CUPL

Ambiente para Projecto de Hardware

Janeiro de 2007

CUPL - Ambiente para Projecto de Hardware

Mário P. Véstias

Instituto Superior de Engenharia de Lisboa - ISEL

ÍNDICE

1 Introdução

As primeiras abordagens para o desenvolvimento de circuitos digitais com tecnologia de lógica programável iniciavam o projecto a partir de um esquemático. A representação esquemática do circuito era depois convertida manualmente para uma descrição textual (ficheiro de texto) equivalente onde se indicava o estado de cada um dos elementos de configuração de um dispositivo programável: um fusível ou um anti-fusível. O ficheiro era depois carregado para o dispositivo programável usando um programador.

Para gerar manualmente o ficheiro de texto com a configuração do dispositivo, exigia-se que o projectista conhecesse a estrutura interna do dispositivo e o formato usado pelo programador, que dependia do fabricante. O processo de projecto tornava-se, desta forma, bastante demorado e sujeito a erros que não eram fáceis de identificar.

Para melhorar o processo, a **JEDEC** (***J**oint **E**lectron **D**evice **E**ngineer **C**ouncil*) propôs um formato standard para os ficheiros de programação de dispositivos lógicos programáveis (PLD – *Programmable Logic Devices*). Na mesma altura, o autor da primeira PAL desenvolveu o PAL *Assembler* (PALASM) que consistia numa linguagem de descrição de hardware bastante rudimentar e numa aplicação software de conversão desta linguagem no respectivo ficheiro de programação. A introdução do PALASM foi um progresso considerável, mas apenas suportava dispositivos PAL e não realizava qualquer minimização ou optimização lógica sobre a descrição hardware inicial. Com o objectivo de ultrapassar estas limitações, a Data I/O lançou o ambiente **ABEL** (***A**dvanced **B**oolean **E**xpression **L**anguage*) em 1983 e a *Assisted Technology* lançou o ambiente **CUPL** (***C**ommon **U**niversal tool for **P**rogrammable **L**ogic*). O ABEL e o CUPL incluem as linguagens de descrição de hardware com o mesmo nome, linguagens ABEL e CUPL, e as aplicações software que convertem a descrição hardware no ficheiro de programação do PLD. Ao contrário do PALASM, os ambientes ABEL e CUPL incluem algoritmos de minimização e de optimização e suportam múltiplos dispositivos PLD de múltiplos fabricantes.

2 Projecto Baseado em Linguagens de Descrição de Hardware

Os ambientes e as metodologias de projecto de sistemas digitais, incluindo os ambientes de projecto ABEL e CUPL, têm sofrido grandes alterações devido, em parte, à evolução tecnológica do fabrico de circuitos integrados que tem permitido a integração de sistemas digitais com funcionalidades cada vez mais complexas e com uma dimensão crescente.

As primeiras abordagens ao projecto de sistemas digitais baseavam-se na utilização de representações esquemáticas como forma de capturar a funcionalidade pretendida para o sistema. A representação do sistema com um esquemático permitia uma análise visual do circuito. Com esta abordagem, o projectista começava por propor um diagrama de blocos da arquitectura do sistema, que servia como especificação preliminar, em geral incompleta, que cumprisse os requisitos funcionais e de projecto (e.g., desempenho, custo, dissipação de potência, etc.). Este diagrama de blocos era depois refinado e entregue a uma equipa de projectistas lógicos que tinham a árdua tarefa de converter cada bloco funcional num esquemático lógico que depois era capturado com ferramentas de captura de esquemáticos para verificação da sua funcionalidade e dos requisitos através de simulação.

Para sistemas de pequena dimensão, esta abordagem não levantava grandes problemas. No entanto, à medida que aumentava a complexidade dos sistemas a projectar, facilmente se concluiu que os fluxos de projecto de sistemas digitais baseados em esquemático não eram viáveis. A visualização, a captura do comportamento, a depuração de erros e a verificação da funcionalidade, a compreensão do circuito e a manutenção do projecto de sistemas com muitas portas lógicas tornou-se incomportável. A hierarquização da representação com vários níveis de esquemáticos melhorou o processo, mas continuava a exigir um grande esforço de análise. Por além disso, o processo de análise e de minimização de funções lógicas era feito sobre o diagrama lógico, um processo bastante trabalhoso e sujeito a erros.

Foi nesta altura que alguns fabricantes começaram a desenvolver fluxos de projecto e ferramentas de apoio ao projecto de circuitos com base em *linguagens de descrição de hardware*. A descrição dos sistemas digitais passou, assim, a ser feita textualmente, em vez de graficamente, usando as designadas linguagens de descrição de hardware.

A maioria das linguagens de descrição de hardware permite descrever comportamentos hardware a diversos níveis de abstracção. Por exemplo, podemos descrever um somador de oito

bits como $S = A + B$ ou então determinar as expressões lógicas das saídas e depois descrevê-las com a linguagem. Quanto maior o nível de abstracção, menos detalhes de implementação são considerados e, consequentemente, é mais tratável pelo projectista. No entanto, exige um ou mais passos de tradução do comportamento lógico em expressões lógicas, que não seria necessário caso a descrição tivesse sido feita ao nível lógico.

Idealmente, o projectista gostaria de trabalhar a um nível de abstracção muito próximo da sua linguagem. No entanto, para tal, é necessário ter disponíveis ferramentas que lhe permitam gerar as expressões lógicas a partir da sua descrição, ou seja, ferramentas de síntese.

Relativamente à linguagem CUPL, esta apenas considera o nível lógico. Apesar de também incluir um mecanismo de descrição comportamental de máquinas de estado e de tabelas de verdade sem ter de descrever directamente as expressões lógicas respectivas.

No capítulo seguinte, iremos abordar o ambiente de projecto CUPL que assenta na linguagem de descrição de hardware, CUPL.

3 Ambiente de Projecto Atmel-WINCUPL

O Atmel-WINCUPL é um ambiente de projecto de sistemas digitais em circuitos lógicos programáveis tipo PAL®. Está adaptado ao projecto e à verificação de circuitos lógicos programáveis de pequena dimensão, nomeadamente circuitos lógicos programáveis da Atmel, sendo que suporta os dispositivos PAL® 16V8, 20V8, 22V10, 750, 750B, 750C, 1500, 2500, 2500B.

O Atmel-WINCUPL aceita uma descrição hardware em CUPL ao nível lógico onde se podem descrever máquinas de estados, tabelas de verdade e equações booleanas. O ambiente inclui vários algoritmos de minimização lógica que podem gerar diferentes resultados para uma mesma descrição, bem como um simulador integrado, com atraso unitário, que permite uma verificação rápida do circuito. A partir da descrição funcional do circuito, o Atmel-WINCUPL gera os ficheiros de programação do dispositivo programável.

3.1 Fluxo de Projecto do CUPL

O CUPL é um ambiente de projecto ao nível lógico que gera um ficheiro de programação do dispositivo programável a partir da descrição funcional em linguagem CUPL de acordo com o fluxo da figura 1.

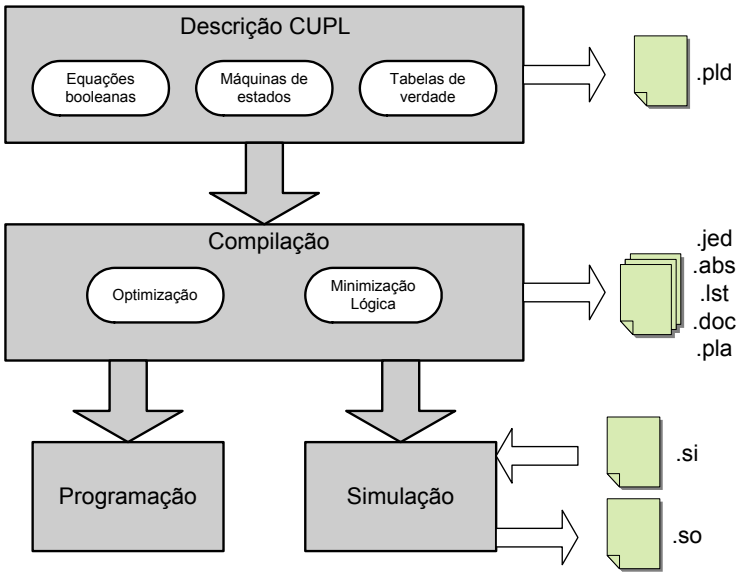

Figura 1 – fluxo de projecto do CUPL

O projecto começa com uma descrição funcional em linguagem CUPL do circuito a implementar na PAL®. A descrição CUPL é guardada num ficheiro com extensão .pld.

De seguida, a descrição CUPL, juntamente com directivas e comandos do projectista, é compilada. A compilação gera um conjunto de ficheiros com informação mais detalhada do circuito final que depois serão usados para programar o dispositivo lógico programável ou para simular o circuito. A fase de compilação inclui os passos de optimização e de minimização lógica com os objectivos seguintes:

- Minimização lógica - durante a compilação do circuito pode ser realizada uma minimização lógica com o objectivo de reduzir os recursos hardware necessários à implementação do circuito. Aqui, cabe ao projectista decidir qual o método de minimização a utilizar de entre o conjunto seguinte:
    - Nenhuma – não é usada qualquer minimização lógica durante o processo de compilação. Esta opção é, em geral, usada apenas quando se trabalha com PROM para evitar a eliminação de termos de produto;
    - Rápida – mantém um compromisso aceitável entre minimização lógica, utilização de memória de programa e tempo de compilação;
    - Quine-McCluskey – é o que consegue a maior minimização, mas a troco de mais recursos de memória e de tempo de compilação;
    - Presto – garante um elevado nível de minimização, mas requer menos memória e menos tempo de compilação;
    - Expresso – garante uma minimização no mínimo ao nível da conseguida com o Presto com menos tempo de compilação, mas requer mais memória de programa.
- Optimização - durante a compilação do circuito pode ser realizada uma optimização das expressões lógicas com o objectivo de ajustar melhor as expressões aos recursos do dispositivo alvo. Também neste caso, cabe ao projectista decidir qual o método de optimização a utilizar de entre o conjunto seguinte:
    - *Best for Polarity* – optimiza a utilização dos termos de produto das variáveis;
    - *Demorgan* – aplica as regras de Demorgan a todas as variáveis;
    - *Keep Xor Equations* – não expande as funções lógicas XOR em equações AND-OR. Esta opção é usada em projectos independentes do dispositivo ou projectos em que o dispositivo alvo suporta portas XOR;
    - *P-Term Sharing* – força a partilha de termos de produto durante a minimização.

Os ficheiros gerados na fase de compilação são os seguintes:

- .jed – ficheiro ASCII usado na programação do dispositivo;
- .abs – ficheiro usado pelo simulador lógico CSIM;

- .lst – ficheiro com os erros encontrados no ficheiro de entrada (.pld);
- .doc – ficheiro que contém as equações lógicas expandidas, uma tabela com as variáveis usadas, a utilização dos termos de produto e informação sobre as ligações programáveis;
- .mx – ficheiro que contém uma lista expandida de todas as macros usadas no ficheiro de descrição CUPL. Também inclui as expressões expandidas que usam a construção REPEAT do CUPL (ver secção 3.2);
- .pla – ficheiro para ser usado por ferramentas como o *PLEASURE* ou outras ferramentas que usem o formato PLA de Berkeley.

A simulação do circuito é um passo opcional cuja eficiência depende da complexidade do circuito e da facilidade de programação do dispositivo. Se não houver grande dificuldade em realizar o teste directamente sobre o dispositivo programável, então é natural que o projectista não realize o passo de simulação. O simulador recebe um ficheiro com a extensão .si criado pelo utilizador, que contém os vectores de teste, e gera um ficheiro de saída com extensão .so com os resultados da simulação. Este ficheiro é usado para gerar os resultados em gráfico.

4 Linguagem CUPL

O CUPL é uma linguagem de descrição de hardware e, tal como qualquer linguagem de descrição de hardware, não deve ser vista como uma linguagem software de alto nível. Sendo uma linguagem de descrição, deve ser usada para descrever os circuitos. Como tal, antes de iniciar a descrição do circuito, o projectista deve começar por estabelecer a arquitectura do circuito, ou seja, identificar os blocos hardware e a sua interligação. Só então deverá usar o CUPL para descrever a arquitectura hardware do circuito para ser posteriormente compilada e programada no dispositivo lógico programável.

4.1 Elementos da linguagem CUPL

Como qualquer linguagem, o CUPL tem variáveis, constantes, números, comentários, operações, etc. Neste sentido, começamos por descrever nesta secção os elementos básicos da linguagem.

4.1.1 Variáveis

Variáveis em CUPL são todas as palavras usadas para identificar pinos, nós internos, constantes, sinais de entrada, sinais de saída, sinais intermédios ou conjuntos de sinais. No CUPL existem regras relativamente à composição das palavras usadas para nomear uma variável, nomeadamente:

- As variáveis podem começar com um número, um carácter, um sublinhado, mas têm de conter pelo menos um carácter;
- As variáveis são sensíveis às maiúsculas e minúsculas;
- As variáveis podem conter no máximo 31 caracteres. Todos os caracteres a mais são truncados;
- As variáveis não podem conter qualquer palavra (ver tabela 1) ou símbolo reservados (ver tabela 2) do CUPL.

APPEND	ASSEMBLY	ASSY	COMPANY	CONDITION
DATE	DEFAULT	DESIGNER	DEVICE	ELSE
FIELD	FLD	FORMAT	FUNCTION	FUSE
GROUP	IF	JUMP	LOC	LOCATION
MACRO	MIN	NAME	NODE	OUT
PARTNO	PIN	PINNODE	PRESENT	REV
REVISION	SEQUENCE	SEQUENCED	SEQUENCEJK	SEQUENCERS
SEQUENCET	TABLE			

Tabela 1 – Palavras reservadas do CUPL

&	#	(	)	-	*
+	[	]	/	:	.
. .	/*	*/	;	,	!
‘	=	@	$	^	

Tabela 2 – Símbolos reservados do CUPL

Exemplo – exemplos válidos de nomes de variáveis

```
xpto
clk0
8251Control
DMA_bus_control
```

Exemplo – exemplos inválidos de nomes de variáveis

```
2007 (não contém um carácter)
I/O (contém um carácter reservado)
bus-function (contém uma palavra reservada)
```

Quando um nome de uma variável termina num número é designada ***variável indexada***. Por exemplo, A0, A1, A2, são variáveis indexadas. Este tipo de variável é usado geralmente para identificar grupos de linhas, de sinais, etc. Os números usados na indexação têm de estar entre 0 e 31. Caso contrário, a variável não é indexada.

Exemplo – exemplos válidos de nomes de variáveis indexadas

```
X0
a01
a1 (diferente da anterior)
DMA_bus_control_4
```

Exemplo – exemplos inválidos de nomes de variáveis indexadas

```
Data (não termina num número)
Data_63 (não termina num número entre 0 e 31)
```

4.1.2 Números

Em CUPL, qualquer número pode variar entre 0 e 2^{32} – 1 (números de 32 bits) e pode ser representado em uma de quatro bases: decimal, binária, octal ou hexadecimal. A base por defeito é a hexadecimal, excepto nos números usados na identificação dos pinos do dispositivo e nas variáveis indexadas, que são sempre decimais.

Para indicar a base do número, basta precedê-lo de um determinado prefixo (que não é sensível ao tamanho da letra) (ver tabela 3).

Base	Prefixo	Exemplo
Binária	‘b’ ou ‘B’	‘b’0100
Octal	‘o’ ou ‘O’	‘O’723
Decimal	‘d’ ou ‘D’	‘d’18967
hexadecimal	‘h’ ou ‘H’	‘h’BF3 ou BF3

Tabela 3 – prefixos usados na identificação da base dos números

Os números em representação binária, octal ou hexadecimal podem incluir indiferenças. Por exemplo, ‘b’11X0, ‘o’63X1, ‘h’7fXX (este último representa os valores hexadecimais entre 7F00 e 7FFF). Existe uma outra forma de representar um intervalo de números com a notação ‘base’[n .. m]. Por exemplo, ‘h’7fXX poderia também ser representado como ‘h’[7F00 .. 7FFF].

4.1.3 Comentários

Os comentários são um instrumento fundamental para a organização, a compreensão e a explicação de uma descrição textual.

Em CUPL, um comentário é tudo aquilo que se encontra entre os símbolos /* e */. Por exemplo, /* isto é um comentário e, como tal, não é interpretado pelo compilador */.

Os comentários não são interpretados pelo compilador, pelo que não influenciam o tempo de compilação de um projecto.

4.1.4 Listas

A linguagem CUPL é caracterizada por conter formas compactas de representação de elementos da linguagem. Uma das formas compactas mais usadas é a lista, com a representação [*variável*, *variável*, ..., *variável*]. Por exemplo, [D0, D1, D2, D3] ou [TRUE, FALSE].

Sempre que as variáveis de uma lista forem indexadas sequencialmente, pode ser usado o formato [*variávelm .. n*], em que *m* é o primeiro índice da lista de variáveis e *n* é o último. Por exemplo, a lista [D0, D1, D2, D3] poderia ser reescrita da forma [D0 .. 3].

As duas formas de representação de listas podem ser misturadas numa única lista. Por exemplo, [D0 .. 2, D3] é equivalente a [D0, D1, D2, D3].

4.2 Descrição CUPL

Uma descrição de um circuito em linguagem CUPL contém três secções: cabeçalho, declarações e descrição funcional, que se descrevem nas secções seguintes.

4.2.1 Cabeçalho

O cabeçalho contém informação útil para a manutenção e gestão do projecto e tem o formato seguinte:

```
Name     xpto;
Partno   xpto;
Date     x/p/to;
Revision xp;
Designer xpto;
Company  xpto;
Assembly xpto;
Location xpto;
Device   xpto;
```

- *NAME* – o nome especificado determina o nome dos ficheiros de projecto: name.jed, name.hex, etc. O nome pode conter até 32 caracteres, que serão truncados para oito em sistemas DOS;
- *PARTNO* – pode ser usado para especificar o número de fabricante do produto;
- *REVISION* – começa com 01 e é incrementado sempre que o ficheiro é alterado;
- *DATE* – é alterado com a data actual sempre que o ficheiro é modificado;
- *DESIGNER* – pode ser usado para especificar o nome do projectista;
- *COMPANY* – pode ser usado para indicar o nome da companhia, servindo apenas para documentação;
- *ASSEMBLY* – pode ser usado para indicar o nome ou número do PCB onde ser usado o dispositivo lógico programável;
- *LOCATION* – pode ser usado para indicar uma referência ao local (e.g., PCB) onde será colocado o dispositivo;
- *DEVICE* – indica o dispositivo a ser usado por defeito na fase de compilação.

Exemplo – cabeçalho de um projecto de um contador

```
Name          Contador;
Partno        C001234;
Revision      01;
Date          10/01/07 ;
Designer      Mário;
Company       ISEL;
Assembly      PC Board;
Location      Inside;
Device        22v10;
```

Todos os campos do cabeçalho têm de estar presentes, mas apenas o campo *Name* terá de conter um nome válido, pois é usado para nomear o ficheiro JEDEC (.jed). Caso algum campo não esteja presente, o compilador gera um erro.

4.2.2 Declarações

A secção de declarações é usada para especificar todas as variáveis usadas na descrição, nomeadamente a atribuição de variáveis a pinos e a nós e a criação de variáveis auxiliares.

Atribuição de variáveis a pinos

A declaração de pinos consiste na atribuição de variáveis a pinos, através do seu número. Para evitar conflitos entre a funcionalidade da variável e a função do pino do dispositivo, deve consultar o manual do dispositivo para identificar a especificidade de cada um dos pinos do dispositivo.

O formato de declaração dos pinos é PIN *número_do_pino* = [!] *variável*;, em que *número_do_pino* é um número decimal correspondente ao número do pino no dispositivo ou uma lista de números agrupados segundo uma lista. O '!' é opcional e serve para definir a polaridade do sinal. Esta opção é bastante útil, já que se o projectista declarar a polaridade do sinal depois não tem de se preocupar com a mesma ao longo da descrição do circuito. *variável* é o nome de uma variável ou uma lista de variáveis. Por fim, a declaração tem de terminar com o símbolo ';'.

Exemplo – declaração de pinos

```
PIN 1 = clk;
PIN 2 = !reset;
PIN [3, 4] = ![on, off]; /* PIN 3 = !on; PIN 4 = off; */
PIN [5..7] = [in0..2]; /* PIN 5 = in0; PIN 6 = in1; PIN 7 = in2;*/
```

Caso se pretenda projectar um circuito sem ter em conta qualquer tipo de dispositivo, os números dos pinos podem ser omitidos. Deste modo, o projectista consegue tirar algumas conclusões quanto ao seu circuito e a partir daí escolher o dispositivo alvo.

Exemplo – declaração de pinos sem atribuição de números

```
PIN  = clk;
PIN  = !reset;
PIN  = ![on, off];
PIN  = [in0..2];
```

É importante referir que na linguagem CUPL, ao contrário de muitas linguagens de descrição de hardware, não se indica se um determinado pino vai ser usado como entrada, saída ou entrada/saída. Esta informação é inferida pelo compilador pela forma como a variável associada ao pino é utilizada nas expressões lógicas. Sempre que houver incompatibilidade entre ambos, o compilador gera uma mensagem de erro.

Atribuição de variáveis a nós

Alguns dispositivos programáveis contêm funções que não estão directamente associadas a um pino externo do dispositivo. Por exemplo, a PAL® ATF750C contém uma macrocélula interna por cada macrocélula ligada a um pino externo.

Para que o projectista possa usar as funções associadas a estes nós internos, terá de lhes associar variáveis. Uma vez que o nó não está associado a um pino externo, a variável respectiva não pode

ser declarada com a palavra PIN. Em vez disso, o projectista deverá usar a palavra NODE. O formato de declaração é

```
NODE [!] var;,
```

em que o símbolo '!' é usado opcionalmente para inverter a polaridade, como acontece na declaração com PIN, e *var* é um nome de variável ou uma lista de variáveis (não esquecer de terminar a declaração com um ';')

Na declaração com NODE não é indicado qualquer número de pino. No entanto, internamente, o CUPL associa-lhe automaticamente um número de um pseudo pino.

🕮 **Nota**: após declarar uma variável associada a um NODE, terá de a usar necessariamente. Caso contrário, o compilador gera uma mensagem de erro.

Exemplo – declaração de nós internos

```
NODE input;
NODE [addr0..3];
```

Como foi referido, a atribuição de variáveis a nós internos com a sintaxe **NODE** não necessita da indicação explícita do número identificar do nó interno, sendo a sua atribuição feita automaticamente. No caso do projectista querer indicar explicitamente o nó interno a usar por uma determinada variável, deverá usar a palavra **PINNODE**. A sua declaração é similar à do PIN, tendo um formato idêntico:

```
PINNODE número_nó = [!]var;
```

em que *número_nó* é o número de um nó interno ou uma lista de nós, o símbolo '!' é usado opcionalmente para definir a polaridade e *var* é o nome de uma variável ou uma lista de variáveis.

🕮 **Nota**: deve consultar a documentação de cada dispositivo programável para identificar os números associados aos nós internos.

Exemplo – declaração de nós internos com associação de números

```
PINNODE [29..34] = [State0..5];
PINNODE 25       = Buried;    /* Buried register part     */
```

Atribuição de variáveis a grupos de bits

O CUPL permite a atribuição de uma variável a um conjunto de bits de acordo com o formato seguinte:

```
FIELD var = [var, var, ... var] ;
```
em que *var* são nomes de variáveis.

Uma variável de grupo pode ser usada em qualquer expressão lógica, sendo que a operação lógica ir-se-á aplicar a cada um dos bits do grupo (ver secção 4.2.3 para saber quais as operações lógicas que suportam variáveis de grupo).

Um dos exemplos de utilização de uma variável de grupo é na identificação de buses.

Exemplo – declaração de uma variável de grupo

```
FIELD Data = [D7,D6,D5,D4,D3,D2,D1,D0] ;
```

Internamente, o compilador associa um campo de 32 bits à variável, sendo que cada bit representa um bit da variável de grupo. Além disso, o compilador faz corresponder o índice da variável, no caso de variáveis indexadas, ao bit respectivo do campo interno. Por exemplo, D5 corresponde ao bit 5 do campo interno. Tal significa que a ordem das variáveis indexadas na declaração de uma variável de grupo não é relevante. Por exemplo, [D3..0] é o mesmo que [D0..3]. Por esta razão, uma variável de grupo não deve conter simultaneamente variáveis indexadas e não indexadas.

Atribuição do nível de minimização lógica a variáveis

Por defeito, todas as variáveis estão sujeitas ao nível de minimização usado na linha de comandos de execução do CUPL ou escolhido nas opções disponíveis no ambiente WinCUPL. Caso o projectista pretenda aplicar um nível de optimização diferente para uma determinada variável, poderá fazê-lo através da palavra MIN, cujo formato é o seguinte:

```
MIN var [.ext] = nível;
```

em que *var* é uma variável ou lista de variáveis, .ext é uma extensão opcional (ver secção 4.2.3) de identificação de uma função da variável e *nível* corresponde a um nível de optimização entre 0 e 4 de acordo com a tabela 4.

Nível	Método de minimização
0	Sem minimização
1	Quick
2	Quine McCluskey
3	Presto
4	Expresso

Tabela 4 – Níveis de minimização disponíveis em CUPL

Uma declaração de minimização associada a uma variável sobrepõe-se à escolha global de minimização. Assim, é possível optar, por exemplo, por não optimizar uma determinada variável, possivelmente para evitar alguma condição de geração de *glitches*, e optar por minimizar uma outra com o objectivo de reduzir o número de termos.

Exemplo – definição do tipo de minimização a usar em determinadas variáveis

```
MIN async_out = 0; /* a variável async_out não deve ser minimizada */
MIN [outa, outb] = 2; /* as variáveis outa e outb devem ser
minimizadas com o método Quine McCluskey */
```

4.2.3 Descrição Funcional

A descrição funcional inclui as expressões lógicas que definem a funcionalidade que se pretende implementar no dispositivo lógico programável. A linguagem CUPL suporta a descrição directa de equações booleanas, com as quais é possível descrever qualquer circuito lógico. No entanto, devido à utilização frequente de máquinas de estados e de tabelas de verdade durante o processo de projecto, o CUPL inclui mecanismos específicos para a descrição de máquinas de estados e de tabelas de verdade. Nas secções seguintes, descreve-se os três mecanismos de descrição de hardware em CUPL.

4.2.3.1 Descrição de equações booleanas

Operadores Lógicos

O CUPL tem quatro operadores lógicos para descrição de expressões lógicas (ver tabela 5).

Função Lógica	Operador CUPL	Precedência	Exemplo
NOT	!	1	!A
AND	&	2	A&B
OR	#	3	A#B
XOR	$	4	A$B

Tabela 5 – Operadores lógicos disponíveis em CUPL

A tabela indica a precedência dos operadores, que está de acordo com álgebra de Boole.

Operadores e funções aritméticas

O CUPL também suporta seis operadores aritméticos, mas apenas podem ser usados na formação de expressões aritméticas nos comandos de pré-processamento $REPEAT e $MACRO (ver secção 4.1). Ver operadores aritméticos na tabela 6.

Operador aritmético	Operador CUPL	Precedência	Exemplo
Expoente	**	1	$2^3 \Rightarrow 2**3$
Multiplicação	*	2	$2\times3 \Rightarrow 2*3$
Divisão	/	2	$2\div3 \Rightarrow 2/3$
Módulo	%	2	$\text{mod}(2\div3) \Rightarrow 2\%3$
Adição	+	3	$2+3 \Rightarrow 2+3$
Subtracção	-	3	$2-3 \Rightarrow 2-3$

Tabela 6 – Operadores aritméticos disponíveis em CUPL

As expressões aritméticas têm de ser declaradas entre chavetas (e.g., {(i + 10) / 8}).

Para além dos seis operadores lógicos, o CUPL inclui ainda uma função aritmética em quatro bases diferentes, o logaritmo (ver tabela 7).

Função aritmética	Operador CUPL	Base
$\log_2 x$	LOG2	2
$\log_8 x$	LOG8	8
$\log_{10} x$	LOG	10
$\log_{16} x$	LOG16	16

Tabela 7 – Funções aritméticas disponíveis em CUPL

As funções aritméticas também só podem ser usadas nos comandos de pré-processamento $REPEAT e $MACRO.

Extensões de variáveis

Uma variável associada a um nó (macrocélula) de um dispositivo programável pode incluir uma extensão como forma de fazer referência a uma função específica associada ao nó. Por exemplo, a entrada de controlo de uma porta tri-state, a entrada de *reset* de um flip-flop, etc. Com a utilização das extensões, o projectista pode facilmente configurar cada um dos nós ou macrocélula do dispositivo (ao usar extensões, o projectista não tem de controlar directamente os fusíveis do dispositivo programável).

Cada dispositivo suporta um conjunto bem definido de extensões, de acordo com as características dos seus nós. Assim, o projectista tem de consultar o manual de cada dispositivo para saber que extensões tem disponíveis. De qualquer forma, o compilador verifica se uma determinada extensão de uma variável é suportada ou não pelo dispositivo alvo (ver extensões dos dispositivos PLD e CPLD da Atmel na tabela 8).

Extensão	Descrição
.AP	Preset assíncrono do FLIP-FLOP
.AR	Reset assíncrono do FLIP-FLOP
.CE	Entrada *enable* de FF tipo D
.CK	Clock programável de FLIP-FLOP
.CKMUX	Selecção de clock
.D	Entrada D de FF tipo D
.DFB	Ligação de retorno registada tipo D
.DQ	Saída Q de FF tipo D
.INT	Ligação interna de retorno
.IO	Selecção de ligação de retorno
.J	Entrada J de FF tipo JK
.K	Entrada K de FF tipo JK
.L	Entrada D de latch transparente
.LE	*Enable* de *latch* programável
.LQ	Saída Q de *latch* transparente
.OE	Entrada de controlo tri-state
.R	Entrada R de FF tipo RS
.S	Entrada S de FF tipo RS

.SP	*Preset* síncrono de FLIP-FLOP
.T	Entrada T de FF tipo T
.TFB	Ligação de retorno registada com FF tipo T

Tabela 8 – Extensões CUPL para os dispositivos PLD/CPLD da Atmel

Por exemplo, para acedermos à entrada de controlo de um tri-state teríamos:

variable.OE

Para o caso específico da extensão .OE, o compilador atribui-lhe um valor específico de acordo com a direcção do pino a que está associado. Se for um pino de saída, o *tri-state* deverá estar sempre activo, pelo que o compilador lhe atribui o valor lógico '1'. Caso seja um pino de entrada, então é-lhe atribuído o valor lógico '0'. Se for um pino bidireccional, então cabe ao projectista atribuir uma função lógica à entrada de controlo do *tri-state*, sobrepondo-se ao valor por defeito atribuído pelo compilador.

Equações Lógicas

Em CUPL a descrição da função do circuito é feita com equações lógicas ou booleanas de acordo com o formato seguinte:

[!] *variável* [.ext] = *expressão*;

em que *variável* é o nome de uma variável ou lista de variáveis, .ext é uma extensão opcional da variável e *expressão* é uma combinação de variáveis e operadores lógicos.

As equações lógicas podem ser definidas não só para variáveis associadas a pinos ou a nós, mas também para variáveis temporárias ou intermédias, ou seja, variáveis que são usadas apenas para facilitar a escrita de expressões lógicas e para estruturar e organizar o código.

Exemplo – equações lógicas

```
SEL_0=A15 & !A14;
Q0.D=Q1 & Q2 & Q3;
Q1.J = Q2 # Q3;
Q1.K = Q2 & !Q3;
MREQ=READ # WRITE; /* variável intermédia */
SEL_1=MREQ & A15;  /* utilização da variável intermédia */
[D0..3] = 'h'FF;
```

A forma de atribuição de expressões lógicas a variáveis descrita anteriormente apenas permite atribuir uma expressão à variável. A linguagem CUPL inclui a palavra **APPEND** que possibilita a atribuição de mais de uma expressão a uma variável, ou seja, as múltiplas expressões são agrupadas com o operador lógico OR. O formato é o seguinte:

APPEND [!]*variável*[.ext] = *expressão*;

similar à definição de uma equação simples, bastando acrescentar o termo **APPEND**.

Caso ainda não tenha sido atribuída uma expressão à variável, o resultado é o mesmo com e sem **APPEND**.

Exemplo – utilização do APPEND na descrição de um sinal

```
APPEND Y = A0 & A1 ;
APPEND Y = B0 & B1 ;
APPEND Y = C0 & C1 ;

Esta forma de descrição é idêntica a:
Y = (A0 & A1) # (B0 & B1) # (C0 & C1) ;
```

O APPEND é bastante útil para adicionar termos às variáveis de uma máquina de estados (ver tópico de descrição de máquinas de estados) ou na construção de funções definidas pelo utilizador (ver tópicos avançados na secção 4.2.4).

Operações sobre conjuntos

A sintaxe da linguagem CUPL permite que qualquer operação aplicável a um único sinal seja também aplicável a múltiplos sinais agrupados num conjunto. É assim possível realizar operações entre uma variável ou uma expressão e um conjunto ou entre conjuntos.

O resultado da aplicação de uma operação entre um conjunto e uma variável ou uma expressão singular é um novo conjunto em que a operação é realizada entre cada um dos elementos do conjunto e a variável ou a expressão.

Exemplo – operação entre um conjunto e uma variável

```
[D0, D1, D2, D3] & read

resulta no conjunto seguinte:
[D0 & read, D1 & read, D2 & read, D3 & read]
```

Se a operação for aplicada entre dois conjuntos, necessariamente do mesmo tamanho, o resultado é um conjunto cujos elementos resultam da aplicação da operação entre elementos de ambos os conjuntos.

Exemplo – operação entre dois conjuntos

```
[A0, A1, A2, A3] & [B0, B1, B2, B3]

resulta no conjunto:
[A0 & B0, A1 & B1, A2 & B2, A3 & B3]
```

O mesmo poderia ter sido feito mais compactamente criando um **FIELD**.

Exemplo – operação entre dois conjuntos definidos com FIELD

```
FIELD a_inputs = [A0, A1, A2 A3];
FIELD b_inputs = [B0, B1, B2, B3];
a_inputs & b_inputs ⇒ [A0 & B0, A1 & B1, A2 & B2, A3 & B3]
```

Com esta sintaxe podemos definir facilmente máscaras de bits. Por exemplo, um contador binário de 4-bits poderia ser descrito do seguinte modo:

Exemplo – contador de 4-bits descrito com operações sobre conjuntos

```
field count = [Q3, Q2, Q1, Q0];
count.d = 'b' 0001 & (!Q0) # 'b' 0010 & (Q1 $ Q0)
   # 'b' 0100 & (Q2 $ Q1 & Q0) # 'b' 1000 & (Q3 $ Q2 & Q1 & Q0);
```

(como exercício, pode confirmar as equações☺).

A linguagem CUPL inclui ainda uma operação de igualdade que tem como resultado uma expressão booleana. A operação de igualdade pode ser usada em dois contextos diferentes: com constante e com operadores lógicos.

Quando usado com constante, o formato é o seguinte:

1. [*var*, *var*, ... *var*]: *constante* ; ou

2. *bit_field_var* : *constante* ;

em que *var* é o nome de uma variável e *constante* é um número em qualquer base, o símbolo ':' é o operador de igualdade.

Quando encontra uma expressão deste tipo, o compilador altera a variável de acordo com o bit da constante respectivo. Se estiver a '1', mantém a variável, se estiver a '0' nega a variável. Caso seja uma indiferença 'X', a variável é removida. As variáveis alteradas são depois reunidas com uma operação lógica AND.

Exemplo – operação de igualdade aplicada a uma variável de quatro bits

```
select = [A3..0]:'h'D ;
```

Neste caso, os elementos A3, A2 e A0 mantêm-se inalterados porque os bits correspondentes da constante estão a '1'. A variável A1 é negada, pois o bit correspondente está a '0'. A expressão final ficaria:

```
select = A3 & A2 & !A1 & A0 ;
```

Exemplo – operador de igualdade com indiferenças

```
select = [A3..0]:'b'1X0X ;
```

neste caso, o elemento A3 permanece inalterado, A1 é negado e os restantes são removidos. A expressão final ficaria:

```
select = A3 & !A1 ;
```

No contexto de aplicação do operador igualdade como operador lógico, o resultado é aplicar o mesmo operador a todas as variáveis. A sintaxe é a seguinte:

```
[var, var, ... , var]:operador
```

em que *var* é o nome de uma variável e *operador* é um dos operadores lógicos binários.

Exemplo – operação do operador igualdade com operador lógico

```
[A3,A2,A1,A0]:&
```

é o mesmo que

```
A3 & A2 & A1 & A0
```

Para além da operação igualdade, a linguagem CUPL inclui a operação intervalo (*range*), que é idêntica à igualdade excepto o facto de que o campo constante é um intervalo de valores em vez de um único.

Exemplo – aplicação do operador igualdade com um intervalo de valores

```
FIELD address = [A3..0];
select = address:[C..F];
```

é equivalente a:

```
select = address:C # address:D # address:E # address:F;
```

4.2.3.2 Descrição de Tabelas de Verdade

O método de descrição baseado em equações lógicas é suficiente para descrever circuitos lógicos. No entanto, ao longo de um projecto, o projectista costuma recorrer a outros mecanismos para chegar às equações lógicas do circuito. Um dos mecanismos bastante utilizado é a tabela de verdade. A sintaxe da linguagem CUPL permite descrever directamente uma tabela de verdade, cabendo depois ao compilador determinar as equações das funções lógicas de saída.

O formato de descrição de uma tabela é o seguinte:

```
TABLE var_list_1 => var_list_2  {
   input_n => output_n ;
         .
         .
      input_n => output_n ;
}
```

em que *var_list_1* define as variáveis de entrada e *var_list_2* define as variáveis de saída, *input_n* é um valor ou lista de valores relativos a *var_list_1* e *output_n* é um valor ou lista de valores relativos a *var_list_2*. Os valores de entrada podem incluir indiferenças.

Exemplo – conversor hexadecimal-BCD descrito com uma tabela de verdade

```
FIELD  input = [in3..0] ;
FIELD  output = [out4..0] ;
TABLE  input => output {

   0=>00; 1=>01; 2=>02; 3=>03;
   4=>04; 5=>05; 6=>06; 7=>07;
   8=>08; 9=>09; A=>10; B=>11;
   C=>12; D=>13; E=>14; F=>15;
}
```

Exemplo – criação de uma tabela de verdade com a atribuição de um mesmo valor de saída a múltiplas combinações de entrada

```
FIELD address = [a15..12] ;
FIELD decodes = [RAM_sel,ROM_sel,timer_sel] ;
TABLE address  => decodes {
   [1000..2FFF] => 'b'100;
   [5000..CFFF] => 'b'010;
   F000 => 'b'001;
}
```

4.2.3.3 Descrição de Máquinas de Estado

A linguagem CUPL também inclui uma sintaxe específica para a especificação de máquinas de estado. A sintaxe é a seguinte

```
SEQUENCE lista_variáveis_estado {
    PRESENT  estado0 atribuições ;
   .
   .
   .
    PRESENT  estado_n atribuições;
}
```

em que *lista_variáveis_estado* é a lista das variáveis de bits de estado, *estado_n* é o número do estado e *atribuições* são as descrições do próximo estado e das saídas.

No caso de o dispositivo alvo apenas conter um tipo de flip-flop, o SEQUENCE gera as expressões para esse tipo de flip-flop. Caso o dispositivo suporte mais de um tipo de flip-flop (e.g., tipo D e tipo JK) então é preciso especificar que tipo de flip-flop se pretende. Por defeito, o SEQUENCE assume flip-flop tipo JK. Caso se pretenda usar tipo D, tem de se usar a palavra SEQUENCED em vez de SEQUENCE. Na mesma linha de raciocínio, existem as palavras SEQUENCEJK, SQUENCERS e SEQUENCET, para os flip-flop tipo JK, SR e T, respectivamente.

Para se definir as atribuições na descrição da máquina de estados, utilizam-se as palavras reservadas IF, NEXT, OUT e DEFAULT, como se descreve nos parágrafos seguintes.

Descrição de uma transição incondicional de estado

A sintaxe de definição de uma transição incondicional de um estado é a seguinte:

```
PRESENT state_n
   NEXT state_m ;
```

em que *state_n* e *state_m* são os números dos estados.

Exemplo – transição incondicional de um estado

```
   PRESENT 'b'01
      NEXT 'b'10 ;
```

definiu-se a transição incondicional do estado 1 para o estado 2. O resultado de compilação seria o seguinte:

```
Q1.D  =  !Q1  &  Q0;
Q0.D  =  Q1 # !Q0;
```

Descrição de uma transição condicional de estado

Caso se pretenda definir uma transição condicional, usaríamos o formato seguinte:

```
PRESENT state_n
   IF  expressão NEXT  state_m;
    .
    .
   IF  expressão NEXT state_l;
[DEFAULT  NEXT state_k;]
```

em que *state_n*, *state_m*, *state_l*, *state_k* são os números dos estados, *expressão* é uma qualquer expressão (ver secção 4.2.3.1).

Podem-se definir as condições que forem necessárias e uma condição opcional por defeito (**DEFAULT**), para o caso de não se verificar nenhuma das condições anteriores.

🕮 **Nota**: a condição por defeito é o complemento de todas as outras condições. Como tal, pode gerar uma expressão bastante complexa. Em determinados casos, pode optimizar a solução final se substituir a condição por defeito por algumas condições.

Exemplo – transição condicional de estados

```
PRESENT 'b'01
   IF INA NEXT 'b'10;
   IF !INA NEXT 'b'11;
```

cujo resultado de compilação seria:

```
Q1.D = !Q1 & Q0;
Q0.D = !Q1 & Q0 & !INA;
```

🕮 **Nota**: A ordem em que são escritas as condições não tem qualquer influência sobre as expressões finais das variáveis de estado. As expressões mais restritivas são avaliadas em primeiro lugar.

Descrição de uma saída incondicional síncrona

A definição de saídas é feita com a palavra **OUT**. O formato de descrição de saídas incondicionais síncronas é o seguinte:

```
PRESENT state_n
   NEXT state_m  OUT [!]var... OUT [!]var;
```

em que *state_n* e o estado presente, *state_m* é o estado seguinte e *var* é o nome da variável de saída, que pode ser negada com o operador '!'.

Exemplo – transição condicional de estados

```
PRESENT 'b'01
   NEXT 'b'10 OUT Y OUT !Z ;
```

define que no estado presente 1, na transição deve passar para o estado 2 e activar a saída Y e desactivar a saída Z, ou seja, teríamos as expressões seguintes:

```
Y.D  =  !Q1  &  Q0;

Z.D = 0;
```

Descrição de uma saída condicional síncrona

No caso de saídas condicionais, a variável de saída depende de uma expressão, de acordo com o formato seguinte:

```
PRESENT state_n
   IF expressão NEXT state_m OUT [!]var...OUT [!] var;
              ...
   IF expressão NEXT state_k OUT [!]var...OUT [!] var;
      [ [DEFAULT] NEXT state_l OUT [!]var;]
```

em que *state_n* é o estado presente, *state_m*, *state_k* e *state_l* são estados seguintes e *var* é o nome da variável de saída e *expressão* é uma qualquer expressão (ver definição de expressão na secção 4.2.3.1).

Exemplo – máquina de estados com saídas condicionais síncronas

```
PRESENT 'b'01
   IF INA NEXT 'b'10 OUT Y;
   IF !INA NEXT 'b'11 OUT Z;
```

indica que no estado presente 1 aquando da transição deve passar para o estado 2 e activar a saída Y se INA for igual a '1'. Caso contrario, deve transitar para o estado 3 e activar a saída Z. Após compilação, teríamos para as saídas:

```
Y.D = !Q1 & Q0 & INA;
Z.D = !Q1 & Q0 & !INA;
```

Descrição de saídas incondicionais assíncronas

Para saídas incondicionais assíncronas, a sintaxe é a seguinte:

```
PRESENT state_n
   OUT  var  ...  OUT  var ;
```

em que *state_n* é o estado presente e *var* é o nome da variável de saída.

Exemplo – máquina de estados com saídas incondicionais assíncronas

```
PRESENT 'b'01
   OUT Y  OUT Z;
```

define que as saídas devem ser activadas incondicionalmente no estado 1. Neste caso, o compilador gera as equações seguintes.

```
Y  =  !Q1  &  Q0;
Z  =  !Q1  &  Q0;
```

Descrição de saídas condicionais assíncronas

No caso de saídas condicionais assíncronas, temos a sintaxe seguinte:

```
PRESENT state_n
   IF  expressão  OUT  var ... OUT var;
                .
                .
   IF  expressão  OUT  var ... OUT var;
      [DEFAULT   OUT  var ... OUT var;]
```

em que *state_n* é o estado presente, *var* é o nome da variável de saída e *expressão* é uma qualquer expressão (ver secção 4.2.3.1).

Exemplo – máquina de estados com saídas condicionais assíncronas

```
PRESENT 'b'01
   IF INA OUT Y;
   IF !INA OUT Z;
```

declara duas variáveis condicionais assíncronas cujo resultado de compilação seria:

```
   Y = !Q1 & Q0 & INA;
   Z = !Q1 & Q0 & !INA;
```

4.2.4 Tópicos Avançados

4.2.4.1 Descrição de funções lógicas como uma condição

A linguagem CUPL inclui a construção CONDITION para descrição de funções lógicas a um nível de abstracção mais elevado que o conseguido com a utilização de equações booleanas. A sintaxe é a seguinte:

```
CONDITION  {
   IF expressão0 OUT  var ;
        .
        .
   IF expressãon OUT  var ;
   DEFAULT  OUT  var ;
}
```

em que *expressãon* é uma qualquer expressão e *var* é o nome de uma variável associada a um pino ou uma lista de variáveis.

Exemplo – descrição de um descodificador 2×4 com a sintaxe CONDITION

Consideremos a especificação de um descodificador 2×4 com entradas A e B, controladas por uma entrada de *enable*, e saídas de dados Y0 a Y3. A descrição CUPL com a construção CONDITION seria a seguinte:

```
CONDITION {
   IF enable & !B & !A out  Y0 ;
   IF enable & !B & A out  Y1 ;
   IF enable & B & !A out  Y2 ;
   IF enable & B & A out  Y3 ;
}
```

no exemplo apenas se especificou a condição associada a cada uma das saídas.

4.2.4.2 Funções definidas pelo Utilizador

Em CUPL é possível encapsular alguma lógica numa função que depois pode ser incluída em expressões lógicas através da palavra FUNCTION com a sintaxe seguinte:

```
FUNCTION nome ([parâmetro0,....,parâmetron])

   {corpo da função}
```

em que *nome* é um qualquer nome válido, com regras idênticas às usadas na construção de nomes de variáveis, *parâmetron* é uma variável opcional de referência a outras variáveis e o *corpo da função* é uma combinação de equações lógicas, tabelas de verdade, máquinas de estado, condições ou funções de utilizador.

Exemplo – definição do operador OR com uma função

```
FUNCTION  or(in1, in2) {
   or  =  in1  #  in2;
}
```

A função *or* pode agora ser usada numa expressão:

```
Y  =  or(A,B) ;
```

resultando na equação lógica seguinte:

```
Y  =  A  # B;
```

🕮 **Nota**: as funções devem ser definidas antes de serem referenciadas e não podem ser recursivas).

4.2.4.3 Comandos de pré-processamento

A linguagem CUPL também inclui um conjunto de comandos de pre-processamento (ver tabela 9), tal como acontece, por exemplo, com a linguagem C. Os comandos de pré-processamento permitem incluir opções de compilação, facilitam a estruturação e manipulação das variáveis ou sinais e facilitam a descrição de muitas funções lógicas.

$DEFINE	$IFDEF	$UNDEF
$ELSE	$IFNDEF	$REPEAT
$ENDIF	$INCLUDE	$REPEND
$MACRO	$MEND	

Tabela 9 – Comandos de pré-processamento do CUPL

$DEFINE

O comando $DEFINE funciona do mesmo modo que na linguagem C, ou seja, substitui uma *string* de caracteres por um operador, um número ou um símbolo, de acordo com o formato seguinte:

```
$DEFINE argumento1 argumento2
```

em que *argumento1* é o nome de uma variável ou um carácter ASCII especial e *argumento2* é um operador válido, um número ou um nome de uma variável.

Este comando força a que em todos os pontos do texto de descrição (após a definição #DEFINE) em que aparecer *argumento1* se substitua por *argumento2*. A substituição ocorre antes de o ficheiro ser enviado para o compilador.

Exemplo – aplicação do comando $DEFINE na definição de números

```
$DEFINE ON 'b'1
$DEFINE OFF 'b'0
$DEFINE MemPos 'h'FFFF
```

Os dois primeiros $DEFINE definem as palavras ON e OFF com os valores binários '1' e '0', respectivamente. O terceiro exemplo define uma palavra com um valor hexadecimal.

Exemplo – utilização do comando $DEFINE na definição de novos operadores lógicos

```
$DEFINE {     /*    Define novos símbolos
$DEFINE }     */    de comentários
$DEFINE /     !     Define novos símbolos para as operações lógicas
$DEFINE *     &
$DEFINE +     #
$DEFINE ^     $
```

$UNDEF

O comando $UNDEF repõe as definições existentes antes da aplicação do comando $DEFINE de acordo com o formato:

$UNDEF *argumento*

em que *argumento* é um argumento previamente definido com o comando $DEFINE.

$INCLUDE

O comando inclui um determinado ficheiro no código CUPL, de acordo com o formato seguinte:

$INCLUDE *ficheiro*

em que *ficheiro* é o nome de um ficheiro na directoria actual.

Pode ser usado, por exemplo, para incluir um ficheiro que contenha definições comummente usadas pelo projectista. Por exemplo, para incluir o ficheiro myDefinition.txt, teríamos:

$INCLUDE myDefinition.txt

$IFDEF

O comando permite definir secções do código cuja compilação depende do valor de um argumento, de acordo com o formato seguinte:

$IFDEF *argumento*

em que *argumento* é um argumento previamente definido com o comando $DEFINE.

Caso o argumento tenha sido previamente definido, então tudo o que surgir após o comando $IFDEF é compilado até encontrar um comando $ELSE ou $ENDIF. Caso contrário, o que estiver entre o comando $IFDEF e um comando $ELSE ou $ENDIF não é compilado.

$IFNDEF

Este comando funciona de modo contrário ao $IFDEF

```
$IFNDEF argumento
```

Se o argumento não tiver sido definido, todo o código entre o $IFNDEF e um comando $ELSE ou $ENDIF é compilado. Caso contrário, não é compilado.

O comando pode ser usado, por exemplo, para suportar a definição de duas descrições alternativas no mesmo ficheiro. Escolher uma ou outra corresponde simplesmente a definir ou não um determinado argumento.

$ENDIF

O comando $ENDIF termina uma secção de código iniciada com os comandos $IFDEF ou $IFNDEF, de acordo com a sintaxe:

```
$ENDIF
```

A definição de secções de código que são condicionalmente compiladas podem ser misturadas.

Exemplo – mistura de comandos $IFDEF

```
$IFDEF  prototype_1
pin 1   = set;
pin 2   = reset;
   $IFDEF     prototype_2
   pin 3 = enable;
   pin 4 = disable;
   $ENDIF
pin 5   = run;
pin 6   = halt;
$ENDIF
```

Os pinos 3 e 4 apenas são definidos se os argumentos *prototype_2* e *prototype*_1 tiverem sido definidos, enquanto que para os restantes pinos basta que o argumento *prototype*_1 tenha sido definido.

$ELSE

O comando $ELSE complementa a condição de compilação definida pelos comandos $IFDEF ou $IFNDEF, de acordo com a sintaxe seguinte:

```
$ELSE
```

Exemplo – utilização do comando $ELSE

No exemplo, a secção associada ao comando $IFDEF não é compilada porque o argumento controlo não foi definido. Neste caso, a secção definida pelo $ELSE é compilada.

```
$IFDEF controlo
   pin 1 = memreq;
   pin 2 = ioreq;
$ELSE
   pin 1 = ioreq;
   pin 2 = memreq;
$ENDIF
```

$REPEAT e $REPEND

Os comandos $REPEAT e $REPEND têm uma funcionalidade similar à do FOR da linguagem C, de acordo com o formato seguinte:

```
$REPEAT index=[number1,number2,…numbern]
   código_a_ser_repetido
$REPEND
```

em que *numbern* pode ser um número entre 0 e 1023

O comando duplica o *código_a_ser_repetido* desde o *número1* até ao *númeron*. O corpo a ser repetido pode ser qualquer código CUPL e podem ser usadas operações aritméticas

🕮 **Nota**: as operações aritméticas devem ser escritas entre chavetas.

Exemplo – descrição de um descodificador de 3 por 8 com o comando $REPEAT

```
FIELD sel = [in2..0]
$REPEAT i = [0..7]
   !out{i} = sel:'h'{i} & enable;
$REPEND
```

Neste exemplo, o índice i varia entre 0 e 7. O resultado após pré-processamento será:

```
FIELD sel = [in2..0];
   !out0 = sel:'h'0 & enable;
   !out1 = sel:'h'1 & enable;
   !out2 = sel:'h'2 & enable;
   !out3 = sel:'h'3 & enable;
   !out4 = sel:'h'4 & enable;
   !out5 = sel:'h'5 & enable;
   !out6 = sel:'h'6 & enable;
   !out7 = sel:'h'7 & enable;
```

Exemplo – descrição de um contador de quatro bits com o comando $REPEAT

```
FIELD count[out3..0]
SEQUENCE count {
      $REPEAT i = [0..15]
         PRESENT S{i}
                  IF advance & !reset NEXT S{(i+1)%(15)};
               IF reset NEXT S{0};
               DEFAULT NEXT S{i};
      $REPEND
}
```

Após pre-processamento teríamos o código seguinte:

```
FIELD count[out3..0]
SEQUENCE count {
   PRESENT S0
            IF advance & !reset NEXT S1;
         IF reset NEXT S{0};
         DEFAULT NEXT S0;
   PRESENT S1
            IF advance & !reset NEXT S2;
         IF reset NEXT S{0};
         DEFAULT NEXT S1;
…
   PRESENT S15
            IF advance & !reset NEXT S0;
         IF reset NEXT S{0};
         DEFAULT NEXT S15;
}
```

$MACRO e $MEND

Os comandos $MACRO e $MEND definem macros, de acordo com o formato seguinte:

```
$MACRO nome argumento1 argumento2...argumenton
   corpo-da-macro
$MEND
```

Exemplo – descrição de um descodificador com um número variável de bits com o comando $MACRO

```
$MACRO descodificador  bits  X  Y enable;
   FIELD select = [Y{bits-1}..0];
   $REPEAT i = [0..{2**(bits-1)}]
         !X{i} = select:'h'{i} & enable;
   $REPEND
$MEND
```

A macro seria chamada com **descodificador(3, out, in, enable)**; para criar um descodificador de 3 bits.

Após o pré-processamento, seria gerado o código seguinte:

```
FIELD sel = [in2..0];
   !out0 = sel:'h'0 & enable;
   !out1 = sel:'h'1 & enable;
   !out2 = sel:'h'2 & enable;
   !out3 = sel:'h'3 & enable;
   !out4 = sel:'h'4 & enable;
   !out5 = sel:'h'5 & enable;
   !out6 = sel:'h'6 & enable;
   !out7 = sel:'h'7 & enable;
```

O projectista pode criar um ficheiro separado com extensão .a com definições de macros. Depois, basta usar o comando $INCLUDE para incluir as macros no seu icheiro de descrição do circuito hardware.

5 Ambiente WINCUPL

O WINCUPL é um ambiente integrado de desenvolvimento de circuitos digitais com base nas ferramentas e no compilador CUPL.

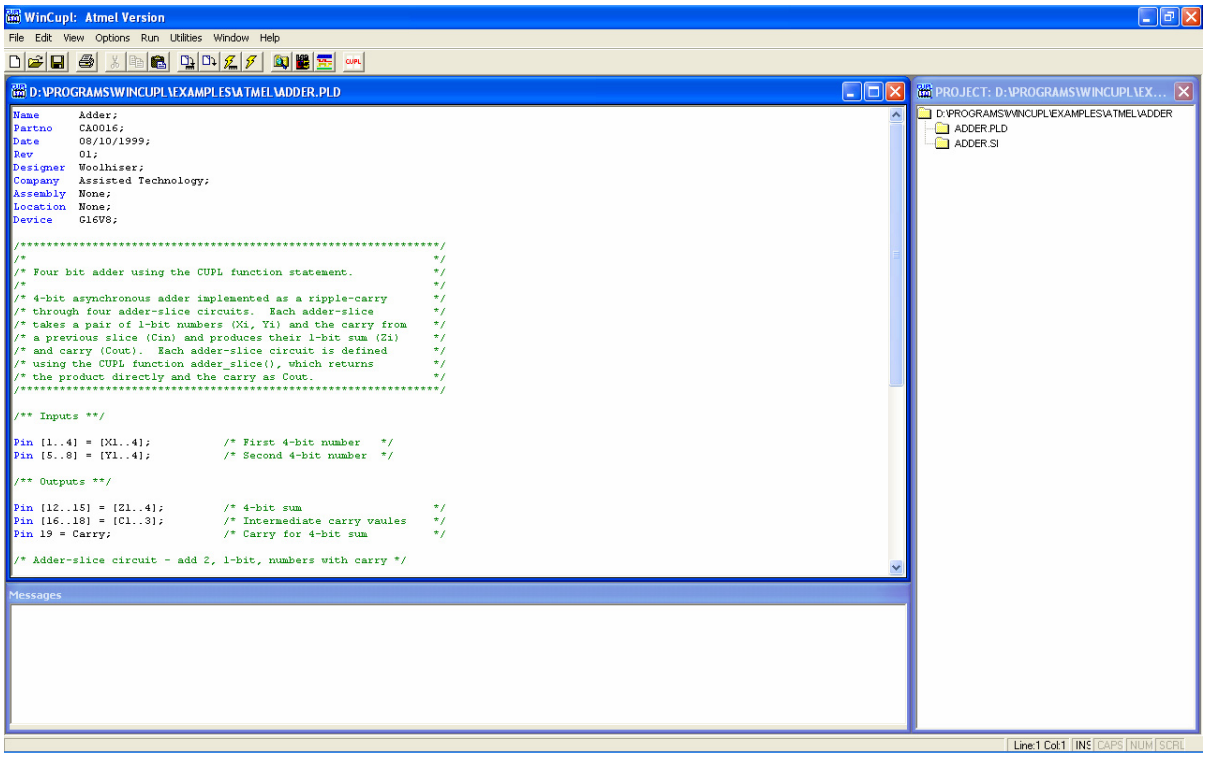

Figura 2 – Ambiente de desenvolvimento CUPL

No ambiente WINCUPL (ver figura 2) é possível editar o ficheiro de descrição do circuito, compilá-lo e simulá-lo. De entre vários ficheiros gerados pelo compilador, é gerado o ficheiro de programação do dispositivo lógico programável alvo.

O ambiente tem uma zona de edição de texto, uma zona de apresentação de mensagens de erro e avisos e uma zona com ligações para os ficheiros gerados pelo compilador e pelo simulador.

A partir dos menus da zona superior da janela do ambiente é possível aceder a todas as funcionalidades do ambiente. Os menus **File, Edit, View, Utilities, Window** e **Help** são auto-explicativos, pelo que não serão detalhados neste documento. Nas secções seguintes, descreve-se em pormenor as opções associadas aos menus **Options** e **Run**.

5.1 Opções de Projecto

O menu **Options** é usado para lançar as janelas de opções do compilador, do simulador, dos dispositivos e do ambiente WinCUPL. Consideremos cada um deles em particular.

Options->Compiler

O menu apresenta as opções relativas ao compilador (ver figura 3).

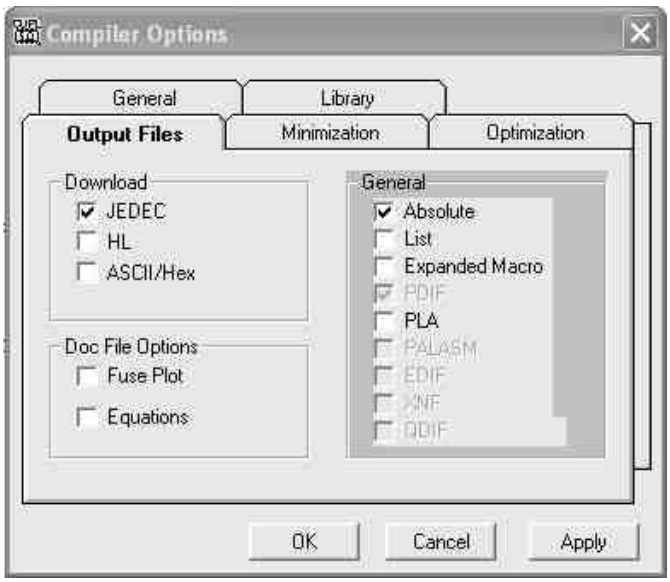

Figura 3 – Menu de opções do compilador

O tab **Output Files** é usado para escolher quais os ficheiros de saída a serem gerados pelo compilador, nomeadamente:

- *JEDEC* – gera um ficheiro ASCII com extensão .jed. O nome do ficheiro é obtido a partir da secção *name* do cabeçalho da descrição CUPL. Se quiser forçar que o nome do ficheiro seja criado com base no nome do ficheiro da descrição (.pld) deverá activar a opção correspondente no Tab **General**;
- *HL* – gera um ficheiro com extensão .hl, que é usado apenas para os dispositivos IFL da Signetics. O nome do ficheiro também é dado pela secção *name* do cabeçalho da descrição CUPL;
- *ASCII/Hex* – gera um ficheiro ASCII com extensão .hex. O formato é apenas usado em PROM. O nome usado segue a mesma convenção dos ficheiros anteriores;
- *Fuse Plot* – gera um diagrama dos fusíveis de dispositivos PAL® no ficheiro com extensão .doc;
- *Equations* - gera as equações lógicas a serem programadas no dispositivo PAL® no ficheiro com extensão .doc;

- *Absolute* – gera um ficheiro com extensão .abs para ser usado com o simulador lógico CSIM;
- *List* – gera um ficheiro com a extensão .lst que contém a lista de erros resultantes da compilação;
- *Expanded Macro* – gera um ficheiro com extensão .mx que contém uma lista expandida de todas as macros usadas na descrição hardware;
- Gera um ficheiro com extensão .pla para ser usado nas ferramentas Berkeley PLA;

O tab **Minimization** é usado para escolher o método de minimização a ser usado pelo compilador (ver lista e descrição dos métodos disponíveis na secção 3.1).

O tab **Optimization** é usado para escolher o método de optimização a ser usado pelo compilador (ver lista e descrição dos métodos disponíveis na secção 3.1).

O tab **General** é usado para escolher opções genéricas do compilador, nomeadamente:

- *Secure Device* – adiciona automaticamente ao ficheiro JEDEC código que permite ao programador do dispositivo alterar o fusível de segurança durante a programação (nem todos os programadores suportam esta opção);
- *Deactivate Unused OR Terms* – em dispositivos IFL, as entradas não usadas do OR de saída são mantidas ligadas, permitindo que novos termos sejam facilmente adicionados. No entanto, esta configuração aumenta o tempo de propagação da entrada para a saída. A opção força a que as entradas não usadas sejam desactivadas para reduzir este tempo de propagação;
- *Simulate* – cria um ficheiro com extensão .abs e executa automaticamente o simulador lógico, CSIM;
- *One hot bit State Machine* – gera as máquina de estados com uma codificação *one-hot*;
- *JEDEC name = PLD name* – esta opção força a que o nome do ficheiro seja criado a partir do ficheiro da descrição (.pld) em vez de ser obtido a partir da secção *name* do cabeçalho da descrição CUPL.

Options->Devices

O menu apresenta uma janela onde se pode escolher o dispositivo alvo a partir de uma lista de dispositivos (ver figura 4).

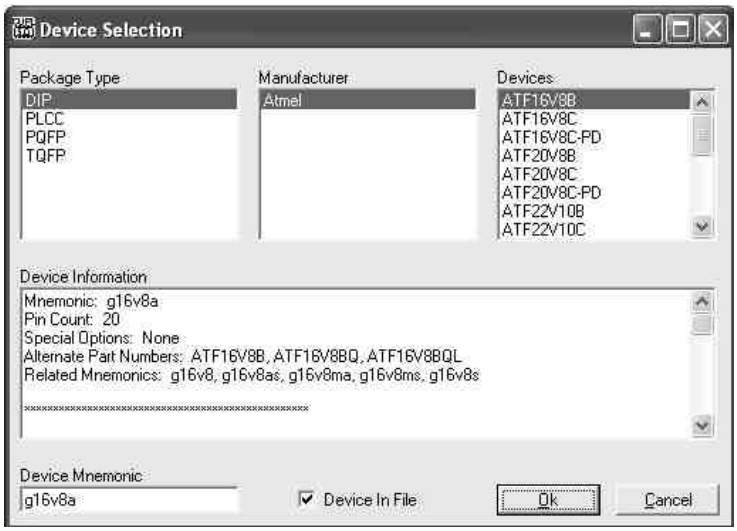

Figura 4 – Menu de opções de escolha do dispositivo

Options->Simulator

O menu apresenta uma janela com opções relativas ao simulador (ver figura 4), nomeadamente:

Figura 5 – Menu de opções relativas ao simulador

- *Listing File* **–** cria um ficheiro de saída com os resultados da simulação;
- *Append Vectors* – adiciona os vectores de teste gerados pela simulação ao ficheiro JEDEC;
- *Display Results* – apresenta os resultados da simulação num gráfico.

Options->WinCUPL

O menu apresenta uma janela com várias opções relativas ao ambiente WinCUPL (ver figura 5). As opções são comuns a ferramentas baseadas em ambiente de janelas, pelo que dispensa explicação detalhada.

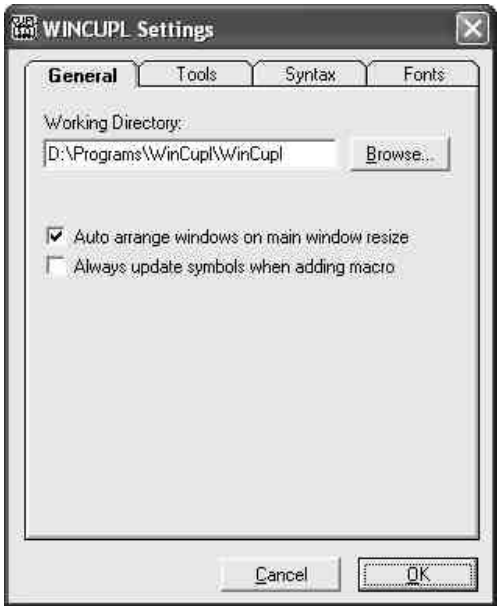

Figura 6 – Menu de opções relativas ao ambiente WinCUPL

5.2 Opções de Execução

O menu **Run** é usado para invocar o compilador e o simulador. Qualquer uma destas acções pode ser realizada de forma independente ou dependente do tipo de dispositivo lógico alvo, nomeadamente:

Device Dependent Compile – compila a descrição tendo em conta o dispositivo alvo especificado através do cabeçalho da descrição ou através das opções de compilação (ver secção 5.1);

Device Independent Compile – compila a descrição usando um dispositivo virtual. Caso tenha sido especificado algum dispositivo, este será ignorado. O mesmo resultado pode ser obtido à custa do comando anterior, mas especificando o dispositivo como sendo virtual;

Device Dependent Simulation – simula a descrição tendo em conta o dispositivo alvo especificado através do cabeçalho da descrição ou através das opções de compilação (ver secção 5.1);

Device Independent Simulation – simula a descrição usando um dispositivo virtual. Caso tenha sido especificado algum dispositivo, este será ignorado. O mesmo resultado pode ser obtido à custa do comando anterior, mas especificando o dispositivo como sendo virtual.

5.3 Simulador

O simulador não será abordado nesta versão do manual.

6 Exemplo de Aplicação do WinCUPL

Neste capítulo, descreve-se um exemplo de aplicação do ambiente WinCUPL no projecto de um somador de 4 bits.

O projecto será feito de acordo com os passos seguintes:

Passo 1 – Especificação e proposta de solução

Passo 2 – Descrição do circuito em linguagem CUPL

Passo 3 – Escolha do dispositivo lógico alvo

Passo 4 – Compilação da descrição

Passo 5 – Criação de um ficheiro de simulação e simulação do circuito

Passo 7 – Programação do dispositivo

6.1 Especificação e Proposta de Solução

O primeiro passo de um projecto hardware consiste na especificação e formulação do problema. Neste passo, são usadas representações auxiliares (e.g., diagrama de blocos) por forma a determinar uma arquitectura que cumpra com os requisitos do projecto. Esta fase de projecto é fundamental para o sucesso do mesmo, pois identifica e define os principais blocos do circuito e tomam-se opções com consequências ao longo de todo o projecto.

No caso do exemplo em estudo, o circuito é bastante simples e optámos por realizá-lo com uma série de somadores completos (ver figura 7).

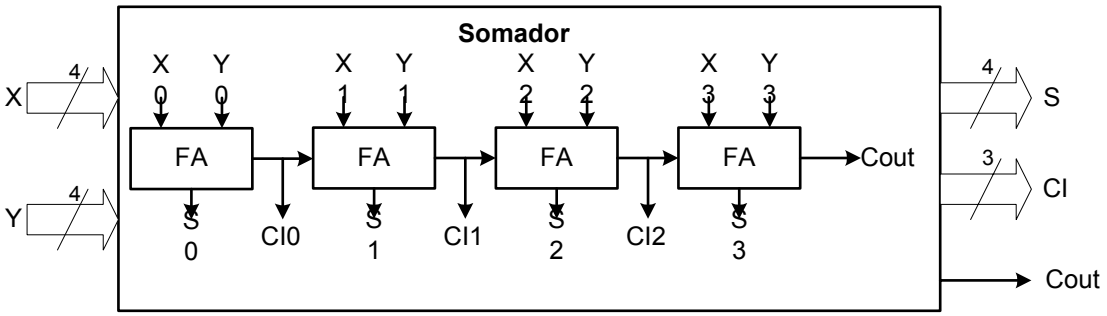

Figura 7 – diagrama de blocos do somador

O diagrama tem duas entradas X e Y, de quatro bits cada, uma saída com o resultado da soma, *Soma*, uma saída com os arrastos intermédios, *CI*, e uma saída com o arrasto do quarto bit, *Cout*. O somador realiza a operação S = X + Y.

6.2 Descrição do Circuito em Linguagem CUPL

No passo seguinte, procede-se à descrição da arquitectura obtida no passo anterior usando a linguagem CUPL. Para tal, começamos por criar um ficheiro com a extensão .pld. O ficheiro pode ser criado num qualquer editor de texto ou dentro do ambiente WinCUPL. No primeiro caso, basta depois ser aberto dentro do WinCUPL. Caso se pretenda criar o ficheiro no editor do WinCUPL basta criar um novo ficheiro. Na criação deste ficheiro, o WinCUPL orienta o utilizador na criação do cabeçalho e da secção de declarações. Assim, após criar um novo ficheiro no menu **File**, surge uma janela com os campos do cabeçalho com valores por defeito (ver figura 8).

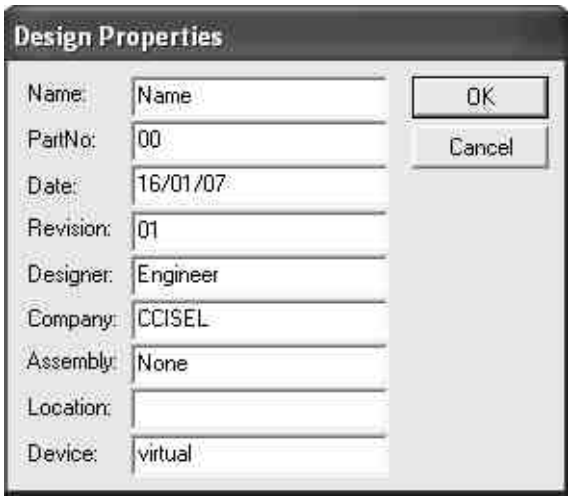

Figura 8 – janela de criação do cabeçalho

O projectista pode deixar os parâmetros por feito e alterá-los posteriormente no ficheiro. Em seguida, surgem mais algumas janelas em que o projectista pode indicar o número de entradas e de saídas ou então cancelá-las. No primeiro caso, o WinCUPL adiciona automaticamente PIN à descrição. Enquanto que se o utilizador tiver cancelado todas as janelas, apenas irá criar um ficheiro com o cabeçalho.

No nosso exemplo, vamos criar automaticamente o cabeçalho e indicar dois pinos de entrada e três de saída. A descrição resultante seria:

```
Name     Somador ;
PartNo   00 ;
Date     16/01/07 ;
Revision 01 ;
Designer Engineer ;
Company  CCISEL ;
Assembly None ;
Location  ;
Device   virtual ;
```

```
/* *************** INPUT PINS *********************/
PIN     =                          ; /*                                */
PIN     =                          ; /*                                */

/* *************** OUTPUT PINS *********************/
PIN     =                          ; /*                                */
PIN     =                          ; /*                                */
PIN     =                          ; /*                                */
```

A partir do *template*, descreve-se o circuito. No caso do nosso exemplo, vamos considerar a descrição seguinte:

```
Name     Somador ;
PartNo   00 ;
Date     16/01/07 ;
Revision 01 ;
Designer Engineer ;
Company  CCISEL ;
Assembly None ;
Location  ;
Device   v750c ;

/* *************** INPUT PINS *********************/

Pin  = [X1..4];           /* First 4-bit number   */
Pin  = [Y1..4];           /* Second 4-bit number  */

/* *************** OUTPUT PINS *********************/

Pin  = [S1..4];         /* 4-bit sum                    */
Pin  = [Ci1..3];         /* Intermediate carry vaules    */
Pin  = Cout;                /* Carry for 4-bit sum          */

/* Adder-slice circuit - add 2, 1-bit, numbers with carry */

function full_adder(X, Y, Cin, Cout) {
        Cout     = Cin & X               /* Compute carry */
                 # Cin & Y
                 # X & Y;
        full_adder = Cin $ (X $ Y);    /* Compute sum */
}

/* Perform 4, 1-bit, additions and keep the final carry */

S1 = full_adder (X1, Y1, 'h'0, Ci1);     /* Initial carry = 'h'0         */
S2 = full_adder (X2, Y2,   Ci1, Ci2);
S3 = full_adder (X3, Y3,   Ci2, Ci3);
S4 = full_adder (X4, Y4,   Ci3, Cout);  /* Get final carry value        */
```

No exemplo utilizou-se uma função que descreve um somador completo. As equações de geração dos bits de soma foram realizadas com quatro somadores completos, de acordo com a arquitectura da figura 7.

6.3 Escolha do Dispositivo Alvo

A escolha do dispositivo alvo tem duas vertente de acordo com o tipo de projecto. Uma das vertentes considera que o tipo de dispositivo alvo é uma restrição de projecto, ou seja, o circuito será implementado necessariamente num determinado tipo de dispositivo. Neste caso, o projectista tem de garantir que o dispositivo tem pinos e recursos hardware suficientes para implementar o circuito. Caso contrário, o projecto não é viável ou então terá de se procurar uma solução com mais de um dispositivo lógico programável.

Uma segunda vertente de projecto consiste em projectar e desenvolver o circuito digital e depois procurar um dispositivo com recursos suficientes para o implementar.

Na realidade, em geral, o projecto acaba por ser um misto dos dois, já que determinadas opções de projecto dependem do tipo de dispositivo alvo e vice-versa (como tal, os passos 2 e 3 estão interrelacionados).

Neste exemplo de aplicação, vamos assumir que o projecto será realizado sobre um dispositivo lógico alvo definido *a priori*, em particular, vamos considerar a PAL® ATF750C. Assim, uma atribuição possível de pinos seria:

```
/* *************** INPUT PINS *********************/

Pin [1..4] = [X1..4];           /* First 4-bit number    */
Pin [5..8] = [Y1..4];           /* Second 4-bit number   */

/* *************** OUTPUT PINS *********************/

Pin [12..15] = [S1..4];         /* 4-bit sum                      */
Pin [16..18] = [Ci1..3];         /* Intermediate carry vaules     */
Pin 19 = Cout;                 /* Carry for 4-bit sum           */
```

6.4 Compilação da Descrição

Antes de iniciar a compilação da descrição, vamos escolher algumas opções de compilação, nomeadamente:

- Ficheiros de saída: escolhemos a criação do ficheiro JEDEC, para programação da PAL®, do ficheiro .doc com a apresentação das equações e o ficheiro .lst com os possíveis erros de compilação;
- Minimização: uma vez que o circuito é de complexidade reduzida não irá criar problemas quanto à sua implementação no dispositivo programável. Como tal, escolhemos o método *quick*;
- Optimização: pelas mesmas razões, e tendo em conta o tipo de dispositivo lógico alvo, não aplicamos nenhuma optimização.

Neste ponto, procedeu-se à compilação da descrição que gera os ficheiros escolhidos nas opções de compilação.

6.5 Criação de um Ficheiro de Simulação e Simulação do Circuito

As fases de simulação não serão abordadas nesta versão do documento.

6.6 Programação do Dispositivo

Finalmente, o ficheiro com extensão .jed é usado na programação do dispositivo. Para tal, deverá usar um programador e o respectivo software de programação.

7 Exemplos de Descrições Hardware em CUPL

Neste capítulo apresentam-se alguns exemplos de descrição de circuitos digitais comuns em CUPL, nomeadamente:

1. Multiplexer
2. Descodificador
3. Somador
4. Registo
5. Registo de deslocamento
6. Contador simples
7. Máquina de estados - Contador *up/down* com *load* e *reset*

Para cada um dos exemplos será apresentada uma solução de descrição sem considerar qualquer dispositivo alvo.

7.1 Multiplexer

Neste exemplo, considerou-se um multiplexer de 4×1. Uma descrição possível seria a seguinte:

```
Name            MUX;
Partno          ;
Revision        01;
Date            20/01/07;
Designer        ISEL;
Company         CCISEL;
Location        None;
Assembly        None;
Device          virtual;

/* Multiplexer 4-to-1 */

pin  = [a0..3];
pin  = [sel0..1];
pin  = y;

y =  (a0 & ([sel0..1]:0))
   # (a1 & ([sel0..1]:1))
   # (a2 & ([sel0..1]:2))
   # (a3 & ([sel0..1]:3));
```

7.2 Descodificador

Neste exemplo, considerou-se um descodificador de 3×8. Uma descrição possível seria a seguinte:

```
Name            DECODER;
Partno          ;
Revision        01;
Date            20/01/07;
Designer        ISEL;
Company         CCISEL;
Location        None;
Assembly        None;
Device          virtual;

/* Decoder 3x8 */
/* ******* PIN declaration ****** */

pin   = [in0..2];
pin   = enable;
pin   = [out0..7];

FIELD sel = [in0..2];


/* ******* Functional declaration ****** */

!out0 = sel:0 & enable;
!out1 = sel:1 & enable;
!out2 = sel:2 & enable;
!out3 = sel:3 & enable;
!out4 = sel:4 & enable;
!out5 = sel:5 & enable;
!out6 = sel:6 & enable;
!out7 = sel:7 & enable;
```

Poder-se-ia ter descrito o descodificador de forma mis compacta usando o comando de pré-processamento **$REPEAT**:

```
/* ******* Functional declaration ****** */

      $repeat i = [0..7]
        !out{i} = sel:'h'{i} & enable;
      $repend
```

Uma outra forma de descrição seria usando o comando **CONDITION**:

```
/* ******* Functional declaration ****** */

      condition {
        if enable & sel:0 out !out0
        if enable & sel:1 out !out1
        if enable & sel:2 out !out2
        if enable & sel:3 out !out3
        if enable & sel:4 out !out4
        if enable & sel:5 out !out5
        if enable & sel:6 out !out6
        if enable & sel:7 out !out7
      }
```

7.3 Somador

Consideremos um somador de 4 bits implementado numa configuração de *ripple-carry adder*, ou seja, com uma cadeia de somadores completos. Uma descrição possível seria a seguinte:

```
Name     Somador ;
PartNo   00 ;
Date     16/01/07 ;
Revision 01 ;
Designer Engineer ;
Company  CCISEL ;
Assembly None ;
Location  ;
Device   virtual ;


/* *************** INPUT PINS *********************/

Pin  = [X0..3];           /* First 4-bit number   */
Pin  = [Y0..3];           /* Second 4-bit number  */

/* *************** OUTPUT PINS *********************/

Pin  = [S0..3];         /* 4-bit sum                    */
Pin  = [Ci0..2];         /* Intermediate carry vaules    */
Pin  = Cout;                /* Carry for 4-bit sum          */


/* Perform 4, 1-bit, additions and keep the final carry */

S0  = (X $ Y);
Ci0 =  X0 & Y0;
S1  = Cin0 $ (X1 $ Y1);
Ci1 = Cin0 & X1 # Cin0 & Y1 # X1 & Y1;
S2  = Cin1 $ (X2 $ Y2);
Ci2 = Cin1 & X2 # Cin1 & Y2 # X2 & Y2;
S3  = Cin2 $ (X3 $ Y3);
Cout = Cin2 & X3 # Cin2 & Y3 # X3 & Y3;
```

Neste exemplo, descreveram-se directamente as funções dos quatro somadores completos. O mesmo circuito poderia ter sido descrito hierarquicamente usando uma função:

```
/* Adder-slice circuit - add 2, 1-bit, numbers with carry */

function full_adder(X, Y, Cin, Cout) {
        Cout    = Cin & X               /* Compute carry */
                # Cin & Y
                # X & Y;
        full_adder = Cin $ (X $ Y);    /* Compute sum */
}

/* Perform 4, 1-bit, additions and keep the final carry */

S1 = full_adder (X1, Y1, 'h'0, Ci1);     /* Initial carry = 'h'0
*/
S2 = full_adder (X2, Y2,   Ci1, Ci2);
```

```
S3 = full_adder (X3, Y3,   Ci2, Ci3);
S4 = full_adder (X4, Y4,   Ci3, Cout);  /* Get final carry value        */
```

Neste exemplo, criou-se uma função correspondente a um somador completo e depois interligaram-se os somadores completos recorrendo a quatro chamadas da função. Note-se que o objectivo principal da utilização de funções é o de organizar a descrição numa hierarquia para melhor descrever e compreender o circuito.

7.4 Registo

Um registo é um circuito sequencial síncrono para armazenamento de bits controlado por uma entrada de relógio. No exemplo, vamos considerar um registo de 4 bits com *enable*. A descrição da entrada de *enable* do registo depende do tipo de dispositivo alvo. Em determinados dispositivos os flip-flop incluem uma entrada de *clock enable* (extensão .ce). Nestes casos, a descrição da funcionalidade é bastante simples pois basta usar a extensão. Caso os flip-flop do dispositivo não incluam este tipo de entrada, então ter-se-á de garantir a funcionalidade com o carregamento em paralelo do valor actual do registo. Uma descrição possível assumindo flip-flop com *enable* seria:

```
Name            Register;
Partno          ;
Revision        01;
Date            20/01/07;
Designer        ISEL;
Company         CCISEL;
Location        None;
Assembly        None;
Device          virtual;

/* Example showing register and latch usage */

pin  = [D0..3];     /* Data Inputs */
pin  = Clk;        /* Global clock pin */
pin  = ClkEn;      /* Clock Enable */

/* Outputs */
pin = [Q0..3];

field output = [Q0..3];
field data = [D0..3];
/* equations */

output.d   = data;
output.ck  = Clk;     /* Global Clock pin */
output.ce  = ClkEn;   /* Clock enable */
output.ar  = !Reset;  /* Use Global Reset pin */
```

Neste exemplo, apenas tivemos de declarar as funções associadas às entradas dos quatro flip-flop do registo. Caso considerássemos um dispositivo com flip-flop sem *enable*, então teríamos a descrição seguinte:

```
Name            Register;
Partno          ;
Revision        02;
Date            20/01/07;
Designer        ISEL;
Company         CCISEL;
Location        None;
Assembly        None;
Device          virtual;

/* Example showing register and latch usage */

pin  = [D0..3];     /* Data Inputs */
pin  = Clk;        /* Global clock pin */
pin  = ClkEn;      /* Clock Enable */

/* Outputs */
pin = [Q0..3];

field output = [Q0..3];
field data = [D0..3];
/* equations */

output.d   = data & ClkEn # output & !ClkEn;
output.ck  = Clk;     /* Global Clock pin */
output.ar  = !Reset;  /* Use Global Reset pin */
```

7.5 Registo de deslocamento

Neste exemplo, consideramos um registo de deslocamento que aceita 8 bits de entrada e roda o conteúdo *n* bits para a direita. A funcionalidade é idêntica à de um *barrel shifter* registado. Uma possível descrição deste circuito seria:

```
Name      RegDesl;
Partno    0123;
Date      20/01/07;
Revision  02;
Designer  ISEL;
Company   CCISEL;
Assembly  None;
Location  None;
Device    virtual;

/* ***************  Inputs  ************ */

PIN  = clock;         /* Register Clock                */
PIN  = [S2..0];       /* Shift Count Inputs    */
PIN  = !out_enable;   /* Register Output Enable        */
PIN  = preset;        /* Set to Ones Input             */

/* ***************  Outputs  ************ */

PIN [15..22] = [Q7..0];         /* Register Outputs              */

/** Declarations and Intermediate Variable Definitions **/

field shift  = [S2..0];         /* Shift Width Field             */
```

```
field output = [Q7..0];          /* Outputs Field                */

/** Logic Equations **/

output.d  =  [Q7, Q6, Q5, Q4, Q3, Q2, Q1, Q0]  &  shift:0
          #  [Q0, Q7, Q6, Q5, Q4, Q3, Q2, Q1]  &  shift:1
          #  [Q1, Q0, Q7, Q6, Q5, Q4, Q3, Q2]  &  shift:2
          #  [Q2, Q1, Q0, Q7, Q6, Q5, Q4, Q3]  &  shift:3
          #  [Q3, Q2, Q1, Q0, Q7, Q6, Q5, Q4]  &  shift:4
          #  [Q4, Q3, Q2, Q1, Q0, Q7, Q6, Q5]  &  shift:5
          #  [Q5, Q4, Q3, Q2, Q1, Q0, Q7, Q6]  &  shift:6
          #  [Q6, Q5, Q4, Q3, Q2, Q1, Q0, Q7]  &  shift:7;

output.sp = preset;              /* synchronous preset          */
output.oe = out_enable;          /* tri-state control           */
output.ar = 'h'00;               /* asynchronous reset not used */
```

7.6 Contador simples

Neste exemplo, vamos descrever um contador de 4 bits com *reset*:

```
Name     count ;
PartNo   00 ;
Date     18/01/07 ;
Revision 01 ;
Designer Engineer ;
Company  CCISEL ;
Assembly None ;
Location  ;
Device   virtual ;

/* ***************  Inputs  ************* */

Pin 1 = clock;
Pin 2 = reset;

/* ***************  Outputs  ************* */

Pin 17 = q0;
Pin 16 = q1;

/* ***************  Logic  ************* */
qo.d = !reset & (!q0);
q1.d = !reset & (q0 & !q1);
q2.d = !reset & (q0 & q1 & !q2);
q3.d = !reset & (q0 & q1 & q2 & !q3);
```

As quatro equações de entrada dos flip-flop foram descritas à custa da contagem presente.

7.7 Máquina de Estados - Contador up/down com Load e Clear

Consideremos a descrição de um contador com modo de contagem crescente e decrescente com entrada síncrona de *clear*. Para tal, vamos usar uma descrição de máquina de estado.

```
Name                Count8;
Partno              CA0018;
```

```
Revision            02;
Date                20/01/07;
Designer            ISEL;
Company             CCISEL;
Location            None;
Assembly            None;
Device              virtual;

/**  Inputs  **/
Pin   = clk;                /* counter clock              */
Pin   = clr;                /* counter clear input        */
Pin   = dir;                /* counter direction input    */
Pin   = !oe;                /* Register output enable     */

/*  Outputs                                                 */

Pin   = [Q2..0];       /* counter outputs            */
Pin   = carry;              /* ripple carry out           */

/*  Declarations and Intermediate Variable Definitions     */
field count = [Q2..0];       /* declare counter bit field  */
$define S0 'b'000
$define S1 'b'001
$define S2 'b'010
$define S3 'b'011
$define S4 'b'100
$define S5 'b'101
$define S6 'b'110
$define S7 'b'111
field mode = [clr,dir];     /* declare field node control */
up = mode:0;                /* define count up mode       */
down = mode:1;              /* define count down mode     */
clear = mode:[2..3];         /* define count clear mode    */

/* Logic Equations */
sequence count {       /*     free running counter     */
present S0          if up         next S1;
                    if down       next S7;
                    if clear      next S0;
present S1          if up         next S2;
                    if down       next S0;
                    if clear      next S0;
present S2          if up         next S3;
                    if down       next S1;
                    if clear      next S0;
present S3          if up         next S4;
                    if down       next S2;
                    if clear      next S0;
present S4          if up         next S5;
                    if down       next S3;
                    if clear      next S0;
present S5          if up         next S6;
                    if down       next S4;
                    if clear      next S0;
present S6          if up         next S7;
                    if down       next S5;
                    if clear      next S0;
present S7          if up         next S0;
```

```
                    if down       next S6;
                    if clear      next S0;
                    out           carry;         /* assert carry output */
}
```

Poderíamos ter descrito a parte funcional do mesmo circuito de forma mais compacta usando o comando **$REPEAT**:

```
sequence count {      /*     free running counter     */
present S0
      if up       next S1;
      if down     next S7;
      if clear    next S0;
$REPEAT i=[1..7]
  present S{i}
      if up       next {(i+1)%8}
      if down     next {(i-1)%8}
      if clear    next 0;
$REPEND
}
```

8 Anexo A – Sintaxe CUPL

8.1 Elementos da linguagem CUPL

8.1.1 Variáveis

- As variáveis podem começar com um número, um carácter ou um sublinhado, mas têm de conter pelo menos um carácter;
- As variáveis são sensíveis às maiúsculas e minúsculas;
- As variáveis podem conter no máximo 31 caracteres. Todos os caracteres a mais são truncados;
- As variáveis não podem conter qualquer palavra (ver tabela A.1) ou símbolo reservados (ver tabela A.2) do CUPL.

APPEND	ASSEMBLY	ASSY	COMPANY	CONDITION
DATE	DEFAULT	DESIGNER	DEVICE	ELSE
FIELD	FLD	FORMAT	FUNCTION	FUSE
GROUP	IF	JUMP	LOC	LOCATION
MACRO	MIN	NAME	NODE	OUT
PARTNO	PIN	PINNODE	PRESENT	REV
REVISION	SEQUENCE	SEQUENCED	SEQUENCEJK	SEQUENCERS
SEQUENCET	TABLE			

Tabela A.1 – Palavras reservadas do CUPL

&	#	(	)	-
*		[	]	/
:	.	. .	/*	*/
;	,	!	‘	=
@	$	^		

Tabela A.2 – Símbolos reservados do CUPL

8.1.2 Números

Os números podem variar entre 0 e $2^{32} - 1$ (números de 32 bits) e podem ser representados em uma de quatro bases: decimal, binária, octal ou hexadecimal (ver tabela A.3).

Base	Prefixo	Exemplo
binária	‘b’ ou ‘B’	‘b’0100

octal	‘o’ ou ‘O’	‘O’723
decimal	‘d’ ou ‘D’	‘d’18967
hexadecimal	‘h’ ou ‘H’	‘h’BF3 ou BF3

Tabela A.3 – prefixos usados na identificação da base dos números

8.1.3 Listas

[*variável, variável, ..., variável*] ou

[variável*m .. n*]

8.2 Descrição CUPL

8.2.1 Cabeçalho

```
Name     xpto;
Partno   xpto;
Date     x/p/to;
Revision xp;
Designer xpto;
Company  xpto;
Assembly xpto;
Location xpto;
Device   xpto;
```

8.2.2 Declarações

PIN *número_do_pino* = [!] *variável*;

NODE [!] *var*;

PINNODE *número_nó* = [!]*var*;

FIELD *var* = [*var*, *var*, ... *var*] ;

MIN *var* [.ext] = *nível*; (ver tabela A.4).

Nível	Método de minimização
0	Sem minimização
1	Quick
2	Quine McCluskey
3	Presto
4	Expresso

Tabela A.4 – Níveis de minimização disponíveis em CUPL

8.2.3 Descrição Funcional

Equações

```
[!] variável [.ext] = expressão;

APPEND [!]variável[.ext] = expressão;

[var, var, ... var]: constante;

bit_field_var : constante;

[var, var, ... , var]:operador;
```

Tabelas de verdade

```
TABLE var_list_1 => var_list_2  {
      input_n => output_n ;
            .
            .
       input_n => output_n ;
}
```

Máquinas de Estados

```
SEQUENCE lista_variáveis_estado {
    PRESENT  estado0 atribuições ;
      .
      .
      .
    PRESENT  estado_n atribuições;
}

PRESENT state_n
     NEXT state_m ;

PRESENT state_n
   IF  expressão NEXT  state_m;
       .
       .
   IF  expressão NEXT state_1;
[DEFAULT  NEXT state_k;]

PRESENT state_n
     NEXT state_m  OUT [!]var... OUT [!]var;

PRESENT state_n
     IF expressão NEXT state_m OUT [!]var...OUT [!] var;
                  .
                  .
     IF expressão NEXT state_k OUT [!]var...OUT [!] var;
           [ [DEFAULT] NEXT state_1 OUT [!]var;]

PRESENT state_n
     OUT  var  ...  OUT  var ;
```

```
PRESENT state_n
     IF  expressão  OUT  var ... OUT var;
                 .
                 .
     IF  expressão  OUT  var ... OUT var;
           [DEFAULT   OUT  var ... OUT var;]
```

8.2.4 Tópicos Avançados

Condição

```
CONDITION  {
     IF expressão0 OUT  var ;
          .
          .
     IF expressãon OUT  var ;
     DEFAULT  OUT  var ;
}
```

Função

```
FUNCTION nome ([parâmetro0,....,parâmetron])
     {corpo da função}
```

Comandos de Pré-processamento

```
$DEFINE argumento1 argumento2

$UNDEF argumento

$INCLUDE ficheiro

$IFDEF argumento

$IFNDEF argumento

$ENDIF

$ELSE

$REPEAT index=[number1,number2,…numbern]
     código_a_ser_repetido
$REPEND

$MACRO nome argumento1 argumento2...argumenton
     corpo-da-macro
$MEND
```

9 Anexo B – Lista de Erros

9.1 Módulo CUPL

Lista de mensagens associadas ao módulo CUPL:

0001ck could not open: "filename" – não conseguiu abrir o ficheiro *filename*;

0002ck could not execute program: "program name" – não conseguiu prosseguir a compilação porque não encontrou o programa *program name*;

0003ck could not find PATH in ENVIRONMENT – a variável PATH não está declarada;

0004ck could not find LIBCUPL in ENVIRONMENT – a atribuição à variável LIBCUPL não foi declarada no *ENVIRONMENT* do sistema operativo;

0005ck could not find program: "program name" – não conseguiu encontrar o programa *program name*;

0006ck insufficient memory to execute program: "filename" – o istema não tem memória suficiente para executar o programa;

0007ck invalid flag: "option flag" – a *option flag* usada na linha de comandos não é válida;

0008ck out of memory: "condition" – já não existe espaço livre na memória reservada ao CUPL;

0009ck file read error, unexpected end of file: "filename" – ocorreram eros na leitura do ficheiro (pode estar corrompido, etc.);

0010ck Fitter could not fit design – o dispositivo alvo não tem recursos suficientes para implementar o circuito descrito;

0011ck Fatal fitter error during processing – ocorreu um erro durante o mapeamento do circuito nos recursos do dispositivo lógico;

0012ck invalid library access key – a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0013ck invalid library interface – a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0014ck bad library file: "filename" – a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0015ck device not in library: "device" – o dispositivo especificado não existe na biblioteca de dispositivos;

0016ck target device not specified – o projectista não especificou o dispositivo alvo;

10xxck program error: "specifics" – problema de interface com o sistema operativo.

9.2 Módulo CUPLX

Lista de mensagens associadas ao módulo CUPLX:

0001cx could not open: "filename" – não conseguiu abrir o ficheiro *filename*;

0002cx could not execute program: "program name" – não conseguiu continuar a compilação porque não encontrou o programa *program name*;

0003cx no label given for command – um dos comandos de pre-processamento $DEFINE, $UNDEF, $IFDEF, or $IFNDEF foi usado sem argumento;

0004cx already defined: "label" – a *label* já foi previamente definida com o comando $DEFINE;

0005cx string error – a *label* de um comando de pre-processamento excedeu o tamanho máximo;

0006cx $else without $ifdef – o comando $ELSE não foi seguido de $IFDEF ou $IFNDEF;

0007cx $endif without $ifdef - o comando $ENDIF não foi precedido de $IFDEF ou $IFNDEF;

0008cx $ifdef nesting too deep – os níveis de $IFDEF encaixados excederam o máximo de doze;

0009cx missing $endif - o comando $IFDEF não foi seguido de $ENDIF;

0010cx invalid preprocessor command: "$command" – comando de pre-processamento inválido;

0011cx disk write error: "filename" – o CUPLX não conseguiu gravar o ficheiro *filename*;

0012cx out of memory: "condition" - já não existe espaço livre na memória alocada ao CUPL;

0013cx illegal character: "hex value" – o CUPLX encontrou um carácter ASCII não utilizável na descrição CUPL;

0014cx unexpected symbol:"symbol" - o CUPLX encontrou um símbolo que não existe porque não faz parte da sintaxe ou porque está mal escrito;

0015cx Repeat nesting too deep – o nível de níveis encaixados do $REPEAT excedeu o máximo de 2;

0016cx duplicate Macro function name:"function" – o nome da macro já foi previamente usado;

0017cx missing Macro name – foi definida uma macro sem nome;

0018cx incorrect number of parameters – o número de parâmetros da macro não está de acordo com o número de parâmetros da chamada à macro;

0019cx out of range – o índice da variável excedeu o valor 1023;

0020cx internal stack overflow – o CUPLX não conseguiu tratar uma expressão matemática por ser demasiado complexa;

0021cx expression contains undefined symbol: "symbol" – a expressão contém um símbolo indefinido;

0022cx invalid library access key - a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0023cx invalid library interface - a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0024cx bad library file: "library" - a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0025cx unexpected end-of-file – o CUPLX chegou inesperadamente ao fim do ficheiro;

0026cx reached end-of-file before ending comment – um determinado comentário não foi terminado após atingir o fim do ficheiro;

0027cx invalid syntax for preprocessor command: "$command" – os comandos $REPEAT ou $MACRO foram usados incorrectamente;

10xxcx program error: "specifics" - problema de interface com o sistema operativo.

9.3 Módulo CUPLA

Lista de mensagens associadas ao módulo CUPLA:

0001ca could not open: "filename" - não conseguiu abrir o ficheiro *filename*;

0002ca invalid number: "number" – o número *number* foi usado num sítio errado ou um erro sintáctico anterior provocou que o número esteja no sítio errado;

0003ca invalid library access key - a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0004ca invalid library interface - a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0005ca bad library file: "library" - a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0006ca device not in library: "device" - o dispositivo especificado não existe na biblioteca de dispositivos;

0007ca invalid syntax: "symbol" - o CUPLX encontrou um símbolo que não existe porque não faz parte da sintaxe ou porque está mal escrito;

0008ca too many errors – o CUPLA encontrou mais de 30 erros;

0009ca missing symbol: "symbol" – o símbolo *symbol* não existe;

0010ca vector too wide – foi declarada uma lista de variáveis que excede o número máximo de 50 elementos;

0011ca expression already assigned to: "variable" – a variável já foi atribuída a uma outra expressão;

0012ca vector size mismatch – o número de membros da lista do lado esquerdo da equação não é o mesmo da lista do lado direito;

0013ca undefined function: "function" – a função *function* não existe;

0014ca variable already declared: "variable" – a variável já foi declarada previamente;

0015ca out of memory: "condition" - já não existe espaço livre na memória reservada ao CUPL;

0016ca invalid number of function arguments: "number" – o número de argumentos de uma função definida pelo utilizador não está de acordo com o número de argumentos da chamada à função;

0017ca disk write error: "filename" – o CUPLA não conseguiu guardar o ficheiro;

0018ca intermediate var not assigned an expression: "variable" – uma determinada variável foi usada como entrada de uma expressão sem lhe ter sido atribuída uma expressão (verifique se o nome da variável está escrito correctamente);

0019ca indexed and non-indexed vars in range or match expression – uma lista contém variáveis com e sem indexação. O compilador não consegue determinar a posição relativa das variáveis;

0020ca index too large for range or match operation – o índice de uma variável numa lista ou num campo excede o intervalo de valores;

0021ca header item already declared – uma das palavras do cabeçalho foi declarada mais que uma vez;

0022ca missing header item(s) – falta pelo menos uma das palavras do cabeçalho;

0023ca invalid range arguments: always true (in range) – foi declarado um intervalo que é sempre verdadeiro. Por exemplo, [0000..FFFF], se considerarmos 16 bits;

0024ca range or match number larger than variable list – o intervalo excede a largura do campo a que está a ser aplicado. Os valores que estiverem a mais são ignorados;

0025ca range minimization error – o intervalo é sempre falso porque nenhum dos bits do intervalo está activo;

0026ca invalid table statement – uma entrada está com mais de uma saída;

0027ca invalid present state number – o número usado na especificação do estado é inválido;

0028ca invalid next state number - o número usado na especificação do estado seguinte é inválido;

0029ca invalid flip-flop type for sequence statement: "type" – os flip-flop existentes no dispositivo não suportam a implementação da máquina de estados descrita;

0030ca intermediate dependent on itself: "variable" – a variável *variable* foi usada numa expressão de definição da mesma variável;

0031ca invalid minimization level: "level" – o nível de minimização especificado é inválido;

0032ca invalid next state: "hex number" - o número usado na especificação do estado seguinte é inválido;

0033ca multiple asynchronous defaults for state: "hex number" – existe mais de uma condição *default* assíncrona por estado (só pode existir uma por estado);

0034ca multiple synchronous defaults for state: "hex number" - existe mais de uma condição *default* síncrona por estado (só pode existir uma por estado);

0035ca multiple unconditional statements for state: "hex number" - existe mais de uma condição síncrona incondicional por estado (só pode existir uma por estado);

0036ca device does not support synchronous state machines – o dispositivo alvo não suporta a implementação de máquinas de estado porque não tem recursos para tal;

0037ca duplicate present state: "hex number" – o mesmo número de estado foi usado em mais de um comando PRESENT;

0038ca target device not specified - o projectista não especificou o dispositivo alvo;

0039ca line exceeds maximum length – a linha excedeu o tamanho máximo de 256 caracteres;

0040ca invalid or duplicate header name: "name" - uma das palavras do cabeçalho foi declarada mais que uma vez ou é inválida;

0041ca don't care(s) not allowed for decimal number, treated as 0 – as indiferenças só podem ser usadas em números representados em base 2, 8 ou 16;

0042ca range or match list completely don't cared, decoded as 0 – todas as variáveis de uma lista passaram a indiferenças. Como tal, a lista ficou vazia;

0043ca invalid GROUP name: "variable name" – um nome de grupo tem de conter uma variável;

0044ca unexpected end-of-file - o CUPLX chegou inesperadamente ao fim do ficheiro;

0045ca reached end-of-file before ending comment - um determinado comentário não foi terminado após atingir o fim do ficheiro;

0046ca invalid DeMorgan level: "number" – o nível de DeMorgan especificado não está entre 0 e 2;

0047ca vector size mismatch in comparison vector: "variable" - o número de membros da lista do lado esquerdo da equação não é o mesmo da lista do lado direito;

0048ca fixed polarity device, reset DeMorgan level to 0: "variable" – o dispositivo alvo não pode ser programado relativamente à polaridade. Como tal, não pode usar níveis diferentes de DeMorgan;

0049ca unknown DECLARE entity: "variable" -

10xxca program error: "specifics" - problema de interface com o sistema operativo.

9.4 Módulo CUPLB

Lista de mensagens associadas ao módulo CUPLB:

0001cb could not open: "filename" - não conseguiu abrir o ficheiro *filename*;

0002cb could not execute program: "program name" - não conseguiu continuar a compilação porque não encontrou o programa *program name*;

0003cb invalid file: "filename" – o ficheiro não foi criado com a versão actual do CUPL;

0004cb missing or mismatched parentheses – o número de parênteses abertos e fechados não corresponde;

0005cb invalid library access key - a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0006cb invalid library interface - a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0007cb bad library file: "library" - a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0008cb device not in library: "device" - o dispositivo especificado não existe na biblioteca de dispositivos;

0009cb pin/node "number" redeclared: "variable" – o mesmo pino ou a mesma variável foram redeclarados;

0010cb pin/node "number" invalid output: "variable" - uma variável recebe o valor de uma expressão, mas foi associada a um pino de entrada;

0011cb unknown extension: "extension" – extensão inválida para o dispositivo alvo escolhido;

0012cb pin/node "number" invalid usage: "variable" – o número de pino é inválido para o dispositivo alvo escolhido;

0013cb pin/node "number" invalid output extension or usage: "variable" – a extensão não foi correctamente usada ou é inválida para um determinado pino ou nó;

0014cb invalid input: "variable" or pin/node "number" invalid input: "variable" – a variável usada como entrada foi previamente atribuída a uma saída que não é bidireccional nem realimenta a matriz de ligações;

0015cb device not yet fully supported: "device" – o dispositivo escolhido ainda não é suportado;

0016cb no expression assigned to: "variable" – é um aviso que indica que uma determinada variável não tem expressão atribuída;

0017cb out of memory: "conditions" - já não existe espaço livre na memória reservada ao CUPL;

0018cb missing flip-flop expression for: "variable" – falta uma determinada expressão de entrada de um flip-flop JK ou SR;

0019cb DeMorgan's theorem invoked for: "variable" – o teorema de DeMorgan foi aplicado a uma determinada expressão;

0020cb invalid mix of banked outputs: "variable" – todas as saídas do m grupo de saídas devem ser usadas da mesma forma, ou seja, não se podem usar umas com registo e outras sem;

0021cb no expression allowed for: "variable" – não é permitida a atribuição de expressões a nós de *reset* ou de *set* quando a saída tiver sido especificada assincronamente;

0022cb pin/node "number" conflicting input architectures: "variable" – conflito na atribuição de números aos pinos e aos nós;

0023cb disk write error: "filename" – o CUPLB não conseguiu guardar o ficheiro *filename*;

0024cb output defined for node which does not exist: "variable" – a variável foi associada a um pino ou a um nó que não existe;

0025cb output mutually excluded by previous output: "variable" – um termo ou um termo de produto foi definido mais de uma vez;

0026cb disk read error, unexpected end of file: "filename" – o CUPLB não conseguiu ler o ficheiro *filename*;

10xxcb program error: "specifics" - problema de interface com o sistema operativo.

9.5 Módulo CUPLM

Lista de mensagens associadas ao módulo CUPLM:

0001cm could not open: "filename" - não conseguiu abrir o ficheiro *filename*;

0002cm could not execute program: "program name" - não conseguiu continuar a compilação porque não encontrou o programa *program name*;

0003cm invalid file: "filename" - o ficheiro não foi criado com a versão actual do CUPL;

0004cm out of memory: "conditions" - já não existe espaço livre na memória reservada ao CUPL;

0005cm disk write error: "filename" - o CUPLM não conseguiu guardar o ficheiro *filename*;

0006cm invalid library access key - a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0007cm invalid library interface - a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0008cm bad library file: “library” - a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0009cm device not in library: “device” - o dispositivo especificado não existe na biblioteca de dispositivos;

0010cm design too complex for this minimization level – o CUPLM excedeu o tamanho da matriz de ligações com este nível de minimização;

0011cm disk read error, unexpected end of file: “filename” – o CUPLM não conseguiu ler o ficheiro porque estava corrompido;

10xxcm program error: “specifics” - problema de interface com o sistema operativo.

9.6 Módulo CUPLC

Lista de mensagens associadas ao módulo CUPLC:

0001cc could not open: “filename” - não conseguiu abrir o ficheiro *filename*;

0002cc invalid file: “filename” - o ficheiro não foi criado com a versão actual do CUPL;

0003cc invalid library access key - a versão de CUPL não é compatível com o ficheiro de descrição do dispositivo, talvez devido a uma actualização do software;

0004cc invalid library interface - a biblioteca de dispositivos não foi criada ou não é compatível com o CUPL;

0005cc bad library file: “library” - a biblioteca de dispositivos não foi criada ou o conteúdo foi danificado;

0006cc excessive number of product terms: “variable” – o número de produtos necessário à implementação de uma expressão lógica excede a capacidade do pino de saída correspondente;

0007cc invalid download format(s) – um dos formatos especificado não é válido para o dispositivo alvo. Por exemplo, o formato .hl não é válido para a PAL®;

0008cc pin can not be used as input: “variable” – o pino não pode ser usado como entrada;

0009cc header name undefined, using no_name – falta o campo *NAME* no cabeçalho;

0010cc disk write error: “filename” – o CUPLC não conseguiu guardar o ficheiro *filename*;

0011cc out of memory: “conditions” - já não existe espaço livre na memória reservada ao CUPL;

0012cc disk read error, unexpected end of file: “filename” – o CUPLC não conseguiu ler o ficheiro *filename*;

0013cc conflicting usage of pinnode:"variable" – um termo de produto ou um termo foi definido mais do que uma vez;

0014cc unknown extension encountered: “extension” - extensão inválida para o dispositivo alvo escolhido;

0015cc invalid local feedback from “variable name” to “variable name” – a ligação de feedback usada é inválida;

0016cc exceeded number of expander product terms – o número de termos de produto necessários à implementação do circuito excedeu a capacidade do dispositivo alvo;

0017cc global feedback in local product term: "variable" – o feedback de uma variável global está a ser usado num termo de produto local;

0018cc couldn't find XILINX symbol: "symbol" – o símbolo não faz parte do dispositivo xilinx especificado;

0019cc couldn't map CUPL symbol to XILINX symbol: "symbol" – uma determinada descrição não pode ser mapeada no dispositivo Xilinx alvo;

0020cc couldn't find CUPL macro symbol: "symbol" – uma determinada macro interna não foi encontrada no ficheiro CUPL2XIL.MAP;

0021cc Error found in XILINX data file – o CUPLC encontrou um erro durante a leitura dos ficheiros de informação Xilinx;

0022cc unsupported extension: "extension" - extensão inválida para o dispositivo alvo escolhido;

0023cc incorrect number of variables in DECLARE statement: "attribute" – o número de variáveis indicadas em DECLARE não está de acordo com o definido em DECLARE:DEF;

0024cc too many XOR gates defined for output: "variable" – o mapeamento das portas XOR no dispositivo PLA não pode ser realizado porque o dispositivo não tem capacidade suficiente;

10xxcc program error: "specifics" - problema de interface com o sistema operativo.

10 Bibliografia

Atmel, *ATMEL – WinCUPL: User's Manual*

Atmel, *CUPL Reference Manual*