UNIVERSIDADE FEDERAL DE SÃO PAULO ICT–UNIFESP
AVENIDA CESARE MANSUETO GIULO LATTES, Nº 1201. CEP: 12247-014

José Jailson Santos Craibas

Prof. Dr. Tiago de Oliveira
Prof. Dr. Sérgio Ronaldo Barros

São José dos Campos - SP
Abril de 2018

LABORATÓRIO DE SISTEMAS COMPUTACIONAIS

ARQUITETURA DE COMPUTADORES

DICAS DE IMPLEMENTAÇÃO DE CIRCUITOS DIGITAIS EM VERILOG ATRAVÉS DO SOFTWARE QUARTUS PRIME

São José dos Campos - SP
Abril de 2018

SUMÁRIO

Lista de Tabelas

LISTINGS

EXEMPLOS HDL VERILOG

1 Objetivo

Este manual visa à introdução de desenvolvimento de circuitos digitais em Verilog, a qual é uma linguagem de descrição de hardware HDL (*Hardware Description Language*) amplamente utilizada. Ela também pode ser usada para desenvolvimento de circuitos analógicos. Independentemente do tipo de circuito, a sintaxe e os conceitos lógicos permanecem os mesmos. Por mais, devido à implementação do processador ser feita via Verilog, o conhecimento de tipos de variáveis, blocos bloqueantes e não-bloqueantes, circuitos combinacionais e sequências e sincronização são relevantes.

2 Conceitos Básicos da HDL Verilog

Esta seção apresenta comandos básicos da linguagem, os quais podem ser utilizados na implementação do processador via Quartus Prime [2].

2.1 Sintaxe HDL Verilog

A elaboração de um circuito em Verilog inicializa-se com a declaração de um módulo, o qual é uma unidade básica que contém o circuito desejado. A declaração do módulo deve conter a lista de portas da interface:

module nome_do_modulo (conjunto_de_portas);

Em que ***nome_do_modulo*** é o nome do módulo e ***conjunto_de_portas*** é a lista de todas as portas, as quais podem ser *input, output* ou *inout*. Cada uma das portas é seperada por uma vírgula (,). No módulo Exemplo 1, os dados C e E são unidimensionais e os A, B, D são multidimensionais. Além disso, a finalização do módulo é identificada pela palavra reservada ***endmodule***.

```
1  module Exemplo (A, B, C, D, E);
2
3  input [3:0] A, B;
4  input C;
5  inout [7:0] D;
6  output E;
7
8  endmodule
```

Exemplo 1: Declaração de módulos em Verilog

2.1.1 Instanciação de Módulos em Verilog

Usando o Exemplo 1, pode-se criar um novo módulo que contenha diversas instâncias de módulos iguais ou diferentes. No Exemplo 2, o Exemplo 1 é instanciado duas vezes dentro de um único módulo. Portanto, essas instanciações são análogas às chamadas de funções, como, por exemplo, na linguagem de programação C. Contudo, em Verilog, são circuitos que estão sendo encapsulados.

```
1  module Instanciacao (A, B, C, D, E, a, b, c, d, e);
2
3  input [3:0] A, B, a , b;
4  input C, c;
5  inout [7:0] D, d;
6  output E, e;
7
8  Exemplo Exemplo_1 (.A(A), .B(B), .C(C), .D(D), .E(E));
9
10 Exemplo Exemplo_2 (.A(a), .B(b), .C(c), .D(d), .E(e));
11
```

```
12 endmodule
```

Exemplo 2: Declaração de módulos encapsulados

2.2 Comentários em Verilog

Ao desenvolver circuitos em HDL Verilog, o ato de adicionar comentários torna o código mais compreensível tanto ao projetista quanto aos possíveis usuários. Em Verilog há comentários aplicados em uma única linha ou em múltiplas. // é o símbolo de inicialização para uma única linha, já para múltiplas linhas, inicia-se com o símbolo /* e finaliza-se com */. Segue abaixo um exemplo:

// Este é um comentário aplicado a uma única linha em Verilog
/ Este é um comentário aplicado a múltiplas linhas em Verilog.*
Note que o comentário é inicializado com o símbolo "/"e finalizado com "*/". */*

2.3 Números

2.3.1 Números Inteiros

Números inteiros em Verilog são declarados através da palavra reservada **integer** e o valor pode ser negativo. O Exemplo 4 demonstra algumas atribuições.

```
1  module NumeroInteiro ();
2
3  integer i,j,k;
4
5  initial
6  begin
7          i = 150;
8          j = -150;
9          k = -32;
10 end
11
12 endmodule
```

Exemplo 3: Módulo com atribuições de valores inteiros

2.3.2 Números Base

Números base são números inteiros que são declarados através de uma determinada base numérica. Eles são declarados da seguinte forma:
$< numero\_inteiro > = < quantidade\_de\_bits >' < numero\_base >< valor >;$
Em que:

- $< numero\_inteiro >$ é o nome do inteiro;
- $< quantidade\_de\_bits >$ é a quantidade de bits que representa o número inteiro;
- $< numero\_base >$ é a base numérica, a qual pode ser o (octal); h (hexadecimal); d (decimal) ou b (binário);
- $< valor >$ é o valor na base especificada;

O Exemplo 5 demonstra algumas possibilidades de atribuição utilizando o recurso.

```
module NumerosBase ();
integer i,j,k,l;
initial
begin
        i = 5'b10111; // Numero binario
        j = 5'o24; // Numero octal
        k = 8'ha9; // Numero hexadecimal
        l = 5'd24; // Numero decimal
end
endmodule
```

Exemplo 4: Módulo com atribuições em bases numéricas diferentes

2.4 Tipos de Dados

Verilog permite a manipulação de dois tipos de dados, ***reg*** e ***net***. Reg (abreviação de Registrador) é um dado de armazenamento. Ele permite que o valor atribuído seja mantido até que outra atribuição seja efetuada. Por mais, é comumente usado em blocos ***always*** e ***initial***. O tipo mais comum de ***net*** usado em Verilog é o ***wire***. Ele é análogo a um fio físico em *hardware*, sendo assim, a informação contida numa variável ***wire*** é atualizada constantemente. Além disso, se na declaração de *input* ou *output* não for especificado o tipo de dado, o compilador do Quartus Prime o define como ***wire***.

O tipo ***reg*** também pode indicar a sintetização de elementos sem memória, similar à declaração de elementos ***wire***. Portanto, a diferenciação é feita com base na implementação do construtor ***always*** do projetista, o qual pode ser combinacional ou sequencial. Ou seja, se o dado declarado for do tipo ***reg*** e sua atribuição ocorre no bloco ***always***, o qual possui parâmetros ***posedge*** ou ***negedge***, certamente, o dado é de armazenamento. Se o construtor ***always*** funciona como um ***latch*** devido a alguma variável de controle que só é alterada em ciclos de ***clock*** dispersos, o valor do dado pode ser mantido até que ocorra outra atribuição à variável de controle. Por fim, se as atribuições internas ocorrem sempre, os dados ***reg*** são sintetizados como ***wire***.

2.5 Laços e Desvios

A sintaxe e a semântica de laços e desvios são similares à lingugem de programação C. Exceto pela exigência de utilização em blocos ***always***. Para desvios ***if*** e ***else***, quando houver mais de uma atribuição, devem ser inicializados com a palavra reservada ***begin*** e finalizados com ***end***. Já o desvio ***case*** é iniciado com a palavra reservada ***case*** e finalizado com ***endcase***. O Exemplo 6 apresenta um módulo que utiliza o laço ***for*** e os demais casos citados acima.

```
module LacosEdesvios(a, b, c, cotrole, reset);

input reset;
input [3:0] a, b, c;
input [2:0] controle;
output reg [3:0] x, y;
integer i;

reg [31:0] dado [31:0];

/* O simbolo "*" e usado para indicar que as atribuicoes internas serao executadas quando houver qualquer alteracao nas
      entradas do bloco. */
always @(*) begin

        if(reset==1) begin
                for(i=0 ; i < 32; i=i + 1) begin
                                dado[i] <= 32'd0;
                end
        end

        else begin
                case(controle);

                        0: begin
                                x = a + b;
```

```
25                                      y = a + c;
26                              end
27
28                              1: begin
29                                      x = 2*a;
30                                      y = c - 10;
31                              end
32
33                              default: begin
34                                      x = 1;
35                                      y = 1;
36                              end
37
38                      endcase
39              end
40      end
41
42      endmodule
```

Exemplo 5: Laço *for* e desvios

3 Operadores

Tabela 1: Operadores HDL Verilog [1]

Operador Verilog	Nome	Grupo Funcional
[ ]	seleção de bit ou parcela de vetor	
( )	parênteses	
!	negação lógica	lógico
~	negação	bit-wise
&	redução AND	redução
|	redução OR	redução
~&	redução NAND	redução
~ |	redução NOR	redução
^	redução XOR	redução
~ ^ ou ^ ~	redução XNOR	redução
+	adição	aritmético
-	subtração	aritmético
*	multiplicação	aritmético
/	divisão	aritmético
%	resto	aritmético
{ }	concatenação	concatenação
{{ }}	replicação	replicação
>	maior que	relacional
>=	maior ou igual a	relacional
<	menor que	relacional
<=	menor ou igual a	relacional
<<	deslocamento à esquerda	relacional
>>	deslocamento à direita	relacional
==	igualdade de caso	igualdade
! =	desigualdade de caso	igualdade
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	AND	lógico
||	OR	lógico
? :	condicional	condicional

Conforme a Tabela 1, a HDL Verilog oferece uma série de operadores e, portanto, nesta seção serão dados exemplos de como usá-los [1].

3.1 Aritméticos

Há cinco operadores aritméticos em Verilog, os quais são utilizados entre dois operandos que podem ser unidimensionais ou multidimensionais. O Exemplo 7 mostra um módulo que utiliza os operadores citados.

```
1
2  module Aritmetico (A, B, Y1, Y2, Y3, Y4, Y5);
```

```
3
4                   input [2:0] A, B;
5                   output [3:0] Y1;
6                   output [4:0] Y3;
7                   output [2:0] Y2, Y4, Y5;
8                   reg [3:0] Y1;
9                   reg [4:0] Y3;
10                  reg [2:0] Y2, Y4, Y5;
11
12                  always @(A or B)
13                  begin
14                                  Y1=A+B; //adicao
15                                  Y2=A-B; //subtracco
16                                  Y3=A*B; //multiplicacao
17                                  Y4=A/B; //divisao
18                                  Y5=A%B; //resto de A dividido por B
19                  end
20  endmodule
```

Exemplo 6: Operadores aritméticos

3.2 Sinais

Estes operadores definem um sinal positivo '+' ou negativo '-' para os operandos de uma determinada operação. Usualmente nenhum operador de sinal é definido, sendo assim, o caso default positivo '+' é usado. O Exemplo 8 demonstra algumas atribuições que usam sinais.

```
1
2   module Sinal (A, B, Y1, Y2, Y3);
3
4                   input [2:0] A, B;
5                   output [3:0] Y1, Y2, Y3;
6                   reg [3:0] Y1, Y2, Y3;
7
8                   always @(A or B)
9                   begin
10                                          // sinal atribuido a cada um dos operandos
11                                  Y1=+A/-B;
12                                  Y2=-A+-B;
13                                  Y3=A*-B;
14                  end
15  endmodule
```

Exemplo 7: Atribuindo sinais aos operandos

3.3 Relacionais

Os operadores relacionais comparam dois operandos e retornam um indicador de quando a relação comparada é verdadeira ou falsa. O resultado da comparação é 0 ou 1. Se a comparação for falsa retorna 0. Caso contrário, retorna 1. O Exemplo 9 atribui algumas variáveis usando operadores relacionais.

```
1
2   module Relacional (A, B, Y1, Y2, Y3, Y4);
3
4                   input [2:0] A, B;
5                   output Y1, Y2, Y3, Y4;
6                   reg Y1, Y2, Y3, Y4;
7
8                   always @(A or B)
9                   begin
10                                  Y1=A<B;   //menor que
11                                  Y2=A<=B;  //menor que ou igual a
12                                  Y3=A>B;   //maior que
13                                  if (A>B)  //obrigatorio uso de parenteses
14                                                  Y4=1; // caso verdadeiro
15                                  else
16                                                  Y4=0; // caso falso
17                  end
18  endmodule
```

Exemplo 8: Atribuição com operadores relacionais

3.4 Igualdade e Desigualdade

Os operadores de igualdade e desigualdade são usados da mesma forma que os relacionais e retornam o indicador 0 para o caso falso e 1 para o verdadeiro. O Exemplo 10 atribui valor a algumas variáveis usando-os.

```
1
2  module Igualdade (A, B, Y1, Y2, Y3);
3
4                  input [2:0] A, B;
5                  output Y1, Y2;
6                  output [2:0] Y3;
7                  reg Y1, Y2;
8                  reg [2:0] Y3;
9
10                 always @(A or B)
11                 begin
12                                 Y1=A==B;  //Y1=1 se A e equivalente a B
13                                 Y2=A!=B;  //Y2=1 se A neo e equivalente a B
14                                 if (A==B) //obrigatorio uso de parenteses
15                                                 Y3=A;
16                                 else
17                                                 Y3=B;
18                 end
19 endmodule
```

Exemplo 9: Atribuições usando operadores de igualdade e desigualdade

3.5 Lógicos

Operadores de comparação lógica são usados conjuntamente com os operadores relacionais e de igualdade. Eles possibilitam múltiplas comparações em uma única expressão. O Exemplo 11 mostra alguns casos de uso.

```
1
2  module Logico (A, B, C, D, E, F, Y);
3
4                  input [2:0] A, B, C, D, E, F;
5                  output Y;
6                  reg Y;
7
8                  always @(A or B or C or D or E or F)
9                  begin
10                                 // Operadores de igualdade, relacionais e logicos
11                                 if ((A==B) && ((C>D) || !(E<F)))
12                                                 Y=1;
13                                 else
14                                                 Y=0;
15                 end
16 endmodule
```

Exemplo 10: Operadores lógicos usados no desvio *if*

3.6 Bit-wise

Os operadores lógicos binários (bit-wise) relacionam dois ou múltiplos operandos em cada lado e retorna um único bit como resultado. A única exceção é o operador NOT, o qual só nega o operando que segue após sua declaração. A HDL Verilog não possui operadores para NAND ou NOR, eles podem ser implementados através da negação dos operadores AND e OR, respectivamente. O Exemplo 12 apresenta diversos casos de atribuições usando-os.

```
1
2  module Bitwise (A, B, Y);
3
4                  input [6:0] A;
5                  input [5:0] B;
6                  output [6:0] Y;
7                  reg [6:0] Y;
8
9                  always @(A or B)
```

```
10                  begin
11                                  Y(0)=A(0)&B(0);     //binario AND
12                                  Y(1)=A(1)|B(1);     //binario OR
13                                  Y(2)=!(A(2)&B(2));  //negacao AND
14                                  Y(3)=!(A(3)|B(3));  //negacao OR
15                                  Y(4)=A(4)^B(4);     //binario XOR
16                                  Y(5)=A(5)~^B(5);    //binario XNOR
17                                  Y(6)=!A(6);         //negacao unitaria
18                  end
19  endmodule
```

Exemplo 11: Atribuições de resultados de operadores binários

3.7 Deslocamentos

Os operadores de deslocamento lógico requerem dois operandos. O operando anterior ao operador contém o dado que será deslocado e o operando posterior contém o número de bits que serão deslocados. o valor 0 é usado para preencher as lacunas deslocadas. O Exemplo 13 apresenta os casos de deslocamentos à esquerda e à direita de três bits.

```
1
2   module Deslocamento (A, Y1, Y2);
3
4                   input [7:0] A;
5                   output [7:0] Y1, Y2;
6                   reg [7:0] Y1, Y2;
7                   parameter B=3;
8
9                   always @(A)
10                  begin
11                                  Y1=A<<B; //deslocamento logico a esquerda
12                                  Y2=A>>B; //deslocamento logico a direita
13                  end
14  endmodule
```

Exemplo 12: Deslocamentos à esquerda e à direita

3.8 Concatenação e Replicação

O operador de concatenação { , } concatena os bits de dois ou mais dados. Os dados podem ser unidimensinais (um único bit) ou multidimensinais (múltiplos bits). Múltiplas concatenações podem ser executadas com uma constante como prefixo e é conhecido como operador de replicação {constante{ }}. O módulo do Exemplo 14 usa desses dois operadores para concatenar dados multidimensionais (vetores de bits).

```
1
2   module Concatecao (A, B, Y);
3
4                   input [2:0] A, B;
5                   output [14:0] Y;
6                   parameter C=3'b011;
7                   reg [14:0] Y;
8
9                   always @(A or B)
10                  begin
11                  /* Concatena os valores de A, B conjuntamente com a replicacao
12                  do parametro C e os 3 bits seguintes 3'b110. */
13                                  Y={A, B, {2{C}}, 3'b110};
14                  end
15  endmodule
```

Exemplo 13: Concatenação com replicação de dados multidimensionais

3.9 Reduções

Verilog possui seis operadores de redução, os quais aceitam um único operando vetor (múltiplos bits). Cada um deles executa uma determinada redução bit-wise (único bit) sobre todos os bits do operando, e retorna um único bit como resultado. No Exemplo 15,

o operando A de quatro bits é usado pelo operador de redução AND, o qual opera sobre todos os bits de A e produz Y1.

```
1
2  module Reducao (A, Y1, Y2, Y3, Y4, Y5, Y6);
3
4                   input [3:0] A;
5                   output Y1, Y2, Y3, Y4, Y5, Y6;
6                   reg Y1, Y2, Y3, Y4, Y5, Y6;
7
8                   always @(A)
9                   begin
10                                  Y1=&A; //reducao AND
11                                  Y2=|A; //reducao OR
12                                  Y3=~&A; //reducao NAND
13                                  Y4=~|A; //reducao NOR
14                                  Y5=^A; //reducao XOR
15                                  Y6=~^A; //reducao XNOR
16                  end
17 endmodule
```

Exemplo 14: Reduções sobre dados multidimensionais

3.10 Condicional

Uma expressão que usa o operador condicional executa a expressão lógica anterior ao símbolo de interrogação '?'. Se a expressão é verdadeira, então a expressão anterior ao símbolo dois pontos ':' é o resultado; caso contrário, a expressão posterior ao símbolo dois pontos ':' é definida como resultado. O Exemplo 16 utiliza deste operador para realizar uma atribuição no interior do bloco ***always***.

```
1
2  module Condicional (Time, Y);
3
4                   input [2:0] Time;
5                   output [2:0] Y;
6                   reg [2:0] Y;
7                   parameter Zero =3b'000;
8                   parameter TimeOut = 3b'110;
9
10                  always @(Time)
11                  begin
12                                          /* Se a entrada 'Time' for diferente do parametro 'TimeOut',
13                                             entao Y recebera 'Time+1'; caso contrario, Y recebera 'Zero'. */
14                              Y=(Time!=TimeOut) ? Time + 1 : Zero;
15                  end
16 endmodule
```

Exemplo 15: Atribuição de variáveis usando o operador condicional

4 Tipos de Circuitos

4.1 Circuitos Combinacionais

Os circuitos combinacionais são caracterizados pela atribuição de sinais em função das entradas atuais do sistema. Ou seja, a determinação da saída é efetuada unicamente através das entradas. No Exemplo 17 os registradores a e b são modificados sempre que houver alterações nas entradas val1 e val2.

```
1  module combinacional (val1, val2, a, b);
2
3  input val1, val2;
4  output reg a, b;
5
6  // As atribuicoes de a e b ocorrem sempre que ha alguma alteracao das entradas val1 e val2
7  always @(val1 or val2) begin
8
9                  a = val1;
10                 b = val2;
11 end
```

```
12
13  endmodule
```

Exemplo 16: Circuito combinacional

4.2 Circuitos Sequenciais

Os circuitos sequenciais são formados por circuitos combinacionais e elementos de memória (*flip-flops*). Sendo assim, a definição da saída do sistema depende tanto das entradas do sistema como do estado atual do sistema. Em verilog, eles são implementados através do construtor ***always*** conjuntamente às palavras reservadas ***posedge*** e ***negedge***. No primeiro caso, a saída do sistema só é atualizada na transição $0 -> 1$ do relógio do sistema (***clock***), ou seja, enquanto a transição não ocorre, o estado anterior do sistema é mantido. Já no segundo caso, a atualização só ocorre na transição $1 -> 0$ do (***clock***). No Exemplo 18, os registradores a e b só são atualizados na transição $1 -> 0$ do (***clock***).

```
1   module sequencial (val1, val2, a, b, clock);
2
3   input val1, val2;
4   output reg a, b;
5
6   // As atribuicoes de a e b so ocorrem na transicao de 0 para 1 do sinal clock
7   always @(negedge clock) begin
8
9                   a <= val1;
10                  b <= val2;
11  end
12
13  endmodule
```

Exemplo 17: Circuito sequencial

4.3 Atribuições

4.3.1 Diferenças Entre Atribuições Contínuas e Procedurais

Tabela 2: Diferenças entre atribuições contínuas e procedurais [3]

Atribuição Contínua	**Atribuição Procedural**
Atribui valor a dados do tipo ***nets (wire)***	Atribui valor a dados do tipo ***reg***
Variáveis e ***nets (wire)*** atribuem valores às portas continuamente	Resultados de operações envolvendo variáveis e ***nets (wire)*** podem ser armazenados em variáveis
Usado para inferir lógica combinacional	Usado para inferir elementos de armazenamento como ***flip-flops*** e ***latches*** e também lógica combinacional
Atribuições ocorrem de forma contínua	Valor atribuído anteriormente à variável é mantido até que outra atribuição seja efetuada
Ocorrem em atribuições para os tipos ***wire***, ***port*** e ***net***	Ocorre em construtores ***always*** e ***initial***

Exemplo:
wire saida = entrada1 & entrada2;
ou
assign saida = entrada1 & entrada2
	Exemplo:
	always @(posedge clock)
	reg <= entrada;
	ou
	always @(a or b or s)
	y = (s == 1) ? a : b;

4.3.2 Diferenças Entre Atribuições nos Construtores *initial* e *always*

Atribuições contidas nos construtores ***initial*** e ***always*** são ambas procedurais e diferem no tempo em que são executadas durante a simulação. Enquanto o primeiro só é executado até a última instrução do bloco, ou seja, só executa uma vez, o segundo executa indefinidamente.

4.3.3 Diferenças Entre Atribuições Bloqueantes e Não-Bloqueantes

Atribuições bloqueantes e não-bloqueantes são ambas atribuições procedurais e diferem no comportamento lógico e de sintetização.

Tabela 3: Diferenças entre atribuições bloqueantes e não-bloqueantes [3]

Atribuição Bloqueante	**Atribuição Não-Bloqueante**
Quando múltiplas atribuições bloqueantes estão sendo executadas, a atribuição posterior é bloqueada até que a anterior seja efetuada, ou seja, as atribuições são feitas em sequência	Múltiplas atribuições não-bloqueantes podem ser executadas paralelamente no próximo ciclo de execução da simulação
Recomenda-se usar em blocos combinacionais do construtor ***always***	Recomenda-se usar em blocos sequênciais do construtor ***always***
Pode ser usada em atribuições procedurais como ***initial***, ***always*** e em atribuições contínuas para tipo ***net (wire)*** como nas declarações de ***assign***	Só pode ser usada em atribuições procedurais como ***initial*** e ***always***. Atribuições contínuas como nas declarações de ***assign*** não são permitas
O sinal do operador é '='	O sinal do operador é '<='

Exemplo:
always @(posedge clock)
a = b;
c = d;
A atribuição do registrador c só ocorrerá após a atribuição do a
	Exemplo:
	always @(posedge clock)
	reg1 <= entrada1;
	reg2 <= entrada2;
	A atribuição do registrador reg1 pode ocorrer depois da atribuição do reg2 ou até mesmo paralelamente

4.4 *Reset* Síncrono e Assíncrono

4.4.1 Reset Síncrono

No ***reset*** síncrono, a reinicialização do sistema só ocorre quando o sinal de ***reset*** estiver ativo e a transição $0 -> 1$ (***posedge***) ou $1 -> 0$ (***negedge***) do ***clock*** ocorrer. No Exemplo 19, há um exemplo básico de inicialização ou reinicialização do circuito sequencial, nele os registradores a e b são setados em 0 quando o sinal ***reset*** está ativo conjuntamente a uma transição $0 -> 1$ (***posedge***) do ***clock***.

```
1  module ResetSincrono (val1, val2, a, b, clock, reset);
2
3  input val1, val2, clock, reset;
4  output reg a, b;
5
6  always @(posedge clock) begin
7
8          if(reset==1) begin
9                  a <= 0;
10                 b <= 0;
11         end
12         else begin
13                 a <= val1;
14                 b <= val2;
15         end
```

```
16  end
17
18  endmodule
```

Exemplo 18: Inicialização ou reinicialização síncrona do circuito

4.4.2 Reset Assíncrono

Na reinicialização assíncrona, o sinal de ***reset*** é independente do sinal de ***clock***. Portanto, quando ocorre a transição $0 -> 1$ (***posedge***) ou $1 -> 0$ (***negedge***) do ***reset***, o circuito será reinicializado. No Exemplo 20, há um exemplo básico de inicialização ou reinicialização do circuito sequencial, nele os registradores a e b são setados em 0 quando o sinal ***reset*** transita de $1 -> 0$ (***negedge***), independentemente da transição do ***clock***.

```
1   module ResetAssincrono (val1, val2, a, b, clock, reset);
2
3   input val1, val2, clock, reset;
4   output reg a, b;
5
6   always @(posedge clock or negedge reset) begin
7
8           if(!reset) begin
9                   a <= 0;
10                  b <= 0;
11          end
12          else begin
13                  a <= val1;
14                  b <= val2;
15          end
16  end
17
18  endmodule
```

Exemplo 19: Inicialização ou reinicialização assíncrona do circuito

5 Templates Para Memórias RAM e ROM

Comumente, os elementos ***reg*** são usados para implementar memórias RAM, conforme o Exemplo 21. Nele, são criadas 32 unidades de memória com 32 bits, as quais são acessadas de acordo com o código Verilog a ser descrito pelo projetista. Dependendo do código descrito e do tamanho dessa memória, o software Quartus Prime pode gerar muitos elementos lógicos para mapeá-las no FPGA, aumentando consideravelmente o tempo de compilação do projeto.

Portanto, recomenda-se utilizar os templates disponibilizados no próprio software Quartus Prime para a implementação de memórias RAM e ROM, o que permitirá ao Quartus Prime inferir módulos de memória interna ao invés de elementos lógicos, otimizando o tempo de compilação. A seguir, apresenta-se o procedimento utilizado no software Quartus Prime para a adição de memórias RAM e ROM utilizando templates.

```
1  module ExemploMemroria();
2
3  parameter tamanho = 32;
4
5  // Declaracao de 32 registradores de 32 bits
6  reg [tamanho-1:0] Registers[tamanho-1:0];
7
8  endmodule
```

Exemplo 20: Modelo básico de memória usando dados do tipo reg

A Figura 1 apresenta a adição de um novo arquivo HDL Verilog no ambiente da ferramenta Quartus II. O PASSO 1 (*new*) possibilita a criação de um novo arquivo e o PASSO 2 (*Verilog HDL File*) especifica o tipo do novo arquivo.

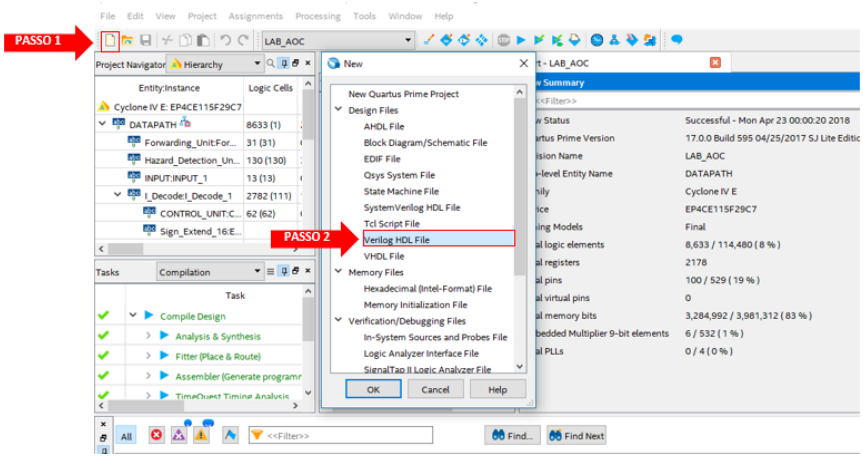

Figura 1: Adiciona novo arquivo do tipo HDL Verilog no Quartus II

Após adicionar o novo arquivo através dos passos da Figura 1, edite-o através da opção *edit* e selecione *Insert Template* conforme os PASSOS 3 e 4, respectivamente, da Figura 2.

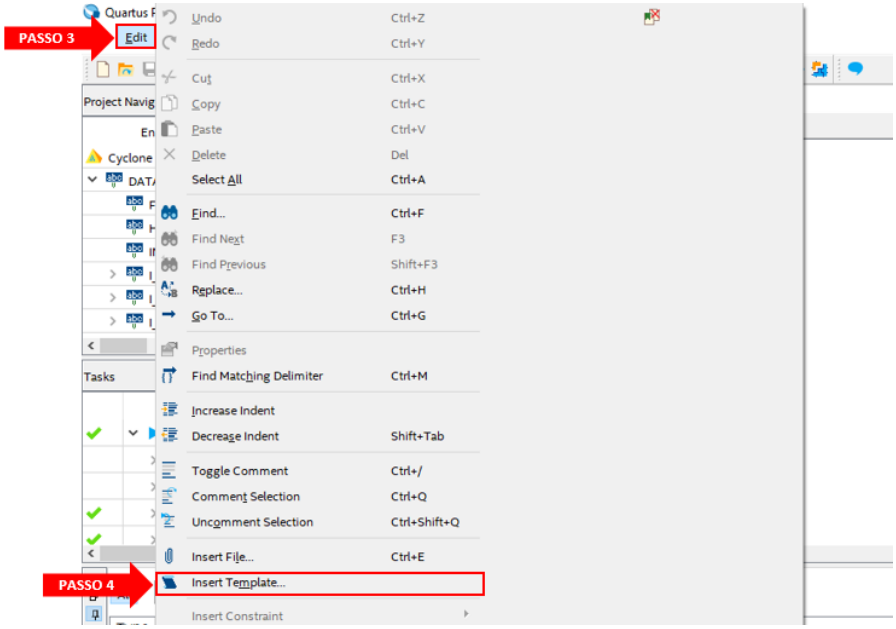

Figura 2: Seleciona novo template a ser adicionado ao arquivo HDL Verilog do Quartus II

A Figura 3 apresenta a tela posterior ao PASSO 4 da Figura 2. Nessa etapa é definida a linguagem do template.

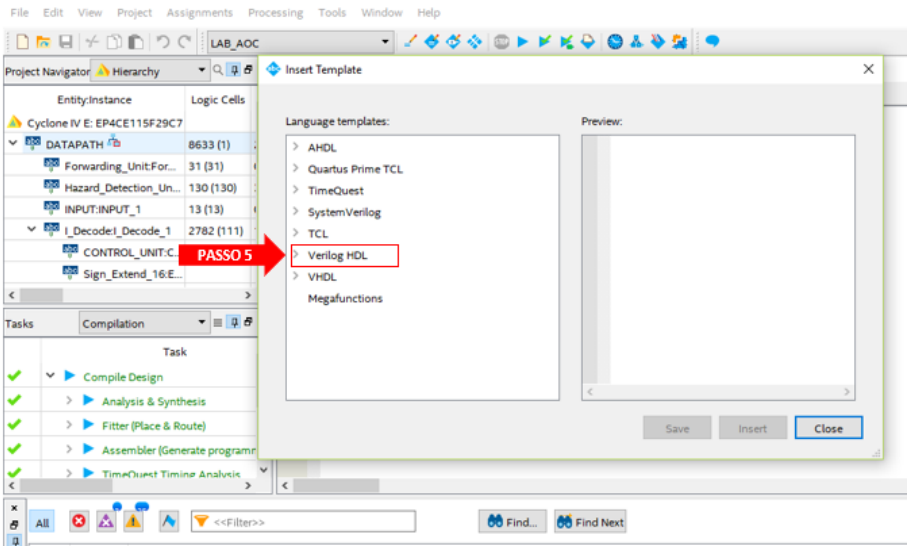

Figura 3: Seleciona linguagem do template a ser adicionado Quartus II

No próximo passo, Figura 4, é selecionada a opção de templates para RAM (*Random Access Memory*) e ROM(*Read Only Memory*).

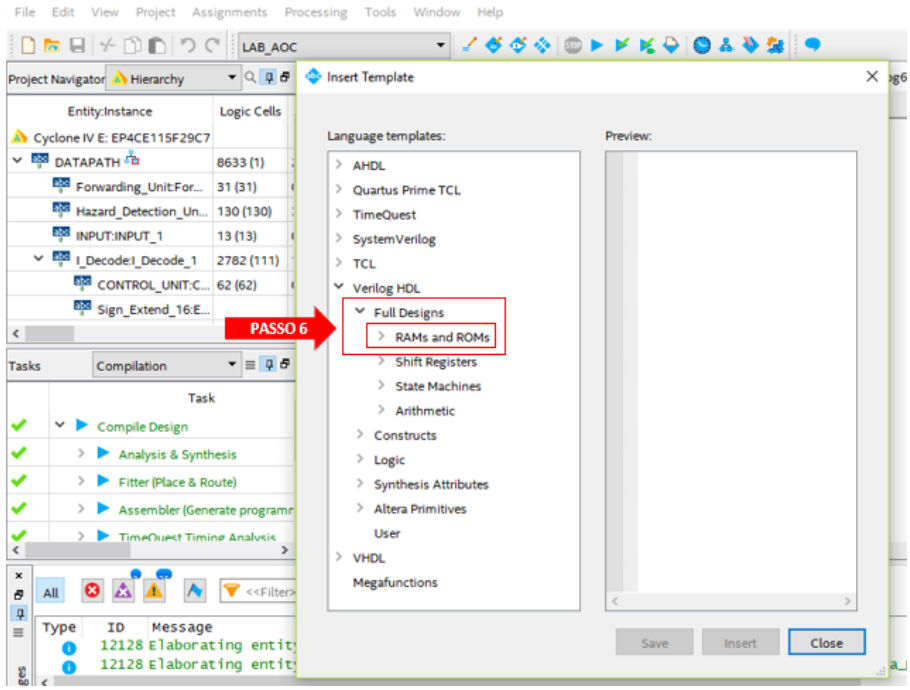

Figura 4: Seleciona templates para RAM e ROM

Na Figura 5, há diversas opções de templates para memórias RAM e ROM, contudo, neste manual são exemplificados os templates RAM com clock duplo (*Simple Dual Port RAM (dual clock)*) e o ROM com porta única (*Single Port ROM*).

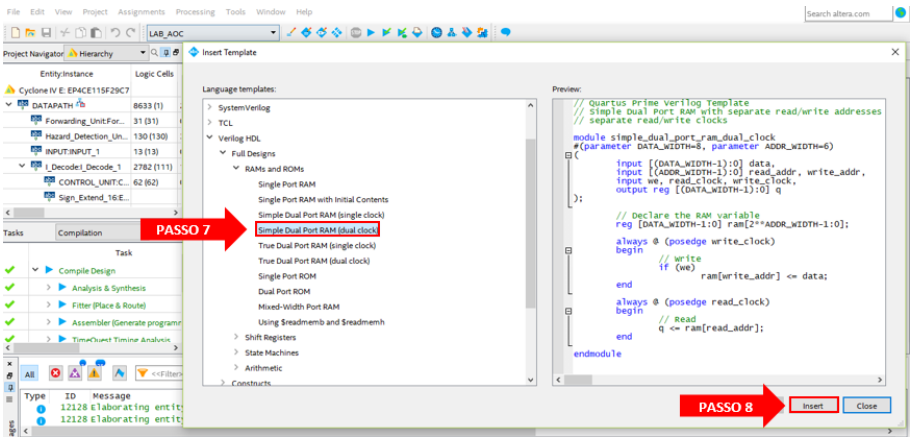

Figura 5: Adição de template para RAM com clock duplo

Após inserir o template via PASSO 8 da Figura 5, o código do Exemplo 21 é inserido no arquivo HDL Verilog criado. Nele, pode-se alterar o nome do módulo HDL Verilog (*simple_dual_port_ram_dual_clock*); o número de bits do dado *DATA_WIDTH* e o tamanho do endereço de escrita e leitura *ADDR_WIDTH*. Por mais, o tamanho da memória é definido como $2^{ADDR\_WIDTH} - 1$ e a escrita e a leitura de dados são feitas em ***clocks*** distintos.

```
1  // Quartus Prime Verilog Template
2  // Simple Dual Port RAM with separate read/write addresses and
3  // separate read/write clocks
4
5  module simple_dual_port_ram_dual_clock
6  #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
7  (
8          input [(DATA_WIDTH-1):0] data,
9          input [(ADDR_WIDTH-1):0] read_addr, write_addr,
10         input we, read_clock, write_clock,
11         output reg [(DATA_WIDTH-1):0] q
12 );
13
14         // Declare the RAM variable
15         reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
16
17         always @ (posedge write_clock)
18         begin
19                 // Write
20                 if (we)
21                         ram[write_addr] <= data;
22         end
23
24         always @ (posedge read_clock)
25         begin
26                 // Read
27                 q <= ram[read_addr];
28         end
29
30 endmodule
```

Exemplo 21: Template RAM com clock duplo

A Figura 6 e o Exemplo 22 apresentam os mesmos passos da Figura 5 e do Exemplo 21. Exceto pelo template adicionado simular uma memória ROM com um único clock e uma única porta. Nesse caso, a inicialização da memória é feita por um arquivo *'single_port_rom_init.txt'*, o qual pode ser renomeado e deve ser adicionado no diretório em que o arquivo do módulo foi criado. Além disso, os dados do arquivo de inicialização devem estar na base binária.

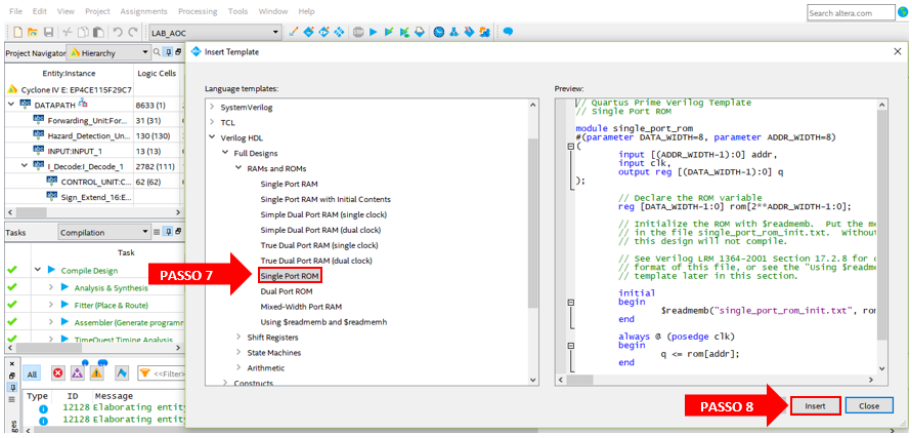

Figura 6: Template ROM HDL Verilog

```
1  // Quartus Prime Verilog Template
2  // Single Port ROM
3
4  module single_port_rom
5  #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
6  (
7          input [(ADDR_WIDTH-1):0] addr,
8          input clk,
9          output reg [(DATA_WIDTH-1):0] q
10 );
11
12         // Declare the ROM variable
13         reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
14
15         // Initialize the ROM with $readmemb.  Put the memory contents
16         // in the file single_port_rom_init.txt.  Without this file,
17         // this design will not compile.
18
19         // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
20         // format of this file, or see the "Using $readmemb and $readmemh"
21         // template later in this section.
22
23         initial
24         begin
25                 $readmemb("single_port_rom_init.txt", rom);
26         end
27
28         always @ (posedge clk)
29         begin
30                 q <= rom[addr];
31         end
32
33 endmodule
```

Exemplo 22: Template ROM com um único clock e uma única porta

Para finalizar, basta salvar o template adicionado ao projeto conforme indica a Figura 7 e compilá-lo.

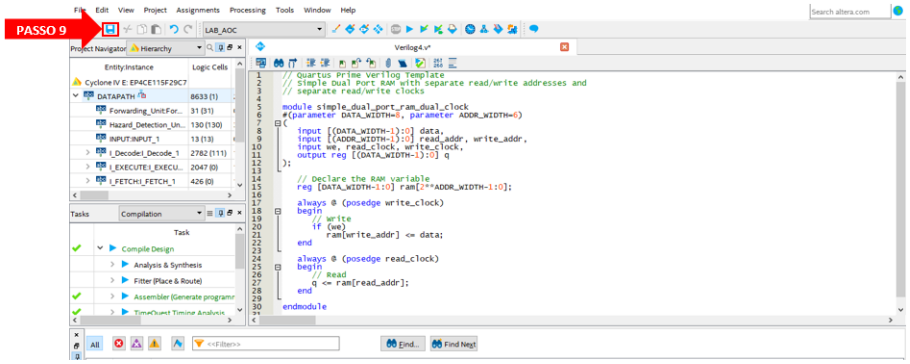

Figura 7: Finalização da inserção do template

5.1 Uso do Operador de Concatenação na Inicialização de Memórias RAM

Nos templates de memória RAM, há a possibilidade de inicializá-los através do construtor ***initial***, no qual é possível usar os operadores vistos na seção 3 deste manual, Tabela 1. O Exemplo 24 apresenta uma aplicação do operador de concatenação.

```
// Quartus Prime Verilog Template
// Simple Dual Port RAM with separate read/write addresses and
// separate read/write clocks

module MEMORY_BIOS
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=8)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] read_addr, write_addr,
        input we, read_clock, write_clock,
        output reg [(DATA_WIDTH-1):0] q
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        initial begin
        ram[0] = {6'd10, 26'd92};                // j 92
        ram[1] = {6'd11 , 5'd0, 5'd0, 16'd0};          // nop R0, R0, 0
        ram[2] = {6'd11 , 5'd0, 5'd0, 16'd0};          // nop R0, R0, 0
        ram[3] = {6'd11 , 5'd0, 5'd0, 16'd0};          // nop R0, R0, 0
        ram[4] = {6'd3 , 5'd24, 5'd31, 16'd0};          // sw R24, R31, 0
        ram[5] = {6'd1 , 5'd0, 5'd30, 16'd5};          // addi R0, R30, 5
        ram[6] = {6'd3 , 5'd24, 5'd30, 16'd1};          // sw R24, R30, 1
        ram[7] = {6'd25 , 5'd0, 5'd0, 16'd0};          // OUTLCD R0, R0, 0
        end

        // Restante do codigo

endmodule
```

Exemplo 23: Aplicação do operador de concatenação para inicialização de templates de memória RAM

6 *Assign* Encadeado

O construtor *assign* é normalmente utilizado em atribuições contínuas e conjuntamente com o operador condicional apresentado na Tabela 1 pode auxiliar na construção de módulos.

O operador condicional pode ser utilizado de forma encadeada, ou seja, a expressão posterior ao símbolo dois pontos ':' pode conter uma nova expressão e assim sucessivamente. No entanto, ao utilizá-lo, deve-se considerar a ordem de precedência das comparações realizadas a fim de que o resultado seja condizente com o esperado. O Exemplo 25 mostra uma atribuição em que o caso falso apresenta uma nova expressão lógica a ser verificada.

```

module AssignCondicional (A, B, C, D, F, sinal1, sinal2, sinal3, sinal4);

input [2:0] B, C, D, F;
input sinal1, sinal2, sinal3, sinal4;
output [2:0] A;

assign A = ((sinal1==0 || sinal2==1) && (sinal3==1 && sinal4==0)) ? B : (sinal3==0) ? D : (sinal4==0) ? F : 3'd0;
```

Exemplo 24: Assign encadeado com o operador condicional

7 CONCLUSÃO

Este manual contém apenas uma introdução ao uso da HDL Verilog como meio de implementação de circuitos. Contudo, apresenta conteúdo suficiente para elaboração de um processador básico. A utilização de sites, tutoriais e livros potencializa o desenvolvimento e a compreensão de circuitos e projetos mais complexos.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] . *Nanometer Design Laboratory. Operadores HDL Verilog.*. Disponível em: $https://www.utdallas.edu/\sim akshay.sridharan/index_files/Page5212.htm$. Acessado em 20/04/2018.

[2] LEE, Weng Fook. Verilog Coding for Logic Synthesis [Chapter Three]. Canada: John Wiley & Sons, 2003.

[3] CHONNAD, Shivakumar & BALACHANDER, Needamangalam. Verilog : Frequently Asked Questions: Language, Applications and Extensions[Chapter one]. Boston: Springer Science + Business Media, Inc., 2004.