UNIVERSIDADE DO MINHO

METODOLOGIAS DE ESPECIFICAÇÃO E ANÁLISE DE SISTEMAS DIGITAIS: DESENVOLVIMENTO DE UM CONTROLADOR APA (GLiTCH)

Henrique Manuel Dinis dos Santos

Braga, Fevereiro de 1996

UNIVERSIDADE DO MINHO

METODOLOGIAS DE ESPECIFICAÇÃO E ANÁLISE DE SISTEMAS DIGITAIS: DESENVOLVIMENTO DE UM CONTROLADOR APA (GLiTCH)

Dissertação submetida à Universidade do Minho

para obtenção do grau de Doutor em Engenharia de Computadores

por

Henrique Manuel Dinis dos Santos

Departamento de Informática

Braga, Fevereiro de 1996

Numa singela homenagem à família e ao espírito de união que representa, dedico este trabalho à Janja, ao Nuno e ao Ricardo

Agradecimentos

Gostaria de deixar aqui expresso o meu reconhecimento público a todos os que, directa ou indirectamente, contribuíram para a execução deste trabalho. Pela sua importância merecem realce as seguintes menções particulares.

Ao Professor Alberto Proença, pela dedicação que revelou na supervisão desta tese, os conselhos, as sugestões, a disponibilidade e a amizade que demonstrou durante todos estes anos.

À JNICT, pelo apoio financeiro na execução do projecto do controlador do GLiTCH.

Ao Professor Erik Dagless, pela oportunidade que me deu de experimentar um aliciante trabalho em equipa, e pelo referencial de comportamento no "mundo da investigação"

Aos Professores José Manuel Valença, Altamiro Machado e José Maia Neves, pelos conselhos e incentivo.

Ao Richard Storer, Andy Duller, Andrew Thomson e Alan Chalmers que, para além do envolvimento directo ou indirecto no projecto, deram todo o seu apoio nas minhas permanências em Bristol, assim como importantes conselhos para a elaboração da tese.

Aos meus amigos do DI, com especial carinho para aqueles que, como eu, percorreram (ou ainda percorrem) o difícil caminho até ao grau académico desejado.

A todos os que contribuíram para a elaboração desta tese, disponibilizando a sua informação na Internet, com o esforço pessoal e com o único intuito de promover a divulgação do conhecimento.

À Universidade do Minho e à Escola de Engenharia, por terem disponibilizado as condições necessárias para a execução deste trabalho.

À Maria Almeida, pela ajuda e o tempo dispensado na revisão do texto da dissertação.

À minha família e amigos, em particular aos meus pais a quem devo o que sou, pelo apoio incondicional, pelo carinho e pela compreensão, principalmente nos momentos em que deveríamos ter estado juntos.

Ao Nuno e ao Ricardo, pelo tempo que lhes roubei e que não lhes posso devolver, pelo incentivo que representa o seu estatuto e por tudo o que de bom evocam.

E finalmente, pelo amor, a paciência, o apoio, a compreensão, principalmente nos numerosos fins-de-semana e noites solitárias, muito obrigado, Janja.

Resumo

Um dos domínios de aplicação dos sistemas digitais de elevada complexidade é a visão por computador e, em particular, o processamento de imagem em tempo real (*video rate*). Os requisitos impostos por este domínio de aplicação motivaram esta tese em metodologias de desenvolvimento de sistemas digitais. A evolução tecnológica tem introduzido novos paradigmas de desenvolvimento, nomeadamente no que respeita à flexibilidade das soluções, à síntese de alto nível, à testabilidade e à prototipagem rápida. Algumas questões relacionadas com a aplicação desses paradigmas são analisadas nesta tese, utilizando como *case study* a implementação de um controlador para um *Array* de processadores associativos, base de uma arquitectura heterogénea para visão por computador. As tarefas de modelação, verificação e síntese apoiaram-se em linguagens de alto nível - em particular a linguagem VHDL - enquanto que na implementação do protótipo utilizou-se a lógica programável, como forma de promover a flexibilidade. No contexto da síntese de alto nível, a utilização da lógica programável levanta algumas questões, nomeadamente o problema da variação sintáctica e a escolha da estrutura lógica programável mais adequada a uma dada descrição. Estes assuntos são analisados utilizando como exemplo algumas das descrições em VHDL realizadas durante o projecto do controlador. As metodologias utilizadas são validadas através da sua aplicação e alguns dos resultados obtidos são generalizados, num contexto alargado à sua aplicação em prototipagem rápida.

ÍNDICE

Página

Agradecimentos iv

Resumo v

Lista de Figuras viii

Lista de Tabelas ix

1. Introdução 1

2. Arquitecturas paralelas para visão por computador 4
- 2.1 Paralelismo 4
- 2.2 Arquitecturas paralelas 5
- 2.3 Arquitecturas para visão por computador 8
- 2.4 Arquitecturas SIMD para processamento de imagem 9
- 2.5 Arquitecturas SIMD com memória associativa 14
- 2.6 Arquitectura heterogénea baseada no GLiTCH 18
  - 2.6.1 O GLiTCH 18
  - 2.6.2 O HVA 20

3. O Projecto de sistemas digitais 23
- 3.1 Introdução 23
- 3.2 Estrutura do processo de desenvolvimento 24
  - 3.2.1 Níveis de abstracção 24
  - 3.2.2 Operações sobre o projecto 26
  - 3.2.3 Testabilidade 27
  - 3.2.4 Conclusão 28
- 3.3 Ferramentas de CAD 28
- 3.4 Tecnologias de implementação 30
  - 3.4.1 Critérios de selecção 32
  - 3.4.2 Componentes programáveis pelo utilizador 34
  - 3.4.3 Exemplo dos CPLD 37
  - 3.4.4 Exemplo dos FPGA 42
- 3.5. Síntese automática 48
  - 3.5.1 Síntese ao nível *Architecture* 49
  - 3.5.2 Síntese ao nível *Logic* 50
- 3.6 Linguagens para descrição de *hardware* (HDL) 51
  - 3.6.1 VHDL 53
  - 3.6.2 VHDL para síntese de alto nível 55
  - 3.6.3 O ViewSynthesis 59
- 3.7 Conclusão 63

4. O controlador do GLiTCH 65
- 4.1 Introdução 65
- 4.2 Metodologia de desenvolvimento 67
  - 4.2.1 Testabilidade no controlador do GLiTCH 70
  - 4.2.2 O transputer na função de teste 71

4.3 Unidade de memória de padrões 72
4.3.1 Modelo VHDL da unidade de memória de padrões 74
4.4 Unidade escalar 78
4.4.1 Modelo VHDL da unidade escalar 80
4.5 Selector de *tags* 83
4.5.1 Modelo VHDL do selector de *tags* 84
4.6 Sequenciador 86
4.6.1 Família Am29300 87
4.6.2 Modelo VHDL do sequenciador 90
4.7 Micromemória 94
4.7.1 Modelo VHDL da micromemória 95
4.8 Processador hospedeiro 98
4.8.1 Modelo VHDL do arbitrador de barramentos 101
4.8.2 Modelo VHDL da operação do processador hospedeiro 103
4.9 Modelo estrutural do controlador em VHDL 106
4.9.1 Utilização do modelo estrutural na simulação 108
4.10 Estrutura de *pipeline* 109
4.11 Finalização do projecto de *hardware* 110
4.12 Interface com o compilador 111

5. Estratégias de implementação com lógica programável 113
5.1 Motivação 113
5.2 Metodologia e parâmetros de avaliação 114
5.2.1 Exploração do espaço de projecto 115
5.2.2 Selecção do tipo de lógica programável (CPLD vs. FPGA) 116
5.3 Estudo da implementação do arbitrador de barramentos 117
5.3.1 Exploração do espaço de projecto e variação sintáctica 117
5.3.2 Análise de resultados 123
5.3.3 Implementação em CPLD e FPGA 125
5.4 Estudo da implementação da unidade escalar 126
5.4.1 Síntese da descrição VHDL 126
5.4.2 Implementação em CPLD e FPGA 127
5.5 Estudo da implementação da AGL 128
5.5.1 Síntese da descrição VHDL 128
5.5.2 Implementação em CPLD e FPGA 129
5.6 Sumário 130

6. Conclusões 133
6.1 Trabalho futuro 135

Bibliografia 137

Apêndice A - Modelo VHDL do controlador do GLiTCH 145

Apêndice B - Diagramas esquemáticos do controlador do GLiTCH 187

Apêndice C - Descrições em PALASM 210

Apêndice D - Microprogramas de teste para controlador do GLiTCH 262

Lista de Figuras

Página

Figura 2.1 Arquitectura SISD 6
Figura 2.2 Arquitectura SIMD 6
Figura 2.3 Arquitectura VLIW 7
Figura 2.4 Arquitectura MIMD 7
Figura 2.5 Modelo XIMD 8
Figura 2.6 Organização típica de uma CAM 15
Figura 2.7 Arquitectura interna do GLiTCH 19
Figura 2.8 Sistema GLiTCH 20
Figura 2.9 Diagrama de blocos do sistema HVA 21
Figura 3.1 Modelo tripartido para domínios e níveis de representação (*Y-chart*) 25
Figura 3.2 Taxinomia das ferramentas de CAD 29
Figura 3.3 Tecnologias de implementação de sistemas digitais 31
Figura 3.4 Diagrama de blocos dos MACH1 e MACH2 38
Figura 3.5 *Logic Allocator* 39
Figura 3.6 Macrocélula dos MACH1 e MACH2 40
Figura 3.7 Estrutura lógica da OSM (*Output Switch Matrix*) 41
Figura 3.8 Estrutura interna (simplificada) de um FPGA da Xilinx 43
Figura 3.9 CLB dos componentes da família XC2000 44
Figura 3.10 Bloco de I/O dos componentes da família XC2000 45
Figura 3.11 *Y-chart* para a síntese baseada na ferramenta ViewSynthesis 61
Figura 3.12 Detalhe da operação da ferramenta ViewSynthesis 62
Figura 4.1 Diagrama de blocos do sistema 67
Figura 4.2 Diagrama de blocos da unidade de memória de padrões 74
Figura 4.3 Diagrama lógico da Unidade escalar 80
Figura 4.4 Diagrama lógico do selector de *tags* e suas ligações no sistema 84
Figura 4.5 Formato da instrução do sequenciador 88
Figura 4.6 Diagrama lógico do sequenciador 90
Figura 4.7 Diagrama de blocos do sequenciador 92
Figura 4.8 Definição da microinstrução do sistema 95
Figura 4.9 Exemplo de uma linha de microprograma 98
Figura 4.10 Diagrama temporal do arbitrador de barramentos 99
Figura 4.11 Diagrama lógico do arbitrador de barramentos 100
Figura 4.12 Estrutura de um microprograma 112
Figura 5.1 Representação dos resultados da síntese 116
Figura 5.2 Resumo dos resultados da síntese do BA 124

Lista de Tabelas

Página

Tabela 3.1 Comparação entre as tecnologias de implementação 34
Tabela 3.2 MACH: componentes mais complexos de cada subfamília 42
Tabela 3.3 FPGA da Xilinx: componentes mais complexos de cada família 47
Tabela 4.1 Instruções da unidade de memória de padrões 73
Tabela 4.2 Instruções da unidade escalar 79
Tabela 4.3 Instruções do selector de *tags* 83
Tabela 4.4 Grupos de instruções do sequenciador 88
Tabela 4.5a Instruções condicionais e incondicionais do sequenciador 89
Tabela 4.5b Instruções especiais do sequenciador 90
Tabela 5.1 Resultados da síntese do BA (alternativa 1) 119
Tabela 5.2 Resultados da síntese do BA (alternativa 2) 120
Tabela 5.3 Resultados da síntese do BA (alternativa 3) 121
Tabela 5.4 Resultados da síntese do BA (alternativa 4) 122
Tabela 5.5 Resultados da síntese do BA (alternativa 5) 123
Tabela 5.6 Resumo da síntese do BA em componentes FPGA e CPLD 126
Tabela 5.7 Resultados da síntese da SU 127
Tabela 5.8 Resumo da síntese da SU em componentes FPGA e CPLD 128
Tabela 5.9 Resultados da síntese da AGL 129
Tabela 5.10 Resumo da síntese da AGL em componentes FPGA e CPLD 130

Capítulo 1

1. Introdução

A Engenharia Electrónica Digital é caracterizada pela interdependência entre dois dos seus elementos fundamentais: os semicondutores e as ferramentas de CAD/CAE/CAM. Os sistemas electrónicos não podem existir sem os semicondutores, e estes são construídos utilizando sistemas electrónicos que suportam as referidas ferramentas e cujas possibilidades dependem grandemente do desempenho dos próprios semicondutores. O resultado desta interdependência é uma evolução tecnológica notável. Este é um exemplo de aplicação de um dos princípios mais importantes da Engenharia em geral, a reutilização, que permite construir produtos cada vez mais evoluídos, em cada iteração do respectivo processo de desenvolvimento.

Muitos domínios de aplicação têm beneficiado e promovido esta evolução tecnológica. Um desses domínios, particularmente exigente, é a visão por computador. A visão por computador, quando aplicada a uma sequência de imagens à razão de transferência do sinal de vídeo (25 imagens por segundo), exige sistemas com consideráveis capacidades de processamento, muito superiores ao que é possível dispor nos equipamentos vulgarmente disponíveis. Como exemplo, supondo uma digitalização de 512x512 pontos, e cada ponto codificado por 8 bits, será necessário processar 6.25Mbytes por segundo. Para além deste facto, na solução de um dado problema as diferentes tarefas a executar exigem, tipicamente, a integração de arquitecturas paralelas diferentes, em computadores ditos heterogénios. Pelas suas características tais computadores exigem a construção de sistemas digitais complexos.

O projecto deste tipo de sistema digital apresenta diversas dificuldades em todas as suas fases de execução: especificação, desenvolvimento e teste. É necessário recorrer a ferramentas de CAD e CAE e utilizar metodologias adequadas ao nível da complexidade em questão, tendo em atenção, entre outros factores, o tempo de execução do projecto, a verificação dos modelos, a tecnologia que melhor se adapta à implementação, a testabilidade do sistema, a coerência e integridade que é necessário garantir na informação que é trocada entre as diversas tarefas a executar.

O resultado de investigações recentes tem demonstrado a inadequação dos convencionais esquemáticos como método de desenho universal, sendo estes preteridos, em algumas tarefas, pelo uso de linguagens de alto nível. A natureza estrutural dos esquemáticos escondem a funcionalidade dos elementos representados, ou permitem a sua descrição através de redes complexas de *gates* elementares, de difícil interpretação. Na simulação de ASICs ou na verificação formal de sistemas digitais, a função do componente é mais relevante que a sua estrutura. A funcionalidade é mais facilmente expressa através do respectivo algoritmo, que por sua vez é mais facilmente expresso através de uma linguagem, sobretudo quando a complexidade é elevada.

Das linguagens disponíveis, a VHDL (norma IEEE e ANSI) tem-se revelado como uma das mais interessantes. A sua capacidade de representação a vários níveis permite a sua utilização eficiente em todo o processo de desenvolvimento, desde a especificação até à síntese, e ainda a aplicação de técnicas multinível, que se traduz na coexistência de descrições a vários níveis de abstracção, num mesmo modelo.

O trabalho aqui apresentado resulta do envolvimento do autor no projecto de uma arquitectura heterogénea para visão por computador, baseada num *Array* de processadores com memória associativa (APA) - designado por GLiTCH e especialmente útil para as tarefas de processamento de imagem - e numa rede de transputers, para executar tarefas de mais alto nível. Este projecto foi desenvolvido em conjunto com as Universidades de Bristol e Manchester (UK), ficando a cargo da Universidade do Minho o desenvolvimento da unidade de controlo do APA.

No mesmo sistema foram integrados diversos tipos de ASICs e componentes *standard*. Cada tipo de componente exige metodologias (e ferramentas) de desenvolvimento diferentes, sendo necessário observar todos os factores associados ao processo de desenho, atrás enunciados.

No capítulo 2 é feita uma breve análise das arquitecturas paralelas, com especial incidência naquelas que foram especialmente concebidas para visão por computador. Nesse contexto é apresentada a arquitectura heterogénea proposta, incluindo os detalhes mais relevantes do APA. O resultado da análise feita neste capítulo conduzirá à especificação que serviu de base ao desenvolvimento do controlador do APA.

No capítulo 3 são abordadas algumas das questões relacionadas com a evolução tecnológica dos sistemas digitais, nomeadamente o processo de desenvolvimento - incluindo a síntese de alto nível - as ferramentas de CAD disponíveis, e as tecnologias de implementação. A diversidade das ferramentas de CAD tornaram possível e útil a definição de uma taxinomia. O aparecimento do VHDL como uma linguagem normalizada com capacidades descritivas a vários níveis de abstracção, tem permitido a aplicação de novos paradigmas de desenvolvimento. Estes tópicos são também abordados neste capítulo.

No que respeita às tecnologias de implementação, a evolução das estruturas lógicas programáveis, implementadas em componentes de elevada escala de integração, tem vindo a viabilizar a sua utilização alternativa no projecto de sistemas digitais complexos, mormente na fase de prototipagem, onde introduzem uma flexibilidade elevada que facilita a eliminação de erros. Esta tecnologia de implementação foi amplamente utilizada no projecto do controlador do GLiTCH, sendo por isso pertinente a sua análise detalhada. Este assunto é também abordado no capítulo 3.

No capítulo 4 é apresentado o projecto do controlador do GLiTCH, com realce para as metodologias e problemas relacionados com a construção da descrição em VHDL, as opções tomadas ao nível da selecção de componentes e o conjunto de testes desenvolvidos para o protótipo - realização física e modelo simulável. A testabilidade é um conceito cuja aplicação é tanto mais importante quanto maior é a complexidade do projecto. A aplicação do transputer neste domínio é discutida. Os detalhes da implementação são apresentados em apêndice.

Utilizando como *case study* o controlador desenvolvido, e tendo em vista a aplicação da lógica programável num processo automático de síntese de alto nível, foi realizado um estudo sobre a variação sintáctica - descrições alternativas em VHDL e o seu impacto na síntese - e uma análise comparativa da aplicação de dois componentes de lógica programável, representativos das classes CPLD e FPGA. Esse estudo e as respectivas conclusões encontram-se no capítulo 5.

De uma forma geral, o projecto desenvolvido demonstrou a validade das opções ao nível da testabilidade e da selecção de componentes, evidenciou as vantagens da utilização de uma linguagem como o VHDL - nomeadamente na resolução dos problemas associados à utilização de diferentes tecnologias de implementação e diferentes níveis de abstracção - e permitiu ainda obter algumas soluções para a problemática da aplicação da lógica programável na prototipagem rápida de sistemas digitais.

Capítulo 2

2. Arquitecturas paralelas para visão por computador

2.1 Paralelismo

Desde a introdução dos computadores, no início dos anos 40, a engenharia associada caracterizou-se por uma constante procura de soluções mais rápidas, mais baratas, mais fiáveis e mais eficazes. Tais objectivos exigiram a investigação de tecnologias cada vez mais eficientes, mas muito cedo se concluiu que apenas o desenvolvimento tecnológico impunha limites naturais a essa ambição. Como exemplo, se desejarmos construir uma máquina com circuitos digitais, atendendo a que a velocidade de propagação dos sinais eléctricos é 0.03 cm/ps, qualquer sinal eléctrico levará 300 ps a percorrer 9 cm. Para reduzir a velocidade de execução é necessário explorar outras soluções, como seja o recurso ao paralelismo. De uma forma simples, pode-se esperar que dispondo de **p** processadores a operar em paralelo, seja possível obter um melhoramento no desempenho de perto de **p** vezes face ao que se obtém com um único processador [Wallach 1981].

A utilização de várias unidades funcionais, interligadas de uma forma organizada para executar mais rapidamente determinada tarefa, encontra-se já referenciada em 1842 por Manabrea que, referindo-se às ideias apresentadas por Charles Babbage, considerado como o "pai" dos computadores, escrevia [Menabrea 1842]:

> *"Secondly, the economy of time: to convince ourselves of this, we need only recollet that the multiplication of two numbers, consisting each of twenty figures, requires at the very utmost three minutes. Likewise, when a long series of identical computations is to be performed, ... the machine can be brought into play so as to give several results at the same time, ..."*

Embora esta forma de paralelismo não estivesse incluída na referida máquina analítica projectada por Babbage [Babbage 1864], é bem claro o reconhecimento da sua necessidade.

Desde então várias arquitecturas paralelas têm sido propostas, na maioria dos casos para satisfazer necessidades em aplicações específicas. A actividade de investigação neste domínio, juntamente com as inovações tecnológicas, tem resultado em melhorias de desempenho que aumentam por um factor de 10 em cada 5 anos [Hockney e Jesshope 1988]. Devido à diversidade das arquitecturas paralelas é possível (e desejável) agrupá-las segundo algum conjunto de características. A taxinomia daí resultante é sempre discutível, sobretudo quando é necessário incluir novas arquitecturas para as quais a taxinomia não está perfeitamente adaptada, ou quando se analisa uma mesma arquitectura sob perspectivas diferentes.

2.2 Arquitecturas paralelas

Das taxinomias propostas a mais divulgada é devida a Flynn [Flynn 1972] e baseia-se nas noções de fluxo de instruções e fluxo de dados. Nessa taxinomia definem-se quatro classes: SISD (*Single Instruction stream, Single Data stream*); SIMD (*Single Instruction stream, Multiple Data stream*); MISD (*Multiple Instruction stream, Single Data stream*); e MIMD (*Multiple Instruction stream, Multiple Data stream*). É difícil conceber uma arquitectura pertencente à classe MISD - várias instruções a executar sobre o mesmo conjunto de dados - embora certos autores [Wallach 1981] considerem nesta categoria as arquitecturas em *pipeline*, assumindo que cada dado é processado em estágios consecutivos, cada um obedecendo a uma microinstrução. Em última análise a classe MISD completa a taxionomia de uma forma elegante, ao considerar todas as possíveis relações entre o fluxo de instruções e o fluxo de dados.

Vários autores criticaram esta taxinomia devido à sua incapacidade para classificar objectivamente algumas arquitecturas, tais como as arquitecturas em *pipeline*, que não podem ser diferenciadas segundo aquele conjunto limitado de características, puramente funcionais. Para tal será necessário usar critérios que permitam diferenciar mais pormenores da estrutura de uma arquitectura, enquanto sistema digital [Hockney e Jesshope 1981].

Assim surgiram várias outras taxinomias, como a de Shore [Shore 1973], que propõe seis classes sobre uma estrutura que considera quatro unidades funcionais (unidade de controlo, unidade de processamento, memória de instruções e memória de dados), a de Johnson [Johnson 1988], que subdivide a classe MIMD de acordo com a estrutura de memória (global ou distribuída) e o mecanismo de comunicação (variáveis partilhadas ou *message passing*), ou ainda a de Hockney [Hockney and Jesshope 1981], que propõe uma classificação baseada numa estrutura ainda mais detalhada.

No contexto do trabalho aqui apresentado é conveniente considerar cinco modelos, agrupados em três classes, que serão descritos mais detalhadamente nos parágrafos seguintes, e que se baseiam na notação apresentada por Wolfe [Wolfe e Shen 1991]. Nesta notação considera-se um sistema composto por uma ou mais unidades de controlo e uma ou mais unidades de processamento de dados, relacionadas entre si por canais de comunicação para as respectivas instruções e para as *flags* que codificam os resultados obtidos na(s) unidade(s) de processamento de dados. De acordo com esta estrutura, nas secções seguintes deve ser usada a seguinte legenda:

- **i**: instrução para a unidade de processamento de dados;
- **C**: sequenciador e memória de programa (unidade de controlo);
- **cc**: indicadores (*flags*) dos resultados das operações executadas sobre os dados;
- **s**: instrução da unidade de controlo;
- **T**: unidade de processamento de dados (ALU, registo, memória de dados, etc.).

Classe SISD

Esta classe inclui o modelo convencional proposto por von Neumann, constituído por um único processador [Burks, et al. 1946], e ainda a arquitectura Harvard. Alguns

autores incluem ainda nesta classe algumas arquitecturas em *pipeline*. A figura 2.1 mostra um diagrama de blocos que ilustra a estrutura típica desta classe de arquitecturas.

Classes SIMD

Nesta classe incluem-se as arquitecturas compostas por uma unidade de controlo, que distribui uma instrução por várias unidades idênticas de processamento de dados. É o caso dos processadores vectoriais e dos *array processors*. Alguns autores incluem nesta classe algumas arquitecturas em *pipeline*, em vez de as incluir na classe SISD. A figura 2.2 mostra um diagrama de blocos que ilustra a estrutura típica desta classe de arquitecturas.

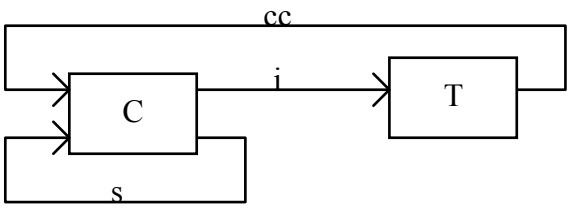


Figura 2.1 Arquitectura SISD

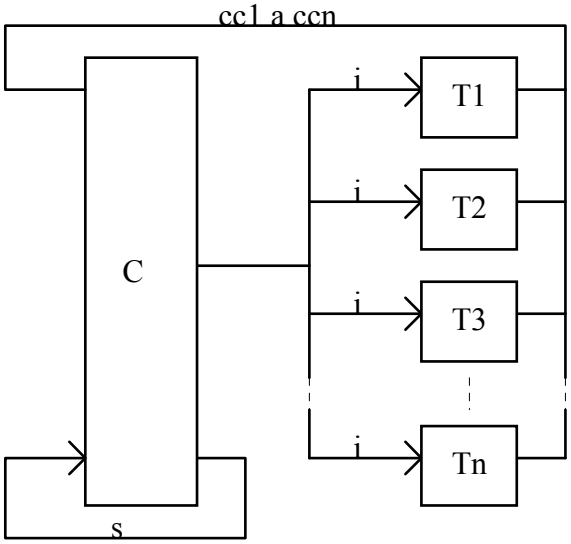


Figura 2.2 Arquitectura SIMD

Um caso particular da classe SIMD é identificado pela existência de unidades diferentes de processamento de dados. Nesse tipo de arquitecturas a unidade de controlo fornece diferentes instruções para as diversas unidades de processamento, conforme mostra o diagrama de blocos da figura 2.3. Alguns autores incluem estas arquitecturas numa classe individual designada por VLIW (*Very Long Instruction Word*) [Fisher 1983]. Como será visto mais adiante a arquitectura de um sistema baseado no GLiTCH enquadra-se na classe SIMD, mas, na perspectiva do controlador, evidencia algumas das propriedades das arquitecturas do tipo VLIW.

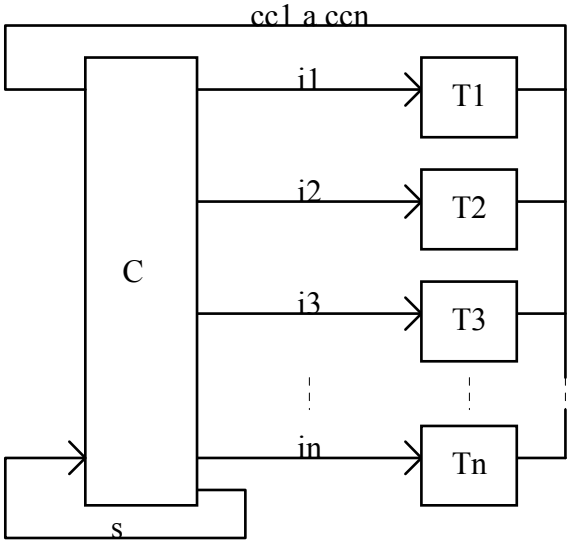


Figura 2.3 Arquitectura VLIW

Classe MIMD

Esta classe inclui todas as arquitecturas constituídas por vários processadores independentes (multiprocessadores), cada um constituído por uma unidade de controlo e uma ou mais unidades de processamento de dados. A figura 2.4 mostra um diagrama de blocos que ilustra a estrutura típica desta classe de arquitecturas. Nessa estrutura, um dos aspectos a realçar é a natureza assíncrona dos processadores, uma vez que não existe nenhuma ligação física entre as respectivas unidades de controlo.

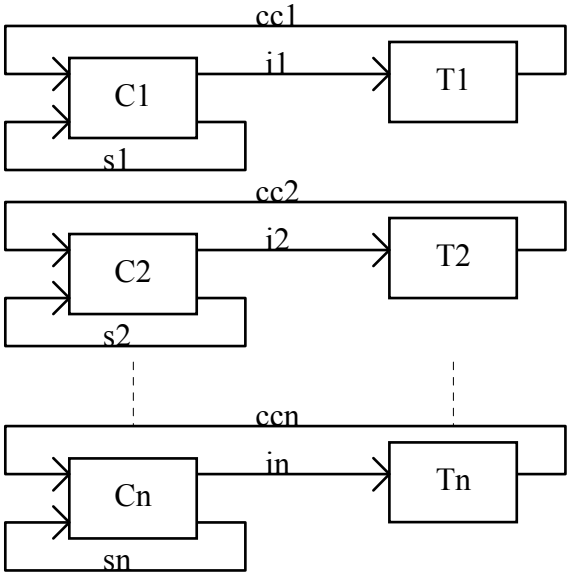


Figura 2.4 Arquitectura MIMD

Em aplicações reais é normalmente desejável relacionar de algum modo os diferentes nós de processamento, que cooperam entre si na resolução de algum problema complexo.

As diferentes topologias que se podem assim obter têm um impacto importante no desempenho do sistema e resultam numa grande variedade de arquitecturas nesta classe. Na taxinomia em análise tais interligações não são consideradas como elementos diferenciadores.

Uma variação interessante da classe MIMD pode ser obtida introduzindo-lhe características da classe SIMD, ao obrigar as diferentes unidades de controlo a executar o mesmo conjunto de instruções. De uma forma resumida, esta organização corresponde a uma estrutura do tipo MIMD, síncrona e fortemente ligada. A figura 2.5 mostra um diagrama de blocos que ilustra a estrutura típica desta classe de arquitecturas. A utilização conjunta de características das classes MIMD e SIMD levaram alguns autores a propor uma classe individual para este tipo de arquitecturas, designada por XIMD [Houzet 1992].

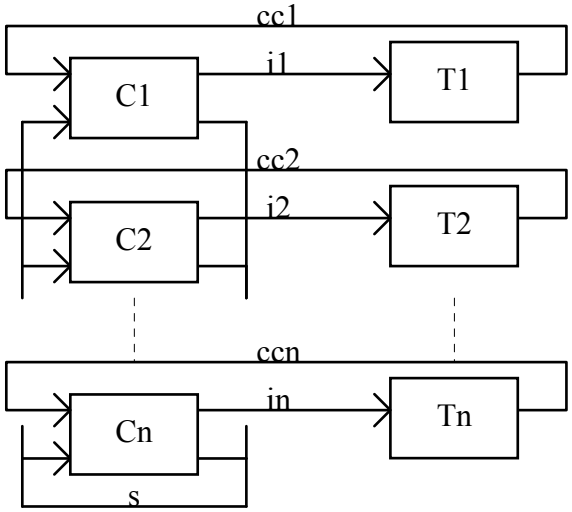


Figura 2.5 Modelo XIMD

As arquitecturas da classe MIMD são vulgarmente utilizadas para implementar tarefas de alto nível em sistemas para visão por computador. A arquitectura heterogénea baseada no GLiTCH baseia-se também neste modelo, implementando cada um dos nós **Cn** com subsistemas construídos com transputers. Um desses nós corresponde ao módulo designado por sistema GLiTCH, especialmente concebido para a implementação de tarefas de processamento de imagem.

2.3 Arquitecturas para visão por computador

No domínio da visão por computador e sobretudo considerando a sua aplicação em tempo real, os processos são caracterizados por exigirem uma potência de cálculo elevada, devido à elevada quantidade de dados a processar, e um conjunto de algoritmos mais ou menos adaptados a diferentes tipos de arquitecturas paralelas. Para facilitar a análise deste tipo de aplicação é vulgar identificar pelo menos três níveis de processamento [Proença 1993]. Ao nível mais baixo cada ponto da imagem (pixel) é tratado como um elemento de uma estrutura 1D ou 2D. As operações são simples e repetitivas e envolvem cada pixel individualmente ou considerando-os numa vizinhança muito limitada. São exemplos os

filtros, correcções geométricas e detecção de contornos. A este nível os processadores de sinais digitais (DSP) e as arquitecturas SIMD (*fine-grain*) constituem os sistemas que, tipicamente, permitem obter os melhores desempenhos.

No nível de processamento intermédio os pixels são considerados em grupos e o objectivo é construir estruturas de dados (informação simbólica) que descrevam determinadas propriedades da imagem. Incluem-se neste nível as operações de segmentação e extracção de determinadas características de imagens. A este nível, e de uma forma genérica, os melhores desempenhos são novamente garantidos pelas arquitecturas SIMD. O nível de processamento mais elevado utiliza aquelas estruturas de dados como base de processamento, através das quais interpreta as imagens de acordo com o objectivo específico da aplicação. A este nível são utilizados algoritmos complexos e não determinísticos, para os quais as arquitecturas SIMD apresentam fraco desempenho, sendo preteridas pelas arquitecturas MIMD (*medium-grain* ou *coarse-grain*).

Como resultado de longos anos de investigação, é bastante elevado o número de arquitecturas propostas para visão por computador [Maresca, et al. 1988]. A maioria dessas arquitecturas baseiam-se apenas numa das classes de arquitecturas SIMD ou MIMD para executar todos os algoritmos necessários. Contudo, mais recentemente, têm sido sugeridas algumas arquitecturas heterogéneas que, globalmente, apresentam melhores resultados, face à heterogeneidade dos algoritmos usados em visão. Como exemplos desses sistemas podem-se referir o UMass IUA [Weems, et al. 1992], o VisTA [Sunwoo e Aggarwal 1990a] e o HVA [Storer, et al. 1992] (*Heterogeneous Vision Architecture* - baseado no GLiTCH e numa rede de transputers, cuja arquitectura será analisada em maior detalhe posteriormente).

No que respeita às arquitecturas SIMD existem alguns parâmetros determinantes para o desempenho (e que naturalmente têm sido devidamente estudados nas diversas soluções propostas): a comunicação entre os PEs (*Processing Elements*) e o esquema de I/O para os dados a processar. Das diversas topologias possíveis a mais vulgar corresponde a um arranjo bidimensional, em que cada PE comunica directamente com cada um dos seus quatro vizinhos (topologia habitualmente designada por *two-dimensional mesh-connected*).

As arquitecturas SIMD são particularmente importantes no contexto do trabalho aqui apresentado, já que é nessa classe que o GLiTCH se enquadra. Por esse motivo, nas secções seguintes vão ser detalhadas algumas das arquitecturas propostas, numa perspectiva histórica, com o objectivo de identificar as principais características desta classe de arquitecturas e que mais condicionam a sua aplicação.

2.4 Arquitecturas SIMD para processamento de imagem

A lei de Amdahl estabelece um limite para a aplicação de máquinas paralelas a problemas reais, ao demonstrar que as partes de um programa que não podem ser implementadas em paralelo limitam a velocidade de execução, podendo assim injustificar a utilização dessas máquinas paralelas [Quinn 1988]. Na prática isto significa que a aplicação de máquinas paralelas deve restringir-se aos algoritmos com componentes sequenciais mínimas. Outro factor importante a condicionar a utilização de máquinas

paralelas é a adequação do tipo de paralelismo do algoritmo ao tipo de paralelismo da máquina.

Intuitivamente pode-se afirmar que as arquitecturas SIMD estão melhor adaptadas a problemas em que grandes quantidades de dados requerem o mesmo processamento e a interdependência entre os dados é reduzida. Grande parte dos algoritmos de processamento de imagem e reconhecimento de padrões pertencem a esta classe.

Os fundamentos teóricos associados à utilização de máquinas da classe SIMD remontam a 1952, quando von Neuman, após propor o primeiro computador baseado no armazenamento de programas, propôs uma arquitectura paralela baseada num vector de autómatos celulares, cada um com 29 estados, dispostos num arranjo bidimensional e capazes de simular uma máquina de Turing [Neumann 1966]. Desde então registaram-se vários desenvolvimentos, alguns deles relacionados com a aplicação desta classe de arquitecturas ao processamento de imagem, como é o caso da arquitectura proposta por Unger, em 1958 [Unger 1958] e [Unger 1959]. Esta arquitectura teve a particularidade de sugerir, pela primeira vez, a utilização de uma única unidade de controlo a executar um programa para comandar todos os PEs (*Processor Elements*).

SOLOMON

Um dos marcos mais importantes na história dos *Array Processors* foi a arquitectura proposta por Slotnick em 1962, e designada por SOLOMON [Slotnick, et al. 1962]. A sua importância é evidenciada por R. Hockney em [Hockney and Jesshope 1981], ao usar como título numa secção: *"SOLOMON and after"*. O SOLOMAN é composto por uma matriz de 32x32 ALUs, a operar no modo série, interligadas num arranjo bidimensional e cada uma com 4K bits de memória. Esta memória está organizada em duas *stacks*, cada uma com capacidade para 64 operandos de 32 bits e com um ponteiro de pilha, que permite o acesso ao nível do bit. Outra característica interessante é a disponibilidade de 4 bits de "estado" em cada PE, que permitem controlar as situações em que cada PE deve ou não executar uma instrução. Tendo sido planeada para manusear grandes quantidades de informação, as suas características de I/O foram cuidadosamente planeadas. O sistema inclui discos duros e permite transferências de dados a partir de qualquer um dos quatro lados da matriz e de qualquer linha ou coluna - esta característica é habitualmente designada por *geometric control*.

O SOLOMAN nunca foi construído, por razões de ordem técnica, nomeadamente devido à elevada complexidade da sua unidade de controlo e devido ainda a limitações da própria arquitectura na aplicação à resolução dos problemas para os quais tinha sido proposto. Este último aspecto levou à proposta de uma versão optimizada (SOLOMAN II), dois anos mais tarde, que incluia em cada PE a capacidade de operar números representados em vírgula flutuante (24 bits) e mais alguns registos. A organização da memória também foi modificada para uma única área de memória RAM.

Os trabalhos de investigação com o SOLOMAN levaram à proposta de algumas arquitecturas importantes, como o ILLIAC IV, o ICL DAP, o PEPE da Burroughs e o STARAN da Goodyear Aerospace. Em seguida será feita uma breve abordagem de algumas dessas arquitecturas, com o objectivo de identificar os problemas mais relevantes que se colocam à engenharia deste tipo de sistemas.

ILLIAC IV

O ILLIAC IV foi proposto em 1967, também por Slotnick, motivo pelo qual apresenta várias semelhanças com o SOLOMON [Slotnick 1967], [Barnes e Slotnick 1968] e [Bouknight, et al. 1972]. A arquitectura inicialmente proposta consistia de quatro quadrantes, cada um com uma matriz de 8x8 PEs, num arranjo bidimensional (*mesh-connected*) e com uma unidade de controlo independente. Esta característica permite a cada quadrante executar diferentes programas e/ou subprogramas, ou então executar em conjunto, como um único vector de processadores. Os quadrantes são interligados por barramentos paralelos, entre si e com um sistema de discos, utilizado para armazenar os programas e os resultados. Cada PE inclui uma ALU de 64 bits, com suporte para operações em vírgula flutuante, uma memória local de 2K palavras e, pela primeira vez, um registo de indexação. A memória foi desenhada para poder ser utilizada como memória local, ao nível dos operandos, e como memória de programas, ao nível do quadrante.

A unidade de controlo do ILLIAC IV é um processador escalar, com a capacidade de controlar o *array* de PEs, de uma forma concorrente. Executa operações escalares sobre operandos de 64 bits (operações lógicas, adição, subtracção e deslocamentos) ficando as operações aritméticas mais complexas para o *array* de PEs. Dispõe de armazenamento local (*cache*) para dados e instruções e um bloco de lógica que descodifica as instruções para o *array*, gerando os respectivos operandos e sinais de controlo [Hwang e Briggs 1989] e [Wallach 1981].

Embora apenas um quadrante tenha sido efectivamente construído e com um desempenho de 50 Mflop/s - uma diminuição acentuada face ao desempenho de 1 Gflop/s esperado para o sistema completo - o ILLIAC IV pode ser considerado como o primeiro vector de processadores efectivamente construído. Foi utilizado em diversas aplicações, tais como meteorologia (na NASA[1]), cálculo matricial e resoluções da transformada de Fourier. Os resultados obtidos validaram esta classe de arquitecturas e marcaram o ponto de partida para diversos trabalhos de investigação, tanto ao nível das aplicações como das arquitecturas. No que respeita à tecnologia utilizada, o ILLIAC IV também foi pioneiro ao utilizar semicondutores para a implementação da memória, lógica ECL e placas de circuito impresso com 15 camadas, que exigiram a utilização de ferramentas automáticas a um nível de complexidade considerável [Hockney and Jesshope 1981].

BSP

A Burroughs Corporation foi uma das empresas envolvidas no trabalho de engenharia do ILLIAC IV e, com o conhecimento adquirido, desenvolveu o seu próprio sistema, o Burroughs BSP. Esta máquina tem apenas 16 PEs, 17 bancos de memória e um esquema de interligações que permitia a cada PE aceder a qualquer um dos bancos de memória. Esta arquitectura procurava uma melhoria de desempenho, face ao ILLIAC IV, através da optimização do fluxo de dados no vector. O BSP não foi desenhado como um sistema independente, transferindo as tarefas de gestão de I/O para um hospedeiro. À semelhança do ILLIAC IV, a unidade de controlo também é implementada por um processador escalar, o qual transforma as instruções vectoriais em sequências de

[1] *National Aeronautical and Space Administration*

microinstruções para o *array*. O BSP foi construído mas não chegou a ser comercializado [Hwang and Briggs 1989].

ICL DAP

O ICL DAP é bastante semelhante ao SOLOMAN, sendo originalmente formado por uma matriz de 32x32 PEs, a operar no modo série, com 1 a 4K bits de RAM e um esquema de interligações bidimensional. Tal como no ILLIAC, também neste caso a memória local dos PEs era usada para armazenar programas. As características mais inovadoras deste sistema foram a sua flexibilidade no acesso à memória do *array*, a partir do exterior (através de barramentos de elevada capacidade) e o facto de ser utilizado como emulador de um módulo de memória de um computador de grande porte, o ICL, para além de poder também executar num modo autónomo [Hockney and Jesshope 1988] e [Reddaway 1978]. Desde o seu aparecimento, esta arquitectura tem sofrido vários melhoramentos, sobretudo graças ao avanço tecnológico. Na sua versão actual (1992) o *array* é composto por 64x64 PEs, cada um com 16 a 64K bits de RAM, ainda a operar no modo série, mas com a possibilidade de instalar uma unidade de processamento para representações em vírgula flutuante - FPU (*Floating Point Unit*) - por cada grupo de 16 PEs, podendo aumentar assim o desempenho do DAP, sobretudo em aplicações que exijam este tipo de recurso. O DAP foi a arquitectura do tipo SIMD que mais tempo perdurou [Thomson 1994].

CLIP

O CLIP (*Cellular Logic Image Processor*) é um exemplo de outra arquitectura que também passou por várias alterações. Como o próprio nome indica, esta arquitectura foi especialmente desenvolvida para aplicações de processamento de imagem. Inicialmente dispondo de PEs a operar no modo série e com muita pouca quantidade de memória, nesta arquitectura procurou-se explorar formas alternativas de conectividade, especialmente adaptadas às características das imagens e à natureza dos algoritmos. O CLIP3 tinha a particularidade de se poder configurar em geometrias hexagonal e octogonal, em que cada PE podia comunicar com, respectivamente, 6 e 8 vizinhos. A comutação entre geometrias era feita por programação. No entanto, a geometria hexagonal não produziu os resultados esperados, tendo sido preterida, no CLIP4, apenas pelo arranjo octogonal. Nesta última versão aumentou o número de PEs, que passaram a constituir um *array* de 96x96, bem como a quantidade de memória por PE, que passou para 32 bits [Duff 1976].

A mais recente versão desta arquitectura, o CLIP7, é já fabricado em VLSI. Cada circuito integrado contém um PE de 16 bits, vários registos de 8 e 16 bits e oito canais de comunicação externos, que permitem a implementação de várias topologias a 2, 3 e 4 dimensões. A memória é externa e pode ocupar o espaço máximo de 32K bytes. A principal característica deste processador é a utilização de um registo de condição que lhe confere alguma autonomia no comportamento face a uma instrução difundida pela unidade de controlo - por exemplo, permite modificar as operações da ALU e determinar a direcção da comunicação com os PEs vizinhos. Esta característica revelou-se muito útil na programação e no desenvolvimento de algoritmos [Fountain, et al. 1988] e [Fountain 1986].

MPP

Outra arquitectura da classe SIMD desenvolvida para processamento de imagem foi o MPP (*Massively Parallel Processor*). Foi desenvolvida para a NASA pela *Goodyear Aerospace*, para análise de imagens de satélite, e o seu nome advém do elevado número de PEs utilizado, 128 x 132 = 16.896, embora apenas 128 x 128 fossem utilizados, ficando os restantes como reserva. O MPP tem uma unidade de controlo microprogramável, flexibilizando assim a definição do conjunto de instruções. Cada PE é constituído por um microprocessador de um bit, com 1K bit de RAM, um registo de condição (à semelhança do que foi atrás referido na descrição do CLIP7), um somador completo e um registo de deslocamento programável. O *array* tem um arranjo bidimensional, comunicando cada PE com cada um dos seus quatro vizinhos. Os canais exteriores dos PEs localizados sobre os lados do *array* podem, por programação, ficar desligados ou serem interligados formando logicamente um cilindro sobre a vertical ou sobre a horizontal, ou ainda sobre ambas as direcções, a que corresponde a topologia habitualmente designada por *torus*. Esta característica fornece ao programador maior flexibilidade na escolha da melhor topologia para a aplicação pretendida. Outra característica importante consiste na possibilidade dos registos de I/O (memória MDA - *multidimensional access*) poderem reformatar os dados quando as imagens são introduzidas ou extraídas do *array*. Esta técnica tinha sido já introduzida na arquitectura do STARAN (abordada mais à frente). O MPP dispõe de um minicomputador para gestão dos dados e programas e pode funcionar sob controlo de um outro sistema, ou no modo *stand-alone*, interactuando com o utilizador através de um terminal e comunicando com sistemas de armazenamento de massa através de barramentos de alta velocidade [Hwang and Briggs 1989] e [Batcher 1985].

CM-1

Uma arquitectura importante e que explora um topologia diferente da bidimensional, em que a maioria das arquitecturas do tipo SIMD se baseiam, é a *Connection Machine* - CM-1. É um sistema de computação que contém 64K PEs de 1 bit, com 4K bits de memória cada e a operar do modo série. Num único circuito integrado VLSI estão implementados 16 PEs, interligados num arranjo bidimensional *mesh-connected*, e um *router*. Todos os *routers* estão interligados numa rede do tipo *packet-switched* baseada num topologia com **n** dimensões, habitualmente designada por hipercubo.

Cada módulo (uma placa de circuito impresso) contém 32 dos circuitos integrados referidos e a respectiva memória, formando as primeiras 5 dimensões. Grupos de 16 módulos são ligados num *backplane*, formando mais 4 dimensões. Cada conjunto de dois grupos é montado num *rack* e quatro *racks* são agrupados na forma de um cubo, implementando as restantes 3 dimensões. Uma parte das ligações é feita directamente sobre os circuitos impressos, mas cada uma das arestas do "cubo exterior" corresponde a 8.192 pares de conductores. Consequentemente, este esquema de interligações coloca dificuldades acrescidas à realização prática desta arquitectura. A unidade de controlo interpreta microinstruções que lhe são passadas por um hospedeiro, convertendo-as em sequências de nanoinstruções que são executadas pelos PEs [Hillis 1985].

PAPIA

A última arquitectura a considerar nesta secção é designada por PAPIA, sendo de realçar o invulgar esquema de interligações entre PEs, em pirâmide. Esta arquitectura foi

desenvolvida com base num circuito integrado VLSI com 5 PEs, organizados na forma de uma pirâmide, com um PE no topo e quatro na base. Os PEs são dispositivos de 1 bit e incluem um adicionador completo, dois registos de deslocamento, um registo de activação e 256 bits de memória local. O principal problema deste circuito integrado é o elevado número de ligações externas necessárias para efectuar a interligação com PEs vizinhos, o que obrigou ao recurso da técnica de multiplexagem [Cantoni, et al. 1985].

2.5 Arquitecturas SIMD com memória associativa

Memória associativa

O mecanismo associativo, identificado desde à muito como uma base importante das teorias que procuram descrever o funcionamento da mente humana, pode ser utilizado com vantagens em sistemas de computação, particularmente em aplicações cujos dados são seleccionados mais naturalmente por associação do que por indexação. Contudo, a implementação eficiente deste mecanismo não é uma tarefa fácil, sobretudo recorrendo às estruturas lógicas desenvolvidas à volta dos conceitos de unidade de controlo e memória, introduzidos por von Newmann. Neste domínio, um passo decisivo foi a descoberta da função dos neurónios no sistema nervoso animal, a qual serviu de guia à forma como o mecanismo associativo poderia ser construído [Storer 1991a].

Os primeiros computadores foram concebidos para executar operações aritméticas ou lógicas sobre dados cuja única característica relevante era o seu valor actual. Com a divulgação dessas máquinas outros domínios de utilização começaram a ser investigados, desde as aplicações militares e científicas até às aplicações comerciais. Muitas dessas aplicações continuavam a exigir apenas máquinas de cálculo automático, mas outras começaram a exigir que os computadores lidassem com dados não numéricos, tais como nomes e listas de produtos. Assim começaram a ter cada vez mais importância tarefas tipicamente associativas, como a consulta de bases de dados a partir de uma determinada chave ou de uma propriedade comum aos seus elementos.

As operações associativas procuram relações entre um operando e todos os dados armazenados numa memória. Isto é algo que num sistema baseado na arquitectura clássica de von Newmann só pode ser feito se essas relações forem expressas aritmeticamente, transformando-as em operações sobre endereços e utilizando algoritmos adequados, o que, em muitos casos é bastante ineficiente. Uma alternativa interessante, que tem sido usada sobretudo em sistemas dedicados, é a memória designada por CAM (*Content Addressable Memory*). Introduzida em 1956 por McMahon - apenas com capacidade de 20 bits - este tipo de memória tinha a desvantagem de ser particularmente exigente em termos tecnológicos. Desde então foram investigadas várias soluções, mas apenas em 1963 foi implementado o primeiro computador que utilizou CAM. Este percurso encontra-se sumariamente descrito em [Thomson 1994].

Um bloco de CAM tem a organização mostrada na figura 2.6. Cada linha da memória forma uma palavra da CAM e cada célula armazena um bit. A função da máscara é a de determinar se o correspondente bit do operando deve ou não ser utilizado numa operação associativa. Na operação básica de associação todas as palavras da CAM comparam o seu conteúdo com o do registo do operando, tendo em conta o valor da

máscara. As palavras da CAM para as quais a verificação for bem sucedida activam, por exemplo a ‘1’, o sinal associado à linha, enquanto que as restantes ficam, nesse caso, com esse mesmo sinal a ‘0’. O resultado é armazenado no registo de resposta.

As palavras da CAM podem ser logicamente divididas em campos, de acordo com uma aplicação particular, para representar determinados atributos, servindo então a máscara para seleccionar o campo (ou campos) a utilizar em cada operação.

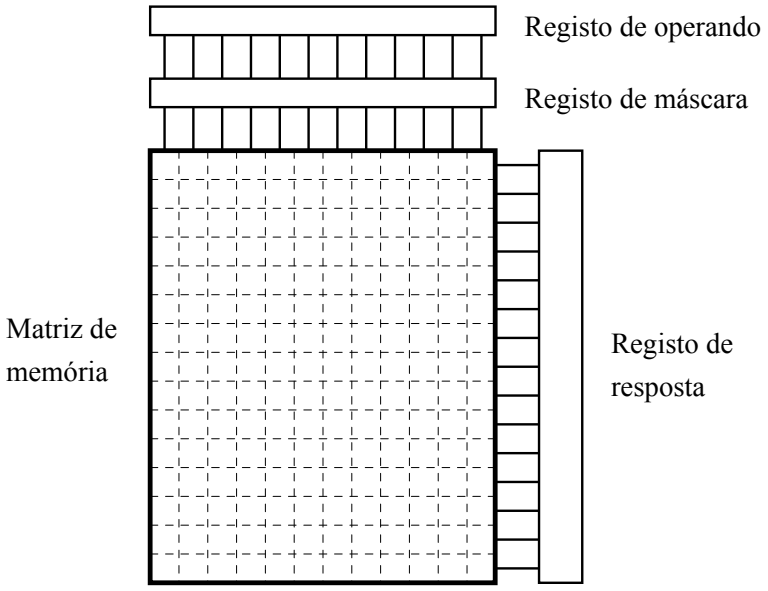


Figura 2.6 Organização típica de uma CAM

A operação de associação é executada sobre todas as palavras em simultâneo (*word-parallel*), mas, ao nível da palavra, tanto pode ser executada de uma só vez (*bit-parallel*), ou processando um bit de cada vez (*bit-serial*). Neste último caso uma coluna da CAM é processada em cada iteração e o resultado dessa coluna é guardado num acumulador. Nas iterações subsequentes é feito o produto lógico do conteúdo do acumulador com o resultado da associação na coluna seguinte, sendo novamente guardado o resultado final no acumulador. Quando todas as colunas forem processadas o conteúdo do acumulador é transferido para o registo de resposta. Uma CAM que utilize o modo *bit-serial* tem a vantagem de necessitar de uma estrutura de *hardware* mais simples e mais compacta, mas à custa de um tempo de operação bastante maior.

Para além da operação de verificação anteriormente descrita, é necessário executar várias outras operações, tal como a escrita. A maioria dessas operações utiliza o resultado de uma operação de verificação anterior, que selecciona o conjunto de palavras da CAM activas. Nestes casos o conteúdo do registo de máscara é usado para delimitar um ou mais campos a serem operados, em cada palavra activa. Dependendo do tipo de operação é possível existirem dois modos de execução. O modo *word-parallel* em que todas as palavras são processadas em simultâneo, como na operação de verificação anteriormente descrita; e o modo *word-serial* em que é processada uma palavra de cada vez. Neste último caso será necessário existir alguma função lógica de resolução que permita escolher uma palavra do conjunto das activas. Como exemplo, tal função pode utilizar o conteúdo do registo de resposta para escolher primeiro a palavra mais ao “topo” e, após a execução da

operação sobre essa palavra, desactivar o respectivo bit no registo de resposta. As restantes palavras activas serão processadas, repetindo a operação até que o conteúdo do registo de resposta indique que não há mais palavras activas.

A CAM pode ser utilizada para executar operações aritméticas e lógicas, bem como para outras operações tipicamente necessárias em processamento de imagem e reconhecimento de padrões, sendo por vezes necessário recorrer a alguns artifícios de programação incluindo a simulação de uma memória indexável. Algumas dessas operações e possíveis implementações encontram-se descritas em [Storer 1991a].

A implementação da CAM é limitada pela sua complexidade. Uma CAM construída com semicondutores ocupa bastante mais espaço do que um RAM e consome mais energia. Por esse motivo, na maioria dos sistemas que utilizam CAM, esta é obtida à custa de RAM adicionando-lhe a lógica necessária para efectuar as operações de associação, no modo *bit-serial*. Neste tipo de solução o tempo de acesso à CAM pode comprometer o ganho de desempenho que seria de esperar pela utilização da memória associativa.

Os avanços na área de VLSI tornaram possível a realização de memórias CAM a executar no modo *bit-parallel*. Esta solução permite implementar uma CAM consideravelmente mais rápida mas de baixa capacidade e justifica-se sobretudo para aplicações que não exijam um tamanho de palavra elevado.

Dada a natureza paralela das operações associativas - execução simultânea sobre todas as palavras de memória - a utilização da CAM justifica naturalmente o recurso a arquitecturas paralelas, nomeadamente às da classe SIMD, associando um PE a cada palavra da CAM [Wallach 1981]. A máquina assim obtida é vulgarmente designada por *Associative Processor Array* (APA). À semelhança das máquinas SIMD, cada PE é tipicamente composto por uma palavra de memória associativa, uma ALU e alguma lógica adicional que lhe permite processar as respostas a operações de associação e comunicar com um ou mais PEs vizinhos. Todos os PEs executam uma mesma instrução fornecida por uma unidade de controlo comum.

Embora várias arquitecturas tenham sido propostas poucas foram realizadas, mormente devido a limitações tecnológicas. Como exemplos bem sucedidos podem-se referir três desses sistemas: STARAN, PEPE e SCAPE. Os dois primeiros usam RAM como memória associativa. O STARAN executa no modo *bit-serial*, o SCAPE executa no modo *bit-parallel* e o PEPE utiliza uma variante deste último modo, normalmente designada por *destributed logic*, que consiste em implementar a lógica de suporte às operações associativas ao nível de um grupo de bits. Assim reduz-se o número de sinais de verificação gerados pela memória associativa. Para além desta diferença, o SCAPE utiliza uma estrutura lógica dedicada para implementação da CAM e assume particular importância porque foi desse projecto que nasceu o GLiTCH.

STARAN

O STARAN [Batcher 1974], [Rudolph e Batcher 1985] e [Wallach 1981], foi projectado para aplicações de processamento de imagem em tempo-real, envolvendo o interface com uma vasta gama de sensores, computadores convencionais, processadores de sinal, monitores interactivos e dispositivos de armazenamento de massa. Por esse motivo, uma das suas principais características é a sua complexa estrutura de barramentos que,

entre outras facilidades, permite que a memória associativa seja acedida externamente (pelo hospedeiro) por palavras, no modo paralelo.

O STARAN pode conter até 32 módulos, cada um com 64K bits de memória RAM, organizados em 256 palavras de 256 bits cada e 256 PEs de 1 bit cada. Contrariamente ao que acontece na maioria dos outros sistemas, os PEs não se encontram ligados directamente à memória, mas sim através de um bloco designado por *FLIP network*, capaz de efectuar permutações durante a transferência de dados de uma coluna da CAM para os PEs, e vice-versa. Para além disso é ainda possível efectuar deslocações de dados entre PEs. As operações de associação são feitas no modo *bit-serial*. Todos os módulos existentes partilham uma única unidade de controlo e uma memória para dados e programas. A unidade de controlo é particularmente poderosa, assemelhando-se a um processador escalar convencional - suporta uma vasta gama de operações aritméticas, lógicas, de controlo e de I/O.

PEPE

O PEPE[2] foi projectado para uma aplicação específica em tempo-real, o seguimento de objectos em imagens de radar para controlo de mísseis antibalísticos. Na solução adoptada, um hospedeiro delega em cada PE a função de seguimento de um objecto, na qual o mecanismo associativo é utilizado com vantagens. A aplicação exige muito pouca comunicação entre PEs, mas obriga a que estes tenham uma capacidade de processamento elevada.

Efectivamente, cada PE é constituído por 32K bytes de memória organizada em palavras de 32 bits, uma ALU de 32 bits e suporte de operações em vírgula flutuante, uma unidade de correlação (CU) e uma unidade associativa de saída (OAU). Três unidades de controlo comums, implementadas por processadores escalares convencionais, controlam as unidades anteriores, criando uma estrutura que mais se assemelha a três máquinas SIMD interligadas por um hospedeiro comum e pelas unidades de controlo. O conjunto de instruções das unidade de controlo da CU e da OAU é rico em instruções associativas do tipo "*match*" e "*compare*". O sistema construído dispunha de 288 PEs, fisicamente dispostos em 8 módulos de 36 PEs cada. Esta máquina "gigantesca" foi utilizada com o fim especifico a que se destinava, mas nunca foi comercializada [Hwang and Briggs 1989] e [Wallach 1981].

SCAPE

O SCAPE[3] começou a ser desenvolvido no final dos anos 70 como um componente orientado para o processamento de *strings* de texto, embora outras aplicações fossem também consideradas. Vários refinamentos têm sido introduzidos pelos seus autores e actualmente é comercializado para aplicações de processamento de sinal e de imagem. O SCAPE é constituído por 256 PEs de 1 bit, cada um com 32 bits de memória CAM *bit-parallel*, tudo isto contido num único circuito integrado VLSI. Os PEs estão dispostos num arranjo unidimensional, o que permite reduzir o número de pinos do circuito. Vários

[2] *Parallel Ensemble of Processing Elements*
[3] *Single Chip Array Processing Element*

destes circuitos podem ser interligados para formar um vector com um elevado número de PEs [Lea 1986].

O SCAPE introduziu algumas características inovadoras, das quais se salientam a existência de dois sinais de *match* de cada linha da CAM para a ALU respectiva (o que permite aumentar a velocidade das operações aritméticas - *bit-serial* - atendendo a que é possível aceder aos dois operandos simultaneamente) e a utilização de cinco bits da CAM para identificação de subconjuntos de PEs, podendo assim logicamente dividir os PEs em, no máximo, 32 subconjuntos.

2.6 Arquitectura heterogénea baseada no GLiTCH

2.6.1 O GLiTCH

O GLiTCH é um APA implementado em VLSI, que sucedeu ao SCAPE e destina-se principalmente a aplicações de processamento de imagem e visão por computador. O projecto do GLiTCH foi precedido de um estudo sobre algumas arquitecturas alternativas e uma análise do código desenvolvido para implementar alguns algoritmos no SCAPE, o qual permitiu identificar as principais dificuldades de programação e os estrangulamentos impostos ao fluxo da informação responsável pela limitação da velocidade de operação, nomeadamente o carregamento de imagens e a comunicação entre PEs [Storer 1987] e [Duller, et al. 1987]. A arquitectura resultante foi designada (numa atitude optimista) por "*Goes Like the Clappers Hopefully*" [Duller, et al. 1987], e um simulador foi desenvolvido [Storer 1991b] e [Storer 1991a].

O estudo inicial permitiu identificar os parâmetros que mais condicionam o desempenho deste tipo de arquitectura:

- o número de PEs incluídos num único circuito integrado;
- o tipo e quantidade de memória para cada PE;
- a largura de banda dos canais de entrada/saída e número de pinos;
- a conectividade entre PEs, dentro e fora do circuito integrado; e
- a unidade de controlo.

Por forma a poder avaliar o efeito de alguns destes parâmetros o simulador permite a sua variação, alterando assim a estrutura da arquitectura. O impacto destes parâmetros foi estudado para vários algoritmos, incluindo a equalização de histogramas, convolução e transformada de Hough. Os resultados obtidos encontram-se publicados em [Duller, et al. 1989a], [Duller, et al. 1989c] e [Duller, et al. 1989d], e validaram os princípios que levaram à definição do GLiTCH, nomeadamente a adequabilidade da arquitectura às tarefas a executar e a melhoria de desempenho esperada.

O simulador foi desenvolvido tendo como principal preocupação o *array* de processadores com memória associativa, relegando para um segundo plano algumas unidades que podem influenciar o desempenho do sistema. Refira-se, como exemplo, a unidade de controlo, a qual, apesar da reconhecida importância, é em muitos aspectos definida implicitamente pelas estruturas de controlo da linguagem de programação usada ('C') na construção do simulador. Ainda outro exemplo é a unidade de armazenamento de operandos e máscaras a utilizar com a memória associativa, que no simulador é modelada

através do uso de constantes da linguagem, mas que na prática terá que estar associada a um bloco de memória. Estas questões serão abordadas oportunamente, quando for analisada a implementação da unidade de controlo desenvolvida para o GLiTCH.

Na configuração actual um circuito integrado denominado GLiTCH inclui 64 PEs, cada um composto por uma ALU de 1 bit, 68 bits de memória associativa, dos quais 4 são utilizados para classificação de subconjuntos de PEs (tal como no SCAPE) e um registo de deslocamento, com a largura de 8 bits, para introduzir no *array* a informação de vídeo a processar (VSR - *Video Shift Register*). O VSR é autónomo relativamente ao *array*, pelo que é possível carregar um bloco de uma imagem enquanto um outro está a ser processado. Uma operação específica permite trocar o conteúdo do VSR pelo conteúdo de uma dada secção da CAM. Os PEs estão organizados num arranjo unidimensional. Para facilitar a comunicação entre PEs distantes foi implementado um mecanismo de deslocamento programável (*barrel shifter*), sob a estrutura da CAM, e que permite a comunicação "directa" entre PEs separados por múltiplos de quatro posições. A figura 2.7 mostra a arquitectura interna do GLiTCH através de um diagrama de blocos. A sua descrição pormenorizada encontra-se em [Duller, et al. 1989b] e [Duller, et al. 1989c].

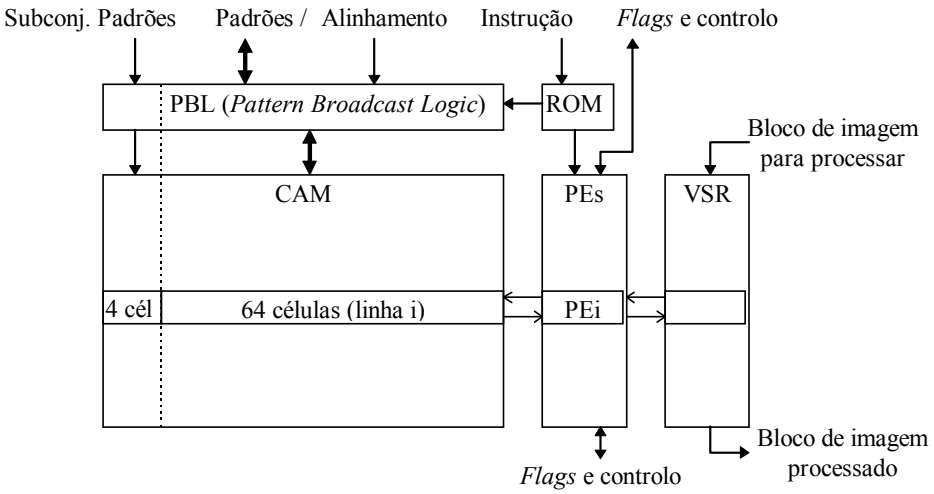


Figura 2.7 Arquitectura interna do GLiTCH

Os sinais de controlo e as *flags* que interligam os PEs internamente formam registos de deslocamento e estão disponíveis na extremidade do *array*, associados aos PEs nos topos, permitindo assim a construção de um *array* de maiores dimensões. A única dificuldade desta integração resulta da necessidade de interligar dinamicamente os sinais de entrada e saída das CAMs (através do PBL - ver figura 2.7) de cada um dos circuitos integrados, entre si e a um barramento global para transferências com a CAM. Para esta função foi desenvolvido um circuito específico, designado por DRN (*Data Routing Network*), o qual permite interligar 4 circuitos integrados GLiTCH para formar um *array* de 256 PEs, constituindo assim aquilo que se pode designar por um módulo GLiTCH. O próprio DRN tem a possibilidade de se interligar com outros DRNs para construir *arrays* de maiores dimensões.

Os resultados da utilização do simulador demonstraram a ineficiência do GLiTCH para algumas operações simples orientadas à palavra (normalmente valores constantes) e a necessidade de dispor de memória indexável (RAM) associada a cada PE, nomeadamente para guardar resultados intermédios. Tendo em conta estas considerações, o sistema GLiTCH foi especificado com base no diagrama de blocos da figura 2.8. Apesar de o protótipo em construção incluir apenas um módulo GLiTCH, deve-se realçar a possibilidade de replicar esse módulo para construir *arrays* com múltiplos de 256 PEs.

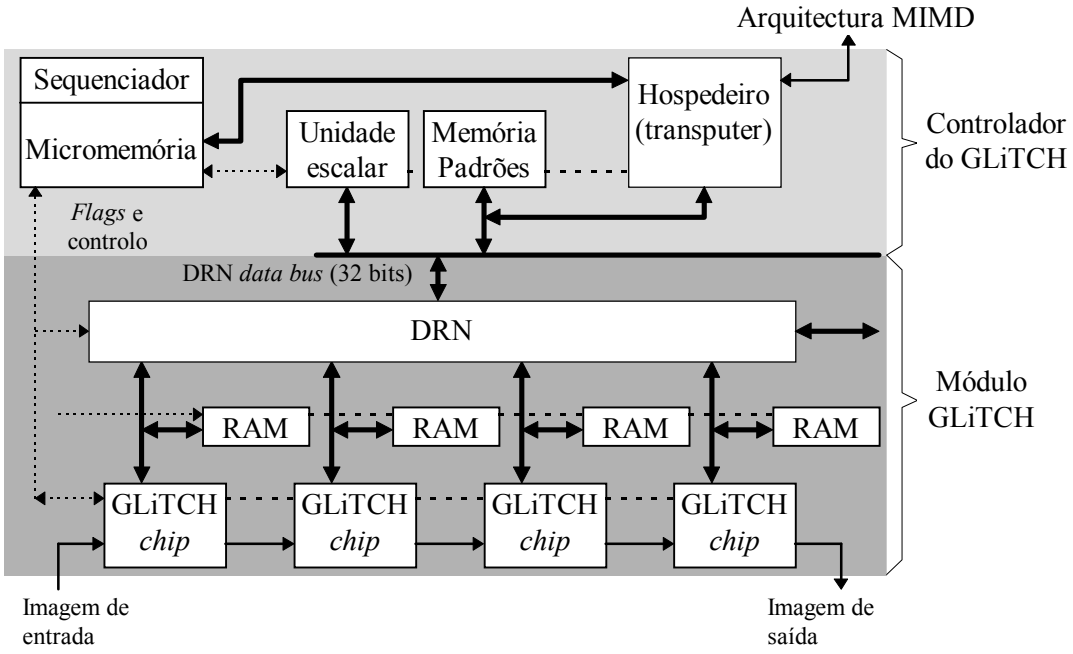


Figura 2.8 Sistema GLiTCH

Por razões de espaço e de modularidade foi decidido construir a unidade de controlo (tal como está assinalada na figura 2.8) numa placa independente. Esta unidade foi desenhada numa placa *Eurocard* de comprimento e largura duplos, com 8 camadas, e comunica com um ou mais módulos GLiTCH através de um *backplane* para o barramento VME (J1 e J2), no qual foram redefinidos todos os sinais lógicos. Esta última opção permite ainda que todo o sistema seja fisicamente montado num *rack*, como aqueles que são habitualmente utilizados em sistemas baseados no barramento VME. O projecto da unidade de controlo é o tema central desta tese e as questões relevantes serão pormenorizadamente descritas mais adiante. No apêndice B estão incluídos os diagramas esquemáticos da unidade de controlo.

2.6.2 O HVA

O sistema HVA (*Heterogeneous Vision Architecture*) procura integrar, numa estrutura lógica em *pipeline*, três subsistemas (adequados aos três paradigmas de processamento associados a visão por computador, e anteriormente identificados), nomeadamente [Duller, et al. 1990], [Pout, et al. 1991] e [Santos, et al. 1995]:

- um módulo de processamento de sinal digital, baseado em circuitos IMS A110;
- um ou mais módulos GLiTCH e uma unidade de controlo para implementação dos algoritmos ao nível do processamento da imagem, e que exigem o tipo de paralelismo *fine-grain*; e
- um módulo com arquitectura MIMD, baseado em transputers, para implementação dos algoritmos de alto nível e que exigem o tipo de paralelismo *coarse-grain*.

A figura 2.9 mostra a arquitectura do HVA através de um diagrama de blocos. Os três blocos de processamento são interligados, de acordo com a hierarquia lógica de funcionamento, através de barramentos de vídeo rápidos (MAXbus ™), aos quais se encontra igualmente ligada uma memória dedicada (*framestore*) para armazenamento temporário das imagens. Cada *framestore* tem capacidade para armazenar vária imagens, dependendo da memória disponível e da dimensão da imagem. A estrutura implementada permite que cada módulo possa operar sobre a mesma imagem mais do que uma vez.

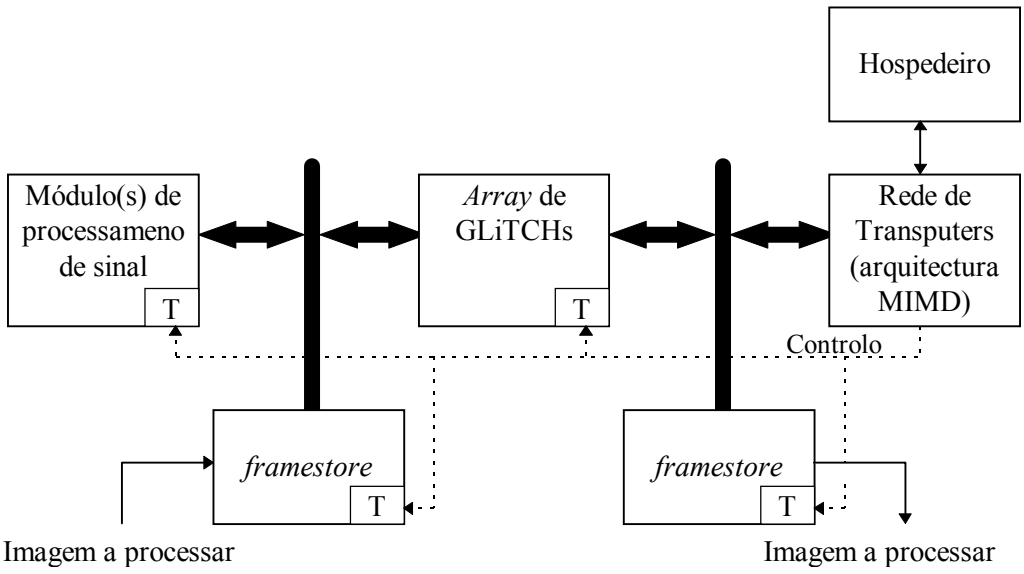


Figura 2.9 Diagrama de blocos do sistema HVA

Todos os módulos de processamento incluem um ou mais transputers que, para além da sua função de processadores escalares, implementam uma rede de controlo assente na versatilidade dos seus canais de comunicação dedicados (*links*). A principal função do hospedeiro é a implementação do interface com o utilizador e a supervisão do funcionamento do sistema, através de um elemento da rede de transputers, dedicado a essa função. O facto do hospedeiro estar interligado apenas com a rede de transputers sugere que, desejavelmente, o utilizador interactua apenas com o sistema ao nível hierárquico mais abstracto. Contudo, não será difícil implementar o acesso indirecto aos restantes módulos de processamento através dos canais de comunicação dos transputers (eventualmente utilizando a rede de controlo já definida na figura 2.9).

Numa sessão de utilização típica, o utilizador, através do hospedeiro e no contexto de uma determinada aplicação, selecciona o conjunto de algoritmos para execução nos diferentes módulos, os quais serão posteriormente enviados para os respectivos processadores. No caso do *array* de GLiTCHs o hospedeiro local da unidade de controlo carregará na micromemória o(s) microprograma(s) recebidos. Na maioria das aplicações não é praticável dispor de um *array* de GLiTCHs de dimensão suficiente para processar uma imagem numa só passagem, pelo que será necessário fragmentar logicamente a imagem numa sequência de quadros, que serão processados sequencialmente. No que respeita ao barramento de vídeo, aos *framestores* e ao módulo de processamento de sinal, a divisão em quadros não levanta qualquer dificuldade. O mesmo não acontece para os restantes módulos, nos quais é frequente utilizar em simultâneo a informação contida em diferentes quadros, sendo por isso necessário adequar os algoritmos à partição efectuada sobre a imagem. No que respeita ao GLiTCH, o impacto dessa partição no desempenho e na metodologia de desenvolvimento dos algoritmos foi estudado em paralelo com a avaliação da própria arquitectura [Duller, et al. 1989a], [Duller, et al. 1989c] e [Duller, et al. 1989d].

Estando preparados todos os módulos o hospedeiro desencadeia a execução do sistema. A primeira imagem é recebida pelo *framestore*, o qual começará imediatamente a enviar para o barramento de vídeo a sequência de quadros, segundo a ordem pré-definida, e ao mesmo tempo que recebe já a imagem seguinte. Cada quadro é processador pelo módulo de processamento de sinal e enviado ao *array* de GLiTCHs. Este armazena o primeiro quadro no VSR e posteriormente transfere-o para a CAM, onde será processado. Em simultâneo um novo quadro é carregado no VSR. Terminado o processamento do quadro, o conteúdo da CAM é trocado com o do VSR e o quadro processado será enviado para o segundo *framestore*, ao mesmo tempo que um terceiro quadro é introduzido no VSR. A imagem processada será reconstruída no segundo *framestore*, onde estará disponível para ser processada, ao nível mais elevado, pela rede de processadores (arquitectura MIMD).

Esta breve descrição do principio de operação do sistema revela um dos aspectos mais complicados do ambiente de desenvolvimento, que é o sincronismo necessário entre as tarefas dos diversos blocos funcionais, sobretudo considerando a possibilidade de ser necessário processar o mesmo quadro mais que uma vez num mesmo módulo, o que se verifica com o *array* de GLiTCHs para alguns dos algoritmos já implementados.

A validação da arquitectura proposta exige substancialmente mais trabalho do que a mera justificação dos módulos individuais para as tarefas que lhe foram atribuídas. Para efectuar esse estudo é essencial dispor de um protótipo. A arquitectura MIMD (rede de transputers) está disponível e, à excepção do módulo GLiTCH, todos os restantes subsistemas estão já concluídos. Dificuldades na produção e teste do circuito integrado GLiTCH são a razão básica do bloqueio actualmente verificado neste projecto, o que por sua vez evoca a necessidade de reanalisar continuamente a adequação dos paradigmas do desenvolvimento de sistemas digitais face à evolução tecnológica. Esta tese procura fazer esse tipo de análise, no contexto da experiência do autor no desenvolvimento da unidade de controlo do *array* de GLiTCHs.

Capítulo 3

3. O Projecto de sistemas digitais

3.1 Introdução

Segundo um estudo elaborado por B. Courtois [Courtois 1993], a actividade industrial na área da electrónica atingiu em 1993 um volume de negócios avaliado em cerca de um décimo do produto bruto mundial. Este indicador evidencia a importância que a electrónica, de um modo geral, assumiu no nosso mundo. O mesmo estudo salienta ainda que, apesar de as ferramentas de CAD representarem apenas uma pequena percentagem daquele montante, a sua importância acentua-se rapidamente, o que resulta do maior grau de automatismo exigido para a utilização dos componentes electrónicos, em constante evolução tecnológica, e ainda da necessidade de construir rapidamente protótipos com o menor número de erros possível.

As ferramentas de CAD já eram usadas na indústria electrónica, no apoio à produção de circuitos impressos, antes da introdução dos circuitos integrados no início dos anos 60. Os primeiros circuitos integrados tinham níveis de complexidade tão baixos que era possível construir manualmente as respectivas "máscaras" associadas ao processo de fabrico. A situação alterou-se no início dos anos 80, com o aparecimento das tecnologias de elevada escala de integração (VLSI), para as quais o apoio automático é imprescindível, mormente ao nível da síntese automática, da verificação, da optimização e da geração das "máscaras". Como justificação e tomando o exemplo de um microprocessador vulgar, basta realçar que o projecto da respectiva "máscara", numa escala em que pudesse ser verificada manualmente, teria o tamanho de um campo de futebol [Newton 1991].

No domínio das tecnologias de implementação, o aparecimento da lógica programável demonstrou-se extremamente importante, sobretudo na prototipagem rápida de sistemas. A sua utilização é economicamente viável porque os componentes programáveis são produzidos com estruturas muita variadas, bastante flexíveis e em elevadas quantidades, o que naturalmente se traduz em custos mais baixos. Por outro lado, a disponibilidade de versões programáveis pelo utilizador permite a construção rápida de sistemas ao evitar o tempo gasto com o processo de fabrico.

De uma forma geral, o projecto de sistemas digitais assenta num espaço multi-dimensional complexo definido por variáveis (muitas vezes com carácter subjectivo e discretas) como a área, o desempenho, o grau de fiabilidade e o grau de testabilidade. Ao projectar sobre este espaço procura-se optimizar o custo, a funcionalidade e o tempo de desenvolvimento, com base num dado conjunto de restrições inerentes à especificação inicial. Vários factores afectam os mecanismos de optimização, tais como o tipo de representação inicial e a tecnologia alvo. A complexidade do espaço obriga muitas vezes a reduzir o número de variáveis, segundo algum critério de prioridades. Tipicamente a área e

o desempenho são as variáveis mais fáceis de determinar, existindo várias técnicas de optimização confinadas a essas variáveis [Micheli 1994].

O aparecimento das tecnologias de implementação (re)programáveis alterou alguns dos paradigmas associados ao desenvolvimento de projectos. A rapidez na obtenção de um protótipo - disciplina habitualmente designada por RSP (*Rapid System Prototyping*) [Dollas e Kanopoulos 1995] - assumiu um papel relevante, implicando uma maior preocupação ao nível da estratégia de desenvolvimento, desde a especificação à síntese. Neste contexto, das tarefas a executar no projecto destacam-se a selecção de ferramentas e de componentes, a modelação, a verificação e a avaliação de alternativas de implementação. Todas estas tarefas constituem uma nova área com crescente necessidade de suporte automático [Katz 1994].

Contudo, o elevado grau de automatismo introduzido no projecto de sistemas digitais pode introduzir alguns aspectos negativos. Um dos mais importantes é a criação de uma dependência elevada das ferramentas de desenvolvimento e o consequente maior afastamento do desenhador relativamente à implementação real, dificultando a intervenção manual que é muitas vezes necessária. Sobretudo em aplicações em que o desempenho é prioritário, a utilização de técnicas manuais e o conhecimento intrínseco do circuito podem conduzir a resultados óptimos, e por vezes essenciais, para satisfazer as restrições inicialmente impostas [Newton e Sangiovanni-Vincentelli 1986].

No contexto das novas metodologias de desenvolvimento, a utilização de linguagens de alto nível, globalmente designadas por HDLs (*Hardware Description Languages*), surge como uma alternativa à utilização dos tradicionais esquemáticos. As vantagens habitualmente realçadas para as HDLs são a melhor capacidade de expressão algorítmica, a melhor adaptação ao processamento automático e a introdução no desenvolvimento de *hardware* de uma perspectiva muito semelhante à do desenvolvimento de *software*, o qual, por se encontrar numa fase mais avançada, pode constituir uma considerável fonte de experiências [Marschner 1992]. Este paralelismo entre as duas actividades tinha sido já notavelmente identificado por Tanenbaum, em 1976, através da famosa afirmação [Tanenbaum 1976]:

"... hardware and software are logically equivalent."

3.2 Estrutura do processo de desenvolvimento

3.2.1 Níveis de abstracção

Um sistema digital pode ser especificado a vários níveis e utilizando diferentes classes de elementos, que no conjunto definem aquilo que é designado por domínios de representação. Os tradicionais esquemáticos correspondem a um desses domínios, assim como os diagramas de estado, os diagramas de fluxos, as HDLs, ou ainda as redes de Petri, correspondem a um outro domínio de representação. O nível de abstracção está relacionado com o nível de detalhe que se adopta e este depende essencialmente dos objectos que definem a implementação no nível em questão, ou seja, do processo de síntese. Como exemplo, se um desenhador pretender construir um micro-computador convencional utilizará como elementos de projecto um microprocessador, um bloco de

memória e algum bloco de interface com periféricos. Mas se pretender construir um controlador específico para uma determinada aplicação é natural que, a partir de um dos processos de especificação atrás referidos, divida o projecto em blocos de dimensão tratável, implementando cada um deles com registos, multiplexadores, descodificadores e outros componentes ao mesmo nível. Em particular neste último caso, a metodologia de especificação e síntese dependem, obviamente, das ferramentas de CAD disponíveis.

Actualmente, é possível identificar três domínios de representação (Comportamental, Estrutural e Físico) e cinco níveis de abstracção (Dispositivo, Lógico, Registo, Arquitectura e Sistema). A figura 3.1 mostra uma forma de visualizar essa estrutura, num arranjo tripartido designado por *Y-chart*, que é útil para caracterizar a operação de um qualquer conjunto de ferramentas, como será demonstrado posteriormente [Gajski e Kuhn 1983].

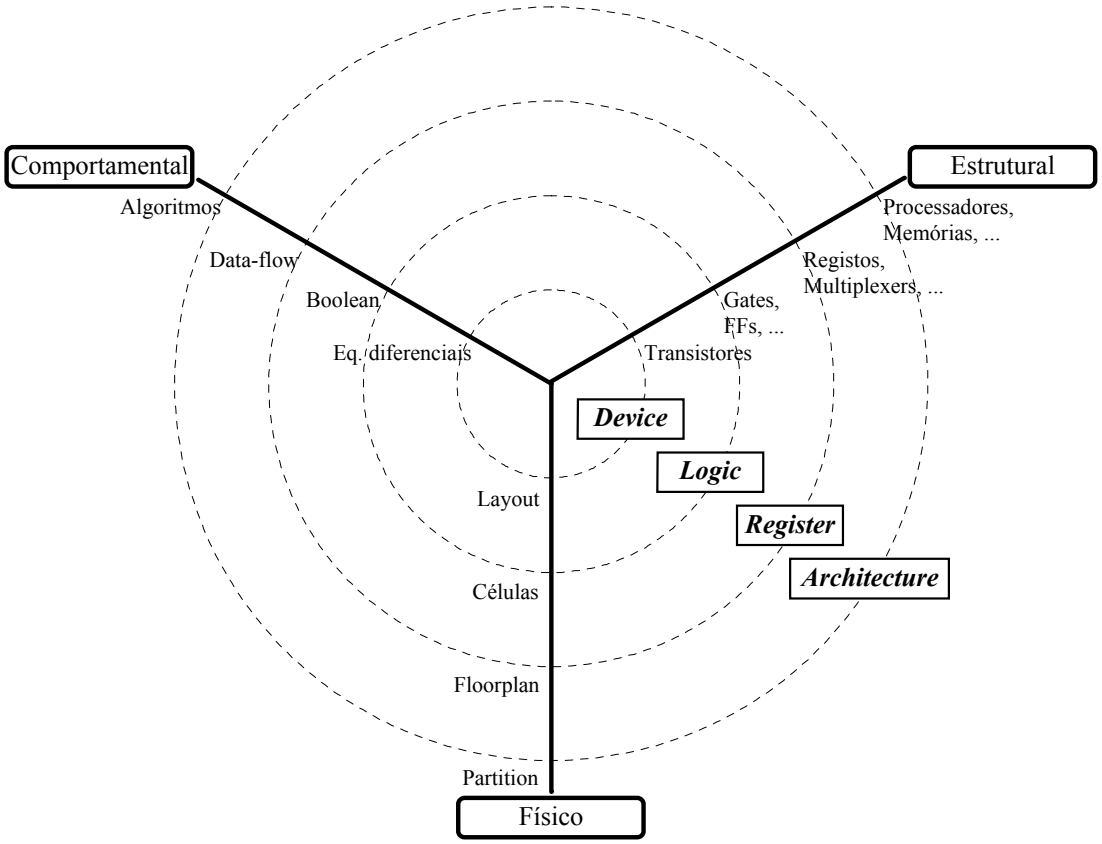


Figura 3.1 Modelo tripartido para domínios e níveis de representação (*Y-chart*)

O nível do sistema não aparece na figura atendendo a que este nível ainda não é vulgarmente utilizado como parte integrante do processo automático de desenvolvimento. No entanto, começam já a aparecer alguns trabalhos publicados que procuram caracterizar o problema e apresentar metodologias e representações, de que é exemplo a notação SpecCharts [Narayan, et al. 1992]. No domínio comportamental, o nível do sistema deverá corresponder a uma especificação, na forma de linguagem, das características do sistema em desenvolvimento, nomeadamente a função e o desempenho. No domínio estrutural, os

elementos de representação serão unidades funcionais independentes, como computadores, discos e monitores.

Apesar de o domínio físico estar descrito essencialmente em função do projecto de circuitos integrados, os problemas e soluções, particularmente ao nível do registo, apresentam muitas semelhanças com o desenho de circuitos impressos. Muitos dos algoritmos de *placement*, *routing* e simulação, como exemplos, foram desenvolvidos em poderosas ferramentas para desenho de circuito impresso, antes do aparecimento dos circuitos integrados. A fusão destas duas tecnologias verificou-se no fim dos anos 70 e início dos anos 80. Nesta perspectiva, é possível considerar o desenho de circuitos integrados como um processo microscópico, variante do desenho de circuitos impressos.

A estruturação atrás apresentada não é universalmente aceite. Outros autores apresentaram esquemas de estruturação distintos, nomeadamente Armstrong, que defende a existência de apenas dois domínios, o comportamental e o estrutural, considerando o projecto físico como um nível adicional, abaixo do nível do dispositivo. Esse nível adicional não tem representação no domínio comportamental, mas apenas no domínio estrutural, sendo os objectos definidos por formas geométricas. Esta perspectiva corresponde a considerar o projecto físico como um processo automático e fechado, não sendo relevante, ou até desejável, a intervenção humana nas diversas fases de execução [Armstrong e Gray 1993].

3.2.2 Operações sobre o projecto

Algumas operações do processo de desenvolvimento de sistemas digitais implicam uma mudança de nível e/ou domínio, enquanto que outras apenas procuram optimizar ou verificar um projecto num particular ponto daquele espaço. No primeiro caso, as operações são realizadas por ferramentas que habitualmente são designadas por ferramentas de síntese. Para o segundo caso podem-se indicar como exemplos a simulação, a simplificação lógica e a verificação de regras de interligação e segurança, estas últimas aplicáveis tipicamente no domínio estrutural (designações habituais: ERC - *Electrical Rule Checking* - e DRC - *Design Rule Checking*).

No processo de desenvolvimento convencional o projectista executa o projecto no domínio estrutural, possivelmente ao nível da arquitectura e/ou do registo, através de um diagrama esquemático. O nível de abstracção pode ser elevado recorrendo à definição hierárquica de componentes estruturais. Esse projecto pode ser simulado, muito embora esta tarefa esteja dependente da disponibilidade de modelos simuláveis dos componentes utilizados e da possibilidade de construir modelos para os componentes criados especificamente para esse projecto (ASICs[1]). Nem todos os editores esquemáticos facilitam este tipo de tarefa. Do diagrama esquemático é extraída, por síntese, uma lista de ligações - *netlist* - que transporta o projecto para um nível de detalhe a partir do qual é possível construir a placa de circuito impresso.

Os ASICs podem ser desenhados a partir de um esquemático, normalmente ao nível do registo, ou a partir de uma descrição comportamental, ao nível da arquitectura ou do registo. Neste segundo caso a descrição obtida é sintetizada, normalmente para uma

[1] *Application Specific Integrated Circuits*

descrição no domínio estrutural, ao nível do registo ou da lógica (um processo habitualmente designado por HLS - *High-Level Synthesis*). Independentemente da especificação inicial, no domínio estrutural são normalmente executadas algumas operações de optimização, e finalmente é feita a síntese para o domínio físico, onde o circuito integrado é realizado. Neste segundo processo, a simulação assume uma importância consideravelmente maior, por ser necessário eliminar eventuais erros de projecto ao nível da especificação e não ao nível do produto final. A detecção desses erros no circuito integrado, especialmente em circuitos complexos, é uma tarefa de difícil execução, por ser igualmente difícil aceder aos nós internos do circuito.

A simulação é limitada naturalmente pelo facto de demonstrar que a unidade em teste funciona correctamente apenas para os vectores de teste utilizados. Principalmente em sistemas de complexidade elevada, é importante ter uma avaliação correcta da efectividade do processo de simulação, o que por si só é uma tarefa difícil e normalmente implica uma divisão do projecto inicial em blocos de menor dimensão e de mais fácil manuseamento. Este problema pode ser minorado utilizando técnicas de especificação formal [Stavridou, et al. 1987] e [Borrione, et al. 1992], ou algoritmos para geração automática de vectores de teste [Wilkins 1986]. A especificação formal permite sintetizar circuitos correctos por construção (no que respeita à especificação) e é uma disciplina promissora. Actualmente está limitada pelo nível de complexidade suportada e a sua integração nas ferramentas de desenvolvimento comerciais.

3.2.3 Testabilidade

Um último aspecto a considerar no processo de desenvolvimento é o teste e diagnóstico da implementação (realização física). Nesta fase procura-se detectar erros associados apenas ao processo de fabrico, quer seja de componentes (ASICs) ou de circuitos impressos. Esta tarefa exige a utilização de equipamento de teste dedicado (e dispendioso), e passa pela concepção de modelos de falhas adequados e de algoritmos para geração de vectores de teste.

Das técnicas de teste disponíveis para sistemas digitais, aquelas que têm suscitado maior interesse na comunidade de projectistas e de investigadores, tanto ao nível do circuito integrado como ao nível das placas de circuito impresso, são as técnicas que utilizam o conceito de *Boundary Scan* (BS), baseadas em *Scan Registers*. De uma forma simples, este tipo de registo tem capacidade para operar no modo paralelo, em funcionamento normal, e no modo série, como um registo de deslocamento, quando se efectuam operações de teste. Utilizando estruturas lógicas deste tipo no circuito a testar, um sistema de teste dedicado pode isolar cada módulo, introduzir-lhe vectores de teste forçando-o a entrar num determinado estado, e verificar em seguida a sua resposta em determinadas condições de operação. Outro contributo importante para a aceitação desta filosofia de teste foi a definição da norma IEEE 1149.1-1990 "*Test Access Port and Boundary Scan Architecture*" (a partir da proposta conhecida por "JTEG *Boundary Scan*"), que define uma arquitectura básica para a estrutura lógica associada ao circuito de teste [Abramovici, et al. 1990] e [McClean 1992].

Esta filosofia de teste é particularmente eficiente em circuitos VLSI, permitindo o acesso a nós internos do circuito sem comprometer a área total do circuito e número de

pinos necessários para a função de teste. Contrariamente, em circuitos de baixa complexidade a área do circuito dedicada a funções de teste revelar-se-á, facilmente, demasiado penalizadora, inviabilizando assim a sua aplicação. O mesmo tipo de argumento pode ser usado ao nível do sistema, isto é, uma placa de circuito impresso que inclua vários componentes complexos poderá certamente suportar a inclusão adicional da estrutura de teste (tipicamente barramentos e processador dedicado), o mesmo não se verificando em placas relativamente simples. Mas mesmo em sistemas complexos, uma dificuldade adicional pode surgir quando são utilizadas diferentes tecnologias de implementação e nem todos os componentes incluem o suporte para BS. Nestes casos, a inclusão ou não deste tipo de teste é uma questão que deve ser devidamente avaliada.

Do ponto de vista do projecto, a utilização das técnicas de teste implica ainda a integração de metodologias e estruturas lógicas específicas nas ferramentas de CAD. Se tal não se verificar, a complexidade adicional associada à descrição das estruturas de teste podem atrasar o desenvolvimento e influenciar negativamente o projectista, distraindo-o das questões relacionadas com o projecto propriamente dito.

3.2.4 Conclusão

Como conclusão desta breve discussão sobre o processo de desenvolvimento de sistemas digitais, deve-se realçar a importância que assume a escolha de uma metodologia de projecto face a questões como o tempo de desenvolvimento, optimização e testabilidade da solução final. Igualmente importante é avaliar correctamente o impacto que a escolha de uma particular tecnologia de implementação tem sobre a metodologia de desenvolvimento, em consequência das ferramentas de CAD disponíveis para essa tecnologia e os eventuais interfaces com outras ferramentas.

3.3 Ferramentas de CAD

As ferramentas de CAD podem ser classificadas atendendo à sua função no processo de desenvolvimento e os níveis de abstracção em que se aplicam. A taxinomia daí resultante é útil na selecção de uma ferramenta mais adaptada a determinada operação, e na caracterização de ferramentas, na perspectiva de uma análise comparativa. A figura 3.2 mostra a composição da taxinomia proposta por Armstrong [Armstrong and Gray 1993].

A optimização e síntese partilham a propriedade de alterar o projecto para uma nova forma, eventualmente para um domínio e/ou nível diferentes. Muito embora estas ferramentas não sejam ainda aplicadas ao nível do sistema, é previsível que num futuro próximo tal se venha a verificar, na consequência da evolução tecnológica.

As ferramentas de análise e verificação são aplicadas em todos os níveis e domínios. No domínio físico, podem ser utilizadas para garantir que um dado *layout* corresponde a um circuito que pode ser fabricado, atendendo às restrições do processo de fabrico em questão. No domínio estrutural, podem ser utilizadas para garantir que as condições de carga de cada saída não excedem um dado limite imposto pela tecnologia associada, ou ainda para determinar o atraso temporal associado a um sinal ou conjunto de sinais. No domínio comportamental, e em particular quando é utilizada uma linguagem descritiva, as

ferramentas de análise são utilizadas para detectar erros que violem a léxica, a sintaxe e a semântica da linguagem.

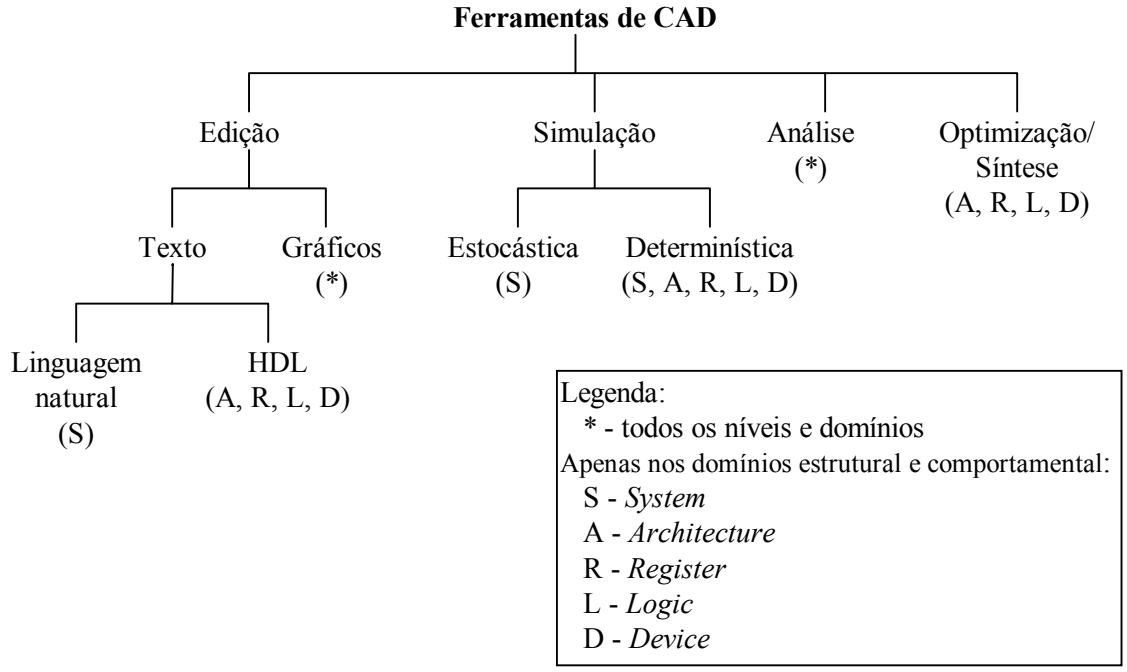


Figura 3.2 Taxinomia das ferramentas de CAD

Os simuladores estocásticos são utilizados ao nível do sistema para extrair parâmetros associados ao desempenho de um sistema, como seja a razão de transferência. Estes simuladores utilizam modelos habitualmente designados por "modelos não interpretados", cujo estado não depende dos dados, mas apenas das acções de controlo externas [Aylor, et al. 1992]. Os simuladores determinísticos operam sobre uma representação mais próxima do objecto simulado, utilizando tipos de dados que representam, por exemplo, tensões, valores binários, ou símbolos mais ou menos abstractos, dependendo do nível de abstracção adoptado. Este tipo de simulador procura sempre determinar a resposta do objecto simulado a um conjunto de estímulos (vectores de teste) e, eventualmente, compará-la com a resposta esperada, de acordo com a especificação inicial.

Os editores gráficos ou esquemáticos são utilizados em todos os níveis e domínios. No domínio estrutural permitem criar os convencionais diagramas esquemáticos, no domínio físico permitem combinar formas geométricas que definem a operação do circuito físico (independentemente de se tratar de um componente ou de um circuito impresso) e no domínio comportamental permitem criar diagramas de fluxos que descrevem o comportamento. Os diagramas de fluxo podem ser de diversos tipos, como por exemplo diagramas de fluxo de dados no editor PICSIL [Pearson 1992], diagramas de estado no editor StateCharts [Harel 1988], ou ainda redes de Petri [Ecker e März 1993].

Os editores de texto são utilizados para construir descrições ao nível do sistema, tipicamente utilizando uma linguagem natural, ou a qualquer um dos níveis inferiores, utilizando uma linguagem específica para a descrição de *hardware* - HDL (*Hardware Description Language*). Muito embora algumas destas linguagens possam ser utilizadas no domínio estrutural, e mesmo físico, elas encontram a sua aplicação mais adequada no domínio comportamental.

3.4 Tecnologias de implementação

A viabilidade dos sistemas baseados em microelectrónica está fortemente condicionada por alguns factores interdependentes, associados ao tipo de componentes utilizados, tais como o volume de produção, o custo de desenvolvimento e o desempenho exigido para um dado nível de competitividade. Como exemplo, um microprocessador deverá ter um desempenho elevado, o que habitualmente se traduz num processo de desenvolvimento dispendioso, mas que será amortizado por um volume de produção elevado, conduzindo a um nível de competitividade desejado. Se o volume de produção não atingir o mínimo exigido o projecto não será viável. Numa situação diversa, um circuito para uma nave espacial pode necessitar igualmente de um desempenho elevado e um grau de confiança superior. Estas exigências implicam também um custo de desenvolvimento elevado, que neste caso não será amortizado, pois o volume de produção será baixo. No entanto, esse custo é desprezável face ao custo da nave, o que justifica a utilização daqueles critérios.

As tecnologias de implementação disponíveis podem ser classificadas segundo uma taxinomia como a que se encontra esquematizada na figura 3.3 [York 1994]. Numa taxinomia mais detalhada é ainda possível identificar várias subclasses das classes *Gate-Array* e *Cell-Based* (*Semi-Custom*), mas essa divisão não é relevante ao nível considerado nesta abordagem.

As metodologias de projecto adoptadas no tipo de projecto *custom*, e no tipo de projecto baseado em componentes *standard*, diferem essencialmente porque, no primeiro caso, é exigido o esforço adicional no desenvolvimento de um ou mais circuitos integrados dedicados a uma determinada operação (ASICs). Esse esforço adicional, e o custo dele decorrente, devem ser objectivamente validados pela necessidade de um nível de desempenho incompatível com os componentes *standard* disponíveis, ou então pela possibilidade de integrar mais funções num espaço mais reduzido, procurando optimizar dessa forma um dos factores de custo do produto final.

Ao optar pelo desenvolvimento de circuitos integrados dedicados, o projectista tem duas opções que diferem basicamente no processo de realização física, uma vez que o projecto lógico pode ser absolutamente idêntico. Na tecnologia *Mask Programmable*, o componente é realizado na fábrica utilizando uma descrição física fornecida pelo desenhador, enquanto que na tecnologia *Field Programmable* o componente é realizado no laboratório, programando uma estrutura lógica genérica fornecida pelo fabricante. Adicionalmente, alguns dos componentes desta última classe são (interessantemente) reprogramáveis, dentro e/ou fora do circuito final, o que introduz uma elevada flexibilidade. De uma forma genérica, os componentes *Mask Programmable* implicam um custo mais elevado e um ciclo de desenvolvimento maior, permitindo um maior nível de

desempenho e uma maior escala de integração (optimização de espaço). Com base nesta análise simples, é já possível concluir as vantagens da utilização de componentes da tecnologia *Field Programmable* na fase de prototipagem de um projecto, e da utilização de componentes *Mask Programmable* na produção maciça.

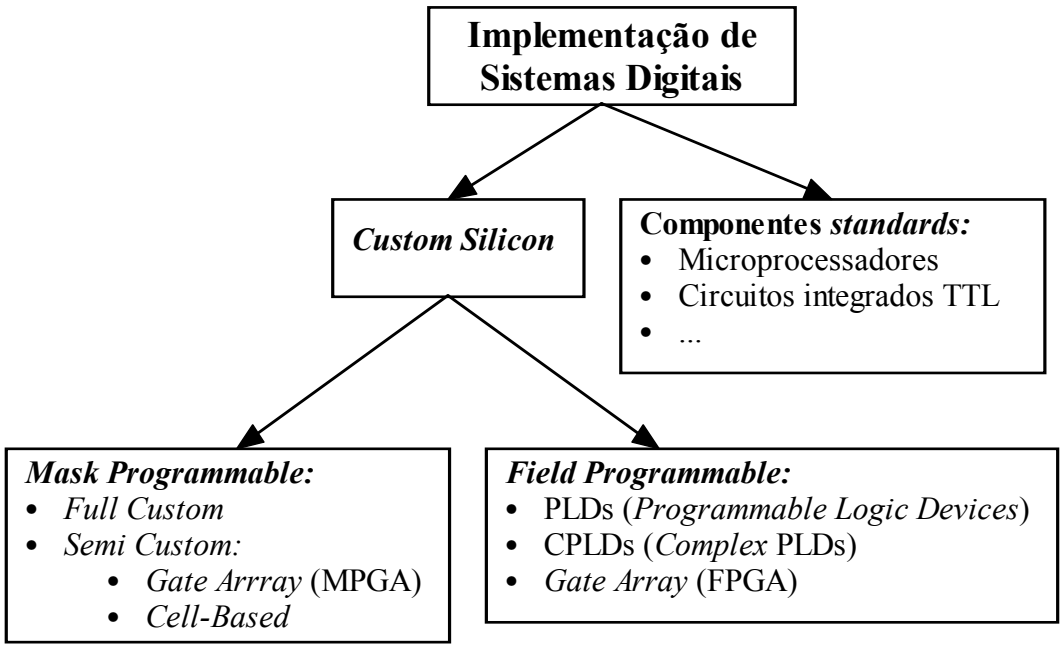


Figura 3.3 Tecnologias de implementação de sistemas digitais

Em ambas as tecnologias é possível balancear aqueles parâmetros através da escolha de um determinado método de projecto ou estrutura lógica. Na tecnologia *Mask Programmable*, o método de projecto *Full-Custom* caracteriza-se por estar fortemente apoiado num esforço de optimização manual, na escolha de melhores soluções para cada detalhe do circuito. O desempenho e a área são óptimos, correspondendo-lhe um custo e um ciclo de desenvolvimento muito elevados. O método de projecto *Semi-Custom* restringe as primitivas de projecto a um número limitado de módulos pré-elaborados, impedindo assim a optimização manual do detalhe, mas acelerando o processo de desenvolvimento, uma vez que o desenhador apenas necessita de se concentrar nas decisões relativas à escolha das primitivas e das suas interligações, num nível de abstracção mais elevado. A este nível, o leque de opções de implementação é mais restrito, o que facilita o desenvolvimento de ferramentas de CAD, que por sua vez permitem encurtar o ciclo de projecto.

Na tecnologia *Field Programmable*, o balanceamento é feito através da escolha de uma determinada estrutura lógica. Nos componentes designados por FPGA, a célula básica permite a implementação de quaisquer funções lógicas de um número reduzido de variáveis, enquanto que nos componentes designados por PLD e CPLD, a célula básica permite a utilização de um número de variáveis consideravelmente maior, mas limita o número de produtos lógicos disponíveis. Nos FPGA as funções lógicas mais complexas são implementadas interligando células, o que se vai traduzir no acumular de atrasos e consequente degradação do desempenho. Nos (C)PLD as funções lógicas mais simples vão

utilizar parcialmente uma célula complexa, o que se vai traduzir por um desperdício de espaço. As técnicas de optimização e o ciclo de projecto também diferem entre estes tipos de componentes, condicionando de alguma forma a sua utilização. Esta tecnologia será abordada com mais detalhe posteriormente, dado o seu especial interesse na prototipagem rápida de um sistema complexo como o controlador do GLiTCH.

A maioria dos sistemas reais a desenvolver exige soluções híbridas, incluindo num mesmo projecto diversas tecnologias de implementação. Este facto acentua-se com o aumento progressivo da complexidade dos sistemas realizáveis, e que é resultante do próprio desenvolvimento tecnológico. Particularmente interessante é a utilização conjunta de componentes *standard* reprogramáveis (microprocessadores) com um elevado grau de flexibilidade e de ASICs com um elevado grau de desempenho. As questões de integração são complexas e levaram ao aparecimento de uma disciplina de investigação, habitualmente designada por *codesign* [Subrahmanayam 1993] e [Buchenrieder e Rozenblit 1995].

3.4.1 Critérios de selecção

Na discussão anterior, foram já introduzidos muitos dos factores que influenciam a escolha de uma determinada tecnologia de implementação. De uma forma mais sistematizada, nessa escolha é útil considerar um conjunto de factores, tais como [Morant 1990]:

1. Custo;
2. Tamanho final do circuito;
3. Velocidade de operação;
4. Potência exigida;
5. Tempo de desenvolvimento;
6. Segurança;
7. Grau de confiança;
8. Risco; e
9. Flexibilidade.

O custo inclui os componentes, a sua montagem, o teste e o custo de desenvolvimento (e eventualmente de comercialização) repartido pelo número de unidades que se espera produzir. No caso de ser necessário adquirir ferramentas de CAD para o desenvolvimento, o seu custo deve igualmente ser considerado de alguma forma. De realçar ainda que os critérios adoptados para a generalidade dos restantes factores têm uma influência directa ou indirecta no custo, para além de condicionarem a escolha de componentes, a metodologia de teste e o processo de desenvolvimento. Este conjunto de características faz com que a função custo seja particularmente sensível, tal como foi anteriormente referido.

O tamanho do circuito, a velocidade de operação e a potência exigida são parâmetros mutuamente dependentes, sendo possível (e desejável) a sua análise conjunta através de ferramentas automáticas. Tipicamente, ao procurar optimizar a velocidade de funcionamento, as opções a tomar implicam componentes com maior consumo de energia e com maiores dimensões. Inversamente, quando o tamanho do circuito é a variável a optimizar, as opções irão condicionar de alguma forma a velocidade de resposta. No

espaço definido por estas três variáveis existem tipicamente algumas restrições físicas impostas pela especificação inicial.

O tempo de desenvolvimento é outro aspecto essencial para a viabilidade de um produto baseado em microelectrónica. Para além dos aspectos comerciais relacionados com a concorrência entre empresas, se o tempo de desenvolvimento for excessivamente longo, a evolução tecnológica pode tornar um sistema antiquado ainda antes da sua introdução no mercado. Neste contexto, a prototipagem rápida para validação da especificação e avaliação de soluções é uma disciplina particularmente importante.

A segurança refere-se à protecção da propriedade de um produto. Neste aspecto, apenas os sistemas baseados em componentes *standards* evidenciam problemas, atendendo a que, ao serem *standards*, os componentes são universalmente conhecidos, o que facilita a duplicação não autorizada desses sistemas. Todos os componentes específicos escondem facilmente a sua (micro) estrutura e função. O acesso a essa informação fica limitado às actividades de espionagem industrial e às tentativas (paranóicas) de relacionar as respostas a estímulos fornecidos.

O risco na realização e o grau de confiança são parâmetros que incidem sobre os aspectos físicos da realização do circuito. Como exemplos do tipo de problema associado a estes parâmetros, um circuito que recorra a um número elevado de pontos de soldagem não terá um grau de confiança muito elevado, já que tecnicamente as soldagens introduzem vários problemas; nos componentes *Full Custom* não existem soldagens, mas a possibilidade de introduzir erros humanos num processo altamente complexo, dispendioso e irreversível, transforma o seu desenvolvimento numa actividade de elevado risco.

A flexibilidade está associada à ideia de alteração da funcionalidade. Nesta perspectiva, consideram-se flexíveis os projecto baseados em componentes *standards* reprogramáveis, cuja função pode ser alterada pela simples alteração de um programa de *software*, e os componentes do tipo *Field Programmable*, cuja função pode ser alterada redefinindo as funções lógicas das células constituintes e as respectivas interligações. Nestes componentes, o grau de flexibilidade depende da forma como a reprogramação é implementada. A menor flexibilidade verifica-se para os componentes em que é necessário o pré-apagamento por radiação. Na grau seguinte enquadram-se os componentes que apenas exigem a utilização de um programador dedicado, obrigando por isso à sua remoção do circuito. A máxima flexibilidade é conferida por os componentes que permitem a sua reprogramação no circuito, numa atitude logicamente semelhante à que é usada com soluções de *software*.

A tabela 3.1 procura sumariar as vantagens e desvantagens de cada uma das classes das tecnologias de implementação, segundo alguns dos elementos mais relevantes do conjunto de parâmetros referidos [DATA 1986]. Ao nível da prototipagem é desejável obter uma solução com as seguintes características:

- tempo de desenvolvimento reduzido, de modo a poder validar a especificação tão cedo quanto o possível;
- grau de fiabilidade elevado, o que proporciona uma elevada confiança na análise de resultados, facilitando assim a detecção de erros; e

- grau de flexibilidade elevado, por forma a poder realizar facilmente as eventuais modificações.

	Componentes standard	Mask Programmable	Field Programmable
Custo[1]	Baixo	Elevado	Médio
Velocidade	Média	Elevada	Baixa/Média
Tamanho	Elevado	Reduzido	Médio
Tempo de desenvolvimento	**Reduzido**	**Longo**	**Reduzido/Médio**
Grau de confiança	**Baixo**	**Elevado**	**Médio**
Risco	Médio	Elevado	Baixo
Flexibilidade	**Média**	**Baixa**	**Elevada[2]**

1. O custo aqui referido constitui uma estimativa baseada apenas nos custos de desenvolvimento e de componente, ao nível do protótipo.
2. Sobretudo considerando as versões reprogramáveis no próprio circuito.

Tabela 3.1 Comparação entre as tecnologias de implementação

Atendendo aos objectivos enunciados para a prototipagem de sistemas digitais, ao consultar a tabela 3.1 verifica-se que os componentes *Field Programmable* estão bem adequados àquela fase de projecto - garantem o mais elevado grau de flexibilidade e evidenciam um compromisso relativamente ao tempo de desenvolvimento e o grau de confiança. Este facto levou a que esta classe de componentes fosse intensivamente utilizada no protótipo da unidade de controlo do GLiTCH. Actualmente existem diversas estruturas lógicas programáveis. A selecção de uma dessas estruturas para um determinado circuito não é um problema de solução trivial, e exige uma análise mais detalhada dos componentes *Field Programmable*.

3.4.2 Componentes programáveis pelo utilizador

O conceito de lógica programável pelo utilizador (daqui em diante designada simplesmente por lógica programável) foi introduzido em meados dos anos 60 pela Harris Semiconductor, através de uma estrutura matricial de díodos configurável por micro fusíveis. Este tipo de estrutura viria a ser especialmente útil no desenvolvimento de memórias PROM, mas, no que respeita à utilização genérica em sistemas digitais, a sua capacidade limitada restringiu a sua aplicação a circuitos simples. Para ultrapassar essa limitação foi então proposta a PLA, que consistia igualmente numa estrutura matricial, mas orientada para a realização de sistemas digitais. A PLA viria a demonstrar-se uma alternativa de implementação bastante versátil, sendo inclusivamente utilizada em diversas secções de circuitos integrados complexos, como os microprocessadores. Os primeiros componentes do tipo PLA eram *mask-programmable*, e apenas em 1975 foram anunciadas as primeiras versões *field-programmable* (FPLA), pela Intersil e a Signetics. Muitos autores consideram estes componentes como os primeiros PLD.

Uma das maiores dificuldades sentidas inicialmente na utilização da lógica programável foi a falta de programadores adequados e a pouca confiança que a tecnologia de programação inspirava. Nos anos subsequentes, várias alternativas foram exploradas, inclusivamente a utilização de células apagáveis por radiações ultra-violeta, tecnologia que entretanto foi vulgarizada nas EPROMs.

Foi apenas nos fins dos anos 70 que a lógica programável se afirmou como alternativa viável para a implementação de sistemas digitais, com a introdução dos componentes designados por PAL (*Programmable Array Logic*), pela Monolithic Memories Inc. (que se viria a fundir com a Advanced Micro Devices - AMD). Um elevado número de empresas investigou e explorou esse conceito em novos produtos comerciais, alterando a tecnologia de implementação subjacente e a estrutura lógica. Actualmente, os componentes de lógica programável incluem os sofisticados *Field Programmable Gate Array* (FPGA), e está banalizada a reprogramabilidade, que confere a esta tecnologia uma versatilidade notável. Alguns exemplos bem conhecidos de famílias de componentes deste tipo são os MAX da Altera, ACT da Actel, MACH da AMD e FPGA da Xilinx. Alguns detalhes sobre as tecnologias de implementação, e as metodologias de desenvolvimento associadas a cada tipo de estrutura lógica programável, são apresentados em [Alford 1989] e [Pellerin e Holley 1991].

Com base na estrutura interna (e de acordo com a figura 3.3), podem-se dividir os componentes de lógica programável em três classes:

- PLD (*Programmable Logic Devices*)
- CPLD (*Complex* PLD)
- FPGA (*Fiel Programmable Gate Arrays*)

Os PLD correspondem a uma estrutura lógica a dois níveis, especialmente orientada para a implementação directa de funções lógicas com um elevado número de variáveis e expressas na forma de soma de produtos ou tabelas de verdade. A estrutura a dois níveis permite obter tempos de atraso consideravelmente reduzidos e de fácil avaliação (mesmo quando existe realimentação). Nesta classe incluem-se as estruturas ROM (bem como as suas versões reprogramáveis), PAL e PLA, que permitem configurar, respectivamente, as somas a partir de todos os possíveis produtos, um número reduzido de produtos pré-atribuidos a cada soma e um número reduzido de produtos que podem ser distribuídos por qualquer soma. Os PLD podem ainda dispor de Flip-Flops para implementação de lógica sequencial. Na estrutura da PAL, é habitual designar por macrocélula o conjunto dos produtos, a respectiva soma e a célula com o Flip-Flop.

Os CPLD constituem uma extensão da estrutura da PAL, ao incluir várias dessas estruturas num mesmo circuito, interligadas por um bloco interno de comutação de sinais que permite múltiplas relações. Para além dos atributos herdados da estrutura PAL, os CPLD implicam um esforço suplementar na repartição das funções lógicas pelas diversas PALs e na configuração das interligações necessárias. De notar que este conceito apenas transporta para dentro do circuito integrado uma tarefa que tipicamente seria realizada ao nível do circuito impresso, conferindo um maior grau de confiança à solução final.

Os FPGA são componentes cuja estrutura é composta por um elevado número de células simples, com uma elevada densidade de interligações programáveis. As características das células variam entre as diversas famílias, nomeadamente quanto ao tipo de lógica que utilizam (NANDs, MUXs, ou RAMs - *look-up tables* -, com ou sem Flip-Flops), mas tipicamente dispõem de um número reduzido de entradas. Estão assim mais adaptadas ao tipo de optimização multi-nível. Comparando com a estrutura lógica dos CPLD, a menor dimensão da célula permite uma melhor utilização efectiva, mas impõe tempos de atraso maiores e de mais difícil previsão (sobretudo com funções de um elevado

número de variáveis) e ainda um nível adicional de complexidade na repartição das funções pelas diversas células e respectiva interligação. Este último aspecto traduz-se num tempo de desenvolvimento maior, nomeadamente quando existem problemas de repartição e/ou de tempo de resposta, sendo necessária intervenção directa do utilizador para conduzir o processo.

É possível conceber estruturas híbridas que procurem explorar os benefícios identificados para os PLD e para os FPGA. No entanto, não é imediato concluir as vantagens da utilização conjunta daquelas estruturas, dado que elas estão adequadas a diferentes tipos de problemas, cuja ocorrência conjunta, numa mesma especificação, não é frequente. Como exemplo de uma tentativa nessa direcção, pode-se referir a família de PLD FLEX 8000, introduzida em 1993 pela Altera. Estes componentes implementam uma arquitectura típica de um FPGA, mas agrupando células em blocos, dentro dos quais foi enriquecida a interconectividade das células. Apesar da estrutura resultante não ser a de um PLD típico, ela permite optimizar os atrasos acumulados para funções que se confinem a um mesmo grupo [Altera 1993].

A tecnologia utilizada nos componentes de lógica programável é uma característica importante, mas que não tem grande relevância no contexto da análise anterior, já que, para cada uma das tecnologias, existe uma ou mais variantes da maioria das estruturas lógicas anteriormente descritas. As tecnologias de programação podem ser divididas em três classes [Bostock 1993]:

- fusíveis e anti-fusíveis[2] integrados;
- células do tipo EPROM;
- células de RAM estática (apenas CPLD e FPGA).

Tal como foram introduzidos, os componentes do tipo PLD destinam-se a implementar funções lógicas simples substituindo, tipicamente, alguns circuitos integrados de baixa complexidade (SSI e MSI) sempre necessários na realização de sistemas digitais. A sua maior vantagem é a flexibilidade que introduzem na possível redefinição da sua função lógica, o que é sempre importante em prototipagem. Do ponto de vista económico, e considerando os diversos factores implicados (incluindo custo dos componentes, espaço na placa de circuito impresso e esforço de desenvolvimento), alguns estudos revelam que é viável a utilização de lógica programável desde que a substituição seja na ordem de 1 PLD de baixa complexidade para 2 SSI, e 1 PLD de complexidade média para 4 SSI [Bostock 1993]. Na prática estes valores são extremamente fáceis de atingir.

Os componentes CPLD e FPGA destinam-se a implementar funções complexas, podendo substituir um elevado número de circuitos integrados de baixa escala de integração, ou implementar funções semelhantes às de circuitos integrados de elevada escala de integração, mas com características específicas para uma dada aplicação. Também neste caso uma das principais vantagens dos CPLD e FPGA é a flexibilidade, mas que, atendendo à possível elevada complexidade do projecto, assume um papel ainda mais importante.

---

[2] Do inglês *antifuses* - enquanto os fusíveis implementam ligações que podem ser eliminadas por programação, os anti-fusíveis implementam ausências de ligações que podem ser estabelecidas por programação.

Da descrição anterior é possível concluir que, nos componentes FPGA, a principal restrição está relacionada com o tempo de resposta, enquanto que, nos componentes CPLD, o principal problema reside no potencial desaproveitamento de recursos lógicos, ao implementar funções lógicas simples numa macrocélula complexa. A selecção de um destes dois tipos de componentes para uma solução particular não é uma tarefa simples, principalmente a partir de uma especificação de alto-nível onde as características estruturais das funções lógicas estão escondidas na riqueza semântica da especificação. Existem já alguns trabalhos comparativos baseados em *benchmarks* e que envolvem apenas as ferramentas de optimização e implementação directa nos componentes [Small, et al. 1993], ou ainda componentes de um mesmo tipo (FPGA) e, por esse motivo, mais facilmente relacionáveis [Weinmann, et al. 1991]. Um dos objectivos desta tese é contribuir para a sistematização da aplicação da síntese de alto nível em CPLD e FPGA. Assim, atendendo à sua importância, estes dois tipos de componentes são em seguida analisados com maior detalhe, através de dois exemplos concretos - a família de componentes MACH da AMD como exemplo dos CPLD, e a família de FPGA da Xilinx.

3.4.3 Exemplo dos CPLD

A AMD introduziu a sua família de componentes MACH no final de 1990 [AMD 1995]. Os primeiros elementos foram designados por subfamílias MACH1 e MACH2, dispondo estes últimos do dobro da capacidade lógica interna, para o mesmo número de pinos. O número de pinos varia entre 44 e 84. Os componentes mais rápidos operam à frequência máxima de 133MHz. Todos os componentes suportam lógica sequencial síncrona e assíncrona, mas neste último caso com algumas limitações.

A figura 3.4 mostra um diagrama de blocos que ilustra a estrutura interna dos componentes MACH, a qual consiste basicamente num conjunto de blocos do tipo PAL interligados por um módulo de comutação, designado por *Switch Matrix* (SM). Por programação, a SM permite ligar cada entrada a um elevado número de saídas, que constituem um grupo entre vários possíveis. A conectividade implementada é suficiente para que o programa de síntese crie a ilusão de uma interligação completa. De notar que os sinais de *clock* globais, para além de alimentarem os blocos PAL, também podem ser utilizados como sinais de entrada normais.

Para os MACH1 e MACH2 o número de blocos de PAL varia entre dois e oito. É na estrutura dos blocos de PAL que se verificam as diferenças entre as duas subfamílias. Enquanto que nos MACH1 cada bloco PAL tem apenas 12 ou 16 macrocélulas de saída (OMs na figura 3.4), nos MACH2 cada bloco de PAL tem 6 ou 8 macrocélulas de saída e 6 ou 8 macrocélulas internas (BMs na figura 3.4 - *Buried Macrocells*). As macrocélulas internas são especialmente úteis para implementar variáveis de estado em lógica sequencial e registos de entrada, visto que elas estão directamente ligadas aos pinos de I/O.

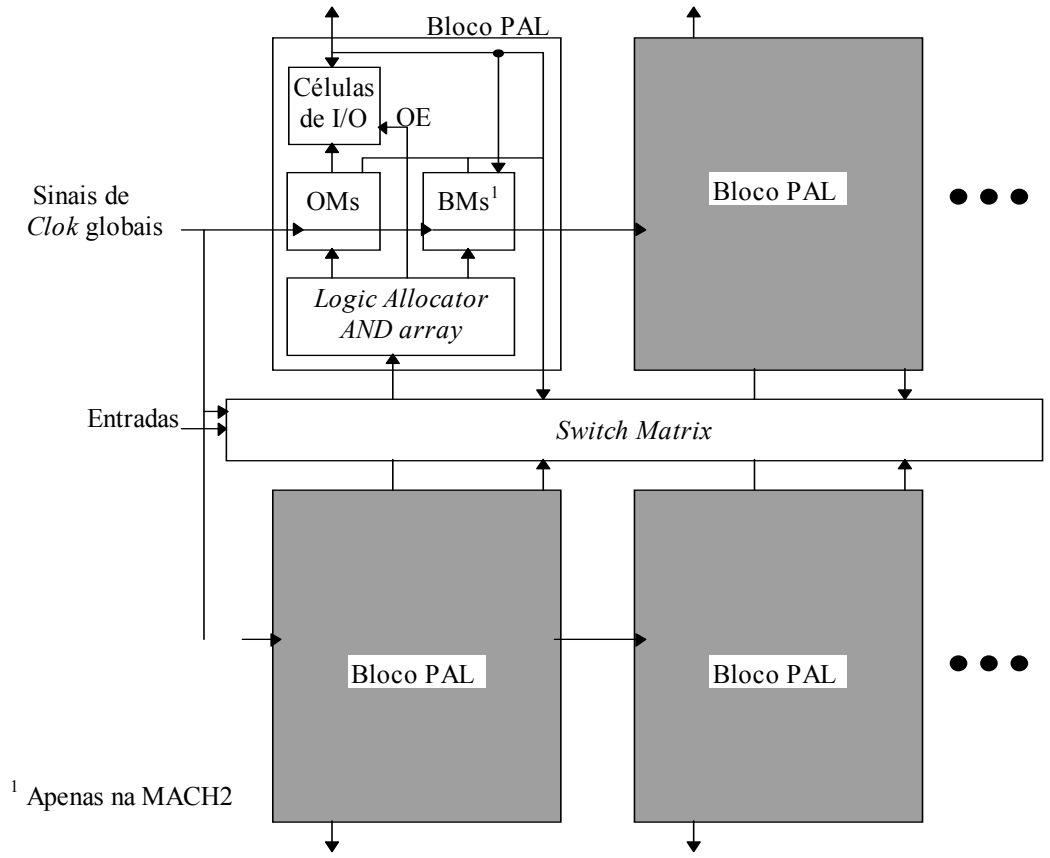


Figura 3.4 Diagrama de blocos dos MACH1 e MACH2

As saídas da SM alimentam o *AND Array* em cada bloco PAL, onde são gerados os produtos lógicos necessários. Numa estrutura PAL, um dos maiores problemas é a determinação do número de produtos que devem ser implementados para cada saída, visto que essa característica não é programável e influencia, por questões de espaço, o próprio número de saídas que se podem disponibilizar. Tradicionalmente, esse problema é resolvido utilizando uma de duas soluções: disponibilizando vários componentes com diferentes relações entre número de produtos e número de saídas; e atribuindo um número variável de produtos a cada saída, num mesmo componente. Nesta última solução, o programa de síntese pode escolher uma determinada saída para implementar uma função que melhor se adapte à sua dimensão. Para explorar este efeito é necessário que o utilizador não atribua *apriori* os pinos do componente a cada função (o que acontece com frequência).

Nos casos dos MACH, esse problema de organização foi resolvido dividindo todos os produtos em grupos de quatro, designados por *Product Term Clusters* (PTCs). Em cada bloco PAL existe um PTC para cada macrocélula (OM ou BM). No entanto, a ligação entre os PTCs e as macrocélulas não é feita directamente, mas sim através de um módulo designado por *Logic Allocator* (LA). O objectivo do LA é permitir alguma flexibilidade na distribuição dos PTCs. Tal como se pode observar na figura 3.5, o LA tem a capacidade de encaminhar o seu PTC de entrada para duas (nos MACH1) ou três (nos MACH2)

macrocélulas vizinhas. De uma forma equivalente, pode receber dos seus vizinhos os PTCs respectivos. Deste modo é possível implementar, em cada bloco PAL, funções lógicas de, no máximo, 12 produtos para os MACH1 e 16 produtos para os MACH2. Quando um LA disponibiliza o seu PTC, a macrocélula associada fica inutilizada. No entanto, no caso das OMs, o respectivo pino de I/O ainda pode ser utilizado como entrada.

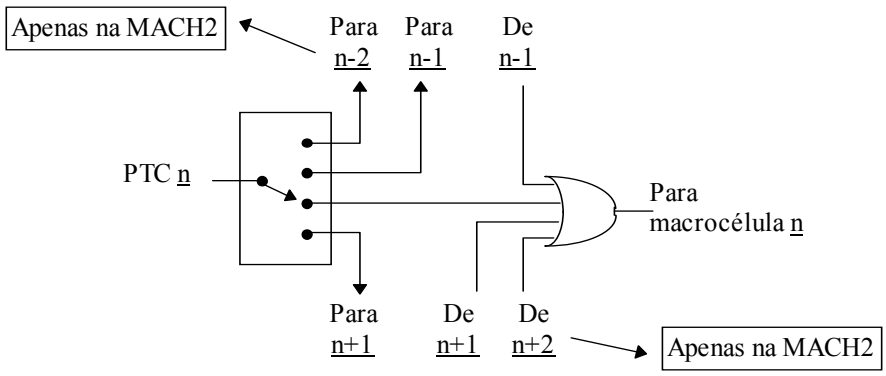


Figura 3.5 *Logic Allocator*

A organização física das macrocélulas é linear, o que implica que as macrocélulas situadas nos extremos do bloco PAL disponham de menor capacidade, já que não têm o mesmo número de vizinhos. Embora este arranjo seja mais flexível que qualquer das soluções anteriormente referidas, continua a verificar-se a necessidade de deixar ao cuidado da ferramenta de síntese a atribuição de pinos, já que a utilização da função de desvio de PTCs pode impedir a utilização de alguns pinos para saídas.

A macrocélula utilizada nos MACH1 e MACH2 tem muitas semelhanças com a macrocélula utilizada na popular PAL22V10, também introduzida pela AMD. A figura 3.6 mostra o esquema dessa macrocélula, onde é possível distinguir os diferentes modos de operação:

- Para lógica combinatória
- Flip-Flop do tipo D
- Flip-Flop do tipo T
- *Latch* (apenas nos MACH2)

Nas configurações para lógica sequencial, o sinal de *clock* pode ser seleccionado entre 2 ou 4 sinais externos, dependendo do componente específico, o que permite implementar circuitos assíncronos, dentro daquelas limitações. Para inicialização dos registos existem dois sinais de *Reset* e *Preset*, assíncronos e comums a cada bloco PAL, correspondendo a dois produtos de quaisquer variáveis de entrada nos MACH. De notar ainda que a saída da macrocélula realimenta sempre a SM e que, para os MACH2, existe um pino de I/O que pode ser ligado à entrada do elemento de memória, para implementação de registos de entrada.

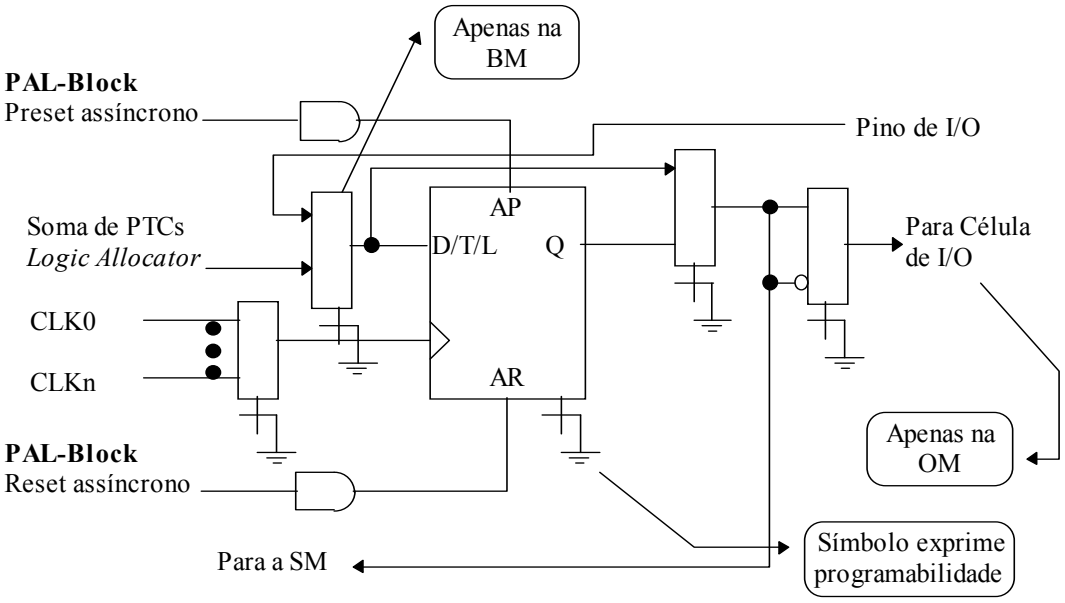


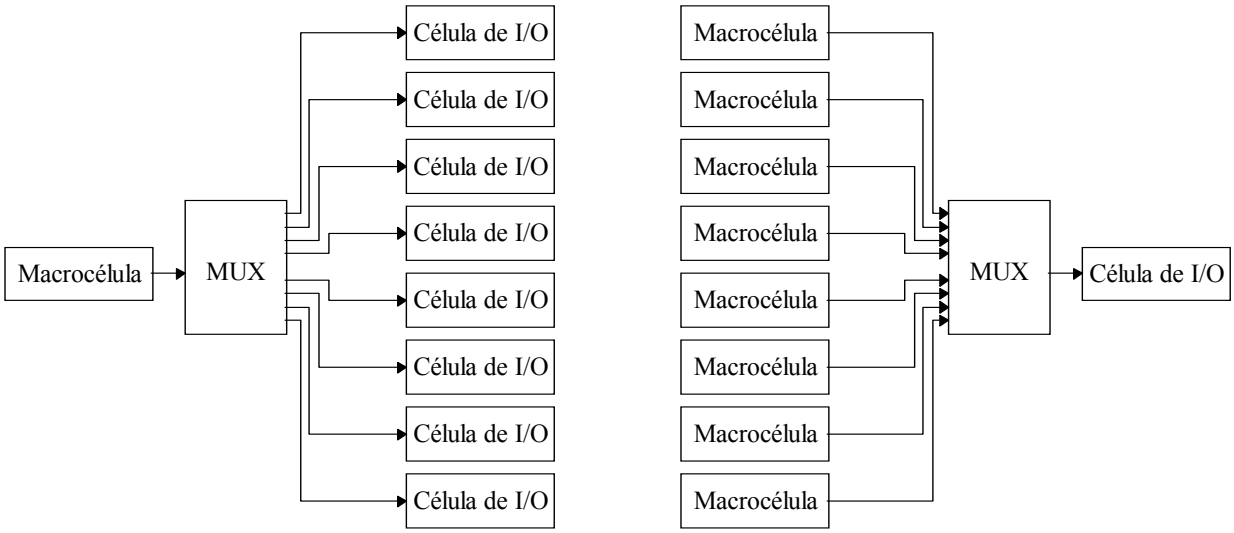
Figura 3.6 Macrocélula dos MACH1 e MACH2

As subfamílias MACH3 e MACH4 utilizam a mesma arquitectura, mas aumentam a funcionalidade, a densidade de lógica, o número de pinos (84 a 208) e as facilidades internas de interligação, por forma a conseguir uma melhor utilização do componente e um processo de desenvolvimento mais rápido. De uma forma resumida, as características mais inovadoras são as seguintes:

- Densidade lógica, considerando os componentes do topo da gama:
  $MACH3 \approx 2 \times MACH1$; $MACH3 \approx MACH2$
  $MACH4 \approx 6 \times MACH1$; $MACH4 \approx 3 \times MACH2$
- Inclusão de uma porta lógica XOR à saída de cada LA para implementação directa de lógica negativa. O controlo deste XOR é feito por um quinto produto pertencente ao PTC respectivo e que pode ser utilizado como um produto normal, caso não seja utilizado o XOR. Assim, cada PTC pode utilizar 5 produtos e, atendendo às possibilidades de encaminhamento do LA, idênticas às dos MACH2 (ver figura 3.5), uma função pode dispor de um máximo de 20 produtos.
- Possibilidades adicionais na configuração do Flip-Flop da macrocélula (tipo T, tipo J-K e tipo S-R), o que permite mais alternativas ao nível da optimização.
- Maior número de sinais de *clock* externos, com maior flexibilidade na forma como esses sinais vão actuar (nível e bordo) e possibilidade de usar como *clock*, em qualquer macrocélula, um produto lógico interno, o que permite a implementação de circuitos assíncronos sem qualquer limitação quanto à geração do *clock*. No entanto, essa facilidade reduz para dois o número de produtos no PTC da macrocélula respectiva.
- Maior flexibilidade nas ligações internas e nas ligações da lógica interna aos pinos de entrada e de saída.

- Existência de um registo associado a cada pino de I/O, o que permite a síntese de registos de entrada sem desperdiçar macrocélulas.
- Implementação da norma IEEE 1149.1-1990, habitualmente designada por JTAG, para teste de circuitos utilizando o conceito de *Boundary Scan*. Esta facilidade apenas existe nos componentes com mais de 84 pinos. Os pinos usados para esse fim (*Test Access Port* - TAP) podem igualmente ser utilizados, num modo de funcionamento extra, para programar o componente.

Dos melhoramentos enunciados e para além do aumento de densidade, o que mais impacto poderá ter num processo de síntese automática, e com vista à prototipagem, é a flexibilidade na selecção das ligações aos pinos de I/O. Entre as macrocélulas e os pinos de I/O existe um módulo designado por *Output Switch Matrix* (OSM). A estrutura lógica do OSM está esquematizada na figura 3.7, para uma macrocélula e um pino de I/O. Cada célula de I/O pode ser ligada a qualquer macrocélula de um grupo pré-determinado de 8 e, por sua vez, cada macrocélula pode ser ligada a qualquer célula de I/O também de um grupo pré-determinado de 8. A OSM proporciona assim uma elevada flexibilidade na atribuição de pinos e facilita alterações ao nível do projecto sem afectar, na maioria dos casos, uma pré-atribuição de pinos. Tal como já foi referido, esta funcionalidade revela-se muito importante em prototipagem.



a) Cada macrocélula alimenta qualquer uma de 8 células de I/O

b) Cada célula de I/O é alimentada por qualquer uma de 8 macrocélulas

Figura 3.7 Estrutura lógica da OSM (*Output Switch Matrix*)

Relativamente às características variantes entre as subfamílias MACH3 e MACH4, verifica-se a continuidade da filosofia subjacente à definição das subfamílias MACH1 e MACH2. Os MACH3 contém o mesmo número de macrocélulas e de células de I/O, ou seja, não dispõe de BMs, enquanto que nos MACH4 o número de macrocélulas existentes

em cada bloco PAL é o dobro do número de células de I/O. Isto significa que metade das macrocélulas são internas, do tipo BM. Atendendo à existência da OSM e à consequente independência das macrocélulas relativamente aos pinos de I/O, a escolha das BMs é, neste caso, dinâmica, tornando a estrutura ainda mais versátil em termos de utilização. Como seria de esperar o *software* de desenvolvimento encarrega-se de todos os detalhes de implementação, criando um componente virtual bastante flexível, que por sua vez liberta a atenção do utilizador para as questões funcionais a um nível de abstracção mais elevado.

A tabela 3.2 mostra as características estruturais mais relevantes para o componente mais complexo de cada subfamília. A densidade de cada componente é expressa pelo número equivalente de *PLD gates*. Por falta de uma definição exacta deste parâmetro na documentação técnica disponível, os valores apresentados devem ser usados apenas para efeitos comparativos entre os MACH e não num contexto mais global (bastante útil) de comparação com outros componentes.

Componente	Pinos	Macrocélulas | blocos PAL	*PLD gates*	Máx. entradas	Máx. saídas	Máx. Flip-Flops	Velocidade
MACH131	84	64 | 4	1.800	70	64	64	7…20 ns
MACH231	84	128 | 8	3.600	70	64	128	7…20 ns
MACH355	144	96 | 6	3.500	102	96	96	15, 20 ns
MACH465	208	256 | 16	10.000	146	128	384	12…20 ns

Tabela 3.2 MACH: componentes mais complexos de cada subfamília

3.4.4 Exemplo dos FPGA

Ao contrário dos CPLD, que implementam funções lógicas com base em estruturas do tipo AND-OR, todas os FPGA da Xilinx fazem-no através de pequenos módulos de memória ($2^n \times 1$) que funcionam como *look-up-tables*, podendo implementar qualquer função lógica combinatória das suas n variáveis de entrada. A memória está contida numa célula, a qual inclui ainda um multiplexer que permite ligar a saída da memória a um Flip-Flop do tipo D ou directamente à saída da célula. A saída do Flip-Flop está igualmente disponível à saída da célula. Na nomenclatura da Xilinx, aquela célula é designada por *Configurable Logic Block* (CLB) e, dependendo da família de componentes em questão, apresenta algumas alterações face à estrutura básica descrita, muito embora o princípio de funcionamento permaneça inalterado.

Cada componente é constituído por uma matriz de CLB, normalmente quadrada, variando a sua dimensão entre 8×8 e 32×32. O espaço compreendido entre os CLB é preenchido por uma elevada densidade de condutores horizontais e verticais, de tamanhos variáveis, com pontos de contacto programáveis para permitir ligar os CLB entre si e com as entradas e saídas. Nos cruzamentos desses condutores existem blocos de comutação, designados por *Switch Matrix* (SM), cuja finalidade é aumentar a conectividade dos sinais internos. Por questões de optimização relativamente aos tempos de atraso, existem alguns condutores que não passam pela SM, ligando directamente os CLB adjacentes ou atravessando totalmente a estrutura. Os sinais que passam pela SM dispõem ainda de

alguns *buffers* para compensar a capacidade eléctrica introduzida pelos comutadores da SM e assim minorar o seu efeito no tempo de resposta.

Na periferia do componente existem vários blocos de I/O para interface com os pinos do componente. Os blocos de I/O incluem um ou dois Flip-Flops (dependendo da família de componentes) permitindo a configuração de pinos como entradas, saídas, entrada/saída, e no modo directo ou com registo. A figura 3.8 mostra um esquema simplificado da estrutura interna de um componente, neste caso com uma matriz de 8×8 CLB e com 68 pinos. Actualmente, a Xilinx comercializa vários componentes de três famílias de FPGA, designadas por XC2000, XC3000 e XC4000 [Xilinx 1994].

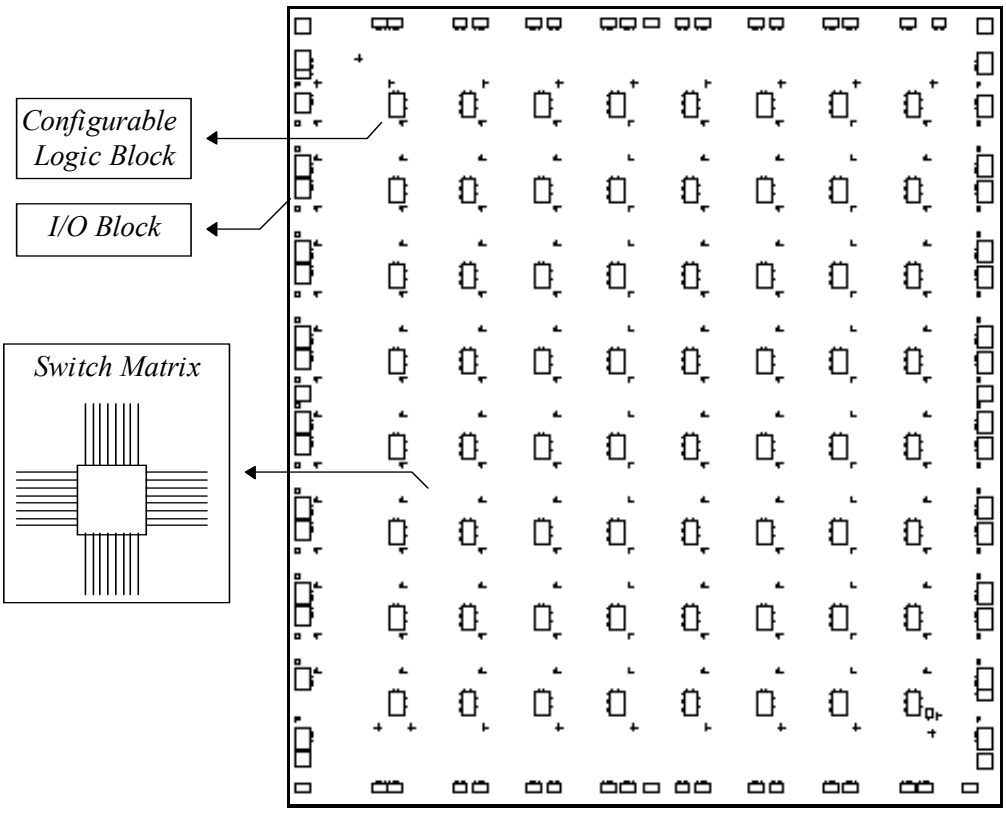


Figura 3.8 Estrutura interna (simplificada) de um FPGA da Xilinx

Todas as interligações e configurações são programadas através de células de memória estática, cujas saídas funcionam como sinais de controlo para multiplexadores ou transístores de passagem. A informação de programação é carregada em série, utilizando um modo de funcionamento especial, podendo igualmente ser lida.

A principal vantagem deste método é a flexibilidade conferida pela possibilidade de programar o componente, ou mesmo alterar a sua funcionalidade, sem o remover do circuito onde está inserido. Quanto às desvantagens, é vulgar identificar duas: do ponto de vista da arquitectura interna, aquelas células de memória exigem algum tipo de rede de distribuição para lhes aceder e ocupam espaço, o qual pode ser considerado desperdício no contexto da aplicação do componente; do ponto de vista da utilização, atendendo a que se

trata de memória estática, o seu conteúdo será perdido assim que for removida a alimentação, sendo necessário recarregar a informação de programação sempre que se liga o circuito. Esta última tarefa exige alguns cuidados do ponto de vista da concepção do sistema e pode ser feita por *software*, utilizando um microprocessador eventualmente disponível, ou através de uma PROM [Bostock 1993].

A arquitectura destes FPGA é substancialmente rica em registos e limitada em termos de geração de funções lógicas. Funções que exijam um número elevado de variáveis levantam problemas de implementação, por um lado, devido ao tempo de atraso elevado resultante da utilização de vários CLB ligados em série e, por outro lado, devido ao congestionamento que poderá existir nas ligações programáveis. Devido a esta dispersão de lógica e à consequente necessidade do adensamento de interligações, os algoritmos para mapeamento de funções e encaminhamento de sinais tornam-se mais complexos, com execução mais demorada e exigem frequentemente a intervenção do utilizador para conseguir soluções com melhor desempenho. De uma forma simples, pode-se concluir que esta é a contrapartida para obter um melhor aproveitamento dos CLB, quando comparado com o obtido pelas macrocélulas dos CPLD.

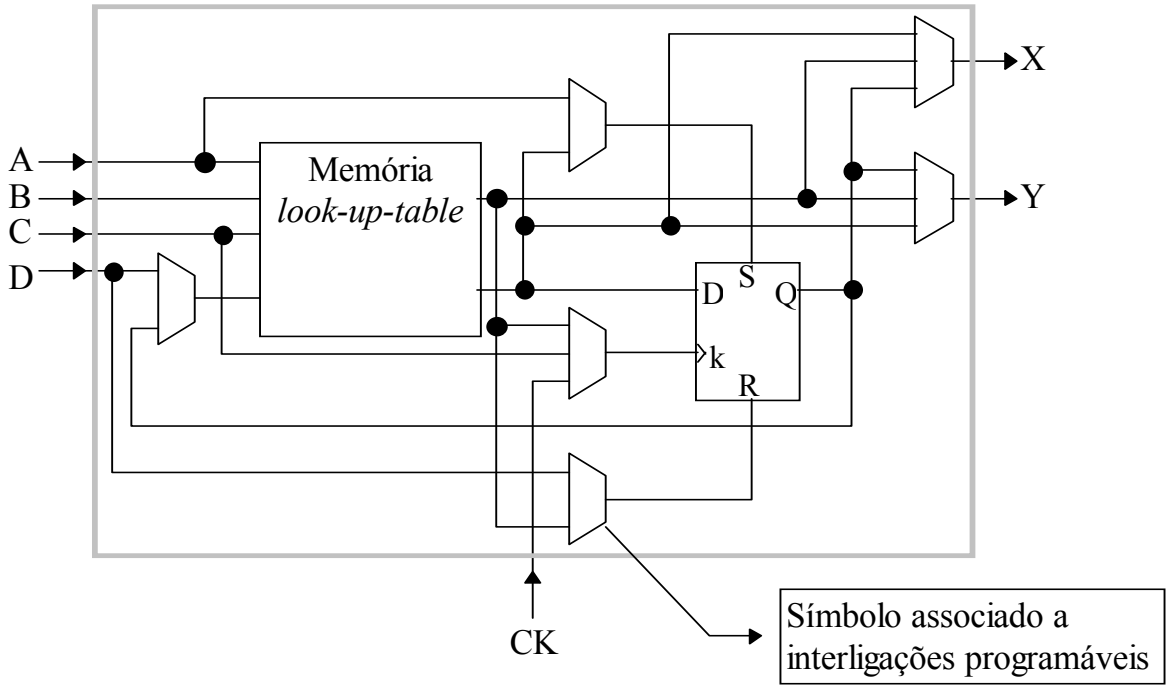


Figura 3.9 CLB dos componentes da família XC2000

A figura 3.9 mostra a arquitectura interna de um CLB, para os componentes da família XC2000, introduzida no mercado em 1985. Ao contrário do que sugere a figura, as duas saídas da memória associada à geração da lógica combinatória não são totalmente independentes, podendo suportar a mesma função das quatro variáveis de entrada, duas funções distintas de três variáveis, ou ainda a mesma função de duas possíveis de três

variáveis e cuja selecção é dinâmica. As duas saídas do CLB podem ligar-se a qualquer uma das duas saídas da memória, ou à saída do registo interno.

O registo interno pode ser configurado como um Flip-Flop do tipo D ou como uma *latch* do tipo D. A entrada do registo é uma saída fixa da memória interna, enquanto que o sinal de *clock* pode estar associado a uma entrada genérica do CLB (entrada D na figura 3.9), uma entrada dedicada para um sinal de *clock* global, ou pode ainda ser gerado na memória interna do CLB. Os sinais de *Set* e *Reset* do registo podem também ser gerados internamente, ou derivar directamente das entradas que alimentam a memória interna. Outro aspecto a realçar é a possibilidade da saída do registo ser usada como entrada na memória interna (em detrimento de uma das entradas genéricas), o que será especialmente útil na implementação de máquinas de estado e em situações em que o conteúdo do registo deve ser mantido aquando da activação do sinal de *clock*.

A figura 3.10 mostra a estrutura interna do bloco de I/O para a família XC2000. Como pode ser facilmente observado, estes blocos facilitam a implementação de registos de entrada sincronizados por um dos sinais de *clock* globais e permitem a configuração de pinos de entrada, de saída, ou de entrada/saída sob controlo de uma função lógica gerada no *array*.

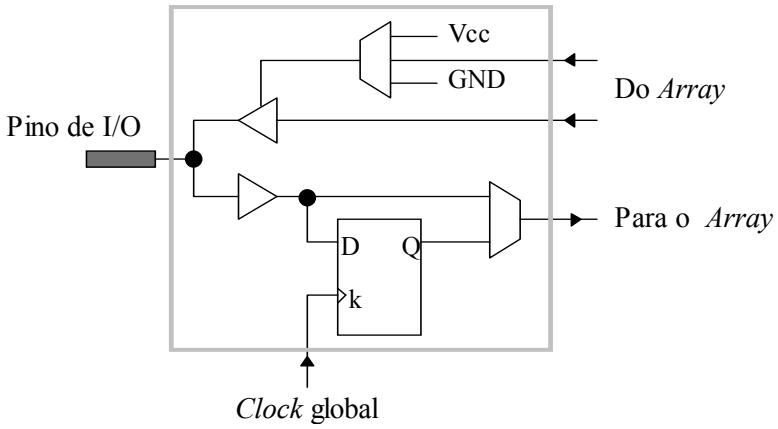


Figura 3.10 Bloco de I/O dos componentes da família XC2000

Desprezando as características físicas de implementação, existem apenas dois componentes na família XC2000. O XC2064 tem 64 CLB, 58 pinos de I/O e 12.038 bits associados à configuração, enquanto que o XC2018 tem 100 CLB, 74 pinos de I/O e 17.878 bits associados à configuração. Actualmente estes componentes são construídos para operar a frequências máximas de, aproximadamente 20 a 40MHz[3].

A família XC3000, introduzida em 87/88, é uma segunda geração da estrutura descrita, aumentando essencialmente a funcionalidade dos CLB e dos blocos de I/O. O

[3] Esta frequência refere-se ao limite típico, uma vez que a documentação técnica apenas refere valores absolutos para o registo interno do CLB, respectivamente 70 a 130MHz. A própria Xilinx admite que em aplicações típicas a máxima frequência de funcionamento dificilmente excederá 1/3 daqueles valores.

CLB inclui mais um registo e mais algumas entradas para o controlo dos registos e para a geração de funções combinatórias. A memória interna passa a ter cinco entradas, para produzir uma ou duas funções lógicas, seguindo o mesmo esquema adoptado para o CLB da família XC2000. Os registos são alimentados igualmente pelas saídas da memória, mas dispõem de uma terceira possibilidade que corresponde a uma entrada dedicada no CLB. Os sinais de controlo dos registos também sofreram algumas alterações. A entrada de *clock* deriva apenas de um sinal global e existe um sinal para autorizar a função do *clock* (realimentando o registo com a própria saída). A entrada de *Set* nos registos foi eliminada e o *Reset* deriva apenas de um sinal global.

No que respeita aos blocos de I/O, as principais inovações consistem na inclusão de um registo adicional para saída, a possibilidade de inverter o sinal de saída e o controlo do atraso temporal do sinal de saída. Os dois registos podem utilizar o mesmo sinal de *clock* ou dois independentes.

Em termos genéricos, a família XC3000 inclui 5 componentes (com algumas variantes, sendo a mais importante a capacidade de interligações internas), que incluem desde 64 até 484 CLB, 64 até 176 pinos de I/O e 14.779 até 94.984 bits de configuração. Actualmente, a máxima frequência externa varia aproximadamente entre 20 e 40MHz, para aquela que se pode considerar como a família básica, e até aproximadamente 80MHz para variantes com desempenho optimizado, pertencente ao que se pode designar por subfamília XC3190 (introduzida em 92/93).

A terceira geração de FPGA, designada por família XC4000, foi introduzida em 1992. A estrutura interna foi melhorada aumentando a capacidade de encaminhamento de sinais e diminuindo os atrasos no tempo de resposta, mas é em relação aos CLB que o aumento de funcionalidade se torna mais evidente. De uma forma resumida, as características inovadoras dos CLB são as seguintes:

- Duas memórias independentes, cada uma com 4 entradas, podem gerar quaisquer duas funções lógicas. Um terceiro gerador de funções lógicas é alimentado por aquelas saídas e mais uma entrada externa. Assim, um CLB pode implementar duas funções independentes de 4 variáveis, ou uma qualquer função de 5 variáveis, ou uma qualquer função de 4 variáveis juntamente com uma função limitada de 5, ou ainda uma função limitada de 9 variáveis. A possibilidade de implementar funções mais complexas num CLB vai reduzir consideravelmente os atrasos impostos nas outras duas famílias, resultantes do encadeamento de vários CLB.
- As duas memórias internas dispõem de um mecanismo configurável de propagação rápida de um sinal de transbordo, especialmente útil se o CLB for utilizado para implementar circuitos aritméticos ou contadores, por exemplo.
- Num modo opcional de funcionamento, as duas memórias podem ser utilizadas como RAM (16×2 ou 32×1), em vez da usual operação como *look-up-tables*. Esta característica é inovadora em termos de componentes lógicos programáveis, e abre perspectivas interessantes à síntese de alto nível.

- O mesmo sinal externo dedicado pode, por configuração, executar as funções de *Set* ou de *Reset* nos registos internos.
- As saídas dos registos internos estão disponíveis como saídas independentes do CLB, que assim passa a ter 4 saídas (2 combinatórias e as duas dos registos).

Os blocos de I/O também sofreram algumas alterações, face aos da família XC3000. Do ponto de vista electrónico é possível controlar algumas características temporais dos sinais de entrada/saída e verifica-se uma maior capacidade de drenagem de corrente. Do ponto de vista funcional, é bastante importante a inclusão de lógica adicional para permitir a execução de testes utilizando a norma IEEE 1149.1-1990, habitualmente designada por JTAG, para teste de circuitos utilizando o conceito de *Boundary Scan*.

Em cada um dos quatro lados do circuito integrado existe ainda um descodificador dos sinais de entrada dos blocos de I/O respectivos, para implementar as funções típicas de um sinal de selecção. Esses descodificadores podem ser logicamente divididos ao meio, podendo então gerar quatro a oito produtos lógicos disponíveis no interior do *array*.

A família XC4000 é formada por 12 componentes (com algumas variantes), que incluem desde 64 até 1024 CLB, 64 até 256 pinos de I/O, 31.628 até 422.128 bits de configuração e 2K até 32K bits de RAM. Actualmente, a máxima frequência externa é de cerca de 70MHz. A tabela 3.3 resume algumas das características mais relevantes dos componentes mais complexos de cada uma das famílias descritas. A listagem do componente XC3195, pertencente à subfamília optimizada XC3100, tem a ver com o facto de ele não ter um correspondente na família básica, constituindo assim uma excepção.

Componente	Pinos	CLB | entradas/CLB	*Typical gates*	Máx. I/O	Máx. Flip-Flops	Máx. frequência
XC2018	100	100 | 4	1K a 1.5K	74	174	40MHz
XC3090	208	320 | 5	5K a 7.5K	144	928	45MHz
XC3195	223	484 | 5	6.5K a 9K	176	1.320	80MHz
XC4025	299	1.024 | 9	20K a 25K	256	2.560	70MHz

Tabela 3.3 FPGA da Xilinx: componentes mais complexos de cada família

O parâmetro *Typical gates*, fornecido pelo fabricante, é obtido por estimativa e refere-se ao número de portas lógicas NAND de 2 entradas que o componente pode substituir, em aplicações típicas. Essa estimativa considera razões de utilização na ordem dos 60 a 90% e assume que cada CLB é equivalente a 12, 20 e 26 portas lógicas, para as famílias XC2000, XC3000 e XC4000, respectivamente. No caso particular da família XC4000, os blocos de memória consideram-se equivalentes a 120 portas lógicas, quando utilizados. Este valor não é considerado na tabela 3.3.

À semelhança do que se verifica para os CPLD com o seu indicador de recursos, também este parâmetro deve ser utilizado com algumas reservas. O estudo comparativo efectuado por Small (atrás referido) revela que a utilização efectiva do componente fica aquém daquele valor. Para obter essa conclusão o autor define "densidade de *gates*" como o quociente entre o número de *gates* (NANDs de 2 entradas) que se obtém da síntese e o número de blocos lógicos utilizados na implementação, no componente. No processo de

síntese o circuito lógico é aumentado por repetição até atingir o limite do componente, indicado pela ferramenta respectiva. Finalmente, a "capacidade efectiva" é calculada como sendo o produto da média das diversas "densidade de *gates*" obtidas com todos os circuitos, pelo "número total de blocos" do componente. Este estudo revela ainda uma outra conclusão importante relacionada com a estrutura do componente e os pesos relativos das componentes combinatória e sequencial do circuito a implementar. Segundo o autor, o máximo da "densidade de *gates*" é obtido quando a percentagem da componente sequencial do problema se aproxima da percentagem de Flip-Flops para o componente.

Uma das dificuldades num estudo deste género, e que pode influenciar significativamente os resultados, está relacionada com os ambientes de desenvolvimento. Para a mesmo descrição de um circuito existem múltiplas soluções, dependendo, entre outros factores, da forma como a solução é optimizada. Frequentemente os fabricantes fornecem bibliotecas de estruturas lógicas vulgarmente utilizadas (primitivas), às quais associam implementações optimizadas, explorando particularidades dos componentes. Quando o processo de projecto está baseado nesse tipo de estruturas é vulgar atingir índices de utilização mais elevados. No entanto, para utilizar essas primitivas é, geralmente, necessário elaborar especificações a um nível baixo, o que se traduzirá por uma menor clareza da especificação atendendo aos detalhes introduzidos e um maior tempo de desenvolvimento.

3.5. Síntese automática

Na secção 3.2 foi já introduzido o conceito de *High-Level Synthesis* (HLS), como sendo o processo de passagem automática de uma descrição comportamental, ao nível *Architecture*, para uma descrição estrutural ao nível *Register* ou *Logic*, a partir da qual é possível implementar directamente um circuito. Este processo será em seguida analisado em mais detalhe, nomeadamente no que respeita ao conjunto de passos tipicamente necessários e às questões mais importantes com ele relacionadas.

A passagem automática referida não tem necessariamente que ser feita num único passo. Muitos dos sistemas propostos distinguem pelo menos três subprocessos, associados a tarefas que podem ser completamente independentes [Micheli 1994]:

- *Architectural-level synthesis*, que tem como objectivo a transformação de uma descrição comportamental, ao nível *Architecture*, para uma outra equivalente, estrutural e ao nível *Architecture* ou *Register*. Esta tarefa passa pela identificação dos operadores usados na descrição comportamental e na selecção dos módulos de *hardware* (designados por recursos) adequadas para a sua implementação. Uma segunda função será a da definição das interligações necessárias entre os módulos de *hardware*. Este subprocesso é muitas vezes designado por *structural synthesis*, sendo ainda frequentemente identificado com o processo de HLS, sobretudo em sistemas que suportam apenas este tipo de síntese, exigindo a utilização de outra ferramenta para completar o processo de projecto.
- *Logic-level synthesis*, que tem como objectivo a transformação de uma descrição comportamental, ao nível *Register* ou *Logic*, para uma outra equivalente, estrutural e ao nível *Logic*. Esta tarefa está associada à

manipulação de uma especificação lógica (incluindo máquinas de estado, tabelas de verdade e equações booleanas) para criar um modelo estrutural que consiste na interligação de primitivas lógicas básicas. A associação destas primitivas com células reais de uma determinada tecnologia é, habitualmente, designada por *library binding* ou *technology mapping*.

- *Layout synthesis*, que tem como objectivo a criação de um *layout* físico para um projecto de um circuito integrado, a partir de uma descrição estrutural, ao nível *Logic*. No contexto do trabalho aqui apresentado esta tarefa não é relevante, pelo que não lhe será feita mais nenhuma referência.

Um sistema que integre todas estes subprocessos é habitualmente designado por *silicon compiler* [Walker e Camposano 1991] e representa o expoente máximo em termos de ferramenta de síntese. A sua caracterização completa e aplicação generalizada são os maiores desafios nesta área, uma vez que, estando disponível, essa ferramenta permite a criação de circuitos electrónicos a partir de concepções abstractas, com o mínimo de intervenção humana.

O processo de síntese é fortemente dependente do tipo de arquitectura que se pretende sintetizar. As questões relacionadas com a implementação de, por exemplo, barramentos, árvores de multiplexadores, microarquitecturas, *pipelines*, tratamento de excepções / interrupções e acesso directo à memória, apresentam características específicas que exigem algoritmos dedicados e, por vezes, linguagens ou métodos de descrição adequados. Um utilizador de um determinado sistema de desenvolvimento deve conhecer antecipadamente essas características, para obter melhores resultados e evitar especificar um circuito cuja síntese está, à partida, muito limitada ou mesmo impossibilitada.

3.5.1 Síntese ao nível *Architecture*

Uma descrição comportamental ao nível da arquitectura pode ser vista como um conjunto de operações cuja execução está restringida pelas dependências respectivas. Para poder inferir essa informação, a descrição comportamental deve primeiro ser transformada, por compilação, numa forma intermédia mais adequada. Os diagramas de fluxo (*data-flow graph* e *control-flow graph*) são o modelo de abstracção mais vulgarizado neste passo. Em seguida, é necessário identificar os recursos de *hardware* necessários para implementar as operações (*allocation*), escalonar as operações no tempo (*scheduling*) e associá-las aos recursos disponíveis (*binding*), definindo as interligações necessárias - o *data path*. Muitos dos algoritmos introduzidos para HLS incluem na operação de *allocation* a operação de *binding*. A unidade de controlo necessária para garantir a sequência de operações - o *control path* - é normalmente descrita directamente ao nível *Logic*, ou ao nível *Register*.

A optimização a este nível é extremamente importante e tem um forte impacto no desempenho e na área da solução final. As decisões tomadas neste processo definem o grau de paralelismo da solução e seleccionam os recursos, em número e tipo, num arranjo mais ou menos eficiente em termos da implementação. A semântica da linguagem descritiva utilizada neste processo tem igualmente um papel relevante, ao limitar, mais ou menos, a síntese e a optimização. De notar que, para completar a síntese do circuito, é necessário que exista uma descrição dos recursos ao nível *Logic* ou ao nível *Register*, ou alguma

ferramenta com capacidade para gerar essas descrições, a partir das quais é possível uma implementação física.

As diversas tarefas do processo de síntese descrito, nomeadamente *allocation, scheduling* e *binding*, têm sido alvo de investigação activa, existindo já um considerável número de algoritmos, com excelentes resultados (ver, por exemplo, [Gajski e Ramachandran 1994], [Walker e Chaudhuri 1995] e [Micheli 1994]) e alguns sistemas de desenvolvimento comerciais que os integram, como o MINT/SYNT e o ViewSynthesis, bem assim como sistemas resultantes de trabalhos de investigação, como o DSS [Roy, et al. 1992], o SAW, o CATHEDRAL, o HIS, o YSC (Yorktown Silicon Compiler), o PIRAMID, o OLYMPUS e o MIMOLA. Uma descrição sumária de um vasto conjunto de sistemas propostos, onde se encontram os citados, é feita em [Walker and Camposano 1991].

Apesar dos progressos verificados, para que este processo de síntese automática seja mais atractivo, é necessário ultrapassar alguns obstáculos resultantes da complexidade do próprio problema, como sejam o suporte de vários tipos de circuitos (a maioria dos sistemas tem capacidade para gerar apenas circuitos síncronos), a diminuição das limitações ao uso das linguagens (frequentemente apenas um subconjunto da linguagem é sintetizável) e a ausência de técnicas de verificação efectivas. A verificação é outra tarefa a necessitar de consideráveis melhoramentos, uma vez que a correcção por construção não será fácil de admitir em aplicações com centenas de milhares de linhas de código, e a simulação é um processo que apenas demonstra que o sistema funciona para os vectores de teste gerados.

Outros problemas de ordem mais específica persistem, nomeadamente os relacionados com as ferramentas e com a metodologia de especificação. Em relação às ferramentas, é de referir a falta de interactividade, a falta de transparência na definição das bibliotecas de recursos e as dificuldades no interface com as ferramentas que criam a implementação física (*back-end tools*). A necessidade da interactividade resulta da utilização frequente de métodos heurísticos na maioria dos algoritmos, tendo a intervenção humana um papel significativo na procura de soluções.

No que respeita à metodologia de especificação, o problema mais relevante é aquele que vulgarmente se designa por variação sintáctica (*syntactic variance*), associado ao facto indesejável de diferentes descrições comportamentais de um mesmo objecto poderem gerar soluções muito diferentes, em termos de optimização [Gajski and Ramachandran 1994]. Na exploração da especificação do controlador do GLiTCH, o problema da variação sintáctica será analisado.

3.5.2 Síntese ao nível *Logic*

A entrada para o subprocesso de síntese ao nível *Logic* é uma descrição de uma máquina de estados e/ou conjunto de equações booleanas, eventualmente com base em tabelas de verdade. Esta entrada poderá ser directamente especificada pelo utilizador através de uma HDL, ou de um diagrama esquemático, ou resultar da síntese de uma descrição ao nível da arquitectura. A tarefa de síntese pode ser diferente, dependendo do tipo de circuito (combinatório ou sequência) e da representação inicial, mas o resultado

será sempre uma descrição estrutural com base em primitivas elementares, materializada por uma lista de ligações (*netlist*) ou estrutura equivalente.

A optimização assume um papel importante, simplificando funções e explorando a possibilidade de reorganizar a estrutura num menor ou maior número de níveis lógicos, na procura do melhor equilíbrio área/desempenho, tendo em atenção um eventual conjunto de critérios pré-definidos. Historicamente, este processo automático foi o primeiro a ser investigado. Os problemas são bastante bem conhecidos, dividindo-se em quatro grandes áreas: optimização de lógica combinatória a dois níveis (especialmente adequada para estruturas lógicas do tipo PLA ou PAL); optimização de lógica combinatória multinível (especialmente adequada para estruturas lógicas do tipo *gate array*); optimização de lógica sequencial; e adaptação de uma estrutura às células disponíveis em biblioteca (*Cell-Libray Binding*). Para todos estes problemas existem algoritmos consideravelmente estáveis, o que se pode comprovar pela sua utilização vulgarizada na maioria dos sistemas comerciais existentes, como é o caso do ViewSynthesis, anteriormente referido. No que respeita a ambientes de desenvolvimento ligados à investigação, merece especial realce o SIS [Sentovich, et al. 1992], desenvolvido na Universidade de Berkeley, devido ao papel pioneiro que tem assumido na síntese ao nível *Logic*.

3.6 Linguagens para descrição de *hardware* (HDL)

Não obstante a elevada importância (legítima) atribuída à síntese, uma motivação, frequentemente assinalada para o uso de HDLs, é a necessidade de desenvolver modelos de circuitos de uma forma abstracta e utilizando metodologias familiares, principalmente herdadas da engenharia de programação. Na realidade, muitas das HDLs propostas são extensões de linguagens de programação, como é o caso da AHPL [Hill 1992], baseada na APL, ou da HardwareC [Ku e Micheli 1990], baseada em 'C', ou ainda da VHDL [IEEE 1994], derivada da ADA. No entanto, às HDLs é exigido um conjunto de características semânticas específicas, que as diferenciam entre si, resultantes da natureza do objecto que se pretende descrever. Tomando como referência uma arquitectura de um computador convencional, como a "máquina" de execução de *software* e o conjunto de circuitos digitais (*hardware*) que irão executar a tarefa especificada por uma HDL, as seguintes diferenças podem ser identificadas [Micheli 1994]:

- O *hardware* é concorrente por natureza - não tem uma arquitectura pré-definida - enquanto que o *software* descreve um modelo que tem subjacente uma arquitectura pré-definida, que normalmente é sequencial.
- Num modelo de *hardware* é frequentemente necessário descrever a estrutura da solução, em termos de blocos constituintes e suas interligações físicas. Assim, para além da possibilidade de poder descrever o comportamento de uma forma semelhante ao que é feito num linguagem de programação - domínio comportamental -, uma HDL deve também incluir a possibilidade de criar modelos no domínio estrutural.
- As questões de temporização das operações descritas em *hardware* são, muitas vezes, indissociáveis da própria função. Por esse motivo, é necessário incorporar numa HDL alguma forma de expressar as dependências temporais.

A primeira diferença enunciada merece mais alguma atenção, atendendo ao impacto que ela tem numa das principais funções de qualquer HDL, isto é, na simulação. Um número considerável de HDLs, e de ferramentas de simulação associadas, assentam no conceito de um tipo de modelo designado por *discrete event-driven*. Segundo este modelo, um circuito digital é um sistema discreto que é interpretado como um conjunto de **processos** concorrentes que comunicam por **sinais**. Em cada instante de tempo, cada processo pode estar em **execução** ou **suspenso**. A passagem do estado de execução para o estado suspenso é feita implicitamente após a execução do processo, ou explicitamente através de uma instrução dedicada. Sempre que estiver suspenso, um processo pode ser **reactivado**, o que se verificará quando ocorrer uma transição de valor (*event*) em qualquer dos sinais a que o processo é **sensível**. Embora a terminologia adoptada seja genérica, ela coincide com a que é utilizada na descrição da norma ANSI/IEEE 1076-1993, que define a linguagem VHDL [IEEE 1994].

Tal como foi anteriormente referido, a descrição de *hardware* deve poder ser feita a vários níveis de abstracção. Atendendo às diferentes representações exigidas a cada um desses níveis, não é fácil conceber uma linguagem que seja igualmente eficaz em todos os níveis/domínios de representação. Desde o início dos anos 60 que esta é uma área de investigação activa, sucedendo-se um vasto conjunto de linguagens, com características próprias, que muitas vezes procuravam solucionar um dado tipo de problemas. Por ordem cronológica, algumas dessas linguagens são a CDL, a DDL, a AHPL, a ISPS, a ISP, a TI-HDL, a Conlan, a ADLIB/SABLE, a ZEUS e a Verilog. Em [IEEE 1992a] e [IEEE 1992b], encontram-se vários artigos dos autores destas linguagens, nos quais exprimem as motivações que levaram à definição das linguagens e em que medida esse trabalho está relacionado com o estado actual de uma das linguagens mais vulgarizadas, a VHDL. De notar que, o simples facto de se verificar o aparecimento dum leque tão diversificado de linguagens revela as dificuldades na selecção das características que devem estar subjacentes à definição do modelo respectivo.

As primeiras linguagens propostas estavam orientadas para implementações específicas, e eram limitadas pela tecnologia disponível, permitindo a sua utilização num conjunto restrito de aplicações/tecnologias. Como exemplos, podem-se referir as linguagens PLPL [Kitson e Ow-Wing 1984] e Abel [Burrier 1984], ambas introduzidas para lógica programável e com capacidades de representação limitadas. Só mais recentemente começaram a aparecer linguagens com capacidade para suportar um maior número de níveis/domínios de abstracção. Como exemplos, podem-se apontar as linguagens Verilog, HardwareC e VHDL.

A evolução das HDL foi (é) dirigida essencialmente por dois vectores: a crescente quantidade de informação e a necessidade de automatizar a síntese de circuitos [Waxman 1986]. A maior quantidade de informação requerida no desenvolvimento de sistemas cada vez mais complexos (principalmente VLSI), exige uma melhor documentação para facilitar a reutilização e a troca de informação entre os elementos das equipas de projecto, também estas com dimensões cada vez maiores. A síntese automática de circuitos reflecte uma preocupação mais recente, atendendo a que a complexidade dos algoritmos e as quantidades de informação a processar exigem sistemas com recursos consideráveis, que só o desenvolvimento tecnológico tornou acessíveis.

Estes dois vectores apresentam exigências contraditórias. Por forma a promover a documentação e exploração de níveis de complexidade superiores, a maioria das linguagens evidenciam uma semântica mais rica do que a funcionalidade do *hardware* a que se destinam e uma expressividade superior à que é necessária para um processamento automático. Não é assim de estranhar o elevado número de restrições que é necessário aceitar para construir um modelo sintetizável numa dada linguagem. Este facto compromete de alguma forma o processo de desenvolvimento automático, ao exigir a construção de dois modelos diferentes (ou quase), um para simulação com o objectivo de validar a especificação e outro para síntese. Para além do esforço exigido na criação de dois modelos, é ainda necessário garantir a sua equivalência.

Tendo em atenção a multiplicidade de critérios que uma HDL deve verificar, não parece ser especialmente útil tentar estabelecer alguma comparação entre si. A selecção de uma linguagem deverá depender muito mais do universo da própria linguagem, isto é, das ferramentas disponíveis e das tecnologias de implementação suportadas. Neste aspecto as linguagens Verilog e VHDL são aquelas que evidenciam melhores argumentos [Micheli 1994]. Entre estas duas linguagens existe algum consenso relativamente ao maior poder expressivo da linguagem VHDL, mais abrangente em termos de níveis e domínios de abstracção [Camposano 1993], e a maior eficiência da linguagem Verilog em termos de simulação [Newton 1991].

3.6.1 VHDL

A definição da linguagem VHDL[4] foi iniciada formalmente em 1983, quando o DoD[5] efectuou um contrato para esse fim com três companhias: Intermetrics, Texas Instruments e IBM. O principal objectivo do DoD era ultrapassar problemas relacionados com a falta de normalização na documentação de sistemas digitais e a consequente dificuldade de comunicação entre diferentes equipas de projecto. A definição da linguagem, bem como a caracterização dos ambientes de desenvolvimento, terminou em 1985. A partir dessa primeira fase, e após verificar-se a estabilidade necessária, os produtores de sistemas de CAD começaram a interessar-se pelo desenvolvimento de ferramentas para suporte da linguagem. Essa actividade foi consideravelmente reforçada quando a linguagem VHDL foi normalizada pelo IEEE e reconhecida pelo ANSI, em 1987 (sobre essa versão foi efectuada uma primeira revisão, sendo a norma actualmente referida por norma ANSI/IEEE 1076-1993 [IEEE 1994]).

No início da década de 90, a linguagem VHDL entrou numa fase de consolidação. Sendo caracterizada inicialmente como uma linguagem multinível orientada à descrição de sistemas digitais, com capacidade de expressão estrutural e comportamental (neste domínio classificada como procedimental e imperativa), começaram então a suceder-se algumas actividades de investigação para alargar o âmbito da sua utilização, incluindo [Dewey e Geus 1992]: aplicação à HLS; a descrição física a um baixo nível de abstracção (necessária para sistemas analógicos ou mistos); a geração automática de modelos de teste - saliente-se a iniciativa WAVES (*Waveform and Vector Exchange Specification*); e, notavelmente, a descrição e síntese ao nível de abstracção do sistema, que permite acompanhar o aumento

---

[4] *Very high speed integrated circuits Hardware Description Language*

[5] *Department of Defense*, USA

de complexidade resultante da evolução tecnológica e do consequente aumento da escala de integração, diminuindo ainda o tempo de execução de projectos. No entanto, a linguagem VHDL evidencia algumas limitações na sua aplicação em descrições ao nível do sistema, nomeadamente a falta de um mecanismo que permita expressar hierarquia no domínio comportamental. A utilização deste paradigma é essencial num processo de desenvolvimento estruturado (*top-down*), a um nível de abstracção elevado.

Os principais objectivos que nortearam a definição da linguagem foram a geração de melhor documentação e a simulação precisa de sistemas digitais mais complexos. Relativamente à documentação, a normalização por si só constitui um passo importante nesse sentido, mas para alcançar maior eficiência é também necessário procurar regulamentar aspectos muito subjectivos, como por exemplo o estilo de escrita usado no desenvolvimento de modelos. As instituições envolvidas na normalização da linguagem não demonstraram preocupações sobre este tópico, o mesmo não acontecendo com algumas organizações que necessitam de utilizar descrições em VHDL produzidas por outras empresas, como é o caso da ESA-Estec[6]. Esta organização impõe regras contratuais à forma como as descrições em VHDL devem ser realizadas e ainda a algumas características que não devem ser utilizadas. Algumas das razões apontadas são a definição pouco precisa da semântica associada, donde poderiam resultar interpretações ambíguas, e a ineficiência de algumas construções em termos de tempo de simulação [Sinander 1994] e [Cohen 1995].

Os objectivos referidos foram, de certa forma, responsáveis pelas maiores dificuldades apontadas à utilização da linguagem em HLS. Para melhorar a documentação, a linguagem foi enriquecida com um elevado poder expressivo, excessivamente verbalizada, o que não é desejável para processamento automático. Para facilitar a simulação, foi introduzido um modelo temporal bastante elaborado, o qual permite, por exemplo, a utilização de mecanismos de sincronismo (entre processos) baseados nos tempos de execução da simulação. Tais mecanismos não podem ser sintetizados, em última análise, porque numa descrição comportamental não são conhecidas as características do *hardware* final, do qual vão depender os atrasos temporais. Para além disso, ao aplicar um mecanismo de HLS é possível que as operações descritas num modelo sejam reordenadas durante a optimização (ver secção 3.5), o que dificulta ainda mais a imposição de tempos de resposta fixos.

Uma das característica inovadoras da linguagem, e com grande impacto na especificação ao nível do sistema, é o suporte de ficheiros. Segundo a norma, um ficheiro pode conter valores de um dos tipos normalizados: BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING e TIME. A norma define ainda as operações elementares de escrita e leitura sobre o ficheiros, que devem estar implementadas no "**package TEXTIO"**. É ainda possível alargar a utilização de ficheiros para outros tipos, mas a responsabilidade de implementar as respectivas subrotinas de suporte é do utilizador. No entanto, a norma evidencia algumas lacunas, como por exemplo a ausência de uma definição do formato físico de armazenamento, o qual é livremente definido pelos fabricantes das ferramentas de CAD, comprometendo a portabilidade dos ficheiros [Bergé, et al. 1993].

---

[6] *European Space Agency - European space research and technology centre*

3.6.2 VHDL para síntese de alto nível

A riqueza semântica de uma linguagem sequencial genérica, com a qual se pretende construir um modelo abstracto de um sistema ou subsistema, considera naturalmente muitas facilidades que não podem ser implementadas directamente em *hardware*, pelo menos tal como o conhecemos hoje. Recursividade e utilização dinâmica de recursos, por exemplo, são características não adaptadas à natureza estática do *hardware*. Por este motivo, a primeira preocupação ao implementar um sistema de HLS é a definição de um conjunto de problemas tratáveis, ou seja, das características das arquitecturas a gerar. Um conjunto típico de restrições é o seguinte [Camposano, et al. 1991]:

- Apenas sistemas síncronos podem ser sintetizados. Objectivamente, esta restrição pretende evitar as realimentações em circuitos (ou partes de circuitos) combinatórios. Numa estrutura típica, o tempo é dividido em intervalos iguais e registos são utilizados para manter os dados estáveis durante um intervalo de tempo (desprezando os atrasos), para eventual processamento por uma rede de lógica combinatória.
- A síntese produz uma máquina de estados para controlar a execução das operações especificadas em cada processo, ou subprograma. Essas máquinas de estado podem explorar o paralelismo implícito que existe nas operações, fazendo executar em cada estado duas ou mais operações não dependentes em termos de dados. O utilizador tem a possibilidade de explicitamente especificar concorrência, utilizando vários processos concorrentes. Tome-se como exemplo uma estrutura em *pipeline*, para a qual o utilizador deve especificar exactamente o que é efectuado entre cada estágio e a forma como são resolvidos os problemas globais, por exemplo, associados a interrupções.
- A especificação comportamental é feita através de um algoritmo expresso numa linguagem sequencial, tal como a linguagem VHDL (mais precisamente o seu subconjunto destinado a descrição comportamental). A síntese vai garantir a sequência de operações necessária para verificar a dependência entre os dados expressa no algoritmo. Desde que se verifique esse aspecto essencial, existe liberdade para, por exemplo, alterar a ordem das operações, "estender" ciclos, ou ainda partilhar registos por várias variáveis.

De notar que este conjunto (ou outro similar) de restrições não está implícito na definição do processo de síntese de alto nível, mas resulta apenas de limitações de implementação, que a investigação permitirá ultrapassar progressivamente.

Para além daquelas limitações de ordem genérica, uma linguagem como a VHDL, que foi desenvolvida sem o objectivo imediato de promover a síntese automática (apesar das referências nesse sentido na introdução do documento que descreve a norma [IEEE 1994]), necessita da introdução de algumas restrições à sua utilização em ambientes de HLS, o que é demonstrado, por exemplo, na descrição do sistema Silicon 1076 [Nagasamy, et al. 1992]. Algumas dessas restrições são bastantes óbvias, atendendo à sua natureza, enquanto outras são de mais difícil percepção e, frequentemente, com um impacto

relevante na síntese. Em seguida, é feita uma análise mais detalhada de vários problemas levantados por estas restrições.

Processos

Tendo como referência uma descrição comportamental, um modelo descrito em VHDL consistirá de um ou mais processos concorrentes, comunicando entre si através de sinais ou variáveis[7] globais.

Na sua forma mais simples, um processo corresponde a uma única instrução concorrente, do tipo CSA (*Concurrent Signal Assignment*) ou do tipo CSSA (*Concurrent Selected Signal Assignment*). Estas instruções são utilizadas para definir o comportamento de um sinal ou vector de sinais, normalmente em função de combinações lógicas de um conjunto de variáveis. Na maioria das aplicações, essa descrição pode ser implementada directamente por um multiplexador ou estrutura combinatorial semelhante, o que não deixa espaço para exploração da síntese ao nível *Architecture*, tal como foi definida. Na realidade, as descrições que utilizam apenas aquele tipo de instruções são feitas ao nível *Register* ou *Logic*, constituindo especificações de entrada para ferramentas de *Logic Synthesis*.

Na forma mais elaborada, um processo descreve as relações entre um conjunto de variáveis e sinais de entrada e um conjunto de variáveis e sinais de saída, através de uma sequência de instruções, semelhantes às que, vulgarmente, são utilizadas em linguagens de programação (procedimentais e imperativas), incluindo operadores condicionais, controlo de ciclos, invocação de funções e procedimentos, e atribuições a sinais e variáveis. Um processo pode incluir um bloco declarativo onde serão declarados, para utilização local, tipos, variáveis (sinais não são permitidos), funções e procedimentos.

No que respeita à simulação, os processos são concorrentes e assíncronos, e executam em tempo "zero" todo o seu conjunto de instruções ou até encontrar uma instrução **WAIT**. Nessa altura, o processo bloqueia até que um dos sinais na sua lista de sensibilidade altere o seu estado. A lista de sensibilidade tem que ser indicada explicitamente no cabeçalho do processo, ou na instrução **WAIT**. Nos modelos para síntese é vulgar impor como restrição a existência de uma só lista de sensibilidade, o que exclui a possibilidade de utilizar várias instruções **WAIT** e o cabeçalho do processo para descrever a lista de sensibilidade em conjunto com uma instrução **WAIT**. De notar que no caso da forma mais simples dos processos, descrita anteriormente, a lista de sensibilidade é implicitamente definida por todos os sinais utilizados no lado direito da instrução CSA ou CSSA. Como é fácil concluir, é na riqueza semântica da descrição sequencial dos processos que os algoritmos de síntese encontram melhor aplicabilidade.

Este modelo de simulação corresponde aproximadamente ao comportamento do *hardware*, excepto no que respeita ao tempo de execução que, no *hardware*, dependerá da implementação, ou seja, da aplicação dos algoritmos de síntese e em particular do *scheduling*. O maior impacto desta discrepância surge na comunicação entre processos, quando o projectista assume um determinado comportamento temporal com base no

[7] A utilização de variáveis globais apenas foi normalizada na revisão conhecida por VHDL'92

modelo de simulação, que uma vez sintetizado pode evidenciar um comportamento diferente. A solução para este problema é não assumir qualquer comportamento temporal na elaboração de um modelo e explicitar um protocolo de sincronização entre processos, ou então forçar a execução das instruções que envolvem a comunicação em determinados ciclos (utilizando várias instruções WAIT, por exemplo), caso tal se possa prever.

Instrução WAIT

A forma completa da instrução WAIT é a seguinte:

WAIT [**ON** lista_sensibilidade] [**UNTIL** condição] [**FOR** time_out];

A lista de sensibilidades identifica os sinais cuja transição de estado provocam a reexecução do processo. A cláusula "condição" impõe restrições adicionais para a reexecução do processo, normalmente limitando as transições de estado possíveis para a lista de sensibilidades. A cláusula "time_out" indica um tempo limite absoluto para a suspensão do processo. Na simulação, esta cláusula pode ser usada para especificar correctamente as operações a executar em cada período do *clock*, enquanto que do ponto de vista da síntese não gera *hardware*, sendo ignorada ou mesmo proibida. No entanto, a sua utilização disciplinada pode servir para condicionar o *scheduling*, impondo a execução de certas operações em determinados períodos de *clock*, tal como é desejável na especificação de protocolos.

É vulgar restringir a lista de sensibilidades a um sinal de *clock* e a um sinal de *Reset* e/ou *Preset* associados, o que na prática corresponde a geração de modelos de processos síncronos, com sinais de inicialização assíncronos. Nesse caso, a cláusula "condição" indicará o bordo activo do sinal de *clock* e o nível a que os sinais de inicialização estarão activos. Com esta informação é possível escolher o tipo de registo adequado para a implementação.

Quando é possível descrever lógica combinatória através de processos, a lista de sensibilidades é normalmente obrigada a conter todos os sinais que são usados como entradas em atribuições e em condições.

Sinais e variáveis

Os sinais e as variáveis são duas classes de objectos básicos da linguagem. Os sinais são utilizados para modelar ligações físicas. As variáveis, por seu lado, servem para modelar registos internos, ou simplesmente como auxiliares em operações complexas, para conter resultados parciais, caso em que não necessitarão de estar associadas a nenhum *hardware*. As variáveis são exclusivamente locais aos processos[8], pelo que a comunicação entre processos só pode ser feita por sinais. No que diz respeito à síntese, a implementação de variáveis não traz qualquer problema, o que não se verifica com os sinais cuja utilização necessita de alguns cuidados.

[8] Na realidade a revisão VHDL'92 permite variáveis globais, mas tal facilidade ainda não se encontra implementada nas ferramentas actualmente disponíveis, não sendo então relevante considerá-la neste contexto.

Muitas das características dos sinais estão relacionadas exclusivamente com o modelo de simulação, ao pressupor um conhecimento detalhado de uma determinada implementação física. Em particular, deve-se ter em atenção que:

- cada sinal tem uma "história", que corresponde ao seu comportamento temporal passado, e uma "previsão", que corresponde a um conjunto de pares **(tempo, valor)** definidos para o futuro do sinal;
- as atribuições a sinais podem ser feitas com dois mecanismos, *transport* e *inertial*. No primeiro caso, o sinal assumirá escrupulosamente todos os valores previstos, enquanto que, no segundo caso, evidenciará uma inércia, que lhe permitirá rejeitar pulsos de duração inferior a um dado limite; e
- os sinais podem ser interligados a barramentos, utilizando um mecanismo de resolução de valor descrito numa *função de resolução*, a qual pode ser especificada pelo utilizador. Na maioria das ferramentas disponíveis existe uma *função de resolução* associada aos tipos **std_logic** e **std_logic_vector** (vector unidimensional do primeiro tipo). O tipo **std_logic** (que inclui os valores '0', '1', 'Z' e 'X') está vulgarizado na comunidade de utilizadores, para modelar sinais que podem assumir o estado de alta impedância ('Z') ou o "estado" *don't care* ('X') [Ashenden 1990] e [Cohen 1995].

A maior parte dessas características estão associadas a atributos, como por exemplo **'STABLE** e **'DELAYED**, cuja utilização é obviamente ignorada ou mesmo proibida em descrições para síntese. A *função de resolução* é obrigatoriamente utilizada num modelo que inclua um ou mais barramentos e apenas vai servir a simulação, já que a síntese não pode "obrigar" ligações físicas a evidenciar determinado comportamento. Nestas situações, é possível que a simulação produza resultados diferentes no modelo simulável e no modelo sintetizado.

A semântica da linguagem obriga a que sinais e variáveis modificados por um processo mantenham o seu valor entre reexecuções. Por esse motivo, no caso mais geral da síntese de processos que incluam um sinal de *clock* na sua lista de sensibilidade, os sinais e variáveis por eles modificados serão associados a registos, excluindo naturalmente as variáveis que são inicializadas em cada execução do processo e os sinais que, ao interligar registos, não necessitam de manter o seu valor entre impulsos do *clock*. Uma situação menos óbvia ocorre em processos combinatoriais, quando os sinais e variáveis são modificados utilizando operações condicionais e não é explicitamente indicado o comportamento para todas as condições possíveis das variáveis envolvidas. Nestas situações, os respectivos sinais e variáveis são implementados com *latches* por forma a poder manter o seu valor para as condições não explicitadas. Esta situação pode originar uma considerável quantidade de componentes desnecessários.

A semântica das variáveis é ligeiramente diferente quando são utilizadas dentro de funções ou procedimentos. O seu valor não é mantido entre chamadas, sendo implicitamente inicializadas caso o não sejam de uma forma explícita. Em qualquer das situações, o seu tratamento é semelhante às variáveis associadas a processos e que são explicitamente inicializadas.

Subrotinas

A linguagem VHDL permite chamadas recursivas em subrotinas e, tal como já foi referido, esse mecanismo ainda não é suportado pelas ferramentas de síntese. Para além desta limitação óbvia, convém ainda realçar a forma como as subrotinas são normalmente utilizadas e em que medida condicionam o processo de síntese.

A semântica da linguagem é extremamente rígida nas atribuições de sinais e variáveis, não permitindo a conversão automática entre tipos de dados, mesmo quando a relação está definida de uma forma clara, como é o caso dos valores binários e inteiros. Por esse motivo, todas as conversões necessárias têm que ser implementadas através de subrotinas (muitas vezes contidas em bibliotecas fornecidas com a ferramenta) ou através da facilidade de redefinição dos operadores, que na prática são também subrotinas. Este tipo de subrotina não gera *hardware*, sendo utilizado apenas para identificar equivalências entre valores de tipos diferentes. Em última análise, a ferramenta de síntese terá que relacionar todos os tipos com o tipo binário, para o qual dispõe de estruturas de *hardware*. Na ausência de uma relação explícita é assumida uma determinada relação, o que de certa forma contraria a rigidez semântica anteriormente referida.

As subrotinas são igualmente utilizadas para definir uma forma particular de hierarquia comportamental em processos. Neste contexto, as ferramentas de síntese usam-nas normalmente como base para a partição, gerando um módulo de *hardware* independente associado à subrotina e implementando um mecanismo apropriado para a passagem de parâmetros. Se o utilizador não pretende pré-definir partições, deverá evitar o uso de subrotinas e optar pela expansão de código. A avaliação correcta da estratégia a assumir passa pela conhecimento prévio da frequência da utilização da subrotina e do volume de lógica associado.

Conclusão

A análise feita não aborda todas as restrições impostas à utilização da linguagem VHDL em HLS, mas apenas aquelas que são mais relevantes na perspectiva do autor e como resultado da experiência de utilização, tanto pelo impacto que têm na síntese, como pela forma obscura com que por vezes se apresentam. Uma análise semelhante e mais alargada, pode ser consultada em [Camposano, et al. 1991]. Para além das restrições da linguagem, a maioria das ferramentas de síntese ainda acrescenta mais restrições especificas, que podem ser classificadas como de implementação. O conhecimento dessas restrições é imprescindível para a utilização e caracterização da ferramenta, atendendo a que é possível que sejam impostos limites relevantes à capacidade de abstracção da linguagem. Por este motivo, na secção seguinte é feita uma descrição sumária da ferramenta de síntese de VHDL utilizada no projecto do controlador do GLiTCH.

3.6.3 O ViewSynthesis

A ferramenta de síntese utilizada neste trabalho (ViewSynthesis [VIEWlogic 1992]) está integrada num ambiente de desenvolvimento de sistemas digitais (WorkView), sendo condicionada pela arquitectura e filosofia de utilização desse mesmo ambiente, que evoluiu com base num sofisticado editor de esquemáticos. Este facto promoveu a especificação ao nível da arquitectura, no domínio estrutural (através de esquemáticos), relegando a utilização da linguagem VHDL para a descrição e síntese de modelos de componentes para

inclusão nos esquemáticos. O nível de abstracção utilizado nestas situações é, tipicamente, o nível *Register*. Isto não significa que não seja possível construir modelos a níveis de abstracção mais elevados, só que com um suporte substancialmente reduzido, principalmente no que respeita à síntese.

Tal como foi referido anteriormente, uma das características importantes a analisar em ferramentas de síntese, é o tipo de arquitectura que pode ser gerada, o que se pode inferir através da análise das bibliotecas disponíveis. A aplicação ViewSynthesis inclui um elevado número de bibliotecas de componentes, para as tecnologias mais usuais. Contudo, essas bibliotecas contém essencialmente descrições de portas lógicas simples (nível *Logic*), com algumas excepções que incluem estruturas lógicas como multiplexadores, contadores e somadores/subtractores (nível *Register*). Este tipo de objecto na síntese está ligado, normalmente, à síntese ao nível *Logic*, que no contexto da HLS corresponde ao subprocesso de síntese anteriormente definido como *Logic Synthesis*.

A síntese de descrições comportamentais ao nível *Architecture* e/ou *Register* está limitada pela utilização de um subconjunto da linguagem VHDL. As limitações mais importantes podem ser resumidas do seguinte modo:

- sinais só podem ser declarados do tipo binário **vlbit**[9], ou como vectores de uma dimensão desse mesmo tipo, **vlbit_1d** ou **vlbit_vector**; esta limitação confina a exploração da abstracção da representação de dados ao interior dos processos, onde é possível declarar variáveis de qualquer tipo, excepto vectores de duas dimensões; mas, nesse caso, o utilizador tem sempre que explicitar as conversões para e do tipo binário na implementação do interface entre processos, atendendo ao imperativo de utilizar sinais para esse propósito;
- vectores de duas dimensões estão restritos à utilização com constantes e com funções de síntese dedicadas, que permitem a especificação através de tabelas tipo PLA; esta limitação significa que o processo de síntese não tem capacidade para gerar estruturas de memória, cuja representação óbvia são exactamente os vectores de duas dimensões;
- em processos sequenciais só pode existir uma instrução de WAIT, que tem que ser a primeira instrução e com a forma **WAIT until (***condição*); a condição tem que definir claramente a opção do utilizador quanto ao tipo de registo a utilizar, podendo ainda incluir um sinal de *reset* ou de *preset* assíncronos, mas cuja função tem que ser imediatamente definida com uma instrução "**IF … THEN … ELSE**"; e
- em processos combinatoriais não podem existir instruções WAIT e o cabeçalho desses processos tem que incluir uma lista de sensibilidade, com todos os sinais utilizados em expressões condicionais e no lado direito de instruções de atribuição.

Para além das limitações à linguagem, o processo de síntese obriga a que eventuais esquemas de repartição de operações por períodos de *clock* distintos sejam explicitamente

[9] Os valores associados a este tipo são '0', '1', 'Z' (para estado de alta impedância) e 'X' (para indicar a condição de *don't care*).

descritos nos modelos, utilizando, por exemplo, procedimentos com instruções WAIT (com as mesmas restrições que foram indicadas para os processos). Na prática isto significa que a ferramenta não implementa qualquer algoritmo de *scheduling*, o que reforça a sua limitada vocação para o processamento de descrições ao nível *Architecture*. A caracterização do processo de síntese disponível com esta ferramenta está sumariamente descrita na *Y-chart* da figura 3.11.

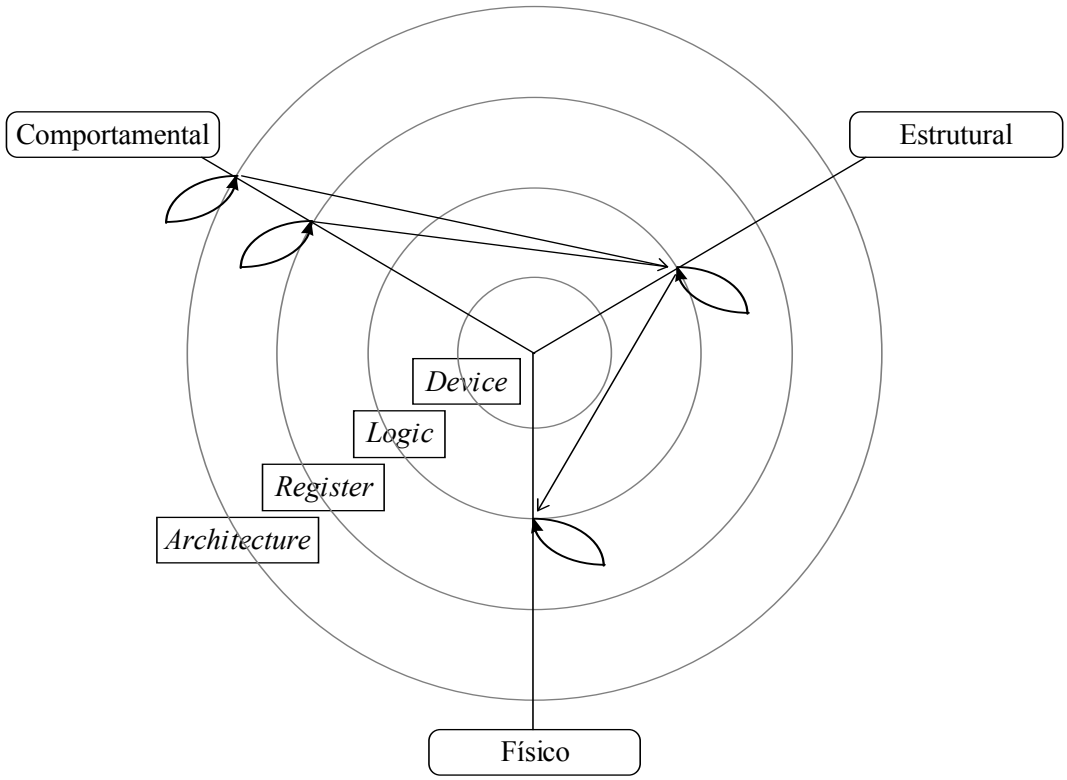


Figura 3.11 *Y-chart* para a síntese baseada na ferramenta ViewSynthesis

Um modelo para síntese é descrito no domínio comportamental, ao nível *Architecture* ou *Register*, o qual deverá ser simulado para aferir a descrição face à especificação inicial. Com base na biblioteca seleccionada pelo utilizador, a síntese desse modelo consiste em procurar uma solução óptima, segundo um dado conjunto de critérios. Essa solução é materializada por um conjunto de componentes e a respectiva lista de ligações, descrita no domínio estrutural e ao nível *Logic*. Nesta fase, o projecto é novamente simulado para validar o processo de síntese, devendo os resultados da simulação ser comparados com os da simulação anterior, para localizar eventuais discrepâncias. Por último, a descrição estrutural é processada por uma ferramenta específica para a tecnologia de implementação, para gerar o componente final. Dependendo dessa ferramenta específica ainda poderá ser possível simular o resultado desta última fase, o que para além de validar a implementação em termos funcionais permitirá ainda verificar a adequação das características temporais da solução. Esta verificação é particularmente importante quando aquela ferramenta específica tem a

capacidade de introduzir alterações para optimizar a implementação, o que se verifica na utilização de lógica programável.

A figura 3.12 mostra com maior detalhe o modo de operação do processo de síntese, desde a geração do modelo comportamental em VHDL, até à geração do modelo estrutural. Em particular, é importante realçar a possibilidade de iterar a tarefa de síntese propriamente dita, introduzindo novas condições de operação, critérios de optimização e restrições, em função dos resultados obtidos. As condições de operação correspondem a características físicas associadas ao processo de fabrico, tais como a tensão de alimentação, temperatura de operação e tipo de caixa. Para algumas tecnologias, estas características são alteráveis e a sua correcta selecção é necessária para obter um circuito fiável. No caso da lógica programável, estas condições de operação são irrelevantes, atendendo a que o circuito físico já foi construído.

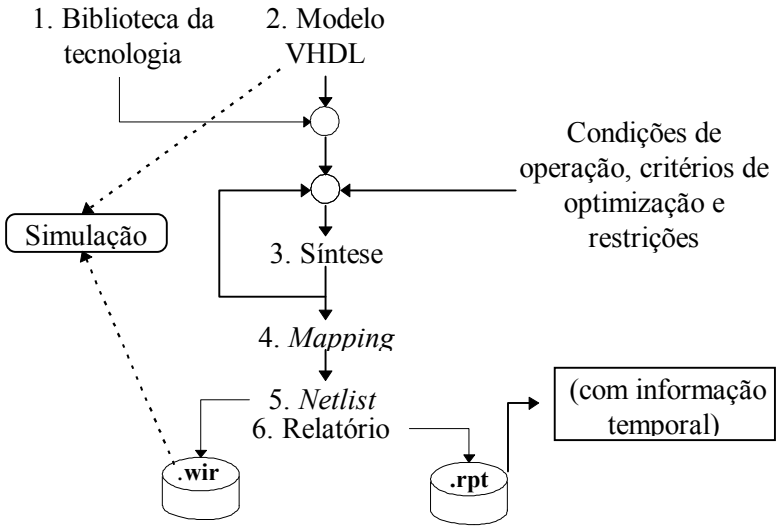


Figura 3.12 Detalhe da operação da ferramenta ViewSynthesis

Frequentemente, é possível identificar sinais que podem condicionar a operação global do sistema. Esses condicionalismos podem ser de ordem eléctrica ou temporal. No primeiro caso, o problema é a capacidade de carga das entradas e saídas, enquanto que, no segundo caso, o problema será o tempo de resposta. A ferramenta de síntese permite a inclusão dessa informação através de restrições. As restrições podem ser aplicadas a sinais individuais, a grupos de sinais (barramentos), ou ainda a todos os sinais. À semelhança do que se verifica com as condições de operação, também os condicionalismos eléctricos não se aplicam à lógica combinatória, e pelo mesmo motivo.

Os critérios de optimização são seleccionados por parâmetros que permitem explorar o espaço de projecto, definido pela área e velocidade de operação do circuito. Um parâmetro indica o tipo de lógica que se está a descrever, e pode assumir três valores: *small* (ou **fsm**) para lógica predominantemente sequencial, *large* (ou **data_path**) para lógica predominantemente combinatória, ou ainda *mixed* para circuitos em que não predomina nenhum daqueles tipos de lógica. Com base neste parâmetro, é seleccionada a técnica de optimização mais adequada, seguindo a mesma filosofia de operação da ferramenta SIS

[Sentovich, et al. 1992]. O outro parâmetro indica a importância relativa que o utilizador pretende dar à área, face à velocidade de operação. Este parâmetro pode variar entre os valores 1 e 100, significando o valor 1 máxima prioridade à área, e o valor 100 máxima prioridade à velocidade de operação. Este parâmetro afecta especialmente a simplificação de funções lógicas no que respeita à factorização e partilha de termos, tendo em conta o seu impacto no número de níveis de portas lógicas, para cada função.

A escolha de um conjunto de parâmetros adequado não é uma tarefa fácil, e passa necessariamente pela experimentação de valores alternativos. A tecnologia de implementação e o "estilo" de descrição comportamental (devido à *syntactic variation*), são os dois factores mais importantes a ter em conta nessa escolha. No entanto, através da experiência, deverá ser possível obter algum conjunto de indicadores que permitam mais facilmente proceder a uma escolha adequada. Nesse sentido, foi efectuado um estudo com o protótipo do controlador do GLiTCH, limitado à lógica programável, e aproveitando a existência de estruturas lógicas representativas de diferentes classes de circuitos. Esse estudo está descrito no capítulo 5.

3.7 Conclusão

O crescente aumento da complexidade dos projectos de sistemas digitais é uma consequência natural da evolução tecnológica. Para suportar esse aumento de complexidade, torna-se necessário abstrair o mais possível as especificações dos detalhes da implementação, o que por sua vez introduz novos problemas, na sua maioria relacionados com a necessária síntese automática (HLS). As linguagens de descrição de *hardware* (HDL) encontram neste domínio uma aplicação importante, porque, para além de permitirem aumentar o nível de abstracção, facilitam ainda o processamento automático. Das HDL propostas, a linguagem VHDL é a única normalizada, o que promoveu a sua divulgação. Este facto é corroborado pelo aparecimento de inúmeras ferramentas de CAD que suportam VHDL, na sua aplicação à modelação, simulação e síntese.

No que respeita às tecnologias de implementação, a lógica programável teve uma evolução notável, passando da utilização como mera substituição de alguns circuitos integrados de baixa escala de integração, para uma alternativa de implementação de ASICs. A lógica programável assume um papel particularmente importante na fase de prototipagem de um projecto, graças à sua flexibilidade em termos da possível alteração da funcionalidade do componente. Algumas famílias mais recentes de componentes expandem ainda mais essa flexibilidade, ao permitir que a reprogramação seja feita no próprio circuito - o que introduz ao nível do *hardware* um paradigma de processamento muito semelhante ao do *software*.

O projecto do controlador do GLiTCH constitui um exemplo de aplicação dos novos paradigmas de desenvolvimento, ajudando a sistematizar algumas tarefas e a verificar a validade de alguns pressupostos, nomeadamente sobre a aplicação da síntese automática tendo como alvo a lógica programável. Os capítulos seguintes descrevem a implementação do protótipo do controlador do GLiTCH, e o estudo efectuado sobre esse protótipo, com vista a analisar mais detalhadamente algumas das questões identificadas.

Capítulo 4

4. O controlador do GLiTCH

4.1 Introdução

O simulador desenvolvido para validar a arquitectura do GLiTCH serviu como o documento de especificação do controlador, muito embora não tenha sido construído com esse propósito. Para justificar essa utilização basta realçar que o simulador serviu como plataforma de desenvolvimento de vários programas de aplicação, cujo aproveitamento pressupõe uma plataforma de *hardware* compatível.

Do ponto de vista da síntese do sistema digital, a existência deste tipo de simulador impõe limitações importantes ao nível de abstracção a utilizar, atendendo a que ele introduz restrições e facilidades não relacionadas directamente com o APA e que portanto envolvem a especificação de outras unidades funcionais. Isto significa que o simulador faz pressupor a existência de algumas soluções estruturais que do ponto de vista do desenvolvimento do *software* são irrelevantes, mas que limitam a exploração de alternativas em *hardware*. As dificuldades impostas pela ausência de uma especificação mais adequada, bem como as suas consequências, encontram-se referidas em [Santos, et al. 1995].

Através da utilização do simulador e da análise dos documentos que o descrevem [Storer 1991b] e [Storer 1991b], foi possível identificar um conjunto de unidades funcionais e a respectiva especificação ao nível *Architecture*:

- uma unidade de memória para padrões binários, utilizados em instruções que exploram a propriedade associativa do GLiTCH; no simulador esses padrões binários são usados como constantes, não sendo assim evidente a relação endereço/padrão que terá necessariamente que existir no sistema real[1]; esta unidade deve estar dividida em dois bancos, Banco de Dados (BD) e Banco de Máscaras (BM); no que concerne à operação do sistema GLiTCH, em operações de leitura os bancos são lidos em simultâneo, em operações de escrita apenas o BD deve ser afectado; esta unidade funcional será frequentemente designada por *Data Store* (DS), por uma questão de compatibilidade com a documentação já produzida no contexto deste projecto de âmbito internacional, e ainda para simplificar a notação;
- uma unidade escalar para operações elementares ao nível do bit, que não se adaptam à arquitectura do processador vectorial GLiTCH; pelas razões

[1] No ambiente simulado esta relação é escondida pelo próprio compilador de 'C' através da utilização de constantes.

anteriormente apontadas, esta unidade funcional será frequentemente designada por *Scalar Unit* (SU);

- uma unidade de selecção para alguns sinais do APA (sinais de *Tag*) que se propagam por todo o *array*; as extremidades encontram-se normalmente interligadas, numa configuração em anel, mas algumas operações exigem que um determinado valor binário seja introduzido nessa cadeia, donde resulta a necessidade desta unidade de selecção; pelas razões anteriormente apontadas, esta unidade funcional será frequentemente designada por *Tag Router* (TR);
- um sequenciador de instruções, usado para garantir uma execução sequencial de microinstruções, com a possibilidade de escolha da microinstrução a executar (operações condicionais) com base em condições resultantes do funcionamento do resto do sistema; o sequenciador poderá ainda suportar a execução de ciclos - implementados no simulador por expansão do código; a micromemória deverá ter um tamanho de célula que permita especificar todas as operações das diversas unidades funcionais, podendo ser optimizada através da partilha de campos;
- um processador escalar para a implementação das funções de interface e operações escalares complexas (como por exemplo operações em vírgula flutuante) para as quais o GLiTCH apresenta fraco desempenho e cuja frequência não justifica o recurso a uma unidade de processamento dedicada; a utilização de um transputer facilitará a integração do sistema na arquitectura MIMD; pelas razões anteriormente apontadas, esta unidade funcional será frequentemente designada por *Host Transputer* (HT).

Ao nível da implementação, e considerando os algoritmos já desenvolvidos, o protótipo deverá executar à frequência máxima de 20MHz, por forma a satisfazer o objectivo de obter um sistema com capacidade para funcionar à razão de transferência do sinal de vídeo.

As operações das unidades funcionais utilizam frequentemente resultados produzidos pelo *array* de GLiTCHs, pelo que há ainda que atender ao funcionamento em *pipeline* (3 níveis) deste circuito, bem como à adição de mais níveis de *pipeline* que venham ser introduzidos na implementação do controlador.

Com base nesta especificação, foi possível construir o diagrama de blocos da figura 4.1, o qual mostra as diversas unidades funcionais e as relações que existem entre si. Do ponto de vista estrutural, esta arquitectura evidencia as propriedades de uma arquitectura do tipo VLIW, sendo de salientar a existência de relações complexas na sua operação conjunta (por exemplo, algumas instruções do GLiTCH exigem a execução prévia de uma instrução do DS, com uma determinada temporização). O sincronismo implícito nessas operações conjuntas deve ser garantido na geração dos microprogramas. No entanto, para facilitar essa tarefa, a arquitectura do controlador deverá procurar garantir, através do recurso a níveis de *pipeline*, que para a maioria das operações conjuntas, as instruções individuais de cada unidade funcional envolvida sejam incluídas numa única microinstrução. Atendendo aos diferentes fluxos de informação, este paradigma não poderá

ser implementado para todas as operações, devendo-se então privilegiar as mais utilizadas, que são as operações associativas de verificação de padrões.

O nível de representação utilizado no diagrama da figura 4.1 esconde alguns detalhes importantes, como por exemplo a estratégia de inicialização da micromemória ou da unidade de memória de padrões binários. Esses detalhes dependem naturalmente da implementação escolhida e serão devidamente definidos no processo de refinamento da especificação, na tarefa de síntese.

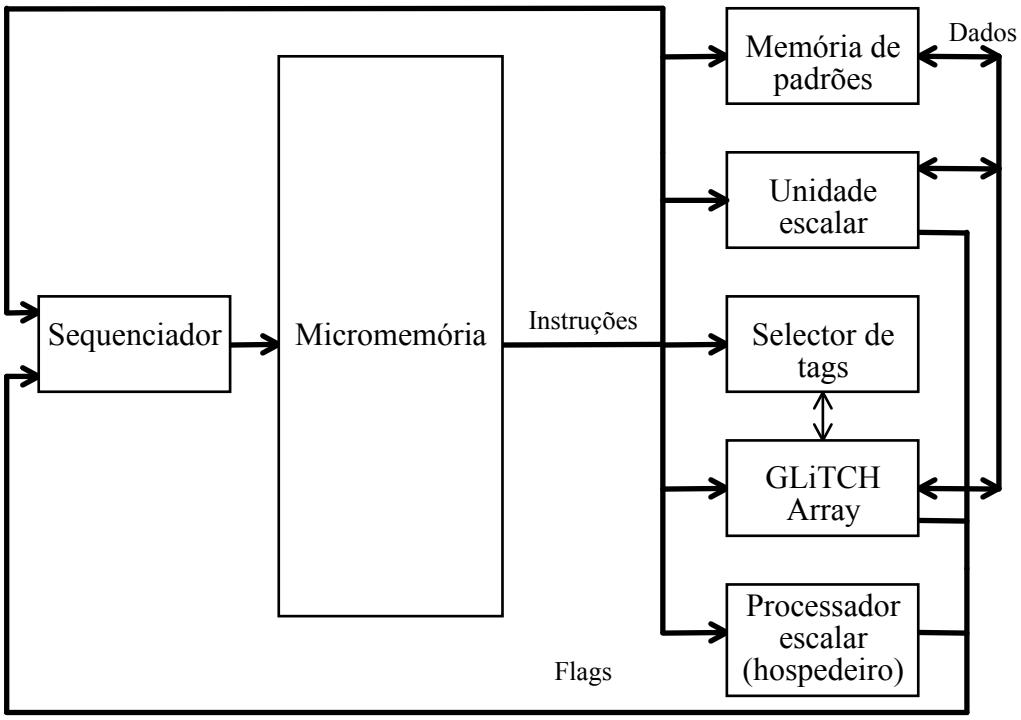


Figura 4.1 Diagrama de blocos do sistema

Na secção seguinte, é abordada a metodologia de desenvolvimento adoptada, a qual pode ser generalizada para o projecto de qualquer sistema digital, e nas secções seguintes são abordadas, em maior detalhe, cada uma das unidades que constituem o controlador, na perspectiva da sua descrição funcional. A descrição técnica pormenorizada do controlador, incluindo a implementação física (estrutural) do controlador, encontra-se em [Santos e Proença 1993].

4.2 Metodologia de desenvolvimento

Seguindo uma metodologia de desenvolvimento estruturada, assente numa filosofia *top-down*, após a partição inicial em blocos cuja complexidade é mais facilmente tratável, é necessário definir a funcionalidade de cada um desses blocos e verificar a sua adequação à respectiva especificação. A verificação será feita por simulação, utilizando modelos construídos em VHDL, no domínio comportamental e a um nível de abstracção tão elevado quanto o possível. A síntese depende obviamente da tecnologia de implementação seleccionada. Nos casos em que for necessário desenvolver *hardware* específico,

procurar-se-à utilizar ferramentas de síntese automática, sobre versões refinadas das descrições abstractas, e tendo o cuidado de verificar (por simulação) todas essas operações de refinamento.

Os níveis *Architecture* ou *Register*, no domínio comportamental, são os níveis usualmente adoptados para criar um modelo funcional de algum módulo complexo, como, por exemplo, um processador ou até um pequeno sistema com um processador e memória [Ashenden 1990], [Armstrong and Gray 1993] e [Ecker and März 1993]. O nível e o domínio de representação escolhidos estão de acordo com as características a avaliar através da simulação, balanceando a abstracção e correspondente detalhe das descrições, com o desempenho do próprio simulador.

Na medida do possível, os diversos modelos desenvolvidos nesta fase deverão ser parametrizados possibilitando a sua adaptação a novas características e a sua reutilização noutros projectos.

Nesta fase, e relativamente a cada um dos blocos constituintes, é possível identificar três tipos de soluções correspondendo a outros tantos métodos de desenvolvimento distintos:

- geração de um ou mais componentes específicos;
- utilização de um componente *standard* (seguindo uma filosofia de desenvolvimento *bottom-up*); e
- utilização de um sistema programável por *software* (tipicamente um processador, memória e componentes de entrada/saída).

A escolha de uma dessas soluções depende essencialmente do respectivo impacto na rapidez do desenvolvimento do protótipo, das características temporais da solução e, em particular para o caso dos componentes *standard*, depende ainda da sua disponibilidade no mercado. As soluções de *software* devem ser privilegiadas uma vez que não implicam o desenvolvimento e teste de *hardware*, mas evidenciam limitações temporais importantes. Quanto às soluções que envolvem *hardware*, com base no conjunto de critérios de selecção e a análise feita no capítulo 3 sobre as tecnologias de implementação, devem ser privilegiadas as soluções com componentes *standard* e/ou lógica programável.

No que respeita à construção do modelo em VHDL, é necessário considerar dois tipos de descrições. Uma descrição estrutural que permita simular o sistema completo, e um conjunto de descrições comportamentais associadas a cada um dos blocos constituintes. Neste último tipo de descrição, as três soluções de implementação identificadas anteriormente podem ser associadas a outros tantos métodos de desenvolvimento:

- **Modelo sintetizável**. Corresponde a uma descrição comportamental, que deverá ser elaborada tendo em atenção as restrições impostas pela ferramenta de síntese. É admissível fazer uma descrição a um nível de abstracção superior ao que é suportado pela ferramenta de síntese, desde que as alterações posteriores não sejam complexas. Não é tomada qualquer decisão quanto à síntese, pelo que as respectivas consequências no desempenho não são consideradas. Após a síntese, é fácil incluir aquelas considerações no mesmo modelo.

- **Modelo de um componente *standard***. Neste caso, o componente já existe e a sua função está completamente definida, bem como as suas características temporais que permitirão avaliar o desempenho. Eventualmente, poderá já existir um modelo que, obviamente, será utilizado. Mas se não existir, a sua construção será baseada na especificação fornecida pelo fabricante, normalmente constituída por um diagrama de blocos e uma descrição textual detalhada. Ao nível de abstracção considerado, é difícil introduzir os tempos de atraso através de um único processo. Contudo, esta dificuldade pode ser ultrapassada utilizando vários processos concorrentes, cada um responsável por um dos blocos constituintes e interligados por sinais [Coelho 1992]. Este tipo de descrição permite já avaliar o desempenho do componente, no contexto do sistema. De realçar ainda que, por vezes, não é utilizada toda a funcionalidade de um componente e que, nesse caso, o modelo poderá ser reduzido.
- **Modelo de *software***. Neste caso, não interessa verificar o *hardware* subjacente, pois tipicamente corresponde a um módulo já construído, ou seja, um sistema cujo desenvolvimento já foi alvo de verificação adequada (em princípio). Do ponto de vista do modelo a construir é mais importante assegurar a correcção dos algoritmos a desenvolver, passando assim a verificação para o domínio do desenvolvimento de *software*. Contudo, para poder simular o sistema completo, é necessário descrever em VHDL este tipo de bloco, para o que basta um ou mais processos VHDL que descrevam algoritmicamente o funcionamento do *software* e o seu interface com o resto do sistema. Atendendo às suas características, o desempenho de um bloco deste tipo é muito difícil de avaliar.

O projecto do controlador do GLiTCH inclui exemplos de cada um destes tipos de blocos, constituindo assim um bom exemplo da metodologia apresentada. As secções seguintes descrevem, em maior detalhe, os procedimentos a adoptar para cada um daqueles tipos de blocos.

Na construção de um modelo VHDL, no domínio comportamental e ao nível *Architecture* ou *Register*, é possível sistematizar algumas tarefas. Se o bloco a desenvolver corresponder a uma unidade programável ou a uma máquina de estados simples, o respectivo conjunto de instruções ou conjunto de estados pode ser modelado em VHDL definindo um tipo associado a um conjunto enumerado. O passo seguinte consiste na identificação do interface externo do bloco, na caracterização dos respectivos sinais e na identificação de eventuais parâmetros para generalizar a descrição. No conjunto, esta informação é descrita através da construtor designado por **entity**. Finalmente, descreve-se a funcionalidade através de um outro construtor, designado por **architecture**, utilizando um ou mais processos e explorando a capacidade (limitada) de descrição hierárquica conferida pelos subprogramas.

No que respeita à descrição estrutural do sistema, algumas simplificações podem ser introduzidas, nomeadamente no que se refere aos barramentos. Em VHDL, sempre que um barramento inclui sinais que interligam duas ou mais saídas de componentes, os tipos de dados usados têm que dispor de uma representação para o necessário estado de alta impedância e ter ainda uma função de resolução associada. Este facto dificulta, de alguma

forma, a possibilidade de utilizar tipos de dados abstractos. Dependendo exactamente do que se pretende simular, esta limitação pode ser ultrapassada de duas formas:

- Utilizar, nas descrições internas, tipos de dados abstractos e, externamente, tipos de dados com as características enunciadas, implementado as necessárias funções de conversão; esta solução permite a construção de um modelo exacto, mas penaliza o tempo de execução da simulação e o grau de complexidade; uma variante desta solução é tratar as ligações aos barramentos como processos independentes.
- Limitar a conectividade, eventualmente duplicando interligações entre componentes, por forma a garantir que cada sinal de um barramento não interligue saídas de componentes; esta solução afasta o modelo do circuito real, mas permite um maior nível de abstracção e melhor desempenho na simulação.

No desenvolvimento do controlador do GLiTCH privilegiou-se esta última opção, limitando a simulação ao que se pode designar por funcionamento normal, isto é, ao conjunto de operações exigidas para implementar a função de controlador do GLiTCH. Neste contexto, são consideradas operações de excepção aquelas que envolvem o acesso partilhado do transputer a alguns componentes, para teste e inicialização. As implicações desta opção serão posteriormente referidos em maior detalhe, na descrição do processo de desenvolvimento dos modelos em VHDL.

4.2.1 Testabilidade no controlador do GLiTCH

Os factores mais importantes que condicionaram as opções a tomar no projecto do controlador do GLiTCH foram o tempo de desenvolvimento, a flexibilidade da solução e a testabilidade. Atendendo a este conjunto de critérios, e com base na análise feita sobre as tecnologias de implementação, a prioridade na selecção de componentes foi assim dada à lógica reprogramável e aos componentes *standard*. Este tipo de componentes ainda não suporta, na sua maioria, técnicas de teste baseadas em *Boundary Scan* (BS). Por outro lado, a ferramenta de desenvolvimento utilizada para o desenho do sistema também não inclui qualquer tipo de facilidade nesse sentido. Estes factores impediram a implementação das referidas técnicas de teste.

O controlador do GLiTCH, à semelhança da maioria dos sistemas digitais, impõe dois tipos de problemas de testabilidade: testes ao nível dos componentes a desenvolver e testes ao nível do sistema. A procura de soluções para o primeiro destes problemas esteve na base da selecção do tipo de lógica programável a utilizar (PLD/CPLD ou FPGA). Com efeito, considerando os sistemas de desenvolvimento disponíveis, os programadores de PLD/CPLD permitem que os vectores de teste usados na simulação com o modelo funcional, sejam igualmente usados com o circuito final programado, o que facilitará substancialmente a tarefa de teste. O problema será então a geração de um conjunto de vectores de teste satisfatório, o que não será difícil atendendo ao nível de complexidade dos módulos descritos na especificação inicial (figura 4.1) e cuja implementação em *hardware* específico pode ser considerada. De notar que os FPGA, por serem programados no próprio circuito, não evidenciam esta propriedade, a não ser que se inclua no circuito o *hardware* de teste equivalente ao existente nos referidos programadores!

O baixo nível de complexidade necessário para viabilizar a metodologia de teste escolhida pode levar a concluir que se está a comprometer uma possível integração de diversas funções num mesmo componente. Na realidade, tal não se verifica porque é possível desenvolver módulos simples separadamente e fazer a junção das descrições e dos respectivos vectores de teste na fase final da síntese.

4.2.2 O transputer na função de teste

O teste ao nível do sistema é feito através do processador escalar disponível, o transputer, cujo desenvolvimento incluíu um conjunto de características que visam exactamente facilitar a função de teste. Do ponto de vista da arquitectura do sistema, é necessário garantir apenas que a estrutura de barramentos dê acesso aos componentes que se pretendem testar. Esta exigência não implica necessariamente o aumento de complexidade da placa de circuito impresso, uma vez que a acessibilidade tem que ser garantida em funcionamento normal, na maioria dos casos.

No contexto das funções de teste e diagnóstico, as características mais relevantes do transputer são as seguintes [Hinton e Pinder 1993] e [INMOS 1992]:

- O transputer é um sistema completo (incluindo alguma memória interna) que pode executar um conjunto de tarefas mínimo e relevantes para o teste e diagnóstico, desde que seja ligado a uma fonte de tensão apropriada e disponha de um sinal de *clock* estável e um sinal de inicialização (*reset*); isto significa que, mesmo que os componentes periféricos estejam defeituosos, é sempre possível utilizar o processador, dentro das limitações da arquitectura do sistema.
- A comunicação do transputer com o exterior (relativamente ao sistema que integra o transputer) é feita, preferencialmente, através de *links*. Um *link* é uma comunicação série bidireccional, realizada apenas por três condutores e com grande fiabilidade; tipicamente, um dos *links* é utilizado para ligar o transputer a um sistema de desenvolvimento, eventualmente baseado também num transputer, onde pode executar um programa de monitorização e interface com o utilizador.
- Se o transputer está configurado para "arrancar" a partir de um *link* (situação mais vulgar), após a inicialização ele testa ciclicamente todos os *links* até detectar uma comunicação por um deles; através deste mecanismo simples, o sistema de desenvolvimento envia ao transputer um programa, ou então uma sequência de instruções especiais designadas por *peek* e *poke*; estas instruções permitem aceder a todo o espaço de memória do transputer, o que significa que permitem aceder também a todos os registos associados aos componentes periféricos, atendendo a que os respectivos endereços são obrigatoriamente mapeados no espaço de memória.

É importante realçar que a filosofia de comunicações do transputer em conjunto com a utilização das instruções *peek* e *poke*, dispensam, de certa forma, a utilização de um dispendioso e complexo emulador (que de qualquer forma não existe para o transputer).

Explorando as características enunciadas, o teste funcional do controlador do GLiTCH, ao nível do sistema, será então executado através da seguinte sequência de passos:

1. Apenas com o transputer e os componentes de suporte inseridos na placa, verificar a comunicação com o sistema de desenvolvimento, utilizando instruções *peek* e *poke* sobre a memória interna.
2. Após adicionar a memória externa do transputer, verificar o acesso a essa memória, novamente através de instruções *peek* e *poke*.
3. Enviar para o transputer um programa de teste da memória externa, programa esse que, preferencialmente, deverá executar a partir da memória interna do transputer.
4. Inserir a micromemória (ver figura 4.1) e enviar para o transputer um programa de teste a essa memória, semelhante ao anterior.
5. Inserir a memória associada à unidade de memória de padrões e repetir o programa usado em 4, para esta memória.
6. Inserir o sequenciador e enviar para o transputer um programa que carregue na micromemória um microprograma adequado para testar a operação do sequenciador.
7. Inserir os componentes associados às restantes unidades funcionais e repetir o processo anterior para cada uma dessas unidades.

Esta sequência de testes não permite verificar completamente o sistema, e apenas demonstra que, para os programas desenvolvidos, não se verificam erros. No entanto, convém não esquecer que os testes individuais ao nível do componente podem ser suficientemente elaborados para garantir um grau de confiança elevado. Os componentes *standard* utilizados foram (pressupostamente) testados pelo fabricante, evidenciando igualmente um grau de confiança elevado. Assim, a preocupação central dos testes ao nível do sistema é verificar as interligações dos componentes e não tanto a sua funcionalidade. Este facto deve estar subjacente à elaboração dos programas e microprogramas necessários na sequência de testes anteriormente descritos, e reduz significativamente a sua complexidade.

4.3 Unidade de memória de padrões

De acordo com a especificação inicial, a unidade de memória de padrões (em seguida designada abreviadamente por DS) é usada para armazenar dados e resultados de um microprograma. Na sua maior parte, os dados a armazenar destinam-se a ser utilizados pelo *array* de GLiTCHs em operações associativas. Neste tipo de operação, cada dígito (frequentemente designado por dígito ternário) elementar pode assumir um de três valores: ‘0’, ‘1’, ou ‘X’, representando este último um valor *don’t care*, isto é, qualquer célula de memória associativa satisfaz a condição de verificação com o dígito ‘X’, independentemente do seu conteúdo. Este mecanismo foi já descrito na análise detalhada da CAM, e é usualmente implementado através de dois dígitos binários, assumindo um deles a função de máscara - sempre que este bit tiver o valor ‘1’ a combinação dos dois codifica o valor ‘X’ do respectivo dígito ternário. No caso do GLiTCH, cada dado é composto por 16 dígitos ternários.

Desta forma, e tendo em conta ainda o reduzido tempo de resposta exigido, o DS é constituído fundamentalmente por 2 bancos de memória estática, o *Data Bank* (DB) e o *Mask Bank* (MB) - com palavras de 16 bits cada. Balanceando o custo, a disponibilidade e

ainda os requisitos dos microprogramas já desenvolvidos para o GLiTHC, foi possível determinar o tamanho desta memória - 8Kx32bits.

No modo de funcionamento normal, os endereços e os sinais de controlo para o DS provêm de uma microinstrução. Atendendo à máxima frequência a que se pretende operar, e ao tempo de atraso introduzido pelo sequenciador e pela micromemória, desde cedo foi perceptível a necessidade de incluir à entrada do DS um registo (nível de *pipeline*) para os endereços e sinais de controlo. Para além dessas funções básicas, e caso se utilize o mecanismo de chamada de subrotinas do sequenciador, será necessário ainda dispor de algum tipo de suporte para passagem de parâmetros. Com esse objectivo, a função do registo referido foi enriquecida com as facilidades adicionais de poder incrementar e decrementar o seu conteúdo, isto é, o endereço para o DS. Para além da utilização na passagem de parâmetros, aquela funcionalidade ainda irá facilitar a tarefa de geração de código de um compilador, no que diz respeito ao acesso a vectores ou estruturas de dados equivalentes. Este bloco de lógica auxiliar para a geração de endereços destinados ao DS é designado abreviadamente por AGL (*Address Generation Logic*) e, do ponto de vista funcional, deverá executar as instruções indicadas na tabela 4.1.

É importante realçar que, logicamente, o AGL podia ser incluído no sequenciador, com vantagens óbvias, sendo esta uma hipótese a considerar num futuro melhoramento do sistema e caso se desenvolva um sequenciador específico.

I1	I0	Mnem.	Instrução
Ø	Ø	HOLD	Preserva endereço anterior
1	Ø	INC	Incrementa endereço anterior
Ø	1	DEC	Decrementa endereço anterior
1	1	LOAD	Carrega novo endereço

EN_B	R/W	Mnem.	Controlo (sinais de controlo da memória)
0	x	DIS	Desactiva ambos os Bancos
1	1	RD	Selecciona ambos os bancos para leitura
1	0	WR	Selecciona DB e desactiva MB, para escrita

Nota:

x *don't care*

Tabela 4.1 Instruções da unidade de memória de padrões

No que respeita à integração do DS no sistema, existem ainda dois problemas a resolver, exclusivos da implementação em *hardware* e que por conseguinte não fazem parte da especificação inicial: o acesso do transputer ao DS (para testes, para introduzir os dados necessários a um microprograma, e ainda para ler resultados); e o controlo do sinal de escrita na memória. O primeiro problema foi resolvido utilizando um sinal especial, designado por **BusTrnotLoc1**, que servirá para forçar um estado de alta impedância nas saídas do AGL para o DS, ao mesmo tempo que mantém o estado interno do AGL inalterado. Enquanto esse sinal estiver activo, o transputer poderá aceder livremente ao DS. O sinal **BusTrnotLoc1** será controlado por um arbitrador de barramentos, o qual será descrito posteriormente.

O sinal de escrita na memória é gerado em conjunto com o *clock* do sistema, o qual, por imposição do próprio GLiTCH e respectivos componentes de suporte, é composto por 8 fases iguais [Thomson 1994]. Nesse esquema, o sinal de escrita para o DS corresponde a

um impulso (negativo) activo desde a fase 3 à fase 7, ou seja, com a duração de 31,25 ns considerando a máxima frequência do *clock*. Esta temporização dá uma margem alargada para a implementação do AGL e assegura o funcionamento para uma largo espectro de memórias estáticas, flexibilizando a respectiva selecção no que respeita ao custo e potência consumida.

Por último, é necessário ainda referir a inclusão no AGL de um terceiro sinal de controlo proveniente da microinstrução, o qual não está directamente ligado à operação do DS, mas sim a uma forma de optimização da micromemória. Esse sinal tem por função forçar o AGL a manter o seu estado interno (ignorando os restantes sinais de instrução e controlo) e foi introduzido para permitir que os campos da micromemória destinados ao DS pudessem ser partilhados com outra unidade. Este mecanismo apresenta o inconveniente de dificultar a tarefa da geração de código automática, mas permite diminuir o tamanho da palavra da micromemória. A figura 4.2 mostra o diagrama de blocos do DS, considerando a estrutura apresentada e todos as facilidades adicionais introduzidas.

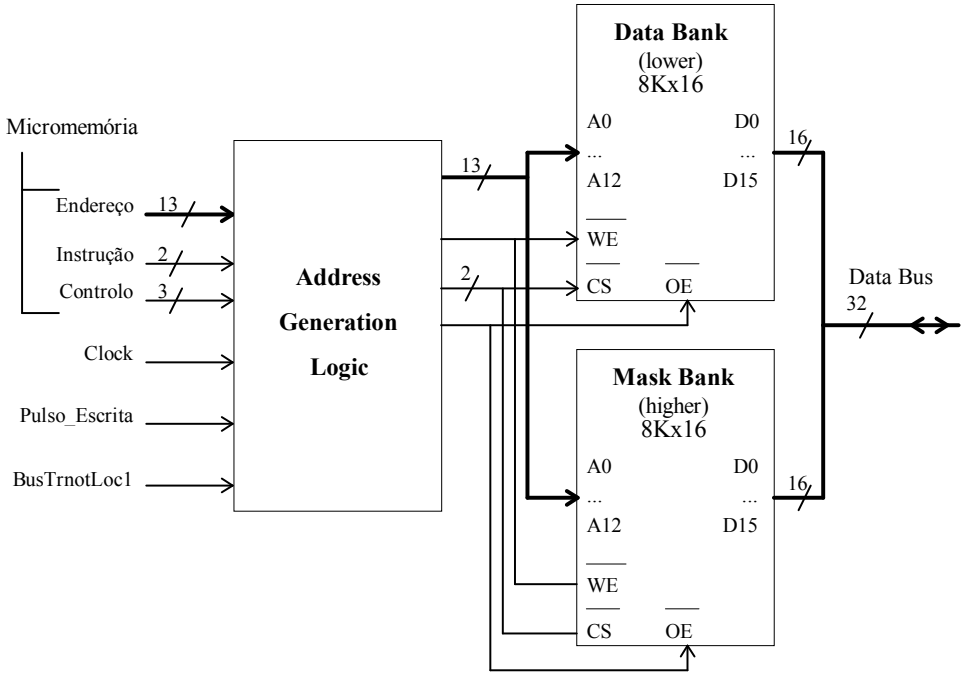


Figura 4.2 Diagrama de blocos da unidade de memória de padrões

4.3.1 Modelo VHDL da unidade de memória de padrões

O desenvolvimento do modelo VHDL para o DS é condicionado, à partida, pelo facto de se pretender sintetizar o bloco AGL e a memória poder ser controlada a partir de um microprograma ou do transputer. Estes pressupostos ditam a construção de duas descrições separadas para cada um dos blocos AGL e memória, a um nível de detalhe mais baixo (*Register*), onde se explicitará a estrutura e a forma como será implementado o mecanismo de partilha da memória. Sem considerar aquelas restrições seria possível conceber um único modelo em que, por exemplo, a memória seria implementada através de uma variável interna do processo associado ao AGL, tendo este capacidade para servir

pedidos de acesso a partir de um microprograma e do transputer. Em seguida, serão analisados os aspectos mais relevantes da construção do modelo do AGL e da memória.

AGL - Descrição do conjunto de instruções

O conjunto de instruções executadas pelo DS (ver tabela 4.1) pode ser modelado através de um conjunto enumerado e um *record* VHDL, do seguinte modo (ver apêndice A - **package Gl_pack.vhdl**):

```
-- Instrucoes para o Address Generation Logic:
-- o campo "instr" comanda o registo de endereco;
-- o campo "ctr" comando os sinais de controlo do DS;
-- o campo "data" e' o operando da instrucao LOAD.
  type agl_instr_type is (HOLD, DEC, INC, LOAD);
  type agl_ctr_type is (DIS, WR, RD);
  type agl_instr is record
    instr: agl_instr_type;
    ctr: agl_ctr_type;
    data: ds_end; -- subtype de integer
  end record;
```

AGL - Descrição do interface externo

Com base no diagrama da figura 4.2, o interface externo é descrito através do construtor **entity**, do seguinte modo:

```
entity agl_c is
port (Instruction: in agl_instr; -- "record"
      clock, write_pulse, BusTrnotLoc1,
      ForceHold: in bit;
      Addr_out: out ds_end;
      notCS: out std_logic_vector(1 downto 0);
      notOE, notWE: out std_logic);
end agl_c;
```

À semelhança da opção tomada para o modelo VHDL do sequenciador, também no AGL a modelação exacta foi preterida pela funcionalidade. A especificação para esta unidade exige que o acesso à memória seja partilhado com o transputer, pelo que as saídas para a memória deveriam ser de um tipo adequado a essa função. O tipo `ds_end` é um subtipo do conjunto dos números inteiros limitado à gama de endereços da memória (ver **package Gl_pack.vhdl** no apêndice A) e não contém uma representação explícita para o estado de alta impedância nem uma função de resolução associada. Tal como foi referido na descrição da metodologia de desenvolvimento, esta opção facilita a descrição interna do AGL, mas vai impor restrições ao modelo estrutural do controlador do GLiTCH.

De notar que os tipos `std_logic` e `std_logic_vector` têm já uma função de resolução associada e uma representação para o estado de alta impedância, pelos que as saídas de controlo para a memória podem ser modeladas com mais exactidão. Face à opção tomada para o endereço de saída, esta utilização não introduz qualquer vantagem ou desvantagem, quer em termos funcionais quer em termos estruturais, podendo apenas considerar-se como um passo necessário para a síntese do AGL.

AGL - Descrição da funcionalidade

A arquitectura do AGL é descrita por um único processo VHDL que inclui o registo de endereço (variável interna) e os registos para os sinais de controlo. A listagem seguinte

mostra o fragmento da descrição da arquitectura do AGL que descreve o função do registo de endereço (a listagem completa encontra-se no apêndice A):

```
architecture behavior of agl_c is
  -- Constantes locais
  ...
  begin
    ctr_unit: process(Clock, BusTrnotLoc1)
      -- Variaveis internas auxiliares.
      variable reg_add: ds_end;
      begin
        if ((Clock'event and (Clock = '1')) and
            (BusTrnotLoc1 = '0') and (ForceHold = '0')) then
          case Instruction.instr is
             when HOLD => NULL;
             when DEC  => reg_add := reg_add - 1;
                          Addr_out <= reg_add;
             when INC  => reg_add := reg_add + 1;
                          Addr_out <= reg_add;
             when LOAD => reg_add := Instruction.data;
                          Addr_out <= reg_add;
          end case;
          ...
         elsif (BusTrnotLoc1 = '1') then
           notWE <= 'Z'; notOE <= 'Z'; notCS <= ('Z', 'Z');
        end if;
      end process;
  end behavior;
```

Tal como foi referido, o sinal **BusTrnotLoc1** é utilizado para controlar o acesso partilhado à memória. De acordo com as opções tomadas, quando este sinal está activo os sinais de controlo para a memória ficam no estado de alta impedância, enquanto que o endereço de saída permanece inalterado.

De notar ainda que a lista de sensibilidade do processo apenas inclui os sinais `clock` e `BusTrnotLoc1`, o que se justifica porque todos os outros sinais de entrada são sincronizados com o `clock`, enquanto que o sinal `BusTrnotLoc1` desencadeia a respectiva acção de uma forma imediata.

Memória - Descrição do interface externo

Relativamente à memória e seguindo a mesma metodologia, o interface com o exterior pode ser descrito da seguinte forma (ver também figura 4.2):

```
entity ds is
   port (endereco: in ds_end;
         dado: inout std_logic_vector(31 downto 0);
         notrd, notwr: in std_logic;
         notcs: in std_logic_vector(1 downto 0) );
                          -- notcs(1) -> Mask Bank
end ds;                   -- notcs(0) -> Data Bank
```

Memória - Descrição da funcionalidade

Este modelo será apenas utilizado na simulação, pelo que se justifica uma descrição a um nível de abstracção mais elevado. Para além disso, atendendo a que se trata de uma estrutura comum, deve procurar-se construir um modelo genérico que facilite a sua reutilização. O primeiro objectivo está, naturalmente, limitado pelo tipo de informação que se pretende armazenar. Neste caso, trata-se de padrões binários e a consideração de

qualquer outro tipo de dado iria aumentar, desnecessariamente, a complexidade da descrição. Quanto ao segundo objectivo, ele é de certa forma sacrificado pelo facto desta memória estar logicamente dividida em dois bancos.

No sistema real, a inicialização da memória será feita através do transputer. No modelo VHDL, essa inicialização poderá ser feita de vários modos: um processo associado à função lógica do transputer (modelo de uma unidade de *software*), a utilização de constantes explícitas no modelo da memória, ou ainda a partir de um ficheiro que será lido pelo processo que descreve a memória, numa fase de inicialização, como se o conteúdo da memória fosse constituído por vectores de teste. No modelo desenvolvido, tanto pode ser utilizado o primeiro como o terceiro método. Na perspectiva da memória, a funcionalidade exigida para o primeiro método está implícita na operação normal do sistema (acesso à memória a partir do transputer). Para implementar o terceiro método foi introduzida uma fase de inicialização dos processos que descrevem a memória e onde pode ser lido um ficheiro - face às restrições anteriormente referidas, no que respeita ao acesso partilhado, este será o método preferencial.

A descrição da arquitectura utiliza então três processos. Um para cada um dos bancos de memória e outro para controlar as linhas de dados. A listagem seguinte mostra um fragmento da descrição da arquitectura da memória do DS:

```
architecture behavior of ds is
  subtype mem_cell is bit_vector (15 downto 0);
  type mem_bloco is array (ds_end'low to
                           ds_end'high) of mem_cell;
  signal mask_in, mask_out, data_in,
         data_out: mem_cell;
  signal MaskRdnotWr, DataRdnotWr: bit;
  ...  Procedimentos de leitura e escrita na memoria
  begin
    Mask:process (endereco, mask_in, MaskRdnotWr)
      variable mem_mask: mem_bloco;
      variable inicio: boolean := TRUE;
      variable pont: ds_end;
      begin
        if inicio then
           -- Bloco de inicializacao da memoria (Mask Bank)
              ...
           inicio := FALSE;
          else
            pont := endereco;
            if (MaskRdnotWr = '1') then       -- Cond. de leitura
               le_ds(mem_mask, pont, mask_out);
              else                             -- Cond. escrita
                 wr_ds(mem_mask, pont, mask_in);
            end if;
        end if;
      end process Mask;
    ...  Data:process (endereco, data_in, DataRdnotWr)
    ...  process (dado, notrd, notwr, notcs, mask_out, data_out)
  end behavior;
```

Na perspectiva dos objectivos gerais definidos para o desenvolvimento de modelos em VHDL, os aspectos mais relevantes desta descrição são os seguintes:

- independência da descrição relativamente ao tamanho da memória, a qual é inferida a partir da gama definida para o tipo `ds_end` no **package**

Gl_pack.vhdl (ver apêndice A), utilizando os atributos `ds_end'low` e `ds_end'high`.

- A dependência relativa ao tipo de dado armazenado está confinada à declaração do tipo `mem_cell` e aos procedimentos de leitura e escrita; a inclusão de dois bancos é um caso particular da geração de duas instâncias de uma mesma estrutura.
- O processo relacionado com os sinais de controlo e o barramento de dados exterior já é mais específico para implementar uma particular noção de bancos de memória, mas poderá ser facilmente alterado para outros tipos de ligações entre a memória e o barramento de dados.

Implementação do AGL

O AGL foi implementado em dois componentes do tipo EPLD - uma PAL26V12 e uma PAL22V10. Esta escolha está relacionada apenas com limitações das ferramentas de síntese utilizadas e podem ser consideradas circunstanciais. Tendo em conta a metodologia de teste escolhida, para poder utilizar o mesmo conjunto de vectores de teste na simulação e na programação, foi necessário reescrever a descrição do AGL na linguagem PALASM (a descrição completa do AGL e dos respectivos vectores de teste encontra-se no apêndice D). Atendendo ao nível de representação utilizado, as alterações a introduzir resumiram-se a:

- modificar a representação do endereço de saída do tipo inteiro para o tipo binário;
- atribuir códigos binários às instruções;
- descrever as operações incrementar e decrementar através de equações booleanas;
- adaptar os vectores de teste utilizados na simulação do modelo VHDL às novas representações, por forma a poder utilizar o mesmo algoritmo de teste nas duas descrições; e
- repartir as funções pelos dois componentes utilizados.

As duas primeiras alterações são consideradas normais num processo de síntese. As três últimas seriam desnecessárias com ferramentas mais evoluídas, e em particular a última resultou de uma mera limitação tecnológica. A descrição das operações incrementar e decrementar, através de equações booleanas, introduziu uma complexidade adicional que exigiu maior cuidado na verificação. Nas versões mais actualizadas das ferramentas de síntese (nomeadamente as que utilizam a linguagem DSL[2], sucessora da linguagem PALASM), este tipo de operação é automaticamente implementado a partir de um nível de abstracção mais elevado. No capítulo seguinte serão estudadas implementações alternativas e o impacto que o tipo de descrição escolhido tem nessas implementações.

4.4 Unidade escalar

A unidade escalar (em seguida abreviadamente referida por SU) é basicamente um registo de deslocamento de 16 bits, carregado em paralelo através do barramento de dados

[2] *Design Synthesis Language*

(ver figura 4.1), cuja função é executar algumas operações para as quais a arquitectura vectorial e associativa do GLiTCH não está vocacionada. De uma forma mais detalhada e de acordo com a experiência de implementação de vários algoritmos no simulador, que ditaram a necessidade da SU, o conjunto de operações que esta deve executar é o seguinte:

- deslocamento aritmético para a direita;
- deslocamento lógico para a esquerda e direita;
- manter permanentemente, como *flags*, os três bits menos significativos, o bit mais significativo e a indicação se o valor do registo é zero; e
- ligação ao DB e MB do DS (ver secção 4.3), através do barramento de dados de 32 bits (figura 4.1).

De acordo com a especificação apresentada, a SU deve executar o conjunto de instruções da tabela 4.2.

EnR/W	I1	I0	Mnem.	Instrução
Ø	Ø	Ø	HOLD	Preserva conteúdo do registo
Ø	Ø	1	ASR	Deslocamento aritmético do registo, à direita (1 bit)
Ø	1	Ø	LSL	Deslocamento lógico do registo, à esquerda (1 bit)
Ø	1	1	LSR	Deslocamento lógico do registo, à direita (1 bit)
1	X	Ø	LDR	Carrega registo
1	X	1	OSR	Envia para a saída o conteúdo do registo

Notas:
X *don't care*

Tabela 4.2 Instruções da unidade escalar

A ligação lógica entre a SU (registo de 16 bits) e o barramento de dados (32 bits) é peculiar e necessita de uma explicação mais detalhada. Em operações de escrita no registo interno da SU (instrução LDR), os 16 bits mais significativos, correspondentes ao MB, devem ser usados apenas como entradas e o seu valor servirá como máscara relativamente ao valor existente nos 16 bits menos significativos do barramento. A operação lógica **AND** será aplicada entre os dois valores de 16 bits para determinar o valor a carregar no registo interno. Em operações de leitura, o conteúdo do registo interno da SU deve ser colocado nos 16 bits menos significativos do barramento de dados, correspondentes ao DB.

Numa análise prévia sobre a temporização da SU, foi possível identificar duas situações críticas. Em primeiro lugar, um operando proveniente da SU para o GLiTCH tem que ser colocado no barramento de dados praticamente no início de um ciclo de *clock*. Em segundo lugar, é bastante elevado o tempo de *setup* imposto pelo sequenciador às *flags* da SU. Atendendo a que num ciclo de *clock* é necessário ainda contar com o tempo de acesso à micromemória, por forma a garantir as exigências anteriores, foi decidido incluir na SU um registo de instrução (nível de *pipeline*). Desta forma, o efeito de uma instrução para a SU apenas se vai fazer sentir no início do próximo ciclo de *clock*.

A figura 4.3 apresenta o diagrama lógico da SU, onde são identificadas as suas relações com os restantes componentes do controlador do GLiTCH.

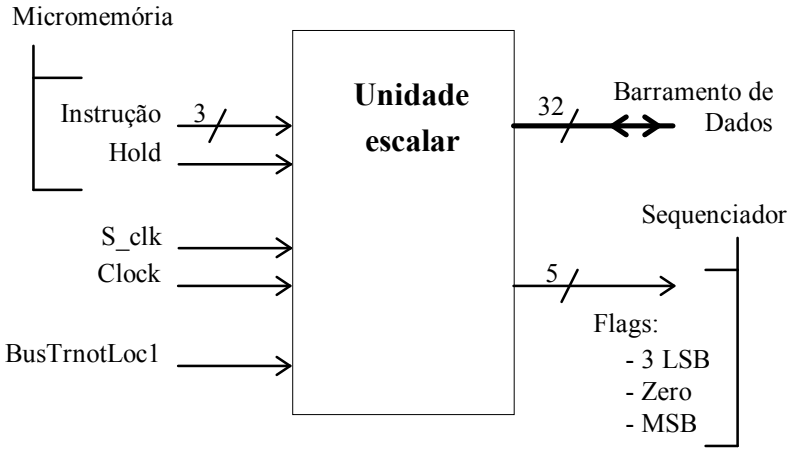


Figura 4.3 Diagrama lógico da Unidade escalar

Na especificação da SU foi decidido introduzir a mesma estratégia que foi utilizada no DS para optimização da micromemória. O sinal designado por **Hold** na figura 4.3 tem por função forçar a SU a manter o seu estado interno (ignorando a instrução) por forma a permitir que os campos da micromemória destinados à SU possam ser partilhados com outra unidade.

O sinal **BusTrnotLoc1** também foi utilizado no DS e tem por função obrigar a que as saídas da SU fiquem no estado de alta impedância, de modo a que o transputer possa aceder à memória do DS, através do barramento de dados partilhado. Enquanto aquele sinal estiver activo, o estado interno da SU deve permanecer inalterado, o que será garantido, por exemplo, realimentando os registos internos com os seus valores de saída. O consequente aumento de complexidade não é crítico para o registo de instrução, atendendo à sua reduzida dimensão, mas o mesmo não se verifica para o registo de dados. Devido a restrições dos componentes de suporte do GLiTCH, foi necessário implementar um segundo sinal de *clock*, exactamente com a mesma temporização, mas que permanece inactivo enquanto o sinal **BusTrnotLoc1** estiver activo [Thomson 1994]. Este segundo sinal de *clock* (designado por **S_clk** na figura 4.3) será usado no registo de dados da SU, evitando assim a inclusão da lógica de realimentação atrás referida.

4.4.1 Modelo VHDL da unidade escalar

A SU vai executar um conjunto de operações binárias, sobre operandos binários, de acordo com a sua especificação. Nesta situação não faz sentido procurar elevar o nível de abstracção, o que corresponderia a complicar uma descrição que por natureza é simples. Por outro lado, a SU será sintetizada em *hardware* específico (lógica programável), pelo que a descrição VHDL deve facilitar a tarefa de refinamento, incluindo inicialmente alguns detalhes, como por exemplo, a existência do registo interno de instrução e os dois sinais de *clock* anteriormente referidos. Este factos constituem os condicionantes que devem nortear o desenvolvimento do modelo VHDL.

Descrição do conjunto de instruções

O conjunto de instruções executadas pela SU (tabela 4.2) pode ser modelado, em VHDL, pelo seguinte conjunto enumerado (ver apêndice A - **package Gl_pack.vhdl**):

```
type su_instr is (HLD, ASR, LSL, LSR, LDR, OSR);
```

Descrição do interface externo

Com base no diagrama da figura 4.3, o interface externo é descrito do seguinte modo:

```
entity su_c is
port (signal io_bus: inout std_logic_vector(31 downto 0);
      signal Instruction: in su_instr;
      signal Hold, clk, s_clk, BusTrnotLoc1: in bit;
      signal LSB_3: out bit_vector(2 downto 0);
      signal Zero, MSB: out bit);
end su_c;
```

De notar a utilização do tipo `std_logic_vector` nos sinais de ligação da SU ao barramento de dados. Atendendo a que não é desejável abstrair mais a representação de dados, a utilização daquele tipo promove a modelação exacta da operação do barramento.

Descrição da funcionalidade

No que respeita à descrição funcional da arquitectura da SU, a um nível de abstracção mais elevado poder-se-ia ignorar, sem problemas, a existência do registo de instrução e utilizar-se apenas um processo. No entanto, atendendo aos condicionantes deste modelo, é preferível utilizar um estilo de descrição mais orientada à síntese. A arquitectura é então descrita por dois processos, cada um sensível a um dos sinais de *clock*. A listagem seguinte mostra um fragmento dessa descrição (a descrição completa encontra-se no apêndice A):

```
architecture behavior of su_c is
...
signal instr_aux: su_instr;          -- registo de instrucao
signal loc_reg: bit_vector(15 downto 0); -- registo de dados
  begin
    ...  reg_instruction: process(BusTrnotLoc1, clk, Hold)
    ...  reg_deslocamento: process(s_clk)
    MSB <= loc_reg(15);
    LSB_3 <= loc_reg(2 downto 0);
    Zero <= '1' when allzero(loc_reg) else '0';
  end behavior;
```

O registo de dados da SU foi modelado com lógica de dois estados, utilizando o tipo pré-definido `bit` (2 estados). No entanto, o barramento externo é do tipo `std_logic_vector` (4 estados). Para estabelecer o interface entre os dois tipos lógicos utilizados foram criadas duas funções de transformação. Apenas a função que transforma um valor do segundo tipo num do primeiro tipo apresenta dificuldades, pois é necessário associar aos valores 'X' e 'Z' um dos valores do tipo `bit`. Na função em questão, cuja listagem está no **package** local da SU (apêndice A), optou-se por associar a ambos o valor '0', mas tendo o cuidado de testar inicialmente a ocorrência de um daqueles valores, uma vez que tal significa uma situação de erro numa simulação (na prática significaria tentar carregar o registo interno da SU com um 'X' ou um 'Z').

Todas as operações sobre o registo de deslocamento estão implementadas sobre a forma de funções ou procedimentos e são genéricas, uma vez que inferem a dimensão dos respectivos operandos (vectores) através de atributos. A respectiva descrição encontra-se no **package** auxiliar do modelo da SU (apêndice A). Em VHDL é possível modelar

operações de deslocamento apenas através de atribuições, no entanto esse método corresponde a uma descrição a um nível de abstracção mais baixo (*data-flow*) e apenas pode ser utilizado no contexto do modelo em questão.

Ao construir um algoritmo para uma operação de deslocamento genérica, sobre um tipo vector, um dos problemas resulta da possibilidade de declarar esses objectos com a ordem do índice crescente ou decrescente, a qual vai, naturalmente, influenciar a forma como a operação deve ser executada. É então necessário determinar a ordem dos índices antes de executar a operação. A listagem seguinte é um fragmento do procedimento de deslocamento lógico à esquerda utilizado no modelo da SU e mostra como os atributos em VHDL permitem obter aquele tipo de informação:

```
procedure l_shl(vecin: inout bit_vector) is
  variable aux: bit;
  begin
    if (vecin'left > vecin'rigth) then
      for i in vecin'range loop
        if (i > vecin'rigth) then aux := vecin(i - 1);
          else aux := '0';
        end if;
        vecin(i) := aux;
      end loop;
    ...
```

A condição `vecin'left > vecin'rigth` permite verificar se o índice do elemento mais à esquerda é maior que o índice do elemento mais à direita. Se assim for a declaração é da forma **x downto y** e a operação de deslocamento deve ser feita conforme mostra a listagem. Se a condição for falsa então a declaração do índice é da forma **x to y**, sendo necessário evoluir no vector atendendo à ordem crescente dos índices. De notar que o atributo `'range` devolve a gama do índice, tal como foi definida no objecto, isto é, **x downto y** ou **x to y**.

Implementação da SU

Um primeiro aspecto a realçar na SU é o desequilíbrio existente entre o elevado número de pinos e a complexidade de lógica necessária, que é relativamente baixa. Este tipo de indicador aconselha a implementar a SU em conjunto com outro bloco que partilhe os mesmos sinais (o que não é possível no caso do controlador do GLiTCH), ou a procurar componentes que se adaptem àquelas características. A versatilidade dos MACH, adapta-se bem a estas situações.

Novamente devido a limitações das ferramentas disponíveis, só foi possível implementar a SU recorrendo a dois componentes, um MACH110 e um MACH210 (em vez de um só MACH120). Dada a estrutura regular da SU, esta divisão não introduziu dificuldades adicionais, mormente no que respeita à verificação. De acordo com a metodologia de testes adoptada e devido às limitações referidas, para poder verificar o processo de síntese utilizando os mesmos vectores de testes da simulação, foi necessário fazer uma descrição PALASM aproximadamente equivalente à descrição VHDL. Este processo foi já adoptado na síntese do AGL mas, neste caso, a tradução não levantou qualquer tipo de problema, atendendo ao nível mais baixo da descrição VHDL da SU (refira-se, como excepção, a necessidade de substituir os procedimentos de deslocamentos por equações lógicas). A descrição em PALASM da SU e dos respectivos vectores de teste,

encontra-se no apêndice D. No capítulo 5 serão apresentadas algumas alternativas de implementação da SU.

4.5 Selector de *tags*

Atendendo à natureza associativa do GLiTCH, os sinais de *tag* (que indicam se dois operandos estão ou não "associados") assumem um papel extremamente importante, por serem o resultado de grande parte das instruções executadas. Num vector de Processadores Elementares (PEs), com memória associativa, é natural assegurar um mecanismo para a comunicação dos sinais de *tag*, entre os diverso processadores. No caso do GLiTCH essa comunicação é feita por dois meios [Duller, et al. 1989a] e [Duller, et al. 1989b]: um deles baseado num mecanismo de *barrel* para comunicações entre processadores a distâncias múltiplas de 4; o outro é linear e permite que cada PE comunique com um dos seus vizinhos. Para este último foram implementadas duas ligações físicas entre todos os sinais de *tag*, uma para cada sentido, fazendo recordar dois enormes registos de deslocamento. Nos topos é necessário fechar essas duas ligações, estabelecendo assim dois anéis.

Em certas operações, é necessário interromper aquelas ligações, o que apenas é possível fazer nos topos, para introduzir um determinado valor binário em ambos os anéis. O selector de *tags* (designado abreviadamente por TR) é utilizado para implementar essa funcionalidade.

De acordo com a descrição anterior, o TR deve executar o conjunto de instruções indicadas na tabela 4.3.

I2	I1	I0	Mnem.	Instrução
Ø	Ø	Ø	HOLD	Preserva última instrução
1	Ø	Ø	CHAIN	Fechar as cadeias de **TAGs**
Ø	1	Ø	SET	Forçar o valor lógico '1' em **TB** e **TA**
Ø	Ø	1	RESET	Forçar o valor lógico '0' em **TB** e **TA**

Notas:

- As restantes combinações estão reservadas
- Na documentação do projecto, o sinal **I2** é também designado por **Loop**

Tabela 4.3 Instruções do selector de *tags*

A figura 4.4 mostra o diagrama esquemático do TR, bem como a sua relação com os restantes componentes do sistema.

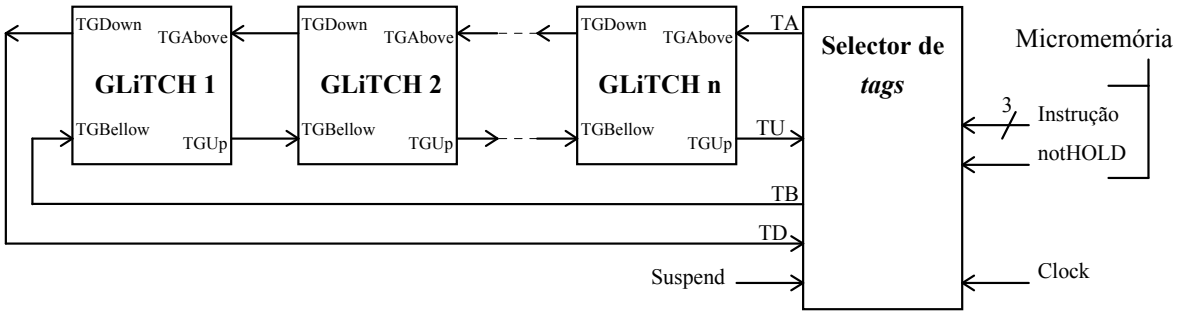


Figura 4.4 Diagrama lógico do selector de *tags* e suas ligações no sistema

À semelhança dos módulos DS e SU, também o TR dispõe de um sinal que o obriga a ignorar a instrução presente nas suas entradas e manter o seu estado interno executando a instrução HOLD. Esse sinal é identificado pelo nome **notHOLD** (activo ao nível lógico '0') no diagrama da figura 4.4, e tem como objectivo a optimização da micromemória. O mesmo tipo de função tem o sinal **Suspend**, muito embora este esteja associado ao acesso do transputer aos componentes partilhados. Apesar de o TR não estar directamente envolvido nesse acesso partilhado, o seu estado interno tem que permanecer inalterado enquanto o transputer assumir o controlo do sistema. De notar que estes sinais de controlo têm prioridade sobre o conjunto de sinais que codificam as instruções.

No que respeita à arquitectura interna, para além da funcionalidade desejada, deve-se ter em atenção o facto de o TR estar directamente ligado ao GLiTCH. De acordo com a descrição geral do sistema feita no início deste capítulo, o GLiTCH, e os respectivos componentes de suporte, impõe três níveis de *pipeline*. Para impedir que uma operação elementar que envolva o TR e o GLiTCH seja dispersa por várias microinstruções é necessário então incluir três registos de atraso no TR - garantindo assim o paradigma que estabelece a correspondência entre uma microinstrução e uma operação elementar do sistema.

Esta estrutura em *pipeline* afecta a operação dos sinais **notHOLD** e **Suspend**. O primeiro destes sinais está ligado apenas ao primeiro registo de *pipeline*, o qual, uma vez detectado o sinal activo, irá manter o seu valor, ou seja, a última instrução carregada. Essa mesma instrução será enviada para o segundo registo de *pipeline*, enquanto o anterior conteúdo deste é enviado para o último registo de *pipeline*. Este mecanismo é diferente do que deve estar associado ao sinal **Suspend**, o qual obriga simplesmente que todos os registos de *pipeline* mantenham o seu valor. Para tal é necessário que este sinal seja distribuído por todos os registos de *pipeline*.

4.5.1 Modelo VHDL do selector de *tags*

As condicionantes para o desenvolvimento do modelo VHDL do TR são muito idênticas às da SU, isto é, a especificação inicial é já feita a um nível bastante baixo e é necessário desenvolver *hardware* específico (lógica programável) para o TR. Este último aspecto exige que se descreva explicitamente a estrutura de *pipeline*.

Descrição do conjunto de instruções suportadas

De acordo com a metodologia adoptada, o conjunto de instruções executadas pelo TR (tabela 4.3) pode ser modelado através do seguinte conjunto enumerado (ver apêndice A - **package Gl_pack.vhdl**):

```
type tr_instr is (HOLD, A_RST, A_SET, CHAIN);
```

Descrição do interface externo

O passo seguinte consiste na descrição do interface externo do TR, o que é feito com base no respectivo diagrama esquemático da figura 4.4. O modelo VHDL daquele interface é o seguinte:

```
entity TR_c is
  port (signal Instruction: in tr_instr;
```

```
        signal TagUp, TagDown, notHold, Suspend,
               Clk: in bit;
        signal TagAbove, TagBellow: out bit);
end TR_c;
```

Descrição da funcionalidade

Atendendo à existência de três níveis de *pipeline* na arquitectura interna do TR, a sua funcionalidade é naturalmente descrita em VHDL através de três processos. Esses três processos são sensíveis apenas ao sinal de *clock*, sendo os sinais de controlo **notHold** e **Suspend** tratados de uma forma síncrona. A listagem seguinte mostra um fragmento dessa descrição (a descrição completa encontra-se no apêndice A):

```
architecture behavioral of TR_c is
-- Variaveis, constantes e sinais globais
signal Pipe1, Pipe2, Pipe3: tr_instr;
  begin
    proc_pipe1: process (Clk) -- pipe 1
    begin
      ...
      if ((Clk'event and (Clk = '1')) and (Suspend = '0')) then
        if ((notHold /= '0') and (Instruction /= HOLD)) then
          Pipe1 <= Instruction;
         else NULL;
      ...
    proc_pipe2: process (Clk) -- pipe 2
    begin
      ...
      if ((Clk'event and (Clk = '1')) and (Suspend = '0')) then
        Pipe2 <= Pipe1;
       else NULL;
      ...
    ...  proc_pipe3: process (Clk) -- pipe 3
    with Pipe3 select
      TagAbove <= '1'     when A_SET,
                  '0'     when A_RST,
                  TagDown when CHAIN,
                  '0'     when others;
                  -- HOLD nunca chega a Pipe3
    with Pipe3 select
      TagBellow <= '1'     when A_SET,
                   '0'     when A_RST,
                   TagUp   when CHAIN,
                   '0'     when others;
                   -- HOLD nunca chega a Pipe3
  end behavioral;
```

De acordo com a arquitectura descrita, o primeiro processo tem a particularidade de filtrar a instrução `HOLD` e de ser o único sensível ao sinal **notHOLD**. Quando aquela instrução é descodificada ou quando aquele sinal está activo, o primeiro processo não carrega o registo respectivo, mantendo a instrução anterior. Quando o sinal **Suspend** está activo todos os processos mantêm o seu registo interno inalterado. De notar que, de acordo com a norma, todos os sinais internos do tipo `tr_instr` assumem como valor inicial o valor `CHAIN`.

O último processo activa o sinal `Pipe3`, que por sua vez vai definir o estado lógico dos sinais de saída `TagAbove` e `TagBellow`. A instrução do VHDL utilizada para esse efeito (atribuição concorrente por selecção) modela directamente um multiplexador. A

semântica desta instrução obriga a que o valor de saída (`TagAbove` ou `TagBellow`) seja recalculado sempre que se verificar uma transição no sinal de selecção (`Pipe3`) ou em qualquer um dos sinais à direita do símbolo de atribuição (`TagDown` ou `TagUp`, respectivamente).

De notar que um multiplexador pode igualmente ser modelado por um processo, como em [Armstrong and Gray 1993] e [Perry 1991], mas nesse caso é necessário explicitar todos os sinais na lista de sensibilidade do processo. Se tal não for feito, o modelo não estará semanticamente correcto e isso só poderá ser detectado na simulação, se forem utilizados vectores de teste adequados.

Implementação do TR

O TR foi sintetizado numa PAL22V10. Tal como nas situações anteriores, por forma a explorar a possibilidade de utilizar o mesmo conjunto de vectores de teste na simulação e na síntese, foi necessário traduzir a descrição VHDL para uma descrição em PALASM. À semelhança do que se verificou com a SU, o nível de abstracção e a baixa complexidade do TR facilitaram essa tarefa. A descrição em PALASM do TR e dos respectivos vectores de teste, encontra-se no apêndice D.

4.6 Sequenciador

De acordo com a especificação, o sistema GLiTCH é microprogramado e neste tipo de arquitectura é o sequenciador ou, mais precisamente, o seu conjunto de instruções, que define o fluxo de controlo de qualquer programa a desenvolver. As facilidades disponíveis a este nível reflectem-se assim directamente no grau de complexidade desses programas. Neste caso particular o sequenciador exigido é bastante simples e, de uma forma resumida, deverá incluir as seguintes características:

- fazer executar em sequência as instruções contidas na micromemória;
- suporte da estrutura de controlo **IF**...**THEN**...**ELSE**..., sendo a condição definida pelo estado de 1 de 8 possíveis *flags*; e
- frequência de *clock* superior a 20MHz.

Admitindo que futuramente será desejável aumentar a funcionalidade do sequenciador, tendo ou não como objectivo o desenvolvimento de um interpretador que permita elevar o nível da programação, foi decidido alargar a especificação para garantir, pelos menos, a existência de um mecanismo de suporte à chamada de subrotinas.

De acordo com os critérios de selecção atrás enunciados a disponibilidade de um componente *standard* deve ser considerada antes do desenvolvimento de um componente específico. Entre as soluções disponíveis ao nível da microprogramação são particularmente versáteis as famílias de componentes introduzidas pela Advanced Micro Devices, donde foi seleccionado o sequenciador Am29C331. Este sequenciador verifica a especificação exigida e introduz ainda um conjunto de funções que poderão ser bastante úteis.

4.6.1 Família Am29300

Na década de 70, a AMD lançou uma família de componentes, designada por Am2900, constituída por um conjunto de componentes LSI destinados a implementar arquitecturas *Bit-Sliced* de sistemas microprogramáveis e controladores. Na base dessas arquitecturas estão os sequenciadores Am2909 e Am2910. O primeiro é um sequenciador de 4 bits, incluindo *stack* interna, um conjunto de instruções flexível e capacidade para operar a frequências até 5 MHz. O Am2910 é uma versão não *Bit-Sliced*, de 12 bits. Esta família de componentes, bem assim como um interessante conjunto de aplicações, encontra-se detalhadamente descrita em [AMD 1976] e [Siewiorek, et al. 1985].

Com a evolução tecnológica esta família de componentes sofreu algumas transformações, dando lugar a uma outra implementada em VLSI, não *Bit-Sliced* e para aplicações de 32 bits. Esta família é designada por Am29300 ou Am29C300 (uma versão implementada em CMOS). De uma forma semelhante, esta família inclui o sequenciador Am29331 (ou Am29C331), o qual satisfaz por excesso as exigências da especificação do controlador do GLiTCH. De uma forma sumária, este componente dispõe das seguintes características [AMD 1988]:

- endereço de saída de 16 bits, o que limita o tamanho da micromemória em 64K palavras;
- contador interno que permite a implementação de ciclos;
- máximo de 12 *flags* externas distintas (muito embora algumas delas ainda possam ser relacionadas logicamente para formar funções típicas como *overflow*);
- pilha interna com 33 níveis para facilitar a implementação de subrotinas e interrupções; de notar que o conteúdo da pilha pode ser acedida externamente para permitir, por exemplo, o *context switching* exigido em multiprocessamento;
- sinais externos dedicados à implementação de estruturas de saltos múltiplos;
- comparador interno para definição de *breakpoints*, especialmente úteis no *debug* de microprogramas;
- suporte de interrupções;
- facilidades de integração num ambiente de multiprocessamento ao nível da microprogramação.

Algumas destas facilidades (interrupções, multiprocessamento, acesso à pilha interna e alguns dos sinais dedicados a saltos múltiplos) não são exploradas no protótipo do controlador do GLiTCH com o objectivo de não aumentar consideravelmente a complexidade do *hardware*, tanto mais que, de momento, elas não são usadas pelas próprias ferramentas de desenvolvimento.

O gerador de endereços do sequenciador emprega, na maioria das instruções[3], uma técnica simples habitualmente designada por *two-way-branching*. De acordo com esta técnica, a próxima instrução a ser executada é a que se encontra na posição de memória imediatamente a seguir ou aquela cujo endereço se encontra na instrução em execução.

---

[3]No contexto desta secção os termos instrução e microinstrução são usados indistintamente.

Adicionalmente, algumas instruções permitem que os quatros bits menos significativos do endereço nelas incluído sejam definidos pelo valor existente no conjunto de quatro sinais de entrada dedicados. Para cada uma dessas instruções é então possível seleccionar a próxima instrução entre 17 possíveis (*multiple-way-branching*), embora com as limitações relativamente à sua localização na memória, decorrentes do mecanismo utilizado.

A figura 4.5 mostra o formato da instrução, o qual necessita de 26 bits - 6 para o código de operação, 4 para selecção das *flags* e 16 para o campo destinado ao próximo endereço.

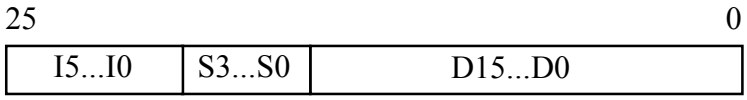


Figura 4.5 Formato da instrução do sequenciador

O conjunto de instruções que o sequenciador suporta encontra-se dividido em 4 grupos, codificados pelos dois bits mais significativos do código de operação, conforme mostra a tabela 4.4. Os três primeiros grupos são constituídos pelo mesmo conjunto de instruções, apenas alterando a forma como a condição seleccionada é utilizada.

I5	I4	Grupo
Ø	Ø	Saltos condicionais; IF cond THEN ...
Ø	1	Saltos condicionais; IF not cond THEN ...
1	Ø	Incondicionais
1	1	Instruções especiais

Tabela 4.4 Grupos de instruções do sequenciador

No formato da instrução estão reservados 4 bits para selecção da condição, o que permite a existência de 16 condições distintas. O sequenciador dispõe apenas de 12 entradas para *flags*, as quais estão linearmente associadas às primeiras 12 combinações binárias dos sinais de selecção referidos. As restantes 4 combinações identificam condições que resultam da aplicação de uma dada operação lógica sobre duas ou mais *flags*, do conjunto das 12 possíveis. Como exemplo, uma dessas equações define a condição "menor ou igual", em números sem sinal, desde que os sinais apropriados estejam ligados às usuais *flags* de *carry* e zero de uma ALU. Esta facilidade não é utilizada no protótipo do sistema GLiTCH (por não ser aplicável), mas pode vir a ser importante se considerarmos a hipótese de utilizar este mesmo controlador como unidade de microprogramação num outro sistema.

As tabelas 4.5a e 4.5b sumariam o subconjunto de instruções do sequenciador que podem ser utilizadas no protótipo, tendo em conta as limitações impostas à funcionalidade total do sequenciador. Essas instruções serão a base da especificação para um compilador (ou ferramenta equivalente) para o sistema GLiTCH. Mais detalhes relativos a este assunto, bem como às *flags* utilizadas e a respectiva semântica, são apresentados em [Santos and Proença 1993] e [Santos, et al. 1995].

I3-I0 (Hex)	Instrução Mnemónica	Próximo Endereço	Contador Interno	Comentário
Ø	Goto D	D	----------	PE = D
1	Call D	D	----------	Push (EA + 1); PE = D
2	Exit D	D	----------	Pop; Goto D
3	End for D	D	C = C-1	If not(C = 1) then PE = D
		EA + 1	C = C-1	If (C = 1) then PE = (EA+1)
8	Goto M	D:MØ	----------	PE = D:M0
9	Call M	D:MØ	----------	Push (EA + 1); PE = D:M0
A	Exit M	D:MØ	----------	Pop; PE = D:M0
B	End for M	D:MØ	C = C-1	If not(C = 1) then PE = D:M0
		EA + 1	C = C-1	If (C = 1) then PE = (EA+1)
C	End Loop[1]	TOS	----------	PE = TOS
D	Call Coroutine	TOS	----------	PE = TOS; Pop; Push (EA + 1)
E	Return	TOS	----------	PE = TOS; Pop
F	End For[2]	TOS	C = C-1	If not(C = 1) then PE = TOS
		EA + 1	C = C-1	If (C = 1) then PE = (EA+1); Pop

Notas:

TOS *Top of Stack*
EA Endereço Actual
PE Próximo Endereço
C Contador interno

[1] A versão condicional desta instrução executa um **Pop** se a condição não se verificar

Tabela 4.5a Instruções condicionais e incondicionais do sequenciador

A figura 4.6 mostra em maior detalhe o conjunto dos sinais do sequenciador utilizados no protótipo do controlador do GLiTCH, assim como a sua relação com as restantes unidades. Em particular deve ser referida a função do sinal **Hold**, o qual permite manter o estado interno do sequenciador e colocar as saídas num estado de alta impedância. Esta facilidade é utilizada para permitir partilhar com o transputer o acesso à micromemória, o que é necessário para teste e para carregar um microprograma. O sinal **Hold** é controlado pelo arbitrador de barramentos, que será descrito posteriormente.

I3-I0 (Hex)	Instrução Mnemónica	*Stack*	Contador Interno	Comentário
Ø	Continue	----------	----------	PE = (EC + 1)
1	For D	Push	C = D	C = D; Push (EC + 1)
2	Decrement	----------	C = C-1	PE = (EC + 1)
3	Loop	Push	----------	Push (EC + 1); PE = (EC + 1)
5	Push D	Push	----------	Push D; PE = (EC + 1)
6	Reset SP	----------	----------	SP = 0; PE = (EC + 1)
8	Pop C	Pop	C = TOS	C = TOS;PE = (EC + 1)
9	Push C	Push	----------	Push C; PE = (EC + 1)
A	Swap	TOS = C	C = TOS	C <> TOS; PE = (EC + 1)
B	Push C Load D	Push	C = D	Push C; C = D; PE = (EC + 1)
C	Load D	----------	C = D	C = D; PE = (EC + 1)
E	Set	----------	----------	BP = D; activa BP; PE = (EC + 1)
F	Clear	----------	----------	Desactiva BP; PE = (EC + 1)

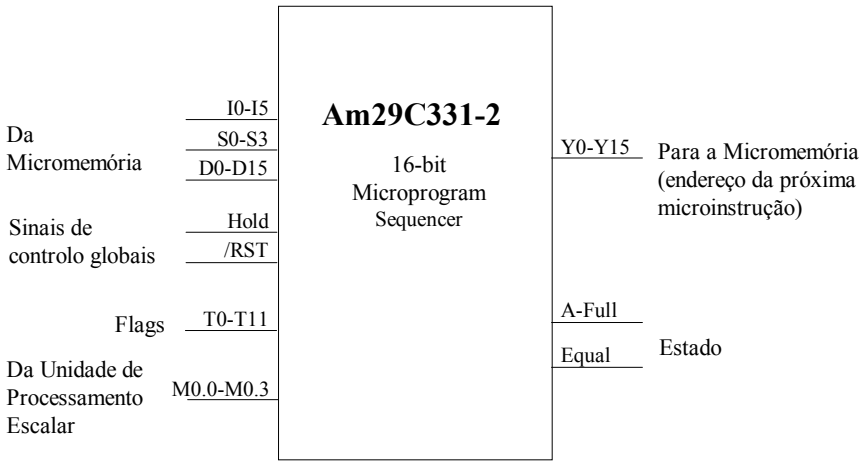
Notas:

Ver notas da tabela 4.5a

SP *Stack Pointer*

BP Registo que define o *Break Point*

Tabela 4.5b Instruções especiais do sequenciador



Legenda:

I0-I5 => Instrução

S0-S3 => Selector de Flag

D0-D15 => Dados (Próximo endereço;Valor a carregar no contador ou no comparador

A-Full => Indicador do estado da pilha interna

Equal => Activo sempre que se atingir um break-point previamente definido

Hold = > Mantém o estado interno e coloca as saídas no estado de alta impedância

Figura 4.6 Diagrama lógico do sequenciador

4.6.2 Modelo VHDL do sequenciador

De acordo com a metodologia adoptada, o modelo VHDL do sequenciador deverá ter a características enunciadas para um modelo de um componente *standard*. A funcionalidade do sequenciador não é totalmente utilizada e consequentemente a descrição

é reduzida. Assim, não será descrito o comportamento do sistema de interrupções, o mecanismo de multiprocessamento, o mecanismo de acesso externo à pilha e utilização completa do mecanismo de *multiple-way-branching*. De qualquer forma, ao nível de representação em questão, não será difícil completar a descrição de forma a incluir as facilidades adicionais, o que se poderá justificar na perspectiva da reutilização do mesmo modelo numa versão optimizada deste controlador, ou numa outra arquitectura.

Descrição do conjunto de instruções suportadas

De acordo com o formato de instrução do sequenciador (figura 4.5), ele pode ser descrito através de um *record*, do seguinte modo (ver apêndice A - **package Gl_pack.vhdl**):

```
type seq_instr is record
  instr: seq_instr_type;
  qualif: qualif_type;
  data: int_16;
  Sfs: integer range 0 to 15;
end record;
```

De acordo com as tabelas 4.4, 4.5a e 4.5b, o campo `qualif_type` serve para qualificar o tipo de instrução enquanto que o campo `seq_instr_type` identifica a instrução em particular. As instruções podem ser descritas por conjuntos enumerados, do seguinte modo:

```
type seq_instr_type is (GOTO_D, CALL_D, EXIT_D,
      END_FOR_D, GOTO_M, CALL_M, EXIT_M, END_FOR_M,
      END_LOOP, CALL_COR, S_RET, END_FOR,
      CONTINUE, FOR_D, DECREMENT, S_LOOP,
      PUSH_D, RESET_SP, POP_C, PUSH_C,
      SWAP, PUSH_C_LOAD_D, LOAD_D, SET, CLEAR);
type qualif_type is (COND, NOT_COND, INCOND);
```

As mnemónicas usadas foram tiradas directamente das anteriores tabelas e o qualificador das instruções apenas é utilizado com as primeiras 12 instruções.

Descrição do interface externo

Com base no diagrama da figura 4.6, o interface externo do sequenciador é descrito do seguinte modo:

```
entity AM29C331 is
  port (I: seq_instr;                  -- record...
        T: in bit_vector(11 downto 0);
        M0: in bit_vector (3 downto 0);
        Cp, Reset, Inten, Intr, Hold, OEd, Slave,
        CyIn, Fcont: in bit;
        Y: out mm_end; -- sutype do tipo integer
        Equal, A_Full, Inta, S_Error: out bit);
end AM29C331;
```

Alguns dos sinais usados estão associados a facilidades que não serão descritas no modelo (os respectivos sinais serão ignorados) e, tal como foi referido anteriormente, apenas constam para facilitar futuras alterações.

O tipo `mm_end` é um subtipo do conjunto dos números inteiros, limitado à gama de valores que o sequenciador pode gerar (ver **package Gl_pack.vhdl** no apêndice A). Esta opção tem duas consequências. Por um lado facilita a construção do modelo, que

internamente efectuará operações sobre inteiros. Por outro lado, ao não considerar a existência de um estado explícito de alta impedância nem de uma função de resolução, impede modelar a ligação física que existe entre as saídas do sequenciador e do transputer, para o acesso partilhado à micromemória. Este tipo de opção foi já considerado na descrição da metodologia de desenvolvimento.

Descrição da funcionalidade

A funcionalidade do sequenciador é demasiado complexa para ser descrita num único processo. Sendo assim, o primeiro passo consiste na sua partição em blocos de dimensão tratável. Embora essa divisão possa ser feita de uma forma arbitrária segundo a interpretação do projectista ao nível de abstracção considerado, o fabricante do componente fornece um diagrama de blocos adequado a esse fim, no respectivo manual técnico (o que pode ser generalizado para a grande maioria dos componentes).

Na figura 4.7 (obtida de [AMD 1988]) encontra-se o diagrama de blocos correspondente à partição feita. Esse diagrama evidencia algumas simplificações, face àquele que é fornecido pelo fabricante, resultantes das facilidades não utilizadas. Os nomes usados como etiquetas dos sinais da figura 4.7 coincidem com os nomes usados na descrição em VHDL, por forma a facilitar a leitura conjunta do diagrama e da descrição.

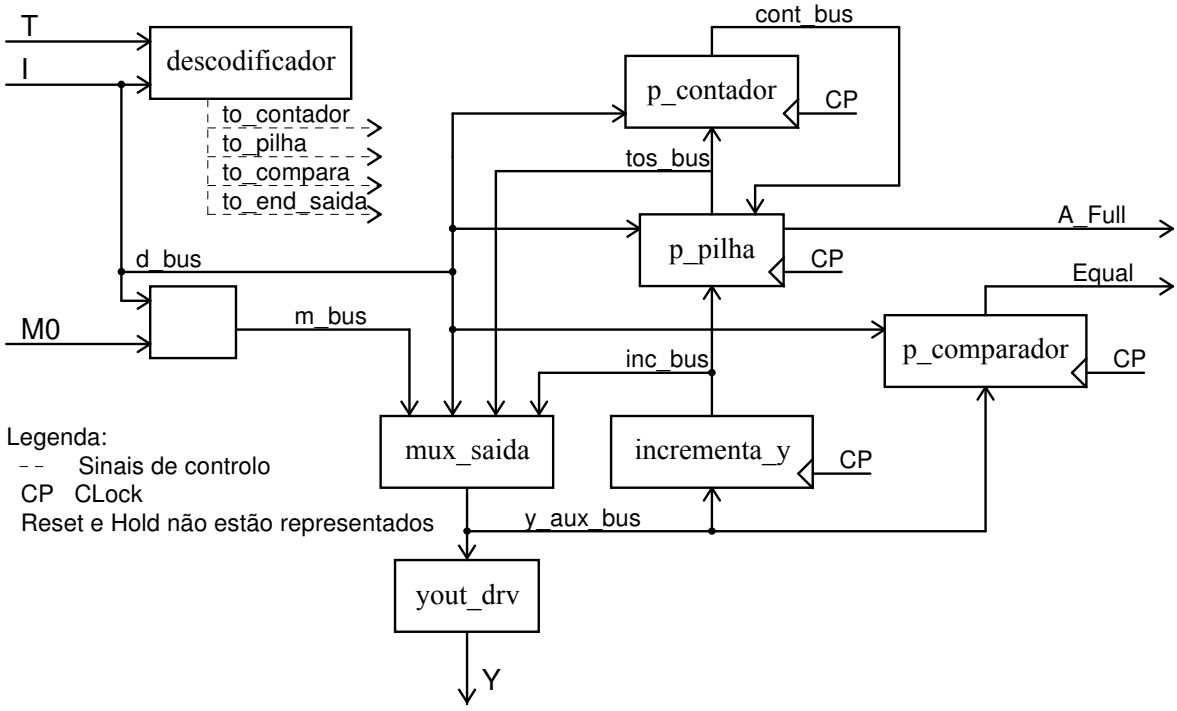


Figura 4.7 Diagrama de blocos do sequenciador

Como se pode observar na figura, o bloco descodificador recebe uma instrução a executar e o conjunto de *flags*, gerando como saída um conjunto de sinais para cada um dos restantes blocos. Esses sinais definem a operação a executar por cada bloco, de acordo com a instrução de entrada. Após resumir as operações de todos os blocos, é possível

identificar um conjunto global dessas operações e descrevê-lo utilizando um único conjunto enumerado, do seguinte modo:

```
type op_icode is (NOP, DE_D, DE_M0, DE_TOS, DE_INC,
      DE_CONT, ZERO, POP_I, DEC, DESAUT);
```

O conjunto `op_icode` contém todas as instruções suportadas por todos os blocos. É da responsabilidade do processo associado ao descodificador assegurar que cada sinal destinado a cada um dos restantes blocos apenas assumirá as instruções respectivas. Em alternativa podia utilizar-se o conceito de **subtype** do VHDL para restringir a cada sinal o subconjunto de instruções associadas e nesse caso o próprio simulador verificaria, dinamicamente, a consistência das atribuições. Em qualquer das soluções, é conveniente realçar que a descrição obtida está adequada à simulação e/ou documentação, mas geraria um desperdício elevado na síntese, uma vez que qualquer dos sinais de controlo declarados com aquele tipo seria implementado com 4 bits (para codificar 10 possíveis operações) e alguns deles apenas necessitam de 2 bits, atendendo ao número de operações do respectivo bloco.

Todos os sinais internos que representam barramentos de dados incluem no nome a sequência de caracteres `_bus`, podendo-lhes ser atribuído um valor de 16 bits. Para simplificar a descrição e a análise do modelo foi definido um tipo designado por `int_16`, que corresponde ao subconjunto dos valores inteiros que se podem representar com 16 bits (ver **package Gl_pack**, no apêndice A). Deste modo todas as operações internas serão feitas utilizando operadores inteiros, o que acelerará consideravelmente a simulação.

A arquitectura do sequenciador pode ser então modelada do seguinte modo, omitindo o corpo dos processos, cuja descrição completa encontra-se no apêndice A:

```
architecture behavioral of AM29C331 is
-- Barramentos de comunicacao entre os diversos modulos
signal d_bus, m_bus, inc_bus, cont_bus, y_aux_bus,
           tos_bus: int_16;
-- Barramentos de controlo:
signal to_pilha,to_comparador, to_contador,
       to_end_saida: op_icode;

  begin
    d_bus <= I.data; -- "pinos" I.data sao o driver do
                        -- barramento interno d_bus
    m_bus <= ((d_bus / 16) * 16) + bit2int(M0);
    ...  descodificador: process(I, T)
    ...  Mux_saida: process(to_end_saida, d_bus, m_bus,
                              tos_bus, inc_bus)
    ...  p_pilha: process(Reset, Hold, Cp)
    ...  incrementa_y: process(Reset, Hold, Cp)
    ...  p_contador: process(Hold, Cp)
    ...  p_comparador: process(Reset, Hold, Cp, y_aux_bus)
    ...  yout_drv: process(Reset, Hold, y_aux_bus)
  end behavioral;
```

O processo associado à pilha contém internamente a estrutura da pilha, implementada através de um vector de valores do tipo `int_16`. As declarações e subrotinas associadas à operação da pilha foram colocadas no **package seq_c** (ver descrição do sequenciador no apêndice A). Em particular as subrotinas utilizam atributos para inferir a dimensão da pilha, podendo assim ser facilmente reutilizadas.

Dois aspectos importantes devem ainda ser realçados. Por um lado a facilidade com que se associa esta descrição ao diagrama da figura 4.7. Cada bloco corresponde a um processo, cuja lista de sensibilidade inclui os respectivos sinais de entrada. De notar que a instrução concorrente de atribuição ao barramento `m_bus` corresponde também a um processo, cuja simplicidade permite o recurso àquela forma elementar. Formalmente, esta descrição podia ser feita num único processo, descrevendo a funcionalidade de cada um dos blocos através de subrotinas. Isto não alteraria a estrutura da solução e poderia ser útil no caso de se considerar a síntese da descrição comportamental do sequenciador com ferramentas que admitem um único processo em descrições comportamentais.

O segundo aspecto a realçar é a ausência da descrição temporal detalhada, apesar de um dos objectivos da divisão em blocos ser exactamente a possibilidade de introduzir essa informação no modelo. Esta opção pode-se justificar pela sua irrelevância no contexto da sua primeira aplicação, que é a avaliação funcional do controlador do GLiTCH. De qualquer modo, a inclusão daquela informação no modelo descrito é uma tarefa simples.

4.7 Micromemória

A micromemória é constituída por 3 bancos de memória estática, de 32Kx32 bits cada, organizada em palavras de 96 bits. O tipo de memória e a dimensão, são características que foram impostas por restrições funcionais e de disponibilidade de componentes. No que respeita ao tamanho da palavra, para conseguir compactar toda a informação pertinente a uma microinstrução em 96 bits, foi necessário colocar algumas unidades funcionais a partilhar alguns campos, utilizando o método anteriormente descrito. A validade do arranjo proposto baseia-se na análise dos programas já desenvolvidos para o simulador, de onde se pode concluir a frequência de utilização e a sobreposição da operação das várias unidades. Outra alternativa considerada foi a redução da largura de cada banco, por forma a ter maior resolução na expansão do tamanho da palavra. Esta solução foi abandonada, por se revelar bastante dispendiosa em termos de custo e área, devido ao elevado número de circuitos de memória necessários.

Deve ser salientado que o mecanismo de partilha de alguns dos campos da microinstrução mostrou ter um impacto pouco relevante no tamanho dos programas já desenvolvidos. No entanto, o seu efeito noutras aplicações, no mesmo domínio ou noutros domínios de aplicação, deverá ser investigada em maior profundidade, o que poderá ser feito de uma forma mais eficiente, com o recurso a uma ferramenta automática de geração de código.

No planeamento da organização da microinstrução, procurou-se que cada campo, ou mesmo conjunto de campos destinados à mesma unidade, não fosse fragmentado pela fronteira de qualquer uma das suas três subpalavras de 32 bits. O objectivo é procurar facilitar a tarefa do gerador de código, ou de qualquer outra aplicação para o transputer que necessite de operar com microinstruções, partindo do princípio que a linguagem de desenvolvimento a utilizar tem uma representação para valores inteiros de 32 bits. A figura 4.8 mostra a organização da microinstrução e, para além dos campos anteriormente descritos para as diversas unidades do controlador, inclui ainda, nos campos a sombreado,

as instruções e sinais de controlo para o GLiTCH e componentes de suporte [Santos and Proença 1993].

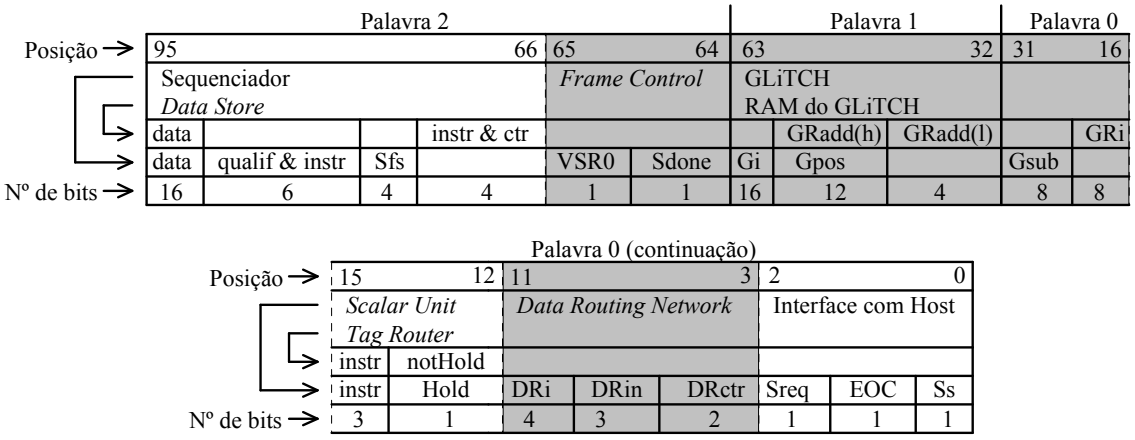


Figura 4.8 Definição da microinstrução do sistema

Na figura 4.8, o campo da microinstrução designado por **Interface com Host**, contém um conjunto de sinais para o transputer: **Sreq** é utilizado para assinalar o pedido de execução de uma operação escalar (tal como era exigido na especificação inicial); **EOC** assinala o fim de um microprograma; e **Ss** assinala uma instrução a ser executada no modo passo-a-passo, facilidade esta que foi acrescentada para facilitar o desenvolvimento de aplicações. A descrição detalhada destas operações é feita na secção que descreve a funcionalidade do transputer.

4.7.1 Modelo VHDL da micromemória

O modelo VHDL para a micromemória tem como objectivo a simulação e a documentação, uma vez que, fisicamente, a micromemória é implementada com componentes *standard*. Sendo assim, justifica-se o desenvolvimento de um modelo comportamental, a um nível de abstracção elevado (*Architecture*), ignorando algumas das características da arquitectura interna.

A micromemória pode ser definida como um vector de microinstruções, sendo cada uma composta por vários campos associados às instruções das diversas unidades funcionais. Nesta definição simples não há qualquer referência ao processo de codificação das instruções, ao tamanho dos campos, ou ainda à partilha de campos da microinstrução. Em particular este último aspecto resultou exactamente duma limitação da implementação, não sendo assim de estranhar que ele possa ser ignorada numa descrição abstracta da micromemória.

Em VHDL a microinstrução pode ser descrita através de um *record*, da seguinte forma:

```
type mm_instr is record
  Seqi: seq_instr;
  DSi:   agl_instr;
--  VSR0, Sdone: bit;
--  Gi:    glitch_instr;
--  Gpos:  integer range 0 to (2**12) - 1;
```

```
--  GRadd: int_16;
--  Gsub:  integer range 0 to (2**8) - 1;
--  GRi:   gram_instr;
  SUi:   su_instr;
  TRi:   tr_instr;
-- DRi:   dr_instr;
-- DRin:  dr_indata;
-- DRctr: dr_ctr;
  Sreq, EOC, Ss: bit;
end record;
```

Os campos comentados correspondem às instruções para o GLiTCH e componentes de suporte (zonas sombreadas na figura 4.8), e apenas serão utilizados se, e quando, existirem modelos VHDL daqueles componentes. Os restantes campos foram já definidos e correspondem todos a unidades funcionais do controlador do GLiTCH.

Descrição do interface externo

À semelhança de uma vulgar memória de leitura e escrita, o interface externo da micromemória inclui os sinais de endereço, o barramento de entrada/saída e os sinais de controlo, podendo ser descrito do seguinte modo:

```
entity mm is
   port (endereco: in mm_end;
         dado: inout mm_instr;
         notcs, notrd, notwr: in bit);
end mm;
```

O tipo `mm_end` está declarado no **package Gl_pack.vhdl** (anexo A), como um subconjunto dos números inteiros, correspondente à gama de endereços, ou seja, à dimensão da micromemória.

De acordo com a especificação e a metodologia de teste assumida, o acesso à micromemória é partilho entre o transputer e os diversos componentes do controlador do GLiTCH. No entanto, por razões de eficiência (ver descrição do DS), o acesso a partir do transputer não é considerado no modelo VHDL, apesar de se procurar incluir o máximo de detalhe possível nesse sentido. Serve de exemplo a esta afirmação a utilização do tipo `std_logic` para os sinais de controlo da memória do DS, apesar de no modelo apenas o AGL controlar esses sinas. Dentro desta lógica, seria de esperar um tratamento idêntico para os sinais de controlo da micromemória. Contudo, no que respeita à utilização da micromemória pelo controlador do GLiTCH, existem dois factos que devem ser observados: a micromemória é utilizada apenas para leitura; e nenhum componente tem atribuída a função de controlar a micromemória. O transputer é, então, o único responsável pelos sinais de controlo, os quais devem ficar activos para leitura sempre que a micromemória estiver a servir o sistema GLiTCH. Deste modo, justifica-se a utilização na listagem anterior do tipo `bit`, para os sinais de controlo da micromemória.

Descrição da funcionalidade

A micromemória pode ser modelada utilizando um só processo, no interior do qual a memória em si é implementada através de uma variável. As operações de escrita e de leitura são descritas por procedimentos, o que facilita a reutilização, ao isolar eventuais detalhes relativos ao tipo de dados utilizado. Tal como nas estruturas similares da *stack* do sequenciador e da memória do DS, também neste modelo são utilizados atributos para

inferir a dimensão e os limites dos endereços. Desta forma, o modelo é independente da organização da memória.

Dado que foi excluído do modelo o acesso à micromemória a partir do transputer, e que a inicialização da micromemória assenta nessa relação, torna-se necessário utilizar um mecanismo alternativo para essa função. No modelo VHDL da micromemória, o carregamento de um microprograma é simulado na fase inicial de execução do processo, onde é lido um ficheiro que pressupostamente contém um microprograma codificado. A estratégia de codificação será descrita mais à frente. A listagem seguinte é um fragmento da descrição da micromemória (a descrição completa encontra-se no anexo A):

```
architecture behavior of mm is
  type mm_bloco is array (mm_end'low to mm_end'high) of
                mm_instr;
  begin
    process (endereco, dado, notcs, notrd, notwr)
      variable memoria: mm_bloco;
      variable celula: mm_instr;
      file TESTVEC: text is "gl_prog.txt";
      ... procedimentos locais
      begin
        if inicio then
          ...  Bloco de inicializacao
        end if;
        if (notcs = '0') then
          pont := endereco;
          temp := notrd & notwr;
          case (temp) is
            ...
            when "01" => le_mm(memoria, pont, celula);
                         dado <= celula;
            ...
          end case;
        end if;
    end process;
end behavior;
```

A declaração do objecto *file* define o nome do ficheiro a utilizar, o qual é lido pelo conjunto de instruções simbolicamente representado na listagem anterior por `... Bloco de inicializacao`.

As limitações da linguagem VHDL, relativamente à estrutura de ficheiros, criou obstáculos à definição de um formato adequado para o ficheiro de inicialização da micromemória. A solução encontrada assenta na relação de ordem existente nos conjuntos enumerados e nos atributos associados. Cada linha daquele ficheiro é, então, constituída por uma sequência de valores inteiros, separados por um ou mais espaços em branco, correspondendo o primeiro valor ao endereço da micromemória onde se pretende carregar a instrução e os valores seguintes ao número de ordem de um identificador no respectivo conjunto enumerado, ou a um valor inteiro que é um operando da instrução associada, conforme se aplique. A ordem destes valores é a definida na declaração do tipo `mm_instr`. O exemplo da figura 4.9 ilustra a definição do formato (ver também a definição das instruções individuais):

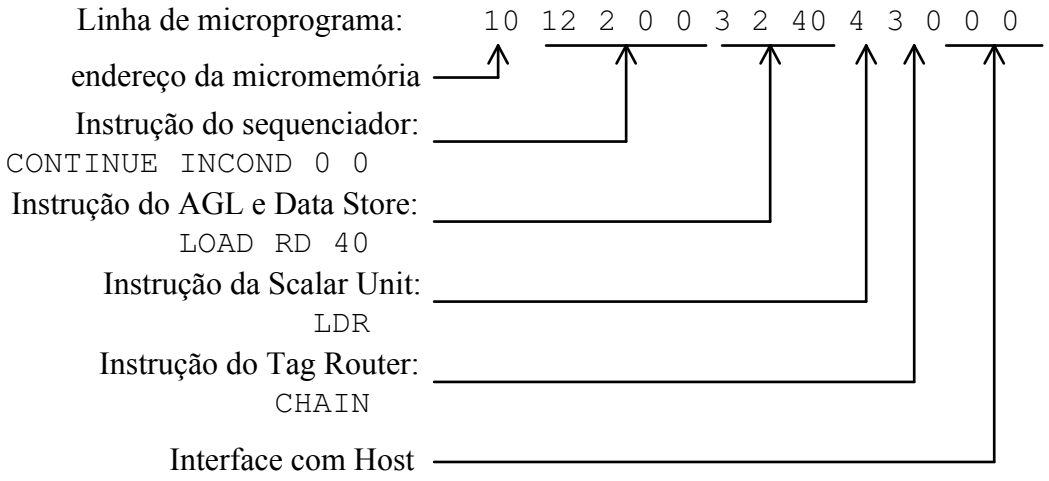


Figura 4.9 Exemplo de uma linha de microprograma

A listagem seguinte é um fragmento do bloco de inicialização da micromemória, correspondente à leitura da instrução para o sequenciador, e mostra a forma como os valores inteiros lidos do ficheiro são relacionados com a representação interna:

```
...
read(prog_linha, aux, arg_ok);
...
memoria(aux_end).Seqi.instr := seq_instr_type'val(aux);
read(prog_linha, aux, arg_ok);
...
memoria(aux_end).Seqi.qualif := qualif_type'val(aux);
read(prog_linha, aux, arg_ok);
...
assert ((aux >= int_16'low) and (aux <= int_16'high))
              report "Operando para Sequenciador fora de gama!"
                severity ERROR;
memoria(aux_end).Seqi.data := int_16'val(aux);
...
```

O atributo `T'val(x)` - em que 'T' representa um tipo escalar e 'x' um valor inteiro - devolve o elemento do tipo 'T' cujo número de ordem corresponde ao valor 'x'. Se o valor estiver fora da gama da numeração do respectivo tipo, o próprio simulador abortará com a indicação do erro. Quanto aos valores inteiros associados a dados da instrução, a sua validade é testada, atendendo ao tipo do objecto a que se destinam, através de uma instrução `assert`, tal como é ilustrado na listagem anterior.

4.8 Processador hospedeiro

A única tarefa atribuída a este processador (também designado por HT), de acordo com a especificação inicial, é a execução de operações escalares complexas, não adequadas à arquitectura vectorial e associativa do GLiTCH, e que não possam ser executadas na SU (como por exemplo operações em vírgula flutuante). Durante as descrições anteriores dos diversos componentes do sistema, foram identificadas outras funções que naturalmente serão executas por este processador, como sejam o carregamento de um microprograma, o teste/diagnóstico do sistema e as funções de suporte ao desenvolvimento de programas - execução de instruções passo-a-passo e com pontos de paragem. São estas funções que

atribuem a este módulo a característica de um hospedeiro. Devido a um conjunto de opções tomadas ao nível do sistema, e já anteriormente referidas, o processador adoptado para esta função foi um transputer.

Para implementar o interface entre o HT e o sistema GLiTCH, foi necessário resolver dois problemas: o sincronismo entre os dois sistemas, atendendo a que utilizam *clocks* diferentes; e a passagem de parâmetros, no pedido de execução de operações escalares. Para resolver este último problema, e utilizando os recursos já existentes, basta definir uma região do DS dedicada a esse fim. Esta decisão não implica a introdução de nenhuma estrutura de *hardware* adicional e não impõe restrições ao tamanho dos operandos. O único inconveniente é a necessidade de suspender a execução do sistema GLiTCH, enquanto o HT acede ao(s) parâmetro(s). No entanto, tendo como referência os algoritmos já desenvolvidos, o impacto deste mecanismo de passagem de parâmetros no desempenho global não será apreciável, já que a sua frequência de utilização é baixa.

Para garantir o sincronismo foi introduzido um arbitrador de barramentos (em seguida designado abreviadamente por BA - *Bus Arbiter*), que utiliza o *clock* do controlador e que implementa um tipo de sincronismo habitualmente designado por *interlocked two-wire handshake* [Leibson 1989]. O BA tem por função garantir a correcta transferência do controlo dos barramentos partilhados, entre o sistema GLiTCH e o HT, e ainda fornecer a este último um conjunto de *flags* que indicam o estado do sistema. Do ponto de vista arquitectural, o BA é composto por uma máquina de estados principal, responsável por gerar os sinais de controlo dos barramentos, e um conjunto de máquinas de estado concorrentes, individualmente responsáveis pelo sincronismo de cada um dos sinais capazes de provocar a comutação do acesso aos recursos partilhados. A função do BA pode ser resumidamente descrita pelo diagrama temporal da figura 4.10 (a sua descrição detalhada é feita em [Santos and Proença 1993]).

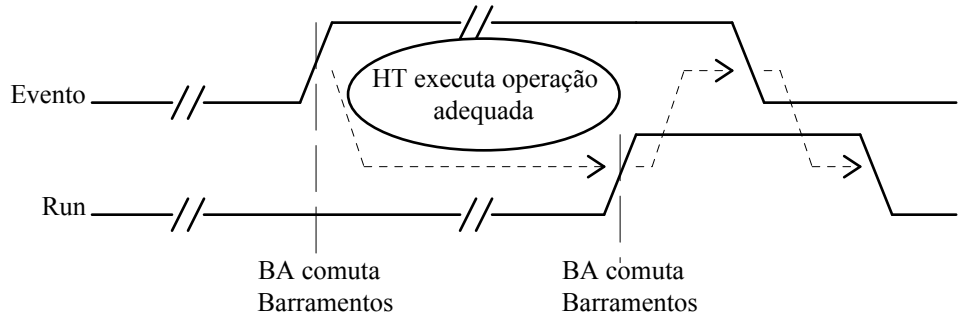


Figura 4.10 Diagrama temporal do arbitrador de barramentos

Na leitura do diagrama deve-se ter em atenção o seguinte:

- **Evento** simboliza um conjunto de sinais capazes de gerar autonomamente um pedido de intervenção ao HT.
- ***Run*** corresponde a um sinal real, que é activado pelo HT para indicar ao BA que o sistema GLiTCH pode (re)começar a executar.
- As linhas a ponteado indicam dependências entre sinais.
- A operação de comutação dos barramentos é executada em dois passos, para evitar a contenção que ocorreria nos barramentos se a comutação fosse instantânea.

A figura 4.11 mostra o diagrama esquemático do BA. No contexto do diagrama da figura 4.10, os sinais capazes de provocar um evento são designados por **BusReq**, **Ss**, **Finish**, **SC_Req** e **Equal**. De uma forma resumida, a função associada a cada um dos eventos é a seguinte:

- **BusReq** é activado pelo HT, para interromper o sistema GLiTCH e aceder aos seus componentes.
- **Ss** provém da micromemória, e activa o mecanismo de execução passo-a-passo, para a microinstrução associada.
- **Finish** (ou **EOC**) provém da micromemória, e permite identificar a última microinstrução de um microprograma.
- **SC_Req** provém da micromemória, e activa o mecanismo associado a um pedido para o HT executar uma operação escalar.
- **Equal** provém do sequenciador, e indica que este atingiu uma microinstrução cujo endereço tinha sido previamente marcado como ponto de paragem.

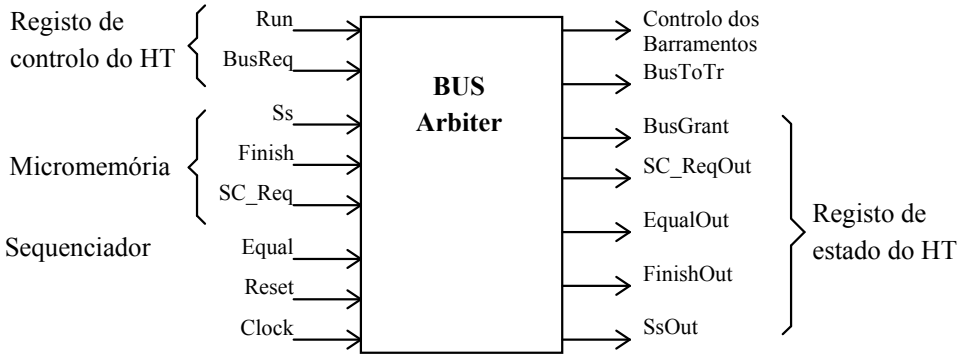


Figura 4.11 Diagrama lógico do arbitrador de barramentos

O conjunto de sinais designado por **controlo dos barramentos**, no diagrama da figura 4.11, inclui os sinais **BusTrnotLoc1**, **SuspendGL** e **BusTrnotLoc2**. Os dois primeiros são logicamente equivalentes, e são utilizados no subsistema do controlador e do subsistema do GLiTCH, respectivamente. Estes sinais controlam o estado de alta impedância dos componentes envolvidos no acesso partilhado[4] (o sinal **BusTrnotLoc1** foi várias vezes referido nas descrições dos restantes componentes do controlador do GLiTCH). O sinal **BusTrnotLoc2** tem uma função idêntica, mas relativamente aos componentes homólogos do bloco do transputer. Tal como foi referido, a comutação de estado destes sinais é feita em períodos de *clock* distintos, para evitar problemas de contenção no acesso aos barramentos.

Atendendo à sua função de arbitrador do acesso aos barramentos partilhados, a (re)inicialização do BA define o comportamento de todo o sistema após o arranque e após a activação do sinal de *reset*. Atendendo à função do HT, após a (re)inicialização, o BA deverá dar-lhe o controlo dos barramentos (no mínimo para transferir um microprograma e

[4] A necessidade de dois sinais resulta de restrições temporais impostas pelos componentes de suporte do GLiTCH.

os dados associados), e aguardar até que o sinal **Run** fique activo, altura em que sistema GLiTCH irá executar um microprograma.

4.8.1 Modelo VHDL do arbitrador de barramentos

Os únicos condicionamentos para o desenvolvimento do modelo VHDL do BA são: o seu comportamento temporal pré-determinado (o que define à partida o *scheduling* de todas as operações); e a necessidade de desenvolver *hardware* específico (lógica programável) para a sua implementação física. Atendendo a que o BA é composto por máquinas de estado, este último condicionamento não se traduz em qualquer restrição em termos de nível de representação, uma vez que todos os sistemas de síntese suportam bem as máquinas de estado - estrutura lógica fundamental em sistemas digitais. Assim sendo, o BA pode ser descrito ao nível *Architecture*, no domínio comportamental.

Descrição do interface externo

A construção do modelo VHDL começa pela descrição o seu interface externo, a qual é obtida a partir do diagrama da figura 4.11, e pode ser feita do seguinte modo:

```
entity bus_arb1 is
  port (Run, BusReq, -- Do registo de controlo do transputer
        Ss, Finish, SC_Req,     -- da Micromemoria
        Equal,                  -- do sequenciador
        Reset, Clock: in bit;   -- Globais do sistema
        SuspendGL, BusTrnotLoc1, -- Controlo dos barramentos
        BusToTr,
        BusTrnotLoc2,  -- este tb. serve de flag -> BusGrant
        SC_ReqOut, EqualOut,    -- Flags para o transputer
        FinishOut, SsOut: out bit);
end bus_arb1;
```

Descrição da funcionalidade

Ao nível *Architecture*, no domínio comportamental, cada máquina de estados síncrona e sem saídas condicionais, pode ser descrita por um único processo e associando um tipo enumerado ao conjunto de estados. A listagem seguinte é um fragmento da descrição da arquitectura do BA (a descrição completa encontra-se no apêndice A), e pretende realçar a organização do modelo:

```
architecture behavior of bus_arb1 is
  type Main_asm is (Normal_OP, Phase1, Phase2, RetPhase1,
                    RetPhase2);
  type Equal_asm is (NoEqual, EqualRun);
  type Finish_asm is (NoFinish, FinishRun);
  type Ss_asm is (NoSs, SsRun);
  type SC_Req_asm is (NoSC_Req, SC_ReqRun);
  type BusReq_asm is (NoBusReq, BusReqRun);
  signal ClrFlags: bit;
  signal BusTrnotLoc1aux, BusToTraux, BusReqOut: bit;
  begin
    BusToTraux <= Equal or Finish or Ss or SC_Req or
                  BusReqOut or BusTrnotLoc1aux;
    BusToTr <= BusToTraux;
    ...  Main_p: process
    ...  Equal_p: process
    ...  Finish_p: process
    ...  Ss_p: process
    ...  SC_Req_p: process
```

```
    ...  BusReq_p: process
  end behavior;
```

Os identificadores usados nas declarações dos tipos que representam os conjuntos de estados, são os nomes usados para identificar os estados nos respectivos diagramas de estado, detalhadamente descritos em [Santos and Proença 1993]. Os sinais **BusToTr** e **ClrFlags** destinam-se a estabelecer a comunicação entre a máquina de estados principal e todas as outras. Adicionalmente, o sinal **BusToTr** é ainda utilizado externamente pelo gerador dos sinais de *clock* (instalado na placa de circuito impressos do módulo GLiTCH), razão pela qual está associado a um **port**.

A utilização do sinal interno **BusToTraux** pode parecer redundante, uma vez que o seu comportamento lógico é exactamente igual ao do sinal **BusToTr**. Esta duplicação resulta do facto deste último sinal corresponder a um **port** de saída, não podendo, por esse motivo, ser utilizado no lado direito de uma expressão de atribuição. Segundo a norma da linguagem, quando essa funcionalidade é pretendida, é necessário declarar o respectivo **port** com o modo entrada/saída, mas tendo consciência da alteração semântica que se está a introduzir. Uma solução alternativa é declaração de um sinal interno equivalente, o que na maioria das vezes modela com mais exactidão a funcionalidade pretendida.

A listagem seguinte mostra a descrição associada à máquina de estados responsável pela detecção de um dos eventos, e tem como objectivo realçar o estilo de descrição utilizado na elaboração deste tipo de processo:

```
Equal_p: process (clock, Reset)
  variable estado: Equal_asm := NoEqual;
  begin
    if Reset = '1' then
      estado := NoEqual; EqualOut <= '0';
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoEqual => if (BusTrnotLoc1aux = '0') and
                              (Equal = '1') then
                            estado := EqualRun;
                            EqualOut <= '1';
                          end if;
           when EqualRun => if ClrFlags = '1' then
                              estado := NoEqual;
                              EqualOut <= '0';
                            end if;
         end case;
       end if;
    end if;
  end process;
```

Esta descrição não é única e impõe naturalmente uma determinada semântica, a qual pode ainda variar, no contexto da sua utilização com uma ferramenta de síntese automática. Uma questão importante (e já levantada no capítulo 3) é conhecer o impacto que um determinado estilo de descrição tem na síntese automática. Atendendo ao nível de abstracção utilizado, o BA é o componente específico desenvolvido neste sistema que mais se presta a esse estudo. Com esse objectivo, foram construídas e sintetizadas algumas descrições alternativas. Os resultados obtidos são analisados no capítulo seguinte.

Implementação do arbitrador de barramentos

O BA foi implementado utilizando dois componentes PAL16V8. Tal como nas situações anteriores, por forma a explorar a possibilidade de utilizar o mesmo conjunto de vectores de teste na simulação e na síntese, foi necessário traduzir a descrição VHDL para uma descrição em PALASM. Nessa tradução as alterações não foram significativas, atendendo a que a linguagem suporta bem a representação de máquinas de estado. A divisão entre componentes também não levantou qualquer problema, uma vez que o BA é composto por um número elevado de máquinas de estado de pequena dimensão. A descrição em PALASM do BA e dos respectivos vectores de teste, encontra-se no apêndice D.

4.8.2 Modelo VHDL da operação do processador hospedeiro

Do ponto de vista do sistema, o HT é um módulo constituído por um transputer T805 - um componente VLSI que inclui um processador de 32 bits, 4K bytes de memória, uma unidade de vírgula flutuante de 64 bits e 4 *links*, para ligação directa com outros tantos transputer*s* ou componentes compatíveis - com 1 ou 4Mbytes de memória dinâmica local, um conjunto de portas de entrada e de saída, e as habituais facilidades de utilização em multiprocessamento, que caracterizam este tipo de componente [INMOS 1992]. No anexo B encontra-se a descrição esquemática detalhada deste módulo.

Do ponto de vista do modelo em desenvolvimento, o HT é uma unidade capaz de executar um conjunto de funções (ou algoritmos) relacionadas com a operação do sistema GLiTCH. De notar que essas funções foram já introduzidas nas descrições dos restantes módulos que constituem o controlador do GLiTCH.

Estas duas visões do mesmo *hardware* envolvem níveis de complexidade muito diferentes e permitem tirar duas conclusões importantes:

- o módulo TH possui mais capacidades do que as necessárias para as funções a implementar;
- a este nível de representação, o que se pretende modelar não é a funcionalidade do transputer em si, mas sim a funcionalidade de um programa que, ao executar, implementa as funções desejadas.

A primeira conclusão permite levantar o problema da adequabilidade do transputer ao projecto. Três argumentos importantes pesam a favor dessa utilização: integração do sistema GLiTCH numa rede de transputer*s*; facilidades de desenvolvimento de um sistema baseado num transputer (facilidades essas que estão inerentes aos princípios que nortearam a definição da sua arquitectura); e possível inclusão de novas funções de controlo e de processamento.

A segunda conclusão traduz-se na não necessidade de construir um modelo do transputer e dos componentes que lhe dão suporte, o que iria adicionar apenas detalhes irrelevantes. Para reforçar este facto, note-se que, do ponto de vista do *hardware*, aquele módulo de processamento é assumido como correctamente construído, o que é possível afirmar com base na sua utilização em muitos outros projectos, a menos da especificidade das entradas e saídas. Esta afirmação assenta no princípio da reutilização, que é uma característica básica do processo da engenharia.

Desta forma, e se o simulador de VHDL disponível permitisse a integração de outras linguagens, o HT poderia até ser modelado através do conjunto de programas reais desenvolvidos para as funções respectivas. Esta facilidade apenas está considerada na revisão de 1992 da norma VHDL [Bergé, et al. 1993], a qual não é ainda suportada pela maioria dos sistemas de desenvolvimento. Sendo assim, é necessário descrever aqueles algoritmos em VHDL, pelo menos os essenciais para o funcionamento restrito que se pretende modelar.

Descrição do interface externo

De acordo com a descrição anterior, o interface externo do HT é definido em VHDL, do seguinte modo:

```
entity TH is
  port(ds: inout std_logic_vector(31 downto 0);
       ds_notcs: out std_logic_vector(1 downto 0);
       ds_notrd, ds_notwr: out std_logic;
       mm_notcs, mm_notrd, mm_notwr: out bit;
       BusReq, Run, Sw_Ini: out bit;
       Equal, Finish, AFull, SC_Req, SDone, SStep,
       BusTrnotLoc2: in bit);
end TH;
```

Os sinais cujos identificadores começam por ‘ds’ e ‘mm’ destinam-se a permitir o acesso ao DS e à micromemória, respectivamente. De notar que, à excepção do barramento de dados do DS, apenas estão presentes os sinais de controlo, pois como anteriormente foi referido, a função de acesso directo a partir do HT não será incluída neste modelo para simulação. Esta opção foi tomada com o objectivo de diminuir a complexidade do modelo e de optimizar o tempo de resposta do simulador, ao evitar o recurso a funções de resolução e de conversão de tipos. Por outro lado, a inclusão de todos os sinais, facilitará um possível actualização do modelo.

O sinal **Sw_Ini** destina-se a fazer o *reset* do sistema GLiTCH por *software*, e os restantes sinais interligam o BA com o HT.

Descrição da funcionalidade

Atendendo aos objectivos do modelo a construir, à excepção da função de execução de uma operação escalar, as restantes funções não necessitam de ser implementadas, uma vez que estão relacionadas com facilidades de teste e desenvolvimento e não intervêm no que se pode chamar a execução normal do sistema GLiTCH. Contudo, a sua implementação deve ser considerada no caso de se pretender utilizar o modelo para verificar microprogramas, uma vez que este representa o sistema mais fielmente do que o simulador anteriormente construído. Este tipo de utilização deve ser encarado com algumas reservas, dado o nível de complexidade do modelo e da própria tarefa de simulação.

De uma forma resumida, o programa a construir para o transputer é assíncrono, relativamente ao resto do sistema, e consistirá de um ciclo contínuo em que são testadas as

flags associadas a cada um dos eventos[5]. Sempre que ocorrer um evento, a respectiva tarefa é executada. Em termos de VHDL, o algoritmo deve ser descrito por um ou mais processos. Mas a semântica da linguagem VHDL, definida para a simulação, baseia-se na noção de transição do valor de um sinal como mecanismo para reexecutar um processo, o que não está de acordo com a descrição anterior. É possível construir um processo em execução aparentemente permanente, mas este tipo de processo degrada significativamente o desempenho do simulador.

Por este motivo, a solução adoptada baseou-se na construção de um processo sensível a todas as *flags*, sacrificando a exactidão do modelo em favor do desempenho. Em termos de análise do funcionamento do sistema GLiTCH, apenas é relevante o erro introduzido no tratamento do sinal **SC_Req**: enquanto no modelo o pedido é executado imediatamente (em termos de simulação), no HT real o tempo de resposta pode depender de vários factores, em particular da posição no ciclo de teste em que o transputer se encontra a executar.

A listagem seguinte é um fragmento da descrição da função do HT e tem como objectivo realçar a forma como o processo foi estruturado (a descrição completa encontra-se no apêndice A).

```
architecture behav_spool of TH is
... Declaracoes globais
begin
  spool: process
  variable inicio: boolean := TRUE;
  variable Suspende: bit; -- simula um pedido de suspensao.
  begin
    ...  Inicializacao
    if Equal = '1' then
       ...  accoes associadas ao "Break Point"
    end if;
    if Finish = '1' then
       ...  accoes associadas ao "Fim de Microprograma"
    end if;
    if AFull = '1' then
       ...  accoes associadas a "Stack quase cheia"
    end if;
    if SC_Req = '1' then
       ...  accoes associadas a "operacao Escalar"
    end if;
    if SStep = '1' then
       ...  accoes associadas ao "Single Step"
    end if;
    if SDone = '1' then
       ...  associadas ao "Swap Done do VSR"
    end if;
    if Suspende = '1' then
       ...  accoes associadas ao "Pedido de suspensao"
    end if;
    WAIT ON SDone, SStep, SC_Req, AFull, Finish, Equal;
  END process spool;
END behav_spool;
```

[5]Embora o transputer disponha de um mecanismo de interrupção, não foi considerada a sua utilização atendendo à complexidade adicional que daí resultaria e à (esperada) baixa frequência de utilização deste recurso.

De salientar que a definição da lista de sensibilidades através da instrução `WAIT ON ...`, no final do processo e com a forma utilizada, é rigorosamente equivalente à sua definição no cabeçalho do processo. Optou-se pela primeira forma, por permitir incluir uma cláusula condicional ou temporal, se isso for desejável em futuras revisões.

Outro aspecto a salientar diz respeito ao evento gerado pelo próprio HT para aceder ao sistema GLiTCH - através do sinal **BusReq**. No sistema real, esta facilidade estará associada a um pedido de um nível hierárquico superior ao próprio HT, com vista a aceder à informação interna do sistema GLiTCH. O VHDL permite simular este tipo de operação de dois modos. Num primeiro modo, existe um sinal associado a um **port**, que é externamente controlado pelo utilizador no decorrer de uma simulação, como o é qualquer outro sinal global ao sistema. No segundo modo, é declarado um sinal local ao módulo, neste caso o HT, que será também controlado pelo utilizador no decorrer de uma simulação. Este último modo só pode ser implementado se o utilizador puder aceder às variáveis internas dos componentes do modelo, facilidade esta habitualmente disponível. Foi esta a solução adoptada no modelo do HT, e o sinal localmente declarado é identificado por **Suspend**, na listagem anterior.

Implementação do processador hospedeiro

A implementação do processador hospedeiro corresponde a uma realização em *hardware* com capacidade para executar os modelos anteriormente descritos, e por esse motivo não existe uma equivalência entre os modelos e o circuito sintetizado. Dito de outra forma, a concepção dos modelos e o projecto da arquitectura do processador hospedeiro foram tarefas independentes. A arquitectura do processador hospedeiro baseia-se num transputer T800, com 256K ou 1M bytes de memória dinâmica, um bloco de lógica para interface com os barramentos partilhados e um conjunto de registos de estado e controlo - o respectivo diagrama esquemático encontra-se no apêndice B.

Na sua maior parte, a lógica auxiliar foi implementada em componentes PLD (PAL16V8 e PAL22V10), nomeadamente o desmultiplexador do barramento do T800, o multiplexador de endereços para a memória dinâmica, os descodificadores de endereços e os registos de controlo e de estado. No registo de estado foi necessário implementar algumas máquinas de estado, para sincronizar os sinais provenientes das restantes unidades funcionais do sistema GLiTCH, e que utilizam um *clock* diferente do utilizado pelo processador hospedeiro. As descrições em PALASM associadas a estes componentes de suporte ao processador hospedeiro, assim como dos respectivos vectores de teste, encontram-se no apêndice C.

4.9 Modelo estrutural do controlador em VHDL

De uma forma resumida, o que foi descrito até este ponto corresponde à aplicação de uma metodologia de desenvolvimento habitual, assente numa filosofia *top-down*, através da qual o sistema foi dividido em blocos e cada bloco foi descrito funcionalmente numa HDL. No caso do sequenciador e das memórias, uma vez que a selecção de componentes foi feita *a priori*, a descrição funcional seguiu uma filosofia *bottom-up*. A coexistência destas duas filosofias de desenho é perfeitamente normal num projecto desta natureza.

Para poder simular o modelo é ainda necessário interligar todos os componentes. A interligação está associada ao conceito de estrutura e, como tal, deve-se passar do domínio comportamental para o domínio estrutural. Em termos de VHDL, uma descrição estrutural assenta no conceito de *component instantiation* (ver, por exemplo, [Hsu, et al. 1995]). Qualquer modelo cujo interface externo foi previamente descrito, através do construtor **entity**, pode ser utilizada como um componente (construtor **component**) dentro de um modelo estrutural. Uma configuração associada selecciona a descrição da arquitectura (eventualmente podem existir várias) para a simulação, e uma sua instância materializa as respectivas interligações.

O modelo do sistema em si tem que ter associado um par de construtores **entity / architecture**, com características particulares, o qual é vulgarmente designado por *test bench* - atendendo à sua função de teste de um modelo. O construtor **entity** define o interface externo, que a este nível hierárquico não existe, enquanto que o construtor **architecture** define a arquitectura estrutural do sistema. A listagem seguinte é um fragmento da descrição estrutural do controlador do sistema GLiTCH (a listagem completa encontra-se no apêndice A), e apenas detalha os aspectos de implementação relacionados com a micromemória.

```
entity test_bench is
end test_bench;

architecture Sistema_GLiTCH of test_bench is
  component MicroMemoria
  port (endereco: in mm_end;
        dado: inout mm_instr;
        notcs, notrd, notwr: in bit);
  end component;
  for U1:Micromemoria use entity MM(BEHAVIOR);
  ...  component Sequenciador
  ...  component AddGenLogic
  ...  component DataStore
  ...  component ScalarUnit
  ...  component TagRouter
  ...  component HostTransputer
  ...  component BusArbiter
  ...  Declaracoes de sinais internos

  begin
    ...
    U1:Micromemoria
      port map(mm_in, mm_out, mm_notcs, mm_notrd, mm_notwr);
    ...  U2:Sequenciador
    ...  U3:AddGenLogic
    ...  U4:DataStore
    ...  U5:ScalarUnit
    ...  U6:TagRouter
    ...  U7:HostTransputer
    ...  U8:BusArbiter
end Sistema_GLiTCH;
```

É importante realçar a equivalência que existe entre esta descrição em VHDL e a descrição esquemática da figura 4.1. Ambas estão ao mesmo nível de abstracção e apenas correspondem a formas (domínios) de expressão diferentes. Pode-se assim concluir que existe uma duplicação de representações a este nível, apesar de ambas serem necessárias - o diagrama esquemático na tarefa da divisão do desenho inicial em blocos e a

descrição VHDL na integração dos componentes num único modelo simulável. O diagrama esquemático, após algumas alterações, vai ainda servir para produzir a placa de circuito impresso, utilizando um programa de CAD apropriado.

Atendendo às múltiplas utilizações referidas, seria desejável manter a coerência entre as descrições esquemática e em VHDL, sendo útil dispor de alguma ferramenta automática que apoie a sua produção, manutenção e verificação. O *software* de CAD utilizado na realização física do controlador integra aquelas funções, sendo possível construir um único diagrama esquemático, atribuir uma descrição em VHDL a cada um dos blocos estruturais e exportar o projecto para o simulador, assim como para um outro sistema especializado no desenho de circuitos impressos. No entanto, no que respeita à simulação, esse sistema de CAD impõe restrições de utilização que dificultam a análise, ao nível de abstracção desejado. Esta limitação impediu a utilização conjunta (análise e implementação) dos modelos estruturais realizados, sendo a simulação executada noutra ferramenta de CAD - o MINT.

O anexo A contém o a descrição completa do modelo do controlador em VHDL, tal como foi definido, e o anexo B contém a documentação produzida para a construção do respectivo circuito impresso. Neste último conjunto de esquemáticos, a maior complexidade está associada ao *hardware* do módulo do processador hospedeiro (recorde-se que o modelo VHDL apenas descreve a sua função no sistema) e o interface com a placa de circuito impresso do GLiTCH. Este interface foi implementado utilizando como suporte físico o barramento VME, redefinindo os sinais disponíveis às necessidades do interface a implementar. Essa redefinição está detalhadamente documentada em [Santos and Proença 1993].

4.9.1 Utilização do modelo estrutural na simulação

A construção dos modelos em VHDL tem dois objectivos. Ao nível do componente, procura-se explorar um método seguro de síntese automática, enquanto que ao nível do sistema, a preocupação central é a verificação do funcionamento integrado dos diversos componentes. A coexistência destes dois objectivos é necessária para garantir a integridade do projecto como um todo e exige um equilíbrio entre o nível de detalhe necessário, imposto pelas descrições para síntese e a consequente degradação do desempenho do modelo simulável. Neste contexto, desempenho não significa apenas o tempo de resposta, mas também a dificuldade em verificar um conjunto numeroso de variáveis e sinais, resultante do nível de detalhe. As facilidades de interface implementadas pelo ambiente de simulação podem minorar este efeito.

A solução para este tipo de problema será gradualmente introduzida pelos sucessivos progressos das ferramentas de HLS, que permitirão fazer descrições para síntese a um nível de abstracção cada vez mais elevado.

No caso do controlador do GLiTCH, procurou-se conciliar os dois objectivos num único modelo, através de restrições à funcionalidade do sistema e consequente diminuição da complexidade do modelo. Isto significa sacrificar a exactidão do modelo simulável do sistema, face à especificação inicial, e centrar o maior esforço de desenvolvimento na síntese automática ao nível dos blocos constituinte. Apesar dessa opção, os resultados obtidos da simulação do modelo estrutural ficaram aquém do que era desejável, pois a

complexidade do processo de simulação não permitiu simular/verificar um conjunto satisfatório de vectores de teste. Dito de outra forma, o esforço associado à construção deste modelo, exclusivamente para simulação, e à análise dos resultados da simulação, é demasiado elevado face à funcionalidade que é possível verificar (quer pela dificuldade, quer pelas restrições impostas).

Uma solução alternativa passaria pela elaboração de dois modelos distintos: um para simulação, a um nível de abstracção mais elevado; e outro para síntese, mais detalhado, incluindo os modelos utilizados na síntese. Os principais obstáculos desta metodologia são a dificuldade em garantir a equivalência entre os modelos e o maior tempo de desenvolvimento em que necessariamente se incorre.

O modelo do controlador do GLiTCH foi verificado com versões reduzidas dos programas utilizados para o teste do sistema real (ver apêndice D), devidamente adaptados às restrições impostas à funcionalidade do modelo.

4.10 Estrutura de *pipeline*

Por forma a assegurar o funcionamento do sistema GLiTCH à frequência mínima de 20MHz, foi necessário introduzir vários registos de *pipeline*. Muitos deles são internos às diversas unidades funcionais, tal como foi explicitado nas descrições anteriores, mas outros há cuja necessidade apenas se torna evidente aquando da integração. Com efeito, para quaisquer duas unidades funcionais a interligar, sempre que a soma dos respectivos tempos de atraso (e de eventuais tempos de *setup* e de *hold*) não for inferior ao período do *clock* (para a frequência mínima), então a interligação necessita de um registo de *pipeline*. Nesta situação encontra-se o sequenciador, para o qual foi necessário incluir dois registos de *pipeline*: um para a instrução e outro para as *flags*.

Para além das características temporais, no caso do sistema GLiTCH um outro objectivo esteve subjacente ao projecto da estrutura de *pipeline*: assegurar que cada operação elementar do sistema GLiTCH seja realizada por uma única microinstrução, por forma a facilitar a programação. Neste contexto, uma operação elementar é caracterizada por exigir que as diversas unidades funcionais envolvidas executem apenas uma instrução. Como exemplo, refira-se a operação de **Match** [Duller, et al. 1989b], para a qual tem que ser executada uma instrução de leitura do DS, uma instrução de encaminhamento do DRN (ver descrição no capítulo 2) e a instrução de *match*, propriamente dita, do GLiTCH. Por forma a que o objectivo enunciado seja satisfeito para a maioria das operações elementares, é necessário introduzir um registo de *pipeline* à saída da micromemória, para todos os sinais que vão ligar ao módulo GLiTCH.

Nem todas as operações elementares evidenciam o mesmo fluxo de informação. As operações que produzam *flags* a ser utilizadas pelo sequenciador, ou aquelas que envolvem a leitura de um valor da CAM do GLiTCH e o armazenamento desse valor no DS ou na SU, evidenciam um fluxo de informação inverso ao da operação **Match**, atrás descrita. Estas operações, para além de exigirem a dispersão das instruções individuais (das unidades funcionais envolvidas) por diversas microinstruções, são ainda penalizadas pelo aumento do atraso resultante da introdução de níveis de *pipeline*. Na prática, esta

penalização é compensada pela baixa frequência de ocorrência destas operações e o ganho em facilidade de programação para a maioria das operações.

À semelhança do que acontece noutras arquitecturas em *pipeline*, um dos maiores problemas é a manutenção da coerência do funcionamento quando é interrompido o modo de execução sequencial. Existem já algumas soluções de *hardware* para minimizar este efeito mas, no caso do sistema GLiTCH, e por forma a não aumentar a complexidade do protótipo, foi decidido transferir essas responsabilidade para a ferramenta de geração de microcódigo.

Um último aspecto a referir em relação aos registos de *pipeline*, diz respeito à possibilidade de suspender a operação do sistema GLiTCH, quando o HT pretende assumir o controlo. Nesta situação, é necessário garantir que o estado dos registos de *pipeline* permaneça inalterado, durante o período de suspensão. Para garantir esse efeito, as unidades funcionais anteriormente descritas dispõem de um mecanismo de realimentação dos seus registos. O mesmo não se verifica com os componentes (*standards*) utilizados como registos de *pipeline* externos, para os quais foi necessário usar como *clock* um sinal especifico, com as mesmas características temporais do *clock* do sistema, mas que permanece inactivo durante o período de suspensão - este sinal foi igualmente utilizado num dos registos de *pipeline* da SU.

4.11 Finalização do projecto de *hardware*

O controlador do GLiTCH é um dos blocos constituintes de uma arquitectura heterogénea, proposta para executar aplicações de visão por computador em "tempo real". Nesta perspectiva, a sequência de desenvolvimento do projecto passaria agora pela integração dos diferentes blocos e a validação da arquitectura, pelo menos para o conjunto de algoritmos que foram inicialmente simulados e que, em última análise, justificaram o projecto. Devido a diversas dificuldades, a equipa de projecto responsável pelo desenho do GLiTCH não conseguiu executar essa tarefa em tempo útil, comprometendo assim a execução global do projecto. Na tentativa de ultrapassar essa dificuldade, foi então proposto um projecto complementar para o desenvolvimento de um emulador do GLiTCH.

O GLiTCH é um circuito bastante complexo (origem das principais dificuldades enunciadas) e, por conseguinte, o seu emulador seria necessariamente um sistema igualmente complexo, e ainda dispendioso em termos de recursos humanos e materiais. Como forma de amortizar o custo associado, foi imposto como um dos objectivos primordiais a reutilização do sistema, o qual passou assim a evidenciar as propriedades de um emulador genérico, sendo o projecto designado por EdGAR[6]. Na prossecução deste objectivo, a lógica reprogramável foi a tecnologia de implementação escolhida, mais precisamente componentes CPLD e FPGA, dado o seu nível de integração e a disponibilidade de ferramentas de desenvolvimento.

[6] Emulador Genérico Altamente Reprogramável

Como inconveniente, refira-se que a utilização de componentes CPLD e FPGA no EdGAR, comprometeu a emulação exacta do GLiTCH[7], a qual exigiria recursos de *hardware* formidáveis, bem além do que seria aceitável para um projecto desta natureza. Para ultrapassar esta dificuldade, foi decidido construir o emulador com duas componentes complementares. Uma componente de *hardware*, para emular os blocos do GLiTCH cujo desempenho é crítico - como por exemplo o interface com um barramento de vídeo externo - e uma componente de *software*, para emular os blocos menos críticos e cuja implementação em *hardware* é difícil - como por exemplo a memória associativa.

Na perspectiva da sua aplicação genérica, o projecto do EdGAR (bem como do seu sistema de desenvolvimento) pode introduzir inovações interessantes em duas disciplinas particularmente activas em termos de investigação, prototipagem rápida/*codesign* e HLS.

4.12 Interface com o compilador

Os testes ao circuito final do controlador do GLiTCH foram executados pelo transputer hospedeiro, segundo a metodologia anteriormente descrita. Para poder executar esses testes, foi necessário desenvolver alguns utilitário, entre os quais um *Assembler* e um *Loader/Debugger*. O *Assembler* converte um ficheiro de texto com as mnemónicas associadas às instruções de cada unidade funcional e as directivas necessárias para definir a estrutura do microprograma. O *Loader/Debugger* é o programa que permite carregar microprogramas (código e dados) e controlar a sua execução, utilizando as facilidades existentes no *hardware* - execução passo-a-passo e com pontos de paragem [Petr 1993].

Atendendo à natureza destes utilitários, é possível (e desejável) que eles venham a ser utilizados no contexto mais vasto de um sistema de desenvolvimento, para um gerador de código, como *software* de interface com o controlador. Deste modo, na sua especificação, um dos aspectos mais importantes foi a definição da estrutura de um microprograma, a qual está detalhada na figura 4.12.

A definição do cabeçalho demonstra a preocupação em flexibilizar a geração de código, ao suportar um mecanismo dinâmico para determinar o endereço destino, quer do microprograma, quer do bloco de dados. O "Tipo de microprograma" pode assumir três valores (0 - microprograma principal; 1 - subrotina; e 2 - fragmento de código) e apesar de não ser utilizado nas ferramentas desenvolvidas, pode vir a ser útil na classificação do código, a um nível mais elevado.

De notar que, na "imagem" do microprograma, cada palavra dispõe de mais 32 bits do que os 96 necessários para definir uma microinstrução. Esses 32 bits adicionais são utilizados para dois fins:

- Armazenar o número da linha de um ficheiro. O campo respectivo será activado pelo *Assembler* e é utilizado pelo *Loader/Debugger* para mostrar a linha de ficheiro de texto que deu origem à microinstrução, quando ela estiver em análise.

[7] De qualquer forma também não existe um especificação temporal exacta do GLiTCH, dada a inexistência de uma implementação física.

- Permitir que o endereço fornecido como operando de uma instrução do sequenciador ou (exclusivo) do AGL - ver organização da micromemória na figura 4.8 - seja utilizado num modo absoluto ou relativo, neste último caso relativamente ao início do respectivo bloco, dentro do microprograma.

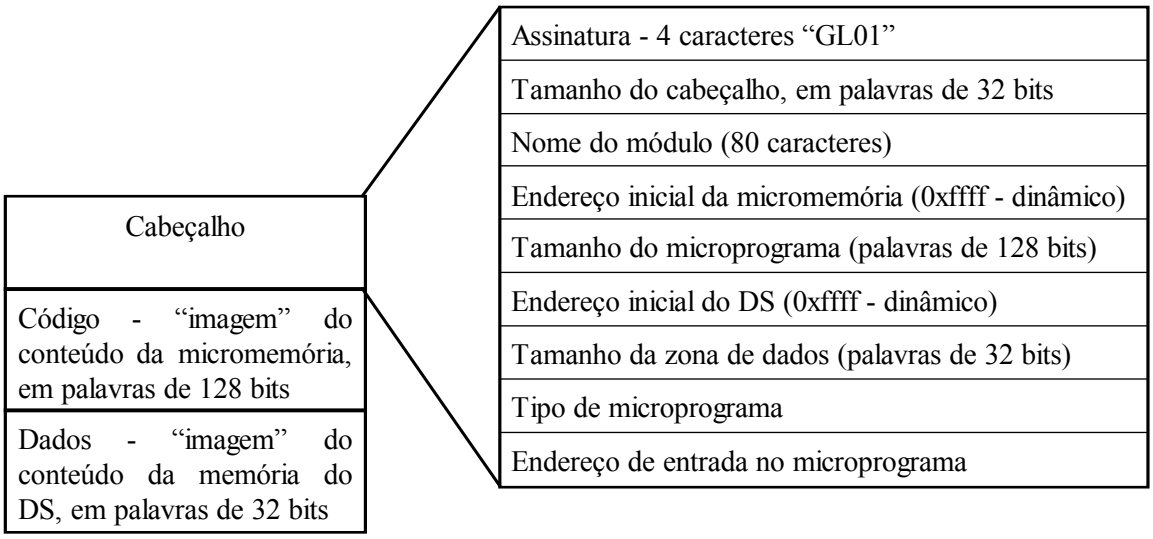


Figura 4.12 Estrutura de um microprograma

Apesar de não ter sido possível integrar todo o sistema e validar a realização da arquitectura proposta, na definição da linguagem *Assembly* incluíram-se todos os campos correspondentes aos comandos para o *array* de GLiTCHs e respectivos componentes de suporte. Na versão actual, o *Assembler* não gera o respectivo código, mas a sua actualização futura está substancialmente simplificada. A definição da sintaxe da linguagem *Assembly* do controlador do GLiTCH, assim como a descrição detalhada dos utilitários desenvolvidos, encontra-se em [Petr 1993].

Capítulo 5

5. Estratégias de implementação com lógica programável

5.1 Motivação

A utilização da lógica programável no projecto de sistemas digitais está hoje amplamente divulgada, o que pode levar a concluir que os seus problemas de aplicação estão já numa etapa madura do ciclo de resolução. Contudo, essa convicção deve ser questionada, face à constante evolução tecnológica, que introduz novos componentes com estruturas mais complexas e novos paradigmas para o processo de desenvolvimento. Na prática esta realidade é responsável pelo aparecimento de novos problemas para resolver.

Na fase de implementação do controlador do GLiTCH foi possível identificar alguns desses problemas, nomeadamente:

- ⇒ a selecção do tipo de lógica programável mais adequada a uma determinada especificação; e
- ⇒ a exploração do espaço de projecto inerente ao processo de síntese automática, a partir das descrições em VHDL; nesta tarefa é ainda possível destinguir dois subproblemas:
  - o impacto da variação sintáctica; e
  - a estratégia de optimização, que consiste na escolha dos parâmetros de optimização a utilizar na síntese, em função das características da descrição.

Deve ser salientado que o objectivo subjacente à resolução destes problemas é a optimização de recursos. Atendendo a que nos componentes de lógica programável a quantidade de recursos é limitada, a questão da optimização assume então particular relevância.

Estes problemas são comuns a qualquer outro projecto de um sistema digital - e em particular ao desenvolvimento do EdGAR - na necessária definição de estratégias para síntese automática em lógica programável. Com esse objectivo e aproveitando o projecto do controlador do GLiTCH - uma aplicação real com alguns circuitos típicos - foi executado um estudo sobre os problemas acima identificadas. O estudo baseou-se em circuitos já descritos - arbitrador de barramentos (BA), unidade escalar (SU) e lógica para geração de endereços (AGL) destinados à memória de padrões - o primeiro representativo da classe das máquinas de estado e os restantes, com níveis de complexidade diferentes entre si, representativos da classe de componentes que incluem essencialmente uma unidade de processamento de dados associada a uma unidade de controlo simples. Atendendo aos objectivos do estudo a encetar, na selecção destes circuitos teve-se ainda em conta que apenas interessa considerar aqueles cujo nível de complexidade permita a sua implementação em lógica programável.

5.2 Metodologia e parâmetros de avaliação

Dada a natureza dos problemas em estudo, a experimentação é actualmente o único método disponível para explorar o espaço das soluções. No entanto, convém realçar que o método experimental evidencia uma limitação importante nesta aplicação: a dependência dos resultados relativamente às ferramentas de síntese utilizadas. Ao nível da optimização lógica este problema não é tão grave, atendendo à maturidade dos respectivos algoritmos, No que respeita à síntese de alto nível o mesmo não se verifica, já que muitos dos algoritmos usados estão eles próprios ainda numa fase experimental. Por este motivo, os resultados obtidos evidenciam por vezes alguma inconsistência, e não podem ser simplesmente generalizados, demonstrando, contudo, um comportamento tipo, o qual pode ser facilmente verificado noutros ambientes de desenvolvimento.

Seguindo um percurso de desenvolvimento normal, e atendendo à complementaridade dos estudos, o primeiro a ser efectuado foi o da exploração do espaço de projecto. De uma forma resumida, este estudo foi efectuado sintetizando cada um dos circuitos com as várias combinações dos parâmetros de configuração da ferramenta de síntese (estratégia de optimização).

O estudo da variação sintáctica procurou avaliar o impacto que tem sobre a síntese algumas descrições alternativas permitidas pela linguagem VHDL, dentro dos limites impostos pela ferramenta de síntese. Neste contexto não é relevante a tecnologia de implementação, nem o tipo de circuito. Dos circuitos disponíveis foi escolhido aquele cujo leque de descrições alternativas é mais vasto, e que por esse motivo mais se presta a este estudo - o arbitrador de barramentos (BA). Para cada uma das descrições alternativas foi aplicado o processo atrás definido para o estudo da estratégia de optimização. No que respeita à variação sintáctica, os resultados obtidos podem ser generalizados, reforçando assim a opção de dispensar o alargamento deste estudo aos restantes circuitos.

Nas experiências com as diversas descrições do BA, muitos dos resultados apresentaram variações mínimas, pelo que se achou pertinente avaliar o impacto dessas variações numa implementação concreta. Com esse objectivo, o melhor resultado de cada uma das descrições alternativas foi utilizado para implementação num componente do tipo FPGA. Esta experiência permitiu ainda identificar algumas relações entre os resultados (estimados) fornecidos pela ferramenta de síntese e os recursos físicos ocupados, para o componente considerado.

Para cada um dos três circuitos em análise, a melhor solução obtida através da exploração do espaço de projecto, foi utilizada para o segundo estudo, a análise da sua adequação às estruturas lógicas programáveis de dois componentes equivalentes, representativos das classes em estudo (CPLD e FPGA).

No que respeita à análise de resultados, é importante reter que o objectivo do estudo é avaliar os resultados da optimização de recursos lógicos, em função de um dado conjunto de factores. Qualquer métrica para avaliar os recursos lógicos deve incluir, no mínimo, a área e o tempo de atraso, mas os tipos de resultados e a forma como podem ser utilizados dependem, obviamente, da ferramenta de síntese. Neste estudo são utilizados dois tipos de ferramentas, um para cada um dos problemas em estudo. As características conjuntas de

cada um desses problemas e das respectivas ferramentas de síntese exigem metodologias diferentes para abordar a questão da avaliação de resultados. De uma forma resumida:

⇒ Exploração do espaço de projecto. A ênfase é colocada na eficiência da ferramenta de síntese (configurável) ao operar descrições algorítmicas. A análise de resultados será facilitada se a síntese for independente de qualquer tecnologia de implementação, já que as bibliotecas para tecnologias específicas incluem frequentemente soluções optimizadas para determinadas situações, o que poderia viciar os resultados.

⇒ Avaliação da adequação do tipo de lógica programável às características duma especificação. O processo implica a utilização de ferramentas de mapeamento de especificações estruturais (*netlist*) nas estruturas lógicas programáveis, mas a ênfase é colocada na adequação dessas estruturas lógicas. Nesta perspectiva é necessário medir e comparar os recursos lógicos ocupados (efectivamente utilizados e inutilizados).

5.2.1 Exploração do espaço de projecto

A ferramenta de síntese utilizada (ViewSynthesis, caracterizada no capítulo 3) aceita descrições em VHDL e explora sobretudo a optimização ao nível *Logic*, o que de certa forma está adequado ao nível de complexidade dos circuitos em consideração e à capacidade efectiva dos componentes de lógica programável disponíveis. A desejada independência da tecnologia de implementação pode ser garantida utilizando na síntese uma biblioteca de componentes genéricos, designada por DIO. Esta biblioteca contém diversas células lógicas elementares de 2 e 4 entradas, *flip-flops* do tipo 'D' e *latches*, com diversas combinações de sinais assíncronos de controlo para as operações de *set* e *reset*.

Os parâmetros de configuração da ferramenta de síntese são:

- o tipo de lógica a sintetizar, que pode assumir os valores *small*, *large*, ou *mixed*, consoante a descrição está associada, respectivamente, a um circuito predominantemente sequencial, predominantemente combinatorial, ou misto; e
- a prioridade pretendida entre área e a velocidade de operação, parâmetro designado por A/S, ao qual pode ser atribuído qualquer valor inteiro entre 1 (prioridade absoluta à área) e 100 (prioridade absoluta à velocidade de operação).

Relativamente ao parâmetro A/S, para a maior parte dos casos considerados, os resultados preliminares demonstraram ser suficiente utilizar apenas os valores 1, 50 e 100, dividindo assim o intervalo em dois subintervalos. Desta forma, considerando todas as possíveis combinações dos dois parâmetros, a partir de cada descrição foi possível obter nove resultados da síntese para analisar. É pertinente salientar ainda que não foram impostas nenhumas restrições iniciais aos sinais, quer em termos dos tempos de atraso quer em termos dos factores de carga.

A utilização de uma biblioteca de síntese genérica implica a indisponibilidade de características físicas dos elementos lógicos usados. Desta forma, os resultados obtidos são expressos por estimativas - uma implementação física conduzirá necessariamente a resultados diferentes - o que é perfeitamente aceitável em termos de um estudo

comparativo entre resultados de síntese, com ênfase na ferramenta de desenvolvimento e não na tecnologia de implementação. Tendo em conta este pormenor, os resultados relevantes que a ferramenta de síntese fornece são: uma lista onde estão descriminadas todas as células lógicas utilizadas (da biblioteca DIO); e um sumário sobre a *netlist* gerada.

O sumário contém o número total de células, uma estimativa para a área ocupada por essas células, o número de ligações e o número máximo de níveis de lógica. Para efeitos da análise de resultados os dois primeiros valores estão estritamente relacionados, pelo que o número total de células pode ser ignorado. Assim, a estimativa para a área ocupada pelas células e o número de ligações exprimem a área total do circuito, enquanto que o número de níveis de lógica constitui o indicador mais próximo da velocidade de operação. A figura 5.1 ilustra como esses resultados vão ser mostrados nas tabelas utilizadas ao longo deste capítulo.

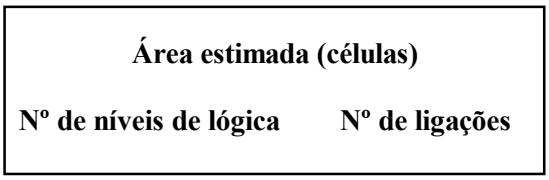


Figura 5.1 Representação dos resultados da síntese

5.2.2 Selecção do tipo de lógica programável (CPLD vs. FPGA)

Neste estudo vão ser analisados os resultados da síntese relativos a implementações físicas em dois tipos de estruturas de lógica programável. Esses resultados são expressos pela velocidade de operação do circuito final e os recursos lógicos ocupados - que de certa forma constitui uma medida da área, mas em termos de componentes de lógica programável. A velocidade de operação é fácil de obter, sendo fornecida directa ou indirectamente pela ferramenta de desenvolvimento. O mesmo não se verifica para a avaliação dos recursos lógicos, cuja análise comparativa requer mais cuidado. Com efeito, à excepção do número de *flip-flops*, a dimensão dos recursos lógicos existentes nos componentes CPLD e nos componentes FPGA não é directamente relacionável (a descrição destas estruturas encontra-se na secção 3.2).

Em vez de procurar relacionar as capacidades (à semelhança do que é feito no estudo idêntico efectuado por Small e já referido na secção 3.2), neste estudo procurou-se utilizar parâmetros mais significativos do ponto de vista do projecto, como sejam o número de pinos e o custo associado à sua utilização. Com efeito, o número de pinos define o espaço ocupado pelo circuito e, do ponto de vista do utilizador, dois componentes de lógica programável que ocupem o mesmo espaço e tenham um custo semelhante deverão ser aproximadamente equivalentes. Um pressuposto deste principio é a existência (habitual), para cada uma das famílias em consideração, de um leque equivalente de componentes, com características temporais diferentes. Assim, por exemplo, os componentes MACH120 (56 pinos) e FPGA XC2018PC68 (68 pinos), cujo custo é aproximado, serão considerados equivalentes, e o que evidenciar melhor aproveitamento relativamente aos seus recursos específicos será aquele que melhor se adaptará ao problema em questão. Este princípio foi

utilizado na selecção dos componentes para implementar cada uma das aplicações em estudo.

No que respeita aos recursos individuais, os componentes FPGA são avaliados pela percentagem de CLBs e de I/OBs utilizados, enquanto que os componentes CPLD são avaliados pela percentagem de macrocélulas, células de I/O e PTs/*clusters* utilizados. Atendendo à maior granularidade dos componentes CPLD, é importante neste caso avaliar ainda o factor de desaproveitamento de PTs, que resulta da utilização parcial de *clusters*. Um elevado factor de desaproveitamento demonstra a inadequação da própria estrutura lógica ao problema e, na prática, resulta numa solução dispendiosa (independentemente das alternativas o poderem ser também).

Atendendo à complementaridade dos estudos efectuados, era desejável (mas não necessário) que a descrição a utilizar para a realização física (*netlist*) fosse a resultante do anterior estudo do processo de síntese automática. Para os componentes FPGA as ferramentas de desenvolvimento disponíveis permitem essa integração, o mesmo não se verificando para os componentes CPLD. Para estes foi necessário traduzir a especificação VHDL para PALASM. A riqueza semântica da linguagem VHDL é muito superior ao da linguagem PALASM e por conseguinte os resultados poderiam ser falseados. Na prática este problema não surgiu porque o nível de abstracção da descrição em VHDL não é muito elevado, o que resulta das restrições impostas pela ferramenta de síntese. De notar que, em alternativa, poderia ainda ter-se optado pela tradução da *netlist* resultante da síntese automática, em vez da tradução manual da especificação VHDL que lhe deu origem. Essa tarefa pode ser facilmente automatizada e será importante no contexto do sistema de desenvolvimento para o emulador genérico, com vista à automatização do processo de síntese com componentes CPLD.

5.3 Estudo da implementação do arbitrador de barramentos

No contexto deste estudo, o BA pode ser caracterizado como um circuito com sete máquinas de estado concorrentes - uma com cinco estados, uma com quatro estados e as restantes cinco com dois estados - oito saídas síncronas e seis entradas (ignorando os sinais de *reset* e *clock*).

Segundo a metodologia apresentada, em seguida são descritos os estudos da exploração do espaço de projecto e da adequação das estruturas lógicas programáveis, por esta ordem. No estudo da exploração do espaço de projecto foram consideradas cinco descrições alternativas em VHDL, para assim avaliar igualmente o efeito da variação sintáctica. A implementação de cada uma dessas alternativas num componente FPGA permitiu determinar o significado prático das diferenças entre os resultados obtidos.

Para o estudo da adequação das estruturas lógicas programáveis foi efectuada uma implementação num componente CPLD. Esta tarefa exigiu a construção de uma descrição em PALASM. Os resultados obtidos são comparados com uma das implementações no componente FPGA anteriormente obtidas, aquela cujas características da descrição em VHDL mais se aproximam das da descrição em PALASM.

5.3.1 Exploração do espaço de projecto e variação sintáctica

De uma forma sintética, as cinco descrições alternativas em VHDL são as seguintes:

⇒ Alternativa 1 - descrição utilizada no modelo simulável do protótipo (secção 4.7), baseada num processo VHDL, com construtores "CASE" ou "IF…THEN…ELSIF…ELSE" e sem a codificação de estados explícita.

⇒ Alternativa 2 - idêntica à alternativa 1, mas com a codificação de estado segundo a técnica "*1-hot*".

⇒ Alternativa 3 - idêntica à alternativa 1, mas com as atribuições condicionais completas, isto é, para todos os sinais cujo valor depende de uma condição é explicitamente indicado o valor a atribuir quando a condição é verdadeira e quando é falsa.

⇒ Alternativa 4 - idêntica à alternativa 3, mas utiliza dois processos, um para descrever a sequência de estados e outro para descrever o comportamento das saídas.

⇒ Alternativa 5 - utiliza funções específicas para síntese, e o seu objectivo é apenas avaliar o *overhead* associado à síntese automática.

Na maioria dos casos, a análise dos resultados obtidos numa dada descrição está na base das opções tomadas para a descrição seguinte. Por este motivo, os detalhes mais relevantes de cada uma das descrições serão discutidos ao longo da apresentação do estudo.

Para todas as descrições são experimentadas as diferentes estratégias de optimização, associadas às diferentes combinações dos parâmetros de configuração da ferramenta de síntese.

Alternativa 1

Esta descrição é a que se situa a um nível mais elevado de abstracção. Sendo um exemplo da classe das máquinas de estado e atendendo às facilidades concedidas pela linguagem VHDL, as principais características dessa descrição são as seguintes:

A) os estados são representados por conjuntos enumerados (não é explicitada a codificação de estados);

B) para cada uma das máquinas de estado é utilizado um único processo para descrever a sequência de estados e o comportamento das saídas; e

C) esses processos baseiam-se essencialmente no construtor "CASE".

Numa variante desta descrição procurou-se avaliar o efeito das características A) e C), explicitando a codificação de estado (recorrendo à técnica da distância unitária entre estados adjacentes) e substituindo o construtor "CASE" por hierarquias de construtores "IF…THEN….ELSIF…ELSE". Os resultados da síntese não diferiram significativamente entre estas duas descrições, o que demonstra a equivalência entre os construtores, em termos de síntese, e a eficiência da ferramenta na escolha dos códigos de estado. Embora esta conclusão seja importante, para efeitos da comparação entre alternativas, estas duas descrições foram consideradas equivalentes, estando os resultados da síntese resumidos na tabela 5.1. Esta implementação utiliza 21 *flip-flops*, dos quais apenas 10 implementam variáveis de estado.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	485 14 175	416 8 143	689 7 312
50	483 14 169	390 8 126	390 8 126
1	485 19 168	392 8 127	392 8 127

Tabela 5.1 Resultados da síntese do BA (alternativa 1)

De uma forma geral, os melhores desempenhos da síntese para este caso ocorrem quando o tipo de lógica é *mixed* ou *small*, o que seria de esperar atendendo ao tipo de circuito em questão. Por esse mesmo motivo não é de estranhar os resultados obtidos quando se utiliza o tipo de lógica *large* - as técnicas de optimização associadas a este parâmetro não estão adequadas às características do circuito.

Em particular nos casos em que o tipo de lógica é *mixed* e *small*, a tendência global para aumentar a área quando se dá prioridade à velocidade (A/S = 100) é um resultado igualmente esperado, não o sendo, contudo, a insensibilidade do número de níveis. Essa insensibilidade resulta do leque reduzido de elementos lógicos existentes na biblioteca de síntese (portas de 2 e 4 entradas), que associado à simplicidade das funções lógicas limita as opções da síntese.

Alternativa 2

Nesta alternativa manteve-se a estrutura da solução e apenas se utilizou uma codificação de estados particular - habitualmente designada por "1-*hot*" - nas máquinas de estado de 5 e 4 estados. Esta codificação consiste em associar um *flip-flop* a cada estado, o que exige um maior número de *flip-flops*, mas, tipicamente, conduz a expressões lógicas mais simples para as funções de próximo estado e de saída. Os resultados da síntese estão resumidos na tabela 5.2. Esta solução utiliza 24 *flip-flops*, dos quais apenas 13 são utilizados para implementar variáveis de estado.

Os aspectos mais relevantes são a diminuição do número de níveis, associada à diminuição da complexidade das funções lógicas, e a melhor adequação dos algoritmos de optimização associados ao tipo de lógica *large* - mas mesmo assim bastante inadequados. De notar ainda que a diminuição da complexidade não tem reflexos relevantes em termos de área, já que esta é afectada pelo aumento do número de *flip-flops*, os quais têm um peso significativo em termos de área.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	538 8 189	423 5 129	424 5 130
50	446 6 142	399 4 112	399 4 112
1	463 7 150	399 4 112	399 4 112

Tabela 5.2 Resultados da síntese do BA (alternativa 2)

Aspectos críticos dos resultados da síntese das alternativas 1 e 2

Uma inspecção à *netlist* resultante da síntese das alternativas anteriores, revelou a existência de lógica desnecessária, associada a atribuições condicionais em que não é explicitado o comportamento quando a respectiva condição não é verificada (de futuro as atribuições que evidenciem esta característica serão designadas por atribuições condicionais incompletas). Em particular refira-se a situação criada pela seguinte secção da descrição VHDL do BA, que define as funções lógicas associadas a duas saídas distintas, mas que na prática evidenciam o mesmo comportamento:

⇒ **stF=F(stF, BusTrnotLoc1aux, Finish, ClrFlags)** - variável de estado cujo valor é definido por um conjunto enumerado (NoFinish, FinishRun)

⇒ **FinishOut= F(stF, BusTrnotLoc1aux, Finish, ClrFlags)** - saída de estado cujo valor é do tipo BIT (‘0’, ‘1’).

```
    if (stF = NoFinish) and (BusTrnotLoc1aux = '0') and (Finish = '1')
then
       stF <= FinishRun; FinishOut <= '1';
      elsif (stF = FinishRun) then
         if (ClrFlags = '1') then
            stF <= NoFinsh; FinishOut <= '0';
         end if;
    end if;
```

Na síntese os elementos do conjunto enumerado têm que ser codificados em binário, o que corresponde à atribuição de estados. Se for associado o valor ‘0’ ao estado representado por “NoFinish” e o valor ‘1’ ao estado representado por “FinishRun”, o comportamento dos dois sinais é idêntico, caso contrário um será a simples negação do outro. Este tipo de dependência não é detectada pela ferramenta de síntese, que em vez disso ainda encobre essa relação através da simplificação. As expressões lógicas resultante da síntese daquela descrição são as seguintes (considerando a atribuição de estado acima indicada):

1) `/stF * /BusTrnotLoc1aux * Finish +`
   `stF * /ClrFlags`
2) `/stF * /BusTrnotLoc1aux * Finish +`
   `/(stF * ClrFlags) * FinishOut`

Com base na descrição, o segundo termo de cada uma destas expressões foi incluído, por decisão da ferramenta de síntese, para garantir que o valor das saídas não se altera no

caso da condição **ClrFlags = '1'** não de verificar, o que obrigou a recorrer a um elemento de memória para a saída "FinishOut". De notar que a forma diferente desses dois termos resulta da simplificação de **/(stF * ClrFlags) * stF**, na expressão 1), e que, a menos desses dois termos, as equações são idênticas. Olhando para a descrição, é possível admitir que este efeito lateral não era desejado para a saída "FinishOut". De notar que a correcção deste tipo de pormenor poderia passar simplesmente pela existência de um parâmetro que indicasse à ferramenta de síntese o estado que deveria atribuir às saídas associadas a atribuições condicionais incompletas.

Alternativa 3

Com base na observação anterior foi construída uma terceira descrição alternativa em VHDL, na qual foram eliminadas todas as atribuições condicionais incompletas - as correspondentes saídas assumem o valor lógico '0' ou '1', conforme se aplique. Este procedimento deverá reduzir essencialmente o número de realimentações. Os resultados da síntese estão resumidos na tabela 5.3. À semelhança da alternativa 1, e apesar das expectativas que a análise anterior poderia criar, esta implementação utiliza os mesmos 21 *flip-flops*.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	557 15 210	320 5 91	320 5 91
50	531 13 199	317 6 89	317 6 89
1	545 17 205	318 7 89	318 7 89

Tabela 5.3 Resultados da síntese do BA (alternativa 3)

Comparando com os resultados da tabela 5.1, obtidos na síntese da alternativa 1, esta alternativa produziu um circuito mais rápido e com uma área sensivelmente mais baixa. Esta melhoria resulta essencialmente da remoção de algumas realimentações e consequente diminuição da complexidade das funções lógicas, já que o número de *flip-flops* permaneceu inalterado.

O facto de utilizar um único processo para descrever a sequência de estados e o comportamento das saídas obriga a implementar as saídas através de *flip-flops*, à semelhança das variáveis de estado, que necessariamente o têm que ser. Este facto resulta de uma restrição da ferramenta de síntese (comum à maioria das ferramentas), que obriga a diferenciar processos associados a lógica combinatória e a lógica sequencial.

Alternativa 4

A utilização de processos distintos para as variáveis de estado e para as saídas poderá então reduzir o número de *flip-flops* e a complexidade do circuito sintetizado. Este facto esteve na base da construção de uma quarta descrição alternativa em VHDL, cujos

resultados da síntese estão resumidos na tabela 5.4. Esta solução utiliza apenas 10 *flip-flops*, sendo de realçar ainda que nesta alternativa também não foram utilizadas atribuições condicionais incompletas.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	438 15 168	202 6 76	202 6 76
50	349 11 121	199 7 74	199 7 74
1	369 16 130	198 7 72	198 7 72

Tabela 5.4 Resultados da síntese do BA (alternativa 4)

Antes de analisar os resultados convém realçar que no circuito associado a este tipo de descrição é possível o aparecimento de ruído nas saídas, resultante do facto de elas passarem a ser implementadas no modo imediato. Este é um pormenor acerca do qual o projectista está normalmente consciente e que, em última análise, pode ser verificado pela simulação (o que aconteceu neste estudo).

Comparando com os resultados da alternativa 3, neste caso obtiveram-se melhores resultados para a área - como consequência da diminuição do número de *flip-flops* - não sendo, contudo, muito significativa a variação do número de níveis - o que significa que o número de níveis não é imposto pelas funções de saída.

Com base nas conclusões anteriores, seria de esperar que, pelo menos para o tipo de lógica *mixed* e *small*, o número de níveis tivesse uma evolução muito semelhante à da alternativa 3. Tal não se verifica devido às características heurísticas dos algoritmos de síntese, que conduzem a soluções diferentes pela simples alteração da estrutura de um processo ou até mesmo da ordem em que os sinais são declarados numa descrição. Aliás, a execução da ferramenta de síntese sobre um circuito previamente optimizado, com as mesmas opções de configuração, conduz, normalmente, a um resultado diferente e não necessariamente melhor. Pelo exposto, pode-se concluir que a diferença do número de níveis entre as duas últimas alternativas resulta apenas das características da ferramenta de síntese e não da complexidade ou de qualquer outra características das funções lógicas.

Alternativa 5

A última alternativa a considerar utiliza uma descrição de mais baixo nível, em *data-flow* e recorre a primitivas específicas da ferramenta, como seja uma tabela tipo PLA, que corresponde a uma descrição através de uma tabela de verdade. Este tipo de descrição não deveria ser à partida utilizado, mas o seu objectivo é apenas avaliar o *overhead* associado a descrições de alto nível - à semelhança do que acontece no domínio do *software*. Os resultados desta alternativa estão resumidos na tabela 5.5. Tal como para a

alternativa 1 e 3, esta solução utiliza 21 *flip-flops*, uma vez que as saídas não foram realizadas no modo imediato.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	211 5 74	206 5 71	206 5 71
50	206 5 71	203 5 69	203 5 69
1	206 5 70	203 5 68	203 5 68

Tabela 5.5 Resultados da síntese do BA (alternativa 5)

No que respeita à área total, deve-se ter em atenção que apenas é legítimo comparar estes resultados com os resultados das alternativas 1,2 e 3, uma vez que na alternativa 4 é utilizado um número muito reduzido de *flip-flops*, os quais, como já foi realçado, têm um peso importante na área total. Desta forma, verifica-se uma redução acentuada na área, o que reflecte, para as outras alternativas, um *overhead* elevado associado à utilização de um mecanismo de síntese automático. Esta questão é particularmente importante no contexto da utilização da lógica programável, atendendo às limitações em termos dos recursos lógicos disponíveis, e traduz-se na necessidade da continuação da investigação nesta área.

Outro aspecto a salientar da análise dos resultados da tabela 5.5 é a insensibilidade à variação dos parâmetros, a qual se deve à "rigidez" semântica da especificação - tabela de verdade com a indicação explícita de todos os estados *don't care* - e consequente limitação das opções de síntese, limitada apenas à simplificação lógica.

5.3.2 Análise de resultados

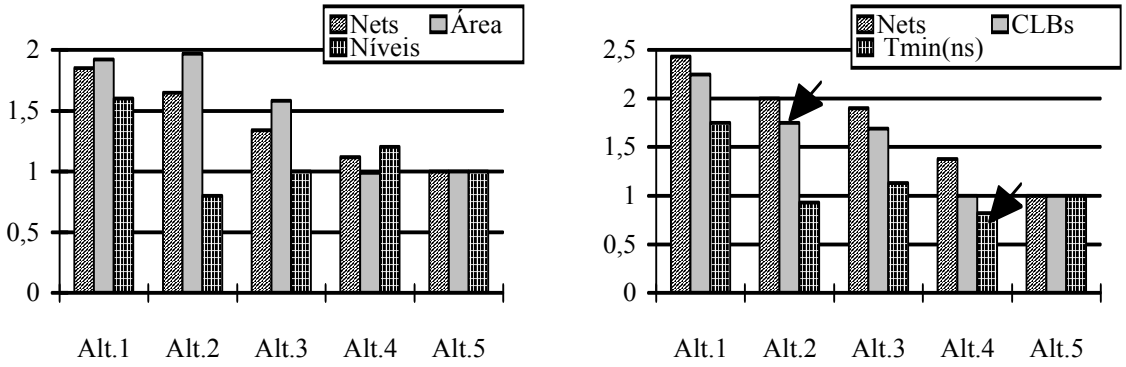
A figura 5.2a) mostra num gráfico de barras os melhores resultados obtidos para cada alternativa, tendo como primeiro critério de selecção o número de níveis e depois a área. Por forma a permitir a visualização conjunta e a facilitar a comparação, os valores apresentados estão normalizados relativamente aos valores obtidos para a alternativa 5. Esta opção justifica-se atendendo às características e ao objectivo da alternativa 5, que a definem como uma referência legítima para o estudo em questão.

No contexto da variação sintáctica apenas interessa comparar as alternativas 1, 2, 3 e 4. Com base na figura 5.2a) podem-se salientar as seguintes conclusões:

A) A alternativa 2 produziu a solução mais rápida.
B) A alternativa 4 produziu a solução com menor área.
C) A alternativa 3 representa um compromisso entre as duas posições extremas das alternativas 2 e 4.
D) A alternativa 1 não evidencia qualquer vantagem sobre as restantes alternativas, pelo menos atendendo às variáveis em análise.

Estes resultados dependem essencialmente da ferramenta de síntese, e podem servir como indicadores na escolha de uma estratégia de modelação. As conclusões A) e B) aplicam-se apenas a máquinas de estado, enquanto que a conclusão C) é relevante para qualquer descrição com atribuições condicionais.

Atendendo a que estes resultados foram obtidos com uma biblioteca de síntese genérica, será útil analisar o significado das variações em termos de uma implementação real. Para isso todas aquelas soluções foram implementadas num FPGA XC2018-125, estando os respectivos resultados resumidos no gráfico da figura 5.2b). Estes resultados são expressos em termos do número de CLBs utilizados, o número de ligações (entre CLBs e entre estes e os pinos) e o período mínimo (Tmin.) associado à máxima frequência de operação. Tal como no gráfico da figura 5.2a) e pelas razões então apontadas, também no gráfico da figura 5.2b) os valores apresentados estão normalizados relativamente aos valores obtidos para a alternativa 5.



a) Biblioteca de síntese genérica b) Implementação física

Figura 5.2 Resumo dos resultados da síntese do BA

De notar que se existisse uma relação directa entre os resultados obtidos na síntese com a biblioteca genérica e os da implementação física, os gráficos das figuras 5.2a) e 5.2b) seriam idênticos. Essa relação seria desejável na resolução de um dos problemas da síntese de alto nível, que é a geração de estimativas precisas e fiáveis para as soluções, a partir da descrição de alto nível. Nesta perspectiva, é útil procurar identificar as razões pelas quais a implementação física evidencia desvios face ao que seriam os valores estimados. Com base nos gráficos da figura 5.2, os principais desvios são os seguintes:

- na alternativa 2 verifica-se uma redução de cerca de 23% no número de CLBs;
- na alternativa 4 verifica-se um aumento de cerca de 63% na frequência máxima; e
- face à alternativa 5 e de uma forma geral, a implementação física amplifica as diferenças verificadas na síntese na biblioteca genérica, sobretudo no que respeita ao número de ligações.

O desvio verificado na alternativa 2 resulta da diminuição da complexidade das funções lógicas, associada à técnica de atribuição de estados utilizada. Deve-se salientar ainda que esta redução ocorre apesar do aumento do número de *flip-flops*, o que pode

sugerir um sobreavaliação da estimativa da área dos *flip-flops*, por parte da ferramenta de síntese, e considerando a sua implementação no componente em questão. A redução da área é fruto da compactação de várias células em cada CLB, o que provoca igualmente a ligeira redução relativa das ligações. O factor de compactação depende muito da especificação e será mais elevado se os *flip-flops* existentes nos CLB forem completamente utilizados, atendendo ao peso a que esse tipo de célula tem na estimativa da área.

O desvio verificado na alternativa 4 ilustra a importância dos *flip-flops* no tempo de resposta do circuito. Com efeito, no componente utilizado cada *flip-flop* implica a utilização de um CLB, e quanto maior o número de CLBs maior será o tempo de atraso nas suas interligações. Isto justifica o aumento de desempenho obtido, apesar de na alternativa 4 a síntese com a biblioteca genérica produzir um número elevado de níveis de lógica.

A degradação da frequência por cada nível adicional introduzido pela síntese de alto nível depende essencialmente do componente e do factor de compactação, o qual poderá permitir a redução de níveis no mapeamento. Este tipo de conclusão será útil no contexto do sistema de desenvolvimento associado ao EdGAR. Atendendo aos motivos que estão na origem dos desvios verificados nas alternativas 2 e 4, a comparação dos gráficos da figura 5.2a) e b) sugere que será possível prever os efeitos da síntese de alto nível no circuito final, em função das características do problema e do componente a utilizar. No entanto, a validação destas conclusões exige um estudo mais alargado.

5.3.3 Implementação em CPLD e FPGA

Para efectuar este estudo foi utilizada a descrição da alternativa 4. A implementação no FPGA foi feita a partir da *netlist* gerada pela ferramenta de síntese, com o parâmetro A/S = 100 e a selecção do tipo de lógica s*mall*. Os resultados foram já utilizados na secção anterior, estando incluídos no gráfico da figura 5.2b) - coluna Alt. 4.

A implementação em CPLD foi efectuada utilizando um componente MACH120-12, o qual, seguindo o critério previamente estabelecido para a selecção de componentes, é equivalente ao XC2018-125, utilizado na implementação em FPGA. Para implementar o BA num CPLD foi necessário traduzir as descrições VHDL para PALASM. Tal como já foi referido, as capacidades expressivas das duas linguagens são bastante diferentes. No entanto, considerando a complexidade do BA e o nível de abstracção utilizado na descrição em VHDL para alternativa 4, essas diferenças não são relevantes, resumindo-se às seguintes regras impostas pelo PALASM:

- o utilizador tem que indicar explicitamente os sinais que utilizam *flip-flops* (essa informação é extraída da descrição no caso do VHDL); e
- por defeito é assumido que os sinais modificados de uma forma condicional são *don't care*, nas condições para as quais o valor não é explicitado (no caso da ferramenta de síntese para VHDL, os sinais nessas condições mantêm o seu valor).

Saliente-se ainda que, dado o elevado grânulo do bloco lógico básico dos CPLD, é previsível que a variação sintáctica não tenha o impacto que se observou para os FPGA. Essa previsão não pode ser demonstrada, dada a rigidez semântica da única linguagem disponível para os CPLD, o PALASM.

Para efeitos comparativos, a tabela 5.6 resume os resultados da síntese nos componentes CPLD e FPGA.

XC2018PC68-130

	Disponível	Utilizado	%
CLBs	100	16	16
I/O Blocks	64	16	25
Ligações	---	29	---

Fmáx.: 23.1 MHz

MACH120-12

	Disponível	Utilizado	%
Macrocélulas	48	13	27
Células de I/O	48	13	27
Pinos	56	21	37
PTs | Clusters	192 | 48	30 | 14	16 | 30

Fmáx.: 50 MHz

Tabela 5.6 Resumo da síntese do BA em componentes FPGA e CPLD

Como se pode observar, a utilização de recursos é ligeiramente melhor no componente FPGA, mas o maior destaque é o melhor desempenho temporal obtido com o componente CPLD, o qual está ainda associado a um processo de desenvolvimento mais rápido - vantagens que naturalmente se esperam de uma estrutura lógica a dois níveis, com um grânulo elevado. O factor de desaproveitamento no CPLD pode ser observado relacionando a utilização de PTs e de *clusters*, e constitui o único aspecto negativo da respectiva implementação. Com efeito, os 14 *clusters* utilizados correspondem a 56 PTs, dos quais apenas 30 foram efectivamente utilizados. Esta situação conduz a um factor de desaproveitamento de cerca de 46%, anormalmente elevado, e resulta da existência de vários sinais com apenas um ou dois produtos, sobretudo aqueles associados às 4 máquinas com apenas 2 estados. De salientar que este factor de desaproveitamento justifica a diferença na utilização de recursos para os dois componentes.

5.4 Estudo da implementação da unidade escalar

A unidade escalar (SU) é um circuito composto por um *data path* - basicamente constituído por um registo de deslocamento de 16 bits - e um *control path* muito simples - inclui um registo de instrução com três bits e um descodificador - dispondo cada um desses blocos de um sinal de *clock* independente (a descrição mais detalhada foi apresentada anteriormente). No contexto do estudo em consideração, a descrição em VHDL tem as seguintes características:

- dois processos com *clock*, associados a cada um dos blocos acima referidos; e
- um processo de lógica combinatória, que combina todas as saídas do registo de deslocamento e assinala a ocorrência do valor zero;

5.4.1 Síntese da descrição VHDL

Os resultados obtidos na síntese da SU estão resumidos na tabela 5.7. Atendendo às características do circuito, e comparando com os resultados obtidos na síntese do BA, verifica-se a melhor adequação dos algoritmos quando o tipo de lógica é *large*. No entanto,

os melhores resultados da síntese verificam-se quando o tipo de lógica seleccionado é *mixed*, o que pode sugerir que a componente combinatorial deste circuito ainda não é suficientemente elevada para produzir uma melhor optimização com o tipo de lógica *large*.

Outro aspecto importante é a insensibilidade do número níveis à variação do parâmetro A/S, quando o tipo de lógica é *mixed* ou *small*. Este tipo de comportamento já foi anteriormente detectado na síntese do BA, e tal como aí foi referido, o principal factor a influenciar esse resultado é o leque restrito de componentes disponíveis na biblioteca de síntese, que limita as opções dos algoritmos de síntese. De qualquer forma é notória a penalização da área quando se procura optimizar para velocidade.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	633 7 293	628 6 274	775 7 354
50	522 8 215	527 6 218	531 7 220
1	527 9 217	519 6 214	521 7 215

Tabela 5.7 Resultados da síntese da SU

5.4.2 Implementação em CPLD e FPGA

O componente FPGA utilizado nesta implementação pertence à família XC3000, já que os da família XC2000 não dispõem dos recursos lógicos suficientes (de notar que em termos de complexidade a SU exige praticamente o dobro dos recursos exigidos pelo BA). De uma forma semelhante, o componente CPLD utilizado pertence à família MACH2, cujas características são próximas das da família XC3000. Os resultados das síntese estão resumidos na tabela 5.8.

Da comparação dos resultados salienta-se claramente o melhor desempenho da implementação no componente CPLD, quer em termos da máxima frequência de operação, quer em termos dos recursos lógicos ocupados. Para o CPLD é de registar ainda o elevado factor de desaproveitamento, que neste caso atinge os 30% - os 30 *clusters* utilizados correspondem a 120 PTs, dos quais apenas 85 (70%) foram efectivamente utilizados.

XC3020PC68-125

	Disponível	Utilizado	%
CLBs	64	58	91
I/O Blocks	58	44	77
Ligações	---	119	---

Fmáx.: *control path* 22.1 MHz
data path 19.2 MHz

MACH221-10

	Disponível	Utilizado	%
Macrocélulas	96	30	31
Células de I/O	48	26	54
Pinos	56	49	88
PTs | Clusters	384 | 96	85 | 30	22 | 31

Fmáx.: *control path* 100 MHz
data path 100 MHz

Tabela 5.8 Resumo da síntese da SU em componentes FPGA e CPLD

Fora do âmbito da comparação entre implementações, e em particular para a implementação no componente CPLD, os valores da tabela 5.8 podem ser também utilizados para inferir a adequação do componente escolhido ao problema. Com efeito, se for comparada a utilização de pinos (88%) com a utilização dos restantes recursos, é notória a discrepância, podendo-se concluir que no componente seleccionado existe um excesso de recursos disponíveis, face ao nível de complexidade do circuito sintetizado. Esta análise sugere que, para a SU, dever-se-ia utilizar um componente com o mesmo número de pinos, mas com menor quantidade de recursos lógicos, o que na prática representaria uma redução no custo do circuito.

5.5 Estudo da implementação da AGL

Do ponto de vista estrutural, a lógica para geração de endereços (AGL) da memória de padrões, corresponde ao mesmo tipo de circuito da SU - um circuito onde predomina o *data path* sobre o *control path*. A principal diferença reside na complexidade da lógica combinatória, que é maior para a AGL. No contexto deste estudo, as características mais relevantes da descrição VHDL da AGL são as seguintes (a descrição mais detalhada foi apresentada anteriormente):

- registo de 13 bits para endereços, com a possibilidade de carregamento paralelo e suporte das operações de incrementação e decrementação;
- lógica para geração de sinais de controlo da memória (DS);
- 21 sinais de entrada e 17 sinais de saída; e
- descrição comportamental através de 1 processo e algumas instruções de atribuição condicionais concorrentes.

5.5.1 Síntese da descrição VHDL

Os resultados da síntese obtidos para a AGL estão resumidos na tabela 5.9. O aumento da complexidade relativamente à SU é facilmente observável pelo aumento da área e do número de níveis.

A/S	*Large* (*Data-path*)	*Mixed*	*Small* (FSM)
100	639 17 268	650 17 282	840 9 359
50	600 17 262	565 17 247	612 15 269
1	599 17 261	567 17 248	614 16 270

Tabela 5.9 Resultados da síntese da AGL

Quanto ao número de níveis, o melhor resultado foi obtido com o tipo de lógica *small*, e isto apesar das características da AGL, que permitem classificar este circuito como predominantemente combinatorial. Este resultado contraria os resultados até aqui obtidos para este parâmetro, e revela a necessidade de alargar este estudo a outras descrições, por forma a poder identificar as causas deste comportamento. No entanto, existe uma diferença importante entre a SU e a AGL, que é o número de variáveis das funções lógicas, a qual pode estar na origem deste resultado. Na SU o *data path* é essencialmente constituído por um registo de deslocamento, pelo que a maioria das funções lógicas utiliza um número reduzido de variáveis, enquanto que na AGL o *data path* é constituído por um pequeno módulo aritmético, com capacidade para incrementar e decrementar o valor contido num registo, o que necessariamente conduz a funções lógicas com um maior número de variáveis. Para além disso, esta última estrutura evidência mais possibilidades de optimização factorizando mais ou menos as funções lógicas.

De notar que o mesmo tipo de variação no número de níveis - mas com resultados globalmente piores - foi igualmente obtido na síntese da maioria das descrições alternativas do BA, nesses casos com tipo de lógica *large* e, portanto, teoricamente também desajustado às características do circuito.

Outro resultado a realçar é o comportamento da ferramenta de síntese quando o tipo o tipo de lógica é *large*. Com efeito, as discrepâncias anteriormente verificadas não são já claramente evidentes, o que deixa antever que este circuito encontra-se na fronteira do que deve ser considerado um circuito predominantemente combinatorial, em termos do parâmetro tipo de lógica.

5.5.2 Implementação em CPLD e FPGA

No que respeita à síntese no componente FPGA, de acordo com a metodologia adoptada, a implementação deveria ser realizada com base na síntese obtida com o tipo de lógica *small* e A/S=100, uma vez que esta conduziria à solução mais rápida. No entanto, a penalização dessa solução em termos de área é demasiado elevada, exigindo a utilização de um componente com mais recursos. À partida este facto conduziria necessariamente a uma depreciação da utilização de recursos, que só se justificaria se houvesse idêntica necessidade com a implementação no componente CPLD, o que não se verificou. Este facto levou a utilizar para implementação no componente FPGA, o resultado da síntese com o tipo de lógica *small* e A/S=50.

Os componentes utilizados na síntese da AGL são os mesmo que foram utilizados na síntese da SU. Os resultados obtidos estão resumidos na tabela 5.10

XC3020PC68-125

	Disponível	Utilizado	%
CLBs	64	54	89
I/O Blocks	58	38	70
Ligações	---	106	---

Fmáx.: 9.1 MHz

MACH221-10

	Disponível	Utilizado	%
Macrocélulas	96	21	22
Células de I/O	48	18	38
Pinos	56	39	70
PTs | Clusters	384 | 96	169 | 51	44 | 53

Fmáx.: ≈31 MHz

Tabela 5.10 Resumo da síntese da AGL em componentes FPGA e CPLD

O componente CPLD evidência novamente um desempenho superior, tanto em termos de recursos ocupados, como em termos da frequência máxima de operação. No entanto, relativamente à frequência, e comparando com os resultados obtidos para a SU, a diferença não é tão acentuada, deixando antever que o aumento de complexidade do circuito aproximaria as duas soluções.

O factor de desaproveitamento do CPLD neste caso é apenas 17% - os 51 *clusters* ocupados correspondem a 204 PTs, dos quais apenas 169 (cerca de 83%) forma efectivamente utilizados. Este resultado decorre da maior complexidade das funções lógicas implementadas, que permitem um menor desperdício de recursos, mas que são simultaneamente responsáveis pela redução da frequência de operação. De notar ainda que, à semelhança do que se verificou para a implementação da SU, também neste caso a diferença que se verifica entre a utilização de pinos e a utilização dos recursos lógicos sugere que se deve considerar a utilização de um componente CPLD com menor volume de recursos lógicos.

5.6 Sumário

Os resultados obtidos com o estudo da variação sintáctica foram anteriormente apresentados, pelo que nesta secção se abordará apenas a questão da exploração do espaço de projecto e da adequação do tipo de lógica programável, face às características das descrições em VHDL.

Exploração do espaço de projecto

Tal como foi referido anteriormente, este estudo analisa apenas o comportamento de uma particular ferramenta de síntese, cuja principal função é a optimização ao nível *Logic*. O utilizador pode influenciar esse comportamento através de dois parâmetros: tipo de lógica e A/S - prioridade relativa entre área e velocidade de operação do circuito sintetizado. Os resultados obtidos são independentes de qualquer tecnologia, o que foi garantido através de uma biblioteca de síntese genérica.

Relativamente ao tipo de lógica, dos três valores possíveis apenas dois conduziram a resultados óptimos - *mixed* e *small* - o que significa que em nenhum dos circuitos experimentados existia uma predominância de lógica combinatória, em termos das referências da ferramenta de síntese. Na maioria dos casos aqueles dois valores produziram resultados muito idênticos.

Na síntese do BA, os resultados obtidos com o tipo de lógica *large* - teoricamente não adaptado às características da descrição - evidenciaram um comportamento inadequado, ao produzir circuitos com um número de níveis exagerado. Inversamente, na síntese da AGL, e com o tipo de lógica *small* - também teoricamente inadequado às características da descrição - os resultados demonstraram um comportamento semelhante, mas desta vez conduzindo à solução mais rápida, à custa de uma elevada penalização em termos de área. Esta questão em particular necessita de um estudo mais alargado.

Relativamente ao parâmetro A/S, era esperado que a ferramenta de síntese produzisse um circuito mais lento e com menor área, quando se dá prioridade à área, ou um circuito mais rápido e com maior área, quando se dá prioridade à velocidade de operação. Quanto à área as expectativas verificaram-se, o mesmo não sendo verdade para o número de níveis, o qual, na maioria dos casos, permaneceu inalterado ou evidenciou ligeiras alterações - exceptuando os casos atrás referidos em que o tipo de lógica é, teoricamente, inadequado. Este resultado está relacionado com o leque restrito de elementos lógicos existentes na biblioteca de síntese utilizada, o qual limita as opções da ferramenta de síntese.

Adequação do tipo de lógica programável

Os resultados obtidos demonstraram uma melhor adequação do componente CPLD, face ao componente FPGA, tanto em termos de velocidade de operação - circuitos 3 a 5 vezes mais rápidos - como da utilização de recursos lógicos. Isto significa que, considerando o nível de complexidade e as características das descrições utilizadas, a estrutura lógica a dois níveis dos CPLD produz melhores resultados práticos do que a estrutura de lógica distribuída - maior número de células com menor capacidade - dos FPGA.

Como seria de esperar, o maior grânulo das células lógicas dos CPLD podem produzir factores de desaproveitamento mais elevados, ao utilizar parcialmente uma célula complexa na implementação de uma função simples. Na prática aquele factor variou entre 46% na implementação do BA, e 17% na implementação da AGL. No primeiro caso, comparando com a implementação no FPGA, a utilização de recursos lógicos foi superior no CPLD, enquanto que no segundo caso foi inferior. Isto demonstra existir uma relação entre estes dois indicadores, podendo o factor de desaproveitamento ser utilizado como uma medida indirecta da utilização de recursos. Esta conclusão pode ser relevante no contexto de uma selecção automática, necessitando, contudo, de ser alargada a mais famílias de componentes, assim como experimentada num maior número de descrições.

Finalmente, deve ser referido que nos FPGA é possível obter soluções mais eficientes, utilizando técnicas manuais ou semi-manuais de mapeamento e encaminhamento de sinais, permitidas pela respectiva ferramenta de implementação. No contexto deste estudo optou-se por não recorrer a essa metodologia, por duas razões: não é

previsível que essas metodologias se venham a enquadrar num processo de síntese automática, como aquele que é preconizado para o sistema de desenvolvimento associado ao EdGAR; e essas metodologias podem introduzir atrasos significativos no tempo de desenvolvimento de um projecto, já que para além do tempo exigido pela sua execução, podem igualmente dar origem à introdução de erros humanos.

Capítulo 6

6. Conclusões

O GLiTCH é um circuito constituído por um *array* de processadores com memória associativa, projectado para executar essencialmente algoritmos de processamento de imagem, integrado num sistema heterogéneo para visão por computador, que incluirá, ao nível mais elevado, uma rede de transputers. Do ponto de vista da arquitectura, o GLiTCH inclui apenas um *data path*, necessitando assim de uma unidade de controlo - situação bastante vulgar neste domínio de aplicação. Para avaliar o desempenho do GLiTCH foi desenvolvido um simulador, o qual, para além de validar vários dos pressupostos subjacentes à especificação do próprio GLiTCH, demonstrou algumas insuficiências resultantes da sua natureza associativa, nomeadamente quanto à execução de operações escalares. Para ultrapassar esses problemas, e ainda como consequência dos requisitos exigidos pela execução das operações associativas, foram incluídos diversos módulos na especificação da unidade de controlo, a qual constitui assim um exemplo de um sistema digital complexo. O tema central desta tese foi o desenvolvimento de um protótipo dessa unidade de controlo, utilizando novos paradigmas do projecto de sistemas digitais, com o objectivo de acelerar o processo de prototipagem, flexibilizar a solução e promover a testabilidade.

Nos últimos anos tem-se verificado algumas alterações na metodologia de desenvolvimento de projectos de sistemas digitais. A metodologia tradicional, designada por *capture-and-simulate* - com ênfase na utilização de descrições estruturais - tem dado lugar a uma outra, habitualmente designada por *describe-and-synthesize* - com ênfase na utilização de descrições funcionais e subsequente síntese automática [Gajski e Ramachandran 1994]. Esta transformação decorreu no contexto da evolução tecnológica, com a necessidade de utilizar níveis de abstracção mais elevados na descrição de sistemas cada vez mais complexos, assim como domínios de representação mais adaptados ao processamento automático. Este novo paradigma de desenvolvimento deu origem a uma nova disciplina, designada por *High-Level Synthesis* (HLS), que estuda as questões relacionadas com os modelos de representação e os algoritmos necessários para automatizar a realização de circuitos [Camposano 1990].

As linguagens de descrição de *hardware* encontraram nesta nova metodologia de desenvolvimento uma área de aplicação importante. Entra essas linguagens, a VHDL aparece como uma das mais divulgadas, não só pelas suas capacidades de representação a vários níveis de abstracção, mas também (e principalmente) pela sua normalização. Muito embora tenha sido inicialmente proposta para documentação de circuitos e como meio de comunicação entre projectistas, as suas aplicações na simulação e síntese foram rapidamente investigadas e deram origem a várias ferramentas, hoje perfeitamente estabelecidas [Dewey e Geus 1992].

A efectiva utilização de técnicas de HLS exige uma mudança de atitude na especificação dos projectos. Com efeito, essas técnicas exigem descrições funcionais, i. e.,

descrições que especifiquem o que o sistema deve fazer e não como o deve fazer. Esta característica não é verificada na maioria das especificações, as quais incluem frequentemente vários pressupostos de implementação, limitando assim (ou mesmo impedindo) a possibilidade de exploração de alternativas por parte das ferramentas de HLS.

A especificação inicial do controlador do GLiTCH é um exemplo do tipo de especificação limitada, uma vez que resulta já de uma proposta de solução para executar determinados algoritmos de processamento de imagem, de uma forma optimizada. Por outro lado, deve-se realçar que não é fácil conceber uma especificação que seja puramente funcional, já que a sua utilidade dependeria da existência de um processo automático com todo o conhecimento acumulado sobre alternativas de implementação. Na situação mais vulgar, o projectista define algumas estratégias de implementação, normalmente com vista à optimização do desempenho, deixando para o processo automático o cuidado de escolher determinados detalhes da realização física, segundo algum critério, e respeitando o conjunto das restrições impostas pela própria estratégia. Na prática o resultado é a coexistência, numa mesma especificação, de vários módulos descritos com diferentes propósitos e a vários níveis de abstracção, o que dificulta o desenvolvimento e utilização de um modelo para essa especificação.

No caso do controlador do GLiTCH, a especificação inicial propunha já uma divisão por blocos, cuja funcionalidade estava condicionada pelo pressuposto de diferentes tipos de soluções. Um dos blocos correspondia a um processador escalar, outros podiam ser implementadas por componentes *standard*, e outras ainda necessitavam de *hardware* específico. Segundo esta divisão foram caracterizados três tipos de modelos: modelo sintetizável, modelo de um componente *standard* e modelo de *software*. O primeiro foi utilizado para simulação e síntese, enquanto que os restantes apenas foram relevantes na construção de um modelo estrutural do circuito completo, para simulação. Por forma a poder lidar com a complexidade deste modelo estrutural, foi necessário introduzir várias restrições à funcionalidade, o que por si só limitou a utilidade desse modelo. Mesmo assim a sua utilidade ficou aquém das expectativas, demonstrando ser necessário dispor de ferramentas mais evoluídas para apoio à construção de vectores de teste e de verificação de resultados. De qualquer forma, a metodologia proposta para a construção dos diversos modelos em VHDL pode ser aplicada em qualquer outro projecto.

A selecção das tecnologias de implementação foi conduzida pelas prioridades previamente estabelecidas: rapidez na obtenção do protótipo e flexibilidade da solução. Estas prioridades foram satisfeitas através da utilização de componentes *standard* e de componentes (re)programáveis pelo utilizador. Relativamente a este último tipo de componentes, apesar de no protótipo terem sido utilizados componentes de reduzida complexidade - por questões circunstanciais - a evolução verificada permite hoje considerá-los como alternativa para a implementação de ASICs. Nessa perspectiva mereceram especial atenção as duas classes designadas por CPLD e FPGA, por incluírem componentes com elevadas escalas de integração e estruturas programáveis adaptadas a diferentes tipos de circuitos.

As questões da testabilidade do circuito final foram igualmente consideradas desde o início do projecto. A possibilidade de incluir uma estrutura lógica dedicada para teste,

baseada no conceito de *Boundary Scan* [Abramovici, et al. 1990], não foi considerada devido às limitações em termos de espaço físico e à existência de diversos componentes que não estavam adaptados a essa filosofia de teste. Em vez disso, foi definida uma estratégia baseada na utilização do processador escalar disponível - o transputer - cujas facilidades para esse fim facilitaram significativamente a tarefa. Em especial refira-se o método de "arranque" do transputer, através do qual é possível verificar o funcionamento do circuito em teste a partir de um outro transputer instalado num sistema onde executam as ferramentas de desenvolvimento. Do ponto de vista do *hardware*, apenas é necessário garantir que os componentes a testar estão ligados ao barramento do transputer existente no circuito em teste [Hinton e Pinder 1993]. O acesso é feito instrução-a-instrução, facilitando assim a detecção de erros ao nível do interface.

Após a construção do protótipo da unidade de controlo do GLiTCH, utilizando como *case study* alguns dos circuitos desenvolvidos (ou mais propriamente as suas descrições em VHDL), foi feito um estudo sobre os problemas da variação sintáctica - sensibilidade da implementação às alternativas de descrição permitidas pela linguagem - e da adaptação dos componentes CPLD e FPGA às características das descrições. Estes dois problemas são pertinentes na aplicação das técnicas de HLS, tendo como tecnologia alvo aquelas classes de componentes. Relativamente à variação sintáctica, os resultados permitiram identificar a interpretação semântica de alguns tipos de descrições e a forma de controlar a relação velocidade/área de um circuito, quer através da descrição, quer através dos parâmetros de configuração da ferramenta de síntese utilizada. É importante salientar que estes resultados apenas são válidos para a ferramenta utilizada.

No estudo da adaptação dos componentes CPLD e FPGA às características das descrições, é necessário inicialmente definir um critério de equivalência entre componentes. Uma das soluções é procurar quantificar os recursos disponíveis em cada componente, como no estudo realizado por Small [Small, et al. 1993]. Neste caso o critério utilizado baseou-se em características de utilização, nomeadamente o número de pinos e o custo. Os resultados do estudo demonstram o melhor desempenho dos CPLD. No entanto, deve ser considerado que nenhuma das descrições necessita de um número elevado de células de memória, características esta frequentemente apontada como essencial para o bom desempenho dos FPGA. Desta forma, a generalização dos resultados obtidos carece do alargamento do estudo a mais alguns circuitos.

6.1 Trabalho futuro

Seguindo uma sequência natural de desenvolvimento, o trabalho que foi descrito deveria culminar com a integração dos diversos subsistemas que constituem a arquitectura heterogénea inicialmente proposta, designada por HVA, e validar a sua aplicação em visão por computador. No entanto, esta finalização do projecto ficou comprometida com as dificuldades sentidas na realização do circuito integrado que implementa o GLiTCH e cujo projecto esteve sob a responsabilidade do grupo de trabalho da Universidade de Bristol, UK. Após reconhecer essas dificuldades, associadas às metodologias de projecto utilizadas, e recorrendo ao conhecimento adquirido na utilização da lógica programável, o grupo de trabalho da Universidade do Minho iniciou um novo projecto para um emulador genérico - EdGAR - cuja primeira aplicação seria a emulação do GLiTCH. Dessa forma,

quando este projecto estiver concluído, será possível implementar o sistema HVA e finalmente validar os princípios que levaram à sua definição - isto sem obstar a que seja concluído o circuito integrado que implementa o GLiTCH, ou um outro que o substitua.

Por necessidades na emulação do GLiTCH e também como forma de promover a sua generalidade, o EdGAR inclui um módulo de lógica reprogramável (CPLD e FPGA) e um módulo de execução de *software* baseado num transputer. Esta arquitectura conferiu ao EdGAR um estrutura com características interessantes para a investigação de *codesign* - desenvolvimento conjunto de *hardware* e *software*, a partir de especificações de alto nível [Subrahmanayam 1993] e [Buchenrieder e Rozenblit 1995]. O estudo encetado sobre a variação sintáctica e a adequação da lógica programável será utilizado no projecto de um sistema de desenvolvimento de *codesign* para o EdGAR. Tal como já foi referido, essa utilização carece da necessária generalização dos resultados obtidos, alargando o estudo a um leque mais significativo de circuitos.

Por outro lado, atendendo a que em última análise o controlador do GLiTCH é apenas um *control path* e que a lógica reprogramável existente no EdGAR tem a capacidade para realizar múltiplos *data path*, a sua utilização conjunta pode ser generalizada para além da relação que existe na prevista emulação do GLiTCH. Esta utilização introduz uma nova dimensão no *codesign*, prevendo uma alternativa de implementação intermédia entre as tradicionais partições de *hardware* e de *software* - correspondendo à implementação de uma descrição através de um microsequenciador. As vantagens e inconvenientes da integração desta solução num ambiente de *codesign* é um problema que exige investigação.

Durante o projecto do controlador do GLiTCH foram evidenciadas as limitações impostas pelas ferramentas de síntese às descrições comportamentais. A natural evolução das técnicas de HLS irão com certeza permitir ultrapassar gradualmente essas dificuldades, permitindo a síntese a partir de níveis de abstracção cada vez mais elevados - um processo muito semelhante ao verificado na evolução do desenvolvimento de *software*. Para tal é necessário continuar a investigar modelos de representação alternativos, e algoritmos mais sofisticados. Esta área merecia já do autor uma especial atenção, mas o trabalho desenvolvido durante o projecto do controlador do GLiTCH contribui significativamente para aumentar os seus conhecimento e o seu interesse por este apaixonante domínio da engenharia de computadores, esperando, um dia, poder vir a contribuir para o seu enriquecimento.

Bibliografia

Abramovici, M., M. A. Breuer e A. D. Friedman, *Digital Systems Testing and Testable Design,* IEEE Press, 1990.

Alford, R., *Programmable Logic Designer's Guide,* Sams and Co., 1989.

Altera, *FLEX 8000 Programmable Logic Device Family,* Altera Corporation, 1993.

AMD, *The Am2900 Family Data Book,* Advanced Micro Devices, Inc., 1976.

AMD, *32-Bit Microprogrammable Products Am29C300/29300 - Data Book,* Advanced Micro Devices, Inc., 1988.

AMD, *MACH 1, 2, 3 and 4 Family Data Book,* Advanced Micro Devices, Inc., 1995.

Armstrong, J. R. e F. G. Gray, *Structured Logic Design with VHDL,* PTR Prentice-Hall, Inc., 1993.

Ashenden, P. J., *The VHDL Cookbook,* Department of Computer Science, University of Adelaide, South Australia, 1990.

Aylor, J. H., R. Waxman, B. W. Johnson e R. D. Williams, *The Integration of Performance and Functional Modeling in VHDL* in Performance and Fault Modeling with VHDL, J. M. Schoen, Prentince-Hall, Inc., 1992.

Babbage, H. P., *Of the analytical engine* in Charles Babage - Passages from the Life of a Philosopher, M. Campbell-Kelly, IEEE Press, 1994, 1864.

Barnes, G. H. e D. L. Slotnick, *The ILLIAC IV Computer,* IEEE Transactions on Computers, Vol C-17, nº 8, p. 746-757, 1968.

Batcher, K. E., *STARAN parallel processor system hardware,* National Computer Conference, 1974.

Batcher, K. E., *MPP: a high-speed image processor* in Algorithmically Specialised Parallel Computers, Academic Press, 1985.

Bergé, J. M., A. Fonkoua, S. Maginot e J. Rouillard, *VHDL'92 the new features of the VHDL hardware description language,* Kluwer Academic Publishers, 1993.

Borrione, D., L. Pierre e A. Salem, *Formal verification pf VHDL descriptions in Boyer-Moore: first results* in VHDL for Simulation, Synthesis and Formal Proofs of Hardware, J. Mermet, Kluwer Academic Publishers, 1992.

Bostock, G., *Programmable Logic Handbook,* Butterworth Heinemann Ltd., 1993.

Bouknight, W. J., S. A. Deneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh e D. L. Slotnick, *The ILLIAC IV System,* Proceedings of the IEEE, Vol 60, nº 4, p. 369-379, 1972.

Buchenrieder, K. e J. W. Rozenblit, *Codesign: An Overview* in Codesign. Computer-Aided Software/Hardware Engineering, J. Rozenblit and K. Buchenrieder, IEEE Press, 1995.

Burks, A. W., H. H. Goldstine e J. V. Newmann, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument Pt. 1 Vol. 1,* Institute for Advanced Study, Princeton, NJ, 1946.

Burrier, D., *Advanced language leads to smoother designing with programmable logic,* Electronic Design, August, p. 247-258,1984.

Camposano, R., *Generic Design Techniques: High-Level Design,* High-Level Digital System Design, An EUROCHIP Advanced Course, 1993.

Camposano, R., L. F. Sounders e R. M. Tabet, *VHDL as Input for High-Level Synthesis,* IEEE Design & Test of Computers, Vol 8, nº 1, p. 43-49, 1991.

Camposano, R., *From Behavior to Structure: High-Level Synthesis,* IEEE Design & Test of Computers, Vol 7, nº 5, p. 8-19, 1990.

Cantoni, V., M. Ferreti, S. Levialdi e R. Stefanelli, *PAPIA: Pyramidal Architecture for Parallel Image Analysis,* Proc. 7th IEEE Symposium on Computer Architecture, 1985.

Coelho, D. R., *The VHDL Handbook,* Kluwer Academic Publishers, 1992.

Cohen, B., *VHDL Coding Styles and Methodologies,* Kluwer Academic Publishers, 1995.

Courtois, B., *CAD and Testing of ICs and systems: where are we going ?,* TIMA, Research Report, 1993.

DATA, I., *Programmable Logic - A Basic Guide for the Designer,* DATA I/O, 1986.

Dewey, A. e A. J. D. Geus, *VHDL: Toward a Unified View of Design,* IEEE Design & Test of Computers, Vol 9, nº 2, p. 8-17, 1992.

Dollas, A. e N. Kanopoulos, *Reducing the Time to Market Through Rapid Prototyping,* IEEE Computer, Vol 28, nº 2, p. 14-15, 1995.

Duff, M., *CLIP4: a large scale integrated circuit array parallel processor,* 3rd International Joint Conference Pattern Recognition, Coronado, Ca., 1976.

Duller, A. W. G., A. D. Morgan e R. Storer, *Associative processor arrays: Simulation and performance estimates for image processing,* 3rd Alvey Vision Conference, 1987.

Duller, A. W. G., R. H. Storer, A. R. Thomson e E. L. Dagless, *An Associative Processor Array for Image Processing,* Image and Vision Computing, Vol 7, nº 2, p. 151-158, 1989a.

Duller, A. W. G., R. H. Storer, A. R. Thomson, E. L. Dagless, M. R. Pout e J. G. (A.P.Marriot, *Design of an Associative Processor Array,* 1989b.

Duller, A. W. G., R. H. Storer, A. R. Thomson, M. R. Pout e E. L. Dagless, *An Associative Processor Array Designed for Computer Vision,* Fifth Alvey Vision Conference, Reading, 1989c.

Duller, A. W. G., R. H. Storer, A. R. Thomson, M. R. Pout e E. L. Dagless, *Image Processing Applications using an Associative Processor Array,* Fifth Alvey Vision Conference, Reading, 1989d.

Duller, A. W. G., R. H. Storer, A. R. Thomson, M. R. Pout e E. L. Dagless, *A Heterogenous Vision Architecture,* Computer Vision - ECCV 90, 1990.

Ecker, W. e S. März, *System-Level Specification and Design Using VHDL. A Case Study,* 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL'93, Ottawa, Ontario Canada, 1993.

Fisher, J. A., *Very Long Instruction Word Architectures and the ELI-512,* 10th Symposium on Computer Architecture, 1983.

Flynn, M. J., *Some computer organizations and their effectiveness,* IEEE Transactions on Computers, Vol C-21, nº 9, p. 948-960, 1972.

Fountain, J., *An evaluation of some chips for image processing* in Evaluation of Multicomputers for Image Processing, L. Uhr, K. P. Jr., S. Levialdi and M. Duff, Academic Press, 1986.

Fountain, T., K. Matthews e M. Duff, *The CLIP7A image processor,* IEEE Transactions PAMI, Vol 10, nº p. 310-319, 1988.

Gajski, D. e R. Kuhn, *Guest's editors introduction: New VLSI Tools,* IEEE Computer, Vol 16, nº 22, p. 11-14, 1983.

Gajski, D. D. e L. Ramachandran, *Introduction to High-Level Synthesis,* IEEE Design & Test of Computers, Vol 11, nº 4, p. 44-54, 1994.

Harel, D., *On Visual Formalism,* Communications of the ACM, Vol 31, nº 5, p. 1988.

Hill, F. J., *AHPL: Then and Now,* IEEE Design & Test of Computers, Vol 9, nº 2, p. 73-75, 1992.

Hillis, W. D., *The Connection Machine,* MIT Press, 1985.

Hinton, J. e A. Pinder, *Transputer Hardware and System Design,* Prentice Hall, 1993.

Hockney, R. W. e C. R. Jesshope, *Parallel Computers. Architecture, Programming and Algorithms,* Adam Hilger Lda, 1981.

Hockney, R. W. e C. R. Jesshope, *Parallel Computers 2. Architecture, Programming and Algorithms,* Adam Hilger Lda, 1988.

Houzet, D., *Conception et Etude de l'Architecture Parallele de Traitement d'Image GFLOPS,* Institut National Polytechnique de Toulouse, Toulouse, France, PhD, 1992.

Hsu, Y., K. F. Tsai e E. S. Lin, *VHDL Modeling for Digital Design Synthesis,* Kluwer Academic Publishers, 1995.

Hwang, K. e F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill International Editions, 1989.

IEEE, *Three Decades of HDLs - Part 1: CDL through TI-HDL,* IEEE Design & Test of Computers, Vol 9, nº 2, p. 69-81, 1992a.

IEEE, *Three Decades of HDLs - Part 2: Conlan through Verilog,* IEEE Design & Test of Computers, Vol 9, nº 3, p. 54-63, 1992b.

IEEE, *IEEE Standard VHDL Language Reference Manual,* Institute of Electrical and Electronics Engineers, 1994.

INMOS, *The Transputer Databook, third edition,* SGS-Thomson Microelectronics, 1992.

Johnson, E. E., *Completing an MIMD multiprocessor taxonomy,* Computer Architecture News (ACM SIGArch), Vol 16, nº 3, p. 44-47, 1988.

Katz, R. H., *Contemporary Logic Design,* The Benjamin/Cummings Publishing Company, Inc., 1994.

Kitson, B. e K. Ow-Wing, *Logic-programming language enriches design process,* 1984.

Ku, D. e G. D. Micheli, *HardwareC - A Language for Hardware Design Version 2.0,* Computer Systems Laboratory, Stanford University, Technical Report, 1990.

Lea, R. M., *SCAPE: a single-chip array processing element for signal and image processing,* IEE Proceedings Part E, Computers and Digital Techniques, Vol 3, nº 133, p. 145-151, 1986.

Leibson, S., *The Handbook of Microcomputer Interfacing,* Tab Books, Inc., 1989.

Maresca, M., M. Lavin e H. Li, *Parallel architectures for vision,* IEEE, 1988.

Marschner, F. E., *Evolutionary Processes in Language, Software and System Design* in VHDL for Simulation, Synthesis and Formal Proofs of Hardware, Kluwer Academic Publishers, 1992.

McClean, J., *Testing and Testable Design,* Dpt. of Electrical & Electronic Engineering, Heriot-Watt University, Edinburgh, Scotland, 1992.

Menabrea, G. L. F., *Sketch of the analytical engine invented by Charles Babbage,* 1842.

Micheli, G. D., *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, 1994.

Morant, M. J., *Integrated Circuit Design and Technology,* Chapman and Hall, 1990.

Nagasamy, V., N. Berry e C. Dangelo, *Specification, Planning, and Synthesis in a VHDL Design Environment,* IEEE Design & Test of Computers, Vol 9, nº 2, p. 58-68, 1992.

Narayan, S., F. Vahid e D. D. Gajski, *System Specification with the SpecCharts Language,* IEEE Design & Test of Computers, Vol 9, nº 4, p. 6-13, 1992.

Neumann, J. v., *A system of 29 states with a general transition rule* in Theory of Self-Reproducing Automata, B. A.W., University of Illinois, 1966.

Newton, A. R., *Introduction to Design Automation for Electronic Systems,* 1991.

Newton, A. R. e A. L. Sangiovanni-Vincentelli, *Computer-Aided Design for VLSI Circuits,* IEEE Computer, Vol 19, nº p. 38-59, 1986.

Pearson, M. W., *PICSIL: Design and Synthesis of Digital ICs from Data Flow Diagrams,* Massey University, PhD thesis, 1992.

Pellerin, D. e M. Holley, *Practical Design Using Programmable Logic,* Prentice-Hall, Inc., 1991.

Perry, D. L., *VHDL,* McGraw-Hill, Inc., 1991.

Petr, H., *GLiTCH Debugger and Loader Reference Manual,* Dpt. Informática, University of Minho, Internal report, 1993.

Pout, M. R., R. Storer, A. R. Thomson e E. L. Dagless, *An Associative Processor Array as part of a Heterogeneous Vision Architecture,* 24th Annual International Conference on Systems Sciences, 1991.

Proença, A. J., *Development Stages in Real-time Computer Vision Architectures,* Vector Parallel Processing Conference VecPar93, Porto, 1993.

Quinn, M. J., *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill International Editions, Computer Science Series, 1988.

Reddaway, S. F., *DAP - a Distributed Processor Array,* National Computing Conference, 1978.

Roy, J., N. Kumar, R. Dutta e R. Vemuri, *DSS: A distributed High-Level Synthesis Sysstem,* IEEE Design & Test of Computers, Vol 9, nº 2, p. 18-32, 1992.

Rudolph, J. A. e K. E. Batcher, *A Productive Implementation of an Array Processor: STARAN* in Computer Structures: Principles and Examples, D. P. Siewioreq, C. G. Bell and A. Newell, McGraw-Hill, 1985.

Santos, H. e A. J. Proença, *GLiTCH System Controller: General Description - version 1.20,* Dtp. Informática, University of Minho, Internal report, 1993.

Santos, H., J. C. Ramalho, J. M. Fernandes e A. Proença, *A Heterogeneous Computer Vision Architecture: Implementation Issues,* Computing Systems in Engineering, Vol 6, nº 4/5, p. 401-408, 1995.

Sentovich, E. M., K. J. Singh, L. Lavagno, C. Moon e e. all, *SIS: A System for Sequential Circuit Synthesis,* Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, Memorandum, 1992.

Shore, J. E., *Second thoughts on parallel processing,* Comput. Elect. Eng., Vol 1, nº p. 95-109, 1973.

Siewiorek, D. P., C. G. Bell e A. Newell, *Computer Structures: Principles and Examples,* MxGraw-Hill, 1985.

Sinander, P., *VHDL Modelling Guidelines,* Onboard Data Division, european space agency, european space research and technology centre, The Netherlands, 1994.

Slotnick, D. L., *Unconventional Systems,* AFIPS Conference Proceedings, 1967.

Slotnick, D. L., W. C. Borck e R. C. McReynolds, *The SOLOMAN computer,* AFIPS Conference Proceedings, 1962.

Small, B., A. Bodmer e S. Mourad, *Evaluating FPGAs: Capacity and Utilization,* 1993.

Stavridou, V., H. Barringer e D. Edwards, *Formal Specification and Verification of Hardware: A Comparative Case Study,* Department of Computer Science, University of Manchester, Technical Report, 1987.

Storer, R., *An instruction set for an Associative processor Array,* Faculty of Engineering, Dpt. of Electric and Electronic Engineering, University of Bristol, Technical report, 1987.

Storer, R., *A Content Addressable Parallel Processor and its Application to Synthetic Image Generation,* Faculty of Engineering, Dpt. of Electric and Electronic Engineering, University of Bristol, PhD Thesis, 1991a.

Storer, R., *The GLiTCH Simulator ver. 2.0,* Faculty of Engineering, Dpt. of Electric and Electronic Engineering, University of Bristol, Internal Report, 1991b.

Storer, R., M. Pout, A. Thomson, E. Dagless, A. Duller, A. Marriot e P. Hicks, *An associative processing module for a Heterogeneous Vision Architecture,* IEEE Micro, Vol 12, nº 3, p. 31-41, 1992.

Subrahmanayam, P., *Hardware-Software Codesign: Cautious Optimism for the Future (Hot Topics),* 1993.

Sunwoo, M. e J. Aggarwal, *VisTA for a general purpose computer vision system,* Proc. 10th International Conference Pattern Recognition, 1990a.

Tanenbaum, A. S., *Structured Computer Organization,* Prentice-Hall, Inc., 1976.

Thomson, A. R., *A Cam-based Processor Array for Real-time Image Applications,* Faculty of Engineering, Dpt. of Electric and Electronic Engineering, University of Bristol, PhD Thesis, 1994.

Unger, S. H., *A computer oriented toward spatial problems,* Proceedings of the IRE, 1958.

Unger, S. H., *Pattern detection and recognition,* Proceedings of the IRE, 1959.

VIEWlogic, *VHDL Reference Manual for Synthesis,* Viewlogic Systems, Inc., 1992.

Walker, R. A. e R. Camposano, *A Survey of High-Level Synthesis Systems,* Kluwer Academic Press, 1991.

Walker, R. A. e S. Chaudhuri, *Introduction to the Scheduling Problem,* IEEE Design & Test of Computers, Vol 12, nº 2, p. 60-69, 1995.

Wallach, Y., *Alternating Sequential/Parallel Processing,* Springer-Verlag, 1981.

Waxman, R., *Hardware Design Languages for Computer Design and Test,* IEEE Computer, Vol 19, nº p. 90-97, 1986.

Weems, C., E. Riseman e A. Hanson, *Image Understanding Architecture: exploiting potential parallelism in machine vision,* IEEE Computer, Vol 25, nº 2, p. 65-68, 1992.

Weinmann, U., A. Kunzmann e U. Strohmeir, *Evaluation of FPGA Architectures,* Abingdon EE&CS Books, 1991.

Wilkins, B., *Testing Digital Circuits,* Van Nostrand Reinhold (UK) Co. Ltd, 1986.

Wolfe, A. e J. P. Shen, *A Variable Instruction Stream Extension to the VLIW Architecture,* ACM, 1991.

Xilinx, *The Programmable Logic Data Book,* Xilinx, 1994.

York, T., *Digital System Design: Introduction to Custom Integrated Circuits,* 1994.

Apêndice A - Modelo VHDL do controlador do GLiTCH

Índice

1. *Package* “Gl_pack.vhdl” 147

2. Unidade de memória de padrões 150
2.1 Bancos de memória 150
2.2 *Address Generation Logic* 153

3. Unidade escalar 155

4. Selector de *tags* 159

5. Sequenciador 161

6. Micromemória 169

7. *Bus Arbiter* 174

8. Operação do processador hospedeiro 179

9. Modelo estrutural do controlador 183

1. *Package* “Gl_pack.vhdl”

```
-- Package para as definicoes e procedimentos globais ao sistema GLiTCH
-- Autor: Henrique Santos  Data: Junho, 1994

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;

package gl_pack is

-- Inteiros positivos de 16 bits
  subtype int_16 is integer range 0 to (2**16) - 1;

-- Enderecos para a micromemoria
  subtype mm_end is integer range 0 to (32*1024) - 1;

-- Enderecos para o DS
  subtype ds_end is integer range 0 to (8*1024) - 1;

-- Instrucoes para o sequenciador. As instrucoes desde GOTO_D ate' END_FOR
-- apresentam tres variacoes: Sao executadas se uma condicao for verdadeira,
-- se nao for verdadeira, ou ainda incondicionalmente. O campo "qualif"
-- qualifica a instrucao a esse respeito. Os campos "data" e "Sfs" podem
-- ser considerados operandos de uma instrucao.
  type seq_instr_type is (GOTO_D, CALL_D, EXIT_D, END_FOR_D,
                          GOTO_M, CALL_M, EXIT_M, END_FOR_M,
                          END_LOOP, CALL_COR, S_RET, END_FOR,  -- Cond./Incond.
                          CONTINUE, FOR_D, DECREMENT, S_LOOP,
                          PUSH_D, RESET_SP, POP_C, PUSH_C,
                          SWAP, PUSH_C_LOAD_D, LOAD_D, SET, CLEAR);
  type qualif_type is (COND, NOT_COND, INCOND);
  type seq_instr is record
    instr: seq_instr_type;
    qualif: qualif_type;
    data: int_16;
    Sfs: integer range 0 to 15;
  end record;

-- Instrucoes para o Address Generation Logic: o campo "instr" comanda o
-- registo de endereco. O campo "ctr" comando os sinais de controlo do DS. O
-- campo "data" e' o operando da instrucao LOAD.
  type agl_instr_type is (HOLD, DEC, INC, LOAD);
  type agl_ctr_type is (DIS, WR, RD);
  type agl_instr is record
    instr: agl_instr_type;
    ctr: agl_ctr_type;
    data: ds_end;
  end record;

-- Instrucoes para a Scalar Unit
  type su_instr is (HLD, ASR, LSL, LSR, LDR, OSR);

-- Instrucoes para a Tag Route unit
  type tr_instr is (HOLD, A_RST, A_SET, CHAIN);


-- Barramento de dados comum ao DS, SU e componentes directamente associados
-- ao GLiTCH. A informacao que passa por este barramento e':
```

```
--    ."patterns ternarios" do DS para o GLiTCH
--    .valores binarios da SU para o GLiTCH e/ou DS e sentido inverso
--    .valores binarios do GLiTCH para o DS.
-- Este barramento transporta bits. O mais natural e' usar o tipo
-- std_logic_vector (ja' "resolvido", do package IEEE)
--             std_logic_vector(31 downto 0);

-- Micro instrucao do GLiTCH - conteudo de uma palavra da micro memoria
-- (os campos comentados nao sao implementados neste modelo).
  type mm_instr is record
    Seqi: seq_instr;
    DSi:   agl_instr;
--    VSR0, Sdone: bit;
--    Gi:    glitch_instr;                   De futuro isto sera' um record?
--    Gpos:  integer range 0 to (2**12) - 1;
--    GRadd: int_16;
--    Gsub:  integer range 0 to (2**8) - 1;
--    GRi:   gram_instr;
    SUi:   su_instr;
    TRi:   tr_instr;
    SU_TRsel: bit;
--    DRi:   dr_instr;                        Idem para aqui, mas isto nem
--    DRin:  ??;                              deve ser necessario!!
--    DRctr: ??;
    Sreq, EOC, Ss: bit;
  end record;

-- Funcao para achar o inteiro equivalente ao valor binario (bit_vector)
-- passado como parametro. Assume sempre que o elemento(0) e' o de menor
-- peso - funciona com os dois tipos de "range"
function bit2int(vec_in: bit_vector) return integer;

-- Procedimentos de transformacao de valores de vectores, entre os tipos
-- bit_vector e std_logic_vector. Os parametros devem ser sinais.
procedure vec2std_vec(signal vecin: bit_vector;
                      signal vecout: out std_logic_vector);
procedure std_vec2vec(signal vecin: std_logic_vector;
                      signal vecout: out bit_vector);
-- NOTA: este ultimo procedimento activa vecout a '1' nas posicoes onde
-- vecin tambem e' '1' e a '0' em todos os outros casos, mas dando a
-- indicacao do possivel erro.

-- Procedimento para envio de uma linha de mensagem para o monitor
procedure putline(constant mess: in string);

end gl_pack;

------------------
-- package body --
------------------
package body gl_pack is

function bit2int(vec_in: bit_vector) return integer is
  variable acom, weight: integer;
  begin
    acom := 0;
    weight := 1;
    if (vec_in'left > vec_in'right) then  -- range is .. downto ..
      for i in vec_in'reverse_range loop
```

```
        if (vec_in(i) = '1') then acom := acom + weight;
        end if;
        weight := weight * 2;
      end loop;
     else                          -- range is .. to ..
       for i in vec_in'range loop
         if (vec_in(i) = '1') then acom := acom + weight;
         end if;
         weight := weight * 2;
       end loop;
    end if;
    return acom;
  end bit2int;

procedure vec2std_vec(signal vecin: bit_vector;
                      signal vecout: out std_logic_vector) is
  begin
    for i in vecin'range loop
      case vecin(i) is
        when '0' => vecout(i) <= '0';
        when '1' => vecout(i) <= '1';
      end case;
    end loop;
    return;
  end vec2std_vec;

procedure std_vec2vec(signal vecin: std_logic_vector;
                      signal vecout: out bit_vector) is
  begin
    for i in vecin'range loop
      case vecin(i) is
        when '1'    => vecout(i) <= '1';
        when '0'    => vecout(i) <= '0';
        when others => assert FALSE
                         report "Conversao std_vec -> vec: Z, X... -> 0";
                       vecout(i) <= '0';
      end case;
    end loop;
    return;
  end std_vec2vec;

procedure putline(constant mess: in string) is
  variable linha_tx: line;
  begin
    write(linha_tx, mess);
    writeline(output, linha_tx);
  end putline;

end gl_pack;
```

2. Unidade de memória de padrões

2.1 Bancos de memória

```
-- Modelo Comportamental para o Data Store
-- Autor: Henrique Santos  Data: Junho, 1994

-- Os sinais de dados do DS encontram-se ligados ao barramento comum da SU
-- e do GLiTCH. Por esse motivo tem que ser de um tipo "resolved". De acordo
-- com a definicao tecnica do controlador, o DS vai armazenar "patterns"
-- ternarios definidos por combinacoes binarias. Sendo assim, o mais simples
-- e' utilizar o tipo 'std_logic_vector' definido na biblioteca
-- IEEE.STD_LOGIC_1164.
-- O DS e' composto por dois bancos ("Mask bank" e "Data Bank"). Do ponto de
-- vista do hardware eles sao identicos e poderiam ser descritos
-- estruturalmente a partir de um "componente" comum. No entanto tal
-- metodologia nao vai ser utilizada, para preservar o mesmo tipo de
-- arquitectura em todas as unidades.
-- As linhas de enderco e controlo sao partilhadas com o Host Transputer, o
-- que tambem implica a utilizacao de sinais "resolved".

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use WORK.gl_pack.all;


entity ds is
   port (endereco: in ds_end;
         dado: inout std_logic_vector(31 downto 0);
         notrd, notwr: in std_logic;
         notcs: in std_logic_vector(1 downto 0) ); -- notcs(1) -> Mask Bank
end ds;                                             -- notcs(0) -> Data Bank

-- A arquitectura implementada utiliza tres processos. Dois para o bloco de
-- memoria propriamente dito e outro para controlar as linhas de dados. Este
-- ultimo processo divide o barramento de dados em dois bancos e coloca as
-- linhas de dados em alta-impedancia se detectar um comando desconhecido.
architecture behavior of ds is

  subtype mem_cell is bit_vector (15 downto 0);
  -- O bloco de memoria e' um array, cuja dimensao e' imposta pela gama de
  -- enderecos pre-defina no package global.
  type mem_bloco is array (ds_end'low to ds_end'high) of mem_cell;

  signal mask_in, mask_out, data_in, data_out: mem_cell;
  signal MaskRdnotWr, DataRdnotWr: bit;
  signal aux_ctr: std_logic_vector(3 downto 0);
  -- signal mask_cel, data_cel: mem_cell; -- auxiliares na transferencia

  --{{{  Procedimentos de leitura e escrita na memoria
    -- Procedimentos de leitura e escrita na memoria
    procedure le_ds (variable memoria: in mem_bloco;
                     variable endereco: in ds_end;
                     signal val: out mem_cell);
    procedure wr_ds (variable memoria: inout mem_bloco;
                     variable endereco: in ds_end;
                     signal val: in mem_cell);
    procedure le_ds (variable memoria: in mem_bloco;
```

```
                    variable endereco: in ds_end;
                    signal val: out mem_cell) is
  begin
    val <= memoria(endereco);
  end le_ds;
  procedure wr_ds (variable memoria: inout mem_bloco;
                   variable endereco: in ds_end;
                   signal val: in mem_cell) is
  begin
    memoria(endereco) := val;
  end wr_ds;
--}}}

 begin
   --{{{  Mask:process (endereco, mask_in, MaskRdnotWr)
        Mask:process (endereco, mask_in, MaskRdnotWr)
          variable mem_mask: mem_bloco;
          variable inicio: boolean := TRUE;
          variable pont: ds_end;
          begin
            if inicio then
              --{{{  Bloco de inicializacao da memoria (Mask Bank)
              -- Por defeito, coloca toda a memoria a 0
              for i in mem_mask'range loop

              end loop;
              inicio := FALSE;
              --}}}
             else
               pont := endereco;
               if (MaskRdnotWr = '1') then       -- Cond. de leitura
                 le_ds(mem_mask, pont, mask_out);
                else                             -- Cond. escrita
                  wr_ds(mem_mask, pont, mask_in);
               end if;
            end if;
          end process Mask;
   --}}}
   --{{{  Data:process (endereco, data_in, DataRdnotWr)
        Data:process (endereco, data_in, DataRdnotWr)
          variable mem_data: mem_bloco;
          variable inicio: boolean := TRUE;
          variable pont: ds_end;
          begin
            if inicio then
              --{{{  Bloco de inicializacao da memoria (Data Bank)
              -- Por defeito, coloca toda a memoria a 0
              for i in mem_data'range loop

              end loop;
              inicio := FALSE;
              --}}}
             else
               pont := endereco;
               if (DataRdnotWr = '1') then       -- Cond. de leitura
                 le_ds(mem_data, pont, data_out);
                else                             -- Cond. escrita
                  wr_ds(mem_data, pont, data_in);
               end if;
```

```
        end if;
      end process;
--}}}
--{{{  process (dado, notrd, notwr, notcs, mask_out, data_out)
     process (dado, notrd, notwr, notcs, mask_out, data_out)
       variable inicio: boolean := TRUE;
       variable aux: std_logic_vector(1 downto 0);
       alias dado_Mask: std_logic_vector(15 downto 0) is dado(31 downto 16);
       alias dado_Data: std_logic_vector(15 downto 0) is dado(15 downto 0);
       constant high_z: std_logic_vector(15 downto 0) :=
                ('Z','Z','Z','Z','Z','Z','Z','Z',
                 'Z','Z','Z','Z','Z','Z','Z','Z');
       -- Funcao para determinar se algum elemento de um array do tipo
       -- std_logic_vector e' diferente de '0' ou '1'.
       function vec_improprio(signal vecin: std_logic_vector) return boolean;
       function vec_improprio(signal vecin: std_logic_vector) return boolean is
         variable aux:boolean := FALSE;
         begin
           for i in vecin'range loop
             if ((vecin(i) = '0') or (vecin(i) = '1')) then
               NULL;
              else
                aux := TRUE; exit;
             end if;
           end loop;
           return aux;
         end vec_improprio;
       begin
         --{{{  if inicio then
         if inicio then
           dado_Mask <= high_z; dado_Data <= high_z;
           MaskRdnotWr <= '1';
           DataRdnotWr <= '1';
           inicio := FALSE;
         --}}}
          else
        aux_ctr <= notcs & notrd & notwr;
            assert not(vec_improprio(aux_ctr))
              report "DataStore control: Comando incorrecto!"
                severity WARNING;
            aux := notrd & notwr;
            case aux is
              when "01" => MaskRdnotWr <= '1'; DataRdnotWr <= '1'; -- Leitura
                           if (notcs(1) = '0') then
                             vec2std_vec(mask_out, dado_Mask);
                            else dado_Mask <= high_z;
                           end if;
                           if (notcs(0) = '0') then
                             vec2std_vec(data_out, dado_Data);
                            else dado_Data <= high_z;
                           end if;
              when "10" => if (notcs(1) = '0') then  -- Escrita
                             MaskRdnotWr <= '0';
                             std_vec2vec(dado_Mask, mask_in);
                           else
                             MaskRdnotWr <= '1'; dado_Mask <= high_z;
                           end if;
                           if (notcs(0) = '0') then
                             DataRdnotWr <= '0';
```

```
                                    std_vec2vec(dado_Data, data_in);
                                  else
                                    DataRdnotWr <= '1'; dado_Data <= high_z;
                                  end if;
                    when others => MaskRdnotWr <= '1'; DataRdnotWr <= '1';
                                    dado_Mask <= high_z; dado_Data <= high_z;
                end case;
              end if;
            end process;
     --}}}
  end behavior;
```

2.2 *Address Generation Logic*

```
-- Modelo behavior para o modulo AGL.
-- Henrique Santos | Junho 94
-- Descricao de funcionamento no relatorio.

use work.gl_pack.all;
library IEEE;
use IEEE.STD_LOGIC_1164.all;

--{{{  entity agl_c
entity agl_c is
port (Instruction: in agl_instr; -- "record"
      clock, write_pulse, BusTrnotLoc1, ForceHold: in bit;
      Addr_out: out ds_end;
      notCS: out std_logic_vector(1 downto 0);
      notOE, notWE: out std_logic);
end agl_c;
--}}}

--{{{  architecture behavior
architecture behavior of agl_c is
  -- Constantes locais
  constant MaskBank: integer := 1;
  constant DataBank: integer := 0;
  begin
    ctr_unit: process(Clock, BusTrnotLoc1)
      -- Variaveis internas auxiliares.
      variable reg_add: ds_end;
      begin
        if ((Clock'event and (Clock = '1')) and (BusTrnotLoc1 = '0') and
            (ForceHold = '0')) then
          case Instruction.instr is
             when HOLD => NULL;
             when DEC  => reg_add := reg_add - 1; Addr_out <= reg_add;
             when INC  => reg_add := reg_add + 1; Addr_out <= reg_add;
             when LOAD => reg_add := Instruction.data; Addr_out <= reg_add;
          end case;
          case Instruction.ctr is
             when WR  => notWE <= '0'; notOE <= '1';
                         notCS(MaskBank) <= '1'; notCS(DataBank) <= '0';
             when RD  => notWE <= '1'; notOE <= '0';
                         notCS(MaskBank) <= '0'; notCS(DataBank) <= '0';
             when DIS => notWE <= '1'; notOE <= '1';
                         notCS(MaskBank) <= '1'; notCS(DataBank) <= '1';
```

```
          end case;
         elsif (BusTrnotLoc1 = '1') then
           notWE <= 'Z'; notOE <= 'Z'; notCS <= ('Z', 'Z');
        end if;
      end process;
  end behavior;
--}}}
```

3. Unidade escalar

```
-- Modelo behavioral para a Scalar Unit
-- Henrique Santos | Junho 94
-- A SU tem por objectivo executar "shifts" e "rotates" sobre um valor de
-- 32 bits, previamente capturado do barramento do Data Store. Tipicamente
-- esse valor nao e' interpretado como um inteiro, mas sim como um valor
-- binario (se vier do GLiTCH) ou um "pattern" (se vier do Data Store).
-- Assim sendo, as operacoes a realizar sao mesmo sobre um bit_vector!
-- A ligacao ao barramento e' feita por um vector de STD_LOGIC_VECTOR
-- 30/Jul/95
--   Os processos relativos ao registo de controlo e registo de dados usam
--   dois clocks distintos mas que tem a mesma freq. e fase, em principio. Se
--   no prototipo nao houve problemas, na simulacao pode ocorrer um erro se
--   o processo do reg. de dados for executado um 'dt' antes do processo do
--   registo. Isto aconteceu na simulacao com o ViewLogic!
library IEEE;
use IEEE.STD_LOGIC_1164.all;

--{{{  package su_c
package su_pack is

-- Funcao para determinar se um vector de bits e' zero. Como resultado
-- devolve TRUE se todos os elementos do vector forem 0.
function allzero(signal vecin: bit_vector) return boolean;

-- Os procedimentos seguintes destinam-se a fazer operacoes de "shift"
-- logico para a esquerda/direita e aritmetico para a esquerda, sobre o
-- operando passado como parametro ("unconstrained array"). Estas operacoes
-- identificam a gama ("to" ou "downto") de indices do array e operam de
-- acordo com essa propriedade, pelo que podem ser utilizadas genericamente.
procedure l_shl(variable vecin: inout bit_vector);
procedure l_shr(variable vecin: inout bit_vector);
procedure a_shr(variable vecin: inout bit_vector);

end su_pack;

package body su_pack is

--{{{  function allzero(signal vecin: bit_vector) return boolean is
function allzero(vecin: bit_vector) return boolean is
  begin
    for i in vecin'range loop
       if (vecin(i) /= '0') then
         return FALSE;
       end if;
    end loop;
    return TRUE;
  end allzero;
--}}}

--{{{  procedure l_shl(variable vecin: inout bit_vector) is
procedure l_shl(variable vecin: inout bit_vector) is
  variable aux: bit;
  begin
    if (vecin'left > vecin'right) then           -- Se e' verdade o array foi
      for i in vecin'range loop                  -- declarado x DOWNTO y
        if (i > vecin'right) then aux := vecin(i - 1);
          else aux := '0';
```

```
        end if;
        vecin(i) := aux;
      end loop;
     else                                          -- Se o array foi declarado
       for i in vecin'range loop                   -- x TO y
         if (i < vecin'right) then aux := vecin(i + 1);
           else aux := '0';
         end if;
         vecin(i) := aux;
       end loop;
    end if;
    return;
  end l_shl;
--}}}

--{{{  procedure l_shr(variable vecin: inout bit_vector) is
procedure l_shr(variable vecin: inout bit_vector) is
  variable aux: bit;
  begin
    if (vecin'left > vecin'right) then           -- Se e' verdade o array foi
      for i in vecin'reverse_range loop          -- declarado x DOWNTO y
        if (i < vecin'left) then aux := vecin(i + 1);
          else aux := '0';
        end if;
        vecin(i) := aux;
      end loop;
     else                                          -- Se o array foi declarado
       for i in vecin'reverse_range loop           -- x TO y
         if (i > vecin'left) then aux := vecin(i - 1);
           else aux := '0';
         end if;
         vecin(i) := aux;
       end loop;
    end if;
    return;
  end l_shr;
--}}}

--{{{  procedure a_shr(variable vecin: inout bit_vector) is
procedure a_shr(variable vecin: inout bit_vector) is
  variable aux: bit;
  begin
    if (vecin'left > vecin'right) then           -- Se e' verdade o array foi
      for i in vecin'reverse_range loop          -- declarado x DOWNTO y
        if (i < vecin'left) then aux := vecin(i + 1);
          else exit; -- else aux := vecin(i);
        end if;
        vecin(i) := aux;
      end loop;
     else                                          -- Se o array foi declarado
       for i in vecin'reverse_range loop           -- x TO y
         if (i > vecin'left) then aux := vecin(i - 1);
           else exit; -- else aux := vecin(i);
         end if;
         vecin(i) := aux;
       end loop;
    end if;
    return;
  end a_shr;
```

```
--}}}

end su_pack;
--}}}

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.su_pack.all;
use work.gl_pack.all;

--{{{  entity su_c
entity su_c is
port (signal io_bus: inout std_logic_vector(31 downto 0);
      signal Instruction: in su_instr;
      signal Hold, clk, s_clk, BusTrnotLoc1: in bit;
      signal LSB_3: out bit_vector(2 downto 0);
      signal Zero, MSB: out bit);
end su_c;
--}}}

--{{{  architecture behavior of su_c
architecture behavior of su_c is
constant high_z: std_logic_vector := ('Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z',
                                      'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z', 'Z');
--  Instrucoes para o registo de dados. Representa o registo interno onde
-- e' carregada cada instrucao a executar
signal instr_aux: su_instr;               -- registo local de instrucao
signal loc_reg: bit_vector(15 downto 0);  -- registo local de dados
signal aux_in: bit_vector(15 downto 0);
signal data_mask: std_logic_vector(15 downto 0);
  begin
    --{{{  reg_instruction: process(BusTrnotLoc1, clk, Hold)
    -- Para alem de registar a instrucao, ainda descodifica a instrucao
    -- OSR, que deve fazer o enable dos buffers de saida.
    reg_instruction: process(BusTrnotLoc1, clk, Hold)
    begin
       if ((clk'event and (clk = '1')) and (BusTrnotLoc1 = '0') and
           (Hold = '0')) then
          instr_aux <= Instruction;
          if (Instruction = OSR) then
            vec2std_vec(loc_reg, io_bus(15 downto 0)); -- enable outputs
           else io_bus(15 downto 0) <= high_z; -- disable outputs
          end if;
       end if;
    end process;
    --}}}
    --{{{  processo para o registo de deslocamento
    reg_saidaproc: process(s_clk)
    variable proc_inst: integer;
    variable aux_loc_reg: bit_vector(15 downto 0);  -- auxiliar de registo dados
    begin
      if (s_clk'event and (s_clk = '1')) then
         aux_loc_reg := loc_reg;
         case instr_aux is
            when HLD | OSR => NULL;
            when ASR => a_shr(aux_loc_reg); loc_reg <= aux_loc_reg;
            when LSR => l_shr(aux_loc_reg); loc_reg <= aux_loc_reg;
            when LSL => l_shl(aux_loc_reg); loc_reg <= aux_loc_reg;
            when LDR => data_mask <= io_bus(31 downto 16) and io_bus(15 downto 0);
```

```
                        std_vec2vec(data_mask, aux_in);
                        aux_loc_reg := aux_in;
          end case;
          loc_reg <= aux_loc_reg;
        end if;
      end process;
      --}}}

      MSB <= loc_reg(15);

      LSB_3 <= loc_reg(2 downto 0);

      Zero <= '1' when allzero(loc_reg) else '0';

    end behavior;
  --}}}
```

4. Selector de *tags*

```
-- Especificacao da TAG Route do GLiTCH - Behavioral
-- Autor: Henrique Santos  Data: Junho 94

--{{{  Definicao do Interface
-- Entradas:
--   TagUp, TagDown            -> do array
--   CloseCH, I[1:0], notHold  -> da micromemoria
--   --------------|
--                 |-> 0 0 0   HOLD "re-utiliza" a ultima instrucao
--                     1 0 0   CHAIN Fecha cadeia das tags
--                     0 1 0   SET Forca tags de saida a 1
--                     0 0 1   RESET Forca tags de saida a 0
--   Suspend, Clk                globais (do sistema)
-- Saidas:
--   TagAbove, TagBellow                Para o array
-- As seguintes saidas sao auxiliares e nao correspondem necessariamente
-- a pinos numa possivel implementacao. Apenas servem para interligar os
-- tres registos de pipe existentes no modelo.
--   Pipe1[2:0], Pipe2[2:0], Pipe3[2:0]
--}}}

use work.gl_pack.all;

--{{{  entity TR_c is
entity TR_c is
--  Portas de I/O
  port (signal Instruction: in tr_instr;
        signal TagUp, TagDown, notHold, Suspend, Clk: in bit;
        signal TagAbove, TagBellow: out bit);
end TR_c;
--}}}

--{{{  architecture behavioral of TR_c is
architecture behavioral of TR_c is

-- Variaveis, constantes e sinais globais
signal Pipe1, Pipe2, Pipe3: tr_instr;
  begin
    --{{{  proc_pipe1: process (Clk) -- nivel 1 dos registos de pipe
    -- Gera os sinais de saida de pipe1.
    -- O comportamento dste nivel depende ainda dos sinais de controlo
    -- Suspend e notHold.
    proc_pipe1: process (Clk)
    variable inicio: boolean := TRUE; -- Controlo da inicializacao
    begin
      --{{{  inicializacao
      if inicio then        -- O registo de saida associado a Pipe1
         Pipe1 <= CHAIN;    -- deve "acordar" com a instrucao HOLD.
         inicio := FALSE;
      end if;
      --}}}
      -- A condicao seguinte condiciona o carregamento de uma nova instrucao.
      -- De notar que a instrucao HOLD implica a manutencao da instrucao
      -- anterior, de acordo com a especificacao.
      if ((Clk'event and (Clk = '1')) and (Suspend = '0')) then
        if ((notHold /= '0') and (Instruction /= HOLD)) then
          Pipe1 <= Instruction;
```

```
        else NULL;
       end if;
     end if;
   end process;
   --}}}
   --{{{  proc_pipe2: process (Clk) -- nivel 2 dos registos de pipe
   -- Este nivel e' sensivel ao clock e assume a saida do anterior (sinal
   -- Pipe1) desde que o sinal Suspend nao esteja activo. A instrucao e'
   -- propagada para a frente atraves do sinal Pipe2.
   proc_pipe2: process (Clk)
   variable inicio: boolean := TRUE; -- Controlo da inicializacao
   begin
     --{{{  inicializacao
     if inicio then        -- O registo de saida associado a Pipe2
        Pipe2 <= CHAIN;    -- deve "acordar" com a instrucao CHAIN.
        inicio := FALSE;
     end if;
     --}}}
     if ((Clk'event and (Clk = '1')) and (Suspend = '0')) then
       Pipe2 <= Pipe1;
      else NULL;
     end if;
   end process;
   --}}}
   --{{{  proc_pipe3: process (Clk) -- nivel 3 dos registos de pipe
   -- Este nivel tem duas funcoes: registar a instrucao que lhe chega do nivel
   -- anterior e activar uma estrutura tipo multiplexar que implementa a
   -- funcao especificada para este componente, o routing das tags.
   proc_pipe3: process (Clk, TagDown, TagUp)
   variable inicio: boolean := TRUE; -- Controlo da inicializacao
   begin
     --{{{  inicializacao
     if inicio then        -- O registo de saida associado a Pipe3
        Pipe3 <= CHAIN;    -- deve "acordar" com a instrucao CHAIN.
        inicio := FALSE;
     end if;
     --}}}
     if ((Clk'event and (Clk = '1')) and (Suspend = '0')) then
       Pipe3 <= Pipe2;
      else NULL;
     end if;
   end process;
   --}}}
   with Pipe3 select
     TagAbove <= '1'     when A_SET,
                 '0'     when A_RST,
                 TagDown when CHAIN,
                 '0'     when others; -- HOLD nunca chega a Pipe3
   with Pipe3 select
     TagBellow <= '1'     when A_SET,
                  '0'     when A_RST,
                  TagUp   when CHAIN,
                  '0'     when others; -- HOLD nunca chega a Pipe3

 end behavioral;
--}}}
```

5. Sequenciador

```
-- Especificacao Sequenciador (Am29C331) do GLiTCH em VHDL
-- Revisao C em Junho 94 - Nivel de abstracao mais elevado: stack usa o
-- package generico e as instrucoes sao definidas por um record.

--{{{  Definicao do Interface
-- Entradas:
--   D[15:0]    Barramento D - APENAS USADO COMO ENTRADA
--   I[5:0]     Barramento de Instrucoes
--   A[15:0]    Barramento A - nao e' utilizado, externamente a GND
--   T[11:0]    Barramento das flags
--   S[3:0]     Barramento de seleccao das flags
--   CP         Clock
--   RST        Reset
--   M0[3:0]    Barramento M0
--   M1[3:0]    Barramento M1 - M1, M2 e M3 nao sao utilizados, externamente a GND
--   M2[3:0]    Barramento M2
--   M3[3:0]    Barramento M3
--   INTEN      Interupt Enable - nao utilizado, externamente a GND
--   INTR       Interrupt - idem
--   HOLD       Sinal para "suspensao"
--   OED        Output Enable D bus - externamente a GND
--   SLAVE      Nao utilizado - externamente a GND
--   CYIN       "Keep output address" - nao utilizado, externamente a GND
--   FCONT      Continue - nao utilizado, externamente a GND
-- Saidas:
--   Y[15:0]    Barramento Endereco de saida - apenas output na forma utilizada
--   EQUAL      Indicador de Breakpoint
--   A_FULL     Stack quase cheia
--   INTA       Interrupt acknowledge - nao utilizado
--   ERROR      nao utilizado
--}}}

use work.gl_pack.all;

--{{{  Package seq_c is
package seq_c is
-------------------------------------------------------------------------------
-- Declaracao de funcoes e procedimentos globais: get_flag() pop() push()
-- le_tos()
-------------------------------------------------------------------------------

-- Conjunto de instrucoes internas, comuns a todos os blocos. Cada bloco
-- reage aos valores de que depende. Estas instrucoes sao geradas pela
-- unidade de controlo
type op_icode is (NOP, DE_D, DE_M0, DE_TOS, DE_INC, DE_CONT, ZERO, POP_I,
                  DEC, DESAUT);

-- Devolve o estado da flag enderecada pelos sinais de seleccao. Ate ao
-- endereco 11 o acesso e' linear, mas a partir dai a condicao de saida e'
-- dada por uma operacao logica entre 2 ou mais flags.
function get_flag(signal flags: bit_vector;
                  constant seleccao: integer range 0 to 15) return boolean;

-- Bloco de declaracoes para implementacao da estrutura de pilha.
subtype pilha_cel is int_16;
subtype pilha_ind is integer range 0 to 32;
type pilha_bloco is array (integer range <>) of pilha_cel;
```

```
-- Funcoes de push e de pop com a stack. Estas funcoes testam se sp atinge
-- os limites minimo e maximo. A stack e' tratada como um buffer circular,
-- mas um aviso e' enviado ao simulador. De notar que acessos fora da gama
-- serao automaticamente rejeitados pelo simulador, segundo a descricao
-- standard do VHDL.
-- SP deve apontar sempre para a primeira posicao vazia.
procedure pop (variable stack: in pilha_bloco;
               variable sp: inout pilha_ind;
               variable val: out pilha_cel);
procedure push (variable stack: inout pilha_bloco;
                variable sp: inout pilha_ind;
                variable val: in pilha_cel);
procedure le_tos (variable stack: in pilha_bloco;
                  variable sp: in pilha_ind;
                  variable val: out pilha_cel);
end;

------------------
-- Package body --
------------------
package body seq_c is

--{{{  function get_flag()
function get_flag(signal flags: bit_vector;
                  constant seleccao: integer range 0 to 15) return boolean is
  variable tmp_ret: boolean := FALSE;
  variable aux_bit: bit;
  begin
    case seleccao is
      when 0 to 11 => if (flags(seleccao) = '1') then tmp_ret := TRUE; end if;
      when 12 => aux_bit := flags(8) OR flags(11);
                 if (aux_bit = '1') then tmp_ret := TRUE; end if;
      when 13 => aux_bit := (not flags(8)) OR flags(11);
                 if (aux_bit = '1') then tmp_ret := TRUE; end if;
      when 14 => aux_bit := flags(9) XOR flags(10);
                 if (aux_bit = '1') then tmp_ret := TRUE; end if;
      when 15 => aux_bit := (flags(9) XOR flags(10)) OR flags(11);
                 if (aux_bit = '1') then tmp_ret := TRUE; end if;
    end case;
    return tmp_ret;
  end;
--}}}

--{{{  procedure pop ()
procedure pop (variable stack: in pilha_bloco;
               variable sp: inout pilha_ind;
               variable val: out pilha_cel) is
begin
   assert (sp > stack'low)
      report "Operacao na STACK: underflow. SP assume stack'high"
         severity warning;
   if (sp = stack'low) then sp := stack'high;
    else sp := sp - 1;
   end if;
   val := stack(sp);
end pop;
--}}}

--{{{  procedure push ()
```

```
procedure push (variable stack: inout pilha_bloco;
                variable sp: inout pilha_ind;
                variable val: in pilha_cel)is
begin
  assert (sp < stack'HIGH)
    report "Operacao na STACK: overflow. SP assume stack'low"
      severity warning;
  stack(sp) := val;
  if (sp < stack'high) then sp := sp + 1;
   else sp := stack'low;
  end if;
end push;
--}}}

--{{{  procedure le_tos ()
procedure le_tos (variable stack: in pilha_bloco;
                  variable sp: in pilha_ind;
                  variable val: out pilha_cel)is
begin
  assert (sp >= stack'LOW) AND (sp <= stack'HIGH)
    report "Operacao na Stack: SP nao aponta a stack!"
      severity error;
  val := stack(sp);
end le_tos;
--}}}

end seq_c;
--}}}

use work.seq_c.all;
use work.gl_pack.all;
use std.textio.all;

--{{{  Declaracao do Interface - entity AM29C331 is
entity AM29C331 is
  port (I: seq_instr;                  -- record...
        T: in bit_vector(11 downto 0);
        M0: in bit_vector (3 downto 0);
        Cp, Reset, Inten, Intr, Hold, OEd, Slave, CyIn, Fcont: in bit;
        Y: out mm_end;
        Equal, A_Full, Inta, S_Error: out bit);
end AM29C331;
--}}}

--{{{  Arquitectura - architecture behavioral of AM29C331 is
-- O conjunto de blocos usado, bem como a sua operacao, estao associados
-- a versao simplificada do SEQ, tal como e' utilizada no controlador do
-- GLiTCH (nao ha interrupcoes e algumas das entradas sao ignoradas). Para
-- alem disso, as instrucoes nao suportadas nao sao executadas.

architecture behavioral of AM29C331 is
-- Barramentos de comunicacao entre os diversos modulos
signal d_bus, m_bus, inc_bus, cont_bus, y_aux_bus, tos_bus: int_16;

-- Barramentos de controlo sao de 4 bits e dedicados a cada unidade.
-- Isto esta' feito de modo a usar um unico conjunto de instrucoes internas,
-- mas leva a um desperdicio enorme, pois algumas das unidades apenas
-- necessitariam de 2 bits para controlo.
signal to_pilha,to_comparador, to_contador, to_end_saida,
```

```
     to_end_inc: op_icode;

begin
  d_bus <= I.data; -- os "pinos" I.data sao o driver do barramento interno d_bus

  m_bus <= ((d_bus / 16) * 16) + bit2int(M0);

  --{{{  descodificador: process(I, T)
  -- Activa as variaveis de ligacao para todos os blocos, de acordo
  -- com a instrucao de entrada.
  -- O descodificador e' um bloco interno, sendo modulado sem atrasos. Nao
  -- impoe qualquer restriccao visto que nao influencia os sinais de saida.
  -- Se o descodificador receber uma instrucao mal codificada, isto e', uma
  -- instrucao que nao seja valida para o identificador de instrucao utilizado
  -- (campo qualif), essa instrucao sera ignorada. Nao indicado o erro.

  descodificador: process(I, T)
    variable condicao: boolean; -- Resultado do teste de flags, ou TRUE no caso
                                -- de instrucoes incondicionais.
    begin
      case I.qualif is
        when INCOND => condicao := TRUE;
        when COND => condicao := get_flag(T, I.Sfs);
        when NOT_COND => condicao := not get_flag(T, I.Sfs);
      end case;
      if (I.qualif = NOT_COND) then
        --{{{  Conjunto de Instrucoes para funcoes especiais
             -- Bloco comparador
             case I.instr is
               when SET => to_comparador <= DE_D;
               when CLEAR => to_comparador <= DESAUT;
               when others => to_comparador <= NOP;
             end case;
             -- Bloco contador
             case I.instr is
               when FOR_D | PUSH_C_LOAD_D |
                    LOAD_D => to_contador <= DE_D;
               when DECREMENT => to_contador <= DEC;
               when POP_C | SWAP => to_contador <= DE_TOS;
               when others => to_contador <= NOP;
             end case;
             -- Bloco do mux de saida
             to_end_saida <= DE_INC;
             -- Bloco da pilha
             case I.instr is
               when FOR_D | S_LOOP => to_pilha <= DE_INC;
               when PUSH_D => to_pilha <= DE_D;
               when RESET_SP => to_pilha <= ZERO;
               when POP_C => to_pilha <= POP_I;
               when PUSH_C | SWAP | PUSH_C_LOAD_D  => to_pilha <= DE_CONT;
               when others => to_pilha <= NOP;
             end case;
       --}}}
       else
         --{{{  Conjunto de Instrucoes in/condicional
         -- Bloco comparador
         to_comparador <= NOP;
         -- Bloco contador
         case I.instr is
```

```
          when END_FOR_D | END_FOR_M | END_FOR => to_contador <= DEC;
          when others => to_contador <= NOP;
        end case;
        -- Bloco do mux de saida
        if condicao then
          case I.instr is
            when GOTO_D | CALL_D | EXIT_D => to_end_saida <= DE_D;
            when GOTO_M | CALL_M | EXIT_M => to_end_saida <= DE_M0;
            when END_LOOP | CALL_COR | S_RET => to_end_saida <= DE_TOS;
            when END_FOR_D => if cont_bus = 1 then
                                to_end_saida <= DE_INC;
                              else
                                 to_end_saida <= DE_D;
                              end if;
            when END_FOR_M => if cont_bus = 1 then
                                to_end_saida <= DE_INC;
                              else
                                 to_end_saida <= DE_M0;
                              end if;
            when END_FOR => if cont_bus = 1 then
                               to_end_saida <= DE_INC;
                            else
                               to_end_saida <= DE_TOS;
                            end if;
            when others => to_end_saida <= NOP;
          end case;
         else
             to_end_saida <= DE_INC;
        end if;
        -- Bloco da pilha
        case I.instr is
          when GOTO_D | END_FOR_D | GOTO_M | END_FOR_M => to_pilha <= NOP;
          when S_RET => to_pilha <= NOP;
          when CALL_D | CALL_M  => if condicao then
                                     to_pilha <= DE_INC;
                                   else
                                     to_pilha <= NOP;
                                   end if;
          when EXIT_D | EXIT_M  => if condicao then
                                     to_pilha <= POP_I;
                                   else
                                     to_pilha <= NOP;
                                   end if;
          when END_LOOP | END_FOR  => if condicao then
                                        to_pilha <= NOP;
                                      else
                                        to_pilha <= POP_I;
                                      end if;
          when CALL_COR  => if condicao then
                              to_pilha <= DE_INC;
                            else
                              to_pilha <= POP_I;
                            end if;
          when others => to_pilha <= NOP;
        end case;
        --}}}
    end if;
end process descodificador;
--}}}
```

```
--{{{  Mux_saida: process(to_end_saida, d_bus, m_bus, tos_bus, inc_bus)
-- Processo para controlar o Multiplexer de saida. Logica combinatoria.
-- Sinal to_end_saida liga este bloco ao descodificador de instrucoes.

Mux_saida: process(to_end_saida, d_bus, m_bus, tos_bus, inc_bus)
variable linha_texto:line;
begin
  case to_end_saida is
    when DE_D => y_aux_bus <= d_bus;
    when DE_M0 => y_aux_bus <= m_bus;
    when DE_TOS => y_aux_bus <= tos_bus;
    when DE_INC => y_aux_bus <= inc_bus;
    when others =>
      write(linha_texto, string'("Mux_saida: op_icode invalido. Y nao muda"));
      writeline(output, linha_texto);
  end case;
end process Mux_saida;
--}}}

--{{{  p_pilha: process(Reset, Hold, Cp)
-- A estrutura pilha esta' definida no package do sequenciador.
-- Ao contrario do circuito real, o pont_pilha
-- e' modelado como um contador modulo 32. A pilha e o ponteiro de pilha
-- sao sensiveis ao clock, sendo o ponteiro de pilha sensivel ainda ao
-- sinal de Reset e Hold (este devido apenas ao sinal A_full). A_Full esta'
-- activo sempre que 23 <= pont_pilha.

p_pilha: process(Reset, Hold, Cp)
variable pont_pilha: pilha_ind;
variable pilha: pilha_bloco(pilha_ind);
variable temp: int_16;
variable linha_texto: line;
begin
  if Reset = '0' then pont_pilha := 0;
   elsif (Hold = '1') then NULL;
    else
      if ((Cp'event) and (Cp = '1')) then
        case to_pilha is
          when DE_CONT => temp := cont_bus; push(pilha, pont_pilha, temp);
          when DE_INC => temp := inc_bus; push(pilha, pont_pilha, temp);
          when DE_D => temp := d_bus; push(pilha, pont_pilha, temp);
          when ZERO => pont_pilha := 0;
          when POP_I => pop(pilha, pont_pilha, temp); -- ?? <= temp
          when NOP => NULL;
          when others =>
            write(linha_texto, string'("Op_icode para a pilha desconhecida."));
          writeline(output, linha_texto);
        end case;
        if pont_pilha <= 23 then A_Full <= '0';
          else A_Full <= '1';
        end if;
      end if;
  end if;
  tos_bus <= pilha(pont_pilha);
end process p_pilha;
--}}}

--{{{  incrementa_y: process(Reset, Hold, Cp)
```

```
-- Registo de incremento do endereco de saida. Quando Reset esta activo, este
-- registo fica com o valor 1, assumindo que y_aux_bus (e portanto Y tem
-- nesse caso o valor 0, o que e' garantido pelo processo que faz a gestao
-- do multiplexer de saida. Quando o sinal Hold esta activo nao faz nada e
-- aguarda que ele fique inactivo. Finalmente, para cada bordo ascendente
-- do clock o registo fica com y_aux_bus+1.
incrementa_y: process(Reset, Hold, Cp)
begin
  if Reset = '0' then inc_bus <= 1;
   elsif Hold = '1' then NULL;
    elsif ((Cp'event) and (Cp = '1')) then inc_bus <= y_aux_bus + 1;
  end if;
end process incrementa_y;
--}}}

--{{{  p_contador: process(Hold, Cp)
-- Registo do contador para loops. Instrucoes do descodificador: NOP, DE_D,
-- DE_TOS, DEC_CONT. Este processo nao precisa ser sensivel ao Reset, mas
-- apenas ao Hold, para o qual deve manter o seu estado interno.
p_contador: process(Hold, Cp)
variable result, contador: int_16;
variable linha_texto: line;
begin
  if (Hold = '1') then NULL;
   else
     if ((Cp'event) and (Cp = '1')) then
       case to_contador is
         when DE_TOS => contador := tos_bus;
         when DE_D => contador := d_bus;
         when DEC => contador := contador - 1;
         when NOP => NULL;
         when others =>
         write(linha_texto, string'("Contador Loop: instrucao desconhecida."));
         writeline(output, linha_texto);
       end case;
     end if;
  end if;
  cont_bus <= contador;
end process p_contador;
--}}}

--{{{  p_comparador: process(Reset, Hold, Cp, y_aux_bus)
-- Registo do comparador para BP. Instrucoes do descodificador: NOP, DE_D,
-- DESAUT. Este processo e' sensivel ao Reset e Hold. Para o Reset faz a saida
-- Equal=0 (inactiva) e desautoriza comparacoes. Para o Hold apenas mantem o
-- seu estado interno e forca a saida Equal a highz (o que alias tambem sucede
-- durante o Reset). Quando o registo e' carregado (DE_D) e' autorizada a
-- operacao de comparacao.
p_comparador: process(Reset, Hold, Cp, y_aux_bus)
variable linha_texto: line;
variable en_comparador: boolean := FALSE;
variable comparador: int_16;
begin
  if Reset = '0' then
    Equal <= '0';
    en_comparador := FALSE;
   elsif Hold = '1' then NULL;
    else
      if (Cp'event and (Cp = '1')) then
```

```
            case to_comparador is
              when DE_D => comparador := d_bus;
                           en_comparador := TRUE;
              when DESAUT => en_comparador := FALSE;
              when NOP => NULL;
              when others =>
            write(linha_texto, string'("Comparador: instrucao desconhecida"));
            writeline(output, linha_texto);
            end case;
          end if;
          if ((en_comparador = TRUE) and (y_aux_bus = comparador)) then
            Equal <= '1';
           else
             Equal <= '0';
          end if;
      end if;
    end process p_comparador;
    --}}}

    --{{{  yout_drv: process(Reset, Hold, y_aux_bus)
    yout_drv: process(Reset, Hold, y_aux_bus)
    begin
      if ((Hold = '1') or (Reset = '0')) then  -- Se Hold ou Reset esta' activo
        NULL; -- Y <= VAL_Z after to_highz_delay;       -- faz Y=zzz...
       else
         Y <= y_aux_bus;
      end if;
    end process yout_drv;
    --}}}

  end behavioral;

--}}}
```

6. Micromemória

```
-- Modelo para a micromemoria:
-- Autor: Henrique Santos  Data: Junho, 94

use work.gl_pack.all;
use STD.TEXTIO.all;

-- Cada celula da micromemoria e' um record do tipo "mm_instr"
-- A micromemoria e' um array de celulas acima referidas, de dimensao
-- definida pela gama de valores do subtipo "mm_end".
-- Do ponto de vista do sistema esta memoria e' partilhada. Nao e' da
-- responsabilidade deste modulo assegurar-se da integridade dos sinais
-- de controlo, enderecos e dados. De notar que o sequenciador nao tem
-- capacidade para gerar os sinais de controlo, apenas lendo a memoria.
-- Assim sendo, tera' que ser o Host a assegurar o correcto funcionamento
-- dos sinais de controlo.

entity mm is
   port (endereco: in mm_end;
         dado: inout mm_instr;
         notcs, notrd, notwr: in bit);
end mm;

-- A arquitectura implementada nao coloca as linhas de dados em
-- alta-impedancia, uma vez que essa funcao nao existe a este nivel. Se a
-- memoria nao esta seleccionada ou se chegar um comando errado os sinais
-- "dados" matem o seu anterior valor. No ultimo caso e' enviada uma mensagem
-- de erro.
architecture behavior of mm is
  type mm_bloco is array (mm_end'low to mm_end'high) of mm_instr;
  begin
    process (endereco, dado, notcs, notrd, notwr)
      variable memoria: mm_bloco;
      variable celula: mm_instr;
      variable inicio: boolean := TRUE;
      variable pont: integer;
      variable temp: bit_vector(0 to 1);
      file TESTVEC: text is "gl_prog.txt"; -- Ficheiro de microprograma
      variable prog_linha: line;           -- variaveis auxiliares para
      variable aux, aux_end: integer;      -- acesso ao ficheiro
      variable aux_linha, temp_linha: line;
      variable arg_ok: boolean;
      -- Rotinas de escrita e leitura na memoria.
      -- Neste caso as rotinas sao tao simples que podiam ser removidas. No
      -- entanto a sua utilizacao pode trazer vantagens futuras!
      procedure le_mm (variable memoria: in mm_bloco;
                       variable endereco: in mm_end;
                       variable val: out mm_instr);
      procedure wr_mm (variable memoria: inout mm_bloco;
                       variable endereco: in mm_end;
                       variable val: in mm_instr);
      --{{{  procedure le_mm ()
      procedure le_mm (variable memoria: in mm_bloco;
                        variable endereco: in mm_end;
                        variable val: out mm_instr) is
      begin
        val := memoria(endereco);
      end le_mm;
```

```
--}}}
--{{{ procedure wr_mm ()
procedure wr_mm (variable memoria: inout mm_bloco;
                 variable endereco: in mm_end;
                 variable val: in mm_instr) is
begin
  memoria(endereco) := val;
end wr_mm;
--}}}
begin
   if inicio then
      --{{{  Bloco de inicializacao da memoria e variaveis
      -- A micromemoria vai ser inicializada a partir de um ficheiro que
      -- contem as microinstrucoes a carregar. O formato do ficheiro
      -- deve seguir aquele que esta definido para o assembler (ver
      -- documentacoa especifica).

      while not (endfile(TESTVEC)) loop
      --{{{  Leitura de uma linha de codigo
      --{{{  Endereco da micromemoria
      readline(TESTVEC, prog_linha);
      read(prog_linha, aux_end);
      -- read(prog_linha, aux_end, arg_ok);
      -- assert arg_ok
      --   report "Campo do endereco!"
      --     severity ERROR;
      write(temp_linha, "Endereco: ", LEFT, 0);
      write(temp_linha, aux_end, LEFT,0);
      writeline(OUTPUT, temp_linha);
      assert ((aux_end >= mm_end'low) and (aux_end <= mm_end'high))
        report "Endereco fora da gama!"
          severity ERROR;
      --}}}

      --{{{  Instrucao para o Sequenciador
      read(prog_linha, aux);
      -- read(prog_linha, aux, arg_ok);
      -- assert arg_ok
      --   report "Leitura da instrucao do Sequenciador-1!"
      --     severity ERROR;
      write(temp_linha, "Instr Seq: ", LEFT, 0);
      write(temp_linha, aux, LEFT,0);
      writeline(OUTPUT, temp_linha);
      memoria(aux_end).Seqi.instr := seq_instr_type'val(aux);
      read(prog_linha, aux);
      -- read(prog_linha, aux, arg_ok);
      -- assert arg_ok
      --   report "Leitura da instrucao do Sequenciador-2!"
      --     severity ERROR;
      write(temp_linha, "Instr Qualif: ", LEFT, 0);
      write(temp_linha, aux, LEFT,0);
      writeline(OUTPUT, temp_linha);
      memoria(aux_end).Seqi.qualif := qualif_type'val(aux);
      read(prog_linha, aux);
      -- read(prog_linha, aux, arg_ok);
      -- assert arg_ok
      --   report "Leitura da instrucao do Sequenciador-3!"
      --     severity ERROR;
      write(temp_linha, "Instr data: ", LEFT, 0);
```

```
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
assert ((aux >= int_16'low) and (aux <= int_16'high))
  report "Operando para Sequenciador fora de gama!"
    severity ERROR;
memoria(aux_end).Seqi.data := int_16'val(aux);
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do Sequenciador-4!"
--     severity ERROR;
write(temp_linha, "Instr Sfs: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
memoria(aux_end).Seqi.Sfs := aux;
--}}}

--{{{  Instrucao para o AGL
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do AGL-1!"
--     severity ERROR;
write(temp_linha, "AGL instr: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
memoria(aux_end).DSi.instr := agl_instr_type'val(aux);
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do AGL-2!"
--     severity ERROR;
write(temp_linha, "AGL ctr: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
memoria(aux_end).DSi.ctr := agl_ctr_type'val(aux);
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do AGL-3!"
--     severity ERROR;
write(temp_linha, "AGL data: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
assert ((aux >= ds_end'low) and (aux <= ds_end'high))
  report "Operando para AGL fora da gama"
    severity ERROR;
memoria(aux_end).DSi.data := ds_end'val(aux);
--}}}

--{{{  Instrucao para a SU
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do SU!"
--     severity ERROR;
write(temp_linha, "SU instr: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
```

```
memoria(aux_end).SUi := su_instr'val(aux);
--}}}

--{{{  Instrucao para o TR
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura da instrucao do TR!"
--     severity ERROR;
write(temp_linha, "TR instr: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
memoria(aux_end).TRi := tr_instr'val(aux);
--}}}

--{{{  Instrucao para o campo SU_TRsel
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura do selector SU_TR!"
--     severity ERROR;
write(temp_linha, "SU_TRsel: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
if (aux = 0) then memoria(aux_end).SU_TRsel := '0';
  else memoria(aux_end).SU_TRsel := '1';
end if;
--}}}

--{{{  Instrucao para os campos Sreq, EOC e Ss
read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura de SReq!"
--     severity ERROR;
write(temp_linha, "SReq: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
if (aux = 0) then memoria(aux_end).Sreq := '0';
  else memoria(aux_end).Sreq := '1';
end if;

read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura de EOC!"
--     severity ERROR;
write(temp_linha, "EOC: ", LEFT, 0);
write(temp_linha, aux, LEFT,0);
writeline(OUTPUT, temp_linha);
if (aux = 0) then memoria(aux_end).EOC := '0';
  else memoria(aux_end).EOC := '1';
end if;

read(prog_linha, aux);
-- read(prog_linha, aux, arg_ok);
-- assert arg_ok
--   report "Leitura de Ss!"
--     severity ERROR;
```

```
            write(temp_linha, "Ss: ", LEFT, 0);
            write(temp_linha, aux, LEFT,0);
            writeline(OUTPUT, temp_linha);
            if (aux = 0) then memoria(aux_end).Ss := '0';
              else memoria(aux_end).Ss := '1';
            end if;
            --}}}
            --}}}
            end loop;

            inicio := FALSE;
            --}}}
           else
             if (notcs = '0') then
               pont := endereco;
               temp := notrd & notwr;
               case (temp) is
                 when "11" => NULL;
                 when "01" => le_mm(memoria, pont, celula);
                              dado <= celula;
                 when "10" => celula := dado;
                              wr_mm(memoria, pont, celula);
                 when "00" => assert FALSE
                                report "MicroMemoria: Operacao invalida!"
                                  severity ERROR;
               end case;
             end if;
         end if;
    end process;
end behavior;
```

```
-- ------------------------------------------------------------------------
-- Bus Arbiter | by: H. Santos | Data: Junho 94
-- Revisao 1.0 Junho/95
-- Arbitrador do barramento para o controlador do GLiTCH. O modelo a seguir
-- apresentado e' o obtido a partir da descricao do manual. Os detalhes da
-- especificacao sao ai apresentados
-- O sinal
-- ------------------------------------------------------------------------
entity bus_arb1 is
  port (Run, BusReq,            -- Do registo de controlo do Transputer
        Ss, Finish, SC_Req,     -- da Micromemoria
        Equal,                  -- do Sequenciador
        Reset, Clock: in bit;   -- Globais do sistema
        SuspendGL, BusTrnotLoc1, -- Controlo dos drivers dos barramentos
        BusToTr,
        BusTrnotLoc2,           -- este tb. serve de flag -> BusGrant
        SC_ReqOut, EqualOut,    -- Flags para o Transputer
        FinishOut, SsOut: out bit);
end bus_arb1;

architecture behavior of bus_arb1 is
-- O Bus Arbiter consiste 1 controlador central responsavel pela funcao de
-- arbitro em si e de 5 controladores, cada um responsavel pelo funcionamento
-- dos sinais que podem dar origem a uma operacao de transferencia de
-- controlo do barramento. Assim teremos seis descricoes de maquinas
-- sincronas que cooperam atraves dos sinais BusToTr
  type Main_asm is (Normal_OP, Phase1, Phase2, RetPhase1, RetPhase2);
  type Equal_asm is (NoEqual, EqualRun);
  type Finish_asm is (NoFinish, FinishRun);
  type Ss_asm is (NoSs, SsRun);
  type SC_Req_asm is (NoSC_Req, SC_ReqRun);
  type BusReq_asm is (NoBusReq, BusReqRun);
  type Run_asm is (WaitRun, StdByRun, ActiveRun, StdBynoRun);

  signal ClrFlags: bit; -- Interliga o controlador principal com os outros
  -- O sinal BusTrnotLoc1 e' de saida. Tais sinais nao podem ser usados em
  -- expressoes, excepto as da sua atribuicao. Ora aquele sinal vai ser
  -- necessario internamente em todos os controladores, excepto o principal.
  -- O mesmo se aplica ao sinal BuToTr. BusReqOut e' um sinal interno.
  -- A alternativa era declarar os respectivos "ports" no modo INOUT.
  signal BusTrnotLoc1aux, BusToTraux, BusReqOut, RunSync: bit;

  begin
    BusToTraux <= Equal or Finish or Ss or SC_Req or BusReqOut
                  or BusTrnotLoc1aux;
    BusToTr <= BusToTraux;
    --{{{  Main_p: process
    Main_p: process (clock, Reset)
      variable estado: Main_asm := Normal_OP;
      variable inicio: boolean := TRUE;
      begin
        if ((Reset = '1') or inicio) then
          estado := Phase2; BusTrnotLoc1 <= '1'; BusTrnotLoc1aux <= '1';
          SuspendGL <= '1'; ClrFlags <= '0'; BusTrnotLoc2 <= '1';
          inicio := FALSE;
         else
           if (clock'event and (clock = '1')) then
```

```
        case estado is
          when Normal_OP => if BusToTraux = '1' then
                              estado := Phase1;
                              BusTrnotLoc1 <= '1';
                              BusTrnotLoc1aux <= '1';
                              SuspendGL <= '1';
                            end if;
          when Phase1 => estado := Phase2;
                         BusTrnotLoc1 <= '1';
                         BusTrnotLoc1aux <= '1';
                         SuspendGL <= '1';
                         BusTrnotLoc2 <= '1';
          when Phase2 => if RunSync = '1' then
                           estado := RetPhase1;
                           BusTrnotLoc1 <= '1';
                           BusTrnotLoc1aux <= '1';
                           SuspendGL <= '1';
                           ClrFlags <= '1';
                           BusTrnotLoc2 <= '0';
                         end if;
          when RetPhase1 => if RunSync = '0' then
                              estado := RetPhase2;
                              BusTrnotLoc1 <= '1';
                              BusTrnotLoc1aux <= '1';
                              ClrFlags <= '1';
                              SuspendGL <= '0';
                            end if;
          when RetPhase2 => estado := Normal_OP;
                            BusTrnotLoc1 <= '0';
                            BusTrnotLoc1aux <= '0';
                            ClrFlags <= '0';
          end case;
       end if;
    end if;
  end process;
--}}}
--{{{  Equal_p: process
Equal_p: process (clock, Reset)
  variable estado: Equal_asm := NoEqual;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := NoEqual; EqualOut <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoEqual => if (BusTrnotLoc1aux = '0') and (Equal = '1') then
                             estado := EqualRun; EqualOut <= '1';
                           end if;
           when EqualRun => if ClrFlags = '1' then
                              estado := NoEqual; EqualOut <= '0';
                            end if;
         end case;
       end if;
    end if;
  end process;
--}}}
--{{{  Finish_p: process
```

```
Finish_p: process (clock, Reset)
  variable estado: Finish_asm := NoFinish;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := NoFinish; FinishOut <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoFinish => if (BusTrnotLoc1aux = '0') and (Finish = '1') then
                              estado := FinishRun; FinishOut <= '1';
                            end if;
           when FinishRun => if ClrFlags = '1' then
                               estado := NoFinish; FinishOut <= '0';
                             end if;
         end case;
       end if;
    end if;
  end process;
--}}}
--{{{  Ss_p: process
Ss_p: process (clock, Reset)
  variable estado: Ss_asm := NoSs;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := NoSs; SsOut <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoSs => if (BusTrnotLoc1aux = '0') and (Ss = '1') then
                          estado := SsRun; SsOut <= '1';
                        end if;
           when SsRun => if ClrFlags = '1' then
                           estado := NoSs; SsOut <= '0';
                         end if;
         end case;
       end if;
    end if;
  end process;
--}}}
--{{{  SC_Req_p: process
SC_Req_p: process (clock, Reset)
  variable estado: SC_Req_asm := NoSC_Req;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := NoSC_Req; SC_ReqOut <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoSC_Req => if (BusTrnotLoc1aux = '0') and (SC_Req = '1') then
                              estado := SC_ReqRun; SC_ReqOut <= '1';
                            end if;
           when SC_ReqRun => if ClrFlags = '1' then
                               estado := NoSC_Req; SC_ReqOut <= '0';
```

```
                              end if;
          end case;
        end if;
      end if;
    end process;
--}}}
--{{{  BusReq_p: process
BusReq_p: process (clock, Reset)
  variable estado: BusReq_asm := NoBusReq;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := NoBusReq; BusReqOut <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when NoBusReq => if (BusTrnotLoc1aux = '0') and (BusReq = '1') then
                              estado := BusReqRun; BusReqOut <= '1';
                            end if;
           when BusReqRun => if ClrFlags = '1' then
                               estado := NoBusReq; BusReqOut <= '0';
                             end if;
         end case;
       end if;
    end if;
  end process;
--}}}
--{{{  Run_p: process
Run_p: process (clock, Reset)
  variable estado: Run_asm := WaitRun;
  variable inicio: boolean := TRUE;
  begin
    if ((Reset = '1') or inicio) then
      estado := WaitRun; RunSync <= '0';
      inicio := FALSE;
     else
       if (clock'event and (clock = '1')) then
         case estado is
           when WaitRun =>  if Run = '1' then
                               estado := StdByRun;
                            end if;
           when StdByRun => if Run = '1' then
                               estado := ActiveRun; RunSync <= '1';
                             else
                                estado := WaitRun;
                            end if;
           when ActiveRun => if Run = '0' then
                                estado := StdBynoRun;
                             end if;
           when StdBynoRun => if Run = '0' then
                         estado := WaitRun; RunSync <= '0';
                       else
                          estado := ActiveRun;
                     end if;
         end case;
       end if;
    end if;
  end process;
```

```
  --}}}
end behavior;
```

```
-- Host Transputer Interface.
-- Autor: Henrique Santos   Data: Julho de 1994
--{{{  Intruducao
-- Modulo de especificacao do interface entre o Host Transputer e o sistema
-- GLiTCH. Pela sua natureza, a maior parte dos algoritmos aqui especifica-
-- dos sao implementados por software. Os sinais envolvidos (portas de I/O
-- e ligacoes ao barramento do Transputer) sao os seguintes:
--   dr[31 DOWNTO 0] -> Barramento do D. Routing, D. Store e Sc. Unit
--   dr_notcs, dr_notrd
--   dr_notwr        -> sinais de interface com o Data Store
-- No sistema real o DS esta' dividido em dois bancos, exigindo multiplos
-- sinais de CS, RD e WR, que sao partilhados com o AGL. No modelo, basta
-- um sinal para cada uma daquelas funcoes, o qual activara' ambos os bancos
-- simultaneamente. E' da responsabilidade do modelo "desligar" estes sinais
-- quando nao estiver com o comando dos componetes do GLiTCH.
--
-- NOTA: nesta versao o HT nao vai aceder a micromemoria
--   mm[95 DOWNTO 0] -> Barr. da micromemoria
--   mm_notcs, mm_notrd
--   mm_notwr        -> sinais de interface com a Micromemoria
-- Aplica-se a esta memoria os mesmos principios descritos para o DS, com a
-- excepcao que nenhum outro componente no controlador do GLiTCH comanda
-- estas linhas, pelo que sera' da responsabilidade do modelo colocar a
-- micro-memoria em leitura, sempre que nao estiver a comandar o sistema.
--   Equal, Finish   -> Entradas; associadas a flags para o HT
--   AFull, SC_Req
--   SDone, SStep
--   BusTrnotLoc2 (BusGrant)
-- A semantica associada a cada uma destas flags encontra-se completamente
-- descrita no relatorio tecnico do Controlador do GLiTCH. Apenas se deve
-- salientar aqui um aspecto. Com excepcao dos sinais AFull e SDone todas
-- as flags ficam activas ate' este modulo executar a accao pertinente.
-- AFull e SDone servem apenas para informar o HT acerca das respectivas
-- funcoes, nao requerem nenhuma accao imediata e nao e' garantido o tempo
-- de activacao minimo para que esses sinais sejam detectados pelo
-- Transputer (isto no ambiente real). Isto significa que estes sinais
-- deverao activar dentro do HT uma unidade de memoria (FF) atraves do
-- bordo ascendente. A implementacao nao corresponde exactamente a esta
-- especificacao, por limitacoes do hardware usado e exige um sinal de
-- interface adicional...
--   BusReq, Run     -> Saidas; associadas a accoes de comando do HT
--   Sw_Ini
-- A semantica associada a cada uma destas flags encontra-se completamente
-- descrita no relatorio tecnico do Controlador do GLiTCH.
--   Suspende        -> Entrada; Pedido de suspensao externo.
--   SuspOut         -> Saida; associado ao anterior
-- Este sinal nao existe no sistema real e permite apenas modelar um pedido
-- externo (possivelmente originado pelo utilizador) para suspender a
-- execucao do Sistema GLiTCH. A partir deste sinal o modulo HT vai activar
-- o sinal BusReq para executar a accao. Enquanto o sinal Suspende estiver
-- activo o GLiTCH nao recomecara' a executar. O sinal SuspOut confirma que
-- o pedido foi aceite.
--}}}

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use WORK.gl_pack.all;
```

```
--{{{  entity ht is
entity ht is
  port(dr: inout std_logic_vector(31 downto 0);
       -- mm: inout mm_instr; para ja' o HT nao tem acesso a micromemoria
       dr_notcs: out std_logic_vector(1 downto 0);
       dr_notrd, dr_notwr: out std_logic;
       mm_notcs, mm_notrd, mm_notwr: out bit; -- Sinais de controlo da mm
       BusReq, Run, Sw_Ini: out bit;          -- logicamente sao gerados aqui
       Equal, Finish, AFull, SC_Req, SDone, SStep,
       BusTrnotLoc2: in bit);
end ht;
--}}}

-- A arquitectura seguinte corresponde a uma descricao comportamental, mas
-- na qual as flags serao verificadas sequencialmente. A alternativa seria
-- fazer corresponder a cada flag um processo, modelando assim um sistema
-- de interrupcoes. A segunda hipotese parece corresponder a um modelo que
-- mais facilmente satisfaz a especificacao. Contudo, nao corresponde ao
-- sistema real, ja' que nao foi utilizada a (fraca) facilidade de
-- interrupcao do Transputer para nao complicar o hardware.

--{{{  architecture behav_spool of ht is
architecture behav_spool of ht is
constant HighZ32: std_logic_vector(31 downto 0) :=
                  ('Z','Z','Z','Z','Z','Z','Z','Z',
                   'Z','Z','Z','Z','Z','Z','Z','Z',
                   'Z','Z','Z','Z','Z','Z','Z','Z',
                   'Z','Z','Z','Z','Z','Z','Z','Z');
begin
  -- O processo que verifica o estado das flags deve executar continuamente
  -- a semelhanca do que faria o programa no transputer. Por esse motivo
  -- nao dispoe de "sensitivity list" e a instrucao "WAIT FOR timeout" final,
  -- necessaria para evitar o "deadlock", indica como timeout 0 ns, ou seja,
  -- execucao continua.

  spool: process
  variable inicio: boolean := TRUE;
  variable Suspende: bit; -- Variavel interna para permitir simular um pedido
                          -- de suspensao a partir do HT. E' interna ao processo
                          -- e portanto so' pode ser activada utilizando
                          -- facilidades do simulador em "run-time".
  begin
    --{{{  Inicializacao
    if inicio then
       dr_notcs <= "11"; dr_notrd <= '1'; dr_notwr <= '1';
       mm_notcs <= '1'; mm_notrd <= '1'; mm_notwr <= '1';
       dr <= HighZ32;

       Suspende := '1'; -- Esta activo o pedido de suspensao. GLiTCH nao corre...

       -- mm <= HighZ32 & HighZ32 & HighZ32;
       BusReq <= '0'; Run <= '0'; Sw_Ini <= '0';
       -- wait until BusTrnotLoc2 = '1'; -- No arranque o controlo e' do HT
       -- Run <= '1';                    -- Faz executar
       -- wait until BusTrnotLoc2 = '0'; -- mas completa protocolo com o arbitrador
       -- Run <= '0';                    -- dos barramentos
       inicio := FALSE;
    end if;
```

```
--}}}
if Equal = '1' then
   --{{{  Conjunto de accoes associadas ao "Break Point"
   putline("Atingido um Break Point.");
   assert BusTrnotLoc2 = '1'
     report "Transferencia de controlo incompleta - BusGrant = 0 !"
       severity ERROR;

   -- Inserir aqui qualquer accao

   Run <= '1';
   wait until BusTrnotLoc2 = '0';
   Run <= '0';
   --}}}
end if;
if Finish = '1' then
   --{{{  Conjunto de accoes associadas ao "Fim de Microprograma"
   putline("Atingido o Fim do Microprograma.");
   assert BusTrnotLoc2 = '1'
     report "Transferencia de controlo incompleta - BusGrant = 0 !"
       severity ERROR;

   -- Inserir aqui qualquer accao

   Run <= '1';
   wait until BusTrnotLoc2 = '0';
   Run <= '0';
   --}}}
end if;
if AFull = '1' then
   --{{{  Conjunto de accoes associadas a "Stack quase cheia"
   putline("Atencao, a Stack esta quase cheia!");

   -- Inserir aqui qualquer accao

   --}}}
end if;
if SC_Req = '1' then
   --{{{  Conjunto de accoes associadas ao "Pedido de exe. de op. Escalar"
   putline("A atender um pedido de execucao de operacao Escalar...");
   assert BusTrnotLoc2 = '1'
     report "Transferencia de controlo incompleta - BusGrant = 0 !"
       severity ERROR;

   -- Inserir aqui qualquer accao

   Run <= '1';
   wait until BusTrnotLoc2 = '0';
   Run <= '0';
   --}}}
end if;
if SStep = '1' then
   --{{{  Conjunto de accoes associadas ao "Single Step"
   putline("Flag de Single Step activa.");
   assert BusTrnotLoc2 = '1'
     report "Transferencia de controlo incompleta - BusGrant = 0 !"
       severity ERROR;

   -- Inserir aqui qualquer accao
```

```
      Run <= '1';
      wait until BusTrnotLoc2 = '0';
      Run <= '0';
      --}}}
    end if;
    if SDone = '1' then
      --{{{  Conjunto de accoes associadas ao "Swap Done do VSR"
      putline("O VSR terminou uma operacao de Swap.");

      -- Inserir aqui qualquer accao

      --}}}
    end if;
    if Suspende = '1' then
      --{{{  Conjunto de accoes associadas ao "Pedido externo de suspensao"
      putline("Pedido de Suspensao...");
      BusReq <= '1';
      wait until BusTrnotLoc2 = '1';
      BusReq <= '0';
      putline(" Aceite.");

      -- Inserir aqui qualquer accao

      Suspende := '0'; -- Desactiva o pedido de suspensao
      Run <= '1';
      wait until BusTrnotLoc2 = '0';
      Run <= '0';
      --}}}
    end if;

    WAIT FOR 0 ns;
  END process spool;

END behav_spool;
--}}}
```

9. Modelo estrutural do controlador

```
-- Descricao estrutural para o sistema GLiTCH
-- Autor: Henrique Santos   Data: Julho de 94

--{{{  Descricao:
-- Esta descricao estrutural vai servir como "test bench" para o modelo
-- do GLiTCH. Todos os algoritmos de geracao de vectores de teste e de
-- verificacao devem ser incluidos e descritos neste modulo.
--}}}

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use WORK.gl_pack.all;
use WORK.all;

entity test_bench is
end test_bench;

architecture Sistema_GLiTCH of test_bench is
  --{{{  component MicroMemoria
  component MicroMemoria
  port (endereco: in mm_end;
        dado: inout mm_instr;
        notcs, notrd, notwr: in bit);
  end component;
  for U1:Micromemoria use entity MM(BEHAVIOR);
  --}}}
  --{{{  component Sequenciador
  component Sequenciador
  port (I: seq_instr;                  -- record...
        T: in bit_vector(11 downto 0);
        M0: in bit_vector (3 downto 0);
        Cp, Reset, Inten, Intr, Hold, OEd, Slave, CyIn, Fcont: in bit;
        Y: out mm_end;
        Equal, A_Full, Inta, S_Error: out bit);
  end component;
  for U2:Sequenciador use entity AM29C331(BEHAVIORAL);
  --}}}
  --{{{  component AddGenLogic
  component AddGenLogic
    port (Instruction: in agl_instr; -- "record"
          clock, write_pulse, BusTrnotLoc1, ForceHold: in bit;
          Addr_out: out ds_end;
          notCS: out std_logic_vector(1 downto 0);
          notOE, notWE: std_logic);
  end component;
  for U3:AddGenLogic use entity AGL_C(BEHAVIOR);
  --}}}
  --{{{  component DataStore
  component DataStore
    port (endereco: in ds_end;
          dado: inout std_logic_vector(31 downto 0);
          notrd, notwr: in std_logic;
          notcs: in std_logic_vector(1 downto 0) ); -- notcs(1) -> Mask Bank
  end component;                                    -- notcs(0) -> Data Bank
  for U4:DataStore use entity DS(BEHAVIOR);
  --}}}
  --{{{  component ScalarUnit
```

```
component ScalarUnit
  port (signal io_bus: inout std_logic_vector(31 downto 0);
        signal Instruction: in su_instr;
        signal Hold, clk, s_clk, BusTrnotLoc1: in bit;
        signal LSB_3: out bit_vector(2 downto 0);
        signal Zero, MSB: out bit);
end component;
for U5:ScalarUnit use entity SU_C(BEHAVIOR);
--}}}
--{{{  component TagRouter
component TagRouter
  port (signal Instruction: in tr_instr;
        signal TagUp, TagDown, notHold, Suspend, Clk: in bit;
        signal TagAbove, TagBellow: out bit);
end component;
for U6:TagRouter use entity TR_C(BEHAVIORAL);
--}}}
--{{{  component HostTransputer
component HostTransputer
  port(dr: inout std_logic_vector(31 downto 0);
       -- mm: inout mm_instr; para ja' o HT nao tem acesso a micromemoria
       dr_notcs: out std_logic_vector(2 downto 0);
       dr_notrd, dr_notwr: out std_logic;
       mm_notcs, mm_notrd, mm_notwr: out bit; -- Sinais de controlo da mm
       BusReq, Run, Sw_ini: out bit;          -- logicamente sao gerados aqui
       Equal, Finish, AFull, SC_Req, SDone, SStep,
       BusTrnotLoc2: in bit);
end component;
for U7:HostTransputer use entity HT(BEHAV_SPOOL);
--}}}
--{{{  component BusArbiter
component BusArbiter
  port (Run, BusReq,            -- Do registo de controlo do Transputer
        Ss, Finish, SC_Req,     -- da Micromemoria
        Equal,                  -- do Sequenciador
        Reset, Clock: in bit;   -- Globais do sistema
        SuspendGL, BusTrnotLoc1, -- Controlo dos drivers dos barramentos
        BusToTr,
        BusTrnotLoc2,           -- este tb. serve de flag -> BusGrant
        SC_ReqOut, EqualOut,    -- Flags para o Transputer
        FinishOut, SsOut: out bit);
end component;
for U8:BusArbiter use entity BUS_ARB1(BEHAVIOR);
--}}}
--{{{  Declaracoes de sinais internos
signal mm_in: mm_end;
signal mm_out: mm_instr;
signal mm_notcs, mm_notrd, mm_notwr: bit;
signal flags: bit_vector(15 downto 0); -- 15..4 sao as flags e 3..0
                                       -- barramento M0 do sequenciador
signal agl_out: ds_end;
signal ds_notcs: std_logic_vector(1 downto 0);
signal ds_notOE, ds_notWE: std_logic;
signal data_bus: std_logic_vector(31 downto 0);
signal TUp, TDn, TAb, TBl: bit;
signal BReq, Exec, st_BP, st_Fim, st_AF, st_SC_Req, st_VdSDone,
       st_SStep, st_BusOk: bit;
signal BP, SGl, TnotL1, Bus2Tr: bit;
signal clock, s_clock, wr_pulse, reset, notreset: bit;
```

```
signal nc1, nc2, nc3: bit; -- "not connected" para saidas nao usadas
signal gnd: bit := '0';
signal vcc: bit := '1';
--}}}

begin
  flags(0) <= gnd; flags(9) <= gnd; flags(10) <= gnd; -- Flags nao usadas

  --{{{  U1:Micromemoria
  U1:Micromemoria
    port map(mm_in, mm_out, mm_notcs, mm_notrd, mm_notwr);
  --}}}

  --{{{  U2:Sequenciador
  U2:Sequenciador
    port map(mm_out.Seqi, flags(11 downto 0), flags(6 downto 3), clock,
             notreset, gnd, gnd, TnotL1 , gnd, gnd, gnd, gnd, mm_in, BP,
             st_AF, nc1, nc2);
  --}}}

  --{{{  U3:AddGenLogic
  U3:AddGenLogic
    port map(mm_out.DSi, clock, wr_pulse, TnotL1, gnd, agl_out, ds_notcs,
             ds_notOE, ds_notWE);
  --}}}

  --{{{  U4:DataStore
  U4:DataStore
    port map(agl_out, data_bus, ds_notOE, ds_notWE, ds_notcs);
  --}}}

  --{{{  U5:ScalarUnit
  U5:ScalarUnit
    port map(data_bus, mm_out.SUi, mm_out.SU_TRsel, clock, s_clock, TnotL1,
             flags(5 downto 3), flags(2), flags(6));
  --}}}

  --{{{  U6:TagRouter
  U6:TagRouter
    port map(mm_out.TRi, TUp, TDn, mm_out.SU_TRsel, TnotL1, clock, TAb, TBl);
  --}}}

  --{{{  U7:HostTransputer
  U7:HostTransputer
    port map(data_bus, ds_notcs, ds_notOE, ds_notWE, mm_notcs, mm_notrd,
             mm_notwr, BReq, Exec, nc3, st_BP, st_Fim, st_AF, st_SC_Req,
             st_VdSDone, st_SStep, st_BusOk);
  --}}}

  --{{{  U8:BusArbiter
  U8:BusArbiter
    port map(Exec, BReq, mm_out.Ss, mm_out.EOC, mm_out.Sreq, BP, reset, clock,
             SGl, TnotL1, Bus2Tr, st_BusOk, st_SC_Req, st_BP, st_Fim,
             st_SStep);
  --}}}

  -- Bloco correspondente 'a placa do GLiTCH ?

  reset <= '0';
```

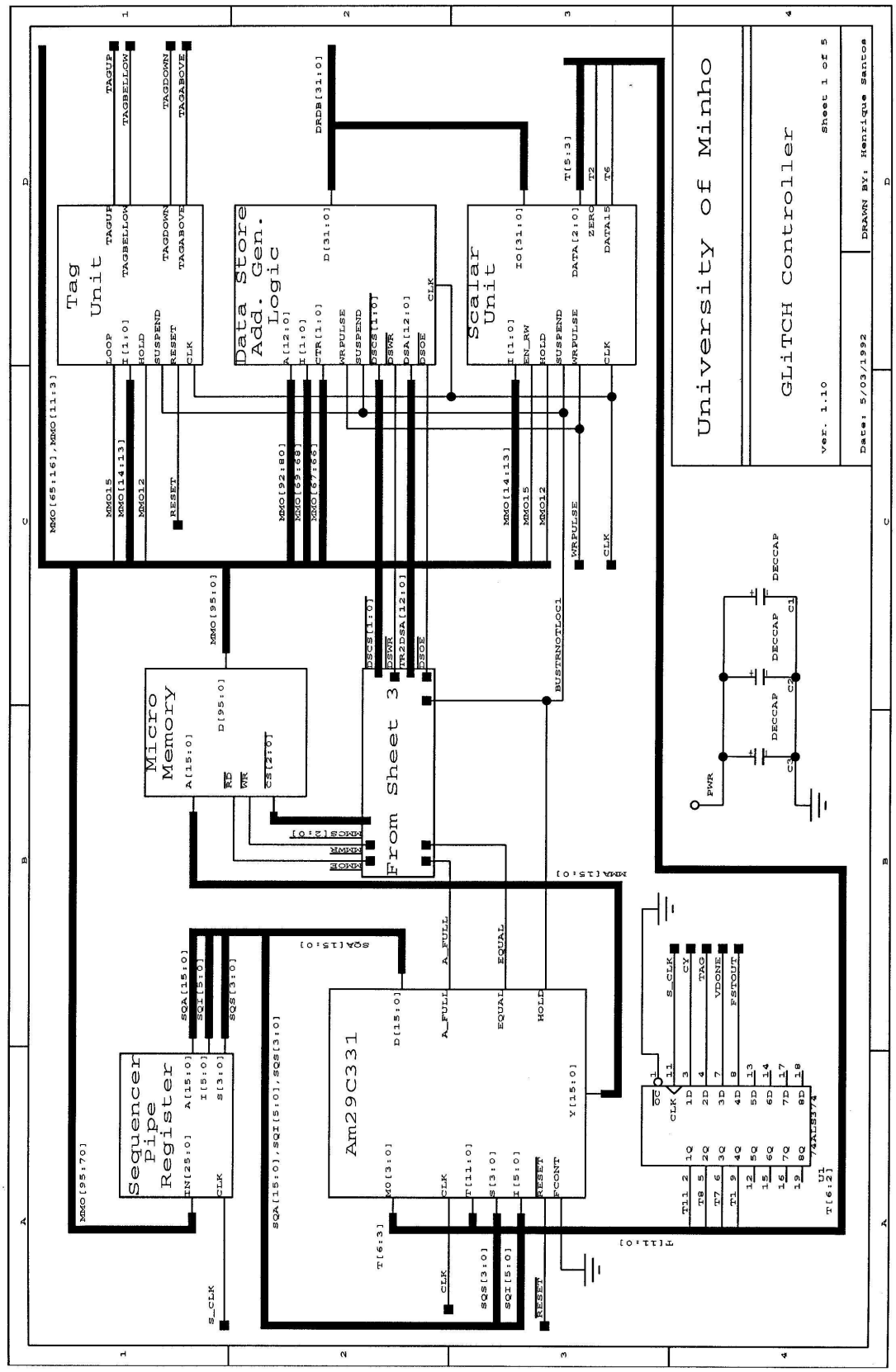
```
    notreset <= '1';
    clock <= not clock after 100 ns;
    s_clock <= clock when Bus2Tr = '0' else
               '0';
    wr_pulse <= not clock;  -- isto nao esta' correcto!
end Sistema_GLiTCH;
```

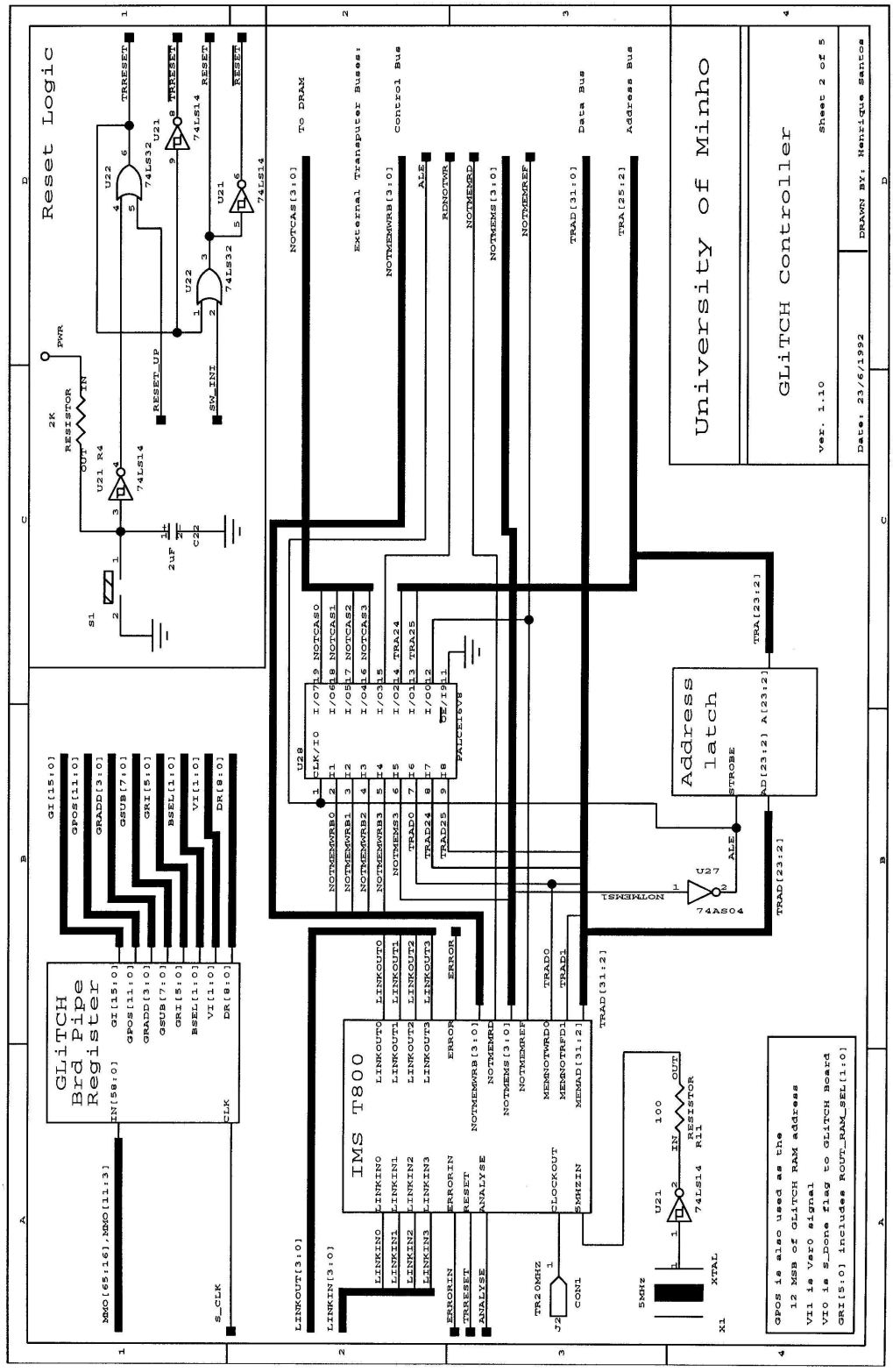
Apêndice B - Diagramas esquemáticos do controlador do GLiTCH

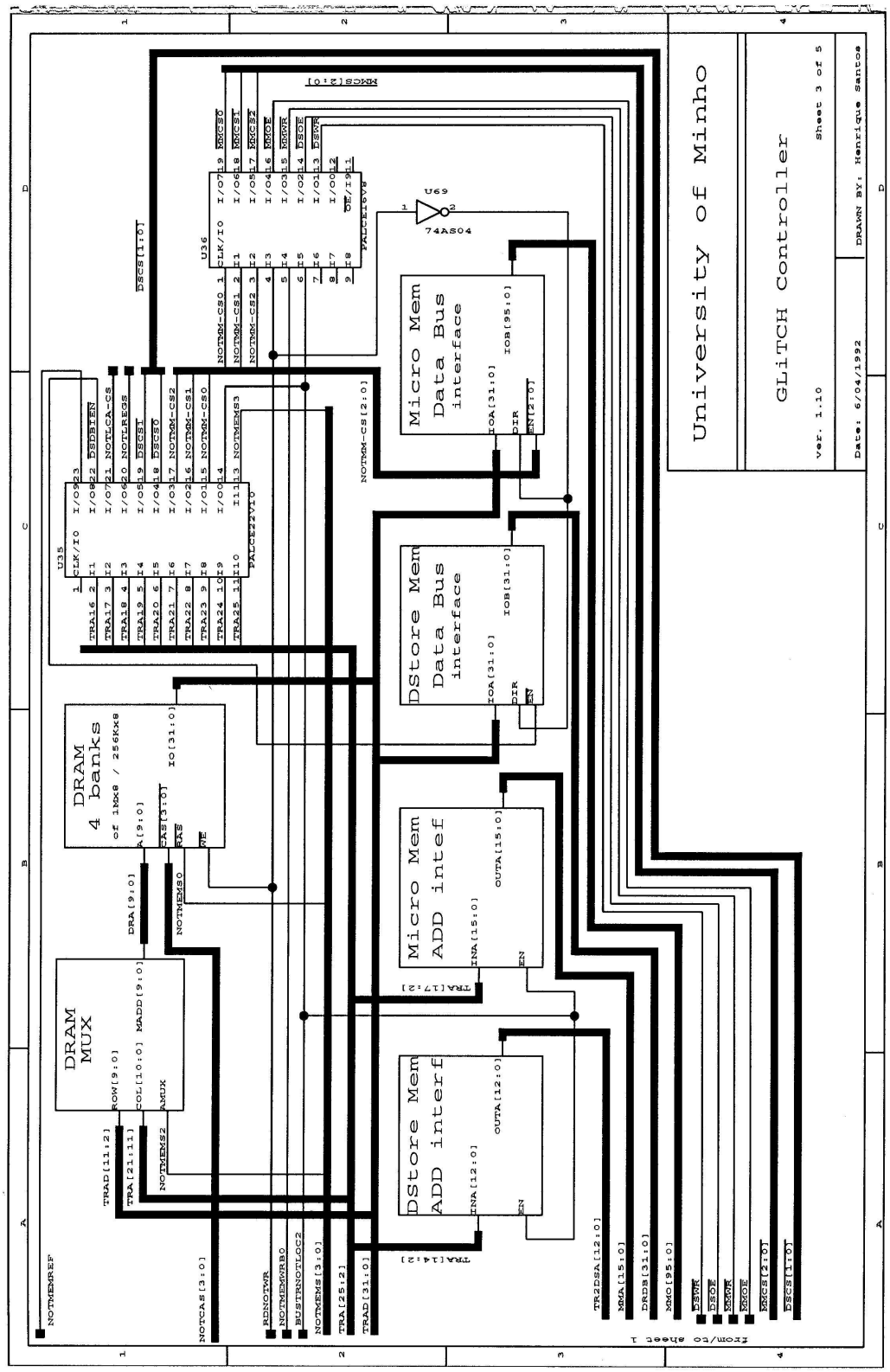
Índice

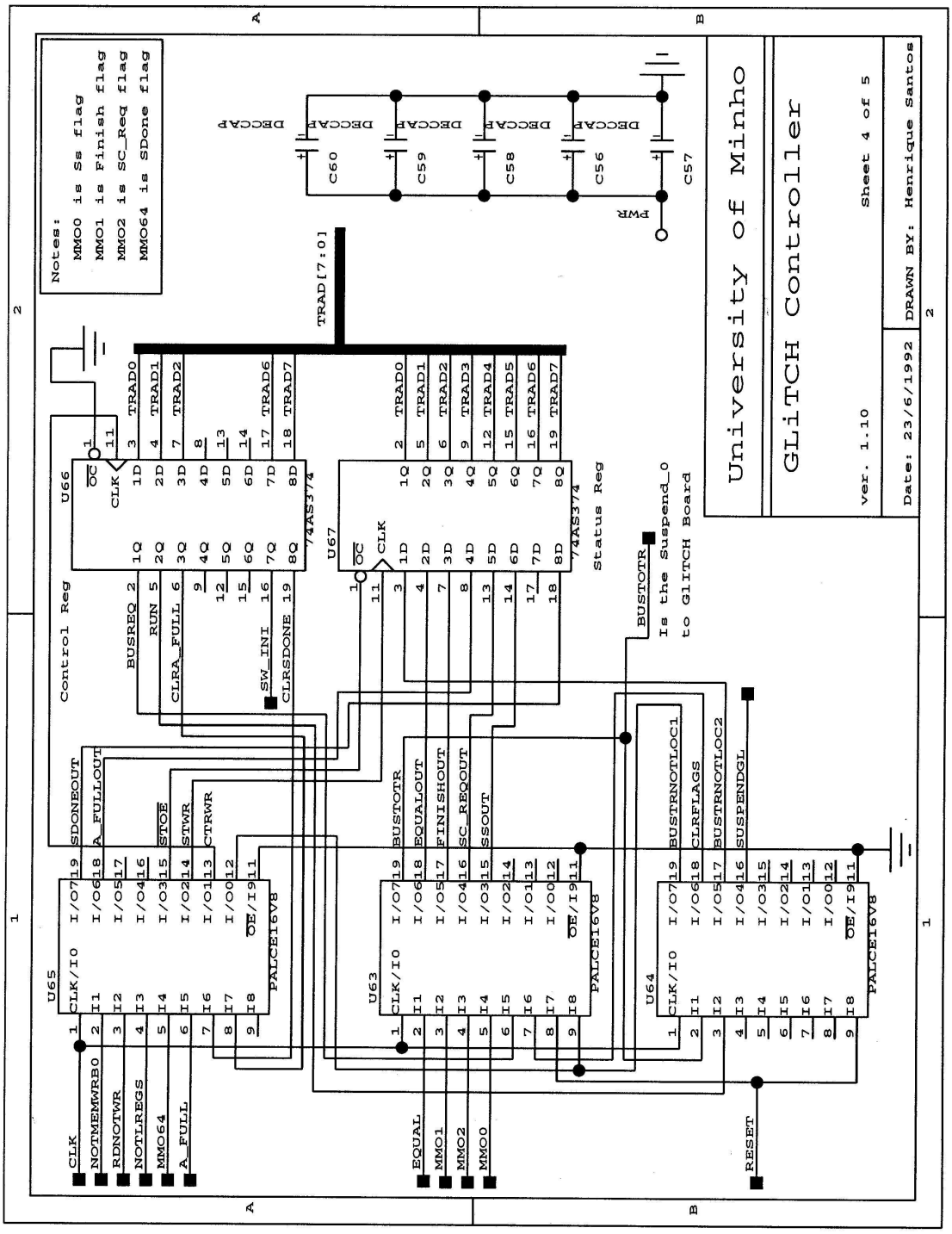
Controlador do GLiTCH - nível hierárquico superior ..... (5 folhas)
Sequenciador Am29C331 ..... (1 folha)
Registos de *pipeline* do sequenciador ..... (1 folha)
Micromemória ..... (1 folha)
Selector de *tags* ..... (1 folha)
Unidade de memória de padrões ..... (1 folha)
Unidade escalar ..... (1 folha)
Registos de *pipeline* para o módulo GLiTCH ..... (1 folha)
IMS T800-G20S - transputer hospedeiro ..... (1 folha)
- *Latch* de endereços ..... (1 folha)
- Multiplexador de endereços para a memória dinâmica ..... (1 folha)
- Memória dinâmica (4 bancos) ..... (1 folha)
- *Interface* com o barramento de dados da micromemória ..... (1 folha)
- *Interface* com o barramento de dados do DS ..... (1 folha)
- *Interface* com o barramento de endereços da micromemória ..... (1 folha)
- *Interface* com o barramento de endereços do DS ..... (1 folha)

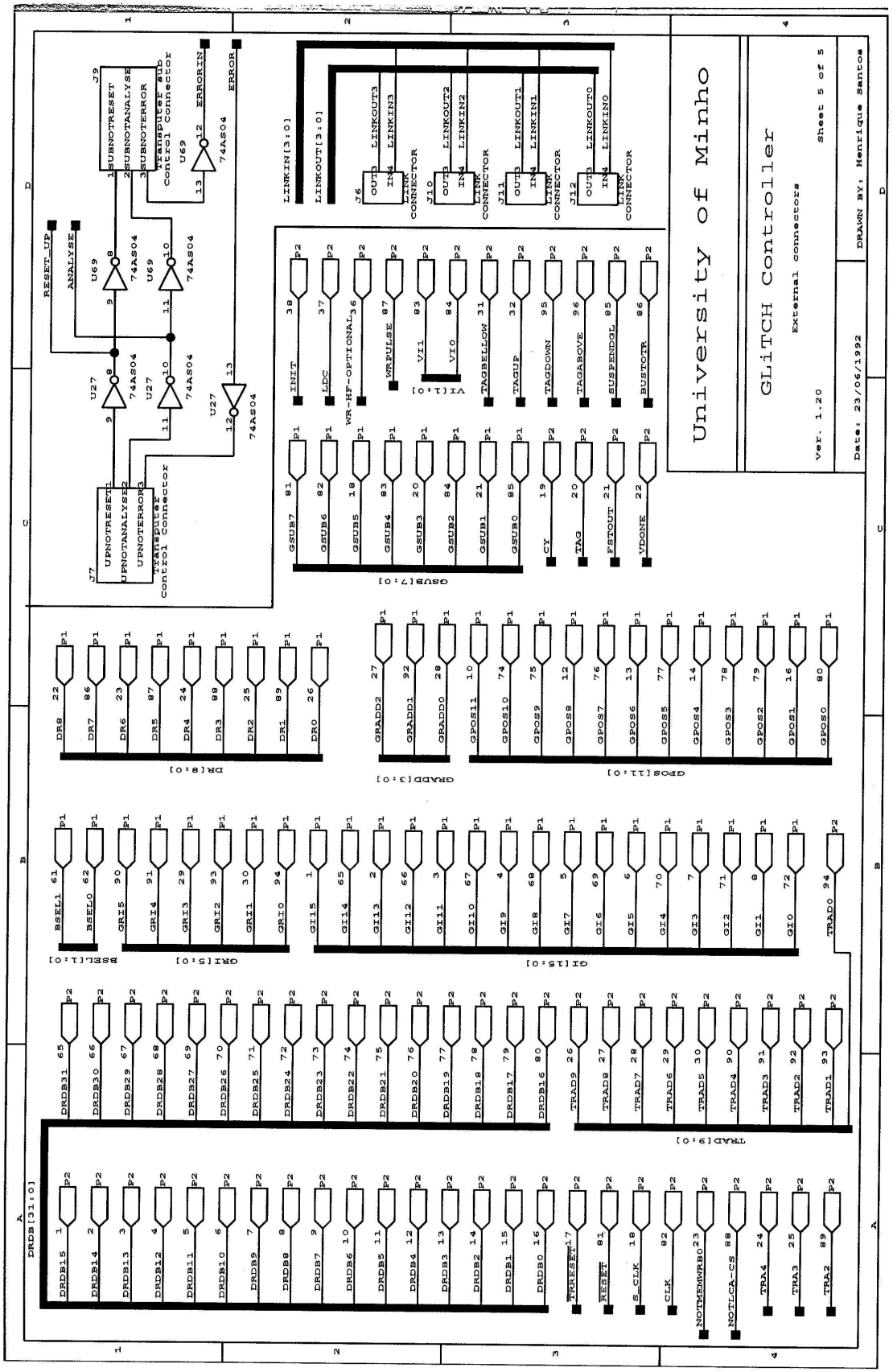
Distribuição de componentes ..... (1 folha)

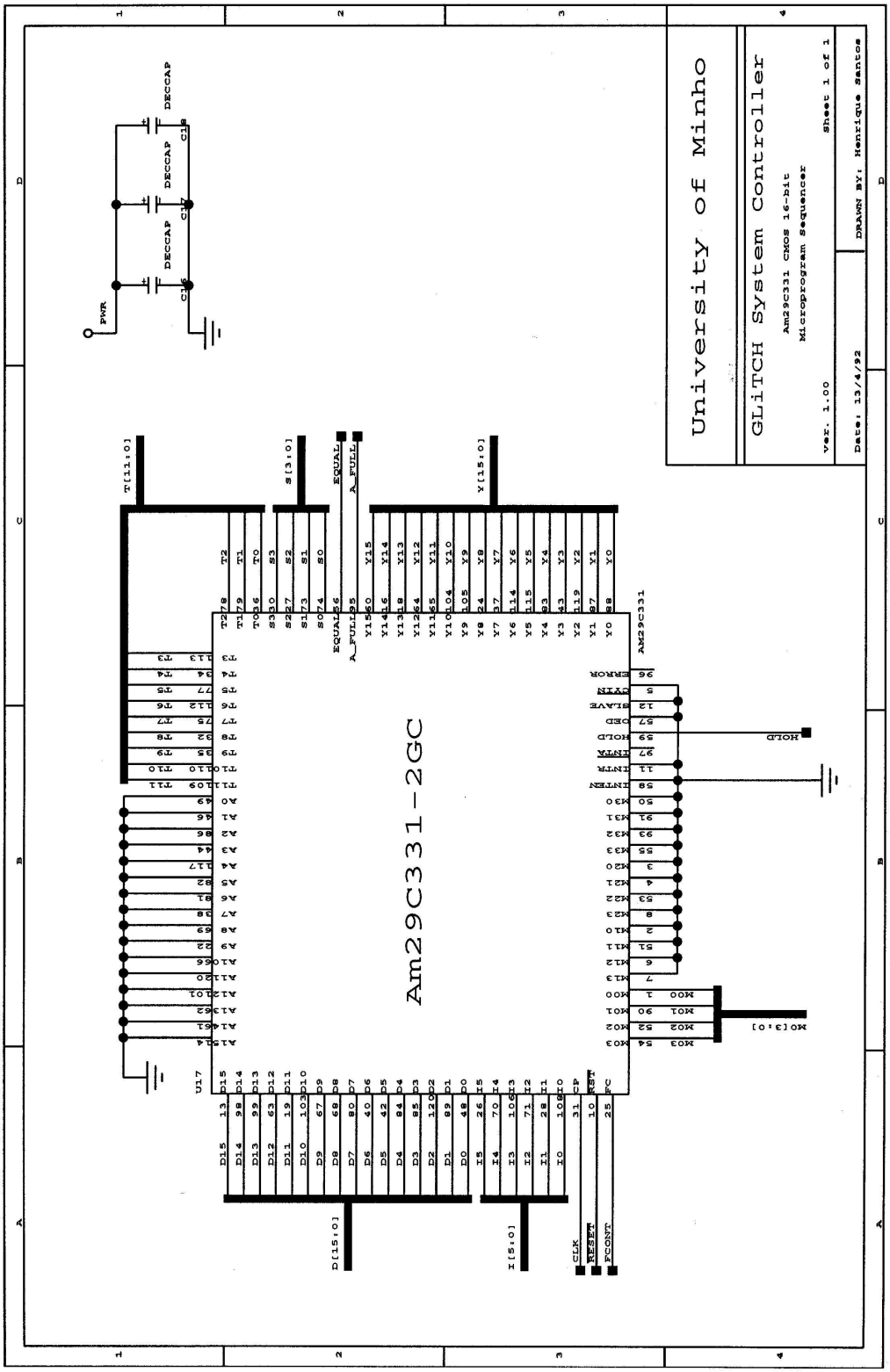

University of Minho
GLiTCH Controller
ver. 1.10
Date: 5/03/1992
Sheet 1 of 5
DRAWN BY: Henrique Santos
Tag Unit
Data Store Add. Gen. Logic
Scalar Unit
Micro Memory
From Sheet 3
Sequencer Pipe Register
Am29C331
74ALS374
MMO[95:70]
MMO[95:0]
SQA[15:0],SQI[5:0],SQS[3:0]
MMA[15:0]
BUSTRNOTLOC1
DECCAP
PWR
S_CLK
RESET

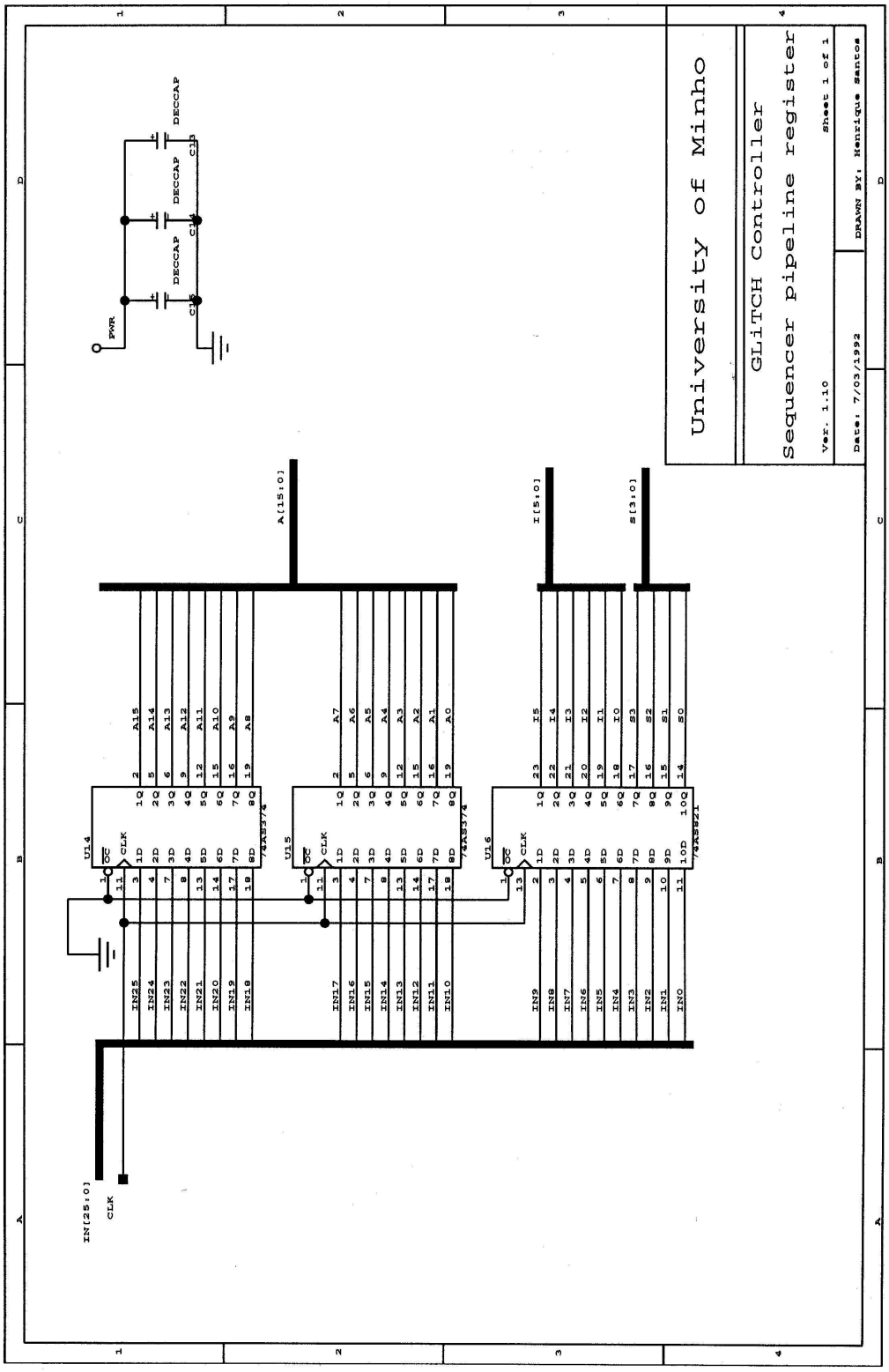

Reset Logic
To DRAM
External Transputer Buses:
Control Bus
Data Bus
Address Bus
University of Minho
GLiTCH Controller
ver. 1.10
Sheet 2 of 5
Date: 23/6/1992
DRAWN BY: Henrique Santos
GLiTCH Brd Pipe Register
IMS T800
Address latch
GPOS is also used as the 12 MSB of GLiTCH RAM address
VI1 is Ver0 signal
VI0 is S_Done flag to GLiTCH Board
GRI[5:0] includes ROUT_RAM_SEL[1:0]

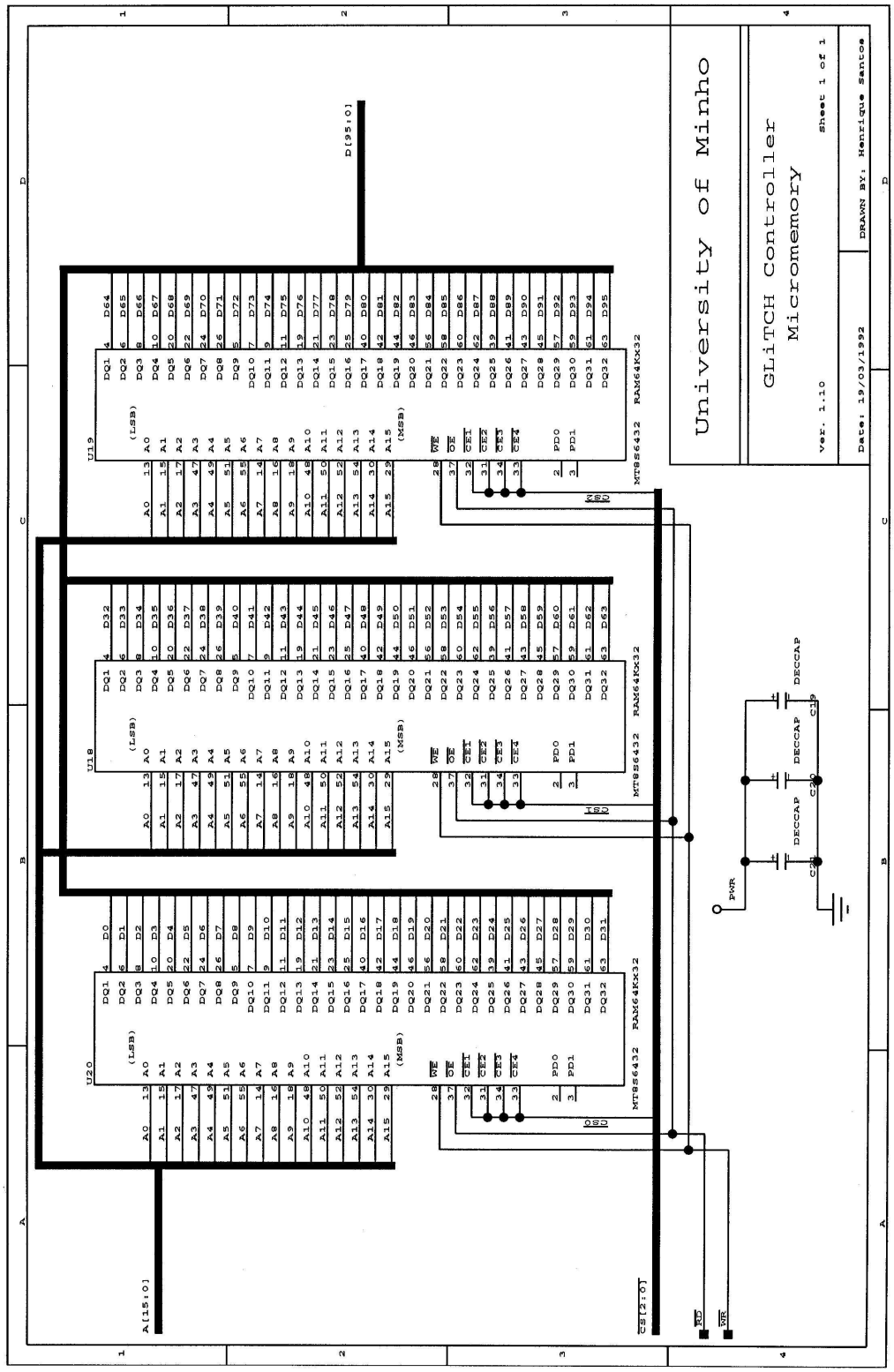

University of Minho
GLiTCH Controller
ver. 1.10
Sheet 3 of 5
Date: 6/04/1992
DRAWN BY: Henrique Santos
DRAM 4 banks
of 1Mx8 / 256Kx8
DRAM MUX
Micro Mem Data Bus interface
DStore Mem Data Bus interface
Micro Mem ADD intef
DStore Mem ADD interf
74AS04
PALCE22V10
from/to sheet 1

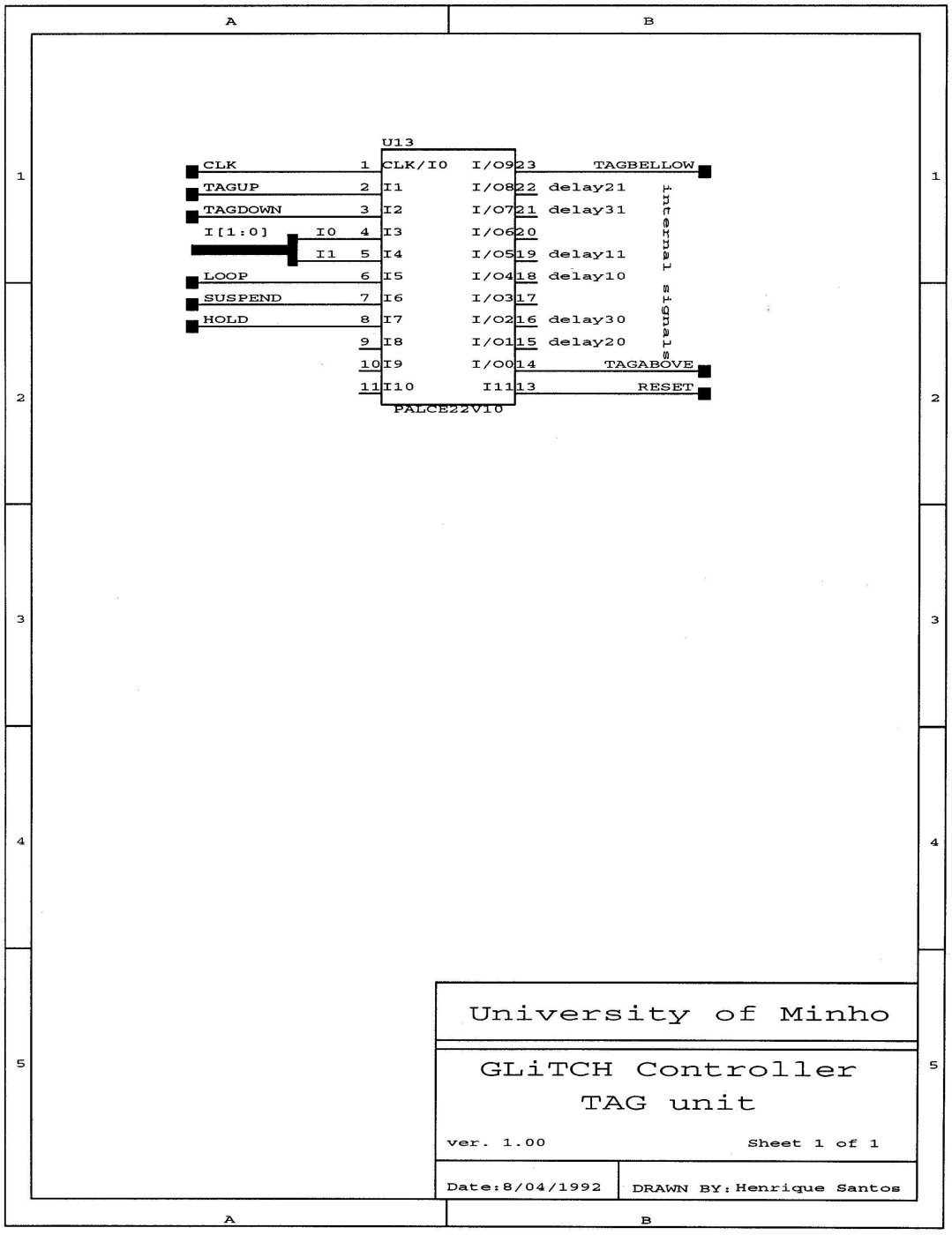

University of Minho
GLiTCH Controller
ver. 1.10
Sheet 4 of 5
Date: 23/6/1992
DRAWN BY: Henrique Santos
Notes:
MM00 is Ss flag
MM01 is Finish flag
MM02 is SC_Req flag
MM064 is SDone flag
TRAD[7:0]
Control Reg
Status Reg
Is the Suspend_0
to GlITCH Board
U65
U66
U67
U63
U64
74AS374
PALCE16V8
DECCAP
PWR
C60
C59
C58
C56
C57
CLK
NOTMEMWRB0
RDNOTWR
NOTLREGS
MM064
A_FULL
EQUAL
MM01
MM02
MM00
RESET
BUSTOTR
SDONEOUT
A_FULLOUT
STOE
STWR
CTRWR
EQUALOUT
FINISHOUT
SC_REQOUT
SSOUT
BUSTRNOTLOC1
CLRFLAGS
BUSTRNOTLOC2
SUSPENDGL
BUSREQ
RUN
CLRA_FULL
SW_INI
CLRSDONE

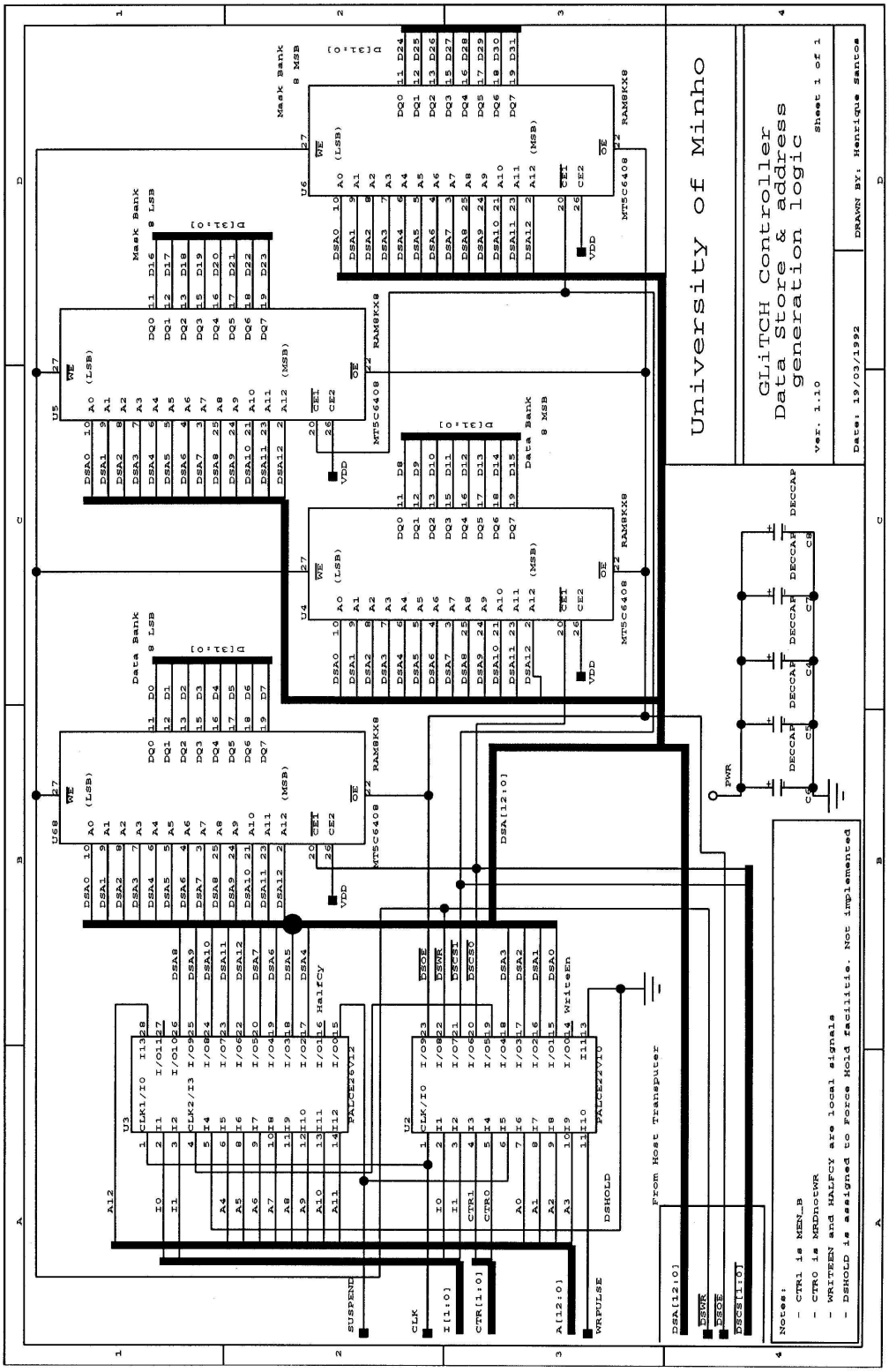

University of Minho
GLiTCH Controller
External connectors
ver. 1.20
Sheet 5 of 5
Date: 23/06/1992
DRAWN BY: Henrique Santos
LINKIN[3:0]
LINKOUT[3:0]
RESET_UP
ANALYSE
ERROR
INIT
LDC
WR-HF-OPTIONAL
WRPULSE
TAGBELLOW
TAGUP
TAGDOWN
TAGABOVE
SUSPENDGL
BUSTOTR
CY
TAG
FSTOUT
VDONE
GSUB[7:0]
VI[1:0]
DR[8:0]
GRADD[3:0]
GPOS[11:0]
BSEL[1:0]
GRI[5:0]
GI[15:0]
TRAD[9:0]
DRDB[31:0]
TRRESET
RESET
S_CLK
CLK
NOTMEMWRB
NOTLCA-CS
TRA4
TRA3
TRA2

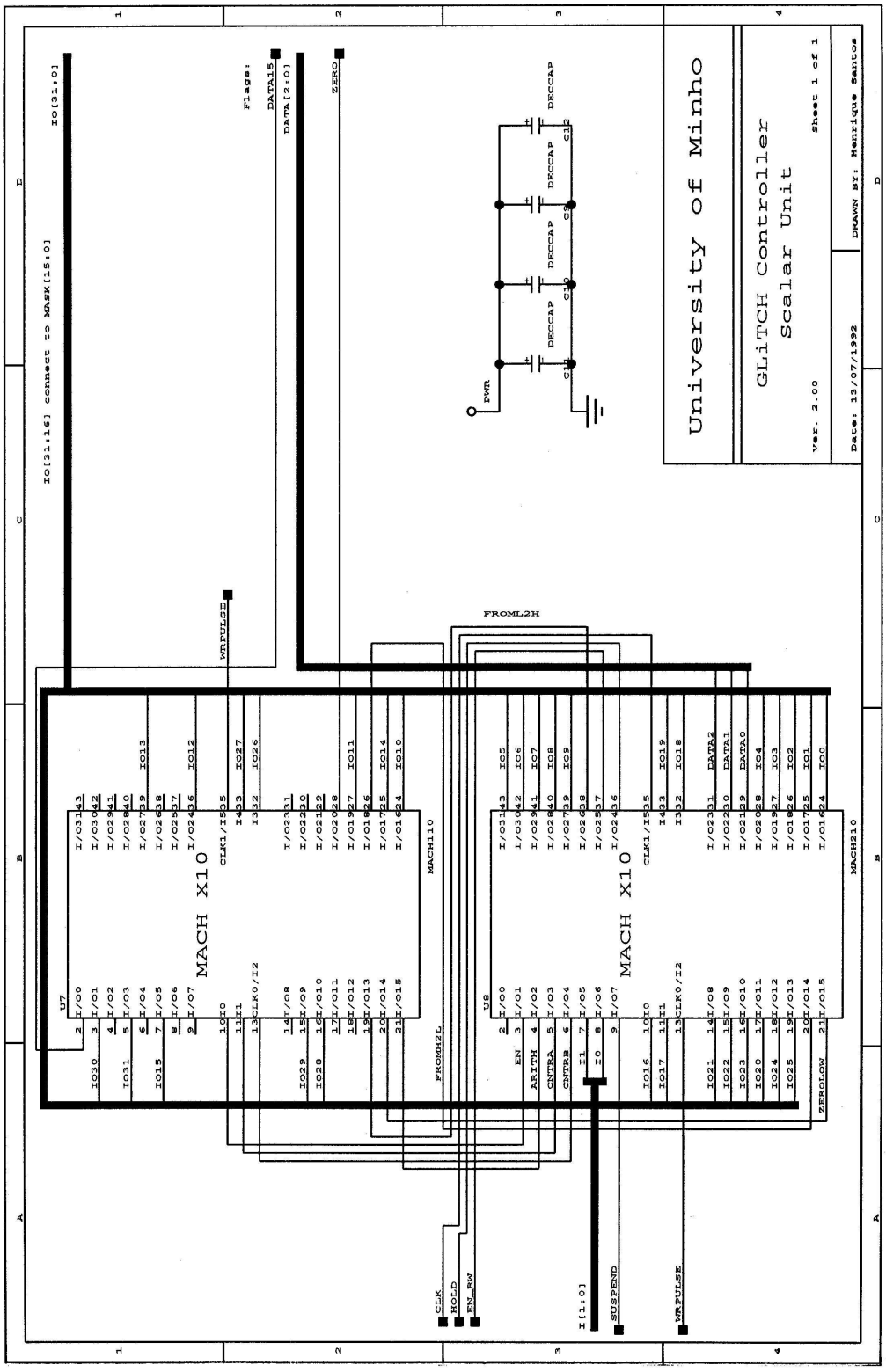

University of Minho
GLiTCH System Controller
Am29c331 CMOS 16-bit
Microprogram Sequencer
ver. 1.00
Date: 13/4/92
Sheet 1 of 1
DRAWN BY: Henrique Santos
Am29C331-2GC
Am29c331
DECCAP
PWR
T[11:0]
S[3:0]
Y[15:0]
EQUAL
A_FULL
HOLD
MO[3:0]
D[15:0]
I[5:0]
CLK
RESET
FCONT

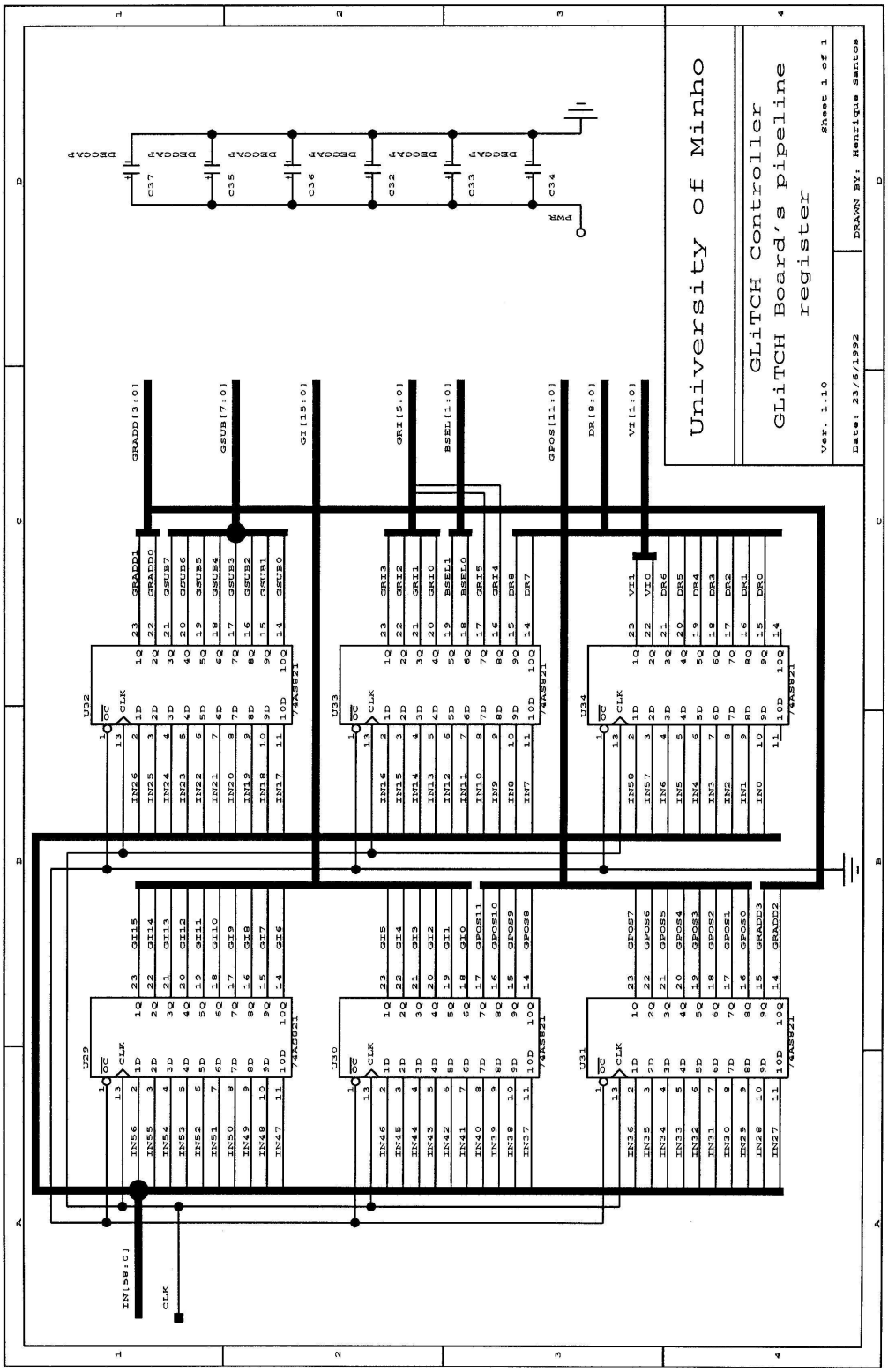

University of Minho
GLiTCH Controller
Sequencer pipeline register
Ver. 1.10
Sheet 1 of 1
Date: 7/03/1992
DRAWN BY: Henrique Santos
A[15:0]
I[5:0]
S[3:0]
IN[25:0]
CLK
PWR
DECCAP
U14
U15
U16
74AS374
74AS821

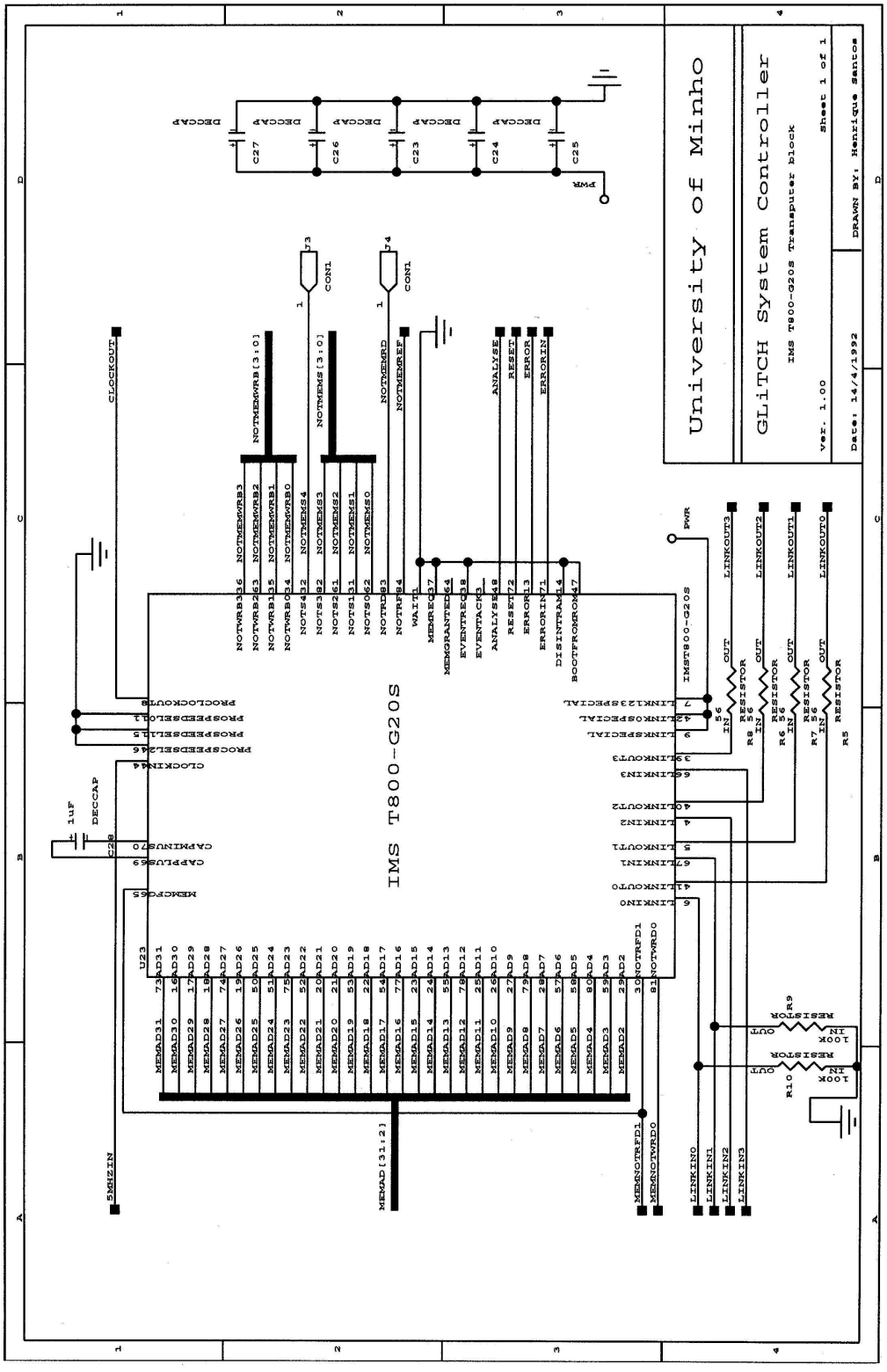

University of Minho
GLiTCH Controller
Micromemory
ver. 1.10
Sheet 1 of 1
Date: 19/03/1992
DRAWN BY: Henrique Santos
D[95:0]
A[15:0]
CS[2:0]
RD
WR
PWR
DECCAP
U18
U19
U20
MT8S6432 RAM64Kx32

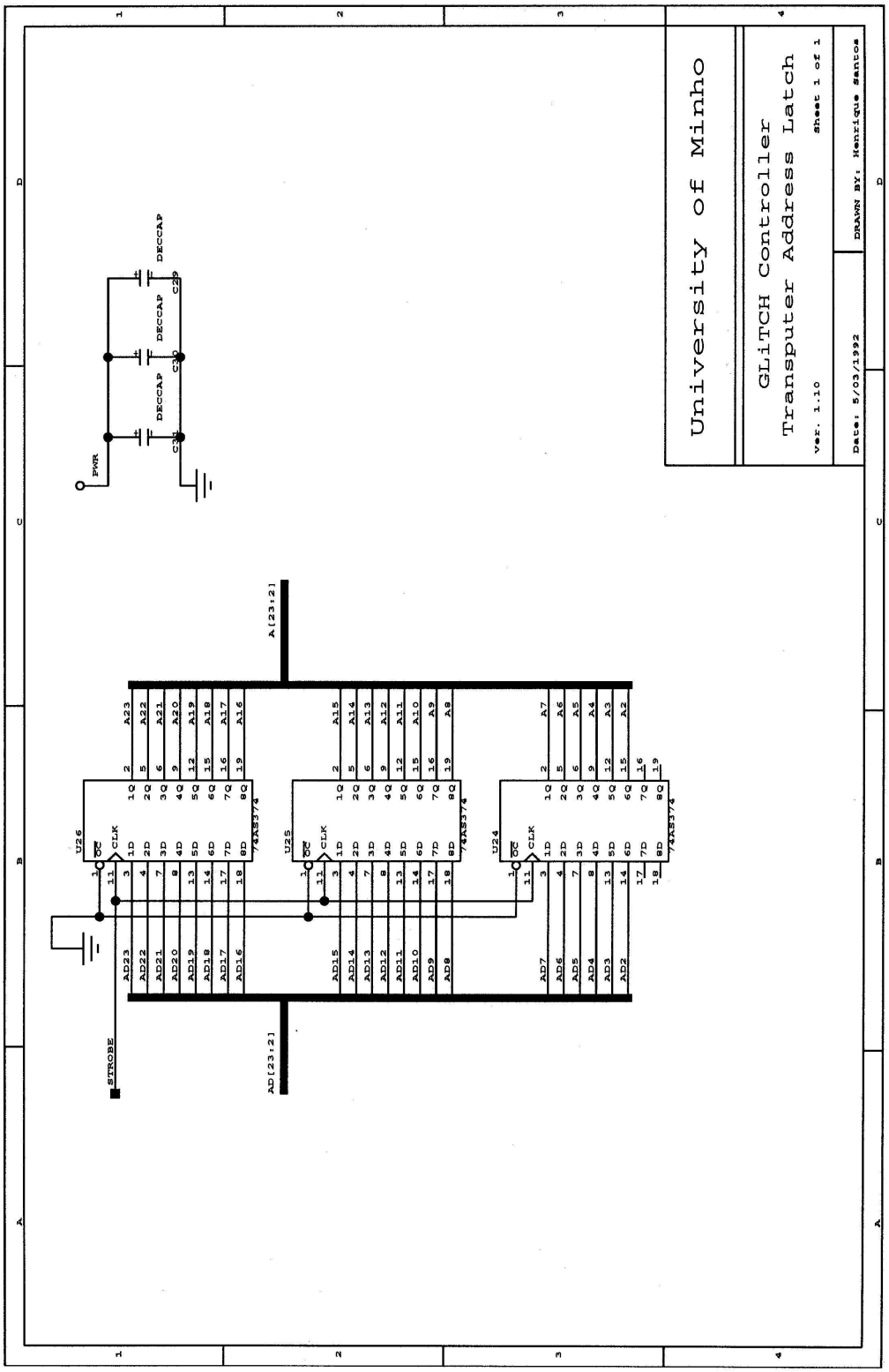

A
B
1
2
3
4
5
U13
CLK
TAGUP
TAGDOWN
I[1:0]
I0
I1
LOOP
SUSPEND
HOLD
CLK/I0
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I/O9
I/O8
I/O7
I/O6
I/O5
I/O4
I/O3
I/O2
I/O1
I/O0
TAGBELLOW
delay21
delay31
delay11
delay10
delay30
delay20
TAGABOVE
RESET
internal signals
PALCE22V10
University of Minho
GLiTCH Controller
TAG unit
ver. 1.00
Sheet 1 of 1
Date:8/04/1992
DRAWN BY:Henrique Santos

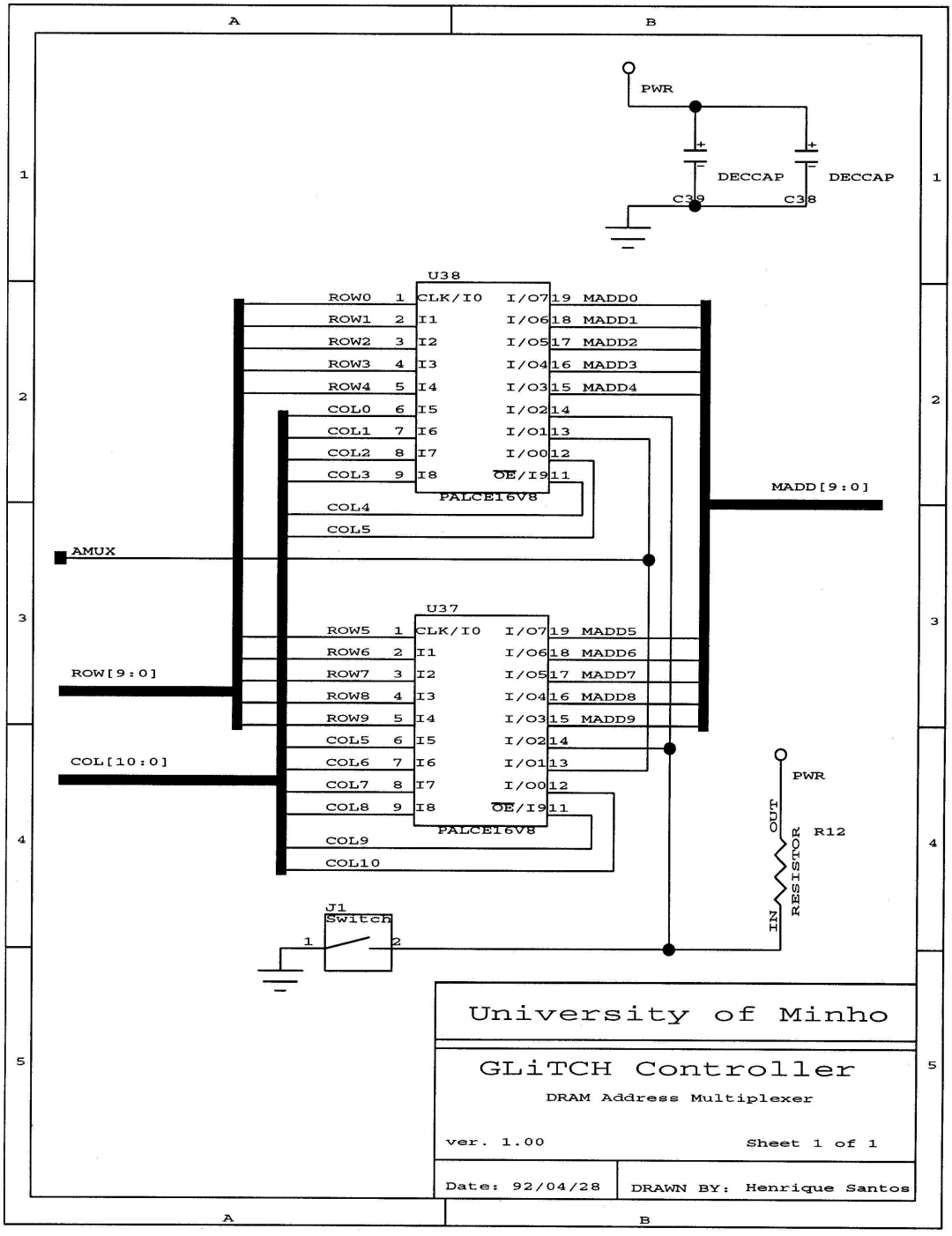

University of Minho
GLiTCH Controller
Data Store & address
generation logic
ver. 1.10
Sheet 1 of 1
Date: 19/03/1992
DRAWN BY: Henrique Santos
Mask Bank
8 MSB
Mask Bank
8 LSB
Data Bank
8 MSB
Data Bank
8 LSB
MT5C6408
RAM8KX8
PALCE22V10
DECCAP
PWR
VDD
From Host Transputer
DSA[12:0]
DSWR
DSOE
DSCS[1:0]
SUSPEND
CLK
I[1:0]
CTR[1:0]
A[12:0]
WRPULSE
DSHOLD
Notes:
- CTR1 is MEN_B
- CTR0 is MRDngtWR
- WRITTEN and HALFCY are local signals
- DSHOLD is assigned to Force Hold facilitie. Not implemented

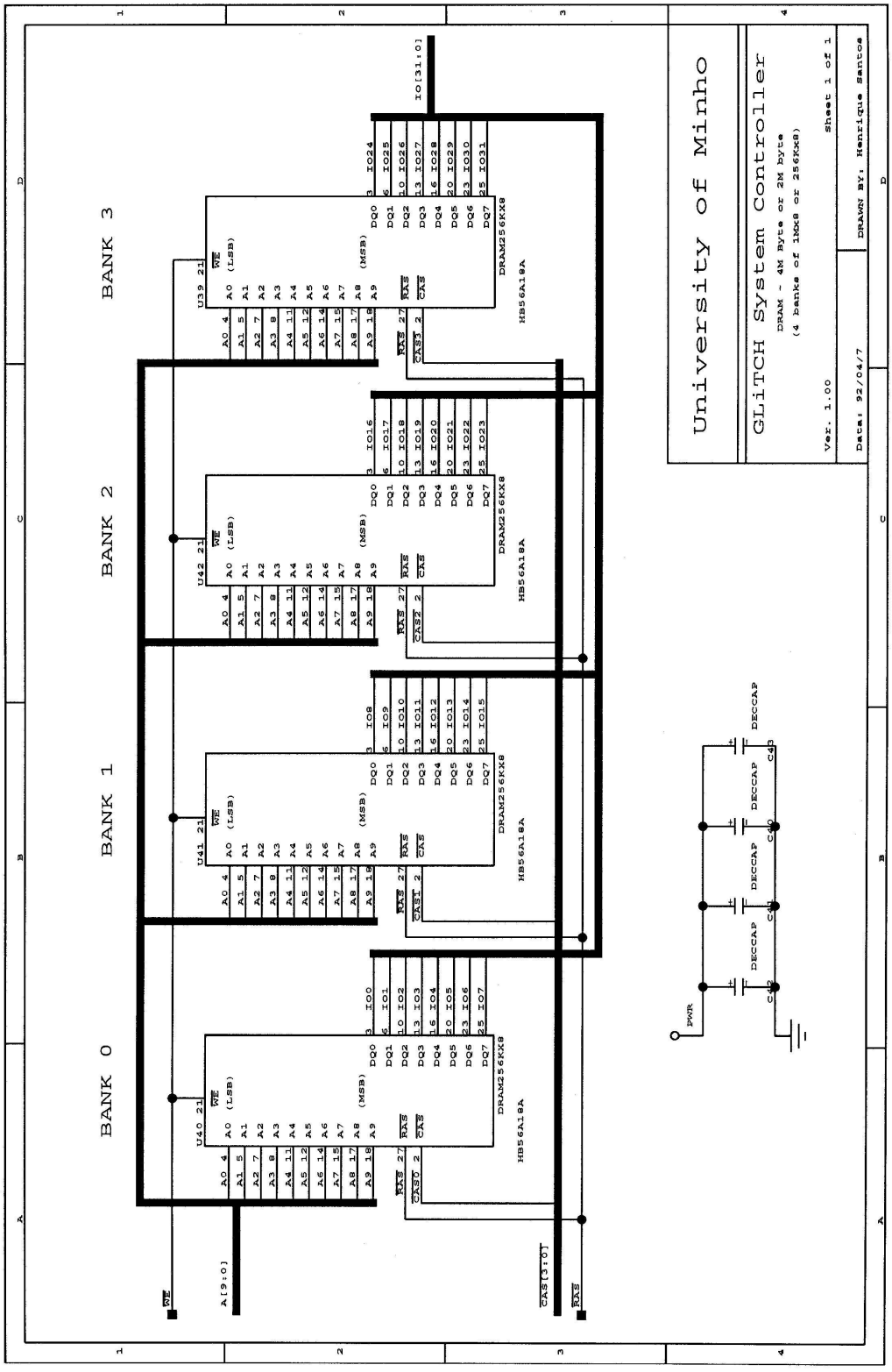

University of Minho
GLiTCH Controller
Scalar Unit
ver. 2.00
Sheet 1 of 1
Date: 13/07/1992
DRAWN BY: Henrique Santos
IO[31:0]
IO[31:16] connect to MASK[15:0]
Flags:
DATA15
DATA[2:0]
ZERO
WRPULSE
FROML2H
FROM12L
MACH110
MACH210
MACH X10
U7
U8
CLK1/I535
CLK0/I2
I403
I332
I[1:0]
SUSPEND
CLK
HOLD
EN_RW
PWR
DECCAP
C1
C2
C3
C4

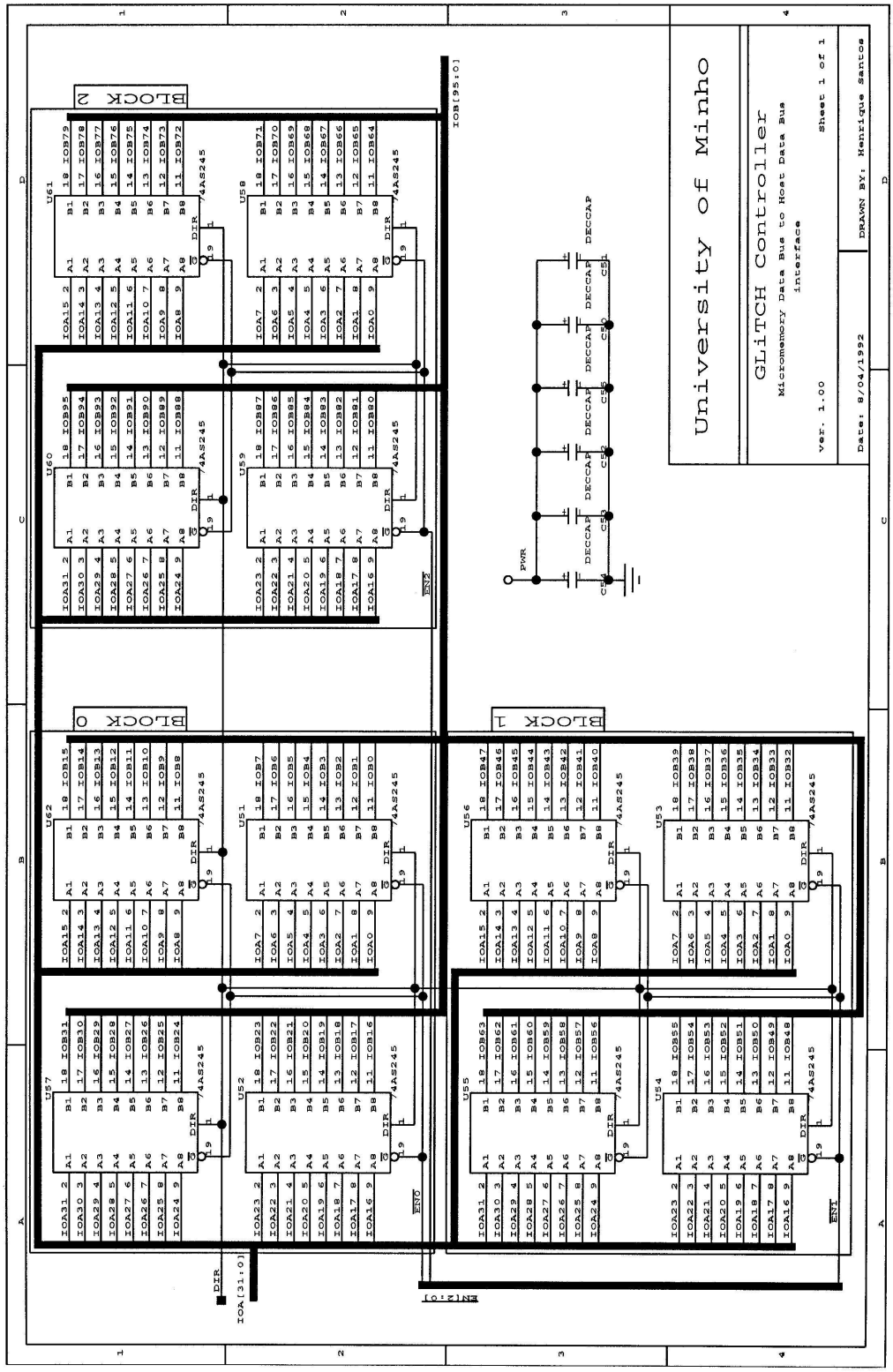

University of Minho
GLiTCH Controller
GLiTCH Board's pipeline register
Ver. 1.10
Date: 23/6/1992
DRAWN BY: Henrique Santos
Sheet 1 of 1
DECCAP
C37
C35
C36
C32
C33
C34
PWR
GRADD[3:0]
GSUB[7:0]
GI[15:0]
GRI[5:0]
BSEL[1:0]
GPOS[11:0]
DR[8:0]
VI[1:0]
U32
U33
U34
U29
U30
U31
74AS821
IN[59:0]
CLK

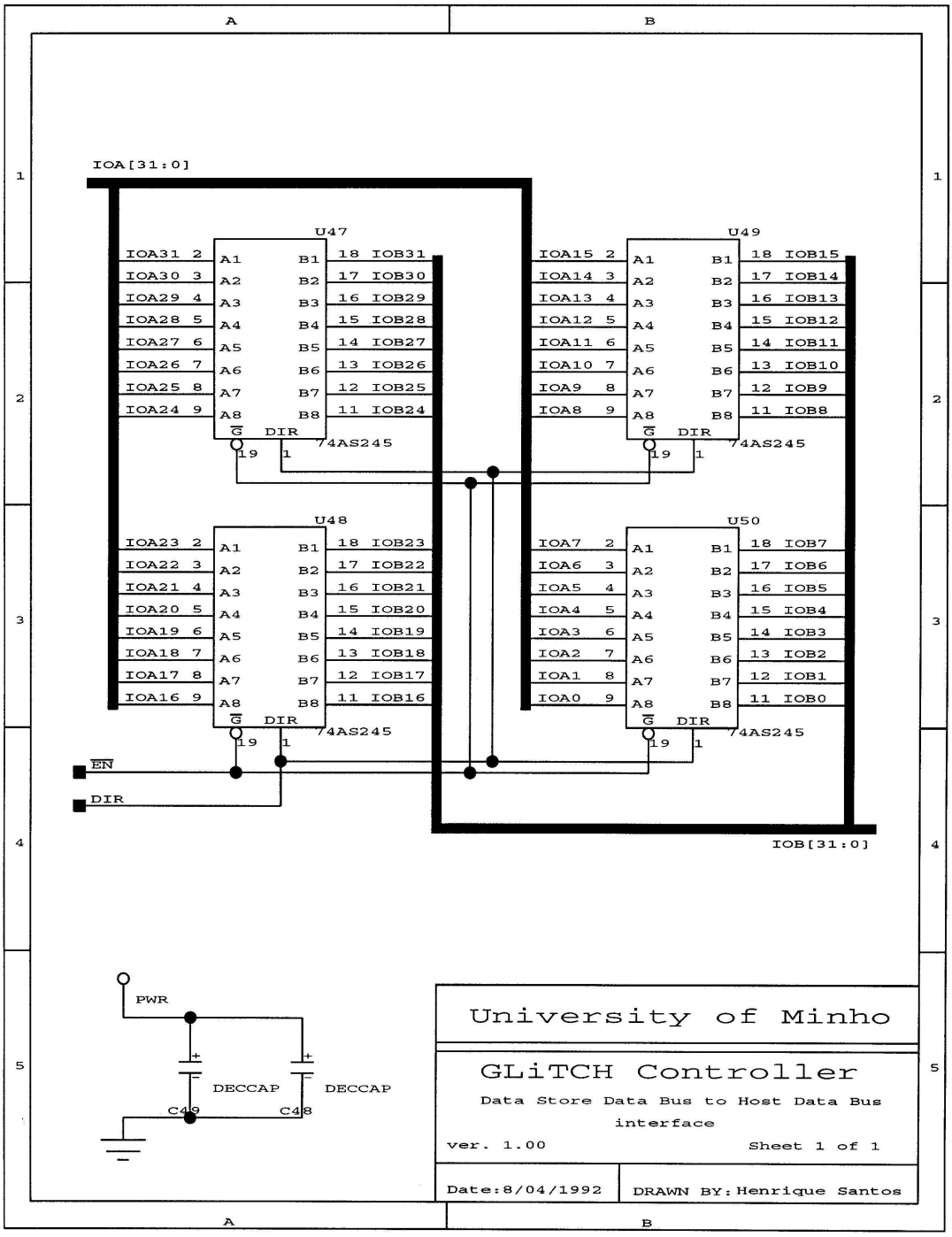

University of Minho
GLiTCH System Controller
IMS T800-G20S Transputer block
ver. 1.00
Date: 14/4/1992
DRAWN BY: Henrique Santos
Sheet 1 of 1
IMS T800-G20S
IMST800-G20S
DECCAP
C27
C26
C23
C24
C25
PWR
J3
J4
CON1
CLOCKOUT
NOTMEMWRB[3:0]
NOTMEMS[3:0]
NOTMEMRD
NOTMEMREF
ANALYSE
RESET
ERROR
ERRORIN
WAIT
MEMREQ
MEMGRANTED
EVENTREQ
EVENTACK
DISINTRAM
BOOTFROMROM
LINK123SPECIAL
LINK0SPECIAL
LINKSPECIAL
LINKOUT3
LINKOUT2
LINKOUT1
LINKOUT0
LINKIN3
LINKIN2
LINKIN1
LINKIN0
RESISTOR
100K
R5
R6
R7
R8
R9
R10
1uF
C28
CAPMINUS
CAPPLUS
MEMCFG
MEMAD[31:2]
MEMNOTRFD
MEMNOTWRD
NOTRFD
NOTWRD
5MHZIN
U23

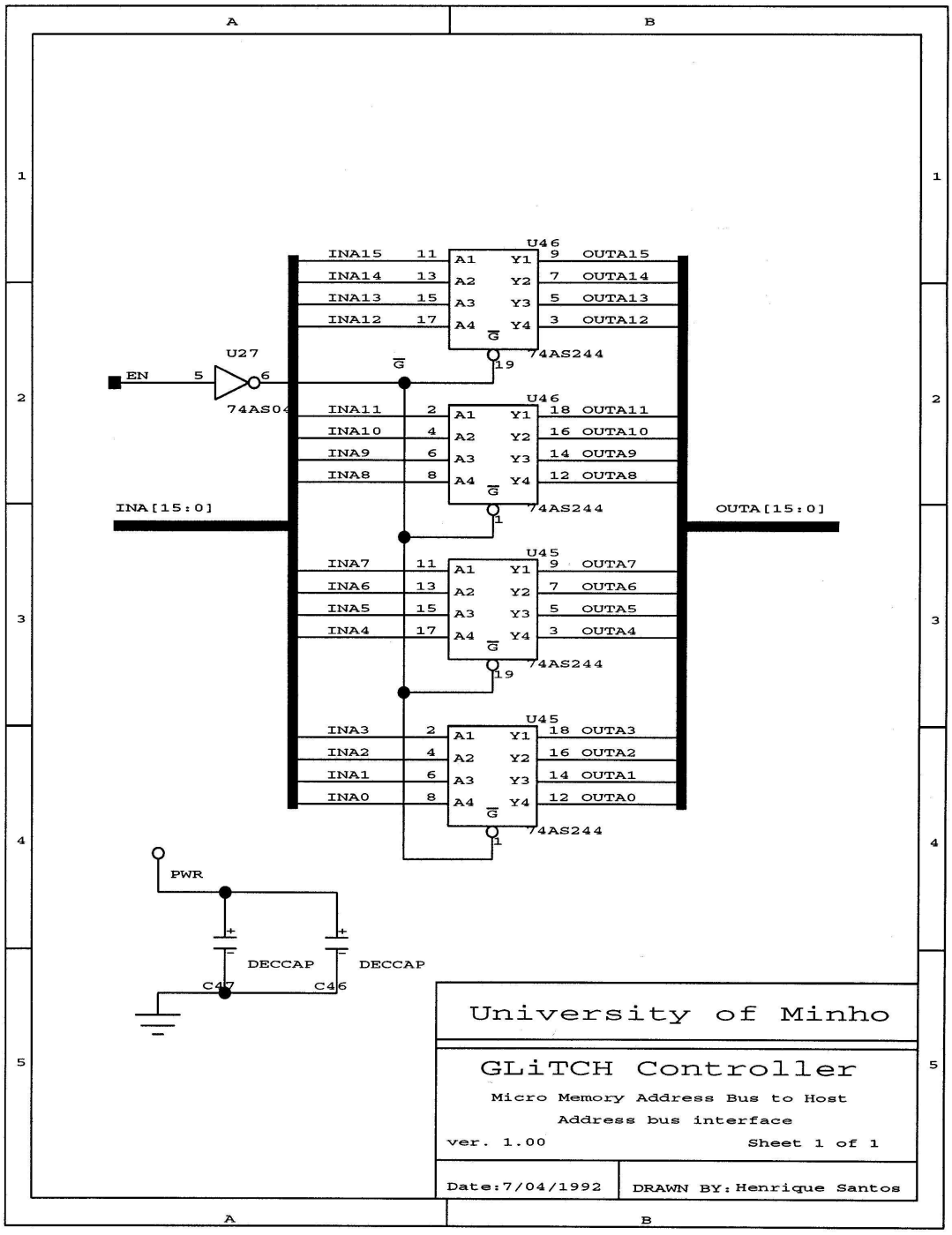

University of Minho
GLiTCH Controller
Transputer Address Latch
ver. 1.10
Sheet 1 of 1
Date: 5/03/1992
DRAWN BY: Henrique Santos
PWR
DECCAP
C30
C31
C32
C33
A[23:2]
AD[23:2]
STROBE
U24
U25
U26
74AS374
AD23
AD22
AD21
AD20
AD19
AD18
AD17
AD16
AD15
AD14
AD13
AD12
AD11
AD10
AD9
AD8
AD7
AD6
AD5
AD4
AD3
AD2

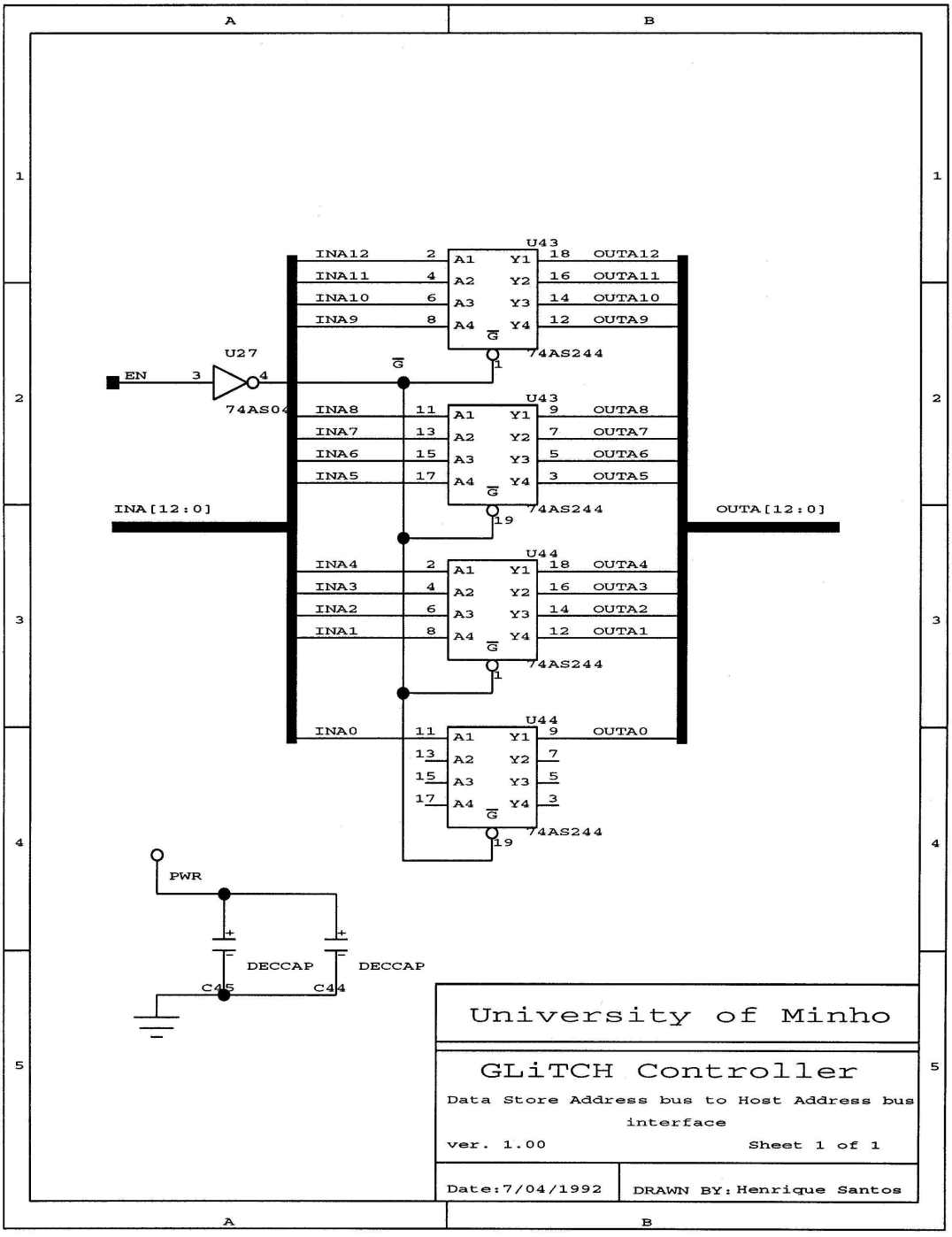

A
B
1
2
3
4
5
PWR
DECCAP
DECCAP
C39
C38
U38
ROW0 1 CLK/I0 I/O7 19 MADD0
ROW1 2 I1 I/O6 18 MADD1
ROW2 3 I2 I/O5 17 MADD2
ROW3 4 I3 I/O4 16 MADD3
ROW4 5 I4 I/O3 15 MADD4
COL0 6 I5 I/O2 14
COL1 7 I6 I/O1 13
COL2 8 I7 I/O0 12
COL3 9 I8 OE/I9 11
PALCE16V8
COL4
COL5
MADD[9:0]
AMUX
U37
ROW5 1 CLK/I0 I/O7 19 MADD5
ROW6 2 I1 I/O6 18 MADD6
ROW7 3 I2 I/O5 17 MADD7
ROW8 4 I3 I/O4 16 MADD8
ROW9 5 I4 I/O3 15 MADD9
COL5 6 I5 I/O2 14
COL6 7 I6 I/O1 13
COL7 8 I7 I/O0 12
COL8 9 I8 OE/I9 11
PALCE16V8
COL9
COL10
ROW[9:0]
COL[10:0]
PWR
OUT
R12
RESISTOR
IN
J1
Switch
1
2
University of Minho
GLiTCH Controller
DRAM Address Multiplexer
ver. 1.00
Sheet 1 of 1
Date: 92/04/28
DRAWN BY: Henrique Santos

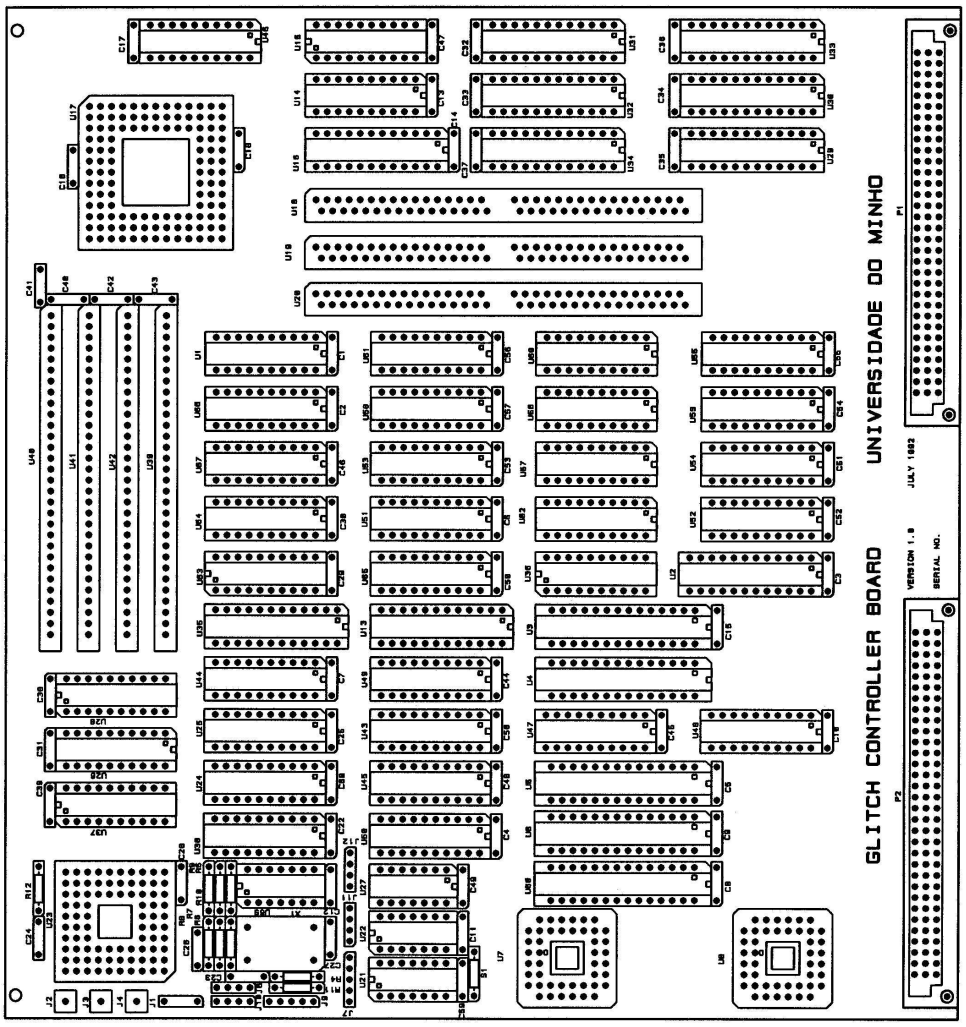

University of Minho
GLiTCH System Controller
DRAM - 4M Byte or 2M byte
(4 banks of 1Mx8 or 256Kx8)
Ver. 1.00
Sheet 1 of 1
Date: 92/04/7
DRAWN BY: Henrique Santos
BANK 0
BANK 1
BANK 2
BANK 3
IO[31:0]
A[9:0]
WE
RAS
CAS[3:0]
PWR
DECCAP
DRAM256KX8
HB56A18A


University of Minho
GLiTCH Controller
Microcomancy Data Bus to Host Data Bus interface
ver. 1.00
Date: 8/04/1992
DRAWN BY: Henrique Santos
Sheet 1 of 1
BLOCK 0
BLOCK 1
BLOCK 2
IOA[31:0]
IOB[95:0]
ENT[2:0]
ENT0
ENT1
ENT2
DIR
PWR
DECCAP
74AS245


IOA[31:0]
U47
IOA31 2 A1 B1 18 IOB31
IOA30 3 A2 B2 17 IOB30
IOA29 4 A3 B3 16 IOB29
IOA28 5 A4 B4 15 IOB28
IOA27 6 A5 B5 14 IOB27
IOA26 7 A6 B6 13 IOB26
IOA25 8 A7 B7 12 IOB25
IOA24 9 A8 B8 11 IOB24
G DIR 74AS245
19 1
U49
IOA15 2 A1 B1 18 IOB15
IOA14 3 A2 B2 17 IOB14
IOA13 4 A3 B3 16 IOB13
IOA12 5 A4 B4 15 IOB12
IOA11 6 A5 B5 14 IOB11
IOA10 7 A6 B6 13 IOB10
IOA9 8 A7 B7 12 IOB9
IOA8 9 A8 B8 11 IOB8
G DIR 74AS245
19 1
U48
IOA23 2 A1 B1 18 IOB23
IOA22 3 A2 B2 17 IOB22
IOA21 4 A3 B3 16 IOB21
IOA20 5 A4 B4 15 IOB20
IOA19 6 A5 B5 14 IOB19
IOA18 7 A6 B6 13 IOB18
IOA17 8 A7 B7 12 IOB17
IOA16 9 A8 B8 11 IOB16
G DIR 74AS245
19 1
U50
IOA7 2 A1 B1 18 IOB7
IOA6 3 A2 B2 17 IOB6
IOA5 4 A3 B3 16 IOB5
IOA4 5 A4 B4 15 IOB4
IOA3 6 A5 B5 14 IOB3
IOA2 7 A6 B6 13 IOB2
IOA1 8 A7 B7 12 IOB1
IOA0 9 A8 B8 11 IOB0
G DIR 74AS245
19 1
EN
DIR
IOB[31:0]
PWR
DECCAP C49
DECCAP C48
University of Minho
GLiTCH Controller
Data Store Data Bus to Host Data Bus interface
ver. 1.00
Sheet 1 of 1
Date:8/04/1992
DRAWN BY: Henrique Santos


A
B
1
2
3
4
5
U46
INA15 11 A1 Y1 9 OUTA15
INA14 13 A2 Y2 7 OUTA14
INA13 15 A3 Y3 5 OUTA13
INA12 17 A4 Y4 3 OUTA12
74AS244
19
U27
EN 5 6
74AS04
U46
INA11 2 A1 Y1 18 OUTA11
INA10 4 A2 Y2 16 OUTA10
INA9 6 A3 Y3 14 OUTA9
INA8 8 A4 Y4 12 OUTA8
74AS244
1
INA[15:0]
OUTA[15:0]
U45
INA7 11 A1 Y1 9 OUTA7
INA6 13 A2 Y2 7 OUTA6
INA5 15 A3 Y3 5 OUTA5
INA4 17 A4 Y4 3 OUTA4
74AS244
19
U45
INA3 2 A1 Y1 18 OUTA3
INA2 4 A2 Y2 16 OUTA2
INA1 6 A3 Y3 14 OUTA1
INA0 8 A4 Y4 12 OUTA0
74AS244
1
PWR
DECCAP
DECCAP
C47
C46
University of Minho
GLiTCH Controller
Micro Memory Address Bus to Host
Address bus interface
ver. 1.00
Sheet 1 of 1
Date:7/04/1992
DRAWN BY:Henrique Santos


A
B
1
2
3
4
5
U43
INA12
INA11
INA10
INA9
OUTA12
OUTA11
OUTA10
OUTA9
74AS244
U27
EN
74AS04
U43
INA8
INA7
INA6
INA5
OUTA8
OUTA7
OUTA6
OUTA5
74AS244
INA[12:0]
OUTA[12:0]
U44
INA4
INA3
INA2
INA1
OUTA4
OUTA3
OUTA2
OUTA1
74AS244
U44
INA0
OUTA0
74AS244
PWR
DECCAP
DECCAP
C45
C44
University of Minho
GLiTCH Controller
Data Store Address bus to Host Address bus interface
ver. 1.00
Sheet 1 of 1
Date:7/04/1992
DRAWN BY:Henrique Santos


GLITCH CONTROLLER BOARD
UNIVERSIDADE DO MINHO
VERSION 1.0
JULY 1992
SERIAL NO.

Apêndice C - Descrições em PALASM

Índice

1. *Address Generation Logic* 212
1.1 Lógica de controlo e 4 LSB do registo de endereço 212
1.2 Vectores de teste 214
1.3 9 MSB do registo de endereço 216
1.4 Vectores de teste 217

2. Unidade escalar 219
2.1 Lógica de controlo e 10 LSB do registo de deslocamento 219
2.2 Vectores de teste 222
2.3 6 MSB do registo de deslocamento 226
2.4 Vectores de teste 227

3. Selector de *tags* 227
3.1 Vectores de teste 230

4. *Bus Arbiter* 232
4.1 Máquina de estados principal 232
4.2 Vectores de teste 235
4.3 Gestão de eventos 238
4.4 Vectores de teste 242

5. Componentes de suporte ao transputer hospedeiro 245
5.1 Desmultiplexador do barramento e gerador dos sinais CAS 245
5.2 Vectores de teste 246
5.3 Multiplexador de endereços para DRAM (5 bits) 247
5.4 Vectores de teste 249
5.5 Descodificador de endereços (componente 1) 251
5.6 Vectores de teste 252
5.7 Gerador de sinais de controlo para a micromemória e o DS 254
5.8 Vectores de teste 255
5.9 Registos de estado e controlo 256
5.10 Vectores de teste 258

1. *Address Generation Logic*

1.1 Lógica de controlo e 4 LSB do registo de endereço

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    DATA STORE ADDRESS GENERATION LOGIC
PATTERN  A (WITH 4 BIT ADDRESS ONLY)
REVISION 1.04.00
AUTHOR   HENRIQUE M. D. SANTOS
COMPANY  UNIVERSITY OF MINHO
DATE     1/12/92

CHIP  DS_AGL  PAL22V10

;---------------------------------- PIN Declarations ---------------
PIN  1          Clock                COMBINATORIAL ; INPUT
PIN  2..3       MInstr[0..1]         COMBINATORIAL ; INPUT
PIN  4          MEN_B                COMBINATORIAL ; INPUT
PIN  5          MRdnotWr             COMBINATORIAL ; INPUT
PIN  6          BusTrnotLoc1         COMBINATORIAL ; INPUT
PIN  7..10      MAdd[0..3]           COMBINATORIAL ; INPUT
PIN  11         WritePulse           COMBINATORIAL ; INPUT
PIN  12         GND                               ; INPUT
PIN  13         ForceHld             COMBINATORIAL ; INPUT
PIN  14         WriteEN              REGISTERED ; OUTPUT
PIN  15..18     A[0..3]              REGISTERED ; OUTPUT
PIN  19         CyOut                COMBINATORIAL ; OUTPUT
PIN  20..21     MemnotCS[0..1]       REGISTERED ; OUTPUT
PIN  22         MemnotWr             COMBINATORIAL ; OUTPUT
PIN  23         MemnotOE             REGISTERED ; OUTPUT
PIN  24         VCC                               ; INPUT

NODE 1          GlobalAR             COMB       ;Global Reset

; Declaration Segment
        ;Microinstructions codes
STRING          Hold    '/Minstr[0] * /MInstr[1]'
STRING          HoldSim '/Minstr[0]   /MInstr[1]';used for simulation
STRING          Inc     '/Minstr[0] *  MInstr[1]'
STRING          IncSim  '/Minstr[0]    MInstr[1]';used for simulation
STRING          IncPat  '#b01'                    ;used in case structures
STRING          Dec     ' Minstr[0] * /MInstr[1]'
STRING          DecSim  ' Minstr[0]   /MInstr[1]';used for simulation
STRING          DecPat  '#b10'                    ;used in case structures
STRING          Load    ' Minstr[0] *  MInstr[1]'
STRING          LoadSim ' Minstr[0]    MInstr[1]';used for simulation
        ;Shared signals with the Host
GROUP Shared_Signals A[0..3] MemnotCS[0..1] MemnotWr MemnotOE

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
Shared_Signals.TRST = /BusTrnotLoc1

;MINIMIZE_OFF
;GlobalAR = GND        ;Gloabal Reset is not used
;MINIMIZE_ON
```

```
        ;Cy generation
CASE (MInstr[0..1])
   BEGIN
      DecPat: BEGIN
              CyOut = /A[0] * /A[1] * /A[2] * /A[3]
              END
      IncPat: BEGIN
              CyOut = A[0] * A[1] * A[2] * A[3]
              END
      OTHERWISE: BEGIN
              CyOut = GND
                 END
   END

        ;Data Store control signals
IF (/BusTrnotLoc1 * /ForceHld) THEN
   BEGIN
        ;Write signal - WriteEN (one cycle delay) is a window for WritePulse
   WriteEN := /MRdnotWr
   MemnotWr = /(WriteEN * /WritePulse)
        ;The following outputs are all registered
   MemnotOE := /MRdnotWr

        ;Chip Selects para a memoria
   /MemnotCS[0] := MEN_B            ;Data Bank, allways enable
   /MemnotCS[1] := MEN_B * MRdnotWr ;Mask Bank, enabled only for read

        ;Contador Up/Down/Load/Hold
   A[0] := Hold * A[0]
           + (Inc + Dec) * /A[0]
           + Load * MAdd[0]
   A[1] := Hold * A[1]
           + Inc * (A[1] :+: A[0])
           + Dec * (A[1] :*: A[0])
           + Load * MAdd[1]
   A[2] := Hold * A[2]
           + Inc * (A[2] :+: (A[1] * A[0]))
           + Dec * (A[2] :*: (A[1] + A[0]))
           + Load * MAdd[2]
   A[3] := Hold * A[3]
           + Inc * (A[3] :+: (A[2] * A[1] * A[0]))
           + Dec * (A[3] :*: (A[2] + A[1] + A[0]))
           + Load * MAdd[3]
   END
   ELSE    ;BusTrnotLoc1 or ForceHld set. Recycle all registers.
      BEGIN
      WriteEN := WriteEN
      MemnotOE := MemnotOE
      MemnotCS[0] := MemnotCS[0]
      MemnotCS[1] := MemnotCS[1]
      A[0] := A[0]
      A[1] := A[1]
      A[2] := A[2]
      A[3] := A[3]
      END
```

1.2 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON MEN_B MRdnotWr BusTrnotLoc1 WritePulse ForceHld A[0..3] MemnotWr
         MemnotOE MemnotCS[0..1] CyOut
    ;Normal operation - control from Micromemory (Hold EN_B RdnotWr)
    ;No access. Note that MemnotCS[0..1] only depends on MEN_B (..[1] also
    ;depends on MRdnotWr), while the other control signals don't depend
    ;on MEN_B
        ;Chip Select only test
;But first we need to put the chip in a known state, since Power-up Reset
;doesnït work with LabPro! Anyway, due to this, the very first Vector will
;produce an error.

SETF /Clock WritePulse /BusTrnotLoc1 /ForceHld LoadSim /MEN_B MRdnotWr
     /MAdd[0..3]
CLOCKF Clock
CHECK MemnotWr /MemnotOE MemnotCS[0..1] /WriteEN /CyOut /A[0..3]

SETF HoldSim
CHECK MemnotWr /MemnotOE MemnotCS[0..1] ; Memory control signals
SETF /WritePulse
CHECK MemnotWr /MemnotOE MemnotCS[0..1] ; Memory control signals
SETF WritePulse /MEN_B /MRdnotWr
CLOCKF Clock
CHECK MemnotWr MemnotOE MemnotCS[0..1] ; Memory control signals
SETF /WritePulse
CHECK /MemnotWr MemnotOE MemnotCS[0..1] ; Memory control signals
        ;Read
SETF WritePulse MEN_B MRdnotWr
CLOCKF Clock
CHECK MemnotWr /MemnotOE /MemnotCS[0..1] ; Memory control signals
SETF /WritePulse
CHECK MemnotWr /MemnotOE /MemnotCS[0..1] ; Memory control signals
        ;Write
SETF WritePulse MEN_B /MRdnotWr
CLOCKF Clock
CHECK MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1] ;only bank 0 must be en.
SETF /WritePulse
CHECK /MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1]
SETF WritePulse /MEN_B MRdnotWr    ;stay in not enable state.
CLOCKF Clock
CHECK  MemnotWr /MemnotOE  MemnotCS[0]  MemnotCS[1]

        ;Register operation
                ;LOAD 0101
SETF LoadSim /MAdd[3] MAdd[2] /MAdd[1] MAdd[0]
CLOCKF Clock
CHECK /A[3] A[2] /A[1] A[0] /Cyout
                ;INC operation
SETF IncSim
FOR i:=1 TO 6 DO
   BEGIN
      CLOCKF Clock
   END
CHECK A[3] /A[2] A[1] A[0] /CyOut ;0101 + '6' = 1011
FOR i:=1 TO 4 DO
   BEGIN
```

```
      CLOCKF Clock
   END
CHECK A[3] A[2] A[1] A[0] CyOut ;1011 + '4' = 1111. CyOut active
CLOCKF
CHECK /A[3] /A[2] /A[1] /A[0] /CyOut ;1111 + '1' = 0000. /CyOut
                ;LOAD 0110
SETF LoadSim /MAdd[3] MAdd[2] MAdd[1] /MAdd[0]
CLOCKF Clock
CHECK /A[3] A[2] A[1] /A[0] /Cyout
                ;DEC operation
SETF DecSim
FOR i:=1 TO 4 DO
   BEGIN
      CLOCKF Clock
   END
CHECK /A[3] /A[2] A[1] /A[0] /CyOut ;0110 - '4' = 0010
CLOCKF Clock
CLOCKF Clock
CHECK /A[3] /A[2] /A[1] /A[0] CyOut ;0010 - '2' = 0000. CyOut active
CLOCKF
CHECK A[3] A[2] A[1] A[0] /CyOut ;0000 - '1' = 1111. CyOut inactive
                ;Hold Instruction
SETF HoldSim
CLOCKF Clock
CHECK A[0..3]

                ;Access from the Host Transputer
CLOCKF Clock
SETF BusTrnotLoc1
CHECK ^A[0..3] ^MemnotCS[0..1] ^MemnotWr ^MemnotOE
CLOCKF Clock
SETF /BusTrnotLoc1

        ;Test ForceHld. A0..A3 should hold 1111 from last operation.
SETF MEN_B MRdnotWr WritePulse ;supose it is in read mode
CLOCKF Clock
CHECK MemnotWr /MemnotOE /MemnotCS[0..1] ; Memory control signals
SETF ForceHld
CLOCKF Clock
CHECK A[0..3] MemnotWr /MemnotOE /MemnotCS[0] /MemnotCS[1]
SETF /WritePulse
CHECK A[0..3] MemnotWr /MemnotOE /MemnotCS[0] /MemnotCS[1]
        ;change all inputs
SETF /MAdd[0..3] LoadSim /MRdnotWr /MEN_B WritePulse
CLOCKF Clock
CHECK A[0..3] MemnotWr /MemnotOE /MemnotCS[0] /MemnotCS[1]
SETF HoldSim /ForceHld MEN_B /MRdnotWr WritePulse ;now for write mode
CLOCKF Clock
CHECK MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1] ;only bank 0 must be en.
SETF ForceHld
CLOCKF Clock
CHECK A[0..3] MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1]
SETF /WritePulse
CHECK A[0..3] /MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1]
        ;change all inputs
SETF /MAdd[0..3] LoadSim /MRdnotWr /MEN_B /WritePulse
CLOCKF Clock
CHECK A[0..3] /MemnotWr MemnotOE /MemnotCS[0] MemnotCS[1]
TRACE_OFF
```

```
;---------------------------------------------------------------------
```

1.3 9 MSB do registo de endereço

```
;PALASM Design Description

;--------------------------------- Declaration Segment ------------
TITLE    9 BIT COUNTER WITH UP/DOWN/HOLD/PARALLEL LOAD OPERATIONS
PATTERN  CONTNBIT
REVISION 1.03
AUTHOR   HENRIQUE
15           BusTrnotLoc1         COMBINATORIAL ; INPUT
PIN  4            CyIn                 COMBINATORIAL ; INPUT
PIN  5            ForceHld             COMBINATORIAL ; INPUT
PIN  7            VCC                               ; INPUT
PIN  6,8..14      MAdd[0..7]           COMBINATORIAL ; INPUT
PIN  28           MAdd[8]              COMBINATORIAL ; INPUT
PIN  16           HalfCy               COMBINATORIAL ; OUTPUT
PIN  17..20       Al[0..3]             REGISTERED ; OUTPUT
PIN  21           GND                           ; INPUT
PIN  22..26       Ah[4..0]             REGISTERED ; OUTPUT
; Declaration Segment
STRING            Hold    '/Minstr[0] * /MInstr[1]'
STRING            HoldSim '/Minstr[0]   /Minstr[1]' ;used for simulation
STRING            Inc     '/Minstr[0] *  MInstr[1]'
STRING            IncSim  '/Minstr[0]    Minstr[1]' ;used for simulation
STRING            IncPat  '#b01'                    ;used for CASE structure
STRING            Dec     ' Minstr[0] * /MInstr[1]'
STRING            DecSim  ' Minstr[0]   /Minstr[1]' ;used for simulation
STRING            DecPat  '#b10'                    ;used for CASE structure
STRING            Load    ' Minstr[0] *  MInstr[1]'
STRING            LoadSim ' Minstr[0]    Minstr[1]' ;used for simulation
        ;Shared signals with the Host
GROUP Reg_Outputs Al[0..3] Ah[0..4]

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
Reg_Outputs.TRST = /BusTrnotLoc1
Reg_Outputs.CLKF = Clock         ;-- only for 26V12
        ;Carry input for the 5 most significant bits
CASE (MInstr[0..1])
   BEGIN
      IncPat: BEGIN
              HalfCy = Al[0] * Al[1] * Al[2] * Al[3] * CyIn
              END
      DecPat: BEGIN
              HalfCy = /Al[0] * /Al[1] * /Al[2] * /Al[3] * CyIn
              END
      OTHERWISE: BEGIN
              HalfCy = GND
                 END
   END

        ;Up/Down and Load counter
IF (/BusTrnotLoc1 * /ForceHld) THEN
   BEGIN
   Al[0] := Hold * Al[0]
```

```
            + Inc * (Al[0] :+: CyIn)
            + Dec * (Al[0] :*: /CyIn)
            + Load * MAdd[0]
   Al[1] := Hold * Al[1]
            + Inc * (Al[1] :+: (Al[0] * CyIn))
            + Dec * (Al[1] :*: (Al[0] + /CyIn))
            + Load * MAdd[1]
   Al[2] := Hold * Al[2]
            + Inc * (Al[2] :+: (Al[1] * Al[0] * CyIn))
            + Dec * (Al[2] :*: (Al[1] + Al[0] + /CyIn))
            + Load * MAdd[2]
   Al[3] := Hold * Al[3]
            + Inc * (Al[3] :+: (Al[2] * Al[1] * Al[0] * CyIn))
            + Dec * (Al[3] :*: (Al[2] + Al[1] + Al[0] + /CyIn))
            + Load * MAdd[3]
   Ah[0] := Hold * Ah[0]
            + Inc * (Ah[0] :+: HalfCy)
            + Dec * (Ah[0] :*: /HalfCy)
            + Load * MAdd[4]
   Ah[1] := Hold * Ah[1]
            + Inc * (Ah[1] :+: (Ah[0] * HalfCy))
            + Dec * (Ah[1] :*: (Ah[0] + /HalfCy))
            + Load * MAdd[5]
   Ah[2] := Hold * Ah[2]
            + Inc * (Ah[2] :+: (Ah[1] * Ah[0] * HalfCy))
            + Dec * (Ah[2] :*: (Ah[1] + Ah[0] + /HalfCy))
            + Load * MAdd[6]
   Ah[3] := Hold * Ah[3]
            + Inc * (Ah[3] :+: (Ah[2] * Ah[1] * Ah[0] * HalfCy))
            + Dec * (Ah[3] :*: (Ah[2] + Ah[1] + Ah[0] + /HalfCy))
            + Load * MAdd[7]
   Ah[4] := Hold * Ah[4]
            + Inc * (Ah[4] :+: (Ah[3] * Ah[2] * Ah[1] * Ah[0] * HalfCy))
            + Dec * (Ah[3] :*: (Ah[3] * Ah[2] + Ah[1] + Ah[0] + /HalfCy))
            + Load * MAdd[8]
   END
      ELSE      ;BusTrnotLoc1 or ForceHld active. Recycle all registers.
         BEGIN
         Al[0] := Al[0]
         Al[1] := Al[1]
         Al[2] := Al[2]
         Al[3] := Al[3]
         Ah[0] := Ah[0]
         Ah[1] := Ah[1]
         Ah[2] := Ah[2]
         Ah[3] := Ah[3]
         Ah[4] := Ah[4]
         END
```

1.4 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON BusTrnotLoc1 CyIn Al[0..3] Ah[0..4] HalfCy ForceHld
SETF /Clock /Cyin LoadSim /BusTrnotLoc1 /ForceHld
CLOCKF Clock
        ;Load address #01010 1100 (160 + 12 = 172)
```

```
SETF /MAdd[8]  MAdd[7] /MAdd[6]  MAdd[5] /MAdd[4]
SETF  MAdd[3]  MAdd[2] /MAdd[1] /MAdd[0]
SETF LoadSim
CLOCKF Clock
CHECK /Ah[4]  Ah[3] /Ah[2]  Ah[1] /Ah[0]  Al[3]  Al[2] /Al[1] /Al[0]

        ;Inc (CyIn is not set, so nothing should happen)
SETF IncSim
CLOCKF Clock
CLOCKF Clock
CHECK /Ah[4]  Ah[3] /Ah[2]  Ah[1] /Ah[0]  Al[3]  Al[2] /Al[1] /Al[0]
SETF CyIn       ;CyIn set (used to check HalfCy also - see trace output)
FOR i := 1 TO 5 DO
   BEGIN
      CLOCKF Clock
   END
                ;Check for address #01011 0001 (176 + 1 = 177)
CHECK /Ah[4]  Ah[3] /Ah[2]  Ah[1]  Ah[0] /Al[3] /Al[2] /Al[1]  Al[0]

        ;Load address #00101 0011 (80 + 3 = 83)
SETF /MAdd[8] /MAdd[7]  MAdd[6] /MAdd[5]  MAdd[4]
SETF /MAdd[3] /MAdd[2]  MAdd[1]  MAdd[0]
SETF LoadSim
SETF /CyIn
CLOCKF Clock
CHECK /Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0] /Al[3] /Al[2]  Al[1]  Al[0]

        ;Dec (CyIn is not set, so nothing should happen)
SETF DecSim
CLOCKF Clock
CLOCKF Clock
CHECK /Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0] /Al[3] /Al[2]  Al[1]  Al[0]
SETF CyIn       ;CyIn set (used to check HalfCy also - see trace output)
FOR i := 1 TO 5 DO
   BEGIN
      CLOCKF Clock
   END
                ;Check for address #00100 1110 (64 + 14 = 78)
CHECK /Ah[4] /Ah[3]  Ah[2] /Ah[1] /Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]

        ;Load address #00000 0010
SETF /MAdd[8] /MAdd[7] /MAdd[6] /MAdd[5] /MAdd[4]
SETF /MAdd[3] /MAdd[2]  MAdd[1] /MAdd[0]
SETF LoadSim
CLOCKF Clock

        ;Dec - test underflow
SETF DecSim
FOR i := 1 TO 5 DO
   BEGIN
      CLOCKF Clock
   END
                ;Check for address #11111 1101
CHECK  Ah[4]  Ah[3]  Ah[2]  Ah[1]  Ah[0]  Al[3]  Al[2] /Al[1]  Al[0]

        ;Load address #11111 1110
SETF  MAdd[8] MAdd[7]  MAdd[6]  MAdd[5]  MAdd[4]
SETF  MAdd[3]  MAdd[2]  MAdd[1] /MAdd[0]
SETF LoadSim
```

```
CLOCKF Clock

        ;Inc – test overflow
SETF IncSim
FOR i := 1 TO 5 DO
   BEGIN
      CLOCKF Clock
   END
                ;Check for address #00000 0011
CHECK /Ah[4] /Ah[3] /Ah[2] /Ah[1] /Ah[0] /Al[3] /Al[2]  Al[1]  Al[0]

        ;Load address #10101 1110
SETF  MAdd[8] /MAdd[7]  MAdd[6] /MAdd[5]  MAdd[4]
SETF  MAdd[3]  MAdd[2]  MAdd[1] /MAdd[0]
SETF LoadSim
CLOCKF Clock

        ;Operation Hold
SETF HoldSim
FOR i := 1 TO 5 DO
   BEGIN
      CLOCKF Clock
   END
                ;Check same address
CHECK  Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]

        ;Test BusTrnotLoc1 active with Inc and Dec operations
SETF BusTrnotLoc1
CHECK ^Ah[0..4] ^Al[0..3]
SETF IncSim
CLOCKF Clock
                ;Check same address
SETF /BusTrnotLoc1
CHECK  Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]
SETF BusTrnotLoc1
SETF DecSim
CLOCKF Clock
                ;Check same address
SETF /BusTrnotLoc1
CHECK  Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]
        ;Test ForceHld pin with Inc and Dec operations
SETF ForceHld
SETF IncSim
CLOCKF Clock
                ;Check same address
CHECK  Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]
SETF DecSim
CLOCKF Clock
                ;Check same address
CHECK  Ah[4] /Ah[3]  Ah[2] /Ah[1]  Ah[0]  Al[3]  Al[2]  Al[1] /Al[0]
TRACE_OFF
;-------------------------------------------------------------------
```

2. Unidade escalar

2.1 Lógica de controlo e 10 LSB do registo de deslocamento

```
; NET2PDS - vers 4.05.21 (12/06/1990) - (c)1990 Advanced Micro Devices, Inc.
; PDS file created    92/07/8
; Sourcefile: "SCALARX.jnf"

TITLE            GLiTCH Controller Design - MACH version for Scalar Unit
REVISION         1.10
PATTERN          A
AUTHOR           Henrique Santos
COMPANY          University of Minho
DATE             12/11/92

CHIP     _scalarx    MACH210

;Notes:
;   1. Version 1.10 doesn't use WrPulse signal. For the same purpose it uses
;the stoppable version of he Controller Clock.
;This is to avoid any unstable period before the Clk rising edge, that would
;be critical for the 20ns setup time required by the sequencer. From this,
;flags will be available only on the "next" clock cycle. THIS UNIT WILL BE
;SENSIBLE TO CLOCK SKEWS from the Seq., Data Store and GLiTCH!!
;

  NODE 5 _2_RI1 REGISTERED
  NODE 9 _2_RHOLD REGISTERED
  PIN 7, 8 I[1..0]
  PIN 9 BUSTRNOTLOC1
  PIN 13 S_Clk          ;New to version 1.10
  PIN 35 CLK
  NODE 3 _2_RI0 REGISTERED
  NODE 7 _2_RENRW REGISTERED
  PIN 6 CNTRB COMBINATORIAL
  PIN 37 ENRW
  PIN 5 CNTRA COMBINATORIAL
  PIN 4 ARITH COMBINATORIAL
  PIN 36 HOLD
  PIN 3 EN COMBINATORIAL
  NODE 58, 56, 54, 52, 50, 42, 40, 38, 36, 34 QOUT[9..0] REGISTERED OPAIR DATA[9..0]
  PIN 39, 40, 41, 42, 43, 28, 27, 26, 25, 24 DATA[9..0] REGISTERED
  PIN 20 SHIN
  PIN 19, 18, 16, 15, 14, 17, 33, 32, 11, 10 MASK[9..0]
  PIN 21 ZERO COMBINATORIAL
  PIN 38 SHOUT COMBINATORIAL
  PIN 31, 30, 29 FLAG[2..0] COMBINATORIAL
  PIN 2 NOTENWP

;-------------------------------------------------------------------------
;The following macros (1 PT each) are used to avoid double array passes on
;outputs NOTENWP and CNTRA

STRING ENstr    '_2_RI0 * _2_RENRW * /_2_RHOLD'
STRING ARITHstr '/_2_RI1 * _2_RI0 * /_2_RENRW * /_2_RHOLD'

GROUP MACH_SEG_C DATA[0..4] FLAG[0..2]
GROUP MACH_SEG_D DATA[5..9] SHOUT
GROUP MACH_SEG_A _2_RI1 _2_RI0 _2_RHOLD _2_RENRW CNTRA CNTRB

EQUATIONS
; Control equations generated by NET2PDS from .SCH file
```

```
 NOTENWP = (/_2_RI1 * /_2_RI0 * /_2_RENRW * /_2_RHOLD) + _2_RHOLD + (ENstr)

 _2_RI1 = (I[1] * (VCC * /BUSTRNOTLOC1)) + (_2_RI1 * (VCC * BUSTRNOTLOC1))
 _2_RI1.clkf = CLK
 _2_RI1.setf = GND
 _2_RI1.rstf = GND

 CNTRB = (/_2_RI0 * _2_RENRW * /_2_RHOLD) + (_2_RI1 * /_2_RI0 * /_2_RENRW *
        /_2_RHOLD)

 _2_RI0 = (I[0] * (VCC * /BUSTRNOTLOC1)) + (_2_RI0 * (VCC * BUSTRNOTLOC1))
 _2_RI0.clkf = CLK
 _2_RI0.setf = GND
 _2_RI0.rstf = GND

 _2_RENRW = (ENRW * (VCC * /BUSTRNOTLOC1)) + (_2_RENRW * (VCC *
        BUSTRNOTLOC1))
 _2_RENRW.clkf = CLK
 _2_RENRW.setf = GND
 _2_RENRW.rstf = GND

 CNTRA = (_2_RI1 * /_2_RI0 * /_2_RENRW * /_2_RHOLD) + (_2_RI1 * _2_RI0 *
        /_2_RENRW * /_2_RHOLD) + (ARITHstr)

 ARITH = ARITHstr

 _2_RHOLD = (HOLD * (VCC * /BUSTRNOTLOC1)) + (_2_RHOLD * (VCC *
        BUSTRNOTLOC1))
 _2_RHOLD.clkf = CLK
 _2_RHOLD.setf = GND
 _2_RHOLD.rstf = GND

 EN = (ENstr) * /BUSTRNOTLOC1

;Shift register operation
 QOUT[9..0].clkf = S_Clk
 QOUT[9..0].setf = GND
 QOUT[9..0].rstf = GND
 DATA[9..0].trst = EN
 DATA[9..0].clkf = S_Clk
 DATA[9..0].setf = GND
 DATA[9..0].rstf = GND

 CASE (CNTRB, CNTRA) BEGIN
   #b00: BEGIN
           QOUT[9..0] = QOUT[9..0]
         END
   #b01: BEGIN
           QOUT[9] = SHIN
           QOUT[8..0] = QOUT[9..1]
         END
   #b10: BEGIN
           QOUT[9..0] = DATA[9..0] * MASK[9..0]
         END
   #b11: BEGIN
           QOUT[9..1] = QOUT[8..0]
           QOUT[0] = GND
         END
   END ;Case
```

```
ZERO = /QOUT[9] * /QOUT[8] * /QOUT[7] * /QOUT[6] * /QOUT[5] * /QOUT[4] *
       /QOUT[3] * /QOUT[2] * /QOUT[1] * /QOUT[0]

SHOUT = QOUT[9]

FLAG[2..0] = QOUT[2..0]

 DATA[9] = { QOUT[9] }
 DATA[8] = { QOUT[8] }
 DATA[7] = { QOUT[7] }
 DATA[6] = { QOUT[6] }
 DATA[5] = { QOUT[5] }
 DATA[4] = { QOUT[4] }
 DATA[3] = { QOUT[3] }
 DATA[2] = { QOUT[2] }
 DATA[1] = { QOUT[1] }
 DATA[0] = { QOUT[0] }
```

2.2 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
;Modifications from previous version:
; - NEW TO VER 1.10: WrPulse is no longer used, See note 1 on SCALARX.PDS.
; - The low nible chip implements 10 bits and generates all control signals.
; - FHld function is now generated inside the chip. These signal was the
;   logical OR of HOLD (from micromemory) and BusTrnotLoc1.
;   HOLD and BusTrnotLoc1 have different meanings. To HOLD current value and
;   disable outputs and BusTrnotLoc1 to recycle instruction register - hold
;   instruction - and also disable output
; - HighBit[0..1] and ShiftIn are no longer used. Associated function is
;   handled inside the chip. Only 2 signals link the two chips.
; - notOE is no longer used.
; - CtrReg[1..0] was changed to CNTRA and CNTRB
; - notEn was changed to EN
; - ExnotEn is no longer used. Associated function is handled in side
;   the chip.
; - notEn_WR was changed to NOTENRW
; - A new signal ARITH was created to identify an arithmatich shift (only
;   afects high nible chip).
; - Internal NODES QOUT[0..9] are not output or trace, except SHOUT (QOUT9),
;   used to pass high bit to high nible, and FLAG[2..0] (QOUT[2..0]) used as
;   flags.
; - NOTENWP can be ignored because the shift register is in hold mode whenever
;   this signal is active. The only exception is when BusTrnotLoc1 goes high
;   but in that case the writepulse is suspended by the clock generator.
;-------------------------------------------------------------------------------
; Declaration Segment
;Scalar Unit's Instruction set conversion
;BUSTRNOTLOC1 HOLD ENRW I1 I0 | CNTRA CNTRB  EN NOTENWP ARITH|
;-------------------------------------------------------------------------
;      1       (recycle all)  |  LastValue   0     LastValue | HOLD State
;      0       1     X  X  X  |   0(X)  0(X) 0     1      0  | HOLD State
;      0       0     0  0  0  |   0(X)  0(X) 0     1      0  | HOLD State
;      0       0     0  0  1  |   1     0    0     0      1  | ASR
;      0       0     0  1  0  |   1     1    0     0      0  | LSL
;      0       0     0  1  1  |   1     0    0     0      0  | LSR
```

```
;      0       0     1  X  0  |   0     1    0     0      0  | LoadDReg
;      0       0     1  X  1  |   0(X)  0(X) 1     1      0  | OutputDReg
;-------------------------------------------------------------------------
;Input instructions
STRING  Hold_sim        '/ENRW   /I[1]   /I[0]'
STRING  ASR_sim         '/ENRW   /I[1]    I[0]'
STRING  LSL_sim         '/ENRW    I[1]   /I[0]'
STRING  LSR_sim         '/ENRW    I[1]    I[0]'
STRING  LDR_sim         ' ENRW           /I[0]'
STRING  ODR_sim         ' ENRW            I[0]'

;Output instructions - CtrReg only!
STRING  OutHOLDSim      '/CNTRA /CNTRB'
STRING  OutLDRSim       '/CNTRA  CNTRB'
STRING  OutSRSim        ' CNTRA /CNTRB'
STRING  OutSLSim        ' CNTRA  CNTRB'

SIMULATION
TRACE_ON CLK S_Clk HOLD ENRW I[0..1] BUSTRNOTLOC1 ;External control interf.
         SHIN ;Shiftin from high nible
         SHOUT ;Shiftout to high nible
         MASK[0..9] DATA[0..9] FLAG[0..2] ;Data Signals
         CNTRA CNTRB ZERO NOTENWP ARITH EN ;Output control signals to high n.

SETF /CLK /BUSTRNOTLOC1 /HOLD
        ;Test Control Generation Block
                ;Hold Operation (Instruction)
SETF Hold_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
      CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
   END
        ;Arithmetic Shift Right
SETF ASR_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
      CHECK  OutSRSim /EN /NOTENWP  ARITH;
   END
        ;Logical Shift Left
SETF LSL_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
      CHECK  OutSLSim /EN /NOTENWP /ARITH;
   END
        ;Logical Shift Right
SETF LSR_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
      CHECK  OutSRSim /EN /NOTENWP /ARITH;
   END
        ;Load Data Register Operation
SETF LDR_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
```

```
      CHECK  OutLDRSim /EN /NOTENWP /ARITH;
   END
SETF /I[1]      ;NOTE: It must not depend on I[1] -- it doesn't care
CLOCKF CLk
CHECK  OutLDRSim /EN /NOTENWP /ARITH;
SETF  I[1]
CLOCKF Clk
CHECK  OutLDRSim /EN /NOTENWP /ARITH;
        ;Output Data Register Operation
SETF ODR_sim
FOR i := 0 TO 2 DO
   BEGIN
      CLOCKF Clk
      CHECK  OutHOLDSim  EN  NOTENWP /ARITH;
   END
SETF /I[1]      ;NOTE: It must not depend on I[1] -- it doesn't care
CLOCKF Clk
CHECK  OutHOLDSim  EN  NOTENWP /ARITH;
SETF  I[1]
CLOCKF Clk
CHECK  OutHOLDSim  EN  NOTENWP /ARITH;
        ;hold signal set. Must keep register value outputing Hold instruction
        ;and disable output.
SETF HOLD
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF Hold_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF ASR_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF LSL_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF LSR_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF LDR_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF ODR_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF /HOLD
        ;BUSTRNOTLOC1 control. Must recycle instruction register and disable
        ;output.
SETF Hold_sim
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH;
SETF BUSTRNOTLOC1
SETF ASR_sim    ;Arithmatic Shift Right
CLOCKF Clk
CHECK  OutHOLDSim /EN  NOTENWP /ARITH ;Must keep instruction and control
                                      ;signals
SETF /BUSTRNOTLOC1
CLOCKF Clk
CHECK  OutSRSim /EN /NOTENWP  ARITH
SETF BUSTRNOTLOC1
```

```
SETF ODR_sim    ;another test, this time with ODR to active EN... later
CLOCKF Clk
CHECK  OutSRSim /EN /NOTENWP  ARITH ;Must keep instruction and control
                                    ;signals
SETF /BUSTRNOTLOC1
CLOCKF Clk
CHECK  OutHOLDSim  EN  NOTENWP /ARITH
SETF  BUSTRNOTLOC1                  ;Suspend here must only disable output.
CLOCKF Clk
CHECK             /EN  NOTENWP /ARITH;

;Test shitf register operation, using microcontroller instructions
SETF /HOLD /BUSTRNOTLOC1
CLOCKF Clk
        ;Load a value into data register (#b01 0010 0101)
SETF  LDR_sim
CLOCKF Clk
SETF  Data[9]  Data[8]  Data[7]  Data[6]  Data[5]  Data[4] /Data[3]  Data[2]
/Data[1]  Data[0]
SETF /Mask[9]  Mask[8] /Mask[7] /Mask[6]  Mask[5] /Mask[4]  Mask[3]  Mask[2]
Mask[1]  Mask[0]
CLOCKF S_Clk
;CHECK ^Data[9..0] -- doesn't test because of previous SETF on DATA lines
CHECK /QOUT[9]  QOUT[8] /QOUT[7] /QOUT[6]  QOUT[5] /QOUT[4] /QOUT[3]  QOUT[2]
/QOUT[1]  QOUT[0]
CHECK                                                                FLAG[2]
/FLAG[1]  FLAG[0]

        ;Test output
SETF  ODR_sim
CLOCKF Clk
CHECK /Data[9]  Data[8] /Data[7] /Data[6]  Data[5] /Data[4] /Data[3]  Data[2]
/Data[1]  Data[0]
CHECK                                                                FLAG[2]
/FLAG[1]  FLAG[0]
        ;logical shift left twice - final #b00 1001 0100
SETF LSL_sim
CLOCKF CLk
CLOCKF S_Clk
CLOCKF S_Clk
CHECK ^Data[9..0]
CHECK /QOUT[9] /QOUT[8]  QOUT[7] /QOUT[6] /QOUT[5]  QOUT[4] /QOUT[3]  QOUT[2]
/QOUT[1] /QOUT[0]
CHECK                                                                FLAG[2]
/FLAG[1] /FLAG[0]
        ;Shift right three times (high bit 1) - final #b11 1001 0010
SETF LSR_sim SHIN
CLOCKF Clk
CLOCKF S_Clk
CLOCKF S_Clk
CLOCKF S_Clk
CHECK ^Data[9..0]
CHECK  QOUT[9]  QOUT[8]  QOUT[7] /QOUT[6] /QOUT[5]  QOUT[4] /QOUT[3] /QOUT[2]
QOUT[1] /QOUT[0]
CHECK                                                               /FLAG[2]
FLAG[1] /FLAG[0]
        ;Shift right ten times (high bit is 0) - final #b00 0000 0000
        ;also test ZERO on all steps
SETF ASR_sim /SHIN
```

```
CLOCKF Clk
FOR i := 1 TO 10 DO
   BEGIN
      CHECK /ZERO
      CLOCKF S_Clk
   END
CHECK ^Data[9..0]
CHECK /QOUT[9] /QOUT[8] /QOUT[7] /QOUT[6] /QOUT[5] /QOUT[4] /QOUT[3] /QOUT[2]
/QOUT[1] /QOUT[0]
CHECK                                                              /FLAG[2]
/FLAG[1] /FLAG[0]
CHECK ZERO
TRACE_OFF
```

2.3 6 MSB do registo de deslocamento

```
; NET2PDS - vers 4.05.21 (12/06/1990) - (c)1990 Advanced Micro Devices, Inc.
; PDS file created    10/09/2026   10:39a
; Sourcefile: "SCALARHX.jnf"

TITLE            GLiTCH Controller - MACH version for High nible of SU
REVISION         1.10
PATTERN          MACH
AUTHOR           Henrique Santos
COMPANY          UM
DATE            12/11/1992

CHIP     _SCALARHX    MACH110
;Notes:
;   1. Version 1.10 doesn't use WrPulse signal. It uses the stoppable version
;of the Controller Clock. See note 1 on SCALARX.PDS file.
;

  PIN 10 EN
  NODE 7, 19, 29, 26, 21, 18 QOUT[15..10] REGISTERED OPAIR DATA[15..10]
  PIN 7, 25, 39, 36, 27, 24 DATA[15..10] REGISTERED
  PIN 13 CNTRB
  PIN 11 CNTRA
  PIN 35 S_Clk          ;Just Clk in version 1.00
  PIN 5, 3, 15, 16, 33, 32 MASK[15..10]
  PIN 19 SHIN
  PIN 21 ARITH
  PIN 20 ZERO_IN
  PIN 28 ZERO_OUT COMBINATORIAL
  PIN 2 FLAG15 COMBINATORIAL
  PIN 26 SHOUT COMBINATORIAL

;-------------------------------------------------------------------------

GROUP MACH_SEG_B DATA[14..10] ZERO_OUT SHOUT
GROUP MACH_SEG_A DATA[15] FLAG15
EQUATIONS

 ZERO_OUT = ZERO_IN * /QOUT[15] * /QOUT[14] * /QOUT[13] * /QOUT[12] *
            /QOUT[11] * /QOUT[10]

 FLAG15 = QOUT[15]
```

```
 SHOUT = QOUT[10]

;Shift register operation
 QOUT[15..10].clkf = S_Clk
 QOUT[15..10].setf = GND
 QOUT[15..10].rstf = GND
 DATA[15..10].trst = EN
 DATA[15..10].clkf = S_Clk
 DATA[15..10].setf = GND
 DATA[15..10].rstf = GND

 CASE (CNTRB, CNTRA) BEGIN
   #b00: BEGIN
           QOUT[15..10] = QOUT[15..10]
         END
   #b01: BEGIN
           IF (ARITH) THEN BEGIN QOUT[15] = QOUT[15] END
              ELSE BEGIN QOUT[15] = GND END
           QOUT[14..10] = QOUT[15..11]
         END
   #b10: BEGIN
           QOUT[15..10] = DATA[15..10] * MASK[15..10]
         END
   #b11: BEGIN
           QOUT[15..11] = QOUT[14..10]
           QOUT[10] = SHIN
         END
   END ;Case

 DATA[15] = { QOUT[15] }
 DATA[14] = { QOUT[14] }
 DATA[12] = { QOUT[12] }
 DATA[13] = { QOUT[13] }
 DATA[11] = { QOUT[11] }
 DATA[10] = { QOUT[10] }
```

2.4 Vectores de teste

Atendendo à semelhança das descrições dos dois blocos que constituem a SU, foi possível utilizar os vectores de teste anteriores (secção 2.3), modificando os índices e ignorando a secção dedicada à verificação da lógica de controlo da SU.

3. Selector de *tags*

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    GLiTCH System Controller - Tag Unit
PATTERN  A
REVISION 2.00
AUTHOR   Henrique Santos
COMPANY  University ofMinho
DATE     04/11/92

CHIP  pal_tu  PAL22V10
```

```
;---------------------------------- PIN Declarations ---------------
PIN  1          Clk                                     COMBINATORIAL ; CLOCK
PIN  2          TagUp                                   COMBINATORIAL ; INPUT
PIN  3          TagDown                                 COMBINATORIAL ; INPUT
PIN  4..5       MInstr[0..1]                            COMBINATORIAL ; INPUT
PIN  6          Mloop                                   COMBINATORIAL ; INPUT
PIN  7          Suspend                                 COMBINATORIAL ; INPUT
PIN  8          notHold                                 COMBINATORIAL ; INPUT
;PIN 9..11 are free inputs
PIN  12         GND                                                   ;
PIN  13         Reset                                   COMBINATORIAL ; INPUT
PIN  23         TagBellow                               COMBINATORIAL ; OUTPUT
PIN  14         TagAbove                                COMBINATORIAL ; OUTPUT
PIN  18..19     delay1[0..1]                            REGISTERED ; OUTPUT
PIN  15,22      delay2[0..1]                            REGISTERED ; OUTPUT
PIN  16,21      delay3[0..1]                            REGISTERED ; OUTPUT
;PIN  17 and 20     can be used as a delay4[0..1]
PIN  24         VCC                                                   ;

NODE 1          GlobalAR                                COMBINATORIAL ; Reset
                                                                      ; node
; Changes from previos version:
;  1. PIN 8 is new. It is an input signal (notHold), from micromemory, to
;allow the instruction field to be shared with other unit. Its effect is the
;same as the Hold instruction, ie, it only affects the content of the
;first pipe register not allowing it to load a new instruction. Instead it
;will recycle its old value. It is active low, so the associated signal in
;the other unit must be active high!
; Notes:
;  1. The input instruction (3 bits) is decoded and then registered (in 2
;flip-flops). This aproach saves one flip-flop for each pipe stage.
;  2. Each group of delay registers is responsable for one clock delay. With
;this PAL only 3 clock cycles are implemented. If one or two of the delays are
;not necessary the correspondent pins can be used for another purpose. But it
;should be better to keep this chip as it is!
;  3. To change the delay used, alter the CASE control statement (see local
;note) and possible the IF statement that implements the pipe structure.
;  4. There is a small diference betwen the HOLD statement and the Suspend
;function. HOLD keeps only the state of delay1, while Suspend recycles all
;pipe registers.
;---------------------------------- Declaration Segment ------------
        ;Microinstruction (input) definition.
STRING  HOLD            '/MLoop * /MInstr[1] * /MInstr[0]'
STRING  HOLD_sim        '/MLoop   /MInstr[1]   /MInstr[0]'
STRING  HOLD_case       '#b000'
STRING  CHAIN           ' MLoop * /MInstr[1] * /MInstr[0]'
STRING  CHAIN_sim       ' MLoop   /MInstr[1]   /MInstr[0]'
STRING  CHAIN_case      '#b100'
STRING  SET             '/MLoop *  MInstr[1] * /MInstr[0]'
STRING  SET_sim         '/MLoop    MInstr[1]   /MInstr[0]'
STRING  SET_case        '#b010'
STRING  CLEAR           '/MLoop * /MInstr[1] *  MInstr[0]'
STRING  CLEAR_sim       '/MLoop   /MInstr[1]    MInstr[0]'
STRING  CLEAR_case      '#b001'
        ;Decoded internal 2-bit instruction:
        ;       delayx[1] delayx[0]   Name
        ;CHAIN     0         0        CHAINint
        ;SET       0         1        SETint
```

```
        ;CLEAR     1         0        CLEARint
        ;-----     1         1        Not used
        ;x represents the order of the last pipe register, the one that
        ;commands the multiplexer. In the following declaration x is
        ;replaced by the actual pipe order.
STRING  CHAINint        '/delay3[1] * /delay3[0]'
STRING  CHAINint_sim    '/delay3[1]   /delay3[0]'
STRING  CHAINint_case   '#b00'
STRING  SETint          '/delay3[1] *  delay3[0]'
STRING  SETint_sim      '/delay3[1]    delay3[0]'
STRING  SETint_case     '#b01'
STRING  CLEARint        ' delay3[1] * /delay3[0]'
STRING  CLEARint_sim    ' delay3[1]   /delay3[0]'
STRING  CLEARint_case   '#b10'

GROUP PipeRegs delay1[0..1] delay2[0..1] delay3[0..1]

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
GlobalAR.RSTF = Reset   ;Asynchronous reset internal node is under control
                        ;of Init pin.
TagBellow.TRST = VCC
TagAbove.TRST = VCC
PipeRegs.TRST = VCC     ;Outputs from pipe regs are allways disable.
                        ;ENABLE THEM JUST FOR SIMULATION PURPOSES
        ;GLiTCH Tag flag's multiplexer control. Use only one and put
        ;comments on the other two. It must match the declarations above.
;CASE (delay1[1..0])
;CASE (delay2[1..0])
CASE (delay3[1..0])
BEGIN
   CHAINint_case:
      BEGIN
         TagAbove = TagDown
         TagBellow = TagUp
      END
   SETint_case:
      BEGIN
         TagAbove = VCC
         TagBellow = VCC
      END
   CLEARint_case:
      BEGIN
         TagAbove = GND
         TagBellow = GND
      END
   OTHERWISE:              ;An incorrect code would never happen! anyway...
      BEGIN
         TagAbove = TagDown
         TagBellow = TagUp
      END
END
        ;Decode Microinstruction and latch it into delay1[]
IF (/Suspend * notHold) THEN
   BEGIN
      CASE (MLoop, MInstr[1..0])
      BEGIN
         HOLD_case:
            BEGIN
```

```
               delay1[1..0] = delay1[1..0]
            END
         CHAIN_case:
            BEGIN
               delay1[1..0] = CHAINint_case
            END
         SET_case:
            BEGIN
               delay1[1..0] = SETint_case
            END
         CLEAR_case:
            BEGIN
               delay1[1..0] = CLEARint_case
            END
         OTHERWISE:     ;If arrives an unexpected code, behaves like HOLD
            BEGIN
               delay1[1..0] = delay1[1..0]
            END
      END ; -- CASE
   END ; -- IF
    ELSE
       BEGIN
          delay1[1..0] = delay1[1..0]
       END

        ;Pipe structure. All other delay registers only depend on Suspend
        ;NOTE: Use comments to avoid using one or both delay registers
        ;Change decode equations above as required.
IF (/Suspend) THEN
   BEGIN
      delay2[1..0] = delay1[1..0]
      delay3[1..0] = delay2[1..0]
   END
    ELSE
       BEGIN
          delay2[1..0] = delay2[1..0]
          delay3[1..0] = delay3[1..0]
       END
```

3.1 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON Clk TagUp TagBellow TagDown TagAbove
         MLoop MInstr[1..0] delay1[1..0] delay2[1..0]  delay3[1..0]
         Suspend Reset notHold
;LabPro Power-up reset doesn't work, so the first vector will give an error.
;Avoid that forcing all output pins to "don't care" before test vector.
SETF /Clk /Suspend  Reset /TagUp /TagDown
CLOCKF Clk
CHECK /delay1[1..0] /delay2[1..0] /delay3[1..0] /TagBellow /TagAbove
        ;Test Reset function
PRELOAD delay1[1..0] delay2[1..0] delay3[1..0]
SETF Reset
CHECK /delay1[1..0] /delay2[1..0] /delay3[1..0] /TagBellow /TagAbove
CLOCKF Clk
CHECK /delay1[1..0] /delay2[1..0] /delay3[1..0] /TagBellow /TagAbove
```

```
SETF /Reset
        ;Check GLiTCH Tag flag's multiplexer (depend on delay3[] only)
PRELOAD CHAINint_sim            ;Code for CHAIN
SETF /TagUp /TagDown
CHECK /TagBellow /TagAbove
SETF  TagUp /TagDown
CHECK  TagBellow /TagAbove
SETF /TagUp  TagDown
CHECK /TagBellow  TagAbove
SETF  TagUp  TagDown
CHECK  TagBellow  TagAbove
PRELOAD SETint_sim              ;Code for SET
SETF /TagUp /TagDown
CHECK  TagBellow  TagAbove
SETF  TagUp /TagDown
CHECK  TagBellow  TagAbove
SETF /TagUp  TagDown
CHECK  TagBellow  TagAbove
SETF  TagUp  TagDown
CHECK  TagBellow  TagAbove
PRELOAD CLEARint_sim            ;Code for CLEAR
SETF /TagUp /TagDown
CHECK /TagBellow /TagAbove
SETF  TagUp /TagDown
CHECK /TagBellow /TagAbove
SETF /TagUp  TagDown
CHECK /TagBellow /TagAbove
SETF  TagUp  TagDown
CHECK /TagBellow /TagAbove
PRELOAD delay3[1] delay3[0]     ;Code not used in this version!!!!
SETF /TagUp /TagDown                    ;behaves like CHAIN
CHECK /TagBellow /TagAbove
SETF  TagUp /TagDown
CHECK  TagBellow /TagAbove
SETF /TagUp  TagDown
CHECK /TagBellow  TagAbove
SETF  TagUp  TagDown
CHECK  TagBellow  TagAbove
;------------------------------ Depend of the pipe size -----------------
        ;Check Decode function (it involves delay1[], MLoop, MInstr[], notHold
        ;and pipeline activity - assuming 3 pipe stage)
        ;Reset takes the chip to CHAINint state
SETF Reset /Suspend /TagUp /TagDown notHold CHAIN_sim
CHECK /delay1[1..0] /delay2[1..0] /delay3[1..0] /TagBellow /TagAbove
SETF /Reset CHAIN_sim
CLOCKF Clk
SETF SET_sim
CLOCKF Clk
SETF CLEAR_sim
CLOCKF Clk              ;3th clock pulse, first instruction must reach
CHECK CHAINint_sim      ;the pipeline's end
CHECK /delay2[1]  delay2[0]      ;Assumes '#b01' is the SET code
CHECK  delay1[1] /delay1[0]      ;Assumes '#b10' is the CLEAR code
        ;Now check Suspend function
SETF Suspend
CLOCKF Clk
CLOCKF Clk
CLOCKF Clk
CHECK CHAINint_sim
```

```
CHECK /delay2[1]  delay2[0]      ;Assumes '#b01' is the SET code
CHECK  delay1[1] /delay1[0]      ;Assumes '#b10' is the CLEAR code
        ;Now check HOLD instruction
SETF /Suspend HOLD_sim  ;HOLD instruction => delay1[] must keep its
CLOCKF Clk              ;previous state, i.e., CLEAR instruction's code
CHECK SETint_sim        ;The rest of the pipeline must keep running
CLOCKF Clk
CHECK CLEARint_sim
CLOCKF Clk
CHECK CLEARint_sim      ;At this point all the pipe registers are loaded
CLOCKF Clk              ;with CLEAR instruction's code
CLOCKF Clk
        ;Now an invalid code '#b111' after a SET instruction
SETF SET_sim
CLOCKF Clk              ;1st stage with SET, 2nd with previous code
CHECK CLEARint_sim      ;3th stage still with previous code
SETF MLoop MInstr[1..0] ;Set invalid code
CLOCKF Clk              ;1st stage must keep SET, 2nd with SET
CHECK CLEARint_sim      ;3th stage still with previous code
        ;Try another invalid code '#b011'
SETF /MLoop MInstr[1..0]
CLOCKF Clk              ;1st stage continues keeping SET, 2nd with SET now
CHECK SETint_sim        ;3th stage with SET
SETF CHAIN_sim
CLOCKF Clk              ;1st stage with CHAIN, 2nd with SET
CHECK SETint_sim        ;3th stage with SET
CLOCKF Clk              ;1st stage --------, 2nd with HOLD
CHECK SETint_sim        ;3th stage with SET
CLOCKF Clk              ;1st stage --------, 2nd --------
CHECK CHAINint_sim      ;3th stage with CHAIN
        ;Now check notHold function. First refill the pipeline regs.
SETF CHAIN_sim
CLOCKF Clk
SETF SET_sim
CLOCKF Clk
SETF CLEAR_sim
CLOCKF Clk              ;3th clock pulse, first instruction must reach
CHECK CHAINint_sim      ;the pipeline's end
SETF /notHold           ;SOME BEHAVIOUR OF HOLD INSTRUCTION
CLOCKF Clk              ;previous state, i.e., CLEAR instruction's code
CHECK SETint_sim        ;The rest of the pipeline must keep running
CLOCKF Clk
CHECK CLEARint_sim
CLOCKF Clk
CHECK CLEARint_sim      ;At this point all the pipe registers are loaded
CLOCKF Clk              ;with CLEAR instruction's code
CLOCKF Clk
CHECK CLEARint_sim
TRACE_OFF
;-------------------------------------------------------------------------
```

4. *Bus Arbiter*

4.1 Máquina de estados principal

```
;PALASM Design Description
```

```
;---------------------------------- Declaration Segment ------------
TITLE    GLITCH CONTROLLER BOARD - Bus Arbiter
PATTERN  Main Status Machine
REVISION 3.20
AUTHOR   HENRIQUE SANTOS
COMPANY  UNIVERSITY OF MINHO
DATE     02/04/92

CHIP  PIPECTR  PALCE16V8
;*********************************************************************
; This version:                                                      *
;   - from 3.00, this version implements the SuspGl signal with      *
;     the same timing of the BusTrnotLoc1 signal that is to be       *
;     used to tri-sate all GLiTCH Board staff. The BusTrnotLoc1      *
;     signal is delayed one clock cycle to accomudate that change.   *
;   - from 3.10, this version correctly implement the Reset cond.    *
;     Previous versions have the Normal_Op state as the reset        *
;     target. Now Phase2 is the target and, when it is required,     *
;     the state machine goes through Phase1, to avoid bus            *
;     contentions. THIS IS ONLY REQ FOR MP state machine.            *
;*********************************************************************

;NOTE: MC-- 0 (pin 12) ---------------- is not used an can be used for any
;purpose, while MC 1 (pin 13) has some dependency on the programmed REG MC's
;(see data sheet, /SG0 use...)
;
; THIS DESIGN MUST BE COMPILED WITH SET UP OPTION POLARITY TO "ALL OUTPUTS
; HIGH. IF NOT, THE COMPILER WILL PROGRAM StateBits[1], StateBits[2] and
; RunAux[0] OUTPUTS ACTIVE LOW!!!!!
;
;---------------------------------- PIN Declarations ---------------
PIN  1          Clk                  COMB ; INPUT
PIN  2          BusToTr              COMB ; INPUT from "interface signals"
                                          ;chip
PIN  3          Run                  COMB ; INPUT from Host
PIN  9          Reset                COMB ; INPUT
PIN  10         GND                       ;
PIN  11         notOEn               COMB ; INPUT Enable outputs (set to 0)
        ;The following states bits are also used as outputs control both for
        ;the ERP state machines (ClrFlags) - pattern "interface signals" - and
        ;the external buffers that link all units to shared buses. Note that
        ;BusTrnotLoc2 is also used as the BusGrant flag for the Host.
PIN  [16..19]   StateBits[0..3]      REG  ; [0] - SuspGl
                                          ; [1] - BusTrnotLoc2
                                          ; [2] - ClrFlags
                                          ; [3] - BusTrnotLoc1
PIN  [14..15]   RunAux[0..1]         REG  ; Auxiliary regs to implement a
                                          ;synchronization circuit for Run
PIN  20         VCC                       ;

;       Declaration segment
                ;Condition that flags the begin of a Switch Bus Access
                ;(SBA) operation
;14/2/1992
;THIS IS NOT WORKING AS IT WAS EXPECTED!!! BusToTr WOULD BE SET IN THE NEXT
;CLOCK CYCLE, FROM THE ONE THE EVENT SIGNAL WAS SET. SO BusTrnotLoc1 WOULD
;BE SET ONE CLOCK CYCLE LATER!!!

        ;MP state machine's state codes amd auxiliary macros
```

```
STRING  SuspGl       'StateBits[0]'
STRING  BusTrnotLoc2 'StateBits[1]'
STRING  ClrFlags     'StateBits[2]'
STRING  BusTrnotLoc1 'StateBits[3]'
        ;State codes for StateBits[0..3] in this order
STRING  Normal_Op '#b0000'      ;to use with case statements
STRING  Phase1    '#b1001'      ;
STRING  Phase2    '#b1101'      ;
STRING  RetPhase1 '#b1011'      ;
STRING  RetPhase2 '#b0001'      ;
                                ;to use with simulation statements (CHECK)
STRING  Normal_OpSim '/StateBits[0] /StateBits[1] /StateBits[2] /StateBits[3]'
STRING  Phase1Sim    ' StateBits[0] /StateBits[1] /StateBits[2]  StateBits[3]'
STRING  Phase2Sim    ' StateBits[0]  StateBits[1] /StateBits[2]  StateBits[3]'
STRING  RetPhase1Sim ' StateBits[0] /StateBits[1]  StateBits[2]  StateBits[3]'
STRING  RetPhase2Sim '/StateBits[0] /StateBits[1] /StateBits[2]  StateBits[3]'

        ;Run flag from synchronization state machine, to Host
STRING  RunSync 'RunAux[0]'
        ;To use with case statements - RunAux[1], RunAux[0] is this order
        ;RunAux[0] is also RunSync qualifier
STRING  Waiting  '#b00'
STRING  Standby1 '#b10'
STRING  Active   '#b01'
STRING  Standby2 '#b11'
        ;To use in simulation statments (CHECK)
STRING  WaitingSim      '/RunAux[1] /RunAux[0]'
STRING  Standby1Sim     ' RunAux[1] /RunAux[0]'
STRING  ActiveSim       '/RunAux[1]  RunAux[0]'
STRING  Standby2Sim     ' RunAux[1]  RunAux[0]'

        ;Reset condition definition (easy to change logic state...)
STRING ResetCond 'Reset'        ;Active High

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
                ;Output buffers control
;StateBits[0..3].TRST = /notOEn         not programmable in thid device
;RunAux[0..1].TRST = /notOEn

;-------------------------------------------------------------------------
;Main Process (MP). State transition equations (see documentation)
; ClrFlags output is a qualifier for all the ERP machines, defined under
; "interface signals pattern" chip.

CASE (StateBits[0..3])
   BEGIN
      Normal_Op:
             BEGIN
                IF (ResetCond + BusToTr) THEN BEGIN StateBits[0..3] = Phase1 END
                  ELSE BEGIN StateBits[0..3] = Normal_Op END
             END
      Phase1:
             BEGIN
                StateBits[0..3] = Phase2
             END
      Phase2:
             BEGIN
                IF (ResetCond + /RunSync) THEN BEGIN StateBits[0..3] = Phase2 END
```

```
                  ELSE BEGIN StateBits[0..3] = RetPhase1 END
            END
      RetPhase1:
            BEGIN
               IF (ResetCond) THEN BEGIN StateBits[0..3] = Phase2 END
                 ELSE BEGIN IF (/RunSync) THEN BEGIN StateBits[0..3] = RetPhase2
END
                    ELSE BEGIN StateBits[0..3] = RetPhase1 END END
            END
      RetPhase2:
            BEGIN
               IF (ResetCond) THEN BEGIN StateBits[0..3] = Phase2 END
                 ELSE BEGIN StateBits[0..3] = Normal_Op END
            END
      OTHERWISE:
            BEGIN StateBits[0..3] = Phase1 END ;... and to Phase2 after...
   END

;Run (control signal from Host) -- this qualifier is tested twice before it
;sets or resets the RunSync (RunAux[0]) output, which is the actual qualifier
;used by the MP status machine.

CASE (RunAux[1..0])
   BEGIN
      Waiting:  BEGIN           ;RunSync = 0
                   IF (/ResetCond * Run) THEN BEGIN RunAux[1..0] = Standby1 END
                     ELSE BEGIN RunAux[1..0] = Waiting END
                END
      Standby1: BEGIN           ;RunSync = 0
                   IF (/ResetCond * Run) THEN BEGIN RunAux[1..0] = Active END
                     ELSE BEGIN RunAux[1..0] = Waiting END
                END
      Active:   BEGIN           ;RunSync = 1
                   IF (ResetCond) THEN BEGIN RunAux[1..0] = Waiting END
                     ELSE BEGIN IF (/Run) THEN BEGIN
                                              RunAux[1..0] = Standby2 END
                                ELSE BEGIN RunAux[1..0] = Active END
                           END
                END
      Standby2: BEGIN           ;RunSync = 1
                   IF (ResetCond + /Run) THEN BEGIN RunAux[1..0] = Waiting END
                     ELSE BEGIN RunAux[1..0] = Active END
                END
   END
```

4.2 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
;PRELOAD acts on the Flip-Flops, and not on the outputs. In 16V8 all
;outputs has an output inverter, wich means to produce a low output it's
;necessary to preload the register with a 1.

TRACE_ON Clk BusToTr Run Reset
         SuspGl BusTrnotLoc2 ClrFlags BusTrnotLoc1
         RunAux[1..0]
SETF /Clk  ResetCond /BusToTr /notOEn  ;notOEn allways set
```

```
                                        ;
CLOCKF Clk
CLOCKF Clk
CHECK Phase2Sim WaitingSim
         ;Test Reset condition on all state transitions
SETF  BusToTr  Run                      ;Set qualifier for transition...
PRELOAD  StateBits[0..3] RunAux[1..0]   ;State - Normal_Op | Waiting
CLOCKF Clk
CHECK Phase1Sim WaitingSim
CLOCKF Clk
CHECK Phase2Sim WaitingSim
        ;State -> Phase1                                        | Standby1
PRELOAD /StateBits[0]  StateBits[1]  StateBits[2] /StateBits[3] /RunAux[1]
RunAux[0]
CLOCKF Clk
CHECK Phase2Sim WaitingSim
        ;State -> Phase2                                        | Active
SETF          /Run                      ;Set qualifier for transition...
PRELOAD /StateBits[0] /StateBits[1]  StateBits[2] /StateBits[3]  RunAux[1]
/RunAux[0]
CLOCKF Clk
CHECK Phase2Sim WaitingSim
        ;State -> RetPhase1                                     | Standby2
PRELOAD /StateBits[0]  StateBits[1] /StateBits[2] /StateBits[3] /RunAux[1]
/RunAux[0]
CLOCKF Clk                              ;ClrFlag already active (RunAux[0])
CHECK Phase2Sim WaitingSim
        ;State -> RetPhase2
PRELOAD  StateBits[0]  StateBits[1]  StateBits[2] /StateBits[3] RunAux[1..0]
CLOCKF Clk                              ;
CHECK Phase2Sim
        ;State -> Not specified #b1111
PRELOAD /StateBits[0..3] RunAux[1..0]
CLOCKF Clk
CHECK Phase1Sim
SETF /ResetCond /BusToTr /Run
        ;End of Reset condition test

        ;Chip starts in initial state
PRELOAD StateBits[0..3]
; -- Test Run sync machine
SETF Run
CLOCKF Clk
CHECK Standby1Sim
SETF /Run
CLOCKF Clk
CHECK WaitingSim
SETF  Run
CLOCKF Clk
CLOCKF Clk
CHECK ActiveSim
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK ActiveSim
SETF /Run
CLOCKF Clk
CHECK Standby2Sim
```

```
SETF  Run
CLOCKF Clk
CHECK ActiveSim
SETF /Run
CLOCKF Clk
CLOCKF Clk
CHECK WaitingSim
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK WaitingSim

; -- MP machine (and also part of Run sync machine)
;    it is in intial state (I hope...)
SETF BusToTr            ;event...
CLOCKF Clk
CHECK Phase1Sim
CLOCKF Clk
CHECK Phase2Sim
SETF /BusToTr           ;This would never happen because it depends on
                        ;BusTrnotLoc1!
CLOCKF Clk
CHECK Phase2Sim
        ;Host has exclusive access...
        ;... until it sets Run signal.
FOR i := 0 TO 10 DO
   BEGIN
      CLOCKF Clk
   END
SETF Run
CLOCKF Clk              ;Run is in standby...
CHECK Phase2Sim /RunSync
CLOCKF Clk              ;Now it is Ok!
CHECK Phase2Sim  RunSync
CLOCKF Clk
CHECK RetPhase1Sim      ;Now ClrFlgag is set - this will clear ERP Machine
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK RetPhase1Sim      ;It must wait for Run to be cleared
SETF /Run
CLOCKF Clk              ;Run is in standby...
CHECK RetPhase1Sim  RunSync
CLOCKF Clk              ;Now it is OK!
CHECK RetPhase1Sim /RunSync
CLOCKF Clk
CHECK RetPhase2Sim
CLOCKF Clk
CHECK Normal_OpSim
; -- end of MP machine

TRACE_OFF
;---------------------------------------------------------------------
```

4.3 Gestão de eventos

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    GLITCH CONTROLLER BOARD - Bus Arbiter
PATTERN  Interface Signals
REVISION 3.00
AUTHOR   HENRIQUE SANTOS
COMPANY  UNIVERSITY OF MINHO
DATE     14/02/92

CHIP  PIPECTR  PALCE16V8
;*********************************************************************
; This version:                                                      *
;   - it uses two 16V8 instead of just one chip to improve speed    *
;   - because it needs to provide the BusToTr signal to the clock   *
;     generator (GLiTCH board) before the BusTrnotLoc1 signal.      *
;     This was a result from February meeting in Bristol
;   - this pattern handles only the interface signals              *
;   - BusReq event is handle in a different way because its assync.*
;     nature. It goes through an extra state to check it is real   *
;     set!
;   - it doesn't generate BusGrant signal (see doc)                *
;   - it implements the Single Step facilitie described in last    *
;     documentation.                                               *
;   - it doesn't implement the Critical Code bit                   *
;   - Reset is sync. because hardware architecture!!!              *
;*********************************************************************
; Note: See previous version notes.                                 *
;1. PIN 2..5 go to high impedance state when system is under       *
;control of the Host. Those pins must pulled-down with resistores. *
;*********************************************************************

;---------------------------------- PIN Declarations ---------------
PIN  1          Clk                 COMB ; INPUT
PIN  2          Equal               COMB ; INPUT from Sequencer
PIN  3          Finish              COMB ; INPUT from micromemory
PIN  4          SC_Req              COMB ; INPUT from micromemory
PIN  5          Ss                  COMB ; INPUT from micromemory
PIN  6          BusReq              COMB ; INPUT from Host
PIN  7          ClrFlags            COMB ; INPUT from Main Process Machine
PIN  8          Reset               COMB ; INPUT
PIN  9          BusTrnotLoc1        COMB ; INPUT from Main Process Machine
PIN  10         GND
PIN  20         VCC
PIN  19         BusToTr             COMB ; To both the GLiTCH Clock Generator
                                         ; and the Man Process Machine
PIN  18         EqualOut            REG  ; OUTPUT to Host
PIN  17         FinishOut           REG  ; OUTPUT to Host
PIN  16         SC_ReqOut           REG  ; OUTPUT to Host
PIN  15         SsOut               REG  ; OUTPUT to Host
PIN  14..13     BRout[1..0]         REG  ; OUTPUT to Host and buried reg.
PIN  11         notOEn              COMB ; Output enable controll (allways
                                         ; enable!
;       Declaration segment
        ;Reset condition -- use to switch between active low or high
STRING ResetCond 'Reset'
```

```
        ;Bus Request stuff
STRING BusReqOut 'BRout[1]'
        ;state machine codes for BRout[1..0] in this order
STRING Waiting '#b00'
STRING Waiting_sim '/BRout[1] /BRout[0]'
STRING Standby '#b01'
STRING Standby_sim '/BRout[1]  BRout[0]'
STRING Active  '#b11'
STRING Active_sim  ' BRout[1]  BRout[0]'

GROUP OutRegs EqualOut FinishOut SC_ReqOut SsOut BRout[1..0]
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
                ;Output buffers control
;OutRegs.TRST = notOEn   not programable for registered outputs
BusToTr.TRST = VCC

        ;Qualifier for Main Process Machine and the Clock Generator stuff.
        ;Equal, Finish, and Ss are time critical if we want the system to be
        ;stopped after the current instruction. They are all sync with the
        ;clock. SC_Req is not so time critical, but it is also sync. BusReq
        ;it'snt either sync and time critical. So it is better to synchronize
        ;it first and use the BusReqOut (= to BRout[1] signal). BusTrnotLoc1
        ;is also used to keep the qualifier set until the end of a Switch Bus
        ;Operation, even if the event generator is reset or in High Imp. state
        ;(as is te case of Micromem. and Seq. signals). BusToTr signal will be
        ;unstable during the begin and the end of a SB op., but never before
        ;the active clock edge. Anyway, the time analysis must bet carried...
BusToTr = Equal + Finish + SC_Req + Ss + BusReqOut + BusTrnotLoc1

;-------------------------------------------------------------------------
;Event Request Process (ERP). The same algorithm is used for all events
;(see documentation) except BusReq. Each ERP machine provides a qualifier for
;the MP machine defined in another file.

;Bus Request event -- Main difference: the qualifier is tested twice before it
;sets the BusReqOut flag.

CASE (BRout[1..0])
   BEGIN
      Waiting: BEGIN
                  IF (/ResetCond * /BusTrnotLoc1 * BusReq) THEN
                     BEGIN
                        BRout[1..0] = Standby
                     END
                   ELSE
                      BEGIN
                         BRout[1..0] = Waiting
                      END
               END
      Standby: BEGIN
                  IF (/ResetCond * /BusTrnotLoc1 * BusReq) THEN
                     BEGIN
                        BRout[1..0] = Active
                     END
                   ELSE
                      BEGIN
                         BRout[1..0] = Waiting
                      END
```

```
               END
      Active:  BEGIN
                  IF (ResetCond + ClrFlags) THEN
                     BEGIN
                        BRout[1..0] = Waiting
                     END
                   ELSE
                      BEGIN
                         BRout[1..0] = Active
                      END
               END
      OTHERWISE:
               BEGIN
                  BRout[1..0] = Waiting
               END
   END
;-------------------------------------------------------------------------
;Scalar Mode Request event.
CASE (SC_ReqOut)
   BEGIN
      #b0: BEGIN
              IF (/ResetCond * /BusTrnotLoc1 * SC_Req) THEN
                 BEGIN
                    SC_ReqOut = #b1
                 END
                ELSE
                   BEGIN
                      SC_ReqOut = #b0
                   END
           END
      #b1: BEGIN
              IF (ResetCond + ClrFlags) THEN
                 BEGIN
                    SC_ReqOut = #b0
                 END
                ELSE
                   BEGIN
                      SC_ReqOut = #b1
                   END
           END
   END
;-------------------------------------------------------------------------
;Finish microprogram event.
CASE (FinishOut)
   BEGIN
      #b0: BEGIN
              IF (/ResetCond * /BusTrnotLoc1 * Finish) THEN
                 BEGIN
                    FinishOut = #b1
                 END
                ELSE
                   BEGIN
                      FinishOut = #b0
                   END
           END
      #b1: BEGIN
              IF (Reset + ClrFlags) THEN
                 BEGIN
                    FinishOut = #b0
```

```
                END
              ELSE
                 BEGIN
                    FinishOut = #b1
                 END
          END
   END
;-------------------------------------------------------------------------
;Sequencer Equal flag (break point event).
CASE (EqualOut)
   BEGIN
      #b0: BEGIN
              IF (/ResetCond * /BusTrnotLoc1 * Equal) THEN
                 BEGIN
                    EqualOut = #b1
                 END
               ELSE
                 BEGIN
                    EqualOut = #b0
                 END
          END
      #b1: BEGIN
              IF (Reset + ClrFlags) THEN
                 BEGIN
                    EqualOut = #b0
                 END
               ELSE
                 BEGIN
                    EqualOut = #b1
                 END
          END
   END
;-------------------------------------------------------------------------
;Single Step event.
CASE (SsOut)
   BEGIN
      #b0: BEGIN
              IF (/ResetCond * /BusTrnotLoc1 * Ss) THEN
                 BEGIN
                    SsOut = #b1
                 END
               ELSE
                 BEGIN
                    SsOut = #b0
                 END
          END
      #b1: BEGIN
              IF (Reset + ClrFlags) THEN
                 BEGIN
                    SsOut = #b0
                 END
               ELSE
                 BEGIN
                    SsOut = #b1
                 END
          END
   END
```

4.4 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
;PRELOAD acts on the Flip-Flops, and not on the outputs. In 16V8 all
;outputs has an output inverter, wich means to produce a low output it's
;necessary to preload the register with a 1.

TRACE_ON Clk Equal Finish Ss SC_Req BusReq ResetCond BusTrnotLoc1 ClrFlags
         BusToTr EqualOut FinishOut SC_ReqOut BRout[1..0] SsOut
         ;Test Reset condition
SETF /Clk ResetCond /Equal /Finish /Ss /SC_Req /BusReq /BusTrnotLoc1
     /ClrFlags /notOEn
PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF  Equal  Finish  Ss  SC_Req  BusReq /BusTrnotLoc1 /ClrFlags
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
PRELOAD /EqualOut /FinishOut /SC_ReqOut /SsOut /BRout[1] /BRout[0]
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
PRELOAD  BRout[1] /BROut[0]
CLOCKF Clk
CHECK Waiting_sim
PRELOAD /BRout[1]  BROut[0]
CLOCKF Clk
CHECK Waiting_sim

SETF /ResetCond
SETF /Equal /Finish /Ss /SC_Req /BusReq /BusTrnotLoc1 /ClrFlags

; -- Test each of the ERP machines
; -- Bus request event
CHECK /BusToTr
SETF /BusReq
;PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
SETF ResetCond
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF /ResetCond
SETF BusTrnotLoc1 /ClrFlags     ;MP starts in Phase1, BusTrnotLoc1 set...
CHECK BusToTr
CLOCKF Clk
CHECK Waiting_sim
SETF BusReq
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK Waiting_sim               ;...ERP machine should not change state
SETF /BusTrnotLoc1              ;Now it should run...
CLOCKF Clk
CHECK Standby_sim
CHECK /BusToTr
SETF /BusReq                    ;if BusReq goes low...
CLOCKF Clk
```

```
CHECK Waiting_sim               ;it must go to first state again
CHECK /BusToTr
SETF BusReq
CLOCKF Clk
CLOCKF Clk
CHECK Active_sim
CHECK  BusToTr
SETF /BusReq
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK  BusToTr
CHECK Active_sim                ;and wait for ClrFlags, even if source signal
SETF ClrFlags                   ;is cleared.
CLOCKF Clk
CHECK Waiting_sim
; -- end of Bus request event.

; -- Scalar mode request event
SETF /SC_Req
CHECK /BusToTr
;PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
SETF ResetCond
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF /ResetCond
SETF BusTrnotLoc1 /ClrFlags     ;MP starts in Phase1, BusTrnotLoc1 set...
CLOCKF Clk
CHECK /SC_ReqOut
SETF SC_Req
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK  /SC_ReqOut               ;...ERP machine should not change state
SETF /BusTrnotLoc1
CHECK  BusToTr
CLOCKF Clk
CHECK SC_ReqOut                 ;Now it should run...
CLOCKF Clk
CLOCKF Clk
CHECK SC_ReqOut
SETF /SC_Req
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK SC_ReqOut                 ;and wait for ClrFlags, even if source signal
SETF ClrFlags                   ;is cleared.
CLOCKF Clk
CHECK /SC_ReqOut
; -- end of Scalar mode request event.

; -- Finish microprogram event
SETF /Finish
CHECK /BusToTr
;PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
SETF ResetCond
```

```
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF /ResetCond
SETF BusTrnotLoc1 /ClrFlags     ;MP starts in Phase1, BusTrnotLoc1 set...
CLOCKF Clk
CHECK /FinishOut
SETF Finish
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK  /FinishOut               ;...ERP machine should not change state
SETF /BusTrnotLoc1
CHECK  BusToTr
CLOCKF Clk
CHECK FinishOut                 ;Now it should run...
CLOCKF Clk
CLOCKF Clk
CHECK FinishOut
SETF /Finish
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK FinishOut                 ;and wait for ClrFlags, even if source signal
SETF ClrFlags                   ;is cleared.
CLOCKF Clk
CHECK /FinishOut
; -- end of Finish microprogram event.

; -- Sequencer Equal flag (break point event)
SETF /Equal
CHECK /BusToTr
;PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
SETF ResetCond
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF /ResetCond
SETF  BusTrnotLoc1 /ClrFlags    ;MP starts in Phase1, BusTrnotLoc1 set...
CLOCKF Clk
CHECK /EqualOut
SETF Equal
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK  /EqualOut               ;...ERP machine should not change state
SETF /BusTrnotLoc1
CHECK  BusToTr
CLOCKF Clk
CHECK EqualOut                 ;Now it should run...
CLOCKF Clk
CLOCKF Clk
CHECK EqualOut
SETF /Equal
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
```

```
CHECK EqualOut                  ;and wait for ClrFlags, even if source signal
SETF  ClrFlags                  ;is cleared.
CLOCKF Clk
CHECK /EqualOut
; -- end of Sequencer Equal flag (break point event).

; -- Single step event
SETF /Ss
CHECK /BusToTr
;PRELOAD  EqualOut  FinishOut  SC_ReqOut  Brout[1..0]  SsOut
SETF ResetCond
CLOCKF Clk
CHECK /EqualOut /FinishOut /SC_ReqOut Waiting_sim /SsOut
SETF /ResetCond
SETF  BusTrnotLoc1 /ClrFlags   ;MP starts in Phase1, BusTrnotLoc1 set...
CLOCKF Clk
CHECK /SsOut
SETF Ss
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK  /SsOut               ;...ERP machine should not change state
SETF /BusTrnotLoc1
CHECK  BusToTr
CLOCKF Clk
CHECK SsOut                 ;Now it should run...
CLOCKF Clk
CLOCKF Clk
CHECK SsOut
SETF /Ss
FOR i := 0 TO 5 DO
   BEGIN
      CLOCKF Clk
   END
CHECK SsOut                 ;and wait for ClrFlags, even if source signal
SETF ClrFlags                   ;is cleared.
CLOCKF Clk
CHECK /SsOut
; -- end of Single step event.

TRACE_OFF
;-----------------------------------------------------------------
```

5. Componentes de suporte ao transputer hospedeiro

5.1 Desmultiplexador do barramento e gerador dos sinais CAS

```
;PALASM Design Description

;--------------------------------- Declaration Segment ------------
TITLE    GLiTCH Controller - Host Transputer subsystem
PATTERN  Bus Interface and DRAM CAS generator
REVISION 1.00
AUTHOR   Henrique Santos
COMPANY  University of Minho
DATE      2/04/92
```

```
CHIP  _businт  PALCE16V8

;---------------------------------- PIN Declarations ---------------
PIN  1          ALE                                     COMB ; CLOCK
PIN  2..5       notMemWrB[0..3]                         COMB ; INPUT
PIN  6          notMemS3                                COMB ; INPUT
PIN  7          notMemWrD0                              COMB ; INPUT
PIN  8..9       AD[24..25]                              COMB ; INPUT
PIN  10         GND
PIN  11         notOEn                                  COMB ; INPUT set to 0
PIN  12         notMemRef                               COMB ; INPUT
PIN  13..14     TrA[25..24]                             REG  ; OUTPUT
PIN  15         RdnotWr                                 REG  ; OUTPUT
PIN  16..19     notCAS[3..0]                            COMB ; OUTPUT
PIN  20         VCC

;-------------------------------------Declaration segment -----------
STRING  LocMemCS        '/TrA[25] *  TrA[24]'   ;Transputer mem Chip Select

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
notCAS[3..0].TRST = VCC
;Registered outputs allways depend on notOEn signal (pin 11) and this
;is not programmable. SO SET notOEn TO 0
;allreg_outputs.TRST = notOEn

;Direct Register outputs
TrA[25] = AD[25]
TrA[24] = AD[24]
RdnotWr = notMemWrD0

;CAS generation.
IF (notMemRef * LocMemCS) THEN
   BEGIN
      CASE (RdnotWr)
         BEGIN
            1:  BEGIN
                   notCAS[3..0] = notMemS3
                END
            0:  BEGIN
                   notCAS[3..0] = notMemWRB[3..0]
                END
         END
   END
  ELSE
     BEGIN
        notCAS[3..0] = VCC
     END
```

5.2 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON ALE notMemWrB[0..3] notMemS3 notMemWrD0 AD[24..25] notMemRef
         TrA[25..24] RdnotWr notCAS[3..0]
                ;Registered outputs allways enable
```

```
                ;ALE This is the 'clock'
SETF /notOEn /ALE /AD[25] /AD[24] /notMemWrD0 /notMemS3 /notMemWrB[0..3]
     /notMemRef
CHECK TrA[25..24] RdnotWr notCAS[3..0]
        ;Registers power-up al logical level 0, but the output inverter will
        ;forca a 1 at output pins. notCAS[3..0] are all 1 because of inputs.
CLOCKF ALE
CHECK /TrA[25] /TrA[24] /RdnotWr
SETF  AD[25]  AD[24]  notMemWrD0
CLOCKF ALE
CHECK  TrA[25]  TrA[24]  RdnotWr
SETF /AD[25]  AD[24]
CLOCKF ALE
CHECK /TrA[25]  TrA[24]  ;This is the Local memory chip select condition...
                         ;notMemRef is 0. notCAS will stay at 0.
        ;CAS generation
SETF  AD[25] /AD[24]            ;Not selected
CLOCKF ALE
CHECK  TrA[25] /TrA[24] notCAS[3..0]
SETF /notMemRef                 ;A refresh cycle
CHECK notCAS[3..0]
SETF  notMemRef                 ;A normal (no refresh) cycle
CHECK notCAS[3..0]
SETF /AD[25] /AD[24]            ;Still not Selected
CLOCKF ALE
CHECK /TrA[25] /TrA[24] notCAS[3..0]
SETF /AD[25]  AD[24]            ;Now it will be selected and notMemRef=1
                                ;ReadnotWr is 1 from last vec., so CAS will
                                ;follow notMemS3 that has initial value 0.
CLOCKF ALE
CHECK /TrA[25]  TrA[24] /notCAS[3..0]
SETF notMemS3
CHECK notCAS[3..0]
SETF /notMemWrD0        ;Simulate a write cycle. notCAS[3..0] must follow
                        ;notMemWrB[3..0], which has initial value 0.
CLOCKF ALE
CHECK /notCAS[3..0]
SETF  notMemWrB[3]  notMemWrB[2]  notMemWrB[1]  notMemWrB[0]
CHECK  notCAS[3]  notCAS[2]  notCAS[1]  notCAS[0]
SETF /notMemWrB[3]  notMemWrB[2]  notMemWrB[1]  notMemWrB[0]
CHECK /notCAS[3]  notCAS[2]  notCAS[1]  notCAS[0]
SETF  notMemWrB[3] /notMemWrB[2]  notMemWrB[1]  notMemWrB[0]
CHECK  notCAS[3] /notCAS[2]  notCAS[1]  notCAS[0]
SETF  notMemWrB[3]  notMemWrB[2] /notMemWrB[1]  notMemWrB[0]
CHECK  notCAS[3]  notCAS[2] /notCAS[1]  notCAS[0]
SETF  notMemWrB[3]  notMemWrB[2]  notMemWrB[1] /notMemWrB[0]
CHECK  notCAS[3]  notCAS[2]  notCAS[1] /notCAS[0]
SETF /notMemWrB[3] /notMemWrB[2] /notMemWrB[1] /notMemWrB[0]
CHECK /notCAS[3] /notCAS[2] /notCAS[1] /notCAS[0]
TRACE_OFF
;---------------------------------------------------------------------
```

5.3 Multiplexador de endereços para DRAM (5 bits)

```
;PALASM Design Description
; NOTE: Two chips are required
;--------------------------------- Declaration Segment ------------
```

```
TITLE    GLiTCH System Controller - Transputer Host sub-system
PATTERN  DRAM 5-bit Multiplexer
REVISION 1.00
AUTHOR   Henrique Santos
COMPANY  University of Minho
DATE      3/04/92

CHIP  _drammux  PALCE16V8

;--------------------------------- PIN Declarations ---------------
PIN  1..5       ARow[0..4]                          COMBINATORIAL ; INPUT
PIN  6..9       ACol[0..3]                          COMBINATORIAL ; INPUT
PIN  10         GND                                             ;
PIN  11..12     ACol[5..4]                          COMBINATORIAL ; INPUT
PIN  13         AMux                                COMBINATORIAL ; INPUT
PIN  14         CTypeSel                            COMBINATORIAL ; INPUT
PIN  15..19     A[4..0]                             COMBINATORIAL ; OUTPUT
PIN  20         VCC                                             ;

;This Mux allows the use of two types of memory - 1M or 256K. The choise is
;based on the state of TypeSel input (1 means 1M while 0 means 256K). To
;implement the complete mux, 2 of this chips are required. There behavier
;must be as follow:
;                                1M
;               LOW CHIP                          HIGH CHIP
;PIN            1       0      <-AMux->          1       0
;A0 Will output A2  or  A12                       A7      A17
;A1             A3      A13                       A8      A18
;A2             A4      A14                       A9      A19
;A3             A5      A15                       A10     A20
;A4             A6      A16                       A11     A21
;                                256k
;               LOW CHIP (out)                    HIGH CHIP (out)
;PIN            1       0      <-AMux->          1       0
;A0             A2      A11                       A7      A16
;A1             A3      A12                       A8      A17
;A2             A4      A13                       A9      A18
;A3             A5      A14                       A10     A19
;A4             A6      A15                       ---     --- (NOT USED)
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
A[4..0].TRST = VCC


IF (AMux) THEN    ;During first half pass low 5 bits
   BEGIN
      A[0..4] = ARow[0..4]
   END
  ELSE            ;During second half...
     BEGIN
        IF (CTypeSel) THEN      ;1M case
           BEGIN
              A[0..4] = ACol[1..5]
           END
          ELSE                  ;256K case
             BEGIN
                A[0..4] = Acol[0..4]
             END
     END
```

5.4 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON CTypeSel ARow[0..4] ACol[0..5] AMux A[0..4]
                ;256K case
SETF AMux /CTypeSel /ARow[0..4] /ACol[0..5] ;Row address first
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;Change one bit a time
SETF  ARow[0] /ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK  A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0]  ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK /A[0]  A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1]  ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1]  A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2]  ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2]  A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2] /ARow[3]  ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3]  A[4]
                        ;changes in ACol should make no difference
SETF /ARow[0..4] ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0]  ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1]  ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2]  ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3]  ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4]  ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;Column address
                        ;Change one bit a time
                        ;Changes in Row address should make no diference
SETF  /AMux ARow[0] /ARow[1] /ARow[2] /ARow[3] /ARow[4] /ACol[0..5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0]  ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1]  ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2]  ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2] /ARow[3]  ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;changes in ACol-diferent for 256K
SETF /ARow[0..4] ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK  A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0]  ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0]  A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1]  ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1]  A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2]  ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2]  A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3]  ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3]  A[4]
```

```
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4]  ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
               ;1M case
               ;Row address first
SETF  CTypeSel AMux /ARow[0..4] /ACol[0..5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;Change one bit a time
SETF  ARow[0] /ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK  A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0]  ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK /A[0]  A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1]  ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1]  A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2]  ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2]  A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2] /ARow[3]  ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3]  A[4]
                        ;changes in ACol should make no difference
SETF /ARow[0..4] ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0]  ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1]  ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2]  ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3]  ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4]  ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;Column address
                        ;Change one bit a time
                        ;Changes in Row address should make no diference
SETF /AMux ARow[0] /ARow[1] /ARow[2] /ARow[3] /ARow[4] /ACol[0..5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0]  ARow[1] /ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1]  ARow[2] /ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2]  ARow[3] /ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF /ARow[0] /ARow[1] /ARow[2] /ARow[3]  ARow[4]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
                        ;changes in ACol-diferent for 1M case
SETF /ARow[0..4] ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0]  ACol[1] /ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK  A[0] /A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1]  ACol[2] /ACol[3] /ACol[4] /ACol[5]
CHECK /A[0]  A[1] /A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2]  ACol[3] /ACol[4] /ACol[5]
CHECK /A[0] /A[1]  A[2] /A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3]  ACol[4] /ACol[5]
CHECK /A[0] /A[1] /A[2]  A[3] /A[4]
SETF   /ACol[0] /ACol[1] /ACol[2] /ACol[3] /ACol[4]  ACol[5]
CHECK /A[0] /A[1] /A[2] /A[3]  A[4]
TRACE_OFF
;--------------------------------------------------------------------
```

5.5 Descodificador de endereços (componente 1)

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    GLITCH SYSTEM CONTROLLER - TRANSPUTER ADDRESS DECODER 1
PATTERN  TRMAINCS
REVISION 2.20
AUTHOR   HENRIQUE SANTOS
COMPANY  UNIVERSITY OF MINHO
DATE      5/06/92

CHIP  TRMAINCS  PAL22V10

;- This version do not generate Chip Select for Scalar Unit. From now on
;the Host does not acess the SU.
;- As a consequence, SL8 is no longer used because the Data Rounting Data Bus
;transceivers can be enable only with the Data Store chip select signal (in
;the last version they should be enable when either the DS or the SU was
;selected).
;- Chip select for local memory is handeled by the 'BUS Interface Unit'
;- A new chip select for Video Interface (LCA chip) is now generated.

;---------------------------------- PIN Declarations ---------------
PIN  2..11      TrA[16..25]          COMB ; INPUT
PIN  12         GND
PIN  13         TrnotMemS3           COMB ; INPUT
PIN  14         BusTrnotLoc2         COMB ; INPUT
PIN  15..17     notMM_CS[0..2]       COMB ; OUTPUT
PIN  18..19     notDS_CS[0..1]       COMB ; OUTPUT
PIN  20         notLRegs_CS          COMB ; OUTPUT
PIN  21         notLCA_CS            COMB ; OUTPUT
PIN  22         notENTrcDS           COMB ; OUTPUT
PIN  23         TrnotMemRef          COMB ; INPUT
;pins 1 (usually the clock) is an input pin. It is available
PIN  24         VCC

;Notas:
;1.Nesta versÆo foi necess rio gerar um novo sinal de Enable para os
;transceivers que ligam o DS ao barramento de dados do transputer.
;Anteriormente aquele sinal de Enable estava ligado ao Out Enable das
;mem¢rias do DS, mas este sinal, para al,m de ficar em Tri-state quando
;o sinal BusTrnotLoc2=0, era apenas igual ao sinal ReadnotWr (negado) do
;Transputer. Isto causava selec‡äes ilegais dos transceivers!!!
;Este sinal passa entÆo a ser igual ao sinal de CS conjugado com
;o BusTrnotLoc2. ESTE SINAL NÇO  SIMULADO, POR SER IGUAL AO CS.

        ; Declaration Segment
STRING MicroB0 'TrA[25] * TrA[24] * TrA[23] * TrA[22] * TrA[21] * /TrA[20] *
                /TrA[19] * /TrA[18]'
STRING MicroB1 'TrA[25] * TrA[24] * TrA[23] * TrA[22] * TrA[21] * /TrA[20] *
                /TrA[19] *  TrA[18]'
STRING MicroB2 'TrA[25] * TrA[24] * TrA[23] * TrA[22] * TrA[21] * /TrA[20] *
                 TrA[19] * /TrA[18]'
STRING DataST 'TrA[25] * TrA[24] * TrA[23] * TrA[22] * TrA[21] * TrA[20] *
               /TrA[19] * /TrA[18] * /TrA[17] * /TrA[16]'
STRING TrRegs '/TrA[25] * /TrA[24] * /TrA[23] * /TrA[22] * /TrA[21] * /TrA[20]
```

```
               * /TrA[19] * /TrA[18] * /TrA[17] * TrA[16]'
STRING LCA    '/TrA[25] * /TrA[24] * /TrA[23] * /TrA[22] * /TrA[21] * /TrA[20]
               * /TrA[19] * /TrA[18] *  TrA[17] * TrA[16]'

;---------------------------------- Boolean Equation Segment ------
EQUATIONS

notMM_CS[0..2].TRST = VCC
notLRegs_CS.TRST = VCC
notLCA_CS.TRST = VCC
notENTrcDS.TRST = VCC
notDS_CS[0..1].TRST = BusTrnotLoc2

/notMM_CS[0] = MicroB0 * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2
/notMM_CS[1] = MicroB1 * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2
/notMM_CS[2] = MicroB2 * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2
/notENTrcDS  = DataST * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2
/notDS_CS[0] = DataST * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2
/notDS_CS[1] = DataST * /TrnotMemS3 * TrnotMemRef * BusTrnotLoc2 ;Is the same
/notLRegs_CS = TrRegs * /TrnotMemS3 * TrnotMemRef               ;signal
/notLCA_CS   = LCA * /TrnotMemS3 * TrnotMemRef
```

5.6 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON TrA[25..16] BusTrnotLoc2 TrnotMemS3 TrnotMemRef
         notMM_CS[0..2] notDS_CS[0..1] notLRegs_CS notLCA_CS
        ;System in 'Local Mode' (Host has no acess)
                ;No decoded Address
SETF /TrA[25..16] /BusTrnotLoc2 TrnotMemS3 /TrnotMemRef
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                  ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                                ;Host mode
CHECK notMM_CS[0..2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                  ;Not in refresh cycle
CHECK notMM_CS[0..2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrA[25]  TrA[24]          ;Host local memory space
CHECK notMM_CS[0..2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrA[25] /TrA[24] TrA[23]  ;GLiTCH local RAM address space
CHECK notMM_CS[0..2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef       ;Disable chip sel, Local
                                                  ;mode and Refresh
CHECK notMM_CS[0..2]  ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS

               ;Micromemory Addresses
                       ;Bank 0
SETF TrA[25..21] /TrA[20..18]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                  ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                  ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                                ;Host mode
CHECK /notMM_CS[0] notMM_CS[1..2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef       ;Disable chip sel, Local
```

```
                                                ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
                        ;Bank 1
SETF TrA[25..21] /TrA[20..19] TrA[18]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                              ;Host mode
CHECK notMM_CS[0] /notMM_CS[1] notMM_CS[2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef     ;Disable chip sel, Local
                                                ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
                        ;Bank 2
SETF TrA[25..21] /TrA[20] TrA[19] /TrA[18]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                              ;Host mode
CHECK notMM_CS[0..1] /notMM_CS[2]  notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef     ;Disable chip sel, Local
                                                ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
                ;Data Store Address
SETF TrA[25..20] /TrA[19..16]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                              ;Host mode
CHECK notMM_CS[0..2] /notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef     ;Disable chip sel, Local
                                                ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
                ;Transputer Registers
SETF /TrA[25..17] TrA[16]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1] /notLRegs_CS  notLCA_CS
SETF  BusTrnotLoc2                              ;Host mode (should make no
CHECK notMM_CS[0..2]  notDS_CS[0..1] /notLRegs_CS  notLCA_CS ;diference)
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef     ;Disable chip sel, Local
                                                ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
                ;Video Interface (LCA chip)
SETF /TrA[25..18] TrA[17..16]
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF /TrnotMemS3                                ;Enable chip select
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
SETF TrnotMemRef                                ;Not in refresh cycle
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS /notLCA_CS
SETF  BusTrnotLoc2                              ;Host mode (should make no
CHECK notMM_CS[0..2]  notDS_CS[0..1]  notLRegs_CS /notLCA_CS ;diference)
```

```
SETF /BusTrnotLoc2  TrnotMemS3 /TrnotMemRef      ;Disable chip sel, Local
                                                 ;mode and Refresh
CHECK notMM_CS[0..2] ^notDS_CS[0..1]  notLRegs_CS  notLCA_CS
TRACE_OFF
;-------------------------------------------------------------------
```

5.7 Gerador de sinais de controlo para a micromemória e o DS

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    GLiTCH System Controller - Host Transputer sub-system
PATTERN  Control and Chip Select Logic
REVISION 1.00
AUTHOR   Henrique Santos
COMPANY  University of Minho
DATE      3/04/92  Last modification:

CHIP  _ctr_csl  PALCE16V8

;---------------------------------- PIN Declarations ---------------
PIN  1..3       MMinnotCS[0..2]                       COMBINATORIAL ; INPUT
PIN  4          RdnotWr                               COMBINATORIAL ; INPUT
PIN  5          MemnotWrB0                            COMBINATORIAL ; INPUT
PIN  6          BusTrnotLoc2                          COMBINATORIAL ; INPUT
PIN  10         GND
PIN  13         DSnotWr                               COMBINATORIAL ; OUTPUT
PIN  14         DSnotOE                               COMBINATORIAL ; OUTPUT
PIN  15         MMnotWr                               COMBINATORIAL ; OUTPUT
PIN  16         MMnotOE                               COMBINATORIAL ; OUTPUT
PIN  17..19     MMnotCS[2..0]                         COMBINATORIAL ; OUTPUT
PIN  20         VCC

;1. Write signals for Data Store and Micromemory are generated only from
;MemnotWrB0. This way is the programmer responsability to assure that
;only 32-bit write operations are executed. If not meaningless data will
;be stored!

;---------------------------------- Declarations ------------------
GROUP DataStore DSnotWr DSnotOE
GROUP MicroMem  MMnotWr MMnotOE MMnotCS[2..0]
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
DataStore.TRST = BusTrnotLoc2   ;This signals are shared with AGL in Data
                                ;Store module
MicroMem.TRST = VCC             ;Micromemory control signals allways enable

        ;Micromemory control
IF (BusTrnotLoc2) THEN BEGIN    ;If Transputer has access
   MMnotWr = MemnotWrB0
   MMnotCS[2..0] = MMinnotCS[2..0]
   MMnotOE = /RdnotWr END
  ELSE BEGIN
     MMnotWr = VCC
     MMnotCS[2..0] = GND
     MMnotOE = GND END
```

```
        ;Data Store control - When in Local mode this lines will be in
        ;high impedance state, so no more logic is required.
DSnotOE = /RdnotWr
DSnotWr = MemnotWrB0
```

5.8 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON MMinnotCS[0..2] RdnotWr MemnotWrB0 BusTrnotLoc2
         DSnotWr DSnotOE MMnotCS[0..2] MMnotWr MMnotOE
                        ;Host has access
        ;Micromemory control
SETF BusTrnotLoc2 MMinnotCS[0..2] RdnotWr MemnotWrB0
CHECK MMnotCS[0..2] /MMnotOE MMnotWr
                ;test one bit at time
SETF /MMinnotCS[0]  MMinnotCS[1]  MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK /MMnotCS[0]  MMnotCS[1]  MMnotCS[2]  MMnotOE  MMnotWr
SETF  MMinnotCS[0] /MMinnotCS[1]  MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK  MMnotCS[0] /MMnotCS[1]  MMnotCS[2]  MMnotOE  MMnotWr
SETF  MMinnotCS[0]  MMinnotCS[1] /MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK  MMnotCS[0]  MMnotCS[1] /MMnotCS[2]  MMnotOE  MMnotWr
SETF /MMinnotCS[0] /MMinnotCS[1] /MMinnotCS[2]  RdnotWr  MemnotWrB0
CHECK /MMnotCS[0] /MMnotCS[1] /MMnotCS[2] /MMnotOE  MMnotWr
SETF /MMinnotCS[0] /MMinnotCS[1] /MMinnotCS[2] /RdnotWr /MemnotWrB0
CHECK /MMnotCS[0] /MMnotCS[1] /MMnotCS[2]  MMnotOE /MMnotWr
SETF  MMinnotCS[0]  MMinnotCS[1]  MMinnotCS[2]  RdnotWr /MemnotWrB0
CHECK  MMnotCS[0]  MMnotCS[1]  MMnotCS[2] /MMnotOE /MMnotWr
        ;Data Store control
SETF /RdnotWr  MemnotWrB0
CHECK  DSnotOE  DSnotWr
SETF  RdnotWr  MemnotWrB0
CHECK /DSnotOE  DSnotWr
SETF /RdnotWr /MemnotWrB0
CHECK  DSnotOE /DSnotWr
SETF  RdnotWr /MemnotWrB0
CHECK /DSnotOE /DSnotWr

                        ;Now in Local mode
        ;Micromemory control
SETF /BusTrnotLoc2 MMinnotCS[0..2] RdnotWr MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
                ;test one bit at time
SETF /MMinnotCS[0]  MMinnotCS[1]  MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
SETF  MMinnotCS[0] /MMinnotCS[1]  MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
SETF  MMinnotCS[0]  MMinnotCS[1] /MMinnotCS[2] /RdnotWr  MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
SETF /MMinnotCS[0] /MMinnotCS[1] /MMinnotCS[2]  RdnotWr  MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
SETF /MMinnotCS[0] /MMinnotCS[1] /MMinnotCS[2] /RdnotWr /MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
SETF  MMinnotCS[0]  MMinnotCS[1]  MMinnotCS[2]  RdnotWr /MemnotWrB0
CHECK /MMnotCS[0..2] /MMnotOE MMnotWr
        ;Data Store control
SETF /RdnotWr  MemnotWrB0
```

```
CHECK ^DSnotOE ^DSnotWr
SETF  RdnotWr  MemnotWrB0
CHECK ^DSnotOE ^DSnotWr
SETF /RdnotWr /MemnotWrB0
CHECK ^DSnotOE ^DSnotWr
SETF  RdnotWr /MemnotWrB0
CHECK ^DSnotOE ^DSnotWr
TRACE_OFF
;-------------------------------------------------------------------
```

5.9 Registos de estado e controlo

```
;PALASM Design Description

;---------------------------------- Declaration Segment ------------
TITLE    GLiTCH System Controller - Host Transputer sub-system
PATTERN  Status and Control Logic
REVISION 1.10
AUTHOR   Henrique Santos
COMPANY  University of Minho
DATE      2/04/93

CHIP  _STAT_CT  PALCE16V8

;---------------------------------- PIN Declarations ---------------
PIN  1          Clk                                     COMB ; CLOCK
PIN  2          MemnotWrB0                              COMB ; INPUT
PIN  3          RdnotWr                                 COMB ; INPUT
PIN  4          notCSel                                 COMB ; INPUT
PIN  5          SDone                                   COMB ; INPUT
PIN  6          A_Full                                  COMB ; INPUT
PIN  7          ClrSDone                                COMB ; INPUT
PIN  8          ClrA_Full                               COMB ; INPUT
PIN  9          Reset                                   COMB ; INPUT
PIN  10         GND
PIN  11         notOE                                   COMB ;Set to 0
PIN  12         BusTrnotLoc1                            COMB ; INPUT
PIN  13         CtrnotWr                                COMB ; OUTPUT
PIN  14         StWr                                    COMB ; OUTPUT
PIN  15         StnotOE                                 COMB ; OUTPUT
PIN  16..17     Aux[0..1]                               REG  ; OUTPUT
PIN  18         A_FullOut                               REG  ; OUTPUT
PIN  19         SDoneOut                                REG  ; OUTPUT
PIN  20         VCC

;1. To clear A_FUll or SDone, the corresponding bit from the control port
;is tested twice, to avoid errors due to different clocks. This requires
;two extra flip-flops.
;2. When BusTrnotLoc1=1 (system under transputer's control) SDone and
;A_Full go to high imp. state and so they must not be tested in state
;machines. Version 1.10 do this.
        ;SDone synchronization state machine declaration
        ;To use with case statements - SDoneOut Aux[1] in this order
STRING  SDWaiting  '#b01'
STRING  SDActive   '#b11'
STRING  SDStandby  '#b10'
        ;To use in simulation statments (CHECK)
```

```
STRING  SDWaitingSim      '/SDoneOut  Aux[1]'
STRING  SDActiveSim       ' SDoneOut  Aux[1]'
STRING  SDStandbySim      ' SDoneOut /Aux[1]'
        ;A_Full synchronization state machine declaration
        ;To use with case statements - A_FullOut Aux[0] in this order
STRING  AFWaiting  '#b01'
STRING  AFActive   '#b11'
STRING  AFStandby  '#b10'
        ;To use in simulation statments (CHECK)
STRING  AFWaitingSim      '/A_FullOut  Aux[0]'
STRING  AFActiveSim       ' A_FullOut  Aux[0]'
STRING  AFStandbySim      ' A_FullOut /Aux[0]'

        ;Reset condition definition (easy to change logic state...)
STRING ResetCond 'Reset'        ;Active High

;---------------------------------- Boolean Equation Segment ------
EQUATIONS
;Enable Outputs: Once there are at least on registered output (fuse SG1=1)
;all outputs TRST must be controlled by the /OE pin or by a PT. This
;selection is made by the fuse SLOn (n=0..7), in each output.
;alloutputs.TRST = /notOE
        ;Status and Control Registers control signals. Assume:
        ;1. CS signal is already qualified with respect to Refresh, what
        ;means it is not generated during refresh.
        ;2. The /OE and Wr (rising edge) for the Status register are the
        ;same signal except in level of activation. The write edge is
        ;used to 'sample' the flag states into the register, and at the
        ;same time the /OE enable the outputs. This is done in 10ns with
        ;a AS374 register.
/StnotOE = /notCSel * RdnotWr
StWr = /notCSel * RdnotWr
/CtrnotWr = /notCSel * /MemnotWrB0

        ;SDone interface
CASE (SDoneOut, Aux[1]) BEGIN
   SDWaiting:  BEGIN ;code #b01
                  IF (/ResetCond * SDone * /BusTrnotLoc1) THEN BEGIN
                     SdoneOut = VCC Aux[1] = VCC END ;goto ..Active
                   ELSE BEGIN
                      SDoneOut = GND Aux[1] = VCC END
               END
   SDActive:   BEGIN ;code #b11
                  IF (ResetCond) THEN BEGIN     ;If reset goto ..Waiting
                     SDoneOut = GND Aux[1] = VCC END
                   ELSE BEGIN
                      IF (ClrSDone) THEN BEGIN
                         Aux[1] = GND SDoneOut = VCC END ;goto ..Standby
                        ELSE BEGIN              ;or stay in ..Active
                           Aux[1] = VCC SDoneOut = VCC END
                   END
               END
    SDStandby: BEGIN ;code #b10
                   IF (ResetCond) THEN BEGIN
                      SDoneOut = GND Aux[1] = VCC END ;goto ..Waiting
                    ELSE BEGIN
                       IF (ClrSDone) THEN BEGIN
                          SDoneOut = GND Aux[1] = VCC END ;goto ..Waiting
                        ELSE BEGIN
```

```
                              Aux[1] = VCC SDoneOut = VCC END ;goto ..Active
                        END
                END
    OTHERWISE: BEGIN ;code #b00
                   SDoneOut = GND Aux[1] = VCC ;goto ..Waiting

                END
END ; /* case */

        ;A_Full interface
CASE (A_FullOut, Aux[0]) BEGIN
   AFWaiting:  BEGIN ;code #b01
                   IF (/ResetCond * A_Full * /BusTrnotLoc1) THEN BEGIN
                      A_FullOut = VCC Aux[0] = VCC END ;goto ..Active
                     ELSE BEGIN
                        A_FullOut = GND Aux[0] = VCC END
                END
   AFActive:   BEGIN ;code #b11
                   IF (ResetCond) THEN BEGIN     ;If reset goto ..Waiting
                      A_FullOut = GND Aux[0] = VCC END
                     ELSE BEGIN
                        IF (ClrA_Full) THEN BEGIN
                           Aux[0] = GND A_FullOut = VCC END ;goto ..Standby
                          ELSE BEGIN             ;or stay in ..Active
                             Aux[0] = VCC A_FullOut = VCC END
                     END
                END
    AFStandby: BEGIN ;code #b10
                    IF (ResetCond) THEN BEGIN
                       A_FullOut = GND Aux[0] = VCC END ;goto ..Waiting
                     ELSE BEGIN
                        IF (ClrA_Full) THEN BEGIN
                           A_FullOut = GND Aux[0] = VCC END ;goto ..Waiting
                         ELSE BEGIN
                            Aux[0] = VCC A_FullOut = VCC END ;goto ..Active
                     END
                END
    OTHERWISE: BEGIN ;code #b00
                   A_FullOut = GND Aux[0] = VCC ;goto ..Waiting
                END
END ; /* case */
```

5.10 Vectores de teste

```
;---------------------------------- Simulation Segment ------------
SIMULATION
TRACE_ON Clk Reset MemnotWrB0 RdnotWr notCSel CtrnotWr StWr StnotOE
         SDone ClrSDone A_Full ClrA_Full Aux[0..1] A_FullOut SDoneOut
         BusTrnotLoc1
;BusTrnotLoc1 action is not tested. It is so elementary...

SETF /Clk /Reset /notOE /BusTrnotLoc1

;Combinatorial outputs
                                        ;Chip select active
SETF /MemnotWrB0 /RdnotWr /notCSel      ;Write
CHECK /CtrnotWr /StWr  StnotOE
```

```
SETF  MemnotWrB0  RdnotWr                ;Read
CHECK  CtrnotWr  StWr /StnotOE
SETF /MemnotWrB0  RdnotWr               ;This never happens!!
CHECK /CtrnotWr  StWr /StnotOE
                                        ;Chip select not active
SETF /MemnotWrB0 /RdnotWr  notCSel      ;Write
CHECK  CtrnotWr /StWr  StnotOE
SETF  MemnotWrB0  RdnotWr                ;Read
CHECK  CtrnotWr /StWr  StnotOE
SETF /MemnotWrB0  RdnotWr               ;This never happens!!
CHECK  CtrnotWr /StWr  StnotOE

;NOTE: Preload operations always reffer to the FF output not to the pin.
;Because PALCE16V8 has low level active outputs on registered MCs, this means
;it is NECESSARY TO PRELOAD THE NEGATION OF THE DESIRED STATE.
        ;Reset Test
SETF Reset
PRELOAD  SDoneOut  Aux[1]  A_FullOut  Aux[0]    ;State 00 (not specified)
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF  SDone  A_Full /ClrSDone /ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF /SDone /A_Full  ClrSDone  ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
PRELOAD  SDoneOut /Aux[1]  A_FullOut /Aux[0]    ;State 01 (..Waiting)
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF  SDone  A_Full /ClrSDone /ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF /SDone /A_Full  ClrSDone  ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
PRELOAD /SDoneOut /Aux[1] /A_FullOut /Aux[0]    ;State 11 (..Active)
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF  SDone  A_Full /ClrSDone /ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF /SDone /A_Full  ClrSDone  ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
PRELOAD /SDoneOut  Aux[1] /A_FullOut  Aux[0]    ;State 10 (..Standby)
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF  SDone  A_Full /ClrSDone /ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
SETF /SDone /A_Full  ClrSDone  ClrA_Full
CLOCKF Clk
CHECK SDWaitingSim AFWaitingSim
                                ;End of Reset - leave in ..Waiting state
        ;Check SDone
SETF /Reset /SDone /ClrSDone
CLOCKF Clk
CHECK SDWaitingSim
FOR i := 1 TO 5 DO BEGIN
```

```
   CLOCKF Clk
END
CHECK SDWaitingSim
SETF  SDone             ;SDone active
CLOCKF Clk
CHECK SDActiveSim
SETF /SDone
FOR i := 1 TO 5 DO BEGIN
   CLOCKF Clk
END
CHECK SDActiveSim
FOR i := 1 TO 5 DO BEGIN
   CLOCKF Clk
END
CHECK SDActiveSim
SETF  ClrSDone          ;Clear SDone...
CLOCKF Clk
CHECK SDStandbySim
SETF /ClrSDone          ;but this was a false pulse
CLOCKF Clk
CHECK SDActiveSim       ;and so it must keep output active
SETF  ClrSDone          ;Now is the real operation
CLOCKF Clk
CLOCKF Clk
CHECK SDWaitingSim

        ;Check A_Full
SETF /A_Full /ClrA_Full
CLOCKF Clk
CHECK AFWaitingSim
FOR i := 1 TO 5 DO BEGIN
   CLOCKF Clk
END
CHECK AFWaitingSim
SETF  A_Full             ;A_Full active
CLOCKF Clk
CHECK AFActiveSim
SETF /A_Full
FOR i := 1 TO 5 DO BEGIN
   CLOCKF Clk
END
CHECK AFActiveSim
FOR i := 1 TO 5 DO BEGIN
   CLOCKF Clk
END
CHECK AFActiveSim
SETF  ClrA_Full          ;Clear A_Full...
CLOCKF Clk
CHECK AFStandbySim
SETF /ClrA_Full          ;but this was a false pulse
CLOCKF Clk
CHECK AFActiveSim       ;and so it must keep output active
SETF  ClrA_Full          ;Now is the real operation
CLOCKF Clk
CLOCKF Clk
CHECK AFWaitingSim

TRACE_OFF
;---------------------------------------------------------------------
```

Apêndice D - Microprogramas de teste para controlador do GLiTCH

Índice

1. Introdução 264
2. Teste à operação do DS e da SU 265
3. Teste à operação do selector de *tags* 267
4. Microprograma no modo absoluto 268
5. Microprograma no modo relativo 269
5. Teste aos registos de *pipeline* do módulo GLiTCH 270

1. Introdução

Os dois primeiros microprogramas foram utilizados, com ligeiras alterações, como vectores de teste na simulação do modelo estrutural em VHDL. Tal como foi referido na análise da implementação do controlador do GLiTCH, atendendo à complexidade do modelo estrutural, às restrições impostas à sua funcionalidade e ainda às dificuldades inerentes ao próprio processo de simulação, não foi possível obter resultados conclusivos. Paradoxalmente, a execução destes microprogramas no protótipo permitiram testar de uma forma aceitável o interface entre as diferentes unidades funcionais. Desta forma, tendo em conta os testes individuais aos componentes produzidos em lógica programável e o grau de confiança dos componentes *standard*, globalmente foi possível atingir um grau de confiança elevado.

Os microprogramas definidos nos modos relativo e absoluto, destinaram-se a verificar a operação conjunta do *assembler*, do *loader* e do *debugger*, sobretudo na sua capacidade de operar com o modo relativo de construção de microprogramas.

O último microprograma foi utilizado apenas para gerar um ficheiro de código onde, manualmente, se podessem incluir instruções destinadas ao GLiTCH e respectivos componentes de suporte. Do ponto de vista do controlador do GLiTCH, o objectivo é apenas poder testar a operação dos registos de *pipeline* que interligam o controlador com o módulo GLiTCH.

2. Teste à operação do DS e da SU

```
;=====================================================================
;4 May 1993
; Sample program to test the Scalar Unit (and Data Store) operation.
;Due to the AGL "address shift" problem some care must be taken when using
;the LDA instruction (this use the misplaced bus). When possible it is
;better to use the INC and DEC AGL instructions.

      Name    "Test SU/DS 1.00"
      CORG    0                       ;Code Origin 0h
      START   0                       ;Starting address inside the module

Begin nop     lda pat0
      nop     rd      ldr             ;Read pat0 and load it in SU
      nop                             ;with mask. Result 1030 (data part)
      nop     wr_inc  osr             ;Check it in DS add 0001. It also
                                      ;tests the INC with WR in one cycle.
      ;Shift right SU until 0. Trace execution writing intermediate values
      ;to DS using sucessive addresses.

cycle1
      nop             lsr
      nop     wr_inc  osr
      brnc cycle1 (2)                 ;Jump if flag 2 (Zero) not set.
                            ;After the cycle check:
                            ;DS add         value (data part)
                            ; 0002          0818
                            ; 0003          040C
                            ; 0004          0206
                            ; 0005          0103
                            ; 0006          0081
                            ; 0007          0040
                            ; 0008          0020
                            ; 0009          0010
                            ; 000A          0008
                            ; 000B          0004
                            ; 000C          0002
                            ; 000D          0001
                            ; 000E          0000
      ;Decrement AGL to get 0001 value into SU -- MASK VALUE MUST BE FFFF
      ;which is the default value after memory test -- and shift it left
      ;until SD15 flaf (6) goes high.
      nop     rd_dec  ldr
      nop     inc                     ;AGL points to 000E address
cycle2
      nop             lsl
      nop     wr_inc  osr             ;Doing trace... from DS address 000E
      brnc cycle2 (6)                 ;Jump if flag 6 (SD15) not set.
                                      ;DS addresses 000F to 001D must
                                      ;have values 0002 to 8000 (data part)
                                      ;shifting left the one from one add
                                      ;to the next.
      ;Test the Arithmetic Shift Right instruction, with the loop SEQ
      ;instruction. Current SU value will be ASR 7 times.
      for 7                           ;Initialize FOR cycle (cycle3 goes
                                      ;to the TOS) and load counter with 7
cycle3
```

```
      nop             asr
      nop     wr_inc  osr             ;Trace with DS (from 001E to 0024)
      djmp_s                          ;Dec counter. Goes to TOS if C != 0
                                      ;When C = 0 go to next instruction
                                      ;and POP the stack.
      nop                             ;
                                      ;
      ;Test other 3 flags shifting right the SU until each one is set.
      ;Once more trace with DS.
cycle4
      nop             lsr
      brnc cycle4 (5)                 ;Jump if flag 5 (SD2) not set.
      nop     wr_inc  osr             ;Final value 03FC in DS address 0025
cycle5
      nop             lsr
      brnc cycle5 (4)                 ;Jump if flag 4 (SD1) not set.
      nop     wr_inc  osr             ;Final value 01FE in DS address 0026
cycle6
      nop             lsr
      brnc cycle6 (3)                 ;Jump if flag 3 (SD0) not set.
      nop     wr_inc  osr             ;Final value 00FF in DS address 0027

      nop fin         ;End of Program.

      DORG    0
pat0  dw      f0f01234        ;With MASK, SU must load 1030
sav0  ds      1               ;Check here SU value!
savn  ds      80              ;Reserve 128 locations to store intermediate
```

3. Teste à operação do selector de *tags*

```
;=====================================================================
; Sample program to test Tag Unit operation. To accomplish the test external
;signals TagUp (P2 rowA 32) and TagDown (P2 rowC 31) must be set and compared
;with TagBellow (P2 rowA 31) and TagAbove (P2 rowC 32) respectibally when
;indicated.

      Name    "Test Tag Unit 1.00"
      CORG    0                       ;Code Origin 0h
      START   0                       ;Starting address inside the module

Begin nop rst         ;Set TagBellow and TagAbove to 0
      nop rst         ;for two cycles...
      nop hld         ;1 more cycle through hld instruction
      nop set         ;Set TagBellow and TagAbove to 1
      nop hld         ;and hold set instruction for two cycles
      nop hld
      nop cls         ;Close chain TagUp-->TagBellow and TagDown-->TagAbove
      nop hld         ;hold it...
      nop hld
      nop hld
      nop fin         ;End of Program.
```

4. Microprograma no modo absoluto

```
;=====================================================================
; Absolute module example
      Name "Test 1"           ; Module name
      CORG  20                ; Code origin = 20h
      DORG  10                ; Data origin = 10h
      START 0                 ; Starting address

Start NOP LDA One             ; DS Load Address
      NOP RD                  ; DS Read
      NOP LDR                 ; SU Load
      NOP LDA Res             ; DS Load Address
      NOP WR                  ; DS Write
      NOP OSR                 ; SU Output
      JMP Lab1                ; Jump to Lab1
;-----------------
Lab1  NOP        Fin          ; NOP and Finish flag
;-----------------

; Data definition
One   dw    ffff0001  ; Definition of word 0xffff0001
Res   ds    1         ; Reserve space for one word
```

5. Microprograma no modo relativo

```
;=====================================================================
; Relative module example
      Name "Test relative"    ; Module name
Code3 CEQU  3                 ; Definition of absolute locations
Data100
      DEQU  100
      START 0                 ; Starting address

Start NOP LDA One             ; DS Load Address
      NOP RD                  ; DS Read
      NOP LDR                 ; SU Load
      BRCC Lab1 (0)           ; Conditional jump to Lab1
      NOP LDA Res             ; DS Load Address
      NOP WR                  ; DS Write
      NOP OSR                 ; SU Output
;-----------------
Lab1  NOP
      NOP        Fin          ; NOP and Finish flag
Self  JMP Self
;-----------------
      NOP LDA Data100         ; Absolute addresses
      CALL Code3

;-----------------
; Data definition
One   dw    ffff0001  ; Definition of word 0xffff0001
Res   ds    1         ; Reserve space for one word
```

5. Teste aos registos de *pipeline* do módulo GLiTCH

```
;=====================================================================
;Sequence of NOP instructions to create a template where it is possible to
;manually insert instructions for GLiTCH module's components - the GLiTCH
;assembler does not generate code for those components. To accomplish the
;test it is necessary to trace the pipeline register outputs.

NAME  "Test GLiTCH Board pipe register v1.00"
      CORG    0
      START   0
begin cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont
      cont    fin
```