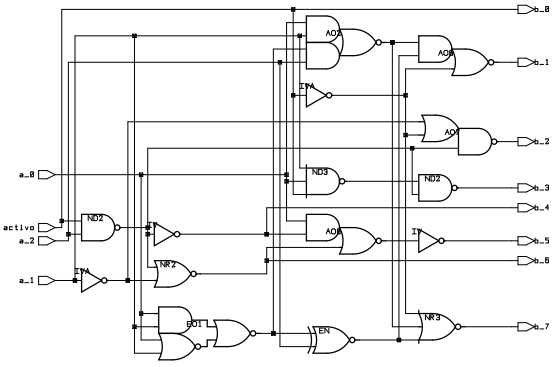
UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

```
entity desc_nivel is
    port(a: in integer range 0 to 7;
         activo: in bit;
         b: out bit_vector(7 downto 0));
end desc_nivel;

architecture desc of desc_nivel is
begin
    process(a, activo)
    begin
        b <= (others => '0');
        if activo = '1' then
            for i in b'range loop
                if i <= a then
                    b(i) <= '1';
                end if;
            end loop;
        end if;
    end process ;
end desc;
```



Especificação Funcional de Sistemas Electrónicos Digitais em Ambiente de Síntese

Paulo Ferreira Godinho Flores

Licenciado

Dissertação para obtenção do grau de mestre em:

Eng. Electrotécnica e de Computadores

Junho de 1993

Tese realizada sob a supervisão do
Doutor Eng. Horácio Cláudio de Campos Neto
Professor Auxiliar do Departamento de
Engenharia Electrotécnica e de Computadores
INSTITUTO SUPERIOR TÉCNICO

Resumo

O projecto de circuitos integrados num ambiente de desenho assistido por computador começa tradicionalmente pela utilização de um editor de esquemáticos. Desta forma, é fornecida ao sistema de produção uma representação, a nível lógico, do circuito a implementar. Com o aumento da complexidade e dimensão dos circuitos a projectar, este método torna-se demasiado moroso e passível de erros. Apenas a especificação do circuitos a níveis de abstracção superior ao nível lógico, possibilita ao projectista lidar com a complexidade dos actuais circuitos.

Através da utilização de linguagens de descrição de *hardware*, forma mais comum de especificar circuitos a níveis de abstracção elevados, é possível descrever o comportamento de circuitos sem a preocupação com os pormenores de realização no nível lógico. Esta representação permite ainda verificar a sua funcionalidade (simulação) e realizar "uma transformação automática" de forma a obter o circuito representado num conjunto de células de uma dada biblioteca (síntese). No entanto, o desempenho do circuito sintetizado nem sempre é o desejado, estando este dependente quer da qualidade da ferramenta de síntese, quer do estilo de descrição adoptado.

Neste trabalho apresenta-se uma metodologia de projecto de circuitos digitais num ambiente de síntese, que recorre à linguagem norma do IEEE (VHDL) para a especificação dos circuitos. Especial destaque é dado à forma como deve ser feita uma descrição em VHDL para que esta seja sintetizável, e para que o circuito dela obtido tenha o desempenho desejado. É apresentado um subconjunto de instruções sintetizáveis e proposto um estilo de descrição que, embora de algum modo restritivo, procure "garantir" a qualidade do circuito final. A metodologia de projecto apresentada e o estilo de descrição proposto foram validados através do desenvolvimento de uma biblioteca de modelos em VHDL e do projecto de um circuito digital de média dimensão.

Abstract

The project of integrated circuits in a computer aided design environment traditionally begins with the use of a schematic editor. In this way, a representation, at the logic level, of the circuit to be implemented is supplied to the production system. With the growing complexity and dimension of the circuits to be designed, this method tends to be time consuming and error prone. Only the specification of circuits in levels of abstraction higher than the logic level, allows the designer to deal with the complexity of the actual circuits.

The use of hardware description languages is the most common way to specify circuits at an higher level of abstraction, making possible the description of the behavior of the circuit without concern about the details of the logic level implementation. This representation also permits a verification of its functionality (simulation) and the realization of "an automatic transformation" in order to obtain the circuit represented as a set of cells of a given library (synthesis). Nevertheless, the performance of the synthesized circuit depends both on the quality of the synthesis tool and on the style of description used.

In this work a methodology for the design of digital circuits in a synthesis environment is presented, using the IEEE standard language (VHDL) for the specification of the circuit. Special attention is given to how a VHDL description should be written so that it can be synthesized, and so that the resulting circuit has the required performance. A subset of VHDL instructions which can be used in a synthesis environment is presented and a description style proposed which, although restrictive, it tries to "guarantee" the quality of the final circuit. The design methodology presented and the style of description proposed were validated through the development of a VHDL library and the design of a medium-sized digital circuit.

Palavras chave

Circuitos integrados digitais

Síntese automática

Linguagens de descrição de hardware

VHDL

Estilos de descrição para síntese

Biblioteca sintetizável de circuitos

Keywords

Digital integrated circuits

Automatic synthesis

Hardware description languages

VHDL

Description styles for synthesis

Synthesized library of circuits

Agradecimentos

A realização desta tese e do trabalho nela desenvolvido, não teria sido possível sem a colaboração de algumas pessoas às quais desejo expressar o meu agradecimento.

Ao meu orientador, Professor Horácio Neto, pelo seu empenho, encorajamento e crítica exigente na realização desta tese. Por me ter proporcionado oportunidades únicas de valorização profissional e pessoal.

Aos Engenheiros Luis Miguel Silveira e João Paulo Silva pela amizade e por terem despertado em mim o interesse pela área de Microelectrónica.

Aos meus colegas, Engenheiros José Carlos Monteiro, Jorge Ribeiro Fernandes, José Teixeira de Sousa, João Paulo Piteira e Leonel Simões pela amizade, apoio e ajuda nos diversos momentos da realização desta tese.

Às pessoas que constituem o grupo de VLSI do INESC e, em particular as do grupo de Simulação, pela compreensão e ambiente de trabalho proporcionado.

Ao meus pais, familia e amigos pela compreensão e constante apoio demonstrado ao longo do desenvolvimento deste trabalho.

Finalmente, à Candy, pelo seu apoio, compreensão e encorajamento nos momentos mais difíceis.

Lisboa, Junho de 1993

Conteúdo

Resumo **i**

Abstract **iii**

Agradecimentos **vii**

Conteúdo **ix**

Lista de Figuras **xiii**

Lista de Tabelas **xvii**

1 Introdução **1**
1.1 Síntese de Circuitos Digitais 2
1.2 Breve Descrição do Trabalho Desenvolvido 4
1.3 Organização da Tese 5

2 Síntese e Sistemas de Síntese **7**
2.1 Níveis de Representação de um Circuito 7
2.2 Diferentes Níveis de Síntese 8
2.2.1 Síntese de alto-nível 10
2.2.2 Síntese a nível de Transferência de Registos 12
2.2.3 Síntese a nível lógico 14
2.3 Conclusões 18

3 A Linguagem VHDL **21**
3.1 As Linguagens de Descrição de *hardware* 21
3.2 A Evolução do VHDL 29
3.2.1 O Passado 29
3.2.2 O Presente 29
3.2.3 O Futuro 31
3.3 Principais Características da Linguagem VHDL 32

3.3.1 Unidades de Projecto em VHDL . . . 32
3.3.2 Objectos, Tipos de Dados e Atributos . . . 35
3.3.3 O Modelo Temporal . . . 36
3.3.4 As Instruções . . . 37
3.4 Conclusões . . . 39

4 VHDL para Síntese 41
4.1 Fluxo de Projecto . . . 41
4.2 Construções de VHDL Sintetizáveis . . . 44
4.2.1 Exemplos de como as Descrições de VHDL são Interpretadas 49
4.3 Estilos de Descrição em VHDL . . . 63
4.4 Interacção com a Ferramenta de Síntese . . . 76
4.5 Conclusões . . . 80

5 Uma Biblioteca de Modelos - Blocos Fundamentais 81
5.1 Unidades Aritméticas e Lógicas . . . 82
5.2 Registos . . . 87
5.3 Descodificadores e Codificadores . . . 87
5.4 Multiplexers . . . 91
5.5 Contadores . . . 92
5.6 Blocos com Alta Impedância . . . 94
5.7 Conclusões . . . 98

6 Projecto de um Circuito de Criptografia Usando VHDL 103
6.1 Objectivo do Circuito . . . 103
6.2 Ambiente de Desenvolvimento . . . 105
6.3 Os Blocos Constituintes do Circuito . . . 106
6.3.1 O módulo `tipos` . . . 107
6.3.2 Conversor série-paralelo . . . 108
6.3.3 Operação de criptografia . . . 110
6.3.4 Memória . . . 112
6.3.5 Máquina de criptografia . . . 116
6.3.6 Conversor paralelo-série . . . 120
6.3.7 Controlador . . . 123
6.4 O Circuito Final . . . 128
6.5 Conclusões . . . 136

7 Conclusões e Trabalho Futuro 137
7.1 Conclusões . . . 137
7.2 Desenvolvimentos Futuros . . . 139

A O módulo `mvl9` **141**

B O módulo `arithmetic` **159**

Bibliografia **181**

Lista de Figuras

2.1 Domínios e níveis de representação de um sistema. 9
2.2 Diferentes tarefas envolvidas num sistema de síntese. 10
2.3 Exemplo da operação de reoptimização temporal sobre uma descrição RTL. 14
2.4 Exemplo de algumas regras do sistema SOCRATES. 18

3.1 Exemplo da descrição de um circuito em HardwareC. 23
3.2 Exemplo da descrição de um circuito em M. 24
3.3 Exemplo da descrição de um circuito em ELLA. 25
3.4 Exemplo de uma descrição de um circuito em Verilog. 26
3.5 Exemplo de uma descrição de um circuito em UDL/I. 27
3.6 Exemplo da descrição de um circuito em VHDL 28
3.7 Exemplo de uma entidade: declaração dos acessos de um somador. . 32
3.8 Descrição estrutural do somador. 33
3.9 Descrição de fluxo de dados do somador. 33
3.10 Descrição comportamental do somador. 34
3.11 Configuração do somador. 35

4.1 Fluxo de projecto 42
4.2 Utilizações válidas da instrução `wait` num ambiente de síntese. . . . 46
4.3 a) Código VHDL; b) Simulação VHDL; c) Circuito sintetizado; d) Simulação lógica. 47
4.4 Resultado da síntese de uma atribuição concorrente em VHDL. . . . 50
4.5 Instanciação de componentes e uso de módulos. 51
4.6 Selecção dum elemento de um vector. 52
4.7 Uso do operador aritmético '`+`' com inteiros. 54
4.8 Síntese de um `if-then-else` dentro de um processo. 55
4.9 Síntese de um `if` sem o ramo `else`. 55
4.10 Uso do ciclo `for` recorrendo ao atributo `range` 56
4.11 Uso da instrução `wait` para a geração de circuitos sequenciais. . . . 57
4.12 Síntese de um *flip-flop* com *reset* assíncrono. 58

4.13 Síntese de um *flip-flop* com *reset* síncrono. 59
4.14 Contador de módulo 10 com *reset* assíncrono. 60
4.15 Máquina de estados de Moore que detecta a sequência `"111"`. . . . 61
4.16 Máquina de estados de Mealy que detecta a sequência `"111"`. 62
4.17 Descrição de um *buffer* com alta impedância usando lógica multi-valor. 64
4.18 Descrição de um *flip-flop* D que não é sintetizável. 68
4.19 O uso de parêntesis como forma de estruturar o circuito. 69
4.20 Uso de adições desnecessárias produz circuitos maiores. 70
4.21 Evidenciando factores consegue-se reduzir a área do circuito. 71
4.22 Descrição de um conversor do código BCD para um mostrador de sete segmentos. 72
4.23 Descrição de um conversor do código BCD para um mostrador de sete segmentos usando lógica multi-valor. 73
4.24 Máquina de estados "mal" sintetizada devido ao estilo de descrição usado. 75
4.25 Descrição de uma função aritmética. 77
4.26 Circuito sintetizado sem intervenção do projectista. 78
4.27 Circuito sintetizado com a intervenção do projectista. 78

5.1 Descrição de um somador utilizando inteiros. 83
5.2 Descrição de um somador utilizando vectores de *bits*. 84
5.3 Descrição de um comparador de dois inteiros positivos. 85
5.4 Descrição da unidade aritmética `u_logica`. 86
5.5 Registo de oito *bits* com *reset* assíncrono. 88
5.6 Descrição de um registo multi-função (com deslocamento e carregamento paralelo). 89
5.7 Descrição de um descodificador de oito *bits*. 91
5.8 Descrição do descodificador de oito *bits* com "memória". 92
5.9 Descrição do codificador de oito para três *bits* com prioridade. . . . 93
5.10 Descrição de um multiplexer de 8 para 1 de um *bit*. 94
5.11 Descrição de um desmultiplexer de 1 para 8. 95
5.12 Descrição de um multiplexer de 2 para 1 de oito *bits*. 96
5.13 Descrição de um contador com carregamento paralelo. 97
5.14 Descrição de um contador de módulo de contagem "programável". . 98
5.15 Descrição de um bloco de *buffers* com saídas em alta impedância. . 99
5.16 Descrição de um *buffer* de oito *bits* bi-direccional. 100
5.17 Descrição de um *buffer* de oito *bits* de interface a um *bus* bi-direccional. 101

6.1 Fluxo de dados do circuito de criptografia. 106
6.2 Descrição do módulo `tipos`. 108
6.3 Descrição do conversor série-paralelo. 109
6.4 Simulação VHDL e lógica do conversor série-paralelo. 110
6.5 Circuito sintetizado para o conversor série-paralelo. 110
6.6 Descrição da operação de criptografia. 111
6.7 Circuito que realiza a operação de criptografia. 111
6.8 Descrição da memória que armazena as palavras da chave. 113
6.9 Simulação VHDL e lógica do bloco de memória. 114
6.10 Circuito sintetizado para o bloco de memória. 115
6.11 Descrição da máquina de estados do criptógrafo. 118
6.12 Simulação VHDL e lógica da máquina de estados de criptografia. . . 119
6.13 Circuito sintetizado para a máquina de criptografia. 120
6.14 Descrição do conversor paralelo-série. 121
6.15 Simulação VHDL e lógica do conversor paralelo-série. 122
6.16 Circuito sintetizado para o conversor paralelo-série. 123
6.17 Descrição (estrutural) do controlador do circuito. 124
6.18 Descrição da máquina de estado do controlador do circuito. 125
6.19 Descrição do "condicionador" das saídas do controlador. 126
6.20 Simulação VHDL e lógica do controlador. 127
6.21 Circuito sintetizado para o controlador. 128
6.22 Descrição estrutural do circuito de criptografia. 130
6.23 Descrição da unidade de teste para o circuito de criptografia. 132
6.24 Simulação VHDL do circuito de criptografia utilizando a unidade de teste. 133
6.25 Circuito de criptografia no nível hierárquico mais elevado. 133
6.26 Circuito lógico do criptógrafo. 135
6.27 Simulação VHDL e lógica do circuito de criptografia. 135

Lista de Tabelas

4.1 Comparação do suporte de algumas construções de três sistemas comerciais de síntese. 49
4.2 Tabela de transição da máquina de estados (máquina de Moore). . . 57
4.3 Tabela de transição da máquina de estados (máquina de Mealy). . . 63
4.4 Principais formas de caracterizar um circuito. 79
4.5 Principais formas de restringir um circuito. 79
4.6 Diferentes compromissos entre a área ocupada e o atraso máximo . 80

5.1 Operações realizadas pela unidade aritmética `u_logica`. 83
5.2 Operações realizadas pelo registo multi-função. 87
5.3 Funcionamento do descodificador. 90
5.4 Funcionamento do descodificador com "memória". 90
5.5 Funcionamento de codificador de oito para três *bits* com prioridade. 94
5.6 Funcionamento do *buffer* bi-direccional. 96
5.7 Funcionamento do *buffer* de interface a um *bus* bi-direccional 98

6.1 Evolução dos estados da máquina de criptografia 104
6.2 Área e atrasos do bloco `mem` quando optimizado com diferentes restrições. 116
6.3 Diferentes áreas e atrasos obtidos para a máquina de estado de criptografia. 120
6.4 Tabela de transições de estados do controlador. 126
6.5 Diferentes áreas e atrasos obtidos para o controlador. 128
6.6 Diferentes áreas e atrasos obtidos para o criptógrafo. 134

Capítulo 1

Introdução

O aumento da utilização de circuitos integrados nas últimas décadas, em especial os circuitos integrados de aplicação específica (ASICS[1]), deve-se não só ao aparecimento de novas metodologias de fabrico (*gate-array, standard-cell, programmable gate-array*) como também, ao uso de ferramentas de CAD que permitiram aumentar a produtividade do projectista e reduzir o tempo de desenvolvimento de um circuito integrado.

A utilização de ferramentas de colocação e encaminhamento[2] automáticas a partir de finais dos anos 70 permitiu ao projectista abstrair-se da implementação física do circuito, passando a realizar o seu projecto a um nível de abstracção superior: o nível lógico. Tradicionalmente, tem sido este o nível no qual o projecto de circuitos digitais é realizado, recorrendo ao uso de editores de esquemáticos para a introdução do circuito num sistema de produção, e a simuladores lógicos para verificação da sua funcionalidade e desempenho.

No entanto, com o aumento da dimensão e complexidade dos circuitos a projectar, resultante de uma maior capacidade de integração do número de transistores num único integrado, este método mostrou-se demasiado moroso e passível de erros. Assim, tornou-se cada vez mais premente a necessidade do projecto de circuitos integrados ser realizado a um nível de abstracção superior ao nível lógico.

Desde o início dos anos 80 que as linguagens de programação, adaptadas ou específicas para a descrição de *hardware*, vêm sendo usadas para a descrição, documentação e simulação de circuitos digitais. Porém, a sua utilização requeria que

[1]Do inglês *Application Specific Integrated Circuits*.

[2]Tradução do inglês de *placement* e *routing* respectivamente.

o projecto do circuito fosse realizado duas vezes: inicialmente usando uma linguagem de descrição para verificar a sua funcionalidade, e depois no nível lógico para ser introduzido, através de um editor de esquemáticos, no sistema de produção. Em meados dos anos 80, com o desenvolvimento da tecnologia de síntese, começou a ser possível obter de uma forma "automática" a representação do circuito ao nível lógico, a partir da sua descrição de "alto nível" utilizando uma linguagem de descrição de *hardware*. Esta forma de descrever o circuito integrado permite ao projectista usar construções semelhantes às das linguagens algorítmicas, abstraindo-se dos pormenores de implementação do nível lógico.

A representação do circuito numa linguagem de descrição de *hardware* é o primeiro passo a ser dado no projecto de circuitos integrados num ambiente de síntese. A utilização de uma linguagem de descrição de *hardware* complexa e poderosa como o VHDL[3], requer do projectista um profundo conhecimento não só da própria linguagem, para aproveitar todas as suas potencialidades, mas também, das restrições impostas pelas ferramentas de síntese à descrição do circuito, para que esta seja sintetizável e se obtenham circuitos com o desempenho desejado. A descrição de um circuito de uma forma menos própria resulta em geral numa implementação não optimizada ou funcionalmente incorrecta, independentemente da capacidade e qualidade da ferramenta de síntese usada. É neste sentido que este trabalho propõe um conjunto de directivas que o projectista deve seguir para obter o maior rendimento desta nova tecnologia.

1.1 Síntese de Circuitos Digitais

O projecto de um circuito integrado começa geralmente por uma especificação de alto nível do seu comportamento, acompanhada de algumas restrições que devem ser satisfeitas (por exemplo, a área máxima que o circuito deve ocupar, o número de pinos disponíveis ou os atrasos mínimos/máximos de alguns sinais).

O processo de síntese de um circuito digital tem como objectivo obter, a partir da sua representação de alto nível, o circuito óptimo mapeado em portas lógicas de uma dada tecnologia e satisfazendo o conjunto de restrições impostas. Sendo este processo demasiado complexo para ser realizado numa única etapa, a síntese de circuitos digitais utiliza habitulamente uma sequência de passos intermédios, correspondentes aos diferentes níveis de abstracção em que um circuito pode ser

[3] *VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.*

representado. Assim, consoante os passos realizados é possível classificar a síntese em três categorias: síntese de alto-nível, síntese a nível de transferência de registos e síntese a nível lógico. A síntese de alto-nível determina, a partir de uma descrição algorítmica sem referências temporais, quais as operações que devem ser efectuadas em cada intervalo de tempo (período de relógio). A síntese a nível transferência de registos (RTL[4]) obtém uma representação lógica do circuito partindo de uma descrição que define, para cada ciclo de relógio, qual o seu comportamento. Finalmente, a síntese a nível lógico obtém o circuito mapeado em portas lógicas de uma dada tecnologia a partir da sua representação lógica, por exemplo, através de funções booleanas.

Actualmente as técnicas de síntese de alto-nível não se encontram suficientemente desenvolvidas, pelo que são apenas usadas em sistemas de investigação, não existindo praticamente nenhuma ferramenta comercial que as aplique. Porém, as técnicas de síntese a nível RTL e lógico já se encontram suficientemente "maduras", ao ponto de existirem várias ferramentas comerciais que as utilizam.

A utilização de ferramentas de síntese no projecto de circuitos digitais requer a transformação da especificação do circuito numa representação que possa ser interpretada por aquelas. São vários os formatos em que essa representação pode ser feita: texto, gráficos, tabelas e/ou formas de onda. Porém, a forma mais divulgada e que permite a representação de um circuito a vários níveis de abstracção é feita no formato textual, utilizando uma linguagem de descrição de *hardware*.

As linguagens de descrição de *hardware* são geralmente baseadas nas linguagens de programação algorítmicas de alto nível, mas têm características específicas para a descrição de *hardware* (por exemplo, noções temporais, objectos e tipos de dados próprios). A maioria destas linguagens permite que o comportamento do circuito descrito seja simulado, sem necessidade de obter uma representação deste no nível lógico. Assim, é possível detectar e corrigir, na fase inicial do projecto, alguns problemas que só se manifestariam quando da verificação da funcionalidade do circuito ao nível lógico. A utilização de linguagens de descrição de *hardware* permite ainda a documentação do circuito de uma forma independente da tecnologia, pois a escolha do fabricante (e portanto o compromisso com uma dada tecnologia) só é feita numa fase posterior do projecto.

A escolha da linguagem de descrição de *hardware* a adoptar num ambiente de síntese de circuitos integrados deve ter em conta, entre outros factores, a capacidade que essa linguagem tem para a descrição de circuitos e a existência de

[4]Do inglês *Register Transfer Level*.

ferramentas de simulação e de síntese para a sua utilização. A linguagem VHDL foi desenvolvida para a descrição de circuitos digitais tendo como principal objectivo a sua documentação, modelação e simulação. Porém, a adopção desta linguagem como *a* linguagem de descrição de *hardware* norma do IEEE[5], fez com que a sua utilização fosse alargada a outras áreas de investigação (por exemplo, síntese, verificação formal e teste). Actualmente, esta linguagem encontra-se amplamente divulgada na comunidade internacional de microelectrónica, existindo várias ferramentas de síntese e simulação que a aceitam como forma de representação de um circuito digital.

1.2 Breve Descrição do Trabalho Desenvolvido

Neste trabalho propõe-se uma metodologia de projecto para o desenvolvimento de circuitos integrados num ambiente de síntese. Esta, permite aumentar a rapidez com que se obtém um circuito ao nível lógico, tendo em conta as várias etapas do fluxo de projecto normalmente usado num ambiente de síntese.

Este trabalho envolveu inicialmente o estudo da linguagem VHDL e a análise da sua utilização na descrição de circuitos num ambiente de síntese. Tendo esta linguagem sido criada a pensar na descrição e simulação de circuitos, mas não na sua síntese, existem construções que não são sintetizáveis ou cuja utilização é restringida de alguma forma. O subconjunto de instruções que são sintetizáveis não está ainda normalizado, apresentando-se por isso a lista daquelas que são suportadas, com ou sem restrições, pela maioria das actuais ferramentas de síntese.

A descrição do circuito usando uma linguagem de descrição de *hardware* tem um papel fundamental no processo de síntese devido ao impacto que pode ter na implementação final do circuito. A simples utilização de construções sintetizáveis não garante que uma descrição seja sintetizável, nem que se o for, que o circuito lógico obtido seja "óptimo". Assim, foram analisados com algum detalhe os efeitos que uma descrição menos cuidada pode ter na eficiência da implementação do circuito no nível lógico, sendo proposto um conjunto de regras claras sobre o modo de descrição que deve ser seguido para garantir a qualidade do circuito sintetizado.

Com o objectivo de facilitar ao projectista a obtenção de uma descrição sintetizável de um circuito, foi ainda descrita uma biblioteca de modelos em VHDL. Sendo esta biblioteca destinada a um ambiente de síntese, a sua concepção teve

[5] *Institute of Electrical and Electronics Engineers*.

em consideração as "restrições" apresentadas para a descrição de circuitos sintetizáveis.

Finalmente, é apresentado um exemplo de um projecto de um circuito de média dimensão, cuja realização seguiu a metodologia de projecto proposta. A partir de uma descrição textual da funcionalidade desejada para o circuito, foram realizados os vários passos de projecto (referidos ao longo deste trabalho) para obter a representação do circuito no nível lógico com o desempenho desejado. A utilização da metodologia de projecto proposta permitiu obter facilmente uma descrição do circuito sintetizável cuja implementação lógica correspondente apresentasse o comportamento pretendido, o que foi verificado através da simulação lógica do circuito sintetizado.

1.3 Organização da Tese

Este trabalho está dividido em sete capítulos. Neste primeiro capítulo pretendeu-se motivar o leitor para a importância da utilização de síntese automática em projecto de circuitos digitais e enquadrar o trabalho realizado no processo de desenvolvimento de circuitos electrónicos.

No próximo capítulo são referidos os vários domínios de representação de um dado circuito e são descritas as principais tarefas de síntese correspondentes a cada um dos níveis de abstracção.

No capítulo 3 são apresentadas as principais características das linguagens de descrição de *hardware* e referidas de forma breve algumas destas linguagens. São focados ainda os principais conceitos do VHDL, sem no entanto entrar nos pormenores sintácticos ou semânticos desta linguagem. Estes, poderão ser encontrados em [Lipsett 90, Perry 91, Bergé 92] ou no manual de referência da linguagem [LRM 88].

No capítulo 4 é descrito o fluxo de projecto usado num ambiente de síntese, é definido um conjunto de construções de VHDL sintetizáveis e é proposto um conjunto de regras de descrição que permitem encurtar o tempo de projecto e procuraram garantir um circuito final de melhor qualidade. Exemplifica-se ainda a influência que a interacção do projectista com a ferramenta de síntese pode ter no circuito final.

No capítulo 5 é desenvolvida uma biblioteca de modelos para síntese com base

nas regras definidas no capítulo anterior. Esta biblioteca permite que uma descrição sintetizável de um circuito possa ser obtida de uma forma mais eficiente pelo projectista.

No capítulo 6 são apresentados os resultados da síntese de um circuito digital de média dimensão partindo da sua representação em VHDL. Esta descrição foi obtida seguindo a metodologia proposta, o que reduziu significativamente o tempo de desenvolvimento necessário para obter o circuito sintetizado no nível lógico.

Finalmente no capítulo 7 são apresentadas as conclusões do trabalho desenvolvido e perspectivado o trabalho futuro.

Capítulo 2

Síntese e Sistemas de Síntese

Neste capítulo apresentam-se os vários níveis de abstracção em que um circuito digital pode ser representado e as diferentes tarefas que envolve a síntese de circuitos digitais. Referem-se ainda certos sistemas de síntese que realizam algumas destas tarefas.

2.1 Níveis de Representação de um Circuito

A representação de um circuito digital pode ser feita em diferentes níveis de abstracção, consoante o grau de detalhe pretendido para descrever o circuito. Quando este é descrito por um conjunto de memórias e unidades de processamento que comunicam entre si através de mensagens, está-se a representar o circuito ao nível de sistema. No nível algorítmico, o circuito é descrito através de um algoritmo que determina como as saídas são obtidas em função das entradas. No nível de transferência de registos (RTL) a representação do circuito é vista como um conjunto interligado de blocos funcionais e registos. Este nível de abstracção difere do nível algorítmico pelo detalhe com que é especificada a estrutura do algoritmo. Enquanto que no nível RTL as variáveis representam registos, as atribuições transferência entre registos e os operadores blocos funcionais, no nível algorítmico essa estrutura não têm significado. A descrição do circuito no nível lógico é feita em termos de portas e funções lógicas. Este é o nível em que habitualmente é realizada a descrição de um circuito digital (através de captura esquemática) para um sistema de produção de circuitos integrados. Se se quiser utilizar uma descrição mais detalhada, é possível descer ao nível eléctrico onde são

utilizados componentes eléctricos e equações matemáticas para descrever o circuito.

Em 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 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 ilustradas na figura 2.1. Nesta, os diferentes níveis de abstracção são representados por circunferências e os domínios de representação pelos eixos. Na intercepção de cada um eixos com as circunferências são indicados quais os elementos envolvidos nas representações do circuito (para um dado nível de abstracção e domínio de representação). Por exemplo, a descrição de um circuito ao nível de transferência de registos no domínio estrutural é feita à custa de registos, multiplexers e unidades aritmética e lógicas.

2.2 Diferentes Níveis de Síntese

O processo de síntese ideal consistiria numa única transformação realizada sobre a descrição comportamental de um circuito, feita num elevado nível de abstracção (por exemplo, nível algoritmo ou de sistema), para obter a representação desse circuito num nível de abstracção inferior (nível lógico ou eléctrico), mapeado numa dada tecnologia e satisfazendo um conjunto de restrições impostas (área máxima, velocidade mínima de funcionamento, etc). No entanto, devido à complexidade que este processo apresenta, ele é dividido em várias transformações mais pequenas, correspondentes ao vários níveis de abstracção intermédios em que um circuito pode ser representado. Como em cada um destes níveis o detalhe com que o circuito é descrito é diferente, as tarefas de síntese a realizar de forma a obter o circuito "óptimo" no nível de abstracção inferior são também diferentes. O agrupamento destas tarefas pelos diferentes níveis de síntese não está ainda perfeitamente definido. Por exemplo, há autores que consideram a tarefa de síntese de máquinas de estados como pertencente ao nível lógico [Allen 90], enquanto outros a consideram como fazendo parte do nível RTL [Michel 92].

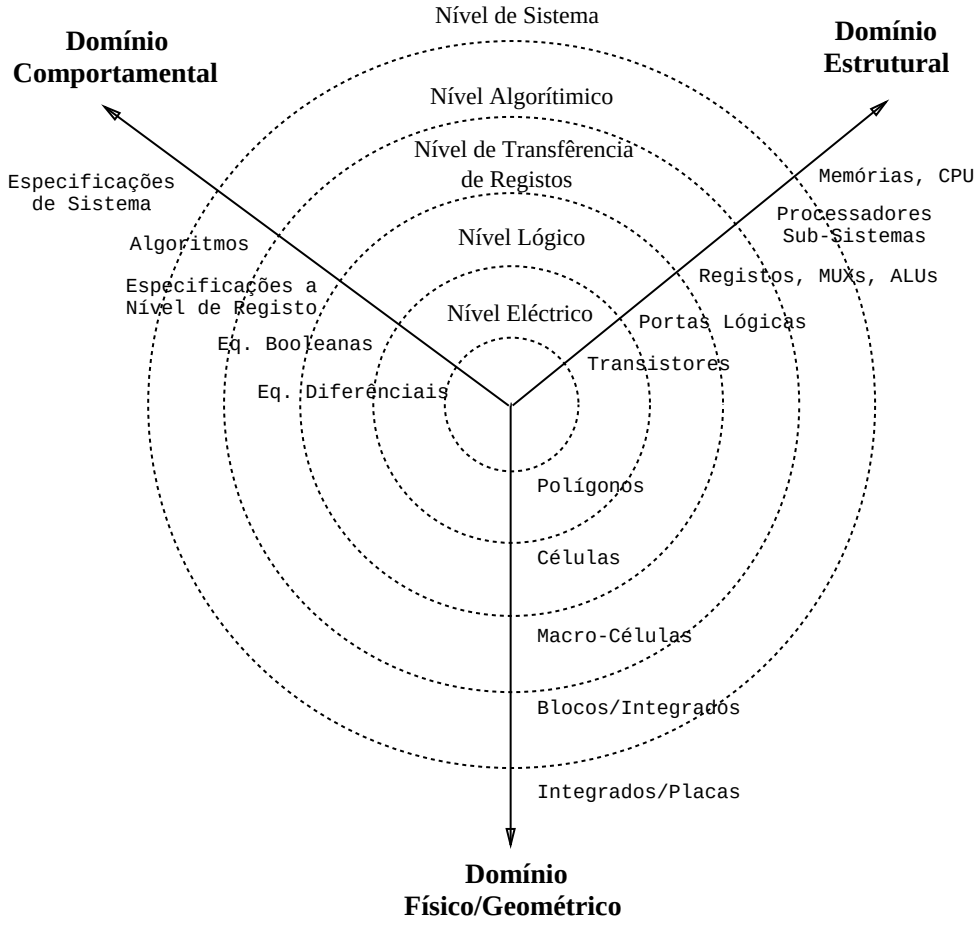


Figura 2.1: Domínios e níveis de representação de um sistema.

O nível de abstracção em que o circuito é obtido após a síntese está dependente do estilo de projecto usado para o fabrico do circuito. Num estilo de projecto personalizado[1], em que todas as máscaras necessárias para o fabrico do circuito têm de ser obtidas, é necessário obter uma representação final do circuito ao nível eléctrico. Se for utilizado um estilo de projecto semi-personalizado[2], onde as máscaras são previamente determinadas pelo fabricante de circuitos integrados, é suficiente obter a representação do circuito ao nível lógico. Esta representação é habitualmente constituída por uma lista de portas que têm uma correspondência unívoca com as células disponíveis numa dada tecnologia. Por ser este último estilo, o mais utilizado para o fabrico de circuitos integrados digitais para aplicações específicas (ASICS), neste trabalho apenas será considerada a síntese até ao nível lógico.

Na figura 2.2 apresentam-se as várias tarefas em que se subdivide habitualmente

[1]Em inglês *full-custom*.

[2]Em inglês *semi-custom*.

o processo de síntese tendo em atenção os vários níveis de abstracção envolvidos.

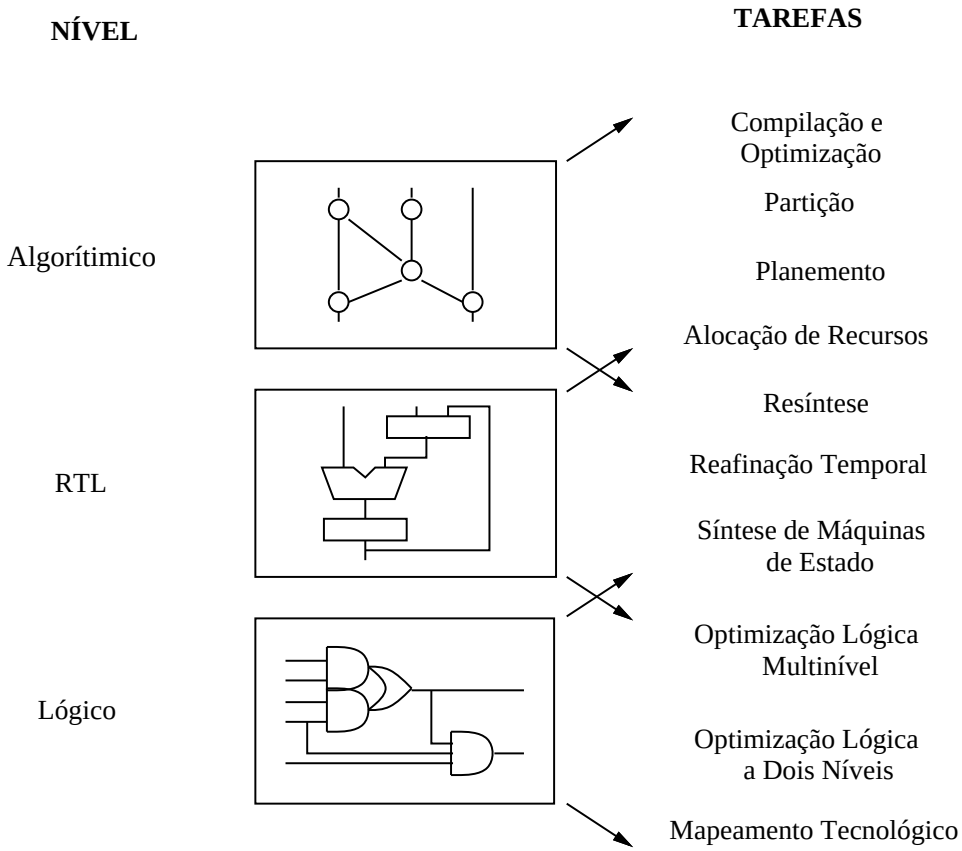


Figura 2.2: Diferentes tarefas envolvidas num sistema de síntese.

2.2.1 Síntese de alto-nível

A síntese de alto-nível é também por vezes designada por síntese comportamental ou algorítmica pois tem como ponto de partida uma especificação do comportamento do circuito geralmente descrita através de algoritmos. Nestes são definidos, de forma similar às linguagens de programação de alto-nível, quais os procedimentos necessários para obter as saídas em função das suas entradas.

A síntese de alto-nível consiste no mapeamento desses algoritmos num conjunto de registos, operadores e estruturas de controlo. A estrutura do circuito é obtida à custa das tarefas que a seguir se descrevem:

Compilação e Optimização - O processo de compilação, idêntico ao realizado nas linguagens de programação de alto-nível, pretende obter uma representação interna do algoritmo descrito numa forma mais conveniente para a

síntese. Este passo inicial tem também a vantagem de permitir que as tarefas seguintes de síntese sejam independentes da linguagem de descrição usada para especificar o circuito.

Após ser obtida a representação interna do sistema, são realizadas algumas transformações de alto-nível. Estas consistem na optimização da descrição anteriormente obtida recorrendo a técnicas utilizadas em *software* (eliminação de código desnecessário, substituição de chamadas a subprogramas pelo seu código, explicitação de ciclos, etc) e/ou a técnicas específicas do *hardware* (selecção de operadores apropriados, aumento do número de operações realizadas ao mesmo tempo, criação de *pipelines*, etc). Algumas destas optimizações podem, e devem, vir explicitadas na descrição que é fornecida ao sistema de síntese.

Partição - A partição decompõe uma representação do circuito em várias mais pequenas com o objectivo de reduzir a complexidade das tarefas de síntese seguintes. Esta tarefa pode dividir o circuito com o fim de aumentar a rapidez de execução ou diminuir a área ocupada.

Planeamento - A tarefa de planeamento consiste na determinação e distribuição das operações de controlo pelos vários intervalos de tempo (períodos de relógio). Esta operação está restringida pelos recursos existentes. Por exemplo, a realização de duas somas pode ser feita num intervalo de tempo se existirem dois somadores ou em dois intervalos de tempo se existir um único somador.

Existe ainda um outro conjunto de restrições que são impostas pelo próprio *hardware* síncrono que se pretende sintetizar. Isto é, em cada intervalo de tempo um registo só pode ser carregado uma única vez, um *bus* só pode ter um valor e obviamente a lógica combinatória só pode determinar um único resultado.

A solução para o problema do planeamento (com restrições) pode ser determinada recorrendo a dois tipos de algoritmos: os que são restringidos pelo número de operações de controlo e os que são limitados pelo número de operadores que podem utilizar. O planeamento sem restrições é um problema NP-completo[3] cuja solução é obtida através de heurísticas.

Alocação de Recursos - Esta tarefa tem como objectivo determinar e atribuir quais os recursos de *hardware* (registos, operadores e *buses*) que irão ser ne-

[3]Designação que é dada aos problemas cujo tempo de computação para encontrar a solução exacta, cresce exponencialmente com a complexidade deste.

cessários para realizar uma dada função. Da mesma forma que o planeamento está dependente dos recursos alocados, a tarefa de alocação de recursos está dependente do planeamento previamente feito.

Esta forte interdependência entre o planeamento e alocação de recursos permite que a estrutura do circuito possa ser obtida quer se comece por uma ou outra tarefa. É mesmo possível entrelaçar as duas tarefas, com o objectivo de reduzir ao mínimo uma determinada função de custo, que dependerá do número de intervalos de tempo e dos recursos de *hardware* necessários.

Actualmente já existem alguns sistemas académicos que realizam a síntese de circuitos digitais a partir de uma descrição do seu comportamento feita através de um algoritmo. No entanto, na grande maioria desses sistemas não são integrados todos os níveis de síntese necessários para obter o circuito ao nível lógico.

O sistema OLYMPUS [Micheli 90] apresenta-se como o primeiro sistema de síntese de alto-nível completamente integrado. A partir de uma descrição do comportamento do circuito escrita em HardwareC[4] é possível obter uma implementação lógica do circuito que satisfaz um conjunto de restrições. Isto é conseguido à custa de dois módulos fundamentais no sistema: o Hercules, que é responsável pela compilação e optimização da descrição de HardwareC e o Hebe que faz o planeamento e alocação de recursos de forma a satisfazer as restrições de área e/ou velocidade impostas pelo projectista.

Existem outros sistemas de síntese de alto-nível, que são específicos para certos domínios de aplicação e/ou arquitecturas. Nestes, os algoritmos usados nas várias tarefas de síntese são alterados de forma a reflectirem as características desses domínios e/ou arquitecturas. O sistema de síntese CATHEDRAL [Man 90] por exemplo, é dedicado à síntese de circuitos para processamento digital de sinais, sendo constituído por vários subsistemas, cada um específico para a implementação numa dada arquitectura .

2.2.2 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

[4]Linguagem de programação similar ao C destinada a descrever circuitos (ver capítulo 3 onde são abordadas as linguagens de descrição de *hardware*).

de registos consiste pois em obter com essa descrição a implementação estrutural do circuito.

A primeira tarefa da síntese RTL consiste na optimização do fluxo de dados. Esta operação pode ser realizada recorrendo a duas técnicas diferentes: a resíntese e a reoptimização temporal[5]. Enquanto que a resíntese tem em conta as características específicas das unidades a utilizar para realizar a optimização, a reoptimização temporal "olha" para a estrutura do fluxo de dados para alterar a posição de certos registos ao longo da lógica combinatória (de forma a minimizar o ciclo de relógio e/ou o número de registos). Por exemplo, na figura 2.3 apresenta-se uma descrição RTL de um circuito que calcula a correlação digital entre uma sequência de entrada e uma constante C de três *bits*. Na descrição deste circuito existem dois tipos de blocos funcionais: somadores e comparadores (representados por `+` e `=`, respectivamente). No primeiro circuito, o ciclo de relógio está limitado pelo tempo necessário para realizar uma comparação e duas somas. No segundo circuito, e após a reoptimização temporal que levou à alteração da posição de dois registos ao longo do fluxo de dados, o ciclo de relógio é apenas limitado pelo tempo de uma comparação e uma soma, pelo que é possível aumentar a velocidade de funcionamento do circuito[6].

A síntese das máquinas de estados que representam os controladores é a segunda tarefa da síntese RTL. Esta tarefa, que parte de uma descrição da máquina em termos de um conjunto de estados e das transições entre eles, tem como objectivo seleccionar uma arquitectura apropriada e efectuar a codificação dos estados, das entradas e das saídas, com o intuito de reduzir a área e/ou os tempos de atraso da implementação final do circuito [Monteiro 92].

A geração da codificação óptima de uma máquina de estados é um problema cuja complexidade cresce exponencialmente com o número de estados. Torna-se assim impossível obter uma solução óptima através da procura exaustiva de todas as codificações existentes quando o número de estados é elevado. Por esta razão, a codificação das máquinas de estado é feita recorrendo a heurísticas. Os sistemas KISS [Micheli 85] e MUSTANG [Devadas 88] utilizam este método para limitar o conjunto de codificações onde deve ser procurada a solução "óptima".

A maioria dos sistemas de síntese comerciais suportam apenas a síntese a nível RTL e lógico. Isto, apesar de alguns se afirmarem como sistemas de síntese de

[5]Na literatura inglesa, *retiming*.

[6]Este aumento de velocidade resultou também num aumento da área do circuito, pois apesar do número de registos ser o mesmo, o registo que guarda o resultado da soma é de dois *bits*.

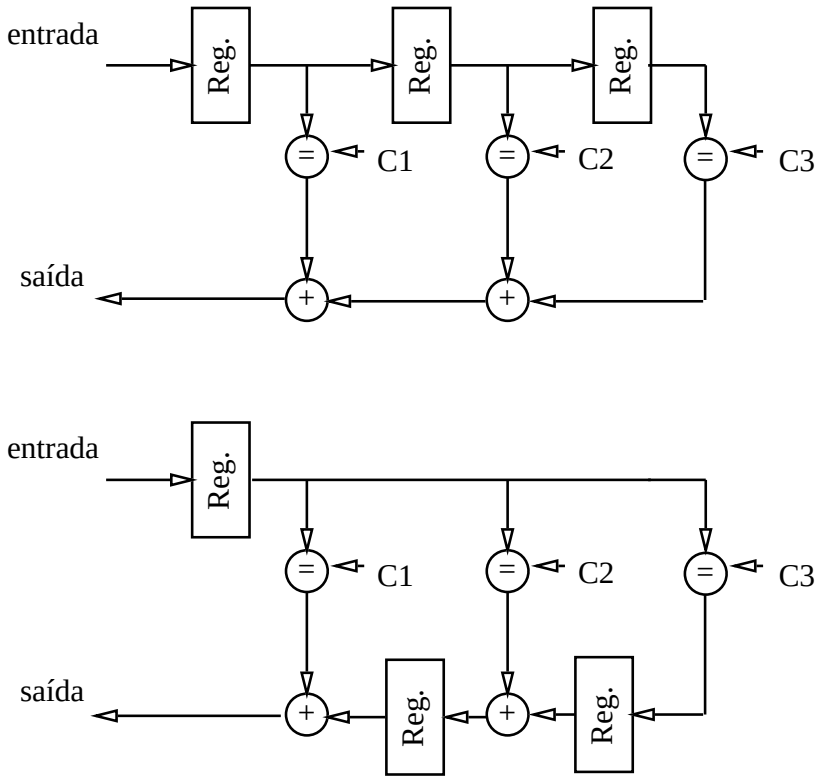


Figura 2.3: Exemplo da operação de reoptimização temporal sobre uma descrição RTL.

alto-nível, por aceitarem uma descrição do circuito através de um algoritmo e efectuarem algumas das tarefas de síntese de alto-nível. Porém, as restrições impostas na forma de descrever o circuito limitam na maioria dos casos a sua representação ao nível de transferência de registos.

2.2.3 Síntese a nível lógico

A síntese a nível lógico 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.

Como o mapeamento de uma função lógica pode ser realizado de uma forma mais ou menos directa por um conjunto de portas lógicas (por exemplo, qualquer função lógica pode ser mapeada em portas *nand*) o passo de maior complexidade da síntese lógica é a optimização da lógica obtida em termos de área e/ou velocidade.

Esta optimização está geralmente dividida em duas fases: optimização da estrutura global do circuito lógico de uma forma independente da tecnologia e mapeamento dessa estrutura num conjunto de portas lógicas de uma dada biblioteca, através de optimizações locais dependentes da tecnologia. Estas duas fases são realizadas pelas seguintes tarefas:

Optimização lógica a dois-níveis - A optimização lógica a dois-níveis foi desenvolvida para a implementação de circuitos em arquitecturas PLA. Neste tipo de arquitectura as funções lógicas que descrevem o circuito são representadas a dois-níveis, ou seja cada saída é obtida como uma soma de produtos da entradas. Assim, esta optimização tem como objectivo minimizar o número de produtos e somas necessários para representar as funções de saída do circuito.

Esta optimização é feita tendo em conta dois conjuntos que caracterizam cada função de saída: um constituído pelas combinações de entrada (minitermos) para os quais a saída da função vale '1' ($conjunto\_UM$) e outro constituído pelas combinações de entrada para os quais a saída não está especificada ($conjunto\_NE$), podendo por isso tomar qualquer valor ('0' ou '1'). Os algoritmos de optimização lógica a dois-níveis utilizam os seguintes passos para obter a representação óptima da função a dois-níveis:

1. Determinar o menor número de literais[7] necessários para representar cada produto que faz parte da descrição da função, recorrendo aos minitermos do $conjunto\_UM$ e $conjunto\_NE$.
2. Encontrar o menor número de somas necessárias para representar a função, eliminando os produtos de literais que são redundantes.

A solução exacta para este problema de optimização não pode ser obtida pela procura exaustiva de todas as soluções devido à complexidade que esta apresenta (tempo de processamento e memória necessária), com o aumento do número de entradas de cada função lógica. Assim, têm que ser usadas heurísticas de forma a limitar a procura da solução óptima. O estudo e aperfeiçoamento destas heurísticas está suficientemente desenvolvido para que os resultados produzidos pelos algoritmos de optimização lógica a dois-níveis possam ser considerados de "óptimos".

[7]Um literal representa uma variável ou a sua negação. Por exemplo se a for uma variável de entrada então a e a' são os literais dessa variável.

Um dos algoritmos heurísticos mais utilizado para a optimização lógica a dois-níveis é aquele que é usado no programa ESPRESSO [Rudell 87]. O ciclo principal deste algoritmo para além de realizar os passos acima referidos utilizando heurísticas, incluiu um terceiro passo que tenta evitar que o processo de optimização convirja para um mínimo local. Neste passo, o número de literais de cada produto que descreve uma função de saída é aumentado, de forma a que no próximo ciclo se obtenha uma solução mais próxima de um mínimo global. A interrupção deste ciclo é feita quando o valor da função de custo, dependente do número de literais e do número de produtos, não se alterar entre duas iterações.

Optimização lógica multi-nível - A representação multi-nível de um conjunto de funções lógicas é feita recorrendo a vários níveis de funções intermédias que podem ser partilhadas na obtenção das saídas. Este tipo de representação apresenta algumas vantagens em relação a uma representação a dois-níveis pois permite obter circuitos mais pequenos e em geral mais rápidos.

A optimização lógica multi-nível tem como objectivo obter a melhor estrutura multi-nível de uma função lógica, minimizando uma dada função de custo, envolvendo a área do circuito e tempos de atraso. Estes parâmetros têm que ser ambos estimados de uma forma simplificada, devido a esta optimização ser feita ainda na fase independente da tecnologia.

Não existindo actualmente nenhuma forma de obter directamente a "melhor" representação multi-nível de um circuito, a optimização lógica multi-nível é feita à custa da aplicação sucessiva de um conjunto de operações, que manipulam a estrutura do circuito de uma forma heurística. As operações mais comuns são [Brayton 90]:

- Decomposição - exprime uma única função booleana como um conjunto de várias funções intermédias. Desta forma é possível dividir funções (intermédias ou não) noutras com menos literais.
- Extracção - operação idêntica à decomposição, mas é realizada sobre várias funções, sendo identificadas diferentes subexpressões comuns, que passam a ser representadas por funções intermédias partilhadas.
- Factorização - operação que obtém uma forma factorizada de uma função representada como uma soma de produtos.
- Substituição - exprime uma função em termos de outra que pode ser intermédia ou não.

- Eliminação - realiza a operação inversa da substituição: elimina certas funções intermédias que não têm vantagem em ser utilizadas.

O sistema de optimização lógica multi-nível MIS [Brayton 87], que inclui também a tarefa de mapeamento tecnológico, constitui um exemplo da conjugação destas operações para obter o circuito com menor área que satisfaz um conjunto de restrições temporais. A qualidade dos resultados obtidos depende não só da ordem de execução das várias operações (ordem esta que pode ser controlada pelo utilizador), como também dos métodos usados em cada operação. Por exemplo, a operação de substituição pode ser realizada utilizando um método algébrico ou booleano. O método algébrico, que considera cada função lógica como um "polinómio", é bastante mais rápido mas, ao contrario do método booleano, não explora todas as possibilidades da estrutura do circuito.

Apesar de os resultados obtidos pela optimização lógica multi-nível serem na maioria dos casos suficientemente bons, esta é ainda uma área de investigação científica actual (por exemplo, para a operação de extracção ainda não existem algoritmos eficientes que utilizem métodos booleanos).

Mapeamento Tecnológico - O mapeamento tecnológico permite obter uma representação do circuito através da interligação de um conjunto de portas lógicas disponíveis numa dada tecnologia. Partindo das funções booleanas obtidas nos passos anteriores, esta tarefa, que considera a estrutura global do circuito como óptima (resultante das tarefas de optimização anteriores), realiza optimizações locais de forma a minimizar a área e garantir as restrições temporais e de *fan-out* que possam existir.

O mapeamento tecnológico pode ser realizado usando duas técnicas diferentes: baseada em regras ou por cobertura de grafos.

As técnicas baseadas em regras são idênticas às habitualmente usadas em sistemas periciais. Isto é, existe um conjunto de regras, representando os conhecimentos adquiridos pelo projectista, que são usadas para realizar as optimizações locais. Assim, a partir da representação das equações booleanas em portas genéricas (por exemplo, portas *nands*), são realizadas sucessivas transformação no circuito (resultantes das aplicação das regras) de forma a melhorar a qualidade final deste. O SOCRATES [Bartlett 86], é um sistema de síntese a nível lógico que realiza a fase de optimização dependente da tecnologia utilizando uma técnica baseada em regras. Na figura 2.4 apresentam-se algumas regras usadas por este sistema: as duas primeiras, têm como objectivo reduzir a área do circuito substituindo um conjunto

de portas lógicas por outro mais pequeno, mas realizando a mesma função lógica; a terceira, realiza uma optimização temporal pela deslocação dos sinais mais atrasados para junto das saídas (aumentando a velocidade máxima de funcionamento do circuito).

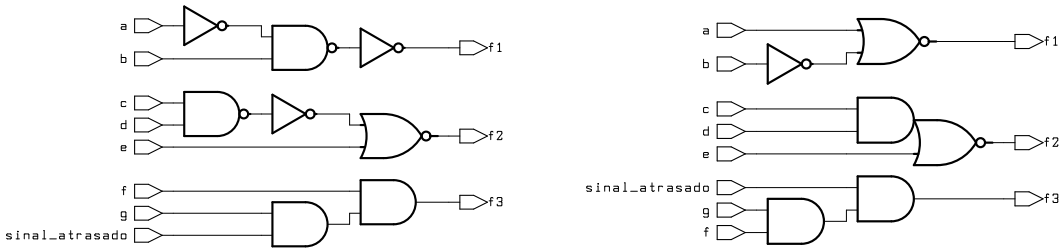


Figura 2.4: Exemplo de algumas regras do sistema SOCRATES.

As técnicas baseadas na cobertura de grafos são análogas às usadas nas fases de geração de código de um compilador, quando se pretende mapear uma expressão num conjunto de instruções máquina. Esta técnica requer que as funções booleanas do circuito e das células da biblioteca, sejam representadas utilizando apenas um conjunto de operações lógicas básicas (por exemplo, *nands* de duas entradas e inversores). Estas representações são transformadas em grafos orientados sem ciclos em que cada nó realiza uma única operação básica. O mapeamento tecnológico é feito pela escolha optimizada (em termos de área e/ou velocidade) de um conjunto de células da biblioteca cujos vários grafos (grafos padrões) cobrem na totalidade o grafo do circuito (grafo base). Ou seja, cada nó do grafo base tem um nó correspondente no conjunto dos grafos padrões escolhido e cada entrada de um grafo padrão é saída de outro grafo padrão. Esta é a técnica de mapeamento utilizada no sistema MIS na fase de optimização dependente tecnologia.

2.3 Conclusões

Neste capítulo foi apresentado o processo de síntese de um circuito digital como a forma de obter a descrição do circuito ao nível lógico (mapeado numa dada tecnologia e satisfazendo um conjunto de restrições) a partir de uma descrição do seu comportamento feita através de um algoritmo. Este processo, que idealmente deveria ser realizado numa única transformação, encontra-se dividido em várias tarefas de síntese que envolvem a representação do circuito em vários níveis de

abstracção intermédios. A optimização dos algoritmos que realizam essas tarefas e a forma de integrar os diferentes níveis de síntese é actualmente uma área de intensa investigação.

A forma mais comum de descrever o comportamento de um circuito digital para fornecer a um sistema de síntese, consiste na utilização de linguagens de descrição de *hardware*. O uso destas linguagens é apresentado no próximo capítulo.

Capítulo 3

A Linguagem VHDL

Neste capítulo referem-se algumas vantagens da utilização de uma linguagem de descrição de *hardware* num ambiente de projecto de circuitos integrados. Estas linguagens, que apresentam um conjunto de características e construções próprias tiveram, na maior parte dos casos, origem nas linguagens de programação de alto-nível. São descritas algumas linguagens mais importantes, sendo dado especial atenção à linguagem norma do IEEE, o VHDL, através de uma descrição sumária da sua evolução e das suas características principais.

3.1 As Linguagens de Descrição de *hardware*

As linguagens descrição de *hardware* começaram por representar circuitos ao nível lógico, pois era a esse nível que era feita a especificação e implementação dos circuitos. Com a necessidade de gerir o crescente aumento de complexidade dos circuitos electrónicos, tornou-se necessária a representação do comportamento dos circuitos em níveis de abstracção mais elevados. As linguagens de descrição de *hardware* actuais permitem descrever e documentar circuitos em que a funcionalidade pretendida pode ser simulada sem a necessidade de se descer ao nível lógico. Devido à representação de um circuito ser feita de uma forma independente da tecnologia a escolha do fabricante pode ser adiada para uma fase posterior do projecto. Assim, a reutilização de projectos anteriores torna-se possível de uma forma mais simples com o uso de bibliotecas que podem ser partilhadas pelos diversos projectistas.

As linguagens de descrição de *hardware* têm ainda a vantagem de representar

um meio importante para a troca de informação de *hardware* entre organizações, vantagem essa que pode ter um significado mais real se a linguagem escolhida estiver amplamente divulgada na comunidade de micro-electrónica. Como a descrição é feita num nível de abstracção elevado, os vendedores podem fornecer informação sobre o comportamento dos seus dispositivos sem necessidade de incluir qualquer tipo de informação confidencial sobre a sua construção.

Originalmente muitas das linguagens de descrição de *hardware* evoluíram das linguagens de programação herdando muitas das características destas. No entanto, por serem destinadas à descrição de *hardware* possuem necessariamente algumas construções e características específicas adicionais. Estas dependem não só do domínio de aplicação da linguagem (processamento digital de sinal, controlo em tempo real, sistemas de telecomunicações ou sistemas de consumo), como também do seu ambiente de utilização (simulação, síntese e/ou verificação formal). As construções mais comuns, próprias destas linguagens, são [Waxman 86, Shiva 79, Augustin 89]:

Declaração de interface - Uma vez que o sistema representado pela descrição vai ter que comunicar com o ambiente que o envolve, há que de alguma forma especificar quais os acessos e as respectivas propriedades, que o sistema possui.

Declarações estruturais - As declarações estruturais permitem a utilização de certas estruturas de *hardware* (por exemplo, contadores, registos, etc) como se fossem funções. O uso destas construções permite a representação de circuitos no domínio estrutural e em diferentes níveis de abstracção, forçando quando necessário uma dada estrutura ao circuito.

Objectos e tipos de dados específicos - De forma a poderem modelar e descrever correctamente o comportamento do *hardware* (sinais, tempos de atraso, etc) é necessário a extensão do conceito do tipo de armazenamento de dados (objectos) bem como os tipos de dados disponíveis.

Operadores e funções específicas - Como forma de complementar os operadores numéricos e booleanos surgem operadores que permitem realizar operações a nível de *bit* (operadores de deslocamento, rotação, *and*, *or*, etc). As funções adicionais oferecidas reflectem em geral o tipo de utilização a que se destina a linguagem.

Referências temporais - Ao contrário das linguagens usuais de programação, onde a noção de tempo não existe, as linguagens de descrição de *hardware*

têm uma semântica que permite explicitamente a representação do comportamento temporal do circuito, não sendo necessário criar formas artificiais para manipular as questões temporais do circuito.

Assincronismo e concorrência - A presença destas duas características no *hardware* (por exemplo, com *sets* e *resets* assíncronos, *pipelines*, etc) implica a existência de mecanismos nas linguagens de descrição de *hardware* que permitam a sua modelação. Consequentemente devem também existir primitivas para a sincronização e comunicação entre processos concorrentes.

As linguagens de descrição de *hardware* mais recentes têm uma estrutura mais adaptada às novas metodologias usadas na produção de sistemas electrónicos. Isto é, para além de permitirem diferentes níveis e estilos de descrição, têm uma semântica que torna possível a sua simulação, síntese e verificação formal. As linguagens de descrição de *hardware* mais conhecidas são [Bergé 92, Maginot 92]:

HardwareC - Linguagem desenvolvida na universidade de Standford com o objectivo de descrever o comportamento de circuitos digitais para o sistema OLYMPUS. O HardwareC tem uma sintaxe semelhante à da linguagem de programação C, mas com uma semântica diferente de forma a permitir não

```
process mdc(x_entrada, y_entrada, inicio, resultado)
    in port     x_entrada[8], y_entrada[8], inicio;
    out port    resultado[8];
{
    boolean x[8], y[8];
    tag a, b;
    constraint mintime from a to b = cycles;

    write resultado = 0;
    a: while(inicio);
    <b: x = read(x_entrada); y = read(y_entrada);>

    if((x > 0) && (y > 0)){
        repeat{
            while(x >= y)
                x = x - y;
            <y=x; x=y;>
        }until(y == 0);
    }
    else
        x = 0;

    write resultado = x;
}
```

Figura 3.1: Exemplo da descrição de um circuito em HardwareC.

só a descrição de circuitos digitais mas também a especificação de restrições. Apesar de ser uma linguagem de domínio público e de fácil compreensão, devido à sua semelhança com o C, o HardwareC é apenas usado em meios académicos, estando em geral associada ao sistema OLYMPUS que permite a simulação e síntese de circuitos descritos nesta linguagem.

Na figura 3.1 apresenta-se a descrição de um circuito em HardwareC que calcula o máximo divisor comum de dois números de oito *bits* (positivos e diferentes de zero), utilizando o algoritmo de Euclid.

M - A linguagem M foi desenvolvida para um simulador específico, o Lsim, da Mentor Graphics (originalmente concebido pela Silicon Compiler System). Esta linguagem, proprietária da Mentor Graphics, é baseada também na linguagem de programação C mas tem um conjunto de características que estão directamente ligadas ao ambiente de simulação para o qual foi desenvolvida.

Na figura 3.2 ilustra-se a descrição na linguagem M de um registo de deslocamento de tamanho (número de andares) e largura (número de *bits* de cada andar) parametrizável.

```
MODULE reg_deslocamnet(n_bits, comp)
    int n_bits;
    int comp;
{
    IN LOGIC entrada[n_bits-1];
    IN LOGIC clock;
    OUT LOGIC saida[n_bits-1];

    MEMORY LOGIC mem[comp-1][n_bits-1];

    INITIALIZE {
        mem = UNKNOWN;
    }
    SIMULATE {
        int i;

        if(RISE(clock)){
            saida = mem[comp-2];
            for(i=comp-2; i>0; i--)
                mem[i] = mem[i-1];
            mem[0] = entrada;
         }
    }
}
```

Figura 3.2: Exemplo da descrição de um circuito em M.

ELLA - Linguagem desenvolvida num projecto iniciado em 1978 pelo Ministério da Defesa da Inglaterra (MoD[1]) que permite a descrição de circuitos electrónicos a vários níveis de abstracção. A pouca divulgação e a falta de ferramentas eficientes levou a que o uso da linguagem ELLA esteja actualmente restringido ao *Royal Signals and Radar Establishment* onde foi desenvolvida [Morison 84].

Na figura 3.3 apresenta-se a descrição de um contador ascendente/descendente de quatro *bits* com carregamento paralelo utilizando a linguagem ELLA.

```
TYPE
   bool = NEW (0 | 1),
   subedesce = NEW (sobe | desce),
   funcao = NEW (contar | carregar).

MAC CONTADORN{INT n}=(bool: ck, [n]bool : contagemcarrega,
   funcao: f, subedesce: dir) -> [n]bool:
BEGIN
   MAKE REG{[n]bool}:  contagem.

   JOIN (ck, CASE f OF
      contar:
            CASE dir OF
               sobe : INC{n}(contagem),
               desce: DEC{n}(contagem)
            ESAC,
      carregar:
         contagemcarrega
   ESAC) -> contagem.

   OUTPUT contagem
END.

FN CONTA16 = (bool: ck, [4]bool : contagemcarrega, funcao: f,
      subedesce: dir) -> [4]bool:
   CONTADORN{4}(ck, contagemcarrega, f, dir).
```

Figura 3.3: Exemplo da descrição de um circuito em ELLA.

Verilog - Linguagem proprietária da Cadence e bastante divulgada nos Estados Unidos da América, não só a nível do número de projectistas que a usam, mas também na quantidade de bibliotecas e de modelos existentes. Desenvolvida originalmente pela Gateway Design Automation que posteriormente se juntou à Cadence sempre foi conhecida e usada como a linguagem do simulador Verilog-XL.

Actualmente, e face à normalização do VHDL, sofre um processo de abertura ao domínio público através da organização Open Verilog International (OVI)

[1]Do Inglês *Ministry of Defense*

que em Outubro de 1991 publicou a primeira versão do manual de referência da linguagem [Verilog 91]. A descrição da figura 3.4 representa um circuito que determina o número de zeros consecutivos em palavras de oito *bits* se estas tiverem uma única sequência de zeros.

```
module conta_zeros(entrada, saida, erro);
   input  [7:0] entrada;
   output [3:0] saida;
   output erro;

   function validacao;
       input [7:0] x;
       reg seqInicio, seqFim;
       integer i;

   begin : _bloco_validacao
       validacao = 1; seqInicio = 0; seqFim = 0;
       for ( i=0; i <= 7; i=i+1 )
       if ( seqFim && (x[i] == 0) ) begin
           validacao = 0;
           disable _bloco_validacao;
       end
       else if ( seqInicio && (x[i] == 1) )
           seqFim = 1;
       else if ( x[i] == 0 )
           seqInicio = 1;
   end
   endfunction

   function [3:0] num_zeros;
       input [7:0] x;
       reg   [3:0] contador;
       integer i;

   begin
       contador = 0;
       for ( i=0; i <= 7; i=i+1 )
           if ( x[i] == 0 )
               contador = contador + 1;
       num_zeros = contador;
   end
   endfunction

   wire ent_valida = validacao(entrada);
   assign erro = ! ent_valida;
   assign saida = ent_valida ? num_zeros(entrada) : 0;
endmodule
```

Figura 3.4: Exemplo de uma descrição de um circuito em Verilog.

UDL/I -*Unified Design Language/for Integrated circuits* foi desenvolvida pela Associação para o Desenvolvimento da Indústria Electrónica Japonesa com o objectivo de ser normalizada como a linguagem de descrição *hardware* [Yasuura 89]. A sua construção foi pensada de forma a ser independente de

qualquer ferramenta ou ambiente de desenvolvimento. Porém, a sua sintaxe e semântica estão adaptadas quer à simulação quer à síntese e verificação formal.

Devido a ser uma das mais recentes linguagens de descrição de *hardware*, o seu manual de referência só recentemente foi divulgado [UDL 92], não existindo ainda qualquer ferramenta disponível que incentive o seu uso. Na figura 3.5 apresenta-se a descrição, em UDL/I, de um registo de oito *bits* com entradas para carregamento síncrono e assíncrono.

```
IDENT: exemplo;

  NAME: registo;
    INPUTS: entrada<0:7>, entrada_asinc<0:7>, carreg_asinc;
    OUTPUTS: saida<0:7>;
    CLOCK: clk;

    BEHAVIOR_SECTION;
      REGISTER: reg<0:7>;
      reg := BEGIN
          AT RISE(clk) DO entrada END_DO;
          AT HIGH(carreg_asinc) DO entrada_asinc END_DO;
        END;
      saida := reg;
    END_SECTION;

  END;

CEND;
```

Figura 3.5: Exemplo de uma descrição de um circuito em UDL/I.

VHDL - *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language* foi também desenvolvida de uma forma independente de qualquer ferramenta, tendo no entanto uma semântica virada para a simulação. Suporta diferentes estilos de descrição (estrutural, fluxo de dados e/ou comportamental) o que permite a sua utilização para a descrição de *hardware* em diferentes níveis de abstracção, desde o nível lógico até ao nível de sistema.

A adopção desta linguagem por parte de várias instituições como norma para descrição de *hardware* fez com que passasse a ser suportada praticamente por todos os sistemas actuais de CAD. Simultaneamente apareceram também modelos e bibliotecas de células escritas em VHDL. Na figura 3.6 é descrito em VHDL o comportamento de uma unidade lógica e aritmética simplificada com dois registos.

Ao contrário de outras linguagens, que estão intimamente ligadas às compa-

```
package dec_tipos is
    type operacoes is (CARREGA_REG, REG_PARA_ACUM, COMPARA, SOMA);
end dec_tipos;

use work.dec_tipos.all;
entity ula is
    port(entrada: in integer range 0 to 255;
         oper: in operacoes;
         clock: in bit;
         menor, igual, maior: out boolean;
         saida: out integer range 0 to 510
         );
end ula;

architecture comportamento of ula is
begin
    process
        variable reg, acum: integer range 0 to 510;
    begin
        wait until clock'event and clock = '1';
        case oper is
            when CARREGA_REG  =>
                reg := entrada;
            when REG_PARA_ACUM =>
                acum := reg;
            when COMPARA =>
                menor <= (reg < acum);
                igual <= (reg = acum);
                maior <= (reg > acum);
            when SOMA =>
                acum := reg + acum;
                saida <= acum;
        end case;
    end process ;
end comportamento;
```

Figura 3.6: Exemplo da descrição de um circuito em VHDL

nhias 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. Por esta razão, a descrição de circuitos electrónicos digitais neste trabalho será feita usando a linguagem VHDL.

3.2 A Evolução do VHDL

3.2.1 O Passado

Os requisitos para o desenvolvimento de uma linguagem como o VHDL surgiram em 1981 por parte do Departamento de Defesa dos Estados Unidos da América (DoD[2]), através do seu programa intitulado *Very High Speed Integrated Circuits* (VHSIC). Com a participação de membros do governo, da indústria e dos meios universitários, pretendia-se reduzir a quantidade de informação que tinha de ser analisada e interpretada pelos projectistas antes de implementar ou testar um dado sistema electrónico.

O VHDL deveria endereçar também os problemas resultantes da crescente complexidade dos sistemas de forma a permitir representações tecnologicamente independentes e permitir assim a reutilização parcial ou total de projectos anteriores.

Com o desenvolvimento da versão 7.2 por parte de uma equipa constituída pela Intermetrics, IBM e Texas Instruments, e após a inclusão de algumas recomendações do meio industrial e académico, o IEEE em 1987 certificou o VHDL como a Norma 1076. A sintaxe e semântica da linguagem são mantidas pelo Grupo de Normalização e Análise do VHDL (VASG[3]) e encontram-se documentadas no manual de referência da linguagem [LRM 88].

3.2.2 O Presente

De forma a "manter viva" uma norma, e torná-la de facto efectiva perante as mudanças tecnológicas, o IEEE obriga a que ela seja rectificada todos os cincos anos. Assim em 1992 deveria ser apresentada uma nova proposta para a normalização da linguagem VHDL. Este processo de normalização iniciou-se em 1990 e envolveu as seguintes etapas [Shahdad 91, Shahdad 92]:

1. Reunião e definição das alterações proposta à linguagem. As origens destes pedidos foram diversas (utilizadores, fabricantes de ferramentas de CAD, produtores de circuitos integrados) e cobriram áreas bastante variadas como a modelação, simulação, síntese e até extensões ao domínio analógico.

[2]Do inglês *Department of Defense*.
[3]Do inglês *VHDL Analysis and Standardization Group*

2. Estudo dos objectivos a ter em conta na nova norma. Todo o conjunto de pedidos foi dividido em quatro categorias: uma que continha aqueles pedidos que teriam que ser satisfeitos na nova norma; outra para aqueles que deveriam ser levados em conta; uma terceira para pedidos que só poderiam ser incluídos no processo de normalização de 1997; e finalmente aqueles que não seriam satisfeitos [Rouillard 92].

3. Desenvolvimento da linguagem. Para o desenvolvimento do VHDL-92 foram tidos em conta certos critérios de entre os quais se destacam a compatibilidade com o VHDL-87, a manutenção da coerência da linguagem, a portabilidade dos modelos existentes e a não inclusão de módulos[4] específicos a certas aplicações ou áreas como parte da norma.

4. Escrita da documentação. O novo manual de referência para além de definir a sintaxe e semântica da linguagem vai incluir também um sumário das alterações efectuadas e um apêndice com as construções que podem ser consideradas de não portáveis.

5. Validação e votação da nova linguagem. O processo de validação consiste na realização de descrições e protótipos de ferramentas que evidenciem as novas características da linguagem [Zamfirescu 91, Guyler 92]. A votação é aberta a todos os utilizadores de VHDL e permite certificar a nova linguagem perante o IEEE.

A segunda versão do manual de referência foi votada e aprovada, estando actualmente a decorrer o processo de submissão da norma ao IEEE, terminando assim o processo de normalização do VHDL-92.

No VHDL existe apenas um único tipo pré-definido que permite a representação dos valores digitais de um circuito, o `bit`. Devido à insuficiência deste tipo para modelar grande parte dos circuitos electrónicos foi necessário alargar o conceito de valor lógico para um sistema de lógica multi-valor. O novo tipo deveria englobar pelo menos também as representações de alta impedância, valor lógico desconhecido e não inicializado, entre outros. Para evitar o aparecimento de várias lógicas multi-valor, uma por cada vendedor de CAD, foi necessário proceder à normalização de um módulo de forma a manter a portabilidade dos modelos. A norma 1164 do IEEE, recentemente aprovada, define precisamente o módulo `STD_LOGIC_1164`

[4]Os módulos permitem descrever um conjunto de procedimentos, funções e tipos que podem ser partilhados por várias descrições de VHDL. Este conceito é explicado com mais detalhe na secção 3.3.1

criando um tipo para representar uma lógica de nove valores e ainda subtipos para as lógica de três, quatro e cinco valores para além das funções necessárias para a manipulação de uma álgebra multi-valor.

3.2.3 O Futuro

O estudo e desenvolvimento de um módulo específico para a síntese vem sendo feito pelo grupo de síntese (SSIG[5]) à cerca de dois anos. Pretende-se com este módulo permitir a portabilidade dos modelos sintetizáveis entre várias ferramentas de síntese. Assim, devem ser definidos: quais os valores lógicos permitidos para descrever um circuito e a forma como estes devem ser sintetizados; o conjunto de representações numéricas possíveis (feitas à custa dos valores lógicos anteriores) com os respectivos operadores e funções de conversão de tipos; as formas de caracterizar e impor restrições num dado circuito através da sua descrição VHDL. Apesar de alguns deste pontos já estarem completamente definidos (por exemplo, uso da lógica de nove valores definida pela norma 1164) existem outros que ainda estão em estudo (por exemplo, como deve ser interpretado pela ferramenta de síntese o valor lógico desconhecido)[Harper 92].

A extensão da linguagem para o domínio analógico não foi introduzida no actual processo de normalização. No entanto, pretende-se continuar a definir as alterações e extensões necessárias para que, quando da nova normalização em 1997, a descrição de circuitos analógicos possa ser introduzida de uma forma coerente [Cottrell 92].

A definição de um formato intermédio de representação do VHDL que tenha um interface comum com as ferramentas de CAD de vários domínios (simulação, síntese e verificação formal) está também a ser objecto de estudo para normalização [Brown 92].

Nota: Todas as menções desta tese no que respeita à linguagem VHDL referem-se à norma IEEE 1076-1987 que se encontra descrita no seu manual de referência [LRM 88].

[5]Do inglês *Synthesis Special Interest Group*

3.3 Principais Características da Linguagem VHDL

3.3.1 Unidades de Projecto em VHDL

De acordo com a definição da linguagem, as unidades de projecto são o conjunto mínimo de construções que podem ser analisadas de uma forma separada. Após esta análise cada unidade é inserida numa biblioteca de projecto, conceito que é suposto ser suportado pelo ambiente de desenvolvimento, e que por defeito se chama `WORK`. As unidades de projecto existentes em VHDL são apresentados nas subsecções seguintes.

Entidade

A entidade é uma abstracção de *hardware* em VHDL que pode representar um sistema completo, uma placa, um integrado, uma célula ou mesmo uma simples porta lógica. Na realidade, em cada entidade é definido o interface do bloco que representa com o ambiente onde esse bloco vai ser integrado. Na declaração de uma entidade são dados a conhecer os acessos ao interior de cada bloco sem no entanto ser descrita a sua funcionalidade. No exemplo da figura 3.7 descrevem-se os acessos de um somador de um *bit*.

```
entity somador is
    port(x, y, transporte_ant: in bit;    -- acessos de entrada
         soma, prox_transporte: out bit); -- acessos de saida
end somador;
```

Figura 3.7: Exemplo de uma entidade: declaração dos acessos de um somador.

Arquitectura

A arquitectura define a funcionalidade de uma dada entidade. Utilizando o conjunto de instruções disponíveis em VHDL (referidas na secção 3.3.4) descreve-se na arquitectura como são obtidas as várias saídas do circuito em função das suas entradas.

Cada declaração de entidade pode ter associada um número indeterminado

de arquitecturas. A "mesma" funcionalidade em cada uma delas pode ser obtida recorrendo a diferentes estilos de descrição:

Descrição Estrutural - o circuito é apresentado como um conjunto de componentes interligados. Estes podem ser células básicas ou outros componentes também descritos em VHDL e existentes na biblioteca. No exemplo da figura 3.8 apresenta-se uma arquitectura possível para o somador onde a estrutura do circuito está completamente determinada.

```
architecture estrutural of somador is

    component meio_somador
        port(a, b: in bit;
             soma, transporte: out bit);
    end component;

    component porta_ou
        port(a, b: in bit;
             z: out bit);
    end component;

    signal r, s, t: bit;

begin
    U1: meio_somador port map(x, y, r, s);
    U2: meio_somador port map(s, transporte_ant, t, soma);
    U3: porta_ou port map(r, t, prox_transporte);
end estrutural;
```

Figura 3.8: Descrição estrutural do somador.

Descrição de Fluxo de Dados - representa o comportamento do circuito de uma forma mais abstracta mas implicando a existência de alguma estrutura. É idêntica a uma descrição a nível de transferência de registos. Na figura 3.9 encontra-se a descrição do somador utilizando este estilo.

```
architecture fluxo_dados of somador is
    signal s: bit;
begin
    s <= x xor y after 10 ns;
    soma <= s xor transporte_ant after 10 ns;
    prox_transporte <= (x and y) or (s and transporte_ant);
end fluxo_dados;
```

Figura 3.9: Descrição de fluxo de dados do somador.

Descrição Comportamental - descreve o comportamento do circuito sem qualquer informação quanto à sua estrutura. Este tipo de descrição utiliza uma sequência de instruções idênticas às da linguagens de programação algorítmicas mas com uma sintaxe e semântica próprias. Uma descrição comportamental do somador é apresentada na figura 3.10.

```
architecture comportamental of somador is
begin
    process
        variable uns: integer;
        constant vector_soma: bit_vector(0 to 3) := "0101";
        constant vector_transporte: bit_vector(0 to 3) := "0011";
    begin
        uns := 0;
        if x = '1' then uns := uns + 1; end if;
        if y = '1' then uns := uns + 1; end if;
        if transporte_ant = '1' then uns := uns + 1; end if;

        soma <= vector_soma(uns) after 20 ns;
        prox_transporte <= vector_transporte(uns) after 30 ns;

        wait on x, y, transporte_ant;
    end process;
end comportamental;
```

Figura 3.10: Descrição comportamental do somador.

Uma descrição mista usando uma combinação dos três estilos acima referidos é também possível.

Configuração

Podendo existir várias arquitecturas para uma mesma entidade é necessário definir para cada instanciação de uma entidade qual a arquitectura envolvida. Isto pode ser feito através da unidade de configuração que define quais os pares entidade-arquitectura que estão envolvidos na descrição de um sistema. Este pares são também muitas vezes designados por entidades de projecto.

No exemplo da figura 3.11 apresenta-se uma configuração possível para o circuito somador. Esta escolhe, entre as várias descrições que vimos atrás, a arquitectura `estrutural` indicando também quais as entidades de projecto que devem ser usadas para os componentes instanciados nessa arquitectura.

```
configuration possivel of somador is
    for estrutural
        for U1, U2: meio_somador use
            entity m_soma(comp);
        end for;
        for U3: porta_ou use
            entity OR2(std_lib);
        end for;
    end for;
end possivel;
```

Figura 3.11: Configuração do somador.

Módulo

O conceito de módulo teve origem na linguagem ADA e representa uma unidade de *software*. Este pode ser constituído por duas partes: a declaração, que indica qual a visão externa que se tem do módulo; e o corpo, onde se encontra a implementação dos procedimentos e funções oferecidas. Qualquer uma destas partes constitui uma unidade de VHDL que pode ser analisada separadamente. O uso de módulos permite uma melhor organização dos circuitos a descrever através da partilha de partes de código que sejam frequentemente usadas, tais como: declarações de tipos, procedimentos de conversão de tipos, etc.

Existem já dois módulos pré-definidos na linguagem: o `STANDARD`, que define alguns tipos e subtipos usuais na descrição de circuitos (`TIME, BIT, BIT_VECTOR`, etc); e o `TEXTIO` que define os tipos e procedimentos necessários para a manipulação de ficheiros de texto. Outros podem ser definidos para "aumentar a funcionalidade" da linguagem, tal como aconteceu com o módulo `STD_LOGIC_1164` (referido na secção 3.2.3).

3.3.2 Objectos, Tipos de Dados e Atributos

Objectos em VHDL são elementos que contêm valores de um dado tipo. Existem três classes de objectos: constantes, variáveis e sinais. As constantes têm um valor fixo que não pode ser alterado, enquanto que as variáveis têm um valor que pode ser alterado ao longo da execução do programa. Os sinais podem ser vistos como a representação abstracta de um fio: têm um conjunto de valores passados, um valor presente e um conjunto de valores projectados para o futuro. Só estes últimos valores podem ser alterados pelas instruções de atribuição.

O tipo de um objecto determina quais os valores que esse objecto pode conter.

Em VHDL é permitido ao projectista declarar qualquer número de tipo de dados para caracterizar os seus objectos.

Os 4 tipos básicos escalares são: inteiros, vírgula flutuante, físicos e enumerados. Tipos compostos como matrizes e agregados podem ser definidos à custa dos tipos básicos. Subtipos podem ser também definidos através da imposição de restrições a um outro tipo. Existem ainda os apontadores, idênticos àqueles que são usados nas linguagens vulgares de programação, e os ficheiros que permitem uma forma de comunicação do circuito com o meio exterior.

Os atributos são características (valores, funções, etc) que podem estar associadas a determinados elementos que constituem o VHDL. Para além dos atributos pré-definidos na linguagem o utilizador pode definir os seus próprios atributos, que podem ser colocados em qualquer um dos elementos apresentados até agora: unidades de projecto, objectos, tipos, subtipos, procedimentos, funções ou componentes.

3.3.3 O Modelo Temporal

O modelo temporal em VHDL é bastante geral e complexo. Tratando-se de uma linguagem pensada para a simulação discreta controlada por acontecimentos, assume-se que todos os sinais se propagam só numa direcção e que algum atraso está sempre envolvido.

A escala temporal pode ser analisada sobre duas perspectivas, uma que mede o tempo real de simulação e outra que representa o tempo de cálculo de um ciclo de simulação. Este último é chamado de *delta delay* e pode ser visto como um atraso infinitesimamente pequeno mas diferente de zero que é incrementado cada vez que um ciclo de simulação termina, no mesmo instante de tempo real.

Os valores projectados para o futuro de uma onda são contidos nos *drivers*. Por cada *driver* associado a um sinal, devido a uma instrução de atribuição, podem existir dois modelos de atraso que controlam a forma como os novos pares instante/valor vão ser colocados nesse *driver*:

Atraso Inercial - é o modelo de defeito em VHDL. Significa que para que um dado valor se repercuta na saída, ele tem que permanecer um determinado tempo mínimo na entrada.

Atraso de Transporte - este modelo é idêntico aos atrasos que sofre uma corrente eléctrica ao atravessar um fio, qualquer conjunto de impulsos na entrada

são reflectidos na saída.

3.3.4 As Instruções

As instruções especificam o modo de organização e de operação de um circuito. Uma vez que os dispositivos electrónicos funcionam na sua maior parte em paralelo, o VHDL inclui capacidades de concorrência. Assim o projectista tem dois níveis para especificar o seu sistema: o nível sequencial e o nível concorrente.

Instruções Sequenciais

As instruções sequenciais são usadas quando se pretende descrever um circuito utilizando um estilo de descrição comportamental. Todos as instruções sequenciais estão encapsuladas dentro de processos, subprogramas ou funções, que são usadas em contextos concorrentes. As instruções sequenciais especificam o algoritmo passo a passo e tal como nas linguagens de programação de alto nível existem construções do tipo if, case, ciclos, procedimentos, atribuição a variáveis, etc, mas com a sua sintaxe própria. Sendo o VHDL no entanto uma linguagem de descrição de *hardware* possui algumas construções próprias:

Atribuição Sequencial a Sinais - permitem "propor" os valores futuros para os sinais. Como se verá mais à frente estes valores poderão não vir a ser os futuros valores que os sinais irão tomar.

Instrução de `wait` - suspende a execução do algoritmo e espera por um conjunto de condições. Essas condições podem ser a sensibilidade à variação de sinais, uma condição lógica dos seus valores, o expirar de um determinado tempo ou uma mistura de quaisquer das três condições anteriores.

Instruções Sequenciais de `assertion` - verificam se uma determinada condição é verificada, caso contrário reportam um erro.

Instruções Concorrentes

As instruções concorrentes são executadas assincronamente, isto é, nada pode ser assumido quanto à sua ordem de execução. São estas instruções que permitem descrever um circuito sob a forma estrutural ou de fluxo de dados:

- **Descrição Estrutural**

  Instrução Bloco - permite que um conjunto de instruções concorrentes sejam agrupadas numa unidade lógica para descrever uma parte do circuito.

  Instruções de Instanciação de Componentes - faz a chamada a um outro componente existente na biblioteca e associa sinais ao seus portos de interface.

  Instrução de Geração - fornece um mecanismo iterativo e/ou condicional para a criação de parte do circuito.

- **Descrição de Fluxo de Dados**

  Instrução Concorrente de Atribuição a Sinais - cria um novo *driver* por cada atribuição, onde os novos valores propostos para o sinal são guardados. A existência de múltiplos *drivers* para um único sinal implica que este tenha associado uma função de resolução para determinar qual o valor que o sinal vai ter de facto. Esta função tem que ser definida pelo utilizador e é indicada quando se declara o tipo do sinal.

  Instrução de Processo - permite de uma forma independente e sequencial descrever o comportamento de uma parte do circuito. Um processo em VHDL pode ser visto como um conjunto de instruções sequenciais que são executadas em ciclo infinito. O controlo da execução deste ciclo pode ser feito ou pela lista de sensibilidade do processo (o processo só é executado se houver uma alteração no valor de alguns dos sinais que constituem essa lista) ou através da instrução sequencial de `wait` (interrompendo a execução do ciclo enquanto a condição do `wait` for falsa).

  Chamadas Concorrentes de Procedimentos - permitem que estes sejam executados de uma forma concorrente sempre que um dos seus parâmetros de entrada variar.

  Instruções Concorrentes de `assertion` - têm o mesmo significado das instruções sequenciais de `assertion` mas usadas num meio concorrente.

3.4 Conclusões

Neste capítulo foram apresentados alguns benefícios da representação de um circuito digital através de uma linguagem de descrição de *hardware* e as características que destinguem estas linguagens das usuais linguagens de programação de alto-nível. Foram também referidas as linguagens de descrição de *hardware* mais importantes, donde se destacou o VHDL. Para esta linguagem, foi feita uma descrição sumária das suas principais características.

No próximo capítulo será analisado como a linguagem VHDL pode ser usada num ambiente de síntese.

Capítulo 4

VHDL para Síntese

Neste capítulo é analisada a utilização da linguagem VHDL para a descrição de circuitos digitais num ambiente de síntese. Descreve-se o fluxo de projecto usado e referem-se as restrições impostas às descrições para que seja possível a sua síntese pelas actuais ferramentas. Algumas destas restrições resultam do VHDL ter sido pensado para outras aplicações que não a síntese, outras provêm da "capacidade limitada" de síntese que as actuais ferramentas oferecem. Apresentam-se ainda a forma como são sintetizadas as descrições de VHDL e o impacto que o estilo de descrição pode ter na qualidade do circuito sintetizado. Salienta-se também a influência que o projectista pode ter na interacção com a ferramenta de síntese para obter o circuito desejado.

4.1 Fluxo de Projecto

O processo de síntese da lista de portas lógicas que implementam um circuito a partir da sua descrição numa linguagem de alto nível não se efectua num único passo. O fluxo de projecto habitualmente seguido na síntese de circuitos digitais a partir de uma descrição VHDL é composto de quatro passos (ver figura 4.1):

1. Obter uma descrição do circuito em VHDL. Esta descrição deve ser feita num nível de abstracção que permita libertar o projectista dos pormenores de implementação e ao mesmo tempo possa ser sintetizada pelas actuais ferramenta de síntese.

2. Validar a descrição obtida anteriormente através de um simulador de VHDL.

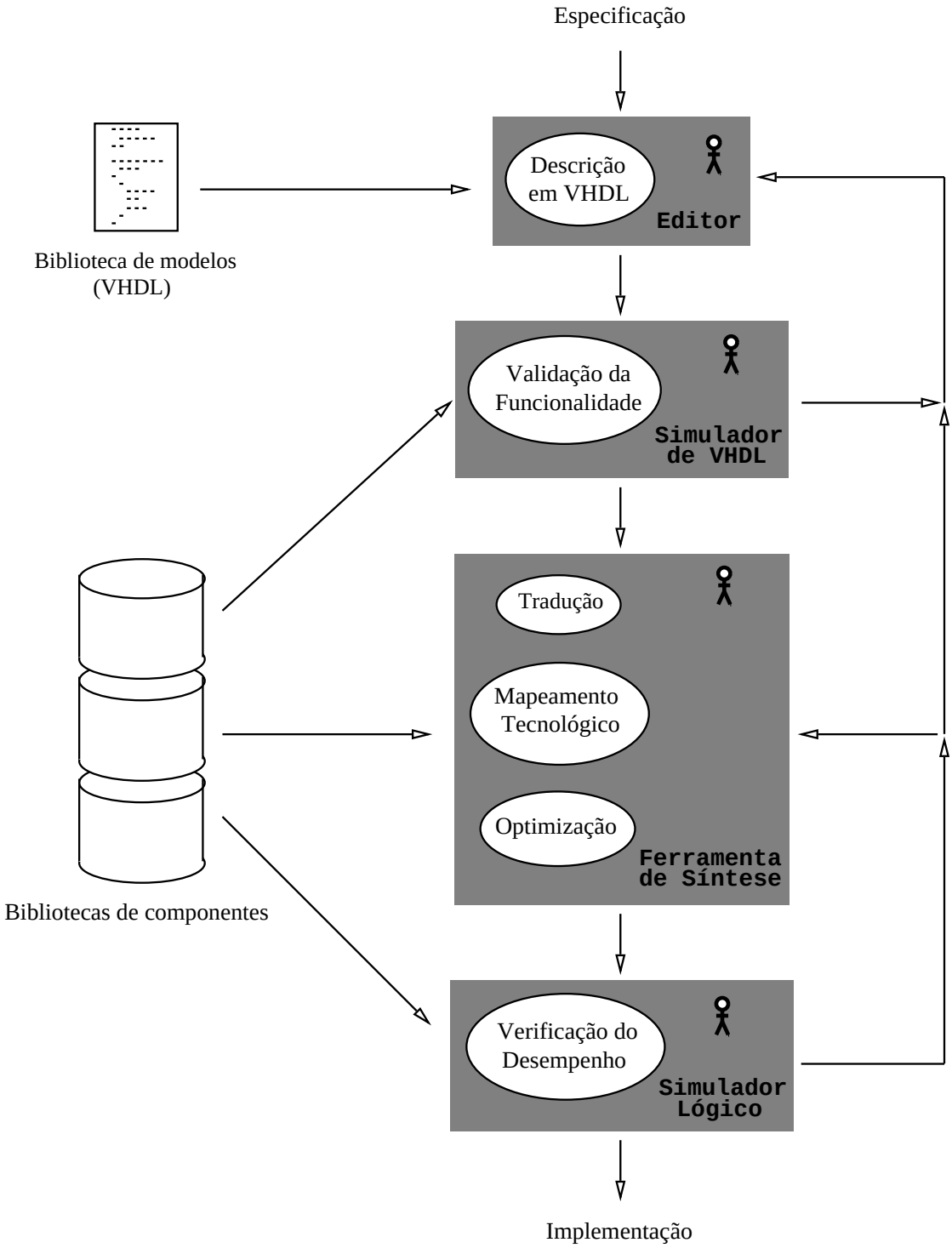


Figura 4.1: Fluxo de projecto

Este passo do projecto garante que a funcionalidade desejada pelo projectista se encontra na descrição VHDL.

3. Utilizar uma ferramenta de síntese para obter uma representação do circuito ao nível lógico que satisfaça um conjunto de restrições impostas pelo projectista.

4. Verificar através de um simulador lógico que o circuito sintetizado tem as características pretendidas pelo projectista.

Sempre que é obtida uma descrição do circuito é necessário verificar a sua funcionalidade através de uma simulação. Quando os resultados da simulação não satisfizerem por completo os intentos do projectista é necessário voltar a um passo anterior, efectuar algumas alterações e repetir novamente os restantes passos. Este processo iterativo pode ser efectuado a três níveis:

- Ao nível da linguagem de descrição: enquanto a descrição do circuito não apresentar a funcionalidade desejada ou não for sintetizável será necessário reescrever o código VHDL.

- Ao nível da ferramenta de síntese: quando as alterações ao circuito são obtidas através da mudança de parâmetros na ferramenta de síntese até serem respeitadas as restrição do projectista.

- A nível global: se o resultado da simulação lógica não apresentar o desempenho ou a funcionalidade pretendida poderá ser necessário reescrever o VHDL e repetir novamente todos os passos de projecto.

A iteração a nível global não seria necessária se as ferramentas de síntese garantissem a geração de circuitos funcional e temporalmente correctos. No entanto, há duas razões que justificam a simulação lógica após a síntese e portanto a necessidade deste ciclo de iteração. A primeira resulta da possibilidade de diferentes interpretações semânticas serem dadas a certas construções por parte do simulador de VHDL e da ferramenta de síntese. A segunda por, só após o mapeamento tecnológico resultante da síntese, ser possível uma simulação lógica com estimativas mais realistas dos tempos de atraso.

O fluxo de projecto apresentado para além de ser iterativo é também interactivo. Isto é, durante o processo de síntese o projectista tem que "guiar" a ferramenta de síntese para obter circuitos de qualidade aceitável. Na secção 4.4

serão abordadas algumas formas do projectista interactuar com a ferramenta de síntese de modo a conseguir os resultados desejados.

As bibliotecas que permitem representar o circuito nos vários níveis de projecto (ver figura 4.1) apresentam características diferentes consoante o objectivo a que se destinam. A biblioteca de modelos é escrita em VHDL, de uma forma independente da tecnologia, e destina-se a auxiliar o projectista na tarefa de descrever e testar o seu circuito. A descrição de alguns elementos que podem pertencer a esta biblioteca é feita no capítulo 5. As bibliotecas de componentes têm como objectivo representar as portas lógicas existentes numa dada tecnologia, através da função que realizam e de um conjunto de parâmetros que variam consoante a ferramenta a que se destina a biblioteca. Enquanto que, por exemplo, para as ferramentas de síntese são importantes a área de cada componente e os tempos de propagação, para os simuladores só estes últimos são necessários. A utilização de várias representações não coerentes de uma mesma biblioteca de componentes pode aumentar o número de iterações a realizar ou mesmo tornar a tarefa de síntese impossível.

4.2 Construções 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 aritméticas com números reais e de permitir a utilização de ficheiros para entrada e saída torna-o muito conveniente e sofisticado para a modelação de sistemas mas requer capacidades irrealistas para as ferramentas de síntese [Levitan 89].

O subconjunto de construções a suportar para síntese não está ainda normalizado, dependendo actualmente da ferramenta de síntese usada. Nas descrições de circuitos apresentadas ao longo deste trabalho procurou-se, sempre que possível, utilizar construções que são suportadas pela maioria dos sistemas de síntese. No entanto, para exemplificar a síntese de circuitos a partir dessas descrições foi utilizada uma ferramenta de síntese comercial [Synthesis 91].

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.

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 suportadas para síntese:

- Os tipos `file`, as declarações de ficheiros e consequentemente o módulo `TEXTIO`. O uso destas construções permite, por exemplo, modelar memórias (ROMs ou FIFOs) que ao não serem sintetizadas pela ferramenta 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`.
- Os objectos do tipo `physical` e `floating`, cuja utilização requereria o suporte de um conjunto de operadores que as actuais ferramentas não têm capacidade de sintetizar.

O uso de ficheiros de configuração não é actualmente suportado por grande parte das ferramentas de síntese, pelo que uma única arquitectura deve ser fornecida à ferramenta por cada entidade. Se assim não acontecer, o sistema deve ter um método explícito de escolher aquela que irá ser sintetizada.

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 conjunto de sinais, têm uma semântica que as torna sensíveis a qualquer variação que estes apresentem. Este facto pode ser muito prático para a modelação de sistemas mas não é facilmente realizado em *hardware*. Por exemplo, os elementos de memória como os *flip-flops* reagem apenas ao flanco positivo ou negativo, mas não a ambos [Lis 89]. Assim, e com o objectivo de permitir a descrição de circuitos síncronos com um dado sinal (em geral o relógio), a instrução `wait` tem a sua utilização restringida às formas representadas na figura 4.2. Isto é, a "sensibilidade" da instrução `wait` é limitada apenas a um sinal e a um flanco. 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 4.2: Utilizações válidas da instrução wait num ambiente de síntese.`

- 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 4.3). 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

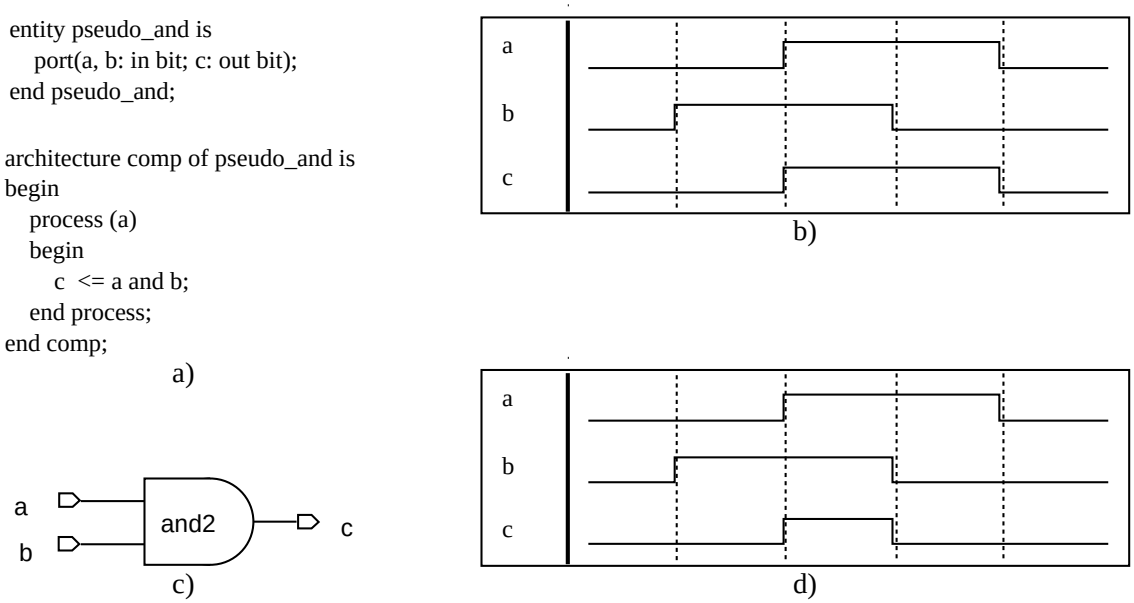


Figura 4.3: a) Código VHDL; b) Simulação VHDL; c) Circuito sintetizado; d) Simulação lógica.

da direita seja um potência de 2 calculável[1], enquanto que no operador ** essa mesma restrição é no operando da esquerda (base). Com estas restrições é possível a realização destes operadores através de registos de deslocamento.

- 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. Em geral tem que ser o projectista que, na interacção com a ferramenta de síntese, tem que garantir/exigir os tempos de atraso pretendidos.

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. Por exemplo, a condição de decisão a usar numa construção `if` está limitada aos tipos de dados permitidos pelo sistema.

As construções suportadas são:

[1] O seu valor pode ser determinado estaticamente.

- 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 módulos.
- 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.

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[2].

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.

A utilização de comentários sintéticos é feita com dois objectivos distintos: alterar de uma forma artificial a linguagem usada ou incluir na própria descrição do sistema comandos para a ferramenta de síntese. O exemplo mais comum e que praticamente todas as ferramentas de síntese implementam são os comentários sintéticos que permitem cancelar ou reinicializar a tradução de partes de código. Desta forma, é possível isolar para a ferramenta as construções não sintetizáveis ou que não se queiram sintetizar. Algumas ferramentas de síntese utilizam ainda os comentários sintéticos para implementar de certa forma construções de VHDL que não são suportadas, por exemplo, funções de resolução.

Como os comentários sintéticos só são interpretados por uma ferramenta em particular (para as outras ferramentas não passam de simples comentários), a sua utilização deve ser evitada.

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

A classificação apresentada reflecte o suporte que a maioria das ferramentas de síntese fazem das principais construções de VHDL. No entanto, existem "outras" construções cujo grau de suporte depende do nível de maturidade que a ferramenta de síntese atingiu. Este facto pode ser observado na tabela 4.1 onde se compara o suporte que três ferramentas comerciais de síntese fazem de algumas construções de VHDL [Selz 92].

Construção	SylcSyn (2.2f)	Synopsys (2.2a)	AutoLogic (8.1)
Genéricos	–	Só inteiros	–
Modos de acesso `InOut`	–	√	–
Procedimentos	Sem parâmetros	√	√
Números Reais	–	–	–
Agregados	–	√	–
`Alias`	–	–	–
Índices variáveis	–	√	√
Pedaços variáveis	–	–	–
Configurações	–	–	–
Operador *	Potência de 2	√	√
Operador / e `mod`	Potência de 2	Potência de 2	–
Operador ** e `rem`	–	Potência de 2	–
Blocos	–	√	√
Blocos com guarda	–	–	√
Múltiplos `waits`	√	–	–

– - Construção não suportada pela ferramenta de síntese.

√ - Construção suportada pela ferramenta de síntese.

Tabela 4.1: Comparação do suporte de algumas construções de três sistemas comerciais de síntese.

4.2.1 Exemplos de como as Descrições de VHDL são Interpretadas

Tão importante como saber qual o subconjunto de VHDL que é sintetizável, é saber como essas construções são mapeadas para o domínio estrutural. Assim, nesta secção apresentam-se alguns exemplos que ilustram como um circuito lógico é sintetizado a partir de uma descrição VHDL.

A síntese de cada entidade de projecto (par entidade-arquitectura em VHDL) produz um circuito lógico que tem as seguintes características:

- As suas entradas e saídas correspondem aos sinais que foram declarados nos acessos da entidade.
- A funcionalidade é aquela descrita pela arquitectura correspondente.

No exemplo da figura 4.4 apresenta-se um circuito com três entradas (`a, b` e `c`) e uma saída (`z`), que realiza a função lógica descrita na atribuição concorrente ao sinal de saída.

```
entity vhdl is
    port(a, b, c: in bit;                    -- acessos de entrada
         z: out bit);                        -- acessos de saida
end vhdl;

architecture logica of vhdl is
begin
    z <= (a and b) or c;                     -- funcao logica a implementar
end logica;
```

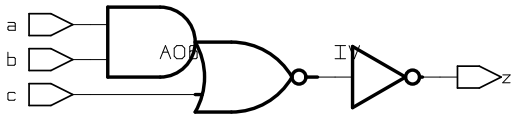


Figura 4.4: Resultado da síntese de uma atribuição concorrente em VHDL.

Se o projectista recorrer a uma descrição estrutural a ferramenta de síntese limita-se a fazer a instanciação e interligação dos vários componentes que constituem essa descrição. No entanto, a aceitação de uma descrição estrutural por parte da ferramenta de síntese tem algumas vantagens: para além de permitir uma descrição hierárquica do circuito e a utilização de células específicas da biblioteca, impondo alguma estrutura sobre o circuito, possibilita a optimização de circuitos provenientes de outras ferramentas que criem uma lista de portas em VHDL (por exemplo, editores de esquemáticos). Na figura 4.5 apresenta-se o resultado da "síntese" de uma descrição estrutural. Esta é constituída pela instanciação de dois componentes existentes na biblioteca tecnológica. Um *flip-flop* tipo D (FD1) e outro que realiza a função lógica $z = \overline{a.b + c.d}$ (AO2). Ilustra-se ainda neste exemplo o uso de um módulo para a declaração dos componentes que são utilizados na descrição do circuito.

```
package portas is

    component FD1                              -- declaracao de um flip-flop
        port(d, cp: in bit;
             q, qn: out bit);
    end component;

    component AO2                              -- declaracao de uma porta logica
        port(a, b, c, d: in bit;
             z: out bit);
    end component;

end portas;


use work.portas.all;

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


architecture estrutural of vhdl is
    signal estado, estado_neg: bit;        -- sinais auxiliares
begin
    u1: FD1                                -- instaciacao dos componentes
        port map(s, clk, estado, estado_neg);
    u2: AO2
        port map(a, estado, b, estado_neg, z);
end estrutural;
```

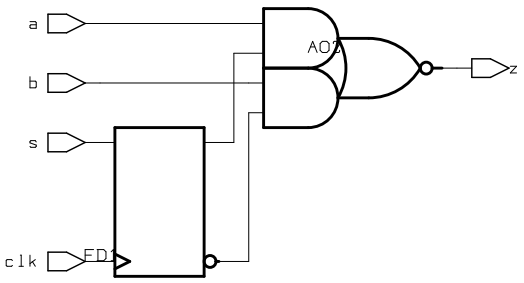


Figura 4.5: Instanciação de componentes e uso de módulos.

O uso de vectores permite de uma forma fácil a representação de *buses*. A posterior selecção de sinais do *bus* pode ser feita pela indexação do vector que o representa. Para a síntese desta indexação 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 sendo necessário a ferramenta de síntese recorrer ao uso de multiplexers. Ambos os casos estão representados no exemplo da figura 4.6.

```
entity vhdl is
    port(a, b: in bit_vector(0 to 3); -- acessos de 4 bits
         sel: in integer range 0 to 3; -- um intervalo de inteiros define
                                       -- o numero de bits do acesso
         saida_a, saida_b: out bit);
end vhdl;

architecture sel_vec of vhdl is
begin
    saida_a <= a(3);                   -- indice constante
    saida_b <= b(sel);                 -- indice variavel
end sel_vec;
```

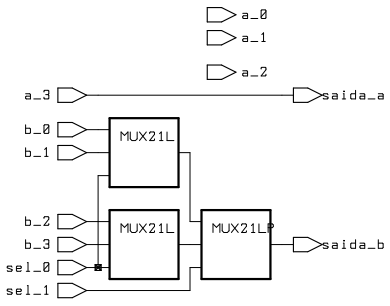


Figura 4.6: Selecção dum elemento de um vector.

No exemplo da figura 4.7 exemplifica-se o uso do operador aritmético soma para realizar a adição de dois inteiros. O circuito lógico obtido pela ferramenta de síntese é um somador de três *bits* com saída de transporte (`z_3`) que pode ser posteriormente optimizado em termos de área e/ou velocidade.

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 na figura 4.8.

A solução apresentada anteriormente não será tão simples se a atribuição a um sinal for feita apenas num dos ramos da instrução. Esta situação encontra-se representada no exemplo da figura 4.9 através de uma instrução `if` sem o ramo `else`. Neste caso o resultado da síntese será um *latch* activado pela condição de selecção do `if`. O uso deste elemento de memória é necessário porque enquanto a condição do `if` for falsa o sinal tem que manter o valor anterior.

A instrução `case`, quando usada com um único processo, produz resultados idênticos aos do `if` resultando neste caso a condição de selecção dos multiplexers da avaliação da expressão do `case`. Mais à frente será apresentada a utilização da instrução `case` noutras situações.

Na figura 4.10 apresenta-se o resultado da síntese de um ciclo `for` com os limites determinados 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.

A descrição de circuitos sequenciais síncronos, que utilizarão elementos de memória (*flip-flops*), pode ser feita recorrendo ao uso da instrução `wait`. A figura 4.11 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. Esta descrição corresponde a um *flip-flop* do tipo T que é sintetizado à custa dos elementos existentes na biblioteca, neste caso um *flip-flop* tipo D e uma porta lógica *xor*.

A representação de circuitos com *flip-flops* pode ser também conseguida utilizando a instrução `if`. Na figura 4.12 apresenta-se a síntese de um *flip-flop* com *reset* assíncrono a partir do encadeamento de duas instruções `if`. A primeira representa o *reset* e a segunda corresponde à descrição do funcionamento síncrono do circuito. Se se pretender um *reset* síncrono, pode-se recorrer à descrição da figura 4.13. Nesta, as instruções que descrevem o funcionamento normal do circuito ou o *reset* só são executadas no flanco ascendente do relógio.

```
entity vhdl is
    port(a, b: in integer range 0 to 7; -- uso de inteiros nos
         z: out integer range 0 to 14);  -- acessos de entrada e saida
end vhdl;

architecture aritmetica of vhdl is
begin
    z <= a + b;                          -- uso do operador adicao
end aritmetica;
```

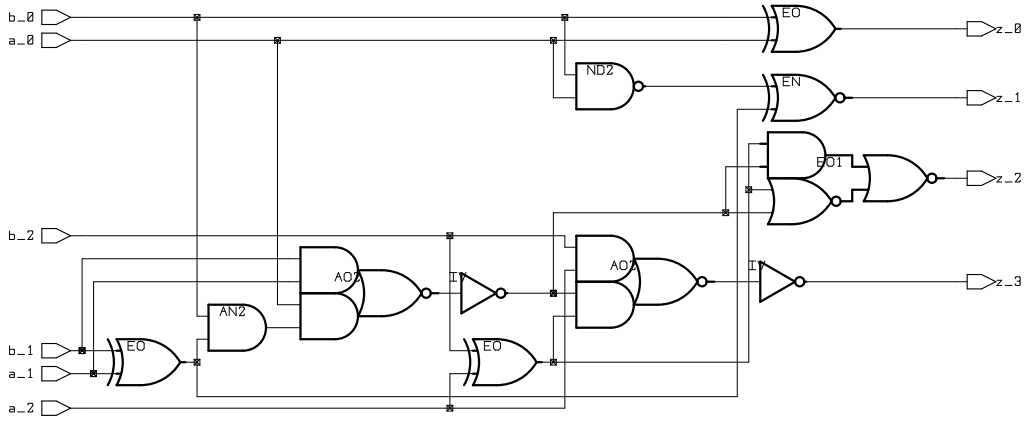


Figura 4.7: Uso do operador aritmético '+' com inteiros.

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

architecture proc_if of vhdl is
begin
    process(sel_a, a, b)
    begin
        if sel_a = '1' then          -- instrucao if
            z <= a;
        else                         -- com ramo else
            z <= b;
        end if;
    end process;
end proc_if;
```

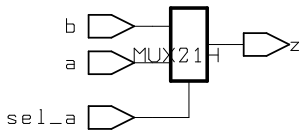


Figura 4.8: Síntese de um `if-then-else` dentro de um processo.

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

architecture proc_if of vhdl is
begin
    process(sel_a, a)
    begin
        if sel_a = '1' then
            z <= a;                  -- z so' e' actualizado se a
        end if;                      -- condicao do if for verdadeira
    end process;
end proc_if;
```

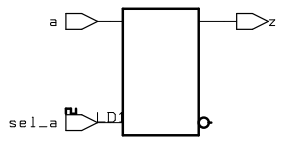


Figura 4.9: Síntese de um `if` sem o ramo `else`.

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

architecture proc_for of vhdl is
begin
    process(a)
        variable par_temp: bit;     -- variavel temporaria
    begin
        par_temp := '0';
        for i in a'range loop       -- utilizacao de um ciclo for
            par_temp := par_temp xor a(i);
        end loop;
        paridade <= par_temp;
    end process;
end proc_for;
```

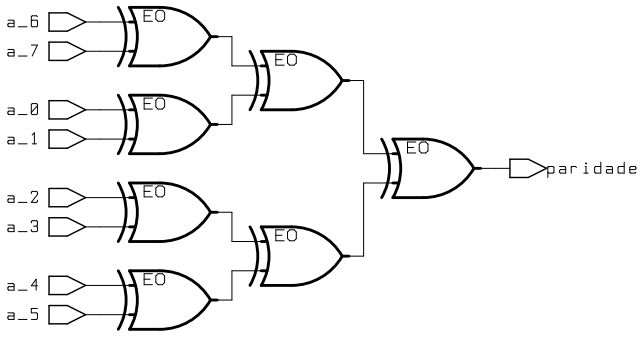


Figura 4.10: Uso do ciclo `for` recorrendo ao atributo `range` .

No exemplo apresentado na figura 4.14 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, volta a zero.

A descrição de máquinas de estado pode ser feita em VHDL de uma forma relativamente fácil 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 estados e as respectivas actualizações dos valores das saídas.

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

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

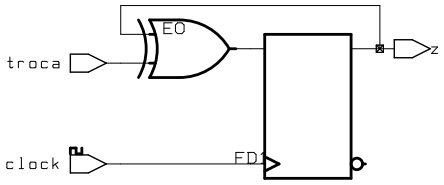


Figura 4.11: Uso da instrução `wait` para a geração de circuitos sequenciais.

No exemplo da figura 4.15 apresenta-se uma máquina de estados de Moore que permite detectar a sequência de entrada `"111"`. A tabela de transição de estados desta máquina, representada na tabela 4.2, 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 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 4.15 com a etiqueta `comb`.

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

Tabela 4.2: Tabela de transição da máquina de estados (máquina de Moore).

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


architecture proc_reset of vhdl is
begin
    process(clock, reset)
    begin
        if reset = '1' then          -- reset assincrono
            z <= '0';
        elsif clock'event and clock = '1' then
            z <= dados;              -- funcionamento sincrono
        end if;
    end process;
end proc_reset;
```

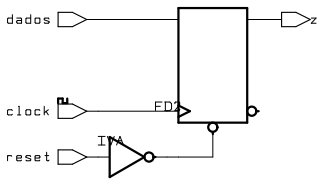


Figura 4.12: Síntese de um *flip-flop* com *reset* assíncrono.

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


architecture proc_reset of vhdl is
begin
    process(clock)
    begin
        if clock'event and clock = '1' then
            if reset = '1' then
                z <= '0';          -- reset sincrono
            else
                z <= dados;        -- funcionamento sincrono normal
            end if;
        end if;
    end process;
end proc_reset;
```

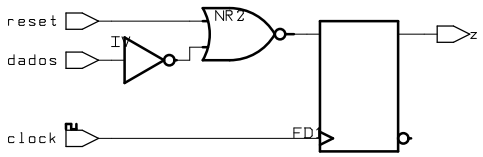


Figura 4.13: Síntese de um *flip-flop* com *reset* síncrono.

```
entity vhdl is
    port(clock, reset: in bit;
         count: buffer integer range 0 to 9);
end vhdl;       -- utilizacao de acessos tipo buffer


architecture contador of vhdl is
begin
    process(reset, clock)
    begin
        if reset = '1' then             -- reset assincrono
            count <= 0;
        elsif clock'event and clock = '1' then
            if count = 9 then
                count <= 0;             -- re-inicio da contagem
            else
                count <= count + 1;  -- contagem normal
            end if;
        end if;
    end process;
end contador;
```

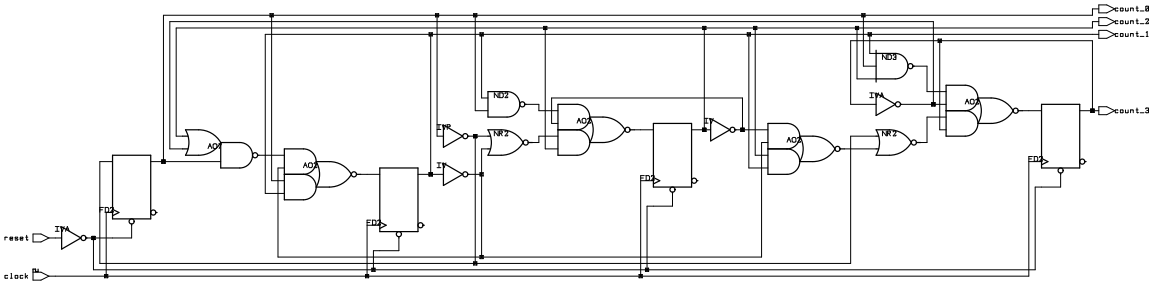


Figura 4.14: Contador de módulo 10 com *reset* assíncrono.

O processo identificado com a etiqueta `sinc` é responsável pelo funcionamento síncrono da máquina de estados. É incluído também um `if` que representa o *reset* assíncrono dos elementos de memória.

A descrição do mesmo circuito usando uma máquina de estados de Mealy (cuja tabela de transição de estados se apresenta na tabela 4.3) pode ser feita usando a descrição da figura 4.16. Esta descrição difere da apresentada anteriormente apenas nos ramos da instrução `case` correspondentes ao estados `dois` e `tres`. Nestes, a definição da saída não está só dependente do estado actual, mas também do valor das entradas do circuito. Por isso a atribuição à saída `sequencia` é feita conjuntamente com a definição do próximo estado, isto é, nos ramos da instrução `if`. No circuito sintetizado esta alteração reflecte-se na forma como a saída é obtida, passando agora a depender também da entrada, como seria de esperar para máquina de Mealy.

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

architecture Moore of cir is
  type estados is (zero, um, dois, tres); -- uso do tipo enumerado
  signal estado_actual,                   -- para as variaveis de
         proximo_estado: estados;         -- estado
begin

  comb: process(entrada, estado_actual)   -- logica combinatoria
  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)             -- funcionamento sincrono
  begin
    if reset = '1' then
      estado_actual <= zero;
    elsif clock'event and clock = '1' then
      estado_actual <= proximo_estado;
    end if;
  end process;

end Moore;
```

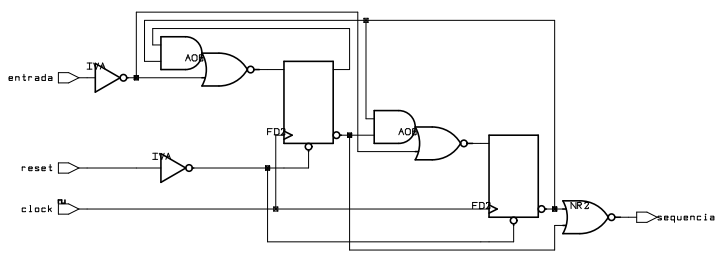


Figura 4.15: Máquina de estados de Moore que detecta a sequência "111".

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

architecture Mealy of cir is
  type estados is (zero, um, dois, tres); -- uso do tipo enumerado
  signal estado_actual,                   -- para as variaveis de
         proximo_estado: estados;         -- estado
begin

  comb: process(entrada, estado_actual)   -- logica combinatoria
  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 =>
        if entrada = '1' then             -- a saida passou a
          sequencia <= '1';               -- depender da entrada
          proximo_estado <= tres;
        else
          sequencia <= '0';
          proximo_estado <= zero;
        end if;
      when tres =>
        if entrada = '1' then             -- a saida passou a
          sequencia <= '1';               -- depender da entrada
          proximo_estado <= tres;
        else
          sequencia <= '0';
          proximo_estado <= zero;
        end if;
    end case;
  end process;

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

end Mealy;
```

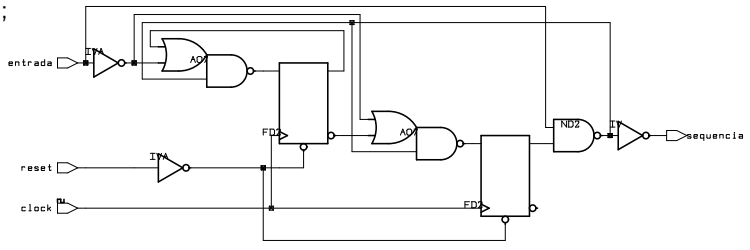


Figura 4.16: Máquina de estados de Mealy que detecta a sequência "111".

Estado Actual	Próximo Estado/Saída(`sequencia`)	
	`entrada` = 0	`entrada` = 1
zero	zero/0	um/0
um	zero/0	dois/0
dois	zero/0	três/1
tres	zero/0	três/1

Tabela 4.3: Tabela de transição da máquina de estados (máquina de Mealy).

A utilização de uma lógica multi-valor, que inclua uma representação do "valor lógico" alta impedância, permite a descrição de elementos com saídas em alta impedância. Os tipos de dados que representam a lógica multi-valor podem estar pré-definidos pela ferramenta de síntese (sendo disponibilizados através de módulos) ou serem definidos pelo próprio projectista. Neste último caso, é necessário indicar à ferramenta qual a correspondência entre as representações definidas e os valores lógicos '0', '1' e alta impedância.

No exemplo da figura 4.17 é ilustrada a utilização da lógica multi-valor na descrição de um *buffer* com saída em alta impedância. O tipo de dados que representa a lógica multi-valor é definido no módulo `tipos`, onde são também indicados à ferramenta de síntese os "valores lógicos" que representam cada um dos elementos desse tipo. Esta indicação é feita através de um atributo definido pelo utilizador (`enum_encoding`) e reconhecido pela ferramenta de síntese que está associado ao tipo de dados definido. Assim, os valores `Zero`, `Um` e `Alta_Imped` são interpretados como os valores lógicos '0', '1' e alta impedância (representado para a ferramenta de síntese por `Z`), respectivamente.

4.3 Estilos de Descrição em VHDL

A adopção de um estilo de descrição apropriado permite reduzir o tempo necessário para obter a descrição a nível lógico do circuito final. Esta economia pode ser feita através da redução do tempo de cada passo do projecto ou pela diminuição do número de iterações necessárias.

A escolha de um estilo de descrição de VHDL é realizada em duas fases: encontrar o subconjunto de instruções suportadas pelas várias ferramentas usadas e determinar qual o estilo de escrita propriamente dito, de forma a obter os melhores resultados dessas ferramentas. Como a mesma descrição do circuito tem de

```
package tipos is
    type logica_3_valores is (Zero, Um, Alta_Imped);
                              -- declaracao e especificacao de um atributo
    attribute enum_encoding: string;
    attribute enum_encoding of logica_3_valores: type is "0 1 Z";
end tipos;

use work.tipos.all;
entity cir is
    port(a, sel_a: in logica_3_valores;
         z: out logica_3_valores);
end cir;

use work.tipos.all;
architecture alta_impedancia of cir is
begin
    process(sel_a, a)
    begin
        if sel_a = Um then
            z <= a;
        else
            z <= Alta_Imped; -- saida em alta impedancia
        end if;
    end process;
end alta_impedancia;
```

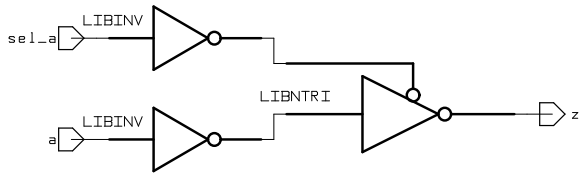


Figura 4.17: Descrição de um *buffer* com alta impedância usando lógica multi-valor.

servir para a simulação e síntese, o estilo de descrição a adoptar deve tomar em consideração ambas as ferramentas.

Conforme já referido anteriormente, a semântica do VHDL está adaptada à simulação, razão pela qual praticamente todos os simuladores suportam a linguagem na totalidade. Aqueles que não o fazem deixam de fora apenas um conjunto pequeno de instruções que também não são certamente suportadas para síntese. Assim, o estilo de descrição deve seguir, tanto quanto possível, as restrições impostas pelas ferramentas de síntese.

O estilo de descrição "imposto" pelo simulador tem como objectivo reduzir o tempo de simulação de um circuito. Este objectivo pode ser conseguido através de

certas optimizações do código que poderiam ser desnecessárias se os "compiladores" de VHDL[3] estivessem tão desenvolvidos como os de C ou Pascal. Assim, a melhor forma de garantir o desempenho do simulador consiste em descrever o circuito tendo em conta quer as optimizações usuais das linguagens de programação de alto nível, quer as optimizações específicas da linguagem VHDL. As optimizações mais comuns são:

Optimizações genéricas das linguagens de alto nível

- Utilização de variáveis temporárias para evitar o cálculo desnecessário de expressões comuns a várias instruções.
- Eliminação do cálculo repetitivo e desnecessário de expressões que não dependam do ciclo onde estão inseridas. Isto pode ser conseguido quer pela utilização de variáveis temporárias ou simplesmente pela passagem de uma expressão para fora dum ciclo.
- Estruturação das sequências encadeadas de condições de forma que a condição mais provável de acontecer esteja no topo. Assim, a maioria das vezes apenas será necessário verificar uma única condição para determinar qual a acção a desencadear.
- Substituição das chamadas a funções que são executadas muitas vezes (por exemplo, por estarem dentro do ciclos) pelo código da própria função. Desta forma evita-se a sobrecarga resultante da passagem de parâmetros sempre que a função é chamada.

Optimizações específicas de VHDL

- Utilização de variáveis em vez de sinais locais. Devido aos sinais serem internamente representados por uma estrutura de dados complexa, que necessita de ser ordenada no tempo (consoante os seus acontecimento programados para o futuro) e poderem ter funções de resolução associadas (se existirem vários *drivers*), o seu uso deve ser evitado. No seu lugar devem ser usadas

[3]Uma descrição de VHDL pode ser vista como um programa que tem que ser compilado para correr numa dada arquitectura de *hardware*. Os actuais simuladores de VHDL compilam a descrição do circuito ou para uma linguagem de alto nível como o C, que posteriormente vai ser passada para código máquina, ou para um formato intermédio que é interpretado durante a simulação.

variáveis que requerem menos memória e são mais facilmente manipuladas pelo simulador.

- Os sinais da lista de sensibilidade de um processo devem ser cuidadosamente escolhidos. Não devem ser colocados na lista de sensibilidade sinais cujas mudanças de valor vão forçar a execução do processo sem no entanto alterarem variáveis ou sinais no seu interior. Por exemplo, os processos síncronos com o relógio não necessitam em geral de serem sensíveis aos dados.

- Substituição do atributo `stable` por `event` sempre que possível. Enquanto que o primeiro é implementado através de um sinal auxiliar, o segundo é apenas uma chamada a uma função, sendo por isso preferível.

O estilo de descrição a adoptar para a síntese tem como objectivo reduzir o número de iterações necessárias para obter uma implementação final do circuito com o desempenho e funcionalidade desejada. No estado actual da tecnologia de síntese, a qualidade final do circuito sintetizado, em termos de área ou velocidade, depende tanto da capacidade da ferramenta usada, como da forma e qualidade da descrição de VHDL que é fornecida [Carlson 91].

Presentemente as ferramentas comerciais de síntese apenas aceitam descrições no nível de abstracção RTL ou inferior. No entanto, a linguagem VHDL permite descrições em níveis de abstracção superiores, mesmo quando restringida ao subconjunto de instruções sintetizáveis apresentado na secção 4.2. Por esta razão, o projectista tem que ter a estrutura do seu circuito representada ao nível RTL, isto é, constituída pelo(s) fluxo(s) de dados, com elementos de funções bem definidas (registos, somadores, comparadores, etc) e o(s) respectivo(s) controlador(es).

A melhor forma de representar essa estrutura consiste na utilização de várias entidades de projecto, uma para cada elemento da representação RTL. Estas devem ser instanciadas e interligadas através de uma descrição estrutural de forma a representarem a arquitectura pretendida para o circuito (a nível RTL). Assim, a descrição do comportamento do circuito "resume-se" a dois tipos de representações: os elementos que constituem o fluxo de dados e os controladores, que em geral são implementados por máquinas de estado.

Na descrição dos elementos que constituem o fluxo de dados devem ser tomados em conta dois factores: as discrepâncias existentes entre uma representação em VHDL e o circuito sintetizado e que, a "qualidade" deste está directamente relacionada com o estilo de descrição e as optimizações feitas nesta, em especial aquelas que se reflectem na estrutura dos elementos a implementar.

Assim, os elementos pertencentes ao fluxo de dados devem ser descritos atendendo às seguintes recomendações:

- Evitar o uso de descrições, cujas simulações em VHDL estão correctas por serem feitas com atrasos nulos, mas que quando implementadas ao nível lógico poderão não funcionar devido aos atrasos introduzidos pelas portas lógicas. Por exemplo, estão neste caso as descrições que podem originar lógica combinatória com realimentações (circuito sequenciais assíncronos).

- Incluir na lista de sensibilidade de um processo todos os sinais que esse processo usa em modo leitura e cuja mudança de valor possa provocar alterações nas variáveis ou sinais que estão no seu interior.

- Providenciar um sinal *reset* ou *set* a todos os elementos de memória do circuito para garantir que o circuito evolui a partir de um estado determinado depois de ser ligado.

- Usar as construções `if` e `wait`, conforme apresentadas na secção 4.2.1, para descrever lógica síncrona, com *set* e/ou *reset* assíncrono. Outras formas de descrever o circuito podem levar a que a ferramenta não aceite essas descrições para síntese. Por exemplo, a descrição de um *flip-flop* tipo D com *reset* assíncrono apresentada na figura 4.18 não é sintetizável pela ferramenta de síntese [Synthesis 91]. Nesta descrição, a instrução `if-then-elseif` foi separada em dois `ifs` e utilizando uma variável temporária (`var_aux`). O erro[4] indicado pela ferramenta resulta de haver atribuições à variável `z` realizadas de uma forma assíncrona e síncrona com um dado sinal, neste caso o relógio (`clock`).

- Usar parêntesis para agrupar operadores, forçando uma dada ordem de avaliação de uma expressão e portanto uma certa estrutura no circuito sintetizado. O exemplo da figura 4.19 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. O circuito sintetizado neste último caso é mais rápido e mais pequeno, pois utiliza dois operadores de quatro entradas em "paralelo" seguido de um de cinco em vez de, um operador de quatro entradas e dois de cinco todos em "série". A optimização de ambos os circuitos pela ferramenta de síntese [Synthesis 91] produziu ao nível lógico circuitos semelhantes (em termos de área e tempos de propagação), tendo-se porém melhores resultados no segundo caso.

[4] `Error: Ilegal assignment to 'z'. It depends on a non-edge.`

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

architecture proc_reset of vhdl is
begin
    process(clock, reset)
    variable var_aux: bit;
    begin
        var_aux := dados;
        if reset = '1' then            -- reset assincrono
            var_aux := '0';
            z <= var_aux;
        end if;
        if clock'event and clock = '1' then
            z <= var_aux;              -- funcionamento sincrono
        end if;
    end process;
end proc_reset;
```

Figura 4.18: Descrição de um *flip-flop* D que não é sintetizável.

- Utilizar variáveis temporárias para calcular sub-expressões comuns evitando assim o uso supérfluo de recursos que podem ser partilhados. Isto é, a utilização de operadores complexos como somadores, comparadores, multiplicadores e outros deve ser feita de forma a evitar a sua duplicação desnecessária. Para isso, deve-se sempre que possível usar variáveis temporárias para colocar em evidência factores comuns em expressões que envolvam este operadores. Na figura 4.20 apresenta-se uma descrição em que este tipo de optimização não foi tido em conta, por isso o circuito sintetizado é maior que aquele representado na figura 4.21 onde apenas existe uma operação de adição.

  Note-se que na ferramenta de síntese [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.

- Descrever os circuitos lógicos cujas saídas não estão completamente especificadas para todas as combinações de entrada, de forma a permitir à ferramenta de síntese tirar partido dessa informação na fase de optimização lógica. Para isso é necessário recorrer a uma lógica multi-valor, que tenha uma representação do "valor lógico" indiferença, para indicar à ferramenta

```
entity vhdl2 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 vhdl2;

architecture aritmetica of vhdl2 is
begin
    x <= a + b + c + d;
    z <= (e +f) + (g + h);
end aritmetica;
```

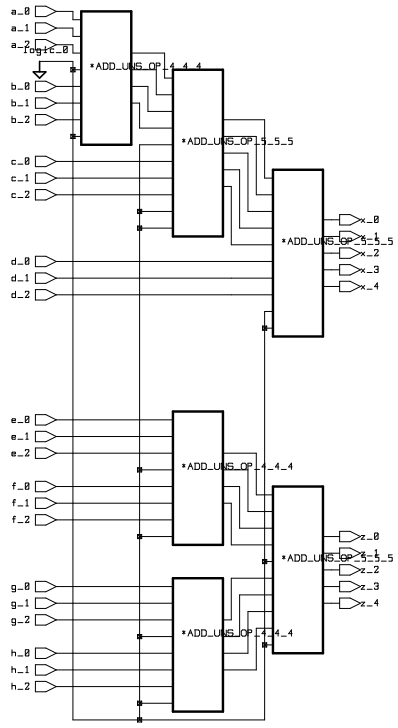


Figura 4.19: O uso de parêntesis como forma de estruturar o circuito.

quando as saídas, ou outros sinais, podem tomar qualquer um dos valores lógicos: '0' ou '1'. Os tipos de dados que representam essa lógica multi-valor podem estar previamente definidos pela ferramenta de síntese ou serem definidos pelo utilizador. Neste caso, e tal como aconteceu para a utilização do "valor lógico" de alta impedância, tem que ser indicado à ferramenta qual a correspondência entre as representações utilizadas e os valores lógico '0', '1' e indiferença (por exemplo, através de atributos definidos pelo utilizador).

Nos exemplos apresentados nas figuras 4.22 e 4.23 ilustra-se como o uso de indiferenças pode influenciar a qualidade do circuito final. Em ambos

```
entity vhdl_2 is
    port(sel_a: in bit;
         a, b, c: in integer range 0 to 3;
         z: out integer range 0 to 6);
end vhdl_2;

architecture soma of vhdl_2 is
begin
    process(a, b, c)
    begin
        if sel_a = '1' then
            z <= a + c;
        else
            z <= b + c;
        end if;
    end process;
end soma;
```

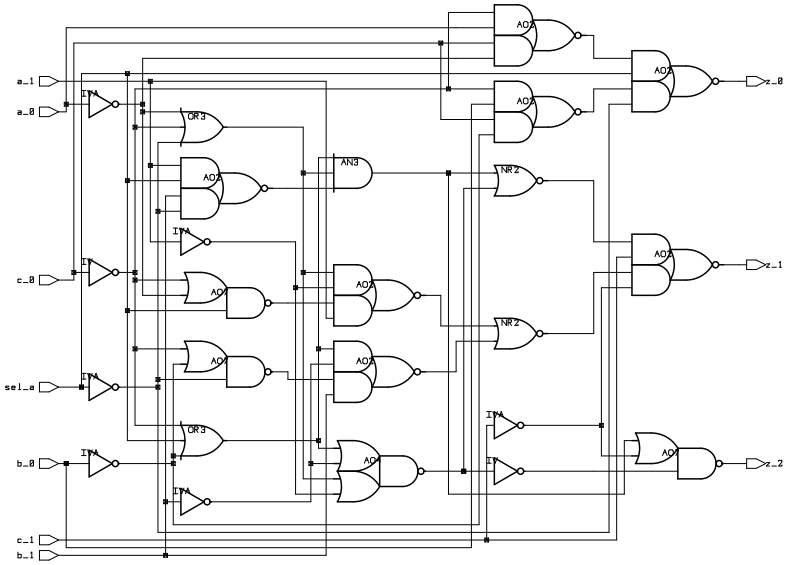


Figura 4.20: Uso de adições desnecessárias produz circuitos maiores.

o casos é descrito um conversor do código BCD[5] para um mostrador de sete segmentos. Enquanto que na primeira descrição foi forçado um valor arbitrário (no exemplo, o valor zero) para as combinações não utilizadas do código BCD, na segunda, a essas combinações foi atribuído o "valor lógico" indiferença (no exemplo representado por '-'). Os circuitos sintetizados a partir destas descrições realizam ambos a função de conversão desejada mas, o segundo circuito (figura 4.23) apresenta uma área menor.

A descrição de máquinas de estados em VHDL pode ser feita de muitas formas,

[5]Do inglês *Binary Coded Decimal*.

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

architecture soma of vhdl is
begin
    process(a, b, c)
    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;
```

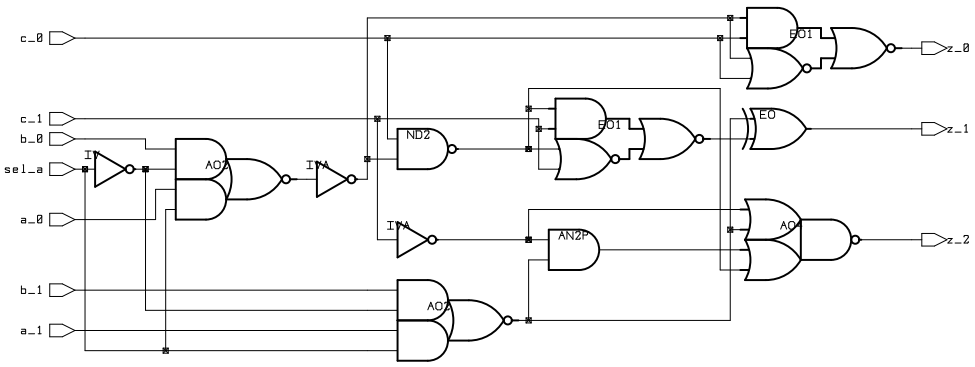


Figura 4.21: Evidenciando factores consegue-se reduzir a área do circuito.

por não existir uma construção específica para a sua representação. No entanto, na descrição sintetizável de máquinas de estado deve ser utilizado o formato apresentado na secção 4.2.1 (figuras 4.15 e 4.16), donde se salienta a utilização das seguintes construções:

- Declaração de um 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 codifica-

```
entity cir1 is
    port(bcd: in bit_vector(3 downto 0);
        sete_seg: out bit_vector(6 downto 0));
end cir1;

architecture conversor of cir1 is
begin
    process(bcd)
    begin
        case bcd is
            when "0000" => sete_seg <= "1111110";
            when "0001" => sete_seg <= "1100000";
            when "0010" => sete_seg <= "1011011";
            when "0011" => sete_seg <= "1110011";
            when "0100" => sete_seg <= "1100101";
            when "0101" => sete_seg <= "0110111";
            when "0110" => sete_seg <= "0111111";
            when "0111" => sete_seg <= "1100010";
            when "1000" => sete_seg <= "1111111";
            when "1001" => sete_seg <= "1110111";
            when others => sete_seg <= "0000000";
-- saida forcada a 0
        end case;
    end process;
end conversor;
```

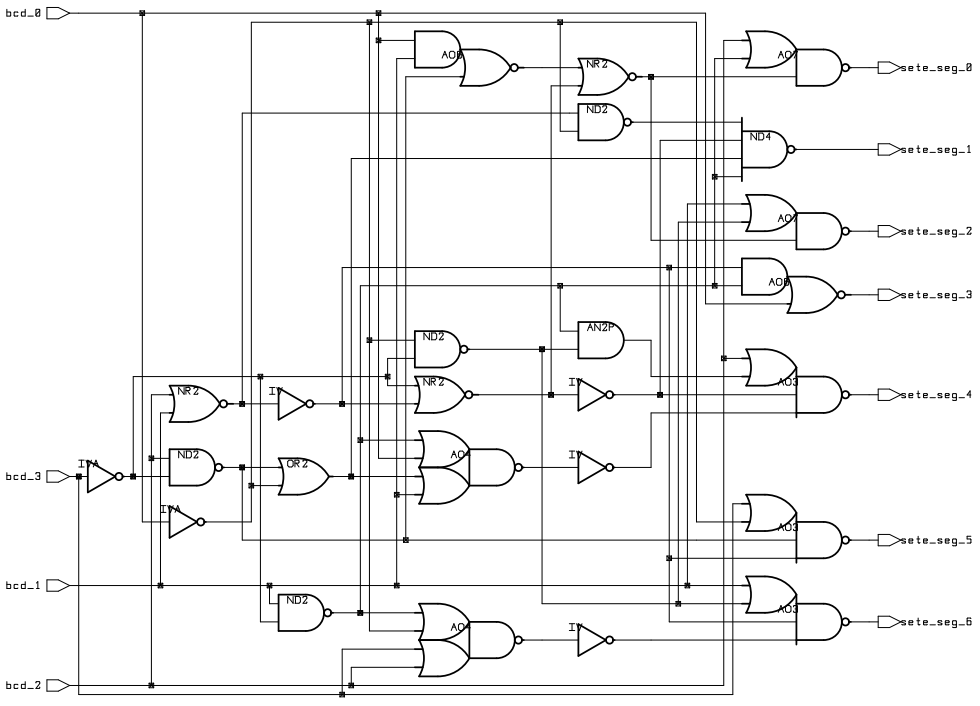


Figura 4.22: Descrição de um conversor do código BCD para um mostrador de sete segmentos.

```
package tipos is
    type bit_3 is ('0', '1', '-');
    attribute enum_encoding: string;
    attribute enum_encoding of bit_3: type is "0 1 D";
    type bit_3_vector is array (integer range <>) of bit_3;
end tipos;

use work.tipos.all;
entity cir is
    port(bcd: in bit_vector(3 downto 0);
        sete_seg: out bit_3_vector(6 downto 0));
end cir;

use work.tipos.all;
architecture conversor of cir is
begin
    process(bcd)
    begin
        case bcd is
            when "0000" => sete_seg <= "1111110";
            when "0001" => sete_seg <= "1100000";
            when "0010" => sete_seg <= "1011011";
            when "0011" => sete_seg <= "1110011";
            when "0100" => sete_seg <= "1100101";
            when "0101" => sete_seg <= "0110111";
            when "0110" => sete_seg <= "0111111";
            when "0111" => sete_seg <= "1100010";
            when "1000" => sete_seg <= "1111111";
            when "1001" => sete_seg <= "1110111";
            when others => sete_seg <= "-------";
-- a saida pode tomar
                                                -- qualquer valor
        end case;
    end process;
end conversor;
```

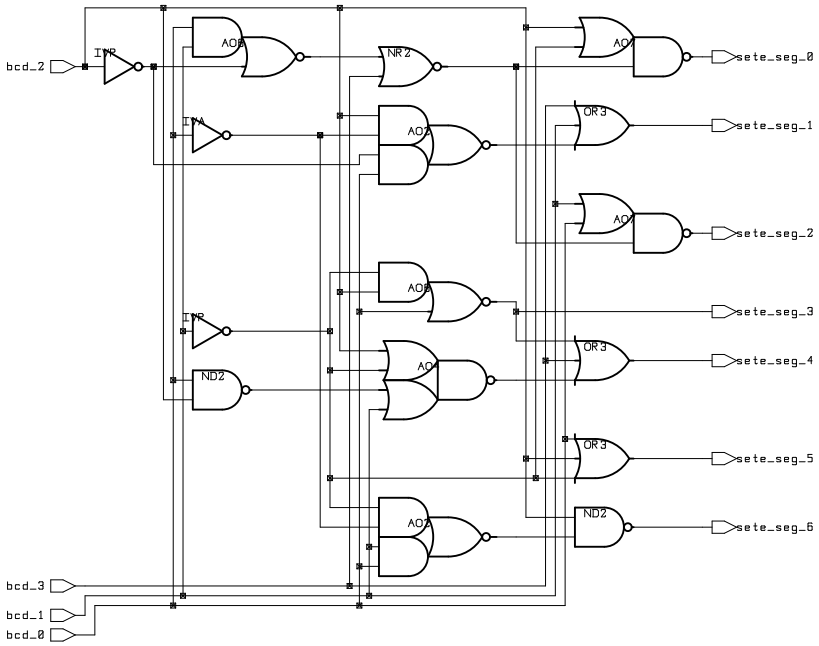


Figura 4.23: Descrição de um conversor do código BCD para um mostrador de sete segmentos usando lógica multi-valor.

ção 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[6].

No exemplo da figura 4.24 descreve-se a máquina de estados apresentada na secção anterior (máquina de Moore para detectar a sequência "`111`") mas, usando um estilo de descrição em que não foram tomadas em consideração algumas da recomendações apresentadas. O sinal de *reset* foi eliminado (no processo responsável pelo funcionamento síncrono deixou de haver qualquer referência a esta sinal), e a definição do próximo estado só é feita se for necessário mudar de estado[7] (por exemplo, no ramo da instrução `case` correspondente ao estado `zero` o `else` do `if` desapareceu, devido a ser desnecessário actualizar o sinal `proximo_estado` com o estado em que a máquina já se encontra). No circuito sintetizado, esta alteração reflectiu-se na utilização de *latches*, antes de cada *flip-flop*. Estes permitem guardar o valor do próximo estado, quando ele não é definido pela lógica combinatória a partir do estado actual. Devido ainda a não ser possível inicializar os elementos de memória, através de uma linha de *reset*, este circuito poderá não funcionar correctamente após ser ligado.

Na descrição de circuitos em VHDL o projectista pode também recorrer ao conjunto de módulos que em geral as ferramentas de síntese e/ou simulação oferecem. Nestes são definidos alguns tipos e subtipos conjuntamente com os opera-

[6]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'.

[7]Considera-se que a entrada só muda de valor uma única vez entre dois impulsos de relógio (entrada sincronizada).

```
entity vhdl_mau is
  port(entrada, clock: in bit;       -- o sinal de reset desapareceu
    sequencia: out bit);
end vhdl_mau;


architecture mal_descrita of vhdl_mau 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        -- so se altra o proximo_estado
          proximo_estado <= um;      -- se realmente houver
        end if;                      -- mudanca de estado
      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 = '0' then
          proximo_estado <= zero;
        end if;
    end case;
  end process;

  sinc: process(clock)               -- funcionamento sincrono
  begin
    if clock'event and clock = '1' then
      estado_actual <= proximo_estado;
    end if;
  end process;

end mal_descrita;
```

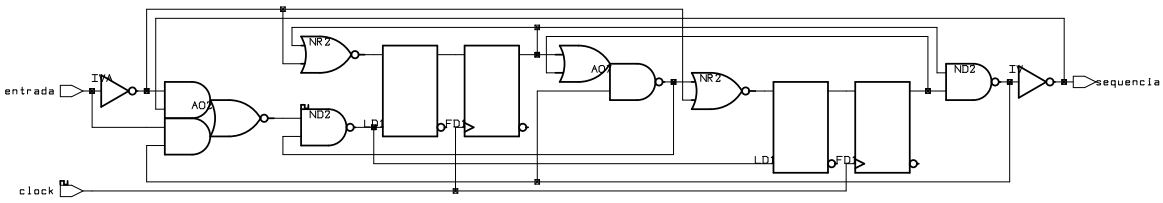


Figura 4.24: Máquina de estados "mal" sintetizada devido ao estilo de descrição usado.

dores, funções de conversão e/ou funções de resolução que permitem a sua utilização. Exemplo de alguns destes tipos definidos num módulo oferecido pelo sistema [Synthesis 91] são: `UNSIGNED, SIGNED` e `MVL7`.

A utilização destes módulos permite ao projectista ter à partida um maior conjunto de tipos e funções definidos para descrever o seu circuito com a garantia que, por estas terem sido escritas pelo fabricantes da ferramenta, aproveitam todas as suas potencialidades. Porém, muitas vezes é apenas fornecida a declaração do módulo não existindo o seu corpo em VHDL (este é escrito noutras linguagens e está directamente inserido na ferramenta), pelo que a descrição do circuito pode ficar em alguns casos dependente da ferramenta de síntese.

4.4 Interacção com a Ferramenta de Síntese

Ter uma boa descrição do circuito (com o estilo apropriado para a síntese) e uma boa ferramenta de síntese são condições necessárias, mas não suficientes para garantir a qualidade do circuito final. Esta, depende também da forma como o projectista aproveita as potencialidades da ferramenta para "conduzir" o processo de síntese. Assim, o circuito obtido a partir de uma descrição sintetizável (sem intervenção do projectista) "garante apenas" que este tem um comportamento logicamente correcto de acordo com a descrição de VHDL fornecida. Porém, e dependendo do tipo de circuito e das bibliotecas usadas, podem surgir problemas a nível temporal, detectados apenas quando o circuito for simulado logicamente ou, em alguns casos, numa fase mais avançada do projecto (na simulação lógica efectuada após a colocação e encaminhamento do circuito).

Para ilustrar a influência que o projectista pode ter no circuito final considere-se a descrição apresentada na figura 4.25. Esta, descreve uma função aritmética que calcula em cada ciclo de relógio a soma da entrada `c` com a média que as entradas `a` e `b` tinham no ciclo de relógio anterior.

Nas figuras 4.26 e 4.27 apresentam-se dois circuitos sintetizados a partir da descrição anterior. O primeiro foi sintetizado sem qualquer intervenção por parte do projectista. O segundo foi sintetizado indicando previamente à ferramenta de síntese um conjunto de valores que caracterizam o "ambiente" onde o circuito irá funcionar (*fan-out* máximo de algumas entradas, cargas nas saídas e atraso na chegada de alguns sinais de entrada) e as restrições que devem ser tomadas em conta durante a síntese (atraso máximo nas saídas e área máxima). Apesar de ambos

```
entity cir is
    port(ck, rst: in bit;
         a, b, c: in integer range 0 to 3;
         soma: out integer range 0 to 6;
         media: buffer integer range 0 to 3);
end cir;

architecture exemp of cir is
    signal temp: integer range 0 to 3;
begin
    process(a, b, c, media)
    begin
        temp <= (a + b)/2;
        soma <= c + media;
    end process;

    process(ck, rst)
    begin
        if rst = '0' then
            media <= 0;
        elsif ck'event and ck = '1' then
            media <= temp;
        end if;
    end process;
end exemp;
```

Figura 4.25: Descrição de uma função aritmética.

os circuitos apresentarem a mesma funcionalidade, têm um comportamento temporal bastante diferente. Por exemplo, o caminho crítico, que limita a frequência máxima de funcionamento do circuito (representado a cheio nas figuras), foi alterado e o seu atraso passou de 6.36 para 5.73 devido às condições mais realistas em que o segundo circuito foi sintetizado.

Assim, antes de realizar a síntese de um circuito o projectista pode (e deve) interactuar com a ferramenta sob duas perspectivas: a de caracterizar o circuito e a de impôr restrições.

A caracterização consiste em fornecer à ferramenta informação adicional sobre o ambiente onde o circuito ou subcircuito vai ser utilizado. Informação esta que está de alguma forma relacionada com as características temporais ou eléctricas do circuito, não podendo por isso ser expressas numa descrição VHDL sintetizável. As principais formas de caracterizar um circuito são apresentadas na tabela 4.4, onde se descrevem as características temporais de tempos de chegada e as características eléctricas de "resistência", cargas capacitivas e *fan-out*.

A restrição de um circuito consiste em fornecer à ferramenta um conjunto de parâmetros que serão interpretados como objectivos a serem atingidos na síntese

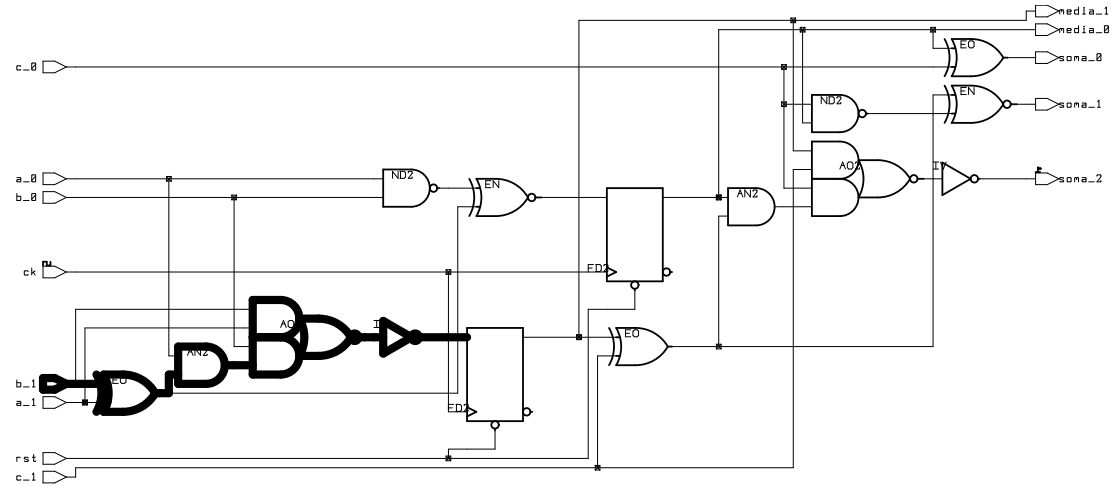


Figura 4.26: Circuito sintetizado sem intervenção do projectista.

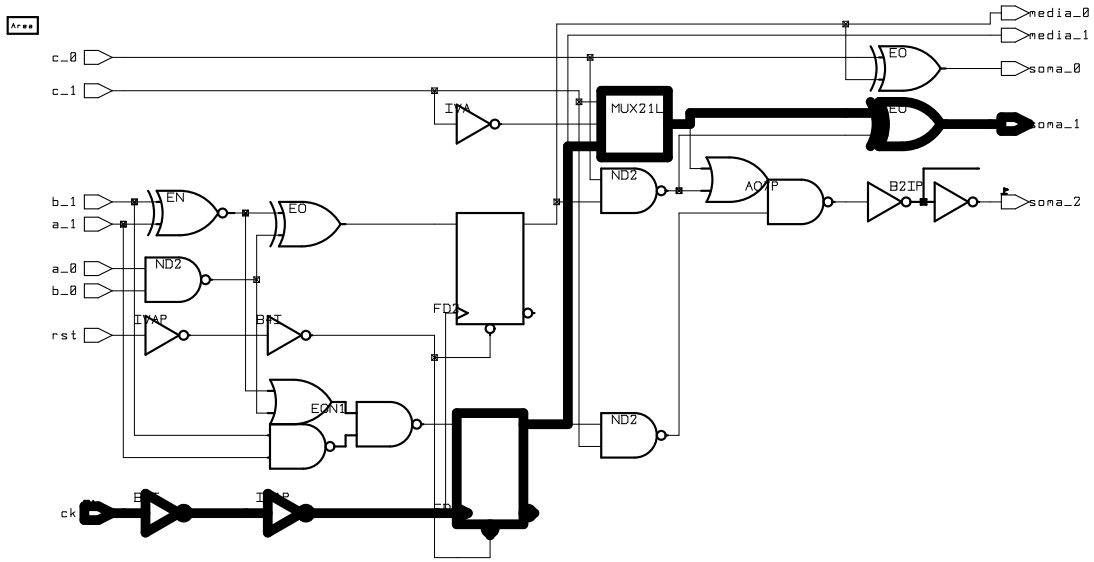


Figura 4.27: Circuito sintetizado com a intervenção do projectista.

do circuito. Na tabela 4.5 são apresentadas as restrições mais comuns que podem ser aplicadas a um circuito, nomeadamente, restrições de área, tempos de atraso, ciclo de relógio e *fan-out*.

Característica	Descrição
Tempos de chegada	Indica a desfasagem temporal entre as várias entradas do circuito.
"Resistência"	Indica a capacidade das entradas forçarem um valor expresso em unidades de tempo por unidades de carga.
Cargas capacitivas	Indica a carga, expressa em unidades de carga, que as entradas e/ou saídas podem ter.
Fan-out	Indica o *fan-out* esperado nas saídas.

Tabela 4.4: Principais formas de caracterizar um circuito.

Restrição	Descrição
Área	Limita a área máxima que o circuito deve ocupar.
Tempos de atraso	Limita o tempo máximo de propagação dos sinais para as saídas.
Ciclo de relógio	Determina as características do(s) relógio(s) do circuito (por exemplo, período, *duty-cycle*, etc)
Fan-out	Limita o *fan-out* máximo que cada entrada pode ter.

Tabela 4.5: Principais formas de restringir um circuito.

As restrições impostas podem ser ou não atingidas pela ferramenta, pelo que devem ser verificadas após a síntese do circuito. Em qualquer dos casos, o projectista pode realizar novas iterações de síntese para explorar o espaço de projecto de um dado circuito. Este espaço é constituído pelo conjunto de todas as soluções de compromisso existentes entre a área ocupada pelo circuito e o seu maior tempo de propagação (que limita a velocidade máxima de funcionamento). Na tabela 4.6 apresentam-se algumas soluções obtidas para o circuito descrito na figura 4.25 (caracterizado com as condições realistas). Observando esta tabela pode-se concluir que várias arquitecturas podem ser exploradas até ser encontrada aquela que melhor satisfaça o projectista.

Área	Atraso máximo
47	6.38
55	5.32
62	5.00

Tabela 4.6: Diferentes compromissos entre a área ocupada e o atraso máximo

4.5 Conclusões

Neste capítulo descreveu-se o fluxo de projecto usado na síntese de um circuito digital a partir de uma descrição VHDL. Devido à complexidade que esta linguagem apresenta, foi descrito do ponto de vista da síntese qual o subconjunto de instruções que são suportados pelas actuais ferramentas. Foram ainda apresentados exemplos de como as descrições de circuitos em VHDL são sintetizadas, assim como, a influência que o estilo de descrição e a interacção com a ferramenta de síntese, pode ter na qualidade final do circuito. Assim, a utilização da linguagem VHDL num ambiente de síntese deve ter em conta o subconjunto de instruções sintetizáveis e recorrer a um estilo de descrição apropriado para a síntese, cujas principais características foram apresentadas neste capítulo.

No próximo capítulo é descrita uma biblioteca de modelos, destinada a auxiliar o projectista a obter uma descrição sintetizável do seu circuito.

Capítulo 5

Uma Biblioteca de Modelos - Blocos Fundamentais

A biblioteca de modelos (introduzida na secção 4.1) destina-se a auxiliar o projectista a obter uma descrição funcionalmente correcta e sintetizável do seu circuito. Esta biblioteca pode conter descrições de circuitos de dois tipos: sintetizáveis e não sintetizáveis. Enquanto que as primeiras podem ser utilizadas para a descrição de circuitos mais complexos (destinados as serem sintetizados), as segundas permitem verificar a funcionalidade dessas descrições (e eventulamente do circuito sintetizado) de uma forma mais simples. Por exemplo, o projecto de um circuito de interface a um microprocessador, pode ser mais facilmente realizado se houver uma descrição precisa, em VHDL, desse mesmo microprocessador.

Devido à descrição/modelação de circuitos não sintetizáveis estar fora do âmbito deste trabalho, a biblioteca de modelos apresentada neste capítulo é constituída apenas por blocos de *hardware* descritos em VHDL sintetizável.

As descrições apresentadas referem-se a circuitos que utilizam palavras de 8 *bits*, podendo ser facilmente adaptadas para palavras de dados de outros comprimentos. Os tipos de dados utilizados são sempre que possível os "pré-definidos" na linguagem (`boolean, bit, bit_vector` e `integer`), excepto nas situações seguintes em que se torna conveniente recorrer a outros tipos de dados:

- Quando é necessário utilizar lógica multi-valor para descrever o circuito. Neste caso, foram utilizados os tipos de dados definidos pela norma IEEE 1164 que se encontram declarados no módulo `mvl9` (apresentado no

apêndice A).

- Quando na descrição do circuito é necessário realizar operações aritméticas ou lógicas com vectores. Neste caso, recorreu-se ao módulo `arithmetic` fornecido pela ferramenta de síntese [Synthesis 91] (apresentado no apêndice B).

Note-se que a utilização dos módulos acima referidos não tornam as descrições que os usem dependentes da ferramenta de síntese, pois estes encontram-se totalmente descritos em VHDL, declaração e corpo.

5.1 Unidades Aritméticas e Lógicas

A realização de operações aritméticas e lógicas pode ser descrita utilizando tipos inteiros. Uma vez que a representação de um inteiro em binário pode ser feita da várias formas (por exemplo, codificação natural, complemento para um, complemento para dois), que podem implicar diferentes implementações dos operadores aritméticas e lógicos, é necessário saber qual a forma utilizada pela ferramenta de síntese.

No caso da ferramenta de síntese [Synthesis 91] é utilizada a codificação natural para variáveis que apenas representam número inteiros positivos e a codificação em complemento para dois para variáveis que podem representar números negativos.

A forma mais simples de descrever um somador de oito *bits* encontra-se representada na figura 5.1.

A descrição de um somador com entrada e saída de transporte é apresentada na figura 5.2. Nesta descrição representam-se os operandos e a soma por um vector de oito *bits* utilizando a codificação natural. Para isso utilizou-se o módulo `arithmetic` onde está definido o tipo `UNSIGNED` e redefinido o operador `'+'` para operandos deste tipo.

A descrição de um comparador de dois inteiros com saídas que indicam quando uma entrada é menor, igual ou maior que outra, é apresentada na figura 5.3. Nesta descrição foram utilizados os operadores '<' e '=' definidos na linguagem que requerem operandos inteiros e produzem resultados do tipo booleano. O operador '>' não foi utilizado pois implicaria a utilização de um recurso desnecessário. Isto porque dois números são exclusivamente menores, iguais ou maiores.

```
entity somador is
    port(a, b: in integer range 0 to 255;
         soma: out integer range 0 to 510);
end somador;

architecture com_inteiros of somador is
begin
    soma <= a + b;
end com_inteiros;
```

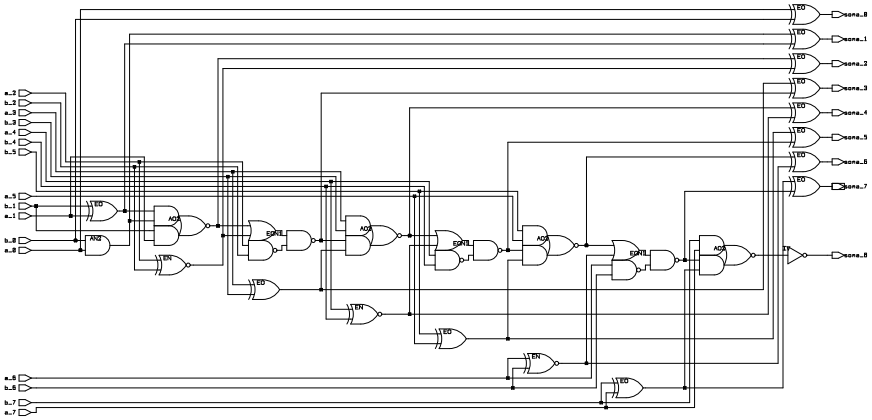


Figura 5.1: Descrição de um somador utilizando inteiros.

operacao	resul
00	A + B
01	A - B
10	-A + B
11	-A - B

Tabela 5.1: Operações realizadas pela unidade aritmética `u_logica`.

Na figura 5.4 é descrita uma unidade aritmética que realiza as operações apresentadas na tabela 5.1. A representação numérica dos operadores e da soma é feita utilizando a codificação em complemento para dois através do tipo de dados `SIGNED` definido no módulo `arithmetic`. Para partilhar o maior número de recursos do circuito, este foi descrito de forma a ser apenas necessário um único somador e dois operadores de negação. Estes operadores encontra-se também redefinidos no módulo `arithmetic` juntamente com a função `CONV_SIGNED` que permite a extensão do número de *bits* dos operandos para representar correctamente o resultado da operação escolhida.

```
use work.arithmetic.all;
entity somador_trans is
    port(trans_ent: in bit;
         a, b: in UNSIGNED(7 downto 0);
         soma_par: out UNSIGNED(7 downto 0);
         trans_sai: out bit);
end somador_trans;

use work.arithmetic.all;
architecture com_vectores of somador_trans is
begin
    process(trans_ent, a, b)
        variable temp: UNSIGNED(9 downto 0);
    begin
        temp := ('0' & a & trans_ent) + ('0' & b & '1');
        soma_par <= temp(8 downto 1);
        trans_sai <= temp(9);
    end process ;
end com_vectores;
```

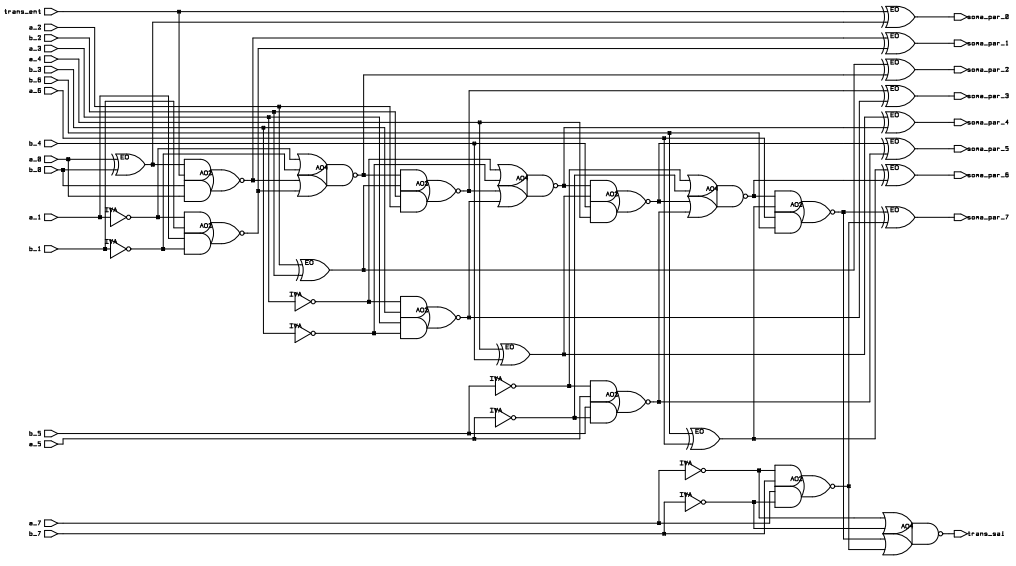


Figura 5.2: Descrição de um somador utilizando vectores de *bits*.

```
entity comparador is
    port(a, b: in integer range 0 to 255;
         menor, igual, maior: buffer boolean);
end comparador;

architecture comp of comparador is
begin
    menor <= (a < b);
    igual <= (a = b);
    maior <= not (menor or igual);
end comp;
```

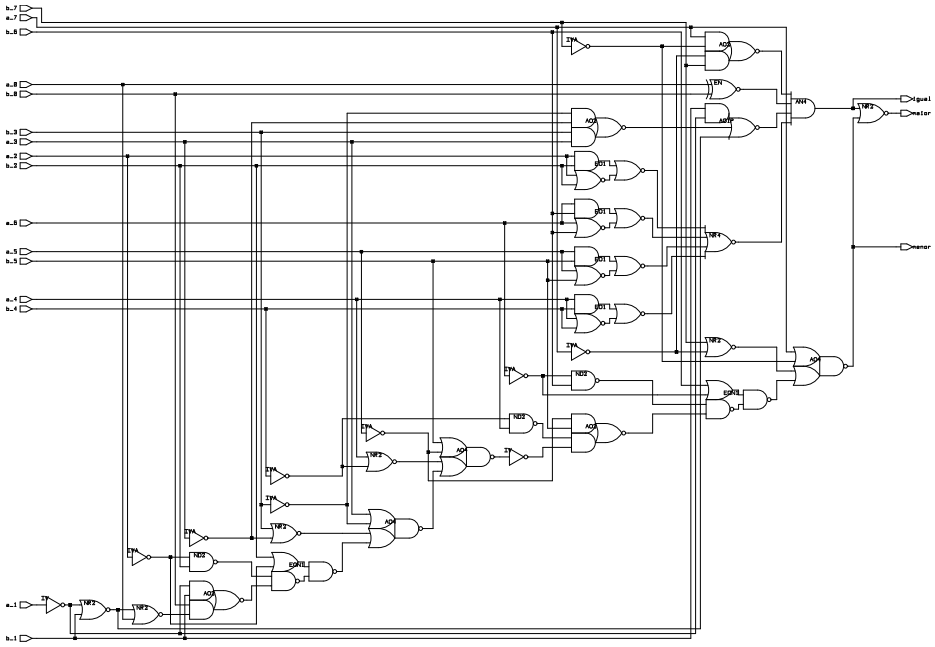


Figura 5.3: Descrição de um comparador de dois inteiros positivos.

```
use work.arithmetic.all;
entity u_logica is
    port(operacao: bit_vector(1 downto 0);
         a, b: SIGNED(7 downto 0);
         resul: out SIGNED(8 downto 0));
end u_logica;

use work.arithmetic.all;
architecture com_int of u_logica is
begin
    process(operacao, a, b)
        variable oper1, oper2: SIGNED(8 downto 0);
    begin
        oper1 := CONV_SIGNED(a, 9);
        oper2 := CONV_SIGNED(b, 9);
        if operacao(1) = '1' then
            oper1 := -oper1;
        end if;
        if operacao(0) = '1' then
            oper2 := -oper2;
        end if;
        resul <= oper1 + oper2;
    end process;
end com_int;
```

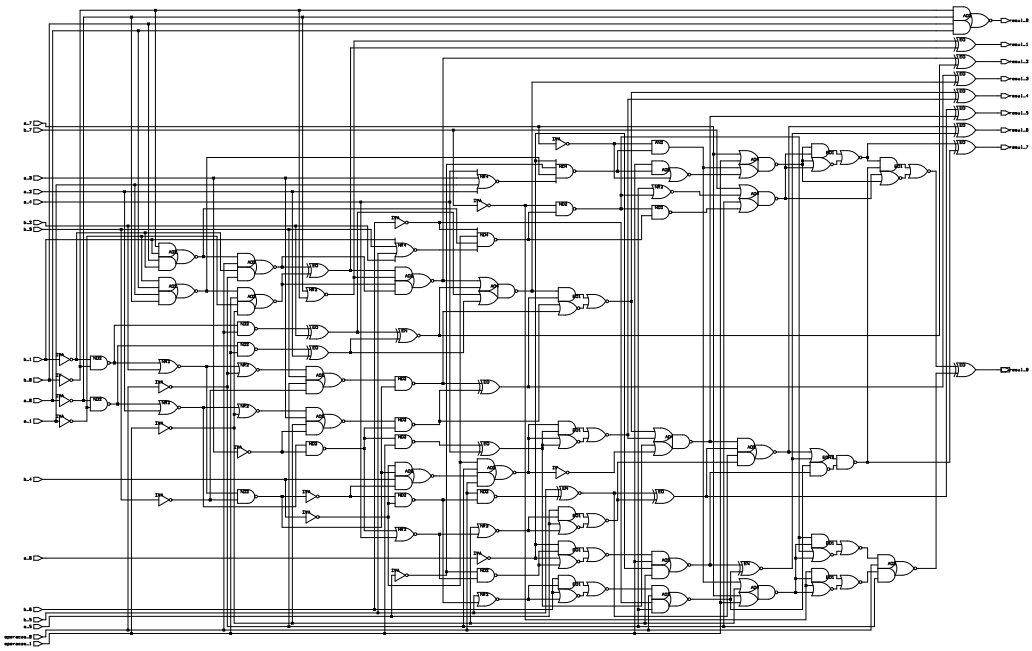


Figura 5.4: Descrição da unidade aritmética `u_logica`.

5.2 Registos

A descrição de um registo de oito *bits* com *reset* assíncrono e sinal de carregamento síncrono é apresentado na figura 5.5. Neste, sempre que o sinal `carrega` estiver activo é lido para o registo, no flanco ascendente do relógio, o valor presente na entrada `ent`. Se o sinal `carrega` estiver a '0' o valor presente no registo não é alterado.

Na figura 5.6 apresenta-se a descrição de um registo multi-função de oito *bits* com *reset* assíncrono. Neste registo, é possível através das entradas `desloca` e `carrega_dir` seleccionar quatro tipos de funções síncronas que podem ser realizadas. Estas funções encontram-se descritas na tabela 5.2.

`desloca`	`carrega_dir`	Função
0	0	mantém o valor do registo
0	1	carregamento paralelo do registo
1	0	deslocamento para a esquerda (carregamento paralelo do *bit* mais à direita)
1	1	deslocamento para a direita (carregamento paralelo do *bit* mais à esquerda)

Tabela 5.2: Operações realizadas pelo registo multi-função.

5.3 Descodificadores e Codificadores

Na figura 5.7 descreve-se um descodificador de três para oito *bits* com sinal de activo. Neste descodificador, cujo funcionamento se resume na tabela 5.3, a saída apresenta um único *bit* a um (cujo peso corresponde ao valor da entrada `a`) se a entrada `activa` estiver ao valor lógico `'1'`. Se esta entrada estiver ao valor lógico `'0'`, a saída apresentará o valor zero independentemente da entrada `a`.

O descodificador descrito na figura 5.8 tem uma estrutura semelhante ao anterior mas a codificação apresentada na saída `b` é diferente e encontra-se descrita na tabela 5.4.

```
entity registo is
     port(clk, rst, carrega: in bit;
          ent: in bit_vector(7 downto 0);
          sai: buffer bit_vector(7 downto 0));
end registo;

architecture reg of registo is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            sai <= (others => '0');
        elsif clk'event and clk = '1' then
            if carrega = '1' then
                sai <= ent;
            else
                sai <= sai;
            end if;
        end if;
    end process;
end reg;
```

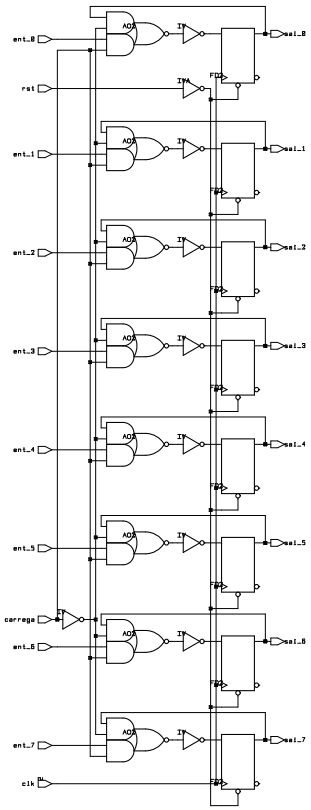


Figura 5.5: Registo de oito *bits* com *reset* assíncrono.

```
entity reg_des is
     port(clk, rst, carrega_dir, desloca: in bit;
          ent: in bit_vector(7 downto 0);
          sai: out bit_vector(7 downto 0));
end reg_des;

architecture reg_des_par of reg_des is
    signal reg: bit_vector(7 downto 0);
begin

    sai <= reg;

    process(clk, rst)
        variable oper: bit_vector(1 downto 0);
    begin
        if rst = '1' then
            reg <= (others => '0');
        elsif clk'event and clk = '1' then
            oper := desloca & carrega_dir;
            case oper is
                when "00" => reg <= reg;
                when "01" => reg <= ent;
                when "10" => reg <= reg(6 downto 0) & ent(0);
                when "11" => reg <= ent(7) & reg(7 downto 1);
            end case;
        end if;
    end process;
end reg_des_par;
```

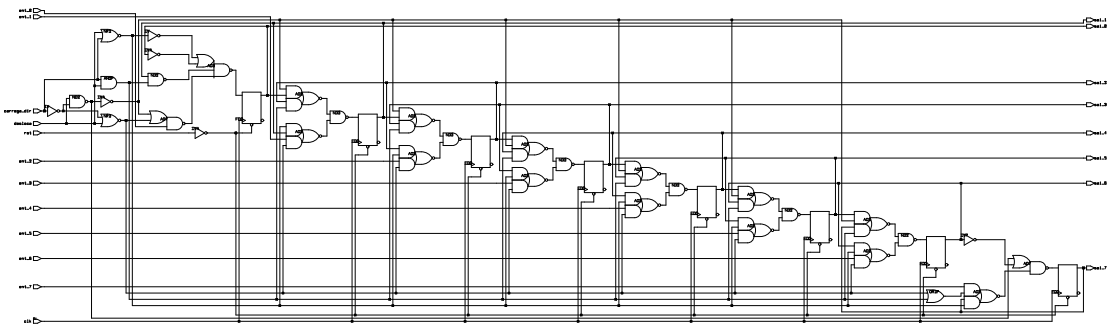


Figura 5.6: Descrição de um registo multi-função (com deslocamento e carregamento paralelo).

A tabela 5.5 representa o comportamento de um codificador de oito para três *bits* com prioridade. Isto é, se várias entradas estiverem com o valor lógico `'1'`, será apenas codificada aquela que se apresenta na linha de entrada de maior peso. A descrição deste codificador encontra-se na figura 5.9. Devido a haver combinações de entradas para as quais a saída `b` pode tomar qualquer valor lógico (correspondentes às duas primeiras linhas da tabela), devem ser usados tipos de dados que contenham uma representação do "valor lógico" indiferença. Desta forma, para

activo	a	b_7 . . . b_0
0	x	00000000
1	0	00000001
1	1	00000010
1	2	00000100
1	3	00001000
1	4	00010000
1	5	00100000
1	6	01000000
1	7	10000000

Nota: "x" significa que a entrada pode ter qualquer valor lógico.

Tabela 5.3: Funcionamento do descodificador.

activo	a	b_7 . . . b_0
0	x	00000000
1	0	00000001
1	1	00000011
1	2	00000111
1	3	00001111
1	4	00011111
1	5	00111111
1	6	01111111
1	7	11111111

Nota: "x" significa que a entrada pode ter qualquer valor lógico.

Tabela 5.4: Funcionamento do descodificador com "memória".

além de o circuito ser correctamente descrito, a ferramenta de síntese pode tirar partido desta informação na fase de optimização lógica do circuito. Assim, a saída `b` foi declarada como um vector de *bits* do tipo `std_ulogic`, tipo este que se encontra definido no módulo `mvl9`.

```
entity descodificador is
    port(a: in integer range 0 to 7;
         activo: in bit;
         b: out bit_vector(7 downto 0));
end descodificador;

architecture desc of descodificador is
begin
    process(a, activo)
    begin
        b <= (others => '0');
        if activo = '1' then
            b(a) <= '1';
        end if;
    end process ;
end desc;
```

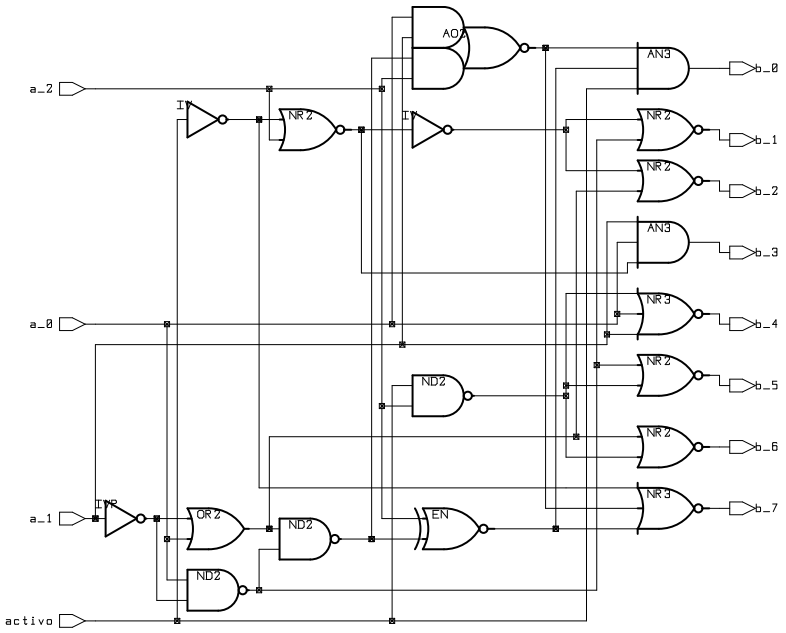


Figura 5.7: Descrição de um descodificador de oito *bits*.

5.4 Multiplexers

Nas figuras 5.10, 5.11 e 5.12 descrevem-se respectivamente, um multiplexer de 8 para 1 de um *bit*, um desmultiplexer de 1 para 8 e um multiplexer de 2 para 1 de oito *bits*. As duas primeiras descrições são feitas à custa da indexação dos vectores de entrada ou saída pelas linhas de selecção dos multiplexers, enquanto que a terceira recorre a uma atribuição concorrente seleccionada pela variável `sel`.

```
entity desc_nivel is
    port(a: in integer range 0 to 7;
         activo: in bit;
         b: out bit_vector(7 downto 0));
end desc_nivel;

architecture desc of desc_nivel is
begin
    process(a, activo)
    begin
        b <= (others => '0');
        if activo = '1' then
            for i in b'range loop
                if i <= a then
                    b(i) <= '1';
                end if;
            end loop;
        end if;
    end process ;
end desc;
```

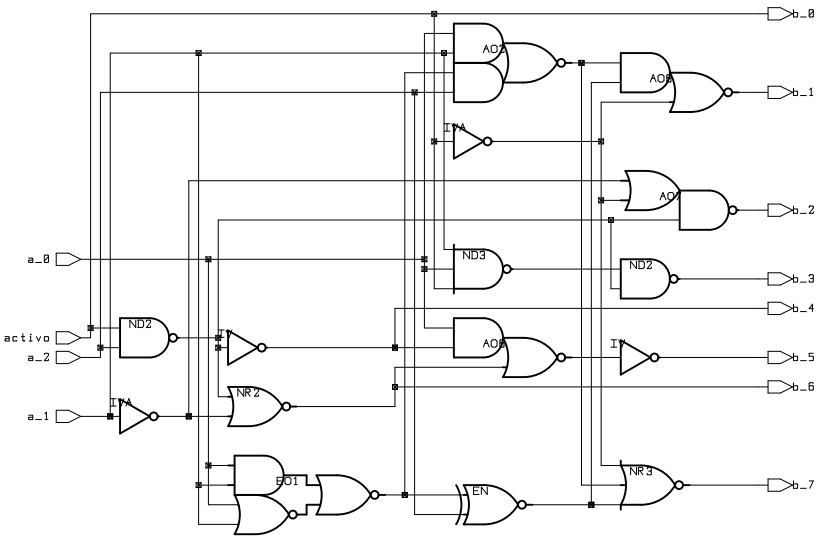


Figura 5.8: Descrição do descodificador de oito *bits* com "memória".

5.5 Contadores

Na figura 5.13 descreve-se um contador de oito *bits* (módulo de contagem de 256) com *reset* assíncrono e entrada para carregamento paralelo síncrono.

Na figura 5.14 é descrito um contador de oito *bits* com *reset* assíncrono em que o valor máximo de contagem é fixado externamente pela entrada `cont_max`.

```
use work.mvl9.all;
entity codificador is
    port(a: in integer range 0 to 255;
         activo_ent: in bit;
         activo_sai: out bit;
         b: out std_ulogic_vector(2 downto 0));
end codificador;

use work.mvl9.all;
architecture com_proioridades of codificador is
begin
    process(a, activo_ent)
    begin
        activo_sai <= '0';
        b <= (others => '-');
        if activo_ent = '1' then
            activo_sai <= '1';
            case a is
                when 255 downto 128 => b <= "111";
                when 127 downto  64 => b <= "110";
                when  63 downto  32 => b <= "101";
                when  31 downto  16 => b <= "100";
                when  15 downto   8 => b <= "011";
                when   7 downto   4 => b <= "010";
                when   3 downto   2 => b <= "001";
                when   1            => b <= "000";
                when   0            => b <= "---";
                                       activo_sai <= '0';
                end case;
        end if;
    end process;
end com_proioridades;
```

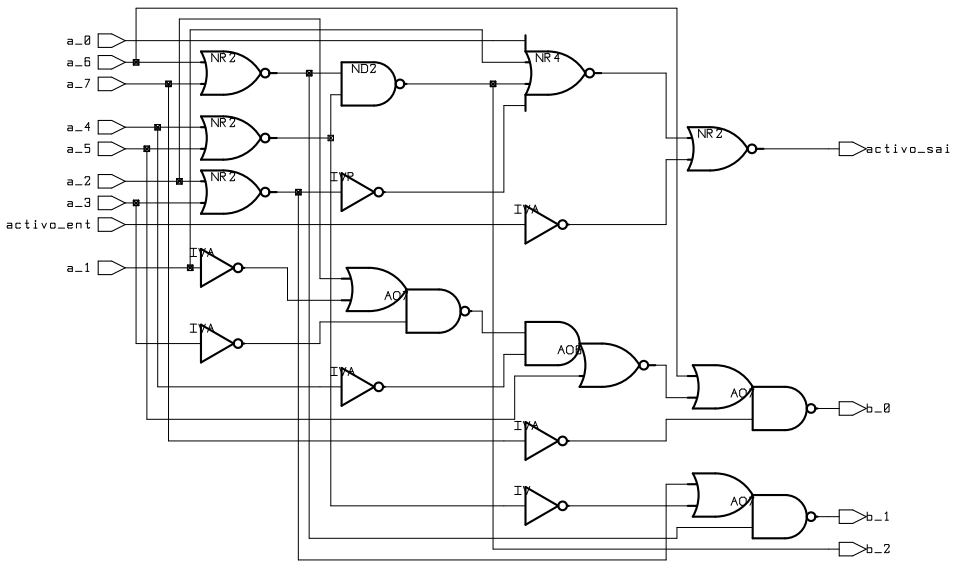


Figura 5.9: Descrição do codificador de oito para três *bits* com prioridade.

activo_ent	a_7 . . . a_0	activo_sai	b_3 . . . b_0
0	xxxxxxx	0	- - -
1	0000000	0	- - -
1	0000001	1	001
1	000001x	1	010
1	00001xx	1	011
1	0001xxx	1	100
1	001xxxx	1	101
1	01xxxxx	1	110
1	1xxxxxx	1	111

Nota: "x" significa que a entrada pode ter qualquer valor lógico.
Nota: "-" significa que a saída pode ter qualquer valor lógico.

Tabela 5.5: Funcionamento de codificador de oito para três *bits* com prioridade.

```
entity mux81_1 is
    port(sel: in integer range 0 to 7;
         a: in bit_vector(7 downto 0);
         b: out bit);
end mux81_1;

architecture multiplexer of mux81_1 is
begin
    b <= a(sel);
end multiplexer;
```

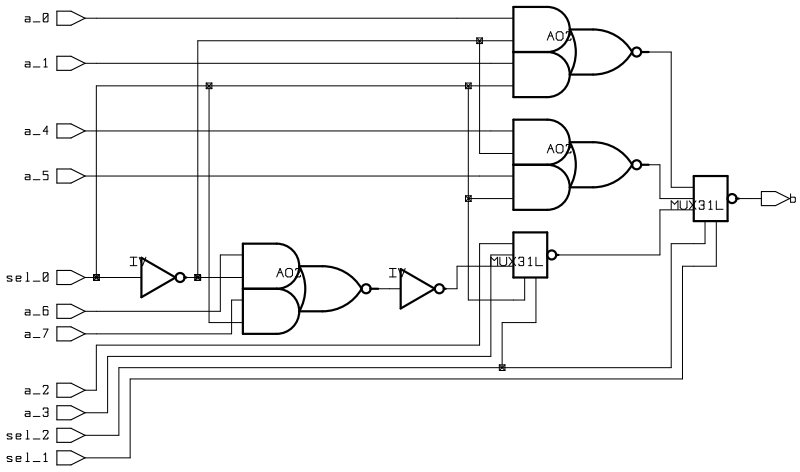


Figura 5.10: Descrição de um multiplexer de 8 para 1 de um *bit*.

5.6 Blocos com Alta Impedância

A descrição de elementos com alta impedância é feita recorrendo à lógica multi-valor representada no módulo `mvl9`, através do tipo de dados "base" `std_ulogic`.

```
entity demux18 is
    port(sel: in integer range 0 to 7;
         a: in bit;
         b: out bit_vector(7 downto 0));
end demux18;

architecture multiplexer of demux18 is
begin
    process(a, sel)
    begin
        b <= (others => '0');
        b(sel) <= a;
    end process;
end multiplexer;
```

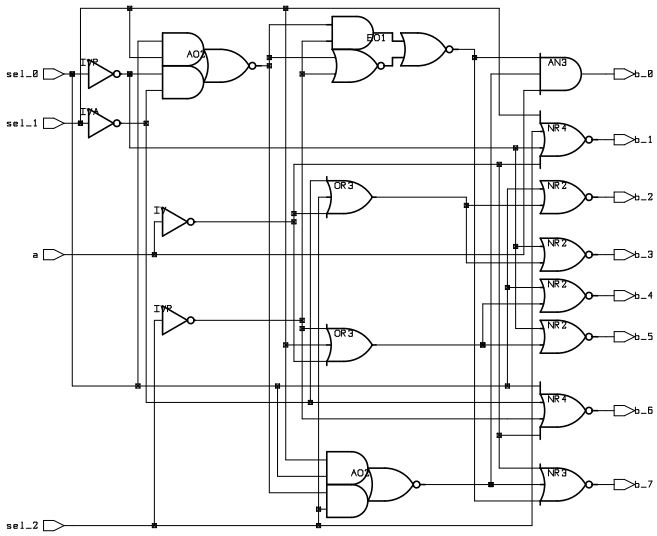


Figura 5.11: Descrição de um desmultiplexer de 1 para 8.

Na figura 5.15 é descrito um bloco de *buffers* para uma linha de dados de oito *bits* com a possibilidade de colocação das saídas em alta impedância.

A descrição de um *buffer* de oito *bits* bi-direccional é apresentada na figura 5.16. Devido às linhas de dados `a` e `b` poderem ter diferentes *drivers* (um por cada sinal dentro desta entidade e outros externos), o tipo de dados destas linhas tem que ser um tipo que tenha associado um função de resolução. Conforme referido no capítulo 3, esta função é responsável por determinar, a partir dos valores forçados pelos diferentes *drivers*, qual o valor que na realidade o sinal vai tomar. No caso do módulo `mvl9` o tipo de dados que tem associado uma função de resolução é designado por `std_logic`. Os vários modos de funcionamento do circuito descrito na figura 5.16 estão representados na tabela 5.6.

A descrição de um circuito que permite passar de um *bus* bi-direccional (`a`)

```
entity mux21_8 is
    port(sel: in bit;
         a, b: in bit_vector(7 downto 0);
         c: out bit_vector(7 downto 0));
end mux21_8;

architecture multiplexer of mux21_8 is
begin
     with sel select
         c <= a when '0',
              b when '1';
end multiplexer;
```

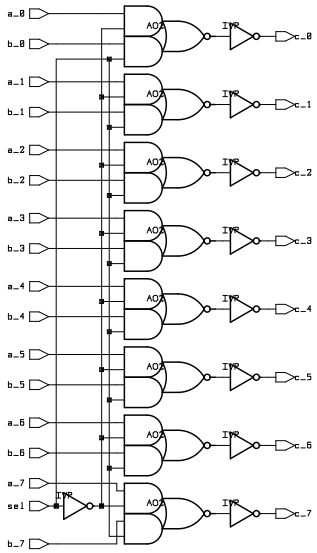


Figura 5.12: Descrição de um multiplexer de 2 para 1 de oito *bits*.

activo	dir	Função
0	x	a e b em alta impedância
1	0	a → b
1	1	b → a

Nota: "x" significa que a entrada pode ter qualquer valor lógico.

Tabela 5.6: Funcionamento do *buffer* bi-direccional.

para dois *buses* separados, um de leitura (sai) e outro de escrita (ent), é apresentado na figura 5.17. Por existirem neste circuito sinais de diferentes tipos (std_ulogic_vector para sinais com um único *driver* e std_logic_vector para sinais com múltiplos *drivers*) têm que ser usadas funções de conversão de tipos

```
entity contador is
    port(clk, rst, carrega: in bit;
         ent: in integer range 0 to 255;
         sai: buffer integer range 0 to 255);
end contador;

architecture comp of contador is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            sai <= 255;
        elsif clk'event and clk = '1' then
            if carrega = '1' then
                sai <= ent;
            elsif sai = 255 then
                sai <= 0;
            else
                sai <= sai + 1;
            end if;
        end if;
    end process;
end comp;
```

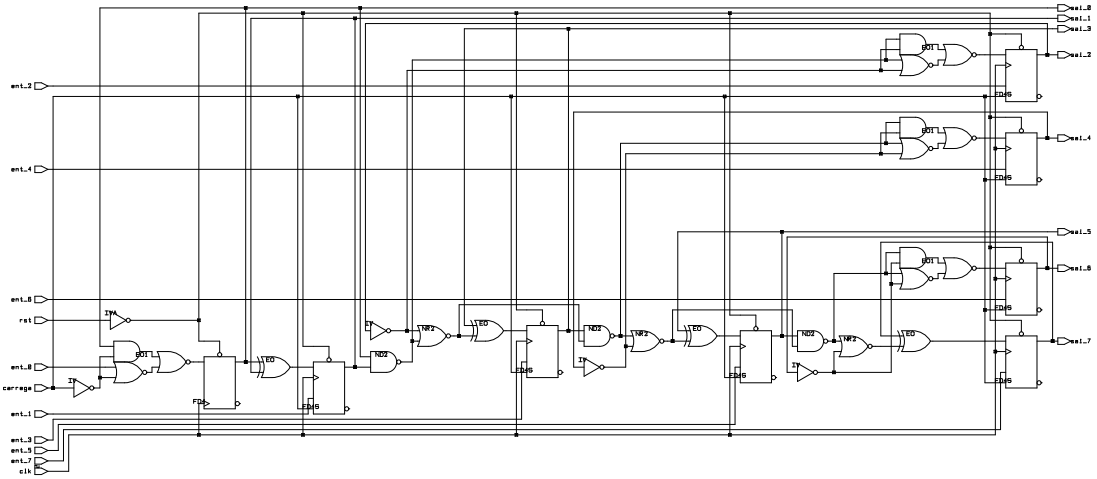


Figura 5.13: Descrição de um contador com carregamento paralelo.

nas atribuição entre estes sinais. Estas funções encontram-se descritas conjuntamente com os tipos de dados referidos no módulo `mvl9`. A tabela 5.7 descreve como devem ser actuados os sinais de controlo para correcto funcionamento deste circuito.

```
entity cont_prog is
    port(clk, rst: in bit;
         cont_max: in integer range 0 to 255;
         sai: buffer integer range 0 to 255);
end cont_prog;

architecture comp of cont_prog is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            sai <= 255;
        elsif clk'event and clk = '1' then
            if sai = cont_max then
                sai <= 0;
            else
                sai <= sai + 1;
            end if;
        end if;
    end process;
end comp;
```

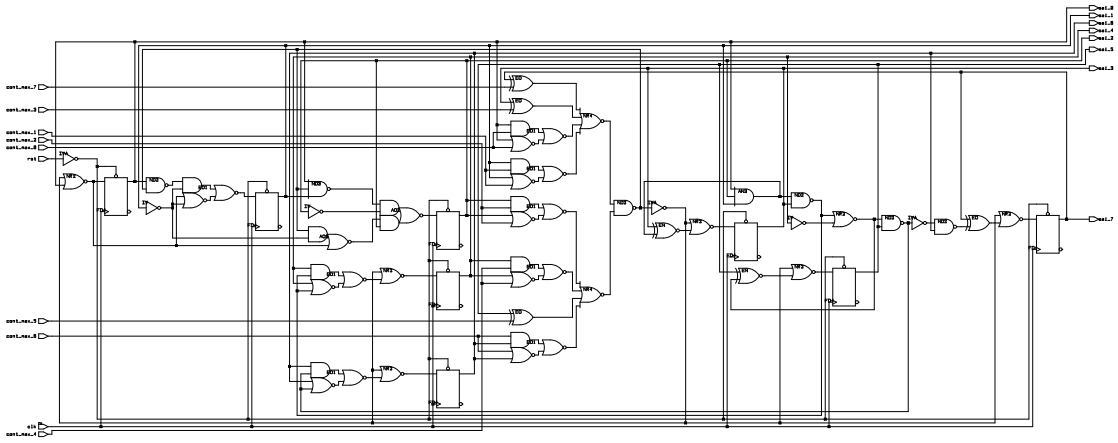


Figura 5.14: Descrição de um contador de módulo de contagem "programável".

ctrl_ent	ctrl_sai	Função
0	0	a e sai em alta impedância
0	1	a → sai
1	0	ent → a
1	1	ent → a , a → sai

Tabela 5.7: Funcionamento do *buffer* de interface a um *bus* bi-direccional

5.7 Conclusões

Neste capítulo apresentou-se a descrição de um conjunto de blocos de *hardware* (unidades lógicas e aritméticas, registos, codificadores, multiplexers, contadores,

```
use work.mvl9.all;
entity buf3state is
     port(a: in std_ulogic_vector(7 downto 0);
          activo: in bit;
          b: out std_ulogic_vector(7 downto 0));
end buf3state;

architecture buf_3 of buf3state is
begin
    process(a, activo)
    begin
        if activo = '0' then
            b <= (others => 'Z');
        else
            b <= a;
        end if;
    end process;
end buf_3;
```

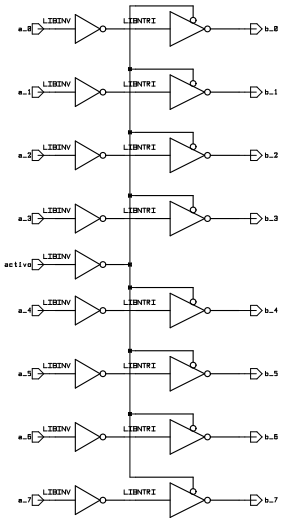


Figura 5.15: Descrição de um bloco de *buffers* com saídas em alta impedância.

etc) que constituem os elementos básicos de uma biblioteca de modelos.

Devido a estes modelos terem sido escritos recorrendo apenas ao subconjunto de instruções sintetizáveis, a sua utilização na descrição de um circuito ao nível RTL facilita a obtenção de uma representação sintetizável deste. Assim, a utilização de uma biblioteca de modelos como a apresentada neste capítulo permite reduzir o tempo de desenvolvimento de um circuito num ambiente de síntese.

No próximo capítulo exemplifica-se o projecto de um circuito digital num ambiente de síntese.

```
use work.mvl9.all;
entity buf_bidir is
     port(a, b: inout std_logic_vector(7 downto 0);
          activo, dir: in bit);
end buf_bidir;

architecture buf_2dir of buf_bidir is
begin

    process(a, activo, dir)
    begin
        if activo = '1' and dir = '0' then
            b <= a;
        else
            b <= (others => 'Z');
        end if;
    end process;

    process(b, activo, dir)
    begin
        if activo = '1' and dir = '1' then
            a <= b;
        else
            a <= (others => 'Z');
        end if;
    end process;

end buf_2dir;
```

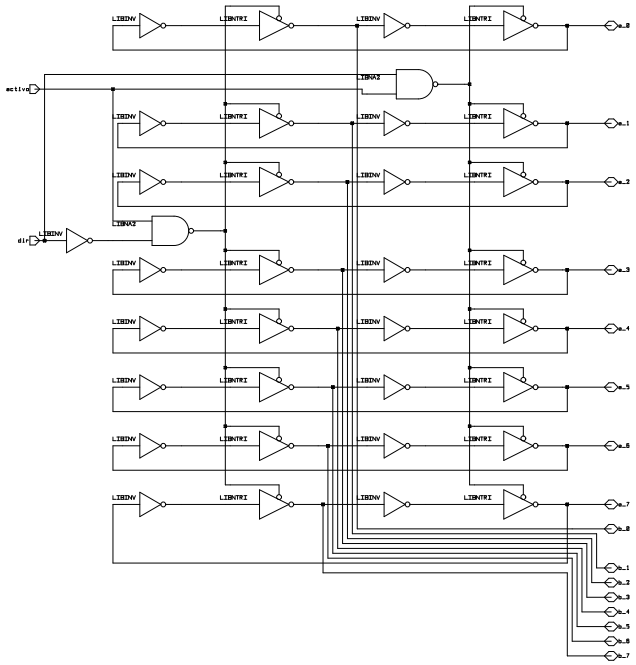


Figura 5.16: Descrição de um *buffer* de oito *bits* bi-direccional.

```
use work.mvl9.all;
entity buf_sep is
     port(a : inout std_logic_vector(7 downto 0);
          ent: in std_ulogic_vector(7 downto 0);
          sai: out std_ulogic_vector(7 downto 0);
          ctrl_ent, ctrl_sai: in bit);
end buf_sep;

architecture buf_2flux of buf_sep is
begin

    process(a, ctrl_sai)
    begin
        if ctrl_sai = '1' then
            sai <= to_stdulogicvector(a);
        else
            sai <= (others => 'Z');
        end if;
    end process;

    process(ent, ctrl_ent)
    begin
        if ctrl_ent = '1' then
            a <= to_stdlogicvector(ent);
        else
            a <= (others => 'Z');
        end if;
    end process;

end buf_2flux;
```

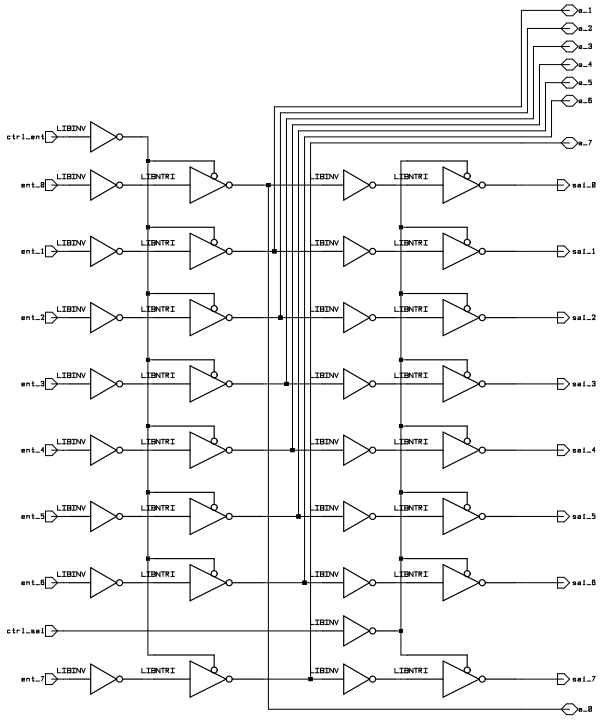


Figura 5.17: Descrição de um *buffer* de oito *bits* de interface a um *bus* bi-direccional.

Capítulo 6

Projecto de um Circuito de Criptografia Usando VHDL

Neste capítulo apresenta-se o projecto de um circuito digital usando o fluxo de projecto apresentado na secção 4.1. Partindo da especificação textual do comportamento pretendido para o circuito, é obtida uma descrição sintetizável deste (atendendo às recomendações apresentadas no capítulo 4) que posteriormente é sintetizada para obter a implementação do circuito ao nível lógico.

6.1 Objectivo do Circuito

O circuito a implementar destina-se a criptografar uma linha de dados série. Esta operação é realizada utilizando uma chave fornecida inicialmente ao sistema pela mesma linha série. A seguinte funcionalidade é pretendida para o circuito:

- Ler a sequência de *bits* que constitui a chave (sem a reproduzir na saída)
- Criptografar as restantes palavras provenientes da linha série segundo o algoritmo que será descrito.
- Escrever cada palavra criptografada na linha série de saída.

A chave é constituída por uma sequência de 64 *bits* que são agrupados em oito palavras de 8 *bits*. O algoritmo de criptografia realiza uma operação reversível[1] sobre dois operandos que são: uma das oito palavras da chave de criptografia e uma palavra de dados de entrada constituída por 8 *bits*. A selecção da palavra da chave a utilizar, que é alterada em cada operação de criptografia, é feita segundo uma sequência pré-estabelecida. Esta sequência, é determinada por uma máquina de criptografia, cuja evolução dos estados depende do *bit* de menor peso da palavra de entrada ($dados_1$). Na tabela 6.1 é apresentada a tabela de transições dos estados desta máquina e qual a palavra da chave seleccionada em cada um dos estados.

Estado interno	Próximo estado interno		Palavra da chave
actual	$dados_1 = 0$	$dados_1 = 1$	seleccionada
A	B	C	0
B	C	E	1
C	D	G	2
D	E	A	3
E	F	A	4
F	G	C	5
G	H	E	6
H	A	G	7

Tabela 6.1: Evolução dos estados da máquina de criptografia

A operação reversível realizada é feita *bit* a *bit* pela seguinte função:

$$resul_i = \begin{cases} dados_i & i = 1 \\ dados_i \text{ xor } pchave_i \text{ xor } pchave_{i-1} & i = 2 \\ dados_i \text{ xor } pchave_i & 3 \leq i \leq 8 \end{cases}$$

em que:

$dados_i$ - *bit* i da palavra de dados.

$pchave_i$ - *bit* i da palavra da chave seleccionada para a criptografia (consoante o estado em que se enconta a máquina de criptografia).

$resul_i$ - *bit* i da palavra resultante (criptografada).

[1]Operação em que um dos operandos pode ser recuperado a partir do resultado produzido e do(s) outro(s) operando(s).

Devido ao *bit* de menor peso da palavra de entrada não ser alterado, e por ser também este *bit* que condiciona a evolução da máquina de criptografia, é possível usar o mesmo circuito para criptografar ou descriptografar dados.

6.2 Ambiente de Desenvolvimento

As ferramentas usadas para realizar os vários passos apresentados no fluxo de projecto proposto no capítulo 4 são de três fabricantes de CAD diferentes.

A simulação do código VHDL foi realizada no ambiente de desenvolvimento da Mentor Graphics (Falcon Framework). Neste sistema a descrição do circuito é compilada pela ferramenta System-1076 [Compiler 91] para posteriormente ser usada no simulador QuickSim II [Simulator 91b].

A síntese do circuito lógico foi realizada pelo sistema da Synopsys [Synthesis 91]. Este sistema aceita uma descrição do circuito em VHDL que depois de sintetizada permite a sua representação em vários formatos (EDIF, VHDL estrutural, etc). Desta forma é possível passar a lista de portas lógicas que constituem o circuito para outras ferramentas.

A simulação lógica do circuito foi realizada pelo simulador Silos 2 no ambiente de desenvolvimento do Solo 2030 da Cadence [Simulator 91a]. A representação do circuito neste sistema foi obtida a partir da conversão da lista de portas lógicas, obtidas pela ferramenta de síntese, descritas no formato EDIF.

As bibliotecas de componentes usadas para as ferramentas de síntese e simulação lógica representam os componentes existentes na tecnologia ECPD10 de 1.0 μm da European Silicon Structures (ES2) [Library 92]. Apesar de ambas as bibliotecas terem sido fornecidas pela ES2 e representarem o mesmo conjunto de componentes, foram detectadas algumas incoerências entre elas [2]. No entanto, estas foram ultrapassadas e não apresentaram problemas no processo de síntese do circuito.

[2]Foi detectada existência de uma porta lógica na biblioteca de síntese que não é reconhecida pelo simulador e a discrepância dos tempos de *hold* dos *flip-flops*. Estes estão definidos como zero na biblioteca de síntese (segundo o manual) mas o simulador lógico indica um erro quando um certo valor de *hold* não é verificado.

6.3 Os Blocos Constituintes do Circuito

O primeiro passo para obter a descrição do circuito em VHDL sintetizável consiste em definir a arquitectura do circuito a nível RTL ou seja, quais os fluxo de dados e controladores que permitem realizar a funcionalidade pretendida para o circuito.

Na figura 6.1 apresenta-se o fluxo de dados com o respectivo controlador que realizam o algoritmo de criptografia anteriormente descrito.

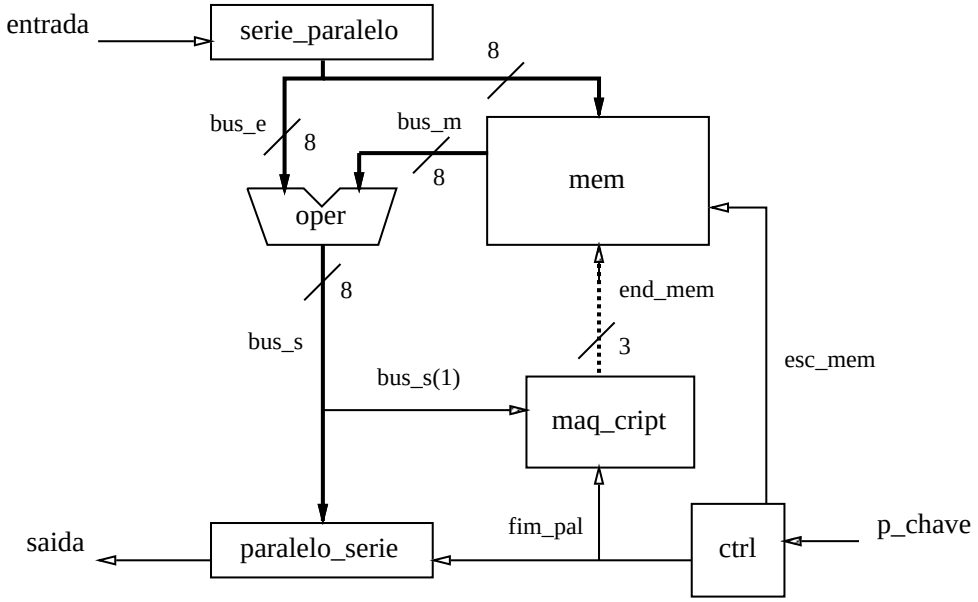


Figura 6.1: Fluxo de dados do circuito de criptografia.

A `entrada` série é convertida para uma palavra de 8 *bits* através do conversor série-paralelo (`série_paralelo`). Se esta palavra pertencer à chave, indicação dada pela linha `p_chave`, ela é guardada na memória (`mem`), através de um impulso na linha `esc_mem` que é gerado pelo controlador (`ctrl`). As palavras que não pertençam à chave são criptografadas (`oper`) recorrendo a uma palavra da chave guardada na memória. Esta, é endereçada pelo valor do *bus* `end_mem`, gerado pela máquina de criptografia (`maq_cript`). Após a operação de criptografia, é gerado pelo controlador um sinal na linha `fim_pal` que permite a evolução do estado da máquina de criptografia e carrega no conversor paralelo-série (`paralelo_serie`) o resultado da operação de criptografia. Este conversor "devolve" ao exterior a

palavra criptografada no formato série.

A descrição do circuito foi realizada utilizando uma entidade de projecto por cada bloco da figura 6.1. Nas sub-secções seguintes são apresentadas as descrições de cada uma destas entidades, conjuntamente com a sua simulação VHDL, o bloco lógico sintetizado correspondente e a respectiva simulação lógica. Note-se que, tanto a síntese como a simulação lógica de cada um dos blocos individualmente, só foram efectuadas depois de verificada, através da simulação VHDL, a funcionalidade de todo o circuito de criptografia. No entanto, optou-se por apresentar esses resultados em conjunto para mais facilmente se relacionar a descrição de cada bloco RTL com o circuito lógico sintetizado correspondente e com as respectivas simulações.

A síntese do circuito de criptografia foi realizada em duas fases: síntese de todas as entidades de uma forma hierárquica seguida de uma caracterização e optimização de cada entidade em separado. Esta forma de sintetizar o circuito apresenta as seguintes vantagens: permite impôr diferentes restrições às várias entidades que constituem o circuito e ainda, uma vez que a hierarquia é mantida, auxilia a detecção e correcção de erros.

O espaço de projecto de cada bloco, e do circuito final, foi explorado utilizando várias estratégias de optimização. Dos vários circuitos obtidos, com os diferentes compromissos de área/tempos de atraso, apenas foram seleccionados os de menor área. Desta forma, pretende-se obter o circuito final mais pequeno e portanto, mais económico.

6.3.1 O módulo `tipos`

A utilização do módulo `tipos` (figura 6.2) facilita a descrição do circuito pois permite a partilha de dados comuns às várias unidades de projecto. Assim, neste módulo são definidos constantes e subtipos necessários para a especificação do circuito e que a seguir se descrevem:

`tam_pal` - constante que define a dimensão da palavra, em número de *bits*, sobre a qual deve ser realizada a operação de criptografia.

`tam_mem` - constante que define a dimensão da memória, em número de palavras, que é necessária para guardar as palavras que constituem a chave de criptografia.

`palavra` - subtipo de dados, declarado com um vector de *bits* de comprimento `tam_pal`, que permite representar as palavras do sistema de criptografia.

`endereco` - subtipo inteiro que representa os possíveis endereços para acesso à memória.

```
package tipos is

    constant tam_pal: integer := 8;
    constant tam_mem: integer := 8;

    subtype palavra is bit_vector(tam_pal downto 1);
    subtype endereco is integer range 0 to tam_mem-1;

end tipos;
```

Figura 6.2: Descrição do módulo `tipos`.

O corpo do módulo `tipos` não existe pois na sua declaração não existem funções ou procedimentos que necessitem de serem implementados.

Note-se que a descrição deste circuito pôde ser realizada recorrendo apenas aos valores lógicos "pré-definidos" na linguagem. Isto é, não foi necessário utilizar tipos de dados que envolvam lógica multi-valor, por não existirem linhas com alta impedância nem blocos cujas saídas não se encontram completamente especificadas.

6.3.2 Conversor série-paralelo

O conversor série-paralelo permite passar os dados de entrada do formato série para uma palavra em paralelo, a qual vai ser criptografada ou guardada na memória (caso seja uma palavra da chave).

Na descrição deste bloco, apresentado na figura 6.3, foi utilizada uma construção típica de elementos de memória síncronos com *reset* assíncrono. No ramo correspondente ao *reset* todos os *bits* da palavra de saída são colocados a `'0'`. No ramo síncrono com o relógio é feita a operação de deslocamento dos *bits* de entrada através do operador de concatenação (`&`).

Devido ao acesso `paralelo` ser de saída, e portanto não poder ser lido para descrever o comportamento deste bloco, o deslocamento dos *bits* de entrada tem que ser realizado utilizando um sinal auxiliar (`paralelo_saida`). A actualização

```
use work.tipos.all;

entity serie_paralelo is
    port(clock, reset,
         serie:in bit;                    -- entrada serie de dados
         paralelo: out palavra);          -- saida paralela de dados
end serie_paralelo;

architecture reg_deslocamento of serie_paralelo is
    signal paralelo_saida: palavra;
begin

    paralelo <= paralelo_saida;

    process(clock, reset)
    begin
        if reset = '1' then
            paralelo_saida <= palavra'(others => '0');
        elsif clock'event and clock = '1' then
            paralelo_saida <= (serie & paralelo_saida(tam_pal downto 2));
        end if;
    end process;

end reg_deslocamento;
```

Figura 6.3: Descrição do conversor série-paralelo.

da saída é feita através de uma atribuição concorrente deste sinal auxiliar à saída (`paralelo`).

A simulação VHDL deste bloco é apresentada na figura 6.4 onde se verifica que este tem o comportamento pretendido. A linha de dados `paralelo` apresenta os últimos 8 *bits* que entram pela linha `serie`.

A síntese da descrição do conversor série-paralelo resultou numa cadeia de *flip-flops* com *reset* assíncrono conforme apresentado na figura 6.5. A optimização para área e velocidade deste circuito não apresentou qualquer variação, quer da área quer dos tempos de atraso obtidos inicialmente e que são respectivamente de $26\,647.0\,\mu m^2$ e $1.59\,ns$. Este comportamento da ferramenta de síntese resulta quer da simplicidade que o circuito sintetizado apresenta, quer da inexistência na biblioteca de componentes de outros *flip-flops*, que permitam a mesma funcionalidade, mas com diferentes compromissos de área/tempos de atraso.

A simulação lógica do conversor série-paralelo, que foi feita com o mesmo conjunto de estímulos usados na simulação VHDL, é apresentada na figura 6.4. Através desta simulação verificou-se que o comportamento apresentado pelo circuito sintetizado corresponde à funcionalidade pretendida para este bloco.

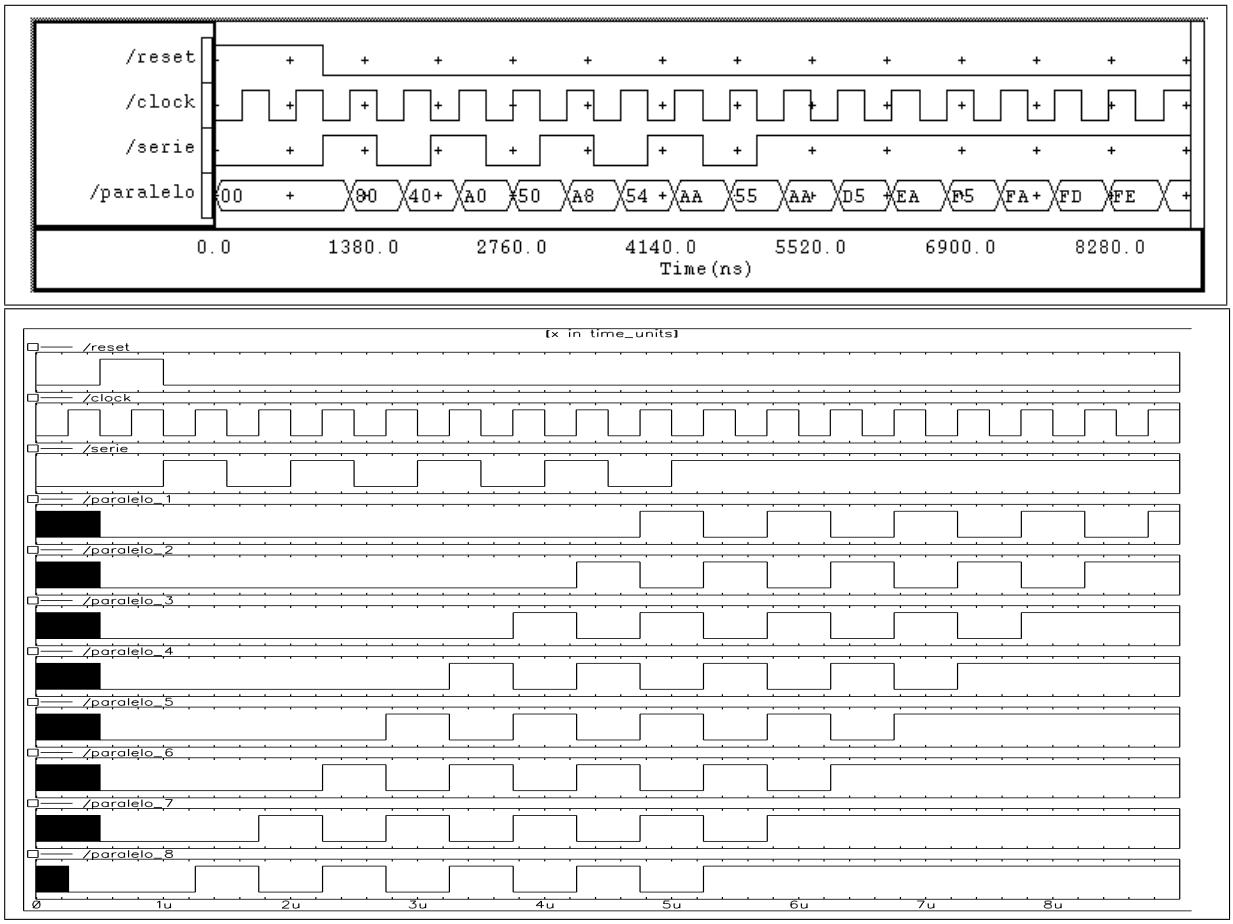


Figura 6.4: Simulação VHDL e lógica do conversor série-paralelo.

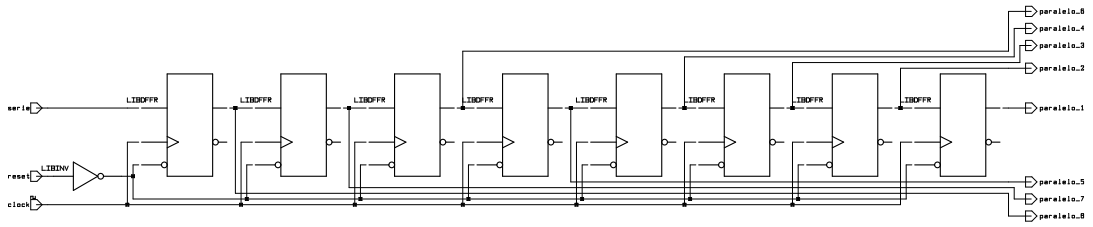


Figura 6.5: Circuito sintetizado para o conversor série-paralelo.

6.3.3 Operação de criptografia

A operação de criptografia, realizada pelo bloco `oper` do fluxo de dados apresentado na figura 6.1, é descrita em VHDL por um conjunto de três atribuições sequenciais descritas na figura 6.6.

```
use work.tipos.all;

entity oper is
    port(x,                                  -- palavra a criptografar
         y: in palavra;                      -- palavra da chave
         z: out palavra);                    -- palavra de saida criptografada
end oper;


architecture logica of oper is
begin
    process(x, y)
    begin
        z <= x xor y;
        z(1) <= x(1);
        z(2) <= x(2) xor y(2) xor y(1);
    end process;
end logica;
```

Figura 6.6: Descrição da operação de criptografia.

A síntese da descrição apresentada resultou no circuito combinatório da figura 6.7. Esta implementação do circuito é aquela que apresenta menor área ($10\,450.0\ \mu m^2$) mas que leva a que as saídas fiquem estáveis mais tarde (ao fim de $10.69\ ns$). A optimização do circuito para minimizar este atraso levou a que a saída `z_2` deixasse de ser obtida através de portas lógicas *xor*. Reduziu-se assim o tempo de atraso nessa saída ($10.45\ ns$) à custa do aumento da área do circuito ($11\,447.5\ \mu m^2$).

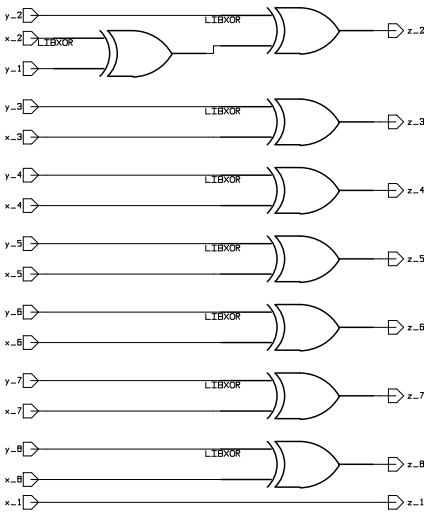


Figura 6.7: Circuito que realiza a operação de criptografia.

Devido à descrição apresentada e o circuito sintetizado serem facilmente analisados, não foi efectuado qualquer tipo de simulação para este bloco.

6.3.4 Memória

O bloco de memória destina-se a guardar o conjunto de 8 palavras que constituem a chave para o processo criptografia. Estas palavras são escritas sequencialmente na memória mas, são lidas de uma forma aleatória na fase de criptografia das palavras de entrada.

A descrição de uma memória em VHDL pode ser feita utilizando um vector bi-dimensional, que não é suportado pela ferramenta de síntese, ou utilizando um vector de vectores. Assim, para que esta descrição fosse sintetizável foi usado o tipo de dados `memoria`, representando um vector de elementos do tipo `palavra` em que cada uma representa um vector de *bits*.

A descrição deste bloco, apresentado na figura 6.8, foi feita utilizando dois processos: um responsável pela escrita sequencial e outro pela leitura aleatória da memória. Estes processos encontram-se identificados no código com as etiquetas `proc_escrita` e `proc_leitura`, respectivamente.

No processo responsável pela escrita na memória é utilizada uma construção com *reset* assíncrono e comportamento síncrono sensível ao flanco ascendente do sinal `escrita`. Este tipo de construção resultará, conforme apresentado no capítulo 4, na síntese de *flip-flops* com *reset* assíncrono.

A escrita das palavra na memória é feita utilizando um `for` que percorre todas as posições da memória recorrendo ao atributo `range`. Assim, por cada impulso no sinal `escrita` deslocam-se todas as palavras para a posição de memória seguinte e é lida uma nova palavra para a posição zero. Desta forma as palavras que constituem a chave vão sendo guardadas sequencialmente na memória. Note-se, que a utilização de uma construção sensível ao nível (e não ao flanco) apresentaria o mesmo resultado na simulação VHDL mas, o circuito sintetizado seria constituído por *latchs* (e não *flip-flops*). Esta solução apresentaria alguns problemas após a síntese e que só seriam detectados na simulação lógica, nomeadamente, o valor da entrada propagar-se-ia pelas várias posições de memória enquanto o sinal `escrita` estivesse activo a um.

O processo responsável pela leitura consiste na atribuição sequencial de uma palavra de memória indexada pela variável `posicao`. A síntese deste processo

```
use work.tipos.all;

entity mem is
    port(reset,
         escrita: in bit;                -- para escrever na memoria
         dados_ent: in palavra;          -- dados de entrada
         posicao: in endereco;           -- indica o endereco de leitura
         dados_sai: out palavra);        -- dados de saida
end mem;

architecture memo of mem is
    type memoria is array (0 to tam_mem-1) of palavra;
    signal linha_mem: memoria;           -- representa a memoria
begin
    proc_escrita: process(escrita, reset)
    begin
        if reset = '1' then
            for n in linha_mem'range loop
                linha_mem(n) <= palavra'(others => '0');
            end loop;
        elsif (escrita'event and escrita = '1') then
            for n in linha_mem'range loop
                if n = 0 then
                   linha_mem(n) <= dados_ent;
                else
                   linha_mem(n) <= linha_mem(n-1);
               end if;
           end loop;
        end if;
    end process;

    proc_leitura: process(posicao, linha_mem)
    begin
        dados_sai <= linha_mem(posicao);
    end process;

end memo;
```

Figura 6.8: Descrição da memória que armazena as palavras da chave.

resulta num multiplexer de oito palavras para uma, que é controlado pelos sinais que representam a variável `posicao`. É assim possível seleccionar qual a palavra da chave que vai ser usada na operação de criptografia.

A simulação da descrição deste bloco, apresentada na figura 6.9, mostra que esta descrição realiza a escrita sequencial e a leitura aleatória de 8 palavras de 8 *bits* conforme pretendido.

A síntese desta descrição não foi obtida de imediato, mas envolveu um ciclo de iterações ao nível da ferramenta de síntese. Estas iterações foram necessárias porque no primeiro circuito sintetizado alguns *flip-flops* tinham o seu relógio ligado

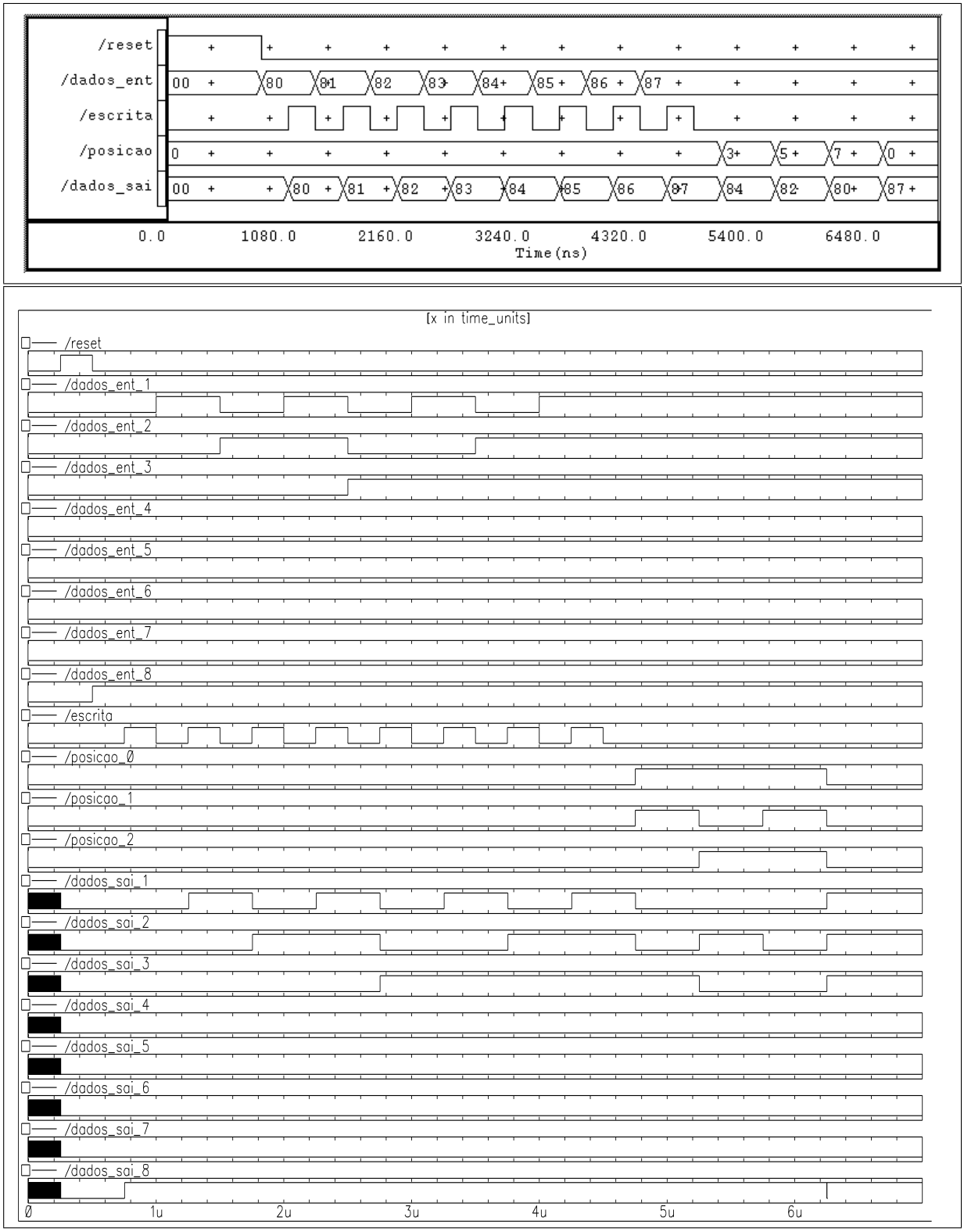


Figura 6.9: Simulação VHDL e lógica do bloco de memória.

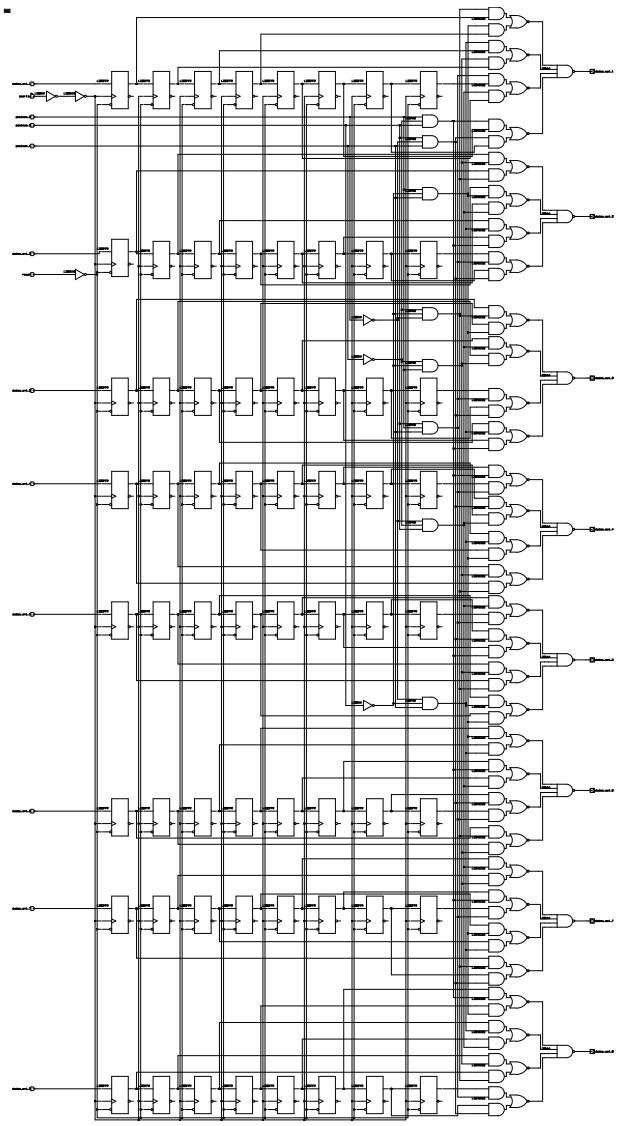


Figura 6.10: Circuito sintetizado para o bloco de memória.

directamente à linha de `escrita`, enquanto que noutros, esse sinal passava primeiro um *buffer*. A síntese desse *buffer* destinava-se a verificar a restrição, imposta pela biblioteca de componentes, do valor máximo de *fan-out* na linha `escrita`. No entanto, a desfasagem introduzida por esse *buffer* entre os relógios de alguns *flip-flops* provocava a violação dos tempos de *set-up* e *hold*. A resolução deste problema consistiu em restringir o valor máximo do *fan-out* da entrada `escrita`, para um valor suficientemente baixo, de forma a que esta apenas possa atacar uma única porta.

Tipo de Optimização	Área ocupada (memória)	Atraso até ao *flip-flop*	Atraso para a saída
Área	259 219.0 μm^2	7.95 ns	9.17 ns
Velocidade	261 474.4 μm^2	7.64 ns	8.86 ns

Tabela 6.2: Área e atrasos do bloco `mem` quando optimizado com diferentes restrições.

O circuito obtido, quando optimizado para ocupar a área mínima, encontra-se apresentado na figura 6.10. Nesta pode-se identificar os 8 *flip-flops* ligados em cadeia de forma a permitir a escrita sequencial e o multiplexer na saída do circuito para seleccionar a palavra a colocar na saída. Foi ainda realizada uma optimização para obter um circuito mais rápido. As áreas e os atrasos máximos obtidos para ambos os circuitos encontram-se apresentados na tabela 6.2, verificando-se que uma diminuição nos tempos de atraso é obtida à custa do aumento da área do circuito.

A simulação lógica deste circuito encontra-se na figura 6.9. Esta foi realizada com a mesma sequência de estímulos que foram utilizados para a simulação VHDL. A comparação dos resultados de ambas as simulações, mostra que o circuito sintetizado apresenta a funcionalidade pretendida.

6.3.5 Máquina de criptografia

A máquina de criptografia, que determina qual a palavra da chave que vai ser lida da memória para a operação de criptografia, pode ser realizada através de uma máquina de estados. Na figura 6.11 apresenta-se a descrição VHDL da máquina de estado de criptografia, cuja tabela de transições de estados corresponde à tabela 6.1. No entanto, como a mudança de estado deve ser apenas efectuada após uma palavra ter sido criptografada (e não por cada impulso do relógio), as entradas desta máquina de estados são para além do *bit* de menor peso da palavra de entrada, uma linha do controlador que indica em que ciclo do relógio deve ser efectuada a mudança de estado.

Na descrição desta máquina de estados foram seguidas as recomendações apresentadas no capítulo 4: os estados da máquina foram representados no nível simbólico utilizando o tipo enumerado `estados_maq_cript`; em cada estado foi

```
use work.tipos.all;

entity maq_cript is
    port(clock, reset,
         bit1,                          -- bit de menor peso da entrada
         muda_estado: in bit;           -- autoriza a mudanca de estado
         pos_mem: out endereco);        -- posicao de memoria da palavra chave
end maq_cript;

architecture maquina_estados of maq_cript is
    type estados_maq_cript is (A, B, C, D, E, F, G, H);
    signal estado_actual, proximo_estado: estados_maq_cript;
begin
    process(bit1, estado_actual, muda_estado)
    begin
        case estado_actual is
            when A =>
                pos_mem <= 0;
                if muda_estado = '0' then
                    proximo_estado <= A;
                elsif bit1 = '0' then
                    proximo_estado <= B;
                else
                    proximo_estado <= C;
                end if;
            when B =>
                pos_mem <= 1;
                if muda_estado = '0' then
                    proximo_estado <= B;
                elsif bit1 = '0' then
                    proximo_estado <= C;
                else
                    proximo_estado <= E;
                end if;
            when C =>
                pos_mem <= 2;
                if muda_estado = '0' then
                    proximo_estado <= C;
                elsif bit1 = '0' then
                    proximo_estado <= D;
                else
                    proximo_estado <= G;
                end if;
            when D =>
                pos_mem <= 3;
                if muda_estado = '0' then
                    proximo_estado <= D;
                elsif bit1 = '0' then
                    proximo_estado <= E;
                else
                    proximo_estado <= A;
                end if;
            when E =>
                pos_mem <= 4;
                if muda_estado = '0' then
                    proximo_estado <= E;
                elsif bit1 = '0' then
                    proximo_estado <= F;
                else
                    proximo_estado <= A;
                end if;
            when F =>
                pos_mem <= 5;
                if muda_estado = '0' then
                    proximo_estado <= F;
                elsif bit1 = '0' then
                    proximo_estado <= G;
                else
                    proximo_estado <= C;
                end if;
```

```
                when G =>
                    pos_mem <= 6;
                    if muda_estado = '0' then
                        proximo_estado <= G;
                    elsif bit1 = '0' then
                        proximo_estado <= H;
                    else
                        proximo_estado <= E;
                    end if;
                when H =>
                    pos_mem <= 7;
                    if muda_estado = '0' then
                        proximo_estado <= H;
                    elsif bit1 = '0' then
                        proximo_estado <= A;
                    else
                        proximo_estado <= G;
                    end if;
                end case;
            end process;

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

    end maquina_estados;
```

Figura 6.11: Descrição da máquina de estados do criptógrafo.

completamente definido o valor da saída e do próximo estado; e foi utilizado um sinal extra (`reset`) explicitamente para inicializar a máquina de estados.

A verificação da funcionalidade desta descrição foi feita através da simulação apresentada na figura 6.12. Nesta, verifica-se a geração dos vários endereços de memória, de acordo com o estado interno da máquina, e a inibição da mudança de estado quando a linha `muda_estado` não está activa.

Na síntese de uma máquina de estados existe uma etapa adicional que o projectista pode realizar, para além das etapas de optimização para redução de área e tempos de atraso. Essa etapa consiste na codificação da máquina de estados de forma a minimizar a área e/ou período de relógio. No caso da ferramenta utilizada [Synthesis 91], a síntese de uma máquina de estados com a codificação "óptima", a partir de uma descrição VHDL, requer vários passos:

1. Leitura e síntese da máquina de estados. A codificação desta corresponderá à numeração sequencial dos estados declarados para a máquina.

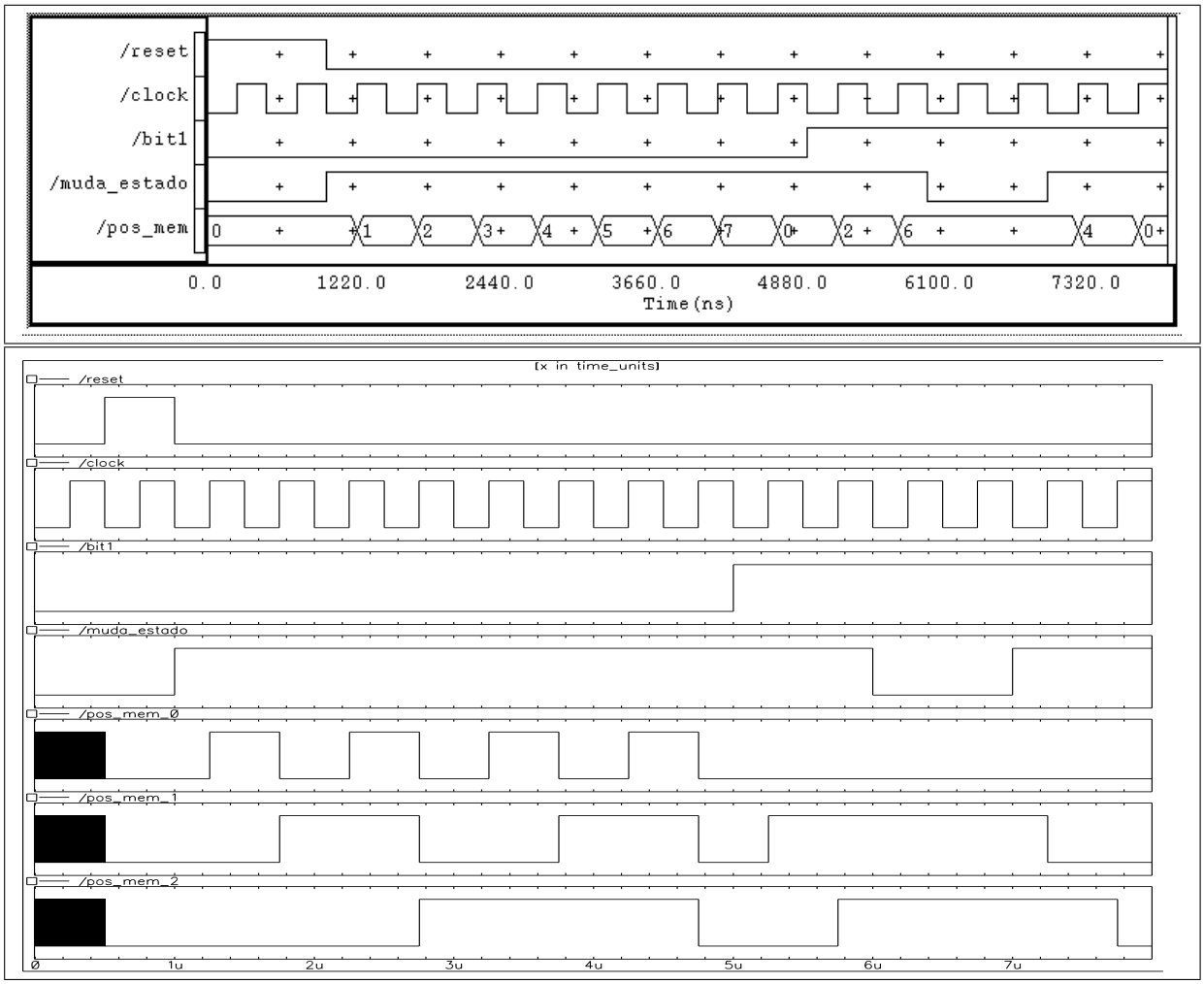


Figura 6.12: Simulação VHDL e lógica da máquina de estados de criptografia.

2. Tradução do circuito sintetizado para um formato interno de representação de máquina de estados.

3. Eliminação da codificação anterior e resintese da máquina de estados, que agora será obtida com uma codificação "óptima".

Na tabela 6.3 é apresentada a evolução da área e tempos de atraso máximos obtidos para a máquina de criptografia, quando realizadas as seguintes etapas: síntese do circuito com a codificação sequencial (etapa α); nova síntese do circuito com uma codificação "óptima" obtida pela ferramenta de síntese (etapa β); optimização do circuito anterior para reduzir a área (etapa γ); e optimização do circuito anterior para minimizar os tempos de atraso (etapa δ). Salienta-se o facto da escolha de uma codificação "óptima", só por si, reduzir a área do circuito e os tempos de atraso.

Etapa de síntese	Área ocupada (maq. criptografia)	Atraso até ao *flip-flop*	Atraso para a saída
α	$36\,912.2\ \mu m^2$	$9.85\ ns$	$4.52\ ns$
β	$24\,894.7\ \mu m^2$	$8.80\ ns$	$1.81\ ns$
γ	$23\,636.5\ \mu m^2$	$9.02\ ns$	$1.93\ ns$
δ	$28\,167.0\ \mu m^2$	$7.50\ ns$	$1.57\ ns$

Tabela 6.3: Diferentes áreas e atrasos obtidos para a máquina de estado de criptografia.

O logigrama apresentado na figura 6.13 representa o circuito obtido com a menor área. A simulação lógica deste circuito, cujos resultados se apresentam na figura 6.12, mostrou o correcto funcionamento da máquina de estados sintetizada. A geração dos endereços para a memória apresenta um comportamento igual ao obtido na simulação VHDL.

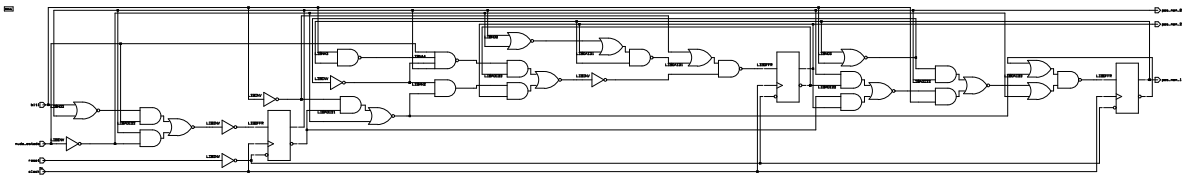


Figura 6.13: Circuito sintetizado para a máquina de criptografia.

6.3.6 Conversor paralelo-série

O conversor paralelo-série converte cada palavra criptografada, constituída por 8 *bits* em paralelo no formato série pretendido para a saída do circuito de criptografia.

A descrição deste conversor, apresentada na figura 6.14, foi feita recorrendo a uma construção típica de elementos de memória síncronos com *reset* assíncorno. No ramo correspondente ao funcionamento síncrono foi usada uma instrução `if-then-else` para seleccionar entre a leitura de uma nova palavra para o conversor ou o deslocar da palavra já existente neste. A síntese desta instrução resultará num multiplexer à entrada de cada *flip-flop*, possibilitando o carregamento síncrono do conversor ou a ligação dos vários *flip-flops* para formarem um registo

de deslocamento.

```
use work.tipos.all;

entity paralelo_serie is
    port(clock, reset,
         carrega:in bit;                    -- carregamento sincrono paralelo
         paralelo: in palavra;              -- entrada de dados em paralelo
         serie: out bit);                   -- saida serie
end paralelo_serie;


architecture ps_carrega of paralelo_serie is
    signal paralelo_saida: palavra;
begin

    serie <= paralelo_saida(1);

    synch: process(clock, reset)
    begin
        if reset = '1' then
            paralelo_saida <= palavra'(others => '0');
        elsif (clock'event and clock = '1') then
            if carrega = '1' then
                paralelo_saida <= paralelo;
            else
                paralelo_saida <=
                ('0' & paralelo_saida(tam_pal downto 2));
            end if;
        end if;
    end process;

end ps_carrega;
```

Figura 6.14: Descrição do conversor paralelo-série.

Através de uma atribuição concorrente ao sinal `serie`, a saída do conversor é obtida pela indexação do último *bit* da palavra que este tem carregada. Como esta indexação é constante, a sua síntese resultará no fio ligado à saída do registo que se está a indexar.

A simulação VHDL da figura 6.15 mostra que o comportamento pretendido para este bloco é realizado pela descrição apresentada. O carregamento paralelo de uma nova palavra só é realizado quando o sinal `carrega` está activo.

O circuito sintetizado para este bloco encontra-se na figura 6.16 e mostra, tal como pretendido, a existência de um multiplexer à entrada de cada *flip-flop* e a saída `serie` ligada à saída do último *flip-flop*. Este circuito, apresenta uma área de $39\,244.7\,\mu m^2$ e atrasos máximos de $11.65\,ns$ e $1.78\,ns$, respectivamente, da entrada até aos *flip-flop* e destes para a saída. Foi ainda realizada uma optimização sobre

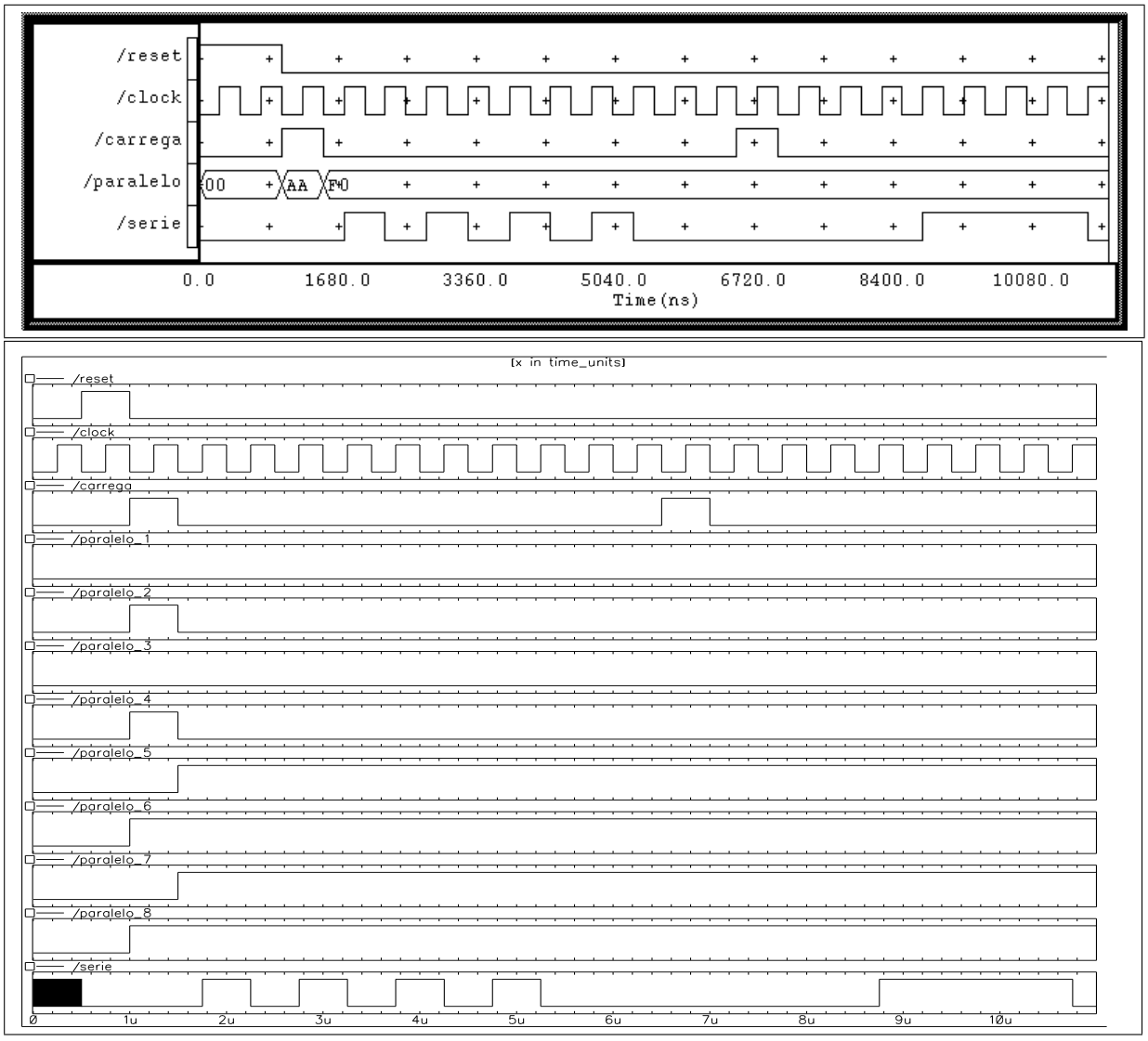


Figura 6.15: Simulação VHDL e lógica do conversor paralelo-série.

o circuito apresentado, com o objectivo de o tornar mais rápido, obtendo-se os atrasos de 11.48 ns e 1.78 ns à custa do aumento da área ocupada pelo circuito, que passou para 39 828.7 μm^2.

A simulação lógica do circuito apresentado, encontra-se na figura 6.15. Pela análise desta, pode-se concluir o correcto funcionamento do circuito: conversão para a linha série feita correctamente e carregamento síncrono do registo só feito quando a linha `carrega` está activa.

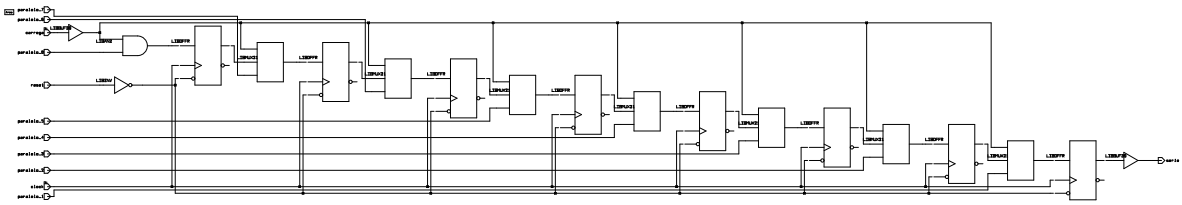


Figura 6.16: Circuito sintetizado para o conversor paralelo-série.

6.3.7 Controlador

O controlador (`ctrl`) gera os sinais necessários para a correcta operação do fluxo de dados. Os sinais `esc_mem` e `fim_pal` indicam, respectivamente, quando deve ser escrita uma palavra da chave na memória ou quando deve ser carregado um novo valor no conversor paralelo-série.

Este bloco encontra-se dividido em dois sub-blocos constituídos pelas entidades `maq_ctrl` e `reduz`, que descrevem a máquina de estados do controlador e fazem o condicionamento das saídas desta. Na figura 6.17 apresenta-se a descrição estrutural do controlador, feito à custa da instanciação das entidades acima referidas e cujas descrições se apresentam nas sub-secções seguintes.

Saliente-se que, para que esta descrição pudesse ser usada pelo simulador de VHDL foi necessário configurar cada elemento instanciado. No entanto, devido à ferramenta de síntese não suportar construções de configuração, teve de se recorrer ao uso de comentários sintéticos para que estas não fossem interpretadas. Assim, a mesma descrição do circuito pôde ser usada para a simulação e para a síntese.

A máquina de estados

A geração dos sinais de controlo `esc_mem` e `fim_pal` pode ser feita através de uma máquina de estados. Esta, só deve activar estes sinais no oitavo impulso

```
entity ctrl is
    port(clock, reset,
        p_chave: in bit;          -- indica uma palvra da chave
        fim_pal,                  -- acabou uma palvra a criptografar
        esc_mem: out bit);        -- acabou uma palvra da chave
end ctrl;

architecture final of ctrl is

component maq_ctrl
    port(clock, reset, p_chave_loc: in bit;
         fim_pal_loc, esc_mem_loc: out bit);
end component;

component reduz
    port(clock, fp, em: in bit; fp_ck, em_ck: out bit);
end component;

-- synopsys translate_off

    for mq: maq_ctrl use entity work.maq_ctrl(controlo_cripto);
    for rd: reduz use entity work.reduz(logica);

-- synopsys translate_on

signal fim, esc: bit;

begin

mq: maq_ctrl
    port map (clock, reset, p_chave, fim, esc);
rd: reduz
    port map (clock, fim, esc, fim_pal, esc_mem);

end final;
```

Figura 6.17: Descrição (estrutural) do controlador do circuito.

de relógio, ou seja, quando a palavra que entrou pela linha série de entrada se encontra disponível à saída do conversor série-paralelo. A identificação das palavras pertencentes à chave é feita pelo sinal `p_chave`. Este sinal deve estar activo durante a leitura do último bit de cada palavra pertencente à chave.

A implementação deste controlador pode ser conseguida à custa de uma máquina de estados cuja tabela de transições se encontra representada na tabela 6.4.

A descrição desta máquina de estados, apresentada na figura 6.18 foi feita tendo em conta as recomendações apresentadas para descrições sintetizáveis de máquinas de estado: no processo combinatório os sinais de saída e o próximo estado estão completamente definidos para cada estado da máquina; no processo sequencial é utilizada uma construção com *reset* assíncoro que coloca o controlador no estado

```
use work.tipos.all;

entity maq_ctrl is
    port(clock, reset,
        p_chave_loc: in bit;     -- indica uma palavra da chave
        fim_pal_loc,             -- acabou uma palavra de dados
        esc_mem_loc: out bit);   -- acabou uma palavra da chave
end maq_ctrl;

architecture controlo_cripto of maq_ctrl is
    type estados_maq_ctrl is (um, dois, tres, quatro, cinco, seis,
                              sete, normal, chave);
    signal estado_actual, proximo_estado: estados_maq_ctrl;
begin

    process(p_chave_loc, estado_actual)
    begin
        fim_pal_loc <= '0';
        esc_mem_loc <= '0';
        case estado_actual is
            when um =>
                proximo_estado <= dois;
            when dois =>
                proximo_estado <= tres;
            when tres =>
                proximo_estado <= quatro;
            when quatro =>
                proximo_estado <= cinco;
            when cinco =>
                proximo_estado <= seis;
            when seis =>
                proximo_estado <= sete;
            when sete =>
                if p_chave_loc = '1' then
                    proximo_estado <= chave;
                else
                    proximo_estado <= normal;
                end if;
            when normal =>
                fim_pal_loc <= '1';
                esc_mem_loc <= '0';
                proximo_estado <= um;
            when chave =>
                fim_pal_loc <= '0';
                esc_mem_loc <= '1';
                proximo_estado <= um;
            end case;
        end process;

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

end controlo_cripto;
```

Figura 6.18: Descrição da máquina de estado do controlador do circuito.

Estado Actual	Próximo Estado		Saídas	
	`p_chave` = 0	`p_chave` = 1	`fim_pal`	`esc_mem`
um	dois	dois	0	0
dois	tres	tres	0	0
tres	quatro	quatro	0	0
quatro	cinco	cinco	0	0
cinco	seis	seis	0	0
seis	sete	sete	0	0
sete	normal	chave	0	0
normal	um	um	1	0
chave	um	um	0	1

Tabela 6.4: Tabela de transições de estados do controlador.

`normal`. Refira-se que, devido às saídas tomarem o valor `'0'` na maioria dos estados, não foi indicado explicitamente em cada estado o valor da saída. Optou-se antes por colocar as saídas sempre a `'0'`, de uma forma independente do estado da máquina (atribuições às saídas antes da instrução `case`), e depois atribuir-lhes um novo valor apenas nos estados específicos em que elas deveriam ser diferentes de `'0'`.

Condicionamento das Saídas

O bloco de condicionamento das saídas, apresentado na figura 6.19, "transforma" um sinal que é activo durante todo um ciclo de relógio, noutro que apenas é activo durante meio ciclo de relógio (quando este vale `'0'`).

```
entity reduz is
    port(clock,
         fp, em: in bit;          -- ent. activas durante um ciclo de relogio
         fp_ck, em_ck: out bit); -- sai. activas durante meio ciclo de relogio
end reduz;

architecture logica of reduz is

begin

    fp_ck <= fp and not clock;
    em_ck <= em and not clock;

end logica;
```

Figura 6.19: Descrição do "condicionador" das saídas do controlador.

A necessidade da utilização deste bloco surgiu após a realização de um ciclo de iteração ao nível global. Isto é, depois de ter sido sintetizado o circuito e realizada a sua simulação lógica foram detectados problemas de *set-up* e *hold* nos *flip-flops* que constituem a memória e o conversor paralelo-série. Com a inserção deste bloco as saídas do controlador, que permitem carregar a memória ou o registo paralelo-série, só são activadas meio ciclo de relógio após a variação das entradas nesses blocos, evitando assim os problemas referidos.

A simulação realizada para o controlador (com os blocos `maq_cripto` e `reduz`) encontra-se na figura 6.20, onde se verifica a geração dos sinais de controlo nos intervalos de tempo pretendidos.

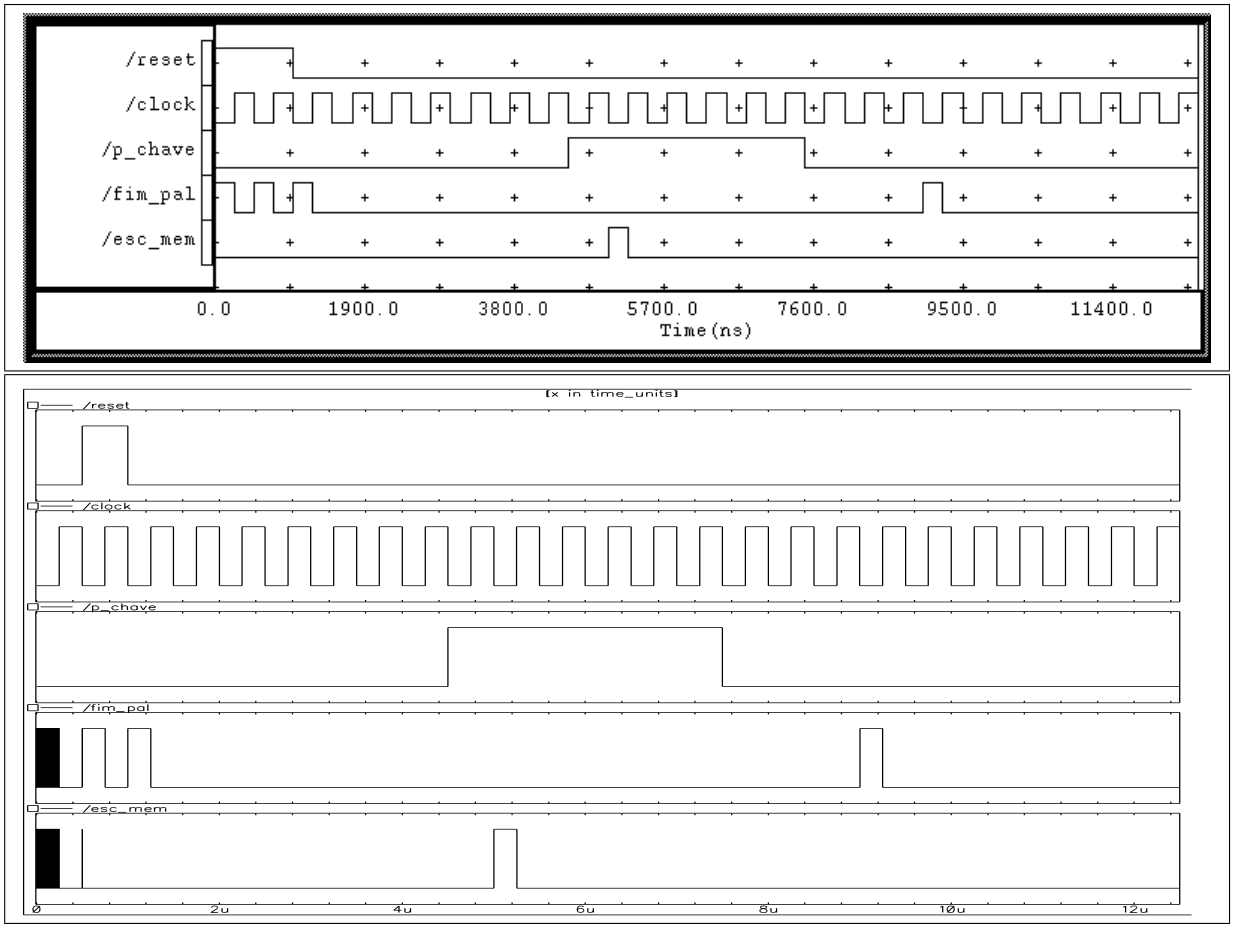


Figura 6.20: Simulação VHDL e lógica do controlador.

A síntese do controlador foi realizada em várias etapas: síntese individual de cada um dos blocos que o constituem (etapa 1); codificação da máquina de estados seguida de uma optimização para minimizar a área (etapa 2); eliminação da hierarquia do controlador, a partir do circuito obtido na etapa 2, e nova optimização

para minimizar a área (etapa 3); e optimização do circuito anterior no sentido de obter um controlador mais rápido (etapa 4). As áreas e os atraso máximos dos circuito obtidos em cada uma das etapas encontram-se na tabela 6.5. Desta, salienta-se a redução de área obtida devido à eliminação da hierarquia no circuito evidenciando a importância que tem a interacção do projectista com a ferramenta de síntese.

Etapa de síntese	Área ocupada (controlador)	Atraso até ao *flip-flop*	Atraso para a saída
1	32 613.9 μm^2	3.90 ns	5.96 ns
2	22 010.1 μm^2	3.28 ns	3.45 ns
3	20 703.3 μm^2	3.52 ns	3.04 ns
4	24 094.4 μm^2	2.36 ns	2.69 ns

Tabela 6.5: Diferentes áreas e atrasos obtidos para o controlador.

Na figura 6.21 encontra-se representado o circuito de menor área que foi obtido para o controlador (resultante da etapa 3). A simulação lógica deste circuito, apresentada na figura 6.20, difere da simulação VHDL no "pico" que aparece na linha `esc_mem`[3]. No entanto, como este surge durante o *reset* do sistema, não altera a funcionalidade do circuito, pelo que não houve necessidade de efectuar qualquer tipo de modificação no controlador do circuito.

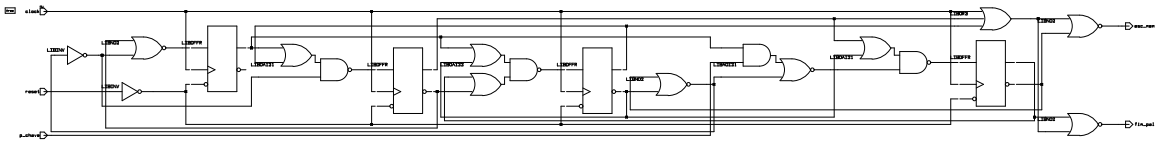


Figura 6.21: Circuito sintetizado para o controlador.

6.4 O Circuito Final

O circuito de criptografia foi representado em VHDL através da descrição estrutural da figura 6.22. Nesta descrição, os unidades de projecto apresentados na secção

[3]O "pico" tem o valor lógico indeterminado e resulta do atraso que o efeito do *reset* nos *flip-flops*, leva a chegar à saída `esc_mem` (e por sinal de relógio já se encontrar a `0`).

anterior foram interligados de forma a representarem a estrutura pretendida para o circuito (figura 6.1). Nessa descrição foi necessário configurar cada elemento instanciado, para que o código VHDL pudesse ser simulado, e recorrer ao uso de comentários sintéticos para que essas configurações não fossem interpretadas pela ferramenta de síntese.

A verificação da funcionalidade do circuito de criptografia foi realizada através da simulação de uma unidade de teste escrita em VHDL. Nesta unidade, apresentada na figura 6.23, são interligados dois circuitos (o primeiro, `C1`, a funcionar como criptógrafo e o segundo, `C2`, a realizar a operação inversa), gerados os vários estímulos para o funcionamento dos circuitos (processos identificados com as etiquetas `simul1` e `simul2`). Note-se que, devido a esta unidade de projecto se destinar apenas a verificar a funcionalidade do circuito (não necessitando por isso de ser sintetizado), pode ser escrita utilizando todas as construções de VHDL suportadas pelo simulador utilizado.

O resultado da simulação da unidade de teste encontra-se na figura 6.24. Nesta, verifica-se o fornecimento aos dois criptógrafos de uma palavra chave igual (sinal `dados_ent` enquanto `p_chave1` activo e `entrada2` enquanto `p_chave2` activo), a criptografia dos dados de entrada (saída `dados_cripto`) e a recuperação dos dados entrados[4] (saída `dados_sai`). O sinal `erro`, que detecta o funcionamento incorrecto da sequência dos dois criptógrafos, nunca é activado mostrando a funcionalidade pretendida para o circuito descrito.

A "síntese" da descrição estrutural apresentada resultou no esquemático da figura 6.25, que apenas representa os blocos e as interligações que constituem o circuito.

A síntese do circuito de criptografia só fica completa depois de seleccionado para cada bloco, de entre as várias opções apresentadas na secção anterior, qual o circuito lógico que o representa. É assim possível escolher um misto de circuitos optimizados em área e em velocidade de forma a explorar o espaço de projecto do criptógrafo. Na tabela 6.6 são apresentadas as áreas e os tempos de atraso obtidos para o circuito de critpografia quando: são seleccionados todos os blocos resultantes da optimização para minimizar a área (etapa A), e quando para os blocos pertencentes ao caminho crítico[5] vão sendo seleccionados os circuitos an-

[4]Conforme foi referido na secção 6.1, os dados só são possíveis de recuperar com o mesmo circuito de criptografia porque o *bit* de menor peso de cada palavra não é alterado.

[5]Caminho cujo tempo de propagação limita a velocidade máxima de funcionamento de circuito.

```
use work.tipos.all;

entity cripto is
    port(entrada,                          -- linha serie de entrada de dados
         p_chave,                          -- indicacao de palavra chave
         clock,                            -- relogio do circuito
         reset: in bit;                    -- reset do circuito
         saida: out bit);                  -- linha serie de saida de dados
end cripto;

architecture est of cripto is

component serie_paralelo
    port(clock, reset, serie:in bit; paralelo: out palavra);
end component;

component paralelo_serie
    port(clock, reset, carrega:in bit; paralelo: in palavra; serie: out bit);
end component;

component oper
    port(x, y: in palavra; z: out palavra);
end component;

component mem
    port(reset, escrita: in bit; dados_ent: in palavra; posicao: in endereco;
         dados_sai: out palavra);
end component;

component maq_cript
    port(clock, reset, bit1, muda_estado: in bit; pos_mem: out endereco);
end component;

component ctrl
    port(clock, reset, p_chave: in bit; fim_pal, esc_mem: out bit);
end component;

signal bus_e, bus_m, bus_s: palavra;
signal fim_pal, esc_mem: bit;
signal end_mem: endereco;

-- synopsys translate_off

    for sp: serie_paralelo use entity work.serie_paralelo(reg_deslocamento);
    for op: oper use entity work.oper(logica);
    for ps: paralelo_serie use entity work.paralelo_serie(ps_carrega);
    for mm: mem use entity work.mem(memo);
    for cc: maq_cript use entity work.maq_cript(maquina_estados);
    for gc: ctrl use entity work.ctrl(final);

-- synopsys translate_on

begin

sp: serie_paralelo
    port map (clock, reset, entrada, bus_e);
op: oper
    port map (bus_e, bus_m, bus_s);
ps: paralelo_serie
    port map (clock, reset, fim_pal, bus_s, saida);
mm: mem
    port map (reset, esc_mem, bus_e, end_mem, bus_m);
cc: maq_cript
    port map (clock, reset, bus_s(1), fim_pal, end_mem);
gc: ctrl
    port map (clock, reset, p_chave, fim_pal, esc_mem);

end est;
```

Figura 6.22: Descrição estrutural do circuito de criptografia.

```
entity teste is
    port(dados_ent, dados_cripto, dados_sai, erro: out bit);
    constant per: time := 500 ns;
    constant meio_per: time := per/2;
end teste;

architecture t of teste is
    subtype pal is bit_vector(1 to 8);
    type pal2 is array (1 to 8) of pal;
    constant chave: pal2 := (
           "11111111",
           "11101011",
           "00100101",
           "10011100",
           "11111111",
           "11101011",
           "00100101",
           "10011100"
           );

     constant pal_a_cod: pal2 := (
           "11111111",
           "00000000",
           "10101010",
           "11001101",
           "01010101",
           "00110011",
           "00000000",
           "11111111"
           );

    procedure seq_de_entrada (constant seq: pal2; signal dados: out bit) is
        variable pa: pal;
    begin
        for p in 1 to 8 loop
            pa := seq(p);
            for i in 1 to 8  loop
                dados <= pa(i);
                wait for per;
            end loop;
        end loop;
    end seq_de_entrada;

    component cripto
        port(entrada, p_chave, clock, reset: in bit;
            saida: out bit);
    end component;

    for all: cripto use entity work.cripto(est);

    signal entrada1, saida1, entrada2_ext, entrada2, saida2: bit;
    signal clock, p_chave1, p_chave2, reset1, reset2: bit;
    signal liga, compara: bit;

begin

    C1: cripto port map (entrada1, p_chave1, clock, reset1, saida1);
    C2: cripto port map (entrada2, p_chave2, clock, reset2, saida2);

    dados_ent <= entrada1;
    dados_cripto <= saida1;
    dados_sai <= saida2;

    clock <= not clock after meio_per;
```

```
    simul1: process
    begin
        reset1 <= '1';
        wait for 2*per;                          -- reset
        reset1 <= '0';

        p_chave1 <= '1';
        seq_de_entrada(chave, entrada1);         -- chave para C1
        p_chave1 <= '0';

        seq_de_entrada(pal_a_cod, entrada1);     -- dados para C1
        seq_de_entrada(pal_a_cod, entrada1);

        assert false report "Fim da simulacao" severity failure;
    end process;

    multiplexer: process(liga, entrada2_ext, saida1 )
    begin
        if liga = '0' then
            entrada2 <= entrada2_ext;
        else
            entrada2 <= saida1;
        end if;
    end process;

    simul2: process
    begin
        reset2 <= '1';
        wait for 2*per + 8*per + per;            -- reset + atraso C1
        reset2 <= '0';

        p_chave2 <= '1';
        seq_de_entrada(chave, entrada2_ext);     -- chave para C2
        p_chave2 <= '0';

        liga <= '1';                             -- liga C1 a C2

        wait for  8*per;                         -- atraso C2
        compara <= '1';
-- inicia a comparacao
        wait;                                    -- de resultados
    end process;

    compara_res: process(compara, clock)
    variable ipa, i: integer range 1 to 8;
    variable pa: pal;
    begin

        if compara = '1' and clock'event and clock = '0' then
            pa := pal_a_cod(ipa);
            if saida2 = pa(i) then
                erro <= '0';
            else
                erro <= '1';
            end if;
            if i = 8 then
                i := 1;
                if ipa = 8 then
                      ipa := 1;
                else
                     ipa := ipa + 1;
                end if;
            else
                i := i + 1;
            end if;
        end if;
    end process;
end t;
```

Figura 6.23: Descrição da unidade de teste para o circuito de criptografia.

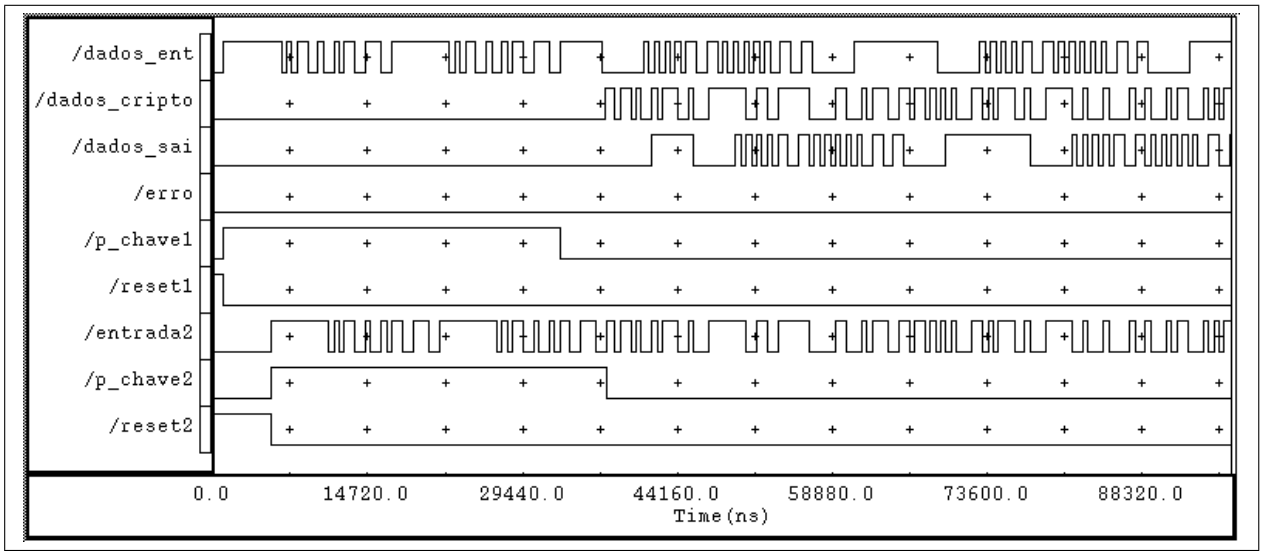


Figura 6.24: Simulação VHDL do circuito de criptografia utilizando a unidade de teste.

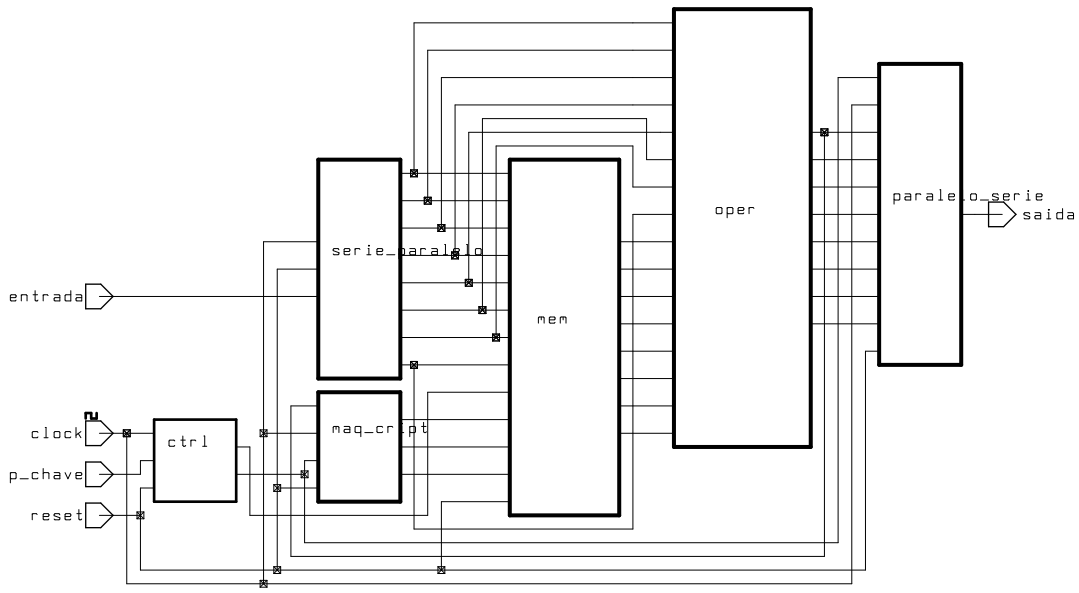


Figura 6.25: Circuito de criptografia no nível hierárquico mais elevado.

teriormente apresentados com menores atrasos. Inicialmente é escolhido apenas o circuito de memória mais rápido (etapa B) e depois, também o controlador mais rápido (etapa C).

A eliminação da hierarquia do circuito de criptografia apresenta desvanta-

Etapa de síntese	Área ocupada (criptógrafo)	Atraso até ao *flip-flop*	Atraso para a saída
A	380 865.4 μm^2	9.58 ns	1.78 ns
B	384 256.2 μm^2	9.22 ns	1.78 ns
C	386 511.5 μm^2	8.92 ns	1.78 ns
D	378 904.4 μm^2	9.72 ns	1.78 ns
E	385 419.0 μm^2	8.67 ns	1.78 ns

Tabela 6.6: Diferentes áreas e atrasos obtidos para o criptógrafo.

gens no que respeita à detecção de erros mas, permite realizar a optimização do circuito para além da "fronteira" de cada bloco. Ou seja, o circuito é optimizado como se tratasse de um grande bloco de lógica sequencial. Na tabela 6.6 apresentam-se também os resultados obtidos resultantes da eliminação da hierarquia do criptógrafo (com os circuitos de menor área) seguido de uma primeira optimização para redução da área (etapa D) e de uma segunda para redução dos tempos de atraso (etapa E).

O circuito obtido para o criptógrafo com menor área encontra-se na figura 6.26. Os resultados da simulação lógica deste circuito são apresentados na figura 6.27. Nesta figura é também apresentada uma simulação VHDL, realizada com a mesma sequência de estímulos da simulação lógica, para permitir a comparação dos resultados obtidos em ambas as simulações. Pela análise desta, pode-se concluir que a interligação dos vários circuitos obtidos anteriormente, para criar o circuito de criptografia, não apresentou problemas quer de funcionalidade quer temporais.

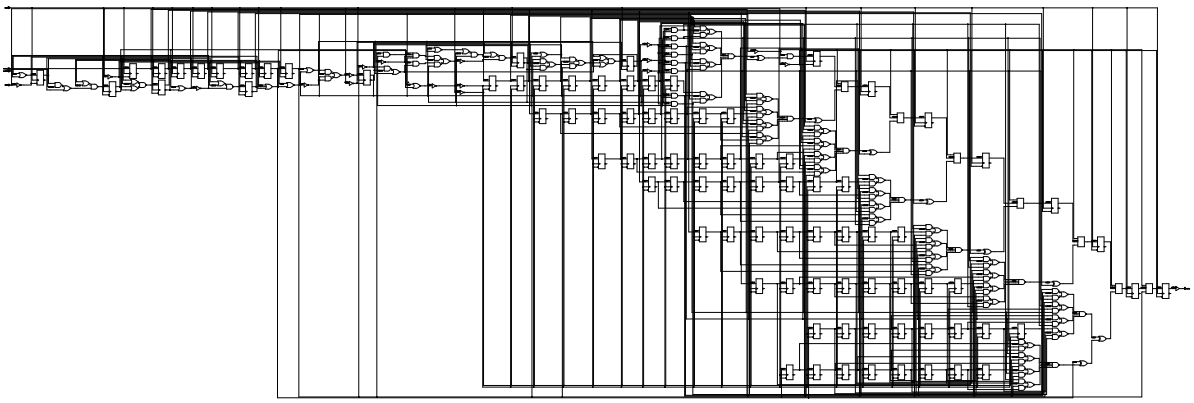


Figura 6.26: Circuito lógico do criptógrafo.

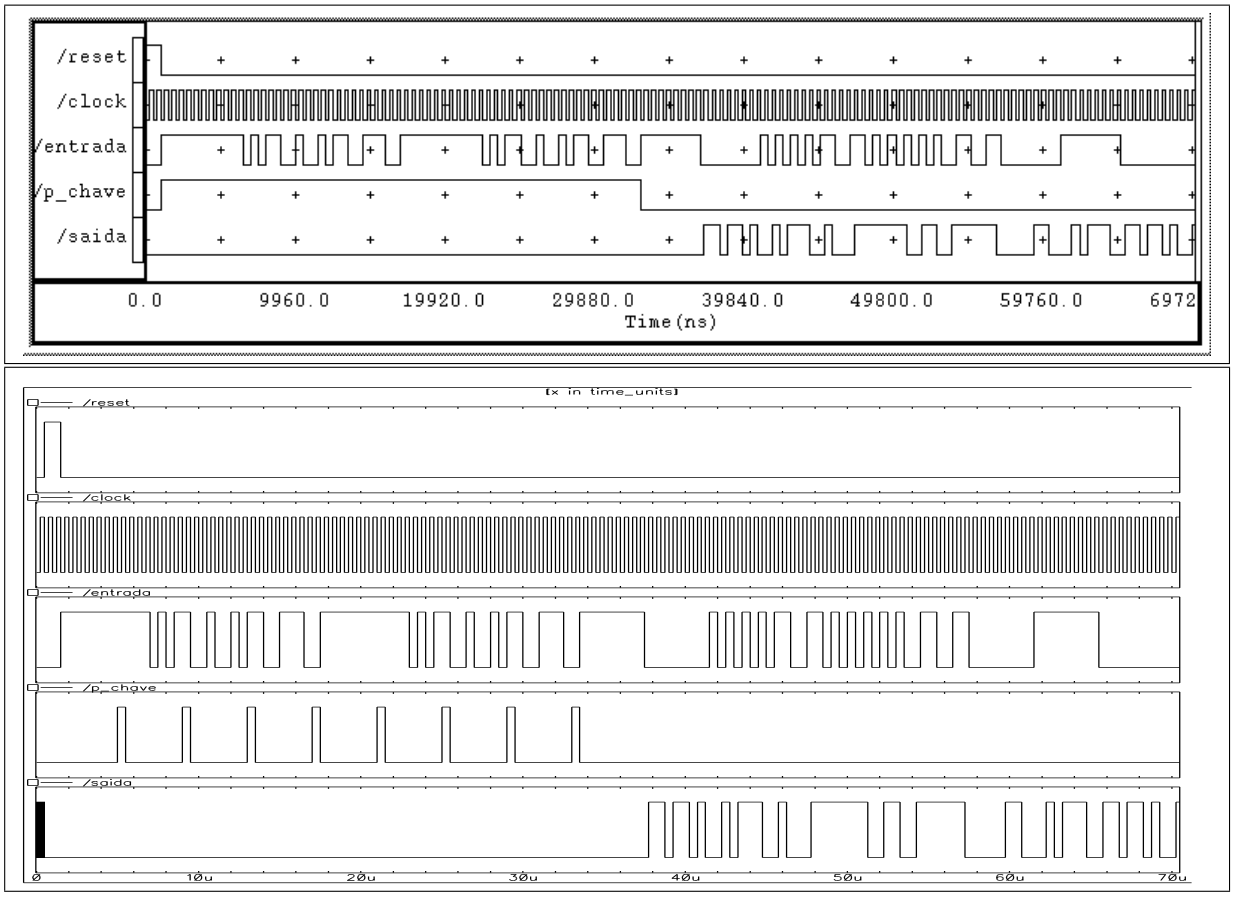


Figura 6.27: Simulação VHDL e lógica do circuito de criptografia.

6.5 Conclusões

O projecto de um circuito integrado, num ambiente de síntese, permitiu exemplificar as várias etapas do fluxo de projecto descrito no capítulo 4. A descrição do circuito a nível RTL, constituída por vários blocos (entidades de projecto), foi realizada atendendo ao subconjunto de instruções sintetizáveis e ao estilo de descrição anteriormente apresentados. A funcionalidade de cada um desses blocos e do circuito foi verificada através da simulação das respectivas descrições. A utilização de várias estratégias de optimização na síntese dos diferentes blocos, possibilitou que o espaço de projecto destes, e consequentemente do circuito, pudesse ser facilmente explorado. Finalmente, foi realizada a simulação lógica dos vários subcircuitos sintetizados, e também do circuito, para verificar que estes apresentavam o desempenho e a funcionalidade pretendida, facto que só foi conseguido após a realização de alguns ciclos de iteração.

Capítulo 7

Conclusões e Trabalho Futuro

Neste capítulo são apresentadas algumas conclusões sobre o trabalho realizado e apontados alguns dos seus possíveis desenvolvimentos futuros.

7.1 Conclusões

A realização de circuitos integrados cada vez mais complexos tornou necessário que o seu projecto fosse iniciado a níveis de abstracção mais elevados que o tradicional nível lógico. A utilização de uma linguagem de descrição de *hardware*, como o VHDL, é a forma mais usual de representar um circuito num nível de abstracção superior ao nível lógico. Esta forma de representação permite ainda que o projectista verifique a funcionalidade do circuito (por simulação), antes de se comprometer com uma dada tecnologia ou fabricante. O uso de ferramentas de síntese possibilita a utilização dessas representações para obter de uma forma "automática" a implementação lógica do circuito, mapeada em células de uma dada tecnologia e satisfazendo um determinado conjunto de restrições (por exemplo, área e/ou tempos de atraso).

Neste trabalho foram apresentadas as principais características da linguagem VHDL e analisados os principais condicionamentos que a sua utilização eficaz num ambiente de síntese implicam. Estes, resultam quer da linguagem ter sido desenvolvida inicialmente para a simulação de circuitos, e não para síntese, quer da juventude da tecnologia actual de síntese, não existindo nomeadamente e por enquanto, ferramentas comerciais que permitam a síntese de circuitos descritos

num nível de abstracção superior ao nível RTL. Assim, a utilização de descrições VHDL em ambientes de síntese, obriga a uma restrição das potencialidades de descrição da linguagem. Não existe ainda um subconjunto normalizado de construções sintetizáveis, embora o grupo de síntese, em associação com o IEEE, esteja empenhado na sua criação. Neste sentido, procurou-se definir no início deste trabalho um subconjunto de instruções sintetizáveis, que reflectisse as construções que são suportadas pela maioria das ferramentas de síntese.

No entanto, a utilização de construções sintetizáveis não garante só por si que a descrição seja sintetizável, nem que se o for, que o circuito obtido tenha o desempenho desejado. A forma de descrever um circuito influencia de forma determinante a qualidade da sua realização lógica, razão pela qual a sua descrição deve ter sempre em mente o tipo de *hardware* que se pretende representar. Neste trabalho foi proposto um estilo de descrição para a representação do conjunto fundamental de blocos que constituem os circuitos electrónicos digitais: blocos de fluxo de dados e blocos de controlo (geralmente realizados através de máquinas de estado). Assim, foram apresentadas formas de descrever os seguintes blocos:

- lógica combinatória, em que nem todas as combinações de entrada têm que estar definidas
- lógica sequencial, com a inicialização dos elementos de memória (*latchs* ou *flip-flops*) a um valor lógico definido
- elementos com saídas em alta impedância
- máquinas de estados de Mealy ou de Moore

Foi também desenvolvida uma biblioteca de modelos destinada a um ambiente de síntese, que tomou em consideração as recomendações apresentadas ao longo deste trabalho. Esta biblioteca permite ao projectista obter, de uma forma mais eficiente, a descrição sintetizável de um circuito.

Finalmente, a metodologia de projecto proposta foi seguida na realização do projecto de um circuito digital de média dimensão. A utilização do estilo de descrição proposto, permitiu reduzir o número de iterações efectuadas para obter a representação do circuito em VHDL sintetizável, permitindo ao projectista explorar mais eficientemente os diferentes compromissos entre a área ocupada pelo circuito e os tempos de atraso deste. Foi assim possível conseguir uma representação lógica do circuito com o desempenho desejado, que foi verificado por simulação lógica, através da utilização da metodologia de projecto proposta.

O uso da linguagem VHDL na especificação de circuitos digitais e a utilização da tecnologia de síntese, permite o desenvolvimento de circuitos de melhor qualidade num menor espaço de tempo. No entanto, estas vantagens só serão significativas se for utilizada uma metodologia de projecto apropriada, na qual são impostas restrições ao estilo de descrição utilizado e são aproveitadas todas as potencialidades da ferramenta de síntese.

7.2 Desenvolvimentos Futuros

O processo de alteração da norma que descreve a linguagem VHDL, no sentido de a adaptar às necessidades actuais dos seus utilizadores, encontra-se na sua fase final. O estudo da "nova" linguagem resultante deste processo de evolução (VHDL'92) e a análise das consequências que as alterações introduzidas podem ter num ambiente de síntese, quer a nível das construções sintetizáveis quer a nível do estilo de descrição, apresentam-se como natural desenvolvimento do trabalho apresentado.

De igual forma, o trabalho que o grupo de síntese está a desenvolver poderá alterar, num futuro próximo, o estilo de descrição para síntese. O módulo que este grupo pretende normalizar define vários tipos de dados e rotinas destinados à síntese que deverão ser suportados por todas as ferramentas, facilitando assim a portabilidade de descrições VHDL entre diferentes ambientes de síntese.

A definição de um estilo de descrição VHDL para um ambiente de síntese permite que sejam desenvolvidas ferramentas de dois tipos: verificadores e geradores. As primeiras verificam se uma dada representação de um circuito em VHDL tem o estilo de descrição apropriado para a síntese, podendo mesmo dar algumas sugestões sobre alterações a efectuar. As segundas, geram automaticamente código VHDL para síntese a partir de outras linguagens de descrição de *hardware* ou de outras formas de representação. Serão exemplo deste tipo de ferramentas, editores de esquemáticos a nível RTL, em que uma descrição VHDL sintetizável dos componentes utilizados (como os da biblioteca de modelos apresentada no capítulo 5) possa ser automaticamente gerada, e ferramentas que utilizam uma interface gráfica para descrever máquinas de estado através de diagramas de estados ou fluxogramas.

Apêndice A

O módulo `mvl9`

Neste apêndice descreve-se o módulo `mvl9` que representa os tipos de dados e funções definidas pela norma IEEE 1164. Este resultou da alteração do módulo norma, `STD_LOGIC_1164`, de forma a poder ser utilizado pelo simulador [Simulator 91b] e sintetizável pela ferramenta de síntese [Synthesis 91]. As alterações efectuadas foram:

- substituição das declarações de `alias`, devido a não serem suportadas pelo simulador, por declaração de variáveis inicializadas com valor definido por aquelas (ver exemplo da especificação da função `"and"` para o tipo de dados `std_logic_vector`),
- especificação do atributo `ENUM_ENCODING` no tipo de dados básico (`std_ulogic`) para indicar à ferramenta de síntese a correspondência entre os valores lógicos definidos e os reconhecidos por esta (`U`-dessconhecido, `D`-indiferença, `0`-zero lógico, `1`-um lógico, `Z`-alta impedância),
- colocação do comentário sintético `resolution_method` na especificação da função de resolução (`resolved`) para que esta seja reconhecida pela ferramenta de síntese como função de resolução do tipo *three-state*,
- colocação do comentário sintético `built_in` na redefinição das funções lógicas básicas e de conversão de tipos para indicar à ferramenta de síntese qual a operação realizada por cada uma destas funções,
- eliminação de código não sintetizável utilizando o par de comentários sintéticos `synthesis_off` e `synthesis_on`.

```
-- ------------------------------------------------------------------------
--
--    Title      :  std_logic_1164 multi-value logic system
--    Library    :  This package shall be compiled into a library
--               :  symbolically named IEEE.
--               :
--    Developers :  IEEE model standards group (par 1164)
--    Purpose    :  This packages defines a standard for designers
--               :  to use in describing the interconnection data types
--               :  used in vhdl modeling.
--               :
--    Limitation :  The logic system defined in this package may
--               :  be insufficient for modeling switched transistors,
--               :  since such a requirement is out of the scope of this
--               :  effort. Furthermore, mathematics, primitives,
--               :  timing standards, etc. are considered orthogonal
--               :  issues as it relates to this package and are therefore
--               :  beyond the scope of this effort.
--               :
--    Note       :  No declarations or definitions shall be included in,
--               :  or excluded from this package. The "package declaration"
--               :  defines the types, subtypes and declarations of
--               :  std_logic_1164. The std_logic_1164 package body shall be
--               :  considered the formal definition of the semantics of
--               :  this package. Tool developers may choose to implement
--               :  the package body in the most efficient manner available
--               :  to them.
--               :
-- ------------------------------------------------------------------------
--    modification history :
-- ------------------------------------------------------------------------
--  version | mod. date:|
--   v4.200 | 01/02/92  |
--   v4.200 | 30/Nov/92 | Added Synopsys synthesis comments and change alias
--                      | declaration to variable for Mentor simulator
--                      | (System-1076 and QuickSim). Package name changed
--                      | to 'mvl9'.       Paulo Flores (pff@inesc.pt)
-- ------------------------------------------------------------------------
PACKAGE mvl9 IS

    -- ------------------------------------------------------------------
    -- logic state system  (unresolved)
    -- ------------------------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
    attribute ENUM_ENCODING : STRING;
    attribute ENUM_ENCODING of std_ulogic: type is "U D 0 1 Z D 0 1 D";

    -- ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with the resolution function
    -- ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

    -- ------------------------------------------------------------------
    -- resolution function
    -- ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

    -- ------------------------------------------------------------------
    -- *** industry standard logic type ***
    -- ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    -- ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    -- ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

    -- ------------------------------------------------------------------
    -- common subtypes
    -- ------------------------------------------------------------------
    SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
    SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
    SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1'
```

```
    SUBTYPE UX01Z    IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')

    -------------------------------------------------------------------
    -- overloaded logical operators
    -------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
--  function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

    -------------------------------------------------------------------
    -- vectorized overloaded logical operators
    -------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

--  ------------------------------------------------------------------------
--  Note : The declaration and implementation of the "xnor" function is
--  specifically commented until at which time the VHDL language has been
--  officially adopted as containing such a function. At such a point,
--  the following comments may be removed along with this notice without
--  further "official" ballotting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  ------------------------------------------------------------------------
--  function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
--  function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;

    FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;

    -------------------------------------------------------------------
    -- conversion functions
    -------------------------------------------------------------------
    FUNCTION To_bit             ( s : std_ulogic
    --synopsys synthesis_off
                                  ; xmap : BIT := '0'
    --synopsys synthesis_on
                                                      ) RETURN BIT;
    FUNCTION To_bitvector       ( s : std_logic_vector
    --synopsys synthesis_off
                                  ; xmap : BIT := '0'
    --synopsys synthesis_on
                                                      ) RETURN BIT_VECTOR;
    FUNCTION To_bitvector       ( s : std_ulogic_vector
    --synopsys synthesis_off
                                  ; xmap : BIT := '0'
    --synopsys synthesis_on
                                                      ) RETURN BIT_VECTOR;
    FUNCTION To_StdULogic       ( b : BIT             ) RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR      ) RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR
) RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector
) RETURN std_ulogic_vector;

    -------------------------------------------------------------------
    -- strength strippers and type convertors
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
```

```
    FUNCTION To_X01  ( b : BIT                ) RETURN  X01;

    FUNCTION To_X01Z ( s : std_logic_vector   ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector  ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic         ) RETURN  X01Z;
    FUNCTION To_X01Z ( b : BIT_VECTOR         ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( b : BIT_VECTOR         ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( b : BIT                ) RETURN  X01Z;

    FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;

    -------------------------------------------------------------------
    -- edge detection
    -------------------------------------------------------------------
    --synopsys synthesis_off
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

    -------------------------------------------------------------------
    -- object contains an unknown
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;
    --synopsys synthesis_on
END mvl9;

--      Std_Logic_1164 Package Body
-- ----------------------------------------------------------------
--
--    Title     :  std_logic_1164 multi-value logic system
--    Library   :  This package shall be compiled into a library
--              :  symbolically named IEEE.
--              :
--    Developers:  IEEE model standards group (par 1164)
--    Purpose   :  This packages defines a standard for designers
--              :  to use in describing the interconnection data types
--              :  used in vhdl modeling.
--              :
--    Limitation:  The logic system defined in this package may
--              :  be insufficient for modeling switched transistors,
--              :  since such a requirement is out of the scope of this
--              :  effort. Furthermore, mathematics, primitives,
--              :  timing standards, etc. are considered orthogonal
--              :  issues as it relates to this package and are therefore
--              :  beyond the scope of this effort.
--              :
--    Note      :  No declarations or definitions shall be included in,
--              :  or excluded from this package. The "package declaration"
--              :  defines the types, subtypes and declarations of
--              :  std_logic_1164. The std_logic_1164 package body shall be
--              :  considered the formal definition of the semantics of
--              :  this package. Tool developers may choose to implement
--              :  the package body in the most efficient manner available
--              :  to them.
--              :
-- ----------------------------------------------------------------
--    modification history :
-- ----------------------------------------------------------------
--   version | mod. date:|
--    v4.200 | 01/02/91  |
--    v4.200 | 30/Nov/92 | Added Synopsys synthesis comments and change alias
--                       | declaration to variable for Mentor simulator
--                       | (System-1076 and QuickSim). Package name changed
--                       | to 'mvl9'.       Paulo Flores (pff@inesc.pt)
-- ----------------------------------------------------------------
PACKAGE BODY mvl9 IS

    -------------------------------------------------------------------
    -- local types
    -------------------------------------------------------------------
    --synopsys synthesis_off
    TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
    TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
```

```
-----------------------------------------------------------------
-- resolution function
-----------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -          |    |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );
--synopsys synthesis_on

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
-- pragma resolution_method three_state
--synopsys synthesis_off
    VARIABLE result : std_ulogic := 'Z';  -- weakest state default
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
    ELSE
        FOR i IN s'RANGE LOOP
            result := resolution_table(result, s(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END resolved;

-----------------------------------------------------------------
-- tables for logical operations
-----------------------------------------------------------------
--synopsys synthesis_off

-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -          |    |
--      ---------------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )  -- | - |
);

-- truth table for "or" function
CONSTANT or_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -          |    |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ), -- | U |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | 1 |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | Z |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | H |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )  -- | - |
);

-- truth table for "xor" function
```

```
CONSTANT xor_table : stdlogic_table := (
--      ----------------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -                  |   |
--      ----------------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ),  -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | 1 |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )   -- | - |
);

-- truth table for "not" function
CONSTANT not_table: stdlogic_1d :=
--      ---------------------------------------------
--      |  U    X    0    1    Z    W    L    H    - |
--      ---------------------------------------------
     ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );
 --synopsys synthesis_on

-------------------------------------------------------------------
-- overloaded logical operators ( with optimizing hints )
-------------------------------------------------------------------
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_AND
BEGIN
--synopsys synthesis_off
    RETURN (and_table(l, r));
--synopsys synthesis_on
END "and";

FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_NAND
BEGIN
--synopsys synthesis_off
    RETURN  (not_table ( and_table(l, r)));
--synopsys synthesis_on
END "nand";

FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_OR
BEGIN
--synopsys synthesis_off
    RETURN (or_table(l, r));
--synopsys synthesis_on
END "or";

FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_NOR
BEGIN
--synopsys synthesis_off
    RETURN  (not_table ( or_table( l, r )));
--synopsys synthesis_on
END "nor";

FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_XOR
BEGIN
--synopsys synthesis_off
    RETURN (xor_table(l, r));
--synopsys synthesis_on
END "xor";

--  function "xnor"  ( l : std_ulogic; r : std_ulogic ) return ux01 is
--  begin
--      return not_table(xor_table(l, r));
--  end "xnor";

FUNCTION "not"  ( l : std_ulogic ) RETURN UX01 IS
-- pragma built_in SYN_NOT
BEGIN
--synopsys synthesis_off
    RETURN (not_table(l));
--synopsys synthesis_on
END "not";

-------------------------------------------------------------------
-- and
```

```
-------------------------------------------------------------------------------
FUNCTION "and"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
-- pragma built_in SYN_AND
--synopsys synthesis_off
    VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
    -- ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
    VARIABLE rv : std_logic_vector ( 1 TO r'LENGTH ) := r;
    -- ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'and'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := and_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "and";
-------------------------------------------------------------------------------
FUNCTION "and"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
-- pragma built_in SYN_AND
--synopsys synthesis_off
    VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_ulogic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'and'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := and_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "and";
-------------------------------------------------------------------------------
-- nand
-------------------------------------------------------------------------------
FUNCTION "nand"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
-- pragma built_in SYN_NAND
--synopsys synthesis_off
    VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_logic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'nand'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := not_table(and_table (lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "nand";
-------------------------------------------------------------------------------
FUNCTION "nand"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
-- pragma built_in SYN_NAND
--synopsys synthesis_off
    VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_ulogic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
```

```
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'nand'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := not_table(and_table (lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "nand";
-------------------------------------------------------------------
-- or
-------------------------------------------------------------------
FUNCTION "or"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
-- pragma built_in SYN_OR
--synopsys synthesis_off
    VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_logic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'or'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := or_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "or";
-------------------------------------------------------------------
FUNCTION "or"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
-- pragma built_in SYN_OR
--synopsys synthesis_off
    VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_ulogic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'or'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := or_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "or";
-------------------------------------------------------------------
-- nor
-------------------------------------------------------------------
FUNCTION "nor"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
-- pragma built_in SYN_NOR
--synopsys synthesis_off
    VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_logic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments␣of␣overloaded␣'nor'␣operator␣are␣not␣of␣the␣same␣length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := not_table(or_table (lv(i), rv(i)));
```

```
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "nor";
-------------------------------------------------------------------------
FUNCTION "nor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
-- pragma built_in SYN_NOR
--synopsys synthesis_off
    VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_ulogic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments of overloaded 'nor' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := not_table(or_table (lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "nor";
-------------------------------------------------------------------------
-- xor
-------------------------------------------------------------------------
FUNCTION "xor"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
-- pragma built_in SYN_XOR
--synopsys synthesis_off
    VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_logic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments of overloaded 'xor' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := xor_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "xor";
-------------------------------------------------------------------------
FUNCTION "xor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
-- pragma built_in SYN_XOR
--synopsys synthesis_off
    VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
    VARIABLE rv : std_ulogic_vector ( 1 TO r'LENGTH ) := r;
    VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
        REPORT "arguments of overloaded 'xor' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := xor_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
--synopsys synthesis_on
END "xor";
-------------------------------------------------------------------------
-- xnor
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- Note : The declaration and implementation of the "xnor" function is
-- specifically commented until at which time the VHDL language has been
-- officially adopted as containing such a function. At such a point,
```

```
--  the following comments may be removed along with this notice without
--  further "official" ballotting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  ------------------------------------------------------------------------
--  function "xnor"  ( l,r : std_logic_vector ) return std_logic_vector is
--      alias lv : std_logic_vector ( 1 to l'length ) is l;
--      alias rv : std_logic_vector ( 1 to r'length ) is r;
--      variable result : std_logic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of the same length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";
--  ------------------------------------------------------------------------
--  function "xnor"  ( l,r : std_ulogic_vector ) return std_ulogic_vector is
--      alias lv : std_ulogic_vector ( 1 to l'length ) is l;
--      alias rv : std_ulogic_vector ( 1 to r'length ) is r;
--      variable result : std_ulogic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of the same length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";
    ------------------------------------------------------------------------
    -- not
    ------------------------------------------------------------------------
    FUNCTION "not"  ( l : std_logic_vector ) RETURN std_logic_vector IS
    -- pragma built_in SYN_NOT
    --synopsys synthesis_off
        VARIABLE lv : std_logic_vector ( 1 TO l'LENGTH ) := l;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH ) := (OTHERS => 'X');
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------------
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector IS
    -- pragma built_in SYN_NOT
    --synopsys synthesis_off
        VARIABLE lv : std_ulogic_vector ( 1 TO l'LENGTH ) := l;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH ) := (OTHERS => 'X');
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;

    ------------------------------------------------------------------------
    -- conversion tables
    ------------------------------------------------------------------------
    --synopsys synthesis_off
    TYPE logic_x01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF X01;
    TYPE logic_x01z_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF X01Z;
    TYPE logic_ux01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF UX01;
    ------------------------------------------------------------------------
```

```
-- table name : cvt_to_x01
--
-- parameters :
--         in   :  std_ulogic  -- some logic value
-- returns      :  x01         -- state value of logic value
-- purpose      :  to convert state-strength to state only
--
-- example      : if (cvt_to_x01 (input_signal) = '1' ) then ...
--
-------------------------------------------------------------------
CONSTANT cvt_to_x01 : logic_x01_table := (
                       'X',  -- 'U'
                       'X',  -- 'X'
                       '0',  -- '0'
                       '1',  -- '1'
                       'X',  -- 'Z'
                       'X',  -- 'W'
                       '0',  -- 'L'
                       '1',  -- 'H'
                       'X'   -- '-'
                      );
-------------------------------------------------------------------
-- table name : cvt_to_x01z
--
-- parameters :
--         in   :  std_ulogic  -- some logic value
-- returns      :  x01z        -- state value of logic value
-- purpose      :  to convert state-strength to state only
--
-- example      : if (cvt_to_x01z (input_signal) = '1' ) then ...
--
-------------------------------------------------------------------
CONSTANT cvt_to_x01z : logic_x01z_table := (
                       'X',  -- 'U'
                       'X',  -- 'X'
                       '0',  -- '0'
                       '1',  -- '1'
                       'Z',  -- 'Z'
                       'X',  -- 'W'
                       '0',  -- 'L'
                       '1',  -- 'H'
                       'X'   -- '-'
                      );
-------------------------------------------------------------------
-- table name : cvt_to_ux01
--
-- parameters :
--         in   :  std_ulogic  -- some logic value
-- returns      :  ux01        -- state value of logic value
-- purpose      :  to convert state-strength to state only
--
-- example      : if (cvt_to_ux01 (input_signal) = '1' ) then ...
--
-------------------------------------------------------------------
CONSTANT cvt_to_ux01 : logic_ux01_table := (
                       'U',  -- 'U'
                       'X',  -- 'X'
                       '0',  -- '0'
                       '1',  -- '1'
                       'X',  -- 'Z'
                       'X',  -- 'W'
                       '0',  -- 'L'
                       '1',  -- 'H'
                       'X'   -- '-'
                      );
--synopsys synthesis_on

-------------------------------------------------------------------
-- conversion functions
-------------------------------------------------------------------
FUNCTION To_bit       ( s : std_ulogic
--synopsys synthesis_off
    ;xmap : BIT := '0'
--synopsys synthesis_on
    ) RETURN BIT IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
        CASE s IS
            WHEN '0' | 'L' => RETURN ('0');
            WHEN '1' | 'H' => RETURN ('1');
```

```
            WHEN OTHERS => RETURN xmap;
        END CASE;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_logic_vector
    --synopsys synthesis_off
        ; xmap : BIT := '0'
    --synopsys synthesis_on
        ) RETURN BIT_VECTOR IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) := s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_ulogic_vector
    --synopsys synthesis_off
        ; xmap : BIT := '0'
    --synopsys synthesis_on
        ) RETURN BIT_VECTOR IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) := s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------
    FUNCTION To_StdULogic       ( b : BIT                ) RETURN std_ulogic IS
    -- pragma built_in SYN_FEED_THRU
    BEGIN
    --synopsys synthesis_off
        CASE b IS
            WHEN '0' => RETURN '0';
            WHEN '1' => RETURN '1';
        END CASE;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR
) RETURN std_logic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) := b;
        VARIABLE result : std_logic_vector ( b'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
```

```
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) := s;
        VARIABLE result : std_logic_vector ( s'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR
) RETURN std_ulogic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) := b;
        VARIABLE result : std_ulogic_vector ( b'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( s : std_logic_vector ) RETURN std_ulogic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) := s;
        VARIABLE result : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;

    ------------------------------------------------------------------------
    -- strength strippers and type convertors
    ------------------------------------------------------------------------
    -- to_x01
    ------------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_logic_vector ( 1 TO s'LENGTH ) := s;
        VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
    --synopsys synthesis_on
    BEGIN
    --synopsys synthesis_off
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01 (sv(i));
        END LOOP;
        RETURN result;
    --synopsys synthesis_on
    END;
    ------------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
    -- pragma built_in SYN_FEED_THRU
    --synopsys synthesis_off
        VARIABLE sv : std_ulogic_vector ( 1 TO s'LENGTH ) := s;
        VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
    --synopsys synthesis_on
    BEGIN
```

```
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01 (sv(i));
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01  ( s : std_ulogic ) RETURN  X01 IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
    RETURN (cvt_to_x01(s));
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01  ( b : BIT ) RETURN  X01 IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
--synopsys synthesis_on
END;

-------------------------------------------------------------------------------
-- to_x01z
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_logic_vector ) RETURN  std_logic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE sv : std_logic_vector ( 1 TO s'LENGTH ) := s;
    VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
--synopsys synthesis_on
```

```
END;
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE sv : std_ulogic_vector ( 1 TO s'LENGTH ) := s;
    VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_ulogic ) RETURN  X01Z IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
    RETURN (cvt_to_x01z(s));
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT ) RETURN  X01Z IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------------
-- to_ux01
-------------------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE sv : std_logic_vector ( 1 TO s'LENGTH ) := s;
    VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
```

```
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE sv : std_ulogic_vector ( 1 TO s'LENGTH ) := s;
    VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_ulogic ) RETURN  UX01 IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
    RETURN (cvt_to_ux01(s));
--synopsys synthesis_on
END;
-------------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
-- pragma built_in SYN_FEED_THRU
--synopsys synthesis_off
    VARIABLE bv : BIT_VECTOR ( 1 TO b'LENGTH ) := b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
--synopsys synthesis_on
BEGIN
--synopsys synthesis_off
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
--synopsys synthesis_on
END;
-------------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT ) RETURN  UX01 IS
-- pragma built_in SYN_FEED_THRU
BEGIN
--synopsys synthesis_off
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
--synopsys synthesis_on
END;
```

```
    ------------------------------------------------------------------
    -- edge detection
    ------------------------------------------------------------------
    --synopsys synthesis_off
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN (s'EVENT AND (To_X01(s) = '1') AND
                            (To_X01(s'LAST_VALUE) = '0'));
    END;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN (s'EVENT AND (To_X01(s) = '0') AND
                            (To_X01(s'LAST_VALUE) = '1'));
    END;
    ------------------------------------------------------------------
    -- object contains an unknown
    ------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    ------------------------------------------------------------------
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    ------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN IS
    BEGIN
        CASE s IS
            WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
            WHEN OTHERS => NULL;
        END CASE;
        RETURN FALSE;
    END;
    --synopsys synthesis_on

END mvl9;
```

Apêndice B

O módulo `arithmetic`

Neste apêndice descreve-se o módulo `arithmetic` fornecido com a ferramenta de síntese [Synthesis 91] para auxiliar o projectista a descrever circuitos sintetizáveis. Neste módulo são definidos tipos de dados que permitem representar números sem sinal (`UNSIGNED`) e números inteiros (`SIGNED`), na representação de complemento para dois, como vectores de *bits*. São ainda definidos operadores aritméticos, relacionais e funções de conversão para esses tipos de dados.

Na especificação do corpo deste módulo são utilizados frequentemente comentários sintéticos que permitem informar a ferramenta de síntese sobre:

- quais os tipos de operações realizadas por certas funções e como estas devem de ser mapeadas em "blocos de lógica" pré-definidos na ferramenta,
- como podem ser partilhados os vários operadores definidos,
- quais as partes de código que não devem de ser sintetizadas, mas que são descritas apenas para que este módulo possa ser usado por um simulador VHDL.

```
-- ------------------------------------------------------------------ --
-- Copyright (c) 1990, 1991 by Synopsys, Inc.  All rights reserved.   --
-- ------------------------------------------------------------------ --
-- This source file may be used and distributed without restriction   --
-- provided that this copyright statement is not removed from the file --
-- and that any derivative work contains this copyright notice.       --
-- ------------------------------------------------------------------ --

package arithmetic is

    type UNSIGNED is array (INTEGER range <>) of BIT;
    type SIGNED is array (INTEGER range <>) of BIT;
    subtype SMALL_INT is INTEGER range 0 to 1;

    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: INTEGER) return SIGNED;
    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: BIT) return UNSIGNED;
    function "+"(L: BIT; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: BIT) return SIGNED;
    function "+"(L: BIT; R: SIGNED) return SIGNED;

    function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: INTEGER) return SIGNED;
    function "-"(L: INTEGER; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: BIT) return UNSIGNED;
    function "-"(L: BIT; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: BIT) return SIGNED;
    function "-"(L: BIT; R: SIGNED) return SIGNED;

    function "+"(L: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED) return SIGNED;
    function "-"(L: SIGNED) return SIGNED;
    function "ABS"(L: SIGNED) return SIGNED;

    function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "*"(L: SIGNED; R: SIGNED) return SIGNED;
    function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;

    function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
    function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;

    function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
    function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;

    function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
    function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
```

```
    function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;

    function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
    function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;

    function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
    function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: SIGNED) return BOOLEAN;

    function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
    function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
    function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
    function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
    function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
    function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;

    function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;

    function CONV_INTEGER(ARG: INTEGER) return INTEGER;
    function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
    function CONV_INTEGER(ARG: SIGNED) return INTEGER;
    function CONV_INTEGER(ARG: BIT) return SMALL_INT;
    function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: BIT; SIZE: INTEGER) return UNSIGNED;
    function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: BIT; SIZE: INTEGER) return SIGNED;

    function AND_REDUCE(ARG: BIT_VECTOR) return BIT;
    function NAND_REDUCE(ARG: BIT_VECTOR) return BIT;
    function OR_REDUCE(ARG: BIT_VECTOR) return BIT;
    function NOR_REDUCE(ARG: BIT_VECTOR) return BIT;
    function XOR_REDUCE(ARG: BIT_VECTOR) return BIT;
    function XNOR_REDUCE(ARG: BIT_VECTOR) return BIT;

end arithmetic;

package body arithmetic is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    -- synopsys synthesis_off
    type tbl_type is array (BIT) of BIT;
```

```
constant tbl_BINARY : tbl_type :=
    ('0', '1');
-- synopsys synthesis_on

function MAKE_BINARY(A : BIT) return BIT is
    -- synopsys built_in SYN_FEED_THRU
begin
    -- synopsys synthesis_off
        return tbl_BINARY(A);
    -- synopsys synthesis_on
end;

function MAKE_BINARY(A : UNSIGNED) return UNSIGNED is
    -- synopsys built_in SYN_FEED_THRU
    variable one_bit : BIT;
    variable result : UNSIGNED (A'range);
begin
    -- synopsys synthesis_off
        for i in A'range loop
            result(i) := tbl_BINARY(A(i));
        end loop;
        return result;
    -- synopsys synthesis_on
end;

function MAKE_BINARY(A : UNSIGNED) return SIGNED is
    -- synopsys built_in SYN_FEED_THRU
    variable one_bit : BIT;
    variable result : SIGNED (A'range);
begin
    -- synopsys synthesis_off
        for i in A'range loop
            result(i) := tbl_BINARY(A(i));
        end loop;
        return result;
    -- synopsys synthesis_on
end;

function MAKE_BINARY(A : SIGNED) return UNSIGNED is
    -- synopsys built_in SYN_FEED_THRU
    variable one_bit : BIT;
    variable result : UNSIGNED (A'range);
begin
    -- synopsys synthesis_off
        for i in A'range loop
            result(i) := tbl_BINARY(A(i));
        end loop;
        return result;
    -- synopsys synthesis_on
end;

function MAKE_BINARY(A : SIGNED) return SIGNED is
    -- synopsys built_in SYN_FEED_THRU
    variable one_bit : BIT;
    variable result : SIGNED (A'range);
begin
    -- synopsys synthesis_off
        for i in A'range loop
            result(i) := tbl_BINARY(A(i));
        end loop;
        return result;
    -- synopsys synthesis_on
end;


-- Type propagation function which returns a signed type with the
-- size of the left arg.
function LEFT_SIGNED_ARG(A,B: SIGNED) return SIGNED is
  variable Z: SIGNED (A'left downto 0);
  -- pragma return_port_name Z
begin
  return(Z);
end;

-- Type propagation function which returns an unsigned type with the
-- size of the left arg.
function LEFT_UNSIGNED_ARG(A,B: UNSIGNED) return UNSIGNED is
  variable Z: UNSIGNED (A'left downto 0);
  -- pragma return_port_name Z
begin
  return(Z);
```

```
end;
-- Type propagation function which returns a signed type with the
-- size of the result of a signed multiplication
function MULT_SIGNED_ARG(A,B: SIGNED) return SIGNED is
  variable Z: SIGNED ((A'length+B'length-1) downto 0);
  -- pragma return_port_name Z
begin
  return(Z);
end;
-- Type propagation function which returns an unsigned type with the
-- size of the result of a unsigned multiplication
function MULT_UNSIGNED_ARG(A,B: UNSIGNED) return UNSIGNED is
  variable Z: UNSIGNED ((A'length+B'length-1) downto 0);
  -- pragma return_port_name Z
begin
  return(Z);
end;
function mult(A,B: SIGNED) return SIGNED is

  variable BA: SIGNED((A'length+B'length-1) downto 0);
  variable PA: SIGNED((A'length+B'length-1) downto 0);
  variable AA: SIGNED(A'length downto 0);
  variable neg: BIT;
  constant one : UNSIGNED(1 downto 0) := ('0', '1');

  -- pragma map_to_operator MULT_TC_OP
  -- pragma type_function MULT_SIGNED_ARG
  -- pragma return_port_name Z

  begin
    PA := (others => '0');
    neg := B(B'left) xor A(A'left);
    BA := CONV_SIGNED(('0' & ABS(B)),(A'length+B'length));
    AA := '0' & ABS(A);
    for i in 0 to A'length-1 loop
      if AA(i) = '1' then
        PA := PA+BA;
      end if;
      BA := SHL(BA,one);
    end loop;
    if (neg= '1') then
      return(-PA);
    else
      return(PA);
    end if;
  end;
function mult(A,B: UNSIGNED) return UNSIGNED is

  variable BA: UNSIGNED((A'length+B'length-1) downto 0);
  variable PA: UNSIGNED((A'length+B'length-1) downto 0);
  constant one : UNSIGNED(1 downto 0) := "01";

  -- pragma map_to_operator MULT_UNS_OP
  -- pragma type_function MULT_UNSIGNED_ARG
  -- pragma return_port_name Z

  begin
    PA := (others => '0');
    BA := CONV_UNSIGNED(B,(A'length+B'length));
    for i in 0 to A'length-1 loop
      if A(i) = '1' then
        PA := PA+BA;
      end if;
      BA := SHL(BA,one);
    end loop;
    return(PA);
  end;
-- subtract two signed numbers of the same length
-- both arrays must have range (msb downto 0)
function minus(A, B: SIGNED) return SIGNED is
    variable carry: BIT;
    variable BV: BIT_VECTOR (A'left downto 0);
    variable sum: SIGNED (A'left downto 0);

    -- pragma map_to_operator SUB_TC_OP
```

```
    -- pragma type_function LEFT_SIGNED_ARG
    -- pragma return_port_name Z
begin
    carry := '1';
    BV := not BIT_VECTOR(B);

    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
                 (A(i) and carry) or
                 (carry and BV(i));
    end loop;
    return sum;
end;

-- add two signed numbers of the same length
-- both arrays must have range (msb downto 0)
function plus(A, B: SIGNED) return SIGNED is
    variable carry: BIT;
    variable BV, sum: SIGNED (A'left downto 0);

    -- pragma map_to_operator ADD_TC_OP
    -- pragma type_function LEFT_SIGNED_ARG
    -- pragma return_port_name Z
begin
    carry := '0';
    BV := B;

    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
                 (A(i) and carry) or
                 (carry and BV(i));
    end loop;
    return sum;
end;

-- subtract two unsigned numbers of the same length
-- both arrays must have range (msb downto 0)
function unsigned_minus(A, B: UNSIGNED) return UNSIGNED is
    variable carry: BIT;
    variable BV: BIT_VECTOR (A'left downto 0);
    variable sum: UNSIGNED (A'left downto 0);

    -- pragma map_to_operator SUB_UNS_OP
    -- pragma type_function LEFT_UNSIGNED_ARG
    -- pragma return_port_name Z
begin
    carry := '1';
    BV := not BIT_VECTOR(B);

    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
                 (A(i) and carry) or
                 (carry and BV(i));
    end loop;
    return sum;
end;

-- add two unsigned numbers of the same length
-- both arrays must have range (msb downto 0)
function unsigned_plus(A, B: UNSIGNED) return UNSIGNED is
    variable carry: BIT;
    variable BV, sum: UNSIGNED (A'left downto 0);

    -- pragma map_to_operator ADD_UNS_OP
    -- pragma type_function LEFT_UNSIGNED_ARG
    -- pragma return_port_name Z
begin
    carry := '0';
    BV := B;

    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
                 (A(i) and carry) or
                 (carry and BV(i));
```

```
    end loop;
    return sum;
end;

function "*"(L: SIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to mult
begin
      return      mult(CONV_SIGNED(L, L'length),
                       CONV_SIGNED(R, R'length)); -- pragma label mult
end;

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to mult
begin
      return    mult(CONV_UNSIGNED(L, L'length),
                     CONV_UNSIGNED(R, R'length)); -- pragma label mult
end;

function "*"(L: UNSIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to plus
begin
    return        mult(CONV_SIGNED(L, L'length+1),
                       CONV_SIGNED(R, R'length)); -- pragma label mult
end;

function "*"(L: SIGNED; R: UNSIGNED) return SIGNED is
    -- pragma label_applies_to plus
begin
    return       mult(CONV_SIGNED(L, L'length),
                      CONV_SIGNED(R, R'length+1)); -- pragma label mult
end;

function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_plus(CONV_UNSIGNED(L, length),
                         CONV_UNSIGNED(R, length)); -- pragma label plus
end;

function "+"(L: SIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := max(L'length, R'length);
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: UNSIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: SIGNED; R: UNSIGNED) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := L'length + 1;
begin
    return CONV_UNSIGNED(
             plus( -- pragma label plus
                 CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)),
             length-1);
end;

function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to plus
```

```
    constant length: INTEGER := R'length + 1;
begin
    return CONV_UNSIGNED(
            plus( -- pragma label plus
                CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)),
            length-1);
end;

function "+"(L: SIGNED; R: INTEGER) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := L'length;
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: INTEGER; R: SIGNED) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := R'length;
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: UNSIGNED; R: BIT) return UNSIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := L'length;
begin
    return unsigned_plus(CONV_UNSIGNED(L, length),
                 CONV_UNSIGNED(R, length)) ; -- pragma label plus
end;

function "+"(L: BIT; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := R'length;
begin
    return unsigned_plus(CONV_UNSIGNED(L, length),
                 CONV_UNSIGNED(R, length)); -- pragma label plus
end;

function "+"(L: SIGNED; R: BIT) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := L'length;
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;

function "+"(L: BIT; R: SIGNED) return SIGNED is
    -- pragma label_applies_to plus
    constant length: INTEGER := R'length;
begin
    return plus(CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)); -- pragma label plus
end;



function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_minus(CONV_UNSIGNED(L, length),
                          CONV_UNSIGNED(R, length)); -- pragma label minus
end;

function "-"(L: SIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := max(L'length, R'length);
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;
```

```
function "-"(L: UNSIGNED; R: SIGNED) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;

function "-"(L: SIGNED; R: UNSIGNED) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;

function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := L'length + 1;
begin
    return CONV_UNSIGNED(
            minus( -- pragma label minus
                CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)),
            length-1);
end;

function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := R'length + 1;
begin
    return CONV_UNSIGNED(
            minus( -- pragma label minus
                CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)),
            length-1);
end;

function "-"(L: SIGNED; R: INTEGER) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := L'length;
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;

function "-"(L: INTEGER; R: SIGNED) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := R'length;
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;

function "-"(L: UNSIGNED; R: BIT) return UNSIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := L'length + 1;
begin
    return CONV_UNSIGNED(
            minus( -- pragma label minus
                CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)),
            length-1);
end;

function "-"(L: BIT; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := R'length + 1;
begin
    return CONV_UNSIGNED(
            minus( -- pragma label minus
                CONV_SIGNED(L, length),
                CONV_SIGNED(R, length)),
            length-1);
end;
```

```
function "-"(L: SIGNED; R: BIT) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := L'length;
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;

function "-"(L: BIT; R: SIGNED) return SIGNED is
    -- pragma label_applies_to minus
    constant length: INTEGER := R'length;
begin
    return minus(CONV_SIGNED(L, length),
                 CONV_SIGNED(R, length)); -- pragma label minus
end;


function "+"(L: UNSIGNED) return UNSIGNED is
begin
    return L;
end;

function "+"(L: SIGNED) return SIGNED is
begin
    return L;
end;

function "-"(L: SIGNED) return SIGNED is
    -- pragma label_applies_to minus
begin
    return 0 - L; -- pragma label minus
end;

function "ABS"(L: SIGNED) return SIGNED is
begin
    if L(L'left) = '0' then
        return L;
    else
        return 0 - L;
    end if;
end;

-- Type propagation function which returns the type BOOLEAN
function UNSIGNED_RETURN_BOOLEAN(A,B: UNSIGNED) return BOOLEAN is
  variable Z: BOOLEAN;
  -- pragma return_port_name Z
begin
  return(Z);
end;
-- Type propagation function which returns the type BOOLEAN
function SIGNED_RETURN_BOOLEAN(A,B: SIGNED) return BOOLEAN is
  variable Z: BOOLEAN;
  -- pragma return_port_name Z
begin
  return(Z);
end;

-- compare two signed numbers of the same length
-- both arrays must have range (msb downto 0)
function is_less(A, B: SIGNED) return BOOLEAN is
    constant sign: INTEGER := A'left;
    variable a_is_0, b_is_1, result : boolean;
    -- pragma map_to_operator LT_TC_OP
    -- pragma type_function SIGNED_RETURN_BOOLEAN
    -- pragma return_port_name Z
begin
    if A(sign) /= B(sign) then
        result := A(sign) = '1';
    else
        result := FALSE;
        for i in 0 to sign-1 loop
            a_is_0 := A(i) = '0';
            b_is_1 := B(i) = '1';
```

```
            result := (a_is_0 and b_is_1) or
                      (a_is_0 and result) or
                      (b_is_1 and result);
        end loop;
    end if;
    return result;
end;

-- compare two signed numbers of the same length
-- both arrays must have range (msb downto 0)
function is_less_or_equal(A, B: SIGNED) return BOOLEAN is
    constant sign: INTEGER := A'left;
    variable a_is_0, b_is_1, result : boolean;

    -- pragma map_to_operator LEQ_TC_OP
    -- pragma type_function SIGNED_RETURN_BOOLEAN
    -- pragma return_port_name Z

begin
    if A(sign) /= B(sign) then
        result := A(sign) = '1';
    else
        result := TRUE;
        for i in 0 to sign-1 loop
            a_is_0 := A(i) = '0';
            b_is_1 := B(i) = '1';
            result := (a_is_0 and b_is_1) or
                      (a_is_0 and result) or
                      (b_is_1 and result);
        end loop;
    end if;
    return result;
end;


-- compare two unsigned numbers of the same length
-- both arrays must have range (msb downto 0)
function unsigned_is_less(A, B: UNSIGNED) return BOOLEAN is
    constant sign: INTEGER := A'left;
    variable a_is_0, b_is_1, result : boolean;

    -- pragma map_to_operator LT_UNS_OP
    -- pragma type_function UNSIGNED_RETURN_BOOLEAN
    -- pragma return_port_name Z

begin
    result := FALSE;
    for i in 0 to sign loop
        a_is_0 := A(i) = '0';
        b_is_1 := B(i) = '1';
        result := (a_is_0 and b_is_1) or
                  (a_is_0 and result) or
                  (b_is_1 and result);
    end loop;
    return result;
end;

-- compare two unsigned numbers of the same length
-- both arrays must have range (msb downto 0)
function unsigned_is_less_or_equal(A, B: UNSIGNED) return BOOLEAN is
    constant sign: INTEGER := A'left;
    variable a_is_0, b_is_1, result : boolean;

    -- pragma map_to_operator LEQ_UNS_OP
    -- pragma type_function UNSIGNED_RETURN_BOOLEAN
    -- pragma return_port_name Z

begin
    result := TRUE;
    for i in 0 to sign loop
        a_is_0 := A(i) = '0';
        b_is_1 := B(i) = '1';
        result := (a_is_0 and b_is_1) or
                  (a_is_0 and result) or
                  (b_is_1 and result);
    end loop;
    return result;
end;
```

```
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_is_less(CONV_UNSIGNED(L, length),
                            CONV_UNSIGNED(R, length)); -- pragma label lt
end;

function "<"(L: SIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := max(L'length, R'length);
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := L'length + 1;
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := R'length + 1;
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: SIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := L'length;
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;

function "<"(L: INTEGER; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to lt
    constant length: INTEGER := R'length;
begin
    return is_less(CONV_SIGNED(L, length),
                   CONV_SIGNED(R, length)); -- pragma label lt
end;


function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_is_less_or_equal(CONV_UNSIGNED(L, length),
                          CONV_UNSIGNED(R, length)); -- pragma label leq
end;

function "<="(L: SIGNED; R: SIGNED) return BOOLEAN is
```

```
    -- pragma label_applies_to leq
    constant length: INTEGER := max(L'length, R'length);
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := L'length + 1;
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := R'length + 1;
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: SIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := L'length;
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;

function "<="(L: INTEGER; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to leq
    constant length: INTEGER := R'length;
begin
    return is_less_or_equal(CONV_SIGNED(L, length),
                            CONV_SIGNED(R, length)); -- pragma label leq
end;


function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_is_less(CONV_UNSIGNED(R, length),
                            CONV_UNSIGNED(L, length)); -- pragma label gt
end;

function ">"(L: SIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := max(L'length, R'length);
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := max(L'length + 1, R'length);
```

```
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := L'length + 1;
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := R'length + 1;
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: SIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := L'length;
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;

function ">"(L: INTEGER; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to gt
    constant length: INTEGER := R'length;
begin
    return is_less(CONV_SIGNED(R, length),
                   CONV_SIGNED(L, length)); -- pragma label gt
end;


function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := max(L'length, R'length);
begin
    return unsigned_is_less_or_equal(CONV_UNSIGNED(R, length),
                                     CONV_UNSIGNED(L, length)); -- pragma label geq
end;

function ">="(L: SIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := max(L'length, R'length);
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                            CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                            CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
```

```
                                CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := L'length + 1;
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                                CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := R'length + 1;
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                                CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: SIGNED; R: INTEGER) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := L'length;
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                                CONV_SIGNED(L, length)); -- pragma label geq
end;

function ">="(L: INTEGER; R: SIGNED) return BOOLEAN is
    -- pragma label_applies_to geq
    constant length: INTEGER := R'length;
begin
    return is_less_or_equal(CONV_SIGNED(R, length),
                                CONV_SIGNED(L, length)); -- pragma label geq
end;


-- for internal use only.  Assumes SIGNED arguments of equal length.
function bitwise_eql(L: BIT_VECTOR; R: BIT_VECTOR)
                                                    return BOOLEAN is
    -- pragma built_in SYN_EQL
begin
    for i in L'range loop
        if L(i) /= R(i) then
            return FALSE;
        end if;
    end loop;
    return TRUE;
end;
-- for internal use only.  Assumes SIGNED arguments of equal length.
function bitwise_neq(L: BIT_VECTOR; R: BIT_VECTOR)
                                                    return BOOLEAN is
    -- pragma built_in SYN_NEQ
begin
    for i in L'range loop
        if L(i) /= R(i) then
            return TRUE;
        end if;
    end loop;
    return FALSE;
end;

function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length);
begin
    return bitwise_eql( BIT_VECTOR( CONV_UNSIGNED(L, length) ),
            BIT_VECTOR( CONV_UNSIGNED(R, length) ) );
end;

function "="(L: SIGNED; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length);
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;
```

```
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    constant length: INTEGER := L'length + 1;
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := R'length + 1;
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "="(L: SIGNED; R: INTEGER) return BOOLEAN is
    constant length: INTEGER := L'length;
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "="(L: INTEGER; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := R'length;
begin
    return bitwise_eql( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;


function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length);
begin
    return bitwise_neq( BIT_VECTOR( CONV_UNSIGNED(L, length) ),
            BIT_VECTOR( CONV_UNSIGNED(R, length) ) );
end;

function "/="(L: SIGNED; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length);
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length + 1, R'length);
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := max(L'length, R'length + 1);
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN is
    constant length: INTEGER := L'length + 1;
begin
```

```
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN is
    constant length: INTEGER := R'length + 1;
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: SIGNED; R: INTEGER) return BOOLEAN is
    constant length: INTEGER := L'length;
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;

function "/="(L: INTEGER; R: SIGNED) return BOOLEAN is
    constant length: INTEGER := R'length;
begin
    return bitwise_neq( BIT_VECTOR( CONV_SIGNED(L, length) ),
            BIT_VECTOR( CONV_SIGNED(R, length) ) );
end;


function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED is
    constant control_msb: INTEGER := COUNT'length - 1;
    variable control: UNSIGNED (control_msb downto 0);
    constant result_msb: INTEGER := ARG'length-1;
    subtype rtype is UNSIGNED (result_msb downto 0);
    variable result, temp: rtype;
begin
    control := MAKE_BINARY(COUNT);
    result := ARG;
    for i in 0 to control_msb loop
        if control(i) = '1' then
            temp := rtype'(others => '0');
            if 2**i < result_msb then
                temp(result_msb downto 2**i) :=
                                result(result_msb - 2**i downto 0);
            end if;
            result := temp;
        end if;
    end loop;
    return result;
end;

function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED is
    constant control_msb: INTEGER := COUNT'length - 1;
    variable control: UNSIGNED (control_msb downto 0);
    constant result_msb: INTEGER := ARG'length-1;
    subtype rtype is SIGNED (result_msb downto 0);
    variable result, temp: rtype;
begin
    control := MAKE_BINARY(COUNT);
    result := ARG;
    for i in 0 to control_msb loop
        if control(i) = '1' then
            temp := rtype'(others => '0');
            if 2**i < result_msb then
                temp(result_msb downto 2**i) :=
                                result(result_msb - 2**i downto 0);
            end if;
            result := temp;
        end if;
    end loop;
    return result;
end;

function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED is
    constant control_msb: INTEGER := COUNT'length - 1;
    variable control: UNSIGNED (control_msb downto 0);
    constant result_msb: INTEGER := ARG'length-1;
    subtype rtype is UNSIGNED (result_msb downto 0);
    variable result, temp: rtype;
begin
```

```
    control := MAKE_BINARY(COUNT);
    result := ARG;
    for i in 0 to control_msb loop
        if control(i) = '1' then
            temp := rtype'(others => '0');
            if 2**i < result_msb then
                temp(result_msb - 2**i downto 0) :=
                                    result(result_msb downto 2**i);
            end if;
            result := temp;
        end if;
    end loop;
    return result;
end;

function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED is
    constant control_msb: INTEGER := COUNT'length - 1;
    variable control: UNSIGNED (control_msb downto 0);
    constant result_msb: INTEGER := ARG'length-1;
    subtype rtype is SIGNED (result_msb downto 0);
    variable result, temp: rtype;
    variable sign_bit: BIT;
begin
    control := MAKE_BINARY(COUNT);
    result := ARG;
    sign_bit := ARG(ARG'left);
    for i in 0 to control_msb loop
        if control(i) = '1' then
            temp := rtype'(others => sign_bit);
            if 2**i < result_msb then
                temp(result_msb - 2**i downto 0) :=
                                    result(result_msb downto 2**i);
            end if;
            result := temp;
        end if;
    end loop;
    return result;
end;


function CONV_INTEGER(ARG: INTEGER) return INTEGER is
begin
    return ARG;
end;

function CONV_INTEGER(ARG: UNSIGNED) return INTEGER is
    variable result: INTEGER;
    -- synopsys built_in SYN_UNSIGNED_TO_INTEGER;
begin
    -- synopsys synthesis_off
    assert ARG'length <= 31
        report "ARG is too large in CONV_INTEGER"
        severity FAILURE;
    result := 0;
    for i in ARG'range loop
        result := result * 2;
        if tbl_BINARY(ARG(i)) = '1' then
            result := result + 1;
        end if;
    end loop;
    return result;
    -- synopsys synthesis_on
end;

function CONV_INTEGER(ARG: SIGNED) return INTEGER is
    variable result: INTEGER;
    -- synopsys built_in SYN_SIGNED_TO_INTEGER;
begin
    -- synopsys synthesis_off
    assert ARG'length <= 32
        report "ARG is too large in CONV_INTEGER"
        severity FAILURE;
    result := 0;
    for i in ARG'range loop
        if i /= ARG'left then
            result := result * 2;
            if tbl_BINARY(ARG(i)) = '1' then
                result := result + 1;
            end if;
```

```
        end if;
    end loop;
    if tbl_BINARY(ARG(ARG'left)) = '1' then
        if ARG'length = 32 then
            result := (result - 2**30) - 2**30;
        else
            result := result - (2 ** (ARG'length-1));
        end if;
    end if;
    return result;
    -- synopsys synthesis_on
end;

function CONV_INTEGER(ARG: BIT) return SMALL_INT is
begin
    if MAKE_BINARY(ARG) = '0' then
        return 0;
    else
        return 1;
    end if;
end;

-- convert an integer to a unsigned BIT_VECTOR
function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED is
    variable result: UNSIGNED(SIZE-1 downto 0);
    variable temp: integer;
    -- synopsys built_in SYN_INTEGER_TO_UNSIGNED
begin
    -- synopsys synthesis_off
    temp := ARG;
    for i in 0 to SIZE-1 loop
        if (temp mod 2) = 1 then
            result(i) := '1';
        else
            result(i) := '0';
        end if;
        if temp > 0 then
            temp := temp / 2;
        else
            temp := (temp - 1) / 2; -- simulate ASR
        end if;
    end loop;
    return result;
    -- synopsys synthesis_on
end;

function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED is
    constant msb: INTEGER := min(ARG'length, SIZE) - 1;
    subtype rtype is UNSIGNED (SIZE-1 downto 0);
    variable new_bounds: UNSIGNED (ARG'length-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_ZERO_EXTEND
begin
    -- synopsys synthesis_off
    result := rtype'(others => '0');
    new_bounds := MAKE_BINARY(ARG);
    result(msb downto 0) := new_bounds(msb downto 0);
    return result;
    -- synopsys synthesis_on
end;

function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED is
    constant msb: INTEGER := min(ARG'length, SIZE) - 1;
    subtype rtype is UNSIGNED (SIZE-1 downto 0);
    variable new_bounds: UNSIGNED (ARG'length-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_SIGN_EXTEND
begin
    -- synopsys synthesis_off
    new_bounds := MAKE_BINARY(ARG);
    result := rtype'(others => new_bounds(new_bounds'left));
    result(msb downto 0) := new_bounds(msb downto 0);
    return result;
    -- synopsys synthesis_on
end;

function CONV_UNSIGNED(ARG: BIT; SIZE: INTEGER) return UNSIGNED is
```

```
    subtype rtype is UNSIGNED (SIZE-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_ZERO_EXTEND
begin
    -- synopsys synthesis_off
    result := rtype'(others => '0');
    result(0) := MAKE_BINARY(ARG);
    return result;
    -- synopsys synthesis_on
end;

-- convert an integer to a 2's complement BIT_VECTOR
function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED is
    variable result: SIGNED (SIZE-1 downto 0);
    variable temp: integer;
    -- synopsys built_in SYN_INTEGER_TO_SIGNED
begin
    -- synopsys synthesis_off
    temp := ARG;
    for i in 0 to SIZE-1 loop
        if (temp mod 2) = 1 then
            result(i) := '1';
        else
            result(i) := '0';
        end if;
        if temp > 0 then
            temp := temp / 2;
        else
            temp := (temp - 1) / 2; -- simulate ASR
        end if;
    end loop;
    return result;
    -- synopsys synthesis_on
end;

function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED is
    constant msb: INTEGER := min(ARG'length, SIZE) - 1;
    subtype rtype is SIGNED (SIZE-1 downto 0);
    variable new_bounds : SIGNED (ARG'length-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_ZERO_EXTEND
begin
    -- synopsys synthesis_off
    result := rtype'(others => '0');
    new_bounds := MAKE_BINARY(ARG);
    result(msb downto 0) := new_bounds(msb downto 0);
    return result;
    -- synopsys synthesis_on
end;
function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED is
    constant msb: INTEGER := min(ARG'length, SIZE) - 1;
    subtype rtype is SIGNED (SIZE-1 downto 0);
    variable new_bounds : SIGNED (ARG'length-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_SIGN_EXTEND
begin
    -- synopsys synthesis_off
    new_bounds := MAKE_BINARY(ARG);
    result := rtype'(others => new_bounds(new_bounds'left));
    result(msb downto 0) := new_bounds(msb downto 0);
    return result;
    -- synopsys synthesis_on
end;

function CONV_SIGNED(ARG: BIT; SIZE: INTEGER) return SIGNED is
    subtype rtype is SIGNED (SIZE-1 downto 0);
    variable result: rtype;
    -- synopsys built_in SYN_ZERO_EXTEND
begin
    -- synopsys synthesis_off
    result := rtype'(others => '0');
    result(0) := MAKE_BINARY(ARG);
    return result;
    -- synopsys synthesis_on
end;
```

```
    function AND_REDUCE(ARG: BIT_VECTOR) return BIT is
        variable result: BIT;
    begin
        result := '1';
        for i in ARG'range loop
            result := result and ARG(i);
        end loop;
        return result;
    end;

    function NAND_REDUCE(ARG: BIT_VECTOR) return BIT is
    begin
        return not AND_REDUCE(ARG);
    end;

    function OR_REDUCE(ARG: BIT_VECTOR) return BIT is
        variable result: BIT;
    begin
        result := '0';
        for i in ARG'range loop
            result := result or ARG(i);
        end loop;
        return result;
    end;

    function NOR_REDUCE(ARG: BIT_VECTOR) return BIT is
    begin
        return not OR_REDUCE(ARG);
    end;

    function XOR_REDUCE(ARG: BIT_VECTOR) return BIT is
        variable result: BIT;
    begin
        result := '0';
        for i in ARG'range loop
            result := result xor ARG(i);
        end loop;
        return result;
    end;

    function XNOR_REDUCE(ARG: BIT_VECTOR) return BIT is
    begin
        return not XOR_REDUCE(ARG);
    end;
end arithmetic;
```

Bibliografia

[Allen 90] Jonathan Allen. Performance-Directed Synthesis of VLSI Systems. *Proceedings of the IEEE*, 78(2), February 1990.

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

[Bartlett 86] K. Bartlett, W. Cohen, A. de Geus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design*, 5(4), October 1986.

[Bergé 92] J. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference*. Kluwer Academic Publishers, 1992.

[Brayton 87] R. Brayton, R. Rudell, A. Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, 6(6), November 1987.

[Brayton 90] R. Brayton, G. Hachtel, and A. Vincentelli. Multilevel Logic Synthesis. In *Proceedings of the IEEE*, volume 78, February 1990.

[Brown 92] Mark Brown. VHDL Intermediate Form Standardization: Process, Issues and Status. In *Proceedings of European Design Automation Conference / Proceedings of Euro-VHDL*, September 1992.

[Carlson 91] Steve Carlson. *Application of VHDL to Circuit Design*, chapter Modeling Style Issues for Synthesis. Kluwer Academic Publishers, 1991. Ed. R. Harr and A. Stanculescu.

[Compiler 91] Menthor Graphics. *System-1076 Design and Model Development Manual*, September 1991. Version 8.0_1.

[Cottrell 92] R. Cottrell, K. Nolan, and Mark Brown. VHDL Analog Extensions: Process, Issues, and Status. In *Proceedings of European Design Automation Conference / Proceedings of Euro-VHDL*, September 1992.

[Devadas 88] S. Devadas, Hi-Keung Ma, A. Newton, and S. Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations. *IEEE Transactions on Computer-Aided Design*, 7(12), December 1988.

[Guyler 92] Andrew Guyler. VHDL 1076-1992 Language Changes. In *Proceedings of European Design Automation Conference / Euro-VHDL*, September 1992.

[Harper 92] P. Harper and K. Scott. Towards A Standard VHDL Synthesis Package. In *Proceedings of European Design Automation Conference / Proceedings of Euro-VHDL*, September 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.

[Library 92] European Silicon Structures. *ES2 ECPD10 Library Databook*, February 1992. Version E01A06.

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

[Lis 89] J. Lis and D. Gajski. VHDL Synthesis Using Structured Modeling. In *Design Automation Conference*, June 1989.

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

[Maginot 92] Serge Maginot. Evaluation Criteria of HDLs: VHDL compared to Verilog, UDL/I & M. *Proceedings of European Design Automation Conference / Euro-VHDL*, 1992.

[Man 90] H. Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Meerbergen, S. Note, and J. Huisken. Architectural-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms. In *Proceedings of the IEEE*, volume 78, 1990.

[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.

[Micheli 85] G. Micheli, R. Brayton, and S. Vincentelli. Optimal State Assignment for Finite State Machines. *IEEE Transactions on Computer-Aided Design*, 4(3), July 1985.

[Micheli 90] G. Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System. *IEEE Design & Test of Computers*, October 1990.

[Monteiro 92] José Carlos Monteiro. Codifição de Máquinas de Estados em Síntese Automática de Circuitos Lógicos. Master's thesis, Instituto Superior Técnico, Agosto 1992.

[Morison 84] J. Morison, N. Peeling, and T. Throp. ELLA: Hardware Description otr Specification ? In *IEEE International Conference on Computer-Aided Design*, 1984.

[Perry 91] Douglas Perry. *VHDL.* McGraw-Hill, Inc., 1991.

[Rouillard 92] Jacques Rouillard. Analysis os user requirements. In *Proceedings of European Design Automation Conference / Euro-VHDL*, September 1992.

[Rudell 87] R. Rudell and A. Vincentelli. Multiple-Value Minimization for PLA Optimization. *IEEE Transactions on Computer-Aided Design*, 6(5), September 1987.

[Selz 92] Manfred Selz. Synthesis with VHDL. Slides, Institute of Computer Aided Circuit Design, June 1992. Presented at the European VHDL Synthesis Working Group (Munich).

[Selz 93] M. Selz and K. Mueller-Glaser. Synthesis with VHDL. In *VHDL-Forum for CAD in Europe*, March 1993.

[Shahdad 91] Moe Shahdad. Overview of VHDL 92 Standardization Process. In *Proceedings of Euro-VHDL'91*, September 1991.

[Shahdad 92] Moe Shahdad. 1992 VHDL Standardization Overview. In *Proceedings of European Design Automation Conference / Euro-VHDL*, September 1992.

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

[Simulator 91a] European Silicon Structures. *Solo 2030 User Guide*, December 1991. Version E02A01.

[Simulator 91b] Mentor Graphics. *QuickSim II User's Manual*, September 1991. Version 8.0_1.

[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.

[Verilog 91] Open Verilog International. *Verilog Hardware Descript. Lang. Reference Manual*, October 1991. Version 1.0.

[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.

[Zamfirescu 91] A. Zamfirescu and V. Berman. Language Validation in VHDL. In *Proceedings of Euro-VHDL'91*, September 1991.