Metodologias de Síntese

Paulo Flores

INESC - Apartado 10105, Lisboa PORTUGAL

Índice

1 Introdução

O projecto de circuitos integrados num ambiente de desenho assistido por computador começa tradicionalmente, pela utilização de um editor de esquemáticos. Desta forma, é fornecido ao sistema de produção uma representação a nível lógico do circuito a implementar. Com o aumento da complexidade e dimensão dos circuitos a projectar, resultante de uma maior capacidade de integração do número de transistores num único integrado, este método torna-se demasiado moroso e passível de erros. Surge então, a necessidade de iniciar o projecto a um nível de abstracção superior ao nível lógico.

A forma mais comum de descrever um circuito a alto nível consiste na utilização de linguagens de descrição de *hardware*. Estas permitem a representação do circuito utilizando construções idênticas às das linguagens algorítmicas, mas com características especiais para *hardware*. Esta representação pode ser transformada numa descrição a nível lógico através de uma ferramenta de síntese. No entanto, a qualidade do circuito obtido depende não só do desempenho da ferramenta utilizada como do estilo de descrição adoptado.

Neste trabalho apresenta-se uma definição do subconjunto de instruções de VHDL que são sintetizáveis e a forma de como estas podem ser agrupadas de modo a influenciarem o resultado final da síntese.

Na próxima secção serão explicadas as diferentes formas de representar um circuito e os vários tipos de síntese. Na secção três são referidas as linguagens de descrição de *hardware* mais importantes justificando-se a utilização do VHDL. Uma metodologia de síntese que permite tirar partido da representação dos circuitos a alto nível é proposta na secção quatro. A secção seguinte apresenta uma série de restrições que são impostas à linguagem de descrição de *hardware* no sentido de ser possível a síntese de circuitos. Exemplos de como as descrições de *hardware* são sintetizadas em circuitos são apresentados na secção seis, onde é também proposto um estilo de descrição que facilita a síntese automática de circuitos optimizados. Finalmente na última secção tiram-se algumas conclusões sobre o trabalho apresentado e possíveis desenvolvimentos futuros.

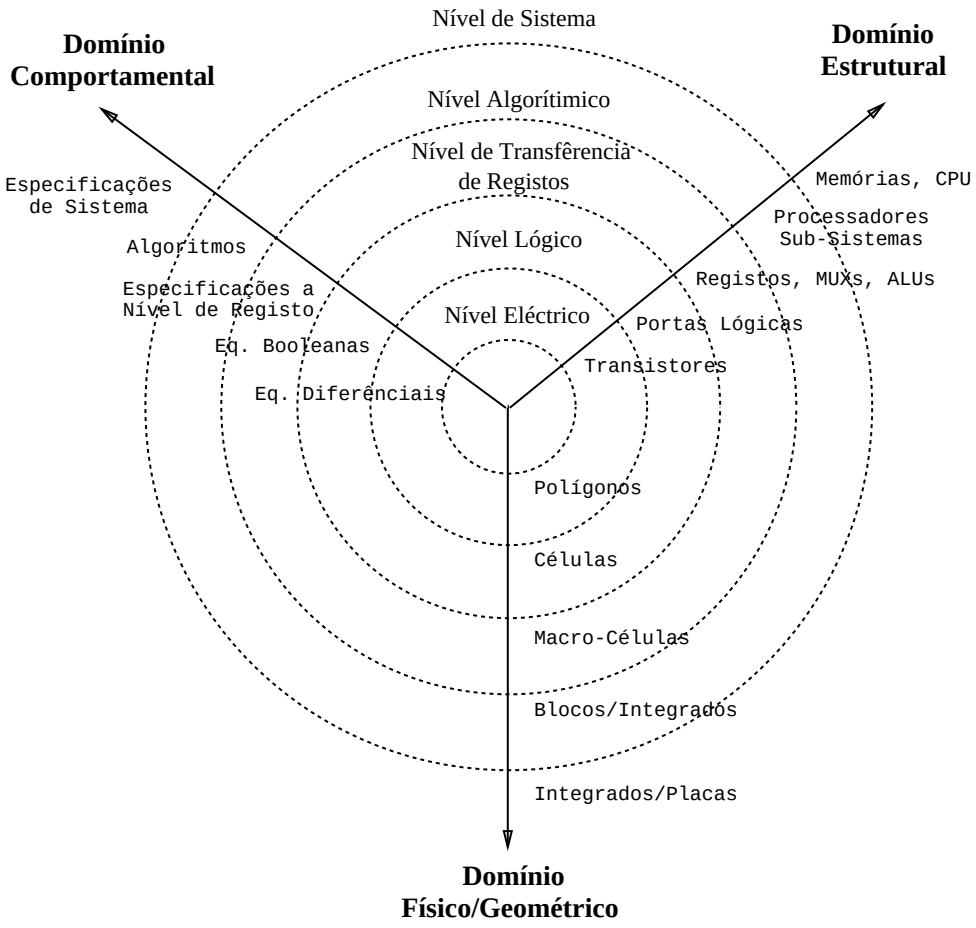

Figura 1: Domínios e níveis de descrição de um sistema.

2 Síntese

A descrição de um circuito digital pode ser feita em diferentes níveis de abstracção, consoante o grau de detalhe pretendido para representar o circuito. Quando este é descrito através de um algoritmo, que determina como as saídas são obtidas em função das entradas, está-se a representar o circuito no nível algorítmico. No entanto, se a representação do circuito for vista como um conjunto de blocos funcionais e elementos de memória, então este está descrito no nível de transferência de registos (RTL[1]). No nível lógico, o circuito é descrito com mais detalhe em termos de portas e funções lógicas.

Para cada nível de abstracção é possível ter uma representação do circuito em diferentes domínios. O domínio comportamental descreve o sistema do ponto de vista de entradas-saídas, idealmente sem qualquer referência à sua estrutura ou à forma como essa

[1]Do inglês *Register-Transfer Level*.

funcionalidade vai ser implementada. O domínio estrutural representa o sistema em termos de elementos de funções bem definidas e das suas interligações. No domínio físico ou geométrico essa estrutura é mapeada no espaço sem qualquer referência à funcionalidade que representa.

Estas várias formas de representar um circuito electrónico encontram-se na figura 1, onde são também indicados os vários elementos envolvidos em cada representação.

O projecto de um circuito integrado a níveis de abstracção superior ao nível lógico passa pelas seguintes fases: obtenção de uma descrição do circuito a alto nível, síntese de uma representação lógica do circuito e o mapeamento tecnológico dessa representação para o domínio físico para que o circuito possa ser fabricado.

Assim o processo de síntese consiste em determinar a melhor estrutura que representa um circuito a partir de uma descrição do seu comportamento. Naturalmente, existirão tantos níveis de síntese quantos os níveis de abstracção possíveis para descrever o comportamento do circuito.

2.1 Síntese de alto-nível

A síntese de alto-nível é também por vezes designada por síntese a nível de algoritmo, pois tem como ponto de partida uma descrição algorítmica que define os procedimentos necessários para obter as saídas em função das entradas.

Neste passo de síntese os algoritmos, descritos de forma idêntica às linguagens de programação de alto-nível, vão ser mapeados em registos, operadores e estruturas de controlo. Este mapeamento é feito à custa das três tarefas que a seguir se descrevem:

Reserva de recursos - são determinados quais os operadores, registos e outros recursos de *hardware* que irão ser utilizados.

Planeamento - determinação e distribuição das operações de controlo pelos vários intervalos de tempo (períodos de relógio).

Atribuição de recursos - criação das várias instâncias referentes aos recursos reservados.

2.2 Síntese a nível de transferência de registos

O resultado da síntese de alto-nível é uma descrição RTL de um conjunto de fluxos de dados e dos respectivos controladores. A síntese a nível de transferência de registos consiste pois em obter com essa descrição a implementação estrutural do circuito.

Na síntese RTL é necessário, a partir da descrição do conjunto de estados que controlam o fluxo de dados, seleccionar uma arquitectura apropriada e efectuar a codificação dos estados, das entradas e das saídas. Este passo de síntese envolve também uma optimização do fluxo de dados tendo em conta as características especificas das unidades a utilizar.

2.3 Síntese lógica

A síntese lógica tem como entrada uma representação do circuito em termos de funções lógicas e de elementos de memória e como saída, o mapeamento e optimização destas funções a nível de portas lógicas considerando uma biblioteca específica de componentes.

A passagem do domínio comportamental para o domínio estrutural pode ser feita de forma mais ou menos directa. O passo de maior complexidade da síntese lógica é a optimização da lógica obtida em termos de área e/ou velocidade. Esta optimização tem em conta a tecnologia que posteriormente vai ser utilizada para a sua implementação. Por exemplo, para uma implementação numa PLA é necessário uma representação optimizada a dois níveis (soma de produtos), enquanto que para um FPGA essa representação pode ser a multi-nível.

O último passo da síntese lógica é o mapeamento tecnológico. Este processo parte de uma dada descrição estrutural, independente da tecnologia e com um conjunto de restrições, implementado-a no mesmo nível de representação, mas agora no domínio físico [Michel 92].

3 Linguagens de Descrição de *Hardware*

As linguagens de descrição de *hardware* representam os sistemas num nível de abstracção mais elevado que a habitual representação em termos de portas lógicas, permitindo a descrição e documentação dos sistemas de uma forma independente da tecnologia em que o sistema irá ser implementado e a simulação da funcionalidade pretendida sem necessidade

de se descer ao nível lógico. Estas linguagens incluem habitualmente mecanismos que possibilitam a sua utilização para a representação de circuitos no domínio estrutural em diferentes níveis de abstracção.

Originalmente muitas das linguagens de descrição de *hardware* evoluíram das linguagens de programação, herdando portanto muitas das características daquelas [Shiva 79, Waxman 86]. No entanto, as linguagens de descrição de *hardware* diferem em vários aspectos das linguagens algorítmicas, nomeadamente no modo de referenciação do tempo que não é importante nas linguagens usuais de programação [Augustin 89].

As linguagens de descrição de *hardware* têm uma estrutura adaptada às metodologias de projecto de sistemas electrónicos. Isto é, para além de permitirem diferentes níveis e estilos de descrição dos sistemas foram pensadas não só para a simulação mas também para a síntese e verificação formal. As linguagens de descrição de *hardware* mais conhecidas são:

Verilog - Linguagem originalmente proprietária da Cadence[2] e bastante divulgada nos Estados Unidos da América. Actualmente sofre um processo de abertura ao domínio público através da organização OVI (Open Verilog International).

UDL/I - *Unified Design Language / for Integrated circuits*. Proposta pela Associação para o Desenvolvimento da Indústria Electrónica Japonesa como linguagem de descrição de *hardware* a normalizar [Yasuura 89, UDL 92].

VHDL - *VHSIC*[3] *Hardware Description Language* tornou-se numa norma do DoD[4]/IEEE para todos os projectos de electrónica. Hoje em dia esta linguagem "começa a ser" suportada praticamente por todos os sistemas comerciais de CAD.

Ao contrário de outras linguagens, que estão intimamente ligadas às companhias que as desenvolveram, o VHDL é uma linguagem pública e independente. A sua escolha para projecto de circuitos integrados beneficia não só do crescente aumento das aplicações de CAD que a aceitam como forma de representar um circuito digital como também da sua ampla divulgação internacional.

[2]Companhia americana que desenvolve e comercializa sistemas de CAD.

[3] *Very High Speed Integrated Circuits*

[4]Departamento de Defesa dos Estados Unidos da América

4 Metodologia de síntese

O fluxo de projecto habitualmente seguido para se conseguir uma descrição lógica a partir de uma representação do circuito em VHDL recorrendo a uma ferramenta de síntese é apresentado na figura 2.

O processo de síntese da lista de portas lógicas que implementam o circuito a partir da sua descrição numa linguagem de alto nível não se efectua num único passo. É habitualmente realizado de uma forma iterativa e interactiva sendo o papel do projectista fundamental para a qualidade dos resultados obtidos. O processo iterativo resulta do baixo grau de maturidade que as actuais ferramentas apresentam, necessitando por isso de serem "guiadas" durante o processo de síntese.

No intuito de reduzir ao mínimo o número de iterações é importante utilizar uma metodologia de síntese que facilite a interacção do projectista com a ferramenta de síntese. Esta não é mais que um conjunto de princípios, procedimentos e práticas que o projectista deve seguir de forma a melhor "comunicar" àquela quais os seus objectivos do projecto.

Uma vez que o ponto de partida de todo este processo é a descrição do sistema em VHDL, torna-se claro que para além de um profundo conhecimento da linguagem, o projectista tem necessidade de saber quais as instruções que melhor se adaptam à ferramenta de síntese e qual o estilo de codificação mais apropriado para o projecto em causa.

5 Instruções de VHDL sintetizáveis

O VHDL foi criado originalmente para a descrição e simulação de sistemas electrónicos. Por este motivo algumas das suas instruções estão intimamente ligadas à simulação e não são em geral sintetizáveis. Por exemplo, a capacidade do VHDL suportar estruturas, aritmética com números reais e utilização de ficheiros para entrada e saída torna-a muito conveniente e sofisticada para a modelação de sistemas mas requer capacidades irrealistas para as ferramentas de síntese [Levitan 89].

A definição do subconjunto de construções a suportar para síntese não está ainda normalizado, dependendo actualmente da ferramenta de síntese usada. No caso deste trabalho, e para exemplificar a síntese de circuitos a partir de descrições em VHDL, utilizou-se uma ferramenta de síntese comercial [Synthesis 91]

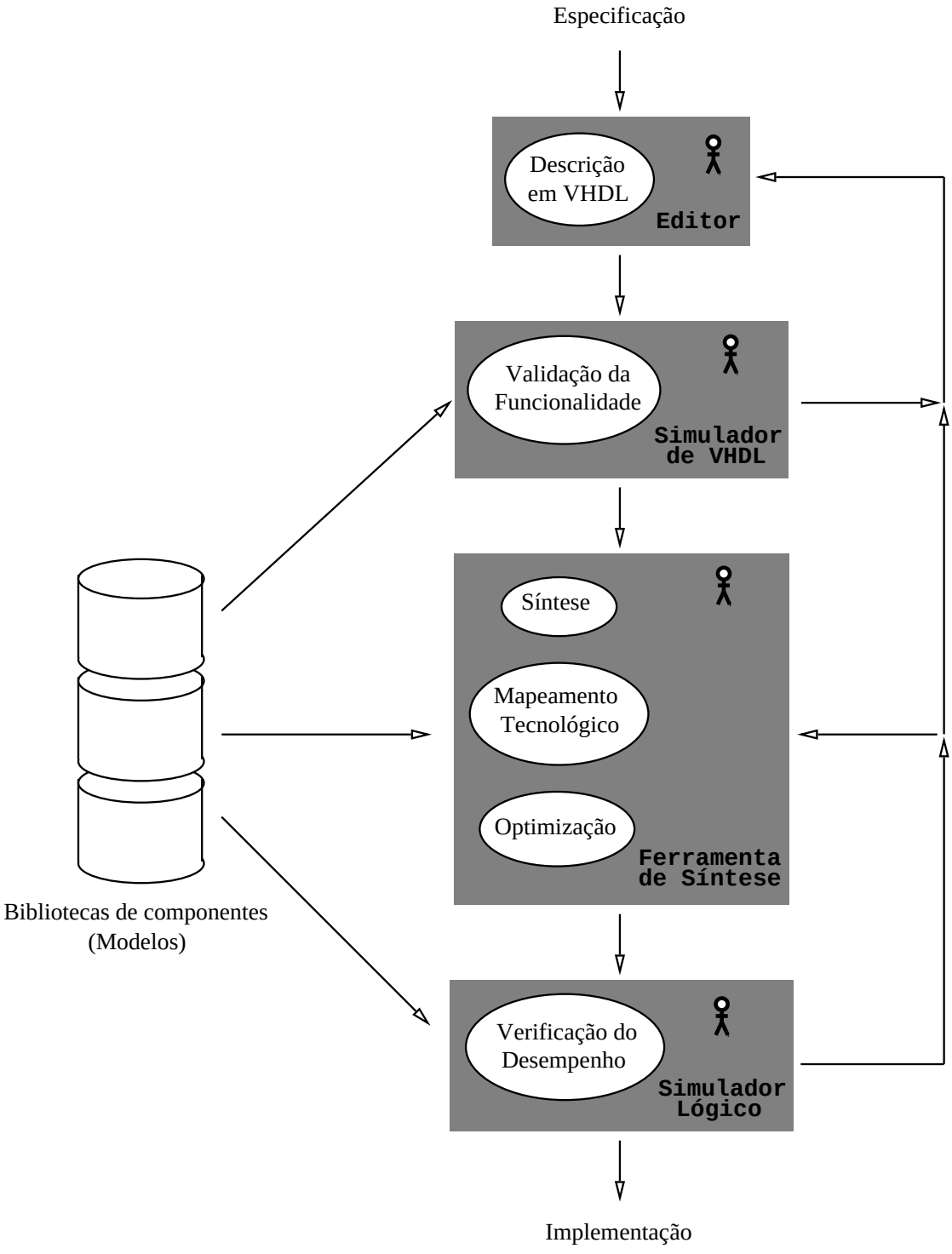

Figura 2: Fluxo de projecto

As construções VHDL do ponto de vista da síntese, podem ser agrupadas em quatro categorias: não sintetizáveis, sintetizáveis com restrições, sintetizáveis e específicas para síntese.

5.1 Não sintetizáveis

As construções não sintetizáveis podem ser divididas em dois tipos: as que a ferramenta de síntese pode ignorar sem perca de validade do modelo descrito e as que ao serem ignoradas podem alterar o significado da descrição.

No primeiro caso encontram-se as instruções de `Assertion` que servem apenas como mecanismos de validação de modelos em VHDL (verificação dos tempos de *setup* e *hold* em *flip-flops*, etc) e não influenciam a funcionalidade do circuito. Podem por isso ser utilizadas numa descrição sintetizável apesar de serem ignoradas pela ferramenta de síntese.

As construções do segundo tipo embora não sejam suportadas pela ferramenta de síntese alteram o significado do modelo descrito, razão pela qual não devem ser incluídas numa descrição sintetizável. São exemplos, deste tipo de construções, não suportados para síntese:

- Os tipos `File`, as declarações de ficheiros e consequentemente o `Package TEXTIO`. Os uso destas construções permite, por exemplo modelar memórias (ROMs ou FIFOs) que ao não serem sintetizadas pela ferramentas de síntese podem levar a que a especificação do circuito fique incompleta.

- Os tipos `Access` e `Incomplete`, que permitem o uso de apontadores, bem como a instrução de reserva dinâmica de memória, `New`, têm capacidades cuja representação em *hardware* não é facilmente obtida de forma a manter a mesma semântica.

- Os atributos pré-definidos na linguagem e que só têm significado para a simulação tais como: `Delayed, Quiet, Transaction, Active, Last_Event` e `Last_Active`.

- Objectos do tipo `Physical` e `Floating`. A sua utilização requereria o suporte de um conjunto de operadores que as actuais ferramentas não têm capacidade de sintetizar.

5.2 Sintetizáveis com restrições

Existem construções de VHDL que apesar de serem suportadas pela ferramenta de síntese têm a sua utilização limitada. Estão neste caso as seguintes construções:

- As instruções de `wait`, que permitem a sincronização de um processo com um determinado sinal (em geral o relógio), têm a sua utilização restringida às formas representadas na figura 3. Adicionalmente, cada processo só poderá conter uma única instrução de `wait`, que terá de ser a primeira instrução do processo.

wait until sinal = `valor_lógico` ;
wait until sinal**'event and** sinal = `valor_lógico` ;
wait until not sinal**'stable and** sinal = `valor_lógico` ;

O `valor_lógico` só poderá tomar os valores '1' ou '0' conforme se pretenda lógica síncrona sensível ao flanco ascendente ou descendente do sinal.

Figura 3: Utilizações válidas da instrução `wait` no sistema [Synthesis 91].

- A lista de sinais ao qual um processo é sensível não é respeitada pela ferramenta de síntese: o processo reage a qualquer alteração dos seus sinais de entrada independentemente de estes constarem ou não da lista de sensibilidade. Se este comportamento não for tido em conta na descrição do circuito o resultado da síntese poderá não apresentar a funcionalidade desejada (ver exemplo da figura 4). Para evitar diferentes resultados entre a simulação e o circuito sintetizado, um processo deve incluir na sua lista de sensibilidade todos os sinais que usa em modo de leitura e cuja mudança de valor pode levar a alterar outros sinais.

- Os operadores aritméticos estão em geral limitados nos cálculos que podem efectuar. Aos operadores `/, mod, rem` requer-se que o operando da direita seja um potência de 2 calculável[5], enquanto que no operador `**` essa mesma restrição é no operando da esquerda (base).

- A instrução de atribuição a sinais não suporta a especificação de tempos de atraso (cláusula `after`). Estes tempos, são ignorados pela ferramenta de síntese, embora pudessem ser interpretados de várias formas: como um atraso máximo, mínimo ou típico.

[5] O seu valor pode ser determinado estaticamente.

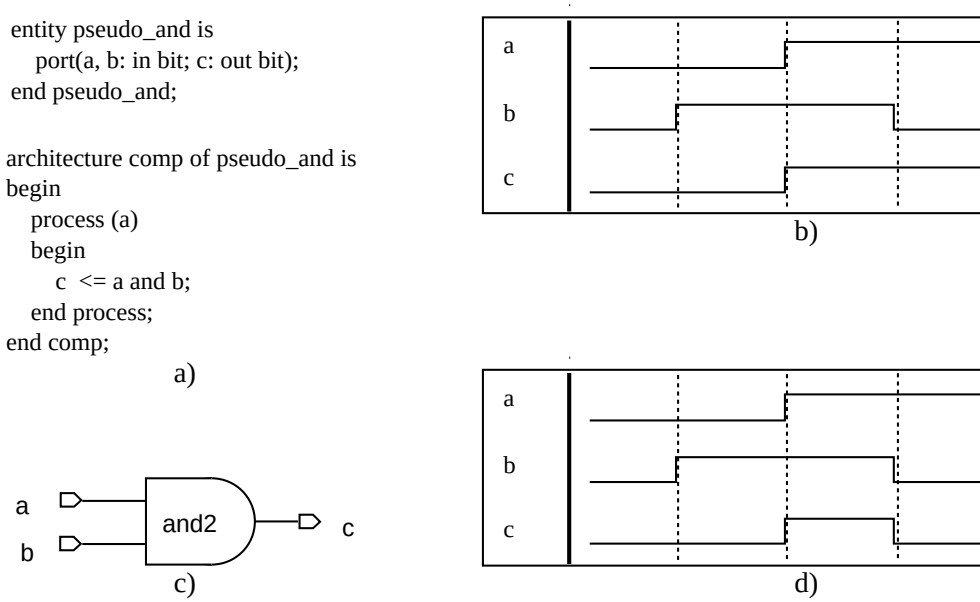

Figura 4: a) Código VHDL; b) Simulação VHDL; c) Circuito sintetizado; d) Simulação lógica.

5.3 Sintetizáveis

As construções sintetizáveis são aquelas que não são directamente restringidas pela ferramenta de síntese, embora o seu uso possa ser de certa forma limitado devido às restrições impostas pelas construções referidas anteriormente.

As construções suportadas são:

- As instruções sequenciais `if, case, next, return, null` e atribuição.
- As instruções concorrentes `generate`, instanciação de componentes e atribuição.
- O uso de procedimentos e funções que podem ser chamados do domínio concorrente ou sequencial.
- A utilização de `packages`.
- Os atributos `right, high, low, base, left, range` e `length`.
- Os tipos de dados enumerados, inteiros (de precisão finita), agregados e vectores.
- Os operadores lógicos, relacionais e de adição, multiplicação e concatenação.

5.4 Específicas para síntese

As construções de VHDL específicas para síntese permitem explicitamente orientar a ferramenta de síntese na obtenção do circuito desejado. Este objectivo pode ser conseguido através de dois tipos de construções: atributos definidos pelo utilizador ou comentários sintéticos[6].

O uso de atributos definidos pelo utilizador, com especial significado para a ferramenta de síntese, permite transmitir a esta (juntamente com a descrição VHDL) quais as restrições impostas ao circuito em termos de área, velocidade, etc.

Os comentários sintéticos para além de alterarem a linguagem de uma forma artificial, permitem incluir na própria descrição do circuito comandos para a ferramenta de síntese. Como os comentários sintéticos só são interpretados por uma ferramenta em particular, a sua utilização deve ser evitada.

6 Estilos de descrição de VHDL

Tão importante como saber quais as instruções de VHDL sintetizáveis, é saber como agrupar essas instruções de forma a conseguir uma representação do comportamento do circuito que seja aceite pela ferramenta de síntese.

Assim, e antes de abordarmos os diferentes estilos de descrição, e a forma como estes podem influenciar o processo de síntese, vamos analisar alguns exemplos que ilustram como um circuito lógico é sintetizado a partir de uma descrição VHDL. Os exemplos apresentados mostram como as descrições de VHDL são interpretadas por um sistema de síntese comercial, utilizando uma biblioteca típica dum fabricante de circuitos integrados.

6.1 Exemplos de como as descrições de VHDL são sintetizadas

Cada descrição em VHDL é sintetizada num circuito lógico cujas entradas e saídas correspondem aos sinais que foram declarados nos acessos da entidade de VHDL, e cuja funcionalidade é a representada pela arquitectura respectiva. A figura 5 apresenta um

[6]Comentários cuja primeira palavra é especial e serve para indicar que o resto da linha deve ser interpretado pela ferramenta. Em inlgês *synthetic comments*.

circuito com três entradas e uma saída que realiza a função lógica descrita na atribuição concorrente ao sinal de saída.

```
entity cir  is
    port(a, b, c: in bit;
         z: out bit);
end cir;


architecture logica of cir  is
begin
    z <= (a and b) or c;
end logica;
```

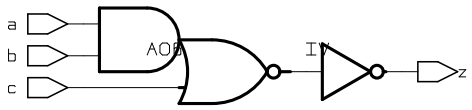

Figura 5: Atribuição concorrente em VHDL

A utilização de células particulares da biblioteca pode ser feita recorrendo a uma descrição estrutural. Este estilo, que consiste na instanciação e interligação dos componentes do circuito, está exemplificado na descrição da figura 6. Nesta, mostra-se também o uso dum `package` para a declaração do componente AO2.

```
package porta_especial is
    component AO2
        port(A, B, C, D: in bit;
             Z: out bit);
    end component;
end porta_especial;


use work.porta_especial.all;

entity cir  is
    port(a, b, s: in bit;
         z: out bit);
end cir;


architecture estrutural of cir  is
    signal s_neg: bit;
begin
    s_neg <= not s;
    u1: AO2
        port map(a, s, b, s_neg, z);
end estrutural;
```

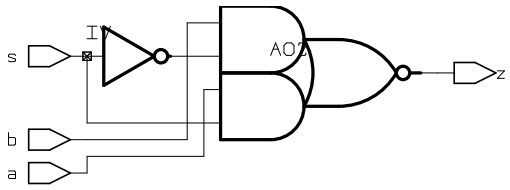

Figura 6: Instanciação de componentes e uso de `packages`

O uso de vectores permite a representação de *buses*. Na síntese da indexação de elementos de um vector há que considerar dois casos distintos. Um, quando o índice de selecção é constante e portanto o resultado será apenas um fio, e outro, quando o índice de selecção é variável, tendo por isso que se recorrer ao uso de multiplexers. Ambos os

casos são representados no exemplo da figura 7.

```
entity cir  is
    port(a, b: in bit_vector(0 to 3);
         sel: in integer range 0 to 3;
         saida_a, saida_b: out bit);
end cir;


architecture sel_vec of cir  is
begin
    saida_a <= a(3);
    saida_b <= b(sel);
end sel_vec;
```

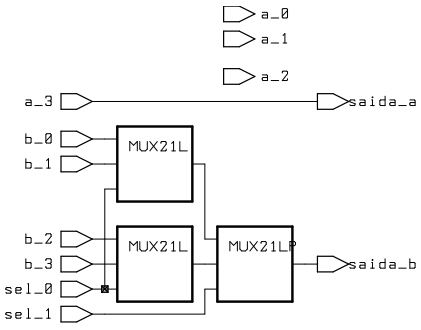

Figura 7: Selecção dum elemento de um vector

Na figura 8 exemplifica-se o uso de operadores aritméticos através da soma de dois inteiros. O circuito lógico sintetizado é um somador de 3 *bits* com *carry*.

```
entity cir  is
    port(a, b: in integer range 0 to 7;
         z: out integer range 0 to 14);
end cir;


architecture aritmetica of cir  is
begin
    z <= a + b;
end aritmetica;
```

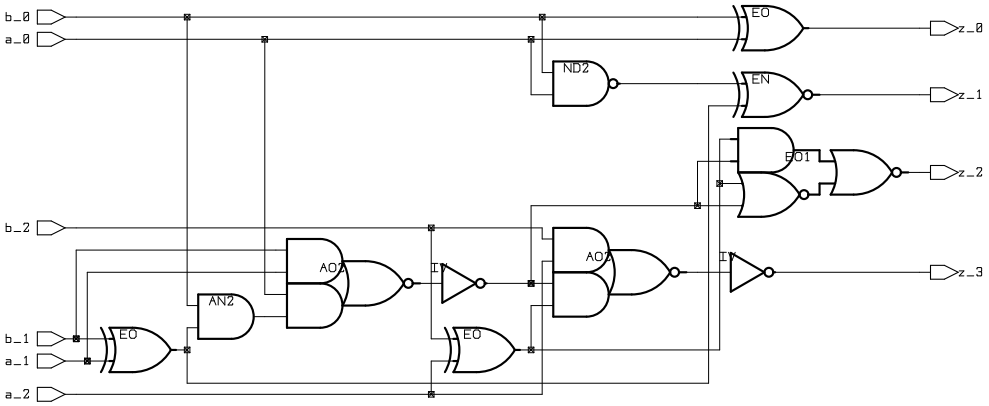

Figura 8: Uso de operadores aritméticos com inteiros

A utilização de processos em VHDL permite a descrição do comportamento do circuito de uma forma sequencial e recorrendo às instruções mais comuns das linguagens de programação de alto nível.

A síntese de uma instrução `if-then-else` inserida dentro de um processo resulta num multiplexer. A selecção das entradas deste está dependente da condição do `if` como se observa no exemplo da figura 9.

```
entity cir  is
    port(a, b, sel_a: in bit;
         z: out bit);
end cir;


architecture proc_if of cir  is
begin
    process(a, b, sel_a)
    begin
        if sel_a = '1' then
            z <= a;
        else
            z <= b;
        end if;
    end process;
end proc_if;
```

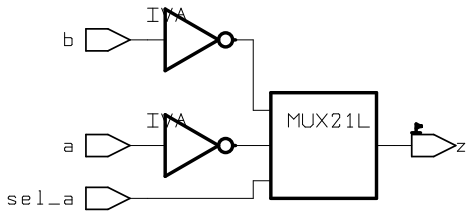

Figura 9: Síntese de um `if-then-else` dentro de um processo

Se a atribuição a um sinal não for feita em ambos os ramos da instrução `if`, o resultado da síntese é um *latch* activado pela condição do `if` (ver exemplo na figura 10). O uso deste elemento de memória é necessário porque enquanto a condição do `if` for falsa, o sinal tem que manter o valor anterior.

```
entity cir  is
    port(a, sel_a: in bit;
         z: out bit);
end cir;


architecture proc_if of cir  is
begin
    process(a, sel_a)
    begin
        if sel_a = '1' then
            z <= a;
        end if;
    end process;
end proc_if;
```

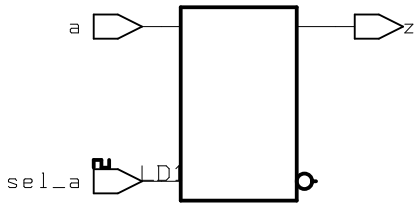

Figura 10: Síntese de um `if` sem o ramo `else`

A instrução `case` produz resultados idênticos aos do `if` mas a condição de selecção dos multiplexers resulta da avaliação da expressão do `case`. Mais à frente veremos a utilização desta instrução noutras situações.

A figura 11 apresenta o resultado da síntese de um ciclo `for` com os limites determina-

dos pelo atributo `range`. Exemplifica-se também a utilização de uma variável temporária, declarada dentro do processo, que permite guardar o valor da paridade dos *bits* já calculados.

```
entity cir  is
    port(a: in bit_vector(0 to 7);
         paridade: out bit);
end cir;


architecture proc_for of cir  is
begin
    process(a)
        variable par_temp: bit;
    begin
        par_temp := '0';
        for i in a'range loop
            par_temp := par_temp xor a(i);
        end loop;
        paridade <= par_temp;
    end process;
end proc_for;
```

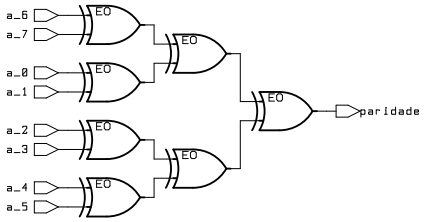

Figura 11: Uso do ciclo `for` recorrendo ao atributo `range`

A descrição de circuitos sequenciais síncronos, onde serão sintetizados elementos de memória (*flip-flops*), pode ser feita recorrendo ao uso da instrução `wait`. A figura 12 representa um circuito síncrono em que a saída `z` alterna entre os valores `0` e `1` sempre que houver um impulso de relógio e a entrada `troca` esteja activa. É a descrição de um *flip-flop* do tipo T que é sintetizado à custa dos elementos de memória existentes na biblioteca, neste caso um *flip-flop* tipo D.

Outra forma de obter circuitos com *flip-flops* pode ser conseguida à custa do uso da instrução `if`. Na figura 13 apresenta-se a síntese de um *flip-flop* com *reset* assíncrono a partir de duas condições de `if`, uma para o *reset*, e outra para a lógica síncrona. Se se pretender um *reset* síncrono, pode-se recorrer à descrição da figura 14.

No exemplo apresentado na figura 15 descreve-se um contador de módulo dez com *reset* assíncrono. A sua descrição apresenta uma construção típica de elementos de memória, activos no flanco ascendente do relógio e com *reset* assíncrono activo a `1`. Assim, por cada impulso de relógio o contador será incrementado de uma unidade e uma vez atingido o valor nove, deverá no próximo flanco do relógio inicializar a sua contagem em zero.

A descrição de máquinas de estado pode ser feita em VHDL de uma forma muito fácil,

```
entity cir  is
    port(troca, clock: in bit;
        z: buffer bit);
end cir;


architecture proc_wait of cir  is
begin
    process
    begin
        wait until clock'event and clock = '1';
        if troca = '1' then
            z <= not z;
        end if;
    end process;
end proc_wait;
```

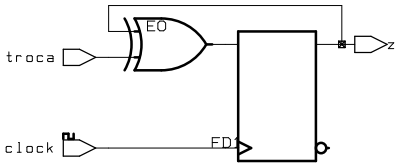

Figura 12: Uso da instrução `wait` para a geração de circuitos sequenciais

```
entity cir is
    port(dados, clock, reset: in bit;
        z: out bit);
end cir;


architecture proc_reset of cir is
begin
    process
    begin
        if reset = '1' then
            z <= '0';
        elsif clock'event and clock = '1' then
            z <= dados;
        end if;
    end process;
end proc_reset;
```

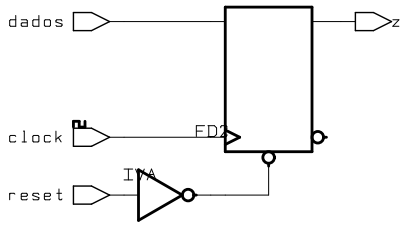

Figura 13: Síntese de um *flip-flop* com *reset* assíncrono

recorrendo a uma arquitectura onde são definidos dois processos. Um deles é responsável pela definição dos elementos síncronos do circuito, isto é, os elementos de memória da máquina de estados. No outro, é descrita a parte combinatória do circuito, ou seja, as transições entre os estados, consoante os valores das entradas e as respectivas actualizações dos valores das saídas.

No exemplo da figura 17 apresenta-se uma máquina de estados de Moore que serve para detectar a sequência de entrada `'111'`. A tabela de transição de estados desta máquina está representada na figura 16.

Esta tabela é descrita no processo correspondente à lógica combinatória através de um `case` controlado pelo estado actual da máquina. Em cada um dos ramos do `case` é

```
entity cir  is
    port(dados, clock, reset: in bit;
        z: out bit);
end cir;


architecture proc_reset of cir  is
begin
    process(clock, reset)
    begin
        if clock'event and clock = '1' then
            if reset = '1' then
                z <= '0';
            else
                z <= dados;
            end if;
        end if;
    end process;
end proc_reset;
```

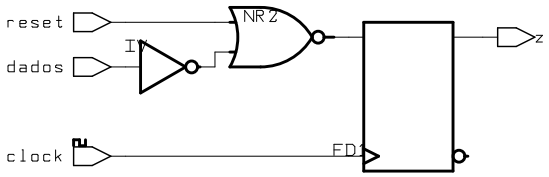

Figura 14: Síntese de um *flip-flop* com *reset* síncrono

actualizado o valor da saída e definido através de um `if` dependente do valor de entrada, qual o estado para o qual a máquina evoluirá. Este processo encontra-se identificado no código da figura 17 com o *label* `comb`.

O processo identificado com o *label* `sinc` é responsável pelo funcionamento síncrono da máquina de estados. É este processo que define que sempre que haja uma transição do relógio de zero para um, o estado actual da máquina vai ser actualizado. É incluído também um `if` para definir o *reset* dos elementos de memória permitindo que a máquina evolua a partir de um estado conhecido.

6.2 Formas de aumentar a "qualidade" do circuito sintetizado

Após termos analisado a forma como é feita a síntese para certas construções de VHDL típicas, iremos agora salientar alguns pontos que devem ser tidos em conta quando se descreve um circuito em VHDL para ser sintetizado. Note-se que, a qualidade do circuito sintetizado depende tanto da capacidade da ferramenta usada, como da forma e qualidade, da descrição de VHDL que é fornecida.

Uma das grandes vantagens das ferramentas de síntese é permitir de uma forma fácil e rápida, explorar uma variedade de arquitecturas possíveis para implementar um dado circuito. No entanto, em muitos dos casos todas essas possibilidades não necessitam de ser exploradas, pois muitas das vezes o projectista sabe qual a estrutura "óptima" que deve

```
entity cir  is
    port(clock, reset: in bit;
        count: buffer integer range 0 to 9);
end cir;


architecture contador of cir  is
begin
    process(clock, reset)
    begin
        if reset = '1' then
            count <= 0;
        elsif clock'event and clock = '1' then
            if count = 9 then
                count <= 0;
            else
                count <= count + 1;
            end if;
        end if;
    end process;
end contador;
```

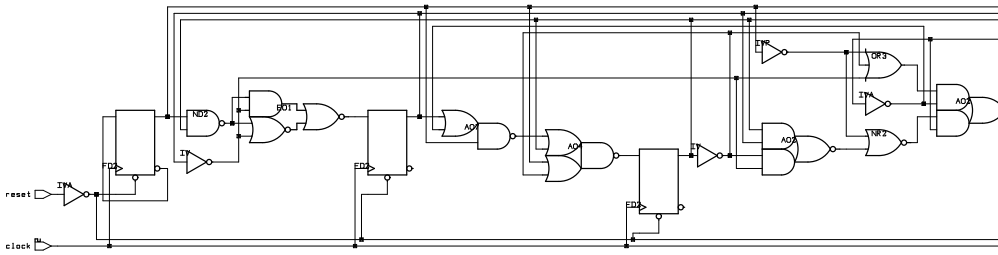

Figura 15: Contador de módulo 10 com *reset* assíncrono

Estado	Próximo Estado		Saída
Actual	`entrada` = 0	`entrada` = 1	`sequencia`
zero	zero	um	0
um	zero	dois	0
dois	zero	tres	0
tres	zero	tres	1

Figura 16: Tabela de transição da máquina de estados

ser usada em certas partes do circuito. Compete, neste caso, ao projectista transmitir à ferramenta a arquitectura desejada. Isto pode ser conseguido de várias formas, das quais a partição do circuito em blocos e a criação de uma hierarquia, usando várias entidades de projecto em VHDL, é o primeiro passo a ser dado. Esta hierarquia pode posteriormente ser mantida, total ou parcialmente, durante a fase de optimização.

```
entity cir  is
  port(entrada, clock, reset: in bit;
    sequencia: out bit);
end cir;


architecture fsm_1 of cir  is
  type estados is (zero, um, dois, tres);
  signal estado_actual, proximo_estado: estados;
begin

  comb: process(entrada, estado_actual)
  begin
    case estado_actual is
      when zero =>
        sequencia <= '0';
        if entrada = '1' then
          proximo_estado <= um;
        else
          proximo_estado <= zero;
        end if;
      when um =>
        sequencia <= '0';
        if entrada = '1' then
          proximo_estado <= dois;
        else
          proximo_estado <= zero;
        end if;
      when dois =>
        sequencia <= '0';
        if entrada = '1' then
          proximo_estado <= tres;
        else
          proximo_estado <= zero;
        end if;
      when tres =>
        sequencia <= '1';
        if entrada = '1' then
          proximo_estado <= tres;
        else
          proximo_estado <= zero;
        end if;
    end case;
  end process;

  sinc: process(reset, clock)
  begin
    if reset = '1' then
      estado_actual <= zero;
    elsif clock'event and clock = '1' then
      estado_actual <= proximo_estado;
    end if;
  end process;

end fsm_1;
```

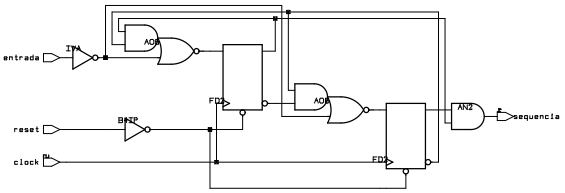

Figura 17: Máquina de estados que detecta a sequência '111'

O uso de subprogramas, procedimentos ou funções, é outra forma de fornecer alguma estrutura ao sistema de síntese. Não porque o uso destes crie explicitamente um novo nível hierárquico mas porque permite facilitar a fase de optimização.

O passo seguinte no sentido de guiar a ferramenta para a arquitectura desejada reflecte-se na forma como o próprio VHDL é escrito. O uso de ciclos `for` ou `generate` permite criar estruturas repetitivas de uma forma fácil. No exemplo da figura 18 utiliza-se a instrução `generate` para instanciar e interligar quatro *flip-flops*, de forma a constituírem um conversor série-paralelo. Note-se que neste caso era também possível obter o mesmo circuito usando a descrição apresentada na figura 19, que para além de ser mais simples, representa o circuito de um forma independente da biblioteca de células usada.

```
entity cir  is
    port(dados, clock, reset: in bit;
        par: out bit_vector(0 to 3));
end cir;


architecture sr_gen of cir  is

    component IVA
        port( A : in BIT;  Z : out BIT);
    end component;

    component FD2
        port( D, CP, CD : in BIT;  Q, QN : out BIT);
    end component;

    signal reset_neg: bit;
    signal aux: bit_vector(0 to 3);
begin
    inv: IVA port map (reset, reset_neg);
    ff: for i in aux'range generate
        ff1: if i = 0 generate
            comp_ff1: FD2 port map (dados, clock, reset_neg, aux(i), open);
        end generate;
        ffi: if i > 0 generate
            comp_ffi: FD2 port map (aux(i-1), clock, reset_neg, aux(i),open);
        end generate;
    end generate;
    par <= aux;
end sr_gen;
```

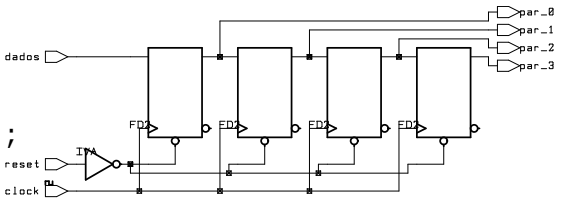

Figura 18: Instanciação de componentes para formar um registo de deslocamento

A própria utilização de parêntesis permite agrupar lógica de forma a obterem-se circuitos com a estrutura desejada. O exemplo da figura 20 ilustra este facto através de duas somas com quatro inteiros, sendo a primeira sem parêntesis e a segunda com os operandos agrupados dois a dois. Como resultado após a síntese verifica-se que usando a segunda opção o circuito obtido é mais rápido e mais pequeno (usam-se dois somadores

```
entity cir  is
    port(dados, clock, reset: in bit;
        par: buffer bit_vector(0 to 3));
end cir;


architecture sr_comp of cir  is
begin
    synch: process(clock, reset)
    begin
        if reset = '1' then
            par <= "0000";
        elsif (clock'event and clock = '1') then
            par <= (dados & par(0 to 2));
        end if;
    end process;
end sr_comp;
```

Figura 19: Registo de deslocamento de 4 *bits*

de 4 entradas e um de 5 em vez de dois de 5 e um de 4).

```
entity cir  is
    port(a, b, c, d: in integer range 0 to 7;
        e, f, g, h: in integer range 0 to 7;
        x, z: out integer range 0 to 28);
end cir;


architecture aritmetica of cir  is
begin
    x <= a + b + c + d;
    z <= (e +f) + (g + h);
end aritmetica;
```

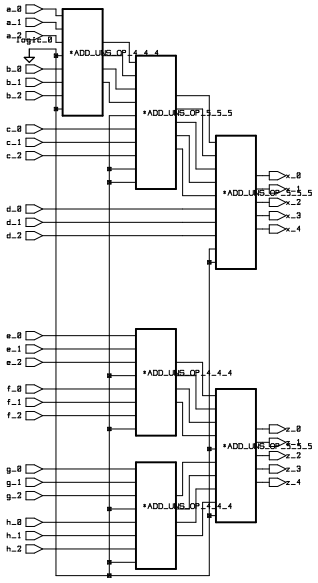

Figura 20: O uso de parêntesis como forma de estruturar o circuito

Outro factor que influencia a qualidade do circuito final obtido é a forma como os vários recursos existentes no circuito são partilhados. Isto é, o uso de operadores complexos como somadores, comparadores, multiplicadores e outros deve ser feito de forma a evitar a sua duplicação desnecessária. Para isso deve-se sempre que possível usar variáveis temporárias para colocar em evidência factores comuns. Na figura 21 apresenta-se uma descrição em que este tipo de optimização não foi tido em conta, e em que o circuito sintetizado é maior

que aquele representado na figura 22 onde apenas existe uma operação de adição.

Note-se que no sistema [Synthesis 91] a partilha de certos recursos existentes no circuito (somadores, subtractores, comparadores, multiplicadores) pode ser feita de uma forma automática, mas existem outros (registos de deslocamento, indexação dinâmica de vectores e chamadas a funções) cuja utilização não é minimizada. No entanto, se a partilha de recursos for logo à partida inserida no código, torna-se possível um melhor controlo sobre os resultados produzidos para além da redução do tempo de síntese.

```
entity cir_2  is
    port(sel_a: in bit;
        a, b, c: in integer range 0 to 3;
        z: out integer range 0 to 6);
end cir_2;


architecture soma of cir_2  is
begin
    process
    begin
        if sel_a = '1' then
            z <= a + c;
        else
            z <= b + c;
        end if;
    end process;
end soma;
```

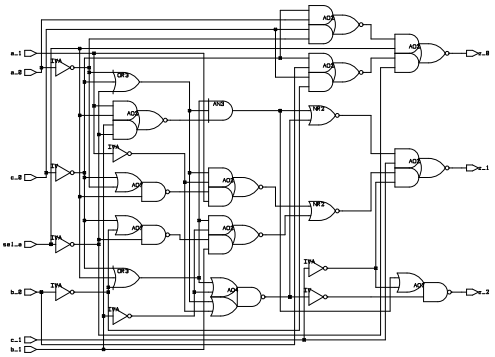

Figura 21: Uso de adições desnecessárias produz circuitos maiores

```
entity cir  is
    port(sel_a: in bit;
        a, b, c: in integer range 0 to 3;
        z: out integer range 0 to 6);
end cir;


architecture soma of cir  is
begin
    process
    variable temp: integer range 0 to 3;
    begin
        if sel_a = '1' then
            temp := a;
        else
            temp := b;
        end if;
        z <= temp + c;
    end process;
end soma;
```

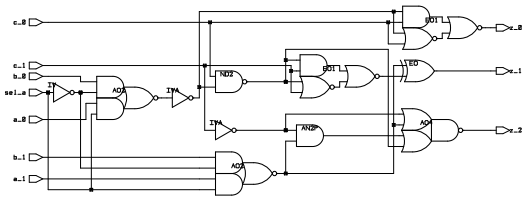

Figura 22: Evidenciando factores consegue-se reduzir a área do circuito

Na descrição de máquinas de estado deve ser utilizado o formato que foi apresentado na figura 17, devendo salientar-se a utilização das seguintes construções:

- Declaração de tipo enumerado para a representação dos vários estados possíveis da máquina. O uso deste tipo de dados permite representar a máquina de estados a um nível simbólico e não ao nível de *bits*. Isto é, existe um grau de liberdade adicional que possibilita à ferramenta de síntese escolher a melhor codificação para os estados. Caso seja conhecida a codificação óptima esta informação pode ser fornecida ao sistema de síntese.

- A inicialização da máquina de estados através de uma linha de *reset*. Outros tipos de inicialização da máquina de estados são possíveis em VHDL, de uma forma implícita, em que a variável de estado fica inicializada a um valor de defeito ou explicitamente quando inicializada na sua declaração. Estes tipos de inicialização, apropriados para a simulação VHDL, quando sintetizados produzem circuitos indesejados em que o estado inicial após se ligar o sistema é imprevisível.

- Utilização de construções `if-then-else` no processo combinatório de forma a especificar completamente para todos os estados qual o valor de cada saída e do próximo estado. Evita-se assim o aparecimento de *latches* que irão inicialmente para um valor lógico desconhecido e podem posteriormente levar o sistema a evoluir para estados indesejáveis[7].

7 Conclusões

A representação de circuitos integrados num nível de abstracção superior ao nível lógico, permite ao projectista lidar com o aumento de complexidade, e dimensão dos actuais circuitos integrados. A descrição do circuito ao nível lógico pode ser obtida de uma forma "automática" através de ferramentas de síntese.

As linguagens de descrição de *hardware* são a forma mais comum e apropriada para representar um circuito digital a alto nível. Das linguagens de descrição de *hardware* mais conhecidas destaca-se o VHDL por ser uma linguagem pública, adoptada como norma por parte do DoD/IEEE, e estar amplamente divulgada.

[7]Note-se que tanto este, como o problema do ponto anterior podem ser detectados na simulação lógica efectuada depois da síntese. Em ambos os casos, e considerando que no início da simulação todos os nós do circuito são inicializados a um valor lógico desconhecido 'X', teremos sempre ao longo da simulação alguns nós que irão permanecer em 'X'.

Na metodologia de projecto apresentada, a ferramenta de síntese é aquela que impõe maiores restrições à descrição do circuito. A "simples" definição de um subconjunto de instruções de VHDL sintetizáveis não é suficiente para garantir a qualidade do circuito final. O estilo de descrição utilizado permite influenciar de forma decisiva o processo de síntese.

A escolha de quais os melhores estilos de descrição a adoptar num ambiente de CAD é assunto de investigação futura. Estes poderão não só depender do tipo de circuito a sintetizar (combinatório ou sequencial síncrono), como também dos objectivos pretendidos para este, em termos de área e/ou velocidade.

Referências

[Augustin 89] Larry M. Augustin. Timing Models in VAL/VHDL. In *IEEE International Conference on Computer-Aided Design*, pages 122–125, 1989.

[Carlson 90] Steve Carlson. *Introduction to HDL-Based Design Using VHDL.* Synopsys, Inc., 1990.

[Carlson 91] Steve Carlson, editor. *Synopsys Methodology Notes*, volume 1. Synopsys, Inc., September 1991.

[CRM 91] Synopsys, Inc. *VHDL Compiler Reference Manual*, October 1991. Version 2.2.

[Gajski 83] D. Gajski and R. Kuhn. Guest Editors' Introduction: New VLSI Tools. *IEEE Computer*, 16(12):11–14, December 1983.

[Gajski 92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis.* Kluwer Academic Publishers, 1992.

[Levitan 89] S. Levitan, A. Martello, R. Owens, and M. Irwin. Using VHDL as a Language for Synthesis of CMOS VLSI Circuits. In *Proceedings of the Ninth IFIP Symposium on Computer Hardware Description Languages and their Applications*, June 1989.

[Lipsett 90] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1990.

[LRM 88] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual.*, March 1988. IEEE Std 1076-1987.

[McFarland 90] M. McFarland, A.C. Parker, and R. Camposano. The High-Level Synthesis of Digital System. *Proceedings of the IEEE*, 78(2), February 1990.

[Michel 92] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[Shiva 79] S. Shiva. Computer Hardware Description Languages - A Tutorial. In *Proceedings of the IEEE*, volume 67, December 1979.

[Synthesis 91] Synopsys, Inc. *Design Compiler Reference Manual*, October 1991. Version 2.2.

[UDL 92] Japan Electronic Industry Development Association. *UDL/I - Languge Reference*, May 1992. Version 1.0m.

[Waxman 86] R. Waxman. Hardware Design Languages for Computer Design and Test. *IEEE Design & Test of Computers*, 19(4), April 1986.

[Yasuura 89] H. Yasuura and N. Ishiura. Semantics of a Hardware Design Language for Japonese Standardization. In *Proceedings of the 26th Design Automation Conference*, 1989.