo filho que ainda não foram conectadas, e cria conexões com os sinais de mesmo nome no módulo pai. Desta forma é possível várias (ou todas) as portas de um módulo com grande agilidade.

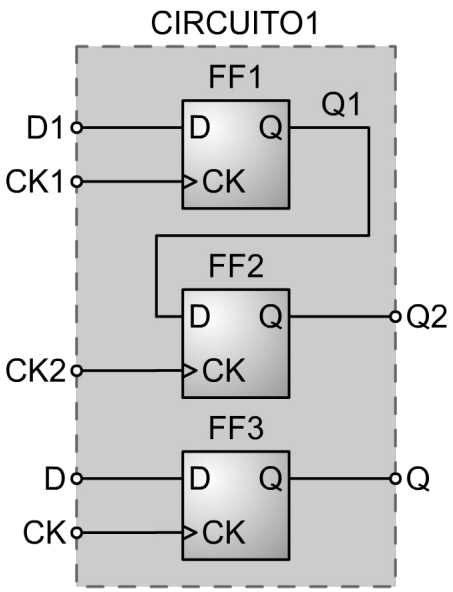
```
// Conecta todos as portas do módulo filho aos sinais de mesmo
// nome no módulo pai.
nome_modulo nome_instancia_1(.*);

// Conecta a porta2 do módulo filho ao sinal2 do módulo pai, e as
// demais portas do filho aos sinais de mesmo nome no módulo pai.
nome_modulo nome_instancia_2(.porta2(sinal2), .*);
```

Para demonstrar consideremos como módulo filho o *flip-flop* tipo D abaixo, e que a partir deste desejamos construir CIRCUITO1 como mostrado na figura abaixo. Note que FF1, FF2 e FF3 são instâncias do módulo de tipo `dff`.

```
// Flip-flop tipo D
module dff(
    input logic D, CK,
    output logic Q
);
    always_ff @(posedge CK) Q <= D;
endmodule
```



```
module circuito1(
    input logic D1, CK1,
    input logic CK2, D, CK,
    output logic Q2, Q
);
    // Sinal interno
    logic Q1;

    // Por ordem
    dff FF1(D1, CK1, Q1);

    // Explícita por nomes
    dff FF2(.CK(CK2),
            .D(Q1),
            .Q(Q2));

    // Por casamento de nomes
    dff FF3(.*);
endmodule
```



6.3 Módulos Genéricos ou Parametrizados

Alguns tipos de circuitos lógicos possuem um comportamento genérico similar, embora diferente apenas por alguns parâmetros operacionais. Exemplos deste tipo de circuitos são multiplexadores, demultiplexadores e memórias. Considere o multiplexador do exemplo abaixo.

```
module mux8(
    input  logic [7:0] X,
    input  logic [2:0] SEL,
    output logic       Y
);
    always_comb
        Y = X[SEL];
endmodule
```

Se quisermos criar um novo multiplexador para 64 entradas seria suficiente modificar a entrada para `[63:0] X` e o seletor para `[5:0] SEL`, sem qualquer alteração da lógica de funcionamento do circuito. Criar um novo módulo somente para esta pequena alteração seria desperdício de esforço, além de complicar a manutenção do código posteriormente. Uma alternativa seria tornar o módulo muxN parametrizado, se

forma que ele possa descrever o comportamento de um multiplexador com N seletores e 2^N entradas.

A declaração de um módulo parametrizado é muito semelhante a de um módulo simples, como pode ser visto abaixo. A lista de parâmetros do módulo é declarada logo antes da lista de portas usando a notação `#( parameter NOME = VALOR, ... )`, onde `NOME` é o nome do parâmetro e `VALOR` é o valor padrão deste parâmetro.

```
module muxN
#(
    parameter N = 3
)
(
    input  logic [2**N-1 : 0] X,
    input  logic [   N-1 : 0] SEL,
    output logic              Y
);
    always_comb
        Y = X[SEL];
endmodule
```

Durante a instanciação do módulo é possível alterar o valor dos parâmetros como mostrado abaixo.

```
muxN        mux8_1 (X1, SEL1, Y1); // N=3, valor padrão
muxN #(6)   mux63_1(X2, SEL2, Y2); // N=6, 64 entradas
muxN #(N=4) mux16_1(X3, SEL3, Y3); // N=4, 16 entradas
```

7 Exemplos

7.1 Memória RAM

Este exemplo mostra como criar uma pequena memória RAM com 8 posições de 4 bits cada.

```
module ram_8x4(
    input logic read, write,
    input logic [2:0] address,
    inout logic [3:0] data
);
    // Variável local para armazenamento dos dados
    logic [3:0] RAM [0:7];

    // Lógica de escrita em memória, na subida de write...
    always_ff @ (posedge write)
    begin
        // ... copiar o valor do barramento para a memória.
        RAM[address] <= data;
    end

    // Lógica de leitura de memória
    always_comb
    begin
        if (read)
        begin
            // Leitura em progresso, mostrar valor no barramento.
            data = RAM[address];
        end
        else
        begin
            // Leitura desativada, barramento em alta-impedância.
            data = 4'bZZZZ;
        end
    end

endmodule
```

7.2 Multiplexador 64:1

```
module mux64(
    input  logic [63:0] X,
    input  logic  [5:0] SEL,
    output logic        Y
);
    always_comb
        Y = X[SEL];
endmodule
```

7.3 Decodificador 8:256

```
module decoder8_256(
    input  logic         ENABLE,
    input  logic   [7:0] SEL,
    output logic [255:0] Y
);
    always_comb
        if (ENABLE)
            Y = 256'b1 << SEL;
        else
            Y = 0;
endmodule
```

7.4 Codificador 8:3 Com Prioridade

```
module priority_encoder_8_3(
    input  logic [7:0] X,
    output logic [2:0] Y,
    output logic       GATE
);
    always_comb casex (X)
        8'b1xxxxxxx: Y=7;
        8'b01xxxxxx: Y=6;
        8'b001xxxxx: Y=5;
        8'b0001xxxx: Y=4;
        8'b00001xxx: Y=3;
        8'b000001xx: Y=2;
        8'b0000001x: Y=1;
        default:     Y=0;
    endcase
    always_comb GATE = |X;
endmodule
```