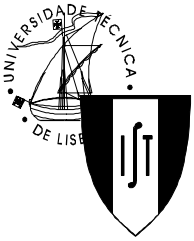
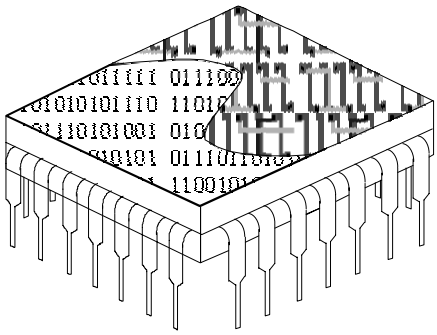


UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO



Co-Síntese de Sistemas Embebidos em Agregados de Células Lógicas

João Manuel Paiva Cardoso

(Licenciado em Eng.ª Electrónica e Telecomunicações)

Dissertação para a obtenção do grau de Mestre em
Eng.ª Electrotécnica e de Computadores

Orientador: Doutor Horácio Claúdio de Campos Neto

Júri:

Doutor Horácio Claúdio de Campos Neto
Doutor Pedro Manuel Barbosa Veiga
Doutor Alberto Manuel Ramos da Cunha

Novembro de 1996

Resumo

Hoje em dia, a maioria dos sistemas embebidos de pequeno/médio porte necessitam de soluções com um único circuito integrado, e de desenvolvimento rápido devido ao tempo de introdução no mercado ser cada vez mais pequeno. São necessários sistemas de prototipagem rápida que permitam alta fiabilidade, sem aumentar drasticamente o tempo dedicado ao teste. Estes sistemas são predominantemente baseados em processadores, com soluções puramente *software*. Contudo, a necessidade de respeitar temporalmente a aquisição/geração de eventos ou execução de tarefas, em determinadas aplicações, origina, por vezes, a utilização de soluções sobredimensionadas, pela utilização de um processador de melhor desempenho, de um co-processador, ou de um circuito de aplicação específica.

O sistema desenvolvido apresenta uma solução capaz de poder integrar no mesmo circuito integrado uma unidade central de processamento que contém memória de programa e de dados, e unidades funcionais capazes de satisfazerem as restrições temporais sem esbanjamento desnecessário de silício. A unidade central é parametrizável, com conjunto de instruções compatível com um microcontrolador comercial, e as unidades funcionais de *hardware* específico, o interface e a comunicação entre o processador e estas unidades são obtidos automaticamente a partir da especificação inicial (*software*).

O processo de mapeamento do binómio *hardware*/*software* é conduzido por directivas especificadas no *assembler* intrínseco do processador. As tarefas de automatização são realizadas pela aplicação computacional desenvolvida, que produz a especificação VHDL necessária para que um ambiente de síntese lógica obtenha o circuito representado por um conjunto de células da biblioteca utilizada.

Abstract

Nowadays, most small/medium digital embedded systems need a one-chip-solution using fast turnaround technologies, because of the small time-to-market requirements. It is necessary to use rapid systems prototyping techniques with high reliability, and without increasing the duration of the test process. These systems are usually processor based with software solutions. However, the need to satisfy timing constraints in some applications is typically solved with overdesign solutions, such as a high performance coprocessor, an ASIC, etc.

The system proposed in this thesis allows a solution based on a parameterizable core that contains data and program memory, which can be extend to use additional hardware functional units in order to satisfy time constraints without undesired silicon waste. The parameterizable processor core, which is instruction set compatible with a commercial microcontroller, and the hardware specific functional units, the interface and communication between the core and these units are obtained automatically from the source program (software specification), and can be integrated in one IC.

The processor assembler has been extended with directives that guide the hardware/software partitioning. A co-synthesis tool has been developed which outputs a VHDL specification suitable for the logic synthesis environment used, in order to implement the final system represented as a logic-level netlist using the selected library.

Palavras Chave

Microcontroladores
Sistemas Embebidos Digitais
Co-Síntese *Hardware/Software*
Co-Projecto *Hardware/Software*
Arquitectura de Processadores
Sistemas de Prototipagem Rápida
Sistemas de Tempo-Real
Agregados de Células Lógicas
VHDL
VLSI
Circuitos Integrados Digitais/Síntese Lógica

Keywords

Microcontrollers
Digital Embedded Systems
Hardware/Software Co-Synthesis
Hardware/Software Co-Project
Processors Architecture
Fast Turnaround Technology
Real Time Systems
Gate Arrays
VHDL
VLSI
Digital Integrated Circuit/Logic Synthesis

Aos meus pais, DEDICO

Agradecimentos

Agradeço ao meu orientador, Professor Horácio Neto, pelo incentivo, discussão de ideias, orientação, conselhos, e total disponibilidade demonstrada ao longo da realização da tese. Os meus agradecimentos por me ter facultado trabalhar no projecto de circuitos integrados digitais assistido por ferramentas computacionais.

Agradeço aos elementos do Grupo de Sistemas Integrados do INESC, em particular ao Eng.º José Pedro Abreu e Eng.º Paulo Flores, pela assistência sempre que por mim solicitada.

Durante os últimos dois anos foi, por mim, descurado o convívio com os meus familiares e amigos, por isso agradeço toda a compreensão e apoio demonstrado pelos meus pais, minha irmã Isabel, meu irmão Zé, o meu cunhado Albertino, Prof. Otília e Prof. Armando. Todos, desde sempre, têm contribuído para a evolução da minha pessoa.

Aos meus colegas de docência, nas disciplinas comuns na Universidade do Algarve, por terem sempre compreendido as minhas constantes deslocações.

Aos meus amigos, pois continuam a demonstrar o quanto são preciosos.

À Teresa, pela compreensão demonstrada e pelo amor que empenhou neste objectivo, como se fosse também um objectivo dela.

Lisboa, Novembro de 1996

Índice

1. INTRODUÇÃO 1

1.1 MOTIVAÇÃO 1
1.2 SOLUÇÕES TRADICIONAIS 4
1.3 A SOLUÇÃO PROPOSTA 6
1.5 ORGANIZAÇÃO DA TESE 7

2. CO-PROJECTO E CO-SÍNTESE 9

2.1 INTRODUÇÃO 10
2.2 ESPECIFICAÇÃO 13
2.3 PARTIÇÃO HARDWARE/SOFTWARE 15
2.4 PROJECTO DE PROCESSADORES COM BASE NO CÓDIGO 17
2.5 AMBIENTE DE CO-PROJECTO 19
2.5.1 PTOLEMY 19
2.6 SISTEMAS DE CO-SÍNTESE 21
2.6.1 ARQUITECTURA GERAL DA MAIORIA DOS SISTEMAS 21
2.6.2 PRISM 23
2.6.3 SISTEMA DE CO-SÍNTESE PARA SIMULAÇÃO DE CIRCUITOS DIGITAIS DESENVOLVIDO NA UNIVERSIDADE DE STANFORD 25
2.6.4 FERRAMENTA PARA SISTEMAS EMBEBIDOS DESENVOLVIDA NA UNIVERSIDADE DE BERKELEY 26
2.6.5 DESIGN ASSISTANT 27
2.6.6 COSYMA 31
2.6.7 VULCAN 38
2.7 CONCLUSÕES 44

3. MICROCONTROLADOR PIC 47

3.1 INTRODUÇÃO 47
3.2 ARQUITECTURA 48
3.2.1 INSTRUÇÕES 50
3.3 CARACTERÍSTICAS DO FICHEIRO DE REGISTOS 53

3.3.1 REGISTO F0 (ENDEREÇAMENTO DE DADOS INDIRECTO) 55
3.3.2 REGISTO F1 (CONTADOR/RELÓGIO DE TEMPO-REAL) 55
3.3.3 REGISTO F2 - PC 56
3.3.4 REGISTO DE ESTADO (F3) 57
3.3.5 REGISTO DE SELECÇÃO INDIRECTA (F4) 57
3.3.6 REGISTOS DE E/S 58
3.4 REGISTOS ESPECÍFICOS 58
3.4.1 REGISTOS TRISA E TRISB 58
3.4.2 REGISTO OPTION 58
3.4.3 REGISTO W 59
3.5 CARACTERÍSTICAS GERAIS 60
3.5.1 CICLOS POR INSTRUÇÃO 60
3.5.2 IMEDIATOS 60
3.5.3 RESET EXTERNO 60
3.5.4 RESET INTERNO 61
3.5.5 CHAMADAS A ROTINAS 61
3.5.6 TABELAS DE CONSTANTES 62
3.5.7 CÃO DE GUARDA (WDT) 62
3.5.8 O MODO SLEEP 62
3.6 SUPORTE SOFTWARE 63
3.7 CONCLUSÕES 64

4. PROJECTO DO PROCESSADOR 67

4.1 ARQUITECTURA 67
4.2 FUNCIONAMENTO GERAL 70
4.3 UNIDADE DE CONTROLO COMPLETA 71
4.3.1 CONTROLO DA PROCURA DE UMA INSTRUÇÃO 74
4.3.2 UNIDADE COMPLETA 77
4.4 FICHEIRO DE REGISTOS 78
4.4.1 O REGISTO DE ESTADO (SR) 81
4.4.2 A PILHA E O PC 81
4.4.3 O REGISTO RTCC 84
4.4.4 OS PORTOS DE ENTRADA/SAÍDA 85
4.5 A ALU 87
4.6 A UNIDADE DE MANIPULAÇÃO DE BITS 88
4.7 DESCODIFICADOR 89

4.8 O DATAPATH **91**
4.9 MEMÓRIA DE PROGRAMA E O REGISTO IR **93**
4.10 OS TEMPORIZADORES E O REGISTO OPTION **95**
4.11 DESEMPENHO **98**
4.13 UM CI DE TESTE **99**
4.14 CONCLUSÕES **103**

5. O SISTEMA DE CO-SÍNTESE **105**

5.1 O SISTEMA PROPOSTO **105**
5.2 ARQUITECTURA DO SISTEMA **107**
5.2.1 ÁREA DAS UNIDADES FUNCIONAIS 109
5.2.2 ÁREA TOTAL DO CI 110
5.3 RESTRIÇÕES TEMPORAIS **112**
5.4 TAREFAS DA APLICAÇÃO BINOMIO **113**
5.4.1 O ASSEMBLADOR 115
5.4.2 ANÁLISE DO TEMPO DE EXECUÇÃO DO CÓDIGO 116
5.4.3 MIGRAÇÃO DO SOFTWARE PARA HARDWARE 116
5.4.4 NÍVEL DE GRANULARIDADE 120
5.4.5 SINCRONIZAÇÃO E COMUNICAÇÃO SW/HW 123
5.4.5 DIRECTIVAS DA APLICAÇÃO BINOMIO 123
5.5 CO-SIMULAÇÃO **124**
5.6 EXEMPLO **125**
5.7 CONCLUSÕES **129**

6. EXEMPLO DE APLICAÇÃO **131**

6.1 FILTRO DIGITAL IIR **131**
6.1.1 RESULTADOS 132
6.2 CONCLUSÕES **137**

7. CONCLUSÕES E TRABALHO FUTURO **139**

7.1 CONCLUSÕES **139**
7.2 INVESTIGAÇÃO E DESENVOLVIMENTOS FUTUROS **140**
7.2.1 AMBIENTE DE CO-SÍNTESE 141
7.2.2 NÚCLEO DE PROCESSAMENTO 142

REFERÊNCIAS **145**

APÊNDICE A A APLICAÇÃO BINOMIO **151**

APÊNDICE B DIRECTIVAS DO BINOMIO **153**

APÊNDICE C MÓDULO DE FUNÇÕES VHDL **157**

APÊNDICE D CÓDIGO DO EXEMPLO **161**

D.1 CÓDIGO ASSEMBLER DO FILTRO IIR **161**
D.2 PARTE DA ESPECIFICAÇÃO VHDL GERADA PARA A ROM **171**
D.3 MÓDULO VHDL DA DEFINIÇÃO DE PARÂMETROS **171**
D.4 PARTE DA ESPECIFICAÇÃO VHDL QUE DESCREVE A ROM DO PROGRAMA COM COMUNICAÇÃO COM A UF **171**
D.5 ESPECIFICAÇÃO VHDL GERADA PARA A UF E INTERFACE COM O PARMIC **172**
D.6 MENSAGENS DO BINOMIO QUANDO SE REALIZA A PARTIÇÃO **175**

Índice de figuras

Figura 1.1. Vendas de microcontroladores em milhões de dólares (de 4, 8 e 16 bits) anualmente e previsões até ao ano 2000. 5
Figura 1.2. Arquitectura do sistema alvo para o ambiente de co-síntese. 6
Figura 2.1. Espaço de exploração do co-projecto. 10
Figura 2.2. Co-projecto *hardware*/*software*. 12
Figura 2.3. Diagrama de blocos da arquitectura alvo da maioria dos sistemas de co-síntese. 21
Figura 2.4. Arquitectura alvo do sistema de co-síntese PRISM. 24
Figura 2.5. Fluxo de co-síntese para simulação de sistemas digitais. 26
Figura 2.6. Fluxo de co-síntese do DESIGN ASSISTANT [13]. 29
Figura 2.7. Fluxo para a determinação das estimativas. 30
Figura 2.8. Arquitectura alvo. 30
Figura 2.9. Sistema COSYMA com o sistema de síntese de co-processadores (BBS). 32
Figura 2.10. Sistema VULCAN. 39
Figura 2.11. Modelo de grafos de um exemplo genérico. 41
Figura 3.1. Diagrama de blocos da arquitectura interna do PIC16C54. 50
Figura 3.2. Formato de cada instrução sobre o ficheiro de registos orientada ao *byte*. 51
Figura 3.3. Formato de instruções com imediatos e de controlo. 51
Figura 3.4. Formato de cada instrução sobre o ficheiro de registos orientada ao *bit*. 52
Figura 3.5. Ficheiro de registos. 54
Figura 3.6. Mapeamento de registos. 54
Figura 3.7. Diagrama de blocos do circuito que faz a atribuição do pré-escalar ao RTCC/WDT. 56
Figura 3.8. Simulador PSIM da Parallax. 64
Figura 4.1. Diagrama de blocos do processador. 68
Figura 4.2. Diagrama temporal da execução de uma instrução. 70
Figura 4.3. Diagrama de blocos da unidade de controlo completa. 72
Figura 4.4. Máquina de controlo mestre e a ligação à unidade de controlo geral e ao *datapath*. 72
Figura 4.5. Diagrama de transição de estados da máquina mestre. 73
Figura 4.6. Geração de sinais de controlo com variações em ambos os flancos do sinal de relógio. 74
Figura 4.7. Sinais de interface da máquina de controlo da procura de uma instrução (FSM1). 75
Figura 4.8. Diagrama de transição de estados da FSM1 responsável pelas instruções de saltos condicionais, pelo funcionamento normal de incremento do PC e procura da instrução. 75
Figura 4.9. Diagramas temporais que ilustram o funcionamento da FSM1 em conjunto com a FSM3. 76
Figura 4.10. Diagramas temporais que ilustram o carregamento da instrução. 76
Figura 4.11. Estrutura de acesso aos registos do ficheiro de registos. 79
Figura 4.12. Representação gráfica da área do descodificador versus o nº de registos. 80

Figura 4.13. Sinais de interface do registo SR. 81
Figura 4.14. Sinais de interface do registo PC. 82
Figura 4.15. Área do módulo (PC+pilha) versus o númro de níveis da pilha. 84
Figura 4.16. Sinais de interface do registo RTCC. 84
Figura 4.17. Sinais de interface do registo de cada porto de E/S. 85
Figura 4.18. Diagrama de blocos da ALU e blocos auxiliares do interface com os barramentos. 88
Figura 4.19. Unidade de manipulação de *bits*. 89
Figura 4.20. Sinais de interface do descodificador de instruções. 90
Figura 4.21. Diagrama de blocos do *datapath*. 92
Figura 4.22. Diagrama de blocos do acesso ao registo W. 92
Figura 4.23. Sinais de interface da unidade que contém a memória de programa e o registo IR. 93
Figura 4.24. Gráfico da área de programa versus o nº de instruções 94
Figura 4.25. Sinais de interface da máquina de sincronismo. 95
Figura 4.26. Geração do sinal PSOUT, que incrementa o registo RTCC. 95
Figura 4.27. Sinais de interface da unidade OPTION. 95
Figura 4.28. Layout do CI de teste. 101
Figura 4.29. Simulação funcional do processador. 102
Figura 4.30. Simulação funcional do processador (continuação). 102
Figura 5.1. Ambiente de co-síntese COSTLES. 107
Figura 5.2. Interface de cada unidade ao núcleo do PARMIC. 108
Figura 5.3. Sinais de interface da unidade desmultiplexadora dos sinais de controlo. 110
Figura 5.4. Tarefas da aplicação BINOMIO. 114
Figura 5.5. Restrição temporal máxima para um segmento de código. 121
Figura 5.6. Migração de todo o segmento de código para o hardware. 122
Figura 5.7. Migração de parte do segmento de código para o hardware. 122
Figura 5.8. Migração de parte do segmento de código utilizando um nível de granularidade mais baixo. 122
Figura 5.9. Níveis de co-simulação no ambiente de co-síntese COSTLES. 125
Figura 5.10. Sinais de interface do circuito que contém o interface e a UF.. 127
Figura 6.1. Forma de segunda ordem. 131
Figura 6.2. Resposta impulsional do filtro implementado. 133
Figura 6.3. Simulação lógica da UF gerada automaticamente que corresponde a um multiplicador de 16 *bits* em complemento para dois. 134
Figura 6.4. Simulação funcional da solução D. 136
Figura 6.5. Simulação funcional da solução F. 136

Índice de tabelas

Tabela 3.1. Instruções sobre o ficheiro de registos orientadas ao *byte*. 51
Tabela 3.2. Instruções com imediatos e de controlo. 52
Tabela 3.3. Instruções sobre o ficheiro de registos orientadas ao *bit*. 53
Tabela 3.4. Descrição de cada *flag* do registo de estado. 57
Tabela 3.5. Descrição de cada *bit* do registo OPTION. 59
Tabela 3.6. Pré-escalares possíveis para o temporizador do "cão de guarda" e para o contador/relógio em tempo-real. 59
Tabela 3.7. Estados dos *bits* TO e PD após *reset*. 61
Tabela 4.1. Descrição dos sinais de controlo da máquina que controla a procura de uma instrução (FSM1). 75
Tabela 4.2. Descrição dos sinais de controlo. 77
Tabela 4.3. Área, número de células e de FFs da unidade de controlo e dos registos de *pipelining*. 78
Tabela 4.4. Área para diferentes números de registos dos circuitos de descodificação e controlo dos sinais do ficheiro de registos. 80
Tabela 4.5. Área e número de células de cada registo. 81
Tabela 4.6. Área do módulo que contém o PC e a pilha parametrizável. 83
Tabela 4.7. Área e número de células de cada registo TRIS. 87
Tabela 4.8. Valor das quatro linhas de controlo e a respectiva operação da ALU. 88
Tabela 4.9. Área, atraso e número de células da ALU para directivas de optimização diferentes. 88
Tabela 4.10. Área, atraso, e número de células da unidade de manipulação de *bits*. 89
Tabela 4.11. Descodificação de instruções. 91
Tabela 4.12. Área, número de células, e atraso do descodificador. 91
Tabela 4.13. Área e tempo de acesso da memória do programa. 94
Tabela 4.14. Área e número de células da unidade de sincronismo e da unidade OPTION. 98
Tabela 4.15. Alocação das micro-operações críticas pelos ciclos do ciclo de instrução. 99
Tabela 5.1. Área do circuito que descodifica os sinais de interface com as UFs. 110
Tabela 5.2. Área de dados. 111
Tabela 5.3. Área restante do núcleo. 112
Tabela 5.4. Tabela de construções *assembler* que correspondem a ciclos. 117
Tabela 5.5. Tempos de execução do MDC. 119
Tabela 5.6. Tabela de construções *assembler* que correspondem a instruções de controlo condicional. 120
Tabela 5.7. Tabela de construções VHDL que correspondem a algumas das construções *assembler* de controlo condicional. 120
Tabela 5.8. Tempo de execução para a rotina de multiplicação. 126
Tabela 5.9. Área, número de células, e tempo de propagação para o exemplo do multiplicador. 129
Tabela 6.1. Resultados da implementação do filtro digital de quarta ordem. 133
Tabela 6.2. Resultados da UF sintetizada. 134

Tabela 6.3. Resultados da implementação do filtro digital de quarta ordem pelo ambiente de co-síntese COSTLES..... 135
Tabela 6.4. Área total do CI..... 136
Tabela A.1. Opções da aplicação BINOMIO. 151

Acrónimos mais usados

ASIC	***A**pplication **S**pecific **I**ntegrated **C**ircuit* (circuito integrado de aplicação específica).
ASIP	***A**pplication **S**pecific **I**nstruction-Set **P**rocessor* (Processador com conjunto de instruções para aplicações específicas).
BINOMIO	*From Software-**B**ased Spec**I**ficatio**N** to Software/Hardware Transf**O**r**M**ation in the* COSTLES *Env**I**r**O**nment*.
bMs	*bit* mais significativo.
bms	*bit* menos significativo.
CAD	***C**omputer **A**ided **D**esign* (Projecto assistido por computador).
CDFG	***C**ontrol/**D**ata **F**low **G**raph* (Grafos de fluxo de dados e controlo).
CI	**C**ircuito **I**ntegrado (Em Inglês IC: ***I**ntegrated **C**ircuit*).
CISC	***C**omplex **I**nstruction **S**et **C**omputer.*
CMOS	***C**omplementary **M**etal **O**xide **S**ilicon.*
COSTLES	***CO-S**ynthesis **T**ool for **L**ow/medium Complexity **E**mbedded **S**ystems* (Ferramenta de co-síntese de sistemas embebidos de pequeno/médio porte).
CPU	***C**entral **P**rocessing **U**nit* (unidade central de processamento).
E/S	**E**ntrada/**S**aída. No Inglês: I/O (***I**nput/**O**utput*).
EDA	***E**lectronic **D**esign **A**utomation.*
ESDA	***E**lectronic **S**ystems **D**esign **A**utomation.*
FIFO	***F**irst **I**n **F**irst **O**ut*.
FPGA	***F**ield **P**rogrammable Gate **A**rray*.

FSM	***F**inite-**S**tate **M**achine.*
GA	***G**ate-**A**rray* (agregado de células lógicas).
LIFO	***L**ast **I**n **F**irst **O**ut*.
MCU	***M**icro**C**ontroller **U**nit.*
MPU	***M**icro**P**rocessor **U**nit.*
NRE	***N**on Recurring **E**ngineering.*
OHE	***O**ne-**H**ot **E**ncoding.*
OTP	***O**ne-**T**ime-**P**rogrammable.*
PARMIC	***PA**rameterized **R**ISC **MIC**roprocessor*.
PIC	***P**eripheral **I**nterface **C**ontroller*.
RAM	***R**andom-**A**ccess **M**emory.*
RISC	***R**educed **I**nstruction-**S**et **C**omputer.*
ROM	***R**ead-**O**nly **M**emory*.
RTCC	***R**eal **T**ime **C**lock/**C**ounter*.
SOG	***S**ea-**O**f-**G**ates*, agregado de células lógicas do tipo "mar de células".
VHDL	***V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage.*
VLSI	***V**ery **L**arge **S**cale **I**ntegration.*
WDT	***W**atch **D**og **T**imer*.

1. Introdução

"...embedded systems have an impact on almost all of our industry."

Gianluigi Castelli

1.1 Motivação

A crescente utilização de sistemas embebidos - como resultado de os componentes electrónicos para aplicações específicas se tornarem cada vez mais económicos, atractivos, e do ciclo de projecto ter diminuído consideravelmente - tem originado a pesquisa de soluções automatizadas que melhor satisfaçam os requisitos deste género de aplicações. Estes sistemas, por geralmente requererem implementações compactas, extrema fiabilidade e baixo consumo de potência - como são os casos das aplicações em sistemas de comunicação pessoal [1] - necessitam de soluções específicas. Muitas vezes realizam funções críticas (*"life-critical"*), e por isso devem ser robustos e fiáveis durante o ciclo de vida do produto, têm requisitos de tempo-real, restrições de potência, de área, etc. O mercado impõe outras características não menos importantes, como é o caso do reduzido "tempo de introdução no mercado" (*time-to-market*), e do ciclo-de-vida do dispositivo ser cada vez mais pequeno.

Este facto origina que o projectista tenha cada vez menos tempo para desenvolver a ideia: exige-se que perceba a especificação e forneça a implementação num curto período de tempo.

A maioria dos sistemas embebidos necessita de uma solução encapsulada em apenas um integrado (*one-chip-solution*), e por este motivo o mercado de microcontroladores

tem aumentado com taxas de crescimento superiores ao dos PCs. As soluções *firmware* (em que o *software* é implementado em EPROMs, EEROMs, etc., que podem pertencer ao próprio micro) oferecem grandes vantagens, entre elas, a dimensão reduzida, o reduzido consumo de potência, e um ciclo de projecto mais rápido. A maioria dos comerciantes fornecem microcontroladores em versões OTP[1] [2] mais económicas, e que permitem implementações em grande escala. Contudo, as fases de implementação continuam parcialmente manuais e os sistemas têm tendência a serem sobredimensionados com implicações directas no consumo de potência e na solução final. O projectista tem um numeroso conjunto de soluções ao dispor que selecciona empiricamente conforme os requisitos da aplicação.

Em todo o processo, a implementação final depende da experiência acumulada do projectista, dos erros e experiências anteriores e é maioritariamente ad hoc.

Muitas das aplicações de sistemas embebidos necessitam de processadores potentes e de baixo custo. São os casos dos sistemas de controlo embebidos, de sistemas para telecomunicações e multimédia. A escolha para aplicações de processamento de sinal continua a recair em DSPs[2], embora, nos últimos anos tenham aparecido soluções baseadas em processadores específicos (ASIPs[3]) com um conjunto de instruções optimizado para o tipo de aplicação a que se destinam [3]. Outras soluções baseiam-se na implementação em um único circuito integrado de um sistema constituído por um núcleo formado por um processador ou DSP comercial ao qual se juntam *datapaths* configuráveis ou co-processadores, que permitem a implementação de certas funcionalidades [1], ou ainda por uma arquitectura híbrida que combina um processador RISC com uma unidade funcional de DSP [4] . Estas soluções continuam a depender da experiência do projectista, baseiam-se no projecto tradicional ao nível de sistema e apresentam sintomas de sobredimensionameto e desaproveitamento de silício.

---

[1] Do Inglês ***O**ne-**T**ime-**P**rogrammable*. Em Português programável apenas uma vez.

[2] Do Inglês ***D**igital **S**ignal **P**rocessor*. Em Português Processador de sinais digitais.

[3] Do Inglês ***A**pplication **S**pecific **I**nstruction-Set **P**rocessor*. Em Português Processador com conjunto de instruções para aplicações específicas.

A possibilidade de integrar sistemas complexos no mesmo CI[4] e utilizando megacélulas complexas, tais como, memória, MPU[5], MCU[6], multiplicadores e outras funções avançadas, disponibilizadas pelo fabricante, é já possível actualmente. Contudo, com custos NRE[7] a começarem em cerca de 20 milhões de escudos, estes dispositivos integrados estão longe da disponibilidade do consumidor médio e não são apropriados para aplicações de pequeno custo e/ou de pequenas séries.

Os sistemas embebidos são constituídos por componentes *hardware* e por componentes *software* executados em processadores dedicados. Tipicamente, o melhor desempenho pode ser obtido por uma solução de *hardware* dedicado, enquanto que projectos mais rápidos e realizações menos dispendiosas podem ser conseguidos por soluções *software*. Um melhor compromisso entre flexibilidade, desempenho e custo é habitualmente conseguido por soluções intermédias constituídas por sistemas mistos *hardware*/*software* [5], [6], [7], [8], [9], [10]. O desenvolvimento de ambientes ESDA[8], denominados de ferramentas de co-projecto *hardware/software*, que permitem o projecto eficiente destes sistemas é um tema de investigação actual. Estas ferramentas têm como objectivo a exploração automática de arquitecturas (soluções) e a simulação e prototipagem de sistemas heterogéneos para uma dada especificação (independente da tecnologia e que permita um elevado grau de abstracção). Esta exploração produz melhores soluções finais, pois caracteriza-se pela exploração de área/tempos de execução ao nível da arquitectura, com alocação pelos diversos recursos disponíveis.

Outra característica importante é o mapeamento automático da especificação em duas imagens que interagem, a imagem *hardware* e a imagem *software*. Esta geração automática designa-se habitualmente por co-síntese e depende das características da aplicação: restrições temporais, ritmos de transmissão, consumo de potência, área máxima do integrado, etc. Existem já algumas ferramentas académicas de co-síntese,

---

[4] **C**ircuito **I**ntegrado. Tradução do Inglês ***I**ntegrated **C**ircuit* (IC).

[5] Do Inglês ***M**icro**P**rocessor **U**nit.*

[6] Do Inglês ***M**icro**C**ontroller **U**nit.*

[7] Do Inglês ***N**on **R**ecurring **E**ngineering.*

[8] Do Inglês ***E**lectronic **S**ystems **D**esign **A**utomation.*

que partem de diferentes modelos de representação da especificação, como são os casos do COSYMA [11], do VULCAN [12] e do DESIGN ASSISTANT [13] entre outros [14], [15], [16].

1.2 Soluções tradicionais

Muitas aplicações embebidas usam um microprocessador clássico, em que no caso da solução não ser num único circuito integrado são necessárias ligações para a memória, que contém o programa utilizado. Esta solução é mais cara, revela-se normalmente sobredimensionada e sub-utilizada (aproveitamento ineficiente do silício), pois muitas vezes a memória é preenchida com poucas instruções. Por último, têm aparecido soluções para pequeno porte utilizando PLDs[9], soluções num âmbito alargado utilizando FPGAs[10], ou soluções utilizando microcontroladores com EPROM, E2PROM ou OTP. No caso das FPGAs, as soluções revelam-se economicamente impraticáveis para produções em série, recaindo a escolha unicamente num âmbito de emulação, de sistemas protótipos, ou pequenas séries.

O microcontrolador, devido às capacidades de *firmware*, torna-se uma solução prática (até mesmo na substituição de todo um conjunto de lógica discreta). Esta solução pode ser chamada de *one-chip-solution* [17], pelo facto de normalmente incluir: CPU, memória de programa, memória de dados, interface de E/S[11], temporizadores, e controlador de interrupções. Por este motivo facilmente reduz os custos de projecto e as dimensões da solução.

A utilização generalizada de microcontroladores em sistemas de monitorização, instrumentação, electrónica automóvel, robótica, controlo, telecomunicações, e muitas outras aplicações [18], devido ao preço reduzido dos microcontroladores de pequeno/médio porte (entre 300 a 1500 escudos), tem vindo a tornar a solução ASIC[12]

---

[9] Do Inglês ***P**rogrammable **L**ogic **D**evices*.

[10] Do Inglês ***F**ield **P**rogrammable **G**ate **A**rrays*.

[11] Entrada/Saída. Em Inglês I/O (***I**nput/**O**utput*).

[12] Do Inglês ***A**pplication **S**pecific **I**ntegrated **C**ircuits*.

numa segunda escolha. A «solução microcontrolador» é vantajosa pelo tamanho reduzido, baixo consumo, flexibilidade, fiabilidade, menor tempo de desenvolvimento, e preços mais baixos. Em caso de aplicações de grande porte pode-se optar pela distribuição de tarefas por vários microcontroladores, controlados por um microcontrolador central ou em muitos casos por um computador.

O grande mercado automóvel talvez seja o mais importante em aplicações de microcontroladores. Contudo, o próprio consumo de lar se tornou um interessante investimento. Segundo a DataQuest, há em média 35 microcontroladores por cada lar Americano, e as previsões apontam para a existência de 240 no ano 2000. Muitas das aplicações são implementadas com microcontroladores de pequeno porte (4 ou 8 *bits*) sem a necessidade do sobredimensionamento provocado pela utilização de microcontroladores com maior comprimento de palavra. A Figura 1.1 mostra, quantitativamente, a utilização destes dispositivos durante os últimos anos e previsões até ao ano 2000. Como se pode observar os microcontroladores de 8 *bits* detêm a maior fracção do mercado, e prevê-se que a sua utilização continue a crescer.

A previsão para o grande aumento de utilização dos microcontroladores de 16 *bits* explica-se pelo aumento da facilidade de produção de circuitos de elevada complexidade, competitivos (a preços cada vez mais reduzidos), e de maior flexibilidade.

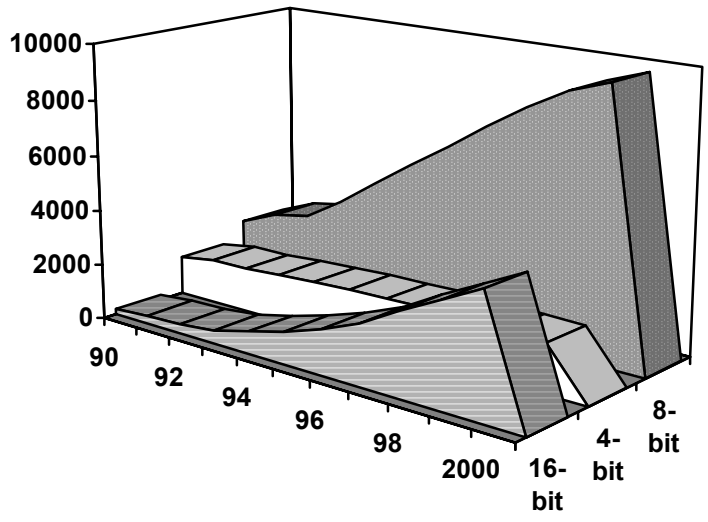


Figura 1.1. Vendas de microcontroladores em milhões de dólares (de 4, 8 e 16 *bits*) anualmente e previsões até ao ano 2000 [Origem: WSTS & ICE - 1994].

1.3 A solução proposta

O trabalho desenvolvido nesta tese apresenta uma solução para sistemas embebidos digitais de pequeno/médio porte num único integrado. A arquitectura alvo encontra-se ilustrada na Figura 1.2, e é constituída por uma unidade central de processamento parametrizável, com conjunto de instruções compatível com um microcontrolador comercial [2], e por unidades funcionais (componentes *hardware*) geradas automaticamente a partir do código fonte, sempre que a solução mais simples (puramente *software*) viole restrições. Restrições temporais e/ou directivas de paralelismo são definidas por directivas adicionadas à especificação da aplicação, e que permitem conduzir o mapeamento. A metodologia descrita foi integrada num ambiente de co-síntese (designado por COSTLES[13]), e embora ainda numa fase embrionária, apresenta resultados não sobredimensionados, permite um ciclo de projecto rápido, e mostra-se adequada para produção de pequenas séries de sistemas embebidos de pequeno/médio porte.

Figura 1.2. Arquitectura do sistema alvo para o ambiente de co-síntese.

A implementação é realizada num ***Gate-Array***[14] (GA) com arquitectura do tipo ***Sea-Of-Gates***[15] (SOGs) em que não existem canais para interligações pré-definidos. Este

[13] ***CO-S****ynthesis* ***T****ool for* ***L****ow/medium Complexity* ***E****mbedded* ***S****ystems*

[14] Em Português designados por agregados de células lógicas.

[15] Em Português habitualmente designada por "mar de células".

tipo de arquitectura usa uma técnica de isolamento por transístor em que o encaminhamento é realizado sobre os transístores não utilizados pelos circuitos. Estes agregados permitem uma maior utilização do silício disponível. A unidade de área para este tipo de arquitectura de GA é definida em *sites*, em que cada *site* corresponde a um par de transístores complementares (um NMOS e um PMOS).

A fábrica para realização de integrados com estrutura SOG existente no INESC [19], [20], permite a produção rápida deste tipo de integrados. A tecnologia utilizada actualmente é CMOS de 1.2μm e o processo consiste no processamento das fases de dupla metalização (dois níveis de metal para interconexão) de bolachas de 6''. No processo adoptado são eliminadas as fases de fabricação de máscaras, ao ser utilizada litografia de escrita directa por laser [21], [22], que permite a redução dos custos e tempo de fabricação.

Partes do trabalho desenvolvido nesta dissertação foram apresentadas em [23] e [24].

1.4 Organização da Tese

No presente capítulo tentou-se motivar e enquadrar o leitor no sistema desenvolvido que será descrito ao longo da presente tese.

No capítulo 2 serão detalhadamente explicados os termos co-projecto e co-síntese, como áreas de investigação emergentes. Serão abordados os problemas inerentes a estes sistemas de projecto, os sistemas de co-síntese mais conhecidos e alguns ambientes de co-projecto.

No capítulo 3 é apresentada a arquitectura do microcontrolador comercial escolhido como base para a unidade de *software* do sistema.

No capítulo 4 são descritos o projecto e a implementação do processador que constitui a unidade central do CI. Neste capítulo é apresentado um circuito integrado fabricado com um programa de teste.

No capítulo 5 descreve-se o ambiente realizado, designado por COSTLES. São explicadas as directivas implementadas e o conversor de *assembler* para VHDL que gera as unidades funcionais.

No capítulo 6 apresenta-se uma aplicação realizada no ambiente de co-síntese desenvolvido, que permite ilustrar a eficiência deste sistema. São apresentados os resultados considerando várias soluções.

Por fim, no capítulo 7 são apresentadas as conclusões e perspectivados os desenvolvimentos futuros.

São apresentados apêndices que se destinam a complementar a tese, fornecendo conteúdos específicos.

1. Co-Projecto e Co-Síntese

"I make a distinction between codesign and cosynthesis. To me codesign is the process of coming up with the specifications - what we might call design technology or design engineering. With cosynthesis, i think of software tools in line with the kind of hardware CAD tools that we've worked with in the past."

Gaetano Borriello

Neste capítulo são descritas algumas ferramentas de co-projecto e co-síntese de sistemas embebidos[1], constituídos por componentes *hardware* e *software*, em que o fluxo de projecto é integrado, conjunto e interactivo. São descritas metodologias que englobam a análise do *software* durante o ciclo de projecto de processadores, e metodologias ao nível de sistema que, partindo da especificação e de uma arquitectura alvo, geram automaticamente os dois componentes respeitando as restrições. São descritas várias ferramentas de co-síntese, entre as quais é dado relevo especial às ferramentas, amplamente divulgadas, COSYMA, VULCAN, e DESIGN ASSISTANT, por serem representativas de diferentes paradigmas de especificação (linguagem de descrição de *software*, linguagem de descrição de *hardware*, e de um modelo de grafos, respectivamente), e das duas possibilidades de orientação da partição (orientada por *software* ou por *hardware*).

[1] Sistemas que usam um microprocessador para desempenhar uma função específica. No entanto este não é usado do modo tradicional, pois normalmente desempenha uma funcionalidade única e é usado para controlar sistemas heterógeneos.

1.1 Introdução

Normalmente, o projecto de sistemas embebidos necessita de um compromisso entre tarefas desempenhadas pelo *software* e tarefas desempenhadas pelo *hardware*. A escolha da solução mais *hard* é estabelecida se o *software* não conseguir respeitar as restrições impostas pela própria aplicação. Estas restrições estão relacionadas com o desempenho, consumo de potência, etc.

A selecção do mapeamento mais adequado é difícil de obter devido ao largo espaço de exploração de soluções (Figura 1.1). Se por um lado a solução *software* está associada a menores gastos de desenvolvimento e menor custo do sistema, por outro lado pode ter desempenho inferior ao pretendido. A solução puramente *hardware* implica maiores gastos de desenvolvimento e maior custo do sistema, contudo é aquela que poderá produzir os melhores desempenhos. É intuitivo que estas duas soluções traçam os limites do espaço de exploração, em que os pontos interiores estão associados a implementações mistas. Este espaço torna-se impossível de pesquisar devidamente sem ferramentas integradas de projecto.

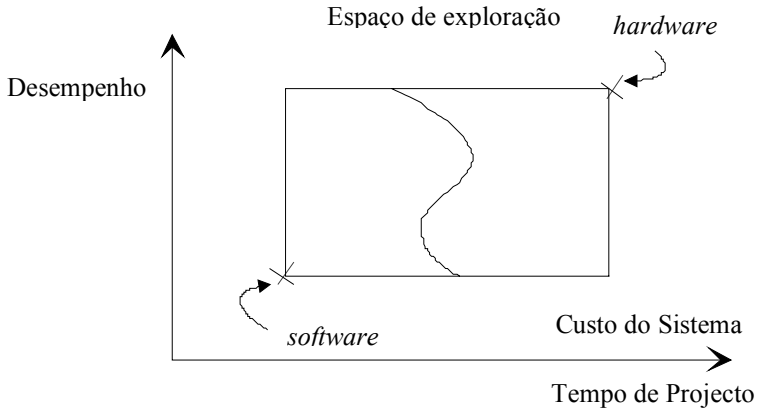


Figura 1.1. Espaço de exploração do co-projecto.

O avanço tecnológico dos últimos anos, nomeadamente o desenvolvimento de ferramentas de projecto automatizado (EDA[2]), de circuitos integrados de aplicação específica (ASICs e FPGAs) - que permitem implementar algoritmos complexos em

[2] Do Inglês ***E**lectronic **D**esign **A**utomation*.

tempo reduzido e preços aceitáveis -, de processadores RISC - possibilitando ao projectista a transferência de funcionalidade de *hardware* para *software* - tornou extremamente complicada a decisão de mapear as funcionalidades. Os avanços no projecto e síntese de circuitos integrados, a disponibilização de processadores avançados e economicamente viáveis tem estimulado o interesse no co-projecto *hardware*/*software* e o aparecimento de ferramentas integradas para o projecto destes sistemas. Contudo, o problema permanece objecto de investigação com grande ênfase nos últimos 3 anos, durante os quais têm sido feitas diversas abordagens aos processos que formam o projecto do binómio *hardware*/*software* com aplicações a sistemas embebidos [5], [6], [7], [8], [9], [10].

É com este objectivo (percorrer eficientemente o conjunto de soluções para estes sistemas, permitindo uma resposta satisfatória ao cada vez menor *time to market*) que têm sido centrados esforços recentes no desenvolvimento de ferramentas computacionais para co-projecto de sistemas mistos *hardware*/*software*[3].

A metodologia de co-projecto distancia-se das metodologias habituais utilizadas por projectistas de *software* e de *hardware*, no sentido em que ao longo de todas as etapas o projecto mantém-se cooperativo [5], e a escolha para uma dada funcionalidade depende dos requisitos do sistema global (desempenho, área de silício, flexibilidade, e custo). Esta metodologia transmite aparentemente mais benefícios ao sistema final do que a utilizada tradicionalmente, na qual o fraccionamento era elaborado na fase inicial do projecto e este ramificado em dois subprojectos distintos (delimitação do projecto de *hardware* do projecto de *software*).

No fluxo de projecto convencional, em que o projecto do *hardware* era distinto do projecto do *software*, destacam-se as seguintes desvantagens:

- Ciclos de projecto independentes, com pouca interacção até à integração de todo o sistema: *hardware* elaborado sem a apreciação completa dos requisitos do

[3] Projecto em que se estabelecem compromissos entre componentes *hardware* e componentes *software*, com vista à integração conjunta, respeitando os objectivos de desempenho e a tecnologia de implementação.

software e restrição da possibilidade de opção de resposta a requisitos de modo comum (*hardware*/*software*) e da modificação do interface *hardware*/*software*;

- O *software* não influencia o projecto do *hardware*, que não é modificado ao longo das opções necessárias no desenvolvimento do *software*;
- O interface do sistema torna muitas vezes necessário modificar *software* e/ou *hardware* (custos mais elevados e perdas de desempenho);
- Selecção prematura do *hardware*, com posteriores correcções feitas em *software*.

O co-projecto é um projecto integrado (ver Figura 1.2) com uma metodologia concorrente e cooperativa entre as análise de *software* e *hardware*, cujas principais vantagens são:

- Implementações mais eficientes, e melhoramento do custo-eficiência;
- Aumento do desempenho do sistema;
- Melhoramento da fiabilidade;
- Resposta mais adaptada aos requisitos.

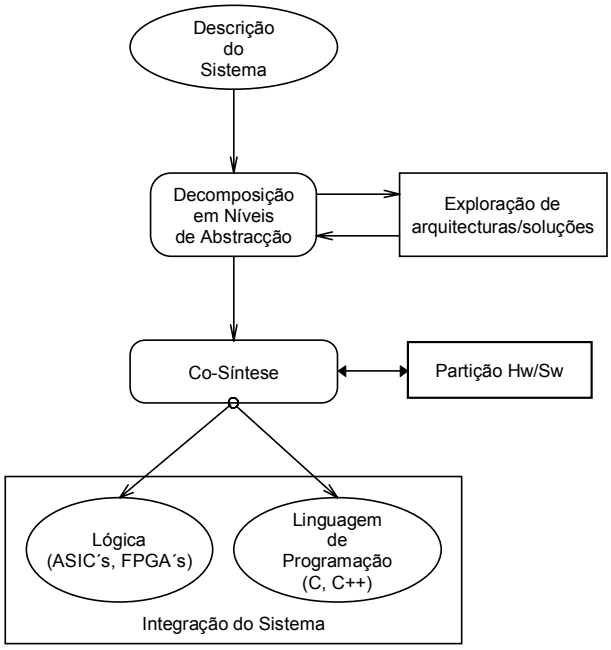


Figura 1.2. Co-projecto *hardware/software*.

Ao co-projecto *hardware*/*software* estão associadas todas as fases do ciclo de desenvolvimento do sistema: desde a especificação até à implementação física. As fases iniciais do co-projecto permitem a exploração de arquitecturas com integração conjunta. Como parte integrante do co-projecto surge a co-síntese. Esta representa a fase do co-projecto em que a arquitectura do sistema está definida. Parte da especificação do sistema e baseia-se em funções de custo que conduzem a distribuição de tarefas pelos componentes *hardware* e *software*. A co-síntese gera os dois componentes, e permite a exploração das duas imagens (a imagem *hardware* e a imagem *software*) pela arquitectura pré-definida. Engloba tarefas como a partição e a síntese do *software* e do *hardware*.

Os processadores apresentam-se como o "coração" dos sistemas embebidos. O projecto destes deve tomar em conta a especificidade do código que executarão. Mesmo os projectistas de processadores de propósito geral analisam os resultados da simulação de *benchmarks* de forma a poderem medir os benefícios de algumas instruções [25]. A exploração da arquitectura e das estruturas mais efectivas de *pipeline* é um problema de co-projecto.

1.2 Especificação

A especificação do sistema (primeira etapa e a que tem mais implicações no sistema final pois é a imagem da resposta aos requisitos) é conduzida por um modelo ou linguagem formal, que descreva o comportamento do sistema, abstraindo-se da implementação física. Estas especificações, mesmo quando sob a forma de escrita, não podendo ser simuladas permanecem ambíguas e incompletas. Pelo motivo exposto, a especificação deve capturar a funcionalidade do sistema e permitir a simulação e verificação da própria captura. O comportamento do sistema deve ser verificado extensivamente, ao longo de todos os níveis de abstracção do co-projecto, de modo a reduzir o risco de se obter um sistema funcionalmente incompatível.

O modelo de captura da especificação depende da aplicação. Vincentelli et al [9] apresentam resumidamente alguns modelos de captura que modelam a maioria das

aplicações dos sistemas embebidos. Em [5] são também descritos alguns modelos e linguagens de especificação das quais o SpecCharts é um exemplo.

Não existe ainda consenso nos modelos a utilizar para a captura da especificação. Para sistemas de processamento de sinal existe um grande número de estilos de projecto utilizados no desenvolvimento do *software*. Uma opção é o uso de linguagens tradicionais de programação (ex. C). Contudo a maioria dos programadores de DSPs e microcontroladores continuam a utilizar preferencialmente o *assembler* devido ao facto do código gerado pelos compiladores, muitas vezes, não atingir a eficiência em termos de desempenho requerida pela aplicação a que se destina, e devido aos próprios sistemas necessitarem de funcionalidades muito próximas do *hardware*. Isto deve-se fundamentalmente à não especificidade destes compiladores, que por enquanto não consideram completamente as arquitecturas irregulares destes integrados. As linguagens de especificação de *software* para sistemas de tempo-real (como Esterel, Statecharts ou algum modelo de C) têm servido directa ou indirectamente como linguagens de descrição de *hardware*. Woo & Wolf [26] apresentam um modelo de especificação para sistemas mistos baseado em C++.

Para os sistemas *hardware*/*software* colocam-se várias questões de difícil resolução. Uma refere-se ao modelo de descrição utilizado na especificação do sistema. Têm sido apresentadas várias soluções, ferramentas que permitem a heterogeneidade da especificação, permitindo vários modelos formais baseados em grafos, com o projectista a escolher o tipo de modelo para cada aplicação. Outras soluções centram-se no uso de linguagens textuais de descrição de *hardware* (VHDL, Verilog, Hardware C, etc.) ou de descrição de *software* (C, C++, etc.) com extensões a construções necessárias. E, por fim, surgem também soluções com a especificação num modelo de grafos estendido, mas único. Se, por um lado as primeiras permitem a integração pela ferramenta de vários sistemas heterogéneos e com possibilidade de implementação de aplicações díspares, por outro lado não retiram partido da especificidade das várias aplicações dos sistemas embebidos. Face aos problemas analisados, supõe-se que o problema da especificação não venha a ser consensual, embora possa convergir lentamente para modelos orientados por objectos.

1.3 Partição *hardware*/*software*

As técnicas de partição[4] têm um papel fundamental no projecto de circuitos integrados digitais. Por exemplo, ao nível de implantação física (*layout*) são utilizados algoritmos de partição para encontrar os agrupamentos de células que permitam minimizar a área e/ou o tempo de propagação. Estes agrupamentos são normalmente escolhidos pela forte conectividade entre células. Quando não é possível integrar a funcionalidade pretendida num único dispositivo, esta é fraccionada por CIs (partição ao nível de sistema).

Considerando uma representação de grafos [5] que modele o sistema em termos do seu comportamento (a representação CDFG[5] é um óptimo exemplo), a partição pode ser a divisão desse grafo em sub-grafos de forma a permitir atingir os objectivos considerados na fase de especificação, tendo em conta concorrência, comunicação e estruturas de controlo. Na partição o sistema é decomposto em agrupamentos denominados de *clusters*. Em [27] são apresentados algoritmos de partição automática divididos em métodos construtivos, que começam por um ou mais nós iniciais do CDFG e vão adicionando nós, um de cada vez, e métodos de melhoria iterativa, que começam com uma partição inicial, que vai sendo sucessivamente melhorada movendo objectos entre as partições.

A partição de funcionalidades com mapeamento em *hardware* ou *software* costuma designar-se por "decomposição *hardware*/*software*", ou simplesmente por decomposição. A decomposição é uma forma de partição com alocação dupla. A decisão de decomposição é alocada em componentes *hardware* ou componentes *software*. O problema revela-se computacionalmente árduo: tendo em conta $\mathbf{m_{hw}}$

---

[4] Tarefa de agrupar objectos de modo a que uma dada função objectivo seja optimizada de acordo com um conjunto de restrições de projecto.

[5] Do Inglês ***C**ontrol*/***D**ata **F**low **G**raph*. Uma forma especializada desta representação, denominada de CDFG disjunto, permite representar o comportamento do sistema, apresentando dois modelos de grafos separados. Um, responsável pela modelação do fluxo de dados, o outro, responsável pela modelação do fluxo de controlo. Esta representação é referida pois é propícia para a modelação de sistemas baseados em duas unidades, *Control* + *Datapath*. As construções de controlo são mapeadas em nós de fluxo de controlo e atribuições dentro dos blocos básicos são mapeadas em fluxos de dados. Esta representação captura sequência, saltos condicionais, construções de ciclo e actividades operacionais descritas pelas instruções de atribuição em VHDL.

possibilidades para implementação de uma dada função em *hardware* e $\mathbf{m_{sw}}$ possibilidades no caso da implementação em *software*, os agrupamentos de decomposição possíveis são $(\mathbf{m_{hw}} + \mathbf{m_{sw}})^n$, sendo n o número de funções. Mesmo abreviando o problema, considerando apenas um tipo de solução para o *hardware* e outro tipo de solução para o *software* (soluções únicas), se por exemplo tivermos um sistema com comportamento constituído por 20 funções obtemos 2^20 agrupamentos de decomposição possíveis (≅1 milhão!). As possibilidades aumentam à medida que se consideram níveis de granularidade mais baixos.

A metodologia de partição tradicionalmente utilizada era manual e assentava na intuição e experiência acumulada, que se traduzia em considerações deficientes de todas as topologias, pelo tempo escasso e dificuldade da mente humana em lidar com sistemas de grande complexidade. Por vezes a partição adequada acabava por não ser considerada. Como resultado destas limitações, o projectista usava mais *hardware* do que o necessário para conseguir atingir os requisitos satisfatoriamente.

O desenvolvimento de ferramentas que permitem, com base na especificação formal do sistema, particionar funcionalidades automaticamente tendo em conta métricas de custo, viabilizou a co-síntese. Normalmente, parte-se do sistema com a funcionalidade atribuída ao *hardware* ou ao *software* e fazem-se migrar tarefas para o *software* ou para o *hardware* respectivamente. Sempre que a funcionalidade inicial é atribuída ao *hardware*, diz-se que a decomposição é orientada pelo *hardware*. No caso contrário, diz-se que a decomposição é orientada pelo *software*.

O objectivo do projectista é implementar o sistema com o mínimo de *hardware* possível, conseguindo satisfazer os desempenhos e restrições pretendidas. O projectista implementa "o mais que pode" em *software*. Normalmente a partição orientada pela especificação do sistema em *software* produz melhores resultados globais do que a metodologia inversa [28].

De forma a que as partições sejam comparadas é necessário definir uma função de custo (função objectivo), cujo domínio é constituído por métricas (área, tempo de execução, etc.) e pelas restrições para cada uma delas (restrição de área, restrição de

tempo de execução, etc.), que condicionam as decisões de projecto. A função é normalmente expressa por:

$$F_{obj.} = k_1 \times F(\text{área}, \text{área\_máxima}) + k_2 \times F(\text{atraso}, \text{atraso\_máximo}) + \ldots \quad (2.1)$$

em que os k_j representam os pesos a atribuir a cada métrica. A função $F_{obj.}$ indica a proximidade da métrica à restrição. Normalmente retorna zero quando não há violação e a distância quando há violação.

A representação da funcionalidade do sistema pode ser especificada por diversos níveis de abstracção que têm consequências directas na granularidade permitida e por conseguinte na obtenção da decomposição óptima. A granularidade representa uma medida do tamanho da especificação em cada objecto ou função. Esta pode ser fina (*fine-granularity*) ou espessa (*coarse-granularity*). Na primeira, cada objecto contém apenas uma pequena parte da especificação e, por isso, são considerados muitos objectos. Na segunda cada objecto contém uma grande parte da especificação e, por isso, são considerados poucos objectos. O nível de abstracção utilizado na especificação condiciona a granularidade da decomposição.

1.4 Projecto de processadores com base no código

Trabalhos recentes, tomando como exemplo [3] e [29], têm revelado a importância da inserção do compilador durante o ciclo de projecto de um processador. A maioria das metodologias conhecidas baseiam-se na análise do código formado por um conjunto de instruções base. Esta análise permitirá seleccionar o conjunto de instruções, algumas das quais são sequências das anteriores, de modo a aumentar o desempenho do processador sob restrições de área e consumo de potência.

Têm sido abordados métodos que permitem a implementação de ASIPs com um conjunto de instruções optimizado para as aplicações a que se destinam. Estes dispositivos são constituídos por um processador de base e por extensões em *hardware* de determinadas operações que se julgam críticas. Os processadores deste

tipo retêm a flexibilidade, pois mantêm um nível de programação, e adicionam funcionalidades específicas[6].

Em [3] são apresentadas análises, elaboradas por um sistema de compilação desenvolvido, de várias *benchmarks* em que são detectadas sequências de instruções que se repetem frequentemente. Estas análises demonstram a eficácia da metodologia, que permite a selecção eficaz de operações a adicionar ao conjunto de intruções base.

Em [29] J. Hennessy apresenta uma metodologia de implementação de um processador baseada na exploração do conjunto de instruções. É referida a simulação hierárquica, capaz de aumentar a possibilidade de projectar um sistema *hardware*/*software*, que seja funcionalmente correcto e vá ao encontro do desempenho pretendido enquanto mantém um tempo curto de projecto. O artigo realça que só com o uso de simulação hierárquica, para todo o sistema, é que se torna possível desenvolver o *hardware* e o *software* simultaneamente. Antes do projecto de processadores deve ser elaborado um compilador genérico, que permita a exploração do conjunto de instruções e a análise quantitativa de alternativas.

J. Hennessy identifica 3 chaves de projecto de processadores (com espaço de grandes dimensões ao nível da exploração de soluções): conjunto de instruções, organização em *pipelining*, e esquemas de memória. O conjunto de instruções define a fronteira entre o *hardware* e o *software*. Por este motivo representa a definição ao nível mais alto do *hardware* e a base para verificação do projecto. É nesta fronteira que é feito o acordo entre o *hardware* e o *software*.

No artigo é apresentado um sistema de simulação hierárquica (PIXIE) que foi utilizado no projecto do MIPS R4000[7]. Este sistema permite a verificação da funcionalidade e do desempenho. O projecto é guiado por restrições de área de silício, potência consumida e número de pinos de encapsulamento.

---

[6] Aplicações de processamento de sinal em que são utilizados DSPs têm uma grande frequência de operações MAC (*multiply and accumulate*) justificando, por isso, a existência de uma instrução com esta cadeia de operações por parte da maioria dos DSPs.

[7] Marca registada da MIPS Computer Systems.

São considerados 5 níveis de simulação: ao nível de instrução, ao nível de sistema, ao nível RTL[8], ao nível de *switch* com atrasos, e ao nível de circuito extraído. Os dois primeiros são os mais críticos no co-projecto de sistemas *hardware*/*software*. São também os mais rápidos, permitindo, por isso, extensão da verificação funcional e do desempenho.

1.5 Ambiente de co-projecto

O co-projecto de sistemas embebidos assistido por computador (integrado em ferramentas de projecto) tem como objectivo assegurar a qualidade da implementação com um ciclo de projecto reduzido. Como ferramenta exemplo é descrito o ambiente de projecto denominado de Ptolemy [30]. Esta ferramenta possibilita o co-projecto e simulação conjunta de sistemas heterogéneos, permitindo a validação e exploração de soluções. Ao permitir a descrição de aplicações complexas de um modo natural e fácil, pelo uso de tarefas a um alto nível de abstracção possibilita a descrição sem que o projectista assuma uma arquitectura particular do sistema a implementar.

1.5.1 PTOLEMY

A ferramenta de co-projecto Ptolemy [30] é um ambiente de simulação e prototipagem de sistemas heterogéneos desenvolvida na Universidade da Califórnia em Berkeley. A ferramenta é usada com sucesso, por cursos em Berkeley, desde 90 e foi distribuída ao exterior a partir de 91. Actualmente é muito utilizada por grande parte da comunidade académica e também por organizações industriais. Existem diversas aplicações bem sucedidas projectadas com o uso da ferramenta.

A ferramenta torna possível a integração quase total do projecto [31] de sistemas para processamento de sinal (incluindo aplicações de tempo-real), comunicações, sistemas embebidos com microcontroladores, sistemas paralelos com processadores de sinais

[8] Do Inglês ***R**egister **T**ransfer **L**evel*. Em Português habitualmente nível de transferência de registos.

digitais (DSPs), etc. A ferramenta aceita a especificação dos subsistemas em diferentes paradigmas, elaborando a integração de modo a permitir a simulação conjunta.

Dada a descrição funcional (o sistema aceita vários modelos de representação [32], dos quais o mais usado é o modelo SDF[9]) de um determinado algoritmo, o sistema gera o código para a arquitectura especificada (o Ptolemy pode gerar código em C, *assembler* para o Motorola 56000 e 96000, para o Sproc[10], etc.).

Com o Ptolemy todo o projecto, desde o desenvolvimento do algoritmo até ao projecto do circuito, é integrado num único ambiente. Como o sistema aceita vários paradigmas de especificação, a exploração de soluções *hardware*/*software* pode ser executada, comparando os resultados da implementação de várias funções em *hardware* ou em *software*. Os paradigmas de especificação são usados sem restrições impostas. A ferramenta pode acomodar, graças ao núcleo não dogmático, modelos funcionais, de fluxo de dados, máquinas de estados finitos, *statecharts*, processos sequenciais de comunicação, redes de Petri, etc. A combinação de descrições torna viável o projecto heterogéneo.

Cada subsistema é modelado internamente numa linguagem de programação de *software* orientada por objectos (C++) capaz de modelar de uma maneira eficiente e natural, e de integrar estes subsistemas num sistema global. Cada subsistema pode ser modelado com níveis de detalhe diferentes.

O Ptolemy usa um interface gráfico amigável, baseado em diagramas de blocos hierárquicos.

Encontra-se em desenvolvimento a extensão da especificação a outros domínios e a integração de ferramentas de síntese alto-nível, de circuitos digitais, existentes em Berkeley.

---

[9] Do Inglês ***S**yncronous **D**ata **F**low graph*. O SDF é um caso especial dos grafos de fluxo de dados, em que o número de amostras consumidas e produzidas é conhecido estatisticamente na fase de compilação.

[10] DSP de multiprocessamento da Star Semiconductor.

1.6 Sistemas de co-síntese

Os sistemas de co-síntese têm como base a especificação formal do sistema (em linguagens textuais ou modelos baseados nos diagramas de fluxo de dados e de controlo) e procedem ao mapeamento da funcionalidade descrita em duas imagens distintas, a imagem *hardware* e a imagem *software*. Estas duas imagens são criadas por processos automatizados, conduzidos pelos recursos disponíveis, pelas restrições temporais de algumas tarefas, pelas restrições ao nível da comunicação de dados, ao nível dos custos de implementações em *hardware*, etc.

A decomposição em partições é mapeada nos recursos disponíveis, reduzindo a complexidade do problema, ao não considerar todas as alocações possíveis, pois normalmente existe um número elevado de partições para uma dada alocação. Ao nível de sistema os recursos de *hardware* podem ser: FPGAs, ASICs, Memórias, Unidades de Interface, etc. Os recursos de *software* podem ser: processadores, microcontroladores, etc.

1.6.1 Arquitectura geral da maioria dos sistemas

A arquitectura alvo dos sistemas de co-síntese é, na maioria dos casos, baseada na arquitectura representada na Figura 1.3. Esta incorpora um processador de uso geral, responsável pela parte da especificação com imagem *software*.

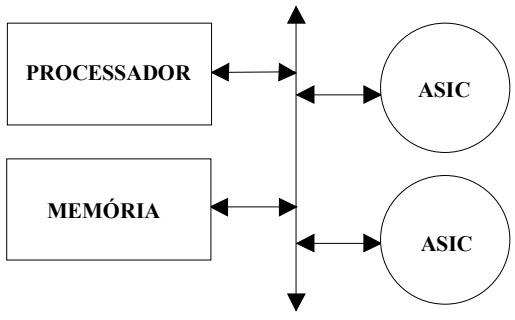


Figura 1.3. Diagrama de blocos da arquitectura alvo da maioria dos sistemas de co-síntese.

A principal motivação para o uso de processadores já existentes, deve-se ao facto da redução de tempo de projecto e custo, implementando a funcionalidade pretendida com um programa no processador. Normalmente para sistemas embebidos,

principalmente sistemas de tempo real, estas soluções falham ao não conseguirem atingir os níveis de desempenho requeridos.

A atracção pela implementação da maior parte possível da funcionalidade em *software* tem explicação pelas vantagens seguintes:

- Existência de compiladores optimizados, com melhores resultados do que a programação directamente em *assembler*.
- Fiabilidade dos processadores, por terem passado por processos de teste e controlo de qualidade que asseguram a funcionalidade correcta, ao invés de integrados de aplicação específica.
- A depuração do *software* é extremamente simples, ao contrário da tarefa de depuração do *hardware*.
- O custo do sistema é menor.
- Facilidades de verificação e simulação.
- Maior flexibilidade no caso de modificação.

As técnicas de co-síntese são também aplicadas em implementações de emulação, permitindo a exploração de soluções, verificação e simulação do sistema com dispositivos reprogramáveis e agregados de células reconfiguráveis. Outras aplicações mais genéricas permitem acelerar aplicações computacionalmente árduas com o uso de estruturas baseadas nos emuladores referidos anteriormente. Em aplicações como a simulação de circuitos digitais, algoritmos de processamento de imagem e algoritmos de computação gráfica 3d entre outros, podem não existir restrições temporais, mas o utilizador necessita dos resultados no menor espaço de tempo possível. É por este motivo que a co-síntese também desempenha em aplicações de carácter não específico uma importância primordial.

É possível dividir as aplicações da co-síntese em três grupos:

- Aplicações específicas com restrições (sistemas embebidos). Neste grupo estão incluídas a maioria das aplicações em tempo real, que geralmente necessitam de

fornecer respostas a estímulos em determinado espaço de tempo, muitas vezes impossíveis de concretizar sem sobredimensionamento da solução.

- Emuladores para a eficiente exploração das soluções. Neste grupo estão incluídos os sistemas de rápida prototipagem para a eficiente simulação da funcionalidade de um sistema em que se substitui a implementação final por um emulador, com características próximas da versão final.

- Aplicações genéricas, em que se pretende aumentar a velocidade dos processos. Este grupo inclui todas as aplicações que consomem demasiado tempo de processamento e para as quais a aceleração da execução dos processos críticos resulta num aumento de velocidade de cada tarefa. As aplicações podem ser de simulação de circuitos, de processamento intensivo de sinal, de imagem, etc. De alguma forma, este grupo tem a mesma finalidade que o processamento paralelo.

Nos últimos anos têm sido desenvolvidas diversas ferramentas de co-síntese, de entre as quais serão abordadas de seguida, e resumidamente, a ferramenta PRISM [14] e uma ferramenta para simulação de sistemas digitais [15]. Dar-se-á relevo, por serem as ferramentas mais divulgadas com aplicações industriais e académicas, às ferramentas COSYMA, VULCAN e DESIGN ASSISTANT apresentando-se detalhadamente a descrição de cada uma.

1.6.2 PRISM

Em [14] é adoptada uma solução constituída por um processador de propósito geral e FPGAs (ver Figura 1.4). A especificação do sistema é feita em C. Os segmentos de código computacionalmente intensivos migram para operações fundamentais (operações adicionais implementadas em FPGAs).

A arquitectura retém as propriedades inatas e adquire melhoramentos de desempenho específicos a arquitecturas para aplicações específicas. A ferramenta apresentada, desenvolvida na Universidade de Brown, é designada por PRISM (***P****rocessor* ***R****econfiguration* ***I****nstruction* ***S****et* ***M****etamorphosis*) e o protótipo inicial por PRISM-I.

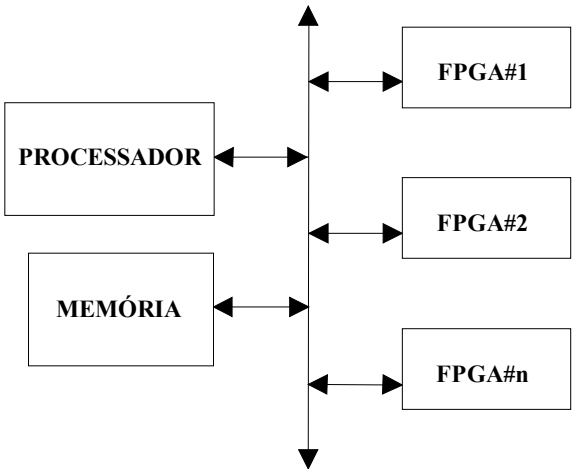


Figura 1.4. Arquitectura alvo do sistema de co-síntese PRISM.

A viabilidade do sistema está relacionada com a capacidade de a síntese de algumas operações estar totalmente integrada no processo de compilação (transparente ao programador). O compilador aceita um programa como entrada e produz uma imagem mista (*hardware*/*software*) como saída. A imagem *hardware* consiste na especificação física usada para programar o agregado reconfigurável, a imagem *software* consiste em código de máquina vulgar com a adição de código que permite a integração de novos elementos. A execução óptima requer um balanceamento entre *hardware*/*software*.

O sistema de co-síntese impõe como resolução mínima (nível de granularidade) funções (sub-rotinas). O primeiro passo é a identificação/extracção de funções que permite separar candidatos a componentes *hardware* e a componentes *software*. O programador escolhe as funções de entre as indicadas como candidatas pela ferramenta.

O sistema foi implementado em duas placas: uma com um Motorola 68010 a 10Mhz *(Armstrong processing node*) e outra que consiste em 4 FPGAs Xilinx 3090 [33]. Um barramento de 16 *bits* liga as duas placas. Sem estados de espera a comunicação revela-se ineficiente, pois demora 48 a 72 ciclos de relógio, contudo o sistema protótipo atinge melhoramentos de desempenho na gama de 3 a 54 para as funções apresentadas pelos autores [14]. Os tempos de compilação para os exemplos foram de 3 a 24 minutos.

A síntese de *hardware* não permite circuitos sequenciais e a ferramenta não executa a partição automaticamente, sendo a migração escolhida pelo utilizador. Outra deficiência, foco admitido pelos autores de futuro desenvolvimento, é a utilização de apenas um subconjunto do C, na especificação inicial, pelo sistema protótipo.

1.6.3 Sistema de co-síntese para simulação de circuitos digitais desenvolvido na Universidade de Stanford

Em [15] é apresentada uma ferramenta de co-síntese desenvolvida na Universidade de Stanford e cujas soluções finais se destinam a acelerar a simulação de sistemas digitais. A partir da descrição do sistema numa HDL, a ferramenta cria um simulador de alto desempenho constituído por componentes *hardware* e por componentes *software* (ver Figura 1.5). A arquitectura alvo é genericamente semelhante à arquitectura da Figura 1.4 com a adição de memória *cache*. A especificação do sistema digital elaborada em Verilog é traduzida para C e os blocos computacionalmente mais demorados, escolhidos iterativamente de modo a maximizarem o desempenho global, são implementados nas FPGAs pela síntese de Verilog utilizando as ferramentas da Synopsys™[11].

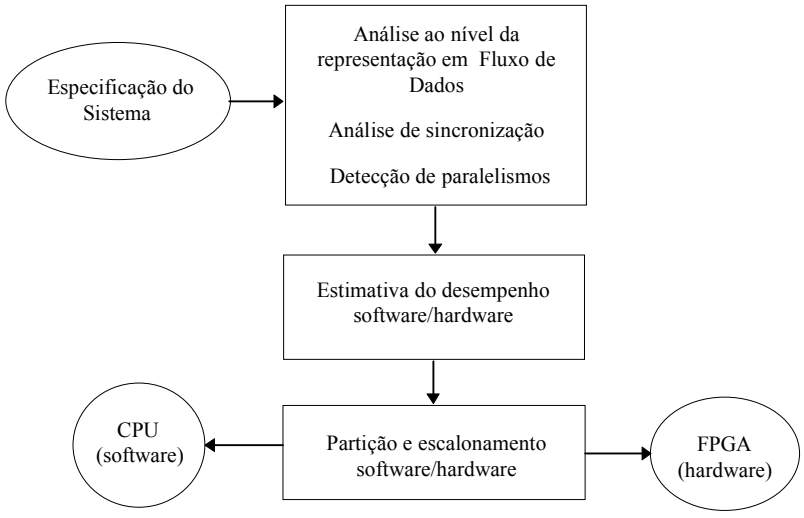


Figura 1.5. Fluxo de co-síntese para simulação de sistemas digitais.

[11] Marca registada da Synopsys Inc..

Os resultados apresentados pelos autores mostram as melhorias no desempenho da simulação de um exemplo com a ferramenta proposta face à simulação do mesmo sistema digital com o simulador Verilog-XL1.6 (são referenciados aumentos da velocidade de simulação de 150 a 300).

O fluxo de co-síntese é totalmente automatizado. Os autores revelam o objectivo de melhorarem as comunicações entre dispositivos da arquitectura (por ex. permitindo a comunicação entre implementações em *hardware* sem intervenção do CPU). A principal deficiência da ferramenta centra-se no facto de não incorporar a optimização do compilador na implementação mista.

1.6.4 Ferramenta para sistemas embebidos desenvolvida na Universidade de Berkeley

As aplicações da ferramenta [16] são pequenos sistemas embebidos, sem grandes exigências computacionais, que usam um ou mais microcontroladores e dispositivos de aplicação específica.

A descrição do sistema é elaborada num modelo designado por CFSM[12] que é uma extensão do modelo clássico de máquinas de estados finitos (FSMs[13]). Esta representação permite a captura de funcionalidades de *hardware* ou *software* para sistemas de controlo dominante. Contudo, como este modelo não é propício para ser utilizado directamente pelo projectista (por ser demasiado baixo-nível), a ferramenta compila a especificação em Esterel para CFSM (tradutores de StateCharts ou de um subconjunto do VHDL estão também em desenvolvimento).

A ferramenta usa dois modelos obtidos do modelo CSFM. O modelo de *netlist* normalizado que é usado pelas ferramentas de síntese lógica e o modelo *s-graph* (modelo de grafos para *software*). O modelo *s-graph* é muito mais simples que a linguagem *assembler* e por isso facilita a optimização para que os compiladores e

---

[12] Do Inglês ***C**odesign **F**inite **S**tate **M**achine*.

[13] Do Inglês ***F**inite-**S**tate **M**achines.*

escalonadores reais tenham a tarefa facilitada. A ferramenta pode traduzir o *s-graph* para uma linguagem de alto nível ou para *assembler* específico do microcontrolador. A ferramenta permite a geração de vários tipos de *s-graphs* com implicações no tamanho do *software* e no tempo de execução deste (permitindo a exploração de diferentes soluções).

Os algoritmos de síntese propostos são baseados em restrições comuns à maioria dos sistemas embebidos de aplicações industriais. Cada partição *hardware* é totalmente síncrona e cada partição *software* é implementada como um programa independente em execução no microcontrolador. Ambas as partições usam o mesmo relógio.

A ferramenta permite a verificação formal da especificação (dispõe de um paradigma de validação que permite verificar se a síntese satisfaz a especificação) e a rápida simulação de casos julgados relevantes para a correcta funcionalidade.

1.6.5 Design Assistant

A ferramenta Design Assistant é um sistema de co-síntese para sistemas embebidos com componentes de processamento de sinal em tempo real em que a especificação é elaborada ao nível de sistema. A ferramenta foi concebida por Kalavade [13] na Universidade da Califórnia em Berkeley e engloba a partição *hardware*/*software*, a co-síntese, a co-simulação e a gestão da metodologia de projecto.

Os principais destinatários do Design Assistant são sistemas para processamento de sinais periódicos em tempo-real. Estas aplicações têm, normalmente, restrições fixas de *throughput* e, muitas delas, podem ser especificadas naturalmente pelo modelo computacional de grafos de fluxo de dados (SDF). A aplicação é representada num modelo de grafos, cujos nós representam tarefas computacionais, e os arcos indicam o fluxo de dados. Um nó dispara depois de receber dados em todas as entradas, e durante o disparo gera dados em todas as saídas.

O fluxo de co-síntese do Design Assistant é ilustrado na Figura 1.6. Após a especificação do sistema em SDF, a ferramenta traduz esta representação para o

modelo de grafos DAG[14]. Os algoritmos de partição são executados sobre este modelo e produzem o modelo com partições em que os nós (que representam tarefas) são mapeados em *hardware* ou em *software*.

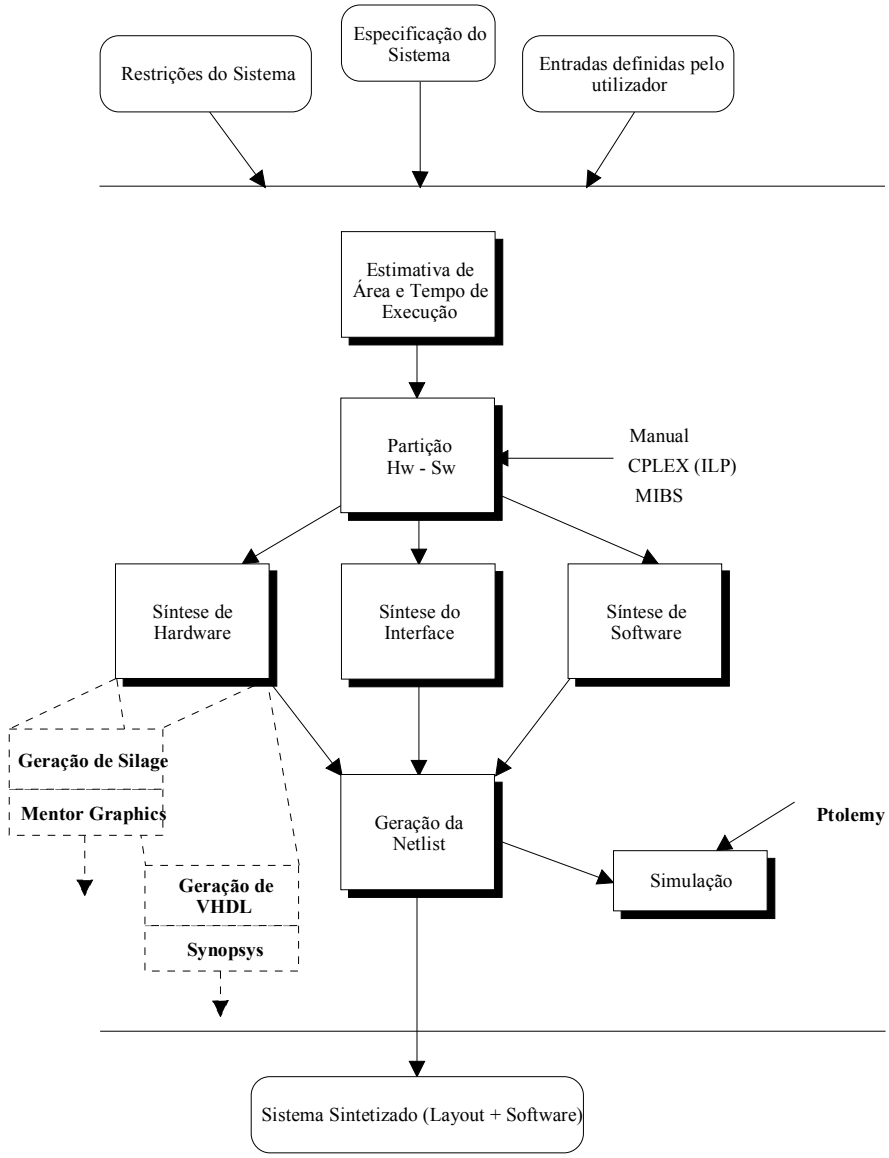


Figura 1.6. Fluxo de co-síntese do Design Assistant [13].

O autor apresenta uma heurística [13], designada por MIBS[15], para resolver o problema da partição. A heurística apresenta resultados muito próximos dos resultados

[14] Do Inglês ***D**irect **A**cyclic **G**raph*.

[15] Do Inglês ***M**apping and **I**mplementation **B**in **S**election.*

óptimos fornecidos pelo algoritmo exacto ILP[16] (CPLEX[17]) com um reduzido tempo de CPU. O algoritmo em cada passo de execução e para cada nó do grafo pode ser orientado por uma medida que define se o aspecto mais crítico para o nó em questão são os recursos necessários ou o tempo de execução. Exemplos aleatórios utilizados revelam que a solução obtida pelo MIBS está dentro de 18% do resultado óptimo fornecido pelo ILP e o tempo de CPU é na ordem de 70 vezes mais rápido. A ferramenta também permite a partição orientada pelo utilizador (manual).

A ferramenta é integrada no ambiente de co-projecto Ptolemy constituído por ferramentas de simulação de sistemas heterogéneos, geradores de *software*, e de *hardware* (Hyper[18]). O Ptolemy foi usado para sintetizar o código para o DSP 56000 da Motorola e código Silage para cada nó do DAG (ver Figura 1.7).

As estimativas da área de *software* e tempo de execução para cada nó são obtidas usando analisadores do código gerado para o DSP. O código Silage para cada nó é entrada do Hyper, que gera estimativas do tempo de execução e da área do *hardware*. O tempo de execução estimado corresponde ao tempo de propagação optimizado do caminho crítico.

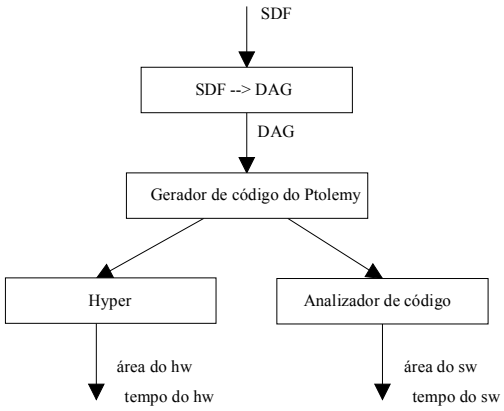


Figura 1.7. Fluxo para a determinação das estimativas.

[16] Do Inglês ***I**nteger **L**inear **P**rogramming*.

[17] *Software* de optimização disponível comercialmente.

[18] Informações podem ser obtidas no endereço: http://infopad.EECS.Berkeley.EDU/~hyper/

A síntese de *hardware* depende do algoritmo de descrição utilizado para uma tarefa. Ao nível da síntese de *software* existem também compromissos entre o tempo de execução e a área de programa (código *inline* é mais rápido que o uso de subrotinas, mas ocupa mais área de programa).

O mapeamento de um nó é feito por um procedimento global de optimização após a especificação. As descrições sintetizáveis são obtidas de um modelo de grafos particionado.

A arquitectura alvo é constituída por um processador de propósito geral, memória, e módulos de *hardware* (ver Figura 1.8). O controlo do interface entre os módulos é implementado em *hardware* específico que depende do mecanismo de interface escolhido.

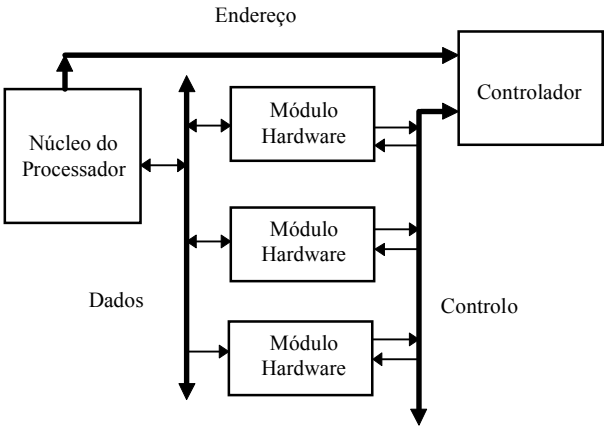


Figura 1.8. Arquitectura alvo.

O interface *hardware*/*software* não é obtido automaticamente embora o autor refira que o problema é de fácil resolução. O sistema impõe a existência de bibliotecas de elementos do DSP e do Silage para cada nó do SDF.

1.6.6 COSYMA

O sistema COSYMA[19] [11] é uma ferramenta de co-síntese de pequenos sistemas

[19] Do Inglês ***COSY****nthesis for e****M****bedded* ***A****rchitectures*. Informações podem ser obtidas através do endereço de WWW: http://sueton.ida.ing.tu-bs.de/cosyma.

embebidos de tempo-real que contém alguns compiladores, sistemas de síntese alto nível, simuladores, várias ferramentas de análise e estimativas de sistemas, e funções de optimização. O desenvolvimento do sistema tem integrado programas comerciais e universitários. Os autores referem que o sistema já foi aplicado no contexto de um projecto industrial.

Este sistema, apresentado na Figura 1.9, é uma ferramenta de co-síntese desenvolvida na Universidade de Braunschardwareeig (Alemanha). Esta ferramenta, particiona a especificação de uma determinada aplicação em blocos e migra os blocos computacionalmente mais intensivos para um co-processador dedicado, de forma a permitir a obtenção de melhores desempenhos finais.

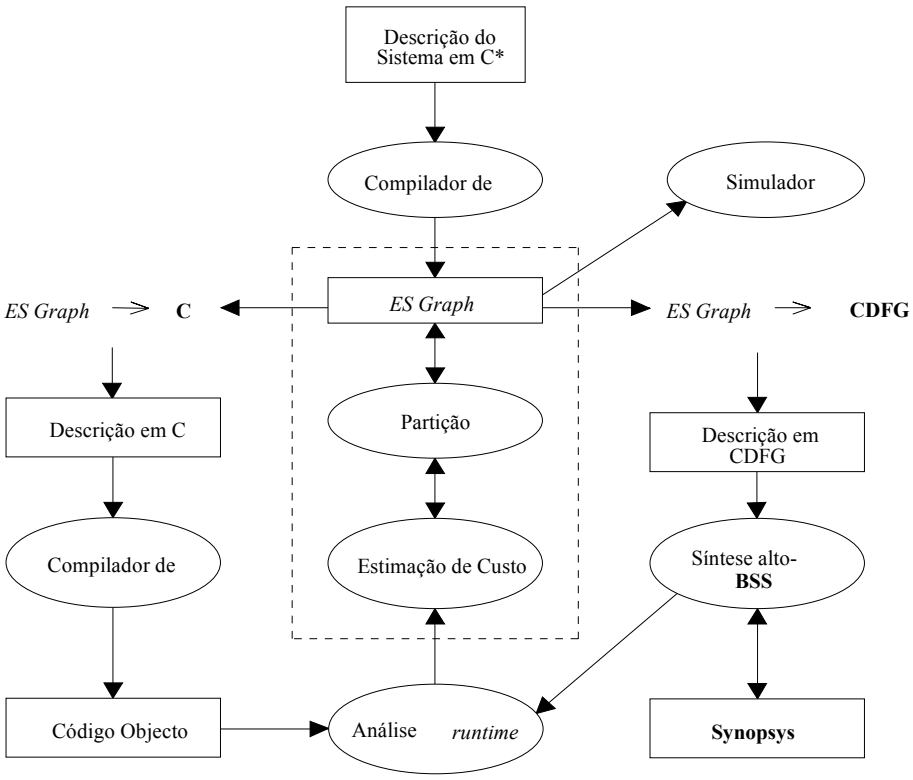


Figura 1.9. Sistema COSYMA com o sistema de síntese de co-processadores (BBS).

A funcionalidade do sistema a sintetizar é descrita numa linguagem textual (programa de *software*), denominada de C^x, que é uma extensão ou superconjunto da linguagem de programação C, ao adicionar, como extensões mais significativas [34], o suporte de restrições de desempenho e processos concorrentes (conceptualmente idênticos aos processos do VHDL). O programador engloba no código o conceito de tarefa, comunicação entre tarefas, directivas temporais (atrasos mínimos e máximos entre

etiquetas da tarefa) e directivas para o processo de co-síntese. Assim que o sistema tenha sido especificado é possível obter o código puramente *software* compilado, eventualmente com desempenho inferior ao pretendido.

O exemplo seguinte, retirado de [35], mostra um segmento de código para calcular o factorial de um número, no qual o programador limitou, através de uma directiva de restrição, o tempo de execução do ciclo em 5μs:

```
from lab1 to lab2: max 5us;
int j;
f=1;
lab1:
for(i=1; i<n; i++)
      f*=i;
lab2:
```

O compilador de C^x traduz o modelo do sistema para um modelo de grafos, intitulado de "ES *graph*" (***E**xtended **S**yntax graph*), baseado na representação de grafos de controlo e fluxo de dados (CDFG). De seguida um algoritmo de decomposição identifica os blocos computacionalmente intensivos e, quando as restrições temporais são violadas, segmentos de código são movidos iterativamente para implementações em *hardware* de aplicação específica (componentes denominados pelos autores de co-processadores, por estarem dependentes da actuação do processador). Os blocos extraídos, são implementados por síntese ou por *hardware* definido pelo utilizador. O projectista pode indicar determinadas funções que não podem migrar para o *hardware*. Desta forma, mesmo após a implementação dos blocos em *hardware* é possível proceder a alterações destas funções. O sistema contorna deste modo o problema que se põe devido à existência de construções em C que não podem ser mapeadas em *hardware* (ex. estruturas dinâmicas de dados). Na metodologia descrita, orientada por *software* que contém todo o conjunto de C, implica que estas construções sejam excluídas de futuras implementações em *hardware*.

O sistema resultante após a partição é constituído por um processador, co-processadores, e periféricos. Os componentes *hardware* eram, inicialmente, sintetizados pelo sistema de síntese OLYMPUS [36], com base na descrição em

HardwareC (gerada pelo COSYMA). Para o componente *software* é gerado código em ANSI-C, que é compilado para o processador SPARC[20] (usando o compilador GNU).

De molde a diminuir o largo espaço de projecto é estabelecido um conceito chamado de extracção de *hardware*. Durante a partição é utilizada uma função de custo que favorece a implementação em *hardware* de funções que se traduza na melhoria do desempenho. Os benefícios da migração de um determinado bloco são estimados, tendo em conta os custos de comunicação entre as unidades. Esta função de custo codifica conhecimentos de síntese, de compiladores e de bibliotecas. Podem ser utilizadas diferentes funções de custo que resultam em diferentes extracções. Como exemplo os autores apresentam uma função de custo que extrai co-processadores para blocos computacionalmente intensivos do sistema, em especial ciclos. Esta identificação é executada na representação intermédia do sistema (modelo de grafos ES). Contudo a função de custo, não adiciona os custos do *hardware*.

A representação intermédia (*ES graph*) gerada a partir da descrição inicial em C^x, é uma combinação de grafos de sintaxe, tabelas de símbolos e grafos de fluxo de dados. A razão da extensão do modelo de grafos [35] deve-se ao facto dos grafos de fluxo de dados serem inapropriados para a representação de estruturas dinâmicas e processos paralelos. Os grafos de sintaxe oferecem essa possibilidade e podem ser transformados facilmente em HardwareC ou ANSI-C. Contudo, devido aos processos de partição e estimativa terem necessidade da informação acerca das dependências entre dados são necessários modelos de grafos que incorporem os grafos de fluxo de dados e grafos de sintaxe.

A ferramenta particiona ao nível do bloco básico ou da função (incluindo chamadas hierárquicas a funções). O algoritmo de partição utilizado (baseado no algoritmo *"simulated annealing"*) é executado com base na minimização de uma função de custo calculada em cada iteração do algoritmo com base na estimativa do tempo de execução dos blocos de *software* (estimado por uma ferramenta de análise temporal), dos blocos de *hardware* (estimado pelo escalonamento no grafo ES), comunicação

[20] Marca registada da Sun Microsystems.

(estimado pelo grafo de fluxo de dados), e custos de *hardware* (com base no número de blocos movidos para implementações *hardware*). O custo da migração de um bloco para o hardware é definido incrementalmente pela expressão (2.2).

$$dc(B) = \omega \times \left[t_{HW}(B) - t_{SW}(B) + t_{com}(Z) - t_{com}(Z \cup B) \right] \times It(B) \qquad (2.2)$$

Em que ω, representa um factor de peso destinado a controlar o algoritmo de partição, $t_{HW}(B)$ representa o tempo de execução estimado do bloco B no co-processador, $t_{SW}(B)$ representa o tempo de execução estimado do bloco B no processador, Z representa o conjunto de blocos atribuídos ao co-processador, $t_{com}(Z)$ representa o atraso introduzido pela comunicação entre o processador e o co-processador, e $It(B)$ representa o número de vezes que o bloco B é executado. A função de custo (2.2) representa a aceleração produzida pela extracção de um bloco B para o *hardware*. Em [11] os autores apresentam um factor ω calculado por uma expressão exponencial, que conduz a execução estimada do sistema T_s em direcção ao tempo de execução requerido T_c, com um número mínimo de blocos implementados no co-processador. Este factor muda de sinal próximo das restrições para evitar a migração desnecessária de mais blocos para o *hardware*.

Depois de realizada a partição, os blocos de *software* são traduzidos para um programa em C, que contém o código necessário para a comunicação com o co-processador. Os restantes blocos são convertidos para HardwareC sintetizável, que também contém o código necessário para a comunicação com o processador de propósito geral. O co-processador pode executar blocos independentes [37]. Cada chamada deve indicar a funcionalidade a executar. O último passo é uma rápida análise temporal de todo o sistema em conjunto.

O sistema alvo, baseado na arquitectura comum de sistemas embebidos, é constituído por um processador genérico (SPARC[21]), memória[22] e *hardware* personalizado (definido anteriormente como co-processador). O SPARC e o *hardware* personalizado

---

[21] LSI SPARC a 33 Mhz com unidade de vírgula flutuante implementado pela LSI Logic.

[22] memória RAM com um ciclo de acesso rápido.

executam em exclusão mútua, com a comunicação[23] entre os dois integrados via memória. No primeiro sistema, o *hardware* personalizado actua como co-processador ou como módulo de interface definido pelo utilizador.

Como exemplo de aplicação é apresentado pelos autores o algoritmo de "Chromakey" para televisão de alta definição (HDTV) implementado num processador SPARC e num ASIC, com a qual obtêm uma melhoria de um factor de 3. No algoritmo é identificado um ciclo crítico que consome 90% do tempo de execução do *software* inicial. A solução apresentada implementa o ciclo num ASIC com 17000 portas lógicas.

As principais vantagens e desvantagens do sistema COSYMA, identificadas em [15], são:

- A metodologia do COSYMA tem como grandes vantagens: a especificação do sistema ser feita numa linguagem que facilita a descrição de sistemas complexos; a permissão de todas as construções da linguagem de programação de *software* C.

- Uma das desvantagens é a impossibilidade da execução em paralelo do processador e dos integrados personalizados, desprezando uma grande vantagem da utilização de co-processadores.

- Outra grande desvantagem, centra-se no facto da estimativa do desempenho do *software* não ter em conta o efeito das optimizações feitas pelos compiladores, podendo os sistemas *hardware*/*software* resultantes produzirem desempenhos mais fracos do que soluções unicamente *software*.

Todo o processo de partição e mapeamento em COSYMA é conduzido por resultados de estimativas, que originam resultados sub-óptimos. Exemplos com o COSYMA têm originado discrepâncias entre as estimativas e os tempos de execução do sistema final. Estas discrepâncias devem-se, sobretudo, aos efeitos da compilação do *software* serem difíceis de prever, pois dependem do próprio compilador utilizado, e do tempo de

---

[23] protocolo do tipo CSP (***C**ommunicating **S**equencial **P**rocesses*).

execução do *hardware* sintetizado ser de difícil predição, pois depende da própria ferramenta de síntese e do grau de optimização.

Recentemente, de forma a atenuar os problemas citados, têm sido investigados métodos para adaptar as estimações ineficientes aos custos reais [38]. Esta metodologia baseia-se no recomeço do processo de partição, adaptando as estimativas, com base na história acumulada das iterações anteriores. Os resultados mostram uma rápida convergência entre as estimativas e os custos reais.

Devido ao processo de síntese do co-processador ser bastante demorado (várias horas de tempo de CPU numa SPARC 10/41) e portanto o processo iterativo ser árduo, os autores decidiram substituir o processo de síntese, que utilizava o OLYMPUS e a descrição do sistema a sintetizar em HardwareC, por um sistema de síntese de co-processadores implementado pelos próprios, designado por BSS[24] [39] que na última fase usa as ferramentas de síntese da Synopsys. A ferramenta (ver Figura 1.9) aceita uma descrição do sistema a sintetizar em CDFG e executa diversas tarefas antes de enviar as descrições a optimizar para o sistema de síntese da Synopsys. As tarefas executadas são o escalonamento, a alocação e geração do controlo para o *pipeline*, o *pipeline* de operandos, o *pipeline* de ciclos, e a computação especulativa com múltipla predição de saltos [40]. A inserção de *pipeline* permite a utilização do mesmo ciclo de relógio para o co-processador (33 MHz), facilitando a sincronização entre os dois integrados.

Exemplos de aplicações podem ser encontrados em [39], com os resultados a mostrarem aumentos de desempenho de 2.66 até 9.65 (incluindo a comunicação entre integrados). Estes resultados melhoram os apresentados na primeira versão do sistema COSYMA (na qual alguns exemplos surgiam com uma diminuição de desempenho [11] devido a deficientes estimativas).

---

[24] Do Inglês ***B****raunschardwareeig* ***S****ynthesis* ***S****ystem.*

A simulação de sistemas ao nível lógico é uma tarefa que consome muito tempo de CPU. Por este motivo em [41] é apresentado um sistema de prototipagem que funciona como um emulador do sistema final. O sistema de prototipagem é uma réplica da arquitectura alvo do COSYMA. É constituído por memória RAM, uma placa com um processador SPARC, uma placa com 4 FPGAs XILINX XC4010 e, na existência de multiplicações, com um AMD 29C323 (para emulação do co-processador), e uma placa com um Motorola MC 68332 (para medidas temporais, depuração e ligação ao computador).

Os autores apresentam exemplos de *benchmarks* destinadas a medir o desempenho. Em [11] são apresentadas 3 *benchmarks* (Diesel[25], Smooth[26] e 3d[27]), com o objectivo de produzir um aumento de desempenho[28]. Para as duas primeiras *benchmarks* foram obtidos aumentos de desempenho de 1.4 e 1.3. Para a última foi obtida uma diminuição de desempenho de 0.9. A percentagem do tempo de execução na comunicação entre componentes foi de 9.9%, 49.6% e 13.8% respectivamente [37]. Em [39] são apresentados os mesmos exemplos implementados com o fluxo de projecto iterativo (ilustrado na Figura 1.9). Neste caso não são obtidas implementações mais lentas do que as estimativas, sendo apresentados aumentos de desempenho, para alguns casos, muito próximos de 10.

O sistema já soluciona quase todas as desvantagens reveladas em [15]. Contudo, continua a não permitir a execução do co-processador em paralelo. Os autores referem que, eventualmente, desejariam utilizar outro tipo de circuitos: estruturas primitivas de interface (contadores, temporizadores, etc), estruturas de interface mais complexas (interfaces a barramentos), e processadores mais pequenos (possibilitando a síntese de alto nível e a exploração com mais do que um processador de propósito geral).

---

[25] Algoritmo de tempo-real para controlo digital de um carregador do turbo de um motor diesel.

[26] Executa um algoritmo de filtragem de uma imagem digital.

[27] Exemplo de parte de um algoritmo de animação 3d para demonstrar que a co-síntese automática não é uma tarefa trivial.

[28] No Inglês *speedup*: relação entre o tempo de execução da implementação apenas em *software* e o tempo de execução da implementação mista *hardware*/*software*.

1.6.7 VULCAN

A ferramenta designada por VULCAN [42] foi desenvolvida na Universidade de Stanford e é constituída por aproximadamente 60000 linhas de código C. O sistema foi elaborado de modo a integrar o Olympus e as ferramentas de compilação e simulação de código para o processador DLX [25], e permite o percurso completo desde a especificação do sistema em HardwareC até à síntese de *hardware* e de *software*.

Em [42], [43] e [12] é apresentado este sistema de co-síntese, cujo fluxo de projecto é ilustrado na Figura 1.10. A funcionalidade do sistema a implementar é especificada numa linguagem de descrição de *hardware* (HDL[29]), designada por HardwareC (conceptualmente o VHDL ou Verilog poderiam também ser utilizados) que é posteriormente compilada para um modelo de grafos baseado em diagramas de fluxo de dados [44]. A linguagem comporta, em comparação com a linguagem C, a declaração de inteiros como único tipo de dados. Não permite estruturas de dados dinâmicas, mas adiciona construções de grande utilidade para a descrição de *hardware*.

A entrada para o sistema é constituída por dois componentes: a descrição da funcionalidade do sistema e o conjunto de restrições. O último engloba as restrições e directivas paramétricas utilizadas pelo processo de co-síntese. As restrições temporais são especificadas na linguagem (por meio de atributos), usando anotações com rótulos (designados pelos autores por *tags*) em operações individuais:

.attribute ´´constraint <minrate|maxrate> [<num>] of <tag> = <num> cps´´	restrições de ritmo
.attribute ´´loop-index <str> [<num>]´´	índice da variável
.attribute ´´clock <str> [<num>]´´	sinal de relógio
.constraint mintime from <tag> to <tag> = <num> cycles	atraso mínimo
.constraint maxtime from <tag> to <tag> = <num> cycles	atraso máximo
.constraint finish|finishedby|before|during <tag> <tag>	

O código do Exemplo 1.1, retirado de [42], apresenta a especificação de restrições de ritmo de dados (que traduzem a frequência com que os dados necessitam ser produzidos ou consumidos) na presença de ciclos encadeados.

---

[29] Do Inglês ***H**ardware **D**escription **L**anguage*.

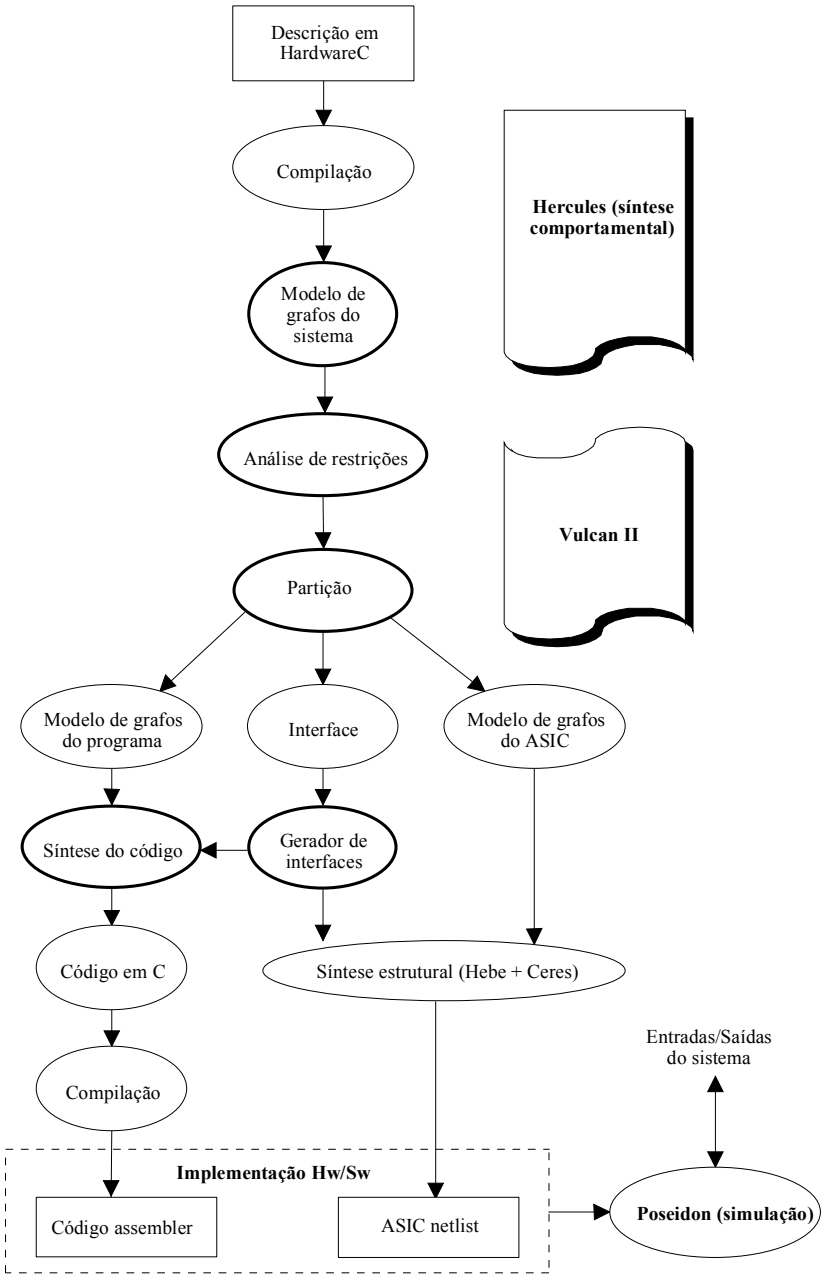


Figura 1.10. Sistema VULCAN.

Depois de capturada, a especificação é compilada para o modelo de grafos sequencial (do Inglês *sequencing graph model*) usando o Hercules, e este modelo é traduzido para o modelo de grafos de fluxo bilógico [42]. Este modelo é constituído por vértices (nós) que representam operações e por ligações (arcos) que representam dependências de dados, dependências de controlo, ou restrições temporais. Na Figura 1.11 é apresentado um modelo de grafos, no qual se encontram capturadas as restrições temporais máximas e mínimas, e restrições de ritmos (frequência com que são executadas determinadas operações). Nos arcos entre nós são ilustradas as restrições

temporais mínimas, enquanto que as restrições temporais máximas são capturadas com arcos em sentido contrário entre os dois nós que representam as operações. As restrições de ritmo são capturadas em arcos com saída e entrada no mesmo nó.

```
process example (frameEN, bitEN, bit, word)
      in port frameEN, bitEN, bit;
      out port word[8];
{
      boolean store[8], temp;
      tag A;
      while (frameEN)
      {
            while (bitEN)
            {
A:                temp = read(bit);
                  store[7:0] = store[6:0] @ temp;
            }
      write word = store;
      }
      attribute "constraint minrate of A = 100 cycles/sample;
      attribute "constraint minrate 0 of A = 1 cycles/sample;
      attribute "constraint minrate 1 of A = 10 cycles/sample;
}
```

Exemplo 1.1. Código HardwareC com restrições de ritmo.

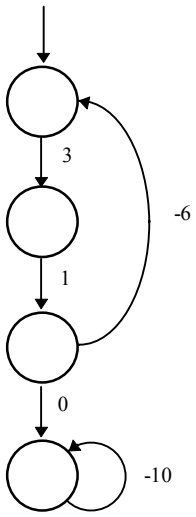


Figura 1.11. Modelo de grafos de um exemplo genérico.

Este diagrama de fluxo de dados é entrada para o VULCAN-II, que produz as partições de modo a satisfazer as restrições impostas. A partição depende da identificação de operações com tempo de execução variável (não determinístico) e da possibilidade de satisfação de restrições que limitem operações deste tipo. Com base neste diagrama são estimados os desempenhos de cada operação individual quando

implementada em *hardware* ou em *software* (com base no processador que o executará). O modelo de custo captura as características do processador (como função do tempo de execução, do endereçamento em memória e do tempo de acesso a esta, e do tempo de resposta a interrupções) e das implementações em *hardware* ou *software*.

O modelo de grafos é entrada para um conjunto de transformações por partição que geram dois conjuntos de grafos de fluxo, um representa a imagem de *hardware*, o outro a imagem de *software*. A heurística de partição tem como base uma partição inicial e é orientada na direcção de minimização da função de custo (2.3) [45].

$$f(\omega)= a_1 \times S_H(\Phi_H) - a_2 \times S^{\pi}(\Phi_S) + b \times \mathrm{B} - c \times \mathrm{P} + d \times |m| \qquad (2.3)$$

Em que $\omega(\Phi)= \Phi_S \cup \Phi_H$ representa uma partição em dois conjuntos de grafos, |m| define o tamanho acumulado das variáveis m, que foram transferidas pela partição, e a_1, a_2 , b, c e d são constantes positivas. A partição obtida satisfaz: as restrições temporais de todos os grafos de fluxo de dados contidos em Φ_S e Φ_H; a utilização do processador $P \leq 1$; e a utilização do barramento $B \leq \overline{B}$. S_H representa o tamanho do *hardware* e S^{Π} o tamanho do *software*.

Para acelerar a determinação da partição óptima, a expressão (2.3) é apenas calculada para a partição inicial. Para cada partição posterior do processo iterativo as modificações da função de custo são definidas incrementalmente:

$$\Delta f = a_1 \times \Delta S_H - a_2 \times \Delta S^{\pi} + b \times \Delta \mathrm{B} - c \times \Delta \mathrm{P} + d \times \Delta |m| \qquad (2.4)$$

Como resultado da partição do sistema são obtidos conjuntos de modelos *hardware* e *software* concorrentes. O componente *software* é constituído por rotinas de execução concorrente designadas por *threads*. Todas estas rotinas começam por um ponto de sincronização (que normalmente são operações que por dependerem dos dados podem ter atrasos ilimitados) e são escalonadas dinamicamente. Contudo dentro de um *thread* todas as operações são escalonadas estatisticamente. Para um determinado componente re-programável o tempo de execução de cada *thread* é conhecido estatisticamente. A sincronização do *software* é feita sequencialmente. Em [46] são descritos os esquemas de comunicação e sincronização utilizados pelo sistema. A implementação de um modelo de grafos em várias rotinas depende dos pontos de

sincronização. Um grafo simples sem múltiplos pontos de sincronização pode ser implementado como uma simples rotina. Um sistema hierárquico é implementado como um conjunto de rotinas, em que cada rotina corresponde a um grafo na hierarquia de modelos. O componente *software* é constituído por um conjunto de *threads* de programas. Estes *threads* podem estar relacionados hierarquicamente, ou podem necessitar de execução concorrente. No último caso a concorrência é obtida usando um modelo de execução de concorrência intercalada[30]. A sincronização do *software* é necessária para assegurar a ordem correcta de operações dentro e entre diferentes *threads* de programa.

A implementação do *hardware* é realizada por ferramentas de síntese de alto-nível. A ferramenta gera a componente s*oftware* em código máquina, a partir dos grafos de fluxo. Devido às diferenças significativas em termos de abstracção entre estas duas representações [44], esta tarefa é feita em passos que culminam no uso do compilador de C para *assembler*.

As unidades de interface *hardware*/*software* são geradas automaticamente. Estas unidades fazem o interface (sincronização e comunicação) entre o processador de propósito geral e o *hardware* de aplicação específica (ASICs).

Para possibilitar diferentes ritmos de execução dos componentes (*hardware* e *software*) e devido à existência de operações não determinísticas, é utilizado o escalonamento dinâmico de diferentes *threads* que tenham diferentes tempos execução. O escalonamento é realizado por uma estrutura de controlo de FIFO[31] que força o consumo de dados à medida que são produzidos [45]. O interface é constituído por filas de dados em cada canal e um FIFO que armazena os identificadores para habilitar a *thread* quando os dados de entrada chegarem. O tamanho máximo do FIFO traduz o número máximo de *threads* no sistema. A unidade de controlo do FIFO e o próprio FIFO podem ser implementados em *software* ou num dos ASICs do sistema. Em [42] são apresentados os custos de cada uma das soluções de interface para um exemplo.

---

[30] Em Inglês *interleaved.*

[31] Do Inglês ***F**irst **I**n **F**irst **O**ut.*

As estimativas são resultado do conhecimento do processador a utilizar, e do nível de optimização do compilador. O atraso de uma determinada operação é determinado pela sequência de instruções e pelos atrasos associados a essa sequência. Estes dependem do processador utilizado. Para generalizar o sistema, a informação específica ao processador é capturada para um modelo de custo [44], geral e flexível. Este modelo é uma especificação de um novo conjunto de instruções que engloba instruções do conjunto de instruções do processador utilizado e macros formadas por sequências de instruções. Estas macros tornam a compilação mais eficiente, preservam a atomicidade de algumas operações do modelo de grafos e ajudam a estimar o tempo de execução do programa.

A arquitectura alvo é constituída por um processador, integrados de aplicação específica, e um nível de memória. Como processador é utilizado o DLX com as ferramentas de compilação e simulação integradas no sistema de co-síntese.

Para verificar se a implementação é correcta é utilizado o POSEIDON, um simulador que executa concorrentemente múltiplos modelos funcionais implementados em *hardware* ou *software*, e que permite a simulação ao nível de grafos ou ao nível lógico. Esta ferramenta suporta a simulação de código *assembler* para o microcontrolador 8051 ou para o processador DLX.

1.7 Conclusões

Foram descritas várias ferramentas que se julgam abrangentes, no sentido em que abordam os vários problemas do projecto e síntese *hardware*/*software* de um modo integrado, desde a especificação até à verificação e simulação do sistema implementado. Ferramentas em que a especificação do sistema é elaborada numa linguagem de descrição de *hardware*, numa linguagem de descrição de *software* ou num modelo de grafos.

Para sistemas embebidos, em que as aplicações são específicas e conhecidas pelo ciclo de projecto, a implementação da unidade de processamento optimizada para o código em questão, pode apresentar-se sobredimensionada face ao desempenho pretendido. Por este motivo, a análise do código deve também ser conduzida por restrições

temporais, que reduzam o espaço de exploração do projecto aos segmentos de código em questão.

As arquitecturas alvo do PRISM-I e do sistema de co-síntese para simulação de circuitos digitais desenvolvido na Universidade de Stanford, baseadas na utilização de agregados reconfiguráveis, são extremamente versáteis e flexíveis para sistemas de emulação com o objectivo de acelerar aplicações genéricas. Contudo, para aplicações específicas (de funcionalidade fixa) em que se preveja um elevado número de vendas, os resultados podem ser optimizados utilizando dispositivos não reconfiguráveis.

Para os sistemas de co-síntese descritos a arquitectura alvo é constituída por apenas um processador (solução monoprocessador). O uso de arquitecturas multiprocessador requer *software* adicional de sincronização e protecção de memória para facilitar a multisegurança, e é pelos motivos apresentados um problema de difícil resolução, embora se suponha o desenvolvimento de sistemas baseados nestas arquitecturas brevemente.

A maioria dos sistemas [43] permite apenas um nível de memória, facilitando a análise e a síntese. Nas arquitecturas apresentadas o processador detém sempre o controlo do barramento (*master*), pois muitos dos processadores incluem já facilidades para esse controlo e a integração destas funcionalidades num ASIC aumentaria o custo do *hardware*.

A comunicação entre os vários elementos que constituem estes sistemas mistos é um problema muito importante e de difícil resolução. A comunicação entre os dois componentes (*hardware-software*) desempenha um papel fundamental no tempo de execução e na partição. A dependência cresce quando se desce para baixos níveis de granularidade. A sincronização devido a diferentes ritmos de execução de cada componente e a operações com execução não determinística, necessita de uma solução com escalonamento dinâmico sempre que se pretenda a execução concorrente entre componentes [46].

O efeito da comunicação na escolha da partição óptima aumenta quando se desce para baixos níveis de granularidade. A ferramenta VULCAN permite o nível de granularidade de instrução, parte da implementação do sistema em *hardware*, e

optimiza a solução de modo a maximizar a migração de operações para o *software*. A ferramenta COSYMA particiona também ao nível da instrução. No começo todos os nós são mapeados em *software* e a migração para *hardware* (com um algoritmo baseado no *simulated annealing*) é feita de modo a satisfazer as restrições minimizando o *hardware*. A ferramenta Design Assistant permite a partição ao nível de tarefas e é orientada por uma heurística eficiente (como comprovam os resultados apresentados pelo autor). Os nós são em primeiro lugar mapeados em *software* e depois alguns movidos para o *hardware* até à satisfação das restrições temporais com um mínimo de área total do *hardware*.

Por razões de flexibilidade, redução do ciclo de projecto, e fiabilidade, a arquitectura alvo dos sistemas embebidos baseia-se em um microprocessador (responsável pela implementação do componente *software*). No capítulo seguinte é descrita a arquitectura do microprocessador, na qual se baseia o microprocessador projectado e usado para implementação do componente *software* no ambiente de co-síntese proposto neste trabalho.

3. Microcontrolador PIC

Para a realização do componente *software* do sistemas de co-síntese escolheu-se um núcleo de processamento, baseado na arquitectura dos microcontroladores da família PIC, que permite alguma versatilidade e, por isso, torna viável o projecto de um microprocessador parametrizável conforme os requisitos da aplicação.

Por este facto este capítulo descreve resumidamente a arquitectura da família de microcontroladores PIC, com ênfase na gama PIC16C5X [2]. As razões desta selecção são apresentadas e alicerçadas ao longo de todo o capítulo, no qual são explicadas a arquitectura interna, o conjunto de instruções, e as operações mais relevantes destes microcontroladores.

3.1 Introdução

Os microcontroladores têm um conjunto de instruções específico, incorporam uma solução num único circuito integrado, têm facilidades de manipulação de *bits*, de portos de E/S, etc. A maioria dos microcontroladores são do tipo CISC[1], que tem a propriedade de possuir um conjunto alargado de instruções. No entanto, graças ao reduzido conjunto de instruções, elevados desempenhos, menor complexidade do integrado, menor área de silício, e menor potência dissipada, a arquitectura RISC[2] tem

[1] Do Inglês ***C**omplex **I**nstruction-**S**et **C**omputer*. Em Português computador com conjunto complexo de instruções.

[2] Do Inglês ***R**educed **I**nstruction-**S**et **C**omputer*. Em Português computador com conjunto reduzido de instruções.

tendência a ser utilizada no projecto de microcontroladores. Os PIC são um exemplo de microcontroladores com uma arquitectura próxima dos RISC.

Os microcontroladores PIC cobrem um hiato entre a lógica discreta e os restantes Micros mais poderosos (ex. 68xx ou 80xx). Pela arquitectura simples, pela existência de elementos da família com poucos pinos (reduzidas dimensões), pelo desempenho, pelo reduzido conjunto de instruções (facilmente memorizável), e pelo preço reduzido, este tipo de microcontrolador de 8 *bits* é excelente solução para aplicações embebidas de pequeno/médio porte ou para substituição de um conjunto de lógica discreta.

Ultimamente têm surgido inúmeras aplicações para os microcontroladores PIC que ilustram a popularidade e o interesse dos projectistas por este tipo de solução [47]. Este facto deve-se à óptima relação preço/desempenho e à existência de várias ferramentas de baixo custo disponíveis para o desenvolvimento de aplicações. Como exemplo de utilização académica salienta-se o estudo deste microcontrolador nas disciplinas relacionadas com arquitectura de microprocessadores do Instituto Superior Técnico.

A família PIC divide-se em três gamas de microcontroladores [18]: alta (gama PIC17CXX), média (gama PIC16CXX), e básica (gama PIC16C5X). As gamas existentes oferecem diversas opções: o tamanho da memória de programa e comprimento de instrução; o número de registos do ficheiro de registos; o número de portos de entrada/saída (E/S); o encapsulamento; a existência de linhas de interrupção, etc.

3.2 Arquitectura

A família de microcontroladores PIC tem como características gerais:

- Arquitectura baseada nos RISC. Reduzido conjunto de instruções ortogonal, não havendo discriminação no acesso a um registo genérico do ficheiro de registos.

- Arquitectura Harvard (barramento de dados independente do barramento de instruções), que possibilita o *pipeline* entre a procura[3] e a execução da instrução.
- Todas as instruções de igual tamanho, com tempo de execução igual (1 ciclo) à excepção de instruções de salto incondicional, instruções que afectam o PC[4], chamadas a sub-rotinas, ou instruções de salto condicional quando a condição é verdadeira (2 ciclos).
- Ficheiro de registos constituído por 32 ou mais registos de 8 *bits*.
- Memória de programa interna (EPROM ou ROM).
- Temporizador do "cão de guarda" e contador/relógio de tempo-real internos.
- Modo de endereçamento directo, indirecto e relativo.
- Pinos de E/S oriundos de registos específicos do ficheiro de registos controlados *bit* a *bit* ou inteiramente.
- Instrução que coloca o microcontrolador em modo de adormecimento e que desactiva parte do integrado (economia no consumo de potência).

O elemento mais simples da gama de microcontroladores PIC16C5X, designado por PIC16C54 e arquitectura apresentada na Figura 3.1, possui 33 instruções (12 *bits* de comprimento), 512 palavras de 12 *bits* como limite máximo de memória de programa, pilha[5] com 2 níveis, 32 registos de 8 *bits*, frequência de relógio permitida de DC a 20 Mhz, 12 linhas de E/S (8 *bits* para o porto B e 4 *bits* para o porto A), 1 temporizador de "cão de guarda" (WDT[6]), 1 contador/relógio de tempo-real, e 18 pinos de encapsulamento.

---

[3] Em Inglês designada por *fetch*.

[4] Do Inglês ***P**rogram **C**ounter*. Em Português contador de programa.

[5] Em Inglês *stack*.

[6] Do Inglês ***W**atch**D**og **T**imer*.

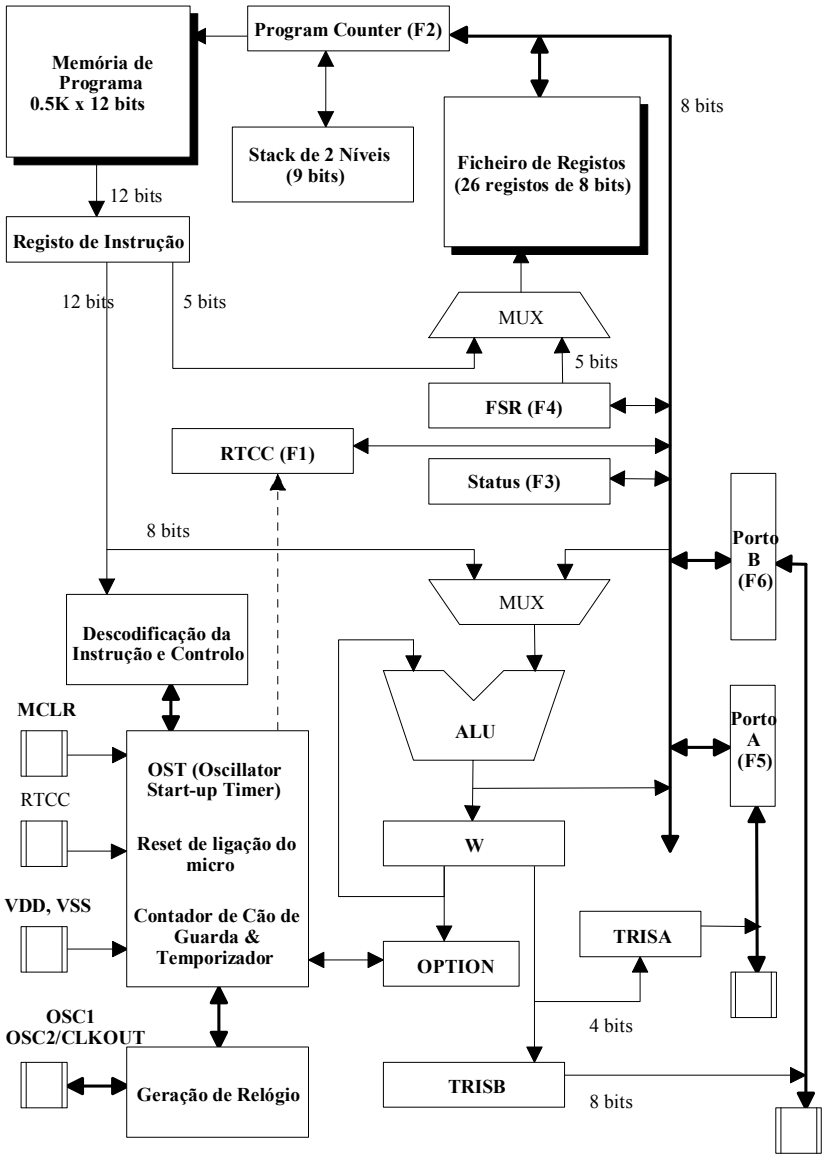


Figura 3.1. Diagrama de Blocos da arquitectura interna do PIC16C54.

3.2.1 Instruções

Nas tabelas seguintes (referentes aos diversos formatos de instruções) apresentam-se: o código para cada instrução, a mnemónica utilizada pelos assembladores da Microchip™, a operação realizada, as *flags* afectadas por cada uma das instruções, e a descrição textual. A Figura 3.2 ilustra o formato das instruções que operam sobre o *byte* de um registo do ficheiro de registos, e a Tabela 3.1 descreve as instruções com este formato.

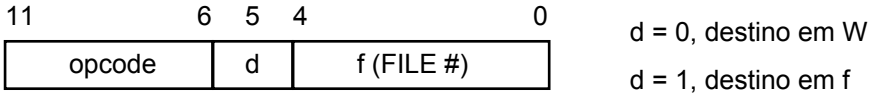


Figura 3.2. Formato de cada instrução sobre o ficheiro de registos orientada ao *byte*.

opcode (Bin)	Mnemónica	Operação	Registo de *Status*	Descrição
0001 11df ffff	**ADDWF f,d**	W + f → d|f	C, DC, Z	Soma o conteúdo de W com o registo f (um registo do ficheiro de registos).
0001 01df ffff	**ANDWF f, d**	W & f → d|f	Z	"AND" de W com o registo f.
0000 011f ffff	**CLRF f**	0 → f	Z	Coloca a zero o registo f.
0000 0100 0000	**CLRW -**	0 → W	Z	Coloca a zero o registo W.
0010 01df ffff	**COMF f, d**	~f → d|f	Z	Complementa o registo f.
0000 11df ffff	**DECF f, d**	f - 1 → d|f	Z	Decrementa o registo f.
0010 11df ffff	**DECFSZ f, d**	f - 1 → d|f	-	Decrementa o registo f. Se o resultado for zero salta a próxima instrução.
0001 00df ffff	**IORWF f, d**	W || f → d|f	Z	"OR" do registo W com o registo f.
0010 10df ffff	**INCF f, d**	f + 1 → d|f	Z	Incrementa o registo f.
0011 11df ffff	**INCFSZ f, d**	f + 1 → d|f	-	Incrementa o registo f. Se o resultado for zero salta a próxima instrução.
0010 00df ffff	**MOVF f, d**	f → d|f	Z	O conteúdo do registo f é movido.
0000 001f ffff	**MOVWF f**	W → f	-	Move o conteúdo de W para o registo f.
0000 0000 0000	**NOP -**	-	-	Nenhuma operação.
0011 01df ffff	**RLF f, d**	f(n) → [d|f](n+1), C → [d|f](0), f(7) → C	C	Rotação de um *bit* para a esquerda do conteúdo do registo f. O *bit* de transporte é envolvido na rotação.
0011 00df ffff	**RRF f, d**	f(n) → [d|f](n-1), C → [d|f](7), f(0) → C	C	Rotação de um *bit* para a direita do conteúdo do registo f. O *bit* de transporte é envolvido na rotação.
0001 10df ffff	**XORWF f, d**	W ⊕ f →d|f	Z	"XOR" de W com o registo f.
0000 10df ffff	**SUBWF f, d**	f - W → d|f	C, DC, Z	Subtrai W ao registo f em complemento para dois.
0011 10df ffff	**SWAPF f, d**	f(0-3)↔ f(4-7) → d|f	-	Troca no registo f os 4 bms com os 4 bMs.

Tabela 3.1. Instruções sobre o ficheiro de registos orientadas ao *byte*.

A Figura 3.3 ilustra o formato das instruções que operam sobre o imediato representado pelos 8 *bits* menos significativos das instruções deste tipo. A Tabela 3.2 descreve as instruções com este formato.

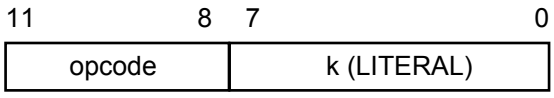


Figura 3.3. Formato de instruções com imediatos e de controlo.

opcode (Bin)	Mnemónica	Operação	Registo de *Status*	Descrição
1110 kkkk kkkk	**ANDLW k**	k & W → W	Z	"AND" de W com o imediato de 8 *bits*, k. O resultado é colocado no registo W.
1001 kkkk kkkk	**CALL k**	PC + 1 → Pilha (PC + 1 → Topo da pilha), k → PC<7:0>, '0' → PC<8>	-	Chamada a uma sub-rotina. Primeiro, o endereço de retorno (PC+1) é colocado na Pilha. O valor de 8 *bits*, k, é carregado no PC<7:0>. O *bit* 8 do PC é colocado a zero.
0000 0000 0100	**CLRWDT -**	00h → WDT, 0 → WDT	TO, PD	*Reset* do WDT e também *reset* do pré-escalar do WDT se estiver atribuído. Os *bits* de *status*, TO e PD, são colocados a um.
101k kkkk kkkk	**GOTO k**	k → PC<8:0>	-	Coloca no PC o conteúdo de k.
1101 kkkk kkkk	**IORLW k**	k || W → W	Z	"OR" de W com os 8 *bits* do imediato k. O resultado é colocado no registo W.
1100 kkkk kkkk	**MOVLW k**	k → W	-	Os 8 *bits* do imediato k são colocados no registo W.
0000 0000 0010	**OPTION -**	k → OPTION	-	Os 6 bms do registo W são carregados no registo OPTION.
1000 kkkk kkkk	**RETLW k**	k → W, Pilha → PC (TOS → PC)	-	O registo W é carregado com os 8 *bits* do imediato k. O PC é carregado do topo da Pilha (o endereço de retorno). Esta instrução demora 2 ciclos.
0000 0000 0011	**SLEEP -**	0 → PD, 1 → TO; 00h → WDT, 0 → WDT	TO, PD	O *bit* de *status* PD (*power down*) é colocado a zero. O *bit* de *status* TO (*time-out*) é colocado a um. O WDT e o pré-escalar se lhe estiver atribuído são colocados a zero. Desliga os relógios internos (modo de adormecimento).
0000 0000 0fff	**TRIS f**	W → TRIS do porto f	-	O registo TRIS de f (f = 5, 6, ou 7) é carregado com o conteúdo do registo W.
1111 kkkk kkkk	**XORLW k**	k ⊕ W → W	Z	XOR de W com os 8 *bits* do imediato k. O resultado é colocado no registo W.

Tabela 3.2. Instruções com imediatos e de controlo.

A Figura 3.4 ilustra o formato das instruções que operam sobre um dos *bits* de um registo do ficheiro de registos. A Tabela 3.3 descreve as instruções com este formato.

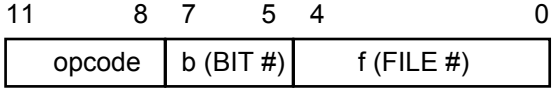


Figura 3.4. Formato de cada instrução sobre o ficheiro de registos orientada ao *bit*.

opcode (Bin)	Mnemónica	Operação	Registo de *Status*	Descrição
0100 bbbf ffff	**BCF f, b**	0 → f(b)	-	O *bit* b do registo f é colocado a zero.
0101 bbbf ffff	**BSF f, b**	1 →f(b)	-	O *bit* b do registo f é colocado a um.
0110 bbbf ffff	**BTFSC f, b**	Testa *bit* b do ficheiro f: Salta se zero.	-	Se o *bit* b do registo f é igual a zero, então salta a próxima instrução.
0111 bbbf ffff	**BTFSS f, b**	Testa *bit* b do ficheiro f: Salta se um.	-	Se o *bit* b do registo f é igual a um, então salta a próxima instrução.

Tabela 3.3. Instruções sobre o ficheiro de registos orientadas ao *bit*.

3.3 Características do Ficheiro de Registos

O ficheiro de registos do PIC16C54 é constituído por registos especiais e por registos de propósito geral (f7 a f31), como se pode observar pela Figura 3.5. O dispositivo PIC16C55 contém mais um porto de E/S (porto C) no endereço 7 do ficheiro de registos. O dispositivo PIC16C56 tem mais um *bit* no PC que lhe permite endereçar 1024 instruções de programa [2].

Qualquer das instruções que endereçam um registo do ficheiro não tem esse endereçamento restrito a apenas uma gama de registos.

Na família de PICs existem dispositivos com mais do que os 32 registos de 8 *bits* descritos anteriormente, como é o exemplo do PIC16C57. O esquema de endereçamento de registos, que ultrapassam a gama de indexação possível pelos 5 *bits*, é baseado no mapeamento de memória ilustrado na Figura 3.6. Cada banco extra é constituído por 16 registos e é mapeado pelos *bits* 5 e 6 do registo FSR. O primeiro banco contém os registos f16:31, o segundo f32:47, o terceiro f48:63, e o quarto f64:79. Este elemento da família PIC tem, como se pode ver pela Figura 3.6, mais 2 *bits* no registo PC que lhe permitem endereçar 2048 instruções de programa.

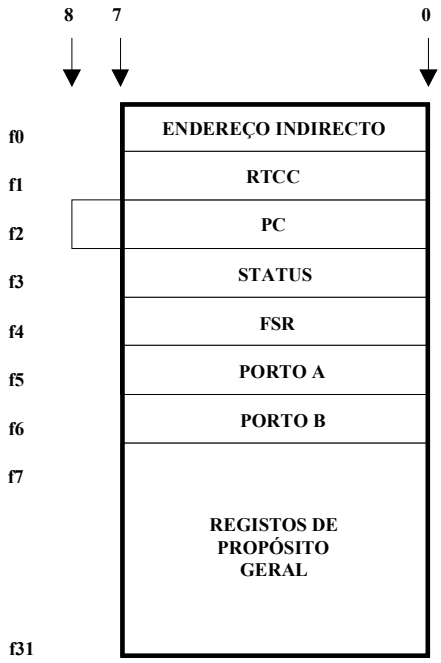


Figura 3.5. Ficheiro de registos.

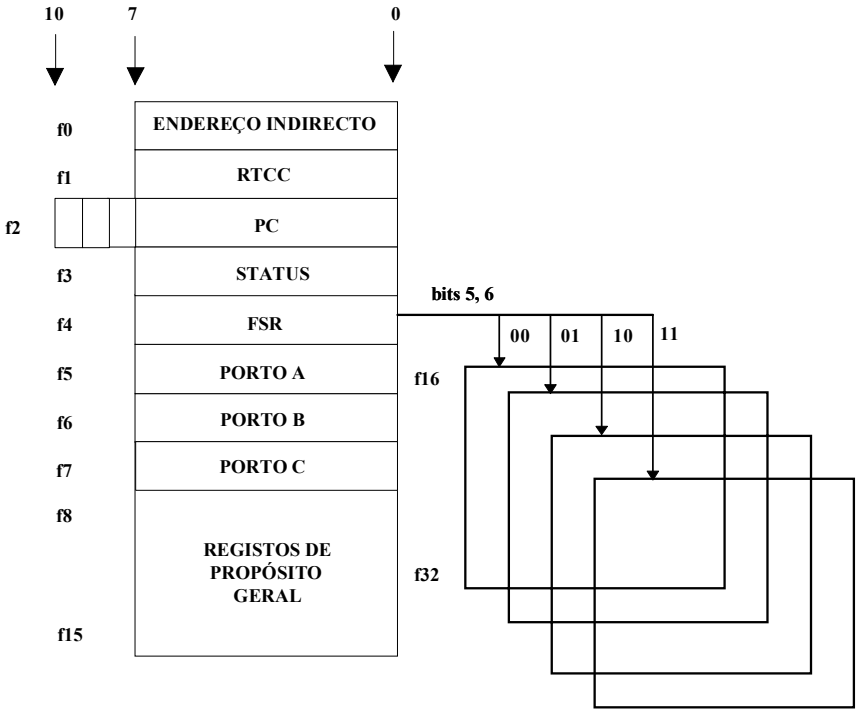


Figura 3.6. Mapeamento de registos.

3.3.1 Registo f0 (endereçamento de dados indirecto)

Este registo não existe fisicamente. Uma instrução que utilize este registo indica que o registo endereçado é o indicado pelos 5 bms do registo f4 (FSR[7]). Caso o endereço do registo a utilizar seja o próprio f0, na leitura é lido zero e a escrita corresponde à execução de um **NOP** (não é efectuada qualquer operação).

3.3.2 Registo f1 (contador/relógio de tempo-real)

Este registo, designado por RTCC[8], pode ser lido ou escrito como qualquer registo genérico. Pode também ser incrementado por um sinal externo conectado ao pino RTCC, ou pelo relógio interno, CLKOUT.

O diagrama de blocos da Figura 3.7 descreve o funcionamento da atribuição do pré-escalar, mediante a programação por *software* do registo OPTION. Ao contador/relógio em tempo-real pode ser atribuído, pelo *bit* PSA, um pré-escalar de 8 *bits*, definido pelos *bits* PS2-0 do registo OPTION. Se o pré-escalar for atribuído, as instruções que escrevem no registo f1 (**CLRF 1**, **BSF 1, 5**, etc) colocam a zero o pré-escalar.

O *bit* RTS do registo OPTION determina se f1 é incrementado por um sinal interno ou externamente. No caso de escolha do sinal externo para incremento, é possível seleccionar, através do *bit* RTE do registo OPTION, o flanco de sinal que despoleta o incremento. O registo f1 é incrementado até atingir o valor 0xFF e reinicia novamente em 0x00.

---

[7] Do Inglês ***F**ile **S**elect **R**egister*.

[8] Do Inglês ***R**eal **T**ime **C**lock/**C**ounter*. Em Português contador/relógio em tempo-real.

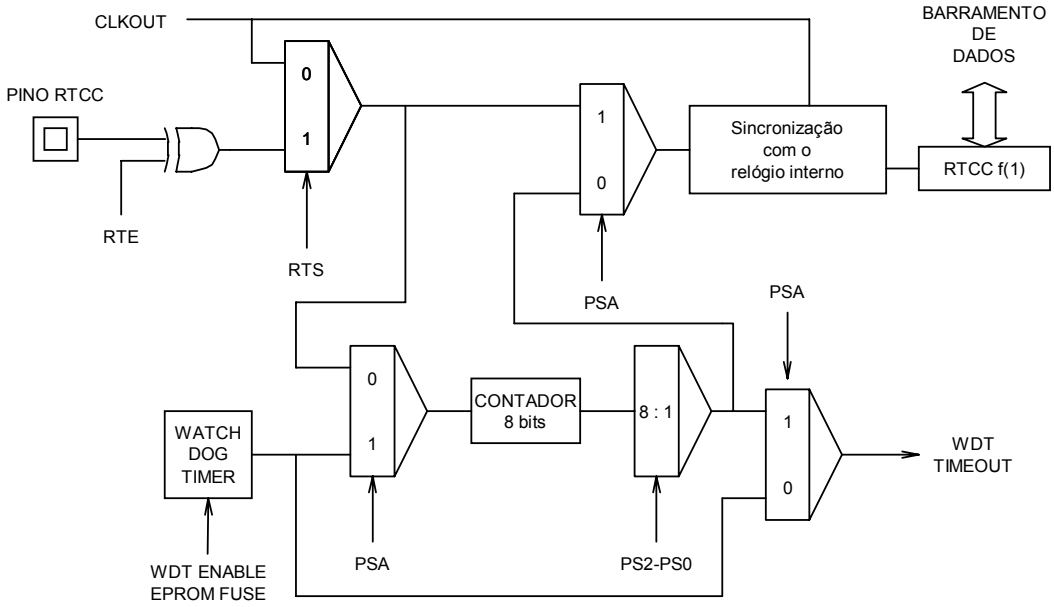


Figura 3.7. Diagrama de blocos do circuito que faz a atribuição do pré-escalar ao RTCC/WDT.

3.3.3 Registo f2 - PC

Este registo permite aceder até 512 instruções (9 *bits* disponíveis no PIC 16C54) do código do programa armazenado na memória de programa. Durante a fase de execução de programa o PC é incrementado de uma unidade por cada instrução a não ser que o resultado da instrução modifique o próprio PC, ou em presença de instruções de salto:

1. A instrução **GOTO** modifica os 9 *bits* do PC, com os 9 *bits* do endereço especificado pela instrução.

2. A instrução **CALL** coloca a zero o nono *bit* do PC e carrega directamente os restantes com o endereço especificado pela instrução. É colocado na pilha o valor de PC+1.

3. A instrução **RETLW** carrega o PC com o valor de topo da pilha.

4. Quando a instrução endereça o PC como registo destino (ex. **MOVWF 2**) os 8 *bits* são carregados com o conteúdo do registo W, o nono *bit* é colocado a zero, e é executado um **NOP** no ciclo de instrução imediatamente a seguir.

3.3.4 Registo de estado (f3)

Este registo, representa o registo de estado (SR[9]), e contém as *flags* de controlo da execução de determinadas operações. Os *bits* TO e PD são apenas de leitura e servem para detectar se o microcontrolador foi ligado, acordou do modo SLEEP pelo *timeout* do WDT, acordou pela aplicação do *reset* externo (sinal lógico zero no pino $\overline{\text{MCLR}}$), ou ocorreu o *timeout* do WDT. A instrução **CLRF 3**, coloca a zero todos os *bits* deste registo, com excepção dos dois anteriores e do *bit* Z que neste caso é colocado a '1'. Recomenda-se, caso se pretenda colocar a zero este registo, a utilização de instruções cujo resultado não afecte o registo de *status* (**BCF**, **BSF** e **MOVWF**).

	bit	Descrição da funcionalidade
0	C	*CARRY/BORROW BIT* Activo se houve transporte no bMs (*bit* mais significativo) do resultado da operação. Usado pelas instruções **ADDWF**, **SUBWF**, **RRF** e **RLF**.
1	DC	*DIGIT CARRY/BORROW BIT* Activo se houve transporte dos 4 bms do resultado da operação. Usado pelas instruções **ADDWF** e **SUBWF**.
2	Z	*ZERO BIT* Activo se o resultado de uma operação aritmética ou lógica é zero, se o conteúdo do registo transferido é zero, ou quando é executada a instrução **CLRF 3**.
3	PD	*POWER DOWN BIT* É activado, durante a ligação (*power up*) ou pela instrução **CLRWDT**. É desactivado pela operação SLEEP.
4	TO	*TIME-OUT BIT* É activado, durante a ligação (*power up*) ou pelas instruções **SLEEP** e **CLRWDT**. É desactivado pelo tempo final do WDT.
5	PA0	*Bit* de leitura/escrita para uso geral. (PIC16C54/55)
6	PA1	*Bit* de leitura/escrita para uso geral. (PIC16C54/55)
7	PA2	*Bit* de leitura/escrita para uso geral (reservado para uso futuro)

Tabela 3.4. Descrição de cada *flag* do registo de estado.

3.3.5 Registo de selecção indirecta (f4)

Este registo permite o endereçamento indirecto de um registo. Os 5 bms seleccionam um dos 32 registos do ficheiro de registos no modo de endereçamento indirecto

[9] Do Inglês ***S**tatus **R**egister*.

(endereçando o registo f0 em qualquer instrução que manipule registos). Os 3 bMs têm o valor lógico 1.

Caso não seja usado o modo de endereçamento indirecto este registo pode ser usado como um registo de propósito geral com 5 *bits*.

3.3.6 Registos de E/S

O registo f5, denominado de PORTO A tem apenas 4 *bits* que acedem a 4 pinos de E/S. Apenas são usados os 4 bms (RA0 - RA3). Os 4 bMs são lidos como zero. O registo f6, denominado de PORTO B, tem 8 *bits* que acedem a 8 pinos de E/S.

Uma instrução de leitura lê sempre o valor lógico do pino, quer este esteja programado como saída ou como entrada.

3.4 Registos específicos

3.4.1 Registos TRISA e TRISB

Estes registos configuram o estado dos pinos de E/S. Caso algum dos *bits* de TRISA ou TRISB esteja no valor lógico ‘1’, implica que o *bit* correspondente no pino está em alta impedância (só para leitura). Assim, cada pino é configurado pelo *bit* correspondente do registo TRIS. Estes registos são carregados com o conteúdo do registo W, pela instrução **TRIS**. Na inicialização do micro estes registos são colocados a ‘1’ para que os portos de E/S estejam em alta impedância.

3.4.2 Registo OPTION

O registo OPTION é constituído pelos *bits* apresentados na Tabela 3.5. Este registo é utilizado para configurar o pré-escalar do temporizador do WDT ou o contador/relógio em tempo-real. O conteúdo deste registo é programado por *software* com o uso da instrução **OPTION**, que carrega os 6 *bits* de OPTION com o conteúdo dos 6 bms do registo W. Em caso de condição de RESET os *bits* são colocados a '1'.

	bit	Descrição da funcionalidade
0	PS0	{PS2, PS1, PS0} valor do pré-escalar.
1	PS1	
2	PS2	
3	PSA	*bit* de atribuição do pré-escalar (atribuição mutualmente exclusiva): (1) '0' RTCC (2) '1' WDT
4	RTE	indica o flanco de sinal utilizado para o incremento do RTCC: (1) '0' incrementa na transição ascendente do pino RTCC. (2) '1' incrementa na transição descendente do pino RTCC.
5	RTS	indica a origem do sinal de incremento: (1) '0' relógio interno de instrução (CLKOUT) (2) '1' sinal no pino RTCC

Tabela 3.5. Descrição de cada *bit* do registo OPTION.

Os estados possíveis do pré-escalar, formado pelos *bits* {PS2, PS1, PS0}, e o período do ciclo de relógio, em relação ao ciclo de instrução estão representados na Tabela 3.6.

PS2	PS1	PS0	RTCC	WDT
0	0	0	1 : 2	1 : 1
0	0	1	1 : 4	1 : 2
0	1	0	1 : 8	1 : 4
0	1	1	1 : 16	1 : 8
1	0	0	1 : 32	1 : 16
1	0	1	1 : 64	1 : 32
1	1	0	1 : 128	1 : 64
1	1	1	1 : 256	1 : 128

Tabela 3.6. Pré-escalares possíveis para o temporizador do "cão de guarda" e para o contador/relógio em tempo-real.

3.4.3 Registo W

Registo utilizado pelas instruções como segundo operando ou de suporte de transferências internas de dados e equivalente ao acumulador de algumas arquitecturas. Todas as intruções lógicas, aritméticas e de movimento de dados usam este registo. As instruções que acabam em "WF", usam, como um dos operandos de entrada o valor de W, e colocam o resultado em W ou no registo que identificou o segundo operando de entrada (Exemplo 3.1).

```
ADDWF 6, W  ; W = W + F6
ADDWF 6     ; F6 = W + F6
```

Exemplo 3.1. Utilização de W como segundo operando.

3.5 Características gerais

Nesta secção complementa-se a descrição do microcontrolador com a explicação de alguns tópicos que necessitam de clarificação.

3.5.1 Ciclos por Instrução

Cada instrução executa num único ciclo (sinal CLKOUT no pino OSC2) que equivale a 4 ciclos do relógio externo (pino OSC1) à excepção das instruções de salto e instruções que mudem o valor do PC. Estas últimas demoram 8 ciclos de OSC1 (2 ciclos de CLKOUT).

3.5.2 Imediatos

O microcontrolador possui instruções equivalentes às instruções com endereçamentos de valores imediatos da maioria dos processadores. Estas instruções, que envolvem imediatos, com mnemónica acabada em "**LW**", permitem a manipulação de constantes codificadas nos 8 bms da própria instrução.

3.5.3 *Reset* externo

O pino $\overline{\text{MCLR}}$ quando colocado a zero coloca o microcontrolador em "*reset* externo", que corresponde à colocação em f2 (PC) do vector de *reset* (0x1FF no PIC16C54), colocação de 0xFF no registo OPTION, registos TRIS (A e B) a 0xFF (no caso de A, como só inclui os 4 *bits* menos significativos, é colocado 0x0F), o temporizador do WDT e o pré-escalar a zero, e os 3 bMs de f3 a zero.

3.5.4 *Reset* interno

O *reset* interno, não faz a inicialização descrita para o *reset* externo, e ocorre quando acontece uma das situações apresentadas na Tabela 3.7.

TO	PD	*RESET* causado por:
0	0	WDT *timeout* durante o SLEEP
0	1	WDT *timeout* (sem ser durante o SLEEP)
1	0	*RESET* externo (pino MCLR) durante o SLEEP
1	1	Ligação do microcontrolador
X	X	*RESET* externo (pino MCLR)

Tabela 3.7. Estados dos *bits* TO e PD após *reset*.

3.5.5 Chamadas a rotinas

Quando é executada uma instrução **CALL**, o topo da pilha é carregado com o valor de PC+1, e ao PC é dado o valor do endereço da primeira instrução da rotina chamada. O retorno da rotina é feito pela instrução **RETLW**, que devolve ao PC os 9 *bits* (PIC15C54) guardados no topo da pilha. O corpo da rotina tem que estar no bloco de memória constituído pelas primeiras 256 instruções (acedidas pelos 8 bMs da instrução **CALL**). A chamada de uma segunda rotina durante a execução da primeira (rotinas encadeadas), provoca a passagem do topo da pilha para baixo, e mais tarde, quando do retorno desta segunda rotina, novamente para o topo. São apenas permitidos dois níveis de encadeamento de rotinas, embora haja a possibilidade de contornar este problema com o uso da instrução **GOTO**. Ao fazer uma terceira chamada, o topo da pilha é substituído pelo PC+1 e o retorno da segunda chamada passa para o nível inferior da pilha. Nesta situação os retornos da 2ª e 3ª são correctos mas é perdido o retorno da 1ª. A família de PICs incluí a gama 16Cxx [2] que contém uma pilha de 8 níveis que funciona em modo circular.

A pilha não pode ser acedida por mais nenhuma instrução e não existe qualquer instrução que permita examinar o conteúdo.

3.5.6 Tabelas de constantes

As tabelas de constantes (*lookup tables*) são implementadas através da modificação do PC pelo *software* (ver Exemplo 3.2). Não existe outra forma de aceder a uma posição específica da memória do programa.

```
tabela      andlw 0x03  ; Limita a tabela a 4 valores
            addwf PC
            retlw 0x01
            retlw 0x02
            retlw 0x03
            retlw 0x04
```

Exemplo 3.2. Uso da instrução RETLW para a implementação de tabelas de valores:

Neste exemplo é retornado para o registo W o valor indexado pelos três bms deste registo. A indexação de valores deve ser feita convenientemente, pois uma indexação fora dos limites das instruções **RETLW** provoca a execução de instruções arbitrárias.

3.5.7 Cão de guarda (WDT)

É um contador especial, que se pode programar para reiniciar uma aplicação quando esta descarrila ou para acordar o microcontrolador de um adormecimento. O WDT executa com um período típico de 18ms e é independente do relógio externo. Mesmo que o microcontrolador esteja no modo SLEEP o temporizador está activo, e quando atinge o *timeout* produz o *reset* interno do integrado. O período de *timeout* pode ser programado pela atribuição do pré-escalar ao WDT colocando PSA a '1' no registo OPTION. No máximo é possível obter *timeouts* de aproximadamente 2,3s (18ms x 128). As instruções **SLEEP** e **CLRWDT** colocam a zero o WDT e o pré-escalar, se estiver atribuído ao WDT, não permitindo o *timeout* e consequente *reset* do integrado. O *timeout* do WDT coloca a zero a *flag* TO do registo de estado. O WDT pode ser desactivado programando o fusível de configuração a '0'.

3.5.8 O modo SLEEP

No modo de adormecimento, provocado pela instrução **SLEEP**, o relógio do microcontrolador é desligado de parte do integrado, mantendo o temporizador do WDT e o contador/relógio em tempo-real a funcionar. A *flag* TO é colocada a um, PD colocada a zero, e é feito o *reset* do WDT. O integrado acorda deste estado ou pelo RESET externo (pino $\overline{\text{MCLR}}$ a zero) ou pelo *timeout* do WDT se estiver activado o fusível de configuração. Ao acordar, o microcontrolador executa a instrução contida no vector de *reset* (0x1FF). Os microcontroladores 16C84 e 16C71 [2], quando acordam do modo SLEEP executam a instrução imediatamente a seguir (em PC+1).

Os pinos de E/S mantêm o estado anterior à execução da instrução **SLEEP**.

3.6 Suporte *Software*

Para esta família de microcontroladores existe um vasto suporte de *software* para PC, desde assembladores cruzados[10], simuladores, compiladores de C, etc.

Da Microchip™ existe um pacote constituído pelo MPALC [48] (*assemblador*) e MPSIM [49] (simulador). O MPALC gera o código objecto no formato INHX8M [49] (*Merged 8-bit Intellec Hex Format*). Além do conjunto de instruções intrínsecas, o assemblador permite a utilização de 27 mnemónicas extra [48] (pseudo-operações) constituídas por uma ou mais instruções intrínsecas e que por vezes permitem maior clarividência de programação. Para o programador aceder-lhes só necessita de colocar no ficheiro de programa a directiva: `include "Pseudoin.mac"`.

O MPSIM simula por discretização de eventos. A simulação pode ser feita passo-a-passo, ou estabelecendo pontos de paragem, que facilitam a depuração (*debug*) do código. Após a execução de cada instrução pode ser visualizado o conteúdo dos registos internos do microcontrolador nesse instante.

O simulador permite a edição de um ficheiro de estímulos (vectores de teste a aplicar aos pinos do microcontrolador).

A Parallax™ também apresenta vários produtos entre os quais um assemblador cruzado (PASM) e um simulador (PSIM) ilustrado na Figura 3.8. Estas ferramentas suportam as mnemónicas da Microchip™ ou pseudo-instruções. Esta redefinição da sintaxe de cada instrução permite instruções ou macro-instruções que, respectivamente, correspondem a múltiplas instruções ou a algumas instruções da Microchip™, tornando menor o conjunto de instruções (ex. a Parallax™ empacota 10 instruções em uma instrução genérica **MOV**).

---

[10] Em Inglês *cross-assemblers.*

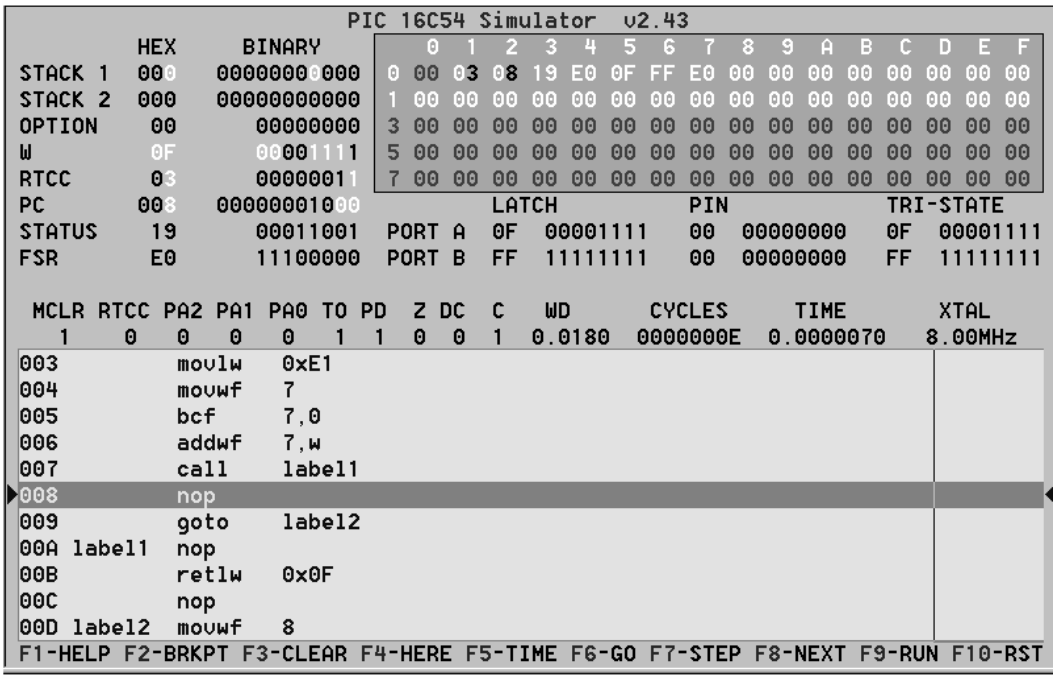


Figura 3.8. Simulador PSIM da Parallax™.

Existem vários compiladores de C para a família PIC. Estes permitem o desenvolvimento de *software* mais rapidamente, por se tratar de uma linguagem de alto nível. Ao C são adicionadas construções que permitem o acesso às portas de entrada/saída, manipulação de dados do tipo *bit*, configuração dos temporizadores, etc., cruciais na maioria das aplicações.

3.7 Conclusões

Para arquitectura alvo do sistema de co-síntese era necessário desenvolver uma unidade central de processamento do CI que permitisse executar o *software*. Como principais requisitos salientava-se a necessidade de ser um processador de dimensão reduzida, com um conjunto de instruções compacto, ortogonal e reduzido (unidade de controlo simples e de dimensão reduzida). Era também necessário que este micro integrasse a memória de programa, possibilitando o interface rápido entre esta e o processador.

Conforme descrito neste capítulo, os microcontroladores PIC são um óptimo exemplo de um microprocessador com as características anteriores, o código é extremamente compacto, as aplicações são inúmeras e nos últimos anos tem tido taxas de

popularidade elevadas, a que não são alheios os vários artigos de aplicações com este integrado em muitas revistas da especialidade.

Neste sentido, optou-se pelo desenvolvimento de um núcleo de processamento com conjunto de instruções compatível com este microcontrolador. Ao longo do capítulo seguinte é descrito o projecto desta unidade central, cuja arquitectura permite a parametrização automática de diversas unidades internas.

4. Projecto do Processador

"Never waste time".

American Proverb

No presente capítulo são descritos o projecto e a implementação do processador que constitui a unidade central (núcleo) do circuito integrado. A metodologia de projecto utilizada assenta na descrição do processador num subconjunto de VHDL sintetizável [50], [51], [52], seguida da síntese do circuito digital em células da biblioteca IMS [53] e posterior mapeamento no SOG. A verificação do processador foi realizada ao nível RTL, através de simulações funcionais, e ao nível lógico, através de simulações lógicas exaustivas, dos diferentes módulos individualmente, em agrupamentos de funcionalidade dependente, e, por último, do CI completo. O projecto requereu várias iterações que permitiram melhoramentos, sendo aqui apenas descrita a versão final, embora sejam referidas versões anteriores sempre que sejam necessárias comparações. No final do capítulo é apresentada uma versão de teste fabricada numa bolacha GF4 [21].

4.1 Arquitectura

O processador realizado é baseado na arquitectura descrita no capítulo anterior, possui um conjunto de instruções compatível com os elementos da família PIC16C5X [2] e foi designado por PARMIC. A arquitectura simplificada encontra-se ilustrada na Figura 4.1 e apresenta algumas diferenças significativas em relação ao PIC que

permitiram aumentar o desempenho do processador e tornar possível a parametrização automática conforme a aplicação a que se destina.

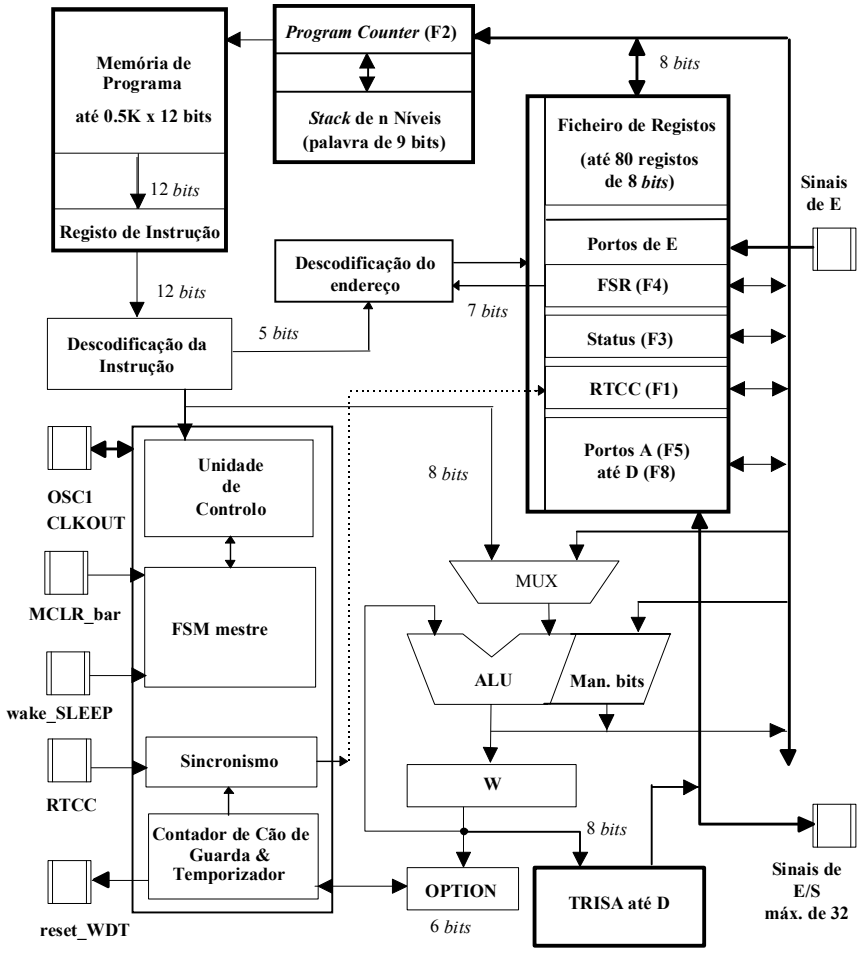


Figura 4.1. Diagrama de blocos do processador.

As melhorias mais notórias são:

- Um ciclo de instrução (CLKOUT) equivale a 3 ciclos de relógio (OSC1) em vez dos 4 ciclos de relógio do PIC.

- Foi adicionado *pipelining* ao controlo, que permite ter os sinais de controlo, para o ciclo actual, disponíveis logo no início do ciclo de relógio.

- As instruções **GOTO**, **CALL**, e **RETLW**, são executadas em apenas um ciclo de instrução (3 ciclos de relógio) contrariamente aos dois ciclos de execução do PIC (8 ciclos de relógio).

- O número de portos de E/S é parametrizável, podendo co-existir até 4 portos de E/S (32 pinos de E/S). O porto de E/S A foi implementado por defeito com 8 *bits* (4 *bits* no PIC). Dadas as facilidades de parametrização qualquer dos portos é reconfigurável, permitindo a redução do números de pinos de E/S do CI em aplicações que o exijam. Qualquer registo do ficheiro de registos pode ser um porto de saída.

- Estão disponíveis portos apenas de entrada. O número de portos de entrada é parametrizável, em que o número máximo é limitado pelo número de pinos de sinal disponíveis pela bolacha e/ou encapsulamento.

- O número de registos do ficheiro de registos é parametrizável (a solução actual tem no máximo 80 registos de 8 *bits*). No caso de exceder os 32 registos do PIC16C54, é utilizado o mapeamento de memória de dados utilizado em outras versões da família PIC. O número de registos pode ser ainda maior (144 registos, ao serem utilizados para mapear os 3 bMs do registo SR), embora faça pouco sentido números superiores, devido ao limite máximo actual da memória de programa.

- A dimensão da memória de programa é parametrizável, conforme o número de instruções do programa implementado. Esta pode armazenar um máximo de 512 instruções. O aumento da memória de programa é possível, aumentando o número de *bits* do PC e das instruções **GOTO** e **CALL**, de modo a poderem permitir o endereçamento de instruções de uma memória de programa com mais do que 512 instruções, ou com a utilização do mapeamento de memória de programa presente em alguns elementos da família PIC (máximo de 2048 palavras com instruções de 12 *bits*).

- O número de rotinas encadeadas não está limitado, pois a dimensão da pilha é também parametrizável de acordo com o nível de encadeamento do programa origem.

A entidade VHDL que descreve o processador completo é formada pela instanciação estrutural das diversas entidades que o constituem. Algumas destas são também

especificadas hierarquicamente conforme o grau de complexidade. O estilo de descrição estrutural, encontra-se por vezes encapsulado com outros estilos de descrição na mesma entidade. Para facilitar a descrição, e torná-la mais compreensível com um nível de abstracção mais elevado, foi criado um módulo VHDL que inclui a redefinição de operadores, funções e procedimentos comummente utilizados, tais como: adição, subtracção, incremento, decremento, conversão de inteiro para binário e vice-versa, etc. Este módulo pode ser consultado no apêndice C.

4.2 Funcionamento Geral

Os passos de execução de uma instrução genérica, representados na Figura 4.2, são: procura da instrução (IF[1]), novo contador de programa (NPC[2]), descodificação da instrução (ID[3]), leitura do operando (RO[4]), execução da operação (EX[5]), escrita do operando (WO[6]), e escrita no registo de estado (WS[7]). A procura e a descodificação da próxima instrução são realizadas em paralelo com a execução da instrução actual.

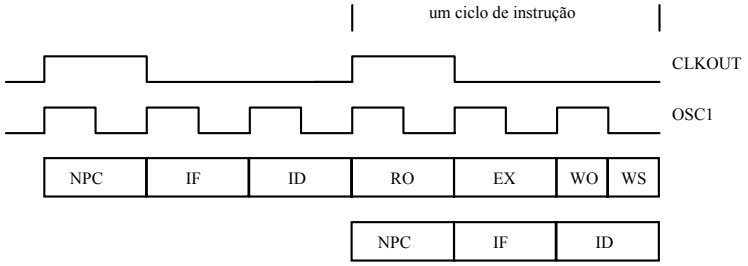


Figura 4.2. Diagrama temporal da execução de uma instrução.

Os saltos condicionais são executados em dois ciclos de instrução (6 períodos do OSC1) sempre que a condição de salto for satisfeita. A razão deste prolongamento deve-se à necessidade de conhecimento do resultado da ALU ou da unidade de

---

[1] Do Inglês ***I**nstruction **F**etch.*

[2] Do Inglês ***N**ext **P**rogram **C**ounter.*

[3] Do Inglês ***I**nstruction **D**ecoder.*

[4] Do Inglês ***R**ead **O**perand.*

[5] Do Inglês *instruction **EX**ecution.*

[6] Do Inglês ***W**rite **O**perand.*

[7] Do Inglês ***W**rite **S**tauts register.*

manipulação de *bits*, apenas disponível no início do terceiro período do primeiro ciclo de instrução, para actualizar o PC da próxima instrução. No segundo ciclo de instrução é capturada e descodificada a próxima instrução a executar, e é executado um **NOP** (***No Op****eration*) em paralelo.

Quando o operando destino é o PC, a execução da instrução é também estendida para dois ciclos, em que no segundo ciclo de instrução é capturada e descodificada a instrução endereçada pelo novo valor do PC e executado um **NOP** em paralelo.

O período NPC é utilizado pela instrução **CALL** - para armazenar o endereço de retorno na pilha - e para determinação do valor do novo contador de programa. O período IF começa no início do segundo ciclo. No início deste ciclo o contador de programa para o próximo ciclo de instrução é carregado no registo PC e a próxima instrução é capturada da memória de programa até ao fim deste ciclo.

O *pipelining* de controlo foi realizado inserindo registos entre a unidade de controlo e o *datapath*. Alguns registos são sensíveis ao flanco ascendente, outros ao flanco contrário. Para a correcta execução de cada instrução estes sinais devem ser gerados no ciclo anterior ao ciclo em que são necessários e por esse motivo o processador começa com um ciclo do OSC1 antes de começar a execução da primeira instrução do programa.

4.3 Unidade de Controlo completa

Na Figura 4.3 é representada a estrutura da unidade de controlo completa, que contém a máquina sequencial mestre, uma sub-unidade de controlo, e os registos de *pipelining*. A unidade de controlo interna é comandada pela máquina mestre (com os sinais de interface ilustrados na Figura 4.4) constituída por três estados e habilitada, para a possível transição de estado, no flanco ascendente do relógio externo (OSC1). O diagrama de transição de estados correspondente é ilustrado na Figura 4.5.

O pino de sinal, “wake_SLEEP”, quando activado, corresponde ao “acordar” do processador por um sinal externo (colocação de “MCLR_bar” a zero durante o modo

SLEEP no PIC). Este pino extra foi colocado para diferenciar o *reset* puro do processador do acordar por *reset* externo.

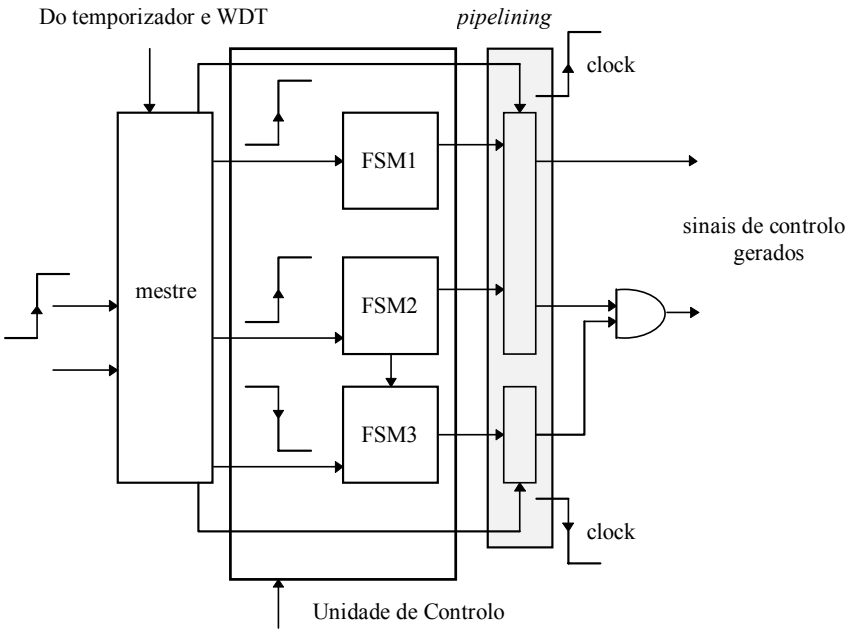


Figura 4.3. Diagrama de blocos da unidade de controlo completa.

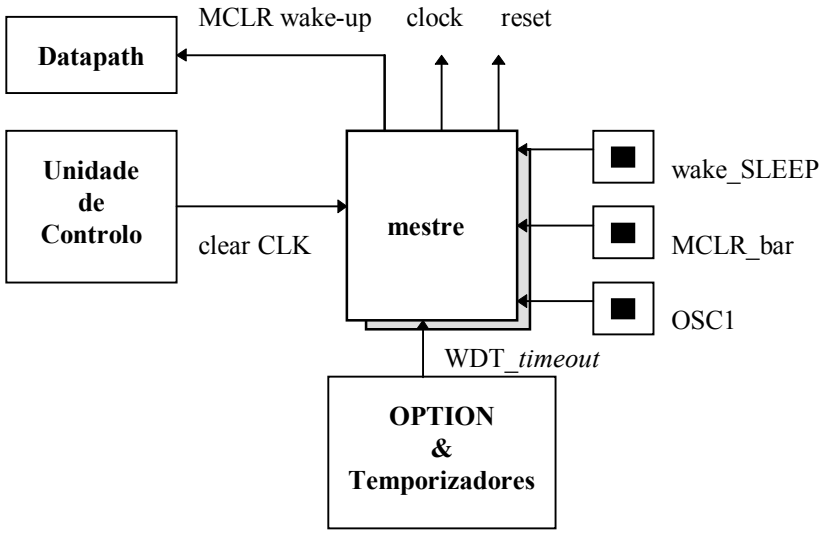


Figura 4.4. Máquina de controlo mestre e a ligação à unidade de controlo geral e ao *datapath*.

Os 3 estados da máquina mestre são: RESET_STATE, SLEEP_STATE, e ACTIVE_STATE. O primeiro estado é o estado do tradicional *reset* para inicialização do micro. Quando o micro é ligado (*POWER UP*) é colocado neste estado, até que o pino assíncrono "MCLR_bar" seja colocado a '1' e posteriormente ocorra uma

transição ascendente no pino OSC1. Neste estado o relógio interno encontra-se desactivado e é feita a inicialização das unidades.

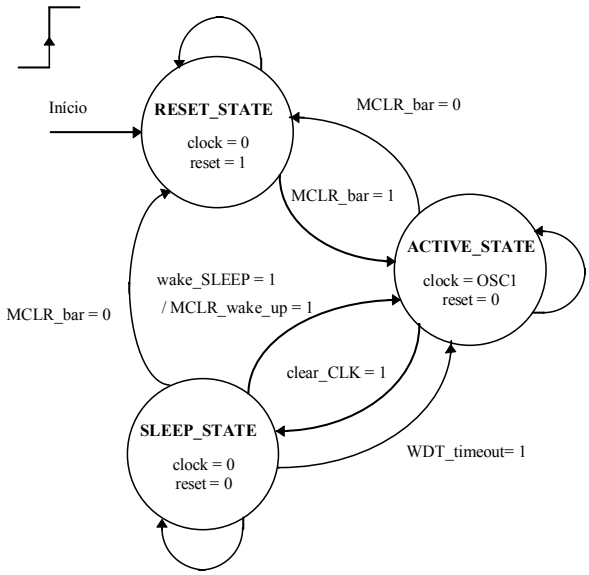


Figura 4.5. Diagrama de transição de estados da máquina mestre.

O estado ACTIVE_STATE representa o estado de execução normal do programa. Neste estado, o relógio interno toma o valor lógico do sinal OSC1 e é desactivado o *reset* interno. A máquina escapa deste estado quando é executada a instrução SLEEP (e a máquina é colocada no estado SLEEP_STATE) ou quando o pino "MCLR_bar" (*reset* externo) é colocado a zero (e a máquina é colocada no estado RESET_STATE).

Quando a máquina é colocada no modo de adormecimento (SLEEP_STATE) o relógio interno é desligado. O micro retorna ao primeiro estado quando é feito um *reset* externo, e ao terceiro estado quando o WDT finaliza a contagem e causa o acordar do estado SLEEP_STATE (por *timeout*) ou quando o pino externo "wake_SLEEP" é colocado ao nível lógico '1' (acordar por um sinal externo). O modo como a máquina retorna de um estado é assinalado nas *flags* TO e PD do registo SR conforme foi descrito no capítulo 3.

A unidade de controlo é constituída por três máquinas de estado: FSM1, FSM2, e FSM3. Estas máquinas são activadas quando a máquina mestre se encontra no estado ACTIVE_STATE.

A máquina FSM2 é constituída por 30 estados e é responsável pela geração de 27 sinais de controlo que guiam a execução da maioria das instruções. Esta máquina de estados gera um sinal que habilita o funcionamento da máquina FSM3 responsável pelos 5 sinais, que podem transitar em ambos os flancos, e é constituída por 4 estados. Os sinais de controlo vindos de cada uma das máquinas são combinados pelo AND lógico (ver Figura 4.6).

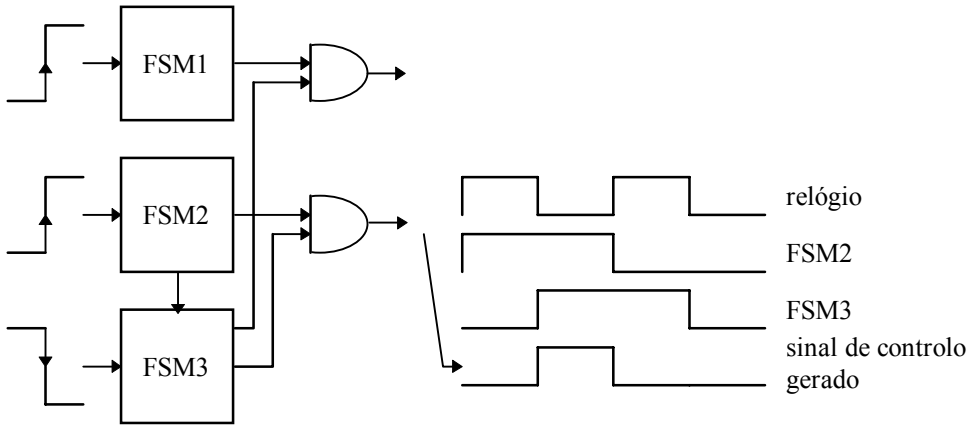


Figura 4.6. Geração de sinais de controlo com variações em ambos os flancos do sinal de relógio.

Cada máquina de estados inclui sempre um estado de inicialização. No primeiro estado de execução é identificada a instrução actual, definido o caminho de transição de estados ao longo do ciclo de instrução, e originados os níveis lógicos dos sinais de controlo necessários.

4.3.1 Controlo da procura de uma instrução

A máquina de estados (FSM1) controla a procura de uma instrução, o incremento do PC, e a execução de instruções que necessitem de mais do que um ciclo de instrução. Os sinais de interface são apresentados na Figura 4.7 e o diagrama de transição de estados na Figura 4.8. A Tabela 4.1 descreve os sinais de controlo gerados.

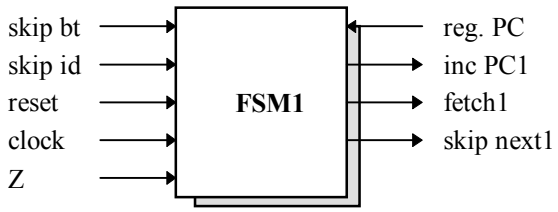


Figura 4.7. Sinais de interface da máquina de controlo da procura de uma instrução (FSM1).

Sinais de Controlo	Designação	Descrição funcional
Load IR	fetch	carrega o registo de instrução IR.
Inc. PC	inc_pc	carrega o PC.
skip next instr.	skip_next	salta a próxima instrução.

Tabela 4.1. Descrição dos sinais de controlo da máquina que controla a procura da instrução.

Na Figura 4.9 é ilustrado o diagrama temporal do funcionamento desta máquina de estados em conjunto com a máquina FSM3. No caso da instrução em execução não escrever no PC (caso A1), é gerado um pulso no sinal "inc_pc" com início no segundo ciclo de relógio. O flanco ascendente deste sinal é responsável pela colocação no registo PC do endereço da próxima instrução (determinado anteriormente), que em execução normal será o valor de PC+1.

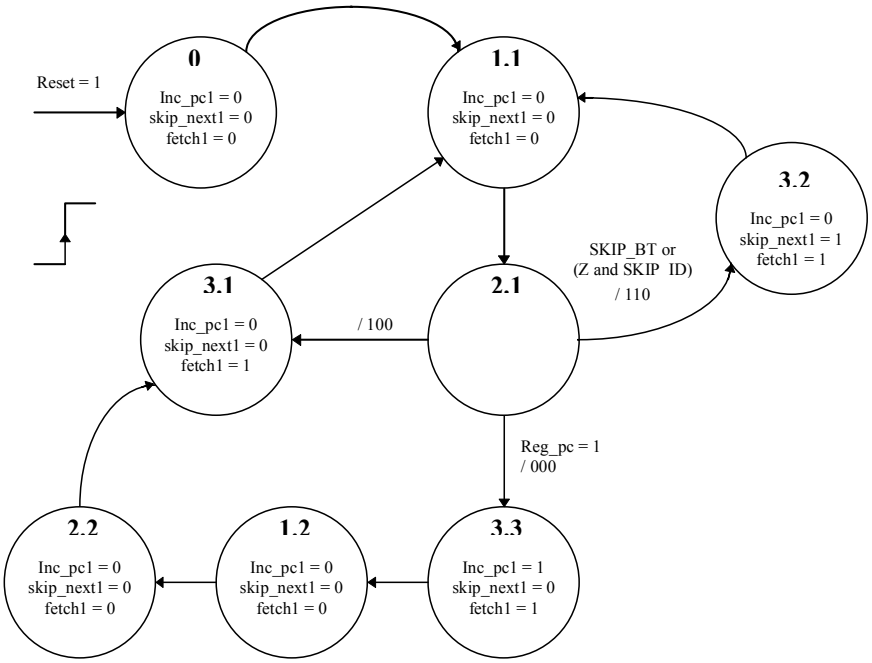


Figura 4.8. Diagrama de transição de estados da FSM1 responsável pelas instruções de saltos condicionais, pelo funcionamento normal de incremento do PC e procura da instrução.

Quando o PC é o registo destino (caso A2), a transição ascendente do sinal "inc_pc" é gerada apenas na segunda metade do terceiro ciclo de relógio (ciclo em que o resultado da ALU já se encontra disponível), o que possibilita o armazenamento do operando resultado no registo PC.

Um pulso gerado no sinal "skip_next" (caso C) indica uma condição verdadeira em uma instrução de salto condicional e origina que a próxima instrução seja um **NOP**.

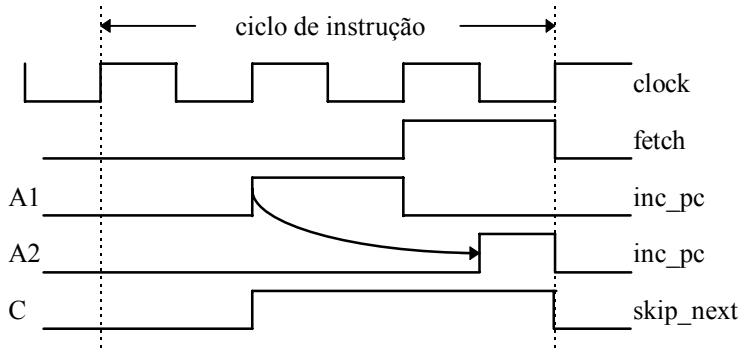


Figura 4.9. Diagramas temporais que ilustram o funcionamento da FSM1 em conjunto com a FSM3.

Um pulso no sinal "fetch" é responsável pelo armazenamento da próxima instrução no registo de instrução IR[8] (Figura 4.10). A instrução armazenada em IR permanece disponível no ciclo de relógio anterior ao início do ciclo de instrução até ao segundo ciclo de relógio (totalizando os três ciclos de relógio por cada ciclo de instrução). No ciclo anterior é realizada a descodificação e a geração dos sinais de controlo, embora só fiquem disponíveis no início do ciclo seguinte. Desta forma o controlo é gerado sempre com um ciclo de avanço relativamente a cada ciclo de relógio durante o ciclo de instrução.

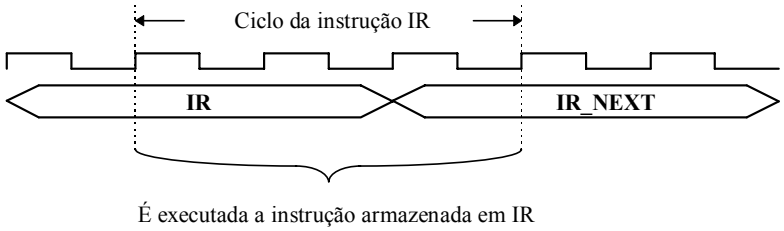


Figura 4.10. Diagramas temporais que ilustram o carregamento da instrução.

Como, na maioria dos casos, a sequência de estados de cada FSM é garantida logo no início do ciclo de instrução, o facto do registo IR mudar de instrução (é carregado com a próxima instrução) ainda antes do fim do ciclo de instrução não interfere no normal funcionamento da instrução em execução.

[8] Do Inglês ***I**nstruction **R**egister*.

4.3.2 Unidade completa

Na Tabela 4.2 são descritos os restantes sinais de controlo gerados pela unidade de controlo completa.

Sinais de Controlo	Designação	Descrição funcional
load W	d	carrega o registo W
clear W	clr	coloca a zero o registo W
enable Res. ALU	disable	coloca o resultado da ALU no barramento
enable W	enable_W	coloca o valor de W no barramento
write	write	escreve o conteúdo do barramento Sin num registo do FR (Ficheiro de Registos)
write status	write_stat	escreve os *bits* de *status* no registo SR
enable	enable	coloca o conteúdo de um registo do FR no barramento Sout
clear F	clear_F	coloca no barramento o valor zero
load B	ld_B	carrega o registo B
sel B	sel_operandoB	selecciona entre o valor imediato definido na instrução ou a palavra do barramento Sout
clear CLK	clear_CLK	coloca o processador no modo de adormecimento.
clear WDT	clear_WDT	coloca a zero o pré-escalar.
clear bit	clear_bit	coloca o *bit* especificado no nível lógico zero.
Disable B1	disable_B	coloca o resultado da unidade de manipulação de *bits* no barramento Sin
Inc. RTCC	inc_rtcc	incrementa o RTCC.
CLKOUT	inic	relógio do ciclo de instrução
Load A	ld_A	carrega o registo A com o conteúdo do barramento Sout
Load B1	ld_B1	carrega o registo B1, da unidade de manipulação de *bits*, com o conteúdo do barramento Sout
Load OPTION	load_option	carrega o registo OPTION com o conteúdo de W.
Load TRIS A…D	load_tris<3:0>	carrega um dos registos TRIS (A, B, C ou D) com o conteúdo de W
set bit	set_bit	coloca o *bit* especificado no nível lógico um

Tabela 4.2. Descrição dos sinais de controlo.

Na Tabela 4.3 encontram-se representados os valores das áreas de cada um dos circuitos que constituem a unidade de controlo completa (para a codificação de estados atribuída pela ferramenta, ou para a codificação de estados OHE[9], permitida pelo compilador de FSMs integrado na ferramenta de síntese [50]). Para a FSM2 foi realizada a "optimização booleana"[10] [54] com a qual o circuito sintetizado mantém aproximadamente o atraso do caminho crítico (≅17.33ns).

[9] Do Inglês ***O**ne-**H**ot **E**ncoding*.

[10] Uso das regras básicas da álgebra booleana para reduzir a área de um circuito (ex. a.!a=0, a+a=a, etc.). Esta opção nao considera o atraso do caminho crítico durante a optimização.

A unidade de controlo utilizada no PARMIC utiliza a codificação de estados atribuída por defeito pela ferramenta de síntese, pois permite economizar área de controlo ao utilizar o número mínimo de FFs[11]. Devido ao nível de *pipelining* entre os sinais de controlo e o *datapath* (as transições indesejadas nos sinais de controlo, devidas aos diferentes tempos de propagação, são isoladas), não foi necessária a utilização de codificações que eliminassem os *hazards* (transições indesejadas) nos sinais de controlo.

	Área (*sites*)	Nº de células	Atraso (ns)	Nº de FFs
FSM1	211	32	7.32	3
+ FSM2	1180/1014*	215/175*	17.71/17.73*	5
+ FSM3	80	10	5.87	2
= 3 FSMs	1305	217		10
FSM1 (OHE)	280	34	5.15	8
+ FSM2 (OHE)	1334	171	12.38	29
+ FSM3 (OHE)	131	15	4.12	5
= 3 FSMs (OHE)	1745	220		42
mestre	121	16	6	2
pipe1	703	64	3.17	25
pipe2	55	5	3.17	2

Tabela 4.3. Área, número de células e de FFs da unidade de controlo e dos registos de *pipelining*. (* "optimização booleana")

4.4 Ficheiro de Registos

O módulo do ficheiro de registos foi dividido em várias entidades VHDL, de modo a facilitar todo o processo de síntese, a verificação do comportamento, e a parametrização. A partição foi orientada pela funcionalidade e corresponde basicamente à estrutura representada no diagrama de blocos da Figura 4.1. Neste contexto, os registos especiais foram implementados em separado.

O ficheiro de registos recebe no máximo 7 *bits* do descodificador de instruções, que permitem endereçar um registo de um máximo de 80. Na Figura 4.11 são apresentados os sinais de interface da unidade que descodifica o endereço do registo especificado e o acesso aos registos do ficheiro de registos.

[11] Do Inglês ***F**lip-**F**lop*.

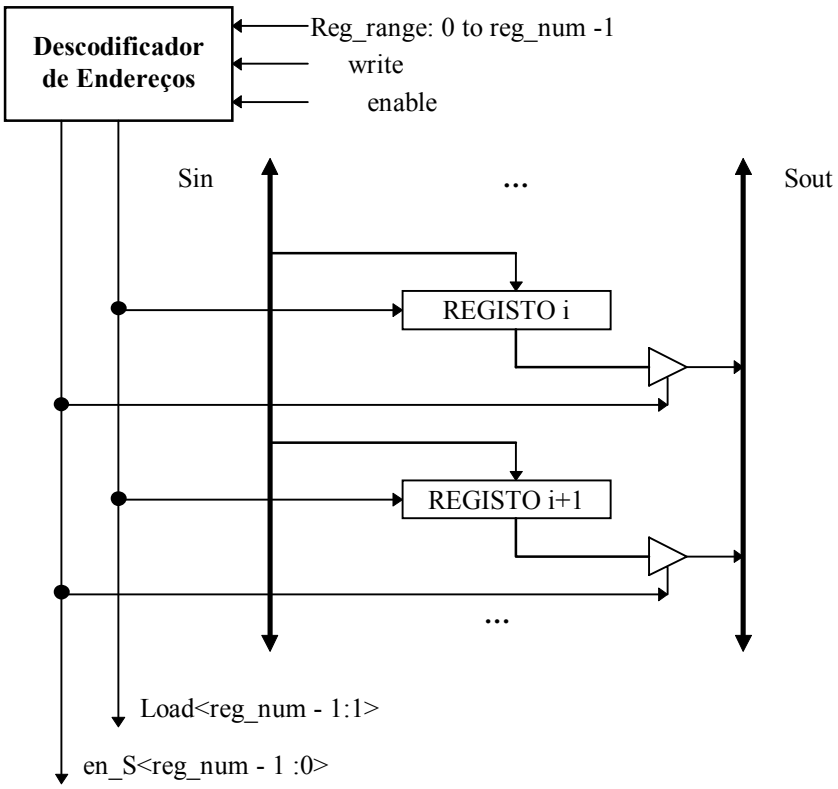


Figura 4.11. Estrutura de acesso aos registos do ficheiro de registos.

O descodificador de endereços produz dois sinais ("Load" e "en_S"), cujo número de *bits* é igual ao número de registos utilizados, com apenas um *bit* activado no sinal "Load" ou no sinal "en_S", que corresponde ao registo seleccionado, caso o comando seja de escrita ("write") ou de colocação no barramento ("enable") respectivamente.

Na Tabela 4.4 são apresentadas quantificações da área dos circuitos de descodificação, para várias dimensões do ficheiro de registos. O circuito referente à entidade "descodificador de endereços" com *clear*, corresponde à opção inicial em que era utilizado o sinal "*clear*" para colocar a zero um registo. Na solução final optou-se pela realização da operação de "*clear*" através da colocação de um operando zero no barramento seguida do carregamento no registo respectivo, o que permitiu reduzir em 31% a área do circuito de descodificação de endereços (ver Figura 4.12). São também apresentados, na última linha da Tabela 4.4, os atrasos máximos do circuito anterior.

	Nº de registos					
	7	13	26	32	64	80
Área (sites): desc_reg com *clear*	205	-	705	827	-	2281
Área (sites): desc_reg	121	243	444	503	1026	1577
Atraso (ns)	3.21	4.74	5.17	4.42	5.56	7.10

Tabela 4.4. Área para diferentes números de registos dos circuitos de descodificação e controlo dos sinais do ficheiro de registos.

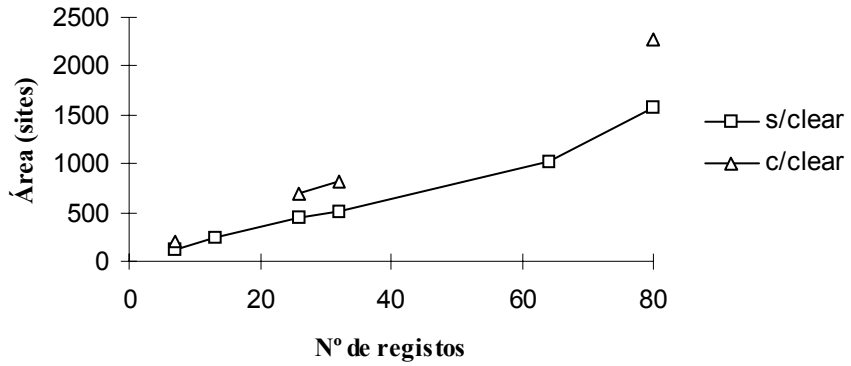


Figura 4.12. Representação gráfica da área do descodificador versus o nº de registos.

Na Tabela 4.5 estão ilustrados os tamanhos dos registos especiais e de cada registo normal. Para cada registo normal com sinal de *clear* obteve-se a área de 27.5 *sites*/*bit*[12], e sem sinal de *clear* 23.75 *sites*/*bit*. O *latch* utilizado inicialmente para cada *bit* de um registo, tem um sinal de *reset*, responsável pela colocação a zero do registo. A solução final, descrita anteriormente (sempre que se deseja colocar a zero um registo), permitiu a utilização do *latch* mais simples da biblioteca originando uma diminuição da área por cada *bit* de 3.75 *sites* (20 %).

Para o PC e RTCC são permitidas duas hipóteses de escrita: podem ser carregados pelo operando especificado na instrução (ex. o resultado da ALU), ou podem ser incrementados no flanco ascendente do relógio.

[12] relativa a 1 *latch* (A=16 *sites*), 1 *tristate-inverte* (A=8 *sites*), e 4 *buffers* por cada registo de 8 *bits* (A=7 *sites*).

Registo	Com sinal de *clear*		Sem sinal de *clear*	
	Área (*sites*)	Nº de células	Área (*sites*)	Nº de células
RTCC (F1)	578*	78*	529*	71*
PORTO E	-	-	195	19
PORTO (E/S: F5, F6, F7, F8)	365	63	339	43
PC (F2) com pilha de 2 níveis	1454*	189*	1513*	208*
SR (F3)	423	58	379*	49*
Fx (Normal), F4	220	20	190	18

Tabela 4.5. Área e número de células de cada registo (* registos implementados com FFs).

4.4.1 O Registo de Estado (SR)

A Figura 4.13 apresenta os sinais de interface do registo de estado, SR. Este registo permite o armazenamento do operando resultado e, em qualquer caso (mesmo que não seja endereçado pela instrução em execução) deve permitir a escrita na(s) *flag*(s) que a instrução em execução afectar. Os três *bits* mais significativos deste registo não representam qualquer *flag* e têm o comportamento de um registo normal.

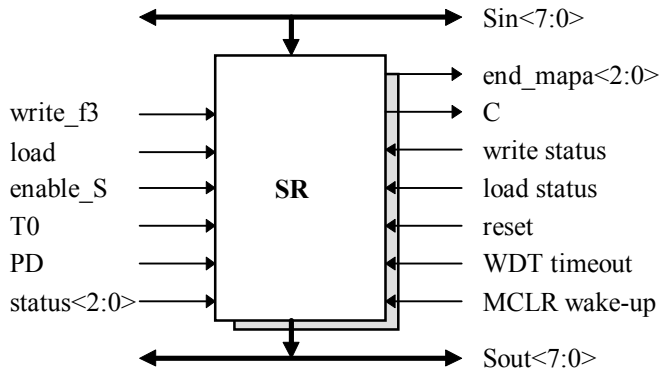


Figura 4.13. Sinais de interface do registo SR.

4.4.2 A pilha e o PC

No mesmo módulo foram englobados o circuito responsável pelo incremento do PC, o próprio PC (que pode ser acedido como f2), e a pilha parametrizável. Os sinais de interface desta unidade estão ilustrados na Figura 4.14.

A Descrição 4.1 especifica a pilha de n níveis de 9 FFs com estrutura do tipo LIFO (***L**ast **I**n **F**irst **O**ut*) e o PC. Ao ser encontrado o primeiro **CALL**, é armazenado no topo da pilha o valor do PC+1 (endereço da próxima instrução) e o ponteiro endereça o topo desta. No segundo **CALL**, e nos subsequentes, é armazenado o valor de PC+1 na posição seguinte da pilha e o ponteiro indica a nova posição. Este ponteiro é um

identificador que representa a posição no encadeamento da rotina e é constituído por um número de *bits* dependente do número de níveis da pilha.

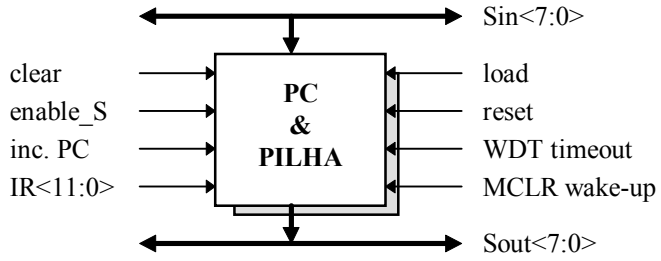


Figura 4.14. Sinais de interface do registo PC.

```
B1: IF (stack_depth > 0) GENERATE
process (inc_pc, reset, IR, load, SInt, MCLR_wake_up, WDT_time_out,
          PcCallInt, IrInt)
      VARIABLE pc_call : std_logic_vector(8 downto 0);
      VARIABLE stack_address : integer range 0 to stack_depth;
      subtype STACK_WORD is integer range 0 to 511;
      type STACK_TYPE is ARRAY (1 to stack_depth) OF STACK_WORD;
      VARIABLE STACK : STACK_TYPE;
      begin
          IF(reset = '1' OR MCLR_wake_up = '1'
            OR WDT_time_out = '1') THEN
            PC <= 0;
            stack_address := 0;
          ELSIF (inc_pc'event and inc_pc = '1' ) THEN
            IF (IR(11 downto 9) = GOTO) THEN
              PC <= IrInt;
            ELSIF (IR(11 downto 8) = CALL) THEN
              stack_address := stack_address + 1;
              STACK(stack_address) := PC + 1;
              PC <= PcCallInt;
            ELSIF (IR(11 downto 8) = RETLW) THEN
              PC <= STACK(stack_address);
              stack_address := stack_address - 1;
            ELSIF (load = '1') THEN
              PC <= SInt;
            ELSE
              PC <= PC + 1;
            END IF;
          END IF;
      end process;
END GENERATE B1;
```

Descrição 4.1. Código VHDL do processo de incremento, escrita do PC e controlo da pilha.

Na entidade VHDL desta unidade existe o mesmo processo para o caso em que não é necessário pilha. O compilador de VHDL opta por um deles através da construção IF ... GENERATE, em que o parâmetro da condição a testar se encontra definido num dos módulos de definições de parametrização.

A Tabela 4.6 e a Figura 4.15 apresentam os valores da área obtida e a representação gráfica desses valores, para vários níveis da pilha. Por interpolação linear foi obtida a fórmula (4.1), que permite estimar a área desta unidade.

Níveis da pilha	Área (*sites*)	Nº de células
0	693	96
2	1513	208
4	2399	328
8	3879	543

Tabela 4.6. Área do módulo que contém o PC e a pilha parametrizável.

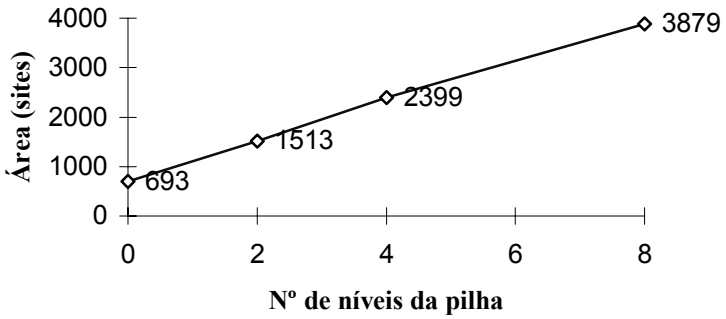


Figura 4.15. Área do módulo (PC + pilha) versus o número de níveis da pilha.

$$A_{PC+pilha} \cong 399 \times Num\_níveis + 725 \qquad (4.1)$$

4.4.3 O registo RTCC

O registo RTCC, com sinais de interface ilustrados na Figura 4.16, funciona como um registo normal do ficheiro de registos ou pode ser incrementado pelo sinal PSOUT gerado na unidade de sincronismo. O funcionamento como registo normal tem prioridade sobre o incremento.

A Descrição 4.2 apresenta a especificação deste registo (a arquitectura completa engloba o processo de interface com o barramento).

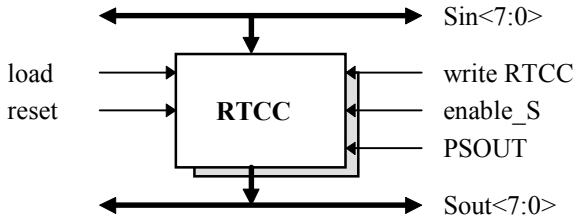


Figura 4.16. Sinais de interface do registo RTCC.

```
inc_RTCC1 <= PSOUT OR write_RTCC;

process(load, inc_RTCC1, temp_int, Sin, reset)
begin
      IF (reset = '1') THEN
            temp <= "00000000";
      ELSIF (inc_RTCC1'event AND inc_RTCC1 = '1') THEN
            IF (load = '1') THEN
                  temp <= Sin;
            ELSE
                  temp <= int2bits(temp_int + 1, 8);
            END IF;
      END IF;
end process;
```

Descrição 4.2. Especificação VHDL do registo RTCC.

4.4.4 Os Portos de entrada/saída

Cada porto de E/S pode ser configurado como porto de entrada ou como porto de saída. No caso de ser configurado como porto de entrada, qualquer instrução que utilize o registo como operando realiza a leitura dos pinos de saída durante meio ciclo do relógio e carrega-o no registo. No caso de ser configurado como saída, as instruções que tenham este registo como operando resultado, carregam-no no penúltimo meio ciclo de relógio. Na Figura 4.17 são apresentados os sinais de interface da unidade de cada porto de E/S.

Nos portos só de entrada, as instruções cujo endereço identifique algum destes registos permitem a leitura dos pinos directamente para o barramento interno (Sout), sendo possível ler o pino e carregar o conteúdo do barramento no registo W com apenas uma instrução: **MOVF #, W**. Estes registos não têm interface ao barramento Sin.

A Descrição 4.3 representa a especificação dos registos de E/S. Esta especificação ao ser sintetizada infere *latchs*, como se pode confirmar por um dos processos da descrição.

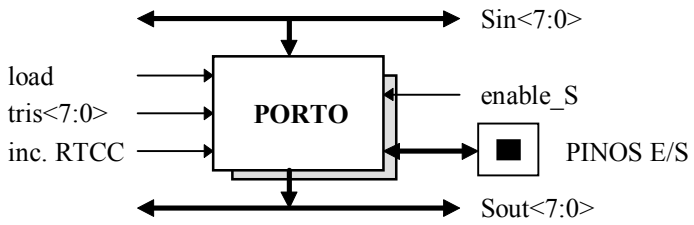


Figura 4.17. Sinais de interface do registo de cada porto de E/S.

```
process(load, S, PORTA)
begin
  IF (load = '1') THEN
    temp1 <= S;
  ELSE
    temp1 <= PORTA;
  END IF;
end process;


process(temp1, load, enable_S)
begin
 IF (load = '1'
     OR enable_S = '1') THEN
   temp <= temp1;
 END IF;
end process;
```

```
process(tris, temp)
begin
  for i in 0 to 7 loop
    IF tris(i) = '0' THEN
      PORTA(i) <= temp(i);
    ELSE
      PORTA(i) <= 'Z';
    END IF;
  end loop;
end process;

process(enable_S, temp)
begin
  IF enable_S = '1' THEN
    S <= temp;
  ELSE
    S <= "ZZZZZZZZ";
  END IF;
end process;
```

Descrição 4.3. Especificação VHDL dos portos de E/S.

Os sinais, "tris", de entrada de cada um dos portos representam o estado lógico de cada um dos *bits* dos registos TRIS conforme se trate do porto de E/S A, B, C, ou D. Estes registos (implementados com *latchs*) armazenam a configuração dos portos e são carregados com o conteúdo do registo W pela instrução **TRIS**. O valor lógico '1' num *bit* do sinal "tris" indica que o pino correspondente do porto de E/S está em alta impedância (configurado como saída), e o valor lógico '0' indica que o respectivo pino de E/S está configurado como entrada. Existem tantos registos TRIS quantos os portos de E/S.

O número máximo de portos de E/S é igual a 4, pois apesar da instrução **TRIS** permitir 3 *bits* de identificação (possibilidade de identificar 8 portos) alguns códigos não podem ser utilizados, porque em conjunto com o *opcode* desta instrução identificam outras instruções (os únicos identificadores permitidos para identificar um porto na instrução TRIS são: “001”, “101”, “110”, “111”). A colocação dos portos de E/S no ficheiro de registos é para o porto A: f5 (b’101), porto B: f6 (b’110), porto C: f7 (b’111), e porto D: f8 (b’001.).

Na Tabela 4.5 são ilustrados os resultados do registo sintetizado a partir de diferentes descrições. No caso A1 são instanciados *latchs* com *reset* e no caso A2 a ferramenta instancia *latchs* sem *reset*.

Registo	Síntese	Área (*sites*)	Nº de células
TRIS (para cada porto de E/S)	A1	136	8
	A2	192	21

Tabela 4.7. Área e número de células de cada registoTRIS.

4.5 A ALU

A Figura 4.18 mostra o bloco da ALU, os sinais de controlo e de entrada/saída, os registos de armazenamento temporário dos operandos (A e B) e o interface aos barramentos. A operação é seleccionada pelos 4 *bits* vindos do descodificador de instruções. Na Tabela 4.8 estão representados os códigos que permitem seleccionar a funcionalidade pretendida.

Os *bits* do registo de estado exercitados pelas operações da ALU são os três representados: Z, DC e C. O *bit* de entrada, denominado de C, é a *flag* de transporte do registo de estado utilizada pelas operações de deslocamento que colocam o valor lógico desta *flag* no bMs ou no bms do registo a deslocar dependendo do deslocamento ser efectuado para a esquerda ou para a direita respectivamente.

Esta unidade fornece o resultado vindo directamente do corpo principal da ALU e o mesmo resultado em *tri-state* activado pelo sinal “*disable*”.

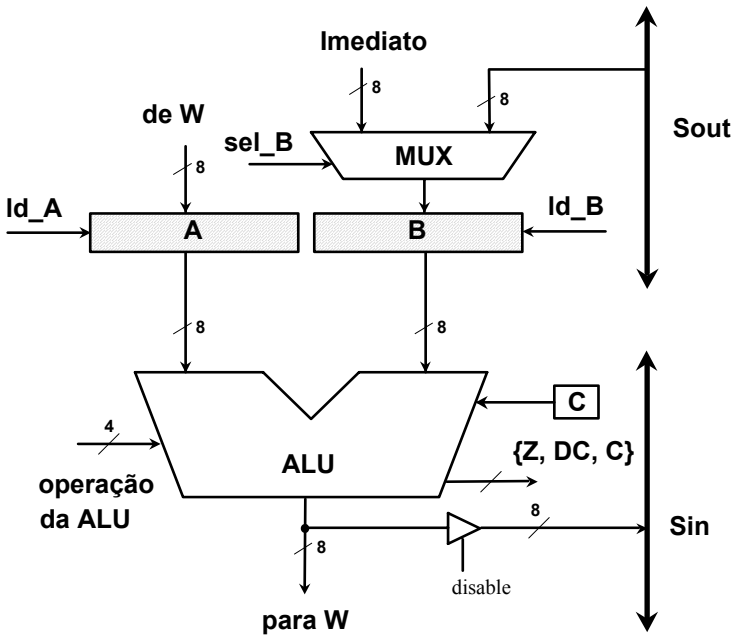


Figura 4.18. Diagrama de blocos da ALU e blocos auxiliares do interface com os barramentos.

São 17 as instruções que utilizam a ALU. É também considerada a detecção de zero do registo movimentado pela instrução **MOVF** (operação que afecta a *flag* **Z**).

Código da operação	Função	Descrição
"0111"	OR	"OR" de A com B
"0001"	AND	"AND" de A com B
"1010"	XOR	"XOR" de A com B
"0010"	COMP	complemento de B
"0000"	+	A + B
"1011"	-	B - A
"1100"	SWAP	troca os 4 bms pelos 4 bMs de B
"0110"	INC	incremento de B
"0011"	DEC	decremento de B
"1000"	RL	rotação para a esquerda de B (inclui o *bit* de transporte)
"1001"	RR	rotação para a direita de B (inclui o *bit* de transporte)
"1111"	ZERO	determina se B é zero

Tabela 4.8. Valor das quatro linhas de controlo e a respectiva operação da ALU.

A Tabela 4.9 ilustra os resultados da síntese da ALU em dois exemplos. No exemplo A, procedeu-se à síntese com os parâmetros de defeito e no exemplo B indicou-se 10 ns como atraso máximo entre a entrada e a saída da ALU (a ferramenta embora não tenha conseguido satisfazer a restrição produziu um circuito com 13.56 ns de atraso).

Síntese	Área (*sites*)	Nº de células	Atraso máximo (ns)
A	1708	300	25.9
B	1890	393	13.56

Tabela 4.9. Área, atraso e número de células da ALU para directivas de optimização diferentes.

4.6 A unidade de manipulação de *bits*

A unidade que manipula um *bit* específico do registo indicado foi implementada num módulo separado da ALU. Esta unidade ilustrada na Figura 4.19, é utilizada pelas instruções: **BCF**, **BSF**, **BTFSC** e **BTFSS**. Estas instruções originam o armazenamemto do valor do registo, colocado no barramento (Sout), no registo B1 e em seguida, de acordo com a instrução:

- o teste se o *bit* especificado tem o valor lógico zero ou um (**BTFSC** e **BTFSS**);
- a colocação do *bit* especificado a zero ou a um (**BCF** e **BSF**).

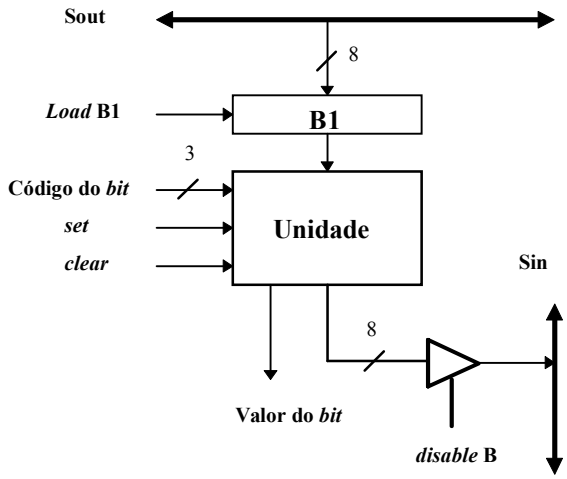


Figura 4.19. Unidade de manipulação de *bits*.

Por último, caso se esteja em presença da instrução **BCF** ou **BSF**, é colocado o novo valor do registo no barramento (Sin) para ser armazenado no registo especificado na instrução.

Na Tabela 4.10 são apresentados: a área, o atraso do caminho crítico e o número de células do circuito final.

Área (*sites*)	Nº de células	Atraso máximo (ns)
459	53	7.85

Tabela 4.10. Área, atraso, e número de células da unidade de manipulação de *bits*.

4.7 Descodificador

O módulo descodificador (com sinais de interface apresentados na Figura 4.20) gera os sinais necessários para a execução de cada instrução que não necessitam de mudar de valor lógico ao longo do ciclo de instrução. A descodificação de alguns dos sinais de saída pode ser observada na Tabela 4.11.

Figura 4.20. Sinais de interface do descodificador de instruções.

Neste módulo é também determinado o registo do operando quando é especificado o registo zero (endereçamento relativo), e indicado se o registo do operando é o PC ("reg_pc"), o RTCC ("write_f1"), ou o SR ("write_f3").

A descodificação é realizada durante o último ciclo do relógio (OSC1) do ciclo de instrução e foi representada por ID na Figura 4.2. Os sinais de saída são guardados no início do ciclo de instrução para a próxima instrução.

Na Tabela 4.12 podem ser observados os resultados obtidos com diferentes directivas de optimização. O circuito A, representa o circuito obtido utilizando as directivas de defeito da ferramenta de síntese, o circuito B foi obtido pela recompilação do anterior indicando à ferramenta a minimização da área, o circuito C foi obtido pela optimização booleana e indicando para que a ferramenta não tivesse em conta a informação temporal, e por fim o circuito D foi obtido indicando a "optimização boolena". Este último apresenta a menor área e um atraso aceitável.

IR	T0	PD	Inst. ALU	load status	sel_W	write_f3	reg_pc	write_fl
SLEEP	1	0	----	11000	0	0	0	0
CLRWDT	1	1	----	11000	0	0	0	0
OPTION || NOP	0	0	----	00000	0	0	0	0
TRIS	0	0	----	00000	0	0	0	0
CLRW	0	0	----	00100	0	0	0	0
CLRF	0	0	----	00100	0	f3	PC	RTCC
MOVWF	0	0	----	00000	1	f3	PC	RTCC
ADDWF	0	0	0000	00111	1	f3 & d	PC & d	RTCC & d
ANDWF	0	0	0001	00100	1	f3 & d	PC & d	RTCC & d
COMF	0	0	0010	00100	1	f3 & d	PC & d	RTCC & d
DECF	0	0	0011	00100	1	f3 & d	PC & d	RTCC & d
DECFSZ	0	0	0011	00000	1	f3 & d	PC & d	RTCC & d
INCF	0	0	0110	00100	1	f3 & d	PC & d	RTCC & d
INCFSZ	0	0	0110	00000	1	f3 & d	PC & d	RTCC & d
IORWF	0	0	0111	00100	1	f3 & d	PC & d	RTCC & d
RLF	0	0	1000	00001	1	f3 & d	PC & d	RTCC & d
RRF	0	0	1001	00001	1	f3 & d	PC & d	RTCC & d
XORWF	0	0	1010	00100	1	f3 & d	PC & d	RTCC & d
SUBWF	0	0	1011	00111	1	f3 & d	PC & d	RTCC & d
SWAPF	0	0	1100	00000	1	f3 & d	PC & d	RTCC & d
MOVF	0	0	1111	00100	1	0	0	0
MOVLW	0	0	0000	00000	0	0	0	0
RETLW	0	0	----	00000	0	0	0	0
ANDLW	0	0	0001	00100	1	0	0	0
IORLW	0	0	0111	00100	1	0	0	0
XORLW	0	0	1010	00000	1	0	0	0
BCF || BSF	0	0	----	00000	-	f3	PC	RTCC
GOTO	0	0	----	00000	-	f3	0	0

Tabela 4.11. Descodificação de instruções.

Circuito	Área (*sites*)	Nº de células	Atraso máximo (ns)
A	975	138	10.96
B	969	128	12.81
C	983	133	17.53
D	926	116	11.89

Tabela 4.12. Área, número de células, e atraso do descodificador.

4.8 O Datapath

O *datapath*, cujo diagrama de blocos se encontra ilustrado a sombreado na Figura 4.21, é constituído pela ALU, unidade de manipulação de *bits*, registos transparentes para o programador, o registo W e o ficheiro de registos, que partilham dois barramentos de 8 *bits*. A utilização de dois barramentos unidireccionais, em vez do barramento bidireccional inicial, permite que a carga total dos circuitos que acedem ao barramento seja dividida pelos dois. Na solução inicial (utilização de um único

barramento) a colocação de um operando do ficheiro de registos no barramento encontrava as capacidades de entrada dos *latchs* que formam os registos.

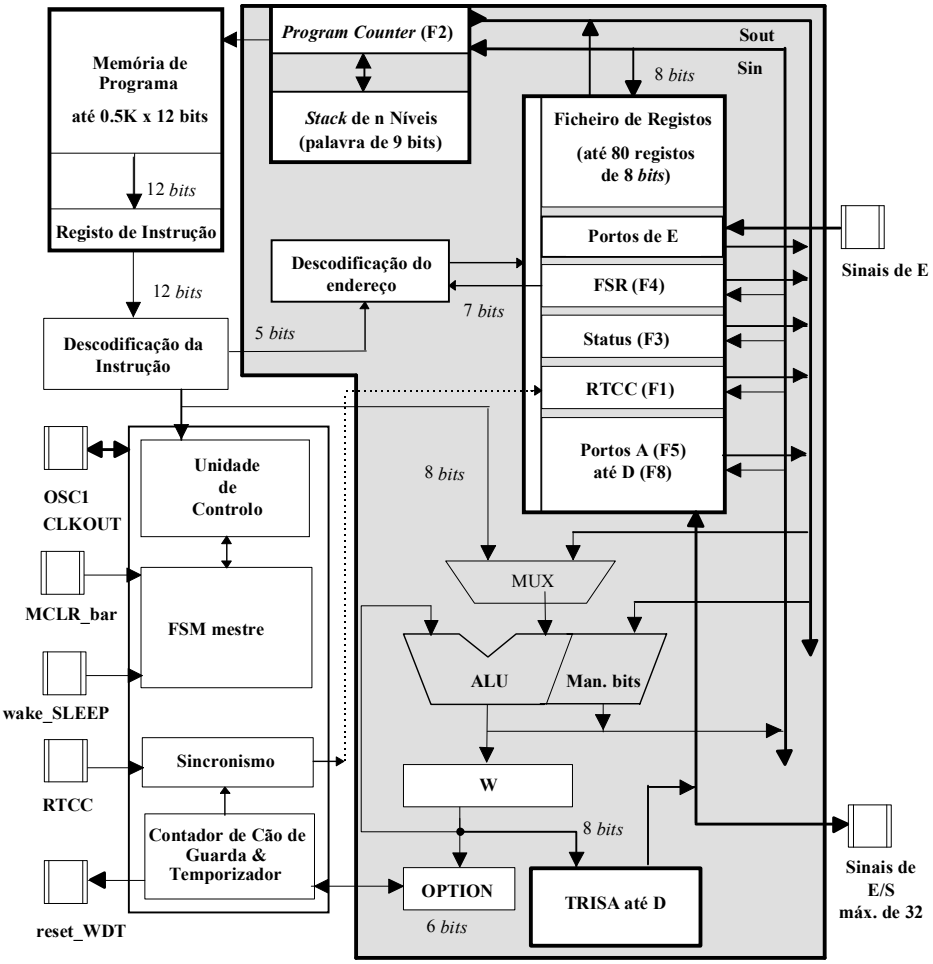


Figura 4.21. Diagrama de blocos do *datapath*.

Na Figura 4.22 podem ser observados os acessos ao registo W, e do registo W e da ALU ao barramento (Sin).

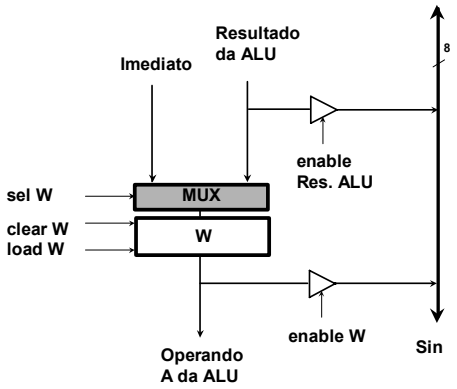


Figura 4.22. Diagrama de blocos do acesso ao registo W.

4.9 Memória de Programa e o registo IR

O circuito da memória de programa, com sinais de interface ilustrados na Figura 4.23 é constituído pelo registo de 12 FFs que armazena a instrução (IR) e pelo programa.

Figura 4.23. Sinais de interface da unidade que contém a memória de programa e o registo IR.

A especificação do funcionamento do registo IR, que representa o carregamento da instrução, é apresentada na Descrição 4.4. Em caso de *reset*, caso a condição de salto seja verdadeira quando em presença de instruções de salto condicional, ou caso o PC seja o registo destino, é colocado um **NOP** no registo de instrução. O sinal "*fetch*" é gerado no início do ultimo período do ciclo de instrução nos casos normais, exceptuando o caso em que há "*reset*" ou "*skip_next*" (é colocado um **NOP** no IR). Quando houver uma transição ascendente no sinal de "*fetch*" e o sinal "*reg_pc*" tiver o valor lógico '1' é também colocada a instrução **NOP** no registo de instrução.

```
FETCH_INST0: process(fetch, reset, reg_pc, skip_next, PC)
begin
      IF (reset = '1' OR skip_next = '1') THEN
            IR <= NOP;
      ELSIF (fetch = '1' AND fetch'event) THEN
            IF (reg_pc = '1') THEN
              IR <= NOP;
            ELSE
              IR <= ROM(PC);
            END IF;
      END IF;
end process;
```

Descrição 4.4. Parte da especificação do registo IR e da memória do programa.

A memória de programa é definida como uma matriz de constantes, que representam as instruções, num módulo de VHDL.

Para estimar o tamanho da memória de programa foram originados programas aleatórios com base no *opcode* de cada instrução, orientados pela distribuição normal do conjunto de instruções. A síntese destes pseudo-programas originou a Tabela 4.13

e a representação gráfica da Figura 4.24. Com base na Tabela 4.13, em que a área para o registo IR e lógica adicional é de aproximadamente 300 *sites*, obtém-se a relação área/*bit* aproximadamente igual a 1.05 *sites*/*bit* para um programa de 32 instruções. Esta relação permanece aproximadamente constante com o aumento do número de instruções do programa (≅1.02 *sites*/*bit* ou 2.04 transístores/*bit*).

Nº de Instruções (12 *bits*/ instrução)	Área de memória no SOG (*sites*)	Tempo de acesso à instrução t_{aces} (ns)
32	704	14
64	1176	15
128	2016	17
256	3938	19
512	6581	24

Tabela 4.13. Área e tempo de acesso da memória do programa.

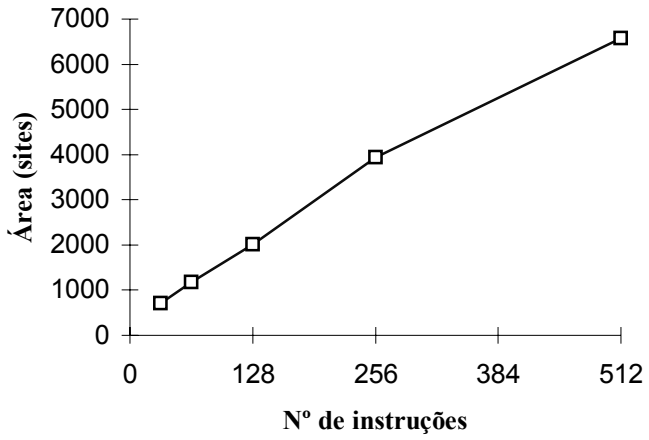


Figura 4.24. Gráfico da área de programa versus o nº de instruções.

Em virtude da relação de área por *bit* ser pequena não foi necessário realizar um compilador de ROMs para o SOG. A memória de programa gerada é constituída por lógica aleatória.

4.10 Os Temporizadores e o registo OPTION

O WDT e o contador/relógio RTCC foram implementados como no PIC. Existe uma FSM que sincroniza o incremento do registo F1 (RTCC) com os sinais de interface apresentados na Figura 4.25.

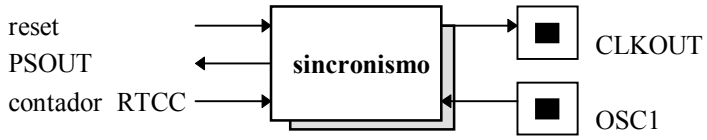


Figura 4.25. Sinais de interface da máquina de sincronismo.

O sinal gerado pelo temporizador é amostrado em todos os flancos ascendentes do relógio OSC1, e quando é encontrado o nível lógico '1' é gerado o sinal que incrementa o registo no terceiro ciclo do OSC1 (para o ciclo de instrução em causa) como se pode observar pela Figura 4.26. Esta FSM também gera o sinal de relógio de instrução (CLKOUT).

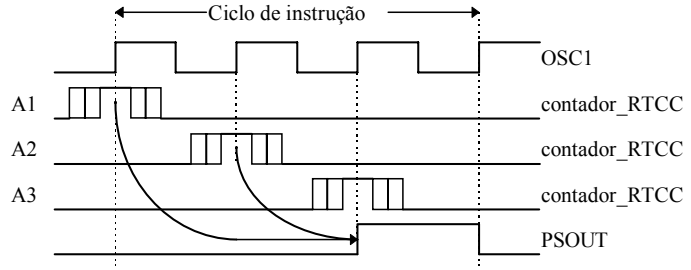


Figura 4.26. Geração do sinal PSOUT, que incrementa o registo RTCC.

A unidade que contém o temporizador é acedida pelos sinais representados na Figura 4.27.

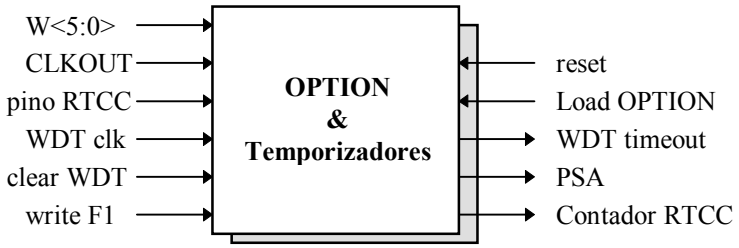


Figura 4.27. Sinais de interface da unidade OPTION.

O processador não engloba o oscilador interno do WDT, com período de 18 ms, presente na família PIC. Como solução, o processador engloba um pino de entrada de

sinal ("WDT clk") que permite ao projectista definir o período (poderá ser implementado por um oscilador externo) com que é controlado o WDT (em conjunto com o pré-escalar) e um pino de saída ("reset_WDT") para reinicializar o oscilador externo, quando o programa em execução ordenou o *clear* do WDT. O sinal de saída "WDT *timeout*" é o sinal oriundo do temporizador do WDT e acorda o processador, caso este esteja em adormecido, ou faz o *reset* interno. Este sinal pode vir directamente do oscilador externo referido ou do pré-escalar caso este se encontre atribuído ao WDT. O sinal de saída "contador RTCC" após ser sincronizado com o último período do ciclo de instrução na unidade de sincronismo controla o incremento do registo RTCC.

A Descrição 4.5 especifica o registo OPTION, que é carregado com os 6 *bits* menos significativos do registo W. O registo OPTION é constituído pelos 3 *bits* que representam o pré-escalar (pre_escalar <= PS2 & PS1 & PS0, com PS2 <= temp(2), PS1 <= temp(1), e PS0 <= temp(0)), pelo *bit* que indica o flanco utilizado para o incremento do RTCC (RTE <= temp(4)), pelo *bit* que indica a origem do sinal que despoleta o incremento (RTS <= temp(5)), e pelo *bit* que atribui o pré-escalar (PSA <= temp(3)).

```
process(load_option, reset, from_W, PSA1, write_f1, clear_WDT)
begin
      IF (reset = '1') THEN
            temp <= "111111";
      ELSIF (load_option'event AND load_option = '0') THEN
            if ((write_f1 and not(PSA1)) = '1') then
                  temp(2 downto 0) <= "000";
            elsif (clear_WDT = '1' and PSA1 = '1') then
                  temp(2 downto 0) <= "000";
            else
                  temp <= from_W;
            end if;
      END IF;
end process;
```

Descrição 4.5. Código VHDL do processo que inicializa o registo OPTION.

Existe apenas um temporizador com pré-escalar para o RTCC e WDT. Por isso, apenas um dos temporizadores pode ser programado de cada vez. A Descrição 4.6 especifica o circuito de temporização, com o incremento a ser efectuado de um em um, ou de dois em dois, conforme a selecção definida na *flag* PSA do registo OPTION

(WDT ou o temporizador). O pré-escalar definido no registo OPTION vai seleccionar uma das saídas deste contador de 8 *bits*, de forma a permitir a programação do incremento do RTCC ou do período do *timeout* do WDT.

```
process(load_option, reset, CLK, PSA1, clear_WDT, pre_escalar,
        WDT_clk)
          variable aux: integer range -2 to 255;
          variable bit_num: integer range 0 to 7;
          variable aux_bit: std_logic_vector(7 downto 0);

begin
      IF ((reset = '1') OR (clear_WDT = '1' and PSA1 = '1') OR
         (load_option = '1')) THEN
            aux := 0;
            aux_bit := "00000000";
            bit_num := 0;
            pre_aux <= '0';
      ELSIF (CLK'event AND CLK = '1') THEN
          --synopsys translate_off
            if (PSA1 = '1' and aux > 253) then
                  aux := -2;
            elsif (aux > 254) then
                  aux := -1;
            end if;
          --synopsys translate_on
            if (PSA1 = '1') then
                  aux := aux +  2;
            else
                  aux := aux +  1;
            end if;
            aux_bit := int2bits(aux,8);
            bit_num := bits2int(pre_escalar);
            pre_aux <= aux_bit(bit_num);
      END IF;
end process;
```

Descrição 4.6. Código VHDL do processo que executa o incremento.

A Descrição 4.7 especifica a selecção do pré-escalar para o temporizador do WDT ou para o contador/relógio RTCC e, no caso da selecção do RTCC, a escolha do sinal que vai habilitar a contagem (o relógio de instrução, CLKOUT, ou o sinal presente no pino "RTCC_pin"). Quando este sinal é oriundo do pino "RTCC_pin", é possível efectuar a contagem no flanco ascendente ou no flanco descendente, ao definir a *flag* RTE do registo OPTION, como '0' ou '1' respectivamente.

A área e número de células das unidades OPTION+Temporizador e de sincronismo estão representadas na Tabela 4.14.

```
process(PSA1, WDT_clk, CLK1)
begin
      IF (PSA1 = '1') THEN
            CLK <= WDT_clk;
      ELSE
            CLK <= CLK1;
      END IF;
end process;
```

```
process(RTE, RTS, RTCC_pin, CLKOUT)
begin
    IF (RTS = '1') THEN
          CLK1 <= RTE XOR RTCC_pin;
    ELSE
          CLK1 <= CLKOUT;
    END IF;
end process;
```

Descrição 4.7. Código VHDL dos processos de selecção do sinal de incremento e da atribuição do pré-escalar.

```
process(pre_aux, PSA1, WDT_clk, CLK1, temp)
begin
      IF (PSA1 = '1' AND temp(2 downto 0) /= "000") THEN
            WDT_time_out <= pre_aux;
            contador_RTCC <= CLK1;
      ELSIF (PSA1 = '1' AND temp(2 downto 0) = "000") THEN
            WDT_time_out <= WDT_clk;
            contador_RTCC <= CLK1;
      ELSE
            WDT_time_out <= WDT_clk;
            contador_RTCC <= pre_aux;
      END IF;
end process;
```

Descrição 4.8. Código VHDL do processo que fornece os dois sinais de saída de incremento do RTCC ou de *timeout* do WDT.

Unidade	Área (*sites*)	Nº de células
OPTION+Temporizador	777	104
sincronismo	279	34

Tabela 4.14. Área e número de células da unidade de sincronismo e da unidade OPTION.

4.11 Desempenho

O caminho crítico, que restringe a frequência máxima do relógio, foi identificado como correspondendo ao endereçamento e captura da instrução na memória do programa, e carregamento do registo de instrução:

$$t_{prop} = t_{mem(PC)} + t_{setup(IR)} \quad (4.2)$$

O atraso relativo à procura da instrução correspondente ao novo valor do PC ($t_{mem(PC)}$) depende do tamanho da memória do programa. O último atraso ($t_{setup(IR)}$) é necessário, e de cerca de 2.75 ns, para que não haja violação do tempo de *setup* dos FFs que constituem o registo da instrução (IR). Esta operação é realizada no segundo período do ciclo de instrução.

A determinação do PC, desde que não seja utilizado como operando destino de uma instrução, para a execução da instrução seguinte é feita no primeiro ciclo do relógio OSC1 e realizada em aproximadamente 8 ns.

Na Tabela 4.15 podem ser visualizados os atrasos críticos e identificadas as operações correspondentes. Nela pode também ser observado o número de ciclos alocados e a frequência máxima do relógio (OSC1) que não viola o atraso para cada um dos caminhos críticos, embora não se considere o atraso dos sinais de controlo (correspondente ao carregamento no *pipelining* de controlo).

Operação	Atraso (ns)	Nº de ciclos (OSC1) alocados	Frequência máxima do relógio OSC1 prevista (MHz)
ALU	21.5	1	46.5
PC+1	8	1	125
MEM(PC)	2.75 + t_{mem}	1	37.3 (c/ 512 inst.)
descodificação	11.89	1	84.1
Ler de um porto de E/S e colocação em Sout	18+2.75	1	48.1
FSM2	12.38	1	80

Tabela 4.15. Alocação das micro-operações críticas pelos ciclos do ciclo de instrução.

Simulações lógicas do processador indicam como frequência de relógio máxima 35 MHz (resultado confirmado pelos cálculos no caminho crítico ≅ 37 MHz).

4.12 Um CI de teste

De forma a aumentar a testabilidade do processador, foi adicionado um modo de teste seleccionado pelo pino TEST_MODE. Este modo de teste permite controlar pontos internos (ligação dos pontos internos a sinais externos: RESET_TEST, CLK_TEST, OSC1_TEST, e CLK_IR). Este circuito com 113 *sites*, contém as ANDs dos sinais de

controlo combinados e os multiplexers dos sinais de relógio e de *reset* que permitem a controlabilidade externa destes sinais em modo de teste.

O primeiro integrado fabricado contém 7 cadeias de varrimento (inseridas automaticamente pela ferramenta de síntese), constituídas por FFs com entrada multiplexada, que permitiram que a testabilidade do integrado aumentasse ligeiramente para 26.15% (com a utilização dos 119 padrões de teste gerados automaticamente pela ferramenta de síntese e posteriormente convertidos para um formato ascii utilizado pelo equipamento de ATE[13], quando do teste físico). Os resultados obtidos para a cobertura de faltas (FC[14]) devem-se ao reduzido FC (7.3%) do módulo que constitui o *datapath*. Este facto baseia-se no elevado número de *latchs* contidos neste módulo. Embora se tenha utilizado na inserção das cadeias de varrimento e na geração de padrões de teste a directiva permitida pela ferramenta que torna estes elementos transparentes [55] (na prática, corresponde a considerá-los elementos combinatórios) os resultados não revelaram melhorias. Estes resultados indicam a necessidade de introdução de um esquema que permita aumentar a FC do *datapath*, originando níveis de defeito (DL[15]) certamente mais aceitáveis.

O CI de teste foi implementado num SOG GF4 (ver Figura 4.28) e é constituído por 16118 *sites* (2272 células, cerca de 4000 portas equivalentes). Este CI contém um programa de teste com 135 instruções, o WDT e o contador/relógio em tempo real, e o ficheiro de registos é constituído por 13 registos (2 portos de E/S).

A Figura 4.29 e Figura 4.30 ilustram a simulação funcional do processador com o programa de teste referido anteriormente. A primeira figura apresenta o início da simulação e a segunda figura apresenta a simulação numa fase posterior. Na fase da simulação, ilustrada na Figura 4.30, o processador encontra-se em modo de adormecimento (instrução **SLEEP** no endereço de memória de programa 133, que desactiva o sinal de relógio interno "clock" responsável pelo funcionamento da unidade

---

[13] Do Inglês ***A**utomatic **T**est **E**quipment*.

[14] Do Inglês ***F**ault **C**overage*.

[15] Do Inglês ***D**efect **L**evel*.

de controlo) e é posteriormente "acordado" por um sinal externo (pino "wake_SLEEP" ao nível lógico '1'). Depois do processador ter entrado em execução normal é produzido o *timeout* do temporizador do WDT (pino "WDT_CLK" a '1') que força o contador de programa a regressar novamente ao endereço zero da memória de programa.

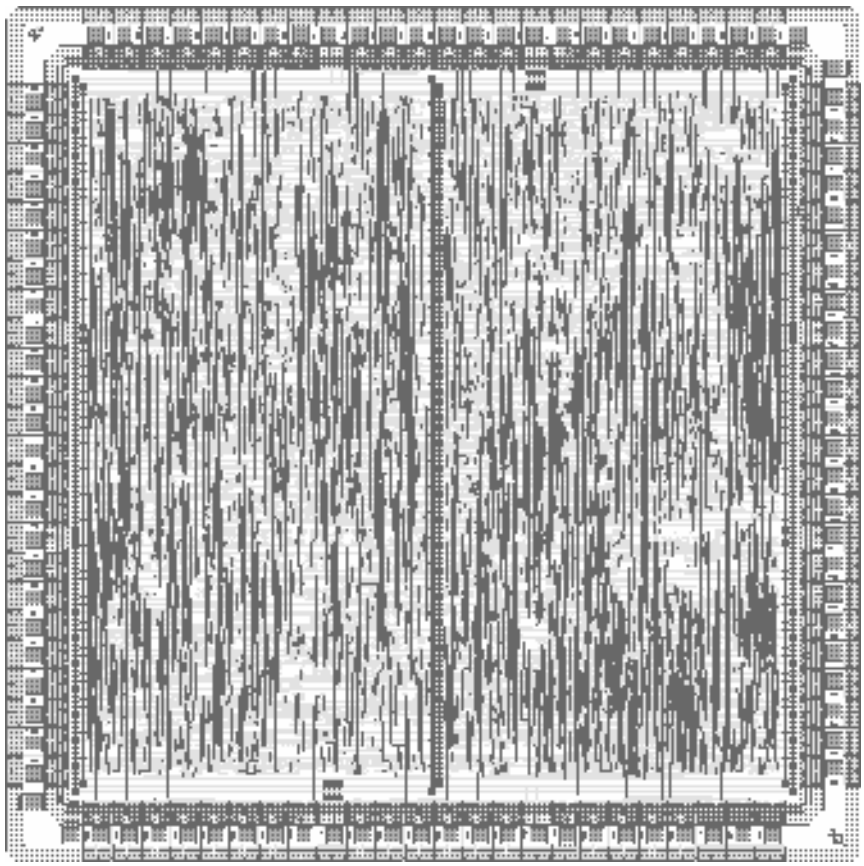


Figura 4.28. *Layout* do CI de teste.

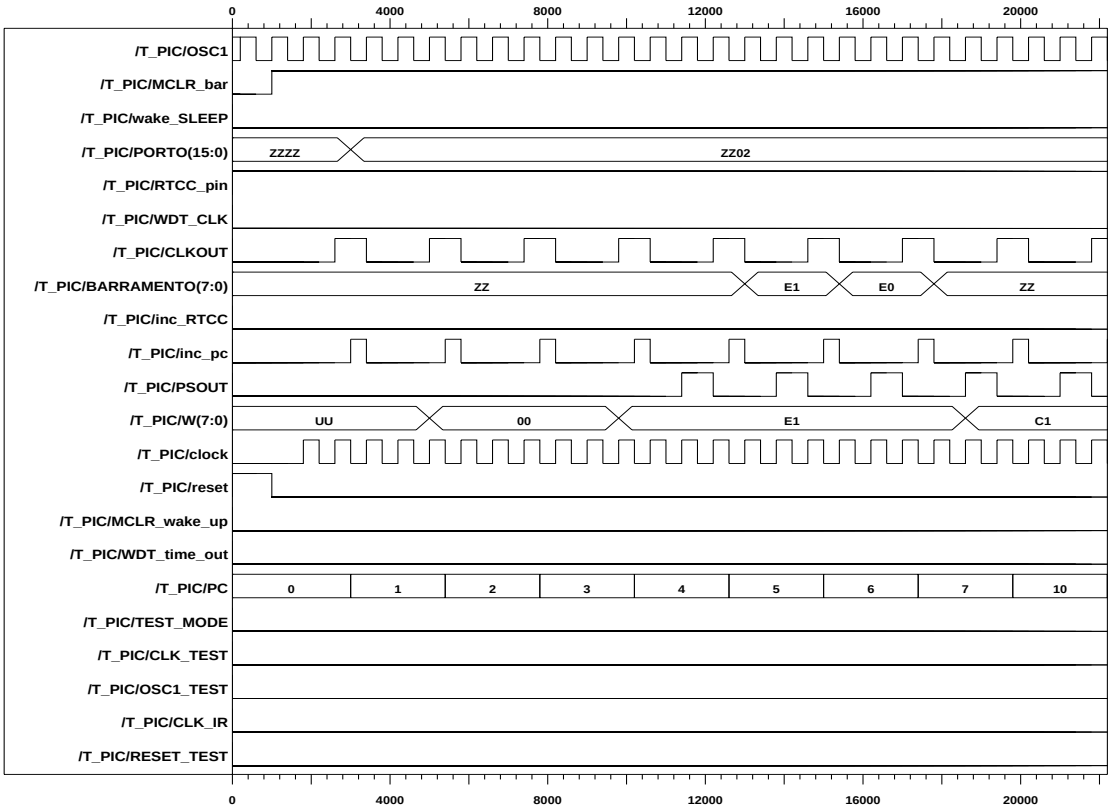


Figura 4.29. Simulação funcional do processador.

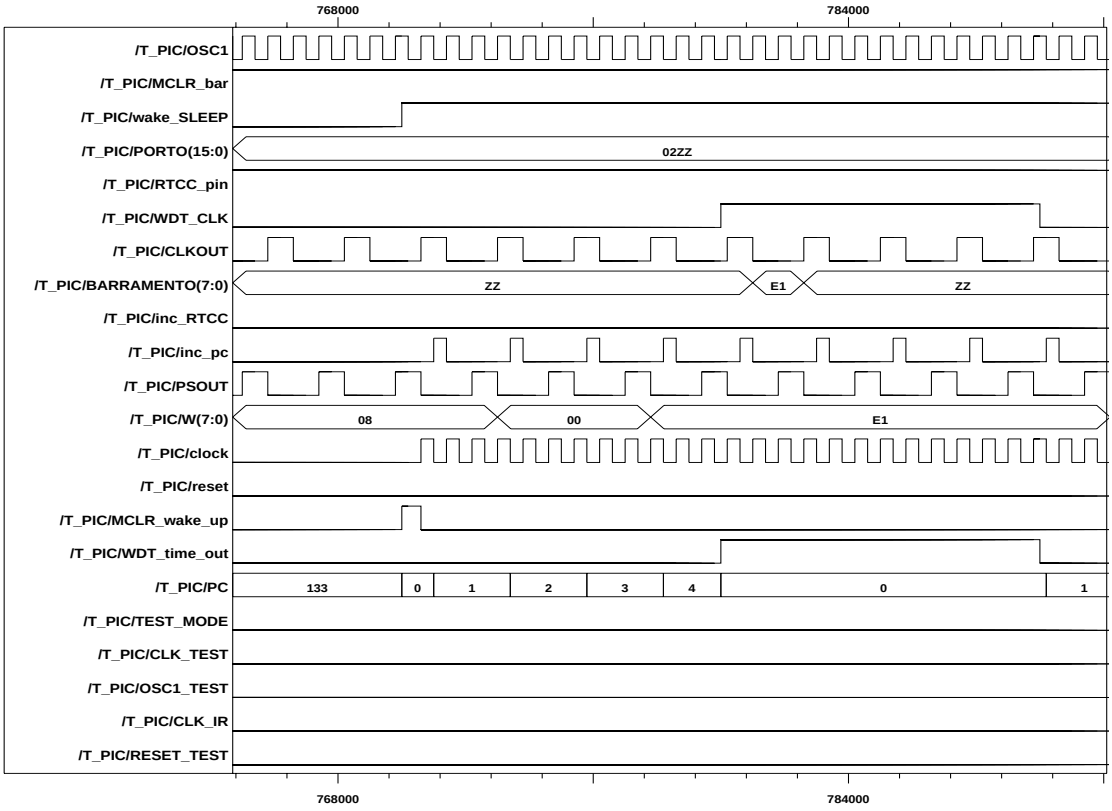


Figura 4.30. Simulação funcional do processador (continuação).

4.13 Conclusões

A capacidade de parametrização do processador descrito é garantida pelas facilidades de parametrização do VHDL, com base em módulos onde são definidos os parâmetros do processador, que permitem a parametrização automática, utilizando construções de instanciação condicional, atribuição condicional de sinais, parametrização da dimensão de vectores, etc. Com estas construções (IF ... GENERATE, FOR ... GENERATE) foi possível tornar a descrição do processador parametrizável sem alterações do código da descrição.

A metodologia de projecto utilizada permitiu testar diversas topologias e partições da arquitectura sem hipotecar demasiado o tempo útil de projecto. O principal objectivo cumprido foi a realização de uma unidade central com área reduzida, sem limitação de desempenho. A frequência máxima de 35.7 MHz de operação conseguida é adequada para a maioria das aplicações alvo, e representa um aumento de pelo menos 133% de desempenho em relação aos microcontroladores da família PIC (@20MHz) e, aumentos de desempenho maiores que 33% (com igual frequência de relógio: ambos a @20MHz). Aumentos adicionais do desempenho são possíveis com um maior número de iterações do fluxo de projecto e re-escalonamento das fases de execução de uma instrução.

A solução implementada quando se deseja colocar a zero um determinado registo permite reduzir no mínimo em 31% a área do circuito de descodificação de endereços e em 20% a área ocupada pelos registos.

A validação do processador foi realizada por simulações funcionais e lógicas exaustivas.

A redução do consumo de potência foi também considerada, tendo sido utilizadas técnicas de desactivação das unidades não utilizadas pela instrução em execução.

A não incorporação da unidade de manipulação de *bits* na ALU garante, fundamentalmente, a instanciação condicional da primeira unidade e reduz a

complexidade da ALU (melhor desempenho). A utilização de dois registos auxiliares B, um para cada uma destas unidades, permite que ao ser colocado um operando em apenas um deles não estejam ambas as unidades em funcionamento.

Em vez da utilização de *latchs* em W e em A, poder-se-ia ter optado pela colocação de FFs em W sem a necessidade do registo auxiliar A. Contudo, esta solução originava, ao ser colocado um operando em W (por ex. um imediato), a activação da ALU desnecessariamente.

Em modo de adormecimento (SLEEP), a máquina mestre encontra-se no estado de SLEEP, e apenas estão activadas a unidade de temporização e a unidade de sincronismo, permitindo a diminuição do consumo de potência.

5. O Sistema de Co-Síntese

"Only by rationalizing and automating embedded system design can we take advantages of the vast increases in programmable computing power delivered by VLSI."

Wayne Wolf

Este capítulo descreve a realização do ambiente COSTLES: sistema de co-síntese de sistemas embebidos digitais em que a arquitectura alvo é constituída por um único circuito integrado num *Gate-Array* do tipo *Sea-Of-Gates* (SOG). O circuito integrado inclui a unidade central descrita no capítulo anterior (componente *software*), e um conjunto de unidades funcionais (componente *hardware*) que permitem acelerar a execução de determinados segmentos de código especificados pelo projectista. São explicadas as directivas que conduzem o processo de mapeamento do binómio *hardware/software*.

5.1 O Sistema Proposto

O sistema de co-síntese desenvolvido neste trabalho baseia-se numa unidade central (PARMIC) com capacidades de parametrização, tais como: memória de programa, memória de dados, número de portos de E/S, número de portos de entrada, número de portos de saída, existência de unidade de manipulação de *bits*, existência de cão de guarda, e existência de um contador/relógio em tempo real de 8 *bits*. Os parâmetros que configuram o PARMIC são gerados automaticamente a partir da especificação inicial, permitindo ajustar cada realização específica do processador à aplicação alvo,

de modo a não desperdiçar potência consumida, memória de dados e de programa, e/ou área ocupada desnecessariamente.

O ambiente de co-síntese, COSTLES, encontra-se representado na Figura 5.1. O sistema é especificado usando uma metodologia orientada pelo *software* directamente compatível com o código *assembler* do processador da unidade central. Esta solução permite um nível de granularidade fino, adequado para as aplicações visadas (sistemas embebidos de pequena/média complexidade). O ambiente permite compatibilidade directa com as ferramentas de desenvolvimento comerciais, disponíveis para os microcontroladores PIC (por ex. MPALC [48] e MPSIM [49]).

A aplicação BINOMIO[1] (ver Apêndice A) reúne as tarefas de compilação, que permitem a migração automática do *software* para o *hardware* (tradutor do código *assembler* para a correspondente especificação em VHDL sintetizável) no ambiente COSTLES.

Esta aplicação extrai, do código *assembler*, a informação (tamanho do programa, número de registos, portos de E/S, etc.) usada para guiar a geração dos blocos configuráveis (ROM, ficheiro de registos, portos de E/S, etc.). A utilização de descrições em VHDL parametrizáveis, permite a geração automática destes blocos, de modo eficiente.

O projectista pode usar directivas (ver Apêndice B) encapsuladas com o código *assembler* para guiar a partição *hardware*/*software*. Estas directivas impõem restrições temporais em determinados segmentos de código e/ou impõem a execução paralela de diferentes segmentos. No caso de violação das restrições temporais associadas a um determinado segmento de código ou da existência de directivas concorrentes, este componente migra para o *hardware* e é substituído pelo código necessário para fazer a comunicação com a unidade funcional auxiliar. A área e o tempo de execução das unidades funcionais são obtidos da ferramenta de síntese.

---

[1] *From Software-**B**ased Spec**I**ficatio**N** to Software/Hardware Transf**O**r**M**ation in the* COSTLES *Env**I**r**O**nment*.

O interface com as unidades funcionais (UFs[2]) é realizado por um esquema de mapeamento de registos, que não requer nenhuma modificação da arquitectura do processador da unidade central.

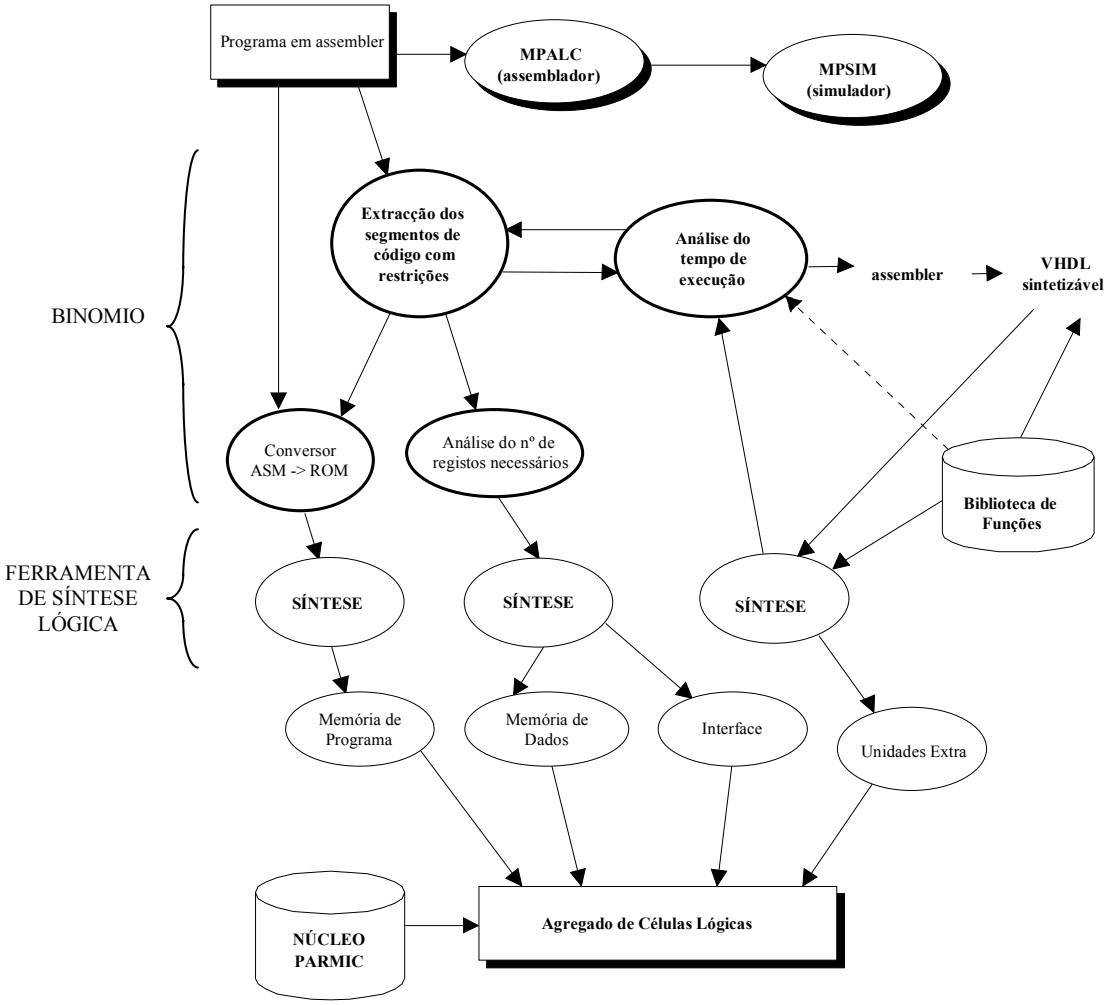


Figura 5.1. Ambiente de co-síntese COSTLES.

5.2 Arquitectura do Sistema

O sistema baseia-se no núcleo descrito no capítulo anterior que partilha os barramentos internos com as unidades extra. O interface das unidades extra ao núcleo é realizado

[2] Muitas vezes designadas ao longo do texto por unidades funcionais extra, unidades funcionais auxiliares, ou, simplificadamente unidades extra ou unidades auxiliares.

sem adicionar instruções específicas e sem alterar o núcleo. É apenas adicionado um circuito baseado num desmultiplexador.

Os dois barramentos internos do núcleo são usados como meio de comunicação entre este e as unidades funcionais (ver Figura 5.2). A versatilidade desta solução de interface permite adicionar unidades funcionais com baixo custo adicional. O núcleo actua como líder do barramento, sendo responsável pela sincronização com as UFs.

A colocação de um operando numa UF é feita utilizando a instrução **MOVF** (transferência de um registo para o registo W, ou para o próprio registo), que, coloca o conteúdo do registo especificado no barramento interno (Sout) e carrega-o no registo B. De modo a realizar a operação anterior a unidade de controlo gera o sinal "ld_B(0)" (ver Figura 5.2). Quando os 3 bMs do registo de estado (f3) forem diferentes de "000", o operando colocado no barramento em vez de ser carregado em B é carregado no registo da UF (B#), identificado pela descodificação desses 3 *bits*.

A colocação do resultado de uma UF é feita pela instrução **MOVWF** de um modo semelhante. Esta instrução transfere o conteúdo do registo W para um registo especificado na instrução, colocando o valor de W no barramento interno (Sin) ao activar o sinal de controlo "en_W(0)". Com base neste sinal e na descodificação dos 3 bMs do registo de estado é seleccionado qual o operando a colocar no barramento (o conteúdo do registo W ou algum dos operandos resultado da unidade, com o máximo de 8 *bits* por cada operando).

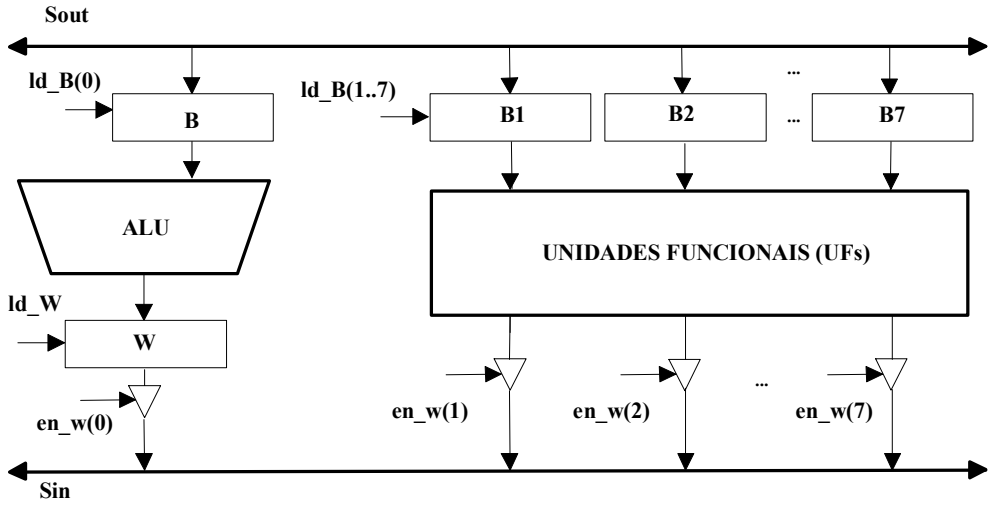


Figura 5.2. Interface de cada unidade ao núcleo do PARMIC.

Para cada transferência de um operando para uma unidade ou de uma unidade para o ficheiro de registos é necessário colocar os 3 *bits* de identificação no registo de estado. Esta operação é realizada por duas instruções (ver Exemplo 5.1 e Exemplo 5.2).

Os registos a transferir têm que ter endereço superior ao endereço do registo de estado (f3) e é permitido o número máximo de 7 operandos de entrada e 7 operandos de saída (8 *bits* cada) para o conjunto de todas as unidades, de forma a que tenham funcionamento exclusivo. O número máximo poderia ser aumentado com um esquema de mapeamento duplo (destinado a identificar cada unidade).

```
MOVLW    identificador do operando na UF
MOVWF    3
MOVF     registo a transferir
```

Exemplo 5.1. Transferência de cada operando do ficheiro de registos para uma unidade funcional.

```
MOVLW    identificador do operando na UF
MOVWF    3
MOVWF    registo a transferir
```

Exemplo 5.2. Transferência de cada operando de uma unidade funcional para o ficheiro de registos.

Para cada transferência de 8 *bits*, do núcleo/UF para UF/núcleo, são necessários 3 ciclos de instrução. O acesso de uma UF aos portos de E/S é também realizado em 3 ciclos de instrução.

5.2.1 Área das Unidades Funcionais

As unidades funcionais consideradas são unidades puramente combinatórias (embora para unidades baseadas em *datapath* + controlo fosse possível a implementação completa ou o controlo ser realizado por *software*) e podem ter execução concorrente.

As UFs são controladas por um circuito de interface cujos sinais, vindos da unidade de controlo e do descodificador, geram os dois sinais que possibilitam o carregamento de um operando do barramento nos registos da UF e a colocação no barramento de um

operando resultado da UF, e podem ser visualizados na Figura 5.3 (para o caso em que são necessários os limites máximos de 7 operandos de entrada e 7 operandos de saída).

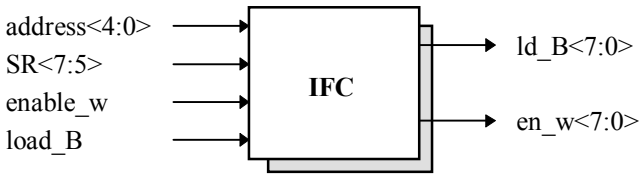


Figura 5.3. Sinais de interface da unidade desmultiplexadora dos sinais de controlo.

A área total da unidade auxiliar é obtida pela soma das áreas de cada um dos circuitos necessários,

$$A_{\text{unidade\#i}} = A_{UF} + A_{INTERFACE} + A_{IFC} \tag{5.1}$$

A área de interface, $A_{INTERFACE}$, corresponde aos registos dos operandos de entrada e *drivers* 3-estados de saída de cada UF, e é estimada por,

$$A_{INTERFACE} \cong \text{num\_operandos\_entrada} \times 110 + \text{num\_operandos\_saida} \times 10 \tag{5.2}$$

A área de uma unidade funcional, $A_{UF,}$ é obtida pela ferramenta de síntese. A área do circuito de controlo dos sinais, A_{IFC}, é apresentada na Tabela 5.1.

Circuito	Área (*sites*)	Nº de células	Atraso (ns)
IFC	173	33	5.63

Tabela 5.1. Área do circuito que desmultiplexa os sinais de interface com as UFs.

5.2.2 Área total do CI

De modo a estimar a área total do CI são avaliadas as áreas de cada uma das unidades que o constituem. A ferramenta calcula a área e indica se ultrapassa a área disponível do agregado lógico pretendido.

A área total do circuito integrado é determinada por:

$$A_{CI} = A_{NÚCLEO} + A_{SW} + \sum_{i} A_{\text{unidade}_i} \tag{5.3}$$

em que a área do *software*, A_{SW}, corresponde à área ocupada pela memória de programa (A_{prog}) e à área de memória de dados (A_{dados}):

$$A_{SW} = A_{prog} + A_{dados} \tag{5.4}$$

A área da memória de programa é estimada, conhecendo o número de instruções do programa, através dos valores representados no gráfico da Figura 4.24 do capítulo anterior. Quando o número de instruções da memória de programa se situa entre dois pontos representados, a área é obtida por interpolação linear entre esses pontos.

A área da memória de dados depende do número e do tipo de registos utilizados (ver Tabela 5.2).

Área (*sites*)	Descrição
80	Área de F0 (interface de um operando nulo ao barramento).
190/(registo de 8 *bits*)	Área relacionada com os registos normais e de saída.
195/(registo de 8 *bits*)	Área para cada porto de entrada.
475/(registo de 8 *bits*)	Área para cada porto de E/S (inclui o registo TRIS).
Determinada com base no gráfico da Figura 4.12.	Área do descodificador de endereços.

Tabela 5.2. Área de dados.

A área do núcleo, obtida pela fórmula (5.5), é constituída pela área do PC e da pilha parametrizável, $A_{PC+Pilha}$, estimada pela fórmula (4.1), e pela restante área que engloba as áreas apresentadas na Tabela 5.3.

$$A_{NÚCLEO} = A_{PC+Pilha} + A_{RESTANTE} \tag{5.5}$$

Área (*sites*)	Descrição	Instanciação
1890	Área da ALU.	fixa
459	Área da unidade de manipulação de *bits*.	condicional
664	Área extra do *datapath*.	fixa
975	Área do descodificador.	fixa
529	Área do registo RTCC.	condicional
379	Área do registo SR.	fixa
113	Área extra.	fixa
777	Área do registo OPTION e temporizador.	condicional
2463	Área da unidade de controlo, máquina mestre, sincronismo e *pipelining*.	fixa

Tabela 5.3. Área restante do núcleo.

5.3 Restrições temporais

A migração de segmentos de código é orientada pela especificação de restrições temporais. A maioria dos sistemas de tempo real necessitam de sincronização entre tarefas, muitas das quais necessitam de ser realizadas num determinado período de tempo para o correcto funcionamento. Na realização deste tipo de sistemas, são habitualmente tidos em conta três tipos de restrições temporais [56]:

- Máximas, em que se considerarmos o início da tarefa no instante de tempo t_i e o final no instante de tempo t_j se requer: $t_j - t_i \leq t_{max}$.

- Mínimas, em que se considerarmos o início da tarefa no instante de tempo t_i e o final no instante de tempo t_j se requer: $t_j - t_i \geq t_{min}$.

- Exactas, em que se considerarmos o fim de uma tarefa no instante de tempo t_i e o início de outra tarefa no instante de tempo t_j se requer: $t_i - t_j = t_{exacto}$.

Para o caso das restrições temporais máximas, sempre que exista violação a única forma possível de resolução dessa mesma violação é a migração do bloco ou tarefa do *software* para o *hardware*. No caso das restrições temporais mínimas, a resolução do problema não passa pela migração do componente, mas sim pelo acréscimo de código,

normalmente NOPs de forma a ser estabelecido o atraso mínimo. O último caso, pode requerer apenas o acréscimo de código, tal como no caso anterior, ou a migração e acréscimo de código de ajuste no caso em que o componente ultrapassa a restrição temporal especificada.

5.4 Tarefas da aplicação BINOMIO

Para tornar automático o processo de migração *hardware*/*software* foi necessário elaborar uma aplicação constituída por vários módulos, ilustrados na Figura 5.4, que em conjunto formam um compilador com dois alvos representativos de dois componentes distintos (o componente *hardware* e o componente *software*), ambos especificados na linguagem de descrição de *hardware*, VHDL.

A aplicação BINOMIO foi realizada em C [57]. As tarefas de análise léxica e sintáctica do ficheiro constituído pelo código *assembler*, da especificação, com directivas foram desenvolvidas utilizando as ferramentas Lex & Yacc [58], [59], disponíveis em ambiente UNIX.

O código *assembler* pode ser automaticamente traduzido para a especificação em VHDL da memória do programa, utilizando um dos assembladores comerciais. Para tal foi realizado um módulo que traduz o código objecto, gerado pelos assembladores, em formato INTELLEC (inhx16 [49]), para o módulo VHDL que infere a memória de programa. Esta tarefa permite a integração no ambiente COSTLES das ferramentas de desenvolvimento para o PIC (existentes comercialmente).

O bloco de extracção de parâmetros do código é feito por analisadores, que percorrem a representação intermédia do código de programa e realizam a determinação dos níveis da pilha, o número de instruções de programa, a necessidade da unidade de manipulação de *bits*, etc.

A extracção dos ciclos e estruturas de controlo condicional permite extrair informação acerca do número de vezes que o código cíclico é efectuado, e efectuar a marcação do ciclo.

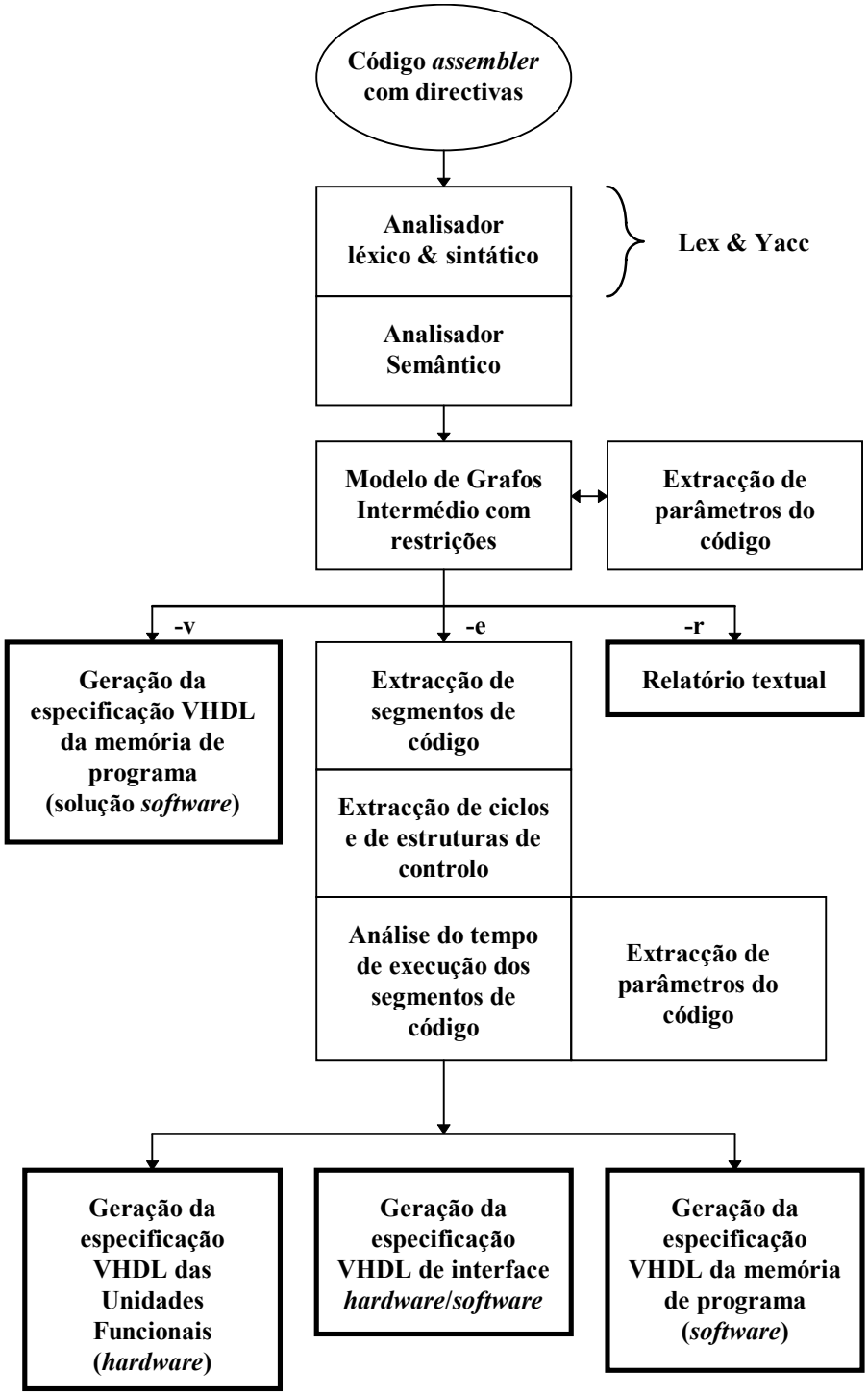


Figura 5.4. Tarefas da aplicação BINOMIO.

Por cada UF é gerada uma entidade VHDL constituída por 3 processos. Dois responsáveis pelo interface aos barramentos internos do PARMIC e o processo obtido pela tradução do segmento de código sob restrições para a correspondente especificação em VHDL.

Nas secções seguintes são descritas as tarefas mais complexas da aplicação BINOMIO.

5.4.1 O Assemblador

O assemblador não suporta definição de áreas de dados (CBLOCK... ENDC do MPASM, que é um meio de controlo da alocação de memória de dados e permite a referência a itens de dados simbolicamente), estruturas alto-nível de controlo condicionais (IFs, FORs, etc.), directivas condicionais de pré-processamento (ex. selecção condicional de blocos de código, `#if`), directiva `#include`, a directiva `#define`, nem macros. Estas directivas-construções de pré-processamento estão disponíveis pela maioria dos assembladores comerciais para o PIC e permitem uma maior clareza de código e facilidade de programação. Contudo, como o principal objectivo não era a realização de um assemblador comercial, mas sim dotar uma plataforma de directivas necessárias para a condução da partição *hardware*/*software*, estas construções não foram consideradas.

A atribuição de um valor a uma determinada etiqueta é efectuada pela directiva **EQU**, em que o valor é definido em decimal (defeito, d'número', ou D'número'), hexadecimal (0xnúmero, h'número', ou H'número'), ou binário (b'número' ou B'número').

O assemblador utiliza um algoritmo de duas passagens necessárias para substituição dos rótulos pelos endereços utilizados pelas instruções de **CALL**, **GOTO** e pela directiva **EQU**.

Caso se opte por uma solução unicamente em *software*, pode ser gerada automaticamente a especificação VHDL, com base no modelo de grafos intermédio, que corresponde à especificação do circuito da memória de programa (opção v do BINOMIO).

O assemblador, em conjunto com a extracção de parâmetros do código, realiza as tarefas de identificação da necessidade da unidade de manipulação de *bits* e de identificação do número de níveis da pilha. No fim gera automaticamente o ficheiro de definição de parâmetros em VHDL para configuração do PARMIC.

A determinação do número de registos necessários não é feita pelo assemblador (existe uma directiva para especificar o número de registos).

5.4.2 Análise do tempo de execução do código

A análise do tempo de execução de um segmento de código é realizada por um simulador que extrai ciclos e que permite determinar o tempo de execução máximo dos segmentos de código acíclicos e de segmentos de código cíclico determinísticos. Para cada segmento de código condicionado pelas directivas de restrição temporal é estimado o tempo de execução para poder ser comparado com a restrição temporal especificada.

Os ciclos e as correspondentes variáveis de contagem são extraídos e analisando as instruções internas é possível determinar o tempo de execução de cada ciclo. O tempo de um ciclo é determinado pela multiplicação do número de vezes que este é executado pelo valor do tempo de execução do código interno a este, que por sua vez pode ter que ser calculado de igual forma sempre que existam ciclos encadeados.

No caso de código com estruturas de controlo condicional é determinado um majorante do tempo de execução, que corresponde ao caminho que percorre o maior número de instruções possível.

5.4.3 Migração do *Software* para *Hardware*

A migração de segmentos de código do *software* para o *hardware* executada pelo BINOMIO é realizada por um conversor de *assembler* para a especificação VHDL sintetizável com a mesma funcionalidade. A especificação VHDL utiliza o módulo de funções e de redefinição de operadores do Apêndice C.

Quando existe uma chamada a uma rotina no segmento de código especificado, o compilador procura no programa se existe alguma chamada a esta rotina. Caso exista é mantida no novo programa, caso contrário é excluída.

O segmento de código que migra é procurado por todo o programa, e no caso de existir o mesmo segmento de código é substituído pela comunicação com a UF (aproveitamento de um recurso disponível).

Os segmentos de código candidatos a migrarem não podem ter instruções de salto, cujo endereço destino seja externo aos segmentos.

Os registos dos operandos de entrada ou de saída de cada UF podem ser mapeados. É apenas necessário que sejam utilizados registos do mesmo banco para cada UF.

Os segmentos de código passíveis de migrarem, devem conter apenas instruções que tenham como operandos registos normais do ficheiro de registos. De cada segmento são extraídos os ciclos com número de iterações definido por um determinado operando cujo valor deve ser conhecido durante a fase de compilação (construções cíclicas inatas, de alto nível, do tipo **FOR**, em que internamente ao ciclo não são atribuídos valores não determinísticos à variável do contador).

No *assembler* do PARMIC, o código cíclico ocorre quando é encontrada uma das sequências de instruções apresentadas na Tabela 5.4. As construções cíclicas 3 e 4 não são suportadas actualmente pelo tradutor de *assembler* para VHDL da aplicação BINOMIO. As construções 1 e 2 para serem representadas pela construção alto nível **FOR** devem ser antecedidas pela colocação no registo identificado por A, na Tabela 5.6, por um valor imediato (instrução **MOVLW** imediato seguida da instrução **MOVW** A).

Tipos de construções cíclicas			
1	2	3	4
{...}	{...}	{...}	{...}
rótulo_1	rótulo_1	rótulo_1	rótulo_1
{...}	{...}	{...}	{...}
DECFSZ A, ?	**INCFSZ** A, ?	**BTFSC** A, ?	**BTFSS** A, ?
GOTO rótulo_1	**GOTO** rótulo_1	**GOTO** rótulo_1	**GOTO** rótulo_1
{...}	{...}	{...}	{...}

Tabela 5.4. Tabela de construções *assembler* que correspondem a ciclos.

Quando o operando que define o número de iterações (designado por A na Tabela 5.4) é conhecido durante a compilação, o tempo de execução do código em questão é bem definido e o componente *software* é habilitado a migrar. Caso contrário, sempre que o número de iterações do ciclo não seja conhecido durante a compilação, e apenas o seja durante a execução do programa, está-se na presença de um segmento de código com execução não determinística, e por isso impossibilitado de migrar. Para este tipo de código a determinação do tempo de execução só poderia ser possível com base em um conjunto de dados de entrada fornecido pelo projectista. O componente *hardware* que implemente a funcionalidade de código não determinístico tem uma estrutura *datapath*+controlo não abordada no âmbito desta tese. Estas estruturas são muito mais complexas, pois além de difícil extracção apresentam diversas soluções possíveis:

- O *datapath* pode ser realizado em *hardware* e o controlo em *software*;
- O *datapath* pode ser realizado em *hardware* e parte do controlo em *software*;
- O *datapath* e o controlo podem ser ambos realizados em *hardware*.

O Exemplo 5.3 apresenta o algoritmo de Euclides para o cálculo do máximo divisor comum (MDC) de dois números representados em 8 *bits* (0..255). Os tempos de execução máximo, mínimo, e médio encontram-se na Tabela 5.5. Este exemplo ilustra o caso de um ciclo não determinístico, com impossibilidade de realização com *hardware* puramente combinatório e, por isso, impossibilitado de migrar para o *hardware* no sistema actual. O número de vezes (iterações) que o corpo do ciclo é executado depende dos valores de xi e de yi (o programa vai actualizando cada umas destas variáveis com a diferença entre elas e salta fora do ciclo quando ambas têm o mesmo valor).

Quando são encontradas as sequências de instruções apresentadas na Tabela 5.6 está-se perante estruturas de controlo condicional do tipo IF. Os tipos A e C correspondem respectivamente à execução da rotina ou da instrução, quando é satisfeita a negação da condição especificada em *assembler*. O tipo B corresponde à execução do bloco até ao rótulo quando o teste da condição especificada é verdadeiro.

```
inicio  ; read xi and yi values
        movf    0x06,w  ;read xi from PORTB
        btfsc   3,2
        goto    fim     ;xi equals zero
        movwf   0x07    ;F7 = xi
        movf    0x06,w  ;read yi from PORTB
        btfsc   3,2
        goto    fim     ;yi equals zero
        movwf   0x08    ;F8 = yi

loop    movf    8,w     ;w = F8 (yi)
        subwf   7,w     ;w = xi - yi
        movwf   9       ;F9 = xi - yi
        movf    9
        btfsc   3,2
        goto    fim  ;xi = yi
        btfsc   3,0
        goto    xiMaior ;xi > yi

xiMenor movf    7,w
        subwf   8   ;yi = yi - xi
        goto    loop

xiMaior movwf   7   ;xi = xi - yi
        goto    loop

fim     movf    7,w
        movwf   0x0A
        end
```

ciclo

Salta fora do ciclo quando xi = yi

Exemplo 5.3. Máximo divisor comum de dois números.

Nº de ciclos (OSC1) para cada iteração (PARMIC)	Tempo de execução para cada iteração (@20MHz)	Nº máximo de iterações	Nº mínimo de iterações	Nº médio de iterações (considerando 1000 pares de valores xi, yi)
30 (xi ≥ yi)	3 μs	254	0	20
33 (xi < yi)	3.3 μs	254	0	20

Tabela 5.5. Tempos de execução do MDC.

Na Tabela 5.7 são apresentadas partes das especificações VHDL que representam as construções para os casos A1, A2, B1, B2, C1, e C3 da Tabela 5.6.

Tipo	1	2	3	4
A	{...} **DECFSZ** A, ? **CALL** rotina_1 {...}	{...} **INCFSZ** A, ? **CALL** rotina_1 {...}	{...} **BTFSC** A, ? **CALL** rotina_1 {...}	{...} **BTFSS** A, ? **CALL** rotina_1 {...}
B	{...} **DECFSZ** A, ? **GOTO** rótulo_1 **{Bloco}** rótulo_1 {...}	{...} **INCFSZ** A, ? **GOTO** rótulo_1 **{Bloco}** rótulo_1 {...}	{...} **BTFSC** A, ? **GOTO** rótulo_1 **{Bloco}** rótulo_1 {...}	{...} **BTFSS** A, ? **GOTO** rótulo_1 **{Bloco}** rótulo_1 {...}
C	{...} **DECFSZ** A, ? instrução {...}	{...} **INCFSZ** A, ? instrução {...}	{...} **BTFSC** A, ? instrução {...}	{...} **BTFSS** A, ? instrução {...}

Tabela 5.6. Tabela de construções *assembler* que correspondem a instruções de controlo condicional.

Tipo	1,2
A	{...} **IF** (F3(2)='0') **THEN** especificação VHDL para a rotina_1 **END IF;** {...}
B	{...} **IF** (F3(2)='1') **THEN** especificação VHDL que corresponde ao Bloco **END IF;** especificação VHDL do *assembler* a partir do rótulo_1 {...}
C	{...} **IF** (F3(2)='0') **THEN** especificação VHDL que corresponde à instrução **END IF;** {...}

Tabela 5.7. Tabela de construções VHDL que correspondem a algumas das construções *assembler* de controlo condicional.

5.4.4 Nível de Granularidade

A migração de todo o segmento de código com restrição temporal produz um componente *hardware* com maior área. O sistema realizado tem como especificação o *assembler*, com um nível de granularidade baixo (próximo do *hardware*), mas, por

enquanto não tira partido desse nível automaticamente, em termos de partição, por migrar todo o segmento. Quando o tempo de execução do componente de *hardware* relativo ao segmento especificado for muito mais baixo do que a restrição temporal, cabe ao projectista forçar o sistema a realizar uma nova migração, definindo a restrição num novo sub-segmento do segmento de código inicial. Este processo acaba por ser iterativo e é uma forma de partição manual.

A Figura 5.5 ilustra o caso em que um determinado segmento de código é especificado com uma restrição temporal máxima excedida pelo tempo de execução. O sistema implementado migra o referido segmento para o *hardware*, caso esta operação permita respeitar a referida restrição temporal, como se ilustra na Figura 5.6. Na Figura 5.7 é apresentado o caso em que a migração era feita utilizando um algoritmo de partição que migraria apenas o código suficiente de modo a respeitar a restrição. Na Figura 5.8 apresenta-se uma situação análoga à anterior, mas para um nível de granularidade mais baixo. Neste caso a área obtida para o componente de *hardware* é menor.

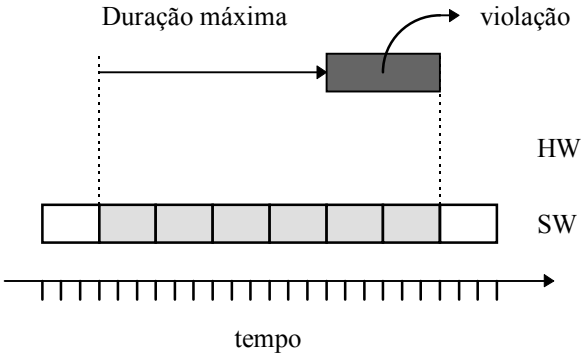


Figura 5.5. Restrição temporal máxima para um segmento de código.

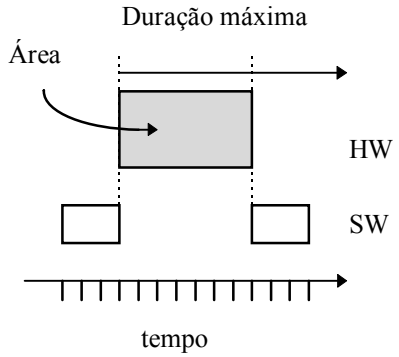


Figura 5.6. Migração de todo o segmento de código para o *hardware*.

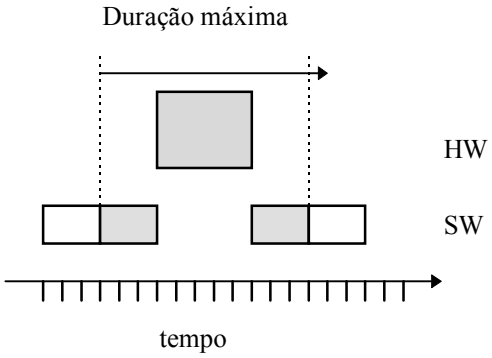


Figura 5.7. Migração de parte do segmento de código para o *hardware*.

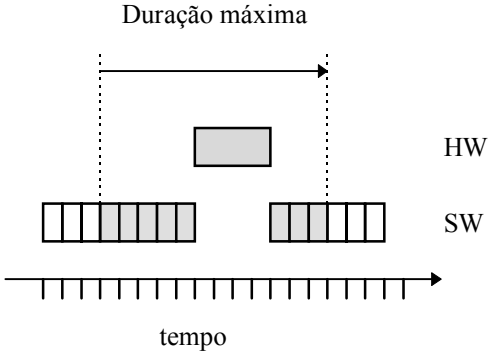


Figura 5.8. Migração de parte do segmento de código utilizando um nível de granularidade mais baixo.

O atraso da transferência de um operando do processador (componente *software*) para uma unidade funcional (componente *hardware*) é importantíssimo. Este atraso desempenha papel tanto mais importante quanto mais fino for o nível de granularidade. A equação (5.6) permite determinar o número de ciclos utilizados para transferir o número de operandos necessários. Para cada transferência, independentemente do sentido com que é feita (do *hardware* para o *software* ou do *software* para o *hardware*) são necessários 3 ciclos de instrução, correspondentes às 3 instruções necessárias já anteriormente referidas. No final da comunicação com a UF devem ser colocados a zero os 3 bMs do registo SR.

$$t_{com} = 3 \times num\_operandos + 1 \quad \text{(ciclos de instrução)} \tag{5.6}$$

5.4.5 Sincronização e comunicação Sw/Hw

Para sincronização do núcleo com a UF são colocados NOPs no novo programa. A colocação destas instruções é automática e para tal é necessário especificar o atraso, obtido do circuito sintetizado, da UF em questão. O projectista pode escolher pela geração de código cíclico que permite a espera pelo resultado da UF sem adicionar um elevado número de NOPs ao programa (preenchendo a memória de programa desnecessariamente), mas deve estar sensibilizado para a adição para contador de um novo registo. Esta solução pode ser também utilizada para satisfazer as restrições de duração temporal mínima ou exacta.

5.4.6 Directivas da aplicação BINOMIO

De modo a que o projectista possa especificar as restrições foram adicionadas ao *assembler* do PARMIC as directivas apresentadas no Apêndice B. A sintaxe das directivas começa por um ponto e vírgula para serem consideradas comentários pelos simuladores e assembladores comerciais existentes. A aplicação BINOMIO não permite encadeamentos de directivas do mesmo tipo.

O paralelismo é assegurado pela especificação estática por parte do projectista de segmentos de código paralelizáveis (que podem ter execução autónoma). Os segmentos especificadas migram para o *hardware* e são executados de forma concorrente. Este mecanismo permite estruturar uma aplicação como uma sequência de actividades independentes. Para cada paralelismo é utilizado um mecanismo muito parecido com o "cobegin - coend". Neste caso o *cobegin* equivale à transferência dos operandos do ficheiro de registos para o segmento implementado em *hardware* e o *coend* à transferência do resultado para o ficheiro de registos. As construções utilizadas para definir o paralelismo permitem uma grande clareza do código e a directiva corresponde ao lançamento automático dos blocos de programa para o *hardware*.

A especificação de um recurso direcciona o sistema para a utilização de um determinado bloco de *hardware* existente numa possível biblioteca de circuitos (multiplicadores, somadores, comparadores, etc). Desta forma é possível direccionar a

migração sem a utilização do conversor *assembler*/VHDL. É necessário que o projectista especifique os registos de entrada e de saída do recurso.

As directivas de atribuição do WDT e do contador/relógio RTCC permitem a instanciação condicional destas unidades no processador.

Existem directivas que permitem especificar o número máximo de portas equivalentes do agregado lógico previsto e o número de sinais disponíveis no encapsulamento.

O Exemplo 5.4 ilustra a restrição de um segmento de código ao tempo de execução máximo de 1μs, de outro segmento ao tempo de execução mínimo de 2 μs, da atribuíção a frequência do relógio do processador de 20MHz e paralelização dos dois segmentos em questão (com execução autónoma).

```
; CONSTRAIN MAXTIME : label1 TO label2 = 1 us
; CONSTRAIN MINTIME : label3 TO label4 = 2 us
; PARALLEL : label1 TO label2 : label3 TO label4
; CLOCK = 20 MHz
```

Exemplo 5.4. Directivas do *assembler*.

5.5 Co-Simulação

A tarefa de so-simulação é realizada pelo simulador de VHDL, ao nível RTL, ou depois de realizada a síntese, por simulações lógicas. A Figura 5.9 ilustra os níveis de co-simulação propostos. A simulação ciclo-a-ciclo só pode ser efectuada parcialmente, pelos simuladores ciclo-a-ciclo comercialmente existentes (ex. MPSIM [49]). Esta simulação é limitada por o simulador não suportar as capacidades de configuração que lhe permitam a simulação do processador configurável, e por não suportar a simulação das UFs. Neste nível a co-simulação é apenas a simulação *software* ciclo-a-ciclo com a arquitectura do PARMIC restrita a elementos da família PIC16C5X.

Os níveis de co-simulação propostos permitem a simulação ao longo de diversos níveis de abstracção, durante os quais o projectista pode verificar o comportamento do sistema.

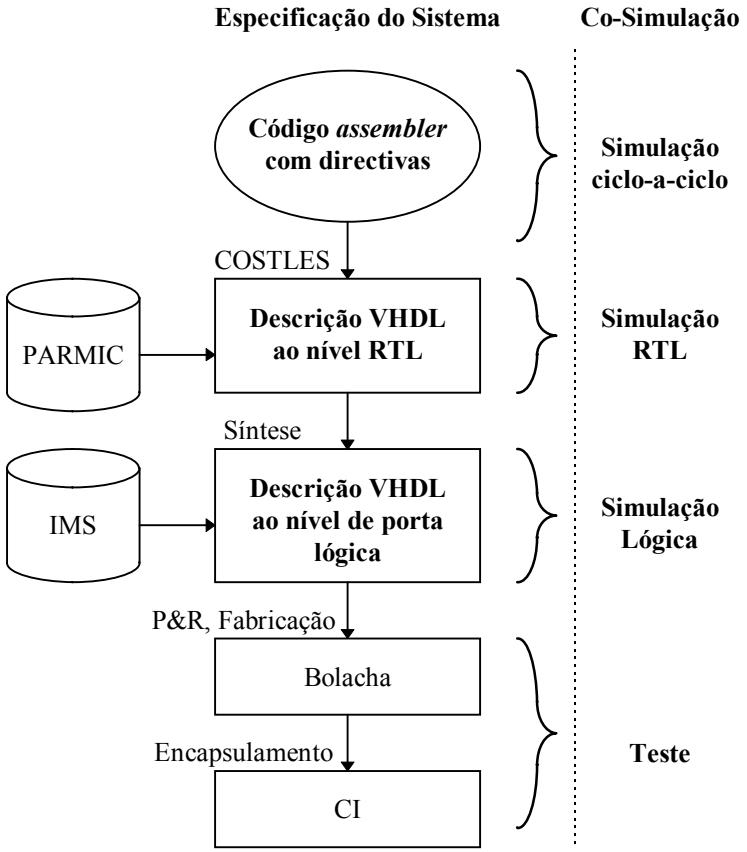


Figura 5.9. Níveis de co-simulação no ambiente de co-síntese COSTLES.

5.6 Exemplo

Os resultados apresentados na Tabela 5.8 referem-se à rotina do Exemplo 5.5 (com 14 instruções + 1 de **CALL**) destinada ao cálculo da multiplicação de dois operandos de 8 *bits* sem sinal, executada no PARMIC (com solução unicamente *software*) e no PIC16C54.

	Nº de ciclos de instrução	Tempo de execução
PARMIC @20MHz	64	9.6 μs
PIC16C54 @20MHz	73	14.6 μs

Tabela 5.8. Tempo de execução para a rotina de multiplicação.

```
{...}
;REGNUM = 13
;WDTimer = TRUE
;Timer = TRUE
;I/O PORTS = 2
;WAFER = 5900
;SIGNALS = 60
;CLOCK = 20 MHz
{...}
;CONSTRAINT  MAXTIME : label1 TO label1 = 4 us
;FU#1 INPUT OPERANDS= f7,f8 : OUTPUT OPERANDS = f9, f10
{...}
;; Rotina de multiplicação
mult        clrf    H_byte
            clrf    L_byte
            movlw   8
            movwf   count
            movf    mulcnd,w
            bcf     STATUS,CARRY
loop        rrf     mulplr
            btfsc   STATUS,CARRY
            addwf   H_byte
            rrf     H_byte
            rrf     L_byte
            decfsz  count
            goto    loop
            retlw   0
{...}
;;       Programa Principal
{...}
main        movf    portb,w
            movwf   mulplr
            movf    portb,w
            movwf   mulcnd
label1
            call  mult
{...}
```

Exemplo 5.5. Parte do código *assembler* que contém a rotina de multiplicação de dois números.

Supondo a necessidade do tempo máximo de execução da multiplicação de 4 μs (violação de +5.6 μs indicada pelo BINOMIO), é especificada a restrição do Exemplo 5.6, e a funcionalidade correspondente ao código da rotina referida é realizada por uma UF específica gerada automaticamente.

```
;CONSTRAINT  MAXTIME : label1 TO label1 = 4 us
;FU#1 INPUT OPERANDS = f7, f8 : OUTPUT OPERANDS = f9, f10
```

Exemplo 5.6. Especificação da restrição temporal com as directivas.

A ferramenta de síntese lógica, com base no código gerado automaticamente pelo BINOMIO (ver Descrição 5.1 e Descrição 5.2) obtém o circuito da unidade referida (área = 2179 *sites* e tempo de execução = 82 ns), com sinais de interface representados na Figura 5.10. O tempo de execução do novo código entre rótulos é de 2.1 μs (mantendo a mesma frequência de relógio). O novo código entre rótulos é constituído por 14 instruções responsáveis pela comunicação e sincronismo com a unidade.

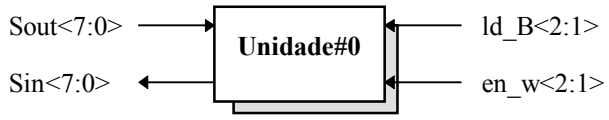


Figura 5.10. Sinais de interface do circuito que contém o interface e a UF.

```
INTERFACE_IN: process(Sin, ld_B)
begin
      IF (ld_B(1) = '1') THEN
            IN7 <= Sin;
      END IF;
      IF (ld_B(2) = '1') THEN
            IN8 <= Sin;
      END IF;
end process;

INTERFACE_OUT: process(enable_W, OUT9, OUT10)
begin
      IF (enable_W(1) = '1') THEN
            Sout <= OUT9;
      ELSIF (enable_W(2) = '1') THEN
            Sout <= OUT10;
      ELSE  Sout <= "ZZZZZZZZ";
      END IF;
end process;
```

Descrição 5.1. Especificação VHDL dos processos de interface da UF aos barramentos.

Para este caso, em que são apenas necessários dois operandos de entrada e dois de saída obtêm-se os resultados apresentados na Tabela 5.9. O multiplicador B foi obtido usando as directivas de defeito da ferramenta de síntese, enquanto que o multiplicador A foi obtido do anterior direccionando a ferramenta de síntese para a "optimização booleana" em conjunto com a restrição de área mínima (A = 0). Neste exemplo optou-se pela unidade com menor área pois possibilitou a satisfação da restrição especificada.

```
process(IN7, IN8)
            VARIABLE W, F3 : std_logic_vector(7 downto 0);
            VARIABLE F2 : std_logic_vector(7 downto 0);
            VARIABLE F2_aux : std_logic_vector(9 downto 0);
{...}
begin
      F7 := IN7;
      F8 := IN8;
       F3 := "00000000";
{...}
       W := "00001000";
       F11 := W;
       F3(2) := allzero(F7);
{...}
      FOR i in 8 downto 1 loop
             F8_aux(7 downto 0) := F3(0) & F8(7 downto 1);
             F3(0) := F8(0);
             F8 := F8_aux(7 downto 0);
            IF (F3(0) = '1') THEN
             F9_aux := W + F9;
             F9 := F9_aux(7 downto 0);
             F3(0) := F9_aux(8);
             F3(1) := F9_aux(9);
             F3(2) := allzero(F9_aux(7 downto 0));
            END IF;
             F9_aux(7 downto 0) := F3(0) & F9(7 downto 1);
             F3(0) := F9(0);
             F9 := F9_aux(7 downto 0);
             F10_aux(7 downto 0) := F3(0) & F10(7 downto 1);
             F3(0) := F10(0);
             F10 := F10_aux(7 downto 0);
      END loop;
       W := "00000000";
      OUT9 <= F9;
      OUT10 <= F10;
end process;
```

Descrição 5.2. Especificação VHDL do processo que corresponde ao segmento de código *assembler* condicionado.

Circuito	Área (*sites*)	Nº de células	Atraso máximo (ns)
multiplicador A	1879	333	79.3
multiplicador B	3275	553	43.32
UF (c/ mult. A)	2179	361	82

Tabela 5.9. Área, número de células, e tempo de propagação para o exemplo do multiplicador.

5.7 Conclusões

Este trabalho propõe um sistema de partição *hardware*/*software* a um nível de granularidade baixa, adequada para o tipo de sistemas embebidos de pequena/média complexidade.

Foi apresentado um ambiente de co-simulação que permite a simulação dos dois componentes ao longo de vários níveis de abstracção e a verificação, por simulação, da aplicação desenvolvida no ambiente proposto. O futuro desenvolvimento de um simulador ciclo-a-ciclo do PARMIC, que permita a simulação integrada das unidades que migram para o *hardware* tornará possível a co-simulação a este nível, e por isso facilitará o projecto e a verificação da funcionalidade. O simulador terá que permitir a visualização do novo código *assembler* gerado pelo BINOMIO (realização de um desassemblador).

A restrição de um determinado segmento de código deve ser sempre acompanhada pela especificação dos operandos de entrada e de saída relativos ao segmento em causa. Este problema poderia ser resolvido pela análise do código.

A parametrização do número de *bits* de cada operando de interface às unidades funcionais deverá ser um aspecto a ter em conta, pois permite a redução da área do CI.

Futuros melhoramentos devem incluir a adição de uma directiva que permita a parametrização automática dos *bits* de um registo que deverão ser de entrada/saída (E/S), de saída, ou de entrada.

O número máximo de operandos de entrada e de saída para as unidades funcionais é de 7 mantendo a identificação pelos 3 bMs do registo f3. Contudo para aplicações com maiores necessidades seria possível utilizar outro tipo de mapeamento, utilizando uma estrutura do tipo de banco de registos, ou outras formas de interface.

A comunicação do PARMIC a uma UF pode ser reduzida de 2 instruções (2 ciclos de instrução) para o caso do funcionamento apenas da UF enquanto o processador espera. Nesta situação a colocação do primeiro operando resultado no barramento

pode utilizar o mapeamento utilizado pelo carregamento do último operando de entrada.

A exploração de partições é conduzida por intervenção do projectista. A solução automática passa pela implementação de um algoritmo ou heurística de partição que permita explorar a migração de forma a respeitar a restrição com um mínimo de área.

O número de registos necessários, especificado pelo projectista, pode ser determinado automaticamente (sem necessidade da directiva). Após a migração de segmentos de código pode ser necessário haver o re-escalonamento dos registos necessários (a migração pode originar que registos anteriores passem a não serem utilizados). Da mesma forma a identificação dos operandos que comunicam com uma UF deveria ser realizada automaticamente (sem utilização da directiva que permite especificar os operandos de entrada e os operandos de saída para cada UF).

6. Exemplo de Aplicação

Neste capítulo é apresentado o projecto de um filtro digital realizado no ambiente de co-síntese desenvolvido neste trabalho. O desempenho pretendido para o filtro não é possível com uma solução puramente *software*, pelo que foi necessário migrar para o *hardware* um segmento de código de forma a respeitar as restrições impostas.

6.1 Filtro Digital IIR

O filtro digital projectado é um filtro IIR[1], passa-baixo de quarta ordem, realizado com base na cadeia de duas estruturas do tipo da representada na Figura 6.1.

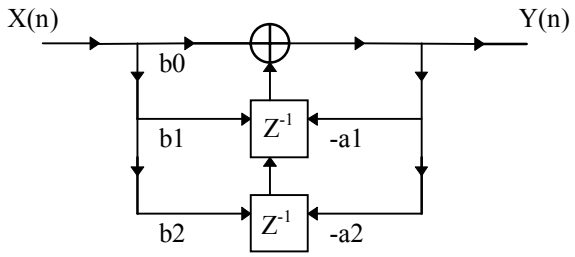


Figura 6.1. Forma de segunda ordem.

A especificação utilizada tomou como base a aplicação apresentada pela Microchip™ [2], realizada com um microcontrolador de alto desempenho da família PIC (PIC17C42 [2]).

[1] Do Inglês ***I**nfinite **I**mpulse **R**esponse.*

Cada amostra do sinal de entrada ($X_{(n)}$) é quantificada por 16 *bits*. Para representar os valores numéricos das variáveis e amostras do sinal de entrada foi utilizado o formato Q15 (com 15 pontos decimais e em que o bMs é o *bit* de sinal). Tal como na aplicação original as rotinas utilizadas não incluem o tratamento de *overflows*.

Por cada amostra do sinal de entrada são realizadas: 8 chamadas a uma rotina que calcula o produto com 32 *bits* de dois números de 16 *bits* com sinal; 6 chamadas a uma rotina de adição de dois números de 16 *bits*.

Pretende-se que a frequência de amostragem permitida seja no mínimo de 10 KHz.

6.1.1 Resultados

Foi adaptado um programa para o PARMIC que permite concretizar a funcionalidade pretendida para o filtro. A Tabela 6.1 apresenta os resultados da aplicação original (solução A), do programa implementado no PIC16C57 executado à frequência de relógio de 16 MHz e máxima (soluções B e C), e no PARMIC executado também à frequência de relógio de 16 MHz e máxima (soluções D e E). Para a aplicação original é possível uma frequência de amostragem máxima de 2.55 KHz. As soluções B e C permitem frequências de amostragem de 1.43 e 1.79 KHz respectivamente.

As soluções D e E apresentam os resultados obtidos com o PARMIC para duas frequências de relógio consideradas. Como seria de prever o PARMIC, mesmo com a mesma frequência de relógio que a família PIC apresenta melhor desempenho. Contudo tanto a solução D como a solução E (frequências de amostragem de 2.01 KHz e 4.40 KHz respectivamente), em que é utilizada a frequência de relógio máxima para o PARMIC, não permitem satisfazer a frequência de amostragem pretendida.

O maior número de ciclos de execução do código realizado em relação à aplicação proposta em [2], deve-se ao facto de não se encontrar optimizado em termos de tempo de execução e à necessidade de utilização de 3 bancos de registos mapeados, pois o número de registos necessário excede os 32 registos base (para os quais não é necessário a identificação mapeada). A rotina de multiplicação é baseada em código cíclico e poderia ser optimizada em cerca de 100 ciclos se tivesse sido programada

com código acíclico. Contudo, esta solução aumentaria em cerca de 200 o número de instruções.

Solução	Microcontrolador utilizado	Nº de ciclos de instrução	Tempo de execução (μs)	Frequência de amostragem máxima (KHz)	Tamanho da memória de programa (nº de instruções)	Nº de registos
A[2]	PIC17C42 @16 MHz	1566	391.5	2.55	426	48+8
B	PIC16C57 @16 MHz	2792	698.0	1.43	309	56
C	PIC16C57 @20 MHz	2792	558.4	1.79	309	56
D	PARMIC@16 MHz	2651	497.06	2.01	309	56
E	PARMIC@35 MHz	2651	227.22	4.40	309	56

Tabela 6.1. Resultados da implementação do filtro digital de quarta ordem.

A resposta impulsional, do filtro implementado, encontra-se ilustrada na Figura 6.2.

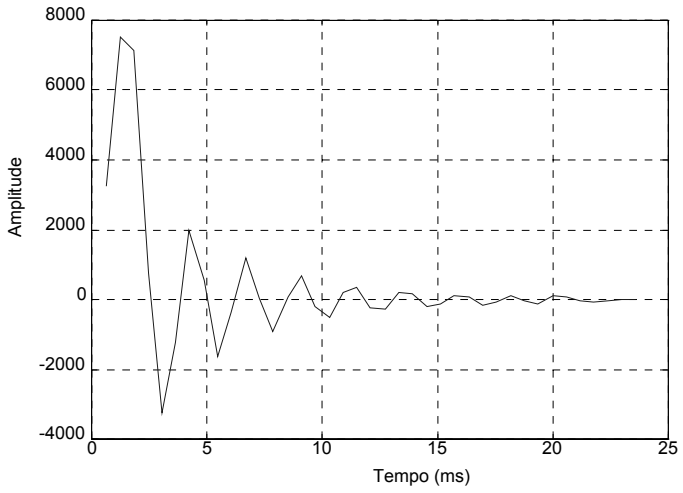


Figura 6.2. Resposta impulsional do filtro implementado (PIC16C54 @16MHz).

Como se pode observar pelos exemplos anteriores é impossível satisfazer os requisitos da aplicação com a família PIC ou com o PARMIC. Foram então incluídas as directivas de restrição temporal na especificação e utilizada a opção da ferramenta BINOMIO, inserida no ambiente de co-síntese COSTLES, que permite a migração

[2] Resultados retirados de [2].

automática dos segmentos de código condicionados para *hardware*, caso haja violação das restrições temporais. No Apêndice E são apresentados o código do programa com restrições e os ficheiros gerados automaticamente pela ferramenta.

A exploração da migração de sub-segmentos do segmento rotulado é realizada pela interacção do utilizador com a ferramenta. Pela análise do código, é possível observar que os segmentos com maior tempo de execução se referem à multiplicação de 16 *bits* em complemento para dois que é utilizada 8 vezes por cada amostra do sinal de entrada. Este segmento de código é o único código cíclico da rotina do filtro, e por isso o que melhores resultados pode originar se for migrado para *hardware*.

Ao especificar-se como restrição temporal máxima 0 μs (opção que força a migração para o *hardware* do segmento de código especificado) o segmento de código especificado é traduzido automaticamente para a especificação VHDL de uma unidade correspondente, incluindo a interface com o PARMIC. A Tabela 6.2 apresenta os resultados depois de realizada a síntese do circuito e a Figura 6.3 apresenta a simulação da UF, para os operandos -32764 e 3841. A unidade inclui os registos (*latchs*) dos operandos de entrada e os *drivers* ao barramento.

	Área (*sites*)	Nº de células	Atraso máximo (ns)
Síntese defeito	18762	3113	171.65
Optimização da área	14579	2574	185.67

Tabela 6.2. Resultados da UF sintetizada.

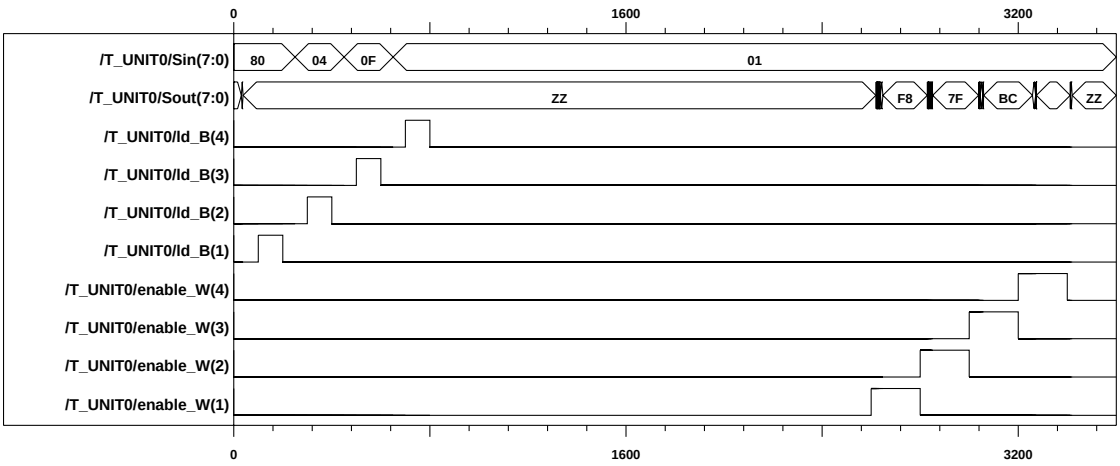


Figura 6.3. Simulação lógica da UF gerada automaticamente que corresponde a um multiplicador de 16 *bits* em complemento para dois.

Na Tabela 6.3 são apresentados os resultados obtidos para três frequências de relógio do novo programa gerado pelo BINOMIO. Nos casos F e G é apenas necessário um **NOP** de espera para que o resultado da UF esteja concluído, enquanto que nos casos H e I são necessários 2 NOPs. Estes NOPs são inseridos automaticamente pelo BINOMIO, desde que lhe seja indicado o atraso da UF correspondente.

O caso F ilustra o resultado obtido quando a restrição é rotulada no código principal do filtro quando da chamada da rotina de multiplicação. Neste caso a ferramenta substitui a chamada pela comunicação e sincronismo com a UF e detecta se existe o mesmo segmento de código ao longo do programa. Como existem 8 chamadas à referida rotina insere comunicação e sincronismo 8 vezes, daí o maior tamanho de memória de programa. Nos restantes casos, a restrição foi rotulada dentro da rotina originando que o código de comunicação e sincronismo fique dentro desta. Esta solução, para o caso considerado, requer uma memória de programa menor embora o tempo de execução seja maior (+ 2 instruções por cada acesso à UF).

Solução	Microcontrolador utilizado	Nº de ciclos de instrução	Tempo de execução (μs)	Frequência de amostragem máxima (KHz)	Tamanho da memória de programa (nº de instruções)
F	PARMIC @16MHz	459	86.06	11.61	464
G	PARMIC @16MHz	475	89.06	11.22	289
H	PARMIC @20MHz	483	72.45	13.80	290
I	PARMIC @35MHz	483	41.4	24.15	290

Tabela 6.3. Resultados da implementação do filtro digital de quarta ordem pelo ambiente de co-síntese COSTLES.

Na Tabela 6.4 são apresentadas as estimativas da área do CI obtidas pelo BINOMIO, e as áreas reais obtidas pela ferramenta de síntese. É apresentada a solução puramente *software* e a solução *software*/*hardware* e como exemplo comparativo o resultado obtido caso se utilizasse um PARMIC sem capacidades de parametrização (com 80 registos, WDT, RTCC, 3 portos de E/S, e pilha de 2 níveis).

A Figura 6.4 e a Figura 6.5 ilustram os resultados obtidos pela simulação funcional da solução D e da solução F respectivamente. Em ambas são apresentados os instantes em que o programa retorna pela primeira vez da rotina do filtro, apresentando o primeiro valor calculado da resposta impulsional (0x0CAC = 3244).

Solução @16 MHz	Nº de registos	Tamanho da memória de Programa (nº de instruções)	Área estimada (*sites*)	Área real (*sites*)
PARMIC	56	309	23639	22939
PARMIC + UF	56	289	38488	37700
PARMIC sem parametrização	80	512	33082	-

Tabela 6.4. Área total do CI.

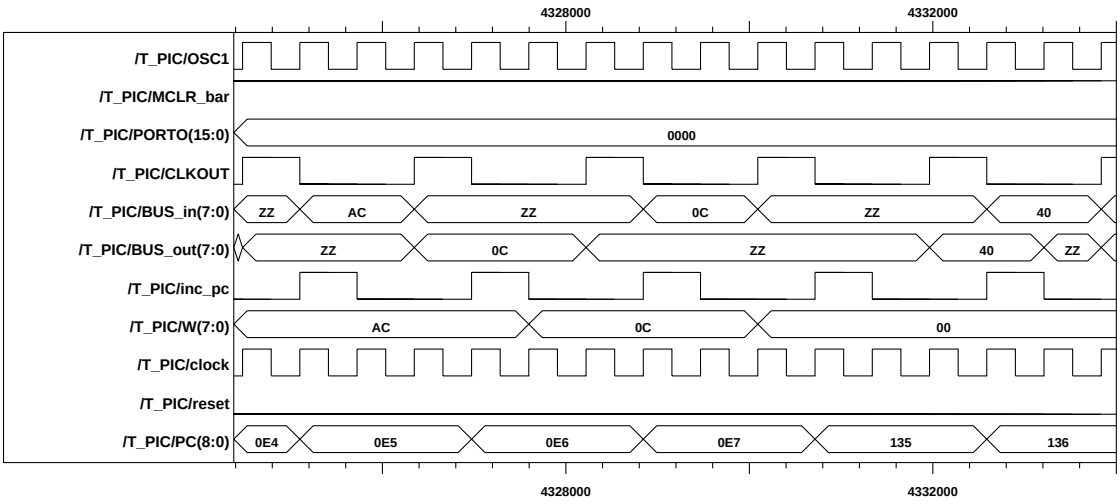


Figura 6.4. Simulação funcional da solução D.

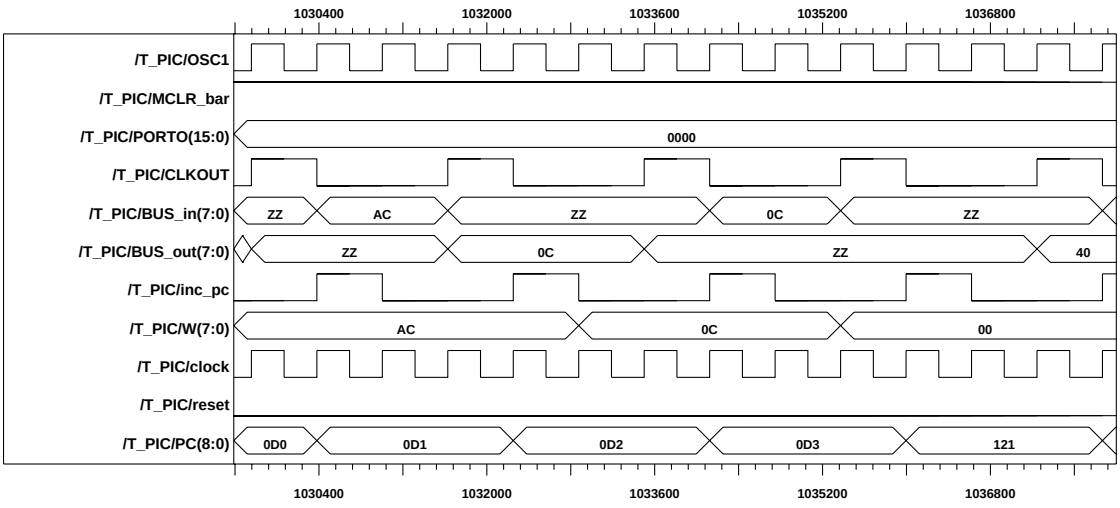


Figura 6.5. Simulação funcional da solução F.

6.2 Conclusões

Neste capítulo descreveram-se os resultados obtidos na realização de um filtro digital, para o qual os requisitos obrigavam a uma frequência de amostragem impossível de respeitar utilizando uma solução puramente *software* com o microcontrolador PARMIC, ou qualquer dos microcontroladores da família PIC. Para tal foi apresentada a solução realizada no ambiente de co-síntese, que permitiu, através da adição ao PARMIC de uma unidade de *hardware*, a satisfação dos requisitos de desempenho pretendidos.

O exemplo ilustra a maior vantagem do sistema de co-síntese apresentado. O projectista não necessita de utilizar um microcontrolador mais potente, ou de maior dimensão, para conseguir o desempenho pretendido pela aplicação. Mantendo o mesmo conjunto de instruções e a mesma frequência de relógio, foi possível satisfazer os requisitos temporais da especificação, com menor tempo de projecto e solução próxima do óptimo (em termos de dimensionamento).

Para o caso de restrições mais severas seria possível migrar uma porção maior de código, que poderia ser, no caso extremo, a rotina completa do filtro.

Por enquanto o projecto de sistemas embebidos digitais de pequena/média complexidade no ambiente COSTLES pode requerer várias iterações até ser encontrada a solução que satisfaça as restrições e que produza resultados próximos do óptimo. Todavia a realização de iterações necessita de pouco tempo de projecto devido aos automatismos da ferramenta.

Sem a integração no ambiente COSTLES de um simulador ciclo-a-ciclo com as capacidades de parametrização do PARMIC o projecto a este nível é extremamente complexo, e impossível de verificar para extensões do processador incompatíveis com os elementos da família PIC.

A obtenção automática de unidades funcionais, sempre que a solução menos dispendiosa não satisfaça os requisitos temporais impostos pelo projectista, permite a redução do tempo de projecto e torna possível a exploração de soluções de forma a obter-se uma solução final não sobredimensionada.

7. Conclusões e Trabalho Futuro

7.1 Conclusões

Neste trabalho foi proposto um ambiente de co-síntese (COSTLES) que integra uma aplicação computacional, designada por BINOMIO, responsável pelas fases automáticas de partição *software/hardware* do ambiente de co-síntese. Para implementar o componente *software* foi especificado em VHDL um microprocessador parametrizável designado por PARMIC. O BINOMIO fornece as especificações VHDL sintetizáveis do módulo de parametrização, das unidades funcionais e respectivo interface com o núcleo, e da memória de programa.

Esta nova solução para co-síntese de sistemas embebidos de pequena/média complexidade permite realizar aplicações críticas, sem a utilização de processadores de alto desempenho e/ou de mais CIs, e sem esbanjamento desnecessário de silício. A arquitectura alvo apresentada, com implementação em agregados de células lógicas, permite um ciclo de projecto rápido desde a exploração do binómio *hardware/software* até à prototipagem, apresenta menores custos de projecto do que soluções *semi-custom*, é economicamente viável para fabricação de pequenas séries de dispositivos, e é vocacionada para aplicações sem utilização intensiva de memória de dados.

Embora se tenha partido da especificação do sistema a realizar em código *assembler*, as ideias descritas e desenvolvidas neste trabalho permitem demonstrar que a partição a este nível produz resultados próximos do óptimo em aplicações de pequeno/médio porte, para as quais o desperdício de silício se torna um factor a ter em conta.

Por vezes, a tarefa para a qual o projectista necessita de impor restrições temporais é um segmento de código grande o que implica, na solução actual, a migração de todo o código definido caso haja violação da restrição. No presente sistema cabe ao projectista, nesta situação, impor restrições internas à tarefa (normalmente a segmentos de código computacionalmente intensivos, como são os casos dos ciclos), em vez de optar pela restrição global.

Os segmentos de código cujo tempo de execução depende do valor, não conhecido, de um ou mais operandos são afastados da implementação automática em *hardware*. Estes segmentos requerem soluções constituídas por *controlo+datapath*. O projectista pode realizar manualmente estas unidades, ou caso forneça uma boa solução, impôr restrições a segmentos de código internos a ciclos não determinísticos. Esta última solução implica a iteração realizada em *hardware*, mas controlada pelo *software*.

Embora nem todas as construções de código *assembler* sejam permitidas pelo tradutor da especificação em *assembler* para a correspondente especificação em VHDL, nos exemplos considerados não houve necessidade de reescrita da especificação inicial.

A extracção de segmentos para o *hardware* pode também ser uma boa alternativa para a redução do consumo de potência, ao permitir a redução da frequência do relógio do núcleo.

O processador que constitui a unidade central da arquitectura do CI apresenta melhor desempenho do que os elementos da família PIC correspondentes, embora tenha sido projectado com a preocupação de minimizar a área ocupada, desde que o desempenho não fosse inferior ao da família PIC. O processador PARMIC realizado retira um ciclo de relógio ao ciclo de instrução dos PIC e todas as instruções são efectuadas em apenas um ciclo de instrução, à excepção de casos particulares como são os casos da escrita no PC e dos saltos condicionais.

7.2 Investigação e Desenvolvimentos Futuros

Nesta secção são sugeridos alguns desenvolvimentos necessários para que o ambiente seja melhorado e possam ser adicionadas novas funcionalidades. Surgem duas linhas de

investigação e desenvolvimento. Uma refere-se ao desenvolvimento de ferramentas computacionais que permitam a especificação do sistema por uma linguagem de alto nível, por exemplo. Outra linha prende-se com a adição de novas facilidades de parametrização e melhoramentos do microprocessador.

7.2.1 Ambiente de Co-Síntese

Para auxiliar o projecto no ambiente proposto, torna-se de grande importância a implementação de um simulador ciclo-a-ciclo configurável de acordo com a parametrização do PARMIC, e capaz de simular ciclo-a-ciclo as soluções mistas *hardware*/*software*. Este simulador deverá eventualmente realizar a simulação do *hardware* específico através do modelo *software* disponível. Contudo, nesta fase de projecto, será necessário obter estimativas do tempo de atraso do *hardware* correspondente a cada unidade funcional.

A adição de directivas que definam tabelas de constantes realizadas numa unidade funcional sob a forma de ROM, pode ser bastante útil, sempre que seja necessária a utilização de um elevado número de constantes. Desta forma não seria necessário utilizar a memória de programa para declarar tabelas de constantes.

Ao nível da partição é necessário que o sistema possa explorar automaticamente os blocos de código a migrar internos a um segmento com restrições. Para isso deveria ser realizado um algoritmo de partição baseado numa função de custo, tal como na maioria dos sistema de co-síntese descritos. O processo de partição *hardware*/*software* deve explorar este binómio entrando em linha de conta com a restrição de área permitida pelo agregado escolhido.

A migração de ciclos de código não determinísticos é um aspecto a ter em conta. Como solução poderia ser integrada no ambiente COSTLES uma ferramenta de síntese de alto nível.

A comunicação *hardware*/*software* tem um papel fundamental na partição, principalmente quando se trabalha com níveis de granularidade baixos (porque quanto mais baixo é o nível de granularidade maior é o efeito do atraso da comunicação *hardware*/*software*). Por isso devem ser investigados novos esquemas de interface que

permitam reduzir o tempo de comunicação entre as UFs e o processador. Uma possível solução poderá ser a integração das UFs como registos especiais do ficheiro de registos (deste modo a transferência de um operando seria feita em 2 ciclos e o número de operandos de entrada e de saída para as UFs era limitado pelo número máximo de registos no ficheiro de registos).

É necessário, do ponto de vista de eficiência e redução do tempo de projecto, que a especificação do *software* seja feita numa linguagem de programação de alto nível, que poderá ser uma extensão do C ou do C++, que permita manipulação de *bits* individuais e que englobe as directivas de projecto apresentadas. Este compilador poderia gerar o código *assembler* e as directivas para que a partição pudesse utilizar o nível de granularidade de instrução ou realizar a partição a um nível mais elevado, neste caso seria necessário um tradutor de C ou C++ para VHDL. Outra solução aponta para que a especificação possa ser feita por um modelo gráfico, em vez das tradicionais linguagens textuais, no qual seria possível especificar as restrições. Contudo, existe um conjunto de representações gráficas (SDF, Redes de Petri, etc.) que dependem do tipo de sistema a implementar e por isso restringem o tipo de aplicação a que o sistema se destina.

7.2.2 Núcleo de processamento

O próximo passo de melhoramento da arquitectura do PARMIC poderá ser o aumento do limite máximo da memória de programa, pois a implementação actual apresenta severas restrições do tamanho desta memória, e a adição de linhas de interrupção, fundamentais para a maioria dos sistemas embebidos.

Poderá ser permitida a extensão a interfaces de memória de dados ou periféricos dentro do CI, embora externas à unidade central (a comunicação poderia ser feita por portos do ficheiro de registos).

Deverão ser estudadas formas de aumentar a testabilidade do núcleo, que com a inserção de cadeias de varrimento e modos de teste, se situa em apenas 26%.

Devem ser centrados esforços no desenvolvimento de arquitecturas completamente configuráveis, que permitam a parametrização do comprimento de palavra de acordo

com a aplicação alvo, por exemplo. A arquitectura do PARMIC poderá ser um bom ponto de partida devido à sua grande flexibilidade.

Apêndice A

A aplicação BINOMIO

A aplicação BINOMIO (*From Software-**B**ased Spec**I**ficatio**N** to Software/Hardware Transf**O**r**M**ation in the* COSTLES *Env**I**r**O**nment*), inserida no ambiente COSTLES, é invocada na linha de comando do sistema operativo, com a seguinte sintaxe:

```
binomio [-rve] <ficheiro de entrada> <ficheiro de saída>
```

O ficheiro de entrada deve conter o programa em *assembler* com as directivas. Os erros encontrados no código fonte são indicados e apresentada a linha onde ocorreu o erro. A aplicação aceita as opções apresentadas na Tabela Apêndice A .1.

Opção	Descrição
-r	É criado o relatório do programa no ficheiro de saída. O relatório apresenta dados referentes ao programa de entrada.
-v	Cria: (1) O ficheiro "param.vhd" que contém as variáveis VHDL de parametrização para o PARMIC. (2) O ficheiro especificado como ficheiro de saída que contém a descrição VHDL da memória de programa.
-e	Cria: (1) O ficheiro "param.vhd" que contém as variáveis VHDL de parametrização para o PARMIC. (2) O ficheiro especificado como ficheiro de saída que contém a descrição VHDL da memória do novo programa. Este novo programa contém a comunicação com a(s) UF(s). (3) Os ficheiros "unit#.vhd" em que cada um contém a descrição VHDL da correspondente UF e do interface ao PARMIC.

Tabela Apêndice A .1. Opções da aplicação BINOMIO.

Apêndice B

Directivas do BINOMIO

A aplicação BINOMIO suporta 15 tipos de directivas, encapsuladas no *assembler* do PARMIC, com a seguinte sintaxe:

- **Directiva para restrições temporais:**

```
;CONSTRAINT <restrição temporal> : [<rótulo> <TO> <rótulo>] = <tempo
de execução> [<unidade>] [;;<comentário>]

<restrição temporal>    : MAXTIME | MINTIME | LATENCY

<unidade>               : NS | MS | US
```

- **Directiva de paralelismo:**

```
;PARALLEL : <rótulo> <TO> <rótulo> : <rótulo> <TO> <rótulo> {:
<rótulo> <TO> <rótulo>} [;;<comentário>]
```

- **Directivas de especificação do período do relógio:**

```
;CLOCK = <frequência do relógio OSC1> [<unidade1>] [;;<comentário>]

<unidade1>  : MHz | KHz
```

- **Especificação de um recurso de *hardware*:**

```
;RESOURCE <rótulo> TO <rótulo> : <nome do recurso> [;;<comentário>]
```

- **Especificação do número de registos:**

```
;REGNUM = <número de registos> [;;<comentário>]
```

- **Especificação do número de portos de E/S (F5..F8):**

```
;I/O PORTS = <número de portos de E/S (defeito = 2, máximo = 4)>
[;;<comentário>]
```

- **Especificação dos portos de Saída:**

```
;O PORTS = {F | f} <número do registo> {, {F | f} <número do registo>
} [;;<comentário>]
```

- **Especificação dos portos de Entrada:**

```
;I PORTS = {F | f} <número do registo> {, {F | f} <número do registo>
} [;;<comentário>]
```

- **Especificação dos operandos de uma unidade funcional:**

```
;FU INPUT OPERANDS = {F | f} <número do registo> {, {F | f} <número
do registo>} : OUTPUT OPERANDS = {F | f} <número do registo> {, {F |
f} <número do registo>} [;;<comentário>]
```

- **Declaração de WDT:**

```
;WDTimer = TRUE | FALSE [;;<comentário>]
```

- **Declaração de contador/relógio em tempo real RTCC:**

```
;TIMER = TRUE | FALSE [;;<comentário>]
```

- **Especificação do número máximo de portas equivalentes da bolacha:**

```
;WAFER = <número de portas equivalentes> [;;<comentário>]
```

- **Especificação do número máximo de sinais do encapsulamento:**

```
;SIGNALS = <número de sinais> [;;<comentário>]
```

- **Directiva genérica de atribuição:**

```
<rótulo> EQU <identificador> [;;<comentário>]
```

- **Directiva genérica de colocação em determinado endereço de programa:**

```
ORG <endereço de memória> | <rótulo> [;;<comentário>]
```

Apêndice C

Módulo de funções VHDL

Neste apêndice é apresentado o módulo de funções em VHDL utilizado no projecto do microprocessador e utilizado pela ferramenta de tradução do *assembler* em VHDL. Este módulo é constituído pela redefinição dos operadores de soma e subtracção, pelas funções de incremento e decremento, pelas funções de conversão de um inteiro para uma palavra binária e vice-versa, e pela função que detecta se uma palavra binária é igual a zero.

```
library IEEE;
use IEEE.std_logic_1164.all;

package aritmetica is
      function int2bits(I: natural; out_length: natural) return
      std_logic_vector;
      function bits2int(A: std_logic_vector) return natural;
      function "+"(A,B: std_logic_vector(7 downto 0)) return
      std_logic_vector;
      function "-"(A,B: std_logic_vector(7 downto 0)) return
      std_logic_vector;
      function inc(A: std_logic_vector(7 downto 0)) return
      std_logic_vector;
      function dec(A: std_logic_vector(7 downto 0)) return
      std_logic_vector;
      function allzero(A: std_logic_vector(7 downto 0)) return
      std_logic;
end aritmetica;

package body aritmetica is

      function bits2int(A: std_logic_vector) return natural is
```

```
  variable ad, j: natural;
  begin
      ad := 0;
      j := 0;
          for i in A'right to A'left loop
                  if A(i) = '1' then
                          ad := 2**j + ad;
                  end if;
            j := j + 1;
          end loop;
          return(ad);
  end bits2int;

function int2bits(I: natural; out_length: natural) return
std_logic_vector is
  variable A: std_logic_vector(out_length-1 downto 0);
  variable I1: natural;
  begin
      A := (others => '0');
          I1 := I;
          for n in 0 to A'left loop
                  if (I1 mod 2) = 0 then
                          A(n) := '0';
                  else A(n) := '1';
                  end if;
                  I1 := I1/2;
          end loop;
          return(A);
  end int2bits;

function "+"(A,B: std_logic_vector(7 downto 0)) return
std_logic_vector is
      variable S: std_logic_vector(9 downto 0);
      variable carry: std_logic;
begin
      carry := '0';
      for i in 0 to 3 loop
            s(i) := a(i) xor b(i) xor carry;
            carry := ((a(i) and b(i)) or (a(i) and carry) or
            (b(i) and carry));
      end loop;
      S(9) := carry;
      for i in 4 to 7 loop
            s(i) := a(i) xor b(i) xor carry;
            carry := ((a(i) and b(i)) or (a(i) and carry) or
            (b(i) and carry));
      end loop;
      S(8) := carry;
      return(S);
end "+";

function "-"(A,B: std_logic_vector(7 downto 0)) return
std_logic_vector is
      variable S: std_logic_vector(9 downto 0);
      variable B1: std_logic_vector(7 downto 0);
      variable carry: std_logic;
begin
      B1 := not(B);
```

```
            carry := '1';
            for i in 0 to 3 loop
                  s(i) := a(i) xor b1(i) xor carry;
                  carry := ((a(i) and b1(i)) or (a(i) and carry) or
                  (b1(i) and carry));
            end loop;
            S(9) := carry;
            for i in 4 to 7 loop
                  s(i) := a(i) xor b1(i) xor carry;
                  carry := ((a(i) and b1(i)) or (a(i) and carry) or
                  (b1(i) and carry));
            end loop;
            S(8) := carry;
            return(S);
      end "-";

      function inc(A: std_logic_vector(7 downto 0)) return
      std_logic_vector is
      variable S: std_logic_vector(7 downto 0);
      variable carry: std_logic;
      begin
            carry := '1';
            for i in 0 to 7 loop
                  s(i) := a(i) xor carry;
                  carry := a(i) and carry;
            end loop;
            return(S);
      end inc;

      function dec(A: std_logic_vector(7 downto 0)) return
      std_logic_vector is
      variable S: std_logic_vector(7 downto 0);
      variable carry: std_logic;
        begin
            carry := '0';
            B1:="11111111";
            for i in 0 to 7 loop
               s(i) := not(a(i)) xor carry;
               carry := a(i) or carry;
            end loop;
            return(S);
       end dec;

      function allzero(A: std_logic_vector(7 downto 0)) return
      std_logic is
      variable Z: std_logic;
      begin
            if (A = "00000000") then
                Z := '1';
            else
                Z := '0';
            end if;
            return(Z);
      end allzero;

end aritmetica;
```

Apêndice D

Código do Exemplo

Neste apêndice apresenta-se o código *assembler* do exemplo apresentado no capítulo 6. É apresentado parte do conteúdo do ficheiro que descreve a memória de programa, para a solução puramente *software* (secção D.2) e para a solução mista (D.4).

Na secção D.3 é apresentado o módulo de definição dos parâmetros de configuração do PARMIC gerado pela aplicação BINOMIO.

Na secção D.5 é apresentado o código VHDL gerado automaticamente pelo BINOMIO para a UF e interface ao PARMIC do exemplo, e na última secção (D.6) são apresentadas as mensagens originadas pelo BINOMIO quando realiza a partição.

D.1 Código *assembler* do filtro IIR

```
;;*****************************************************************
;;          Directivas:
;;*****************************************************************

;WAFER = 5900      ;; Number of maximum equivalent gates of the wafer
;SIGNALS = 60      ;; Number of Signal Pins
;CLOCK = 16 MHz    ;; Frequency of the OSC1 clock
;REGNUM = 56

;I/O PORTS = 0           ;; Number of the I/O ports
;I PORTS = F5, F6        ;; Specification of the Input ports
;O PORTS = 0             ;; Specification of the Output ports
```

```
;WDTimer = false        ;; Existance of the WDT
;TIMER = false          ;; Existance of the Timer (RTCC)

;;CASE 1: constraint defined inside routine mult
;CONSTRAINT  MAXTIME : Inic_mult TO end_mult = 0 us

;FU#1   INPUT   OPERANDS   =   f14,f15,f12,f13   :   OUTPUT   OPERANDS   =
f8,f9,f10,f11

;;*****************************************************************
;;                        IIR.ASM PIC16C57/PARMIC
;;
;;*****************************************************************
;;       4th order IIR Elliptic Lowpass Filter
;;
;;       Performance :
;;                        Program Memory  :  309 instructions
;;                        Data Memory     :  49 + 7 (56 registers)
;;                       # of cycles     :  2651/2792(PIC16C57/PARMIC)
;;
;;*****************************************************************
;;
;;       The specifications of the filter are :
;;
;;       Filter Type = 4th Order Elliptic Lowpass Filter
;;
;;                                Band1           Band2
;;       Lower Band Edge          0.0             600 Hz
;;       Upper Band Edge          500 Hz          1 Khz
;;       Nominal Gain             1.0             0.0
;;       Nominal Ripple           0.01            0.05
;;       Maximum Ripple           0.00906         0.04601
;;       Ripple in dB             0.07830         -26.75
;;
;;  The Filter Co-efficients for the above specifications
;; of the filter are computed as follows :
;;
;;      1st Section :
;;                        A11 = -0.133331
;;                        A12 = 0.167145
;;                        B10 = 0.285431
;;                        B11 = 0.462921
;;                        B12 = 0.285431
;;      2nd Section
;;                        A21 = 0.147827
;;                        A22 = 0.765900
;;                        B20 = 0.698273
;;                        B21 = 0.499908
;;                        B22 = 0.698273
;;
;;*****************************************************************

A0      equ     d'15'      ;; lower 8 bit multiplicand
A1      equ     d'14'      ;; higher 8 bit multiplicand
B0      equ     d'13'      ;; lower 8 bit multiplier
B1      equ     d'12'      ;; higher 8 bit multiplier
P0      equ     d'11'      ;; 1st byte of the 32 bit result
P1      equ     d'10'      ;; 2nd byte of the 32 bit result
```

```
P2      equ     d'09'       ;; 3rd byte of the 32 bit result
P3      equ     d'08'       ;; 4rd byte of the 32 bit result

count   equ     7           ;; loop counter

STATUS  equ     3           ;; Identifiers
CARRY   equ     0
FSR     equ     4
                            ;; Filter coefficients
A11     equ     d'16'       ;; MAP 00
A11_Hi  equ     d'17'
A12     equ     d'18'
A12_Hi  equ     d'19'
B10     equ     d'20'
B10_Hi  equ     d'21'
B11     equ     d'22'
B11_Hi  equ     d'23'
B12     equ     d'24'
B12_Hi  equ     d'25'
A21     equ     d'16'       ;; MAP 01
A21_Hi  equ     d'17'
A22     equ     d'18'
A22_Hi  equ     d'19'
B20     equ     d'20'
B20_Hi  equ     d'21'
B21     equ     d'22'
B21_Hi  equ     d'23'
B22     equ     d'24'
B22_Hi  equ     d'25'
                                ;; Aux. variables
Dn1             equ     d'26'   ;; MAP 00
Dn1_Hi          equ     d'27'
Dn1_1           equ     d'28'
Dn1_1_Hi        equ     d'29'
Dn1_2           equ     d'30'
Dn1_2_Hi        equ     d'31'
Dn2             equ     d'26'   ;; MAP 01
Dn2_Hi          equ     d'27'
Dn2_1           equ     d'28'
Dn2_1_Hi        equ     d'29'
Dn2_2           equ     d'30'
Dn2_2_Hi        equ     d'31'

                        ;; MAP 10
X0      equ     d'19'   ;; Input
X1      equ     d'18'
Y0      equ     d'17'   ;; Output
Y1      equ     d'16'

ACC0    equ     d'23'
ACC1    equ     d'22'
ACC2    equ     d'21'
ACC3    equ     d'20'

;;*****************************************************************
;;                  Program start here for PARMIC
;;*****************************************************************
      org   0x000
```

```
      nop
      goto  start
;;*****************************************************************
;;                16x16 Software Signed Multiplier
;;*****************************************************************
;;
;; The 32 bit result is stored in 4 bytes P3 P2 P1 P0 = A1 A0 x B1 B0
;;
;;*****************************************************************
;;
Inic_mult
mult    clrf    P2
        clrf    P3
        clrf    P1
        clrf    P0

        btfss   B1,7            ;; Test sign of operand B
        goto    argpos          ;; if positive, ok
                                ;; else negate both operands
        comf    B0              ;; first negate B
        comf    B1
        movlw   d'1'
        addwf   B0
        btfss   STATUS,CARRY
        goto    carry10
        incf    B1
carry10
        comf    A0              ;; second negate A
        comf    A1
        movlw   d'1'
        addwf   A0
        btfss   STATUS,CARRY
        goto    argpos
        incf    A1

argpos  movlw   d'15'        ;; If unsigned multiplication count = 16
        movwf   count
        bcf     STATUS,CARRY
loop
        rrf     B1
        rrf     B0
        btfss   STATUS,CARRY
        goto    skip_Ad
        movf    A0,w
        addwf   P2
        btfsc   STATUS,CARRY
        incf    P3
        movf    P3,w
        addwf   A1,w
        movwf   P3
skip_Ad rlf     A1,w
        rrf     P3
        rrf     P2
        rrf     P1
        rrf     P0
        decfsz  count
        goto    loop
                             ;; When unsigned multiplication this part
```

```
                            ;; of code disapear
        rlf     A1,w        ;; since operand B is always made positive
        rrf     P3
        rrf     P2
        rrf     P1
end_mult
        rrf     P0

        retlw   0
;;
;;*****************************************************************
;;                            add32 (32+32)
;;*****************************************************************
;;
;;       Performance :
;;                        Program Memory  :  15 instructions
;;                        # of cycles     :  15
;;
;;*****************************************************************
;;
add32   movf    P0,w            ;; Begin ADD32
        addwf   ACC0
        btfsc   STATUS,CARRY
        incf    ACC1
        movf    P1,w
        addwf   ACC1
        btfsc   STATUS,CARRY
        incf    ACC2
        movf    P2,w
        addwf   ACC2
        btfsc   STATUS,CARRY
        incf    ACC3
        movf    P3,w
        addwf   ACC3            ;; End ADD32

        retlw   0x00
;;
;;*****************************************************************
;;                             IIR_Filter
;;*****************************************************************
;;
;;
;;       Performance :
;;                Program Memory  :  168 instructions
;;                # of cycles : 2651/2792 (PIC16C57/PARMIC)c/ CALL
;;
;;*****************************************************************
;;
IIR_Fil clrf    FSR
                                ;; Begin 1st BIQUAD Filter Section
        movf    Dn1_2,w
        movwf   A0
        movf    Dn1_2_Hi,w
        movwf   A1
        movf    A12,w
        movwf   B0
        movf    A12_Hi,w
        movwf   B1
```

```
label1
        call    mult            ;; A2*D(n-2)

        bcf     FSR,5
        bsf     FSR,6
label2
        call    add32

        clrf    FSR
        movf    Dn1_1,w
        movwf   A0
        movf    Dn1_1_Hi,w
        movwf   A1
        movf    A11,w
        movwf   B0
        movf    A11_Hi,w
        movwf   B1

        call    mult            ;; A1*D(n-1)

        bcf     FSR,5
        bsf     FSR,6

        call    add32

        rlf     ACC1,w
        rlf     ACC2,w
        clrf    FSR
        movwf   Dn1
        bcf     FSR,5
        bsf     FSR,6
        rlf     ACC3,w
        clrf    FSR
        movwf   Dn1_Hi
        movf    Dn1,w           ;; A = D(n) + D(n-2)
        addwf   Dn1_2,w
        movwf   A0
        btfsc   STATUS,CARRY
        incf    Dn1_Hi
        movf    Dn1_Hi,w
        addwf   Dn1_2_Hi,w
        movwf   A1
        movf    B10,w           ;; B = B10
        movwf   B0
        movf    B10_Hi,w
        movwf   B1

        call    mult            ;; ACC = B2*(Dn + Dn_2)

        bcf     FSR,5
        bsf     FSR,6
        movf    P0,w
        movwf   ACC0
        movf    P1,w
        movwf   ACC1
        movf    P2,w
        movwf   ACC2
        movf    P3,w
```

```
        movwf   ACC3
        clrf    FSR
        movf    Dn1_1,w
        movwf   A0
        movwf   Dn1_2
        movf    Dn1_1_Hi,w
        movwf   A1
        movwf   Dn1_2_Hi        ;; A = D(n-1)
        movf    B11,w           ;; B = B11
        movwf   B0
        movf    B11_Hi,w
        movwf   B1

        call    mult            ;; ACC = B11*D(n-1)

        bsf     FSR,6
        bcf     FSR,5

        call    add32
        clrf    FSR
        movf    Dn1,w           ;; D(n-1) = D(n)
        movwf   Dn1_1
        movf    Dn1_Hi,w
        movwf   Dn1_1_Hi        ;; End 1st BIQUAD Filter Section

                                ;; Begin 2nd BIQUAD Filter Section
        bsf     FSR,5
        movf    Dn2_2,w
        movwf   A0
        movf    Dn2_2_Hi,w
        movwf   A1
        movf    A22,w
        movwf   B0
        movf    A22_Hi,w
        movwf   B1

        call    mult            ;; A2*D(n-2)

        bsf     FSR,6
        bcf     FSR,5

        call    add32

        bcf     FSR,6
        bsf     FSR,5
        movf    Dn2_1,w
        movwf   A0
        movf    Dn2_1_Hi,w
        movwf   A1
        movf    A21,w
        movwf   B0
        movf    A21_Hi,w
        movwf   B1

        call    mult            ;; A1*D(n-1)

        bsf     FSR,6
        bcf     FSR,5
```

```
        call    add32

        rlf     ACC1,w
        rlf     ACC2,w
        bcf     FSR,6
        bsf     FSR,5
        movwf   Dn2
        bsf     FSR,6
        bcf     FSR,5
        rlf     ACC3,w
        bcf     FSR,6
        bsf     FSR,5
        movwf   Dn2_Hi
        movf    Dn2,w           ;; A = D(n) + D(n-2)
        addwf   Dn2_2,w
        movwf   A0
        btfsc   STATUS,CARRY
        incf    Dn2_Hi
        movf    Dn2_Hi,w
        addwf   Dn2_2_Hi,w
        movwf   A1
        movf    B20,w
        movwf   B0
        movf    B20_Hi,w
        movwf   B1

        call    mult            ;; ACC = B2*(Dn + Dn_2)

        bsf     FSR,6
        bcf     FSR,5
        movf    P0,w
        movwf   ACC0
        movf    P1,w
        movwf   ACC1
        movf    P2,w
        movwf   ACC2
        movf    P3,w
        movwf   ACC3
        bcf     FSR,6
        bsf     FSR,5
        movf    Dn2_1,w
        movwf   A0
        movwf   Dn2_2
        movf    Dn2_1_Hi,w
        movwf   A1
        movwf   Dn2_2_Hi        ;; A = D(n-1)
        movf    B21,w           ;; B = B11
        movwf   B0
        movf    B21_Hi,w
        movwf   B1

        call    mult            ;; ACC = B11*D(n-1)

        bsf     FSR,6
        bcf     FSR,5

        call    add32
```

```
        bcf     FSR,6
        bsf     FSR,5
        movf    Dn2,w            ;; D(n-1) = D(n)
        movwf   Dn2_1
        movf    Dn2_Hi,w
        movwf   Dn2_1_Hi         ;; End 2nd BIQUAD Filter Section

        bsf     FSR,6
        bcf     FSR,5
        movf    ACC2,w
        movwf   Y0
        movf    ACC3,w
        movwf   Y1

        retlw   0x00

;;*****************************************************************
;;    Main Program
;;*****************************************************************
start
                                ;; Begin Init. Filter
        movlw   0x11            ;; -A11
        movwf   A11
        movlw   0x11
        movwf   A11_Hi

        movlw   0x9B            ;; -A12
        movwf   A12
        movlw   0xEA
        movwf   A12_Hi

        movlw   0x98
        movwf   B10
        movlw   0x24
        movwf   B10_Hi

        movlw   0x41
        movwf   B11
        movlw   0x3B
        movwf   B11_Hi

        movlw   0x98
        movwf   B12
        movlw   0x24
        movwf   B12_Hi

        bsf     FSR,5

        movlw   0x14            ;; -A21
        movwf   A21
        movlw   0xED
        movwf   A21_Hi

        movlw   0xF7          ;; -A22
        movwf   A22
        movlw   0x9D
        movwf   A22_Hi
```

```
        movlw   0x61
        movwf   B20
        movlw   0x59
        movwf   B20_Hi

        movlw   0xFD
        movwf   B21
        movlw   0x3F
        movwf   B21_Hi

        movlw   0x61
        movwf   B22
        movlw   0x59
        movwf   B22_Hi

        bcf     FSR,5
        clrf    Dn1             ;; Clear Dn's    (6 coefficients)
        clrf    Dn1_Hi
        clrf    Dn1_1
        clrf    Dn1_1_Hi
        clrf    Dn1_2
        clrf    Dn1_2_Hi

        bsf     FSR,5
        clrf    Dn2             ;; Clear Dn's    (+6 coefficients)
        clrf    Dn2_Hi
        clrf    Dn2_1
        clrf    Dn2_1_Hi
        clrf    Dn2_2
        clrf    Dn2_2_Hi        ;; End of clear Dn's
                                ;; End Init. Filter

NexPo   clrf    FSR
        movf    5,w
        bsf     FSR,6
        bcf     FSR,5
        movwf   X0
        clrf    FSR
        movf    6,w
        bsf     FSR,6
        bcf     FSR,5
        movwf   X1
        bcf     STATUS,CARRY    ;; ACC = scaled input : X*2^15
        clrf    ACC0
        clrf    ACC1
        rrf     X1
        rrf     X0
        rrf     ACC1
        movf    X1,w
        movwf   ACC3
        movf    X0,w
        movwf   ACC2            ;; End scale input

        call    IIR_Fil
        goto    NexPo

     END
```

D.2 Parte da especificação VHDL gerada para a ROM

```
library IEEE;
use IEEE.std_logic_1164.all;

package rom is
constant ROM_WIDTH: integer := 12;
constant ROM_LENGTH: integer := 309;
subtype ROM_WORD is std_logic_vector (ROM_WIDTH-1 downto 0);
subtype ROM_RANGE is INTEGER range 0 to ROM_LENGTH-1;
type ROM_TABLE is array (0 to ROM_LENGTH-1) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
                  ROM_WORD'("000000000000"),
                  ROM_WORD'("101011101000"),
                  {...}
                  ROM_WORD'("100100101100"),
                  ROM_WORD'("101100001100"));
end rom;
```

D.3 Módulo VHDL da definição de parâmetros

```
library IEEE;
use IEEE.std_logic_1164.all;

package param is
      constant stack_depth: integer := 2;
      constant BITMAN: boolean := TRUE;
      constant Timer: boolean := FALSE;
      constant WDTimer: boolean := FALSE;
      constant NumberOfFU: integer := 1;
      constant reg_with: integer := 8;
      constant reg_num: integer range 0 to 79:= 55;
      constant reg_IO: integer range 0 to 4:= 0;
      constant reg_I: integer := 2;
      subtype REG is integer range 0 to 79;
      type ARRAY_IPorts is array (1 to 2) of REG;
      constant IN_PORT :  ARRAY_IPorts:= (5, 6);
      constant reg_O: integer := 0;
      subtype PORT_WORD is std_logic_vector(7 downto 0);
      subtype PORTS is std_logic_vector(15 downto 0);
      subtype reg_range is integer range 0 to reg_num-1;
end param;
```

D.4 Parte da especificação VHDL que descreve a ROM do programa com comunicação com a UF

```
library IEEE;
use IEEE.std_logic_1164.all;

package rom is
constant ROM_WIDTH: integer := 12;
constant ROM_LENGTH: integer := 289;
subtype ROM_WORD is std_logic_vector (ROM_WIDTH-1 downto 0);
subtype ROM_RANGE is INTEGER range 0 to ROM_LENGTH-1;
```

```
type ROM_TABLE is array (0 to ROM_LENGTH-1) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
                  ROM_WORD'("000000000000"),
                  ROM_WORD'("101011010100"),
            {...}
                  ROM_WORD'("100100101100"),
                  ROM_WORD'("101100001100"));
end rom;
```

D.5 Especificação VHDL gerada para a UF e interface com o PARMIC

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.aritmetica.all;

entity unit0 is
port(       Sin: in std_logic_vector(7 downto 0);
            Sout: out std_logic_vector(7 downto 0);
            ld_B: in std_logic_vector(4 downto 1);
            enable_W: in std_logic_vector(4 downto 1));
end unit0;

architecture comportamental of unit0 is
      SIGNAL IN14 :  std_logic_vector(7 downto 0);
      SIGNAL IN15 :  std_logic_vector(7 downto 0);
      SIGNAL IN12 :  std_logic_vector(7 downto 0);
      SIGNAL IN13 :  std_logic_vector(7 downto 0);
      SIGNAL OUT8 :  std_logic_vector(7 downto 0);
      SIGNAL OUT9 :  std_logic_vector(7 downto 0);
      SIGNAL OUT10 :  std_logic_vector(7 downto 0);
      SIGNAL OUT11 :  std_logic_vector(7 downto 0);

begin
      ----------------------------------------------------------------
      --  input Operand Latch's
      ----------------------------------------------------------------
      INTERFACE_IN: process(Sin, ld_B)
      begin
                  if ld_B(1) = '1' then
                        IN14 <= Sin;
                  end if;
                  if ld_B(2) = '1' then
                        IN15 <= Sin;
                  end if;
                  if ld_B(3) = '1' then
                        IN12 <= Sin;
                  end if;
                  if ld_B(4) = '1' then
                        IN13 <= Sin;
                  end if;
      end process;

      ----------------------------------------------------------------
      --  Output result into the bus
      ----------------------------------------------------------------
```

```
INTERFACE_OUT: process(enable_W, OUT8, OUT9, OUT10, OUT11)
begin
      if enable_W(1) = '1' then
            Sout <= OUT8;
      elsif enable_W(2) = '1' then
            Sout <= OUT9;
      elsif enable_W(3) = '1' then
            Sout <= OUT10;
      elsif enable_W(4) = '1' then
            Sout <= OUT11;
      else Sout <= "ZZZZZZZZ";
      end if;
end process;

process(IN14, IN15, IN12, IN13)
      VARIABLE W, F3 : std_logic_vector(7 downto 0);
      VARIABLE F9 : std_logic_vector(7 downto 0);
      VARIABLE F9_aux : std_logic_vector(9 downto 0);
      VARIABLE F8 : std_logic_vector(7 downto 0);
      VARIABLE F8_aux : std_logic_vector(9 downto 0);
      VARIABLE F10 : std_logic_vector(7 downto 0);
      VARIABLE F10_aux : std_logic_vector(9 downto 0);
      VARIABLE F11 : std_logic_vector(7 downto 0);
      VARIABLE F11_aux : std_logic_vector(9 downto 0);
      VARIABLE F12 : std_logic_vector(7 downto 0);
      VARIABLE F12_aux : std_logic_vector(9 downto 0);
      VARIABLE F22 : std_logic_vector(7 downto 0);
      VARIABLE F22_aux : std_logic_vector(9 downto 0);
      VARIABLE F13 : std_logic_vector(7 downto 0);
      VARIABLE F13_aux : std_logic_vector(9 downto 0);
      VARIABLE F1 : std_logic_vector(7 downto 0);
      VARIABLE F1_aux : std_logic_vector(9 downto 0);
      VARIABLE F15 : std_logic_vector(7 downto 0);
      VARIABLE F15_aux : std_logic_vector(9 downto 0);
      VARIABLE F14 : std_logic_vector(7 downto 0);
      VARIABLE F14_aux : std_logic_vector(9 downto 0);
      VARIABLE F7 : std_logic_vector(7 downto 0);
      VARIABLE F7_aux : std_logic_vector(9 downto 0);
      VARIABLE F36 : std_logic_vector(7 downto 0);
      VARIABLE F36_aux : std_logic_vector(9 downto 0);
      VARIABLE F24 : std_logic_vector(7 downto 0);
      VARIABLE F24_aux : std_logic_vector(9 downto 0);

      begin
            F14 := IN14;
            F15 := IN15;
            F12 := IN12;
            F13 := IN13;
             F3 := "00000000";
             F3(0) := '1';
             F9 := "00000000";
             F3(0) := '1';
             F8 := "00000000";
             F3(0) := '1';
             F10 := "00000000";
             F3(0) := '1';
             F11 := "00000000";
```

```
 F3(0) := '1';
IF (F12(7) = '1') THEN
 F13 := NOT(F13);
 F3(2) := allzero(F13);
 F12 := NOT(F12);
 F3(2) := allzero(F12);
 W := "00000001";
 F13_aux := W + F13;
 F13 := F13_aux(7 downto 0);
 F3(0) := F13_aux(8);
 F3(1) := F13_aux(9);
 F3(2) := allzero(F13_aux(7 downto 0));
IF (F3(0) = '1') THEN
 F12 := inc(F12);
 F3(2) := allzero(F12);
END IF;
 F15 := NOT(F15);
 F3(2) := allzero(F15);
 F14 := NOT(F14);
 F3(2) := allzero(F14);
 W := "00000001";
 F15_aux := W + F15;
 F15 := F15_aux(7 downto 0);
 F3(0) := F15_aux(8);
 F3(1) := F15_aux(9);
 F3(2) := allzero(F15_aux(7 downto 0));
IF (F3(0) = '1') THEN
 F14 := inc(F14);
 F3(2) := allzero(F14);
END IF;
END IF;
 W := "00001111";
 F7 := W;
 F3(0) := '0';
FOR i in 15 downto 1 loop
 F12_aux(7 downto 0) := F3(0) & F12(7 downto 1);
 F3(0) := F12(0);
 F12 := F12_aux(7 downto 0);
 F13_aux(7 downto 0) := F3(0) & F13(7 downto 1);
 F3(0) := F13(0);
 F13 := F13_aux(7 downto 0);
IF (F3(0) = '1') THEN
 F3(2) := allzero(F15);
 W := F15;
 F9_aux := W + F9;
 F9 := F9_aux(7 downto 0);
 F3(0) := F9_aux(8);
 F3(1) := F9_aux(9);
 F3(2) := allzero(F9_aux(7 downto 0));
IF (F3(0) = '1') THEN
 F8 := inc(F8);
 F3(2) := allzero(F8);
END IF;
 F3(2) := allzero(F8);
 W := F8;
 F14_aux := W + F14;
 W := F14_aux(7 downto 0);
 F3(0) := F14_aux(8);
```

```
                    F3(1) := F14_aux(9);
                    F3(2) := allzero(F14_aux(7 downto 0));
                    F8 := W;
                   END IF;
                    F14_aux(7 downto 0) := F14(6 downto 0) & F3(0);
                    F3(0) := F14(7);
                    W := F14_aux(7 downto 0);
                    F8_aux(7 downto 0) := F3(0) & F8(7 downto 1);
                    F3(0) := F8(0);
                    F8 := F8_aux(7 downto 0);
                    F9_aux(7 downto 0) := F3(0) & F9(7 downto 1);
                    F3(0) := F9(0);
                    F9 := F9_aux(7 downto 0);
                    F10_aux(7 downto 0) := F3(0) & F10(7 downto 1);
                    F3(0) := F10(0);
                    F10 := F10_aux(7 downto 0);
                    F11_aux(7 downto 0) := F3(0) & F11(7 downto 1);
                    F3(0) := F11(0);
                    F11 := F11_aux(7 downto 0);
                   END loop;
                    F14_aux(7 downto 0) := F14(6 downto 0) & F3(0);
                    F3(0) := F14(7);
                    W := F14_aux(7 downto 0);
                    F8_aux(7 downto 0) := F3(0) & F8(7 downto 1);
                    F3(0) := F8(0);
                    F8 := F8_aux(7 downto 0);
                    F9_aux(7 downto 0) := F3(0) & F9(7 downto 1);
                    F3(0) := F9(0);
                    F9 := F9_aux(7 downto 0);
                    F10_aux(7 downto 0) := F3(0) & F10(7 downto 1);
                    F3(0) := F10(0);
                    F10 := F10_aux(7 downto 0);
                    F11_aux(7 downto 0) := F3(0) & F11(7 downto 1);
                    F3(0) := F11(0);
                    F11 := F11_aux(7 downto 0);
                   OUT8 <= F8;
                   OUT9 <= F9;
                   OUT10 <= F10;
                   OUT11 <= F11;
      end process;

end comportamental;
```

D.6 Mensagens do BINOMIO quando se realiza a partição

```
End of the Assembler.

The Total Area of the IC (23639 sites) exceeds the SOG area (18880
sites)!

Identified Loop with 1:
00000000000000000000000011111111111111111100000
cycles = 28  cycles for loop code = 270
Code Segment Execution Time = 55.875000 (us)

Constraint Violation: +55.875000 (us)
```

```
Execution time for total communication Hw/Sw: 4.503000 (us)

Start VHDL Generation...
End VHDL Generation.

How many instruction cycles FU#0 takes (1 cycle = 0.187500 us)? 1
Total execution time between the labels SW/HW: 4.690500 us
The new program with FU interface has: 289 instructions

Total FU Area (in sites)? 14579
The Total Area of the IC (38488 sites) exceeds the SOG area (18880
sites)!

End of the program ROM generation.
```