Departamento de Engenharia Eletrônica
Escola de Engenharia da UFMG
Disciplina: Sistemas Digitais
Hermes Aguiar Magalhães / Luciano de Errico

ATRIBUIÇÃO DE SINAIS – CONSTRUÇÃO CONCORRENTE *VERSUS* SEQUENCIAL

Atribuímos valores a sinais usando uma sentença ou *"statement"* de atribuição de sinal. Em sua forma mais simples, a atribuição de sinal é da seguinte forma:

Objeto_do_tipo_sinal <= expressão [***after*** valor_de_atraso];

A expressão entre colchetes especifica um atraso de tempo e é opcional. Se for omitida, um atraso (ou "*delay"*) ΔT *default* será assumido. Esta construção usando *"after"* (= após) é utilizada na construção e modelagem de *testbenches* de simulação similares às entradas *"vector waveform files"* (arquivos *.vwf*), que servem para teste de projetos mais complexos. Aqui vamos focar na **síntese** de circuitos lógicos e não na construção de *testbenches*. Especificamente no caso da ferramenta Quartus II, quando na simulação adotamos a modalidade "*timing"*, o atraso *ΔT* é levado em conta automaticamente pelo software. Seu valor é resultante da escolha do chip. Para o EPM3064ALC44-10 [1], este "10" no final do *part-number* indica que neste chip o atraso por porta é de 10 ns.

Código "Comportamental × Dataflow"
ou "Código Sequencial × Orientado a eventos"

Um código VHDL difere radicalmente de uma codificação de software convencional, pois nasce a princípio orientado a eventos. Ou seja, toda sentença tem execução concorrente com as demais sentenças, i.e., suas execuções podem ser "simultâneas" a depender dos eventos que as disparam (conceito de paralelismo), atentando-se para o fato de que uma sentença só é avaliada se algum dos seus membros à direita da atribuição sofrer modificação. Essa forma de codificação em VHDL leva o nome de codificação concorrente ou DATAFLOW. No entanto, é possível abrir "janelas" de codificação sequencial dentro do código concorrente que se comportam de forma similar à codificação de software que estamos familiarizados, onde várias sentenças são avaliadas uma após a outra em sequência. A estas janelas de codificação para avaliação das sentenças sequencialmente denominamos de codificação comportamental. No entanto, há certas peculiaridades no fato de o resultado final ser um circuito (e não um programa de computador), que alteram radicalmente a forma de codificar o projeto.

Em VHDL, código comportamental é de execução sequencial (não confundir com o conceito de circuito sequencial em sistemas digitais, que é outra coisa), programado dentro de construções como "PROCESS", por exemplo. Nele, as sentenças têm uma sequência de execução similar a um programa de computador [2], declaram-se variáveis (keyword "VARIABLE") locais para uso interno ao "PROCESS" e os sinais (criados previamente com a keyword "SIGNAL" ou como itens da interface "ENTITY") são usualmente carregados uma única vez, sendo usados para sinalizar o resultado final do processo sobre os circuitos externos à estrutura. Dentro de um "PROCESS" são válidas as construções sintáticas "IF", "CASE", "LOOP" e "WAIT" [3]. Uma vez que o processamento entrar em um "PROCESS", as sentenças serão avaliadas em sequência e independente dos estímulos (i.e. independente da existência de eventos nos sinais e variáveis do lado direito das atribuições), onde o fluxo do processamento se dará

[1] O Datasheet desse componente está disponível no link: http://www.altera.com/literature/ds/m3000a.pdf.

[2] Atente para o fato de que na síntese de circuitos, você não pode contar com os resultados das operações de atribuição onde o destino é um SIGNAL dentro da estrutura PROCESS. Por serem as atribuições dos sinais sempre *concorrentes*, na hora da codificação podemos "imaginar" na prática que a atualização nas modificações dos sinais somente acontecerão ao final da execução do bloco PROCESS. Esta característica diferencia substancialmente esta programação da programação de computadores convencional, daí a palavra "similar". Para acessar valores intermediários dentro de um PROCESS, é necessário que você os declare como VARIABLE.

[3] Vale lembrar que a variação "WAIT FOR" da construção "WAIT" não é sintetizável, i.e., não pode ser usada para construir circuitos digitais. Ela é usada apenas na construção de ambientes de simulação ou *testbenches*.

baseando-se apenas nas decisões lógicas especificadas nas condições de teste codificadas pelo programador. Daí dizer-se que o código nesse caso é "sequencial".

Já códigos concorrentes estão hierarquicamente "fora" das declarações "PROCESS", e os próprios blocos "PROCESS" são elementos constituintes de sua estrutura. Toda sentença nesse caso tem execução concorrente com as demais sentenças, inclusive com blocos "PROCESS" eventualmente presentes, ou seja, suas execuções são concorrentes e usam construções como "WHEN", "SELECT", GENERATE, etc. Se os eventos nos sinais ocorrerem de forma a acionar tais caminhos concorrentes ao mesmo tempo, as ações de causa e efeito podem ocorrer de forma "simultânea", originando assim o conceito de paralelismo. A necessidade de ligações é suprida pela declaração "SIGNAL" (e não por "VARIABLE" como no caso comportamental). Note que a declaração de um "SIGNAL" ou de uma "VARIABLE" não equivale à criação de uma "variável" como a conhecemos da programação de computadores (naquele caso a declaração de uma variável resultaria na reserva de uma posição de armazenamento de memória[4]). No caso do "SIGNAL", trata-se apenas à explicitação de uma ligação elétrica relevante para o programador. Nas construções concorrentes, uma sentença só é avaliada se algum dos seus sinais à direita da sentença sofrer modificação e um "PROCESS" presente nessa estrutura só é avaliado se algum item da sua lista de sensibilidades sofrer alteração.

Note que existe uma equivalência entre os construtos COMPORTAMENTAIS ⇔ DATAFLOW que pode ser explorada: "IF" ⇔ "WHEN", "CASE" ⇔ "SELECT" e "LOOP"⇔ "GENERATE".

Uma categoria à parte de codificação em VHDL que também está no nível dos comandos concorrentes é conhecida como codificação ESTRUTURAL e usa construções com a *keyword* "PORT MAP" para descrever textualmente dentro de um código VHDL as ligações, de forma equivalente às ligações existentes em um diagrama esquemático. A codificação *estrutural* serve pra descrever as ligações entre componentes instanciados em uma hierarquia superior que necessita deles, o que é usado para construir sistemas de maior complexidade.

Uma atribuição de sinal pode aparecer dentro de um *PROCESS* ou fora dele. Se a atribuição estiver fora do *PROCESS*, ela é considerada uma sentença de atribuição "concorrente" (a menos do tempo de propagação da cadeia de eventos) com as demais sentenças do projeto. Quando uma sentença de atribuição aparece dentro de um *PROCESS*, a atribuição é considerada sequencial e é executada em sequência com as demais atribuições presentes no mesmo bloco *PROCESS*. Neste caso, quando a sentença de atribuição é executada, seu valor é calculado naquele instante, mas o resultado não é "carregado" imediatamente no sinal. Você pode imaginar que ele é "agendado" para ser atribuído ao sinal após o atraso de tempo ΔT. Portanto, normalmente não podemos contar com este resultado na linha de programação seguinte. Se for necessário usar o "conteúdo" atribuído nas linhas seguintes dentro do *PROCESS*, o mais adequado é usar uma *VARIABLE*, que é de caráter temporário e é atualizada instantaneamente.

Resumindo, podemos então enfatizar que existem dois comportamentos diferentes, dependendo de onde está a sentença de atribuição de sinal:

- Sentença **concorrente** de atribuição de sinal (fora de um *process: DATAFLOW*): este tipo de atribuição é disparado por eventos (*event-triggered*), ou seja, elas são executadas sempre que algum dos sinais presentes em sua expressão no lado direito da atribuição sofrer uma alteração. Veja o exemplo a seguir [5]:

```
architecture ATRIBUI_SIG_CON of TRECHO1 is
-- A, B e Z são sinais.
begin  -- Seguem sentenças de atribuição concorrentes:
        A<=B;
        Z<=A;
```

[4] A criação de elementos de memória será tratado mais adiante.

[5] Livro: A VHDL Primer, 3rd Ed. J. Bhasker, Prentice Hall, New Jersey, 1999.

```
end;
```

Na arquitetura ATRIBUI_SIG_CON, as duas sentenças são de execução concorrente. Note que não importa a ordem em que as sentenças aparecem no código. Quando ocorre um evento no sinal B no instante T, o sinal A assume o valor de B após um atraso ΔT, ou seja, no instante T+ΔT. Quando o tempo (na simulação) avança para T+ΔT, o sinal A vai assumir seu novo valor. Se este tiver sido alterado, vai disparar a execução da segunda sentença, o que causa a atribuição do novo valor de A em Z após um novo atraso ΔT, ou seja, no instante T+2ΔT. O valor final de Z será o valor de B.

- Sentenças em um bloco **sequencial** de atribuição de sinal (dentro de um *process: COMPORTAMENTAL*): a atribuição sequencial não é orientada a eventos *(event-triggered)*, mas sim executada consecutivamente às sentenças do corpo de seu *process*. Para entrar em um bloco *PROCESS*, é necessário que pelo menos um dos itens de sua lista de sensibilidade sofra modificação. Por exemplo, se temos um *PROCESS(A,B,C)*, sua *lista de sensibilidade* são os itens *A, B e C* e basta que um deles sofra modificação para seja executado o bloco correspondente. Neste bloco TODAS as sentenças são avaliadas *sequencialmente*, independente de seus termos à direita das expressões de atribuição terem se modificado ou não. Veja o exemplo:

```
architecture ATRIBUI_SIG_SEQ of TRECHO2 is
begin
  process (B)
    begin   -- Seguem sentenças de atribuição sequenciais:
          A<=B;
          Z<=A;
    end process;
end;
```

Na arquitetura ATRIBUI_SIG_SEQ, as duas sentenças são de execução sequencial. Aqui, a ordem que a sentença aparece no código é relevante para o resultado final. Na ocorrência de um evento no sinal B no instante T (neste caso a lista de sensibilidade do bloco *process* tem apenas um item), a primeira sentença de atribuição é executada e em seguida ocorre a execução da segunda sentença, ambas em "tempo zero". Contudo, o novo valor de ambos os sinais somente será atribuído a eles após um atraso ΔT, ou seja, no instante T+ΔT. Quando o tempo (ou a simulação) avança para T+ΔT, o sinal A vai assumir seu novo valor igual a B, e Z receberá o antigo valor de A.

Após estudar estes dois exemplos, note que as definições de atribuição *concorrente* e *sequencial* são pertinentes, mas fazem mais jus à denominação que recebem quando usamos variáveis para programar estruturas comportamentais. Ao contrário do que acontece com os sinais dentro de um *process*, que são avaliados no tempo zero e atualizados posteriormente, as variáveis (declaradas com a *keyword VARIABLE*) são atualizadas já para uso de seu novo valor na sentença seguinte. Portanto, voltando aos exemplos acima, ao olharmos o que acontece com a atribuição dos sinais ao longo do tempo, vemos que os papéis foram trocados: houve uma sequência (orientada a eventos sucessivos) na atribuição concorrente e uma simultaneidade (sincronia) na atribuição sequencial (efeito consumado apenas no final do bloco *"PROCESS"*). O entendimento correto desta característica sutil irá com certeza economizar muitas horas de depuração de código.

Código comportamental/sequencial com sentenças conflitantes

Uma questão surge: se o código comportamental "atualiza" os seus sinais apenas no "final" do bloco, se forem programadas sentenças com resultados conflitantes, qual prevalecerá? Prevalecerá o resultado da sentença que for executada por último na sequência. O exemplo da figura a seguir determina duas ações opostas, ambas a serem executadas na borda de subida do sinal de entrada "b". No primeiro teste da sequência de sentenças, a saída "x" recebe "b". A seguir, temos uma sentença com a mesma condição, mas que agora determina que "x" receba "b negado". O circuito resultante do código como ele se encontra é mostrado no destaque à direita. Experimente

programar o exemplo abaixo e explorar suas variações (por exemplo, comentando as linhas conflitantes, fazendo pequenas alterações de código, etc.). Veja se o código a seguir gera (ou não) algum erro de compilação.

```
1  entity simples is
2      port
3      (
4          b   : in  bit; -- Entrada
5          x   : out bit  -- Saída
6      );
7  end simples;
8
9  architecture sequencial of simples is
10 begin
11     process(b) -- Monitore mudanças em "b"
12     begin
13      if (b'event and b='1') -- Borda de subida de "b"
14        then
15          x <= b;
16      end if;
17      -- Apenas o código a seguir "sobreviverá":
18      if (b'event and b='1') -- Borda de subida de "b"
19         then
20         x <= not b;
21        end if;
22     end process;
23 end;
24
```