Projeto comportamental: declaração *process*

Como o nome diz, no projeto comportamental o objetivo é descrever o *comportamento* do circuito, ao invés de indicar precisamente *como* o circuito funciona. A tarefa de montar o circuito é deixada a cargo do sintetizador. Nesse modelo, o projetista se distancia mais do circuito, e modela utilizando algumas construções de linguagem de alto nível. Apesar de ser muito útil, devemos tomar cuidado com estruturas comportamentais pois nem todos os códigos gerados são *sintetizáveis*, como os comandos "wait" e "after".

I. Exemplo

Vários exemplos dados neste documento são feitos com base no circuito de maioria. O circuito de *maioria* é um circuito no qual a saída é $\mathbf{1}$ apenas quando a maioria de suas entradas é $\mathbf{1}$. Considere o exemplo do circuito de maioria com 3 entradas:

$$\mathbf{F} = \mathbf{A} \cdot \mathbf{B} + \mathbf{A} \cdot \mathbf{C} + \mathbf{B} \cdot \mathbf{C}$$

Uma solução possível para este circuito, em modelo combinacional, é dada por:

Algoritmo 1 Circuito de Maioria de 3 Entradas

```
 1: library IEEE;
 2: use IEEE.STD_LOGIC_1164.ALL;
 3:
 4: entity maioria is
 5:   port ( A,B,C: in STD_LOGIC;
 6:           F: out STD_LOGIC);
 7: end maioria;
 8:
 9: architecture maioria_dataflow_1 of maioria is
10:         signal P1,P2,P3 : STD_LOGIC;
11: begin
12:         P1 <= A and B;
13:         P2 <= A and C;
14:         P3 <= B and C;
15:         F <= P1 or P2 or P3;
16: end maioria_dataflow_1;
```

II. O Modelo Comportamental e a Declaração *process*

O modelo comportamental (*behavioral*) não fornece nenhum detalhe de como implementar o módulo em hardware, ou seja, o código VHDL escrito nesse modelo não reflete necessariamente como o circuito vai ser implementado quando sintetizado. Ao invés disso, o estilo comportamental descreve como a saída do circuito reage (*se comporta*) dadas as entradas do circuito. É trabalho do sintetizador decidir como será a implementação do circuito. O conceito mais importante do modelo comportamental é o *process statement*.

Em VHDL, um *process* é uma coleção de **sequential statements** que executam em paralelo com outros *concurrent statements*. Isto é, dentro de um *process*, as declarações são sequenciais, mas um *process* é concorrente com as outras declarações concorrentes que vimos (*concurrent signal assignment*, *conditional signal assignment*, *selected signal assignment* e inclusive outros *process*).

Utilizando um *process*, podemos especificar uma interação complexa de sinais e eventos de tal forma que ela é executada em "tempo zero" na simulação e que gere um circuito combinacional ou sequencial que execute a operação modelada diretamente.

A. Sintaxe da Declaração de Processo

A sintaxe de um processo é dada por:

label:	**process**	(sensitivity_list)
	begin	
		sequential_statement
		...
		sequential_statement
	end	**process** label;

Nos operadores *concurrent signal assignment* (<=) do modelo combinacional, a qualquer momento que ocorrer uma mudança nos sinais listados do lado direito do operador, essa mudança faz com que o lado esquerdo (isto é, o alvo) da expressão seja re-avaliado. No caso do *process*, a qualquer momento que ocorrer uma mudança em um dos sinais listados na *sensitivity list* do *process*, **todos** os *sequential statements* do *process* são executados. Ou seja, a execução do processo é controlada pelos sinais que são colocados na sua *sensitivity list*.

Um *process* em VHDL tem apenas dois estados: *running* ou *suspended*. Um *process* inicia no estado *suspended*. Ele vai para o estado *running* quando qualquer sinal em sua *sensitivity list* mudar. Quando iniciado, ele executa de maneira sequencial as linhas do *process* até o fim do *process*. Se, durante a execução do *process* algum outro sinal de sua *sensitivity list* mudar, ele será executado novamente. Isto continua até que o *process* seja executado sem que nenhum desses sinais mude (ou seja, até que ele se estabilize).

Outra condição que pode levar um *process* para o estado *suspended* são as condições **wait**. Existem quatro tipos: **wait;**, **wait for time**', **wait on signal** ou **wait until condition**. Quando essa instrução é encontrada, o *process* volta para o estado *suspended* até que a condição do **wait** ocorra (exceto no wait sem condição).

É **muito importante** definir bem a lista de sensibilidade de um processo. Um processo **só é executado** quando um dos sinais na sua lista de sensibilidade muda - ele **não é executado** se algum outro sinal mudar, mesmo que o processo utilize esse outro sinal!

label:	**process**	(signal1, signal2, ..., signalN)
		type declarations
		variable declarations
		constant declarations
		function declarations
		procedure declarations
	begin	
		sequential_statement
		...
		sequential_statement
	end	**process** label;

B. Declarações Sequenciais

Os tipos principais de *sequential statements*:

- *signal and variable assignment*
- *if statement*
- *case statement*
- *loop statement*

C. Signal and Variable Assignement

Dentro de um *process*, podemos usar dois tipos de *assignment operators*. Como todas as declarações dentro de um *process*, ambos são **sequenciais**.

<=	*signal assignemnt*
:=	*variable assignment*

Algoritmo 2 Circuito de Maioria de 3 Entradas

```
1: architecture maioria_behavioral_1 of maioria is
2: begin
3:         process (A,B,C)
4:             variable P1,P2,P3 : STD_LOGIC;
5:         begin
6:             P1 := A and B;
7:             P2 := A and C;
8:             P3 := B and C;
9:             F <= P1 or P2 or P3;
10:        end process;
11: end maioria_behavioral_1;
```

1) VHDL Variables: Dentro de um *process*, não podemos declarar novos sinais, podemos apenas declarar variáveis. Uma variável em VHDL não é visível de fora do *process*, mas elas mantém seu valor quando o *process* não estiver sendo executado (isto é, quando ele estiver no estado *suspended*). Dependendo do seu uso, uma variável pode ou não se tornar um sinal no circuito sintetizado. Enquanto um sinal tipicamente corresponde a um fio no circuito físico, as variáveis são efêmeras. O uso de variáveis não é recomendado em código sintetizável.

A principal diferença é que, dentro de um *process*, a designação de variáveis é executada **imediatamente**, enquanto a designação de sinais é executada **apenas** quando o *process* retorna ao estado *suspended* (isto é, no fim da execução ou quando uma condição de **wait** for encontrada). Logo, se sinais forem usados em condições ou expressões, devemos lembrar desse detalhe, pois isso pode levar a códigos que não funcionam da forma desejada.

D. IF Statement

Outro *sequential statement* deve ser familiar: *IF statement*. Diferentemente do *conditional signal assignment*, as condições não precisam cobrir todas as opções possíveis. Sua sintaxe é:

```
if (condition) then
        sequence of statements
elsif (condition) then
        sequence of statements
else (condition) then
        sequence of statements
end if;
```

Algoritmo 3 Circuito de Maioria de 3 Entradas

```
1: architecture maioria_behavioral_2 of maioria is
2: begin
3:         process(A,B,C)
4:         begin
5:             if ((A = '1') and (B = '1')) then F <= '1';
6:             elsif ((A = '1') and (C = '1')) then F <= '1';
7:             elsif ((B = '1') and (C = '1')) then F <= '1';
8:             else F <= '0';
9:             end if;
10:        end process;
11: end maioria_behavioral_2;
```

E. CASE Statement

O outro *sequential statement* também deve ser familiar: *CASE statement*. Neste caso, similar ao *selected signal assignment*, as escolhas precisam cobrir todas as opções possíveis e devem ser mutuamente excludentes. Sua sintaxe é:

```
case (expression) is
        when choices = >sequential statements
        ...
        when choices = >sequential statements
end case;
```

Algoritmo 4 Circuito de Maioria de 3 Entradas

```
1:  architecture maioria_behavioral_3 of maioria is
2:  begin
3:      process(A,B,C)
4:          variable ABC : STD_LOGIC_VECTOR (2 downto 0);
5:      begin
6:          ABC := A & B & C;
7:          case ABC is
8:              when "011" =>F <= '1';
9:              when "101" =>F <= '1';
10:             when "110" |"111" =>F <= '1';
11:             when others =>F <= '0';
12:         end case;
13:     end process;
14: end maioria_behavioral_3;
```

F. LOOP Statements

O tipo mais simples de estrutura de repetição em VHDL é o *loop statement*, que cria um *loop* infinito. Como é uma estrutura sequencial, ele pode ser usado apenas em *process* e *functions*. A sintaxe é:

```
loop
        sequential statement
        ...
        sequential statement
end     loop;
```

Dois *sequential statements* úteis quando trabalhamos com *loop*s são: *exit* (para sair do *loop*, executando a instrução seguinte ao laço) e *next* (para pular as instruções que faltam e iniciar um novo laço).

1) FOR Statement: Um tipo mais familiar é a estrutura *for*, que cria um laço de repetição finito. Note que a variável do *loop* é criada implicitamente. Como é uma estrutura sequencial, ele pode ser usado apenas em *process* e *functions*. A sintaxe é:

```
for     identifier in range loop
        sequential statement
        ...
        sequential statement
end     loop;
```

2) WHILE Statement: Outro tipo familiar é a estrutura *while*, que cria um laço condicionado. Novamente, como é uma estrutura sequencial, ele pode ser usado apenas em *process* e *functions*. A sintaxe é:

while	expression **loop**
	sequential statement
	...
	sequential statement
end	**loop**;

Algoritmo 5 Circuito de Maioria de 3 Entradas

```
1: architecture maioria_behavioral_4 of maioria is
2: begin
3:         process(A,B,C)
4:             variable ABC : STD_LOGIC_VECTOR (1 to 3);
5:             variable cont: integer;
6:         begin
7:             ABC := A & B & C;
8:             cont := 0;
9:             for i in 1 to 3 loop
10:                if (ABC(i) = '1') then
11:                    cont := cont + 1;
12:                end if;
13:            end loop;
14:
15:            if (cont >= 2) then F <= '1';
16:            else F <= '0';
17:            end if;
18:        end process;
19: end maioria_behavioral_4;
```